

Surfpack Version 1.0 User's Manual

Anthony A. Giunta¹, Mark D. Richards², Eric C. Cyr²
Laura P. Swiler¹, Shane L. Brown¹, Michael S. Eldred¹

March 15, 2006

¹Sandia National Laboratories*
P.O. Box 5800, Mail Stop 0828
Albuquerque, NM 87185-0828 USA
Email: surfpack@scico.sandia.gov
Web: <http://endo.sandia.gov/Surfpack>

²University of Illinois at Urbana-Champaign
Dept. of Computer Science
201 North Goodwin Ave.
Urbana, IL 61801-2302 USA
Web: <http://cs.engr.uiuc.edu>

Copyright 2006, Sandia National Laboratories

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

1 Overview

Surfpack is a collection of surface-fitting methods and accompanying metrics to evaluate or predict the quality of the generated surfaces. The concept of creating a global approximation or “fit” from a collection of data samples is utilized in many scientific disciplines, but the nomenclature varies widely from field to field. The results from the application of such methods are commonly called empirical models, response surfaces, surrogate models, function approximations, or neural networks. Many different algorithms have been developed to generalize from a set of data; these algorithms have different strengths and weaknesses. The goals of Surfpack are

1. to give users the option to use any of several methods, depending on the nature of the specific application; and
2. to put data-fitting methods that are commonly used in various disciplines into a common framework, where their properties can be more easily compared and analyzed.

Surfpack’s API includes a small set of commands, centered on the following general operations:

- **Prepare a data set for use.** This typically involves reading a formatted text file from disk. Alternatively, the user may specify upper- and lower-bounds along one or more dimensions and generate a set of data points from those boundaries (either a grid or a set of Monte Carlo samples).
- **Create an empirical model from a set of data.** The user may choose one of several algorithms to create the surface approximation: Least-squares regression using polynomials, Multivariate Adaptive Regression Splines (MARS), Kriging interpolation, Artificial Neural Networks, or Radial Basis Function Networks.
- **Evaluate an empirical model on a set of data.** For the non-interpolating algorithms (e.g. polynomial regression), it may be of interest to evaluate the model at the same data sites that were used to generate it, to see how closely the model fits the data. All of the algorithms all the user to evaluate the model at other data points where the true function value is not available.
- **Obtain measures of the “goodness of fit” of the model.** Surfpack supports metrics such as mean squared error or maximum absolute error for data sets where the true function values are known. Cross-validation metrics (e.g. PRESS) are also available for situations where all of the known data points for the function that is being approximated were used to create the empirical model.
- **Save the data and/or empirical models for future use.** Data can be saved for later use, e.g., with a plotting package. The approximating

surfaces themselves may also be saved, so that a user can evaluate the model on a data set at a later time without having to recompute it.

2 Installation

Surfpack is being developed primarily under Linux and is targeted to all flavors of UNIX. Platforms on which Surfpack has been successfully built include Linux, SunOS, IRIX, OSF, and AIX. Surfpack does not currently build under Windows or Mac OS X; these may be supported in future releases.

2.1 Requirements

- C++. A majority of the source code for Surfpack is written in C++. Surfpack makes heavy use of the standard C and C++ libraries, including the standard template libraries.
- Fortran 77. The MARS and Kriging algorithms are implemented in Fortran. The Kriging algorithm uses the constrained optimizer CONMIN, which is also written in Fortran.
- BLAS. Many of the data-fitting algorithms rely on the Basic Linear Algebra routines to perform rudimentary linear algebra operations.
- LAPACK. Surfpack makes use of the following LAPACK driver routines: dgetrf, dgetri, dgels, dggls. Implementations of LAPACK across different platforms seem to vary in terms of how many of these algorithms they support. Therefore, source for these functions and their dependences is included in the Surfpack distribution. However, BLAS and LAPACK routines are usually heavily optimized for each platform; performance is likely to be best if pre-compiled, native versions of these functions are available.

2.2 Options

- CPP Unit. A suite of unit tests is available on platforms where CPP Unit has been built.
- Flex and Bison. Surfpack may be used as a library or as a stand-alone program. The stand-alone executable requires Flex and Bison (a lexical analyzer generator and parser generator, respectively), which are freely available on the Internet.

2.3 Standard build

The Surfpack build system uses the GNU autotools: autoconf, automake, and libtool. The autotools do not need to be installed to build Surfpack, but they should be used by developers who wish to extend Surfpack. To build Surfpack

from source, first obtain the distribution tarball `surfpack-0.1.tar.gz`. From the directory containing this file, execute the following commands:

1. `gunzip surfpack-0.1.tar.gz`
2. `tar xf surfpack-0.1.tar`
3. `cd surfpack-0.1`
4. `./configure` (or `./configure --prefix='pwd'`, see text below)
5. `make`
6. `make install`

Users with access to Surfpack's CVS repository may checkout the source directly instead of downloading the gzipped distribution tar-file. In this case, the user must manually run the GNU autotools to create the configure script and `Makefile.in` files. The appropriate sequence of commands would be

1. `cvs checkout Surfpack`
2. `cd Surfpack`
3. `autoreconf --install`
4. items 4–6 above

By default, the Surfpack library is installed in `/usr/local/lib`. If appropriate Flex and Bison distributions are available, the stand-alone executable, named `surfpack`, will be installed in `/usr/local/bin`. Adding files to these directories generally requires super-user privileges. To install Surfpack in another location, use the `--prefix` option to configure. For example, the command

```
./configure --prefix=/home/userid
```

would lead to the installation of the Surfpack library and executable as `/home/userid/lib/libsurfpack.a` and `/home/userid/bin/surfpack` respectively. If these files are not present at the conclusion of the build, see section xx for installation troubleshooting tips.

2.4 Advanced options

Most deviations from the standard build process involve special arguments to configure.

2.5 Testing the installation

Surfpack is distributed with its test suite. The unit testing suite utilizes the CPP Unit libraries, available at <http://cppunit.sourceforge.net/cppunit-wiki>, which must be installed in the user's machine if the tests are to be executed. If the CPP Unit libraries are installed in a location that is not automatically searched by

the compiler, the full path of the libraries should be specified as an argument to configure: `./configure --with-cppunit-prefix=/home/userid/cppunit` To run the test suite, type `make check` at the command line, after the successful execution of `make` or `make install`. CPP Unit summarize the results of the tests.

If Surfpack is built as a stand-alone executable (as it is by default), additional tests are available in the examples subdirectory.

2.5.1 Specifying your compiler

By default, configure will use the first available C++ compiler from the following list: `xlc`, `CC`, `cxx`, `c++`, `g++`. To override this selection process, use the `CXX` option. The command

```
./configure CXX=g++
```

will cause `g++` to be used, regardless of the presence of any other C++ compilers. Likewise, the default search order for a Fortran 77 compiler is `xlf`, `f77`, `g77`. To override the default, use configure's `F77` option.

2.5.2 Specifying non-standard locations for libraries

Surfpack requires the presence of libraries for BLAS and LAPACK, as well as the standard C, C++, and Fortran 77 libraries. Most compilers search for libraries based on the contents of certain shell environment variables (e.g. `LIB_PATH`, `LD_LIBRARY_PATH`). Some libraries may be present on a particular system, but if the appropriate environment variables are not set correctly, they will not be found during the Surfpack build process. If the names of the library path environment variables are known, the directories containing the libraries on which Surfpack depends should be added to the library search path. Otherwise, additional directories may be added to the linker's search path by using configure's `LD_FLAGS` argument. Directories in the search path should be prepended by `"-L"` and separated by spaces. The entire list should be enclosed in quotation marks `" "`, e.g., `./configure LD_FLAGS="-L/home/userid/lib -L/usr/pub/lib"`

2.5.3 Building with Dakota

Surfpack has its origins in the DAKOTA software toolkit, and Surfpack can be built so that it serves as a library in DAKOTA. See the DAKOTA web page (<http://endo.sandia.gov/DAKOTA>), and the `INSTALL` notes that come with the DAKOTA source code for details on how to link Surfpack and DAKOTA.

2.6 Troubleshooting

This section enumerates possible causes and solutions for build failures.

2.6.1 Configuration failures

If the configure script terminates with an error, then appropriate makefiles have not been created and Surfpack cannot be built. A failure during the configure script generally means that some essential program or feature was not detected (e.g., a C++ compiler). If configure fails to find features that are believed to be present on the system, check to make sure that these features are located in standard places or specify them explicitly as arguments to configure. See section xx for a list of commonly used arguments to configure. If required packages are not present on the system, these can usually be downloaded for free from the Internet. For example, BLAS and LAPACK are available from netlib.org; Flex and Bison can be downloaded from gnu.org.

2.6.2 Failures during make install

The most common problem here is insufficient permissions. By default, the Surfpack library and executable are installed under /usr/local, which is usually writeable only by the superuser. To install Surfpack in an alternative directory, use the `-prefix=/some/other/path` argument to configure.

3 Getting Started

This chapter outlines the basic commands in the Surfpack API. These commands may be executed interactively from the Surfpack command prompt (by executing the 'surfpack' command with no arguments) or, more commonly, from a Surfpack script file. To execute a script file, execute the 'surfpack' command with the name of the script file as a command-line argument, e.g. `./surfpack scriptfile.spk`. General conventions are presented in section xx. The various Surfpack commands are discussed sections xx-xx. The chapter concludes with a series of detailed examples that show how the commands can be combined in an application.

3.1 Conventions

Surfpack commands consist of a (capitalized command name followed by a comma-delimited list of arguments in square brackets []). Whitespace is ignored. Comments begin with a '#' symbol and continue until the end of the line. Lines beginning with '!' are passed along to the underlying shell. (The leading '!' is first removed.) This allows the user to, for example, echo information to the terminal or do pre- or post-processing on the data files. Multiline shell commands may be delimited by /* and */.

The Surfpack interpreter internally maintains three types of variables: axes, data sets, and surfaces. Axes variables are created using the CreateAxes command. Data sets may be created by the LoadData, GridPoints, or MonteCarloSample commands. Surfaces are created with the LoadSurface or CreateSurface commands. Any command that creates an axes, data, or surface variable

must have a `name` argument, which gives the variable a name by which it can be referred to in future commands. When an existing variable is used in a subsequent command, it is designated by an argument which names its type (axes, data, or surface).

```
# Read an axes file from disk. Argument 'name' tells Surfpack
# how this axes variable will be referred to in future commands
CreateAxes[name = boundaries_2d, file = '2d.axb']

# Load a data file from disk. This data will be used later in
# an evaluate command.
LoadData[name = test_rosen, file = 'dense_rosenbrock_2d.txt']

# Use the axes and the Rosenbrock test function to create a new data set.
# Since a variable is being created-- a data set in this case-- the 'name'
# argument must be present. The axes variable created in the previous
# command is denoted by an 'axes' argument that specifies the existing
# variable's name.
GridPoints[name = rosenbrock_2d, axes = boundaries_2d,
           test_function = rosenbrock]

# Use the data set to create a new surface. Note the use of the 'name'
# argument to create a new surface variable, and the 'data' argument which
# refers to a data set created in the previous command.
CreateSurface[name = krig_rosen, data = rosenbrock_2d, type = kriging]

# Evaluate the surface on a different set of points. The 'data' argument
# refers to a data set variable that must have been created in a previous
# command (See LoadData command above). Here, the surface will be evaluated
# at each point in the data set, and the results will be appended as a new
# response variable for that data set.
Evaluate[surface = krig_rosen, data = test_rosen]
```

3.1.1 Command Arguments

Each argument has the following format: `argument_name = argument_value`. An argument name is an identifier—a letter followed by a combination of letters, digits, and underscores. An `argument_value` may be an identifier, an integer or real-valued number, a string literal (enclosed in single quotes `'`), or a tuple (a comma-separated list of numbers enclosed in parenthesis, e.g. `(1.0,3.5,4.0)`). Arguments may appear in any order.

3.2 Preparing data for use

The first step in a Surfpack application is usually to prepare a data set for future computation. The user has typically generated a relatively small set of points

(commonly called "training data") using another program and wishes to create a model that can be used to generalize from these data. Surfpack expects data files in a very specific format.

The first line of a data file should specify the number of points in the file. The second line should give the dimensionality of the data set (i.e., the number of predictor or independent variables). The third line tells how many response or dependent variables are included for each point. For these first three lines, Surfpack ignores everything after the first (integer) token. This allows for some annotation of the data, as shown below. The rest of the lines in the data file contain the points themselves, one point per line.

```
100 Crime data for US cities
2 pop. density, persons / sq. mi; median household income in thousands of $
1 crime rate (reported crimes per 10,000 people per yr)
250 58 27
700 35 45
96 41 4
400 78 42
...
```

The simplest way to prepare data for use by Surfpack is to read it in from an existing file. The LoadData command requires two arguments: a name by which the data set can be referenced in future commands and name of the file (full or relative path) from which the data are to be read. Consider the following data stored in a file named simple7.txt.

```
7 made-up points
1 x
2 f1(x) = x^3 - 5x with some added noise , f2(x) = x^2
-3 -22 9
-2 -6 4
-1 4 1
0 0 0
1 -4 1
2 3 4
3 22 9
```

This file can be loaded into Surfpack using the following command:

```
LoadData[name = simple7, file = 'simple7.txt']
```

A common use of empirical models is to predict a response at a set of data points for which the true response is not available. This data set contains seven sample points along the real-number line on the interval $[-3,3]$. After we compute a model to generalize from these seven points, suppose that we want to evaluate our model at 100 evenly spaced points on the same interval. We can prepare this 100-point data set using the CreateAxes and GridPoints commands.

```
CreateAxes[name = test_axis, bounds = '1 | v -3 3 100']
GridPoints[name = test100, axes = test_axis]
```

The CreateAxes command specifies the lower-bound, upper-bound, and number of raster points along one or more dimensions. These values can be read from a file or can be specified as a singly-quoted string literal. See section xxx for more information about the syntax.

The GridPoints command creates a new named data set of evenly spaced points, based on the values in an axes variable. The GridPoints command requires two arguments: the name of the resulting data set, and the name of the axes variable used to compute the data set. An excerpt from the data set created by these commands is shown in figure xx.

```
100
1
0
-3
-2.94
-2.87
...
3
```

Alternatively, an axes variable can be used to create a set of Monte Carlo points. The MonteCarloSample command takes the same arguments as GridPoints, plus an additional 'size' argument that specifies how many points should be generated. Each point is created by randomly selecting a value along each dimension, observing the boundaries given by the axes variable. By default the computer's internal timer is used to seed the random number generator. An optional 'seed' argument ensures the reproducibility of the results.

```
MonteCarloSample[name = random100, axes = test_axis,
                 size = 100, seed = 92613]
```

An excerpt from the data set created by this command is shown in figure xx.

```
100
1
0
1.25
2.47
-3.12
...
0.66
```

3.3 Creating a model from existing data

An set of data samples can be used to create an empirical model, a global function approximation. The CreateSurface command takes at least three arguments: a name for the surface, the data from which the model is to be created,

and the name of a data-fitting algorithm. Additional arguments may specify algorithm-specific parameters. Consider the following command:

```
CreateSurface[name = cubic_poly_1st, data = simple7,  
              type = polynomial, order = 3]
```

Surfpack employs least-squares regression, using up to cubic-polynomial terms, to fit the simple7 data, which must have been created or loaded in a previous commands. Future commands may refer to this model as cubic_poly.

By default, the CreateSurface command uses the first response variable in the supplied defense. The optional 'response_index' argument can be used to specify another response. Indexing of the responses is zero-based. To create a quadratic fit to the second response in the simple7 data set, use the following command.

```
CreateSurface[name = cubic_poly_2nd, data = simple7, type = polynomial,  
              order = 3, response_index = 1]
```

3.4 Evaluating an existing model on a set of data

Suppose we wish to now use our model to predict the response at 100 evenly spaced points on that interval. Assume that these 100 test points have been loaded into Surfpack as a variable named test100. The following command will evaluate our model at these 100 points and append the predictions as a new response variable in the test100 data set.

```
Evaluate[surface = cubic_poly_1st, data = test100]
```

Instead of appending our predictions to an existing data set, Surfpack can create an entirely new data set and store the predictions as the sole response. This is accomplished by adding a name argument to the evaluate command.

```
Evaluate[name = new100, surface = cubic_poly_1st, data = test100]
```

3.5 Analyzing the quality of a model

The quality of an approximation can vary widely, depending on the properties of the function being approximated and on the data samples and algorithm used to create the model. Surfpack provides several metrics to help quantify the quality of an approximation. These are accessible via the Fitness command, which takes at least two parameters: one to specify the surface to be analyzed and another to specify the metric to be used.

```
Fitness[surface = cubic_poly_1st, metric = mean_squared]
```

In this example, the mean-squared-error for the cubic_poly model is printed to the terminal. Since we used a least-squares regression to fit our seven sample data points, our model is not guaranteed to match these seven points exactly (i.e., the model's prediction at those points may not match the true values). The

difference between the true response value (the value given in the data set used to create the model) and the predicted response value at a certain point is often referred to as the residual. The mean-squared-error is the arithmetic mean of all the squared residuals. The mean-squared-error is always non-negative; values close to zero indicate a close fit of the model to the training data.

By default, the Fitness command computes the value of a goodness-of-fit metric using the same data that was provided when the model was created. In some cases, it is desirable to use a different set of data. A common example is to retain a “holdout set” when creating the model, i.e., to use only a subset of the available data to generate the approximation and then use the remaining data to assess the quality of the fit. In this case, an additional argument is supplied to the Fitness command to specify which data should be used to produce the goodness-of-fit metric. See section xxx to learn more about which goodness-of-fit metrics Surfpack supports.

```
LoadData[name = holdout_data, file = 'holdout.txt']
Fitness[surface = cubic_poly_1st, metric = mean_squared, data = holdout_data]
```

Holdout set:

```
5 made-up points
1 x
2 f1(x) = x^3 - 5x with some added noise , f2(x) = x^2
-1.5 4.125 2.25
-0.5 2.375 0.25
0.5 -2.375 0.25
1.5 -4.0 2.25
2.5 3.0 6.25
```

When the data used to compute a goodness-of-fit metric is different than the data used to generate the model, a 'response_index' argument should be used to specify that a response other than the first should be used. Indexing of responses is zero-based.

```
Fitness[surface = cubic_poly_2nd, metric = mean_squared,
data = holdout_data, response_index = 1]
```

3.6 Saving the results of Surfpack computations

The Save command make it possible to store the results of Surfpack computations to files for inspection and future use. The commands requires two arguments: the name of an existing surface or data variable and the name of the file to be written.

```
Save[data = test100, file = 'test100.txt']
Save[surface = cubic_poly_1st, file = 'cubic_poly_1st.txt']
```

The files resulting from Save commands can be read into future Surfpack scripts, using the LoadData and LoadSurface commands.

```

# Load data and surface from previous script
LoadData[name = random100, file = 'random100.txt']
LoadSurface[name = quad_poly_2nd, file = 'quad_poly_2nd.txt']

# Evaluate model on the random data
Evaluate[surface = cubic_poly_2nd, data = random100]

# Save results
Save[data = random100, file = 'random100_test.txt']

```

3.7 Putting it all together

So far in our example, we have created cubic-polynomial fits to two different responses for a small set of data and have computed the mean-squared-error for the models. How would our goodness-of-fit metric be affected if we used a quadratic instead? Figure xx shows the full listing of a Surfpack script that includes all of the commands discussed so far, plus the additional commands needed to answer this question. The output is shown in figure xx.

```

# Read in seven one-dimensional data points with
# two responses per point.
# Data points are on interval [-3,3]
LoadData[name = simple7, file = 'simple7.txt']

# Create a test set with 100 equally-spaced points
# along the interval [-3,3].
CreateAxes[name = test_axis, bounds = '1 | v -3 3 100']
GridPoints[name = test100, axes = test_axis]

# Create a test set with 100 random points from the
# interval [-3,3].
MonteCarloSample[name = random100, axes = test_axis,
                 size = 100, seed = 92613]

# Do third-order regression on each response
CreateSurface[name = cubic_poly_1st, data = simple7,
              type = polynomial, order = 3]
CreateSurface[name = cubic_poly_2nd, data = simple7,
              type = polynomial, order = 3, response_index = 1]

# Evaluate the first model on the equally-spaced
# test data
Evaluate[surface = cubic_poly_1st, data = test100]

```

```

# Calculate the mean-squared-error for the models
# on the data that was used to create the approximations
Fitness[surface = cubic_poly_1st, metric = mean_squared]
Fitness[surface = cubic_poly_2nd, metric = mean_squared]

# Read in data from a hold-out set-- five more points 'true'
# points for each of the two responses
LoadData[name = holdout_data, file = 'holdout.txt']

# Calculate mean-squared_error for the models on holdout set
Fitness[surface = cubic_poly_1st, metric = mean_squared,
data = holdout_data]
Fitness[surface = cubic_poly_2nd, metric = mean_squared,
data = holdout_data, response_index = 1]

## Create a quadratic fit to the two response and compare
## mean-squared-error on the training and holdout data
## with results from the cubic polynomial fits
CreateSurface[name = quad_poly_1st, data = simple7,
type = polynomial, order = 2]
CreateSurface[name = quad_poly_2st, data = simple7,
type = polynomial, order = 2, response_index = 1]
Fitness[surface = quad_poly_1st, metric = mean_squared]
Fitness[surface = quad_poly_1st, metric = mean_squared,
data = holdout_data]
Fitness[surface = quad_poly_2nd, metric = mean_squared]
Fitness[surface = quad_poly_2nd, metric = mean_squared,
data = holdout_data, response_index = 1]

# Save the results for future reference
Save[data = test100, file = 'test100.txt']
Save[surface = cubic_poly_1st, file = 'cubic_poly_1st.txt']

## Save additional results
Save[data = holdout_data, file = 'holdout_test.txt']
Save[data = random100, file = 'random100.txt']
Save[surface = cubic_poly_2nd, file = 'cubic_poly_2nd.txt']
Save[surface = quad_poly_1st, file = 'quad_poly_1st.txt']
Save[surface = quad_poly_2nd, file = 'quad_poly_2nd.txt']

```

4 Examples

4.1 Timing Data

Many scientific computations involve expensive linear algebra operations, such as matrix inversion. Although modern processors can perform billions of operations per second, the computational complexity of popular matrix inversion algorithms means that many interesting problems are simply intractable (i.e., would require several lifetimes worth of supercomputing power). And while the numerical algorithms involved in matrix inversion are well-understood, the complexities of memory hierarchies, scheduling algorithms, and hardware architectures can make it difficult to predict wall-clock performance for various sizes of matrices. Perhaps the best way to evaluate the limits of problem size on a particular computer is through the analysis of empirical data.

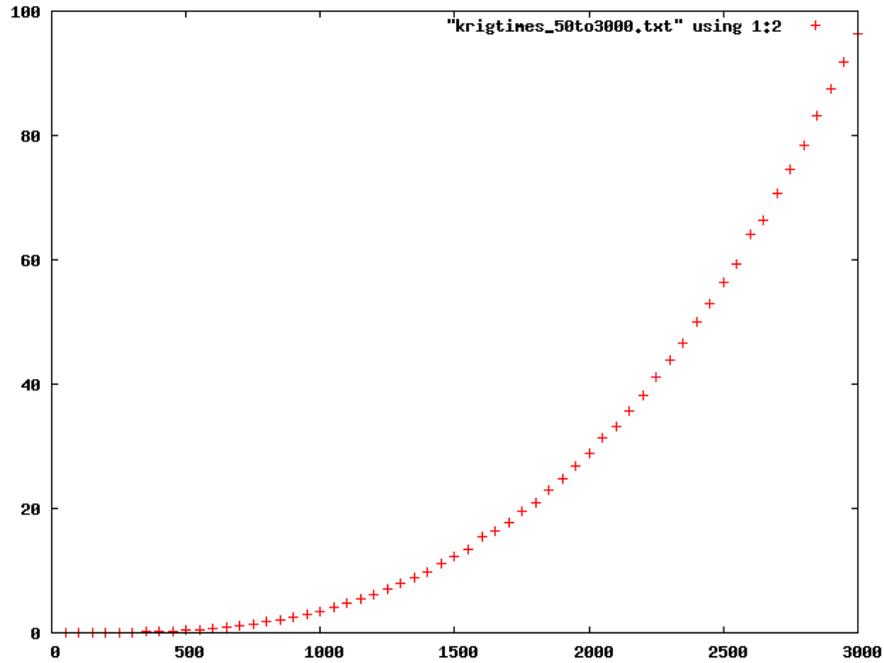
Suppose we want to characterize the speed of inversion for matrices of various sizes on a particular machine. What size of matrices can be handled in one second, one minute, one hour, etc? Figure xxx shows data for such-and-such machine. The Kriging algorithm used in Surfpack—wherein the running time is dominated by a matrix inversion operation—was run using 50–3000 data points, with tests at intervals of 50 points. If the algorithm is run using n points, the inversion of an n -by- n matrix is required. For each matrix size, the median time for five runs of the algorithm is reported. (The experiments were run when the machine was not heavily loaded with other processes, but there is still some variation in running times.)

We will use Surfpack to generate an empirical model from these data, which we can then use to predict running times on problem sizes for which we have not gathered actual data.

Parts of the data file are shown in figure xx.

```
60
1
1
'num_pts''time_in_seconds'
 5.0000000000000000e+01  7.05000013113021851e-04
 1.0000000000000000e+02  3.78199992701411247e-03
 1.5000000000000000e+02  1.05759999714791775e-02
 2.0000000000000000e+02  2.33480000169947743e-02
 2.5000000000000000e+02  4.59529999643564224e-02
... ..
 2.9000000000000000e+03  8.74331180000444874e+01
 2.9500000000000000e+03  9.18161309999413788e+01
 3.0000000000000000e+03  9.63073910000966862e+01
```

This data includes problems using up to 3000 points (which requires the inversion of a 3000 by 3000 matrix). After we create a model to fit these data, we will evaluate the model to predict running times for problems with up to 5000 points.



The first step is to load the data from the file.

```
LoadData[name = timing_data, file = 'krigtines_50to3000.txt']
```

From the plot of the data, we can see that the trend in the data is definitely not linear. We will attempt to fit the data using a quadratic polynomial.

```
CreateSurface[name = poly2, data = timing_data, type = polynomial,
order =2 ]
```

We can use Surfpack to generate the set of test data points and then evaluate the model on those data.

```
CreateAxes[name = test_axes, bounds = '1 | v 50 5000 100']
```

```
GridPoints[name = test_timing_data, axes = test_axes]
```

```
Evaluate[surface = poly2, data = test_timing_data]
```

```
Save[surface = poly2, file = 'quad_poly_timing.txt']
```

```
Save[data = test_timing_data, file = 'test_timing_data.txt']
```

Figure xxx shows a portion of the output file test_timing_data.txt, which lists the predictions of the model for problem sizes of 50 to 5000, at 50 point intervals.

```
100
1
1
          'x0'    'f0 '
5.0000000000000000e+01  4.19372236767611373e+00
```

```

1.0000000000000000e+02  3.32891937004349936e+00
1.5000000000000000e+02  2.54549093625508993e+00
...
4.9000000000000000e+03  2.99187800855931528e+02
4.9500000000000000e+03  3.06216330551186843e+02
5.0000000000000000e+03  3.13326234810286337e+02

```

A plot of the observed data and model predictions is shown in figure xxx.

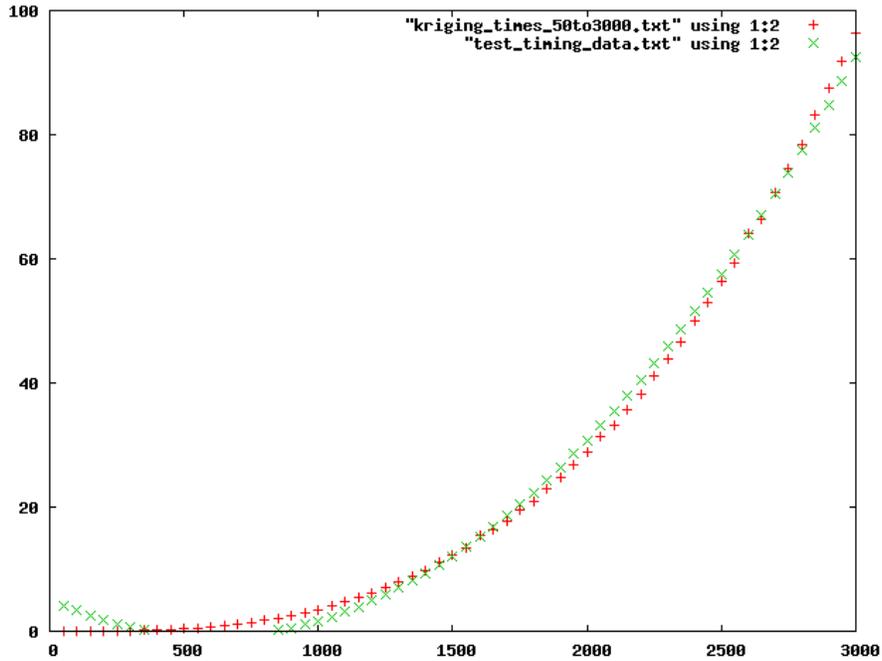


Figure xxx shows an excerpt from quad_poly_snippet.txt, which shows the formula for the quadratic approximation.

```

Polynomial
1 dimensions
2 order
5.139899929152933      +
-0.019737296867978441  x1 +
1.6274912768841024e-05  x1^2
....

```

$$time = \hat{f}(numpts) \approx 1.62x^2 - 0.02x + 5.14$$

The predictions appear to follow the general trend of the data fairly well. The model does curve away from the observed values at the lower-valued data

points, but we are more likely to be concerned about the predictions for larger-sized problems.

We can use Surfpack's Fitness command to quantify the error between the model and the data. We will use the `mean_abs` metric as an example, which computes the absolute value of the difference between each data point used to create the model and the prediction of the model at that point. The reported value is the mean of those residuals.

```
mean_abs for poly2: 1.48252
```

The value of 1.48 means that, on average, the predicted running time differs from the reported running time by 1.48 seconds. The shape of the data and/or knowledge of the underlying matrix inversion algorithm may suggest to us that a cubic polynomial would provide a better fit here.

Two other common goodness-of-fit metrics are PRESS and R-Squared.

```
Fitness[surface = poly2, metric = press]
Fitness[surface = poly2, metric = rsquared]
```

```
press for poly2: 1.87209
rsquared for poly2: 0.996222
```

PRESS gives an average for what the error would be at each data point, if that point were not included in building the model. Values close to zero are more desirable. The R-Squared metric measures the fraction of variance in the model that can be attributed to the variance in the data. Values close to 1 are more desirable.

The plots of the data and/or knowledge of the underlying matrix inversion algorithm may motivate us to try to fit a cubic polynomial to the data.

All of the metrics are worse; this raises some red flags. In particular, it is not possible for the R-Squared value to be lower for a least-squares fit to a cubic polynomial than for a quadratic.

The file `cubicpolytiming.txt` shows the coefficients for the model.

```
Polynomial
1 dimension(s)
3 order
0.10952408396481332      +
-0.00072681435666253696  x1 +
0                        x1^2 +
0                        x1^3
```

$$time = \hat{f}(numpts) \approx -0.0007x + 0.1095$$

The cause of the problem is matrix ill-conditioning. The range of the problem sizes is 50–3000, while the running times range from a fraction of a second up to about 100 seconds. To address this problem, we scale the data so that data fall in the range from 0–1.

```
CreateSurface[name = poly3, data = timing_data, type = polynomial,  
              order = 3, norm_scale = ('num_pts')]
```

```
mean_abs for poly3: 0.10484  
press for poly3: 0.197664  
rsquared for poly3: 0.999956
```

(Include plot)

Now all the metrics are improvements over the quadratic fit, which suggests that the cubic-polynomial more accurately reflects the trends in the data. In the absence of any additional information, we would likely use the cubic-polynomial model to make predictions about running times.

Suppose there were computational resources available to generate a few more data points. Running times for problem sizes of 3050–5000 points are given in the file `test_times.txt`. Now we can evaluate our quadratic and cubic models on these new data and get a better comparison of their predictive capabilities.

```
LoadData[name = timing_data_3050, file = 'kriging_times_3050to5000.txt']  
Evaluate[surface = poly2, data = timing_data_3050]  
Evaluate[surface = poly3, data = timing_data_3050]  
Fitness[surface = poly2, metric = mean_abs, data = timing_data_3050]  
Fitness[surface = poly3, metric = mean_abs, data = timing_data_3050]
```

```
mean_abs for poly2 on timing_data_3050: 48.9  
mean_abs for poly3 on timing_data_3050: 1.03499
```

The predictions for the cubic-polynomial are truly impressive. For a problem size of 5000, the true running time was 437 seconds and the prediction was 439 seconds.

4.2 Topology

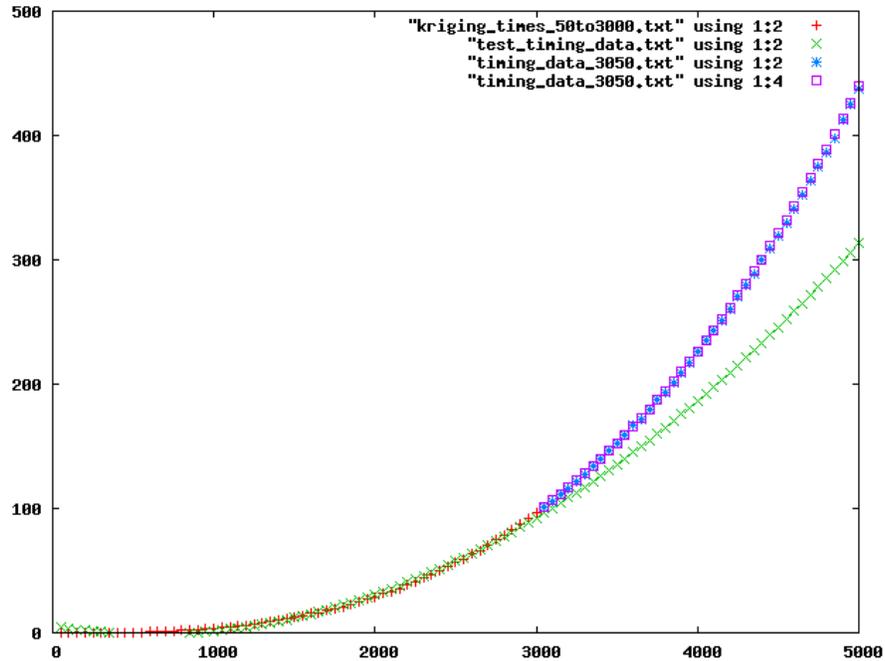
4.3 Rosenbrock

5 Troubleshooting

List of error messages with probable causes and suggestions for resolving them.

5.0.1 Bad surface name in file

When a surface object is read in from a file, the first item listed should be the name of the surface type (Polynomial, Kriging, etc.) Check to make sure that the file being read in is indeed a surface file and that it has a valid type identifier.



5.0.2 Cannot add another response: the number of new response values does not match the size of the physical data set.

This happens when some of the points in a data set have been designated as "excluded." A list of new response values cannot be added in this state, because if the currently excluded points were to be included again, they would not have a needed value for the new response. Future releases will support multi-response data sets in which values for some responses may be missing.

To circumvent this problem, activate all points prior to adding a response, or copy the active points into a new SurfData object, and add the response to the new set.

5.0.3 Cannot add response because there are no data points

The SurfData object to which an attempt is being made to add response data contains no data points. The number of points in the data set should correspond to the number of new response values being added.

5.0.4 Cannot compute euclidean distance. Vectors have different sizes.

When computing the distance between two vectors, v1 and v2, make sure that `v1.size() == v2.size()`.

5.0.5 Cannot create surface with zero dimensionality

A query has been made to `Polynomial::minPointsRequired` in which the dimensionality of the data set has been declared to be zero. All data sets must have dimensionality of at least one.

5.0.6 Cannot set response index on NULL data

A `response_index` argument has been passed to a `Surface` object for which the data set has not yet been specified. First, specify the data set, using either the constructor or the `setData` method. Then call the `config` method with an `response_index Arg` object that specifies which response value will be used to create the surface.

5.0.7 Cannot specify both data and surface.

The `Save` command can be used to write either a data object or a surface object to a file, but not both. Specify one or the other. If both a surface and a data set need to be saved, use two `Save` commands.

5.0.8 Data variable not found in symbol table

The variable name given for the data argument in a `CreateSurface`, `Fitness`, or `Evaluate` command was not found in the symbol table. Make sure that the data object of that name was previously loaded from a file or created using a `GridPoints` or `MonteCarloSample` command. Check for misspellings.

5.0.9 Data unacceptable: there is no data.

An attempt was made to create a `Surface` object without specifying any data. Pass data into the `Surface` object through the constructor or through the `setData` method before invoking `createModel`.

5.0.10 Axes variable not found in symbol table.

The variable name given for the axes argument in a `GridPoints` or `MonteCarloSample` command was not found in the symbol table. Make sure that the axes object of that name was previously loaded from a file or created using a `CreateAxes` command. Check for misspellings.

5.0.11 Dimensionality of data needed to determine number of required samples.

This error occurs when a request is made to know the minimum number of required samples for some surfaces before the dimensionality of the data is determined. In many cases the required number of points is a function of the dimension of the data.

5.0.12 Dimension mismatch: conmin seed and data dimensionality.

By default the correlation parameters for Kriging are computed using doing a maximum likelihood estimation. If a seed for the optimization is specified, it should be a tuple with the same dimensionality as the data set.

5.0.13 Dimension mismatch: correlations and data dimensionality

Kriging expects one correlation value per dimension in the data set.

5.0.14 Dim mismatch in SurfData::setFLabels

The wrong number of labels was given for the data set. These labels are only for the response variables. Use setXLabels to specify tags for the predictor variables.

5.0.15 Dim mismatch in SurfData::setXLabels

The wrong number of labels was given for the data set. These labels are only for the predictor variables. Use setFLabels to specify tags for the response variables.

5.0.16 End of file reached unexpectedly.

When reading data in from a file, there were fewer points than expected in the file or fewer values for a particular point than were expected. Please check the data file.

5.0.17 Error in dgglse

The info flag to the LAPACK routine dgglse returned a non-zero value. See the LAPACK documentation for details. The dgglse routine is used in conjunction with constrained least-squares solves in the PolynomialSurface class.

5.0.18 Cannot add response because physical set size is different than logical set size.

Before adding another response, clear excluded points or create a new data set by using the SurfData::copyActive method. This inconvenience will be resolved in future releases.

5.0.19 Cannot write SurfData object to stream. No active data points.

Clear the excluded data points before writing the data to a file.

5.0.20 Data unacceptable: this surface requires....

The various data-fitting algorithms have their own requirements for how many points are necessary to compute an approximation. Use the `numPointsRequired` method to find how many points are required. Note that this is only the minimum number of points for the algorithm to perform its computations. The number of points needed to get a model that gives useful predictions may be much, much greater. Quantifying these needs is the subject of current research.

5.0.21 Error in SurfData::addPoint. Points in this data set have....

The collection of points in a `SurfData` object must all have the same number of predictor variables. Currently, they must also have the same set of response variables, although this requirement will be relaxed in future releases.

5.0.22 Error in SurfData::sanityCheck....

Surfpack has discovered a mismatch in the dimensionality of at least two data points in a single `SurfData` object. This error can be caused by the modification of individual `SurfPoint` objects (through external handles) after they have been added to a `SurfData` object. While there are legitimate uses of external handles to a `SurfData` object's data points, care must be taken to avoid this kind of inconsistency.

5.0.23 Requested ... max index

A data point was requested from a `SurfData` object, but the index given is equal to or greater than the number of points in the set. Remember that if there are n points, the indices from those points are $0, 1, \dots, n - 1$.

5.0.24 Exception caught and rethrown in SurfPoint::readText

5.0.25 Exception rethrown in SurfPoint::readBinary

An unknown error occurred while reading a file. Check the integrity of your data.

5.0.26 Expected: ... found: ...

The name found in a surface file is inconsistent with the type of `Surface` object that is being create from the file. Check the file contents and object constructor.

5.0.27 Expected 'f' or 'v' on line

The first line of an axes object should be the number of dimensions desired in the resulting data set. Each line after the first should either give a minimum, maximum, and number of raster points for one dimension, or it should give a fixed value for a dimension, which all points in the set will share. Lines with

fixed values should begin with the flag ‘f’; all others should begin with ‘v’ (for ‘variable’).

5.0.28 Index ... specified, but there are zero...

Either a request has been made for a data point in a data set where there are no active points, or a request has been made for a non-existent response variable. Remember that if there are n response variables, they are indexed from 0 to $n - 1$. When requesting data points from a SurfData object, remember that some points may be inactive (excluded), which would reduce the maximum valid index.

5.0.29 In Surface::checkData: No data was passed in

Data for a surface may be specified in the constructor of a Surface object, or using the setData method. If neither of these things occurs before the createModel method is invoked (either directly or indirectly), this error could result.

5.0.30 Integer overflow: number of terms exceeds maximum integer

There are too many terms in the regression model. Use a lower-order polynomial to fit the data or project the data into a lower-dimensional space.

5.0.31 Must know data dimension to use uniform correlation value.

Kriging allows for the specification that the same correlation parameter should be used for each dimension, but the number of dimensions must be known in advance. Specify the data for the KrigingSurface object before invoking this method.

5.0.32 Expected on this line...

The number of predictor and/or response variables on some line in the data file does not match the specified number(s) for the file. Check the format of the file.

5.0.33 No axes argument specified.

A GridPoints or MonteCarloSample command was given, but no axes variable was specified. The axes variable must be created in a previous command. It specifies the (hypercube) boundaries for the data set, and in the case of the GridPoints command, the number of raster points per dimension.

5.0.34 No data argument specified.

A data argument is required for the CreateSurface and Evaluate commands. The data object must have been created (and named) previously in a LoadData, GridPoints, or MonteCarloSample command.

5.0.35 No error metric of that type in this class.

Not all metrics are supported by all methods. Consider using an alternate metric or extending Surfpack to support the desired metric.

5.0.36 No existing surface variable specified.

The Fitness command requires the name of a Surface object that has already been created by a LoadSurface or CreateSurface command.

5.0.37 No filename specified.

All of the Load and Save commands require a valid file name to be given. In all cases, the name of the appropriate argument is 'file'.

5.0.38 No fitness metric specified.

The Fitness command requires the specification of a metric. See section xx for a discussion of supported metrics. See section xx for an explanation of how to extend Surfpack with a new metric. In arguments to the Fitness command, names of metrics should not be quoted.

5.0.39 No name argument specified.

The LoadData, LoadSurface, CreateAxes, CreateSurface, GridPoints, and MonteCarloSample all create new objects that are to be stored in the symbol table for future reference. Each command requires a name argument that gives a designation to the new entity.

5.0.40 No surface or data argument specified.

The Save command expects either a surface argument or a data argument (but not both). Check for misspellings.

5.0.41 No surface type specified.

The CreateSurface command requires a type argument to specify which algorithm should be used to approximate the data: polynomial, kriging, mars, ann, or rbf. See section xx for an explanation of these algorithms.

5.0.42 Not enough data to compute PRESS.

If a Surface object is created using the minimum number of required samples, then the PRESS error metric may not be computed. PRESS creates the n additional models using the same algorithm, but excluding one of the given points each time and then predicting the value of that point after the model has been created. However, if leaving one point out causes the amount of available data to fall below what is required, there is no way to compute the metric.

5.0.43 Out of range in SurfPoint

The i th dimension was requested, but the data has i or fewer dimensions. Remember that if the data has n dimensions, they are indexed from 0 to $n - 1$.

5.0.44 Size of set of excluded points exceeds size of SurfPoint set

Some of the indices passed in to `setExcludedPoints` must either be out of the range of acceptable indices for the data set, or duplications of other excluded points.

5.0.45 Surface variable not found in symbol table

The variable name given for the surface argument in a `Fitness` or `Evaluate` command was not found in the symbol table. Make sure that the surface object of that name was previously loaded from a file or created using a `LoadSurface` or `CreateSurface` command. Check for misspellings.

5.0.46 There are no response values associated with this point

A response value has been requested for a data point for which there are no responses. If the data were read in from a file, check to make sure the contents of the file are as expected.

5.0.47 This Rval class does not have such a value

This error generally means that the type of an argument in a `Surfpack` command was different than what was expected. Common mistakes are using quoted string literals where unquoted identifiers are expected (or vice versa), or specifying a single item when a tuple (a parenthesized list of values) is expected.

5.0.48 Unrecognized filename extension. Use .sd or .txt

5.0.49 Unrecognized filename extension. Use .srf or .txt

`Surfpack` uses file extensions to determine the formatting of information that is read from or written to files. Currently all `Data` and `Surface` files should have a `txt` extension. Future releases will support a binary format for both data and surfaces. The binary formats will be more compact and will provide better I/O performance for large data sets.

Command	Argument	Type	Req.	Meaning
CreateAxes	name	identifier	yes	Variable name by which this definition can be referred to in later commands.
	file	string	yes	File to be read.
CreateSurface	name	identifier	yes	Variable name by which this surface can be referred to in later commands.
	type	identifier	yes	Must be one of ann, mars, kriging, polynomial, or rbf.
	data	identifier	yes	Name of the data set used to build surface.
Evaluate	response_index	integer	no	Specifies which response should be used to create this surface. The default value is 0 (i.e., the first response value listed).
	surface	identifier	yes	The name of the surface to be evaluated. Must have been created or loaded in a previous command.
	data	identifier	yes	Data set that the surface is to be evaluated on. Must have been created or loaded in a previous command. In absence of 'name' argument, results are appended to this data as a new response variable.
Fitness	name	identifier	no	Name of a new data set, to which the results will be appended as the first response variable. It must NOT be the name of a data set that was created or loaded in a previous command.
	surface	identifier	yes	Name of surface to be evaluated.
	metric	identifier	yes	Must be one of press, rsquared, mean_squared, sum_squared, max_squared, mean_scaled, sum_scaled, max_scaled, mean_abs, sum_abs, max_abs, mean_relative, max_relative, root_mean_squared.
	data	identifier	no	The data used to compute the metric. If not specified, the default is to use the same data that were used to build the surface.
	response_index	integer	no	Specifies which response variable in the data set should be used as the "true" value when the fitness of the surface is being computed.
	name	identifier	yes	Variable name by which the resulting data set can be referred to in later commands.
GridPoints	axes	identifier	yes	Axes variable that specifies the boundaries and number of points along each dimension.
	test_function	identifier	no	Name of a test function to add on as a response variable for this data. Must be one of 'rosenbrock', 'rastrigin', 'quasine', 'sphere' or 'sumofall'. Multiple test_function arguments may be included in a single command.
LoadData	name	identifier	yes	Variable name by which this data can be referred to in later commands.
	file	string	yes	Name of file to be read. Must have .txt extension.
LoadSurface	name	identifier	yes	Name by which this surface variable can be referred to in later commands.
	file	string	yes	Name of file to be read. Must have .txt extension.
MonteCarloSample	name	identifier	yes	Name by which the resulting data set can be referred to in later commands.
	axes	identifier	yes	Axes variable that specifies the boundaries along each dimension. Must have been created in a previous command.
	test_function	identifier	no	Same as for the GridPoints command.
	size	integer	no	Number of random samples. Default is 100.
Save	surface/data	identifier	yes	Name of data or surface to be saved. Must have been previously defined.
	file	string	yes	Name of file to be written to. Must have .txt extension.

Table 1: Surfpack Command Summary

Argument	For	Type	Req.	Default	Meaning
order	polynomial	integer	no	2	Maximum order of polynomial regression.
norm_bound	ann	real	no	0.8	?
svdfactor	ann	real	no	0.9	Range 0-1.
fraction_withheld	ann	real	no	0.0	Fraction of data set used for validation.
conmin_seed	kriging	tuple	no	(1.0, ..., 1.0)	Initial values of correlation parameters passed to conmin optimizer. The tuple (vector) must have the same dimensionality as the data set.
theta_vars	kriging	tuple	no	conmin	Exact correlation parameters to be used. If theta_vars are specified, conmin is bypassed altogether. The tuple (vector) must have the same dimensionality as the data set.
max_bases	mars	integer	no	25	Maximum number of basis functions.
max_interactions	mars	integer	no	2	Maximum number of variable interactions that will be considered.
interpolation	mars	string	no	'cubic'	Must be either 'linear' or 'cubic'. Determines what kind of splines are used after surface is built.
radius	rbf	real	no	0.1	Radius of the basis functions.

Table 2: Surface Method Argument Summary