User's Guide for SNCTRL

Philip E. Gill[*] Elizabeth Wong[†]
Department of Mathematics
University of California, San Diego
La Jolla, CA 92093-0112, USA

July 30, 2015

Abstract

SNCTRL is an optimal control interface for the software package SNOPT, written in Fortran 2003 and built and tested on version 7.5-2 of SNOPT. The optimal control problems are solved by discretizing and transforming the problem into a finite-dimensional nonlinear program (NLP). Several methods exist for discretizing the problem including single shooting, multiple shooting and collocation; SNCTRL implements a collocation method. Once the problem has been discretized, SNCTRL sets up the NLP and the required data structures for SNOPT.

Keywords: optimal control problem, discretization by collocation, large-scale nonlinear programming, nonlinear constraints, SQP methods, Fortran software.

1. Introduction

The goal of an optimal control problem is to determine the optimal set of control variables that minimizes an objective function and, at the same time, satisfies a set of ordinary differential equations defining the dynamics of the system.

SNCTRL is an optimal control interface for the software package SNOPT [3], written in Fortran 2003 and built and tested on version 7.5-2 of SNOPT. The optimal control problems are solved by discretizing and transforming the problem into a finite-dimensional nonlinear program (NLP). Several methods exist for discretizing the problem including single shooting, multiple shooting and collocation; SNCTRL implements a collocation method [1]. Once the problem has been discretized, SNCTRL sets up the NLP and the required data structures for SNOPT.

Problems are assumed to be of the form

$$\begin{array}{ll}\underset{y,u,p}{\text{minimize}} & J(y(t_f),u(t_f),p)\\ \text{subject to} & \dot{y}=F(y(t),u(t),p)\\ & C_{\text{low}}\le C(y(t),u(t),p)\le C_{\text{upp}}\\ & y^0_{\text{low}}\le y(t_0)\le y^0_{\text{upp}},\quad y^f_{\text{low}}\le y(t_f)\le y^f_{\text{upp}}\\ & y_{\text{low}}\le y(t)\le y_{\text{upp}},\quad u_{\text{low}}\le u(t)\le u_{\text{upp}},\quad p_{\text{low}}\le p\le p_{\text{upp}}\end{array}$$

where $y(t)$ is an `nY`-vector of state variables, $u(t)$ is an `nU`-vector of control variables and p is an `nP`-vector of time-independent parameters. F is an `nY`-vector of functions known as the state equations. C is an `nC`-vector of algebraic constraints and C_{low} and C_{upp} are the lower and upper bounds of the constraints. Initial and final conditions on the state variables are given by the vectors y^0_{low}, y^0_{upp}, y^f_{low}, and y^f_{upp}. Bounds on the state and control variables and the parameters are given by y_{low}, y_{upp}, u_{low}, u_{upp}, p_{low}, and p_{upp}.

[*]`pgill@ucsd.edu (http://www.ccom.ucsd.edu/~peg)`
[†]`elwong@math.ucsd.edu (http://www.ccom.ucsd.edu/~elwong)`

1.1. Differential equations

SNCTRL discretizes the control problem using a collocation method. The solution of the control problem is approximated via a collocation formula defined on a grid over the interval $[t_0, t_f]$.

The system of ordinary differential equations (ODEs) is given by

$$\dot{y}(t) = \frac{dy}{dt} = F(y(t), u(t), p), \quad t \in [t_0, t_f] \tag{1.1}$$

where $y(t)$ is an `nY`-vector of state variables, $u(t)$ is an `nU`-vector of control variables and p is an `nP`-vector of time-independent parameters. F is an `nY`-vector of functions known as the state equations.

The interval $[t_0, t_f]$ is divided into `N` subintervals $[t_{k-1}, t_k]$ of length h, where $t_k = t_{k-1} + hk$ for $k = 1, \ldots,$ `N` and $t_N = t_f$. At each of these points t_k, a quadrature formula is used to approximate the state and control variables using either the Trapezoid rule or the Hermite-Simpson rule, both described below.

For ease of notation, we use the subscript k to denote the value of a variable at $t = t_k$, e.g., $y_k = y(t_k)$.

1.1.1. Trapezoid method

For the Trapezoid method, the quadrature formula is given by

$$-y_{k+1} + y_k + \frac{h}{2}(F(y_k, u_k, p) + F(y_{k+1}, u_{k+1}, p)) = 0, \quad k = 0, \ldots, \mathtt{N} - 1. \tag{1.2}$$

These equations approximate the values of the state variables at each collocation point t_k for $k = 0, \ldots, \mathtt{N} - 1$, creating $\mathtt{N} \times \mathtt{nY}$ equality constraints for the nonlinear program input to SNOPT. The total number of discretized state and control variables for the nonlinear program is $(\mathtt{N} + 1) \times (\mathtt{nY} + \mathtt{nU})$.

1.1.2. Hermite-Simpson method

The quadrature formula for the Hermite-Simpson method is Simpson's rule, defined as

$$-y_{k+1} + y_k + \frac{h}{6}(F(y_k, u_k, p) + 4F(y_{k2}, u_{k2}, p) + F(y_{k+1}, u_{k+1}, p)) = 0 \tag{1.3}$$

for $k = 0, \ldots, \mathtt{N} - 1$ where $t_{k2} = \frac{1}{2}(t_k + t_{k+1})$.

Since the quadrature formulas require the values of the state and control variables at the midpoints t_{k2}, additional equations are required to interpolate the states and controls. Hermite interpolation is used for the state variables so that

$$y_{k2} = \tfrac{1}{2}(y_k + y_{k+1}) + \frac{h}{8}(F(y_k, u_k, p) - F(y_{k+1}, u_{k+1}, p)),$$

while controls are interpolated using a simple midpoint rule

$$u_{k2} = \frac{1}{2}(u_k + u_{k+1}).$$

The quadrature formula defines $\mathtt{N} \times \mathtt{nY}$ equality constraints with an additional $\mathtt{N} \times (\mathtt{nU} + \mathtt{nY})$ equality constraints interpolating the variables at the midpoints. The total number of discretized state and control variables is $(2\mathtt{N} + 1) \times (\mathtt{nY} + \mathtt{nU})$ for the nonlinear program with the number of collocation points equal to $2\mathtt{N} + 1$.

1.2. Objective function

The objective function $J(y(t_f), u(t_f), p)$ in the control problems for SNCTRL may be defined in Lagrange form or in Mayer form. In Lagrange form, the objective function is given by

$$J(y(t_f), u(t_f), p) = \int_{t_0}^{t_f} \varphi(y(t), u(t), p)\, dt,$$

for some function φ. An extra state variable y_j is defined to represent the objective, i.e.,

$$\dot{y_j}(t) = \varphi(y(t), u(t), p),$$

which implies $J(y(t_f), u(t_f), p) = y_j(t_f)$. The user must indicate to SNCTRL that the jth state variable represents the objective function.

In Mayer form, an extra control variable and an algebraic constraint are needed to represent the objective function. Let u_j denote the extra control variable. Then define an algebraic equality constraint

$$u_j(t) - J(y(t), u(t), p) = 0.$$

The user again must indicate to SNCTRL that the jth control variable represents the objective, i.e., $J(y(t_f), u(t_f), p) = u_j(t_f)$.

In both cases, appropriate bounds and initial conditions need to be set for the extra variable representing the objective function. Examples of both formats are available in Section 7.

1.3. Bounds and algebraic constraints

The control problem may contain simple bounds on the state variables, control variables and parameters as well as initial and terminal conditions on the state variables. General bounds may be specified in the form

$$y_{\text{low}} \le y(t) \le y_{\text{upp}}, \quad u_{\text{low}} \le u(t) \le u_{\text{upp}} \text{ and } p_{\text{low}} \le p \le p_{\text{upp}}. \tag{1.4}$$

In addition, the user may specify initial and terminal conditions on the state variables and initial values for the parameters

$$y^0_{\text{low}} \le y(t_0) \le y^0_{\text{upp}}, \quad y^f_{\text{low}} \le y(t_f) \le y^f_{\text{upp}} \text{ and } p = p_{\texttt{init}}. \tag{1.5}$$

In the nonlinear program input to SNOPT, the discretized state and control variables y_k and u_k must satisfy the bounds of the continuous problem, i.e., for $k = 0, \ldots, \texttt{N}$, y_k and u_k must satisfy

$$y_{\text{low}}(t_k) \le y_k \le y_{\text{upp}}(t_k) \text{ and } u_{\text{low}}(t_k) \le u_k \le u_{\text{upp}}(t_k)$$

and the initial and terminal conditions

$$y^0_{\text{low}} \le y(t_0) \le y^0_{\text{upp}} \text{ and } y^f_{\text{low}} \le y(t_f) \le y^f_{\text{upp}}.$$

Additionally, the user may define algebraic constraints of the form

$$C_{\text{low}} \le C(y(t), u(t), p) \le C_{\text{upp}}, \quad t \in [t_0, t_f] \tag{1.6}$$

where C is an $\texttt{nC}$-vector of algebraic constraints and C_{low} and C_{upp} are the lower and upper bounds of the constraints. If these constraints exist, then the discretized variables and parameters must also satisfy them in the discretized problem, i.e.,

$$C_{\text{low}} \le C(y_k, u_k, p) \le C_{\text{upp}}$$

for $k = 0, \ldots, \texttt{N}$, creating $(\texttt{N} + 1) \times \texttt{nC}$ constraints for the NLP.

1.4. Features

- Various input problem formats
- Matlab interface
- Adaptive refinement
- Support for multiple-phase problems
- Differentiation of linear equations

1.4.1. Problem Formats

There are three versions of SNCTRL allowing users to specify certain problem data in different formats. The versions are `snctrlA`, `snctrlD` and `snctrlS`. The most important differences lie in the format of the subroutines evaluating functions and derivatives and in the structures storing the Jacobian matrices of the state equations and algebraic constraints (if they exist).

The “A” version is the simplest version and is recommended for new users. Jacobian matrices defined by the user are stored in a two-dimensional array. The “D” version is similar to the “A” version but requires a more efficient coding of the subroutines evaluating the functions and derivatives. The ”S” version requires Jacobian matrices to be stored in sparse structures by columns.

A more thorough description is available in Section 2.

1.4.2. Matlab Interface

Matlab m-files and a Fortran mex-file are included in the package for `snctrlD`, the dense version of `snctrl`. Relevant files (in particular, a README file) are located in the directories `matlab` and `mex` in the top directory containing SNCTRL. Matlab scripts for seven examples are also included.

1.4.3. Adaptive Refinement

When adaptive refinement option is turned on, SNCTRL will solve the control problem using the Trapezoid rule and re-solve the problem on a finer grid using the previous solution as a starting point.

Collocation points or nodes are added at the midpoints of $[t_k, t_{k+1}]$ for $k = 0, \ldots, N-1$ if $\|y_{k+1} - z\| > \tau$, where τ is the refinement tolerance and z is a vector of the values of the state variables approximated by Simpson’s rule (Hermite-Simpson method) at t_{k+1}. The interface will attempt to solve the problem until no new nodes are added or until a user-defined limit has been reached. For more details on adaptive refinement (see Section 6.6).

1.4.4. Phases

SNCTRL suppports optimal control problems with multiple phases. If there are k phases, then $[t_0, t_f] = [t_0, t_1) \cup [t_1, t_2) \cup \ldots \cup [t_{k-1}, t_f]$. Each phase can have a different number of intervals as well as different state equations, algebraic constraints, initial conditions, and bounds. Continuity of the state variables is assumed at the end of one phase and at the start of the next. The objective in multiple-phase problems is a function of state or control variables at the end point of each phase.

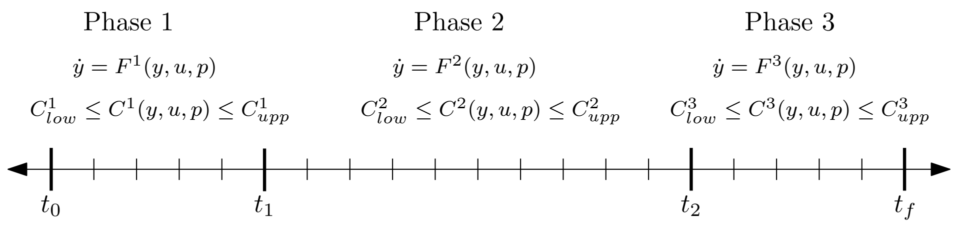
Phase 1
$\dot{y} = F^1(y,u,p)$
$C^1_{low} \le C^1(y,u,p) \le C^1_{upp}$
Phase 2
$\dot{y} = F^2(y,u,p)$
$C^2_{low} \le C^2(y,u,p) \le C^2_{upp}$
Phase 3
$\dot{y} = F^3(y,u,p)$
$C^3_{low} \le C^3(y,u,p) \le C^3_{upp}$
t_0
t_1
t_2
t_f

Figure 1: An example of a problem with multiple phases.

1.4.5. Linear Equations

SNCTRL tries to take full advantage of the features in SNOPT. Of note is the distinction between linear and nonlinear constraints. As a result, SNCTRL tries to exploit this feaeture by allowing the user to specify whether state equations and algebraic constraints are linear or nonlinear with respect to the states, controls and parameters (see Section 1.5).

1.5. `ctProb` Type

The control package is comprised of the Fortran modules `snctrl`, which contains the type definition of `ctProb`, a structure that the user must utilize to pass problem data into the interface.

```
type ctProb
    character(8) :: probName
    integer(ip) :: nY, nU, nP, nC, nPhs
    integer(ip), pointer :: ytype(:,:), ctype(:,:), objL(:), npInt(:)
    real(rp),    pointer :: phsPt(:)
    integer(ip), pointer :: neJ(:), neG(:)
end type ctProb
```

Type components:

All values should be specified by the user in the main program. Arrays in the `ctProb` type must be allocated by the user and must have the correct dimensions.

`probName` is a 8-character name for the problem. `probName` is used in the printed solution. A blank name may be used.

`nY, nU, nP, nC, nPhs` are integers specifying the number of states, controls, parameters, algebraic constraints and phases.

`ytype(:,:),ctype(:,:)` are pointers to two-dimensional integer arrays specifying the nonlinearity or linearity of the state equations and algebraic constraints in each phase. Linearity should be indicated with a value of 1, while nonlinearity is indicated by 0. If these arrays are not allocated, then all equations are assumed to be nonlinear. `ytype` should be of size `nY` by `nPhs`, while `ctype` should be of size `nC` by `nPhs`. For example, if $\texttt{ytype}(2,1) = 1$, then the second state equation in the first phase is linear.

`objL(:)` should point to an integer array of size `nPhs` specifying the location of the objective in each phase. If $1 \leq \texttt{objL(i)} \leq \texttt{nY}$, then the objective is one of the state variables. If $\texttt{nY}+1 \leq \texttt{objL(i)} < \texttt{nY}+\texttt{nU}$, then the objective is one of the control variables. If $\texttt{nY}+\texttt{nU}+1 \leq \texttt{objL(i)} < \texttt{nY}+\texttt{nU}+\texttt{nP}$, then the objective is one of the parameters.

`npInt(:)` is a pointer to an integer array allocated by the user with dimension `nPhs`. This array specifies the number of subintervals in each phase.

`phsPt(:)` is a pointer to a real array allocated by the user with dimension $\texttt{nPhs} + 1$. This array specifies the start and end points of each phase.

The components `nPhs`, `npInt(:)` and `phsPt(:)` are best described with an example.

Suppose the optimal control problem is to be solved in a single phase on the interval $[0, 1]$ with 10 subintervals. The relevant components of the `ctProb` structure need to be set as follows:

```
nPhs     = 1
npInt(1) = 10
phsPt(1) = 0d+0
phsPt(2) = 1d+0
```

For multiple phases, we specify the same data for each phase. Consider a two-phase problem with the interval $[0, 1]$. The interval is split into two pieces, $[0, 0.4]$ and $[0.4, 1.0]$ with 6 subintervals in the first phase and 4 subintervals in the second. The relevant components of the `ctProb` structure need to be set as follows:

```
nPhs      = 2
npInt(1) = 6
npInt(2) = 4
phsPt(1) = 0d+0
phsPt(2) = 0.4d+0
phsPt(3) = 1d+0
```

`neJ(:)` is a pointer to an integer array of size `nPhs` and should be allocated and defined if the sparse version of the interface is being used. The array specifies the number of elements in the Jacobian matrix for the state equations in a particular phase.

`neG(:)` points to an integer array of size `nPhs` and should be allocated and defined if the sparse version of the interface is being used and if algebraic constraints exist in the problem. Similar to `neJ(:)`, the array specifies the number of elements in the Jacobian matrix for the algebraic constraint in a particular phase.

$y_1(t_0)$	$y_1(t_1)$	$\dots$	$y_1(t_f)$
$\vdots$	$\vdots$		$\vdots$
$y_{nY}(t_0)$	$y_{nY}(t_1)$	$\dots$	$y_{nY}(t_f)$
$u_1(t_0)$	$u_1(t_1)$	$\dots$	$u_1(t_f)$
$\vdots$	$\vdots$		$\vdots$
$u_1(t_0)$	$u_{nU}(t_1)$	$\dots$	$u_{nU}(t_f)$

$y_1(t_0)$	$y_1(t_{1,2})$	$y_1(t_1)$	$\dots$	$y_1(t_f)$
$\vdots$	$\vdots$			$\vdots$
$y_{nY}(t_0)$	$y_{nY}(t_{1,2})$	$y_{nY}(t_1)$	$\dots$	$y_{nY}(t_f)$
$u_1(t_0)$	$u_1(t_{1,2})$	$u_1(t_1)$	$\dots$	$u_1(t_f)$
$\vdots$	$\vdots$			$\vdots$
$u_1(t_0)$	$u_{nU}(t_{1,2})$	$u_{nU}(t_1)$	$\dots$	$u_{nU}(t_f)$

Table 1: Layout of discretized variables in the TR(top) and HS(bottom) method

1.6. Subroutine `sncInit`

The module `snctrl` contains all of the subroutines associated with the control interface available to the user. The first subroutine that the user needs to call is `sncInit`. It must be called before any other `snctrl` routine. The routine defines the Print and Summary files, prints a title on both files, and sets all user options to be undefined.

```
subroutine sncInit &
   ( iPrint, iSumm, prob, cw, lencw, iw, leniw, rw, lenrw )
  intent(ip),   intent(in)    :: iPrint, iSumm, lencw, leniw, lenrw
  intent(ip),   intent(inout) :: iw(leniw)
  real(rp),     intent(inout) :: rw(lenrw)
  character(8), intent(inout) :: cw(lencw)
  type(ctProb), target : prob
```

On entry:

`iPrint` defines a unit number for the Print file. Typically `iPrint` $= 9$.

On some systems, the file may need to be opened before `snInit` is called.
If `iPrint` ≤ 0, there will be no Print file output.

`iSumm` defines a unit number for the Summary file. Typically `iSumm` $= 6$.
(In an interactive environment, this usually denotes the screen.)

On some systems, the file may need to be opened before `snInit` is called.
If `iSumm` ≤ 0, there will be no Summary file output.

`cw(lencw), iw(leniw), rw(lenrw)` must be the same arrays that are passed to other `snctrl` routines. `lencw`, `leniw`, `lenrw` must all be at least 500.

On exit:

Some elements of `cw`, `iw`, `rw` are given values to indicate that most optional parameters are undefined. Elements of `prob` are initialized to default values.

2. SNCTRL Interfaces

SNCTRL contains three interfaces that allow the user-defined subroutines specifying variable bounds (`varbds`) and the subroutines that evaluate the state equations (`odecon`) and algebraic constraints (`algcon`), as well as their Jacobians, to be defined in different formats.

The `snctrlA` interface is the easiest to use. The user can specify bounds at the starting and ending points of a phase and general bounds at inner points (see Section 3.2).

State equations and algebraic constraints as well as their Jacobian matrices only need to be evaluated at one given point (see Sections 3.3, 3.4).

In the `snctrlD` and `snctrlS` interfaces, the user has more control over variable bounds. Different bounds can be specifed on any of the discretized variables at any point (see Sections 4.2,5.2).

State equations, algebraic constraints and Jacobian matrices need to be evaluated at all points in a given phase. For `snctrlD`, the Jacobian matrices are assumed to be in dense form and are stored in a two dimensional array (see Section 4.3). For `snctrlS`, the user must store the matrices by columns in sparse structures (see Section 5.3).

All three interfaces have the same parameter list (see Section 2.2). The differences lie in the user-defined subroutines `varbds`, `odecon`, and `algcon`.

A typical invocation is

```
call sncInit( iPrint, iSumm, prob, ... )
call sncSpec( ... )
call snctrlX( Start, prob, ... )
```

with a structure of type `ctProb` defined and initialized before the call to `snctrl`.

2.1. Subroutines associated with `snctrl`

`snctrl` is accessed via the following structures and subroutines:

`ctProb` (Section 1.5) is a type definition in the module `control` used to supply problem data to the interface.

`sncInit` (Section 1.6) must be called before any calls to other `snctrl` routines.

`sncSpec` (Section 6.3) may be called to input a Specs file (a list of run-time options).

`sncSet, sncSeti, sncSetr` (Section 6.4) may be called to specify a single option.

`sncGet, sncGetc, sncGeti, sncGetr` (Section 6.5) may be called to obtain an option's current value.

`snctrlA, snctrlD, snctrlS` (Section 2.2) are the main solvers.

`varbds, odecon, algcon` are all of the subroutines that the user needs to define. These subroutines differ depending on which interface is used.

2.2. Subroutine `snctrl`

This is the main solver for the optimal control problem. The subroutine described here applies to all three interfaces (just replace `snctrl` with `snctrlS`, `snctrlD`, or `snctrlA`).

```
subroutine snctrl ( Start, prob, x, hs, odecon, algcon, varbds, &
                    INFO, mincw, miniw, minrw, nS, nInf, sInf, &
```

```
                      cu, lencu, iu, leniu, ru, lenru, &
                      cw, lencw, iw, leniw, rw, lenrw )
integer(ip), intent(in) :: Start, lencu, leniu, lenru, lencw, leniw, lenrw
integer(ip), intent(inout) :: nS, nInf, iu(leniu), iw(leniw)
real(rp),    intent(inout) :: sInf, rw(lenrw), ru(lenru)
character(8),intent(inout) :: cw(lencw), cu(lencu)
integer(ip), intent(inout), pointer :: hs(:)
real(rp),    intent(inout), pointer :: x(:)
integer(ip), intent(out)   :: INFO, mincw, miniw, minrw
external     :: odecon, algcon, varbds
type(ctProb) :: prob
```

On entry:

`Start` is an integer that specifies how a starting point is to be obtained.

`Start = 0` (Cold start) requests that the CRASH procedure be used.

`Start = 2` (Warm start) means that the `hs` and `x` components of `prob` defines a valid starting point (probably from an earlier call, though not necessarily).

`prob` is a structure of type `ctProb`. The components of `prob` must be defined as described in Section 1.5.

`x` is a pointer to a real array containing a set of initial values for the variables. The user does not need to initialize the pointer if no initial point is provided.

If initialized, `x` should point to an arroy of dimension $(\texttt{nY} + \texttt{nU})(\texttt{N} + 1) + \texttt{nP}$ for the Trapezoid method and $(\texttt{nY} + \texttt{nU})(2\texttt{N} + 1) + \texttt{nP}$ for the Hermite-Simpson method where `N` is the total number of intervals in $[t_0, t_f]$. See Table 1 for a diagram of the variables.

At the end of a call to `snctrl`, `x` points to an array containing the final values of the state variables, control variables and parameters at each time node.

1. For Cold starts (`Start` = `'Cold'`), the elements of `hs` and `x` must be defined. By default, if the user does not allocate the arrays `hs` and `x`, then `snctrl` will set $\texttt{hs}(j) = 0$, $\texttt{x}(j) = 0.0$ for all j. All variables will be eligible for the initial basis.

   Less trivially, to say that the optimal value of variable j will probably be equal to one of its bounds, set $\texttt{hs}(j) = 4$ and $\texttt{x}(j)$ to its lower bound or $\texttt{hs}(j) = 5$ and $\texttt{x}(j)$ to its upper bound as appropriate.

   A CRASH procedure is used to select an initial basis. The initial basis matrix will be triangular (ignoring certain small entries in each column). The values $\texttt{hs}(j) = 0, 1, 2, 3, 4, 5$ have the following meaning:

$\texttt{hs}(j)$	State of variable j during CRASH
$\{0, 1, 3\}$	Eligible for the basis. 3 is given preference
$\{2, 4, 5\}$	Ignored

   After CRASH, variables for which $\texttt{hs}(j) = 2$ are made superbasic. Other entries not selected for the basis are made nonbasic at the value $\texttt{x}(j)$ (or the closest value inside their bounds). See `hs` (on exit) in the description of `snctrl`.

2. For Warm starts, all of `hs` must be 0, 1, 2 or 3 and all of `x` must have values (perhaps from some previous call).

`hs` is a pointer to an integer array containing the initial states for the variables in `x`. It is not necessary for the user to initialize the pointer if no initial point is provided.

`hs` should be of dimension $(\texttt{nY} + \texttt{nU})(\texttt{N} + 1) + \texttt{nP}$ for the Trapezoid method and $(\texttt{nY}+\texttt{nU})(2\texttt{N}+1)+\texttt{nP}$ for the Hermite-Simpson method where `N` is the total number of intervals in $[t_0, t_f]$. See Table 1 for a diagram of the variables and also `x` below for a description of state values.

At the end of a call to `snctrl`, this array will be allocated (if not done by the user) and will contain the final state of the variables. See `hs` (on exit) under the `snctrl` description for more details.

`odecon` is the name of a subroutine that evaluates the vector of state equations $F(y, u, p)$ and (optionally) the Jacobian at some given point. `odecon` must be declared `external` in the routine that calls `snctrl`. See the sections describing each of the three interfaces for more information.

`algcon` is the name of a subroutine that evaluates the vector of algebraic constraint functions $C(y, u, p)$ and (optionally) the Jacobian at some given point. `algcon` must be declared `external` in the routine that calls `snctrl`. See the sections describing each of the three interfaces for more information.

`varbds` is the name of a subroutine that provides upper and lower bounds on the state and control variables, parameters, and algebraic constraints. `varbds` must be declared `external` in the routine that calls `snctrl`. See the sections describing each of the three interfaces for more information.

`cu(lencu), iu(leniu), ru(lenru)` are 8-character, integer and real arrays of user workspace. They may be used to pass data or workspace to your function routines `odecon` and `algcon`(which have the same parameters). They are not accessed or modified by `snctrl`.

If `odecon` and `algcon` don't reference these parameters, you may use any arrays of the appropriate type, such as `cw`, `iw`, `rw` (see next paragraph). Conversely, you should use the `cw`, `iw`, `rw` arrays if `odecon` needs to access `snctrl`'s workspace.

`cw(lencw), iw(leniw), rw(lenrw)` are 8-character, integer and real arrays of workspace for `snctrl`. Their lengths `lencw`, `leniw`, `lenrw` must all be at least 500.

On exit:

`x` points to an array containing the final values of the state and control variables at each time node and the final values of the parameters. This pointer (and `hs`) should be deallocated by the user at the end of the main program.

`hs` contains the final state of the variables as follows:

`hs`(j)	State of variable j	Usual value of `x`(j)
0	nonbasic	Lower bound
1	nonbasic	Upper bound
2	superbasic	Between lower and upper bounds
3	basic	Between lower and upper bounds

Basic and superbasic variables may be outside their bounds by as much as the `Minor feasibility tolerance`. Note that if scaling is specified, the feasibility tolerance applies to the variables of the *scaled* problem. In this case, the variables of the original problem may be as much as 0.1 outside their bounds, but this is unlikely unless the problem is very badly scaled. Check the "Primal infeasibility" printed after the `EXIT` message.

Very occasionally some nonbasic variables may be outside their bounds by as much as the `Minor feasibility tolerance`, and there may be some nonbasics for which `x`(j) lies strictly between its bounds.

If `nInf` > 0, some basic and superbasic variables may be outside their bounds by an arbitrary amount (bounded by `sInf` if scaling was not used).

`INFO` reports the result of the call to `snctrl`. See the SNOPT User's Guide for more info.

`mincw, miniw, minrw` say how much character, integer, and real storage is needed to solve the problem. If `Print level` > 0, these values are printed. If `snctrl` terminates because of insufficient storage (`INFO` = 82, 83 or 84), the values may be used to define better values of `lencw`, `leniw` or `lenrw`.

If `INFO` = 82, the work array `cw(lencw)` is too small. `snctrl` may be called again with `lencw` = `mincw`.

If `INFO` = 83 or 84, the work arrays `iw(leniw)` or `rw(lenrw)` are too small. `snctrl` may be called again with `leniw` or `lenrw` suitably larger than `miniw` or `minrw`. (The bigger the better because it is not certain how much storage the basis factorization needs.)

`nS` is the final number of superbasic variables.

`nInf, sInf` give the number and the sum of the infeasibilities of constraints that lie outside one of their bounds by more than the `Minor feasibility tolerance` *before the solution is unscaled.*

If any *linear* constraints are infeasible, `x` minimizes the sum of the infeasibilities of the linear constraints subject to the upper and lower bounds being satisfied. In this case `nInf` gives the number of variables and linear constraints lying outside their bounds. The nonlinear constraints are not evaluated.

Otherwise, `x` minimizes the sum of the infeasibilities of the *nonlinear* constraints subject to the linear constraints and upper and lower bounds being satisfied. In this case `nInf` gives the number of components of $F(x)$ lying outside their bounds by more than the `Minor feasibility tolerance`. Again this is *before the solution is unscaled.*

3. The `snctrlA` interface

`snctrlA` assumes Jacobian matrices for the state equations and (if applicable) the algebraic constraints are dense. The user also only has to evaluate the functions and Jacobian matrices at one given point.

The user-defined subroutines are described below.

3.1. Subroutines associated with `snctrlA`

`snctrlA` requires the following user-defined subroutines.

`varbds` (Sections 3.2) specifies variable bounds.

`odecon` (Sections 3.3) evaluates the state equations and the corresponding Jacobian matrix at a given point.

`algcon` (Sections 3.4) evaluates the algebraic constraints and the corresponding Jacobian matrix at a given point (if algebraic constraints exist in the problem).

3.2. Subroutine `varbds`

This subroutine is where the user specifies the upper and lower bounds on the variables, parameters, and algebraic constraints.

```
  subroutine varbds ( curPhs, nPhs, nY, nU, nP, nC, y0low, y0upp, &
                      yflow, yfupp, ylow, yupp, ulow, uupp, &
                      plow, pupp, p, clow, cupp )
    integer(ip), intent(in)  :: nPhs, curPhs, nY, nU, nP, nC
    real(rp),    intent(out) :: y0low(nY), y0upp(nY), &
                                yflow(nY), yfupp(nY), &
                                ylow(nY),  yupp(nY),  &
                                ulow(nU),  uupp(nU), &
                                plow(nP), pupp(nP), p(nP), &
                                clow(nC), cupp(nC)
```

On entry:

`curPhs` is an integer specifying the current phase that the user needs to define upper and lower bounds in.

`nPhs, nY, nU, nP, nC` are integers specifying the number of phases, state variables, control variables, parameters and algebraic constraints.

On exit:

`y0low(nY), y0upp(nY)` are real arrays defining lower and upper bounds on the state variables at the starting point of the current phase.

`yflow(nY), yfupp(nY)` are real arrays defining lower and upper bounds on the state variables at the end point of the current phase.

`ylow(nY), yupp(nY)` are real arrays defining the lower and upper bounds on the state variables in the entire phase.

`ulow(nU), uupp(nU)` define the lower and upper on the control variables in the current phase.

`plow(nP), pupp(nP), p(nP)` define the lower and upper bounds on the parameters as well as the initial values of the parameters.

`clow(nC), cupp(nC)` define the lower and upper bounds on the algebraic constraints in the current phase.

3.3. Subroutine `odecon`

This subroutine is where the user defines the differential equations as well as the Jacobian matrix. In general, all derivatives should be specified by the user.

```
subroutine odecon ( snStat, curPhs, nPhs, nY, nU, nP, F, J, y, u, p, &
                    needF, needJ, cu, lencu, iu, leniu, ru, lenru )
  integer(ip), intent(in)    :: snStat, curPhs, nPhs, nY, nU, nP, &
                                needF, needJ, lencu, leniu, lenru
  real(rp),    intent(in)    :: y(nY), u(nU), p(nP)

  integer(ip), intent(inout) :: iu(leniu)
  real(rp),    intent(inout) :: ru(lenru)
  character(8),intent(inout) :: cu(lencu)

  real(rp),    intent(out)   :: F(nY), J(nY, nY+nU+nP)
```

On entry:

`snStat` indicates the first and last calls to `odecon`.

If `snStat` = 0, there is nothing special about the current call.

If `snStat` = 1, `snctrlA` is calling your subroutine for the *first* time. Some data may need to be input or computed and saved.

If `snStat` ≥ 2, `snctrlA` is calling your subroutine for the *last* time. You may wish to perform some additional computations on the final solution.

`curPhs` is an integer specifying the current phase.

`nPhs, nY, nU, nP` are integers specifying the number of phases, state and control variables and parameters.

`y(nY), u(nU), p(nP)` contain the values of the states, controls, and parameters atwhich the functions are to be calculated. These should not be altered.

`needF, needJ` indicate whether or not `F` and `J` need to be assigned during this call of `odecon`.

If `needF` = 0, `F` is not required and is ignored.

If `needF` > 0, then the components of F need to be calculated and assigned to `F`.

If `needJ` = 0, `J` is not required and is ignored.

If `needJ` > 0, then the derivatives of F need to be calculated and assigned to `J`.

`cu(lencu), iu(leniu), ru(lenru)` are the character, integer and real arrays of user workspace provided to `snctrlA`. They may be used to pass information into the function routine and to preserve data between calls.

On exit:

`F(nY)` contains the computed state equations $F(y, u, p)$.

`J(nY,nY+nU+nP)` contains the computed Jacobian matrix of the functions with respect to the states, controls, and the parameters.

3.4. Subroutine `algcon`

This subroutine is where the user defines the algebraic constraint functions as well as its Jacobian matrix. In general, all derivatives should be specified by the user.

```
subroutine algcon ( snStat, curPhs, nPhs, nC, nY, nU, nP, C, G, y, u, p, &
                    needC, needG, cu, lencu, iu, leniu, ru, lenru)
  integer(ip), intent(in)    :: snStat, curPhs, nPhs, nC, nY, nU, nP, &
                                needC, needG, lencu, leniu, lenru
  real(rp),    intent(in)    :: y(nY), u(nU), p(nP)

  integer(ip), intent(inout) :: iu(leniu)
  real(rp),    intent(inout) :: ru(lenru)
  character(8),intent(inout) :: cu(lencu)

  real(rp),    intent(out)   :: C(nC), G(nC, nY+nU+nP)
```

On entry:

`snStat` indicates the first and last calls to `algcon`.

If `snStat` = 0, there is nothing special about the current call.

If `snStat` = 1, `snctrlA` is calling your subroutine for the *first* time. Some data may need to be input or computed and saved.

If `snStat` ≥ 2, `snctrlA` is calling your subroutine for the *last* time. You may wish to perform some additional computations on the final solution.

`curPhs` is an integer specifying the current phase.

`nPhs, nC, nY, nU, nP` are integers specifying the number of phases, algebraic constraints, states, controls and parameters.

`y(nY), u(nU), p(nP)` contain the values of the states, controls, and parameters at which the functions are to be calculated. These should not be altered.

`needC, needG` indicate whether or not `C` and `G` need to be assigned during this call of `algcon`.

If `needC` = 0, `C` is not required and is ignored.

If `needC` > 0, then the components of C need to be calculated and assigned to `C`.

If `needG` = 0, `G` is not required and is ignored.

If `needG` > 0, then the derivatives of C need to be calculated and assigned to `G`.

`cu(lencu), iu(leniu), ru(lenru)` are the character, integer and real arrays of user workspace provided to `snctrlA`. They may be used to pass information into the function routine and to preserve data between calls.

On exit:

`C(nC)` contains the computed algebraic constraints $C(y, u, p)$.

`G(nC,nY+nU+nP)` contains the Jacobian matrix of the algebraic constraints with respect to the states, controls and the parameters.

4. The `snctrlD` interface

`snctrlD` is a dense version of the control interface. The Jacobian matrices of the state equations and (if applicable) the algebraic constraints are stored in dense structures. The user must also evaluate the equations and Jacobian matrices at every point in a phase.

The user-defined subroutines are described below.

4.1. Subroutines associated with `snctrlD`

`snctrlD` requires the following user-defined subroutines.

`varbds` (Section 4.2) specifies variable bounds.

`odecon` (Section 4.3) evaluates the state equations and the corresponding Jacobian matrix at all points in a particular phase. The Jacobian is assumed to be dense.

`algcon` (Section 4.4) evaluates the algebraic constraints and the corresponding Jacobian matrix at all points in a particular phase (if algebraic constraints exist in the problem). The Jacobian is assumed to be dense.

4.2. Subroutine `varbds`

This subroutine is where the user specifies the upper and lower bounds on the discretized variables, parameters and algebraic constraints.

```
subroutine varbds ( curPhs, nPhs, nY, nU, nP, nC, nNodes, lbds, ubds, &
                    x, plbds, pubds, p, clbds, cubds )
  integer(ip), intent(in)  :: curPhs, nPhs, nY, nU, nP, nC, nNodes
  real(rp),    intent(out) :: lbds(nY+nU,nNodes), ubds(nY+nU,nNodes), &
                              x(nY+nU,nNodes), plbds(nP), pubds(nP), &
                              p(nP), clbds(nC), cubds(nC)
```

On entry:

`curPhs` is an integer specifying the current phase that the interface requires bounds in.

`nPhs, nY, nU, nP, nC` are integers specifying the total number of phases, states, controls, parameters, and algebraic constraints in the problem.

`nNodes` is an integer specifying the total number of nodes in the current phase.

On exit:

`lbds(nY+nU,nNodes), ubds(nY+nU,nNodes), x(nY+nU,nNodes)` are real two dimensional arrays that specify the lower bounds, upper bounds and initial values of every discretized variable in the current phase. Refer to Table 1 for the layout of the discretized variables.

Note that an initial point can be specified via this subroutine or in the main solver routine `snctrl`. If a point is specified in both, the one from `snctrl` is taken.

`plbds(nP), pubds(nP), p(nP)` are real arrays that specify the lower bounds, upper bounds, and initial values for the parameters.

`clbds(nC), cubds(nC)` are real arrays specifying the lower and upper bounds of the algebraic constraints in the current phase.

4.3. Subroutine `odecon`

This subroutine is where the user defines the differential equations as well as the dense Jacobian matrix. In general, all derivatives should be specified by the user.

```
  subroutine odecon ( snStat, curPhs, nPhs, nY, nU, nP, nNodes, &
                      F, J, dvar, pvar, needF, needJ, &
                      cu, lencu, iu, leniu, ru, lenru )
    integer(ip), intent(in) :: snStat, curPhs, nPhs, nY, nU, nP, nNodes, &
                               needF, needJ, lencu, leniu, lenru
    real(rp),    intent(in) :: dvar(nY+nU,nNodes), pvar(nP)

    integer(ip), intent(inout) :: iu(leniu)
    real(rp),    intent(inout) :: ru(lenru)
    character(8),intent(inout) :: cu(lencu)

    real(rp),    intent(out) :: F(nY,nNodes), J(nY,nY+nU+nP,nNodes)
```

On entry:

`snStat` indicates the first and last calls to `odecon`.

If `snStat` $= 0$, there is nothing special about the current call.

If `snStat` $= 1$, `snctrlD` is calling your subroutine for the *first* time. Some data may need to be input or computed and saved.

If `snStat` ≥ 2, `snctrlD` is calling your subroutine for the *last* time. You may wish to perform some additional computations on the final solution.

`curPhs` is an integer specifying the current phase.

`nPhs, nY, nU, nP` are integers specifying the number of phases, state and control variables and parameters.

`nNodes` is an integer specifying the total number of nodes in the current phase.

`dvar(nY+nU,nNodes)` is a real two dimensional array containing the values of the discretized variables. These should not be altered. Refer to Table 1 for the layout of the discretized variables.

`pvar(nP)` is a real array containing the values of the parameters.

`needF, needG` indicate whether or not `F` and `G` need to be assigned during this call of `odecon`.

If `needF` $= 0$, `F` is not required and is ignored.

If `needF` > 0, then the components of F need to be calculated and assigned to `F`.

If `needG` $= 0$, `G` is not required and is ignored.

If `needG` > 0, then the derivatives of F need to be calculated and assigned to `G`.

`cu(lencu), iu(leniu), ru(lenru)` are the character, integer and real arrays of user workspace provided to `snctrlD`. They may be used to pass information into the function routine and to preserve data between calls.

On exit:

`F(nY,nNodes)` contains the computed state equations $F(y, u, p)$ at each node.

`J(nY,nY+nU+nP,nNodes)` is a 3-dimensional real array containing the computed Jacobian matrix of the functions with respect to the states, controls, and the parameters at each node in the current phase.

4.4. Subroutine `algcon`

This subroutine is where the user defines the algebraic constraint functions as well as the dense Jacobian matrix. In general, all derivatives should be specified by the user.

```
subroutine algcon ( snStat, curPhs, nPhs, nC, nY, nU, nP, nNodes, &
                    C, G, dvar, pvar, needC, needG, &
                    cu, lencu, iu, leniu, ru, lenru )
  integer(ip), intent(in) :: snStat, curPhs, nPhs, nC, nY, nU, nP, &
                             nNodes, needC, needG, lencu, leniu, lenru
  real(rp),    intent(in) :: dvar(nY+nU,nNodes), pvar(nP)

  integer(ip), intent(inout) :: iu(leniu)
  real(rp),    intent(inout) :: ru(lenru)
  character(8),intent(inout) :: cu(lencu)

  real(rp),    intent(out) :: C(nC,nNodes), G(nC,nY+nU+nP,nNodes)
```

On entry:

`snStat` indicates the first and last calls to `algcon`.

If `snStat` $= 0$, there is nothing special about the current call.

If `snStat` $= 1$, `snctrlD` is calling your subroutine for the *first* time. Some data may need to be input or computed and saved.

If `snStat` ≥ 2, `snctrlD` is calling your subroutine for the *last* time. You may wish to perform some additional computations on the final solution.

`curPhs` is an integer specifying the current phase.

`nPhs, nC, nY, nU, nP` are integers specifying the number of phases, algebraic constraints, states, controls and parameters.

`nNodes` is an integer specifying the total number of nodes in the current phase.

`dvar(nY+nU,nNodes)` is a real two dimensional array containing the values of the discretized variables. These should not be altered. Refer to Table 1 for the layout of the discretized variables.

`pvar(nP)` is a real array containing the values of the parameters.

`needC, needG` indicate whether or not `C` and `Grow, Gval, Gcol` need to be assigned during this call of `algcon`.

If `needC` $= 0$, `C` is not required and is ignored.

If `needC` > 0, then the components of C need to be calculated and assigned to `C`.

If `needG` $= 0$, `Grow, Gval, Gcol` are not required and are ignored.

If `needG` > 0, then the derivatives of C need to be calculated and assigned to `Grow, Gval, Gcol`.

`cu(lencu), iu(leniu), ru(lenru)` are the character, integer and real arrays of user workspace provided to `snctrlD`. They may be used to pass information into the function routine and to preserve data between calls.

On exit:

`C(nC,nNodes)` contains the computed algebraic constraints $C(y, u, p)$ at each node.

`G(nC,nY+nU+nP,nNodes)` is a 3-dimensional real array containing the computed Jacobian matrix of the algebraic constraints with respect to the states, controls, and the parameters at each node in the current phase.

5. The `snctrlS` interface

`snctrlS` is the sparse version of the control interface. The Jacobian matrices of the state equations and (if applicable) the algebraic constraints are stored by columns in sparse structures. The user must also evaluate the equations and Jacobian matrices at every point in a phase.

The user-defined subroutines are described below. Note the subroutine `snctrl` is identical to `snctrlS`.

5.1. Subroutines associated with `snctrlS`

`snctrlS` requires the following user-defined subroutines.

`varbds` (Section 5.2) specifies variable bounds.

`odecon` (Section 5.3) evaluates the state equations and the corresponding Jacobian matrix at all points in a particular phase. The Jacobian is assumed to be sparse.

`algcon` (Section 5.4) evaluates the algebraic constraints and the corresponding Jacobian matrix at all points in a particular phase (if algebraic constraints exist in the problem). The Jacobian is assumed to be sparse.

5.2. Subroutine `varbds`

This subroutine is where the user specifies the upper and lower bounds on the discretized variables, parameters and algebraic constraints.

```
subroutine varbds ( curPhs, nPhs, nY, nU, nP, nC, nNodes, &
                    lbds, ubds, x, plbds, pubds, p, clbds, cubds )
  integer(ip), intent(in)  :: curPhs, nPhs, nY, nU, nP, nC, nNodes
  real(rp),    intent(out) :: lbds(nY+nU,nNodes), ubds(nY+nU,nNodes), &
                              x(nY+nU,nNodes), plbds(nP), pubds(nP), &
                              p(nP), clbds(nC), cubds(nC)
```

On entry:

`curPhs` is an integer specifying the current phase that the interface requires bounds in.

`nPhs, nY, nU, nP, nC` are integers specifying the total number of phases, states, controls, parameters, and algebraic constraints in the problem.

`nNodes` is an integer specifying the total number of nodes in the current phase.

On exit:

`lbds(nY+nU,nNodes), ubds(nY+nU,nNodes), x(nY+nU,nNodes)` are real two dimensional arrays that specify the lower bounds, upper bounds and initial values of every discretized variable in the current phase. Refer to Table 1 for the layout of the discretized variables.

Note that an initial point can be specified via this subroutine or in the main solver routine `snctrl`. If a point is specified in both, the one from `snctrl` is taken.

`plbds(nP), pubds(nP), p(nP)` are real arrays that specify the lower bounds, upper bounds, and initial values for the parameters.

`clbds(nC), cubds(nC)` are real arrays specifying the lower and upper bounds of the algebraic constraints in the current phase.

5.3. Subroutine `odecon`

This subroutine is where the user defines the differential equations as well as the sparse Jacobian matrix. In general, all derivatives should be specified by the user.

```
subroutine odecon ( snStat, curPhs, nPhs, nY, nU, nP, nNodes, F, &
                    Jrow, Jval, Jcol, lenJ, dvar, pvar, &
                    needF, needJ, cu, lencu, iu, leniu, ru, lenru )
  integer(ip), intent(in) :: snStat, curPhs, nPhs, nY, nU, nP, &
                             nNodes, lenJ, needF, needJ, &
                             lencu, leniu, lenru
  real(rp),    intent(in) :: dvar(nY+nU,nNodes), pvar(nP)

  integer(ip), intent(inout) :: iu(leniu)
  real(rp),    intent(inout) :: ru(lenru)
  character(8),intent(inout) :: cu(lencu)

  integer(ip), intent(out) :: Jrow(lenJ), Jcol(nY+nU+nP)
  real(rp),    intent(out) :: F(nY,nNodes), Jval(lenJ,nNodes)
```

On entry:

`snStat` indicates the first and last calls to `odecon`.

If `snStat` $= 0$, there is nothing special about the current call.

If `snStat` $= 1$, `snctrlS` is calling your subroutine for the *first* time. Some data may need to be input or computed and saved.

If `snStat` ≥ 2, `snctrlS` is calling your subroutine for the *last* time. You may wish to perform some additional computations on the final solution.

`curPhs` is an integer specifying the current phase.

`nPhs, nY, nU, nP` are integers specifying the number of phases, state and control variables and parameters.

`nNodes` is an integer specifying the total number of nodes in the current phase.

`dvar(nY+nU,nNodes)` is a real two dimensional array containing the values of the discretized variables. These should not be altered. Refer to Table 1 for the layout of the discretized variables.

`pvar(nP)` is a real array containing the values of the parameters.

`needF, needG` indicate whether or not `F` and `G` need to be assigned during this call of `odecon`.

If `needF` $= 0$, `F` is not required and is ignored.

If `needF` > 0, then the components of F need to be calculated and assigned to `F`.

If `needG` $= 0$, `G` is not required and is ignored.

If `needG` > 0, then the derivatives of F need to be calculated and assigned to `G`.

`cu(lencu), iu(leniu), ru(lenru)` are the character, integer and real arrays of user workspace provided to `snctrlS`. They may be used to pass information into the function routine and to preserve data between calls.

On exit:

`F(nY,nNodes)` contains the computed state equations $F(y, u, p)$ at each node.

`Jrow(lenJ),Jval(lenJ,nNodes),Jcol(nY+nU+nP+1)` are the sparse data structures containing the computed Jacobian matrix of the functions with respect to the states, controls, and the parameters at each node in the current phase. In a given phase, the Jacobian matrix is assumed to have the same structure so `Jrow` and `Jcol` should be the same at each point in a particular phase.

5.4. Subroutine `algcon`

This subroutine is where the user defines the algebraic constraint functions as well as the sparse Jacobian matrix. In general, all derivatives should be specified by the user.

```
subroutine algcon ( snStat, curPhs, nPhs, nC, nY, nU, nP, nNodes, &
                    C, Grow, Gval, Gcol, lenG, dvar, pvar, &
                    needC, needG, cu, lencu, iu, leniu, ru, lenru )
  integer(ip), intent(in) :: snStat, curPhs, nPhs, nC, nY, nU, nP, &
                             nNodes, lenG, needC, needG, &
                             lencu, leniu, lenru
  real(rp),    intent(in) :: dvar(nY+nU,nNodes), pvar(nP)

  integer(ip), intent(inout) :: iu(leniu)
  real(rp),    intent(inout) :: ru(lenru)
  character(8),intent(inout) :: cu(lencu)

  integer(ip), intent(out) :: Grow(lenG), Gcol(nY+nU+nP+1)
  real(rp),    intent(out) :: C(nC,nNodes), Gval(lenG,nNodes)
```

On entry:

`snStat` indicates the first and last calls to `algcon`.

If `snStat` $= 0$, there is nothing special about the current call.

If `snStat` $= 1$, `snctrlS` is calling your subroutine for the *first* time. Some data may need to be input or computed and saved.

If `snStat` ≥ 2, `snctrlS` is calling your subroutine for the *last* time. You may wish to perform some additional computations on the final solution.

`curPhs` is an integer specifying the current phase.

`nPhs, nC, nY, nU, nP` are integers specifying the number of phases, algebraic constraints, states, controls and parameters.

`nNodes` is an integer specifying the total number of nodes in the current phase.

`dvar(nY+nU,nNodes)` is a real two dimensional array containing the values of the discretized variables. These should not be altered. Refer to Table 1 for the layout of the discretized variables.

`pvar(nP)` is a real array containing the values of the parameters.

`needC, needG` indicate whether or not `C` and `Grow, Gval, Gcol` need to be assigned during this call of `algcon`.

If `needC` $= 0$, `C` is not required and is ignored.

If `needC` > 0, then the components of C need to be calculated and assigned to `C`.

If `needG` $= 0$, `Grow, Gval, Gcol` are not required and are ignored.

If `needG` > 0, then the derivatives of C need to be calculated and assigned to `Grow, Gval, Gcol`.

`cu(lencu), iu(leniu), ru(lenru)` are the character, integer and real arrays of user workspace provided to `snctrlS`. They may be used to pass information into the function routine and to preserve data between calls.

On exit:

`C(nC,nNodes)` contains the computed algebraic constraints $C(y, u, p)$ at each node.

`Grow(lenG),Gval(lenG,nNodes),Gcol(nY+nU+nP+1)` are the sparse data structures containing the computed Jacobian matrix of the algebraic constraints with respect to the states, controls, and the parameters at each node in the current phase. In a given phase, the Jacobian is assumed to have the same structure so `Grow` and `Gcol` are the same at each point in a particular phase.

6. Optional parameters

Like SNOPT, the performance of SNCTRL is controlled by a number of parameters or "options". SNCTRL supports optional parameters for SNOPT (see [3]) in addition to several SNCTRL-specific parameters (Section 6.6). Each option has a default value that should be appropriate for most problems. Other values may be specified in two ways:

- By calling subroutine `sncSpec` to read a Specs file (Section 6.1, 6.3).
- By calling the option-setting routines `sncSet`, `sncSeti`, `sncSetr` (Section 6.4).

The current value of an optional parameter may be examined by calling one of the routines `sncGet`, `sncGetc`, `sncGeti`, `sncGetr` (Section 6.5).

6.1. The SPECS file

The Specs file contains a list of options and values in the following general form:

```
Begin options
   Iterations limit             500
   Minor feasibility tolerance  1.0e-7
   Solution                     Yes
End options
```

We call such data a Specs file because it specifies various options. The file starts with the keyword `Begin` and ends with `End`. The file is in free format. Each line specifies a single option, using one or more items as follows:

1. A *keyword* (required for all options).
2. A *phrase* (one or more words) that qualifies the keyword (only for some options).
3. A *number* that specifies an integer or real value (only for some options). Such numbers may be up to 16 contiguous characters in Fortran 77's `I`, `F`, `E` or `D` formats, terminated by a space or new line.

The items may be entered in upper or lower case or a mixture of both. Some of the keywords have synonyms, and certain abbreviations are allowed, as long as there is no ambiguity. Blank lines and comments may be used to improve readability. A comment begins with an asterisk (`*`) anywhere on a line. All subsequent characters on the line are ignored.

The `Begin` line is echoed to the Summary file.

6.2. Multiple sets of options in the Specs file

The keyword `Skip` allows you to collect several sets of options within a single Specs file. In the following example, only the second set of options will be input.

```
Skip Begin options
   Scale all variables
End options

Begin options 2
   Scale linear variables
End options 2
```

The keyword `Endrun` prevents subroutine `sncSpec` from reading past that point in the Specs file while looking for `Begin`.

6.3. Subroutine sncSpec

Subroutine `sncSpec` may be called to input a Specs file (to specify options for a subsequent call of SNCTRL).

```
subroutine sncspec( iSpecs, iExit, cw, lencw, iw, leniw, rw, lenrw )
  integer(ip),  intent(in)    :: iSpecs, lencw, leniw, lenrw
  integer(ip),  intent(inout) :: iw(leniw)
  real(rp),     intent(inout) :: rw(lenrw)
  character(8), intent(inout) :: cw(lencw)
  integer(ip),  intent(out)   :: iExit
```

On entry:

`iSpecs` is a unit number for the Specs file (`iSpecs` > 0). Typically `iSpecs` $= 4$.

On some systems, the file may need to be opened before `sncSpec` is called.

On exit:

`cw(lencw), iw(leniw), rw(lenrw)` contain the specified options.

`INFO` reports the result of calling `sncSpec`. Here is a summary of possible values.

	Finished successfully
101	Specs file read.
	Errors while reading Specs file
131	No Specs file specified (`iSpecs` ≤ 0 or `iSpecs` > 99).
132	End-of-file encountered while looking for Specs file. `sncSpec` encountered end-of-file or `Endrun` before finding `Begin` (see Section 6.2). The Specs file may not be properly assigned.
133	End-of-file encountered before finding `End`. Lines containing `Skip` or `Endrun` may imply that all options should be ignored.
134	`Endrun` found before any valid sets of options.
> 134	There were $i =$ `INFO` $- 134$ errors while reading the Specs file.

6.4. Subroutines sncSet, sncSeti, sncSetr

These routines specify an option that might otherwise be defined in one line of a Specs file.

```
subroutine sncset &
   ( buffer,         iPrint, iSumm, Errors, &
                     cw, lencw, iw, leniw, rw, lenrw )
subroutine sncseti &
   ( buffer, ivalue, iPrint, iSumm, Errors, &
                     cw, lencw, iw, leniw, rw, lenrw )
subroutine sncsetr &
   ( buffer, rvalue, iPrint, iSumm, Errors, &
                     cw, lencw, iw, leniw, rw, lenrw )

character*(*) :: &
     buffer
integer(ip) :: &
     Errors, ivalue, iPrint, iSumm, lencw, leniw, lenrw, iw(leniw)
real(rp) :: &
     rvalue, rw(lenrw)
character :: &
     cw(lencw)*8
```

On entry:

`buffer` is a string to be decoded. Restriction: `len(buffer)` $\leq$ 72 (`sncSet`) or $\leq$ 55 (`sncSeti`, `sncSetr`). Use `sncSet` if the string contains all relevant data. For example,

```
call sncset ( 'Iterations 1000',    iPrint, iSumm, Errors, ... )
```

`ivalue` is an integer value associated with the keyword in `buffer`. Use `sncSeti` if it is convenient to define the value at run time. For example,

```
itnlim = 1000
if (m .gt. 500) itnlim = 8000
call sncseti( 'Iterations', itnlim, iPrint, iSumm, Errors, ... )
```

`rvalue` is a real value associated with the keyword in `buffer`. For example,

```
factol = 100.0d+0
if ( illcon ) factol = 5.0d+0
call sncsetr( 'LU factor tol', factol, iPrint, iSumm, Errors, ... )
```

`iPrint` is a file number for printing each line of data, along with any error messages. `iPrint` = 0 suppresses this output.

`iSumm` is a file number for printing any error messages. `iSumm` = 0 suppresses this output.

`Errors` is the cumulative number of errors, so it should be 0 before the first call in a group of calls to the option-setting routines.

On exit:

`cw(lencw)`, `iw(leniw)`, `rw(lenrw)` hold the specified option.

`Errors` is the number of errors encountered so far.

6.5. Subroutines `sncGet`, `sncGetc`, `sncGeti`, `sncGetr`

These routines obtain the current value of a single option or indicate if an option has been set.

```
integer function sncget &
   ( buffer,         Errors, cw, lencw, iw, leniw, rw, lenrw )
subroutine sncgetc &
   ( buffer, cvalue, Errors, cw, lencw, iw, leniw, rw, lenrw )
subroutine sncgeti &
   ( buffer, ivalue, Errors, cw, lencw, iw, leniw, rw, lenrw )
subroutine sncgetr &
   ( buffer, rvalue, Errors, cw, lencw, iw, leniw, rw, lenrw )

character*(*) :: &
     buffer
integer(ip) :: &
     Errors, ivalue, lencw, leniw, lenrw, iw(leniw)
character(8) :: &
     cvalue, cw(lencw)
real(rp) :: &
     rvalue, rw(lenrw)
```

On entry:

`buffer` is a string to be decoded. Restriction: `len(buffer)` ≤ 72.

`Errors` is the cumulative number of errors, so it should be 0 before the first call in a group of calls to option-getting routines.

`cw(lencw), iw(leniw), rw(lenrw)` contain the current options data.

On exit:

`sncget` is 1 if the option contained in buffer has been set, otherwise 0. Use `sncGet` to find if a particular optional parameter has been set. For example: if

```
i = sncget( 'Hessian limited memory', Errors, ... )
```

then i will be 1 if SNOPT is using a limited-memory approximate Hessian.

`cvalue` is a string associated with the keyword in `buffer`. Use `sncGetc` to obtain the names associated with an MPS file. For example, for the name of the bounds section use

```
call sncgetc( 'Bounds', MyBounds, Errors, ... )
```

`ivalue` is an integer value associated with the keyword in `buffer`. Example:

```
call sncgeti( 'Iterations limit', itnlim, Errors, ... )
```

`rvalue` is a real value associated with the keyword in `buffer`. Example:

```
call sncgetr( 'LU factor tol', factol, Errors, ... )
```

`Errors` is the number of errors encountered so far.

6.6. Description of Optional Parameters

The following shows all valid *keywords* specific to the SNCTRL interface and their *default values.*

```
BEGIN checklist of SPECS file parameters and their default values
  Discretization          TR        * Trapezoid method
  Control Solution        No        * No control solution in the Print file
  Refinement              No        * No refinement
  Refinement Limit        ##        * No limit
  Refinement Tolerance    1d-2      * Default value
End of SPECS file checklist
```

The following is an alphabetical list of the options specific to SNCTRL that may appear in the Specs file, and a description of their effect. For more keywords, see the SNOPT User's Guide [3].

`Discretization`	`TR`	Default
`Discretization`	`HS`	

This option allows the user to choose the collocation method used to discretize the differential system. `TR` and `HS` indicate the Trapezoid method and the Hermite-Simpson method, respectively.

`Control Solution`	`No`	Default
`Control Solution`	`Yes`	

This option specifies whether a brief summary of the problem run and the final state, control and parameter values will be printed to the Print file. The control output is appended to the end of the file below the usual SNOPT output.

`Refinement`	`No`	Default
`Refinement`	`Yes`	

Adaptive refinement will first solve the optimal control problem using the Trapezoid method on a coarse grid. A finer grid is generated based on a refinement tolerance (see below) and the problem is resolved on the new grid. By default, the SNCTRL interface will continue to refine the grid until no new nodes are added.

`Refinement Limit`	i	Default = ∞

This option allows the user to specify how many times the problem will be refined and resolved. By default, the adaptive refinement option will refine and resolve the problem until no new nodes are added to the grid.

`Refinement Tolerance`	τ	Default = `1.0d-2`

This option allows the user to specify the tolerance in the adaptive refinement run. A new node is added at the midpoint of the interval $[t_k, t_{k+1}]$ if $\|y_{k+1} - z\| > \tau$ where z is the vector of state variables with values given by Simpson's rule. The problem is then rerun on the finer grid. This process continues until no new nodes are added to the grid.

7. Examples

We provide examples demonstrating how to use the optimal control interface.

7.1. Breakwell

Consider the Breakwell problem in its original form:

$$\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & \frac{1}{2}\int_0^1 u^2 \, dt \\
\text{subject to} \quad & \dot{y_1} = y_2 \\
& \dot{y_2} = u \\
& y_1(0) = 0 \quad y_2(0) = 1 \\
& y_1(1) = 0 \quad y_2(1) = -1 \\
& y_1 \leq 0.1
\end{aligned}$$

We add an extra state variable to represent the objective and set appropriate bounds.

$$\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & J(t_f) = y_1(1) \\
\text{subject to} \quad & \dot{y_1} = \tfrac{1}{2}u^2 \\
& \dot{y_2} = y_3 \\
& \dot{y_3} = u_1 \\
& y_1(0) = 0 \quad y_2(0) = 0 \quad y_3(0) = 1 \\
& y_2(1) = 0 \quad y_3(1) = -1 \\
& y_2 \leq 0.1
\end{aligned}$$

7.2. Brachistochrone

This example is the classic brachistochrone problem formulated as in [2]. The equations describe the motion of a bead sliding down a frictionless wire in a constant gravitational field. The problem finds the optimal trajectory of a bead sliding down a frictionless wire between two specified points in minimal time.

$$\begin{aligned} \underset{y,u}{\text{minimize}} \quad & \int_0^1 \sqrt{\frac{1+u^2}{1-y}}\, dt \\ \text{subject to} \quad & \dot{y} = u \\ & y(0) = 0 \\ & y(1) = -\tfrac{1}{2} \end{aligned}$$

We add an extra state variable to represent the objective to fit the format required by `snctrl`.

$$\begin{aligned} \underset{y,u}{\text{minimize}} \quad & J(t_f) = y_1(1) \\ \text{subject to} \quad & \dot{y_1} = \sqrt{\frac{1+u^2}{1-y_2}} \\ & \dot{y_2} = u \\ & y_1(0) = 0 \quad y_2(0) = 0 \\ & y_2(1) = -\tfrac{1}{2} \end{aligned}$$

7.3. Catalyst mixing

The catalyst mixing problem in its original form:

$$\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & y_1(1) + y_2(1) - 1 \\
\text{subject to} \quad & \dot{y_1} = u_1(10y_2 - y_1) \\
& \dot{y_2} = u_1(y_1 = 10y_2) - (1 - u_1)y_2 \\
& y_1(0) = 1 \quad y_2(0) = 0 \\
& 0 \le u_1 \le 1
\end{aligned}$$

We add an extra control variable to represent the algebraic objective function and set the appropriate bounds.

$$\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & J(t_f) = u_2(1) \\
\text{subject to} \quad & \dot{y_1} = u_1(10y_2 - y_1) \\
& \dot{y_2} = u_1(y_1 - 10y_2) - (1 - u_1)y_2 \\
& y_1(0) = 1 \quad y_2(0) = 0 \\
& 0 \le u_1 \le 1 \\
& u_2 - y_1 - y_2 = -1
\end{aligned}$$

7.4. Double-integrator plant

The double integrator plant problem was formulated in [4].

$$\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & t_f \\
\text{subject to} \quad & \dot{y_1} = y_2 \\
& \dot{y_2} = u \\
& y_1(0) = \tfrac{1}{2} \quad y_2(0) = 1 \\
& -\tfrac{1}{2} \leq y_1(t_f) \leq \tfrac{1}{2} \\
& y_2(t_f) = 0 \\
& -1 \leq u \leq 1
\end{aligned}$$

For this variable-time problem, we add the extra parameter p_1, to represent t_f. The time interval is then scaled so that the optimization is performed over the fixed interval $[0, 1]$.

$$\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & J(t_f) = p_1 \\
\text{subject to} \quad & \dot{y_1} = y_2 * p_1 \\
& \dot{y_2} = u_1 * p_1 \\
& y_1(0) = \tfrac{1}{2} \quad y_2(0) = 1 \quad y_3(0) = 0 \\
& -\tfrac{1}{2} \leq y_1(t_f) \leq \tfrac{1}{2} \\
& y_2(t_f) = 0 \\
& -1 \leq u_1 \leq 1 \\
& p_1 \geq 0
\end{aligned}$$

7.5. Pendulum

This is a parameter estimation problem that estimates the gravitational constant g [5]. We consider a pendulum moving in a vertical plane.

$$\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & \tfrac{1}{2}((y_1(t_f) - x_f)^2 + (y_2(t_f) - y_f)^2) \\
\text{subject to} \quad & \dot{y_1} = y_3 \\
& \dot{y_2} = y_4 \\
& \dot{y_3} = u_1 y_1/m \\
& \dot{y_4} = u_1 y_2/m \\
& y_1(0) = 0.4 \quad y_2(0) = -0.3 \\
& y_3(0) = 0 \quad y_4(0) = 0 \\
& y_3^2 + y_4^2 + u_1 \ell^2/m - y_2 p_1 = 0 \\
& \tfrac{1}{100} \le p_1 \le 100
\end{aligned}$$

where x_f, y_f, ℓ, and m are constants. ℓ and m represent the length and mass of the pendulum respectively. For this problem, we set $\ell = \frac{1}{2}$, $m = 0.3$, $x_f = -.231625$ and $y_f = -.443109$.

The parameter p_1 represents the unknown gravitational constant to be estimated. y_1 and y_2 represent the coordinates of the pendulum.

We add an extra control variable to represent the objective function and set the appropriate bounds.

$$\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & J(t_f) = u_2(t_f) \\
\text{subject to} \quad & \dot{y_1} = y_3 \\
& \dot{y_2} = y_4 \\
& \dot{y_3} = u_1 y_1/m \\
& \dot{y_4} = u_1 y_2/m \\
& y_1(0) = 0.4 \quad y_2(0) = -0.3 \\
& y_3(0) = 0 \quad y_4(0) = 0 \\
& y_3^2 + y_4^2 + u_1 \ell^2/m - y_2 p_1 = 0 \\
& u_2 - \tfrac{1}{2}\big((y_1 - x_f)^2 + (y_2 - y_f)^2\big) = 0 \\
& \tfrac{1}{100} \le p_1 \le 100
\end{aligned}$$

7.6. Rocket

Here we solve the one-dimensional rocket problem as formulated in [5]. The objective of the problem is to minimize the amount of fuel consumed by the rocket to get to a particular altitude x_f.

Given $[t_0, t_f]$, we split the interval into two phases so that $[t_0, t_f] = [t_0, t_1] \cup (t_1, t_2]$, where $t_2 = t_f$. In the first phase, the rocket has constant thrust while in the second phase, the rocket has no thrust. Assuming fuel consumption is directly proportional to time, we therefore want to minimize t_1.

$$
\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & t_1 \\
\text{subject to} \quad & \dot{y_1} = y_2 \\
& \dot{y_2} = u_1 a - g \\
& y_1(0) = 0 \quad y_2(0) = v_0 \\
& y_1(t_f) = x_f \\
& u_1 = 1 \text{ for } t \in [t_0, t_1] \\
& u_1 = 0 \text{ for } t \in (t_1, t_f]
\end{aligned}
$$

Here, we let $a = 2$, $g = 1$, $v_0 = 0$, and $x_f = 100$. We add an extra state variable and parameter to represent time.

$$
\begin{aligned}
\underset{y,u}{\text{minimize}} \quad & p_1 \\
\text{subject to} \quad & \dot{y_1} = y_2 p_1 \\
& \dot{y_2} = (2u_1 - 1) p_1 \\
& y_1(0) = 0 \quad y_2(0) = 0 \quad y_3(0) = 0 \\
& y_1(t_f) = 100 \\
& u_1 = 1 \text{ for } t \in [t_0, t_1] \\
& u_1 = 0 \text{ for } t \in (t_1, t_f] \\
& p_1 \geq 0
\end{aligned}
$$

7.7. Vanderpol

The Vanderpol problem in its original form:

$$\begin{aligned} \underset{y,u}{\text{minimize}} \quad & \tfrac{1}{2}\int_0^5 y_1^2 + y_2^2 + u^2 \, dt \\ \text{subject to} \quad & \dot{y_1} = y_2 \\ & \dot{y_2} = -y_1 + (1-y_1)^2 y_2 + u \\ & y_1(0) = 1 \quad y_2(0) = 0 \\ & y_1(5) = -1 \quad y_2(5) = 0 \\ & u \leq 0.75 \end{aligned}$$

We add an extra state variable to represent the objective function and set the appropriate bounds.

$$\begin{aligned} \underset{y,u}{\text{minimize}} \quad & J(t_f) = y_1(5) \\ \text{subject to} \quad & \dot{y_1} = \tfrac{1}{2}(y_2^2 + y_3^2 + u_1^2) \\ & \dot{y_2} = -y_3 + (1-y_3)^2 y_2 + u_1 \\ & \dot{y_3} = y_2 \\ & y_1(0) = 0 \quad y_2(0) = 0 \quad y_3(0) = 1 \\ & y_2(5) = 0 \quad y_3(5) = -1 \\ & u_1 \leq 0.75 \end{aligned}$$

7.8. Variable time brachistochrone

The following is the different formulation of the brachistochrone problem [4]. Here, y_1 represents the horizontal distance, y_2 the vertical distance, and u the path angle to the horizontal. Again, we want to find the optimal trajectory of a bead sliding down a frictionless wire between two specified points in minimal time.

$$\begin{array}{rl}
\underset{y,u}{\text{minimize}} & t_f \\
\text{subject to} & \dot{y_1} = y_3 \cos u \\
& \dot{y_2} = y_3 \cos u \\
& \dot{y_3} = \sin u \\
& y_1(0) = 0 \quad y_2(0) = 0 \quad y_3(0) = 0 \\
& y_1(t_f) = 1 \\
& 0 \leq y_2(t_f) \leq 2 \\
& y_2 - \frac{1}{2} y_2 \leq 0.2
\end{array}$$

For this variable time problem, we add an extra parameter, say p, to represents t_f. Accordingly, the time interval is set so that $[t_0, t_f] = [0, 1]$ in `snctrl`.

$$\begin{array}{rl}
\underset{y,u}{\text{minimize}} & J(t_f) = p_1 \\
\text{subject to} & \dot{y_1} = y_3 \cos u_1 * p_1 \\
& \dot{y_2} = y_3 \cos u_1 * p_1 \\
& \dot{y_3} = \sin u_1 * p_1 \\
& y_1(0) = 0 \quad y_2(0) = 0 \quad y_3(0) = 0 \quad y_4(0) = 0 \\
& y_1(t_f) = 1 \\
& 0 \leq y_2(t_f) \leq 2 \\
& y_2 - \frac{1}{2} y_1 \leq 0.2 \\
& 0 \leq p_1
\end{array}$$

References

[1] A. Barclay, P. E. Gill, and J. B. Rosen. SQP methods in optimal control. In R. Bulirsch, L. Bittner, W. H. Schmidt, and K. Heier, editors, *Variational Calculus, Optimal Control and Applications*, volume 124 of *International Series of Numerical Mathematics*, pages 207–222, Basel, Boston and Berlin, 1998. Birkhäuser. 1

[2] P. Dyer and S. R. McReynolds. *The Computation and Theory of Optimal Control*. Academic Press, New York, 1970. 34

[3] P. E. Gill, W. Murray, and M. A. Saunders. User's guide for SNOPT Version 7: Software for large-scale nonlinear programming. Numerical Analysis Report 06-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 2006. 1, 28, 32

[4] A. Y. Lee. Dynamic optimization problems with bounded terminal conditions. *J. Optim. Theory Appl.*, 52(1):151–162, 1987. 36, 40

[5] O. von Stryk. User's guide for DIRCOL: A direct collocation method for the numerical solution of optimal control problems. Technical report, Fachgebiet Simulation und Systemoptimierung, Technische Universität Darmstadt, 2002. 37, 38