Data Collaboration Toolbox v2.0

July 14, 2010

The Data Collaboration toolbox is a free, open-source software package for the MATLAB programming environment for non-stochastic uncertainty quantification using predictive models and experiment data. The software is available at http://collab-sci.sourceforge.net/

The software allows a user to enter experiment data with uncertainty along with a corresponding parameterized model. These combine to constrain a set of feasible parameter values that match the data and model. Questions about this feasible set of parameters, such as whether it is nonempty (consistency) and what the range of values a given predictive model can take over the set (response prediction), are posed as constrained optimization problems.

The techniques utilized in Data Collaboration (DC) are those described in several journal articles.

1. M. Frenklach, A. Packard, and P. Seiler. Prediction uncertainty from models and data. In Proc. American Control Conference, pages 4135–4140, 2002.

2. M. Frenklach, A. Packard, P. Seiler, and R. Feeley. Collaborative data processing in developing predictive models of complex reaction systems. International Journal of Chemical Kinetics, 36(1):57–66, 2004.

3. R. Feeley, P. Seiler, A. Packard, and M. Frenklach. Consistency of a reaction dataset. Journal of Physical Chemistry A, 108(44):9573–9583, October 2004.

4. R. Feeley, M. Frenklach, M. Onsum, T. Russi, A. Arkin, and A. Packard. Model discrimination using data collaboration. Journal of Physical Chemistry A, 110(21):6803–6813, March 2006.

5. P. Seiler, M. Frenklach, A. Packard, and R. Feeley. Numerical approaches for collaborative data processing. Optimization and Engineering, 7(4):459–478, December 2006.

6. T. Russi, A. Packard, R. Feeley, and M. Frenklach. Sensitivity analysis of uncertainty in model prediction. Journal of Physical Chemistry A, 112(12):2579–2588, February 2008.

This document is designed to act both as a manual for the use of the toolbox, as well as providing some high-level description of the underlying algorithms. Each section describes

either the creation of some objects or an algorithm. If you feel you would like to jump right in, we suggest you first read the requirements and downloading/installation instructions in Sections 1 and 2. After that skip to the examples in Section 11.

1 Requirements

- **"Any" computer operating system**: DC has been tested on machines running Windows XP, Windows Vista x86 (32-bit), Windows Vista x64 (64-bit), and Ubuntu. In principle, it should run on any fairly new desktop or laptop computer that can run MATLAB.
- **MATLAB release 2008a or later**
- **Optimization toolbox for MATLAB**
- **SeDuMi optimization package**: SeDuMi is a free MATLAB toolbox for self-dual minimization. DC provides version 1.21, but you may use another version if you like. DC has been tested with SeDuMi versions 1.1 and 1.21.

2 Download and Installation

To download the software,

1. Visit the project home page at http://collab-sci.sourceforge.net/
2. Click the "Download Page" link.
3. Click the large green button that says "Download Now!"
4. Depending on your internet browser's preferences, you will either be prompted to choose a location, or the zip-file will be saved to your default download folder.

Once the download is complete, unzip the file to a convenient location (e.g. your default MATLAB working directory). To unzip a file you need a zip decompression program. For windows, we recommend the free utility, 7zip, available at http://www.7zip.com/. For linux/unix, there may already be a utility installed called gzip.

Inside the main folder are several subdirectories and a few files. One of these files is named "DCsetup.m," a MATLAB script for setting path directories and importing the main package. From the MATLAB command prompt run the script:

```
>> DCsetup
```

This will add the necessary directories to the current MATLAB path. It will also import the `+DClab` package. This setup script should be executed from the main DC directory each time you start MATLAB if you wish to use the DC toolbox. This script can be executed from a startup file.

All of the main DC methods and objects have the prefix `DClab.`, but this import will make it such that using this prefix will be unnecessary. If you wish to use DC objects and methods inside of a function or custom class, you either need to include the `DClab.` prefix, or use the command

```
import DClab.*
```

at the top of each file. This command is in the "DCsetup" file, but that only will import the package for the base workspace. For more information, see MATLAB documentation on packages. Throughout this documentation, we will not use the prefix for the sake of simplicity.

To test the directory setup, run the test script from the MATLAB command prompt

```
>> dctest
```

This will take a minute or two. One issue you may run into the first time you run the package is that the SeDuMi mex-files may need to be recompiled. This is only necessary one time.

If the test script gives an error regarding SeDuMi, change the current directory in MATLAB to the "SeDuMi_1_21" directory in the DC toolbox main directory. From the command prompt type

```
>> mex -setup
```

and select a c-compiler of your choice. For 32-bit operating systems, MATLAB ships with the free compiler lcc. If no compiler is shown, you may need to get a c-compiler. Please consult the MATLAB documentation for supported c-compilers. Once the compiler is chosen, type the following at the command prompt

```
>> install_sedumi
```

This will take a few minutes. Once complete, change back to the main DC directory and re-run dctest. Note: if the test script worked the first time, you should not have to reinstall sedumi.

3 Version 2.0

The software has changed significantly from the previous release (1.0) to this version (2.0). Most notably almost all of the object names has changed. Section 12 provides a table to help you convert your own code to the names.

There are many changes to the underlying methods, and some ways that you can interact with the software (e.g. the ability to define multiple features from a single simulation file). All these are described in this document.

4 Dataset Formulation

The Data Collaboration Dataset is a collection of parameterized models, associated experiment observations, and constraints on the individual parameters. Let Y_e be some scalar observable. Model M_e predicts the value of the observable as a function of n unknown parameters. Each parameter x_i has a set of prior knowledge bounds α_i and β_i such that $\alpha_i \leq x_i \leq \beta_i$. Also associated with each observable is an experimental measurement d_e with lower and upper bound uncertainty l_e and u_e such that the value of the observable y_e is bounded as $l_e \leq y_e - d_e \leq u_e$. All this information collected together for m observables $(e = 1, \ldots, m)$ is known as a dataset.

The following few sections describe how to build up these objects in the DC software.

4.1 Listing of FreeParameters

The FreeParameter class describes objects associated with the model parameters.

```
>> fp = FreeParameter(NAME,RANGE);
```

creates a FreeParameter object. `NAME` is a char array denoted the parameter's name (each parameter should have a unique name) and `RANGE` is a 1x2 numeric array denoted the lower and upper limits of the parameters value.

```
>> fp = FreeParameter(NAME,NOMINAL,UNCERTAINTY);
```

creates a FreeParameter object, using a nominal value with uncertainty. `NOMINAL` is a scalar value. `UNCERTAINTY` can be a scalar such that the parameter is in the range `NOMINAL+/-UNCERTAINTY`, or it can be a 1x2 numeric array so that the parameter is in the range `[NOMINAL+UNCERTAINTY(1), NOMINAL+UNCERTAINTY(2)]`. Uncertainty can also be specified relatively or with log transformations. The behavior of these types of uncertainty specification are not as stable, however. If you are interested, read the command line help for the FreeParameter object.

The FreeParameter objects can be stacked into vertical arrays. If you would like to create the FreeParameters using a for-loop, you can pre-allocate empty FreeParameter arrays. For example,

```
>> fps = FreeParameters(5);
```

initializes a 5x1 FreeParameter object array. This can then be filled using one of the above syntaxes. For example,

```
>> fps(1) = FreeParameter('param1',[3 10]);
```

4.2 ResponseObservations

The observation of some scalar value of interest (the observable) related to an experimental response is formulated as a ResponseObservation object.

```
>> ro = ResponseObservation(VALUE,UNCERTAINTY);
```

VALUE is a numeric scalar representing the measured observation. UNCERTAINTY can either be a scalar or a 1x2 double. If it is a scalar, then the assertion is that the observable y is constrained as | y - VALUE | $\leq$ UNCERTAINTY. If UNCERTAINTY is a 1x2 double then the assertion is that UNCERTAINTY(1) $\leq$ y - VALUE $\leq$ UNCERTAINTY(2). UNCERTAINTY can also be denoted as relative or with log transformations, but as with the FreeParameter object, we suggest you use the above syntax.

4.3 ResponseModels

A response model is an algebraic function or simulation code that maps a vector of fixed parameter values to the scalar observable of the simulation response. For example, if the model simulates some response over time, the scalar observable may be the maximum value, or the rise-time to the peak value. Each model represents a different observable. The model does not necessarily depend on all FreeParameters, but it does depend on a subset of them.

There are two main types of ResponseModel objects that can be created. The first is an algebraic model; specifically, it is an affine or quadratic function of the parameters. The second is a general simulation model. Their creation is outlined in the next two sections.

4.3.1 Linear or Quadratic Models

```
>> rm = ResponseModel(COEFFS,MODELDOMAIN);
```

creates an algebraic ResponseModel object.

MODELDOMAIN is an Nx1 struct array with two fields, name and range. N is the number of parameters that this model depends on. For the i^{th} element of MODELDOMAIN, .name is a char array, that matches the name of one of the FreeParameter objects and .range is the range of this parameter over which the model is valid. The range must be a superset of the range define in the corresponding FreeParameter object. The MODELDOMAIN may only contain parameters described in the FreeParameter objects. The order does not need to correspond to the order of the FreeParameter array in any way. The order of parameters in MODELDOMAIN is the order in which the ResponseModel expects parameters for calculation.

If the ResponseModel is affine, COEFFS will be an (N+1)x1 numeric array, such that the model corresponds to the form

```
y = COEFFS'*[1;x];
```

where x is a Nx1 vector of parameters in the same order as MODELDOMAIN.

If the ResponseModel is quadratic, COEFFS will be an (N+1)x(N+1) symmetric numeric array, such that the model corresponds to the form

```
y = [1 x']*COEFFS*[1;x];
```

where x is a Nx1 vector of parameters in the same order as MODELDOMAIN. This is the same as

```
y = COEFFS(1,1) + 2*COEFFS(1,2:end)*x + x'*COEFFS(2:end,2:end)*x;
```

The ResponseModel can also be specified with output uncertainty/error as

```
>> rm = ResponseModel(COEFFS,MODELDOMAIN,OUTPUTUNC);
```

If the algebraic form is a surrogate fit, then `OUTPUTUNC` may be the fitting error. For example, this would be the peak fitting error over the model domain when the algebraic model is a low-order approximation to a more detailed model. `OUTPUTUNC` should be a scalar if the uncertainty in the model output is symmetric. If it is asymmetric this should be a 1x2 vector, with `OUTPUTUNC(1) <= eta(x) - M(x) <= OUTPUTUNC(2)`, where `M(x)` is the algebraic model and `eta(x)` is the true model.

4.3.2 Models Using Simulation m-files

```
rm = ResponseModel(MODELHANDLE);
```

creates a response model given the function handle `MODELHANDLE`. It is also possible to create a ResponseModel object with some additional inputs.

```
rm = ResponseModel(MODELHANDLE,ADDLINPUT1,...,ADDLINPUT2);
```

The function that is pointed to by `MODELHANDLE` has a special form. The easiest way to learn this form is to use the template file "dcModel.m" found in the "fileTemplates" subdirectory of the main DC directory. If `MODELHANDLE` is `@myModel`, then the first line of "myModel.m" is

```
function out = myModel(flag,paramMatrix,varargin)
```

or something similar. This function not only needs to return evaluations of a set of parameter values, but also return information about the function. This first input, `flag`, will be a char array denoting the information requested by the toolbox's methods. The following list summarizes possible inputs for `flag`, and what the corresponding output needs to be.

`'simulate'` (Required) The input `paramMatrix` will be an NxL matrix (N is the number of parameters for the model and L the number of points) where each column represents one parameter vector. The output should be a 1xL array of values.

`'getModelDomain'` (Required) This should return a Nx1 structure with fields `.name` and `.range` corresponding to the parameters used by the model in the order it expects them. See the description above for the case when the ResponseModel is created with an algebraic form.

`'isSavedEnabled'` (Optional, default=false) This should return either true or false. If true, evaluations of this model are saved in the "savedEvaluations" subdirectory of the main DC directory.

`'isMultipleResponsesEnabled'` (Optional, default=false) This should return either true or false. If true, then this single file returns multiple responses (or features) for each simulation. If there are p responses then the output in the `'simulate'` case should have dimension pxL. Additionally, a response list must be defined as in the next flag. We very highly recommend that this feature only be used when `'isSavedEnabled'` is set to true.

`'getResponseList'` (Required if `'isMultipleReponsesEnabled'` is true) This case returns a px1 cell array chars. Each entry corresponds to the one-word name of one of the features. This should be in the same order as returned by the `'simulate'` case.

`'getName'` (Optional, default=filename) If the model can be different depending on the additional inputs then this lets you define a character array (string) that will be used to name files when saving points. Note: if `'isMultipleResponseEnabled'` is true this should NOT reflect a specific feature. Feature names are handled automatically. This name can depend on the other input arguments to this function as they are passed in when getting this string.

There are a few other possible flags, that are optional, and described in the template file. You do not need to provide output for the optional flags, the toolbox will catch any errors when requesting this information, and assume defaults. The additional inputs are passed in to the function each time, so they may be used to determine the model domain structure, or name (for example).

When multiple features is enabled, each feature gets its own ResponseModel object, even though they share an m-file. In this case, the first additional input will be the char array specifying the feature. For example, imagine a model simulated by "car.m" based on various driving and road conditions. It has two features we're interested in, `'top_speed'` and `'mileage'`. Then the two response models would be created via

```
>> rm1 = ResponseModel(@car,'top_speed');
>> rm2 = ResponseModel(@car,'mileage');
```

How the m-file simulates the model is entirely up to you. It can run basic MATLAB code. It can call other files. If you have the means for MATLAB to call another language (e.g. C-code via MATLAB's mex interface), the simulation can be in that other language. You can even use it to parallelize the simulation at the various parameter vectors. So all this m-file is, is a MATLAB wrapper file for whatever simulation code you would like to use.

4.4 DCDataset

The last few steps in creating a dataset, is to pair up the ResponseObservation objects with their corresponding ResponseModel objects and putting everything together with the FreeParameters.

First, for each observable, create a ModelAndObservationPair object.

```
>> mop = ModelAndObservationPair(RO,RM,NAME);
```

where `RO` is a ResponseObservation object, `RM` is a ResponseModel object, and `NAME` is a char array. These objects should be vertically concatenated into a ModelAndObservationPair array. If these objects are being created in a for-loop, you can pre-allocate the array using the syntax

```
>> mops = ModelAndObservationPair(m);
```

which will create an mx1 array.

Once an mx1 ModelAndObservationPair array `mops` and a nx1 FreeParameter array `fps` have been created, a DCDataset object can be formed.

```
>> dset = DCDataset(mops,fps);
```

This object is the basis for all further analysis.

To create a dataset with no model/experiment data, simply create an empty ModelAndObservationPair by using the constructor with no inputs.

```
>> dset = DCDataset(ModelAndObservationPair,fps);
```

5 Surrogate Model Formulation

During the calculation of the consistency measure and response prediction (see §6) and parameter optimization (see §7), for each ResponseModel that is not quadratic (or affine) the DC toolbox generates a quadratic surrogate model. This happens automatically. The subdivision of the prior knowledge parameter domain will be discussed in Section 8. This section discusses how a quadratic is fit for each ResponseModel on each subdomain.

Surrogate models and all fitting information is stored in a PolyDataset object, which inherits from the DCDataset class.

5.1 Active Parameters and Parameter Transformations

Before the surrogate fit is made, some evaluations of the ResponseModel are made to help determine "active" parameters. These are the parameters which are most essential to the change in the ResponseModel's output. Those parameters that are considered not active, are set at their nominal values and then ignored (the surrogate fit will not depend on them). This can reduce fitting time, as less evaluations are needed to make a fit, but will potentially add fitting error.

Also to improve fit accuracy, the quadratic surrogate may have a logarithmic dependence on some of its parameters, meaning the surrogate depends on the $\log_{10}$ of some parameters. Which parameters are chosen to be log can be determined automatically.

5.2 Iterative Fitting Algorithm for Surrogate Convergence

On the current subdomain, a quadratic might fit quite poorly. The fit will eventually be improved through a branch and bound algorithm. However, straight assessment of the fitting error will not gage whether this fit is the best possible quadratic fit. The iterative fitting algorithm continues to add samples to the regression, until the quadratic fit stops changing. When the surrogate has "settled," errors are assessed.

Two quadratic forms Q_1 and Q_2 are compared over the domain $\mathcal{H}$ by evaluating the normalized deviation between the functions

$$\text{deviation} = \frac{\int_{\mathcal{H}} Q_1(\mathbf{x}) - Q_2(\mathbf{x})\ d\mathbf{x}}{\int_{\mathcal{H}} Q_2(\mathbf{x})\ d\mathbf{x}}$$

If the deviation has not met a tolerance, then more points are added to the regression to create a new fit.

5.3 Evaluation Storage for Rapid Restarts

When the `'isSavedEnabled'` flag of an m-file based ResponseModel is set to true, then evaluations of that model are stored in the "savedEvaluations" subdirectory of the main DC directory. When a single m-file produces multiple responses, then all responses are saved, even if only one of them is the current response being fitted.

Suppose that an m-file produces multiple responses and evaluations are flagged for saving. When the surrogate model is generated for the ResponseModel associated with one of the responses, all responses are saved to the hard disk. Subsequently, when the ResponseModel for another response is generated, the points are loaded from the disk, thereby reducing, and possibly eliminating the need for further evaluation of that m-file.

If the computation is restarted, any saved evaluations can be reused for fitting. Thus reducing the computation for creating surrogate fits from scratch. This also provides a pseudo-backup for power outages and system crashes that occur during long computations.

WARNING: If at any point, the simulation files are changed, you need to make sure to clear all saved files. This can be done by either deleting the files in the "savedEvaluations" subdirectory (and NOT the "savedEvaluationsDir.m") or by using a method:

```
>> deleteSavedEvaluations(dset);  %clears evaluations
                                  %for a DCDataset
>> deleteSavedEvaluations(rm);    %clears evaluations
                                  %for a single ResponseModel
```

5.4 Fitting Error Approximation

Once a quadratic surrogate has been generated, the maximum fitting error over the domain is estimated. This is done with a combination of determining the error of the fit points and validation points and via local searches with `fmincon`. The maximum error found from these three methods is then added to the ResponseObservation error to get the total error for a ModelAndObservationPair.

6 Response Prediction and Consistency Measure Calculations

Once a DCDataset has been created a consistency analysis can be performed by creating ConsistencyTest object.

```
>> ctestObj = ConsistencyTest(dset);
```

This calculates the **relative** consistency measure, $\mathrm{C}_{\mathcal{D}}$

$$\begin{aligned}\mathrm{C}_{\mathcal{D}} := &\max\ \gamma \\ &\text{s.t.}\ \begin{cases} l_e(1-\gamma) + l_{e,\text{fit}} \leq M_e(\mathbf{x}) - d_e \leq u_e(1-\gamma) + u_{e,\text{fit}},\ e = 1,\ldots,m \\ \mathbf{x} \in \mathcal{H} \end{cases}\end{aligned}$$

where l_e, $l_{e,\text{fit}}$, u_e, and $u_{e,\text{fit}}$ are the lower and upper experiment and surrogate fitting errors, d_e is the ModelObservation value, M_e is a ResponseModel, and $\mathcal{H}$ is the bounds defined by the FreeParameter object. We stress that this is the relative consistency measure, because it was defined in an absolute sense in some of our published work. If the consistency measure is positive, the dataset is said to be consistent, if the consistency is negative it is inconsistent. Further, the consistency measure is the percentage that all experiment uncertainty must be reduced in order to achieve consistency.

Likewise, if a DCDataset is constructed, we can make predictions of the range of values that an additional ResponseModel can take over the set of constrained feasible parameters. This is posed as the creation of a ResponsePrediction object.

```
>> predObj = ResponsePrediction(rm0,dset);
```

Creating this object solves two optimization problems

$$\begin{aligned}L_0 := &\min\ M_0(\mathbf{x}) \\ &\text{s.t.}\ \begin{cases} l_e + l_{e,\text{fit}} \leq M_e(\mathbf{x}) - d_e \leq u_e + u_{e,\text{fit}},\ e = 1,\ldots,m \\ \mathbf{x} \in \mathcal{H} \end{cases}\end{aligned}$$

$$\begin{aligned}R_0 := &\max\ M_0(\mathbf{x}) \\ &\text{s.t.}\ \begin{cases} l_e + l_{e,\text{fit}} \leq M_e(\mathbf{x}) - d_e \leq u_e + u_{e,\text{fit}},\ e = 1,\ldots,m \\ \mathbf{x} \in \mathcal{H} \end{cases}\end{aligned}$$

where M_0 is the function corresponding to the predicted ResponseModel.

Both the consistency and predictions can be customized using a DCOptions object (see §10).

```
>> ctestObj = ConsistencyTest(dset,opts);
>> predObj = ResponsePrediction(rm0,dset,opts);
```

In each calculation, quadratic surrogate models are fitted where required, and the problem is formulated as a nonconvex quadratically constrained quadratic program (NQCQP). Techniques for solving the NQCQP provide both outer and inner bounds on the solutions.

6.1 NQCQP Formulation & Bounding

After surrogate fitting, the problems are formulated as Nonconvex Quadratically Constrained Quadratic Programs. Outer bounds (upper bound on a maximization and a lower bound on a minimization) are solved for using what is known as the S- procedure. This formulates the problem as a semidefinite program which is solved with the SeDuMi optimization package.

Inner bounds (lower bound on a maximization and an upper bound on a minimization) are solved using a call to `fmincon`. Once an optimal parameter vector is found for the quadratic surrogates, the original model is evaluated to create a true inner bound.

6.2 Certification of Results

Once computation of the ConsistencyTest object or the ResponsePrediction object is complete, the objects contain a large amount of information regarding the calculation. This includes, but is not limited to,

- All branching locations for the branch and bound algorithm (see §8)
- All surrogate fits at each iteration of the branch and bound algorithm, with fitting errors, and number of function evaluations
- Bounds on the objective for each subdomain during the branching
- Sensitivities of the optimal value to experiment and parameter uncertainty (see §6.2.1).

The `report` method will generate a report on the calculations in HTML format (which can be viewed with a web-browser) in the current directory.

```
>> report(ctestObj);
>> report(predObj);
```

The bounds on the consistency measure are retrieved via

```
>> LB = ctestObj.LB; %lower bound on measure
>> UB = ctestObj.UB; %upper bound on measure
```

The bounds on the response prediction interval are retrieved in a similar fashion.

```
>> LBo = predObj.LBo; %outer bound on minimum
>> LBi = predObj.LBi; %inner bound on minimum
>> UBi = predObj.UBi; %inner bound on maximum
>> UBo = predObj.UBo; %outer bound on maximum
```

Furthermore, the parameter vectors that give the inner bounds are available.

```
>> LBx = predObj.LBx;
>> UBx = predObj.UBx;
```

6.2.1 Sensitivities

Outer bound computations automatically produce Lagrange multipliers. These are extracted from the object by

```
>> ctestMults = ctestObj.upperBndMults;
>> predMults = predObj.outerBndMults;
```

The prediction multipliers are in a structure with two fields `.lower` for the minimization problem and `.upper` for the maximization problem. `ctestMults`, `predMults.lower`, and `predMults.upper` are each a struct object with four fields: `.paraml`, `.paramu`, `.expl`, and `.expu` representing the multipliers for the lower and upper bound constraints for the parameter and experiments.

The objects can also convert the Lagrange multipliers to be sensitivities. These are the partial derivatives of the optimal outer bound objective with respect to the uncertainty bounds.

```
>> ctestSens = ctestObj.upperBndSens;
>> predSens = predObj.outerBndSens;
```

These are struct objects with the same form as the multiplier structs.

7 Parameter Optimization

Parameter optimization is a process of selecting a vector of parameters that minimizes some objective. The DC toolbox has a parameter optimization algorithm using the NQCQP techniques.

```
>> paramOptimObj = ParameterOptimization(dset);
```

Attempts to find the vector of parameter values $\mathbf{x}$ that solves the problem

$$\min_{\mathbf{x}\in\mathcal{H}} \sum_{e=1}^{m} w_e(M_e(\mathbf{x}) - d_e)^2$$

where w_e are weights that are by default set to $1/u_e$. This does so in the same general method as consistency and prediction. It creates polynomial fits where required, constructs an NQCQP, and computes upper and lower bounds on the objective. These bounds are retrieved via

```
>> LB = paramOptimObj.costLB;
>> UB = paramOptimObj.costUB;
```

The optimal parameter vector is retrieved via

```
>> x = paramOptimObj.bestx;
```

A DCOptions can be provided to set various options.

```
>> paramOptimObj = ParameterOptimization(dset,opts);
```

It is also possible to minimize the 1-norm or the infinity-norm instead of the 2-norm. This is done with

```
>> paramOptimObj = ParameterOptimization(dset,opts,norm);
```

where `norm` is either `'one'`, `'two'`, or `'inf'`.

Furthermore, you can specify the weights w_e as a mx1 vector (m being the number of ModelAndObservationPairs in the DCDataset).

```
>> paramOptimObj = ParameterOptimization(dset,opts,norm,weights);
```

8 Automatic Branch and Bound Algorithm

A branch and bound algorithm is provided for the computation of the ResponsePrediction, ConsistencyTest, and ParameterOptimization objects. By default, the computations do not perform any branching, but by setting `'maxBranchBoundIter'` option in the DCOptions object to 2 or greater (see §10), the branch and bound algorithm will performed automatically. The goal is to improve the surrogate fits by examining smaller subdomains (making the surrogate a piecewise quadratic) and to reduce the gap between inner and outer bounds.

Even when quadratic surrogates are already created, the branch and bound algorithm can improve the gap between inner and bounds that is due to the convex relation made the outer bounds.

8.1 Piecewise Surrogate Model Binary Tree

The branch and bound algorithm divides the domain into subdomains, and keeps track of each of the relevant subproblems in a binary tree data structure. This structure is contained by the PiecewiseSurrogateModelTree object. The user generally will not see, nor deal with this object, but it is available in the ResponsePrediction, and ConsistencyTest objects.

Each node of the tree represents a subdomain of the prior knowledge parameter domain. The children of a node represent the partition of the node into subdomains. Each division is along a single parameter. Each node includes surrogate fits and optimization bounds. For a minimization, the optimal value is the minimum of the optimal values over each of the leaves of the tree.

The tree structure provides a history of the algorithms behavior. It tracks where splits are made, the intermediate surrogate fits, and the optimal values.

9 Warm Starting Using Previously Calculated Piecewise Surrogates

Suppose that you run a consistency analysis, and results show that the consistency measure is bounded in the interval $[-0.23,\ 0.3]$. More iterations of the branch and bound algorithm should be able to resolve this gap. However, we have potentially spent lots of time generating

piecewise surrogate models for each of the ResponseModels in the DCDataset. However, we do not have to completely restart the algorithm. The ConsistencyTest constructor can take a ConsistencyTest object as one of its inputs to warm start the branch and bound iteration.

```
>> ctestObj = ConsistencyTest(dset);
>> opt = DCOptions('maxBranchBoundIter',2);
>> ctestObjNew = ConsistencyTest(ctestObj,opt);
```

This can also be done with the ResponsePrediction objects and ParameterOptimization objects. Furthermore, the ResponsePrediction constructor can be warm started with a ConsistencyTest object.

```
>> ctestObj = ConsistencyTest(dset); %first verify consistency
>> predObj = ResponsePrediction(rm0,ctestObj);
```

10 Fine Tuning Via User Options

There are many options that a user can set when creating a ConsistencyTest, ResponsePrediction, or ParameterOptimization object. These are set using the DCOptions object.

```
>> opts = DCOptions;
```

This creates a DCOptions object with all options set to their default. If no DCOptions object is provided to the algorithm, this is the default that is generated internally.

Options are set in one of two ways. The first is to list them during creation as property/value pairs.

```
>> opts = DCOptions(PROPERTY1,VALUE1,PROPERTY2,VALUE2,...)
```

The second way is to use dot-referencing to change the defaults or current options in an already created object. For example, say we wanted to turn off the `'display'` option.

```
>> opts = DCOptions;
>> opts.display = 'none';
```

The following is a list of all options the user can set, including their possible values and meaning. First a simple option for the display level.

`'display'` (default=`'iter'`). Can be 'off', 'final', 'notify', 'iter', 'all', or 'ALL'. Defines the amount of information printed to the screen during the algorithms.

The following properties affect the optimization algorithms.

`'omitInnerBound'` (default=false) Can be true or false. If true, inner bounds are not calculated. However, if `'maxBranchBoundIter'`>1, then inner bounds are calculated once for the branch and bound algorithm.

'omitOuterBound' (default=false) Can be true or false. If true, outer bounds are not calculated.

'maxBranchBoundIter' (default=1) Can be any positive integer. Maximum number of iterations of the branch and bound algorithm. If reached before tolerances are met, the algorithm exits. The first iteration equates to the first calculation before any branching. The second iteration is the calculations after the first branch.

'branchBoundTermTol' (default=0.02) Can be any positive number. The branch and bound algorithm will stop if the gap between inner and outer bounds is smaller than this number.

'nRestart' (default=2) Can be any positive integer. The number of times the inner bound optimizations are restarted with a new seed. The result returned is the optimum over all restarts.

'tolFun' (default=1e-5) Can be any positive number. This is the function tolerance used by the inner bound calculations (passed to `fmincon`).

'tolCon' (default=1e-5) Can be any positive number. This is the constraint tolerance used by the inner bound calculations (passed to `fmincon`).

'sedumiParEps' (default=1e-9) Can be any positive number. This is the optimization tolerance used by outer bound calculations (passed to SeDuMi).

'constraints' (default=[1 0 0 0]) A 1x4 array of 1's and 0's. Defines which transformations to perform the optimization on. The first column is linXlinY, the second is logXlinY, the third is linXlogY, and the fourth column is logXlogY, where X is the parameters and Y is the model outputs. 1 means include the indicated transformation in the optimization, 0 means exclude them. See the 'surfaceTransformation' option.

The following properties affect how surrogate fits are made.

'fitNorm' (default=2) Can be 2 or inf (the values, not the strings). This is the norm of residual fitting error that is to be minimized by the fit. Ultimately, the infinity-norm error is what is added to experiment uncertainties in the optimization. However, the infinity-norm problem takes longer to solve, and for large problems can run into memory issues.

'findIOTransforms' (default=true) Can be true or false. If true, uses some function evaluations to determine which parameters should have a $\log_{10}$ transformation for the best surrogate fit. Similarly, determines if the output should have a $\log_{10}$ transformation.

'activeParamSelCutOff' (default=0.05) Can be any positive number. A tolerance used for truncating the active parameter list. Setting this value to zero will insure all model parameters are used in the corresponding surrogate model. 100*activeParamSelCutOff roughly corresponds to the percent error introduced by eliminating nonactive parameters. We suggest never exceeding 0.2.

'nPntsPerCoeff4ActiveParamSel' (default=25) Can be any positive integer. Corresponds to the number of function evaluations per coefficients in the surrogate fit that are computed to determine active parameters.

'surfaceFittingMode' (default='iterative') Can be 'iterative' or 'oneShot'. If 'iterative', then surrogates are fitted using the iterative fitting algorithm (see §5.2). If 'oneShot', then surrogates are fit only once per iteration of the branch and bound algorithm.

'plotFitProgress' (default='off') Can be 'off' or 'on'. If on, and 'surfaceFittingMode' is set to 'iterative', then the difference metric between successive surrogate fits during the iterative fitting algorithm is plotted.

'minFitIter' (default=3) Can be any positive integer less than or equal to the 'maxFitIter' option. The minimum number of iterations of the iterative fitting algorithm to perform if 'surfaceFittingMode' is set to 'iterative'.

'maxFitIter' (default=7) Can be any positive integer greater than or equal to the 'minFitIter' option. The maximum number of iterations of the iterative fitting algorithm to perform if 'surfaceFittingMode' is set to 'iterative'.

'nSuccessfulFitIter' (default=2) Can be any positive integer. The number of times successive iterations a fit must meet the 'fitConvergenceTol' on the error convergence, before it is considered a good fit (if 'surfaceFittingMode' is set to 'iterative').

'fitConvergenceTol' (default=0.05) Can be any positive number. The tolerance that the metric between successive surrogate fits must meet to be considered "successful" (if 'surfaceFittingMode' is set to 'iterative').

'maxPnts4Fit' (default=Inf) Can be any positive integer. The maximum number of points that can be used for a surrogate fit.

'nPntsPerCoeff4OneShot' (default=20) Can be any positive integer. When 'surfaceFittingMode' is set to 'oneShot', this option determines the number of evaluations to use per coefficients in the surrogate to create the surrogate fit.

'subspaceDiscovery' (default='off') Can be 'off' or 'on'. When 'on' and when 'surfaceFittingMode' is set to 'oneShot', the surrogate fitting algorithm attempts to find a lower dimensional active subspace upon which the model depends.

'subspaceThreshold' (default=0.1) Can be any positive number. When 'subspaceDiscovery' is set to 'on', this option determines the cutoff of the singular values of a gradient matrix of the ResponseModel, thus determining the dimension of the active subspace. The cutoff is the first singular value σ_i that satisfies opt.subspaceThreshold*$\sigma_1 > \sigma_i$.

'derivRange' (default=0.01) Can be any positive number. When 'subspaceDiscovery' is set to 'on', this option determines the spread of evaluations used to approximate

gradients of the ResponseModel. If the parameter domain is normalized to be $[-1,\ 1]^n$, then `opt.derivRange` is the radius of the circle in these coordinates that contains all points used to estimate a gradient. Larger values may be required for ResponseModels with lots of noise.

`'surfaceTransformation'` (default=`{'linXlinY'}`) This is a cell array of strings listing the different transformations to be used when making a fit. This must be a 'superset' of the `'constraints'` option. Valid values are any cell array containing one or more of the following: `'linXlinY'`, `'logXlinY'`, `'linXlogY'`, and `'logXlogY'`.

`'useAllPnts4Fit'` (default=true) Can be true or false. After loading saved evaluations, the fitting algorithm will use all points already available to create the fit if this option is set to true. If set to false, the algorithm will only use the amount it would have computed had there been no saved points.

`'nLocalValidationSearches'` (default=3) Can be any nonnegative integer. When validating a surrogate fit, the algorithm uses `fmincon` to search for the location of the worst error. This option determines the number of restarts of this procedure.

`'nPntsPerCoeff4Validation'` (default=250) Can be any nonnegative integer. When validating a surrogate fit, the algorithm evaluates the ResponseModel and the surrogate at a set of validation points to look for average and worst case errors. This option determines the number of evaluations per coefficient in the surrogate for this validation.

11 Examples

The toolbox comes with several built-in examples. These are very useful for demonstration of the setup and basic operations associated with the toolbox. They can be found in the "examples" subdirectory of the main DC toolbox directory. The following is a list and brief description of each example. We recommend that you look at the m-file code for each example to see what it is doing and to read the comments before running the example. It may also be useful to run the examples one cell at a time using MATLAB's "code cell" features.

Each example requires that the proper directories are added to the path via the `DCsetup` script.

General Demos: In the "generalDemos" subdirectory of the "examples" directory, there are 3 examples. The three demo files — "demo1_linearModels.m", "demo2_quadraticModels.m", and "demo3_generalModel.m" — demonstrate creating some toy DCDatasets using each of the three types of ResponseModel, linear, quadratic, and general. These demos do nothing more than create the DCDataset object.

GRI-Mech 3.0: In the "GRI" subdirectory of the "examples" directory is a demo relating the GRI-Mech 3.0 dataset. This is a dataset of 77 experimental observables, with data and pre-created quadratic surrogate models. There are 102 parameters in the system. The file "gri_mech_demo.m" includes a demonstration of consistency testing, sensitivity

analysis, and prediction with this system. Since all model are already quadratic, the analysis is relatively quick; however, it still takes a few minutes due to the large size of the system.

Simple nonlinear dynamic system: In the "Prajna" subdirectory of the "examples" directory is a modified example presented by Stephen Prajna at the 2003 IEEE Conference on Decision and Control. This example has a nice HTML file in the "html" subdirectory of the "Prajna" directory called "runPrajnaExample.html" that can be viewed in a web browser. The HTML file shows the output and comments from the file "runPrajnaExample.m." This example demonstrates a consistency check on the dataset, and uses multiple features for nonquadratic models as well as evaluation saving.

Mass-Spring-Damper System: In the "massSpringDamper" subdirectory of the "examples" directory is an example involving the displacement of a mass attached to a spring and damper when a step input force is applied. The example does not have quadratic models. There are 3 features examined, and the model evaluations are saved on the hard disk. We suggest looking not only at the example m-file "runMSDexample.m," but also the model file "msdLinearModel.m."

12 Conversion From Version 1.0

Conversion from version 1.0 to version 2.0 is pretty simple, albeit, possibly tedious. The main thing you need to worry about is to make sure to run DCsetup at the beginning of each MATLAB session, and to make sure that if any DC objects are created inside a function to include the command `import DClab.*`. Lastly, Table 1 shows the object equivalence from version 1.0 to version 2.0.

Table 1: Object conversion table, showing the names of equivalent objects from versions 1.0 and 2.0 of the Data Collaboration toolbox.

Version 1.0	Version 2.0
ParameterAssertion	FreeParameter
ExperimentAssertion	ResponseObservation
ModelAssertion	ResponseModel
DatasetUnit	ModelAndObservationPair
ConsistTest	ConsistencyTest
Prediction	ResponsePrediction
ParameterOptimization	ParameterOptimization
DCOptions	DCOptions

All other objects are internal, and the user will most likely not interact with directly.

13 Troubleshooting, Help, and Feature requests

There are several common mistakes that are made when using the Data Collaboration toolbox that can lead to errors or incorrect results. They include

- If your models are flagged for saving evaluations, make sure that all old evaluations are cleared when you make changes to the model file.
- Check the units on the parameters in your models and in your FreeParameter objects. The analysis assumes that the FreeParameters are defined in the same units as the ResponseModels (including transformations). The analysis also assumes the output of the models is in the same units as the experiment data and uncertainty with the default behavior. It is possible for the data and uncertainty to defined in terms of the log of the ResponseModel output (see the ResponseObservation command line help).
- Each FreeParameter should have a unique name, that is case-dependent.
- The FreeParameter array that is given to the DCDataset constructor should include all FreeParameters listed by the `MODELDOMAIN` structures of the ResponseModels.

Each of the objects and their methods have command line help. This can be accessed by typing `help` followed by a space and the function name. For example to read about the DCOptions object and all its properties, type

```
>> help DCOptions
```

If you have any more questions, come across any bugs, have suggestions, or want to suggest features, please visit http://collab-sci.sourceforge.net/ and click on the "Tracker" link.