NITSOL: A Newton Iterative Solver for Nonlinear Systems

A FORTRAN-to-MATLAB Implementation

By

Bijaya Padhy

A Masters Project

Submitted to the Faculty

Of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

In

Applied Mathematics

May 2006

APPROVED:

_____

Professor Homer Walker

_____

Professor Bogdan Vernescu, Department Head

Table of Contents

Acknowledgements

I would like to thank my husband Timothy for all the encouragement, energy and time he gave so I could focus on my work. I would like to give thanks to my advisor, Professor Homer Walker, for his time and efforts every week as I worked through this project with him. Finally, I give thanks to God for the opportunity to complete this project and fulfill my Master's Degree requirements.

Abstract

NITSOL: A Newton Iterative Solver for Nonlinear Systems describes an algorithm for solving nonlinear systems. Michael Pernice and Homer F. Walker, the authors of the paper NITSOL [3], implemented this algorithm in FORTRAN. The goal of the project has been to use the modern and robust language MATLAB to implement the NITSOL algorithm. In this paper, the main mathematical and algorithmic background for understanding NITSOL are described, and a user guide is included outlining how to use the MATLAB implementation of NITSOL. A nonlinear system example problem, the 2D Bratu problem, and the solution obtained by MATLAB NITSOL's are also included.

NITSOL: A Newton Iterative Solver for Nonlinear Systems

A FORTRAN-to-MATLAB Implementation of NITSOL

NITSOL is an algorithm for solving nonlinear systems. This algorithm was implemented in FORTRAN by the authors of the paper [3]. The goal of this project has been to use the modern and robust language MATLAB to implement the NITSOL algorithm. The MATLAB code takes advantage of built-in features and routines of MATLAB. Since FORTRAN needs work arrays and the names of all input and output variables, the FORTRAN NITSOL routines are cumbersome. MATLAB does not need these, and therefore the execution of the solver is simplified. By incorporating initialization of all variables, error checking, and the main driver of the solver in one routine, the number of routines necessary for execution is reduced.

NITSOL Background

The algorithmic background described below summarizes the methods described in [3], Globally Convergent Inexact Newton Methods [2], A Simpler GMRES [9], and Numerical Methods for Nonlinear Equations [8] course notes.

Consider the system of nonlinear equations $F(x) = 0$, where $F : R^n \to R^n$ is continuously differentiable. Assume that $F(x) = 0$ has a solution x_* such that $F'(x_*)$ is invertible. A standard algorithm for solving this nonlinear system is Newton's Method [2].

Newton's Method

Let x_0, tol be given.

For k=0, 1, … until $\|F(x_k)\| \le tol$

Solve $F'(x_k)s_k = -F(x_k)$ for s_k

Set $x_{k+1} = x_k + s_k$

□

The main strength of Newton's method is that if x_0 is sufficiently close to x_* then $\{x_k\}$ quadratically converges to x_*. At each iteration of this method, $F'(x_k)s_k = -F(x_k)$ must be solved. Computing the exact solution to this can be expensive if n is large. Also, the resulting s_k may not be useful if x_k is far from a solution. This leads us to prefer computing an approximate solution, which brings us to the Inexact Newton Method [2].

Inexact Newton Method

Let x_0, tol be given.

For k=0, 1, … until $\|F(x_k)\| \le tol$

Find some $\eta_k \in [0,1)$ and s_k such that

$$\| F(x_k) + F'(x_k)s_k \| \le \eta_k \| F(x_k) \|$$

Set $x_{k+1} = x_k + s_k$

□

Here each η_k reflects how accurately s_k solves $F'(x_k)s_k = -F(x_k)$. This is referred to as s_k satisfying an *inexact Newton condition*. Like Newton's method, the convergence of this method is local. The iterates may not converge if x_0 is not near a solution. This inexact Newton method can be modified to offer strong global convergence properties and by choosing the η_k's appropriately, give faster local convergence. One method of improving global convergence is backtracking, which gives us the Inexact Newton Backtracking method [2].

Inexact Newton Backtracking Method

Let x_0, tol be given.

Let $\eta_{\max} \in [0,1), t \in (0,1), \text{and } 0 < \theta_{\min} < \theta_{\max} < 1$ be given.

For k=0, 1, … until $\|F(x_k)\| \le tol$

Choose initial $\eta_k \in [0, \eta_{\max}]$ and s_k such that

$$\| F(x_k) + F'(x_k)s_k \| \le \eta_k \| F(x_k) \|$$

While $\| F(x_k + s_k) \| > [1 - t(1 - \eta_k)] \| F(x_k) \|$

Choose $\theta \in [\theta_{\min}, \theta_{\max}]$

Update $s_k = \theta \cdot s_k$ and $\eta_k = 1 - \theta(1 - \eta_k)$

Set $x_{k+1} = x_k + s_k$

□

The principal theorem for the Inexact Newton Backtracking (INB) algorithm is the following [3]:

Theorem: *Assume that F is continuously differentiable. If {x_k} produced by the INB algorithm has a limit point x_* such that $F'(x_*)$ is invertible, then $F(x_*) = 0$ and $x_k \to x_*$. Furthermore, the initial s_k and η_k are accepted without modification in the while- loop for all sufficiently large k.*

From this theorem, it follows that one of the following must hold:

- $\| x_k \| \to \infty, i.e. \{x_k\}$ has no limit points.
- *{x_k}* has one or more limit points, and *F'* is singular at each of them.
- *{x_k}* converges to a solution x_* at which *F'* is invertible.

The idea with backtracking is that if a step is not acceptable, then it is shortened until it is acceptable. The reduction $s_k = \theta \cdot s_k$ with $\theta \in [\theta_{\min}, \theta_{\max}]$ is called *"safeguarded" backtracking*. The condition $\theta \le \theta_{\max}$ guarantees that the backtracking loop will end with an acceptable step. The condition $\theta \ge \theta_{\min}$ guarantees that the accepted step will not be too short. The practical recommendation is to choose t small, such as $t = 10^{-4}$, so that a step will be accepted if there is minimal progress. Typically $\theta_{\min} = 0.1$ and $\theta_{\max} = 0.5$ are chosen in practice [1].

The NITSOL algorithm uses the Inexact Newton Backtracking Method to solve the nonlinear system $F(x) = 0$. In using the backtracking method, the backtracking safeguard values are set to $\theta_{\min} = 0.1$ and $\theta_{\max} = 0.5$. The norm $\|\cdot\|$ is an inner-product

norm, and $\theta \in [\theta_{\min}, \theta_{\max}]$ is chosen such that it minimizes a quadratic that interpolates $\|F\|$ in the direction of the inexact Newton step. For sufficient reduction, *t* is set to 10^{-4}. Also the upper bound $\eta_{\max} = 0.9$. The user supplies FTOL (function tolerance) and STPTOL (step tolerance), and convergence is declared when $\|F(x_k)\| \le FTOL$ or $\|s_k\| \le STPTOL \cdot \|x_k\|$.

Finite Differences

In all these methods, the product of $F'(x_k)$ with a vector is required. This can be evaluated analytically or approximated by using a finite-difference approach. A few examples of finite-difference formulas are [3]:

$$F'(x_k)v \approx \frac{1}{\delta}[F(x_k + \delta v) - F(x_k)]$$

$$F'(x_k)v \approx \frac{1}{2\delta}[F(x_k + \delta v) - F(x_k - \delta v)]$$

$$F'(x_k)v \approx \frac{1}{6\delta}[8F(x_k + \frac{\delta}{2}v) - 8F(x_k - \frac{\delta}{2}v) - F(x_k + \delta v) + F(x_k - \delta v)]$$

A high-level option for the user is whether or not to use finite differences in computing the Jacobian $F'(x_k)$. The finite-difference routine incorporates 1st order, 2nd order and 4th order formulas as shown respectively above.

Right Preconditioning

For difficult problems, preconditioners can be used to reduce computational time in approximately solving $F'(x_k)s_k = -F(x_k)$. For practical purposes, the benefits from using a preconditioner must outweigh the cost of its use. There are left and right preconditioners [8]. Using a left preconditioner $M \in R^{n \times n}$, one solves $M^{-1}F'(x)s = -M^{-1}F(x)$. With a right preconditioner, one solves $F'(x)M^{-1}z = -F(x)$, and then sets $s = M^{-1}z$.

The Krylov Subspace Methods

These methods are used to solve the linear problem $Ax = b,\ A \in R^{n\,x\,n},\ b \in R^n$. Assume that A is nonsingular. A Krylov subspace method in general has this form [8]:

Given x_0, determine…

$$x_k = x_0 + z_k$$

$$z_k \in \kappa_k \equiv span\{r_0, Ar_0, ..., A^{k-1}r_0\},$$

where κ_k is the k^{th} Krylov subspace.

There are many Krylov subspace methods, including the Generalized Minimal Residual Method (GMRES) [9], Biconjugate Gradient Stabilized Method (BICGSTAB) [7], and Conjugate Gradient Squared Method (CGS) [5]. These three methods require only products of A with vectors for their implementation and are the three methods offered by MATLAB having this property. The methods available with NITSOL include these and also an alternative implementation of GMRES from [9].

Simpler GMRES

Simpler GMRES (Gram-Schmidt implementation) [9]:

Given A, b, x, tol.

Initialize: Set $r \equiv b - Ax, \quad \rho_0 = \|r\|_2$.

If $\rho_0 \le tol$, accept *x* and exit; otherwise update $r = \frac{r}{\rho_0}$ and set $\rho = 1$.

Iterate: For *k = 1,...,m*, do:

1. Evaluate $v_k \equiv Av_{k-1}$, where $v_1 \equiv Ar$
2. If $k > 1$, then for i = 1, … , k-1, do:
   a. Set $\rho_{i,k} = v_i^T v_k$.
   b. Update $v_k = v_k - \rho_{i,k} v_i$.
3. Set $\rho_{k,k} = \|v_k\|_2$.
4. Update $v_k = \frac{v_k}{\rho_{k,k}}$.

5. Set $R_k = \begin{pmatrix} & \rho_{1,k} \\ R_{k-1} & \vdots \\ 0\cdots 0 & \rho_{k,k} \end{pmatrix}$, where $R_1 \equiv \rho_{1,1}$

6. Set $\xi_k \equiv r^T v_k$.

7. Update $\rho = \rho \sin(\cos^{-1}(\xi_k / \rho))$. If $\rho \cdot \rho_0 \le tol$, go to Solve.

8. Update $r = r - \xi_k v_k$.

Solve: Let k be the final iteration number from Iterate.

1. Solve $R_k y = (\xi_1, \cdots, \xi_k)^T$ for $y = (\eta_1, \cdots, \eta_k)^T$.

2. Form $z = \begin{cases} \eta_1 r, & \text{if } k = 1 \\ \eta_1 r + \sum_{i=1}^{k-1} (\eta_{i+1} + \eta_1 \xi_i) v_i, & \text{if } k > 1 \end{cases}$.

3. Update $x = x + \rho_0 z$.

4. If $\rho \cdot \rho_0 \le tol$, accept *x* and exit;

   otherwise update $r = \dfrac{r - \xi_k v_k}{\rho}$, update $\rho_0 = \rho \cdot \rho_0$, and set $\rho = 1$, and return to Iterate.

□

Another way to update *r* and ρ at the end in step 4 of Solve is by:

$r = \dfrac{(b - Ax)}{\|b - Ax\|_2}$ and $\rho_0 = \|b - Ax\|_2$.

In some cases, this method of updating those variables may be cheaper and more accurate.

CGS

This method, in contrast to GMRES, which is a long-recurrence method, can be implemented with short recurrences.

Conjugate Gradient Squared Method [5]:

Given x_0.

Set $p_0 = w_0 = r_0 = b - Ax_0, \quad v_0 = Ap_0$.

Choose $\tilde{r}_0$ such that $\rho_0 = \tilde{r}_0^T r_0 \neq 0$.

For k = 1, 2, …, do:

Compute

$$\sigma_{k-1} = \tilde{r}_0^T v_{k-1}$$

$$\alpha_{k-1} = \frac{\rho_{k-1}}{\sigma_{k-1}}$$

$$q_k = u_{k-1} - \alpha_{k-1} v_{k-1}$$

$$x_k = x_{k-1} + \alpha_{k-1}(u_{k-1} + q_k)$$

$$r_k = r_{k-1} - \alpha_{k-1} A(u_{k-1} + q_k)$$

$$\rho_k = \tilde{r}_0^T r_k$$

$$\beta_k = \frac{\rho_k}{\rho_{k-1}}$$

$$u_k = r_k + \beta_k q_k$$

$$p_k = u_k + \beta_k (q_k + \beta_k p_{k-1})$$

$$v_k = A p_k$$

□

BICGSTAB

This method like CGS has short recurrences and requires only A products. This method produces much smoother residual norms performance than CGS but for some applications, the residual norms behave wildly. There are many variations of this method. The one described here is similar to the CGS method. The CGS residuals are given by $r_k^{CGS} = \psi_k(A)^2 r_0$, where ψ_k is the k^{th} BCG residual polynomial, i.e., $r_k^{BCG} = \psi_k(A) r_0$. The k^{th} BiCGSTAB residual is $r_k = \tilde{\psi}_k(A)\psi_k(A) r_0$. A specific choice of $\tilde{\psi}_k$ is $\tilde{\psi}_k = (1 - w_k t)\tilde{\psi}_{k-1}(t)$ [6], where w_k is chosen so that $\left\| r_k^{BiCGSTAB} \right\|_2 = \left\| (1 - w_k A)\tilde{\psi}_{k-1}(A)\psi_k(A) r_0 \right\|_2$ is minimal.

Biconjugate Gradient Stabilized Method [7]

Given x_0.

Set $r_0 = b - A x_0$.

Choose $\tilde{r}_0$ such that $\rho_0 = \tilde{r}_0^T r_0 \neq 0$.

For k = 1, 2, …, do:

Compute

$\rho_{i-1} = \tilde{r}_0^T r_{i-1}$

if i=1

$p_i = r_{i-1}$

else

$\beta_{i-1} = \left(\frac{\rho_{i-1}}{\rho_{i-2}}\right)\left(\frac{\alpha_{i-1}}{\omega_{i-1}}\right)$

$p_i = r_{i-1} + \beta_{i-1}(p_{i-1} - \omega_{i-1} v_{i-1})$

endif

$v_i = Ap_i$

$\alpha_i = \frac{\rho_{i-1}}{\tilde{r}^T v_i}$

$s = r_{i-1} - \alpha_i v_i$

If $\|s\|_2 \le stptol$

$x_i = x_{i-1} + \alpha_i p_i$

exit

endif

$t = As$

$\omega_i = \frac{t^T s}{t^T t}$

$x_i = x_{i-1} + \alpha_i p_i + \omega_i s$

$r_i = s - \omega_i t$

If $\|r\|_2 \le tol$ and $\omega_i \neq 0$ continue.

□

User Guide

The MATLAB Implementation of NITSOL

Overview

The MATLAB NITSOL was created to replace legacy FORTRAN code coauthored by Michael Pernice and Homer Walker [3]. The MATLAB version has some modifications that take advantage of built-in functions and simpler routine calls.

This section gives a general overview of how to use the MATLAB NITSOL package.

Software Requirements

MATLAB is required in order to run the NITSOL program. MATLAB version 7.0.4.365 (R14) SP 2 was used in creating the routines; a previous version of MATLAB may not contain all necessary built in functions.

Files Included in this Version

nitset.m – Routine which sets default variables in the option structure. Options can be changed as desired.

nitsol.m – Routine which executes the entire NITSOL non-linear solver program.

nitgm.m – Routine which solves a linear system using the Simpler GMRES Krylov solver.

nitjv.m – Routine which decides which Jacobian vector product to return based on option settings. Either Jv, $JP^{-1}v$, or $P^{-1}v$ are returned, where J is the Jacobian and P is the right preconditioner.

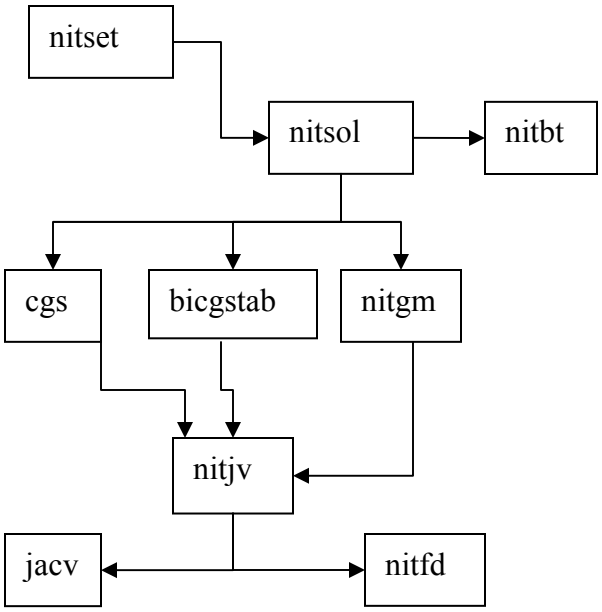
nitfd.m – Routine which approximates Jacobian vector products using a finite-difference approach. An option setting declares which order finite difference to implement.

nitbt.m – Routine which handles backtracking.

f_ros.m – Sample non-linear function to assist in proper 'funct_name' function format.

jacv_ros.m – Sample Jacobian vector product routine including right preconditioner to assist in proper 'jacv' routine format.

Diagram of routine dependencies and routine calls:



Basic Routines

The basic routines are nitset.m and nitsol.m. nitset.m sets all default and necessary variables and must be executed first. nitsol.m runs the NITSOL program and solves the desired nonlinear system. All other routines are called from nitsol.m and do not need to be executed independently. To end a session, type 'exit' at the MATLAB prompt. If MATLAB is 'Busy' and hangs on a process, then press Ctrl-C to quit the process.

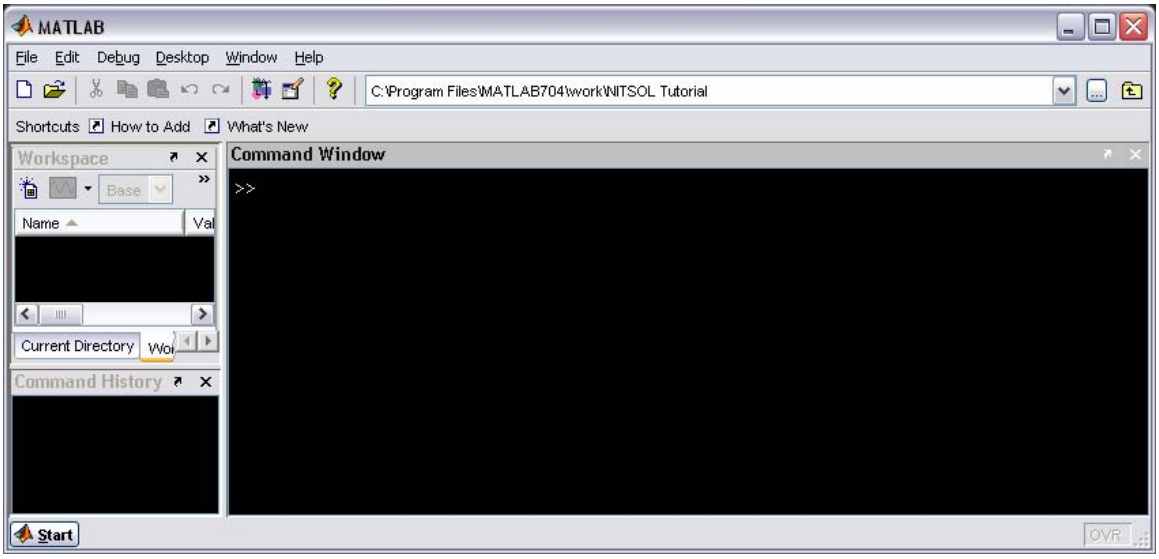
Tutorial

The tutorial section of the User Guide presents the various capabilities of the NITSOL program. While this tutorial does not cover every capability of the NITSOL program, it gives a good starting point for beginning users of the routines. The tutorial will go through an example of executing the routine to solve a simple nonlinear system. The files used are included to assist in the tutorial. The steps are to be followed sequentially.

Getting started

1. Unzip/Copy all the MATLAB routines into a designated directory.
2. Open MATLAB.
3. Make sure the routine files reside in the working directory.

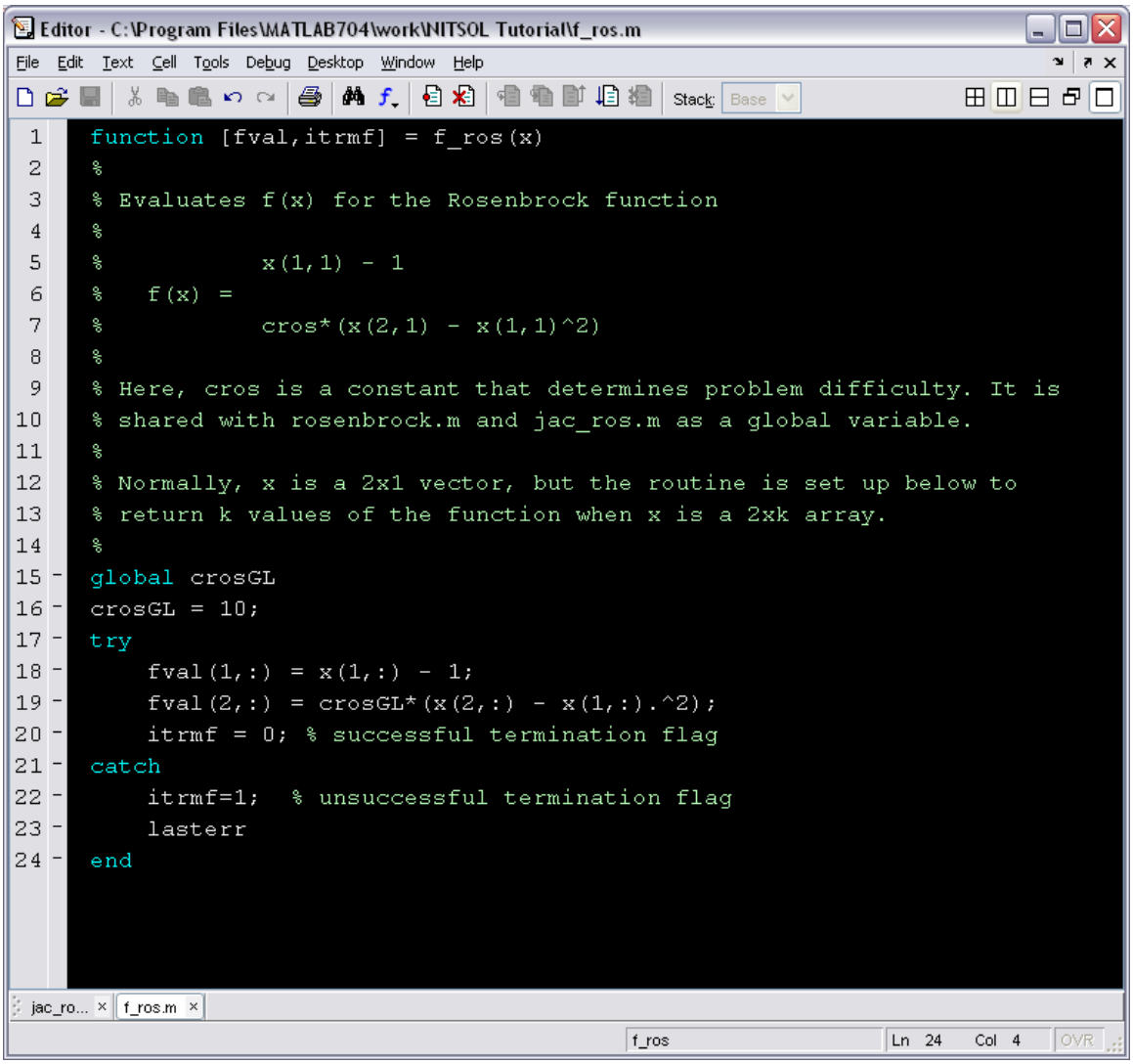
   Example: Here the working directory is C:\Program Files\MATLAB704\work\NITSOL Tutorial



Creating Required Routines

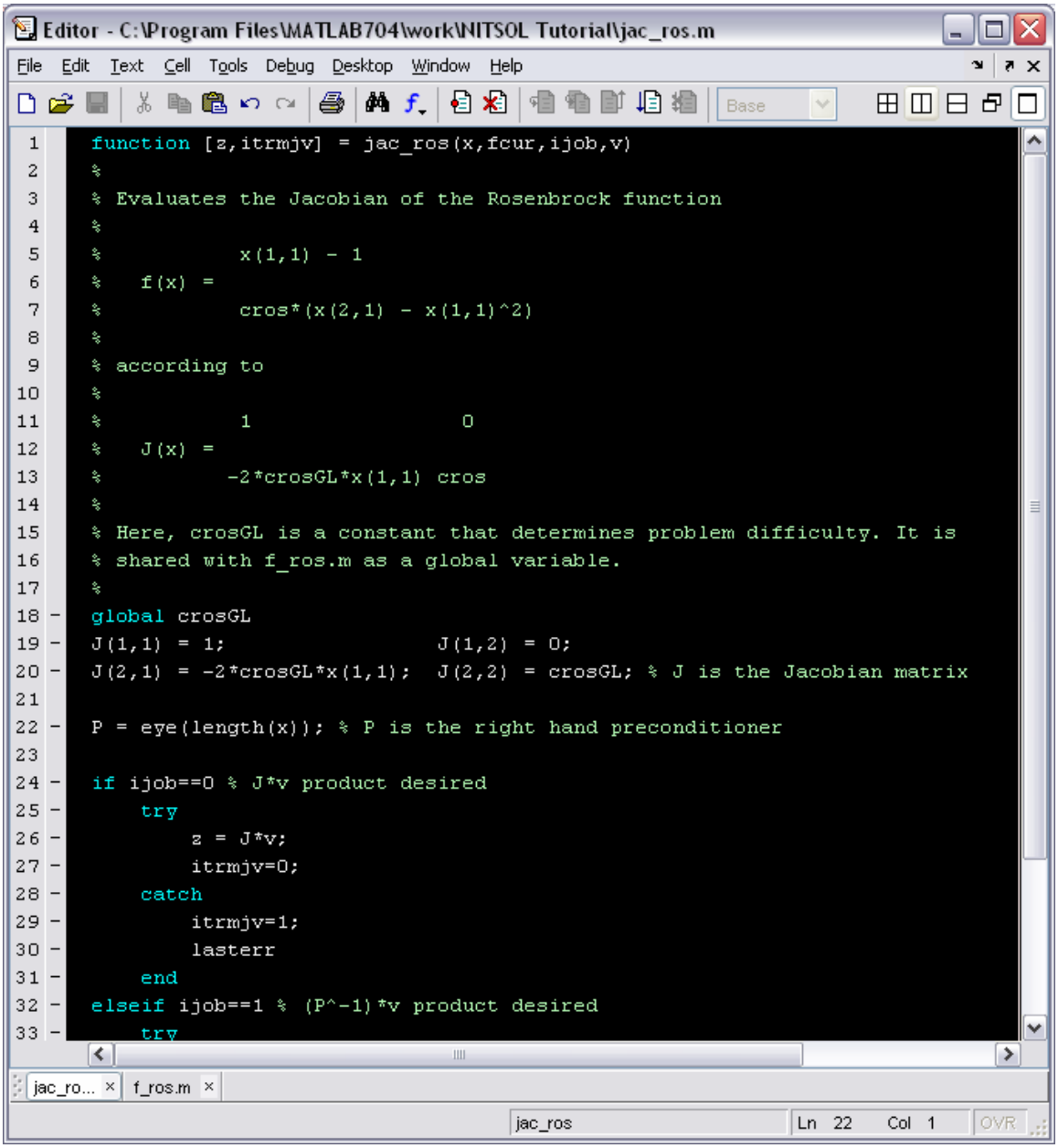
1. Create f.m (this file can take on any name) which evaluates the nonlinear function describing the system which is to be solved.

   Example: Type 'edit f_ros.m' at the prompt. This will open up the test example included in this package. The input and output variables of the created function should be in the form shown.



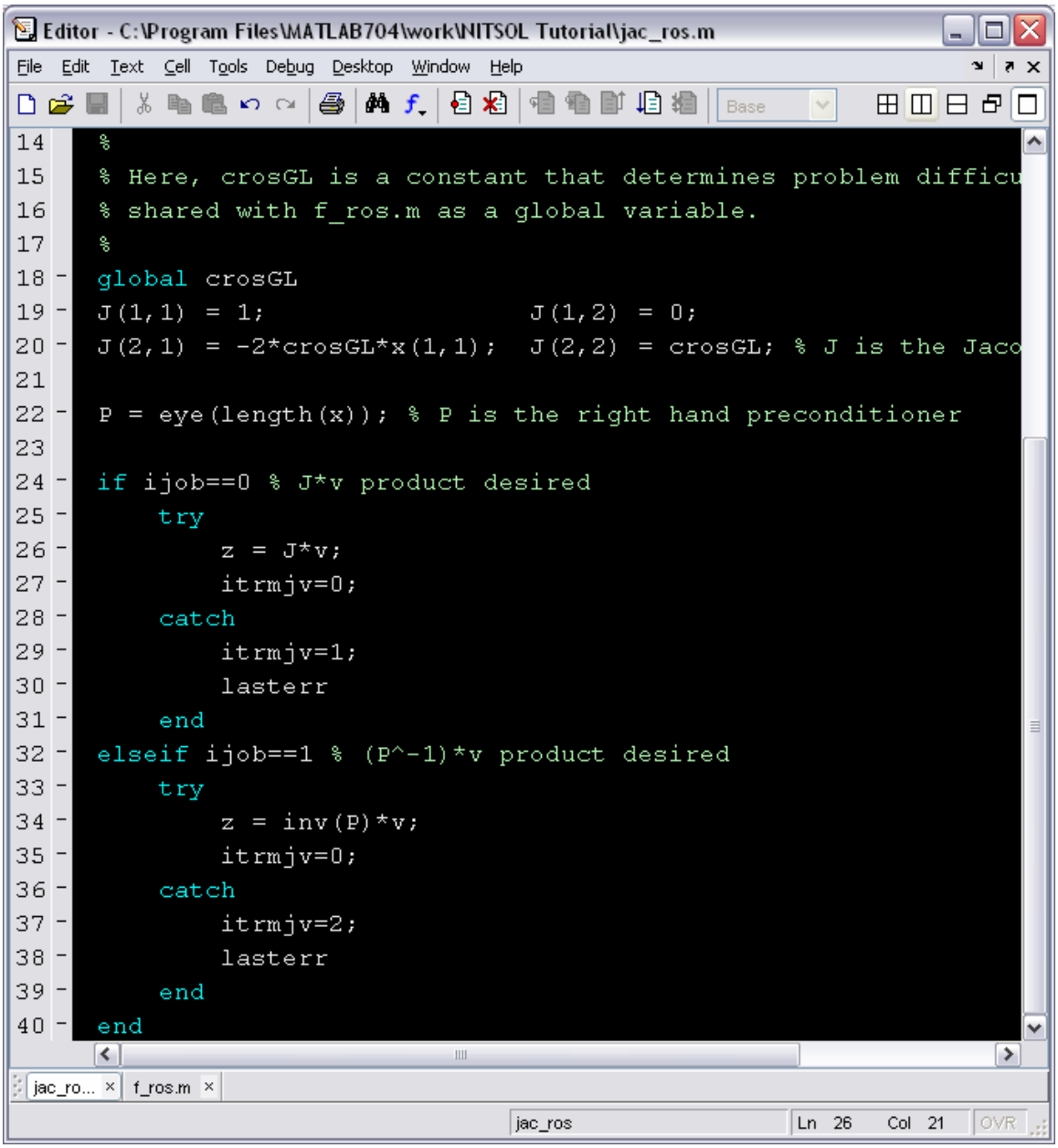
2. Create jacv.m (this file can have any name) if an analytic Jacobian-vector product routine can be provided.

   Example: Type 'edit jac_ros.m' at the prompt. This will open up the test example included in this package. The input and output variables of the created function should be in the form shown.



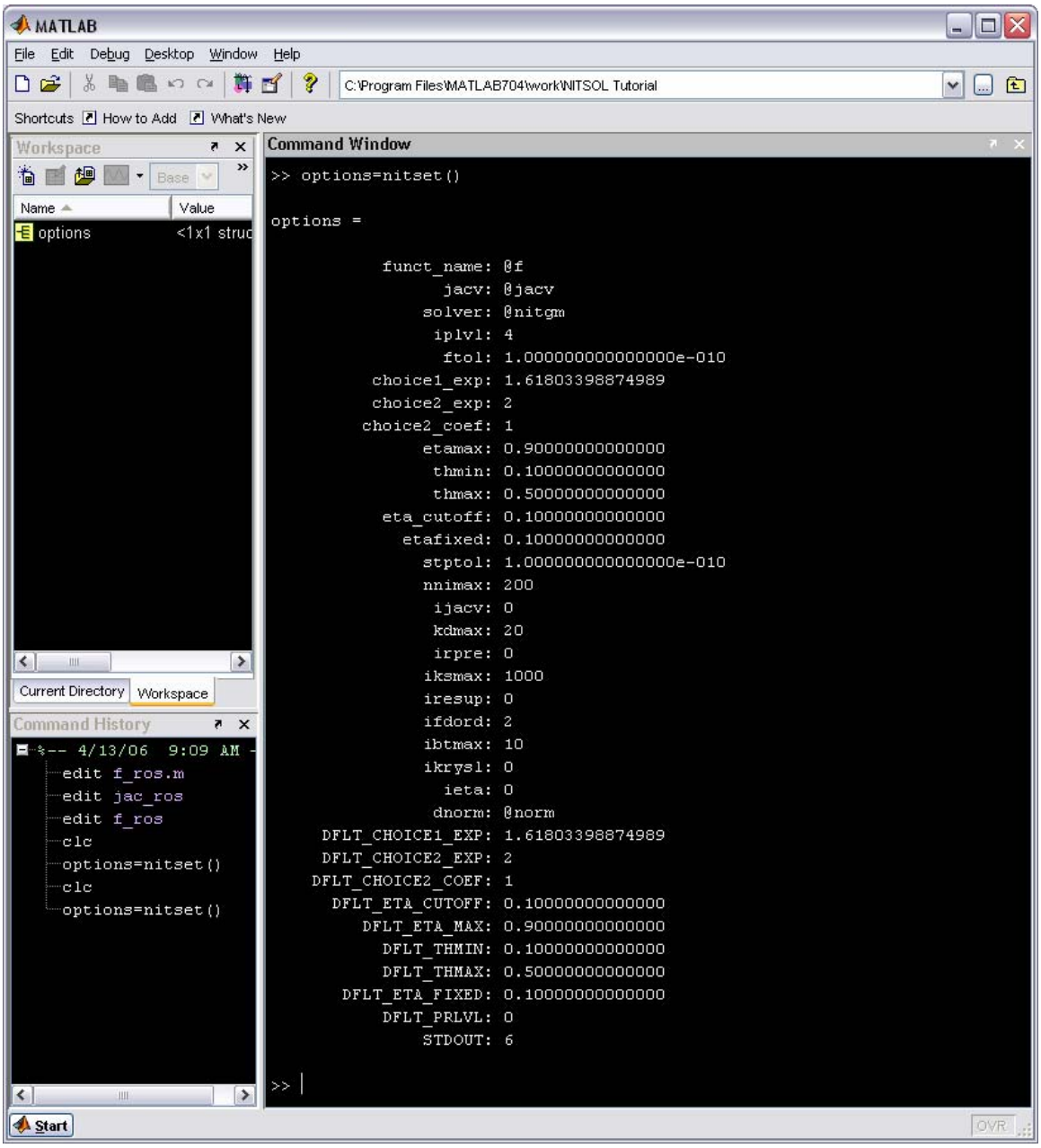
3. Include within the jacv.m file (this file can have any name) the code to evaluate a preconditioner solve, if right preconditioning is desired.

   Example: Within jac_ros.m is included the code for implementing the right preconditioner.



Setting and Changing Options and Settings

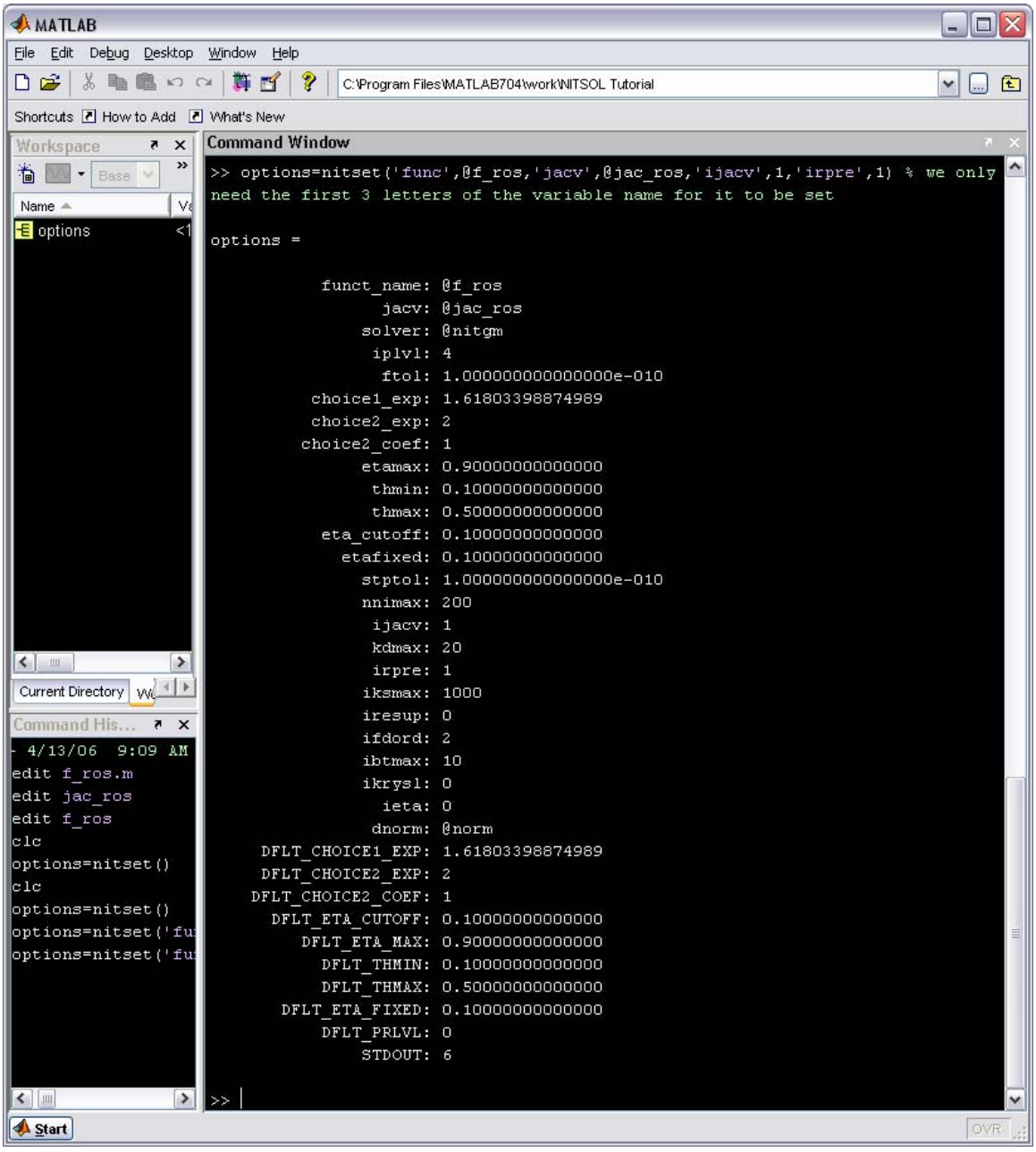
1. Type options = nitset() to set default options.



2. To change any option variables, type options = nitset('var1 name', new var1 value, 'var2 name', new var2 value, …). If the function name isn't 'f', set that function handle name here. If the Jacobian product/preconditioner routine name is not 'jacv', set that function handle name here. If variables have been changed

from the default values, they will remain changed as long as ‘options’ exists in the workspace. Type ‘clear’ and ‘options = nitset()’ to clear and reset default variables.

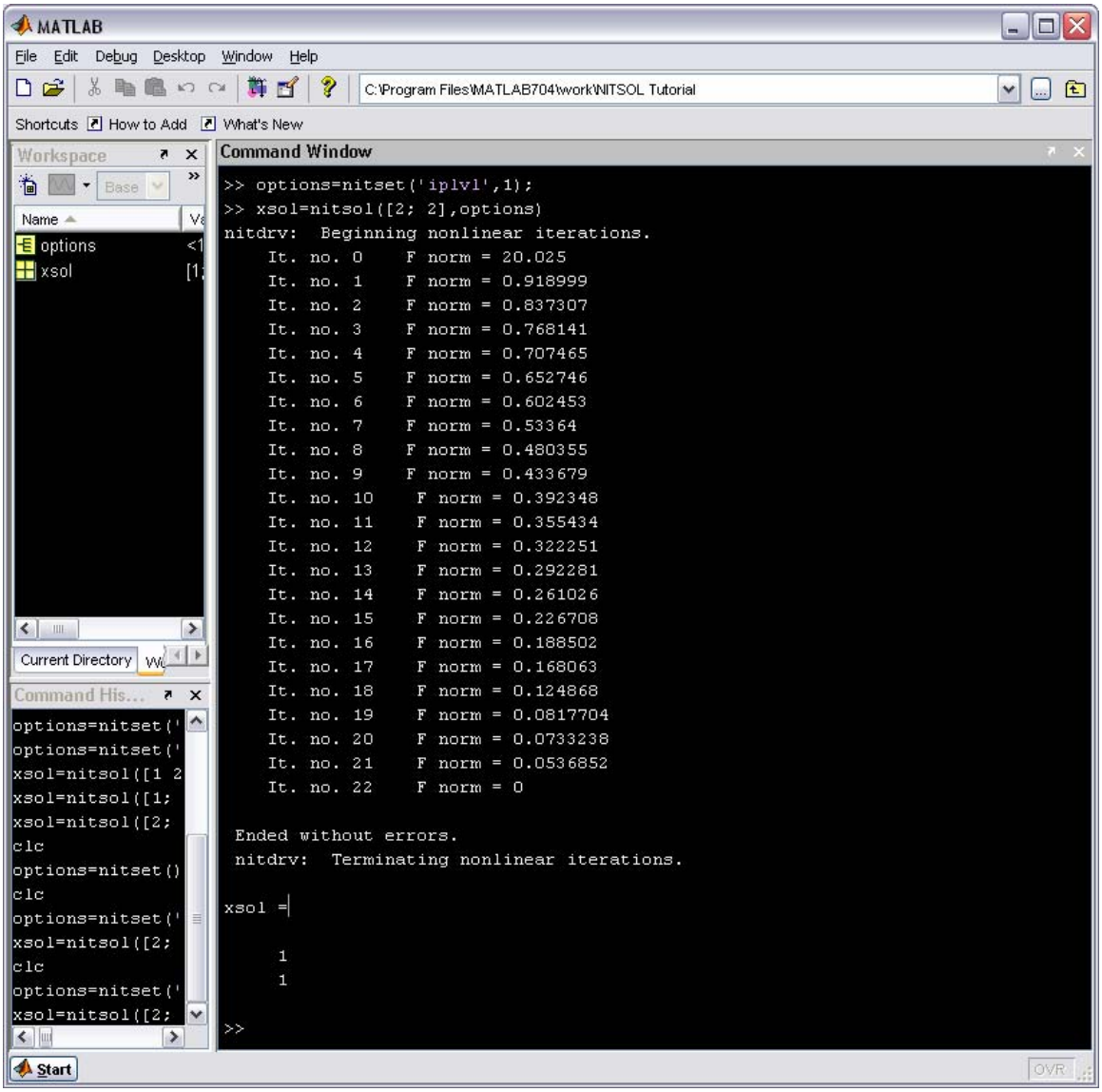
Example: In the example, the function name is ‘f_ros’ and the Jacobian/preconditioner is ‘jac_ros’. Also to use this analytic Jacobian and to use the preconditioner, we set ijacv to 1 and irpre to 1 as well.



Running Main Routine

1. Type ‘xsol = nitsol(x0, options)’ where x0 is the value of the initial guess for x. xsol can be any variable name which will be the final solution.

   Ex: For this example, set printout level to 1 so everything appears on 1 screen. Also use [2; 2] as x0.



Troubleshooting

Tips

Type ‘help’ and filename at the MATLAB prompt to get specialized help on the individual routines. In these help sections, the usage of the routines and descriptions of the input and output variables will be displayed.

Issue Reporting

Bijaya Padhy

bijaya23@gmail.com

Example Nonlinear System Problem

To show an example of a more complex nonlinear system, the 2D Bratu problem is illustrated below. The 2D Bratu problem function along with the Jacobian vector product and right preconditioner are included. Shown below are the results of NITSOL with the option of using analytic Jacobian-vector products vs. finite-difference Jacobian vector products and right preconditioner vs. no preconditioner. MATLAB's commands 'tic' and 'toc' give the execution time of a given routine.

2D Bratu problem function routine:

Editor - D:\Personal\thesis\M_Files_working_directory\4.22.06\f_bratu2d.m

File Edit Text Cell Tools Debug Desktop Window Help

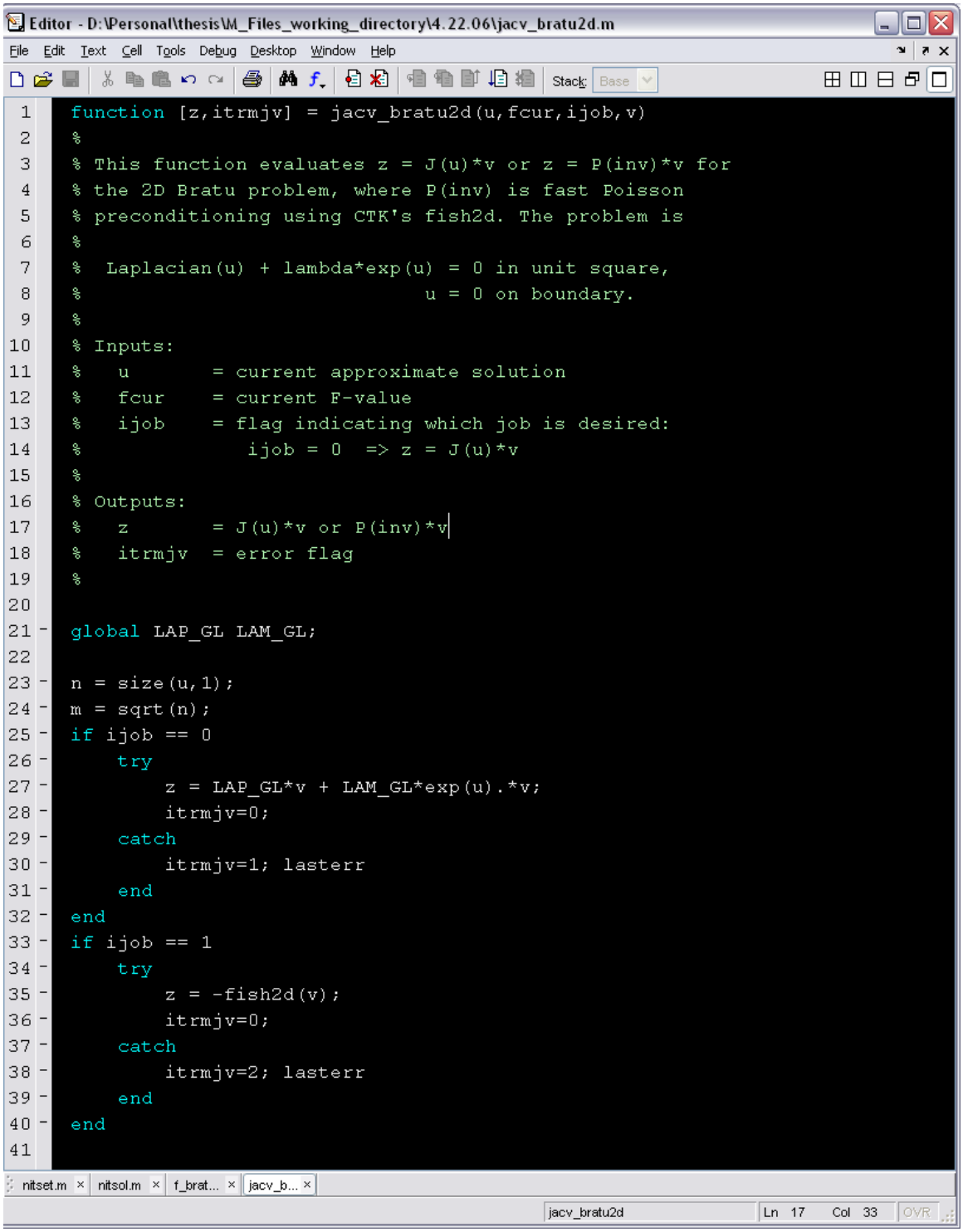
Stack: Base

```
1  function [fval,itrmf] = f_bratu2d(u)
2  %
3  % This function evaluates the residual function F(u)
4  % for the 2D Bratu problem
5  %
6  %  Laplacian(u) + lambda*exp(u) = 0 in unit square,
7  %                               u = 0 on boundary.
8  %
9  % Input:
10 %    u      = current approximate solution
11 %
12 % Outputs:
13 %    fval   = F-value F(u)
14 %    itrmf  = error flag
15 %
16
17 global LAP_GL LAM_GL;
18
19 LAM_GL = 5.0;
20
21 itrmf = 0;
22
23 n = size(u,1);
24 m = sqrt(n);
25
26 if round(m)^2 ~= n, error('Problem dimension is not a square.'); end
27
28 if ~(exist('LAP_GL')==1)
29   LAP_GL = laplacian(m);
30 else
31   if size(LAM_GL,1) ~= n
32     LAP_GL = laplacian(m);
33   end
34 end
35
36 fval = LAP_GL*u + LAM_GL*exp(u);
37
```

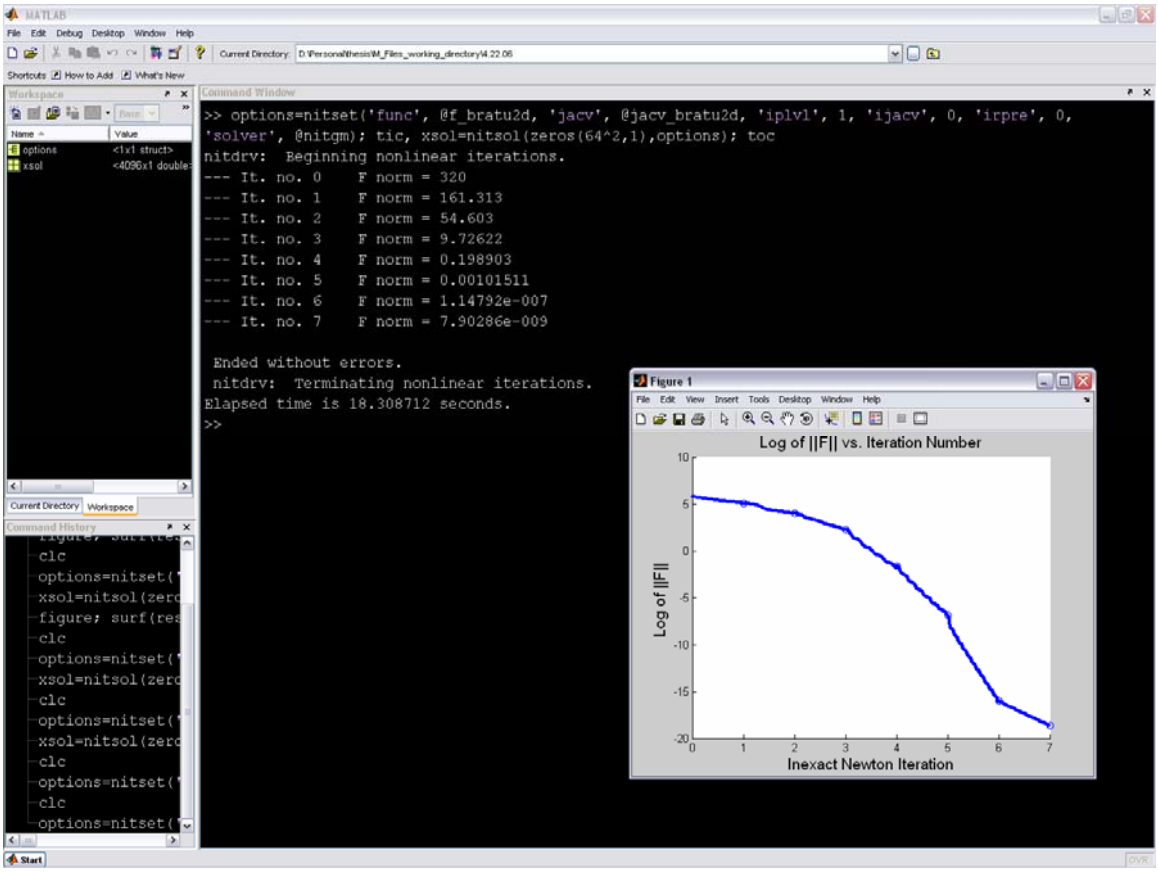
nitset.m | nitsol.m | f_brat... | jacv_b...

f_bratu2d Ln 1 Col 1 OVR

2D Bratu problem Jacobian and right preconditioning routine:

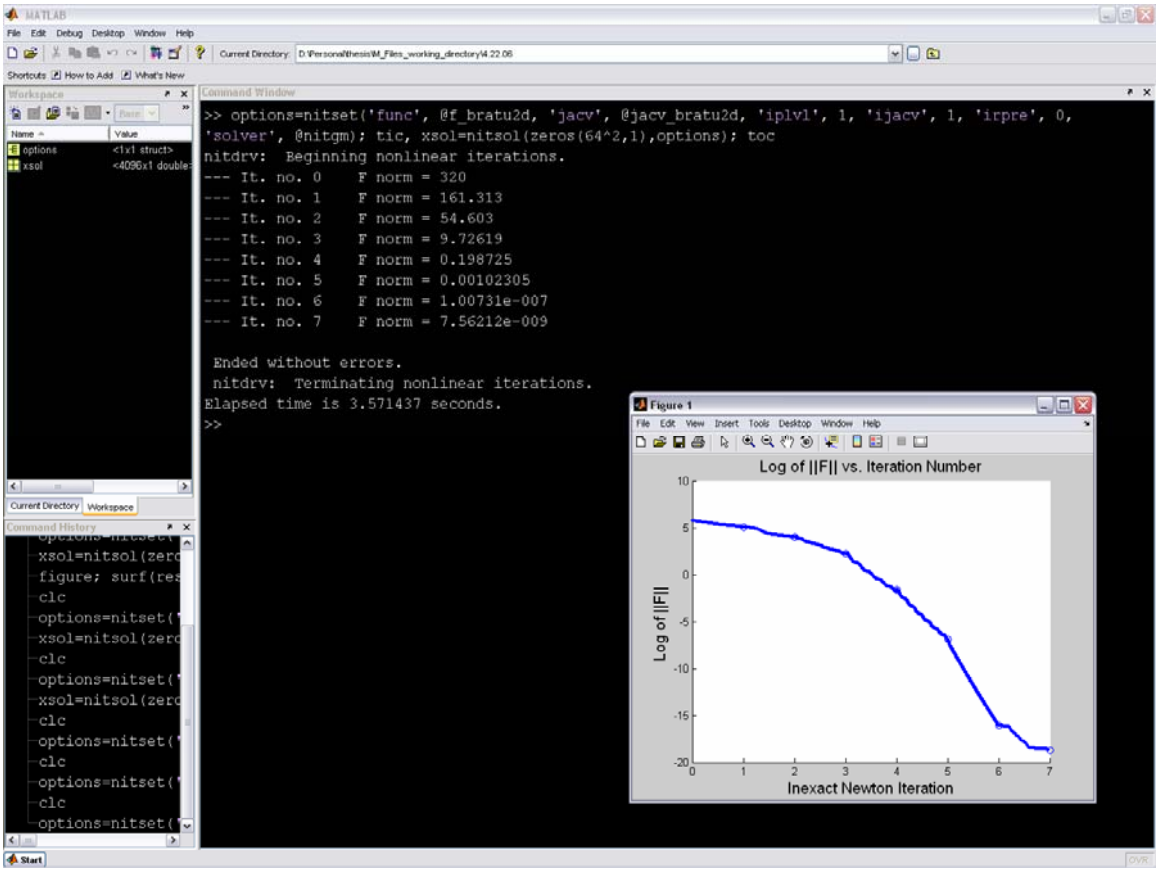


Solution using finite difference Jacobian vector products and no preconditioner:



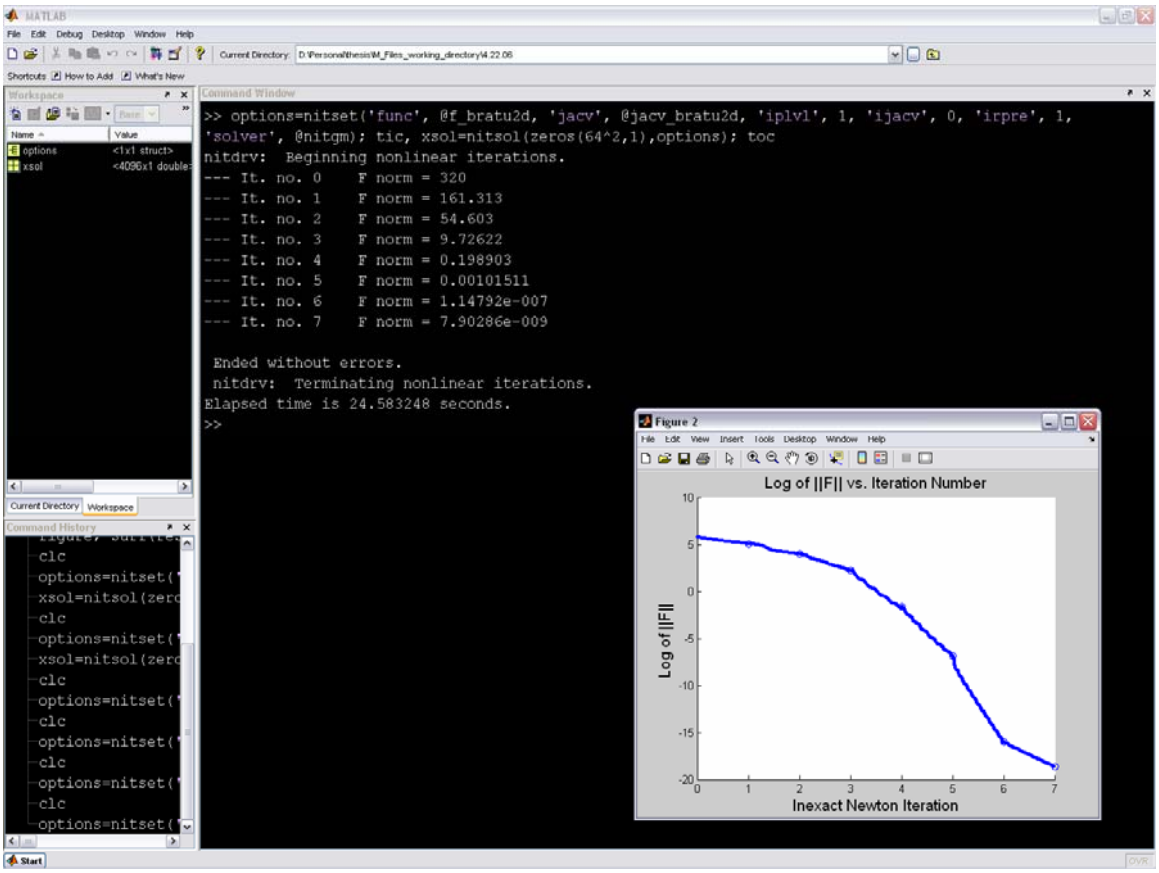
NITSOL took ~18 seconds to converge to the solution.

Solution using the given Jacobian vector product routine and no preconditioner:



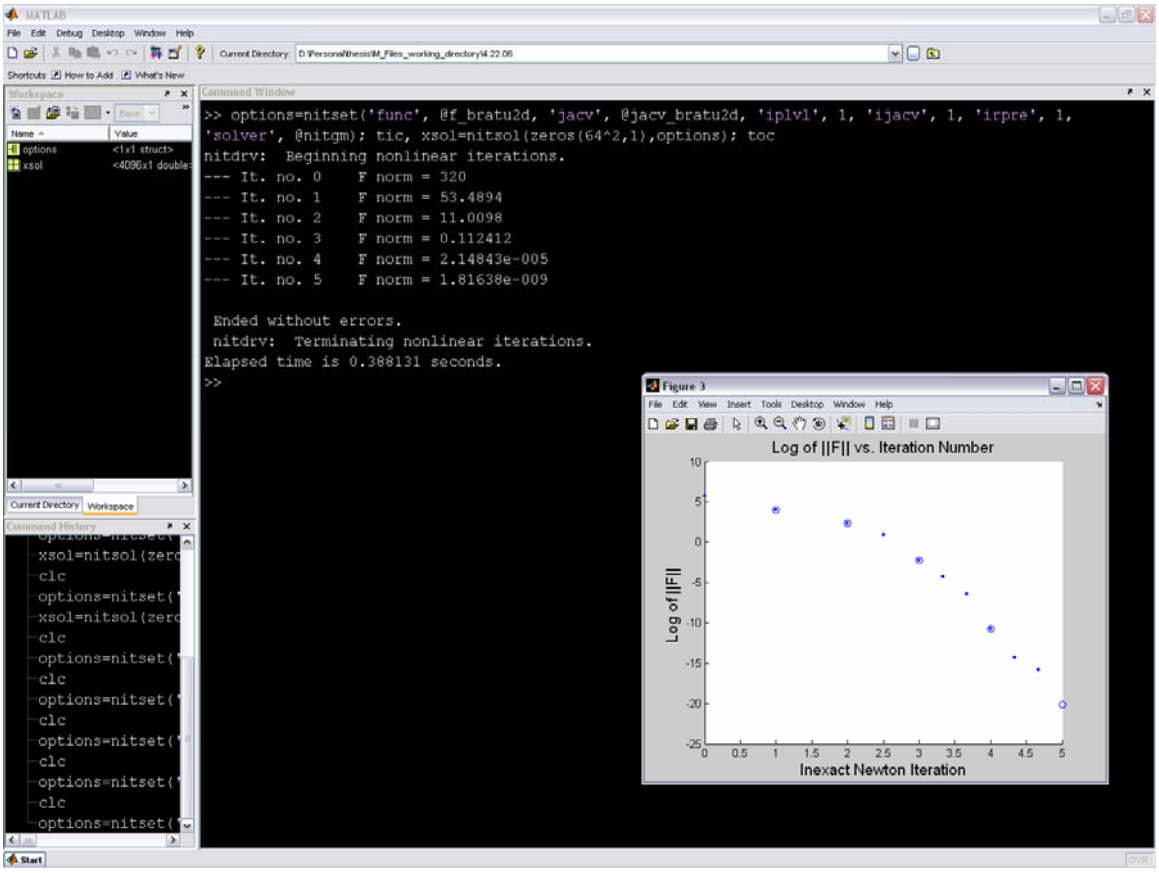
NITSOL took ~3.57 seconds to converge to the solution.

Solution using finite-difference Jacobian vector products and the given right preconditioning routine:



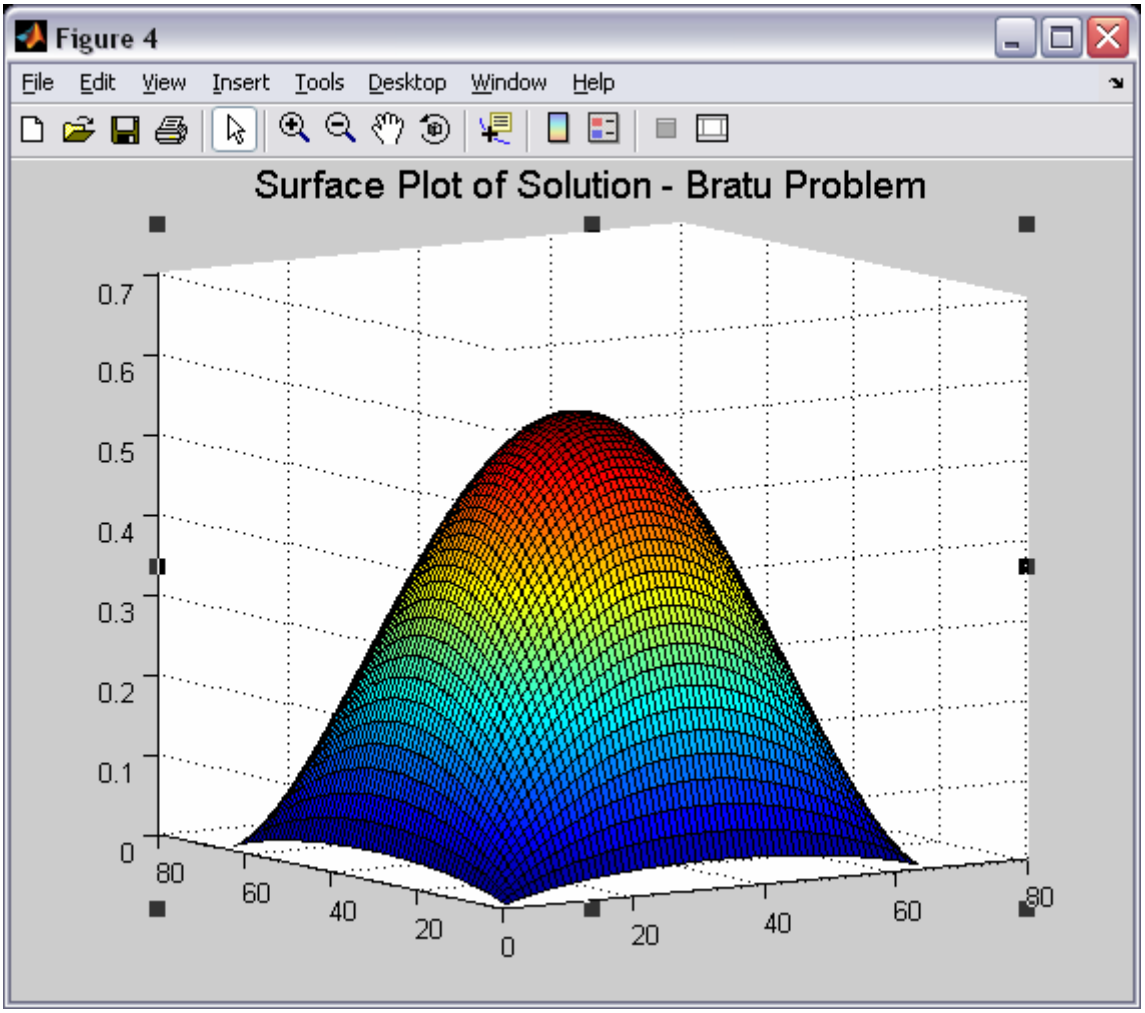
NITSOL took ~24.58 seconds to converge to the solution.

Solution using the given Jacobian vector products and the given right preconditioning routine:



NITSOL took ~0.38 seconds to converge to the solution.

To get the surface plot, type at the MATLAB prompt:
>> surf(reshape(xsol,64,64))
('reshape' will take the 4096 x 1 vector and reshape it into a 64 x 64 matrix as indicated)



Appendix

NITSET.M

```
% NISET - Sets default values in options structure.  Also allows user to
% change default values.
%
% USAGE:
% nitset
%   To see short variable descriptions and default values
%
% OR
% options = nitset
%   To set default values
%
% OR
% options = nitset('var1 name', new var1 value, 'var2 name', new var2 value,...)
%   To set new values for specific variables.  This can also be accomplished
%   by typing 'options.<var1name> = <new_var1_value>' to set new variable
%   value to a given variable.
%
% Type 'doc nitset' to see Help Document containing full edscriptions of
% all the option variables.
%
%
% Here are full descriptions of all option variables:
%
%   @f      = name of user-supplied subroutine for evaluating the function
%                the zero of which is sought; this routine has the form
%
%
%               [fcur,itrmf] =  f(xcur)
%
%               where xcur is the array containing the current x value, fcur
%               is f(xcur) on output, and itrmf is an integer termination flag.
%               The meaning of itrmf is as follows:
%                  0 => normal termination; desired function value calculated.
%                  1 => failure to produce f(xcur).
%   @jacv   = name of user-supplied subroutine for optionally evaluating J*v
%             or inv(P)*v, where J is the Jacobian of f and P is a
%             right preconditioning operator. If neither analytic J*v
%             evaluations nor right preconditioning is used, this can
%             be a dummy subroutine; if right preconditioning is used but
%             not analytic J*v evaluations, this need only evaluate
%             P(inverse)*v. The form is
%
%                [z,itrmjv] = jacv(xcur, fcur, ijob, v)
%
%             where xcur and fcur are vectors containing the current x and
%             f values, ijob is an integer flag indicating which product
%             is desired, v is a vector to be multiplied.  z is a vector
%             containing the desired product on output, and itrmjv is an
%             integer termination flag. The meaning of ijob is as follows:
%                0 => z = J*v
%                1 => z = P(inverse)*v
%             The meaning of itrmjv is as follows:
%                0 => normal termination; desired product evaluated.
%                1 => failure to produce J*v.
%                2 => failure to produce P(inverse)*v.
%             This subroutine is called only from nitjv, and is always
%             called with v ~= z.
%   ftol          = stopping tolerance on the f-norm.
%   stptol        = stopping tolerance on the steplength.
%   @dnorm  = name of user-supplied function for calculating vector norms.
%                     This function must have the form
%
%                           [xnrm] = dnorm(xcur)
%
%                    xcur is vector containing the current x value.
%   nnimax  = maximum number of nonlinear iterations (default 200).
%   ijacv   =  flag for determining the method of J*v evaluation:
%                       0 => finite-difference evaluation (nitfd) (default)
%                       1 => analytic evaluation (jacv)
%   solver  = name for determining the Krylov solver:
%                       nitgm => GMRES (default)
```

```
%                      gmres => MATLAB GMRES
%                      bicgstab => MATLAB BICGSTAB
%                      cgs => MATLAB CGS
%                   For brief descriptions of the solvers plus references,
%                   see the subroutine nitgm and MATLAB's GMRES, BICGSTAB
%                   and CGS.
%   kdmax   = maximum Krylov subspace dimension when 'nitgm' is used
%                    (default 20).
%   irpre   = flag for right preconditioning when 'nitgm' is used:
%                     0 => no right preconditioning
%                     1 => right preconditioning
%   iksmax  = maximum allowable number of iterations per call
%                   to the Krylov solver routine (default 1000).
%   iresup  = residual update flag when 'nitgm' is used; on
%                  restarts, the residual is updated as follows:
%                     0 => linear combination (default)
%                     1 => direct evaluation
%             The first is cheap (one n-vector saxpy) but may lose
%             accuracy with extreme residual reduction; the second
%             retains accuracy better but costs one J*v product per
%             restart.
%   ifdord  = order of the finite-difference formula (sometimes)
%                used when ijacv = 0. When ijacv = 0,
%                this must be 0, 1, 2, or 4 on input; otherwise, it is
%                irrelevant. With ijacv = 0, the precise
%                meaning is as follows:
%
%                If 'nitgm' is used, then ifdord matters only if iresup = 1,
%                in which case it determines the order of
%                the finite-difference formula used in evaluating the
%                initial residual at each GMRES restart (default 2); if
%                ifdord = 0 on input, then it is set to 2 below. NOTE: This
%                only affects initial residuals at restarts; first-order
%                differences are always used within each GMRES cycle. Using
%                higher-order differences at restarts only should give
%                the same accuracy as if higher-order differences were
%                used throughout; see K. Turner and H. F. Walker, "Efficient
%                high accuracy solutions with GMRES(m)," SIAM J. Sci.
%                Stat. Comput., 13 (1992), pp. 815--825.
%
%                If BiCGSTAB or CGS is used, then ifdord determines the
%                order of the finite-difference formula used at each
%                iteration (default 1); if ifdord = 0 on input, then it
%                is set to 1 below.
%   ibtmax  = maximum allowable number of backtracks (step
%             reductions) per call to nitbt (default 10).
%             USAGE NOTE: Backtracking can be turned off by setting
%             ibtmax  = -1. Other negative values of ibtmax are not
%             valid.
%   ieta    = flag determining the forcing term eta as follows:
%                0 => abs( ||fcur|| - ||fprev+Jprev*sprev|| )/||fprev||
%                     (default)
%                1 => (||fcur||/||fprev||)^2
%                2 => gamma*(||fcur||/||fprev||)^alpha
%                     for user-supplied gamma in (0,1] and alpha in (1,2]
%                3 => fixed (constant) eta in (0,1), either 0.1 (default)
%                or specified by the user (see USAGE NOTE below)
%
%             Here, fcur = current f, fprev = previous f, etc. The Krylov
%             iterations are terminated when an iterate s satisfies
%             an inexact Newton condition ||F + J*s|| <= eta*||F||.
%             USAGE NOTE: If ieta = 2, then alpha and gamma
%             must be set in common block nitparam.h as described below.
%
%             If ieta = 3, then the desired constant eta may
%             be similarly set if a value other than the
%             default of 0.1 is desired.
%
%             The first three expressions above are from S. C. Eisenstat
%             and H. F. Walker, "Choosing the forcing terms in an inexact
%             Newton method", SIAM J. Scientific Computing, 17 (1996),
%             pp. 16--32. (They may be modified according to certain
%             safeguards in subroutine nitdrv.) The first gives convergence
%             that is q-superlinear and of r-order (1+sqrt(5))/2. The
%             second gives convergence that is r-quadratic and of q-order
%             p for every p in [1,2). The third gives convergence that is
%             of q-order alpha when gamma < 1 and, when gamma = 1, of
%             r-order alpha and q-order p for every p in [1,alpha). The
%             fourth gives q-linear convergence with asymptotic rate
%             constant eta in a certain norm; see R. S. Dembo, S. C.
%             Eisenstat, and T. Steihaug, "Inexact Newton methods",
```

```
%                SIAM J. Numer. Anal., 18 (1982), pp. 400-408.
%
%             Of these four choices, the 1st is usually satisfactory,
%             the 2nd or 3rd is sometimes preferred, and the 4th may be
%             useful in some situations, e.g., it may be desirable to
%             choose a fairly large fixed eta in (0,1), such as eta = .1,
%             when numerical inaccuracy prevents the Krylov solver
%             from obtaining much residual reduction.
%   iplvl   = 0 => no printout
%             1 => iteration numbers and F-norms
%             2 => ... + some stats, step norms, and linear model norms
%             3 => ... + some Krylov solver and backtrack information
%                 (default)
%             4 => ... + more Krylov solver and backtrack information
%
% FOR ADVANCED USERS:
%
%     choice1_exp  = parameter used in the update of the forcing term
%                    eta when ieta = 0 (default).  This is the exponent
%                    for determining the etamin safeguard.  The default
%                    value is choice1_exp = (1+sqrt(5))/2.  A larger
%                    value will allow eta to decrease more rapidly,
%                    while a smaller value will result in a larger
%                    value for the safeguard.
%     choice2_exp  = parameter used in the update of the forcing term
%                    eta when ieta = 2.  This is the exponent alpha
%                    in the expression gamma*(||fcur||/||fprev||)^alpha;
%                    it is also used to determine the etamin safeguard.
%                    The default value is 2.  Valid values are in the
%                    range (1, 2].
%     choice2_coef = parameter used in the update of the forcing term eta
%                    when ieta = 2.  This is the coefficient gamma used
%                    in the expression gamma*(||fcur||/||fprev||)^alpha;
%                    it is also used to determine the etamin safeguard.
%                    The default value is 1.  Valid values are in the
%                    range (0, 1].
%     eta_cutoff   = parameter used to determine when to disable
%                    safeguarding the update of the forcing term.  It
%                    only has meaning when ieta ~= 3.  The default
%                    value is 0.1.  A value of 0 will enable
%                    safeguarding always; a value of 1 will disable
%                    safeguarding always.
%     etamax       = parameter used to provide an upper bound on the
%                    forcing terms when ieta ~= 3. This is
%                    necessary to ensure convergence of the inexact Newton
%                    iterates and is imposed whenever eta would otherwise
%                    be too large. (An overly large eta can result from
%                    the updating formulas when ieta ~= 3 or from
%                    safeguarding when the previous forcing term has been
%                    excessively increased during backtracking.) The
%                    default value of etamax is 0.9.  When
%                    backtracking occurs several times during a nonlinear
%                    solve the forcing term can remain near etamax for several
%                    nonlinear steps and cause the nonlinear iterations
%                    to nearly stagnate.  In such cases a smaller value of
%                    etamax may prevent this.  Valid values are in the
%                    range (0, 1).
%     etafixed     = this is the user-supplied fixed eta when ieta = 3.
%                    The  default value is etafixed = 0.1.  Valid values
%                    are in the range (0,1).
%     thmin        = when backtracking occurs, this is the smallest
%                    reduction factor that will be applied to the current
%                    step in a single backtracking reduction.  The default
%                    value is 0.1.  Valid  values are in the range
%                    [0, thmax].
%     thmax        = when backtracking occurs, this is the largest
%                    reduction factor that will be applied to the current
%                    step in a single backtracking reduction.  The default
%                    value is 0.5.  Valid values are in the range
%                    [thmin, 1).
%
% See also nitset, nitsol, nitgm, nitjv, nitfd, nitbt, bicgstab, cgs
%
% Bijaya Zenchenko, May 2006
% Reference: Homer Walker, NITSOL
%

%% Function to set and change option variables
function opt_disp = nitset(varargin)

global options
```

```
%% Print out possible values of properties.
if (nargin == 0) && (nargout == 0)
    fprintf('           iplvl: [ output level desired: 0 | 1 | 2 | {3} | 4 ]\n');
    fprintf('      funct_name: [ function {@f} ]\n');
    fprintf('            jacv: [ user-supplied function handle for evaluatiing J*v
{@jacv} ]\n');
    fprintf('          solver: [ Krylov solver function handle: {@nitgm} | @bicgstab |
@cgs ]\n');
    fprintf('           kdmax: [ maximum Krylov subspace if solver is @nitgm {20} ]\n');
    fprintf('          iresup: [ residual update flag when solver is @nitgm {0} ]\n');
    fprintf('           dnorm: [ user-supplied function handle for norm {MATLAB @norm}
]\n');
    fprintf('            ftol: [ stopping tolerance on the f-norm {10^-8} ]\n');
    fprintf('          stptol: [ stopping tolerance on the steplength {10^-8} ]\n');
    fprintf('          nnimax: [ maximum number of nonlinear iterations {200} ]\n');
    fprintf('           ijacv:   flag for determining method of J*v evaluation: \n
\t\t\t\t\t [ {0} - finite difference | 1 - analytic with @jacv ]\n');
    fprintf('           irpre:   flag for right preconditioning: \n \t\t\t\t\t [ {0} - no
right preconditioning | 1 - right preconditioning ]\n');
    fprintf('          iksmax: [ maximum number of iterations per call to Krylov Solver
{1000} ]\n');
    fprintf('          ifdord: [ 0 function tolerance {10^-4} ]\n');
    fprintf('          ibtmax:   maximum number of backtracks per call to @nitbt: \n
\t\t\t\t\t [ {10} | any positive integer | -1 - to turn off backtracking ]\n');
    fprintf('            ieta: [ flag determining the forcing term eta : {0} ]\n\n');
    fprintf(' FOR ADVANCED USERS: \n');
    fprintf('     choice1_exp: [ used in updating eta when ieta = 0. {(1 + sqrt(5))/2}
]\n');
    fprintf('     choice2_exp: [ used in updating eta when ieta = 2. This is the exponent
alpha \n \t\t\t\t\t in gamma*(||fcur||/||fprev||)^alpha. {2}] \n');
    fprintf('    choice2_coef: [ used in updating eta when ieta = 2. This is the exponent
gamma \n \t\t\t\t\t in gamma*(||fcur||/||fprev||)^alpha. {1}] \n');
    fprintf('      eta_cutoff: [ if ieta ~= 3, when to disable safeguarding the \n
\t\t\t\t\t update of the eta. {0.1} ]\n');
    fprintf('          etamax: [ provides an upper bound on eta when ieta ~= 3. {0.9}
]\n');
    fprintf('        etafixed: [ user-supplied fixed eta when ieta = 3. {0.1} ]\n');
    fprintf('           thmin: [ when backtracking occurs, this is the smallest reduction
factor \n \t\t\t\t\t applied to the current step in a single reduction {0.1} ]\n');
    fprintf('           thmax: [ when backtracking occurs, this is the largest reduction
factor \n \t\t\t\t\t applied to the current step in a single reduction {0.5} ]\n');
    fprintf('\n');
    if isempty(options)
        opt_disp = 'Set by ''options = nitset''.  Type ''help nitset'' for complete usage
help.';
    else
        opt_disp = '''options'' already exists. Type ''options = nitset'' to see current
settings. Type ''help nitset'' for complete usage help.';
    end

    return;
end


%% Sets initial options structure
if isempty(options)
    options = struct('iplvl',3,...
        'funct_name',@f,...
        'jacv',@jacv,...
        'solver',@nitgm,...
        'kdmax',20,...
        'iresup',0,...
        'dnorm',@norm,...
        'ftol',10^-8, ...
        'stptol',10^-8,...
        'nnimax',200,...
        'ijacv',0,...
        'irpre',0,...
        'iksmax',1000,...
        'ifdord',0,...
        'ibtmax',10,...
        'ieta',0,...
        'choice1_exp',(1 + sqrt(5))/2,...
        'choice2_exp',2,...
        'choice2_coef',1,...
        'etamax',0.9,...
        'eta_cutoff',0.1,...
        'etafixed',0.1,...
        'thmin',0.1,...
        'thmax',0.5);
```

```
end

Names = {'funct_name',...
    'solver',...
    'iplvl',...
    'ftol', ...
    'choice1_exp',...
    'choice2_exp',...
    'choice2_coef',...
    'etamax',...
    'thmin',...
    'thmax',...
    'eta_cutoff',...
    'etafixed',...
    'stptol',...
    'jacv',...
    'dnorm',...
    'nnimax',...
    'ijacv',...
    'kdmax',...
    'irpre',...
    'iksmax',...
    'iresup',...
    'ifdord',...
    'ibtmax',...
    'ieta'};

%% Change all variables that are requested by user
for i=1:2:nargin

    if ~ischar(varargin{i})
        error('Expect 1st argument to be a string property name.');
    end
    if (~strncmp(varargin{i},'jacv',3) && ~strncmp(varargin{i},'dnorm',3) && ...
            ~strncmp(varargin{i}, 'dinpr',3) && ~strncmp(varargin{i}, 'funct_name',3) ...
            && ~strncmp(varargin{i}, 'solver',3)) ...
            && ~isnumeric(varargin{i+1})
        error('Expect 2nd argument to be a numeric value.');
    end
    if (strncmp(varargin{i},'jacv',3) || strncmp(varargin{i},'dnorm',3) || ...
            strncmp(varargin{i}, 'dinpr',3) && strncmp(varargin{i}, 'funct_name',3)...
            && strncmp(varargin{i}, 'solver',3))...
            && ~isa(varargin{i+1}, 'function_handle')
        error('Expect 2nd argument to be a function handle.');
    end

    if ischar(varargin{i})
        arg = varargin{i};
        val = i+1;
    else
        arg = varargin{i+1};
        val = i;
    end

    j = strncmpi(arg,Names,3);
    match_at=find(j);
    if isempty(match_at)                        % if no matches
        error('Unrecognized property name or can not change ''%s''.', arg);
    elseif length(match_at) > 1                 % if more than one match
        % Check for any exact matches (in case any names are subsets of others)
        k = strmatch(arg,Names,'exact');
        if length(k) == 1
            j = k;
        else
            msg = sprintf('Ambiguous property name ''%s'' ', arg);
            msg = [msg '(' Names{match_at(1)}];
            for k = match_at(2:length(match_at))
                msg = [msg ', ' Names{k}];
            end
            msg = sprintf('%s).', msg);
            error(msg);
        end
    end
    options.(Names{j}) = varargin{val};
end

%% Input error checking
if  ~strcmpi(func2str(options.solver),'nitgm')...
        && ~strcmpi(func2str(options.solver),'bicgstab') ...
        && ~strcmpi(func2str(options.solver),'gmres') ...
        && ~strcmpi(func2str(options.solver),'cgs')
```

```
    give_error('solver');
end

if options.nnimax <= 0
    give_error('nnimax');
end

if options.ijacv ~= -1 &&  options.ijacv ~= 0 && options.ijacv ~= 1
    give_error('ijacv');
end

if options.kdmax <= 0
    give_error('kdmax');
end

if strcmpi(func2str(options.solver),'nitgm')
    if options.irpre ~= 0 && options.irpre ~= 1
        give_error('irpre');
    end
elseif options.irpre == 1
    error('Can only set right preconditioning if using solver ''nitgm''.');
end


if options.iksmax <= 0
    give_error('iksmax');
end

if options.iresup ~= 0 && options.iresup ~= 1
    give_error('iresup');
end

if options.ijacv == 0
    if options.ifdord == 0
        if strcmpi(func2str(options.solver),'nitgm')
            options.ifdord = 2;
        else
            options.ifdord = 1;
        end
    elseif options.ifdord ~= 1 && options.ifdord ~= 2 && options.ifdord ~= 4
        give_error('ifdord');
    end
end

if options.ibtmax <= 0 && options.ibtmax ~= -1
    give_error('ibtmax');
end

if options.ieta < 0 || options.ieta > 3
    give_error('ieta');
end

% All these above will give error and exit out of program

%% ----------------------------------------
% These are the default values that are set if the advanced user variables
% are set incorrectly.
dflt_values = struct('DFLT_CHOICE1_EXP',(1 + sqrt(5))/2,...
    'DFLT_CHOICE2_EXP',2,...
    'DFLT_CHOICE2_COEF',1,...
    'DFLT_ETA_CUTOFF',.1,...
    'DFLT_ETA_MAX',0.9,...
    'DFLT_THMIN',0.1,...
    'DFLT_THMAX',0.5,...
    'DFLT_ETA_FIXED',0.1,...
    'DFLT_PRLVL',0);

%% ------------------------------------------------------------------------
%   Check possible invalid value for printout level.  In
%   case the value is invalid the default is restored.
%  ------------------------------------------------------------------------
if ( options.iplvl < 0 || options.iplvl > 4 )
    options.iplvl = dflt_values.DFLT_PRLVL;
end

%% ------------------------------------------------------------------------
%   Check possible invalid values for various parameters.  In
%   case the values are invalid the defaults are restored.
%  ------------------------------------------------------------------------
if ( options.choice1_exp <= 1.0   || options.choice1_exp > 2.0   )
    options.choice1_exp = dflt_values.DFLT_CHOICE1_EXP; end
```

```
if ( options.choice2_exp <= 1.0   || options.choice2_exp > 2.0   )
    options.choice2_exp = dflt_values.DFLT_CHOICE2_EXP; end
if ( options.choice2_coef < 0.0   || options.choice2_coef > 1.0   )
    options.choice2_coef = dflt_values.DFLT_CHOICE2_COEF; end
if ( options.etamax <= 0.0  )
    options.etamax = dflt_values.DFLT_ETA_MAX;
end
if ( options.thmin < 0.0   || options.thmin > options.thmax )
    options.thmin = dflt_values.DFLT_THMIN; end
if ( options.thmax > 1.0   || options.thmax < options.thmin )
    options.thmax = dflt_values.DFLT_THMAX; end

%% Error is given
    function give_error(input_term)
        error(strcat('Illegal value for input number/type', ' ',input_term));
    end % end of function give_error()

    opt_disp = options;

end % end of function nitset
```

NITSOL.M

```
% NITSOL - This function solves a nonlinear system given the initial guess
%          for x for the function defined in options.funct_name.  Need to
%          set options = nitset() accordingly prior to executing this
%          function.
%
%          This is the basic algorithm is an inexact Newton method with
%          a backtracking globalization; the model is Algorithm INB of
%          S. C. Eisenstat and H. F. Walker, "Globally convergent
%          inexact Newton methods", SIAM J. Optimization, 4 (1994),
%          pp. 393--422. Initial inexact Newton steps are obtained by
%          approximately solving the Newton equation with a transpose-free
%          Krylov subspace method; the current choices are GMRES,
%          BiCGSTAB, and CGS. Jacobian-vector products are evaluated
%          either by finite-difference approximation or a user-supplied
%          analytic-evaluation subroutine. An option is provided for
%          user-supplied right preconditioning. Left preconditioning is
%          not explicitly included as an option, but the user may
%          provide this in the subroutines for evaluating the function
%          and Jacobian-vector products. Various algorithmic options
%          can be selected in the options vector. Optional common
%          blocks are also available for printing diagnostic
%          information, passing information about the nonlinear
%          iterations to user subroutines, and controlling the behavior
%          of the nonlinear iterations. Summary statistics are provided
%          by the output vector on output.
%
% USAGE:
% [xcur, output, iterm] = nitsol(xcur, options)
%
% INPUT:
% xcur    = initial guess i.e. x0.  Same size as solution vector/matrix.
% options = 'options' structure created by options = nitset().  NITSOL can
%           not run without this value set.
%
% OUTPUT:
% xcur    = final solution.
% output  = a structure containing all the counter variables:
%              nfe   - number of function evaluations.
%              njve  - number of J*v evaluations.
%              nrpre - number of inverse(P)*v evaluations.
%              nli   - number of linear iterations.
%              nni   - number of nonlinear iterations.
%              nbt   - number of backtracks.
% iterm   = termination flag; values have the following meanings:
%              0 => normal termination: ||F||<= ftol or ||step||<= stptol.
%              1 => nnimax nonlinear iterations reached without success.
%              2 => failure to evaluate F.
%              3 => in nitjv, J*v failure.
%              4 => in nitjv, inverse(P)*v failure.
%              5 => in nitsol, insufficient initial model norm reduction
%                   for adequate progress. NOTE: This can occur for several
%                   reasons; examine itrmks on return from the Krylov
%                   solver for further information.
%              6 => in nitbt, failure to reach an acceptable step through
%                   backtracking.
%
% See also nitset, nitsol, nitgm, nitjv, nitfd, nitbt, bicgstab, cgs
%
% Bijaya Zenchenko, May 2006
% Reference: Homer Walker, nitsol.f, NITSOL
%

%% Function NITSOL: A Newton Interative Solver for Nonlinear Systems

function  [xcur, output, iterm] = nitsol(xcur, options)

global output

X = []; % initialization for graphing
figure;

if nargin ~= 2
    error('There must be 2 arguments only: initial guess , ''options''.');
end
```

```
output = struct('nfe',  0, ...
    'njve', 0, ...
    'nrpre',0, ...
    'nli',  0, ...
    'nni',  0, ...
    'nbt',  0);

%% ------------------------------------------------------------------------
%  Initialize.
%  ------------------------------------------------------------------------
epsmach = 2*eps;
avrate = 1;
if (options.ieta == 0)
    alpha = options.choice1_exp; end
if (options.ieta == 2)
    alpha = options.choice2_exp; end
if (options.ieta == 2)
    gamma = options.choice2_coef; end
if (options.ieta == 3)
    eta = options.etafixed;
else eta = 0.5;
end

%% ------------------------------------------------------------------------
%  Evaluate f at initial x.
%  ------------------------------------------------------------------------
[fcur, itrmf] = options.funct_name(xcur);
if ( itrmf ~= 0 )
    iterm = 2;
    go_to_end();
    return;
end
output.nfe = output.nfe + 1;
fcnrm = norm(fcur);

%% ------------------------------------------------------------------------
%  For printing:
if (options.iplvl >= 1)
    fprintf('\nnitdrv:  Beginning nonlinear iterations.\n');
end

%% Start Loop here:
while (output.nni <= options.nnimax)
    if (options.iplvl > 0)
        fprintf('--- It. no. %g    F norm = %g  ', output.nni, fcnrm);
        if (options.iplvl > 1)
            fprintf(' ------------------------------------------ \n');
            if (options.iplvl > 2)
                fprintf(' CURRENT TOTALS:\t nfe = %g \t njve = %g \t nrpre = %g \t nli =
%g \n', ...
                    output.nfe, output.njve, output.nrpre, output.nli);
            end
        else fprintf('\n');
        end
    end
    infe = output.nfe;
    injve = output.njve;
    inrpre = output.nrpre;
    inli = output.nli;

    %  ------------------------------------------------------------------------
    %  Test for stopping.
    % ------------------------------------------------------------------------
    if  (fcnrm <= options.ftol)
        iterm=0;
        go_to_end();
        return;
    end
    if (output.nni > 0) &&(stpnrm <= options.stptol) &&(itrmks == 0)
        iterm=0;
        go_to_end();
        return;
    end
    if (output.nni >= options.nnimax)
        iterm=1;
        go_to_end();
        return;
    end

    %  ------------------------------------------------------------------------
    %  Compute the (trial) inexact Newton step with the Krylov solver.
```

```
    %  Update data in nitinfo to mark the start of a new inexact Newton step.
    %  -------------------------------------------------------------------------
    newstep = 0;
    fcurnrm = fcnrm;

    %  -------------------------------------------------------------------------
    %  Compute the original value of options.funct_name(transpose)*Js for backtracking;
the
    %  original value of options.funct_name(transpose)*(linear model) is also computed
for
    %  later use. NOTE: rsvec contain the residual
    %  vector for the Newton equation, which is -(linear model).
    % -------------------------------------------------------------------------
    if strcmpi(func2str(options.solver),'nitgm')
        [rsnrm, rsvec, instep, itrmks] = nitgm(xcur, options.funct_name, fcur, fcnrm,
...
            eta, options.jacv, options.ijacv, options.irpre, options.iksmax,
options.iresup, ...
            options.ifdord, options.kdmax, options.iplvl);
        oftlm =   -fcur'*rsvec;
        oftjs = oftlm - fcnrm^2;
    else
        itask = 0;
        afun = @(v)nitjv(xcur, options.funct_name, fcur, options.jacv, ...
            options.ijacv, options.ifdord, v, itask);
        if strcmpi(func2str(options.solver),'gmres')
            [instep, flag, relres, iter, resvec] = options.solver(afun, -fcur, [], eta);
        else
            [instep, flag, relres, iter, resvec] = options.solver(afun, -fcur, eta);
        end
        output.nli = output.nli + iter(length(iter));
        ojs=afun(instep);
        oftjs=fcur'*ojs;
        oftlm=oftjs+fcnrm^2;
        rsnrm = resvec(length(resvec));
        itrmks = 0;
    end

    %  -------------------------------------------------------------------------
    %  Set values in nitinfo that reflect state of iterative solver.
    %  -------------------------------------------------------------------------
    krystat = itrmks;
    avrate = (rsnrm/fcnrm)^(1/output.nli);

    %  -------------------------------------------------------------------------
    %  Check itrmks and decide whether to terminate or continue:
    %        0 => continue, inexact Newton condition successfully met
    %    1 or 2 => terminate unconditionally, J*v or P(inverse)*v failure)
    %    >= 3   => terminate if the model norm has increased or if reduction at
    %              the current rate would at best require more than 1000 time
    %              the maximum remaining number of nonlinear iterations.
    %  -------------------------------------------------------------------------
    if (itrmks == 1 ||itrmks == 2)
        iterm = itrmks + 2;
        go_to_end();
        return;
    end

    if (itrmks >= 3)
        if (rsnrm/fcnrm >= 1)
            iterm = 5;
            go_to_end();
            return;
        else
            temp = log(options.ftol/fcnrm)/log(rsnrm/((1 + 10*epsmach)*fcnrm));
            if (temp > 1000*(options.nnimax - output.nni))
                iterm = 5;
                go_to_end();
                return;
            end
        end
    end

    %  -------------------------------------------------------------------------
    %  Determine an acceptable step via backtracking.
    %  -------------------------------------------------------------------------
    [instep, eta, xpls, fpls, fpnrm, redfac, ibt, itrmbt]  = nitbt(xcur, ...
        options.funct_name, fcnrm, instep, eta, oftjs, options.ibtmax, ...
        options.iplvl, options.thmin, options.thmax );

    if (itrmbt == 1)
```

```
        iterm = 6;
        go_to_end();
        return;
    elseif (itrmbt == 2)
        iterm = 2;
        go_to_end();
        return;
    end
    output.nbt = output.nbt + ibt;

    %  ------------------------------------------------------------------------
    %  Set eta for next iteration.
    %  ------------------------------------------------------------------------
    if (options.ieta == 0)
        etamin = eta^alpha;
        temp = 1 - redfac;
        flmnrm = sqrt((temp*fcnrm)^2 + 2*redfac*temp*oftlm + (redfac*rsnrm)^2);
        eta = abs(fpnrm - flmnrm)/fcnrm;
    elseif (options.ieta == 1)
        etamin = eta^2;
        eta = (fpnrm/fcnrm)^2;
    elseif (options.ieta == 2)
        etamin = gamma*eta^alpha;
        eta = gamma*(fpnrm/fcnrm)^alpha;
    elseif (options.ieta == 3)
        eta = options.etafixed;
    end

    if (options.ieta ~= 3)
        if (etamin <= options.eta_cutoff)
            etamin = 0; end
        if (eta < etamin)
            eta = etamin; end
        if (eta > options.etamax)
            eta = options.etamax; end
        if (eta*fpnrm <= 2*options.ftol)
            eta = (.8*options.ftol)/fpnrm; end
    end

    %  ------------------------------------------------------------------------
    %  Update xcur, fcur, fcnrm, stpnrm, output.nni for next iteration.
    %  ------------------------------------------------------------------------
    xcur = xpls;
    fcur = fpls;

    fcnrm = fpnrm;
    stpnrm = norm(instep);
    output.nni = output.nni + 1;

    %  ------------------------------------------------------------------------
    %  For printing:
    if (options.iplvl >= 2)
        infe = output.nfe - infe;
        injve = output.njve - injve;
        inrpre = output.nrpre  - inrpre;
        inli = output.nli - inli;
        if (options.ieta > 0)
            temp = 1 - redfac;
            flmnrm = sqrt((temp*fcnrm)^2 + 2*redfac*temp*oftlm + (redfac*rsnrm)^2);
        end
        fprintf('\n At this step: \t\t nfe = %g \t njve = %g \t nrpre = %g \t nli = %g
\n', ...
                infe, injve, inrpre, inli);
        fprintf('   \t\t\t\t\t stepsize norm = %g \n   \t\t\t\t\t final lin. model norm =
%g \n', stpnrm, flmnrm);
    end

    %  ------------------------------------------------------------------------
    %  Return to top of loop for next iteration.
    %  ------------------------------------------------------------------------

    if fcnrm ~= 0
        Y(output.nni) = log(fcnrm); % for plotting
        X(output.nni) = output.nni;
    end

end % of while loop

go_to_end();
return;
```

```
%% ------------------------------------------------------------------------
%  All returns made here.
%  ------------------------------------------------------------------------

    function go_to_end()
        switch iterm
            case 0
                fprintf('\nEnded without errors.');
            case 1
                error('nnimax nonlinear iterations without success.');
            case 2
                error('Failure to evaluate F.');
            case 3
                error('In nitjv, J*v failure.');
            case 4
                error('In nitjv, P(inverse)*v failure.');
            case 5
                error('In nitdrv, insufficient initial model norm reduction for adequate
progress. NOTE: This can occur for several reasons; examine itrmks on return from the
Krylov solver for further information. (This will be printed out if iplvl >= 3).');
            case 6
                error('In nitbt, failure to reach an acceptable step through
backtracking.');
        end

        %  ------------------------------------------------------------------------
        %%  For printing:
        if (options.iplvl >= 1)
            fprintf('\nnitdrv:  Terminating nonlinear iterations.\n\n');
        end

        if ~isempty(X)
            hold on
            plot(X,Y,'o');
            title('Log of ||F|| vs. Iteration Number', 'FontSize',14);
            xlabel('Inexact Newton Iteration','FontSize',14);
            ylabel('Log of ||F||','FontSize',14);
        end
    end % end of go_to_end()

end % end of function nitsol
```

NITFD.M

```
% NITFD - This is  the routine for finite-difference evaluation of
%         products z = J*v, where J is the Jacobian of F.
%
% USAGE:
% [z, itrmjv] = nitfd(xcur, f, fcur, ijacv, ifdord, v)
%
% INPUT:
% xcur   = current value of x
% f      = function name handle of F
% fcur   = F(xcur) value
% ijacv  = flag for determining method of J*v evaluation. In this
%          subroutine, this should be 0 or -1 on input, as follows:
%              -1 => finite-difference evaluation of order ifdord.
%               0 => first-order finite-difference evaluation.
% ifdord = order of the finite-difference formula used when ijacv = -1.
%          Either 0, 1, or 2.
% v      = vector by which approximation of J is multiplied by
%
% OUTPUT:
% z      = final product of J*v
% itrmjv = termination flag; values have the following meanings:
%              0 => normal termination; desired product evaluated.
%              1 => failure to produce J*v.
%
% See also nitset, nitsol, nitgm, nitjv, nitfd, nitbt, bicgstab, cgs
%
% Bijaya Zenchenko, May 2006
% Reference: Homer Walker, nitfd.f, NITSOL
%

%% ----------------------------------------------------------------------
%  This is nitfd, the routine for finite-difference evaluation of
%  products z = J*v, where J is the Jacobian of f.
%  ----------------------------------------------------------------------

function [z, itrmjv] = nitfd(xcur, f, fcur, ijacv, ifdord, v)

global output

z = zeros(length(xcur),1);

%% ----------------------------------------------------------------------
%  Set epsmach (machine epsilon) on first call.
%  ----------------------------------------------------------------------
epsmach = 2.0*eps;

%% ----------------------------------------------------------------------
%  Compute z = J*v by finite-differences: First, set eps2 = ||v||for later
%  use in computing the difference step; then evaluate the difference
%  formula according to ijacv and ifdord.
%  ----------------------------------------------------------------------
eps2 = norm(v);
if (eps2 == 0)
    itrmjv = 1;
    return;
end

%% ----------------------------------------------------------------------
%  Here ijacv = 0 or ifdord = 1 => first-order forward difference.
%  ----------------------------------------------------------------------
if ijacv== 0 || ifdord == 1
    eps2 = sqrt((1 + norm(xcur))*epsmach)/eps2;
    v = xcur + eps2*v;
    [z, itrmf] = f(v);
    if (itrmf ~= 0)
        itrmjv = 1;
        return;
    end
    output.nfe = output.nfe + 1;
    z = (z - fcur)/eps2;
    itrmjv = 0;
    return;

    %  ----------------------------------------------------------------------
    %  Here ijacv = -1 and ifdord = 2 => second-order central difference.
```

```
    %  ------------------------------------------------------------------------
elseif ijacv == -1 && ifdord == 2
    eps2 = (((1 + norm(xcur))*epsmach)^(1/3))/eps2;
    vtemp = xcur + eps2*v;
    [z, itrmf] = f(vtemp);
    output.nfe = output.nfe + 1;
    if (itrmf ~= 0)
        itrmjv = 1;
        return;
    end
    vtemp = xcur - eps2*v;
    [v, itrmf] = f(vtemp);
    output.nfe = output.nfe + 1;
    if (itrmf ~= 0)
        itrmjv = 1;
        return;
    end

    z = (z - v)/(2*eps2);

    itrmjv = 0;
    return;

    %  ------------------------------------------------------------------------
    %  Here ijacv = -1 and ifdord = 4 => fourth-order difference.
    %  ------------------------------------------------------------------------
elseif ijacv==-1 && ifdord == 4
    eps2 = (((1 + norm(xcur))*epsmach)^(1/5))/eps2;
    [z, itrmf] = f(xcur + eps2*v);
    output.nfe = output.nfe + 1;
    if (itrmf ~= 0)
        itrmjv = 1;
        return;
    end

    xcur = -eps2*v + xcur;
    [vtemp, itrmf] = f(xcur);
    output.nfe = output.nfe + 1;
    if (itrmf ~= 0)
        itrmjv = 1;
        return;
    end
    z = vtemp - z;

    xcur = (eps2/2)*v + xcur;
    [vtemp, itrfm] = f(xcur);
    output.nfe = output.nfe + 1;
    if (itrmf ~= 0)
        itrmjv = 1;
        return;
    end
    z = -8*vtemp + z;
    xcur = eps2*v + xcur;
    [vtemp, itrmf] = f(xcur);
    output.nfe = output.nfe + 1;
    if (itrmf ~= 0)
        itrmjv = 1;
        return;
    end
    z = (8*vtemp + z)*(1/(6*eps2));
    xcur = (-eps2/2)*v + xcur;
    itrmjv = 0;
end

end % end of function nitfd
```

NITBT.M

```
% NITBT - This is the backtracking routine for the (inexact) Newton
%         iterations.
%
% USAGE:
% [instep, eta, xpls, fpls, fpnrm, redfac, ibt, itrmbt] = nitbt(xcur,...
%     f, fcnrm, instep, eta, oftjs, ibtmax, iplvl, thmin, thmax)
%
% INPUT:
% xcur   = current x value
% f      = function name handle of F
% fcnrm  = ||F(xcur)||
% instep = initial (trial) step
% eta    = initial inexact Newton level
% oftjs  = original value of transpose(f)*J*s
% ibtmax = maximum number of backtracking
% iplvl  = printing output level
% thmin  = minimum theta
% thmax  = maximum theta
%
% OUTPUT:
% instep = final acceptable step
% eta    = final inexact Newton level
% xpls   = next approximate solution
% fpls   = F(xpls) value
% fpnrm  = ||F(xpls)|| value
% redfac = scalar factor by which the original step is reduced
% ibt    = number of backtracks during this execution
% itrmbt = termination flag; values have the following meanings:
%              0 => normal termination: acceptable step found.
%              1 => acceptable step not found in ibtmax reductions.
%              2 => error in evaluation of f.
%
% See also nitset, nitsol, nitgm, nitjv, nitfd, nitbt, bicgstab, cgs
%
% Bijaya Zenchenko, May 2006
% Reference: Homer Walker, nitbt.f, NITSOL
%

%% ----------------------------------------------------------------------
%  This is nitbt v0.3, the backtracking routine for the (inexact) Newton
%  iterations.
%  ----------------------------------------------------------------------
function [instep, eta, xpls, fpls, fpnrm, redfac, ibt, itrmbt] = nitbt(xcur,...
    f, fcnrm, instep, eta, oftjs, ibtmax, iplvl, thmin, thmax)

global output

%% ----------------------------------------------------------------------
%  Initialize.
%  ----------------------------------------------------------------------
t = 10^-4;
ibt = 0;
redfac = 1;
xpls=zeros(2,1);

%% ----------------------------------------------------------------------
%  Backtracking loop.
%  ----------------------------------------------------------------------
while 1==1
    xpls = xcur + instep;
    [fpls, itrmf] = f(xpls);

    if (itrmf ~= 0)
        itrmbt = 2;
        go_to_end();
        return;
    end

    output.nfe = output.nfe + 1;
    fpnrm = norm(fpls);

    %  ------------------------------------------------------------------
    %  If t-condition is met or backtracking is turned off, return.
    %  ------------------------------------------------------------------
    if (fpnrm <= (1  - t*(1 - eta))*fcnrm || ibtmax == -1)
```

```
        itrmbt = 0;
        go_to_end();
        return;
    end

    %  ------------------------------------------------------------------------
    %  Otherwise, ...
    %  ------------------------------------------------------------------------
    ibt = ibt + 1;
    if (ibt > ibtmax)
        itrmbt = 1;
        go_to_end();
        return;
    end

    %  ------------------------------------------------------------------------
    %  ... choose theta ...
    %  ------------------------------------------------------------------------
    theta = -(oftjs*redfac)/(fpnrm^2 - fcnrm^2 - 2 *oftjs*redfac);
    if (theta < thmin)
        theta = thmin;
    elseif (theta > thmax)
        theta = thmax;
    end

    %  ------------------------------------------------------------------------
    %  ... then reduce the instep, increase eta, update redfac ...
    %  ------------------------------------------------------------------------
    instep = theta*instep;
    eta = 1  - theta*(1  - eta);
    redfac = theta*redfac;

    %  ------------------------------------------------------------------------
    %  ... and return to the top of the loop.
    %  ------------------------------------------------------------------------
    %  For printing:
    if (iplvl >= 4)
        if (ibt == 1)
            fprintf('\n \n nitbt:  Stepsize reduction no., \t trial F norm, \t current
reduction factor \n');
        end
        fprintf('%.3f \t %.3f \t %.3f \n', ibt, fpnrm, theta);
    end

end % end of while loop

%%  ------------------------------------------------------------------------
%  All returns made here.
%  ------------------------------------------------------------------------
    function go_to_end()

        %  --------------------------------------------------------------------
        %  For printing:
        if (iplvl >= 3 && ibtmax ~= -1)
            if (ibt == 0)
                fprintf('\n nitbt:  no. of instep reductions. = 0\n');
            else
                fprintf('\n nitbt:  no. of instep reductions. = %d \n total reduction
factor = %.3e\n',ibt, redfac);
            end
        end
    end % end of go_to_end()

end % end of function nitbt
```

NITJV.M

```
% NITJV - This is the routine for controlling evaluation of products
%         J*v or J*P(inverse)*v or P(inverse)*v, where J is the
%         Jacobian of f and P is a right preconditioning operator.
%
% USAGE:
% [z, itrmjv] = nitjv(xcur, f, fcur, jacv, ijacv, ifdord, v, itask)
%
% INPUT:
% xcur   = initial x value
% f      = function name handle of F
% fcur   = current F(xcur) value
% jacv   = handle of function with jacobian product J*v
%          and/or preconditioner product inv(P)*v
% ijacv  = flag for determining the method of J*v evaluation:
%              0 => finite-difference evaluation (nitfd) (default)
%              1 => analytic evaluation (jacv)
% ifdord = order of the finite-difference formula (sometimes) used on
%          GMRES restarts when J*v products are evaluated using finite-
%          differences. When ijacv = 0 on input to nitsol, ifdord is set
%          to 1, 2, or 4 in nitsol; otherwise, it is irrelevant.
% v      = vector by which J or J*inv(P) or inv(P) is multiplied by
% itask  = flag for determining which product is produced:
%              0 => z = J*v
%              1 => z = J*inverse(P)*v
%              2 => z = inverse(P)*v
%
% OUTPUT:
% z      = final product based on itask flag
% itrmjv = termination flag; values have the following meanings:
%              0 => normal termination; desired product evaluated.
%              1 => failure to produce J*v.
%              2 => failure to produce inverse(P)*v.
%
% See also nitset, nitsol, nitgm, nitjv, nitfd, nitbt, bicgstab, cgs
%
% Bijaya Zenchenko, May 2006
% Reference: Homer Walker, nitjv.f, NITSOL
%

%% ----------------------------------------------------------------------
%  This is nitjv, the routine for controlling evaluation of products
%  J*v or J*P(inverse)*v or P(inverse)*v, where J is the Jacobian of f
%  and P is a right preconditioning operator.
%  ----------------------------------------------------------------------
function [z, itrmjv] = nitjv(xcur, f, fcur, jacv, ijacv, ifdord, v, itask)

global output

z = zeros(length(xcur),1);

%% ----------------------------------------------------------------------
%  If z = J*v is desired (itask = 0), then copy v into vtemp; if
%  z = J*P(inverse)*v or z = P(inverse)*v is desired (itask = 1,2),
%  then compute P(inverse)*v in vtemp.
%  ----------------------------------------------------------------------
if (itask == 0)
    vtemp = v;
else
    ijob = 1;
    [vtemp, itrmjv] = jacv(xcur, fcur, ijob, v);
    output.nrpre = output.nrpre + 1;
    if (itrmjv ~= 0)
        return;
    end
end

%% ----------------------------------------------------------------------
%  If only z = P(inverse)*v is desired (itask = 2), then copy vtemp into
%  z and exit.
%  ----------------------------------------------------------------------
if (itask == 2)
    z = vtemp;
    return;
end
```

```
%% ------------------------------------------------------------------------
%  If z = J*v or z = J*P(inverse)*v is desired (itask = 0, 1), then
%  compute J*vtemp in z by either analytic evaluation (ijacv = 1) or
%  finite-differences (ijacv = 0, -1).
%  ------------------------------------------------------------------------
if (ijacv == 1)
    ijob = 0;
    [z, itrmjv] = jacv(xcur, fcur, ijob, vtemp);
else
    [z, itrmjv] = nitfd(xcur, f, fcur, ijacv, ifdord, v);
end
output.njve = output.njve + 1;

end % end of function nitjv
```

NITGM.M

```
% NITGM - This function solves Js=-F for s using the General Minimized
%         Risidual Method appraoch in solving linear systems.
%         This implementation is the "simpler" Gram-Schmidt GMRES
%         implementation from L. Zhou and H. F. Walker, "A simpler GMRES,"
%         J. Numerical Lin. Alg. Appl., 1 (1994), pp. 571-581.
%
% USAGE:
% [rsnrm, rsvec, instep, itrmks] = nitgm(xcur, f, fcur, fcnrm, ...
%     eta, jacv, ijacv, irpre, iksmax, iresup, ifdord, kdmax, iplvl)
%
% INPUT:
% xcur   = current x value
% f      = function name handle of F
% fcur   = F(xcur) value
% fcnrm  = ||F(xcur)|| value
% eta    = current eta value
% jacv   = handle of function with jacobian product J*v
%          and/or preconditioner product inv(P)*v
% ijacv  = flag for determining method of J*v evaluation.
%              0 => finite-difference evaluation (nitfd) (default).
%              1 => analytic evaluation (jacv).
% irpre  = flag for right preconditioning:
%              0 => no right preconditioning
%              1 => right preconditioning
% iksmax = maximum allowable number of iterations per call
%          to the Krylov solver routine (default 1000).
% iresup = residual update flag; on restarts, the residual is
%          updated as follows:
%              0 => linear combination (default)
%              1 => direct evaluation
% ifdord = order of the finite-difference formula (sometimes) used on
%          GMRES restarts when J*v products are evaluated using finite-
%          differences. When ijacv = 0 on input to nitsol, ifdord is set
%          to 1, 2, or 4 in nitsol; otherwise, it is irrelevant.
% kdmax  = maximum Krylov subspace dimension (default 20).
% iplvl  = output level
%
% OUTPUT:
% rsnrm  = residual norm
% rsvec  = residual vector
% instep = solution for s where J*s = -F
% itrmks = termination flag
%
% See also nitset, nitsol, nitgm, nitjv, nitfd, nitbt, bicgstab, cgs
%
% Bijaya Zenchenko, May 2006
% Reference: Homer Walker, nitgm.f, NITSOL
%

%% ----------------------------------------------------------------------
%  This is nitgm, the GMRES routine for determining (trial) inexact
%  Newton steps. This implementation is the "simpler" Gram-Schmidt GMRES
%  implementation from L. Zhou and H. F. Walker, "A simpler GMRES,"
%  J. Numerical Lin. Alg. Appl., 1 (1994), pp. 571-581.
%  ----------------------------------------------------------------------
function [rsnrm, rsvec, instep, itrmks] = nitgm(xcur, f, fcur, fcnrm, ...
    eta, jacv, ijacv, irpre, iksmax, iresup, ifdord, kdmax, iplvl)

global output

vv = zeros(length(xcur), kdmax+1);
rr = zeros(kdmax, kdmax);
w = zeros(kdmax+1,1);

%% ----------------------------------------------------------------------
%  Initialize.
%  ----------------------------------------------------------------------
epsmach = 2*eps;
cndmax = 1/(100*epsmach);
instep = zeros(length(xcur),1);
igm = 0;

%% ----------------------------------------------------------------------
%  For printing:
if (iplvl >= 3)
```

```
    fprintf('\n nitgm:\t eta = %.8f \n', eta);
end
if (iplvl >= 4)
    fprintf(' nitgm:\t GMRES iteration no. \t linear residual norm \t condition no.
estimate \n');
    fprintf(' \t\t\t %.8f \t\t\t %.8f \n', igm, fcnrm);
end

%% ------------------------------------------------------------------------
%  Set the stopping tolerance, etc.
%  ------------------------------------------------------------------------
rsnrm0 = fcnrm;
abstol = eta*rsnrm0;

%% ------------------------------------------------------------------------
%  Place the normalized initial residual in the first column of the vv array.
%  ------------------------------------------------------------------------
vv(:,1) = -fcur/fcnrm;

%% ------------------------------------------------------------------------
%  Top of the outer GMRES loop.
%  ------------------------------------------------------------------------
counter = 0;
while 1==1 % infinite loop but different triggers in code exit out of while loop
    kd = 0;
    rsnrm = 1;

    %  --------------------------------------------------------------------
    %  Top of the inner GMRES loop.
    %  --------------------------------------------------------------------
    while 1==1 % infinite loop but different triggers in code exit out of while loop

        counter=counter+1; % for plotting
        X(counter) = counter;
        Y(counter) = log(rsnrm*rsnrm0);

        kd = kd + 1;
        kdp1 = kd + 1;
        output.nli = output.nli + 1;
        igm = igm + 1;

        %  ----------------------------------------------------------------
        %  Evaluate J*(kd-th Krylov subspace basis vector) in vv(.,kdp1).
        %  ----------------------------------------------------------------
        if (irpre == 0)
            itask = 0;
        else
            itask = 1;
        end

        [vv(:,kdp1), itrmjv] = nitjv(xcur, f, fcur, jacv, ijacv, ...
            ifdord, vv(:,kd), itask);

        if itrmjv ~= 0
            itrmks = itrmjv;
            go_to_end();
            return;
        end

        %  ----------------------------------------------------------------
        %  Do modified Gram-Schmidt.
        %  ----------------------------------------------------------------
        for i=2:kd
            rr(i-1,kd) = vv(:,i)'*vv(:,kdp1);
            vv(:,kdp1) = -rr(i-1,kd)*vv(:,i) + vv(:,kdp1);
        end
        rr(kd,kd) = norm(vv(:,kdp1));

        %  ----------------------------------------------------------------
        %  Terminate if the estimated condition number is too great.
        %  ----------------------------------------------------------------
        est_cond = cond(rr(1:kd,1:kd));
        if est_cond >= cndmax
            if (kd == 1)
                itrmks = 5;
                go_to_end();
                return;
            else
                kdp1 = kd;
                kd = kd - 1;
                vv(:,1) = w(kd)*vv(:,kdp1)+vv(:,1);
```

```
                break;
            end
        end

        %  ------------------------------------------------------------------------
        %  Normalize vv(.,kdp1).
        %  ------------------------------------------------------------------------
        vv(:,kdp1) = vv(:,kdp1)/rr(kd,kd) ;

        %  ------------------------------------------------------------------------
        %  Update w and the residual norm by rsnrm <- rsnrm*dsin(dacos(w(kd)/rsnrm).
        %  ------------------------------------------------------------------------
        w(kd) = vv(:,1)'*vv(:,kdp1) ;
        temp = max(min(w(kd)/rsnrm,1),-1);
        rsnrm = rsnrm*sin(acos(temp));

        %  ------------------------------------------------------------------------
        %  For printing:
        if (iplvl >= 4)
            fprintf(' \t\t\t %.8f \t\t\t %.8f \t\t\t %.8f \n', igm,  rsnrm*rsnrm0,
est_cond);
        end

        %  ------------------------------------------------------------------------
        %  Test for termination of the inner loop.
        %  If not terminating the inner loop, update the residual vector
        %  and go to the top of the inner loop.
        %  ------------------------------------------------------------------------

        if ( (rsnrm0*rsnrm <= abstol) || (kd == kdmax)  ||  (igm >= iksmax) )
            break;
        end

        vv(:,1) = -w(kd)*vv(:,kdp1) + vv(:,1);

    end %  Bottom of inner loop. -----------------------------------------

    %  ------------------------------------------------------------------------
    %  For printing:
    if (iplvl >= 4)
        fprintf('------- \n');
    end

    %  ------------------------------------------------------------------------
    %  Compute the solution:
    %  Use svbig for storage of the original components of w.
    %  ------------------------------------------------------------------------
    svbig = w(1:kd);

    %  ------------------------------------------------------------------------
    %  Overwrite w with the solution of the upper triangular system.
    %  ------------------------------------------------------------------------
    for i = kd:-1:1
        w(i) = w(i)/rr(i,i);
        if (i > 1)
            w(1:i-1) = -w(i)*rr(1:i-1,i) + w(1:i-1);
        end
    end

    %  ------------------------------------------------------------------------
    %  Now form the linear combination to accumulate the correction in
    %  the work vector.
    %  ------------------------------------------------------------------------
    vtemp = vv(:,1)*w(1);

    if (kd > 1)
        w(2:kd) = w(1)*svbig(1:kd-1) + w(2:kd);
        for i = 2:kd
            vtemp = w(i)*vv(:,i) + vtemp;
        end
    end

    %  ------------------------------------------------------------------------
    %  If iresup == 0, then update the residual vector by linear
    %  combination. This frees vv(.,kdp1) for use as a work array.
    %  ------------------------------------------------------------------------
    if (iresup == 0)
        vv(:,1) = -svbig(kd)*vv(:,kdp1) + vv(:,1);
    end

    %  ------------------------------------------------------------------------
```

```
%  If right preconditioning is used, overwrite
%  correction <-- P(inverse)*correction, using vv(.,kdp1) as a work array.
%  Note: vv(.,kdp1) can be used for both work arrays in this call because
%  the second is not referenced within nitjv.
%  ------------------------------------------------------------------------
if (irpre > 0) && (ijacv == 1)
    itask = 2;
    [vtemp, itrmjv] = nitjv(xcur, f, fcur, jacv, ijacv, ifdord, vtemp, itask);

    if (itrmjv > 0)
        itrmks = 2;
        go_to_end();
        return;
    end
end

%  ------------------------------------------------------------------------
%  Update the instep. This frees vtemp for use as a work array.
%  ------------------------------------------------------------------------
instep = rsnrm0*vtemp + instep;

%  ------------------------------------------------------------------------
%  If iresup == 1, then update the residual vector by direct evaluation,
%  using vtemp and vv(.,kdp1) as work arrays. Note: Two distinct work
%  arrays are needed in this call because both are referenced within nitjv
%  if the J*instep product is evaluated with a finite-difference of order
%  two or higher. If finite-differences are used (ijacv= 0), then ijacv
%  is temporarily set to -1 to signal to nitjv that the order of the
%  finite-difference formula is to be determined by ifdord.
%  ------------------------------------------------------------------------
if (iresup == 1)
    itask = 0;
    if (ijacv == 0)
        ijacv = -1;
    end

    [vv(:,1),  itrmjv] = nitjv(xcur, f, fcur, jacv, ijacv, ifdord, instep, itask);

    if (ijacv == -1)
        ijacv = 0;
    end

    if (itrmjv > 0)
        itrmks = 1;
        go_to_end();
        return;
    end
    vv(:,1) = -fcur(:) - vv(:,1);
end

%  ------------------------------------------------------------------------
%  Test for termination.
%  ------------------------------------------------------------------------
if (rsnrm0*rsnrm <= abstol)
    itrmks = 0;
    go_to_end();
    return;
end
if (igm >= iksmax)
    itrmks = 3;
    go_to_end();
    return;
end
temp = kd*log(abstol/(rsnrm0*rsnrm))/log(rsnrm/(1 + 10*epsmach));
if (temp >= 1000*(iksmax - igm))
    itrmks = 4;
    go_to_end();
    return;
end

%  ------------------------------------------------------------------------
%  If not terminating, then normalize the initial residual, etc., and
%  return to the top of the outer loop.
%  ------------------------------------------------------------------------
if (iresup == 0)
    rsnrm0 = rsnrm0*rsnrm;
    temp = 1/rsnrm;
else
    rsnrm0 = norm(vv(:,1));
    temp = 1/rsnrm0;
end
```

```
    vv(:,1) = temp*vv(:,1);

end % Bottom of outer while loop. ----------------------------------------

%% ------------------------------------------------------------------------
%  All returns made here.
%  ------------------------------------------------------------------------

go_to_end();

    function go_to_end()

        if (itrmks ~= 1 && itrmks ~= 2)
            if (iresup == 0)
                vv(:,1) = rsnrm0*vv(:,1);
                rsnrm = rsnrm0*rsnrm;
            else
                rsnrm = norm(vv(:,1));
            end
        end

        %  ------------------------------------------------------------------
        %  For printing:
        if (iplvl >= 3)
            if (itrmks ~= 1 && itrmks ~= 2)
                fprintf(' nitgm:  itrmks = %.3f \t   final lin. res. norm = %.3f \n',
...
                    itrmks,     rsnrm);
            else
                fprintf(' nitgm: itrmks: %.3f \n', itrmks);
            end
        end
        rsvec = vv(:,1);
        if ~isempty(X)
            hold on
            X = [output.nni:1/(length(X)):output.nni + 1 - 1/(length(X))];
            plot(X,Y, '.');
        end
    end % end of go_to_end()

end % end of function nitgm
```

References

[1] J. E. DENNIS, JR. AND R. B. SCHNABEL, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Series in Automatic Computation, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[2] S. C. EISENSTAT, H. F. WALKER, *Globally Convergent Inexact Newton Methods,* SIAM J. Optimization, 4 (1994), pp. 393-422.

[3] M. PERNICE, H. F. WALKER, *NITSOL: A Newton Iterative Solver for Nonlinear Systems,* SIAM J. Sci. Comput., 19 (1998), pp. 302-318.

[4] Y. SAAD, M. H. SCHULTZ, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems,* SIAM J. Sci. Stat. Comput., 7 (1986), pp. 856-869.

[5] P. SONNEVELD, *CGS, a fast Lanczos-type solver for nonsymmetric linear systems,* SIAM J. Sci. Stat. Comput., 10 (1989), pp. 36-52.

[6] H. A. VAN DER VORST, *Bi-CGSTAB: a fast and smoothly convergent variant of bi-cg for the solution of nonsymmetric linear systems,* SIAM J. Sci. Stat. Comput., 13 (1992), pp. 631-644.

[7] H. A. VAN DER VORST, *Iterative Krylov Methods for Large Linear Systems,* Cambridge University Press, Cambridge UK, 2003.

[8] H. F. WALKER, *Course Notes: Numerical Methods for Nonlinear Equations,* Worcester Polytechnic Institute Mathematical Sciences Department, Technical Report, March 2002.

[9] H. F. WALKER, L. ZHOU, *A Simpler GMRES,* Numerical Linear Algebra with Applications, 1(6) (1994), pp. 571-581.