PUQ Documentation

Release 2.2.2

Martin Hunt

February 13, 2014

CONTENTS

CHAPTER

ONE

CONTENTS:

1.1 Overview

PUQ is a general purpose parameter sweep framework, which can perform uncertainty quantification (UQ), do sensitivity analysis, and build response surfaces.

You provide a program that takes parameters as arguments. This is called the *Test Program* or simulation. It can be written in any language as long as it can take command line arguments. PUQ treats the test program like a "black box". It need not know anything about it except how to send it inputs and how to get the outputs.

Then you tell the framework details about the input parameters (name, mean, deviation, PDF, etc). You also need to either have the test program write data in a standard format readable by PUQ, or you need to supply a bit of code to PUQ that can process the output of the test program.

Finally you must tell PUQ what kind of sweep to perform (Smolyak Sparse Grid, Monte Carlo, etc) and what host to run it on. This might change in the future as a more intelligent UQ framework should be able to automatically select the correct method and find a host to run the sweep.

The framework then launches all the required jobs, providing the proper parameters to your test program.

If there are any problems as the jobs run (such as exceeding the allocated walltime), individual jobs can be modified and restarted.

The framework keeps track of the outputs and, once all jobs have completed, collects the outputs into a single HDF5 file. Then analysis can be performed on the collected data. That analysis can be repeated as necessary, without rerunning the jobs. If more data is required, a new sweep can be performed, adding to the existing data.

1.1.1 Goals

- Easy to use. Tools that are hard to use don't get used, or get used improperly.
- Expandable. Should be able to handle any necessary UQ or optimization methods.

1.1.2 In Progress

- Integration with HUBzero and nanoHUB platforms.

1.1.3 Future Plans

- Adaptive response surface builder. Chooses the best method without user intervention. Can refine until it meets a specific accuracy goal.

- Better job control.

1.2 User's Guide and Tutorial

1.2.1 Getting Started

PUQ works on Linux and MacOSX.

nanoHUB Workspaces

If you have access to a nanoHUB workspace, PUQ is already installed. If 'puq' is not in your path, use the 'use' command to load it.

```
~> which puq
~> use -e puq-2.2.1
~> which puq
/apps/share64/debian7/puq/puq-2.2.1/build-debian7/bin/puq
```

Type 'use' without arguments to see a list of packages. You should use the latest version of PUQ.

Installing from PyPI

PUQ uses distutils, which is the default way of installing python modules. Simply do:

```
pip install puq
```

The installer might ask you to install some other packages first. Just use 'pip' to install them. For the Mac, we have very successfully used Anaconda, a free Python distribution containing all the required components to install PUQ. https://store.continuum.io/cshop/anaconda/

Building and Installing from Sources

If you are a developer and have a local copy of the sources,

To install in your home directory, use:

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

To install for all users on Unix/Linux or Mac:

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

For more information: https://pypi.python.org/pypi/puq/ and https://github.com/martin-hunt/puq

---

To test that everything is set up properly, just try the **puq** command without any arguments. Correct output should look something like this:

```
> puq

This program does parameter sweeps for uncertainty
quantification and optimization.
```

```
Usage: puq [-q|-v] [start|stop|status|resume|analyze|plot]
Options:
-q                          Quiet.  Useful for scripts.

-v                          Verbose. Show even more information.

-k                          Keep temporary directories.

start script[.py] [args]    Start PUQ using script 'script'

status [id]                 Returns the number of jobs remaining in the sweep.

resume [id]                 Resumes execution of sweep 'id'.

analyze [options] [id]      Does post-processing.

extend [id]                 Extend a sweep by adding additional jobs.

plot [options] [id]         Plots output pdf(s) or response surface. Type
'puq plot -h' for options.

read                        Read a parameter or PDF from a python input script,
                            json file, csv file, or URI.  Visualize, modify, and save it.

strip                       PUQ keeps all stdout from every job.  This command
                            removes output lines not containing data and
                            repacks the HDF5 file, reducing its size.

dump                        Dumps output data in CSV format.
```

Requirements

The Control Script

To get started, you will need to write a control script. This script is actually a small Python function. Other parameter sweep frameworks usually use text file or XML for this. However by using Python, we can have much more flexibility. You will not have to know much about Python to create a simple control script.

The control script must declare three objects:

```
1. At least one parameter, which will be swept by the run.
2. A test program which takes the parameter(s) as a command line argument.
3. A host, which right now is either the local host running interactively,
or the PBS batch scheduler, running locally.
```

These three objects are combined to form a Sweep object which the UQ system will use. Each of the above objects will be explained in detail.

Good examples of control script are found in puq/examples.

Parameters

Parameters represent inputs to our test program. Parameters are values that will be changing in different runs of our test program.

For scaling sweeps or optimization runs, the parameter represents values that can be changed for different runs.

For uncertainty quantification, parameters are random variables which can be discrete or continuous.

A *discrete* parameter has a finite number of values.

Continuous parameters can be *aleatoric* or *epistemic*.

- *Aleatoric* parameters are variables that contain inherent randomness that cannot be reduced by better data. They are specified by a PDF (Probability Density Function).
- *Epistemic* parameters are values that are uncertain due to a lack of data or understanding. You may choose to represent this uncertainty about their value as a PDF and treat them as aleatoric. However, there are other ways to model uncertainty of epistemic parameters which will be supported in the future.

Parameters are represented as PDFs. They might be approximated by a standard PDF, such as uniform, or gaussian (normal). Or they can be created from the results of experiments or measurements.

See Also:

Parameters

Test Program

The test program is a function (typically a simulation) that takes inputs and generates outputs. PUQ requires that the test program take numerical parameters on the command line. If you have a test program that reads parameters some other way, it may be necessary to create a wrapper script for it.

Host

Jobs can be run sequentially or in parallel on an interactive scheduler. This would be the typical desktop or workstation. PBS, TORQUE, and HUBzero submit are also supported for use in cluster environments.

1.2.2 Monte Carlo Sampling

This is a simple example of using PUQ with a test program written in Python.

Rather than use a complex simulation as a test program, we will use the *Rosenbrock* function. This function is well understood and widely used as an example for optimization and UQ. Because it can be solved analytically, it is useful as a test case for our software.

The *Rosenbrock* function is $f(x, y) = 100(y - x^2)^2 + (1 - x)^2$

And here is our python test program implementing it.

```
#!/usr/bin/env python

import optparse
from puqutil import dump_hdf5

usage = "usage: %prog --x x --y y"
parser = optparse.OptionParser(usage)
parser.add_option("--x", type=float)
parser.add_option("--y", type=float)
(options, args) = parser.parse_args()
x = options.x
y = options.y

z = 100*(y-x**2)**2 + (1-x)**2
```

```
15
16 dump_hdf5('z', z)
```

Most of the lines are concerned with parsing the command line. By default, PUQ expects test programs to take command line parameters in the format *–varname=value*. This is convenient for Python programs (using the optparse module) and C/C++ (using getopt). For existing test programs, you can easily instruct PUQ to pass parameters in another format. See *Making PUQ Work With Your Code* and puq/examples/matlab.

Line 16 calls `dump_hdf5()` with our output. `dump_hdf5()` takes three arguments, a name, a value, and an optional description. It formats the data so that PUQ can easily recognize it in standard output. It is placed in the HDF5 output file under 'output/data/varname' ('output/data/z' for our example). If we run our test program, we get:

```
~/puq/examples/rosen> ./rosen_prog.py --x=0 --y=1
HDF5:{'name': 'z', 'value': 101.0, 'desc': ''}:5FDH
```

So we have a working test program. We will limit x and y to [-2,2] and try to calculate an output PDF.

The next step is to write a control script to set up the sweep.

Here is what we will use

```
1  from puq import *
2
3  def run(num=20):
4      # Declare our parameters here. Both are uniform on [-2, 2]
5      p1 = UniformParameter('x', 'x', min=-2, max=2)
6      p2 = UniformParameter('y', 'y', min=-2, max=2)
7
8      # Create a host
9      host = InteractiveHost()
10
11     # Declare a UQ method.
12     uq = MonteCarlo([p1, p2], num=num)
13
14     # Our test program
15     prog = TestProgram(exe='./rosen_prog.py --x=$x --y=$y',
16         desc='Rosenbrock Function')
17
18     return Sweep(uq, host, prog)
```

Every control script should look basically like this. There needs to be:

```
from puq import *
```

to import all the required PUQ code. There must be a function `run()` defined that returns a `Sweep` object. And the `Sweep` object is made up of a UQ method, a host, and a test program.

The UQ method is selected in the line:

```
uq = MonteCarlo([x,y], num=num)
```

This tells PUQ to use the Monte Carlo method, with two parameters 'x', and 'y'. It will generate *num* random numbers each for 'x' and 'y', using a uniform distribution over [-2,2].

Now we are ready to run. We do this with 'puq start rosen_mc'. Because control scripts are python scripts, the '.py' is optional. Here are three runs, all doing 20 samples.

```
~/puq/examples/rosen> puq start rosen_mc
Saving run to sweep_52410436.hdf5

Processing <HDF5 dataset "z": shape (20,), type "<f8">
```

```
Mean   = 463.467551273
StdDev = 499.225504146

~/puq/examples/rosen> puq start rosen_mc 20
Saving run to sweep_52410475.hdf5

Processing <HDF5 dataset "z": shape (20,), type "<f8">
Mean   = 266.371029691
StdDev = 362.362400696

~/puq/examples/rosen> puq start -f mc.hdf5 rosen_mc 20
Saving run to mc.hdf5

Processing <HDF5 dataset "z": shape (20,), type "<f8">
Mean   = 322.364238869
StdDev = 246.016024931
```

As you can see, 20 samples is not enough to get an accurate mean. To get Monte Carlo Sampling to converge on a more accurate value, you would need to run thousands of samples. That is fine in a simple case like the Rosenbrock function, but a complex simulation might take days or weeks per sample.

To plot the response surface:

```
~/puq/examples/rosen> puq plot -r mc.hdf5
```

Extending the Sample Size

In the example above, only 20 samples were used. If you wish to use more samples, you can modify your control script and rerun it from the start. However, if you do it this way, you are wasting the work that has already been done running the 20 jobs. That might be fine for a trivial example like this, but not for a complex simulation that ran for days to complete 20 jobs.

Fortunately you can easily add more samples to an existing Monte Carlo run:

```
~/puq/examples/rosen> puq extend --num 80 mc.hdf5
    Extending mc.hdf5 using MonteCarlo

    Processing <HDF5 dataset "z": shape (100,), type "<f8">
    Mean   = 377.988520682
    StdDev = 599.985509368
```

The response surface now looks more accurate although the mean is still not close to the real mean. To plot the PDF of z, you can do:

```
~/puq/examples/rosen> puq plot mc.hdf5
```

1.2.3 Latin Hypercube Sampling

Latin Hypercube Sampling is an improvement over Monte Carlo Sampling. In one dimension, it generates numbers that are evenly spaced in probability. For multiple dimensions, the numbers are randomly paired, so there is more randomness in the output. However it avoids clusters and generally outperforms Monte Carlo.

If you looked at the previous section on *Monte Carlo Sampling*, you will need to make just a minor change to your control script to use Latin Hypercube Sampling.

Change the line that says:

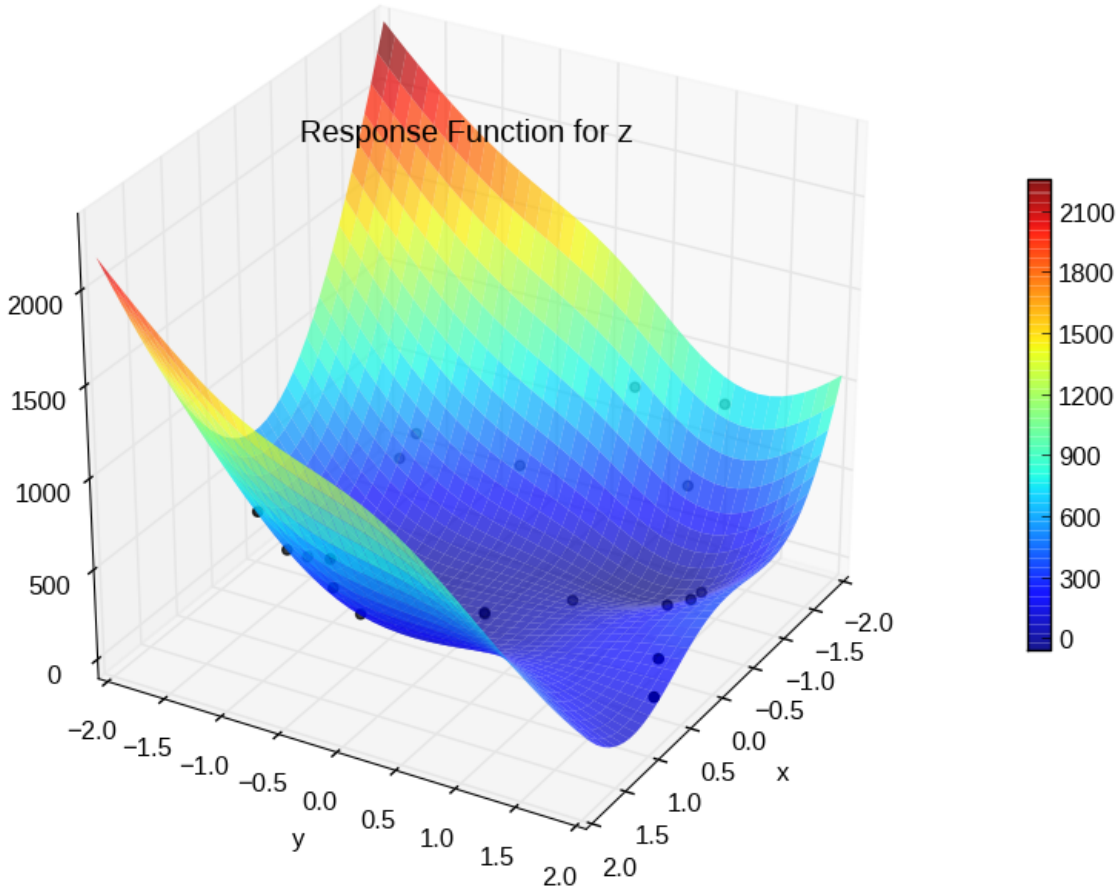


Figure 1.1: Scatter plot for Rosenbrock function using Monte Carlo with 20 samples.

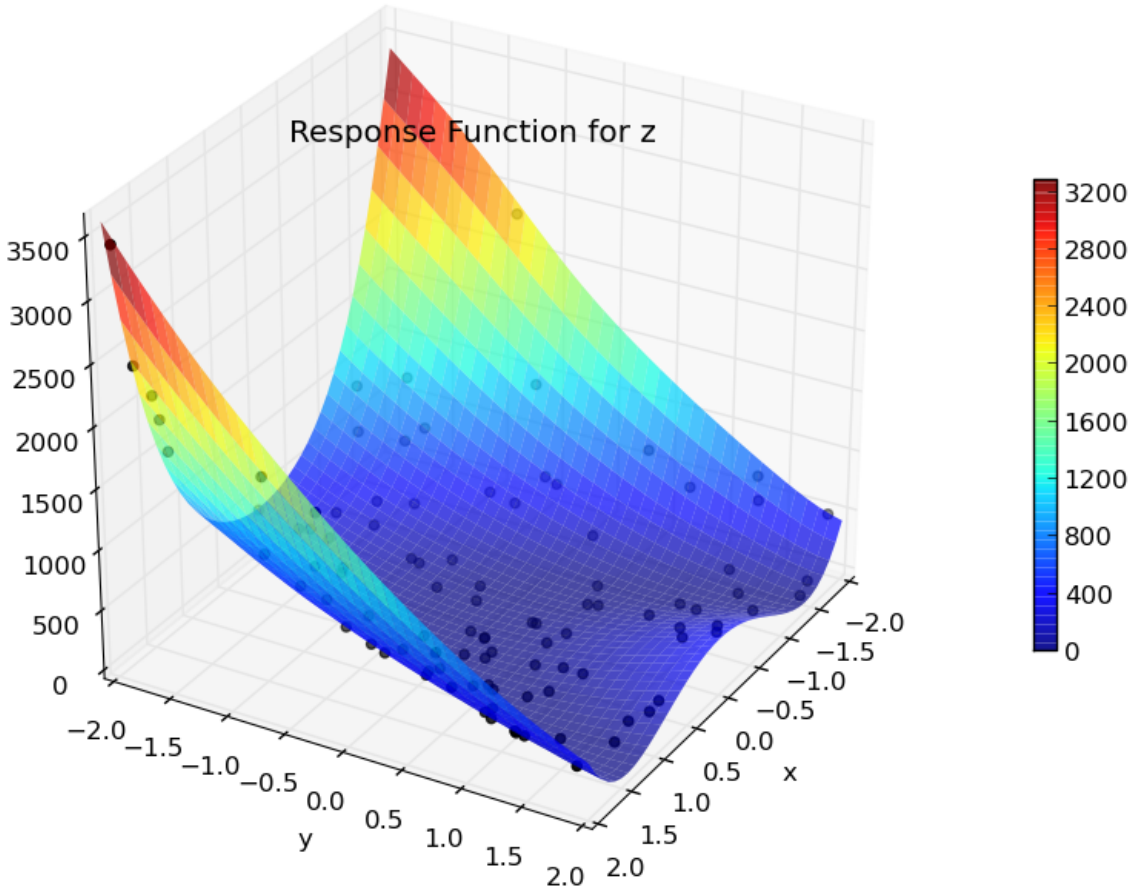


Figure 1.2: Scatter plot for Rosenbrock function using Monte Carlo with 100 samples.

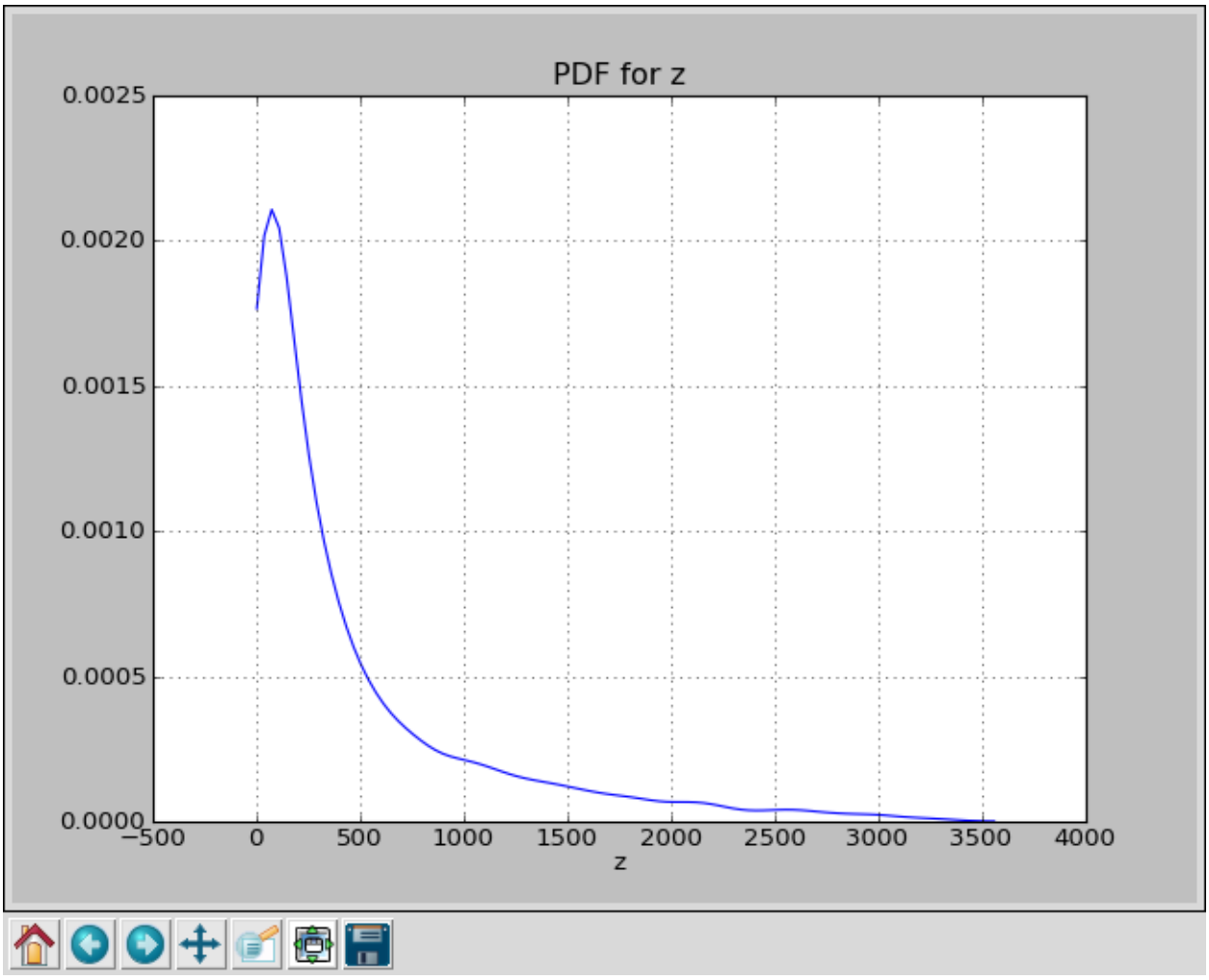


Figure 1.3: PDF for Rosenbrock function using Monte Carlo with 100 samples.

```
uq = MonteCarlo([x,y], num=num)
```

to:

```
uq = LHS([x,y], num=num)
```

Or use 'rosen_lhs.py' in puq/examples/rosen.

```
~/puq/examples/rosen> puq start -f rosen_lhs.hdf5 rosen_lhs
Saving run to rosen_lhs.hdf5

Processing <HDF5 dataset "z": shape (20,), type "<f8">
Mean   = 516.828
StdDev = 634.386086052

~/puq/examples/rosen> puq plot -r rosen_lhs.hdf5
```

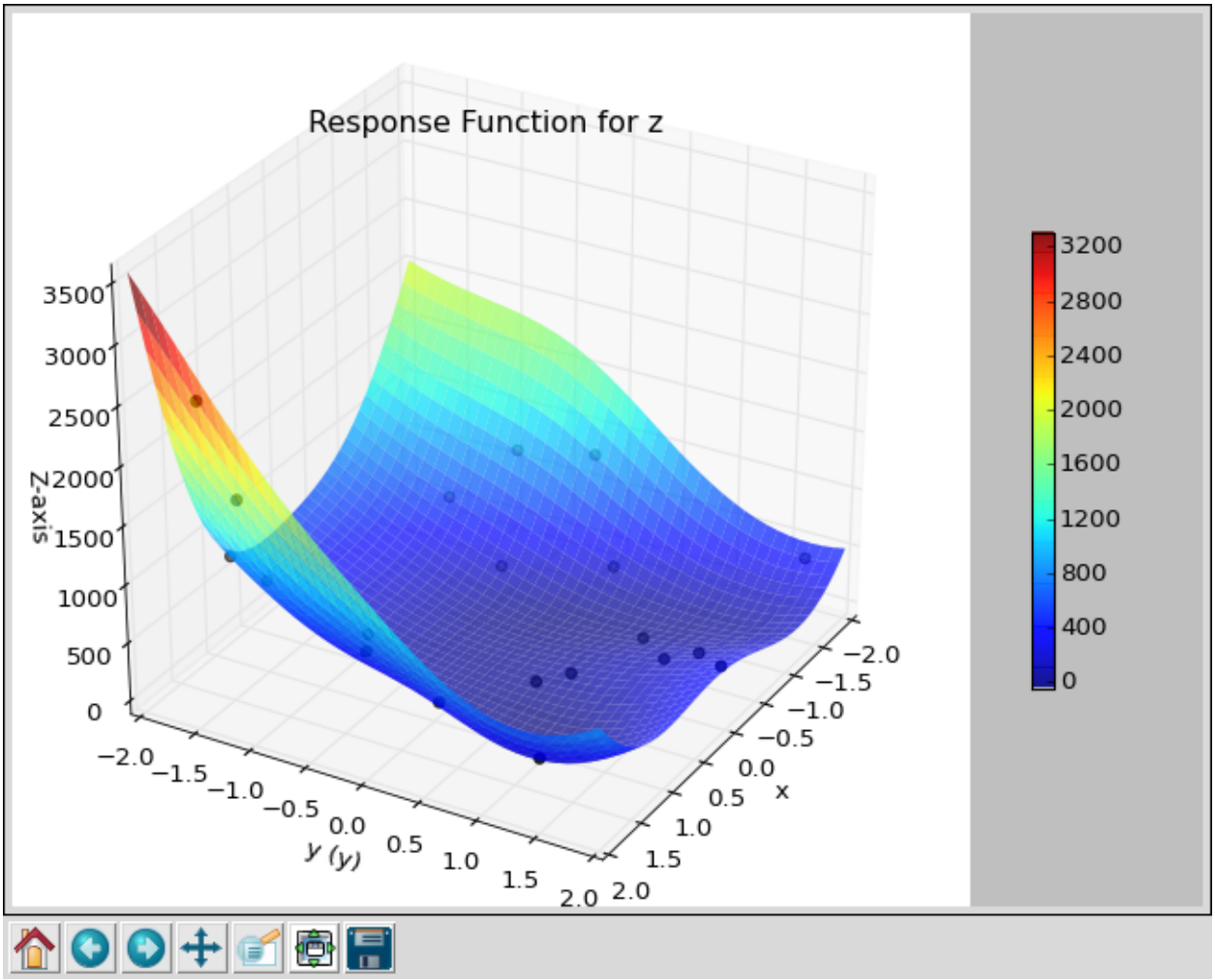


Figure 1.4: Scatter plot for Rosenbrock function using LHS with 20 samples.

```
~/puq/examples/rosen> puq extend rosen_lhs.hdf5
    Extending rosen_lhs.hdf5 using LHS
    Extending Descriptive Sampling run to 60 samples.
```

```
Processing <HDF5 dataset "z": shape (60,), type "<f8">
Mean   = 495.208576132
StdDev = 605.837656938
~/puq/examples/rosen> puq plot -r rosen_lhs.hdf5
plotting z
~/puq/examples/rosen> puq plot rosen_lhs.hdf5
plotting PDF for z
```

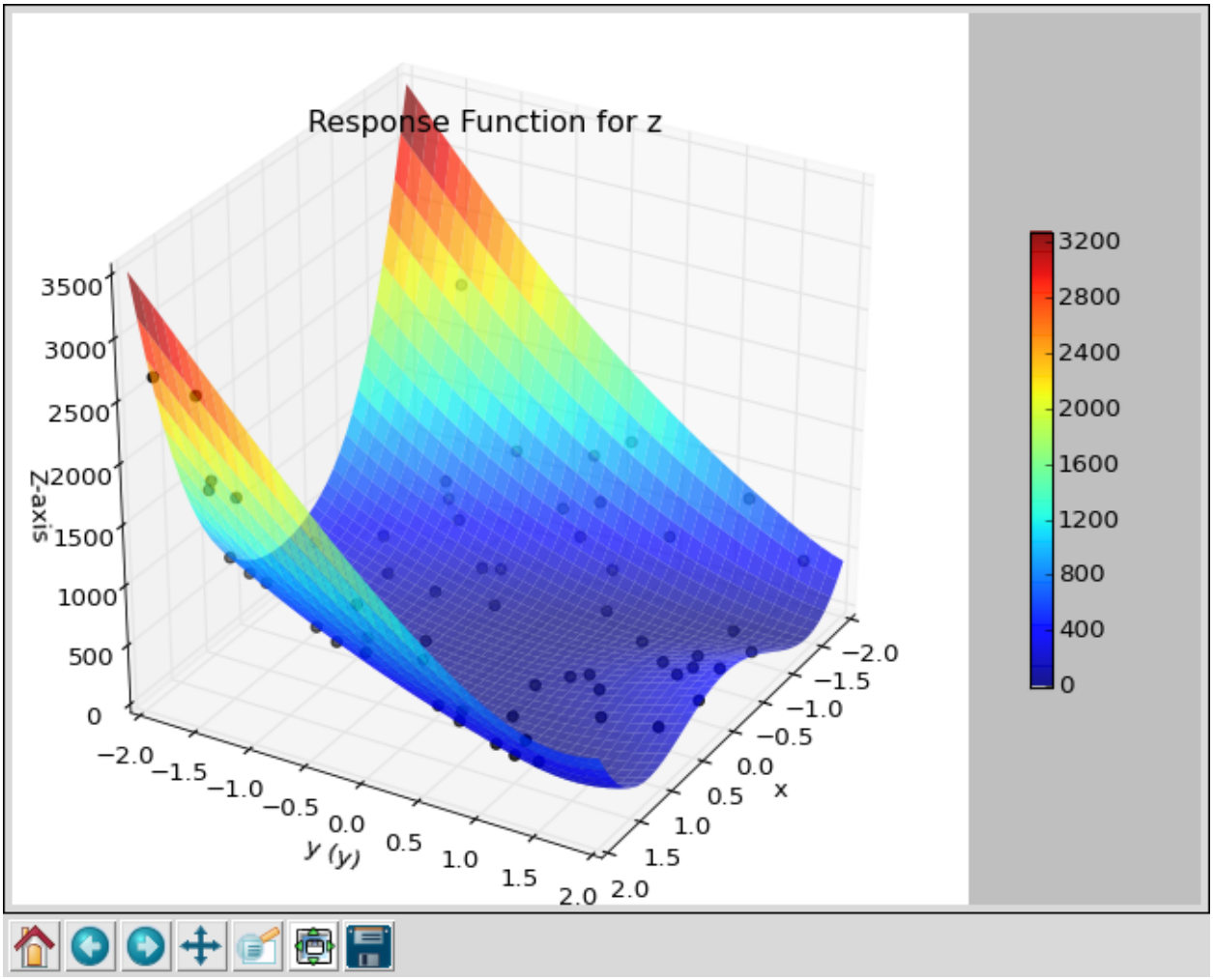


Figure 1.5: Scatter plot for Rosenbrock function using LHS with 60 samples.

1.2.4 Smolyak Sparse Grid Algorithm

In the last two sections, we have used *Monte Carlo Sampling* and *Latin Hypercube Sampling* on the Rosenbrock function. Both of those algorithms suffered from some common problems; they were slow to converge on a correct answer and their results were non-deterministic, varying from run to run due to the inherent randomness.

The Smolyak algorithm is a significant improvement. It uses a smaller number of runs to generate a response surface.

To move from Monte Carlo to Latin Hypercube sampling, we had to change a single line in our control script. To use Smolyak, we simply change that same line:

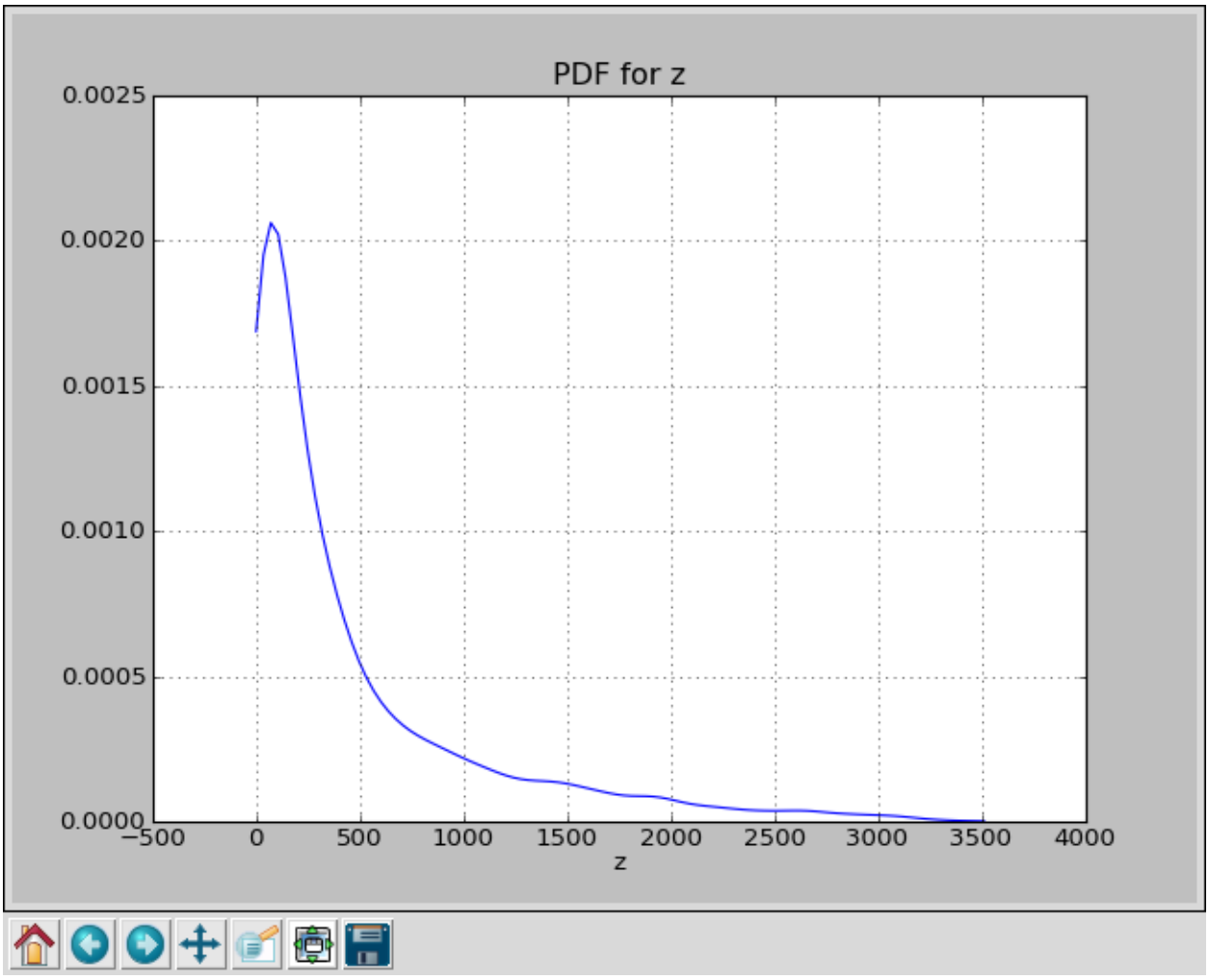


Figure 1.6: PDF for Rosenbrock function using LHS with 60 samples.

```
uq = LHS([x,y], num=num)
```

to:

```
uq = Smolyak([x,y], level=2)
```

Or use 'rosen.py' in puq/examples/rosen.:

```
~/puq/examples/rosen> puq start rosen
Saving run to sweep_140682290.hdf5

Processing <HDF5 dataset "z": shape (13,), type "<f8">
        Surface   = 301.0*x**2 - 2.0*x + 300.0*y**2 - 266.667*y - 345.667
        RMSE      = 7.40e+02 (2.05e+01 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    3.9402e+03    5.2374e+03
y    2.0828e+03    1.8510e+03
```

The puq program first created a file using the unique id it generated. That file is sweep_140682290.hdf5. All information about this run is placed in there. The puq program next ran 13 jobs, using the parameters calculated by the Smolyak method. Then it calculated a response surface.

For the Smolyak method, the response surface should ge a perfect fit if the polynomial degree of the function is less than or equal to the Smolyak level parameter. For our example, we used a level 2 Smolyak run to approzimate a The Rosenbrock function, which is 4th degree. Because of this, the response surface does not exactly fit the data points. The RMSE is 20.5%.

PUQ also calculates the sensitivity of the parameters using the Elementary Effects Method

We can plot the response surface and pdf:

```
~/puq/examples/rosen> puq plot -r sweep_140682290.hdf5
plotting PDF for z
```

And if you look at the response surface, you see the actual data as blue dots. Many of them are not on the response surface, indicating a bad fit.

For comparison, here is the expected rosenbrock function.

Next, if we bring up an editor and change the Smolyak level first to 3 then to 4, or do 'puq extend' twice, we get the following.

```
 ~/puq/examples/rosen> puq extend sweep_140682290.hdf5
Extending sweep_140682290.hdf5 Using Smolyak
Extending Smolyak to level 3

Processing <HDF5 dataset "z": shape (29,), type "<f8">
        Surface   = -200.0*x**2*y + 343.857*x**2 - 2.0*x + 100.0*y**2 - 136.143
        RMSE      = 2.40e+02 (6.66e+00 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    4.8307e+03    6.5106e+03
y    2.0022e+03    1.7623e+03
```

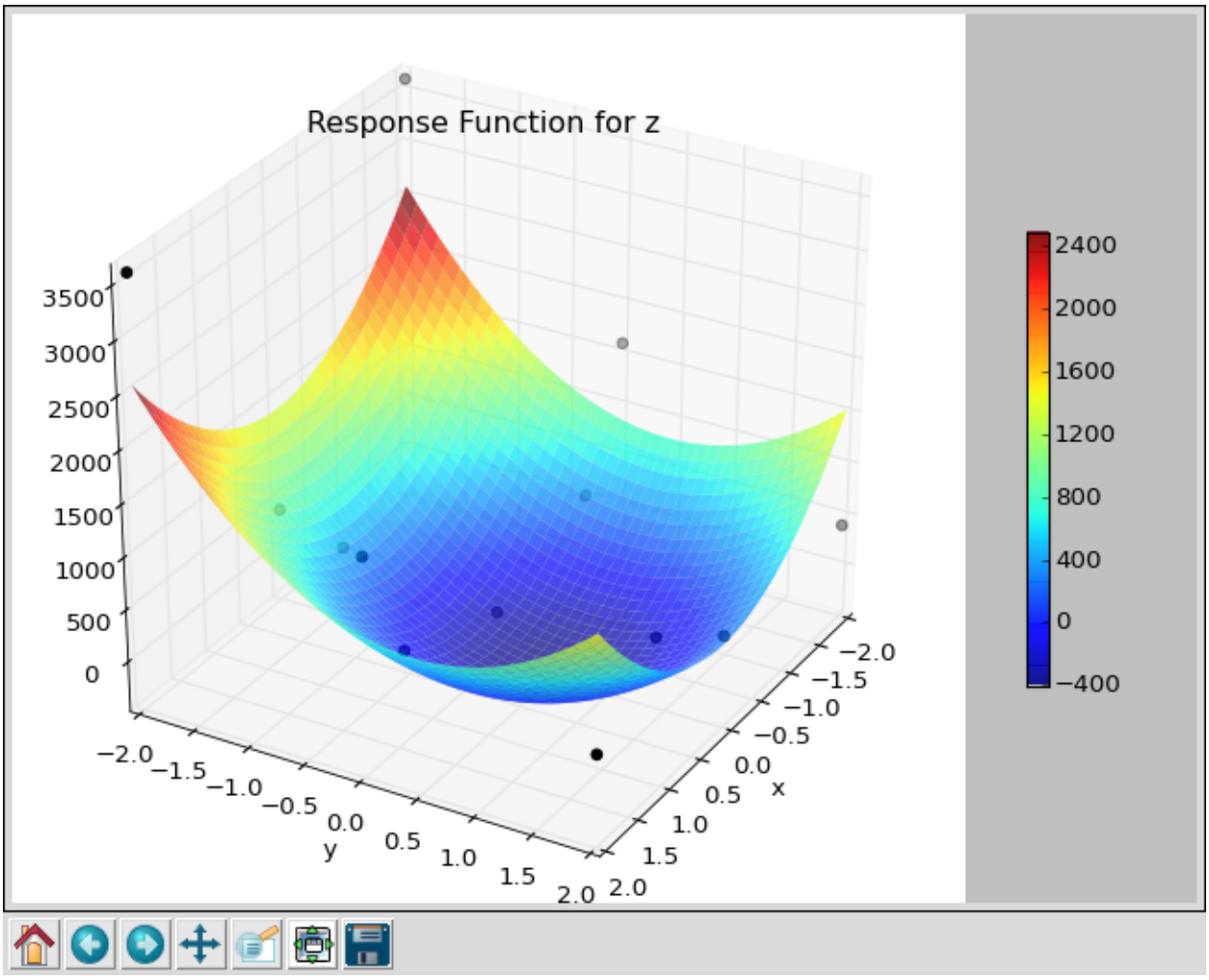


Figure 1.7: **This is the output from a level-2 Smolyak, which required 13 jobs.**

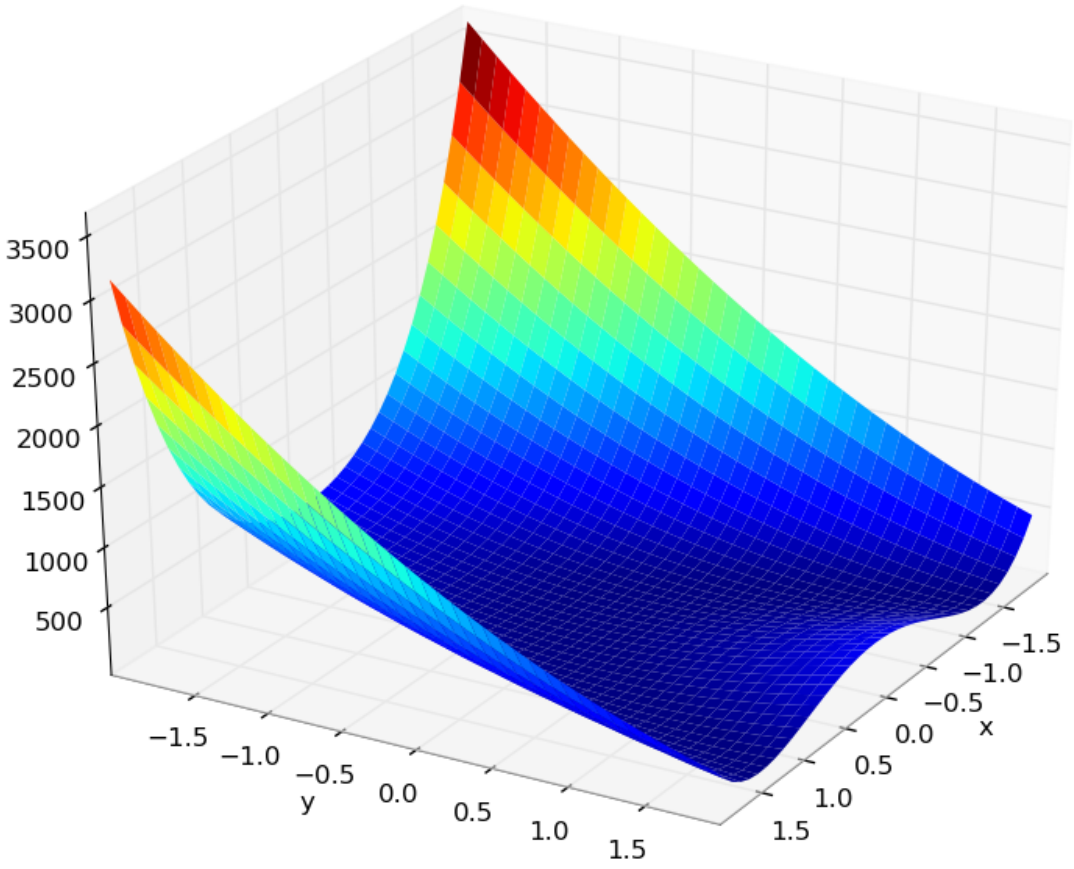


Figure 1.8: **This is the rosenbrock function.**

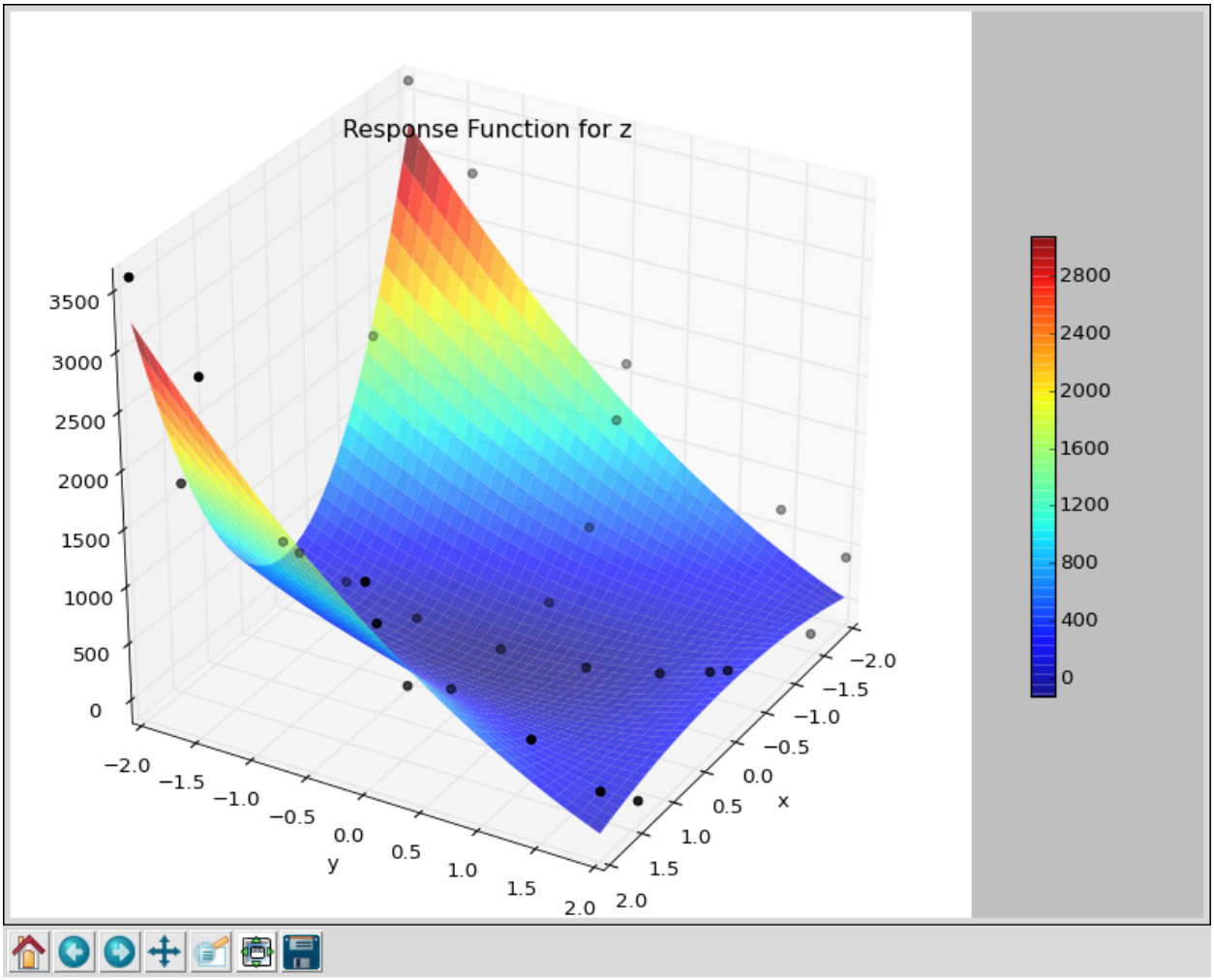


Figure 1.9: **This is the output from a level-3 Smolyak, which required 29 jobs**

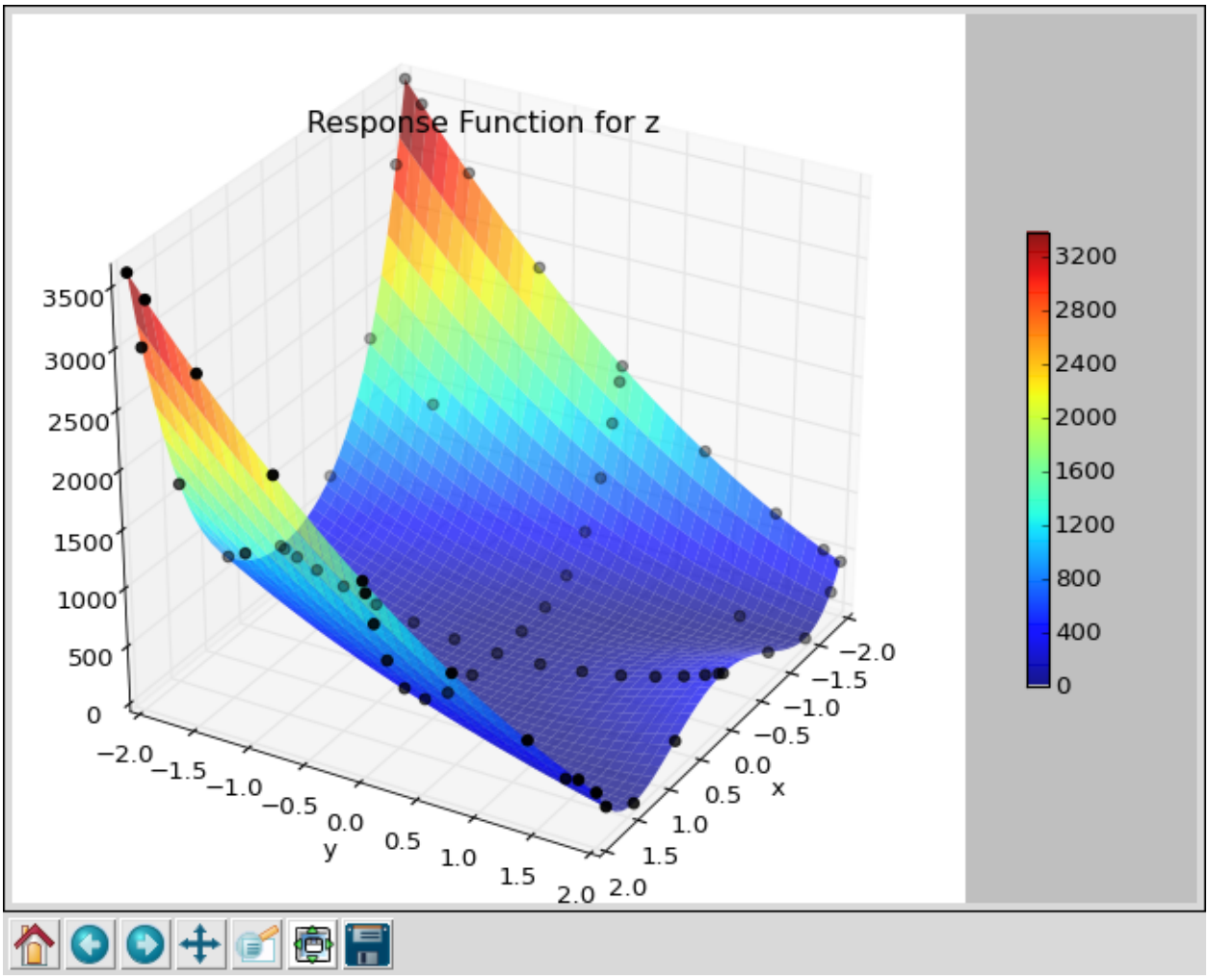


Figure 1.10: **This is the output from a level-4 Smolyak, which required 65 jobs**

```
~/puq/examples/rosen> puq extend sweep_140682290_A.hdf5
Extending sweep_140682290.hdf5 using Smolyak
Extending Smolyak to level 4

Processing <HDF5 dataset "z": shape (65,), type "<f8">
        Surface   = 100.0*x**4 - 200.0*x**2*y + 1.0*x**2 - 2.0*x + 100.0*y**2 + 1.0
        RMSE      = 3.96e-09 (1.10e-10 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    5.0835e+03    6.9637e+03
y    1.9661e+03    1.7441e+03
```

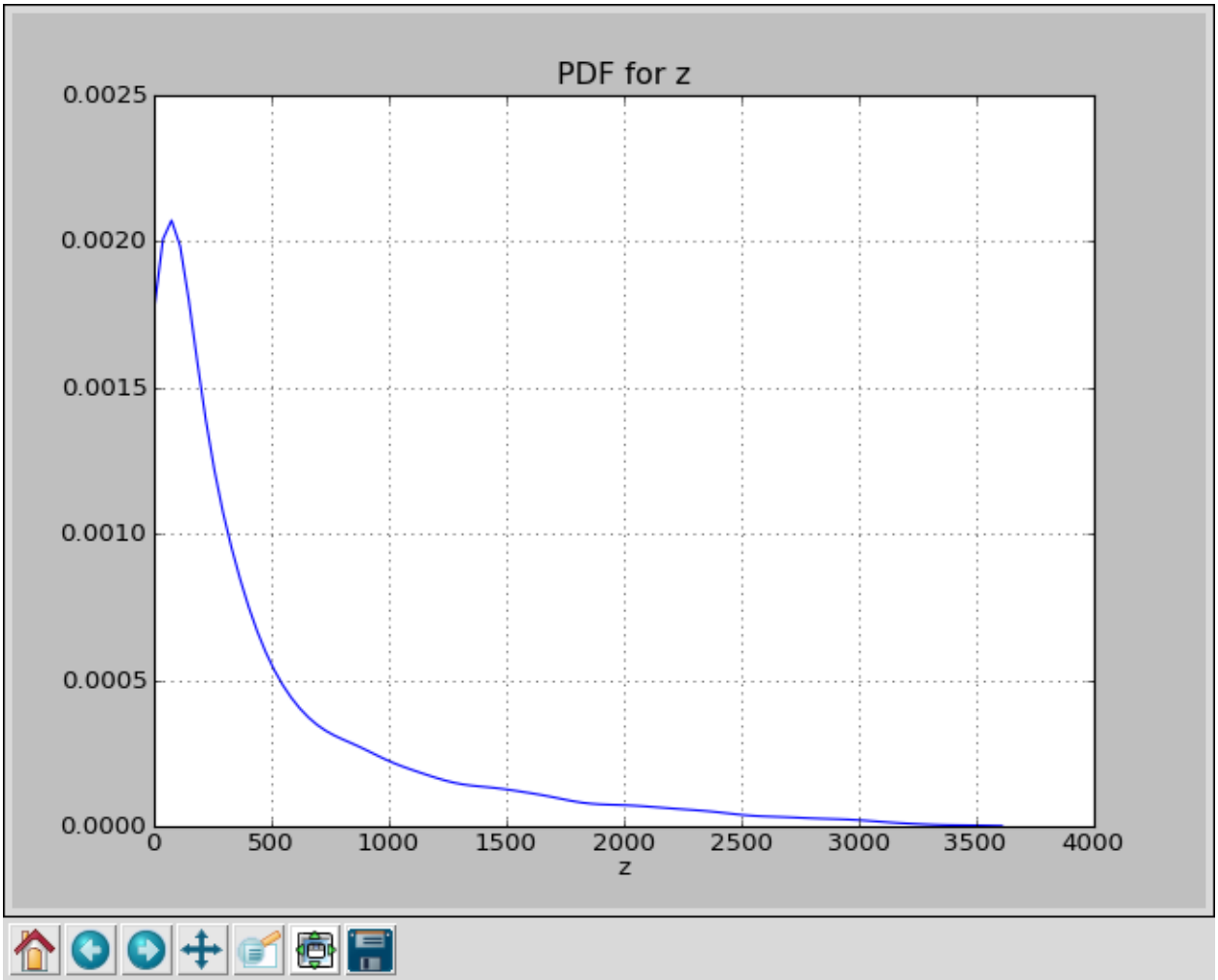


Figure 1.11: **PDF for Smolyak level 4, Rosenbrock function on [-2,2]**

With 65 samples Smolyak outperforms Monte Carlo with 1000 samples.

Note: Smolyak is only suitable for cases where the response function can be approximated as a polynomial, at least over the input range with which we are concerned. For functions with abrupt changes or discontinuities, *Latin Hypercube Sampling* is probably a better choice.

1.2.5 Modifying Smolyak Sweeps

Extending

If a Smolyak Sweep does not produce a good response surface, it is likely because the polynomial level was set too low. Fortunately we can easily extend a previous run, increasing its level by one. This is actually very efficient because each level uses all the information from the lower level. So some additional jobs will run and those results will be combined with all the previous results. For example:

```
~/puq/examples/rosen> puq extend sweep_140682290.hdf5
Extending sweep_140682290.hdf5 using Smolyak
Extending Smolyak to level 4

Processing <HDF5 dataset "z": shape (65,), type "<f8">
        Surface   = 100.0*x**4 - 200.0*x**2*y + 1.0*x**2 - 2.0*x + 100.0*y**2 + 1.0
        RMSE      = 3.96e-09 (1.10e-10 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    5.0835e+03    6.9637e+03
y    1.9661e+03    1.7441e+03
```

Modifying Parameters & Reusing Response Surfaces

Sometimes after you run a simulation, you get new information that causes you to adjust your input parameters by a bit. Perhaps a new set of measurements allowed you to tighten up the deviation a bit. Or maybe the mean shifted some. If the input range has not widened significantly, the response surface should still be valid and you can use it with your updated input PDF(s). All PUQ has to do is sample the response surface with the updated PDF(s) to generate new output PDF(s).

For example, take a look at puq/examples/test1.

```
#!/usr/bin/env python
"""
Example of using a UQ method with a sweep.

Usage: test1.py
"""
from puq import *

def run():
    # Declare our parameters here
    v0 = Parameter('v', 'velocity', mean=10.0, dev=1.5)
    mass = Parameter('m', 'mass', mean=95, dev=8)

    # Which host to use
    host = InteractiveHost()
    #host = PBSHost(env='/scratch/prism/memosa/env.sh', cpus_per_node=8, walltime='2:00')

    # select a parameter sweep method
    # These control the parameters sent to the test program
    # Currently the choices include Smolyak, and PSweep
    # For Smolyak, first arg is a list of parameters and second is the level
```

```
    uq = Smolyak([v0, mass], 3)

    # If we create a TestProgram object, we can add a description
    # and the plots will use it.
    prog = TestProgram('./test1_prog.py', desc='Test Program One.')

    # Create a Sweep object
    return Sweep(uq, host, prog)
```

Run it to generate the response surface:

```
~/puq/examples/test1> puq start test1
Saving run to sweep_145215760.hdf5

Processing <HDF5 dataset "kinetic_energy": shape (29,), type "<f8">
        Surface   = 0.5*m*v**2
        RMSE      = 4.15e-08 (3.53e-10 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
v    8.7761e+03    3.0974e+03
m    2.7263e+03    1.6263e+03
~/puq/examples/test1> mv sweep_145215760.hdf5 test1.hdf5
```

Later you want to recompute the output with new input PDFs. You will need to put the new Parameters in a file, with the same names (the first arg to *Parameter*). For example,

```
from puq import *

v = Parameter('v', 'velocity', mean=11.0, dev=1)
m = Parameter('m', 'mass', mean=105.0, dev=5)
```

Then you do 'puq plot –using filename':

```
/puq/examples/test1> puq plot --using new_pdfs.py  test1.hdf5
plotting PDF for kinetic_energy
REPLACING
        NormalParameter m (mass)
        PDF [63.9 - 126] 100 intervals
    mean=95  dev=7.99  mode=94.7
WITH
        NormalParameter m (mass)
        PDF [85.5 - 124] 100 intervals
    mean=105  dev=5  mode=105

REPLACING
        NormalParameter v (velocity)
        PDF [4.16 - 15.8] 100 intervals
    mean=10  dev=1.5  mode=9.94
WITH
        NormalParameter v (velocity)
        PDF [7.11 - 14.9] 100 intervals
    mean=11  dev=0.999  mode=11
```

test2.py is test1.py updated with the new parameters. You can do 'puq start' on it and compute a new response surface and PDF from scratch. It should match the one from test1 using the new parameters.

Note: PDFs are plotted by sampling the response surface with random numbers generated by the input PDFs. The

resulting plot has some minor variances due to the randomness of the inputs.

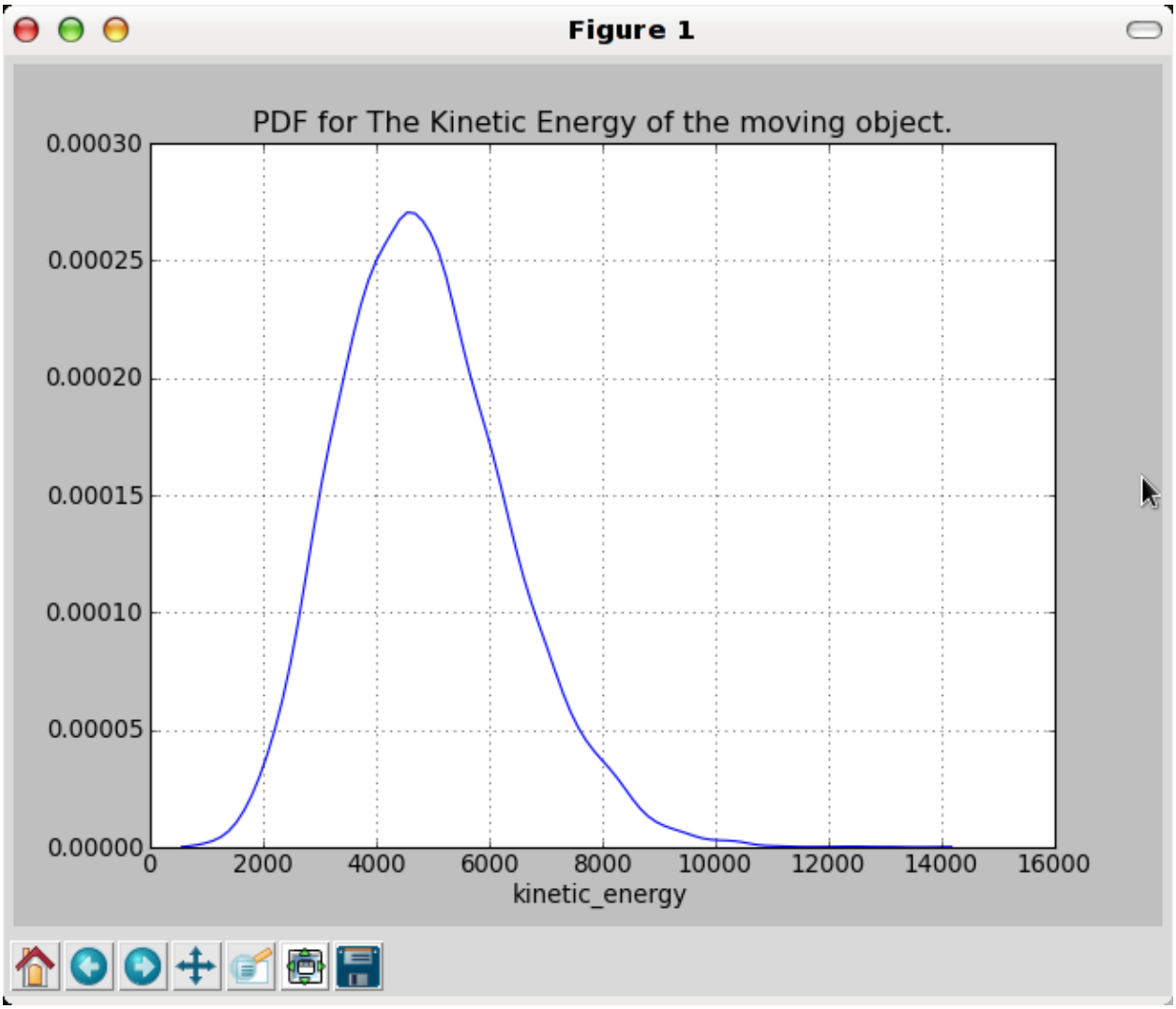


Figure 1.12: **PDF from test1.py**

1.2.6 Callbacks

Each UQ method can call a function after a sweep. This callback function returns a boolean. If it returns True, the sweep is over. Otherwise, sweep parameters can be changed and the sweep continued. The same thing could be done with a script, but the callback function makes it more convenient.

A good example of this is examples/iter/rosen_sm.py

```
from puq import *

# Example callback that causes the sweep
# to repeat until a specific condition is met.
def callback(sw, hf):
    rmsep = hdf.get_response(hf,'z').rmse()[1]
```

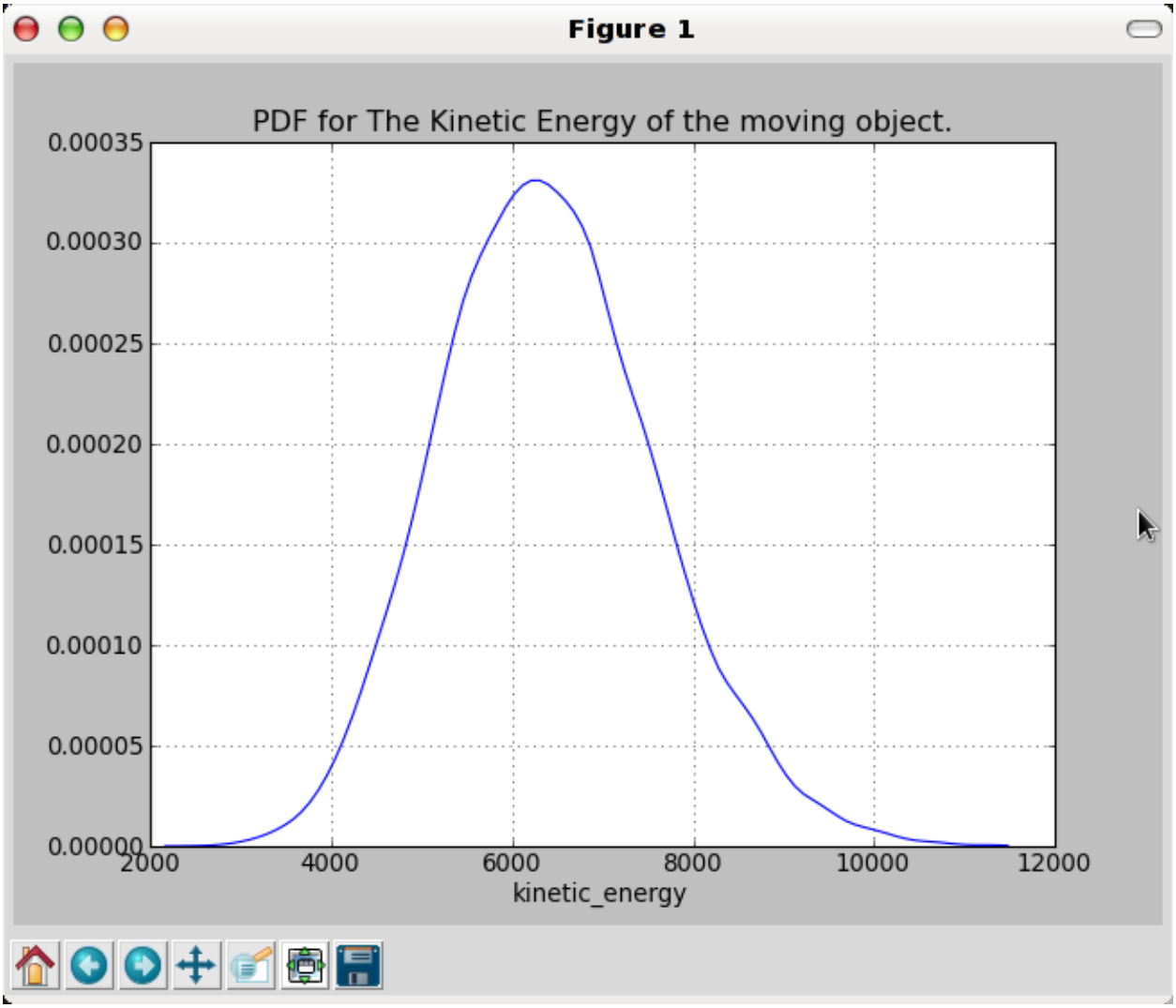


Figure 1.13: **PDF from test1 –using new_pdfs.py**

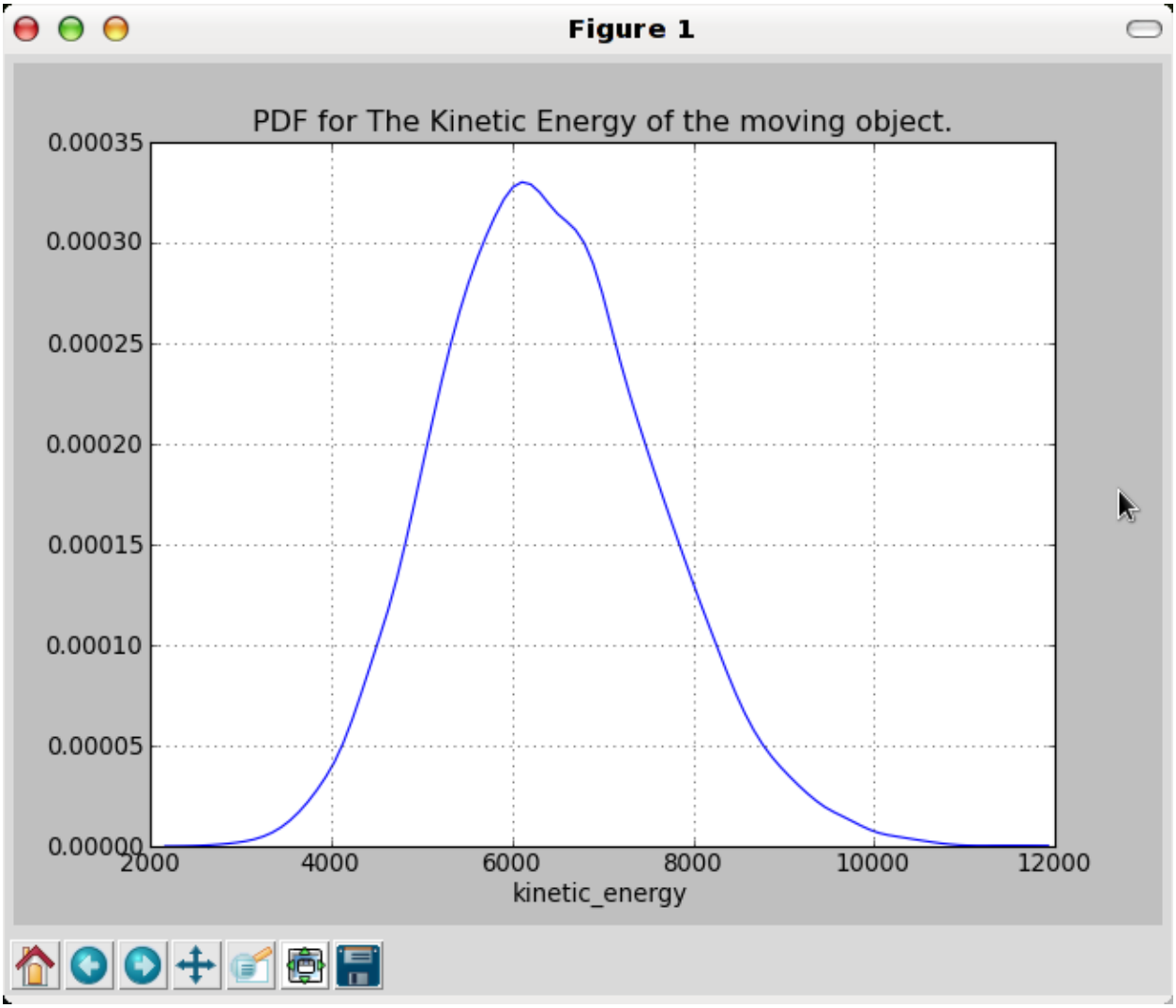


Figure 1.14: **PDF from test2.py.**

```
    print 'callback: rmse=%s%%' % rmsep

    # iterate until error < 1%
    if rmsep > 1:
        sw.psweep.extend()
        return False
    return True

def run():
    # Declare our parameters here. Both are uniform on [-2, 2]
    x = UniformParameter('x', 'x', min=-2, max=2)
    y = UniformParameter('y', 'y', min=-2, max=2)

    # Create a host
    host = InteractiveHost()

    # Declare a UQ method.
    uq = Smolyak([x,y], level=1, iteration_cb=callback)

    # Our test program
    prog = TestProgram('./rosen_prog.py', desc='Rosenbrock Function')

    return Sweep(uq, host, prog)
```

```
~/puq/examples/iter> puq start rosen_sm
Saving run to sweep_46691621.hdf5

Processing <HDF5 dataset "z": shape (5,), type "<f8">
    Surface   = -1.99999999999997*x + 669.0
    RMSE      = 6.84e+02 (4.26e+01 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    3.2080e+03    3.2080e+03
y    8.0000e+02    8.0000e+02
callback: rmse=42.5626179426%
Extending Smolyak to level 2

Processing <HDF5 dataset "z": shape (13,), type "<f8">
    Surface   = 300.99999999956*x**2 - 1.59872115546023e-14*x*y ...
    RMSE      = 7.40e+02 (2.05e+01 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    3.9402e+03    5.2374e+03
y    2.0828e+03    1.8510e+03
callback: rmse=20.4983861252%
Extending Smolyak to level 3

Processing <HDF5 dataset "z": shape (29,), type "<f8">
    Surface   = 3.10862446895043e-13*x**3 - 199.999999998686*x**2*y ...
    RMSE      = 2.40e+02 (6.66e+00 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
```

```
x    4.8307e+03    6.5106e+03
y    2.0022e+03    1.7623e+03
callback: rmse=6.66126889943%
Extending Smolyak to level 4

Processing <HDF5 dataset "z": shape (65,), type "<f8">
    Surface   = 100.000000001004*x**4 - 8.16013923099492e-14*x**3*y ...
    RMSE      = 9.29e-09 (2.57e-10 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    5.0835e+03    6.9637e+03
y    1.9661e+03    1.7441e+03
callback: rmse=2.57395820254e-10%
```

1.2.7 Calibration of Input Variables

Frequently computer models will have input parameters that cannot be measured and are not well known. In this case, Bayesian calibration can be used to estimate the input parameters. To do this, experimental data is required for the output and for all known input variables. The calibration process adjusts the unknown parameters to fit the observed data.

Using PUQ for Calibration

To use PUQ to do calibration, you will have to add experimental data to all the known input parameters. For the TestProgram, you will need to add experimental data and (optionally) a measurement error. The measurement error is a standard deviation.

The presence of experimental data tells PUQ to do calibration. The steps are as follows:

1. Build a response surface over the range of the input paramaters.
2. Do Bayesian calibration using the response surface for the likelihood function.
3. Manually adjust calibrated input parameter PDFs, if necessary. For example, truncate PDFs that go negative.
4. Generate output PDF.

Example

test2 in puq/examples/calibrate calibrates z. The experimental data was generated using a z with a Normal distribution of 12 and deviation of 2. Some noise was added with a sigma of 0.1. For calibration we use a uniform prior [5, 20] for z.

```
1  import puq
2  import numpy as np
3
4  def run():
5
6      # experimental data for z=Normal(12,2)
7      exp_x =np.array([5.04,  5.14,  4.78,  5.12,  5.11,  5.13,  4.97,  5.1 ,  5.53,  5.09])
8      exp_y = np.array([3.33,  3.56,  2.94,  3.27,  3.54,  3.4 ,  3.52,  3.63,  3.45,  3.19])
9      exp_data = np.array([-26.16, -18.39, -20.57, -45.24, -29.16, -22., -46.47,  -3.15, -10.48, -16.88
10
11     # measurement error
```

```
    sigma = 0.1

    # Create parameters. Pass experimental input data to
    # non-calibration parameters.
    x = puq.NormalParameter('x', 'x', mean=5, dev=0.2, caldata=exp_x)
    y = puq.NormalParameter('y', 'y', mean=3.4, dev=0.25, caldata=exp_y)
    z = puq.UniformParameter('z', 'z', min=5, max=20)

    # set up host, uq and prog normally
    host = puq.InteractiveHost()
    uq = puq.Smolyak([x,y,z], level=2)
    prog = puq.TestProgram('./model_1.py', desc='model_1 calibration')

    # pass experimental results to Sweep
    return puq.Sweep(uq, host, prog, caldata=exp_data, calerr=sigma)
```

```
~/puq/examples/calibrate> puq start test2.py
Saving run to sweep_85212814.hdf5

Processing <HDF5 dataset "f": shape (25,), type "<f8">
        Surface   = 1.0*x**2 + 1.0*x*y + 0.75*y**2 + 2.0*y - 7.0*z + 2.0
        RMSE      = 2.15e-11 (1.74e-11 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
z    1.0500e+02    1.8394e-11
y    1.8630e+01    1.1297e+00
x    1.6505e+01    1.0054e+00
Performing Bayesian Calibration...
[****************100%******************]  12000 of 12000 complete
Calibrated z to Normal(12.0786402402, 2.2152608414).
```

1.2.8 Sensitivity Analysis

With any Smolyak run, sensitivity analysis is done automatically during the analysis phase. The elementary effects method is used on the points in the sparse grid. Because of the way sampling points are laid out in a grid covering the input parameter space, the elementary effects can be computed at no additional cost.

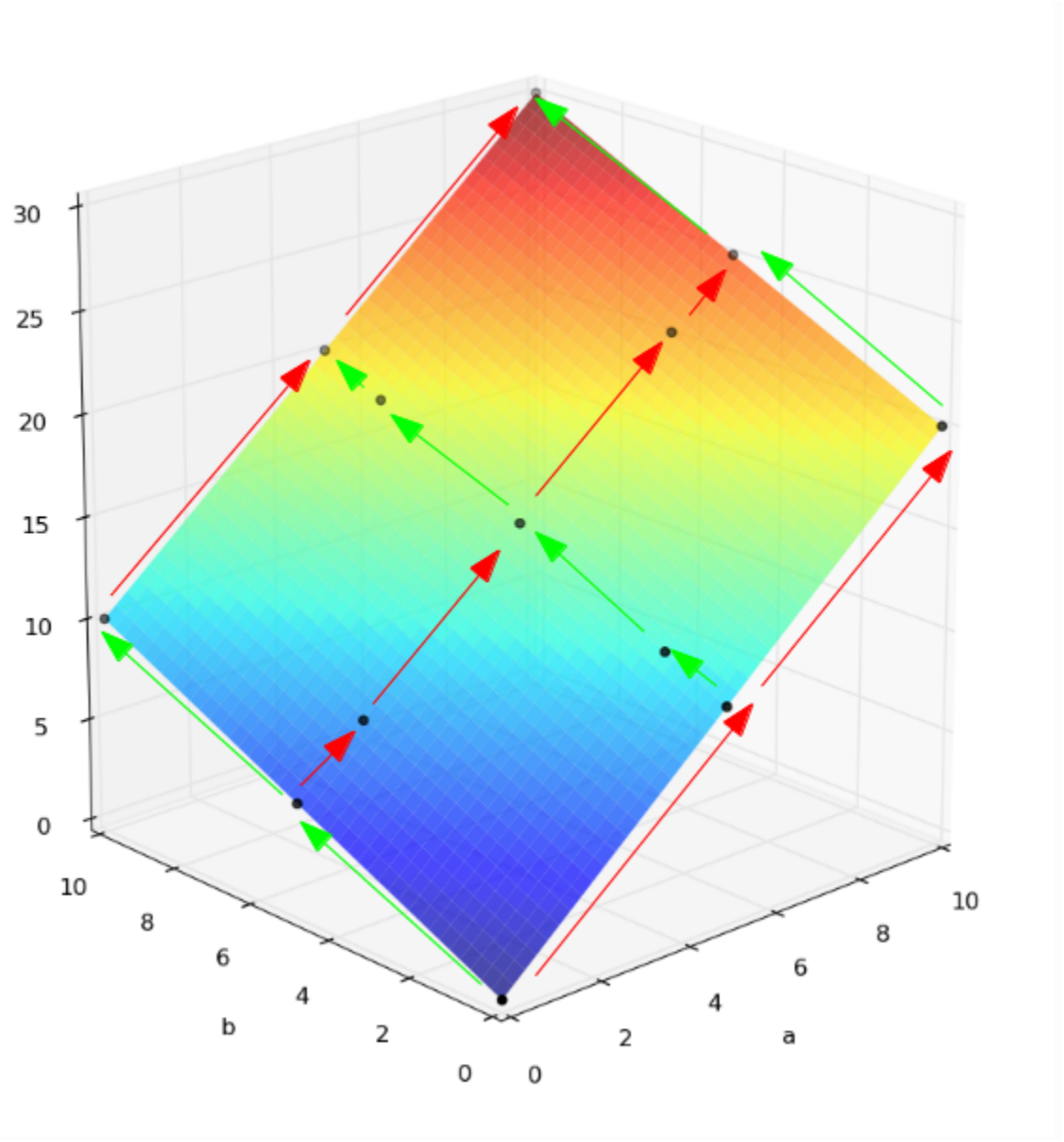
The sensitivity measurements are **u*** and **dev** (σ). u* is the mean of the distribution of the absolute values of the elementary effects of the input factors, $\mu^* = \frac{1}{r}\sum_{j=1}^{r}\left|d\left(X^{(j)}\right)\right|$ where $d(X)$ is the elementary effect for the input parameter. It is the overall influence of the input parameter on the output.

dev is the standard deviation of the distribution of the elementary effects for the parameter. $\sigma = \sqrt{\frac{1}{(r-1)}\sum_{j=1}^{r}\left(d\left(X^{(j)}\right)-\mu\right)^2}$ If dev is high then the input parameter's influence on the output varies widely depending on the sample point. A linear response will have a dev of 0.

An example is shown below.

Elementary effects for input parameter **a** are computed along each red arrow. Elementary effects for input parameter **b** are computed along each green arrow. Each elementary effect is the change in output value across the arrow scaled by the length. For example, the output goes up by 10 when **a** changed by half its range, so that elementary effect is 20. Because the above example is linear, all the elementary effects for **a** are 20.

Sensitivity test 1 has two input parameters with range [0 10]:


30
25
20
15
10
5
0
10
8
6
4
2
0
b
0
2
4
6
8
10
a

```
~/puq/examples/sensitivity> puq start test1
Sweep id is 30573805

Processing <HDF5 dataset "p": shape (13,), type "<f8">
        Surface   = 2.0*a + 1.0*b
        RMSE      = 2.40e-12 (8.00e-12 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
a    2.0000e+01    1.9476e-11
b    1.0000e+01    9.7339e-12
```

PUQ prints the list of input variables (parameters) in order from the highest u* to the lowest.

For more information about this method, see Elementary_Effects and Sensitivity_Analysis.

Effect of Smolyak Level on Sensitivity

The **level** parameter to the Smolyak method sets the polynomial degree for the response surface. A level 1 Smolyak run require only 2*ndim+1 (ndim = number of dimensions, or input parameters) calculations and the response surface will be linear. This will usually be a poor fit for a response surface, bit it might be sufficient for basic sensitivity analysis.

A More Complex Example

In Sensitivity_Analysis, an example is worked using Sobol's g-function,

$$y = \prod_{i=1}^{k} \frac{|4x_i - 2| + a_i}{1 + a_i} \quad x \in [0,1]^k$$

For the worked example, $k = 6$ and $a = [78, 12, 0.5, 2, 97, 33]$. The estimated sensitivity measures are:

	u*	dev
x3	1.76	2.05
x4	1.19	1.37
x2	0.28	0.32
x6	0.10	0.12
x1	0.06	0.06
x5	0.04	0.04

Using PUQ, level 1 Smolyak:

```
~/puq/examples/sensitivity> puq start sobol
Sweep id is 30605799

SENSITIVITY:
Var       u*            dev
-----------------------------
x3    1.5566e+00    1.5566e+00
x4    3.8914e-01    3.8914e-01
x2    6.4856e-02    6.4856e-02
x6    2.3584e-02    2.3584e-02
x1    9.9779e-03    9.9779e-03
x5    8.0235e-03    8.0235e-03
```

The estimates don't agree exactly because they use different sample points, however they do agree on the order of significance for the inputs. If would be easy to increase the accuracy by using a level 2 Smolyak grid.

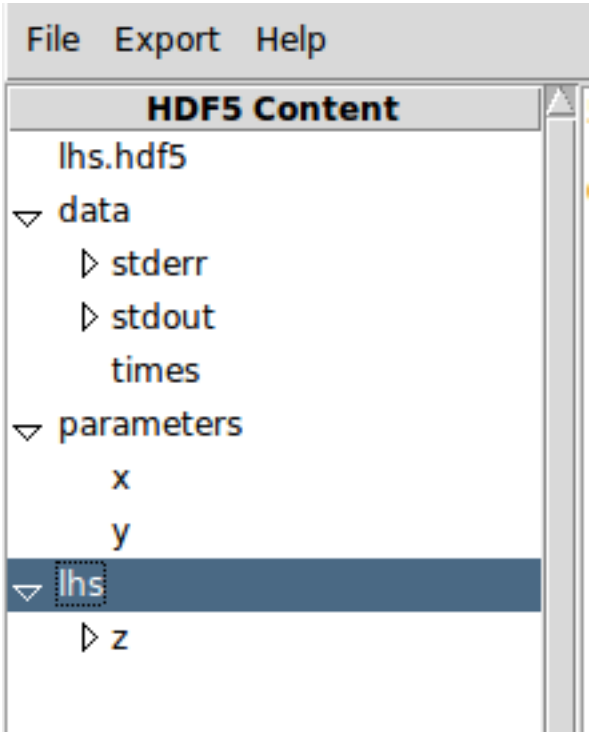
1.2.9 Using PUQ Analyze

Use *puq analyze* to see all the available information stored in an HDF5 file after a UQ run. You can also use it to tweak PDFs, plot PDFs and response surfaces, or save them to files:

```
Usage: puq analyze [options] [hdf5_filename].
Type 'puq analyze -h' for option descriptions.

Options:
  -h, --help           show this help message and exit
  -v                   Verbose.
  --psamples=PSAMPLES  Filename CSV table of parameter samples.
  -r                   Re-analyze the data.
```

The **- -psamples** option is for use with externally generated samples for correlated input parameters. It is deprecated and may be removed in the future.

The **-r** option tells *PUQ* to reparse the output files and extract the data again before analysis. It is implied by **- - filter**. This option is not usually necessary.



Here is a typical menu from *puq analyze*. Under **data** is the captured stdout and stderr for each job. The run times for each job is stored in **times**.

Under **parameters** is the list of input parameters.

All analysis, such as mean, deviation, sensitivity, response surface, and PDF, are stored in a section with the name of the method, followed by the output variable name. In the above example, it is "/lhs/z".

Response Surfaces

If you are using a method that employs response surfaces, you can see (and sometimes tweak) it using *puq analyze*. What you see will differ depending on the response surface type. For example, `Smolyak` generates a best-fit polynomial response surface of the requested degree. `MonteCarlo` and `LHS` build a response function using Radial Basis Functions. By default it uses a multiquadric function, although you can change that.

Here are some examples from puq/examples/discontinuous/dome which is a very difficult surface to fit:

```
~/puq/examples/discontinuous/dome> puq start -f smolyak.hdf5 smolyak.py
Saving run to smolyak.hdf5

Processing <HDF5 dataset "z": shape (321,), type "<f8">
        Surface   = -10.1722431182861*x**6 - 31.8686981201172*x**5 + 47.3970642089844*x**4*y**2 -
        47.3970642089844*x**4*y + 130.954284667969*x**4 - 143.717315673828*x**3*y**2 +
        143.717315673828*x**3*y - 140.402770996094*x**3 + 113.759033203125*x**2*y**4 -
        227.51806640625*x**2*y**3 + 285.599304199219*x**2*y**2 - 171.840301513672*x**2*y +
        62.2936553955078*x**2 - 118.73616027832*x*y**4 + 237.472320556641*x*y**3 -
        197.646301269531*x*y**2 + 78.9101409912109*x*y - 11.4179744720459*x + 202.004089355469*y**6 -
        606.01220703125*y**5 + 703.17041015625*y**4 - 396.320373535156*y**3 + 111.905548095703*y**2 -
        14.7473888397217*y + 0.763792037963867
        RMSE      = 2.36e-01 (2.36e+01 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    8.9079e-01    2.8561e+00
y    8.5187e-01    1.4274e+00
```

The sample points are in black. They are not on the response surface so you can see it is a poor fit. Notice also that the RMSE (Root Mean Square Error) is 23.6% which is high.

```
~/puq/examples/discontinuous/dome> puq start -f lhs.hdf5 lhs.py
Saving run to lhs.hdf5

Processing <HDF5 dataset "z": shape (500,), type "<f8">
Mean   = 0.211760030354
StdDev = 0.298137023113
```

In the multiquadric fit, some spikes at the corners distort the response surface. A linear RBF appears to be the best fit.

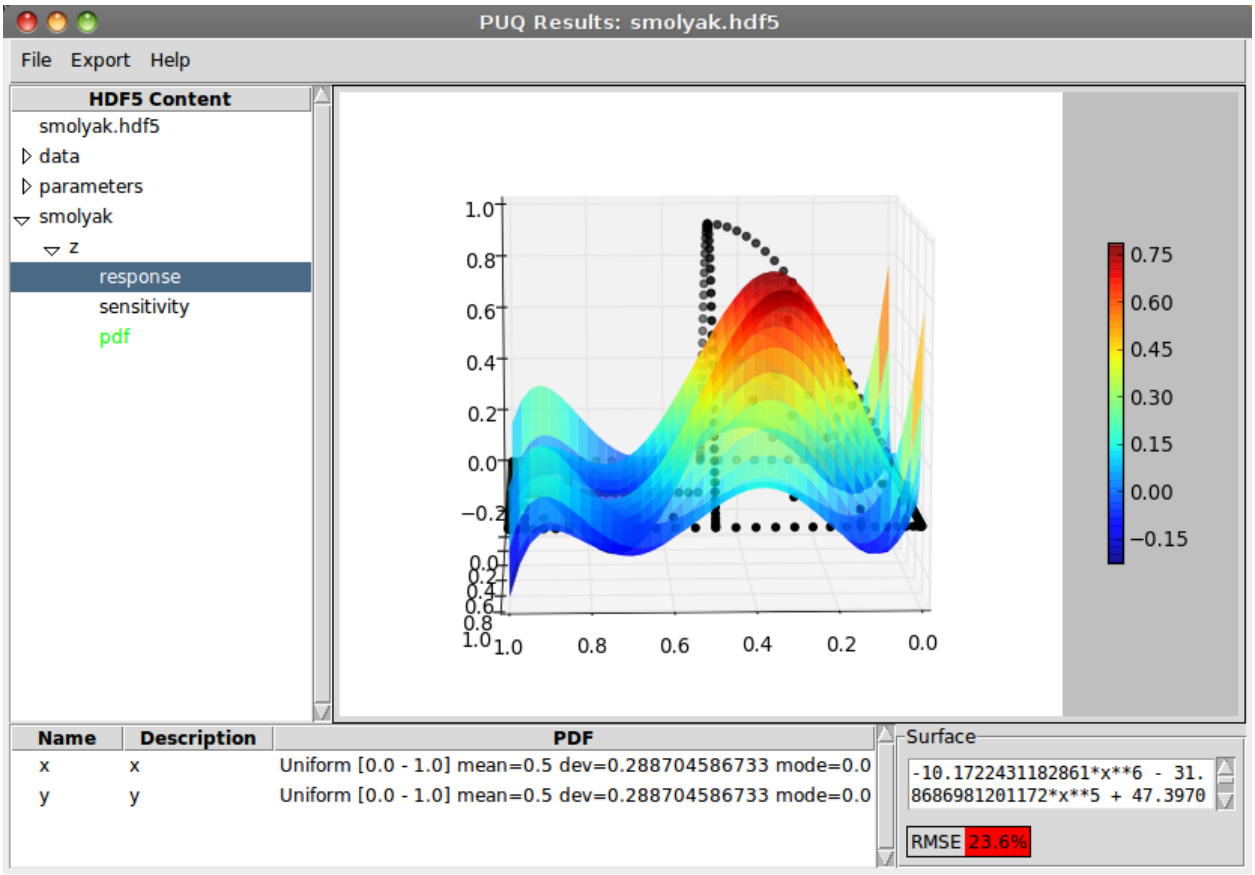
Probability Density Functions (PDFs)

PDFs will be listed in method/variable/pdf. If the word "pdf" is green, that indicates it is a placeholder for a PDF that will be calculated when you click on it. PDFs are calculated by sampling each input parameter (10000 times by default) and running those values through the response surface. The results are then displayed using a Kernel Density Estimate or a Linear fit. You can adjust those values as well as the minimum and maximum of the PDF. When you are finished adjusting the fit, the PDF can be saved as a plot or exported to the clipboard or a file. Exported PDFs can later be used as input parameters in other simulations or compared using *puq read*.

1.2.10 Using PUQ Read

Use **puq read** to read parameters, PDFs, and response surfaces from json files or python control scripts.

```
~/puq/examples/basic> puq read -h
Usage: puq read [options] [object] ...
```


PUQ Results: smolyak.hdf5
File
Export
Help
HDF5 Content
smolyak.hdf5
data
parameters
smolyak
z
response
sensitivity
pdf
1.0
0.8
0.6
0.4
0.2
0.0
0.75
0.60
0.45
0.30
0.15
0.00
−0.15
Name
Description
PDF
x
x
Uniform [0.0 - 1.0] mean=0.5 dev=0.288704586733 mode=0.0
y
y
Uniform [0.0 - 1.0] mean=0.5 dev=0.288704586733 mode=0.0
Surface
-10.1722431182861*x**6 - 31.
8686981201172*x**5 + 47.3970
RMSE 23.6%

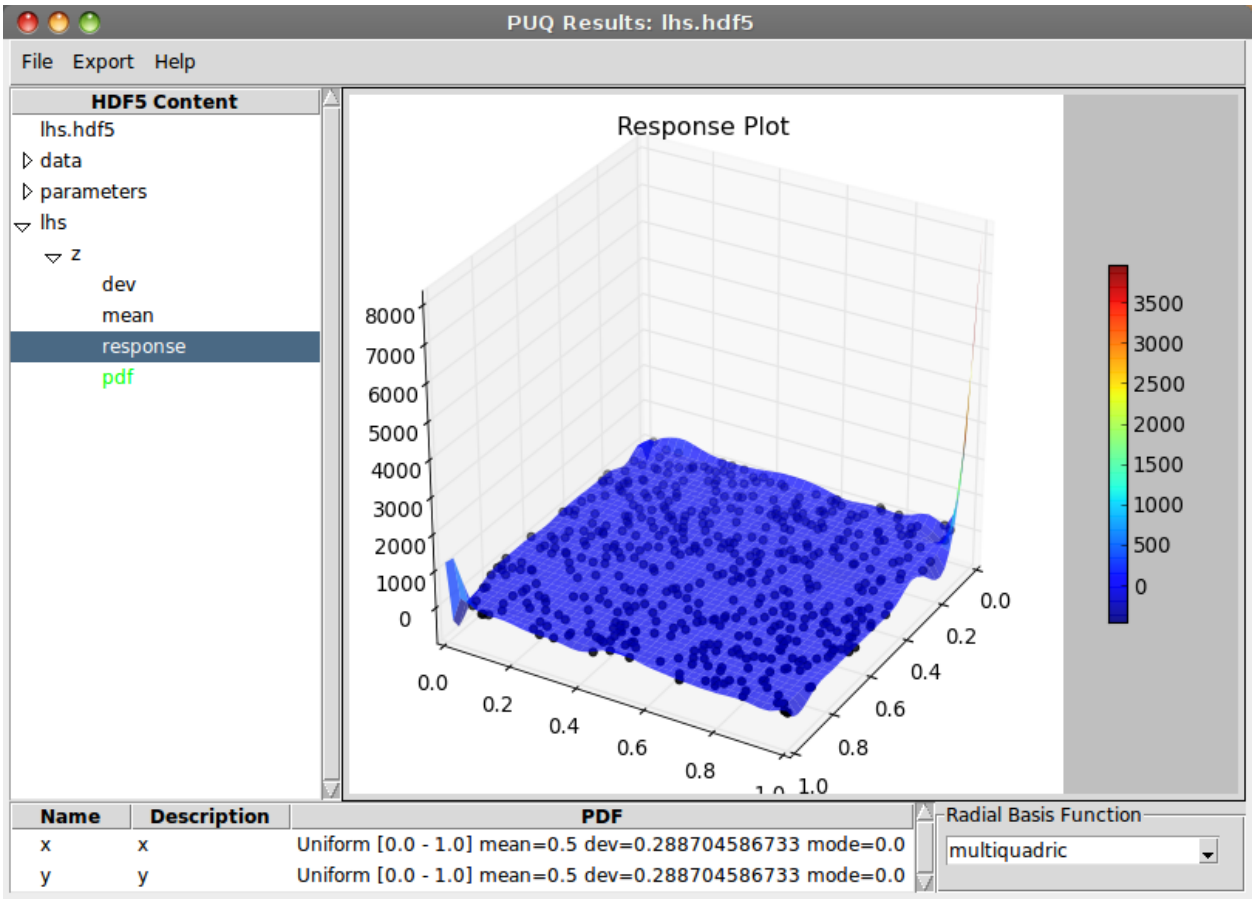


Figure 1.15: LHS with multiquadric RBF

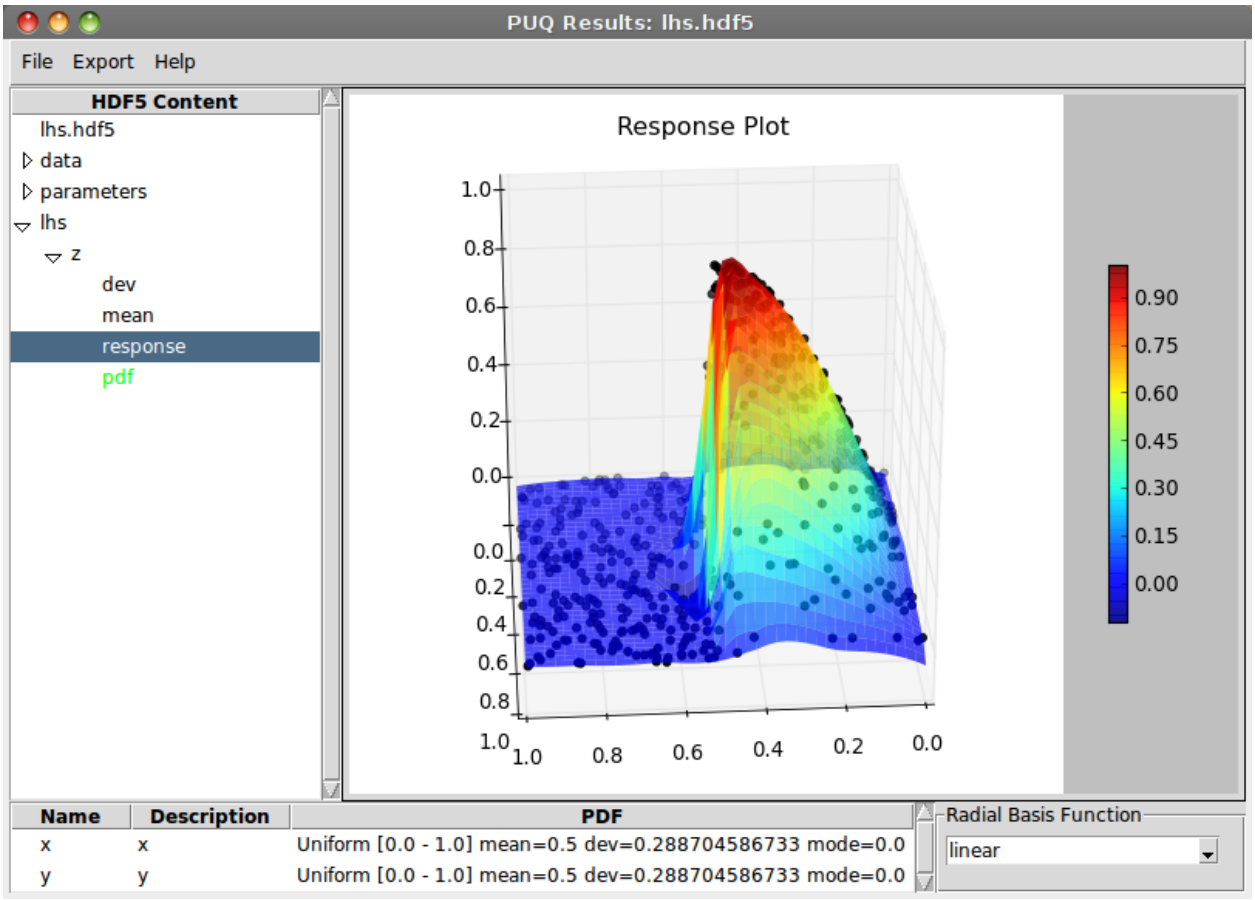


Figure 1.16: LHS with linear RBF

```
    where 'object' is a URI, python file, or JSON file

Options:
  -h, --help  show this help message and exit
  -c          Compare plots.

~/puq/examples/basic> puq read basic.py

List PDFs you want displayed, separated by commas.

Found the following PDFs:
0: x
1: y
Which one(s) to display? (* for all) *
```

You can use *puq analyze* to save PDFs to json files, then *puq read* to compare them. Or you can compare input parameters from a control file.

```
~/puq/examples/basic> puq read -c f.json g.json
```

1.2.11 Making PUQ Work With Your Code

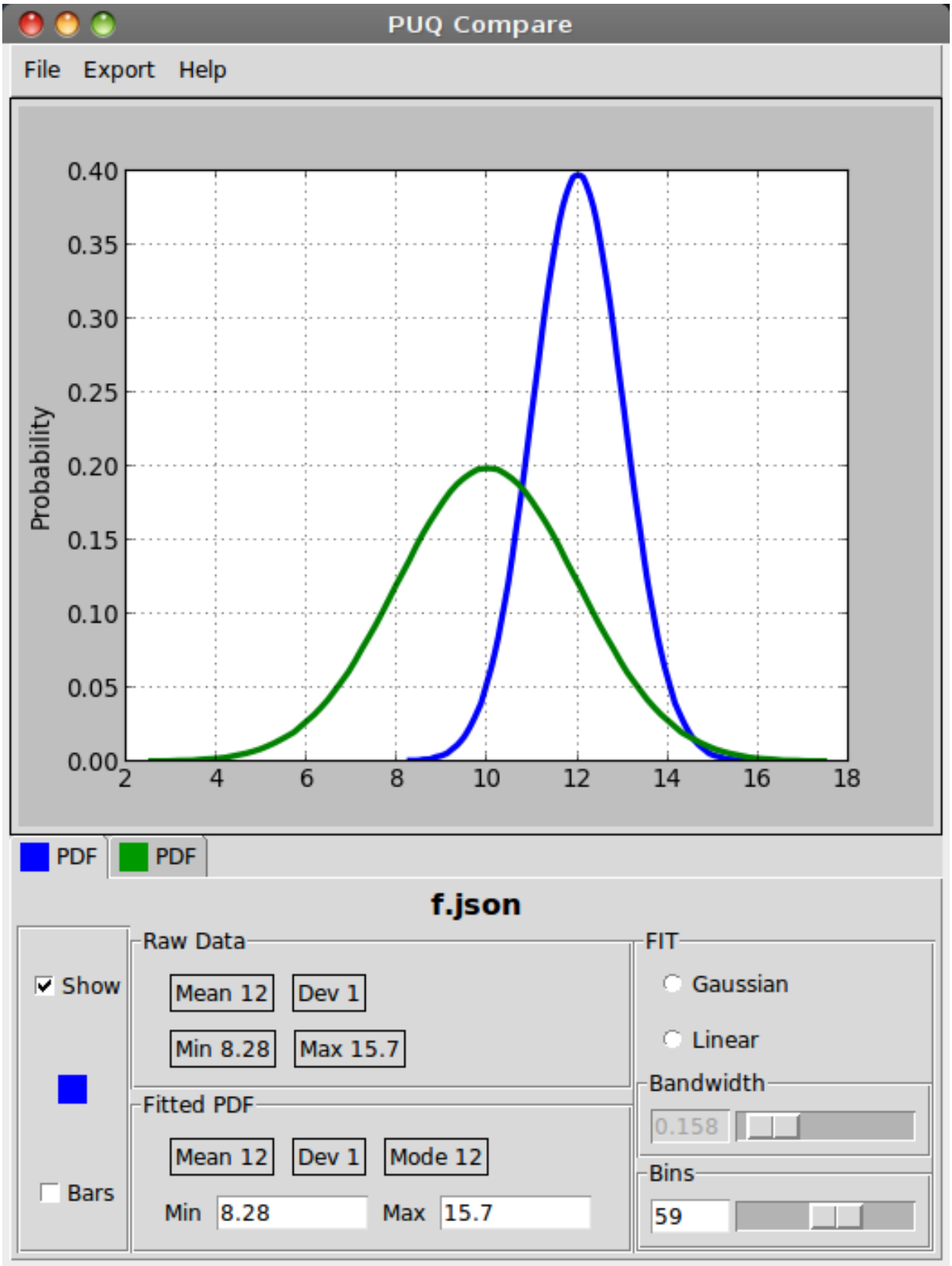
PUQ makes no assumptions about how your test program (or simulation) code works. However it does need a way to pass parameter values into the test program and read results.

C Programs

In this example, we use a test program compiled in C. The point of this is to show how to pass parameters to test programs that do not use the "–varname=value" pattern we have been using.

The test program is puq/examples/rosen/rosen_cprog.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#include "dump_hdf5.h"

int main(int argc, char *argv[]) {
        int c;
        double x, y, z;
        while ((c = getopt(argc, argv, "x:y:")) != -1)
                switch (c) {
                case 'x':
                        x = atof(optarg);
                        break;
                case 'y':
                        y = atof(optarg);
                        break;
                default:
                        printf("Usage: %s -x val -y val\n", argv[0]);
                        return(1);
                }

        /* compute rosenbrock function */
        z = 100.0 * (y - x*x)*(y - x*x) + (1.0 - x)*(1.0 - x);
```


PUQ Compare
File
Export
Help
Probability
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
2
4
6
8
10
12
14
16
18
PDF
PDF
f.json
Show
Bars
Raw Data
Mean 12
Dev 1
Min 8.28
Max 15.7
Fitted PDF
Mean 12
Dev 1
Mode 12
Min 8.28
Max 15.7
FIT
Gaussian
Linear
Bandwidth
0.158
Bins
59

```
25
26          dump_hdf5_d("z", z, "f(x,y)");
27
28          /* important! return code of 0 indicates success */
29          return 0;
30  }
```

Note: dump_hdf5_d() dumps a double in our output format. It is included in dump_hdf5.h.

See Also:

puqutil - Output Functions for PUQ

You can compile it and test it out:

```
~/puq/examples/rosen> gcc -o rosen_cprog rosen_cprog.c
~/puq/examples/rosen> ./rosen_cprog -x 0 -y 1
HDF5:{'name':'z','value':1.0100000000000000e+02,'desc':'f(x,y)'}:5FDH
```

It works the same as the python version. To use it, we need to make a simple change to our control script.

```
1  from puq import *
2
3  def run(lev=2):
4      # Declare our parameters here
5      p1 = Parameter('x', 'x', min=-2, max=2)
6      p2 = Parameter('y', 'y', min=-2, max=2)
7
8      # Create a host
9      host = InteractiveHost()
10
11     # Declare a UQ method.
12     uq = Smolyak([p1, p2], level=lev)
13
14     prog = TestProgram(desc='Rosenbrock Function (C)',
15         exe="./rosen_cprog -x $x -y $y")
16
17     return Sweep(uq, host, prog)
```

The important line is where it says:

```
exe="./rosen_cprog -x $x -y $y"
```

See Also:

TestProgram

Finally, do 'puq start rosen_c' and try it out.

Matlab Programs

In this example, we use a test program written in Matlab and executed either by Matab or Octave. The example is very much like the previous one using C.

The test program is puq/examples/rosen/rosen.m

```
1  function[]=rosen(x,y)
2
3  z = 100*(y-x^2)^2 + (1-x)^2;
```

```
4
5  fprintf('HDF5:{"name": "z", "value": %.16g, "desc": ""}:5FDH\n', z);
```

The biggest difference between this and our other example is that we do not (yet) have a dump_hdf5() function for Matlab. Instead we must write out output variable(s) in carefully formatted fprintfs. This puts our data in a tagged format such that PUQ can recognize it.

You should test the test program:

```
~/puq/examples/rosen> octave -q --eval 'rosen(1,.5)'
HDF5:{"name": "z", "value": 25, "desc": ""}:5FDH
```

It works the same as the python version. To use it, we need to make a simple change to our control script.

```
1  from puq import *
2
3  def run(lev=4):
4      # Declare our parameters here. Both are uniform on [-2, 2]
5      p1 = UniformParameter('x', 'x', min=-2, max=2)
6      p2 = UniformParameter('y', 'y', min=-2, max=2)
7
8      # Create a host
9      host = InteractiveHost()
10
11     # Declare a UQ method.
12     uq = Smolyak([p1, p2], level=lev)
13
14     prog = TestProgram(
15       exe="octave -q --eval 'rosen($x, $y)'",
16       # exe="matlab -nodisplay -r 'rosen($x, $y);quit()'",
17       desc='Rosenbrock Function (octave)')
18
19     return Sweep(uq, host, prog)
```

The syntax to execute a function from the command line differs between Octave and Matlab. If you have Matlab, you can comment the Octave line out and uncomment the Matlab line.

Finally, do 'puq start rosen_ml' and try it out.

See Also:

TestProgram

Fortran Programs

There is a Fortran version of the Rosenbrock test in examples/fortran. It works very much like the C version.

1.2.12 Handling Output

The Standard Way

In our previous examples, we wrote test programs that used the functions in *puqutil - Output Functions for PUQ* to output data. What these functions do is to simply format the data written to standard output in such a way that PUQ can automatically recognize the data it is supposed to save. Anything else written to stdout, as well as other files, is skipped.

When looking at stdout, you will see lines like this:

```
HDF5:{'name':'z','value':1.0,'desc':'f(x,y)'}:5FDH
```

This tells PUQ that it should save the value "1.0" in the HDF5 file under 'output/data/z'. Actually it will save the results from all jobs in the sweep as an array of output values under 'output/data/z'.

If you write the same variable more than once in a job, it will confuse PUQ. It expects one value per variable per job. And by default it will run the UQ method on each output variable. So it you output three variables, you will get three output PDFs.

However, each output variable can be an array of values. For example, you can output a 10x10 array (or an n-dimensional array!). In 'output/data/varname' will be an array of 10x10 arrays. And by default, PUQ will generate a 10x10 array of output PDFs. And 'puq plot' will plot all 100 PDFs. See puq/examples/test1 for an example of outputting arrays of data.

When the Standard Way Doesn't Work

This is all very good, but what if your test program is some huge binary that cannot be modified to output data in PUQ's format?

One solution would be to write a wrapper in a scripting language like Python, Tcl, or Ruby. The wrapper might modify the parameters, pass them to the the big test program, capture the output and reformat it for PUQ.

Wrappers

What do you do when you have a complex simulation that cannot be easily modified to support PUQs input and output requirements? Perhaps it is commercial simulation code that cannot be modified. In these cases, you must write a *wrapper*; a small program that translates between PUQs I/O requirements and those of the wrapped simulation code.

Example wrapper scripts are in puq/examples/wrapper. Examples include:

sim.py Simulation. Reads parameters on the command line and writes the output embedded in some text.

sim_file.py Like the previous, but reads parameters from a file and writes output to a file.

---

sim_wrap.py Example python wrapper for sim.py.

sim_wrap.sh Example bash wrapper for sim.py.

sim_file_wrap.py Example python wrapper for sim_file.py.

sim_file_wrap.sh Example bash wrapper for sim_file.py.

---

quad.py Control script for PUQ that uses sim_wrap wrappers.

quad_file.py Control script for PUQ that uses sim_file_wrap wrappers.

---

sim.py and *sim_file.py* are treated as black box simulations. They have particular ways they work and the wrapper scripts need to wrap them.

The python and bash example wrappers are interchangeable. You can write wrappers in any language.

The control scripts interact with the wrappers through the *TestProgram* class. They control how parameters are passed on the command line. They also can cause every job to be run in a separate directory, and can have certain common datafile copied into those directories.

1.2.13 Using The PBS Scheduler

There are two different ways to run PUQ on compute clusters.

Batch Scripts

The first method uses a batch script to run PUQ. The PUQ control script uses `InteractiveHost` like normal. A typical PBS script would look like this:

```
#!/bin/bash -l
#PBS -q standby
#PBS -l nodes=1:ppn=48
#PBS -l walltime=0:02:00
#PBS -o run_pbs.pbsout
#PBS -e run_pbs.pbserr
cd  $PBS_O_WORKDIR
source /scratch/prism/memosa/env-hansen.sh
puq start my_control_script
```

Of course you need to set the walltime, nodes and ppn for your run.

Using PBSHost

Another way to use PUQ with clusters is by using `PBSHost`. This method runs a PUQ monitor process on the frontend which submits PBS or Moab jobs as needed. This allows you to monitor the progress of PUQ. The disadvantage of this approach is that submission of new batch jobs stops if the PUQ monitor is killed.

To use PBSHost, in any script where you see:

```
host = InteractiveHost()
```

simply replace **InteractiveHost** with **PBSHost**. You will need to give PBSHost a few additional arguments.

- **env** [Bash environment script (.sh) to be sourced.] The is a sh or bash script that normally loads modules and sets paths.
- **cpus** : Number of cpus each process uses.
- **cpus_per_node** [How many cpus to use on each node. PUQ has no] way to determine the number of cpus per node, so there is no default. You must supply this. It does not have to be the actual number of CPUs in the clusters' nodes, but it should not be more.
- **qname** : The name of the queue to use. 'standby' is the default
- **walltime** [How much time to allow for the process to complete. Format] is HH:MM:SS. Default is 1 hour.
- **modules** [List of additional required modules. Default is none.] Used when the testprogram requires modules to be loaded to run. For example, to run a matlab script you will need to set this to ['matlab']
- **pack** [Number of sequential jobs to run in each PBS script. Default is 1.] This is used to pack many small jobs into one PBS script.

Examples

1. **You want to run Monte Carlo with 10000 jobs, each just a few seconds:**
    - cpus=1
    - cpus_per_node=8

- pack=1000
- walltime='00:10:00'

PBSHost will create 10 PBS scripts, each with 1000 jobs, running 8 at a time (because each takes only 1 CPU and nodes have 8). Walltime is 1000 * 3 / 8 seconds which is a bit over 6 minutes. I rounded up to 10.

2. **Like the previous, except each job uses lots of memory, so you can have only two running in a node at a time.**

   - cpus=1
   - cpus_per_node=2
   - pack=1000
   - walltime='00:25:00'

   PBSHost will create 10 PBS scripts, each with 1000 jobs, running 2 at a time (because each takes only 1 CPU and nodes have 2). Walltime is 1000 * 3 / 2 seconds which is 25 minutes.

3. **You want to run Monte Carlo with 10000 jobs, each just a few seconds:**

   - cpus=1
   - cpus_per_node=8
   - walltime='00:00:01'

   PBSHost will create 1250 PBS scripts, each with 8 jobs, running 8 at a time (because each takes only 1 CPU and nodes have 8).

Note: There is currently a hardcoded limit of 200 PBS jobs queued at once. PUQ will monitor PBS jobs, and as they complete, will submit more until all 1250 have completed.

---

4. **You have 128 mpi jobs that run on 48 CPUs.**

   - cpus=48
   - cpus_per_node=8
   - pack=2

   PBSHost will allocate 6 nodes for each job. 64 PBS jobs will be submitted. Each PBS job will run two mpi jobs sequentially. walltime should be set to twice what each job takes to run.

1.2.14 Plotting

Note: The following information shows how to use the old 'plot' function. You may find it more convenient to use the plot function in *analyze* instead.

There are two main options for plotting:

puq plot Plots PDFs

puq plot -r Plots response surface of sampled function.

By default, plots will be rendered to the current display. If you want the to plot to a file, use **-f**.

```
~> puq plot -h
Usage: puq plot [options] hdf5_filename.

Examples:
plot                  Plots all output PDFs. This is the default.
plot -k               Plots output PDFs using Gaussian Kernel Density.
plot -l -k            Plots output PDFs using Gaussian KDE and linear interpolation.
plot -v temp          If temp is an output variable, plots its PDF.
plot -v 'v1,v2'       Same as before, except plot v1 and v2.
plot -r               Plot response surface of output variables.
plot -r -v 'v1,v2'    Plots response surface of output variables v1 and v2.
plot -f fmt           Save plots. Valid values for 'fmt' are eps, pdf, png, ps, raw,
                      rgba, svg, and svgz.
plot -h               Help with additional options.

Options:
  -h, --help           show this help message and exit
  -r                   Response Surface Plot
  -v V                 Variable list. If multiple, put them in quotes and
                       separate by spaces or commas.
  -f F                 Format [pdf|ps|png|svg|i] [i]
  -l                   Plot output PDF using linear interpolation from a
                       histogram. [False]
  -k                   Use Gaussian Kernel Density Estimator on output PDFs.
                       [True]
  --nogrid             Don't show grid
  --title=TITLE        Title. Default is the Test Program description.
  --xlabel=XLABEL      Label for the X-axis. Overrides the default which
                       depends on the plot type.
  --ylabel=YLABEL      Label for the Y-axis. Overrides the default which
                       depends on the plot type.
  --zlabel=ZLABEL      Label for the Z-axis. Overrides the default which
                       depends on the plot type.
  --fontsize=FONTSIZE  Normal font size in points.
  --using=USING        Filename containing substitute parameter(s).
```

PDF Plots

If your UQ run has more than one output variable, by default all of the output response surfaces or PDFs will be plotted. If you only wish to plot certain output variables, list them using the **-v** option.

When using **-v**, you don't have to use quotes unless you have whitespace in your variable list.

Example

In examples/basic, there are two input variables, 'x' and 'y' and three outputs, $f(x, y) = x$, $g(x, y) = y$, and $h(x, y) = x + y$.

Run PUQ:

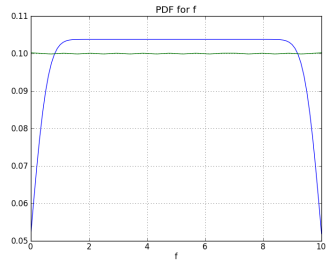
```
~/puq/examples/basic> puq start basic
Sweep id is 172906455

Processing <HDF5 dataset "f": shape (5,), type "<f8">
        Surface   = 1.0*x
        RMSE      = 0.00e+00 (0.00e+00 %)
```

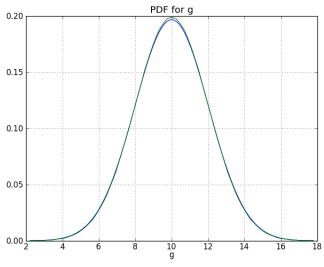
```
SENSITIVITY:
Var      u*            dev
-----------------------------
x    1.0000e+01    0.0000e+00
y    0.0000e+00    0.0000e+00

Processing <HDF5 dataset "g": shape (5,), type "<f8">
        Surface   = 1.0*y
        RMSE      = 8.95e-12 (7.26e-11 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
y    1.2317e+01    2.0001e-11
x    0.0000e+00    0.0000e+00

Processing <HDF5 dataset "h": shape (5,), type "<f8">
        Surface   = x + y
        RMSE      = 8.95e-12 (7.26e-11 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
y    1.2317e+01    2.0002e-11
x    1.0000e+01    0.0000e+00
```

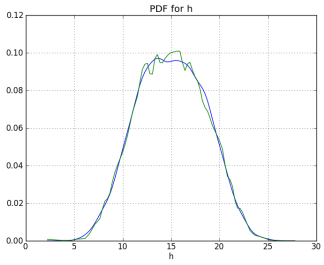
Note: The response of 'f' is '1.0*x', the response of 'g' is '1.0*y' and the response of 'h' is 'x + y'. These are exactly as expected.

Plot the PDFs. Use the 'l' and 'k' flags to see both linear and Gaussian KDE fits:

```
~/puq/examples/basic> puq plot -lk -f png
plotting PDF for f
plotting PDF for g
plotting PDF for h
~/puq/examples/basic> ls -l *.png
-rw-rw-r--. 1 mmh mmh 28623 Jul  8 23:03 pdf-f.png
-rw-rw-r--. 1 mmh mmh 39468 Jul  8 23:03 pdf-g.png
-rw-rw-r--. 1 mmh mmh 40094 Jul  8 23:03 pdf-h.png
```







Click on an image to see it larger. You can see that the 'f' PDF is a Uniform distribution between 0 and 10. And the 'g' PDF is a Normal with a mean of 10 and deviation of 2. These are exactly as expected.

1.2.15 Using Output PDFs as Inputs

Often the output of one simulation needs to be used as an input parameter in another simulation. Currently the easiest way to do this is by *Using PUQ Analyze*. You simply export the PDF as a json file, then in the next simulation, import that json file as an input PDF. This scenario is explained in great detail in the next section, *Handling Correlated PDFs*.

1.2.16 Scripting PUQ

All the examples so far have involved using the 'puq' command to start a sweep and again to analyze it. For more complicated jobs, you can script the whole process in Python.

Running Sweeps

Example scripts are in puq/examples/scripting. test1.py runs a level-1 Smolyak sweep, reads the response surface from the HDF5 file, plots the PDF, then extends the level to 2 and repeats.

```
#!/usr/bin/env python
"""
Example of scripting a PUQ run, extending it, and plotting PDFs
"""

import puq
import numpy as np
import matplotlib
matplotlib.use('tkagg', warn=False)
import matplotlib.pyplot as plt

# save to this filename
fname = 'script_test.hdf5'

# our input parameters
```

```
v1 = puq.Parameter('x', 'velocity1', mean=10, dev=2)
v2 = puq.Parameter('y', 'velocity2', mean=100, dev=3)

# run a level 1 Smolyak
uq = puq.Smolyak([v1, v2], 1)
sw = puq.Sweep(uq, puq.InteractiveHost(), './basic_prog.py')
sw.run(fname)

# get the response surface for output variable 'h'
rs = puq.hdf.get_response(fname, 'h')
# sample the response surface to get a PDF
val, samples = rs.pdf(fit=True, return_samples=True)
# plot it in blue
val.plot(color='b')
print 'Run 1: mean=%s   dev=%s\n' % (np.mean(samples), np.std(samples))

# not extend the run to a level 2
sw.extend()
sw.run()

# plot the PDF like before, but in green
rs = puq.hdf.get_response(fname, 'h')
val, samples = rs.pdf(fit=True, return_samples=True)
val.plot(color='g')
print 'Run 2: mean=%s   dev=%s' % (np.mean(samples), np.std(samples))

plt.show()
```

To run it, simply do:

```
./test1.py
```

or:

```
python test1.py
```

Reading Results

The previous example showed how to read a response surface and sample it to produce a pdf. The full list of python functions for reading and writing the HDF5 file is *hdf - Functions for Accessing HDF5 Files*

1.2.17 Handling Correlated PDFs

PUQ assumes input parameters are independent, but sometimes this is not true. In that case, you need to have either a Joint PDF, or a large number of samples drawn from that Joint PDF. In this section we'll show how to take two correlated output PDFs from one PUQ run and use them as inputs to another PUQ run.

These examples are from puq/examples/correlated. The first simulation, which creates obviously correlated outputs is c1_prog.py.

```
#!/usr/bin/env python

import optparse
from puqutil import dump_hdf5
import time

```

```
usage = "usage: %prog --x x --y y"
parser = optparse.OptionParser(usage)
parser.add_option("--x", type=float)
parser.add_option("--y", type=float)

(options, args) = parser.parse_args()
x = options.x
y = options.y

# need to produce two correlated output variables, f and g

f = x
g = x + y
z = f * g

dump_hdf5('f', f, "$f(x,y)=x+y$")
dump_hdf5('g', g, "$g(x,y)=x+y$")
dump_hdf5('z', z, "$z(x,y)=x(x+y)$")
```

This has three outputs, **f**, **g**, and **z**, which will be created as PDFs by *PUQ*.

The input script is

```
from puq import *

def run():
    x = UniformParameter('x', 'x', min=-5, max=5)
    y = UniformParameter('y', 'y', min=-5, max=5)

    host = InteractiveHost()
    uq = Smolyak([x,y], level=2)
    prog = TestProgram('./c1_prog.py')
    return Sweep(uq, host, prog)
```

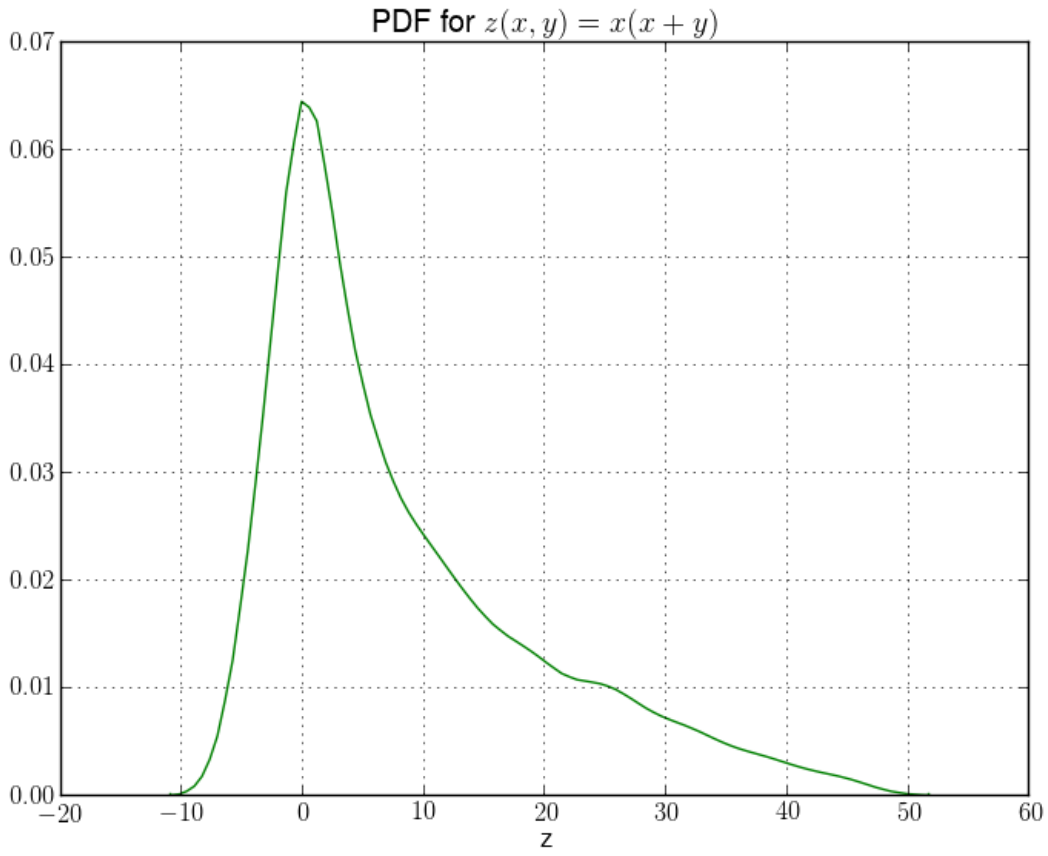
Lets run *PUQ* and create the PDFs:

```
~/puq/examples/correlated> puq start -f c1.hdf5 c1.py
Saving run to c1.hdf5

Processing <HDF5 dataset "f": shape (13,), type "<f8">
        Surface   = 1.0*x
        RMSE      = 1.07e-12 (1.07e-11 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    1.0000e+01    9.7391e-12
y    0.0000e+00    0.0000e+00

Processing <HDF5 dataset "g": shape (13,), type "<f8">
        Surface   = x + y
        RMSE      = 1.52e-12 (7.59e-12 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
y    1.0000e+01    9.7391e-12
x    1.0000e+01    9.7391e-12
```

```
Processing <HDF5 dataset "z": shape (13,), type "<f8">
        Surface   = x*(x + y)
        RMSE      = 7.59e-12 (1.52e-11 %)

SENSITIVITY:
Var      u*            dev
-----------------------------
x    5.5178e+01    6.8073e+01
y    2.5000e+01    3.5355e+01
```

As you can see, PUQ correctly generated each response surface. Let's look at **z**.

```
~/puq/examples/correlated> puq plot -v z c1.hdf5
```

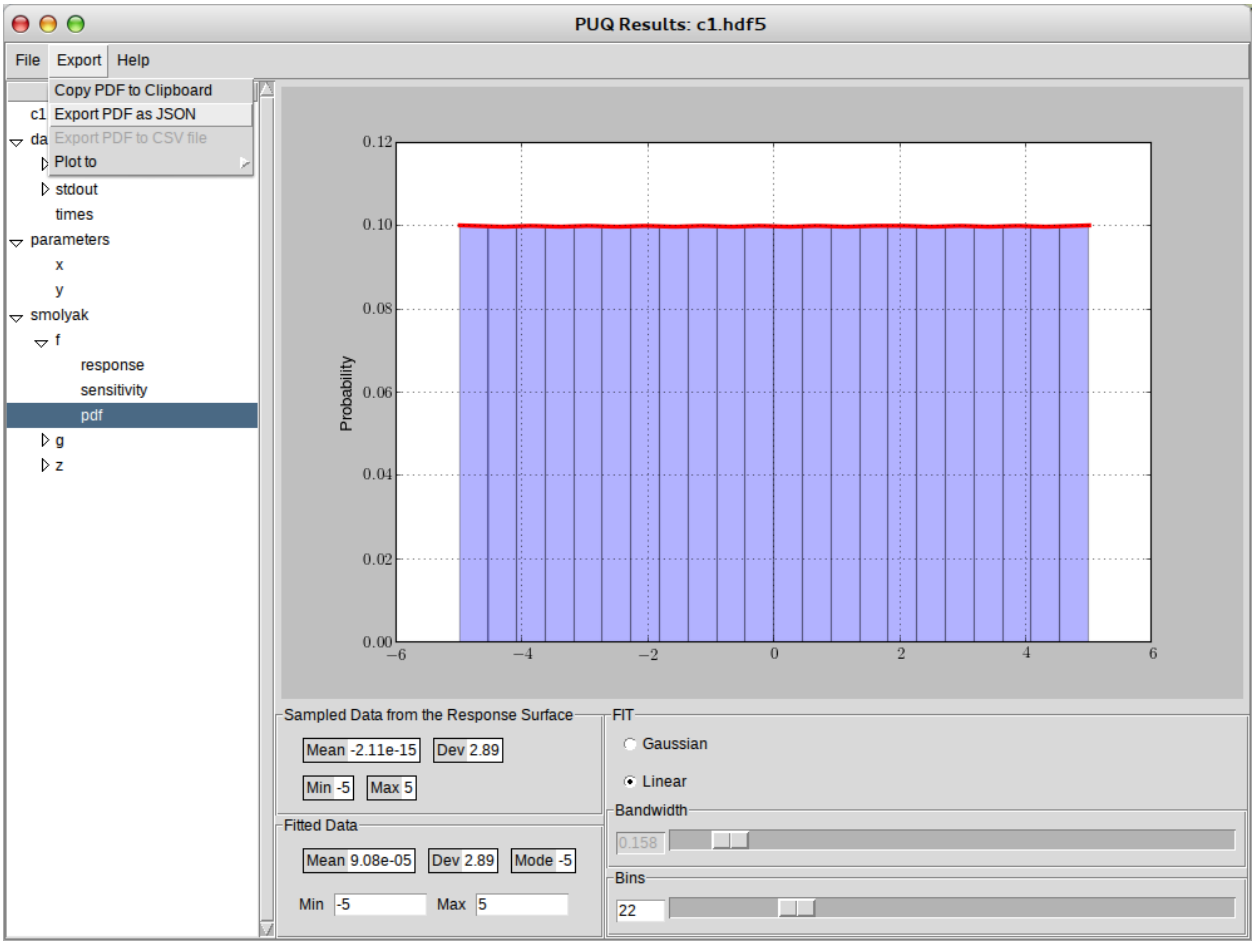


OK. So far things look good. The second simulation takes **f** and **g** as input PDFs and outputs **f*g**. So it should look the same as **z**. However, **f** and **g** are very correlated so it won't work as we expect. But before we try that, we need to save **f** and **g** to files.

```
~/puq/examples/correlated> puq analyze c1.hdf5
```

Save **f** and **g** to f.json and g.json.

You can look at c2_prog.py and verify it simply multiplies the two input parameters. Then take a look at c2.py.


PUQ Results: c1.hdf5
File
Export
Help
Copy PDF to Clipboard
Export PDF as JSON
Export PDF to CSV file
Plot to
c1
da
stdout
times
parameters
x
y
smolyak
f
response
sensitivity
pdf
g
z
Probability
0.12
0.10
0.08
0.06
0.04
0.02
0.00
−6
−4
−2
0
2
4
6
Sampled Data from the Response Surface
Mean -2.11e-15
Dev 2.89
Min -5
Max 5
Fitted Data
Mean 9.08e-05
Dev 2.89
Mode -5
Min -5
Max 5
FIT
Gaussian
Linear
Bandwidth
0.158
Bins
22

```
1  from puq import *
2
3  def run():
4      f = Parameter('f', 'f', pdf=LoadObj('f.json'), use_samples=True)
5      g = Parameter('g', 'g', pdf=LoadObj('g.json'), use_samples=True)
6
7      host = InteractiveHost()
8      uq = Smolyak([f,g], level=2)
9      prog = TestProgram('./c2_prog.py')
10     return Sweep(uq, host, prog)
```

For the first run, change lines 4 and 5 so use_samples is False. This is the default and will cause the input parameters to be treated as independent. Then run **puq**

```
~/puq/examples/correlated> puq start -f c2_indep.hdf5 c2.py
Saving run to c2_indep.hdf5

Processing <HDF5 dataset "k": shape (13,), type "<f8">
        Surface   = 1.0*f*g
        RMSE      = 7.92e-14 (7.96e-14 %)

SENSITIVITY:
Var      u*            dev
------------------------------
f    4.9775e+01    7.0000e+01
g    4.9498e+01    7.0000e+01
```

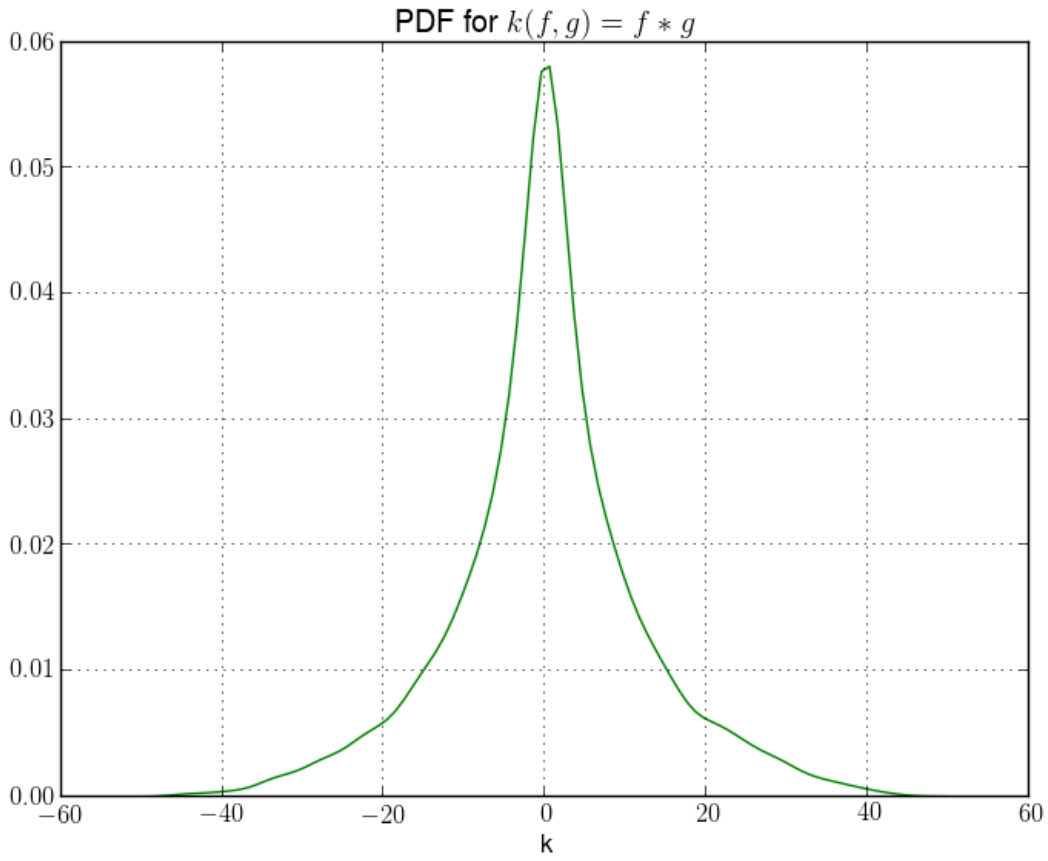
The response surface is correct (f*g). Looking at the output, **k**,

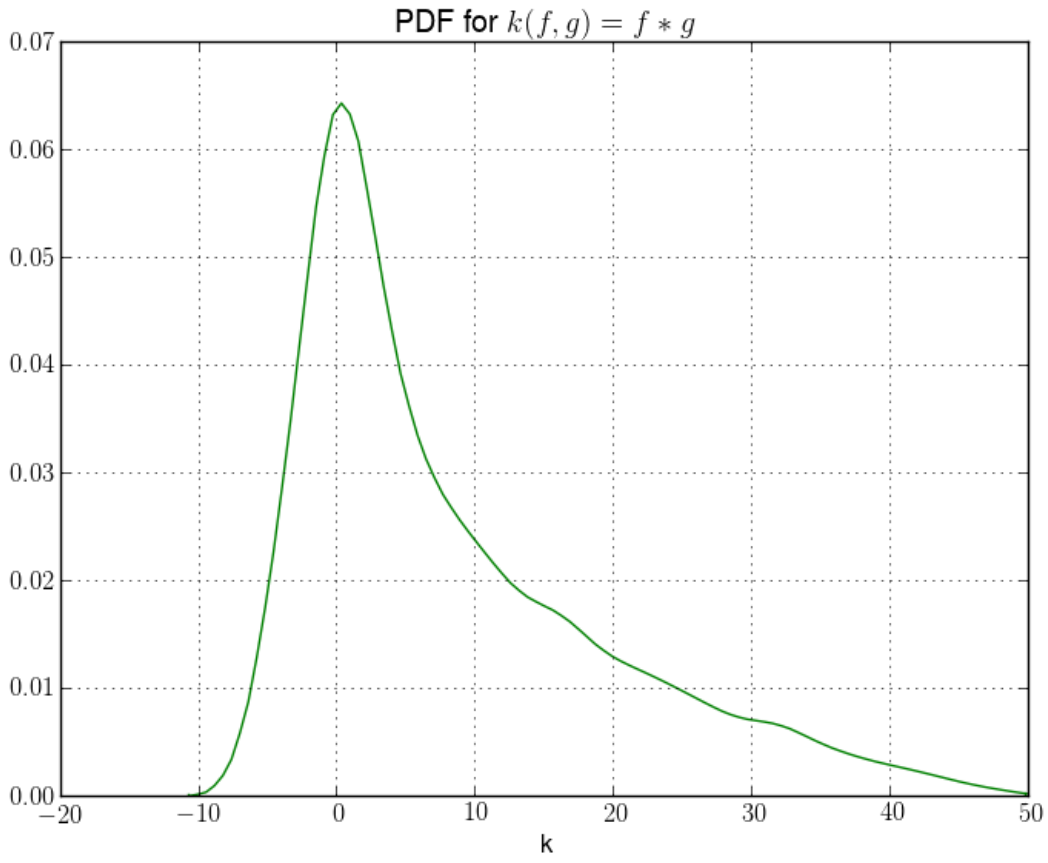
```
~/puq/examples/correlated> puq plot -v k c2_indep.hdf5
```

This looks nothing like the plot for **z**. Now let's try it with correlated input parameters. Change c2.py back so lines 4 and 5 have the use_samples arg set to True.

```
1  from puq import *
2
3  def run():
4      f = Parameter('f', 'f', pdf=LoadObj('f.json'), use_samples=True)
5      g = Parameter('g', 'g', pdf=LoadObj('g.json'), use_samples=True)
6
7      host = InteractiveHost()
8      uq = Smolyak([f,g], level=2)
9      prog = TestProgram('./c2_prog.py')
10     return Sweep(uq, host, prog)
```

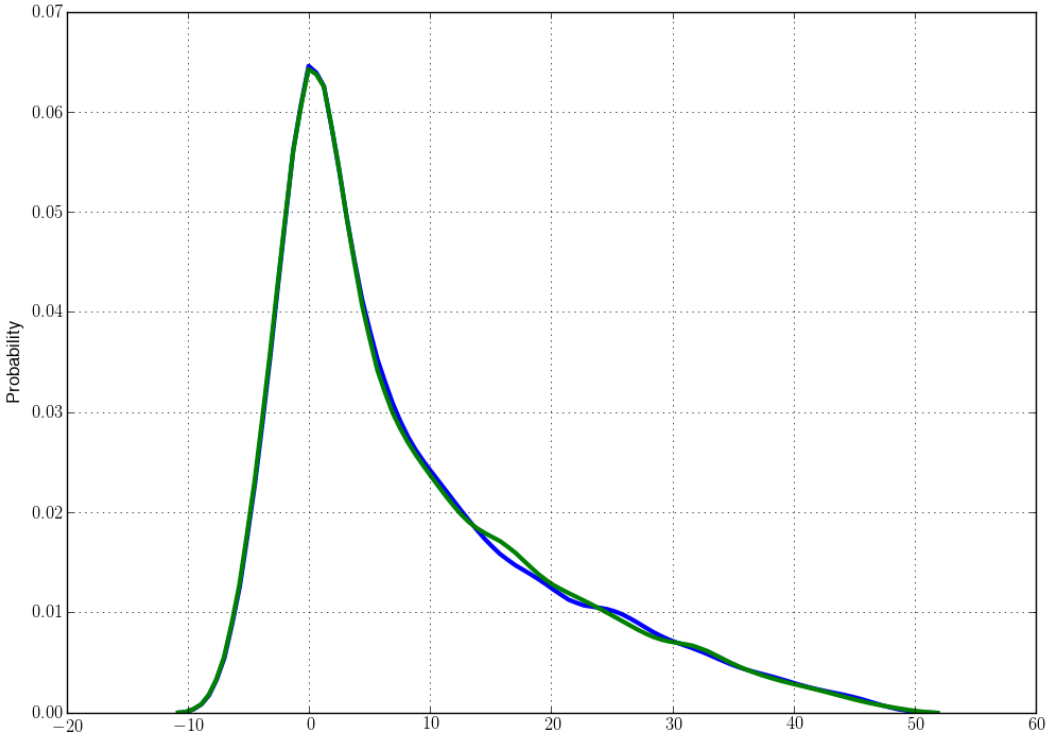
```
~/puq/examples/correlated> puq start -f c2.hdf5 c2.py
     Saving run to c2.hdf5

     Processing <HDF5 dataset "k": shape (13,), type "<f8">
             Surface   = 1.0*f*g
             RMSE      = 7.92e-14 (7.96e-14 %)

     SENSITIVITY:
     Var      u*            dev
     ------------------------------
     f    4.9775e+01    7.0000e+01
     g    4.9498e+01    7.0000e+01

     ~/puq/examples/correlated> puq plot -v k c2.hdf5
```


PDF for $k(f,g) = f * g$
0.06
0.05
0.04
0.03
0.02
0.01
0.00
−60
−40
−20
0
20
40
60
k


PDF for $k(f,g) = f * g$
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
−20
−10
0
10
20
30
40
50
k

This looks very much like the plot for **z** we got in the first section. For a better comparison, you can use *puq analyze* to save **z** and **k** to files, then do

```
~/puq/examples/correlated> puq read -c z.json k.json
```



1.3 Reference

1.3.1 calibrate

puq.**calibrate**(*params*, *caldata*, *err*, *func*, *weight=20*)

Calibrate a variable or variables.

Parameters

- **params** (*list*) – Input parameters.
- **caldata** (*list or array*) – Experimental output values.
- **err** (*float*) – Deviation representing the measurement error in caldata.
- **func** – Response function.

Returns A copy of **params** modified with the calibrated variables.

1.3.2 hdf - Functions for Accessing HDF5 Files

Convenience functions that read or write to the HDF5 file

puq.hdf.**data_description**(*hf*, *var*)
Returns the description of an output variable. If the description is empty, returns the variable name.

Parameters

- **hf** – An open HDF5 filehandle or a string containing the HDF5 filename to use.
- **var** – Output variable name.

puq.hdf.**get_output_names**(*hf*)
Returns a list of the output variables names in the HDF5 file.

Parameters **hf** – An open HDF5 filehandle or a string containing the HDF5 filename to use.

Returns A sorted list of the output variable names in the HDF5 file.

puq.hdf.**get_param_names**(*hf*)
Returns a list of the input parameter names in the HDF5 file.

Parameters **hf** – An open HDF5 filehandle or a string containing the HDF5 filename to use.

puq.hdf.**get_params**(*hf*)
Returns a list of arrays of input parameter values.

Parameters **hf** – An open HDF5 filehandle or a string containing the HDF5 filename to use.

puq.hdf.**get_response**(*hf*, *var*)
Returns the response function for an output variable.

Parameters

- **hf** – An open HDF5 filehandle or a string containing the HDF5 filename to use.
- **var** – Output variable name.

puq.hdf.**get_result**(*hf*, *var=None*)
Returns an array containing the values of the output variable.

Parameters

- **hf** – An open HDF5 filehandle or a string containing the HDF5 filename to use.
- **var** – Output variable name. Only required if there is more than one output variable.

Returns An array

Raises `ValueError` – if **var** is not found or **var** is None and there are multiple output variables.

puq.hdf.**param_description**(*hf*, *var*)
Returns the description of an input variable. If the description is empty, returns the variable name.

Parameters

- **hf** – An open HDF5 filehandle or a string containing the HDF5 filename to use.
- **var** – Input parameter name.

puq.hdf.**set_result**(*hf*, *var*, *data*, *desc=''*)
Sets values of the output variable in the HDF5 file. Writes array to '/output/data/**var**'.

Parameters

- **hf** – An open HDF5 filehandle or a string containing the HDF5 filename to use.

- **var** – Output variable name.
- **data** – Array of data to write.
- **desc** – Description.

1.3.3 Hosts

class puq.**InteractiveHost**(*cpus=1*, *cpus_per_node=0*)

Create a host object that runs all jobs on the local CPU.

Parameters

- **cpus** – Number of cpus each process uses. Default=1.
- **cpus_per_node** – How many cpus to use on each node. Default=all cpus.

class puq.**PBSHost**(*env*, *cpus=0*, *cpus_per_node=0*, *qname='standby'*, *walltime='1:00:00'*, *modules=''*, *pack=1*, *qlimit=200*)

Queues jobs using the PBS batch scheduler. Supports Torque and PBSPro.

Parameters

- **env** (*str*) – Bash environment script (.sh) to be sourced.
- **cpus** (*int*) – Number of cpus each process uses. Required.
- **cpus_per_node** (*int*) – How many cpus to use on each node. Required.
- **qname** (*str*) – The name of the queue to use.
- **walltime** (*str*) – How much time to allow for the process to complete. Format is HH:MM:SS. Default is 1 hour.
- **modules** (*list*) – Additional required modules. Default is none.
- **pack** (*int*) – Number of sequential jobs to run in each PBS script. Default is 1.
- **qlimit** (*int*) – Max number of PBS jobs to submit at once. Default is 200.

1.3.4 Parameters

class puq.**Parameter**(**args*, ***kargs*)

Superclass for all Parameter subclasses. For backwards compatibility it can be called directly and will return the proper subclass.

- `NormalParameter` is returned if *dev* is set.
- `UniformParameter` is returned if either *min* or *max* is set.
- `CustomParameter` is returned if *pdf* or *caldata* is set.

Parameters

- **name** – Name of the parameter. This should be a short name, like a variable.
- **description** – A longer description of the parameter.
- **kargs** – Keyword args defined by the distribution.

See Also:

`NormalParameter`, `UniformParameter`, `CustomParameter`

class puq.**CustomParameter**(*name*, *description*, ***kwargs*)
Class implementing a Parameter with a Custom distribution.

Parameters

- **name** – Name of the parameter. This should be a short name, like a variable.
- **description** – A longer description of the parameter.
- **kwargs** – Keyword args. Valid args are:

Arg	Description
pdf	PDF
use_samples	Use data samples attached to pdf (if available)

class puq.**ExponParameter**(*name*, *description*, ***kwargs*)
Class implementing a Parameter with an Exponential distribution.

Parameters

- **name** – Name of the parameter. This should be a short name, like a variable.
- **description** – A longer description of the parameter.
- **kwargs** – Keyword args. Valid args are:

Arg	Description
rate	The rate parameter. Must be > 0.

See Also:

ExponPDF

class puq.**NormalParameter**(*name*, *description*, ***kwargs*)
Class implementing a Parameter with a Normal distribution.

Parameters

- **name** – Name of the parameter. This should be a short name, like a variable.
- **description** – A longer description of the parameter.
- **kwargs** – Keyword args. Valid args are:

Arg	Description
mean	The mean of the distribution
dev	The standard deviation

class puq.**RayleighParameter**(*name*, *description*, ***kwargs*)
Class implementing a Parameter with a Rayleigh distribution.

Parameters

- **name** – Name of the parameter. This should be a short name, like a variable.
- **description** – A longer description of the parameter.
- **kwargs** – Keyword args. Valid args are:

Arg	Description
scale	The scale. Must be > 0.

See Also:

RayleighPDF

class puq.**UniformParameter**(*name*, *description*, ***kwargs*)
Class implementing a Parameter with a Uniform distribution.

Parameters

- **name** – Name of the parameter. This should be a short name, like a variable.
- **description** – A longer description of the parameter.
- **kwargs** – Keyword args. Valid args are:

Property	Description
mean	The mean of the distribution
max	The maximum
min	The minimum

You **must** specify two of the above properties. If you give all three, they will be checked for consistency. $mean = (min + max)/2$

class puq.**WeibullParameter**(*name*, *description*, ***kwargs*)
Class implementing a Parameter with a Weibull distribution.

Parameters

- **name** – Name of the parameter. This should be a short name, like a variable.
- **description** – A longer description of the parameter.
- **kwargs** – Keyword args. Valid args are:

Arg	Description
shape	The shape. Must be > 0.
scale	The scale. Must be > 0.

See Also:

WeibullPDF

class puq.**PSweep**(*iteration_cb=None*)

run(*sweep*)
Gets the parameters then adds the command plus parameters to the host object job list. Then tells the host to run. After all jobs have completed, collect the data and call analyze(). If iteration_cb is defined, call it.

Returns True on success.

1.3.5 PDF

class puq.**PDF**(*xvals*, *yvals*)
Create a PDF (Probability Density Function) object.

Use this to create a PDF object given a list or array of x values the corresponding PDF values.

Parameters

- **xvals** (*1D array or list*) – x values
- **yvals** (*1D array or list*) – values for PDF(x)

__add__(*b*)
Add two PDFs, returning a new one.

__div__(*b*)
Divide two PDFs, returning a new PDF

__mul__(*b*)
Multiply two PDFs, returning a new PDF

__sub__(*b*)
Subtract two PDFs, returning a new PDF

cdf(*arr*)
Computes the Cumulative Density Function (CDF) for some values.

Parameters **arr** – Array of x values.

Returns Array of cdf(x).

ds(*num*)
Generates a descriptive sample for this distribution.

The order of the numbers in the array is random, so it can be combined with other arrays to form a latin hypercube. This method is used by LHS.

Parameters **num** – Number of samples to generate.

Returns 1D array of length *num*.

ds1(*num*)
Generates a descriptive sample in [-1,1] for this distribution.

The order of the numbers in the array is random, so it can be combined with other arrays to form a latin hypercube. Note that this *can* return values outside the range [-1,1] for distributions with long tails. This method is used by puq.Smolyak.

Parameters **num** – Number of samples to generate.

Returns 1D array of length *num*.

lhs(*num*)
Latin Hypercube Sample for this distribution.

The order of the numbers in the array is random, so it can be combined with other arrays to form a latin hypercube. This method is used by LHS.

Parameters **num** – Number of samples to generate.

Returns 1D array of length *num*.

lhs1(*num*)
Latin Hypercube Sample in [-1,1] for this distribution.

The order of the numbers in the array is random, so it can be combined with other arrays to form a latin hypercube. Note that this can return values outside the range [-1,1] for distributions with long tails. This method is used by puq.Smolyak.

Parameters **num** – Number of samples to generate.

Returns 1D array of length *num*.

mode
Find the mode of the PDF. The mode is the x value at which pdf(x) is at its maximum. It is the peak of the PDF.

pdf(*arr*)
Computes the Probability Density Function (PDF) for some values.

Parameters **arr** – Array of x values.

Returns Array of pdf(x).

plot (*color=''*, *fig=False*)
Plot a PDF.

Parameters

- **color** (*String.*) – Optional color for the plot.
- **fig** (*Boolean.*) – Create a new matplotlib figure to hold the plot.

Returns A list of lines that were added.

ppf (*arr*)
Percent Point Function (inverse CDF)

Parameters **arr** – Array of x values.

Returns Array of ppf(x).

random (*num*)
Generate random numbers fitting this parameter's distribution.

This method is used by `MonteCarlo`.

Parameters **num** – Number of samples to generate.

Returns 1D array of length *num*.

range
The range for the PDF. For PDFs with long tails, it is truncated to 99.99% by default. You can customize this by setting options['pdf']['range'].

Returns A tuple containing the min and max.

srange
The small range for the PDF. For PDFs with long tails, it is truncated to 99.8% by default. You can customize this by setting options['pdf']['srange'].

Returns A tuple containing the min and max.

puq.**ExponPDF** (*rate*)
Creates Exponential Probability Density Function.

Parameters **rate** – The rate parameter for the distribution. Must be > 0.

Returns A PDF object

See http://en.wikipedia.org/wiki/Exponential_distribution

puq.**NormalPDF** (*mean*, *dev*, *min=None*, *max=None*)
Creates a normal (gaussian) Probability Density Function.

Parameters

- **mean** – The mean.
- **dev** – The standard deviation.
- **min** – A minimum value for the PDF (default None).
- **max** – A maximum value for the PDF (default None).

Returns A PDF object

For the normal distribution, you must specify **mean** and **dev**.

Example

```
>>> n = NormalPDF(10,1)
>>> n = NormalPDF(mean=10, dev=1)
>>> n = NormalPDF(mean=10, dev=1, min=10)
```

puq.**RayleighPDF**(*scale*)
Creates Rayleigh Probability Density Function.

Parameters **scale** – The scale. Must be > 0.

Returns A PDF object

See http://en.wikipedia.org/wiki/Rayleigh_distribution

puq.**TrianglePDF**(*min*, *mode*, *max*)
Creates a triangle Probability Density Function.

See http://en.wikipedia.org/wiki/Triangular_distribution

Parameters

- **min** – The minimum value
- **mode** – The mode
- **max** – The maximum value

Returns A PDF object

You can enter the parameters in any order. They will be sorted so that the mode is the middle value.

puq.**UniformPDF**(*min=None*, *max=None*, *mean=None*)
Creates a uniform Probability Density Function.

Parameters

- **min** – The minimum value
- **max** – The maximum value
- **mean** – The mean value

Returns A PDF object

For the uniform distribution, you must specify two of (min, max, and mean). The third parameter will be calculated automatically.

Example

```
>>> u = UniformPDF(10,20)
>>> u = UniformPDF(min=10, max=20)
>>> u = UniformPDF(min=10, mean=15)
```

puq.**WeibullPDF**(*shape*, *scale*)
Creates Weibull Probability Density Function.

Parameters

- **shape** – The shape. Must be > 0.
- **scale** – The scale. Must be > 0.

Returns A PDF object

See http://en.wikipedia.org/wiki/Weibull_distribution

puq.**ExperimentalPDF**(*data*, *min=None*, *max=None*, *fit=False*, *bw=None*, *nbins=0*, *prior=None*, *error=None*, *force=False*)

Create an experimental PDF.

An experimental PDF is derived from the results of an experiment or measurement of some parameter. It has actual data attached to it. That data is then used to create a PDF by one of three different methods.

The PDF can built by binning the data and linearly interpolating, using a Gaussian KDE, or using Bayesian Inference.

Parameters

- **data** (*Array of scalars*) – Our quantity of interest.
- **nbins** (*int*) – Number of bins (used if fit is false). Default is 2*IQR/n^(1/3) where IQR is the interquartile range of the data.
- **fit** (*True or "Gaussian"*) – Use Gaussian KDE (default=False)
- **bw** (*string or float. String must be 'scott' or 'silverman'*) – Bandwidth for Gaussian KDE (default=None)
- **prior** (*PDF*) – Prior PDF to use for Bayesian Inference. [default=None (uninformative)]
- **error** (*PDF. Typically a NormalPDF with a mean of 0.*) – Error in the data. For example, the measurement error. Required for Bayesian.

puq.**HPDF**(*data*, *min=None*, *max=None*)

Histogram PDF - initialized with points from a histogram.

This function creates a PDF from a histogram. This is useful when some other software has generated a PDF from your data.

Parameters

- **data** (*2D numpy array*) – A two dimensional array. The first column is the histogram interval mean, and the second column is the probability. The probability values do not need to be normalized.
- **min** – A minimum value for the PDF range. If your histogram has values very close to 0, and you know values of 0 are impossible, then you should set the ***min*** parameter.
- **max** – A maximum value for the PDF range.

Returns A PDF object.

puq.**NetPDF**(*addr*)

Retrieves a PDF from a remote address.

Parameters **addr** – URI. PDF must be stored in JSON format

Returns A PDF object

Example

```
>>> u = NetPDF('http://foo.com/myproject/parameters/density')
```

1.3.6 puqutil - Output Functions for PUQ

Output Format

The purpose of these functions is to provide a simple way to output in the format PUQ expects.

PUQ expects to see output values written to standard output. It captures all standard output, and scans it looking for a pattern at the beginning of every line. Lines which don't match this pattern are ignored.

HDF5:{'name': n, 'desc': d, 'value':v}:5FDH

where

n Name of the variable

d Description of the variable

v Value of the variable

Python Functions

```
from puqutil import dump_hdf5

dump_hdf5('v', v, 'velocity of the swallow')
```

puqutil.**dump_hdf5**(*name*, *val*[, *desc=''*])

Writes data to stdout with formatting so PUQ can automatically recognize it and save it.

Parameters

- **name** – Name of the variable
- **val** (*integer, float, or array*) – Value of the variable
- **desc** – A description of the variable. Saved as an attribute for the variable in the HDF file. Used as labels in plots. Default is an empty string.

Functions for C/C++

These are included in "dump_hdf5.h"

void **dump_hdf5_d**(char * *name*, double *val*, char **desc*)

Writes data to stdout with formatting so PUQ can automatically recognize it and save it.

Parameters

- **name** (*C string*) – Name of the variable
- **val** (*double*) – The value of the variable
- **desc** (*C string*) – A description of the variable. Saved as an attribute for the variable in the HDF file. Used as labels in plots. Default is an empty string.

void **dump_hdf5_l**(char * *name*, long *val*, char **desc*)

Writes data to stdout with formatting so PUQ can automatically recognize it and save it.

Parameters

- **name** (*C string*) – Name of the variable
- **val** (*long integer*) – The value of the variable
- **desc** (*C string*) – A description of the variable. Saved as an attribute for the variable in the HDF file. Used as labels in plots. Default is an empty string.

1.3.7 Smolyak UQ Method

class puq.**Smolyak**(*params*, *level*, *iteration_cb=None*)
Class implementing gPC using Smolyak Sparse Grids

Parameters

- **params** – Input list of Parameters.
- **level** – Polynomial degree for the response function.

If *level* is set too low, then the response surface will not precisely fit the observed responses. The goodness of the fit is calculated as by RMSE. A perfect fit will have RMSE=0.

1.3.8 Sweep Reference

class puq.**Sweep**(*psweep*, *host*, *prog*, *caldata=None*, *calerr=None*, *description=''*)
Creates an object that contains all the information about a parameter sweep.

Parameters

- **psweep** – Parameter Sweep object. See PSweep.
- **host** – Host object. See Host.
- **prog** – TestProgram object: See TestProgram.
- **caldata(array** – Experimental data for calibration. Optional.
- **calerr** (*float*) – Measurement error in the experimental data.
- **description** (*string*) – Optional description of this run.

analyze(*verbose=False*)
Collects the output from all the jobs into an HDF5 file. Parses any tagged data in the output and puts it in the /data group in the HDF5 file.

collect_data(*hf=None*)
Collects data from captured stdout files and puts it in arrays in 'output/data'. Returns True on success.

run(*fn=None*, *overwrite=False*)
Calls PSweep.run() to run all the jobs in the Sweep. Collect the data from the outputs and call the PSweep analyze method. If the PSweep method has an iterative callback defined, call it, otherwise return.

Parameters

- **fn** (*string*) – HDF5 filename for output. '.hdf5' will be appended to the filename if necessary. If fn is None, a filename will be generated starting with "sweep_" followed by a timestamp.
- **overwrite** (*boolean*) – If True and fn is not None, will silently overwrite any previous files of the same name.

Returns True on success.

1.3.9 TestProgram

class puq.**TestProgram**(*name=''*, *exe=''*, *newdir=False*, *infiles=''*, *desc=''*, *outfiles=''*)
Class implementing a TestProgram object representing the simulation to run.

Parameters

- **name** (*string*) – Name of the program. This is the executable if no *exe* is defined.
- **exe** (*string*) – An executable command template to run. Strings of the form '$var' are replaced with the value of *var*. See python template strings (http://docs.python.org/2/library/string.html#template-strings)
- **desc** – Optional description of the test program.
- **newdir** (*boolean*) – Run each job in its own directory. Necessary if the simulation generates output files. Default is False.
- **infiles** (*list*) – If *newdir* is True, then this is an optional list of files that should be copied to each new directory.
- **outfiles** (*list*) – An optional list of files that will be saved into the HDF5 file upon completion. The files will be in /output/jobs/n where 'n' is the job number.

Example1:

```
p1 = UniformParameter('x', 'x', min=-2, max=2)
p2 = UniformParameter('y', 'y', min=-2, max=2)

prog = TestProgram('./rosen_prog.py', desc='Rosenbrock Function')

# or, the equivalent using template strings

prog = TestProgram(exe='./rosen_prog.py --x=$x --y=$y',
  desc='Rosenbrock Function')
```

Example2:

```
# Using newdir and infiles. Will run each job in a new directory
# with a copy of all the infiles.

prog = TestProgram('PM2', newdir=True, desc='MPM Scaling',
  infiles=['pm2geometry', 'pm2input', 'pmgrid_geom.nc',
  'pmpart_geom.nc'])
```

1.3.10 PSweep Reference

class puq.**LHS** (*params*, *num*, *ds=False*, *response=True*, *iteration_cb=None*)
Class implementing Latin hypercube sampling (LHS).

Parameters

- **params** – Input list of Parameters.
- **num** – Number of samples to use.
- **ds** (*boolean*) – Use a modified LHS which always picks the center of the Latin square.
- **response** (*boolean*) – Generate a response surface using the sample points.
- **iteration_cb** (*function*) – A function to call after completion.

class puq.**MonteCarlo** (*params*, *num*, *response=True*, *iteration_cb=None*)

class puq.**Smolyak** (*params*, *level*, *iteration_cb=None*)
Class implementing gPC using Smolyak Sparse Grids

Parameters

- **params** – Input list of Parameters.

- **level** – Polynomial degree for the response function.

If *level* is set too low, then the response surface will not precisely fit the observed responses. The goodness of the fit is calculated as by RMSE. A perfect fit will have RMSE=0.

1.4 Examples

The PUQ examples are located in the puq/examples directory. Many of them are used as part of the tutorial in the *User's Guide and Tutorial*. Here you may be able to find more details about a particular example.

1.4.1 rosen

The rosen test directory has examples of the *Rosenbrock* function which is $f(x, y) = 100(y - x^2)^2 + (1 - x)^2$

Because the Rosenbrock function is a simple polynomial, the Smolyak method provides the best approach. However, Monte Carlo and Latin Hypecube examples are provided for comparison.

Different test programs which implement the Rosenbrock function are provided as examples of using Python, C, and Matlab code.

Contents

rosen.py Example using `Smolyak`

rosen_c.py Example using rosen_cprog with `Smolyak`

rosen_ml.py Example using rosen.m with `Smolyak`

rosen_mc.py Example using `MonteCarlo`

rosen_lhs.py Example using `LHS`

---

rosen_prog.py Test program written in Python

rosen_cprog.c Test program written in C.

rosen.m Test program for Matlab or Octave.

Makefile Makefile for rosen_cprog

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p

INDEX

T

U

W