

ARGONNE NATIONAL LABORATORY  
9700 South Cass Avenue  
Argonne, Illinois 60439

## **TAO Users Manual**

**Lois Curfman McInnes**  
**Jorge J. Moré**  
**Todd Munson**  
**Jason Sarich**

Mathematics and Computer Science Division

Technical Report ANL/MCS-TM-242-Revision 1.10.1

This manual is intended for use with TAO version 1.10.1

April 19, 2011

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.



# Contents

<b>1</b>	<b>Introduction to TAO</b>	<b>1</b>
1.1	TAO Design Philosophy . . . . .	1
1.2	Performance Results . . . . .	3
<b>2</b>	<b>Getting Started</b>	<b>5</b>
2.1	Writing Application Codes with TAO . . . . .	5
2.2	A Simple TAO Example . . . . .	6
2.3	Include Files . . . . .	6
2.4	TAO Initialization . . . . .	6
2.5	TAO Finalization . . . . .	8
2.6	TAO Solvers . . . . .	8
2.7	Function Evaluations . . . . .	9
2.8	TAO Programming with PETSc . . . . .	9
2.9	Compiling and Running TAO . . . . .	10
2.10	Error Checking . . . . .	11
2.11	Makefiles . . . . .	12
2.12	Directory Structure . . . . .	13
<b>3</b>	<b>Basic Usage of TAO Solvers</b>	<b>15</b>
3.1	Initialize and Finalize . . . . .	15
3.2	Creation and Destruction . . . . .	15
3.3	Convergence . . . . .	16
3.4	Viewing Solutions . . . . .	17
<b>4</b>	<b>TAO Solvers</b>	<b>19</b>
4.1	Unconstrained Minimization . . . . .	19
4.1.1	Nelder-Mead . . . . .	19
4.1.2	Limited-Memory, Variable-Metric Method . . . . .	20
4.1.3	Nonlinear Conjugate Gradient Method . . . . .	23
4.1.4	Newton Line-Search Method . . . . .	24
4.1.5	Newton Trust-Region Method . . . . .	26
4.2	Bound Constrained Optimization . . . . .	28
4.2.1	Newton Trust Region . . . . .	28
4.2.2	Gradient Projection–Conjugate Gradient Method . . . . .	29
4.2.3	Interior Point Newton Algorithm . . . . .	29

4.2.4	Limited Memory Variable Metric Method . . . . .	29
4.2.5	KT Method . . . . .	29
4.3	Complementarity . . . . .	29
4.3.1	Semismooth Methods . . . . .	30
<b>5</b>	<b>TAO Applications using PETSc</b>	<b>35</b>
5.1	Header File . . . . .	35
5.2	Create and Destroy . . . . .	35
5.3	Defining Variables . . . . .	36
5.4	Application Context . . . . .	36
5.5	Objective Function and Gradient Routines . . . . .	37
5.6	Hessian Evaluation . . . . .	38
5.6.1	Finite Differences . . . . .	39
5.6.2	Matrix-Free methods . . . . .	39
5.7	Bounds on Variables . . . . .	40
5.8	Complementarity . . . . .	40
5.9	Monitors . . . . .	41
5.10	Linear Solvers . . . . .	41
5.11	Application Solutions . . . . .	41
5.12	Linear Algebra Abstractions . . . . .	42
5.13	Compiling and Linking . . . . .	43
5.14	TAO Applications using PETSc and FORTRAN . . . . .	44
5.14.1	Include Files . . . . .	44
5.14.2	Error Checking . . . . .	45
5.14.3	Compiling and Linking Fortran Programs . . . . .	45
5.14.4	Additional Issues . . . . .	46
<b>6</b>	<b>Advanced Options</b>	<b>47</b>
6.1	Convergence Tests . . . . .	47
6.2	Line Searches . . . . .	47
<b>7</b>	<b>Adding a solver</b>	<b>49</b>
7.1	Adding a Solver to TAO . . . . .	49
7.2	TAO Interface with Solvers . . . . .	50
7.2.1	Solver Routine . . . . .	50
7.2.2	Creation Routine . . . . .	52
7.2.3	Destroy Routine . . . . .	53
7.2.4	SetUp Routine . . . . .	54

## Preface

The Toolkit for Advanced Optimization (TAO) focuses on the development of algorithms and software for the solution of large-scale optimization problems on high-performance architectures. Areas of interest include nonlinear least squares, unconstrained and bound-constrained optimization, and general nonlinear optimization.

The development of TAO was motivated by the scattered support for parallel computations and the lack of reuse of external toolkits in current optimization software. Our aim is to use object-oriented techniques to produce high-quality optimization software for a range of computing environments ranging from serial workstations and laptops to massively parallel high-performance architectures. Our design decisions are strongly motivated by the challenges inherent in the use of large-scale distributed memory architectures and the reality of working with large, often poorly structured legacy codes for specific applications.

This manual describes the use of TAO. Since TAO is still under development, changes in usage and calling sequences may occur. TAO is fully supported; see the web site <http://www.mcs.anl.gov/tao> for information on contacting the TAO developers.

## Acknowledgments

The initial development of TAO was funded by the ACTS Toolkit Project in the Office of Advanced Scientific Computing Research, U.S. Department of Energy. We gratefully acknowledge their support.

TAO owes much to the developers of PETSc. We have benefitted from their experience, tools, software, and advice. In many ways, TAO is a natural outcome of the PETSc development. TAO has also benefitted from the work of various researchers who have provided solvers, test problems, and interfaces. In particular, we acknowledge

- Lisa Grignon for contributing the least squares examples `chebyq.c`, `coating.c` and `enzreac1.c`;
- Yurii Zinchenko and Mike Gertz for the interface to the OOQP solver for quadratic problems with linear constraints;
- Liz Dolan for developing the HTML version of the TAO user guide
- Boyana Norris for developing prototype CCA-compliant optimization component interfaces.
- Gabriel Lopez-Calva for integrating ADIC with TAO on Distributed Array applications and developing examples that use them; and
- Jarek Nieplocha, Limin Zhang, and Manojkumar Krishnan for the interface between Global Arrays and TAO and implemented example applications.

Finally, we thank all TAO users for their comments, bug reports, and encouragement.



# Chapter 1

## Introduction to TAO

The Toolkit for Advanced Optimization (TAO) focuses on the design and implementation of optimization software for the solution of large-scale optimization applications on high-performance architectures. Our approach is motivated by the scattered support for parallel computations and lack of reuse of linear algebra software in currently available optimization software. The TAO design allows the reuse of toolkits that provide lower-level support (parallel sparse matrix data structures, preconditioners, solvers), and thus we are able to build on top of these toolkits instead of having to redevelop code. The advantages in terms of efficiency and development time are significant.

The TAO design philosophy uses object-oriented techniques of data and state encapsulation, abstract classes, and limited inheritance to create a flexible optimization toolkit. This chapter provides a short introduction to our design philosophy by describing the objects in TAO and the importance of this design. Since a major concern in the TAO project is the performance and scalability of optimization algorithms on large problems, we also present some performance results.

### 1.1 TAO Design Philosophy

The TAO design philosophy place strong emphasis on the reuse of external tools where appropriate. Our design enables bidirectional connection to lower-level linear algebra support (e.g. parallel sparse matrix data structures) provided in toolkits such as PETSc [3] [4, 2] as well as higher-level application frameworks. Our design decisions are strongly motivated by the challenges inherent in the use of large-scale distributed memory architectures and the reality of working with large and often poorly structured legacy codes for specific applications. Figure 1.1 illustrates how the TAO software works with external libraries and application code.

The TAO solvers use four fundamental objects to define and solve optimization problems: vectors, index sets, matrices, and linear solvers. The concepts of vectors and matrices are standard, while an index set refers to a set of integers used to identify particular elements of vectors or matrices. An optimization algorithm is a sequence of well defined operations on these objects. These operations include vector sums, inner products, and matrix-vector multiplication. TAO makes no assumptions about the representation of these objects by

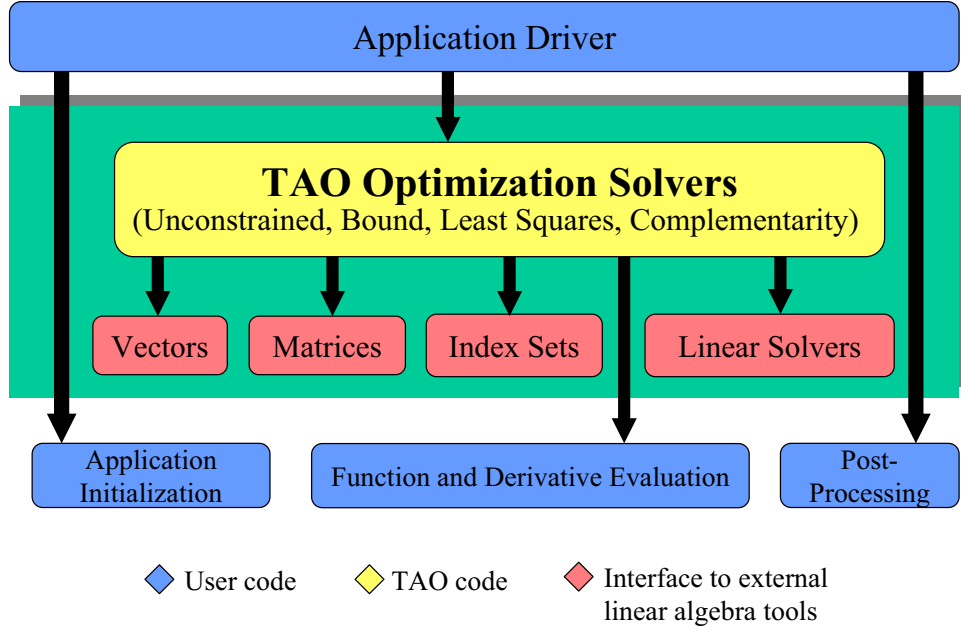


Figure 1.1: TAO Design

passing pointers to data-structure-neutral objects for the execution of these numerical operations.

With sufficiently flexible abstract interfaces, TAO can support a variety of implementations of data structures and algorithms. These abstractions allow us to more easily experiment with a range of algorithmic and data structure options for realistic problems, such as within this case study. Such capabilities are critical for making high-performance optimization software adaptable to the continual evolution of parallel and distributed architectures and the research community’s discovery of new algorithms that exploit their features.

Our current TAO implementation uses the parallel system infrastructure and linear algebra objects offered by PETSc, which uses MPI [13] for all interprocessor communication. The PETSc package supports objects for vectors, matrices, index sets, and linear solvers.

The TAO design philosophy eliminates some of the barriers in using independently developed software components by accepting data that is independent of representation and calling sequence written for particular data formats. The user can initialize an application with external frameworks, provide function information to a TAO solver, and call TAO to solve the application problem.

The use of abstractions for matrices and vectors in TAO optimization software also enables us to leverage automatic differentiation technology to facilitate the parallel computation of gradients and Hessians needed within optimization algorithms. We have demonstrated the viability of this approach through preliminary interfacing between TAO solvers and the automatic differentiation tools ADIFOR and ADIC. We are currently working on developing TAO interfaces that use special problem features (for example, partial separability, stencil information) in automatic differentiation computations.



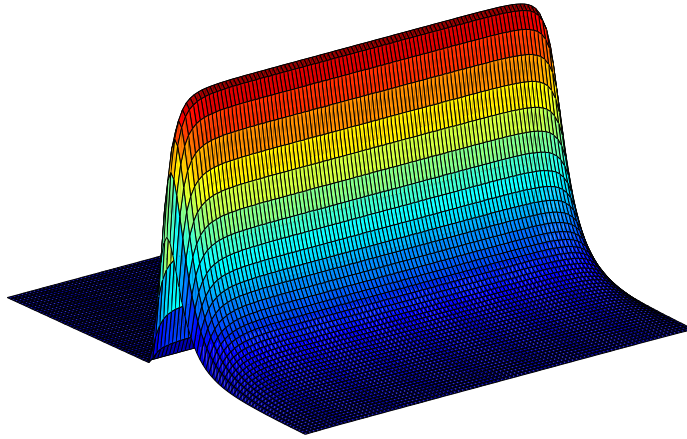


Figure 1.2: The journal bearing problem with  $\epsilon = 0.9$

## 1.2 Performance Results

A major concern in the TAO project is the performance and scalability of optimization algorithms on large problems. In this section we focus on the GPCG (gradient projection, conjugate gradient) algorithm for the solution of bound-constrained convex quadratic programming problems. Originally developed by Moré and Toraldo [20], the GPCG algorithm was designed for large-scale problems but had only been implemented for a single processor. GPCG combines the advantages of the identification properties of the gradient projection method with the finite termination properties of the conjugate gradient method. Moreover, the performance of the TAO implementation on large optimization problems is noteworthy.

We illustrate the performance of the GPCG algorithm by presenting results for a journal bearing problem with over 2.5 million variables. The journal bearing problem is a finite element approximation to a variational problem over a rectangular two-dimensional grid. A grid with 1600 points in each direction, for example, is formulated as a bound constrained quadratic problem with  $1600^2 = 2,560,000$  variables. The triangulation of the grid results in a matrix that has the usual five diagonal nonzero structure that arises from a difference approximation to the Laplacian operator. The journal bearing problem contains an eccentricity parameter,  $\epsilon \in (0, 1)$ , that influences the number of active variables at the solution and the difficulty in solving it. Figure 1.2 shows the solution of the journal bearing problem for  $\epsilon = 0.9$ . The steep gradient in the solution makes this problem a difficult benchmark.

The performance results in Table 1.1 are noteworthy in several ways. First of all, the number of faces visited by GPCG is remarkably small. Other strategies can lead to a large number of gradient projection iterates, but the GPCG algorithm is remarkably efficient. Another interesting aspect of these results is that due to the low memory requirements of iterative solvers, we were able to solve these problems with only  $p = 8$  processors. Strategies that rely on direct solvers are likely to need significantly more storage, and thus

more processors. Finally, these results show that the GPCG implementation has excellent efficiency. For example, the efficiency of GPCG with respect to  $p = 8$  processors ranges between 70% and 100% when  $\varepsilon = 0.1$ . This sustained efficiency is remarkable since the GPCG algorithm is solving a sequence of linear problems with a coefficient matrix set to the submatrix of the Hessian matrix with respect to the free variables for the current iterate. Thus, our implementation's repartitioning of submatrices effectively deals with the load-balancing problem that is inherent in the GPCG algorithm.

$\varepsilon$	$p$	faces	$n_{CG}$	time	$t_{CG}\%$	$\mathcal{E}$
0.1	8	46	431	7419	86	100
0.1	16	45	423	3706	83	100
0.1	32	45	427	2045	82	91
0.1	64	45	427	1279	82	73
0.9	8	37	105	2134	70	100
0.9	16	37	103	1124	71	95
0.9	32	38	100	618	69	86
0.9	64	38	99	397	68	67

Table 1.1: Performance of GPCG on the journal bearing problem with  $2.56 \cdot 10^6$  variables.

An important aspect of our results that is not apparent from Table 1.1 is that for these results we were able to experiment easily with all the preconditioners offered by PETSc. In particular, we were able to compare the diagonal Jacobi preconditioner with block Jacobi and overlapping additive Schwarz preconditioners that use a zero-fill ILU solver in each block. We also experimented with a parallel zero-fill incomplete Cholesky preconditioner provided by a PETSc interface to the BlockSolve95 [15] package of Jones and Plassmann. Interestingly enough, the diagonal Jacobi preconditioner achieved better performance on this problem.

## Chapter 2

# Getting Started

TAO can be used on a personal computer with a single processor or within a parallel environment. Its basic usage involves only a few commands, but fully understanding its usage requires time. Application programmers can easily begin to use TAO by working with some examples provided in the package and then gradually learn more details according to their needs. The current version of TAO and the most recent help concerning the installation and usage of TAO can be found at <http://www.mcs.anl.gov/tao/>.

The current version (1.10.1) of TAO requires an ANSI C++ compiler, an implementation of MPI, Version 3.1 of PETSc compiled with the C++ compiler, (PETSc must be configured with the `--with-clanguage=C++` option) and at least 15 MB of free disk space. During the setup process, the user will have to set an environmental variable, `TAO_DIR`, indicating the full path of the TAO home directory. This variable will be used in this manual to refer to the location of files, and by computers that will compile TAO source code.

## 2.1 Writing Application Codes with TAO

The examples throughout the library demonstrate the software usage and can serve as templates for developing custom applications. We suggest that new TAO users examine programs in

```
${TAO_DIR}/src/examples .
```

Additional examples are available on our website and in

```
${TAO_DIR}/src/<unconstrained,bound,...>/examples/tutorials,
```

where `<component>` denotes any of the TAO components, such as `bound` or `unconstrained`. The HTML version of the manual pages located at

```
${TAO_DIR}/docs/manualpages/index.html
```

or

```
http://www.mcs.anl.gov/tao/documentation/manualpages/index.html
```

provides indices (organized by both routine names and concepts) to the tutorial examples.

We suggest the following procedure for writing a new application program using TAO:

1. Install TAO according to the instructions in <http://www.mcs.anl.gov/tao/documentation/index.html>.

2. Copy the examples and makefile from the directory `${TAO_DIR}/examples/`, compile the examples, and run the programs.
3. Select the example program matching the application most closely, and use it as a starting point for developing a customized code.

## 2.2 A Simple TAO Example

To help the user start using TAO immediately, we use a simple uniprocessor example. The code in Figure 2.1 is for minimizing the extended Rosenbrock function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  defined by

$$f(x) = \sum_{i=0}^{m-1} (\alpha(x_{2i+1} - x_{2i}^2)^2 + (1 - x_{2i})^2),$$

where  $n = 2m$  is the number of variables. The code in Figure 2.1 is only the main program. We have not included the code for evaluating the function and gradient or for evaluating the Hessian matrix.

Note that while we use the C language to introduce the TAO software, the package is also fully usable from C++ and Fortran77/90. Section 5.14 discusses additional issues concerning Fortran usage.

The code in Figure 2.1 contains many of the components needed to write most TAO programs and thus, is illustrative of the features present in complex optimization problems. Note that we have omitted the code required for the routine `FormFunctionGradient`, which evaluates the function and gradient, and the code for `FormHessian`, which evaluates the Hessian matrix for Rosenbrock's function. The following sections annotate the lines of code in Figure 2.1.

## 2.3 Include Files

The C++ include file for TAO should be used via the statement

```
#include "tao.h"
```

The required lower level include files are automatically included within this high-level file.

## 2.4 TAO Initialization

All TAO programs contain a call to

```
info = TaoInitialize(int *argc, char ***argv, char *file_name,
                    char *help_message);
```

This command initializes TAO (and also MPI and PETSc if these have not yet been initialized elsewhere). The arguments `argc` and `argv` are the command line arguments delivered in all C and C++ programs. The argument `file_name` optionally indicates an alternative name for an options file, which by default is called `.petsrc` and resides in the user's home directory. See the PETSc users manual for details regarding runtime option specification.

```

#include "tao.h"
/* ----- User-defined constructs ----- */
typedef struct {
    int n;          /* dimension */
    double alpha;   /* condition parameter */
} AppCtx;
int FormFunctionGradient(TAO_APPLICATION,Vec,double*,Vec,void*);
int FormHessian(TAO_APPLICATION,Vec,Mat*,Mat*,MatStructure*,void*);

int main(int argc,char **argv)
{
    int      info;          /* used to check for functions returning nonzeros */
    double   zero=0.0;
    Vec      x;             /* solution vector */
    Mat      H;             /* Hessian matrix */
    TAO_SOLVER tao;         /* TAO_SOLVER solver context */
    TAO_APPLICATION taoapp; /* TAO application context */
    AppCtx    user;         /* user-defined application context */

    /* Initialize TAO and PETSc */
    PetscInitialize(&argc,&argv,(char *)0,0);
    TaoInitialize(&argc,&argv,(char *)0,0);
    user.n = 2; user.alpha = 99.0;

    /* Allocate vectors for the solution and gradient */
    info = VecCreateSeq(PETSC_COMM_SELF,user.n,&x); CHKERRQ(info);
    info = MatCreateSeqBDiag(PETSC_COMM_SELF,user.n,user.n,0,2,0,0,&H);CHKERRQ(info);

    /* Create TAO solver with desired solution method */
    info = TaoCreate(PETSC_COMM_SELF,"tao_lvm",&tao); CHKERRQ(info);
    info = TaoApplicationCreate(PETSC_COMM_SELF,&taoapp); CHKERRQ(info);

    /* Set solution vec and an initial guess */
    info = VecSet(&zero,x); CHKERRQ(info);
    info = TaoAppSetInitialSolutionVec(taoapp,x); CHKERRQ(info);

    /* Set routines for function, gradient, hessian evaluation */
    info = TaoAppSetObjectiveAndGradientRoutine(taoapp,FormFunctionGradient,(void *)&user);
    CHKERRQ(info);
    info = TaoAppSetHessianMat(taoapp,H,H); CHKERRQ(info);
    info = TaoAppSetHessianRoutine(taoapp,FormHessian,(void *)&user); CHKERRQ(info);

    /* SOLVE THE APPLICATION */
    info = TaoSolveApplication(taoapp,tao); CHKERRQ(info);

    /* Free TAO data structures */
    info = TaoDestroy(tao); CHKERRQ(info);
    info = TaoAppDestroy(taoapp); CHKERRQ(info);

    /* Free PETSc data structures */
    info = VecDestroy(x); CHKERRQ(info);
    info = MatDestroy(H); CHKERRQ(info);

    /* Finalize TAO */
    TaoFinalize();
    PetscFinalize();
    return 0;
}

```

Figure 2.1: Example of Uniprocessor TAO Code

The final argument, `help_message`, is an optional character string that will be printed if the program is run with the `-help` option.

As illustrated by the `TaoInitialize()` statement above, TAO routines return an integer indicating whether an error has occurred during the call. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. For the C or C++ interface, the error variable is the routine's return value, while for the Fortran version, each TAO routine has as its final argument an integer error variable. Error tracebacks are discussed in Section 2.8.

## 2.5 TAO Finalization

All TAO programs should call `TaoFinalize()` as their final (or nearly final) statement

```
info = TaoFinalize();
```

This routine handles options to be called at the conclusion of the program, and calls `PetscFinalize()` if `TaoInitialize()` began PETSc. If PETSc was initiated externally from TAO (by either the user or another software package), the user is responsible for calling `PetscFinalize()`.

## 2.6 TAO Solvers

The primary commands for solving an unconstrained optimization problem using TAO are shown in Figure 2.2.

```
TaoCreate(MPI_Comm comm, TaoMethod method, TAO_SOLVER *tao);
TaoApplicationCreate(MPI_Comm comm, TAO_APPLICATION *taoapp);
TaoSetInitialSolutionVec(TAO_APPLICATION taoapp, Vec x);
TaoSetObjectiveAndGradientRoutine(TAO_APPLICATION taoapp,
    int (*FormFGGradient)(TAO_APPLICATION,Vec,double,Vec,void*),void *user);
TaoSetHessianMat(TAO_APPLICATION taoapp, Mat H, Mat Hpre);
TaoSetHessianRoutine(TAO_APPLICATION taoapp,
    int (*Hessian)(TAO_APPLICATION, Vec, Mat*, Mat*, MatStructure*, void*), (void *));
TaoSolveApplication(TAO_APPLICATION taoapp, TAO_SOLVER tao);
TaoApplicationDestroy(TAO_APPLICATION taoapp);
TaoDestroy(TAO_SOLVER tao);
```

Figure 2.2: Commands for solving an unconstrained optimization problem

The user first creates the `TAO_SOLVER` and `TAO_APPLICATION` contexts. He then sets call-back routines as well as vector (`Vec`) and matrix (`Mat`) data structures that the TAO solver will use for evaluating the minimization function, gradient, and optionally the Hessian matrix. The user then solves the minimization problem, and finally destroys the `TAO_SOLVER` and `TAO_APPLICATION` contexts. Details of these commands are presented in Chapter 3.

Note that `TaoCreate()` enables the user to select the solution method at runtime by using an options database. Through this database, the user not only can select a minimization method (e.g., limited-memory variable metric, conjugate gradient, Newton with line search or trust region), but also can prescribe the convergence tolerance, set various

monitoring routines, indicate techniques for linear systems solves, etc. See Chapter 3 for more information on the solver methods available in TAO.

## 2.7 Function Evaluations

Users of TAO are required to provide routines that perform function evaluations. Depending on the solver chosen, they may also have to write routines that evaluate the gradient vector and Hessian matrix.

## 2.8 TAO Programming with PETSc

### Include Files

Applications using the PETSc package for vectors, matrices, and linear solvers should include the appropriate header files. For example

```
#include "petscksp.h"
```

includes the appropriate information for most TAO applications using PETSc.

### The Options Database

The user can input control data at run time using an options database. The command

```
PetscOptionsGetInt(PETSC_NULL, "-n", &user.n, &flg);
```

checks whether the user has provided a command line option to set the value of `n`, the number of variables. If so, the variable `n` is set accordingly; otherwise, `n` remains unchanged. A complete description of the options database may be found in the PETSc users manual.

### Vectors

In the example in Figure 2.1, the vector data structure (`Vec`) is used to store the solution and gradient for TAO unconstrained minimization solvers. A new parallel or sequential vector `x` of global dimension `M` is created with the command

```
info = VecCreate(MPI_Comm comm,int m,int M,Vec *x);
```

where `comm` denotes the MPI communicator. The type of storage for the vector may be set with either calls to `VecSetType()` or `VecSetFromOptions()`. Additional vectors of the same type can be formed with

```
info = VecDuplicate(Vec old,Vec *new);
```

The commands

```
info = VecSet(Vec X,PetscScalar value);  
info = VecSetValues(Vec x,int n,int *indices,  
                    Scalar *values,INSERT_VALUES);
```

respectively set all the components of a vector to a particular scalar value and assign a different value to each component. More detailed information about PETSc vectors, including their basic operations, scattering/gathering, index sets, and distributed arrays, may be found in the PETSc users manual.

## Matrices

Usage of matrices and vectors is similar. The user can create a new parallel or sequential matrix `H` with `M` global rows and `N` global columns, with the routine

```
info = MatCreate(MPI_Comm comm,int m,int n,int M,int N,Mat *H);
```

where the matrix format can be specified at runtime. The user could alternatively specify each processes' number of local rows and columns using `m` and `n`. `H` can then be used to store the Hessian matrix, as indicated by the above routine `TaoSetHessianMat()`. Matrix entries can then be set with the command

```
info = MatSetValues(Mat H,int m,int *im,int n,int *in,  
    Scalar *values,INSERT_VALUES);
```

After all elements have been inserted into the matrix, it must be processed with the pair of commands

```
info = MatAssemblyBegin(Mat H,MAT_FINAL_ASSEMBLY);  
info = MatAssemblyEnd(Mat H,MAT_FINAL_ASSEMBLY);
```

The PETSc users manual discusses various matrix formats as well as the details of some basic matrix manipulation routines.

## Parallel Programming

Since TAO uses the message-passing model for parallel programming and employs MPI for all interprocessor communication, the user is free to employ MPI routines as needed throughout an application code. However, by default the user is shielded from many of the details of message passing within TAO, since these are hidden within parallel objects, such as vectors, matrices, and solvers. In addition, TAO users can interface to external tools, such as the generalized vector scatters/gathers and distributed arrays within PETSc, to assist in the management of parallel data.

The user must specify a communicator upon creation of any TAO objects (such as a vector, matrix, or solver) to indicate the processors over which the object is to be distributed. For example, some commands for matrix, vector, and solver creation are:

```
info = MatCreate(MPI_Comm comm,int m,int n,int M,int N,Mat *H);  
info = VecCreate(MPI_Comm comm,int m,int M,Vec *x);  
info = TaoCreate(MPI_Comm comm,TaoMethod method,TAO_SOLVER *tao);
```

The creation routines are collective over all processors in the communicator; thus, all processors in the communicator *must* call the creation routine. In addition, if a sequence of collective routines is being used, the routines *must* be called in the same order on each processor.

## 2.9 Compiling and Running TAO

Compilation of the TAO numerical libraries and TAO application codes requires three environmental variables to be set. These three variables, `TAO_DIR`, `PETSC_ARCH`, and `PETSC_DIR`, are discussed more fully in the TAO installation instructions.

TAO uses a portable makefile system provided by the PETSc [2, 4] library, which is discussed further in Section 2.11. The TAO library can be compiled with the command



```
make
```

from the TAO\_DIR directory.

Running a TAO application on a single processor can be done in the usual way by entering the name of the executable and any command line options. Running programs in parallel, however, requires use of the MPI library. All TAO programs use the MPI (Message Passing Interface) standard for message-passing communication [21]. Thus, to execute TAO programs, users must know the procedure for beginning MPI jobs on their selected computer system(s). For instance, when using the MPICH implementation of MPI [12] and many others, the following command initiates a program that uses eight processors:

```
mpirun -np 8 tao_program_name tao_options
```

## 2.10 Error Checking

All TAO commands begin with the `Tao` prefix and return an integer indicating whether an error has occurred during the call. The error code equals zero after the successful completion of the routine and is set to a nonzero value if an error has been detected. The macro `CHKERRQ(info)` checks the value of `info` and calls an error handler upon error detection. `CHKERRQ()` should be used in all subroutines to enable a complete error traceback.

In Figure 2.3 we indicate a traceback generated by error detection within a sample program. The error occurred on line 1007 of the file `${TAO_DIR}/src/interface/tao.c` in the routine `TaoSetUp()` and was caused by nonconforming local lengths of the parallel gradient and solution vectors, which must be identically partitioned. The `TaoSetUp()` routine was called from the `TaoSolveApplication()` routine, which was in turn called on line 229 of the `main()` routine in the program `ex2.c`. The PETSc users manual provides further details regarding error checking, including information about the Fortran interface.

```
[ember] mpirun -np 2 ex2 -tao_lmvm
[0]PETSC ERROR: TaoSetUp() line 1007 in src/interface/tao.c
[0]PETSC ERROR:   Nonconforming object sizes!
[0]PETSC ERROR:   Gradient and solution vectors must be identically partitioned!
[0]PETSC ERROR: TaoSolveApplication() line 1739 in src/interface/tao.c
[0]PETSC ERROR: main() line 229 in src/unconstrained/examples/tutorials/ex2.c
[0] MPI Abort by user Aborting program !
[0] Aborting program!
p0_911:  p4_error:  : 1
bm_list_912:  p4_error:  interrupt SIGINT: 2
```

Figure 2.3: Example of Error Traceback

When running the debugging version of the TAO software (PETSc configured with the `--with-debugging` option), checking is performed for memory corruption (writing outside of array bounds, etc). The macros `CHKMEMQ` and `CHKMEMA` can be called anywhere in the code to check the current status of the memory for corruption. By putting several (or many) of these macros into an application code, one can usually track down the code segment where corruption has occurred.

## 2.11 Makefiles

To manage code portability across a wide variety of UNIX systems, TAO uses a makefile system that is part of the PETSc software. This section briefly discusses makefile usage from the perspective of application programmers; see the “makefiles” chapter of the PETSc users manual for additional details.

### Compiling TAO Programs

To make a program named `rosenbrock1`, one may use the command

```
make PETSC_ARCH=arch rosenbrock1
```

which compiles a debugging or optimized version of the example and automatically link the appropriate libraries. The architecture, `arch`, is one of `solaris`, `rs6000`, `IRIX`, `hpux`, etc. Note that when using command line options with `make` (as illustrated above), one must *not* place spaces on either side of the “=” signs. The variable `PETSC_ARCH` can also be set as an environmental variable.

### Sample Makefiles

```
CFLAGS      =
FFLAGS      =
CPPFLAGS    =
FPPFLAGS    =

include ${TAO_DIR}/bmake/tao_common

rosenbrock1: rosenbrock1.o tao_chkopts
    -${CLINKER} -o rosenbrock1 rosenbrock1.o ${TAO_LIB} ${PETSC_SNES_LIB}
    ${RM} rosenbrock1.o
```

Figure 2.4: Sample TAO Makefile for a Single Program

Maintaining portable TAO makefiles is very simple. Figure 2.4 presents a minimal makefile for maintaining a single program that uses the TAO library. The most important line in this makefile is the line starting with `include`:

```
include ${TAO_DIR}/bmake/tao_common
```

This line includes other makefiles that provide the needed definitions and rules for the particular base software installations (specified by `${TAO_DIR}` and `${PETSC_DIR}`) and architecture (specified by `${PETSC_ARCH}`), which are typically set as environmental variables prior to compiling TAO source or programs. As listed in the sample makefile, the appropriate `include` file is automatically completely specified; the user should *not* alter this statement within the makefile.

Note that the variable `${TAO_LIB}` (as listed on the link line in this makefile) specifies *all* of the various TAO and supplementary libraries in the appropriate order for correct linking.

Some additional variables that can be used in the makefile are defined as follows:

- `CFLAGS`, `FFLAGS` - user-specified additional options for the C++ compiler and fortran compiler.

- `CPPFLAGS`, `FPPFLAGS` - user-specified additional flags for the C++ preprocessor and fortran preprocessor.
- `CLINKER`, `FLINKER` - the C++ and Fortran linkers.
- `RM` - the remove command for deleting files.
- `TAO_LIB` - all of the base TAO libraries and required supplementary libraries.
- `TAO_FORTRAN_LIB` - the TAO Fortran interface library.
- `PETSC_FORTRAN_LIB` - the PETSc Fortran interface library.

## 2.12 Directory Structure

The home directory of TAO contains the following subdirectories:

- `docs` - All documentation for TAO. The files `tao_manual.ps` and `manual/manual.html` contain the users manual in PDF and HTML formats, respectively. Includes the subdirectory
  - `manualpages` (manual pages for individual TAO routines).
- `bmake` - Base TAO makefile directory.
- `include` - All include files for TAO that are visible to the user.
- `examples` - Example problems and makefile.
- `src` - The source code for all TAO components, which currently includes
  - `unconstrained` - unconstrained minimization,
  - `bound` - bound constrained minimization.
  - `complementarity` - mixed complementarity solvers.
  - `least_squares` - nonlinear least squares,

Each TAO source code component directory has the following subdirectories:

- `examples` - Example programs for the component, including
  - `tutorials` - Programs designed to teach users about TAO. These codes can serve as templates for the design of custom applicatinos.
  - `tests` - Programs designed for thorough testing of TAO. As such, these codes are not intended for examination by users.
- `interface` - The calling sequences for the abstract interface to the component. Code here does not know about particular implementations.
- `impls` - Source code for one or more implementations.
- `utils` - Utility routines. Source here may know about the implementations, but ideally will not know about implementations for other components.



## Chapter 3

# Basic Usage of TAO Solvers

TAO contains unconstrained minimization, bound constrained minimization, and nonlinear complementarity solvers. The structure of these problems can differ significantly, but TAO has a similar interface to all of its solvers. Routines that most solvers have in common will be discussed in this chapter. A complete list of options can be found by consulting the manual pages. Many of the options can also be set at the command line. These options can also be found in manual pages or by running a program with the `-help` option.

### 3.1 Initialize and Finalize

The first TAO routine in any application should be `TaoInitialize()`. Most TAO programs begin with a call to

```
info = TaoInitialize(int *argc, char ***argv, char *file_name,
                    char *help_message);
```

This command initializes TAO, as well as MPI, PETSc, and other packages to which TAO applications may link (if these have not yet been initialized elsewhere). In particular, the arguments `argc` and `argv` are the command line arguments delivered in all C and C++ programs; these arguments initialize the options database. The argument `file_name` optionally indicates an alternative name for an options file, which by default is called `.petsrc` and resides in the user's home directory.

One of the last routines that all TAO programs should call is

```
info = TaoFinalize();
```

This routine finalizes TAO and any other libraries that may have been initialized during the `TaoInitialize()` phase. For example, `TaoFinalize()` calls `MPI_Finalize()` if `TaoInitialize()` began MPI. If MPI was initiated externally from TAO (by either the user or another software package), then the user is responsible for calling `MPI_Finalize()`.

### 3.2 Creation and Destruction

A TAO solver can be created with the command

```
info = TaoCreate(MPI_Comm comm, TaoMethod method, TAO_SOLVER *newsolver);
```

The first argument in this routine is an MPI communicator indicating which processes are involved in the solution process. In most cases, this should be set to `MPI_COMM_WORLD`. The second argument in this creation routine specifies the default method that should be used to solve the optimization problem. The third argument in `TaoCreate()` is a pointer to a TAO solver object. This routine creates the object and returns it to the user. The TAO object is then to be used in all TAO routines.

The various types of TAO solvers and the flags that identify them will be discussed in the following chapters. The solution method should be carefully chosen depending upon the problem that is being solved. Some solvers, for instance, are meant for problems with no constraints, while other solvers acknowledge constraints in the problem and solve them accordingly. The user must also be aware of the derivative information that is available. Some solvers require second-order information, while other solvers require only gradient or function information. The `TaoMethod` can also be set to `TAO_NULL` in the `TaoCreate()` routine if the user selects a method at runtime using the options database. The command line option `-tao_method` followed by an TAO method will override any method specified by the second argument. The command line option `-tao_method tao_lmvm`, for instance, will specify the limited memory variable metric method for unconstrained optimization. Note that the `TaoMethod` variable is a string that requires quotation marks in an application program, but quotation marks are not required at the command line. The method that TAO uses to solve an optimization problem can be changed at a later point in the program with the command `TaoSetMethod()`, whose arguments are a TAO solver and a string that uniquely identifies a method for solving the problem.

Each TAO solver that has been created should also be destroyed using the command

```
info = TaoDestroy(TAO_SOLVER solver);
```

This routine frees the internal data structures used by the solver.

### 3.3 Convergence

Although TAO and its solvers set default parameters that are useful for many problems, it may be necessary for the user to modify these parameters to change the behavior and convergence of various algorithms.

One convergence criterion for most algorithms concerns the number of digits of accuracy needed in the solution. In particular, one convergence test employed by TAO attempts to stop when the error in the constraints is less than  $\epsilon_{crtol}$ , and either

$$\frac{|f(X) - f(X^*)|}{|f(X)| + 1} \leq \epsilon_{frrtol} \text{ or } f(X) - f(X^*) \leq \epsilon_{fatol},$$

where  $X^*$  is the current approximation to  $X$ . TAO estimates  $f(X) - f(X^*)$  with either the square of the norm of the gradient or the duality gap. A relative tolerance of  $\epsilon_{frrtol} = 0.01$  indicates that two significant digits are desired in the objective function. Each solver sets its own convergence tolerances, but they can be changed using the routine `TaoSetTolerances()`. Another set of convergence tolerances can be set with `TaoSetGradientTolerances()`. These tolerances terminate the solver when the norm of the gradient function (or Lagrangian function for bound-constrained problems) is sufficiently close to zero.

Other stopping criteria include a minimum trust region radius or a maximum number of iterations. These parameters can be set with the routines `TaoSetTrustRegionTolerance()` and `TaoSetMaximumIterates()`. Similarly, a maximum number of function evaluations can be set with the command `TaoSetMaximumFunctionEvaluations()`.

### 3.4 Viewing Solutions

The routine

```
int TaoSolveApplication(TAO_APPLICATION, TAO_SOLVER);
```

will apply the solver to the application that has been created by the user.

To see parameters and performance statistics for the solver, the routine

```
int TaoView(TAO_SOLVER);
```

can be used. This routine will display to standard output the number of function evaluations need by the solver and other information specific to the solver.

The progress of the optimization solver can be monitored with the runtime option `-tao_monitor`. Although monitoring routines can be customized, the default monitoring routine will print out several relevant statistics to the screen.

The user also has access to information about the current solution. The current iteration number, objective function value, gradient norm, infeasibility norm, and step length can be retrieved with the command

```
int TaoGetSolutionStatus(TAO_SOLVER tao, int* iterate, double* f,
                        double* gnorm, double *cnorm, double *xdiff,
                        TaoTerminateReason *reason)
```

The last argument returns a code that indicates the reason that the solver terminated. Positive numbers indicate that a solution has been found, while negative numbers indicate a failure. A list of reasons can be found in the manual page for `TaoGetTerminationReason()`.

The user set vectors containing the solution and gradient before solving the problem, but pointers to these vectors can also be retrieved with the commands `TaoGetSolution()` and `TaoGetGradient()`. Dual variables and other relevant information are also available. This information can be obtained during user-defined routines such as a function evaluation and customized monitoring routine, or after the solver has terminated.





# Chapter 4

## TAO Solvers

### 4.1 Unconstrained Minimization

Unconstrained minimization is used to minimize a function of many variables without any constraints on the variables, such as bounds. The methods available in TAO for solving these problems can be classified according to the amount of derivative information required:

1. Function evaluation only – Nelder-Mead method (`tao_nm`)
2. Function and gradient evaluations – limited-memory, variable-metric method (`tao_lmvm`) and nonlinear conjugate gradient method (`tao_cg`)
3. Function, gradient, and Hessian evaluations – Newton line-search method (`tao_nls`) and Newton trust-region method (`tao_ntr`)

The best method to use depends on the particular problem being solved and the accuracy required in the solution. If a Hessian evaluation routine is available, then the Newton line-search and Newton trust-region methods will be the best performers. When a Hessian evaluation routine is not available, then the limited-memory, variable-metric method is likely to perform best. The Nelder-Mead method should be used only as a last resort when no gradient information is available.

Each solver has a set of options associated with it that can be set with command line arguments. A brief description of these algorithms and the associated options are discussed in this chapter.

#### 4.1.1 Nelder-Mead

The Nelder-Mead algorithm [24] is a direct search method for finding a local minimum of a function  $f(x)$ . This algorithm does not require any gradient or Hessian information of  $f$ , and therefore has some expected advantages and disadvantages compared to the other TAO solvers. The obvious advantage is that it is easier to write an application when no derivatives need to be calculated. The downside is that this algorithm can be very slow to converge or can even stagnate, and performs poorly for large numbers of variables.

This solver keeps a set of  $N + 1$  sorted vectors  $x_1, x_2, \dots, x_{N+1}$  and their corresponding objective function values  $f_1 \leq f_2 \leq \dots \leq f_{N+1}$ . At each iteration,  $x_{N+1}$  is removed from

the set and replaced with

$$x(\mu) = (1 + \mu) \frac{1}{N} \sum_{i=1}^N x_i - \mu x_{N+1},$$

where  $\mu$  can be one of  $\mu_0, 2\mu_0, \frac{1}{2}\mu_0, -\frac{1}{2}\mu_0$  depending upon the values of each possible  $f(x(\mu))$ .

The algorithm terminates when the residual  $f_{N+1} - f_1$  becomes sufficiently small. Because of the way new vectors can be added to the sorted set, the minimum function value and/or the residual may not be impacted at each iteration.

There are two options that can be set specifically for the Nelder-Mead algorithm, `-tao_nm_lamda <value>` sets the initial set of vectors ( $x_0$  plus `value` in each cartesian direction), the default value is 1. `tao_nm_mu <value>` sets the value of  $\mu_0$ , the default is  $\mu_0 = 1$ .

### 4.1.2 Limited-Memory, Variable-Metric Method

The limited-memory, variable-metric method solves the system of equations

$$H_k d_k = -\nabla f(x_k),$$

where  $H_k$  is a positive definite approximation to the Hessian matrix obtained by using the BFGS update formula with a limited number of previous iterates and gradient evaluations. The inverse of  $H_k$  can readily be applied to obtain the direction  $d_k$ . Having obtained the direction, a Moré-Thuente line search is applied to compute a step length,  $\tau_k$ , that approximately solves the one-dimensional optimization problem

$$\min_{\tau} f(x_k + \tau d_k).$$

The current iterate and Hessian approximation are updated and the process is repeated until the method converges. This algorithm is the default unconstrained minimization solver and can be selected using the TaoMethod `tao_lmvm`. For best efficiency, function and gradient evaluations should be performed simultaneously when using this algorithm.

The primary factors determining the behavior of this algorithm are the number of vectors stored for the Hessian approximation and the scaling matrix used when computing the direction. The number of vectors stored can be set with the command line argument `-tao_lmm_vectors <int>`; 5 is the default value. Increasing the number of vectors results in a better Hessian approximation and can decrease the number of iterations required to compute a solution to the optimization problem. However, as the number of vectors increases, more memory is consumed and each direction calculation takes longer to compute. Therefore, a trade off must be made between the quality of the Hessian approximation, the memory requirements, and the time to compute the direction.

During the computation of the direction, the inverse of an initial Hessian approximation  $H_{0,k}$  is applied. The choice of  $H_{0,k}$  has a significant impact on the quality of the direction obtained and can result in a decrease in the number of function and gradient evaluations required to solve the optimization problem. However, the calculation of  $H_{0,k}$

at each iteration can have a significant impact on the time required to update the limited-memory BFGS approximation and the cost of obtaining the direction. By default,  $H_{0,k}$  is a diagonal matrix obtained from the diagonal entries of a Broyden approximation to the Hessian matrix. The calculation of  $H_{0,k}$  can be modified with the command line argument `-tao_lmm_scale_type <none,scalar,broyden>`. Each scaling method is described below. The `scalar` and `broyden` techniques are inspired by [?].

**none** This scaling method uses the identity matrix as  $H_{0,k}$ . No extra computations are required when obtaining the search direction or updating the Hessian approximation. However, the number of functions and gradient evaluations required to converge to a solution is typically much larger than the number required when using other scaling methods.

**scalar** This scaling method uses a multiple of the identity matrix as  $H_{0,k}$ . The scalar value  $\sigma$  is chosen by solving the one-dimensional optimization problem

$$\min_{\sigma} \|\sigma^{\alpha} Y - \sigma^{\alpha-1} S\|_F^2,$$

where  $\alpha \in [0, 1]$  is given, and  $S$  and  $Y$  are the matrices of past iterate and gradient information required by the limited-memory BFGS update formula. The optimal value for  $\sigma$  can be written down explicitly. This choice of  $\sigma$  attempts to satisfy the secant equation  $\sigma Y = S$ . Since this equation cannot typically be satisfied by a scalar, a least norm solution is computed. The amount of past iterate and gradient information used is set by the command line argument `tao_lmm_scalar_history <int>`, which must be less than or equal to the number of vectors kept for the BFGS approximation. The default value is 5. The choice for  $\alpha$  is made with the command line argument `tao_lmm_scalar_alpha <double>`; 1 is the default value. This scaling method offers a good compromise between no scaling and `broyden` scaling.

**broyden** This scaling method uses a positive-definite diagonal matrix obtained from the diagonal entries of the Broyden approximation to the Hessian for the scaling matrix. The Broyden approximation is a family of approximations parametrized by a constant  $\phi$ ;  $\phi = 0$  gives the BFGS formula and  $\phi = 1$  gives the DFP formula. The value of  $\phi$  is set with the command line argument `-tao_lmm_broyden_phi <double>`. The default value for  $\phi$  is 0.125. This scaling method requires the most computational effort of available choices, but typically results in a significant reduction in the number of function and gradient evaluations taken to compute a solution.

An additional rescaling of the diagonal matrix can be applied to further improve performance when using the `broyden` scaling method. The rescaling method can be set with the command line argument `-tao_lmm_rescale_type <none,scalar,gl>`; `scalar` is the default rescaling method. The rescaling method applied can have a large impact on the number of function and gradient evaluations necessary to compute a solution to the optimization problem, but increases the time required to update the BFGS approximation. Each rescaling method is described below. These techniques are inspired by [?].

**none** This rescaling method does not modify the diagonal scaling matrix.

**scalar** This rescaling method chooses a scalar value  $\sigma$  by solving the one-dimensional optimization problem

$$\min_{\sigma} \|\sigma^{\alpha} H_{0,k}^{\beta} Y - \sigma^{\alpha-1} H_{0,k}^{\beta-1} S\|_F^2,$$

where  $\alpha \in [0, 1]$  and  $\beta \in [0, 1]$  are given,  $H_{0,k}$  is the positive-definite diagonal scaling matrix computed by using the Broyden update, and  $S$  and  $Y$  are the matrices of past iterate and gradient information required by the limited-memory BFGS update formula. This choice of  $\sigma$  attempts to satisfy the secant equation  $\sigma H_{0,k} Y = S$ . Since this equation cannot typically be satisfied by a scalar, a least norm solution is computed. The scaling matrix used is then  $\sigma H_{0,k}$ . The amount of past iterate and gradient information used is set by the command line argument `tao_lmm_rescale_history <int>`, which must be less than or equal to the number of vectors kept for the BFGS approximation. The default value is 5. The choice for  $\alpha$  is made with the command line argument `tao_lmm_rescale_alpha <double>`; 1 is the default value. The choice for  $\beta$  is made with the command line argument `tao_lmm_rescale_beta <double>`; 0.5 is the default value.

**gl** This scaling method is the same as the **scalar** rescaling method, but the previous value for the scaling matrix  $H_{0,k-1}$  is used when computing  $\sigma$ . This is the rescaling method suggested in [?].

Finally, a limit can be placed on the difference between the scaling matrix computed at this iteration and the previous value for the scaling matrix. The limiting type can be set with the command line argument `-tao_lmm_limit_type <none,average,relative,absolute>`; **none** is the default value. Each of these methods is described below when using the **scalar** scaling method. The techniques are the same when using the **broyden** scaling method, but are applied to each entry in the diagonal matrix.

**none** Set  $\sigma_k = \sigma$ , where  $\sigma$  is the value computed by the scaling method.

**average** Set  $\sigma_k = \mu\sigma + (1 - \mu)\sigma_{k-1}$ , where  $\sigma$  is the value computed by the scaling method,  $\sigma_{k-1}$  is the previous value, and  $\mu \in [0, 1]$  is given.

**relative** Set  $\sigma_k = \text{median}\{(1 - \mu)\sigma_{k-1}, \sigma, (1 + \mu)\sigma_{k-1}\}$ , where  $\sigma$  is the value computed by the scaling method,  $\sigma_{k-1}$  is the previous value, and  $\mu \in [0, 1]$  is given.

**absolute** Set  $\sigma_k = \text{median}\{\sigma_{k-1} - \nu, \sigma, \sigma_{k-1} + \nu\}$ , where  $\sigma$  is the value computed by the scaling method,  $\sigma_{k-1}$  is the previous value, and  $\nu$  is given.

The value for  $\mu$  is set with the command line argument `-tao_lmm_limit_mu <double>`; 1 is the default value. The value for  $\nu$  is set with the command line argument `-tao_lmm_limit_nu <double>`. The default value is 100.

The default values for the scaling are based on many tests using the unconstrained problems from the MINPACK-2 test set. These tests were used to narrow the choices to a few sets of values. These values were then run on the unconstrained problems from the CUTer test set to obtain the default values supplied.

Table 4.1: Summary of `lmm` options

Name	Value	Default	Description
<code>-tao_lmm_vectors</code>	int	5	Number of vectors for Hessian approximation
<code>-tao_lmm_scale.type</code>	none, scalar, broyden	broyden	Type of scaling method to use
<code>-tao_lmm_scalar_history</code>	int	5	Number of vectors to use when scaling
<code>-tao_lmm_scalar.alpha</code>	double	1	Value of $\alpha$ for scalar scaling method
<code>-tao_lmm_broyden_phi</code>	double	0.125	Value of $\alpha$ for scalar scaling method
<code>-tao_lmm_rescale.type</code>	none, scalar, gl	scalar	Type of rescaling method to use
<code>-tao_lmm_rescale_history</code>	int	5	Number of vectors to use when rescaling
<code>-tao_lmm_rescale.alpha</code>	double	1	Value of $\alpha$ for rescaling method
<code>-tao_lmm_rescale.beta</code>	double	0.5	Value of $\beta$ for rescaling method
<code>-tao_lmm_limit.type</code>	none, average, relative, absolute	none	Type of limit to impose on scaling matrix
<code>-tao_lmm_limit.mu</code>	double	1	Value of $\mu$ for limit type
<code>-tao_lmm_limit.nu</code>	double	100	Value of $\nu$ for limit type

### 4.1.3 Nonlinear Conjugate Gradient Method

The nonlinear conjugate gradient method can be viewed as an extensions of the conjugate gradient method for solving symmetric, positive-definite linear systems of equations. This algorithm requires only function and gradient evaluations as well as a line search. The TAO implementation uses a Moré-Thuente line search to obtain the step length. The nonlinear conjugate gradient method can be selected by using the TaoMethod `tao_cg`. For the best efficiency, function and gradient evaluations should be performed simultaneously when using this algorithm.

Five variations are currently supported by the TAO implementation: the Fletcher-Reeves method, the Polak-Ribière method, the Polak-Ribière-Plus method[25], the Hestenes-Stiefel method, and the Dai-Yuan method. These conjugate gradient methods can be specified by using the command line argument `tao_cg.type <fr,pr,prp,hs,dy>`, respectively. The default value is `prp`.

The conjugate gradient method incorporates automatic restarts when successive gradients are not sufficiently orthogonal. TAO measures the orthogonality by dividing the inner product of the gradient at the current point and the gradient at the previous point by the square of the Euclidean norm of the gradient at the current point. When the absolute value of this ratio is greater than  $\eta$ , the algorithm restarts using the gradient direction. The parameter  $\eta$  can be set using the command line argument `-tao_cg_eta <double>`; 0.1 is

the default value.

#### 4.1.4 Newton Line-Search Method

The Newton line-search method solves the symmetric system of equations

$$H_k d_k = -g_k$$

to obtain a step  $d_k$ , where  $H_k$  is the Hessian of the objective function at  $x_k$  and  $g_k$  is the gradient of the objective function at  $x_k$ . For problems where the Hessian matrix is indefinite, the perturbed system of equations

$$(H_k + \rho_k I) d_k = -g_k$$

is solved to obtain the direction, where  $\rho_k$  is a positive constant. If the direction computed is not a descent direction, the (scaled) steepest descent direction is used instead. Having obtained the direction, a Moré-Thuente line search is applied to obtain a step length,  $\tau_k$ , that approximately solves the one-dimensional optimization problem

$$\min_{\tau} f(x_k + \tau d_k).$$

The Newton line-search method can be set using the TaoMethod **tao\_nls**. For the best efficiency, function and gradient evaluations should be performed simultaneously when using this algorithm.

The system of equations is approximately solved by applying the conjugate gradient method, Steihaug-Toint conjugate gradient method, generalized Lanczos method, or an alternative Krylov subspace method supplied by PETSc. The method used to solve the systems of equations is specified with the command line argument **-tao\_nls\_ksp\_type** `<cg,stcg,gltr,petsc>`; **cg** is the default. When the type is set to **petsc**, the method set with the PETSc **-ksp\_type** command line argument is used. For example, to use GMRES as the linear system solver, one would use the the command line arguments **-tao\_nls\_ksp\_type petsc -ksp\_type gmres**. Internally, the PETSc implementations for the conjugate gradient methods and the generalized Lanczos method are used. See the PETSc manual for further information on changing the behavior of the linear system solvers.

A good preconditioner reduces the number of iterations required to solve the linear system of equations. For the conjugate gradient methods and generalized Lanczos method, this preconditioner must be symmetric and positive definite. The available options are to use no preconditioner, the absolute value of the diagonal of the Hessian matrix, a limited-memory BFGS approximation to the Hessian matrix, or one of the other preconditioners provided by the PETSc package. These preconditioners are specified by the command line argument **-tao\_nls\_pc\_type** `<none,ahess,bfgs,petsc>`, respectively. The default is the **bfgs** preconditioner. When the preconditioner type is set to **petsc**, the preconditioner set with the PETSc **-pc\_type** command line argument is used. For example, to use an incomplete Cholesky factorization for the preconditioner, one would use the command line arguments **-tao\_nls\_pc\_type petsc -pc\_type icc**. See the PETSc manual for further information on changing the behavior of the preconditioners.

The choice of scaling matrix can have a significant impact on the quality of the Hessian approximation when using the **bfgs** preconditioner and affect the number of iterations required by the linear system solver. The choices for scaling matrices are the same as those discussed for the limited-memory, variable-metric algorithm. For Newton methods, however, the option exists to use a scaling matrix based on the true Hessian matrix. In particular, the implementation supports using the absolute value of the diagonal of the Hessian matrix or the absolute value of the diagonal of the perturbed Hessian matrix. The scaling matrix to use with the **bfgs** preconditioner is set with the command line argument `-tao_nls_bfgs_scale_type <bfgs,ahess,phess>`; **phess** is the default. The **bfgs** scaling matrix is derived from the BFGS options. The **ahess** scaling matrix is the absolute value of the diagonal of the Hessian matrix. The **phess** scaling matrix is the absolute value of the diagonal of the perturbed Hessian matrix.

The perturbation  $\rho_k$  is added when the direction returned by the Krylov subspace method is either not a descent direction, the Krylov method diverged due to an indefinite preconditioner or matrix, or a direction of negative curvature was found. In the two latter cases, if the step returned is a descent direction, it is used during the line search. Otherwise, a steepest descent direction is used during the line search. The perturbation is decreased as long as the Krylov subspace method reports success and increased if further problems are encountered. There are three cases: initializing, increasing, and decreasing the perturbation. These cases are described below.

1. If  $\rho_k$  is zero and a problem was detected with either the direction on the Krylov subspace method, the perturbation is initialized to

$$\rho_{k+1} = \text{median} \{ \text{imin}, \text{imfac} * \|g(x_k)\|, \text{imax} \},$$

where **imin** is set with the command line argument `-tao_nls_imin <double>` with a default value of  $10^{-4}$ , **imfac** by `-tao_nls_imfac` with a default value of 0.1, and **imax** by `-tao_nls_imax` with a default value of 100. When using the **gltr** method to solve the system of equations, an estimate of the minimum eigenvalue  $\lambda_1$  of the Hessian matrix is available. This value is use to initialize the perturbation to  $\rho_{k+1} = \max \{ \rho_{k+1}, -\lambda_1 \}$ .

2. If  $\rho_k$  is nonzero and a problem was detected with either the direction or Krylov subspace method, the perturbation is increased to

$$\rho_{k+1} = \min \{ \text{pmax}, \max \{ \text{pgfac} * \rho_k, \text{pmgfac} * \|g(x_k)\| \} \},$$

where **pgfac** is set with the command line argument `-tao_nls_pgfac` with a default value of 10, **pmgfac** by `-tao_nls_pmgfac` with a default value of 0.1, and **pmax** by `-tao_nls_pmax` with a default value of 100.

3. If  $\rho_k$  is nonzero and no problems were detected with either the direction or Krylov subspace method, the perturbation is decreased to

$$\rho_{k+1} = \min \{ \text{psfac} * \rho_k, \text{pmsfac} * \|g(x_k)\| \},$$

where **psfac** is set with the command line argument `-tao_nls_psfac` with a default value of 0.4, and **pmsfac** by `-tao_nls_pmsfac` with a default value of 0.1. Moreover,

if  $\rho_{k+1} < \text{pmin}$  then  $\rho_{k+1} = 0$ , where **pmin** is set with the command line argument **-tao\_nls\_pmin** and has a default value of  $10^{-12}$ .

When using **stcg** or **gltr** to solve the linear systems of equation, a trust-region radius need to be initialized and updated. This trust-region radius limits the size of the step computed. The method for initializing the trust-region radius is set with the command line argument **-tao\_nls\_init\_type <constant,direction,interpolation>**; **interpolation**, which chooses an initial value based on the interpolation scheme found in [5], is the default. This scheme performs a number of function and gradient evaluations to determine a radius such that the reduction predicted by the quadratic model along the gradient direction coincides with the actual reduction in the nonlinear function. The iterate obtaining the best objective function value is used as the starting point for the main line-search algorithm. The **constant** method initializes the trust-region radius by using the value specified with the **-tao\_trust0 <double>** command line argument, where the default value is 100. The **direction** technique solves the first quadratic optimization problem by using a standard conjugate gradient method and initializes the trust-region to  $\|s_0\|$ .

Finally, the method for updating the trust-region radius is set with the command line argument **-tao\_nls\_update\_type <step,reduction,interpolation>**; **step** is the default. The **step** method updates the trust-region radius based on the value of  $\tau_k$ . In particular,

$$\Delta_{k+1} = \begin{cases} \omega_1 \min(\Delta_k, \|d_k\|) & \text{if } \tau_k \in [0, \nu_1) \\ \omega_2 \min(\Delta_k, \|d_k\|) & \text{if } \tau_k \in [\nu_1, \nu_2) \\ \omega_3 \Delta_k & \text{if } \tau_k \in [\nu_2, \nu_3) \\ \max(\Delta_k, \omega_4 \|d_k\|) & \text{if } \tau_k \in [\nu_3, \nu_4) \\ \max(\Delta_k, \omega_5 \|d_k\|) & \text{if } \tau_k \in [\nu_4, \infty) \end{cases}$$

where  $0 < \omega_1 < \omega_2 < \omega_3 = 1 < \omega_4 < \omega_5$  and  $0 < \nu_1 < \nu_2 < \nu_3 < \nu_4$  are constants. The **reduction** method computes the ratio of the actual reduction in the objective function to the reduction predicted by the quadratic model for the full step,  $\kappa_k = \frac{f(x_k) - f(x_k + d_k)}{q(x_k) - q(x_k + d_k)}$ , where  $q_k$  is the quadratic model. The radius is then updated as:

$$\Delta_{k+1} = \begin{cases} \alpha_1 \min(\Delta_k, \|d_k\|) & \text{if } \kappa_k \in (-\infty, \eta_1) \\ \alpha_2 \min(\Delta_k, \|d_k\|) & \text{if } \kappa_k \in [\eta_1, \eta_2) \\ \alpha_3 \Delta_k & \text{if } \kappa_k \in [\eta_2, \eta_3) \\ \max(\Delta_k, \alpha_4 \|d_k\|) & \text{if } \kappa_k \in [\eta_3, \eta_4) \\ \max(\Delta_k, \alpha_5 \|d_k\|) & \text{if } \kappa_k \in [\eta_4, \infty) \end{cases}$$

where  $0 < \alpha_1 < \alpha_2 < \alpha_3 = 1 < \alpha_4 < \alpha_5$  and  $0 < \eta_1 < \eta_2 < \eta_3 < \eta_4$  are constants. The **interpolation** method uses the same interpolation mechanism as in the initialization to compute a new value for the trust-region radius.

#### 4.1.5 Newton Trust-Region Method

The Newton trust-region method solves the constrained quadratic programming problem

$$\begin{aligned} \min_d \quad & \frac{1}{2} d^T H_k d + g_k^T d \\ \text{subject to} \quad & \|d\| \leq \Delta_k \end{aligned}$$



to obtain a direction  $d_k$ , where  $H_k$  is the Hessian of the objective function at  $x_k$ ,  $g_k$  is the gradient of the objective function at  $x_k$  and  $\Delta_k$  is the trust-region radius. If  $x_k + d_k$  sufficiently reduces the nonlinear objective function, then the step is accepted and the trust-region radius is updated. However, if  $x_k + d_k$  does not sufficiently reduce the nonlinear objective function, then the step is rejected, the trust-region radius is reduced, and the quadratic program is re-solved using the updated trust-region radius. The Newton trust-region method can be set using TaoMethod `tao_ntr`. For the best efficiency, function and gradient evaluations should be performed separately when using this algorithm.

The quadratic optimization problem is approximately solved by applying the Steihaug-Toint conjugate gradient method or generalized Lanczos method to the symmetric system of equations  $H_k d = -g_k$ . The method used to solve the system of equations is specified with the command line argument `-tao_ntr_ksp_type <stcg,gltr>`; `stcg` is the default. Internally, the PETSc implementations for the Steihaug-Toint method and the generalized Lanczos method are used. See the PETSc manual for further information on changing the behavior of these linear system solvers.

A good preconditioner reduces the number of iterations required to compute the direction. For the Steihaug-Toint conjugate gradient method and generalized Lanczos method, this preconditioner must be symmetric and positive definite. The available options are to use no preconditioner, the absolute value of the diagonal of the Hessian matrix, a limited-memory BFGS approximation to the Hessian matrix, or one of the other preconditioners provided by the PETSc package. These preconditioners are specified by the the command line argument `-tao_ntr_pc_type <none,ahess,bfgs,petsc>`, respectively. The default is the `bfgs` preconditioner. When the preconditioner type is set the to `petsc`, the preconditioner set with the PETSc `-pc_type` command line argument is used. For example, to use an incomplete Cholesky factorization for the preconditioner, one would use the command line arguments `-tao_ntr_pc_type petsc -pc_type icc`. See the PETSc manual for further information on changing the behavior of the preconditioners.

The choice of scaling matrix can have a significant impact on the quality of the Hessian approximation when using the `bfgs` preconditioner and affect the number of iterations required by the linear system solver. The choices for scaling matrices are the same as those discussed for the limited-memory, variable-metric algorithm. For Newton methods, however, the option exists to use a scaling matrix based on the true Hessian matrix. In particular, the implementation supports using the absolute value of the diagonal of the Hessian matrix. The scaling matrix to use with the `bfgs` preconditioner is set with the command line argument `-tao_ntr_bfgs_scale_type <ahess,bfgs>`; `ahess` is the default. The `bfgs` scaling matrix is derived from the BFGS options. The `ahess` scaling matrix is the absolute value of the diagonal of the Hessian matrix.

The method for computing an initial trust-region radius is set with the command line argument `-tao_ntr_init_type <constant,direction,interpolation>`; `interpolation`, which chooses an initial value based on the interpolation scheme found in [5], is the default. This scheme performs a number of function and gradient evaluations to determine a radius such that the reduction predicted by the quadratic model along the gradient direction coincides with the actual reduction in the nonlinear function. The iterate obtaining the best objective function value is used as the starting point for the main line-search algorithm. The `constant` method initializes the trust-region radius by using the value specified with

the `-tao_trust0 <double>` command line argument, where the default value is 100. The **direction** technique solves the first quadratic optimization problem by using a standard conjugate gradient method and initializes the trust-region to  $\|s_0\|$ .

Finally, the method for updating the trust-region radius is set with the command line argument `-tao_ntr_update_type <reduction,interpolation>`; **reduction** is the default. The **reduction** method computes the ratio of the actual reduction in the objective function to the reduction predicted by the quadratic model for the full step,  $\kappa_k = \frac{f(x_k) - f(x_k + d_k)}{q(x_k) - q(x_k + d_k)}$ , where  $q_k$  is the quadratic model. The radius is then updated as:

$$\Delta_{k+1} = \begin{cases} \alpha_1 \min(\Delta_k, \|d_k\|) & \text{if } \kappa_k \in (-\infty, \eta_1) \\ \alpha_2 \min(\Delta_k, \|d_k\|) & \text{if } \kappa_k \in [\eta_1, \eta_2) \\ \alpha_3 \Delta_k & \text{if } \kappa_k \in [\eta_2, \eta_3) \\ \max(\Delta_k, \alpha_4 \|d_k\|) & \text{if } \kappa_k \in [\eta_3, \eta_4) \\ \max(\Delta_k, \alpha_5 \|d_k\|) & \text{if } \kappa_k \in [\eta_4, \infty) \end{cases}$$

where  $0 < \alpha_1 < \alpha_2 < \alpha_3 = 1 < \alpha_4 < \alpha_5$  and  $0 < \eta_1 < \eta_2 < \eta_3 < \eta_4$  are constants. The **interpolation** method uses the same interpolation mechanism as in the initialization to compute a new value for the trust-region radius.

## 4.2 Bound Constrained Optimization

Bound constrained optimization algorithms minimize  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , subject to upper or lower bounds on some of the variables. These solvers also bounds on the variables as well as objective function, gradient, and possibly Hessian information.

### 4.2.1 Newton Trust Region

The TRON [18] algorithm is an active set method that uses a combination of gradient projections and a preconditioned conjugate gradient method to minimize an objective function. Each iteration of the TRON algorithm requires function, gradient, and Hessian evaluations. In each iteration, the algorithm first applies several conjugate gradients. After these iterates, the TRON solver momentarily ignores the variables that equal one of its bounds and applies a preconditioned conjugate gradient method to a quadratic model of the free variables.

The TRON algorithm solves a reduced linear system defined by the rows and columns corresponding to the variables that lie between the upper and lower bounds. When running in parallel, these rows can either remain on their current processor or be redistributed evenly over all of the processors with the command `TaoSelectSubset()`. The TRON algorithm applies a trust region to the conjugate gradients to ensure convergence. The initial trust region can be set using the command `TaoSetTrustRegionRadius()` and the current trust region size can be found using the command `TaoGetTrustRegionRadius()`. The initial trust region can significantly alter the rate of convergence for the algorithm and should be tuned and adjusted for optimal performance.

### 4.2.2 Gradient Projection–Conjugate Gradient Method

The GPCG [20] algorithm is much like the TRON algorithm, discussed in Section 4.2.1, except that it assumes that the objective function is quadratic and convex. Therefore, it evaluates the function, gradient, and Hessian only once. Since the objective function is quadratic, the algorithm does not use a trust region. All of the options that apply to TRON, except for trust region options, also apply to GPCG.

### 4.2.3 Interior Point Newton Algorithm

The BQPIP algorithm is an interior point algorithm for bound constrained quadratic optimization. It can be set using the TaoMethod of `tao_bqpip`. Since it assumes the objective function is quadratic, it evaluates the function, gradient, and Hessian only once. In this algorithm all of the variables are free variables. This method also requires the solution of systems of linear equations, whose solver can be accessed and modified with the command `TaoGetLinearSolver()`.

### 4.2.4 Limited Memory Variable Metric Method

This method is the bound constrained variant of the LMVM method for unconstrained optimization. It uses projected gradients to approximate the Hessian – eliminating the need for Hessian evaluations. The method can be set using TaoMethod `tao_blvm`. The command `TaoLMVMSetSize()`, which sets the number of vectors to be used in the Hessian approximation, also applies to this method.

### 4.2.5 KT Method

This method calculates points satisfying the first-order necessary optimality conditions. The method uses the mixed complementarity problem solvers from Section 4.3 to calculate the solutions. The choice of complementarity solver is specified with the runtime option `-tao_kt_method` with the default being the `tao_ssils` method.

## 4.3 Complementarity

Mixed complementarity problems, or box-constrained variational inequalities, are related to nonlinear systems of equations. They are defined by a continuously differentiable function,  $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , and bounds,  $\ell \in \{\mathbb{R} \cup \{-\infty\}\}^n$  and  $u \in \{\mathbb{R} \cup \{\infty\}\}^n$ , on the variables such that  $\ell \leq u$ . Given this information,  $x^* \in [\ell, u]$  is a solution to  $\text{MCP}(F, \ell, u)$  if for each  $i \in \{1, \dots, n\}$  we have at least one of the following:

$$\begin{aligned} F_i(x^*) &\geq 0 && \text{if } x_i^* = \ell_i \\ F_i(x^*) &= 0 && \text{if } \ell_i < x_i^* < u_i \\ F_i(x^*) &\leq 0 && \text{if } x_i^* = u_i. \end{aligned}$$

Note that when  $\ell = \{-\infty\}^n$  and  $u = \{\infty\}^n$  we have a nonlinear system of equations, and  $\ell = \{0\}^n$  and  $u = \{\infty\}^n$  corresponds to the nonlinear complementarity problem [6].

Simple complementarity conditions arise from the first-order optimality conditions from optimization [16, 17]. In the simple bound constrained optimization case, these conditions correspond to MCP( $\nabla f$ ,  $\ell$ ,  $u$ ), where  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is the objective function. In a one-dimensional setting these conditions are intuitive. If the solution is at the lower bound, then the function must be increasing and  $\nabla f \geq 0$ . However, if the solution is at the upper bound, then the function must be decreasing and  $\nabla f \leq 0$ . Finally, if the solution is strictly between the bounds, we must be at a stationary point and  $\nabla f = 0$ . Other complementarity problems arise in economics and engineering [9], game theory [23], and finance [14].

Evaluation routines for  $F$  and its Jacobian must be supplied prior to solving the application. The bounds,  $[\ell, u]$ , on the variables must also be provided. If no starting point is supplied, a default starting point of all zeros is used.

#### 4.3.1 Semismooth Methods

TAO has two implementations of semismooth algorithms [22, 7, 8] for solving mixed complementarity problems. Both are based upon a reformulation of the mixed complementarity problem as a nonsmooth system of equations using the Fischer-Burmeister function [10]. A nonsmooth Newton method is applied to the reformulated system to calculate a solution. The theoretical properties of such methods are detailed in the aforementioned references.

The Fischer-Burmeister function,  $\phi : \mathbb{R}^2 \rightarrow \mathbb{R}$ , is defined as,

$$\phi(a, b) := \sqrt{a^2 + b^2} - a - b.$$

This function has the following key property

$$\phi(a, b) = 0 \iff a \geq 0, b \geq 0, ab = 0$$

used when reformulating the mixed complementarity problem the system of equations  $\Phi(x) = 0$  where  $\Phi : \mathbb{R}^n \rightarrow \mathbb{R}^n$ . The reformulation is defined component-wise as

$$\Phi_i(x) := \begin{cases} \phi(x_i - l_i, F_i(x)) & \text{if } -\infty < l_i < u_i = \infty, \\ -\phi(u_i - x_i, -F_i(x)) & \text{if } -\infty = l_i < u_i < \infty, \\ \phi(x_i - l_i, \phi(u_i - x_i, -F_i(x))) & \text{if } -\infty < l_i < u_i < \infty, \\ -F_i(x) & \text{if } -\infty = l_i < u_i = \infty, \\ l_i - x_i & \text{if } -\infty < l_i = u_i < \infty. \end{cases}$$

We note that  $\Phi$  is not differentiable everywhere, but satisfies a semismoothness property [19, 26, 27]. Furthermore, the natural merit function,  $\Psi(x) := \frac{1}{2} \|\Phi(x)\|_2^2$ , is continuously differentiable.

The two semismooth TAO solvers both solve the system  $\Phi(x) = 0$  by applying a non-smooth newton method with a line-search. We calculate a direction,  $d^k$ , by solving the system  $H^k d^k = -\Phi(x^k)$  where  $H^k$  is an element of the  $B$ -subdifferential [27] of  $\Phi$  at  $x^k$ . If the direction calculated does not satisfy a suitable descent condition, then we use the negative gradient of the merit function,  $-\nabla \Psi(x^k)$ , as the search direction. A standard Armijo search [1] is used to find the new iteration. Non-monotone searches [11] are also available by setting appropriate run-time options. See Section 6.2 for further details.

The first semismooth algorithm available in TAO is not guaranteed to remain feasible with respect to the bounds,  $[\ell, u]$ , and is termed an infeasible semismooth method. This method can be specified using the TaoMethod `tao_ssils`. In this case, the descent test used is that

$$\nabla \Psi(x^k)^T d^k \leq -\delta \|d^k\|^\rho.$$

Both  $\delta > 0$  and  $\rho > 2$  can be modified using the run-time commands `-tao_ssils_delta <delta>` and `-tao_ssils_rho <rho>` respectively. By default,  $\delta = 10^{-10}$  and  $\rho = 2.1$ .

An alternative is to remain feasible with respect to the bounds by using a projected Armijo line-search. This method can be specified using the TaoMethod `tao_ssfls`. The descent test used is the same as above where the direction in this case corresponds to the first part of the piece-wise linear arc searched by the projected line-search. Both  $\delta > 0$  and  $\rho > 2$  can be modified using the run-time commands `-tao_ssfls_delta <delta>` and `-tao_ssfls_rho <rho>` respectively. By default,  $\delta = 10^{-10}$  and  $\rho = 2.1$ .

The recommended algorithm is the infeasible semismooth method, `tao_ssils`, because of its strong global and local convergence properties. However, if it is known that  $F$  is not defined outside of the box,  $[\ell, u]$ , perhaps due to the presence of *log* functions, the feasible algorithm, `tao_ssfls`, is a reasonable alternative.

Table 4.2: Summary of `nls` options

Name	Value	Default	Description
<code>-tao_nls_ksp_type</code>	cg, stcg, gltr, petsc	cg	Type of Krylov subspace method to use when solving linear system
<code>-tao_nls_pc_type</code>	none, ahess, bfgs, petsc	bfgs	Type of preconditioner to use when solving linear system
<code>-tao_nls_bfgs_scale_type</code>	ahess, phess, bfgs	phess	Type of scaling matrix to use with BFGS preconditioner
<code>-tao_nls_sval</code>	double	0	Initial perturbation value
<code>-tao_nls_imin</code>	double	$10^{-4}$	Minimum initial perturbation value
<code>-tao_nls_imax</code>	double	100	Maximum initial perturbation value
<code>-tao_nls_imfac</code>	double	0.1	Factor applied to norm of gradient when initializing perturbation
<code>-tao_nls_pmax</code>	double	100	Maximum perturbation when increasing value
<code>-tao_nls_pgfac</code>	double	10	Growth factor applied to perturbation when increasing value
<code>-tao_nls_pmgfac</code>	double	0.1	Factor applied to norm of gradient when increasing perturbation
<code>-tao_nls_pmin</code>	double	$10^{-12}$	Minimum perturbation when decreasing value; smaller values set to zero
<code>-tao_nls_psfac</code>	double	0.4	Shrink factor applied to perturbation when decreasing value
<code>-tao_nls_pmsfac</code>	double	0.1	Factor applied to norm of gradient when decreasing perturbation

Table 4.3: Summary of `nls` options (continued)

Name	Value	Default	Description
<code>-tao_nls_init_type</code>	constant, direction, interpolation	interpolation	Method used to initialize trust-region radius when using <code>stcg</code> or <code>gltr</code>
<code>-tao_nls_mu1_i</code>	double	0.35	$\mu_1$ in interpolation init
<code>-tao_nls_mu2_i</code>	double	0.50	$\mu_2$ in interpolation init
<code>-tao_nls_gamma1_i</code>	double	0.0625	$\gamma_1$ in interpolation init
<code>-tao_nls_gamma2_i</code>	double	0.50	$\gamma_2$ in interpolation init
<code>-tao_nls_gamma3_i</code>	double	2.00	$\gamma_3$ in interpolation init
<code>-tao_nls_gamma4_i</code>	double	5.00	$\gamma_4$ in interpolation init
<code>-tao_nls_theta_i</code>	double	0.25	$\theta$ in interpolation init
<code>-tao_nls_update_type</code>	step, reduction, interpolation	step	Method used to update trust-region radius when using <code>stcg</code> or <code>gltr</code>
<code>-tao_nls_nu1</code>	double	0.25	$\nu_1$ in step update
<code>-tao_nls_nu2</code>	double	0.50	$\nu_2$ in step update
<code>-tao_nls_nu3</code>	double	1.00	$\nu_3$ in step update
<code>-tao_nls_nu4</code>	double	1.25	$\nu_4$ in step update
<code>-tao_nls_omega1</code>	double	0.25	$\omega_1$ in step update
<code>-tao_nls_omega2</code>	double	0.50	$\omega_2$ in step update
<code>-tao_nls_omega3</code>	double	1.00	$\omega_3$ in step update
<code>-tao_nls_omega4</code>	double	2.00	$\omega_4$ in step update
<code>-tao_nls_omega5</code>	double	4.00	$\omega_5$ in step update
<code>-tao_nls_eta1</code>	double	$10^{-4}$	$\eta_1$ in reduction update
<code>-tao_nls_eta2</code>	double	0.25	$\eta_2$ in reduction update
<code>-tao_nls_eta3</code>	double	0.50	$\eta_3$ in reduction update
<code>-tao_nls_eta4</code>	double	0.90	$\eta_4$ in reduction update
<code>-tao_nls_alpha1</code>	double	0.25	$\alpha_1$ in reduction update
<code>-tao_nls_alpha2</code>	double	0.50	$\alpha_2$ in reduction update
<code>-tao_nls_alpha3</code>	double	1.00	$\alpha_3$ in reduction update
<code>-tao_nls_alpha4</code>	double	2.00	$\alpha_4$ in reduction update
<code>-tao_nls_alpha5</code>	double	4.00	$\alpha_5$ in reduction update
<code>-tao_nls_mu1</code>	double	0.10	$\mu_1$ in interpolation update
<code>-tao_nls_mu2</code>	double	0.50	$\mu_2$ in interpolation update
<code>-tao_nls_gamma1</code>	double	0.25	$\gamma_1$ in interpolation update
<code>-tao_nls_gamma2</code>	double	0.50	$\gamma_2$ in interpolation update
<code>-tao_nls_gamma3</code>	double	2.00	$\gamma_3$ in interpolation update
<code>-tao_nls_gamma4</code>	double	4.00	$\gamma_4$ in interpolation update
<code>-tao_nls_theta</code>	double	0.05	$\theta$ in interpolation update

Table 4.4: Summary of `ntr` options

Name	Value	Default	Description
<code>-tao_ntr_ksp_type</code>	stcg, gltr	stcg	Type of Krylov subspace method to use when solving linear system
<code>-tao_ntr_pc_type</code>	none, ahess, bfgs, petsc	bfgs	Type of preconditioner to use when solving linear system
<code>-tao_ntr_bfgs_scale_type</code>	ahess, bfgs	ahess	Type of scaling matrix to use with BFGS preconditioner
<code>-tao_ntr_init_type</code>	constant, direction, interpolation	interpolation	Method used to initialize trust-region radius
<code>-tao_ntr_mu1_i</code>	double	0.35	$\mu_1$ in interpolation init
<code>-tao_ntr_mu2_i</code>	double	0.50	$\mu_2$ in interpolation init
<code>-tao_ntr_gamma1_i</code>	double	0.0625	$\gamma_1$ in interpolation init
<code>-tao_ntr_gamma2_i</code>	double	0.50	$\gamma_2$ in interpolation init
<code>-tao_ntr_gamma3_i</code>	double	2.00	$\gamma_3$ in interpolation init
<code>-tao_ntr_gamma4_i</code>	double	5.00	$\gamma_4$ in interpolation init
<code>-tao_ntr_theta_i</code>	double	0.25	$\theta$ in interpolation init
<code>-tao_ntr_update_type</code>	reduction, interpolation	reduction	Method used to update trust-region radius
<code>-tao_ntr_eta1</code>	double	$10^{-4}$	$\eta_1$ in reduction update
<code>-tao_ntr_eta2</code>	double	0.25	$\eta_2$ in reduction update
<code>-tao_ntr_eta3</code>	double	0.50	$\eta_3$ in reduction update
<code>-tao_ntr_eta4</code>	double	0.90	$\eta_4$ in reduction update
<code>-tao_ntr_alpha1</code>	double	0.25	$\alpha_1$ in reduction update
<code>-tao_ntr_alpha2</code>	double	0.50	$\alpha_2$ in reduction update
<code>-tao_ntr_alpha3</code>	double	1.00	$\alpha_3$ in reduction update
<code>-tao_ntr_alpha4</code>	double	2.00	$\alpha_4$ in reduction update
<code>-tao_ntr_alpha5</code>	double	4.00	$\alpha_5$ in reduction update
<code>-tao_ntr_mu1</code>	double	0.10	$\mu_1$ in interpolation update
<code>-tao_ntr_mu2</code>	double	0.50	$\mu_2$ in interpolation update
<code>-tao_ntr_gamma1</code>	double	0.25	$\gamma_1$ in interpolation update
<code>-tao_ntr_gamma2</code>	double	0.50	$\gamma_2$ in interpolation update
<code>-tao_ntr_gamma3</code>	double	2.00	$\gamma_3$ in interpolation update
<code>-tao_ntr_gamma4</code>	double	4.00	$\gamma_4$ in interpolation update
<code>-tao_ntr_theta</code>	double	0.05	$\theta$ in interpolation update



## Chapter 5

# TAO Applications using PETSc

The solvers in TAO address applications that have a set of variables, an objective function, and constraints on the variables. Many solvers also require derivatives of the objective and constraint functions. To use the TAO solvers, the application developer must define a set of variables, implement routines that evaluate the objective function and constraint functions, and pass this information to a TAO application object.

TAO uses vector and matrix objects to pass this information from the application to the solver. The set of variables, for instance, is represented in a vector. The gradient of an objective function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , evaluated at a point, is also represented as a vector. Matrices, on the other hand, can be used to represent the Hessian of  $f$  or the Jacobian of a constraint function  $c : \mathbb{R}^n \rightarrow \mathbb{R}^m$ . The TAO solvers use these objects to compute a solution to the application.

The PETSc package provides parallel and serial implementations of these objects and offers additional tools intended for high-performance scientific applications. The **Vec** and **Mat** types in PETSc represent the vectors and matrices in a TAO application. This chapter will describe how to create these an application object and give it the necessary properties. This chapter will also describe how to use the TAO solvers in conjunction with this application object.

### 5.1 Header File

TAO applications written in C/C++ should have the statement

```
#include "tao.h"
```

in each file that uses a routine in the TAO libraries. All of the required lower level include files such as “tao\_solver.h” and “taoapp.h” are automatically included within this high-level file.

### 5.2 Create and Destroy

To create an application object, first declare a `TAO_APPLICATION` variable. This variable is only a pointer. The application object associated with it can be created using the routine

```
TaoApplicationCreate(MPI_Comm, TAO_APPLICATION*);
```

Much like creating PETSc vector and matrix objects, the first argument is an MPI *communicator*. An MPI [13] communicator indicates a collection of processors that will be used to evaluate the objective function, compute constraints, and provide derivative information. When only one processor is being used, the communicator `MPI_COMM_SELF` can be used with no understanding of MPI. Even parallel users need to be familiar with only the basic concepts of message passing and distributed-memory computing. Most applications running TAO in parallel environments can employ the communicator `MPI_COMM_WORLD` to indicate all processes in a given run.

The second argument is the address of a `TAO_APPLICATION` variable. This routine will create a new application object and set the variable, which is a pointer, to the address of the object. This application variable can now be used by the developer to define the application and by the TAO solver to solve the application.

Elsewhere in this chapter, the `TAO_APPLICATION` variable will be referred to as the *application object*.

After solving the application, the command

```
TaoAppDestroy(TAO_APPLICATION);
```

will destroy the application object and free the work space associated with it.

## 5.3 Defining Variables

In all of the optimization solvers, the application must provide a **Vec** object of appropriate dimension to represent the variables. This vector will be cloned by the solvers to create additional work space within the solver. If this vector is distributed over multiple processors, it should have a parallel distribution that allows for efficient scaling, inner products, and function evaluations. This vector can be passed to the application object using the routine

```
TaoAppSetInitialSolutionVec(TAO_APPLICATION,Vec);
```

When using this routine, the application should initialize the vector with an approximate solution of the optimization problem before calling the TAO solver. If you do not know of a solution that can be used, the routine `TaoAppSetDefaultSolutionVec(TAO_APPLICATION,Vec);` can be used to declare variables that will in be set to zero or some other default solution.

This vector will be used by the TAO solver to store the solution. Elsewhere in the application, this solution vector can be retrieved from the application object using the routine

```
TaoAppGetSolutionVec(TAO_APPLICATION, Vec *);
```

This routine takes the address of a **Vec** in the second argument and sets it to the solution vector used in the application.

## 5.4 Application Context

Writing an application using the `TAO_APPLICATION` object may require use of an *application context*. An application context is a structure or object defined by an application developer, passed into a routine also written by the application developer, and used within the routine to perform its stated task.

For example, a routine that evaluates an objective function may need parameters, work vectors, and other information. This information, which may be specific to an application and necessary to evaluate the objective, can be collected in a single structure and used as one of the arguments in the routine. The address of this structure will be cast as type `(void*)` and passed to the routine in the final argument. There are many examples of these structures in the TAO distribution.

This technique offers several advantages. In particular, it allows for a uniform interface between TAO and the applications. The fundamental information needed by TAO appears in the arguments of the routine, while data specific to an application and its implementation is confined to an opaque pointer. The routines can access information created outside the local scope without the use of global variables. The TAO solvers and application objects will never access this structure, so the application developer has complete freedom to define it. In fact, these contexts are completely optional – a NULL pointer can be used.

## 5.5 Objective Function and Gradient Routines

TAO solvers that minimize an objective function require the application to evaluate the objective function. Some solvers may also require the application to evaluate derivatives of the objective function. Routines that perform these computations must be identified to the application object and must follow a strict calling sequence.

Routines that evaluate an objective function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , should follow the form:

```
EvaluateObjective(TAO_APPLICATION,Vec,double*,void*);
```

The first argument is the application object, the second argument is the  $n$ -dimensional vector that identifies where the objective should be evaluated, and the fourth argument is an application context. This routine should use the third argument to return objective value, evaluated at the given point specified by the vector in the second argument.

This routine, and the application context, should be passed to the application object using the routine

```
TaoAppSetObjectiveRoutine(TAO_APPLICATION,
                          int(*) (TAO_APPLICATION,Vec,double*,void*),
                          void*);
```

The first argument in this routine is the application object, the second argument is a function pointer to the routine that evaluates the objective, and the third argument is the pointer an appropriate application context.

Although final argument may point to anything, it must be cast as a `(void*)` type. This pointer will be passed back to the developer in the fourth argument of the routine that evaluates the objective. In this routine, the pointer can be cast back to the appropriate type. Examples of these structures and their usage are provided in the distribution.

Most TAO solvers also require gradient information from the application. The gradient of the objective function can be specified in a similar manner. Routines that evaluate the gradient should have the calling sequence

```
EvaluateTheGradient(TAO_APPLICATION,Vec,Vec,void*);
```

In this routine, the first argument is the application object, the second argument is the variable vector, the third argument is the gradient, and the fourth argument is the user-defined application context. Only the third argument in this routine is different from the arguments in the routine that evaluates the objective function. The numbers in the gradient vector have no meaning when passed into this routine, but should represent the gradient of the objective at the specified point at the end of the routine. This routine, and the user-defined pointer, can be passed to the application object using the routine:

```
TaoAppSetGradientRoutine(TAO_APPLICATION,
                        int (*)(TAO_APPLICATION,Vec,Vec,void*),
                        void *);
```

In this routine, the first argument is the application object, the second argument is the function pointer, and the third object is the application context, cast to (void\*).

Instead of evaluating the objective and its gradient in separate routines, TAO also allows the user to evaluate the function and the gradient at the same routine. In fact, some solvers are more efficient when both function and gradient information can be computed in the same routine. These routines should follow the form

```
EvaluateFunctionGradient(TAO_APPLICATION,Vec,double*,Vec,void*);
```

where the first argument is the TAO solver, and the second argument points to the input vector for use in evaluating the function and gradient. The third argument should return the function value, while the fourth argument should return the gradient vector, and the fifth argument is a pointer to a user-defined context. This context and the name of the routine should be set with the call:

```
TaoAppSetObjectiveAndGradientRoutine(TAO_APPLICATION,
                                     int (*)(TAO_APPLICATION,Vec,double*,Vec,void*),
                                     void *);
```

The arguments of this routine are the TAO application, a function name, and a pointer to a user-defined context.

The TAO example problems demonstrate the use of these application contexts as well as specific instances of function, gradient, and Hessian evaluation routines. All of these routines should return the integer 0 after successful completion and a nonzero integer if the function is undefined at that point or an error occurred.

## 5.6 Hessian Evaluation

Some optimization routines also require a Hessian matrix from the user. The routine that evaluates the Hessian should have the form:

```
EvaluateTheHessian(TAO_APPLICATION,Vec,Mat*,Mat*,MatStructure*,void*);
```

The first argument of this routine is a TAO application. The second argument is the point at which the Hessian should be evaluated. The third argument is the Hessian matrix, and the sixth argument is a user-defined context. Since the Hessian matrix is usually used in solving a system of linear equations, a preconditioner for the matrix is often needed. The fourth argument is the matrix that will be used for preconditioning the linear system. In most cases, this matrix will be the same as the Hessian matrix. The fifth argument is the flag used to set the Hessian matrix and linear solver in the routine `KSPSetOperators()`.

One can set the Hessian evaluation routine by calling

```

int TaoAppSetHessianRoutine(TAO_APPLICATION,
                           int (*)(TAO_APPLICATION,Vec,Mat*,Mat*,MatStructure*,void*),
                           void *)

```

The first argument is the TAO application, the second argument is the function that evaluates the Hessian, and the third argument is a pointer to a user defined context, cast as a `void*` pointer.

For solvers that evaluate the Hessian, the matrices used to store the Hessian should be set using

```
TaoAppSetHessianMat(TAO_APPLICATION,Mat,Mat);
```

The first argument is the TAO application, the second argument is the Hessian matrix, and the third argument is the preconditioning matrix. In most applications, these two matrices will be the same structure.

### 5.6.1 Finite Differences

Finite differences approximations can be used to compute the gradient and the Hessian of an objective function. These approximations will slow down the solve considerably and are only recommended for checking the accuracy of hand-coded gradients and Hessians. These routines are

```
TaoAppDefaultComputeGradient(TAO_APPLICATION, Vec, Vec, void*);
```

```
TaoAppDefaultComputeHessian( TAO_APPLICATION, Vec, Mat*, Mat*,
                             MatStructure*, void*);
```

and

```
TaoAppDefaultComputeHessianColor( TAO_APPLICATION, Vec, Mat*, Mat*,
                                  MatStructure*, void* );
```

These routines can be set using `TaoAppSetGradientRoutine()` and `TaoAppSetHessianRoutine()` or through the options database. If finite differencing is used with coloring, the routine

```
TaoAppSetColoring(TAO_APPLICATION, ISColoring);
```

should be used to specify the coloring.

It is also possible to use finite difference approximations to directly check the correctness of an application's gradient and/or Hessian evaluation routines. This can be done using the special TAO solver `tao_fd_test` together with the options `-tao_test_gradient` or `-tao_test_hessian`.

### 5.6.2 Matrix-Free methods

TAO fully supports matrix-free methods. The matrices specified in the Hessian evaluation routine need not be conventional matrices; instead, they can point to the data required to implement a particular matrix-free method. The matrix-free variant is allowed *only* when the linear systems are solved by an iterative method in combination with no preconditioning (PCNONE or `-pc_type none`), a user-provided preconditioner matrix, or a user-provided preconditioner shell (PCSHELL); that is, obviously matrix-free methods cannot be used if a direct solver is to be employed. Details about using matrix-free methods are provided in the PETSc Users Manual.

## 5.7 Bounds on Variables

Some optimization problems also impose constraints upon the variables. The constraints may impose simple bounds upon the variables, or require that the variables satisfy a set of linear or nonlinear equations.

The simplest type of constraint upon an optimization problem puts lower or upper bounds upon the variables. Vectors that represent lower and upper bounds for each variable can be set with the command

```
TaoAppSetVariableBoundsRoutine(TAO_APPLICATION,
                                int (*)(TAO_APPLICATION, Vec,Vec,void*),void *);
```

The first vector and second vectors should contain the lower and upper bounds, respectively. When no upper or lower bound exists for a variable, the bound may be set to `TAO_INFINITY` or `TAO_NINFINITY`. After the two bound vectors have been set, they may be accessed with the with the command `TaoGetApplicationVariableBounds()`. Since not all solvers use bounds on variables, the user must be careful to select a type of solver that acknowledges these bounds.

## 5.8 Complementarity

Constraints in the form of nonlinear equations have the form  $C(X) = 0$  where  $C : \mathbb{R}^n \rightarrow \mathbb{R}^m$ . These constraints should be specified in a routine, written by the user, that evaluates  $C(X)$ . The routine that evaluates the constraint equations should have the form:

```
int EqualityConstraints(TAO_APPLICATION,Vec,Vec,void*);
```

The first argument of this routine is a TAO application object. The second argument is the variable vector at which the constraint function should be evaluated. The third argument is the vector of function values  $C$ , and the fourth argument is a pointer to a user-defined context. This routine and the user-defined context should be set in the TAO solver with the command

```
TaoAppSetConstraintRoutine(TAO_APPLICATION,
                           int (*)(TAO_APPLICATION,Vec,Vec,void*),
                           void*);
```

In this function, first argument is the TAO application, the second argument is vector in which to store the function values, and the third argument is a pointer to a user-defined context that will be passed back to the user.

The Jacobian of the function  $C$  is the matrix in  $\mathbb{R}^{m \times n}$  such that each column contains the partial derivatives of  $f$  with respect to one variable. The evaluation of the Jacobian of  $f$  should be performed in a routine of the form

```
int J(TAO_APPLICATION,Vec,Mat*,Mat*,MatStructure*,void*);
```

In this function, the second argument is the variable vector at which to evaluate the Jacobian matrix, the third argument is the Jacobian matrix, and the sixth argument is a pointer to a user-defined context. This routine should be specified using

```
TaoAppSetJacobianRoutine(TAO_APPLICATION,Mat,
                         int (*)(TAO_APPLICATION,Vec,Mat*,Mat*, MatStructure*,void*),
                         void*);
```

The first argument is the TAO application, the second argument is the matrix in which the information can be stored, the third argument is the function pointer, and the fourth argument is an optional user-defined context. The Jacobian matrix should be created in a way such that the product of it and the variable vector can be put in the constraint vector.

For solvers that evaluate the Jacobian, the matrices used to store the Jacobian should be set using

```
TaoAppSetJacobianMat(TAO_APPLICATION,Mat,Mat);
```

The first argument is the TAO application, the second argument is the Jacobian matrix, and the third argument is the preconditioning matrix. In most applications, these two matrices will be the same structure.

## 5.9 Monitors

By default the TAO solvers run silently without displaying information about the iterations. The user can initiate monitoring with the command

```
int TaoSetMonitor(TAO_SOLVER solver,
                  int (*mon)(TAO_SOLVER tao,void* mctx),
                  void *mctx);
```

The routine, `mon` indicates a user-defined monitoring routine and `mctx` denotes an optional user-defined context for private data for the monitor routine.

The routine set by `TaoAppSetMonitor()` is called once during each iteration of the optimization solver. Hence, the user can employ this routine for any application-specific computations that should be done after the solution update. .

```
TaoAppSetMonitor(TAO_APPLICATION,
                  int (*)(TAO_APPLICATION,void*),void *);
```

## 5.10 Linear Solvers

One of the most computationally intensive phases of many optimization algorithms involves the solution of systems of linear equations. The performance of the linear solver may be critical to an efficient computation of the solution. Since linear equation solvers often have a wide variety of options associated with them, TAO allows the user to access the linear solver with the command

```
TaoAppGetKSP(TAO_APPLICATION, KSP *);
```

With access to the KSP object, users can customize it for their application to achieve additional performance.

## 5.11 Application Solutions

Once the application object has the objective function, constraints, derivatives, and other features associated with it, a TAO solver can be applied to the application. For further information about how to create a TAO solver, see the previous chapter.

Once the TAO solver and TAO application object have been created and customized, they can be matched with one another using the routine

```
TaoSetupApplicationSolver( TAO_APPLICATION, TAO_SOLVER);
```

This routine will set up the TAO solver for the application. Different solvers may set up differently, but they typically create the work vectors and linear solvers needed to find a solution. These structures were not created during the creation of the solver because the size of the application was not known. After calling this routine the routine `TaoAppGetTaoSolver()` can be used to obtain the TAO solver object. If not called directly by the application, `TaoSetupApplicationSolver()` will be executed inside of the subroutine `TaoSolveApplication()`.

The routine

```
TaoGetGradientVec( TAO_SOLVER, Vec*);
```

will set a pointer to a `Vec` to the vector object containing the gradient vector and the routine

```
TaoGetVariableBoundVecs( TAO_SOLVER, Vec*, Vec*);
```

will set the pointers to the lower and upper bounds on the variables – if they exist. These vectors may be viewed at before, during, and after the solver is running.

Options for the application and solver can be set from the command line using the routine

```
TaoSetOptions( TAO_APPLICATION, TAO_SOLVER);
```

This routine will call `TaoSetupApplicationSolver()` if it has not been called already. This command also provides information about runtime options when the user includes the `-help` option on the command line.

Once the application and solver have been set up, the solver can be called using the routine

```
TaoSolveApplication( TAO_APPLICATION, TAO_SOLVER);
```

This routine will call the TAO solver. If the routine `TaoSetupApplicationSolver()` has not already been called, this routine will call it.

After a solution has been found, the routine

```
TaoCopyDualsOfVariableBounds( TAO_APPLICATION, Vec, Vec );
```

can compute the dual values of the variables bounds and copy them into the vectors passed into this routine.

## 5.12 Linear Algebra Abstractions

Occasionally TAO users will have to interact directly with the linear algebra objects used by the solvers. Solvers within TAO use vector, matrix, index set, and linear solver objects that have no native data structures. Instead they have methods whose implementation is uses structures and routines provided by PETSc or other external software packages.

Given a PETSc `Vec` object `X`, the user can create a `TaoVec` object. By declaring the variables

```
TaoVec *xx;
```

the routine

```
TaoWrapPetscVec(Vec,TaoVec **);
```



takes the `Vec x` and creates and sets `TaoVec *xx` equal to a new `TaoVec` object. This object actually has the derived type `TaoVecPetsc`. Given a `TaoVec` whose underlying representation is a PETSc `Vec`, the command

```
TaoVecGetPetscVec( TaoVec *, Vec *);
```

will retrieve the underlying vector. The routine `TaoVecDestroy()` will destroy the `TaoVec` object, but the `Vec` object must also be destroyed.

The routine

```
TaoWrapPetscMat(Mat, TaoMat **);
```

takes the `Mat H` and creates and sets `TaoMat *HH` equal to the new `TaoMat` object. The second argument specifies whether the `Mat` object should be destroyed when the `TaoVec` object is destroyed. This object actually has the derived type `TaoMatPetsc`. Given a `TaoMat` whose underlying representation is a PETSc `Vec`, the command

```
TaoMatGetPetscMat( TaoMat *, Mat *);
```

will retrieve the underlying matrix. The routine `TaoMatDestroy()` will destroy the `TaoMat` object, but the `Mat` object must also be destroyed.

Similarly, the routine

```
TaoWrapKSP( KSP, TaoLinearSolver **);
```

takes a `KSP` object and creates a `TaoLinearSolver` object. The

```
TaoLinearSolverGetKSP( TaoLinearSolver *, KSP *);
```

gets the underlying `KSP` object from the `TaoLinearSolver` object.

For index sets, the routine

```
TaoWrapPetscIS( IS, int, TaoIndexSet **);
```

creates a `TaoIndexSet` object. In this routine, however, the second argument is the local size of the vectors that this object will describe. For instance, this object may describe with elements of a vector are positive. The second argument should be local length of the vector. The `IS` object will be destroyed when the `TaoIndexSet` is destroyed. The routine

```
TaoIndexSetGetPetscIS( TaoIndexSet *, IS *);
```

will return the underlying `IS` object.

## 5.13 Compiling and Linking

Portable TAO makefiles follow the rules and definitions of PETSc makefiles. In Figures 5.1 we present a sample makefile.

This small makefile is suitable for maintaining a single program that uses the TAO library. The most important line in this makefile is the line starting with `include`:

```
include ${TAO_DIR}/bmake/tao_common
```

This line includes other makefiles that provide the needed definitions and rules for the particular base software installations (specified by `${TAO_DIR}` and `${PETSC_DIR}`) and architecture (specified by `${PETSC_ARCH}`), which are typically set as environmental variables prior to compiling TAO source or programs. As listed in the sample makefile, the appropriate `include` file is automatically completely specified; the user should *not* alter this statement within the makefile.

TAO applications using PETSc should be linked with the to the `PETSC_SNES_LIB` library as well as the `TAO_LIB` library. This version uses PETSc 3.1, and the `PETSC_DIR` variable should be set accordingly. Many examples of makefiles can be found in the `examples` directories.

```

CFLAGS      =
FFLAGS      =
CPPFLAGS    =
FPPFLAGS    =

include ${TAO_DIR}/bmake/tao_common

minsurf1: minsurf1.o tao_chkopts
    -${CLINKER} -o minsurf1 minsurf1.o ${TAO_LIB} ${PETSC_SNES_LIB}
    ${RM} minsurf1.o

```

Figure 5.1: Sample TAO makefile for a single C program

## 5.14 TAO Applications using PETSc and FORTRAN

Most of the functionality of TAO can be obtained by people who program purely in Fortran 77 or Fortran 90. Note, however, that we recommend the use of C and/or C++ because these languages contain several extremely powerful concepts that the Fortran77/90 family does not. The TAO Fortran interface works with both F77 and F90 compilers.

Since Fortran77 does not provide type checking of routine input/output parameters, we find that many errors encountered within TAO Fortran programs result from accidentally using incorrect calling sequences. Such mistakes are immediately detected during compilation when using C/C++. Thus, using a mixture of C/C++ and Fortran often works well for programmers who wish to employ Fortran for the core numerical routines within their applications. In particular, one can effectively write TAO driver routines in C++, thereby preserving flexibility within the program, and still use Fortran when desired for underlying numerical computations.

Only a few differences exist between the C and Fortran TAO interfaces, all of which are due to differences in Fortran syntax. All Fortran routines have the same names as the corresponding C versions, and command line options are fully supported. The routine arguments follow the usual Fortran conventions; the user need not worry about passing pointers or values. The calling sequences for the Fortran version are in most cases identical to the C version, except for the error checking variable discussed in Section 5.14.2. In addition, the Fortran routine `TaoInitialize(char *filename,int info)` differs slightly from its C counterpart; see the manual page for details.

### 5.14.1 Include Files

Currently, TAO users must employ the Fortran file suffix `.F` rather than `.f`. This convention enables use of the CPP preprocessor, which allows the use of the `#include` statements that define TAO objects and variables. (Familiarity with the CPP preprocessor is not needed for writing TAO Fortran code; one can simply begin by copying a TAO Fortran example and its corresponding makefile.)

The TAO directory `${TAO_DIR}/include/finclude` contains the Fortran include files and should be used via statements such as the following:

```
#include "include/finclude/includefile.h"
```

Since one must be very careful to include each file no more than once in a Fortran routine, application programmers must manually include each file needed for the various TAO (or other supplementary) components within their program. This approach differs from the TAO C++ interface, where the user need only include the highest level file, for example, `tao.h`, which then automatically includes all of the required lower level files. As shown in the various Fortran example programs in the TAO distribution, in Fortran one must explicitly list *each* of the include files.

### 5.14.2 Error Checking

In the Fortran version, each TAO routine has as its final argument an integer error variable, in contrast to the C++ convention of providing the error variable as the routine's return value. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. For example, the Fortran and C++ variants of `TaoSolveApplication()` are given, respectively, below, where `info` denotes the error variable:

```
call TaoSolveApplication(TAO_APPLICATION taoapp, TAO_SOLVER tao, int info)
info = TaoSolveApplication(TAO_APPLICATION taoapp, TAO_SOLVER tao)
```

Fortran programmers can use the error codes in writing their own tracebacks. For example, one could use code such as the following:

```
call TaoSolveApplication(taoapp, tao, info)
if (info .ne. 0) then
    print*, 'Error in routine ...'
    return
endif
```

In addition, Fortran programmers can check these error codes with the macro `CHKERRQ()`, which terminates all process when an error is encountered. See the PETSc users manual for details. The most common reason for crashing PETSc Fortran code is forgetting the final `info` argument.

Additional interface differences for Fortran users:

- `TaoGetConvergenceHistory()` – returns only the number of elements in the history. Storage for the convergence information must be preallocated by the user and then registered with `TaoSetConvergenceHistory()`.
- `TaoSetLinesearch()` – use only the first and fourth arguments. The setup, options, view, and destroy routines do not apply.

### 5.14.3 Compiling and Linking Fortran Programs

Figure 5.2 shows a sample makefile that can be used for TAO Fortran programs. You can compile a debugging version of the program `rosenbrock1f` with `make rosenbrock1f`.

Note that the TAO Fortran interface library, given by `$(TAO_FORTRAN_LIB)`, *must* precede the base TAO library, given by `$(TAO_LIB)`, on the link line.

```

CFLAGS      =
FFLAGS      =
CPPFLAGS    =
FPPFLAGS    =

include ${TAO_DIR}/bmake/tao_common

rosenbrock1f: rosenbrock1f.o  tao_chkopts
               -${FLINKER} -o rosenbrock1f rosenbrock1f.o ${TAO_FORTRAN_LIB} ${TAO_LIB} \
               ${PETSC_FORTRAN_LIB} ${PETSC_SNES_LIB}
               ${RM} rosenbrock1f.o

```

Figure 5.2: Sample TAO makefile for a single Fortran program

#### 5.14.4 Additional Issues

The TAO library currently interfaces to the PETSc library for low-level system functionality as well as linear algebra support. The PETSc users manual discusses additional Fortran issues in these areas, including

- array arguments (e.g., `VecGetArray()`),
- calling Fortran Routines from C (and C Routines from Fortran),
- passing null pointers,
- duplicating multiple vectors, and
- matrix and vector indices.

## Chapter 6

# Advanced Options

This section discusses options and routines that apply to all TAO solvers and problem classes. In particular, we focus on convergence tests and line searches.

### 6.1 Convergence Tests

There are many different ways to define convergence of a solver. The methods TAO uses by default are mentioned in Section 3.3. These methods include absolute and relative convergence tolerances as well as a maximum number of iterations of function evaluations. If these choices are not sufficient, the user can even specify a customized test.

Users can set their own customized convergence tests of the form

```
int conv(TAO_SOLVER tao, void *cctx);
```

The second argument is a pointer to a structure defined by the user. Within this routine, the solver can be queried for the solution vector, gradient vector, or other statistic at the current iteration through routines such as `TaoGetSolutionStatus()` and `TaoGetTolerances()`.

To use this convergence test within a TAO solver, use the command

```
int TaoSetConvergenceTest(TAO_SOLVER solver,
                          int (*conv)(TAO_SOLVER tao,
                                      void *cctx),
                          void *cctx);
```

The second argument of this command is the convergence routine, and the final argument of the convergence test routine, `cctx`, denotes an optional user-defined context for private data. The convergence routine receives the TAO solver and this private data structure. The termination flag can be set using the routine

```
int TaoSetTerminationReason(TAO_SOLVER , TaoTerminationReason*);
```

### 6.2 Line Searches

Many solver in TAO require a line search. While these solver always offer a default line search, alternative line searches can also be used. Line searches must have the form:

```
int L(TAO_SOLVER tao, TaoVec *xx, TaoVec *gg, TaoVec *dx, TaoVec *ww,
      double *f, double *step, double *gdx, int *flg, void *lsctx);
```

In this routine the first argument is the TAO solver, the second argument is the current solution vector, the third argument is the gradient at the current point, the fourth argument is the step direction, the fourth vector is a work vector, the fifth argument is the function value, the sixth argument is the step length, the seventh argument is the inner product of the gradient and direction vector used for the Armijo condition, the eighth argument is a flag indicating success or failure of the line search, and the last argument is a pointer to a structure that can be used to define the line search. When the routine is finished the solution vector **xx**, gradient vector **gg**, function value **f**, step size **step**, and **flg** should be updated to reflect the new solution.

This routine can be set with the function

```
int TaoSetLineSearch(TAO_SOLVER solver,
                    int (*setup)(TAO_SOLVER, void*),
int (*options)(TAO_SOLVER,void*),
                    int (*line)(TAO_SOLVER,TaoVec*,TaoVec*,TaoVec*,TaoVec*,
                                double*,double*,double*,int*,void*),
                    int (*viewit)(TAO_SOLVER,void*),
int (*destroy)(TAO_SOLVER,void*),
                    void *ctx);
```

In this routine, the fourth argument is the function pointer to the line search routine, and the seventh argument is the pointer that will be passed to the line search routine. The other arguments are optional function pointers than can be used to set up, view, and deallocate the solver.

## Chapter 7

# Adding a solver

### 7.1 Adding a Solver to TAO

New optimization solvers can be added to TAO. TAO provides tools for facilitate the implementation of a solver. The advantages of implementing a solver using TAO are several.

1. TAO includes other optimization solvers with an identical interface, so application problems may conveniently switch solvers to compare their effectiveness.
2. TAO provides support for function evaluations and derivative information. It allows for the direct evaluation of this information by the application developer, and contains limited support for finite difference, and allows the uses of matrix-free methods. The solvers can obtain this function and derivative information through a simple interface while the details of its computation are handled within the toolkit.
3. TAO provides line searches, convergence tests, monitoring routines, and other tools which are helpful within an optimization algorithm. The availability of these tools means that the developers of the optimization solver do not have to write these utilities.
4. TAO offers vectors, matrices, index sets, and linear solvers that can be used by the solver. These objects are standard mathematical constructions that have many different implementations. The objects may be distributed over multiple processors, restricted to a single processor, have a dense representation, use a sparse data structure, or vary in many other ways. TAO solvers do not need to know how these objects are represented or how the operations defined on them have been implemented. Instead, the solvers apply these operations through an abstract interface that leaves the details to TAO and external libraries. This abstraction allows solvers to work seamlessly with a variety of data structures while allowing application developers to select data structures tailored for their purposes.
5. TAO supports an interface to PETSc and allows the integration of other libraries as well. When used with PETSc, TAO provides the user a convenient method for setting options at runtime, performance profiling, and debugging.

## 7.2 TAO Interface with Solvers

TAO solvers must be written in C++ and include several routines with a particular calling sequence. Two of these routines are mandatory: one that initializes the TAO structure with the appropriate information and one that applies the algorithm to a problem instance. Additional routines may be written to set some options within the solver, view the solver, setup appropriate data structures, and destroy these data structures. In each of these routines except the initialization routine, there are two arguments.

The first argument is always the TAO structure. This structure may be used to obtain the vectors used to store the variables and the function gradient, evaluate a function and gradient, solve a set of linear equations, perform a line search, and apply a convergence test.

The second argument is specific to this solver. This pointer will be set in the initialization routine and cast to an appropriate type in the other routines. To implement the Fletcher - Reeves conjugate gradient algorithm, for instance, the following structure may be useful.

```
typedef struct{

    double beta;

    TaoVec *gg;
    TaoVec *dx;    /* step direction */
    TaoVec *ww;    /* work vector    */

} TAO_CG;
```

This structure contains two work vectors and a scalar. Vectors for the solution and gradient are not needed here because the TAO structure has pointers to them.

### 7.2.1 Solver Routine

All TAO solvers have a routine that accepts a TAO structure and computes a solution. TAO will call this routine when the application program uses the routine `TaoSolve()` and pass to the solver information about the objective function and constraints, pointers to the variable vector and gradient vector, and support for line searches, linear solvers, and convergence monitoring. As an example, consider the following code which solves an unconstrained minimization problem using the Fletcher-Reeves conjugate gradient method.

```
static int TaoSolve_CG_FR(TAO_SOLVER tao, void *solver){

    TAO_CG *cg = (TAO_CG *) solver;
    TaoVec *xx,*gg=cg->gg;    /* solution vector, gradient vector */
    TaoVec *dx=cg->dx, *ww=cg->ww;
    int    iter=0,lsflag=0,info;
    double gnormPrev,gdx,f,gnorm,step=0;
    TaoTerminateReason reason;

    TaoFunctionBegin;
    info=TaoCheckFG(tao);CHKERRQ(info);
```



```

info=TaoGetSolution(tao,&xx);CHKERRQ(info);

info = TaoComputeMeritFunctionGradient(tao,xx,&f,gg);CHKERRQ(info);
info = gg->Norm2(&gnorm);  CHKERRQ(info);

info = dx->SetToZero(); CHKERRQ(info);

cg->beta=0;
gnormPrev = gnorm;

/* Enter loop */
while (1){

    /* Test for convergence */
    info = TaoMonitor(tao,iter++,f,gnorm,0.0,step,&reason);CHKERRQ(info);
    if (reason!=TAO_CONTINUE_ITERATING) break;

    cg->beta=(gnorm*gnorm)/(gnormPrev*gnormPrev);
    info = dx->Axpby(-1.0,gg,cg->beta); CHKERRQ(info);

    info = dx->Dot(gg,&gdx); CHKERRQ(info);
    if (gdx>=0){          /* If not a descent direction, use gradient */
        cg->beta=0.0;
        info = dx->Axpby(-1.0,gg,cg->beta); CHKERRQ(info);
        gdx=-gnorm*gnorm;
    }

    /* Line Search */
    gnormPrev = gnorm;  step=1.0;
    info = TaoLineSearchApply(tao,xx,gg,dx,ww,&f,&step,&lsflag);
    info = gg->Norm2(&gnorm);CHKERRQ(info);

}

TaoFunctionReturn(0);
}

```

The first line of this routine cast the second argument to a pointer to a TAO\_CG data structure. This structure contains pointers to three vectors and a scalar which will be needed in the algorithm.

After declaring and initializing several variables, the solver first checks that the function and gradient have been defined using the routine `TaoCheckFG()`. Next, the solver gets the variable vector which was passed to TAO by the application program. Other solvers may also want to get pointers to Hessian matrices, Jacobian matrices, or vectors containing bounds on the variables. The commands for these routines are `TaoGetSolution()`, `TaoGetVariableBounds()`, `TaoGetHessian()`, and `TaoGetJacobian()`.

This solver lets TAO evaluate the function and gradient at the current point in the using the routine `TaoComputeFunctionGradient()`. Other routines may be used to evaluate the Hessian matrix or evaluate constraints. TAO may obtain this information using direct evaluation of other means, but the these details do not affect our implementation of the algorithm.

The norm of the gradient is a standard measure used by unconstrained minimization solvers to define convergence. This quantity is always nonnegative and equals zero at the solution. The solver will pass this quantity, the current function value, the current iteration number, and a measure of infeasibility to TAO with the routine

```
int TaoMonitor(TAO_SOLVER,int,double,double,double,double,
               TaoTerminateReason*);
```

Most optimization algorithms are iterative in nature, and solvers should include this command somewhere in each iteration. This routine records this information, applies any monitoring routines and convergence tests set by default or the user.

In this routine, the second argument is the current iteration number, and the third argument is the current function value. The fourth argument is a nonnegative error measure associated with the distance between the current solution and the optimal solution. Examples of this measure are the norm of the gradient or the square root of a duality gap. The fifth measure is a nonnegative error that is nonnegative and usually represents a residual between the current function value and the optimal solution, such as the norm of the gradient. The sixth argument is a nonnegative steplength, or the multiple of the step direction added to the previous iterate. The results of the convergence test are returned in the last argument. If the termination reason is `TAO_CONTINUE_ITERATING`, the algorithm should continue.

After this monitoring routine, the solver computes a step direction using methods defined on the `TaoVec` object. These methods include adding vectors together and computing an inner product. A full list of these methods can be found in the manual pages.

Nonlinear conjugate gradient algorithms also require a line search. TAO provides several line searches and support for using them. The routine

```
int TaoLineSearchApply(TAO_SOLVER tao, TaoVec *xx, TaoVec *gg, TaoVec *dx,
                       TaoVec *ww, double *f, double *step,
                       int*flag)
```

passes the current solution, gradient, and objective value to the solver and returns a new solution, gradient, and objective value. More details on line searches can be found in the Section 6.2 The details of this line search are should be specified elsewhere, when the line search is created.

TAO also includes support for linear solvers. Although this algorithm does not require one, linear solvers are an important part of many algorithms. Details on the use of these solvers can be found in Section 5.10.

## 7.2.2 Creation Routine

The TAO solver is initialized to for a particular algorithm in a separate routine. This routine sets default convergence tolerances, creates a line search or linear solver if needed, and creates structures needed by this solver. For example, the routine that creates the nonlinear conjugate gradient algorithm shown above can be implemented as follows.

```

EXTERN_C_BEGIN
int TaoCreate_CG_FR(TAO_SOLVER tao)
{
    TAO_CG *cg;
    int    info;

    TaoFunctionBegin;

    info = TaoNew(TAO_CG,&cg); CHKERRQ(info);

    info = TaoSetMaximumIterates(tao,2000); CHKERRQ(info);
    info = TaoSetTolerances(tao,1e-4,1e-4,0,0); CHKERRQ(info);
    info = TaoSetMaximumFunctionEvaluations(tao,4000); CHKERRQ(info);

    info = TaoCreateMoreThuenteLineSearch(tao,0,0.1); CHKERRQ(info);

    info = TaoSetTaoSolveRoutine(tao,TaoSolve_CG_FR,(void*)cg); CHKERRQ(info);
    info = TaoSetTaoSetUpDownRoutines(tao,TaoSetUp_CG,TaoDestroy_CG); CHKERRQ(info);
    info = TaoSetTaoOptionsRoutine(tao,TaoSetOptions_CG_FR); CHKERRQ(info);
    info = TaoSetTaoViewRoutine(tao,TaoView_CG); CHKERRQ(info);

    TaoFunctionReturn(0);
}
EXTERN_C_END

```

The first thing this routine does after declaring some variables, is allocate memory for the TAO\_CG data structure. Clones of the the variable vector assed into TAO in the TaoCreate() routine are used as the two work vectors. This routine also sets some default convergence tolerances and creates a particular line search. These defaults could be specified in the routine that solves the problem, but specifying them here gives the user the opportunity to modify these parameters.

Finally, this solvers passes to TAO the names of all the other routines used by the solver.

Note that the lines EXTERN\_C\_BEGIN and EXTERN\_C\_END surround this routine. These macros are required to preserve the name of this function without any name-mangling from the C++ compiler.

### 7.2.3 Destroy Routine

Another routine needed by most solvers destroys the data structures creates by earlier routines. For the nonlinear conjugate gradient method discussed earlier, the following routine destroys the two work vectors, the line search, and the TAO\_CG structure.

```

int TaoDestroy_CG(TAO_SOLVER tao, void *solver)
{
    TAO_CG *cg = (TAO_CG *) solver;
    int    info;

```

```

TaoFunctionBegin;

info = TaoVecDestroy(cg->gg); CHKERRQ(info);
info = TaoVecDestroy(cg->ww);CHKERRQ(info);
info = TaoVecDestroy(cg->dx);CHKERRQ(info);

info = TaoLineSearchDestroy(tao);CHKERRQ(info);
TaoFree(cg);

TaoFunctionReturn(0);
}

```

Other algorithms may destroy matrices, linear solvers, index sets, or other objects needed by the solver. This routine is called from within the `TaoDestroy()` routine.

### 7.2.4 SetUp Routine

If this routine has been set by the initialization routine, TAO will call it during the `TaoSetApplication()`. This routine is optional, but is often used to allocate the gradient vector, work vectors, and other data structures required by the solver. It should have the form

```

int TaoSetUp_CG(TAO_SOLVER,void*);
{
    int    info;
    TaoVec *xx;
    TaoFunctionBegin;

    info = TaoGetSolution(tao,&xx);CHKERRQ(info);
    info = xx->Clone(&cg->gg); CHKERRQ(info);
    info = xx->Clone(&cg->ww); CHKERRQ(info);
    info = xx->Clone(&cg->dx); CHKERRQ(info);
    TaoFunctionReturn(0);
}

```

The second argument can be cast to the appropriate data structure. Many solvers use a similar routine to allocate data structures needed by the solver but not created by the initialization routine.

# Index

application, 35  
application context, 36  
application object, 36  
  
bounds, 28, 40  
  
command line arguments, 6, 15  
convergence tests, 16, 47  
  
finite differences, 39  
  
gradients, 20, 23, 37  
  
Hessian, 38  
  
line search, 20, 23, 24, 47  
linear solver, 43  
  
matrix, 10, 43  
matrix-free options, 39  
  
Newton method, 24, 26  
Newton's method, 28, 29  
  
options, 42  
  
TAO\_DIR, 5  
TAO\_FORTRAN\_LIB, 13, 45  
TAO\_LIB, 12, 13, 43, 45  
TaoAppDefaultComputeGradient(), 39  
TaoAppDefaultComputeHessian(), 39  
TaoAppDefaultComputeHessianColor(), 39  
TaoAppGetKSP(), 41  
TaoAppGetSolutionVec(), 36  
TaoApplicationCreate(), 8, 35  
TaoApplicationDestroy(), 8, 36  
TaoAppSetColoring(), 39  
TaoAppSetConstraintsRoutine(), 40  
TaoAppSetDefaultSolutionVec(), 36  
TaoAppSetGradientRoutine(), 38  
TaoAppSetHessianMat(), 39  
TaoAppSetHessianRoutine(), 38  
TaoAppSetInitialSolutionVec(), 36  
TaoAppSetJacobianMat(), 41  
TaoAppSetJacobianRoutine(), 40  
TaoAppSetMonitor(), 41  
TaoAppSetObjectiveAndGradientRoutine(), 38  
TaoAppSetObjectiveRoutine(), 37  
TaoCreate(), 8, 15  
TaoDestroy(), 8, 16  
TaoFinalize(), 8, 15  
TaoGetGradient(), 17  
TaoGetGradientVec(), 42  
TaoGetSolution(), 17  
TaoGetSolutionStatus(), 17  
TaoGetVariableBoundVecs(), 42  
TaoInitialize(), 6, 15  
TaoSetConvergenceTest(), 47  
TaoSetGradientTolerances(), 16  
TaoSetHessianMat(), 8  
TaoSetHessianRoutine(), 8  
TaoSetInitialSolutionVec(), 8  
TaoSetLineSearch(), 48  
TaoSetMaximumFunctionEvaluations(), 17  
TaoSetMaximumIterates(), 17  
TaoSetMethod(), 16  
TaoSetMonitor(), 41  
TaoSetObjectiveAndGradientRoutine(), 8  
TaoSetOptions(), 42  
TaoSetTolerances(), 16  
TaoSetTrustRegionTolerance, 17  
TaoSetupApplicationSolver(), 41  
TaoSetVariableBoundsRoutine, 40  
TaoSolveApplication(), 8, 42  
trust region, 17, 26, 28  
  
vectors, 42



# Bibliography

- [1] L. Armijo. Minimization of functions having Lipschitz-continuous first partial derivatives. *Pacific Journal of Mathematics*, 16:1–3, 1966.
- [2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc Web page. See <http://www.mcs.anl.gov/petsc>.
- [3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.
- [4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.1.0, Argonne National Laboratory, Apr 2001.
- [5] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. *Trust-Region Methods*. SIAM, Philadelphia, Pennsylvania, 2000.
- [6] R. W. Cottle. *Nonlinear programs with positively bounded Jacobians*. PhD thesis, Department of Mathematics, University of California, Berkeley, California, 1964.
- [7] T. De Luca, F. Facchinei, and C. Kanzow. A semismooth equation approach to the solution of nonlinear complementarity problems. *Mathematical Programming*, 75:407–439, 1996.
- [8] F. Facchinei, A. Fischer, and C. Kanzow. A semismooth Newton method for variational inequalities: The case of box constraints. In M. C. Ferris and J. S. Pang, editors, *Complementarity and Variational Problems: State of the Art*, pages 76–90, Philadelphia, Pennsylvania, 1997. SIAM Publications.
- [9] M. C. Ferris and J. S. Pang. Engineering and economic applications of complementarity problems. *SIAM Review*, 39:669–713, 1997.
- [10] A. Fischer. A special Newton-type optimization method. *Optimization*, 24:269–284, 1992.
- [11] L. Grippo, F. Lampariello, and S. Lucidi. A nonmonotone line search technique for Newton’s method. *SIAM Journal on Numerical Analysis*, 23:707–716, 1986.

- [12] William Gropp and Ewing Lusk. MPICH Web page. <http://www.mcs.anl.gov/mpi/mpich>.
- [13] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.
- [14] J. Huang and J. S. Pang. Option pricing and linear complementarity. *Journal of Computational Finance*, 2:31–60, 1998.
- [15] Mark T. Jones and Paul E. Plassmann. BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Laboratory, December 1995.
- [16] W. Karush. Minima of functions of several variables with inequalities as side conditions. Master’s thesis, Department of Mathematics, University of Chicago, 1939.
- [17] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In J. Neyman, editor, *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. University of California Press, Berkeley and Los Angeles, 1951.
- [18] C.-J. Lin and J. J. Moré. Newton’s method for large bound-constrained optimization problems. *SIOPT*, 9(4):1100–1127, 1999.
- [19] R. Mifflin. Semismooth and semiconvex functions in constrained optimization. *SIAM Journal on Control and Optimization*, 15:957–972, 1977.
- [20] Jorge J. Moré and G. Toraldo. On the solution of large quadratic programming problems with bound constraints. *SIOPT*, 1:93–113, 1991.
- [21] MPI: A message-passing interface standard. *International J. Supercomputing Applications*, 8(3/4), 1994.
- [22] T. S. Munson, F. Facchinei, M. C. Ferris, A. Fischer, and C. Kanzow. The semismooth algorithm for large scale complementarity problems. *INFORMS Journal on Computing*, forthcoming, 2001.
- [23] J. F. Nash. Equilibrium points in N-person games. *Proceedings of the National Academy of Sciences*, 36:48–49, 1950.
- [24] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.
- [25] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.
- [26] L. Qi. Convergence analysis of some algorithms for solving nonsmooth equations. *Mathematics of Operations Research*, 18:227–244, 1993.
- [27] L. Qi and J. Sun. A nonsmooth version of Newton’s method. *Mathematical Programming*, 58:353–368, 1993.