ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

TAO 3.10 Users Manual

Alp Dener
Adam Denchfield
Todd Munson
Jason Sarich
Stefan Wild
Steven Benson
Lois Curfman McInnes

Mathematics and Computer Science Division

Technical Memorandum ANL/MCS-TM-322

This manual is intended for use with TAO version 3.10

September 12, 2018

This product includes software produced by UChicago Argonne, LLC under Contract No. DE-AC02-06CH11357 with the Department of Energy.

Contents

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 unconstrained and bound-constrained optimization, nonlinear least squares problems, optimization problems with partial differential equation constraints, and variational inequalities and complementarity constraints.

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 produce high-quality optimization software for computing environments ranging from 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.

Changes in Version 3.5

TAO is now included in the PETSc distribution and the PETSc repository, thus it versions will always match the PETSc version. The TaoSolver object is now simply Tao and there is no TaoInitialize() or TaoFinalize(). Numerious changes have been made to make the source code more PETSc-like. All future changes will be listed in the PETSc changes documentation.

Changes in Version 2.0

TAO version numbers will now adhere to the new PETSc standard of Major-Minor-Patch. Any patch-level changes will have an attempt to keep the applicatin programming interface (API) untouched, but in any case backward compatibility with previous version of the minor version will be maintained.

Many new features and interface changes were introduced in TAO version 2.0 (and continue to be used in version 2.2.0). TAO applications created for previous versions will need to be updated to work with the new version. We apologize for any inconvenience this situation may cause; these changes were needed to keep the interface clean, clear, and easy to use. Some of the most important changes are highlighted below.

Elimination of the TaoApplication Object. The largest change to the TAO programming interface was the elimination of the TaoApplication data structure. In previous versions of TAO, this structure was created by the application programmer for application-specific data and routines. In order to more closely follow PETSc design principles, this information is now directly attached to a Tao object instead. See Figure 2.1 for a listing of what the most common TAO routines now look like without the TaoApplication object.

New Algorithms. TAO has a new algorithm for solving derivative-free nonlinear least squares problems, POUNDerS, that can efficiently solve parameter optimization problems

when no derivatives are available and function evaluations are expensive. See Section 4.4.1 for more information on the details of the algorithm and Section 4.4 for how to use it. TAO now also provides a new algorithm for the solution of optimization problems with partial differential equation (PDE) constraints based on a linearly constrained augmented Lagrangian (LCL) method. More information on PDE-constrained optimization and LCL can be found in Section 4.3.

TaoLineSearch Object. TAO has promoted the line search to a full object. Any of the available TAO line search algorithms (Armijo, Moré-Thuente, GPCG, and unit) can now be selected regardless of the overlying TAO algorithm. Users can also create new line search algorithms that may be more suitable for their applications. More information is available in Section 5.4.

Better Adherence to PETSc Style. TAO now features a tighter association with PETSc standards and practices. All TAO constructs now follow PETSc conventions and are written in C. There is no longer a separate abstract class for vectors, matrices, and linear solvers. TAO now uses these PETSc objects directly. We believe these changes make TAO applications much easier to create and maintain for users already familiar with PETSc programming. These changes also allow TAO to relax some of the previously imposed requirements on the PETSc configuration. TAO now works with PETSc configured with single-precision and quad-precision arithmetic when using GNU compilers and no longer requires a C++ compiler. However, TAO is not compatible with PETSc installations using complex data types.

Acknowledgments

We especially thank Jorge Moré for his leadership, vision, and effort on previous versions of TAO.

TAO relies on PETSc for the linear algebra required to solve optimization problems, and we have benefited from the PETSc team's experience, tools, software, and advice. In many ways, TAO is a natural outcome of the PETSc development.

TAO has benefited from the work of various researchers who have provided solvers, test problems, and interfaces. In particular, we acknowledge Lisa Grignon, Elizabeth Dolan, Boyana Norris, Gabriel Lopez-Calva, Yurii Zinchenko, Michael Gertz, Jarek Nieplocha, Limin Zhang, Manojkumar Krishnan, and Evan Gawlik. We also thank all TAO users for their comments, bug reports, and encouragement.

The development of TAO is supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. We also thank the Argonne Laboratory Computing Resource Center and the National Energy Research Scientific Computing Center for allowing us to test and run TAO applications on their machines.

Chapter 1

Introduction

The Toolkit for Advanced Optimization (TAO) focuses on the design and implementation of optimization software for solving 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. This chapter provides a short introduction to our design philosophy and the importance of this design.

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, 5] vas 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 fundamental PETSc objects to define and solve optimization problems: vectors, matrices, index sets, 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.

With sufficiently flexible abstract interfaces, PETSc 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 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.

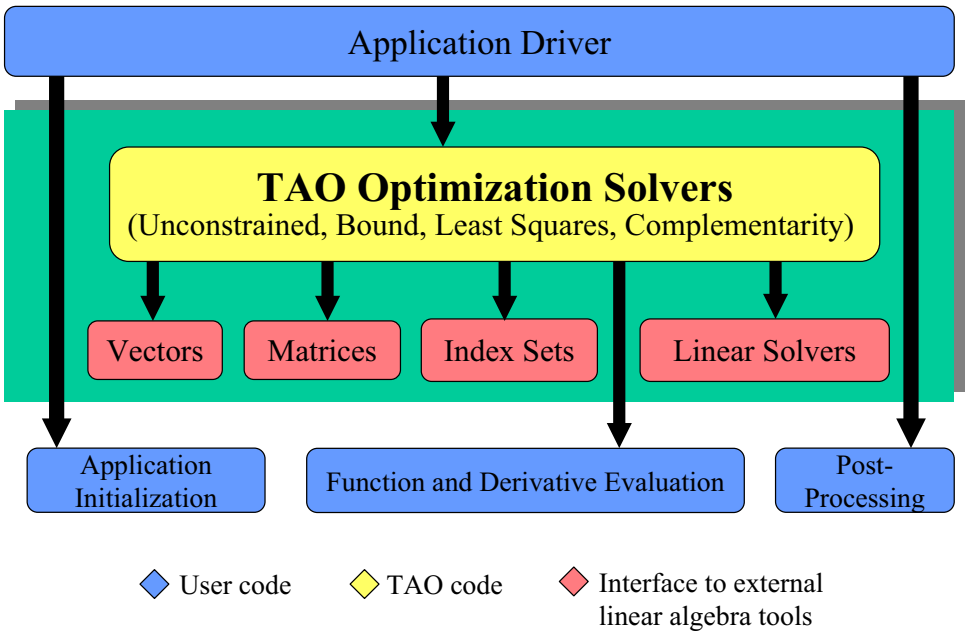

Figure 1.1: TAO Design

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 the examples provided and then gradually learn more details according to their needs. The current version of TAO and the most recent help concerning installation and usage can be found at http://www.mcs.anl.gov/tao.

See the PETSc users manual and http://www.mcs.anl.gov/petsc for how to install and start using PETSc/TAO.

2.1 Writing Application Codes with TAO

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

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

The HTML version of the manual pages located at

```
${PETSC_DIR}/docs/manpages/index.html
```

and

```
http://www.mcs.anl.gov/petsc/documentation/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 PETSc/TAO according to the instructions in http://www.mcs.anl.gov/petsc/documentation/installation.html.
2. Copy an example and makefile from the directories

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

   compile the example, and run the program.
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 introduce here a simple uniprocessor example. Please read Section 3 for a more in-depth discussion on using the TAO solvers. The code presented in Figure 2.2 minimizes the extended Rosenbrock function $f : \mathbb{R}^n \to \mathbb{R}$ defined by

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

where $n = 2m$ is the number of variables. Note that while we use the C language to introduce the TAO software, the package is fully usable from C++ and Fortran77/90. Section **??** discusses additional issues concerning Fortran usage.

The code in Figure 2.2 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 for display purposes we have omitted some nonessential lines of code as well as the (essential) 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 complete code is available in `$TAO_DIR/src/unconstrained/examples/tutorials/rosenbrock1.c`. The following sections annotates the lines of code in Figure 2.2.

2.3 Include Files

The include file for TAO should be used via the statement

```
#include <petsctao.h>
```

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

2.4 TAO Solvers

Many TAO applications will follow an ordered set of procedures for solving an optimization problem: The user creates a `Tao` context and selects a default algorithm. Call-back routines as well as vector (`Vec`) and matrix (`Mat`) data structures are then set. These call-back routines will be used for evaluating the objective function, gradient, and perhaps the Hessian matrix. The user then invokes TAO to solve the optimization problem and finally destroys the `Tao` context. A list of the necessary functions for performing these steps using TAO are shown in Figure 2.1. Details of these commands are presented in Chapter 3.

```
#include "petsctao.h"

typedef struct {
  PetscInt  n;       /* dimension */
  PetscReal alpha;   /* condition parameter */
} AppCtx;

/* -------------- User-defined routines ---------- */
PetscErrorCode FormFunctionGradient(Tao,Vec,PetscReal*,Vec,void*);
PetscErrorCode FormHessian(Tao,Vec,Mat,Mat,void*);

int main(int argc,char **argv)
{
  PetscErrorCode ierr;  /* used to check for functions returning nonzeros */
  Vec            x;         /* solution vector */
  Mat            H;         /* Hessian matrix */
  Tao            tao;       /* Tao context */
  AppCtx         user;      /* user-defined application context */

  ierr = PetscInitialize(&argc,&argv,(char*)0,0);CHKERRQ(ierr);

  /* Initialize problem parameters */
  user.n = 2; user.alpha = 99.0;

  /* Allocate vectors for the solution and gradient */
  ierr = VecCreateSeq(PETSC_COMM_SELF,user.n,&x); CHKERRQ(ierr);
  ierr = MatCreateSeqBAIJ(PETSC_COMM_SELF,2,user.n,user.n,1,NULL,&H);

  /* Create TAO solver with desired solution method */
  ierr = TaoCreate(PETSC_COMM_SELF,&tao); CHKERRQ(ierr);
  ierr = TaoSetType(tao,TAOLMVM); CHKERRQ(ierr);

  /* Set solution vec and an initial guess */
  ierr = VecSet(x, 0); CHKERRQ(ierr);
  ierr = TaoSetInitialVector(tao,x); CHKERRQ(ierr);

  /* Set routines for function, gradient, hessian evaluation */
  ierr = TaoSetObjectiveAndGradientRoutine(tao,FormFunctionGradient,&user);
  ierr = TaoSetHessianRoutine(tao,H,H,FormHessian,&user); CHKERRQ(ierr);

  /* Check for TAO command line options */
  ierr = TaoSetFromOptions(tao); CHKERRQ(ierr);

  /* Solve the application */
  ierr = TaoSolve(tao); CHKERRQ(ierr);

  /* Free data structures */
  ierr = TaoDestroy(&tao); CHKERRQ(ierr);
  ierr = VecDestroy(&x); CHKERRQ(ierr);
  ierr = MatDestroy(&H); CHKERRQ(ierr);
  ierr = PetscFinalize();
  return ierr;
}
```

Figure 2.2: Example of Uniprocessor TAO Code

```
TaoCreate(MPI_Comm comm, Tao *tao);
TaoSetType(Tao tao, TaoType type);
TaoSetInitialVector(Tao tao, Vec x);
TaoSetObjectiveAndGradientRoutine(Tao tao,
     PetscErrorCode (*FormFGradient)(Tao,Vec,PetscReal*,Vec,void*),
     void *user);
TaoSetHessianRoutine(Tao tao, Mat H, Mat Hpre,
     PetscErrorCode (*FormHessian)(Tao,Vec,Mat,Mat,
     void*), void *user);
TaoSolve(Tao tao);
TaoDestroy(Tao tao);
```

Figure 2.1: Commands for Solving an Unconstrained Optimization Problem

Note that the solver algorithm selected through the function `TaoSetType()` can be overridden 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, set iterative methods and preconditions for solving the linear systems, and so forth. See Chapter 3 for more information on the solver methods available in TAO.

2.5 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.

2.6 Programming with PETSc

TAO relies heavily on PETSc not only for its vectors, matrices, and linear solvers but also for its programming utilities such as command line option handling, error handling, and compiling system. We provide here a quick overview of some of these PETSc features. Please refer to the PETSc manual [5] for a more in-depth discussion of PETSc.

Vectors

In the example in Figure 2.2, the vector data structure (`Vec`) is used to store the solution and gradient for the 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 calls either to `VecSetType()` or to `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 [5].

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 routines

```
ierr = MatCreate(MPI_Comm comm,Mat *H);
ierr = MatSetSizes(H,PETSC_DECIDE,PETSC_DECIDE,M,N);
```

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

```
ierr = MatSetValues(Mat H,PetscInt m,PetscInt *im, PetscInt n,
                    PetscInt *in, PetscScalar *values,INSERT_VALUES);
```

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

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

The PETSc users manual [5] discusses various matrix formats as well as the details of some basic matrix manipulation routines.

The Options Database

A TAO application can access the command line options presented at runtime through the PETSc options database. This database gives the application author the ability to set and change application parameters without the need to recompile the application. For example, an application may have a grid discretization parameter `nx` that can be set with the command line option `-nx <integer>`. The application can read this option with the following line of code:

```
PetscOptionsGetInt(NULL,NULL, "-nx", &nx, &flg);
```

If the command line option is present, the variable `nx` is set accordingly; otherwise, `nx` remains unchanged. A complete description of the options database may be found in the PETSc users manual [5].

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(ierr)` checks the value of `ierr` and calls an error handler upon error detection. `CHKERRQ()` should be used after 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 2110 of the file `${PETSC_DIR}/src/mat/interface/matrix.c` in the routine `MatMult()` and was caused by failure to assemble the matrix in the Hessian evaluation routine. The `MatMult()` routine was called from the `TaoSolve_NLS()` routine, which was in turn called on line 154 of `TaoSolve()` from the `main()` routine in the program `rosenbrock1.c`. The PETSc users manual [5] provides further details regarding error checking, including information about error handling in Fortran.

```
> rosenbrock1 -tao_type nls
[0]PETSC ERROR: --------------------- Error Message ------------------------------------
[0]PETSC ERROR: Object is in wrong state!
[0]PETSC ERROR: Not for unassembled matrix!
[0]PETSC ERROR: ------------------------------------------------------------------------
[0]PETSC ERROR: Petsc Development HG revision: b95ffff514b66a703d96e6ae8e78ea266ad2ca19
[0]PETSC ERROR: See docs/changes/index.html for recent updates.
[0]PETSC ERROR: See docs/faq.html for hints about trouble shooting.
[0]PETSC ERROR: See docs/index.html for manual pages.
[0]PETSC ERROR: ------------------------------------------------------------------------
[0]PETSC ERROR: Libraries linked from petsc/arch-linux2-c-debug/lib
[0]PETSC ERROR: Configure run at Tue Jul 19 14:13:14 2011
[0]PETSC ERROR: Configure options --with-shared-libraries --with-dynamic-loading
[0]PETSC ERROR: ------------------------------------------------------------------------
[0]PETSC ERROR: MatMult() line 2110 in petsc/src/mat/interface/matrix.c
[0]PETSC ERROR: TaoSolve_NLS() line 291 in src/unconstrained/impls/nls/nls.c
[0]PETSC ERROR: TaoSolve() line 154 in src/interface/tao.c
[0]PETSC ERROR: main() line 94 in src/unconstrained/examples/tutorials/rosenbrock1.c
application called MPI_Abort(MPI_COMM_WORLD, 73) - process 0
```

Figure 2.3: Example of Error Traceback

When running the debugging version of the TAO software (PETSc configured with the (default) `--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 and, when used together with the command line option `-malloc_debug`, 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.

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. By default, however, 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, for assistance in managing parallel data.

The user must specify a communicator upon creation of any PETSc or TAO object (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 as follows.

```
ierr = MatCreate(MPI_Comm comm,Mat *H);
ierr = VecCreate(MPI_Comm comm,Vec *x);
ierr = TaoCreate(MPI_Comm comm,Tao *tao);
```

In most cases, the value for `comm` will be either `PETSC_COMM_SELF` for single-process objects or `PETSC_COMM_WORLD` for objects distributed over all processors. 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.

Chapter 3

Using TAO Solvers

TAO contains unconstrained minimization, bound-constrained minimization, nonlinear complementarity, nonlinear least squares solvers, and solvers for optimization problems with partial differential equation constraints. The structure of these problems can differ significantly, but TAO has a similar interface to all its solvers. Routines that most solvers have in common are 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 by running a program with the `-help` option.

3.1 Header File

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

```
#include <petsctao.h>
```

in each file that uses a routine in the TAO libraries.

3.2 Creation and Destruction

A TAO solver can be created by calling the

```
TaoCreate(MPI_Comm comm,Tao *newsolver);
```

routine. Much like creating PETSc vector and matrix objects, the first argument is an MPI *communicator*. An MPI [15] 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 `PETSC_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 `PETSC_COMM_WORLD` to indicate all processes known to PETSc in a given run.

The routine

```
TaoSetType(Tao tao,TaoType type);
```

can be used to set the algorithm TAO uses to solve the application. 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 on the problem being solved. Some solvers, for instance, are meant for problems with no constraints, whereas other solvers acknowledge constraints in the problem and handle 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 command line option `-tao_method` (or equivalently `-tao_type`) followed by a 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 `TaoType` variable is a string that requires quotation marks in an application program, but quotation marks are not required at the command line.

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

```
TaoDestroy(Tao tao);
```

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

3.3 TAO Applications

The solvers in TAO address applications that have a set of variables, an objective function, and possibly 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 $f : \mathbb{R}^n \to \mathbb{R}$, 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 $c : \mathbb{R}^n \to \mathbb{R}^m$. The TAO solvers use these objects to compute a solution to the application.

3.3.1 Defining Variables

In all 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 by using the

```
TaoSetInitialVector(Tao,Vec);
```

routine. When using this routine, the application should initialize the vector with an approximate solution of the optimization problem before calling the TAO solver. 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 by using the

```
TaoGetSolutionVector(Tao,Vec *);
```

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

3.3.2 Application Context

Writing a TAO application 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. Many examples of these structures are included 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. If no such structure or needed by the application then a NULL pointer can be used.

3.3.3 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 should follow the form

```
PetscErrorCode EvaluateObjective(Tao,Vec,PetscReal*,void*);
```

in order to evaluate an objective function $f : \mathbb{R}^n \to \mathbb{R}$. The first argument is the TAO Solver 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 the objective value evaluated at the point specified by the vector in the second argument.

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

```
TaoSetObjectiveRoutine(Tao,
                       PetscErrorCode(*)(Tao,Vec,PetscReal*,void*),
                       void*);
```

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

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

```
PetscErrorCode EvaluateGradient(Tao,Vec,Vec,void*);
```

where the first argument is the TAO solver object, the second argument is the variable vector, the third argument is the gradient vector, 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 for evaluating the objective function. The numbers in the gradient vector have no meaning when passed into this routine, but they 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 by using the

```
TaoSetGradientRoutine(Tao,
                      PetscErrorCode (*)(Tao,Vec,Vec,void*),
                      void *);
```

routine. In this routine, the first argument is the Tao 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 in 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

```
PetscErrorCode EvaluateFunctionAndGradient(Tao,Vec,
                      PetscReal*,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. 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

```
TaoSetObjectiveAndGradientRoutine(Tao,
                      PetscErrorCode (*)(Tao,Vec,PetscReal*,Vec,void*),
                      void *);
```

where the arguments 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 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.

3.3.4 Hessian Evaluation

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

```
PetscErrorCode EvaluateHessian(Tao,Vec,Mat,Mat,void*);
```

where the first argument of this routine is a TAO solver object. 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 the

```
TaoSetHessianRoutine(Tao,Mat H, Mat Hpre,
                     PetscErrorCode (*)(Tao,Vec,Mat,Mat,
                     void*), void *);
```

routine. The first argument is the TAO Solver object. The second and third arguments are, respectively, the Mat object where the Hessian will be stored and the Mat object that will be used for the preconditioning (they may be the same). The fourth argument is the function that evaluates the Hessian, and the fifth argument is a pointer to a user-defined context, cast to `(void*)`.

Finite Differences

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

```
TaoDefaultComputeGradient(Tao, Vec, Vec, void*);
```

and

```
TaoDefaultComputeHessian(Tao, Vec, Mat*, Mat*,void*);
```

respectively. They can be set by using `TaoSetGradientRoutine()` and `TaoSetHessianRoutine()` or through the options database with the options `-tao_fdgrad` and `-tao_fd`, respectively.

The efficiency of the finite-difference Hessian can be improved if the coloring of the matrix is known. If the application programmer creates a PETSc `MatFDColoring` object, it can be applied to the finite-difference approximation by setting the Hessian evaluation routine to

```
TaoDefaultComputeHessianColor(Tao, Vec, Mat*, Mat*,void* );
```

and using the `MatFDColoring` object as the last (`void *`) argument to `TaoSetHessianRoutine()`.

One also can use finite-difference approximations to directly check the correctness of the gradient and/or Hessian evaluation routines. This process can be initiated from the command line by using the special TAO solver `tao_fd_test` together with the option `-tao_test_gradient` or `-tao_test_hessian`.

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`). In other words, 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].

3.3.5 Bounds on Variables

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

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

```
TaoSetVariableBounds(Tao,Vec,Vec);
```

command. The first vector and second vector 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 command `TaoGetVariableBounds()`.

Alternatively, it may be more convenient for the user to designate a routine for computing these bounds that the solver will call before starting its algorithm. This routine will have the form

```
PetscErrorCode EvaluateBounds(Tao,Vec,Vec,void*);
```

where the two vectors, representing the lower and upper bounds respectfully, will be computed.

This routine can be set with the

```
TaoSetVariableBoundsRoutine(Tao
                  PetscErrorCode (*)(Tao,Vec,Vec,void*),void*);
```

command.

Since not all solvers recognize the presence of bound constraints on variables, the user must be careful to select a solver that acknowledges these bounds.

3.4 Solving

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

```
TaoSolve(Tao);
```

routine. We discuss several universal options below.

3.4.1 Convergence

Although TAO and its solvers set default parameters that are useful for many problems, the user may need to modify these parameters in order 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, the convergence test employed by TAO attempts to stop when the error in the constraints is less than ϵ_{crtol} and either

$$\begin{array}{lcll} ||g(X)|| & \leq & \epsilon_{gatol}, & \\ ||g(X)||/|f(X)| & \leq & \epsilon_{grtol}, & \text{or} \\ ||g(X)||/|g(X_0)| & \leq & \epsilon_{gttol}, & \end{array}$$

where X is the current approximation to the true solution X^* and X_0 is the initial guess. X^* is unknown, so TAO estimates $f(X) - f(X^*)$ with either the square of the norm of the gradient or the duality gap. A relative tolerance of $\epsilon_{frtol} = 0.01$ indicates that two significant digits are desired in the objective function. Each solver sets its own convergence tolerances, but they can be changed by using the routine `TaoSetTolerances()`. Another set of convergence tolerances terminates 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 `TaoSetMaximumIterations()`. Similarly, a maximum number of function evaluations can be set with the command `TaoSetMaximumFunctionEvaluations()`. `-tao_max_it`, and `-tao_max_funcs`.

3.4.2 Viewing Status

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

```
TaoView(Tao tao)
```

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. This same output can be produced by using the command line option `-tao_view`.

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 follwing command.

```
TaoGetSolutionStatus(Tao tao, PetscInt *iterate, PetscReal *f,
                     PetscReal *gnorm, PetscReal *cnorm, PetscReal *xdiff,
                     TaoConvergedReason *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 `TaoGetConvergedReason()`.

3.4.3 Obtaining a Solution

After exiting the `TaoSolve()` function, the solution, gradient, and dual variables (if available) can be recovered with the following routines.

```
TaoGetSolutionVector(Tao, Vec *X);
TaoGetGradientVector(Tao, Vec *G);
TaoComputeDualVariables(Tao, Vec X, Vec Duals);
```

Note that the `Vec` returned by `TaoGetSolutionVector` will be the same vector passed to `TaoSetInitialVector`. This information can be obtained during user-defined routines such as a function evaluation and customized monitoring routine or after the solver has terminated.

3.4.4 Additional Options

Additional options for the TAO solver can be be set from the command line by using the

```
TaoSetFromOptions(Tao)
```

routine. This command also provides information about runtime options when the user includes the `-help` option on the command line.

3.5 Special Problem Structures

Below we discuss how to exploit the special structures for three classes of problems that TAO solves.

3.5.1 PDE-Constrained Optimization

TAO can solve PDE-constrained optimization applications of the form

$$\begin{aligned} \min_{u,v} \quad & f(u,v) \\ \text{subject to} \quad & g(u,v) = 0, \end{aligned}$$

where the state variable u is the solution to the discretized partial differential equation defined by g and parametrized by the design variable v, and f is an objective function. In this case, the user needs to set routines for computing the objective function and its gradient, the constraints, and the Jacobian of the constraints with respect to the state and design variables. TAO also needs to know which variables in the solution vector correspond to state variables and which to design variables.

The objective and gradient routines are set as for other TAO applications, with `TaoSetObjectiveRoutine()` and `TaoSetGradientRoutine()`. The user can also provide a fused objective function and gradient evaluation with `TaoSetObjectiveAndGradientRoutine()`. The input and output vectors include the combined state and design variables. Index sets for the state and design variables must be passed to TAO by using the function

```
TaoSetStateDesignIS(Tao, IS, IS);
```

where the first IS is a PETSc IndexSet containing the indices of the state variables and the second IS the design variables.

Nonlinear equation constraints have the general form $c(x) = 0$, where $c : \mathbb{R}^n \to \mathbb{R}^m$. 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

```
PetscErrorCode EvaluateConstraints(Tao,Vec,Vec,void*);
```

The first argument of this routine is a TAO solver 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 $c(x)$, 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

```
TaoSetConstraintsRoutine(Tao,Vec,
                  PetscErrorCode (*)(Tao,Vec,Vec,void*),
                  void*);
```

command. In this function, the first argument is the TAO solver object, the second argument a vector in which to store the constraints, the third argument is a function point to the routine for evaluating the constraints, and the fourth argument is a pointer to a user-defined context.

The Jacobian of $c(x)$ is the matrix in $\mathbb{R}^{m \times n}$ such that each column contains the partial derivatives of $c(x)$ with respect to one variable. The evaluation of the Jacobian of c should be performed by calling the

```
PetscErrorCode JacobianState(Tao,Vec,Mat,Mat,Mat,void*);
PetscErrorCode JacobianDesign(Tao,Vec,Mat*,void*);
```

routines. In these functions, The first arguemnt is the TAO solver object. The second argument is the variable vector at which to evaluate the Jacobian matrix, the third argument is the Jacobian matrix, and the last argument is a pointer to a user-defined context. The fourth and fifth arguments of the Jacobian evaluation with respect to the state variables are for providing PETSc matrix objects for the preconditioner and for applying the inverse of the state Jacobian, respectively. This inverse matrix may be `PETSC_NULL`, in which case TAO will use a PETSc Krylov subspace solver to solve the state system. These evaluation routines should be registered with TAO by using the

```
TaoSetJacobianStateRoutine(Tao,Mat,Mat,Mat,
                  PetscErrorCode (*)(Tao,Vec,Mat,Mat,
                  void*), void*);
TaoSetJacobianDesignRoutine(Tao,Mat,
                  PetscErrorCode (*)(Tao,Vec,Mat*,void*),
                  void*);
```

routines. The first argument is the TAO solver object, and the second argument is the matrix in which the Jacobian information can be stored. For the state Jacobian, the third argument is the matrix that will be used for preconditioning, and the fourth argument is

an optional matrix for the inverse of the state Jacobian. One can use `PETSC_NULL` for this inverse argument and let PETSc apply the inverse using a KSP method, but faster results may be obtained by manipulating the structure of the Jacobian and providing an inverse. The fifth argument is the function pointer, and the sixth argument is an optional user-defined context. Since no solve is performed with the design Jacobian, there is no need to provide preconditioner or inverse matrices.

3.5.2 Nonlinear Least Squares

For nonlinear least squares applications, we are solving the optimization problem

$$\min_x \quad \sum_i (f_i(x) - d_i)^2.$$

For these problems, the objective function value should be computed as a vector of residuals, $f_i(x) - d_i$, for better algorithmic results using a separable objective function, computed with a function of the form

```
PetscErrorCode EvaluateSeparableFunction(Tao,Vec,Vec,void*);
```

and set with the

```
TaoSetSeparableObjectiveRoutine(Tao,
                  PetscErrorCode (*)(Tao,Vec,Vec,void*),
                  void *);
```

routine.

3.5.3 Complementarity

Complementarity applications have equality constraints in the form of nonlinear equations $C(X) = 0$, where $C : \mathbb{R}^n \to \mathbb{R}^m$. These constraints should be specified in a routine written by the user with the form

```
PetscErrorCode EqualityConstraints(Tao,Vec,Vec,void*);
```

that evaluates `C(X)`. The first argument of this routine is a TAO Solver object. The second argument is the variable vector X at which the constraint function should be evaluated. The third argument is the output vector of function values $C(X)$, and the fourth argument is a pointer to a user-defined context.

This routine and the user-defined context must be registered with TAO by using the

```
TaoSetConstraintRoutine(Tao, Vec,
                  PetscErrorCode (*)(Tao,Vec,Vec,void*),
                  void*);
```

command. In this command, the first argument is TAO Solver object, the second argument is vector in which to store the function values, the third argument is the user-defined routine that evaluates $C(X)$, and the fourth argument is a pointer to a user-defined context that will be passed back to the user.

The Jacobian of the function 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 C should be performed in a routine of the form

```
PetscErrorCode EvaluateJacobian(Tao,Vec,Mat,Mat,void*);
```

In this function, the first argument is the TAO Solver object and 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. Since the Jacobian matrix may be used in solving a system of linear equations, a preconditioner for the matrix may be 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 Jacobian matrix and linear solver in the routine `KSPSetOperators()`.

This routine should be specified to TAO by using the

```
TaoSetJacobianRoutine(Tao,Mat J, Mat Jpre,
                  PetscErrorCode (*)(Tao,Vec,Mat,Mat,
                  void*), void*);
```

command. The first argument is the TAO Solver object; the second and third arguments are the Mat objects in which the Jacobian will be stored and the Mat object that will be used for the preconditioning (they may be the same), respectively. The fourth argument is the function pointer; and the fifth 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 stored in the constraint vector.

Chapter 4

TAO Solvers

TAO includes a variety of optimization algorithms for several classes of problems (unconstrained, bound-constrained, and PDE-constrained minimization, nonlinear least-squares, and complementarity). The TAO algorithms for solving these problems are detailed in this section, a particular algorithm can chosen by using the `TaoSetType()` function or using the command line arguments `-tao_type <name>`. For those interested in extending these algorithms or using new ones, please see Chapter 6 for more information.

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 likely perform best. 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. These algorithms and the associated options are briefly discussed in this chapter.

4.1.1 Nelder-Mead Method

The Nelder-Mead algorithm [26] 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 slow to converge or can even stagnate, and it performs poorly for large numbers of variables.

This solver keeps a set of $N+1$ sorted vectors $x_1, x_2, \ldots, x_{N+1}$ and their corresponding objective function values $f_1 \leq f_2 \leq \ldots \leq f_{N+1}$. 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 μ can be one of $\mu_0, 2\mu_0, \frac{1}{2}\mu_0, -\frac{1}{2}\mu_0$ depending on 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.

Two options 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 coordinate direction); the default value is 1.

`-tao_nm_mu <value>` sets the value of μ_0; the default is $\mu_0 = 1$.

4.1.2 Limited-Memory, Variable-Metric Method

The limited-memory, variable-metric method computes a positive definite approximation to the Hessian matrix from a limited number of previous iterates and gradient evaluations. A direction is then obtained by solving the system of equations

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

where H_k is the Hessian approximation obtained by using the BFGS update formula. 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, τ_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 by using the TAO solver `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 type of Hessian approximation used, the number of vectors stored for the approximation and the initialization/scaling of the approximation. These options can be configured using the `-tao_lmvm_mat_lmvm` prefix. For further detail, we refer the reader to the `MATLMVM` matrix type definitions in the PETSc Manual.

The LMVM algorithm also allows the user to define a custom initial Hessian matrix $H_{0,k}$ through the interface function `TaoLMVMSetH0()`. This user-provided initialization overrides any other scalar or diagonal initialization inherent to the LMVM approximation. The provided $H_{0,k}$ must be a PETSc `Mat` type object that represents a positive-definite matrix. The approximation prefers `MatSolve()` if the provided matrix has `MATOP_SOLVE` implemented. Otherwise, `MatMult()` is used in a KSP solve to perform the inversion of the user-provided initial Hessian.

In applications where `TaoSolve()` on the LMVM algorithm is repeatedly called to solve similar or related problems, `-tao_lmvm_recycle` flag can be used to prevent resetting the LMVM approximation between subsequent solutions. This recycling also avoids one extra function and gradient evalution, instead re-using the values already computed at the end of the previous solution.

This algorithm will be deprecated in the next version and replaced by the bounded quasi-Newton Line Search (BQNLS) algorithm that can solve both bound constrained and unconstrained problems.

4.1.3 Nonlinear Conjugate Gradient Method

The nonlinear conjugate gradient method can be viewed as an extension 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 TAO solver `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-Ribiére method, the Polak-Ribiére-Plus method [27], 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 η, the algorithm restarts using the gradient direction. The parameter η can be set by using the command line argument `-tao_cg_eta <real>`; 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 ρ_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, τ_k, that approximately solves the one-dimensional optimization problem

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

The Newton line search method can be selected by using the TAO solver `tao_nls`. The options available for this solver are listed in Table 4.1. 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, Nash 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,nash,stcg,gltr,petsc>`; `stcg` 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 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 arguments `-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 significantly affect 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 either the Hessian matrix or 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`

Table 4.1: Summary of `nls` options

Name	Value	Default	Description
`-tao_nls_ksp_type`	cg, nash, stcg, gltr, petsc	stcg	Type of Krylov subspace method to use when solving linear system
`-tao_nls_pc_type`	none, ahess, bfgs, petsc	bfgs	Type of preconditioner to use when solving linear system
`-tao_nls_bfgs_scale_type`	ahess, phess, bfgs	phess	Type of scaling matrix to use with BFGS preconditioner
`-tao_nls_sval`	real	0	Initial perturbation value
`-tao_nls_imin`	real	10^{-4}	Minimum initial perturbation value
`-tao_nls_imax`	real	100	Maximum initial perturbation value
`-tao_nls_imfac`	real	0.1	Factor applied to norm of gradient when initializing perturbation
`-tao_nls_pmax`	real	100	Maximum perturbation when increasing value
`-tao_nls_pgfac`	real	10	Growth factor applied to perturbation when increasing value
`-tao_nls_pmgfac`	real	0.1	Factor applied to norm of gradient when increasing perturbation
`-tao_nls_pmin`	real	10^{-12}	Minimum perturbation when decreasing value; smaller values set to zero
`-tao_nls_psfac`	real	0.4	Shrink factor applied to perturbation when decreasing value
`-tao_nls_pmsfac`	real	0.1	Factor applied to norm of gradient when decreasing perturbation
`-tao_nls_init_type`	constant, direction, interpolation	interpolation	Method used to initialize trust-region radius when using `nash`, `stcg`, or `gltr`

Table 4.2: Summary of `nls` options (continued)

Name	Value	Default	Description
`-tao_nls_mu1_i`	real	0.35	μ_1 in `interpolation` init
`-tao_nls_mu2_i`	real	0.50	μ_2 in `interpolation` init
`-tao_nls_gamma1_i`	real	0.0625	γ_1 in `interpolation` init
`-tao_nls_gamma2_i`	real	0.50	γ_2 in `interpolation` init
`-tao_nls_gamma3_i`	real	2.00	γ_3 in `interpolation` init
`-tao_nls_gamma4_i`	real	5.00	γ_4 in `interpolation` init
`-tao_nls_theta_i`	real	0.25	θ in `interpolation` init
`-tao_nls_update_type`	step, reduction, interpolation	step	Method used to update trust-region radius when using `nash`, `stcg`, or `gltr`
`-tao_nls_nu1`	real	0.25	ν_1 in `step` update
`-tao_nls_nu2`	real	0.50	ν_2 in `step` update
`-tao_nls_nu3`	real	1.00	ν_3 in `step` update
`-tao_nls_nu4`	real	1.25	ν_4 in `step` update
`-tao_nls_omega1`	real	0.25	ω_1 in `step` update
`-tao_nls_omega2`	real	0.50	ω_2 in `step` update
`-tao_nls_omega3`	real	1.00	ω_3 in `step` update
`-tao_nls_omega4`	real	2.00	ω_4 in `step` update
`-tao_nls_omega5`	real	4.00	ω_5 in `step` update
`-tao_nls_eta1`	real	10^{-4}	η_1 in `reduction` update
`-tao_nls_eta2`	real	0.25	η_2 in `reduction` update
`-tao_nls_eta3`	real	0.50	η_3 in `reduction` update
`-tao_nls_eta4`	real	0.90	η_4 in `reduction` update
`-tao_nls_alpha1`	real	0.25	α_1 in `reduction` update
`-tao_nls_alpha2`	real	0.50	α_2 in `reduction` update
`-tao_nls_alpha3`	real	1.00	α_3 in `reduction` update
`-tao_nls_alpha4`	real	2.00	α_4 in `reduction` update
`-tao_nls_alpha5`	real	4.00	α_5 in `reduction` update
`-tao_nls_mu1`	real	0.10	μ_1 in `interpolation` update
`-tao_nls_mu2`	real	0.50	μ_2 in `interpolation` update
`-tao_nls_gamma1`	real	0.25	γ_1 in `interpolation` update
`-tao_nls_gamma2`	real	0.50	γ_2 in `interpolation` update
`-tao_nls_gamma3`	real	2.00	γ_3 in `interpolation` update
`-tao_nls_gamma4`	real	4.00	γ_4 in `interpolation` update
`-tao_nls_theta`	real	0.05	θ in `interpolation` update

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 ρ_k is added when the direction returned by the Krylov subspace method is 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 last two 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 ρ_k is zero and a problem was detected with either the direction or the Krylov subspace method, the perturbation is initialized to

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

   where $g(x_k)$ is the gradient of the objective function and `imin` is set with the command line argument `-tao_nls_imin <real>` 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 λ_1 of the Hessian matrix is available. This value is used to initialize the perturbation to $\rho_{k+1} = \max\left\{\rho_{k+1}, -\lambda_1\right\}$ in this case.

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

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

   where $g(x_k)$ is the gradient of the objective function and `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 ρ_k is nonzero and no problems were detected with either the direction or Krylov subspace method, the perturbation is decreased to

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

   where $g(x_k)$ is the gradient of the objective function, `psfac` is set with the command line argument `-tao_nls_psfac` with a default value of 0.4, and `pmsfac` is set 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}.

Near a local minimizer to the unconstrained optimization problem, the Hessian matrix will be positive-semidefinite; the perturbation will shrink toward zero, and one would eventually observe a superlinear convergence rate.

When using `nash`, `stcg`, or `gltr` to solve the linear systems of equation, a trust-region radius needs to be initialized and updated. This trust-region radius simultaneously limits

the size of the step computed and reduces the number of iterations of the conjugate gradient method. 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 [7], 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 <real>` 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 $\|s_0\|$.

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 τ_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.

This algorithm will be deprecated in the next version and replaced by the bounded Newton Line Search (BNLS) algorithm that can solve both bound constrained and unconstrained problems.

4.1.5 Newton Trust-Region Method

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

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

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 Δ_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 by using the updated trust-region radius. The Newton trust-region method can be set by using the TAO solver `tao_ntr`. The options available for this solver are listed in Table 4.3. 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 Nash or Steihaug-Toint conjugate gradient methods or the 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 <nash,stcg,gltr>`; `stcg` is the default. Internally, the PETSc implementations for the Nash method, 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 Nash and Steihaug-Toint 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_ntr_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_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 significantly affect 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 arguments `-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 arguments `-tao_ntr_init_type <constant,direction,interpolation>`; `interpolation`, which chooses an initial value based on the interpolation scheme found in [7], 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 trust-region algorithm.

Table 4.3: Summary of `ntr` options

Name	Value	Default	Description
`-tao_ntr_ksp_type`	nash, stcg, gltr	stcg	Type of Krylov subspace method to use when solving linear system
`-tao_ntr_pc_type`	none, ahess, bfgs, petsc	bfgs	Type of preconditioner to use when solving linear system
`-tao_ntr_bfgs_scale_type`	ahess, bfgs	ahess	Type of scaling matrix to use with BFGS preconditioner
`-tao_ntr_init_type`	constant, direction, interpolation	interpolation	Method used to initialize trust-region radius
`-tao_ntr_mu1_i`	real	0.35	μ_1 in `interpolation` init
`-tao_ntr_mu2_i`	real	0.50	μ_2 in `interpolation` init
`-tao_ntr_gamma1_i`	real	0.0625	γ_1 in `interpolation` init
`-tao_ntr_gamma2_i`	real	0.50	γ_2 in `interpolation` init
`-tao_ntr_gamma3_i`	real	2.00	γ_3 in `interpolation` init
`-tao_ntr_gamma4_i`	real	5.00	γ_4 in `interpolation` init
`-tao_ntr_theta_i`	real	0.25	θ in `interpolation` init
`-tao_ntr_update_type`	reduction, interpolation	reduction	Method used to update trust-region radius
`-tao_ntr_eta1`	real	10^{-4}	η_1 in `reduction` update
`-tao_ntr_eta2`	real	0.25	η_2 in `reduction` update
`-tao_ntr_eta3`	real	0.50	η_3 in `reduction` update
`-tao_ntr_eta4`	real	0.90	η_4 in `reduction` update
`-tao_ntr_alpha1`	real	0.25	α_1 in `reduction` update
`-tao_ntr_alpha2`	real	0.50	α_2 in `reduction` update
`-tao_ntr_alpha3`	real	1.00	α_3 in `reduction` update
`-tao_ntr_alpha4`	real	2.00	α_4 in `reduction` update
`-tao_ntr_alpha5`	real	4.00	α_5 in `reduction` update
`-tao_ntr_mu1`	real	0.10	μ_1 in `interpolation` update
`-tao_ntr_mu2`	real	0.50	μ_2 in `interpolation` update
`-tao_ntr_gamma1`	real	0.25	γ_1 in `interpolation` update
`-tao_ntr_gamma2`	real	0.50	γ_2 in `interpolation` update
`-tao_ntr_gamma3`	real	2.00	γ_3 in `interpolation` update
`-tao_ntr_gamma4`	real	4.00	γ_4 in `interpolation` update
`-tao_ntr_theta`	real	0.05	θ in `interpolation` update

The `constant` method initializes the trust-region radius by using the value specified with the `-tao_trust0 <real>` 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 $\|s_0\|$.

The method for updating the trust-region radius is set with the command line arguments `-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.

This algorithm will be deprecated in the next version and replaced by the bounded Newton Trust Region (BNTR) algorithm that can solve both bound constrained and unconstrained problems.

4.1.6 BMRM

The Bundle Method for Regularized Risk Minimization (BMRM)[**?**] is a numerical approach to optimizing an unconstrained objective in the form of $f(x) + 0.5 * \lambda \|x\|^2$. Here f is a convex function that is finite on the whole space. λ is a positive weight parameter, and $\|x\|$ is the Euclidean norm of x. The algorithm only requires a routine which, given an x, returns the value of $f(x)$ and the gradient of f at x.

4.1.7 OWL-QN

The Orthant-Wise Limited-memory Quasi-Newton algorithm (OWL-QN)[1] is a numerical approach to optimizing an unconstrained objective in the form of $f(x) + \lambda \|x\|_1$. Here f is a convex and differentiable function, λ is a positive weight parameter, and $\|x\|_1$ is the $L1$ norm of x: $\sum_i |x_i|$. The algorithm only requires evaluating the value of f and its gradient.

4.2 Bound-Constrained Optimization

Bound-constrained optimization algorithms solve optimization problems of the form

$$\begin{array}{ll} \min\limits_x & f(x) \\ \text{subject to} & l \le x \le u. \end{array}$$

These solvers use the bounds on the variables as well as objective function, gradient, and possibly Hessian information.

For any unbounded variables, the bound value for the associated index can be set to `PETSC_INFINITY` for the upper bound and `PETSC_NINFINITY` for the lower bound. If all bounds are set to infinity, then the bounded algorithms are equivalent to their unconstrained counterparts.

Before introducing specific methods, we will first define two projection operations used by all bound constrained algorithms.

- Gradient projection:

$$\mathfrak{P}(g) = \begin{cases} 0 & \text{if } (x \leq l_i \wedge g_i > 0) \vee (x \geq u_i \wedge g_i < 0) \\ g_i & \text{otherwise} \end{cases}$$

- Bound projection:

$$\mathfrak{B}(x) = \begin{cases} l_i & \text{if } x_i < l_i \\ u_i & \text{if } x_i > u_i \\ x_i & \text{otherwise} \end{cases}$$

4.2.1 Bounded Newton-Krylov Methods

TAO features three bounded Newton-Krylov (BNK) class of algorithms, separated by their globalization methods: projected line search (BNLS), trust region (BNTR), and trust region with a projected line search fall-back (BNTL). They are available via the TAO solvers `TAOBNLS`, `TAOBNTR` and `TAOBNTL`, respectively, or the `-tao_type bnls/bntr/bntl` flag.

The BNK class of methods use an active-set approach to solve the symmetric system of equations,

$$H_k p_k = -g_k,$$

only for inactive variables in the interior of the bounds. The active-set estimation is based on Bertsekas [6] with the following variable index categories:

$$\begin{aligned} \text{lower bounded}: \quad \mathcal{L}(x) &= \{i \,:\, x_i \leq l_i + \epsilon \,\wedge\, g(x)_i > 0\}, \\ \text{upper bounded}: \quad \mathcal{U}(x) &= \{i \,:\, x_i \geq u_i + \epsilon \,\wedge\, g(x)_i < 0\}, \\ \text{fixed}: \quad \mathcal{F}(x) &= \{i \,:\, l_i = u_i\}, \\ \text{active-set}: \quad \mathcal{A}(x) &= \{\mathcal{L}(x) \bigcup \mathcal{U}(x) \bigcup \mathcal{F}(x)\}, \\ \text{inactive-set}: \quad \mathcal{I}(x) &= \{1, 2, \ldots, n\} \setminus \mathcal{A}(x). \end{aligned}$$

At each iteration, the bound tolerance is estimated as $\epsilon_{k+1} = \min(\epsilon_k, ||w_k||_2)$ with $w_k = x_k - \mathfrak{B}(x_k - \beta D_k g_k)$, where the diagonal matrix D_k is an approximation of the Hessian inverse H_k^{-1}. The initial bound tolerance ϵ_0 and the step length β have default values of 0.001 and can be adjusted using `-tao_bnk_as_tol` and `-tao_bnk_as_step` flags, respectively. The active-set estimation can be disabled using the option `-tao_bnk_as_type none`, in which case the algorithm simply uses the current iterate with no bound tolerances to determine which variables are actively bounded and which are free.

BNK algorithms invert the reduced Hessian using a Krylov iterative method. Trust-region conjugate gradient methods (`KSPNASH`, `KSPSTCG`, and `KSPGLTR`) are required for the BNTR and BNTL algorithms, and recommended for the BNLS algorithm. The preconditioner type can be changed using the `-tao_bnk_pc_type none/ilu/icc/jacobi/lmvm`. The

`lmvm` option, which is also the default, preconditions the Krylov solution with a `MATLMVM` matrix. The remaining supported preconditioner types are default PETSc types. If Jacobi is selected, the diagonal values are safeguarded to be positive. `icc` and `ilu` options produce good results for problems with dense Hessians. The LMVM and Jacobi preconditioners are also used as the approximate inverse-Hessian in the active-set estimation. If neither are available, or if the Hessian matrix does not have `MATOP_GET_DIAGONAL` defined, then the active-set estimation falls back onto using an identity matrix in place of D_k (this is equivalent to estimating the active-set using a gradient descent step).

A special option is available to "accelerate" the convergence of the BNK algorithms by taking a finite number of BNCG iterations at each Newton iteration. By default, the number of BNCG iterations is set to zero and the algorithms do not take any BNCG steps. This can be changed using the option flag `-tao_bnk_max_cg_its <i>`. While this reduces the number of Newton iterations, in practice it simply trades off the Hessian evaluations in the BNK solver for more function and gradient evaluations in the BNCG solver. However, it may be useful for certain types of problems where the Hessian evaluation is disproportionately more expensive than the objective function or its gradient.

Bounded Newton Line Search (BNLS)

BNLS safeguards the Newton step by falling back onto a BFGS, scaled gradient, or gradient steps based on descent direction verifications. For problems with indefinite Hessian matrices, the step direction is calculated using a perturbed system of equations,

$$(H_k + \rho_k I)p_k = -g_k,$$

where ρ_k is a dynamically adjusted positive constant. The step is globalized using a projected More-Thuente line search. If a trust-region conjugate gradient method is used for the Hessian inversion, the trust radius is modified based on the line search step length.

Bounded Newton Trust Region (BNTR)

BNTR globalizes the Newton step using a trust region method based on the predicted versus actual reduction in the cost function. The trust radius is increased only if the accepted step is at the trust region boundary. The reduction check features a safeguard for numerical values below machine epsilon, scaled by the latest function value, where the full Newton step is accepted without modification.

Bounded Newton Trust Region with Line Search Fall-back (BNTL)

BNTL safeguards the trust-region globalization such that a line search is used in the event that the step is initially rejected by the predicted versus actual decrease comparison. If the line search fails to find a viable step length for the Newton step, it falls back onto a scaled gradient or a gradient descent step. The trust radius is then modified based on the line search step length.

4.2.2 Bounded Nonlinear Conjugate Gradient

BNCG extends the unconstrained nonlinear conjugate gradient algorithm to bound constraints via gradient projections and a bounded More-Thuente line search.

Like its unconstrained counterpart, BNCG offers gradient descent and a variety of CG updates: Fletcher-Reeves, Polak-Ribiére, Polak-Ribiére-Plus, Hestenes-Stiefel, Dai-Yuan, Hager-Zhang, Dai-Kou, Kou-Dai, and the Self-Scaling Memoryless (SSML) BFGS, DFP, and Broyden methods. These methods can be specified by using the command line argument `-tao_bncg_type <gd,fr,pr,prp,hs,dy,hz,dk,kd,ssml_bfgs,ssml_dfp,ssml_brdn>`, respectively. The default value is `ssml_bfgs`. We have scalar preconditioning for these methods, and it is controlled by the flag `tao_bncg_alpha`. To disable rescaling, use $\alpha = -1.0$, otherwise $\alpha \in [0, 1]$. BNCG is available via the TAO solver `TAOBNCG` or the `-tao_type bncg` flag.

Some individual methods also contain their own parameters. The Hager-Zhang and Dou-Kai methods have a parameter that determines the minimum amount of contribution the previous search direction gives to the next search direction. The flags are `-tao_bncg_hz_eta` and `-tao_bncg_dk_eta`, and by default are set to 0.4 and 0.5 respectively. The Kou-Dai method has multiple parameters. `-tao_bncg_zeta` serves the same purpose as the previous two; set to 0.1 by default. There is also a parameter to scale the contribution of $y_k \equiv \nabla f(x_k) - \nabla f(x_{k-1})$ in the search direction update. It is controlled by `-tao_bncg_xi`, and is equal to 1.0 by default. There are also times where we want to maximize the descent as measured by $\nabla f(x_k)^T d_k$, and that may be done by using a negative value of ξ; this achieves better performance when not using the diagonal preconditioner described next. This is enabled by default, and is controlled by `-tao_bncg_neg_xi`. Finally, the Broyden method has its convex combination parameter, set with `-tao_bncg_theta`. We have this as 1.0 by default, i.e. it is by default the BFGS method. One can also individually tweak the BFGS and DFP contributions using the multiplicative constants `-tao_bncg_<bfgs,dfp>_scale`; both are set to 1 by default.

All methods can be scaled using the parameter `-tao_bncg_alpha`, which continuously varies in $[0, 1]$. The default value is set depending on the method from initial testing.

BNCG also offers a special type of method scaling. It employs Broyden diagonal scaling as an option for its CG methods, turned on with the flag `-tao_bncg_diag_scaling`. Formulations for both the forward (regular) and inverse Broyden methods are developed, controlled by the flag `-tao_bncg_mat_lmvm_forward`. It is set to True by default. Whether one uses the forward or inverse formulations depends on the method being used. For example, in our preliminary computations, the forward formulation works better for the SSML_BFGS method, but the inverse formulation works better for the Hestenes-Stiefel method. The convex combination parameter for the Broyden scaling is controlled by `-tao_bncg_mat_lmvm_theta`, and is 0 by default. We also employ rescaling of the Broyden diagonal, which aids the line-search immensely. The rescaling parameter is controlled by `-tao_bncg_mat_lmvm_alpha`, and should be $\in [0, 1]$. One can disable rescaling of the Broyden diagonal entirely by setting `-tao_bncg_mat_lmvm_sigma_hist 0`.

One can also supply their own preconditioner, serving as a Hessian initialization to the above diagonal scaling. The appropriate user function in the code is `TaoBNCGSetH0(tao, H0)` where `H0` is the user-defined `Mat` object that serves as a preconditioner. For an example

of similar usage, see `tao/examples/tutorials/ex3.c`.

The active set estimation uses the Bertsekas-based method described in Section 4.2.1, which can be deactivated using `-tao_bncg_as_type none`, in which case the algorithm will use the current iterate to determine the bounded variables with no tolerancing and no look-ahead step. As in the BNK algorithm, the initial bound tolerance and estimator step length used in the Bertsekas method can be set via `-tao_bncg_as_tol` and `-tao_bncg_as_step`, respectively.

In addition to automatic scaled gradient descent restarts under certain local curvature conditions, we also employ restarts based on a check on descent direction such that $\nabla f(x_k)^T d_k \in [-10^{11}, -1^{-9}]$. Furthermore, we allow for a variety of alternative restart strategies, all disabled by default. The `-tao_bncg_unscaled_restart` flag allows one to disable rescaling of the gradient for gradient descent steps. The `-tao_bncg_spaced_restart` flag tells the solver to restart every Mn iterations, where n is the problem dimension and M is a constant determined by `-tao_bncg_min_restart_num` and is 6 by default. We also have dynamic restart strategies based on checking if a function is locally quadratic; if so, go do a gradient descent step. The flag is `-tao_bncg_dynamic_restart`, disabled by default since the CG solver usually does better in those cases anyway. The minimum number of quadratic-like steps before a restart is set using `-tao_bncg_min_quad` and is 6 by default.

4.2.3 Trust-Region Newton Method

The TRON [20] 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 gradient iterations. 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 remaining set of "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. The TRON algorithm applies a trust region to the conjugate gradients to ensure convergence. The initial trust-region radius can be set by using the command `TaoSetInitialTrustRegionRadius()`, and the current trust region size can be found by using the command `TaoGetCurrentTrustRegionRadius()`. The initial trust region can significantly alter the rate of convergence for the algorithm and should be tuned and adjusted for optimal performance.

4.2.4 Bound-constrained Limited-Memory Variable-Metric Method

BLMVM is a limited-memory, variable-metric method and 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 by using the TAO solver `tao_blmvm`. For more details, please see the LMVM section in the unconstrained algorithms as well as the LMVM matrix documentation in the PETSc manual.

This algorithm will be deprecated in the next version in favor of the bounded quasi-Newton line search (BQNLS) algorithm.

4.2.5 Bounded Quasi-Newton-Krylov

BQNK algorithms use the BNK infrastructure, but replace the exact Hessian with a quasi-Newton approximation. The matrix-free forward product operation based on quasi-Newton update formulas are used in conjunction with Krylov solvers to compute step directions. The quasi-Newton inverse application is used to precondition the Krylov solution, and typically helps converge to a step direction in $\mathcal{O}(10)$ iterations. This approach is most useful with quasi-Newton update types such as Symmetric Rank-1 that cannot strictly guarantee positive-definiteness. The BNLS framework with Hessian shifting, or the BNTR framework with trust region safeguards, can successfully compensate for the Hessian approximation becoming indefinite.

Similar to the full Newton-Krylov counterpart, BQNK algorithms come in three forms separated by the globalization technique: line search (BQNKLS), trust region (BQNKTR) and trust region w/ line search fall-back (BQNKTL). These algorithms are available via `tao_type <bqnkls, bqnktr, bqnktl>`.

4.2.6 Bounded Quasi-Newton Line Search (BQNLS)

BQNLS algorithm uses the BNLS infrastructure, but replaces the step calculation with a direct inverse application of the approximate Hessian based on quasi-Newton update formulas. No Krylov solver is used in the solution, and therefore the quasi-Newton method chosen must guarantee a positive-definite Hessian approximation. This algorithm is available via `tao_type bqnls`.

4.3 PDE-Constrained Optimization

TAO solves PDE-constrained optimization problems of the form

$$\begin{aligned} \min_{u,v} \quad & f(u,v) \\ \text{subject to} \quad & g(u,v) = 0, \end{aligned}$$

where the state variable u is the solution to the discretized partial differential equation defined by g and parametrized by the design variable v, and f is an objective function. The Lagrange multipliers on the constraint are denoted by y. This method is set by using the linearly constrained augmented Lagrangian TAO solver `tao_lcl`.

We make two main assumptions when solving these problems: the objective function and PDE constraints have been discretized so that we can treat the optimization problem as finite dimensional and $\nabla_u g(u,v)$ is invertible for all u and v.

4.3.1 Linearly-Constrained Augmented Lagrangian Method

Given the current iterate (u_k, v_k, y_k), the linearly constrained augmented Lagrangian method approximately solves the optimization problem

$$\begin{aligned} \min_{u,v} \quad & \tilde{f}_k(u,v) \\ \text{subject to} \quad & A_k(u-u_k) + B_k(v-v_k) + g_k = 0, \end{aligned}$$

where $A_k = \nabla_u g(u_k, v_k)$, $B_k = \nabla_v g(u_k, v_k)$, and $g_k = g(u_k, v_k)$ and

$$\tilde{f}_k(u, v) = f(u, v) - g(u, v)^T y^k + \frac{\rho_k}{2}\|g(u, v)\|^2$$

is the augmented Lagrangian function. This optimization problem is solved in two stages. The first computes the Newton direction and finds a feasible point for the linear constraints. The second computes a reduced-space direction that maintains feasibility with respect to the linearized constraints and improves the augmented Lagrangian merit function.

Newton Step

The Newton direction is obtained by fixing the design variables at their current value and solving the linearized constraint for the state variables. In particular, we solve the system of equations

$$A_k du = -g_k$$

to obtain a direction du. We need a direction that provides sufficient descent for the merit function

$$\frac{1}{2}\|g(u, v)\|^2.$$

That is, we require $g_k^T A_k du < 0$.

If the Newton direction is a descent direction, then we choose a penalty parameter ρ_k so that du is also a sufficient descent direction for the augmented Lagrangian merit function. We then find α to approximately minimize the augmented Lagrangian merit function along the Newton direction.

$$\min_{\alpha \geq 0} \ \tilde{f}_k(u_k + \alpha du, v_k).$$

We can enforce either the sufficient decrease condition or the Wolfe conditions during the search procedure. The new point,

$$\begin{aligned} u_{k+\frac{1}{2}} &= u_k + \alpha_k du \\ v_{k+\frac{1}{2}} &= v_k, \end{aligned}$$

satisfies the linear constraint

$$A_k(u_{k+\frac{1}{2}} - u_k) + B_k(v_{k+\frac{1}{2}} - v_k) + \alpha_k g_k = 0.$$

If the Newton direction computed does not provide descent for the merit function, then we can use the steepest descent direction $du = -A_k^T g_k$ during the search procedure. However, the implication that the intermediate point approximately satisfies the linear constraint is no longer true.

Modified Reduced-Space Step

We are now ready to compute a reduced-space step for the modified optimization problem:

$$\begin{aligned} \min_{u,v} \quad & \tilde{f}_k(u, v) \\ \text{subject to} \quad & A_k(u - u_k) + B_k(v - v_k) + \alpha_k g_k = 0. \end{aligned}$$

We begin with the change of variables

$$\begin{array}{ll} \min\limits_{du,dv} & \tilde{f}_k(u_k + du, v_k + dv) \\ \text{subject to} & A_k du + B_k dv + \alpha_k g_k = 0 \end{array}$$

and make the substitution

$$du = -A_k^{-1}(B_k dv + \alpha_k g_k).$$

Hence, the unconstrained optimization problem we need to solve is

$$\min_{dv} \quad \tilde{f}_k(u_k - A_k^{-1}(B_k dv + \alpha_k g_k), v_k + dv),$$

which is equivalent to

$$\min_{dv} \quad \tilde{f}_k(u_{k+\frac{1}{2}} - A_k^{-1} B_k dv, v_{k+\frac{1}{2}} + dv).$$

We apply one step of a limited-memory quasi-Newton method to this problem. The direction is obtain by solving the quadratic problem

$$\min_{dv} \quad \tfrac{1}{2} dv^T \tilde{H}_k dv + \tilde{g}_{k+\frac{1}{2}}^T dv,$$

where $\tilde{H}_k$ is the limited-memory quasi-Newton approximation to the reduced Hessian matrix, a positive-definite matrix, and $\tilde{g}_{k+\frac{1}{2}}$ is the reduced gradient.

$$\begin{array}{rcl} \tilde{g}_{k+\frac{1}{2}} & = & \nabla_v \tilde{f}_k(u_{k+\frac{1}{2}}, v_{k+\frac{1}{2}}) - \nabla_u \tilde{f}_k(u_{k+\frac{1}{2}}, v_{k+\frac{1}{2}}) A_k^{-1} B_k \\ & = & d_{k+\frac{1}{2}} + c_{k+\frac{1}{2}} A_k^{-1} B_k \end{array}$$

The reduced gradient is obtained from one linearized adjoint solve

$$y_{k+\frac{1}{2}} = A_k^{-T} c_{k+\frac{1}{2}}$$

and some linear algebra

$$\tilde{g}_{k+\frac{1}{2}} = d_{k+\frac{1}{2}} + y_{k+\frac{1}{2}}^T B_k.$$

Because the Hessian approximation is positive definite and we know its inverse, we obtain the direction

$$dv = -H_k^{-1} \tilde{g}_{k+\frac{1}{2}}$$

and recover the full-space direction from one linearized forward solve,

$$du = -A_k^{-1} B_k dv.$$

Having the full-space direction, which satisfies the linear constraint, we now approximately minimize the augmented Lagrangian merit function along the direction.

$$\min_{\beta \geq 0} \quad \tilde{f}_k(u_{k+\frac{1}{2}} + \beta du, v_{k+\frac{1}{2}} + \beta dv)$$

We enforce the Wolfe conditions during the search procedure. The new point is

$$\begin{aligned} u_{k+1} &= u_{k+\frac{1}{2}} + \beta_k du \\ v_{k+1} &= v_{k+\frac{1}{2}} + \beta_k dv. \end{aligned}$$

The reduced gradient at the new point is computed from

$$\begin{aligned} y_{k+1} &= A_k^{-T} c_{k+1} \\ \tilde{g}_{k+1} &= d_{k+1} - y_{k+1}^T B_k, \end{aligned}$$

where $c_{k+1} = \nabla_u \tilde{f}_k(u_{k+1}, v_{k+1})$ and $d_{k+1} = \nabla_v \tilde{f}_k(u_{k+1}, v_{k+1})$. The multipliers y_{k+1} become the multipliers used in the next iteration of the code. The quantities $v_{k+\frac{1}{2}}$, v_{k+1}, $\tilde{g}_{k+\frac{1}{2}}$, and $\tilde{g}_{k+1}$ are used to update H_k to obtain the limited-memory quasi-Newton approximation to the reduced Hessian matrix used in the next iteration of the code. The update is skipped if it cannot be performed.

4.4 Nonlinear Least-Squares

Given a function $F : \mathbb{R}^n \to \mathbb{R}^m$, the nonlinear least-squares problem minimizes

$$f(x) = \|F(x)\|_2^2 = \sum_{i=1}^{m} F_i(x)^2. \tag{4.1}$$

The nonlinear equations F should be specified with the function `TaoSetSeparableObjectiveFunction()`. Although the current version of TAO has no derivative-based algorithms for nonlinear least squares, we anticipate that such an algorithm will be available in the next release.

4.4.1 POUNDerS

One algorithm for solving the least squares problem (4.1) when the Jacobian of the residual vector F is unavailable is the model-based POUNDerS (Practical Optimization Using No Derivatives for sums of Squares) algorithm (`tao_pounders`). POUNDerS employs a derivative-free trust-region framework as described in [8] in order to converge to local minimizers. An example of this version of POUNDerS applied to a practical least-squares problem can be found in [18].

Derivative-Free Trust-Region Algorithm

In each iteration k, the algorithm maintains a model $m_k(x)$, described below, of the nonlinear least squares function f centered about the current iterate x_k.

If one assumes that the maximum number of function evaluations has not been reached and that $\|\nabla m_k(x_k)\|_2 >$`gtol`, the next point x_+ to be evaluated is obtained by solving the trust-region subproblem

$$\min \{m_k(x) : \|x - x_k\|_p \leq \Delta_k, \}, \tag{4.2}$$

where Δ_k is the current trust-region radius. By default we use a trust-region norm with $p = \infty$ and solve (4.2) with the BLMVM method described in Section 4.2.4. While the subproblem is a bound-constrained quadratic program, it may not be convex and the BQPIP and GPCG methods may not solve the subproblem. Therefore, either BLMVM (the default) or TRON should be used. Note: TRON uses its own internal trust region that may interfere with the infinity-norm trust region used in the model problem (4.2).

The residual vector is then evaluated to obtain $F(x_+)$ and hence $f(x_+)$. The ratio of actual decrease to predicted decrease,

$$\rho_k = \frac{f(x_k) - f(x_+)}{m_k(x_k) - m_k(x_+)},$$

as well as an indicator, `valid`, on the model's quality of approximation on the trust region is then used to update the iterate,

$$x_{k+1} = \begin{cases} x_+ & \text{if } \rho_k \geq \eta_1 \\ x_+ & \text{if } 0 < \rho_k < \eta_1 \text{ and } \texttt{valid=true} \\ x_k & \text{else,} \end{cases}$$

and trust-region radius,

$$\Delta_{k+1} = \begin{cases} \min(\gamma_1 \Delta_k, \Delta_{\max}) & \text{if } \rho_k \geq \eta_1 \text{ and } \|x_+ - x_k\|_p \geq \omega_1 \Delta_k \\ \gamma_0 \Delta_k & \text{if } \rho_k < \eta_1 \text{ and } \texttt{valid=true} \\ \Delta_k & \text{else,} \end{cases}$$

where $0 < \eta_1 < 1$, $0 < \gamma_0 < 1 < \gamma_1$, $0 < \omega_1 < 1$, and $\Delta_{\max}$ are constants.

If $\rho_k \leq 0$ and `valid` is `false`, the iterate and trust-region radius remain unchanged after the above updates, and the algorithm tests whether the direction $x_+ - x_k$ improves the model. If not, the algorithm performs an additional evaluation to obtain $F(x_k + d_k)$, where d_k is a model-improving direction.

The iteration counter is then updated, and the next model m_k is obtained as described next.

Forming the Trust-Region Model

In each iteration, POUNDerS uses a subset of the available evaluated residual vectors $\{F(y_1), F(y_2), \cdots\}$ to form an interpolatory quadratic model of each residual component. The m quadratic models

$$q_k^{(i)}(x) = F_i(x_k) + (x - x_k)^T g_k^{(i)} + \frac{1}{2}(x - x_k)^T H_k^{(i)}(x - x_k), \qquad i = 1, \ldots, m \tag{4.3}$$

thus satisfy the interpolation conditions

$$q_k^{(i)}(y_j) = F_i(y_j), \qquad i = 1, \ldots, m;\ j = 1, \ldots, l_k$$

on a common interpolation set $\{y_1, \cdots, y_{l_k}\}$ of size $l_k \in [n+1, \texttt{npmax}]$.

The gradients and Hessians of the models in (4.3) are then used to construct the master model,

$$m_k(x) = f(x_k)+2(x-x_k)^T \sum_{i=1}^{m} F_i(x_k) g_k^{(i)} + (x-x_k)^T \sum_{i=1}^{m} \left(g_k^{(i)} \left(g_k^{(i)} \right)^T + F_i(x_k) H_k^{(i)} \right) (x-x_k). \tag{4.4}$$

The process of forming these models also computes the indicator `valid` of the model's local quality.

Parameters

POUNDerS supports the following parameters that can be set from the command line or PETSc options file:

`-tao_pounders_delta <delta>` The initial trust-region radius (> 0, real). This is used to determine the size of the initial neighborhood within which the algorithm should look.

`-tao_pounders_npmax <npmax>` The maximum number of interpolation points used ($n + 2 \leq$ `npmax` $\leq 0.5(n+1)(n+2)$). This input is made available to advanced users. We recommend the default value (`npmax`$= 2n+1$) be used by others.

`-tao_pounders_gqt` Use the gqt algorithm to solve the subproblem (4.2) (uses $p = 2$) instead of BQPIP.

`-pounders_subsolver` If the default BQPIP algorithm is used to solve the subproblem (4.2), the parameters of the subproblem solver can be accessed using the command line options prefix `-pounders_subsolver_`. For example,

```
-pounders_subsolver_tao_gatol 1.0e-5
```

sets the gradient tolerance of the subproblem solver to 10^{-5}.

Additionally, the user provides an initial solution vector, a vector for storing the separable objective function, and a routine for evaluating the residual vector F. These are described in detail in Sections 3.3.3 and 3.5.2. Here we remark that because gradient information is not available for scaling purposes, it can be useful to ensure that the problem is reasonably well scaled. A simple way to do so is to rescale the decision variables x so that their typical values are expected to lie within the unit hypercube $[0, 1]^n$.

Convergence Notes

Because the gradient function is not provided to POUNDerS, the norm of the gradient of the objective function is not available. Therefore, for convergence criteria, this norm is approximated by the norm of the model gradient and used only when the model gradient is deemed to be a reasonable approximation of the gradient of the objective. In practice, the typical grounds for termination for expensive derivative-free problems is the maximum number of function evaluations allowed.

4.5 Complementarity

Mixed complementarity problems, or box-constrained variational inequalities, are related to nonlinear systems of equations. They are defined by a continuously differentiable function, $F : \mathbb{R}^n \to \mathbb{R}^n$, 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, $\boldsymbol{x}^* \in [\ell, u]$ is a solution to MCP(F, ℓ, u) if for each $i \in \{1, \ldots, n\}$ we have at least one of the following:

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

Note that when $\ell = \{-\infty\}^n$ and $u = \{\infty\}^n$, we have a nonlinear system of equations, and $\ell = \{0\}^n$ and $u = \{\infty\}^n$ correspond to the nonlinear complementarity problem [9].

Simple complementarity conditions arise from the first-order optimality conditions from optimization [17, 19]. In the simple bound-constrained optimization case, these conditions correspond to MCP(∇f, ℓ, u), where $f : \mathbb{R}^n \to \mathbb{R}$ 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 $\nabla f \geq 0$. If the solution is at the upper bound, then the function must be decreasing and $\nabla f \leq 0$. 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 [12], game theory [25], and finance [16].

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.

4.5.1 Semismooth Methods

TAO has two implementations of semismooth algorithms [24, 10, 11] for solving mixed complementarity problems. Both are based on a reformulation of the mixed complementarity problem as a nonsmooth system of equations using the Fischer-Burmeister function [13]. 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, $\phi : \mathbb{R}^2 \to \mathbb{R}$, is defined as

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

This function has the following key property,

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

used when reformulating the mixed complementarity problem as the system of equations $\Phi(x) = 0$, where $\Phi : \mathbb{R}^n \to \mathbb{R}^n$. The reformulation is defined componentwise 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 Φ is not differentiable everywhere but satisfies a semismoothness property [21, 28, 29]. 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 nonsmooth 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 [29] of Φ 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 [2] is used to find the new iteration. Nonmonotone searches [14] are also available by setting appropriate runtime options. See Section 5.4 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 by using the `tao_ssils` solver. 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 by using the runtime options `-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 by using the `tao_ssfls` solver. The descent test used is the same as above where the direction in this case corresponds to the first part of the piecewise linear arc searched by the projected line search. Both $\delta > 0$ and $\rho > 2$ can be modified by using the runtime options `-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 because of the presence of log functions, the feasibility-enforcing version of the algorithm, `tao_ssfls`, is a reasonable alternative.

4.6 Quadratic Solvers

Quadratic solvers solve optimization problems of the form

$$\begin{array}{ll} \min\limits_{x} & \frac{1}{2}x^T Q x + c^T x \\ \text{subject to} & l \geq x \geq u \end{array}$$

where the gradient and the Hessian of the objective are both constant.

4.6.1 Gradient Projection Conjugate Gradient Method

The GPCG [23] algorithm is much like the TRON algorithm, discussed in Section 4.2.3, 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 the options that apply to TRON except for trust-region options also apply to GPCG. It can be set by using the TAO solver `tao_gpcg` or via the optio flag `-tao_type gpcg`.

4.6.2 Interior-Point Newton's Method

The BQPIP algorithm is an interior-point method for bound constrained quadratic optimization. It can be set by using the TAO solver of `tao_bqpip` or via the option flag `-tao_type bgpip`. Since it assumes the objective function is quadratic, it evaluates the function, gradient, and Hessian only once. This method also requires the solution of systems of linear equations, whose solver can be accessed and modified with the command `TaoGetKSP()`.

Chapter 5

Advanced Options

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

5.1 Linear Solvers

One of the most computationally intensive phases of many optimization algorithms involves the solution of linear systems of 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

```
TaoGetKSP(Tao, KSP *);
```

command. With access to the KSP object, users can customize it for their application to achieve improved performance. Additional details on the KSP options in PETSc can be found in the PETSc users manual [5].

5.2 Monitors

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

```
TaoSetMonitor(Tao,
                  PetscErrorCode (*mon)(Tao,void*),
                  void*);
```

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

The routine set by `TaoSetMonitor()` 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.

5.3 Convergence Tests

Convergence of a solver can be defined in many ways. The methods TAO uses by default are mentioned in Section 3.4.1. 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 specify a customized test.

Users can set their own customized convergence tests of the form

```
PetscErrorCode  conv(Tao, void*);
```

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, one uses the command

```
TaoSetConvergenceTest(Tao,
                      PetscErrorCode (*conv)(Tao,void*),
                      void*);
```

The second argument of this command is the convergence routine, and the final argument of the convergence test routine 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 by using the routine

```
TaoSetConvergedReason(Tao, TaoConvergedReason);
```

5.4 Line Searches

By using the command line option `-tao_ls_type`. Available line searches include Moré-Thuente [22], Armijo, gpcg, and unit.

The line search routines involve several parameters, which are set to defaults that are reasonable for many applications. The user can override the defaults by using the following options

- `-tao_ls_maxfev <max>`
- `-tao_ls_stepmin <min>`
- `-tao_ls_stepmax <max>`
- `-tao_ls_ftol <ftol>`
- `-tao_ls_gtol <gtol>`
- `-tao_ls_rtol <rtol>`

One should run a TAO program with the option `-help` for details. Users may write their own customized line search codes by modeling them after one of the defaults provided.

Chapter 6

Adding a Solver

One of the strengths of both TAO and PETSc is the ability to allow users to extend the built-in solvers with new user-defined algorithms. It is certainly possible to develop new optimization algorithms outside of TAO framework, but Using TAO to implement a solver has many advantages,

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, contains limited support for finite difference approximations, 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 that are helpful in an optimization algorithm. The availability of these tools means that the developers of the optimization solver do not have to write these utilities.

4. PETSc 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 PETSc 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. PETSc provides the user a convenient method for setting options at runtime, performance profiling, and debugging.

6.1 Header File

TAO solver implementation files must include the TAO implementation file `taoimpl.h`:

```
#include "petsc/private/taoimpl.h"
```

This file contains data elements that are generally kept hidden from application programmers, but may be necessary for solver implementations to access.

6.2 TAO Interface with Solvers

TAO solvers must be written in C or 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 options within the solver, view the solver, setup appropriate data structures, and destroy these data structures. In order to implement the conjugate gradient algorithm, for example, the following structure is useful.

```
typedef struct{

  PetscReal beta;
  PetscReal eta;
  PetscInt  ngradtseps;
  PetscInt  nresetsteps;
  Vec X_old;
  Vec G_old;

} TAO_CG;
```

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

6.2.1 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 will 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 that solves an unconstrained minimization problem using the conjugate gradient method.

```
PetscErrorCode TaoSolve_CG(Tao tao){

  TAO_CG  *cg = (TAO_CG *) tao->data;
  Vec x = tao->solution;
  Vec g = tao->gradient;
```

```
Vec s = tao->stepdirection;
PetscInt     iter=0;
PetscReal  gnormPrev,gdx,f,gnorm,steplength=0;
TaoLineSearchConvergedReason lsflag=TAO_LINESEARCH_CONTINUE_ITERATING;
TaoConvergedReason reason=TAO_CONTINUE_ITERATING;
PetscErrorCode ierr;

PetscFunctionBegin;

ierr = TaoComputeObjectiveAndGradient(tao,x,&f,g);CHKERRQ(ierr);
ierr = VecNorm(g,NORM_2,&gnorm);  CHKERRQ(ierr);

ierr = VecSet(s,0); CHKERRQ(ierr);

cg->beta=0;
gnormPrev = gnorm;

/* Enter loop */
while (1){

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

  cg->beta=(gnorm*gnorm)/(gnormPrev*gnormPrev);
  ierr = VecScale(s,cg->beta); CHKERRQ(ierr);
  ierr = VecAXPY(s,-1.0,g); CHKERRQ(ierr);

  ierr = VecDot(s,g,&gdx); CHKERRQ(ierr);
  if (gdx>=0){     /* If not a descent direction, use gradient */
    ierr = VecCopy(g,s); CHKERRQ(ierr);
    ierr = VecScale(s,-1.0); CHKERRQ(ierr);
    gdx=-gnorm*gnorm;
  }

  /* Line Search */
  gnormPrev = gnorm;  step=1.0;
  ierr = TaoLineSearchSetInitialStepLength(tao->linesearch,1.0);
  ierr = TaoLineSearchApply(tao->linesearch,x,&f,g,s,&steplength,&lsflag);
  ierr = TaoAddLineSearchCounts(tao); CHKERRQ(ierr);
  ierr = VecNorm(g,NORM_2,&gnorm);CHKERRQ(ierr);
  iter++;
}

PetscFunctionReturn(0);
```

```
}
```

The first line of this routine casts the second argument to a pointer to a `TAO_CG` data structure. This structure contains pointers to three vectors and a scalar that will be needed in the algorithm.

After declaring an initializing several variables, the solver lets TAO evaluate the function and gradient at the current point in the using the routine `TaoComputeObjectiveAndGradient()`. Other routines may be used to evaluate the Hessian matrix or evaluate constraints. TAO may obtain this information using direct evaluation or other means, but 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

```
PetscErrorCode TaoMonitor(Tao tao, PetscInt iter, PetscReal f,
               PetscReal res, PetscReal cnorm, PetscReal steplength,
               TaoConvergedReason *reason);
```

Most optimization algorithms are iterative, and solvers should include this command somewhere in each iteration. This routine records this information, and 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 argument is a nonnegative error that usually represents a measure of the infeasibility such as the norm of the constraints or violation of bounds. This number should be zero for unconstrained solvers. 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 the conjugate gradient algorithm and computations using Vec objects. 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

```
TaoLineSearchApply(TaoLineSearch ls, Vec x, PetscReal *f, Vec g,
                        TaoVec *s, PetscReal *steplength,
                        TaoLineSearchConvergedReason *lsflag)
```

passes the current solution, gradient, and objective value to the line search and returns a new solution, gradient, and objective value. More details on line searches can be found in Section 5.4. The details of the line search applied are specified elsewhere, when the line search is created.

TAO also includes support for linear solvers using PETSc KSP objects. 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 the PETSc users manual.

6.2.2 Creation Routine

The TAO solver is initialized 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.

```
PETSC_EXTERN PetscErrorCode TaoCreate_CG(Tao tao)
{
  TAO_CG *cg = (TAO_CG*)tao->data;
  const char *morethuente_type = TAOLINESEARCH_MT;
  PetscErrorCode ierr;

  PetscFunctionBegin;

  ierr = PetscNewLog(tao,&cg); CHKERRQ(ierr);
  tao->data = (void*)cg;
  cg->eta = 0.1;
  cg->delta_min = 1e-7;
  cg->delta_max = 100;
  cg->cg_type = CG_PolakRibierePlus;

  tao->max_it = 2000;
  tao->max_funcs = 4000;

  tao->ops->setup = TaoSetUp_CG;
  tao->ops->solve = TaoSolve_CG;
  tao->ops->view = TaoView_CG;
  tao->ops->setfromoptions = TaoSetFromOptions_CG;
  tao->ops->destroy = TaoDestroy_CG;

  ierr = TaoLineSearchCreate(((PetscObject)tao)->comm, &tao->linesearch);
  CHKERRQ(ierr);
  ierr = TaoLineSearchSetType(tao->linesearch, morethuente_type); CHKERRQ(ierr);
  ierr = TaoLineSearchUseTaoRoutines(tao->linesearch, tao); CHKERRQ(ierr);

  PetscFunctionReturn(0);
}
EXTERN_C_END
```

This routine declares some variables and then allocates memory for the `TAO_CG` data structure. Notice that the `Tao` object now has a pointer to this data structure (`tao->data`) so

it can be accessed by the other functions written for this solver implementation.

This routine also sets some default parameters particular to the conjugate gradient algorithm, sets 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 either by using direct calls setting parameters or by using options.

Finally, this solver 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 (if used).

6.2.3 Destroy Routine

Another routine needed by most solvers destroys the data structures created by earlier routines. For the nonlinear conjugate gradient method discussed earlier, the following routine destroys the two work vectors and the `TAO_CG` structure.

```
PetscErrorCode TaoDestroy_CG(TAO_SOLVER tao)
{
  TAO_CG *cg = (TAO_CG *) tao->data;
  PetscErrorCode    ierr;

  PetscFunctionBegin;

  ierr = VecDestroy(&cg->X_old); CHKERRQ(ierr);
  ierr = VecDestroy(&cg->G_old);CHKERRQ(ierr);

  PetscFree(tao->data);
  tao->data = NULL;

  PetscFunctionReturn(0);
}
```

This routine is called from within the `TaoDestroy()` routine. Only algorithm-specific data objects are destroyed in this routine; any objects indexed by TAO (`tao->linesearch`, `tao->ksp`, `tao->gradient`, etc.) will be destroyed by TAO immediately after the algorithm-specific destroy routine completes.

6.2.4 SetUp Routine

If the SetUp routine has been set by the initialization routine, TAO will call it during the execution of `TaoSolve()`. While this routine is optional, it is often provided to allocate the gradient vector, work vectors, and other data structures required by the solver. It should have the following form.

```
PetscErrorCode TaoSetUp_CG(Tao tao)
{
```

```
  PetscErrorCode ierr;
  TAO_CG *cg = (TAO_CG*)tao->data;
  PetscFunctionBegin;

  ierr = VecDuplicate(tao->solution,&tao->gradient); CHKERRQ(ierr);
  ierr = VecDuplicate(tao->solution,&tao->stepdirection); CHKERRQ(ierr);
  ierr = VecDuplicate(tao->solution,&cg->X_old); CHKERRQ(ierr);
  ierr = VecDuplicate(tao->solution,&cg->G_old); CHKERRQ(ierr);

  PetscFunctionReturn(0);
}
```

6.2.5 SetFromOptions Routine

The SetFromOptions routine should be used to check for any algorithm-specific options set by the user and will be called when the application makes a call to `TaoSetFromOptions()`. It should have the following form.

```
PetscErrorCode TaoSetFromOptions_CG(Tao tao, void *solver);
{
  PetscErrorCode ierr;
  TAO_CG *cg = (TAO_CG*)solver;
  PetscFunctionBegin;
  ierr = PetscOptionsReal("-tao_cg_eta","restart tolerance","",cg->eta,
                          &cg->eta,0);  CHKERRQ(ierr);
  ierr = PetscOptionsReal("-tao_cg_delta_min","minimum delta value","",
                          cg->delta_min,&cg->delta_min,0); CHKERRQ(ierr);
  ierr = PetscOptionsReal("-tao_cg_delta_max","maximum delta value","",
                          cg->delta_max,&cg->delta_max,0); CHKERRQ(ierr);
  PetscFunctionReturn(0);
}
```

6.2.6 View Routine

The View routine should be used to output any algorithm-specific information or statistics at the end of a solve. This routine will be called when the application makes a call to `TaoView()` or when the command line option `-tao_view` is used. It should have the following form.

```
PetscErrorCode TaoView_CG(Tao tao, PetscViewer viewer)
{
  TAO_CG *cg = (TAO_CG*)tao->data;
  PetscErrorCode ierr;

  PetscFunctionBegin;
  ierr = PetscViewerASCIIPushTab(viewer);
  ierr = PetscViewerASCIIPrintf(viewer,"Grad. steps: %d\n",cg->ngradsteps);
```

```
  ierr = PetscViewerASCIIPrintf(viewer,"Reset steps: %d\n",cg->nresetsteps);
  ierr = PetscViewerASCIIPopTab(viewer);
  PetscFunctionReturn(0);
}
```

6.2.7 Registering the Solver

Once a new solver is implemented, TAO needs to know the name of the solver and what function to use to create the solver. To this end, one can use the routine

```
TaoRegister(const char *name,
                const char *path,
                const char *cname,
                PetscErrorCode (*create) (Tao));
```

where `name` is the name of the solver (i.e., `tao_blmvm`), `path` is the path to the library containing the solver, `cname` is the name of the routine that creates the solver (in our case, `TaoCreate_CG`), and `create` is a pointer to that creation routine. If one is using dynamic loading, then the fourth argument will be ignored.

Once the solver has been registered, the new solver can be selected either by using the `TaoSetType()` function or by using the `-tao_type` command line option.

Index

Bibliography

[1] Galen Andrew and Jianfeng Gao. Scalable training of l1-regularized log-linear models. In *Proceedings of the 24th international conference on Machine learning (ICML)*, pages 33–40, 2007.

[2] L. Armijo. Minimization of functions having Lipschitz-continuous first partial derivatives. *Pacific Journal of Mathematics*, 16:1–3, 1966.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc Web page. See http://www.mcs.anl.gov/petsc.

[4] 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.

[5] 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.

[6] Dimitri P. Bertsekas. Projected Newton methods for optimization problems with simple constraints. *SIAM Journal on Control and Optimization*, 20:221–246, 1982.

[7] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. *Trust-Region Methods*. SIAM, Philadelphia, Pennsylvania, 2000.

[8] Andrew R. Conn, Katya Scheinberg, and Luís N. Vicente. *Introduction to Derivative-Free Optimization*. MPS/SIAM Series on Optimization. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[9] R. W. Cottle. *Nonlinear programs with positively bounded Jacobians*. PhD thesis, Department of Mathematics, University of California, Berkeley, California, 1964.

[10] 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.

[11] 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.

[12] M. C. Ferris and J. S. Pang. Engineering and economic applications of complementarity problems. *SIAM Review*, 39:669–713, 1997.

[13] A. Fischer. A special Newton–type optimization method. *Optimization*, 24:269–284, 1992.

[14] 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.

[15] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[16] J. Huang and J. S. Pang. Option pricing and linear complementarity. *Journal of Computational Finance*, 2:31–60, 1998.

[17] W. Karush. Minima of functions of several variables with inequalities as side conditions. Master's thesis, Department of Mathematics, University of Chicago, 1939.

[18] M. Kortelainen, T. Lesinski, J. Moré, W. Nazarewicz, J. Sarich, N. Schunck, M. V. Stoitsov, and S. M. Wild. Nuclear energy density optimization. *Physical Review C*, 82(2):024313, 2010.

[19] 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.

[20] C.-J. Lin and J. J. Moré. Newton's method for large bound-constrained optimization problems. *SIOPT*, 9(4):1100–1127, 1999.

[21] R. Mifflin. Semismooth and semiconvex functions in constrained optimization. *SIAM Journal on Control and Optimization*, 15:957–972, 1977.

[22] Jorge J. Moré and David Thuente. Line search algorithms with guaranteed sufficient decrease. Technical Report MCS-P330-1092, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[23] Jorge J. Moré and G. Toraldo. On the solution of large quadratic programming problems with bound constraints. *SIOPT*, 1:93–113, 1991.

[24] 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.

[25] J. F. Nash. Equilibrium points in N–person games. *Proceedings of the National Academy of Sciences*, 36:48–49, 1950.

[26] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[27] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.

[28] L. Qi. Convergence analysis of some algorithms for solving nonsmooth equations. *Mathematics of Operations Research*, 18:227–244, 1993.

[29] L. Qi and J. Sun. A nonsmooth version of Newton's method. *Mathematical Programming*, 58:353–368, 1993.