Short Developer's Reference for COCO

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

22. October 2014

Table of Contents

1 Running a Continuation....2
1.1 COCO Function Reference....3
2 Defining a Continuation Problem....5
2.1 Adding a Zero Problem....5
2.2 Defining Continuation Parameters....6
2.3 Adding Test Functions and Events....6
2.4 Customizing Output....7
The signal 'bddat'....8
The signals 'cont_print' and 'corr_print'....9
The signal 'save_full'....9
3 Developing COCO-compatible Toolboxes....10
3.1 Constructor functions....10
3.2 Parser Functions....10
Starting at an Initial Point....11
Restarting at a Saved Solution Point....11
Branch-Switching at a Bifurcation Point....12
3.3 Defining Toolbox Properties....15
Properties of the Continuation Algorithm....16
Properties of the Correction Algorithm....17
4 Functions for post-processing....17
5 Utility Functions....18
5.1 Toolbox Data Structures with Shared Fields....18
5.2 Numerical Differentiation....19

1 Running a Continuation

A continuation problem consists of at least one *zero problem* and, usually, the definition of at least one *active continuation 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 by treating one of the components of the vector u as an active continuation parameter. In a first step, we must add the function F as a zero problem. In the second step, we define u_2 as the active continuation parameter. A COCO-compatible Matlab-encoding 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 a vector of *continuation variables*. All other formal parameters must be present, but will be ignored for now. Detailed explanations are given in Section 2.1.

We construct an empty continuation problem, associate the function `circle` with a zero problem, and identify the component `u(2)` with the initially inactive continuation parameter `'mu'` by executing the three commands

```
prob = coco_prob();
prob = coco_add_func(prob, 'circle', @circle, [], 'zero', 'u0', [1;0]);
prob = coco_add_pars(prob, '', 2, 'mu');
```

COCO stores all information about the continuation problem in the *continuation problem structure* `prob`. The core constructor `coco_prob` initializes an empty continuation problem structure. This structure is typically the first argument when calling a function from the toolbox COCO.

After defining our continuation problem, we run the continuation by executing the command

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

which activates the continuation parameter `'mu'` on the interval `[-2,2]`. This produces 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      TIME        ||U||  LABEL  TYPE            mu
    0  00:00:00   1.0000e+00      1  EP      0.0000e+00
   10  00:00:02   1.3095e+00      2         -8.4540e-01
   19  00:00:03   1.4142e+00      3  FP     -1.0000e+00
   20  00:00:03   1.4114e+00      4         -9.9597e-01
   30  00:00:04   1.2453e+00      5         -7.4212e-01
   40  00:00:04   1.0131e+00      6          1.6224e-01
```

```
  50  00:00:05   1.3527e+00      7          9.1098e-01
  57  00:00:06   1.4142e+00      8  FP      1.0000e+00
  60  00:00:06   1.4010e+00      9          9.8126e-01
  70  00:00:07   1.1855e+00     10          6.3674e-01
  80  00:00:08   1.0636e+00     11         -3.6223e-01
  90  00:00:09   1.3830e+00     12         -9.5533e-01
  95  00:00:09   1.4142e+00     13  FP     -1.0000e+00
 100  00:00:10   1.3829e+00     14  EP     -9.5525e-01

STEP      TIME        ||U||   LABEL  TYPE           mu
   0  00:00:10   1.0000e+00     15  EP      0.0000e+00
  10  00:00:11   1.3095e+00     16          8.4540e-01
  19  00:00:12   1.4142e+00     17  FP      1.0000e+00
  20  00:00:12   1.4114e+00     18          9.9597e-01
  30  00:00:13   1.2453e+00     19          7.4212e-01
  40  00:00:14   1.0131e+00     20         -1.6224e-01
  50  00:00:15   1.3527e+00     21         -9.1098e-01
  57  00:00:16   1.4142e+00     22  FP     -1.0000e+00
  60  00:00:16   1.4010e+00     23         -9.8126e-01
  70  00:00:18   1.1855e+00     24         -6.3674e-01
  80  00:00:19   1.0636e+00     25          3.6223e-01
  90  00:00:20   1.3830e+00     26          9.5533e-01
  95  00:00:20   1.4142e+00     27  FP      1.0000e+00
 100  00:00:21   1.3829e+00     28  EP      9.5525e-01
```

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

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

Note that this does not plot the circle, but rather the graph of the norm of the solution vector, parametrized by the continuation parameter `'mu'`, as these are included by default in the array of bifurcation data returned by the `coco` command.

1.1 COCO Function Reference

The Matlab function for running a continuation with the toolbox COCO is `coco`. For one-dimensional solution manifolds, this function has two forms of calling syntax:

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

and

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

In the first form, the continuation problem is created manually by the user and stored, in advance of continuation, in the continuation problem structure `prob`. In the second form, a COCO-compatible toolbox appends the continuation problem to the continuation problem structure stored in `prob`. The

arguments are

bd	The entry-point function `coco` returns a cell array containing the bifurcation data. 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 cell array; see Section 4.
`prob`	A continuation problem structure is typically the first argument to any of COCO's toolbox functions.
run	*Run name* or *run identifier*. A unique, user-defined run name or run identifier must be associated with the computation to be performed. This can either be a Matlab string, or a cell array of Matlab strings. A common choice is `'1'`, `'2'`, ... `coco` saves the bifurcation data and solution data in the sub-directory `'data'` of the current directory. Unique run names organize this data for later access, for example, plotting. The run identifier is used to construct the name of a sub-directory 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 bifurcation data from disk and for post-processing bifurcation data are described in Section 4.
PARS	Continuation parameters. This is either a Matlab string or a cell array of Matlab strings corresponding to labels for a subset of the continuation parameters. The values of these parameters are included in the screen output. Parameters are activated in the order listed to the extent necessary to ensure that the number of continuation variables and active continuation parameters exceeds the dimension of the zero problem by 1. The inclusion of additional parameters is called *parameter over-specification* and is useful, for example, to output values of test functions during a continuation.
PInt	Parameter Intervals. This defines the *computational domain* in terms of a set of intervals within which the corresponding continuation parameter may vary. *PInt* is a 1x2 vector or a cell array of 1x2 vectors. The empty interval `[]` is identical to `[-inf, inf]`. Trailing instances of the empty interval may be omitted.
TBNM	**T**ool**b**ox **Nam**e. *TBNM* is a Matlab string representing the name of a COCO-compatible toolbox. COCO uses *TBNM* to construct the name of a so-called parser function; see Section 3.2.
FPT	**F**rom-**P**oint-**T**ype. The Matlab string *FPT* contains an acronym for the type of initial solution. COCO uses *FPT* to construct the name of a parser function; see Section 3.2.
TPT	**T**o-**P**oint-**T**ype. The Matlab string *TPT* contains an acronym for the type of solution that should be continued. COCO uses *TPT* to construct the name of a parser function; see Section 3.2.
TBARGS	**T**ool**b**ox **Arg**ument**s**. The input argument *TBARGS* represents the sequence of additional input 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 $F(u)=0,\ F:\mathbb{R}^m\to\mathbb{R}^n,$ where $m>n.$ A zero problem is defined as a Matlab function of the COCO-compatible form

```
function [data y] = FunctionName(prob, 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 COCO-compatible form of a Matlab function for defining the derivative of a zero problem explicitly is

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

The formal parameters of both functions are

`data`	Structure with *function data* or *toolbox data*. This structure is defined when calling `coco_add_func` and stores information about the 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: $J=\frac{\partial F}{\partial u}(u)\in\mathbb{R}^{n\times m}$.
`prob`	A continuation problem structure. The function has read-access to the contents of `prob`. This argument is only required in very advanced applications and should be ignored in most common situations.
`u`	*m*-dimensional vector of continuation variables.

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

```
prob = coco_add_func(prob, FID, @func, [@func_DFDU,] data, ...
          'zero', 'u0', u0);
```

`prob`	A continuation problem structure.
FID	A unique function identifier given by a Matlab string. This is useful for debugging purposes and advanced applications.
@func	A function handle to the encoding of a zero problem as defined above.

@func_DFDU	A function handle to the encoding 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`	Initial content for the function data structure. In advanced applications this may change during execution.
u0	An initial guess for the vector u such that $F(u) \approx 0$.

2.2 Defining Continuation Parameters

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

`prob = coco_add_pars(prob, '', PIdx, PNM);`	
`prob`	A continuation problem structure.
PIdx	Parameter indices. Set of integer indices such that `u(PIdx)` corresponds to the set of problem parameters.
PNM	Parameter names. A string or a cell array of strings assigning short descriptive names to each associated continuation parameter.

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 monitor function. A COCO-compatible encoding of a monitor function has exactly the same form as that associated with a zero problem:

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

`data`	Structure with s*function data*. This structure is defined when calling `coco_add_func` and stores 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`	*V*ector with function values.
`prob`	A continuation problem structure. The function has read-access to the content of `prob`. This argument is only required in very advanced applications and should be ignored in most common situations.
`u`	*V*ector of continuation variables.

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

```
prob = coco_add_func(prob, FID, @func, data, EVType, PNM);
```

prob	A continuation problem structure.
FID	A unique function identifier given by a Matlab string. This is useful for debugging purposes and advanced applications.
@func	A function handle to a COCO-compatible encoding 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	Parameter names. A string or a cell array of strings assigning short descriptive labels to the continuation parameters associated with each component of the vector returned by the monitor function. 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 4.

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

prob = **coco_add_event**(prob, *EVLab*, [*EVType*,] *PNM*, *EVVals*);	
prob	A continuation problem structure.
EVLab	Event label (point type). A short (2-4 characters) Matlab string providing a descriptive label to identify an event in the bifurcation data, for example, 'LP' for limit point.
EVType	Event type. Optional argument describing the type of an event. 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'. 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 with an event surface associated with an event value will be detected and located. Typically, one uses EVVals=0 (detect zero crossings only).

2.4 Customizing Output

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

prob = **coco_add_slot**(prob, *SFID*, *@func*, data, *Signal*);	
prob	A continuation problem structure.
SFID	A slot function identifier given by a Matlab string. The slot function identifier must be unique for each signal. This is useful for debugging purposes and advanced applications.

@func	A function handle to a signal-compatible encoding of a slot function as defined below.
data	The slot 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 'update'.

A slot function has the general form

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

data	The function data structure.
res	Optional output arguments. Whether or not a slot function should return any output is defined by the signal the function is connected to; see below.
prob	A continuation problem structure.
...	Additional input/output arguments depending on the signal the slot 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(prob, data, command, varargin)
switch command
  case 'init'
    res = { ListOfNames };
  case 'data'
    chart = varargin{1};
    res = { ListOfValues(chart) };
end
end
```

chart	The solution chart contains information about the current solution point. The most useful fields are the full solution vector chart.x, the point type chart.pt_type and the solution label chart.lab. The point type and the solution label are printed on screen in columns TYPE and LABEL.
ListOfNames	This list of names will be stored as column headers in the first row of the bifurcation data cell array and allow easy access to the data associated with these columns 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	This list of values will be stored in the bifurcation data cell array. 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'` adds output during continuation, and the signal `'corr_print'` adds output during the correction. The form of either `'*_print'`-slot function must be

```
function data = print_slot_func(prob, data, command, LogLevel, varargin)
switch command
  case 'init'
    coco_print(prob, LogLevel, format1, Headline);
  case 'data'
    x = varargin{1}.x;
    coco_print(prob, LogLevel, format2, Dataline(x));
end
end
```

format1	An `fprintf` format string.
Headline	A descriptive headline for the additional output.
`x`	The full solution vector.
format2	An `fprintf` format string.
Dataline	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. COCO provides the pre-defined slot function `coco_save_data` to simplify this common task:

```
prob = coco_add_slot(prob, SFID, @coco_save_data, data, 'save_full');
```

This will save the function data structure together with the solution. It is customary to use the function identifier `FID` of a zero problem also as the slot function identifier `SFID` for saving its function data, because `coco_read_solution` will then automatically retrieve the function data together with the solution data associated to the same identifier. To restore the solution structure and function data use `coco_read_solution` as in

```
[data chart] = coco_read_solution([ID], run, lab);
```

ID	Slot function identifier. Optional Matlab string identifier used when adding the slot function. If omitted, return the entire solution. If present, return the solution and function data associated with the identifier `ID`.

run	Run identifier of the run during which the solution was computed, given by a string or a cell array of strings.
lab	Integer label for the corresponding solution.

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 of the toolbox,
- set-up a zero problem,
- define the set of continuation parameters,
- add relevant test functions and events, and
- add useful information to the screen output, the bifurcation data, and solution files.

A constructor function should always add the toolbox data structure 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 prob = TBXName_construct(prob, ARGS)
FunctionBody
end
```

TBXName	The name of the toolbox. A usual naming convention for constructor functions is toolbox name + `'_construct'` or toolbox name + `'_create'`.
`prob`	A continuation problem structure.
ARGS	Any number of arguments required to construct a toolbox instance.

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 prob = TBXName_FPT2TPT(prob, oid, varargin)
str = coco_stream(varargin);
FunctionBody(str);
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).
`prob`	A continuation problem structure.
`oid`	An object identifier. The object identifier should be used with the function `coco_get_id` to form identifiers. For example, the *toolbox identifier* is defined to be `coco_get_id(oid, TBXName)`. This is typically used as the function identifier for the zero problem and the `'save_full'`-slot function used to save the function data of the zero problem.
`varargin`	Additional arguments from the call to `coco` that are passed to the parser function. 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. It is important that any parser function converts `varargin` to a `coco_stream` object and accesses arguments only through this object. To simplify input parsing, COCO provides the function `coco_parse`.

Starting at an Initial Point

A simple implementation of a parser function has no optional arguments and simply forwards its arguments to a toolbox constructor:

```
function prob = parser(prob, oid, varargin)
str = coco_stream(varargin{:});
[ARG1 ... ARGN] = str.get();
prob = constructor(opts, ARG1, ..., ARGN);
end
```

Restarting at a Saved Solution Point

A simple implementation of a restart parser loads the data of a solution computed in a previous run from disk, construct all arguments required by the toolbox constructor and then calls the toolbox constructor. 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 data returned by `coco`. The basic algorithm is

```
function prob = parser(prob, oid, varargin)
str = coco_stream(varargin{:});
[rrun rlab ARG1 ... ARGM] = str.get();
tbid = coco_get_id(oid, TBXName);
[data sol] = coco_read_solution(tbid, rrun, rlab);
[ARG1, ..., ARGN] = ReconstructArgsForConstructor(data, sol, ARG1, ..., ARGM);
prob = constructor(prob, ARG1, ..., ARGN);
end
```

`prob`	A continuation problem structure.
`oid`	An object identifier.
`rrun`	The restart run identifier; typically, a Matlab string.
`rlab`	The integer restart label.
`tbid`	A unique toolbox identifier.

Branch-Switching at a Bifurcation Point

Branch-switching at a bifurcation point is almost identical to starting at an initial guess, if the bifurcation point is a regular solution point. All that needs to be done is to compute an approximation of a regular solution point on the new branch close to the bifurcation point. One then calls the appropriate toolbox parser designed to compute the new branch starting at an initial guess.

Branch switching at a singular point (branch point) is more involved and, to some extent, beyond the scope of this short reference. We illustrate the fundamentals with an example and refer to other documentation or toolboxes for further reference. Essentially, branch switching at a branch point requires passing an initial direction vector together with the initial solution point in the call to `coco_add_func`. A vector that is suitable for use in most situations is the singular vector of the Jacobian of the full continuation problem, which is saved by COCO's linear solver in *chart data* associated with a(n approximately located) branch point.

Let us illustrate this methodology with the example of two unit circles defined by the zero-set of the function

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

intersecting each other in two points. A COCO-compatible encoding of this function is

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

and running a continuation produces the output

```
>> prob = coco_prob();
>> prob = coco_add_func(prob, 'circles', @circles, [], 'zero', 'u0', [1;0]);
>> prob = coco_add_pars(prob, '', [1 2], {'x' 'y'});
>> bd = coco(prob, '1', [], {'x' 'y'}, [-2 2]);
```

```
   STEP   DAMPING               NORMS              COMPUTATION TIMES
 IT SIT     GAMMA     ||d||     ||f||     ||U||   F(x)  DF(x)  SOLVE
  0                          0.00e+00  1.41e+00    0.0    0.0    0.0

STEP      TIME        ||U||  LABEL  TYPE            x            y
   0  00:00:00   1.4142e+00      1  EP      1.0000e+00   0.0000e+00
   1  00:00:00   1.4142e+00      2  FP      1.0000e+00  -3.8061e-08
  10  00:00:01   1.4142e+00      3          6.8992e-01  -7.2389e-01
  13  00:00:01   1.4142e+00      4  BP      5.0000e-01  -8.6603e-01
  20  00:00:02   1.4142e+00      5         -5.9291e-02  -9.9824e-01
  30  00:00:03   1.4142e+00      6         -7.7075e-01  -6.3713e-01
  39  00:00:03   1.4142e+00      7  FP     -1.0000e+00   3.9613e-06
  40  00:00:04   1.4142e+00      8         -9.9157e-01   1.2956e-01
  50  00:00:04   1.4142e+00      9         -5.8117e-01   8.1378e-01
  60  00:00:05   1.4142e+00     10          1.9919e-01   9.7996e-01
  64  00:00:06   1.4142e+00     11  BP      5.0000e-01   8.6603e-01
  70  00:00:06   1.4142e+00     12          8.5275e-01   5.2232e-01
  77  00:00:07   1.4142e+00     13  FP      1.0000e+00   8.0110e-06
  80  00:00:07   1.4142e+00     14          9.6347e-01  -2.6782e-01
  90  00:00:08   1.4142e+00     15  BP      5.0000e-01  -8.6603e-01
  90  00:00:08   1.4142e+00     16          4.6086e-01  -8.8747e-01
 100  00:00:09   1.4142e+00     17  EP     -3.3512e-01  -9.4218e-01

STEP      TIME        ||U||  LABEL  TYPE            x            y
   0  00:00:09   1.4142e+00     18  EP      1.0000e+00   0.0000e+00
  10  00:00:10   1.4142e+00     19          6.8992e-01   7.2389e-01
  13  00:00:11   1.4142e+00     20  BP      5.0000e-01   8.6603e-01
  20  00:00:11   1.4142e+00     21         -5.9291e-02   9.9824e-01
  30  00:00:12   1.4142e+00     22         -7.7075e-01   6.3713e-01
  39  00:00:13   1.4142e+00     23  FP     -1.0000e+00  -3.9613e-06
  40  00:00:13   1.4142e+00     24         -9.9157e-01  -1.2956e-01
  50  00:00:14   1.4142e+00     25         -5.8117e-01  -8.1378e-01
  60  00:00:15   1.4142e+00     26          1.9919e-01  -9.7996e-01
  64  00:00:16   1.4142e+00     27  BP      5.0000e-01  -8.6603e-01
  70  00:00:16   1.4142e+00     28          8.5275e-01  -5.2232e-01
  77  00:00:17   1.4142e+00     29  FP      1.0000e+00  -8.0110e-06
  80  00:00:17   1.4142e+00     30          9.6347e-01   2.6782e-01
  90  00:00:18   1.4142e+00     31  BP      5.0000e-01   8.6603e-01
  90  00:00:18   1.4142e+00     32          4.6086e-01   8.8747e-01
 100  00:00:19   1.4142e+00     33  EP     -3.3512e-01   9.4218e-01
```

The intersection points are detected as branch points. Inspecting the solution and chart data of the branch point with label 4 reveals the tangent vector as well as the singular vector:

```
>> [data chart uidx] = coco_read_solution('circles', '1', 4, ...
'data', 'chart', 'uidx');
>> chart.x
```

```
ans =

    0.5000
   -0.8660

>> chart.t

ans =

   -0.6124
   -0.3535

>> cd = coco_get_chart_data(chart, 'lsol');
>> cd.v(uidx)

ans =

    0.3976
   -0.5847
```

Note that the singular vector `cd.v(uidx)` is not tangent to the intersecting circle. We achieve a continuation of the intersecting circle with the sequence of commands (continued from above)

```
>> prob = coco_prob();
>> prob = coco_add_func(prob, 'circles', @circles, [], 'zero', ...
'u0', chart.x, 't0', cd.v(uidx));
>> prob = coco_add_pars(prob, '', [1 2], {'x' 'y'});
>> prob = coco_set(prob, 'cont', 'NullItMX', 1);
>> bd = coco(prob, '2', [], {'x' 'y'}, [-2 2]);

 STEP      TIME        ||U||  LABEL  TYPE             x            y
    0  00:00:00   1.4142e+00      1  EP      5.0000e-01  -8.6603e-01
    1  00:00:00   1.4142e+00      2  BP      5.0000e-01  -8.6603e-01
   10  00:00:01   4.2737e-01      3          4.5661e-02  -2.9873e-01
   14  00:00:01   1.1287e-05      4  FP     -1.7708e-07  -7.9790e-06
   20  00:00:02   7.2350e-01      5          1.3086e-01   4.9457e-01
   27  00:00:02   1.4142e+00      6  BP      5.0000e-01   8.6603e-01
   30  00:00:03   1.7542e+00      7          7.6934e-01   9.7303e-01
   40  00:00:04   2.4937e+00      8          1.5547e+00   8.3208e-01
   50  00:00:05   2.8191e+00      9          1.9869e+00   1.6144e-01
   52  00:00:05   2.8284e+00     10  FP      2.0000e+00   2.1660e-06
   60  00:00:06   2.6765e+00     11          1.7909e+00  -6.1197e-01
   70  00:00:06   2.0894e+00     12          1.0914e+00  -9.9581e-01
   78  00:00:07   1.4142e+00     13  BP      5.0000e-01  -8.6603e-01
   80  00:00:08   1.1554e+00     14          3.3377e-01  -7.4574e-01
   90  00:00:08   2.9626e-02     15          2.1943e-04  -2.0948e-02
   91  00:00:09   1.2708e-05     16  FP     -1.4312e-07   8.9845e-06
  100  00:00:09   1.1011e+00     17  EP      3.0311e-01   7.1718e-01
```

```
STEP      TIME        ||U||  LABEL  TYPE            x            y
   0  00:00:10   1.4142e+00     18  EP      5.0000e-01  -8.6603e-01
  10  00:00:10   2.2647e+00     19          1.2822e+00  -9.5936e-01
  20  00:00:11   2.7526e+00     20          1.8943e+00  -4.4757e-01
  26  00:00:12   2.8284e+00     21  FP      2.0000e+00  -6.7235e-06
  30  00:00:12   2.7836e+00     22          1.9371e+00   3.4914e-01
  40  00:00:13   2.3523e+00     23          1.3834e+00   9.2360e-01
  50  00:00:14   1.5305e+00     24          5.8562e-01   9.1010e-01
  52  00:00:15   1.4142e+00     25  BP      5.0000e-01   8.6603e-01
  60  00:00:15   4.5458e-01     26          5.1661e-02   3.1726e-01
  64  00:00:16   7.2730e-06     27  FP     -1.5190e-08   5.1428e-06
  70  00:00:16   6.9683e-01     28          1.2139e-01  -4.7755e-01
  77  00:00:17   1.4142e+00     29  BP      5.0000e-01  -8.6603e-01
  80  00:00:17   1.7325e+00     30          7.5043e-01  -9.6836e-01
  90  00:00:18   2.4806e+00     31          1.5383e+00  -8.4273e-01
 100  00:00:19   2.8168e+00     32  EP      1.9836e+00  -1.8064e-01
```

Note the passing of the branch point `chart.x` and the singular vector `cd.v(uidx)` in the call to `coco_add_func`. We also set the toolbox property `'NullItMX'` of the toolbox `'cont'` to 1. This enables the application of an algorithm for improving the initial direction to obtain a vector closer to the actual tangent vector of the intersecting circle. See the next section for information on toolbox properties.

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 provides the function `coco_merge` to merge two property structures. The general syntax of a call to `coco_set` is

prob = **coco_set**(prob, *TBXName*, *PropName*, *PropValue*, ...);	
prob	A continuation problem 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.
...	Further pairs of *PropName* and *PropValue* for multiple assignments in one line.

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

```
tb_opts = coco_get(prob, TBXName);
```

`tb_opts`	A structure containing all fields associated with the toolbox *`TBXName`* that were set with the function `coco_set`. If no properties were set, `coco_get` returns an empty structure.
`prob`	A continuation problem structure.
`TBXName`	Name or an acronym of the name of the toolbox used when calling `coco_set`. This name is also referred to as a *class name*.

To merge two property structures, for example, the properties set with `coco_set` and a structure with default values, use `coco_merge`.

```
props = coco_merge(props1, props2);
```

Merge the two property structures `props1` and `props2` recursively. The resulting structure `props` will have the union of the fields of the structures `props1` and `props2`. The merge operation gives precedence to fields in `props2`, that is, a field present in `props2` will overwrite a field with the same name in `props1`. The most common situation for calling `coco_merge` 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(prob, TBXName);
tb_opts = coco_merge(defaults, tb_opts);
```

Here, `coco_get` extracts any user settings for the toolbox with name `TBXName` from the continuation problem structure. Subsequently, `coco_merge` merges these settings with default settings stored in the structure `defaults`.

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 for one-dimensional manifolds is `[ItBW,ItFW]`, 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 data with the point type 'MX' (maximum number of iterations exceeded).
SubItMX	4	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-006	Convergence criterion on the norm of the Newton correction.
ResTOL	1.00E-006	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 bifurcation data and for reading bifurcation data from disk.

To read previously computed bifurcation data from disk, use

bd = **coco_bd_read**(*run*);	
bd	The bifurcation data 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 the bifurcation data cell array, use

col = **coco_bd_col**(bd, *Name*);	
col	A Matlab array of values of a column in the bifurcation diagram. The coco_bd_col function tries to concatenate all values into a numerical array and will return a cell array if this fails.
bd	A bifurcation data cell array 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' slot function; see Section 2.4. In addition, all names of continuation parameters and parameters associated with monitor functions can be used.

To extract all solution labels of corresponding to an event label (point type), 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 with label *PTType* was detected. These solution labels can be used for branch-switching at a bifurcation point with label *PTType*, or for plotting bifurcation points in a bifurcation diagram; see function coco_bd_val below.
bd	A bifurcation data cell array as returned by coco or coco_bd_read.
PTType	A string event label (point type), which must match *EVLab* as defined by the function coco_add_event; see Section 2.3. The event label is printed in the column TYPE of the bifurcation data.

The function coco_bd_val extracts a single value from a bifurcation data cell array. One can interpret this function as accessing the content of the bifurcation data cell array in the form bd(lab,col), where lab is an integer 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 data cell array in column *Name* of the solution with integer label *lab*.
bd	A bifurcation data cell array 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' slot function; see Section 2.4. In addition, all names of continuation parameters and parameters associated with monitor functions can be used.

5 Utility Functions

5.1 Toolbox Data Structures with Shared Fields

Sometimes it is necessary to allow different functions from a toolbox to modify a shared copy of the toolbox data structure. COCO's way of making this possible is to make function data an instance of coco_func_data. Any structure may be converted to coco_func_data and, when doing so, all fields that were present at the time of conversion, or that will be added subsequently, are shared between copies of this instance.

`data = coco_func_data(data);`	
`data`	On output: An instance of `coco_func_data`. After this conversion, the fields of `data` can be accessed using Matlab's usual methods of structure field access. If one creates a copy of `data`, for example, `d2=data`, then assignments to `d2` will affect `data` and vice versa. The fields `data.data`, `data.sh`, and `data.pr` have a special meaning and may not be used in simple assignments.
`data`	On input: A toolbox data structure. If `data` is already of type `coco_func_data`, noting happens (`data` is returned as is).

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 the helper functions `coco_ezDFDX` and `coco_ezDFDP` 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 Jacobian with respect to x, use

`J = coco_ezDFDX('f(x,p)', @func, x, p);`	
`J`	The Jacobian matrix $\partial f / \partial x$.
`@func`	Function handle to the encoding of the corresponding function. 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 Jacobian of a function with respect to p ,use

`J = coco_ezDFDP('f(x,p)', @func, x, p);`	
`J`	The Jacobian matrix $\partial f / \partial p$.
`@func`	Function handle to the encoding of the corresponding function. 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 .