Animating geometric optimization: small polygons

John C. Nash, Greg Snow

2016-12-12

Abstract

Roptanimation is an experimental R package to display the progress of geometric animations. A classic example is the **largest small polygon** where we aim to maximize the area of the polygon subject to the constraint that no two vertices are separated by more than one unit of distance. This article discusses the problem and how the animation is created, but only shows snapshots of the animation.

TODOS

- sort out what is going on with tkrplot – does not always work
- try nloptr
- try shiny for running the animation ??
- Do we want to make VMPOLY files available??

Background

The **The Largest Small Hexagon** is the title of a paper by Graham (1975). This did not introduce this problem, but served to bring it to wider attention. The problem statement asks for the vertices of a hexagon with maximal area such that no two vertices are more than 1 unit distant from each other. There is even a Wikipedia entry for this problem (https://en.wikipedia.org/wiki/Biggest_little_polygon). The approximate area of the optimal hexagon 0.674981, while it is fairly easy to show that a regular hexagon of diameter 1 has area 6 times the area of an equilateral triangle of side 0.5, i.e., 6 * 0.5 * 0.5 * sin(pi/3) / 2 = 0.6495191 (approximately).

The interest in this article is that problems like this have a natural visual quality that can be used to interest a non-technical audience, especially if the progress of an optimizater can be animated. Note that there are many mathematical aspects of such problems that we ignore in our treatment. See Audet, Hansen, and Messine (2007).

To provide a visual presentation of the optimization, Nash coded a display for the IBM PC family of MS DOS computers running GWBASIC. This code was then used to to illustrate constrained optimization using the tools in J. C. Nash and Walker-Smith (1987) (now available online at https://archive.org/details/ost-engineering-jnmws2004). The actual files for the polygon problem are part of this **Roptanimation** package. ?? In May 2016, the discovery that these files could still execute, albeit clumsily, under Linux using DosBox or PCBasic raised the possibility of bringing them up to date. R was a logical choice for such an implementation, given that the authors and many others work with this software system.

There are other animations of optimization tools. Using Javascript, Ben Frederickson developed a very attractive demonstration of a selection of optimization algorithms. See http://www.benfrederickson.com/numerical-optimization/. Duncan Murdoch prepared a very nice illustration of the behaviour of a Nelder-Mead polytope minimization, the code for which can be found at https://github.com/florianhartig/LearningBayes/blob/master/CommentedCode/02-Samplers/Optimization/Duncan-Murdoch-Nelder-Mead-demonstration.R.

Parametrization of the polygon

For a polygon with `nv` vertices, we have `2*nv` cartesian (i.e., x, y) coordinates. However, use of cartesian coordinates as parameters for this problem leads to a very complicated specification, since 2 parameters can be set at a fixed value right away as defining an origin of the polygon. That is, we can arbitrarily fix vertex 1 at the `(0, 0)` or origin of our [2D] space. Moreover, we can put the second vertex at `(b[1], 0)` where `b` is a vector of `(2*nv - 3)` parameters. Changing to a representation that uses a radius from the origin for vertex `L` equal to `b[L-1]`, we could use the angle of this vertex from the positive x axis as a parameter. Call this angle `alpha[L]`. Clearly `alpha[1]` for vertex 2 is 0, so the 2nd vertex is still at `(b[1], 0)`. Note that there is no `b[0]`, which would be the distance of vertex 1 from itself.

We could put the alpha angles in the parameter vector as `b[L+lshift]` where `lshift = nv - 3`. Thus the first non-zero angle is for vertex 3 and is parameter `3 + nv - 3 = nv`. Check: there are `(nv - 1)` radius parameters, so the first angle parameter is in position `nv`.

There may be good implementations based on having parameters `b[nv] ... b[2*nv-3]` equal to the angles for vertices `3...nv` from the x axis, e.g., as in Dolan, Moré, and Munson (2004) and Audet, Hansen, and Messine (2007). However, that then requires the angles to be monotonically increasing. By specifying instead that `b[L+lshift] = alpha[L-1] - alpha[L-2]` for `L=3 ... nv`, and noting that `alpha[1]=0` we automatically get the angles alpha monotonic by imposing a lower bound of 0 on the parameters `b`.

Note that the radii already cannot be negative (in fact, zero is a bad idea too), so a lower bound of 0 can be applied to all the parameters `b`. An upper bound of 1 clearly applies to the first `(nv - 1)` parameters. The other `(nv - 2)` parameters are angles in radians. If we are to have the polygon in the positive y half-space in cartesian coordinates, then pi is an obvious (and likely conservative) bound on these angles. In fact, `pi` is a bound on their sum.

We make no assertion that this is the only or best parametrization of this problem, and welcome suggestions for other ways to prepare the optimization.

Problem setup

The above parametrization is implemented in the function `polysetup(nv, defsize)`, where `defsize` is the default "size" of a regular polygon for which initial parameters are established. Generally we will begin our optimization with a polygon for which the size is smaller than 1, and also commence with a regular polygon for convenience. This ensures that our initial polygon is feasible, and some optimization methods such as `nmkb` require that. Note that for drawing the polygon, it is useful to think of a vertex `L+1` which is at the same position as vertex 1.

```r
polysetup <- function(nv, defsize=0.98){
# Function to set up animation of the "largest small polygon"
#   problem. This attempts to find the polygon in nv vertices
#   that has the largest area inside the polygon subject to
#   the constraint that no two vertices are more than 1 unit
#   distant from each other.
# Ref. Graham, "The largest small hexagon" ....???
    cat("polysetup with ",nv," vertices\n")
#    nv <- readline("number of vertices = ")
    nvmax <- 100 # Arbitrary limit -- change ??
    cat("nv, nvmax:",nv, nvmax, "\n")
    if (nv > nvmax) { stop("Too many vertices for polygon") }
    mcon <- (nv-2)*(nv-1)/2 # Number of distance constraints
    n <- 2*nv - 3 # Number of parameters in the problem
    # Thus we use a vector b[] of length n
    # Note that we use RADIAL coordinates to simplify the
    # optimization, but convert to cartesian to plot them
    # First point is always at the origin (0,0) cartesian
    # Second point is at (b[1],0) in both cartesian or polar
    # where cartesian is (x, y) and radial is (radius, angle)
    # Choice: angle in radians. ??
    # There are 2*nv cartesian coordinate values
    # i.e., (x, y) for nv point
    # But first point is (0,0) and second has angle 0
    #   since point 2 fixed onto x axis (angular coordinate 0).
    # So b[1] ... b[nv-1] give radial coordinates of points 2:nv
    # and b[nv] ... b[2*nv-3] give angle coordinates of points 3:nv
    # ?? not needed LET L8=nv-3: REM so l+l8 indexes angles as l=3..nv
    # Distances between points can be worked out by cosine rule for
    # triangles i.e. D = sqrt(ra^2 + rb^2 - 2 ra rb cos(angle)
    # Now set lower and upper bounds
    lb <- rep(0, n) # all angles and distances non-negative
    ub <- c(rep(1, (nv-1)), rep(pi, (nv-2))) # distances <=1, angles <= pi
    # if we have angles > pi, then we are reflecting the polygon about an edge
    # set inital parameters to a regular polygon of size .98
   # defsize <- 1
    regangle <- pi/nv #  pi/no. of vertices
# test to define polygon
    q5<-defsize*sin(regangle) # REM regangle/nv = alpha
    b<-rep(NA,n)
#    x <- rep(NA, nv)
#    y <- rep(NA, nv)
#    x[1] <- 0
#    y[1] <- 0
#    x[2] <- q5
#    y[2] <- 0
    b[1]<-q5
    q1 <- q5
    q2 <- 0 # x2 and y2
    l8 <- nv - 3 # offset for indexing
    for (ll in 3:nv){
        b[ll+l8] <- regangle
        q1 <- q1+q5*cos(2*(ll-2)*regangle)
        q2 <- q2+q5*sin(2*(ll-2)*regangle)
#        x[ll]<-q1
#        y[ll]<-q2
        b[ll-1]<-sqrt(q1*q1+q2*q2)
    }
#    par0 <- b # return the parameters as par0
    res <- list(par0 = b, lb = lb, ub =ub)
}
```

The parameters of a regular hexagon of size 1 can be created as follows.

```
# A regular hexagon of size 1
reghex1 <- polysetup(6, defsize=1)
```

```
## polysetup with  6  vertices
## nv, nvmax: 6 100
```

```
cat("Parameters of the regular hexagon of unit size\n")
```

```
## Parameters of the regular hexagon of unit size
```

```
print(reghex1$par0)
```

```
## [1] 0.5000000 0.8660254 1.0000000 0.8660254 0.5000000 0.5235988 0.5235988
## [8] 0.5235988 0.5235988
```

The polygon area

The parameterization of the problem allows the area to be computed as the sum of the areas of the triangles made up from vertices `1`, `L` and `(L+1)` where `L` runs from `2` to `(nv-1)`. That is, there are `nv - 2` triangles. For the hexagon, the 4 triangles are made up of the vertices (1 2 3), (1 3 4), (1 4 5), and (1 5 6).

```
polyarea<-function(b) {
   # compute area of a polygon defined by radial coordinates
   # This IGNORES constraints
   nv <- (length(b)+3)/2
   area <- 0
   l8 <- nv-3
   for (l in 3:nv){ # nv - 2 triangles
      q1 <- b[[l-2]] # side 1
      q2 <- b[[l-1]] # side 2
      q3 <- b[[l+l8]] # angle
      atemp <- q1*q2*sin(q3)
      area <- area + atemp
   }
   area <- area * 0.5
   area
}
```

For reference, let us compute the area of this hexagon.

```
reg1area <- polyarea(reghex1$par)
cat("Reference area of regular hexagon of unit size=",reg1area,"\n")
```

```
## Reference area of regular hexagon of unit size= 0.6495191
```

This is in accord with Graham (1975) and our result above.

Conversion of radial to cartesian coordinates

For drawing the current polygon, we need cartesian coordinates rather than the specially organized radial coordinates defined by the optimization parameters. The R function `polypar2XY` carries out this computation and puts the `x, y` coordinates in a two-vector list `XY`. `XY\$x` gives the `x` coordinates and `XY\$y` gives the `y` coordinates. To simplify the plotting of the polygon the first and last values of each list are both 0 so that a graph that uses lines to join the vertices automatically gives the closed figure polygon.

```
polypar2XY <- function(b) {
# converts radial coordinates for polygon into Cartesian coordinates
#  that are more suitable for plotting
    nv <- (length(b)+3)/2
    l8 <- nv - 3 # offset for indexing
    x <- rep(NA, nv+1)
    y <- rep(NA, nv+1)
    # One extra point to draw polygon (return to origin)
    x[1] <- 0
    y[1] <- 0
    x[2] <- b[1]
    y[2] <- 0
    cumangle <- 0 # Cumulative angle of points so far
    q5 <- b[1]
    q1 <- q5 # x2
    q2 <- 0 #  y2
    for (ll in 3:nv){
        cumangle <- cumangle + b[ll+18]
        cradius <- b[ll-1]
        q1 <- cradius*cos(cumangle)
        q2 <- cradius*sin(cumangle)
        x[ll]<-q1
        y[ll]<-q2
    }
    x[nv+1] <- 0 # to close the polygon
    y[nv+1] <- 0
    XY <- list(x=x, y=y)
    XY
}
```

Distance between polygon vertices

To verify constraints and to construct penalty or barrier functions for the optimization process for this problem, we also need vertex to vertex distances. These are computed by the function `polydistXY`. This function uses the cartesian coordinates for the current polygon that result from running the function `polypar2XY`

```
polydistXY <- function(XY) {
#   compute point to point distances from XY data
   nv <- length(XY$x)-1
   ncon <- (nv - 1)*(nv)/2
   dist2 <- rep(NA, ncon) # squared distances
   ll <- 0 # index of constraint
   for (i in 1:(nv-1)){
      for (j in ((i+1):nv)){
         xi <- XY$x[i]
         xj <- XY$x[j]
         yi <- XY$y[i]
         yj <- XY$y[j]
         dd <- (xi-xj)^2 + (yi-yj)^2
         ll <- ll + 1
         dist2[ll] <- dd
      }
   }
   dist2
}
```

Computing vertex distances from radial parameters

We can compute these vertex distances from the radial parameters of the polygon by computing the XY coordinates and then the distances. The following function calls the conversion from radial to cartesian coordinates, then computes the distances.

```
polypar2distXY <- function(pars) {
# compute the pairwise distances using two calls
   nv <- (length(pars) + 3)/2
   XY <- polypar2XY(pars)
   dist2 <- polydistXY(XY)
}
```

Alternatively, and perhaps more efficiently or at least more elegantly, we can do a one-step calculation. However, the following function ONLY computes the non-radial inter-vertex differences. The first `nv - 1` parameters where `nv` is the number of vertices give the other distances. Moreover, the positions of these distances in the output of polypar2distXY are not obvious at first glance.

```
polypardist2 <- function(b) {
# compute the pairwise distances for non-radii lines
   nv <- (length(b) + 3)/2
   l8 <- nv - 3 # end of radii params
   ll <- 0 # count the distances (non-radii ones)
   sqdist <- rep(NA, (nv-1)*(nv-2)/2)
   for (ii in 2:(nv-1)){
      for (jj in (ii+1):nv) {
          ra <- b[ii-1]
          rb <- b[jj-1]
          angleab <- 0
          for (kk in (ii+1):jj) { angleab <- angleab + b[kk+l8] }
          d2 <- ra*ra+rb*rb -2*ra*rb*cos(angleab) # Cosine rule for squared dist
          ll <- ll+1
          sqdist[[ll]] <- d2
      }
   }
   sqdist
}
```

Testing functions.

Note that we tested our functions to create the original polygon and compute its area. This is a step that we recommend. In fact, one of us (JN) refuses to look at user queries about his optimization routines unless there is evidence that objective functions and gradients have been checked. It is an important part of the solution of EVERY optimization problem that users verify that they are solving the intended problem. Moreover, even in our own work, the simple checks often reveal silly but critical errors.

An example of a test script follows.

```
## @knitr polyex0

# Example code
nv <- 6
cat("Polygon data:\n")
```

```
## Polygon data:
```

```
myhex <- polysetup(nv)
```

```
## polysetup with  6  vertices
## nv, nvmax: 6 100
```

```
print(myhex)
```

```
## $par0
## [1] 0.4900000 0.8487049 0.9800000 0.8487049 0.4900000 0.5235988 0.5235988
## [8] 0.5235988 0.5235988
##
## $lb
## [1] 0 0 0 0 0 0 0 0 0
##
## $ub
## [1] 1.000000 1.000000 1.000000 1.000000 1.000000 3.141593 3.141593 3.141593
## [9] 3.141593
```

```
x0 <- myhex$par0 # initial parameters
cat("Area:\n")
```

```
## Area:
```

```
myhexa <- polyarea(x0)
print(myhexa)
```

```
## [1] 0.6237981
```

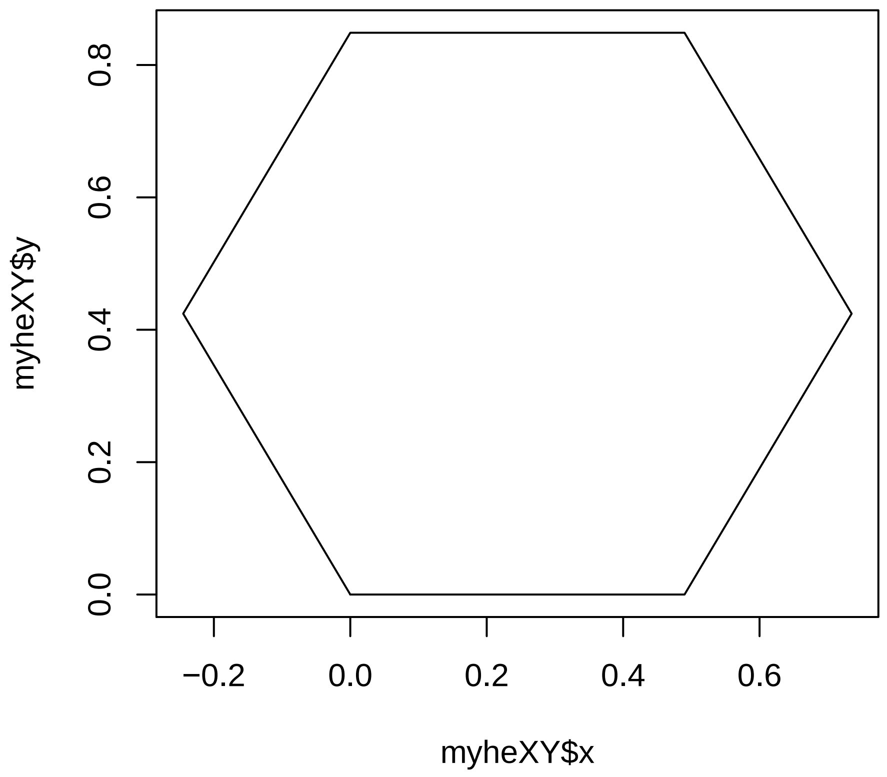
```
cat("XY coordinates\n")
```

```
## XY coordinates
```

```
myheXY <- polypar2XY(x0)
print(myheXY)
```

```
## $x
## [1]  0.000000e+00  4.900000e-01  7.350000e-01  4.900000e-01  5.196819e-17
## [6] -2.450000e-01  0.000000e+00
##
## $y
## [1] 0.0000000 0.0000000 0.4243524 0.8487049 0.8487049 0.4243524 0.0000000
```

```
plot(myheXY$x, myheXY$y, type="l")
```



```
cat("Constraints:\n")
```

```
## Constraints:
```

```
myhexc<-polydistXY(myheXY)
print(myhexc)
```

```
##  [1] 0.2401 0.7203 0.9604 0.7203 0.2401 0.2401 0.7203 0.9604 0.7203 0.2401
## [11] 0.7203 0.9604 0.2401 0.7203 0.2401
```

```
cat("Vertex distances:")
```

```
## Vertex distances:
```

```
print(sqrt(myhexc))
```

```
##  [1] 0.4900000 0.8487049 0.9800000 0.8487049 0.4900000 0.4900000 0.8487049
##  [8] 0.9800000 0.8487049 0.4900000 0.8487049 0.9800000 0.4900000 0.8487049
## [15] 0.4900000
```

```
cat("check distances with polypar2distXY\n")
```

```
## check distances with polypar2distXY
```

```
try1 <- polypar2distXY(x0)
print(try1)
```

```
##  [1] 0.2401 0.7203 0.9604 0.7203 0.2401 0.2401 0.7203 0.9604 0.7203 0.2401
## [11] 0.7203 0.9604 0.2401 0.7203 0.2401
```

```
cat("check distances with polypardist2 augmenting output with parameter squares\n")
```

```
## check distances with polypardist2 augmenting output with parameter squares
```

```r
try2 <- polypardist2(x0)
try2 <- c(x0[1:(nv-1)]^2, try2)
print(try2)
```

```
##  [1] 0.2401 0.7203 0.9604 0.7203 0.2401 0.2401 0.7203 0.9604 0.7203 0.2401
## [11] 0.7203 0.9604 0.2401 0.7203 0.2401
```

```r
cat("Max abs difference = ",max(abs(try1-try2)),"\n")
```

```
## Max abs difference =  2.220446e-16
```

Setup of the optimization

The constrained optimization to maximize the area actually minimizes the negative area. This is because most optimization solvers minimize, and we recommend keeping the direction of progress consistent to avoid errors.

We do, however, need to account for the constraints. Clearly since the radial parameters start at one vertex of the polygon, they are bounded above by 1. And naturally, we cannot have a polygon with negative lengths, so 0 is an obvious lower bound, though realistically, some modest positive value would likely be workable. This accounts for constraints on distance from the first, or base, vertex. For the other distances, we will apply a penalty function which will be added to the negative area. We can also put 0 as a lower bound on the angular parameters, and a reasonable upper bound as well. pi serves as a conservative bound for these parameters.

The solvers in the package **optimrx** represent the majority of the unconstrained and bounds constrained function minimizers commonly available in **R**. Typically, we create a penalty or barrier function that is added to our objective (the negative area) to impose the constraint. Penalty functions typically increase the objective more as we increasingly violate the constraints. Barrier functions start to add to the objective before the constraint boundary, increasing rapidly as we get very close to the boundary. We will use a number of these techniques. Our constrained objective functions are

- polyobjbig.R: the objective is assigned a very large value whenever a constraint is violated.
- polyobjq.R: a multiple (`penfactor`) of the squared constraint violation is added to the negative area.
- polyobj.R: a multiple (`penfactor`) of the negative sum of the logs of the **slacks** is added to the negative area. The slacks are the (positive) distances to the constraint boundaries. We must remain in the feasible region or this objective is undefined. We do not apply slacks to the radial coordinates in this objective function, for which we attempt solutions only with solvers that can handle bounds constraints.
- polyobju.R: this is essentially the same objective function as polyobj.R, but we now compute slacks for the radial parameters, so that unconstrained minimizers can be applied to this function.

Because the log() function increases extremely rapidly for small arguments, the `penfactor` for the barrier functions is generally quite small, while that for the quadratic penalty is quite large. We also allow for the slacks / violations to be modified by shifting the constraint boundary slightly using a quantity `epsilon`. This latter option has not been examined closely yet.

There are two specific adjustments to the codes above we can make:

- Except for polyobjbig.R, we can compute gradients and the code is given below.
- It is not uncommon for minimizers to make steps into an infeasible region. An attempt to avoid a halt in the minimization process due to an error, we have recoded polyobj.R to polyobjp.R which attempts to provide a large number for the objective in such cases. The gradient, however, may not be computable, so we try to provide a warning rather than an error.

Here are the codes.

Making objective very large on constraint violation

This is an old "trick" in optimization of using a non-gradient direct search method that assigns the objective function its correct value when the parameters are feasible and a very large value when they are violated. We supply the value of `bignum` to the objective function via the call.

```
polyobjbig <- function(x, bignum=1e10, epsilon=0) {
 # Put objective to bignum when constraints violated
 nv = (length(x)+3)/2 # number of vertices
 area <- polyarea(x)
 d2 <- c(x[1:(nv-1)]^2, polypardist2(x)) # distances
 slacks <- 1.0 + epsilon - d2 # slack vector
 if (any(d2 >=1) ) {
     f <- bignum
     attr(f,"area") <- -area
 } else {
    f <-  -area
    attr(f,"area") <- area
 } # negative area
 attr(f,"minslack") <- min(slacks)
 f
}
```

To test several optimizers at once, we use the `opm()` function of the R-forge package `optimrx`.

```
## @knitr polyexbig

library(optimrx)
cat("Attempt with setting objective big on violation\n")
```

```
## Attempt with setting objective big on violation
```

```
x0 <- myhex$par0 # starting parameters (slightly reduced regular hexagon)
cat("Starting parameters:")
```

```
## Starting parameters:
```

```
print(x0)
```

```
## [1] 0.4900000 0.8487049 0.9800000 0.8487049 0.4900000 0.5235988 0.5235988
## [8] 0.5235988 0.5235988
```

```
meths <- c("Nelder-Mead", "nmkb", "hjkb", "newuoa")
solb <- opm(x0, polyobjbig, method=meths, bignum=1e+10)
```

```
## Warning in optimr(par, fn, gr, method = meth, lower = lower, upper =
## upper, : Successful convergence Restarts for stagnation =0
```

```
print(summary(solb, order=value, par.select=1:2))
```

```
##                    p1        p2      value fevals gevals convergence  kkt1
## nmkb        0.4747329 0.7745629 -0.6634911   1025     NA           0 FALSE
## Nelder-Mead 0.5026671 0.8694462 -0.6495846   1396     NA           0 FALSE
## hjkb        0.4904883 0.8487049 -0.6494160    740     NA           0 FALSE
## newuoa      0.4961325 0.8482360 -0.6364518    151     NA           0 FALSE
##              kkt2 xtime
## nmkb        FALSE 0.168
## Nelder-Mead FALSE 0.148
## hjkb        FALSE 0.076
## newuoa      FALSE 0.020
```

The two Nelder-Mead inspired codes are the best of a bad lot here, with Kelley's variant (`nmkb`) doing a little better, though neither has got really close to the solution for the small hexagon problem. Note how different the solutions appear (we only include the first 2 parameters to save space). Let us draw them.

```
NMpar <- unlist(solb["Nelder-Mead",1:9])
nmkbpar <- unlist(solb["nmkb",1:9])
print(NMpar)
```

```
##        p1        p2        p3        p4        p5        p6        p7
## 0.5026671 0.8694462 1.0000000 0.8564697 0.4877712 0.5194433 0.5395326
##        p8        p9
## 0.5278314 0.5106744
```

```
cat("Nelder-Mead area=", polyarea(NMpar))
```

```
## Nelder-Mead area= 0.6495846
```

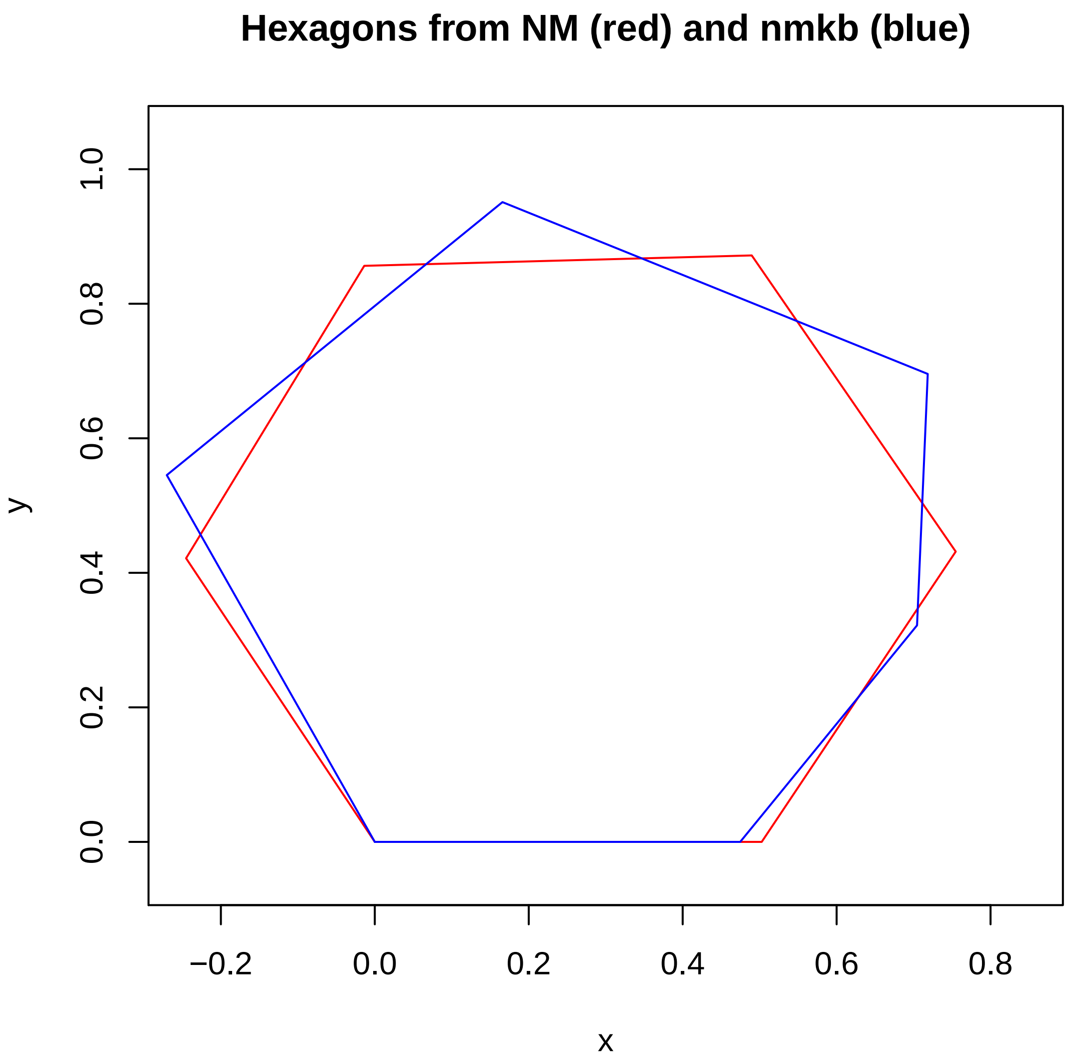
```
print(nmkbpar)
```

```
##        p1        p2        p3        p4        p5        p6        p7
## 0.4747329 0.7745629 0.9999993 0.9654443 0.6085645 0.4286187 0.3406976
##        p8        p9
## 0.6288541 0.6327525
```

```
cat("nmkb area=", polyarea(nmkbpar))
```

```
## nmkb area= 0.6634911
```

```
NMXY <- polypar2XY(NMpar)
nmkbXY <- polypar2XY(nmkbpar)
plot(NMXY$x, NMXY$y, col="red", type="l", xlim=c(-.25,0.85), ylim=c(-.05, 1.05), xlab="x", ylab="y")
points(nmkbXY$x, nmkbXY$y, col="blue", type="l")
title(main="Hexagons from NM (red) and nmkb (blue)")
```



Caution: If the x and y scales of the plot surface are not equal, these drawings will not give a clear view of the results.

Drawing these polygons so we can visually compare them involves transformations that must align the polygons so one edge is on the x axis. But which edge to choose as the first? Then we must note that a vertical reflection of the polygon about the mid-point of the chosen edge will result in an equivalent solution. These options remind us that the optimization problem has multiple solutions, all with equal optimal area, which is part of the difficulty of the largest small polygon problem.

Quadratic penalty function

Our first try (which we will state in advance does not work well) is to add a multiple of the sum of the distance violations. These are the pairwise squared distances for those inter-vertex distances that are not given by the radial parameters. We assume the simple bounds are in force for the radial and angular parameters. This results in the following objective function with its associated gradient.

```
polyobjq <- function(x, penfactor=0, epsilon=0) {
 # negative area + penfactor*(sum(squared violations))
 nv = (length(x)+3)/2 # number of vertices
 area  <-  polyarea(x) # negative area
 f <- -area
 XY <- polypar2XY(x)
 dist2 <- polydistXY(XY)
 viol <- dist2[which(dist2 > 1)] - 1.0
 f <- f + penfactor * sum(viol)
 slacks <- 1.0 + epsilon - dist2 # slack vector
 if (any(slacks <= 0)) {
    attr(f,"area") <- -area
 } # in case of step into infeasible zone
 else {
    attr(f,"area") <- area
 }
 attr(f,"minslack") <- min(slacks)
 f
}
```

Setting the penalty factor (`penfactor`) at 100, we use M. J. Powell's bobyqa minimizer to try to find the solution.

```
start <- myhex$par0 # starting parameters (slightly reduced regular hexagon)
lb <- myhex$lb
ub <- myhex$ub
cat("Starting parameters:")
```

```
## Starting parameters:
```

```
print(start)
```

```
## [1] 0.4900000 0.8487049 0.9800000 0.8487049 0.4900000 0.5235988 0.5235988
## [8] 0.5235988 0.5235988
```

```
library(minqa)
```

```
## Loading required package: Rcpp
```

```
cat("Attempt with quadratic penalty\n")
```

```
## Attempt with quadratic penalty
```

```
sol1 <- bobyqa(start, polyobjq, lower=lb, upper=ub, control=list(iprint=2), penfactor=100)
```

```
## npt = 11 , n =  9
## rhobeg =  0.196 , rhoend =  1.96e-07
## ctrl$force.start =  FALSE
##    0.020:  12:    -0.598980;0.490000 0.804000  1.00000 0.804000 0.490000 0.523599 0.523599 0.523599
##   0.0020:  28:    -0.619909;0.489342 0.824029  1.00000 0.801911 0.492375 0.544427 0.536770 0.533561
##  0.00020:  39:    -0.620960;0.491029 0.822850 0.999660 0.803549 0.493529 0.544781 0.536990 0.535136
##  2.0e-05: 123:    -0.640065;0.512451 0.833198  1.00000 0.798692 0.494576 0.559090 0.521389 0.569904
##  2.0e-06: 134:    -0.640069;0.512438 0.833189  1.00000 0.798724 0.494587 0.559084 0.521398 0.569898
##  2.0e-07: 153:    -0.640096;0.512442 0.833206 0.999998 0.798741 0.494597 0.559092 0.521412 0.569909
## At return
## 222:   -0.64010710: 0.512441 0.833218  1.00000 0.798738 0.494595 0.559108 0.521424 0.569917 0.553680
```

```
print(sol1)
```

```
## parameter estimates: 0.512441454563166, 0.83321811734493, 0.999999713433929, 0.798738098337695, 0.49
## objective: -0.640107103929075
## number of function evaluations: 222
```

```
cat("area = ",polyarea(sol1$par),"\n")
```

```
## area =  0.6401071
```

The objective is the negative area PLUS the penalty, so (-1) times this value is a lower bound on the area. But we see that it is smaller than the reference value (approximately 0.64952) of the unit regular hexagon. We also display the computed area directly, and it shows that the constraint penalty is not appreciably contributing to the objective. We clearly have more work to do.

Logarithmic barrier constraint

A different kind of penalty is provided by the logarithmic barrier. This aims to keep the parameters feasible by adding a steeply increasing function to the objective (the negative area) as the constraint is approached. Let us first define the **slack** in a distance constraint as `slack = (1 - squared.distance)`. We could use the distance itself, but might as well avoid the extra computation. As each slack goes to zero, then `- log(slack)` goes to infinity. We can scale this, as in the quadratic penalty, with `penfactor`, but the actual numerical value will be much smaller now because `- log()` increases much more quickly than the quadratic.

There is an annoying computational practicality that some optimization methods may take steps in the parameter vector that push some distances into infeasible territory. This will cause exceptions to be generated when the logarithm of a negative "slack" is attempted. To avoid this, we will simply make the objective function very large at any time when there is a violation. However, this does cause grief for the evaluation of numerical approximations to gradients, so we may want to revise this policy later and seek more elegant (but likely more complicated) techniques to deal with this possibility. The large number for now will be `bignum = 1e20`. The code polyobjjp.R attempts to avoid some of these difficulties, but there is still work to do.

Since the logarithmic barrier does not let the parameters end actually ON then bound, we may wish to move the constraint an epsilon beyond 1 by redefining the slack as

`slack = (1 + epsilon - squared.distance)`

But what should `epsilon` be? We may revisit this later, but for the moment set the value at 0, and the resulting code is as follows, with its associated gradient. And here is the gradient code. Note that we need to be careful about the indexing. We don't show it here, but we did perform a quick check with package `numDeriv` that the gradient is correctly computed for the starting vector `x0`.

```
polyobj <- function(x, penfactor=1e-8, epsilon=0) {
# log barrier objective function for small polygon
# epsilon <- 0
 bignum <- 1e+20
 # (negative area) + penfactor*(sum(squared violations))
 nv = (length(x)+3)/2 # number of vertices
 area <- polyarea(x) # area
 f <- - area
 dist2 <- polypardist2(x) # from radial coords, excluding radii (bounded)
 slacks <- 1.0 + epsilon - dist2 # slack vector
 if (any(slacks <= 0)) {
#     cat("polygrad: Infeasible parameters at\n")
#     print(x)
     f <- bignum
     area <- -area # to code for infeasible and avoid plotting
 } # in case of step into infeasible zone
 else {  f <- f - penfactor*sum(log(slacks)) }
 attr(f,"area") <- area
 attr(f,"minslack") <- min(slacks)
 f
}
```

```
polygrad <- function(x, penfactor=1e-8, epsilon=0) {
# log barrier gradient function for small polygon
 nv <- (length(x)+3)/2
 l8 <- nv - 3 # end of radii params
# epsilon <- 0
 bignum <- 1e+20
 # (negative area) + penfactor*(sum(squared violations))
 nn <- length(x)
 gg <- rep(0, nn)
 dist2 <- polypardist2(x) # from radial coords, excluding radii (bounded)
 slacks <- 1.0 + epsilon - dist2 # slack vector
 if (any(slacks <= 0)) {
    cat("polygrad: Infeasible parameters at\n")
    print(x)
    stop("polygrad: Infeasible")
 }
 for (ll in 3:nv) {
    ra<-x[ll-1]
    rb<-x[ll-2]
    abangle <- x[l8 + ll]
    # are is 0.5*ra*rb*sin(abangle)
    gg[ll-2] <- gg[ll-2] - 0.5*ra*sin(abangle)
    gg[ll-1] <- gg[ll-1] - 0.5*rb*sin(abangle)
    gg[ll+l8] <- gg[ll+l8] - 0.5*ra*rb*cos(abangle)
 }
 ll <- 0
 for (ii in 2:(nv-1)){
    for (jj in (ii+1):nv) {
       ll <- ll+1
       ra <- x[ii-1]
       rb <- x[jj-1]
       angleab <- 0
       for (kk in (ii+1):jj) { angleab <- angleab + x[kk+l8] }
       gg[ii-1] <- gg[ii-1] + 2*penfactor*(ra-rb*cos(angleab))/slacks[ll]
       gg[jj-1] <- gg[jj-1] + 2*penfactor*(rb-ra*cos(angleab))/slacks[ll]
       for (kk in (ii+1):jj){
          gg[kk+l8]<-gg[kk+l8]+2*penfactor*ra*rb*sin(angleab)/slacks[ll]
       }
    }
 }
 gg
}
```

To allow for unconstrained minimizers to act on this problem, we include the radial parameters in the logarithmic barriers, rather than use traditional active-set bounds. This results in the code polyobju.R.

```
polyobju <- function(x, penfactor=1e-5, epsilon=0, penv) {
  # polyobj with radial parameters constrained by log barrier
  # epsilon <- 0
  bignum <- 1e+20
  # (negative area) + penfactor*(sum(squared violations))
  nv = (length(x)+3)/2 # number of vertices
  area <-  polyarea(x)
  f <- -area # negative area
  dist2 <- polypardist2(x) # from radial coords, excluding radii (bounded)
  dist2 <- c(x[1:(nv-1)]^2, dist2) # Add in radials. Note the squared distances used
  slacks <- 1.0 + epsilon - dist2 # slack vector
  if (any(slacks <= 0)) {
    f <- bignum
    area <- -area # invalid polygon
    attr(f,"area") <- area
  } # in case of step into infeasible zone
  else {
    f <- f - penfactor*sum(log(slacks))
    attr(f,"area") <- area
    addplot(penv, x, f, area)
  }
  attr(f,"minslack") <- min(slacks)
  f
}
```

```
polygradu <- function(x, penfactor=1e-8, epsilon=0, penv) {
  nv <- (length(x)+3)/2
  l8 <- nv - 3 # end of radii params
  # epsilon <- 0
  # (negative area) + penfactor*(sum(squared violations))
  nn <- length(x)
  gg <- rep(0, nn)
  dist2 <- polypardist2(x) # from radial coords, excluding radii (bounded)
  dist2 <- c(x[1:(nv-1)]^2, dist2)
  slacks <- 1.0 + epsilon - dist2 # slack vector

  if (any(slacks <= 0)) { # Leave gradient at 0, rely on bignum in polyobju
    cat("polygrad: Infeasible parameters at\n")
    print(x)
    oldw <- getOption("warn")
    options(warn = -1)
    warning("Polygradu -- Infeasible")
    options(warn = oldw)
  } else {
    for (ll in 3:nv) {
      ra<-x[ll-1]
      rb<-x[ll-2]
      abangle <- x[l8 + ll]
      # are is 0.5*ra*rb*sin(abangle)
      gg[ll-2] <- gg[ll-2] - 0.5*ra*sin(abangle)
      gg[ll-1] <- gg[ll-1] - 0.5*rb*sin(abangle)
      gg[ll+l8] <- gg[ll+l8] - 0.5*ra*rb*cos(abangle)
    }
  }
  ll <- 0
  # components from radial parameter constraints (upper bounds)
  for (ii in 1:(nv-1)){
    ll <- ll+1
    gg[ll] <- gg[ll] + 2*penfactor*x[ll]/slacks[ll]
  }
  # components from other distances
  for (ii in 2:(nv-1)){
    for (jj in (ii+1):nv) {
      ll <- ll+1
      ra <- x[ii-1]
      rb <- x[jj-1]
      angleab <- 0
      for (kk in (ii+1):jj) { angleab <- angleab + x[kk+l8] }
      gg[ii-1] <- gg[ii-1] + 2*penfactor*(ra-rb*cos(angleab))/slacks[ll]
      gg[jj-1] <- gg[jj-1] + 2*penfactor*(rb-ra*cos(angleab))/slacks[ll]
      for (kk in (ii+1):jj){
        gg[kk+l8]<-gg[kk+l8]+2*penfactor*ra*rb*sin(angleab)/slacks[ll]
      }
    }
  }
  gg
}
```

Finally, our first attempt to overcome non-computability when a step crosses the constraint boundary and the log barrier is inadmissible.

Running the log barrier constrained functions

With Powell's bobyqa, we attempt to minimize this objective from the same start as before. We set the penalty factor quite small, in fact 0.01, as the barrier is non-zero within the feasible region.

```
## @knitr polyex2
library(minqa)
cat("Attempt with logarithmic barrier polyobj.R\n")
```

```
## Attempt with logarithmic barrier polyobj.R
```

```
x0 <- myhex$par0 # starting parameters (slightly reduced regular hexagon)
lb <- myhex$lb
ub <- myhex$ub
cat("Starting parameters:")
```

```
## Starting parameters:
```

```
print(x0)
```

```
## [1] 0.4900000 0.8487049 0.9800000 0.8487049 0.4900000 0.5235988 0.5235988
## [8] 0.5235988 0.5235988
```

```
sol2 <- bobyqa(x0, polyobj, lower=lb, upper=ub, control=list(iprint=2), penfactor=1e-3)
```

```
## npt = 11 , n =  9
## rhobeg =  0.196 , rhoend =  1.96e-07
## ctrl$force.start =  FALSE
##    0.020:  12:    -0.588498;0.490000 0.804000  1.00000 0.804000 0.490000 0.523599 0.523599 0.523599 
##   0.0020:  27:    -0.604093;0.499975 0.846173  1.00000 0.823279 0.483358 0.516184 0.526795 0.518259 
##  0.00020:  46:    -0.625888;0.507189 0.860220  1.00000 0.836838 0.491940 0.521810 0.539439 0.531134 
##  2.0e-05:  75:    -0.629827;0.511862 0.864128  1.00000 0.840691 0.493209 0.524863 0.542018 0.534689 
##  2.0e-06: 2387:    -0.650552;0.639200  1.00000  1.00000 0.786610 0.554503 0.665309 0.560473 0.311998
##  2.0e-07: 3201:    -0.650565;0.636984  1.00000  1.00000 0.792990 0.558537 0.664535 0.560359 0.305870
## At return
## 3961:   -0.65056501: 0.637031  1.00000  1.00000 0.793361 0.558558 0.664509 0.560293 0.305339 0.41622
```

```
print(sol2)
```

```
## parameter estimates: 0.637030820202343, 1, 1, 0.793360773838786, 0.558558288879639, 0.66450877965671
## objective: -0.650565009139302
## number of function evaluations: 3961
```

```
cat("Area found=",polyarea(sol2$par),"\n")
```

```
## Area found= 0.670969
```

This is not too bad. We can save the result then try with a smaller penalty factor.

```
x0a <- sol2$par
sol2a <- bobyqa(x0a, polyobj, lower=lb, upper=ub, control=list(iprint=2), penfactor=1e-6)
```

```
## npt = 11 , n =  9
## rhobeg =  0.2 , rhoend =  2e-07
## ctrl$force.start =  FALSE
##    0.020:  12:    -0.670949;0.637031  1.00000  1.00000 0.793361 0.558558 0.664509 0.560293 0.305339
##   0.0020:  15:    -0.670949;0.637031  1.00000  1.00000 0.793361 0.558558 0.664509 0.560293 0.305339
##  0.00020:  32:    -0.671863;0.638073  1.00000  1.00000 0.794591 0.559395 0.664587 0.560642 0.305090
##  2.0e-05:  45:    -0.672442;0.638447  1.00000  1.00000 0.794616 0.559680 0.665070 0.561034 0.305482
##  2.0e-06:  64:    -0.672567;0.638510  1.00000  1.00000 0.794669 0.559712 0.665121 0.561120 0.305559
##  2.0e-07: 103:    -0.672676;0.638565  1.00000  1.00000 0.794721 0.559743 0.665162 0.561195 0.305630
## At return
## 126:   -0.67268438: 0.638570  1.00000  1.00000 0.794725 0.559746 0.665165 0.561202 0.305636 0.416508
```

```
print(sol2a)
```

```
## parameter estimates: 0.638570222992048, 1, 1, 0.794725053365082, 0.559745952614796, 0.66516519894800
## objective: -0.672684376780446
## number of function evaluations: 126
```

```
cat("Area found=",polyarea(sol2a$par),"\n")
```

```
## Area found= 0.6727129
```

And again, reducing the penfactor to 1e-9.

```
x0b <- sol2a$par
sol2b <- bobyqa(x0b, polyobj, lower=lb, upper=ub, control=list(iprint=2), penfactor=1e-9)
```

```
## npt = 11 , n =  9
## rhobeg =  0.2 , rhoend =  2e-07
## ctrl$force.start =  FALSE
##    0.020:  12:    -0.672713;0.638570  1.00000  1.00000 0.794725 0.559746 0.665165 0.561202 0.305636
##   0.0020:  15:    -0.672713;0.638570  1.00000  1.00000 0.794725 0.559746 0.665165 0.561202 0.305636
##  0.00020:  32:    -0.673085;0.638786  1.00000  1.00000 0.792886 0.560085 0.665504 0.562955 0.305975
##  2.0e-05:  44:    -0.673250;0.638350 0.999975 0.999912 0.792655 0.560491 0.665975 0.563454 0.305807
##  2.0e-06:  55:    -0.673251;0.638343 0.999977 0.999909 0.792672 0.560488 0.665980 0.563438 0.305818
##  2.0e-07:  74:    -0.673324;0.638369  1.00000 0.999940 0.792693 0.560501 0.666001 0.563472 0.305849
## At return
##  97:   -0.67332988: 0.638371  1.00000 0.999943 0.792695 0.560503 0.666003 0.563475 0.305851 0.415825
```

```
print(sol2b)
```

```
## parameter estimates: 0.638371107806375, 0.999999986222725, 0.999942803383823, 0.792695070477583, 0.5
## objective: -0.67332988391386
## number of function evaluations: 97
```

```
cat("Area found=",polyarea(sol2b$par),"\n")
```

```
## Area found= 0.6733299
```

But a further attempt does very poorly.

```
x0c <- sol2b$par
sol2c <- bobyqa(x0c, polyobj, lower=lb, upper=ub, control=list(iprint=2), penfactor=1e-12)
```

```
## npt = 11 , n =  9
## rhobeg =  0.2 , rhoend =  2e-07
## ctrl$force.start =  FALSE
##    0.020:  12:    -0.596071;0.638371  1.00000 0.800000 0.792695 0.560503 0.666003 0.563475 0.305851 
##   0.0020:  25:    -0.596071;0.638371  1.00000 0.800000 0.792695 0.560503 0.666003 0.563475 0.305851 
##  0.00020:  40:    -0.596853;0.640432  1.00000 0.801867 0.791897 0.561442 0.662747 0.563765 0.306243 
##  2.0e-05:  49:    -0.596944;0.640496 0.999992 0.802040 0.791888 0.561429 0.662812 0.563664 0.306221 
##  2.0e-06:  60:    -0.596954;0.640475 0.999995 0.802041 0.791904 0.561479 0.662803 0.563601 0.306298 
##  2.0e-07:  71:    -0.596954;0.640474 0.999996 0.802041 0.791904 0.561479 0.662804 0.563600 0.306298 
## At return
##  87:   -0.59695455: 0.640474 0.999996 0.802041 0.791904 0.561479 0.662804 0.563600 0.306298 0.416339
```

```
print(sol2c)
```

```
## parameter estimates: 0.640474391667351, 0.999996417062993, 0.802041283838145, 0.791903551091848, 0.5
## objective: -0.596954550526281
## number of function evaluations: 87
```

```
cat("Area found=",polyarea(sol2c$par),"\n")
```

```
## Area found= 0.5969546
```

Possibly another method could do better.

In polyobju.R, we adjust the objective so the radial parameters are constrained using a logarithmic barrier, then use an unconstrained optimization method. There are a number of unconstrained optimization methods, and the `optimrx` package lets us try them out all at once. To avoid too much computing time, we use three methods, all of which could use bounds if we provided them.

```
library(optimrx)
methset <- c("Rvmmin", "L-BFGS-B", "nlminb")
suall <- opm(x0, polyobju, polygradu, method=methset, control=list(trace=0, kkt=FALSE), penfactor=1e-5)
# NOTE: Got complex Hessian eigenvalues when trying for KKT tests
suall <- summary(suall, order=value)
print(suall)
```

```
##             p1        p2   p3        p4   p5        p6        p7        p8
## L-BFGS-B   NA        NA   NA        NA   NA        NA        NA        NA
## nlminb     NA        NA   NA        NA   NA        NA        NA        NA
## Rvmmin   0.49 0.8487049 0.98 0.8487049 0.49 0.5235988 0.5235988 0.5235988
##                 p9         value fevals gevals convergence kkt1 kkt2 xtime
## L-BFGS-B        NA 8.988466e+307     NA     NA        9999   NA   NA 0.004
## nlminb          NA 8.988466e+307     NA     NA        9999   NA   NA 0.004
## Rvmmin   0.5235988 1.797693e+308      1      0          20   NA   NA 0.004
```

```
resu <- coef(suall)
nmeth <- dim(resu)[1]
best0 <- resu[1,]
```

```
polyarea(best0)
```

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

This is quite good, and reasonably fast, though we note that the solver has terminated on too many gradients in the two best cases. Nevertheless, we have the area to 4 decimals. We can, of course, use the explicit bounds with these particular methods. Let us see how they perform.

Check with bounds

```
sall <-  opm(x0, polyobj, polygrad, method=methset, lower=lb, upper=ub,
             control=list(trace=0, kkt=FALSE), penfactor=1e-5)
```

```
## polygrad: Infeasible parameters at
## [1] 0.7018826 1.0000000 1.0000000 1.0000000 0.7018826 0.7034137 0.8832286
## [8] 0.8832286 0.7034137
```

```
sall <- summary(sall, order=value)
print(sall)
```

```
##                  p1        p2 p3        p4        p5        p6        p7
## Rvmmin    0.3491722 0.7792728  1 0.7792728 0.3491722 0.6983897 0.6965715
## nlminb    0.3998874 0.8245674  1 0.7444245 0.2969221 0.6926894 0.6762570
## L-BFGS-B         NA        NA NA        NA        NA        NA        NA
##                  p8        p9          value fevals gevals convergence kkt1
## Rvmmin    0.6965715 0.6983897  -6.744546e-01    314    124           3   NA
## nlminb    0.7026573 0.6850014  -6.733607e-01    339    151           1   NA
## L-BFGS-B         NA        NA  8.988466e+307     NA     NA        9999   NA
##           kkt2 xtime
## Rvmmin      NA 0.124
## nlminb      NA 0.080
## L-BFGS-B    NA 0.004
```

```
best1 <- coef(sall)[1,]
polyarea(best1)
```

```
## [1] 0.6749314
```

Here, one of the methods – L-BFGS-B – has failed. Note that we did get a warning that the gradient was infeasible, and it is likely the method has stepped into the infeasible region. We also note that once again, the other two methods have hit gradient evaluation limits. Such limits are INSIDE the methods, though there are ways to set them. However, most users will not bother to learn how to do this (it is often quite obscure), and we believe that tests should be with stock versions of codes.

We note also that the best result is not quite so good as with the function polyobju.R.

All methods available to optimrx

We can use the function polyobju.R with all the methods available in the package **optimrx** from https://r-forge.r-project.org/projects/optimizer/. The results of running all the methods from `x0` where

```
> print(polyobju(x0, penfactor=1e-5))
[1] -0.6236083
attr(,"area")
[1] 0.6237981
```

```
attr(,"minslack")
[1] 0.0396
```

are as follows (run on machine J6, 2016-11-29):

```
p9      value fevals gevals convergence kkt1 kkt2 xtime
ucminf      0.1757560 -0.6743258    402    402           0   NA   NA 0.186
nlm         0.6984392 -0.6743258     NA    471           0   NA   NA 0.522
Rvmmin      0.6984390 -0.6743258    386   1501           1   NA   NA 0.563
Rcgmin      0.6984414 -0.6743258   3023   1501           1   NA   NA 0.819
BFGS        0.6979991 -0.6743241    342     83           0   NA   NA 0.062
Rtnmin      0.6987669 -0.6743190    630    630           2   NA   NA 1.128
hjkb        0.7062961 -0.6743079  11018     NA           0   NA   NA 1.135
nlminb      0.6299766 -0.6704514    317    151           1   NA   NA 0.077
CG          0.6202124 -0.6695781   4706   1501           1   NA   NA 0.915
nmkb        0.6513722 -0.6630782   1225     NA           0   NA   NA 0.205
hjn         0.3795476 -0.6607589  16223     NA           0   NA   NA 1.852
newuoa      0.5264272 -0.6507394  15000     NA           1   NA   NA 1.729
Nelder-Mead 0.5199981 -0.6496394   1502     NA           1   NA   NA 0.146
spg         0.5278788 -0.6491492   1816   1501           1   NA   NA 0.812
bobyqa      0.5383891 -0.6449078    409     NA           0   NA   NA 0.042
L-BFGS-B    0.5235988 -0.6236083      3      3           0   NA   NA 0.000
lbfgsb3     0.5235988 -0.6236083      3      3           0   NA   NA 0.004
lbfgs       0.5235988 -0.6236083     NA     NA       -1001   NA   NA 0.000
```

We see that gradient methods occupy the top 6 positions, but also the lbfgs family of methods has not been able to proceed at all.

Issues in this specification and minimization

The experiences above show that the minimization is sensitive to how we set up the problem and to the choice of the penalty factor. Some of the failures of methods with logarithmic barrier methods suggest that methods can easily step into infeasible territory. A private communication from Prof. S. G. Nash of George Mason University pointed out that the truncated Newton (and similarly the lbfgs family) methods need to employ a modified line search when the logarithmic barrier function is applied. See S. G. Nash and Sofer (1993).

We have not (yet) investigated the use of a transfinite function approach to this problem? This transforms the range (a, b) into (-Inf, Inf) using an inverse hyperbolic tangent, thereby converting a bounds constrainted problem to an unconstrained one. This is, in fact, the method used by`nmkb()` from package dfOptim, though we have not applied that method here. The transformation can destabilize the optimization, and we still have to use a barrier or penalty function to handle the bounds that are not directly imposed on parameters.

Animating the progress to an optimum

If we want to visualize the progress of our optimization, then we need to draw the polygons as the relevant parameter vectors are tried. However, we probably only want to draw the polygon when we have found a feasible one that increases the area. Furthermore, as we progress, it is helpful to indicate the rank of the polygons in order of area. This can be accomplished by "fading out" polygons that are already drawn, though that means keeping a record of at least some of the feasible, larger-area polygons.

We can achieve all these desiderata by means of the following function.

```
# 160807 -- need to look at symmetry of result
#           Alternative parametrization
```

```r
nvex <- 6 # default to hexagon

# Now try to ONLY plot "best so far" polygons


addplot <- function(penv, x, f, area) {
   val <- 1 # OK if returns 0
   nplot <- 5
   ncol <- dim(penv$psave)[2] - 2
#- To add point to the penv$psave matrix and plot the
#- last nplot polygons
   npoint <- dim(penv$psave)[1]
   if (area > penv$besta) {
     penv$besta <- area
     penv$psave <- rbind(penv$psave, c(x, f, area))
     if (npoint > nplot + 1) {
       nset <- nplot
       prh <- penv$psave[1,1:ncol]
       xyrh <- polypar2XY(prh)
       plot.new()
       plot.window(xlim = c(-0.5, 0.7), ylim=c(-0.1, 1.1))
       txt<-paste("Area to reg. polygon = ",area/penv$regarea,sep='')
       title(txt)
       box()
       axis(1)
       axis(2)
       points(xyrh, col='pink', type='l', lwd=2, )
       colrs <- hsv(0.6, (1:nset)/nset, 1, 0.5)
       for (ii in 1:nset) {
         ppoint <- penv$psave[npoint-ii+1, 1:ncol]
         xy <- polypar2XY(ppoint)
         points(xy, col=colrs[ii], type='l', lwd=2)
       }
     }
   }
   val <- 0
}
```

Calling the PolyTrack function creates a new environment to hold all the sets of points. Each time a new "better" polygon is found, the `nPolys` most recent (the code above uses 5) polygons are redrawn using the vector of (fading) colours.

As of Nov 10, 2016, replay of the stored points with `tkrplot` does not seem to work reliably.

Saved results

Because the code actually builds a list of the points where the polygon is "better", we can play back the progress. There are tools in the `tkrplot` package that allow this to be made interactive. Thus, when the optimization terminates, we can redraw the polygons at will using the following code.

Acknowledgement

Email discussions with Prof. S. G. Nash of George Mason University have been helpful in developing some of the ideas in this article.

References

Audet, Charles, Pierre Hansen, and Frédéric Messine. 2007. "Extremal Problems for Convex Polygons." *Journal of Global Optimization* 38 (2): 163–79. doi:10.1007/s10898-006-9065-5.

Dolan, Elizabeth D. Dolan, Jorge J. Moré, and Todd S. Munson. 2004. "Benchmarking Optimization Software with Cops 3.0." Technical memorandum ANL/MCS-TM-273. Argonne National Laboratory.

Graham, R.L. 1975. "The Largest Small Hexagon." *Journal of Combinatorial Theory, Series A* 18 (2): 165–70. doi:http://dx.doi.org/10.1016/0097-3165(75)90004-7.

Nash, John C., and Mary Walker-Smith. 1987. *Nonlinear Parameter Estimation: An Integrated System in Basic*. Book. Marcel Dekker Inc.: New York.

Nash, Stephen G., and Ariela Sofer. 1993. "A Barrier Method for Large-Scale Constrained Optimization." *INFORMS (Formerly ORSA) Journal on Computing* 5 (1): 40–53.