# Surfpack User's Manual
# Version 1.1

Keith R. Dalbey[1], Anthony A. Giunta[1], Mark D. Richards[2], Eric C. Cyr[1]
Laura P. Swiler[1], Shane L. Brown[1], Michael S. Eldred[1], Brian M. Adams[1]

December 27, 2017

[1]Sandia National Laboratories*
P.O. Box 5800, Mail Stop 1318
Albuquerque, NM 87185-1318 USA
Email: surfpack-developers@development.sandia.gov
Web: http://dakota.sandia.gov/packages/surfpack

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

*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 meta-models. 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, Moving Least Squares, 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, AIX, Windows (Cygwin), and Mac OS X. Native Visual C++ compilation is not currently supported.

## 2.1 Obtaining Surfpack

Surfpack source packages are distributed from `http://dakota.sandia.gov/packages/surfpack`. It is also included in all Dakota distributions and can be obtained via anonymous Subversion checkout from `https://software.sandia.gov/svn/public/surfpack`. Binary packages are no longer supported.

## 2.2 Requirements

- **C++**. A majority of the source code for Surfpack is written in C++. Surfpack makes frequent use of the standard C and C++ libraries, including the Standard Template Library. The Kriging algorithm calls some FORTRAN code (pivoted Cholesky and some optimizers such as CONMIN).
- **Fortran 77**. The MARS algorithm is implemented in FORTRAN.
- **BLAS**. Many of the data-fitting algorithms rely on the Basic Linear Algebra Subroutines to perform rudimentary linear algebra operations.
- **LAPACK**. Surfpack makes use of the following LAPACK [1] driver routines: dpotrf, dpocon, dpotrs, dlange, dgetrf, dgetri, dgels, dgglse. 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 dependencies is included in the Surfpack distribution. However, BLAS and LAPACK routines are usually highly optimized for each platform; performance is best when pre-compiled, native versions of these functions are available.

## 2.3 Options

- **Boost serialization**. As of Version 1.1, Surfpack's model save/load facility requires the Boost serialization binary component.
- **CPP Unit**. A suite of unit tests is available on platforms where CPP Unit has been built.
- **lex and yacc**. Surfpack may be used as a library or as a stand-alone program. The stand-alone executable requires lex and yacc (a lexical analyzer generator and parser generator, respectively), which are normally distributed with Unix-like operating systems. They are also freely available on the Internet.

## 2.4 Standard build

To compile Surfpack from source, extract the source, configure with CMake, and compile with your native make system, for example:

```
tar xzf surfpack-1.1.tar.gz
cd surfpack-1.1
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=/apps/surfpack -DCMAKE_BUILD_TYPE=Release ..
make -j 4
[make install]
```

## 2.5 Testing the Installation

*Installation testing is not currently supported.* Surfpack is distributed with its test suite. The unit testing suite utilizes the CPP Unit libraries (`http://www.sourceforge.net/projects/cppunit`), which must be present 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.

# 3 Getting Started

This chapter outlines the basic commands in the Surfpack API. Normally, a script file is passed to `surfpack` as a command-line argument; the commands in the file are executed sequentially. If no command-line arguments are given, Surfpack reads a list of commands from standard input.

General conventions are presented in Section 3.1. The various Surfpack commands are discussed in Sections 3.2–3.6 and are illustrated using the sample script `examples/GettingStarted/getting_started.spk`.

## 3.1 Conventions

Surfpack commands consist of a capitalized command name followed by a comma-delimited list of arguments in square brackets `[ ]`. Lines beginning with '#' are interpreted as comments and are ignored. White-space is ignored inside commands. 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 are delimited by `/*` and `*/`.

The Surfpack interpreter internally maintains lists of three types of variables: `axes`, `data`, and `surfaces`. Axes variables are created using the `CreateAxes` command. Data variables may be created by the `Load`, or `CreateSample` commands. Surfaces variables are created with the `Load` or `CreateSurface` commands. Any command that creates an `axes`, `data`, or `surface` variable must have a `name` argument, so that the variable can be used 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`). Figure 1 shows examples of commands that create and/or use the different types of variables.

```
# Define boundaries for a future data set.  Min/max values are separated by
# white space; values for different dimensions are delimited by '|':
# Then use the axes variable to create a new data set
CreateAxes[name = boundaries_2d, bounds = '-2 2 | -2 2']
CreateSample[name = rosenbrock_2d, axes = boundaries_2d, grid_points = (11,11),
  labels = (x0,x1), test_functions = (rosenbrock)]

# Load a data file from disk.  This data will be used later in
# an evaluate command.
Load[name = test_rosen, file = 'rosenbrock_2d.spd', n_predictors = 2,
         n_responses = 1]

# Use the data set to create a new surface. The 'name' argument gives a name
# for the newly create surface object.  The 'data' argument refers to a data
# object previously created using Load or CreateSample.  Then use the newly
# created surface object to predict the responses for an existing data set
CreateSurface[name = krig_rosen, data = rosenbrock_2d, type = kriging]
Evaluate[surface = krig_rosen, data = test_rosen, label = 'krig_est']
```

Figure 1: Examples showing the creation and usage of axes, data, and surface variables. The file, `conventions.spk`, is located in the `examples/GettingStarted` subdirectory.

### 3.1.1 Command Arguments

Each argument has the 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 comma-separated list of values enclosed in parentheses, *e.g.*, `(1.0,3.5,4.0)`. Arguments may appear in any order.

## 3.2 Loading Data From a File

The first step in a Surfpack application is usually to prepare a data set for future computation. Typically, data have already been collected and are stored in a text file. Surfpack requires data files to conform to a specific format. The data points in the file should be listed one point per line. Each point should consist of one or more predictor variables followed by zero or more response variables. All data points in a single file must have the same number of variables; the variables for each point must appear in the same order. An optional header line, beginning with a '%', lists the labels for each of the variables. Figure 2 shows an example.

```
%Yr     World   Africa  Asia    Europe  Lt. Am. No.Am.  Oceania
1960    3021475 277398  1701336 604401  218300  204152  15888
1965    3334874 313744  1899424 634026  250452  219570  17657
1970    3692492 357283  2143118 655855  284856  231937  19443
1975    4068109 408160  2397512 675542  321906  243425  21564
1980    4434682 469618  2632335 692431  361401  256068  22828
1985    4830979 541814  2887552 706009  401469  269456  24678
1990    5263593 622443  3167807 721582  441525  283549  26687
1995    5674380 707462  3430052 727405  481099  299438  28924
2000    6070581 795671  3679737 727986  520229  315915  31043
2005    6453628 887964  3917508 724722  558281  332156  32998
```

Figure 2: A data file containing ten data points with one predictor variable and seven response variables. The file is `examples/GettingStarted/world_pop.spd`.

Data files can be read into Surfpack using the `Load` command, which expects four arguments: a `name` identifier for the data set, a string argument `file` specifying the full or relative path of the data file, and integers `n_predictors` and `n_responses` which indicate the number of predictor and response variables, respectively, in the data set. Surfpack expects data files to have a `.spd` extension. The data set shown in Figure 2 can be loaded into Surfpack using the following command:

```
Load[name = world_pop, file = 'world_pop.spd', n_predictors= 1, n_responses = 7]
```

## 3.3 Creating a model from existing data

The `CreateSurface` command creates a global approximation to a function using a sample of known points. `CreateSurface` takes at least three arguments: a `name` for the surface, the `data` from which the model is to be created, and the `type` of data-fitting algorithm. Additional arguments may specify algorithm-specific parameters and/or data scaling options. Consider the following command:

```
CreateSurface[name = world_poly, data = world_pop, response = World,
  type = polynomial, order = 2, log_scale = (World),  norm_scale = (Yr)]
```

The `polynomial` value for the `type` argument tells Surfpack to use linear regression to fit the `world_pop` data, which must have been created or loaded in a previous command. The `order = 2` argument specifies that up to quadratic terms may be used in the regression model. The `name` argument specifies that future commands may refer to this model as `world_poly`. The `response` parameter indicates which of the response variables in the `world_poly` data set should be used to create the global approximation. The `log_scale` and `norm_scale` arguments take parenthesized lists of variables that are to be scaled before the global approximation is created. When a variable is scaled using `norm_scale`, all of the values for that variable in the given data set are mapped to the interval $[0, 1]$.

### 3.4 Evaluating an existing model on a set of data

The `world_pop` data set contains world population data from the years 1960–2005. Suppose we wish to create a model to predict the size of the population at five year intervals up to 2050. If a file with these query points already exists, it may be read in using the `Load` command as described above. Alternatively, the data set may be created on the fly using Surfpack's `CreateAxes` and `CreateSample` commands. The `CreateAxes` command defines minimum/maximum pairs for a list of variables on a Cartesian coordinate system. These range pairs serve as the boundaries inside which future data sets may be created. In this example, there is only one variable (time) and the range of values that we are interested in is $[2010, 2050]$. `CreateAxes` takes two arguments: normally a `name` identifier for the resulting `axes` variable that is created, and a string `bounds` that defines the boundaries for the data set. For multidimensional data sets, the min/max pairs for each dimension should be delimited by '|'. Since the population data set has only one predictor variable (year), the appropriate `CreateAxes` command is

```
CreateAxes[name = test_years, bounds = ’2010 2050’]
```

With appropriate boundaries for our data set defined in an `axes` variable, we can use the `CreateSample` command to generate a set of query points. `CreateSample` expects at least three arguments. The `name` and `axes` identifiers give a designation and a reference to an existing `axes` variable, respectively, for the new data set. For a random sampling of Monte Carlo points (*i.e.*, a data set where each variable for each point receives a random value drawn uniformly from the boundaries defined by the `axes` variable), the `size` argument specifies the number of points in the data set. Alternatively, to generate regularly spaced points on a grid, the `grid_points` argument is used. The value for the `grid_points` argument is a list of integers which specifies the number of grid points along each dimension. The optional `labels` argument specifies a list of identifiers that are to be used as the headings for the variables in the new data set. To generate query points at five-year intervals, we can use

```
CreateSample[name = test_years, axes = test_ax, grid_points = (9),
  labels = (Yr)]
```

Now we can use Surfpack's `Evaluate` command to make the predictions. `Evaluate` takes two required parameters: a `surface` argument indicates which existing surface is to be evaluated; a `data` argument gives the set of data that are to be evaluated. Surfpack appends a new response variable to the data set. An optional `label` argument gives a name for the new response:

```
Evaluate[surface = world_poly, data = test_years, label = WorldEst]
```

### 3.5 Quantifying Model Fitness

The quality of an approximation depends on the properties of the function being approximated and on the data samples and algorithm used to create the model. Surfpack provides several metrics for quantifying how well the model fits the data used to create it and for predicting how well the model might generalize to unseen data. These are accessible via the `Fitness` command, which takes at least two arguments: the `surface` to be analyzed and the quality-of-fit `metric` to be used:

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

In this example, the mean squared error (MSE) for the `world_poly` model is printed to the terminal. Since we used a least-squares regression to fit the sample data points, our model is not guaranteed to match these ten points exactly (*i.e.*, the model's prediction at those points may not match the response values of the training data). The difference between the model's prediction and the true response value is the *residual.* The MSE is the arithmetic mean of all the squared residuals. MSE values near zero indicate a close fit of the model to the training data.

## 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: an existing surface or data variable and the name of the file to be written:

```
Save[data = test_years, file = ’pop_est.spd’]
Save[surface = world_poly, file = ’world_poly.sps’]
```

Filename extensions should be .spd for data files and .sps for surface files. The files resulting from Save commands can be read into future Surfpack scripts using the Load command.

## 3.7 Putting it all together

The full listing for the world population example is shown in Figure 3.

```
Load[name = world_pop, file = ’world_pop.spd’, n_predictors= 1, n_responses = 7]

CreateAxes[name = years_ax, bounds = ’2010 2050’]
CreateSample[name = test_years, axes = years_ax, grid_points = (9),
  labels = (Yr)]

CreateSurface[name = world_poly, data = world_pop, response = World,
  type = polynomial, order = 2, log_scale = (World),  norm_scale = (Yr)]
Evaluate[surface = world_poly, data = test_years, label = WorldEst]
Fitness[surface = world_poly, data = world_pop, metric = mean_squared]
Fitness[surface = world_poly, data = world_pop, metric = root_mean_squared]
Fitness[surface = world_poly, data = world_pop, metric = press]
Fitness[surface = world_poly, data = world_pop, metric = rsquared]
Fitness[surface = world_poly, data = world_pop, metric = max_abs]
Fitness[surface = world_poly, data = world_pop, metric = mean_abs]
Fitness[surface = world_poly, data = world_pop, metric = mean_scaled]
Fitness[surface = world_poly, data = world_pop, metric = max_scaled]
Save[data = test_years, file = ’pop_est.spd’]
Save[surface = world_poly, file = ’world_poly.sps’]
```

Figure 3: Full listing of the world population example presented in throughout this chapter.

# 4 Surface Fitting Algorithms

## 4.1 Linear, Quadratic, and Cubic Polynomial Models

Linear, quadratic, and cubic polynomial models are available in Surfpack. The form of the linear polynomial model is

$$\hat{f}(\mathbf{x}) = c_0 + \sum_{i=1}^{n} c_i x_i \tag{1}$$

the form of the quadratic polynomial model is:

$$\hat{f}(\mathbf{x}) = c_0 + \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n} \sum_{j \geq i}^{n} c_{ij} x_i x_j \tag{2}$$

and the form of the cubic polynomial model is:

$$\hat{f}(\mathbf{x}) = c_0 + \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{n} \sum_{j \geq i}^{n} c_{ij} x_i x_j + \sum_{i=1}^{n} \sum_{j \geq i}^{n} \sum_{k \geq j}^{n} c_{ijk} x_i x_j x_k \tag{3}$$

In all of the polynomial models, $\hat{f}(\mathbf{x})$ is the response of the polynomial model; the $x_i, x_j, x_k$ terms are the components of the $n$-dimensional design parameter values; the $c_0$ , $c_i$ , $c_{ij}$ , $c_{ijk}$ terms are the polynomial coefficients, and $n$ is the number of design parameters. The number of coefficients, $n_c$, depends on the order of the polynomial model and the number of design parameters. For the linear polynomial, $n_c = n + 1$; for the quadratic polynomial, $n_c = (n+1)(n+2)/2$; and for the cubic polynomial, $n_c = (n^3 + 6n^2 + 11n + 6)/6$. There must be at least $n_c$ data samples in order to form a fully determined linear system and solve for the polynomial coefficients. Surfpack employs a standard least-squares approach using subroutines from the LAPACK software library to solve the linear system for the unknown coefficients.

The utility of the polynomial models stems from two sources: (1) over a small region of the parameter space, a low-order polynomial model is often an accurate approximation to the true data trends, and (2) the least-squares procedure provides a surface fit that smooths out noise in the data. However, a polynomial surface fit may not be the best choice for modeling data trends over the entire parameter space, unless it is known a priori that the true data trends are close to linear, quadratic, or cubic.

The polynomial regression model can take a single parameter: an integer `order` that specifies the maximum degree of the polynomial approximation. Permissible values are 1, 2, or 3, for linear, quadratic, or cubic polynomials respectively. The default is a quadratic fit.

## 4.2 Kriging Interpolation

The Kriging model in Surfpack version 1.0 was that of Giunta and Watson [2], with reference to [3] and [4]. However, the Kriging model has been re-implemented with numerous additional options for Surfpack 1.1.

### 4.2.1 Notational Conventions

The notational conventions used in this section on Kriging are that vectors are indicated by single underlines, matrices are indicated by double underlines, capital letters are associated with data used to build the model, and lower case letters are used for arbitrary points (e.g. where the emulator is to be evaluated). The standard convention is that each row of a sample design matrix contains a point.

### 4.2.2 Introduction

The set of interpolation techniques known as Kriging, also referred to as Gaussian processes, were originally developed in the geostatistics and spatial statistics communities to produce maps of underground geologic deposits based on a data from widely and irregularly spaced boreholes [5]. Building a Kriging model typically requires

1. Choice of a trend function;
2. Choice of a correlation function; and
3. Estimation of correlation parameters from data.

The Surfpack 1.1 Kriging model can use both function value and gradient information (when available) to construct the emulator. The default is to construct a Kriging model without using derivative information, but gradient-enhanced Kriging (GEK) can be selected with the `derivative_order` keyword followed by the value “1” to indicate first order derivative (gradient) information.

GEK is available when Surfpack is called by Sandia's DAKOTA software package for optimization and uncertainty quantification, and via direct C++ interfacing to the Kriging model class (as it supports reading and writing derivative data through its supporting nfm::SurfData class). However, since the current Surfpack data `.spd` file format does not support derivative data, GEK is not available in stand-alone Surfpack executables.

A Kriging emulator, $\hat{f}\left(\underline{x}\right)$, consists of a trend function (frequently a least squares fit to the data, $\underline{g}\left(\underline{x}\right)^T \underline{\beta}$) plus a Gaussian process error model, $\epsilon\left(\underline{x}\right)$, that is used to correct the trend function.

$$\hat{f}\left(\underline{x}\right) = \underline{g}\left(\underline{x}\right)^T \underline{\beta} + \epsilon\left(\underline{x}\right)$$

This represents an estimated distribution for the unknown true surface, $f\left(\underline{x}\right)$. The error model, $\epsilon\left(\underline{x}\right)$, makes an adjustment to the trend function so that the emulator will interpolate, and have zero uncertainty at, the data points it was built from. The covariance between the error at two arbitrary points, $\underline{x}$ and $\underline{x'}$, is modeled as

$$\operatorname{Cov}\left(y\left(\underline{x}\right), y\left(\underline{x'}\right)\right) = \operatorname{Cov}\left(\epsilon\left(\underline{x}\right), \epsilon\left(\underline{x'}\right)\right) = \sigma^2 \; r\left(\underline{x}, \underline{x'}\right).$$

Here $\sigma^2$ is known as the unadjusted variance and $r\left(\underline{x}, \underline{x'}\right)$ is a correlation function. Measurement error can be modeled explicitly by modifying this to

$$\operatorname{Cov}\left(\epsilon\left(\underline{x}\right), \epsilon\left(\underline{x'}\right)\right) = \sigma^2 \; r\left(\underline{x}, \underline{x'}\right) + \Delta^2 \delta\left(\underline{x} - \underline{x'}\right)$$

where

$$\delta\left(\underline{x} - \underline{x'}\right) = \begin{cases} 1 & \text{if } \underline{x} - \underline{x'} = \underline{0} \\ 0 & \text{otherwise} \end{cases}$$

and $\Delta^2$ is the variance of the measurement error. In this work, the term "nugget" refers to the ratio $\eta = \frac{\Delta^2}{\sigma^2}$.

### 4.2.3 Trend Function

By convention, the terms simple Kriging, ordinary Kriging, and universal Kriging are used to indicate the three most common choices for the trend function. In simple Kriging, the trend is treated as a known constant, usually zero, $g\left(\underline{x}\right)\beta \equiv 0$. Universal Kriging [6] uses a general polynomial trend model $\underline{g}\left(\underline{x}\right)^T \underline{\beta}$ whose coefficients are determined by least squares regression. Ordinary Kriging is essentially universal Kriging with a trend order of zero, i.e. the trend function is treated as an unknown constant, so $g\left(\underline{x}\right) = 1$ and $\underline{\beta}$ is estimated from data. Let $N_\beta$ denote the number of basis functions in $\underline{g}\left(\underline{x}\right)$ and therefore number of elements in the vector $\underline{\beta}$.

Universal Kriging is implemented in Surfpack and the default trend function is a main effects (no interaction terms) quadratic polynomial. If the `order` keyword is used to specify a different polynomial order (any nonnegative integer is valid input for `order`) then a full polynomial with be used unless the `reduced_polynomial` keyword is also used to indicate a main effects polynomial. If the build data is insufficient to construct a polynomial of the requested order, the order will be automatically reduced.

For a finite number of sample points, $N$, there will be uncertainty about the most appropriate value of the vector, $\underline{\beta}$, which can therefore be described as having a distribution of possible values. If one assumes zero prior knowledge about this distribution, which is referred to as the "vague prior" assumption, then the maximum likelihood value of $\underline{\beta}$ can be computed via least squares generalized by the inverse of the error model's correlation matrix, $\underline{\underline{R}}$

$$\underline{\hat{\beta}} = \left(\underline{\underline{G}} \; \underline{\underline{R}}^{-1} \underline{\underline{G}}^T\right)^{-1} \left(\underline{\underline{G}} \; \underline{\underline{R}}^{-1} \underline{Y}\right).$$

Here $\underline{\underline{G}}$ is a $N_\beta$ by $N$ matrix that contains the evaluation of the least squares basis functions at all points in the sample design matrix, $\underline{\underline{X}}$, such that $G_{l,i} = g_l\left(\underline{X_i}\right)$. If $\underline{\underline{G}} \; \underline{\underline{R}}^{-1} \underline{\underline{G}}^T$ is singular or if the number of basis functions in the requested trend is greater than half of the number of equations in $\underline{\underline{R}}$ then a pivoted Cholesky factorization of $\underline{\underline{G}}\;\underline{\underline{R}}^{-1}\underline{\underline{G}}^T$ will be perform to select the retainable subset of basis functions with the maximum one norm condition number. This protects against a singular $\underline{\underline{G}}\;\underline{\underline{R}}^{-1}\underline{\underline{G}}^T$ matrix, but if too high a trend order is specified it will "steal" the most useful information for the Gaussian Process error model and result in a Kriging model with relatively poor prediction quality.

### 4.2.4 Correlation Functions and Lengths

The real, symmetric, positive-definite correlation matrix, $\underline{\underline{R}}$, of the error model contains evaluations of the correlation function, $r$, at all pairwise combination of points (rows) in the sample design, $\underline{\underline{X}}$.

$$R_{i,j} = R_{j,i} = r\left(\underline{X_i}, \underline{X_j}\right) = r\left(\underline{X_j}, \underline{X_i}\right)$$

There are many options for $r$, among them are the following families of correlation functions:

- **Powered-Exponential**

  $$r\left(\underline{X_i}, \underline{X_j}\right) = \exp\left(-\sum_{k=1}^{M} \theta_k \left|X_{i,k} - X_{j,k}\right|^{\gamma}\right) \tag{4}$$

  where $1 < \gamma \leq 2$ and $0 < \theta_k$. The squared-exponential correlation function is also known as the "Gaussian" correlation function.

- **Matern**

  $$r\left(\underline{X_i}, \underline{X_j}\right) = \prod_{k=1}^{M} \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\theta_k \left|X_{i,k} - X_{j,k}\right|\right)^{\nu} \mathcal{K}_{\nu}\left(\theta_k \left|X_{i,k} - X_{j,k}\right|\right)$$

  where $0 < \nu$, $0 < \theta_k$, and $\mathcal{K}_{\nu}(\cdot)$ is the modified Bessel function of order $\nu$. The Matern function is commonly implemented with $\nu$ restricted to $\nu = i + \frac{1}{2}$, where $i$ is a non negative integer because these values of $\nu$ result in greatly simplified formulas. For example, Matern with $\nu = \frac{1}{2}$ is the exponential correlation function. Matern with $\nu = \infty$ is the squared-exponential or "Gaussian" correlation function. Discontinuities in the Matern correlation function's derivatives can only occur at the coordinates of build points, and there the Matern correlation function is $\mathrm{ceil}\,(\nu)$ times differentiable. However note that although the second derivative of the 1D Matern function with $\nu = 3/2$ is not defined at build points, its limits from both sides are defined and equal to each other. This allows the Matern 3/2 correlation function to be used for Gradient Enhanced Kriging.

- **Cauchy**

  $$r\left(\underline{X_i}, \underline{X_j}\right) = \prod_{k=1}^{M} \left(1 + \theta_k \left|X_{i,k} - X_{j,k}\right|^{\gamma}\right)^{-\nu}$$

  where $0 < \gamma \leq 2$, $0 < \nu$, and $0 < \theta_k$.

Gneiting, et al., [7] provide a more thorough discussion of the properties of and relationships between these three families. Some additional correlation functions include the Dagum family [8] and cubic splines.

Surfpack supports the powered-exponential correlation function with $1 \leq \gamma \leq 2$ and Matern correlation function with $\nu \in \{1/2, 3/2, 5/2, \infty\}$. The powered-exponential correlation function can be specified with `powered_exponential` keyword followed a real number between 1 and 2 (inclusive). The Matern correlation function can be specified with the `matern` keyword followed by 0.5 or 1.5 or 2.5 or `infinity`. For gradient-enhanced Kriging, only the Gaussian, Matern 3/2, and Matern 5/2 correlation functions may be used. In empirical studies, the Gaussian correlation function was often the most accurate (for both Kriging and gradient-enhanced Kriging) and as such it is Surfpack's default. Its infinite smoothness or differentiability is beneficial for leveraging sparse data (which is the most typical case).

For the Gaussian correlation function, the correlation parameters, $\underline{\theta}$, are related to the correlation lengths, $\underline{L}$, by

$$\theta_k = \frac{1}{2\ L_k^2}. \tag{5}$$

Here, the correlation lengths, $\underline{L}$, are analogous to standard deviations in the Gaussian or normal distribution and often have physical meaning. Similarly, for the powered-exponential correlation function

$$\theta_k = \frac{1}{\gamma\ L_k^\gamma}, \tag{6}$$

and for the Matern function

$$\theta_k = \frac{\sqrt{2\ \nu}}{L_k}. \tag{7}$$

When the user has knowledge or intuition of reasonable correlation lengths to build the Kriging model, they can be directly specified by setting `optimization_method` = `none` and the keyword `correlation_lengths` followed a list of $M$ correlation lengths, where $M$ is the number of dimensions.

#### 4.2.5 Methods of Handling Ill-Conditioned $\underline{\underline{R}}$ Matrices

Ill-conditioning of $\underline{\underline{R}}$ and other matrices is widely recognized as a significant challenge for Kriging. Davis and Morris [9] gave a thorough review of six factors affecting the condition number of matrices associated with Kriging (from the perspective of semivariograms rather than correlation functions). They concluded that "Perhaps the best advice we can give is to be mindful of the condition number when building and solving Kriging systems." In the context of estimating the optimal $\underline{\theta}$, Martin [10] stated that Kriging's "three most prevalent issues are (1) ill-conditioned correlation matrices, (2) multiple local optima, and (3) long ridges of near-optimal values." Martin used constrained optimization to address ill-conditioning of $\underline{\underline{R}}$. Rennen [11] advocated that ill-conditioning be handled by building Kriging models from a uniform subset of available sample points. This last mitigation approach has been available in DAKOTA's "Gaussian process" model since version 4.1 [12].

Adding a nugget, $\eta$, to the diagonal entries of $\underline{\underline{R}}$ (the generalization to gradient-enhanced Kriging used in Surfpack, which does not have all ones on its diagonal, is to multiply diagonal elements by $1+\eta$) is a popular approach for both accounting for measurement error in the data and alleviating ill-conditioning. However, doing so will cause the Kriging model to smooth or approximate rather than interpolate the data. Methods for choosing a nugget include:

- Choosing a nugget based on the variance of measurement error (if any); this will be an iterative process if $\sigma^2$ is not known in advance. The keyword `nugget` followed by a non negative real number can be used to directly specify $\eta$ for the Surfpack Kriging model.

- Iteratively adding a successively larger nugget until $\underline{\underline{R}} + \eta\underline{\underline{I}}$ is no longer ill-conditioned. This approach is not supported in Surfpack.

- Exactly calculating the minimum nugget needed for a target condition number from $\underline{\underline{R}}$'s maximum $\lambda_{max}$ and minimum $\lambda_{min}$ eigenvalues. Note, however, that calculating eigenvalues is expensive and can be used to obtain a desired 2-norm condition number. For linear algebraic operations, the 1 norm condition number is generally more appropriate and LAPACK can produce inexpensive estimates of "rcond" or the reciprocal of the 1-norm condition number The estimated rcond of $\underline{\underline{R}}$ can be used to calculate a still quite small upper bound on the nugget which might be needed to alleviate ill conditioning. This is the what is added, as needed, when the `find_nugget` option of the Surfpack Kriging model is added.

- Treating $\eta$ as an independent parameter to be selected via the same process used to choose $\underline{\theta}$, has elsewhere been used to handle ill-conditioning but this option is not supported in Surfpack. Two approaches for determining $\underline{\theta}$ are discussed below.

### 4.2.6 The Adjusted Mean and Variance

The adjusted (by data) mean of the emulator is a best linear unbiased estimator of the unknown true function,

$$\hat{y} = \mathrm{E}\left(\hat{f}\left(\underline{x}\right)|\underline{f}\left(\underline{\underline{X}}\right)\right) = \underline{g}\left(\underline{x}\right)^T \underline{\hat{\beta}} + \underline{r}\left(\underline{x}\right)^T \ \underline{\underline{R}}^{-1}\underline{\epsilon}. \tag{8}$$

Here, $\underline{\epsilon} = \left(\underline{Y} - \underline{\underline{G}}^T \underline{\hat{\beta}}\right)$ is the known vector of differences between the true outputs and trend function at all points in $\underline{\underline{X}}$ and the vector $\underline{r}\left(\underline{x}\right)$ is defined such that $r_i\left(\underline{x}\right) = r\left(\underline{x}, \underline{X_i}\right)$. This adjustment can be interpreted as the projection of prior belief (the least squares fit) into the span of the data. The adjusted mean of the emulator will interpolate the data that the Kriging model was built from as long as its correlation matrix, $\underline{\underline{R}}$, is numerically non-singular.

The Kriging model's adjusted variance is commonly used as a spatially varying measure of uncertainty. Knowing where, and by how much, the model "doubts" its own predictions helps build user confidence in the predictions and can be utilized to guide the selection of new sample points during optimization or to otherwise improve the surrogate. The adjusted variance is

$$\begin{aligned}
\mathrm{Var}\left(\hat{y}\right) &= \mathrm{Var}\left(\hat{f}\left(\underline{x}\right)|\underline{f}\left(\underline{\underline{X}}\right)\right) \\
&= \hat{\sigma}^2 \left(1 - \underline{r}\left(\underline{x}\right)^T \ \underline{\underline{R}}^{-1}\underline{r}\left(\underline{x}\right) + ... \right. \\
&\quad \left. \left(\underline{g}\left(\underline{x}\right)^T - \underline{r}\left(\underline{x}\right)^T \ \underline{\underline{R}}^{-1}\underline{\underline{G}}^T\right) \left(\underline{\underline{G}}\ \underline{\underline{R}}^{-1}\underline{\underline{G}}^T\right)^{-1} \left(\underline{g}\left(\underline{x}\right)^T - \underline{r}\left(\underline{x}\right)^T \ \underline{\underline{R}}^{-1}\underline{\underline{G}}^T\right)^T\right)
\end{aligned}$$

where the maximum likelihood estimate of the unadjusted variance is

$$\hat{\sigma}^2 = \frac{\underline{\epsilon}^T \underline{\underline{R}}^{-1}\underline{\epsilon}}{N - N_\beta}.$$

### 4.2.7 Methods of Choosing $\underline{\theta}$

There are at least two types of numerical approaches for choosing $\underline{\theta}$. One of these is to use Bayesian techniques such as Markov chain Monte Carlo (MCMC) to obtain a distribution represented by an ensemble of vectors $\underline{\theta}$. In this case, evaluating the emulator's mean involves taking a weighted average of Equation 8 over the ensemble of $\underline{\theta}$ vectors.

Another more common estimation approach uses optimization to find the set of correlation parameters $\underline{\theta}$ that maximizes the likelihood of the model given the data. It is equivalent, and more convenient to maximize the natural logarithm of the likelihood, which assuming a vague prior is,

$$\begin{aligned}
\log\left(\mathrm{lik}\left(\underline{\theta}\right)\right) &= -\frac{1}{2}\left(\left(N - N_\beta\right)\left(\frac{\hat{\sigma}^2}{\sigma^2} + \log\left(\sigma^2\right) + \log(2\pi)\right) + ... \right. \\
&\qquad \left. \log\left(\det\left(\underline{\underline{R}}\right)\right) + \log\left(\det\left(\underline{\underline{G}}\ \underline{\underline{R}}^{-1}\underline{\underline{G}}^T\right)\right)\right).
\end{aligned}$$

And, if one substitutes the maximum likelihood estimate $\hat{\sigma}^2$ in for $\sigma^2$, then it is equivalent to minimize the following objective function

$$\mathrm{obj}\left(\underline{\theta}\right) = \log\left(\hat{\sigma}^2\right) + \frac{\log\left(\det\left(\underline{\underline{R}}\right)\right) + \log\left(\det\left(\underline{\underline{G}}\ \underline{\underline{R}}^{-1}\underline{\underline{G}}^T\right)\right)}{N - N_\beta}.$$

Because of the division by $N - N_\beta$, this "per-equation" objective function is mostly independent of the number of sample points, $N$. It is therefore useful for comparing the (estimated) "goodness" of Kriging models that have different numbers of sample points; this will be important later.

In the Surfpack Kriging model, the domain of correlation length space over which the objective function is optimized is $d/4 \leq L_k \leq 8d$ where $d$ is the average distance between points; $d = N^{-1/M}$ when the input space has be normalized to a unit hypercube or unit hyper-rectangle (centered at zero). Here again $M$ is the number of input variables. In Surfpack this normalization is always done because it make the definition of $d$ simple and significantly improves the conditioning of $\left(\underline{\underline{G}}\;\underline{\underline{R}}^{-1}\underline{\underline{G}}^T\right)$. The user can use the `lower_bounds` and `upper_bounds` keywords (each of these keywords is followed by a list of $M$ real numbers) to specify the size of the input space. If bounds are not specified, then the minimum and maximum values of the coordinates of build data points in each dimension are used. The `dimension_groups` keyword can be used to specify $M$ integers to indicate which group each input belongs to, if this is done then the distance aspect ratios within each group is preserved when the build data is scaled, i.e. the scaled inputs space will be a unit hyper-rectangle rather than a unit hypercube. By default each dimension is considered to have its own group.

The options for `optimization_method` available in the Surfpack Kriging model are

- `global` optimization using the DIRECT (DIvision of RECTangles) algorithm (this is the default),
- `local` gradient-based optimization using the CONMIN (CONstrained MINimization) optimizer, (one or multiple starting locations can be used, by default a single starting location, the center of the search region in $\log_2(L)$ space, is used you can also specify a starting location using the `correlation_lengths` keyword),
- `global_local` or coarse `global` polished by `local` optimization,
- `sampling` or optimizing by guessing randomly and picking the best guess (by default $2M + 1$ guesses are used, you can use the `correlation_lengths` keyword to specify one guess and increase the total to $2M + 2$ guesses, and or directly specify the maximum number of guesses using the `max_trials` keyword)
- `none` uses the center of the search region in $\log_2(L)$ space or the set of correlation lengths specified using the `correlation_lengths` keyword.

The optimization used to determine $\underline{\theta}$ is performed under the constraint that $\underline{\underline{R}}$ is not ill-conditioned; specifically that $2^{-40} < \text{rcond}\left(\underline{\underline{R}}\right)$. As indicated above, the keywords `nugget` and `find_nugget` can be used to avoid ill-conditioning and enable a larger set of $\underline{\theta}$ to be considered during the optimization process.

Ill-conditioning of $\underline{\underline{R}}$ can also be addressed by a using pivoted Cholesky decomposition of $\underline{\underline{R}}$ to rank points according to how much unique information they containing (given an assumed $\underline{\theta}$). Trailing "low information" points can then be discarded until the retained portion of $\underline{\underline{R}}$ is no longer ill-conditioned. The discarded points are the ones with the least unique information and are therefore the ones that are easiest to predict. A different optimal set of points is retained for each $\underline{\theta}$ considered during the optimization process. The set of retained points associated with the optimal $\underline{\theta}$ is therefore "the best of the best" possible subsets. This is done by default in the Surfpack Kriging model when neither the `nugget` nor `find_nugget` keywords are specified. This approach uses the negative of the per-equation log likelihood as the objective function to make a fair comparison between different sizes of subsets.

Because the Kriging $\underline{\underline{R}}$ matrix has unit diagonal, the first point (row) will always be retained by the pivoted Cholesky algorithm. The user can alternately specify rentention of a different "anchor point" via the `anchor_index` keyword followed by a integer $0 \leq a < N$.

| Keyword | Value | Default |
|---|---|---|
| `derivative_order` | 0\|1 | 0 |
| `anchor_index` | $0 \leq$ integer $< N$ | 0 |
| `lower_bounds` | $M$ real numbers | minimum coordinate in each of the $M$ input dim |
| `upper_bounds` | $M$ real numbers | maximum coordinate in each of the $M$ input dim |
| `dimension_groups` | $M$ integers/group numbers | each of the $M$ dimension is scaled independe |
| `optimization_method` | `global`\|`local`\|`global_local`\|`sampling`\|`none` | `global` |
| `num_starts` | $1 \leq$ integer | 1 |
| `correlation_lengths` | $M$ real numbers | center of the search region in log(correlation lengt |
| `max_trials` | $1 \leq$ integer | varies by optimization method |
| `order` | $0 \leq$ integer | |
| `reduced_polynomial` | 1 | |
| `powered_exponential` | $1.0 \leq$ real number $\leq 2.0$ | |
| `matern` | 0.5\|1.5\|2.5\|`infinity` | |
| `find_nugget` | 1 | |
| `nugget` | $0.0 \leq$ real number | 0.0 |

Table 1: Table of the options available for the Kriging Model

#### 4.2.8 Table Of Kriging Options

## 4.3 Artificial Neural Network

The artificial neural network (ANN) surface fitting method in Surfpack employs a stochastic layered perceptron (SLP) based on the direct training approach of Zimmerman [13]. The SLP ANN method is designed to have a lower training cost than traditional ANNs. It uses fixed-value weights on some of the links within the network. That is, only a portion of the network weights must be computed in the ANN training process. While this approach offers a lower training cost than traditional ANNs, it also sacrifices some modeling flexibility. The form of the SLP ANN model is

$$\hat{f}(\mathbf{x}) = \tanh(\tanh((\mathbf{x}\mathbf{A}_0 + \theta_0)\mathbf{A}_1 + \theta_1)) \tag{9}$$

where $\mathbf{x}$ is the current point in $n$-dimensional parameter space, and the terms $\mathbf{A}_0, \theta_0, \mathbf{A}_1, \theta_1$ are the matrices and vectors that correspond to the neuron weights and offset values in the ANN model. These terms are computed during the ANN training process and are analogous to the coefficients in a polynomial function approximation. A singular value decomposition method is used to compute the network weights and offsets.

The SLP ANN is a non-parametric surface fitting method. Thus, along with kriging and MARS, it can be used to model data trends that have slope discontinuities as well as multiple maxima and minima. However, unlike kriging, the ANN surface is not guaranteed to exactly match the response values of the data points from which it was constructed. Thus, this ANN method provides some data smoothing similar to that provided by the low-order polynomials.

## 4.4 Multivariate Adaptive Regression Spline (MARS) Models

The multivariate adaptive regression splines (MARS) function approximation method is based on a recursive partitioning algorithm involving truncated power spline basis functions [14]. The form of the MARS model is

$$\hat{f}(\mathbf{x}) = a_0 + \sum_{m=1}^{M_1} a_m B_m(x_i) + \sum_{m=1}^{M_2} a_m B_m(x_i, x_j) + \ldots \tag{10}$$

where the $B_m$ terms are the basis functions, the $a_m$ terms are the coefficients and $M_n$ is the number of $n$-parameter basis functions. The MARS software partitions the parameter space into subregions and then applies a forward/backward selection process to add/remove basis functions from the model. The $a_m$ coefficients are generated using a regression algorithm. The user may choose linear or cubic-spline basis functions. With cubic basis functions, the resulting model is $C^2$ continuous.

MARS is a nonparametric surface fitting method and can represent complex multimodal data trends. The regression component of MARS generates a surface model that is not guaranteed to pass through all of the response data values. Thus, like the quadratic polynomial model, it provides some smoothing of the data. While the MARS algorithm is capable of producing a model from a very small number of samples, the user should not expect such models to generalize well.

MARS may take any of the following parameters:

- **Integer** `max_bases`: the maximum number of basis functions that can be incorporated into the model. With a greater number of basis functions, MARS has more flexibility to fit the data well but is also more prone to over-fitting. Increasing the number of allowable basis functions also increases the time needed to create the model. The default is 15.

- **Integer** `max_interactions`: the maximum number of variables that can be used in any single basis function. This is analagous to the `order` parameter for polynomial regression. The default is 2.

- **Identifier** `interpolation`. The argument value should be `linear` for first-order basis functions or `cubic` for third-order basis functions. The default is `cubic`, and this causes MARS to create a $C^2$-continuous model.

## 4.5 Radial Basis Functions

Radial basis functions are functions whose value typically depends on the distance from a center point, called the centroid, **c**. The surrogate model approximation is then built up as the sum of K weighted radial basis functions:

$$\hat{f}(\mathbf{x}) = \sum_{k=1}^{K} w_k \phi(\| \mathbf{x} - \mathbf{c_k} \|) \tag{11}$$

where the $\phi$ are the individual radial basis functions. These functions can be of any form, but often a Gaussian bell-shaped function or splines are used. Our implementation uses a Gaussian radial basis function. The weights are determined via a linear least squares solution approach. See [15] for more details.

## 4.6 Moving Least Squares

Moving Least Squares can be considered a more specialized version of linear regression models. In linear regression, one usually attempts to minimize the sum of the squared residuals, where the residual is defined as the difference between the surrogate model and the true model at a fixed number of points. In weighted least squares, the residual terms are weighted so the determination of the optimal coefficients governing the polynomial regression function, denoted by $\hat{f}(\mathbf{x})$, are obtained by minimizing the weighted sum of squares at N data points:

$$\sum_{n=1}^{N} w_n(\| \hat{f}(\mathbf{x_n}) - f(\mathbf{x_n}) \|) \tag{12}$$

Moving least squares is a further generalization of weighted least squares where the weighting is "moved" or recalculated for every new point where a prediction is desired. [16] The implementation of moving least squares is still under development. We have found that it works well in trust region methods where the surrogate model is constructed in a constrained region over a few points. It does not appear to be working as well globally, at least at this point in time.

# 5 Fitness Metrics

Surfpack provides several error metrics which can be used to assess the quality of a function approximation and to predict how well the model might generalize to unseen data. All of these metrics require a set of data for which the true function values are known. The error measures summarize the differences between the true response values and the approximating model's estimates at the same points.

For a given data point $i$, the difference between the true (observed) response value $o_i$ and the model's prediction $p_i$ is the *residual.* Since the residuals for known data points are often added together to produce a summary error measure, the absolute values of the residuals $|o_i - p_i|$ or squared residuals $(o_i - p_i)^2$ can be used to ensure that positive and negative residuals do not cancel each other out. In applications where the response values in different regions of the parameter space vary by orders of magnitude, scaled residuals $|\frac{o_i - p_i}{o_i}|$ can also be useful. Surfpack supports metrics which give the sum, arithmetic mean, or maximum of the absolute, squared, or scaled residuals. The name of the error measure is given as the value of the `metric` argument in the `Fitness` command. The metrics are named `sum_squared, mean_squared, max_squared, mean_abs, sum_scaled`, etc. The square root of the mean squared error (RMS) is also commonly used in many fields and is available in Surfpack as `root_mean_squared`.

The $R^2$ fitness metric was developed for use with polynomial regressions. The formula is

$$R^2 = \frac{\sum_{i=1}^{n} (p_i - \bar{o})^2}{\sum_{i=1}^{n} (o_i - \bar{o})^2}, \tag{13}$$

where $n$ is the number of data points used to create the model, and $\bar{o}$ is the mean of the true response values. The metric, named `rsquared` in Surfpack, quantifies the amount of variability in the data that is captured by the model. The value of $R^2$ falls on in the interval $[0, 1]$. Values close to 1 indicate that the model matches the data closely.

The class of $k$-fold cross-validation metrics is used to predict how well a model might generalize to unseen data. The training data is randomly divided into $k$ partitions. Then $k$ models are computed, each excluding the corresponding $k^{th}$ partition of the data. Each model is evaluated at the points that were excluded in its generation. The sum of the squared residuals over all $k$ models is the cross-validation error for a model that uses all of the available data. To use a cross-validation metric, the user should enter `cv` as the value of for the `metric` argument to the `Fitness` command and supply an additional integer parameter $k$. A special case, when $k$ is equal to the number of data points, is known as leave-one-out cross-validation or prediction error sum of squares (PRESS) and can be accessed in the `Fitness` command with `metric = press`.

Users should exercise great care in applying and interpreting the results of these error metrics. Not all metrics are applicable in to every surface fitting algorithm, or to every application. For example, metrics involving scaled residuals are probably not appropriate for data sets which include response values at or near 0 because the scaled residuals would be undefined. Users should also be aware that surface approximations with "better" values for some particular metric are not necessarily more desirable models. In particular, algorithms with many degrees of freedom can be prone to over-fitting (producing models that match the training data very closely but generalize poorly to unseen data). It should also be noted that in some cases, choosing a metric is a matter of preference rather than principle (*e.g.* the difference between `mean_squared` and `sum_squared` is only a constant factor). Goodness-of-fit metrics provide a valuable tool for analyzing and comparing models but must not be applied blindly.

# 6 Examples

## 6.1 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 even the fastest matrix inversion algorithms means that many interesting problems are simply intractable. 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 6.1 shows data for a Pentium IV machine.

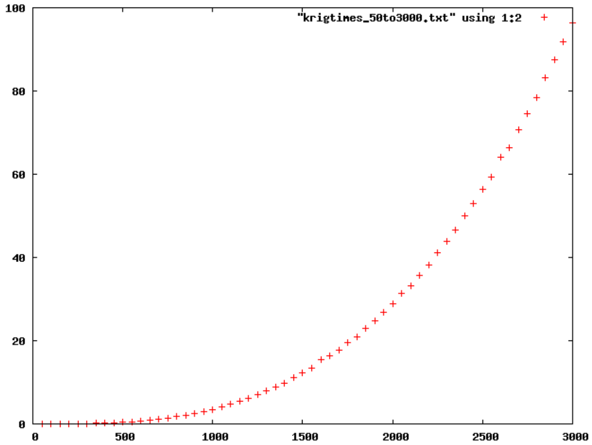


Figure 4: Running times in seconds for the execution of the Kriging algorithm, which is dominated by a matrix inversion, for data sets of size 50 to 3000.

The Kriging algorithm used in Surfpack—for which 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. A portion of the data file is shown in Figure 6.1.

This data was gathered for 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.

```
Load[name = timing_data, file = 'krigtimes_50to3000.txt', n_predictors = 1,
       n_responses = 2]
```

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 = timing_poly2, data = timing_data, response = 'median5',
        type = polynomial, order = 2]
```

```
% num_pts time_in_seconds
   5.00000000000000000e+01   7.05000013113021851e-04
   1.00000000000000000e+02   3.78199992701411247e-03
   1.50000000000000000e+02   1.05759999714791775e-02
   2.00000000000000000e+02   2.33480000169947743e-02
   2.50000000000000000e+02   4.59529999643564224e-02
   ...       ...
   2.90000000000000000e+03   8.74331180000444874e+01
   2.95000000000000000e+03   9.18161309999413788e+01
   3.00000000000000000e+03   9.63073910000966862e+01
```

Figure 5: Timing data for execution of the Kriging algorithm on data sets with 50–3000 points.

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 = '50 5000']
CreateSample[name = test_timing_data, axes = test_axes, grid_points = (50)]
Evaluate[surface = timing_poly2, data = test_timing_data]
Save[surface = timing_poly2, file = 'timing_poly2.sps']
Save[data = test_timing_data, file = 'test_timing_data.spd']
```

Figure 6.1 shows a portion of the output file `test_timing_data.txt`, which lists the predictions of the model for problem sizes of 50 to 5050, at 100 point intervals. A plot of the observed data and model predictions is shown in Figure 6.1.

```
%       'num_pts'              'time_in_seconds_est'
   5.00000000000000000e+01   4.19372236767611373e+00
   1.00000000000000000e+02   3.32891937004349936e+00
   1.50000000000000000e+02   2.54549093625508993e+00
   ...
   4.90000000000000000e+03   2.99187800855931528e+02
   4.95000000000000000e+03   3.06216330551186843e+02
   5.00000000000000000e+03   3.13326234810286337e+02
```

Figure 6: Timing predictions for quadratic polynomial fit.

Figure 6.1 shows an excerpt from quad_poly_snippet.txt, which shows the formula for the quadratic approximation.

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

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 timing_poly2 on timing_data: 1.48252
```

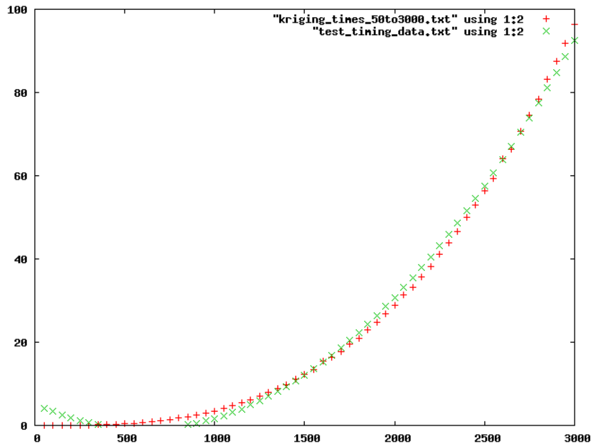


Figure 7: Measured execution times and quadratic polynomial model predictions for Kriging timing study.

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

Figure 8: Analytic form of quadratic polynomial model.

The value of 1.48 means that, on average, the predicted running time differs from the reported running time by 1.48 seconds.

Two other common goodness-of-fit metrics are PRESS and $R^2$.

```
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 $R^2$ 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.

```
Load[name = timing_data, file = ’krigtimes_50to3000.spd’, n_predictors = 1,
        n_responses = 2]
CreateAxes[name = test_axes, bounds = ’50 5000’]
CreateSample[name = test_timing_data, axes = test_axes, grid_points = (50)]
CreateSurface[name = poly3, data = timing_data, response = ’median5’,
type = polynomial, order = 3]
Evaluate[surface = poly3, data = test_timing_data]
Fitness[surface = poly3, data = timing_data, metric = mean_abs]
Fitness[surface = poly3, data = timing_data, metric = press]
Fitness[surface = poly3, data = timing_data,  metric = rsquared]
Save[surface = poly3, file = ’cubic_poly_timing.txt’]
Save[data = test_timing_data, file = ’test_timing_data.txt’]

#mean_abs for poly3: 26.095
#press for poly3: 38.8014
#rsquared for poly3: 0
```

All of the metrics are worse; this raises some red flags. In particular, it is not possible for the $R^2$ value to be lower for a least-squares fit to a cubic polynomial than for the corresponding quadratic.

The file `TimingMatrixOp/timing_poly3.txt` shows the coefficients for the model.

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

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

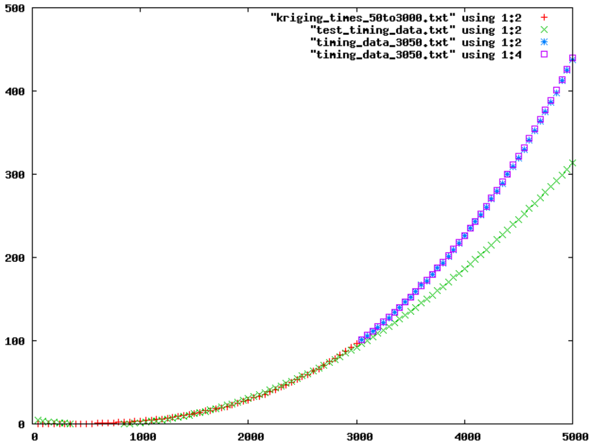
The cause of the problem is matrix ill-conditioning. The range of the problem sizes is 50–3000, while the running times range from a fraction of a second up to about 100 seconds. To address this problem, we scale the data so that data fall in the range $[0, 1]$.

```
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
```

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 impressive. For a problem size of 5000, the true running time was 437 seconds and the prediction was 439 seconds.

## 6.2 SAT Scores

The data set in `examples/TestScores/sat_scores.spd` give the average Scholastic Achievement Test (SAT) scores, by state, for students during 1982 [17], [18]. The predictor variables for this study were the percentage of the high school seniors taking the test, the median household income for the test takers, the average number of years (in high school) of core courses taken (math, science, English, etc.), the percentage of test takers attending public schools, the average per capita expenditure of the state for education, and the median rank of the test takers in their respective high school classes.

The script shown in Figure 6.2 shows commands that construct several possible models for the data. The constructed surfaces are saved to files for possible future use. The PRESS statistic is computed for each model. The results of the script suggest that the MARS model may have the best predictive capabilities for this application.

```
Load[name = sat_scores, file = 'sat_scores.spd', n_predictors = 6,
  n_responses = 1]
CreateSurface[name = sat_poly1, data = sat_scores, type = polynomial, order = 1]
CreateSurface[name = sat_poly2, data = sat_scores, type = polynomial, order = 2]
CreateSurface[name = sat_kriging, data = sat_scores, type = kriging]
#CreateSurface[name = sat_mars, data = sat_scores, type = mars]
CreateSurface[name = sat_ann, data = sat_scores, type = ann]

#Save[surface = sat_poly1, file = 'sat_poly1.sps']
#Save[surface = sat_poly2, file = 'sat_poly2.sps']
#Save[surface = sat_kriging, file = 'sat_kriging.sps']
#Save[surface = sat_mars, file = 'sat_mars.sps']
#Save[surface = sat_ann, file = 'sat_ann.sps']

Fitness[surface = sat_poly1, data = sat_scores, metric = press]
Fitness[surface = sat_poly2, data = sat_scores, metric = press]
Fitness[surface = sat_kriging, data = sat_scores, metric = press]
#Fitness[surface = sat_mars, data = sat_scores, metric = press]
Fitness[surface = sat_ann, data = sat_scores, metric = press]
```

Suppose that education policy makers in Wisconsin wish to use this data to help them get an idea of how their students' SAT scores might be influenced by factors over which they might have some influcence *e.g.*, average expenditure per pupil or number of core courses taken in high school. (Certainly, they should not place too much emphasis on the analysis of this data, since the predictor variables themselves are summaries of many other variables with potentially complicated interactions. Modeling of this data could be used as one of many tools in a broader analysis.)

The data `what_ifs.spd` include minor variations from the actual Wisconsin data point: an increase in the number of core courses, changes in per pupil spending, etc. The script `sat_scores2.spk` reads in the MARS model that was saved in the earlier script, and then evaluates the model on these specific query points.

```
Load[name = what_ifs, file = 'what_ifs.spd', n_predictors = 6, n_responses = 0]
Load[name = sat_mars, file = 'sat_mars.sps']
Evaluate[surface = sat_mars, data = what_ifs, label = SAT_est]
Save[data = what_ifs, file = 'what_ifs_estimates.spd']
```

## 6.3 Martian Topology

Figure 6.3 shows a script that uses both MARS and Kriging to fit data taken from the surface of (the planet) Mars. The data were sampled from 25–26° N latitude and 176–177° E longitude. The models are constructed using a sparse training set and then analyzed for their accuracy on both the training set and a more densely sampled test set. (Data courtesy of NASA.)

```
#Data courtesty of NASA
#Downloaded from http://pds-geosciences.wustl.edu/missions/mgs/megdr.html
#Accessed June 2006
Load[name = topo, file = ’n26e171.spd’, n_predictors = 2, n_responses = 1]
Load[name = dense_topo, file = ’n26e171dense.spd’, n_predictors = 2, n_responses = 1]
! echo data loaded
CreateSurface[name = topo_kriging, data = topo, type = kriging, correlations = (1.e3,1.e3)]
! echo created kriging
CreateSurface[name = topo_mars, data = topo, type = mars]
! echo created mars
Evaluate[surface = topo_kriging, data = dense_topo, response = ’kriging_est’]
! echo evaluated kriging
Evaluate[surface = topo_mars, data = dense_topo, response = ’mars_est’]
! echo evaluated mars
Save[data = dense_topo, file = ’n26e171dense_estimates.spd’]
! echo saved
Fitness[surface = topo_kriging, data = topo, metric = mean_scaled]
Fitness[surface = topo_mars, data = topo, metric = mean_scaled]
! echo fitness mean scaled
Fitness[surface = topo_kriging, data = topo, metric = max_scaled]
Fitness[surface = topo_mars, data = topo, metric = max_scaled]
! echo fitness max scaled
Fitness[surface = topo_kriging, data = topo, metric = rsquared]
Fitness[surface = topo_mars, data = topo, metric = rsquared]
! echo fitness rsquared
Fitness[surface = topo_kriging, metric = mean_scaled, data = dense_topo]
Fitness[surface = topo_mars, metric = mean_scaled, data = dense_topo]
! echo fitness mean scaled
Fitness[surface = topo_kriging, metric = max_scaled, data = dense_topo]
Fitness[surface = topo_mars, metric = max_scaled, data = dense_topo]
! echo fitness max scaled
Fitness[surface = topo_kriging, metric = rsquared, data = dense_topo]
Fitness[surface = topo_mars, metric = rsquared, data = dense_topo]
! echo fitness rsquared
```

## 6.4 Sampling Techniques

This example explores the interaction between sampling method (Monte Carlo, Latin Hypercube, Orthogonal Array) and surface-fitting algorithm (Kriging and MARS) on a commonly used test function, the Rosenbrock "banana" function. Since this is one of Surfpack's built-in test functions, large test data sets are easily created and analyzed.

### 6.5 Computational Fluid Dynamics

```
# Read in data
Load[name = cfd, file = ’cfd.spd’, n_predictors = 1,
  n_responses = 1]

# Create a test set with 27 points, evenly spaced .2 apart
CreateAxes[name = ax_1d, bounds = ’-4 1.2 ’]
CreateSample[name = test_data, axes = ax_1d, grid_points = (27),
  labels = (x)]

CreateSurface[name = poly1_cfd, data = cfd, type = polynomial, order = 1]
CreateSurface[name = poly2_cfd, data = cfd, type = polynomial, order = 2]
CreateSurface[name = poly3_cfd, data = cfd, type = polynomial, order = 3]
CreateSurface[name = mars_cfd, data = cfd, type = mars]
CreateSurface[name = kriging_cfd, data = cfd, type = kriging]
CreateSurface[name = kriging_cfd_user_corr, data = cfd, type = kriging,
  correlations = (1.0)]
CreateSurface[name = ann_cfd, data = cfd, type = ann]

Evaluate[surface = poly1_cfd, data = test_data, label = poly1]
Evaluate[surface = poly2_cfd, data = test_data, label = poly2]
Evaluate[surface = poly3_cfd, data = test_data, label = poly3]
Evaluate[surface = mars_cfd, data = test_data, label = mars]
Evaluate[surface = kriging_cfd, data = test_data, label = kriging]
Evaluate[surface = kriging_cfd_user_corr, data = test_data, label = krig_usr]
Evaluate[surface = ann_cfd, data = test_data, label = ann]
Save[data = test_data, file = ’test_data.spd’]

Fitness[surface = poly1_cfd, data = cfd, metric = press]
Fitness[surface = poly2_cfd, data = cfd, metric = press]
Fitness[surface = poly3_cfd, data = cfd, metric = press]
Fitness[surface = mars_cfd, data = cfd, metric = press]
Fitness[surface = kriging_cfd, data = cfd, metric = press]
Fitness[surface = kriging_cfd_user_corr, data = cfd, metric = press]
Fitness[surface = ann_cfd, data = cfd, metric = press]

Fitness[surface = poly1_cfd, data = cfd, metric = root_mean_squared]
Fitness[surface = poly2_cfd, data = cfd, metric = root_mean_squared]
Fitness[surface = poly3_cfd, data = cfd, metric = root_mean_squared]
Fitness[surface = mars_cfd, data = cfd, metric = root_mean_squared]
Fitness[surface = kriging_cfd, data = cfd, metric = root_mean_squared]
Fitness[surface = kriging_cfd_user_corr, data = cfd, metric = root_mean_squared]
Fitness[surface = ann_cfd, data = cfd, metric = root_mean_squared]

Fitness[surface = poly1_cfd, data = cfd, metric = rsquared]
Fitness[surface = poly2_cfd, data = cfd, metric = rsquared]
Fitness[surface = poly3_cfd, data = cfd, metric = rsquared]
```

### 6.6 Matlab Peaks Function

```
# Read in data
Load[name = peaks_data, file = 'matlab_peaks.spd', n_predictors = 2,
  n_responses = 1]

# Create a test set with 27 points, evenly spaced .2 apart
CreateAxes[name = ax_2d, bounds = '-3 3 | -3 3 ']
CreateSample[name = test_data, axes = ax_2d, grid_points = (13,13),
  labels = (x1,x2)]

CreateSurface[name = poly1_peaks, data = peaks_data, type = polynomial, order = 1]
CreateSurface[name = poly2_peaks, data = peaks_data, type = polynomial, order = 2]
CreateSurface[name = poly3_peaks, data = peaks_data, type = polynomial, order = 3]
CreateSurface[name = kriging_peaks_global_corr, data = peaks_data, type = kriging, lower_bounds = (-3.0, -3.0), up
CreateSurface[name = kriging_peaks_local_corr, data = peaks_data, type = kriging, lower_bounds = (-3.0, -3.0), upp
CreateSurface[name = kriging_peaks_multi_local_corr, data = peaks_data, type = kriging, lower_bounds = (-3.0, -3.0
CreateSurface[name = kriging_peaks_sampling_corr, data = peaks_data, type = kriging, lower_bounds = (-3.0, -3.0),
CreateSurface[name = kriging_peaks_user_corr, data = peaks_data, type = kriging,
  correlation_lengths = (0.335428, 2.68286), optimization_method = none] #these lengths are the ones found using g
CreateSurface[name = ann_peaks, data = peaks_data, type = ann]

Evaluate[surface = poly1_peaks, data = test_data, label = poly1]
Evaluate[surface = poly2_peaks, data = test_data, label = poly2]
Evaluate[surface = poly3_peaks, data = test_data, label = poly3]
Evaluate[surface = kriging_peaks_global_corr, data = test_data, label = kriging_global]
Evaluate[surface = kriging_peaks_local_corr, data = test_data, label = kriging_local]
Evaluate[surface = kriging_peaks_multi_local_corr, data = test_data, label = kriging_multi_local]
Evaluate[surface = kriging_peaks_sampling_corr, data = test_data, label = kriging_sampling]
Evaluate[surface = kriging_peaks_user_corr, data = test_data, label = krig_usr]
Evaluate[surface = ann_peaks, data = test_data, label = ann]
Save[data = test_data, file = 'test_data.spd']

Fitness[surface = poly1_peaks, data = peaks_data, metric = press]
Fitness[surface = poly2_peaks, data = peaks_data, metric = press]
Fitness[surface = poly3_peaks, data = peaks_data, metric = press]
Fitness[surface = kriging_peaks_global_corr, data = peaks_data, metric = press]
Fitness[surface = kriging_peaks_local_corr, data = peaks_data, metric = press]
Fitness[surface = kriging_peaks_multi_local_corr, data = peaks_data, metric = press]
Fitness[surface = kriging_peaks_sampling_corr, data = peaks_data, metric = press]
Fitness[surface = kriging_peaks_user_corr, data = peaks_data, metric = press]
Fitness[surface = ann_peaks, data = peaks_data, metric = press]

Fitness[surface = poly1_peaks, data = peaks_data, metric = root_mean_squared]
Fitness[surface = poly2_peaks, data = peaks_data, metric = root_mean_squared]
Fitness[surface = poly3_peaks, data = peaks_data, metric = root_mean_squared]
Fitness[surface = kriging_peaks_global_corr, data = peaks_data, metric = root_mean_squared]
Fitness[surface = kriging_peaks_local_corr, data = peaks_data, metric = root_mean_squared]
Fitness[surface = kriging_peaks_multi_local_corr, data = peaks_data, metric = root_mean_squared]
Fitness[surface = kriging_peaks_sampling_corr, data = peaks_data, metric = root_mean_squared]
Fitness[surface = kriging_peaks_user_corr, data = peaks_data, metric = root_mean_squared]
Fitness[surface = ann_peaks, data = peaks_data, metric = root_mean_squared]

Fitness[surface = poly1_peaks, data = peaks_data, metric = rsquared]
Fitness[surface = poly2_peaks, data = peaks_data, metric = rsquared]
Fitness[surface = poly3_peaks, data = peaks_data, metric = rsquared]
```

### 6.7 Central Composite Design and Latin Hypercube Sampling

```
Load[name = ccd, file = 'ccd_lhs_4d_42p.spd', n_predictors = 4,
 n_responses = 1]
CreateAxes[name = ax4d, bounds = '0.5 1.5 | 1.5 4.5 | 0.7 1.0 | 0.7']
CreateSample[name = test_data, axes = ax4d, grid_points = (11,11,11,1)]
CreateSurface[name = ccd_poly1, data = ccd, type = polynomial, order =1]
CreateSurface[name = ccd_poly2, data = ccd, type = polynomial, order =2]
CreateSurface[name = ccd_poly3, data = ccd, type = polynomial, order =3]
CreateSurface[name = ccd_kriging, data = ccd, type = kriging]
CreateSurface[name = ccd_ann, data = ccd, type = ann]

Evaluate[surface = ccd_poly1, data = test_data, label = poly1]
Evaluate[surface = ccd_poly2, data = test_data, label = poly2]
Evaluate[surface = ccd_poly3, data = test_data, label = poly3]
Evaluate[surface = ccd_kriging, data = test_data, label = kriging]
Evaluate[surface = ccd_ann, data = test_data, label = ann]
Save[data = test_data, file = 'test_data.spd']

Fitness[surface = ccd_poly1, data = ccd, metric = rsquared]
Fitness[surface = ccd_poly2, data = ccd, metric = rsquared]
Fitness[surface = ccd_poly3, data = ccd, metric = rsquared]

Fitness[surface = ccd_poly1, data = ccd, metric = root_mean_squared]
Fitness[surface = ccd_poly2, data = ccd, metric = root_mean_squared]
Fitness[surface = ccd_poly3, data = ccd, metric = root_mean_squared]
Fitness[surface = ccd_kriging, data = ccd, metric = root_mean_squared]
Fitness[surface = ccd_ann, data = ccd, metric = root_mean_squared]

Fitness[surface = ccd_poly1, data = ccd, metric = press]
Fitness[surface = ccd_poly2, data = ccd, metric = press]
Fitness[surface = ccd_poly3, data = ccd, metric = press]
Fitness[surface = ccd_kriging, data = ccd, metric = press]
Fitness[surface = ccd_ann, data = ccd, metric = press]
```

# 7 Troubleshooting

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

#### 7.0.1 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.

#### 7.0.2 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 setsin 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.

### 7.0.3 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.

### 7.0.4 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().

### 7.0.5 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.

### 7.0.6 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.

### 7.0.7 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.

### 7.0.8 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.

### 7.0.9 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.

### 7.0.10 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.

### 7.0.11 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 sample for some surfaces before the dimensionality of the data is determined. In many cases the required number of points is a function of the arity of the data.

### 7.0.12 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.

### 7.0.13 Dimension mismatch: correlations and data dimensionality

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

### 7.0.14 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.

### 7.0.15 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.

### 7.0.16 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.

### 7.0.17 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.

### 7.0.18 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.

### 7.0.19 Cannot write SurfData object to stream. No active data points.

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

### 7.0.20 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.

### 7.0.21 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.

#### 7.0.22 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.

#### 7.0.23 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 $0, 1, \ldots, n-1$.

#### 7.0.24 Exception caught and rethrown in SurfPoint::readText

#### 7.0.25 Exception rethrown in SurfPoint::readBinary

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

#### 7.0.26 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.

#### 7.0.27 Expected 'f' or 'v' on line

The first line of an axes object should be the number of dimensions desired in theresulting 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 valuefor 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').

#### 7.0.28 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 the 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.

#### 7.0.29 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.

#### 7.0.30 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.

#### 7.0.31 Must know data arity 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.

### 7.0.32 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.

### 7.0.33 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.

### 7.0.34 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.

### 7.0.35 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.

### 7.0.36 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.

### 7.0.37 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’.

### 7.0.38 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.

### 7.0.39 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.

### 7.0.40 No surface or data argument specified.

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

### 7.0.41 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.

### 7.0.42 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.

### 7.0.43 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 $n - 1$.

### 7.0.44 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 duplicatations of other excluded points.

### 7.0.45 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.

### 7.0.46 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.

### 7.0.47 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.

### 7.0.48 Unrecognized filename extension. Use .sd or .txt

### 7.0.49 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.

## A Surfpack Syntax Summary

TODO: Briefly explain the types of arguments: integer, real, identifier, string, list, etc. In the status column, R means required and O means optional. Status designations that span multiple rows signify that these arguments are mutually exclusive.

# References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorenson, Lapack user's guide (1999).

[2] A. A. Giunta, L. Watson, A comparison of approximation modeling techniques: polynomial versus interpolating models, in: Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO, 1998, pp. 392–404.

[3] V. J. Romero, L. P. Swiler, A. A. Giunta, Construction of response surfaces based on progressiv-lattice-sampling experimental designs, Structural Safety 26 (2) (2004) 201–219.

[4] J. R. Koehler, A. B. Owen, Computer experiments, Handbook of Statistics 13 (1996) 261–308.

[5] N. A. Cressie, Statistics for Spatial Data, John Wiley and Sons, New York, 1993.

[6] G. Matheron, The theory of regionalized variables and its applications, Les Cahiers du Centre de morphologie mathématique de Fontainebleau, École national supérieure des mines, 1971.
URL `http://books.google.com/books?id=TGhGAAAAYAAJ`

[7] T. Gneiting, M. Genton, P. Guttorp, Geostatistical space-time models, stationarity, separability and full symmetry, in: B. Finkenstadt, L. Held, V. Isham (Eds.), Statistical Methods for Spatio-Temporal Systems, Boca Raton: Chapman & Hall/CRC, 2007, Ch. 4, pp. 151–172.

[8] C. Berg, J. Mateu, E. Porcu, The dagum family of isotropic correlation functions, Bernoulli 14 (4) (2008) 1134–1149.

[9] G. Davis, M. Morris, Six factors which affect the condition number of matrices associated with kriging, Mathematical geology 29 (5) (1997) 669–683.

[10] J. Martin, Computational improvements to estimating kriging metamodel parameters, Journal of Mechanical Design 131 (2009) 084501.

[11] G. Rennen, Subset selection from large datasets for kriging modeling, Structural and Multidisciplinary Optimization 38 (6) (2009) 545–569.

[12] M. Eldred, B. Adams, D. Gay, L. Swiler, K. Haskell, W. Bohnhoff, J. Eddy, W. Hart, J. Watson, J. Griffin, P. Hough, T. Kolda, P. Williams, M. Martinez-Canales, Dakota version 4.1 users manual, Sandia Technical Report SAND2006-6337, Sandia National Laboratories, Albuquerque, NM (2007).
URL `http://dakota.sandia.gov/licensing/release/Users4.1.pdf`

[13] D. C. Zimmerman, Genetic algorithms for navigating expensive and complex design spaces, Tech. Rep. AO-7736 CA 02, Sandia National Laboratories (1996).

[14] J. Friedman, Multivariate adaptive regression splines, Annals of Statistics 19 (1) (1991) 1–141.

[15] M. J. L. Orr, Introduction to radial basis function networks, Tech. rep., University of Edinburgh, Edinburgh, Scotland (1996).

[16] A. Nealen, A short-as-possible introduction to the least squares, weighted least squares, and moving least squares methods for scattered data approximation and interpolation, Tech. rep., Discrete Geometric Modeling Group, Technishe Universitaet, Berlin, Germany (2004).

[17] B. Powell, L. C. Steelman, Variations in state sat performance: Meaningful or misleading?, Harvard Educational Review 54 (4) (1984) 389–412.

[18] F. L. Ramsey, D. W. Schafer, The Statistical Sleuth: A Course in Methods of Data Analysis, Duxbury, Pacific Grove, CA, 2002.

| Command | Argument | | Type | Description |
|---|---|---|---|---|
| CreateAxes | `name` | R | identifier | unique name for new `axes` object |
| | `bounds` | R | string | min/max range pairs for each dimension |
| | `file` | | string | name of .axb file containing min/max range pairs for each dimension |
| | `labels` | O | identifier list | names for predictor variables in new `data` object |
| CreateSample | `name` | R | identifier | unique name for new `data` object |
| | `axes` | R | identifier | existing `axes` object to be used |
| | `grid_points` | R | integer list | number of points along each dimension in grid |
| | `size` | | integer | number of random samples to draw |
| | `test_functions` | O | identifier list | names of built-in test functions |
| CreateSurface | `name` | R | identifier | unique name for new `surface` object |
| | `type` | R | identifier | surface-fitting algorithm: `polynomial`, `mars`, `kriging`, `ann` |
| | `data` | R | identifier | existing `data` object from which to create new `surface` object |
| | `response` | O | identifier | name of response variable to be used |
| | `response_index` | | integer | index of response variable to be used |
| | `log_scale` | O | string list | names of variables to be scaled logarithmically |
| | `norm_scale` | O | string list | names of variables to be normalized to [0,1] |
| Evaluate | `surface` | R | identifer | existing `surface` to be evaluated |
| | `data` | R | identifier | existing `data` to evaluate |
| | `label` | O | string | name for new response variable |
| Fitness | `surface` | R | identifier | existing surface to be analyzed |
| | `metric` | R | identifier | goodness-of-fit metric to be used |
| | `data` | O | identifier | existing `data` object to be used to evaluate fitness |
| | `response` | O | identifier | response variable to be used as “true” function value |
| | `response_index` | | integer | index of response variable to be used |
| Load | `name` | R | identifier | unique name for `data` object |
| | `file` | R | string | data file (.spd) or surface file (.sps) |
| | `n_predictors` | R | integer | number of predictor variables per point (for `data` load only) |
| | `n_responses` | R | integer | number of response variables per point (for `data` load only) |
| Save | `data` | R | identifier | existing `data` object |
| | `surface` | | identifier | existing `surface` object |
| | `file` | R | string | filename for `data`/`surface` to be saved |

| Surface Type | Argument | Status | Type | Description |
|---|---|---|---|---|
| Polynomial | order | O | integer | maximum order of regression terms |
| Mars | max_bases | O | integer | maximum number of basis functions |
| | max_interactions | O | integer | maximum number of interactions between variables per basis |
| | interpolation | O | identifier | type of splines used: linear or cubic |
| Kriging | correlations | O | real list | correlation paramter for each variable |
| | uniform_correlation | O | real | uniform correlation value for all variables |
| | conmin_seed | O | real list | starting values maximum likelihood estimation of correlations |
| ANN | norm_bound | O | real | Does anyone know what this is? |
| | svd_factor | O | real | Does anyone know what this is? |
| | fraction_withheld | O | real | fraction of data to be excluded from training set |