MATH 170, Optimization Theory
An Implementation of Revised Simplex Method

Matthew Pollard*

8th May 2006

Abstract

This report presents an implementation of the revised simplex method for linear programs, called *limpit*. This implementation is written in the open-source statistical language R. Limpit solves problems in the canonical form min $c^T x$ such that $Ax = b$, $x \geq 0$. An additional program, *boundedlimpit,* solves the canonical form with the added constrains $l \leq x \leq u$, and efficiently solves using Rs built in sparse matrix functions. Examples of limpit solving LPs with finite solutions, unbounded solutions, degenerate matrices infeasible matrices and redundant constraints.

*Faculty of Science, Department of Mathematics, University of California

Contents

1 Introduction **2**

2 The Implementation **3**

2.1 limpit 3

2.2 phase1 4

2.3 phase2 5

2.4 phase2revised 7

2.5 boundedlimpit 10

3 Examples **10**

3.1 Random Matrix with Finite Solution 10

3.2 Redundant Rows 12

3.3 Inconsistent Systems 13

3.4 Infeasible Systems 13

3.5 Degenerate Systems 13

3.6 Unbounded Systems 14

3.7 Bounded x Examples 15

1 Introduction

This report presents a two-phase implementation of the revised simplex method for linear programs, called *limpit*. This implementation is written in the open-source statistical language R. Limpit solves problems in the canonical form

$$\min\ c^T x,\ \text{ such that }\ Ax = b,\ \ x \geq 0$$

where A is an m by n matrix, b is a m-length real-valued vector and c is real-valued cost vector. An additional program, *boundedlimpit,* solves the related problem

$$\min\ c^T x,\ \text{ such that }\ Ax = b,\ \ l \leq x \leq u$$

This problem is efficiently solved by converting this into canonical form and using Rs built in sparse matrix functions to speed up inversion and multiplication of the basis matrix. Specifically, the transformation used is to let $\hat{x} = x - l$. The objective is thus

$$\min c^T(\hat{x} + l)\ \text{ such that }\ A\hat{x} = b - Al,\ \ \hat{x} \leq u - l,\ \ \hat{x} \geq 0$$

Now, we add the slack variable $\hat{w} \geq 0$ so $\hat{x} + \hat{w} = u - l$, and the canonical form is thus

$$\min\ \ (c, 0)(\hat{x}, \hat{w})^T$$

$$\text{s.t.}\ \begin{pmatrix} A & O \\ I & I \end{pmatrix} \begin{pmatrix} \hat{x} \\ \hat{w} \end{pmatrix} = \begin{pmatrix} b - Al \\ u - l \end{pmatrix}$$

We solve for optimal $\hat{x}$ through *limpit* and add $c^T l$ to the answer to give x. Now the transformed constraint matrix is $n+m$ by $2n$, thus doing matrix inversion and multiplication is considerably slower. However, the matrix is fairly sparse - there are lots of zeros in O and the two I sub matrices. R has built in sparse matrix treatment to considerably speed up operations. This is used, and adding these constrains seems to only increase computation time by around thirty percent.

Examples of limpit solving LPs with finite solutions, unbounded solutions, degenerate matrices infeasible matrices and redundant constraints.

2 The Implementation

The original and revised simplex algorithm is implemented in R program *limpit.*

Global variables

Argument	**Description**
a	constraint coefficient matrix, size m by n
b	equality constrain vector, length m
c	cost vector, length m
bfs	basic feasible solution with m non-zero components
B_indices	vector of a columns comprising the solution basis
V_indices	vector of a columns not in the solution basis
eps1	error tolerance term
diag	diagnostic mode, shows redundant rows and phase 1 solution
dual	if true, calculates the dual solution using equilibrium theorem
revised	if true, uses phase2revised instead of phase2

Functions

	Operation
limpit	Acts as interface, pre-conditions matrix a, checks for inconsistent and/or redundant rows. Calls **phase1** and **phase2**
phase1	Phase 1 of Simplex algorithm. Conditions so b is non-negative, adds slack variables z. Calls **phase2** to find bfs
phase2	Phase 2 of Simplex algorithm. Calculates basis inverse at every iteration.
phase2revised	Revised Phase 2 of Simplex algorithm. Updates basis inverse by pivot matrix multiplication.
boundlimpit	Simplex algorithm for bounded solutions, x such that $l < x < u$. Invokes **limpit**, **phase1** and **phase2revised**

2.1 limpit

Interface and pre-condition function which calls **phase1** and **phase2.** Puts matrix a into row-reduced echelon form, checks for inconsistent constraints. Checks for redundant rows,

and removes them accordingly.

```
limpit=function(a,b,c,diag=F,dual=F,revised=T){

m=dim(a)[1]
n=dim(a)[2]

Step 1: put into row reduced echelon form, check for inconsistent program

redundant=1:m
ech_ab=rowEchelonForm(cbind(a,b))

for (i in 1:m){
if (sum(abs(ech_ab[i,1:n]))<eps1)
redundant[i]=0
if (ech_ab[i,(n+1)]>eps1 && sum(abs(ech_ab[i,1:n]))<eps1){
print(ech_ab)
stop("constraint matrix inconsistent")}}

redundrows=a[which(redundant==0),]
a=ech_ab[,1:(n)][redundant,]
b=ech_ab[,(n+1)][redundant]
m=dim(a)[1]
n=dim(a)[2]
}

Step 2: phase 1, find BFS
xbasic=phase1(a,b,diag=diag,revised)

Step 3: phase 2, solve given BFS
if (revised==T)
result=phase2revised(a,b,c,bfs=xbasic,dual)
else
result=phase2(a,b,c,bfs=xbasic,dual)

Step 4: return result
if (diag==T && min(redundant)==0)
result$redundant=redundrows
return(result)}
```

2.2 phase1

Performs phase 1 calculations by adding dummy variable z, solves $Ax + z = b$ using **phase2** or **phase2revised** with initial basic feasible solution $x = 0$, $z = b$.

```
phase1=function(a,b,eps1=10^-5,diag=F,revised=T){
    m = dim(a)[1]
    n=dim(a)[2]
    max.int=1000

Step 1 pre-condition so b>0
bindex=which(b<0)
a[bindex,]=-a[bindex,]
b[bindex]=-b[bindex]

Step 2 add z dummy variables, make bfs
A=cbind(a,diag(m))
bfs=c(seq(0,0,length=n),b)
c2=c(seq(0,0,length=n),seq(1,1,length=m))

Step 3 check if problem is degenerate, if so, add random perturbation to find BFS
if (sum(b==0)>0){
print("Phase 1 BFS given perturbation",quote=F)
bfs[which(b==0)+n]=0.0001}

Step 4 do phase 2 calculation on program
if (revised==T)
result=phase2revised(A,b,c2,bfs=bfs)
else
result=phase2(A,b,c2,bfs=bfs)

Step 5 check for infeasibility: can we drive cost to zero?
if (result$objective>eps1){
print(result$solution)
stop("System Not Feasible")}

xbfs=result$solution[1:n]
if (diag!=F){
print("Phase 1 Basic Feasible Solution",quote=F)
print(xbfs)}

return(xbfs)
}
```

2.3 phase2

Phase 2 of the non-revised simplex method. Takes constraint matrix a, conditions b, cost c, basic feasible solution bfs. If dual=T, ouputs the optimal dual solution if exists.

```
phase2=function(a,b,c,bfs,eps1=10^-5,dual=F){

m=dim(a)[1]
n=dim(a)[2]
max.int=1000

B_indices=which(bfs!=0)   #vector of columns in A comprising the solution basis
V_indices=which(bfs==0)  #vector of columns in A not in solution basis

for (i in 1:max.int){  #initiate loop

Step 1 compute B^-1
Binv = solve(a[,B_indices])

Step 2 compute d = B^-1 * b
d = Binv %*% b

Step 3 compute c_tilde = c_V - c_B * B^-1 * V
c_tilde = seq(0,0,length=n)
c_tilde[V_indices] = c[V_indices] - c[B_indices] %*% Binv %*% a[,V_indices]

Step 4 compute j with minimum cost value of non-basic columns
j=which.min(c_tilde)
cj=c_tilde[j]

Step 5 optimality check: if cj > = 0 , then done, return solution which is optimal
if (cj > = -eps1){
    solution = seq(0,0,length=n)
    solution[B_indices] = d
    objective = sum(solution*c)
    result=list()
    result$objective=objective
    result$solution=solution

if (dual!=F){
    B_indices=which(solution>-eps1)
    Binv = solve(a[,B_indices])
    c_tilde=seq(0,0,n)
    c_tilde=c[B_indices]
    y=Bind %*% c_tilde
    result$dual_solution=t(y)[,1]}
```

```
return(result)
}

Step 7 compute w = B^-1 *a[j]
w = Binv %*% a[,j]

Step 8 compute i s.t. w[i]>0 and d[i]/w[i] is a smallest positive ratio, swap i and j from basis
zz = which (w > eps1)

if (sum(zz)==0){     # Check unboundedness: if w < 0, can drive cost to −∞
    print("Error in Phase 2: Linear Program Unbounded",quote=F)
    solution = seq(0,0,length=n)
    solution[B_indices] = d
    objective = sum(solution*c)
    result=list()
    result$objective=objective
    result$solution=solution
    result$direction=(solution-bfs)
    return(solution)}

yy = min((d[zz]/w[zz]))
ii = which(d[zz]/w[zz]==yy)
p=zz[ii]

if (length(p)>1)
    print("Problem is Degenerate",quote=F)
k = B_indices[p[1]]
B_indices[p[1]] = j
V_indices[j = = V_indices] = k
}
}
```

2.4 phase2revised

Phase 2 of the revised simplex method. Takes constraint matrix a, conditions b, cost c, basic feasible solution bfs. If dual=T, ouputs the optimal dual solution if exists.

```
phase2revised=function(a,b,c,bfs=c(),redund=T,eps1=10^-5,eps2=10^-5,dual=F,diag=F){

m=dim(a)[1]
n=dim(a)[2]
max.int=1000
```

```
B_indices=which(bfs!=0)
V_indices=which(bfs==0)
```

Step 1 *Compute B^-1 for the first and only time*

```
Binv = solve(a[,B_indices])

for (i in 1:max.int){
```

Step 2 *compute d = B^-1 * b*

```
d = Binv %*% b
```

Step 3 *compute c_tilde = c_V - c_B * B^-1 * V*

```
c_tilde = seq(0,0,length=n)
c_tilde[V_indices] = c[V_indices] - c[B_indices] %*% Binv %*% a[,V_indices]
```

Step 4 *compute j s.t. c_tilde[j] < = c_tilde[k] for all k in V_indices*

```
# cj minimum cost value (negative) of non-basic columns
# j column in A corresponding to minimum cost value

j=which.min(c_tilde)
cj=c_tilde[j]
```

Step 5 *if cj > = 0 , then we're done, return optimal solution*

```
if (cj > = -eps1){
    solution = seq(0,0,length=n)
    solution[B_indices] = d
    objective = sum(solution*c)
    result=list()
    result$objective=objective
    result$solution=solution

    if (dual!=F){
        B_indices=which(solution>-eps1)
        Binv = solve(a[,B_indices])
        c_tilde=seq(0,0,n)
        c_tilde=c[B_indices]
        y=Bind %*% c_tilde
        result$dual_solution=t(y)[,1]}
        return(result)
```

```
    }

***Step 6*** compute w = B^-1 %*% a[j]
w = Binv %*% a[,j]

***Step 7*** compute i s.t. w[i]>0 and d[i]/w[i] is a smallest positive ratio swap column j
into basis and swap column i out of basis
zz = which (w > eps1)

if (sum(zz)==0){
    print("Error in Phase 2: Linear Program Unbounded",quote=F)
    print(" ",quote=F)
    solution = seq(0,0,length=n)
    solution[B_indices] = d
    objective = sum(solution*c)
    result=list()
    result$objective=objective
    result$solution=solution
    result$direction = (bfs-result$solution)
    return(result)}
yy = min((d[zz]/w[zz]))
ii = which(d[zz]/w[zz]==yy)
p=zz[ii]

if (length(p)>1)
    print("Error in Phase 2: Problem is Degenerate",quote=F)

k = B_indices[p]
B_indices[p] = j
V_indices[j = = V_indices] = k

#Step 8 Update B^-1 through pivot matrix

Q = diag(m)
Q[,p] = (Binv%*%a[,B_indices])[,p]
rs=Q[p,p]
Q[,p]=Q[,p]/-rs
Q[p,p]=Q[p,p]/-rs
Binv = Q%*%Binv
}
}
```

2.5 boundedlimpit

Implements a modified revised simplex algorithm to solve $Ax = b$ such that $l \leq x \leq u$ where l and u are vectors.

boundlimpit=function(a,b,c,l=0,u=2,diag=F,dual=F,revised=T){

m=dim(a)[1]
n=dim(a)[2]

if (length(l)==1)
  l=seq(l,l,length=n)
if (length(u)==1)
  u=seq(u,u,length=n)

Step 1 *Transform problem to canonical form*
b2=c(b-a%*%l,u-l)
a2=cbind(a,matrix(nrow=m,ncol=n,data=0))
a2=rbind(a2,cbind(diag(n),diag(n)))
c2=c(c,seq(0,0,length=n))

Step 2 *Make a2 a sparse.matrix object for fast inversion and multiplication*
library(sparseM)
a2=sparse(a2,cholskey.decom=T)

Step 3 *Solve canonical problem using*
result=limpit(a2,b2,c2,diag,dual,revised)
result$objective=result$obj+sum(c*l)
result$solution=(result$sol+l)[1:n]
return(result)
}

3 Examples

3.1 Random Matrix with Finite Solution

Let A be a 80 by 200 matrix with standard normal variate coefficients, $c = (1, 1, ..., 1)$. Let b be given by Ax where x is a random positive vector. In R, we have

```
a=matrix(nrow=80,ncol=200,data=rnorm(80*200))
x=abs(rnorm(200))
b=a%*%x
c=seq(1,1,length=200)
```

This construction ensures the system is feasible. With this x and c, the objective is

```
sum(x*c)
[1] 1328.212
```

Now to solve the system. Since the solution has 200 elements, I'll print out the first 20. Using the revised simplex method,

```
result = limpit(a,b,c, revised=T)
result$objective
[1] 651.2778
result$solution
[1] 0.00000000 0.14156463 0.00000000 1.63122313 0.00000000 1.39889861 0.00000000
0.00000000
[9] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.35704713
0.00000000
[17] 0.00000000 0.00000000 0.97867279 0.00000000
```

The result using the non-revised solution is identical. The calculation time[1] for both is found by

```
system.time(limpit(a,b,c, revised=T) )
[1] 2.67
system.time(limpit(a,b,c,revised=F))
[1] 3.45
```

Thus the revised simplex algorithm is approximately 30% faster here. We can confirm the solution is optimal by the package lpSolver, written in C.

```
lpsolve(a,b,c)
Success!  Optimal Objective is 651.27
[1] 0.00000000 0.14156463 0.00000000 1.63122313 0.00000000 1.39889861 0.00000000
0.00000000
[9] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.35704713
0.00000000
[17] 0.00000000 0.00000000 0.97867279 0.00000000

system.time(lpsolve(a,b,c))
[1] 0.98
```

Unsurprisingly, the C compiled lpsolver is considerably faster.

[1] Calculated on a P4 1.8 ghz Dell laptop while listening to Simon and Garfunkel, Bridge Over Troubled Waters

3.2 Redundant Rows

Now lets consider a system with redundant rows. Let A be a 7 by 20 random matrix, and we'll add three redundant rows to the end to make A 10 by 20. To see redundant rows, limpit's diagnostic mode must be activated by diag=T. This additionally displays a basic feasible solution calculated by phase1.

```
a=matrix(nrow=7,ncol=20,rnorm(7*20))
x=abs(rnorm(20))
b=a%*%x
a=rbind(a,a[3,])
a=rbind(a,a[1,]*2+a[2,])
a=rbind(a,a[5,]+a[6,]+a[7,])
b[8:10]=c(b[3],b[1]*2+b[2],b[5]+b[6]+b[7])
c=seq(1,1,length=20)

limpit(a,b,c,diag=T)
[1] Phase 1 Basic Feasible Solution
[1] 0.00000000 0.00000000 0.00000000 0.05017025 0.00000000 0.00000000 0.00000000
0.00000000
[9] 0.08839515 1.36997183 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000
0.65753854
[17] 0.75414893 3.51851188 0.37499744 0.00000000

$objective
[1] 6.676953

$solution
[1] 0.1018519 0.0000000 0.0000000 0.0000000 0.0000000 0.1494267 0.0000000 0.2608727
[9] 0.0000000 1.5472984 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.6179498
[17] 0.4583644 3.5411893 0.0000000 0.0000000

$redundant
   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,] 0.4195069 -0.7462091 0.6695236 -1.688643 0.3302239 1.151725 -1.0422081 0.09802664
[2,] 1.3650131 0.9149003 -2.3164716 -1.105991 -0.9189400 -1.430844 0.9079954
0.53129315
[3,] 2.1744298 -2.1183574 0.1777595 1.920029 -0.5624909 1.405332 -1.5885212 1.99345843
[,9] [,10] [,11] [,12] [,13] [,14] [,15]
[1,] -0.2069636 -0.5645749 -1.05720862 1.4241085 0.5747375 -0.3821928 0.7816632
[2,] 1.4152472 -0.6030334 0.14146878 -0.4856640 0.4289051 -1.0033910 0.4038265
[3,] -0.7025811 2.7104462 -0.08054498 -0.7836474 -0.7097724 -2.7690020 -0.5575780
   [,16] [,17] [,18] [,19] [,20]
[1,] -1.62268336 1.0502097 0.3702018 0.01412806 -1.3791214
```

```
[2,] -1.15506225 -0.7058649 0.4991516 0.40828709 -1.4665350
[3,] 0.03998178 -0.2697745 -0.7718145 3.88968815 -0.7758856
```

The redundant printout gives three row vectors equal to the three last rows of a.

3.3 Inconsistent Systems

Consider the same matrix as before, but now we change the last b element to equal 0. Now the system is inconsistent. Limpit detects this quickly by transforming {a,b} into the row-reduced form and checks for rows of form (0,0,...,0,c), where $c! = 0$. If detected, it then prints out the row reduced form and stops with an error message.

```
b[10]=0
limpit(a,b,c)
Error in limpit(a, b, c, diag = T) : constraint matrix inconsistent
[returns rref form]
```

3.4 Infeasible Systems

Lets construct an infeasible system with A being 14 by 15. This is easy with random matrices: instead of forcing the system to be feasible by setting $b = A$, now just let b be an independent random vector. With high probability, the system is infeasible. Limpit detects infeasibility in phase1 by checking if the optimal phase1 solution has positive slack variable z components. The system is feasible if and only if $z = 0$ at the optimal solution. If infeasible, limpit prints the phase1 solution (x, z).

```
a<-rnorm(nrow=14,ncol=15,data=rnorm(15*14))
b<-rnorm(15*14)
c<-seq(1,1,length=15)

limpit(a,b,c)
Error in phase1(a, b, diag = diag) :  System Not Feasible
result$solution
[1] 0 1.12570765 0.42997935 0 0 0 0
[9] 0.76879344 0.65591181 3.40399055 0 0.29890642 2.25552841 0.14330857 0.03892769
[17] 0 2.77516246 0 2.70835271 0.19284017 0.63283715 0 0
[25] 0 0 2.16134571 0 0
```

3.5 Degenerate Systems

Consider the LP $A = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & -1 & 0 & 1 \end{pmatrix}$, $b = \begin{pmatrix} 6 \\ 0 \end{pmatrix}$, $c8 = (1, 1, 0, 0)$. This system is degenerate since b is a linear combination of fewer than $m = 2$ columns of A, namely

$b = A[,3] * 6$. Limpit checks for degeneracy in phase2, where uniqueness of the swapping basis column is checked each cycle. If it is not unique, this prints the degenerate error message, and the program continues until it finds a solution. Lets feed this LP into limpit.

```
limpit(a,b,c)
[1] Error in Phase 2:  Problem is Degenerate
$objective
[1] 0
$solution
[1] 0 0 6 0
```

This solution is feasible, has fewer than m positive elements and is optimal - the objective is 0, which is the lowest possible objective since c is non-negative.

3.6 Unbounded Systems

Let A be a 5 by 15 random matrix, b be a random vector obtained by solving $b = Ax$ where $x = \text{abs}(rnorm(15))$. Let $c = (-1, ..., -1)$. Then the system is unbounded with high likelihood. Limpit detects unboundedness in phase2, and, once detected, stops with an error message and outputs the current feasible solution and objective. We see that the objective is

```
a = matrix(nrow=5,ncol=15,data=rnorm(5*15))
x = abs(rnorm(15))
b = a%*%x
c = seq(-1,-1,length=15)

limpit(a,b,c)
[1] Error in Phase 2:  Linear Program Unbounded
$objective
[1] -29.86130
$solution
[1] 7.7720497 3.9663998 0.0000000 0.0000000 0.0000000 5.7922064 0.0000000 0.0000000
[9] 0.0000000 0.0000000 0.0000000 11.9937338 0.3369134 0.0000000 0.0000000
$direction
[1] -5.8352197 -3.5358798 0.0000000 0.0000000 0.0000000 -5.7922064 0.0000000
1.4327143
[9] 0.0000000 0.0000000 0.0000000 -9.7712530 0.3024684 0.0000000 0.0000000
```

The direction we can drive cost to $-\infty$ is given by result$direction. Here, reducing x_1, x_2, x_6, x_{12} and increasing x_8, x_{13} achieves this.

3.7 Bounded x Examples

Here, we will use boundedlimpit to solve the problem min $c^T x$ such that $Ax = b$, $l \leq x \leq u$. Consider a 20 by 40 random matrix A with random b.

```
a = matrix(nrow=20,ncol=40,data=rnorm(40*20))
x = abs(rnorm(40))
b = a%*%x
c = seq(1,1,length=40)
```

Let $l = 0$ and $u = 2$. The optimum solution for the normal unbounded problem is

```
limpit(a,b,c)
$objective
[1] 25.11814
$solution
[1] 0.50358970 0.00000000 0.00000000 0.12268044 1.45092732 0.34847158
[7] 2.31110438 1.61985488 0.00000000 0.00000000 0.00000000 0.00000000
[13] 0.00000000 0.00000000 0.04307466 4.61076766 0.00000000 2.69995147
[19] 0.06326909 2.59659522 0.16372529 0.00000000 3.12627716 0.00000000
[25] 0.43325652 0.00000000 1.01343874 0.42200484 0.00000000 0.00000000
[31] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.21155895
[37] 0.00000000 2.10689365 0.15274681 1.11795046
```

Notice that several elements are greater than 2. Now for the bounded solution,

```
boundedlimpit(a,b,c,l=0,u=2)
$objective
[1] 30.49419
$solution
[1] 0.00000000 0.00000000 0.81700725 0.99327640 1.08510929 0.00000000
[7] 2.00000000 1.02519831 0.03430606 0.00000000 0.00000000 0.22238645
[13] 1.17345517 0.00000000 0.00000000 2.00000000 0.00000000 1.71059418
[19] 1.74779300 2.00000000 0.00000000 0.00000000 2.00000000 1.90900319
[25] 1.73254543 1.94618529 0.37498547 1.17896030 0.00000000 0.00000000
[31] 0.00000000 0.44996321 0.00000000 0.50118566 0.31349343 1.47703971
[37] 0.00000000 2.00000000 0.91428517 0.88741655
```

Unsurprisingly, the objective is higher. Interestingly, the solution basis has completely changed. Now to compare calculation times.

```
system.time(limpit(a,b,c))
[1] 0.08
```

```
system.time(boundedlimpit(a,b,c))
[1] 0.11
```

Thus bounded limpit takes approximately 30% longer. Lets see if this holds for a harder. Let A be a 80 by 200 random matrix, and let $u = 2$, $l = -2$. For brevity, I'll print only the first 10 elements of the optimal x.

```
a=matrix(nrow=80,ncol=200,data=rnorm(80*200))
x=abs(rnorm(200))
b=a%*%x
c=seq(1,1,length=200)

limpit(a,b,c)
$objective
[1] 81.36619
$solution
[1] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000
0.00000000
[7] 0.00000000 0.00000000 0.00000000 0.00000000 1.33574484

boundedlimpit(a,b,c,l=-2,u=2)
$objective
[1] -133.6113
$solution
[1] -1.32761222 -2.00000000 -0.12272632 -1.46031168 -2.00000000 -0.70415344 -
0.09711611 -1.35870483 -2.00000000
[10] -2.00000000
```

The calculation times are

```
system.time(limpit(a,b,c))
[1] 3.19

system.time(boundedlimpit(a,b,c,l=-2,u=2))
[1] 4.9
```

We see that boundedlimpit is around 40 percent slower.

References

[1] Franklin J., "Methods of Mathematical Economics", *Siam Economics*, 4, 1977, 1998.