Defining Your Own SNOBFit Optimisation Problem

Barnaby Walker, James Bannock, Adrian Nightingale, and John de Mello

In our article '*Tuning Reaction Products by Constrained Optimisation*' we describe a simple method for carrying out multiobjective chemical optimisations, in which a compromise must be reached between several competing properties. The optimisation is carried out by treating the problem as a "constrained optimisation", in which a lead property is minimised subject to upper and lower limits being placed on the values that the other properties may attain. The optimisation routine used to carry out the optimisation is "Stable Noisy Optimisation by Branch and Fit (SNOBFit)" by Huyer and Neumaier (https://www.mat.univie.ac.at/neum/ms/snobfit.pdf). For convenience, we have developed a class-based wrapper for their Matlab-based implementation of SNOBFit (http://www.mat.univie.ac.at/neum/software/snobfit/), which simplifies both the installation of SNOBFit and its use, especially for chemical and blackbox optimisations.

In this folder you will find tutorials explaining how to install our MATLAB package, how to run a standard (unconstrained) optimisation with SNOBFit, and how to run a constrained optimisation.

Defining Your Own Functions

Our SNOBFit interface package includes some example objective and constraint functions that are commonly used for benchmarking the performance of global optimisation routines. You should first familiarise yourself with running these example functions by following the instructions in *using_snobfit.mlx*. The remainder of this document describes the procedure for defining your own objective and constraint functions.

SNOBFit Package Organisation

Our SNOBFit package is organised in the following folders:

```
+snobfitclass
├── +confcn
├── +minq5
├── +objfcn
├── +snobfcn
├── @snobHandler
└── @snobclass
```

The **@snobclass** folder contains files defining the SNOBFit class, and the functions that it uses to run an optimisation. The **@snobHandler** folder contains functions that allow the class to handle tasks such as plotting and saving data. As our SNOBFit interface is a wrapper around the original SNOBFit implementation provided by Huyer and Neumaier, it uses parts of their code to call the SNOBFit algorithm. The folders **+snobfcn** and **+minq5** contain files from their original implementation.

The important folders for defining your own custom optimisation are **+objfcn** and **+confcn**. *Objective function* definitions are stored in **+objfcn**, while *constraint function* definitions are stored in **+confcn**. Once you have written your function definitions (following the procedure outlined below), you will need to save them to the appropriate folders.

Writing an Objective Function for a Mathematical Optimisation

Your objective function must: * Take a SNOBFit object (an instance of the SNOBFit class) as its only argument * Return a 1D array of objective function values, with each cell of the output array corresponding to the function value obtained using the input parameters specified in the corresponding row of SNOB.next. (Note, SNOB.next is a 2D array generated internally by SNOBFit that contains a batch of data points for testing; each row of SNOB.next corresponds to a different set of input parameters that SNOBFit will test in the current batch).

An example objective function from the SNOBFit package is the *hsf18* 2D surface in **+objfcn**:

```
function f = hsf18(SNOB)

    x1 = SNOB.next(:,1);      % x1 is a 1D array (column vector) for the first
input parameter
    x2 = SNOB.next(:,2);      % x2 is a 1D array (column vector) for the second
input parameter

    f = 0.01*x1.^2 + x2.^2;   % f is a 1D array of objective function values

end
```

In this example: * SNOB is the SNOBFit object * Each row of the 2D array *SNOB.next* represents a set of input parameter values to be tested * Each column of *SNOB.next* represents a separate input parameter * Since we are carrying out a 2D optimisation SNOB.next has two columns, one for each input parameter * For clarity, the input parameters have been unpacked from SNOB.next and assigned to *x1* and *x2* * There is a single output property *f* that we wish to minimise * The objective function returns a 1D array *f*, where the *i*-th element *f*(*i*) holds the function value at [*x1*(*i*), *x2*(*i*)]

Writing a Constraint Function for a Mathematical Optimisation

A single MATLAB function is used to define all constraints. Your constraint MATLAB function must: * Take a SNOBFit object as its only argument * Return the values of all the constrained properties as a single *n*-by-*m* array, where *n* is the number of experiments (rows) and *m* is the number of constraints (columns)

An example:

```
function F = hsf18(SNOB)

```

```
    x1 = SNOB.next(:,1);            % first input parameter
    x2 = SNOB.next(:,2);            % second input parameter

    F(:,1) = x1.*x2 - 25;           % first constraint
    F(:,2) = x1.^2 + x2.^2 - 25;    % second constraint

end
```

In this example: * SNOB is the SNOBFit object * There are two constraints * The two input parameters have again been unpacked from SNOB.next as x1 and x2 * Each constraint is evaluated and stored as a separate column of the 2D-array *F*

Naming Your Function Files

In accordance with Matlab convention, each function file should be given the same name as the function name specified in the first line of code. For example, both functions above should be saved as **hsf18.m**. (It is not a requirement that the objective and constraint functions have the same name, but for clarity they should be named in a way that makes it obvious they are paired, e.g. "hsf18_obj" and "hsf18_con"). You must save *objective* functions in the **+objfcn** folder and *constraint* functions in the **+confcn** folder. (Having separate folders for the two types of function avoids conflicts when using the same function/file names for a pair of objective and constraint functions).

You need to supply the names of the objective and constraint functions to your SNOBFit object as follows:

```
snobfit_object.fcn = 'hsf18'       % objective function
snobfit_object.constraintFcn = 'hsf18'  % constraint function
```

Note: You can only assign functions in the **+objfcn** folder to *'snobfit_object.fcn'*, and similarly you can only assign functions in **+confcn** to *'snobfit_object.constraintFcn'*. If you do not define the constraint function, then SNOBFit will carry out an unconstrained optimisation.

To run the optimisation, you should follow the steps described in *using_constrained_snobfit.mlx*.

Defining a Chemical (or Blackbox) Optimisation

The instructions provided above include all of the information needed to define a mathematical optimisation problem, in which the objective function and constraint functions are known algebraic functions of the input parameters. Chemical optimisation is an example of blackbox optimisation, in which we do not know the functional dependence of the output properties on the input parameters, and so we must carry out an experiment to determine the output properties for a given set of input parameters. Blackbox optimisation is handled in a slightly different manner to mathematical optimisation as described below.

Formulating a Chemical Optimisation

Most chemical optimisations are examples of multiobjective optimisation problems, in which we wish to find an acceptable compromise between several criteria. As described in our article '*Tuning Reaction Products by Constrained Optimisation*', this may be conveniently achieved by framing the problem as a constrained optimisation, in which we optimise a lead property subject to constraints being placed on the values that the other properties may attain. For the purposes of this discussion, a "property" is anything that we can measure or calculate and wish to control. The only requirement is that the property can be expressed as a scalar quantity, i.e. a single number. Example properties include: the concentration of a product or side-product (measured), the mean crystal size in a colloidal dispersion (measured), the average chain length in a polymer solution (measured), or the total cost of reagents for the specific reaction conditions chosen (calculated).

For instance, suppose we wish to maximise the yield of a target molecule, while suppressing the formation of certain unwanted side products. To do this, we would set the yield of our target molecule as our lead property (i.e. the one that we wish to optimise), while asserting that the concentration of certain unwanted side products should not exceed specified values. The lead property is handled by the objective function, while the other properties are handled by the constraint function.

Writing Your Chemical Optimisation Files

Your experimental objective function should take the general form shown below, where *run_reaction* is a function that controls your experimental equipment and launches a sequence of reactions at the conditions specified in SNOB.next. *run_reaction* should return an array containing the values of all relevant properties at each set of reaction conditions. The property values should be stored as a *n*-by-*m* array in SNOB.valuesToPass (where *n* is the number of experiments and *m* is the number of output properties), allowing them to be retrieved later by the constraint function. In the code below we have assumed that values for the lead property are stored in the first column of output_properties. Hence, the objective function returns the first column of output_properties when called.

```
function f = my_objective_function(SNOB)

    input_parameter_1 = SNOB.next(:,1);
    input_parameter_2 = SNOB.next(:,2);

    output_properties = run_reaction(input_parameter_1, input_parameter_2);
%output_properties is a n x m array of property values, where each row corresponds
to a different set of input parameters and each column corresponds to a different
property
    SNOB.valuesToPass = output_properties; %store the property values for
subsequent retrieval by the constraint function

    f = output_properties(:,1); % the first column corresponds to the lead property
to minimise

end
```

In this example: * *input_parameter_1* and *input_parameter_2* represent the first and second reaction parameters, respectively * *run_reaction* is a function that takes the input parameters as an argument, performs **sequential reactions** and returns a *n*-by-*m* array of required output properties, where *n* is the number of experiments and *m* is the number of output properties * the output property values are stored in SNOB.valuesToPass, allowing them to be subsequently retrieved by the constraint function * the objective function as written above is configured to **minimise** output_properties(:,1)

Note: SNOBFit is configured to minimise the lead property. If you wish to maximise the lead property, this can be achieved by minimising the negative of the lead property by writing f = -output_properties(:,1);

In '*Tuning Reaction Products by Constrained Optimisation*' we optimised a cascadic synthesis with four competing products: X0, X1, X2, and X3. For instance, Run IV involved minimising [X3] (our lead property), while setting a minimum value of 90 % for [X1+X2] = [X1] + [X2] (our first constrained property) and a maximum value of 0.5 for the ratio R = [X1]/[X2] (our second constrained property) where [X] signifies the mole fraction of X in the sample.

This problem may be summarised as:

- Minimise [X3] subject to:
- [X1+X2] > 0.9
- R < 0.5

For the specific case of Run IV our objective function had the following form, where *run_reaction* was a function that accepted as its inputs the flow rates of the two reagents plus the temperature, and returned as its output a four column array containing [X0],[X1],[X2] and [X3] in columns one to four, respectively:

```
function f = my_objective_function(SNOB)

    flow_rate1 = SNOB.next(:,1);
    flow_rate2 = SNOB.next(:,2);
    temperature = SNOB.next(:,3);

    mole_fractions = run_reactor(flow_rate1, flow_rate2, temperature); % perform
experiments and return corresponding mole_fractions

    X3 = mole_fractions(:,4);   % lead property to be minimised is mole fraction of
X3 (which is stored in column four of mole_fractions)

    f = X3;

    SNOB.valuesToPass = mole_fractions; % save mole fractions for later retrieval
by constraint function

end
```

The constraint function had the following form:

```
function F = my_constraint_function(SNOB)

    mole_fractions = SNOB.valuesToPass; % retrieve mole fractions from SNOB object

    X1 = mole_fractions(:,2); %X1 is stored in second column of mole_fractions
    X2 = mole_fractions(:,3); %X2 is stored in third column of mole_fractions

    F(:,1) = X1 + X2; % calculate first constrained property
    F(:,2) = X1 / X2; % calculate second constrained property

end
```

Setting The Limits on the Constrained Properties

The limits for each constrained property should be set after creating a SNOBFit object in MATLAB, and before running the experiment.

To set up your SNOBFit object for the above optimisation type:

```
snobfit_object = snobclass(); %create object
snobfit_object.name = 'constrained_optimisation'; %specify file name
snobfit_object.fcn = 'my_objective_function'; %specify objective function
snobfit_object.constraintFcn = 'my_constraint_function'; %specify constraint
function
snobfit.constrained = true; %specify problem as a constrained optimisation; if set
to false, the constraints will be ignored
```

The values of the constraints are then set as *upper* and *lower* limits on each of the constraint functions:

```
snobfit_object.F_upper = [1.0; 0.5];
snobfit_object.F_lower = [0.9; 0.0];
```

In this example: * *snobfit_object.F_upper* stores the *upper limits* for each constrained property * *snobfit_object.F_lower* stores the *lower limits* for each constrained property * Both are *n*-by-*1* arrays, where *n* is the number of constrained properties * The order of the elements in *F_lower* and *F_upper* must match the column order of *F* * The first column of *F* corresponds to [X1+X2]. Hence the first elements of *F_lower* and *F_upper* must correspond to the limits on [X1+X2] * The second column of *F* corresponds to R. Hence the second elements of *F_lower* and *F_upper* must correspond to the limits on R

SNOBFIT treats the bounds specified in *F_lower* and *F_upper* as "preferred limits" that can be partially violated if this leads to a better solution, i.e. one with a lower merit value. The extent to which the preferred bounds may be violated is specified using the σ parameter:

```
snobfit_object.sigma = [0.3; 0.3]
```

Here we have set σ to the same value of 0.3 for both constraints, but you can chose any values that suit your purpose. The simplest way to choose the σ values is to set them equal to the maximum tolerable violation of each constraint. For the example given, our choice of σ values would mean that we could tolerate anything down to 0.6 for [X1+X2] (*F_lower* - σ), and anything up to 0.8 for R (*F_upper* + σ). The order of the elements in snobfit_object.sigma must match the column order of *F*.

You can also set different values of σ for the upper and lower limits of each constraint. If we wanted to do this for the above example:

```
snobfit_object.sigmaUpper = [0.3; 0.3];
snobfit_object.sigmaLower = [0.1; 0.1];
```

Setting Bounds on the Input Parameters

SNOBFit is a bounded optimisation algorithm, which means you must specify upper and lower limits for each input parameter. This makes it a good fit for chemical optimisations, where the range of usable reaction conditions is typically limited due to physical limitations. For instance, in a flow-based reaction, the boiling point of a solvent places an upper limit on the reaction temperature. It may sometimes be the case that you know from previous experience the range of conditions within which the optimium lies, meaning you should limit your search to include this range only.

The bounds on the input parameters can be set by changing properties on the SNOBFit object. For a 2D optimisation (i.e. one with two variable reaction conditions) you should enter something like this:

```
snobfit_object.x_lower = [5; 30];    % lower bounds
snobfit_object.x_upper = [25; 80];   % upper bounds
```

In this example: * The lower bounds are set with the *x_lower* property, and the upper bounds are set with the *x_upper* property. * SNOBfit infers the dimensionality of the optimisation from the number of lower/upper bounds you declare. You do not need to specify the dimensionality elsewhere. Hence, since there are two elements in *x_lower* and *x_upper*, SNOBfit knows that you are carrying out a 2D optimisation. * *x_lower* and *x_upper* are both *n*-by-*1* arrays where *n* is the number of reaction conditions, or dimensions, you are changing in your optimisation. * In the above example SNOBFit is allowed to test any value between 5 and 25 for input parameter *1* and any value between 30 and 80 for parameter *2*. The exact values you specify will of course depend on the nature of your own experiment. * The order of the bounds in *x_lower* and *x_upper* must match the column order in *snobfit_object.next*. Hence, if the first column of *snobfit_object.next* is the reaction temperature, then the first elements of *x_lower* and *x_upper* must correspond to the lower and upper bounds on the temperature.

Linked Bounds

If you are using a flow reactor, it is frequently necessary to set limits on the total flow-rate of reagents and the ratio of those flow rates: if the total flow-rate is too low, the reaction will take too long; if the total flow-rate is too high or the individual flow rates are too different, the flow may become unstable. For an optimisation in which we are varying two flow rates, applying limits on the total flow rate and the flow rate ratio results in a trapezoidal boundary that cannot be directly handled by SNOBFit (see our article for further details). To allow SNOBFit to handle the trapezoidal constraints, we use a spatial transformation that converts the trapezoidal boundary into a rectangular one that can be handled by SNOBFit. This is done automatically by following the procedure described below. **In its current implementation the method works for two reagent flow rates only, plus a third (optional) unlinked reaction condition**.

To use linked reaction conditions, you need to change the property on the SNOBFit object:

```
snobfit_object.linked = true
```

You can then set the bounds for the linked reaction conditions:

```
snobfit_object.xyMin = 50;      % the minimum overall flow rate
snobfit_object.xyMax = 300;     % the maximum overall flow rate

snobfit_object.minRatio = 0.5;  % the minimum ratio between the flow rates
snobfit_object.maxRatio = 2.0;  % the maximum ratio between the flow rates
```

The bounds on a third reaction condition can then also be set:

```
snobfit_object.zMin = 100;
snobfit_object.zMax = 150;
```

Termination Criteria

The final thing that you might want to change (depending on the nature of your optimisation) is the termination criterion, which is used to decide when the optimisation has finished. The options that have been incorporated into the SNOBFit object are called **'n_runs'**, **'minimised'**, and **'no_change'**.

The default option is **'n_runs'. This terminates the optimisation after a set number of evaluations of the objective function (and is the one we used for the work described in our article):

```
snobfit_object.termination = 'n_runs'; % termination criterion
snobfit_object.ncall = 100;            % maximum number of function evaluations
```

Another termination criterion is **'minimised'**, which will end the optimisation when the best objective function value lies within a threshold of a target value fglob, or after a maximum number of objective function evaluations. You can set the threshold and target values:

```
snobfit_object.termination = 'minimised';  % termination criterion
snobfit_object.fglob = 0;                  % target minimum, defaults to zero if
not known
snobfit_object.threshold = 0.001;          % threshold for termination
snobfit_object.ncall = 100;                % maximum number of objective function
evaluations
```

If you are running a constrained optimisation, there is an additional check to make sure the point(s) that satisfy the termination criterion also satisfy the constraints. If an objective value smaller than (fglob + threshold) is found under conditions that do not satisfy the constraints, the optimisation will continue.

The final termination criterion included in the SNOBFit object is **'no_change'**. This ends the optimisation if there has been no change in the best objective function value for a set number of calls to the SNOBFit algorithm. There is a chance that applying this criterion may cause the optimisation to terminate too early, so you can also set a minimum number of objective function evaluations before checking for a change:

```
snobfit_object.termination = 'no_change'; % termination criterion
snobfit_object.ncallNoChange = 5;         % number of SNOBFit calls without a
change before terminating
snobfit_object.minCalls = 50;             % minimum number of function evaluations
before checking for a change
```

These termination conditions have been included in the SNOBFit object for ease of use. There may be other criteria that are more suitable to your particular use. If you want to add any conditions, you can add them to the **'+snobfitclass/@snobclass/checkTermination.m'** file.

This should be all of the information that you need to set up your own chemical optimisation procedure. Good luck!