

# **SANDIA REPORT**

SAND2010-1422  
Unlimited Release  
Printed March 2010

## **Poblano v1.0: A Matlab Toolbox for Gradient-Based Optimization**

Daniel M. Dunlavy, Tamara G. Kolda, and Evrim Acar

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,  
a Lockheed Martin Company, for the United States Department of Energy's  
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** 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, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent 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, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.osti.gov/bridge>

Available to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



# Poblano v1.0: A Matlab Toolbox for Gradient-Based Optimization

[Daniel M. Dunlavy](#)

Computer Science & Informatics Department  
Sandia National Laboratories, Albuquerque, NM 87123-1318  
Email: [dmdunla@sandia.gov](mailto:dmdunla@sandia.gov)

[Tamara G. Kolda](#) and [Evrin Acar](#)

Information and Decision Sciences Department  
Sandia National Laboratories  
Livermore, CA 94551-9159  
Email: [tgkolda@sandia.gov](mailto:tgkolda@sandia.gov), [evrim.acarataman@gmail.com](mailto:evrim.acarataman@gmail.com)

## Abstract

We present Poblano v1.0, a Matlab toolbox for solving gradient-based unconstrained optimization problems. Poblano implements three optimization methods (nonlinear conjugate gradients, limited-memory BFGS, and truncated Newton) that require only first order derivative information. In this paper, we describe the Poblano methods, provide numerous examples on how to use Poblano, and present results of Poblano used in solving problems from a standard test collection of unconstrained optimization problems.

## Acknowledgments

We thank Dianne O'Leary for providing the Matlab translation of the MINPACK implementation of the Moré-Thuente line search. This product includes software developed by the University of Chicago, as Operator of Argonne National Laboratory. This work was fully supported by Sandia's Laboratory Directed Research & Development (LDRD) program.

# Contents

1	Toolbox Overview	7
1.1	Introduction	7
1.2	Optimization Methods	7
1.3	Globalization Strategies	8
1.4	Optimization Input Parameters	8
1.5	Optimization Output Parameters	12
1.6	Examples of Running Poblano Optimizers	15
2	Optimization Methods	19
2.1	Nonlinear Conjugate Gradient	19
2.2	Limited-memory BFGS	23
2.3	Truncated Newton	27
3	Checking Gradient Calculations	31
3.1	Difference Formulas	31
3.2	Gradient Check Input Parameters	31
3.3	Gradient Check Output Parameters	32
3.4	Examples	32
4	Numerical Experiments	35
4.1	Description of Test Problems	35
4.2	Results	35
5	Conclusions	43
	References	44

## Tables

1	Optimization method input parameters for controlling amount of information displayed. ....	8
2	Optimization method input parameters for controlling iteration termination. ....	9
3	Optimization method input parameters for controlling what information is saved for each iteration. ....	9
4	Optimization method input parameters for controlling line search methods. ....	9
5	Output parameters returned by all Poblano optimization methods. ....	12
6	Additional trace output parameters returned by all Poblano optimization methods if requested. ....	12
7	Conjugate direction updates available in Poblano. ....	19
8	Method specific parameters for Poblano's <code>ncg</code> optimization method. ....	20
9	Method specific parameters for Poblano's <code>lbfgs</code> optimization method. ....	24
10	Method specific parameters for Poblano's <code>tn</code> optimization method. ....	28
11	Difference formulas available in Poblano for checking user-defined gradients. ....	31
12	Input parameters for Poblano's <code>gradientcheck</code> function. ....	31
13	Output parameters generated by Poblano's <code>gradientcheck</code> function. ....	32
14	Results of <code>ncg</code> using PR updates on the Moré, Garbow, Hillstrom test collection. Errors greater than $10^{-8}$ are highlighted in bold, indicating that a solution was not found within the specified tolerance. ....	37
15	Parameter changes that lead to solutions using <code>ncg</code> with PR updates on the Moré, Garbow, Hillstrom test collection. ....	37
16	Results of <code>ncg</code> using HS updates on the Moré, Garbow, Hillstrom test collection. Errors greater than $10^{-8}$ are highlighted in bold, indicating that a solution was not found within the specified tolerance. ....	38
17	Parameter changes that lead to solutions using <code>ncg</code> with HS updates on the Moré, Garbow, Hillstrom test collection. ....	38
18	Results of <code>ncg</code> using FR updates on the Moré, Garbow, Hillstrom test collection. Errors greater than $10^{-8}$ are highlighted in bold, indicating that a solution was not found within the specified tolerance. ....	39
19	Parameter changes that lead to solutions using <code>ncg</code> with FR updates on the Moré, Garbow, Hillstrom test collection. ....	39
20	Results of <code>lbfgs</code> on the Moré, Garbow, Hillstrom test collection. Errors greater than $10^{-8}$ are highlighted in bold, indicating that a solution was not found within the specified tolerance. ....	40
21	Parameter changes that lead to solutions using <code>lbfgs</code> on the Moré, Garbow, Hillstrom test collection. ....	40
22	Results of <code>tn</code> on the Moré, Garbow, Hillstrom test collection. Errors greater than $10^{-8}$ are highlighted in bold, indicating that a solution was not found within the specified tolerance. ...	41
23	Parameter changes that lead to solutions using <code>tn</code> on the Moré, Garbow, Hillstrom test collection. ....	41

# 1 Toolbox Overview

Poblano is a toolbox of large-scale algorithms for nonlinear optimization. The algorithms in Poblano require only first-order derivative information (e.g., gradients for scalar-valued objective functions).

## 1.1 Introduction

Poblano optimizers find local minimizers of scalar-valued objective functions taking vector inputs. Specifically, the problems solved by Poblano optimizers are of the following form:

$$\min_x f(x), \quad \text{where } f: \mathbb{R}^N \rightarrow \mathbb{R}$$

The gradient of the objective function,  $\nabla f(x)$ , is required for all Poblano optimizers. The optimizers converge to a stationary point,  $x^*$ , where

$$\nabla f(x^*) \approx 0$$

A line search satisfying the strong Wolfe conditions is used to guarantee global convergence of the Poblano optimizers.

## 1.2 Optimization Methods

The following methods for solving the problem in (1.1) are available in Poblano. See §2 for detailed descriptions of the methods.

- Nonlinear conjugate gradient method (**ncg**) [9]
  - Uses Fletcher-Reeves, Polak-Ribiere, and Hestenes-Stiefel conjugate direction updates
  - Restart strategies based on number of iterations or orthogonality of gradients across iterations
  - Steepest descent method is a special case
- Limited-memory BFGS method (**lbfgs**) [9]
  - Uses a two-loop recursion for approximate Hessian-gradient products
- Truncated Newton method (**tn**) [1]
  - Uses finite differencing for approximate Hessian-vector products

### 1.2.1 Calling a Poblano Optimizer

All Poblano methods are called using the name of the method along with two required arguments and one or more optional arguments. The required arguments are 1) a handle to the function being minimized, and 2) the initial guess of the solution (as a scalar or column vector). For example, the following is a call to the **ncg** method to minimize the **example1** function distributed with Poblano starting with an initial guess of  $x = \pi/4$  and using the default **ncg** parameters.

```
>> ncg(@example1, pi/4);
```

Parameterize functions can be optimized using Poblano as well. For such functions, the function handle can be used to specify the function parameters. For example, Poblano's **example1** function takes an optional scalar parameter as follows:

```
>> ncg(@(x) example1(x,3), pi/4);
```

Functions taking vectors as inputs can be optimized using Poblano as well. For functions which can take input vectors of arbitrary sizes (e.g., Matlab functions such as `sin`, `fft`, etc.), the size of the initial guess (as a scalar or column vector) determines the size of the problem to be solved. For example, Poblano's `example1` function can take as input a vector (in this case a vector in  $\mathbb{R}^3$ ) as follows:

```
>> ncg(@(x) example1(x,3), [pi/5 pi/4 pi/3]');
```

The optional arguments are input parameters specifying how the optimization method is to be run (see §1.4 for details about the input parameters.)

### 1.3 Globalization Strategies

- Line search methods
  - Moré-Thuente cubic interpolation line search (`cvsrch`) [8]

### 1.4 Optimization Input Parameters

Input parameters are passed to the different optimization methods using Matlab `inputParser` objects. Some parameters are shared across all methods and others are specific to a particular method. Below are descriptions of the shared input parameters and examples of how to set and use these parameters in the minimization methods. The Poblano function `poblano_params` is used by the optimization methods to set the input parameters.

**Display Parameters.** The following parameters control the information that is displayed during a run of a Poblano optimizer.

Parameter	Description	Type	Default
Display	Controls amount of printed output 'iter': Display information every iteration 'final': Display information only after final iteration 'off': Do not display any information	char	'iter'

Table 1: Optimization method input parameters for controlling amount of information displayed.

The per iteration information displayed contains the iteration number (`Iter`), total number of function evaluations performed thus far (`FuncEvals`), the function value at the current iterate (`F(X)`), and the norm of the gradient at the current iterate scaled by the problem size (`||G(X)||/N`). After the final iteration is performed, the total iteration and function evaluations, along with the function value and scaled gradient norm at the solution found is displayed. Below is an example of the information displayed using the default parameters.

Iter	FuncEvals	F(X)	G(X)  /N
0	1	0.70710678	2.12132034
1	14	-0.99998885	0.01416497
2	16	-1.00000000	0.00000147



**Stopping Criteria Parameters.** The following parameters control the stopping criteria of the optimization methods.

Parameter	Description	Type	Default
MaxIters	Maximum number of iterations allowed	int	100
MaxFuncEvals	Maximum number of function evaluations allowed	int	100
StopTol	Gradient norm stopping tolerance, i.e., the method stops when the norm of the gradient is less than <code>StopTol</code> times the number of variables	double	$10^{-5}$
RelFuncTol	Relative function value change stopping tolerance, i.e., the method stops when the relative change of the function value from one iteration to the next is less than <code>RelFuncTol</code>	double	$10^{-6}$

Table 2: Optimization method input parameters for controlling iteration termination.

**Trace Parameters.** The following parameters control the information that is saved and output for each iteration.

Parameter	Description	Type	Default
TraceX	Flag to save a history of the current best point at each iteration	boolean	false
TraceFunc	Flag to save a history of the function values of the current best point at each iteration	boolean	false
TraceRelFunc	Flag to save a history of the relative difference between the function values of the best point at the current and previous iterations ( $\ F(X_k) - F(X_{k-1})\ /\ F(X_{k-1})\ $ ) at each iteration	boolean	false
TraceGrad	Flag to save a history of the gradients of the current best point at each iteration	boolean	false
TraceGradNorm	Flag to save a history of the norm of the gradients of the current best point at each iteration	boolean	false
TraceFuncEvals	Flag to save a history of the number of function evaluations performed at each iteration	boolean	false

Table 3: Optimization method input parameters for controlling what information is saved for each iteration.

**Line Search Parameters.** The following parameters control the behavior of the line search method used in the optimization methods.

Parameter	Description	Type	Default
LineSearch_xtol	Stopping tolerance for minimum change in input variable	double	$10^{-15}$
LineSearch_ftol	Stopping tolerance for sufficient decrease condition	double	$10^{-4}$
LineSearch_gtol	Stopping tolerance for directional derivative condition	double	$10^{-2}$
LineSearch_stpmin	Minimum step size allowed	double	$10^{-15}$
LineSearch_stpmax	Maximum step size allowed	double	$10^{15}$
LineSearch_maxfev	Maximum number of iterations allowed	int	20
LineSearch_initialstep	Initial step to be taken in the line search	double	1

Table 4: Optimization method input parameters for controlling line search methods.

### 1.4.1 Default Parameters

The default input parameters are returned using the sole input of 'defaults' to one of the Poblano optimization methods:

```
>> ncg_default_params = ncg('defaults');  
>> lbfgs_default_params = lbfgs('defaults');  
>> tn_default_params = tn('defaults');
```

### 1.4.2 Method Specific Parameters

The parameters specific to each particular optimization method are described later in the following tables:

- Nonlinear conjugate gradient (**ncg**): [Table 8](#)
- Limited-memory BFGS (**lbfgs**): [Table 9](#)
- Truncated Newton (**tn**): [Table 10](#)

### 1.4.3 Passing Parameters to Methods

As mentioned above, input parameters are passed to the Poblano optimization methods using Matlab `inputParser` objects. Below are several examples of passing parameters to Poblano methods. For more detailed description of `inputParser` objects, see the Matlab documentation.

**Case 1: Using default input parameters.** To use the default methods, simply pass the function handle to the function/gradient method and a starting point to the optimization method (i.e., do not pass any input parameters into the method).

```
>> ncg(@(x) example1(x,3), pi/4);
```

Iter	FuncEvals	F(X)	G(X)  /N
0	1	0.70710678	2.12132034
1	14	-0.99998885	0.01416497
2	16	-1.00000000	0.00000147

**Case 2: Passing parameter-value pairs into a method.** Instead of passing a structure of input parameters, pairs of parameters and values may be passed as well. In this case, all parameters not specified as input use their default values. Below is an example of specifying a parameter in this way.

```
>> ncg(@(x) example1(x,3), pi/4, 'Display', 'final');
```

Iter	FuncEvals	F(X)	G(X)  /N
2	16	-1.00000000	0.00000147

**Case 3: Passing input parameters as fields in a structure.** Input parameters can be passed as fields in a structure. Note, though, that since each optimization method uses method specific parameters, it is suggested to start from a structure of the default parameters for a particular method. Once the structure of default parameters has been created, the individual parameters (i.e., fields in the structure) can be changed. Below is an example of this for the `ncg` method.

```
>> params = ncg('defaults');
>> params.MaxIters = 1;
>> ncg(@(x) example1(x,3), pi/4, params);
```

Iter	FuncEvals	F(X)	G(X)  /N
0	1	0.70710678	2.12132034
1	14	-0.99998885	0.01416497

**Case 4: Using parameters from one run in another run.** One of the outputs returned by the Poblano optimization methods is the `inputParser` object of the input parameters used in that run. That object contains a field called `Results`, which can be passed as the input parameters to another run. For example, this is helpful when running comparisons of methods where only one parameter is changed. Shown below is such an example, where default parameters are used in one run, and the same parameters with just a single change are used in another run.

```
>> out = ncg(@(x) example1(x,3), pi./[4 5 6]');
>> params = out.Params.Results;
>> params.Display = 'final';
>> ncg(@(x) example1(x,3), pi./[4 5 6]', params);
```

Iter	FuncEvals	F(X)	G(X)  /N
0	1	2.65816330	0.77168096
1	7	-0.63998759	0.78869570
2	11	-0.79991790	0.60693819
3	14	-0.99926100	0.03843827
4	16	-0.99999997	0.00023739
5	18	-1.00000000	0.00000000

Iter	FuncEvals	F(X)	G(X)  /N
5	18	-1.00000000	0.00000000

## 1.5 Optimization Output Parameters

Each of the optimization methods in Poblano outputs a single structure containing fields described in [Table 5](#).

Parameter	Description	Type
<b>X</b>	Final iterate	vector, double
<b>F</b>	Function value at <b>X</b>	double
<b>G</b>	Gradient at <b>X</b>	vector, double
<b>Params</b>	Input parameters used for the minimization method (as parsed Matlab <code>inputParser</code> object)	struct
<b>FuncEvals</b>	Number of function evaluations performed	int
<b>Iters</b>	Number of iterations performed (see individual optimization routines for details on what each iteration consists of)	int
<b>ExitFlag</b>	Termination flag, with one of the following values 0 : scaled gradient norm < <code>StopTol</code> input parameter) 1 : maximum number of iterations exceeded 2 : maximum number of function values exceeded 3 : relative change in function value < <code>RelFuncTol</code> input parameter 4 : NaNs found in <b>F</b> , <b>G</b> , or <code>  G  </code>	int

Table 5: Output parameters returned by all Poblano optimization methods.

### 1.5.1 Optional Trace Output Parameters

Additional output parameters returned by the Poblano optimization methods are presented in [Table 6](#). The histories (i.e., traces) of different variables and parameters at each iteration are returned as output parameters if the corresponding input parameters are set to `true` (see [§1.4](#) for more details on the input parameters).

Parameter	Description	Type
<b>TraceX</b>	History of <b>X</b> (iterates)	matrix, double
<b>TraceFunc</b>	History of the function values of the iterates	vector, double
<b>TraceRelFunc</b>	History of the relative difference between the function values at the current and previous iterates	vector, double
<b>TraceGrad</b>	History of the gradients of the iterates	matrix, double
<b>TraceGradNorm</b>	History of the norm of the gradients of the iterates	vector, double
<b>TraceFuncEvals</b>	History of the number of function evaluations performed at each iteration	vector, int

Table 6: Additional trace output parameters returned by all Poblano optimization methods if requested.

## 1.5.2 Example Output of Optimization Methods

The following example shows the output produced when the default input parameters are used.

```
>> out = ncg(@(x) example1(x,3), pi/4)
```

Iter	FuncEvals	F(X)	G(X)  /N
0	1	0.70710678	2.12132034
1	14	-0.99998885	0.01416497
2	16	-1.00000000	0.00000147

```
out =  
  Params: [1x1 inputParser]  
  ExitFlag: 0  
  X: 70.6858  
  F: -1.0000  
  G: -1.4734e-06  
  FuncEvals: 16  
  Iters: 2
```

The following example presents an example where a method terminates before convergence (due to a limit on the number of iterations allowed).

```
>> out = ncg(@(x) example1(x,3), pi/4, 'Display', 'final', 'MaxIters', 1)
```

Iter	FuncEvals	F(X)	G(X)  /N
1	14	-0.99998885	0.01416497

```
out =  
  Params: [1x1 inputParser]  
  ExitFlag: 1  
  X: 70.6843  
  F: -1.0000  
  G: -0.0142  
  FuncEvals: 14  
  Iters: 1
```

The following shows the ability to save traces of the different information for each iteration.

```
>> out = ncg(@(x) example1(x,3), rand(3,1), 'TraceX', true, 'TraceFunc', true, ...  
  'TraceRelFunc', true, 'TraceGrad', true, 'TraceGradNorm', true, 'TraceFuncEvals', true)
```

Iter	FuncEvals	F(X)	G(X)  /N
0	1	0.92763793	1.63736413
1	5	-2.71395809	0.72556878

```

2         9        -2.95705211      0.29155981
3         12       -2.99999995      0.00032848
4         14       -3.00000000      0.00000000

```

```
out =
```

```

      Params: [1x1 inputParser]
      ExitFlag: 0
           X: [3x1 double]
           F: -3
           G: [3x1 double]
      FuncEvals: 14
           ITERS: 4
      TraceX: [3x5 double]
      TraceFunc: [0.9276 -2.7140 -2.9571 -3.0000 -3]
      TraceRelFunc: [3.9257 0.0896 0.0145 1.7984e-08]
      TraceGrad: [3x5 double]
      TraceGradNorm: [4.9121 2.1767 0.8747 9.8545e-04 2.3795e-10]
      TraceFuncEvals: [1 4 4 3 2]

```

We can examine the final solution and its gradient (which list only their sizes when viewing the output structure):

```
>> X = out.X
>> G = out.G
```

```

X =
    3.6652
   -2.6180
    3.6652

G =
    1.0e-09 *
    0.0512
    0.1839
    0.1421

```

We can also see the values of X (current iterate) and its gradient G for each iteration (including iteration 0, which just computes the function and gradient values of the initial point):

```
>> TraceX = out.TraceX
>> TraceGrad = out.TraceGrad
```

```

TraceX =
    0.9649    3.7510    3.7323    3.6652    3.6652
    0.1576   -2.4004   -2.6317   -2.6179   -2.6180
    0.9706    3.7683    3.7351    3.6653    3.6652

TraceGrad =
   -2.9090    0.7635    0.5998    0.0002    0.0000
    2.6708    1.8224   -0.1231    0.0008    0.0000
   -2.9211    0.9131    0.6246    0.0006    0.0000

```

## 1.6 Examples of Running Poblano Optimizers

To solve a problem using Poblano, the following steps are performed:

- **Step 1: Create M-file for objective and gradient.** An M-file which takes a vector as input and provides a scalar function value and gradient vector (the same size as the input) must be provided to the Poblano optimizers.
- **Step 2: Call the optimizer.** One of the Poblano optimizers is called, taking an anonymous function handle to the function to be minimized, a starting point, and an optional set of optimizer parameters as inputs.

The following examples of functions are provided with Poblano.

### 1.6.1 Example 1: Multivariate Sum

The following example is a simple multivariate function,  $f_1 : \mathbb{R}^N \rightarrow \mathbb{R}$ , that can be minimized using Poblano:

$$f_1(x) = \sum_{n=1}^N \sin(ax_i)$$
$$(\nabla f_1(x))_i = a \cos(ax_i), \quad i = 1, \dots, N$$

where  $a$  is a scalar parameter.

Listed below are the contents of the `example1.m` M-file distributed with the Poblano code. This is an example of a self-contained function requiring a vector of independent variables and an *optional* scalar input parameter for  $a$ .

```
function [f,g]=example1(x,a)
    if nargin < 2, a = 1; end
    f = sum(sin(a*x));
    g = a*cos(a*x);
```

The following presents a call to the `ncg` optimizer using the default parameters (see §2.1 for more details) along with the output of Poblano. By default, at each iteration Poblano displays the number of function evaluations (`FuncEvals`), the function value at the current iterate (`F(X)`), and the Euclidean norm of the scaled gradient at the current iterate (`||G(X)||/N`, where `N` is the size of `X`) for each iteration. The output of Poblano optimizers is a Matlab structure containing the solution. The structure of the output is described in detail in §1.5.

```
>> out = ncg(@(x) example1(x,3), pi/4)
```

Iter	FuncEvals	F(X)	G(X)  /N
0	1	0.70710678	2.12132034
1	14	-0.99998885	0.01416497
2	16	-1.00000000	0.00000147

```

out =
  Params: [1x1 inputParser]
  ExitFlag: 0
         X: 70.6858
         F: -1.0000
         G: -1.4734e-06
  FuncEvals: 16
  ITERS: 2

```

## 1.6.2 Example 2: Matrix Decomposition

The following example is a more complicated function involving matrix variables. As the Poblano methods require scalar functions with vector inputs, variable matrices must be reshaped into vectors first. The problem in this example is to find a rank- $k$  decomposition,  $UV^T$ , of a  $m \times n$  matrix,  $A$ , which minimizes the Frobenius norm of the fit of the decomposition to the matrix. More formally,

$$\begin{aligned}
 f_2 &= \frac{1}{2} \|A - UV^T\|_F^2 \\
 \nabla_U f_2 &= -(A - UV^T)V \\
 \nabla_V f_2 &= -(A - UV^T)^T U
 \end{aligned}$$

where  $A \in \mathbb{R}^{m \times n}$  is a matrix with  $\text{rank} \geq k$ ,  $U \in \mathbb{R}^{m \times k}$ , and  $V \in \mathbb{R}^{n \times k}$ . This problem can be solved using Poblano by providing an M-file that computes the function and gradient shown above but that takes  $U$  and  $V$  as input in vectorized form.

Listed below are the contents of the `example2.m` M-file distributed with the Poblano code. Note that the input `Data` is required and is a structure containing the matrix to be decomposed,  $A$ , and the desired rank,  $k$ . This example also illustrates how the vectorized form of the factor matrices,  $U$  and  $V$ , are converted to matrix form for the function and gradient computations.

```

function [f,g]=example2(x,Data)
% Data setup
[m,n] = size(Data.A);
k = Data.rank;
U = reshape(x(1:m*k),m,k);
V = reshape(x(m*k+1:m*k+n*k),n,k);

% Function value (residual)
AmUVt = Data.A-U*V';
f = 0.5*norm(AmUVt,'fro')^2;

% First derivatives computed in matrix form
g = zeros((m+n)*k,1);
g(1:m*k) = -reshape(AmUVt*V,m*k,1);
g(m*k+1:end) = -reshape(AmUVt'*U,n*k,1);

```

Included with Poblano are two helper functions and which can be used to generate problems instances along with starting points (`example2_init.m`) and extract the factors  $U$  and  $V$  from a solution vector (`example2_extract.m`). We show an example of their use below.



```

>> randn('state',0);
>> m = 4; n = 3; k = 2;
>> [x0,Data] = example2_init(m,n,k)
>> out = ncg(@(x) example2(x,Data), x0, 'RelFuncTol', 1e-16, 'StopTol', 1e-8, ...
    'MaxFuncEvals',1000,'Display','final')

```

```

x0 =
    -0.588316543014189
         2.1831858181971
    -0.136395883086596
         0.11393131352081
         1.06676821135919
    0.0592814605236053
    -0.095648405483669
    -0.832349463650022
         0.29441081639264
        -1.3361818579378
         0.714324551818952
         1.62356206444627
    -0.691775701702287
         0.857996672828263

```

```

Data =
    rank: 2
    A: [4x3 double]

```

Iter	FuncEvals	F(X)	G(X)  /N
29	67	0.28420491	0.00000001

```

out =
    Params: [1x1 inputParser]
    ExitFlag: 0
         X: [14x1 double]
         F: 0.284204907556308
         G: [14x1 double]
    FuncEvals: 67
         ITERS: 29

```

Extracting the factors from the solution, we see that we have found a solution, since the the Euclidean norm of the difference between the matrix and the approximate solution is equal to the the  $k + 1$  singular value of  $A$  [4, Theorem 2.5.3].

```
>> [U,V] = example2_extract(4,3,2,out.X);  
>> norm_diff = norm(Data.A-U*V')  
>> sv = svd(Data.A);  
>> sv_k_plus_1 = sv(k+1)
```

```
norm_diff =  
          0.753929582330216
```

```
sv_k_plus_1 =  
          0.753929582330215
```

## 2 Optimization Methods

### 2.1 Nonlinear Conjugate Gradient

Nonlinear conjugate gradient (NCG) methods [9] are used to solve unconstrained nonlinear optimization problems. They are extensions of the conjugate gradient iterative method for solving linear systems adapted to solve unconstrained nonlinear optimization problems.

The Poblano function for the nonlinear conjugate gradient methods is called `ncg`.

#### 2.1.1 Method Description

The general steps of NCG methods are given below in high-level pseudo-code:

1. Input:  $x_0$ , a starting point
2. Evaluate  $f_0 = f(x_0)$ ,  $g_0 = \nabla f(x_0)$
3. Set  $p_0 = -g_0$ ,  $i = 0$
4. **while**  $\|g_i\| > 0$
5.     Compute a step length  $\alpha_i$  and set  $x_{i+1} = x_i + \alpha_i p_i$
6.     Set  $g_i = \nabla f(x_{i+1})$
7.     Compute  $\beta_{i+1}$
8.     Set  $p_{i+1} = -g_{i+1} + \beta_{i+1} p_i$
9.     Set  $i = i + 1$
10. **end while**
11. Output:  $x_i \approx x^*$

**Conjugate direction updates.** The computation of  $\beta_{i+1}$  in Step 7 above leads to different NCG methods. The update methods for  $\beta_{i+1}$  available in Poblano are listed in Table 7. Note that the special case of  $\beta_{i+1} = 0$  leads to the steepest descent method [9], which is available in Poblano by specifying this update using the input parameter described in Table 8.

Update Type	Update Formula
Fletcher-Reeves [3]	$\beta_{i+1} = \frac{g_{i+1}^T (-g_{i+1})}{g_i^T g_i}$
Polak-Ribiere [12]	$\beta_{i+1} = \frac{g_{i+1}^T (g_{i+1} - g_i)}{g_i^T g_i}$
Hestenes-Stiefel [5]	$\beta_{i+1} = \frac{g_{i+1}^T (g_{i+1} - g_i)}{p_i^T (g_{i+1} - g_i)}$

Table 7: Conjugate direction updates available in Poblano.

**Negative update coefficients.** In cases where the update coefficient is negative, i.e.,  $\beta_{i+1} < 0$ , it is set to 0 to avoid directions that are not descent directions [9].

**Restart procedures.** The NCG iterations are restarted every  $k$  iterations, where  $k$  is specified by user by setting the `RestartIters` parameter. The default number of iterations performed before restart is 20.

Another restart modification available is taking a step in the direction of steepest descent when two consecutive gradients are far from orthogonal [9]. Specifically, a steepest descent step is taking when

$$\frac{|g_{i+1}^T g_i|}{\|g_{i+1}\|} \geq \nu$$

where  $\nu$  is specified by the user by setting the `RestartNWTol` parameter. This modification is off by default, but can be used by setting the `RestartNW` parameter to `true`.

### 2.1.2 Method Specific Input Parameters

The input parameters specific to the `ncg` method are presented in Table 8. See §1.4 for details on the shared Poblano input parameters.

Parameter	Description	Type	Default
<code>Update</code>	Conjugate direction update 'FR': Fletcher-Reeves 'PR': Polak-Ribiere 'HS': Hestenes-Stiefel 'SD': Steepest Descent	char	'PR'
<code>RestartIters</code>	Number of iterations to run before conjugate direction restart	int	20
<code>RestartNW</code>	Flag to use restart heuristic of Nocedal and Wright	boolean	false
<code>RestartNWTol</code>	Tolerance for Nocedal and Wright restart heuristic	double	0.1

Table 8: Method specific parameters for Poblano's `ncg` optimization method.

### 2.1.3 Examples

**Example 1 (from §1.6.1).** In this example, we have  $x \in \mathbb{R}^{10}$ ,  $a = 3$ , and use a random starting point.

```
>> randn('state',0);
>> x0 = randn(10,1)
>> out = ncg(@(x) example1(x,3), x0)
```

```
x0 =
-0.432564811528221
-1.6655843782381
 0.125332306474831
 0.287676420358549
-1.14647135068146
 1.190915465643
 1.1891642016521
-0.0376332765933176
 0.327292361408654
 0.174639142820925
```

Iter	FuncEvals	F(X)	G(X)  /N
0	1	1.80545257	0.73811114
1	5	-4.10636797	0.54564169
2	8	-5.76811976	0.52039618
3	12	-7.62995880	0.25443887
4	15	-8.01672533	0.06329092
5	20	-9.51983614	0.28571759
6	25	-9.54169917	0.27820083
7	28	-9.99984082	0.00535271
8	30	-10.00000000	0.00000221

```

out =
  Params: [1x1 inputParser]
  ExitFlag: 0
  X: [10x1 double]
  F: -9.9999999997292
  G: [10x1 double]
  FuncEvals: 30
  Iters: 8

```

**Example 2 (from §1.6.2).** In this example, we compute a rank-4 approximation to a  $4 \times 4$  Pascal matrix (generated using the Matlab function `pascal(4)`). The starting point is random vector. Note that in the interest of space, Poblano is set to display only the final iteration in this example.

```

>> m = 4; n = 4; k = 4;
>> Data.rank = k;
>> Data.A = pascal(m);
>> randn('state',0);
>> x0 = randn((m+n)*k,1);
>> out = ncg(@(x) example2(x,Data), x0, 'Display', 'final')

```

Iter	FuncEvals	F(X)	G(X)  /N
39	101	0.00005424	0.00136338

```

out =
  Params: [1x1 inputParser]
  ExitFlag: 2
  X: [32x1 double]
  F: 5.42435362453638e-05
  G: [32x1 double]
  FuncEvals: 101
  Iters: 39

```

The fact that `out.ExitFlag>0` indicates that the method did not converge to the specified tolerance (i.e., the default `StopTol` input parameter value of  $10^{-5}$ ). From [Table 5](#), we see that this exit flag indicates that the maximum number of function evaluations was exceeded. Increasing the number of maximum numbers of function evaluations and iterations allowed, the optimizer converges to a solution within the specified tolerance.

```
>> out = ncg(@(x) example2(x,Data), x0, 'MaxIters',1000, ...
    'MaxFuncEvals',10000,'Display','final')
```

Iter	FuncEvals	F(X)	G(X)  /N
76	175	0.00000002	0.00000683

```
out =
    Params: [1x1 inputParser]
    ExitFlag: 0
           X: [32x1 double]
           F: 1.64687596791954e-08
           G: [32x1 double]
    FuncEvals: 175
           ITERS: 76
```

Verifying the solution, we see that we find a matrix decomposition which fits the matrix with very small relative error (given the stopping tolerance of  $10^{-5}$  used by the optimizer).

```
>> [U,V] = example2_extract(m,n,k,out.X);
>> norm(Data.A-U*V')/norm(Data.A)
```

```
ans =
    5.48283898063955e-06
```

## 2.2 Limited-memory BFGS

Limited-memory quasi-Newton methods [9] are a class of methods that compute and/or maintain simple, compact approximations of the Hessian matrices of second derivatives, which are used determining search directions. Poblano includes the limited-memory BFGS (L-BFGS) method, a variant of these methods whose Hessian approximations are based on the BFGS method (see [2] for more details).

The Poblano function for the L-BFGS method is called `lbfgs`.

### 2.2.1 Method Description

The general steps of L-BFGS methods are given below in high-level pseudo-code [9]:

1. Input:  $x_0$ , a starting point;  $M > 0$ , an integer
2. Evaluate  $f_0 = f(x_0)$ ,  $g_0 = \nabla f(x_0)$
3. Set  $p_0 = -g_0$ ,  $\gamma_0 = 1$ ,  $i = 0$
4. **while**  $\|g_i\| > 0$
5.     Choose an initial Hessian approximation:  $H_i^0 = \gamma_i I$
6.     Compute a step direction  $p_i = -r$  using `TwoLoopRecursion` method
7.     Compute a step length  $\alpha_i$  and set  $x_{i+1} = x_i + \alpha_i p_i$
8.     Set  $g_i = \nabla f(x_{i+1})$
9.     **if**  $i > M$
10.         Discard vectors  $\{s_{i-m}, y_{i-m}\}$  from storage
11.     **end if**
12.     Set and store  $s_i = x_{i+1} - x_i$  and  $y_i = g_{i+1} - g_i$
13.     Set  $i = i + 1$
14. **end while**
15. Output:  $x_i \approx x^*$

**Computing the step direction.** In Step 6 in the above method, the computation of the step direction is performed using the following method (assume we are at iteration  $i$ ) [9]:

- TwoLoopRecursion**
1.  $q = g_i$
  2. **for**  $k = i - 1, i - 2, \dots, i - m$
  3.      $a_k = (s_k^T q) / (y_k^T s_k)$
  4.      $q = q - a_k y_k$
  5. **end for**
  6.  $r = H_i^0 q$
  7. **for**  $k = i - m, i - m + 1, \dots, i - 1$
  8.      $b = (y_k^T r) / (y_k^T s_k)$
  9.      $r = r + (a_k - b) s_k$
  10. **end for**
  11. Output:  $r = H_i g_i$

### 2.2.2 Method Specific Input Parameters

The input parameters specific to the `lbfgs` method are presented in Table 9. See §1.4 for details on the shared Poblano input parameters.

Parameter	Description	Type	Default
M	Limited memory parameter (i.e., number of vectors $s$ and $y$ to store from previous iterations)	int	5

Table 9: Method specific parameters for Poblano’s `lbfgs` optimization method.

### 2.2.3 Examples

Below are the results of using the `lbfgs` method in Poblano to solve example problems solved using the `ncg` method in §2.1.3. Note the different methods leads to slightly different solutions and a different number of function evaluations.

**Example 1.** In this example, we have  $x \in R^{10}$  and  $a = 3$ , and use a random starting point.

```
>> randn('state',0);
>> x0 = randn(10,1)
>> out = lbfgs(@(x) example1(x,3), x0)
```

```
x0 =
-0.432564811528221
-1.6655843782381
0.125332306474831
0.287676420358549
-1.14647135068146
1.190915465643
1.1891642016521
-0.0376332765933176
0.327292361408654
0.174639142820925
```

Iter	FuncEvals	F(X)	G(X)  /N
0	1	1.80545257	0.73811114
1	5	-4.10636797	0.54564169
2	9	-5.78542020	0.51884777
3	12	-6.92220212	0.41545340
4	15	-7.59202813	0.29710231
5	18	-7.95865922	0.25990099
6	21	-8.18606788	0.22457133
7	24	-9.21285389	0.35468422
8	27	-9.68427296	0.23233142
9	30	-9.83727919	0.16936958
10	32	-9.91014546	0.12619212
11	35	-9.94394171	0.10012075
12	37	-9.96737038	0.07645454
13	39	-9.99998661	0.00155264
14	40	-10.00000000	0.00002739
15	42	-10.00000000	0.00000010



```

out =
  Params: [1x1 inputParser]
  ExitFlag: 0
  X: [10x1 double]
  F: -9.999999999999995
  G: [10x1 double]
  FuncEvals: 42
  ITERS: 15

```

**Example 2.** In this example, we compute a rank 2 approximation to a  $4 \times 4$  Pascal matrix (generated using the Matlab function `pascal(4)`). The starting point is a random vector.

```

>> m = 4; n = 4; k = 4;
>> Data.rank = k;
>> Data.A = pascal(m);
>> randn('state',0);
>> x0 = randn((m+n)*k,1);
>> out = lbfgs(@(x) example2(x,Data), x0, 'Display', 'final')

```

Iter	FuncEvals	F(X)	G(X)  /N
46	100	0.00000584	0.00032464

```

out =
  Params: [1x1 inputParser]
  ExitFlag: 2
  X: [32x1 double]
  F: 5.84160204756528e-06
  G: [32x1 double]
  FuncEvals: 100
  ITERS: 46

```

As for the `ncg` method, the fact that `out.ExitFlag>0` indicates that the method did not converge to the specified tolerance (i.e., the default `StopTol` input parameter value of  $10^{-5}$ ). Since the maximum number of function evaluations was exceeded, we can increase the number of maximum numbers of function evaluations and iterations allowed, and the optimizer converges to a solution within the specified tolerance.

```

>> out = lbfgs(@(x) example2(x,Data), x0, 'MaxIters',1000, ...
  'MaxFuncEvals',10000,'Display','final')

```

Iter	FuncEvals	F(X)	G(X)  /N
67	143	0.00000000	0.00000602

```
out =  
  Params: [1x1 inputParser]  
  ExitFlag: 0  
  X: [32x1 double]  
  F: 3.39558864327836e-09  
  G: [32x1 double]  
  FuncEvals: 143  
  Iters: 67
```

Verifying the solution, we see that we find a matrix decomposition which fits the matrix with very small relative error (given the stopping tolerance of  $10^{-5}$  used by the optimizer).

```
>> [U,V] = example2_extract(m,n,k,out.X);  
>> norm(Data.A-U*V')/norm(Data.A)
```

```
ans =  
 2.54707192131722e-06
```

For Example 2, we see that `lbfgs` requires slightly fewer function evaluations than `ncg` to solve the problem (143 versus 175). Performance of the different methods in Poblano is dependent on both the method and the particular parameters chosen. Thus, it is recommended that several test runs on smaller problems are performed initially using the different methods to help decide which method and set of parameters works best for a particular class of problems.

## 2.3 Truncated Newton

Truncated Newton (TN) methods for minimization are Newton methods in which the Newton direction is only approximated at each iteration (thus reducing computation). Furthermore, the Poblano implementation of the truncated Newton method does not require an explicit Hessian matrix in the computation of the approximate Newton direction (thus reducing storage requirements).

The Poblano function for the truncated Newton method is called `tn`.

### 2.3.1 Method Description

The general steps of the TN method in Poblano is given below in high-level pseudo-code [1]:

1. Input:  $x_0$ , a starting point;  $N > 0$ , an integer;  $\eta_0$
2. Evaluate  $f_0 = f(x_0), g_0 = \nabla f(x_0)$
3. Set  $i = 0$
4. **while**  $\|g_i\| > 0$
5.     Compute the conjugate gradient stopping tolerance,  $\eta_i$
6.     Compute  $p_i$  by solving  $\nabla^2 f(x_i)p = -g_i$  using a linear conjugate gradient (CG) method
7.     Compute a step length  $\alpha_i$  and set  $x_{i+1} = x_i + \alpha_i p_i$
8.     Set  $g_i = \nabla f(x_{i+1})$
9.     Set  $i = i + 1$
10. **end while**
11. Output:  $x_i \approx x^*$

### Notes

- In Step 5, the linear conjugate gradient (CG) method stopping tolerance is allowed to change at each iteration. The input parameter `CGTolType` determines how  $\eta_i$  is computed.
- In Step 6
  - One of Matlab’s CG methods is used to solve for  $p_i$ : `symmlq` (designed for symmetric indefinite systems) or `pcg` (the classical CG method for symmetric positive definite systems). The input parameter `CGSolver` controls the choice of CG method to use.
  - The maximum number of CG iterations,  $N$ , is specified using the input parameter `CGIters`.
  - The CG method stops when  $\| -g_i - \nabla^2 f(x_i)p_i \| \leq \eta_i \|g_i\|$  .
  - In the CG method, matrix-vector products involving  $\nabla^2 f(x_i)$  times a vector  $v$  are approximated using the following finite difference approximation [1]:

$$\nabla^2 f(x_i)v \approx \frac{\nabla f(x_i + \sigma v) - \nabla f(x_i)}{\sigma}$$

The difference step,  $\sigma$ , is specified using the input parameter `HessVecFDStep`. The computation of the finite difference approximation is performed using the `hessvec_fd` provided with Poblano.

### 2.3.2 Method Specific Input Parameters

The input parameters specific to the `tn` method are presented in Table 10. See §1.4 for details on the shared Poblano input parameters.

Parameter	Description	Type	Default
<code>CGIters</code>	Maximum number of conjugate gradient iterations allowed	int	5
<code>CGTolType</code>	CG stopping tolerance type used 'quadratic': $\ R\ /\ G\  < \min(0.5, \ G\ )$ 'superlinear': $\ R\ /\ G\  < \min(0.5, \sqrt{\ G\ })$ 'fixed': $\ R\  < \text{CGTol}$ $R$ is the residual and $G$ is the gradient	char	'quadratic'
<code>CGTol</code>	CG stopping tolerance when <code>CGTolType</code> is 'fixed'	double	1e-6
<code>HessVecFDStep</code>	Hessian vector product finite difference step 0 : Use iterate-based step: $10^{-8}(1 + \ X\ _2)$ > 0 : Fixed value to use at the difference step	double	$10^{-10}$
<code>CGSolver</code>	Matlab CG method used to solve for search direction 'symmlq': symmetric LQ method [11] 'pcg': classical CG method [5]	string	'symmlq'

Table 10: Method specific parameters for Poblano's `tn` optimization method.

### 2.3.3 Examples

Below are the results of using the `tn` method in Poblano to solve example problems solved using the `ncg` method in §2.1.3 and `lbfgs` method in §2.2.3.

**Example 1.** In this example, we have  $x \in R^{10}$  and  $a = 3$ , and use a random starting point.

```
>> randn('state',0);
>> x0 = randn(10,1)
>> out = tn(@(x) example1(x,3), x0)
```

```
x0 =
-0.432564811528221
-1.6655843782381
 0.125332306474831
 0.287676420358549
-1.14647135068146
 1.190915465643
 1.1891642016521
-0.0376332765933176
 0.327292361408654
 0.174639142820925
```

Iter	FuncEvals	F(X)	G(X)  /N
0	1	1.80545257	0.73811114
tn: line search warning = 0			
1	10	-4.10636797	0.54564169
2	18	-4.21263331	0.41893997
tn: line search warning = 0			
3	28	-7.18352472	0.36547546
4	34	-8.07095085	0.11618518
tn: line search warning = 0			
5	41	-9.87251163	0.15057476
6	46	-9.99999862	0.00049753
7	50	-10.00000000	0.00000000

```

out =
  Params: [1x1 inputParser]
  ExitFlag: 0
  X: [10x1 double]
  F: -10
  G: [10x1 double]
  FuncEvals: 50
  ITERS: 7

```

Note that in this example the Moré-Thuente line search in `tn` method displays a warning during iterations 1, 3 and 5, indicating that the norm of the search direction is nearly 0. In those cases, the steepest descent direction is used for the search direction during those iterations.

**Example 2.** In this example, we compute a rank 2 approximation to a  $4 \times 4$  Pascal matrix (generated using the Matlab function `pascal(4)`). The starting point is a random vector.

```

>> m = 4; n = 4; k = 4;
>> Data.rank = k;
>> Data.A = pascal(m);
>> randn('state',0);
>> x0 = randn((m+n)*k,1);
>> out = tn(@(x) example2(x,Data), x0, 'Display', 'final')

```

Iter	FuncEvals	F(X)	G(X)  /N
16	105	0.00013951	0.00060739

```

out =
  Params: [1x1 inputParser]
  ExitFlag: 2
  X: [32x1 double]
  F: 1.395095104051596e-04
  G: [32x1 double]
  FuncEvals: 105
  ITERS: 16

```

As for the `ncg` and `lbfgs` methods, the fact that `out.ExitFlag>0` indicates that the method did not converge to the specified tolerance (i.e., the default `StopTol` input parameter value of  $10^{-5}$ ). Since the maximum number of function evaluations was exceeded, we can increase the number of maximum numbers of function evaluations and iterations allowed, and the optimizer converges to a solution within the specified tolerance.

```
>> out = tn(@(x) example2(x,Data), x0, 'MaxIters',1000, ...
    'MaxFuncEvals',10000,'Display','final')
```

```

Iter  FuncEvals      F(X)          ||G(X)||/N
-----
    21      155      0.00000001      0.00000606
```

```
out =
  Params: [1x1 inputParser]
  ExitFlag: 0
  X: [32x1 double]
  F: 9.418354535331997e-09
  G: [32x1 double]
  FuncEvals: 155
  Iters: 21
```

Verifying the solution, we see that we find a matrix decomposition which fits the matrix with very small relative error (given the stopping tolerance of  $10^{-5}$  used by the optimizer).

```
>> [U,V] = example2_extract(m,n,k,out.X);
>> norm(Data.A-U*V')/norm(Data.A)
```

```
ans =
    4.442629309639341e-06
```

### 3 Checking Gradient Calculations

Analytic gradients can be checked using finite difference approximations. The Poblano function `gradientcheck` computes the gradient approximations and compares the results to the analytic gradient using a user-supplied objective function/gradient M-file. The user can choose one of several difference formulas as well as the difference step used in the computations.

#### 3.1 Difference Formulas

The difference formulas for approximating the gradients in Poblano are listed in [Table 11](#). For more details on the different formulas, see [\[9\]](#).

Formula Type	Formula
Forward Differences	$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x)}{h}$
Backward Differences	$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x) - f(x - he_i)}{h}$
Centered Differences	$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x - he_i)}{2h}$

Note:  $e_i$  is a vector the same size as  $x$  with a 1 in element  $i$  and zeros elsewhere, and  $h$  is a user-defined parameter.

Table 11: Difference formulas available in Poblano for checking user-defined gradients.

The type of finite differences to use is specified using the `DifferenceType` input parameter, and the value of  $h$  is specified using the `DifferenceStep` input parameter. For a detailed discussion on the impact of the choice of  $h$  on the quality of the approximation, see [\[10\]](#).

#### 3.2 Gradient Check Input Parameters

The input parameters available for the `gradientcheck` function are presented in [Table 12](#).

Parameter	Description	Type	Default
<code>DifferenceType</code>	Difference formula to use 'forward': $g_i = (f(x + he_i) - f(x))/h$ 'backward': $g_i = (f(x) - f(x - he_i))/h$ 'centered': $g_i = (f(x + he_i) - f(x - he_i))/(2h)$	char	'forward'
<code>DifferenceStep</code>	Value of $h$ in difference formulae	double	$10^{-8}$

Table 12: Input parameters for Poblano's `gradientcheck` function.

### 3.3 Gradient Check Output Parameters

The fields in the structure of output parameters generated by the `gradientcheck` function are presented in Table 13.

Parameter	Description	Type
<code>G</code>	Analytic gradient	vector, double
<code>GFD</code>	Finite difference approximation of gradient	vector, double
<code>MaxDiff</code>	Maximum difference between <code>G</code> and <code>GFD</code>	double
<code>MaxDiffInd</code>	Index of maximum difference between <code>G</code> and <code>GFD</code>	int
<code>NormGradientDiffs</code>	Norm of the difference between gradients: $\ G - GFD\ _2$	double
<code>GradientDiffs</code>	<code>G - GFD</code>	vector, double
<code>Params</code>	Input parameters used to compute approximations	struct

Table 13: Output parameters generated by Poblano's `gradientcheck` function.

### 3.4 Examples

We use Example 1 (described in detail in §1.6.1) to illustrate how to use the `gradientcheck` function to check user-supplied gradients. The user provides a function handle to the M-file containing their function and gradient computations, a point at which to check the gradients, and the type of difference formula to use. Below is an example of running the gradient check using each of the difference formulas.

```
>> outFD = gradientcheck(@(x) example1(x,3), pi./[4 5 6]', 'DifferenceType', 'forward')
>> outBD = gradientcheck(@(x) example1(x,3), pi./[4 5 6]', 'DifferenceType', 'backward')
>> outCD = gradientcheck(@(x) example1(x,3), pi./[4 5 6]', 'DifferenceType', 'centered')
```

```
outFD =
        G: [3x1 double]
       GFD: [3x1 double]
   MaxDiff: 6.4662e-08
 MaxDiffInd: 1
NormGradientDiffs: 8.4203e-08
 GradientDiffs: [3x1 double]
       Params: [1x1 struct]
```

```
outBD =
        G: [3x1 double]
       GFD: [3x1 double]
   MaxDiff: -4.4409e-08
 MaxDiffInd: 3
NormGradientDiffs: 5.2404e-08
 GradientDiffs: [3x1 double]
       Params: [1x1 struct]
```



```
outCD =
      G: [3x1 double]
     GFD: [3x1 double]
  MaxDiff: 2.0253e-08
  MaxDiffInd: 1
  NormGradientDiffs: 2.1927e-08
  GradientDiffs: [3x1 double]
  Params: [1x1 struct]
```

Note the different gradients produced using the various differencing formulas:

```
>> format long
>> [outFD.G outFD.GFD outBD.GFD outCD.GFD]
```

```
ans =
-2.121320343559642 -2.121320408221550 -2.121320319403708 -2.121320363812629
-0.927050983124842 -0.927051013732694 -0.927050969323773 -0.927050991528233
 0.000000000000000 -0.00000044408921  0.00000044408921  0
```

*This page intentionally left blank.*

## 4 Numerical Experiments

To demonstrate the performance of the Poblano methods, we present results of runs of the different methods in this section. All experiments were performed using Matlab 7.9 on a Linux Workstation (RedHat 5.2) with 2 Quad-Core Intel Xeon 3.0GHz processors and 32GB RAM.

### 4.1 Description of Test Problems

The test problems used in the experiments presented here are from the Moré, Garbow, and Hillstom collection, which is described in detail in [7]. The Matlab code used for these problems is provided as part of the SolvOpt optimization software [6], available at <http://www.kfunigraz.ac.at/imawww/kuntsevich/solvopt/>. There are 34 test problems in this collection.

### 4.2 Results

Results in this section are for optimization runs using the default parameters, with the exception of the stopping criteria parameters, which were changed to allow more computation and find more accurate solutions. The parameters changed from their default values are as follows:

```
params.Display = 'off'  
params.MaxIters = 20000;  
params.MaxFuncEvals = 50000;  
params.RelFuncTol = 1e-16;  
params.StopTol = 1e-12;
```

For all of the results presented in this section, the function value computed by the Poblano methods is denoted by  $\hat{F}^*$ , the solution is denoted by  $F^*$ , and the error is  $|F^* - \hat{F}^*| / \max\{1, |F^*|\}$ . The “solutions” are those reported as having the lowest function values throughout the literature. A survey of the problem characteristics, along with known local optimizers can be found on the SolvOpt web site (<http://www.kfunigraz.ac.at/imawww/kuntsevich/solvopt/results.html>).

For this test collection, we say the problem is solved if the error as defined above is less than  $10^{-8}$ . We see that the Poblano methods solve most of the problems using the default input parameters, but have difficulty with a few that require particular parameter settings. Specifically, the initial step size in the line search method appears to be the most sensitive parameter across all of the methods. More investigation into the effective use of this parameter for other problem classes is planned for future work. Below are more details of the results for the different Poblano methods.

#### 4.2.1 Nonlinear Conjugate Gradient Methods

The results of the tests for the `ncg` method using Polak-Ribiere (PR) conjugate direction updates are presented in Table 14. Note that several problems (3, 6, 10, 11, 17, 20, 27 and 31) were not solved using the parameters listed above. With more testing, specifically with different values of the initial step size used in the line search (specified using the `LineSearch_initialstep` parameter) and the number of iterations to perform before restarting the conjugate directions using a step in the steepest direction (specified using the `RestartIters` parameter), solutions to all problems except #10 were found. Table 15 presents parameter choices leading to successful runs of the `ncg` method using PR conjugate direction updates.

The results of the tests for the `ncg` method using Hestenes-Stiefel (HS) conjugate direction updates are presented in Table 16. Again, several problems (3, 6, 10, 11 and 31) were not solved using the parameters

listed above. [Table 17](#) presents parameter choices leading to successful runs of the `ncg` method using HS conjugate direction updates. Note that we did not find parameters to solve problem #10 with this method either.

Finally, the results of the tests for the `ncg` method using Fletcher-Reeves (FR) conjugate direction updates are presented in [Table 18](#). We see that even more problems (3, 6, 10, 11, 17, 18, 20 and 31) were not solved using the parameters listed above. [Table 19](#) presents parameter choices leading to successful runs of the `ncg` method using FR conjugate direction updates. Note that we did not find parameters to solve problem #10 with this method either.

#### 4.2.2 Limited Memory BFGS Method

The results of the tests for the `lbfgs` method are presented in [Table 20](#). Compared to the `ncg` methods, fewer problems (3, 6 and 31) were not solved using the parameters listed above with the `lbfgs` method. Most notably, `lbfgs` was able to solve problem #10, illustrating that some problems are better suited to the different Poblano methods. With more testing, specifically with different values of the initial step size used in the line search (specified using the `LineSearch_initialstep` parameter), solutions to all problems were found. [Table 21](#) presents parameter choices leading to successful runs of the `lbfgs` method.

#### 4.2.3 Truncated Newton Method

The results of the tests for the `tn` method are presented in [Table 22](#). Note that several problems (10, 11, 20 and 31) were not solved using this method. However, problem #6 was solved (which was not solved using any of the other Poblano methods), again illustrating that some problems are better suited to `tn` than the other Poblano methods. With more testing, specifically with different values of the initial step size used in the line search (specified using the `LineSearch_initialstep` parameter), solutions to all problems were found, with the exception of problem #10. [Table 23](#) presents parameter choices leading to successful runs of the `tn` method.

#	Exit	Iters	FuncEvals	$\hat{F}^*$	$F^*$	Error
1	3	20	90	1.3972e-18	0.0000e+00	1.3972e-18
2	3	11	71	4.8984e+01	4.8984e+01	5.8022e-16
3	3	3343	39373	6.3684e-07	0.0000e+00	<b>6.3684e-07</b>
4	3	14	126	1.4092e-21	0.0000e+00	1.4092e-21
5	3	12	39	1.7216e-17	0.0000e+00	1.7216e-17
6	0	1	8	2.0200e+03	1.2436e+02	<b>1.5243e+01</b>
7	3	53	158	5.2506e-18	0.0000e+00	5.2506e-18
8	3	28	108	8.2149e-03	8.2149e-03	1.2143e-17
9	0	5	14	1.1279e-08	1.1279e-08	2.7696e-14
10	3	262	1232	9.5458e+04	8.7946e+01	<b>1.0844e+03</b>
11	0	1	2	3.8500e-02	0.0000e+00	<b>3.8500e-02</b>
12	0	15	62	3.9189e-32	0.0000e+00	3.9189e-32
13	3	129	352	1.5737e-16	0.0000e+00	1.5737e-16
14	3	47	155	2.7220e-18	0.0000e+00	2.7220e-18
15	3	58	165	3.0751e-04	3.0751e-04	3.9615e-10
16	3	34	168	8.5822e+04	8.5822e+04	4.3537e-09
17	3	3122	7970	5.5227e-05	5.4649e-05	<b>5.7842e-07</b>
18	3	1181	2938	2.1947e-16	0.0000e+00	2.1947e-16
19	3	381	876	4.0138e-02	4.0138e-02	2.9355e-10
20	3	14842	30157	1.4233e-06	1.4017e-06	<b>2.1546e-08</b>
21	3	20	90	6.9775e-18	0.0000e+00	6.9775e-18
22	3	129	352	1.5737e-16	0.0000e+00	1.5737e-16
23	0	90	415	2.2500e-05	2.2500e-05	2.4991e-11
24	3	142	520	9.3763e-06	9.3763e-06	7.4196e-15
25	3	5	27	7.4647e-25	0.0000e+00	7.4647e-25
26	3	47	144	2.7951e-05	2.7951e-05	2.1878e-13
27	3	2	8	8.2202e-03	0.0000e+00	<b>8.2202e-03</b>
28	3	79	160	1.4117e-17	0.0000e+00	1.4117e-17
29	3	7	16	9.1648e-19	0.0000e+00	9.1648e-19
30	3	30	75	1.0410e-17	0.0000e+00	1.0410e-17
31	3	41	97	3.0573e+00	0.0000e+00	<b>3.0573e+00</b>
32	0	2	5	1.0000e+01	1.0000e+01	0.0000e+00
33	3	2	5	4.6341e+00	4.6341e+00	0.0000e+00
34	3	2	5	6.1351e+00	6.1351e+00	0.0000e+00

Table 14: Results of `ncg` using PR updates on the Moré, Garbow, Hillstrom test collection. Errors greater than  $10^{-8}$  are highlighted in bold, indicating that a solution was not found within the specified tolerance.

#	Input Parameters	Iters	FuncEvals	$\hat{F}^*$	Error
3	LineSearch.initialstep = 1e-5 RestartIters = 40	19	152	9.9821e-09	9.9821e-09
6	LineSearch.initialstep = 1e-5 RestartIters = 40	19	125	1.2436e+02	3.5262e-11
11	LineSearch.initialstep = 1e-2 RestartIters = 40	407	2620	2.7313e-14	2.7313e-14
17	LineSearch.initialstep = 1e-3	714	1836	5.4649e-05	1.1032e-11
20	LineSearch.initialstep = 1e-1 RestartIters = 50	3825	8066	1.4001e-06	1.6303e-09
27	LineSearch.initialstep = 0.5	11	38	1.3336e-25	1.3336e-25
31	LineSearch.initialstep = 1e-7	17	172	1.9237e-18	1.9237e-18

Table 15: Parameter changes that lead to solutions using `ncg` with PR updates on the Moré, Garbow, Hillstrom test collection.

#	Exit	Iters	FuncEvals	$\hat{F}^*$	$F^*$	Error
1	3	20	89	1.0497e-16	0.0000e+00	1.0497e-16
2	3	10	69	4.8984e+01	4.8984e+01	2.9011e-16
3	2	4329	50001	4.4263e-08	0.0000e+00	<b>4.4263e-08</b>
4	3	9	83	1.3553e-18	0.0000e+00	1.3553e-18
5	3	12	39	4.3127e-19	0.0000e+00	4.3127e-19
6	0	1	8	2.0200e+03	1.2436e+02	<b>1.5243e+01</b>
7	3	27	98	1.8656e-20	0.0000e+00	1.8656e-20
8	3	25	92	8.2149e-03	8.2149e-03	1.2143e-17
9	3	5	31	1.1279e-08	1.1279e-08	2.7696e-14
10	3	49	332	1.0856e+05	8.7946e+01	<b>1.2334e+03</b>
11	0	1	2	3.8500e-02	0.0000e+00	<b>3.8500e-02</b>
12	0	15	63	5.8794e-30	0.0000e+00	5.8794e-30
13	3	84	258	2.7276e-17	0.0000e+00	2.7276e-17
14	3	44	152	3.4926e-24	0.0000e+00	3.4926e-24
15	3	50	165	3.0751e-04	3.0751e-04	3.9615e-10
16	3	39	176	8.5822e+04	8.5822e+04	4.3537e-09
17	3	942	2754	5.4649e-05	5.4649e-05	4.3964e-12
18	3	748	1977	2.7077e-18	0.0000e+00	2.7077e-18
19	3	237	607	4.0138e-02	4.0138e-02	2.9356e-10
20	3	4685	9977	1.3999e-06	1.4017e-06	1.7946e-09
21	0	22	93	6.1630e-32	0.0000e+00	6.1630e-32
22	3	84	258	2.7276e-17	0.0000e+00	2.7276e-17
23	0	83	381	2.2500e-05	2.2500e-05	2.4991e-11
24	3	170	691	9.3763e-06	9.3763e-06	7.3568e-15
25	3	5	47	7.4570e-25	0.0000e+00	7.4570e-25
26	3	43	144	2.7951e-05	2.7951e-05	2.1878e-13
27	3	9	40	2.5738e-20	0.0000e+00	2.5738e-20
28	3	33	70	8.3475e-18	0.0000e+00	8.3475e-18
29	3	7	16	9.1699e-19	0.0000e+00	9.1699e-19
30	3	29	73	3.4886e-18	0.0000e+00	3.4886e-18
31	3	33	117	3.0573e+00	0.0000e+00	<b>3.0573e+00</b>
32	0	2	5	1.0000e+01	1.0000e+01	0.0000e+00
33	3	2	5	4.6341e+00	4.6341e+00	0.0000e+00
34	3	2	5	6.1351e+00	6.1351e+00	0.0000e+00

Table 16: Results of `ncg` using HS updates on the Moré, Garbow, Hillstrom test collection. Errors greater than  $10^{-8}$  are highlighted in bold, indicating that a solution was not found within the specified tolerance.

#	Input Parameters	Iters	FuncEvals	$\hat{F}^*$	Error
3	LineSearch.initialstep = 1e-5 RestartIters = 40	57	395	3.3512e-09	3.3512e-09
6	LineSearch.initialstep = 1e-5	18	105	1.2436e+02	3.5262e-11
11	LineSearch.initialstep = 1e-2 RestartIters = 40	548	3288	8.1415e-15	8.1415e-15
31	LineSearch.initialstep = 1e-7	17	172	1.6532e-18	1.6532e-18

Table 17: Parameter changes that lead to solutions using `ncg` with HS updates on the Moré, Garbow, Hillstrom test collection.

#	Exit	Iters	FuncEvals	$\hat{F}^*$	$F^*$	Error
1	3	41	199	2.4563e-17	0.0000e+00	2.4563e-17
2	3	25	104	4.8984e+01	4.8984e+01	4.3517e-16
3	3	74	538	1.2259e-03	0.0000e+00	<b>1.2259e-03</b>
4	3	40	304	2.9899e-13	0.0000e+00	2.9899e-13
5	3	27	76	4.7847e-17	0.0000e+00	4.7847e-17
6	0	1	8	2.0200e+03	1.2436e+02	<b>1.5243e+01</b>
7	3	48	153	3.5829e-19	0.0000e+00	3.5829e-19
8	3	46	144	8.2149e-03	8.2149e-03	3.4694e-18
9	3	7	72	1.1279e-08	1.1279e-08	2.7696e-14
10	3	388	2098	3.0166e+04	8.7946e+01	<b>3.4201e+02</b>
11	0	1	2	3.8500e-02	0.0000e+00	<b>3.8500e-02</b>
12	3	19	65	2.6163e-18	0.0000e+00	2.6163e-18
13	3	308	720	1.8955e-16	0.0000e+00	1.8955e-16
14	3	53	219	3.0620e-17	0.0000e+00	3.0620e-17
15	3	83	232	3.0751e-04	3.0751e-04	3.9615e-10
16	3	42	206	8.5822e+04	8.5822e+04	4.3537e-09
17	3	522	2020	5.5389e-05	5.4649e-05	<b>7.4046e-07</b>
18	3	278	644	5.6432e-03	0.0000e+00	<b>5.6432e-03</b>
19	3	385	1003	4.0138e-02	4.0138e-02	2.9355e-10
20	3	14391	29061	2.7316e-06	1.4017e-06	<b>1.3298e-06</b>
21	3	41	199	1.2282e-16	0.0000e+00	1.2282e-16
22	3	308	720	1.8955e-16	0.0000e+00	1.8955e-16
23	3	192	783	2.2500e-05	2.2500e-05	2.4991e-11
24	3	1239	4256	9.3763e-06	9.3763e-06	7.3782e-15
25	0	5	29	1.7256e-31	0.0000e+00	1.7256e-31
26	3	60	187	2.7951e-05	2.7951e-05	2.1877e-13
27	3	10	34	1.0518e-16	0.0000e+00	1.0518e-16
28	3	112	226	1.7248e-16	0.0000e+00	1.7248e-16
29	3	7	16	1.9593e-18	0.0000e+00	1.9593e-18
30	3	30	74	1.1877e-17	0.0000e+00	1.1877e-17
31	3	58	166	3.0573e+00	0.0000e+00	<b>3.0573e+00</b>
32	0	2	5	1.0000e+01	1.0000e+01	3.5527e-16
33	3	2	5	4.6341e+00	4.6341e+00	0.0000e+00
34	3	2	5	6.1351e+00	6.1351e+00	0.0000e+00

Table 18: Results of `ncg` using FR updates on the Moré, Garbow, Hillstrom test collection. Errors greater than  $10^{-8}$  are highlighted in bold, indicating that a solution was not found within the specified tolerance.

#	Input Parameters	Iters	FuncEvals	$\hat{F}^*$	Error
3	LineSearch_initialstep = 1e-5 RestartIters = 50	128	422	2.9147e-09	2.9147e-09
6	LineSearch_initialstep = 1e-5	52	257	1.2436e+02	3.5262e-11
11	LineSearch_initialstep = 1e-2 RestartIters = 40	206	905	4.3236e-13	4.3236e-13
17	LineSearch_initialstep = 1e-3 RestartIters = 40	421	1012	5.4649e-05	1.7520e-11
18	LineSearch_initialstep = 1e-4	1898	12836	3.2136e-17	3.2136e-17
20	LineSearch_initialstep = 1e-1 RestartIters = 50	3503	7262	1.4001e-06	1.6392e-09
31	LineSearch_initialstep = 1e-7	16	162	8.2352e-18	8.2352e-18

Table 19: Parameter changes that lead to solutions using `ncg` with FR updates on the Moré, Garbow, Hillstrom test collection.

#	Exit	Iters	FuncEvals	$\hat{F}^*$	$F^*$	Error
1	3	43	145	1.5549e-20	0.0000e+00	1.5549e-20
2	3	14	73	4.8984e+01	4.8984e+01	4.3517e-16
3	3	206	730	5.8432e-20	0.0000e+00	5.8432e-20
4	3	14	29	7.8886e-31	0.0000e+00	7.8886e-31
5	3	20	55	1.2898e-24	0.0000e+00	1.2898e-24
6	0	1	8	2.0200e+03	1.2436e+02	<b>1.5243e+01</b>
7	0	40	103	8.6454e-28	0.0000e+00	8.6454e-28
8	0	27	63	8.2149e-03	8.2149e-03	1.7347e-18
9	0	4	9	1.1279e-08	1.1279e-08	2.7696e-14
10	3	705	2406	8.7946e+01	8.7946e+01	4.2594e-13
11	0	1	2	3.8500e-02	0.0000e+00	<b>3.8500e-02</b>
12	3	24	76	3.2973e-20	0.0000e+00	3.2973e-20
13	3	77	165	1.7747e-17	0.0000e+00	1.7747e-17
14	3	66	187	1.2955e-19	0.0000e+00	1.2955e-19
15	0	29	72	3.0751e-04	3.0751e-04	3.9615e-10
16	3	22	96	8.5822e+04	8.5822e+04	4.3537e-09
17	3	221	602	5.4649e-05	5.4649e-05	9.7483e-13
18	3	56	165	5.6556e-03	0.0000e+00	<b>5.6556e-03</b>
19	3	267	713	4.0138e-02	4.0138e-02	2.9355e-10
20	3	5118	10752	1.3998e-06	1.4017e-06	1.9194e-09
21	3	44	146	2.8703e-24	0.0000e+00	2.8703e-24
22	3	77	165	1.7747e-17	0.0000e+00	1.7747e-17
23	0	218	786	2.2500e-05	2.2500e-05	2.4991e-11
24	3	458	1492	9.3763e-06	9.3763e-06	7.3554e-15
25	0	4	31	1.4730e-29	0.0000e+00	1.4730e-29
26	3	41	135	2.7951e-05	2.7951e-05	2.1879e-13
27	3	15	37	9.7350e-24	0.0000e+00	9.7350e-24
28	3	67	140	1.2313e-16	0.0000e+00	1.2313e-16
29	3	7	15	2.5466e-21	0.0000e+00	2.5466e-21
30	3	25	54	1.5036e-17	0.0000e+00	1.5036e-17
31	3	27	100	3.0573e+00	0.0000e+00	<b>3.0573e+00</b>
32	0	2	4	1.0000e+01	1.0000e+01	7.1054e-16
33	3	2	4	4.6341e+00	4.6341e+00	0.0000e+00
34	3	2	4	6.1351e+00	6.1351e+00	0.0000e+00

Table 20: Results of `lbfgs` on the Moré, Garbow, Hillstrom test collection. Errors greater than  $10^{-8}$  are highlighted in bold, indicating that a solution was not found within the specified tolerance.

#	Input Parameters	Iters	FuncEvals	$\hat{F}^*$	Error
6	<code>LineSearch_initialstep = 1e-5</code>	29	293	1.2436e+02	3.5262e-11
11	<code>LineSearch_initialstep = 1e-2</code>	220	1102	1.5634e-20	1.5634e-20
18	<code>LineSearch_initialstep = 0</code> <code>M = 1</code>	220	1102	9.3766e-17	9.3766e-17
31	<code>LineSearch_initialstep = 1e-7</code>	16	209	8.1617e-19	8.1617e-19

Table 21: Parameter changes that lead to solutions using `lbfgs` on the Moré, Garbow, Hillstrom test collection.



#	Exit	Iters	FuncEvals	$\hat{F}^*$	$F^*$	Error
1	0	105	694	4.9427e-30	0.0000e+00	4.9427e-30
2	3	9	107	4.8984e+01	4.8984e+01	2.9011e-16
3	3	457	3607	3.1474e-09	0.0000e+00	3.1474e-09
4	0	6	36	0.0000e+00	0.0000e+00	0.0000e+00
5	0	13	96	4.4373e-31	0.0000e+00	4.4373e-31
6	3	13	133	1.2436e+02	1.2436e+02	3.5262e-11
7	0	39	245	6.9237e-46	0.0000e+00	6.9237e-46
8	0	9	80	8.2149e-03	8.2149e-03	8.6736e-18
9	0	3	28	1.1279e-08	1.1279e-08	2.7696e-14
10	2	6531	50003	1.1104e+05	8.7946e+01	<b>1.2616e+03</b>
11	0	1	5	3.8500e-02	0.0000e+00	<b>3.8500e-02</b>
12	0	9	74	1.8615e-28	0.0000e+00	1.8615e-28
13	0	25	241	2.3427e-29	0.0000e+00	2.3427e-29
14	0	36	263	1.0481e-26	0.0000e+00	1.0481e-26
15	0	449	4967	3.0751e-04	3.0751e-04	3.9615e-10
16	3	28	237	8.5822e+04	8.5822e+04	4.3537e-09
17	3	2904	30801	5.4649e-05	5.4649e-05	3.3347e-10
18	3	2622	28977	6.3011e-17	0.0000e+00	6.3011e-17
19	3	526	5671	4.0138e-02	4.0138e-02	2.9355e-10
20	2	4654	50006	6.0164e-06	1.4017e-06	<b>4.6147e-06</b>
21	0	96	638	3.9542e-28	0.0000e+00	3.9542e-28
22	0	25	241	2.3427e-29	0.0000e+00	2.3427e-29
23	0	17	207	2.2500e-05	2.2500e-05	2.4991e-11
24	0	63	827	9.3763e-06	9.3763e-06	7.3554e-15
25	0	3	19	2.3419e-31	0.0000e+00	2.3419e-31
26	3	18	249	2.7951e-05	2.7951e-05	2.1878e-13
27	0	6	53	2.7820e-29	0.0000e+00	2.7820e-29
28	3	82	882	1.9402e-17	0.0000e+00	1.9402e-17
29	0	4	33	9.6537e-32	0.0000e+00	9.6537e-32
30	3	16	104	3.4389e-21	0.0000e+00	3.4389e-21
31	3	13	145	3.0573e+00	0.0000e+00	<b>3.0573e+00</b>
32	3	4	63	1.0000e+01	1.0000e+01	5.3291e-16
33	3	4	48	4.6341e+00	4.6341e+00	1.9166e-16
34	3	3	19	6.1351e+00	6.1351e+00	0.0000e+00

Table 22: Results of `tn` on the Moré, Garbow, Hillstrom test collection. Errors greater than  $10^{-8}$  are highlighted in bold, indicating that a solution was not found within the specified tolerance.

#	Input Parameters	Iters	FuncEvals	$\hat{F}^*$	Error
11	LineSearch.initialstep = 1e-3 HessVecFDStep = 1e-12	1018	17744	4.0245e-09	4.0245e-09
20	CGIters = 50	34	1336	1.3998e-06	1.9715e-09
31	LineSearch.initialstep = 1e-7	7	133	4.4048e-25	4.4048e-25

Table 23: Parameter changes that lead to solutions using `tn` on the Moré, Garbow, Hillstrom test collection.

*This page intentionally left blank.*

## 5 Conclusions

We have presented Poblano v1.0, a Matlab Toolbox for unconstrained optimization requiring only first order derivatives. Details of the methods available in Poblano as well as how to use the toolbox to solve unconstrained optimization problems were provided. Demonstration of the Poblano solvers on the Moré, Garbow and Hillstom collection of test problems indicates good performance in general of the Poblano optimizer methods across a wide range of problems.

## References

- [1] R. DEMBO AND T. STEIHAUG, *Truncated-Newton algorithms for large-scale unconstrained optimization*, *Mathematical Programming*, 26 (1983), pp. 190–212.
- [2] J. E. DENNIS, JR. AND R. B. SCHNABEL, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, SIAM, Philadelphia, PA, 1996. Corrected reprint of the 1983 original.
- [3] R. FLETCHER AND C. REEVES, *Function minimization by conjugate gradients*, *The Computer Journal*, 7 (1964), pp. 149–154.
- [4] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, Johns Hopkins Univ. Press, 1996.
- [5] M. R. HESTENES AND E. STIEFEL, *Methods of conjugate gradients for solving linear systems*, *J. Res. Nat. Bur. Standards Sec. B.*, 48 (1952), pp. 409–436.
- [6] A. KUNTSEVICH AND F. KAPPEL, *SolvOpt: The solver for local nonlinear optimization problems*, tech. rep., Institute for Mathematics, Karl-Franzens University of Graz, June 1997. <http://www.kfunigraz.ac.at/imawww/kuntsevich/solvopt/>.
- [7] J. J. MORÉ, B. S. GARBOW, AND K. E. HILLSTROM, *Testing unconstrained optimization software*, *ACM Trans. Math. Software*, 7 (1981), pp. 17–41.
- [8] J. J. MORÉ AND D. J. THUENTE, *Line search algorithms with guaranteed sufficient decrease*, *ACM Transactions on Mathematical Software*, 20 (1994), pp. 286–307.
- [9] J. NOCEDAL AND S. J. WRIGHT, *Numerical Optimization*, Springer, 1999.
- [10] M. L. OVERTON, *Numerical Computing with IEEE Floating Point Arithmetic*, Society for Industrial and Applied Mathematics, 2001.
- [11] C. C. PAIGE AND M. A. SAUNDERS, *Solution of sparse indefinite systems of linear equations*, *SIAM J. Numer. Anal.*, Vol.12 (1975), pp. 617–629.
- [12] E. POLAK AND G. RIBIERE, *Note sur la convergence de methods de directions conjuges*, *Revue Française Informat. Recherche Operationnelle*, 16 (1969), pp. 35–43.

## DISTRIBUTION:

- 1 MS 0899      Technical Library, 9536 (electronic)
- 1 MS 0123      D. Chavez, LDRD Office, 1011 (electronic)







**Sandia National Laboratories**