Optimizers, Hessians, and Other Dangers

Benjamin S. Skrainka
University College London

July 31, 2009

Overview

We focus on how to get the most out of your optimizer(s):

1. Scaling
2. Initial Guess
3. Solver Options
4. Gradients & Hessians
5. Dangers with Hessians
6. Validation
7. Diagnosing Problems
8. Ipopt

Scaling

Scaling can help solve convergence problems:

- Naive scaling: scale variables so their magnitudes are ~ 1
- Better: scale variables so solution has magnitude ~ 1
- A good solver may automatically scale the problem

Computing an Initial Guess

Computing a good initial guess is crucial:

- To avoid bad regions in parameter space
- To facilitate convergence
- Possible methods:
  - Use a simpler but consistent estimator such as OLS
  - Estimate a restricted version of the problem
  - Use Nelder-Mead or other derivative-free method (beware of `fminsearch`)
  - Use Quasi-Monte Carlo search
- Beware: the optimizer may only find a local max!

Explore Your Objective Function

Visualizing your objective function will help you:

- Catch mistakes
- Choose an initial guess
- Determine if variable transformations, such as log or $x' = 1/x$, are helpful

Some tools:

- Plot objective function while holding all variables except one fixed
- Explore points near and far from the expected solution
- Contour plots may also be helpful
- Hopefully, your function is convex...

Solver Options

A state of the art optimizer such as knitro is highly tunable:

- You should configure the options to suit your problem: scale, linear or non-linear, concavity, constraints, etc.
- Experimentation is required:
  - Algorithm: Interior/CG, Interior/Direct, Active Set
  - Barrier parameters: `bar_murule`, `bar_feasible`
  - Tolerances: X, function, constraints
  - Diagnostics
- See Nocedal & Wright for the gory details of how optimizers work

Which Algorithm?

Different algorithms work better on different problems:

Interior/CG

- Direct step is poor quality
- There is negative curvature
- Large or dense Hessian

Interior/Direct

- Ill-conditioned Hessian of Lagrangian
- Large or dense Hessian
- Dependent or degenerate constraints

Active Set

- Small and medium scale problems
- You can choose a (good) initial guess

The default is that knitro chooses the algorithm.

$\Rightarrow$ There are no hard rules. You must experiment!!!

Knitro Configuration

Knitro is highly configurable:

- Set options via:
  - C, C++, FORTRAN, or Java API
  - MATLAB options file
- Documentation in ${KNITRO_DIR}/Knitro60_UserManual.pdf
- Example options file in ${KNITRO_DIR}/examples/Matlab/knitro.opt

Calling Knitro From MATLAB

To call Knitro from MATLAB:

1. Follow steps in InstallGuide.pdf I sent out
2. Call `ktrlink`:

```
% Call Knitro
[ xOpt, fval, exitflag, output, lambda ] = ktrlink( ...
  @(xFree) myLogLikelihood( xFree, myData ),       ...
  xFree, [], [], [], [], lb, ub, [], [], 'knitro.opt' )
% Check exit flag
if exitflag <= -100 | exitflag >= -199
  % Success
end
```

- Note: older versions of Knitro modify `fmincon` to call `ktrlink`
- Best to pass options via a file such as '`knitro.opt`'

Listing 1: knitro.opt Options File

```
# KNITRO 6.0.0 Options file
# http://ziena.com/documentation.html

# Which algorithm to use.
#   auto   = 0 = let KNITRO choose the algorithm
#   direct = 1 = use Interior (barrier) Direct algorithm
#   cg     = 2 = use Interior (barrier) CG algorithm
#   active = 3 = use Active Set algorithm
algorithm    0

# Whether feasibility is given special emphasis.
#   no       = 0 = no emphasis on feasibility
#   stay     = 1 = iterates must honor inequalities
#   get      = 2 = emphasize first getting feasible before optimiz
#   get_stay = 3 = implement both options 1 and 2 above
bar_feasible no

# Which barrier parameter update strategy.
#   auto     = 0 = let KNITRO choose the strategy
#   monotone = 1
#   adaptive = 2
#   probing  = 3
#   dampmpc  = 4
#   fullmpc  = 5
#   quality  = 6
bar_murule   auto
```

```
# Initial trust region radius scaling factor, used to determine
# the initial trust region size.
delta        1

# Specifies the final relative stopping tolerance for the feasibil
# error. Smaller values of feastol result in a higher degree of ac
# in the solution with respect to feasibility.
feastol      1e-06

# How to compute/approximate the gradient of the objective
# and constraint functions.
#   exact        = 1 = user supplies exact first derivatives
#   forward      = 2 = gradients computed by forward finite differ
#   central      = 3 = gradients computed by central finite differ
gradopt      exact

# How to compute/approximate the Hessian of the Lagrangian.
#   exact        = 1 = user supplies exact second derivatives
#   bfgs         = 2 = KNITRO computes a dense quasi-Newton BFGS H
#   sr1          = 3 = KNITRO computes a dense quasi-Newton SR1 H
#   finite_diff  = 4 = KNITRO computes Hessian-vector products by
#   product      = 5 = user supplies exact Hessian-vector products
#   lbfgs        = 6 = KNITRO computes a limited-memory quasi-Newt
hessopt      exact
```

```
# Whether to enforce satisfaction of simple bounds at all iteratio
#   no      = 0 = allow iterations to violate the bounds
#   always  = 1 = enforce bounds satisfaction of all iterates
#   initpt  = 2 = enforce bounds satisfaction of initial point
honorbnds    initpt

# Maximum number of iterations to allow
# (if 0 then KNITRO determines the best value).
# Default values are 10000 for NLP and 3000 for MIP.
maxit        0

# Maximum allowable CPU time in seconds.
# If multistart is active, this limits time spent on one start poi
1e+08

# Specifies the final relative stopping tolerance for the KKT (opt
# error. Smaller values of opttol result in a higher degree of acc
# the solution with respect to optimality.
opttol       1e-06

# Step size tolerance used for terminating the optimization.
xtol         1e-15  # Should be sqrt( machine epsilon )
```

Numerical Gradients and Hessians Overview

Gradients and Hessians are often quite important:

- Choosing direction and step for Gaussian methods
- Evaluating convergence/non-convergence
- Estimating the information matrix (MLE)
- Note:
  - Solvers need accurate gradients to converge correctly
  - Solvers do not need precise Hessians
  - But, the information matrix does require accurate computation
- Consequently, quick and accurate evaluation is important:
  - Hand-coded, analytic gradient/Hessian
  - Automatic differentiation
  - Numerical gradient/Hessian

Forward Finite Difference Gradient

```
function [ fgrad ] = NumGrad( hFunc, x0, xTol )
  x1    = x0 + xTol ;
  f1    = feval( hFunc, x1 ) ;
  f0    = feval( hFunc, x0) ;
  fgrad = ( f1 - f0 ) / ( x1 - x0 ) ;
```

Centered Finite Difference Gradient

```
function [ fgrad ] = NumGrad( hFunc, x0, xTol )
  x1 = x0 + xTol ;
  x2 = 2 * x0 - x1 ;
  f2 = feval( hFunc, x2 ) ;
  fgrad = ( f1 - f2 ) / ( x1 - x2 ) ;
```

Complex Step Differentiation

Better to use CSD whose error is $O(h^2)$:

```
function [ vCSDGrad ] = CSDGrad( func, x0, dwStep )
  nParams = length( x0 ) ;
  vCSDGrad = zeros( nParams, 1 ) ;
  if nargin < 3
    dx = 1e-5 ;
  else
    dx = dwStep ;
  end
  xPlus = x0 + 1i * dx ;
  for ix = 1 : nParams
    x1 = x0 ;
    x1( ix ) = xPlus( ix ) ;
    [ fval ] = func( x1 ) ;
    vCSDGrad( ix ) = imag( fval / dx ) ;
  end
```

CSD vs. FD vs. Analytic

The official word from Paul Hovland (Mr. AD):

- AD or analytic derivatives:
    - 'Best'
    - Hand-coding is error-prone
    - AD doesn't work (well) with all platforms and functional forms
- CSD
    - Very accurate results, especially with $h \approx 1e - 20$ or $1e - 30$ because error $\sim O\left(h^2\right)$
    - Cost $\geq$ FD
    - Some FORTRAN and MATLAB functions don't work correctly
- FD: 'Idiotic' – Munson

CSD Hessian

```
function [ fdHess ] = CSDHessian( func, x0, dwStep )
  nParams = length( x0 ) ;
  fdHess = zeros( nParams ) ;
  for ix = 1 : nParams
    xImagStep = x0 ;
    xImagStep( ix ) = x0( ix ) + 1i * dwStep ;
    for jx = ix : nParams
      xLeft = xImagStep ;
      xLeft( jx ) = xLeft( jx ) - dwStep ;
      xRight = xImagStep ;
      xRight( jx ) = xRight( jx ) + dwStep ;
      vLeftGrad = func( xLeft ) ;
      vRightGrad = func( xRight ) ;
      fdHess( ix, jx ) = imag( ( vRightGrad ...
           - vLeftGrad ) / ( 2 * dwStep^2 ) ) ;
      fdHess( jx, ix ) = fdHess( ix, jx ) ;
    end
  end
```

Overview of Hessian Pitfalls

'The only way to do a Hessian is to do a Hessian' – Ken Judd

- The 'Hessian' returned by `fmincon` is not a Hessian:
  - Computed by BFGS, sr1, or some other approximation scheme
  - A rank 1 update of the identity matrix
  - Requires at least as many iterations as the size of the problem
  - Dependent on quality of initial guess, $x0$
  - Often built with convexity restriction
- Therefore, you must compute the Hessian either numerically or analytically
- `fmincon`'s 'Hessian' often differs considerably from the true Hessian – just check eigenvalues or condition number

Condition Number

Use the condition number to evaluate the stability of your problem:

- $\texttt{cond}\,(A) = \dfrac{\max\,[\texttt{eig}\,(A)]}{\min\,[\texttt{eig}\,(A)]}$
- Large values $\Rightarrow$ trouble
- Also check eigenvalues: negative or nearly zero eigenvalues $\Rightarrow$ problem is not concave
- If the Hessian is not full rank, parameters will not be identified

Estimating the Information Matrix

To estimate the information matrix:

1. Calculate the Hessian – either analytically or numerically
2. Invert the Hessian
3. Calculate standard errors

```
StandardErrors = sqrt( diag( inv( YourHessian ) ) ) ;
```

Assuming, of course, that your objective function is the likelihood...

Validation

Validating your results is a crucial part of the scientific method:

- Generate a Monte Carlo data set: does your estimation code recover the target parameters?
- *T*est *D*riven *D*evelopment:
  1. Develop a unit test (code to exercise your function)
  2. Write your function
  3. Validate function behaves correctly for all execution paths
  4. The sooner you find a bug, the cheaper it is to fix!!!
- Start simple: e.g. logit with linear utility
- Then slowly add features one at a time, such as interactions or non-linearities
- Validate results via Monte Carlo
- Or, feed it a simple problem with an analytical solution

Diagnosing Problems

Solvers provide a lot of information to determine why your problem can't be solved:

- Exit codes
- Diagnositic Output

Exit Codes

It is crucial that you check the optimizer's exit code and the gradient and Hessian of the objective function:

- Optimizer may not have converged:
  - Exceeded CPU time
  - Exceeded maximum number of iterations
- Optimizer may not have found a global max
- Constraints may bind when they shouldn't ($\lambda \neq 0$)
- Failure to check exit flags could lead to public humiliation and flogging

Diagnosing Problems

The solver provides information about its progress which can be used to diagnose problems:

- Enable diagnostic output
- The meaning of output depends on the type of solver: Interior Point, Active Set, etc.
- In general, you must RTM: each solver is different

Interpreting Solver Output

Things to look for:

- Residual should decrease geometrically towards the end (Gaussian)
  - Then solver has converged
  - Geometric decrease follwed by wandering around:
    - At limit of numerical precision
    - Increase precision and check scaling
- Linear convergence:
  - $\|residual\| \to 0$: rank deficient Jacobian $\Rightarrow$ lack of identification
  - Far from solution $\Rightarrow$ convergence to local min of $\|residual\|$
- Check values of Lagrange multipliers:
  - `lambda.{ upper, lower, ineqlin, eqlin, ineqnonlin, eqnonlin }`
  - Local min of constraint $\Rightarrow$ infeasible or locally inconsistent (IP)
  - Non convergence: failure of constraint qualification (NLP)
- Unbounded: λ or $x \to \pm\infty$

Ipopt

Ipopt is an alternative optimizer which you can use:

- Interior point algorithm
- Part of the COIN-OR collection of free optimization packages
- Supports C, C++, FORTRAN, AMPL, Java, and MATLAB
- Can be difficult to build – see me for details
- www.coin-or.org
- COIN-OR provides free software to facilitate optimization research