Nonlinear Parameter Continuation with COCO

Lecture given during
Advanced Summer School on
Continuation Methods for Nonlinear Problems

Harry Dankowicz

Department of Mechanical Science and Engineering
University of Illinois at Urbana-Champaign

August 13-24, 2018

Outline

Aphorisms

- Extend, don't reduce!
- If your problem doesn't have a solution, solving for it may be the definition of insanity!
- COCO doesn't solve problems, it creates them!
- If your code doesn't work (at all or as expected), don't blame the problem, blame your code!
- If your code doesn't work as expected, don't start by changing parameter settings. Ask why!

Coding paradigm

The philosophy implemented in COCO is one of differentiating between the construction of a continuation problem and the analysis of its solutions. First build a problem, then solve it!

This means that ideas about starting and restarting continuation do not immediately apply to COCO (although this insight didn't get fully internalized even to the developers until late in the game).

Construction puts upper limits on the dimension of the solution manifold, but the complete dimension is not known until the last zero function and monitor function has been added.

Coding paradigm

COCO is intentionally modular. This means that you, as a user or developer, can add to and expand the COCO universe of functionality without having to touch parts that others depend on.

You can trust the core to remain stable and backward compatible (as long as I'm in the game). Similarly, if you build a toolbox, don't put out a final release until you have defined an interface that will remain backward compatible. You can extend and refine, but tell others what to expect!

Code is a bit like an old 78 RPM phonograph recording. Without a Victrola, it's just a piece of bakelite.

Coding paradigm

By decomposing a problem into its constituent parts, COCO encourages you to build reusable tools and components. More sophisticated functionality (slot functions, event handlers, adaptation) can be added at later stages without wreaking havoc to basic behaviors.

As you realize further uses for your tools, additional constructors can be added to assemble input information into the data required to construct a function instance. At some point, code that appears in multiple constructors can be brought out into a stand-alone function. Code that you don't want a user to call directly can be shielded in a private folder.

Toolbox projects

Toolboxes not discussed yet in this course: `dft`, `calcvar`, `multishoot`, `dae`, `dde`, `atlas_kd_par`, `fem`, `fp`, ... yours? Some ideas for practice can be found in Chapter 21 of *Recipes*

Functionality not implemented (yet): higher-order numerical differentiation, automatic differentiation, GUIs, normal forms, continuation of codimension-2 bifurcations, error estimation for variational equations.

COCO...CO stands for continuation, core, construction, collocation, covering, composition, collaboration!

Intellectual property

When referencing COCO in a publication, you should cite *Recipes for Continuation* as the authoritative source for the general platform. If you use more recent functionality (optimization), cite the April 2018 SIADS paper. The Sourceforge repository can be cited as a location for the code and tutorial documentation, but you would need to include a "last accessed date".

When disseminating your own COCO-compatible code, specify the release of COCO that this has been tested with. If you want to be certain to receive updates on any bug fixes to COCO sign up for the COCO mailing list, or check the Sourceforge Wiki.

The COCO license is the GPLv3:
https://www.gnu.org/licenses/gpl-3.0.en.html