Model Analysis ToolKit (MATK) Documentation

Release 0

Dylan R. Harp

November 09, 2015

CONTENTS

Homepage: http://matk.lanl.gov

MATK facilitates model analysis within the Python computational environment. MATK expects a *model* defined as a Python function that accepts a dictionary of parameter values as the first argument and returns model results as a dictionary, array, integer, or float. Many model analyses are provided by MATK. These model analyses can be easily modified and/or extended within the Python scripting languange. New model analyses can easily be hooked up to a MATK model as well.

CHAPTER

ONE

OBTAINING MATK

1.1 Downloading MATK

MATK can be obtained by:

1. Using git:

```
git clone https://github.com/dharp/matk
```

2. Clicking here
3. Going to https://github.com/dharp/matk and clicking on the **Download ZIP** button on the right

1.2 Installing MATK

To install MATK, enter the main directory in a terminal and

```
python setup.py install
```

Depending on your system setup and privileges, you may want to do this as root (*nix and mac systems):

```
sudo python setup.py install
```

or as user:

```
python setup.py install --user
```

If all these fail, you can set your PYTHONPATH to point to the MATK *src* directory

1. bash:

```
export PYTHONPATH=/path/to/matk/src
```

2. tcsh:

```
setenv PYTHONPATH /path/to/matk/src
```

3. Windows, I have no clue!

1.3 Testing installation

To test that the MATK module is accessible, open a python/ipython terminal and

```
import matk
```

If the MATK module is accessible, this will load without an error.

For more in depth analysis of MATK functionality on your system, the test suite can be run by entering the MATK *tests* directory in a terminal and:

```
python -W ignore matk_unittests.py -v
```

Test results will be printed to the terminal.

CHAPTER

TWO

GETTING STARTED

The easiest way to get started with MATK is to open an ipython/python terminal and copy and paste the example code below into your terminal. After that, the *Examples* section can be explored for additional ideas of how MATK can facilitate your model analysis.

2.1 Load MATK module

Start by importing the MATK module:

```
import matk
```

We'll use numpy and some scipy functions also, so load those modules:

```
import numpy
from scipy import arange, randn, exp
```

2.2 Define Model

To perform a model analysis, MATK needs a model in the form of a python function that accepts a dictionary of parameter values keyed by parameter names as the first argument and returns an integer, float, array, or dictionary of model results keyed by result names. For demonstration purposes, we'll define a simple python function that computes a summation of exponential function and returns the results as an array:

```
def dbexpl(p):
    t=arange(0,100,20.)
    y = (p['par1']*exp(-p['par2']*t) + p['par3']*exp(-p['par4']*t))
    return y
```

Of course the python function can be much more complicated, including advanced scipy (http://www.scipy.org) functions or calls to external programs (e.g. *External Simulator (FEHM Groundwater Flow Simulator)*).

To test that the function does what you expect it to do, you can create a parameter dictionary and pass it to the function:

```
pars = {'par1':0.5,'par2':0.1,'par3':0.5,'par4':0.1}
print dbexpl(pars)
```

```
[  1.00000000e+00   1.35335283e-01   1.83156389e-02   2.47875218e-03
   3.35462628e-04]
```

This is only to test the function, MATK will be generating the parameter dictionaries during the model analysis.

If you haven't added observations to your MATK object, the first time the MATK *model* is called, observations are automatically created and given default names of *obs1*, *obs2*, ..., *obsN*, where *N* is the number of observations. If you have added observations to your MATK object and your *model* returns an array, the array ordering must match the order in which you added observations. If your *model* returns a dictionary, the order is irrelevant. However, if your *model* returns a dictionary and you have defined observations in your MATK object, your *model* must return a dictionary with keys that match all the defined observations. An example of *dbexpl* that returns a dictionary is:

```
def dbexpl(p):
    t=arange(0,100,20.)
    y = (p['par1']*exp(-p['par2']*t) + p['par3']*exp(-p['par4']*t))
    ydict = dict([('obs'+str(i+1), v)  for i,v in enumerate(y)])
    return ydict
```

```
print dbexpl(pars)
```

```
{'obs4': 0.0024787521766663585, 'obs5': 0.00033546262790251185, 'obs2': 0.1353352832366127, 'obs3': 0
```

Note that the dictionary is out of order. As mentioned above, this is irrelevant since the keys indicate to which observation the values are associated.

To maintain the order of the returned dictionary, you can return an *OrderedDict* from the *collections* package (https://docs.python.org/2/library/collections.html) included in MATK:

```
from matk.ordereddict import OrderedDict

def dbexpl(p):
    t=arange(0,100,20.)
    y = (p['par1']*exp(-p['par2']*t) + p['par3']*exp(-p['par4']*t))
    ydict = OrderedDict([('obs'+str(i+1), v)  for i,v in enumerate(y)])
    return ydict
```

```
print dbexpl(pars)
```

```
OrderedDict([('obs1', 1.0), ('obs2', 0.1353352832366127), ('obs3', 0.018315638888734179), ('obs4', 0.
```

As mentioned, while dictionary ordering may be desirable, it is not required by MATK.

2.3 Create MATK Object

Create an instance of the MATK class (*matk*) specifying the function created above as the *model* using a keyword argument:

```
p = matk.matk(model=dbexpl)
```

2.4 Add Parameters

Add parameters to the model analysis matching those in the MATK model using *add_par*:

```
p.add_par('par1',min=0,max=1,value=0.5)
p.add_par('par2',min=0,max=0.2,value=0.1)
p.add_par('par3',min=0,max=1,value=0.5)
p.add_par('par4',min=0,max=0.2,value=0.1)
```

Check current parameter values:

```
print p.parvalues
```

```
[0.5, 0.1, 0.5, 0.1]
```

and parameter names:

```
print p.parnames
```

```
['par1', 'par2', 'par3', 'par4']
```

and other useful information:

```
print p.parmins
```

```
[0, 0, 0, 0]
```

```
print p.parmaxs
```

```
[1, 0.2, 1, 0.2]
```

You can also access parameters using the MATK *pars* dictionary:

```
print p.pars
```

```
OrderedDict([('par1', <Parameter 'par1', 0.5, bounds=[0:1]>), ('par2', <Parameter 'par2', 0.1, bounds
```

Individual parameters can be accessed using the *pars* dictionary as:

```
print p.pars['par1']
```

```
<Parameter 'par1', 0.5, bounds=[0:1]>
```

```
print p.pars['par1'].value
```

```
0.5
```

```
print p.pars['par1'].min
```

```
0
```

```
print p.pars['par1'].max
```

```
1
```

2.5 Add Observations

Observations are values that you want to compare model results to. These may be measurements that have been collected from the system you are modeling. Let's assume we have the following measurements for our system:

```
observations = [ 1., 0.14, 0.021, 2.4e-3, 3.4e-4]
```

We'll add these to the model analysis with generic names using *add_obs*:

```
for i,o in enumerate(observations): p.add_obs( 'obs'+str(i+1), value=o)
```

In cases where there are no observations (measurements) for comparison, the *value* keyword argument can be omitted. Check observation values and names:

```
print p.obsvalues
```

```
[1.0, 0.14, 0.021, 0.0024, 0.00034]
```

```
print p.obsnames
```

```
['obs1', 'obs2', 'obs3', 'obs4', 'obs5']
```

Similar to parameters, observations can be accessed using the *obs* dictionary:

```
print p.obs
```

```
OrderedDict([('obs1', <Observation 'obs1', observed=1.0, weight=1.0>), ('obs2', <Observation 'obs2',
```

```
print p.obs['obs1']
```

```
<Observation 'obs1', observed=1.0, weight=1.0>
```

```
print p.obs['obs1'].value
```

```
1.0
```

```
print p.obs['obs1'].weight
```

```
1.0
```

2.6 Run Forward Model

A single forward run of the model using the current parameter values can be performed with the *forward* method as:

```
sims = p.forward()
print sims
```

```
OrderedDict([('obs1', 1.0), ('obs2', 0.1353352832366127), ('obs3', 0.018315638888734179), ('obs4', 0.
```

Now if we look at the *obs* dictionary, we see that it includes simulated values:

```
print p.obs
```

```
OrderedDict([('obs1', <Observation 'obs1', observed=1.0, simulated=1.0, weight=1.0>), ('obs2', <Obser
```

The simulated values can be accessed directly also:

```
print p.simvalues
```

```
[1.0, 0.1353352832366127, 0.018315638888734179, 0.0024787521766663585, 0.00033546262790251185]
```

The sum-of-squares can be calculated using the *ssr* method:

```
print p.ssr
```

```
2.89715995514e-05
```

You have now completed the most basic model analysis using MATK, the forward model run. The next step is to take a look at the *Examples* section for details on more useful model analyses involving many forward model runs.

CHAPTER

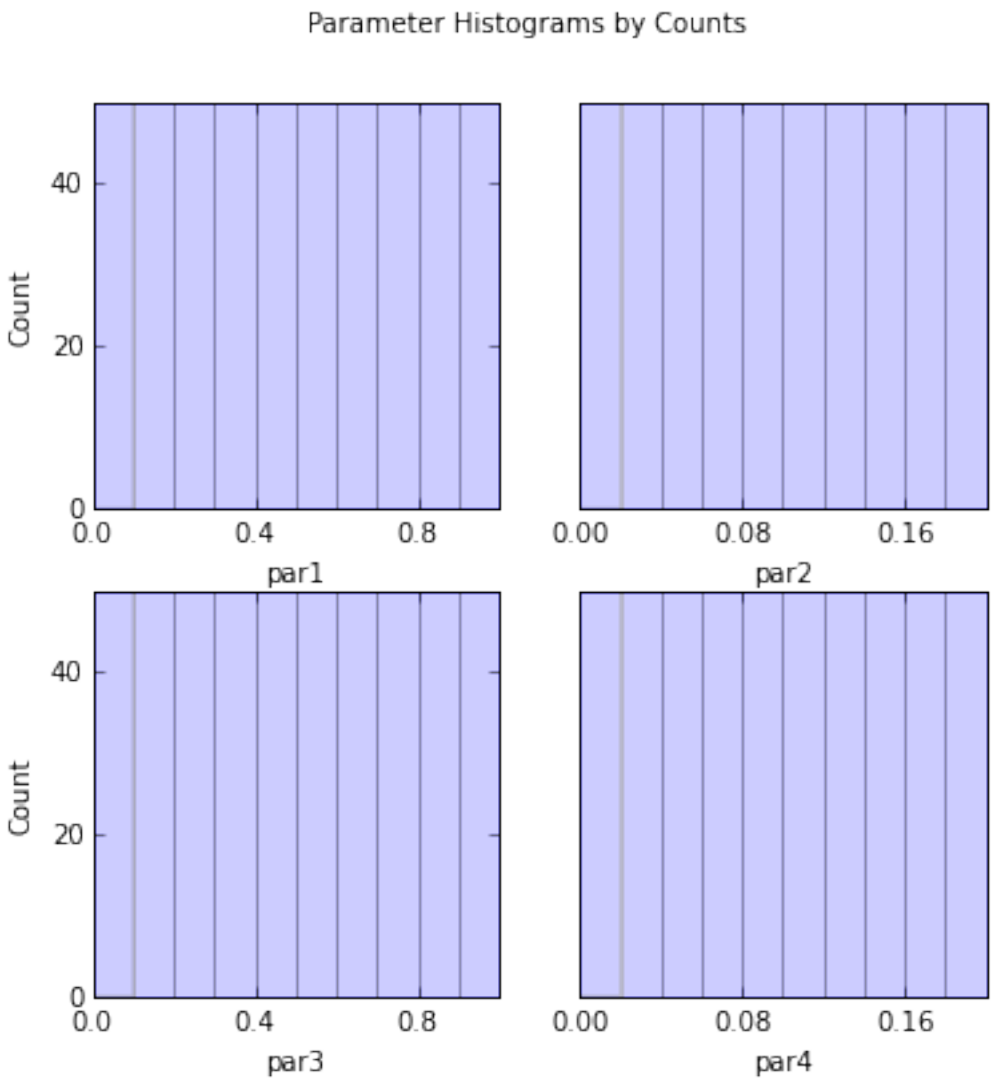
THREE

EXAMPLES

3.1 Sampling

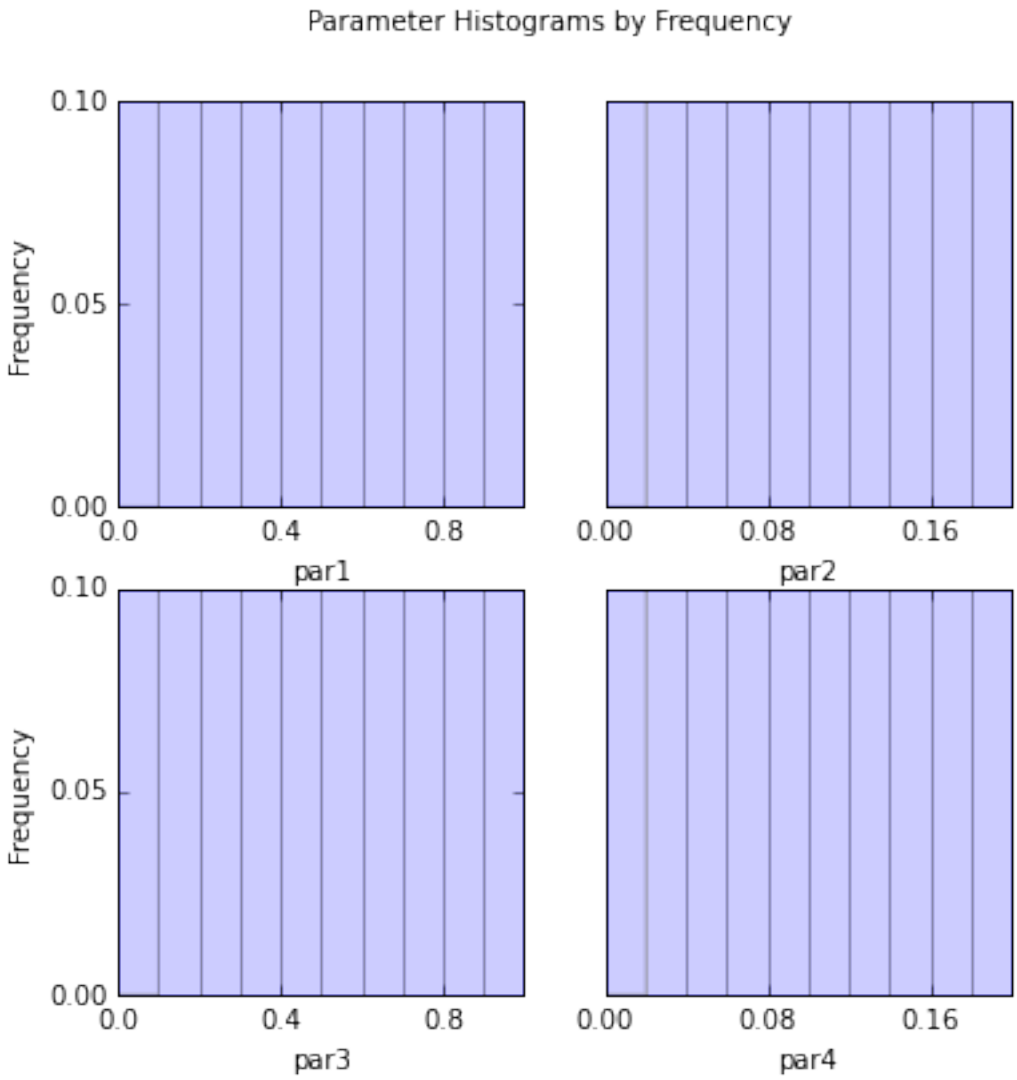
This example demonstrates a Latin Hypercube Sampling of a 4 parameter 5 response model using the *lhs* function. The models of the parameter study are run using the *run* function. The generation of diagnostic plots is demonstrated using *hist*, *panels*, and *corr*.

```
%matplotlib inline
import sys,os
try:
    import matk
except:
    try:
        sys.path.append(os.path.join('..','src'))
        import matk
    except ImportError as err:
        print 'Unable to load MATK module: '+str(err)
import numpy
from scipy import arange, randn, exp
from multiprocessing import freeze_support

# Model function
def dbexpl(p):
    t=arange(0,100,20.)
    y =  (p['par1']*exp(-p['par2']*t) + p['par3']*exp(-p['par4']*t))
    #nm =  ['o1','o2','o3','o4','o5']
    #return dict(zip(nm,y))
    return y

# Setup MATK model with parameters
p = matk.matk(model=dbexpl)
p.add_par('par1',min=0,max=1)
p.add_par('par2',min=0,max=0.2)
p.add_par('par3',min=0,max=1)
p.add_par('par4',min=0,max=0.2)

# Create LHS sample
s = p.lhs('lhs', siz=500, seed=1000)

# Look at sample parameter histograms, correlations, and panels
out = s.samples.hist(ncols=2,title='Parameter Histograms by Counts')
```




```
par1:
Count:            50       50       50       50       50       50       50       50       50       50
 Bins:         0      0.1      0.2      0.3      0.4      0.5      0.6      0.7      0.8      0.9

par2:
Count:            50       50       50       50       50       50       50       50       50       50
 Bins:         0     0.02     0.04     0.06     0.08      0.1     0.12     0.14     0.16     0.18

par3:
Count:            50       50       50       50       50       50       50       50       50       50
 Bins:         0      0.1      0.2      0.3      0.4      0.5      0.6      0.7      0.8      0.9

par4:
Count:            50       50       50       50       50       50       50       50       50       50
 Bins:         0     0.02     0.04     0.06     0.08      0.1     0.12     0.14     0.16     0.18
```

```
out = s.samples.hist(ncols=2,title='Parameter Histograms by Frequency',frequency=True)
```

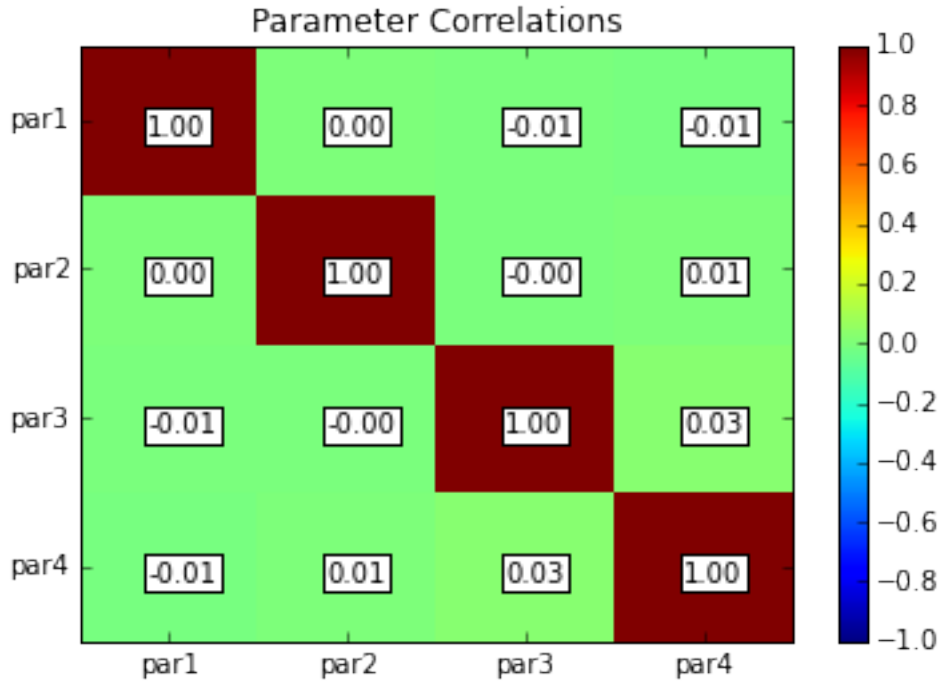



```
par1:
 Freq:          0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10
 Bins:         0       0.1       0.2       0.3       0.4       0.5       0.6       0.7       0.8       0.9

par2:
 Freq:          0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10
 Bins:         0      0.02      0.04      0.06      0.08       0.1      0.12      0.14      0.16      0.18

par3:
 Freq:          0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10
 Bins:         0       0.1       0.2       0.3       0.4       0.5       0.6       0.7       0.8       0.9

par4:
 Freq:          0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10      0.10
 Bins:         0      0.02      0.04      0.06      0.08       0.1      0.12      0.14      0.16      0.18
```

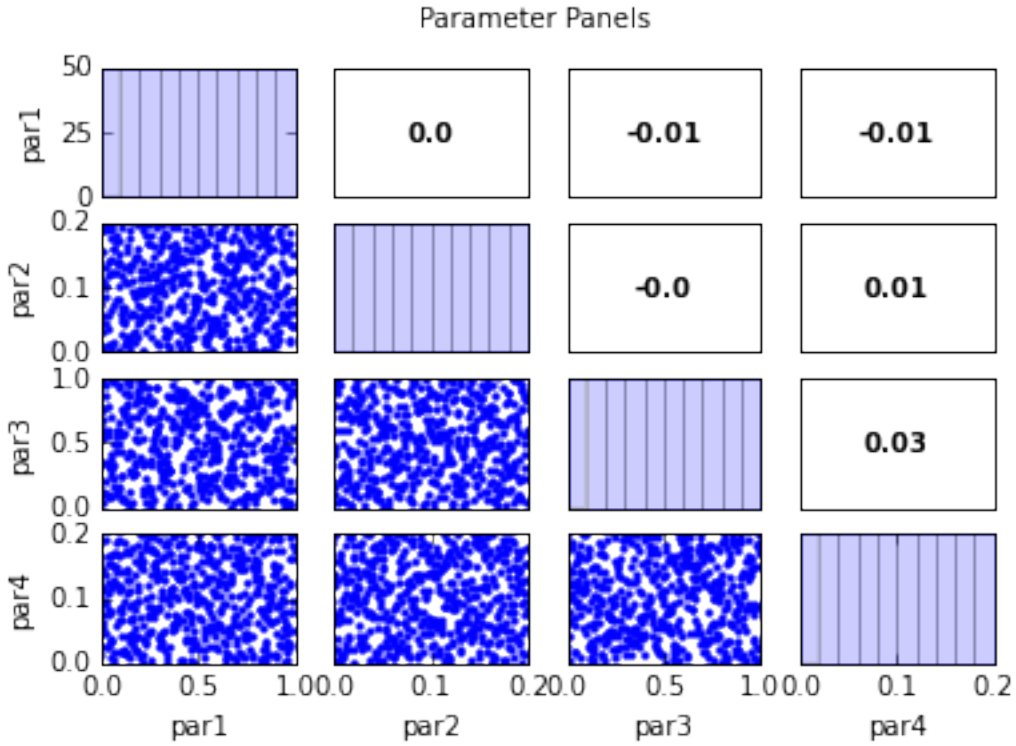
```
parcor = s.samples.corr(plot=True, title='Parameter Correlations')
```

```
             par1      par2      par3      par4
par1         1.00      0.00     -0.01     -0.01
```

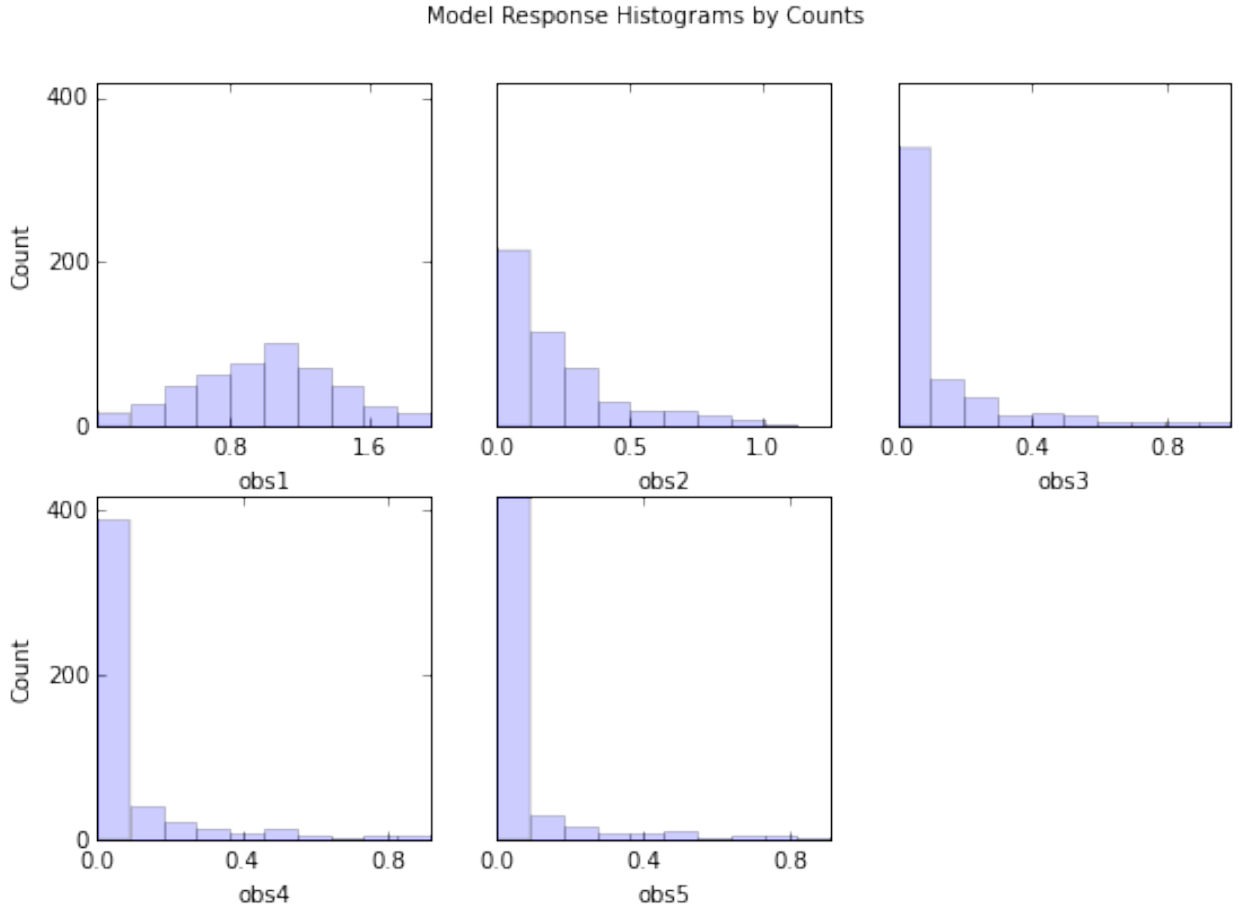
```
par2          0.00      1.00     -0.00      0.01
par3         -0.01     -0.00      1.00      0.03
par4         -0.01      0.01      0.03      1.00
```




```
out = s.samples.panels(title='Parameter Panels')
```

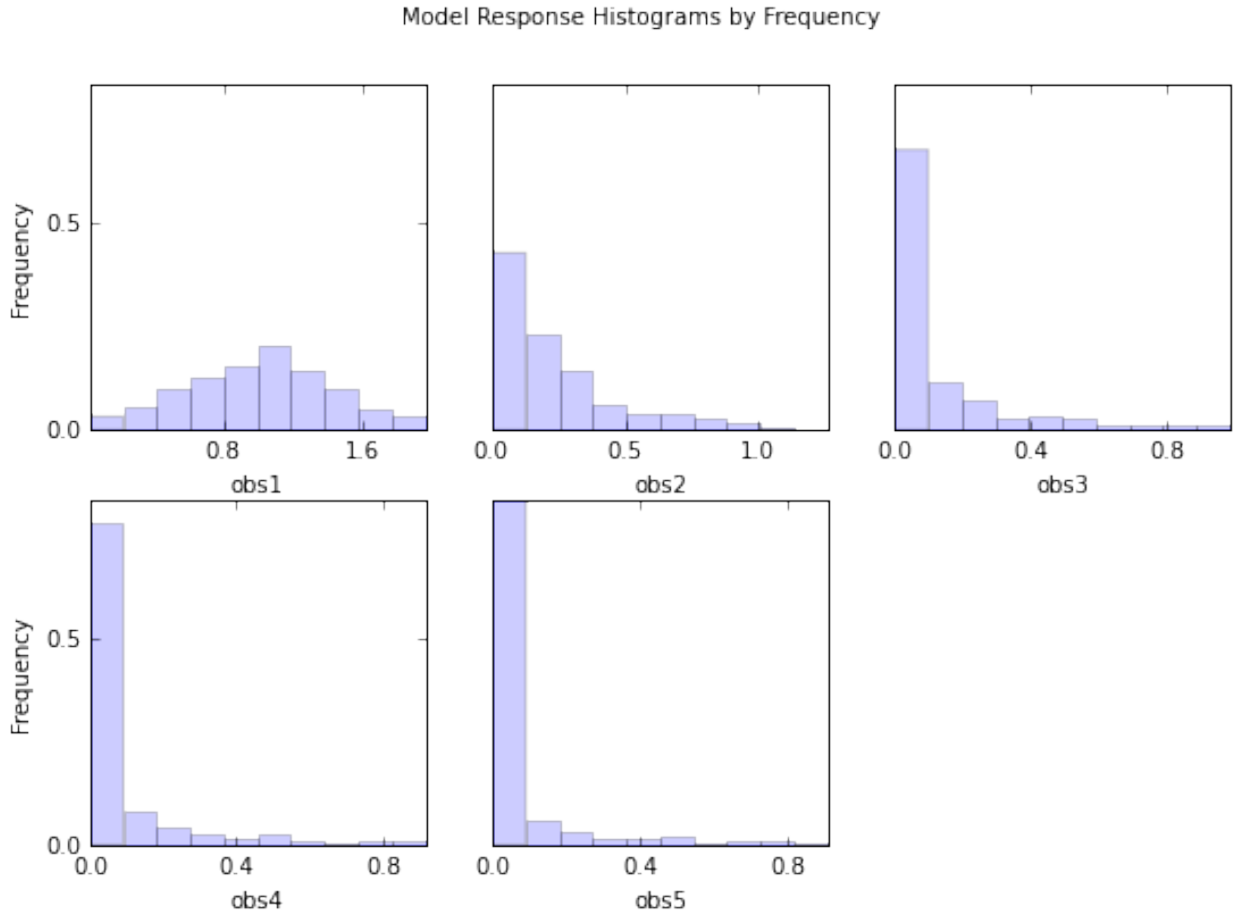


```
# Run model with parameter samples
s.run( cpus=2, outfile='results.dat', logfile='log.dat',verbose=False)

# Look at response histograms, correlations, and panels
out = s.responses.hist(ncols=3,title='Model Response Histograms by Counts')
```




```
obs1:
Count:          16       28       50       63       78      102       72       50       25       16
 Bins:    0.033     0.23     0.42     0.61      0.8     0.99      1.2      1.4      1.6      1.8

obs2:
Count:         214      115       71       32       20       21       14        9        3        1
 Bins:   0.0036     0.13     0.25     0.38     0.51     0.63     0.76     0.88        1      1.1

obs3:
Count:         339       59       37       13       16       14        6        6        5        5
 Bins:  8.5e-05    0.099      0.2      0.3     0.39     0.49     0.59     0.69     0.79     0.89

obs4:
Count:         388       40       22       13        7       12        5        3        4        6
 Bins:  1.8e-06    0.092     0.18     0.27     0.37     0.46     0.55     0.64     0.73     0.82

obs5:
Count:         418       29       15        8        7       10        1        5        4        3
 Bins:  4.2e-08    0.091     0.18     0.27     0.36     0.46     0.55     0.64     0.73     0.82
```

```
out = s.responses.hist(ncols=3,title='Model Response Histograms by Frequency',frequency=True)
```

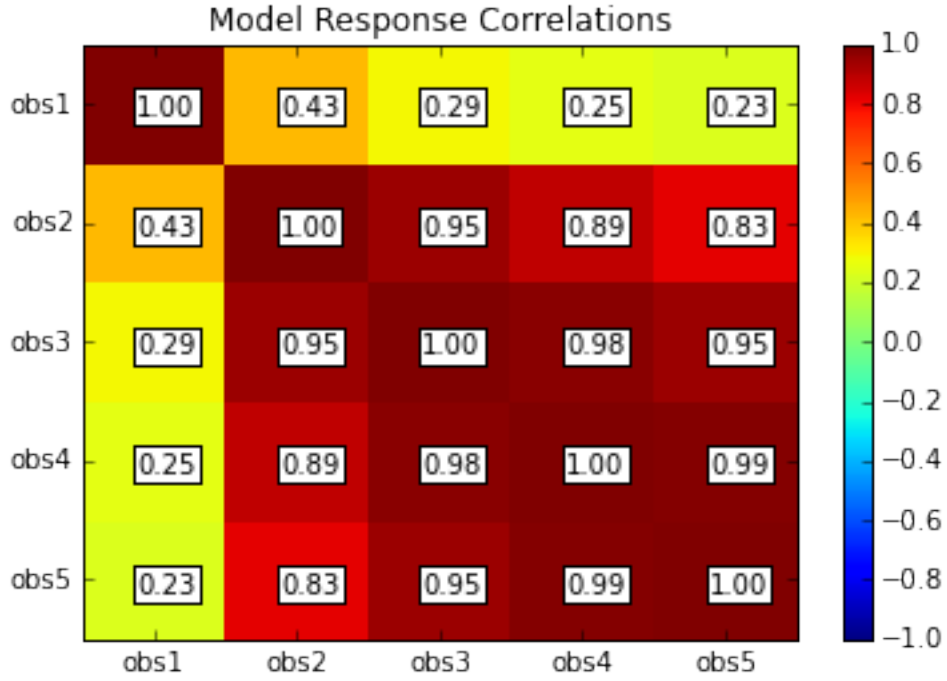



```
obs1:
 Freq:          0.03      0.06      0.10      0.13      0.16      0.20      0.14      0.10      0.05      0.03
 Bins:     0.033      0.23      0.42      0.61       0.8      0.99       1.2       1.4       1.6       1.8

obs2:
 Freq:          0.43      0.23      0.14      0.06      0.04      0.04      0.03      0.02      0.01      0.00
 Bins:    0.0036      0.13      0.25      0.38      0.51      0.63      0.76      0.88         1       1.1

obs3:
 Freq:          0.68      0.12      0.07      0.03      0.03      0.03      0.01      0.01      0.01      0.01
 Bins:   8.5e-05     0.099       0.2       0.3      0.39      0.49      0.59      0.69      0.79      0.89

obs4:
 Freq:          0.78      0.08      0.04      0.03      0.01      0.02      0.01      0.01      0.01      0.01
 Bins:   1.8e-06     0.092      0.18      0.27      0.37      0.46      0.55      0.64      0.73      0.82

obs5:
 Freq:          0.84      0.06      0.03      0.02      0.01      0.02      0.00      0.01      0.01      0.01
 Bins:   4.2e-08     0.091      0.18      0.27      0.36      0.46      0.55      0.64      0.73      0.82
```

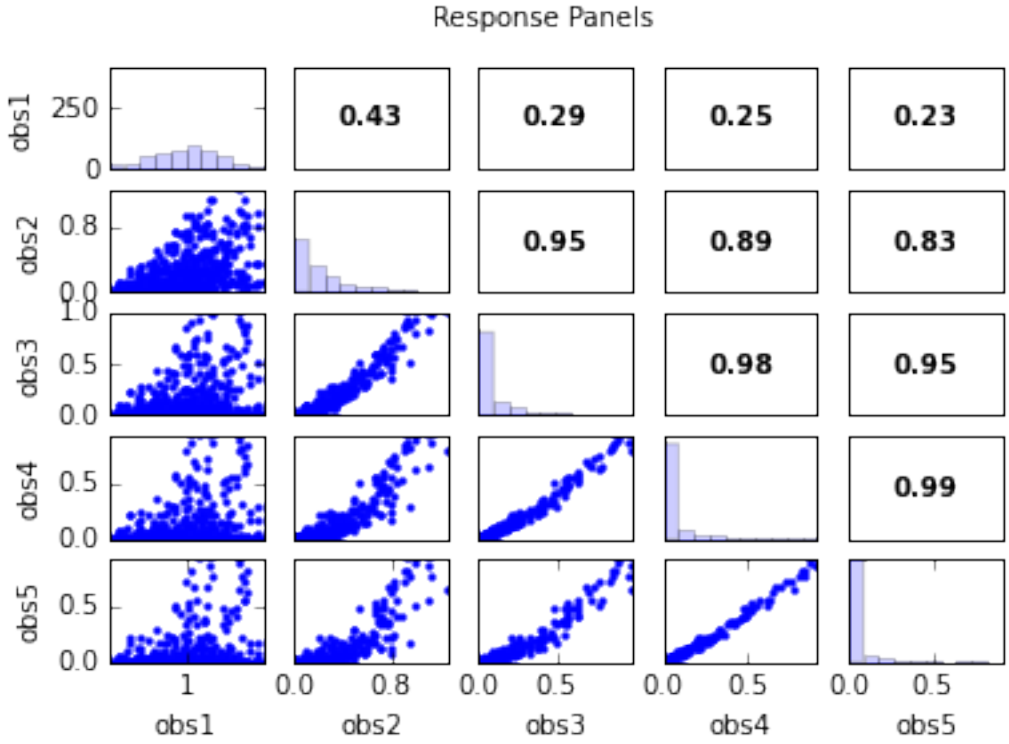
```
rescor = s.responses.corr(plot=True, title='Model Response Correlations')
```

```
             obs1      obs2      obs3      obs4      obs5
obs1         1.00      0.43      0.29      0.25      0.23
```

```
obs2          0.43      1.00      0.95      0.89      0.83
obs3          0.29      0.95      1.00      0.98      0.95
obs4          0.25      0.89      0.98      1.00      0.99
obs5          0.23      0.83      0.95      0.99      1.00
```

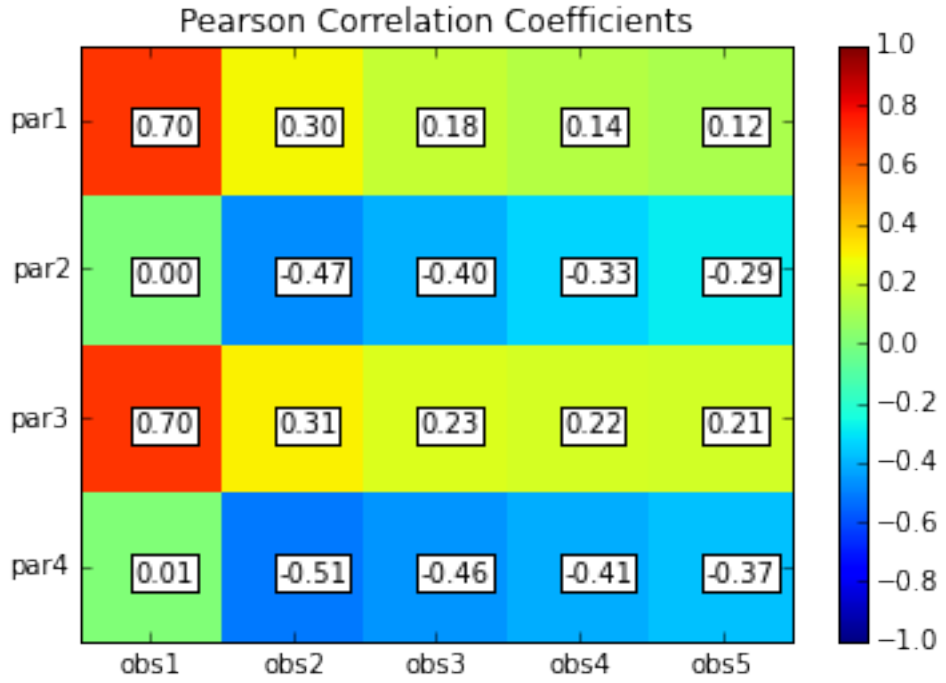



```
out = s.responses.panels(title='Response Panels')
```



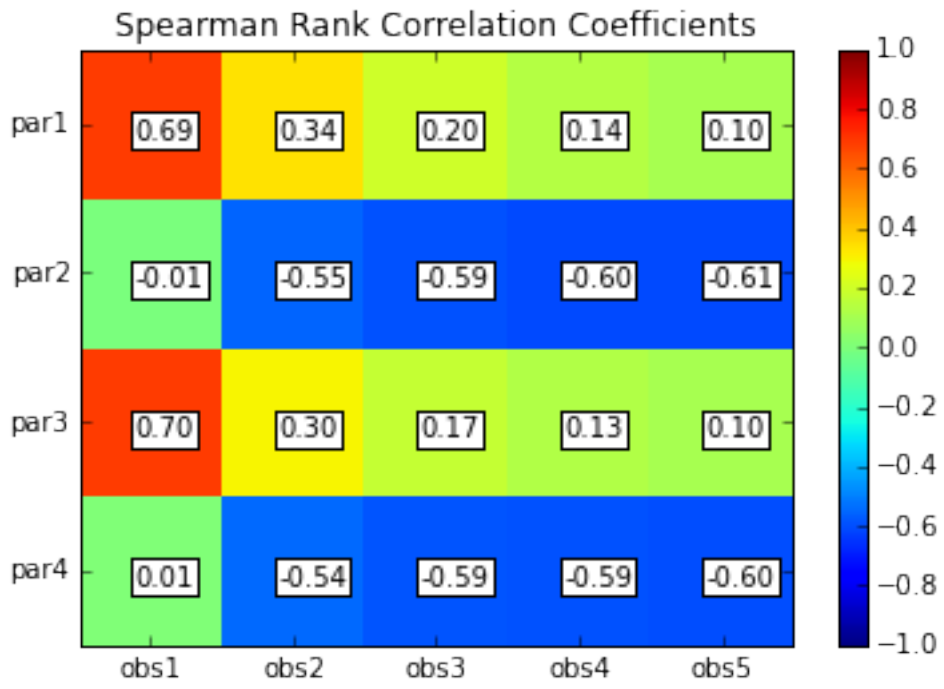
```
# Print and plot parameter/response correlations
print "\nPearson Correlation Coefficients:"
pcorr = s.corr(plot=True,title='Pearson Correlation Coefficients')
```

```
Pearson Correlation Coefficients:
             obs1     obs2     obs3     obs4     obs5
par1         0.70     0.30     0.18     0.14     0.12
par2         0.00    -0.47    -0.40    -0.33    -0.29
par3         0.70     0.31     0.23     0.22     0.21
par4         0.01    -0.51    -0.46    -0.41    -0.37
```

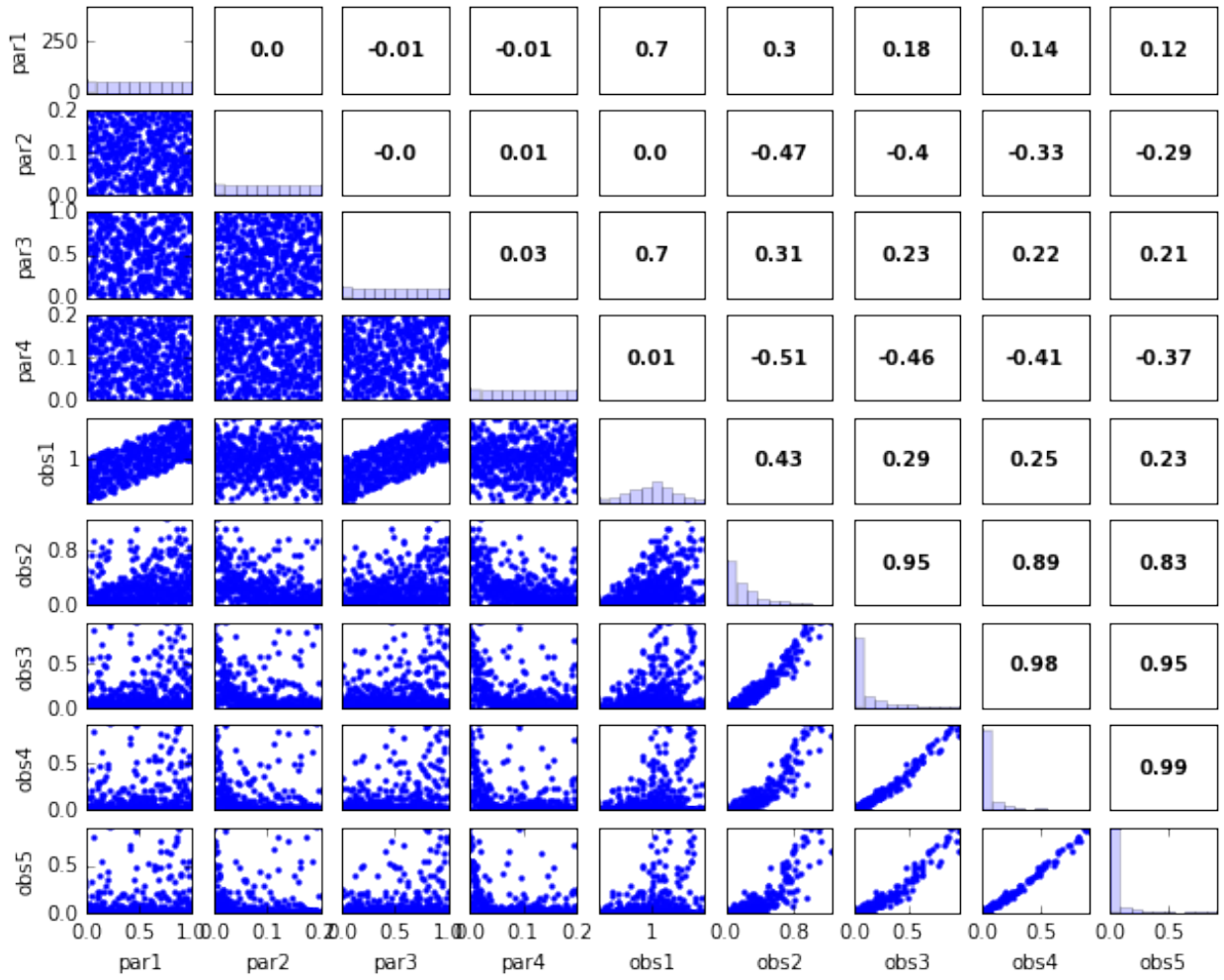



```
print "\nSpearman Correlation Coefficients:"
scorr = s.corr(plot=True,type='spearman',title='Spearman Rank Correlation Coefficients')
```

```
Spearman Correlation Coefficients:
             obs1     obs2     obs3     obs4     obs5
par1         0.69     0.34     0.20     0.14     0.10
par2        -0.01    -0.55    -0.59    -0.60    -0.61
par3         0.70     0.30     0.17     0.13     0.10
par4         0.01    -0.54    -0.59    -0.59    -0.60
```




```
out = s.panels(figsize=(10,8))
```



3.2 Parameter Study

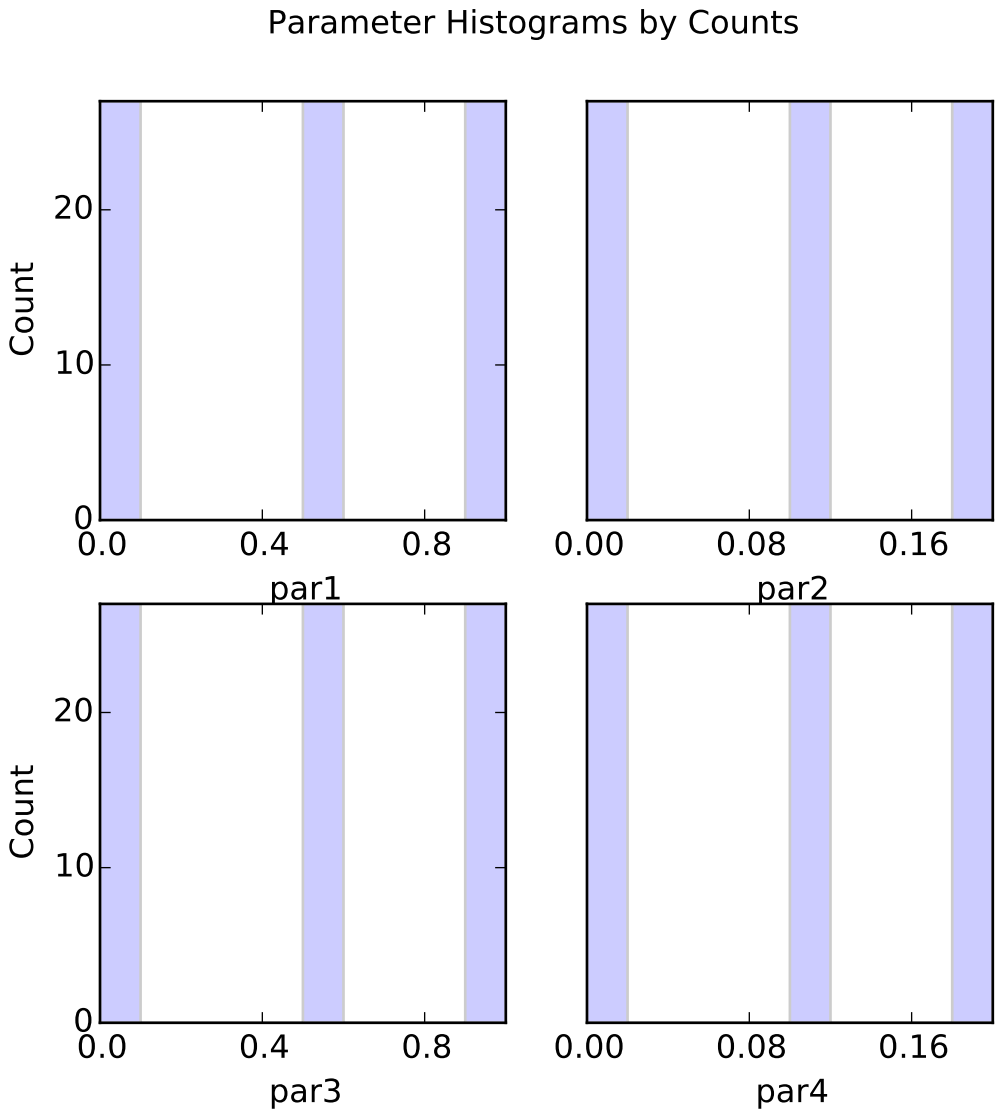
This example demonstrates a parameter study of a 4 parameter 5 response model using the *parstudy* function. The models of the parameter study are run using the *run* function. The generation of diagnostic plots is demonstrated using *hist*, *panels*, and *corr*.

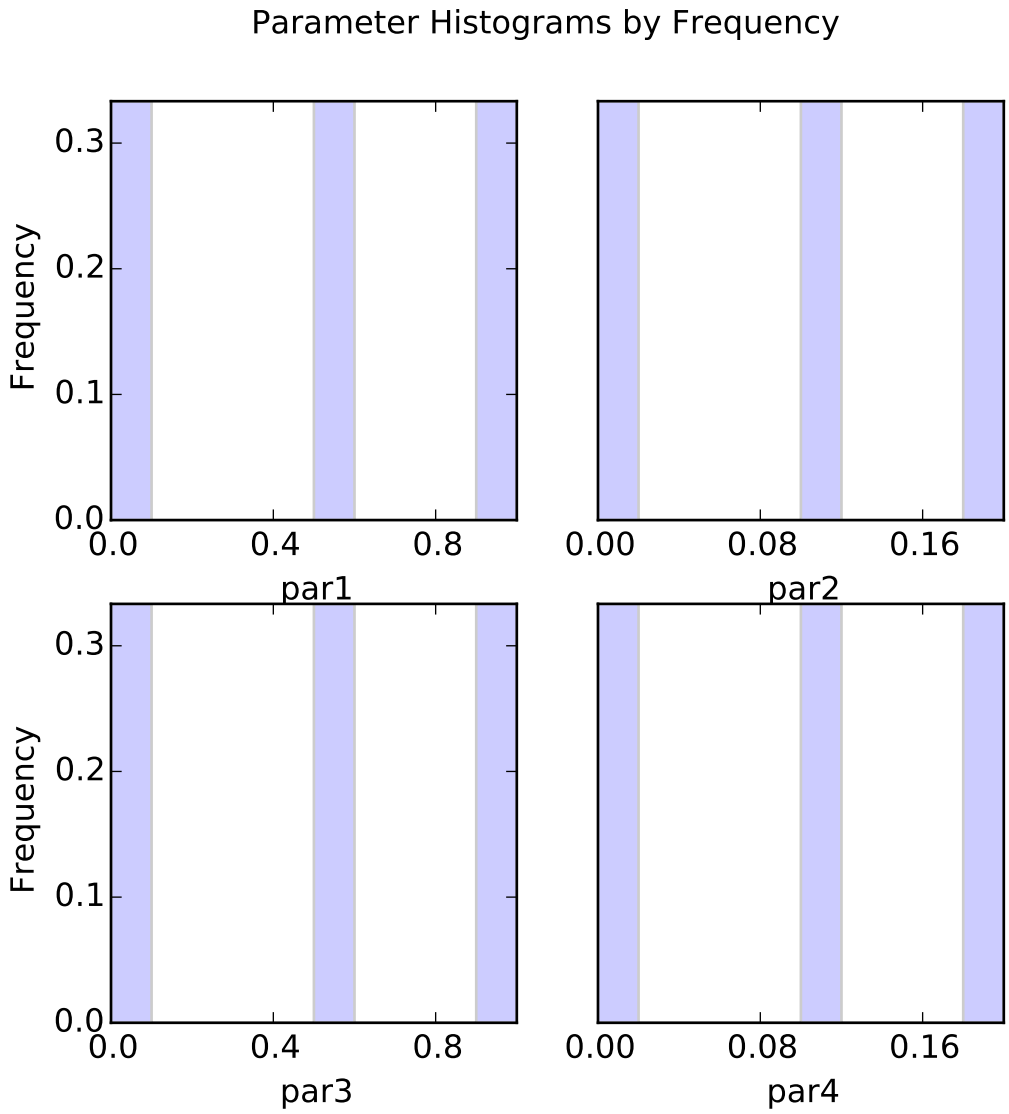
The plots generated by the script are displayed below the code block.

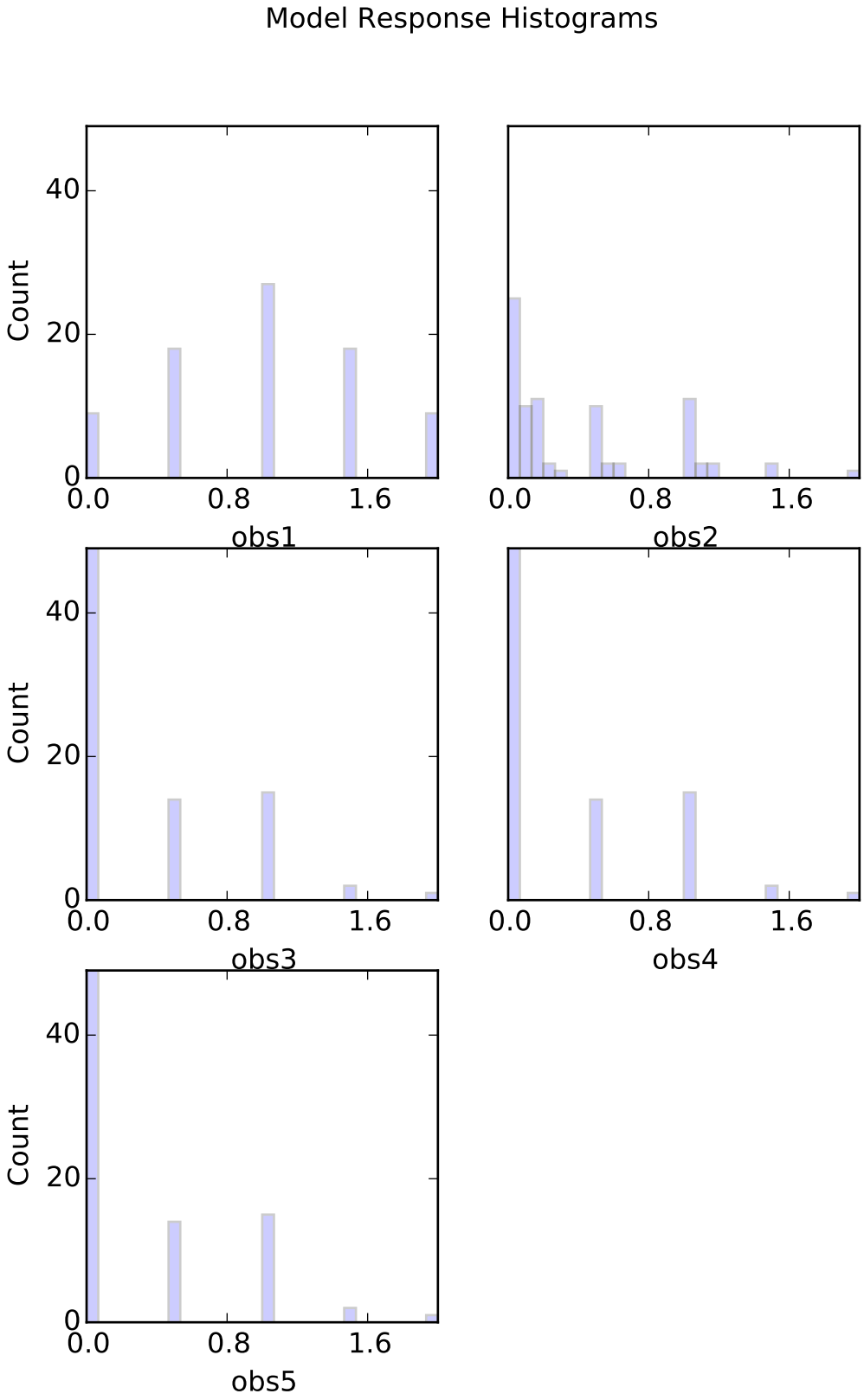
DOWNLOAD SCRIPT

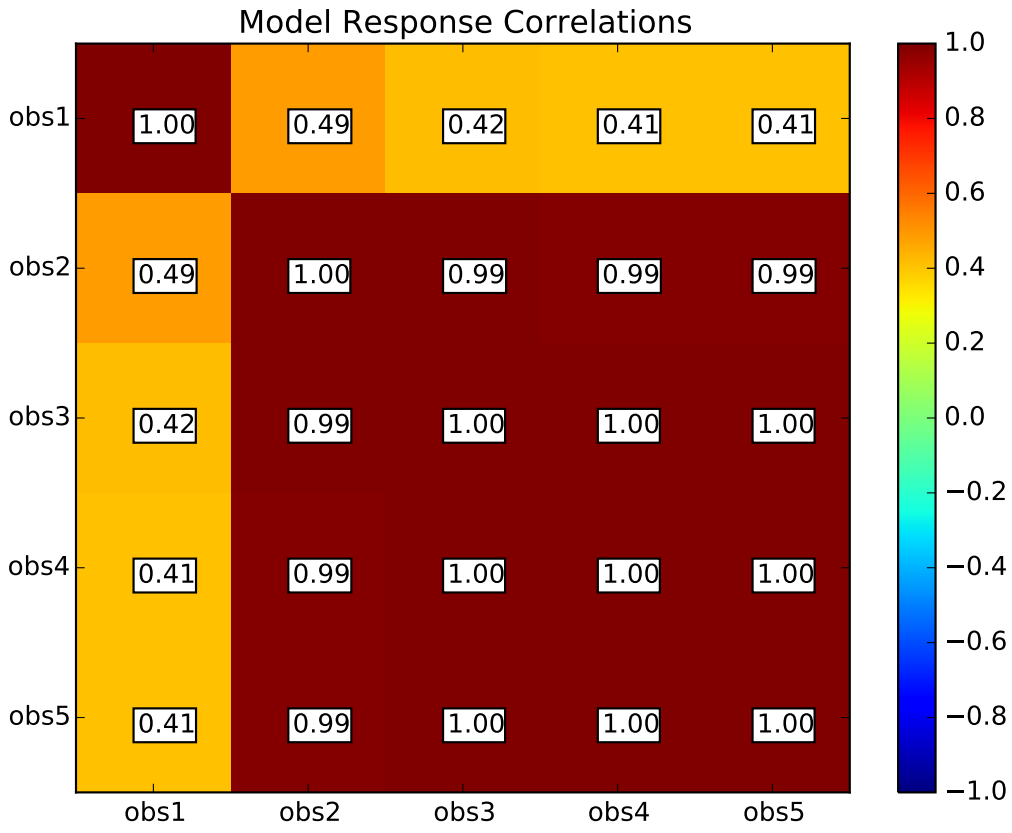
```
import sys,os
try:
    import matk
except:
    try:
        sys.path.append(os.path.join('..','src'))
        import matk
    except ImportError as err:
        print 'Unable to load MATK module: '+str(err)
import numpy
from scipy import arange, randn, exp
from multiprocessing import freeze_support
```

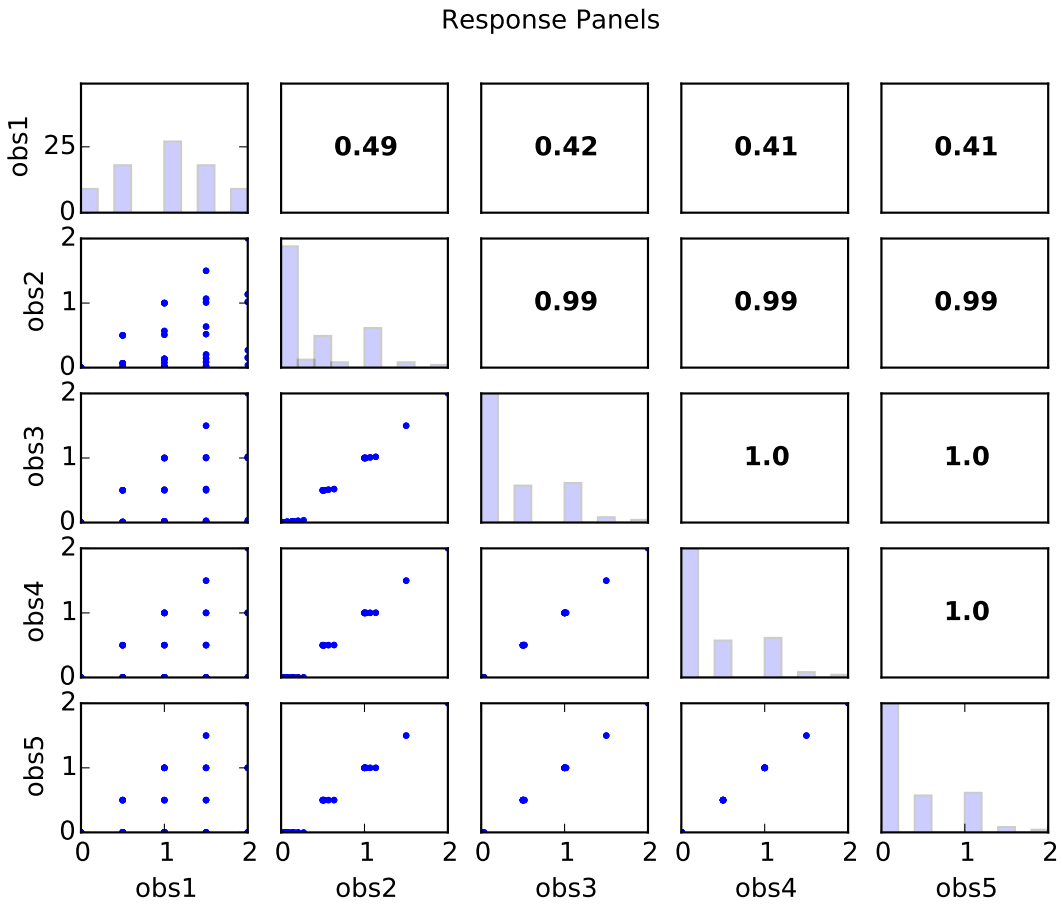
```
# Model function
def dbexpl(p):
    t=arange(0,100,20.)
    y =  (p['par1']*exp(-p['par2']*t) + p['par3']*exp(-p['par4']*t))
    return y

def run():
        # Setup MATK model with parameters
        p = matk.matk(model=dbexpl)
        p.add_par('par1',min=0,max=1)
        p.add_par('par2',min=0,max=0.2)
        p.add_par('par3',min=0,max=1)
        p.add_par('par4',min=0,max=0.2)

        # Create full factorial parameter study with 3 values for each parameter
        s = p.parstudy(nvals=[3,3,3,3])

    # Print values to make sure you got what you wanted
        print "\nParameter values:"
        print s.samples.values

        # Look at sample parameter histograms
        s.samples.hist(ncols=2,title='Parameter Histograms by Counts')
        s.samples.hist(ncols=2,title='Parameter Histograms by Frequency',frequency=True)

        # Run model with parameter samples
        s.run( cpus=2, outfile='results.dat', logfile='log.dat',verbose=False)

        # Look at response histograms, correlations, and panels
        s.responses.hist(ncols=2, bins=30, title='Model Response Histograms')
        rescor = s.responses.corr(plot=True, title='Model Response Correlations')
        s.responses.panels(title='Response Panels')

        # Print and plot parameter/response correlations
        print "\nPearson Correlation Coefficients:"
        pcorr = s.corr(plot=True,title='Pearson Correlation Coefficients')
        print "\nSpearman Correlation Coefficients:"
        scorr = s.corr(plot=True,type='spearman',title='Spearman Rank Correlation Coefficients')
        s.panels(figsize=(10,8))

# Freeze support is necessary for multiprocessing on windows
if __name__== "__main__":
        freeze_support()
        run()
```

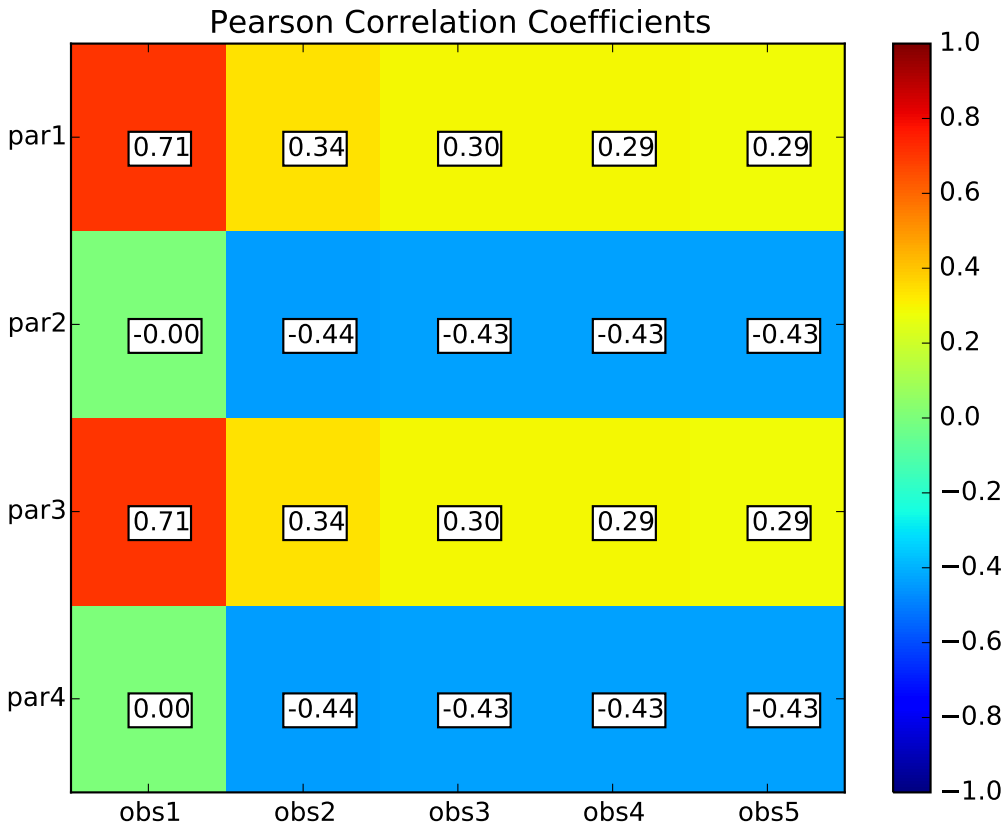

Parameter Histograms by Counts
Count
20
10
0
0.0
0.4
0.8
par1
0.00
0.08
0.16
par2
Count
20
10
0
0.0
0.4
0.8
par3
0.00
0.08
0.16
par4

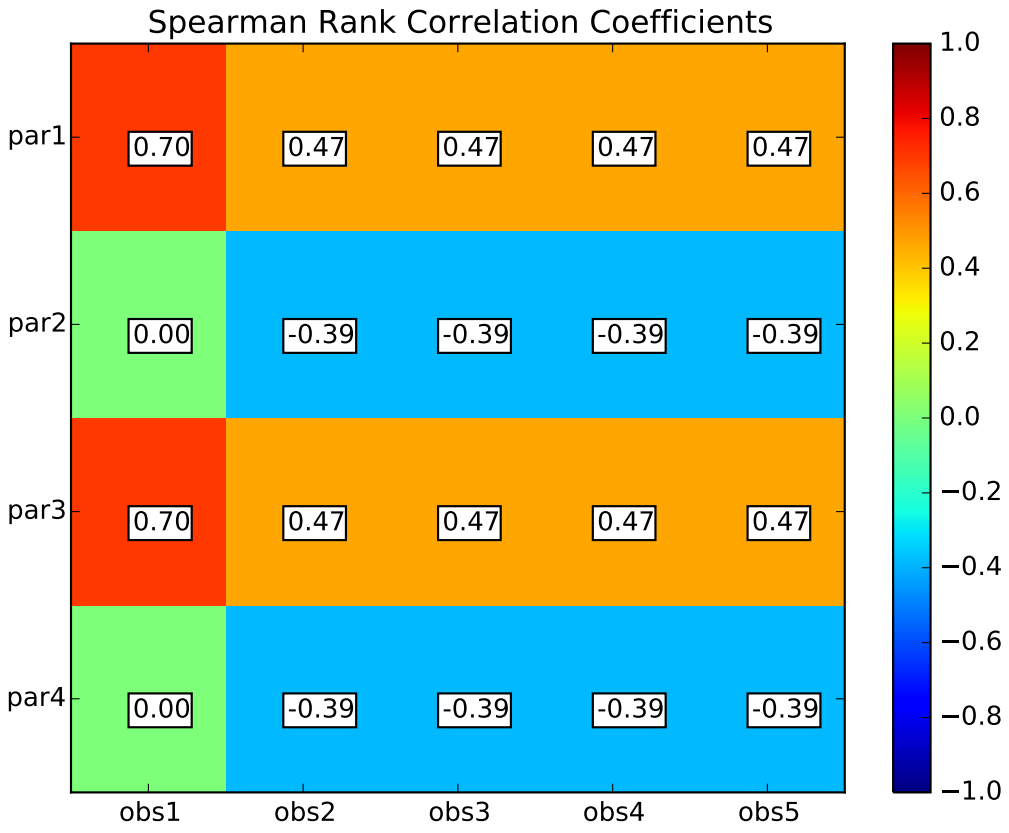

Parameter Histograms by Frequency
Frequency
0.3
0.2
0.1
0.0
0.0
0.4
0.8
par1
0.00
0.08
0.16
par2
Frequency
0.3
0.2
0.1
0.0
0.0
0.4
0.8
par3
0.00
0.08
0.16
par4

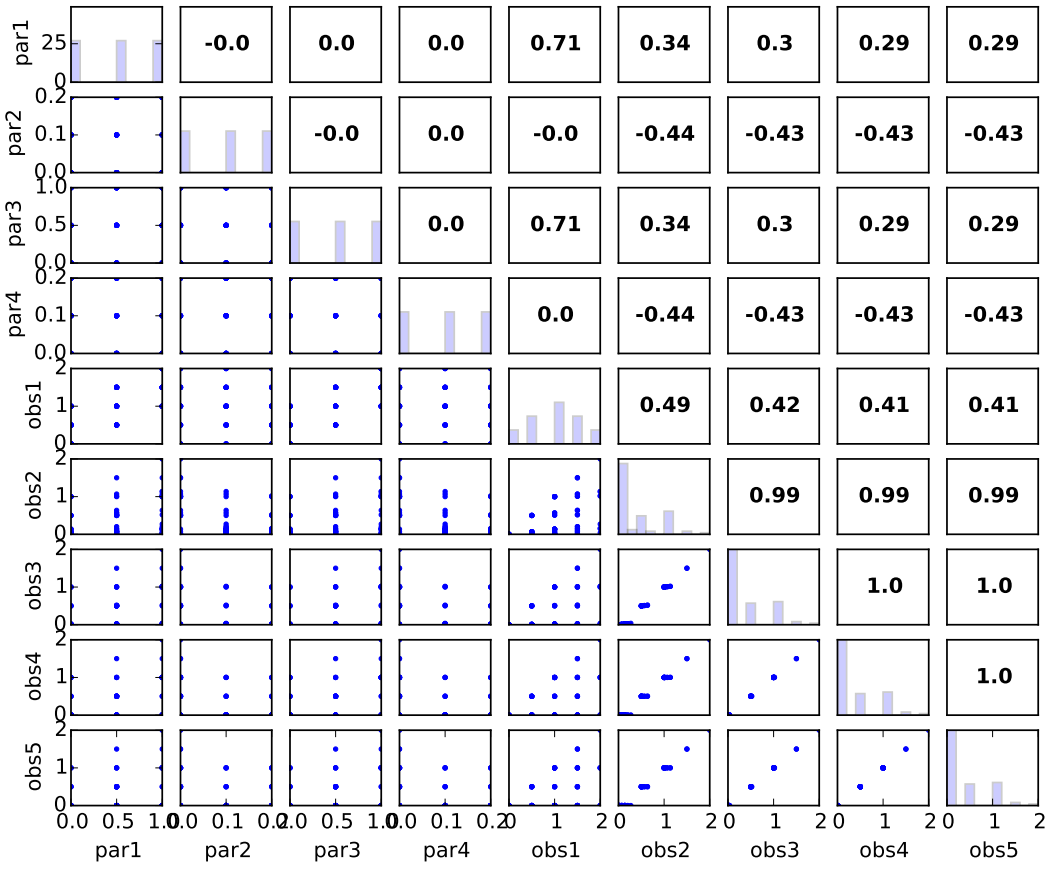

Model Response Histograms
Count
40
20
0
0.0
0.8
1.6
obs1
obs2
obs3
obs4
obs5


Model Response Correlations
obs1
obs2
obs3
obs4
obs5
1.00
0.49
0.42
0.41
0.41
0.49
1.00
0.99
0.99
0.99
0.42
0.99
1.00
1.00
1.00
0.41
0.99
1.00
1.00
1.00
0.41
0.99
1.00
1.00
1.00
obs1
obs2
obs3
obs4
obs5
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0


Response Panels
obs1
obs2
obs3
obs4
obs5
0.49
0.42
0.41
0.41
0.99
0.99
0.99
1.0
1.0
1.0
0
1
2
25


Pearson Correlation Coefficients
par1
par2
par3
par4
0.71
0.34
0.30
0.29
0.29
-0.00
-0.44
-0.43
-0.43
-0.43
0.71
0.34
0.30
0.29
0.29
0.00
-0.44
-0.43
-0.43
-0.43
obs1
obs2
obs3
obs4
obs5
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0


Spearman Rank Correlation Coefficients
par1
par2
par3
par4
0.70
0.47
0.47
0.47
0.47
0.00
-0.39
-0.39
-0.39
-0.39
0.70
0.47
0.47
0.47
0.47
0.00
-0.39
-0.39
-0.39
-0.39
obs1
obs2
obs3
obs4
obs5
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0

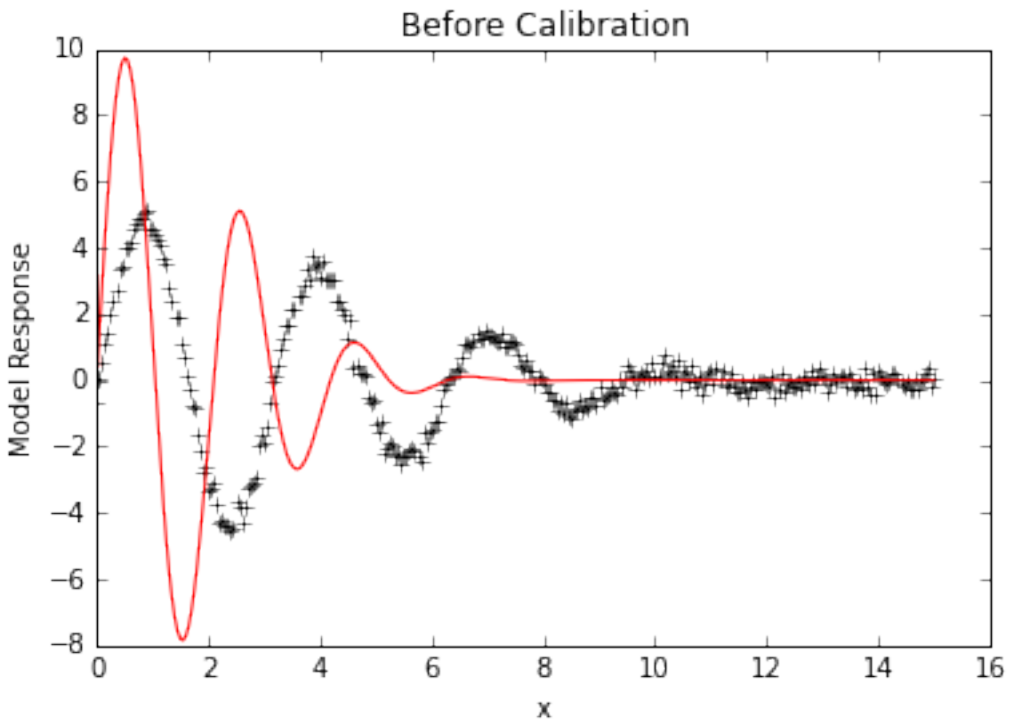


3.3 Calibration Using LMFIT

This example demonstrates the calibration of a simple sinusoidal decay model using the *lmfit* function. *lmfit* uses the MINPACK Levenberg-Marquardt algorithm via the lmfit python module.

```
# %load calibrate_sine_lmfit.py
%matplotlib inline
# Calibration example modified from lmfit webpage
# (http://cars9.uchicago.edu/software/python/lmfit/parameters.html)
import sys,os
try:
    import matk
except:
    try:
        sys.path.append(os.path.join('..','src'))
        import matk
    except ImportError as err:
        print 'Unable to load MATK module: '+str(err)
import numpy as np
from matplotlib import pyplot as plt
from multiprocessing import freeze_support
```

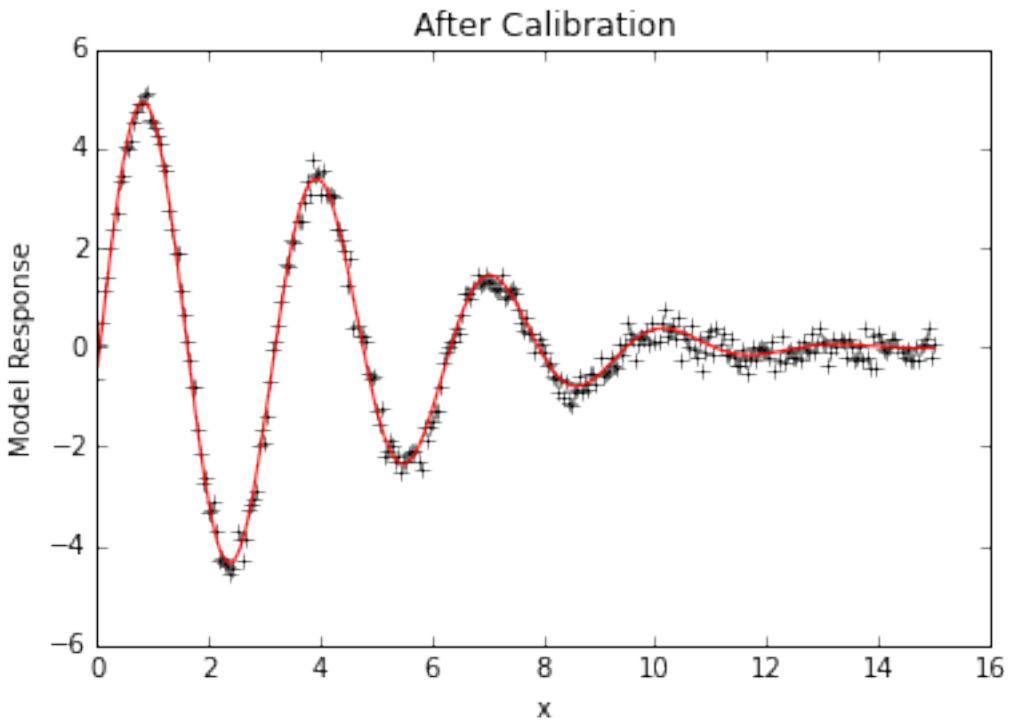
```
# define objective function: returns the array to be minimized
def sine_decay(params, x, data):
    """ model decaying sine wave, subtract data"""
    amp = params['amp']
    shift = params['shift']
    omega = params['omega']
    decay = params['decay']

    model = amp * np.sin(x * omega + shift) * np.exp(-x*x*decay)

    obsnames = ['obs'+str(i) for i in range(1,len(data)+1)]
    return dict(zip(obsnames,model))


# create data to be fitted
x = np.linspace(0, 15, 301)
np.random.seed(1000)
data = (5. * np.sin(2 * x - 0.1) * np.exp(-x*x*0.025) +
        np.random.normal(size=len(x), scale=0.2) )

# Create MATK object
p = matk.matk(model=sine_decay, model_args=(x,data,))

# Create parameters
p.add_par('amp', value=10, min=0.)
p.add_par('decay', value=0.1)
p.add_par('shift', value=0.0, min=-np.pi/2., max=np.pi/2.)
p.add_par('omega', value=3.0)

# Create observation names and set observation values
for i in range(len(data)):
    p.add_obs('obs'+str(i+1), value=data[i])

# Look at initial fit
p.forward()
#f, (ax1,ax2) = plt.subplots(2,sharex=True)
plt.plot(x,data, 'k+')
plt.plot(x,p.simvalues, 'r')
plt.ylabel("Model Response")
plt.xlabel("x")
plt.title("Before Calibration")
plt.show()
```




```
# Calibrate parameters to data, results are printed to screen
print "Calibration results:"
lm = p.lmfit(cpus=2)
```

```
Calibration results:
[[Variables]]
     amp:       5.011399 +/- 0.040469 (0.81%) initial =  5.011398
     decay:     0.024835 +/- 0.000465 (1.87%) initial =  0.024835
     omega:     1.999116 +/- 0.003345 (0.17%) initial =  1.999116
     shift:    -0.106184 +/- 0.016466 (15.51%) initial = -0.106207
[[Correlations]] (unreported correlations are <  0.100)
    C(omega, shift)              = -0.785
    C(amp, decay)                =  0.584
    C(amp, shift)                = -0.117
None
SSR:  12.8161378922
```

```
# Look at calibrated fit
plt.plot(x,data, 'k+')
plt.plot(x,p.simvalues, 'r')
plt.ylabel("Model Response")
plt.xlabel("x")
plt.title("After Calibration")
plt.show()
```



3.4 Linear Analysis of Calibration Using PYEMU

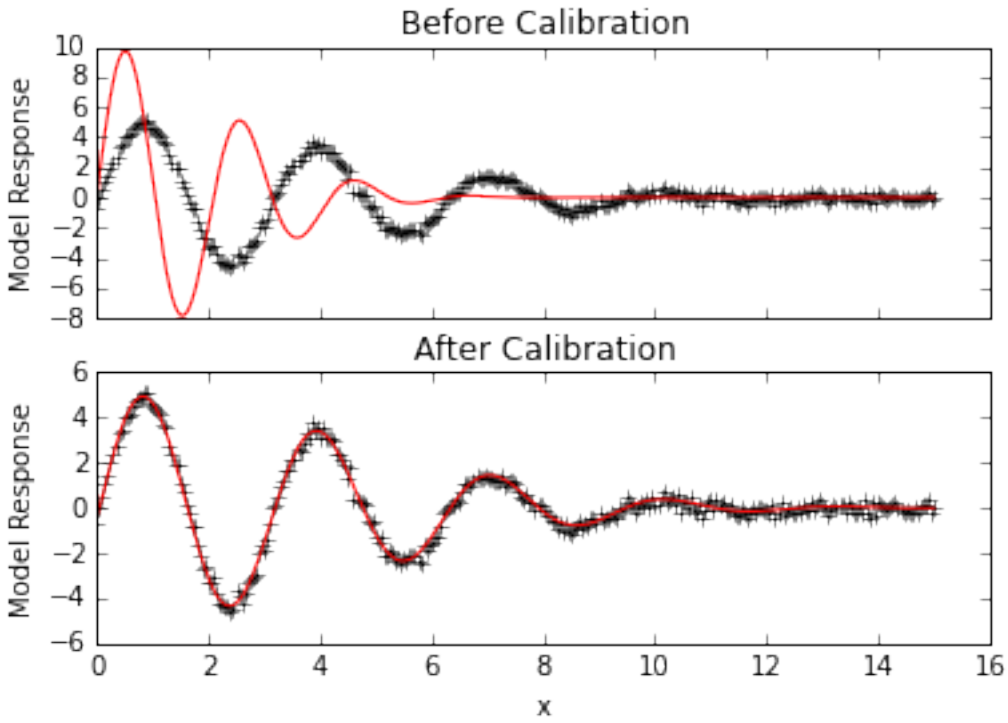
This example demonstrates a linear analysis of the *Calibration Using LMFIT* example using the pyemu module (https://github.com/jtwhite79/pyemu). Singular values from pyemu's eigenanalysis of the jacobian are plotted and identifiability of parameters are printed. The resulting identifiability values indicate that one of the parameters (**amp**) is significantly less identifiable than the others.

```
# %load calibrate_sine_lmfit_pyemu.py
%matplotlib inline
```

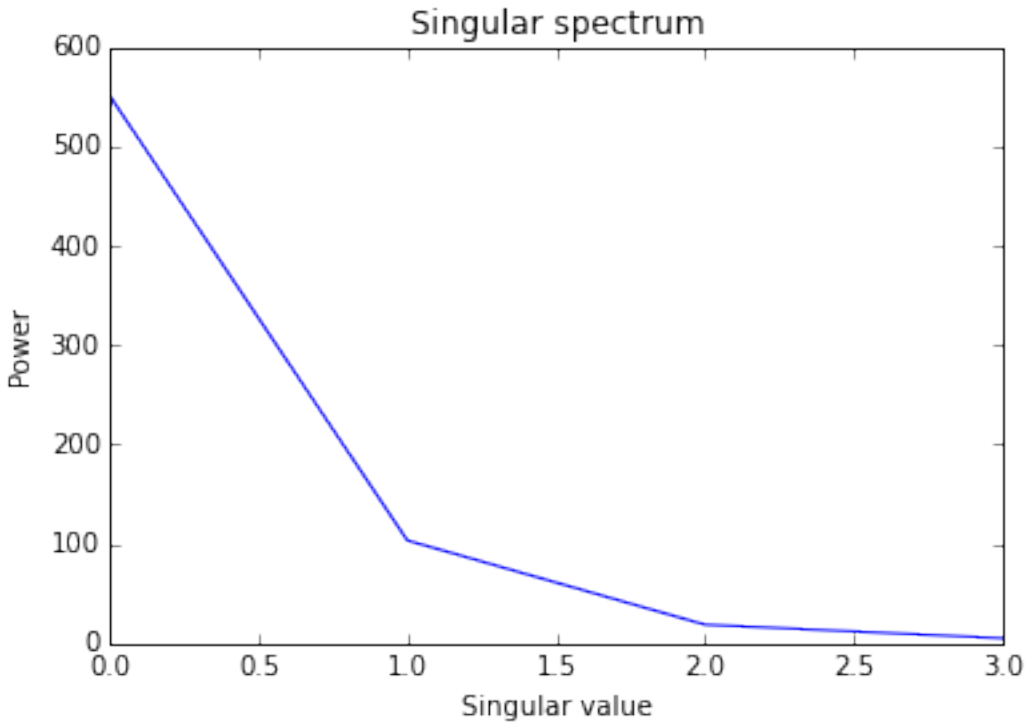
```
import sys,os
import matk
import numpy as np
from matplotlib import pyplot as plt
from multiprocessing import freeze_support
sys.path.append('/Users/dharp/source-mac/pyemu')
import pyemu
from mat_handler import matrix,cov

# define objective function: returns the array to be minimized
def sine_decay(params, x, data):
    """ model decaying sine wave, subtract data"""
    amp = params['amp']
    shift = params['shift']
    omega = params['omega']
    decay = params['decay']

    model = amp * np.sin(x * omega + shift) * np.exp(-x*x*decay)

    obsnames = ['obs'+str(i) for i in range(1,len(data)+1)]
```

```
    return dict(zip(obsnames,model))


# create data to be fitted
x = np.linspace(0, 15, 301)
np.random.seed(1000)
data = (5. * np.sin(2 * x - 0.1) * np.exp(-x*x*0.025) +
        np.random.normal(size=len(x), scale=0.2) )

# Create MATK object
p = matk.matk(model=sine_decay, model_args=(x,data,))

# Create parameters
p.add_par('amp', value=10, min=5., max=15.)
p.add_par('decay', value=0.1, min=0, max=10)
p.add_par('shift', value=0.0, min=-np.pi/2., max=np.pi/2.)
p.add_par('omega', value=3.0, min=0, max=10)

# Create observation names and set observation values
for i in range(len(data)):
    p.add_obs('obs'+str(i+1), value=data[i])

# Look at initial fit
p.forward()
f, (ax1,ax2) = plt.subplots(2,sharex=True)
ax1.plot(x,data, 'k+')
ax1.plot(x,p.simvalues, 'r')
ax1.set_ylabel("Model Response")
ax1.set_title("Before Calibration")

# Calibrate parameters to data, results are printed to screen
lm = p.lmfit(cpus=2)
# Look at calibrated fit
ax2.plot(x,data, 'k+')
ax2.plot(x,p.simvalues, 'r')
ax2.set_ylabel("Model Response")
ax2.set_xlabel("x")
ax2.set_title("After Calibration")
plt.show()
```

```
[[Variables]]
     amp:       5.011593 +/- 0.013966 (0.28%) initial =  10.000000
     decay:     0.024837 +/- 0.000231 (0.93%) initial =  0.100000
     omega:     1.999111 +/- 0.013378 (0.67%) initial =  3.000000
     shift:    -0.106200 +/- 0.016466 (15.50%) initial =  0.000000
[[Correlations]] (unreported correlations are <  0.100)
    C(omega, shift)              = -0.785
    C(amp, decay)                =  0.584
    C(amp, shift)                = -0.117
None
SSR:  12.8161392911
```




```
# Recompute jacobian at calibration point
J = p.Jac(cpus=2)

# Use pyemu module to analyze identifiability of parameters within calibration
# Create matrix object of jacobian for pyemu
m = matrix(x=J,row_names=p.obsnames,col_names=p.parnames)
# Create prior parameter covariance matrix using parameter bounds (uniform priors)
parcov_arr = np.array([((mx-mn)/4.)**2 for mx,mn in zip(p.parmaxs,p.parmins)])*np.eye(len(p.pars))
parcov = cov(parcov_arr,names=p.parnames)
# Create prior observation covariance matrix based on observation weights (p.obsweights)
# In this case, it is an identity matrix since all weights are one
obscov_arr = np.eye(len(p.obs))*p.obsweights
obscov = cov(obscov_arr,names=p.obsnames)

# Create pyemu error variance object using jacobian and parameter and observation covariances
#la = pyemu.errvar(jco=m,parcov=parcov,obscov=obscov,forecasts=['obs1'],omitted_parameters=['omega'],
la = pyemu.errvar(jco=m,parcov=parcov,obscov=obscov,forecasts=['obs1'])

# Plot the singular values from the eigenanalysis of the jacobian
s = la.qhalfx.s
plt.title("Singular spectrum")
plt.ylabel("Power")
plt.xlabel("Singular value")
plt.plot(s.x)
plt.show()

# Print identifiability of parameters
# The results indicate that 'amp' has low identifiability relative to other parameters
ident_df = la.get_identifiability_dataframe(3)
print "\nIdentifiability of parameters:"
print ident_df['ident']
```




```
Identifiability of parameters:
amp      0.001426
decay    0.999956
shift    0.998698
omega    0.999919
Name: ident, dtype: float64
```

3.5 Markov Chain Monte Carlo Using PYMC

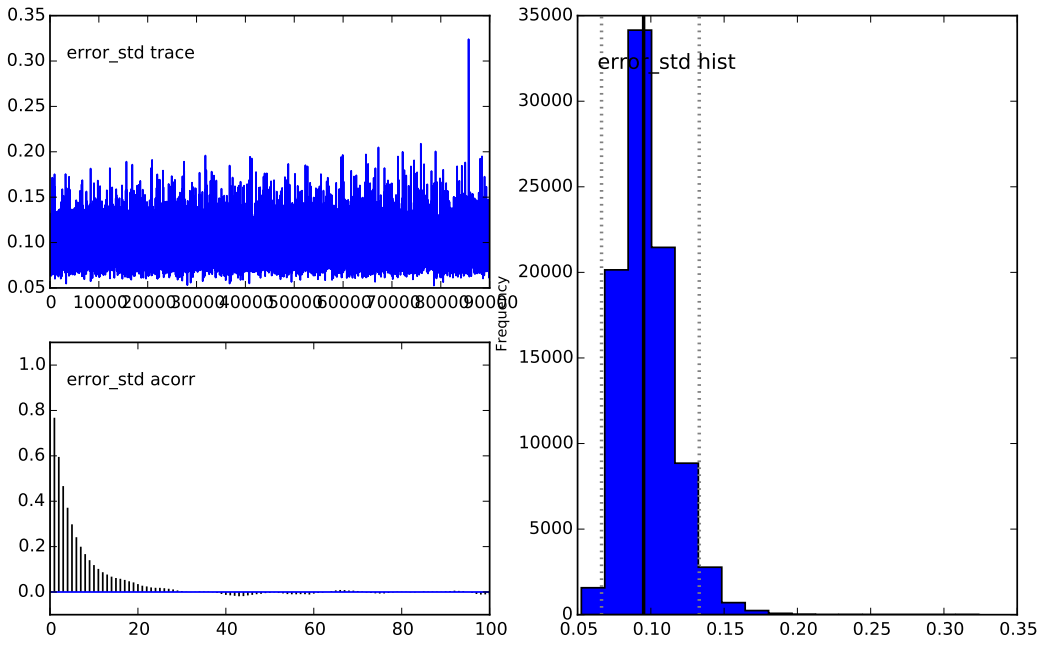
This example demonstrates Markov Chain Monte Carlo using PYMC (https://pymc-devs.github.io/pymc/) with the *MCMC* function.

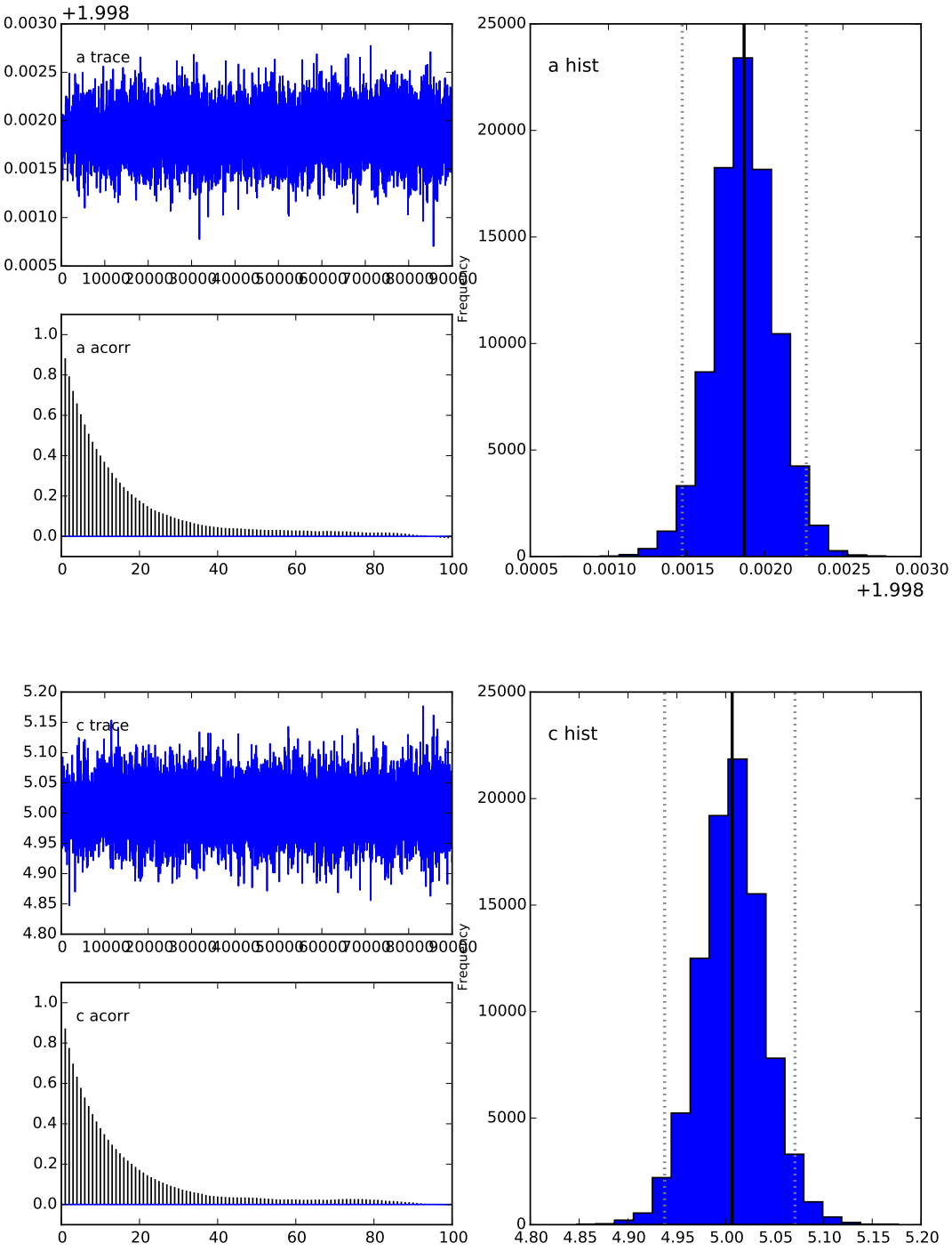
The plots generated by the script are displayed below the code block.

DOWNLOAD SCRIPT

```
import sys,os
from numpy import array, double, arange, random
try:
    import matk
except:
    try:
        sys.path.append(os.path.join('..','src'))
        import matk
    except ImportError as err:
        print 'Unable to load MATK module: '+str(err)
from multiprocessing import freeze_support

# Define basic function
def f(pars):
```

```
    a = pars['a']
    c = pars['c']
    m=double(arange(20))
    m=a*(m**2)+c
    return m

def run():
    # Create matk object
    prob = matk.matk(model=f)

    # Add parameters with 'true' parameters
    prob.add_par('a', min=0, max=10, value=2)
    prob.add_par('c', min=0, max=30, value=5)

    # Run model using 'true' parameters
    prob.forward()

    # Create 'true' observations with zero mean, 0.5 st. dev. gaussian noise added
    prob.obsvalues = prob.simvalues + random.normal(0,0.1,len(prob.simvalues))

    # Run MCMC with 100000 samples burning (discarding) the first 10000
    M = prob.MCMC(nruns=100000,burn=10000)

    # Plot results, PNG files will be created in current directory
    prob.MCMCplot(M)

# Freeze support is necessary for multiprocessing on windows
if __name__== "__main__":
        freeze_support()
        run()
```





3.6 External Simulator (Python script)

This example demonstrates the same calibration as *Calibration Using LMFIT*, but sets up the MATK model as an python script to demonstrate how to use an external simulator. Similar to the *External Simulator (FEHM Groundwater*

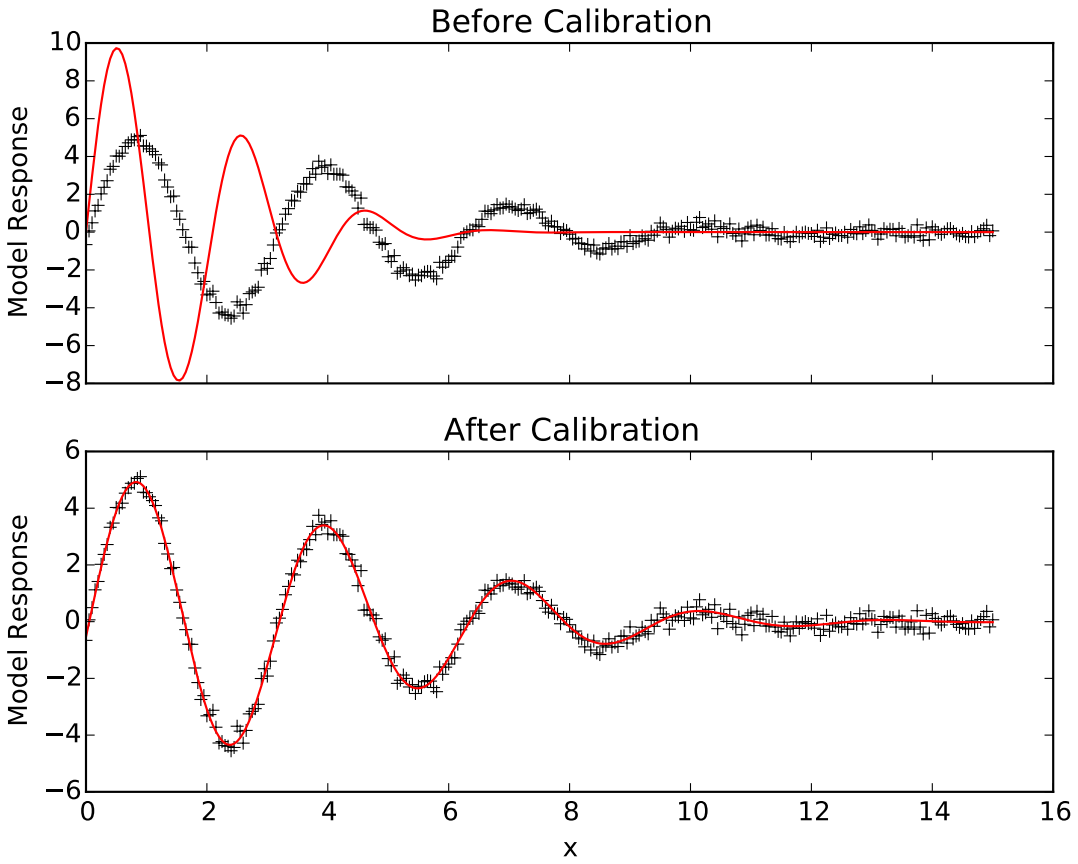
Flow Simulator) example, the subprocess call (https://docs.python.org/2/library/subprocess.html) method is used to make system calls to run the *model* and MATK's `pest_io.tpl_write` is used to create model input files with parameters in the correct locations. The pickle package (https://docs.python.org/2/library/pickle.html) is used for I/O of the model results between the external simulator (sine.tpl) and the MATK model.

DOWNLOAD SCRIPT

DOWNLOAD MODEL TEMPLATE FILE

```
# Calibration example modified from lmfit webpage
# (http://cars9.uchicago.edu/software/python/lmfit/parameters.html)
# This example demonstrates how to calibrate with an external code
# The idea is to replace `python sine.py` in run_extern with any
# terminal command to run your model.
import sys,os
import numpy as np
from matplotlib import pyplot as plt
from multiprocessing import freeze_support
from subprocess import Popen,PIPE,call
from matk import matk, pest_io
import cPickle as pickle

def run_extern(params):
    pest_io.tpl_write(params,'../sine.tpl','sine.py')
    ierr = call('python sine.py', shell=True)
    out = pickle.load(open('sine.pkl','rb'))
    return out

# create data to be fitted
x = np.linspace(0, 15, 301)
np.random.seed(1000)
data = (5. * np.sin(2 * x - 0.1) * np.exp(-x*x*0.025) +
        np.random.normal(size=len(x), scale=0.2) )

# Create MATK object
p = matk(model=run_extern)

# Create parameters
p.add_par('amp', value=10, min=0.)
p.add_par('decay', value=0.1)
p.add_par('shift', value=0.0, min=-np.pi/2., max=np.pi/2.)
p.add_par('omega', value=3.0)

# Create observation names and set observation values
for i in range(len(data)):
    p.add_obs('obs'+str(i+1), value=data[i])

# Look at initial fit
init_vals = p.forward(workdir='initial',reuse_dirs=True)
f, (ax1,ax2) = plt.subplots(2,sharex=True)
ax1.plot(x,data, 'k+')
ax1.plot(x,p.simvalues, 'r')
ax1.set_ylabel("Model Response")
ax1.set_title("Before Calibration")

# Calibrate parameters to data, results are printed to screen
p.lmfit(cpus=2,workdir='calib')

# Look at calibrated fit
```

```
ax2.plot(x,data, 'k+')
ax2.plot(x,p.simvalues, 'r')
ax2.set_ylabel("Model Response")
ax2.set_xlabel("x")
ax2.set_title("After Calibration")
plt.show()
```



Template file used by *pest_io.tpl_write*. Note the header **ptf %** and parameter locations indicated by **%** in the file.

```
ptf %
import numpy as np
import cPickle as pickle

# define objective function: returns the array to be minimized
def sine_decay():
    """ model decaying sine wave, subtract data"""
    amp = %amp%
    shift = %shift%
    omega = %omega%
    decay = %decay%

    x = np.linspace(0, 15, 301)
    model = amp * np.sin(x * omega + shift) * np.exp(-x*x*decay)

    obsnames = ['obs'+str(i) for i in range(1,model.shape[0]+1)]
```

```
    return dict(zip(obsnames,model))


if __name__== "__main__":
    out = sine_decay()
    pickle.dump(out,open('sine.pkl', 'wb'))
```

3.7 External Simulator (FEHM Groundwater Flow Simulator)

This example demonstrates a simple parameter study with an external simulator (FEHM groundwater simulator) using the subprocess call (https://docs.python.org/2/library/subprocess.html) function to make system calls. MATK's *pest_io.tpl_write* is used to create model input files with parameters in the correct locations.

DOWNLOAD SCRIPT

```
import sys,os
try:
    import matk
except:
    try:
        sys.path.append('..'+os.sep+'..'+os.sep+'src')
        import matk
    except ImportError as err:
        print 'Unable to load MATK module: '+str(err)
import numpy
import pest_io
from subprocess import call
import fpost
from multiprocessing import freeze_support

# Model function
def fehm(p):
    # Create simulator input file
    pest_io.tpl_write(p, '../intact.tpl', 'intact.dat')
    # Call simulator
    ierr = call('xfehm ../intact.files', shell=True)
    # Collect result of interest and return
    o = fpost.fnodeflux('intact.internode_fluxes.out')
    return [o[o.nodepairs[-1]]['liquid'][-1]]

def run():
    # Setup MATK model with parameters
    p = matk.matk(model=fehm)
    p.add_par('por0',min=0.1,max=0.3)

    # Create LHS sample
    s = p.parstudy(nvals=[3])

    # Run model with parameter samples
    s.run( ncpus=2, workdir_base='workdir', outfile='results.dat', logfile='log.dat',verbose=False,re

    # Look at response histograms, correlations, and panels
    print 'Parameter Response'
    for pa,re in zip(s.samples.values, s.responses.values): print pa[0], re[0]
    s.responses.hist(ncols=2,title='Model Response Histograms')
```

```
# Freeze support is necessary for multiprocessing on windows
if __name__== "__main__":
        freeze_support()
        run()
```

Template file used by *pest_io.tpl_write*. Note the header **ptf %** and parameter location **%por0%** in the file. This example illustrates how to use an external simulator and other files required to run this example are not included.

```
ptf %
title: 1-d heat pipe calculation                11/28/12
#   All nodes are crushed salt
#*************************************************************************75
zone
        1
nnum
        6 1 2 3 4 5 6

salt
saltvapr
1
saltnum
permavg 1.0
poravg  1.0
pormin 1.d-5

saltvcon
  3   26.85  5.4 1.14
  4   26.85  1.08 -270. 370. -136. 1.5 5  1.14

  1 0 0 2
  1 0 0 1

saltppor
7
  1 0 0   4.866e-9 4.637 1.e-3  %por0%

saltadif
 333
saltend
node
6
1 2 3 4 5 6
perm
   1 0 0 -12 -12 -12

rlp
        3  0.05 1.0  4   1.56   -10.  0.06
        1  0.0  0.   1.0    1.0   0.15  1.0

        1      0       0       2

rock
 1 1 1   2165.  931.  0.01
 2 2 1   2165.  931.  0.1
 3 3 1   2165.  931.  0.3
 4 4 1   2165.  931.  0.5
 5 5 1   2165.  931.  0.7
 6 6 1   2165.  931.  0.9
```

```
#vcon
#  3   26.85  5.4   1.14
#  4   26.85  1.08 -270. 370. -136. 1.5 5  1.14
#
#  1 0 0 2
#
#ppor
#7
#  1 0 0   4.866e-9 4.637 1.e-3  0.20
#
flxo
5
1 2
2 3
3 4
4 5
5 6
# - - - - - - - - - - - - - - - - -
time
1.e-4  1.e-2    600    01  1995  5  0.0
0.5 -2.0 1 1
1.0 -2.0 1 1
5.0 -2.0 1 1
10.0 -2.0 1 1

ctrl
  -15   1.e-04   24  100 gmre
1   0 0 2
0
1.0   0.0  1.
15   1.5  1.e-09 1.e-4
0 +1
iter
1.e-5 1.e-5 1.e-3 -1.e-5 1.2
00 0 0 10 1000.
sol
1    -1
#- - - - - - - - - - - - - - - -
#vapl    #gaz debug comment out
#adif
#333
#- - - - - - - - - - - - -
pres       #initial pres sat
1 0 0 0.1   0.10  2

ngas  reset P
3
1  1  1 -20.
2  2  1 -40.
3  3  1 -60.
4  4  1 -80.
5  5  1 -100.
6  6  1 -120.



hflx
 1 1 1 20.   1.e6
```

```
 6 6 1 120.   1.e6

#- - - - - - - - - - - - -
#cont
#avsx  100  10000.
#temp
#sat
#porosity
#perm
#mat
#conc
#pres
#vap
#density
#liquid
#end
#cden now in moles 226/7.5 = ****************************************
cden
1
30.1
#*************************************************************
trac
 0  1  1.e-7  1.0
 0. 3652.5  1.e6  1.e6
 50  1.6 1.e-1 1.  1
 2
 1
0 0 0 1  1.e-9 .33333 .33333 .33333

  1 0 0 1

  1 0 0  6.16


0
  1 0 0 17.2414


rxn
** NCPLX,NUMRXN
         0, 1
** Coupling of the aqueous components (dRi/dUj)
1
1
** IDCPNT(IC),CPNTNAM(IC),IFXCONC(IC),CPNTPRT(IC) (comp,name,cond.; NCPNT rows)
    1     A[aq]    0      0    1.e-9
** IDCPLX(IX), CPLXNAM(IX),CPLXPRT(IX) (ID # and name of complex, NCPLX rows)
** IDIMM(IM), IMMNAM(IM),IMMPRT(IM)(ID # and name of immobile spec, NIMM rows)
    1  A[s]    0
** IDVAP(IV), VAPNAM(IM),VAPPRT(IV) (ID # and name of vapor species, NVAP rows)
** skip nodes? **
0   -1
** RSDMAX
   1.0e-9
******  Chemical reaction information  ********
** LOGKEQ (=0 if stability constants are given as K, =1 if given as log(K))
** CKEQ(IX), HEQ(IX) (Stability constants and Enthalpys, NCPLX rows)
** STOIC(IX,IC) (Stoichiometric coeff: NCPLX rows, NCPNT columns)
```

```
** Precipiation/Dissolution REACTION (type 7) **
        8
** Where does the reaction take place ? ***
1 0 0

** immobile species participating in reaction **
        1
** the number of total aqueous species in reaction **
        1
** total aqueous species in reaction **
        1
** stoichiometry of the immobilie species **
        1
** stoichiometry of the aqueous species **
        1
** solubility product **
lookup 8
     10 40 100 150 200 250 300 350
    6.12 6.23  6.65 7.21  8.00  9.06  10.45  12.27
porosity change
** molecular weight of mineral (kg/mol), density of mineral (kg/m^3) SALT Wikipedia**
0.0558 , 2165.
** rate constant (moles/(m^2*sec)) **
        0.01
** surface area of the mineral (m^2) **
        1
stop

** rate constant (moles/(m^2*sec)) **
        0.1
```

3.8 Running on Cluster

This example demonstrates the same calibration as *Calibration Using LMFIT*, but sets up the MATK model as an external simulator to demonstrate how to utilize cluster resources. Similar to the *External Simulator (FEHM Groundwater Flow Simulator)* example, the subprocess call (https://docs.python.org/2/library/subprocess.html) method is used to make system calls to run the *model* and MATK's `pest_io.tpl_write` is used to create model input files with parameters in the correct locations. The pickle package (https://docs.python.org/2/library/pickle.html) is used for I/O of the model results between the external simulator (sine.tpl) and the MATK model. This example is designed for a cluster using slurm and moab and will have to be modified for use on clusters using other resource managers.

DOWNLOAD SCRIPT

DOWNLOAD MODEL TEMPLATE FILE

```
# Calibration example modified from lmfit webpage
# (http://cars9.uchicago.edu/software/python/lmfit/parameters.html)
# This example demonstrates how to utilize cluster resources
# for calibration with an external simulator.
# The idea is to replace `python sine.py` in run_extern with any
# terminal command to run your model.
# Also note that the hosts dictionary can contain any remote hosts
# accessible by passwordless ssh, for instance other workstations
# on your network.
import sys,os
import numpy as np
```

```
from matplotlib import pyplot as plt
from multiprocessing import freeze_support
from subprocess import Popen,PIPE,call
from matk import matk, pest_io
import cPickle as pickle

def run_extern(params, hostname=None, processor=None):
    pest_io.tpl_write(params,'../sine.tpl','sine.py')
    ierr = call('ssh '+hostname+" 'cd "+os.getcwd()+" && python sine.py'", shell=True)
    out = pickle.load(open('sine.pkl','rb'))
    return out

# Automatically determine the hostnames available on system using slurm resource manager
# This will have to be modified for other resource managers
hostnames = Popen(["scontrol","show","hostnames"],stdout=PIPE).communicate()[0]
hostnames = hostnames.split('\n')[0:-1]
host = os.environ['HOST'].split('.')[0]
#hostnames.remove(host) # Remove host to use as designated master if desired

# Create dictionary of lists of processor ids to use keyed by hostname
hosts = {}
for h in hostnames:
    hosts[h] = range(0,16,6) # create lists of processor numbers for each host
print 'host dictionary: ', hosts

# create data to be fitted
x = np.linspace(0, 15, 301)
np.random.seed(1000)
data = (5. * np.sin(2 * x - 0.1) * np.exp(-x*x*0.025) +
        np.random.normal(size=len(x), scale=0.2) )

# Create MATK object
p = matk(model=run_extern)

# Create parameters
p.add_par('amp', value=10, min=0.)
p.add_par('decay', value=0.1)
p.add_par('shift', value=0.0, min=-np.pi/2., max=np.pi/2.)
p.add_par('omega', value=3.0)

# Create observation names and set observation values
for i in range(len(data)):
    p.add_obs('obs'+str(i+1), value=data[i])

# Look at initial fit
init_vals = p.forward(workdir='initial',hostname=hosts.keys()[0],processor=0,reuse_dirs=True)
plt.plot(x,data, 'k+')
plt.plot(x,p.sim_values, 'r')
plt.title("Before Calibration")
plt.show(block=True)

# Calibrate parameters to data, results are printed to screen
p.lmfit(cpus=hosts,workdir='calib')

# Look at calibrated fit
plt.plot(x,data, 'k+')
plt.plot(x,p.sim_values, 'r')
plt.title("After Calibration")
```

```
plt.show()
```

Template file used by *pest_io.tpl_write*. Note the header **ptf %** and parameter locations indicated by **%** in the file.

```
ptf %
import numpy as np
import cPickle as pickle

# define objective function: returns the array to be minimized
def sine_decay():
    """ model decaying sine wave, subtract data"""
    amp = %amp%
    shift = %shift%
    omega = %omega%
    decay = %decay%

    x = np.linspace(0, 15, 301)
    model = amp * np.sin(x * omega + shift) * np.exp(-x*x*decay)

    obsnames = ['obs'+str(i) for i in range(1,model.shape[0]+1)]
    return dict(zip(obsnames,model))


if __name__== "__main__":
    out = sine_decay()
    pickle.dump(out,open('sine.pkl', 'wb'))
```

CHAPTER

FOUR

CLASS DOCUMENTATION

4.1 MATK

class matk.matk.**matk**(*model=''*, *model_args=None*, *model_kwargs=None*, *cpus=1*, *workdir_base=None*, *workdir=None*, *results_file=None*, *seed=None*, *sample_size=10*, *hosts={}*)

Class for Model Analysis ToolKit (MATK) module

Jac(*h=None*, *cpus=1*, *workdir_base=None*, *save=True*, *reuse_dirs=False*, *verbose=False*)

Numerical Jacobian calculation

Parameters **h** (*fl64 or ndarray(fl64)*) – Parameter increment, single value or array with npar values

Returns ndarray(fl64) – Jacobian matrix

MCMC(*nruns=10000*, *burn=1000*, *init_error_std=1.0*, *max_error_std=100.0*, *verbose=1*)

Perform Markov Chain Monte Carlo sampling using pymc package

Parameters

- **nruns** (*int*) – Number of MCMC iterations (samples)
- **burn** (*int*) – Number of initial samples to burn (discard)
- **verbose** (*int*) – verbosity of output
- **init_error_std** (*fl64*) – Initial standard deviation of residuals
- **max_error_std** (*fl64*) – Maximum standard deviation of residuals that will be considered

Returns pymc MCMC object

add_obs(*name*, *sim=None*, *weight=1.0*, *value=None*)

Add observation to problem

Parameters

- **name** (*str*) – Observation name
- **sim** (*fl64*) – Simulated value
- **weight** (*fl64*) – Observation weight
- **value** (*fl64*) – Value of observation

Returns Observation object

add_par(*name*, *value=None*, *vary=True*, *min=None*, *max=None*, *expr=None*, *discrete_vals=[]*, *discrete_counts=[]*, ***kwargs*)

Add parameter to problem

Parameters

- **name** (*str*) – Name of parameter
- **value** (*float*) – Initial parameter value
- **vary** (*bool*) – Whether parameter should be varied or not, currently only used with lmfit
- **min** (*float*) – Minimum bound
- **max** (*float*) – Maximum bound
- **expr** (*str*) – Mathematical expression to use to calculate parameter value
- **discrete_vals** (*[float]*) – list of values defining histogram bins
- **discrete_counts** (*[int]*) – list of counts associated with discrete_vals
- **kwargs** – keyword arguments passed to parameter class

calibrate(*cpus=1*, *maxiter=100*, *lambdax=0.001*, *minchange=1e-16*, *minlambdax=1e-06*, *verbose=False*, *workdir=None*, *reuse_dirs=False*, *h=1e-06*)

Calibrate MATK model using Levenberg-Marquardt algorithm based on original code written by Ernesto P. Adorio PhD. (UPDEPP at Clarkfield, Pampanga)

Parameters

- **cpus** (*int*) – Number of cpus to use
- **maxiter** (*int*) – Maximum number of iterations
- **lambdax** (*fl64*) – Initial Marquardt lambda
- **minchange** (*fl64*) – Minimum change between successive ChiSquares
- **minlambdax** (*fl4*) – Minimum lambda value
- **verbose** (*bool*) – If True, additional information written to screen during calibration

Returns best fit parameters found by routine

Returns best Sum of squares.

Returns covariance matrix

copy_sampleset(*oldname*, *newname=None*)

Copy sampleset

Parameters

- **oldname** (*str*) – Name of sampleset to copy
- **newname** (*str*) – Name of new sampleset

cpus

Set number of cpus to use for concurrent model evaluations

create_sampleset(*samples*, *name=None*, *responses=None*, *indices=None*, *index_start=1*)

Add sample set to problem

Parameters

- **name** (*str*) – Name of sample set
- **samples** (*list(fl64),ndarray(fl64)*) – Matrix of parameter samples with npar columns in order of matk.pars.keys()

- **responses** (*list(fl64),ndarray(fl64)*) – Matrix of associated responses with nobs columns in order matk.obs.keys() if observation exists (existence of observations is not required)
- **indices** (*list(int),ndarray(int)*) – Sample indices to use when creating working directories and output files

emcee(*lnprob=None*, *nwalkers=100*, *nsamples=500*, *burnin=50*, *pos0=None*)
Perform Markov Chain Monte Carlo sampling using emcee package

Parameters

- **lnprob** (*function*) – Function specifying the natural logarithm of the likelihood function
- **nwalkers** (*int*) – Number of random walkers
- **nsamples** (*int*) – Number of samples per walker
- **burnin** (*int*) – Number of "burn-in" samples per walker to be discarded
- **pos0** (*list*) – list of initial positions for the walkers

Returns numpy array containing samples

forward(*pardict=None*, *workdir=None*, *reuse_dirs=False*, *job_number=None*, *hostname=None*, *processor=None*)
Run MATK model using current values

Parameters

- **pardict** (*dict*) – Dictionary of parameter values keyed by parameter names
- **workdir** (*str*) – Name of directory where model will be run. It will be created if it does not exist
- **reuse_dirs** (*bool*) – If True and workdir exists, the model will reuse the directory
- **job_number** (*int*) – Sample id
- **hostname** (*str*) – Name of host to run job on, will be passed to MATK model as kwarg 'hostname'
- **processor** (*str or int*) – Processor id to run job on, will be passed to MATK model as kwarg 'processor'

Returns int – 0: Successful run, 1: workdir exists

levmar(*workdir=None*, *reuse_dirs=False*, *max_iter=1000*, *full_output=True*)
Calibrate MATK model using levmar package

Parameters

- **workdir** (*str*) – Name of directory where model will be run. It will be created if it does not exist
- **reuse_dirs** (*bool*) – If True and workdir exists, the model will reuse the directory
- **max_iter** (*int*) – Maximum number of iterations
- **full_output** – If True, additional output displayed during calibration

Returns levmar output

lhs(*name=None*, *siz=None*, *noCorrRestr=False*, *corrmat=None*, *seed=None*, *index_start=1*)
Draw lhs samples of parameter values from scipy.stats module distribution

Parameters

- **name** (*str*) – Name of sample set to be created
- **siz** (*int*) – Number of samples to generate, ignored if samples are provided
- **noCorrRestr** (*bool*) – If True, correlation structure is not enforced on sample, use if siz is less than number of parameters
- **corrmat** (*matrix*) – Correlation matrix
- **seed** (*int*) – Random seed to allow replication of samples
- **index_start** – Starting value for sample indices

Type int

Returns matrix – Parameter samples

lmfit(*maxfev=0*, *report_fit=True*, *cpus=1*, *epsfcn=None*, *xtol=1e-07*, *ftol=1e-07*, *workdir=None*, *verbose=False*, ***kwargs*)

Calibrate MATK model using lmfit package

Parameters

- **maxfev** (*int*) – Max number of function evaluations, if 0, 100*(npars+1) will be used
- **report_fit** (*bool*) – If True, parameter statistics and correlations are printed to the screen
- **cpus** (*int*) – Number of cpus to use for concurrent simulations during jacobian approximation
- **epsfcn** (*float or lst[float]*) – jacobian finite difference approximation increment (single float of list of npar floats)
- **xtol** (*float*) – Relative error in approximate solution
- **ftol** (*float*) – Relative error in the desired sum of squares
- **workdir** (*str*) – Name of directory to use for model runs, calibrated parameters will be run there after calibration
- **verbose** (*bool*) – If true, print diagnostic information to the screen

Returns lmfit minimizer object

Additional keyword argments will be passed to scipy leastsq function: http://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.optimize.leastsq.html

make_workdir(*workdir=None*, *reuse_dirs=False*)

Create a working directory

Parameters

- **workdir** (*str*) – Name of directory where model will be run. It will be created if it does not exist
- **reuse_dirs** (*bool*) – If True and workdir exists, the model will reuse the directory

Returns int – 0: Successful run, 1: workdir exists

minimize(*method='SLSQP'*, *maxiter=100*, *workdir=None*, *bounds=()*, *constraints=()*, *options={'eps': 0.001}*, *save_evals=False*)

Minimize a scalar function of one or more variables

Parameters

- **maxiter** (*int*) – Max number of iterations

- **workdir** (*str*) – Name of directory to use for model runs, calibrated parameters will be run there after calibration

Returns OptimizeResult; if save_evals=True, also returns a MATK sampleset of calibration function evaluation parameters and responses

model
Python function that runs model

model_args
Tuple of extra arguments to MATK model expected to come after parameter dictionary

model_kwargs
Dictionary of extra keyword arguments to MATK model expected to come after parameter dictionary and model_args

nomvalues
Nominal parameter values used in info gap analyses

obsnames
Get observation names

obsvalues
Observation values

obsweights
Get observation names

pardist_pars
Get parameters needed by parameter distributions

pardists
Get parameter probabilistic distributions

parmaxs
Get parameter lower bounds

parmins
Get parameter lower bounds

parnames
Get parameter names

parstudy (*name=None*, *nvals=2*)
Generate parameter study samples

Parameters

- **name** (*str*) – Name of sample set to be created
- **outfile** (*str*) – Name of file where samples will be written. If outfile=None, no file is written.
- **nvals** (*int or list(int)*) – number of values for each parameter

Returns ndarray(fl64) – Array of samples

parvalues
Parameter values

read_sampleset (*file*, *name=None*)
Read MATK output file and assemble corresponding sampleset with responses.

Parameters

- **name** (*str*) – Name of sample set
- **file** (*str*) – Path to MATK output file

residuals
Get least squares values

results_file
Set the name of the results_file for parallel runs

seed
Set the seed for random sampling

simdict
Simulated values :returns: lst(fl64) – simulated values in order of matk.obs.keys()

simvalues
Simulated values :returns: lst(fl64) – simulated values in order of matk.obs.keys()

ssr
Sum of squared residuals

workdir
Set the base name for parallel working directories

workdir_base
Set the base name for parallel working directories

workdir_index
Set the working directory index for parallel runs

4.2 Parameter

class matk.parameter.**Parameter**(*name*, *value=None*, *vary=True*, *min=None*, *max=None*, *expr=None*, *nominal=None*, *discrete_vals=[]*, *discrete_counts=[]*, ***kwargs*)
MATK parameter class

dist
Probabilistic distribution of parameter belonging to scipy.stats module

dist_pars
Distribution parameters required by self.dist e.g. if dist == uniform, dist_pars = (min,max-min) if dist == norm, dist_pars = (mean,stdev))

expr
Mathematical expression to use to evaluate value

nominal
Nominal parameter value, used in info gap decision analyses

setup_bounds()
set up Minuit-style internal/external parameter transformation of min/max bounds.

returns internal value for parameter from self.value (which holds the external, user-expected value). This internal values should actually be used in a fit....

As a side-effect, this also defines the self.from_internal method used to re-calculate self.value from the internal value, applying the inverse Minuit-style transformation. This method should be called prior to passing a Parameter to the user-defined objective function.

This code borrows heavily from JJ Helmus' leastsqbound.py

value
Parameter value

vary
Boolean indicating whether or not to vary parameter

4.3 Observation

class matk.observation.**Observation**(*name*, *sim=None*, *weight=1.0*, *value=None*)
MATK observation class

name
Observation name

residual
Observation value minus simulated value

sim
Simulated value generated by MATK model

value
Observation value

weight
Weight to apply to simulated values

4.4 SampleSet

class matk.sampleset.**SampleSet**(*name*, *samples*, *parent*, *index_start=1*, ***kwargs*)
MATK SampleSet class - Stores information related to a sample including parameter samples, associated responses, and sample indices

corr(*type='pearson'*, *plot=False*, *printout=True*, *plotvals=True*, *figsize=None*, *title=None*)
Calculate correlation coefficients of parameters and responses

Parameters

- **type** (*str*) – Type of correlation coefficient (pearson by default, spearman also avaialable)
- **plot** (*bool*) – If True, plot correlation matrix
- **printout** (*bool*) – If True, print correlation matrix with row and column headings
- **plotvals** (*bool*) – If True, print correlation coefficients on plot matrix
- **figsize** (*tuple(fl64,fl64)*) – Width and height of figure in inches
- **title** (*str*) – Title of plot

Returns ndarray(fl64) – Correlation coefficients

index_start
Starting integer value for sample indices

indices
Array of sample indices

main_effects()
For each parameter, compile array of main effects.

name
Sample set name

obsnames
Array of observation names

panels (*type='pearson'*, *alpha=0.2*, *figsize=None*, *title=None*, *tight=False*, *symbol='.'*, *fontsize=None*, *corrfontsize=None*, *ms=5*, *mins=None*, *maxs=None*, *frequency=False*, *bins=10*, *ylim=None*, *labels=[]*, *filename=None*, *xticks=2*, *yticks=2*)
Plot histograms, scatterplots, and correlation coefficients in paired matrix

Parameters

- **type** (*str*) – Type of correlation coefficient (pearson by default, spearman also avaialable)
- **alpha** (*float*) – Histogram color shading
- **figsize** (*tuple(fl64,fl64)*) – Width and height of figure in inches
- **title** (*str*) – Title of plot
- **tight** (*bool*) – Use matplotlib tight layout
- **symbol** (*str*) – matplotlib symbol for scatterplots
- **fontsize** (*fl64*) – Size of font for axis labels
- **corrfontsize** (*fl64*) – Size of font for correlation coefficients
- **ms** (*fl64*) – Scatterplot marker size
- **frequency** (*bool*) – If True, the first element of the return tuple will be the counts normalized by the length of data, i.e., n/len(x)
- **bins** (*int*) – Number of bins in histograms
- **ylim** (*tuples - 2 element tuples with y limits for histograms*) – y-axis limits for histograms.
- **labels** (*lst(str)*) – Names to use instead of parameter names in plot
- **filename** (*int*) – Name of file to save plot. File ending determines plot type (pdf, png, ps, eps, etc.). Plot types available depends on the matplotlib backend in use on the system. Plot will not be displayed.
- **xticks** – Number of ticks along x axes
- **yticks** – Number of ticks along y axes

pardict (*index*)
Get parameter dictionary for sample with specified index

Parameters **index** (*int*) – Sample index

Returns dict(fl64)

parnames
Array of observation names

rank_parameter_frequencies ()
Yields a printout of parameter value frequencies in the sample set

returns An array of tuples, each containing the parameter name tagged as min or max and a second tuple containing the parameter value and the frequency of its appearance in the sample set.

recarray
Structured (record) array of samples

run(*cpus=1*, *workdir_base=None*, *save=True*, *reuse_dirs=False*, *outfile=None*, *logfile=None*, *verbose=True*, *hosts={}*)
Run model using values in samples for parameter values If samples are not specified, LHS samples are produced

Parameters

- **cpus** (*int,dict(lst)*) – number of cpus; alternatively, dictionary of lists of processor ids keyed by hostnames to run models on (i.e. on a cluster); hostname provided as kwarg to model (hostname=<hostname>); processor id provided as kwarg to model (processor=<processor id>)
- **workdir_base** (*str*) – Base name for model run folders, run index is appended to workdir_base
- **save** (*bool*) – If True, model files and folders will not be deleted during parallel model execution
- **reuse_dirs** (*bool*) – Will use existing directories if True, will return an error if False and directory exists
- **outfile** (*str*) – File to write results to
- **logfile** (*str*) – File to write details of run to during execution
- **hosts** (*lst(str)*) – Option deprecated, use cpus instead

Returns tuple(ndarray(fl64),ndarray(fl64)) - (Matrix of responses from sampled model runs siz rows by npar columns, Parameter samples, same as input samples if provided)

savetxt(*outfile*)
Save sampleset to file

Parameters **outfile** (*str*) – Name of file where sampleset will be written

sse
Sum of squared errors (sse) for all samples

subset(*boolfcn*, *obs*, **args*, ***kwargs*)
Collect samples based on response values, remove all others

Parameters

- **boofcn** – Function that returns true for samples to keep and false for samples to remove
- **obs** (*str*) – Name of response to apply boolfcn to
- **args** – Additional arguments to add to boolfcn
- **kwargs** – Keyword arguments to add to boolfcn

matk.sampleset.**hist**(*rc*, *ncols=4*, *figsize=None*, *alpha=0.2*, *title=None*, *tight=False*, *mins=None*, *maxs=None*, *frequency=False*, *bins=10*, *ylim=None*, *printout=True*, *labels=[]*, *filename=None*, *fontsize=None*, *xticks=3*)
Plot histograms of dataset

Parameters

- **ncols** (*int*) – Number of columns in plot matrix
- **figsize** (*tuple(fl64,fl64)*) – Width and height of figure in inches
- **alpha** (*float*) – Histogram color shading
- **title** (*str*) – Title of plot
- **tight** (*bool*) – Use matplotlib tight layout

- **mins** (*lst(fl64)*) – Minimum values of recarray fields
- **maxs** (*lst(fl64)*) – Maximum values of recarray fields
- **frequency** (*bool*) – If True, the first element of the return tuple will be the counts normalized by the length of data, i.e., n/len(x)
- **bins** (*int or lst(lst(int))*) – If an integer is given, bins + 1 bin edges are returned. Unequally spaced bins are supported if bins is a list of sequences for each histogram.
- **ylim** (*tuples - 2 element tuple with y limits for histograms*) – y-axis limits for histograms.
- **labels** (*lst(str)*) – Names to use instead of parameter names in plot
- **filename** (*str*) – Name of file to save plot. File ending determines plot type (pdf, png, ps, eps, etc.). Plot types available depends on the matplotlib backend in use on the system. Plot will not be displayed.
- **fontsize** (*fl64*) – Size of font
- **xticks** (*int*) – Number of ticks on xaxes

Returns dict(lst(int),lst(fl64)) - dictionary of histogram data (counts,bins) keyed by name

matk.sampleset.**corr**(*rc1*, *rc2*, *type='pearson'*, *plot=False*, *printout=True*, *plotvals=True*, *figsize=None*, *title=None*)

Calculate correlation coefficients of parameters and responses

Parameters

- **rc1** – Data
- **rc2** – Data
- **type** (*str*) – Type of correlation coefficient (pearson by default, spearman also avaialable)
- **plot** (*bool*) – If True, plot correlation matrix
- **printout** (*bool*) – If True, print correlation matrix with row and column headings
- **plotvals** (*bool*) – If True, print correlation coefficients on plot matrix
- **figsize** (*tuple(fl64,fl64)*) – Width and height of figure in inches
- **title** (*str*) – Title of plot

Returns ndarray(fl64) – Correlation coefficients

4.5 Parameter

Utilities to handle reading and writing PEST files

matk.pest_io.**tpl_write**(*pardict*, *f*, *outflnm*)

Write model input file using PEST template file

Parameters

- **pardict** (*dict*) – Dictionary of parameter values
- **f** (*str or file handle*) – File handle or file name of PEST template file
- **outflnm** (*str*) – Name of model input file to be written

matk.pest_io.**read_par_files**(**files*)

Read in one or more PEST parameter files

Parameters

- **files** (*str*) – Strings of file names, glob characters are supported
- **output_format** (*str ('numpy_array' or 'recarray')*) – Format to return in

Returns parameter names list and numpy array of parameters or recarray of parameters

CHAPTER

FIVE

COPYRIGHT AND LICENSE

CHAPTER

SIX

OTHER USEFUL PYTHON MODULES:

PYEMU: Jeremy White's linear-based computer model uncertainty analysis python module; used in *Linear Analysis of Calibration Using PYEMU* example.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

m

S

T

V

W