Coupling COCO with fmincon for constrained optimization of dynamical systems

Mingwu Li

July 27, 2020

Contents

1 Introduction

This tutorial presents how to solve constrained optimization of dynamical systems using packages COCO [1, 2] and `fmincon` [3]. `fmincon` is a MATLAB nonlinear programming solver for finding the minimum of *finite-dimensional* constrained optimization problems [REF]. However, the constrained optimization of dynamical systems is *infinite-dimensional*, which can be approximated by a finite-dimensional one with discretization. Here, we use toolboxes in COCO to perform such discretization.

In an optimization problem of a dynamical system, the state equations are infinite-dimensional constraints, time-dependent system states and control inputs are infinite dimensional design variables, and the optimization objective can be a functional of system states and control inputs. COCO includes a `coll` toolbox, which provides discretization to trajectory segments of autonomous and non-autonomous dynamical systems. The approximation of an integral can be easily implemented with consistent discretization in COCO. In this repository [REF], we also provide a toolbox for differential-algebraic equations (DAEs) to support discretization to trajectory segments in DAE systems.

The rest of this tutorial is organized as follows: we give a brief introduction to COCO and `fmincon` in section 2. We then present a few wrapper functions to couple this two packages in section 3. Several optimization examples are solved to demonstrate the effectiveness of the wrapper functions. Each example corresponds to fully documented code in the `coco_fmincon/examples` folder of this repository. We conclude this report with some discussions on future work.

2 A brief introduction to COCO and fmincon

2.1 COCO

Continuation Core (COCO) is a MATLAB-based open-source package for computational nonlinear analysis of dynamical systems. A unique feature of COCO is the embedded construction philosophy of nonlinear systems, where a large problem is assembled from small subproblems with weak couplings. This object-oriented construction paradigm enables us to construct a composite problem from building blocks.

COCO provides a predefined library for the building blocks. Specifically, it has fully documented toolboxes for nonlinear analysis of dynamical systems as follows

- `ep` toolbox for the equilibria of smooth dynamical systems;
- `coll` toolbox for the trajectory segments of autonomous system $\dot{x}(t) = f(x(t), p)$ and non-autonomous system $\dot{x}(t) = f(t, x(t), p)$; and
- `po` toolbox for the periodic orbits of dynamical systems.

In this repository, we also provide a toolbox for the trajectory segments of DAE systems in the form $\dot{x}(t) = f(t, x(t), y(t), p)$ and $g(t, x(t), y(t), p) = 0$, where $x(t)$ and $y(t)$ are referred to as *differential* and *algebraic* variables respectively. This toolbox can be used in optimal control problems where the control inputs are interpreted as algebraic variables.

2.2 fmincon

`fmincon` is a nonlinear programming solver for optimization problems in the following form [ref]

$$\min f(x), \text{ such that } \begin{cases} c(x) \leq 0 \\ ceq(x) = 0 \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub \end{cases} \tag{1}$$

where $f(x)$ is a scaler valued function of x, $c(x)$ and $ceq(x)$ can be vector valued nonlinear functions of x.

A syntax of `fmincon` is given by `x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)`, where

- `fun`: a function that returns the objective function $f(x)$;
- `x0`: initial point;
- `A,b`: the coefficient matrix and right-hand side vector in linear inequalities $A \cdot x \leq b$;
- `Aeq,beq`: the coefficient matrix and right-hand side vector in linear equalities $Aeq \cdot x = beq$;
- `lb,ub`: the lower and upper bounds of x; and
- `nonlcon`: a function that returns two arrays $c(x)$ and $ceq(x)$, which gives *nonlinear* inequality constraints $c(x) \leq 0$ and equality constraints $ceq(x) = 0$ respectively.

3 Wrapper functions

In COCO, linear systems and nonlinear systems are not differentiated. In addition, linear constraints can be regarded as a special type of nonlinear constraints. It follows that the syntax can be reduced to `x = fmincon(fun,x0,[],[],[],[],[],[],nonlcon)`. We need two wrapper functions corresponding to the objective function `fun` and constraints `nonlcon` respectively.

We present the following two wrapper functions

- `[y, Dy] = objfunc(u, prob, objfid)` for `fun`. The objective function is defined as a monitor function in `prob` with function identifier `objfid`. Here `prob` is a continuation problem structure with a collection of zero functions and monitor functions, and `u` is a collection of continuation variables. More details about `prob` and `u` can be found at [REF].

- `[c, y, Dc, Dy] = nonlincons(u, prob)` for `nonlcon`. All *zero* functions in `prob` will be defined as equality constraints and all *inequality* monitor functions in `prob` will be defined as inequality constraints.

We also provide a few wrapper functions for data processing and visualization

- `[x,y] = opt_read_sol(u, prob, fid)`. This function returns the design variables `x` specific to the function with identifier `fid`, and the evaluation of this function at such variables.

- `sol = opt_read_coll_sol(u, prob, oid)`. This function returns a trajectory segment corresponds to the collocation object with identifier `oid`.

- `sol = opt_read_ddaecoll_sol(u, prob, oid)`. This function returns trajectory segment corresponds to a collocation object for the DAE system with identifier `oid`.

4 An algebraic optimization – `alg`

Consider $\min(x^2 + y)$ s.t. $y = 1$. One can easily find the optimal solution is given by $(x, y) = (0, 1)$.

The sequence of MATLAB commands

```
>> fcn  = @(p,d,u) deal(d, u(2)-1);
>> obj  = @(p,d,u) deal(d, u(1)^2+u(2));
```

define the anonymous functions `fcn` to represent the constraint, and `obj` to represent the objective. For initial point $(x_0, y_0) = (1, 2)$, we proceed to append the zero function `fcn` and monitor function `obj` to the continuation problem structure `prob`, as shown in the following sequence of commands

```
>> u0   = [1;2]; % initial point
>> prob = coco_prob;
>> prob = coco_add_func(prob, 'constraint', fcn, [], 'zero', 'u0', u0);
>> prob = coco_add_func(prob, 'obj', obj, [], 'inactive', 'obj', 'uidx', [1;2]);
```

Now we can call `fmincon` with the developed wrapper functions `objfunc` and `nonlincons` as follows

```
x  = fmincon(@(u) objfunc(u,prob,'obj'), u0,[],[],[],[],[],[],...
    @(u) nonlincons(u,prob));
```

5 Optimization of periodic orbits – **linode**

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

$$\dot{x}_1 = x_2, \quad \dot{x}_2 = -x_2 - kx_1 + \cos(t + \theta) \tag{2}$$

with period 2π. Analytical studies show that minima occur at $k = 1$ and $\theta = (2n + 1)\pi$ for any integer n.

We proceed to implement a continuation problem in COCO using the `coll` toolbox. Specifically, we use `ode_isol2coll` toolbox constructor to encode the trajectory constraint as follows

```
>> [t0, x0]  = ode45(@(t,x) linode(t, x, [0.98; 0.3]), [0 2*pi], ...
   [0.276303; 0.960863]); % Initial trajectory
>> prob = coco_prob;
>> prob = coco_set(prob, 'ode', 'autonomous', false);
>> coll_args = {@linode, @linode_dx, @linode_dp, @linode_dt,...
     t0, x0, {'k' 'th'}, [0.98; 0.3]};
>> prob = ode_isol2coll(prob, '', coll_args{:});
```

Here `linode` denotes the vector field of the linear dynamical system and `linode_dx`, `linode_dp` and `linode_dt` represent the derivatives of the vector field. They are encoded as follows

```
function y = linode(t, x, p)

x1 = x(1,:);
x2 = x(2,:);
k  = p(1,:);
th = p(2,:);

y(1,:) = x2;
y(2,:) = -x2-k.*x1+cos(t+th);

end


function J = linode_dx(t, x, p)

k = p(1,:);

J = zeros(2,2,numel(t));
J(1,2,:) = 1;
J(2,1,:) = -k;
J(2,2,:) = -1;

end


function J = linode_dp(t, x, p)

x1 = x(1,:);
th = p(2,:);
```

```
J = zeros(2,2,numel(t));
J(2,1,:) = -x1;
J(2,2,:) = -sin(t+th);

end


function J = linode_dt(t, x, p)

th = p(2,:);

J = zeros(2,numel(t));
J(2,:) = -sin(t+th);

end
```

We proceed to append boundary conditions to obtain periodic solutions of the dynamical system

```
>> [data, uidx] = coco_get_func_data(prob, 'coll', 'data', 'uidx');
>> maps = data.coll_seg.maps;
>> bc_funcs = {@linode_bc, @linode_bc_du};
>> prob = coco_add_func(prob, 'po', bc_funcs{:}, [], 'zero', 'uidx', ...
  uidx([maps.x0_idx; maps.x1_idx; maps.T0_idx; maps.T_idx]));
```

Here `linode_bc` and `linode_bc_du` are the encodings of the boundary condition and its Jacobian, respectively

```
function [data, y] = linode_bc(prob, data, u)

x0 = u(1:2);
x1 = u(3:4);
T0 = u(5);
T  = u(6);

y = [x1(1:2)-x0(1:2); T0; T-2*pi];

end


function [data, J] = linode_bc_du(prob, data, u)

J = [-1 0 1 0 0 0; 0 -1 0 1 0 0; 0 0 0 0 1 0; 0 0 0 0 0 1];

end
```

As shown in the command below, we further proceed to append a monitor function for the optimization objective

```
>> prob = coco_add_pars(prob, 'vel', uidx(maps.x0_idx(2)), 'v');
```

Now, the construction of continuation problem `prob` is completed and we can call `fmincon` as follows

```
>> options = optimoptions('fmincon','Display','iter');  % fmincon settings
```

```
>> options.SpecifyObjectiveGradient  = true;
>> options.SpecifyConstraintGradient = true;

>> u0 = prob.efunc.x0; % initial point
>> x  = fmincon(@(u) objfunc(u,prob,'vel'), u0,[],[],[],[],[],[],...
   @(u)nonlincons(u,prob),options);
```

Here `options` gives optimization options. Specifically

- `options = optimoptions('fmincon','Display','iter')` indicates the iteration history in searching minimum solutions will be displayed;
- `options.SpecifyObjectiveGradient = true` indicates the gradient provided by the wrapper function `objfunc` will be used if the optimization algorithm asks for gradient information; and
- `options.SpecifyConstraintGradient = true` indicates the gradient provided by the wrapper function `nonlincos` will be used if the optimization algorithm asks for gradient information

Once a candidate optimum is located, we may perform data processing and visualization using the wrappers in following way

```
>> [~,yy]   = opt_read_sol(x, prob, 'vel');
>> fprintf('Objetive at located optimum: obj=%d\n', yy);

% optimal state trajectory
>> coll_sol = opt_read_coll_sol(x, prob, '');
>> figure(1);
>> plot(coll_sol.tbp, coll_sol.xbp(:,1), 'r-'); hold on
>> plot(coll_sol.tbp, coll_sol.xbp(:,2), 'b--');
>> xlabel('$t$', 'interpreter', 'latex');
>> legend('$x_1(t)$', '$x_2(t)$', 'interpreter', 'latex');
>> set(gca, 'Fontsize', 14);

>> figure(2)
>> plot(coll_sol.xbp(:,1), coll_sol.xbp(:,2), 'ko-', 'MarkerSize', 8);
>> xlabel('$x_1$', 'interpreter', 'latex');
>> ylabel('$x_2$', 'interpreter', 'latex');
>> set(gca, 'Fontsize', 14);
```

Exercises

1. Compare the results obtained above with the ones from `coco/coll/examples/linode_optim`, where the latter demonstrates how to solve the first-order necessary conditions to locate local stationary points.
2. Modify the code of the `linode` example to also account for the inequality constraint on stiffness k and repeat the analysis in this section.

3. Consider the optimization along the solution manifold to a two-point boundary-value problem [REFs]. Specifically, consider the following objective functional

$$\frac{1}{10}(p_1^2+p_2^2+p_3^2)+\int_0^1 (x_1(t)-1)^2\mathrm{d}t$$

subject to constraints

$$\dot{x}_1=x_2,\ \dot{x}_2=-p_1\exp(x_1+p_2x_1^2+p_3x_1^4),\ x_1(0)=x_1(1)=0.$$

Implement your code following the code in this section and compare your results with the ones from `coco_fmincon/examples/dodel`.

---

6 Optimal control – `moon_lander`

Consider the optimal control of a soft lunar landing as follows [ref]: minimize

$$J=\int_0^{t_f} u\mathrm{d}t \tag{3}$$

subject to

$$\dot{h}=v,\ \dot{v}=-g+u,\quad h(0)=10,\ v(0)=-2,\ h(t_f)=0,\ v(t_f)=0 \tag{4}$$

and the control path constraint $0\leq u\leq 3$. Here $g=1.5$ is the gravity of acceleration and t_f is free.

We first construct a solution satisfying the differential equations and boundary conditions in (4). Parameter continuation will be used to achieve such a goal. Specifically, we assume $u(t)=kt$ and continue in t_f to arrive at $h(t_f)=0$, and then continue in k until $v(t_f)=0$. Such a solution will be used as the initial point for locating the minimum.

We proceed to implement the continuation problem using `coll` toolbox as follows

```
>> t0 = 0;
>> x0 = [10 -2];
>> prob = coco_prob();
>> prob = coco_set(prob, 'cont', 'h_max', 100);
>> prob = coco_set(prob, 'ode', 'autonomous', false);
>> prob = ode_isol2coll(prob, '', @lander0, t0, x0, 'k', 0);
>> data = coco_get_func_data(prob, 'coll', 'data'); % Extract function data
>> maps = data.coll_seg.maps;
>> prob = coco_add_pars(prob, 'pars', ...
       [maps.x0_idx; maps.x1_idx; maps.T0_idx; maps.T_idx], ...
       {'x1s' 'x2s' 'x1e' 'x2e' 'T0' 'T'});
```

Here `lander0` defines the vector field with given control input $u(t)=kt$. It is encoded as follows

```
function y = lander0(t,x,p)

y = [x(2,:); -1.5+t.*p(1,:)];

end
```

We then continue in t_f until $h(t_f) = 0$

```
>> cont_args = {1, {'x1e' 'T' 'x2e'}, {[0 20],[0, 100]}};
>> bd = coco(prob, 'coll1', [], cont_args{:});
```

We restart continuation from the one with $h(t_f) = 0$ and continue in k until $v(t_f) = 0$, as shown in the following commands.

```
>> labs = coco_bd_labs(bd, 'EP');
>> prob = coco_prob();
>> prob = coco_set(prob, 'cont', 'h_max', 100);
>> prob = ode_coll2coll(prob, '', 'coll1', max(labs));
>> data = coco_get_func_data(prob, 'coll', 'data');
>> maps = data.coll_seg.maps;
>> prob = coco_add_pars(prob, 'pars', ...
        [maps.x0_idx; maps.x1_idx; maps.T0_idx; maps.T_idx], ...
        {'x1s' 'x2s' 'x1e' 'x2e' 'T0' 'T'});
>> cont_args = {1, {'x2e' 'k' 'T'}, {[-6 0],[-100, 10]}};
>> bd = coco(prob, 'coll2', [], cont_args{:});
```

Next we construct the optimization problem using the obtained initial solution. We use `dae` toolbox to encode the state equations and path constraints. Specifically, we use `ddae_isol2coll` toolbox constructor to encode the state equation and `alg_dae_isol2seg` to encode the path constraints, as shown in the following commands.

```
>> labs = coco_bd_labs(bd, 'EP');
>> sol  = coll_read_solution('', 'coll2', max(labs));
>> t0   = sol.tbp;
>> x0   = sol.xbp;
>> y0   = sol.p*sol.tbp;
>> prob = coco_prob();
>> prob = coco_set(prob, 'ddaecoll', 'NTST', 20);
>> prob = coco_set(prob,'ddaecoll','Apoints','Gauss');
>> prob = ddaecoll_isol2seg(prob, '', @lander, t0, x0, y0, []);
>> prob = alg_dae_isol2seg(prob, 'g1', '', @g1func, 'inequality'); % -u<=0
>> prob = alg_dae_isol2seg(prob, 'g2', '', @g2func, 'inequality'); % u<=3
```

Here `lander` denotes the vector field with control input, `g1func` and `g2func` represents the two path constraints. They are encoded as follows

```
function y = lander(t,x,y,p)

y = [x(2,:); -1.5+y(1,:)];

end


function f = g1func(t,x,y,p)
```

```
f = -y;

end


function f = g2func(t,x,y,p)

f = y-3;

end
```

We proceed to append the boundary conditions as follows

```
>> [data, uidx] = coco_get_func_data(prob, 'ddaecoll', 'data', 'uidx');
>> bc_funcs = {@lander_bc, @lander_bc_du};
>> prob = coco_add_func(prob, 'bc', bc_funcs{:}, [], 'zero', 'uidx', ...
        uidx([data.x0_idx; data.x1_idx; data.T0_idx]));
```

Here `lander_bc` and `lander_bc_du` are the encodings of the boundary conditions and its Jacobian, respectively

```
function [data, y] = lander_bc(prob, data, u)

x0 = u(1:2);
x1 = u(3:4);
T0 = u(5);

y = [x0(1)-10; x0(2)+2; x1; T0];

end


function [data, J] = lander_bc_du(prob, data, u)

J = eye(5);

end
```

We finally append the objective functional to `prob`

```
>> prob = coco_add_func(prob, 'obj', @int_u, data, 'inactive', 'obj',...
        'uidx', uidx([data.ybp_idx; data.T_idx]));
```

Here `int_u` is the encoding of the integral of control input

```
function [data, y] = int_u(prob, data, u)

ybp = u(1:end-1);
T   = u(end);
ycn = data.Wda*ybp;
y   = 0.5*T*data.wts1*ycn/data.ddaecoll.NTST;

end
```

Now we can call `fmincon` to solve the optimization problem as follows

```
>> options = optimoptions('fmincon','Display','iter');
>> options.SpecifyObjectiveGradient  = true;
>> options.SpecifyConstraintGradient = true;

>> u0 = prob.efunc.x0;
>> fprintf('Optimization algorithm: interior-point (default)\n');
>> x = fmincon(@(u) objfunc(u,prob,'obj'), u0,[],[],[],[],[],[],...
        @(u) nonlincons(u,prob),options);
```

Once a candidate optimum is located, we can plot the time-history of states and control input using wrapper functions as follows

```
>> sol = opt_read_ddaecoll_sol(x, prob, '');
>> figure(1)
>> plot(sol.tbpd, sol.xbp(:,1), 'r-'); hold on
>> plot(sol.tbpd, sol.xbp(:,2), 'b--');
>> xlabel('$t$', 'interpreter', 'latex');
>> legend('$x_1(t)$', '$x_2(t)$', 'interpreter', 'latex');
>> set(gca, 'Fontsize', 14);

>> figure(2)
>> plot(sol.tbpa, sol.ybp, 'ro');
>> xlabel('$t$', 'interpreter', 'latex');
>> ylabel('$u(t)$', 'interpreter', 'latex');
>> set(gca, 'Fontsize', 14);
```

Exercises

1. `fmincon` does not require the initial point satisfying constraints. In this section, we performed continuation to satisfy the final state constraint. Consider the case without control and perform forward simulation to $t_f = 1$ and the final state constraint will be generally violated. Taking this solution as initial point and see whether `fmincon` returns convergent solution.

2. In this section, the optimal control is bang-bang. Change objective functional to $\int_0^{t_f} u^2 \mathrm{d}t$ and see how the optimal control changes.

---

7 Discussion

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`.

References

[1] F. Schilder and H. Dankowicz, "COCO." [Online]. Available: http://sourceforge.net/projects/cocotools

[2] H. Dankowicz and F. Schilder, *Recipes for continuation.* SIAM, 2013.

[3] MATLAB, "`fmincon`." [Online]. Available: https://www.mathworks.com/help/optim/ug/fmincon.html