Short Developer's Reference of COCO

Frank Schilder, Department of Mathematics, DTU, Denmark
Harry Dankowicz Department of Mechanical Sciences and Engineering, UIUC, USA

8. September 2011

Table of Contents

1 Running a Continuation

A continuation problem consists of at least one zero problem and requires the definition of at least one parameter. As an example, consider the zero problem

$$F(u)=0,\ \ F(u):=u_1^2+u_2^2-1,\ \ F:\mathbb{R}^2\to\mathbb{R},$$

defining a circle with radius 1 in the (u_1, u_2) - plane. We can compute this circle using continuation in one of the components of the vector u. In a first step we need to add the function F as a zero problem and in a second step we define u_2 as the continuation parameter. A COCO compatible Matlab-function for implementing the function F is

```
function [data y] = circle(opts, data, u)
y = u(1)^2 + u(2)^2 - 1;
end
```

Here, y is the function value and u is the argument vector. All other formal parameters must be present, but will be ignored for now. Detailed explanations are given in Section 2.1. We define this function as a zero problem and the component u(2) as the parameter 'mu' by executing the two commands

```
opts = []
opts = coco_add_func(opts, 'circle', @circle, [], 'zero', 'x0', [1;0]);
opts = coco_add_parameters(opts, '', 2, 'mu');
```

COCO stores all information about a continuation problem in the options structure `opts`. This structure is always the first argument when calling a function from the toolbox COCO. After defining our continuation problem we can run the continuation by executing

```
bd   = coco(opts, '1', [], 'mu', [-2 2]);
```

This will produce the screen output

```
   STEP   DAMPING              NORMS              COMPUTATION TIMES
  IT SIT     GAMMA     ||d||     ||f||     ||U||   F(x)  DF(x)  SOLVE
   0                          0.00e+00  1.00e+00    0.0    0.0    0.0

 STEP  TYPE   LABEL          mu        ||U||      TIME
    0  EP         1  0.0000e+00   1.0000e+00  00:00:00
   10             2  8.4540e-01   1.3095e+00  00:00:00
   20             3  9.9597e-01   1.4114e+00  00:00:01
   30             4  7.4212e-01   1.2453e+00  00:00:01
   40             5 -1.6224e-01   1.0131e+00  00:00:01
   50             6 -9.1098e-01   1.3527e+00  00:00:01
   60             7 -9.8126e-01   1.4010e+00  00:00:01
   70             8 -6.3674e-01   1.1855e+00  00:00:01
   80             9  3.6223e-01   1.0636e+00  00:00:02
   90            10  9.5533e-01   1.3830e+00  00:00:02
  100  EP        11  9.5525e-01   1.3829e+00  00:00:02

 STEP  TYPE   LABEL          mu        ||U||      TIME
    0  EP        12  0.0000e+00   1.0000e+00  00:00:02
   10            13 -8.4540e-01   1.3095e+00  00:00:02
   20            14 -9.9597e-01   1.4114e+00  00:00:02
   30            15 -7.4212e-01   1.2453e+00  00:00:03
   40            16  1.6224e-01   1.0131e+00  00:00:03
   50            17  9.1098e-01   1.3527e+00  00:00:03
   60            18  9.8126e-01   1.4010e+00  00:00:03
   70            19  6.3674e-01   1.1855e+00  00:00:03
   80            20 -3.6223e-01   1.0636e+00  00:00:04
   90            21 -9.5533e-01   1.3830e+00  00:00:04
  100  EP        22 -9.5525e-01   1.3829e+00  00:00:04
```

After running the continuation we can plot a simple bifurcation diagram by executing

```
u  = coco_bd_col(bd, '||U||');
mu = coco_bd_col(bd, 'mu');
plot(mu, u)
```

Note, however, that this will not plot the circle, because the bifurcation diagram will by default not contain the solution vector, but only the continuation parameter and the norm of the solution vector.

1.1 Coco Function Reference

The Matlab function for running a continuation with the toolbox COCO is `coco`. This function has two forms of calling:

```
bd = coco(opts, run, [], PARS, PInt);
```

and:

```
bd = coco(opts, run, TBNM, FPT, TPT, TBARGS, PARS, PInt);
```

In the first form the continuation problem is created manually by the user and in the second form a COCO compatible toolbox constructs the continuation problem. The arguments are

bd	COCO returns a cell array containing the bifurcation diagram. The contents of *bd* can be modified; see Section 2.4. The functions `coco_bd_col`, `coco_bd_labs` and `coco_bd_val` provide an interface to access data in the bifurcation diagram; see Section 4.
opts	COCO's options structure is always the first argument to any of COCO's toolbox functions.
run	*Run name* or *run identifier.* A unique, user-defined identifier for the computation to be performed. *Run* can either be a Matlab string, or a cell array of Matlab strings. A typical choice is `'1'`, `'2'`, `...` `Coco` saves the bifurcation diagram and solution data in the sub-directory `'data'` of the current directory. Defining names for runs allows to organise this data for later access, for example, plotting. The run identifier is used to construct the name of a sub-directory for a run such that data computed in a specific run does not overwrite data from another run. For example, setting `run='1'` will save all data to the sub-directory `'1'` of `'data'`. Setting `run={'1','a'}` on the other hand will save all data to the sub-directory `'a'` of `'data/1'`, that is, one can think of `run={'1','a'}` as being sub-run `'a'` of `run='1'`. Functions for reading a bifurcation diagram from disk and for post-processing bifurcation diagrams are described in Section 4.
PARS	Name of continuation parameter. *PARS* can either be a Matlab string or a cell array of Matlab strings. If more than one parameter is specified, the values of the additional parameters will be included in the screen output and in the bifurcation diagram. This is called *parameter over-specification* and is useful, for example, to output values of test functions during a continuation.
PInt	Parameter interval. *PInt* defines the *continuation window*, the interval within which the continuation parameter may vary (also referred to as the *computational domain*). This is a 1x2 vector.
TBNM	Name of a COCO compatible toolbox. *TBNM* is a Matlab string. COCO uses *TBNM* to construct the name of a so-called parser function; see Section 3.2.
FPT	Acronym for the type of initial solution (**F**rom-**P**oint-**T**ype). COCO uses *FPT* to construct the name of a parser function; see Section 3.2.

TPT	Acronym for the type of solution that should be continued (**T**o-**P**oint-**T**ype). COCO uses *TPT* to construct the name of a parser function; see Section 3.2.
TBARGS	Arguments that are passed to the parser function of the toolbox *TBNM*. The list of accepted arguments is defined by the parser function.

2 Defining a Continuation Problem

2.1 Adding a Zero Problem

A zero problem is a function $F(u)=0,\ F:\mathbb{R}^m \rightarrow \mathbb{R}^n,$ where $m>n.$ A zero problem is defined as a Matlab function of the general form

```
function [data y] = FunctionName(opts, data, u)
y = FunctionBody(u);
end
```

By default, derivatives are computed using finite differences. While this is sufficient for simple zero problems, it is usually too slow or too inaccurate for complex zero problems. The general form of a Matlab function for defining the derivative of a zero problem explicitly is

```
function [data J] = FunctionName_DFDU(opts, data, u)
J = FunctionBody(u);
end
```

The formal parameters of both functions are

`data`	Structure with so-called *function data* or *toolbox data*. This structure is defined when calling `coco_add_func` and allows to store information about a continuation problem, for example, information about the contents of `u`. The function has read-write access to the contents of `data`.
`y`	*N*-dimensional vector with function values $y=F(u)\in\mathbb{R}^n$.
`J`	*N*-by-*M* Jacobian matrix of the function $J=\frac{\partial F}{\partial u}(u)\in\mathbb{R}^{m\times n}$.
`opts`	COCO's options structure. Functions have read-access to the options structure. This argument is only required in very advanced applications and should be ignored in most common situations.
`u`	*M*-dimensional argument vector.

A zero problem is added to a continuation problem using the function `coco_add_func`. The general syntax is

```
opts = coco_add_func(opts, Name, @func, [@func_DFDU,] data, ...
         'zero', 'x0', U0);
```

`opts`	COCO's options structure.

Name	A short descriptive name of the function. This is useful for debugging purposes and very advanced applications.
@func	A function handle of a zero problem as defined above.
@func_DFDU	A function handle of the derivative of the zero problem. This argument is optional. If a derivative is not specified explicitly, numerical differentiation is used. This is acceptable for simple algorithms, but will be too inaccurate or too slow for more advanced applications.
`data`	The function data structure.
U0	An initial guess for the vector u such that $F(u) \approx 0$.

2.2 Defining Parameters

Typically, part of the vector u of a zero problem will correspond to parameters of the problem. The function `coco_add_parameters` allows to define which components of u should be treated as parameters. Typically, one needs to define $m-n \geq 1$ parameters.

`opts = coco_add_parameters(opts, '', PIdx, PNM);`	
`opts`	COCO's options structure.
PIdx	Set of indices such that `u(`*PIdx*`)` is the set of parameters. *PIdx* is an array of integers.
PNM	List of short descriptive names or an array of numbers. In the first form, *PNM* is a string or a cell array of strings assigning a name to each parameter. In the second form a list of default names of the form `'PAR(`*n*`)'` is constructed, where *n* iterates through all elements of *PNM*.

2.3 Adding Test Functions and Events

Detecting and locating special points along a solution curve is called event handling. To use event handling one has to define a monitor- or test function, add this function to the continuation problem and assign events to the parameters associated with the event function. A monitor function has exactly the same form as a zero problem

```
function [data y] = FunctionName(opts, data, u)
y = FunctionBody(u);
end
```

`data`	Structure with so-called *function data*. This structure is defined when calling `coco_add_func` and allows to store information about a continuation problem, for example, information about the contents of `u`. The function has read-write access to the contents of `data`.
`y`	*N*-dimensional vector with function values.
`opts`	COCO's options structure. Functions have read-access to the opts structure. This argument is only required in very advanced applications and should be ignored in most common situations.
`u`	*M*-dimensional argument vector.

The syntax for adding a test function is different from the syntax for adding a zero problem:

opts = **coco_add_func**(opts, *Name*, @*func*, data, *EVType*, *PNM*);	
opts	COCO's options structure.
Name	A short descriptive name of the monitor function. This is useful for debugging purposes and very advanced applications.
@*func*	A function handle of a monitor function as defined above.
data	The function data structure.
EVType	The type of event associated with this monitor function. In most applications *EVType* can be set to either `'regular'` (events are regular solution points) or `'singular'` (events are singular solution points).
PNM	List of short descriptive names. *PNM* is either a string or a cell array of strings assigning a parameter name to each component of the vector the monitor function returns. These names can later be used to assign an event to a specific monitor function or for additional output; see parameter over-specification on Page 3.

The function coco_add_event allows to assign an event to any parameter associated with a monitor function or defined with `coco_add_parameters`. The general syntax is

opts = **coco_add_event**(opts, *EVLab*, [*EVType*,] *PNM*, *EVVals*);	
opts	COCO's options structure.
EVLab	Short descriptive label to identify an event in the bifurcation diagram. This is a Matlab string and should contain between 2-4 characters, for example, `'LP'` for limit point.
EVType	Type of an event (default `= 'special point'`). This argument is optional. Typically, events are bifurcation points, which have the default event type `'special point'`. It is also possible to use event handling to define computational boundaries, in which case one has to use *EVType*=`'boundary'`. The continuation will stop whenever such a boundary-event is detected, while it will continue after detecting special points.
PNM	Name of parameter the event will be assigned to. This is a Matlab string.
EVVals	A list of event values. Each crossing of the value of a monitor function of an event value will be detected and located. Typically, one uses *EVVals*=`0` (detect zero crossings only).

2.4 Customising Output

COCO uses a signal-slot mechanism to allow the modification of output to the screen, the bifurcation diagram and the disk. A slot function is connected to a signal with `coco_add_slot`

opts = **coco_add_slot**(opts, *Name*, @*func*, data, *Signal*);	
opts	COCO's options structure.
Name	A short descriptive name of the slot function. This is useful for debugging purposes and very advanced applications.
@*func*	A function handle of a slot function as defined below.

`data`	The function data structure.
`Signal`	The name of the signal. The most commonly used signals are `'bddat'`, `'cont_print'`, `'corr_print'` and `'save_full'`; see details below. An important but less commonly used signal is `'FSM_update'`.

A slot function has the general form

```
function [data [res]] = slot_func(opts, data, ...)
Function Body
end
```

`data`	The function data structure.
`res`	Optional output argument. Whether or not a slot function should return `res` is defined by the signal the function is connected to; see below.
`opts`	COCO's options structure.
`...`	Additional arguments present depending on the signal the function is connected to; see below.

The signal 'bddat'

This signal is used to add data to the cell array `bd` returned by a call to `coco`. The form of a `'bddat'`-slot function must be

```
function [data res] = bddat_slot_func(opts, data, command, sol)
switch command
  case 'init'
    res = { ListOfNames };
  case 'data'
    res = { ListOfValues };
end
end
```

`sol`	The solution structure `sol` contains information about the current solution point. The most useful fields are the full solution vector `sol.x`, the point type `sol.pt_type` and the solution label `sol.lab`. The point type and the solution label are printed on screen in columns `TYPE` and `LABEL`.
`ListOfNames`	List of names. These names will be stored in the first row of the bifurcation diagram and allow easy access to the data associated with these rows using `coco_bd_col`; see below. The number of names must match the number of values. Note that a value may be a vector or matrix.
`ListOfValues`	List of values. These values will be stored in the bifurcation diagram. The number of values must match the number of names. Note that a value may be a vector or a matrix.

The signals 'cont_print' and 'corr_print'

These signals are used to print additional output on screen. The signal `'cont_print'` will add output during continuation, and the signal `'corr_print'` during the correction. The form of either `'*_print'`-slot function must be

```
function data = print_slot_func(opts, data, command, x)
switch command
  case 'init'
    fprintf(Headline);
  case 'data'
    fprintf(Dataline);
end
end
```

`x`	The full solution vector.
`Headline`	An `fprintf` statement printing a descriptive headline for the additional output.
`Dataline`	An `fprintf` statement printing the additional output.

The signal 'save_full'

Since the continuation algorithm will save a solution structure containing extensive information about the solution point for each labelled solution, it is usually only necessary to save a toolbox' data structure in addition to this solution structure. There are two pre-defined slot functions that simplify this common task, the functions `coco_save_data` and `coco_save_ptr_data`. Use these functions as in

```
opts = coco_add_slot(opts, Name, @coco_save_data,     data,     'save_full');
opts = coco_add_slot(opts, Name, @coco_save_ptr_data, data_ptr, 'save_full');
```

The first form will save the function data structure and the second form will extract the data structure from a pointer. This pointer must have been created with `coco_ptr`. Use a unique descriptive name when adding these slots. One can later restore the solution structure and function data with `coco_read_solution` as in

```
[data sol] = coco_read_solution(Name, run, lab);
```

`Name`	Name used when adding the slot function. This is a Matlab string.
`run`	Identifier of the run during which the solution was computed. *`Run`* is either a string or a cell array of strings.
`lab`	Label of the solution data to read. *`Lab`* is an integer.

3 Developing COCO Compatible Toolboxes

A COCO compatible toolbox consists of a set of constructor and parser functions. Constructor functions typically

- assemble the function data structure `data` of the toolbox,
- set-up a zero problem,
- define the set of parameters,

- add relevant test functions and events, and
- add useful information to the screen output, the bifurcation diagram and solution files.

A constructor function should always add the toolbox data structure `data` to solution files; see below. A parser function typically

- parses the arguments provided by the user and
- calls an appropriate constructor function.

Parser functions are selected by the function `coco` according to the three input arguments *`TBNM`*, *`FPT`*, *`TPT`*; see Section 1.1. This supports an easy and systematic selection of a parser function depending on the task to perform, for example, start a computation from a user-provided initial point, or switch branches at a bifurcation point. Each parser function can define its own set of arguments that a user needs to specify when calling `coco`.

3.1 Constructor functions

A toolbox usually has at least one constructor function. The general form of a constructor function is

```
function opts = TBXName_create(opts, ARGS)
FunctionBody
end
```

`TBXName`	The name of the toolbox. A usual naming convention for constructor functions is toolbox name + `'_create'`.
`opts`	COCO's options structure.
`ARGS`	Any number of arguments required to construct a zero problem.

3.2 Parser Functions

A toolbox usually has a collection of parser functions. Most commonly, parser functions are available for

- starting at an initial point provided by the user,
- re-starting at a solution point from a previous continuation run, and
- branch-switching at bifurcation points.

The general form of a parser function is

```
function [opts argnum] = TBXName_FPT2TPT(opts, prefix, ARGS, varargin)
FunctionBody
argnum = Number_of_ARGS + 1;
end
```

`TBXName`	The name of the toolbox.
`FPT`	An acronym for the type of initial solution to start from (**F**rom-**P**oint-**T**ype).
`TPT`	An acronym for the type of solution to continue (**T**o-**P**oint-**T**ype).
`opts`	COCO's options structure.
`argnum`	Since the function `coco` cannot a-priori know how many arguments a specific parser will use, it simply passes all arguments to parser functions. These, in turn, must return

	to `coco` how many arguments from the argument list were actually used. This number does not count the argument `opts`, but counts all arguments including the argument `prefix`, which is only relevant for very advanced toolboxes.
`prefix`	This argument is only relevant for very advanced uses and can be ignored.
`ARGS`	Arguments from the call to `coco` that are passed to the parser function.
`varargin`	This formal parameter must always be present to allow surplus arguments being passed to a parser. These additional arguments will usually include parameters for the actual continuation algorithm and are to be ignored by a parser.

Starting at an Initial Point

A simple implementation of a parser function would just forward its arguments to a toolbox constructor and return the number of arguments used back to `coco`. The basic algorithm is

```
function [opts argnum] = parser(opts, prefix, ARG1, ..., ARGN, varargin)
opts = constructor(opts, ARG1, ..., ARGN);
argnum = N+1;
end
```

Restarting at a Saved Solution Point

A simple implementation of a re-start parser would load the data of a solution computed in a previous run from disk, call the toolbox constructor and return the number of arguments used back to `coco`. A solution from a previous continuation run is uniquely identified by a run and a label. The run is called *restart run* and the label *restart label*. The run is usually just the string that a user passed to the function `coco` and the label is an integer, which was printed on screen as well as stored in the bifurcation diagram returned by `coco`. The basic algorithm is

```
function [opts argnum] = parser(opts, prefix, rrun, rlab, ARG1, ..., ARGN, varargin)
[data sol] = coco_read_solution(save_SlotName, rrun, rlab);
CARGS = ReconstructArgsForConstructor(data, sol);
opts = constructor(opts, CARGS, ARG1, ..., ARGN);
argnum = N + 3;
end
```

`opts`	COCO's options structure.
`argnum`	Number of arguments used by the constructor.
`prefix`	Ignore this argument.
`rrun`	Identifyer of the restart run. Typically, this is a Matlab string.
`rlab`	Restart label. This is an integer.
`save_SlotName`	Name that was used by the constructor when adding a slot function to the signal `'save_full'`; see Section 2.4.
`CARGS`	Arguments that are required by the constructor function and can be restored from the data saved to disk, for example, the solution point and the toolbox data structure.

Branch-Switching at a Bifurcation Point (Obsolete.)

Branch-switching at a bifurcation point is very similar to re-starting at a saved solution point. The key difference is, that, in addition, one needs to compute an approximation to the tangent vector of and an initial point on the bifurcating branch. The basic algorithm is (the differences to the re-start parser are marked in reed)

```
function [opts argnum] = parser(opts, prefix, rrun, rlab, ARG1, ..., ARGN, varargin)
[data sol] = coco_read_solution(save_SlotName, rrun, rlab);
[x0 p0] = ExtractPointFromSolutionStructure(sol);
t = ComputeTangentAtNewBranch;
[x0 p0] = computeFirstPoint(opts, data, x0, p0, t);
CARGS = ReconstructArgsForConstructor(data, sol, x0, p0);
opts = constructor(opts, CARGS, ARG1, ..., ARGN);
argnum = N + 3;
end
```

`opts`	COCO's options structure.
`argnum`	Number of arguments used by the constructor.
`prefix`	Ignore this argument.
`rrun`	Identifyer of the restart run. Typically, this is a Matlab string.
`rlab`	Restart label. This is an integer.
`save_SlotName`	Name that was used by the constructor when adding a slot function to the signal `'save_full'`; see Section 2.4.
`CARGS`	Arguments that are required by the constructor function and can be restored from the data saved to disk, for example, the solution point and the toolbox data structure.

The function `computeFirstPoint` is part of the example toolbox `curve05`. This function predicts an initial point in the direction of the tangent vector `t` with some step length `h`. By default, `h` is set to 0.001. This may not be appropriate for some applications and can be modified by setting the property `'h0'` of the class `'cont'` as in `opts = coco_set(opts, 'cont', 'h0', h)`. Make sure to adjust also the values of `'h_max'` and `'h_min'` of the class `'cont'` such that `h_min<=h0<=h_max` holds.

Note that `computeFirstPoint` will only work properly if the fields `data.x_idx`, `data.p_idx`, `data.TB_F` (toolbox function, the actual zero problem) and `data.acp_idx` (index of active continuation parameter) of the toolbox data structure are set as in the template toolbox `curve05`. Do not change these parts of the code. Furthermore, the argument `data` must be a valid toolbox data structure for solutions of the branch to switch to, in other words, make sure you update the restored data structure before calling `computeFirstPoint` if necessary.

3.3 Defining Toolbox Properties

Toolbox properties are a user-friendly way to adapt a toolbox to a specific situation, for example, by allowing a user to switch certain features on or off. Properties are defined using the function `coco_set`, and can be accessed using the function `coco_get`. Toolbox properties are typically stored in a structure. To simplify working with property structures, `coco_set` has two different forms of use, the *set-form* and the *merge-form*. The general syntax of the set-form is

`opts = coco_set(opts, TBXName, PropName, PropValue);`	
`opts`	COCO's options structure.
`TBXName`	Name or an acronym of the name of the toolbox, also referred to as a *class name*. Pick a unique and somewhat descriptive name to avoid name clashes. This argument is a string.
`PropName`	Name of the property to set. This argument is a string.
`PropValue`	Value to assign to the property. This can be any Matlab data type.

The general syntax of the merge-form is

`opts3 = coco_set(opts1, opts2);`

In this form `coco_set` merges two Matlab structures recursively. The resulting structure `opts3` will have the union of the fields of the structures `opts1` and `opts2`. The merge operation gives precedence to fields in `opts2`, that is, a field present in `opts2` will overwrite a field with the same name in `opts1`. The most common situation for calling the merge-form of `coco_set` is to overwrite settings in a structure containing default values for all toolbox properties with the actual user settings, if present, as in
`tb_opts = coco_get(opts, TBXName);`
`tb_opts = coco_set(defaults, tb_opts);`
Here, `coco_get` will extract any user settings for the toolbox with name `TBXName` from COCO's options structure; see description of `coco_get` below. Subsequently, `coco_set` will merge these settings with the default settings in the structure `defaults`.

The function `coco_get` extracts any toolbox properties defined with `coco_set` from COCO's options structure.

`tb_opts = coco_get(opts, TBXName);`	
`tb_opts`	A structure containing all fields that were set with the function `coco_set` in the set-form. If no properties were set, `coco_get` returns an empty structure.
`opts`	COCO's options structure.
`TBXName`	Name or an acronym of the name of the toolbox as used when calling `coco_set` in the set-form. This name is also referred to as a *class name*.

Properties of the Continuation Algorithm

Commonly used properties of the continuation algorithm, class `'cont'`.		
Property	Default	Description
`h0`	`0.1`	Initial continuation step size.

h_max	0.5	Maximal continuation step size.
h_min	0.01	Minimal continuation step size.
ItMX	100	Maximum number of continuation steps. The general form is [ItFW,ItBW], where ItFW is the number of steps in forward and ItBW in backward direction. If only one number is specified, ItFW and ItBW are set to the same value.
LogLevel	[1 0]	Controls the amount of diagnostic output on screen. The first number affects the continuation and the second number the correction algorithm. When set to zero, no output will be produced. Higher levels increase the amount of information printed. For the continuation algorithm the values 0, 1, 2 and 3, and for the correction algorithm the values 0, 1 can be chosen.
NPR	10	Print and save information about the current solution point at least every NPR continuation steps. A unique solution label will be assigned to each printed and saved solution.

Properties of the Correction Algorithm

Commonly used properties of the correction algorithm, class 'corr'.		
Property	Default	Description
ItMX	10	Maximum number of iterations. If the solution cannot be computed within this number of steps, the continuation step size will be reduced and another attempt of correction is made. This is repeated until the minimum continuation step size is reached. If it is not possible to compute a new solution, the continuation of the branch in this direction will terminate. This is indicated in the bifurcation diagram with the point type 'MX' (maximum number of iterations exceeded).
SubItMX	8	Maximum number of damping steps. If set to 1 the corrector becomes the classical Newton method. Higher values result in a damped Newton method with increasing damping. Some damping typically improves the convergence properties of Newton's method.
TOL	1.00E-008	Convergence criterion on the norm of the Newton correction.
ResTOL	1.00E-012	Convergence criterion on the norm of the residuum.
LogLevel	1	Controls the amount of diagnostic output on screen. When set to zero, no output will be produced.

4 Functions for post-processing

To simplify branch-switching and plotting of bifurcation diagrams COCO offers functions for post-processing of a bifurcation diagram and for reading a bifurcation diagram from disk.

To read a previously computed bifurcation diagram from disk use

```
bd = coco_bd_read(run);
```

`bd`	The bifurcation diagram as returned by `coco` after running a continuation with run name *run*.
run	The run name or run identifier of the bifurcation diagram.

To extract a full column from a bifurcation diagram use

`col = coco_bd_col(bd, Name);`	
`col`	A Matlab array of values of a column in the bifurcation diagram. `Coco_bd_col` tries to merge all values into a numerical array and will return a cell array if this fails.
`bd`	A bifurcation diagram as returned by `coco` or `coco_bd_read`.
Name	The name of the column to extract. This is a string, which must match a name in *ListOfNames* as defined by the corresponding `'bddat'` call back function; see Section 2.4.

To extract all solution labels of a bifurcation point use

`labs = coco_bd_labs(bd, PTType);`	
`labs`	A list of solution labels. This is a numerical array of integers and may be empty if no point of type *PTType* was detected. These labels can be used for branch-switching at a bifurcation point of type *PTType*, or for plotting bifurcation points in a bifurcation diagram; see function `coco_bd_val` below.
`bd`	A bifurcation diagram as returned by `coco` or `coco_bd_read`.
PTType	The type of the bifurcation point. This is a string, which must match *EVLab* as defined by the function `coco_add_event`; see Section 2.3.

The function `coco_bd_val` extracts a single value from a bifurcation diagram. One can interpret this function as accessing a bifurcation diagram in the form `bd(row,col)`, where `row` is a solution label and `col` is the name of a column. Another interpretation is, that `coco_bd_val` is a combination of `coco_bd_col` and `coco_bd_lab`. The calling syntax is

`val = coco_bd_val(bd, lab, Name);`	
`val`	The value in the bifurcation diagram in column *Name* of the solution with label *lab*.
`bd`	A bifurcation diagram as returned by `coco` or `coco_bd_read`.
lab	A solution label.
name	The name of the column. This is a string, which must match a name in *ListOfNames* as defined by the corresponding `'bddat'` call back function; see Section 2.4.

5 Utility Functions

5.1 Pointers to Toolbox Data Structures

Sometimes it is necessary to allow different functions from a toolbox to modify a shared copy of the toolbox data structure. A simple way of making this possible is, to create a pointer or reference to the data structure using `coco_ptr`

`data_ptr = coco_ptr(data);`	
`data_ptr`	A pointer to the structure data. After this call, the structure `data` can be accessed using the pointer as `data_ptr.data`. Any changes to `data_ptr.data` will affect `data` and vice versa. If one creates a copy of `data_ptr`, for example `ptr2`, then `data_ptr.data` and `ptr2.data` will access the same copy of the structure `data`.
`data`	A toolbox data structure.

5.2 Numerical Differentiation

For many test functions one needs to compute the derivative of a function with respect to its arguments or parameters. COCO contains a toolbox FDM (Finite Difference Methods) for computing numerical approximations of derivatives. These functions are quite flexible and allow the differentiation of functions with a variety of input and output arguments. The most common application, however, is the differentiation of a function of the form `y=f(x,p)`. To compute the Jacobi matrix use

`J = fdm_ezDFDX('f(x,p)', @func, x, p);`	
`J`	The Jacobi matrix *df/dx*.
@func	Function handle. The function must return a vector *y* and must have two input arguments *x* and *p*.
x,p	The point at which to compute the Jacobian.

To compute derivatives of a function with respect to parameters use

`J = fdm_ezDFDP('f(x,p)', @func, x, p, pars);`	
`J`	The matrix of derivatives *df/dp* for the parameters selected with *pars*.
@func	Function handle. The function must return a vector *y* and must have two input arguments *x* and *p*.
x,p	The point at which to compute the derivatives.
pars	An index vector specifying with derivatives to compute. Setting *pars*=`1:numel(p)` will compute the derivatives with respect to all parameters. Setting *pars* to a subset thereof will compute the corresponding subset of derivatives.