Continuous Optimization and TAO

J. Moré, T. Munson, and J. Sarich

Mathematics and Computer Science Division, Argonne

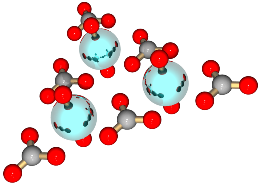
August 18, 2006

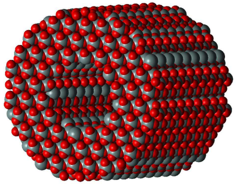
Outline

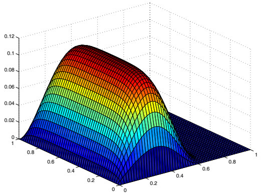
- Introduction
- Unconstrained optimization
    - Model-based methods
    - Limited-memory variable metric methods
    - Newton's method
- Automatic Differentiation
- Solving optimization problems with TAO

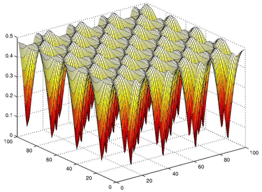
Nonlinearly Constrained Optimization

$$\min\{f(x) : x_l \le x \le x_u,\ c_l \le c(x) \le c_u\}$$







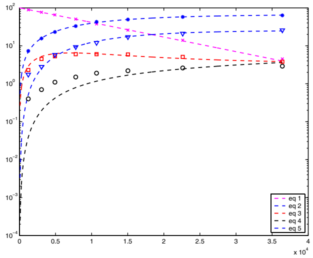


Isomerization of α-pinene

Determine the reaction coefficients in the thermal isomerization of α-pinene from measurements $z_1, \ldots z_8$ by minimizing

$$\sum_{j=1}^{8} \|y(\tau_j;\theta) - z_j\|^2$$

$$\begin{aligned}
y_1' &= -(\theta_1 + \theta_2) y_1 \\
y_2' &= \theta_1 y_1 \\
y_3' &= \theta_2 y_1 - (\theta_3 + \theta_4) y_3 + \theta_5 y_5 \\
y_4' &= \theta_3 y_3 \\
y_5' &= \theta_4 y_3 - \theta_5 y_5
\end{aligned}$$



Classification of Constrained Optimization Problems

$$\min \{ f(x) : x_l \le x \le x_u,\ c_l \le c(x) \le c_u \}$$

- Number of variables n
- Number of constraints m
- Number of linear constraints
- Number of equality constraints n_e
- Number of degrees of freedom $n - n_e$
- Sparsity of $c'(x) = (\partial_i c_j(x))$
- Sparsity of $\nabla_x^2 \mathcal{L}(x, \lambda) = \nabla^2 f(x) + \sum_{k=1}^{m} \nabla^2 c_k(x) \lambda_k$

Classification of Constrained Optimization Software

- Formulation
- Interfaces: MATLAB, AMPL, GAMS
- Second-order information options:
  - Differences
  - Limited memory
  - Hessian-vector products
- Linear solvers
  - Direct solvers
  - Iterative solvers
  - Preconditioners
- Partially separable problem formulation
- Documentation
- License

Unconstrained Optimization: Background

Given a continuously differentiable $f : \mathbb{R}^n \mapsto \mathbb{R}$ and

$$\min \{ f(x) : x \in \mathbb{R}^n \}$$

generate a sequence of iterates $\{x_k\}$ such that the gradient test

$$\| \nabla f(x_k) \| \le \tau$$

is eventually satisfied

Theorem. If $f : \mathbb{R}^n \mapsto \mathbb{R}$ is continuously differentiable and bounded below, then there is a sequence $\{x_k\}$ such that

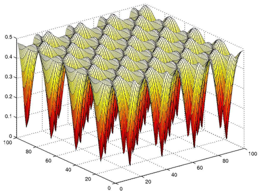
$$\lim_{k \to \infty} \| \nabla f(x_k) \| = 0.$$

Ginzburg-Landau Model

Minimize the Gibbs free energy for a homogeneous superconductor

$$\int_{\mathcal{D}} \left\{ -|v(x)|^2 + \tfrac{1}{2}|v(x)|^4 + \left\| [\nabla - iA(x)]\, v(x) \right\|^2 + \kappa^2 \left\| (\nabla \times A)(x) \right\|^2 \right\} dx$$

$v : \mathbb{R}^2 \to \mathbb{C}$ (order parameter)
$A : \mathbb{R}^2 \to \mathbb{R}^2$ (vector potential)



Unconstrained problem with a non-convex objective function. The Hessian matrix is singular, but has a unique minimizer and saddle points.

Unconstrained Optimization

What can I use if the gradient $\nabla f(x)$ is not available?

- Geometry-based methods: Pattern search, Nelder-Mead, ...
- Model-based methods: Quadratic, radial-basis models, ...

What can I use if the gradient $\nabla f(x)$ is available?

- Conjugate gradient methods
- Limited-memory variable metric methods
- Variable metric methods

What can I use if the gradient $\nabla f(x)$ and Hessian $\nabla^2 f(x)$ are available?

- Newton's method with a trust region or line search

Quadratic Model-Based Methods

Question: How do we find a minimizer of $f : \mathbb{R}^n \mapsto \mathbb{R}$ if we are not able to compute the gradient?

At each iterations we have m points $x_1, \ldots, x_m$, and we construct a quadratic q that interpolates f at each point, that is,

$$q(x_k) = f(x_k), \qquad 1 \le k \le m.$$

We also require that the Hessian approximation B be such that

$$\min \{ \|B - B_0\|_F : q(x_k) = f(x_k),\ 1 \le k \le m \}$$

where B_0 is the Hessian approximation obtained on the previous iteration.

Quadratic Model-Based Methods

If x_0 is the current approximation to the minimizer, then the next iterate is determined by solving the trust region subproblem

$$\min \{ q(x_0 + w) : \|w\| \le \Delta \}$$

and setting $x_+ = x_0 + w$.

Research Issues

- How do we compute the quadratic q?
- How do we compute the initial set of points $x_1, \ldots, x_m$?
- How do we update the basis points $x_1, \ldots, x_m$?

Line Search Methods

A sequence of iterates $\{x_k\}$ is generated via

$$x_{k+1} = x_k + \alpha_k p_k,$$

where p_k is a descent direction at x_k, that is,

$$\nabla f(x_k)^T p_k < 0,$$

and α_k is determined by a line search along p_k.

Line searches

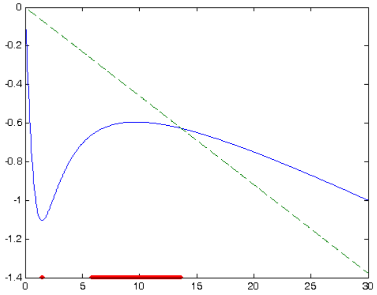
- Geometry-based: Armijo, . . .
- Model-based: Quadratics, cubics, . . .

Powell-Wolfe Conditions on the Line Search

Given $0 \le \mu < \eta \le 1$, require that

$$f(x + \alpha p) \le f(x) + \mu\, \alpha \nabla f(x_k)^T p_k \qquad \text{sufficent decrease}$$

$$|\nabla f(x + \alpha p)^T p| \le \eta\, |\nabla f(x)^T p| \qquad \text{curvature condition}$$



Conjugate Gradient Algorithms

Given a starting vector x_0 generate iterates via

$$x_{k+1} = x_k + \alpha_k p_k$$
$$p_{k+1} = -\nabla f(x_k) + \beta_k p_k$$

where α_k is determined by a line search.

Three reasonable choices of β_k are ($g_k = \nabla f(x_k)$):

$$\beta_k^{FR} = \left(\frac{\|g_{k+1}\|}{\|g_k\|} \right)^2, \qquad \text{Fletcher-Reeves}$$
$$\beta_k^{PR} = \frac{\langle g_{k+1}, g_{k+1} - g_k \rangle}{\|g_k\|^2}, \qquad \text{Polak-Rivière}$$
$$\beta_k^{PR+} = \max \left\{ \beta_k^{PR}, 0 \right\}, \qquad \text{PR-plus}$$

Limited-Memory Variable-Metric Algorithms

Given a starting vector x_0 generate iterates via

$$x_{k+1} = x_k - \alpha_k H_k \nabla f(x_k)$$

where α_k is determined by a line search.

The matrix H_k is defined in terms of information gathered during the previous m iterations.

- H_k is positive definite.
- Storage of H_k requires $2mn$ locations.
- Computation of $H_k \nabla f(x_k)$ costs $(8m + 1)n$ flops.

Limited-Memory Algorithms: Updating H_k

$$H_{k+1} = \left(I - \frac{s_k y_k^T}{\rho_k} \right) H_k \left(I - \frac{y_k s_k^T}{\rho_k} \right) + \frac{s_k s_k^T}{\rho_k},$$

$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k), \qquad s_k = x_{k+1} - x_k, \qquad \rho_k = y_k^T s_k$$

Store information from the last m iterations

$$y_1, \ldots, y_m,$$

$$s_1, \ldots, s_m,$$

$$\rho_1, \ldots, \rho_m$$

How can we compute $H_{m+1} w$?

Limited-Memory Algorithms: Computing $H_{m+1}w$

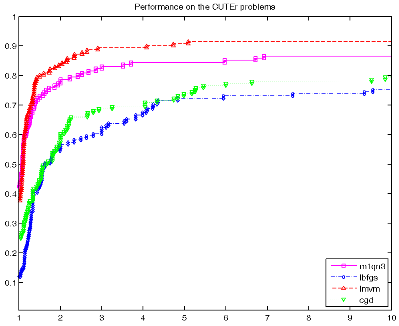
Recursion for $q_i = V_i^T q_{i+1}$

```
$q_{m+1} = w$
do i = m, . . . , 1
    $\beta_i = (s_i^T q_{i+1})/\rho_i$
    $q_i = q_{i+1} - \beta_i y_i$
end do
```

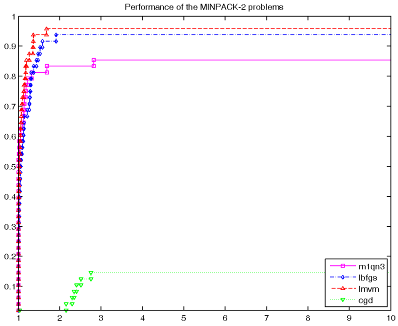
Recursion for $r_i = H_i q_i$

```
do i = 1, . . . , m
    $r_{i+1} = r_i + s_i \left(\beta_i - (y_i^T r_i)/\rho_i\right)$
end do
$r_{m+1} = H_{m+1} w$
```

Performance



CUTEr



MINPACK-2

Trust-Region Newton Algorithm

At each iteration the step s_k (approximately) minimizes

$$\min \{ q_k(x_k + s) : \|s\| \leq \Delta_k \}$$

where q_k is the quadratic approximation,

$$q_k(w) = \langle \nabla f(x_k), w \rangle + \tfrac{1}{2} \langle w, \nabla^2 f(x_k) w \rangle,$$

to the function, and Δ_k is the trust-region bound.

The trust-region subproblem solved with preconditioned Steihaug-Toint conjugate gradient method.

Recommendations

But what algorithm should I use?

- If the gradient $\nabla f(x)$ is not available, then a model-based method is a reasonable choice. Methods based on quadratic interpolation are currently the best choice.
- If the gradient $\nabla f(x)$ is available, then a limited-memory variable metric method is likely to produce an approximate minimizer in the least number of gradient evaluations.
- If the Hessian is also available, then a state-of-the-art implementation of Newton's method is likely to produce the best results if the problem is large and sparse.

Computing the Gradient

Hand-coded gradients

- Generally efficient
- Error prone
- The cost is usually less than 5 function evaluations

Difference approximations

$$\partial_i f(x) \approx \frac{f(x + he_i) - f(x)}{h_i}$$

- Choice of h_i may be problematic in the presence of noise.
- Costs n function evaluations
- Accuracy is about the $\varepsilon_f^{1/2}$ where ε_f is the noise level of f

Inexpensive Gradient via Automatic Differentiation

Code generated by automatic differentiation tools

- Accurate to full precision
- For the reverse mode the cost is $\Omega_T \, T\{f(x)\}$.
- In theory, $\Omega_T \le 5$.
- For the reverse mode the memory is proportional to the number of intermediate variables.

TAO: Toolkit for Advanced Optimization

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

- Object-oriented techniques
- Component-based interaction
- Leverage of existing parallel computing infrastructure
- Reuse of external toolkits (linear solvers, preconditioners, . . .)

TAO Status

- Version 1.9 (December 2006)
- Source code, documentation, tutorials, example problems, . . .
- TAO components: MPQC (Sandia) and NWChem (PNNL)
- Grid sequencing via Distributed Arrays (PETSc)
- Gradients of grid functions via ADIC

Powered by PETSc and ADIC!

Using TAO with PETSc

```
TAO_SOLVER      tao;             /* TAO Optimization solver         */
TAO_APPLICATION app;             /* TAO Application using PETSc     */
AppCtx          user;            /* User-defined application context */
Vec             x;               /* Solution vector                 */
Mat             H;               /* Hessian Matrix                  */

VecCreateSeq(PETSC_COMM_SELF,n,&x);
MatCreateSeqAIJ(PETSC_COMM_SELF,n,n,nz,PETSC_NULL,&H);
TaoCreate(PETSC_COMM_SELF,''tao_lmvm'',&tao);
TaoApplicationCreate(PETSC_COMM_SELF,&app);
TaoAppSetInitialSolutionVec(app,x);
TaoAppSetObjectiveRoutine(app, FormFunction,(void *)&user);
TaoAppSetGradientRoutine(app,FormGradient,(void *)&user);
TaoAppSetHessianMat(app,H,H);
TaoAppSetHessianRoutine(app,FormHessian,(void *)&user);
TaoSolveApplication(app,tao);
VecView(x,PETSC_VIEWER_STDOUT_SELF);
```

Objective Function and Gradient Evaluation

```
typedef struct {          /* Used in the minimum surface area problem */
  int          mx, my;              /* discretization in x, y directions */
  int          bmx, bmy, bheight;                /* The size of the plate */
  double       bheight;                        /* The height of the plate */
  double      *bottom, *top, *left, *right;            /* boundary values */
} AppCtx;

int FormFunction(TAO_APPLICATION app, Vec x, double *fcn, void *userCtx){
   AppCtx *user = (AppCtx *)userCtx;
   ...
}
int FormGradient(TAO_APPLICATION app, Vec x, Vec g, void *userCtx){
   AppCtx *user = (AppCtx *)userCtx;
   ...
}
int FormHessian(TAO_APPLICATION app, Vec x, Mat *H, Mat *H, int *flag,
                void *userCtx){
   AppCtx *user = (AppCtx *)userCtx;
   ...
}
```

Creating and Using a TAO Application

```
TAO_SOLVER      tao;                /* TAO Optimization solver        */
TAO_APPLICATION app;                /* TAO Application using PETSc    */
AppCtx          user;               /* User-defined application context */
Vec             x;                  /* Solution vector                */
Mat             H;                  /* Hessian Matrix                 */

VecCreateSeq(PETSC_COMM_SELF,n,&x);
MatCreateSeqAIJ(PETSC_COMM_SELF,n,n,nz,PETSC_NULL,&H);
TaoCreate(PETSC_COMM_SELF,"tao_lmvm",&tao);
TaoApplicationCreate(PETSC_COMM_SELF,&app);
TaoAppSetInitialSolutionVec(app,x);
TaoAppSetObjectiveRoutine(app,FormFunction,(void *)&user);
TaoAppSetGradientRoutine(app,FormGradient,(void *)&user);
TaoAppSetHessianMat(app,H,H);
TaoAppSetHessianRoutine(app,FormHessian,(void *)&user);
TaoSolveApplication(app,tao);
VecView(x,PETSC_VIEWER_STDOUT_SELF);
```

Creating and Using a TAO Solver

```
TAO_SOLVER      tao;                 /* TAO Optimization solver          */
TAO_APPLICATION app;                 /* TAO Application using PETSc      */
AppCtx          user;                /* User-defined application context */
Vec             x;                   /* Solution vector                  */
Mat             H;                   /* Hessian Matrix                   */

VecCreateSeq(PETSC_COMM_SELF,n,&x);
MatCreateSeqAIJ(PETSC_COMM_SELF,n,n,nz,PETSC_NULL,&H);
TaoCreate(PETSC_COMM_SELF,"tao_lmvm",&tao);
TaoApplicationCreate(PETSC_COMM_SELF,&app);
TaoAppSetInitialSolutionVec(app,x);
TaoAppSetObjectiveRoutine(app,FormFunction,(void *)&user);
TaoAppSetGradientRoutine(app,FormGradient,(void *)&user);
TaoAppSetHessianMat(app,H,H);
TaoAppSetHessianRoutine(app,FormHessian,(void *)&user);
TaoSolveApplication(app,tao);
VecView(x,PETSC_VIEWER_STDOUT_SELF);
```

TAO Program Outline

```
TAO_SOLVER      tao;              /* TAO Optimization solver         */
TAO_APPLICATION app;              /* TAO Application using PETSc     */
AppCtx          user;             /* User-defined application context */
Vec             x;                /* Solution vector                 */
Mat             H;                /* Hessian Matrix                  */

VecCreateSeq(PETSC_COMM_SELF,n,&x);
MatCreateSeqAIJ(PETSC_COMM_SELF,n,n,nz,PETSC_NULL,&H);
TaoCreate(PETSC_COMM_SELF,"tao_lmvm",&tao);
TaoApplicationCreate(PETSC_COMM_SELF,&app);
TaoAppSetInitialSolutionVec(app,x);
TaoAppSetObjectiveRoutine(app,FormFunction,(void *)&user);
TaoAppSetGradientRoutine(app,FormGradient,(void *)&user);
TaoAppSetHessianMat(app,H,H);
TaoAppSetHessianRoutine(app,FormHessian,(void *)&user);
TaoSolveApplication(app,tao);
VecView(x,PETSC_VIEWER_STDOUT_SELF);
```

Using PETSc Objects on Multiple Processors

```
TAO_SOLVER      tao;                 /* TAO Optimization solver         */
TAO_APPLICATION app;                 /* TAO Application using PETSc     */
AppCtx          user;                /* user-defined application context */
Vec             x;                   /* solution vector                 */
Mat             H;                   /* Hessian Matrix                  */

VecCreateMPI(PETSC_COMM_WORLD,n,&x);
MatCreateMPIAIJ(PETSC_COMM_WORLD,nlocal,nlocal,n,n,d_nz,d_nnz,o_nz,o_nnz,&H);
TaoCreate(PETSC_COMM_WORLD,"tao_lmvm",&tao);
TaoApplicationCreate(PETSC_COMM_WORLD,&app);
TaoAppSetInitialSolutionVec(app,x);
TaoAppSetObjectiveRoutine(app,FormFunction,(void *)&user);
TaoAppSetGradientRoutine(app,FormGradient,(void *)&user);
TaoAppSetHessianMat(app,H,H);
TaoAppSetHessianRoutine(app,FormHessian,(void *)&user);
TaoSolveApplication(app,tao);
VecView(x,PETSC_VIEWER_STDOUT_WORLD);
```

Convergence Tolerances

Absolute tolerances specify acceptable errors in the optimality of the function and the constraints.

$$f(x) \leq f(x^*) + \epsilon_{fatol}$$

Relative tolerances specify the number of significant digits required in the solution and the constraints.

$$f(x) \leq f(x^*) + \epsilon_{frtol} |f(x^*)|$$

These tolerance can be changed

```
int TaoSetTolerances(TAO_SOLVER solver,double fatol,double frtol,
                                       double catol,double crtol)
```

TAO Basic Facilities

- TaoAppSetInitialSolutionVec
- TaoAppSetVariableBounds
- TaoGetLinearSolver
- TaoFromOptions
- TaoAppSetMonitor
- TaoView
- . . .