Using the Scalable Nonlinear Equations Solvers Package

by

William D. Gropp, Lois Curfman McInnes, and Barry F. Smith
Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

---

**DISCLAIMER**

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Reproduced from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Prices available from (615) 576-8401

Available to the public from the National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Using the Scalable Nonlinear Equations Solvers Package

by

William D. Gropp, Lois Curfman McInnes, and Barry F. Smith

Mathematics and Computer Science Division

Technical Memorandum No. 193

February 1995

The work of the first and third authors was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38. The second author's work was supported by an appointment to the Distinguished Postdoctoral Research Program sponsored by the U.S. Department of Energy, Office of University and Science Education, and administered by the Oak Ridge Institute for Science and Education.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Contents

Abstract

1 Introduction
   1.1 Library Overview ..................................... 2
   1.2 Support for Diverse Users ................................ 3
   1.3 Restrictions ............................................ 4
   1.4 Further Information .................................... 4

2 Simple Examples ........................................ 6
   2.1 Commands for Basic Usage ................................ 6
   2.2 Context Variables ...................................... 7
   2.3 Sample Programs ...................................... 7

3 Solution Methods ....................................... 12
   3.1 Overview of Newton-like Methods .......................... 12
   3.2 Choosing the Technique .................................. 13
   3.3 Extending the Domain of Convergence .................... 13

4 Basic Library Usage .................................. 15
   4.1 Creating the Nonlinear Context .......................... 15
   4.2 Getting Methods from the Command Line .................. 15
   4.3 Solving a Nonlinear System ............................. 16
      4.3.1 Core Routines ................................... 16
      4.3.2 Vector Routines .................................. 17
      4.3.3 Routines for Function Evaluation .................... 17
      4.3.4 Phases of Computation ............................. 17
      4.3.5 Matrix Routines .................................. 18
      4.3.6 Optional Routines ................................ 18
   4.4 Altering the Default Parameters ......................... 19
   4.5 Solving Several Nonlinear Systems ....................... 19

5 The Linear System Solver .............................. 20
   5.1 The Linear Solver Context ............................. 20
      5.1.1 The SLES Context ................................ 20
      5.1.2 Manipulating the SLES Context ...................... 21
   5.2 Inexact Newton-like Methods ......................... 21
      5.2.1 Choosing the Convergence Strategy .................. 22
      5.2.2 Krylov Methods within PETSc ...................... 23
Using the Scalable Nonlinear Equations Solvers Package

by

William D. Gropp, Lois Curfman McInnes, and Barry F. Smith

Abstract

SNES (Scalable Nonlinear Equations Solvers) is a software package for the numerical solution of large-scale systems of nonlinear equations on both uniprocessors and parallel architectures. SNES also contains a component for the solution of unconstrained minimization problems, called SUMS (Scalable Unconstrained Minimization Solvers). Newton-like methods, which are known for their efficiency and robustness, constitute the core of the package. As part of the multilevel PETSc library, SNES incorporates many features and options from other parts of PETSc. In keeping with the spirit of the PETSc library, the nonlinear solution routines are data-structure-neutral, making them flexible and easily extensible.

This users guide contains a detailed description of uniprocessor usage of SNES, with some added comments regarding multiprocessor usage. At this time the parallel version is undergoing refinement and extension, as we work toward a common interface for the uniprocessor and parallel cases. Thus, forthcoming versions of the software will contain additional features, and changes to parallel interface may result at any time. The new parallel version will employ the MPI (Message Passing Interface) standard for interprocessor communication. Since most of these details will be hidden, users will need to perform only minimal message-passing programming.
Chapter 1

Introduction

The solution of large-scale systems of nonlinear equations and unconstrained minimization problems pervades many areas of computational science and demands robust and flexible solution strategies. While various software packages are currently available for solving such problems, their portability, versatility, and scalability are restricted, especially within parallel environments. The SNES (Scalable Nonlinear Equations Solvers) package addresses these problems by focusing on key issues that promote the effectiveness, ease of use, and extensibility of numerical software. SNES also contains a component for the solution of unconstrained minimization problems called SUMS (Scalable Unconstrained Minimization Solvers). Thus, SNES and SUMS together provide a uniform and versatile framework for developing optimization software and solving application problems.

1.1 Library Overview

The SNES package, which is part of the multilevel PETSc (Portable, Extensible Tools for Scientific Computing) library [8], provides a suite of routines for the numerical solution of large-scale systems of nonlinear equations of the form

\[ F(x) = 0, \quad (1.1) \]

where \( F : \mathbb{R}^n \to \mathbb{R}^n \), and \( \mathbb{R}^n \) denotes the n-dimensional Euclidean subspace of real n-vectors. The SUMS component of SNES encompasses routines for numerically solving unconstrained minimization problems of the form

\[ \min \{ f(x) \}, \quad (1.2) \]

where \( f : \mathbb{R}^n \to \mathbb{R} \).

The SNES library is designed to take advantage of many similarities in the approaches for solving systems of nonlinear equations and unconstrained minimization problems. Much infrastructure within SNES is common to both the code for solving systems of nonlinear equations and the SUMS component for unconstrained minimization. The construction of two components within the SNES package enables us to exploit individual features of the solution strategies and to communicate more effectively with different user communities. However, from the high-level user's perspective the components of SNES differ only slightly.

Newton-like methods, which are known for their efficiency and robustness, constitute the core of the software. In particular, several variants of line search strategies and trust region techniques are included. In addition to the numerical routines, a variety of performance monitors are available, such as operation counts, timing routines, and graphical displays using the X Windows interface.
As part of the PETSc library, the SNES package is supported by various other modules for data structures, matrix-vector operations, linear solution strategies, and communication routines. Figure 1.1 indicates the hierarchical PETSc library structure. Although the full power of lower-level PETSc routines is available, the user need not be familiar with their details.

![Figure 1.1: Organization of the PETSc Library](image)

In keeping with the spirit of PETSc, the nonlinear equations solution routines use an aggressive data-structure-neutral implementation that minimizes dependence on particular data structures. Thus, the code works as written, without change on both serial and parallel machines, independent of storage schemes used for vectors, matrices, etc. In addition, the library routines employ short calling sequences, with a consistent interface among different algorithms that solve a particular class of problems.

### 1.2 Support for Diverse Users

A modular design is used in the nonlinear solvers and minimization routines, thereby facilitating experimentation with newly developed solution strategies and customization according to the user's needs. In addition, a simple yet powerful interface allows the user to control virtually all aspects of the solution process, so that SNES serves a wide range of users with varying needs and different levels of sophistication. Listed below are the categories of SNES users, along with the chapters of this document that are most pertinent for each.

- **Beginning** - User employs defaults for nonlinear solvers, data structures, and linear solvers; Chapters 2, 3, and 4.

- **Intermediate** - User modifies various defaults and customizes methods for preconditioning and/or linear solution; Chapters 5 and 6.
• **Advanced** - User provides customized data structures and/or defines alternative nonlinear solution techniques; Chapters 6 and 7.

As indicated above, novice users should begin with Chapter 2, which illustrates the SNES package's utility and simplicity by considering some introductory C and Fortran examples. Chapter 3 then describes the available Newton-like methods for solving systems of nonlinear equations and unconstrained minimization problems, and Chapter 4 explains basic usage of the SNES library. Chapter 5 focuses on solving the linearized Newton systems, with some discussion of inexact Newton techniques. This chapter explains the PETSc suites of preconditioners and iterative solvers, thereby targeting intermediate users who wish to control this aspect of the solution process. Matrix evaluation, including use of the PETSc data structures, is discussed in Chapter 6. Chapter 7 addresses the needs of advanced users by indicating the procedure for customizing the nonlinear solvers.

The remainder of the manual addresses general topics that are potentially of interest to all users. Chapter 8 provides information about performance monitoring, and Chapter 9 discusses the registry process. Some comments about the multiprocessor version of SNES are given in Chapter 10. Finally, the appendixes illustrate the installation and practical use of the SNES library. In particular, Appendix A discusses the details of retrieving and installing SNES as part of the PETSc library. Appendix B explains the SNES organization, including the details of linking the various component libraries, using customized makefiles, and employing the Fortran interface. Lastly, Appendix C summarizes the SNES routines.

### 1.3 Restrictions

The current version of SNES is designed for solving nonlinear systems of equations and unconstrained minimization problems by using double-precision arithmetic. Routines for finite difference approximation of derivatives, automatic differentiation, and matrix-free variants will be also included in a future release. In addition, methods for handling nonlinear least squares and simple constraints are under development and will be included at a later date.

This manual focuses on uniprocessor usage of the SNES package, with some added comments regarding the multiprocessor version. At this time the parallel version is undergoing refinement and extension, as we work toward a common interface for the uniprocessor and parallel cases. Thus, forthcoming versions of the software will contain additional features, which will be described in a future version of the users guide.

Only minor differences exist in using the C and Fortran versions of the software. Thus, the manual focuses on the uniprocessor C variant and indicates additional considerations necessary for Fortran usage.

### 1.4 Further Information

All routines within the SNES package and the rest of the PETSc library are extensively documented with on-line man pages. The shell script `tools.core/bin/toolman` controls an X Windows session of `xman`, which provides information regarding these routines. Note that `tools.core` indicates the root PETSc directory, and the man pages themselves are stored in `tools.core/man`.

Since the SNES package is intended for use within the larger PETSc library, users should refer to the on-line PETSc documentation for an overview of the software as a whole. The SNES users manual only briefly discusses certain topics that are covered in depth by other guides for the various
PETSc components. Additional information can be found in the *SLES Users Manual* [6], the *Users Manual for KSP: Data-Structure-Neutral Codes Implementing Krylov Space Methods* [7], and the *Users Manual for the Chameleon Parallel Programming Tools* [5].

These documents, as well as additional information regarding our design philosophy and the use of PETSc for solving large-scale applications, are available via the World Wide Web at the PETSc home page:

http://www.mcs.anl.gov/petsc/petsc.html
Chapter 2

Simple Examples

As explained in Section 1.2, SNES is intended for a wide range of users with varying needs and different levels of experience. Default solution procedures and data structures are provided, so that to solve a system of nonlinear equations, the user must simply supply routines to evaluate the residual (1.1) and its corresponding Jacobian matrix. Likewise, to solve an unconstrained minimization problem, the user need only supply routines to evaluate the objective function (1.2), gradient, and Hessian matrix. To reduce the burden of providing derivative information, future versions of SNES will also provide options for automatic differentiation and finite difference approximation of derivatives.

2.1 Commands for Basic Usage

Usage of the SNES package follows the style common to other PETSc libraries, where a simple sequence of commands executes the solution process. These commands enable the user to manipulate the nonlinear and user contexts, which provide an interface to SNES and are described in the following section.

- Create the nonlinear and user-defined contexts (NLCreate).
- Select the vector operations (NLCreate<type>Vectors, where <type> denotes the vector type).
- Establish the routines to evaluate the residual and Jacobian or the objective function, gradient, and Hessian (NLSet<type>Routine, where <type> denotes the routine type).
- Specify the step procedure for the global iterations (NLSet<XXX>).
- Set optional parameters and performance monitors (NLSet<XXX>).
- Set up the nonlinear solver (NLSetUp).
- Execute the nonlinear solver (NLSSolve).
- Destroy the nonlinear context (NLDDestroy).

Chapter 4 discusses these commands in detail.
2.2 Context Variables

Context variables are the key to the organization of SNES (and the other PETSc components as well). The nonlinear context, NLCtx, contains descriptive information about the nonlinear solvers. As discussed in Section 2.1 of the SLES Users Manual [6], the use of such a context offers the advantages of easy-to-use nested algorithms, short argument lists, and ease of extension. Embedded within this structure are vector and matrix contexts, controls for performance monitoring, and miscellaneous information regarding various phases of the solution process.

The nonlinear context provides a simple yet powerful interface to SNES. This context is available throughout all SNES routines, where it usually is passed as the first function argument. The user need not be familiar with the context’s detailed structure, since simple routines can access and manipulate its contents. Even Fortran programmers with no knowledge of C have full control of context variables from Fortran. A user-defined context, which typically contains problem specific information, can also be embedded within the nonlinear context. Thus, user-defined data is available throughout the entire solution process.

2.3 Sample Programs

Figures 2.1 (Fortran) and 2.2 (C) illustrate the stages for solving a system of nonlinear equations on a single processor. The test case is the solid fuel ignition problem from the MINPACK-2 test suite [1]. In the Fortran program of Figure 2.1, the user directly employs the MINPACK-2 routines for forming the initial guess, residual, and Jacobian matrix. In this case, the user controls matrix memory allocation using the dense format. The C program of Figure 2.2 forms the Jacobian matrix with a customized variant of the MINPACK-2 routine. Here the appropriate storage is dynamically allocated using the default PETSc data structures for sparse matrix storage. In both cases the SLES routines are used to solve the resulting linear systems.

Even at the high level illustrated by these simple examples, the user can easily select among the suites of nonlinear solvers, data structures, and linear solution methods (either direct or iterative) provided within PETSc. The user can also set customized monitoring routines and parameters.

In addition, the SNES package provides flexibility that enables the user to control all aspects of the solution procedure, including data structures and linear solution methods. By employing the SNES infrastructure, the user can easily augment the nonlinear solvers package to include other techniques. Also, the nonlinear solution routines are designed to facilitate customization of the solution process. For example, the Newton-like methods can be subdivided into three basic phases of computation within each global step of the solution process. In addition to providing default routines for these phases, we present several alternatives that can be used as templates in the design of custom routines. These features are discussed further in the following chapters.
program main

include ' ../../nonlin/nlfort.h'

parameter (null = 0, mx = 5, my = 4, ndim = mx*my)

integer neP, its, mx, my, ndim, null
integer nlmethod, svmethod, kspmethod
integer NLCreate, NLSolve
double precision x(ndim)

external residual, initguess, fortjacobian

c *** Set method defaults
nlmethod = NLE_NTR1
svmethod = SVBDD
kspmethod = ITGMRES

c *** Create nonlinear solver
neP = NLCreate(nlmethod, 0)

c *** Set SLES information
call NLSlesSetRoutines(neP, fortjacobian)
call NLSlesSetMethod(neP, svmethod)
call NLSlesSetKSPMethod(neP, kspmethod)

c *** Set matrix information: user allocation of dense Jacobian
call NLSpMatSetType(neP, MATDENSE)
call NLSpMatSetRows(neP, ndim)
call NLSpMatSetCols(neP, ndim)
call NLSpMatSetDeclaredRows(neP, ndim)
call NLSpMatSetUserAllocate(neP)

c *** Set data structures and routines
call ILSetSolution(neP, x)
call NLSetResidualRoutine(neP, residual, 1)
call NLSetInitialGuessRoutine(neP, initguess)
call NLCreateDVectors(neP, ndim)
call NLSetUp(neP)

c *** Execute nonlinear solver
its = NLSolve(neP)
write(6,*) 'number of Newton iterations = ', its

c *** Destroy nonlinear solver
call NLDestroy(neP)

stop
end


c *** Continued on next page ...
c *** Continued from previous page ...

subroutine residual( nlP, x, f )

parameter (null = 0, mx = 5, my = 4, ndim = mx*my)
double precision x(ndim), f(ndim)
integer nlP, mx, my, ndim, null

call dsfijj( mx, my, x, f, null, ndim, 'F', 6.0 )

return
end

c subroutine initguess( nlP, x )

parameter (null = 0, mx = 5, my = 4, ndim = mx*my)
double precision x(ndim)
integer nlP, mx, my, ndim, null

call dsfijj( mx, my, x, null, null, ndim, 'XS', 6.0 )

return
end

c subroutine fortjacobian( nlP, x )

parameter (null = 0, mx = 5, my = 4, ndim = mx*my)
double precision x(ndim), fjac(ndim,ndim)
integer nlP, mx, my, ndim, null

call dsfijj( mx, my, x, null, fjac, ndim, 'J', 6.0 )
call NLSMatSetDenseData( nlP, fjac )

return
end

Figure 2.1: Simple Fortran Example: Using SNES to Solve a Nonlinear System
#include "tools.h"
#include "nonlin/nlall.h"
#include "user.h"

void FormJacobian(), FormResidual(), FormInitialGuess();

main( argc, argv )
int argc;
char **argv;
{
    NLCtx *nlP;        /* nonlinear solver context */
    UserCtx *user;     /* user-defined context */
    NLMETHOD method = NLE-NLS1;  /* nonlinear solution method */
    SVMETHOD svmethod = SVBDD;  /* preconditioner */
    ITMETHOD kspmethod = ITGMRES; /* iterative solver */
    double *x;          /* final solution */
    int mx = 5, my = 4;  /* test problem size */
    int param = 6.0;     /* test problem nonlinearity parameter */
    int ndim = mx*my;    /* problem dimension */
    int i, iters;

    /* Form optional user-defined context */
    user = NEW(UserCtx);
    user->mx = mx; user->my = my; user->problem.param = param;

    /* Create nonlinear solver context */
    NLMGetMethod( argc, argv, 1, "-nlmethod", &method);
    nlP = NLCreate( method, (void *)user );

    /* Set various routines */
    NLSetInitialGuessRoutine( nlP, FormInitialGuess );
    NLSetResidualRoutine( nlP, FormResidual, 1 );

    /* Set SLES information */
    NLSlesSetRoutines( nlP, FormJacobian );
    NLSlesSetMethod( nlP, svmethod );
    NLSlesSetKSPMethod( nlP, kspmethod );
    NLSpMatSetRows( nlP, ndim );
    NLSpMatSetCols( nlP, ndim );

    /* Set internal data structures */
    NLCreateDVectors( nlP, ndim );
    NLSetUp( nlP );

    /* Execute solution method */
    iters = NLSolve( nlP );
    x = (double *)NLGetSolution( nlP );
    for (i=0; i<ndim; i++) printf( "sol[%d] = %g\n", i, x[i] );

    /* Free data structures */
    NLDestroy( nlP ); FREE( user );

    return 0;
}
/* Continued on next page ... */
/* Continued from previous page ... */

void FormResidual( nlp, x, f )
NLCtx *nlp;
void *x, *f;
{
    int iflag = 1, i, err;

    err = Dsfi( nlp, x, f, iflag );
    if (err) SETERRC(1,"Error in residual evaluation.");
}

void FormInitialGuess( nlp, x )
NLCtx *nlp;
void *x;
{
    int iflag = -1, err;

    err = Dsfi( nlp, x, 0, iflag );
    if (err) SETERRC(1,"Error in initial approximation.");
}

void FormJacobian( nlp, x )
NLCtx *nlp;
void *x;
{
    int iflag = 2, err;

    err = Dsfi( nlp, x, 0, iflag );
    if (err) SETERRC(1,"Error in Jacobian evaluation.");
}

Figure 2.2: Simple C Example: Using SNES to Solve a Nonlinear System
Chapter 3

Solution Methods

Newton-like techniques provide the foundation of the SNES package, since they are robust and efficient for solving large-scale systems of nonlinear equations and unconstrained minimization problems. These methods generally exhibit fast local convergence and can be combined with various strategies to extend the domain of convergence (see, e.g., [13]). We briefly discuss these techniques, emphasizing those issues that are most important for efficient software usage.

3.1 Overview of Newton-like Methods

The following properties are assumed when using Newton’s method to solve nonlinear systems of the form (1.1): There exists \( x^* \in \mathbb{R}^n \) such that \( F(x^*) = 0 \), \( F \) is continuously differentiable within a neighborhood of \( x^* \), and \( F'(x^*) \) is nonsingular. The notation \( F'(x) \) indicates the Jacobian matrix of \( F \) evaluated at \( x \), as given by

\[
F(x) = \begin{bmatrix}
\frac{\partial F_i}{\partial x_j}
\end{bmatrix}.
\]

The general form of the \( n \)-dimensional Newton’s method for solving (1.1) is

\[
x_{k+1} = x_k - [F'(x_k)]^{-1}F(x_k), \quad k = 0, 1, \ldots,
\]

where \( x_0 \) is an initial approximation to the solution and \( F'(x_k) \) is nonsingular. In practice, the Newton iteration (3.1) is implemented by the following two steps:

1. Solve \( F'(x_k)\Delta x_k = -F(x_k) \).
2. Update \( x_{k+1} = x_k + \Delta x_k \).

Similarly, when using Newton’s methods for solving an unconstrained minimization problem of the form (1.2), the following properties are assumed: \( f \) is twice continuously differentiable within a neighborhood of \( x^* \), and \( \nabla^2 f(x^*) \) is positive definite. The general form of Newton’s method for solving (1.2) is

\[
x_{k+1} = x_k - [\nabla^2 f(x_k)]^{-1}\nabla f(x_k), \quad k = 0, 1, \ldots,
\]

where \( x_0 \in \mathbb{R}^n \) is an initial approximation to the solution, and \( \nabla^2 f(x_k) \) is positive definite. The iteration (3.4) is usually implemented by

1. Solve \( \nabla^2 f(x_k)\Delta x_k = -\nabla f(x_k) \).
2. Update \( x_{k+1} = x_k + \Delta x_k \).
Since Newton's method is only locally convergent, we extend its domain of convergence by using two basic approaches: line search and trust region (or restricted step) techniques. The salient features of these techniques are discussed in the Section 3.3, while more detailed discussions can be found in, for example, [3], [4].

3.2 Choosing the Technique

As listed below and described in Section 3.3, the current version of the SNES package includes several variants of Newton's method. All methods for solving nonlinear systems of equations begin with the prefix NLE, while those within the SUMS component for solving unconstrained minimization problems use the prefix NLM.

- NLE_NLS1: Line search
- NLE_NTR1: Trust region variant 1
- NLE_NTR2_DOG: Trust region variant 2 with a dogleg step
- NLE_NTR2_LIN: Trust region variant 2 with a linear step
- NLM_NLS1: Line search (SUMS)
- NLM_NTR1: Trust region (SUMS)

3.3 Extending the Domain of Convergence

Any solution of the system of nonlinear equations, given by (1.1), is also a solution of the following related minimization problem:

\[
\min_{x \in \mathbb{R}^n} f(x), \tag{3.7}
\]

where the objective function is given by

\[
f(x) = \frac{1}{2} \| F(x) \|_2^2 = \frac{1}{2} F(x)^T F(x). \tag{3.8}
\]

This formulation facilitates the discussion of schemes to increase the domain of convergence of Newton-like methods, since a natural strategy is to require that each step decrease the value of some norm of \( F \). Both the line search and trust regions strategies employ models of the objective function (3.8), which enable the prediction of a local minimizer at each step of the solution process. Note, however, that since local minimizers of (3.7) do not necessarily solve the original problem, we maintain the structure of (1.1) whenever feasible.

The basic premise of line search strategies is to backtrack along the Newton direction if the full Newton step is unsatisfactory. We implement a Newton method with a line search for solving systems of nonlinear equations, where a cubic backtracking strategy is employed as the default [3]. This method is called NLE_NLS1. In addition, the line search method NLM_NLS1 [12] is provided for solving unconstrained minimization problems.

Trust region techniques approximately minimize a local model of the objective function within an estimated region in which the model is a satisfactory representation of the function. We implement several versions of trust region methods for solving systems of nonlinear equations. The first, called NLE_NTR1, is taken from the MINPACK project [11]. An additional trust region method,
NLE\textunderscore NTR2, is based on the Levenberg-Marquardt algorithm [10]. This method allows the selection of various step types, where defaults are provided for both a dogleg step ([14], [15]) and a linear step, as given respectively by the methods NLE\textunderscore TR2\textunderscore DOG and NLE\textunderscore TR2\textunderscore LIN. Finally, the method NLM\textunderscore NTR1, which is based on the work of Steihaug [16], is provided for solving unconstrained minimization problems.
Chapter 4

Basic Library Usage

This chapter describes in more detail the basic stages mentioned in Chapter 2 for using the SNES library.

4.1 Creating the Nonlinear Context

As discussed in Section 2.2, the nonlinear context, NLCtx, provides a flexible interface to the SNES package in the form of a C structure that contains descriptive information for the nonlinear solvers. The nonlinear context includes data that is specific to each solution technique as well as data that is common to all solution methods. This information regarding vectors, matrices, user-defined data, and default interfaces to other components of PETSc is available throughout the entire SNES package.

The user context, which typically contains problem-specific data, is represented by a void pointer and thus can take any form desired by the user. The routine NLCreate forms the nonlinear context and has the format

\[
\text{NLCtx } \ast\text{NLCreate( NLMETHOD nlmethod, void } \ast\text{usrP )},
\]

where nlmethod indicates the solution method as listed in Section 3.2, and usrP denotes the optional user-defined context, which is set to NULL if not in use.

4.2 Getting Methods from the Command Line

The nonlinear solution method can be specified by command line arguments, which are extracted by the routine NLGetMethod:

\[
\text{void NLGetMethod( Argc, argv, fg, sname, nlmethod )}
\]

int *Argc, fg;
char **argv, *sname;
NLMETHOD *nlmethod;

Argc and argv are the main routine’s arguments. The flag fg indicates whether the argument should be removed from the list if found. The variable sname is to be used on the command line; if sname is NULL, it has the default value -nlmethod. If a nonlinear solution method is specified on the command line, the method is returned in nlmethod; however, if no method is indicated, nlmethod remains unchanged. Methods currently supported by the SNES package are given below:
NLE-NLS1: nlmethod = enlsl,
NLE-NTR1: nlmethod = entr1,
NLE-NTR2_DOG: nlmethod = entr2_dog,
NLE-NTR2_LIN: nlmethod = entr2_lin,
NLM-NLS1: nlmethod = mnlsl (SUMS),
NLM-NTR1: nlmethod = mntrl (SUMS).

Note the naming convention for the solution techniques: Newton-like methods for solving systems of nonlinear equations begin with the prefix en, while Newton-like techniques for solving unconstrained minimization problems begin with mn. Since Fortran has no portable methods for accessing command line arguments, NLGetMethod is restricted to C programs only.

4.3 Solving a Nonlinear System

We next elaborate on the steps mentioned in Section 2.1 for basic use of the SNES library.

4.3.1 Core Routines

The four core SNES commands are listed below, where these routines must be used in the specified order:

- NLCreaten
- NLSQtUp
- NLSolve
- NLDestroy.

As discussed in Section 4.1, the user first employs NLCreaten to form a nonlinear solver context. Next, NLSQtUp initializes the nonlinear solver, and NLSolve executes the algorithm. Finally, NLDestroy frees the memory used by the nonlinear solver. The formats of these last three routines are

```c
void NLSQtUp( NLCtx *nlP )
int NLSolve( NLCtx *nlP )
void NLDestroy( NLCtx *nlP ),
```

where nlP is the nonlinear context formed by NLCreaten. Note that NLSolve returns the number of iterations required for convergence of the Newton-like method.

In addition to these four routines, several others are required for using the SNES package. The following routines must be used after calling NLCreaten but before calling NLSQtUp, although their internal order is unspecified:

- NLCreatenDVectors or NLCreatenDVectors,
- NLSQtSetResidualRoutine or NLSQtFunctionRoutine
  and NLSQtGradientRoutine (SUMS),
- NLSQtSetSetRoutines or NLSQtMatrix
  and NLSQtStepSetUp
  and NLSQtStepSolve
  and NLSQtStepDestroy
  and NLSQt<XXX>Search.

We discuss these routines in detail in the following sections. Note that a variant of NLSQt<XXX>Search is needed only for the trust region methods NLE_NTR2_DOG and NLE_NTR2_LIN to determine the step search procedure. The options for this routine are discussed in Chapter 7.
4.3.2 Vector Routines

When using the PETSc uniprocessor vector defaults within the nonlinear solver, one can use either `NLCreateDVectors` or `NLCreateDBVectors` to set double-precision or BLAS double-precision vector operations, respectively. These routines have the form

```c
void NLCreateD<B>Vectors( NLCtx *nlP, int ndim ),
```

where `ndim` indicates the vector length. The user can also define alternative vector operations by merely following the conventions of the vector component of the PETSc package.

4.3.3 Routines for Function Evaluation

When solving a system of nonlinear equations, `NLSetResidualRoutine` is used to indicate the routine for evaluating the residual, given by \( F(x) \) within \((1.1)\). This routine has the format

```c
void NLSetResidualRoutine( NLCtx *nlP, void (*Funct)(), int rneg ),
```

where `rneg` can be set to zero to indicate that `Funct` evaluates the residual itself; a nonzero value of `rneg` indicates that `Funct` instead provides the negative of the residual, \(-F(x)\). The user-defined routine `Funct` has the calling sequence

```c
Funct( NLCtx *nlP, void *invector, void *outvector ),
```

where `invector` and `outvector`, respectively, denote the current iterate and the function evaluated at this iterate.

When using SUMS to solve an unconstrained minimization problem, the routines for evaluating the objective function and gradient are set, respectively, with

```c
void NLSetFunctionRoutine( NLCtx *nlP, void (*Funct)() )
void NLSetGradientRoutine( NLCtx *nlP, void (*Funct)() ),
```

where `Funct` has the format indicated above. These routines are used instead of `NLSetResidualRoutine`, which is employed only for solving systems of nonlinear equations.

4.3.4 Phases of Computation

One of the distinctive features of the SNES package is its modular format, which allows easy customization of the nonlinear solvers. Each global step of the Newton-like methods can be partitioned into three basic stages, which we label `step-setup`, `step-compute`, and `step-destroy`. Within these phases, the user can completely control the computation of a step within the nonlinear solution process. For example, the user can easily switch the techniques for solving linear systems during successive global iterations by minor revisions to the default `step-setup` phase. These phases of computation can be set with the routines

- `NLSetStepSetUp`
- `NLSetStepCompute`
- `NLSetStepDestroy`,

which are discussed in Chapter 7. Alternatively, the user can employ SLES defaults for these phases by calling `NLSlesSetRoutines` once at the outset of the computation.
As the core of the solution process, the step-compute phase is executed one or more times during each iteration of the Newton-like schemes to compute prospective steps. This stage usually involves solving a linear system.

The step-setup stage, which can optionally be executed a single time before one or more step-compute phases, typically consists of evaluating the Jacobian (or Hessian) matrix and initializing the step-compute process. Separating the step-compute and step-setup phases is advantageous in many cases when several steps must be attempted before determining one that is satisfactory. In this instance, such tasks as evaluating the matrix and initializing the preconditioner (e.g., factoring the matrix for an incomplete LU preconditioner) need to be done a single time during the step-setup phase. These tasks need not be repeated until a successful step has been computed.

Finally, the step-destroy phase can optionally be performed near the conclusion of a global iteration to handle such tasks as releasing memory space used by the Jacobian (or Hessian) matrix and the linear solver. The step-destroy routine is called once after completing a series of step-compute procedures, but before issuing another call of the step-setup routine.

### 4.3.5 Matrix Routines

The user can set the routine for evaluating the Jacobian (or Hessian) matrix with `NLSlesSetRoutines`, which has the format

```c
void NLSlesSetRoutines( NLCtx *nlP, void (*Matrix)(NLCtx *nlP, double *x) );
```

The user-defined routine `Matrix` has the calling sequence

```c
void Matrix( NLCtx *nlP, double *x );
```

where `x` is the current iterate. Alternatively, `NLSlesSetRoutines` can be used to specify the matrix evaluation routine as well set the default SLES routines for preconditioning and solving linear systems within the the Newton-like methods. The format of `NLSlesSetRoutines` is

```c
void NLSlesSetRoutines( NLCtx *nlP, void (*Matrix)(NLCtx *nlP, double *x) );
```

Further information about matrix formation is given in Chapter 6.

### 4.3.6 Optional Routines

Finally, commands of the form `NLSlesSet<XXX>` can optionally be employed to specify various routines, parameters, and performance monitors. For example, `NLSlesSetInitialGuess` sets the routine that determines the initial approximation used by the Newton-like methods. The format of `NLSlesSetInitialGuess` is

```c
void NLSlesSetInitialGuess( NLCtx *nlP, void (*Initial)(NLCtx *nlP, void *outvec) );
```

where the user-defined routine for forming the initial guess `outvec` is

```c
void Initial( NLCtx *nlP, void *outvec );
```

If no such routine is given, an initial approximation of zero is assumed.

Another useful routine is `NLSlesSetLog`, which has the form

```c
void NLSlesSetLog( NLCtx *nlP, FILE *file );
```
This routine indicates that verbose information regarding the solution process will be written to file, or to standard output in the absence of such a file.

The routine `NLSetSolution`, given by

```c
void NLSetSolution( NLCtx *nlP, void *x ),
```

can be used to set the location of the solution for the system to be solved. If no such location is indicated, then the appropriate space will be allocated by `NLSetUp`.

Refer to the manual pages and routine summaries for additional routines of the form `NLSet<XXX>`. Also see `tools.core/nonlin/snes/examples` and `tools.core/nonlin/sums/examples` for several example programs.

### 4.4 Altering the Default Parameters

All of the default parameters employed by the nonlinear solvers can be altered by the routine `NLSetParameter`, which has the format

```c
void NLSetParameter( NLCtx *nlP, char *par, double *value ),
```

where `par` is a character string indicating the parameter to be set, and `value` is the parameter value. A related routine that returns the current value of a particular parameter is

```c
double NLGetParameter( NLCtx *nlP, char *par ).
```

The manual pages for `NLSetParameter` and `NLGetParameter` list the parameters for each method, along with a brief description of their use.

### 4.5 Solving Several Nonlinear Systems

Often, the solution of a nonlinear system arises within the context of a larger problem, such as in the time-dependent case. In this instance, the recommend use of the SNES package involves creating the nonlinear context a single time and then using it repeatedly to solve successive nonlinear systems. This situation is illustrated with the following pseudocode fragment:

```c
NLCtx *nlP;
NLMETHOD nlmethen = NLE_NLS1;
int iters;

nlP = NLCreat(e( nlmethen, 0 );
loop {
    NLSetUp( nlP );
    iters = NLSolve( nlP );
} NLDestroy( nlP );
```
Chapter 5

The Linear System Solver

By default, the nonlinear solvers use the SLES (Simplified Linear Equation Solvers) [6] package to perform preconditioning and to solve the linear systems that arise during the nonlinear solution process. The SLES package provides easy access to a variety of direct and iterative linear solvers as well as preconditioners, so that the SNES/SLES combination facilitates experimentation. Bypassing the SLES module and instead working directly with the linear solvers themselves through the PETSc interface affords even greater control to the experienced user. For example, the KSP (Krylov Space Methods) package [7] of iterative methods can be accessed directly.

5.1 The Linear Solver Context

A context containing all relevant information about linear solves and preconditioning can be embedded within the nonlinear context by using the routine NLSetLinearSolverCtx. As shown below by the variable svctx, this context can take any form desired by the user, since it is implemented as a void pointer within the NLCtx:

```c
void NLSetLinearSolverCtx( NLCtx *nlP, void *svctx, int is_sles );
```

Here the flag is_sles should be zero if the linear solver context is not the SLES context; otherwise is_sles should be nonzero. The linear solver context can be extracted from the NLCtx with NLGetLinearSolverCtx and then manipulated by the user. This routine has the format

```c
void *NLGetLinearSolverCtx( NLCtx *nlP );
```

5.1.1 The SLES Context

When using the SLES package within SNES, the routine NLSlesSVCreate can be used instead of NLSetLinearSolverCtx to create the SLES context svctx (if necessary) and set it within the nonlinear context. As an analogue to the SLES routine SVCreate, this routine has the format

```c
NLSlesSVCreate( NLCtx *nlP, SpMat *mat, SVMETHOD method );
```

where mat is the Jacobian (or Hessian) matrix and method is the linear solver (preconditioner or direct method). The currently available SLES methods are listed below

- **SVLU** Direct (possibly sparse) factorization
- **SVNOPRE** No preconditioning. This method and all that follow it are preconditioners for an iterative method.
SVJacobi  Jacobi preconditioning
SVSSOR  SSOR preconditioning
SVILU  Incomplete factorization
SVICC  Incomplete Cholesky factorization, variant 1
SVICCJP  Incomplete Cholesky factorization, variant 2 [9]
SVBDD  Block diagonal
SVOSM  Overlapping Schwarz (additive and multiplicative)

5.1.2 Manipulating the SLES Context

Several other routines of the form NLSlesSV<XXX> handle basic manipulations of the SVctx within the nonlinear context, including default customizations for the nonlinear solvers:

- NLSlesSVSetUp - set up SLES method
- NLSlesSVSetOperators - set up SLES method (alternative)
- NLSlesSVSolve - execute SLES method
- NLSlesSVDestroy - free SLES data structures (if necessary)
- NLSlesSVSetConvergenceTest - set default convergence test
- NLSlesSVSetMonitor - set default monitor routine

Most other work with SLES should be done by first extracting the SVctx from the nonlinear context with NLGetLinearSolverCtx and then directly using the SLES package. Routines of the form NLSles<XXX> provide a default interface to the SLES package. Refer to the manual pages and the SLES Users Manual [6] for additional information.

5.2 Inexact Newton-like Methods

Since exact solution of the linear systems within (3.1) and (3.4) at each iteration can be costly, modifications are often introduced that significantly reduce these expenses and yet retain the rapid convergence of Newton's method. Inexact or truncated Newton techniques approximately solve the linear systems using an iterative scheme. In comparison with using direct methods for solving the Newton systems, iterative methods have the virtue of requiring little space for matrix storage and potentially saving significant computational work. Within the class of inexact Newton methods, of particular interest are Newton-Krylov methods, where the subsidiary iterative technique for solving the Newton system is chosen from the class of Krylov subspace projection methods.
5.2.1 Choosing the Convergence Strategy

Two levels of iterations occur for the inexact techniques, where during each global or outer Newton iteration a sequence of subsidiary inner iterations of a linear solver is performed. Appropriate control of the accuracy to which the subsidiary iterative method solves the Newton system (3.1) or (3.4) at each global iteration is critical, since these inner iterations determine the asymptotic convergence rate for inexact Newton techniques. While the Newton systems must be solved well enough to retain fast local convergence of the Newton’s iterates, use of excessive inner iterations, particularly when \( \| x_k - x^* \| \) is large, is neither necessary nor economical. Thus, the number of required inner iterations typically increases as the Newton process progresses, so that the truncated iterates approach the true Newton iterates.

A sequence of nonnegative numbers \( \{ \eta_k \} \) can be used to indicate the variable convergence criterion [2]. In this case, when solving a system of nonlinear equations, the update step of the Newton process remains unchanged, and direct solution of the linear system is replaced by iteration on the system until the residuals satisfy

\[
\tau_k^{(i)} = F'(x_k)\Delta x_k + F(x_k) \tag{5.1}
\]

Here \( x_0 \) is an initial approximation of the solution, and \( \| \cdot \| \) denotes an arbitrary norm in \( \mathbb{R}^n \).

The inexact Newton iterates are locally convergent if \( F' \) is uniformly continuous and nonsingular, and (5.2) holds [2]. Moreover, the convergence rate is linear in the sense that

\[
\| F'(x_*) (x_{k+1} - x_*) \| \leq \eta \| F'(x_*) (x_k - x_*) \|. \tag{5.2}
\]

Assuming that the truncated Newton iterates \( \{ x_k \} \) converge to \( x_* \), the convergence rate is \( Q \)-superlinear if and only if

\[
\limsup_{k \to \infty} \frac{\| r_k \|}{\| F(x_k) \|} = 0, \tag{5.3}
\]

where \( r_k \) is the final linear residual at the \( k^{th} \) outer iteration [2]. In addition, the inexact Newton iterates converge \( Q \)-quadratically if and only if

\[
\limsup_{k \to \infty} \frac{\| r_k \|}{\| F(x_k) \|^2} < \infty. \tag{5.4}
\]

Thus, to attain \( Q \)-quadratic convergence, (5.3) suggests choosing

\[
\eta_k = c \| F(x_k) \| \tag{5.4}
\]

for some nonnegative constant \( c \), where \( c < \| F(x_k) \|^{-1} \) to ensure local convergence.

As determined by (5.2) and (5.4), the following convergence criterion promotes quadratic convergence of the Newton-like methods:

\[
\| r_k \| < \hat{\eta}_k = \max \{ \tilde{\eta}_{min}, c \| F(x_k) \|^2 \}, \tag{5.5}
\]

where \( \tilde{\eta}_{min} \) is the minimum allowable convergence criterion for the linear iterations for a particular problem. In many cases various approximations, such as estimation of the Jacobian matrix, may prohibit quadratic convergence anyway, so that this criterion may become excessively stringent. Thus, the following alternative is often preferable:

\[
\| r_k \| < \hat{\eta}_k = \max \{ \tilde{\eta}_{min}, c \| F(x_k) \| \}. \tag{5.6}
\]
5.2.2 Krylov Methods within PETSc

The KSP package [7] within the PETSc library provides a variety of Krylov techniques that are easily employed in conjunction with the preconditioners of the SLES package. In addition, experienced users can bypass the SLES module and work directly with the KSP techniques.

The KSP package currently supports the following methods:

- **ITCG**: Conjugate gradient
- **ITCGS**: Conjugate gradient squared
- **ITBCGS**: BiCG-Stab
- **ITTFQMR**: Freund’s transpose-free QMR
- **ITGMRES**: Generalized minimum residual
- **ITTCQMR**: Chan’s transpose-free QMR
- **ITCHEBYCHEV**: Chebychev
- **ITRICHARDSON**: Richardson

As shown in Figure 7.1, the user should set the iterative method within the SVctx with the routine `SVSetAccelerator` before calling `NLSlesSVSetUp`. The form of `SVSetAccelerator` is

```c
void SVSetAccelerator(SVctx *svctx, ITMETHOD type),
```

where `type` denotes one of the iterative methods listed above.

By default the SNES/SLES Newton-like schemes used the convergence test (5.6), as implemented by `NLKSPDefaultConverged`. The routine `NLKSPQuadraticConverged` provides the alternative test given by (5.5).
Chapter 6
Matrix Evaluation

The matrix data structures of PETSc provide a variety of storage schemes for Jacobian and Hessian matrices, including two variants of compressed sparse row storage and dense storage by columns, as in Fortran. Additional matrix data structures are available (diagonal, block diagonal, block row, distributed sparse row), although these currently support only a limited number of matrix operations. Use of the PETSc data structures is recommended, since the user is then freed from the concern of manipulating matrix data. However, since the SNES package is completely data-structure-neutral, the user can choose to handle this aspect of the solution process.

6.1 The Matrix Context

The routine \texttt{NLSetMatrixCtx} sets the matrix context within the nonlinear context. As shown below, the matrix context, \texttt{matrix}, is implemented as a void pointer so that it can be completely defined by the user:

\begin{verbatim}
void NLSetMatrixCtx( NLCtx *nlP, void *matrix, int is_spmat ).
\end{verbatim}

Here \texttt{is_spmat} is a flag that indicates whether the matrix has the PETSc matrix format \texttt{SpMat}. The matrix context can be extracted from the nonlinear context with \texttt{NLGetMatrixCtx}, as given by

\begin{verbatim}
void *NLGetMatrixCtx( NLCtx *nlP ),
\end{verbatim}

and then manipulated by the user (for example, to assemble the matrix).

6.2 The PETSc Matrix Context

The PETSc matrix context is \texttt{SpMat}, and the SNES defaults for manipulating this context have the form \texttt{NLSpMat<XXX>}. The recommended routines for matrix creation within SNES are \texttt{NLSpMatCreateMatrix}, which automatically controls storage allocation, and \texttt{NLSpMatCreateMatrixFromData}, which enables the user to allocate matrix data. These routines have the format

\begin{verbatim}
NLSpMatCreateMatrix<FromData>( nlP, x, type, rows, cols ),
\end{verbatim}

where \texttt{x} is the current iterate, \texttt{type} denotes the type of matrix format, \texttt{rows} is the number of rows, and \texttt{cols} is the number of columns. To create the \texttt{SpMat} context, these routines employ the
matrix package within PETSc, where they use \texttt{Sp<format>Create} for the given matrix format. The routines also set the \texttt{SpMat} context within the nonlinear context (so that \texttt{NLSetMatrixCtx} need not be used in this instance) and form the matrix with the user-defined routine that has been set by \texttt{NLSetMatrixRoutine} or \texttt{NLSlesSetRoutines}. Refer to Section 7.1 for information about creating the matrix context with the SLES default routines.

6.2.1 Matrix Formats

Currently the PETSc package includes the following matrix types

- **MATROW**: Compressed row oriented storage
- **MATAIJ**: Conventional AIJ compressed row storage
- **MATDENSE**: Dense storage by columns

The \texttt{MATROW} default format should be employed by most users. The \texttt{MATAIJ} and \texttt{MATDENSE} formats are particularly useful when Fortran routines are used to generate the matrix, since they allow the user to control the allocation of storage space. Additional formats are available but are currently only partially supported within the PETSc package

- **MATBLOCK**: Compressed block row storage
- **MATDIAG**: Diagonal storage
- **MATBDIAG**: Block diagonal storage
- **MATROWDIST**: Matrix distributed by rows across processors

6.2.2 Assembling the Matrix

To assemble the matrix using the PETSc formats, first extract the \texttt{SpMat} context from the nonlinear context with \texttt{NLGetMatrixCtx}. Nonzero values can be added to the matrix one at a time with \texttt{SpAddValue}, by rows with \texttt{SpAddInRow}, or by dense blocks with \texttt{SpAddDenseSubmatrix}. A single value may be set with the routine \texttt{SpSetValue}. Note that matrices created by all versions of \texttt{Sp<format>Create} are indexed with zero origins, so that the rows range from zero through \texttt{rows-1}, while the columns range from zero through \texttt{cols-1}. Refer to Chapter 3 of the \textit{SLES Users Manual} for the format details of these routines.

Additional routines can be used to streamline the matrix assembly process with \texttt{NLSpMatCreateMatrix}. For example, \texttt{NLSpMatSetEstRowElements} enables the user to set the estimated maximum number of nonzeros in each row of the matrix, so that this space can be preallocated for the \texttt{MATROW} format, thereby accelerating the process of adding elements to the matrix. Note that this parameter does \textit{not} limit the number of nonzeros in each row of the matrix. In particular, the matrix will grow dynamically if the parameter is its default value of zero.

Routines to facilitate user allocation of matrix space are \texttt{NLSpMatSetAIJData}, \texttt{NLSpMatGetAIJData}, \texttt{NLSpMatSetDenseData}, and \texttt{NLSpMatGetDenseData}.
Chapter 7

Creating Custom Methods

As described in Section 4.3.4, the modular design of SNES partitions each global step of the Newton-like methods into three basic stages, labeled *step-setup*, *step-compute*, and *step-destroy*. We discuss in detail these phases, which facilitate the creation of customized nonlinear solution techniques.

7.1 Step-Setup Phase

The step-setup stage, which can optionally be executed a single time before one or more step-compute phases, typically consists of evaluating the Jacobian (or Hessian) matrix and initializing the step-compute process. The following routine is used to specify this phase:

\[
\text{NLSetStepSetUp( NLCtx *nlP, void (*setup)( ),}
\]

where \text{setup} is a routine that executes the setup process. The calling sequence of the setup routine is

\[
\text{void setup( NLCtx *nlP, void *invector ),}
\]

where \text{invector} is the current iterate.

A simple setup routine for a basic linear step within SNES is given in Figure 7.1 and can serve as a guide for the development of customized routines. This routine employs \text{NLSpMatCreateMatrix} for matrix creation and thereby assumes that the matrix is stored with one of the PETSc formats.

The uniprocessor default SLES step-setup routine is \text{NLSlesSetUp}, which automatically calls \text{NLSpMatCreateMatrix} for matrix creation. Thus, when using this SLES routine, the user must merely set the matrix size with \text{NLSpMatSetRows} and \text{NLSpMatSetCols} before calling \text{NLSlesSetUp}. As shown in the example program of Figure 2.2, this initialization usually occurs before initiating the solution process with \text{NLSolve}. These routines have the format

\[
\text{void NLSpMatSetRows( NLCtx *nlP, int rows )}
\]
\[
\text{void NLSpMatSetCols( NLCtx *nlP, int cols ),}
\]

where \text{nrows} and \text{ncols}, respectively, indicate the numbers of rows and columns of the matrix. Likewise, \text{NLSpMatSetType} can be used to override the default compressed sparse row \text{SpMat} data structure, \text{MATROW}.

By default, \text{NLSpMatCreateMatrix} internally allocates matrix space assuming use of one of the \text{SpMat} formats. The user can choose to control matrix allocation for the matrix types \text{MATDENSE}, \text{MATAIJ}, and \text{MATBDIAG} by calling \text{NLSpMatSetUserAllocate} once at the program’s outset, as shown in the program of Figure 2.1. In this case, \text{NLSpMatCreateMatrixFromData} will be employed.
void UserStepSetUp( n1P, x )
NLCtx *n1P;
void *x;
{
  SVctx *svctx; /* SLES context */
  SpMat *amat; /* matrix context */
  ITMETHOD itmethod = ITGMRES; /* iterative method */
  SVMETHOD svmethod = SVBDD; /* preconditioner or direct solver */

  /* Create the Jacobian (or Hessian) matrix */
  NLSpMatCreateMatrix( n1P, x, NLSpMatGetType(n1P),
                      NLSpMatGetRows(n1P), NLSpMatGetCols(n1P) );

  /* Create the linear solver context */
  amat = (SpMat*)NLGetMatrixCtx(n1P);
  svctx = NLSlesSVCreate( n1P, amat, svmethod );

  /* Set the iterative method (optional) */
  SVSetAccelerator( svctx, itmethod );

  /* Initialize the linear solver */
  NLSlesSVSetUp( n1P );

  /* Optional routines to customize linear solver */
  SVSetMonitor( svctx, ITDefaultMonitor, (void *)0 );
  SVSetRelativeTol( svctx, .000001 );
  SVSetIts( svctx, 100 );
}

Figure 7.1: Sample Step-Setup Routine

7.2 Step-Compute Phase

The step-compute phase is the core of the solution process, and it is executed one or more times
during each iteration of the Newton-like schemes to compute prospective steps. The following
routine can be used to set this phase:

\[ \text{NLSetStepCompute( NLCtx \texttt{*n1P}, \texttt{void (*compute)()}, } \]

where compute is a routine that computes the next step to try. Note that the step-compute routine
must return the norm of the step and a predicted value of the function at the new iterate. The
calling sequence is

\[ \text{fun (NLCtx \texttt{*n1P}, void \texttt{*x}, void \texttt{*fg}, void \texttt{*y}, double \texttt{*fgnorm}, double \texttt{*ymax}, double \texttt{*rtol}, double \texttt{*gpnorm}, double \texttt{*ynorm}, void \texttt{*misc })} \]

where the function input arguments are

- \texttt{x} - current iterate,
- \texttt{fg} - residual (+/-) or gradient evaluated at \texttt{x},
- \texttt{fgnorm} - 2-norm of \texttt{fg},
- \texttt{ymax} - maximum allowed 2-norm of step (trust region size),
- rtol - truncation tolerance, and
- misc - miscellaneous input/output,

and the function output arguments are
- y - the resulting step vector,
- ynorm - 2-norm of the step,
- gpnorm - predicted value of the function at the new iterate, assuming local linearization, and
- misc - miscellaneous input/output.

Figure 7.2 illustrates a sample step-compute routine that contains the core portions of NLSlesSolveScale, the default uniprocessor routine for a basic linear step in solving a nonlinear system of equations. This routine solves

\[ M(x)y = -F(x), \text{ where } \|y\|_2 \leq \Delta. \]

Here \( \Delta \) is a positive real number, and \( M(x) \) is an approximation of the Jacobian matrix. For trust region methods \( \Delta \) is the trust region parameter; for other Newton-like methods \( \Delta \) is set to be the maximum allowable step length. Note that the routine NLScaleStep is used to scale the step, while the routine NLSlesSVSolve provides an interface to the SLES package for the linear system solution.

```c
int UserStepCompute( nsP, x, f, fnorm, delta, rtol, gpnorm, 
                     ynorm, misc )
NLCtx *nsP;
void *f, *x, *y, *misc;
{
    int its;

    /* Solve linear system */
    its = NLSlesSVSolve( nsP, f, y );

    /* Scale step so its length is less than delta */
    NLSScaleStep( nsP, f, fnorm, delta, gpnorm, ynorm );
    return its;
}
```

Figure 7.2: Sample Step-Compute Routine

### 7.3 Step-Destroy Phase

Finally, near the conclusion of a global iteration, the step-destroy phase can perform such tasks as releasing memory used by the Jacobian (or Hessian) matrix and the linear solver. The optional step-destroy routine is called once after completing a series of step-compute procedures, but before issuing another call of the step-setup routine. NLSSetStepDestroy should be used to specify this phase:

```c
NLSSetStepDestroy( NLCtx *nsP, void (*destroy)() );
```
where the calling sequence of the destroy routine is

```c
void destroy( NLCtx *nlP );
```

The default uniprocessor routine for freeing the memory used for a basic linear step with the SLES package is `NLSlesSVDestroy`, which by default destroys both the `SVctx` and `SpMat` contexts with `SVDestroy` and `SpDestroy`, respectively.

Retaining the matrix and/or linear solver contexts between successive Newton iterations can reduce unnecessary repeated work when solving systems with the same size and same preconditioning method. Savings are especially evident for the multiprocessor version when the nonzero matrix structure remains unchanged, so that preliminary organizational work can be done a single time, thereby being amortized over the course of the total solution process. The user can indicate that either of these contexts should be saved between Newton iterations by calling `NLSlesSetSaveSolverCtx` and/or `NLSpMatSetSaveMatrixCtx` prior to calling `NLSlesSVDestroy`.

When saving the preconditioning and/or linear system matrix between iterations, the routine `NLSlesSVSetOperators` should be used instead of the usual `NLSlesSVSetUp`. Note, however, that the current PETSc version does not fully support `NLSlesSVSetOperators` for all SLES techniques. See the manual pages for further information.

### 7.4 Alternative Steps

Although the previous examples employ a linear step, the user can employ alternative step types. In particular, the trust region methods `NLE_NTR2_LIN` and `NLE_NTR2_DOG` use the same basic nonlinear solution routine with a linear and a dogleg step, respectively. As mentioned in Section 4.3.1, additional routines must be specified for the use of these techniques. This information is automatically set with `NLSlesSetRoutines`; however, if this routine is not used, then either `NLStepLinearSetRoutines` or `NLStepDoglegSetRoutines` should be called prior to the `NLSetUp` phase.

#### 7.4.1 Linear Step

The routine `NLStepLinearSetRoutines`, which has the format

```c
NLStepLinearSetRoutines( NLCtx* nlP, void (*ScaleM)(), void (*Mv)() ),
```

should be used in conjunction with `NLE_NTR2_LIN` to indicate a linear search step. `ScaleM` forms a diagonal scaling matrix, `D`, and has the format

```c
ScaleM( void *nlP, double *D );
```

The routine `NLSpMatScaleM` forms a scaling matrix of slightly modified $L_2$ row norms and is provided for use with PETSc data structures. The routine `Mv`, which has the format

```c
Mv( NLCtx *nlP, *double invector, *double outvector ),
```

computes the matrix-vector product. The routine `NLSpMatMv` forms this product for a matrix having the `SpMat` format.
7.4.2 Dogleg Step

The routine `NLStepDoglegSetRoutines`, which has the format

```c
NLStepDoglegSetRoutines( NLCtx *nlP, void (*SetUpSolveM)(), void (*ScaleM)(),
                       void (*MTv)(), void (*Mv)(), int (*SolveM)(), void (*DestroyM)() ),
```

should be used with the method `NLE_NTR2_DOG` to specify a dogleg step. The following routines are needed for this type of step:

- **SetUpSolveM**( NLCtx *nlP, double *x )
  set up a linear solver, where x is the current iterate

- **ScaleM**( void *nlP, double *D )
  form diagonal scaling matrix, D

- **MTv**( NLCtx *nlP, *double invector, *double outvector )
  compute product of matrix transpose and vector

- **Mv**( NLCtx *nlP, *double invector, *double outvector )
  compute matrix-vector product

- **SolveM**( NLCtx *nlP, double *f, double *g )
  solve a linear system, where f is the right-hand-side vector and g is the solution vector

- **DestroyM**( NLCtx *nlP )
  destroy the linear solver

As indicated below, default functions are provided for use with the SLES package and PETSc data structures

- **SetUpSolveM** - NLSlesSetUp
- **ScaleM** - NLSMatScaleM
- **MTv** - NLSMatMTv
- **Mv** - NLSMatMv
- **SolveM** - NLSlesSVSolve
- **DestroyM** - NLSlesSVDestroy
Chapter 8

Monitoring Performance

The SNES package includes provisions for counting the floating-point operations and calls to various routines, as well as for monitoring timing and memory usage. The routine NLGetStatistics writes performance statistics, including operation counts and times for the various phases of the solution process. NLOutput writes information regarding the solver's overall performance. For a summary of the total solution process, the user may call either of these routines after NLSolve but before NLDestroy.

Within all of the SNES solvers, an optional monitoring routine can be called after successfully completing the step-compute phase (discussed in Section 7.2). This routine can be set before calling NLSolve with the following routine:

```c
void NLSetMonitor( NLCtx *nlP, void (*monitor)(1, void *mctx),
```

where `monitor` is the monitoring function and `mctx` is a context for private data for the monitoring (may be null).

### 8.1 Systems of Nonlinear Equations

When solving a system of nonlinear equations, the monitoring routine has the format

```c
void monitor(NLCtx *nlP, void *x, void *f, double *fnorm),
```

where `x` is the current iterate, `f` is the current residual (either positive or negative, as set by the user with `NLSetResidualRoutine`), and `fnorm` is the $L_2$ norm of the residual. The monitoring routines provided for solving systems of nonlinear equations are

- NLENewtonDefaultMonitor - default, prints the residual norm; and
- NLENewtonStatisticsMonitor - prints the residual norm and calls NLGetStatistics.

### 8.2 Unconstrained Minimization Problems

When solving an unconstrained minimization problem with SUMS, the monitoring routine has the format

```c
void monitor( NLCtx *nlP, void *x, void *g, double *f, double *gnorm ),
```

where `x` is the current iterate, `g` is the gradient, `f` is the function, and `gnorm` is the $L_2$ norm of the gradient. The monitoring routines provided for SUMS are
• **NLMNewtonDefaultMonitor** - default, prints the function value and gradient norm; and

• **NLMNewtonStatisticsMonitor** - prints the function value and gradient norm, calls NLGetStatistics.

### 8.3 Additional Information

X Window displays of the residual norm as well as solution and residual contours are also available, as described in the manual pages. In addition, see the manual pages for further information about the use of NLGetStatistics, including setting the operation counts for user-defined phases of computation.
Chapter 9

Method Registry

The SNES registry permits the user to customize the suite of solution methods that are loaded into the software library. In particular, the registry allows the user to add new methods to the package without altering the existing SNES code. This feature facilitates the use of newly developed algorithms and implementations within the context of the PETSc package, so that the user's code does not require significant modifications. In addition, the user can exclude from the library certain methods deemed unnecessary for a particular application.

While the following information pertains to the SNES package, the SLES Users Manual provides related instructions regarding customizing the suites of linear solvers, iterative methods, matrix orderings, and sparse matrix formats.

9.1 Adding New Methods

The SNES registry, which is regulated by the routine NLRegister, is the means by which new methods can augment the SNES package. As indicated below, a particular method can be added to the pool of available solvers by registering its associated creation routine:

\[
\text{NLRegister}( \text{int id, char *name, void *(*routine)()} ),
\]

where id is a unique identifier, name is a string that names the method, and routine is a routine that creates the method. After this routine has been called, NLGetMethod will recognize name as a known nonlinear solver method and return id and the corresponding solver's type.

The creation routine for a particular solver method typically allocates memory associated with the method and sets default parameters and intermediate routines. Various creation routines within the SNES package can serve as templates for new methods. For example, see NLENewtonTRICreate, which is located in the file ‘tools.core/nonlin/snes/newtr1/newtr1.[ch]’, for the creation routine associated with the trust region method NLE_NTR1.

9.2 Restricting the Choices

As discussed in the SLES Users Manual, one disadvantage of providing a wide variety of methods, all of which are available at run time, is that the executable image can become considerably large. Although this situation may not pose difficulties for virtual-memory machines, problems can develop for massively parallel architectures, where each processor has a copy of the executable program. The PETSc registration process allows the user to determine which routines are loaded during the creation of the executable, and thus conserve memory space.
The default registration for the SNES package is determined by the routine NLRegisterAll, which is located in the file ‘tools.core/nonlin/nldregis.c’. To alter this default, the user must merely write another routine called NLRegisterAll, which contains NLRegister calls for each of the desired methods, and link it to the user program before the PETSc library. For example, the routine below restricts the available choices of the SNES executable to only the line search method NLE-NLS1 and the trust region method NLE-NTR1.

```c
void NLRegisterAll()
{
    NLRegister(NLE-NLS1, "enls1", NLENewtonLS1Create);
    NLRegister(NLE-NTR1, "entr1", NLENewtonTR1Create);
}
```
Chapter 10

The Multiprocessor Version of SNES

The foundation for the multiprocessor version of SNES is Chameleon [5], a highly portable and easy-to-use message-passing system that facilitates use of diverse distributed-memory architectures. As part of the multilevel PETSc library, much of the support provided by Chameleon is embedded in the library routines of the parallel versions of the SNES and SLES packages, so that a detailed knowledge of the communication package is not required. The user is responsible only for subdividing the problem domain among the processor group and evaluating the residual and its associated Jacobian matrix (or function, gradient, and Hessian matrix). The necessary communication for these tasks can be performed with either Chameleon or BlockComm, a higher level package within PETSc that manages repeated communication patterns.

At this time the multiprocessor version SNES is undergoing refinement and extension, as we work toward a common interface for the uniprocessor and parallel cases. Interprocessor communication within the new parallel version will be handled with the MPI (Message Passing Interface) standard. Thus, forthcoming versions of the software will contain additional features, and changes to parallel interface may result at any time. Refer to the examples within the package for additional information.
Appendix A

Installing SNES

The uniprocessor version SNES can be obtained as part of the PETSc library by anonymous ftp from 'info.mcs.anl.gov' in the directory 'pub/petsc'. The code and supporting manual page information for the entire PETSc library are contained in the compressed tar file 'tools.<version>.tar.Z', where <version> indicates the particular software version. The postscript file corresponding to this document is 'snes.ps', while the SLES Users Manual and the KSP Users Manual are respectively contained in the postscript files 'sles.ps' and 'ksp.ps', respectively.

To install the PETSc package, transfer and then uncompress the desired tar file(s). Then extract the archived files by typing

```
tar xf tools.<version>.tar
```

so that a directory 'tools.core' will be created as a subdirectory of the current directory.

The GNU configure approach is the recommended means of installation, as controlled by the script in the PETSc root directory, 'tools.core/configure'. A simple installation of the debugging version can be produced by doing

```
configure -bopt=g.
```

See the file 'tools.core/readme' for detailed instructions, including options for system-specific installation and generation of various library versions. Documentation for the entire package can then be generated by the command

```
make manpages.
```

Likewise, the uniprocessor SNES library (and all lower level PETSc components) can be built with the command

```
make snes.
```
Appendix B

Library Organization

SNES maintains different libraries for various levels of optimization, as indicated by <bopt>:

- \(<\text{bopt}> = \text{g}:\) code development, debugging
- \(<\text{bopt}> = \text{pg}:\) program profiling
- \(<\text{bopt}} = \text{0:}\) optimized production runs.

In addition, a single filesystem can support libraries for different architectures, as indicated by \(<\text{arch}>\) (e.g., \(<\text{arch}> = \text{sun4}\) or \(<\text{arch}> = \text{rs6000}\)). The PETSc libraries for particular choices of \(<\text{bopt}>\) and \(<\text{arch}>\) are located in the directory

\text{tools.core/libs/libs<bopt>/<arch>}

For example, the libraries for optimized production runs for the Sun4 architecture are in the directory

\text{tools.core/libs/libs0/sun4}.

As illustrated in Figure 1.1, the PETSc libraries are subdivided by components, which are listed below for the uniprocessor version in roughly increasing levels of abstraction. The SNES package is contained in \text{libsnes.a}, which is built on lower-level PETSc modules.

- \text{libpecore.a} - system portability
- \text{libpetsc.a} - vectors and miscellaneous routines
- \text{libpemat.a} - matrices
- \text{libpetscgrf.a} - graphics
- \text{libbiter.a} - Krylov space methods
- \text{libsles.a} - linear equations solvers
- \text{libsnes.a} - nonlinear equations solvers
- \text{domain.a} - PDE solvers

B.1 Linking

The order in which the libraries are linked (from higher-level modules to lower-level modules) is critically important for successful linking. The following example indicates the linkage order for the uniprocessor SNES and SUMS libraries:
LIBS = -L$(LDIR) -lsnes -lsles -liter -lpemat -lpetsc -lpetscgrf -lpecore
     $(LAPACK) $(BLAS) $(XllLIB) $(VLIBS) $(FCLIB) -lm $(SLIB).

Here the variable $LDIR$ indicates the PETSc library directory for a particular architecture and level of optimization. The above variables are defined during the PETSc installation process as follows:

- **BLAS** - BLAS library
- **LAPACK** - LAPACK library
- **SLIB** - any required system libraries
- **XllLIB** - Xll window system library
- **FCLIB** - libraries for a C linker to use code containing Fortran code.

If PETSc graphical displays are not employed, then ‘libperscgrf.a’ and ‘XllLIB’ can be omitted.

A partial makefile for the optimized rs6000 version of the uniprocessor C program ‘example1’ is given below to illustrate proper usage of the library routines. Here CLINKER denotes the linker to use for C main programs.

```
IToolSdir = /usr/local/tools.core
ARCH = rs6000
BOPT = 0
LDIR = $(IToolSdir)/libs/libs$(BASEOPT)$(PROFILE)/$(ARCH)
LIBS = -L$(LDIR) -lsnes -lsles -liter -lpemat -lpetsc -lpetscgrf -lpecore
     $(LAPACK) $(BLAS) $(XllLIB) $(VLIBS) $(FCLIB) -lm $(SLIB)

example1: example1.o
         $(CLINKER) -o example1 example1.o $(LIBS)
```

### B.2 Makefiles

The makefile system of the PETSc package enables a single makefile to build libraries and programs on various architectures. A few variables define the architecture, optimization level, compiler flags, and system libraries and thus determine the particular options used by the makefile. Refer to the manual page entitled "ToolsMake" for additional details, and see the examples in the directories ‘tools.core/nonlin/snes/examples’ and ‘tools.core/nonlin/sums/examples’.

### B.3 Using SNES with Fortran

SNES includes a fully functional Fortran interface, thus allowing Fortran users to employ virtually all routines within the package. The usual Fortran conventions are upheld, so that the user need not be concerned about the issue of pointers versus values. The procedures for using the C and Fortran versions of SNES are very similar, so that Fortran users must be aware of only a few additional guidelines.

A few SNES routines are unavailable to Fortran users because of restrictions inherent in the language itself. For instance, the C routine **NLGetMethod** has no Fortran analogue, since no portable access to the command line exists.

Parameters used by the nonlinear solvers are defined in the include file ‘tools.core/nonlin/nlfort.h’, which must be positioned at the start of a Fortran program. All
context variables within SNES and the rest of the PETSc package should be declared as integers; the PETSc interface then provides the appropriate link to the context’s structure.

The Fortran interface files to the PETSc library are automatically generated from the C program files. The command

```
make fort_iface
```

compiles the uniprocessor Fortran interface library, ‘LDIR/libpetscfort.a’, where ‘LDIR’ denotes the PETSc library directory. This library must precede the PETSc libraries in the linking sequence for all Fortran programs. Otherwise, the linking sequence, as described in Section B.1, remains unchanged.

A partial makefile for the debugging sun4 version of the uniprocessor Fortran program 'fort1' is given below, where FLINKER denotes the linker to use for Fortran main programs.

```
ITOOLSDIR = /usr/local/tools.core
ARCH = sun4
B0PT = g
LDIR = $(ITOOLSDIR)/libs/$(BASEOPT)/$(PROFILE)/$(ARCH)
LIBS = -L$(LDIR) -lsnes -lsles -liter -lpemat -lpetsc -lpetscgrf -lpecore
      $(LAPACK) $(BLAS) $(XMLIB) $(VLIBS) $(FCLIB) -lm $(SLIB)
FLIBS = -L$(LDIR) -lpetscfort

fort1: fort1.o
      $(FLINKER) -o fort1 example1.o $(FLIBS) $(LIBS)
```
Appendix C

Summary of Routines

This appendix consists of a brief summary of the routines that constitute the SNES package. The most basic routines are listed in Section C.1. Sections C.2 and C.3 contain routines for the SLES and SpMat interfaces, respectively. Finally, Section C.4 lists a variety of other miscellaneous routines. Refer to the manual pages for complete details.

The following include files are required for using the SNES package with C.

```c
#include "tools.h"
#include "nonlin/nlall.h"
```

Likewise, the file `nonlin/nlfort.h` is necessary for using the Fortran version of the SNES package.

C.1 Core Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void NLCreatDBVectors(NLCtx *nlP, int n)</code></td>
<td>Creates a vector context within the nonlinear context for BLAS based double-precision serial vectors.</td>
</tr>
<tr>
<td><code>void NLCreatDVectors(NLCtx *nlP, int n)</code></td>
<td>Creates a vector context within the nonlinear context for double-precision serial vectors.</td>
</tr>
<tr>
<td><code>NLCtx *NLCreat(NLMETHOD nlmethod, void *usrP)</code></td>
<td>Builds a NLCtx structure for a particular nonlinear solver; this structure will then be used in solving systems of nonlinear equations or unconstrained minimization problems.</td>
</tr>
<tr>
<td><code>void NLDestroy(NLCtx *nlP)</code></td>
<td>Destroys the nonlinear solver context, NLCtx.</td>
</tr>
<tr>
<td><code>double *NLLGetFunction(NLCtx *nlP)</code></td>
<td>Gets the objective function used within the NLM routines.</td>
</tr>
<tr>
<td><code>void *NLLGetGradient(NLCtx *nlP)</code></td>
<td>Gets the gradient vector used within NLM routines.</td>
</tr>
<tr>
<td><code>void *NLLGetResidual(NLCtx *nlP)</code></td>
<td>Gets the residual vector used within the NLE routines.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>void *NLGetSolution( NLCtx *nlP )</code></td>
<td>Gets the solution vector from the nonlinear context.</td>
</tr>
<tr>
<td><code>void *NLGetUserCtx( NLCtx *nlP )</code></td>
<td>Extracts the optional user-defined context from the nonlinear context.</td>
</tr>
<tr>
<td><code>int NLHelpMessage( int *argc, char **args )</code></td>
<td>Prints a help message for the SNES packages if &quot;-help&quot; is one of the command line options.</td>
</tr>
<tr>
<td><code>void NLSetFunctionRoutine( NLCtx *nlP, void(*func)() )</code></td>
<td>Sets the function evaluation routine for use by the NLM routines.</td>
</tr>
<tr>
<td><code>void NLSetGradientRoutine( NLCtx *nlP, void(*func)() )</code></td>
<td>Sets the gradient evaluation routine for use by the NLM routines.</td>
</tr>
<tr>
<td><code>void NLSetGradient( NLCtx *nlP, void *g )</code></td>
<td>Sets the location of the gradient vector for use by NLM routines. If no space is indicated, default space (with its dimension determined by the vector context) is allocated in NLSetUp().</td>
</tr>
<tr>
<td><code>void NLSetInitialGuessRoutine( NLCtx *nlP, void(*func)() )</code></td>
<td>Sets the function that calculates the initial guess for the nonlinear solver. If this function is not set, an initial guess of zero is used.</td>
</tr>
<tr>
<td><code>void NLSetMatrixRoutine( NLCtx *nlP, void(*func)() )</code></td>
<td>Sets the Jacobian (or Hessian) evaluation routine that will be called by NLFormMatrix().</td>
</tr>
<tr>
<td><code>void NLSetResidualRoutine( NLCtx *nlP, void(*func)(), int rneg )</code></td>
<td>Sets the residual evaluation routine for use by the NLE routines.</td>
</tr>
<tr>
<td><code>void NLSetResidual( NLCtx *nlP, void *f )</code></td>
<td>Sets the location of the residual vector for use by the NLE routines. If no space is indicated, default space (with its dimension determined by the vector context) is allocated in NLSetUp().</td>
</tr>
<tr>
<td><code>void NLSetSolution( NLCtx *nlP, void *x )</code></td>
<td>Sets the location of the solution vector for the system to be solved. If no space is indicated, default space (with its dimension determined by the vector context) is allocated in NLSetUp().</td>
</tr>
<tr>
<td><code>void NLSetStepCompute( NLCtx *nlP, int(*func)() )</code></td>
<td>Sets a user-provided routine that computes the next step to try.</td>
</tr>
<tr>
<td><code>void NLSetStepDestroy( NLCtx *nlP, void(*func)() )</code></td>
<td>A user-provided routine which is called after step-compute has been applied to several vectors, but before step-set-up is called for another point.</td>
</tr>
</tbody>
</table>
### void NLSetStepSetUp( NLCtx *nlP, void(*func)() )

Sets a user-provided routine which is called once before several calls to the step-compute routine.

### void NLSetUp( NLCtx *nlP )

Sets up the internal data structures for the later use of a nonlinear solver. Call NLSetUp() after calling NLCreatet), and optional routines of the form NLSetXXX(), but before calling NLSolve().

### void NLSlesSetRoutines( NLCtx *nlP, void(*GetM)() )

Sets the default SLES routines for the various nonlinear solvers.

### int NLSolve( NLCtx *nlP )

Solves a nonlinear system. Call NLSolve after calling NLCreatet), optional routines of the form NLSetXXX(), and NLSetUp().

## C.2 Interface Routines for SLES

### void NLSlesApplyBackwardSolve( NLCtx *nlP, void *y, void *x )

Applies a backward solve to a dense vector, where the upper triangular matrix is hidden in the NLCtx context.

### void NLSlesApplyForwardSolve( NLCtx *nlP, void *y, void *x )

Applies a forward solve to a dense vector, where the lower triangular matrix is hidden in the NLCtx context.

### ITCntx *NLSlesGetKSPCtx( NLCtx *nlP )

Extracts the Krylov iterative method context from the database of the nonlinear context.

### ITMETHOD NLSlesGetKSPMethod( NLCtx *nlP )

Extracts the Krylov space iterative method (accelerator) from the database of the nonlinear context.

### SVMETHOD NLSlesGetMethod( NLCtx *nlP )

Extracts the SLES method (preconditioner or direct solver) from the database of the nonlinear context.

### SVctx *NLSlesSVCreate( NLCtx *nlP, SpMat *matrix, SVMETHOD svmethod )

Creates the SLES context if it does not already exist and sets it as the linear solver within the nonlinear context. If the context already exists, then its pointer is merely returned.

### void NLSlesSVDestroy( NLCtx *nlP )

Destroys the SLES linear solver context, unless the user has indicated otherwise by first calling NLSlesSetSaveSolverCtx(). Also frees the Jacobian (or Hessian) matrix context, unless the user has indicated otherwise by first calling NLSpmSetSaveMatrixCtx().
| void NLSlesSVSetConvergenceTest( NLCtx *nlP, int(*converge)(), void *cctx ) |
| Sets the convergence test to be used by a SLES iterative linear solver within the SNES package. |
| void NLSlesSVSetMonitor( NLCtx *nlP, void(*monitor)(), void *cctx ) |
| Sets the routine within the SNES package that monitors the residual at each iteration of a SLES iterative method. |
| void NLSlesSVSetOperators( NLCtx *nlP, SpMat *amat, SpMat *pmat, int amat_has_same_nz, int pmat_has_same_nz ) |
| Sets up the SLES linear solver context within the nonlinear context, allowing user specification of the linear system matrix and/or the associated preconditioning matrix. Also imposes the default convergence test for the iterative linear solvers. Use NLSlesSVSetOperators() instead of NLSlesSVSetUp() when flexibility is needed in terms of matrix specification. |
| void NLSlesSVSetUp( NLCtx *nlP ) |
| Sets up the SLES linear solver context within the nonlinear context and imposes the default convergence test for the iterative linear solvers. |
| int NLSlesSVSolve( NLCtx *nlP, void *b, void *x ) |
| Calls a SLES routine to solve a linear system. |
| int NLSlesSaveSolverCtx( NLCtx *nlP ) |
| Returns a flag that indicates whether NLSlesSetSaveSolverCtx() has been called. |
| void NLSlesSetENTR2 - DOG( NLCtx *nlP, void(*GetM)() ) |
| Sets the SLES defaults for the NLN_TR2_DOG method (trust region scheme with a dogleg step). |
| void NLSlesSetENTR2 - LIN( NLCtx *nlP, void(*GetM)() ) |
| Sets the SLES defaults for the NLEXTR2_LIN method (trust region scheme with a linear step). |
| void NLSlesSetGeneralStep( NLCtx *nlP, void(*GetM)() ) |
| Sets the routine for forming the Jacobian (or Hessian) matrix. Also sets the SLES defaults for setting up, computing, and destroying a step. |
| void NLSlesSetKSPMethod( NLCtx *nlP, ITMETHOD method ) |
| Sets the Krylov space iterative method (accelerator) within the database of the nonlinear context. |
| void NLSlesSetMethod( NLCtx *nlP, SVMETHOD method ) |
| Sets the SLES method (preconditioner or direct solver) within the database of the nonlinear context. |
| void NLSlesSetRoutines( NLCtx *nlP, void(*GetM)() ) |
| Sets the default SLES routines for the various nonlinear solvers. |
void NLSpMatCreateMatrixFromData( NLCtx *nlP, void *x, int type, int rows, int cols )

Forms the Jacobian (or Hessian) matrix with the user routine that has been assigned with either NLSetMatrixRoutine() or NLSlesSetRoutines(), assuming that the user controls memory allocation. If the SpMat matrix context does not already exist, it is created. This option is valid only for the MATDENSE, MATAIJ, and MATBDIAG storage schemes.

void NLSpMatCreateMatrix( NLCtx *nlP, void *x, int type, int rows, int cols )

Forms the Jacobian (or Hessian) matrix with the user routine that has been assigned with either NLSetMatrixRoutine() or NLSlesSetRoutines(). If the SpMat matrix context does not already exist, it is created.

void NLSpMatGetAIJData( NLCtx *nlP, int *ia, int *ja, double *a )

Extracts the location of the MATAIJ data structures for the Jacobian (or Hessian) matrix from within the nonlinear context, assuming use of the SpMat matrix context.

void NLSpMatGetBDiagData( NLCtx *nlP, int *nd, *nb, *diag, double **diagv )

Extracts the location of the MATBDIAG data structures for the Jacobian (or Hessian) matrix from within the nonlinear context, assuming use of the SpMat matrix context.

void NLSpMatGetColNormsLp( NLCtx *nlP, int p, double *norm )

Provides an interface to the routine SpGetColNormsLp(), which computes the L-p norm of each column of the matrix, assuming storage with the SpMat format.

int NLSpMatGetCols( NLCtx *nlP )

Returns the number of columns of the Jacobian (or Hessian) matrix as set in the database of the nonlinear context.

NLSpMat *NLSpMatGetCtx( NLCtx *nlP )

Returns a pointer to the NLSpMat database context, which provides an interface between the nonlinear solvers and the SpMat matrix context.

C.3 Interface Routines for PETSc Matrix Data Structures

void NLSlesSetSaveSolverCtx( NLCtx *nlP )

Sets a flag to indicate that the SLES context should NOT be destroyed during the step-destroy phase. This can reduce the overhead for repeated linear solves, but should be used with caution.

void NLSlesSetUp( NLCtx *nlP, void *x )

Forms the Jacobian (or Hessian) matrix and sets up the SLES solver context.

int NLSlesSolveScale( NLCtx *nlP, void *x, void *f, void *y, double *fnorm, double *delta, double *rtol, double *gpnorm, double *ynorm, void *misc )

Solves the linear system M(x) y = -f, where || y || <= delta. For systems of nonlinear equations, M and f respectively denote the Jacobian matrix and the residual.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int NLSpMatGetDeclaredRows(NLCtx *nlP)</code></td>
<td>Returns the number of declared rows of the Jacobian (or Hessian) matrix as set in the database of the nonlinear context.</td>
</tr>
<tr>
<td><code>double *NLSpMatGetDenseData(NLCtx *nlP)</code></td>
<td>Extracts the location of the dense Jacobian (or Hessian) matrix elements from the nonlinear context, assuming use of the SpMat matrix context.</td>
</tr>
<tr>
<td><code>int NLSpMatGetEstRowElements(NLCtx *nlP)</code></td>
<td>Returns the estimated number of nonzero elements in each row of the Jacobian (or Hessian) matrix as set in the database of the nonlinear context.</td>
</tr>
<tr>
<td><code>int NLSpMatGetMaxNonzeros(NLCtx *nlP)</code></td>
<td>Returns the maximum number of nonzero elements of the Jacobian (or Hessian) matrix as set in the database of the nonlinear context.</td>
</tr>
<tr>
<td><code>void NLSpMatGetNormLp(NLCtx *nlP, int p, int *info, double *norm)</code></td>
<td>Provides an interface to the routine SpGetNormLp(), which computes the L-p norm of a matrix stored with the SpMat format. Only the L-1 and L-infinity norms are supported.</td>
</tr>
<tr>
<td><code>void NLSpMatGetRowNormsLp(NLCtx *nlP, int p, double *norm)</code></td>
<td>Provides an interface to the routine SpGetRowNormsLp(), which computes the L-p norm of each row of the matrix, assuming storage with the SpMat format.</td>
</tr>
<tr>
<td><code>int NLSpMatGetRows(NLCtx *nlP)</code></td>
<td>Returns the number of rows of the Jacobian (or Hessian) matrix as set in the database of the nonlinear context.</td>
</tr>
<tr>
<td><code>SPTYPE NLSpMatGetType(NLCtx *nlP)</code></td>
<td>Returns the type of sparse matrix storage format as set in the database of the nonlinear solver context.</td>
</tr>
<tr>
<td><code>int NLSpMatGetUserAllocate(NLCtx *nlP)</code></td>
<td>Returns an integer value of one if NLSpMatSetUserAllocate() has been used to indicate that the user controls memory allocation for the Jacobian (or Hessian) matrix. Returns zero if the default internal allocation is being used.</td>
</tr>
<tr>
<td><code>void NLSpMatMTv(NLCtx *nlP, double *f, double *g)</code></td>
<td>Calls the default SpMat routine for multiplying a vector by the transpose of the Jacobian (or Hessian) matrix.</td>
</tr>
<tr>
<td><code>void NLSpMatMv(NLCtx *nlP, double *f, double *g)</code></td>
<td>Calls the default SpMat routine for multiplying a vector by the Jacobian (or Hessian) matrix.</td>
</tr>
<tr>
<td><code>int NLSpMatSaveMatrixCtx(NLCtx *nlP)</code></td>
<td>Returns a flag that indicates whether NLSpMatSetSaveMatrixCtx() has been called.</td>
</tr>
</tbody>
</table>
void NLSpMatScaleM( NLCtx *nlP, double *D )

Forms the diagonal scaling matrix which scales the Jacobian (or Hessian) matrix by slightly modified L2 row norms, assuming storage with the SpMat scheme.

void NLSpMatSetAIJData( NLCtx *nlP, int *ia, int *ja, double *a )

Sets the location of the MATAIJ data structures for the Jacobian (or Hessian) matrix within the database of the nonlinear context.

void NLSpMatSetBDiagData( NLCtx *nlP, int nd, int nb, int *diag, double **diagv )

Sets the location of the MATBDIAG data for the Jacobian (or Hessian) matrix within the database of the nonlinear context.

void NLSpMatSetCols( NLCtx *nlP, int cols )

Sets the number of columns of the Jacobian (or Hessian) matrix within the database of the nonlinear context.

void NLSpMatSetDeclaredRows( NLCtx *nlP, int rows )

Sets the number of declared rows of the Jacobian (or Hessian) matrix within the database of the nonlinear context.

void NLSpMatSetDenseData( NLCtx *nlP, double *data )

Sets the location of the dense Jacobian (or Hessian) matrix elements within the database of the nonlinear context.

void NLSpMatSetEstRowElements( NLCtx *nlP, int elements )

Sets the estimated number of nonzero elements in each row of the Jacobian (or Hessian) matrix within the database of the nonlinear context.

void NLSpMatSetMaxNonzeros( NLCtx *nlP, int nonzeros )

Sets the maximum number of nonzero elements of the Jacobian (or Hessian) matrix within the database of the nonlinear context.

void NLSpMatSetRows( NLCtx *nlP, int rows )

Sets the number of rows of the Jacobian (or Hessian) matrix within the database of the nonlinear context.

void NLSpMatSetSaveMatrixCtx( NLCtx *nlP )

Sets a flag to indicate that the matrix context should NOT be destroyed during the step-destroy phase. This can reduce the overhead for repeated linear solves, but should be used with caution.

void NLSpMatSetType( NLCtx *nlP, SPTYPE type )

Sets the type of storage format for the Jacobian (or Hessian) matrix within the database of the nonlinear context.
void NLSpMatSetUserAllocate( NLCtx *nlP )

Sets a switch within the database of the nonlinear context to indicate that the user will allocate space for the Jacobian (or Hessian) matrix instead of using the default internal allocation. This option is valid only for the MATAIJ, MATDENSE, and MATBDIAG storage types.

C.4 Additional Routines

void NLCommandLineInterface( NLCtx *nlP, int *argc, char **argv, FILE *fp )

Provides a command line interface to various routines.

int NLENewtonDefaultConverged( NLCtx *nlP, double *xnorm, double *pnorm, double *fnorm )

Default test for monitoring the convergence of the NLE solvers.

void NLENewtonDefaultMonitor( NLCtx *nlP, void *x, void *f, double *fnorm )

Default monitor for NLE solvers. Prints the residual norm at each iteration.

void NLENewtonStatisticsMonitor( NLCtx *nlP, void *x, void *f, double *fnorm )

Statistics monitor routine for NLE solvers. Prints the residual norm at each iteration and calls NLGetStatistics().

int NLENewtonTR1DefaultConverged( NLCtx *nlP, double *xnorm, double *pnorm, double *fnorm )

Default test for monitoring the convergence of the method NLENewtonTR1Solve.

int NLENewtonTR2DefaultConverged( NLCtx *nlP, double *xnorm, double *pnorm, double *fnorm )

Default test for monitoring the convergence of the method NLENewtonTR2Solve.

void NLFormMatrix( NLCtx *nlP, void *x )

Calls the user's matrix formation routine that has been set with NLSetMatrixRoutine(). Also times and counts the calls.

void *NLGetConvergenceContext( NLCtx *nlP )

Gets the convergence context that has been set with NLSetConvergenceTest().

double *NLGetConvergenceHistory( NLCtx *nlP )

Gets the convergence history, where the residual norm (NLE) or gradient norm (NLM) is stored at each step of the iterative solution process.

int NLGetFlops( NLCtx *nlP, NLPHASE phase )

Returns the number of floating point operations (flops) that have been performed during the specified phase since the context's creation or since a call to NLSetFlopsZero().

int NLGetIterationNumber( NLCtx *nlP )

Gets the current iteration number of the nonlinear solver.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void *NLGetLinearSolverCtx( NLCtx *nlP )</code></td>
<td>Extracts the linear solver context from the nonlinear context.</td>
</tr>
<tr>
<td><code>void *NLGetMatrixCtx( NLCtx *nlP )</code></td>
<td>Extracts the Jacobian (or Hessian) matrix context from the nonlinear context.</td>
</tr>
<tr>
<td><code>int NLGetMethodFromCtx( NLCtx *nlP )</code></td>
<td>Extracts the chosen method from the nonlinear context.</td>
</tr>
<tr>
<td><code>char *NLGetMethodName( NLMETHOD nlmethod )</code></td>
<td>Gets the name (as a string) from the method type.</td>
</tr>
<tr>
<td><code>int NLGetMethodType( NLCtx *nlP )</code></td>
<td>Returns the method type (NLE or NLM).</td>
</tr>
<tr>
<td><code>void *NLGetMonitorContext( NLCtx *nlP )</code></td>
<td>Gets the convergence context that has been set with NLSetMonitor().</td>
</tr>
<tr>
<td><code>int NLGetNumberInnerIterations( NLCtx *nlP )</code></td>
<td>Gets the cumulative iteration count that is returned by the step-compute routine. If the SLES or KSP linear solvers are used, this returns the total number of iterations of the linear solvers.</td>
</tr>
<tr>
<td><code>int NLGetNumberOfCalls( NLCtx *nlP, NLPHASE phase )</code></td>
<td>Gets the number of times the specified phase has been executed since the context’s creation or since a call to NLSetNumberOfCallsZero().</td>
</tr>
<tr>
<td><code>int NLGetNumberUnsuccessfulSteps( NLCtx *nlP )</code></td>
<td>Gets the number of unsuccessful steps attempted by the nonlinear solver.</td>
</tr>
<tr>
<td><code>double NLGetParameter( NLCtx *nlP, char *param )</code></td>
<td>Returns the value of a chosen parameter, which is unique to a particular solution method.</td>
</tr>
<tr>
<td><code>void NLGetStatistics( NLCtx *nlP, FILE *fp )</code></td>
<td>Writes performance statistics (operation counts and times) for various phases of the nonlinear solver. See NLPhaseDescriptions() for descriptions of the phases.</td>
</tr>
<tr>
<td><code>char *NLGetTerminationType( NLCtx *nlP )</code></td>
<td>Returns information regarding the type of termination that occurred for the nonlinear solver.</td>
</tr>
<tr>
<td><code>void NLGetTime( NLCtx *nlP, NLPHASE phase, double *telp, double *tcpu )</code></td>
<td>Returns the aggregate time in seconds for the specified phase.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VECntx *NLGetVectorCtx( NLCtx *nlP )</td>
<td>Extracts the vector context from the nonlinear context.</td>
</tr>
<tr>
<td>int NLGetVectorDimension( NLCtx *nlP )</td>
<td>Extracts the vector dimension from the vector context within the nonlinear context.</td>
</tr>
<tr>
<td>int NLIsContextValid( NLCtx *nlP )</td>
<td>Checks whether the given context is valid.</td>
</tr>
<tr>
<td>NLMETHOD NLIsMethodAvailable( char *name )</td>
<td>Returns a nonlinear solver method specified by name.</td>
</tr>
<tr>
<td>void NLMNewtonDefaultMonitor( NLCtx *nlP, void *x, void *g, double *f, double *gnorm )</td>
<td>Default monitor for NLM solvers. Prints the function value and the gradient norm at each iteration.</td>
</tr>
<tr>
<td>void NLMNewtonStatisticsMonitor( NLCtx *nlP, void *x, void *g, double *f, double *gnorm )</td>
<td>Statistics monitor routine for NLM solvers. Prints the function value and the gradient norm at each iteration and calls NLGetStatistics().</td>
</tr>
<tr>
<td>int NLMNewtonTR1DefaultConverged( NLCtx *nlP, double *xnorm, double *gnorm, double *f )</td>
<td>Default test for monitoring the convergence of the NLMNewtonTR1Solve() routine.</td>
</tr>
<tr>
<td>void NLMNewtonTR1SetRoutines( NLCtx *nlP, void(*NormM)(), void(*SetUpM)(), void(*DestroyM)(), void(*Mv)(), void(*FSolve)(), void(*BSolve)() )</td>
<td>Sets the routines for various aspects of the method NLM.NTR1.</td>
</tr>
<tr>
<td>void NLOutput( NLCtx *nlP, int *argc, char **argv, FILE *fp )</td>
<td>Writes output regarding performance and results of the nonlinear solver to the specified file.</td>
</tr>
<tr>
<td>void NLRegisterAll( )</td>
<td>This routine registers all of the solution methods in the SNES package.</td>
</tr>
<tr>
<td>void NLRegisterDestroy( )</td>
<td>Frees the list of nonlinear solvers that have been registered by NLRegister().</td>
</tr>
<tr>
<td>void NLRegister( int name, char *sname, void(*create)() )</td>
<td>Adds the method to the nonlinear solver package, given a function pointer and a nonlinear solver name of the type NLMETHOD.</td>
</tr>
<tr>
<td>void NLSaveConvergenceHistory( NLCtx *nlP )</td>
<td>Allocates and sets the location of where the residual norm (NLE) or gradient norm (NLM) should be stored at each iteration.</td>
</tr>
</tbody>
</table>
### void NLScaleStep( NLCtx *nlP, double *y, double *fnorm, double *delta, double *gpnorm, double *ynorm )

Scales a step so that its length is less than the positive parameter delta.

### void NLSetAbsConvergenceTol( NLCtx *nlP, double atol )

Sets the absolute convergence tolerance.

### void NLSetConvergenceTest( NLCtx *nlP, int(*func)(), void *cctx )

Sets the function that is to be used to test for convergence of the nonlinear iterative solution.

### void NLSetDBVectors( NLCtx *nlP, int n )

Sets the vector operations within the nonlinear context for BLAS based serial double-precision vectors. The vector context should have been created with VECreate(). Use NLCreateDBVectors() to create the vector context and fill it with BLAS vectors in one step.

### void NLSetDVectors( NLCtx *nlP, int n )

Sets the vector operations within the nonlinear context for serial double-precision vectors. The vector context should have been created with VECreate(). Use NLCreateDVectors() to create and fill the vector context in one step.

### void NLSetDefaults( NLCtx *nlP, void *usrP )

Sets the default parameters and routines for the nonlinear solvers.

### void NLSetFlopsZero( NLCtx *nlP )

Resets all floating point operations (flop) counters associated with a nonlinear solver context.

### void NLSetFlops( NLCtx *nlP, NLPHASE phase, int flops )

Sets the number of floating point operations (flops) that have been used during the specified phase.

### void NLSetLineSearchRoutine( NLCtx *nlP, int(*func)() )

Sets the line search routine to be used by the method NLE-NLS1.

### void NLSetLinearSolverCtx( NLCtx *nlP, void *linsvctx, int sles )

Sets the linear solver context within the nonlinear context.

### void NLSetLog( NLCtx *nlP, FILE *file )

Indicates that log information from the nonlinear solvers will be written to the specified file. If no file is given, the information will be written to standard output.

### void NLSetMatrixCtx( NLCtx *nlP, void *matrix, int spmat )

Sets the Jacobian (or Hessian) matrix context within the nonlinear context.

### void NLSetMatrixRoutine( NLCtx *nlP, void(*func)() )

Sets the Jacobian (or Hessian) evaluation routine that will be called by NLFormMatrix().
void NLSetMaxFunctionEvaluations( NLCtx *nlP, int maxf )
Sets the maximum number of function evaluations to use by the NLM solvers.

void NLSetMaxIterations( NLCtx *nlP, int its )
Sets the maximum number of global iterations to use.

void NLSetMaxResidualEvaluations( NLCtx *nlP, int maxres )
Sets the maximum number of residual evaluations to use by the NLE solvers.

void NLSetMinFunctionTol( NLCtx *nlP, double fmin )
Sets the lower bound on the value of the objective function. Used by the NLM convergence testing routines.

void NLSetMonitor( NLCtx *nlP, void(*func)(), void *mctx )
Sets the function which is to be used at every iteration of the nonlinear solution to display the iteration’s progress.

void NLSetParameter( NLCtx *nlP, char *param, double *value )
Sets a chosen parameter used by the solution method to the specified value.

void NLSetRelConvergenceTol( NLCtx *nlP, double rtol )
Sets the relative convergence tolerance.

void NLSetRelativeLinearTol( NLCtx *nlP, double tol )
Sets the relative tolerance to be used by the iterative linear solvers. This parameter is used by the convergence monitoring routines NLKSPDefaultConverged() and NLKSPQuadraticConverged().

void NLSetSolutionTol( NLCtx *nlP, double tol )
Sets the convergence tolerance in terms of the norm of the change in the solution between steps.

void NLSetTruncationTol( NLCtx *nlP, double tol )
Sets the tolerance which may be used by the step routines to control the accuracy of the step computation.

void NLSetTrustRegionTol( NLCtx *nlP, double dtol )
Sets the trust region parameter tolerance.

void NLSetUserCtx( NLCtx *nlP, void *userctx, )
Sets the optional user-defined context within the nonlinear context.

void NLSetVectorCtx( NLCtx *nlP, VECntx *vctx )
Sets the vector context within the nonlinear context.

int NLStepDefaultLineSearch( NLCtx *nlP, void *x, void *f, void *g, void *y, void *w, double fnorm, double *ynorm, double *gnorm )
This routine performs a cubic line search.
void NLStepDoglegSetMonitor( NLCtx *nlP, void(*func)() )

Sets the function that is to be called at every iteration of the dogleg step routine. The default is to print nothing.

void NLStepDoglegSetRoutines( NLCtx *nlP, void(*SetUpSolveM)(), void(*ScaleM)(), void(*MTv)(), void(*Mv)(), int(*SolveM)(), void(*DestroyM)() )

Sets the routines for various aspects of the dogleg step, which is employed by the method NLE_NTR2DOG.

int NLStepDoglegSolve( NLCtx *nlP, void *x, void *f, void *p, double *fnormp, double *deltap, double *rtol, double *gpnorm, double *pnorm, void *mix )

This routine computes an approximate solution, p, to the minimization problem

void NLStepLinearSetRoutines( NLCtx *nlP, void(*ScaleM)(), void(*Mv)() )

Sets the routines for various aspects of the linear step, which is used by the method NLE_NTR2LIN.

int NLStepSimpleLineSearch( NLCtx *nlP, void *x, void *f, void *g, void *y, void *w, double fnorm, double *ynorm, double *gnorm )

This routine is not a line search at all; it simply uses the full Newton step. Thus, this routine is intended to serve as a template and is not recommended for general use.
Acknowledgments

The work described in this report has benefited from conversations with and use by many of people. We especially thank David Keyes and Jorge Moré, as well as the early users who requested additional functionality and were patient with our bug fixes.
Bibliography


54


Function Index

N
NLCreate ............................................. 15, 16
NLCreateDBVectors ................................. 17
NLCreateDVectors .................................. 17
NLDestroy ............................................ 16, 31
NLENewtonDefaultMonitor ......................... 31
NLENewtonStatisticsMonitor ...................... 31
NLENewtonTRICreate ................................ 33
NLGetLinearSolverCtx ............................ 20, 21
NLGetMatrixCtx .................................... 24, 25
NLGetMethod ......................................... 15, 16, 38
NLGetParameter ...................................... 19
NLGetStatistics .................................... 31, 32
NLKSPDefaultConverged ........................... 23
NLKSPQuadraticConverged ......................... 23
NLMNewtonDefaultMonitor ......................... 32
NLMNewtonStatisticsMonitor ...................... 32
NLOutput ............................................... 31
NLRegister ............................................ 33
NLRegisterAll ...................................... 34
NLScaleStep ......................................... 28
NLSetInitialGuess .................................. 18
NLSetLinearSolverCtx .............................. 20
NLSetLog .............................................. 18
NLSetMatrixCtx .................................... 24, 25
NLSetMatrixRoutine ............................... 18, 25
NLSetParameter ...................................... 19
NLSetResidualRoutine ............................. 17, 31
NLSetSolution ........................................ 19
NLSetStepDestroy ................................... 28
NLSetUp ............................................... 16, 19, 29
NLSlesSetRoutines ................................ 17, 18, 25, 29
NLSlesSetSaveSolverCtx .......................... 29
NLSlesSetUp ......................................... 26
NLSlesSolveScale ................................... 28
NLSlesSVCreate ..................................... 20
NLSlesSVDestroy .................................... 21, 29
NLSlesSVSetConvergenceTest ...................... 21
NLSlesSVSetMonitor .................................. 21
NLSlesSVSetOperators ............................. 21, 29
NLSlesSVSetUp ...................................... 21, 23, 29
NLSlesSVSolve ....................................... 21, 28
NLSolve ................................................ 16, 26, 31
NLSpMatCreateMatrix .............................. 24, 25, 26

NLSpMatCreateMatrixFromData ................. 24, 26
NLSpMatGetAIJData ................................. 25
NLSpMatGetDenseData .............................. 25
NLSpMatMv ........................................... 29
NLSpMatScaleM ....................................... 29
NLSpMatSetAIJData ................................. 25
NLSpMatSetCols ..................................... 26
NLSpMatSetDenseData .............................. 25
NLSpMatSetEstRowElements ....................... 25
NLSpMatSetRows ..................................... 26
NLSpMatSetSaveMatrixCtx ......................... 29
NLSpMatSetUserAllocate ......................... 26
NLSpMatSetType ..................................... 26
NLStepDoglegSetRoutines ......................... 29, 30
NLStepLinearSetRoutines ......................... 29
NLKSPDefaultConverged ........................... 23
NLKSPQuadraticConverged ......................... 23
NLNewtonDefaultMonitor ......................... 32
NLNewtonStatisticsMonitor ...................... 32
Sp<format>Create ................................. 25
SpAddDenseSubmatrix .............................. 25
SpAddInRow ......................................... 25
SpAddValue ......................................... 25
SpDestroy ............................................ 29
SpSetValue ........................................... 25
SVCreate ............................................. 20
SVDestroy ............................................ 29
SVSetAccelerator ................................... 23