Final Exam

Yuan Hu

Problem 1

(a)

The objective function u is a convex function (if to consider the negative sign, it should be concave). This is clear to see by defining dx_i=x_i-x_{i-1}. The original function becomes a quadratic function of dx_i plus |dx_i|^(1+beta) term. Because |dx_i|^(1+beta) is convex, the objective function u (as sum of two convex functions) is convex.

I am coding in R.

I first tried implementing the objective function and using optim with "BFGS" method to maximize the objective function. The only trick in the implementation is to scale x_t because large number causes problem in numeric calculation. Scaling is done in the objective function. After optimizer returns results, I scale parameters back. The problem with this approach is that there is no control on the precision of individual x_t.

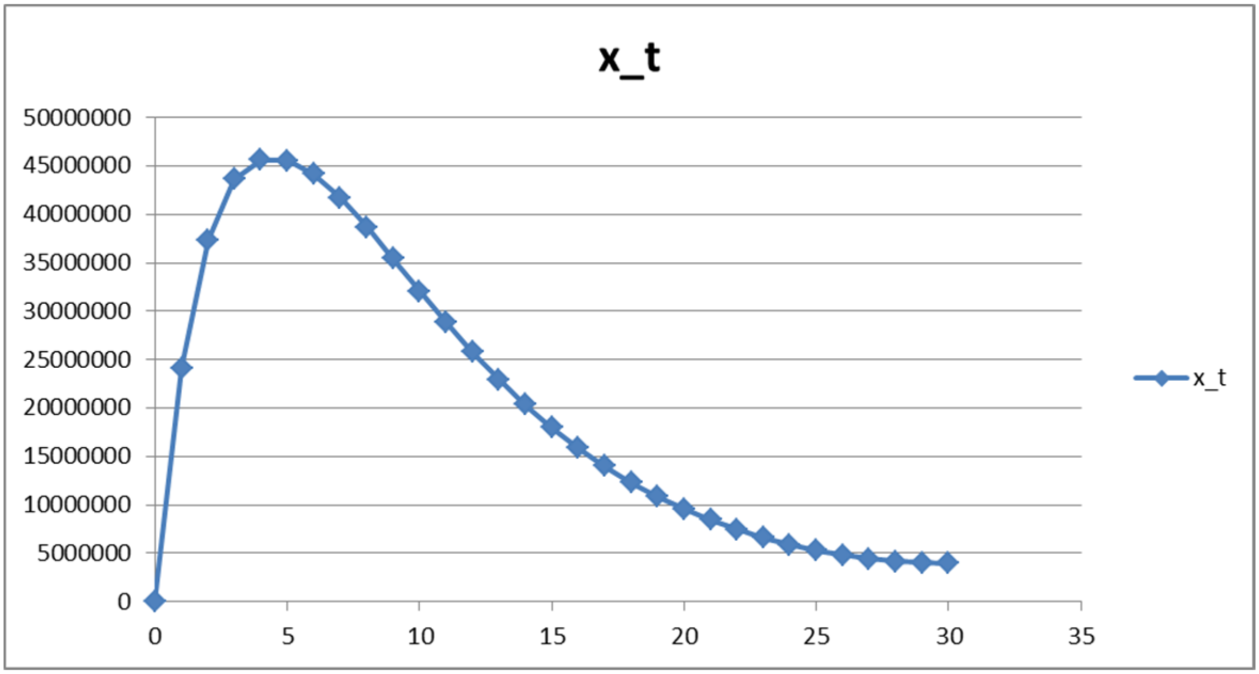
Below is R code for the first approach.

```
startTime <- proc.time()

r=0.314
n=0.142
beta=0.6
P=40
V=2e6
theta=2e8
sigma=0.02
k=1e-7
T=30
alpha=50*(1e-4)*2^(-(1:T)/5)
# we need to scale x in objective function because if x too large it makes function value insensitive to
numerical bumping
scaling = 100000

# coefficient of delta^2 in function c
coeff1 = 0.5*r*sigma/(V*P)*(theta/V)^0.25
# coefficient of |delta|^(1+beta) in function c
coeff2 = n*sigma*(abs(1/(P*V)))^beta

ObjFunc <- function(x)
{
```

```
        x=x*scaling
        dx=c(x[1], diff(x))

        c=coeff1*dx*dx+coeff2*(abs(dx))^(1+beta)
        u=sum(x*(alpha-(0.5*k*sigma*sigma)*x)-c)
        return(u)
}

# test ObjFunc
#x=sample(1:100000, T, replace=TRUE)
#print(x)
#y=ObjFunc(x)
#print(y)

x0=rep(0,T)
result = optim(x0, ObjFunc, method="BFGS", control = list(maxit = 10000, fnscale = -1))
options(digits=10)
print(result)
print(result$par*scaling)

endTime <- proc.time()
totalTime = endTime-startTime
print(totalTime)
```

Then I tried coordinate descent, which allows me to control the convergence of each dx_i. The objective function is a quadratic part plus a separable convex term of dx_i. Hence coordinate descent works. Similarly to the first approach, I need to scale parameters. However, in coordinate descent, I scale coefficients instead of the parameters. Coordinate descent gives slightly better result.

Below is the plot of optimal x_t and their values.



0	0.0
1	24160646.9
2	37387258.4
3	43628826.8
4	45593509.4
5	45541249.3
6	44144909.5
7	41725109.7
8	38711910.6
9	35427863.4
10	32093130.5
11	28849450.2
12	25782412.4
13	22938856.4
14	20339651.3
15	17988796.8
16	15879777.6
17	13999934.0
18	12333428.5
19	10863231.9
20	9572439.0
21	8445131.8
22	7466937.8
23	6625395.2

24	5910186.1
25	5313270.9
26	4828914.1
27	4453521.0
28	4185062.2
29	4021470.8
30	3955994.3

Objective function value is 538557.36291393684. Computation time is 0.89 seconds.

Below is R code for coordinate descent.

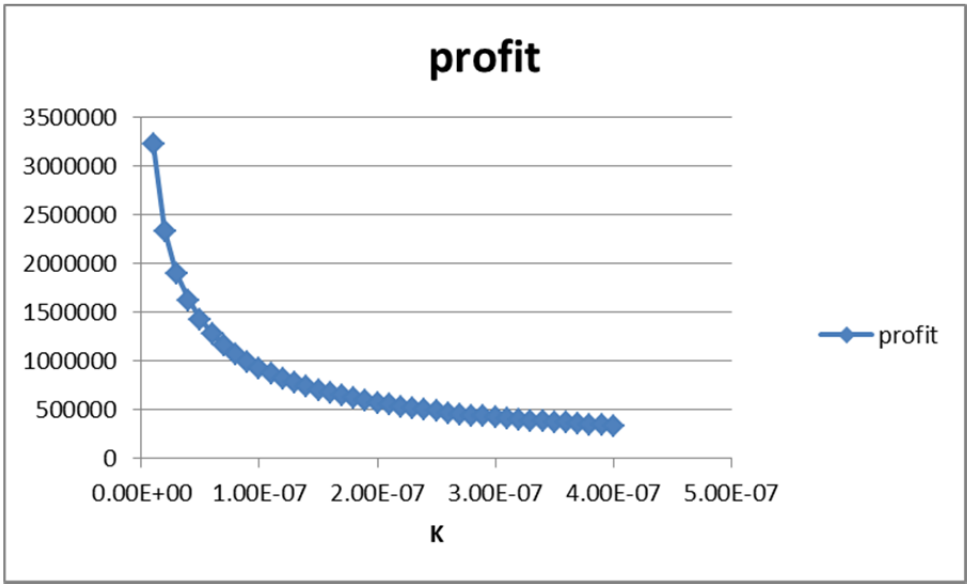
```
startTime <- proc.time()

# return value of a*x^2+b*x+c+d*|x|^p
ObjFunc1D <- function(x,a,b,c,d,p)
{
	y = (a*x+b)*x+c+d*(abs(x))^p
	return(y)
}

# return x which maximizes a*x^2+b*x+c+d*|x|^p
Max1D <- function(a,b,c,d,p)
{
	quadraticMax = -0.5*b/a
	start = min(0, quadraticMax)
	end = max(0, quadraticMax)
	# search max between the max of quadratic function and max of power function
	optimum = optimize(f=ObjFunc1D,
				lower=start,
				upper=end,
				maximum=TRUE,
				tol=1e-4,
				a=a, b=b, c=c, d=d, p=p
				)
	return(optimum$maximum)
}

#ObjFunc1D(-7.5,0.5,-1,3,1,1.6)
#Max1D(0.5, 1.5, 3, 2, 1.6)

# scaling of x_i
# instead of scaling x_i directly, we scale it on coefficients
scaling = 1e4

r=0.314
n=0.142
beta=0.6
```

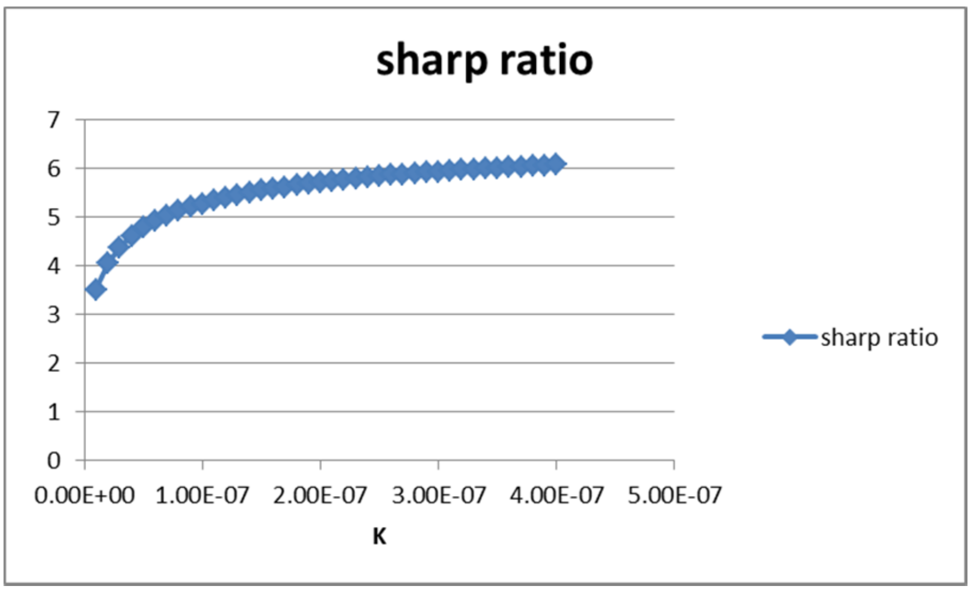
```
P=40
V=2e6
theta=2e8
sigma=0.02
k=1e-7
T=30
alpha=50*(1e-4)*2^(-(1:T)/5)*scaling

# define dx_i = x_i - x_{i-1}
# we will optimize dx_i using coordinate descent since they are separable
# initial guess of dx is a vector of 0
dx=vector("numeric", T)

# coefficient of delta^2 in function c
coeff1 = 0.5*r*sigma/(V*P)*(theta/V)^0.25*scaling^2
# coefficient of |delta|^(1+beta) in function c
coeff2 = n*sigma*(abs(1/(P*V)))^beta*scaling^(1+beta)

# coefficient of dx_i term from x_t*alpha_t in u(x)
coeff3=vector("numeric", T)
coeff3=rev(cumsum(rev(alpha)))
half_k_sigma2 = 0.5*k*sigma^2*scaling^2

# problem requires x_t within distance of one dollar to true optimal path
# we translate one dollar into 1/T dollar for dx_t and make it conservative by multiplying 0.1
tolerance=0.1/T/scaling
iter=0
while (TRUE)
{
	iter=iter+1
	old_dx=dx
	allWithinTol=TRUE
	for (i in 1:T)
	{
		# optimize dx_i while keep the others fixed
		# calculate coefficient of dx_i^2
		secondOrderCoeff = -half_k_sigma2*(T+1-i)-coeff1

		# calculate coefficient of dx_i coming from x_i^2, x_(i+1)^2, ...
		x=cumsum(dx)
		firstOrderCoeff_2 = sum(x[i:T]) - dx[i]*(T+1-i)
		firstOrderCoeff = coeff3[i]-half_k_sigma2*2*firstOrderCoeff_2

		# use c=0 because const does not affect result
		dx[i] = Max1D(secondOrderCoeff,firstOrderCoeff,0,-coeff2,1+beta)
		allWithinTol = allWithinTol && (abs(dx[i]-old_dx[i])<tolerance)
	}
```

```
	#print(c(iter, dx))
	if (allWithinTol)
	{
		break
	}
}

# change number of digits to print
options(digits=20)

dx=dx*scaling
x=cumsum(dx)
print(x)

endTime <- proc.time()
totalTime = endTime-startTime
print(totalTime)



# calculate objective function value
# coefficient of delta^2 in function c
coeff_1 = 0.5*r*sigma/(V*P)*(theta/V)^0.25
# coefficient of |delta|^(1+beta) in function c
coeff_2 = n*sigma*(abs(1/(P*V)))^beta

alpha=50*(1e-4)*2^(-(1:T)/5)

ObjFunc <- function(x)
{
	dx=c(x[1], diff(x))

	c=coeff_1*dx*dx+coeff_2*(abs(dx))^(1+beta)
	u=sum(x*(alpha-(0.5*k*sigma*sigma)*x)-c)
	return(u)
}
ObjFunc(x)
```

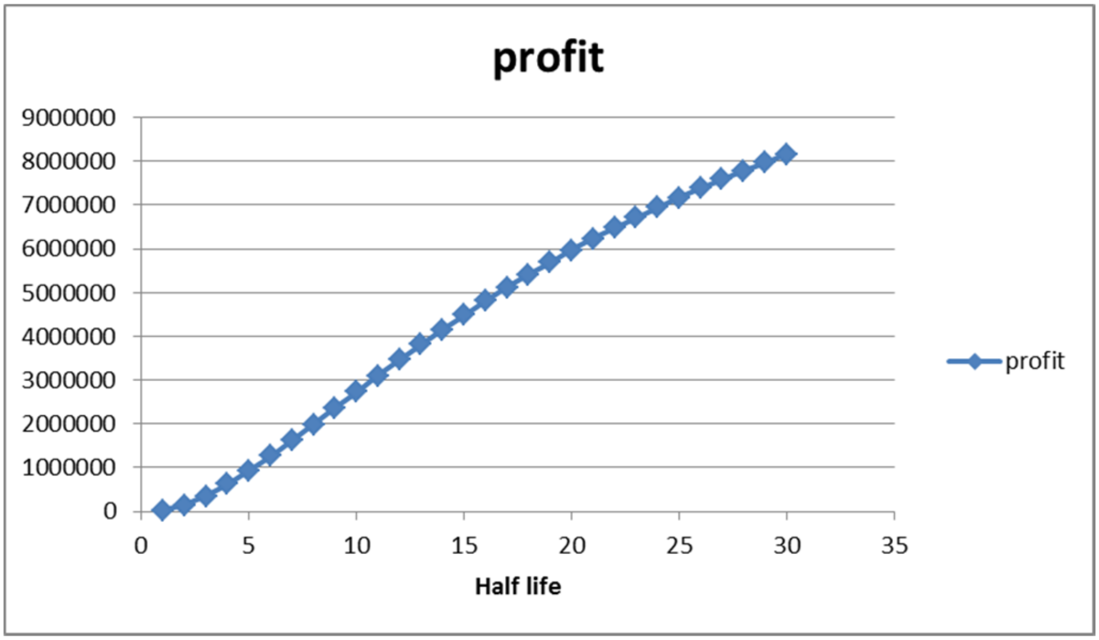
(b)

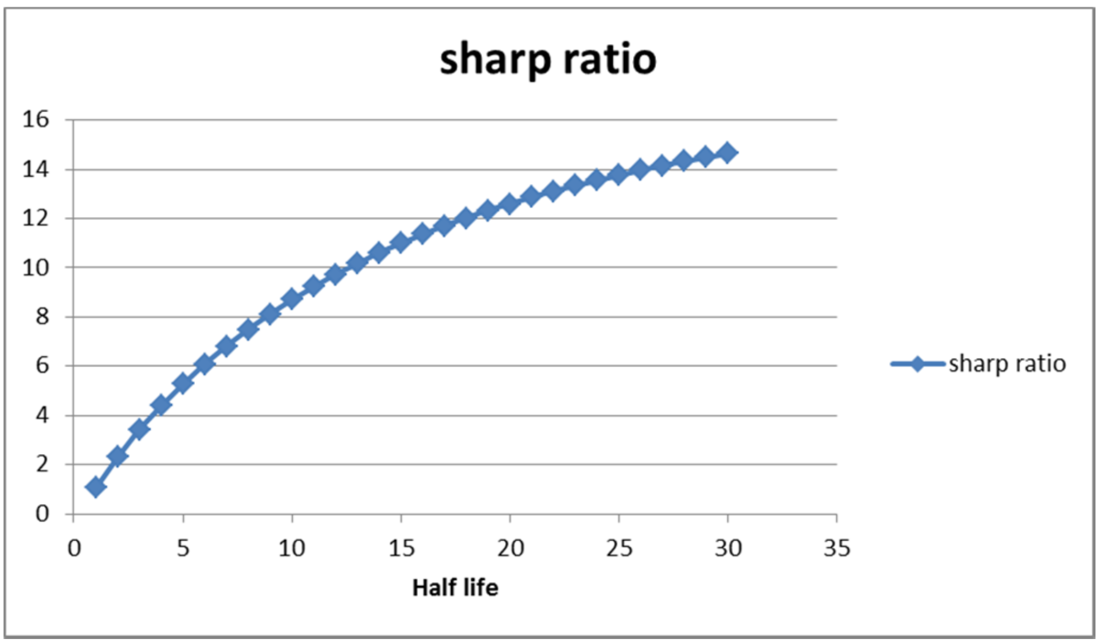
Below is plot of K sliding



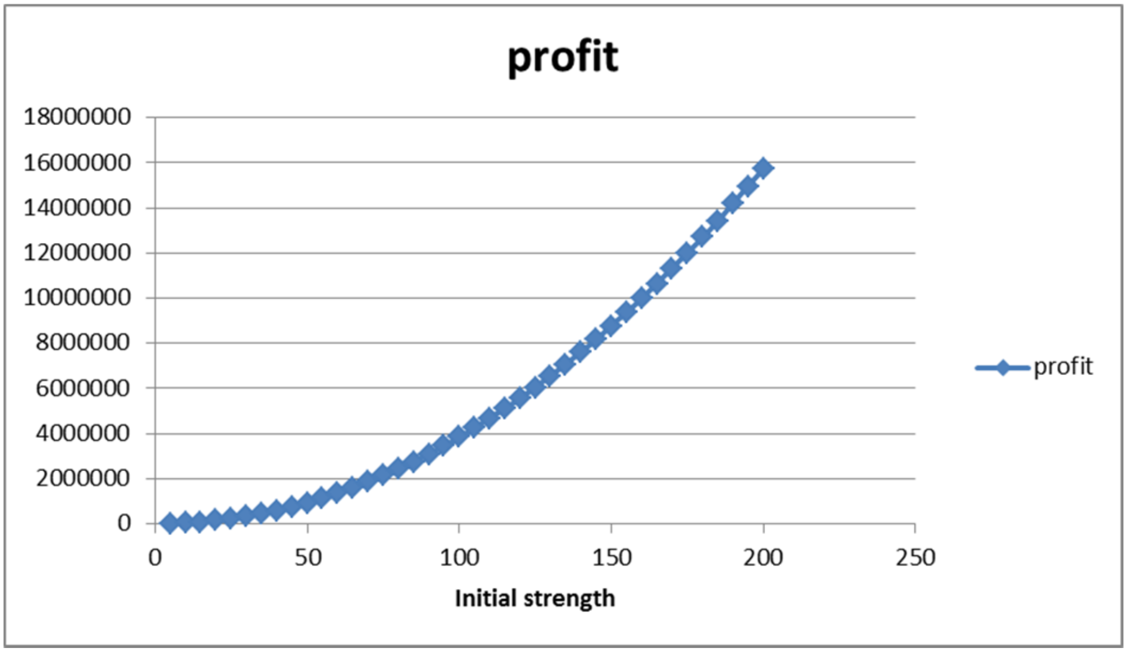


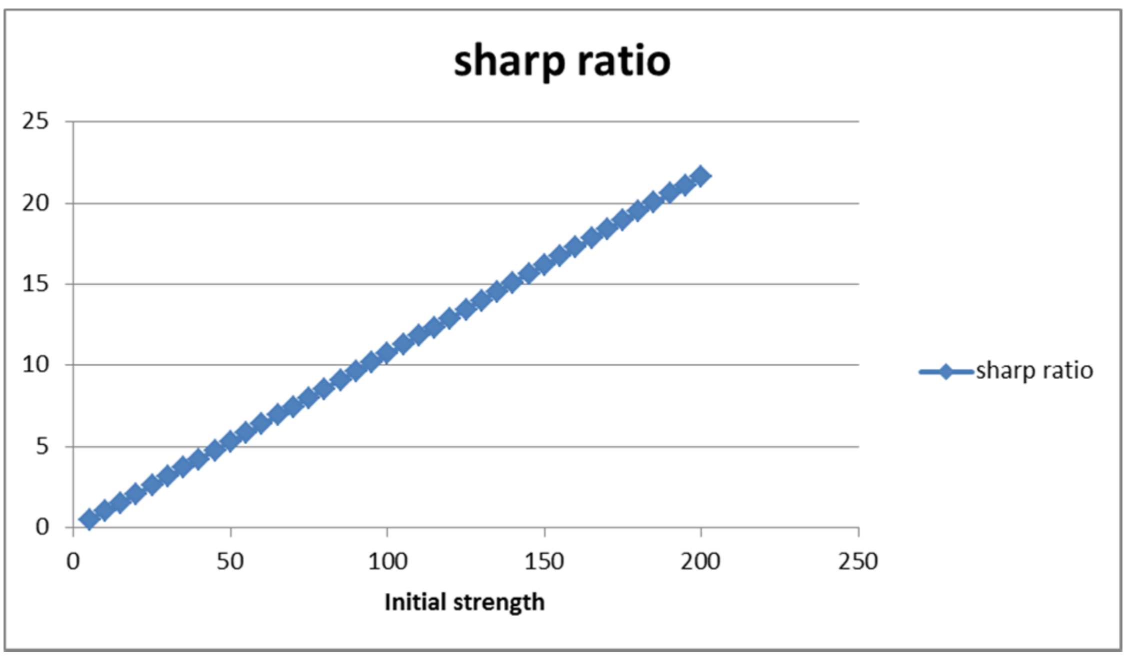
Below is half-life sliding



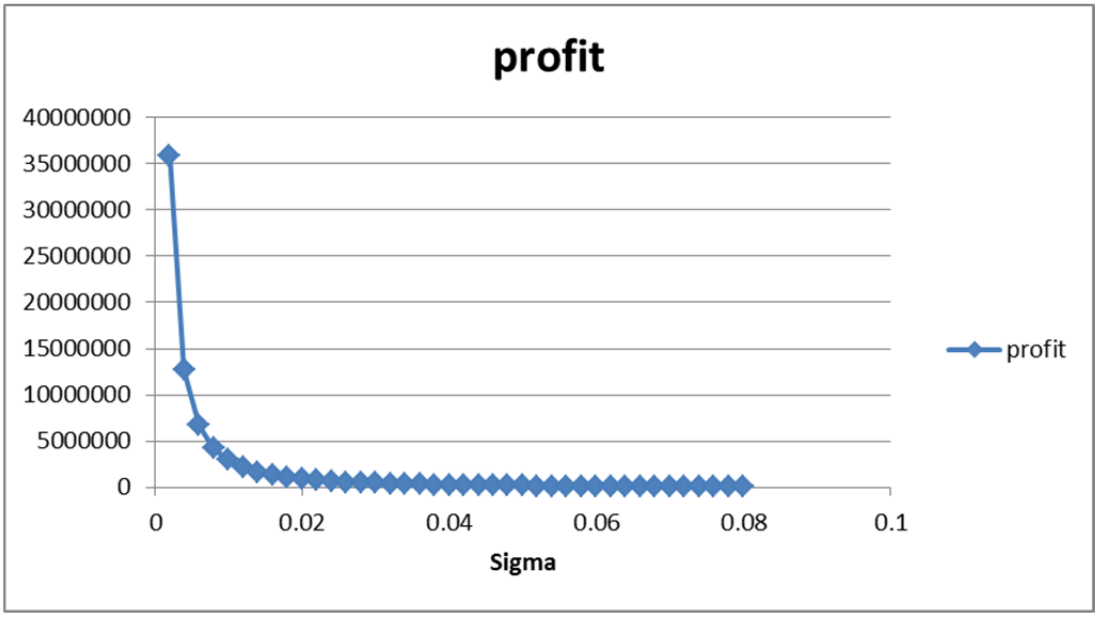


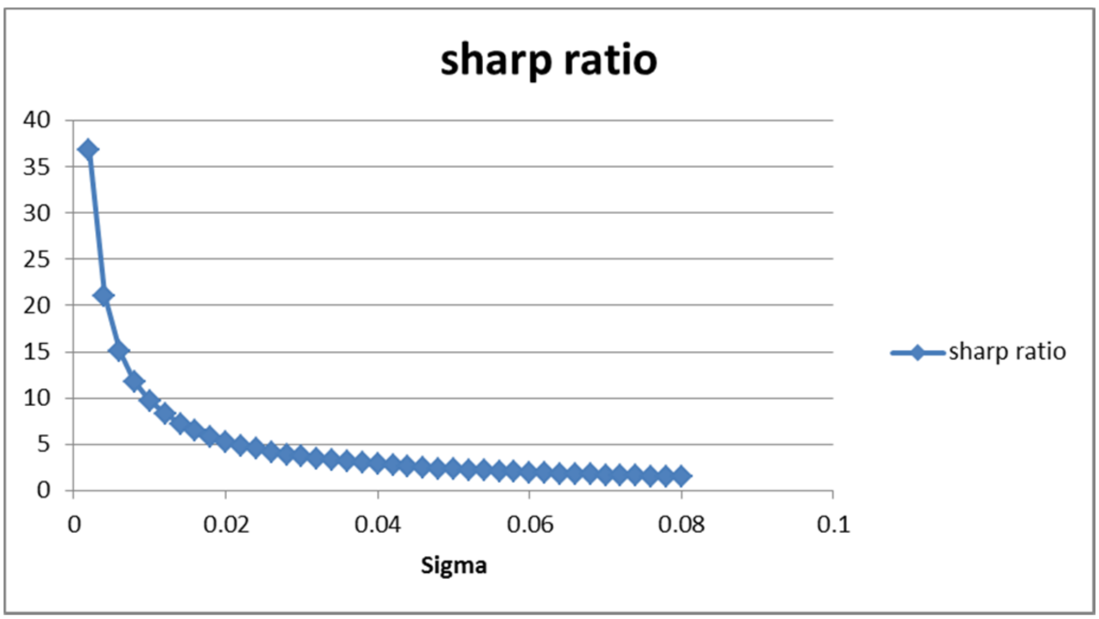
Below is initial strength sliding





Below is sigma sliding





Below is R code for sliding (the first approach is used to do optimization because it is faster and result should be very close to coordinate descent.)

```
Sliding <- function(k, halfLife, initStrength, sigma)
{
        #halfLife=5
        #initStrength=50

        r=0.314
        n=0.142
```

```
	beta=0.6
	P=40
	V=2e6
	theta=2e8
	#sigma=0.02
	#k=1e-7
	T=30
	alpha=initStrength*(1e-4)*2^(-(1:T)/halfLife)
	# we need to scale x in objective function because if x too large it makes function value
insensitive to numerical bumping
	scaling = 100000

	# coefficient of delta^2 in function c
	coeff1 = 0.5*r*sigma/(V*P)*(theta/V)^0.25
	# coefficient of |delta|^(1+beta) in function c
	coeff2 = n*sigma*(abs(1/(P*V)))^beta

	ObjFunc <- function(x)
	{
		x=x*scaling
		dx=c(x[1], diff(x))

		c=coeff1*dx*dx+coeff2*(abs(dx))^(1+beta)
		u=sum(x*(alpha-(0.5*k*sigma*sigma)*x)-c)
		return(u)
	}

	# test ObjFunc
	#x=sample(1:100000, T, replace=TRUE)
	#print(x)
	#y=ObjFunc(x)
	#print(y)

	x0=rep(0,T)
	result = optim(x0, ObjFunc, method="BFGS", control = list(maxit = 10000, fnscale = -1))
	x=result$par*scaling

	CalcExpAndSharpRatio <- function(x)
	{
		dx=c(x[1], diff(x))

		c=coeff1*dx*dx+coeff2*(abs(dx))^(1+beta)
		profit=sum(x*alpha-c)

		variance=sum((x*sigma)^2)
		sharpRatio=(252^0.5)*profit/(variance^0.5)
		return(c(profit,sharpRatio))
	}
```

```
	return (CalcExpAndSharpRatio(x))
}

k=1e-7
halfLife=5
initStrength=50
sigma=0.02

SlidingK<-function()
{
	step=0.1
	mat<-matrix(0, ncol=3,nrow=0)
	for (pecentage in -9:30)
	{
		kk=k*(1+pecentage*step)
		result = Sliding(kk, halfLife, initStrength, sigma)
		mat<-rbind(mat, c(kk, result))
	}

	options(digits=8)
	print(mat)
}

SlidingHalfLife<-function()
{
	mat<-matrix(0, ncol=3,nrow=0)
	for (hl in 1:30)
	{
		result = Sliding(k, hl, initStrength, sigma)
		mat<-rbind(mat, c(hl, result))
	}

	options(digits=8)
	print(mat)
}

SlidingInitStrength<-function()
{
	step=0.1
	mat<-matrix(0, ncol=3,nrow=0)
	for (pecentage in -9:30)
	{
		is=initStrength*(1+pecentage*step)
		result = Sliding(k, halfLife, is, sigma)
		mat<-rbind(mat, c(is, result))
	}
```

```
	options(digits=8)
	print(mat)
}

SlidingSigma<-function()
{
	step=0.1
	mat<-matrix(0, ncol=3,nrow=0)
	for (pecentage in -9:30)
	{
		ss=sigma*(1+pecentage*step)
		result = Sliding(k, halfLife, initStrength, ss)
		mat<-rbind(mat, c(ss, result))
	}

	options(digits=8)
	print(mat)
}
```

final exam Yuan Hu

2. The original problem is equivalent to the following problem with Lagrange multiplier λ

$$\min_{h,\lambda} \frac{1}{2} h^T A h + b^T h + h^T X \lambda$$

$$\Leftrightarrow \min_{h,\lambda} \frac{1}{2} (h^T, \lambda^T) \begin{pmatrix} A & X \\ X^T & 0 \end{pmatrix} \begin{pmatrix} h \\ \lambda \end{pmatrix} + b^T h$$

let $C = \cancel{X^T A^{-1} X} - A^{-1} X$

then $\begin{pmatrix} I & 0 \\ C^T & I \end{pmatrix} \begin{pmatrix} A & X \\ X^T & 0 \end{pmatrix} \begin{pmatrix} I & C \\ 0 & I \end{pmatrix} = \begin{pmatrix} A & X \\ C^T A + X^T & C^T X \end{pmatrix} \begin{pmatrix} I & C \\ 0 & I \end{pmatrix} = \begin{pmatrix} A & AC + X \\ C^T A + X^T & C^T A C + X^T C + C^T X \end{pmatrix} = \begin{pmatrix} A & 0 \\ 0 & -X^T A^{-1} X \end{pmatrix}$

$A^{-1} \succ 0$, columns of X are linear independent, so $X^T A^{-1} X \succ 0$

Hence $\begin{pmatrix} A & X \\ X^T & 0 \end{pmatrix}$ is nonsingular.

Take derivative w.r.t $\begin{pmatrix} h \\ \lambda \end{pmatrix}$ in $\frac{1}{2}(h^T, \lambda^T) \begin{pmatrix} A & X \\ X^T & 0 \end{pmatrix} \begin{pmatrix} h \\ \lambda \end{pmatrix} + b^T h$ and set it to 0

we get $\begin{pmatrix} A & X \\ X^T & 0 \end{pmatrix} \begin{pmatrix} h \\ \lambda \end{pmatrix} + \begin{pmatrix} b \\ 0 \end{pmatrix} = 0$ ---------- ①

There is unique solution $\begin{pmatrix} h^* \\ \lambda^* \end{pmatrix} = \begin{pmatrix} A & X \\ X^T & 0 \end{pmatrix}^{-1} \begin{pmatrix} -b \\ 0 \end{pmatrix}$

By construction, h^* is one feasible solution to the original problem (i.e. $h^{*T} X = 0$).

Now we need to prove h^* is the optimal one

Assume there is another feasible solution h, i.e. $h^T X = 0$.

let $p = h^* - h$ (we have $p^T X = 0$)

$$\begin{aligned}
\frac{1}{2} h^T A h + b^T h &= \frac{1}{2} (h^* - p)^T A (h^* - p) + b^T (h^* - p) \\
&= \frac{1}{2} h^{*T} A h^* + b^T h^* - p^T A h^* + \frac{1}{2} p^T A p - b^T p
\end{aligned}$$

(since $A h^* + b + X\lambda^* = 0$ by equation ①

$p^T A h^* = -p^T b - p^T X \lambda^*$)

$$\begin{aligned}
&= \frac{1}{2} h^{*T} A h^* + b^T h^* + p^T X \lambda^* + \frac{1}{2} p^T A p \\
&= \frac{1}{2} h^{*T} A h^* + b^T h^* + \frac{1}{2} p^T A p \quad (\text{since } p^T X = 0)
\end{aligned}$$

because $A \succ 0$, $\frac{1}{2} h^T A h + b^T h \geq \frac{1}{2} h^{*T} A h^* + b^T h^*$

i.e. h^* is the optimal solution

3. factor model says $r = Xf + \varepsilon$

Markowitz mean-variance objective function is

$$h' E[r] - \frac{\kappa}{2} h' \mathrm{cov}(r) h = h' E[Xf + \varepsilon] - \frac{\kappa}{2} h' X \mathrm{Cov}(f) X' h - \frac{\kappa}{2} h' \mathrm{cov}(\varepsilon) h.$$

$$= h' X E[f] - \frac{\kappa}{2} h' X \mathrm{Cov}(f) X' h - \frac{\kappa}{2} h' \mathrm{cov}(\varepsilon) h.$$

$\kappa > 0$ is risk aversion, $\mathrm{cov}(\varepsilon)$ is idiosyncratic variance

variance decomposition: total var $= h' X \mathrm{Cov}(f) X' h + h' \mathrm{cov}(\varepsilon) h$.

4. singular value decomposition says: suppose X is $m \times n$ real matrix, Then there exists a decomposition of X is the form $X = U \Sigma V^T$

where U is a $m \times m$ orthogonal matrix

Σ is a $m \times n$ diagonal matrix with non-negative real number on the diagonal

V^T is a $n \times n$ orthogonal matrix.

suppose $X = U \Sigma V^T$, Moore-Penrose pseudoinverse $X^+ = V \Sigma^+ U^T$

Σ^+ is calculated by taking reciprocal of each non-zero element on the diagonal of Σ, leaving the zeros in place, and then transposing the matrix.

In numerical computation, only elements larger than some small tolerance are taken to be non-zero, and the others are replaced by zero.

(I reference "Moore-Penrose pseudoinverse" on wikipedia, section 5.3 Singular Value Decomposition).

$X^+ = \lim_{\delta \to 0} (X'X + \delta^2 I)^{-1} X'$

on the other hand, target function of ridge regression is $\min_f \left[\| R - Xf \|^2 + \lambda \|f\|^2 \right]$ $\lambda > 0$

the solution $\hat{f} = (X'X + \lambda I)^{-1} X'R$. when $\lambda \to 0$, $(X'X + \lambda I)^{-1} X' \to X^+$

final exam Yuan Hu

5\. assume x is locally optimal. then $\exists R$, s.t. $f_0(x) = \inf\{f_0(z) \mid z \text{ feasible}, \|z-x\|^2 \le R\}$

now suppose that x is not globally optimal, i.e. there is a feasible y s.t. $f_0(y) < f_0(x)$

consider $z = \theta y + (1-\theta)x$, $\theta = \frac{R}{2\|y-x\|}$. since $\|y-x\| > R$, $0 < \theta < \frac{1}{2}$

z is feasible because x, y are feasible, and it is convex problem.

$\|z-x\| = \|\theta y + x - \theta x - x\| = \theta\|y-x\| = \frac{R}{2}$

from convexity, $f_0(z) \le \theta f_0(y) + (1-\theta)f_0(x) < \theta f_0(x) + (1-\theta)f_0(x) = f_0(x)$

This is contridiction with $f_0(z) \ge f_0(x)$.

(I reference lecture notes)

6\. (reference to Home Work 2. problem 1)

$f(x) = ax^2 + bx + c + \phi(x)$ $\quad a > 0$

$$\phi(x) = \begin{cases} lx & x < 0 \\ rx & x \ge 0 \end{cases} \qquad r \ge l$$

$\min_x f(x)$ is a convex problem with close form solution

$$x^* = \begin{cases} -\frac{b+l}{2a} & \text{if } b+l > 0 \\ -\frac{b+r}{2a} & \text{if } b+r \le 0 \\ 0 & \text{otherwise} \end{cases}$$

first define 1-D minimization function in pseudo code:

```
double Min1D(double a, double b, double c, double l, double r)
{  if b+l>0 {x = -(b+l)/(2a)}
   else if b+r<=0 {x = -(b+r)/(2a)}
   else {x = 0}
   return x
}
```

assume $\beta = (\beta_1, \beta_2 \cdots, \beta_n)^T$, $X = (X_1\ X_2 \cdots X_n)$ where $X_1 \cdots X_n$ are column vectors

$$L(\beta) = (y^T - \beta^T X^T)(y - X\beta) + \lambda \cdot \sum_i |\beta_i|$$

$$= (y^T - \beta_1 X_1^T - \beta_2 X_2^T - \cdots - \beta_n X_n^T)(y - X_1\beta_1 - X_2\beta_2 - \cdots - X_n\beta_n) + \lambda \sum_i |\beta_i|$$

when we ~~fix~~ fit β_i, $a = X_i^T \cdot X_i$

$$b = 2\left(\sum_{j \ne i} X_j^T \beta_j - y^T\right) \cdot X_i$$

$$c = \left(y^T - \sum_{j \ne i} \beta_j X_j^T\right)\left(y - \sum_{j \ne i} X_j \beta_j\right) + \lambda \sum_{j \ne i} |\beta_j|$$

$l = -\lambda$

$r = \lambda$.

pseudo code to solve LASSO regression is :

```
start with initial guess β^(0) = (β_1^(0), β_2^(0), ... β_n^(0))
k=0   f[k] = L(β)   // f[k] stores objective function value in k-th iteration
while (true)
{
   for i = 1 to n
   {
       a = x_i^T · x_i
       b = 2( Σ_{j≠i} x_j^T β_j^(k) - y^T ) x_i
       c = (y^T - Σ_{j≠i} β_j^(k) x_j^T)(y - Σ_{j≠i} x_j β_j^(k)) + λ Σ_{j≠i} |β_j^(k)|
       l = -λ
       r = λ
       β_i = Min1D(a, b, c, l, r)    // call the function defined earlier
   }
   k = k+1
   f[k] = L(β)
   if |f[k] - f[k-1]| < ε     // ε is a pre-defined threshold
   {
     exit while loop
   }
}
```

when non-differential part is separable (e.g. in LASSO), coordinate descent converges to the global optimum. (refer to "Coordinate descent" by Geoff Gordon & Ryan Tibshirani optimization 10-725/36-725 page 8)

Lets denote the optimum of LASSO problem $L(\beta)$ β^*, the optimum of $\min_{\beta} \|y - X\beta\|^2$ $\hat{\beta}$. β^*, $\hat{\beta}$ should satisfy $\|\beta^*\|_1 \le \|\hat{\beta}\|_1$ otherwise $\hat{\beta}$ will be the optimum of LASSO. Hence if $t = \|\beta^*\|_1$ constraint of the constrained least square problem is binding β^* should be the optimum of constrained least square. Otherwise, the optimum of constrained least square would be more optimal than β^* in LASSO. Hence let $t = \|\beta^*\|_1$, constrained least square is equivalent to form (0.4).