Animating geometric optimization: small polygons

John C. Nash, A. N. Other

2016-05-13

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.

Background

The **The Largest Small Hexagon** is the title of a paper by Ron Graham (J. Combinatorial Theory (A), vol. 18, pp. 165-170, 1975). This did not introduce this problem, but served to bring it to wider attention. One of the authors (JCN) used this problem to illustrate constrained optimization using the tools in J.C. Nash and M. Walker-Smith (1987, Nonlinear parameter estimation: an integrated system in BASIC, now available online at https://archive.org/details/ost-engineering-jnmws2004).

To provide a visual presentation of the optimization, Nash coded a display for the IBM PC family of MS DOS computers running GWBASIC. In May 2016, the discovery that files for this example would still execute more or less satisfactorily raised the possibility of bringing them up to date. R was a logical choice for such an implementation, given that the authors all work with this software system.

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 3 parameters can be fixed right away. That is, we can fix one vertex 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)`.

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 points 2...nv. However, that then requires the angles to be monotonically increasing. By specifying that b[L+lshift] = alpha[L-1] - alpha[L-2] for L=3 ... nv, we automatically get the angles alpha monotonic by imposing a lower bound of 0 on the parameters b.

Note that the radii 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.

Problem setup

The above parametrization is implemented in the function `polysetup(nv, defsize, qpen)`, where `defsize` is the default "size" of a regular polygon for which initial parameters are established. `qpen` is a multiplicative factor for a penalty function that may be used to impose the distance constraints. However, the form of this penalty is not yet given.

```
polysetup <- function(nv, defsize=0.98, qpen=.1){
# 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" ....???
#    nv <- readline("number of vertices = ")
    nvmax <- 100 # Arbitrary limit -- change ??
    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 <- 0.98
    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 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 vertex 1 with vertex L and (L+1) where L runs from 2 to (nv-1).

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

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 with joining lines automatically gives the closed figure polygon.

```
polypar2XY <- function(nv, b) {
    l8 <- nv - 3 # offset for indexing
    x <- rep(NA, nv+1)
    y <- rep(NA, nv+1)
    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+l8]
        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(nv, XY) {
#   compute point to point distances from XY data
   ncon <- (nv - 1)*(nv - 2)/2
   dist2 <- rep(NA, ncon) # squared distances
   ll <- 0 # index of constraint
   for (i in 2:nv){
      for (j in (1:(i-1))){
         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
}
```

Testing the functions presented so far.

A very simple script calling the above functions allows us to verify that they are working as expected. Note that for this script to work properly, we need to ensure that the graph is drawn on a square grid.

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

```
## Polygon data:
```

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

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

```
## Area:
```

```
myhexa <- polyarea(nv, myhex$par0)
print(myhexa)
```

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

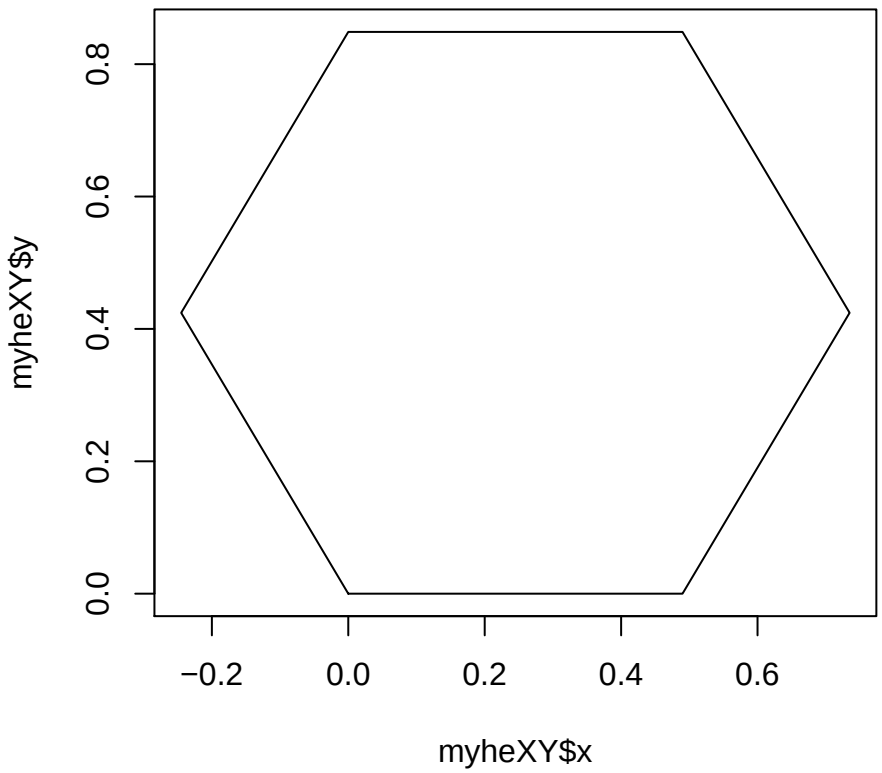
```
cat("XY coordinates\n")
```

```
## XY coordinates
```

```
myheXY <- polypar2XY(nv, myhex$par0)
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(nv, myheXY)
print(myhexc)
```

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

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

```
## Vertex distances:
```

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

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