The Geodise OptionsMatlab Toolbox

A User's Guide

Geodise

Release:	OptionsMatlab v.0.13.0
Version:	OptionsMatlabTutorial.doc 1.6.0
Title:	Geodise OptionsMatlab Tutorial – A User's Guide
Authors:	Dr Graeme Pound Dr Andrew Price, a.r.price@soton.ac.uk
PI:	Prof Andy Keane, andy.keane@soton.ac.uk Prof Simon Cox, s.j.cox@soton.ac.uk
Web:	http://www.geodise.org/
Copyright:	Copyright © 2007, The Geodise Project, University of Southampton

Contents

1 Introduction

OptionsMatlab integrates the Options design exploration and optimization package [1] into the Matlab environment [2]. The advantages of this approach include; the potential to rapidly prototype and debug objective and constraint functions, and the ability to directly leverage the functionality available within the Matlab environment. Matlab provides data analysis and visualisation capabilities. Additional functionality is available from a variety of toolboxes, including the Geodise toolboxes for Grid-enabled computational and data management [3].

OptionsMatlab provides access to all of the design search and optimisation algorithms within the Options package whilst retaining the maximum flexibility. Users define the objective and constraint functions that describe their problem as Matlab functions. These functions can therefore include interpreted Matlab, compiled MEX functions, or callbacks to external applications or to the Grid. The modular structure of OptionsMatlab is shown by Figure 1.

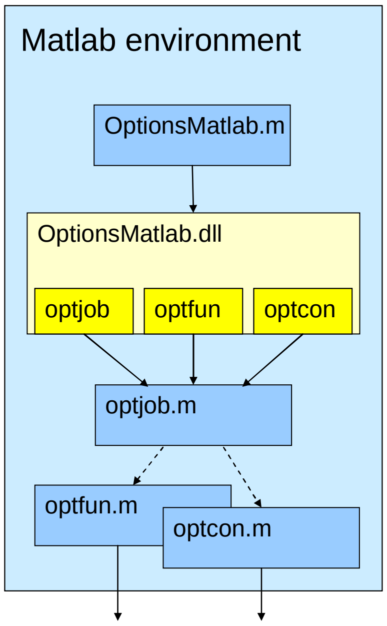


Figure 1 The modular structure of OptionsMatlab.

OptionsMatlab is invoked by calling the Matlab function `OptionsMatlab`. An input structure describes the user's problem, and configures the design search and optimisation algorithm to be used. Additionally a large number of optional fields may be used to adjust the Options control parameters. The results are returned to the Matlab workspace in an output structure

The usage of OptionsMatlab differs from the original Options interface. A programme of design search and optimisation is based upon a sequence of invocations of the OptionsMatlab function from the Matlab workspace. By comparison the Options interface supports a number of operations upon a data-set managed by the internal Options database. OptionsMatlab can be used to perform operations upon the results of a previous optimisation, such as building a Response Surface Model, by passing the results as a second input argument.

The modular structure of OptionsMatlab also allows the user to customise the evaluation of the objective and constraint functions by replacing the function `OPTJOB`. For example, the default implementation of `OPTJOB` supplied with OptionsMatlab, `optjob.m`, evaluates the objective and constraint functions in serial. An alternative job manager is provided which supports parallel function evaluations (see section 5.10).

This document provides an introduction to the use of the OptionsMatlab package. For further details of the theory of design search and optimisation, and the use of the Options package, please consult the Options manual [1].

OptionsMatlab has three modes of operation:

1. Direct Search. The specified optimisation algorithm is run over the user's problem, directly invoking the Matlab functions that define the objective and constraint functions.
2. Search Response Surface Model. A Response Surface Model (RSM) is built which models the behaviour of the user's problem. The RSM is built from the results of a previous design search, and is searched with the specified optimisation algorithm.
3. Hyper-parameter tuning. The Stochastic Process Model hyper-parameters which describe a Stochastic Process Model RSM must be tuned against an existing data set. The specified optimisation is used to tune the hyper-parameters against a data set describing the user's objective function (and/or constraints).

These modes are invoked depending upon the fields of the input structure (Figure 2).

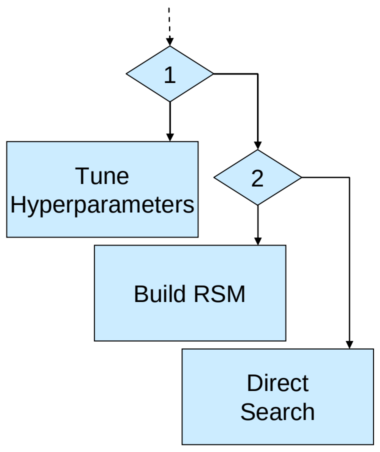


Figure 2 OptionsMatlab’s modes of operation. Hyper-parameter tuning will be invoked if the input field `TUNEHYPER` is set (#1). If the input field `OBJMOD` (or `CONMOD`) is set the specified optimisation will be run over a RSM (#2). In all other conditions a direct search over the user's objective function is used.

2 Installing OptionsMatlab

This section describes the steps needed to install OptionsMatlab.

2.1 Obtain a Gendat license file

Gendat licenses are linked to the MAC address (Windows) or hostid (UNIX) of the machine running OptionsMatlab, and are available from Prof. Andy Keane.

OptionsMatlab looks for a Gendat license file at the location specified by the environment variable `GENDAT_CODES`. If this environment variable is not specified then the UNIX version of OptionsMatlab looks for a Gendat license in the file `/usr/local/geodise/OptionsMatlab/gendat.cds` and if this file does not exist it will look for `gendat.cds` in the current directory. The Windows version of OptionsMatlab will look for a Gendat license in the file `C:\fortran\gendat\GENDAT.CDS` if the environment variable does not exist.

2.2 Add OptionsMatlab to the Matlab search path

The directory containing the OptionsMatlab functions should be added to the Matlab search path.

If using the Matlab desktop navigate to the 'Set Path' dialog ('File' > 'Set Path'). Click the 'Add Folder' button and browse to the directory containing the OptionsMatlab, select 'OK' to confirm. You may wish to click the 'Save' button to preserve the configuration between sessions. Click 'Close' to dismiss the dialog.

If using Matlab via the Unix terminal use the `addpath` function at the Matlab command line.

```
>> addpath /home/USER/OptionsMatlab
```

System administrators configuring a multi-user Matlab installation may find it preferable to edit $MATLABROOT/toolbox/local/pathdef.m to make changes to the Matlab search path available to all users.

To confirm that the Matlab search path has been successfully configured run:

```
>> str = which('OptionsMatlab')
```

The variable `str` should contain the path of the OptionsMatlab function.

2.3 View the documentation

To read the OptionsMatlab documentation type `'help OptionsMatlab'` at the Matlab command prompt. This text contains details of the input arguments to OptionsMatlab and the output structures returned. For information about the interfaces required by user-defined objective and constraint functions type `'help optjob'`.

3 OptionsMatlab Tutorial

3.1 Create a input structure

Use the function `createBeamStruct` to create an OptionsMatlab input structure. At the command line enter:

```
>> input = createBeamStruct
```

```
input =


      DNULL: -777
     OLEVEL: 2
    MAXJOBS: 10
       NVRS: 2
       VNAM: {'BREADTH'  'HEIGHT'}
      LVARS: [5 2]
      UVARS: [50 25]
       VARS: [30 20]
      NCONS: 5
       CNAM: {'SIGMA-B'  'TAU'  'DEFLN'  'H-ON-B'  'F-CRIT'}
      LCONS: [-777 -777 -777 -777 5000]
      UCONS: [200 100 5 10 -777]
    NPARAMS: 7
       PNAM: {'LENGTH'  'FORCE'  'FACTOR'  'EE'  'GG'  'NU'
'SIGMAY'}
     PARAMS: [1500 5000 2 216620 86650 0.2700 200]
       ONAM: 'AREA'
     OMETHD: 2.8000
     DIRCTN: -1
     NITERS: 500
     OPTFUN: 'beamobjfun'
     OPTCON: 'beamobjcon'
     OPTJOB: 'optjob'
```

The OptionsMatlab input structure describes the problem to be searched, including the design variables, constraints and parameters. The input structure will also include details of the optimisation or design search to be run over the problem. The function `createBeamStruct` is a utility function which creates an input structure specific to the Beam problem.

The fields of the structure `input` are described in detail by the documentation for `OptionsMatlab`. Of particular interest are the fields `OPTFUN` and `OPTCON` that specify the Matlab functions that describe the objective and constraint functions respectively. The objective and constraint functions used, `beamobjfun.m` and `beamobjcon.m`, is a Matlab implementation of the Beam problem described in the Options manual [1].

The field `OMETHD` is a scalar which specifies the search method to be used (see FAQ section 5.2 for further details). This example uses a Design of Experiments study, `OMETHD = 2.8`.

3.2 Run the search

OptionsMatlab can now be invoked with the input structure returned by `createBeamStruct`. At the command line enter:

```
>> results = OptionsMatlab(input)
```

```
results =


       VARS: [2x1 double]
     OBJFUN: 2.9455e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

OptionsMatlab will quickly perform a Design of Experiments study, evaluating the objective and constraint functions `NITERS` times. The results are returned in the structure `results`. The fields `OBJFUN` and `VARS` contain the minimum objective function found and the corresponding design variables. The field `CONS` contains the values of the constraints at the design variables `VARS`.

Other fields `OBJTRC` and `CONSTRC` contain the search histories over the objective and constraint functions. This information is valuable to examine the history of the optimisation and build Response Surface Models.

3.3 View the search histories

A simple tool is provided to view the search histories of problems with two design variables. At the command line enter:

```
    >> plotOptionsSurfaces(results, input)
```

This will produce plots for the objective function and each of the constraints against the two design variables at each of the design variables evaluated. The final plot shows the objective function plotted against the two design variables, in which each point is coloured depending whether it exceeds the constraints (red), or not (blue) (Figure 3).

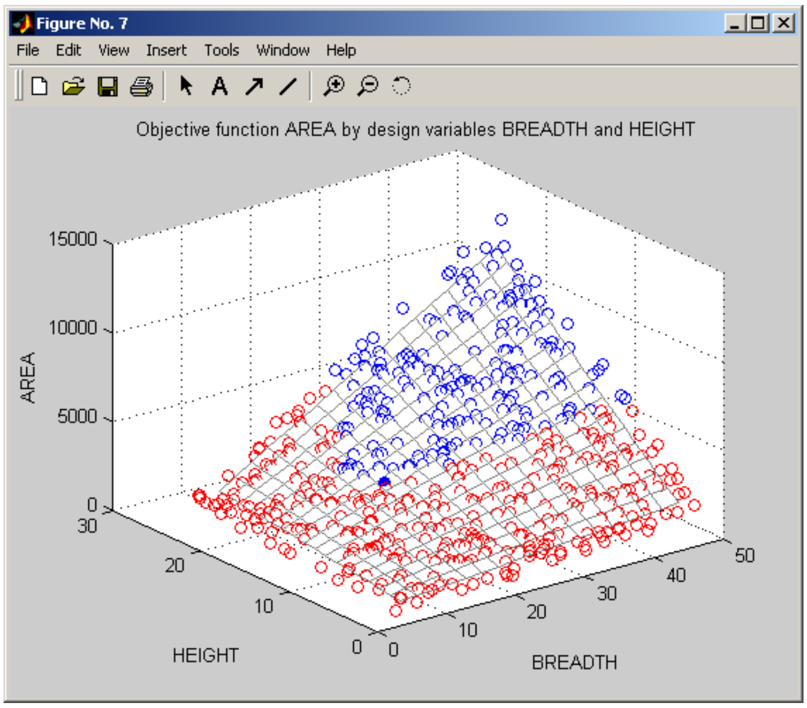


Figure 3 The results of a 500 point DoE plotted with `plotOptionsSurfaces`

3.4 Build and search a Response Surface Model

The results returned by the Design of Experiments can be used to build a Response Surface Model (RSM) that can be searched very rapidly. This approach may be suitable when either the objective or constraint functions are expensive to evaluate. To do this we must create another input structure, with the same problem definition. We will modify this input structure to specify that a RSM is used to evaluate the objective and constraint functions. At the command line enter:

```
    >> inputRSM = createBeamStruct;
```

```
>> inputRSM.OBJMOD = 3.5;
>> inputRSM.CONMOD = 3.5;
```

By specifying `OBJMOD` and `CONMOD` equal to 3.5 OptionsMatlab will produce a RSM using a second order polynomial regression model. For a list of the alternative RSM approximation methods available within OptionsMatlab see the FAQ section 5.4.

```
>> inputRSM.OMETHD = 4;
>> inputRSM.NITERS = 1000;
```

`OMETHD` equal to 4 specifies a Genetic Algorithm with 1,000 function evaluations. OptionsMatlab will perform the function evaluations required for the Genetic Algorithm against the RSM (rather than evaluating the user's objective or constraint functions directly).

The input structure `inputRSM` must be passed into OptionsMatlab together with the results of the Design of Experiments contained in the variable `results`.

```
>> resultsRSM = OptionsMatlab(inputRSM, results)
```

```
resultsRSM = 


      VARS: [2x1 double]
    OBJFUN: 2.4824e+003
      CONS: [5x1 double]
```

The results structure returned, `resultsRSM`, does not contain search histories. This is because the model used to evaluate the design variables is an approximation of the user’s model and should not be considered to be equivalent to direct evaluation. It is good practice to verify the results of a search over a RSM by direct evaluation of the objective and constraint functions at the returned optimum design.

4 Function Reference

Banana problem

An example of the unconstrained Banana problem based upon Rosenbrock's function.

$$f(x) = 100 \cdot (x_2 - x_1^2)^2 + (1 - x_1)^2$$

Equation 1 Rosenbrock's function

Example

This problem may be extended into multiple dimensions; however by default the problem is 2D. This example plots the objective function surface of the Banana problem.

```
>> input = createbananastruct(2.8, 2);
>> input.OMETHD = 2.8;
>> input.MC_TYPE = 4;
>> input.NITERS = 500;
>> input.LVARS = [-1, -1];
>> output = OptionsMatlab(input)
```

```
output =


      VARS: [2x1 double]
    OBJFUN: 0
    OBJTRC: [1x1 struct]
```

```
>> optimisationTerrain(output,input,3)
```

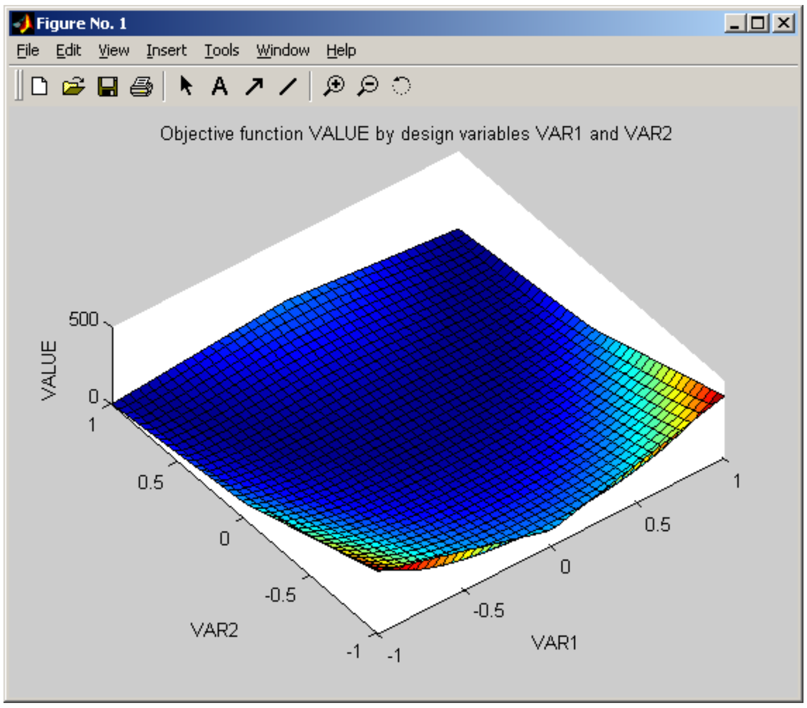


Figure 4 The objective function surface of the Banana problem

Functions

`bananafun`	objective function
`bananafun_parallel`	parallel version of the objective function
`bananafun_parallel_parse`	parallel version of the objective function
`createbananastruct`	creates an input structure for the banana problem
`createbananastructparallel`	creates an input structure for the parallel invocation of the banana problem

Beam problem

An example of the constrained Beam problem

Example

This example plots the objective function surface of the Beam problem.

```
>> input= createBeamStruct;
>> input.MC_TYPE = 4;
>> input.NITERS = 500;
>> output = OptionsMatlab(input)
```

```
output =


       VARS: [2x1 double]
     OBJFUN: 2.9269e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

```
>> optimisationTerrain(output,input,3)
```

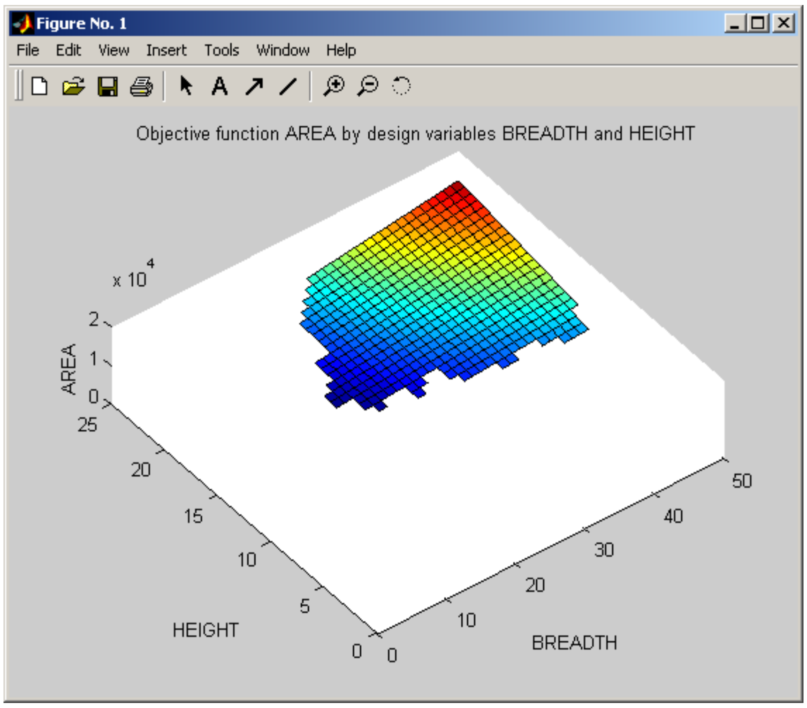


Figure No. 1
File Edit View Insert Tools Window Help
Objective function AREA by design variables BREADTH and HEIGHT
x 10^4
AREA
2
1
0
25
20
15
10
5
0
HEIGHT
0
10
20
30
40
50
BREADTH

Figure 5 The valid objective function surface of the Beam problem

Functions

`beamcon_parallel`	parallel version of the constraint function
`beamcon_parallel_parse`	parallel version of the constraint function
`beamcon_parallel2`	parallel version of the constraint function
`beamcon_parallel2_parse2`	parallel version of the constraint function
`beamfun_parallel`	parallel version of the objective function
`beamfun_parallel_parse`	parallel version of the objective function
`beamfun_parallel2`	parallel version of the objective function
`beamfun_parallel2_parse`	parallel version of the objective function
`beamobjcon`	constraint function
`beamobjfun`	objective function
`createBeamStruct`	creates an input structure for the beam problem
`createBeamStructParallel`	creates an input structure for the parallel invocation of the beam problem
`createBeamStructParallel2`	creates an input structure for the parallel invocation of the beam problem using `optjobparallel2`
`createBeamStructRSM`	creates an input structure for the generation of a RSM for the beam problem

Bump problem

An example of the combined objective and constraint function of the Bump problem.

Example

The Bump problem may be extended into multiple dimensions. This example plots the objective function surface of the Bump problem in two dimensions.

```
>> input= createbumpstruct(2.8, 2);
>> input.MC_TYPE = 4;
>> input.NITERS = 500;
>> output = OptionsMatlab(input)
```

```
output =


       VARS: [2x1 double]
     OBJFUN: 0.2021
       CONS: [2x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

```
>> optimisationTerrain(output,input,3)
```

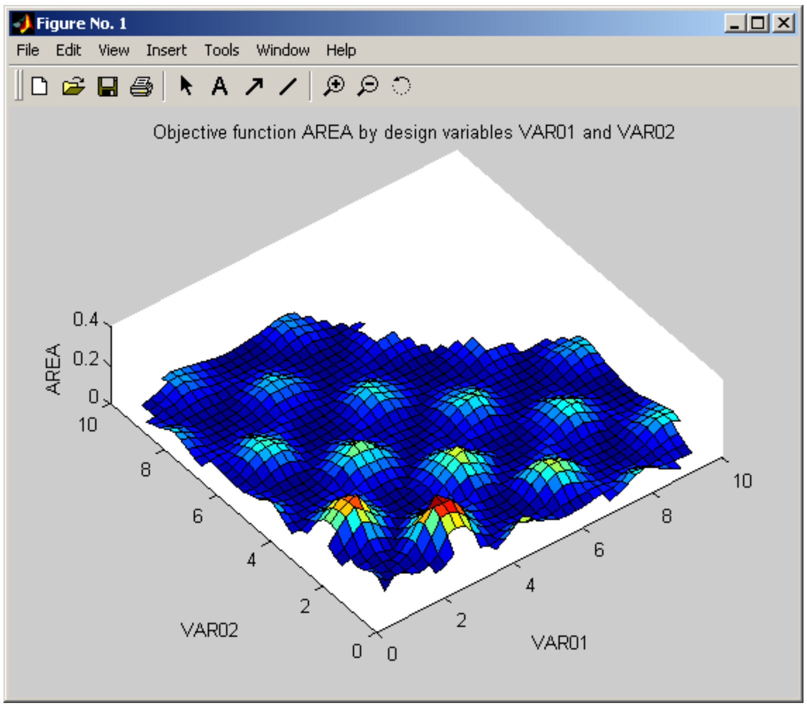


Figure 6 The valid objective function surface of the Bump problem in two dimensions

Functions

`bumpfuncombined`	combined objective and constraint function
`bumpfuncombined_parallel`	parallel version of the combined objective and constraint function
`bumpfuncombined_parallel_parse`	parallel version of the combined objective and constraint function
`createbumpstruct`	creates an input structure for the bump problem
`createbumpstructparallel`	creates an input structure for the parallel invocation of the bump problem

optimisationAppendDataPoints

Append data points to an output structure

This function appends data points to an OptionsMatlab output structure from a second OptionsMatlab output structure. The function can either copy all of the points, specified points, or the best point returned by the optimiser, from the second output structure.

The edited structure is returned as an output argument.

Syntax

```
      STRUCTOUTEDIT = optimisationAppendDataPoints(STRUCTOUT,
STRUCTOUT2)
      STRUCTOUTEDIT = optimisationAppendDataPoints(STRUCTOUT,
STRUCTOUT2, POINTS)
```

Description

`STRUCTOUTEDIT = optimisationAppendDataPoints(STRUCTOUT, STRUCTOUT2)` where `STRUCTOUT` is the results structure returned by `OptionsMatlab` containing data points in a field `OBJTRC` (and `CONSTRC`) or `RSMTRC` to which the data points are appended. `STRUCTOUT2` is a results structure from which *all* data points are copied. `STRUCTOUTEDIT` is a copy of STRUCTOUT to which all of the points are copied.

Note that RSM results can only be copied from to a structure containing RSM results (`RSMTRC`). Also unconstrained data points cannot be copied to a structure containing constrained data.

`STRUCTOUTEDIT = optimisationAppendDataPoints(STRUCTOUT, STRUCTOUT2, POINTS)` as above where POINTS is a string specifying the mode of operation, or a vector specifying the points to be copied. POINTS may be a string with the following values:

`all`	all points from `STRUCTOUT2` will be appended to `STRUCTOUT`. This is the default operation if `POINTS` is empty (`[]`).
`best`	the best point returned specified by `STRUCTOUT2` will be appended to `STRUCTOUT`.

If `POINTS` is a vector it must contain the indices of points in `STRUCTOUT2.OBJTRC.OBJFUN` (or `STRUCTOUT2.RSMTRC.OBJFUN`) to be appended to `STRUCTOUT`.

Examples

These examples will demonstrate the three modes of operation of `optimisationAppendDataPoints`:

```
structin = createBeamStruct(2.8);
structin.NITERS = 10;                %Do a DoE of ten points
structout = OptionsMatlab(structin);

structin2 = structin;
structin2.OMETHD = 4;                %Do a GA of ten points
structout2 = OptionsMatlab(structin2);
```

In this example all data points from the second output structure will be appended to the first.

```
structoutedit = optimisationAppendDataPoints(structout,
structout2);
structoutedit.OBJTRC
structoutedit.CONSTRC
```

```
ans =
    OBJFUN: [1x20 double]
      VARS: [2x20 double]
    NCALLS: 20


ans =
      CONS: [5x20 double]
      VARS: [2x20 double]
     LCONS: [5x20 double]
     UCONS: [5x20 double]
    NCALLS: 20
```

In this example the best data point from the second output structure will be appended to the first.

```
structoutedit2 = optimisationAppendDataPoints(structout,
structout2, 'best');
structoutedit2.OBJTRC
structoutedit2.CONSTRC
```

```
ans =
    OBJFUN: [1x11 double]
      VARS: [2x11 double]
    NCALLS: 11


ans =
      CONS: [5x11 double]
      VARS: [2x11 double]
     LCONS: [5x11 double]
     UCONS: [5x11 double]
    NCALLS: 11
```

In this example the first, fifth and tenth data points from the second output structure will be appended to the first.

```
structoutedit3 = optimisationAppendDataPoints(structout,
structout2, [1,5,10]);
structoutedit3.OBJTRC
structoutedit3.CONSTRC
```

```
ans =
    OBJFUN: [1x13 double]
      VARS: [2x13 double]
    NCALLS: 13


ans =
      CONS: [5x13 double]
      VARS: [2x13 double]
     LCONS: [5x13 double]
     UCONS: [5x13 double]
    NCALLS: 13
```

See also

`optimisationCropDataPoints`, `optimisationReplaceDataPoints`

optimisationCropDataPoints

Crops data points from an output structure

This function crops data points from an `OptionsMatlab` output structure. The output structure can contain data points in a field `OBJTRC` (and `CONSTRC`) or `RSMTRC`.

The points to be cropped are specified by a vector of indices for points in the vector of objective function evaluations. The edited structure is returned as an output argument.

Syntax

`STRUCTOUTEDIT = optimisationCropDataPoints(STRUCTOUT, POINTS)`

Description

`STRUCTOUTEDIT = optimisationCropDataPoints(STRUCTOUT, POINTS)` where `STRUCTOUT` is the results structure returned by `OptionsMatlab` containing data points in a field `OBJTRC` (and `CONSTRC`) or `RSMTRC`. `POINTS` is a vector of indices to data points in `STRUCTOUT.OBJTRC.OBJFUN` (or `STRUCTOUT.RSMTRC.OBJFUN`).

`STRUCTOUTEDIT` is a copy of `STRUCTOUT` with the specified points cropped.

Example

In this example the first, fifth and tenth points are cropped from an output structure containing ten points:

```
structin = createBeamStruct(2.8);
structin.NITERS = 10;               %Do a DoE of ten points
structout = OptionsMatlab(structin);

structoutedit = optimisationCropDataPoints(structout,
[1,5,10]);
structoutedit.OBJTRC
structoutedit.CONSTRC
```

```
ans =
    OBJFUN: [4.1998e+003 2.5211e+003 2.3857e+003 3.2492e+003
7.9283e+003 708.6411 1.7318e+003]
      VARS: [2x7 double]
    NCALLS: 7

ans =
      CONS: [5x7 double]
      VARS: [2x7 double]
     LCONS: [5x7 double]
     UCONS: [5x7 double]
    NCALLS: 7
```

See also

`optimisationAppendDataPoints`, `optimisationReplaceDataPoints`

optimisationReplaceDataPoints

Replace data points based upon strategy

This function will replace data points from an OptionsMatlab output structure with data points from a second structure selected depending upon the specified strategy. The attribute used to select the data points may the value of the objective function, or the normalized Euclidian distance from the best point specified in STRUCTOUT.

Syntax

```
      STRUCTOUTEDIT = optimisationAppendDataPoints(STRUCTOUT,
STRUCTOUT2,STRATEGY,STRUCTIN)
      STRUCTOUTEDIT = optimisationAppendDataPoints(STRUCTOUT,
STRUCTOUT2,STRATEGY,STRUCTIN,NUMPOINTS)
```

Description

`STRUCTOUTEDIT = optimisationAppendDataPoints(STRUCTOUT, STRUCTOUT2, STRATEGY, STRUCTIN)` where `STRUCTOUT` and `STRUCTOUT2` are results structure returned by `OptionsMatlab` containing data points in a field `OBJTRC` (and `CONSTRC`) or `RSMTRC`. `STRATEGY` is a string that specifies the strategy used to replace data points:

`select_best`	selects data points from `STRUCTOUT` and `STRUCTOUT2` depending upon the value of the objective function
`select_closest`	selects data points from `STRUCTOUT` and `STRUCTOUT2` depending upon the normalized Euclidian distance from `STRUCTOUT.VARS`
`replace_worst`	replaces the *worst* `NUMPOINTS` data points from `STRUCTOUT` with the *best* points from `STRUCTOUT2` depending upon the value of the objective function
`replace_furthest`	replaces the *furthest* `NUMPOINTS` data points from `STRUCTOUT` with the *closest* points from `STRUCTOUT2` depending upon the normalized Euclidian distance from `STRUCTOUT.VARS`

STRUCTIN is the OptionsMatlab input structure for the problem that was used to generate the data points. STRUCTOUTEDIT is the edited copy of STRUCTOUT.

Note that RSM results can only be copied from to a structure containing RSM results (RSMTRC). Also unconstrained data points cannot be copied to a structure containing constrained data.

Note that duplicate points are not detected and may occur in STRUCTOUTEDIT.

STRUCTOUTEDIT = optimisationAppendDataPoints(STRUCTOUT, STRUCTOUT2,STRATEGY,STRUCTIN,NUMPOINTS) where NUMPOINTS is an integer value that has alternative meanings depending upon the strategy. Where STRATEGY:

select_*	NUMPOINTS is the number of data points in STRUCTOUTEDIT. If NUMPOINTS is not specified, or is empty ([]), it will default to STRUCTOUT.*TRC.NCALLS
replace_*	NUMPOINTS is the number of data points in STRUCTOUT replaced with points from STRUCTOUT2

Examples

These examples will demonstrate the alternative strategies to replace data points.

```
structin = createBeamStruct(2.8);
structin.NITERS = 10;              %Do a DoE of ten points
structout = OptionsMatlab(structin)

structin2 = structin;
structin2.OMETHD = 4;              %Do a GA of ten points
structout2 = OptionsMatlab(structin2)
```

In this example the *best* data points are selected from between structout and structout2.

```
structoutedit = optimisationReplaceDataPoints(structout, ...
                    structout2,'select_best',structin)
structoutedit.OBJTRC.OBJFUN
structoutedit.OBJTRC.VARS
```

In this example 15 data points are selected from between `structout` and `structout2` depending upon their normalized Euclidian distance from `structout.VARS`.

```
structoutedit = optimisationReplaceDataPoints(structout, ...
                    structout2,'select_closest',structin,15)
structoutedit.OBJTRC.OBJFUN
structoutedit.OBJTRC.VARS
```

In this example the 5 *worst* data points from `structout` are replaced by the 5 *best* data points from `structout2`.

```
structoutedit = optimisationReplaceDataPoints(structout, ...
                    structout2,'replace_worst',structin,5)
structoutedit.OBJTRC.OBJFUN
structoutedit.OBJTRC.VARS
```

In this example the 5 data points from `structout` that are *furthest* from `structout.VARS` are replaced by the 5 *closest* points from `structout2`.

```
structoutedit = optimisationReplaceDataPoints(structout, ...
                    structout2,'replace_furthest',structin,5)
structoutedit.OBJTRC.OBJFUN
structoutedit.OBJTRC.VARS
```

See also
`optimisationCropDataPoints`, `optimisationAppendDataPoints`

optimisationDigest

Prints the results of an optimisation and returns validity of optimum

Syntax

```
ISVALID = optimisationDigest(STRUCTOUT,STRUCTIN)
ISVALID = optimisationDigest(STRUCTOUT,STRUCTIN,FILENAME)
ISVALID = optimisationDigest(STRUCTOUT,STRUCTIN,[])
```

Description

`ISVALID = optimisationDigest(STRUCTOUT, STRUCTIN)` Prints digest to standard output, where `STRUCTOUT` is the output, and `STRUCTIN` the input, from `OptionsMatlab`. `ISVALID` is 1 where the optimum point returned by the optimisation does not violate the constraints or the design variable limits, otherwise `ISVALID` is 0.

`ISVALID = optimisationDigest(STRUCTOUT,STRUCTIN,FILENAME)` Prints direct to `FILENAME`

`ISVALID = optimisationDigest(STRUCTOUT,STRUCTIN,[])` Suppresses digest output

Example

The output of `optimisationDigest` is illustrated by the following example:

```
>> input = createBeamStruct
>> results = OptionsMatlab(input)
>> isvalid = optimisationDigest(results,input)
```

```
============================================================

 Optimisation of the problem defined by "beamobjfun" and
"beamobjcon"
 Optimisation method: 2.8

 Status after 500 evaluations is:-

 Trial vector
 Lwr Bound       Vector       Uppr Bound    Variable (units)

   5.00000000 <  19.96577454 >  50.00000000  BREADTH
   2.00000000 <  14.75254536 >  25.00000000  HEIGHT

 No of V. Boundary Violations =   0

 Objective Function (min.)    =  2945.4599   AREA

 Constraints vector
  Lwr Bound       Vector       Uppr Bound    Variable
(units)

              < 103.56009357 > 200.00000000  SIGMA-B
              <   2.54629163 > 100.00000000  TAU
              <   4.86091675 >   5.00000000  DEFLN
              <   7.38891713 >  10.00000000  H-ON-B
 5000.00000000 < 184550.01793812             F-CRIT

 No of Constraint Violations =   0
============================================================

isvalid =

     1
```

See also

OptionsMatlab

optimisationHistory

Plots a trace of the optimisation search history

`optimisationHistory` plots a trace of the objective function over the search history. `optimisationHistory` provides a convenient way to view the search history over a number of searches by combining this information in a single plot.

Syntax

```
optimisationHistory(RESULTS)
optimisationHistory(RESULTS,LABELS)
optimisationHistory(RESULTS,LABELS,WITHMARKERS)
optimisationHistory(RESULTS,LABELS,WITHMARKERS,ISLOG)
```

Description

`optimisationHistory(RESULTS)` Where `RESULTS` is a cell array containing all of the search results to be plotted. The elements of this array may be either `OptionsMatlab` output structures or vectors containing objective function values.

`optimisationHistory(RESULTS,LABELS)` Where `LABELS` is a cell array of strings containing the labels for a legend which annotates each of the searches plotted. `LABELS` must be the same length as `RESULTS`, otherwise `LABELS` may be empty if no legend is required.

`optimisationHistory(RESULTS,LABELS,WITHMARKERS)` Where `WITHMARKERS` specifies whether markers are to be used on the plot. If `WITHMARKERS` equals 0 markers will not be used, otherwise markers are generated automatically (default).

`optimisationHistory(RESULTS,LABELS,WITHMARKERS,ISLOG)` Where `ISLOG` specifies whether the scale of the Y-axis is logarithmic. If `WITHMARKERS` equals 0 a linear scale will be used (default), otherwise a logarithmic scale will be used for the Y-axis.

Example

The following example illustrates the use of `optimisationHistory`:

```
>> input = createBeamStruct;
>> input.OMETHD = 1.6;
>> resultscell{1} = OptionsMatlab(input);
>> resultscell{2} = rand(200,1)*3000+1000;
>> labels = {'Optivar SEEK','Random values'}
>> optimisationHistory(resultscell, labels)
```

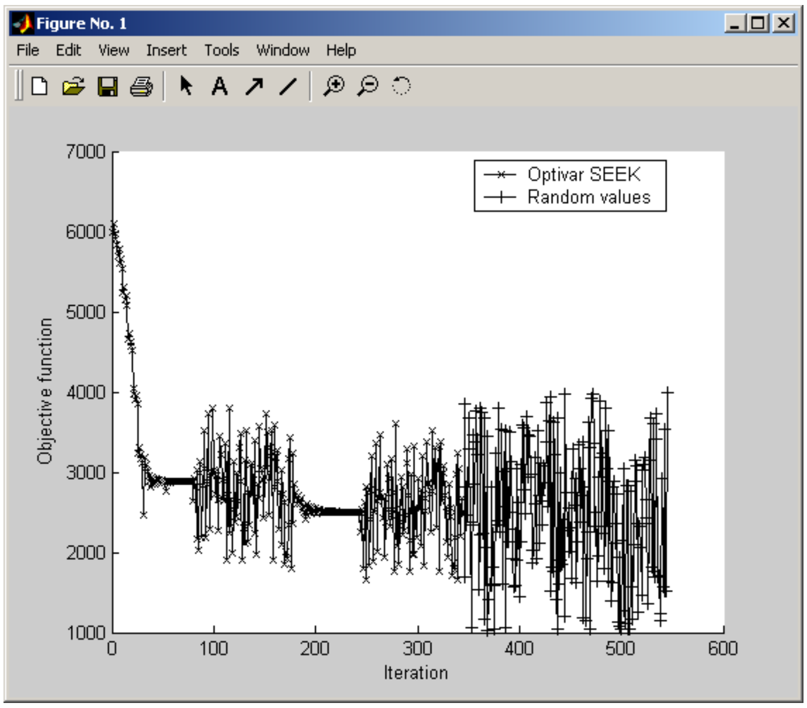


Figure 7 The plot produced by `optimisationHistory`

See also

`OptionsMatlab`, `plotOptionsSurfaces`

optimisationSampleRSM

Builds and samples a Response Surface Model.

This function will generate an array of candidate points and then invokes OptionsMatlab to build a Response Surface Model (RSM) and samples the candidate points. The structure returned by `optimisationSampleRSM` can then be plotted.

Syntax

```
      STRUCTOUT = optimisationSampleRSM(STRUCTIN,RESULTS,
NUMPOINTS)
      STRUCTOUT = optimisationSampleRSM(...,LVARS,UVARS)
      [STRUCTOUT, VECTORS] = optimisationSampleRSM(...)
```

Description

`STRUCTOUT = optimisationSampleRSM(STRUCTIN,RESULTS, NUMPOINTS)` where `STRUCTIN` is an `OptionsMatlab` input structure which specifies the RSM, and `RESULTS` is an output structure containing the results over which the RSM is built. If `NUMPOINTS` is a scalar value, this will specify the total number of sample points which will be distributed evenly across `NVRS` dimensions. Otherwise `NUMPOINTS` must be a vector of length `NVRS` which specifies the number of sample points in each dimension (the total number of sample points will equal `PROD(NUMPOINTS)`). The return argument `STRUCTOUT` will contain the output structure returned by `OptionsMatlab`.

To hold a design variable constant set the corresponding element of `NUMPOINTS` equal to zero. All design variables for which `NUMPOINTS` is zero will be sampled at the value specified by `STRUCTIN.VARS`.

`STRUCTOUT = optimisationSampleRSM(...,LVARS,UVARS)` as above where `LVARS` and `UVARS` are vectors that specify the upper and lower limits between which the design variables are sampled. If these vectors are not specified the values of `LVARS` and `UVARS` defined in `STRUCTIN` are used.

`[STRUCTOUT, VECTORS] = optimisationSampleRSM(...)` as above where `VECTORS` is a cell array containing `NVRS` vectors of the points at which the each of the design variables were sampled.

Examples

The first example will sample response surface models built over the beam problem.

```
%Run a DOE in OptionsMatlab
input1 = createBeamStruct;
input1.NITERS = 50;
output1 = OptionsMatlab(input1);

%Create an input structure to search a RSM
input2 = createBeamStruct;
input2.OBJMOD = 3.3;
input2.CONMOD = 3.3;

%Sample 100 evenly spaced points
output2 = optimisationSampleRSM(input2, output1, 100)
```

```
output2 =


      VARS: [2x1 double]
    OBJFUN: 2.3606e+003
      CONS: [5x1 double]
    RSMTRC: [1x1 struct]
```

```
%Plot an interpolated surface over the sampled points
fig = optimisationTerrain(output2, input2);
%Plot the original points
optimisationTrace(output1, input1, 1, fig);
```

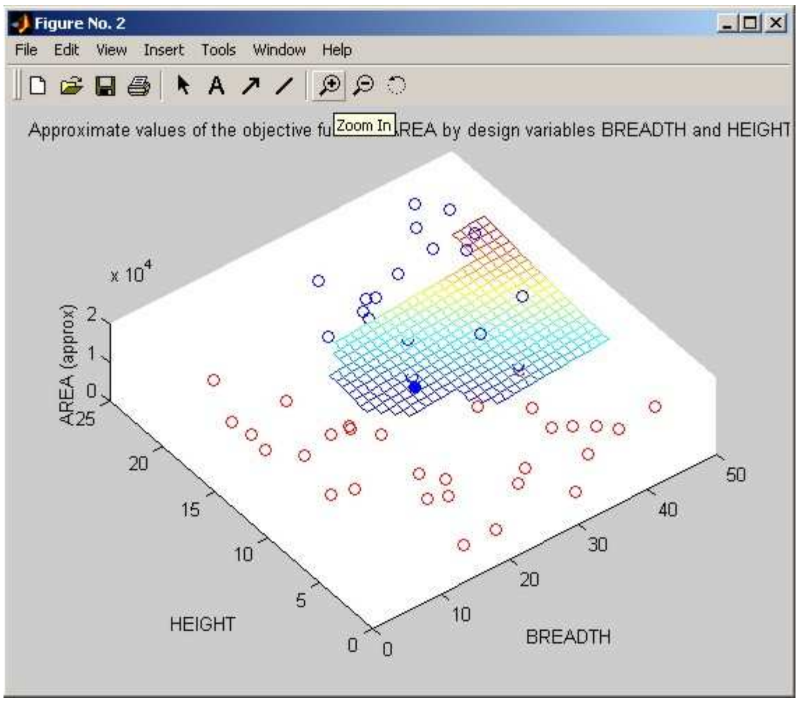


Figure No. 2
File Edit View Insert Tools Window Help
Zoom In
Approximate values of the objective function AREA by design variables BREADTH and HEIGHT
x 10^4
AREA (approx)
2
1
0
25
20
15
10
5
0
HEIGHT
0
10
20
30
40
50
BREADTH

Figure 8 First plot of the output of `optimisationSampleRSM`

```
%Sample 5 points in the first dimension and 20 points in the
%second dimension
output3 = optimisationSampleRSM(input2, output1, [5, 20])
optimisationTerrain(output3, input2);
```

```
output3 = 


      VARS: [2x1 double]
    OBJFUN: 2.3606e+003
      CONS: [5x1 double]
    RSMTRC: [1x1 struct]
```

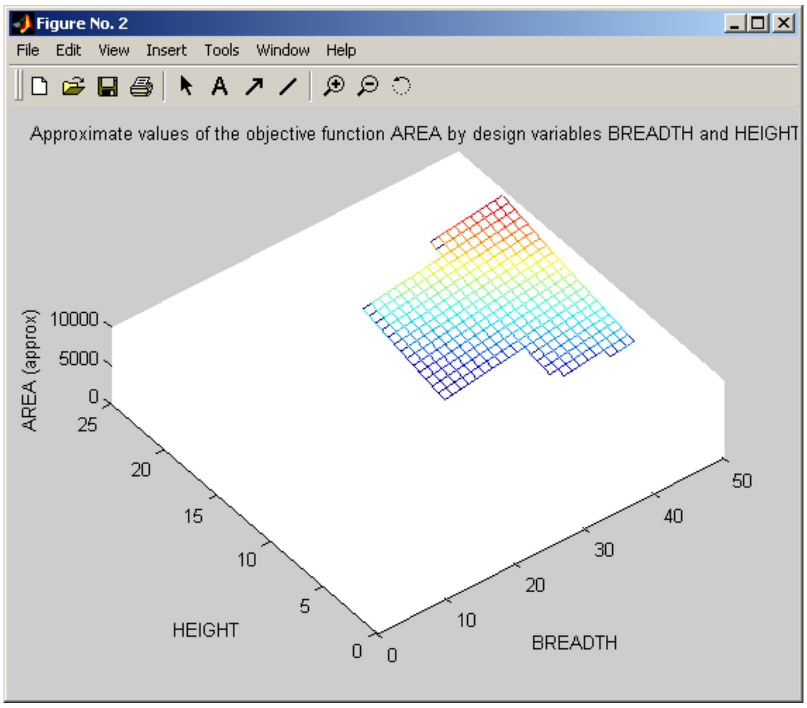


Figure 9 Second plot of the output of `optimisationSampleRSM`

The second example samples the bump problem over two dimensions of a five dimensional problem. The values of the design variables which are held constant are specified by `input5.VARS`.

```
%Run a DOE over the bump function in 5 dimensions
input4 = createbumpstruct(2.8, 5);
input4.NITERS = 50;
output4 = OptionsMatlab(input4);

%Build a RSM over the DOE and sample in the second and third
%dimensions
input5 = createbumpstruct(2.8, 5);
input5.OBJMOD = 3.3;
input5.CONMOD = 3.3;
output5 = optimisationSampleRSM(input5,output4,[0,20,20,0,0]);

%Plot the sampled points in the second and third dimensions
optimisationTerrain(output5, input5, 1, [], [-37.5,30], [2,3]);
```

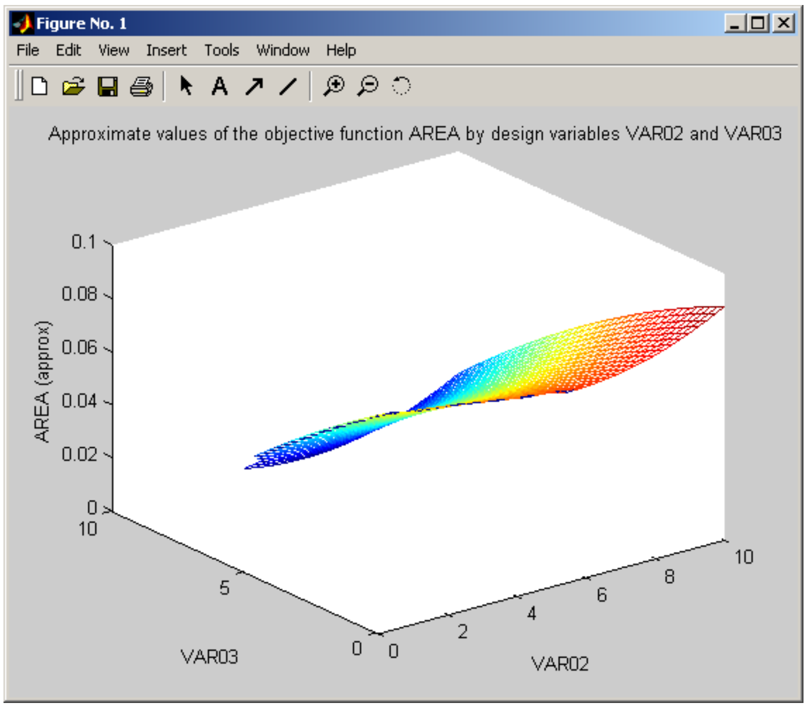


Figure 10 Third plot of the output of `optimisationSampleRSM`

See also

`optimisationTerrain`, `OptionsMatlab`

optimisationSearchTrace

Search trace history for values at optimum `VARS`

This function searches the optimisation trace history(ies) in `OBJTRC` (and `CONSTRC`) or `RSMTRC` fields of an `OptionsMatlab` output structure for the values of the objective and constraint functions at the optimum vector in the `VARS` field. The function will only operate on structures for which the values of `OBJFUN` and/or `CONS` are zero. This function is intended for use when a search has been performed at `OLEVEL<2` and the values at the optimum point have not been returned.

The edited structure is returned as an output argument.

Syntax

```
      STRUCTOUTEDIT = optimisationSearchTrace(STRUCTOUT)
```

Description

`STRUCTOUTEDIT = optimisationSearchTrace(STRUCTOUT)` where `STRUCTOUT` is the results structure returned by `OptionsMatlab` containing data points in a field `OBJTRC` (and `CONSTRC`) or `RSMTRC`.

Example

This example demonstrates how the values of the objectives and constraints are retrieved from the trace history when the search has been performed at `OLEVEL=0`:

```
      structin = createBeamStruct(2.8);
      structin.OLEVEL = 0;            %Validation call not made
      structin.NITERS = 10;           %Do a DoE of ten points
      structout = OptionsMatlab(structin)

      structout.OBJFUN
      structout.CONS
```

```
structout =


       VARS: [2x1 double]
     OBJFUN: 0
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]



ans =


     0



ans =


     0
     0
     0
     0
     0
```

This searches the problem at `OLEVEL=0` and consequently the values of the objective and constraints at `structout.VARS` are returned as zeros in `structout.OBJFUN` and `structout.CONS`. The values of the objective and constraints at the optimum point are assigned to these variables by searching the trace history:

```
structout2 = optimisationSearchTrace(structout)
structout2.OBJFUN
structout2.CONS
```

```
structout2 =


       VARS: [2x1 double]
     OBJFUN: 5.0853e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]



ans =


  5.0853e+003



ans =


  1.0e+005 *


    0.0004
    0.0000
    0.0000
    0.0001
    5.4913
```

See also

OptionsMatlab

optimisationTerrain

Mesh, surface & contour plots of optimisation results

This function plots surfaces produced by interpolation between the points at which the objective function was evaluated. The optimisation terrain may be represented as a mesh, surface or contour plot. The points which do not meet the optimisation constraints will be cropped from the surface.

Syntax

```
optimisationTerrain(STRUCTOUT,STRUCTIN)
optimisationTerrain(STRUCTOUT,STRUCTIN,PLOTTYPE)
optimisationTerrain(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG)
optimisationTerrain(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,VIEW)
optimisationTerrain(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,VIEW,DIMS)
FIG = optimisationTerrain(...)
```

Description

`optimisationTerrain(STRUCTOUT,STRUCTIN)` where `STRUCTOUT` is the results structure returned by `OptionsMatlab` and `STRUCTIN` is the `OptionsMatlab` input structure.

`optimisationTerrain(STRUCTOUT,STRUCTIN,PLOTTYPE)` as above where `PLOTTYPE` is a scalar which indicates the type of plot. The valid values of `PLOTTYPE` are:

1 = Mesh of valid points [default]
2 = Mesh of valid points in a single colour
3 = Surface of valid points
4 = 3D contour plot of valid points
5 = 3D contour plot of valid points with a mesh
6 = Mesh of all points
7 = Mesh of all points in a single colour
8 = Surface of all points
9 = 3d contour plot of all points
10 = 3d contour plot of all points with a mesh

`optimisationTerrain(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG)` as above where `FIG` is the figure in which to plot the optimisation terrain. If `FIG` is not provide a new figure will be generated. FIG can also be empty [].

`optimisationTerrain(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,VIEW)` as above where `VIEW` is a two element vector that sets the view of the 3D plot. For example `VIEW` = [0 90] for overhead plots. The default view is [-37.5, 30]. VIEW can also be empty [].

`optimisationTerrain(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,VIEW, DIMS)` as above where `DIMS` is a two element vector specifying the two design variables to be plotted. By default the first two design variables are plotted.

`FIG = optimisationTerrain(...)` as above where `FIG` is a the number of figure in which the terrain was plotted.

Example

```
input = createBeamStruct;
results = OptionsMatlab(input)
optimisationTerrain(results, input)
```

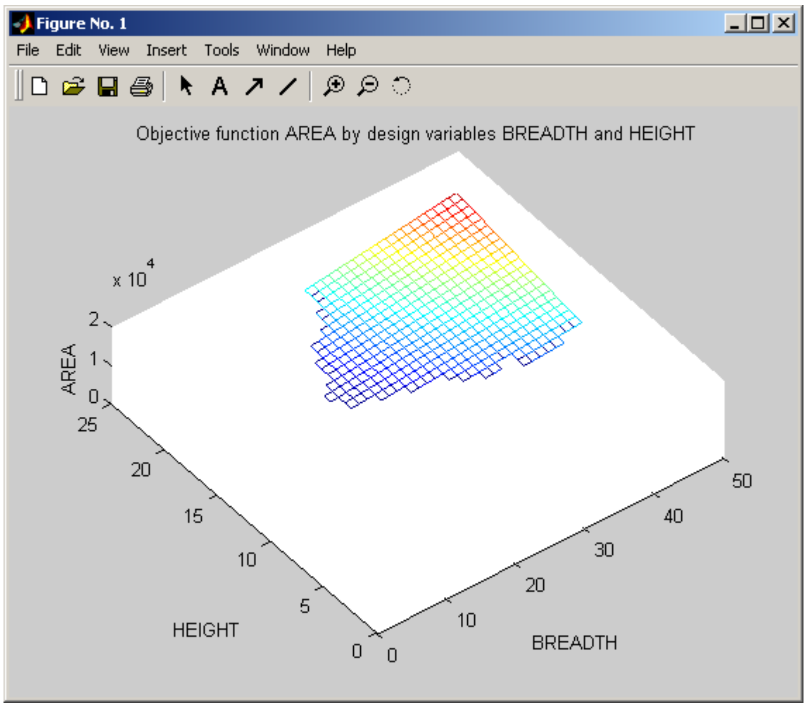


Figure No. 1
File Edit View Insert Tools Window Help
Objective function AREA by design variables BREADTH and HEIGHT
x 10^4
2
1
0
AREA
25
20
15
10
5
0
HEIGHT
0
10
20
30
40
50
BREADTH

Figure 11 Plot produced by `optimisationTerrain`

See also

`view`, `mesh`, `griddata`

optimisationTestSuite

A script which demonstrates the functionality of `OptionsMatlab` with the Beam problem.

See also

`beamcon_parallel`, `beamcon_parallel_parse`, `beamfun_parallel`, `beamfun_parallel_parse`, `beamobjcon`, `beamobjfun`, `createBeamStruct`, `createBeamStructParallel`, `createBeamStructRSM`

optimisationTestSuiteComb

A script which demonstrates the functionality of `OptionsMatlab` with the combined objective and constraint function of the Bump problem.

See also

`bumpfuncombined`, `bumpfuncombined_parallel`, `bumpfuncombined_parallel_parse`, `createbumpstruct`, `createbumpstructparallel`

optimisationTestSuiteUncon

A script which demonstrates the functionality of `OptionsMatlab` with unconstrained Banana problem based upon Rosenbrock's function.

See also

`bananafun`, `bananafun_parallel`, `bananafun_parallel_parse`, `createbananastruct`, `createbananastructparallel`

optimisationTestSuiteSPM

A script which demonstrates the stochastic process model functionality of `OptionsMatlab` with constrained or unconstrained problems.

This function is intended as an extension to the test suites and performs additional tests that search and sample stochastic process model RSMs using quick tuned hyper-parameters. These tests can be invoked on constrained and unconstrained design problems and demonstrate how to build and investigate stochastic process models.

Syntax

```
      optimisationTestSuiteSPM(STRUCTIN, STRUCTOUT_DOE,
STRUCTOUT_HP)
      optimisationTestSuiteSPM(..., PLOTTYPE)
```

Description

`optimisationTestSuiteSPM(STRUCTIN, STRUCTOUT_DOE, STRUCTOUT_HP)` where `STRUCTIN` is the default input data structure for the design problem, `STRUCTOUT_DOE` is the trace history of a previous design search which contains the information required to generate the RSM and `STRUCTOUT_HP` is the results of tuning the hyperparameters of a stochastic process model over the points in the DoE.

`optimisationTestSuiteSPM(..., PLOTTYPE)` where `PLOTTYPE` is a scalar which indicates the type of plot used in calls to `optimisationTerrain`. The valid values of `PLOTTYPE` are:

0 = No plotting
1 = Mesh of valid points [default]
2 = Mesh of valid points in a single colour
3 = Surface of valid points
4 = 3D contour plot of valid points
5 = 3D contour plot of valid points with a mesh
6 = Mesh of all points
7 = Mesh of all points in a single colour
8 = Surface of all points
9 = 3d contour plot of all points
10 = 3d contour plot of all points with a mesh

Example

This example demonstrates how the SPM tests can be invoked on the Banana problem

```
>> inputStruct = createbananastruct;
```

Perform a 50 point DoE over the problem

```
>> input1 = inputStruct;
>> input1.OLEVEL = 0;
>> input1.OMETHD = 2.8;        %Design of Experiments
>> input1.NITERS = 50;         %Number of iterations
>> input1.MC_TYPE = 4;         %Full factorial DoE
>> output1 = OptionsMatlab(input1);
```

Perform a quick tuning of the hyperparameters of the stochastic process model RSM

```
>> input10 = inputStruct;
>> input10.OLEVEL = 0;
>> input10.OBJMOD = 4.1;       %Stochastic process model
>> input10.CONMOD = 4.1;       %Stochastic process model
>> input10.RSM_QCK_HP = 1;     %Quick hyperparameter tuning
>> output10 = OptionsMatlab(input10, output1);
```

Invoke the stochastic process model test suite on the problem

```
>> optimisationTestSuiteSPM(inputStruct, output1, output10, 5)
```

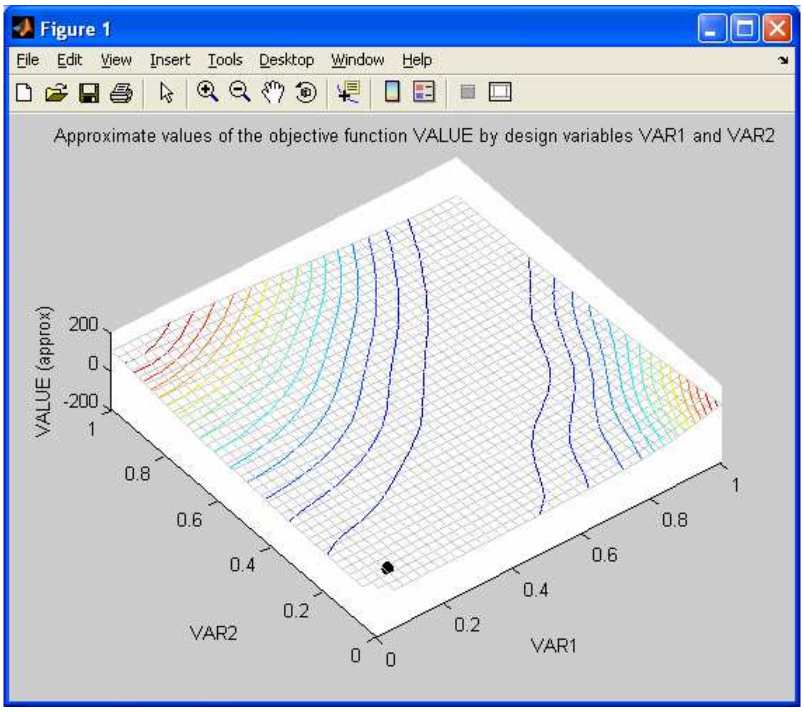


Figure 12 Sampled stochastic process model RSM surface and result of the search for the optimum

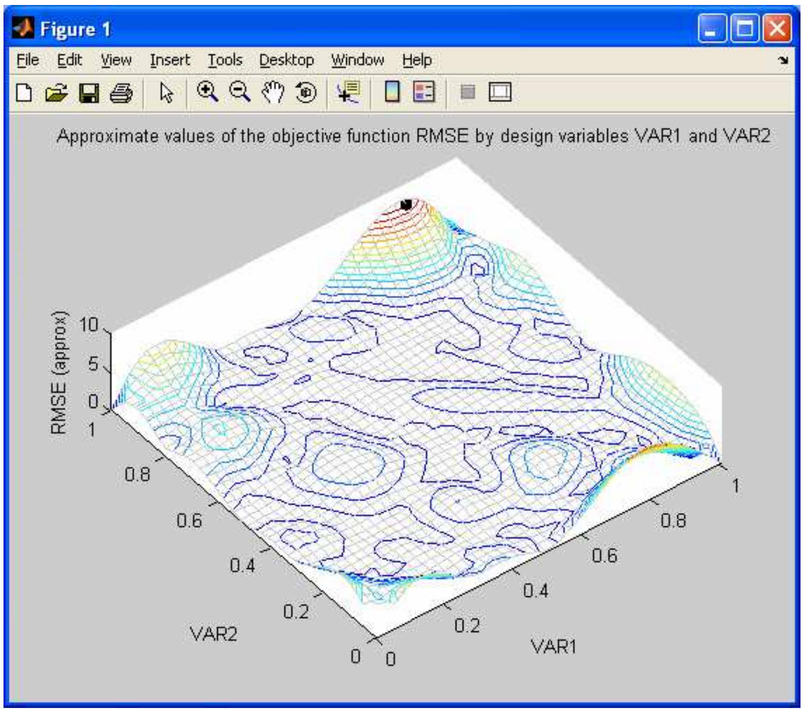


Figure 13 Sampled Root Mean Square Error of the stochastic process model RSM and the result of the search for the maximum in the surface.

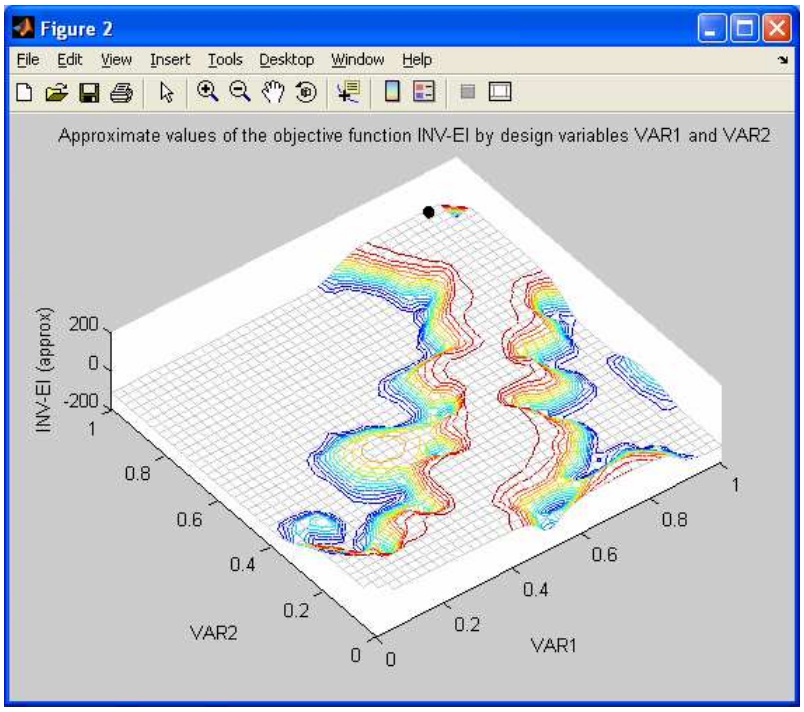


Figure 14 Sampled Expected Improvement of the stochastic process model RSM and the result of the search for the maximum in the surface. NB. When the direction of the underlying search is negative (minimisation) Options automatically inverts the surface to seek the numerical minimum in the EI surface which will be the point of maximum EI in the true problem – the test suite plots the raw minimisation search in figure 1 and plots the inverted surface in figure 2.

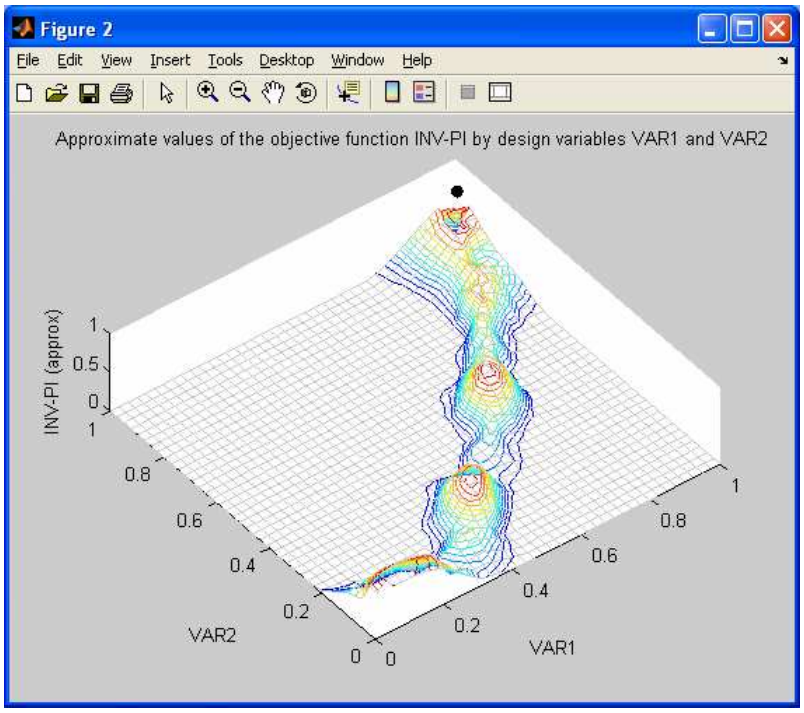


Figure 15 Sampled Probability of Improvement of the stochastic process model RSM and the result of the search for the maximum in the surface. NB. When the direction of the underlying search is negative (minimisation) Options automatically inverts the surface to seek the numerical minimum in the PI

surface which will be the point of maximum PI in the true problem – the test suite plots the raw minimisation search in figure 1 and plots the inverted surface in figure 2.

See also

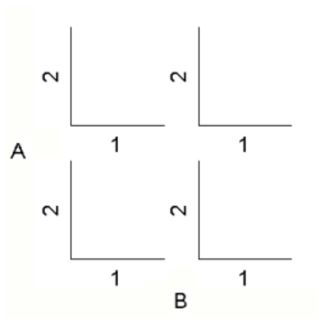
`optimisationTestSuite`, `optimisationTestSuiteComb`, `optimisationTestSuiteUncon`

optimisationTilePlot

Tile plot of four dimensions of a problem

This plots the behaviour of the objective function over four dimensions of a problem. The first two of the design variables (A and B) are plotted across rows and columns of tiles. The third and fourth design variables (1 and 2) will be plotted across the x and y axes of each tile.

Each design variable will be sampled at the specified number of points between the limits defined within the fields `LVARS` and `UVARS` of `STRUCTIN`. For example a problem in which the variables A and B are each sampled at two points the resulting tile plot will have four tiles.



The value of the objective function is plotted as a surface within each 2D tile. The surface colormap is consistent between the tiles. The tile plot is interactive, and by clicking on a tile it is visible as a 3D plot in a separate figure window.

Syntax

```
      optimisationTilePlot(STRUCTOUT,STRUCTIN,DESIGNVARS,
NUMPOINTS,TILETYPE)
      optimisationTilePlot(...,PLOTPOINTS)
      optimisationTilePlot(...,FIG)
      FIG = optimisationTilePlot(...)
      [FIG,TILESOUT,TILESIN] = optimisationTilePlot(...)
      optimisationTilePlot(TILESOUT,TILESIN)
```

Description

`optimisationTilePlot(STRUCTOUT,STRUCTIN,DESIGNVARS, NUMPOINTS,TILETYPE)` where `STRUCTOUT` is the results structure returned by OptionsMatlab and `STRUCTIN` is the corresponding OptionsMatlab input structure. If the `TILETYPE` is direct search `STRUCTOUT` can be empty `[]`.

`DESIGNVARS` must be a four element vector that defines the design variables to be plotted [A,B,1,2] based upon their index in `STRUCTIN.VARS`. `NUMPOINTS` must also be a four element vector that defines the number of points to be evaluated for each of the `DESIGNVARS`.

`TILETYPE` is an integer that defines how the tile is to be evaluated. The valid values of `TILETYPE` are:

1 = Evaluation of the RSM defined by the fields `OBJMOD` and `CONMOD` of `STRUCTIN`

2 = Direct search of the objective function

`optimisationTilePlot(...,PLOTPOINTS)` as above when `PLOTPOINTS` is a flag that indicates whether to plot the data points. For a RSM if `PLOTPOINTS` = 1 the original data points contained in `STRUCTOUT` will be plotted in each tile, otherwise for a direct search the evaluated points will be plotted. If `PLOTPOINTS` = 0 the points will not be plotted. Default value `PLOTPOINTS` = 0.

`optimisationTilePlot(...,FIG)` as above where `FIG` is the figure in which to plot the tile plot. If `FIG` is not provide a new figure will be generated. `FIG` can also be empty `[]`.

`FIG = optimisationTilePlot(...)` as above where `FIG` is a the number of figure in which the tiles were plotted.

`[FIG,TILESOUT,TILESIN] = optimisationTilePlot(...)` as above where `TILESOUT` and `TILESIN` are cell arrays containing the OptionsMatlab output and input structures that were used to generate the surfaces for each of the tiles.

`optimisationTilePlot(TILESOUT,TILESIN)` replots the tile plot with data returned in the cell arrays `TILESOUT` and `TILEIN`. All other input arguments are optional. The `PLOTPOINTS` argument can be supplied to indicate that the data points should be plotted.

Examples

The following example demonstrates a tile plot of the peaks4d problem using direct search:

```
>> structin = createpeaks4dstruct(2.8);
>> optimisationTilePlot([],structin,[3,4,1,2],[2,3,15,15],2)
```

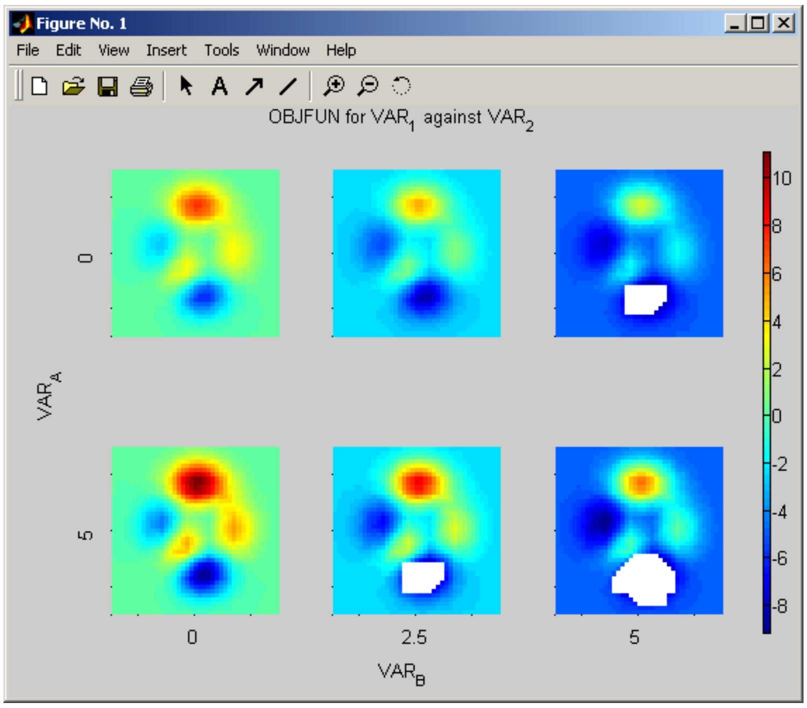


Figure No. 1
File Edit View Insert Tools Window Help
OBJFUN for VAR_1 against VAR_2
0
5
VAR_A
0
2.5
5
VAR_B
10
8
6
4
2
0
-2
-4
-6
-8

Figure 16 Tile plot of the peaks4d problem produced by direct search

By clicking on the tiles of the tile plot with the mouse that tile will be displayed in 3D. For example by clicking on the tile in the top left of the figure the following plot will be displayed:

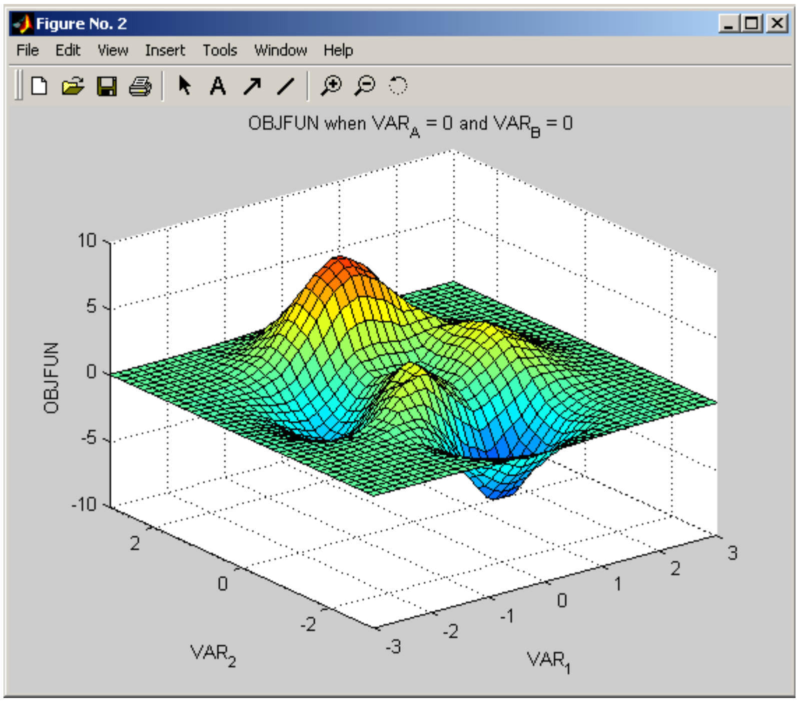


Figure 17 Tiles may be viewed in 3D by clicking on the tile plot

The second example demonstrates a tile plot of the peaks4d problem produced using a Shepard RSM. Using the PLOTPOINTS argument the points of the original data set are also plotted:

```
>> structin = createpeaks4dstruct(2.8);
>> structin.NITERS = 25;
>> structout = OptionsMatlab(structin);

>> structin.OBJMOD = 1;  %Shepard RSM
>> structin.CONMOD = 1;
>> optimisationTilePlot(structout,structin,[3,4,1,2],
[2,3,15,15],1,1)
```

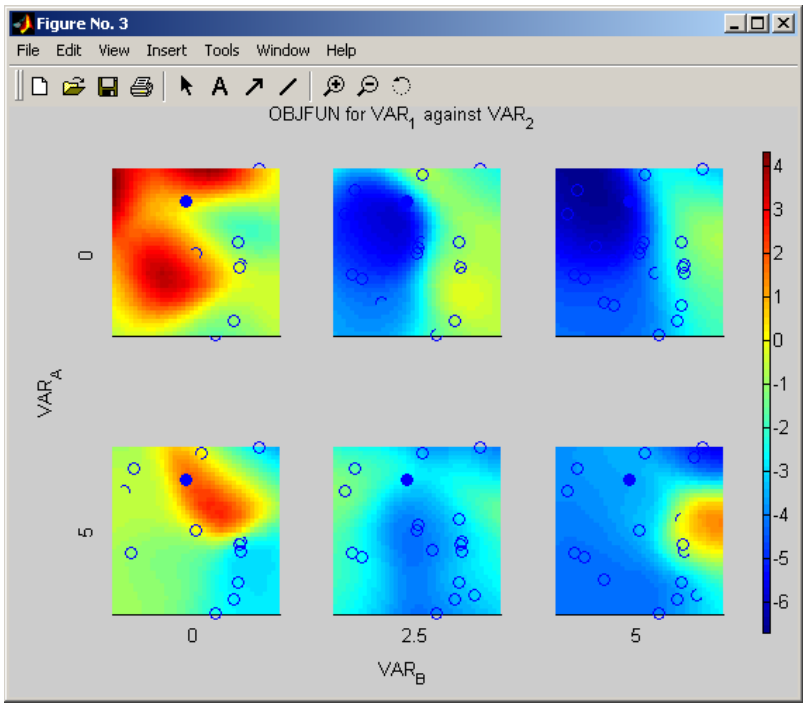


Figure No. 3
File Edit View Insert Tools Window Help
OBJFUN for VAR_1 against VAR_2
VAR_A
0
5
0
2.5
5
VAR_B
4
3
2
1
0
-1
-2
-3
-4
-5
-6

Figure 18 Tile plot of the peaks4d problem produced with a Shepard RSM and the original data set

See also

`optimisationTerrain`, `optimisationTrace`

optimisationTrace

Plots the objective function against two design variables

This function plots points at which the objective function was evaluated. The objective function points may be plotted in colour or black and white. The points may also be joined to represent the sequence of function evaluations.

Syntax

```
      optimisationTrace(STRUCTOUT,STRUCTIN)
      optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE)
      optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG)
      optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,VIEW)
      optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,VIEW,
DIMS)
      optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,
VIEW,DIMS,LABELS)
      FIG = optimisationTrace(...)
```

Description

`optimisationTrace(STRUCTOUT,STRUCTIN)` where `STRUCTOUT` is the results structure returned by `OptionsMatlab` and `STRUCTIN` is the `OptionsMatlab` input structure.

`optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE)` as above where `PLOTTYPE` is a scalar which indicates the type of plot. The valid values of `PLOTTYPE` are:

1 = Coloured point plot [default]
2 = Black and white point plot
3 = Coloured joined point plot
4 = Back and white joined point plot

`optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG)` as above where `FIG` is the figure in which to plot the optimisation terrain. If `FIG` is not provide a new figure will be generated. FIG can also be empty [].

`optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,VIEW)` as above where `VIEW` is a two element vector that sets the view of the 3D plot. For example `VIEW` = [0 90] for overhead plots. The default view is [-37.5, 30]. VIEW can also be empty [].

`optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,VIEW, DIMS)` as above where `DIMS` is a two element vector specifying the two design variables to be plotted. By default the first two design variables are plotted.

`optimisationTrace(STRUCTOUT,STRUCTIN,PLOTTYPE,FIG,VIEW, DIMS,LABELS)` as above where `LABELS` is a flag specifying whether the plot should be labelled. By default labelling is switched off (`LABELS` = 0).

`FIG = optimisationTrace(...)` as above where `FIG` is a the number of figure in which the terrain was plotted.

Example

```
input = createBeamStruct;
results = OptionsMatlab(input)
optimisationTrace(results, input)
```

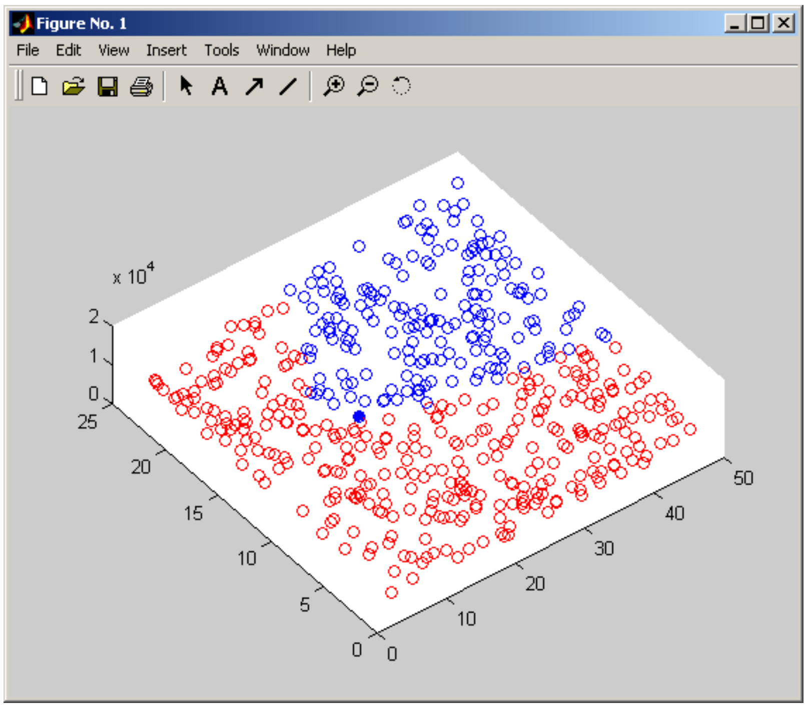


Figure 19 Plot produced by `optimisationTrace`

See also

`view`, `mesh`, `griddata`

OptionsMatlab

Options optimisation and design search package

`OptionsMatlab` makes the Options optimisation and design search package available to Matlab, calling user-defined constraint and objective functions defined as Matlab functions. `OptionsMatlab` also supports a number of Response Surface Model algorithms that allow optimisation to be carried out cheaply using approximations of the values of the objective function and/or constraints.

Syntax

```
STRUCTOUT = OptionsMatlab(STRUCTIN)
STRUCTOUT = OptionsMatlab(STRUCTIN,STRUCTOUT)
```

Description

`STRUCTOUT = OptionsMatlab(STRUCTIN)` where `STRUCTIN` is a Matlab structure containing the problem definition and control parameters for the optimisation algorithms, and `STRUCTOUT` is a structure containing optimum design variables and the objective function and constraint values at this point.

`STRUCTOUT2 = OptionsMatlab(STRUCTIN,STRUCTOUT)` where `STRUCTIN` is a Matlab structure containing the problem definition and control parameters for an optimisation over a Response Surface Model (RSM), and where `STRUCTOUT` is the trace history of a previous design search which contains the information required to generate the RSM. `STRUCTIN` should contain values for either of the parameters `OBJMOD` or `CONMOD` which specify the RSM used, if any, for the objective function and constraints. The design search used to generate data points from which the RSM is produced should ideally be a space-filling search such as a Genetic Algorithm (GA) or Design of Experiments (DoE).

Input argument

The structure `STRUCTIN` must contain a number of mandatory fields, and may also contain a number of optional control parameters. The mandatory fields required are:

`DNULL`: A number that corresponds to a NULL value in the problem setup

`OLEVEL`: [optional] The output level of the `OptionsMatlab` package [0-10]. Default value `OLEVEL` = 1.

`MAXJOBS`: [optional] Allows the user to limit the number of parallel jobs. Default value `MAXJOBS` = 1.

NVRS: The number of design variables

VARS: A vector of NVRS design variables corresponding to the initial design variables to be evaluated

VNAM: A cell array of length NVRS containing the names of the design variables for the internal GENDAT database (variable names must not exceed 10 chars)

LVARS: A vector of length NVRS representing the lower limits to the design variable values.

UVARS: A vector of length NVRS representing the upper limits to the design variable values.

NDVRS: [optional] The maximum number of discrete design variable values for any single design variable. Default value NDVRS = 0, where all design variables are contiguous.

DVARS: [required if NDVRS > 0] A matrix of size NVRS by NDVRS of the discrete design variable values (set to DNULL if contiguous)

NCONS: [optional] The number of design constraints. If NCONS is set to zero the problem will be unconstrained, and OPTCON will not be invoked. Default value NCONS = 0.

CNAM: [required if NCONS > 0] A cell array of length NCONS containing the names of the design constraints for the internal GENDAT database (constraint names must not exceed 10 chars)

LCONS: [required if NCONS > 0] A vector of length NCONS representing the lower limits to the design constraints (set to DNULL if no lower limit)

UCONS: [required if NCONS > 0] A vector of length NCONS representing the upper limits to the design constraints (set to DNULL if no upper limit)

NPARAMS: [optional] The number of user-defined parameters. If NPARAMS are set to zero an empty parameter array will be passed to the user-defined functions. Default value NPARAMS = 0.

PARAMS: [required if NPARAMS > 0] A vector of NPARAMS user-defined parameter values

PNAM: [required if NPARAMS > 0] A cell array of length NPARAMS containing the names of the user-defined parameters for the internal GENDAT database (parameter names must not exceed 10 chars)

ONAM: A char array (max length 10 chars) containing the name of the objective function in the internal GENDAT database.

`OMETHD`: The number of the optimisation or design search algorithm to be used. The available search methods are:

0.0	to just evaluate the user's problem code at the point specified
1.1	for OPTIVAR routine ADRANS
1.2	for OPTIVAR routine DAVID
1.3	for OPTIVAR routine FLETCH
1.4	for OPTIVAR routine JO
1.5	for OPTIVAR routine PDS
1.6	for OPTIVAR routine SEEK
1.7	for OPTIVAR routine SIMPLX
1.8	for OPTIVAR routine APPROX
1.9	for OPTIVAR routine RANDOM
2.1	for user specified routine OPTUM1
2.2	for user specified routine OPTUM2
2.3	for NAG routine E04UCF
2.4	for bit climbing
2.5	for dynamic hill climbing
2.6	for population based incremental learning
2.7	for numerical recipes routines
2.8	for design of experiment based routines
3.11	for Schwefel library Fibonacci search
3.12	for Schwefel library Golden section search
3.13	for Schwefel library Lagrange interval search
3.2	for Schwefel library Hooke and Jeeves search
3.3	for Schwefel library Rosenbrock search
3.41	for Schwefel library DSCG search
3.42	for Schwefel library DSCP search
3.5	for Schwefel library Powell search
3.6	for Schwefel library DFPS search
3.7	for Schwefel library Simplexsearch
3.8	for Schwefel library Complexsearch
3.91	for Schwefel library two-membered evolution strategy
3.92	for Schwefel library multi-membered evolution strategy
4	for genetic algorithm search
5	for simulated annealing
6	for evolutionary programming

7	for evolution strategy

`DIRCTN`: The search direction (in the range +/-2). The optimizers try to minimize the objective function if this argument is -1, maximize it if is +1, minimize the log of the function if it is -2 or maximize the log if it is +2

`NITERS`: The maximum number of iterations to be used

`OPTJOB`: The name of the Matlab function responsible for calling the user-defined objective and constraint functions (maximum length 255 chars)

`OPTFUN`: A string describing the user-defined objective function routine to be called by the `OPTJOB` (maximum length 255 chars)

`OPTCON`: [required if `NCONS` > 0] A string describing the user-defined constraint function routine to be called by the `OPTJOB` (maximum length 255 chars)

`OBJMOD`: [optional] The RSM method to be used to approximate the value of objective function. The available methods are:

1.0	for a Shepard response surface model should
2.1	for linear Radial Basis Function
2.2	for thin plate Radial Basis Function
2.3	for cubic splines Radial Basis Function
2.4	for cubic splines Radial Basis Function with regression via reduced bases
3.1	for mean polynomial regression model
3.2	for first order polynomial regression model
3.3	for first order polynomial regression model plus squares
3.4	for first order polynomial regression model plus products (cross-terms)
3.5	for second order polynomial regression model
3.6	for second order polynomial regression model plus cubes
4.1	for a Stochastic Process Model
4.2	for the root mean square error of the Stochastic Process Model
4.3	for the expected improvement of the Stochastic Process Model
4.31	for the expected improvement of the constrained Stochastic Process Model [requires CONMOD=4.1]
4.32	for the feasibility of improvement of the constrained Stochastic Process Model [requires CONMOD=4.1]
4.33	for the probability of improvement of the Stochastic Process Model

0.0	if the underlying user supplied function is to be called.

CONMOD: [optional] The RSM method to be used to approximate the values of the constraints. The available methods are:

1.0	for a Shepard response surface model should
2.1	for linear Radial Basis Function
2.2	for thin plate Radial Basis Function
2.3	for cubic splines Radial Basis Function
2.4	for cubic splines Radial Basis Function with regression via reduced bases
3.1	for mean polynomial regression model
3.2	for first order polynomial regression model
3.3	for first order polynomial regression model plus squares
3.4	for first order polynomial regression model plus products (cross-terms)
3.5	for second order polynomial regression model
3.6	for second order polynomial regression model plus cubes
4.1	for a Stochastic Process Model
4.2	for the root mean square error of the Stochastic Process Model
4.3	for the expected improvement of the Stochastic Process Model
0.0	if the underlying user supplied function is to be called.

NUMUPDATE: [optional] is a scalar which determines the number of update points to be returned when a search routine is run over a RSM. Update points can be used to improve the accuracy of the dataset that was used to generate the RSM. The update points are return in a sub-structure DOE_TRACE in the output structure. If NUMUPDATE is not specified then update points are not returned by OptionsMatlab.

DOE_TRACE: [optional] is a structure containing the user-supplied DOE points to be used when the control parameter MC_TYPE equals 7. DOE_TRACE requires two mandatory fields:

DOE_TRACE.NCALLS: the number of user-supplied DOE points. Note that DOE_TRACE.NCALLS must equal NITERS-1 as the DOE will first evaluate the design variables VARS.

DOE_TRACE.VARS: the design points to be evaluated during the DOE (size NVARS by DOE_TRACE.NCALLS)

OBJHYPER: [optional] is a structure containing Stochastic Process Model hyper-parameters used to approximate the value of the objective function. OBJHYPER has three recognised fields:

> OBJHYPER.OBJ_LAMBDA: the value of hyper-parameter LAMBDA
>
> OBJHYPER.OBJ_THETA: the values of hyper-parameter THETA (length NVARS; see RSM_QCK_HP)
>
> OBJHYPER.OBJ_EXP: the values of hyper-parameter EXP (length NVARS; see RSM_QCK_HP)

CONHYPER: [optional] is a structure containing Stochastic Process Model hyper-parameters used to approximate the value of the constraints. CONHYPER has three recognised fields:

> CONHYPER.CST_LAMBDA: the value of hyper-parameter LAMBDA
>
> CONHYPER.CST_THETA: the values of hyper-parameter THETA (length NVARS; see RSM_QCK_HP)
>
> CONHYPER.CST_EXP: the values of hyper-parameter EXP (length NVARS; see RSM_QCK_HP)

RSM_QCK_HP: [optional] is a flag that indicates whether quick hyper-parameter tuning should be used when building and searching a Stochastic Process Model RSM. Quick tuning will be used when RSM_QCK_HP is true (e.g. 1). In this condition single values of the hyper-parameters THETA and EXP will be tuned across all design variables, rather than NVARS values of THETA and EXP corresponding to each design variable. This approach is faster but less accurate, and may be appropriate for some problems. If true the values of OBJ_EXP and OBJ_THETA, and of CST_EXP and CST_THETA (in the structures OBJHYPER and CONHYPER) will be scalar, rather than a vector of length NVARS. Quick hyper-parameter tuning is not available when manually tuning the hyper-parameters (i.e. when TUNEHYPER>0).

USERDATA: [optional] is an optional field which can contain any type of Matlab variable. This variable will be passed to the user-defined objective and constraint functions via the OPTJOB function.

TUNEHYPER: [optional] is a flag that indicates whether Stochastic Process Model hyper-parameters should be tuned over the search history contained in the second input argument. Hyper-parameters will be tuned if TUNEHYPER is true (e.g. 1). When TUNEHYPER is called the hyper-parameters are tuned using the search method specified by the input structure. Note that the user's problem is not searched, and the output structure will return the structures

OBJHYPER (and CONHYPER where appropriate) in addition to the objective function OBJ_CLF (and CST_CLF).

It is possible to tune the values of specific hyper-parameters with following values of TUNEHYPER:

0	No tuning
1	Tune THETA, EXP and LAMBDA
2	Tune THETA and EXP
3	Tune THETA and LAMBDA
4	Tune THETA

If a value of TUNEHYPER greater than 1 is specified, and no user-defined hyper-parameters are supplied (via OBJHYPER or CSTHYPER), then initial values for all hyper-parameters will be generated but only the specified hyper-parameters will be tuned with the designated search method.

CHKPT_INTV: [optional] is an integer value that specifies the interval with which the search history is checkpointed to a MAT file. If parallel optimiser is used (OMETHD 2.8 or 4) CHKPT_INTV should be a multiple of MAXJOBS. If CHKPT_INTV equals 0 there will be no checkpointing (default). If OMETHD equals 4 CHKPT_INTV will contain the structure GA_VARS (once available) that will allow the genetic algorithm to be restarted.
CHKPT_FILE: [optional] specifies the file name that the checkpoint file is written (maximum length 20 characters). The default checkpoint file name is 'OptionsCHKPNT.mat'.

OPTUM1: [optional] A string describing the user-defined sequential optimisation routine to be called when OPTUM1 = 2.1 (maximum length 255 chars). The default value 'optum1' corresponds to the example implementation of a random optimiser (see help optum1 for more details).

Other valid STRUCTIN fields correspond to scalar Options control parameters documented in the Options manual (http://www.soton.ac.uk/~ajk/options.ps) sections 8.8 and 8.9. See also FAQ section 5.16. These control parameters include:

BC_NBIN, BC_NRANDM, BC_PENAL, CST_BAD_PT, DHC_INITSZ, DHC_NRANDM, DHC_PENAL, DHC_THRESH, DOE_NRANDM, EP_IMUTNT, EP_NBIN, EP_NPOP, EP_NRANDM, EP_PENAL, EP_TOURN, ES_DELSIG, ES_MDSCRT, ES_NCPOP, ES_NPPOP, ES_NRANDM, ES_PENAL, ES_UCHILD, ES_VDSCRT, FUSION_TYP, GA_ALPHA, GA_DMAX, GA_DMIN, GA_NBIN, GA_NBREED, GA_NCLUST, GA_NPOP, GA_NRANDM, GA_PBEST, GA_PCROSS, GA_PENAL, GA_PINVRT, GA_PMUTNT, GA_PRPTNL, GA_PSEED, MC_MAND, MC_P1, MC_P2, MC_PENAL, MC_TYPE, NAG_BIGBND, NAG_ETA, NAG_RHO, OBJ_BAD_PT, OPT_CTOL, OPT_STEP, OPT_TOL, OPT_TSIZE, OVR_CONV, OVR_MAND, OVR_NPTS, OVR_PENAL, OVR_SEED, OVR_SHRK, OVR_SIMP, OVR_STEP, OVR_STOP, PL_LRATE, PL_NBIN, PL_NPOP, PL_NRANDM, PL_PENAL, PL_PMUTNT, RSM_EIF_W, RSM_NCSKIP, RSM_NSKIP, RSM_NULL_T, SA_NBIN, SA_NRANDM, SA_PCOLD, SA_PENAL, SA_PMUTNT, SA_PTEMP, SA_PWIDTH, SA_SCHED, SC_BKORRL, SC_CONV, SC_DELI, SC_DELP, SC_DELS, SC_IELTER, SC_IREKOM, SC_KONVKR, SC_LR, SC_LS, SC_NACHKO, SC_NITERS, SC_NRANDM, SC_NS, SC_PENAL, SC_SN, SC_TYPE

Output argument

The structure STRUCTOUT contains the following fields:

VARS: The optimum design variables
OBJFUN: The objective function value at VARS
CONS: The constraint values at VARS

Following a direct search over the user's code the objective function and constraint search histories are returned to the user in to sub-structures, OBJTRC and CONSTRC (respectively). Following evaluation of a RSM search histories are returned in the field RSMTRC.

OBJTRC: The history of evaluations of the objective function
OBJTRC.NCALLS: The number of objective function evaluations
OBJTRC.OBJFUN: The values of the objective function (a vector of length OBJTRC.NCALLS)
OBJTRC.VARS: The variables at which the objective function was evaluated (size NVARS by OBJTRC.NCALLS)

CONSTRC: The history of evaluations of the constraints
CONSTRC.NCALLS: The number of constraint evaluations
CONSTRC.CONS: The values of the constraints (size NCONS by CONSTRC.NCALLS)
CONSTRC.VARS: The variables at which the constraints were evaluated (size NVARS by CONSTRC.NCALLS)
CONSTRC.UCONS: The upper limits to the constraints at each evaluation (size

NCONS by CONSTRC.NCALLS)

CONSTRC.LCONS: The lower limits to the constraints at each evaluation (size NCONS by CONSTRC.NCALLS)

If the field NUMUPDATE is specified in the input structure for a search over a RSM a sub-structure DOE_TRACE is returned containing suggested points that would improve the initial dataset.

DOE_TRACE: Suggested points that would improve the dataset

DOE_TRACE.NCALLS: The number of suggested update points

DOE_TRACE.VARS: The design variables

Following an optimisation over a RSM OptionsMatlab will return the search history in the following field of the output structure (OptionsMatlab 0.9.0+):

RSMTRC: Search history of points evaluated over a RSM

RSMTRC.NCALLS: The number of user specified points used

RSMTRC.VARS: The user-specified design points used

RSMTRC.OBJFUN: The value of the objective function RSM at the user-specified design points.

RSMTRC.CONS: The value of the problem constraint RSM at the user-specified design points.

RSMTRC.UCONS: The upper limits of the problem constraint at the user-specified design points.

RSMTRC.LCONS: The upper limits of the problem constraint at the user-specified design points.

If a genetic algorithm (OMETHD=4) is used OptionsMatlab will return the values of the GA variables that may be used to restart the genetic algorithm. This information is contained in the following field of the output structure:

GA_VARS: The GA restart variables

GA_VARS.GA_POP: The GA population design variable and fitness values

GA_VARS.GA_CODE: The final GA code string values

GA_VARS.GA_NRANDM: The random number sequence used by the genetic algorithm.

Following an optimisation over approximate values of the objective and constraint functions using a Stochastic Process Model (OBJMOD and CONMOD respectively) the values and limits of the hyper-parameters will be returned. The hyper-parameters used to approximate values of the objective function will be returned in the structure

OBJHYPER, and the constraint hyper-parameters will be returning in the structure CONHYPER. The structures OBJHYPER and CONHYPER are identical to the optional fields of the input structure described above.

Notes

OptionsMatlab requires a valid Options licence file.

See also

optjob, createBeamStruct

optjob

Multiple objective function and constraint evaluation for `OptionsMatlab`

`Optjob.m` provides an example implementation of a broker for multiple objective function and constraint evaluations for `OptionsMatlab`. These evaluations are done in serial, other implementations of `optjob` support concurrent evaluations.

The `optjob` function to be used by `OptionsMatlab` must be set in the `OPTJOB` field of the structure passed to `OptionsMatlab`. The function name of user-defined objective function and constraint functions should be set in the `OPTFUN` and `OPTCON` fields for the structure passed to `OptionsMatlab`. If the `OPTFUN` and `OPTCON` fields are equal it is assumed that the objective function routine will return constraint values as the fifth output argument (see below).

The user-defined objective function called by `optjob` should conform to the following function prototype:

```
      [eval,gd,H,PARAMS,CONS,U_CONS,L_CONS]=opfun(VARS,PARAMS,
U_CONS,L_CONS,DATA)
```

The user-defined constraint function called by `optjob` should conform to the following function prototype:

```
      [CONS,ceq,GC,Gceq,PARAMS,U_CONS,L_CONS]=objcon(VARS,
PARAMS,U_CONS,L_CONS,DATA)
```

Reimplementing optjob

`OPTJOB` takes a matrix of size `NJOBS` by `NVARS` and returns a vector of function evaluations of length `NJOBS`. Other arguments include the names of the user-defined `OPTFUN` and `OPTCON` functions, as well as user defined parameters, constraints and upper and lower constraint limits.

The minimum required function prototype of the `optjob` function is:

```
      [evals,cons] = optjob(optfunname,optconname,vars)
```

where the input arguments are:

`optfunname`: the name of the user defined objective function

`optconname`: the name of the user defined constraint functions

`vars`: the matrix of design variables size `NJOBS` by `NVARS` (where `NJOBS` is the number of design points to be evaluated and `NVARS` the number of design variables)

where the output arguments are:

`evals`: a vector of `NJOBS` function evaluations

`cons`: a matrix of `NJOBS` by `NCONS` constraints

The complete function prototype of the `optjob` function is:

```
    [evals,cons,params,u_cons,l_cons]=optjob(optfunnam,
optconnam,vars,params,cons,u_cons,l_cons,data)
```

as above where additional input arguments are

`params`: a vector of size `NPARAMS` of user-defined parameters

`cons`: a vector of `NCONS` constraints at the design variables to be evaluated (applies only to a single function evaluation for `OPTFUN` only otherwise empty)

`u_cons`: a vector of size `NCONS` of the upper limits for the user defined constraints (where there is no limit set to `inf`)

`l_cons`: a vector of size `NCONS` of the lower limits for the user defined constraints (where there is no limit set to `-inf`)

`data`: the user-supplied data passed unaltered from the field `USERDATA` of the input structure

and as above where additional output arguments are:

`params`: a revised vector of size `NPARAMS` of user defined parameters

`u_cons`: a revised vector of size `NCONS` of the upper limits for the user defined constraints

`l_cons`: a revised vector of size `NCONS` of the lower limits for the user defined constraints

The `optjob` Matlab function is invoked from three places with `OptionsMatlab`, `OPTJOB`, `OPTFUN` and `OPTCON`, each of these FORTRAN subroutines will use the `optjob` function in a different fashion.

`OPTJOB`: Calls the `optjob` Matlab function for `NJOBS` function (and constraint) evaluations. Passes all of the available input parameters, apart from the vector cons for which is substituted an empty array. The user-defined routine optfun must be called for each of the NJOBS designs supplied. If the problem is constrained the constraints will be can be evaluated by a combined objective/constraint function (where optfunname == optconname), or by a separate constraint function (defined by optconname). For unconstrained optimisations the argument optconname will be empty. OPTJOB requires the output arguments evals and cons to be returned, all other output arguments will be ignored.

OPTFUN: Calls the optjob Matlab function for a single function evaluation. Passes all of the available input parameters, apart from the string optconnam for which is substituted an empty string. The user-defined routine optfunname is called once. Requires the output argument evals to be returned, the cons output argument will be ignored, and the params, u_cons and l_cons output arguments will be used to update the corresponding values in the internal GENDAT database if returned.

OPTCON: Calls the optjob Matlab function for a single constraint evaluation. Passes all of the available input parameters, apart from the string optfunnam for which is substituted an empty string and the vector cons for which is substituted an empty array. The user-defined routine optconname is called once. Requires the output argument cons to be returned, the evals output argument will be ignored, and the params, u_cons and l_cons output arguments will be used to update the corresponding values in the internal GENDAT database if returned.

See also

OptionsMatlab, createBeamStruct

optjobparallel

Multiple objective function and constraint evaluation for OptionsMatlab

`Optjobparallel` evaluates user defined objective and constraint functions in parallel. To evaluate the objective function the user must define two functions, the first which initiates the calculation of the objective function, and the second which returns the values of the objective function.

In practice the first function will typically perform a Globus GRAM job submission returning a handle which can be polled and an application specific job ID. The second function will typically use the application specific job ID to retrieve the output of the GRAM job and parse the objective function (and optionally the values of the constraints also).

The user-defined objective function called by `optjobparallel` to perform the job submission should conform to the following function prototype:

`[JOBHANDLE,RETRIEVALID]=objfun(VARS,PARAMS,U_CONS,L_CONS, DATA)` where `JOBHANDLE` is a GRAM job handle which can be polled by `gd_jobpoll`, and `RETRIEVALID` is an identifier used by retrieve the results. If `JOBHANDLE` is empty it will not be polled. The only mandatory input argument is `VARS`, the other input arguments `PARAMS`, `U_CONS`, `L_CONS` and `DATA` are all optional.

This function must be specified in the `OPTFUN` field of the `OptionsMatlab` input structure.

A second retrieval function is be defined to return the value of the objective function. This function must have the same name as the job submission function appended with `'_parse'`. For example when the objective function submission function is saved in the file `'objfun.m'` the retrieval function must be saved in the file `'objfun_parse.m'`.

The retrieval function should conform to the following function prototype:

`[EVAL,PARAMS,CONS,U_CONS,L_CONS]=objfun_parse(RETRIEVALID )` where `RETRIEVALID` is the identifier returned by the job submission function.

EVAL is the value of the objective function. The other output arguments PARAMS, CONS, U_CONS and L_CONS are all optional. CONS is the value of the constraints.

If the value of the constraints and the objective function are return by the same function the field OPTCON should be set to equal OPTFUN. Alternatively if the constraints are evaluated independently of the objective function the user may also define two separate functions to perform the job submission and to parse the constraints. In this case the functions indicated by the field OPTCON should conform to the following function prototypes:

```
[JOBHANDLE,RETRIEVALID]=objcon(VARS,PARAMS,U_CONS,L_CONS,DATA)
[CONS,PARAMS,U_CONS,L_CONS] = objcon_parse(RETRIEVALID)
```

See also
optjob, OptionsMatlab, optjobparallel2

optjobparallel2

Multiple objective function and constraint evaluation for OptionsMatlab

optjobparallel2 evaluates user defined objective and constraint functions in parallel. To evaluate the objective function the user must define two functions, the first which initiates the calculation of the objective function, and the second which determines the state of the job and, if complete, return the value of the objective or constraint functions.

The user-defined objective function called by optjobparallel2 to perform the job submission should conform to the following function prototype:

RETRIEVALID = objfun(VARS,PARAMS,U_CONS,L_CONS,DATA) where RETRIEVALID is an identifier used by retrieve the results, for example this may be a structure containing a number of fields. The only mandatory input argument is VARS, the other input arguments PARAMS, U_CONS, L_CONS and DATA are all optional.

This function must be specified in the OPTFUN field of the OptionsMatlab input structure.

A second retrieval function is be defined to determine whether the job has completed, and if so return the value of the objective function. This function must have the same name as the job submission function appended with '_parse2'. For example, when the objective function submission function is saved in the file 'objfun.m' the retrieval function must be saved in the file 'objfun_parse2.m'.

The retrieval function should conform to the following function prototype:

[EVAL,PARAMS,CONS,U_CONS,L_CONS]=objfun_parse2(RETRIEVALID) where RETRIEVALID is the identifier returned by the job submission function. EVAL is the value of the objective function (or empty if the job has not completed). The other output arguments PARAMS, CONS, U_CONS and L_CONS are all optional. CONS is the value of the constraints.

This function should determine whether the job has completed. If the job has completed the value of EVAL (and that of CONS) should be returned. If the job is still running the function should return an empty value for EVAL (i.e. EVAL = []), in which case the status of other jobs will be determined before the `'_parse2'` function is invoked again for this job. If the job has failed a suitable bad point indicator should be returned.

If the value of the constraints and the objective function are return by the same function the field `'OPTCON'` should be set to equal `'OPTFUN'`. Alternatively if the constraints are evaluated independently of the objective function the user may also define two separate functions to perform the job submission and to parse the constraints. In this case the functions indicated by the field `'OPTCON'` should conform to the following function prototypes:

```
RETRIEVALID = objcon(VARS,PARAMS,U_CONS,L_CONS,DATA)
[CONS,PARAMS,U_CONS,L_CONS] = objcon_parse2(RETRIEVALID)
```

See also

`optjob`, `OptionsMatlab`

optum1

Example user-defined sequential optimiser for `OptionsMatlab`

`optum1` is a random sequential optimiser that demonstrates how to define an arbitrary optimisation strategy to be invoked by `OptionsMatlab`. This optimiser can be invoked by specifying the fields `OMETHD` = 2.1 and `OPTUM1` = `'optum1'` in the input structure of `OptionsMatlab`.

To implement your own optimiser your function should conform to the following function prototype. User-defined optimisers should minimise the objective function irrespective of the search direction specified by the input structure.

Syntax

```
      [VARS, STOPOPT] = OPTUM1(VARS, FVAL, CONS, UVARS, LVARS,
UCONS, LCONS, MAXCALLS, CALLNUM, TOL, STEPSIZE, OLEVEL)
```

Description

`[VARS, STOPOPT] = OPTUM1(VARS, FVAL, CONS, UVARS, LVARS, UCONS, LCONS, MAXCALLS, CALLNUM, TOL, STEPSIZE, OLEVEL)` where the meaning of the input arguments are:

`VARS`	vector containing the last evaluated value of `VARS`
`FVAL`	objective function value at `VARS`
`CONS`	vector of constraint values at `VARS` (empty if unconstrained)
`UVARS`	vector of upper limits for `VARS`
`LVARS`	vector of lower limits for `VARS`
`UCONS`	vector of upper limits for `CONS` (may vary)
`LCONS`	vector of lower limits for `CONS` (may vary)
`MAXCALLS`	maximum number of function evaluations, must be honoured by your implementation of `optum1`
`CALLNUM`	number of iterations performed
`TOL`	requested tolerance of the optimiser
`STEPSIZE`	requested step-size of the optimiser
`OLEVEL`	requested output level of the optimiser

where the meaning of the output arguments are:

`VARS`	vector containing the next value of `VARS` to be evaluated. If `STOPOPT` indicates that the optimiser is complete `VARS` should contain the minimum variable values detected by the optimiser

`STOPOPT`	a flag indicating the whether the optimiser has completed. The optimiser will run whilst `STOPOPT` = 0, and will complete when `STOPOPT = 1` is returned. If `STOPOPT` is not set to 1 the optimiser will run indefinitely.

Example

This example invokes the user-defined optimiser defined by `optum1` over the Beam problem.

```
input = createBeamStruct;
input.NITERS = 20;
input.OMETHD = 2.1;
input.OPTUM1 = 'optum1';
output = OptionsMatlab(input);
optimisationTrace(output,input,3)
```

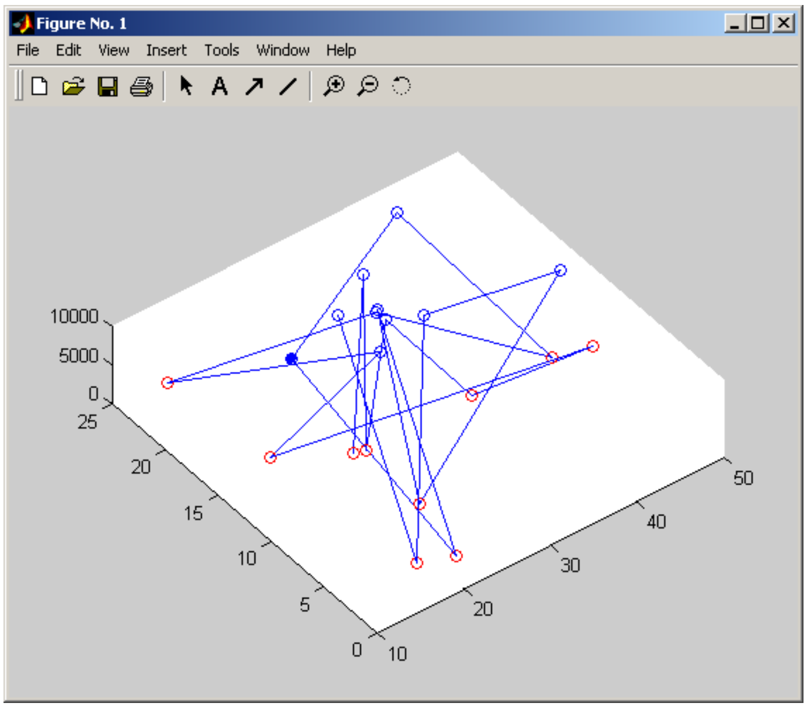


Figure 20 Trace produced by random optimiser `optum1`

See also

OptionsMatlab

Peaks4d problem

A four dimension problem based upon the Matlab `peaks` function.

Example

This example plots the objective function surface of the Peaks4D problem.

```
>> input = createpeaks4dstruct(2.8);
>> input.MC_TYPE = 4;
>> input.NITERS = 500;
>> input.UVARS = [3,3,0.01,0.01]; % hold vars 3 and 4 constant
>> output = OptionsMatlab(input)
```

```
output =

       VARS: [4x1 double]
     OBJFUN: 7.4643
       CONS: 0
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

```
>> optimisationTerrain(output,input,3)
```

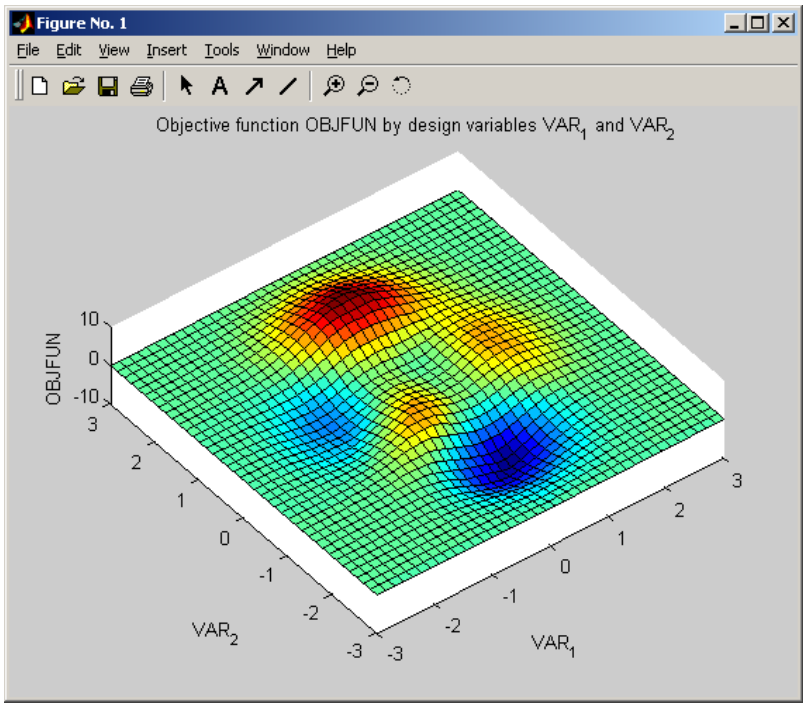


Figure No. 1
File Edit View Insert Tools Window Help
Objective function OBJFUN by design variables VAR$_1$ and VAR$_2$
OBJFUN
10
0
-10
VAR$_2$
3
2
1
0
-1
-2
-3
VAR$_1$
-3
-2
-1
0
1
2
3

Figure 21 The valid objective function surface of the Peaks4d problem

Functions

`peaks4d`	objective function
`createpeaks4dstruct`	creates an input structure for the peaks4d problem

5 Frequently Asked Questions

5.1 Why does Matlab crash when I call OptionsMatlab?

When OptionsMatlab is invoked with an invalid Gendat license file the error message below will be printed:

```
>> input = createBeamStruct;
>> output = OptionsMatlab(input);
```

```
??? Optimization failed. OPTIONS Error code:    -2147483648
```

Gendat license files may be invalid if they have expired, or if they contain incorrect machine details.

Write statements by some of the third party algorithms available within the Options have also caused OptionsMatlab to crash. These can be diagnosed by examining the temporary files generated by OptionsMatlab; `.OPTIONS*.opt` and `.OPTSDTO*.opt`. In some cases this behaviour can be ameliorated by reducing the output level of OptionsMatlab, `OLEVEL = 0`.

When creating a new problem definition conflicts between user-specified design variable, parameter or constraint names can cause OptionsMatlab to crash. Conflicts occur when there is an ambiguity between a variable name and the name of an existing Options variable. For example the variable name `FACT` would be ambiguous if the parameter `FACTOR` had previously been defined. If a variable name conflict has caused Matlab to crash this may be diagnosed by examining the temporary file `.OPTIONS*.opt`.

Please report any reoccurring problems to me by email. Bugs are documented in the buglists included in the OptionsMatlab distribution.

5.2 How do I specify the search method?

The search method is specified by the field `OMETHD` of the Options input structure. The scalar values correspond to the search methods listed below. For more details of each of the search methods please see the Options manual [1].

0.0	to just evaluate the user's problem code at the point specified
1.1	for OPTIVAR routine ADRANS
1.2	for OPTIVAR routine DAVID
1.3	for OPTIVAR routine FLETCH
1.4	for OPTIVAR routine JO
1.5	for OPTIVAR routine PDS
1.6	for OPTIVAR routine SEEK
1.7	for OPTIVAR routine SIMPLX
1.8	for OPTIVAR routine APPROX
1.9	for OPTIVAR routine RANDOM
2.1	for user specified routine OPTUM1
2.2	for user specified routine OPTUM2
2.3	for NAG routine E04UCF
2.4	for bit climbing
2.5	for dynamic hill climbing
2.6	for population based incremental learning
2.7	for numerical recipes routines
2.8	for design of experiment based routines
3.11	for Schwefel library Fibonacci search
3.12	for Schwefel library Golden section search
3.13	for Schwefel library Lagrange interval search
3.2	for Schwefel library Hooke and Jeeves search
3.3	for Schwefel library Rosenbrock search
3.41	for Schwefel library DSCG search
3.42	for Schwefel library DSCP search
3.5	for Schwefel library Powell search
3.6	for Schwefel library DFPS search
3.7	for Schwefel library Simplexsearch
3.8	for Schwefel library Complexsearch
3.91	for Schwefel library two-membered evolution strategy
3.92	for Schwefel library multi-membered evolution strategy
4	for genetic algorithm search
5	for simulated annealing
6	for evolutionary programming

7	for evolution strategy

5.3 How do I run a Design of Experiments?

A Design of Experiments search can be used to efficiently sample points across the multi-dimensional parameter space represented by large numbers of design variables. A Design of Experiments search can be invoked by setting `OMETHD = 2.8`. The number of points to be evaluated can be configured by altering the input structure field `NITERS`.

A number of different Design of Experiments search methods are available within the Options package. These can be configured using the optional input field `MC_TYPE`, where;

1	Random (default)
2	$LP\tau$
3	Central composite and $LP\tau$
4	Full factorial and $LP\tau$
5	Latin hypercubes
6	Cell-based latin hypercubes
7	User supplied candidate points

For more details about these Design of Experiments search methods please consult the Options manual [1].

User supplied candidate points to be evaluated during a Design of Experiments can be supplied with the optional input field `DOE_TRACE` when the control parameter `MC_TYPE = 7`. `DOE_TRACE` requires two mandatory fields:

`DOE_TRACE.NCALLS` containing the number of user-supplied DOE points

`DOE_TRACE.VARS` the design points to be evaluated during the DOE (size `NVARS` by `DOE_TRACE.NCALLS`)

When using user supplied candidate points `NITERS` must equal `DOE_TRACE.NCALLS` plus one as the Design of Experiments will first evaluate the design point specified by `VARS`.

5.4 How do I build a Response Surface Model?

A Response Surface Model is used to approximate the value of objective or constraint functions based upon the results of direct evaluation of the user's model. Response Surface Models can be built independently over the objective and constraints, and are configured using the optional input fields `OBJMOD` and `CONMOD` respectively. If these fields are not set `OptionsMatlab` will directly evaluate the user supplied objective and constraint functions.

A number of Response Surface Model methods are available to be used to approximate the values of the objective function and constraints. The possible settings for the optional input fields `OBJMOD` and `CONMOD` are:

1.0	for a Shepard response surface model
2.1	for linear Radial Basis Function
2.2	for thin plate Radial Basis Function
2.3	for cubic splines Radial Basis Function
2.4	for cubic splines Radial Basis Function with regression via reduced bases
3.1	for mean polynomial regression model
3.2	for first order polynomial regression model
3.3	for first order polynomial regression model plus squares
3.4	for first order polynomial regression model plus products (cross-terms)
3.5	for second order polynomial regression model
3.6	for second order polynomial regression model plus cubes
4.1	for a Stochastic Process Model
4.2	for the root mean square error of the Stochastic Process Model
4.3	for the expected improvement of the Stochastic Process Model
4.31	for the expected improvement of the constrained Stochastic Process Model [requires CONMOD=4.1]
4.32	for the feasibility of improvement of the constrained Stochastic Process Model [requires CONMOD=4.1]
4.33	for the probability of improvement of the Stochastic Process Model
0.0	if the underlying user supplied function is to be called.

5.5 How do I plot my Response Surface Model?

Following OptionsMatlab version 0.9.0+ search histories are available for optimisations which are run over a Response Surface Model in the output structure field RSMTRC. Previously OptionsMatlab would only return a search history when candidate points were provided.

To evaluate a factorial search of the RSM that is suitable for plotting it may be appropriate to evaluate a list of candidate points. The candidate points must be provided in a field DOE_TRACE of the input structure (see section 5.3). In versions of OptionsMatlab 0.9.0+ it is necessary to specify that the optimisation is a candidate points Design of Experiments (OMETHD=2.8, MC_TYPE=7).

```
>> %Create the initial dataset
>> DOEinput = createBeamStruct;
>> DOEoutput = OptionsMatlab(DOEinput);
>> %Define a RSM input structure
>> RSMinput = createBeamStruct;
>> RSMinput.OBJMOD = 3.3;
>> RSMinput.CONMOD = 3.3;
>> %Create a list of candidate points to be evaluated
>> ii = linspace(DOEinput.LVARS(1),DOEinput.UVARS(1),10);
>> jj = linspace(DOEinput.LVARS(2),DOEinput.UVARS(2),10);
>> [x,y] = meshgrid(ii,jj);
>> RSMinput.DOE_TRACE.VARS(1,:)=
reshape(x,1,prod(size(x)));
>> RSMinput.DOE_TRACE.VARS(2,:)=
reshape(y,1,prod(size(y)));
>> RSMinput.DOE_TRACE.NCALLS = prod(size(x));
>> %Define the search a candidate points DoE
>> RSMinput.OMETHD = 2.8;
>> RSMinput.MC_TYPE = 7;
>> RSMoutput = OptionsMatlab(RSMinput, DOEoutput);
>> disp(RSMoutput.RSMTRC)
```

```
     OBJFUN: [1x100 double]
       VARS: [2x100 double]
     NCALLS: 100
       CONS: [5x100 double]
      LCONS: [5x100 double]
      UCONS: [5x100 double]
```

The contents of RSMTRC can then be plotted to show the surface of the Response Surface Model.

```
>> optimisationTerrain(RSMoutput, RSMinput);
```

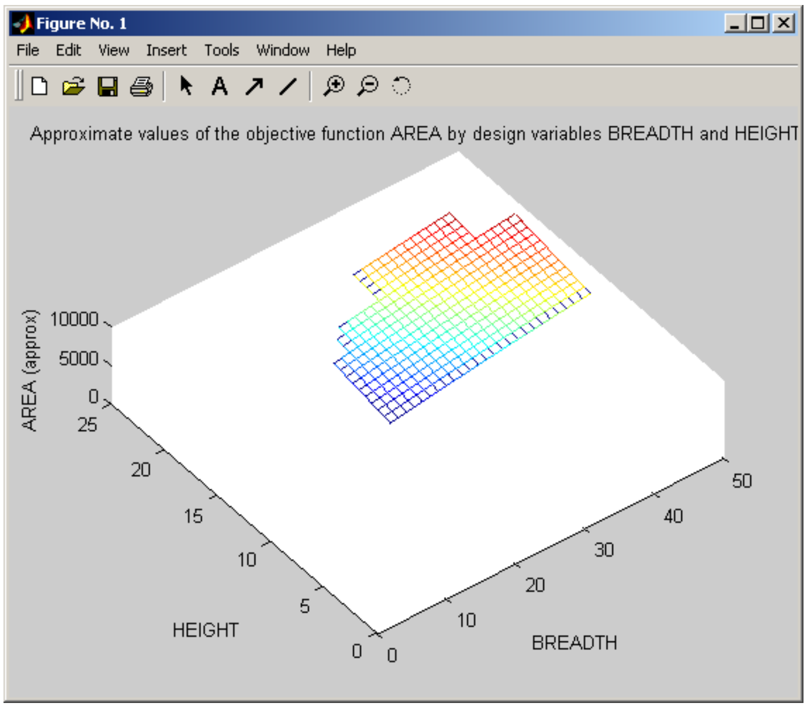


Figure 22 Plotting approximate values of the Beam objective function generated by a RSM

The utility function optimisationSampleRSM automates the process of sampling a RSM built over the user's problem.

5.6 How do I generate Design of Experiment update points?

It is possible to improve the quality of a Response Surface Model by improving to original dataset by selectively adding new points. The Genetic Algorithm (OMETHD = 4) and Dynamic Hill Climbing (OMETHD = 2.5) optimisation algorithms, when run over a Response Surface Model, are capable of returning a list of points that would improve the dataset.

Update points will be returned if the OptionsMatlab input structure contains the optional field `NUMUPDATE`. The value of `NUMUPDATE` is a scalar which determines the number of update points to be returned when a search routine is run over a RSM. The update points will be returned in the field `DOE_TRACE` of the output structure.

In the following example a Genetic Algorithm is run over a RSM generated from the search history contained in the structure `DOEoutput`. `NUMUPDATE` is set to equal 10, meaning that the Genetic Algorithm will suggest ten update points at which the original data set can be improved.

Note that the optimisation algorithm may return less than `NUMUPDATE` update points, in this case the remaining elements of `DOE_TRACE.VARS` will contain zeros.

```
>> %Create the initial dataset
>> DOEinput = createBeamStruct;
>> DOEoutput = OptionsMatlab(DOEinput);
>> %Define a RSM input structure
>> RSMinput = createBeamStruct;
>> RSMinput.OMETHD = 4;
>> RSMinput.OBJMOD = 3.3;
>> RSMinput.CONMOD = 3.3;
>> RSMinput.NUMUPDATE = 10;
>> RSMoutput = OptionsMatlab(RSMinput, DOEoutput);
>> disp(RSMoutput.DOE_TRACE)
```

```
    NCALLS: 10
      VARS: [2x10 double]
```

The update points contained in the field `DOE_TRACE` of the structure `RSMoutput` can now be used as candidate points for a second Design of Experiments study.

```
>> DOEinput2 = createBeamStruct;
>> DOEinput2.OMETHD = 2.8;
>> DOEinput2.MC_TYPE = 7;
>> DOEinput2.DOE_TRACE = RSMoutput.DOE_TRACE;
```

```
      >> DOEinput2.NITERS = RSMoutput.DOE_TRACE.NCALLS+1;
      >> DOEoutput2 = OptionsMatlab(DOEinput2);
```

Note that `DOEinput2.NITERS` must equal `DOEinput2.DOE_TRACE.NCALLS` plus one as the Design of Experiments will first evaluate the design point specified by `DOEinput2.VARS`.

5.7 How do I define an unconstrained optimisation?

From version 0.5 of OptionsMatlab onwards users do not have to define a null constraint function for unconstrained optimisation problems. To indicate that an optimisation problem is unconstrained the field `NCONS` should be set to 0. In this case the fields `CNAM`, `LCONS`, `UCONS`, `CONS` and `OPTCON` are not mandatory and will be ignored.

5.8 How do I write my own objective and constraint functions?

The default implementation of `OPTJOB` (`optjob.m`) requires user-defined objective and constraint functions to conform to well-defined interfaces. These interfaces are design to be compatible with objective and constraint functions used with the Matlab Optimization Toolbox [4].

The full function signature for the user-defined objective function is:

```
      [eval,gd,H,PARAMS,CONS,U_CONS,L_CONS]=objfun(VARS,PARAMS,
U_CONS,L_CONS,DATA)
```

Where `eval` is the value of the objective function at the design variables `VARS`. The objective function corresponding to this header can return the constraint values for the design point, `CONS`, and also alter the values of the parameters, `PARAMS`, and constraint limits `U_CONS` and `L_CONS`. The argument `DATA` contains the Matlab variable contained in the optional `USERDATA` field of the input structure. The parameters `gd` and `H` are relevant to the Matlab Optimization Toolbox [4] and are not used by OptionsMatlab.

NOTE: The full function signature for user-defined objective function has changed in OptionsMatlab version 0.7. In earlier versions the third optional input argument was `CONS`, the value of the constraints at `VARS`. However this feature was unreliable and has been removed. Please update objective functions that use the earlier form of the function signature.

The minimum function signature required by `optjob.m` is:

```
eval = objfun(VARS)
```

The full function signature for the user-defined constraint function is:

```
      [CONS,ceq,GC,Gceq,PARAMS,U_CONS,L_CONS]=objcon(VARS,PARAM
S,U_CONS,L_CONS,DATA)
```

Where `CONS` are the constraint values at the design variables `VARS`. The parameters `ceq`, `GC` and `Gceq` are relevant to the Matlab Optimization Toolbox [4] and are not used by OptionsMatlab.

Again the minimum function signature required by `optjob.m` is a lot smaller:

```
CONS = objcon(VARS)
```

Alternative implementations of `OPTJOB` may require different function signatures from user-defined objective and constraint functions. Please consult the documentation of alternative implementations of `OPTJOB` to confirm that your objective and constraint functions conform to the requirements.

Note that the OptionsMatlab may ignore altered values of the parameters, `PARAMS`, and constraint limits `U_CONS` and `L_CONS` if it is not appropriate to change them, for example during a Design of Experiments.

5.9 How do I evaluate a combined objective and constraint function?

The default implementation of `OPTJOB` (`optjob.m`) supports combined objective and constraint functions. The combined function must conform to following objective function signature;

```
[eval,gd,H,PARAMS,CONS,...] = objfun(VARS,...)
```

`optjob.m` will evaluate this function once when evaluating objective and constraint functions if the input fields `OPTFUN` and `OPTCON` specify the same function.

NOTE: The full function signature for user-defined objective function has changed in OptionsMatlab version 0.7. In earlier versions the third optional input argument was `CONS`, the value of the constraints at `VARS`. However this feature was unreliable and has been removed. Please update objective functions that use the earlier form of the function signature.

5.10 Can OptionsMatlab calculate function evaluations in parallel?

The standard OptionsMatlab job manager, `optjob.m`, will evaluate the objective and constraint functions sequentially. However a parallel job manager, `optjobparallel2`, is included in the OptionsMatlab distribution (this supersedes the parallel job manager `optjobparallel`). When your objective or constraint function is expensive and you wish to use a search method with inherent parallelism it may be more considerably efficient to use the parallel job manager.

To run the demo of parallel objective function evaluations enter the following commands:

```
>> input = createBeamStructParallel2
>> output = OptionsMatlab(input)
```

To make your objective and constraint functions available to `optjobparallel2` different function signatures are required to those described in section 5.8. To evaluate the objective function the user must define two functions, the first which initiates the calculation of the objective function, and a second which determines whether the calculation has completed, and if so returns the value of the objective function.

In practice the first function could perform a Globus GRAM job submission [5] returning a handle which can be used to query the status of the job, and an application specific job ID. The second function will typically use the application specific job ID to retrieve the output of the GRAM job and parse the objective function (and optionally the values of the constraints also). The interaction between these functions is shown by Figure 23.

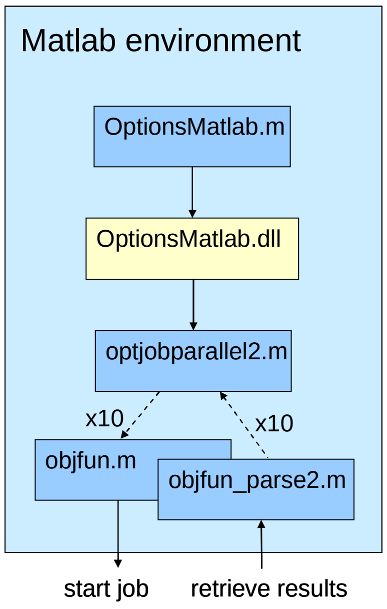


Figure 23 Parallel objective function evaluation in OptionsMatlab. Objfun.m is called ten times to begin the objective function evaluation at ten points. When these jobs are complete objfun_parse2.m is called ten times to retrieve and parse the results

The user-defined objective function called by `optjobparallel2` to perform the job submission should conform to the following function prototype:

```
[RETRIEVALID] = objfun(VARS,...)
```

where `RETRIEVALID` is an identifier used to determine the status of the job, and to retrieve the results. The only mandatory input argument is `VARS`, the other input arguments `PARAMS`, `U_CONS` and `L_CONS` are all optional. This function must be specified in the `OPTFUN` field of the OptionsMatlab input structure.

A second retrieval function is be defined to return the value of the objective function. This function must have the same name as the job submission function appended with `'_parse2'`. For example when the objective function submission function is saved in the file `'objfun.m'` the retrieval function must be saved in the file `'objfun_parse2.m'`.

The retrieval function should conform to the following function prototype:

```
[EVAL,PARAMS,CONS,U_CONS,L_CONS]=objfun_parse2(RETRIEVALID)
```

where `RETRIEVALID` is the identifier returned by the job submission function. `EVAL` is the value of the objective function (or empty if the job has not completed). The other output arguments PARAMS, CONS, U_CONS and L_CONS are all optional. CONS is the value of the constraints.

If the value of the constraints and the objective function are returned by the same function the field OPTCON should be set to equal OPTFUN. Alternatively if the constraints are evaluated independently of the objective function the user may also define two separate functions to perform the job submission and to parse the constraints. In this case the functions indicated by the field OPTCON should conform to the following function prototypes:

```
[JOBHANDLE] = objcon(VARS,PARAMS,U_CONS,L_CONS)
[CONS,PARAMS,U_CONS,L_CONS] = objcon_parse2(RETRIEVALID)
```

5.11 How do I tune the hyper-parameters for a stochastic process model RSM?

Instead of searching the user's problem OptionsMatlab can be used to tune the hyper-parameters for a stochastic process model RSM. This can be done by setting up the OptionsMatlab input structure as though you are going to build a RSM (see section 5.4) over an existing search history. Hyper-parameter tuning is specified by setting the input structure field TUNEHYPER equal to 1.

When TUNEHYPER is set the hyper-parameters are tuned using the search method specified by the input structure. The output structure will return the structures OBJHYPER (and/or CONHYPER where appropriate) in addition to the final value of the concentrated likelihood function which is used as the objective function OBJ_CLF (or CST_CLF). Note that the user's problem is not searched, and no optimum for the user's problem is returned.

To use the tuned hyper-parameters to build and search a RSM, or to further tune the hyper-parameters, the structures OBJHYPER and CONHYPER can be passed as fields in the OptionsMatlab input structure. These structures contain the hyper-parameter values, and upper and lower limits to these values.

The example below demonstrates hyper-parameter tuning by performing the following steps:

- training hyper-parameters over a data set

- refining hyper-parameters with further training
- searching a RSM with user supplied hyper-parameters
- searching a RSM with starting at the previous 'best-point'

This example uses the Beam problem.

```
% Build initial dataset
input1 = createBeamStruct;
input1.OMETHD = 2.8;        %Design of Experiments
input1.NITERS = 50;         %Number of iterations
input1.OLEVEL = 2;
input1.MC_TYPE = 4;         %Full factorial DoE
output1 = OptionsMatlab(input1)
```

```
output1 =


       VARS: [2x1 double]
     OBJFUN: 3.6877e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

```
% Tune hyper-parameters with SA
input2 = createBeamStruct;
input2.OLEVEL = 2;
input2.OBJMOD = 4.1;     %Tune Stochastic Process Model
                         %hyper-parameters over the objective
                         %function
input2.CONMOD = 4.1;     %Tune Stochastic Process Model
                         %hyper-parameters over the constraints
input2.TUNEHYPER = 1;    %Tune the hyper-parameters
                         %(do not search the user's problem)
input2.OMETHD = 5;       %Simulated Annealing
output2 = OptionsMatlab(input2, output1)
```

```
output2 =


    OBJHYPER: [1x1 struct]
     OBJ_CLF: 712.6938
    CONHYPER: [1x1 struct]
     CST_CLF: 824.2750
```

```
% Further train user-supplied hyper-parameters with GA
input3 = input2;
% Note that if OBJHYPER or CONHYPER are provided these
% hyper-parameters will be used in preference to those
% generated by OPTRSS
input3.OBJHYPER = output2.OBJHYPER;
input3.CONHYPER = output2.CONHYPER;
input3.OMETHD = 4;
output3 = OptionsMatlab(input3, output1)
```

```
output3 =


    OBJHYPER: [1x1 struct]
     OBJ_CLF: 842.2571
    CONHYPER: [1x1 struct]
     CST_CLF: 892.1499
```

```
% Search RSM using user-supplied hyper-parameters
input4 = input1;
input4.OBJMOD = 4.1;
input4.CONMOD = 4.1;
input4.OBJHYPER = output3.OBJHYPER;
input4.CONHYPER = output3.CONHYPER;
input4.OMETHD = 5;
input4.NITERS = 5000;
input4.OLEVEL = 2;
output4 = OptionsMatlab(input4, output1)
```

```
output4 =


        VARS: [2x1 double]
      OBJFUN: 2.1522e+003
        CONS: [5x1 double]
    OBJHYPER: [1x1 struct]
    CONHYPER: [1x1 struct]
```

```
% Search RSM using user-supplied hyper-parameters at the
% previous best point
input5 = input4;
input5.OMETHD = 4;
input5.NITERS = 50;
% Reset starting point to previous best
input5.VARS = output4.VARS';
output5 = OptionsMatlab(input5, output1)
```

```
output5 =


        VARS: [2x1 double]
      OBJFUN: 2.4426e+003
        CONS: [5x1 double]
    OBJHYPER: [1x1 struct]
    CONHYPER: [1x1 struct]
```

For more details on the stochastic process model and hyper-parameter tuning see chapter 10 of the Options manual [1].

5.12 Can I checkpoint the progress of an optimisation?

During a lengthy optimisation it can be reassuring to checkpoint its progress. OptionsMatlab can write the current objective function and constraint search histories to file following a call to `OPTJOB`. Checkpointing can be switched on by setting the checkpoint interval in the field `CHKPT_INTV` of the input structure (`CHKPT_INTV` should be a multiple of `MAXJOBS`).

When checkpointing is used the search histories for the objective function and constraint search histories are written to file. The file format used is the binary Matlab `.MAT` format. The file name can be specified with the optional field `CHKPT_FILE` of the input structure.

5.13 How do I pass Matlab variables to my objective function?

OptionsMatlab supports the optional input structure field `USERDATA`. This field can be used to pass any Matlab variable (including structures or cell arrays) to the user-defined objective and constraint functions. To use the information contained within `USERDATA` in your objective function you must you must accept a sixth input argument `DATA` (see section 5.8). To access the variable from a separate constraint function the constraint function must accept a fifth input argument `DATA`.

Please note that the `USERDATA` field is supported by the `OPTJOB` functions supplied with OptionsMatlab (`optjob.m` and `optjobparallel.m`), however the `USERDATA` field may not be supported by older `OPTJOB` functions.

5.14 How do I define discrete design variables?

By default design variables in OptionsMatlab are contiguous between upper and lower limits; however it is possible to specify discrete values for one or more of the design variables. To use discrete variables the fields `NDVRS` and `DVARS` of the input structure must be configured appropriately.

The field `NDVRS` must be set equal to the maximum number of discrete design variable values for any single design variable. In the example below one of the design variables has three possible discrete states, whilst the second is contiguous; therefore we set `NDVRS` equal to 3.

The field `DVARS` is a matrix of size `NVRS` by `NDVRS` which contains the discrete design variable values for each of the design variables. Therefore in the example below the three possible discrete states of the first design variable are place in the first row of `DVARS`. Because the second design variable is contiguous all values of the second row are set equal to `DNULL`. If a design variable has fewer possible discrete values fewer than `NDVRS`, the remaining elements of `DVARS` should be set to `DNULL`.

The example below illustrates the use of discrete design variable values with the Banana problem.

```
>> % Create an unconstrained input structure
>> input = createbananastruct;
>> % Set the maximum number of discrete variable states
(between all design variables)
>> input.NDVRS = 3;
>> % Resize the matrix of discrete design variable values (set
to DNULL for contiguous design variables)
>> input.DVARS = ones(input.NVRS, input.NDVRS) * input.DNULL;
>> % Set discrete values for the first design variable (the
second design variable will remain contiguous)
>> input.DVARS(1,:) = [0, 0.5, 1]
>> disp(input.DVARS)
```

```
         0    0.5000    1.0000
 -777.0000 -777.0000 -777.0000
```

```
>> % Run the optimisation
>> results = OptionsMatlab(input);
>> % Plot the output of the optimisation to demonstrate
discrete variables
>> optimisationTrace(results, input, 1, 1, [-37.5, 30], [], 1)
```

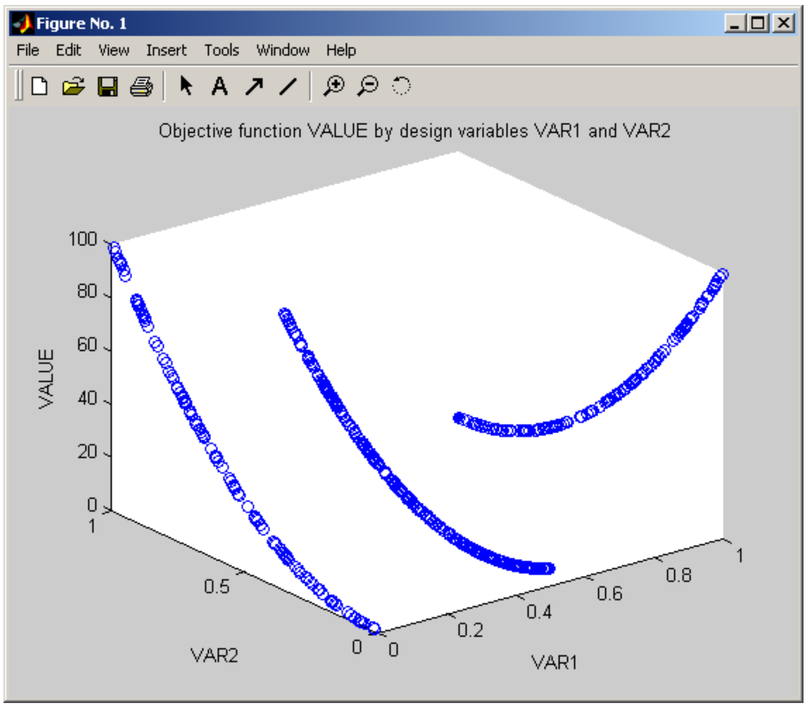


Figure 24 Example of a problem with one discrete variable and one contiguous variable

5.15 How do I restart a Genetic Algorithm?

The structure `GA_VARS`, which is contained in the `OptionsMatlab` output and checkpoint structures when a Genetic Algorithm is used (`OMETHD` = 4), allows the user to restart a Genetic Algorithm from its previous state. The following example demonstrates a Genetic Algorithm restarted from the output of an earlier calculation:

```
>> %Run a Genetic Algorithm
>> input1 = createBeamStruct;
>> input1.NITERS = 500;
>> input1.OMETHD = 4;
>> input1.GA_NPOP = 50;
>> output1 = OptionsMatlab(input1)
```

```
output1 =


       VARS: [2x1 double]
     OBJFUN: 2.6884e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
    GA_VARS: [1x1 struct]
```

```
>> %Restart a Genetic Algorithm
>> input2 = input1;
>> input2.GA_VARS = output1.GA_VARS;
>> input2.NITERS = 50;
>> output2 = OptionsMatlab(input2)
```

```
output2 =


       VARS: [2x1 double]
     OBJFUN: 2.6884e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
    GA_VARS: [1x1 struct]
```

```
>> %Plot the history of the two optimisations
>> optimisationHistory({output1, output2}, {'First run',
'Second run'})
```

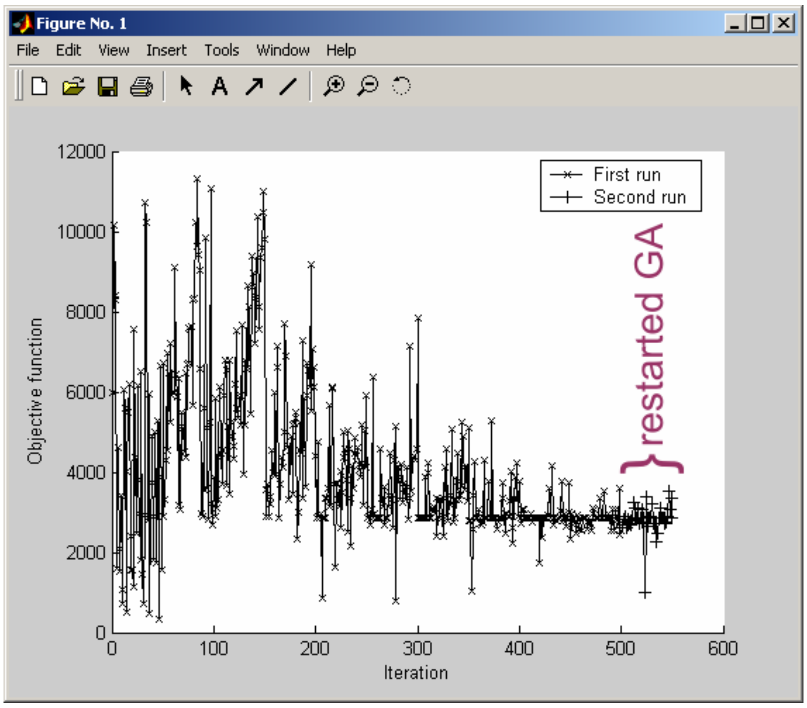


Figure 25 A Genetic Algorithm restarted following 500 iterations is already adapted to the objective function surface

5.16 What is the meaning of the optional control parameters?

Table 1 contains the meaning and default value of the optional control parameters. Since the meaning of the control parameters may differ depending upon the optimisation method in use the control parameters are organised with respect to the optimisation method.

Optimisation Method	Control Parameter	Meaning	Default value
Response Surface Modelling	`FUSION_TYP`	Flag to indicate RSM fusion type (differences=0, ratios=1)	0
	`CST_BAD_PT`	The outer limit of acceptable constraint function values in RSMs	None
	`OBJ_BAD_PT`	The outer limit of acceptable objective function values in RSMs	None
	`RSM_EIF_W`	The weighting between exploitation and exploration used when applying expected improvement methods in RSM	None
	`RSM_NCSKIP`	Number of radial basis functions skipped for constraints	0
	`RSM_NSKIP`	Number of radial basis functions skipped for objective function	0
	`RSM_NULL_T`	Percentage worsening required in RBF regression to halt fitting	10%
1.1 OPTIVAR routine ADRANS	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	0.001

	`OPT_STEP`	The step size used	0.02
	`OVR_MAND`	Turns on mandatory design constraints	0 (off)
	`OVR_PENAL`	Selects the kind of penalty function used by a number of the OPTIVAR routines: 1 = one pass external 2 = Fiacco-McCormick 3 = Powell 4 = Schuldt	1
	`OVR_SEED`	Sets the seed for random number sequences	128
1.2 OPTIVAR routine DAVID	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	0.001
	`OPT_STEP`	The step size used	1.00E-06
	`OVR_MAND`	Turns on mandatory design constraints	0 (off)
	`OVR_PENAL`	Selects the kind of penalty function used by a number of the OPTIVAR routines: 1 = one pass external 2 = Fiacco-McCormick 3 = Powell 4 = Schuldt	1
	`OVR_CONV`	Sets the convergence criterion	1D-4/1D-5
1.3 OPTIVAR routine FLETCH	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	0.001
	`OPT_STEP`	The step size used	1.00E-06
	`OVR_MAND`	Turns on mandatory design constraints	0 (off)
	`OVR_PENAL`	Selects the kind of penalty function used by a number of the OPTIVAR routines: 1 = one pass external 2 = Fiacco-McCormick 3 = Powell 4 = Schuldt	1
	`OVR_CONV`	Sets the convergence criterion	1D-4/1D-5
1.4 OPTIVAR routine JO	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	1.00E-06
	`OVR_MAND`	Turns on mandatory design constraints	0 (off)
	`OVR_PENAL`	Selects the kind of penalty function used by a number of the OPTIVAR routines: 1 = one pass external 2 = Fiacco-McCormick 3 = Powell 4 = Schuldt	1
	`OVR_CONV`	Sets the convergence criterion	1D-4/1D-5
1.5 OPTIVAR routine PDS	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	0.1
	`OVR_MAND`	Turns on mandatory design constraints	0 (off)
	`OVR_PENAL`	Selects the kind of penalty function used by a number of the OPTIVAR routines: 1 = one pass external 2 = Fiacco-McCormick	1

		3 = Powell 4 = Schuldt	
	`OVR_CONV`	Sets the convergence criterion	1D-4/1D-5
1.6 OPTIVAR routine SEEK	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	0.01
	`OVR_MAND`	Turns on mandatory design constraints	0 (off)
	`OVR_PENAL`	Selects the kind of penalty function used by a number of the OPTIVAR routines: 1 = one pass external 2 = Fiacco-McCormick 3 = Powell 4 = Schuldt	1
	`OVR_STOP`	sets the minimum step length stopping criterion	0.01
1.7 OPTIVAR routine SIMPLX	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	0.1
	`OVR_MAND`	Turns on mandatory design constraints	0 (off)
	`OVR_PENAL`	Selects the kind of penalty function used by a number of the OPTIVAR routines: 1 = one pass external 2 = Fiacco-McCormick 3 = Powell 4 = Schuldt	1
	`OVR_CONV`	Sets the convergence criterion	1D-4/1D-5
1.8 OPTIVAR routine APPROX	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	0.001
	`OVR_MAND`	Turns on mandatory design constraints	0 (off)
	`OVR_PENAL`	Selects the kind of penalty function used by a number of the OPTIVAR routines: 1 = one pass external 2 = Fiacco-McCormick 3 = Powell 4 = Schuldt	1
	`OVR_STEP`	Sets the fraction of range limiting step lengths	0.1
	`OVR_SIMP`	Sets the maximum number of simplex iterations	46
1.9 OPTIVAR routine RANDOM	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	0.02
	`OVR_MAND`	Turns on mandatory design constraints	0 (off)
	`OVR_PENAL`	Selects the kind of penalty function used by a number of the OPTIVAR routines: 1 = one pass external 2 = Fiacco-McCormick 3 = Powell 4 = Schuldt	1

	OVR_NPTS	Sets the number of points retained per iteration	5
	OVR_SHRK	Sets the shrinkage factor	4
2.3 NAG routine E04UCF	NAG_BIGBND	Sets the size of non-existent upper bounds.	1.00E+10
	NAG_ETA	Sets the accuracy of the linear minimizations	0.5
	NAG_RHO	Used in the definition of the augmented Lagrangian function	1
	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	5.0^{N}
	MC_MAND	Turns on mandatory design constraints	0 (off)
2.4 bit climbing	BC_NBIN	The number of bits used per variable in binary discretisation	12
	BC_PENAL	Set the penalty function control parameter, r, with values less than one invoking the modified Fiacco and McCormick function (OPTIM2) otherwise the one pass method is used (OPTIM1)	1.00E+20
	BC_NRANDM	The number of random numbers drawn and discarded before starting the optimiser	0
	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	MC_MAND	Turns on mandatory design constraints	0 (off)
2.5 dynamic hill climbing	DHC_INITSZ	Sets the non-dimensional size of the initial steps in the hill climbing search	0.5
	DHC_THRESH	The hill climbing searches proceed with reducing step sizes until they are less than the value set by this parameter	0.01
	DHC_PENAL	Sets the penalty function control parameter, r, with values less than one invoking the modified Fiacco and McCormick function (OPTIM2) otherwise the one pass method is used (OPTIM1)	1.00E+20
	DHC_NRANDM	The number of random numbers drawn and discarded before starting the optimiser	0
	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	MC_MAND	Turns on mandatory design constraints	0 (off)
2.6 population based incremental learning	PL_NBIN	The number of bits used per variable in binary discretisation	12
	PL_NPOP	The number of random guesses	100
	PL_PENAL	Sets the penalty function control parameter, r, with values less than one	1.00E+20

		invoking the modified Fiacco and McCormick function (OPTIM2) otherwise the one pass method is used (OPTIM1)	
	PL_LRATE	The learning rate controls how rapidly the probability vector changes towards the successful solutions at the end of each generation	0.05
	PL_PMUTNT	mutation is applied to the probability vector randomly at the end of each generation with this probability per element	0.02
	PL_NRANDM	The number of random numbers drawn and discarded before starting the optimiser	0
	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	MC_MAND	Turns on mandatory design constraints	0 (off)
2.7 numerical recipes routines	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	MC_MAND	Turns on mandatory design constraints	0 (off)
	MC_TYPE	Selects the kind of optimizer used by the numerical recipes routines: 1 = Powell 2 = Polak-Ribiere 3 = Fletcher-Reeves 4 = Broyden-Fletcher	1
	MC_PENAL	Selects the kind of penalty function used by the numerical recipes routines: 1 = one pass external 2 = Fiacco-McCormick	1
2.8 design of experiment based routines	DOE_NRANDM	DoE sequence random number seed	
	MC_TYPE	DoE search methods: 1 = Random 2 = Lptau 3 = Central composite + Lptau 4 = Full factorial + Lptau 5 = Latin hypercubes 6 = Cell-based latin hypercubes 7 = User supplied candidate points	1
	MC_MAND	Turns on mandatory design constraints	0 (off)
2.9 design of experiment based routines (without function calls)	DOE_NRANDM	Six Design of Experiment search methods	0
	MC_MAND	Turns on mandatory design constraints	0 (off)
3.11 Schwefel library Fibonacci search	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints	1.00E-03

		must be met to be considered satisfied	
	OPT_STEP	The step size used	1.00E-05
	SC_PENAL	Selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines: 1 = one pass external 2 = Fiacco-McCormick	1
3.12 Schwefel library Golden section search	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	SC_PENAL	Selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines: 1 = one pass external 2 = Fiacco-McCormick	1
3.13 Schwefel library Lagrange interval search	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	SC_PENAL	Selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines: 1 = one pass external 2 = Fiacco-McCormick	1
3.2 Schwefel library Hooke and Jeeves search	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	SC_PENAL	Selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines: 1 = one pass external 2 = Fiacco-McCormick	1
3.3 Schwefel library Rosenbrock search	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
3.41 Schwefel library DSCG search	OPT_TOL	The accuracy with which solutions are found	1.00E-03
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	SC_PENAL	Selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines: 1 = one pass external 2 = Fiacco-McCormick	1

3.42 Schwefel library DSCP search	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	1.00E-05
	`SC_PENAL`	Selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines: 1 = one pass external 2 = Fiacco-McCormick	1
3.5 Schwefel library Powell search	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	1.00E-05
	`SC_PENAL`	selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines : 1 = one pass external 2 = Fiacco-McCormick	1
	`SC_TYPE`	Selects the default convergence criterion or an alternate criterion: 1 = default convergence 2 = alternate convergence	1
3.6 Schwefel library DFPS search	`OPT_TOL`	The accuracy with which solutions are found	0
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	1.00E-05
	`SC_PENAL`	Selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines: 1 = one pass external 2 = Fiacco-McCormick	1
	`SC_CONV`	Defines the expected solution value of the objective function at the optimum, default zero (50% improvement)	0
3.7 Schwefel library Simplex search	`OPT_TOL`	The accuracy with which solutions are found	1.00E-03
	`OPT_CTOL`	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	`OPT_STEP`	The step size used	1.00E-05
	`SC_PENAL`	Selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines: 1 = one pass external 2 = Fiacco-McCormick	1
	`SC_NITERS`	The number of iterations before convergence testing is applied, default zero (the total number of function calls to be used divided by 25 times the number of design variables)	0
3.8 Schwefel library Complex search	`OPT_TOL`	The accuracy with which solutions are found	0

	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	SC_PENAL	Selects the kind of penalty function used by unconstrained search methods in the Schwefel library routines: 1 = one pass external 2 = Fiacco-McCormick	1
3.91 Schwefel library two-membered evolution strategy (EVOL)	SC_LS	How severe convergence testing is, with bigger values requiring the objective function to remain essentially stationary for longer before convergence is considered complete	2
	SC_NRANDM	The number of random numbers drawn and discarded before starting the optimiser	0
	OPT_TOL	The accuracy with which solutions are found	1.00E-03
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	SC_LR	Controls step size management, with bigger values giving a slower but more accurate search	1
	SC_SN	Controls step size adjustment, which can be kept constant using a value of unity	0.85
3.92 Schwefel library multi-membered evolution strategy (KORR)	SC_IELTER	The number of parents in a generation	10
	SC_NACHKO	The number of descendants of a generation	100
	SC_NS	The number of different step size parameters	N
	SC_DELS	The global random step sizes	1/sqtr(2N)
	SC_DELI	The local random step sizes	1/sqtr(2N)/ sqtr(NS)
	SC_DELP	The correlation ellipsoid angles	5 × 0. 01745 = 5°
	SC_BKORRL	Switches on the rotation of the correlation ellipsoid if non-zero	1
	SC_KONVKR	Number of generations used when applying convergence tests	1
	SC_NRANDM	The number of random numbers drawn and discarded before starting the optimiser	0
	OPT_TOL	The accuracy with which solutions are found	0
	OPT_CTOL	The accuracy with which constraints must be met to be considered satisfied	1.00E-03
	OPT_STEP	The step size used	1.00E-05
	SC_TYPE	Controls whether the "comma" or "plus" version of the code is used: 1 = comma 2 = plus	1
	SC_IREKOM	Controls the recombination type (n.b., each digit in this variable must lie between 1 and 5)	333

4 genetic algorithm search	GA_NBIN	The number of bits used per variable in binary discretisation	12
	GA_NPOP	Population size each generation	50
	GA_PENAL	Set the penalty function control parameter, r, with values less than one invoking the modified Fiacco and McCormick function (OPTIM2) otherwise the one pass method is used (OPTIM1)	1.00E+20
	GA_PBEST	The proportion of the solutions that are used to form the parents of the next generation	0.8
	GA_PCROSS	The proportion of the solutions in the population that are crossed to form new solutions	0.8
	GA_PINVRT	The proportion of the solutions in the population that have their ordering codes inverted to form new solutions	0.2
	GA_PMUTNT	Mutation is allowed at a level set by this parameter, i.e., this fraction of the total number of binary digits are reversed at each pass (n.b. greater than 0.5 results in randomisation)	0.005
	GA_PRPTNL	If .TRUE. the make-up of the following generation is then biased in favour of the most successful according to their objective function values, otherwise survival is proportional to ranking but scaled to prevent dominance and stagnation	1 (.TRUE.)
	GA_ALPHA	The cluster penalising function. Small values giving less severe penalties than those nearer one, and a value less than zero turning the mechanism off	0.2
	GA_DMIN	The minimum distance between cluster centroids	0.05
	GA_DMAX	The furthest distance a new solution can be from an existing cluster centroid without a new cluster being formed	0.2
	GA_NCLUST	The initial number of clusters, either in absolute terms or, if it is <1. 0, as a fraction of the population size	0.1
	GA_NBREED	Breeding is restricted to be between members of the same cluster if there are at least this many members in the cluster	0.1
	GA_PSEED	Seeding of the initial, randomly generated members of the population is allowed at a level set by this parameter (0 = random, 1.0 clones of initial point)	0
	GA_NRANDM	The number of random numbers drawn and discarded before starting the optimiser	0
5 simulated annealing	SA_NBIN	The number of bits used per variable in binary discretisation	12
	SA_PTEMP	The power to which the number of iterations must be raised to calculate the number of annealing temperatures	1/3
	SA_PWIDTH	The range of temperatures in the annealing schedule, with large values	5

		giving a wide range of temperatures, which carries the risk of rapid freezing but gives a wider ranging search	
	SA_PCOLD	The bottom temperature in the annealing schedule, with values over two giving lower temperatures and thus more accurate results at the expense of perhaps missing the global optimum	2
	SA_SCHED	If this parameter exists and contains an array of variables it is taken to be a cooling schedule which is to be used in place of the preceding three parameters	
	SA_PENAL	Sets the penalty function control parameter, r, with values less than one invoking the modified Fiacco and McCormick function (OPTIM2) otherwise the one pass method is used (OPTIM1)	1.00E+20
	SA_PMUTNT	Mutation is allowed at a level set by this parameter, i.e., this fraction of the total number of binary digits are reversed at each evaluation (setting SA_PMUTNT negative causes the mutations to be made to the actual variables rather than the binary digits)	0.1
	SA_NRANDM	The number of random numbers drawn and discarded before starting the optimiser	0
6 evolutionary programming	EP_NBIN	The number of bits used per variable in binary discretisation	12
	EP_NPOP	Population size each generation	50
	EP_PENAL	Set the penalty function control parameter, r, with values less than one invoking the modified Fiacco and McCormick function (OPTIM2) otherwise the one pass method is used (OPTIM1)	1.00E+20
	EP_IMUTNT	Mutation is controlled so that the best members are mutated least and the worst, most, this parameter governs the order of the mutation with ranking, a value of one thus gives a linear change, two a quadratic one and so on (only positive values being allowed), default two;	2
	EP_TOURN	The number of members in the ranking tournament, either in absolute terms or, if it is <1. 0, as a fraction of the population size	0.5
	EP_NRANDM	The number of random numbers drawn and discarded before starting the optimiser	0
7 evolution strategy	ES_NPPOP	The population size	100
	ES_NCPOP	The parent populations size, a fraction of the total population size	1
	ES_PENAL	Sets the penalty function control parameter, r, with values less than one invoking the modified Fiacco and McCormick function (OPTIM2)	1.00E+20

	otherwise the one pass method is used (OPTIM1)	
`ES_DELSIG`	Used to set the standard deviation of a random number whose exponential is then used to scale the previous mutation control parameter.	0.1
`ES_UCHILD`	When selecting the next generation all the children may be used or a mixture of the best children and parents used; if this parameter is non-zero it is taken to be .TRUE. and the children are used in preference to parents.	0 (false)
`ES_VDSCRT`	Controls the crossover type between parents for design variables. Either discrete crossover (.TRUE.) or intermediate crossover (.FALSE.).	1 (true)
`ES_MDSCRT`	Controls the crossover type between parents for mutation control parameters. Either discrete crossover (.TRUE.) or intermediate crossover (.FALSE.).	0 (false)
`ES_NRANDM`	The number of random numbers drawn and discarded before starting the optimiser	0

Table 1 OptionsMatlab optional control parameters

5.17 How do I deal with failed calculations when constructing a response surface model?

Failures may occur when calculating the value of an objective function during a direct search. These failures may be stochastic (perhaps due to the unexpected failure of a Grid resource), or they may be indicative of a problematic area of the parameter space (perhaps representing an unfeasible geometry). There are a couple of possible strategies to ensure that failed calculations are correctly handled by `OptionsMatlab` when constructing and searching a Response Surface Model.

The optional control parameter `OBJ_BAD_PT` may be used to define an outer bound for acceptable values of an objective function. When `OptionsMatlab` encounters objective function values exceeding `OBJ_BAD_PT` during the construction of a Response Surface Model these values will be ignored. During minimisation `OptionsMatlab` will ignore any objective function values greater than `OBJ_BAD_PT`, whereas during maximisation values less than `OBJ_BAD_PT` will be ignored.

It is possible to use `OBJ_BAD_PT` to filter stochastic failures that occur during the evaluation of the objective function. For a minimisation problem the Matlab function defining the user's objective function should return a very large value for the objective function (which exceeds expected values) upon failure. When building and searching a Response Surface Model of the objective function the `OptionsMatlab` input structure should contain the field `OBJ_BAD_PT` with a value less than that of the failed calculations. The bad points will therefore not influence the Response Surface Model of the objective function.

When a failed calculation represents a problematic area of the parameter space it is sometimes desirable to steer a design search away from these areas. To do this it is possible to define an extra constraint to indicate bad points. In this case when a calculation fails this constraint should be set to indicate an invalid point. As the design search proceeds the constraint may steer the optimiser away from these problematic areas. When searching over a Response Surface Model this strategy may be used in conjunction with `OBJ_BAD_PT`.

5.18 How do I build and evaluate a RSM faster?

There are a number of ways to make OptionsMatlab run faster when building and evaluating a Response Surface Model.

If additional output information is requested from OptionsMatlab (`OLEVEL>0`) further calculations may be performed. This may significantly increase the time taken to build and evaluate a RSM, in particular for large datasets. Therefore to perform faster searches of a RSM it may be advantageous to set `OLEVEL=0` in the OptionsMatlab input structure.

When performing multiple searches of a Stochastic Process Model (SPM), i.e. when `OBJMOD` or `CONMOD` equal to 4.1, 4.2 or 4.3, it is possible to avoid rebuilding the SPM by passing the hyper-parameters for the model in the input structure. When a SPM is first built and searched (or when the hyper-parameters are explicitly tuned, see section 5.11) the hyper-parameters are returned in the output structure fields `OBJHYPER` (and/or `CONHYPER`). By adding these fields to the OptionsMatlab input structure when subsequently searching the SPM the hyper-parameters will not be rebuilt. However, please note that it is important to rebuild the hyper-parameters following changes to dataset otherwise they may become ill-defined for your dataset.

6 OptionsMatlab Examples

6.1 DoE Direct search

Perform a DoE over the problem defined by the input structure, and then plot the results of the DoE. The results of this DoE are used to build RSM in many of the subsequent examples.

```
input1 = createBeamStruct;
input1.OMETHD = 2.8;         %Design of Experiments
input1.NITERS = 50;          %Number of iterations
input1.OLEVEL = 2;
input1.MC_TYPE = 4;          %Full factorial DoE
output1 = OptionsMatlab(input1)
```

```
output1 =


       VARS: [2x1 double]
     OBJFUN: 3.6877e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

```
%Print a digest of the optimisation and determine if
%optimum returned is valid
isvalid = optimisationDigest(output1, input1)
```

```
============================================================


 Optimisation of the problem defined by "beamobjfun" and
"beamobjcon"
 Optimisation method: 2.8

 Status after 50 evaluations is :-

 Trial vector

 Lwr Bound       Vector       Uppr Bound    Variable (units)

   5.00000000 <  24.68750000 >  50.00000000  BREADTH
   2.00000000 <  14.93750000 >  25.00000000  HEIGHT

 No of V. Boundary Violations =   0

 Objective Function (min.)    =  3687.6953   AREA



 Constraints vector

 Lwr Bound       Vector       Uppr Bound    Variable (units)

              <  81.69200669 > 200.00000000  SIGMA-B
              <   2.03379058 > 100.00000000  TAU
              <   3.78699170 >   5.00000000  DEFLN
              <   6.05063291 >  10.00000000  H-ON-B
 5000.00000000 < 290554.98816615             F-CRIT

 No of Constraint Violations =   0

============================================================
```

```
%Plot the results of the optimisation
plotOptionsSurfaces(output1, input1)
optimisationTerrain(output1, input1)
optimisationTrace(output1, input1)
```

```
optimisationHistory({output1}, {'Design of Experiments'})
```

```
isvalid =


     1
```

6.2 RSM returning update points

Build and search a Response Surface Model using the results of example 6.1. This search will return up to 10 update points where the quality of the DoE would be best improved.

```
input2 = createBeamStruct;
input2.OMETHD = 4;           %Genetic Algorithm
input2.NITERS = 50;
input2.OLEVEL = 2;
input2.OBJMOD = 3.3;         %First order polynomial regression
                             %model plus squares
input2.CONMOD = 3.3;         %First order polynomial regression
                             %model plus squares
input2.NUMUPDATE = 10;       %10 update points
output2 = OptionsMatlab(input2, output1)
```

```
output2 =


         VARS: [2x1 double]
       OBJFUN: 2.5149e+003
         CONS: [5x1 double]
    DOE_TRACE: [1x1 struct]
```

6.3 DoE evaluating candidate points

Perform a candidate point DoE search to evaluate the update points suggested by example 6.2.

```
input3 = createBeamStruct;
input3.OLEVEL = 2;
```

```
input3.OMETHD = 2.8;         %Design of Experiments
                             %Specify update points as candidate
                             %points
input3.DOE_TRACE = output2.DOE_TRACE;
                             %Set the number of iterations
input3.NITERS = output2.DOE_TRACE.NCALLS+1;
input3.MC_TYPE = 7;          %Specify that the DOE uses
                             %candidate points
                             %Note that the meaning of MC_TYPE
                             %has changed since version 0.6.5
output3 = OptionsMatlab(input3)
```

```
output3 =


       VARS: [2x1 double]
     OBJFUN: 6000
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

```
%Concatenate the output structures from examples 6.1 and 6.3
output3_cat = optimisationAppendDataPoints(output1,output3)
```

```
output3_cat =


       VARS: [2x1 double]
     OBJFUN: 3.6877e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

6.4 RSM using candidate points

Build and evaluate an RSM at specified points. The utility function `optimisationSampleRSM` can assist you to do this (see example 6.9).

```
input4 = createBeamStruct;
```

```
input4.OLEVEL = 2;
input4.OMETHD = 2.8;          %Design of Experiments
                              %Specify the candidate points to
                              %be evaluated
input4.DOE_TRACE.NCALLS = output1.OBJTRC.NCALLS;
input4.DOE_TRACE.VARS = output1.OBJTRC.VARS;
input4.NITERS = input4.DOE_TRACE.NCALLS +1;
input4.MC_TYPE = 7;           %DoE using candidate points
input4.OBJMOD = 3.3;          %First order polynomial
                              %regression model plus squares
input4.CONMOD = 3.3;          %First order polynomial
                              %regression model plus squares
output4 = OptionsMatlab(input4, output3_cat)
```

```
output4 =


      VARS: [2x1 double]
    OBJFUN: 2.6319e+003
      CONS: [5x1 double]
    RSMTRC: [1x1 struct]
```

```
%Plot the RSM
fig = optimisationTerrain(output4, input4, 2);
optimisationTrace(output4, input4, 2, fig);
```

6.5 Direct search with checkpointing

Checkpoint the search history of a direct search every 300 generations in a file 'optimTest5.mat'.

```
input5 = createBeamStruct;
input5.OLEVEL = 2;
input5.OMETHD = 2.8;
input5.NITERS = 500;    %500 iterations
input5.MAXJOBS = 100;   %Submit jobs in groups of 100
input5.CHKPT_INTV = 300;%Checkpoint every 300 generations
input5.CHKPT_FILE = 'optimTest5.mat'; %Checkpoint file name
delete('optimTest5.mat')%Remove existing checkpoint file
```

```
output5 = OptionsMatlab(input5)
```

```
output5 =


       VARS: [2x1 double]
     OBJFUN: 2.9455e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

```
load('optimTest5.mat')  %Load checkpoint file
whos CHKPOINT
```

```
  Name           Size                        Bytes  Class


  CHKPOINT       1x1                         49256  struct array


Grand total is 6012 elements using 49256 bytes
```

6.6 Parallel job submission with userdata

This example uses the Geodise compute toolbox [3] that provides client functionality to Globus Grid resources that may be used to evaluate computational jobs. The jobs will be submitted to the Globus resource to run concurrently. When the jobs are complete the results will be retrieved and parsed to determine the objective function values. Note that you must have the Geodise compute toolbox installed, and have valid credentials with permissions to submit jobs to the specified compute resource.

```
%Define the Globus server to which to submit the jobs
GLOBUSSERVER = 'escience-dept2.sesnet.soton.ac.uk';

gd_createproxy
```

```
Paused: Press any key...
```

```
input6 = createBeamStructParallel;
input6.OLEVEL = 0;
```

```
input6.MAXJOBS = 10;     %The number of the jobs to be run
                         %concurrently
input6.NITERS = 20;      %The number of iterations

                         %USERDATA field is used to pass the
                         %host name upon which to run the
                         %objective function to the Matlab
                         %function
input6.USERDATA.hostname = GLOBUSSERVER;
output6 = OptionsMatlab(input6)
```

```
[...]

ohandle =

https://escience-
dept2.sesnet.soton.ac.uk:30040/10303/1134728028/



uniquedir =

20051216T101347_57891/



EVAL =

  3.9666e+003

output6 =

       VARS: [2x1 double]
     OBJFUN: 3.9666e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

6.7 Hyper-parameter tuning

This example will tune hyper-parameters for a Stochastic Process Model over the results of the DoE produced in example 6.1. The hyper-parameters will be tuned using the optimisation algorithm specified by `OMETHD`. The tuned hyper-parameters will be returned in fields of the output structure `OBJHYPER` (or `CONHYPER`) that be supplied in the input structure when building a Stochastic Process Model RSM.

```
input7 = createBeamStruct;
input7.OLEVEL = 0;
input7.OBJMOD = 4.1;     %Tune stochastic Process Model
                         %hyper-parameters over the
                         %objective function
input7.CONMOD = 4.1;     %Tune stochastic Process Model
                         %hyper-parameters over the
                         %constraints
input7.TUNEHYPER = 1;    %Tune the hyper-parameters (do not
                         %search the user's problem)
input7.OMETHD = 5;       %Simulated Annealing

                         %Note that if OBJHYPER or CONHYPER are
                         %provided these hyper-parameters will
                         %be used in preference to those
                         %generated by OPTRSS
output7 = OptionsMatlab(input7, output1)
```

```
output7 =


    OBJHYPER: [1x1 struct]
     OBJ_CLF: 712.6938
    CONHYPER: [1x1 struct]
     CST_CLF: 824.2750
```

6.8 User-defined sequential optimiser

This example invokes the sequential optimiser defined by the Matlab function `'optum1.m'`, which randomly generates searches points within the parameters space. It is possible to write a Matlab function that provides alternative behaviour for a sequential optimiser.

```
input8 = createBeamStruct;
input8.OLEVEL = 2;
input8.OMETHD = 2.1;      %User-defined optimiser 1
input8.OPTUM1 = 'optum1'; %Specifies function 'optum1.m' as
                          %user-defined optimiser
output8 = OptionsMatlab(input8)
```

```
output8 =


       VARS: [2x1 double]
     OBJFUN: 2.6409e+003
       CONS: [5x1 double]
     OBJTRC: [1x1 struct]
    CONSTRC: [1x1 struct]
```

6.9 Sample a Response Surface Model

This example uses the utility function `optimisationSampleRSM` to build an RSM and sample the RSM at 100 evenly spaced points within the parameter space. Compare this method to example 6.4.

```
%Create an input structure to search an RSM
input9 = createBeamStruct;
input9.OLEVEL = 2;
input9.OBJMOD = 3.3;
input9.CONMOD = 3.3;

%Sample 100 evenly spaced points
output9 = optimisationSampleRSM(input9, output1, 100)

%Plot the points sampled from the RSM
optimisationTerrain(output9, input9)
```

```
output9 =


      VARS: [2x1 double]
    OBJFUN: 2.4349e+003
      CONS: [5x1 double]
    RSMTRC: [1x1 struct]
```

6.10 Build a stochastic process model RSM with quick tuning

This example builds a stochastic process model RSM using quick hyper-parameter tuning (by setting the flag `RSM_QCK_HP`). Here the hyper-parameters `THETA` and `EXP` will be tuned across all design variables, rather than `NVARS` values of `THETA` and `EXP` corresponding to each design variable. The values of `OBJ_EXP` and `OBJ_THETA`, and of `CST_EXP` and `CST_THETA` (in the structures `OBJHYPER` and `CONHYPER`) will be scalar, rather than a vector of length `NVARS`.

```
%Create an input structure to search an SPM RSM with quick
tuning
input10 = createBeamStruct;
input10.OLEVEL = 0;
input10.OBJMOD = 4.1;
input10.CONMOD = 4.1;
input10.RSM_QCK_HP = 1;

output10 = OptionsMatlab(input10, output1);
output10.OBJHYPER
output10.CONHYPER
```

```
ans =
      OBJ_LAMBDA: -6
    U_OBJ_LAMBDA: 3
    L_OBJ_LAMBDA: -20
       OBJ_THETA: 0.1548
     U_OBJ_THETA: 3
     L_OBJ_THETA: -10
         OBJ_EXP: 2
       U_OBJ_EXP: 2
       L_OBJ_EXP: 1

ans =
      CST_LAMBDA: -6
    U_CST_LAMBDA: 3
    L_CST_LAMBDA: -20
       CST_THETA: 0.1563
     U_CST_THETA: 3
     L_CST_THETA: -10
         CST_EXP: 2
       U_CST_EXP: 2
       L_CST_EXP: 1
```

6.11 Search a tuned stochastic process model RSM

This example samples and then searches the stochastic process model RSM built using the quick tuned hyper-parameters. The scalar hyper-parameter values `OBJ_THETA` and `OBJ_EXP` are duplicated across the design variables of the problem and assigned to the field `OBJHYPER` of the input structure.

```
% Duplicate the scalar hyperpameter values across the design
variables
inputStruct = createBeamStruct;
inputStruct.OBJHYPER.OBJ_THETA  =
output10.OBJHYPER.OBJ_THETA*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_EXP    =
output10.OBJHYPER.OBJ_EXP*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_LAMBDA = output10.OBJHYPER.OBJ_LAMBDA;
```

```
%Sample the RSM surface
input11a = inputStruct;
input11a.OLEVEL = 0;
input11a.OBJMOD = 4.1;  %Evaluate SPM RSM objective function
input11a.CONMOD = 0.0;  %Evaluate constraint function directly
output11a = optimisationSampleRSM(input11a, output1, 400);

%Create an input structure to search the SPM RSM using a GA
input11b = input11a;    %Copy the sampling input structure
input11b.OMETHD = 4;    %Genetic Algorithm
input11b.NITERS = 500   %10 generations
output11b = OptionsMatlab(input11b, output1);
output11b = optimisationSearchTrace(output11b)  %Retrieve
optimum from the trace history
```

```
output11b = 


        VARS: [2x1 double]
      OBJFUN: 2.6948e+003
        CONS: [5x1 double]
      RSMTRC: [1x1 struct]
    OBJHYPER: [1x1 struct]
```

```
%Plot the RSM and optimum point
optimisationTerrain(output11a, input11a, 5)
hold on;
plot3(output11b.VARS(1,1), output11b.VARS(2,1), ...
      output11b.OBJFUN, 'ko', 'MarkerFaceColor', 'k')
```

6.12 Search the root mean square error of a tuned stochastic process model RSM

This example samples and then searches the Root Mean Square Error of the stochastic process model RSM built using the quick tuned hyper-parameters. The scalar hyper-parameter values `OBJ_THETA` and `OBJ_EXP` are duplicated across the design variables of the problem and assigned to the field `OBJHYPER` of the input structure.

The RMSE surface is invariant to a change in the direction of search for the underlying problem. This means that the surface can be searched in either direction for points of maximum or minimum error. The test first verifies that the RSM is identical when the direction of search is reversed.

The reader will be aware that the root mean square error of the SPM falls to zero at all sampled points (since the values of the objective and constraints are known at these points) so searching for the minimum of the surface is of little value. To find the maximum error in the stochastic process model RSM the direction of search in the input field `DIRCTN` is always set to +1 regardless of the direction of search of the underlying problem. This is worth highlighting because this differs from the searches of the other stochastic process model properties. In the cases of expected improvement (`OBJMOD=4.3`), constrained expected improvement (`OBJMOD=4.31`), constrained feasibility of improvement (`OBJMOD=4.32`) and probability of improvement (`OBJMOD=4.33`) the RSM surface that is built is critically dependent on the direction of search of the underlying problem. Any searches of these surfaces are hard-coded within OPTIONS to build the surface according to the direction of search for the underlying problem and seek the maximum in that surface accordingly. Only in the case of RMSE must the direction of search be explicitly set to +1 to find the maximum in the root mean square error of the RSM.

```
% Duplicate the scalar hyperpameter values across the design
variables
inputStruct = createBeamStruct;
inputStruct.OBJHYPER.OBJ_THETA  =
output10.OBJHYPER.OBJ_THETA*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_EXP    =
output10.OBJHYPER.OBJ_EXP*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_LAMBDA = output10.OBJHYPER.OBJ_LAMBDA;

%Sample the RSM surface
input12a = inputStruct;
input12a.OLEVEL = 0;
input12a.OBJMOD = 4.2;  %Evaluate RMSE of SPM RSM over
objective function
input12a.CONMOD = 0.0;  %Evaluate constraint function directly
input12a.ONAM = 'RMSE'; %Label objective
input12a.DIRCTN = +1;   %The error surface should not change
with DIRCTN
```

```
output12a = optimisationSampleRSM(input12a, output1, 400);
      %Sample RSM

input12b = input12a;    %Copy the sampling input structure
input12b.DIRCTN = -1;   %The error surface should not change
with DIRCTN
output12b = optimisationSampleRSM(input12b, output1, 400);

%Check that the RMSE surface is invariant under change of
DIRCTN
if (sum(abs(output12a.RSMTRC.OBJFUN - output12b.RSMTRC.OBJFUN))
> 0)
    error('*** RMSE of Stochastic Process Model is not
invariant under change of DIRCTN ***')
end

input12c = input12b;    %Copy the sampling input structure
input12c.DIRCTN = +1;   %Search for maximum in RMSE of the SPM
                        %(NB. This value is set to +1
regardless of the
                        % direction of the underlying problem)
input12c.OMETHD = 4;    %Genetic Algorithm
input12c.NITERS = 500   %10 generations
output12c = OptionsMatlab(input12c, output1);
output12c = optimisationSearchTrace(output12c)  % Search the
trace history for optimum
```

```
output12c =


        VARS: [2x1 double]
      OBJFUN: 746.8510
        CONS: [5x1 double]
      RSMTRC: [1x1 struct]
    OBJHYPER: [1x1 struct]
```

```
%Plot the RSM and optimum point
optimisationTerrain(output12a, input12a, 5)
hold on;
```

```
plot3(output12c.VARS(1,1), output12c.VARS(2,1), ...
      output12c.OBJFUN, 'ko', 'MarkerFaceColor', 'k')
```

6.13 Search the expected improvement of a tuned stochastic process model RSM

This example samples and then searches the Expected Improvement of the stochastic process model RSM built using the quick tuned hyper-parameters. The scalar hyper-parameter values `OBJ_THETA` and `OBJ_EXP` are duplicated across the design variables of the problem and assigned to the field `OBJHYPER` of the input structure. Note that for a minimisation problem OPTIONS inverts the Expected Improvement calculation, returning a minimum value of the inverted problem, at the point of maximum expected improvement of the RSM.

```
% Duplicate the scalar hyperpameter values across the design
variables
inputStruct = createBeamStruct;
inputStruct.OBJHYPER.OBJ_THETA  =
output10.OBJHYPER.OBJ_THETA*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_EXP    =
output10.OBJHYPER.OBJ_EXP*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_LAMBDA = output10.OBJHYPER.OBJ_LAMBDA;

%Sample the RSM surface
input13a = inputStruct;
input13a.OLEVEL = 0;
input13a.OBJMOD = 4.3;  %Evaluate EI of SPM RSM over objective
function
input13a.CONMOD = 0.0;  %Evaluate constraint function directly
input13a.ONAM = 'EI';   %Label objective
output13a = optimisationSampleRSM(input13a, output1, 400);
      %Sample RSM

%Create an input structure to search the SPM RSM using a GA
input13b = input13a;    %Copy the sampling input structure
input13b.OMETHD = 4;    %Genetic Algorithm
input13b.NITERS = 500   %10 generations
output13b = OptionsMatlab(input13b, output1);
```

```
output13b = optimisationSearchTrace(output13b)  %Search the
trace history for optimum
```

```
output13b =


        VARS: [2x1 double]
      OBJFUN: 115.1293
        CONS: [5x1 double]
      RSMTRC: [1x1 struct]
    OBJHYPER: [1x1 struct]
```

```
%Plot the RSM and optimum point
optimisationTerrain(output13a, input13a, 5)
hold on;
plot3(output13b.VARS(1,1), output13b.VARS(2,1), ...
      output13b.OBJFUN, 'ko', 'MarkerFaceColor', 'k')
```

6.14 Search the probability of improvement of a tuned stochastic process model RSM

This example samples and then searches the Probability of Improvement of the stochastic process model RSM built using the quick tuned hyper-parameters. The scalar hyper-parameter values `OBJ_THETA` and `OBJ_EXP` are duplicated across the design variables of the problem and assigned to the field `OBJHYPER` of the input structure. Note that for a minimisation problem OPTIONS inverts the Probability of Improvement calculation, returning a minimum value of the inverted problem, at the point of maximum probability of improvement of the RSM (this is why this calculation may return negative value for the probability when searching a minimisation problem).

```
% Duplicate the scalar hyperpameter values across the design
variables
inputStruct = createBeamStruct;
inputStruct.OBJHYPER.OBJ_THETA  =
output10.OBJHYPER.OBJ_THETA*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_EXP    =
output10.OBJHYPER.OBJ_EXP*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_LAMBDA = output10.OBJHYPER.OBJ_LAMBDA;
```

```
%Sample the RSM surface
input14a = inputStruct;
input14a.OLEVEL = 0;
input14a.OBJMOD = 4.33; %Evaluate PI of SPM RSM over objective
function
input14a.CONMOD = 0.0;  %Evaluate constraint function directly
input14a.ONAM = 'PI';   %Label objective
output14a = optimisationSampleRSM(input14a, output1, 400);
      %Sample RSM

%Create an input structure to search the SPM RSM using a GA
input14b = input14a;    %Copy the sampling input structure
input14b.OMETHD = 4;    %Genetic Algorithm
input14b.NITERS = 500   %10 generations
output14b = OptionsMatlab(input14b, output1);
output14b = optimisationSearchTrace(output14b)  %Search the
trace history for optimum
```

```
output14b = 


        VARS: [2x1 double]
      OBJFUN: -1.0776e-042
        CONS: [5x1 double]
      RSMTRC: [1x1 struct]
    OBJHYPER: [1x1 struct]
```

```
%Plot the RSM and optimum point
optimisationTerrain(output14a, input14a, 5)
hold on;
plot3(output14b.VARS(1,1), output14b.VARS(2,1), ...
      output14b.OBJFUN, 'ko', 'MarkerFaceColor', 'k')
```

6.15 Search the constrained expected improvement of a tuned stochastic process model RSM

This example samples and then searches the constrained Expected Improvement of the stochastic process model RSM built using the quick tuned hyper-parameters. The scalar hyper-parameter values `OBJ_THETA`, `OBJ_EXP`, `CST_THETA` and `CST_EXP` are duplicated across the design variables of the problem and assigned to the fields `OBJHYPER` and `CONHYPER` of the input structure. Note that for a minimisation problem OPTIONS inverts the constrained Expected Improvement calculation, returning a minimum value of the inverted problem, at the point of maximum expected improvement of the constrained RSM.

```
% Duplicate the scalar hyperpameter values across the design
variables
inputStruct = createBeamStruct;
inputStruct.OBJHYPER.OBJ_THETA  =
output10.OBJHYPER.OBJ_THETA*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_EXP    =
output10.OBJHYPER.OBJ_EXP*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_LAMBDA = output10.OBJHYPER.OBJ_LAMBDA;
inputStruct.CONHYPER.CST_THETA  =
output10.CONHYPER.CST_THETA*ones(inputStruct.NVRS,1);
inputStruct.CONHYPER.CST_EXP    =
output10.CONHYPER.CST_EXP*ones(inputStruct.NVRS,1);
inputStruct.CONHYPER.CST_LAMBDA = output10.CONHYPER.CST_LAMBDA;

%Sample the RSM surface
input15a = inputStruct;
input15a.OLEVEL = 0;
input15a.OBJMOD = 4.31;       %Evaluate constrained EI of SPM
RSM over objective function
input15a.CONMOD = 4.1;        %Evaluate constraint function
using SPM RSM
input15a.ONAM = 'CST-EI';     %Label objective
output15a = optimisationSampleRSM(input15a, output1, 400);
      %Sample RSM

%Create an input structure to search the SPM RSM using a GA
input15b = input15a;    %Copy the sampling input structure
input15b.OMETHD = 4;    %Genetic Algorithm
input15b.NITERS = 500   %10 generations
output15b = OptionsMatlab(input15b, output1);
output15b = optimisationSearchTrace(output15b)  %Search the
```

```
trace history for optimum
```

```
output15b =


        VARS: [2x1 double]
      OBJFUN: -7.5469
        CONS: [5x1 double]
      RSMTRC: [1x1 struct]
    OBJHYPER: [1x1 struct]
    CONHYPER: [1x1 struct]
```

```
%Plot the RSM and optimum point
optimisationTerrain(output15a, input15a, 5)
hold on;
plot3(output15b.VARS(1,1), output15b.VARS(2,1), ...
      output15b.OBJFUN, 'ko', 'MarkerFaceColor', 'k')
```

6.16 Search the constrained feasibility of improvement of a tuned stochastic process model RSM

This example samples and then searches the constrained Feasibility of Improvement of the stochastic process model RSM built using the quick tuned hyper-parameters. The scalar hyper-parameter values `OBJ_THETA`, `OBJ_EXP`, `CST_THETA` and `CST_EXP` are duplicated across the design variables of the problem and assigned to the fields `OBJHYPER` and `CONHYPER` of the input structure. Note that for a minimisation problem OPTIONS inverts the constrained Feasibility of Improvement calculation, returning a minimum value of the inverted problem, at the point of maximum feasibility of improvement of the constrained RSM.

```
% Duplicate the scalar hyperpameter values across the design
variables
inputStruct = createBeamStruct;
inputStruct.OBJHYPER.OBJ_THETA  =
output10.OBJHYPER.OBJ_THETA*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_EXP    =
output10.OBJHYPER.OBJ_EXP*ones(inputStruct.NVRS,1);
inputStruct.OBJHYPER.OBJ_LAMBDA = output10.OBJHYPER.OBJ_LAMBDA;
inputStruct.CONHYPER.CST_THETA  =
output10.CONHYPER.CST_THETA*ones(inputStruct.NVRS,1);
inputStruct.CONHYPER.CST_EXP    =
output10.CONHYPER.CST_EXP*ones(inputStruct.NVRS,1);
inputStruct.CONHYPER.CST_LAMBDA = output10.CONHYPER.CST_LAMBDA;

%Sample the RSM surface
input16a = inputStruct;
input16a.OLEVEL = 0;
input16a.OBJMOD = 4.32;       %Evaluate constrained FI of SPM
RSM over objective function
input16a.CONMOD = 4.1;        %Evaluate constraint function
using SPM RSM
input16a.ONAM = 'CST-FI';     %Label objective
output16a = optimisationSampleRSM(input16a, output1, 400);
      %Sample RSM

%Create an input structure to search the SPM RSM using a GA
input16b = input16a;    %Copy the sampling input structure
input16b.OMETHD = 4;    %Genetic Algorithm
input16b.NITERS = 500   %10 generations
output16b = OptionsMatlab(input16b, output1);
output16b = optimisationSearchTrace(output16b)  %Search the
trace history for optimum
```

```
output16b =

        VARS: [2x1 double]
      OBJFUN: 0
        CONS: [5x1 double]
      RSMTRC: [1x1 struct]
    OBJHYPER: [1x1 struct]
    CONHYPER: [1x1 struct]
```

```
%Plot the RSM and optimum point
optimisationTerrain(output16a, input16a, 5)
hold on;
plot3(output16b.VARS(1,1), output16b.VARS(2,1), ...
      output16b.OBJFUN, 'ko', 'MarkerFaceColor', 'k')
```

7 References

[1] OPTIONS Design Exploration System: http://www.soton.ac.uk/~ajk/

[2] Matlab 6.5: http://www.mathworks.com/

[3] Geodise Project: http://www.geodise.org/

[4] Matlab Optimization Toolbox:
http://www.mathworks.com/products/optimization/

[5] Globus Project; GRAM: http://www.globus.org/gram/