Optimization problems constrained by parameter sums

John C. Nash, retired professor, Telfer School of Management, University of Ottawa
Gabor Grothendieck, GKX Group
Ravi Varadhan, Johns Hopkins University Medical School

2023-7-31

Abstract

This article presents a discussion of optimization problems where the objective function $f(\mathbf{x})$ has parameters that are constrained by some scaling, so that $q(\mathbf{x}) = constant$, where this function $q()$ involves a sum of the parameters, their squares, or similar simple function. Our focus is on ways to use standardized optimization programs to solve such problems rather than specialized codes.

Background

We consider problems where we want to minimize or maximize a function subject to a constraint that the sum of some function of the parameters, e.g., their sum of squares, must equal some constant. Because these problems all have an objective that is dependent on a scaled set of parameters where the scale is defined by a sum, sum of squares, or similar sum of the paramters, we will refer to them as **sumscale** optimization problems.

We have observed questions about problems like this on the R-help mailing list:

```
Jul 19, 2012 at 10:24 AM, Linh Tran <Tranlm```berkeley.edu> wrote:
> Hi fellow R users,
>
> I am desperately hoping there is an easy way to do this in R.
>
> Say I have three functions:
>
> f(x) = x^2
> f(y) = 2y^2
> f(z) = 3z^2
>
> constrained such that x+y+z=c (let c=1 for simplicity).
>
> I want to find the values of x,y,z that will minimize
f(x) + f(y) + f(z).
```

If the parameters x, y and z are non-negative, this problem can actually be solved as a Quadratic Program. We revisit this problem at the end of this article.

Other examples of this type of objective function are:

- The maximum volume of a regular polyhedron where the sum of the lengths of the sides is fixed.
- The minimum negative log likelihood for a multinomial model.
- The Rayleigh Quotient for the maximal or minimal eigensolutions of a matrix, where the eigenvectors should be normalized so the square norm of the vector is 1.

- The minimum of the extended Rosenbrock function of the form given by the `adagio` package (Borchers (2022)) on the unit ball, that is, where the sum of squares of the parameters is 1.

For the moment, let us consider a basic example, which is

Problem A: Minimize $(-\prod \mathbf{x})$ subject to $\sum \mathbf{x} = 1$

This is a very simplified version of the multinomial maximum likelihood problem.

Difficulties using general optimization with sumscale problems

Let us use the basic example above to consider how we might formulate Problem A for a computational solution in R.

One possibility is to select one of the parameters and solve for it in terms of the others. Let this be the last parameter x_n, so that the set of parameters to be optimized is $\mathbf{y} = (x_1, x_1, ..., x_{n-1})$ where n is the original size of our problem. We now have the unconstrained problem

$$minimize(-(\prod \mathbf{y}) * (1 - \sum y))$$

This is easily coded and tried. We will use a very simple start, namely, the sequence $1, 2, ..., (n-1)$ scaled by $1/n^2$. We will also specify that the gradient is to be computed by a central approximation (see `gcentral.R` from package **optimx**).

```
require(optimx, quietly=TRUE)
pr <- function(y) {
- prod(y)*(1-sum(y))
}
cat("test the simple product for n=5\n")
```

```
## test the simple product for n=5
```

```
meth <- c("Nelder-Mead", "BFGS")
n<-5
  st<-1:(n-1)/(n*n)
   ans<-opm(st, pr, gr="grcentral", control=list(trace=0))
   ao<-summary(ans,order=value)
print(ao)
```

```
##                    p1 s1        p2 s2        p3 s3        p4 s4    value fevals
## BFGS        0.2000000    0.1999986    0.2000037    0.1999985    -0.00032    102
## Nelder-Mead 0.2000034    0.1999983    0.2000017    0.2000021    -0.00032    331
##             gevals hevals conv kkt1 kkt2 xtime
## BFGS            96      0    0 TRUE TRUE 0.002
## Nelder-Mead      0      0    0 TRUE TRUE 0.016
```

While these codes work fine for small n, it is fairly easy to see that there are computational problems as the size of the problem increases. Since the sum of the parameters is constrained to be equal to 1, the parameters are of the order of $1/n$, and the function therefore of the order of $1/(n^n)$, which underflows around $n = 144$ in R.

Other formulations

Traditionally, statisticians solve maximum likelihood problems by **minimizing** the negative log-likelihood. That is, the objective function is formed as (-1) times the logarithm of the likelihood. This converts our product to a sum. Choosing the first parameter to be the one determined by the summation constraint, we can write the function and gradient quite easily. As programs that try to find the minimum may change the parameters so that logarithms of non-positive numbers are attempted, we have put some safeguards in the function `nll`. At this point we have assumed the gradient calculation is only attempted if the function can be computed satisfactorily, so we have not put safeguards in the gradient.

```
nll <- function(y) {
  if ((any(y <= 10*.Machine$double.xmin)) || (sum(y)>1-.Machine$double.eps))
         .Machine$double.xmax
  else   - sum(log(y)) - log(1-sum(y))
}
nll.g <- function(y) { - 1/y + 1/(1-sum(y))} # so far not safeguarded
n<-5
x0<-(2:n)/n^2
library(numDeriv)
dx0<-nll.g(x0)
dx0n<-grad(nll, x0)
cat("Max Abs diff between analytic and approx. gradient =",max(abs(dx0-dx0n)),"\n")
```

```
## Max Abs diff between analytic and approx. gradient = 5.92224e-10
```

We can easily try several optimization methods using the `optimx` package. Here are the calls, which overall did not perform as well as we would like. Note that we do not ask for `method="ALL"` as we found that some of the methods, in particular those using Powell's quadratic approximation methods, seem to get "stuck". Instead, we have specified a list of methods `mset`, though some of these also run into scaling problems.

```
require(optimx, quietly=TRUE)
mset<-c("L-BFGS-B", "BFGS", "CG", "spg", "ucminf", "nlm", "nlminb", "nvm", "ncg", "tnewt")
# numerical approximation using forward difference
a5<-opm(2:n/n^2, nll, gr="grfwd", method=mset,
          control=list(dowarn=FALSE, kkt=FALSE,trace=0))
```

```
## Error in optim(par = par, fn = efn, gr = egr, method = method, hessian = FALSE,  :
##   non-finite value supplied by optim
## Gradient check details:  max. relative difference in gradients=  1.100932
##
##   analytic gradient: -10.22727 -12.87881 -13.63639 -13.6364
##
##   numerical gradient: -10.22727 -6.060606 -3.977273 -2.727273Error in BB::spg(par = spar, fn = efn, g
##   Analytic gradient does not seem correct! See comparison above. Fix it, remove it, or increase chec
## Error in nlm(f = fghfn, p = spar, iterlim = iterlim, print.level = print.level) :
##   probable coding error in analytic gradient
```

```
summary(a5, order=value)
```

```
##                  p1 s1        p2 s2        p3 s3        p4 s4         value
## nvm       0.2000002    0.2000000    0.1999998    0.1999996     8.047190e+00
## tnewt     0.1999999    0.1999999    0.2000002    0.1999995     8.047190e+00
## ncg       0.2000008    0.2000006    0.1999990    0.1999996     8.047190e+00
## CG        0.2000007    0.2000008    0.1999991    0.1999989     8.047190e+00
## nlminb    0.2000054    0.1999971    0.1999976    0.1999975     8.047190e+00
## BFGS      0.1999961    0.1999887    0.2000367    0.2000092     8.047190e+00
## ucminf    0.1736718    0.1929237    0.2013754    0.2203217     8.063850e+00
## L-BFGS-B         NA           NA           NA           NA    8.988466e+307
## spg              NA           NA           NA           NA    8.988466e+307
## nlm              NA           NA           NA           NA    8.988466e+307
##           fevals gevals hevals conv kkt1 kkt2 xtime
## nvm          130     36      0    0   NA   NA 0.005
## tnewt        167    166      0    0   NA   NA 0.007
## ncg         2245    314      0    1   NA   NA 0.017
## CG          362     94      0    0    NA    NA 0.003
## nlminb       43     29      0    0    NA    NA 0.001
## BFGS        231     31      0    0    NA    NA 0.001
## ucminf       12     12      0    0    NA    NA 0.003
## L-BFGS-B      2      2      0 9999    NA    NA 0.000
## spg          35      1      0 9999    NA    NA 0.004
## nlm           6      6      0 9999    NA    NA 0.000
```

```
# analytical gradient
a5g<-opm(2:n/n^2, nll, nll.g, method=mset,
          control=list(dowarn=FALSE,kkt=FALSE, trace=0))
```

```
## Error in optim(par = par, fn = efn, gr = egr, method = method, hessian = FALSE,  :
##   non-finite value supplied by optim
```

```
summary(a5g, order=value)
```

```
##                  p1  s1         p2  s2         p3  s3         p4  s4          value
## ucminf    0.2000000    0.2000000    0.2000000    0.2000000     8.047190e+00
## nvm       0.2000000    0.2000000    0.2000000    0.2000000     8.047190e+00
## ncg       0.2000000    0.2000000    0.2000000    0.2000000     8.047190e+00
## tnewt     0.2000000    0.2000000    0.2000000    0.2000000     8.047190e+00
## spg       0.2000000    0.2000000    0.2000000    0.2000000     8.047190e+00
## CG        0.2000000    0.2000000    0.2000000    0.2000000     8.047190e+00
## nlm       0.2000006    0.1999995    0.2000000    0.2000000     8.047190e+00
## BFGS      0.2000007    0.1999989    0.2000012    0.1999981     8.047190e+00
## nlminb    0.2000004    0.1999990    0.1999989    0.1999992     8.047190e+00
## L-BFGS-B         NA           NA           NA           NA    8.988466e+307
##           fevals gevals hevals conv kkt1 kkt2 xtime
## ucminf        14     14      0    0    NA    NA 0.001
## nvm           31     12      0    0    NA    NA 0.001
## ncg           29     12      0    0    NA    NA 0.001
## tnewt         27     26      0    0    NA    NA 0.001
## spg           51     15      0    0    NA    NA 0.001
## CG            59     21      0    0    NA    NA 0.000
## nlm           19     19      0    0    NA    NA 0.001
## BFGS          33      9      0    0    NA    NA 0.000
## nlminb        24     12      0    0    NA    NA 0.000
## L-BFGS-B       2      2      0 9999    NA    NA 0.003
```

```
# analytical gradient and bounds on parameters
a5gb<-opm(2:n/n^2, nll, nll.g, lower=0, upper=1, method=mset,
          control=list(dowarn=FALSE,kkt=FALSE,trace=0))
```

```
## Warning in opm(2:n/n^2, nll, nll.g, lower = 0, upper = 1, method = mset, :
## method requested does not handle bounds
```

```
## Error in optim(par = par, fn = efn, gr = egr, lower = lower, upper = upper,  :
##   non-finite value supplied by optim
```

```
summary(a5gb, order=value)
```

```
##                  p1  s1        p2  s2         p3  s3         p4  s4          value
## nvm       0.2000000    0.200000    0.2000000    0.2000000     8.047190e+00
## tnewt     0.2000000    0.200000    0.2000000    0.2000000     8.047190e+00
## ncg       0.2000000    0.200000    0.2000000    0.2000000     8.047190e+00
## nlminb    0.2000004    0.199999    0.1999989    0.1999992     8.047190e+00
## L-BFGS-B         NA          NA           NA           NA    8.988466e+307
##          fevals gevals hevals conv kkt1 kkt2 xtime
## nvm          28     14      0    0   NA   NA 0.002
## tnewt        20     19      0    0   NA   NA 0.001
## ncg          26     12      0    0   NA   NA 0.000
## nlminb       24     12      0    0   NA   NA 0.000
## L-BFGS-B      2      2      0 9999   NA   NA 0.000
```

Most, but not all, of the methods find the solution for the $n = 5$ case. The exception (L-BFGS-B) is due to the optimization method trying to compute the gradient where sum(x) is greater than 1. We have not tried to determine the source of this particular issue. However, it is almost certainly a consequence of too large a step. The particular form of $log(1 - sum(x))$ is undefined once the argument of the logarithm is negative. Indeed, this is the basis of logarithmic barrier functions for constraints. There is a similar issue with the $n - 1$ parameters near zero. Negative values will cause difficulties.

Numerical gradient approximations will similarly fail, particularly as step sizes are often of the order of 1E-7 in size. There is generally no special check within numerical gradient routines to apply bounds. Note also that a lower bound of 0 on parameters is not adequate, since $log(0)$ is undefined. Choosing a bound large enough to avoid the logarithm of a zero or negative argument while still being small enough to allow for parameter optimization is non-trivial.

Transformed problems or parameters

When problems give difficulties, it is common to re-formulate them by transformations of the function or the parameters.

Using a projection

Objective functions defined by $(-1) * \prod \mathbf{x}$ or $(-1) * \sum log(\mathbf{x})$ will change with the scale of the parameters. Moreover, the constraint $\sum \mathbf{x} = 1$ effectively imposes the scaling

$$\mathbf{x}_{scaled} = \mathbf{x} / \sum \mathbf{x}$$

The optimizer `spg` from package `BB` allows us to project our search direction to satisfy constraints. Thus, we could use the following approach.

```
require(BB, quietly=TRUE)
nllrv <- function(x) {- sum(log(x))}
nllrv.g <- function(x) {- 1/x }
proj <- function(x) {x/sum(x)}
n <- 5
aspg <- spg(par=(1:n)/n^2, fn=nllrv, gr=nllrv.g, project=proj,
            control=list(trace=TRUE, triter=1))
```

```
## iter:  0  f-value:  11.30689  pgrad:  0.3607565
## iter:  1  f-value:  8.881109  pgrad:  0.3554228
## iter:  2  f-value:  8.095091  pgrad:  0.09254548
## iter:  3  f-value:  8.048691  pgrad:  0.018282
## iter:  4  f-value:  8.047191  pgrad:  0.0004657523
## iter:  5  f-value:  8.04719  pgrad:  7.331566e-06
```

```
aspgn <- spg(par=(1:n)/n^2, fn=nllrv, project=proj,
             control=list(trace=TRUE, triter=1)) # using internal grad approx.
```

```
## iter:  0  f-value:  11.30689  pgrad:  0.1333334
## iter:  1  f-value:  8.04719  pgrad:  1.225824e-07
```

```
cat("F_optimal: with gradient=",aspg$value,"  num. approx.=",aspgn$value,"\n")
```

```
## F_optimal: with gradient= 8.04719   num. approx.= 8.04719
```

```
pbest<-rep(1/n, n)
cat("fbest = ",nllrv(pbest),"  when all parameters = ", pbest[1],"\n")
```

```
## fbest =  8.04719   when all parameters =  0.2
```

```
cat("deviations:  with gradient=",max(abs(aspg$par-pbest)),
    "   num. approx.=",max(abs(aspg$par-pbest)),"\n")
```

```
## deviations:  with gradient= 3.81244e-06    num. approx.= 3.81244e-06
```

Here the projection `proj` is the key to success of method `spg`. Other methods (as yet) do not have the flexibility to impose the projection directly. We would need to carefully build the projection into the function(s) and/or the method codes. This was done by Geradin (1971) for the Rayleigh quotient problem, but requires a number of changes to the program code. Why `spg()` does (for this case) much better using the internal numerical gradient approximation than the analytic gradient is an open question.

log() transformation of parameters

A common method to ensure parameters are positive is to transform them. In the present case, optimizing over parameters that are the logarithms of the parameters above ensures we have positive arguments to most of the elements of the negative log likelihood. Here is the code. Note that the parameters used in optimization are "lx" and not x.

```
enll <- function(lx) {
    x<-exp(lx)
    fval<-  - sum( log( x/sum(x) ) )
}
enll.g <- function(lx){
    x<-exp(lx)
    g<-length(x)/sum(x) - 1/x
    gval<-g*exp(lx)
}
```

But where is our constraint? Here we have noted that we could define the objective function only to within the scaling $\mathbf{x}/\sum(\mathbf{x})$. There is a minor nuisance, in that we need to re-scale our parameters after solution to have them in a standard form. This is most noticeable if one uses `optimx` and displays the results of `all.methods`. In the following, we extract the best solution for the 5-parameter problem.

```
require(optimx, quietly=TRUE) # just to be sure
st<-1:5/10 # 5 parameters, crude scaling to start
a5x<-opm(st, enll, enll.g, method="MOST", control=list(trace=0))
a5x<-a5x[order(a5x$value),]
cat("Proposed best solution has minimum=",a5x[1,length(st)+1],"\n")
```

```
## Proposed best solution has minimum= 8.04719
```

```
cat("Coeffs:")
```

```
## Coeffs:
```

```
print(a5x[1,1:length(st)])
```

```
##         p1  p2  p3  p4  p5
## nlminb 0.3 0.3 0.3 0.3 0.3
```

While there are reasons to think that the indeterminacy might upset the optimization codes, in practice, the objective and gradient above are generally well-behaved, though they did reveal that tests of the size of the gradient used, in particular, to decide to terminate iterations in `Rcgmin()` were too hasty in stopping progress for problems with larger numbers of parameters. A user-specified tolerance is now allowed; for example `control=list(tol=1e-12)`.

Let us try a larger problem in 100 parameters.

```
require(optimx, quietly=TRUE)
st<-1:100/1e3 # large
stenll<-enll(st)
cat("Initial function value =",stenll,"\n")
```

```
## Initial function value = 460.5587
```

```
tym<-system.time(acgbig<-Rcgmin(st, enll, enll.g,
                                  control=list(trace=0, tol=1e-32)))[[3]]
cat("Time = ",tym,"  fval=",acgbig$value,"\n")
```

```
## Time =  0.017   fval= 460.517
```

```
xnor<-acgbig$par/sum(acgbig$par)
print(xnor)
```

```
##   [1] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
##  [16] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
##  [31] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
##  [46] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
##  [61] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
##  [76] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
##  [91] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
```

One worrying aspect of the solution is that the objective function at the start and end differ by a tiny amount.

Another transformation

A slightly different transformation or projection is inspired by spherical coordinates.

```
library(optimx)
proj2 <- function(theta) {
    theta2 <- theta^2
    s2 <- theta2 / (1 + theta2)
    cumprod(c(1, s2)) * c(1-s2, 1)
 }
obj <- function(theta) - sum(log(proj2(theta)))
 n <- 5
 ans <- spg(seq(n-1), obj)
```

```
## iter:  0  f-value:  11.15175  pgrad:  3
## iter:  10  f-value:  8.78015  pgrad:  0.5806909
## iter:  20  f-value:  8.04719  pgrad:  3.925749e-06
```

```
 proj2(ans$par)
```

```
## [1] 0.2000000 0.2000007 0.2000002 0.1999996 0.1999995
```

```
n<-100
# check
obj(seq(n-1))
```

```
## [1] 843.9598
```

```
ans100 <- spg(seq(n-1), obj, control=list(trace=FALSE), quiet=TRUE)
ans100$value
```

```
## [1] 460.517
```

```
proj2( (ans100$par) )
```

```
##   [1] 0.010000000 0.010000002 0.009999998 0.009999998 0.010000001 0.010000001
##   [7] 0.009999999 0.010000001 0.010000001 0.010000001 0.010000001 0.009999999
##  [13] 0.010000002 0.010000001 0.010000001 0.009999999 0.009999999 0.010000002
##  [19] 0.010000001 0.010000000 0.010000003 0.010000002 0.010000003 0.010000003
##  [25] 0.010000002 0.010000000 0.010000001 0.010000003 0.010000000 0.010000000
##  [31] 0.010000003 0.009999998 0.010000001 0.010000002 0.009999999 0.009999999
##  [37] 0.009999998 0.010000000 0.009999999 0.009999998 0.010000001 0.010000002
##  [43] 0.009999999 0.010000001 0.010000002 0.010000000 0.010000002 0.009999999
##  [49] 0.010000004 0.010000002 0.010000002 0.009999999 0.009999998 0.010000001
##  [55] 0.010000002 0.010000002 0.009999999 0.010000000 0.010000000 0.010000000
##  [61] 0.010000002 0.010000000 0.009999999 0.009999999 0.010000000 0.009999999
##  [67] 0.009999999 0.009999999 0.009999999 0.009999999 0.009999999 0.009999999
##  [73] 0.009999999 0.009999999 0.009999999 0.009999999 0.009999999 0.009999999
##  [79] 0.010000000 0.009999999 0.009999999 0.009999999 0.009999998 0.010000000
##  [85] 0.009999999 0.009999999 0.009999999 0.009999999 0.010000000 0.009999999
##  [91] 0.009999999 0.010000000 0.009999999 0.009999988 0.010000001 0.010000001
##  [97] 0.010000001 0.010000007 0.010000018 0.009999973
```

Since this transformation is embedded into the objective function, we could run all the optimizers in `optimx` as follows. This takes some time, as the derivative-free methods appear to have more difficulty with this formulation. Moreover, `Rcgmin` and `Rvmmin` are not recommended when an analytic gradient is not provided.

```
# turn off kkt to save time
tmeth<-c("ncg", "nvm", "lbfgs", "ucminf", "bobyqa", "tnewt", "slsqp")
mfv<-10*(n-1)^2 # set fn eval count big enough to avoid commonArgs error (bobyqa)
allansf<- opm(seq(n-1)/n, obj, gr="grfwd", method=tmeth,
              control=list(kkt=FALSE, dowarn=FALSE,maxfeval=mfv))
summary(allansf, order = "list(round(value, 3), fevals)", par.select = FALSE)
```

```
##              value fevals gevals hevals  conv kkt1 kkt2 xtime
## bobyqa    460.5170   8818      0      0     0   NA   NA 1.958
## lbfgs     494.4718     35     35      0 -1001   NA   NA 0.115
## slsqp     520.9416   2255   2254      0     0   NA   NA 1.285
## nvm       584.0714     67      6      0     3   NA   NA 0.006
## ncg      1784.4344      3      1      0     0   NA   NA 0.001
## tnewt    1818.2848     38     37      0  9999   NA   NA 0.020
## ucminf  13140.9297      9      9      0     0   NA   NA 0.011
```

```
allansc<- opm(seq(n-1)/n, obj, gr="grcentral", method=tmeth,
              control=list(kkt=FALSE, dowarn=FALSE,maxfeval=mfv))
summary(allansc, order = "list(round(value, 3), fevals)", par.select = FALSE)
```

```
##           value fevals gevals hevals conv kkt1 kkt2 xtime
## lbfgs  460.5170     42     42      0    0   NA   NA 0.176
## slsqp  460.5170    108    107      0    0   NA   NA 0.207
## ucminf 460.5170    773    773      0    0   NA   NA 1.640
## nvm    460.5170   1464   1356      0    0   NA   NA 7.385
## bobyqa 460.5170   8818      0      0    0   NA   NA 2.009
## tnewt  529.5788   4027   4026      0    0   NA   NA 4.230
## ncg    906.3752      3      1      0    0   NA   NA 0.001
```

```
allansp<- opm(seq(n-1)/n, obj, gr="grpracma", method=tmeth,
              control=list(kkt=FALSE, dowarn=FALSE,maxfeval=mfv))
summary(allansp, order = "list(round(value, 3), fevals)", par.select = FALSE)
```

```
##           value fevals gevals hevals conv kkt1 kkt2 xtime
## lbfgs  460.5170     42     42      0    0   NA   NA 0.126
## slsqp  460.5170    108    107      0    0   NA   NA 0.217
## ucminf 460.5170    756    756      0    0   NA   NA 1.616
## nvm    460.5170   1663   1550      0    0   NA   NA 8.920
## bobyqa 460.5170   8818      0      0    0   NA   NA 2.012
## tnewt  548.8028   4079   4078      0    0   NA   NA 4.550
## ncg    906.3752      3      1      0    0   NA   NA 0.002
```

Note that the more accurate gradient approximations appear to give slightly better results.

Use the gradient equations

Another approach is to “solve” the gradient equations. We can do this with a sum of squares minimizer, though the `nls` function in R is specifically NOT useful as it cannot, by default, deal with small or zero residuals. However, `nlfb` from package `nlsr` is capable of dealing with such problems. Unfortunately, it will be slow as it has to generate the Jacobian by numerical approximation unless we can provide a function to prepare the Jacobian analytically. Moreover, the determination of the Jacobian is still subject to the unfortunate scaling issues we have been confronting throughout this article.

This approach is yet to be tried.

The Rayleigh Quotient

The maximal and minimal eigensolutions of a symmetric matrix A are extrema of the Rayleigh Quotient

$$R(x) = (x'Ax)/(x'x)$$

We can also deal with generalized eigenproblems of the form

$$Ax = eBx$$

where B is symmetric and positive definite by using the Rayleigh Quotient

$$R_g(x) = (x'Ax)/(x'Bx)$$

Once again, the objective is scaled by the parameters, this time by their sum of squares. Alternatively, we may think of requiring the **normalized** eigensolution, which is given as

$$x_{normalized} = x/sqrt(x'x)$$

We will first try the projected gradient method `spg` from `BB`. Below is the code, where our test uses a matrix called the Moler matrix Nash (1979, Appendix 1). This matrix is simple to generate and is positive definite, but has one small eigenvalue that may not be computed to high relative precision. That is, we may get a number that is precise relative to the largest eigenvalue, but having few trustworthy digits.

Let us set up some infrastructure for the Rayleigh Quotient of the matrix.

```
molerbuild<-function(n){ # Create the moler matrix of order n
   # A[i,j] = i for i=j, min(i,j)-2 otherwise
   A <- matrix(0, nrow = n, ncol = n)
   j <- 1:n
   for (i in 1:n) {
      A[i, 1:i] <- pmin(i, 1:i) - 2
   }
   A <- A + t(A)
   diag(A) <- 1:n
   A
}

raynum<-function(x, A){
   raynum<-as.numeric((t(x)%*%A)%*%x)
}

RQ <- function(x, A){
  RQ <- raynum(x, A)/sum(x*x)
}

# proj<-function(x) { x/sqrt(sum(x*x)) } # Original
proj <- function(x) {sign(x[1]) * x/sqrt(c(crossprod(x))) } # from ravi

# proj1 <- function(x) {sign(x[1]) * x/max(x)} # no use!
```

Let us now try to use `BB::spg()` to find the largest eigenvalue by minimizing the Rayleigh Quotient of $-A$.

```
require(BB, quietly=TRUE)
n<-10
set.seed(4321)
x<-runif(n)
AA<-molerbuild(n)
cat("Eigenvalues from eigen:")
```

```
## Eigenvalues from eigen:
```

```
meig<-eigen(AA)
print(meig$values)
```

```
##  [1] 3.158981e+01 5.506463e+00 3.375284e+00 2.770820e+00 2.517158e+00
##  [6] 2.388872e+00 2.317702e+00 2.277391e+00 2.256491e+00 8.582807e-06
```

```
cat("Vector for smallest ev:")
```

```
## Vector for smallest ev:
```

```
vecmin<-meig$vectors[,n]
print(vecmin)
```

```
##  [1] 0.866015080 0.433009398 0.216509345 0.108264428 0.054151958 0.027115582
##  [7] 0.013637056 0.006977088 0.003805678 0.002537115
```

```
cat("RQ for minimal eigenvector=",raynum(vecmin,AA))
```

```
## RQ for minimal eigenvector= 8.582807e-06
```

```
x<-proj(x)  # Need to have parameters feasible
tmax<-system.time(asprqmax<-spg(x, fn=raynum, project=proj, A=-AA,
    control=list(trace=TRUE, triter=1,maxit=1000)))[[3]]
```

```
## iter:  0  f-value:  -20.09423  pgrad:  0.8771025
## iter:  1  f-value:  -31.46235  pgrad:  0.03087639
## iter:  2  f-value:  -31.55143  pgrad:  0.01613852
## iter:  3  f-value:  -31.58762  pgrad:  0.00362995
## iter:  4  f-value:  -31.58967  pgrad:  0.0009427463
## iter:  5  f-value:  -31.5898  pgrad:  0.0002548985
## iter:  6  f-value:  -31.58981  pgrad:  7.104836e-05
## iter:  7  f-value:  -31.58981  pgrad:  2.018526e-05
## iter:  8  f-value:  -31.58981  pgrad:  5.803606e-06
```

```
asprqmax
```

```
## $par
##  [1]  0.085465700 -0.002889881 -0.091145405 -0.176319834 -0.255536076
##  [6] -0.326113850 -0.385669103 -0.432189677 -0.464103537 -0.480334777
##
## $value
## [1] -31.58981
##
## $gradient
## [1] 5.803606e-06
##
## $fn.reduction
## [1] 11.49558
##
## $iter
## [1] 8
##
## $feval
## [1] 9
##
## $convergence
## [1] 0
##
## $message
## [1] "Successful convergence"
```

```
cat("Maximal eigenvalue is calculated as ", -asprqmax$value,"\n")
```

```
## Maximal eigenvalue is calculated as  31.58981
```

```
cat("Compare eigen:",meig$values[1],"  difference=",-asprqmax$value-meig$values[1], "\n")
```

```
## Compare eigen: 31.58981   difference= -5.409149e-09
```

```
xy <- rep(1/sqrt(n), n)
tmax2<-system.time(asprqmax2<-spg(xy, fn=raynum, project=proj, A=-AA,
    control=list(trace=TRUE, triter=1,maxit=1000)))[[3]]
```

```
## iter:  0  f-value:  -20.5  pgrad:  0.7776636
## iter:  1  f-value:  -31.51629  pgrad:  0.02149796
## iter:  2  f-value:  -31.56598  pgrad:  0.01227049
## iter:  3  f-value:  -31.58786  pgrad:  0.003475818
## iter:  4  f-value:  -31.58965  pgrad:  0.0009871069
## iter:  5  f-value:  -31.5898  pgrad:  0.0002837933
## iter:  6  f-value:  -31.58981  pgrad:  8.226663e-05
## iter:  7  f-value:  -31.58981  pgrad:  2.397799e-05
## iter:  8  f-value:  -31.58981  pgrad:  7.015653e-06
```

```
cat("maximal eigensolution: Value=",-asprqmax2$value,"in time ",tmax2,"\n")
```

```
## maximal eigensolution: Value= 31.58981 in time  0.001
```

```
print(asprqmax2$par)
```

```
##  [1]  0.085466806 -0.002889806 -0.091147788 -0.176323486 -0.255538009
##  [6] -0.326114501 -0.385668490 -0.432188271 -0.464102668 -0.480333914
```

```
tmin<-system.time(asprqmin<-spg(x, fn=raynum, project=proj, A=AA,
    control=list(trace=TRUE, triter=1,maxit=1000)))[[3]]
```

```
## iter:  0  f-value:  20.09423  pgrad:  0.8738632
## iter:  1  f-value:  0.1610048  pgrad:  0.6373408
## iter:  2  f-value:  1.253068  pgrad:  1.072525
## iter:  3  f-value:  1.44168  pgrad:  0.7840171
## iter:  4  f-value:  1.016774  pgrad:  0.6186038
## iter:  5  f-value:  3.940719  pgrad:  0.7924
## iter:  6  f-value:  0.03187537  pgrad:  0.4797277
## iter:  7  f-value:  0.02150955  pgrad:  0.3810032
## iter:  8  f-value:  0.01218731  pgrad:  0.3041359
## iter:  9  f-value:  0.001266454  pgrad:  0.07078474
## iter:  10  f-value:  0.000528509  pgrad:  0.08489315
## iter:  11  f-value:  0.00165654  pgrad:  0.1904217
## iter:  12  f-value:  0.0001417218  pgrad:  0.02495577
## iter:  13  f-value:  0.0001184352  pgrad:  0.0224623
## iter:  14  f-value:  8.854465e-05  pgrad:  0.01922957
## iter:  15  f-value:  9.035708e-06  pgrad:  0.001780281
## iter:  16  f-value:  9.541349e-06  pgrad:  0.005057676
## iter:  17  f-value:  1.10221e-05  pgrad:  0.008389123
## iter:  18  f-value:  8.598903e-06  pgrad:  0.0003246327
## iter:  19  f-value:  8.595024e-06  pgrad:  0.0002693203
## iter:  20  f-value:  8.584687e-06  pgrad:  8.147665e-05
## iter:  21  f-value:  8.582937e-06  pgrad:  2.117746e-05
## iter:  22  f-value:  8.582811e-06  pgrad:  6.798477e-06
```

```
cat("minimal eigensolution: Value=",asprqmin$value,"in time ",tmin,"\n")
```

```
## minimal eigensolution: Value= 8.582811e-06 in time  0.004
```

```
# print(asprqmin$par)
# Compare
cat("Diff from value from eigen():", (asprqmin$value-meig$values[n]))
```

```
## Diff from value from eigen(): 4.538548e-12
```

```
# cat("Vector difference:"); asprqmin$par - meig$vectors[,n]
# cat("Max abs relative difference=",max(abs((asprqmin$value-meig$values[n])/(abs(meig$values)+1e-18)))
```

If we ignore the constraint, and simply perform the optimization, we can sometimes get satisfactory solutions, though comparisons require that we normalize the parameters post-optimization. We can check if the scale of the eigenvectors is becoming large by computing the norm of the final parameter vector. In tests on the Moler matrix up to dimension 100, none grew to a worrying size.

For comparison, we also ran a specialized Geradin routine as implemented in R by one of us (JN). This gave equivalent answers, albeit more efficiently. For those interested, the Geradin routine is available as referenced in Nash (2012).

The extended Rosenbrock function on the unit ball

The adagio package (Borchers (2022)) gives an extended version of the Rosenbrock banana-shaped valley problem. This becomes a sumscale problem if we constrain the parameters to be on the unit ball, that is, where the sum of squares of the parameters is 1.

```
library(alabama)
library(optimx)
library(nloptr)
```

```
##
## Attaching package: 'nloptr'

## The following object is masked from 'package:alabama':
##
##     auglag
```

```
library(BB)
####################################
# Minimizing a function on the unit ball:  Min f(x), s.t. ||x|| = 1
#
rosbkext.f <- function(x){
    p <- x
    n <- length(p)
    sum (100*(p[1:(n-1)]^2 - p[2:n])^2 + (p[1:(n-1)] - 1)^2)
}

heq <- function(x){
    1 - sum(x*x)
}

transform <- function(x){
# transforms x into z such that ||z|| = 1
    p <- length(x)
    z <- rep(NA, p)
    z[1] <- cos(x[1])
    z[p] <- prod(sin(x[-p]))
    if (p > 2) z[2:(p-1)] <- cumprod(sin(x[1:(p-2)]))*cos(x[2:(p-1)])
    return(z)
}

rosbkext.t <- function(x){
    n <- length(x)
    p <- transform(x)
    sum (100*(p[1:(n-1)]^2 - p[2:n])^2 + (p[1:(n-1)] - 1)^2)
}

ProjSphere <- function(x){
    x/sqrt(sum(x*x))
}

ProjSpheresgn <- function(x){
    sign(x[1])*x/sqrt(sum(x*x))
}

n <- 10
set.seed(1234)
p0 <- runif(n, 0, 3)

# Unconstrained optimization with parameter transformation to satisfy unit-length constraint
#
ans <- optim(par=p0, fn=rosbkext.t, method="BFGS", control=list(maxit=1000))
proptimr(ans)
```

```
## Result  ans (  ->  ) calc. min. = 9.418838  at
## -29.00744     156.519     109.596     -59.93688     -22.61133     4.065798     0.7760579     -0.0073
## After  74  fn evals, and  45  gr evals and  NA  hessian evals
## Termination code is  0 :
##
## -------------------------------------------------
```

```
system.time(ans2 <- alabama::auglag(p0, rosbkext.f, heq=heq, control.outer = list(trace=FALSE, kktchk=F
```

```
##    user  system elapsed
##   0.028   0.000   0.027
```

```
proptimr(ans2)
```

```
## Result  ans2 (  ->  ) calc. min. = 6.426809  at
## 0.7478999     0.5648426     0.327218     0.1165006     0.02342218     0.01045982     0.01001565     
## After  694  fn evals, and  117  gr evals and  NA  hessian evals
## Termination code is  0 :
## Gradient: [1] 4.816521e-02 4.430184e-02 1.899406e-02 8.673540e-03 2.029531e-03
##  [6] 2.142209e-03 7.416204e-04 8.101936e-04 8.108949e-04 1.135815e-05
##
## -------------------------------------------------
```

```
system.time(ans3 <- nloptr::slsqp(p0, rosbkext.f, heq=heq))
```

```
##    user  system elapsed
##   0.026   0.017   0.015
```

```
proptimr(ans3)
```

```
## Result  ans3 (  ->  ) calc. min. = 6.426807  at
## 0.7478806     0.5648321     0.3272499     0.116557     0.02348033     0.01050433     0.01006523     
## After   fn evals, and   gr evals and  NA  hessian evals
## Termination code is  4 : NLOPT_XTOL_REACHED: Optimization stopped because xtol_rel or xtol_abs (abov
##
## -------------------------------------------------
```

```
system.time(ans4 <- spg(p0, rosbkext.f, project=ProjSphere))
```

```
## iter:  0  f-value:  5363.139  pgrad:  0.3108113
## iter:  10  f-value:  6.761595  pgrad:  1.170664
## iter:  20  f-value:  6.426807  pgrad:  0.0008499551
```

```
##    user  system elapsed
##   0.004   0.008   0.005
```

```
proptimr(ans4)
```

```
## Result  ans4 (  ->  ) calc. min. = 6.426807  at
## 0.7478804     0.564832     0.3272502     0.1165575     0.02348068     0.01050432     0.01006517     
## After   fn evals, and   gr evals and  NA  hessian evals
## Termination code is  0 : Successful convergence
## Gradient:[1] 2.012061e-05
##
## --------------------------------------------------
```

```
system.time(ans4a <- spg(p0, rosbkext.f, project=ProjSpheresgn))
```

```
## iter:  0  f-value:  5363.139  pgrad:  0.3108113
## iter:  10  f-value:  6.761595  pgrad:  1.170664
## iter:  20  f-value:  6.426807  pgrad:  0.0008499551
```

```
##    user  system elapsed
##   0.004   0.000   0.005
```

```
proptimr(ans4a)
```

```
## Result  ans4a (  ->  ) calc. min. = 6.426807  at
## 0.7478804     0.564832     0.3272502     0.1165575     0.02348068     0.01050432     0.01006517     
## After   fn evals, and   gr evals and  NA  hessian evals
## Termination code is  0 : Successful convergence
## Gradient:[1] 2.012061e-05
##
## --------------------------------------------------
```

```
c(ans$value, ans2$value, ans3$value, ans4$value, ans4a$value)
```

```
## [1] 9.418838 6.426809 6.426807 6.426807 6.426807
```

```
sevmeth <- c("ncg", "nvm", "BFGS", "L-BFGS-B", "tnewt", "ucminf", "spg")
several <- opm(p0, fn=rosbkext.t, gr="grcentral", method=sevmeth, control=list(trace=0))
```

```
## Warning in BB::spg(par = spar, fn = efn, gr = egr, lower = slower, upper =
## supper, : Unsuccessful convergence.
```

```
sumrbk<-summary(several, order=value, par.select=1:5)
print(sumrbk)
```

```
##                   p1 s1          p2 s2         p3 s3         p4 s4         p5 s5
## nvm      -7.0091180    -53.9602431     3.4947023     0.2599813     5.5715970
## ucminf    0.7259327     -0.5531680     3.4947023     0.2599813     0.7115884
## ncg       5.5572526    -22.5443166     3.4947023     0.2599812     0.7115869
## BFGS     -7.0090845    -53.9600862     3.4936038     0.2464820     5.6408459
## L-BFGS-B  0.7258983     -0.5530069     2.7895973     2.8951592     3.7844436
## spg       0.7258107      0.5526231    -0.3496154     2.9288467     3.5030706
## tnewt    -8.6916438    -52.8463130     0.3607824     0.2591798     0.6887679
##              value fevals gevals hevals conv  kkt1  kkt2 xtime
## nvm      6.426807    115     81      0    0  TRUE FALSE 0.024
## ucminf   6.426807     62     62      0    0  TRUE FALSE 0.009
## ncg      6.426807    921    354      0    0  TRUE FALSE 0.063
## BFGS     6.436763     78     41      0    0  TRUE FALSE 0.007
## L-BFGS-B 6.436766    352    352      0    0  TRUE FALSE 0.073
## spg      6.456637   1900   1503      0    1 FALSE FALSE 0.349
## tnewt    9.408793    292    291      0    0  TRUE FALSE 0.054
```

```
stp <- function(xx, fn){
  # standardize parameters in an opm output data-frame and check fn
  nr <- dim(xx)[1]
  npar <- which(colnames(xx)=="value") - 1
  newxx <- xx[, 1:(npar+1)]
  rownames(newxx)<-rownames(xx)
  colnames(newxx)<-colnames(xx)[1:(npar+1)]
  for (ii in 1:nr){
    meth <- rownames(xx)[ii]
    upar <- coef(xx[ii, ])
    tpar <- transform(upar) # spherical transform
    fval <- fn(upar)
    newxx[ii, 1:npar] <- tpar
    newxx[ii, npar+1] <- fval
    cat(meth," fval=",fval," at "); print(tpar[1:5])
  }
  newxx
}
tt<-stp(several, rosbkext.t)
```

```
## ncg  fval= 6.426807  at [1] 0.74788061 0.56483208 0.32724994 0.11655698 0.02348039
## nvm  fval= 6.426807  at [1] 0.74788061 0.56483208 0.32724994 0.11655697 0.02348037
## BFGS  fval= 6.436763  at [1] 0.74790285 0.56486547 0.32728669 0.11658214 0.02348588
## L-BFGS-B  fval= 6.436766  at [1] 0.74790344 0.56486635 0.32728606 0.11657767 0.02347118
## tnewt  fval= 9.408793  at [1] -0.74308067  0.56671363  0.33299006  0.12143978  0.02485856
## ucminf  fval= 6.426807  at [1] 0.74788061 0.56483208 0.32724994 0.11655697 0.02348037
## spg  fval= 6.456637  at [1] 0.74796158 0.56494437 0.32733520 0.11665340 0.02357039
```

```
print(tt)
```

```
##                  p1        p2        p3        p4         p5            p6
## ncg       0.7478806 0.5648321 0.3272499 0.1165570 0.02348039  1.050437e-02
## nvm       0.7478806 0.5648321 0.3272499 0.1165570 0.02348037  1.050432e-02
## BFGS      0.7479028 0.5648655 0.3272867 0.1165821 0.02348588  1.050428e-02
## L-BFGS-B  0.7479034 0.5648663 0.3272861 0.1165777 0.02347118  1.053857e-02
## tnewt    -0.7430807 0.5667136 0.3329901 0.1214398 0.02485856  1.066550e-02
## ucminf    0.7478806 0.5648321 0.3272499 0.1165570 0.02348037  1.050432e-02
## spg       0.7479616 0.5649444 0.3273352 0.1166534 0.02357039 -4.991341e-08
##                   p7          p8            p9           p10    value
## ncg      0.010065153 0.010052391  9.860523e-03  9.605723e-05 6.426807
## nvm      0.010065226 0.010052399  9.860579e-03  9.585220e-05 6.426807
## BFGS     0.010041703 0.009878878  1.889037e-07 -3.083038e-05 6.436763
## L-BFGS-B 0.010155391 0.009738523  6.684097e-07 -9.084156e-05 6.436766
## tnewt    0.010158957 0.010144358  9.947400e-03  9.842338e-05 9.408793
## ucminf   0.010065225 0.010052400  9.860580e-03  9.585004e-05 6.426807
## spg      0.008808479 0.001056753 -3.395910e-04  7.733199e-04 6.456637
```

```
####################################
```

Here we note that the `spg()` method that works well with a suitable projection, does much less well on the unconstrained minimization of the transformed objective `rosbkext.t()`.

The R-help example

As a final example, let us use our present techniques to solve the problem posed by Lanh Tran on R-help. We will use only a method that scales the parameters directly inside the objective function and not bother with gradients for this small problem.

```
ssums<-function(x){
  n<-length(x)
  tt<-sum(x)
  ss<-1:n
  xx<-(x/tt)*(x/tt)
  sum(ss*xx)
}

cat("Try penalized sum\n")
```

```
## Try penalized sum
```

```
require(optimx)
st<-runif(3)
aos<-opm(st, ssums, gr="grcentral", method="MOST")
```

```
## Warning in opm(st, ssums, gr = "grcentral", method = "MOST"): 'snewtonm'
## removed from 'method' -- no hess()
```

```
# rescale the parameters
nsol<-dim(aos)[1]
for (i in 1:nsol){
  tpar<-aos[i,1:3]
  ntpar<-sum(tpar)
  tpar<-tpar/ntpar
#  cat("Method ",aos[i, "meth"]," gives fval =", ssums(tpar))
  aos[i, 1:3]<-tpar
}
summary(aos,order=value)[1:5,]
```

```
##                p1 s1        p2 s2        p3 s3     value fevals gevals hevals
## subplex 0.5454545    0.2727273    0.1818182    0.5454545    416      0      0
## nlminb  0.5454545    0.2727273    0.1818182    0.5454545      9      8      0
## ncg     0.5454545    0.2727273    0.1818182    0.5454545     15      7      0
## nvm     0.5454545    0.2727273    0.1818182    0.5454545     15      9      0
## Rvmmin  0.5454545    0.2727273    0.1818182    0.5454545     15      9      0
##         conv kkt1  kkt2 xtime
## subplex    0 TRUE FALSE 0.002
## nlminb     0 TRUE FALSE 0.000
## ncg        0 TRUE FALSE 0.000
## nvm        0 TRUE FALSE 0.001
## Rvmmin     0 TRUE FALSE 0.002
```

```
ssum<-function(x){
  n<-length(x)
  ss<-1:n
  xx<-x*x
  sum(ss*xx)
}
proj.simplex <- function(y) {
# project an n-dim vector y to the simplex Dn
# Dn = { x : x n-dim, 1 >= x >= 0, sum(x) = 1}
# Ravi Varadhan, Johns Hopkins University
# August 8, 2012
n <- length(y)
sy <- sort(y, decreasing=TRUE)
csy <- cumsum(sy)
rho <- max(which(sy > (csy - 1)/(1:n)))
theta <- (csy[rho] - 1) / rho
return(pmax(0, y - theta))
}
as<-spg(st, ssum, project=proj.simplex)
```

```
## iter:  0  f-value:  1.314879  pgrad:  0.3785277
## iter:  10  f-value:  0.5454545  pgrad:  9.544587e-06
```

```
cat("Using project.simplex with spg: fmin=",as$value," at \n")
```

```
## Using project.simplex with spg: fmin= 0.5454545  at
```

```
print(as$par)
```

```
## [1] 0.5454512 0.2727297 0.1818191
```

Apart from the parameter rescaling, this is an entirely "doable" problem. Note that we can also solve the problem as a Quadratic Program using the `quadprog` package.

```
library(quadprog)
Dmat<-diag(c(1,2,3))
Amat<-matrix(c(1, 1, 1), ncol=1)
bvec<-c(1)
meq=1
dvec<-c(0, 0, 0)
ans<-solve.QP(Dmat, dvec, Amat, bvec, meq=0, factorized=FALSE)
ans
```

```
## $solution
## [1] 0.5454545 0.2727273 0.1818182
##
## $value
## [1] 0.2727273
##
## $unconstrained.solution
## [1] 0 0 0
##
## $iterations
## [1] 2 0
##
## $Lagrangian
## [1] 0.5454545
##
## $iact
## [1] 1
```

Conclusion

Sumscale problems can present difficulties for optimization (or function minimization) codes. These difficulties are by no means insurmountable, but they do require some attention.

While specialized approaches are "best" for speed and correctness, a general user is more likely to benefit from a simpler approach of embedding the scaling in the objective function and rescaling the parameters before reporting them. We also note that the use of a projected gradient via `spg` from package `BB` works very well, but the projection needs to be set up carefully, as with the use of the sign of the first element in dealing with the Rayleigh Quotient.

References

Borchers, Hans W. 2022. *Adagio: Discrete and Global Optimization Routines*. https://CRAN.R-project.org/package=adagio.

Geradin, M. 1971. "The Computational Efficiency of a New Minimization Algorithm for Eigenvalue Analysis." *J. Sound Vib.* 19: 319–31.

Nash, John C. 1979. *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*. Bristol: Adam Hilger.

———. 2012. "Timing Rayleigh Quotient minimization in R."