User Manual for SPOC v1.2: Software for Parameter-Uncertainty Optimal Control Problems

Claire Walton, cwalton@soe.ucsc.edu

Abstract

SPOC utilizes direct methods in optimal control in conjunction with the nonlinear programming software SNOPT to solve optimal control problems with uncertain parameters.

Contents

1 Introduction

The optimization of control problems given uncertainty has been studied extensively in recent decades. Different goals and models of uncertainty have created large areas of research, including stochastic optimal control, robust control, and fuzzy control. SPOC finds numerical solutions to a class of problems created by constant but unknown parameter uncertainty. These problems have arisen in several recent applications, including the optimal search problem, [4], the optimization of batch processes under uncertainty [7], [6], and ensemble control [1].

Standard nonlinear control problems seek to find the control policy for a deterministic system which minimizes some cost functional evaluated over a time domain. Parameter uncertainty increases the difficulty of this standard control problem by inserting constant but probabilistic uncertainty into problem components such as the cost function or the state dynamics. The objective in these problems then becomes to minimize the expectation of the cost function of the resulting random variable. This adds an integration over probability space to the original integration over time.

Though analytic solutions to special cases of these problems are possible, analytic solutions in general are not an option. This has spurred the development of various methods for approximating numerical solutions to problems. SPOC uses consistent discretization methods, detailed in Section 2.5, to create a high-dimensional, but sparse, nonlinear programming problem (NLP). This NLP is then solved using SQP methods by the commercial solver SNOPT [2].

1.1 General Problem

SPOC addresses the following problem:

Problem P: For a parameter space $\Omega = [\omega_{0,1}, \omega_{f,1}] \times [\omega_{0,2}, \omega_{f,2}] \cdots \times [\omega_{0,N_\omega}, \omega_{f,N_\omega}] \subset \mathbb{R}^{N_\omega}$, determine the state-control pair, $t \to (x, u) \in \mathbb{R}^{N_x} \times \mathbb{R}^{N_u}$ that minimizes the cost function:

$$J[x(\cdot), u(\cdot)] = \int_\Omega G\left(\int_0^T r(x(t), u(t), t, \omega)dt\right)\phi(\omega)d\omega \tag{1}$$

subject to the dynamics:

$$\dot{x}(t) = f(x(t), u(t))$$

with initial condition $x(0) = x_0$, control constraint $K_1 \leq u(t) \leq K_2, \forall t \in [0, T]$. The function $\phi : \Omega \to \mathbb{R}$ is a continuous single-valued function, as are $G : \mathbb{R} \to \mathbb{R}$ and $r : \mathbb{R}^{N_x} \times \mathbb{R}^{N_u} \times [0, T] \times \Omega \to \mathbb{R}$.

This is a nonstandard optimal control problem due to the integration over parameter space and the additional evaluation of the function G. It can be viewed as a generalization of the standard Mayer-Bolza form.

1.2 Numerical Methods

Extensive knowledge of the numerical methods underpinning SPOC is not necessary for use. However, a general understanding is beneficial for assessing the possibilities and limitations of problem design when using this tool, especially in regards to parameter space. The method utilized by SPOC for solving Problem P is to first approximate the objective J by a new function J^M created from the discretization of parameter space. Given a set of quadrature nodes $\{\omega_i^M\}_{i=1}^M \in \Omega$ and weights $\{a_i^M\}_{i=1}^M \in \mathbb{R}$ the discretization of parameter space forms the following approximate problem:

Problem $\mathbf{P^M}$: Given probability density function $\phi : \Omega \to \mathbb{R}$, determine the control $u : [0, T] \to U \in \mathbb{R}^{n_u}$ that minimizes the sum

$$J^M = \sum_{i=1}^{M} a_i^M \left[F(x(T), \omega_i^M) + G\left(\int_0^T r(x(t), u(t), t, \omega_i^M) dt \right) \right] \tag{2}$$

subject to

$$\dot{x}(t) = f(x(t), u(t))$$

with initial condition $x(0) = x_0$ and control constraint $g(u(t)) \leq 0$, $\forall t \in [0, T]$.

This discretized problem is a standard control problem which can be solved using established methods. In [4] the consistency of this approach was shown for the class of problems described above when certain assumptions on compactness and continuity are satisfied. Specifically, it was shown that given any set of quadrature nodes $\{\omega_i^M\}_{i=1}^M \in \Omega$ and weights $\{a_i^M\}_{i=1}^M \in \mathbb{R}$ such that for any continuous function $h : \Omega \to \mathbb{R}$, the following convergence holds:

$$\int_\Omega h(\omega) d\omega = \lim_{M \to \infty} \sum_{i=1}^{M} h(\omega_i^M) a_i^M$$

and given the approximate problem $\mathbf{P^M}$, then under the assumptions in [4], accumulation points of the sequence of optimal solutions to the series $\mathbf{P^M}$ as $M \to \infty$ are optimal solutions of problem $\mathbf{P}$.

To implement problem $\mathbf{P^M}$, we utilize the method of direct pseudospectral collocation. As described in [5] and [3], in direct collocation, both the state and control are approximated over a discretized time grid $\pi^N = [t_0, \ldots, t_N]$ as:

$$x(t) \approx x^N(t) = \sum_{k=0}^{N} \bar{x}^{Nk} \phi_k(t), \quad u(t) \approx u^N(t) = \sum_{k=0}^{N} \bar{u}^{Nk} \phi_k(t)$$

where $\{\phi_k\}_{k=0}^N$ is set of interpolating functions, with orthogonality condition $\phi_k(t_j) = \delta_{kj}$. In local collocation methods, the degree of the interpolating functions is constant with respect to N, whereas in global collocation methods, like the pseudospectral method, the degree increases with N. The choice of interpolating functions also creates a differentiation scheme such that if $\bar{x}^N = [\bar{x}^{N0}, \ldots, \bar{x}^{NN}]$, differentiation can be approximated as the matrix multiplication $D^N \bar{x}^N$, where D^N is determined by the values of $\{\dot{\phi}_k\}$. This in coordination with a quadrature scheme with weights $\{b_k^N\}_{k=0}^N$ for the grid π^N creates the following problem:

Problem $\mathbf{P^{MN}}$: Determine the decision variables $\bar{x}^N = [\bar{x}^{N0}, \ldots, \bar{x}^{NN}]$ and $\bar{u}^N = [\bar{u}^{N0}, \ldots, \bar{u}^{NN}]$ that minimize the sum:

$$J^{MN} = \sum_{i=1}^{M} a_i^M \left[F(\bar{x}^{NN}, \omega_i^M) + G\left(\sum_{k=0}^{N} b_k^N r(\bar{x}^{Nk}, \bar{u}^{Nk}, t_k, \omega_i^M) \right) \right]$$

Subject to

$$D^N \bar{x}^N - f(\bar{x}^N, \bar{u}^N) = 0$$
$$\bar{x}^{N0} = x_0$$
$$g(\bar{u}^{Nk}) \leq 0 \text{ for all } k = 0, \ldots, N$$

This is a finite-dimensional nonlinear programming problem which is solved using the commercial solver SNOPT [2].

1.3 Example: A Linear Quadratic System

The following example looks at a simple system for which an analytical solution is available. Due to the simplicity of this problem, this analytical solution can in fact be arrived at both through the methods of standard control and through the methods described in 2.5. We consider the following linear quadratic system:

$$\text{Problem P1:} \quad \begin{cases} \text{Minimize} & J = \int\limits_{\Omega} \left(\int\limits_0^1 \sum\limits_{k=1}^{K} [(x_k - \omega_k)^2 + u_k^2] dt \right) \phi(\omega) d\omega \\ \text{Subject to} & \dot{x}(t) = u \\ & x(0) = 0 \end{cases} \tag{3}$$

where $x = (x_1, x_2, \ldots, x_K) \in \mathbb{R}^K$, $u = (u_1, u_2, \ldots, u_K) \in \mathbb{R}^K$. The parameters $\omega = (\omega_1, \omega_2, \ldots, \omega_K) \in \Omega$ are independent random variables with joint probability density function $\phi(\omega)$. This objective function can represent the K-dimensional distance from a stationary target with possible positions $(\omega_1, \omega_2, \ldots, \omega_K)$, and with a penalty function $\sum_{k=1}^K u_k^2$ to keep each control u_k within a reasonable range. Exchanging the order of integration allows

this to be formulated as a standard control problem:

$$\begin{cases} \text{Minimize} & J = \int_0^1 \left(\int_\Omega \sum_{k=1}^K [(x_k - \omega_k)^2 + u_k^2] \phi(\omega) d\omega \right) dt \\ \text{Subject to} & \dot{x}(t) = u \\ & x(0) = 0 \end{cases}$$

and independence of the random variables allows for explicit integration over Ω, reducing the objective function to:

$$J = \int_0^1 \sum_{k=1}^K \left(x_k^2 + u_k^2 - 2x_k E[\omega_k] + E[\omega_k^2] \right) dt.$$

This is a standard control objective, and thus we can find extremal solutions by solving the associated Hamiltonian boundary value problem. The Hamiltonian of this system is:

$$\mathbf{H}(x, u, \lambda) = \lambda^T u + \sum_{k=1}^K \left(x_k^2 + u_1^2 - 2x_k E[\omega_k] + E[\omega_k^2] \right)$$

and extremal solutions (x^*, u^*) satisfy the adjoint equations

$$\begin{array}{rcl} \frac{\partial \mathbf{H}}{\partial u}\Big|_{u=u^*} & = & 0 \\ \frac{\partial \mathbf{H}}{\partial x}\Big|_{x=x^*} & = & -\dot{\lambda}^*. \end{array}$$

This creates the dynamical system

$$\begin{bmatrix} \dot{x}_k^* \\ \dot{\lambda}_k^* \end{bmatrix} = \begin{bmatrix} 0 \\ 2E[\omega_k] \end{bmatrix} + \begin{bmatrix} 0 & -\frac{1}{2} \\ -2 & 0 \end{bmatrix} \begin{bmatrix} x_k^* \\ \lambda_k^* \end{bmatrix}$$

with boundary conditions from the Transversality condition:

$$\begin{array}{c} x_k^*(0) = 0 \\ \lambda_k^*(1) = 0 \end{array}$$

This linear system is solvable analytically, yielding the following expressions for the extremal trajectories, adjoints, and optimal objective value:

$$x_k^* = E[\omega_k] \left(1 - \tfrac{e^t + e^{2-t}}{1+e^2} \right), \quad u_k^* = -E[\omega_k] \left(\tfrac{e^t - e^{2-t}}{1+e^2} \right), \quad \lambda_k^* = 2E[\omega_k] \left(\tfrac{e^t - e^{2-t}}{1+e^2} \right),$$

$$J(x^*, u^*) = \sum_{k=1}^K \left(E[\omega_k]^2 \left(\frac{e^2 - 1}{e^2 + 1} - 1 \right) + E[(\omega_k)^2] \right).$$

Though we have solved this problem as a standard control problem, it can also be analyzed as an example of an optimal control problem with parameter uncertainty and solved using the numerical methods in Section 2.5. These numerical methods begin by discretizing the parameter space, creating an approximate problem, Problem P^M. Let $\{\omega_{k,i}^M\}_{i=1}^M$ be a set of nodes for Ω_k and let $\{a_{k,i}^M\}_{i=1}^M$ be a set of weights for a convergent quadrature scheme. Because ω_k are independent random variables, we can separate $\phi(\omega)$ into component probability distributions $\phi_k(\omega_k)$. Introducing the following useful constants

$$c_k^M = \sum_{i=1}^M \phi_k(\omega_{k,i}^M) a_{k,i}^M, \qquad c_{-k}^M = \prod_{j \neq k} c_j^M, \qquad c^M = \prod_j c_j^M$$

$$\tilde{E}[\omega_k] = \sum_{i=1}^M \omega_{k,i}^M \phi_k(\omega_{k,i}^M) a_{k,i}^M \qquad \tilde{E}[(\omega_k)^2] = \sum_{i=1}^M (\omega_{k,i}^M)^2 \phi_k(\omega_{k,i}^M) a_{k,i}^M$$

we can write Problem P^M as

$$\begin{cases} \text{Minimize} & J^M = \sum_{k=1}^K c_{-k}^M \sum_{i=1}^M \left[\int_0^1 ((x_k^2 - \omega_{k,i}^M)^2 + u_k^2) dt \right] \phi_k(\omega_{k,i}^M) a_{k,i}^M \\ \\ \text{Subject to} & \dot{x}(t) = u \\ & x(0) = 0. \end{cases}$$

We can now apply the methods of standard control to this approximate problem. The Hamiltonian for P^M is

$$\begin{aligned} \mathbf{H}(x, u, \lambda) &= \lambda^T u + \sum_{k=1}^K c_{-k}^M \sum_{i=1}^M \left[(x_k^2 - \omega_{k,i}^M)^2 + u_k^2 \right] \phi_k(\omega_{k,i}^M) a_{k,i}^M \\ &= \lambda^T u + \sum_{k=1}^K c_{-k}^M \left(c_k^M (x_k^2 + u_k^2) - 2 x_k \tilde{E}[\omega_k] + \tilde{E}[(\omega_k)^2] \right) \end{aligned}$$

and extremal solutions (x_M^*, u_M^*) satisfy the adjoint equations

$$\begin{aligned} \left. \frac{\partial \mathbf{H}}{\partial u_k} \right|_{u_k = u_{k,M}^*} &= \lambda_{k,M}^* + 2 c^M u_{k,M}^* &= 0 \\ \left. \frac{\partial \mathbf{H}}{\partial x_k} \right|_{x_k = x_{k,M}^*} &= 2 c^M u_{k,M}^* - 2 \tilde{E}[\omega_k] &= -\dot{\lambda}_{k,M}^*. \end{aligned}$$

This creates the linear system

$$\begin{bmatrix} \dot{x_{k,M}^*} \\ \dot{\lambda_{k,M}^*} \end{bmatrix} = \begin{bmatrix} 0 \\ 2\tilde{E}[\omega_k] \end{bmatrix} + \begin{bmatrix} 0 & -\frac{1}{2c^M} \\ -2c^M & 0 \end{bmatrix} \begin{bmatrix} x_{k,M}^* \\ \lambda_{k,M}^* \end{bmatrix}.$$

Solving the system yields the following expressions for the extremal trajectories, adjoints, and optimal objective value:

$$x^*_{k,M} = \frac{\tilde{E}[\omega_k]}{c^M}\left(1 - \frac{e^t + e^{2-t}}{1+e^2}\right), \quad u^*_{k,M} = -\frac{\tilde{E}[\omega_k]}{c^M}\left(\frac{e^t - e^{2-t}}{1+e^2}\right),$$

$$\lambda^*_{k,M} = 2\tilde{E}[\omega_k]\left(\frac{e^t - e^{2-t}}{1+e^2}\right), \quad J^M(x^*_M, u^*_M) = \sum_{k=1}^{K}\left(\tilde{E}[\omega_k]^2\left(\frac{e^2-1}{e^2+1} - 1\right) + \tilde{E}[(\omega_k)^2]\right)$$

Notice that c_k^M is the quadrature approximation of $\int_{\omega_k} \phi_k(\omega_k)d\omega_k$, $\tilde{E}[\omega_k]$ is the approximation of $E[\omega_k]$, and $\tilde{E}[(\omega_k)^2]$ is the approximation of $E[(\omega_k)^2]$:

$$\lim_{M\to\infty} c_k^M = 1 \qquad \lim_{M\to\infty} \tilde{E}[\omega_k] = E[\omega_k] \qquad \lim_{M\to\infty} \tilde{E}[(\omega_k)^2] = E[(\omega_k)^2].$$

Thus we can see that the answer we arrived at by using the method of discretization of parameter space converges to the true optimal answer which we derived by reorganizing the problem into a standard optimal control format. This convergence is based on the quantity of nodes used to approximate the parameter space and the efficacy of the chosen quadrature scheme. An example of the rate of this convergence for this problem is demonstrated in Figure 1.

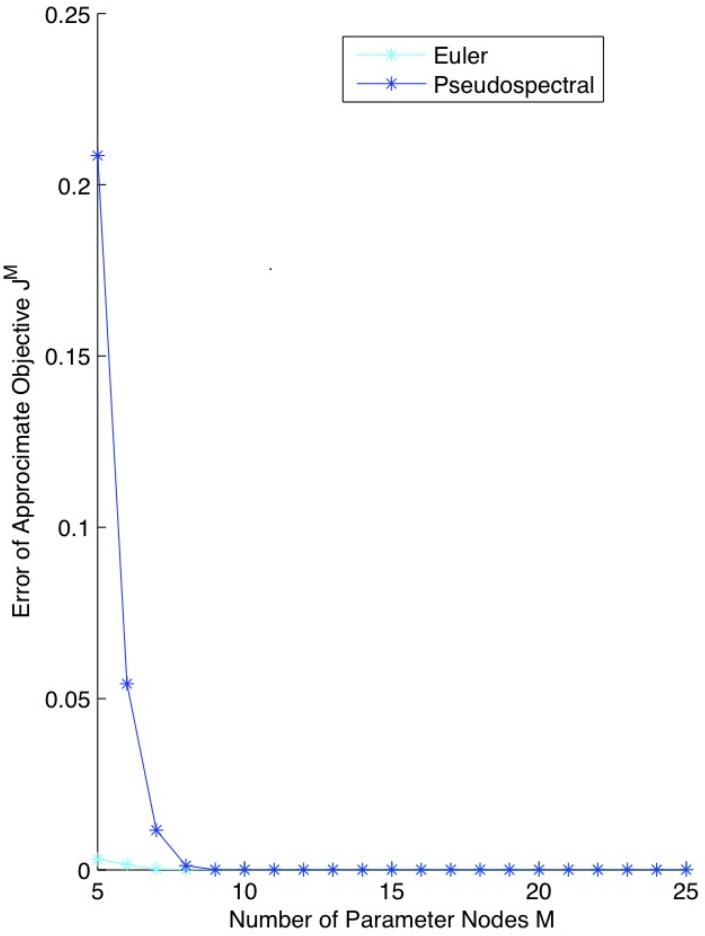

Figure 1: Convergence of analytical solutions for $J(x^*, u^*)$ and $J^M(x^*_M, u^*_M)$ using Euler's method vs Legendre pseudospectral. The dimension of the problem has been set at $K = 2$, and a Beta(10,10) distribution over the domain [0,1] has been given to each parameter.

2 Using SPOC

2.1 Overview

To solve a problem using SPOC, the user must write a set of required problem routines which define the basic components of the problem of interest. The basic problem components in need of definition are summarized below:

$$\min_{x,u} \int_{\Omega} G\left(\int_0^T r(x(t),u(t),t,\omega)dt\right)\phi(\omega)d\omega$$

subject to: $\dot{x}(t) = f(x(t),u(t)), \qquad x(0) = x_0$

$$K_1 \leq u(t) \leq K_2$$

- Inner function: $r(x(t),u(t),t,\omega)$
- Outer function: $G(z)$
- Dynamics function: $f(x(t),u(t))$
- Control bounds and initial conditions: x_0, K_1, K_2
- Other problem constants: Ω, T, $\phi(\omega)$

In addition to these components, an initial guess must also be supplied to seed the optimization algorithm. A 'problem', as run by SPOC, is a struct containing the names of each of the required user-written routines. For instance:

```
%=========================================================================%
% My_Problem is a struct with certain required entries
%=========================================================================%
% Example of required entries:
My_Problem.Problem_Definitions = 'My_Problem_Definitions';
My_Problem.Inner_Function = 'My_Inner_Function';
My_Problem.Outer_Function = 'My_Outer_Function';
My_Problem.State_Dynamics = 'My_State_Dynamics';
My_Problem.Initial_Guess = 'My_Initial_Guess';
My_Problem.Optimization_Bounds = 'My_Optimization_Bounds';
```

The required fields in the problem struct depend on whether the user has chosen to manually enter gradient information about the problem or to use automated finite difference approximations. Choosing to enter gradient information requires more user-written routines (and drastically speeds up optimization!). Setting the gradient options is discussed in section

2.3.1. The required files when using automated gradient approximations is the subject of section 2.3, and the requirements when using manual gradients is the subject of section 2.4.

Once this file structure has been established, SPOC is called with the problem struct, a choice of discretization methods for each dimension (such as Euler's method or Legendre pseudospectral), and a choice of discretization levels for each dimension (e.g. 25 nodes or 150 nodes). Methods and discretization levels are discussed in section 2.5. On completion, SPOC outputs the struct 'Results.' The fields included in this struct are discussed in 2.6.

```
%=========================================================================%
% Example run of a problem
%=========================================================================%

setpaths;
    %load all necessary file paths

% Example of required entries:
My_Problem.Problem_Definitions = 'My_Problem_Definitions';
My_Problem.Inner_Function = 'My_Inner_Function';
My_Problem.Outer_Function = 'My_Outer_Function';
My_Problem.State_Dynamics = 'My_State_Dynamics';
My_Problem.Initial_Guess = 'My_Initial_Guess';
My_Problem.Optimization_Bounds = 'My_Optimization_Bounds';

Discretization = [25,10,10];
Methods = [1,1,1];
    % These arrays are examples of method and discretization
    % settings in a problem with a two-dimension parameter
    % space. They are explained in the section 'Settings
    % Methods and Discretization Levels'.

Results = SPOC(My_Problem, Discretization, Methods);
```

2.2 Main Variable Structures

SPOC runs on a foundation of discretization which shapes some of its variable choices. The following variables are present in many of the required problem routines.

Discretization in the time domain (see 2.5) yields N time nodes at points $[t_1, \ldots, t_N]$. Variables containing these values are defined automatically after the choice of methods and discretization level is set:

$$\text{CONSTANTS.N} = N, \qquad \text{CONSTANTS.time\_nodes} = \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix}$$

Functions such as the Inner_Function and the State_Dynamics are defined as functions of the states and controls at these N points in time. The control variable in these functions is

a matrix with the columns holding the control (in order of some numbering) and the rows representing these controls at different time nodes. The state variable in these functions is in the same format.

$$\hat{u} = \begin{bmatrix} u_1(t_1) & u_2(t_1) & \cdots & u_{N_u}(t_1) \\ u_1(t_2) & u_2(t_2) & \cdots & u_{N_u}(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ u_1(t_N) & u_2(t_N) & \cdots & u_{N_u}(t_N) \end{bmatrix}, \quad \hat{x} = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \cdots & x_{N_x}(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_{N_x}(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_N) & x_2(t_N) & \cdots & x_{N_x}(t_N) \end{bmatrix} \tag{4}$$

Parameter space is also treated as a discrete space evaluated at a number of nodes. Some functions, such as Inner_Function, are functions of these values as well. The values of these nodes depend on choice of method and are automatically generated. They are stored in an $M \times N_\omega$ matrix labeled MESHED_DISCRETIZATION_VALUES (a global variable), where M is the number of different combination of parameter values which will have to be evaluated over the course of optimization (another amount automatically generated by choice of method/discretization level) and N_ω is the dimension of parameter space. Each row contains a single value for the vector $\omega = (\omega_1, \ldots, \omega_{N_\omega})$. Functions which may need to access these values, such as Inner_Function or Inner_Function_Gradient, have as one of their arguments the variable mesh_index. Automated optimization procedures will evaluate these functions down all rows of MESHED_DISCRETIZATION_VALUES, requesting the function value at row number mesh_index. For a value of mesh_index, the value of the k-th parameter is given by:

```
omega_k = MESHED_DISCRETIZATION_VALUES(mesh_index,k);
```

2.3 Required User Routines for Automatic Gradient

When automatic gradient calculations are chosen, the following fields require definitions in the problem struct:

- Problem_Definitions
- Inner_Function
- Outer_Function
- State_Dynamics
- Initial_Guess
- Optimization_Bounds

The requirements of the routines these entries specify are described in the sections below.

2.3.1 Problem_Definitions

The Problem_Definitions function stores the gradient setting choice and a set of required problem constants. These settings and values are stored in a global struct named CONSTANTS.

```
%=====================================================================%
% My_Problem_Definitions.m: Settings and the various physical
% constants utilized in subroutines are defined here and stored
% in the global struct CONSTANTS.
%=====================================================================%

function [] = My_Problem_Definitions()

global CONSTANTS
```

The choice of manual versus automatic gradient calculations is set in this file:

```
%----------Gradient Options-----------%
% A 'yes'commits to a manual gradient
% entry, sparsity pattern, and specification
%  of linear component.
CONSTANTS.User_Gradient = 'no';
    % CONSTANTS.User_Gradient = 'yes';
%-------------------------------------%
```

The file also requires specifications of the number of states and controls:

```
%=====================================================================%
% Specify features of state space
%=====================================================================%
% Example: one dubin's vehicle
CONSTANTS.Nx = 3;
    % Number of state variables
CONSTANTS.Nu = 1;
    % Number of controls variables
```

and the time interval the objective function is being calculated over:

```
%=========================================================================%
% Specify Time Range
%=========================================================================%
% Example: t=0 to t=1
CONSTANTS.Time.T0=0;
    %start time
CONSTANTS.Time.TF=1;
    %end time
```

Finally, the features of parameter space need to be encoded. This includes the dimension of the space, the start and end points of each parameter, and the choice of p.d.f. ($\phi(\omega)$). CONSTANTS.ParameterSpace.W0 and CONSTANTS.ParameterSpace.WF should be 1×CONSTANTS.ParameterSpace.Dimension arrays with the starting points and ending points respectively of each parameter's domain.

```
%=========================================================================%
% Specify features of parameter space
%=========================================================================%
% Example: Parameter space = [0,1]x[2,5]
CONSTANTS.ParameterSpace.Dimension = 2;
  % dimension of Parameter Space
CONSTANTS.ParameterSpace.W0 = [0,2];
    % start of each parameter:
CONSTANTS.ParameterSpace.WF = [1,5];
    % end of each parameter:

CONSTANTS.ParameterSpace.PFD_Choices = ...
    {'Independent','Beta_PDF','Beta_PDF'};
    % specify name of pdf routine
CONSTANTS.PDF.alpha = [2,2];
CONSTANTS.PDF.beta = [3,3];
    % and any necessary pdf parameters
```

PDF_Choices specifies whether the parameters have independent p.d.f's or are dependent with one joint p.d.f. If the parameters are independent, a filename must be specified for each dimension which points to a pdf function. Additionally, any necessary feature of the p.d.f (such as standard deviation) must be defined and given a value for each parameter. Currently supported p.d.f.'s are in /SPOC_Kernel/Supported_PDFs. Adding a p.d.f. merely requires writing a p.d.f. function in a certain format, and is described in section 2.7.1. If 'Dependent' is chosen, only one joint pdf file needs to be specified:

```
CONSTANTS.ParameterSpace.PFD_Choices = ...
                    {'Dependent','Example_Joint_PDF'};
```

See section 2.7.1 for details on the format of this file.

In summary, The Problem_Definitions function must define the following constants:

```
%=========================================================================%
%---------------NECESSARY CONSTANTS FOR ALL PROBLEMS-------------%
%=========================================================================%
CONSTANTS.ParameterSpace.Dimension
CONSTANTS.ParameterSpace.W0
CONSTANTS.ParameterSpace.WF
CONSTANTS.ParameterSpace.PFD_Choices
    % If chosen pdf has parameters, their definition is also
    % required, for each parameter.
    % E.g. CONSTANTS.PDF.alpha(i) = ...
CONSTANTS.Nx
CONSTANTS.Nu
CONSTANTS.Time.T0
CONSTANTS.Time.TF
CONSTANTS.User_Gradient
```

This function can additionally be used to define any other physical constants relevant to the problem.

2.3.2 Outer_Function

The Outer_Function creates the function known as $G(\cdot)$.

```
%=========================================================================%
% The outer function is a single-valued function of a single
% variable.
%=========================================================================%
function [y] = My_Outer_Function(z)

% In this example, the outer function is just identity.
y=z;
```

2.3.3 Inner_Function

The Inner_Function creates the function known as $r(\cdot)$. The arguments of this function are the control matrix, $\hat{u}$, the state matrix, $\hat{x}$, and the mesh_index, as defined in section 2.2.

The output is a column vector of function values at all N points in time. That is,

$$\begin{bmatrix} r(x(t_1), u(t_1), t_1, \omega) \\ \vdots \\ r(x(t_N), u(t_N), t_N, \omega) \end{bmatrix} = Inner\_Function(\hat{u}, \hat{x}, \omega\_index)$$

```
%=====================================================================%
% The inner function is a function of the controls, states,
% and the index of the parameter values.
% Returns column vector of function values at all time nodes.
%=====================================================================%
function [z]=My_Inner_Function(controls, states, mesh_index)
% Example: linear quadratic cost function from section 1.3
global CONSTANTS...
       MESHED_DISCRETIZATION_VALUES ...

z = zeros(CONSTANTS.N,1);

K = CONSTANTS.ParameterSpace.Dimension;
%time_nodes = CONSTANTS.time_nodes;

for k=1:K
    omega_k = MESHED_DISCRETIZATION_VALUES(mesh_index,k);
    z = z + (states(:,k)-omega_k).^2 + controls(:,k).^2;
end
```

2.3.4 State_Dynamics

In this problem, each state has a differential constraint. For N_x states, this creates N_x constraints:

$$\dot{x}_k(t) = f_k(x(t), u(t), t), \qquad k = 1, \ldots, N_x$$

The State_Dynamics function outputs a matrix of these N_x functions at all at all N points in time. I.e., for given values of $x(t) = [x_1(t), \ldots, x_{N_x}(t)]$, and $u(t) = [u_1(t), \ldots, u_{N_u}(t)]$ at time nodes $[t_1, \ldots, t_N]$ the function outputs the matrix:

$$\begin{bmatrix} f_1(x(t_1), u(t_1), t_1) & f_2(x(t_1), u(t_1), t_1) & \cdots & f_{N_x}(x(t_1), u(t_1), t_1) \\ f_1(x(t_2), u(t_2), t_2) & f_2(x(t_2), u(t_2), t_2) & \cdots & f_{N_x}(x(t_2), u(t_2), t_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x(t_N), u(t_N), t_N) & f_2(x(t_N), u(t_N), t_N) & \cdots & f_{N_x}(x(t_N), u(t_N), t_N) \end{bmatrix}$$

The arguments of the State_Dynamics function are the control matrix, $\hat{u}$, and the state matrix, $\hat{x}$, as defined in section 2.2.

```
%================================================================%
% This function defines the dynamics function for each state's
% differential constraint.
% Returns matrix of all function values at all time nodes.
% The CONSTANTS.Nx constraints are defined in order of state
% indices. Each function is defined as a column of function
% values at each point in time, which fills a column on the
% 'dynamics' variable. I.e. the dynamics variable is a
% CONSTANTS.N by CONSTANTS.Nx matrix.
%================================================================%
function [dynamics] = My_State_Dynamics(controls, states)
% Example: linear quadratic dynamics from section 1.3
global CONSTANTS
dynamics = zeros(CONSTANTS.N, CONSTANTS.Nx);

% In this example the dynamics function for x_k is equal to u_k.
for k = 1:CONSTANTS.Nx
    dynamics(:,k) = controls(:,k);
end
```

2.3.5 Initial_Guess

The Initial_Guess function constructs and initial guess for the optimal values of the control matrix, $\hat{u}$, and the state matrix, $\hat{x}$, as defined in section 2.2. Output is a control guess, and a state guess, evaluated at all N time nodes, in the format of the matrices of section 2.2.

```
function [uguess, xguess] = My_Initial_Guess()

global CONSTANTS
time_nodes = CONSTANTS.time_nodes;
    % the time values the guesses provide values for
N = CONSTANTS.N;
    % Total number of time points.
Nx =  CONSTANTS.Nx;
    % number of state variables.
Nu = CONSTANTS.Nu;
    % number of control variables.

% This example just guesses zero values.
uguess = zeros(N,Nu);
xguess = zeros(N,Nx);
    %Each state needs a guessed value at each time node.
```

2.3.6 Optimization_Bounds

The Optimization_Bounds function sets the initial state values, bounds on the possible control and state values, and a few options related to bounds. These bounds and settings are returned as the output of the function.

```
%=========================================================================%
% Optimization_Bounds returns eight required optimization settings
%=========================================================================%
function [Objective_lower, Objective_upper, epsilon,...
          Max_u, Min_u, ...
          Max_x,Min_x,x_0] = My_Optimization_Bounds()
```

The initial values of the state variables are loaded into the $1 \times$CONSTANTS.Nx vector x_0.

```
%=========================================================================%
%                Initial Conditions
%=========================================================================%
% Create an array of initial values for each state.
% In this example x_k(0) = 0
x_0 = zeros(1,CONSTANTS.Nx);
%=========================================================================%
```

Upper and lower bounds are placed on controls and states. A bound is required for each control and each state. If control or state is unconstrained, put an unreasonably large bound in this place.

```
%=========================================================================%
%            State and Control Bounds
%=========================================================================%
% Set bounds for u:
Max_u = 1000*ones(1,CONSTANTS.Nu);
Min_u = -1000*ones(1,CONSTANTS.Nu);
% If x or u is unconstrained we simulate that by puttting a
% large finite bound on its magnitude
Max_x = 1000*ones(1,CONSTANTS.Nx);
Min_x = -1000*ones(1,CONSTANTS.Nx);
%=========================================================================%
```

Additionally, bounds are inputed for the objective value. If an approximate (or precise) range of possible objective values are known, inputting tight bounds here can speed up optimization.

```
%=================================================================%
%               Objective Bounds
%=================================================================%
Objective_lower = 0;
Objective_upper = 100;
%=================================================================%
```

Finally, possible slackness in the dynamics constraints is considered:

```
%=================================================================%
%               Dynamic Bound Slackness
%=================================================================%
epsilon=0;
% This creates bounds of with possible slackness of -epsilon to
% epsilon. epsilon = 0 enforces equality.
%=================================================================%
```

In summary, the Optimization_Bounds function must define the following constants: x_0, Max_u, Min_u, Max_x, Min_x, Objective_lower, Objective_upper, epsilon.

2.4 Required User Routines for User Entered Gradient

When user entered gradients are chosen (see 2.3.1), additional function definitions are required and some of the requirements of the functions are modified. It's quite a bit more effort, but well worth it for the benefits! The following required fields remain unchanged from section 2.3:

- Problem_Definitions
- Inner_Function
- Outer_Function
- Initial_Guess
- Optimization_Bounds

Notice that this excludes the previously required 'State_Dynamics' routine. This routine has now been split into several functions. Additional required problem struct fields are:

- Inner_Function_Gradient
- Inner_Function_Sparsity
- Outer_Function_Gradient

- Nonlinear_Dynamics
- Nonlinear_Dynamics_Sparsity
- Linear_Dynamics

2.4.1 Inner_Function_Gradient

The Inner_Function_Gradient returns the values of the partial derivatives, w.r.t. controls and states, of the inner objective function, at all time nodes. The arguments of this function are the control matrix, $\hat{u}$, the state matrix, $\hat{x}$, and the mesh_index, as defined in section 2.2. The output is the following two matrices, which hold the gradient values w.r.t. to the controls and the gradient values w.r.t. to the states for given values of $\hat{u}$ and $\hat{x}$:

$$gradient\_controls = \begin{bmatrix} \frac{\partial r}{\partial u_1}\Big|_{t_1,\omega} & \frac{\partial r}{\partial u_2}\Big|_{t_1,\omega} & \cdots & \frac{\partial r}{\partial u_{N_u}}\Big|_{t_1,\omega} \\ \frac{\partial r}{\partial u_1}\Big|_{t_2,\omega} & \frac{\partial r}{\partial u_2}\Big|_{t_2,\omega} & \cdots & \frac{\partial r}{\partial u_{N_u}}\Big|_{t_2,\omega} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial r}{\partial u_1}\Big|_{t_N,\omega} & \frac{\partial r}{\partial u_2}\Big|_{t_N,\omega} & \cdots & \frac{\partial r}{\partial u_{N_u}}\Big|_{t_N,\omega} \end{bmatrix}$$

$$gradient\_states = \begin{bmatrix} \frac{\partial r}{\partial x_1}\Big|_{t_1,\omega} & \frac{\partial r}{\partial x_2}\Big|_{t_1,\omega} & \cdots & \frac{\partial r}{\partial x_{N_x}}\Big|_{t_1,\omega} \\ \frac{\partial r}{\partial x_1}\Big|_{t_2,\omega} & \frac{\partial r}{\partial x_2}\Big|_{t_2,\omega} & \cdots & \frac{\partial r}{\partial x_{N_x}}\Big|_{t_2,\omega} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial r}{\partial x_1}\Big|_{t_N,\omega} & \frac{\partial r}{\partial x_2}\Big|_{t_N,\omega} & \cdots & \frac{\partial r}{\partial x_{N_x}}\Big|_{t_N,\omega} \end{bmatrix}$$

where $\omega = MESHED\_DISCRETIZATION\_VALUES(mesh\_index, :)$.

```
%=================================================================%
% The inner function is a function of the controls, states,
% and the index of the parameter values.
% Returns the gradient values of the inner function w.r.t
% controls and states, at each time node.
% If the inner function is independent of a control or state,
% the value down the column is zero. The columns with nonzero
% entries should correspond to the dependencies encoded in
% Inner_Function_Sparsity().
%=================================================================%
function [gradient_controls, gradient_states] =...
          My_Inner_Function_Gradient(controls, states, mesh_index)
% Example: gradients for linear quadratic problem from section 1.3
global CONSTANTS...
       MESHED_DISCRETIZATION_VALUES ...

gradient_controls = zeros(CONSTANTS.N, CONSTANTS.Nu);
gradient_states = zeros(CONSTANTS.N, CONSTANTS.Nx);

% the partial derivative w.r.t. each u_k is 2u_k:
for k=1:CONSTANTS.Nu
    gradient_controls(:,k) = 2*controls(:,k);
end

% the partial derivative w.r.t. each x_k is 2(x_k-omega_k)
for k=1:CONSTANTS.Nx
    omega_k = MESHED_DISCRETIZATION_VALUES(mesh_index,k);
    gradient_states(:,k) = 2*(states(:,k)-omega_k);
end
```

2.4.2 Inner_Function_Sparsity

In order to fully take advantage of the problem structure, SPOC uses variable dependencies to discard inactive gradient elements. The function Inner_Function_Sparsity returns two indicator vectors which specify which controls and states (respectively) the inner objective function $r(\cdot)$ depends on.

```
%=================================================================%
% This function returns the indicator vectors which specify which
% controls and states the inner function is a function of.
%=================================================================%
function [indicator_vector_controls, indicator_vector_states] =...
          My_Inner_Function_Sparsity()
```

The vector indicator_vector_controls is a row vector with N_u entries and indicator_vector_states is a row vector with N_x entries. These entries are 0's or 1's where indicator_vector_controls(i)=1 specifies that the inner function is a function of the i-th control

(resp., state) and indicator_vector_controls(i)=0 specifies that the inner function is independent of the i-th control (resp., state).

```
% In this example, the inner function is a function of all
% controls and states:
global CONSTANTS
indicator_vector_controls = ones(1, CONSTANTS.Nu);
indicator_vector_states = ones(1, CONSTANTS.Nx);
```

2.4.3 Outer_Function_Gradient

The Outer_Function_Gradient returns the derivative of the outer function.

```
%=========================================================================%
% Returns the derivative of the outer function.
%=========================================================================%
function [y] = My_Outer_Function_Gradient(z)
% In this example, the outer function is the identity function,
% so the so the derivative is one.
y=1;
```

2.4.4 Separating Linear and Nonlinear Dynamics

[1] Since the gradient of linear functions is a constant which only needs to be computed once per problem, separating the linear portions of functions from the nonlinear portions can yield significant computational gains. When user entered gradients are chosen, the separation of linear and nonlinear portions of the dynamics functions is required. For each dynamics constraint this creates the decomposition:

$$\dot{x}_k(t) = f_k(x(t), u(t), t) = f_k^{linear}(x(t), u(t), t) + f_k^{nonlinear}(x(t), u(t), t)$$

In SPOC, this decomposition has the additional constraint that the nonlinear and linear components cannot overlap in variable dependencies. That is, variable is not allowed to appear in both components–if a variable has a nonlinear contribution, then all other contributions of that variable are considered 'nonlinear'.

Example:

$$f_k(x_1, x_2, u_1, u_2) = x_1 + x_2 + x_1^2 + u_1 + u_2 + u_1^2$$

$$\Longrightarrow \begin{cases} f_k^{linear}(x_1, x_2, u_1, u_2) = x_2 + u_2, \\ f_k^{nonlinear}(x_1, x_2, u_1, u_2) = x_1 + x_1^2 + u_1 + u_1^2 \end{cases}$$

[1] Before reading the following sections, please acquaint yourself with section 2.3.4.

The two components of this decomposition are required inputs, as are their gradients and variable dependencies. The format of these inputs is described in the next few sections.

2.4.5 Nonlinear_Dynamics

The Nonlinear_Dynamics function outputs a matrix of the nonlinear component of the N_x dynamics functions at all, at all N points in time. In addition to this, the function outputs two cell arrays, gradient_controls and gradient_states, which contain the gradients with respect to controls (resp. states) for each of the N_x dynamics constraints.

```
%=========================================================================%
% Returns nonlinear component of dynamics function and gradient
% information for each constraint
%=========================================================================%
function [nonlinear_dynamics, gradient_controls,gradient_states]...
                        = My_Nonlinear_Dynamics(controls, states)

global CONSTANTS

nonlinear_dynamics = zeros(CONSTANTS.N, CONSTANTS.Nx);
gradient_states = cell(1,CONSTANTS.Nx);
gradient_controls = cell(1,CONSTANTS.Nx);
```

The nonlinear_dynamics matrix in the same format as the state_dynamics matrix in section 2.3.4. That is, for given values of $x(t) = [x_1(t), \ldots, x_{N_x}(t)]$, and $u(t) = [u_1(t), \ldots, u_{N_u}(t)]$ at time nodes $[t_1, \ldots, t_N]$ the nonlinear_dynamics matrix is:

$$\begin{bmatrix} f_1^{nonlinear}(x(t_1), u(t_1), t_1) & f_2^{nonlinear}(x(t_1), u(t_1), t_1) & \cdots & f_{N_x}^{nonlinear}(x(t_1), u(t_1), t_1) \\ f_1^{nonlinear}(x(t_2), u(t_2), t_2) & f_2^{nonlinear}(x(t_2), u(t_2), t_2) & \cdots & f_{N_x}^{nonlinear}(x(t_2), u(t_2), t_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1^{nonlinear}(x(t_N), u(t_N), t_N) & f_2^{nonlinear}(x(t_N), u(t_N), t_N) & \cdots & f_{N_x}^{nonlinear}(x(t_N), u(t_N), t_N) \end{bmatrix}$$

In addition to this, each nonlinear component has its gradient elements w.r.t. states and controls stored. Each matrix filled an entry of the cell arrays gradient_controls and gradient_states (resp.)

$$gradient\_controls\{k\} = \begin{bmatrix} \left.\frac{\partial f_k^{nonlinear}}{\partial u_1}\right|_{t_1} & \left.\frac{\partial f_k^{nonlinear}}{\partial u_2}\right|_{t_1} & \cdots & \left.\frac{\partial f_k^{nonlinear}}{\partial u_{N_u}}\right|_{t_1} \\ \left.\frac{\partial f_k^{nonlinear}}{\partial u_1}\right|_{t_2} & \left.\frac{\partial f_k^{nonlinear}}{\partial u_2}\right|_{t_2} & \cdots & \left.\frac{\partial f_k^{nonlinear}}{\partial u_{N_u}}\right|_{t_2} \\ \vdots & \vdots & \ddots & \vdots \\ \left.\frac{\partial f_k^{nonlinear}}{\partial u_1}\right|_{t_N} & \left.\frac{\partial f_k^{nonlinear}}{\partial u_2}\right|_{t_N} & \cdots & \left.\frac{\partial f_k^{nonlinear}}{\partial u_{N_u}}\right|_{t_N} \end{bmatrix}$$

$$gradient\_states\{k\} = \begin{bmatrix} \left.\frac{\partial f_k^{nonlinear}}{\partial x_1}\right|_{t_1} & \left.\frac{\partial f_k^{nonlinear}}{\partial x_2}\right|_{t_1} & \cdots & \left.\frac{\partial f_k^{nonlinear}}{\partial x_{N_x}}\right|_{t_1} \\ \left.\frac{\partial f_k^{nonlinear}}{\partial x_1}\right|_{t_2} & \left.\frac{\partial f_k^{nonlinear}}{\partial x_2}\right|_{t_2} & \cdots & \left.\frac{\partial f_k^{nonlinear}}{\partial x_{N_x}}\right|_{t_2} \\ \vdots & \vdots & \ddots & \vdots \\ \left.\frac{\partial f_k^{nonlinear}}{\partial x_1}\right|_{t_N} & \left.\frac{\partial f_k^{nonlinear}}{\partial x_2}\right|_{t_N} & \cdots & \left.\frac{\partial f_k^{nonlinear}}{\partial x_{N_x}}\right|_{t_N} \end{bmatrix}$$

```
%=========================================================================%
% Returns nonlinear component of dynamics function and gradient
% information for each constraint
%=========================================================================%
function [nonlinear_dynamics, gradient_controls,gradient_states]...
                        = My_Nonlinear_Dynamics(controls, states)

global CONSTANTS

nonlinear_dynamics = zeros(CONSTANTS.N, CONSTANTS.Nx);
gradient_states = cell(1,CONSTANTS.Nx);
gradient_controls = cell(1,CONSTANTS.Nx);
```

2.4.6 Nonlinear_Dynamics_Sparsity

The function Nonlinear_Dynamics_Sparsity returns two cell arrays with indicator vectors which specify which controls and states (respectively) each nonlinear dynamics function $f_k^{nonlinear}$ depends on.

```
%=========================================================================%
% This function returns the indicator vectors which specify which
% controls and states the nonlinear dynamics functions are
% functions of.
%=========================================================================%
function [controls_indicator_vectors, states_indicator_vectors] ...
                            = My_Nonlinear_Dynamics_Sparsity()
global CONSTANTS

controls_indicator_vectors =  cell(1,CONSTANTS.Nx);
states_indicator_vectors  =  cell(1,CONSTANTS.Nx);
```

Each entry in control_indicator_vectors (resp. states_indicator_vectors) is a row vector with N_u entries (resp. N_x entries). These entries are 0's or 1's where entry(i)=1 specifies that the k-th nonlinear dynamics function is a function of the i-th control (resp., state) and entry(i)=0 specifies that the k-th nonlinear dynamics function is independent of the i-th control (resp., state).

```
%=====================================================================%
% This function returns the indicator vectors which specify which
% controls and states the nonlinear dynamics functions are
% functions of.
%=====================================================================%
function [controls_indicator_vectors, states_indicator_vectors] ...
                                = My_Nonlinear_Dynamics_Sparsity()
% In this example, the dynamics are fully linear, so the
% nonlinear each dynamics function is independent from all
% controls and states:
global CONSTANTS

controls_indicator_vectors =  cell(1,CONSTANTS.Nx);
states_indicator_vectors   =  cell(1,CONSTANTS.Nx);

for k = 1:CONSTANTS.Nx
    controls_indicator_vectors{k} = zeros(1,CONSTANTS.Nu);
    states_indicator_vectors{k} = zeros(1,CONSTANTS.Nx);
end
```

2.4.7 Linear_Dynamics

Given the naming structures which have preceded it, the name Linear_Dynamics is slightly misleading, in that this routine does not actually return the linear dynamics. This is because the linear dynamics are uniquely determined by their gradient and so their entry is superfluous. The Linear_Dynamics routine is instead in charge of returning the gradient and sparsity information for the linear components of the state dynamics.

```
%=====================================================================%
% Returns gradient and sparsity information for each linear
% component of the dynamics functions.
%=====================================================================%
function [gradient_controls, gradient_states,...
          controls_indicator_vectors, states_indicator_vectors] ...
                                      = My_Linear_Dynamics()
```

The function outputs two cell arrays, gradient_controls and gradient_states, which contain the gradients with respect to controls (resp. states) for each of the N_x dynamics constraints. These two outputs are identical in format to those described in section 2.4.5, except they

are calculated in relation to the linear components:

$$gradient\_controls\{k\} = \begin{bmatrix} \left.\frac{\partial f_k^{linear}}{\partial u_1}\right|_{t_1} & \left.\frac{\partial f_k^{linear}}{\partial u_2}\right|_{t_1} & \cdots & \left.\frac{\partial f_k^{linear}}{\partial u_{N_u}}\right|_{t_1} \\ \left.\frac{\partial f_k^{linear}}{\partial u_1}\right|_{t_2} & \left.\frac{\partial f_k^{linear}}{\partial u_2}\right|_{t_2} & \cdots & \left.\frac{\partial f_k^{linear}}{\partial u_{N_u}}\right|_{t_2} \\ \vdots & \vdots & \ddots & \vdots \\ \left.\frac{\partial f_k^{linear}}{\partial u_1}\right|_{t_N} & \left.\frac{\partial f_k^{linear}}{\partial u_2}\right|_{t_N} & \cdots & \left.\frac{\partial f_k^{linear}}{\partial u_{N_u}}\right|_{t_N} \end{bmatrix}$$

$$gradient\_states\{k\} = \begin{bmatrix} \left.\frac{\partial f_k^{linear}}{\partial x_1}\right|_{t_1} & \left.\frac{\partial f_k^{linear}}{\partial x_2}\right|_{t_1} & \cdots & \left.\frac{\partial f_k^{linear}}{\partial x_{N_x}}\right|_{t_1} \\ \left.\frac{\partial f_k^{nlinear}}{\partial x_1}\right|_{t_2} & \left.\frac{\partial f_k^{linear}}{\partial x_2}\right|_{t_2} & \cdots & \left.\frac{\partial f_k^{linear}}{\partial x_{N_x}}\right|_{t_2} \\ \vdots & \vdots & \ddots & \vdots \\ \left.\frac{\partial f_k^{linear}}{\partial x_1}\right|_{t_N} & \left.\frac{\partial f_k^{linear}}{\partial x_2}\right|_{t_N} & \cdots & \left.\frac{\partial f_k^{linear}}{\partial x_{N_x}}\right|_{t_N} \end{bmatrix}$$

In addition to these, the function returns two cell arrays with indicator vectors which specify which controls and states (respectively) each nonlinear dynamics function $f_k^{nonlinear}$ depends on. These two outputs are identical in format to those described in section 2.4.6, except they are calculated in relation to the linear components.

```
%=========================================================================%
% Returns gradient and sparsity information for each linear
% component of the dynamics functions.
%=========================================================================%
function [gradient_controls, gradient_states,...
          controls_indicator_vectors, states_indicator_vectors] ...
                                        = My_Linear_Dynamics()
% Example: linear quadratic dynamics from section 1.3
global CONSTANTS

gradient_controls = cell(1,CONSTANTS.Nx);
gradient_states = cell(1,CONSTANTS.Nx);
controls_indicator_vectors = cell(1,CONSTANTS.Nx);
states_indicator_vectors = cell(1,CONSTANTS.Nx);

for k = 1:CONSTANTS.Nx
    % The dynamics for x_k are given by u_k. The gradient is thus
    % 1 w.r.t. u_k (at all time points) and zero everywhere else.
    gradient_controls{k} = zeros(CONSTANTS.N, CONSTANTS.Nu);
    gradient_controls{k}(:,k) = 1;

    % The k-th dynamic only depends on the k-th control:
    controls_indicator_vectors{k} = zeros(1,CONSTANTS.Nu);
    controls_indicator_vectors{k}(k) = 1;

    % None of the dynamics depend on any state values:
    gradient_states{k} = zeros(CONSTANTS.N,CONSTANTS.Nx);
    states_indicator_vectors{k} = zeros(1, CONSTANTS.Nx);
end
```

2.5 Setting Methods and Discretization Levels

SPOC uses methods which discretize time and each dimension of parameter space. The currently provided methods are Euler (Euler's method with forward integration) and Legendre pseudospectral (Legendre-Gauss-Lobatto (LGL) points and Gaussian quadrature). Distinct methods can be chosen for the time domain and for each dimension of parameter space. These choices are recorded as entries in the Methods array (see 2.1). The Methods array has 1+CONSTANTS.ParamaterSpace.Dimension entries–one for the time domain, followed by choices for each parameter space dimension. Possible entries are $Methods(i) = 0$ for Euler's method, and $Methods(i) = 1$ for Legendre pseudospectral.

Each discretization method is computed in terms of number of nodes. More nodes creates more accuracy but costs more time. The Discretization array specifies the number nodes used in each dimension. The Discretization array is the same size as the Methods array. The number of nodes used in the time domain is the first entry, followed by the number of nodes used in each dimension of parameter space.

2.6 Outputs

The Results struct contains the following fields:

- Results.controls: extremal controls, in the format of 2.2
- Results.states: extremal states, in the format of 2.2
- Results.objective_value: the value of the cost function evaluated at the extremal solutions
- Results.time_nodes: the nodes of time the answer is returned at
- Results.build_time: the time it took to do any calculations which preceded running the optimization routines
- Results.run_time: the time it took to run the optimization routines

Additionally, it returns two fields which are useful to users with knowledge of SNOPT:

- Results.INFO: SNOPT exit code
- Results.F: vector containing value of cost function and all dynamics constraints. These constraints are ideally zero-valued or nearly zero if slackness has been included in Optimization_Bounds.

2.7 Miscellaneous Features

2.7.1 Defining Additional P.D.F.'s

Additional p.d.f.'s can be defined by adding functions of appropriate format to the folder SPOC_Kernel/Supported_PDFs. P.d.f. to be used on independent parameters have the following format:

[outputted_pdf_values] = pdf_function(discretization_values, parameter_index)

For parameter variable ω_k, discretization_values is a column vector of parameter values the p.d.f. is being evaluated at. The parameter_index is the index which specifies which parameter variable is being evaluated (i.e. k). This is used to access the information encoded in Problem_Definitions. For example,

w0 = CONSTANTS.ParameterSpace.W0(parameter_index)

wf = CONSTANTS.ParameterSpace.WF(parameter_index)

gives the endpoints of the parameter space, as set in Problem_Definitions (2.3.1), which is necessary information for most distributions. The parameter_index is also used to access any other features of the p.d.f. which may vary by parameter variable and were established in

Problem_Definitions (for instance, the alpha and beta values for distinct beta distributions). The pdf_function function outputs a column vector, the same size as discretization_values, which contains the value of the p.d.f. at each point in discretization_values.

Joint p.d.f. functions have similar demands, except they are evaluated along a matrix of values, each row of which is a set of joint parameter values which require an outputted p.d.f. value.

[outputted_pdf_values] = pdf_function(meshed_discretization_values)

The input, meshed_discretization_values, is in the format described in 2.2. The output is a column vector with the same number of rows as meshed_discretization_values.

2.7.2 SNOPT settings

For those with knowledge of SNOPT, setting modifications can be made in:

SPOC_Kernel/Optimization_Settings

Settings related to iteration limits and tolerances can be modified, however those relating to structural decisions such as manual gradient entry vs not or the location of the cost function vs dynamic constraints.

References

[1] Aaron Trent Becker. Ensemble control of robotic systems. *DISSERTATION*, 2012.

[2] Philip E. Gill, Walter Murray, and Michael A. Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005.

[3] Qi Gong, Wei Kang, Nazareth S. Bedrossian, Fariba Fahroo, Sekhavat, and Kevin Bollino. Pseudospectral optimal control for military and industrial applications.

[4] Chris Phelps, Qi Gong, Johannes O. Royset, Claire Walton, and Isaac Kaminer. Consistent approximation of a parameter-distributed optimal control problem. *Publication pending*, 2013.

[5] I. Michael Ross. A roadmap for optimal control: The right way to commute. *Annals of the New York Academy of Sciences*, 1065:210–233, January 2006.

[6] D. Ruppen, C. Benthack, and D. Bonvin. Optimization of batch reactor operation under parametric uncertainty- computational aspects. *Journal of Process Control*, 5(4):235–240, 1995.

[7] Peter Terwiesch, Dag Ravemark, Benedikt Schenker, and David W. T. Rippinl. Semi-batch process optimization under uncertainty: Theory and experiments. *Computers and Chemical Engineering*, 22.1:201–213, 1998.