

# Numerical Optimization and the Toolkit for Advanced Optimization

Jason Sarich, Todd Munson, Stefan Wild

Mathematics and Computer Science Division,  
Argonne National Laboratory

August 15, 2012

# Part I

## Nonlinear Optimization

# Nonlinear Optimization

- Unconstrained Optimization
- Bound-constrained Optimization
- General Constrained Optimization

# Nonlinear Optimization

## Unconstrained Optimization Problem

$$f : \mathbb{R}^N \mapsto \mathbb{R}$$
$$\min_{x \in \mathbb{R}^N} f(x)$$

# Nonlinear Optimization

## Bound-constrained Optimization Problem

$$\begin{array}{ll} \min & f(x) \quad \text{(objective function)} \\ \text{subject to} & x_l \leq x \leq x_u \quad \text{(bounds)} \end{array}$$

# Nonlinear Optimization

## Constrained Optimization Problem

$$\begin{array}{ll} \min & f(x) \quad \text{(objective function)} \\ \text{subject to} & c_l \leq c(x) \leq c_u \quad \text{(constraints)} \end{array}$$

**Note:** TAO is not able to solve constrained optimization problems directly.

# Part II

## Algorithms

# Algorithms

Nonlinear optimization algorithms are iterative processes. In many cases, each iteration involve calculating a *search direction*, then function values and gradients along that direction are calculated until certain conditions are met.

- Newton's Method
- Quasi-Newton Methods
- Conjugate Gradient

# Algorithms

## Newton's Method

- **Step 0** Choose initial vector  $x_0$
- **Step 1** Compute gradient  $\nabla f(x_k)$  and Hessian  $\nabla^2 f(x_k)$
- **Step 2** Calculate the direction  $d_{k+1}$  by solving the system:

$$\nabla^2 f(x_k)d_{k+1} = -\nabla f(x_k)$$

- **Step 3** Apply line search algorithm to obtain “acceptable” new vector:

$$x_{k+1} = x_k + \tau d_{k+1}$$

- **Return to Step 1**

## Problems with Newton's Method

- Hessian must be derived, computed, and stored
- Linear solve must be performed on Hessian

## Quasi-Newton Methods

Use approximate Hessian  $B_k \approx \nabla^2 f(x_k)$ . Choose a formula for  $B_k$  so that:

- $B_k$  relies on first derivative information only
- $B_k$  can be easily stored
- $B_k d_{k+1} = -\nabla f(x_k)$  can be easily solved

# Algorithms

## Conjugate Gradient Algorithms

These algorithms are an extension of the conjugate gradient methods for solving linear systems.

$$d_{k+1} = -\nabla f(x_k) + \beta_k d_k$$

Some possible choices of  $\beta_k$  ( $g_k = \nabla f(x_k)$ ):

$$\beta_k^{FR} = \left( \frac{\|g_{k+1}\|}{\|g_k\|} \right)^2, \quad \text{Fletcher-Reeves}$$

$$\beta_k^{PR} = \frac{\langle g_{k+1}, g_{k+1} - g_k \rangle}{\|g_k\|^2}, \quad \text{Polak-Ribière}$$

$$\beta_k^{PR+} = \max \{ \beta_k^{PR}, 0 \}, \quad \text{PR-plus}$$

## Derivate Free Algorithms

There are some applications for which it is not feasible to find the derivative of the objective function. There are some algorithms available that can solve these applications, but they can be very slow to converge.

- Pattern Searches
- Nelder-Mead Simplex
- Model-based methods
- Use finite differences

## Part III

# TAO

The process of nature by which all things change and which is to be followed for a life of harmony

## What does TAO do for you?

- Contains a library of optimization solvers for solving unconstrained, bound-constrained, and complementarity optimization problems. These solvers include Newton methods, Quasi-Newton methods, conjugate gradients, derivative free, and semi-smooth methods.
- Provides C, C++, and Fortran interfaces to these libraries
- Allows for large scale, sparse objects, and parallel applications
- Uses PETSc data structures and utilities

# TAO Solvers

	handles bounds	requires gradient	requires Hessian
lmvm	no	yes	no
nls	no	yes	yes
ntr	no	yes	yes
ntl	no	yes	yes
cg	no	yes	no
nm	no	no	no
blmvm	yes	yes	no
tron	yes	yes	yes
gpcg	yes	yes	no
pounders	yes	no	no

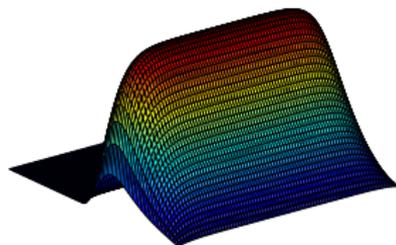
## Pressure in a Journal Bearing

$$\min \left\{ \int_{\mathcal{D}} \left\{ \frac{1}{2} w_q(x) \|\nabla v(x)\|^2 - w_l(x) v(x) \right\} dx : v \geq 0 \right\}$$

$$w_q(\xi_1, \xi_2) = (1 + \epsilon \cos \xi_1)^3$$

$$w_l(\xi_1, \xi_2) = \epsilon \sin \xi_1$$

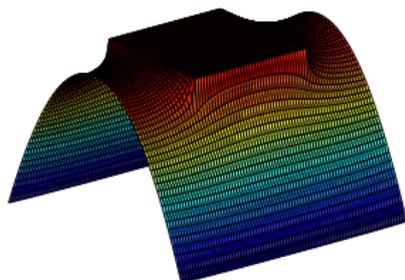
$$\mathcal{D} = (0, 2\pi) \times (0, 2b)$$



Number of active constraints depends on the choice of  $\epsilon$  in  $(0, 1)$ .  
 Nearly degenerate problem. Solution  $v \notin C^2$ .

## Minimal Surface with Obstacles

$$\min \left\{ \int_{\mathcal{D}} \sqrt{1 + \|\nabla v(x)\|^2} dx : v \geq v_L \right\}$$



Number of active constraints depends on the height of the obstacle. The solution  $v \notin C^1$ . Almost all multipliers are zero.

## Parallel Performance

Processors Used	BLMVM Iterations	Execution Time	Percentage of Time		
			AXPY	Dot	FG
8	996	1083.8	31	9	60
16	991	538.2	30	10	60
32	966	267.7	29	11	60
64	993	139.5	27	13	60
128	987	72.4	25	15	60
256	996	39.2	26	18	56
512	1000	21.6	23	22	53

Table: Scalability of BLMVM on Obstacle Problem with 2,560,000 variables.

## Mesh Sequencing

Mesh	niters	Time (s)
$71 \times 71$	6	0.58
$141 \times 141$	8	1.45
$281 \times 281$	10	2.85
$561 \times 561$	21	9.34
$1121 \times 1121$	†	†
$2241 \times 2241$	†	†
$4481 \times 4481$	†	†

Performance results without mesh sequencing on 140 nodes. The symbol † is used if there is no convergence after 100 iterations.

The results in the table show that the number of iterations grows as the mesh is refined but that for the finest meshes we terminate Newton's method after  $niters = 100$  iterations. Although Newton's method is mesh invariant, the starting point is assumed to be in the region of quadratic convergence. A more careful analysis of these results, however, shows that the starting point is not in the region of quadratic convergence for any of the grids.

## Mesh Sequencing

We now consider the use of mesh sequencing. We start with a mesh with  $71 \times 71$  grid points and, at each stage, use linear interpolation on the coarse mesh solution to obtain the starting point for the fine mesh. Thus, an  $n_x \times n_y$  coarse mesh becomes a fine  $(2n_x - 1) \times (2n_y - 1)$  mesh. We use the same termination condition at each level of refinement.

## Mesh Sequencing

Mesh	niters	Time (s)
$71 \times 71$	6	0.58
$141 \times 141$	3	0.44
$281 \times 281$	2	0.52
$561 \times 561$	2	1.31
$1121 \times 1121$	2	5.51
$2241 \times 2241$	2	19.5
$4481 \times 4481$	2	189

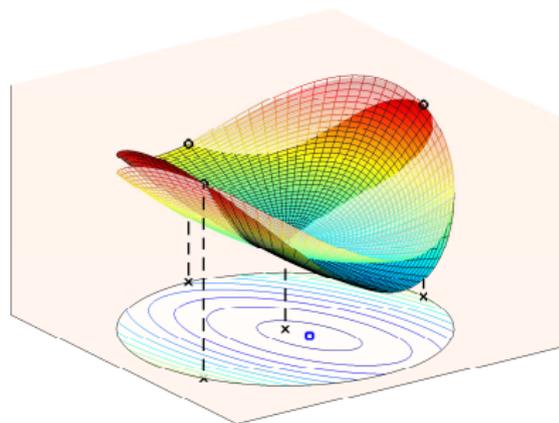
Performance results with mesh sequencing on 140 nodes.

The results in this table show the performance of the TRON solver on each mesh. After the solution is obtained on the coarsest mesh, the number of iterations per mesh is either two or three. This is the desired behavior for mesh sequencing. Because the number of Krylov iterations increases on finer meshes, solution times per level grow at a faster than linear rate. Better preconditioners may further improve performance.

# POUNDER - Model-based Derivate-free optimization

using an interpolating quadratic,

$$q_k(x_k + y_i) = f(x_k + y_i), \quad \forall y_i \in \mathcal{Y}_k.$$



$$n = 2, |\mathcal{Y}_k| = 4$$

- Function values are all you have
- Other models possible
  - Only provide local approximation
  - Coarse models  $\leftrightarrow$  smooth noise

# POUNDERS - Nonlinear Least Squares

$$f(x) = \frac{1}{2} \sum_{i=1}^p (S_i(x) - d_i)^2$$

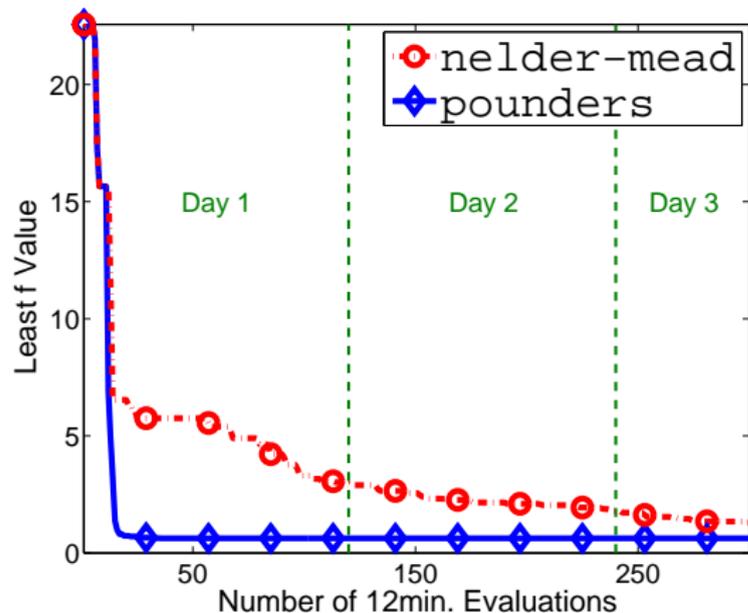
- Obtain a vector of output  $S_1(x), \dots, S_p(x)$  with each simulation
- Approximate:

$$\begin{aligned} \nabla f(x) &= \sum_i \nabla \mathbf{S}_i(\mathbf{x})(S_i(x) - d_i) \\ &\rightarrow \sum_i \nabla \mathbf{m}_i(\mathbf{x})(S_i(x) - d_i) \end{aligned}$$

$$\begin{aligned} \nabla^2 f(x) &= \sum_i \nabla \mathbf{S}_i(\mathbf{x}) \nabla \mathbf{S}_i(\mathbf{x})^T + \sum_i (S_i(x) - d_i) \nabla^2 \mathbf{S}_i(\mathbf{x}) \\ &\rightarrow \sum_i \nabla \mathbf{m}_i(\mathbf{x}) \nabla \mathbf{m}_i(\mathbf{x})^T + \sum_i (S_i(x) - d_i) \nabla^2 \mathbf{m}_i(\mathbf{x}) \end{aligned}$$

- Model  $f$  via Gauss-Newton or similar

## POUNDERS for hfbtho



- 72 cores on Jazz
- 12 wall-clock minutes per  $f(x)$
- POUNDERS: acceptable  $x$  in 3.2 hours
- Nelder-Mead: no acceptable  $x$  in 60 hours

## What TAO doesn't do

- Application Modeling
- Derivatives
- Linear programming
- Constrained optimization
- Integer programming
- Global minimization

# TAO Applications

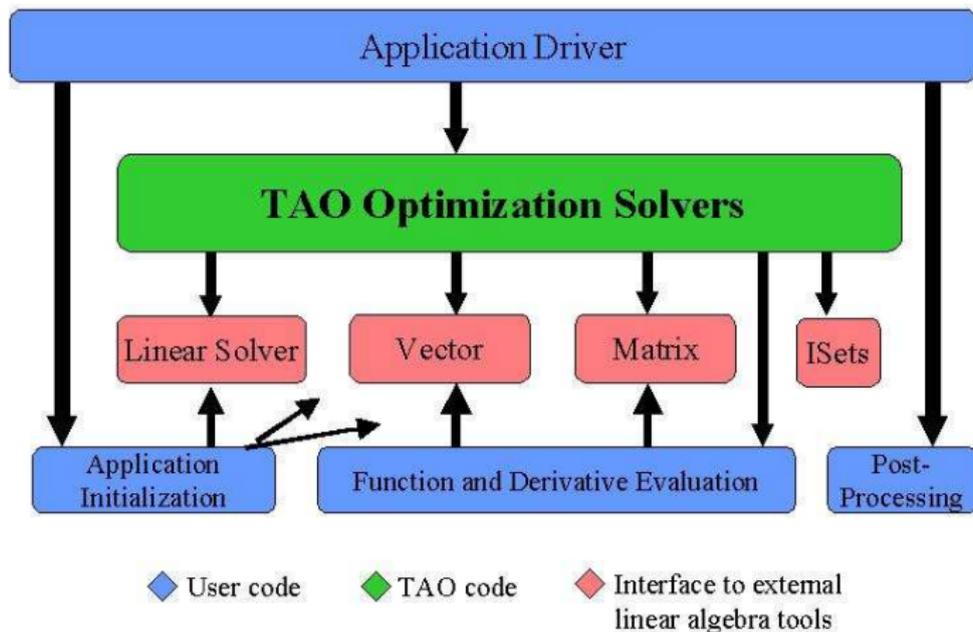
## Using TAO

There are two parts to solving an optimization application with TAO:

- Routines to evaluate an objective function, define constraints on the variables, and provide derivative information.
- A driver program (`main`) that creates a **TAO solver** with desired algorithmic options and tolerances and sets up user evaluation routines.

TAO uses **Matrix**, **Vector**, and **KSP** objects from PETSc.

# TAO Application



# TAO Applications

What do you need to do for the [User Routines](#)?

You need to write C, C++, or Fortran functions that:

- Set the initial variable vector (optional)
- Compute the objective function value at a given vector
- Compute the gradient at a given vector
- Compute the Hessian matrix at a given vector (for Newton methods)
- Set the variable bounds (for bounded optimization)

## TAO Examples

TAO has some example applications (in C and Fortran) included in the source distribution for you to test the TAO installation, learn about TAO features, and reference for creating your own applications

unconstrained	bound	least-squares
eptorsion1.c	jbearing2.c	chwirut1.c
eptorsion2.c	plate2.c	chwirut2.c
eptorsion2f.F	plate2f.F	chwirut1f.F
minsurf1.c		chwirut2f.F
minsurf2.c		
rosenbrock1.c		
rosenbrock1f.F		

## TAO Applications

Create a data structure that contains any state information, such as parameter values or data viewers, that the evaluation routines will need.  
For example:

```
typedef struct {
    PetscReal    epsilon; /* application parameter */
    PetscInt     n;       /* Size of problem      */
    PetscInt     rank;
    PetscInt     size;
} UserContext;
```

The objective function evaluation routine should look like:

```
PetscErrorCode MyFunction(TaoSolver tao, Vec x,
                          PetscReal *fcnval, void *userCtx){
    UserContext *user = (UserContext *)userCtx;
    ...
}
```

## TAO Applications

The routines for computing the gradient and Hessians look similar:

```
PetscErrorCode MyGradient(TaoSolver tao, Vec x, Vec g,
                          void *userCtx){
    UserContext *user = (UserContext *)userCtx;
    ...
}

PetscErrorCode MyHessian(TaoSolver tao, Vec x, Mat *H,
                        Mat *Hpre, MatStructure *flag, void *userCtx,
                        UserContext *user = (UserContext *)userCtx;
    ...
}
```

## Writing the Driver

A *driver* program is used to hook up the user's application to the TAO library. This driver performs the following steps:

- Create the TAO Solver
- Create the variable vector and Hessian matrix
- Hook up the user evaluation routines to the TAO solver
- Solve the application
- Analyze the solution

# TAO Applications

## Create the TAO Solver

```
TaoSolver      tao; /* TAO Optimization solver      */
UserContext    user; /* user-defined structure      */
Vec            x;   /* solution vector          */
Mat            H;   /* Hessian Matrix          */

PetscInitialize(&argc,&argv,0,0);
TaoInitialize(&argc,&argv,0,0);
TaoCreate(PETSC_COMM_SELF,&tao);
TaoSetType(tao,"tao_lmvm");

...
```

# TAO Applications

Create storage for the solution vector and Hessian matrix

```
TaoSolver      tao; /* TAO Optimization solver      */
UserContext    user; /* user-defined structure      */
Vec            x;   /* solution vector            */
Mat            H;   /* Hessian Matrix             */
```

...

```
VecCreateSeq(PETSC_COMM_SELF,n,&x);
MatCreateSeqAIJ(PETSC_COMM_SELF,n,n,nz,PETSC_NULL,&H);
```

...

## TAO Applications

Hook up the application to TAO

```
TaoSolver      tao; /* TAO Optimization solver      */
UserContext    user; /* user-defined structure      */
Vec            x;   /* solution vector            */
Mat            H;   /* Hessian Matrix             */

...
user.epsilon = 0.1;
TaoSetInitialVector(tao,x);
TaoSetObjectiveRoutine(tao,MyFunction,(void *)&user);
TaoSetGradientRoutine(tao,MyGradient,(void *)&user);
TaoSetHessianRoutine(tao,H,H,MyHessian,(void *)&user);
...
```

# TAO Applications

Solve the application

```
TaoSolver      tao; /* TAO Optimization solver      */
UserContext    user; /* user-defined structure      */
Vec            x;   /* solution vector            */
Mat            H;   /* Hessian Matrix             */

...
TaoSetFromOptions(tao);
TaoSolve(tao);
VecView(x,PETSC_VIEWER_STDOUT_SELF);
```

# TAO Applications

Solve a multiple processor application

The most important and difficult part of solving a multiple processor application is writing the function, gradient, and Hessian evaluation routines to run in parallel.

Once that is done, it is trivial to get TAO to run in parallel:

```
...  
TaoCreate(PETSC_COMM_WORLD,&tao);  
VecCreateMPI(PETSC_COMM_WORLD,n,&x);  
MatCreateMPIAIJ(PETSC_COMM_WORLD,n,n,nz,PETSC_NULL,&H);  
...
```

# Toolkit for Advanced Optimization

- You can download TAO 2.1 from the webpage <http://www.mcs.anl.gov/tao>
- The documentation online includes installation instructions, a user's manual and a man page for every TAO function.
- The download includes several examples for using TAO in C and Fortran. We will use some of these examples in the tutorial.