# Frequently asked questions:

These frequently asked questions and corresponding video tutorials were produced in January 2014 and are compatible with the release and working copy of COCO available through the SourceForge repository at that time. Changes to the file names or folder structure since then are highlighted through footnotes in this transcript. The footnotes also include some minor errata.

This version is published on November 2, 2014.



## FAQ 1:

**Q. What is** COCO**?**

**A.** Welcome! So, what is COCO? Well, first of all, COCO is short for the Computational Continuation Core. Since that is a bit of a mouthful, I just say COCO.

COCO is an analysis and development platform meant for solving continuation problems. In the most abstract sense, COCO allows you to compute implicitly defined embedded submanifolds. In the most concrete sense, and the one that you will typically encounter, COCO allows you to construct finite-dimensional systems of nonlinear algebraic equations and to describe their solutions.

COCO is built on top of MATLAB, so all of MATLAB's functionality is immediately available to you. You interact with COCO in two ways: through a set of utility functions that adhere to regular MATLAB syntax, as well as through a set of functions that encode your continuation problem. I recommend that you use MATLAB's debugging environment for testing the execution of any function that you author.

A core feature of COCO is its object-oriented view of problem construction and analysis. A major benefit of the object-oriented paradigm is modularity of construction and reusability of code. Another benefit is the decoupling of the task of constructing a continuation problem from the task of describing its solutions using covering algorithms. Here, again, COCO's adherence to an object-oriented paradigm means that you can substitute your own covering algorithms, or inherit from and modify algorithms developed by others, including those shipped with COCO.

You can read about the concept of task embedding in Parts I and II of the textbook "Recipes for Continuation". Covering algorithms are discussed in detail in Part III of "Recipes for Continuation". The book contains simple as well as advanced examples of these concepts with the hope that you'll be able to mix and match example code to serve your needs.

So, to summarize, COCO is an object-oriented development platform for constructing continuation problems and mapping out their solutions.

The instructional videos on this site are meant to guide you through key steps in the use of COCO. I hope you find them useful.

## FAQ 2

**Q. What is continuation?**

**A.** Welcome! So, what is continuation? And what is a continuation problem?

A continuation problem seeks to find the zeros of a function *Phi* under the assumption that these zeros are locally homeomorphic to some finite-dimensional real space[1].

A continuation problem may be either **finite-dimensional** or **infinite-dimensional**, depending on the dimension of the domain of the function *Phi*. When analyzed using numerical methods, an infinite-dimensional problem must be replaced by a finite-dimensional discretization, but the choice of discretization may change during the analysis.

In the finite-dimensional case, a continuation problem is said to be **closed** when the dimension of the domain of the function *Phi* equals the dimension of its image[2]. When the dimension of the domain is greater than the dimension of the image, we say that the problem has a **dimensional deficit** equal to the excess.

For continuously-differentiable functions, the **implicit-function theorem** states that there exists a locally unique and differentiable zero-manifold near every regular zero and that the manifold dimension equals the dimensional deficit.

Continuation problems may be characterized by the operational meaning played by the coordinate components that describe the domain of the function *Phi*. COCO relies heavily on such a characterization.

Suppose that the coordinate components that describe the domain have no distinguishing operational meaning. In this case, we refer to these coordinates as **continuation variables** *u*, to the components of the function *Phi* as **zero functions**, and to the continuation problem *Phi(u)=0* as a **zero problem**. The continuation variables are free to vary arbitrarily along the zero manifold, as long as such variations ensure that the components of *Phi* equal zero throughout.

---

[1] The audio narration states that this should be a real vector space, but the “vector” part should be omitted from the definition.

[2] It is more correct to refer to the dimension of the range, rather than of the image, as the former is defined by the number of components of *Phi*.

We obtain an **extended continuation problem** by introducing a set of **monitor functions** *Psi* on the domain of *Phi* and by appending the expressions *Psi(u)-mu* to the function defining the zero problem. Here, the components of *mu* are called **continuation parameters**. Together with the continuation variables, the continuation parameters describe the domain of the extended continuation problem. The extended continuation problem has the same dimensional deficit as the original zero problem, but the embedding of the zero manifold in the new domain is of course different.

There are two advantages to using the extended continuation problem. One lies in the opportunity to **detect and locate special points** on the zero manifold, points associated with particular values of some component of *Psi*.

The second advantage lies in the ease with which we can **impose additional constraints** on the continuation variables that must be satisfied along the corresponding zero manifold. We do this simply by insisting that a subset of the continuation parameters not be allowed to vary, and refer to the corresponding elements of *mu* as **inactive continuation parameters**. The other elements of *mu* are then referred to as **active continuation parameters**.

We can make this decision at the time of construction or at run-time, long after the construction of the continuation problem. We refer to the continuation problem obtained after the imposition of such a restriction as a **restricted continuation problem**. The new zero manifold is obtained by intersection with a collection of hyperplanes corresponding to fixed values of the inactive continuation parameters followed by projection onto the continuation variables and the active continuation parameters.

If we project the zero manifold of the restricted continuation problem onto the space spanned by the continuation variables, then we obtain the zero manifold of the **reduced continuation problem**. This is a submanifold of the zero manifold of the original zero problem.

So, to summarize, a continuation problem is a system of equations for finding the zeros of some function on a finite- or infinite-dimensional domain. By extending the continuation problem with monitor functions and continuation parameters, the tasks of locating special points on the zero manifold and of restricting the analysis to constrained submanifolds can be handled long after the construction of the continuation problem.

To become a proficient analyst, it helps to visualize simple examples of these concepts and to make frequent use of the different terms.

Let's take a look at a couple of simple examples from Section 2.2.5 of "Recipes for Continuation". We start with the zero function *Phi* in terms of *u1* and *u2*. The zero manifold is the circle

*u1=cos(theta)*, *u2=1+sin(theta)*. The zero manifold of the extended continuation problem obtained by appending the equation

*u1^2+u2^2-mu1=0*

is the curve *u1=cos(theta)*, *u2=1+sin(theta)*, and *mu1=2+2sin(theta)*. The zero manifold of the restricted continuation problem obtained by fixing the value of *mu1* to *2* consists of the points *(u1,u2)=(1,1)* and *(-1,1)*. In this case, the corresponding reduced continuation problem is identical to the restricted continuation problem.

Suppose, instead, that we start with the zero function *Phi* in terms of *u1*, *u2*, and *u3*. The zero manifold is the cylinder *u1=cos(theta)*, *u2=1+sin(theta)*, and *u3=z*. The zero manifold of the extended continuation problem obtained by introducing three monitor functions and the corresponding continuation parameters is the surface *u1=cos(theta)*, *u2=1+sin(theta)*, *u3=z*, *mu1=Sqrt(2+2sin(theta))*, *mu2=cos(theta)*, and *mu3=z-cos(theta)+0.5*. The zero manifold of the restricted continuation problem obtained by fixing the value of *mu3* to -0.5 is the curve *u1=cos(theta)*, *u2=1+sin(theta)*, *u3=cos(theta)-1*, *mu1=Sqrt(2+2sin(theta))*, and *mu2=cos(theta)*. Its projection onto the space spanned by the continuation variables is the zero manifold of the reduced continuation problem obtained by omitting the second and third equations.

These examples illustrate the differences between the domains of the original zero problem, of its extension in an extended continuation problem, of a corresponding restricted continuation problem, and of the equivalent reduced continuation problem.

## FAQ 3

**Q. How do I install COCO?**

**A.** Welcome! In this video, we illustrate the steps required to either download and install the most recent release of COCO, or check out and install a version-controlled working copy of COCO.

To follow along with this video, you need an internet-connected computer with a current MATLAB license. While we guarantee full support for COCO running on MATLAB versions 2009b-2011b, the functionality described in this and other instructional videos is compatible with version 2012b. This is also the platform used in these videos. To check out a version-controlled copy of COCO, make sure that your computer also provides you with a command-line or graphical user interface to your directory structure, with support for the Subversion command.

Let's start with showing you how to download and install the most recent release of COCO. If you want to check out and install a version-controlled working copy of COCO, skip to the corresponding section of this video.

The most recent release of COCO is stored in a compressed package file in a SourceForge repository. To access this file, open an Internet browser window. Type sourceforge.net/projects/cocotools/files/latest/download into the address bar and press the Enter key on your keyboard.

Depending on your browser settings, the compressed package file may be automatically saved to a designated folder on your disk, or a dialog box will open up on your screen.

**Windows:** This gives you the option of opening the compressed package file using a specific application, or saving the file to disk.

**Mac:** This gives you the option of saving the file to disk.

For your information, the name of the compressed package file includes the date of the release. Save the file to a location of your choice and make a note of this location.

In a directory browser, navigate to the location containing the compressed package file. If this has not been automatically decompressed, use a decompression routine to unpack the file. By default, this process creates a sub-directory named `coco`. This directory contains all the files associated with the release, including a startup script file named `startup.m`. Make a note of the full path to this file, for later use.

Next, switch to MATLAB. When MATLAB starts up, its working directory defaults to the MATLAB startup directory. If your working directory differs from the startup directory, enter the command "userpath" on the command line to determine the path to the startup directory. Then, enter the command "cd" followed by this path on the command line.

Next, enter the command "edit" followed by "startup.m" on the command line. If the MATLAB startup directory contains a file named `startup.m`, this action opens the file using the MATLAB editor. If no such file exists, a MATLAB editor dialog box appears asking you to confirm that you wish to create a new file with this name. In this case, click on the "Yes" button. An empty file with the name `startup.m` is automatically added to the directory and opened in the MATLAB editor for further editing. Add a line to the `startup.m` file in the MATLAB editor containing the command "run" followed by the full path to the `startup.m` file in the `coco` directory. Then, save your changes.

Finally, quit MATLAB. The startup script in the `coco` directory will be automatically executed every time MATLAB is started. This ensures that COCO and all associated toolboxes are in the MATLAB search path.

Let's continue by showing you how to check out and install a working copy of COCO from a version-controlled SourceForge repository. This approach allows you to regularly update your local COCO installation by monitoring changes to the source code and making the appropriate file substitutions.

**Windows:** In this video, the graphical user Subversion interface TortoiseSVN is accessed through the directory browser context menu.

**Mac:** In this video, we access the svn command using the Terminal application.

**Windows:** Open a directory browser window and navigate to a location of your choice. Create a sub-directory for holding the working copy of COCO. Next, right-click on this folder to open the context menu! To open the TortoiseSVN checkout dialog box, click on SVN Checkout.

**Mac:** Open a terminal window and navigate to a location of your choice. Create a sub-directory for holding the working copy of COCO.

Next, open an Internet browser window. Type sourceforge.net/p/cocotools/wiki/checkout into the address bar and press the Enter key. This webpage contains a list of subversion checkout commands.

**Windows:** Copy the source URL, associated with the first checkout command, and paste this into the repository URL textbox in the TortoiseSVN Checkout dialog box. Copy the target directory path, associated with the first checkout command, and append this to the Checkout directory path in the TortoiseSVN Checkout dialog box. Click OK.

**Mac:** Copy the first checkout command and paste this into the terminal window. Click the Enter button on your keyboard.

Repeat these steps with each of the checkout commands to check out a working copy of COCO and associated toolboxes into your target directory. This process stores a startup script file named `startup.m` in the subdirectory `toolbox/core/admin`. Make a note of the full path to this file, for later use.

Next, switch to MATLAB. When MATLAB starts up, its working directory defaults to the MATLAB startup directory. If your working directory differs from the startup directory, enter the command "userpath" on the command line to determine the path to the startup directory. Then, enter the command "cd" followed by this path on the command line.

Next, enter the command "edit" followed by "startup.m" on the command line. If the MATLAB startup directory contains a file named `startup.m`, this action opens the file using the MATLAB editor. If no such file exists, a MATLAB editor dialog box appears asking you to confirm that you wish to create a new file with this name. In this case, click on the "Yes" button. An empty file with the

name `startup.m` is automatically added to the directory and opened in the MATLAB editor for further editing. Add a line to the `startup.m` file in the MATLAB editor containing the command "run" followed by the full path to the `startup.m` file in the `toolbox/core/admin` directory. Then, save your changes.

Finally, quit MATLAB. The startup script in the `coco` directory will be automatically executed every time MATLAB is started. This ensures that COCO and all associated toolboxes are in the MATLAB search path.

**Windows:** From time to time, you may wish to update your working copy of COCO. To do this with TortoiseSVN, open a directory browser window, and navigate to the directory holding the working copy of COCO. To update a version-controlled folder inside this directory, right-click on the folder to open the context menu, and click on SVN Update.

**Mac:** From time to time, you may wish to update your working copy of COCO. To do this, open a terminal window, and navigate to the directory holding the working copy of COCO. To update a version-controlled folder inside this directory, enter svn update on the command line.

With the addition of new toolboxes, additional lines will be added to the list of checkout commands. The complete list can be found at sourceforge.net/p/cocotools/wiki/checkout. Please remember to visit often!

## FAQ 4

**Q. What is coco_prob?**

**A.** Welcome! So, what is coco_prob? And what is a continuation problem structure?

In this video, we illustrate the steps required to initialize the data structure used by COCO to encode a continuation problem.

To follow along with this video, you need a computer with a current MATLAB license. While we guarantee full support for COCO running on MATLAB versions 2009b-2011b, the functionality described in this and other instructional videos is compatible with version 2012b. This is also the platform used in these videos. You should also review the terminology and basic definitions introduced in Chapter 2 and in the beginning of Chapter 3 of the book "Recipes for Continuation".

Switch to MATLAB and navigate to your preferred working directory. In COCO, the information used to encode a restricted continuation problem and to explore its solution manifold is stored in a continuation problem structure. To initialize an empty continuation problem structure, assign the output of the core utility coco_prob to a MATLAB variable. For example, if the MATLAB variable that

will hold the continuation problem structure is named 'prob', enter "prob=coco_prob" followed[3] by a semicolon on the command line. The MATLAB variable 'prob' is now added to the MATLAB workspace. This variable is a single element of type `struct`.

In this video, we illustrated the steps required to initialize an empty continuation problem structure. You may also wish to view the video on "What is coco_add_func?" or the video on "What is coco_set?"

## FAQ 5

**Q. What is a COCO-compatible function file?**

**A.** Welcome! So, what is a COCO-compatible function file? And how does one encode zero or monitor functions for analysis by COCO?

In this video, we illustrate the syntax requirements and a general paradigm for encoding a collection of zero or monitor functions in a COCO-compatible MATLAB file. To follow along with this video, you need a computer with a current MATLAB license. While we guarantee full support for COCO running on MATLAB versions 2009b-2011b, the functionality described in this and other instructional videos is compatible with version 2012b. This is also the platform used in these videos. You should also review the terminology and basic definitions introduced in Chapter 2 and in the beginning of Chapter 3 of the book "Recipes for Continuation".

Switch to MATLAB and navigate to your preferred working directory. Enter "edit" followed by "coco_funcfile" on the command line[4]. This action opens the template file `coco_funcfile.m` using the MATLAB editor. Remember to save a copy of this file to your working directory before making any changes.

The template file contains several example encodings. These may be used to guide your own code development. It is good practice to include some comments at the beginning of the file explaining the origin of the encoding and the use of the input arguments.

The function definition syntax is the required format for the first line in the template file. This is what defines the encoding to be compatible with COCO. As seen in the function definition, a COCO-compatible encoding has three input arguments and two output arguments. At the conclusion of execution, the second output argument contains a vector[5] of numerical values obtained by evaluating each of the zero or monitor functions. In a call to a COCO-compatible encoding by the

---

[3] In "Recipes for Continuation", calls to coco_prob are given in the form coco_prob(), but the output is the same as when the empty parentheses are omitted.

[4] The `coco_funcfile.m` is located at the top level of the `coco` directory created from a current release. Copy this file to your working directory before opening the file in the MATLAB editor.

[5] This should better be referred to as an array.

COCO core, the first input argument contains a continuation problem structure initialized by the coco_prob utility. The second input argument contains the function data structure. The content of this variable may be modified in the function body and is returned in the first output argument. The third input argument contains a vector[6] of numerical values of a subset of the continuation variables associated with the function dependency index set. The function data structure and the dependency index set are initialized in the call to the coco_add_func constructor.

Look at the first example encoding in the template file. This demonstrates a direct association between a functional expression in terms of the components of the input argument 'u', on the one hand, and the return argument 'y', on the other. Here, the first three components of 'u' are used to calculate a value for 'y'.

For more sophisticated applications, the functional expression may be replaced by the output of some algorithm. This algorithm may be encoded within the same function, or in a separate function. It may include conditional and flow control statements, just like in any MATLAB function. Look at the second example encoding in the template file. This obtains a value for the output argument 'y' from a difference calculation involving the application of the MATLAB quad algorithm. The encoding also demonstrates the use of fields of the function data structure to parameterize the execution of the algorithm.

It is often useful to provide for an encoding for a particular class of zero or monitor functions. Such an encoding relies heavily on a parameterization stored in the function data structure. Look at the third example encoding in the template file. This assumes that the zero functions are encoded in a separate function, whose function handle is stored in the 'fhan' field of the function data structure. In addition, the function class assumes two distinct input arguments. The 'x_idx' and 'p_idx' index sets are used here to extract the corresponding elements from the 'u' input argument.

Changes to the function data structure within the encoding allow for an adaptive response to changes in its execution between successive calls. In the fourth example encoding in the template file, the 'opts' field of the function data structure parameterizes the execution of the function whose handle is stored in 'data.fhan'. Modifications to this parameterization survive execution and affect the execution of this function in any subsequent call.

In this video, we illustrated the basic principles and syntax requirements for encoding zero or monitor functions in a COCO-compatible MATLAB file. You may also wish to view the video on "What is coco_prob?" or the video on "What is coco_add_func?"

## FAQ 6

**Q. How do I encode Jacobians?**

[6] See previous footnote.

**A.** Welcome! So, how do you encode Jacobians? And is it really necessary to do so?

In this video, we illustrate the syntax requirements for encoding the Jacobian of a collection of zero or monitor functions in a COCO-compatible MATLAB file. To follow along with this video, you need a computer with a current MATLAB license. While we guarantee full support for COCO running on MATLAB versions 2009b-2011b, the functionality described in this and other instructional videos is compatible with version 2012b. This is also the platform used in these videos. You should also review the terminology and basic definitions introduced in Chapter 2 and in the beginning of Chapter 3 of the book "Recipes for Continuation". You may also wish to view the video "What is a COCO-compatible function file?"

Switch to MATLAB and navigate to your preferred working directory. Enter "edit" followed by "coco_funcfile" on the command line. This action opens the template file `coco_funcfile.m` using the MATLAB editor. Remember to save this file to your working directory before making any changes.

The template file contains several example encodings of zero functions. Delete all but the first one in "Example usage 1". This encodes two zero functions with a dependency on the first three components of the 'u' input argument. The actual number of components of the 'u' input argument, however, is not given in the encoding, but is determined by the size of the function dependency index set.

Save another copy of the edited file to your working directory, but add `_DFDU` to the file name. Change the function name accordingly. Also change the variable name of the second output argument from 'y' to 'J' to reflect the intent to have this contain the numerical value of the Jacobian matrix, rather than of the zero functions themselves.

Next, suppose that the number of components of the 'u' input argument is exactly three. It follows that the Jacobian of the zero functions with respect to the corresponding elements of the vector of continuation variables is a two by three rectangular matrix. It is good practice to initialize the output argument 'J' to a correspondingly sized zero array. Proceed to assign functional expressions to each nonzero element of the Jacobian matrix. Here, the first index refers to the corresponding zero function and the second index refers to the corresponding component of 'u'.

During development, it is often useful to verify the encoding of the Jacobian by a comparison with an approximation obtained using finite-difference methods. The core utility coco_ezDFDX provides such functionality. Specifically, an approximate Jacobian of the zero functions encoded in the function with function handle '@demo_func' is obtained in the second return argument of the following call:

```
[data Jt] = coco_ezDFDX('f(o,d,x)', prob, data, @demo_func, u);
```

With the introduction of a breakpoint in the function body, execution will pause when reaching this line, allowing a comparison between the content of the MATLAB variables 'J' and 'Jt'.

For more sophisticated applications, the functional expressions may be replaced by the output of some algorithm. This algorithm may be encoded within the same function, or in a separate function. It may include conditional and flow control statements, just like in any MATLAB function. The toolbox examples in the book "Recipes for Continuation" illustrate this generalized approach.

In this video, we illustrated the syntax requirements for encoding the Jacobian of a collection of zero or monitor functions in a COCO-compatible MATLAB file. You may also wish to view the videos on "What is a COCO-compatible function file?" and "What is coco_add_func?"

## FAQ 7

**Q. What is coco_add_func?**

**A.** Welcome! So what is coco_add_func? And what does it mean to build a continuation problem structure?

In this video, we illustrate the steps required to add a collection of zero functions to a continuation problem structure. To follow along with this video, you need a computer with a current MATLAB license. While we guarantee full support for COCO running on MATLAB versions 2009b-2011b, the functionality described in this and other instructional videos is compatible with version 2012b. This is also the platform used in these videos. You should also review the terminology and basic definitions introduced in Chapter 2 and in the beginning of Chapter 3 of the book "Recipes for Continuation". You can read about coco_add_func and its role as an object constructor in Chapter 4 of this textbook.

In this example, we assume that the MATLAB variable 'prob' in the workspace contains a continuation problem structure used by COCO to encode a continuation problem. In particular, 'prob' stores information about function handles for COCO-compatible encodings of zero and monitor functions, with associated function data structures and dependency index sets. The individual function data structures parameterize the execution of the corresponding encodings. The individual function dependency index sets identify the subset of continuation variables whose values are used during execution.

Consider the COCO-compatible encoding of a collection of zero functions in the function `demo_func1.m`. To add this to the continuation problem structure, we rely on the coco_add_func constructor. At a minimum, a call to the constructor requires that we provide the corresponding function handle, the initial content of the function data structure, and information regarding the function dependency index set. The call to the constructor must also associate a unique function identifier to this encoding.

Let's associate the function identifier 'demo1' to the collection of zero functions encoded in `demo_func1.m`. Since the function data structure is not used in the function body, this can be initialized to the empty array.

We provide information regarding the function dependency index set in two steps. First we identify any dependencies on already declared continuation variables. This is accomplished with the 'uidx' flag followed by an array of integer indices. In this example, we assume that the first two components of the 'u' input argument to `demo_func1.m` correspond to the $6^{th}$ and $9^{th}$ elements of the collection of already declared continuation variables.

We proceed by declaring additional continuation variables and the corresponding values in the initial solution guess. In this example, the third component of the 'u' input argument is a new continuation variable whose initial value is assumed to equal *-3*. The 'zero' flag in the constructor call identifies the encoding as corresponding to a collection of zero functions.

Next, consider the COCO-compatible encoding of a collection of zero functions in the function `demo_func2.m`. To add this to the continuation problem structure, we again rely on the coco_add_func constructor. As before, a call to the constructor requires that we provide the corresponding function handle, the initial content of the function data structure, and information regarding the function dependency index set. The call to the constructor must also associate a unique function identifier to this encoding.

Let's associate the function identifier 'demo2' to the collection of zero functions encoded in `demo_func2.m`. Since the function data structure is used in the function body, initial content must be provided in the call to the coco_add_func constructor. In this case, we assign a numerical value to the 'tol' field of the data structure outside of the call to coco_add_func, and include this data variable in the call to coco_add_func.

As before, we provide information regarding the function dependency index set in two steps. First we identify any dependencies on already declared continuation variables. This is accomplished with the 'uidx' flag followed by an array of integer indices. In this example, we assume that the first component of the 'u' input argument to the `demo_func2.m` function corresponds to the $2^{nd}$ element of the collection of already declared continuation variables.

We proceed by declaring additional continuation variables and the corresponding values in the initial solution guess. In this example, the second component of the 'u' input argument is a new continuation variable whose initial value is assumed to equal *1*. The 'zero' flag in the constructor call identifies the encoding as corresponding to a collection of zero functions.

The coco_add_func constructor may also be used to add a collection of monitor functions to the continuation problem structure. In this case, the 'zero' flag must be replaced by either of the flags 'active', 'inactive', or 'internal' for embedded monitor functions, and 'regular' or 'singular' for

nonembedded monitor functions. In addition, it is necessary to introduce string labels for each of the continuation parameters associated with the component monitor functions.

The coco_add_func constructor may also be used to associate an encoding of the Jacobian of a collection of zero or monitor functions to the corresponding encoding of these functions. In this case, the function handle to the encoding of the Jacobian is included as an additional argument following the function handle to the encoding of the zero or monitor functions, and before the function data structure.

Finally, in more sophisticated applications, the coco_add_func constructor may be used to provide a guess for components of the initial tangent vector to the solution manifold. In this case, we add the 't0' flag followed by a numerical array to the list of arguments.

In this video, we illustrated the steps required to add a collection of zero functions to a continuation problem structure. You may also wish to view the videos on "What is a COCO-compatible function file?" and "What are coco_add_pars and coco_add_glue?"

## FAQ 8

**Q. What are coco_add_pars and coco_add_glue?**

**A.** Welcome! So, what are coco_add_pars and coco_add_glue? And how do they differ from coco_add_func?

In this video, we demonstrate the encoding of special-purpose wrappers that enclose one or several calls to the core constructor coco_add_func. To follow along with this video, you need a computer with a current MATLAB license. While we guarantee full support for COCO running on MATLAB versions 2009b-2011b, the functionality described in this and other instructional videos is compatible with version 2012b. This is also the platform used in these videos. You should also review the material on special-purpose wrappers in Chapter 3 of the book "Recipes for Continuation", with emphasis on Example 3.13 in this textbook.

Switch to MATLAB and navigate to your preferred working directory. On the command line, enter "copyfile" followed by the path to the `henon` folder[7] in the `coco\examples\recipes` subdirectory[8]. This action copies the files from this folder to your working directory. Open the files `demo_chap3_v3.m`, `hen.m`, `period_A.m`, and `period_B.m` in the MATLAB editor and explore their content.

---

[7] In newer releases, the content of this folder is in the `cmds_demo` folder of the recipes subdirectory. The file `hen.m` is renamed `henon.m` (as in "Recipes for Continuation") and `demo_chap3_v3.m` is renamed `demo_henon.m`.

[8] Or in the corresponding folder of a checked out working copy of COCO.

In this example, we seek to continue period-n orbits of the two-dimensional Henon map, described in terms of the system parameters *a* and *b*. Since we are looking for fixed points of iterates of this map, it is convenient to introduce the function *f* with domain in *R^6* and image in *R^2*. Here, the first two arguments represent the system parameters and the remaining arguments the coordinates of a point in the domain of the Henon map and the corresponding point in its image. As an example, a fixed point of the Henon map corresponds to a zero of the function *f* with *z3=z5* and *z4=z6*.

A COCO-compatible encoding of the function f is given in the function `hen.m`[9] in your working directory. We add some comments to the beginning of this file to explain its origin.

For *n>=2,* period-*n* orbits of the Henon map correspond[10] to zeros of the zero problem *Phi(u)=0*, where *u* is a *2n+2*-dimensional vector of continuation variables. We note that each new row in the zero problem depends partially on elements of *u* that have been introduced in previous rows and partially on new elements of *u* that have yet to be declared. Accordingly, it is natural to decompose the construction of this zero problem following the principles described in "Recipes for Continuation", by a series of distinct calls to the coco_add_func constructor. A candidate encoding of this zero problem is given in the function period_A.m in your working directory. The mixed use of the 'uidx' and 'u0' flags in the calls to coco_add_func reflects this decomposition. We add some comments to the beginning of this file to explain its origin. We note, in particular, that the 'u0' input argument needs to contain a vector with *2n+2* components.

For *n>=1*, period-*n* orbits of the Henon map may be obtained alternatively from the zeros of this modified zero problem, where *u* is a *2n+4*-dimensional vector of continuation variables. This differs from the previous formulation in that the calls to the function *f* are all of the same form. The two final rows in the zero problem correspond to gluing conditions that each impose the constraint that two continuation variables be equal along the zero manifold.

The special-purpose wrapper coco_add_glue provides a convenient implementation of one or several gluing conditions. A candidate encoding of the modified zero problem that relies on coco_add_glue is shown in the function period_B.m in your working directory. We note, in particular, that the 'u0' input argument needs to contain a vector with *2n+4* components, where the values of the last two components should equal the values of the third and fourth components.

Using either of the two encodings of the zero problem, the MATLAB commands in the `demo_chap3_v3.m` file[11] demonstrate the use of the coco_add_pars special-purpose wrapper to add two monitor functions whose values equal the values of the system parameters. The array following the function identifier 'pars' provides the integer indices of the corresponding elements of

[9] Function is renamed `henon.m`; see previous footnote.
[10] This is also true for *n=1*, but with a redundant encoding of the fixed point.
[11] Function is renamed demo_henon.m; see earlier footnote.

the vector of continuation variables. The string labels 'a' and 'b' are here used to identify the corresponding continuation parameters.

In this video, we described mechanisms for enclosing one or several calls to the core constructor coco_add_func within a special-purpose wrapper. We also gave examples of the use of the special-purpose wrappers coco_add_pars and coco_add_glue. You may also wish to view the videos on "What is a COCO-compatible function file?" and "What is coco_add_func?"

## FAQ 9

**Q. What is coco_get_func_data?**

**A.** Welcome! In this video, we demonstrate the use of the core utility coco_get_func_data to extract information from a continuation problem structure about a collection of zero or monitor functions encoded therein. We explain the nature of embeddable constructors and how coco_get_func_data provides access to context-dependent information during construction.

To follow along with this video, you need a computer with a current MATLAB license. While we guarantee full support for COCO running on MATLAB versions 2009b-2011b, the functionality described in this and other instructional videos is compatible with version 2012b. This is also the platform used in these videos. It's a good idea to review the material on function data in Chapter 3 of the book "Recipes for Continuation", as well as the discussion of embeddable constructors in Section 4.2 of this textbook.

A constructor is said to be **embeddable** if it satisfies two conditions: first, its purpose[12] is to append zero functions, monitor functions, and elements of the initial solution guess to a given continuation problem structure and second, its design makes no context-independent assumptions about the initial content of this continuation problem structure.

The coco_add_func core constructor is an embeddable constructor, since its design makes no context-independent assumptions about the content of the continuation problem structure in its first input argument. Specifically, any overlap between the function dependency index set and the indices of already declared continuation variables must be provided to coco_add_func through explicit use of the 'uidx' flag. The associated array of integer indices provides context-dependent information to coco_add_func. By the same token, the special-purpose wrappers, coco_add_pars and coco_add_glue are also embeddable constructors.

If a constructor relies on context-dependent information that is not available a priori, such information may be obtained using the coco_get_func_data utility. As an example, consider the function my_add_pars. Since the function encloses a call to the embeddable constructor

[12] We may generalize the notion of embeddable to include any action that appends information to an existing continuation problem structure without context-independent assumptions.

coco_add_pars in order to modify the content of the continuation problem structure 'prob', it is also a constructor. Notably, the function body includes no hard-coded context-independent information about the initial content of the continuation problem structure. Context-dependent information is instead obtained using the coco_get_func_data utility. The function my_add_pars is therefore an embeddable constructor.

As described in the comments at the beginning of the encoding, the purpose of the my_add_pars constructor is twofold: i) to append monitor functions that evaluate to a subset of the previously declared continuation variables, and ii) to declare the corresponding continuation parameters as initially inactive. In the call to coco_get_func_data, the second input argument is a function identifier that is assumed to be stored in the continuation problem structure prob. The optional flags 'data' and 'uidx' indicate that the coco_get_func_data utility should return the corresponding function data structure and function dependency index set. The subset of the latter indexed by the 'pars' field of the function data structure is subsequently associated with a set of inactive continuation parameters identified by string labels obtained from the call to the coco_get_def_par_names utility.

We proceed to illustrate the use of the my_add_pars constructor. First, consider the COCO-compatible wrapper my_funcfile for a COCO-incompatible encoding of zero or monitor functions that take two input arguments. Here, index sets stored in the function data structure are used to extract subsets of the input argument 'u' that constitute the input arguments of the function whose handle is stored in the 'fhan' field of the function data structure.

In this example, we use an inline definition of a function of two variables *x* and *p* and assign its handle to the 'fhan' field of the data variable. This information is associated with the 'cusp' function identifier in the subsequent call to coco_add_func. Finally, the call to my_add_pars introduces two additional monitor functions that evaluate to the second and third continuation variables associated with the corresponding function dependency index set. The call further assigns the string labels 'cusp(1)' and 'cusp(2)' to the corresponding continuation parameters.

So, to summarize, we support the object-oriented paradigm of COCO by relying on embeddable constructors, whose design makes no context-independent assumptions about the initial content of the continuation problem structure. In this video, we demonstrated the use of the coco_get_func_data utility to provide context-dependent information to an embeddable constructor, ensuring its intended application independently of the order of construction. For further information, you may wish to view the videos "What is coco_add_func?" and "What is coco_add_pars and coco_add_glue?" as well as the general introduction "What is COCO?"

## FAQ 10

**Q. What are COCO toolboxes?**

**A.** Welcome! From the video "What is COCO?" we recall that COCO is a development platform that takes an object-oriented view of problem construction and analysis. This viewpoint helps to interpret the structure of COCO toolboxes, a number of which are developed in the textbook "Recipes for Continuation". The basic tenets of the object-oriented design paradigm are described in Sections 4.3 and 5.1 of this book. In this video, we review this paradigm in the context of the 'alg' toolbox, as developed in Chapters 4, 5, 16, and 17 of "Recipes for Continuation".

To follow along with this video, you need a computer with a current MATLAB license. While we guarantee full support for COCO running on MATLAB versions 2009b-2011b, the functionality described in this and other instructional videos is compatible with version 2012b. This is also the platform used in these videos.

A COCO toolbox is a definition of one or several related abstract classes for construction and analysis of continuation problems. A COCO toolbox is distinguished by its implementation of zero and/or monitor functions, by its encoding of slot functions responding to signals emitted during continuation, by its definition of special points and associated event-handlers, and by auxiliary utility functions. A COCO toolbox encodes one or several embeddable constructors associated with different calling syntaxes or distinct continuation problem objects. A unique toolbox instance identifier distinguishes each instance of a COCO toolbox. The dynamic state of the toolbox instance during continuation is parameterized by the content of its toolbox data: in other words, the function data structures of its constituent zero functions, monitor functions, slot functions, and event handlers.

Toolbox documentation should begin by describing the mathematical class of problems for which a toolbox is designed. It should also include a description of the toolbox interface: the format and expected content of the arguments to any of the toolbox constructors; the interpretation and identifiers of toolbox-specific zero and monitor functions; the interpretation and context-independent integer indices of continuation variables contributed by the toolbox; the default status and string labels of continuation parameters introduced by the toolbox; the interpretation and default values of any toolbox settings; and the interpretation of toolbox-specific content stored to disk during continuation.

The 'alg' toolbox implements a class of continuation problems for which the solution manifold is given by the zeros of a user-defined function Phi of two sets of variables referred to respectively as the problem variables and the problem parameters. The first instance of an embeddable version of the 'alg' toolbox occurs in Section 4.2.1 of "Recipes for Continuation" and is found in the alg_v5 folder in `coco/examples/recipes`. The COCO-compatible encodings of the collection of zero functions and their Jacobian in 'alg_F' and 'alg_DFDU' rely on integer index sets stored in the 'x_idx' and 'p_idx' fields of the function data structure. These are used to extract numerical values of the problem variables and the problem parameters from the 'u' input argument. A handle to the user-defined encoding of *Phi* is assumed to be stored in the 'fhan' field of the function data structure. The implementation allows for an optional encoding of the Jacobians of Phi with respect to the problem

variables and problem parameters, respectively, in the 'dfdxhan' and 'dfdphan' fields of the function data structure.

The initial content of the function data structure is assigned in the call to the coco_add_func constructor in the 'alg_construct_eqn' toolbox constructor. The 'prob' input argument of this constructor is an encoding of an arbitrary continuation problem structure. The implementation does not depend on the initial content of this continuation problem structure: the constructor may consequently be applied to an empty continuation problem structure.

It is assumed that the 'data' input argument to the toolbox constructor is a MATLAB variable of type `struct` with all required fields appropriately populated. The toolbox definition allows for an optional encoding of strings in the 'pnames' field of the function data structure. When this is present, the constructor appends a collection of monitor functions that evaluate to the problem parameters. The corresponding continuation parameters are initially inactive and associated with the string labels given by the components of the 'pnames' field.

The 'tbid' input argument is a string representing a unique toolbox instance identifier. This is also the identifier associated with the zero functions encoded in alg_F. By appending the string 'pars' to the toolbox instance identifier we obtain the function identifier associated with the collection of monitor functions added to the continuation problem structure by the call to coco_add_pars.

Finally, the 'sol' input argument to the alg_construct_eqn constructor is a MATLAB variable of type `struct` whose 'u' field contains an initial solution guess for the combined vector of problem variables and problem parameters.

The call to the coco_add_slot utility ensures that the coco_save_data slot function is triggered during continuation in response to the 'save_full' signal. This implies that the content of the associated function data structure is saved with each solution file stored to disk during continuation. The alg_read_solution utility demonstrates the use of this information for extracting the values of the problem variables and the problem parameters corresponding to a point on the solution manifold associated with a particular solution file.

The alg_isol2eqn and alg_sol2eqn constructors implement a COCO-specific argument parsing mechanism for assigning appropriate content to the arguments of alg_construct_eqn. These constructors, in turn, rely on the toolbox utility functions alg_arg_check, alg_init_data, and alg_read_solution.

The modification to the 'alg' toolbox in Section 5.3.1 of "Recipes for Continuation" is found in the alg_v6 folder in `coco/examples/recipes`. The toolbox definition demonstrates the use of toolbox settings and the introduction of the toolbox utility function alg_get_settings. In particular, it is assumed that the 'alg' field of the data input argument to the alg_construct_eqn toolbox constructor contains a Boolean value in the 'norm' subfield. A value of 'true' triggers a sequence of

actions by the constructor. These append the additional slot function alg_bddat to the continuation problem structure and associate this function with the 'bddat' signal emitted during continuation. The function ensures that the Euclidean norm of the vector of problem variables is added to the return argument from the coco entry-point function and to the `bd.mat` file stored to disk.

We make additional use of toolbox settings in the modified version of the 'alg' toolbox in Section 16.2.2 of "Recipes for Continuation", found in the alg_v7 folder in `coco/examples/recipes`. Here, the values of 'active' or 'regular' for the 'FO' setting trigger the construction of a further extended continuation problem with an additional monitor function of function type given by the value of the 'FO' setting. The string obtained by appending 'test.FO' to the toolbox instance identifier is assigned as both function identifier for the monitor function and string label for the corresponding continuation parameter. The encodings of the monitor function and its Jacobian in alg_fold, and alg_fold_DFDU rely on the utility functions alg_fhan_DFDX and alg_fhan_DFDP for computing the Jacobians of Phi with respect to the problem variables and problem parameters, respectively. The definition ensures that a simple zero-crossing of the monitor function is implied by passage through a simple geometric fold along a curve segment on the solution manifold. The charge to detect and locate such a zero crossing is encoded in the continuation problem structure by the call to the coco_add_event utility. Here, the slot function alg_update reparameterizes the toolbox data structure after each successful step of continuation, in support of the successful application of the linear solver in the encoding of alg_fold.

The modification to the 'alg' toolbox in Section 17.1.1 is found in the alg_v9 folder in `coco/examples/recipes`. Here, a value of 'true' for the 'HB' toolbox setting is used to trigger the construction of an even further extended continuation problem. In this case, the constructor appends an additional monitor function of function type 'regular'. The string obtained by appending 'test.HB' to the toolbox instance identifier is assigned as both function identifier for the monitor function and string label for the corresponding continuation parameter. The encoding in alg_hopf ensures that a simple zero-crossing of the monitor function is implied by the crossing of the imaginary axis by a complex conjugate pair of eigenvalues of the Jacobian of the zero functions with respect to the problem variables at a Hopf bifurcation point. The charge to detect and locate such a zero crossing is encoded in the continuation problem structure by the call to the coco_add_event utility.

The final version of the 'alg' toolbox in Section 17.1.3 of "Recipes for Continuation" is found in the alg_v10 folder in `coco/examples/recipes`. The toolbox definition assigns the event-handler alg_evhan_HB to the special point associated with a zero-crossing of the alg_hopf monitor function. The modifications to the calling syntax to the coco_add_func constructor, as well as the alg_hopf encoding demonstrate the use of chart data to store an array of the eigenvalues of the Jacobian of *Phi* with respect to the problem variables. The event-handler relies on this array in order to distinguish between zero crossings of the monitor function associated with so-called neutral saddles, as opposed to Hopf bifurcation points.

At long last, we close this review of the object-oriented paradigm relied upon in the design of COCO toolboxes. Each of the template toolboxes in Part II of “Recipes for Continuation” adheres to this paradigm and implements toolbox-specific features to illustrate the versatility of the COCO platform. The further modifications in Parts IV and V of the textbook provide support for event handling and adaptive reconstruction of toolbox objects during continuation. The template toolboxes are meant to inspire and instruct you in the design of production-ready toolboxes that encode comprehensive sets of constructors particular to each problem class. I hope you take me up on the challenge!