

Run-and-Twiddle

Mengran Xu

December 17, 2009

Contents

1	Introduction	1
2	Run-and-Twiddle	1
3	The first implementation of RMP	2
3.1	R code implementation	2
3.2	Test	4
4	The second implementation of RMP	4
4.1	R code implementation	5
4.2	Test	7
4.3	Observations and discussion	8
5	Conclusions	8

1 Introduction

This is my report for the 8th assignment of CS545 in Colorado State University.

In this report, I implement a machine learning algorithm, Run-and-Twiddle. This algorithm appears in a paper about Biological Cybernetics, named Biologically Plausible Learning in Neural Networks: A Lesson from Bacterial Chemotaxis. I studied this paper, especially on this algorithm.

In the first part of this paper, I introduce this algorithm. Then, I show my first implement and test it using the example from that paper. Finally, I modify the implement above to deal with minimum problem and compare with steepest-descent and with scg algorithm.

2 Run-and-Twiddle

Run-and-Twiddle is introduced as a special method of optimizing the bacterium's spatial coordinates. This special method is called regulation of modification probability (RMP).

Biologically, bacteria usually do not have the capacity of determining the direction either toward a chemo-attractant or away from a repellent. The only information available to them is the temporal gradient of the attractant (or repellent) concentration. In order to be capable of relating that information to a specific direction, they need to be in constant motion. The main strategy of bacterial chemotaxis consists of periods of swimming in a certain direction separated from each other by an abrupt, random change of the direction

(tumbling). If the frequency of tumbling is maintained constant, this strategy results in chaotic, random, search-type movements. In reality, however, the frequency of tumbling depends on the temporal gradient of attractant concentration: it increases or decreases when the gradient of the attractant becomes positive or negative, respectively. In the case of swimming away from a repellent, the dependence of the tumbling frequency on the concentration of the repellent is opposite.

I will utilize the following computational model of bacterial chemotaxis to show this method's detail.

The dynamics of a bacterium's motion in 3D is described by the following equation system:

$$\begin{aligned}x(t + \Delta t) &= x(t) + v \cdot \cos(\psi)\cos(\theta)\Delta t, \\y(t + \Delta t) &= y(t) + v \cdot \sin(\psi)\cos(\theta)\Delta t, \\z(t + \Delta t) &= z(t) + v \cdot \sin(\theta)\Delta t.\end{aligned}$$

where $x(t)$, $y(t)$, and $z(t)$ are the bacterium's spatial coordinates at the time point t , Δt is the time step, v is the movement speed, and the angles ψ and θ represent movement direction: yaw (orientation on a horizontal plane) and elevation, respectively. The dependence of tumbling probability $P(t)$ on the temporal gradient of attractant concentration, $\Delta F(t) = F(t) - F(t - \Delta t)$, is described by a sigmoid nonlinear function ranging between 0 and 1:

$$P = 1/(1 + e^{-k\Delta F(t)}), \tag{1}$$

where k is a constant coefficient. Tumbling was modeled by replacing ψ and θ with new values randomly selected according to a uniform probability distribution within the angle ranges $[0^\circ, 360^\circ]$ and $[-90^\circ, 90^\circ]$, respectively. Thus, the new movement direction was completely independent from the previous direction.

3 The first implementation of RMP

This is my first implementation for RMP. It is used to search the destination from a initial point in a 2-dimensional panel.

3.1 R code implementation

```
#####
### Regulation of Modification Probability (RMP)-2d-goal
#####

#####
### Return new state

nextState <- function(s,dir,dt) {

  ## Update state
  s[1] <- s[1] + s[3] * cos(dir) * dt
  s[2] <- s[2] + s[3] * sin(dir) * dt
  s[4] <- dir

  ## Return new state
  s
}
```

```

#####
### The distance of two points
D <- function(x,y,Xd,Yd) {

  return(sqrt((x-Xd)^2+(y-Yd)^2))
}

#####
### RMP algorithm

rmp <- function(x,y,Xd,Yd,
               v=0.1,dt=1,k=100,
               nIterations=1000,
               Precision=0.01,
               xtracep=TRUE,
               ftracep=TRUE) {

  if (missing(x) || missing(y) || missing(Xd) || missing(Yd) ) {
    cat("Missing x, y, Xd, Yd \n")
    return()
  }

  ## Initial state
  ## s[1] is x-axis position, s[2] is y-axis position, s[3] is velocity, s[4] is direction.
  s <- c(x,y,v,(sample(360,1)*(pi/180)))
  P <- 0.5
  F <- D(x,y,Xd,Yd)

  i <- 1
  xtrace <- NULL
  if (xtracep)
    xtrace <- c(x)
  ftrace <- NULL
  if (ftracep)
    ftrace <- c(y)

  while (i < nIterations) {

    if(P>0.5)
    {
      dir <- sample(360,1)*(pi/180)
      s <- nextState(s,dir,dt)
    }
    else
      s <- nextState(s,s[4],dt)

    newF <- D(s[1],s[2],Xd,Yd)
    dF <- newF - F
    F <- newF

    P <- 1/(1+exp(-k*dF))

    if (i %% (nIterations/10) == 0)

```

```

    cat("RMP: Iteration",i,"\n")
  if (xtracep)
    xtrace <- cbind(xtrace,s[1])
  if (ftracep)
    ftrace <- c(ftrace,s[2])
  if (any(is.nan(s[1])) || is.nan(s[2]))
    stop("Error: RMP produced newx or newy that is NaN. Stepsize may be too large.")
  if (any(is.infinite(s[1])) || is.infinite(s[2]))
    stop("Error: RMP produced newx or newy that is infinite. Stepsize may be too large.")
  if (D(s[1],s[2],Xd,Yd) < Precision)
    return (list(x=s[1], f=s[2], xtrace=xtrace, ftrace=ftrace, reason="precision"))

  i <- i + 1
}
return (list(x=s[1], f=s[2], xtrace=xtrace, ftrace=ftrace, reason="did not converge"))
}

```

3.2 Test

Here, I use the task from the paper to test the implementation above.

The bacterium was required to reach an attractant located at a point 10mm away from the bacterium's initial position. The target point was considered reached if the bacterium came to it closer than 0.1 mm. The model's parameter values were selected as $\Delta t=1$ s, $v=0.1$ mm/s, and $k=100$ (making the elementary step size $v\Delta t=0.1$ mm).

R code for this test is followed.

```

source("RMP-2d.R")

x <- 0
y <- 0

r <- rmp(x,y,10,0,nIterations=500,Precision=0.1)

print(r$reason)
plot(r$xtrace,r$ftrace,type="b",col="blue",xlab="x",ylab="y")
lines(c(10,10),c(-2,6),col="red",lty=2)
points(x,y,col="purple",cex=1.5,pch=19)
points(10,0,col="red",cex=1.5,pch=19)

```

A typical trajectory of the bacterium consisted of an initial part comprising several relatively long segments that quickly brought the bacterium to the target neighborhood, and its more chaotic search in that neighborhood until it found the target (Figure 1).

4 The second implementation of RMP

I modified the R code above to obtain this second implementation. Its purpose is to get a minimum of a function.



Figure 1: Typical trajectory. The dots designate the bacterium's position between movement steps.

4.1 R code implementation

```
#####
### Regulation of Modification Probability (RMP)-2d-min
#####

#####
### Return new state

nextState <- function(s,dir,dt) {

  ## Update state
  s[1] <- s[1] + s[3] * cos(dir) * dt
  ##s[2] <- s[2] + s[3] * sin(dir) * dt
  s[4] <- dir

  ## Return new state
  s
}

#####
### RMP algorithm

rmp <- function(x,f,...,
               v=0.1,dt=1,k=100,
               nIterations=1000,
               xPrecision=0.001*mean(x),
               fPrecision=0.001*mean(f(x,...)),
```

```

        xtracep=TRUE,
        ftracep=TRUE) {

if (missing(x) || missing(f)) {
  cat("Missing x and f \n")
  return()
}

## Initial state
## s[1] is x-axis position, s[2] is y-axis position, s[3] is velocity, s[4] is direction.
s <- c(x,f(x,...),v,(sample(360,1)*(pi/180)))
P <- 0.5
F <- f(x,...)

i <- 1
xtrace <- NULL
if (xtracep)
  xtrace <- c(x)
ftrace <- NULL
if (ftracep)
  ftrace <- c(f(x,...))

while (i < nIterations) {

  if(P>0.5)
  {
    dir <- sample(360,1)*(pi/180)
    s <- nextState(s,dir,dt)
  }
  else
    s <- nextState(s,s[4],dt)

  newx <- s[1]
  newF <- f(s[1],...)
  dF <- newF - F

  P <- 1/(1+exp(-k*dF))

  #if (i %% (nIterations/10) == 0)
  # cat("RMP: Iteration",i,"\n")
  if (xtracep)
    xtrace <- cbind(xtrace,s[1])
  if (ftracep)
    ftrace <- c(ftrace,f(s[1],...))
  if (any(is.nan(s[1])) || is.nan(f(s[1],...)))
    stop("Error: RMP produced newx or newy that is NaN. Stepsize may be too large.")
  if (any(is.infinite(s[1])) || is.infinite(f(s[1],...)))
    stop("Error: RMP produced newx or newy that is infinite. Stepsize may be too large.")
  if (max(abs(newx - x)) < xPrecision)
    return (list(x=newx, f=f(s[1],...), xtrace=xtrace, ftrace=ftrace, reason="x precision"))
  if (abs(newF - F) < fPrecision)
    return (list(x=newx, f=f(s[1],...), xtrace=xtrace, ftrace=ftrace, reason="f precision"))

  x <- newx

```

```

    F <- newF
    i <- i + 1
  }
  return (list(x=s[1], f=f(s[1],...), xtrace=xtrace, ftrace=ftrace, reason="did not converge"))
}

```

4.2 Test

Here, I use a parabola function to test this second implementation and compare its convergence speed with steepest-descent and with scg algorithm.

The relevant R code to test is followed.

```

source("gradientDescents.R")
source("RMP-2d-min.R")

```

```

parabola <- function(x,xmin,s) {
  d <- x - xmin
  t(d) %>% s %>% d / 20 + sin(x*2)
}
parabolaGrad <- function(x,xmin,s) {
  d <- x - xmin
  2 * s %>% d / 20 + 2*cos(x*2)
}

f <- parabola
df <- parabolaGrad
center <- 5
S <- 3

xs <- seq(0,10,len=100)
fx <- NULL
fg <- NULL
for (x in xs) {
  fx <- c(fx,f(x,center,S))
  fg <- c(fg,df(x,center,S))
}
plot(xs,fx,type="l");

loc <- 1
while (!is.null(loc)) {

  cat("Click left button to start search. To stop, click right.\n");
  loc <- locator(1)
  if (!is.null(loc)) {

    firstx <- loc$x

    resultSCG <- scg(firstx, f, df, center, S, xPrecision=0.001, xtracep=TRUE)
    print(resultSCG$x)
    resultSteepest <- steepest(firstx,f,df,center,S,stepsize=0.1,
                              xPrecision=0.001, xtracep=TRUE)

```

```

resultRMP <- rmp(firstx,f,center,S,xPrecision=0.001)

plot(xs,fx,type="l")

## first steepest
fsearch <- c()
r <- resultSteepest
for (x in r$xtrace)
  fsearch <- c(fsearch,f(x,center,S))
lines(r$xtrace,fsearch,col=2)
points(r$xtrace,fsearch,col=2)

r <- resultSCG
fsearch <- c()
for (x in r$xtrace)
  fsearch <- c(fsearch,f(x,center,S))
lines(r$xtrace,fsearch,col=4)
points(r$xtrace,fsearch,col=4)

r <- resultRMP
lines(r$xtrace,r$ftrace,col=7)
points(r$xtrace,r$ftrace,col=7)

text(2,3,paste(length(resultSteepest$xtrace),"steps for Steepest Descent"),pos=4,col=2)
text(2,2.5,paste(length(resultSCG$xtrace),"steps for SCG"),pos=4,col=4)
text(2,2,paste(length(resultRMP$xtrace),"steps for RMP"),pos=4,col=7)
}
}

```

Figure2 show the test results.

4.3 Observations and discussion

Based on the output result above, the performance of scg and steepest-descent is much better than RMP. However, the final results also depend on the parameter selection for each algorithms. Besides, occasionally, in some cases, RMP's convergence speed may be faster.

5 Conclusions

By this report, I learn Run-and-Twiddle algorithm and implement it all by myself. This is a valuable experience.

Why do I pick this topic? The main reason is for my instructor. When I was thinking hard which topic I should select, the teacher's idea woke me up. He pointed a algorithm from a paper, which is interesting and easy. After I learn the paper generally, I'm happy to find this is the topic I want. Thanks my teacher!

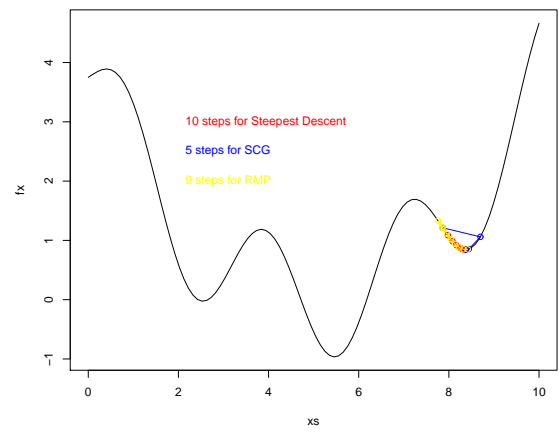
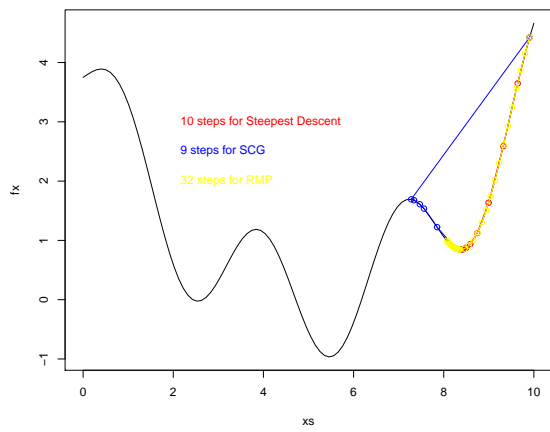
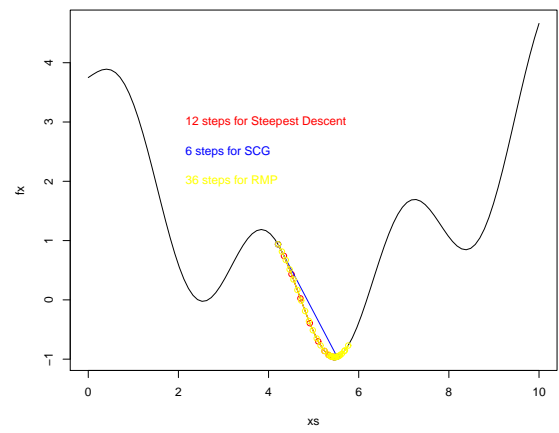
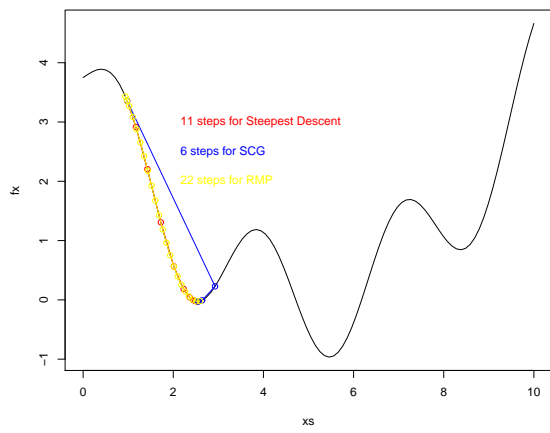


Figure 2: The convergence test of scg, steepest-descent and RMP

References

- [1] Yury P. Shimansky, *Biological plausible learning in neural networks: a lesson from bacterial chemotaxis*, Biological Cybernetics 101:379-385, 2009.
- [2] Anderson, C., *Scaled Conjugate Gradient*, <http://www.cs.colostate.edu/~anderson/cs545/Lectures/week6day2/week6day2.pdf>, 2009.
- [3] Anderson, C., *gradientDescents.R*, <http://www.cs.colostate.edu/~anderson/cs545/Lectures/week6day1/gradientDescents.R>, 2009.
- [4] Anderson, C., *scgTest1d.R*, <http://www.cs.colostate.edu/~anderson/cs545/Lectures/week6day1/scgTest1d.R>, 2009.