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

November 15, 2017

Contents

1 Problem formulation

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

$$\begin{pmatrix} \Phi(u) \\ \Psi(u) - \mu \\ \Lambda_\Phi^\top(u)\lambda + \Lambda_\Psi^\top(u)\eta \\ \eta - \nu \end{pmatrix} = 0 \tag{1}$$

where $\Phi : \mathbb{R}^n \to \mathbb{R}^m$, $\Psi : \mathbb{R}^n \to \mathbb{R}^s$, $\Lambda_\Phi : \mathbb{R}^n \to \mathbb{R}^{l\times k}$, and $\Lambda_\Psi : \mathbb{R}^n \to \mathbb{R}^{h\times k}$. Elements of Φ are called *zero functions*. Elements of Ψ are called *monitor functions*. Elements of Λ_Ψ and Λ_Ψ are called *adjoint functions*. Elements of the vector u are called *continuation variables*. Elements of μ and ν are called *continuation parameters*. Elements of λ and η are called *continuation multipliers*. The representation in COCO of the adjoint conditions is of the form $\pi_{\{\lambda,\eta\}}^\top \Lambda(u) = 0$, where $\pi_{\{\lambda,\eta\}}$ is some permutation of the continuation multipliers, determined during construction, and $\Lambda^\top(u)$ is a corresponding permutation of the columns of $\begin{pmatrix} \Lambda_\Phi^\top(u) & \Lambda_\Psi^\top(u) \end{pmatrix}$.

The vectors u, λ, and η are said be *initialized* when they are associated with identically-sized numerical vectors u_0, λ_0, η_0. Unless otherwise stated, the vectors μ and ν are initialized with $\Psi(u_0)$ and η_0, respectively.

Consider an indexing of the elements of μ by integers in the set $\{1, \ldots, s\}$ and of the elements of ν by integers in the set $\{s+1, \ldots, s+h\}$. Then, during continuation, the index sets $\mathbb{I}_\mu \subseteq \{1, \ldots, s\}$ and $\mathbb{I}_\nu \subseteq \{s+1, \ldots, s+h\}$ identify continuation parameters that are fixed and not included among the unknowns whose values are defined implicitly by the continuation problem. Continuation parameters indexed by 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., $n + l + h - |\mathbb{I}_\mu| - |\mathbb{I}_\nu| - m - k$. When this is greater than 0, it equals the dimension of the unique solution manifold through any regular solution point.

The general form of the continuation problem is implemented in COCO in one of two possible ways. In the first, and less common approach, a call to the `coco_add_func` constructor is used to define the function Φ and to initialize u. This is followed by another call to the `coco_add_func` constructor to define the function Ψ, associate string labels with the elements of μ, and initialize the index set $\mathbb{I}_\mu$. This is then followed by a call to the `coco_add_adjt` constructor in order to define the function Λ_Φ and to initalize λ. Finally, a second call to the `coco_add_adjt` constructor is used to define the function Λ_Ψ, initialize η, associate string labels with the elements of ν, and initialize the index set $\mathbb{I}_\nu$. Following this initial construction, elements of $\mathbb{I}_\mu \cup \mathbb{I}_\nu$ may be removed without replacement or switched one-to-one with elements in $\{1, \ldots, s+h\} \setminus (\mathbb{I}_\mu \cup \mathbb{I}_\nu)$ 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 $\tilde{\Phi}$, $\tilde{\Psi}$, and $\tilde{\Lambda}$, vectors $\tilde{u}$, $\tilde{\mu}$, $\tilde{\pi}_{\{\tilde{\lambda},\tilde{\eta}\}}$, and $\tilde{\nu}$, with $\tilde{u}$, $\tilde{\lambda}$, $\tilde{\eta}$, $\tilde{\mathbb{I}}_\mu$, and $\tilde{\mathbb{I}}_\nu$ initialized. This staged approach to problem

construction supports the development of toolboxes dedicated to constructing embeddable subproblems and appending these to an existing continuation problem structure. Such toolboxes encapsulate one or several calls to the `coco_add_func` constructor and at most an equal number of calls to the `coco_add_adjt` constructor.

2 Staged construction

In the COCO paradigm of staged construction, a general continuation problem may be represented in terms of two Boolean matrices, associated with calls to `coco_add_func` and `coco_add_adjt`, respectively, that 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. An example of such a pair of matrices is shown below.

In general, the first of the two matrices has n columns, representing the elements of the vector of continuation variables u, in order. Each call to the `coco_add_func` constructor 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 call to `coco_add_func`.

The k columns of second of the two 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 reference 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 $\Lambda(u)$.

3 Constructor syntax

We construct a function object to represent a zero or monitor function 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_func`.

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 $D^2 f : \mathbb{R}^p \to \mathbb{R}^{q \times p \times p}$ 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 $m - q$ is greater than or equal to number of components of Φ introduced in previous stages of construction. For a monitor function, $s - q$ 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 $D^2 f$. 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')
```

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 discretizations of an infinite-dimensional problem or with 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 a zero function. For a monitor function, `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;
- `'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 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 continuation parameter using `coco_xchg_pars` or, in the case of parameters labeled as `'internal'`, by an automatic exchange with over-specified inactive 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. While 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 with nonsingular problem Jacobian or approximate detection of singular special points 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.

The input argument to the 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 parallel to the initial direction of continuation. When the `'t0'` flag is not present, these components default to 0.

A copy of the function dependency index set 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, fid1, '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 `fid1`, 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. 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.

Each call to `coco_add_func` 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 and initializes a corresponding subset of the continuation multipliers λ. In this case, the integer $l - q_1$ 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, then the call to `coco_add_adjt` adds content to Λ_Ψ and initializes a corresponding subset of the continuation multipliers η. In this case, the integer $h - q_1$ is greater than or equal to the number of rows of Λ_Ψ introduced in previous stages of construction.

As in the calling syntax to `coco_add_func`, the `opts` argument is a placeholder for an ar-

bitrary 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 the vector parallel to the initial direction of continuation.

If the preceding call to `coco_add_func` defined a monitor function, then each continuation multiplier is associated with an element of the continuation parameter vector ν. In this case, the argument `par_names` must contain a cell array of q_1 string labels assigned to these parameters, which are 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`, then g is assumed to equal Df, in which case $q_1 = q$ and $q_2 = p$. If a function handle to $D^2 f$ 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 of the corresponding continuation multipliers in $\pi_{\{\lambda,\eta\}}$. This integer array may also 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.

4 Application – **`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

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

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

$$\|u\|^2 - 1 = 0,\ \sum_{i=1}^{4} u_i - \mu_{\text{sum}} = 0,\ u - \mu_u = 0,\ 2\lambda u + \eta_{\text{sum}}\mathbf{1} + \eta_u = 0, \quad (3)$$

$1 - \eta_{\text{sum}} = 0$, 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[1] applied to the extended continuation problem obtained by combining (3) 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

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

the corresponding restricted continuation problem of the form

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

are located on three one-dimensional manifolds given by

$$\mu_{u,1} = \rho \cos\theta,\ \mu_{u,4} = \rho \sin\theta,\ \lambda = \nu_{\text{sum}} = \nu_{u,2} = \nu_{u,3} = 0 \tag{5}$$

and

$$\mu_{u,1} = \mu_{u,4} = \pm\frac{\rho}{\sqrt{2}},\ \lambda = \mp\frac{\nu_{\text{sum}}}{\sqrt{2}\rho}, \tag{6}$$

$$\nu_{u,2} = \nu_{\text{sum}} \left(\pm\frac{\sqrt{2}\mu_{u,2}}{\rho} - 1 \right),\ \nu_{u,3} = \nu_{\text{sum}} \left(\pm\frac{\sqrt{2}\mu_{u,3}}{\rho} - 1 \right) \tag{7}$$

parameterized by θ and ν_{sum}, respectively. The manifolds in (6) intersect the manifold in (5) at the points given by

$$\mu_{u,1} = \mu_{u,4} = \pm\frac{\rho}{\sqrt{2}},\ \lambda = \nu_{u,2} = \nu_{u,3} = 0, \tag{8}$$

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

$$\mu_{u,1} = \mu_{u,4} = \pm\frac{\rho}{\sqrt{2}},\ \mu_{u,2}^2 = 1 - \rho^2 - \mu_{u,3}^2,\ \lambda = \mp\frac{1}{\sqrt{2}\rho} \tag{9}$$

$$\nu_{u,2} = \pm\frac{\sqrt{2}\mu_{u,2}}{\rho} - 1,\ \nu_{u,3} = \pm\frac{\sqrt{2}\mu_{u,3}}{\rho} - 1, \tag{10}$$

parameterized by ρ.

There is a unique point along each of the tertiary manifolds in (9)-(10) at which $\eta_{u,2} = 0$, obtained with

$$\rho = \sqrt{\frac{2}{3}(1 - \mu_{u,3}^2)},\ \mu_{u,2} = \pm\sqrt{\frac{1}{3}(1 - \mu_{u,3}^2)}. \tag{11}$$

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

$$\mu_{u,1} = \mu_{u,2} = \mu_{u,4} = \pm\frac{\rho}{\sqrt{2}},\ \mu_{u,3}^2 = 1 - \frac{3}{2}\rho^2,\ \lambda = \mp\frac{1}{\sqrt{2}\rho},\ \nu_{u,3} = \pm\frac{\sqrt{2}\mu_{u,3}}{\rho} - 1, \tag{12}$$

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 the extended continuation problem in COCO 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 inactive and two 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

$$\Phi : u \mapsto \|u\|^2 - 1, \; \Psi : u \mapsto \begin{pmatrix} u_1 + u_2 + u_3 + u_4 \\ u_2 \\ u_3 \\ u_1 \\ u_4 \end{pmatrix}, \tag{13}$$

$u_0 = (1, 0, 0, 0)$, 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 $\tilde{\Lambda} : u \mapsto \begin{pmatrix} 2u_1 & 2u_2 & 2u_3 & 2u_4 \end{pmatrix}$ and $\tilde{\lambda}_0 = 0$. Similarly, the call

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

results in

$$\tilde{\Lambda} : u \mapsto \begin{pmatrix} 2u_1 & 2u_2 & 2u_3 & 2u_4 \\ 1 & 1 & 1 & 1 \end{pmatrix} \tag{14}$$

and $\tilde{\eta}_0 = 0$. 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

$$\Lambda : u \mapsto \begin{pmatrix} 2u_1 & 2u_2 & 2u_3 & 2u_4 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{15}$$

with $\pi_{\{\lambda_0,\eta_0\}} = 0$ and $\mathbb{I}_\nu = \{4, 5, 6, 7, 8\}$. The second of the two commands below is then equivalent to the computation of the manifold in (5) 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 (5). 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 (6)-(7) 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 (9)-(10) 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 (12) 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. 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.

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

---

5 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 included 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.
