Core Constructors and Utilities

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

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

March 22, 2020

Contents

1 Introduction

The software package COCO is the result of joint development since 2007 by Harry Dankowicz and Frank Schilder, and, since 2016, Mingwu Li, with additional contributions from Michael E. Henderson, Erika Fotsch, and Yuqing Wang. Helpful feedback and contributions are also acknowledged from Jan Sieber and David Barton, and a growing user community. Extensive discourse on the design philosophy and mathematical underpinnings of the COCO platform is available in *Recipes for Continuation*[1], which includes a large collection of template toolboxes and example problems.

The first official release of COCO coincided with the publication of *Recipes for Continuation* in 2013. The November 2015 release introduced fully documented, production-ready toolboxes for common forms of bifurcation analysis of equilibria and periodic orbits in dynamical systems. These provided support for continuation of

- equilibria in smooth dynamical systems (`'ep'`);
- constrained trajectory segments with independent and adaptive discretizations in autonomous and non-autonomous dynamical systems (`'coll'`); and
- single-segment periodic orbits in smooth, autonomous or non-autonomous dynamical systems, and multi-segment periodic orbits in hybrid, autonomous dynamical systems (`'po'`).

The November 2017 release made significant updates to the COCO core and library of toolboxes and demos to provide support for constrained design optimization on integro-differential boundary-value problems[2]. These updates enabled the staged construction of the adjoint equations associated with equality-constrained optimization problems, and provided support for adaptive remeshing of these equations in parallel with updates to the problem discretization of the corresponding boundary-value problems. The March 2020 release extends this functionality to also allow for finite-dimensional inequality constraints, bounding the search for extrema to an implicitly-defined feasible region[3].

The original release of COCO included a default atlas algorithm for one-dimensional solution manifolds. An implementation of Henderson's MULTIFARIO package as a COCO-compatible atlas algorithm for multi-dimensional manifolds of solutions to non-adaptive continuation problems was included as an alpha version in the November 2017 release. The March 2020 release includes the updated atlas algorithm `'atlas_kd'` for multi-dimensional solution manifolds for adaptive continuation problems (with varying embedding dimension

[1] Dankowicz, H. & Schilder, F., *Recipes for Continuation*, Society for Industrial and Applied Mathematics, 2013.

[2] Li, M. & Dankowicz, H., "Staged Construction of Adjoints for Constrained Optimization of Integro-Differential Boundary-Value Problems," *SIAM J. Applied Dynamical Systems* **17(2)**, pp. 1117–1151, 2018.

[3] Li, M. & Dankowicz, H., "Optimization with Equality and Inequality Constraints Using Parameter Continuation," *Applied Mathematics and Computation* **375**, art. no. 125058, 2020.

and variable interpretation)[4]. This is described in a separate tutorial on COCO atlas algorithms.

It has not been the intent of the COCO development to build graphical user interfaces to the methods and data invoked and processed, respectively, during analysis of a continuation problem. Some low-level data processing and visualization routines are included with the COCO core and described in this tutorial. Support for run-time access to data is available in COCO through a signal-and-slot mechanism as described in *Recipes for Continuation*.

Usage of basic COCO constructors and utilities is illustrated in several examples in this tutorial. Each example corresponds to fully documented code in the `coco/core/examples` folder in the COCO release. Slight differences between the code included below and the example implementations in `coco/core/examples` show acceptable variations in the COCO syntax and demonstrate alternative solutions to construction and analysis. To gain further insight, please run the code to generate and explore figures and screen output.

Additional detailed information about COCO utilities deployed in these examples may be found in the document "Short Developer's Reference for COCO," available in the `coco/help` folder in the COCO release, and in *Recipes for Continuation*.

2 Problem formulation

The COCO platform supports construction and analysis of continuation problems of the general form

$$\begin{pmatrix} \Phi(u) \\ \Psi(u) - \mu \\ \Lambda^\top(u)\lambda \\ \Xi(u,\lambda,v) \\ \Theta(u,\lambda,v) - \nu \end{pmatrix} = 0 \tag{1}$$

in terms of collections of *zero functions* $\Phi : \mathbb{R}^{n_u} \to \mathbb{R}^{n_\Phi}$, *monitor functions* $\Psi : \mathbb{R}^{n_u} \to \mathbb{R}^{n_\Psi}$, *adjoint functions* $\Lambda : \mathbb{R}^{n_u} \to \mathbb{R}^{n_\lambda \times n_\Lambda}$, *complementary zero functions* $\Xi : \mathbb{R}^{n_u} \times \mathbb{R}^{n_\lambda} \times \mathbb{R}^{n_v} \to \mathbb{R}^{n_\Xi}$, and *complementary monitor functions* $\Theta : \mathbb{R}^{n_u} \times \mathbb{R}^{n_\lambda} \times \mathbb{R}^{n_v} \to \mathbb{R}^{n_\Theta}$, as well as collections of *continuation variables* u, *continuation parameters* μ, *continuation multipliers* λ, *complementary continuation variables* v, and *complementary continuation parameters* ν. The vectors u, λ, and v are said to be *initialized* when they are associated with identically-sized numerical vectors u_0, λ_0, and v_0. Unless otherwise stated, the vectors μ and ν are initialized with $\Psi(u_0)$ and $\Theta(u_0, \lambda_0, v_0)$, respectively.

Consider an indexing of the elements of μ by integers in the set $\{1, \ldots, n_\Psi\}$ and of the elements of ν by integers in the set $\{n_\Psi + 1, \ldots, n_\Psi + n_\Theta\}$. Then, during continuation, the index sets $\mathbb{I}_\mu \subseteq \{1, \ldots, n_\Psi\}$ and $\mathbb{I}_\nu \subseteq \{n_\Psi + 1, \ldots, n_\Psi + n_\Theta\}$ identify continuation parameters that are fixed and not included among the unknowns whose values are defined implicitly by the corresponding *restricted* continuation problem. Continuation parameters indexed by

[4]Dankowicz, H., Wang, Y., Schilder, F. & Henderson, M.E., "Multidimensional Manifold Continuation for Adaptive Boundary-Value Problems," *J. Computational and Nonlinear Dynamics* **15(5)**, art. no. 051002, 2020.

integers in $\mathbb{I}_\mu \cup \mathbb{I}_\nu$ are said to be *inactive*. The remaining continuation parameters are said to be *active*. The *dimensional deficit* of the continuation problem is the difference between the number of unknowns and the number of equations, i.e., $n_u + n_\lambda + n_v - n_\Phi - n_\Lambda - n_\Xi - |\mathbb{I}_\mu| - |\mathbb{I}_\nu|$. When this is greater than 0, it equals the dimension of the unique solution manifold through any regular solution point.

Special cases of the general form include

- the (extended) continuation problem with $n_\Lambda = n_\Xi = n_\Theta = 0$:

$$\begin{pmatrix} \Phi(u) \\ \Psi(u) - \mu \end{pmatrix} = 0 \tag{2}$$

  for analysis of the manifold of solutions to the zero problem $\Phi(u) = 0$, possibly constrained to submanifolds associated with inactive elements of μ. In this case, the dimensional deficit reduces to $n_u - n_\Phi - |\mathbb{I}_\mu|$.

- the (augmented) continuation problem with $n_\Xi = n_v = 0$ and $n_\Theta = n_{\lambda_\Psi}$:

$$\begin{pmatrix} \Phi(u) \\ \Psi(u) - \mu \\ \Lambda_\Phi^\top(u)\lambda_\Phi + \Lambda_\Psi^\top(u)\lambda_\Psi \\ \lambda_\Psi - \nu \end{pmatrix} = 0 \tag{3}$$

  for locating stationary points of an element of Ψ along the manifold of solutions to the zero problem $\Phi(u) = 0$, possibly constrained to submanifolds associated with inactive elements of μ. Here, the adjoint functions $\Lambda_\Phi : \mathbb{R}^{n_u} \to \mathbb{R}^{n_{\lambda_\Phi}} \times \mathbb{R}^{n_\Lambda}$ and $\Lambda_\Psi : \mathbb{R}^{n_u} \to \mathbb{R}^{n_{\lambda_\Psi}} \times \mathbb{R}^{n_\Lambda}$ are related to the adjoints $(D\Phi(u))^*$ and $(D\Psi(u))^*$ of the Frechét derivatives of the zero functions and monitor functions prior to problem discretization. In this case, the dimensional deficit reduces to $n_u + n_{\lambda_\Phi} + n_{\lambda_\Psi} - n_\Phi - n_\Lambda - |\mathbb{I}_\mu| - |\mathbb{I}_\nu|$.

- the (augmented) continuation problem with $n_\Xi = n_v = 0$ and $n_\Theta = n_{\lambda_\Psi} + n_{\lambda_G}$:

$$\begin{pmatrix} \Phi(u) \\ \Psi(u) - \mu \\ \Lambda_\Phi^\top(u)\lambda_\Phi + \Lambda_\Psi^\top(u)\lambda_\Psi + \Lambda_G^\top(u)\lambda_G \\ \lambda_\Psi - \nu_{\lambda_\Psi} \\ \Theta_G(\lambda_G, -G(u)) - \nu_G \end{pmatrix} = 0 \tag{4}$$

  for locating local minima of an element of Ψ along the manifold of solutions to the zero problem $\Phi(u) = 0$, possibly constrained to submanifolds associated with inactive elements of μ, within the computational domain defined by the inequality constraints $G(u) \leq 0$ for some function $G : \mathbb{R}^{n_u} \to \mathbb{R}^{n_{\lambda_G}}$. Here, the adjoint functions additionally include a representation $\Lambda_G : \mathbb{R}^{n_u} \to \mathbb{R}^{n_{\lambda_G}} \times \mathbb{R}^{n_\Lambda}$ of the adjoint $(DG(u))^*$ of the Frechét derivative of the function G prior to problem discretization, while Θ_G represents a nonlinear complementarity function that vanishes on solutions to the complementarity conditions $0 \leq \lambda_G \perp -G(u) \geq 0$. In this case, the dimensional deficit reduces to $n_u + n_{\lambda_\Phi} + n_{\lambda_\Psi} + n_{\lambda_G} - n_\Phi - n_\Lambda - |\mathbb{I}_\mu| - |\mathbb{I}_\nu|$.

The general form of the continuation problem is implemented in COCO in one of two possible ways. In the first, and less common approach,

1. a call to the `coco_add_func` constructor defines the function Φ and initializes u;
2. a second call to `coco_add_func` defines the function Ψ, associates string labels with the elements of μ, and initializes the index set $\mathbb{I}_\mu$;
3. two consecutive calls to the `coco_add_adjt` constructor (one for each preceding call to `coco_add_func`) define the function Λ and initalize λ;
4. a call to the `coco_add_comp` constructor defines the function Ξ and initializes v;
5. a second call to `coco_add_comp` defines the function Θ, associates string labels with the elements of ν, and initializes the index set $\mathbb{I}_\nu$;
6. Following this initial construction, elements of $\mathbb{I}_\mu \cup \mathbb{I}_\nu$ may be removed without replacement in a call to the `coco` entry-point function or switched one-to-one with elements in $\{1, \ldots, n_\Psi + n_\Theta\} \setminus (\mathbb{I}_\mu \cup \mathbb{I}_\nu)$ using the `coco_xchg_pars` utility.

More commonly, the COCO constructors are used following a staged approach. At the conclusion of each stage, the COCO continuation problem structure encodes an embedded subproblem of the full continuation problem, in terms of some functions $\tilde{\Phi}$, $\tilde{\Psi}$, $\tilde{\Lambda}$, $\tilde{\Xi}$, $\tilde{\Theta}$, vectors $\tilde{u}$, $\tilde{\lambda}$, $\tilde{v}$, $\tilde{\mu}$, $\tilde{\nu}$, initial values $\tilde{u}_0$, $\tilde{\lambda}_0$, $\tilde{v}_0$, and designations $\tilde{\mathbb{I}}_\mu$ and $\tilde{\mathbb{I}}_\nu$. This staged approach to problem construction supports the development of toolboxes dedicated to appending embeddable subproblems to an existing continuation problem structure, and the subsequent coupling of subproblems using *gluing conditions*. Toolbox constructors and specialized constructors like `coco_add_pars`, `coco_add_glue`, `coco_add_functionals`, or `coco_add_complementarity` encapsulate calls to the `coco_add_func`, `coco_add_adjt`, and `coco_add_comp` constructors.

3 The Henon map – **henon**

We illustrate the formulation of a continuation problem of the form (2) in the context of continuation of period-N orbits of the Henon map[5]

$$h(x,p) := \begin{pmatrix} x_2 + 1 - ax_1^2 \\ bx_1 \end{pmatrix} \tag{5}$$

in terms of the vector of state variables $x = (x_1, x_2) \in \mathbb{R}^2$ and vector of problem parameters $p = (a, b) \in \mathbb{R}^2$. Specifically, we seek a sequence $x^{(i)}$, $i = 1, \ldots, N$, such that

$$x^{(i+1)} - h(x^{(i)}, p) = 0 \tag{6}$$

and

$$x^{(1)} - h(x^{(N)}, p) = 0. \tag{7}$$

The sequence of MATLAB commands

[5]This example was developed jointly with Jan Sieber from an earlier version in *Recipes for Continuation*.

```
>> henon = @(x,a,b) [ x(2)+1-a*x(1)^2; b*x(1) ];
>> henon_res = @(x,p,y) y-henon(x, p(1), p(2));
>> ip = 1:2;
>> ix = 3:4;
>> iy = 5:6;
>> f  = @(u) henon_res(u(ix), u(ip), u(iy));
```

define the anonymous functions `henon` to represent the map (5), `henon_res` to represent the left-hand sides of (6) and (7), and `f` to represent a corresponding zero function that depends on six elements of the vector of continuation variables.

For $p = p_0 := (1, 0.3)$, a period-4 orbit is approximated by the sequence $x^{(1)} = x_0^{(1)} := (1.3, 0)$, $x^{(2)} = x_0^{(2)} := (-0.7, 0.4)$, $x^{(3)} = x_0^{(3)} := (1.0, -0.2)$, and $x^{(4)} = x_0^{(4)} := (-0.1, 0.3)$. To continue a family of such orbits, we proceed to append four copies of the zero function f to the continuation problem structure `prob`, as shown in the following sequence of commands.

```
>> p0 = [ 1.0  0.3 ];
>> x0 = [ 1.3  0.0; -0.7  0.4; 1.0 -0.2; -0.1  0.3 ];
>> period = size(x0,1);
>> prob = coco_prob;
>> fcn  = @(f) @(p,d,u) deal(d, f(u));
>> for i=1:period
     inxt = mod(i, period)+1;
     prob = coco_add_func(prob, ['henon' num2str(i)], fcn(f), [], 'zero', ...
       'u0', [p0 x0(i,:) x0(inxt,:)]');
   end
```

Here, the anonymous function `fcn` is used to convert `f` into a COCO-compatible format with input arguments `p`, `d`, and `u`, and output arguments `d` and `f(u)`. This conversion is made possible here by the fact that neither of the first input arguments are used in the construction of the function value. Even though `f` is used repeatedly here, the sequence of calls to the `coco_add_func` constructor are distinguished by unique function identifiers and appropriate sets of numerical values for the corresponding elements of the vector of continuation variables. At this stage of construction, $\tilde{\Phi} : \mathbb{R}^{24} \to \mathbb{R}^8$, $\tilde{\Psi}$ is empty, and

$$\tilde{u}_0 = (p_0, x_0^{(1)}, x_0^{(2)}, p_0, x_0^{(2)}, x_0^{(3)}, p_0, x_0^{(3)}, x_0^{(4)}, p_0, x_0^{(4)}, x_0^{(1)}). \tag{8}$$

In a second stage of construction, we append three monitor functions that evaluate to the first three elements of u and associate the initially inactive continuation parameters labeled by `'a'`, `'b'`, and `'x1'`.

```
>> prob = coco_add_pars(prob, 'pars', ip, {'a' 'b'});
>> prob = coco_add_pars(prob, 'x1', ix(1), 'x1');
```

Consequently, $\tilde{\Psi}(\tilde{u}) = \tilde{u}_{\{1,2,3\}}$ and $\tilde{\mathbb{I}}_\mu = \{1, 2, 3\}$. We proceed to eliminate the duplication of problem parameters and consecutive orbit points in the vector of continuation variables using the following for-loops.

```
>> for i=2:period
     uidx = coco_get_func_data(prob, ['henon' num2str(i)], 'uidx');
     prob = coco_add_glue(prob, ['pglue' num2str(i)], ip, uidx(ip));
```

```
   end
>> for i=1:period
     uidx = coco_get_func_data(prob, ['henon' num2str(i)], 'uidx');
     inxt = mod(i, period)+1;
     uidxnxt = coco_get_func_data(prob, ['henon' num2str(inxt)], 'uidx');
     prob = coco_add_glue(prob, ['yglue' num2str(i)], uidx(iy), uidxnxt(ix));
   end
```

As this completes the problem construction, it follows that $\Phi : \mathbb{R}^{24} \to \mathbb{R}^{22}$ and $\Psi : \mathbb{R}^{24} \to \mathbb{R}^3$ are given by

$$\Phi : u \mapsto \begin{pmatrix} u_{\{5,6\}} - h(u_3, u_4, u_1, u_2) \\ u_{\{11,12\}} - h(u_9, u_{10}, u_7, u_8) \\ u_{\{17,18\}} - h(u_{15}, u_{16}, u_{13}, u_{14}) \\ u_{\{23,24\}} - h(u_{21}, u_{22}, u_{19}, u_{20}) \\ u_{\{7,8\}} - u_{\{1,2\}} \\ u_{\{13,14\}} - u_{\{1,2\}} \\ u_{\{19,20\}} - u_{\{1,2\}} \\ u_{\{9,10\}} - u_{\{5,6\}} \\ u_{\{15,16\}} - u_{\{11,12\}} \\ u_{\{21,22\}} - u_{\{17,18\}} \\ u_{\{3,4\}} - u_{\{23,24\}} \end{pmatrix} \tag{9}$$

and $\Psi : u \mapsto u_{\{1,2,3\}}$ with $\mathbb{I}_\mu = \{1, 2, 3\}$. Since the dimensional deficit equals -1, we must release two inactive continuation parameters in order to obtain a one-dimensional solution manifold. This is accomplished in the following call to the `coco` entry-point function, in which variations in `'a'` and `'x1'` are limited to the range $a \in [0.8, 1.2]$.

```
>> bd1 = coco(prob, 'run1', [], 1, {'a' 'x1'}, [0.8 1.2]);
```

The screen output shows the existence of a fold point along the solution branch at $a \approx 0.9125$ that approximately coincides with a branch point. We continue along a secondary branch through the branch point using the following sequence of commands.

```
>> lab   = coco_bd_labs(bd1, 'BP');
>> chart = coco_read_solution('run1', lab, 'chart');
>> cdata = coco_get_chart_data(chart, 'lsol');
>> prob  = coco_prob;
>> for i=1:period
     fid = ['henon' num2str(i)];
     [chart, uidx] = coco_read_solution(fid, 'run1', lab, 'chart', 'uidx');
     inxt = mod(i, period)+1;
     prob = coco_add_func(prob, fid, fcn(f), [], 'zero', ...
       'u0', chart.x, 't0', cdata.v(uidx));
   end
>> prob  = coco_add_pars(prob, 'pars', ip, {'a' 'b'});
>> prob  = coco_add_pars(prob, 'x1', ix(1), 'x1');
>> for i=2:period
     uidx = coco_get_func_data(prob, ['henon' num2str(i)], 'uidx');
     prob = coco_add_glue(prob, ['pglue' num2str(i)], ip, uidx(ip));
   end
>> for i=1:period
```

```
    uidx = coco_get_func_data(prob, ['henon' num2str(i)], 'uidx');
    inxt = mod(i, period)+1;
    uidxnxt = coco_get_func_data(prob, ['henon' num2str(inxt)], 'uidx');
    prob = coco_add_glue(prob, ['yglue' num2str(i)], uidx(iy), uidxnxt(ix));
  end
>> bd2 = coco(prob, 'run2', [], 1, {'a' 'x1'}, [0.8 1.2]);
```

Here, the content of the `cdata.v` vector is used to introduce a candidate continuation direction in the construction of the vector of continuation variables.

Exercises

1. The branch point along the family of period-4 orbits in the example is a period-doubling point along a family of period-2 orbits that coincides with the secondary family of period-4 orbits found in the second run. Construct a suitable continuation problem of the form (2) for continuation of period-2 orbits through this point and verify the claim.

2. Append a monitor function that evaluates to the eigenvalues of the Jacobian with respect to x of the composition $h \circ h$ and use this to characterize the stability of the period-2 orbits found in the previous exercise.

3. Use the approach of the previous exercise to locate a period-doubling point along the family of period-4 orbits in the example. Hint: use `coco_add_event` to detect an eigenvalue crossing -1.

4. Use the `'t0'` flag to branch switch at such a period-doubling point to a branch of period-8 solutions. Hint: use the eigenvector corresponding to the -1 eigenvalue.

---

4 Stationary points – **sphere_optim**

Consider the problem of finding stationary points of the function $u \mapsto u_1 + u_2 + u_3 + u_4$ on the unit 3-sphere in $\mathbb{R}^4$. To this end, consider the Lagrangian

$$L(u, \mu_{\text{sum}}, \mu_u, \lambda, \eta_{\text{sum}}, \eta_u) = \mu_{\text{sum}} + \lambda(\|u\|^2 - 1) + \eta_{\text{sum}} \left(\sum_{i=1}^{4} u_i - \mu_{\text{sum}} \right) + \eta_u^T \cdot (u - \mu_u) \quad (10)$$

in terms of the Lagrange multipliers λ, η_{sum}, and η_u. Necessary conditions for stationary points along the constraint manifold correspond to points $(u, \mu_{\text{sum}}, \mu_u, \lambda, \eta_{\text{sum}}, \eta_u)$ for which $\delta L = 0$ for any infinitesimal variations δu, $\delta\mu_{\text{sum}}$, $\delta\mu_u$, $\delta\lambda$, $\delta\eta_{\text{sum}}$, and $\delta\eta_u$. In this case, these conditions take the form

$$\|u\|^2 - 1 = 0,\ \sum_{i=1}^{4} u_i - \mu_{\text{sum}} = 0,\ u - \mu_u = 0,\ 2\lambda u + \eta_{\text{sum}}\mathbf{1} + \eta_u = 0, \quad (11)$$

$1 - \eta_{\text{sum}} = 0$, and $\eta_u = 0$. There are two distinct solutions to these conditions, namely the points $u = \mu_u = \pm\frac{1}{2}\mathbf{1}$, $\mu_{\text{sum}} = \pm 2$, $\lambda = \mp 1$, $\eta_{\text{sum}} = 1$, and $\eta_u = 0$.

Stationary points along the solution manifold may be located using a method of successive continuation[6] applied to the continuation problem obtained by combining (11) with $\eta_{\text{sum}} - \nu_{\text{sum}} = 0$ and $\eta_u - \nu_u = 0$ in terms of the continuation variables u, continuation multipliers $(\lambda, \eta_{\text{sum}}, \eta_u)$, and continuation parameters $(\mu_{\text{sum}}, \mu_u, \nu_{\text{sum}}, \nu_u)$. The dimensional deficit of this extended continuation problem equals 5. We get one-dimensional solution manifolds by designating four of the continuation parameters as inactive. Alternatively, if $\mathbb{I}_\mu = \{1, \ldots, 5\}$ and $\mathbb{I}_\nu = \{6, \ldots, 10\}$, then the dimensional deficit of the corresponding restricted continuation problem equals -5, and we get one-dimensional solution manifolds by designating six of the continuation parameters as active.

Suppose, for example, that μ_{sum}, $\mu_{u,\{1,4\}}$, ν_{sum}, and $\nu_{u,\{2,3\}}$ are active and $\mu_{u,\{2,3\}}$ and $\nu_{u,\{1,4\}}$ are inactive with $\rho^2 := 1 - \mu_{u,2}^2 - \mu_{u,3}^2 > 0$, $\rho > 0$, and $\nu_{u,1} = \nu_{u,4} = 0$. Solutions to the corresponding restricted continuation problem of the form

$$(u, \mu_{\text{sum}}, \mu_u, \lambda, \eta_{\text{sum}}, \eta_u, \nu_{\text{sum}}, \nu_u) = \left(\mu_u, \sum_{i=1}^{4} \mu_{u,i}, \mu_u, \lambda, \nu_{\text{sum}}, \nu_u, \nu_{\text{sum}}, \nu_u\right) \tag{12}$$

are located on three one-dimensional manifolds given by

$$\mu_{u,1} = \rho\cos\theta,\ \mu_{u,4} = \rho\sin\theta,\ \lambda = \nu_{\text{sum}} = \nu_{u,2} = \nu_{u,3} = 0 \tag{13}$$

and

$$\mu_{u,1} = \mu_{u,4} = \pm\frac{\rho}{\sqrt{2}},\ \lambda = \mp\frac{\nu_{\text{sum}}}{\sqrt{2}\rho}, \tag{14}$$

$$\nu_{u,2} = \nu_{\text{sum}}\left(\pm\frac{\sqrt{2}\mu_{u,2}}{\rho} - 1\right),\ \nu_{u,3} = \nu_{\text{sum}}\left(\pm\frac{\sqrt{2}\mu_{u,3}}{\rho} - 1\right), \tag{15}$$

parameterized by θ and ν_{sum}, respectively. The manifolds in (14) intersect the manifold in (13) at the points given by

$$\mu_{u,1} = \mu_{u,4} = \pm\frac{\rho}{\sqrt{2}},\ \lambda = \nu_{u,2} = \nu_{u,3} = 0, \tag{16}$$

corresponding to local extrema in the value of μ_{sum} along the first manifold.

Notably, there is a unique point on each of the latter manifolds at which $\eta_{\text{sum}} = 1$. If we consider the restricted continuation problem obtained with μ_{sum}, $\mu_{u,\{1,2,4\}}$, and $\nu_{u,\{2,3\}}$ active, and $\mu_{u,3}$, $\nu_{u,\{1,4\}}$, and ν_{sum} inactive with $\nu_{u,1} = \nu_{u,4} = 0$ and $\nu_{\text{sum}} = 1$, then solutions are located on the one-dimensional manifolds given by

$$\mu_{u,1} = \mu_{u,4} = \pm\frac{\rho}{\sqrt{2}},\ \mu_{u,2}^2 = 1 - \rho^2 - \mu_{u,3}^2,\ \lambda = \mp\frac{1}{\sqrt{2}\rho} \tag{17}$$

$$\nu_{u,2} = \pm\frac{\sqrt{2}\mu_{u,2}}{\rho} - 1,\ \nu_{u,3} = \pm\frac{\sqrt{2}\mu_{u,3}}{\rho} - 1, \tag{18}$$

[6]Kernévez, J. and Doedel, E. , "Optimization in bifurcation problems using a continuation method," in *Bifurcation: Analysis, Algorithms, Applications*, Springer, pp. 153–160, 1987.

parameterized by ρ.

There is a unique point along each of the tertiary manifolds in (17)-(18) at which $\eta_{u,2} = 0$, obtained with

$$\rho = \sqrt{\frac{2}{3}(1 - \mu_{u,3}^2)},\ \mu_{u,2} = \pm\sqrt{\frac{1}{3}(1 - \mu_{u,3}^2)}. \tag{19}$$

If we consider the restricted continuation problem obtained with μ_{sum}, μ_u, and $\nu_{u,3}$ active, and $\nu_{u,\{1,2,4\}}$, and ν_{sum} inactive with $\nu_{u,1} = \nu_{u,2} = \nu_{u,4} = 0$ and $\nu_{\text{sum}} = 1$, then solutions are located on the one-dimensional manifolds given by

$$\mu_{u,1} = \mu_{u,2} = \mu_{u,4} = \pm\frac{\rho}{\sqrt{2}},\ \mu_{u,3}^2 = 1 - \frac{3}{2}\rho^2,\ \lambda = \mp\frac{1}{\sqrt{2}\rho},\ \nu_{u,3} = \pm\frac{\sqrt{2}\mu_{u,3}}{\rho} - 1, \tag{20}$$

parameterized by ρ. Notably, the points along each of these manifolds with $\eta_{u,3} = 0$ coincide with the stationary points found previously

We proceed to implement in COCO a corresponding continuation problem of the form in (3) by making repeated use of the `coco_add_func` and `coco_add_adjt` constructors. We initialize the continuation problem structure and two useful cell arrays in the following commands.

```
>> prob = coco_prob;
>> fcn1 = { @sphere, @sphere_du, @sphere_dudu };
>> fcn2 = { @comb, @comb_du, @comb_dudu };
```

The function handles `@sphere`, `@sphere_du`, and so on point to the COCO compatible encodings shown below.

```
function [data, f] = sphere(prob, data, u)
f = u(1)^2 + u(2)^2 + u(3)^2 + u(4)^2 - 1;
end

function [data, J] = sphere_du(prob, data, u)
J = [2*u(1), 2*u(2), 2*u(3), 2*u(4)];
end

function [data, dJ] = sphere_dudu(prob, data, u)

dJ = zeros(1,4,4);
dJ(1,1,1) = 2;
dJ(1,2,2) = 2;
dJ(1,3,3) = 2;
dJ(1,4,4) = 2;

end

function [data, f] = comb(prob, data, u)
f = u(1)+u(2)+u(3)+u(4);
end

function [data, J] = comb_du(prob, data, u)
J = [1, 1, 1, 1];
end
```

```
function [data, dJ] = comb_dudu(prob, data, u)
dJ = zeros(1,4,4);
end
```

In the first stage of construction, we define a zero function and initialize part of the vector of continuation variables, as shown below.

```
>> prob1 = coco_add_func(prob, 'sphere', fcn1{:}, [], 'zero', ...
      'u0', [1 0 0 0]);
```

At this point, $\tilde{\Phi} : \mathbb{R}^4 \to \mathbb{R}$ is defined by $\tilde{\Phi} : \tilde{u} \mapsto \|\tilde{u}\|^2 - 1$, $\tilde{\Psi}$ is empty, and $\tilde{u}_0 = (1, 0, 0, 0)$. In the second stage of construction, the call

```
>> prob1 = coco_add_func(prob1, 'sum', fcn2{:}, [], 'inactive', ...
      'sum', 'uidx', 1:4);
```

results in no change to $\tilde{u}$ or $\tilde{\Phi}$, whereas now $\tilde{\Psi} : \mathbb{R}^4 \to \mathbb{R}$ is defined by $\tilde{\Psi}(\tilde{u}) = \tilde{u}_1 + \tilde{u}_2 + \tilde{u}_3 + \tilde{u}_4$. This function is associated with an initially inactive continuation parameter with string label `'sum'`.

In the third and fourth stages of construction, shown below, we use the `coco_add_pars` special-purpose wrapper to append four more monitor functions associated with two initially inactive and two initially active continuation parameters.

```
>> prob1 = coco_add_pars(prob1, 'pars1', [2 3], {'u2' 'u3'}, 'inactive');
>> prob1 = coco_add_pars(prob1, 'pars2', [1 4], {'u1' 'u4'}, 'active');
```

Each call passes the arguments to an encapsulated call to `coco_add_func` with function given by the identity map and its derivatives, and with `'uidx'` equal to `[2 3]` and `[1 4]`, respectively. As this concludes the construction of zero or monitor functions, we conclude that $\Phi : \mathbb{R}^4 \to \mathbb{R}$ and $\Psi : \mathbb{R}^4 \to \mathbb{R}^5$, where

$$\Phi : u \mapsto \|u\|^2 - 1, \; \Psi : u \mapsto \begin{pmatrix} u_1 + u_2 + u_3 + u_4 \\ u_2 \\ u_3 \\ u_1 \\ u_4 \end{pmatrix}, \tag{21}$$

$u_0 = (1, 0, 0, 0)$, and $\mathbb{I}_\mu = \{1, 2, 3\}$.

We proceed to append adjoint function objects associated with each of the calls to `coco_add_func`. Specifically, the call

```
>> prob1 = coco_add_adjt(prob1, 'sphere');
```

results in $\tilde{\Lambda}_\Phi : u \mapsto \begin{pmatrix} 2u_1 & 2u_2 & 2u_3 & 2u_4 \end{pmatrix}$ and $\tilde{\lambda}_{\Phi,0} = 0$. Similarly, the call

```
>> prob1 = coco_add_adjt(prob1, 'sum', 'd.sum', 'aidx', 1:4);
```

results in

$$\tilde{\Lambda}_\Psi : u \mapsto \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} \tag{22}$$

and $\tilde{\lambda}_{\Psi,0} = 0$. The corresponding element of ν is here associated with the string label `'d.sum'`. In each of the two following calls to `coco_add_adjt`, the elements of the identity matrix are distributed among the two new rows appended to Λ according to the column indices indicated by the flag `'aidx'`.

```
>> prob1 = coco_add_adjt(prob1, 'pars1', {'d.u2' 'd.u3'}, 'aidx', [2 3]);
>> prob1 = coco_add_adjt(prob1, 'pars2', {'d.u1' 'd.u4'}, 'aidx', [1 4]);
```

while the corresponding continuation multipliers are initialized to 0. Since this completes the construction of adjoint functions, it follows that

$$\Lambda_\Phi : u \mapsto \begin{pmatrix} 2u_1 & 2u_2 & 2u_3 & 2u_4 \end{pmatrix}, \Lambda_\Psi : u \mapsto \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{23}$$

$\lambda_{\Phi,0} = 0$, $\lambda_{\Psi,0} = 0$, and $\mathbb{I}_\nu = \{6, 7, 8, 9, 10\}$. The second of the two commands below is then equivalent to the computation of the manifold in (13) with $\rho = 1$.

```
>> cont_args = { 1, { 'sum' 'd.sum' 'd.u2' 'd.u3' }, [1 2] };
>> bd1 = coco(prob1, 'sphere1', [], cont_args{:});
```

Continuation detects a branch point coincident with a local extremum in the continuation parameter `'sum'` when this equals $\sqrt{2}$, i.e., for $\theta = \pi/4$ in the notation of (13). At the branch point, COCO stores a second unit vector that is perpendicular to the tangent vector to the solution manifold, such that the two vectors span a plane that also contains the tangent vector to the second branch through the branch point. We extract this second unit vector from the `'lsol'` field of the chart data array stored with the solution file, as shown here:

```
>> lab   = coco_bd_labs(bd1, 'BP');
>> chart = coco_read_solution('sphere1', lab, 'chart');
>> cdata = coco_get_chart_data(chart, 'lsol');
```

The vector we seek is in the `v` field of the `cdata` structure.

We proceed to reconstruct the continuation problem, using the branch point as the initial solution guess and appending the second unit vector as a suggested initial direction of continuation in order to continue along the second branch through the branch point. The following five commands reconstruct the zero and monitor functions:

```
>> [chart, uidx] = coco_read_solution('sphere', 'sphere1', lab, ...
     'chart', 'uidx');
>> prob2 = coco_add_func(prob, 'sphere', fcn1{:}, [], 'zero', ...
     'u0', chart.x,  't0', cdata.v(uidx));
>> prob2 = coco_add_func(prob2, 'sum', fcn2{:}, [], 'inactive', ...
     'sum', 'uidx', 1:4);
>> prob2 = coco_add_pars(prob2, 'pars1', [2 3], {'u2' 'u3'}, 'inactive');
>> prob2 = coco_add_pars(prob2, 'pars2', [1 4], {'u1' 'u4'}, 'active');
```

Here, the call to the `coco_read_solution` utility extracts the portion of the chart data array and the function dependency index set associated with the function identifier `'sphere'` from the branch point solution file. The `x` field of the `chart` variable reinitializes the continuation variables in the first stage of construction.

The reconstruction of the adjoint functions shown below relies on repeated calls to the `coco_read_adjoint` utility to extract portions of the chart data array and the row index sets associated with the different function identifiers from the branch point solution file.

```
>> [chart, lidx] = coco_read_adjoint('sphere', 'sphere1', lab, ...
     'chart', 'lidx');
>> prob2 = coco_add_adjt(prob2, 'sphere', 'l0', chart.x, ...
     'tl0', cdata.v(lidx));
>> [chart, lidx] = coco_read_adjoint('sum', 'sphere1', lab, ...
     'chart', 'lidx');
>> prob2 = coco_add_adjt(prob2, 'sum', 'd.sum', 'aidx', 1:4, ...
     'l0', chart.x, 'tl0', cdata.v(lidx));
>> [chart, lidx] = coco_read_adjoint('pars1', 'sphere1', lab, ...
     'chart', 'lidx');
>> prob2 = coco_add_adjt(prob2, 'pars1', {'d.u2' 'd.u3'}, 'aidx', [2 3], ...
     'l0', chart.x, 'tl0', cdata.v(lidx));
>> [chart, lidx] = coco_read_adjoint('pars2', 'sphere1', lab, ...
     'chart', 'lidx');
>> prob2 = coco_add_adjt(prob2, 'pars2', {'d.u1' 'd.u4'}, 'aidx', [1 4], ...
     'l0', chart.x, 'tl0', cdata.v(lidx));
```

The second of the two commands below is then equivalent to the computation along the manifold in (14)-(15) with $\rho = 1$, $\mu_{u,1} = \mu_{u,4} = 1/\sqrt{2}$, and $\mu_{u,2} = \mu_{u,3} = 0$.

```
>> cont_args = { 1,  { 'd.sum' 'sum' 'd.u2' 'd.u3' }, { [0 1], [-2 2] } };
>> bd2 = coco(prob2, 'sphere2', [], cont_args{:});
```

Continuation terminates once `'d.sum'` equals 1, at which point `'d.u2'` and `'d.u3'` both equal -1. The following sequence of commands reconstructs the continuation problem and initializes the continuation variables and continuation multipliers using information from this terminal solution point.

```
>> lab   = coco_bd_labs(bd2, 'EP');
>> chart = coco_read_solution('sphere', 'sphere2', lab(2), 'chart');
>> prob3 = coco_add_func(prob, 'sphere', fcn1{:}, [], 'zero', ...
     'u0', chart.x);
>> prob3 = coco_add_func(prob3, 'sum', fcn2{:}, [], 'inactive', ...
     'sum', 'uidx', 1:4);
>> prob3 = coco_add_pars(prob3, 'pars1', [2 3], {'u2' 'u3'}, 'inactive');
>> prob3 = coco_add_pars(prob3, 'pars2', [1 4], {'u1' 'u4'}, 'active');
>> chart = coco_read_adjoint('sphere', 'sphere2', lab(2), 'chart');
>> prob3 = coco_add_adjt(prob3, 'sphere', 'l0', chart.x);
>> chart = coco_read_adjoint('sum', 'sphere2', lab(2), 'chart');
>> prob3 = coco_add_adjt(prob3, 'sum', 'd.sum', 'aidx', 1:4, ...
     'l0', chart.x);
>> chart = coco_read_adjoint('pars1', 'sphere2', lab(2), 'chart');
>> prob3 = coco_add_adjt(prob3, 'pars1', {'d.u2' 'd.u3'}, ...
     'aidx', [2 3], 'l0', chart.x);
```

```
>> chart = coco_read_adjoint('pars2', 'sphere2', lab(2), 'chart');
>> prob3 = coco_add_adjt(prob3, 'pars2', {'d.u1' 'd.u4'}, ...
      'aidx', [1 4], 'l0', chart.x);
```

In this case, we allow for default initialization of the tangent vector, since there is a unique solution manifold of the restricted continuation problem through the initial point. In the next command, the `coco_add_event` utility is used to introduce a special point, designated by the label `'OPT'` whenever `'d.u2'` equals 0.

```
>> prob3 = coco_add_event(prob3, 'OPT', 'd.u2', 0);
```

The second of the two commands below is then equivalent to the computation along the manifold in (17)-(18) with $\mu_{u,1} = \mu_{u,4} = \rho/\sqrt{2}$ and $\mu_{u,3} = 0$.

```
>> cont_args = {1, {'d.u2' 'sum' 'u2' 'd.u3'}, {[] [-2 2]}};
>> bd3 = coco(prob3, 'sphere3', [], cont_args{:});
```

Continuation results in a unique point with `'d.u2'` equal to 0, at which `'sum'` equals $\sqrt{3}$, `'u2'` equals $1/\sqrt{3}$, and `'d.u3'` equals -1.

We conclude our analysis by again reconstructing the continuation problem, this time initializing the continuation variables and continuation multipliers using solution data from the `'OPT'` point in the previous run.

```
>> lab   = coco_bd_labs(bd3, 'OPT');
>> chart = coco_read_solution('sphere', 'sphere3', lab, 'chart');
>> prob4 = coco_add_func(prob, 'sphere', fcn1{:}, [], 'zero', ...
      'u0', chart.x);
>> prob4 = coco_add_func(prob4, 'sum', fcn2{:}, [], 'inactive', ...
      'sum', 'uidx', 1:4);
>> prob4 = coco_add_pars(prob4, 'pars1', [2 3], {'u2' 'u3'}, 'inactive');
>> prob4 = coco_add_pars(prob4, 'pars2', [1 4], {'u1' 'u4'}, 'active');
>> chart = coco_read_adjoint('sphere', 'sphere3', lab, 'chart');
>> prob4 = coco_add_adjt(prob4, 'sphere', 'l0', chart.x);
>> chart = coco_read_adjoint('sum', 'sphere3', lab, 'chart');
>> prob4 = coco_add_adjt(prob4, 'sum', 'd.sum', 'aidx', 1:4, ...
      'l0', chart.x);
>> chart = coco_read_adjoint('pars1', 'sphere3', lab, 'chart');
>> prob4 = coco_add_adjt(prob4, 'pars1', {'d.u2' 'd.u3'}, ...
      'aidx', [2 3], 'l0', chart.x);
>> chart = coco_read_adjoint('pars2', 'sphere3', lab, 'chart');
>> prob4 = coco_add_adjt(prob4, 'pars2', {'d.u1' 'd.u4'}, ...
      'aidx', [1 4], 'l0', chart.x);
```

This encoding again allow for default initialization of the tangent vector, since there is a unique solution manifold of the restricted continuation problem through the initial point. In the next command, the `coco_add_event` utility is used to introduce a special point, designated by the label `'OPT'` whenever `'d.u3'` equals 0.

```
>> prob4 = coco_add_event(prob4, 'OPT', 'd.u3', 0);
```

The second of the two commands below is then equivalent to the computation along the manifold in (20) with $\mu_{u,1} = \mu_{u,2} = \mu_{u,4} = \rho/\sqrt{2}$.

```
>> cont_args = {1, {'d.u3' 'sum' 'u2' 'u3'}, {[] [-2 2]}};
>> coco(prob4, 'sphere4', [], cont_args{:});
```

Continuation results in a unique point with `'d.u3'` equal to 0, at which `'sum'` equals 2, and `'u2'` and `'u3'` both equal 1/2.

Exercises

1. Repeat the analysis in this section for a different initial designation of continuation parameters to $\mathbb{I}_\mu$ and a different permutation of the order in which the corresponding complementary continuation parameters are driven to 0.

2. Consider the problem of finding stationary points in $\mathbb{R}^3$ of the function $u \mapsto u_2$ on the manifold defined by $u_2 - u_1(u_3 - u_1^2) = 0$. Repeat the analysis in this section and verify your theoretical predictions using COCO.

3. Visualize different projections of the solution manifolds considered in the search for stationary points on the sphere and in the previous exercise.

4. Use the methodology described here and in the `'coll'` and `'po'` tutorials to implement the examples in Li, M. & Dankowicz, H., "Staged Construction of Adjoints for Constrained Optimization of Integro-Differential Boundary-Value Problems," *SIAM J. Applied Dynamical Systems* **17(2)**, pp. 1117–1151, 2018.

---

5 Karush-Kuhn-Tucker conditions – `linode_optim`

Consider the problem of finding local minima of the function $(x(t), k, f, \theta) \mapsto x_2(0)$ along a manifold of periodic solutions of the dynamical system

$$\dot{x}_1 = x_2,\ \dot{x}_2 = -x_2 - kx_1 + f\cos(t+\theta) \tag{24}$$

with period 2π and restricted to the feasible region $f \leq f_0$. Since such periodic solutions are given by

$$x_1(t) = f\frac{(k-1)\cos(t+\theta) + \sin(t+\theta)}{(k-1)^2+1},\ x_2(t) = f\frac{\cos(t+\theta) + (1-k)\sin(t+\theta)}{(k-1)^2+1}, \tag{25}$$

local minima are obtained by analysis of the Lagrangian

$$\begin{aligned} L\left(k, f, \theta, \mu_v, \mu_k, \mu_f, \mu_\theta, \sigma_f, \eta_v, \eta_k, \eta_f, \eta_\theta\right) &= \mu_v + \eta_v\left(f\frac{\cos\theta + (1-k)\sin\theta}{(k-1)^2+1} - \mu_v\right) \\ &+ \eta_k(k-\mu_k) + \eta_f(f-\mu_f) + \eta_\theta(\theta-\mu_\theta) + \sigma_f(f-f_0), \end{aligned} \tag{26}$$

from which we obtain the Karush-Kuhn-Tucker conditions

$$f\frac{\cos\theta+(1-k)\sin\theta}{(k-1)^2+1}-\mu_v=0, \tag{27}$$

$$k-\mu_k=f-\mu_f=\theta-\mu_\theta=0, \tag{28}$$

$$f\eta_v\frac{k(k-2)\sin\theta+2(1-k)\cos\theta}{\left((k-1)^2+1\right)^2}+\eta_k=0, \tag{29}$$

$$\eta_v\frac{\cos\theta+(1-k)\sin\theta}{(k-1)^2+1}+\eta_f+\sigma_f=0, \tag{30}$$

$$f\eta_v\frac{(1-k)\cos\theta-\sin\theta}{(k-1)^2+1}+\eta_\theta=0, \tag{31}$$

with $\eta_v=1$, $\eta_k=\eta_f=\eta_\theta=0$, and the linear complementarity condition $\sigma_f(f-f_0)=0$ on $\{\sigma_f\geq 0,\ f\leq f_0\}$. These imply that the local minimum is located at

$$\mu_k=k=1,\ \mu_f=f=f_0,\ \mu_\theta=\theta=\pi,\ \mu_v=-f_0,\ \sigma_f=1,\ \eta_k=\eta_f=\eta_\theta=0,\eta_v=1. \tag{32}$$

Since the value of $x_2(0)$ is linear in f, it follows that this is, in fact, a global minimum in the feasible region $f\leq f_0$.

We may arrive at this minimum using a method of successive continuation applied to the continuation problem obtained by combining (27)-(31) with $\eta_k-\nu_k=0$, $\eta_f-\nu_f=0$, $\eta_\theta-\nu_\theta=0$, $\eta_v-\nu_v=0$ and

$$\sqrt{\sigma_f^2+(f-f_0)^2}-\sigma_f+f-f_0-\kappa_f=0 \tag{33}$$

in terms of the original continuation variables (k,f,θ), original continuation parameters (μ_k,μ_f,μ_θ), continuation multipliers $(\eta_k,\eta_f,\eta_\theta,\eta_v,\sigma_f)$, and complementary continuation parameters $(\nu_k,\nu_f,\nu_\theta,\nu_v,\kappa_f)$. With the substitution of (33) for the complementarity condition, the local minimum is obtained only when $\kappa_f=0$.

The successive continuation technique assumes that an initial solution to the continuation problem may be obtained with all zero continuation multipliers. Here, this can be accomplished with f both inside and outside the feasible region.

For example, with $f<f_0$ and $\sigma_f=0$ initially, it follows that $\kappa_f=0$ initially, and that $\kappa_f=\sigma_f=0$ as long as f remains smaller than f_0. Assuming μ_v, μ_k, ν_v, ν_f, and ν_θ active and μ_f, μ_θ, and ν_k inactive with $\mu_\theta\neq 0,\pi$ and $\nu_k=0$, solutions to the corresponding restricted continuation problem are obtained on one of the three one-dimensional manifolds

$$\mu_v=\mu_f\frac{\cos\mu_\theta+(1-\mu_k)\sin\mu_\theta}{(\mu_k-1)^2+1},\ \nu_v=\nu_f=\nu_\theta=0, \tag{34}$$

$$\mu_v=\mu_f\cos^2\frac{\mu_\theta}{2},\ \mu_k=1-\tan\frac{\mu_\theta}{2},\ \nu_\theta=\frac{\nu_v\mu_f}{2}\sin\mu_\theta,\ \nu_f=-\nu_v\cos^2\frac{\mu_\theta}{2}, \tag{35}$$

or

$$\mu_v=-\mu_f\sin^2\frac{\mu_\theta}{2},\ \mu_k=1+\cot\frac{\mu_\theta}{2},\ \nu_\theta=\frac{\nu_v\mu_f}{2}\sin\mu_\theta,\ \nu_f=\nu_v\sin^2\frac{\mu_\theta}{2}. \tag{36}$$

The manifold in (34) intersects the manifolds in (35) and (36) at the points

$$\mu_v = \mu_f \cos^2 \frac{\mu_\theta}{2},\ \mu_k = 1 - \tan \frac{\mu_\theta}{2},\ \nu_v = \nu_f = \nu_\theta = 0 \tag{37}$$

and

$$\mu_v = -\mu_f \sin^2 \frac{\mu_\theta}{2},\ \mu_k = 1 + \cot \frac{\mu_\theta}{2},\ \nu_v = \nu_f = \nu_\theta = 0, \tag{38}$$

respectively, corresponding to local extrema in the value of μ_v along the first manifold.

Notably, there is a unique point on each of the latter manifolds where $\eta_v = 1$. If we consider the restricted continuation problem obtained with μ_v, μ_k, μ_θ, ν_f and ν_θ active and μ_f, ν_k, and ν_v inactive, with $\nu_k = 0$ and $\nu_v = 1$, then solutions are located on the one-dimensional manifolds

$$\mu_v = \mu_f \cos^2 \frac{\mu_\theta}{2},\ \mu_k = 1 - \tan \frac{\mu_\theta}{2},\ \nu_f = -\cos^2 \frac{\mu_\theta}{2},\ \nu_\theta = \frac{\mu_f}{2} \sin \mu_\theta \tag{39}$$

and

$$\mu_v = -\mu_f \sin^2 \frac{\mu_\theta}{2},\ \mu_k = 1 + \cot \frac{\mu_\theta}{2},\ \nu_f = \sin^2 \frac{\mu_\theta}{2},\ \nu_\theta = \frac{\mu_f}{2} \sin \mu_\theta. \tag{40}$$

We reach a point with $\nu_\theta = 0$ along the first manifold when $\mu_\theta = 2n\pi$ for some integer n. Similarly, we reach a point with $\nu_\theta = 0$ along the second manifold when $\mu_\theta = (2n+1)\pi$ for some integer n. Indeed, if we consider the restricted continuation problem obtained with μ_v, μ_k, μ_f, μ_θ, and ν_f active and ν_k, ν_θ, and ν_v inactive and equal to 0, 0 and 1, respectively, then solutions are located on the one-dimensional manifolds

$$\mu_v = \mu_f,\ \mu_k = 1,\ \mu_\theta = 2n\pi,\ \nu_f = -1 \tag{41}$$

and

$$\mu_v = -\mu_f,\ \mu_k = 1,\ \mu_\theta = (2n+1)\pi,\ \nu_f = 1, \tag{42}$$

which intersect the manifolds in (39) and (40), respectively. Notably, ν_f is constant along each manifold and $\nu_f = 0$ cannot be reached under variations in μ_f. In fact, as $\mu_f \to f_0$, the solution limits on the singularity at $\sigma_f = f - f_0 = 0$ of the left-hand side of (33).

Alternatively, suppose that initially $f > f_0$ and $\sigma_f = 0$, from which it follows that κ_f must be initially positive. Now suppose that μ_v, μ_k, μ_f, ν_v, and ν_θ are active and μ_θ, ν_k, ν_f, and κ_f are inactive with $\mu_\theta \neq 0, \pi$ and $\nu_k = \nu_f = 0$. This time, solutions to the corresponding restricted continuation problem are located on one of the three one-dimensional manifolds

$$\mu_v = \mu_f \frac{\cos \mu_\theta + (1 - \mu_k) \sin \mu_\theta}{(\mu_k - 1)^2 + 1},\ \mu_f = f_0 + \frac{\kappa_f}{2},\ \nu_v = \nu_\theta = \sigma_f = 0 \tag{43}$$

$$\mu_v = \mu_f \cos^2 \frac{\mu_\theta}{2},\ \mu_k = 1 - \tan \frac{\mu_\theta}{2},\ \nu_\theta = \frac{\nu_v \mu_f}{2} \sin \mu_\theta,\ \sigma_f = -\nu_v \cos^2 \frac{\mu_\theta}{2}, \tag{44}$$

or

$$\mu_v = -\mu_f \sin^2 \frac{\mu_\theta}{2},\ \mu_k = 1 + \cot \frac{\mu_\theta}{2},\ \nu_\theta = \frac{\nu_v \mu_f}{2} \sin \mu_\theta,\ \sigma_f = \nu_v \sin^2 \frac{\mu_\theta}{2}, \tag{45}$$

where

$$\kappa_f = \sqrt{\sigma_f^2 + (\mu_f - f_0)^2} - \sigma_f + \mu_f - f_0 \tag{46}$$

along each of the latter manifolds.

The manifold in (43) intersects the manifolds in (44) and (45) at the points

$$\mu_v = \mu_f \cos^2 \frac{\mu_\theta}{2},\ \mu_k = 1 - \tan \frac{\mu_\theta}{2},\ \mu_f = f_0 + \frac{\kappa_f}{2},\ \nu_v = \nu_\theta = \sigma_f = 0 \tag{47}$$

and

$$\mu_v = -\mu_f \sin^2 \frac{\mu_\theta}{2},\ \mu_k = 1 + \cot \frac{\mu_\theta}{2},\ \mu_f = f_0 + \frac{\kappa_f}{2}, \nu_v = \nu_\theta = \sigma_f = 0, \tag{48}$$

respectively, corresponding to local extrema in the value of μ_v along the first manifold. Notably, there is a unique point on each of the latter manifolds where $\eta_v = 1$. If we consider the restricted continuation problem obtained with μ_v, μ_k, μ_f, μ_θ and ν_θ active, κ_f inactive, and ν_k, ν_f and ν_v inactive and equal to 0, 0, and 1, respectively, then solutions are located on the one-dimensional manifolds

$$\mu_v = \mu_f \cos^2 \frac{\mu_\theta}{2},\ \mu_k = 1 - \tan \frac{\mu_\theta}{2},\ \nu_\theta = \frac{\mu_f}{2} \sin \mu_\theta,\ \sigma_f = -\cos^2 \frac{\mu_\theta}{2} \tag{49}$$

and

$$\mu_v = -\mu_f \sin^2 \frac{\mu_\theta}{2},\ \mu_k = 1 + \cot \frac{\mu_\theta}{2},\ \nu_\theta = \frac{\mu_f}{2} \sin \mu_\theta,\ \sigma_f = \sin^2 \frac{\mu_\theta}{2}, \tag{50}$$

with κ_f given in (46). We reach a point with $\nu_\theta = 0$ along the first manifold when $\mu_\theta = 2n\pi$ for some integer n. Similarly, we reach a point with $\nu_\theta = 0$ along the second manifold when $\mu_\theta = (2n+1)\pi$ for some integer n. Indeed, if we consider the restricted continuation problem obtained with μ_v, μ_k, μ_θ, μ_f and κ_f active and ν_k, ν_f, ν_θ and ν_v inactive and equal to 0, 0, 0, and 1, respectively, then solutions are located on the one-dimensional manifolds

$$\mu_v = \mu_f,\ \mu_k = 1,\ \mu_\theta = 2n\pi,\ \sigma_f = -1 \tag{51}$$

and

$$\mu_v = -\mu_f,\ \mu_k = 1,\ \mu_\theta = (2n+1)\pi,\ \sigma_f = 1, \tag{52}$$

with κ_f again given in (46). We reach a point with $\kappa_f = 0$ along the second manifold when $\mu_f = f_0$, consistent with the earlier analysis.

We proceed to implement the extended continuation problem in COCO in the case that $f_0 = 1$. In this example, we make use of a set of automatic differentiation routines included with the SYMCOCO package[7]. Specifically, by first adding the `coco/contributed/symcoco` directory to the MATLAB path and then executing the commands

```
>> u = sym('u',[3,1]);
>> fu = u(2)*(cos(u(3))+(1-u(1))*sin(u(3)))/((u(1)-1)^2+1);
>> sco_sym2funcs(fu, {u}, {'u'}, 'filename', 'sym_linode');
```

we generate a SYMCOCO-compatible encoding of the function

$$u \mapsto f(u) = u_2 \frac{\cos u_3 + (1 - u_1) \sin u_3}{(u_1 - 1)^2 + 1}. \tag{53}$$

In terms of the variable

[7]The SYMCOCO package is included in the March, 2020, COCO release, courtesy of Jan Sieber.

```
>> Fu = sco_gen(@sym_linode);
```

function handles to the functions $u \mapsto f(u)$, $u \mapsto Df(u)$, and $D^2 f(u)$ are returned by the commands `Fu('')`, `Fu('u')`, and `Fu({'u','u'})`, respectively. We again call upon the anonymous function

```
>> fcn = @(f) @(p,d,u) deal(d, f(u));
```

to convert a function of u to a COCO-compatible format, as shown in the command below.

```
>> funcs = { fcn(Fu('')), fcn(Fu('u')), fcn(Fu({'u','u'})) };
```

The successive stages of continuation considered above may be represented in terms of three different problem constructions, namely, i) construction from an initial solution guess with vanishing continuation multipliers, ii) construction from a branch point with the intent of switching to a secondary branch of solutions to the same restricted continuation problem, and iii) construction from a previously-found solution along an embedded submanifold defined by different sets $\mathbb{I}_\mu$ and $\mathbb{I}_\nu$. We collect repeated calls to `coco_add_func` and `coco_add_adjt` in composite constructors to avoid obscuring the essence of the methodology. Specifically, we envision a calling sequence of the form

```
>> prob  = coco_prob;
>> prob  = coco_set(prob, 'cont', 'PtMX', [0 50], 'NPR', inf);
>> probb = isol2prob(prob, data, [1; 2; 4]);
>> coco(probb, 'run1', [], 1, {'v' 'k' 'f' 'd.v' 'd.theta'});
>> bd1   = coco_bd_read('run1');
>> BPlab = coco_bd_labs(bd1, 'BP');
>> probb = BP2prob(prob, data, 'run1', BPlab(1));
>> coco(probb, 'run2', [], 1, {'d.v' 'k' 'f' 'v' 'd.theta'}, [0 1]);
>> probb = sol2prob(prob, data, 'run2', 2);
>> coco(probb, 'run3', [], 1, {'d.theta' 'k' 'f' 'theta' 'v'}, [-1 0]);
>> probb = sol2prob(prob, data, 'run3', 4);
>> coco(probb, 'run4', [], 1, {'ncp.f' 'k' 'f' 'theta' 'v'}, [0 2]);
```

where, initially, $(k, f, \theta) = (1, 2, 4)$, i.e., outside of the feasible region.

We prepare useful data for the three distinct constructors and store it in the variable `data`.

```
>> bound = { fcn(@(u) u-1), fcn(@(u) 1), fcn(@(u) 0) };
>> data  = struct('funcs', {funcs}, 'bound', {bound});
```

The `isol2prob`, `BP2prob`, and `sol2prob` constructors shown below then rely on the input arguments to create appropriately initialized instances of the extended continuation problem.

```
function prob = isol2prob(prob, data, u0)

prob = coco_add_func(prob, 'vel', data.funcs{:}, [], 'inactive', 'v', ...
  'u0', u0);
prob = coco_add_pars(prob, 'pars', 1:3, {'k' 'f' 'theta'});
prob = coco_add_func(prob, 'bound', data.bound{:}, [], ...
  'inequality', 'bd', 'uidx', 2);
prob = coco_add_adjt(prob, 'vel', 'd.v');
```

```
prob = coco_add_adjt(prob, 'pars', {'d.k' 'd.f' 'd.theta'}, 'aidx', 1:3);
prob = coco_add_adjt(prob, 'bound', 'ncp.f', 'aidx', 2);

end


function prob = BP2prob(prob, data, run, lab)

chart = coco_read_solution(run, lab, 'chart');
cdata = coco_get_chart_data(chart, 'lsol');

[chart, uidx] = coco_read_solution('vel', run, lab, 'chart', 'uidx');
prob = coco_add_func(prob, 'vel', data.funcs{:}, [], 'inactive', 'v', ...
  'u0', chart.x, 't0', cdata.v(uidx));
prob = coco_add_pars(prob, 'pars', 1:3, {'k' 'f' 'theta'});
prob = coco_add_func(prob, 'bound', data.bound{:}, [], ...
  'inequality', 'bd', 'uidx', 2);
[chart, lidx] = coco_read_adjoint('vel', run, lab, 'chart', 'lidx');
prob = coco_add_adjt(prob, 'vel', 'd.v', 'l0', chart.x, ...
  'tl0', cdata.v(lidx));
[chart, lidx] = coco_read_adjoint('pars', run, lab, 'chart', 'lidx');
prob  = coco_add_adjt(prob, 'pars', {'d.k' 'd.f' 'd.theta'}, ...
  'aidx', 1:3, 'l0', chart.x, 'tl0', cdata.v(lidx));
[chart, lidx] = coco_read_adjoint('bound', run, lab, 'chart', 'lidx');
prob  = coco_add_adjt(prob, 'bound', 'ncp.f', 'aidx', 2, 'l0', chart.x, ...
  'tl0', cdata.v(lidx));

end


function prob = sol2prob(prob, data, run, lab)

chart = coco_read_solution('vel', run, lab, 'chart');
prob  = coco_add_func(prob, 'vel', data.funcs{:}, [], 'inactive', 'v', ...
  'u0', chart.x);
prob  = coco_add_pars(prob, 'pars', 1:3, {'k' 'f' 'theta'});
prob  = coco_add_func(prob, 'bound', data.bound{:}, [], ...
  'inequality', 'bd', 'uidx', 2);
chart = coco_read_adjoint('vel', run, lab, 'chart');
prob  = coco_add_adjt(prob, 'vel', 'd.v', 'l0', chart.x);
chart = coco_read_adjoint('pars', run, lab, 'chart');
prob  = coco_add_adjt(prob, 'pars', {'d.k' 'd.f' 'd.theta'}, ...
  'aidx', 1:3, 'l0', chart.x);
chart = coco_read_adjoint('bound', run, lab, 'chart');
prob  = coco_add_adjt(prob, 'bound', 'ncp.f', 'aidx', 2, 'l0', chart.x);

end
```

Exercises

1. Use the code included here to verify the theoretical predictions for an initial point inside or outside the feasible region. Explore different choices of initially inactive continuation parameters in the first stage of continuation, e.g., μ_k or μ_f rather than μ_θ. In the first

case, you may drive ν_k instead of ν_θ to zero in the third stage of continuation.

2. Consider the case of an initial solution guess in the feasible region. In the first stage, let ν_v, μ_v, ν_k, ν_f, and ν_θ be active and μ_k, μ_f, μ_θ, and κ_f be inactive. Show that the solution manifold includes a point where $\nu_v = 1$. From this point, use your COCO implementation to try to drive ν_k, ν_θ, and ν_f to zero successively.

3. Consider the case of an initial solution guess in the infeasible region. In the first stage, let ν_v, μ_v, ν_k, μ_f, and ν_θ be active and ν_k, μ_f, μ_θ, and κ_f be inactive. Show that the solution manifold includes a point where $\nu_v = 1$. From this point, use your COCO implementation to try to drive ν_k, ν_θ, and κ_f to zero successively.

4. Modify the code for the `linode_optim` example in `coco/coll/examples` to also account for the inequality constraint and repeat the analysis in this section.

---

6 Staged construction

In the COCO paradigm of staged construction, a general continuation problem is represented in terms of three Boolean matrices associated, respectively, with calls to `coco_add_func` to construct elements of Φ and Ψ, calls to `coco_add_adjt` to construct elements of Λ, and calls to `coco_add_comp` to construct elements of Ξ and Θ. These matrices satisfy the following two properties: i) no column consists entirely of zeroes and ii) if $i(j)$ denotes the row index of the first nonzero entry in the j-th column, then $i(1) = 1$ and the sequence $\{i(1), \ldots\}$ is nondecreasing. There is a one-to-one relationship between the rows of the second matrix and a subset of the rows of the first matrix.

In general, the first of the three matrices has n_u columns representing, in order, the elements of the vector of continuation variables u. Each call to `coco_add_func` used to construct elements of Φ or Ψ appends a row to this matrix, and associates this row with a COCO-compatible function encoding. Nonzero entries in this row indicate dependence of this function on a subset of already initialized elements of u, as well as on elements of u that are initialized in this call. In the notation of the previous paragraph, the j-th element of u is initialized in the $i(j)$-th such call to `coco_add_func`.

The n_Λ columns of the second of the three matrices represent the columns of $\Lambda(u)$. Each call to the `coco_add_adjt` constructor appends a row to this matrix, and associates this row with a COCO-compatible function encoding. Nonzero entries in this row indicate columns whose content is partially assigned from the output of this function. The dependence of this function on a subset of the elements of u is identical to that indicated by the uniquely associated row of the first matrix.

The one-to-one association between rows of the second matrix and a subset of rows of the first matrix allows for a default behavior of `coco_add_adjt`, in which construction relies on information provided to COCO by the associated call to `coco_add_func`. Specifically,

provided that the associated call to `coco_add_func` includes a function handle to an explicit encoding of the Jacobian of the zero or monitor function, then omission of a function handle in the call to `coco_add_adjt` implies that this explicit Jacobian should be used to compute the corresponding elements of $\Lambda(u)$.

Finally, the third of the three matrices has n_v columns representing, in order, the elements of the vector of complementary continuation variables v. Each call to `coco_add_comp` used to construct elements of Ξ or Θ appends a row to this matrix, and associates this row with a COCO-compatible function encoding. Nonzero entries in this row indicate dependence of this function on a subset of already initialized elements of v, as well as on elements of v that are initialized in this call. In the notation of the first paragraph, the j-th element of v is initialized in the $i(j)$-th such call to `coco_add_comp`.

7 Constructor syntax

7.1 Zero and monitor functions - `coco_add_func`

We construct a function object to represent elements of Φ or Ψ and append this to a partially implemented continuation problem structure `prob` by adhering to the appropriate argument syntax for the `coco_add_func` constructor:

```
>> prob = coco_add_func(prob, fid, varargin);
```

where

```
varargin = ( @f, [ @df, [ @ddf, ] ] | @fdf [ @ddf, ] ) data, type_spec, opts
```

Here, the *function identifier* `fid` denotes a string variable that is uniquely identified with this call to `coco_add_func` and that can be used to reference the function object that is instantiated in this call, e.g., in subsequent calls to `coco_add_adjt`.

The argument `@f` denotes a function handle to a COCO-compatible encoding of a *realization* $f : \mathbb{R}^p \to \mathbb{R}^q$, `@df` denotes a function handle to a COCO-compatible encoding of the function $Df : \mathbb{R}^p \to \mathbb{R}^{q\times p}$ whose component functions are first partial derivatives of f with respect to its arguments, and `@ddf` denotes a function handle to a COCO-compatible encoding of the function $D^2f : \mathbb{R}^p \to \mathbb{R}^{q\times p\times p}$ whose component functions are second partial derivatives of f with respect to its arguments. The integer p is less than or equal to the number of continuation variables introduced in this and previous stages of construction. For a zero function, the integer $n_\Phi - q$ is greater than or equal to number of components of Φ introduced in previous stages of construction. For a monitor function, $n_\Psi - q$ is greater than or equal to the number of components of Ψ introduced in previous stages of construction. In lieu of separate encodings for f and Df, the notation `@fdf` denotes a function handle to a COCO-compatible encoding $\{f, Df\} : \mathbb{R}^p \to \{\mathbb{R}^q, \mathbb{R}^{q\times p}\}$.

The *function data structure* `data` contains a structure array with function-specific content that can be accessed and modified by the encodings of f, Df and D^2f. If this variable is an instance of the `coco_func_data` class, then its content may be accessed and modified by

any other function to which the variable is sent. A write-protected copy of the function data structure associated with the function identifier `fid` is returned by the call

```
>> coco_get_func_data(prob, fid, 'data')
```

Such a write-protected copy should not be passed as the function data structure argument in another call to `coco_add_func`. Typically, the function data structure contains information that can be precomputed and reused in multiple calls to the encodings of f and its derivatives, for example during the application of a sequence of Newton iterations. Changes to `data` between continuation steps are commonly associated with adaptive changes to the discretization of an infinite-dimensional problem or updates to parameterizations that depend on previous points on the solution manifold.

The argument `type_spec` is the single string `'zero'` in the case that f represents a realization of elements of Φ. In the case that f represents a realization of elements of Ψ, `type_spec` is the string

- `'active'` followed by a cell array of q string labels assigned to the corresponding embedded continuation parameters, which are designated as initially active;
- `'inactive'` followed by a cell array of q string labels assigned to the corresponding embedded continuation parameters, which are designated as initially inactive;
- `'internal'` followed by a cell array of q string labels assigned to the corresponding embedded continuation parameters, which are designated as initially active;
- `'inequality'` followed by a cell array of q string labels assigned to the corresponding embedded continuation parameters, which are designated as initially active;
- `'regular'` followed by a cell array of q string labels assigned to the corresponding non-embedded continuation parameters; or
- `'singular'` followed by a cell array of q string labels assigned to the corresponding non-embedded continuation parameters.

Initially inactive continuation parameters may be activated by an exchange with an active (complementary) continuation parameter using the `coco_xchg_pars` utility, or by explicitly releasing them in the call to the `coco` entry-point function. Initially active continuation parameters may be deactivated by an exchange with an inactive (complementary) continuation parameter using `coco_xchg_pars` or, in the case of parameters labeled as `'internal'`, by an automatic exchange with overspecified inactive (complementary) continuation parameters in the call to the `coco` entry-point function.

Monitor functions associated with embedded continuation parameters must be continuously differentiable. In contrast, non-embedded continuation parameters are associated with monitor functions that may not be differentiable, although they need to be continuous functions along the solution manifold if used in event detection. While active embedded continuation parameters are treated as unknowns and solved for together with the continuation

variables, non-embedded continuation parameters are assigned values by evaluating the corresponding monitor function after the continuation variables have been found. Non-embedded continuation parameters allow for detection of regular special points (`'regular'`) with non-singular problem Jacobian or approximate detection of singular special points (`'singular'`) with singular problem Jacobian.

The argument `opts` is a placeholder for an arbitrary sequence of additional arguments that modify the construction of the function object. For example, in the call

```
>> prob = coco_add_func(prob, ..., 'f+df');
```

the flag `'f+df'` indicates that the first function handle in `varargin` is of the form `@fdf`. In the call

```
>> prob = coco_add_func(prob, ..., 'fdim', 3);
```

the flag `'fdim'` indicates that the output dimension q equals 3, thereby eliminating the need to determine q by evaluation of the function f during construction.

For elements of Φ or Ψ, the input argument to the corresponding function f is populated at run-time with a subset of p elements of u, indexed by a function-specific, ordered, dependency-index set $\mathbb{K}$. For example, if 12 continuation variables have been introduced in previous stages of construction, then the call

```
>> prob = coco_add_func(prob, ..., 'uidx', [2 4:10], 'u0', [0.3 2.5]);
```

results in the assignments $\mathbb{K} = \{2, 4, 5, 6, 7, 8, 9, 10, 13, 14\}$ and $u_{0,\{13,14\}} = (0.3, 2.5)$. When the `'u0'` flag is present, an optional additional inclusion of the flag `'t0'` as in this call

```
>> prob = coco_add_func(prob, ..., 'u0', [0.3 2.5], 't0', [1.4 3.9]);
```

results in the assignment $t_{0,\{13,14\}} = (1.4, 3.9)$ of components of a vector used in the construction of the initial direction of continuation. When the `'t0'` flag is not present, these components default to 0.

For a representation of elements of Φ or Ψ, a copy of the function dependency index set $\mathbb{K}$ may be obtained with the call

```
>> coco_get_func_data(prob, fid, 'uidx');
```

In a typical application, the call

```
>> [data, uidx] = coco_get_func_data(prob, fid, 'data', 'uidx');
```

may be followed by a construction of the form

```
>> prob = coco_add_func(prob, ..., 'uidx', uidx(data.x_idx));
```

where `uidx(data.x_idx)` evaluates to a subset of the function dependency index set of the function with function identifier `fid`, indexed by the `x_idx` field of the function data structure of this function. This type of formulation uses the relative indexing of `data.x_idx` to accommodate any dependence on preceding stages of construction, without necessitating ex-

plicit reference to the detailed implementation of each such stage. The function dependency index set associated with the function identifier `fid` may also be extracted from data stored to disk during continuation using the `coco_read_solution` utility according to the syntax

```
>> uidx = coco_read_solution(fid, run, lab, 'uidx');
```

where `run` is a string that denotes the run identifier and `lab` is an integer that identifies the solution label.

The call

```
>> uidx = coco_get_func_data(prob, 'efunc', 'uidx');
```

may be used to obtain the union of all function dependency index sets for functions constructed using `coco_add_func` or related constructors (including the special-purpose wrappers `coco_add_pars`, `coco_add_glue`, and `coco_add_functionals`).

7.2 Adjoint functions - `coco_add_adjt`

Each call to `coco_add_func` used to construct elements of Φ or Ψ may be uniquely associated to a subsequent call to the `coco_add_adjt` constructor according to the syntax:

```
>> prob = coco_add_adjt(prob, fid, varargin);
```

where

```
varargin = [ ( @g, | @gdg, ) [ @dg, ] data, ] [ par_names, ['active']] opts
```

and the function identifier `fid` is identical to the function identifier used in the preceding call to `coco_add_func`.

Here, the argument `@g` denotes a function handle to a COCO-compatible encoding of a realization $g : \mathbb{R}^p \to \mathbb{R}^{q_1 \times q_2}$, while `@dg` denotes a function handle to a COCO-compatible encoding of the function $Dg : \mathbb{R}^p \to \mathbb{R}^{q_1 \times q_2 \times p}$ whose component functions are first partial derivatives of g with respect to its arguments. In lieu of separate encodings for g and Dg, the notation `@gdg` denotes a function handle to a COCO-compatible encoding $\{g, Dg\} : \mathbb{R}^p \to \{\mathbb{R}^{q_1 \times q_2}, \mathbb{R}^{q_1 \times q_2 \times p}\}$. The `data` argument again denotes a function data structure. This may be distinct from the function data structure of the corresponding zero or monitor function. In more sophisticated applications, an instance of the `coco_func_data` class may be used to share data between a zero function and the corresponding adjoint function.

If the preceding call to the `coco_add_func` constructor defined a zero function, then the call to `coco_add_adjt` adds content to Λ_Φ in (3) or (4) and initializes a corresponding subset of the continuation multipliers λ_Φ. In this case, the integer $n_{\lambda_\Phi} - q_1$ is greater than or equal to the number of rows of Λ_Φ introduced in previous stages of construction. Similarly, if the associated call to `coco_add_func` defined a monitor function of type `'inactive'`, `'active'`, or `'internal'`, then the call to `coco_add_adjt` adds content to Λ_Ψ in (3) or (4) and initializes a corresponding subset of the continuation multipliers λ_Ψ. In this case, the integer $n_{\lambda_\Psi} - q_1$ is greater than or equal to the number of rows of Λ_Ψ introduced in previous stages of construction. Finally, if the associated call to `coco_add_func` defined a monitor

function of type `'inequality'`, then the call to `coco_add_adjt` adds content to Λ_G in (4) and initializes a corresponding subset of the continuation multipliers λ_G. In this case, the integer $n_{\lambda_G} - q_1$ is greater than or equal to the number of rows of Λ_G introduced in previous stages of construction.

As in the calling syntax to `coco_add_func`, the `opts` argument is a placeholder for an arbitrary sequence of additional arguments that modify the construction of the adjoint function object. For example, in the call

```
>> prob = coco_add_adjt(prob, ..., 'f+df');
```

the flag `'f+df'` indicates that the first function handle in `varargin` is of the form `@gdg`.

The integer q_1 equals the number of continuation multipliers associated with this stage of construction. These multipliers are initialized to 0 by default. The default behavior can be overridden by including the flag `'l0'` among the `opts` arguments followed by an array of real numbers of length q_1, as in the call

```
>> prob = coco_add_adjt(prob, ..., 'l0', [3.8 1.5 -0.43]);
```

where the corresponding continuation multipliers are initialized to 3.8, 1.5, and -0.43, respectively. An optional additional inclusion of the flag `'tl0'` followed by an array of real numbers of length q_1 may be used to initialize the corresponding components of a vector used to construct the initial direction of continuation.

If the preceding call to `coco_add_func` defined a monitor function of type `'inactive'`, `'active'`, or `'internal'`, then the optional inclusion of a cell array of q_1 string labels in the `par_names` argument results in the automatic association of each continuation multiplier with an element of the continuation parameter vector ν in (3) or ν_{λ_Ψ} in (4) labeled by the corresponding string and inactive by default. The optional `'active'` flag may be used to override this default behavior.

If the preceding call to `coco_add_func` defined a monitor function of type `'inequality'`, then the optional inclusion of a cell array of q_1 string labels in the `par_names` argument results in the automatic association of each continuation multiplier and corresponding monitor function with an element of Θ_G in (4) given by the nonsmooth Fischer-Burmeister complementarity function $(a, b) \mapsto \sqrt{a^2 + b^2} - a - b$, as well as an element of the continuation parameter vector ν_G in (4) labeled by the corresponding string and inactive by default. The optional `'active'` flag may be used to override this default behavior.

If the first set of optional arguments is omitted in a call to `coco_add_adjt` and provided that the preceding call to `coco_add_func` included a function handle to an encoding of the corresponding Jacobian Df, then g is assumed to equal Df, in which case $q_1 = q$ and $q_2 = p$. In this case, if a function handle to an encoding of $D^2 f$ is provided in the call to `coco_add_func`, then Dg is assumed to equal $D^2 f$.

The input dimension p is inherited from the preceding call to `coco_add_func`. The output dimensions q_1 and q_2 may be determined by evaluation of g during construction. Such evaluation is suppressed if `opts` includes the flag `'adim'` followed by a vector of two integers, assigned to q_1 and q_2, respectively.

If the top left 5×8 submatrix of Λ has been defined in previous stages of construction, then the call

```
>> prob = coco_add_adjt(prob, ..., 'adim', [3 6], 'aidx', [1 3:5]);
```

uses the flag `'aidx'` to indicate that the first four columns of the output of g should be assigned to columns 1, 3, 4, and 5 of the three rows added to Λ, while the remaining two columns of the output of g are padded from the top with five 0's and appended as entire columns to Λ.

As with `coco_add_func`, relative indexing may be used to avoid hard-coding dependencies on the detailed implementations of previous stages of construction. To this end, the call

```
>> [data, axidx] = coco_get_adjt_data(prob, fid, 'data', 'axidx');
```

provides a write-protected copy of the function data structure and an array of column indices associated with potentially nonzero columns in the rows of Λ associated with the function identifier `fid`. Similarly,

```
>> coco_get_adjt_data(prob, fid, 'afidx');
```

returns an integer array whose entries identify rows of Λ associated with the function identifier `fid`, as well as with the location in λ of the corresponding continuation multipliers. An integer array identifying the location of the corresponding continuation multipliers in the collection (u, λ, v) may be extracted from data stored to disk during continuation using the `coco_read_adjoint` utility according to the syntax:

```
>> lidx = coco_read_adjoint(fid, run, lab, 'lidx');
```

where `run` is a string that denotes the run identifier and `lab` is an integer that identifies the solution label.

The call

```
>> coco_get_adjt_data(prob, 'adjoint', 'lidx');
```

may be used to obtain the indices in λ of all continuation multipliers constructed using `coco_add_adjt` or similar toolbox constructors. More refined access to the indices corresponding to λ_Φ, λ_Ψ, and λ_G is afforded by each of the calls

```
>> coco_get_adjt_data(prob, 'adjoint', 'lidx_Phi');
>> coco_get_adjt_data(prob, 'adjoint', 'lidx_Psi');
>> coco_get_adjt_data(prob, 'adjoint', 'lidx_G');
```

These may be used to impose additional conditions on the continuation multipliers in a subsequent call to `coco_add_comp`.

7.3 Complementary zero and monitor functions - `coco_add_comp`

As described in the previous section, the special forms of Θ in (3) and (4) may be obtained by the default response of `coco_add_adjt` to the inclusion of string labels for the corresponding

components of ν in (3) and (4) and ν_G in (4). We construct a function object to represent general elements of Ξ or Θ and append this to a partially implemented continuation problem structure `prob` by adhering to the appropriate argument syntax for the `coco_add_comp` constructor:

```
>> prob = coco_add_comp(prob, fid, varargin);
```

where

```
varargin = ( @f, [ @df,] | @fdf, ) data, type_spec, opts
```

As with `coco_add_func`, the *function identifier* `fid` denotes a string variable that is uniquely identified[8] with this call to `coco_add_comp` and that can be used to reference the function object that is instantiated in this call.

The argument `@f` denotes a function handle to a COCO-compatible encoding of a *realization* $f : \mathbb{R}^{p_1} \times \mathbb{R}^{p_2} \times \mathbb{R}^{p_3} \to \mathbb{R}^q$, `@df` denotes a function handle to a COCO-compatible encoding of the function $Df : \mathbb{R}^{p_1} \times \mathbb{R}^{p_2} \times \mathbb{R}^{p_3} \to \mathbb{R}^{q \times p_1} \times \mathbb{R}^{q \times p_2} \times \mathbb{R}^{q \times p_3}$ whose component functions are first partial derivatives of f with respect to its arguments. The integer p_3 is less than or equal to the number of complementary continuation variables introduced in this and previous stages of construction. For a complementary zero function, the integer $n_\Xi - q$ is greater than or equal to number of components of Ξ introduced in previous stages of construction. For a complementary monitor function, $n_\Theta - q$ is greater than or equal to the number of components of Θ introduced in previous stages of construction. In lieu of separate encodings for f and Df, the notation `@fdf` denotes a function handle to a COCO-compatible encoding $\{f, Df\} : \mathbb{R}^{p_1} \times \mathbb{R}^{p_2} \times \mathbb{R}^{p_3} \to \{\mathbb{R}^q, \mathbb{R}^{q \times p_1} \times \mathbb{R}^{q \times p_2} \times \mathbb{R}^{q \times p_3}\}$.

As with `coco_add_func`, the *function data structure* `data` contains a structure array with function-specific content that can be accessed and modified by the encodings of f and Df. If this variable is an instance of the `coco_func_data` class, then its content may be accessed and modified by any other function to which the variable is sent. A write-protected copy of the function data structure associated with the function identifier `fid` is returned by the call

```
>> coco_get_comp_data(prob, fid, 'data')
```

Such a write-protected copy should not be passed as the function data structure argument in another call to `coco_add_comp`. As with zero and monitor functions, the function data structure contains information that can be precomputed and reused in multiple calls to the encodings of f and its derivatives, for example during the application of a sequence of Newton iterations. Adaptive updates to `data` during continuation are associated with changes to problem discretizations or problem parameterizations.

The argument `type_spec` is the single string `'zero'` in the case that f represents a realization of elements of Ξ. In the case that f represents a realization of elements of Θ, `type_spec` is one of the strings `'active'`, `'inactive'`, or `'internal'` followed by a cell array of q string labels assigned to the corresponding embedded complementary continua-

[8]There can be no duplicate use of function identifiers in separate calls to `coco_add_func`, `coco_add_comp`, or special-purpose wrappers to either of these constructors.

tion parameters, which are designated as initially active, inactive, or active, respectively. Initially inactive complementary continuation parameters may be activated by an exchange with an active (complementary) continuation parameter using the `coco_xchg_pars` utility, or by explicitly releasing them in the call to the `coco` entry-point function. Initially active complementary continuation parameters may be deactivated by an exchange with an inactive (complementary) continuation parameter using `coco_xchg_pars` or, in the case of parameters labeled as `'internal'`, by an automatic exchange with overspecified inactive (complementary) continuation parameters in the call to the `coco` entry-point function.

In identical fashion to `coco_add_func`, the argument `opts` is a placeholder for an arbitrary sequence of additional arguments that modify the construction of the function object. For example, in the call

```
>> prob = coco_add_comp(prob, ..., 'f+df');
```

the flag `'f+df'` indicates that the first function handle in `varargin` is of the form `@fdf`. In the call

```
>> prob = coco_add_comp(prob, ..., 'fdim', 3);
```

the flag `'fdim'` indicates that the output dimension q equals 3, thereby eliminating the need to determine q by evaluation of the function f during construction.

For elements of Ξ or Θ, the input argument to the corresponding function f is populated at run-time with a subset of p_1 elements of u, p_2 elements of λ, and p_3 elements of v, indexed by function-specific, ordered, dependency-index sets $\mathbb{K}_u$, $\mathbb{K}_\lambda$, and $\mathbb{K}_v$. Note that only complementary continuation variables may be added to a continuation problem in a call to `coco_add_comp`. For example, if 5 continuation variables, 6 continuation multipliers, and 8 complementary continuation variables have been introduced in previous stages of construction, then the call

```
>> prob = coco_add_comp(prob, ..., 'uidx', 1:3, 'lidx', [1 3 5], ...
     'vidx', [2 4:6], 'v0', [0.3 2.5]);
```

results in the assignments $\mathbb{K}_u = \{1, 2, 3\}$, $\mathbb{K}_\lambda = \{1, 3, 5\}$, $\mathbb{K}_v = \{2, 4, 5, 6, 9, 10\}$ and $v_{0,\{9,10\}} = (0.3, 2.5)$. When the `'v0'` flag is present, an optional additional inclusion of the flag `'tv0'` as in this call

```
>> prob = coco_add_comp(prob, ..., 'v0', [0.3 2.5], 'tv0', [1.4 3.9]);
```

results in the assignment $t_{v0,\{9,10\}} = (1.4, 3.9)$ of components of a vector used in the construction of the initial direction of continuation. When the `'t0'` flag is not present, these components default to 0.

For a representation of elements of Ξ or Θ, copies of the function dependency index sets $\mathbb{K}_u$, $\mathbb{K}_\lambda$, and $\mathbb{K}_v$, respectively, may be obtained with the calls

```
>> coco_get_comp_data(prob, fid, 'uidx');
>> coco_get_comp_data(prob, fid, 'lidx');
>> coco_get_comp_data(prob, fid, 'vidx');
```

In a typical application, the call

```
>> [data, vidx] = coco_get_comp_data(prob, fid, 'data', 'vidx');
```

may be followed by a construction of the form

```
>> prob = coco_add_comp(prob, ..., 'vidx', vidx(data.x_idx));
```

where `vidx(data.x_idx)` evaluates to a subset of the $\mathbb{K}_v$ function dependency index set of the function with function identifier `fid`, indexed by the `x_idx` field of the function data structure of this function. This type of formulation uses the relative indexing of `data.x_idx` to accommodate any dependence on preceding stages of construction, without necessitating explicit reference to the detailed implementation of each such stage. An integer array identifying the location of the corresponding complementary continuation variables in the collection (u, λ, v) may also be extracted from data stored to disk during continuation using the `coco_read_complementary` utility according to the syntax

```
>> coco_read_complementary(fid, run, lab, 'vidx');
```

where `run` is a string that denotes the run identifier and `lab` is an integer that identifies the solution label.

7.4 Special purpose wrappers

The COCO release includes several special purpose wrappers that provide shortcuts to particular forms of problem construction. Different calling syntaxes separate default behaviors from more sophisticated constructions.

`coco_add_pars`: The `coco_add_pars` special-purpose wrapper obeys the calling syntax

```
>> prob = coco_add_pars(prob, fid, varargin);
```

where

```
varargin = ( pidx  par_names | par_names pvals [ tvals ] ) [ par_type ]
```

and

```
par_type = ( 'active' | 'inactive' | 'internal' )
```

Specifically, provided that the number of integers in `pidx` equals the number of string labels in `par_names`, the command

```
>> prob = coco_add_pars(prob, fid, pidx, par_names, par_type);
```

extends the continuation problem structure with embedded monitor functions of function type `par_type` that evaluate to previously introduced continuation variables identified by the index array `pidx`, and continuation parameters labeled by the elements of `par_names`. Here, the omission of `par_type` results in continuation parameters that are initially inactive.

Similarly, provided that the number of string labels in `par_names` equals the number of elements of `pvals` (and, if included, `tvals`), the embedded monitor functions appended to `prob` with the command

```
>> prob = coco_add_pars(prob, fid, par_names, pvals, tvals, par_type);
```

evaluate to new continuation variables initialized by elements of `pvals`. Here, inclusion of `tvals` assigns potentially nonzero values to the corresponding components of the vector used to construct the initial direction of continuation. This second usage of `coco_add_pars` can be used to introduce continuation variables without associated elements of Φ or Ψ.

`coco_add_glue`: The `coco_add_glue` special-purpose wrapper obeys the calling syntax

```
>> prob = coco_add_glue(prob, fid, varargin);
```

where

```
varargin = x1_idx x2_idx [ gap ] [ ( par_names [ par_type ] | 'zero' ) ]
```

and

```
par_type = ( 'active' | 'inactive' | 'internal' )
```

Specifically, provided that `x1_idx` and `x2_idx` are equal-length index arrays, the command

```
>> prob = coco_add_glue(prob, fid, x1_idx, x2_idx, gap, 'zero');
```

extends the continuation problem structure with zero functions that evaluate to pairwise differences between continuation variables identified by the index arrays `x1_idx` and `x2_idx` and added to the numerical value `gap`, which defaults to 0 when omitted. The command

```
>> prob = coco_add_glue(prob, fid, x1_idx, x2_idx, gap, par_names, par_type);
```

instead extends the continuation problem structure with corresponding embedded monitor functions of function type `par_type` and associated continuation parameters that are labeled by the elements of `par_names` and initially inactive by default.

`coco_add_functionals`: The `coco_add_functionals` special-purpose wrapper obeys the calling syntax

```
>> prob = coco_add_functionals(prob, fid, varargin);
```

where

```
varargin = A b x_idx [ par_names ( 'active' | 'inactive' | 'internal' ) ]
```

and the number of rows of A equals the number of elements in the one-dimensional array b. Specifically, provided that `x_idx` is a one-dimensional index array with as many elements as the number of columns of the matrix A, the command

```
>> prob = coco_add_functionals(prob, fid, A, b, x_idx);
```

extends the continuation problem structure with zero functions that evaluate to $Au-b$, where u consists of continuation variables indexed by `x_idx`. If, instead, `x_idx` is a two-dimensional array of the same dimensions as A, the same command extends the continuation problem structure with zero functions that evaluate to $\hat{A}u - b$, where u consists of continuation variables indexed by the concatenation of the rows of `x_idx`, and $\hat{A}$ is a block diagonal matrix with blocks given by the rows of A. The inclusion of the optional arguments `par_names` and `par_type` result in an encoding of monitor functions of function type given by `par_type` and associated with continuation parameters with string labels given by `par_names`.

`coco_add_comp_pars:` The `coco_add_comp_pars` special-purpose wrapper obeys the calling syntax

```
>> prob = coco_add_comp_pars(prob, fid, pidx, par_names, varargin);
```

where

```
varargin = [ ( 'active' | 'inactive' | 'internal' ) ]
```

Specifically, provided that the number of integers in `pidx` equals the number of string labels in `par_names`, the command

```
>> prob = coco_add_comp_pars(prob, fid, pidx, par_names, par_type);
```

extends the continuation problem structure with embedded complementary monitor functions of function type `par_type` that evaluate to previously introduced continuation multipliers identified by the index array `pidx`, and complementary continuation parameters labeled by the elements of `par_names`. Here, the omission of `par_type` results in complementary continuation parameters that are initially inactive.

`coco_add_complementarity:` The `coco_add_complementarity` special-purpose wrapper obeys the calling syntax

```
>> prob = coco_add_complementarity(prob, fid, varargin);
```

where `fid` is the function identifier used in a preceding call to `coco_add_func` to append a monitor function $u \mapsto G(u)$ of function type `'inequality'` to the continuation problem structure `prob`,

```
varargin = [ fhan [ dfhan ] v0 [ tv0 ] ] par_names
```

and the number of string labels in `par_names` equals the output dimension of G. If the corresponding continuation multipliers are denoted by λ_G, then the command

```
>> coco_add_complementarity(prob, fid, par_names);
```

extends the continuation problem structure with the complementary monitor function

$$(u, \lambda, v) \mapsto \Theta_G(\lambda_G, -G(u)), \tag{54}$$

where Θ_G is a vectorized implementation of the Fischer-Burmeister complementarity function $(a, b) \mapsto \sqrt{a^2 + b^2} - a - b$ for $a, b \in \mathbb{R}$. The associated continuation parameters are labeled by the string labels given in `par_names`.

The default assignment of the Fischer-Burmeister function may be replaced by a function $(a, b, v) \mapsto f(a, b, v) \in \mathbb{R}$ for $a, b \in \mathbb{R}$ and some parameterization v using the following calling syntax:

```
>> coco_add_complementarity(prob, fid, fhan, dfhan, v0, tv0, par_names);
```

Here, `fhan` is a function handle to a vectorized (in the first two arguments) encoding of f. The corresponding vectorized implementation of the function Jacobian is optionally given in `dfhan`. The parameterization v corresponds to a set of new complementary continuation variables that are initialized with `v0`. The optional argument `tv0` assigns potentially nonzero values to the corresponding components of the vector used to construct the initial direction of continuation.

8 Data processing and visualization

During continuation, two forms of data are stored to disk for later processing. Small amounts of data associated with all successfully located points on the solution manifold are recorded in a single location in order to enable analysis and visualization of properties of the solution manifold as a whole. Large amounts of data associated with each of a sampled selection of successfully located points along the solution manifold are recorded in a sequence of separate files in order to enable analysis and visualization of properties of individual solutions.

We refer to data describing properties of the solution manifold as a whole, rather than a subset of individual points, as *bifurcation data* and use the abbreviation `bd` in associated COCO commands and scripts. For example, to extract saved bifurcation data for further processing, we use the `coco_bd_col` utility, as shown below.

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

Here, `run` is the run identifier associated with the stored data. This command assigns a rectangular cell array to `bd`. This array includes a single header row with string labels identifying the content of each column. The `coco_bd_col` utility can be used to extract data from the column with string label `name`, as shown below.

```
>> coco_bd_col(bd, name)
```

Data associated with multiple columns can be extracted by replacing `name` with a cell array of corresponding string labels.

The utility `coco_plot_bd` can be used to visualize bifurcation data associated with a specific continuation run. A call to `coco_plot_bd` must adhere to the following argument syntax:

```
[theme], run, [col1, [idx1], [col2, [idx2], [col3, [idx3]]]]
```

Here, the `run` argument is the run identifier associated with the stored bifurcation data. In the absence of any further arguments, `coco_plot_bd` executes a behavior defined by a default visualization theme.

As an example, the `coco_plot_theme` utility defines the default visualization theme for a family of solution points that are not associated with a particular toolbox. The command

```
>> thm = coco_plot_theme();
```

assigns the corresponding struct to the `thm` variable. For a family of solution points associated with a particular toolbox, a toolbox-specific visualization theme defines the default behavior. The optional argument `theme` in the call to `coco_plot_bd` is a struct whose fields substitute for, or add to, the content of the default visualization theme, in order to override the default behavior or define new behaviors. To use a toolbox-specific visualization theme associated with a toolbox instance in a composite continuation problem, assign the corresponding object instance identifier to the `oid` field of the `theme` argument.

By default, `coco_plot_bd` produces a two-dimensional graph of simultaneous variations in two quantities that are each computable from the bifurcation data. As an example, the command below generates a two-dimensional plot of a piecewise-linear interpolant connecting points with coordinates given by data in the `'col1'` and `'col2'` columns of the bifurcation data cell array.

```
>> coco_plot_bd(run, 'col1', 'col2')
```

Similarly, the command below generates a three-dimensional plot of a piecewise-linear interpolant connecting points with coordinates given by data in the `'col1'`, `'col2'`, and `'col3'` columns of the bifurcation data cell array.

```
>> coco_plot_bd(run, 'col1', 'col2', 'col3')
```

In the case of a two-dimensional plot, it is possible to omit both or only the second column labels, provided that the visualization theme includes default labels in the `bd.col1` and/or `bd.col2` fields, respectively. Notably, for the default visualization theme defined by `coco_plot_theme`, the fields `bd.col1` and `bd.col2` are empty.

In general, the arguments `col1`, `col2` and, in the case of three-dimensional graphs, `col3` are either single string labels or cell arrays of string labels associated with columns of the bifurcation data cell array with numerical content. The optional arguments `idx1`, `idx2`, and `idx3` are either single integers or handles to vectorized functions. In the former case, the preceding argument must be a single string label. The integer then defines a component of the numerical array in the corresponding column of the bifurcation data. In the command

```
>> coco_plot_bd(run, 'col1', 3, 'col2')
```

the integer 3 indicates that the horizontal coordinate is given by the third component of the data in each row of the `'col1'` column. In contrast, the command

```
>> coco_plot_bd(run, 'col1', 'col2')
```

is equivalent to the command

```
>> coco_plot_bd(run, 'col1', 1, 'col2', 1)
```

while the command

```
>> coco_plot_bd(run, 'col1', 'col1')
```

is equivalent to the command

```
>> coco_plot_bd(run, 'col1', 1, 'col1', 2)
```

which, of course, throws an error if the `'col1'` column contains scalar data.

In the case that the `idx1`, `idx2`, or `idx3` optional argument is a function handle, then the number of inputs to this function must equal the number of string labels in the preceding argument. The corresponding function must return a one-dimensional array obtained by applying a suitable operation to the content of the corresponding columns of the bifurcation data. As an example, in the command

```
>> coco_plot_bd(run, 'col1', 3, {'col2', 'col3'}, @(x,y) x+y)
```

the fourth and fifth arguments indicate that the vertical coordinate is given by the sum of the data in the `'col2'` and `'col3'` columns.

The utility `coco_plot_sol` provides an interface to toolbox-specific visualization of properties of individual solutions from a specific continuation run. A call to `coco_plot_sol` must adhere to the following argument syntax:

```
[theme], run, [labs], oid, [oidx], [col1, [idx1], [col2, [idx2], [col3, [idx3]]]]
```

The meaning of the `run` and `theme` arguments is identical to the case of `coco_plot_bd`. To visualize the properties associated with a subset of solutions along the solution manifold, the corresponding solution labels are assigned to the optional `labs` argument. In its absence, all stored solutions are visualized in the same plot.

To visualize solution properties associated with a single toolbox instances in a composite continuation problem, assign the corresponding object identifier to the `oid` argument. To visualize solution properties associated with multiple instances of the same toolbox with object identifiers of the form `'oid1'`, `'oid2'`, and so on, assign the string `'oid'` to the `oid` argument and the corresponding integer array to the optional `oidx` argument.

The behavior of `coco_plot_sol` is determined by a toolbox-specific visualization theme. Such a visualization theme defines string labels that may be used in the `col1`, `col2` and, as applicable, `col3` arguments, in addition to the headers for columns of bifurcation data. As an example, the `'coll'` toolbox visualization theme supports use of the `'t'` and `'x'` string labels in order to generate two- or three-dimensional visualizations of the spacetime trajectory segment. The optional `idx1`, `idx2`, and `idx3` arguments play the same role for `coco_plot_sol` as in the case of `coco_plot_bd`. Examples of their use are included with the toolbox demos.

An alternative use of `coco_plot_sol` relies on assigning a function handle to the `plot_sol`

field of the optional `theme` argument. An example of such use is demonstrated in the `pdeeig` demo of the `'ep'` toolbox.

Exercises

1. Use an example to show that the command

   ```
   >> coco_plot_bd(run, 'col1', 'col2', 'col2')
   ```

   is equivalent to the command

   ```
   >> coco_plot_bd(run, 'col1', 1, 'col2', 1, 'col2', 2)
   ```

2. Use an example to show that the command

   ```
   >> coco_plot_bd(run, 'col1', @(x) x(1,:) 'col1', @(x) x(2,:))
   ```

   is equivalent to the command

   ```
   >> coco_plot_bd(run, 'col1', 'col1')
   ```

3. The `coco_plot_bd` utility uses the `lspec`, `ustab`, `ustabfun`, and `usept` properties of the visualize theme to highlight different parts of a solution manifold according to properties of the corresponding solutions. Investigate the corresponding implementation in the `'ep'` and `'po'` toolbox visualization themes, and construct an example in which three different line styles are used to differentiate portions of the solution manifold of a continuation problem.
