Optimization with R –Tips and Tricks

Hans W Borchers, DHBW Mannheim

R User Group Meeting, Köln, September 2017

Introduction

Optimization

"optimization : an act, process, or methodology of making something (such as a design, system, or decision) as fully perfect, functional, or effective as possible;
specifically: the mathematical procedures (such as finding the maximum of a function) involved in this."

– Merriam-Webster Online Dictionary, 2017 (*)

Forms of optimization (cf. Netspeak: "? optimization"):

- Code / program / system optimization
- Search / website / server ... optimization
- Business / process / chain ... optimization
- Engine / design / production optimization

(*) First Known Use: 1857

Mathematical Optimization

A mathematical optimization problem consists of maximizing (or minimizing) a real objective function on a defined domain:

Given a set $A \subseteq R^n$ and a function $f : A \rightarrow R$ from A to the real numbers, find an element $x_0 \in A$ such that $f(x_0) \leq f(x)$ for all x in an *environment* of x_0.

Typical problems:

- finding an optimum will be computationally expensive
- different types of objective functions and domains
- need to compute the optimum with very high accuracy
- need to find a global optimum, restricted resources
- etc.

Classification of Optimization Tasks

- Unconstrained optimization
- Nonlinear least-squares fitting (parameter estimation)
- Optimization with constraints
- Non-smooth optimization (e.g., minimax problems)
- Global optimization (stochastic programming)
- Linear and quadratic programming (LP, QP)
- Convex optimization (resp. SOCP, SDP)
- Mixed-integer programming (MIP, MILP, MINLP)
- Combinatorial optimization (e.g., graph problems)

100+ Packages on the Optimization TV

adagio alabama BB boot bvls cccp cec2005benchmark cec2013 CEoptim clpAPI CLSOCP clue cmaes cmaesr copulaedas cplexAPI crs dclone DEoptim DEoptimR desirability dfoptim ECOSolveR GA genalg GenSA globalOptTests glpkAPI goalprog GrassmannOptim gsl hydroPSO igraph irace isotone kernlab kofnGA lbfgs lbfgsb3 limSolve linprog localsolver LowRankQP lpSolve lpSolveAPI matchingMarkets matchingR maxLik mcga mco minpack.lm minqa neldermead NlcOptim nleqslv nlmrt nloptr nls2 NMOF nnls onls optimx optmatch parma powell pso psoptim qap quadprog quantreg rcdd RCEIM Rcgmin rCMA Rcplex RcppDE Rcsdp Rdsdp rgenoud Rglpk rLindo Rmalschains Rmosek rneos ROI Rsolnp Rsymphony Rvmmin scs smoof sna soma subplex tabuSearch trust trustOptim TSP ucminf

Optimization in Statistics

- Maximum Likelihood
- Parameter estimation
- Quantile and density estimation
- LASSO estimation
- Robust regression
- Nonlinear equations
- Geometric programming problems
- Deep Learning / Support Vector Machines
- Engineering and Design, e.g. optimal control
- Operations Research, e.g. network flow problems
- Economics, e.g. portfolio optimization

Goals for this Talk

- Overview of (large, rapidly changing, still incomplete) set of tools for solving optimization problems in R
- Appreciation of the types of problems and types of methods to solve them
- Advice on setting up problems and solvers
- Suggestions for interpreting results
- Some almost real-world examples

Unfortunately, there is no time to talk about the new and exciting developments in *convex optimization* and optimization *modelling languages*.

Unconstrained Optimization

Univariate (1-dim.) Minimization

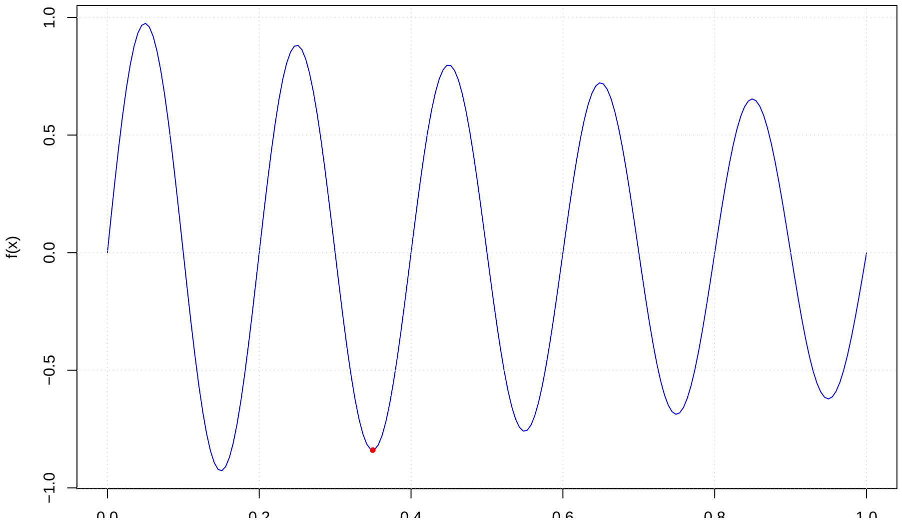
```
optimize(f = , interval = ,  ..., lower = min(interval),
  upper = max(interval), maximum = FALSE,
  tol = .Machine$double.eps^0.25)

optim(par = , fn = , gr = NULL, ...,
  method = "Brent",
  lower = -Inf, upper = Inf)

optimizeR(f, lower, upper, ..., tol = 1e-20,
  method = c("Brent", "GoldenRatio"),
  maximum = FALSE,
  precFactor = 2.0, precBits = -log2(tol) * precFactor,
  maxiter = 1000, trace = FALSE)
```

1-dimensional Example

```r
f <- function(x) exp(-0.5*x) * sin(10*pi*x)
curve(f, 0, 1, n = 200, col=4); grid()
opt <- optimize(f, c(0, 1))
points(opt$minimum, opt$objective, pch = 20, col = 2)
```



optim() and Friends

```r
optim(par, fn, gr = NULL, ...,
  method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B",
             "SANN", "Brent"),
  lower = -Inf, upper = Inf,
  control = list(), hessian = FALSE)
```

Methods / Algorithms:

- **Nelder-Mead** - downhill simplex method
- **BFGS** - “variable metric” quasi-Newton method
- **CG** - conjugate gradient method
- **L-BFGS-B** - Broyden-Fletcher-Goldfarb-Shannon
- **Brent** - univariate minimization, same as `optimize`
- **SANN** - Simulated Annealing [don’t use !]

Nelder-Mead

Nelder-Mead iteratively generates a sequence of simplices to approximate a minimal point.

At each iteration, the vertices of the simplex are ordered according to their objective function values and the simplex 'distorted' accordingly.

- **Sort** function values on simplex
- **Reflect** compute the reflection point
- **Expand** compute the expansion point
- **Contract** (outside | inside)
- **Shrink** the simplex

Stop when the simplex is small enough ('tolerance').

Nelder-Mead in Action

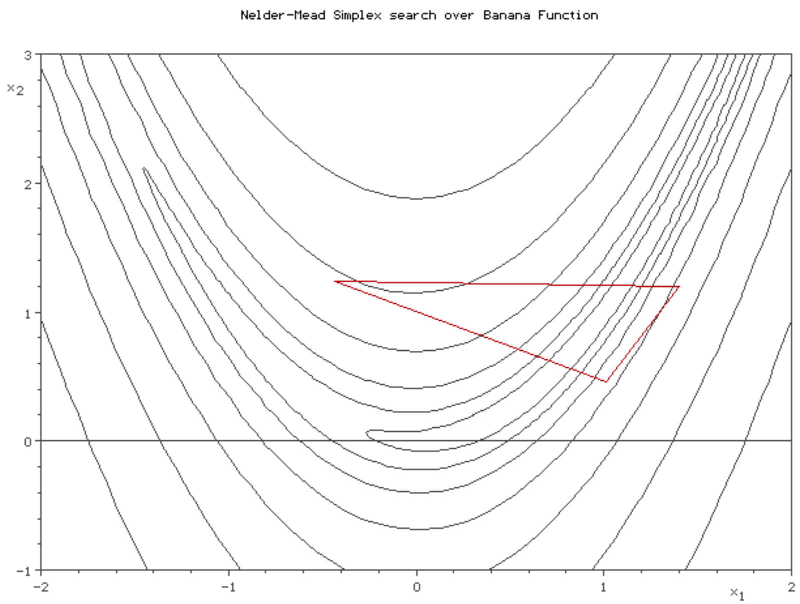


Figure 1: Source: de.wikipedia.org

Showcase Rosenbrock

As a showcase we use the *Rosenbrock function*, defined for $n \geq 2$. It has has a very flat valley leading to its minimal point.

$$f(x_1, \ldots, x_n) = \sum_{i=2}^{n} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$$

The global minimum obviously is $(1, \ldots, 1)$ with value 0.

```
fnRosenbrock <- function (x) {
    n <- length(x)
    x1 <- x[2:n]; x2 <- x[1:(n - 1)]
    sum(100 * (x1 - x2^2)^2 + (1 - x2)^2)
}
```

Available in package *adagio* as `fnRosenbrock()`,
with exact gradient `grRosenbrock()`.

`optim()` w/ Nelder-Mead

```
fn <- adagio::fnRosenbrock; gr <- adagio::grRosenbrock
sol <- optim(rep(0, 2), fn, gr, control=list(reltol=1e-12))
sol$par
```

```
## [1] 0.9999996 0.9999992
```

```
fn <- adagio::fnRosenbrock; gr <- adagio::grRosenbrock
sol <- optim(rep(0, 10), fn, gr,
             control=list(reltol=1e-12, maxit=10000))
sol$par; sol$counts
```

```
##  [1] 0.487650105 0.218747555 0.074772474 0.008069353 0.(
##  [6] 0.037545739 0.013695922 0.027284322 0.023147646 0.(
```

```
## function gradient
##     9707       NA
```

Nelder-Mead Solvers

- **dfoptim**

```
nmk(par, fn, control = list(), ...)
nmkb(par, fn, lower=-Inf, upper=Inf,
     control = list(), ...)
```

- **adagio**

```
neldermead(fn, x0, ..., adapt = TRUE,
           tol = 1e-10, maxfeval = 10000,
           step = rep(1.0, length(x0)))
```

- **pracma** [new]

```
anms(fn, x0, ...,
     tol = 1e-10, maxfeval = NULL)
```

Adaptive Nelder-Mead

anms in *pracma* implements a new (Gao and Han, 2012) adaptive Nelder-Mead algorithm, adapting to the size of the problem (i.e., dimension of the objective function).

```
fn <- adagio::fnRosenbrock
pracma::anms(fn, rep(0, 20), tol = 1e-12, maxfeval = 25000)
```

```
## $xmin
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##
## $fmin
## [1] 5.073655e-25
##
## $nfeval
## [1] 22628
```

Gradient-Based Approaches

Exploiting the direction of "steepest descent" as computed by the negative gradient $-\nabla f(x)$ of a multivariate function.

- **Steepest descent**
  $d_k = -\nabla f(x_k)$
- **Conjugate Gradient** (GC)

$d_k = -\nabla f(x_k) + \beta_k d_{k-1}$, $d_0 = -\nabla f(x_0)$, e.g., $\beta_k = \frac{||\nabla f(x_{k+1})||}{||\nabla f(x_k)||}$ (Fletcher and Reeves).

- **BFGS** and **L-BFGS-B**
  $d_k = -H_f(x_k)^{-1}\nabla f(x_k)$, $H_f(x)$ Hessian of f in x.

Line Searches

Given a function $f : R^n \to R$ and a direction $d \in R^n$, a *line search method* **approximately** minimizes f along the line $\{x + t\, d \mid t \in R\}$.

Armijo-Goldstein inequality: $0 < c, \nu < 1$

$$f(x_0 + t^* d) \leq f(x_0) + c\, \nu^k f'(x_0; d), \quad k = 0, 1, 2, \dots$$

(Weak) Wolf condition: $0 < c_1 < c_2 < 1$

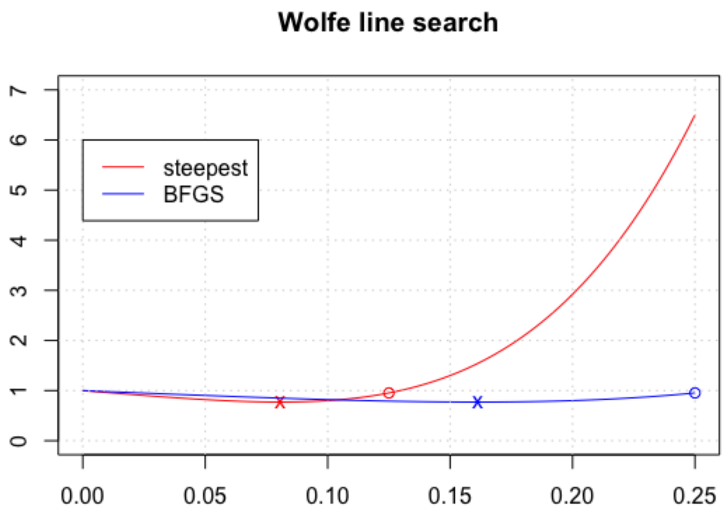
$$f(x_k + t_k d_k) \leq f(x_k) + c_1 t_k f'(x_k; d_k)$$

$$c_2 f'(x_k; d_k) \leq f'(x_k + t_k d_k; g_k)$$

Rosenbrock with Line Search

Steepest descent direction vs. BFGS direction

Wolfe line search these two directions



BFGS and L-BFGS-B

The **Broyden–Fletcher–Goldfarb–Shanno** (BFGS) algorithm

Iteration: While $\|\nabla f_k\| > \epsilon$ do

- compute the search direction: $d_k = -H_k \nabla f_k$
- proceed with line search: $x_{k+1} = x_k + \alpha d_k$
- Update approximate Hessian inverse: $H_{k+1} \approx H_f(x_{k+1})^{-1}$

`L-BFGS` – low-memory BFGS stores matrix H_k in $O(n)$ storage.

`BFGS-B` – BFGS with bound constraints ('active set' approach).

optim() w/ BFGS

```r
optim(rep(0, 20), fn, gr, method = "BFGS",
      control=list(reltol=1e-12, maxit=1000))$par
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```r
optim(rep(0, 20), fn, method = "L-BFGS-B",
      control=list(factr=1e-12, maxit=1000))$par  # factr :
```

```
##  [1] 0.9999987 0.9999984 0.9999982 0.9999981 0.9999980 (
##  [8] 0.9999979 0.9999977 0.9999974 0.9999969 0.9999958 (
## [15] 0.9999797 0.9999613 0.9999243 0.9998500 0.9997011 (
```

```r
optim(rep(0, 20), fn, gr, method = "L-BFGS-B",    # works ;
      control=list(factr=1e-12, maxit=1000))$par
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Best optim() usage

```r
optim(par, fn, gr = function(x) pracma::grad(fn, x), ...,
  method = "L-BFGS-B"",
  lower = -Inf, upper = Inf,
  control = list(factr = 1e-10,
                 maxit = 50*length(par)))
```

- use only `method = "L-BFGS-B"`
  (faster, more accurate, less memory, bound constraints)
- use `factr = 1e-10` for tolerance, default 1e07
- set `maxit = 50*d ... 50*d^2` (default is 100)
- use *dfoptim* or *pracma* for gradients
  (if you don't have an analytical or exact gradient)
- look carefully at the output

More BFGS Packages

- **lbfgsb3** interfaces the Nocedal et al. 'L-BFGS-B.3.0' (2011) (FORTRAN) minimizer with bound constraints.

BUT: Options like "maximum number of function calls" are not accessible. (And the result is returned as 'invisible'.)

```
    sol <- lbfgsb3(par, fn, gr = NULL, lower=-Inf, upper=In
    sol
```

- **lbfgs** interfaces the 'libBFGS' C library by Okazaki with Wolfe line search (based on Nocedal).

BUT: Bound constraints are not accessible through the API.

```
    lbfgs(fn, gr, par, invisible=1)
```

More quasi-Newton type Algorithms

- **stats**::nlm [don't ever use!]
- **stats**::nlminb [PORT routine]

```
nlminb(start, objective, gradient = NULL, hessian = NUL
       scale = 1, control = list(), lower = -Inf, upper
```

- **trustOptim**::trust.optim [trust-region approach] no linesearch, suitable for sparse Hessians

```
trust.optim(x, fn, gr, hs = NULL, control = list(),
            method = c("SR1", "BFGS", "Sparse"), ...)
```

- **ucminf**::ucminf [BFGS + line search + trust region]

```
ucminf(par, fn, gr = NULL, ..., control = list(), hessi
```

ucminf with Rosenbrock

```
fn <- adagio::fnRosenbrock; gr <- adagio::grRosenbrock
sol <- ucminf::ucminf(rep(0, 100), fn, gr, control=list(ma
list(par=sol$par, value = sol$value, conv = sol$conv, mess
```

```
## $par
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##
## $value
## [1] 1.223554e-15
##
## $conv
## [1] 1
##
## $mess
## [1] "Stopped by small gradient (grtol)."
```

More John Nash Work

Thorough implementation of quasi-Newton solvers in pure R.

- **Rcgmin** ("conjugate gradient"")
- **Rvmmin** ("variable metric"")
- **Rtnmin** ("truncated Newton")

Apply, test, and compare different nonlinear optimization solvers for smooth, possibly bound constrained multivariate functions:

- **optimx**, **optimr**, or **optimrx**?

```
optimrx::opm(rep(0, 10), fnRosenbrock, grRosenbrock,
             method = "ALL")
```

Comparison of Nonlinear Solvers

method	value	fevals	gevals	convd	xtime
BFGS	3.127628e-21	291	98	0	0.003
CG	1.916095e-12	1107	408	0	0.010
Nelder-Mead	8.147198e+00	1501	NA	1	0.008
L-BFGS-B	5.124035e-10	78	78	0	0.001
nlm	4.342036e-13	NA	55	0	0.002
nlminb	4.243607e-18	121	97	0	0.002
lbfgsb3	5.124035e-10	78	78	0	0.029
Rcgmin	3.656125e-19	300	136	0	0.004
Rtnmin	5.403094e-13	105	105	0	0.013
Rvmmin	2.935561e-27	116	72	0	0.007
ucminf	1.470165e-15	77	77	0	0.002
newuoa	3.614733e-11	1814	NA	0	0.022
bobyqa	6.939585e-10	2142	NA	0	0.025
nmkb	9.099242e-01	1500	NA	1	0.083
hjkb	8.436900e-07	4920	NA	0	0.033
lbfgs	9.962100e-13	NA	NA	0	0.001

Excurse: Computing Gradients

- manually
- symbolically: package **Deriv**
- numerically: packages **numDeriv** or **pracma**

```r
gr <- function(x) numDeriv::grad(fn, x)  # simple, or:
```

```r
gr <- function(x) pracma::grad(fn, x, heps=6e-06)  # ce
```

- *complex step* derivation

```r
gr <- function(x) pracma::grad_csd(fn, x)
```

- automated differentiation [not yet available]

Central-difference Formula

$\nabla f(x) = (\frac{f(x)}{\partial x_1}, \ldots, \frac{f(x)}{\partial x_n})$ and $\frac{df(x)}{dx}(x) \approx \frac{f(x+h)-f(x-h)}{2 \cdot h}$

```
pracma::grad
```

```
function (f, x0, heps = .Machine$double.eps^(1/3), ...)
{
    # [...input checking...]
    n <- length(x0)
    hh <- rep(0, n)
    gr <- numeric(n)
    for (i in 1:n) {
        hh[i] <- heps
        gr[i] <- (f(x0 + hh) - f(x0 - hh))/(2 * heps)
        hh[i] <- 0
    }
    return(gr)
}
```

Optimization with Constraints

Constraints

- box/bound constraints: $l_i \leq x_i \leq u_i$
  [trick: the 'transfinite' approach]
- linear inequality constraints: $A\,x \leq 0$
- linear equality constraints: $A\,x = b$
  [trick: the 'hyperplane' approach]
- quadratic constraints
- inequality constraints in general
- equality *and* inequality constraints

The 'transfinite' Trick

If the solver does not support bound constraints $l_i \leq x_i \leq u_i$, the *transfinite* approach will do the trick.

Generate a smooth (surjective) function $h : R^n \to [l_i, u_i]$, e.g.

$$h : x_i \to l_i + (u_i - l_i)/2 \cdot (1 + \tanh(x_i))$$

and optimize the composite function $g(x) = f(h(x))$, i.e.

$$g : R^n \to [l_i, u_i] \to R$$

$$x^* = \mathrm{argmin_x\ g(x) = f(h(x))}$$

then $x_{min} = h(x^*)$ will be a minimum of f in $[l_i, u_i]$.

Example: 'Transfinite' Approach

Minimize the Rosenbrock function in 10 dimensions with $0 \le x_i \le 0.5$.

```
Tf <- adagio::transfinite(0, 0.5, 10)
h <- Tf$h; hinv <- Tf$hinv
p0 <- rep(0.25, 10)
f <- function(x) fn(hinv(x))  # f: R^n --> R
g <- function(x) pracma::grad(f, x)

sol <- lbfgs::lbfgs(f, g, p0, epsilon=1e-10, invisible=1)
hinv(sol$par); sol$value
```

```
##  [1] 0.5000000000 0.2630659827 0.0800311137 0.016574234
##  [6] 0.0102120052 0.0102084108 0.0102042121 0.010004085
```

```
## [1] 7.594813
```

Linear Inequality Constraints

Optimization with linear constraints only: $A\,x \ge 0$ (or $A\,x \le 0$)

```
    constrOptim(theta, f, grad, ui, ci, mu = 1e-04, contro
                method = if(is.null(grad)) "Nelder-Mead" e
                outer.iterations = 100, outer.eps = 1e-05,
                hessian = FALSE)
```

- `ui %*% theta - ci >= 0` corresponds to $Ax \ge 0$
- Bounds formulated as linear constraints (even $x_i \ge 0$)
- `theta` must be in the interior of the feasible region
- Inner iteration still calls `optim`

Recommendation: Do not use `constrOptim`. Instead, use an 'augmented Lagrangian' solver, e.g. `alabama::auglag`.

Trick: Linear Equality Constraints

Task: $\min!\ f(x_1, ..., x_n) \quad \text{s.t. } Ax = b$

Let $b_1, ..., b_m$ be a basis of the *nullspace* of A, i.e. $Ab_i = 0$, and x_0 a special solution $Ax_0 = b$. Define a new function $g(s_1, ..., s_m) = f(x_0 + s_1 b_1 + ... + s_m b_m)$ and solve this as a minimization problem *without* constraints:

$$s = \text{argmin } g(s_1, ..., s_m)$$

Then $xmin = x_0 + s_1 b_1 + ... + s_m b_m$ is a (local) minimum.

```
xmin <- lineqOptim(rep(0, 3), fnRosenbrock, grRosenbroc
                   Aeq = c(1,1,1), beq = 1)
xmin
[1] 0.5713651 0.3263519 0.1022830
```

Example: Linear Equality

```
A <- matrix(1, 1, 10)                # x1 + ... + xn = 1
N <- pracma::nullspace(A)            # size 10 9
x0 <- qr.solve(A, 1)                 # A x = 1
fun <- function(x) fn(x0 + N %*% x)   # length(x) = 9
sol <- ucminf::ucminf(rep(0, 9), fun)
xmin <- c(x0 + N %*% sol$par)
xmin; sum(xmin)
```

```
##  [1]  0.559312323  0.314864715  0.102103618  0.01369578:
##  [6]  0.003318010  0.003316801  0.003316309  0.00325210:
```

```
## [1] 1
```

```
fn(xmin)
```

```
## [1] 7.421543
```

Augmented Lagrangian Approach

Task: $\min!\ f(x) \quad \text{s.t. } g_i(x) \geq 0, h_j(x) = 0$

Define the *augmented Lagrangian* function L as

$$L(x, \lambda; \mu) = f(x) - \sum_j \lambda_j h_j(x) + \frac{1}{2\mu} \sum_j h_j^2(x)$$

The inequality constraints $g_i(x) \geq 0$ are included by introducing *slack* variables s_i and replacing the inequality constraints with

$$g_i(x) - s_i = 0, \quad s_i \geq 0$$

The bound constraints are treated differently (e.g., through the LANCELOT algorithm).

Augmented Lagrangian Solvers

- **alabama**

```
auglag(par, fn, gr, hin, hin.jac, heq, heq.jac,
         control.outer=list(), control.optim = list(), ..
```

- **NLoptr**

```
auglag(x0, fn, gr = NULL, lower = NULL, upper = NULL,
          hin = NULL, hinjac = NULL, heq = NULL, heqjac =
          localsolver = c("COBYLA"), localtol = 1e-6, ine
          nl.info = FALSE, control = list(), ...)
```

- **Rsolnp**
- **NlcOptim** (Sequential Quadratic programming, SQP)
- **Rdonlp2** (removed from CRAN, see R-Forge's Rmetrics)

Example with alabama::auglag

Minimize the Rosenbrock function with constraints
$x_1 + \ldots + x_n = 1$ and $0 \leq x_i \leq 1$ for all $i = 1, \ldots, n$.

```
fheq <- function(x) sum(x) - 1
fhin <- function(x) c(x)

sol <- alabama::auglag(rep(0, 10), fn, gr, heq = fheq, hin
             control.outer = list(trace = FALSE, method = "r
print(sol$par, digits=5)
```

```
##  [1]  5.5707e-01  3.1236e-01  1.0052e-01  1.3367e-02  3.
##  [6]  3.3082e-03  3.3071e-03  3.3069e-03  3.2854e-03 -7.
```

```
sum(sol$par)
```

```
## [1] 1
```

The *nloptr* Package (NLopt Library)

- COBYLA (Constrained Optimization By Linear Approximation)

```
cobyla(x0, fn, lower = NULL, upper = NULL, hin = NULL,
    nl.info = FALSE, control = list(), ...)
```

- slsqp (Sequential Quadratic Programming, SQP)

```
slsqp(x0, fn, gr = NULL, lower = NULL, upper = NULL,
  hin = NULL, hinjac = NULL, heq = NULL, heqjac = NULL,
  nl.info = FALSE, control = list(), ...)
```

- auglag (Augmented Lagrangian)

```
auglag(x0, fn, gr = NULL, lower = NULL, upper = NULL,
        hin = NULL, hinjac = NULL, heq = NULL, heqjac =
        localsolver = c("COBYLA", "LBFGS", "MMA", "SLSQ
        nl.info = FALSE, control = list(), ...)
```

Quadratic Optimization

Quadratic Programming

Quadratic Programming (QP) is the problem of optimizing a quadratic expression of several variables subject to linear constraints.

$$\begin{aligned} \text{Minimize} \quad & \frac{1}{2}x^T Q x + c^T x \\ s.t. \quad & Ax \leq b \end{aligned}$$

where Q is a symmetric, positive (semi-)definite $n \times n$-matrix, c an n-dim. vector, A an $m \times n$-matrix, and b an m-dim. vector.

For some solvers, linear equality constraints are also allowed.

Example: The *enclosing ball problem*

Quadratic Solvers

Standard solver for quadratic problems in R is `solve.QP` in package *quadprog*. The matrix Q has to be positive definite.

```
solve.QP(Dmat, dvec, Amat, bvec, meq=0, factorized=FALS
```

Package	Function	Matrix	Timings
quadprog	solve.QP	pdef	1
kernlab	ipop	spdef	50
LowRankQP	LowRankQP	spdef	2
DWD	solve_QP_SOCP	pdef	9500
coneproj	qprog	pdef	–

Nonsmooth Optimization

Nonsmoothness: Minimax Problems

Functions defined as maximum are not smooth and cannot be optimized through a straightforward gradient-based approach.

Task: $\min!\ f(x) = max(f_1(x), \dots, f_m(x))$

Instead, define a smooth function $g(x_1, \dots, x_n, x_{n+1}) = x_{n+1}$ and minimze it under constraints

$$x_{n+1} \geq f_i(x_1, \dots, x_n) \quad \text{for all } i = 1, \dots, m$$

The solution $(x_1, \dots, x_n, x_{n+1})$ returns the minimum point $xmin = (x_1, \dots, x_n)$ as well as the minimal value $fmin = x_{n+1}$.

[Cf. the example in Chapter ?? in the *bookdown* text.]

Least Squares Solvers

Linear Least-squares

A linear least-squares (LS) problem means solving $\min \|Ax = b\|_2$, possibly with bounds or linear constraints.

The function `qr.solve(A, b)` from Base R solves over- and underdetermined linear systems in the least-squares sense.

- **nnls** (Lawson-Hansen algorithm)
  linear LS with non-negative/-positive constraints
- **bvls** (Stark-Parker algorithm)
  linear LS with bound constraints $l \leq x \leq u$
- **pracma**::`lsqlincon(A, b, ...)`
  linear LS with linear equality and inequality constraints (applies a quadratic solver)

Nonlinear Least-squares

The standard nonlinear LS estimator for model parameter, given some data, in Base R is:

```
nls(formula, data, start, control, algorithm[="plinear
    trace, subset, weights, na.action, model,
    lower, upper, ...)
```

Problems:

- too small or zero residuals
- "singular gradient" error message (R-help, Stackoverflow)
- too many local minima, proper starting point
  (cf. `nls2` with random or grid-based start points)
- bounds require the 'port' algorithm (Port library)
  (recommended anyway)

'Stabilized' Nonlinear LS

Modern nonlinear LS solvers use the Levenberg-Marquardt method (not Gauss-Newton) to minimize sums of squares.

- **minpack.lm**

```
nlsLM(formula, data = parent.frame(), start, jac = NULL
      algorithm = "LM", control = nls.lm.control(),
      lower = NULL, upper = NULL, trace = FALSE, ...)
```

- **nlmrt**

```
nlxb(formula, start, trace=FALSE, data, lower=-Inf, upp
     masked=NULL, control, ...)
```

Cf. also **pracma**::lsqnonlin(fun, x0, options = list(), ...)

Tip: Rosenbrock as LS Problem

Redefine Rosenbrock as vector-value function:

```
fn <- function(x) {
    n <- length(x)
    x1 <- x[2:n]; x2 <- x[1:(n - 1)]
    c(10*(x1 - x2^2), 1 - x2)
}
```

and now apply **pracma**'s `lsqnonlin`:

```
lsqnonlin(fn, rep(0, 20))
## $x
## [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## $ssq
## [1] 3.037124e-19
```

Quantile Regression

Median (or: L^1) Regression: $\min! \sum |y - A\,x|$
(aka "least absolute deviation"" (LAD) regression)

- quantreg

```
rq(formula, tau = 0.5, data, subset, weights, na.action,
   method = "br", model = TRUE, contrasts, ...)
```

- pracma

```
L1linreg(A, b, p = 1, tol = 1e-07, maxiter = 200)
```

solves the linear system $A\,x = b$ in an L^p sense, i.e. minimizes the term $\sum |b - A\,x|^p$ (for $0 < p \leq 1$) by applying an "iteratively reweighted least square" (IRLS) method.

Global Optimization

DE Solvers

Differential Evolution (DE) is a relatively simple genetic algorithm variant, specialized for real-valued functions (10-20 dims).

- **DEoptim**

```
DEoptim(fn, lower, upper,
    control = DEoptim.control(trace = FALSE), ..., fnMa
```

- **RcppDE**

```
DEoptim(fn, lower, upper, control = DEoptim.control(),
```

- **DEoptimR**

```
JDEoptim(lower, upper, fn,
     constr = NULL, meq = 0, eps = 1e-05, NP = 10*d[, .
```

CMA-ES Solvers

Covariance Matrix Adaptation – Evolution Strategy (CMA-ES) is an evolutionary algorithm for continuous optimization problems (adapting the covariance matrix). *It is quite difficult to implement*, but is applicable to dimensions up to 50 or more.

- Packages that contain CMA-ES solvers:
  cmaes
  cmaesr
  rCMA
  parma::cmaes
  Rmalschains
  adagio::pureCMAES

```
pureCMAES(par, fun, lower = NULL, upper = NULL, sigma =
          stopfitness = -Inf, stopeval = 1000*length(pa
```

More Evolutionary Approaches

- Simulated Annealing (SA)
  GenSA
- Genetic Algorithms (GA)
  GA, **genalg**, **SOMA**, **rgenoud**
- Particle Swarm Optimization (PSO)
  pso, **psoptim**, **hydroPSO**

NMOF: DEopt, GAopt, PSopt

NLoptr: crs2lm, direct, mlsl, isres, stogo

The *gloptim* Package

Package **gloptim** incorporates and compares 25 stochastic solvers. The following is a typical output, here only showing the results of CMA-ES and DE solvers for the 'Runge' problem:

```
         solver      package  fmin          time
 1    purecmaes       adagio  0.06546780  43.583
 2        cmaes        parma  0.06546780  23.523
 3     cmaoptim         rCMA  0.06546780  91.257
 4   malschains  Rmalschains  0.06546781  76.457
 5        deopt         NMOF  0.06546876  75.809
 6     deoptimr     DEoptimR  0.06549435  57.712
 7     simplede       adagio  0.06573988  84.000
 8       cma_es        cmaes  0.07430865   7.208
 9        cmaes       cmaesr  0.07503498   8.305
...
22   cppdeoptim       RcppDE  6.82525344  17.050
23      deoptim      DEoptim  7.28454226  39.287
```

Future Developments

ROI – R Optimization Infrastructure

Available Plugins:
glpk, symphony, quadprog, ipop, ecos, scs, nloptr, cplex, ...

```
library(ROI); library(ROI.plugin.glpk)  # ...
v <- c(15, 100, 90, 60, 40, 15, 10,  1)
w <- c( 2,  20, 20, 30, 40, 30, 60, 10)

mat <- matrix(w, nrow = 1)
con <- L_constraint(L = mat, dir = "<=", rhs = 105)

pro <- OP(objective = v, constraints = con,
          types = rep("B", 8), maximum = TRUE)

ROI_applicable_solvers(pro)     # [1] "clp" "glpk" ...
sol <- ROI_solve(pro, solver = "ecos")
## Optimal solution found.
## The objective value is: 2.800000e+02
```

CVXR

CVXR provides an R modeling language for convex optimization problems (announced UseR!2016, not yet ready).

Example: Estimating a discrete distribution, e.g.

$$\begin{array}{ll} \max! & \sum_{i=1}^{m} -w_i \log w_i \\ \text{s.t.} & w_i \geq 0, \quad \sum w_i = 1, \quad X^T w = b \end{array}$$

```
library(CVXR)
w <- Variable(m)
obj <- SumEntries(Entr(w))   # entropy function
constr <- list(w >= 0, SumEntries(w) == 1, t(X) %*% w == b)
pro <- Problem(Maximize(obj), constr)
sol <- solve(pro)
sol$w
```

Using *Julia* Solvers

Ipopt (Interior Point OPTimizer) is a software package for large-scale nonlinear optimization (with nonlinear equality and inequality constraints).

- difficult to install (extra components needed)
- ECLIPSE license (not allowed on CRAN?)

There is an easy-to-install *Ipopt.jl* package for Julia.

With the R packages *XR* and *XRJulia* (John Chambers, 2016) it will be possible to utilize this with a new R package *ipoptjlr*.

```
library(ipoptjlr)
julia_setup("path_to_julia")
IPOPT(x, x_L, x_U, g_L, g_U, eval_f, eval_g,
      eval_grad_f, jac_g1, jac_g2, h1, h2)
```

Using the NEOS Solvers

"The **NEOS Server** https://neos-server.org/neos/ is a free internet-based service for solving numerical optimization problems. [It] provides access to more than 60 state-of-the-art [free and commercial] solvers."

rneos: XML-RPC Interface to NEOS

```
# submit job to the NEOS solver
neosjob <- NsubmitJob(xmlstring, user = "hwb", interface =
                      id = 8237, nc = CreateNeosComm())
neosjob
# The job number is: 3838832
# The pass word is : wBgHomLT

# getting info about job
NgetJobInfo(neosjob)            # "nco"   "MINOS" "AMPL"
NgetFinalResults(neosjob)
```

Epilogue

“What can go wrong?”

- Modell, constraints, gradients, ...
- Local: bad starting values
  Global: no guaranteed optimum
- Applying appropriate solvers
- Setting solver controls
- Special problems, e.g.
  Non-smooth objective functions, noise, ...
- Understanding solver output (and error messages)
  convergence, accuracy, no. of loops and function calls
- Checking results

“Most methods work most of the time.” – John Nash

References

- Theussl, S., and H. W. Borchers (2017). CRAN Task View: **Optimization and Mathematical Programming**.
  URL: https://CRAN.R-project.org/view=Optimization
- Nash, J. C. (2014). **Nonlinear Parameter Optimization Using R Tools**. John Wiley and Sons, Chichester, UK.
- Varadhan, R., Editor (2014). Special Issue: **Numerical Optimization in R: Beyond optim**. Journal of Statistical Software, Vol. 60.
- Bloomfield, V. A. (2014). **Using R for Numerical Analysis in Science and Engineering**. CRC Press, USA. (Chapter 7, 40 pp.)
- Cortez, P. (2014). **Modern Optimization With R**. Use R! Series, Springer Intl. Publishing, Switzerland.