Chapter 22
Function Optimization

Most data-driven scientific inference, qualitative, quantitative, and visual analytics involve formulating, understanding the behavior of, and optimizing objective (cost) functions. Presenting the mathematical foundations of representation and interrogation of diverse spectra of objective functions provides mechanisms for obtaining effective solutions to complex big data problems. (Multivariate) *function optimization* (minimization or maximization) is the process of searching for variables x_1, x_2, x_3, …, x_n that either minimize or maximize the multivariate cost function $f(x_1, x_2, x_3, \ldots, x_n)$. In this chapter, we will specifically discuss (1) constrained and unconstrained optimization; (2) Lagrange multipliers; (3) linear, quadratic and (general) non-linear programming; and (4) data denoising.

22.1 Free (Unconstrained) Optimization

We will start with function optimization without restrictions for the domain of the cost function, $\Omega \ni \{x_i\}$. The extreme value theorem suggests that a solution to the free optimization processes, $\min_{x_1,x_2,x_3,\ldots,x_n} f(x_1, x_2, x_3, \ldots, x_n)$ or $\max_{x_1,x_2,x_3,\ldots,x_n} f(x_1, x_2, x_3, \ldots, x_n)$, may be obtained by a gradient vector descent method. This means that we can minimize/maximize the objective function by finding solutions to $\nabla f = \left\{\frac{df}{dx_1}, \frac{df}{dx_2}, \ldots, \frac{df}{dx_1}\right\} = \{0, 0, \ldots, 0\}$. Solutions to this equation, x_1, …, x_n, will present candidate (local) minima and maxima.

In general, identifying critical points using the gradient or tangent plane, where the partial derivatives are trivial, may not be sufficient to determine the *extrema* (*minima* or *maxima*) of multivariate objective functions. Some critical points may represent inflection points, or local extrema that are far from the global *optimum* of the objective function. The eigenvalues of the Hessian matrix, which includes the second order partial derivatives, at the critical points provide clues to pinpoint extrema. For instance, invertible Hessian matrices that (i) are positive definite (i.e.,

I. D. Dinov, *Data Science and Predictive Analytics*,
https://doi.org/10.1007/978-3-319-72347-1_22

all eigenvalues are positive), yield a local minimum at the critical point, (ii) are negative definite (all eigenvalues are negative) at the critical point suggests that the objective function has a local maximum, and (iii) have both positive and negative eigenvalues yield a saddle point for the objective function at the critical point where the gradient is trivial.

There are two complementary strategies to avoid being trapped in *local* extrema. First, we can run many iterations with different initial vectors. At each iteration, the objective function may achieve a (local) maximum/minimum/saddle point. Finally, we select the overall minimal (or maximal) value from all iterations. Another adaptive strategy involves either adjusting the step sizes or accepting solutions *in probability*, e.g., simulated annealing is one example of an adaptive optimization.

22.1.1 Example 1: Minimizing a Univariate Function (Inverse-CDF)

The cumulative distribution function (CDF) of a real-valued random process X, also known as the distribution function of X, represents the probability that the random variable X does not exceed a certain level. Mathematically speaking, the CDF of X is $F_X(x) = P(X \leq x)$. Recall the Chap. 2 discussions of Uniform, Normal, Cauchy, Binomial, Poisson and other discrete and continuous distributions. Also explore the dynamic representations of density and distribution functions included in the Probability Distributome Calculators (http://distributome.org).

For each $p \in [0, 1]$, the *inverse distribution function*, also called *quantile function* (e.g., `qnorm`), yields the critical value (x) at which the probability of the random variable is less than or equal to the given probability (p). When the CDF F_X is continuous and strictly increasing, the value of the inverse CDF at p, $F^{-1}(p) = x$, is the unique real number x such that $F(x) = p$.

Below, we will plot the probability density function (PDF) and the CDF for *Normal* distribution in R (Fig. 22.1).

```
par(mfrow=c(1,2), mar=c(3,4,4,2))
z<-seq(-4, 4, 0.1)  # points from -4 to 4 in 0.1 steps
q<-seq(0.001, 0.999, 0.001)  # probaility quantile values from 0.1%
to 99.9% in 0.1% steps
dStandardNormal <- data.frame(Z=z, Density=dnorm(z, mean=0, sd=1),
Distribution=pnorm(z, mean=0, sd=1))
plot(z, dStandardNormal$Density, col="darkblue",xlab="z",
ylab="Density", type="l",lwd=2, cex=2, main="Standard Normal PDF",
cex.axis=0.8)
# could also do
# xseq<-seq(-4, 4, 0.01); density<-dnorm(xseq, 0, 1); plot (density,
main="Density")
# Compute the CDF
xseq<-seq(-4, 4, 0.01); cumulative<-pnorm(xseq, 0, 1)
# plot (cumulative, main="CDF")
plot(xseq, cumulative, col="darkred", xlab="", ylab="Cumulative
Probability", type="l",lwd=2, cex=2, main="CDF of (Simulated)
Standard Normal", cex.axis=.8)
```

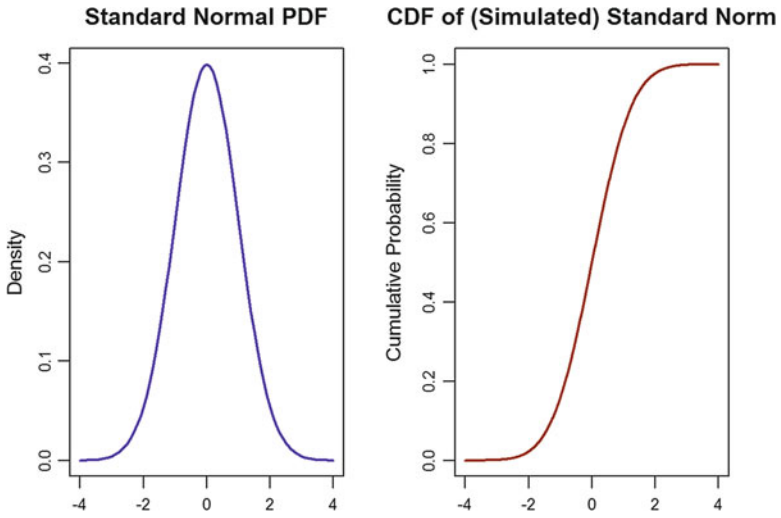

Fig. 22.1 Plots of the density and cumulative distribution functions of the simulated data

Suppose we are interested in computing, or estimating, the inverse-CDF from first principles. Specifically, to invert the CDF, we need to be able to solve the following equation (representing our objective function):

$$CDF(x) - p = 0.$$

The `uniroot` and `stats::nlm` R functions do *non-linear minimization* of a function f using a Newton-Raphson algorithm.

```
set.seed(1234)
x <- rnorm(1000, 100, 20)
pdf_x <- density(x)

# Interpolate the density, the values returned when input x values are outsi
de [min(x): max(x)] should be trivial
f_x <- approxfun(pdf_x$x, pdf_x$y, yleft=0, yright=0)

# Manual computation of the cdf by numeric integration
cdf_x <- function(x){
  v <- integrate(f_x, -Inf, x)$value
  if (v<0) v <- 0
  else if(v>1) v <- 1
  return(v)
}

# Finding the roots of the inverse-CDF function by hand (CDF(x)-p=0)
invcdf <- function(p){
  uniroot(function(x){cdf_x(x) - p}, range(x))$root
  # alternatively, can use
  # nlm(function(x){cdf_x(x) - p}, 0)$estimate
  # minimum - the value of the estimated minimum of f.
  # estimate - the point at which the minimum value of f is obtained.
}
```

```
invcdf(0.5)

## [1] 99.16995

# We can validate that the inverse-CDF is correctly computed: F^{-1}(F(x))==
x
cdf_x(invcdf(0.8))

## [1] 0.8
```

The ability to compute exactly, or at least estimate, the inverse-CDF function is important for many reasons. For instance, generating random observations from a specified probability distribution (e.g., normal, exponential, or gamma distribution) is an important task in many scientific studies. One approach for such random number generation from a specified distribution evaluates the inverse CDF at random uniform $u \sim U(0, 1)$ values. Recall that in Chap. 16 we showed an example of generating random uniform samples using atmospheric noise. The key step is ability to quickly, efficiently and reliably estimate the inverse CDF function, of which we just showed one example.

Let's see why inverting the CDF using random uniform data works. Consider the cumulative distribution function (CDF) of a probability distribution from which we are interested in sampling. If the CDF has a closed form analytical expression and is invertible, then we generate a random sample from that distribution by evaluating the inverse CDF at u, where $u \sim U(0, 1)$. This is possible since a continuous CDF, F, is a one-to-one mapping of the domain of the CDF (range of X) into the interval $[0, 1]$. Therefore, if U is a uniform random variable on $[0, 1]$, then $X = F^{-1}(U)$ has the distribution F. Suppose $U \sim Uniform[0, 1]$, then $P(F^{-1}(U) \leq x) = P(U \leq F(x))$, by applying F to both sides of this inequality, since F is monotonic. Thus, $P(F^{-1}(U) \leq x) = F(x)$, since $P(U \leq u) = u$ for uniform random variables.

22.1.2 Example 2: Minimizing a Bivariate Function

Let's look at the function $f(x_1, x_2) = (x_1 - 3)^2 + (x_2 + 4)^2$. We define the function in R and utilize the `optim()` function to obtain the extrema points in the support of the objective function and/or the extrema values at these critical points.

```
require("stats")
f <- function(x) { (x[1] - 3)^2 + (x[2] +4)^2 }
initial_x <- c(0, -1)
x_optimal <- optim(initial_x, f, method="CG") # performs minimization
x_min <- x_optimal$par
# x_min contains the domain values where the (local) minimum is attained
x_min   # critical point/vector

## [1]  3 -4

x_optimal$value  # extrema value of the objective function

## [1] 8.450445e-15
```

`optim` allows the use of six candidate optimization strategies:

- **Nelder-Mead**: robust but relatively slow, works reasonably well for non-differentiable functions.
- **BFGS**: quasi-Newton method (also known as a variable metric algorithm), uses function values and gradients to build up a picture of the surface to be optimized.
- **CG**: conjugate gradients method, fragile, but successful in larger optimization problems because it's unnecessary to save large matrix.
- **L-BFGS-B**: allows box constraints.
- **SANN**: a variant of simulated annealing, belonging to the class of stochastic global optimization methods.
- **Brent**: for one-dimensional problems only, useful in cases where `optim()` is used inside other functions where only method can be specified.

22.1.3 *Example 3: Using Simulated Annealing to Find the Maximum of an Oscillatory Function*

Consider the function $f(x) = 10 \sin(0.3x) \times \sin(1.3x^2) - 0.00002x^4 + 0.3x + 35$. Maximizing $f()$ is equivalent to minimizing $-f()$. Let's plot this oscillatory function, then find and report its critical points and extremum values.

The function `optim` returns two important results:

- `par`: the best set of domain parameters found to optimize the function
- `value`: the extreme values of the function corresponding to par (Fig. 22.2).

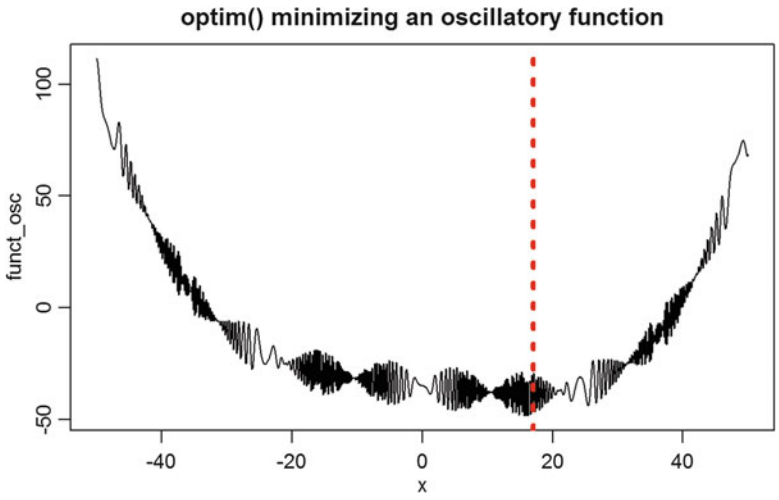

Fig. 22.2 Example of minimizing and oscillatory function, $f(x) = 10 \sin(0.3x) \times \sin(1.3x^2) - 0.00002x^4 + 0.3x + 35$, using `optim`

```
funct_osc <- function (x) { -(10*sin(0.3*x)*sin(1.3*x^2) - 0.00002*x^4 +
0.3*x+35) }
plot(funct_osc, -50, 50, n = 1000, main = "optim() minimizing an oscillatory
function")
abline(v=17, lty=3, lwd=4, col="red")
```

```
res <- optim(16, funct_osc, method = "SANN", control = list(maxit = 20000, t
emp = 20, parscale = 20))
res$par

## [1] 15.66197

res$value

## [1] -48.49313
```

22.2 Constrained Optimization

22.2.1 Equality Constraints

When there are support restrictions, dependencies, or other associations between the domain variables $x_1, x_2, \ldots, x_n$, constrained optimization needs to be applied.

For example, we can have k equations specifying these restrictions, which may specify certain model characteristics:

$$\begin{cases} g_1(x_1, x_2, \ldots, x_n) = 0 \\ \ldots \\ g_k(x_1, x_2, \ldots, x_n) = 0 \end{cases}.$$

Note that the right hand sides of these equations may always be assumed to be trivial (0), otherwise we can just move the non-trivial parts within the constraint functions g_i. Linear Programming, Quadratic Programming, and Lagrange multipliers may be used to solve such equality-constrained optimization problems.

22.2.2 Lagrange Multipliers

We can merge the equality constraints within the objective function ($f \rightarrow f^*$). Lagrange multipliers represent a typical solution strategy that turns the *constrained* optimization problem ($\min_x f(x)$ subject to $g_i(x_1, x_2, \ldots, x_n)$, $1 \leq i \leq k$), into an *unconstrained* optimization problem:

$$f^*(x_1, x_2, \ldots, x_n; \lambda_1, \lambda_2, \ldots, \lambda_k) = f(x_1, x_2, \ldots, x_n) + \sum_{i=1}^{k} \lambda_i g_i(x_1, x_2, \ldots, x_n).$$

Then, we can apply traditional unconstrained optimization schemas, e.g., extreme value theorem, to minimize the unconstraint problem:

$$f^*(x_1, x_2, \ldots, x_n; \lambda_1, \lambda_2, \ldots, \lambda_k) = f(x_1, x_2, \ldots, x_n) + \lambda_1 g_1(x_1, x_2, \ldots, x_n) + \cdots + \lambda_k g_k(x_1, x_2, \ldots, x_n).$$

This represents an unconstrained optimization problem using Lagrange multipliers.

The solution of the constrained problem is also a solution to:

$$\nabla f^* = \left[\frac{df}{dx_1}, \frac{df}{dx_2}, \ldots, \frac{df}{dx_n}; \frac{df}{d\lambda_1}, \frac{df}{d\lambda_2}, \ldots, \frac{df}{d\lambda_k}\right] = [0, 0, \ldots, 0].$$

22.2.3 Inequality Constrained Optimization

There are no general solutions for arbitrary inequality constraints; however, partial solutions do exist when some restrictions on the form of constraints are present.

When both the constraints and the objective function are `linear functions` of the domain variables, then the problem can be solved by Linear Programming.

Linear Programming (LP)

LP works when the objective function is a linear function. The constraint functions are also linear combination of the same variables.

Consider the following elementary (`minimization`) example:

$$\min_{x_1, x_2, x_3} (-3x_1 - 4x_2 - 3x_3)$$

subject to:

$$\begin{cases} 6x_1 + 2x_2 + 4x_3 & \leq 150 \\ x_1 + x_2 + 6x_3 & \geq 0 \\ 4x_1 + 5x_2 + 4x_3 & = 40 \end{cases}.$$

The exact solution is $x_1 = 0$, $x_2 = 8$, $x_3 = 0$, and can be computed using the package `lpSolveAPI` to set up the constraint problem and the generic `solve()` method to find its solutions.

```
# install.packages("lpSolveAPI")
library(lpSolveAPI)

lps.model <- make.lp(0, 3) # define 3 variables
# add the constraints as a matrix of the linear coefficients, relations and
RHS
add.constraint(lps.model, c(6, 2, 4), "<=", 150)
add.constraint(lps.model, c(1, 1, 6), ">=",   0)
add.constraint(lps.model, c(4, 5, 4), "=" ,  40)
# set objective function (default: find minimum)
set.objfn(lps.model, c(-3, -4, -3))
# you can save the model to a file
# write.lp(lps.model, 'c:/Users/LPmodel.lp', type='lp')

# these commands define the constraint linear model
# /* Objective function */
#   min: -3 x1 -4 x2 -3 x3;
#
# /* Constraints */
# +6 x1 +2 x2 +4 x3 <= 150;
# +  x1 +  x2 +6 x3 >=   0;
# +4 x1 +5 x2 +4 x3  =  40;
#
# writing it in the text file named 'LPmodel.lp'

solve(lps.model)

## [1] 0

# Retrieve the values of the variables from a solved linear program model
get.variables(lps.model)  # check against the exact solution x_1 = 0,
x_2 = 8, x_3 = 0

## [1] 0 8 0

get.objective(lps.model) # get optimal (min) value

## [1] -32
```

In lower dimensional problems, we can also plot the constraints to graphically demonstrate the corresponding support restriction. For instance, here is an example of a simpler 2D constraint and its Venn diagrammatic representation (Fig. 22.3).

$$\begin{cases} x_1 & \leq \dfrac{150 - 2x_2}{6} \\ x_1 & \geq -x_2 \end{cases}.$$

```
library(ggplot2)
ggplot(data.frame(x = c(-100, 0)), aes(x = x)) +
  stat_function(fun=function(x) {(150-2*x)/6}, aes(color="Function 1")) +
  stat_function(fun=function(x) { -x }, aes(color = "Function 2")) +
  theme_bw() +
  scale_color_discrete(name = "Function") +
  geom_polygon(
```

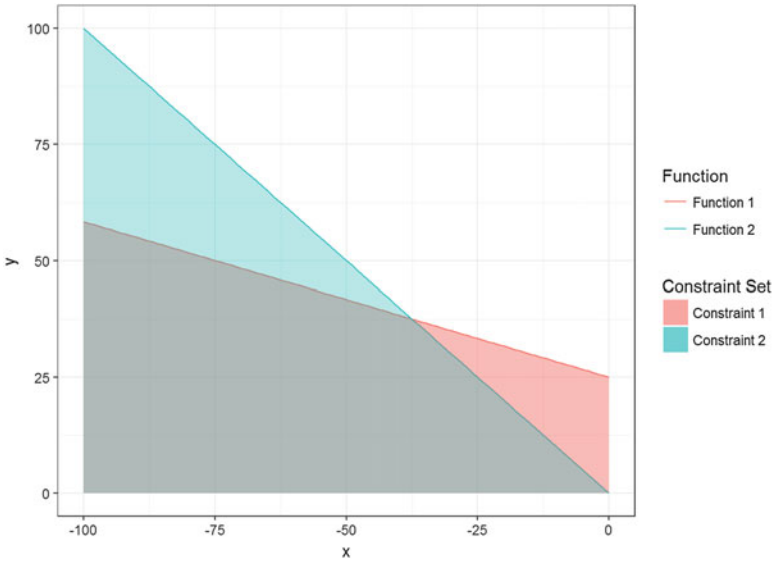

Fig. 22.3 A 2D graphical depiction of the function optimization support restriction constraints

```
    data = data.frame(x = c(-100, -100, 0, 0, Inf), y = c(0,350/6, 150/6,
0, 0)),
    aes(x = x, y = y, fill = "Constraint 1"),
    inherit.aes = FALSE, alpha = 0.5) +
  geom_polygon(
    data = data.frame(x = c(-100, -100, 0, Inf), y = c(0, 100, 0, 0)),
    aes(x = x, y = y, fill = "Constraint 2"),
    inherit.aes = FALSE, alpha = 0.3) +
  scale_fill_discrete(name = "Constraint Set") +
  scale_y_continuous(limits = c(0, 100))
```

Here is another example of `maximization` of a trivariate cost function, $f(x_1, x_2, x_3) = 3x_1 + 4x_2 - x_3$, subject to:

$$\begin{cases} -x_1 + 2x_2 + 3x_3 & \leq 16 \\ 3x_1 - x_2 - 6x_3 & \geq 0 \\ x_1 - x_2 & \leq 2 \end{cases}.$$

```
lps.model2 <- make.lp(0, 3)
add.constraint(lps.model2, c(-1, 2, 3), "<=", 16)
add.constraint(lps.model2, c(3, -1, -6), ">=",  0)
add.constraint(lps.model2, c(1, -1, 0), "<=",  2)
set.objfn(lps.model2, c(3, 4, -1), indices = c(1, 2, 3))
lp.control(lps.model2, sense='max')     # changes to max: 3 x1 + 4 x2 - x3

## $anti.degen
## [1] "fixedvars" "stalling"
##
## $basis.crash
## [1] "none"
##
## $bb.depthlimit
## [1] -50
##
## $bb.floorfirst
## [1] "automatic"
##
## $bb.rule
## [1] "pseudononint" "greedy"       "dynamic"      "rcostfixing"
##
## $break.at.first
## [1] FALSE
##
## $break.at.value
## [1] 1e+30
##
## $epsilon
##       epsb       epsd      epsel     epsint epsperturb   epspivot
##      1e-10      1e-09      1e-12      1e-07      1e-05      2e-07
##
## $improve
## [1] "dualfeas" "thetagap"
##
## $infinite
## [1] 1e+30
##
## $maxpivot
## [1] 250
##
## $mip.gap
## absolute relative
##    1e-11    1e-11
##
## $negrange
## [1] -1e+06
##
## $obj.in.basis
## [1] TRUE
##
## $pivoting
## [1] "devex"    "adaptive"
##
## $presolve
## [1] "none"
```

```
##
## $scalelimit
## [1] 5
##
## $scaling
## [1] "geometric"   "equilibrate" "integers"
##
## $sense
## [1] "maximize"
##
## $simplextype
## [1] "dual"   "primal"
##
## $timeout
## [1] 0
##
## $verbose
## [1] "neutral"
```

```
solve(lps.model2)         # 0 suggests that this solution convergences
```

```
## [1] 0
```

```
get.variables(lps.model2) # get point of maximum
```

```
## [1] 20 18  0
```

```
get.objective(lps.model2) # get optimal (max) value
```

```
## [1] 132
```

In 3D, we can utilize the `rgl::surface3d()` method to display the constraints. This output is suppressed, as it can only be interpreted via the pop-out 3D rendering window.

```
library("rgl")
n <- 100
x <- y <- seq(-500, 500, length = n)
region <- expand.grid(x = x, y = y)

z1 <- matrix(((150 -2*region$x -4*region$y)/6), n, n)
z2 <- matrix(-region$x + 6*region$y, n, n)
z3 <- matrix(40 -5*region$x - 4*region$y, n, n)

surface3d(x, y, z1, back = 'line', front = 'line', col = 'red', lwd = 1.5,
alpha = 0.4)
surface3d(x, y, z2, back = 'line', front = 'line', col = 'orange', lwd =
1.5, alpha =0.4)
surface3d(x, y, z3, back = 'line', front = 'line', col = 'blue', lwd = 1.5,
alpha = 0.4)
axes3d()
```

It is possible to restrict the domain type to contain only solutions that are:

- *integers*, which makes it an Integer Linear Programming (ILP),
- *binary/boolean* values (BLP), or
- *mixed* types, Mixed Integer Liner Programming (MILP).

Some examples are included below.

Mixed Integer Linear Programming (MILP)

Let's demonstrate MILP with an example where the type of x_1 is unrestricted, x_2 is dichotomous (binary), and x_3 is restricted to be an integer.

```
lps.model <- make.lp(0, 3)
add.constraint(lps.model, c(6, 2, 4), "<=", 150)
add.constraint(lps.model, c(1, 1, 6), ">=", 0)
add.constraint(lps.model, c(4, 5, 4), "=", 40)
set.objfn(lps.model, c(-3, -4, -3))

set.type(lps.model, 2, "binary")
set.type(lps.model, 3, "integer")
get.type(lps.model) # This is Mixed Integer Linear Programming (MILP)

## [1] "real"    "integer" "integer"

set.bounds(lps.model, lower=-5, upper=5, columns=c(1))

# give names to columns and restrictions
dimnames(lps.model) <- list(c("R1", "R2", "R3"), c("x1", "x2", "x3"))

print(lps.model)

## Model name:
##             x1    x2    x3
## Minimize    -3    -4    -3
## R1           6     2     4  <=  150
## R2           1     1     6  >=    0
## R3           4     5     4   =   40
## Kind       Std   Std   Std
## Type      Real   Int   Int
## Upper        5     1   Inf
## Lower       -5     0     0

solve(lps.model)

## [1] 0

get.objective(lps.model)

## [1] -30.25

get.variables(lps.model)

## [1] 4.75 1.00 4.00

get.constraints(lps.model)

## [1] 46.50 29.75 40.00
```

The next example limits all three variable to be dichotomous (binary).

```
lps.model <- make.lp(0, 3)
add.constraint(lps.model, c(1, 2, 4), "<=", 5)
add.constraint(lps.model, c(1, 1, 6), ">=", 2)
add.constraint(lps.model, c(1, 1, 1), "=",  2)
set.objfn(lps.model, c(2, 1, 2))

set.type(lps.model, 1, "binary")
set.type(lps.model, 2, "binary")
set.type(lps.model, 3, "binary")

print(lps.model)

## Model name:
##            C1   C2   C3
## Minimize    2    1    2
## R1          1    2    4  <=  5
## R2          1    1    6  >=  2
## R3          1    1    1   =  2
## Kind      Std  Std  Std
## Type      Int  Int  Int
## Upper       1    1    1
## Lower       0    0    0

solve(lps.model)

## [1] 0

get.variables(lps.model)

## [1] 1 1 0
```

22.2.4 Quadratic Programming (QP)

QP can be used for second order (quadratic) objective functions, but the constraint functions are still linear combinations of the domain variables.

A matrix formulation of the problem can be expressed as minimizing an objective function:

$$f(X) = \frac{1}{2} X^T D X - d^T X,$$

where X is a vector $[x_1, x_2, \ldots, x_n]^T$, D is the matrix of weights of each association pair, x_i, x_j, and d are the weights for each individual feature, x_i. The $\frac{1}{2}$ coefficient ensures that the weights matrix D is symmetric and each x_i, x_j pair is not double-counted. This cost function is subject to the constraints:

$$A^T X \; [= \mid \geq] \; b,$$

where the first k constrains may represent equalities ($=$) and the remaining ones are inequalities ($\geq$), and b is the constraints right hand size (RHS) constant vector.

Here is an example of a QP objective function and its R optimization:

$$f(x_1, x_2, x_3) = 2x_1^2 - x_1x_2 - 2x_2^2 + x_2x_3 + 2x_3^2 - 5x_2 + 3x_3.$$

Subject to the following constraints:

$$\begin{aligned} -4x_1 + -3x_2 &= -8 \\ 2x_1 + x_2 &= 2 \\ -2x_2 + x_3 &\geq 0 \end{aligned} \quad .$$

```
library(quadprog)

Dmat       <- matrix(c( 2, -1, 0,
                       -1, 2, -1,
                        0, -1, 2), 3, 3)
dvec       <- c(0, -5, 3)
Amat       <- matrix(c(-4, -3, 0,
                        2, 1, 0,
                        0, -2, 1), 3, 3)
bvec       <- c(-8, 2, 0)
n.eqs      <- 2 # the first two constraints are equalities
sol <- solve.QP(Dmat, dvec, Amat, bvec=bvec, meq=2)
sol$solution # get the (x1, x2, x3) point of minimum

## [1] -1  4  8

sol$value  # get the actual cost function minimum

## [1] 49
```

The minimum value, 49, of the QP solution is attained at $x_1 = -1, x_2 = 4, x_3 = 8$.

When D is a positive definitive matrix, i.e., $X^TDX > 0$, for all non-zero X, the QP problem may be solved in polynomial time. Otherwise, the QP problem is NP-hard. In general, even if D has only one negative eigenvalue, the QP problem is still NP-hard.

The QP function `solve.QP()` expects a positive definitive matrix D.

22.3 General Non-linear Optimization

The package `Rsolnp` provides a special function `solnp()`, which solves the general non-linear programming problem:

$$\min_x f(x)$$

subject to:

$$\begin{aligned} g(x) &= 0 \\ l_h \leq h(x) &\leq u_h \\ l_x \leq x &\leq u_x, \end{aligned}$$

where $f(x)$, $g(x)$, $h(x)$ are all smooth functions.

22.3.1 Dual Problem Optimization

Duality in math really just means having two complementary ways to think about an optimization problem. The *primal problem* represents an optimization challenge in terms of the original decision variable x. The *dual problem*, also called *Lagrange dual*, searches for a lower bound of a minimization problem or an upper bound for a maximization problem. In general, the primal problem may be difficult to analyze, or solve directly, because it may include non-differentiable penalty terms, e.g., l_1 norms, recall LASSO/Ridge regularization in Chap. 18. Hence, we turn to the corresponding *Lagrange dual problem* where the solutions may be more amenable, especially for convex functions, that satisfy the following inequality:

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y).$$

Motivation

Suppose we want to borrow money, x, from a bank, or lender, and $f(x)$ represents the borrowing cost to us. There are natural "design constraints" on money lending. For instance, there may be a cap in the interest rate, $h(x) \leq b$, or we can have many other constraints on the loan duration. There may be multiple lenders, including self-funding, that may "charge" us $f(x)$ for lending us x. *Lenders goals are to maximize profits*. Yet, they can't charge you more than the prime interest rate, plus some premium based on your credit worthiness. Thus, for a given fixed λ, a lender may make us an offer to lend us x aiming to minimize

$$f(x) + \lambda \times h(x).$$

If this cost is not optimized, i.e., minimized, you may be able to get another loan y at lower cost $f(y) < f(x)$, and the funding agency loses your business. If the cost/objective function is minimized, the lender may maximize their profit by varying λ and still get us to sign on the loan.

The customer's strategy represents a *game theoretic interpretation* of the **primal problem**, whereas the **dual problem** corresponds to the strategy of the lender.

In solving complex optimization problems, *duality* is equivalent to existence of a saddle point of the Lagrangian. For convex problems, the double-dual is *equivalent* to the primal problem. In other words, applying the convex conjugate (Fenchel transform) twice returns the *convexification* of the original objective function, which in most situations is the same as the original function.

The *dual of a vector space* is defined as the space of all continuous linear functionals on that space. Let $X = R^n$, $Y = R^m$, $f : X \to R$, and $h : X \to Y$. Consider the following optimization problem:

$$\min_x f(x)$$
$$\text{subject to}$$
$$x \in X$$
$$h(x) \leq 0.$$

Then, this *primal problem* has a corresponding *dual problem*:

$$\min_{\lambda} \inf_{x \in X} (f(x) + \langle \lambda, h(x) \rangle)$$
$$\text{subject to}$$
$$\lambda_i \geq 0, \forall 0 \leq i \leq m.$$

The parameter $\lambda \in R^m$ is an element of the dual space of Y, i.e., Y^*, since the inner product $\langle \lambda, h(x) \rangle$ is a *continuous linear functional on* Y. Here Y is finite dimensional and by the Riesz representation theorem Y^* is isomorphic to Y. Note that in general, for infinite dimensional spaces, Y and Y^* are not guaranteed to be isomorphic.

Example 1: Linear Example

Minimize $f(x, y) = 5x - 3y$, constrained by $x^2 + y^2 = 136$, which has a minimum value of -68 attained at $(-10, 6)$. We will use the `Rsolnp::solnp()` method in this example.

```
# install.packages("Rsolnp")
library(Rsolnp)

fn1 <- function(x) { # f(x, y) = 5x-3y
  5*x[1] - 3*x[2]
}

# constraint z1: x^2+y^2=136
eqn1 <- function(x) {
  z1=x[1]^2 + x[2]^2
  return(c(z1))
}
constraints = c(136)

x0 <- c(1, 1) # setup initial values
sol1 <- solnp(x0, fun = fn1, eqfun = eqn1, eqB = constraints)

##
## Iter: 1 fn: 37.4378   Pars:  30.55472 38.44528
## Iter: 2 fn: -147.9181     Pars:  -6.57051 38.35517
```

```
## Iter: 3 fn: -154.7345     Pars:  -20.10545  18.06907
## Iter: 4 fn: -96.4033  Pars:  -14.71366   7.61165
## Iter: 5 fn: -72.4915  Pars:  -10.49919   6.66517
## Iter: 6 fn: -68.1680  Pars:  -10.04485   5.98124
## Iter: 7 fn: -68.0006  Pars:   -9.99999   6.00022
## Iter: 8 fn: -68.0000  Pars:  -10.00000   6.00000
## Iter: 9 fn: -68.0000  Pars:  -10.00000   6.00000
## solnp--> Completed in 9 iterations
```

```
sol1$values[10]  # sol1$values contains all steps of the iteration algorithm
and the last value is the min value
```

```
## [1] -68
```

```
sol1$pars
```

```
## [1] -10   6
```

Example 2: Quadratic Example

Minimize $f(x, y) = 4x^2 + 10y^2 + 5$ subject to the inequality constraint $0 \leq x^2 + y^2 \leq 4$, which has a minimum value of 5 attained at the origin $(0, 0)$.

```
fn2 <- function(x) {  # f(x, y) = 4x^2 + 10y^2 +5
  4*x[1]^2 + 10*x[2]^2 +5
}

# constraint z1: x^2+y^2 <= 4
ineq2 <- function(x) {
  z1=x[1]^2 + x[2]^2
  return(c(z1))
}

lh <- c(0)
uh <- c(4)

x0 = c(1, 1) # setup initial values
sol2 <- solnp(x0, fun = fn2, ineqfun = ineq2, ineqLB = lh, ineqUB=uh)

##
## Iter: 1 fn: 7.8697    Pars:  0.68437 0.31563
## Iter: 2 fn: 5.6456    Pars:  0.39701 0.03895
## Iter: 3 fn: 5.1604    Pars:  0.200217 0.002001
## Iter: 4 fn: 5.0401    Pars:  0.10011821 0.00005323
## Iter: 5 fn: 5.0100    Pars:  0.0500592618 0.0000006781
## Iter: 6 fn: 5.0025    Pars:   0.02502983706 -0.00000004425
## Iter: 7 fn: 5.0006    Pars:   0.01251500215 -0.00000005034
## Iter: 8 fn: 5.0002    Pars:   0.00625757145 -0.00000005045
## Iter: 9 fn: 5.0000    Pars:   0.00312915970 -0.00000004968
## Iter: 10 fn: 5.0000   Pars:   0.00156561388 -0.00000004983
## Iter: 11 fn: 5.0000   Pars:   0.0007831473 -0.0000000508
## Iter: 12 fn: 5.0000   Pars:   0.00039896484 -0.00000005045
## Iter: 13 fn: 5.0000   Pars:   0.00021282342 -0.00000004897
## Iter: 14 fn: 5.0000   Pars:   0.00014285437 -0.00000004926
## Iter: 15 fn: 5.0000   Pars:   0.00011892066 -0.00000004976
## solnp--> Completed in 15 iterations
```

```
sol2$values
```

```
##  [1] 19.000000  7.869675  5.645626  5.160388  5.040095  5.010024
5.002506
##  [8]  5.000627  5.000157  5.000039  5.000010  5.000002  5.000001
5.000000
## [15]  5.000000  5.000000
```

```
sol2$pars
```

```
## [1]  1.189207e-04 -4.976052e-08
```

There are a number of parameters that control the `solnp` procedure. For instance, `TOL` defines the tolerance for optimality (which impacts the convergence) and `trace=0` turns off the printing of the results at each iteration.

```
ctrl <- list(TOL=1e-15, trace=0)
sol2 <- solnp(x0, fun = fn2, ineqfun = ineq2, ineqLB = lh, ineqUB=uh, contro
l=ctrl)
sol2$pars
```

```
## [1]  1.402813e-08 -5.015532e-08
```

Example 3: More Complex Non-linear Optimization

Let's try to minimize

$$f(X) = -x_1 x_2 x_3$$

subject to

$$4x_1x_2 + 2x_2x_3 + 2x_3x_1 = 100$$
$$1 \leq x_i \leq 10, i = 1, 2, 3$$

```
fn3 <- function(x, ...){
  -x[1]*x[2]*x[3]
}

eqn3 <- function(x, ...){
    4*x[1]*x[2]+2*x[2]*x[3]+2*x[3]*x[1]
}
constraints3 = c(100)

lx <- rep(1, 3)
ux <- rep(10, 3)

pars <- c(2, 1, 7) # setup: Try alternative starting-parameter vector (pars)
ctrl <- list(TOL=1e-6, trace=0)
sol3 <- solnp(pars, fun=fn3, eqfun=eqn3, eqB = constraints3, LB=lx, UB=ux, c
ontrol=ctrl)
sol2$values
```

```
## [1] 19.000000  7.869675  5.645626  5.160388  5.040095  5.010024  5.002506
## [8]  5.000626  5.000157  5.000039  5.000010  5.000002  5.000001  5.000000
##[15]  5.000000  5.000000  5.000000  5.000000  5.000000  5.000000  5.000000
##[22]  5.000000  5.000000  5.000000  5.000000  5.000000  5.000000  5.000000
##[29]  5.000000
```

```
sol3$pars
```

```
## [1] 2.886751 2.886751 5.773505
```

The non-linear optimization is sensitive to the initial parameters (pars), especially when the objective function is not smooth or if there are many local minima. The function `gosolnp()` may be employed to generate initial (guesstimates of the) parameters.

Example 4: Another Linear Example

Let's try another minimization of a linear objective function $f(x, y, z) = 4y - 2z$ subject to

$$\begin{aligned} 2x - y - z &= 2 \\ x^2 + y^2 &= 1. \end{aligned}$$

```
fn4 <- function(x)  # f(x, y, z) = 4y-2z
{
  4*x[2] - 2*x[3]
}

# constraint z1: 2x-y-z  = 2
# constraint z2: x^2+y^2 = 1
eqn4 <- function(x){
  z1=2*x[1] - x[2] - x[3]
  z2=x[1]^2 + x[2]^2

  return(c(z1, z2))
}
constraints4 <- c(2, 1)

x0 <- c(1, 1, 1)
ctrl <- list(trace=0)
sol4 <- solnp(x0, fun = fn4, eqfun = eqn4, eqB = constraints4, control=ctrl)
sol4$values

## [1]   2.000000  -5.078795 -11.416448  -5.764047  -3.584894  -3.224531
## [7]  -3.211165  -3.211103  -3.211103

sol4$pars

## [1]  0.55470019 -0.83205030 -0.05854932
```

The materials in the linear algebra and matrix computing, Chap. 5, and the regularized parameter estimation, Chap. 18, provide additional examples of least squares parameter estimation, regression, and regularization.

22.4 Manual Versus Automated Lagrange Multiplier Optimization

Let's manually implement the Lagrange Multipliers procedure and then compare the results to some optimization examples obtained by automatic R function calls. The latter strategies may be more reliable, efficient, flexible, and rigorously validated.

The manual implementation provides a more direct and explicit representation of the actual optimization strategy.

We will test a simple example of an objective function:

$$f(x, y, z) = 4y - 2z + x^2 + y^2,$$

subject to two constraints:

$$\begin{aligned} 2x - y - z &= 2 \\ x^2 + y^2 + z &= 1. \end{aligned}$$

The R package `numDeriv` may be used to calculate numerical approximations of partial derivatives.

```
# define the main Lagrange Multipliers Optimization strategy from scratch
require(numDeriv)

lagrange_multipliers <- function(x, f, g) {  # Objective/cost function,
f, and constrains, g
    k <- length(x)
    l <- length(g(x))

    # Compute the derivatives
    grad_f <- function(x) { grad(f, x) }

    # g, representing multiple constrains, is a vector-valued function:
its first derivative is a matrix
    grad_g <- function(x) {  jacobian(g, x) }

    # The Lagrangian is a scalar-valued function:
    #   L(x, lambda) = f(x) - lambda * g(x)
    # whose first derivative roots give the optimal solutions
    #   h(x, lambda) = c( f'(x) - lambda * g'(x), - g(x) ).
    h <- function(y) {
        c(grad_f(y[1:k]) - t(y[-(1:k)]) %*% grad_g(y[1:k]), -g(y[1:k]))
    }

    # To find the roots of the first derivative, we can use Newton's method:
    # iterate  y <- y - h'(y)^{-1} h(y)  until certain convergence criterion
is met   # e.g., (\delta <= 1e-6)
    grad_h <- function(y) { jacobian(h, y) }

    y <- c(x, rep(0, l))
    previous <- y + 1
    while(sum(abs(y-previous)) > 1e-6) {
        previous <- y
        y <- y - solve( grad_h(y), h(y) )
    }
    y[1:k]
}

x <- c(0, 0, 0)

# Define the objective cost function
fn4 <- function(x)  # f(x, y, z) = 4y-2z + x^2+y^2
```

```
{
  4*x[2] - 2*x[3] + x[1]^2+ x[2]^2
  #sum(x^2)
}
# check the derivative of the objective function
grad(fn4, x)

## [1]  0  4 -2

# define the domain constraints of the problem
# constraint z1: 2x-y-z  = 2
# constraint z2: x^2+y^2 +z = 1
eqn4 <- function(x){
  z1=2*x[1] - x[2] - x[3] -2
  z2=x[1]^2 + x[2]^2 + x[3] -1
return(c(z1, z2))
}

# Check the Jacobian of the constraints
jacobian(eqn4, x)

##      [,1] [,2] [,3]
## [1,]    2   -1   -1
## [2,]    0    0    1

# Call the Lagrange-multipliers solver

# check one step of the algorithm
k <- length(x)
l <- length(eqn4(x));
h <- function(x) {
    c(grad(fn4, x[1:k]) - t(-x[(1:2)]) %*% jacobian(eqn4, x[1:k]),
-eqn4(x[1:k]))
}
jacobian(h, x)

##      [,1] [,2]          [,3]
## [1,]    4    0  0.000000e+00
## [2,]   -1    2  5.482583e-15
## [3,]   -1    1  0.000000e+00
## [4,]   -2    1  1.000000e+00
## [5,]    0    0 -1.000000e+00

# Lagrange-multipliers solver for f(x, y, z) subject to g(x, y, z)
lagrange_multipliers(x, fn4, eqn4)

## [1]  0.3416408 -1.0652476 -0.2514708
```

Now, let's double-check the above manual optimization results against the automatic `solnp` solution minimizing

$$f(x, y, z) = 4y - 2z + x^2 + y^2$$

subject to:

$$\begin{aligned} 2x - y - z &= 2 \\ x^2 + y^2 &= 1. \end{aligned}$$

```
library(Rsolnp)
fn4 <- function(x)  # f(x, y, z) = 4y-2z + x^2+y^2
{
  4*x[2] - 2*x[3] + x[1]^2+ x[2]^2
}

# constraint z1: 2x-y-z  = 2
# constraint z2: x^2+y^2 +z = 1
eqn4 <- function(x){
  z1=2*x[1] - x[2] - x[3]
  z2=x[1]^2 + x[2]^2 + x[3]
  return(c(z1, z2))
}
constraints4 <- c(2, 1)

x0 <- c(1, 1, 1)
ctrl <- list(trace=0)
sol4 <- solnp(x0, fun = fn4, eqfun = eqn4, eqB = constraints4, control=ctrl)
sol4$values

## [1]  4.0000000 -0.1146266 -5.9308852 -3.7035124 -2.5810141 -2.5069444
## [7] -2.5065779 -2.5065778 -2.5065778
```

The results of both (manual and automated) experiments identifying the optimal (x, y, z) coordinates minimizing the objective function $f(x, y, z) = 4y - 2z + x^2 + y^2$ are in agreement.

- *Manual* optimization: `lagrange_multipliers(x, fn4, eqn4)`: 0.3416408 −1.0652476 −0.2514708.
- *Automated* optimization: `solnp(x0, fun = fn4, eqfun = eqn4, eqB = constraints4, control = ctrl)`: 0.3416408 −1.0652476 −0.2514709.

22.5 Data Denoising

Suppose we are given x_{noisy} with n noise-corrupted data points. The noise may be additive ($x_{noisy} \sim x + \epsilon$) or not additive. We may be interested in denoising the signal and recovering a version of the original (unobserved) dataset x, potentially as a smoothed representation of the original (uncorrupted) process. Smoother signals suggest less (random) fluctuations between neighboring data points.

One objective function we can design to denoise the observed signal, x_{noisy}, may include a *fidelity term* and a *regularization term*; see the regularized linear modeling in Chap. 18.

`Total variation` denoising assumes that for each time point t, the observed noisy data

$$\underbrace{x_{noisy}(t)}_{\text{observed signal}} \sim \underbrace{x(t)}_{\text{native signal}} + \underbrace{\epsilon(t)}_{\text{random noise}} .$$

To recover the *native signal*, $x(t)$, we can optimize ($\text{argmin}_x f(x)$) the following objective cost function:

$$f(x) = \underbrace{\frac{1}{2}\sum_{t=1}^{n-1} \| y(t) - x_{noisy}(t)\|^2}_{\text{fidelity term}} + \lambda \underbrace{\sum_{t=2}^{n-1} | x(t) - x(t-1) |}_{\text{regularization term}},$$

where λ is the regularization smoothness parameter, $\lambda \to 0 \implies y \to x_{noisy}$. Minimizing $f(x)$ provides a minimum total-variation solution to the data denoising problem.

Below is an example illustrating total variation (TV) denoising using a simulated noisy dataset. We start by generating an oscillatory noisy signal. Then, we compute several smoothed versions of the noisy data, plot the initial and smoothed signals, define and optimize the TV denoising objective function, which is a mixture of a fidelity term and a regularization term (Fig. 22.4).

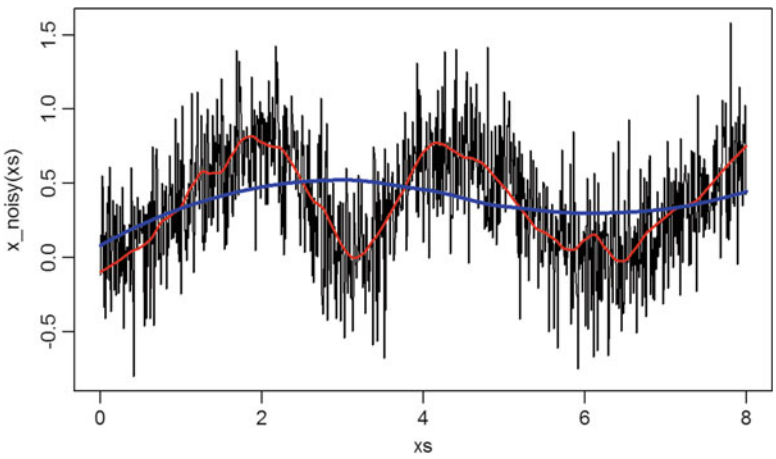

Fig. 22.4 Denoising by smoothing, raw noisy data and two smoothed models (`loess`)

```
n  <- 1000
x <- rep(0, n)
xs <- seq(0, 8, len=n)   # seq(from = 1, to = 1, le ngth)
noise_level = 0.3  # sigma of the noise, try varying this noise -level

# here is where we add  the zero-mean noise
set.seed(1234)

x_noisy <- function (x) {
  sin(x)^2/(1.5+cos(x)) + rnorm(length(x), 0, noise_level)
}

# initialize the manual denoised signal
x_denoisedManu <- rep(0, n)

df <- as.data.frame(cbind(xs, x_noisy(xs)))
# loess fit a polynomial surface determined by numerical predictors,
# using local fitting
poly_model1 <- loess(x_noisy(xs) ~ xs, span=0.1, data=df) # tight model
poly_model2 <- loess(x_noisy(xs) ~ xs, span=0.9, data=df) # smoother model
# To see some of numerical results of hte model -fitting:
# View(as.data.frame(cbind(xs, x_noisy, predict (poly_model1))))

plot(xs, x_noisy(xs), type='l')
lines(xs, poly_model1$fitted,  col="red", lwd=2)
lines(xs, poly_model2$fitted,  col="blue", lwd=3)
```

Next, let's initiate the parameters, define the objective function and optimize it, i.e., estimate the parameters that minimize the cost function as a mixture of fidelity and regularization terms (Fig. 22.5).

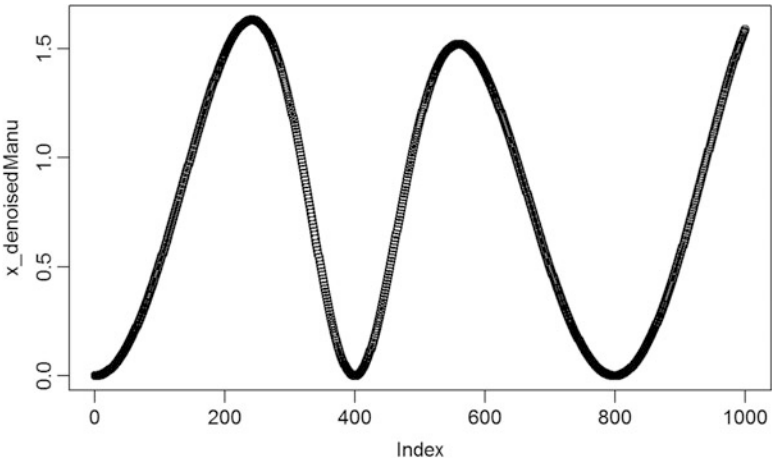

Fig. 22.5 Manual denoising signal recovery using non-linear constaint optimization (`solnp`)

```
# initialization of parameters
betas_0 <- c(0.3, 0.3, 0.5, 1)
betas <- betas_0

# Denoised model
x_denoised <- function(x, betas) {
  if (length(betas) != 4) {
    print(paste0("Error!!! length(betas)=", length(betas), " != 4!!! Exiting
...."))
    break();
  }
  # print(paste0("   .... betas = ", betas, "...\n"))
  # original noise function definition: sin(x)^2/(1.5+cos(x))
  return((betas[1]*sin(betas[2]*x)^2)/(betas[3]+cos(x)))
}

library(Rsolnp)

fidelity <- function(x, y) {
  sqrt((1/length(x)) * sum((y - x)^2))
}

regularizer <- function(betas) {
  reg <- 0
  for (i in 1:(length(betas-1))) {
      reg <- reg + abs(betas[i])
  }
  return(reg)
}

# Objective Function
objective_func <- function(betas)  {
# f(x) = 1/2 * \sum_{t=1}^{n-1} {|y(t) - x_{noisy}(t)\|^2} + \lambda *
\sum_{t=2}^{n-1} | x(t) - x(t-1)|
  fid <- fidelity(x_noisy(xs), x_denoised(xs, betas))
  reg <- abs(betas[4])*regularizer(betas)
  error <- fid + reg
  # uncomment to track the iterative optimization state
  # print(paste0(".... Fidelity =", fid, " ... Regularizer = ", reg, " ...
TotalError=", error))
  # print(paste0("....betas=(",betas[1],",",betas[2],",",betas[3],",",betas
[4],")"))
  return(error)
}

# inequality constraint forcing regularization parameter lambda=beta[4]>0
inequalConstr <- function(betas){
   betas[4]
}
inequalLowerBound <- 0; inequalUpperBound <- 100

# should we list the value of the objective function and the parameters at
every iteration (default trace=1; trace=0 means no interim reports)
# constraint problem
# ctrl <- list(trace=0, tol=1e-5)  ## could specify: outer.iter=5,
inner.iter=9)
set.seed(121)
```

```
sol_lambda <- solnp(betas_0, fun = objective_func, ineqfun = inequalConstr,
ineqLB = inequalLowerBound, ineqUB = inequalUpperBound, control=ctrl)

# unconstraint optimization
# ctrl <- list(trace=1, tol=1e-5)  ## could specify: outer.iter=5,
inner.iter=9)
# sol_lambda <- solnp(betas_0, fun = denoising_func, control=ctrl)

# suppress the report of the the functional values (too many)
# sol_lambda$values

# reprot the optimal parameter estimates (betas)
sol_lambda$pars

## [1] 2.5649689 0.9829681 1.7605481 0.9895268

# Reconstruct the manually-denoised signal using the optimal betas
betas <- sol_lambda$pars
x_denoisedManu <- x_denoised(xs, betas)
print(paste0("Final Denoised Model:", betas[1], "*sin(", betas[2],
"*x)^2/(", betas[3], "+cos(x)))"))

## [1] "Final Denoised Model:2.56496893433154*sin(0.982968123322892*x)^2/(1.
76054814253387+cos(x)))"

plot(x_denoisedManu)
```

Finally, we can validate our manual denoising protocol against the automated TV denoising using the R package tvd (Fig. 22.6).

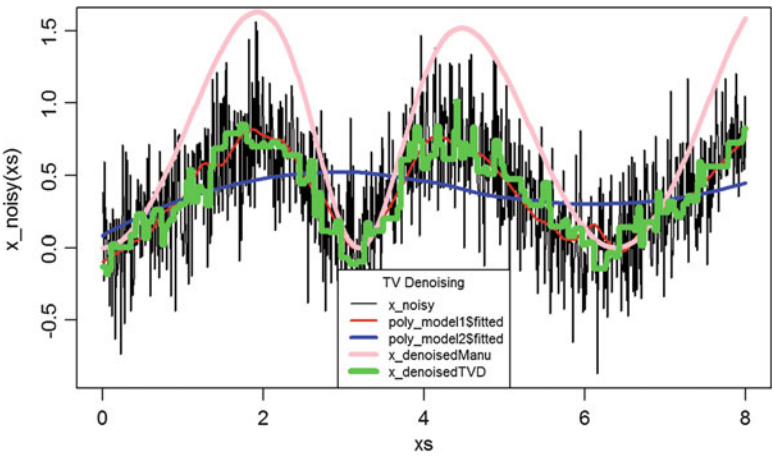

Fig. 22.6 Plot of the observed noisy data and four alternative denoised reconstructions

```
# install.packages("tvd")
library("tvd")

lambda_0 <- 0.5
x_denoisedTVD <- tvd1d(x_noisy(xs), lambda_0, method = "Condat")
# lambda_o is the total variation penalty coefficient
# method is a string indicating the algorithm to use for denoising.
# Default method is "Condat"

# plot(xs, x_denoisedTVD, type='l')
plot(xs, x_noisy(xs), type='l')
lines(xs, poly_model1$fitted, col="red", lwd=2)
lines(xs, poly_model2$fitted, col="blue", lwd=3)
lines(xs, x_denoisedManu, col="pink", lwd=4)
lines(xs, x_denoisedTVD, col="green", lwd=5)
# add a legend
legend("bottom", c("x_noisy", "poly_model1$fitted", "poly_model2$fitted",
"x_denoisedManu", "x_denoisedTVD"), col=c("black", "red", "blue", "pink",
"green"), lty=c(1,1, 1,1), cex=0.7, lwd= c(1,2,3,4,5), title="TV Denoising")
```

22.6 Assignment: 22. Function Optimization

22.6.1 Unconstrained Optimization

Apply `optim()` to solve the following unconstrained optimization problems:

1. $\min_x f(x) = x^4$.
2. $\max_x \left(2\sin x - \frac{x^2}{10}\right)$.
3. $\max_{x,\, y}(2xy + 2x - x^2 - 2y^2)$.

22.6.2 Linear Programming (LP)

Solve the following LP problem:

$$\max_{x_1, x_2, x_3, x_4} (x_1 + 2x_2 + 3x_3 + 4x_4 + 5)$$

subject to:

$$\begin{cases} 4x_1 + 3x_2 + 2x_3 + x_4 & \le 10 \\ x_1 - x_3 + 2x_4 & = 2 \\ x_1 + x_2 + x_3 + x_4 & \ge 1 \\ x_1 \ge 0, x_3 \ge 0, x_4 & \ge 0 \end{cases}.$$

Apply `lpSolveAPI` and `Rsolnp` and compare the results.

22.6.3 Mixed Integer Linear Programming (MILP)

Apply `lpSolveAPI` to solve the following MILP problem:

$$\min_{x_1, x_2} 4x_1 + 6x_2$$

subject to:

$$\begin{cases} 2x_1 + 2x_2 & \geq 5 \\ x_1 - x_2 & \leq 1 \\ x_1, x_2 & \geq 0 \\ x_1, x_2 & \in \text{integers} \end{cases}.$$

22.6.4 Quadratic Programming (QP)

Solve the following QP problem:

$$\min_{x_1, x_2} 2x_1^2 + x_2^2 + x_1 x_2 + x_1 + x_2$$

subject to:

$$\begin{cases} x_1 + x_2 & = 1 \\ x_1, x_2 & \geq 0 \end{cases}.$$

- Apply `quadprog` to solve the QP.
- Use `Rsolnp` to solve the QP.
- Write the Lagrange multiplier form.
- Apply `numDeriv` to solve this Lagrange multiplier optimization manually.
- Compare the three versions of the results above.

22.6.5 Complex Non-linear Optimization

Solve the following nonlinear problem:

$$\min_{x_1, x_2} \left(100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2\right)$$

subject to $x_1, x_2 \geq 0$.

22.6.6 Data Denoising

Based on the signal denoising example presented in this chapter, try to change the noise level, replicate the denoising process, and report your findings.

References

Cortez, P. (2014) *Modern Optimization with R*, Springer, ISBN 3319082639, 9783319082639.

CRAN Optimization & Math Programming Site provides details about a broad range of R optimization functions.

Vincent Zoonekynd's Optimization Blog http://zoonek.free.fr/blosxom/R/2012-06-01_Optimization.html.