Final Project

2016-11794 한기현

2017년 12월 10일

제 1 절 세 가지 Sampling 방법

연속확률변수 X는 그에 대응되는 확률밀도함수 f가 존재한다. 확률밀도함수는 실수 범위에서 그 함수값이 음이 아니고 전체 실수 범위에서 적분하였을 때 1이 나오는 적분 가능한 함수이다. 확률변수 X가 a와 b 사이에 있을 때 확률은 a부터 b까지 f의 적분값과 일치하게 된다. 본 보고서는 계산 가능한 함수 f에 ars대하여 확률밀도함수가 정확히 f인 분포를 따르는 표본을 만드는 방법을 설명한다. 본 보고서가 다루는 sampling 방법은 rejection sampling, adaptive rejection sampling, derivative-free adaptive sampling이다.

1.1 Rejection Sampling

첫째로 rejection sampling은 확률밀도함수 f를 계산하는 방법이 쉽지 않을 때 계산하기 쉬운 확률밀도함수 g를 활용하여 간접적으로 표본을 만드는 방법이다. 이 때 g의 support는 f의 support를 포함하여야 한다. 먼저 envelope이라 불리는 함수 e를 1보다 작거나 같은 상수 alpha에 대하여 e(x)=g(x)/alpha로 정의한다. e는 임의의 점에서 f보다 함숫값이 크거나 같아 f를 포함하는 덮개와 같은 역할을 한다. 이 때 다음 단계를 거치며 확률밀도함수를 f로 가지는 분포를 따르는 확률변수 X의 표본 S을 생성한다.

(1) 확률변수 Y가 g를 따르도록 하여 크기 1인 sample을 선정한다.

(2) 확률변수 U가 표준균등분포를 따르도록 하여 크기 1인 sample을 선정한다.

(3-i) U가 f(Y)/e(Y)보다 클 때 Y를 S의 원소로 채택하지 않는다. 이를 reject라 한다.

(3-ii) U가 f(Y)/e(Y)보다 작거나 같을 때 Y를 S의 원소로 채택한다.

이 때 S의 원소로 채택한 Y가 특정 값 y보다 작거나 같을 확률은 조건부 확률의 식을 통해 확률밀도함수 f를 음의 무한대부터 y까지 적분한 값과 일치함이 쉽게 확인된다. 따라서 한 개의 sample을 뽑는 (1)부터 (3)까지의 과정은 매 번 확률분포함수를 f로 가지는 분포에서 뽑는 것이므로 매 원소의 추출이 독립이고 envelope이 달라짐에 따라 확률이 변화하지 않는다는 것을 알 수 있다.

rejection sampling에서 alpha는 1보다 작거나 같은 값이다. 확률밀도함수 g에 대하여 e는 g/alpha로 정의되므로 reject가 일어날 확률은 alpha가 작을수록 커진다. 즉, alpha가 1과 가까운 값으로 선택되어야 iteration의 진행 횟수를 감소시킬 수 있고 효율적인 sampling이 된다.

1.2 Adaptive Rejection Sampling

adaptive rejection sampling은 rejection sampling 방법에서 envelope을 함수 f에 알맞게 자체적으로 설정해주는 것이 추가되었다. envelope은 정의역 내의 원소들 k개를 모은 T_k 집합을 이용하여 만들어진다. 그 과정은

다음과 같다.

(1) T_k의 원소들에 대하여 각 원소를 x좌표로 하는 log f 함수의 그래프 위의 점들에서 그래프의 접선을 긋는다.

(2) 크기 순으로 나열했을 때 이웃한 두 원소의 접선이 만나는 점들의 집합을 Z라 한다. Z의 원소들과 접선들, 그리고 정의역 interval의 양 끝 구간으로 형성되는 그래프를 e_k로 설정한다.

(3) envelope e를 exp(e_k)로 정의한다.

위 과정에서 target 함수 f에 대해 log f의 기울기가 x가 커짐에 따라 작아져야 하므로 log f가 오목함수라는 조건이 필요하다.

adaptive rejection sampling에는 squeezed rejection sampling의 이론이 적용된다. 이것은 rejection이 이루어지지 않을 때, 즉 선택된 확률변수가 표본 S에 채택될 때를 두 종류로 나눈 것이다. squeezed rejection sampling에서 쓰이는 squeezing function s은 target function f보다 함숫값이 작아야 한다. adaptive rejection sampling에 적용되는 경우에는 다음 과정을 통해 함수 s를 생성한다. f가 log-concave 함수이므로 아래에서 생성된 s는 f보다 함숫값이 작다.

(1) T_k의 원소들에 대하여 각 원소를 x좌표로 하는 log f 함수의 그래프 위의 점들에서 x 좌표가 이웃한 점을 선분으로 이은 그래프를 s_k로 정의한다.

(2) squeezing function s를 exp(s_k)로 정의한다.

squeezed rejection sampling을 적용한 adaptive rejection sampling 알고리즘은 다음과 같다.

(1) envelope 함수 e를 정의역 구간 안에서 적분한 값의 역수를 alpha로 두고, g를 e에 상수 alpha를 곱한 함수로 정의한다. 이 때 g는 정의역 구간에서의 확률밀도함수가 된다.

(2) 확률변수 Y가 g를 따르도록 하여 크기 1인 sample을 선정한다.

(3) 확률변수 U가 표준균등분포를 따르도록 하여 크기 1인 sample을 선정한다.

(4-i) U가 f(Y)/e(Y)보다 클 때 Y를 S의 원소로 채택하지 않는다.

(4-ii) U가 s(Y)/e(Y)보다 작거나 같을 때 Y를 S의 원소로 채택한다.

(4-iii) U가 s(Y)/e(Y)보다 크고 f(Y)/e(Y)보다 작거나 같을 때 Y를 S의 원소로 채택하고, T_k에 원소 Y를 추가한다.

T_k에 원소를 추가했을 때 e 함수가 더 f와 가까워지므로 alpha가 증가하여 rejection이 일어날 확률이 감소한다. 즉 sample s를 구성하는 과정이 iteration을 거치며 더욱 효율적으로 변화한다.

1.3 Derivative-Free Adaptive Rejection Sampling

derivative-free adaptive rejection sampling의 알고리즘은 envelope 함수 e를 제외하고는 adaptive rejection sampling과 동일하다. 위에서 언급한 adaptive rejection sampling은 log f 함수의 그래프 위의 점에서 접선을 그어 envelope를 구성한다. 그러나 이 경우 함수의 각 점에서 도함수가 계산되어야 한다는 단점이 있다. 도함수가 복잡한 함수의 경우 일반적인 adaptive rejection sampling을 적용하기 까다롭다는 것이다. 이를 해결하기 위하여 고안된derivative-free adaptive rejection sampling은 새로운 envelope 함수를 생성한다. s_k의 그래프를

구성하는 선분의 연장선들로 이루어진 그래프로 e_k 함수를 구성하는 것이다. log f 그래프 위에 뾰족뾰족한 가시가 달려있는 듯한 그래프가 탄생한다. 그리고 exp(e_k(x))를 e(x)로 하는 함수 e가 envelope 함수가 된다.

제 2 절 ars 함수의 구성

2.1 test에 쓰이는 함수 예시

testf 함수는 표준편차가 2인 정규분포의 확률밀도함수이다. testdf는 이 함수의 central difference로 구한 도함수가 들어간다. 또한 ldexp2 함수는 x=0일 때 값이 1/2인 지수분포의 확률밀도함수에 log를 씌운 함수이다.

```
testf=function(x){
  return(log(exp(-(x^2)/8)/sqrt(8*pi)))
}
testdf=function(x,h=0.000001){
  (testf(x+h/2)-testf(x-h/2))/h
}
ldexp2=function(x){
  if(x<0){
    -Inf
  } else{
    log(exp(-x*2)/2)
  }
}
ldnorm=function(x){
  return(log(dnorm(x)))
}
```

2.2 입력 인수와 출력 인수

1) 입력 인수

f는 log of log-concave 함수이다. nsample은 표본의 개수이다. method는 'rs', 'ars', 'dfars'으로 위에서 소개한 세 sampling 방법 중 하나를 선택해서 넣는다. envelope는 'n', 'e'로 표준정규분포나 표준지수분포 중 하나의 분포를 갖도록 한다. derivative는 f의 도함수를 넣는 곳이다. lower.bound와 upper.bound는 정의역의 하한과 상한을 의미한다. initial은 T_k의 초기값을 의미한다. 이 인수들은 method들에 따라 사용되기도 하고 사용되지 않기도 한다.

2) 출력 인수

xsample은 ars 함수를 이용하여 선정한 표본들을 담은 벡터이다. iter는 총 iteration 횟수를 의미한다. 이 두 변수는 각각 sample, iteration_number라는 이름으로 저장된다. 추가적으로 alpha는 'rs' method에서 쓰이는

alpha를 의미하며, Tk는 'ars', 'dfars' method에서 쓰이는 T_k를 의미한다. lb와 ub는 각각 최종적으로 저장된 lower.bound, upper.bound 변수의 값을 의미한다. s는 squeezing 함수 s, e는 envelope 함수 e이다. 모든 출력변수는 리스트로 합쳐져 result로 저장되어 반환된다.

2.3 method와 관계없는 기본 setting

realf 함수를 정의하여 realf(x)=exp(f(x))가 되도록 한다. 입력 인수 f는 log-concave 성질을 가지는 표적 함수의 log 값이므로 표적함수를 나타내는 함수 realf가 필요하다. 또한 lower.bound와 upper.bound는 표준지수분포, 표준정규분포일 때마다 값이 달라지므로 realf가 두 분포 중 해당하는 것이 있는지 여부를 밝혀야 한다. 이는 lower.bound와 upper.bound에 함수 f에 맞는 값을 저장하기 위해서이다. isfexp, isfnorm는 정의역 구간에서 realf 함수가 각각 dexp, dnorm 함수와 0.00001 이상의 함수값의 차이가 있는 x 좌표가 있다면 F를 반환한다. 두 함수가 일치하는 경우로 두지 않은 이유는 함수의 계산 과정에서 오차가 발생할 수 있기 때문이다. isfexp가 참인 함수에는 lower.bound와 upper.bound에 –4와 4를, isfnorm가 참인 함수에는 lower.bound와 upper.bound에 0와 8를 대입한다. 그렇지 않은 함수들에는 다음과 같은 조작을 한다. lower.bound는 그 하한을 –1000으로 설정하고 default로 주어진 값에서 음의 무한대로부터 lower.bound까지의 함수 realf의 적분 값이 0.0001 미만이 될 때까지 1씩 감소시킨다. upper.bound는 상한을 1000으로 두고 그 반대로 upper.bound로부터 양의 무한대까지의 realf의 적분 값이 0.0001 미만이 될 때까지 1씩 증가시킨다. 이 방식대로라면 realf 함수의 lower.bound와 upper.bound 사이 정의역에서의 적분 값은 0.9998 이상이므로 전체 실수 범위에서 적분 값 1과 유사하다. 즉 realf 함수가 정의역 내에서 확률밀도함수의 성질을 잃지 않는다는 것을 알 수 있다.

2.4 rs method

rs 방식은 envelope이 'n'인 경우와 'e'인 경우에 따라 나뉜다. envelope 옵션이 'n'인 경우 함수 e는 dnorm 함수와 동일하고, envelope 옵션이 'e'인 경우 함수 e는 dexp 함수와 동일하다. 만약 그 둘 중 하나에 해당하지 않는 envelope이 들어간다면 "retry with appropriate envelope option"이라는 문장이 출력된다.

```
set.seed(51)
ars(testf,nsample=100,method='rs')
```

```
## [1] "retry with appropriate envelope option"
```

rs 방식에서 가장 먼저 하는 작업은 알파를 구하는 것이다. 초기값을 1로 둔 후에 lower.bound와 upper.bound로 구성된 구간을 등간격으로 나눈 1000개의 점 중 realf(t)가 0.01보다 큰 곳에 대해서 e(t)/realf(t)의 값들의 최솟값을 구하는 것이다. realf(t)가 0이 되면 분모에 0이 들어가 에러가 발생할 수 있기 때문에 0.05이라는 하한을 두었다. envelope 옵션이 'n'인 경우 예시이다.

```
v=seq(lower.bound, upper.bound,length=1000)
       alpha=1
       for(t in v){
      if(exp(f(t))>0.05){
       alpha=min(alpha,dnorm(t)/realf(t))
   }
```

```
  }
```

iteration 단계에서는 while 문을 이용하였다. xsample 변수의 길이가 입력 변수 nsample 보다 짧고, iteration 횟수 iter가 1000 이하일 때 다음 문장을 실행하도록 한다. envelope에 따라 표준정규분포 혹은 표준지수분포에서 random하게 크기 1인 원소 y를 추출한다. 그 원소가 lower.bound와 upper.bound 내에 들어올 때에만 초기값이 0인 변수 iter에 1을 추가한다. 또한 표준균등분포를 따르는 변수 u가 realf(y)/g(y)보다 크거나 같을 때에만 reject 시키지 않고 sample에 원소 y를 추가한다. 마찬가지로 envelope 옵션이 'n'인 경우 예시이다.

```
while(length(xsample)<nsample & iter<1000){
        y=rnorm(1)
        if(y<=upper.bound & y>=lower.bound){
          iter=iter+1
          u=runif(1)
          if(u<=exp(f(y))/(dnorm(y)/alpha)){
            xsample=c(xsample,y)
          }
        }
      }
```

이 과정을 통해 얻은 ars 함수의 출력 값은 다음과 같다. 각각 정규분포와 지수분포에서 데이터가 추출되었음을 유추할 수 있다. 만약 envelope 함수 e가 realf 함수와 차이가 크게 나는 경우에는 iteration 과정에서 rejection이 발생할 수 있다. 아래 예시에서 iteration_number는 각각 473과 115로 sample의 개수인 20을 크게 넘는다. 각각 453회, 95회 rejection이 일어났다는 것이다. 이를 감소시키기 위해서 alpha를 크게 해야 하지만 두 번째 경우 모두 alpha가 1이기 때문에 더 이상 개선이 불가능하다.

```
set.seed(51)
ars(testf, nsample=20,method='rs',envelope='n')
```

```
## $sample
##  [1] -0.95006318 -2.52534098 -1.16486763 -2.11104806 -0.83857417
##  [6]  1.01634209  1.22888168 -3.31445748 -1.84513148 -1.40111977
## [11] -0.23591326  0.08499851 -0.29872636 -1.43482936  1.65756204
## [16]  1.10888352  1.99940695  1.09390863  0.36250944 -2.88494746
##
## $iteration_number
## [1] 473
##
## $alpha
## [1] 0.03179227
##
## $lb
## [1] -8
##
```

```
## $ub
## [1] 8
```

```
set.seed(2)
ars(ldexp2,nsample=20,method='rs',envelope='e',lower.bound = 0,upper.bound = 8)
```

```
## $sample
##  [1] 0.14665267 0.52102663 0.79659722 0.33445130 0.62079453 0.25024346
##  [7] 0.78233927 2.47384373 0.12567304 0.38730817 0.25893965 0.27313300
## [13] 0.05494174 1.09073422 0.11425387 0.15716333 0.23107082 0.34332848
## [19] 0.74374659 1.22334358
##
## $iteration_number
## [1] 115
##
## $alpha
## [1] 1
##
## $lb
## [1] 0
##
## $ub
## [1] 8
```

f에 대입한 분포와 동일하게 sampling된다는 것을 밝히기 위해 히스토그램을 그린다. ldexp2의 sample 개수를 1000개로 키우면 iteration의 최댓값으로 지정된 1000회의 iteration을 모두 실행한 후 244개 크기의 sample이 만들어진다. 676회의 rejection이 일어났음을 의미한다. 이 표본을 히스토그램을 그리면 지수분포가 나온다는 것을 시각적으로 확인할 수 있다. 이는 rs method가 성공적으로 작성되었음의 증거가 된다.

```
set.seed(51)
rs1000=ars(ldexp2,nsample=1000,method='rs',envelope='e')
rs1000[2:5]
```

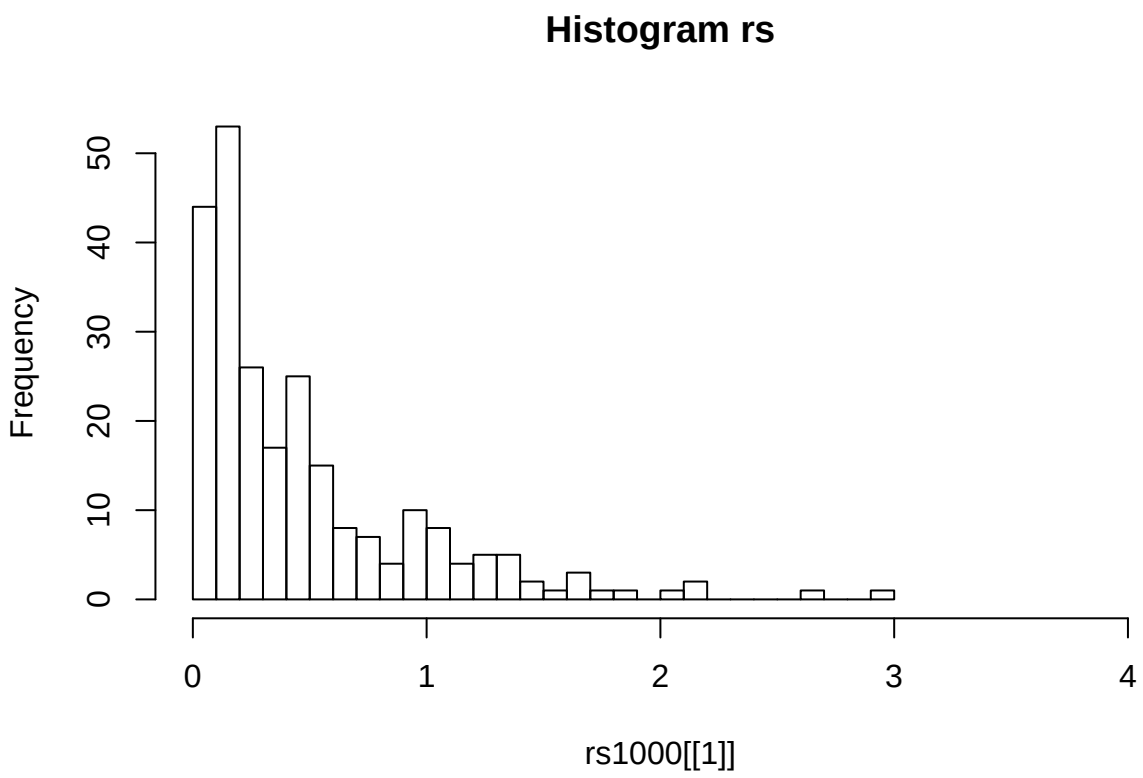
```
## $iteration_number
## [1] 1000
##
## $alpha
## [1] 1
##
## $lb
## [1] -4
##
## $ub
```

```
## [1] 4
```

```
length(rs1000[[1]])
```

```
## [1] 244
```

```
hist(rs1000[[1]],breaks=30,xlim=c(0,4),main="Histogram rs")
```



2.5 ars method

ars method를 적용하기 위해서는 함수 f의 미분계수 f를 알아야 한다. ars 함수의 derivative 옵션에 그 내용을 적게 되어있다. 만약 그 옵션을 적지 않았다면 Central difference로 구한 f의 도함수를 자동적으로 df에 저장하여 error를 방출하지 않는다.

```
df=function(x,h=0.000001){
   (f(x+h/2)-f(x-h/2))/h
}
df=derivative
```

다음 작업은 initial 옵션의 값 중 정의역 안, 즉 lower.bound와 upper.bound 사이에 있는 것만을 추출하는 것이다. 이 작업을 통해 s를 만들고 e를 만드는 과정에서 에러가 발생하지 않는다.

```
    newinitial=NULL
    for(j in 1: length(initial)){
      if(initial[j]>=lower.bound & initial[j]<=upper.bound)
      {
        newinitial=c(newinitial,initial[j])
      }
    }
    initial=newinitial
```

그 후 iteration을 진행할 때 쓰일 논리형 변수 upd를 생성한다. while문 밖에서 upd의 초기값은 T이다. while 문 내부를 반복하면서 upd는 직전 iteration에서 T_k에 원소가 추가되어 s_k, e_k의 update가 필요할 때 T가 되고 그렇지 않을 때 F가 된다. if(upd)로 시작하는 조건문에서는 T_k의 변화에 따른 s_k, e_k의 update가 진행된다. 이 과정은 모든 iteration을 거치며 함수의 update를 반복하여 시간을 소비하지 않도록 하여 효율적인 코딩이라 볼 수 있다.

T_k 집합에 따른 e_k 함수와 s_k 함수는 참고문헌에 있는 식을 통해 정의할 수 있다. T_k의 이웃한 원소의 그래프 위 점에서 그은 접선이 만나는 점을 벡터 z에 저장한 후 e_k(x)를 x가 z의 원소 사이에 어디 구간에 위치하는지 찾아 계산해낼 수 있다. 이 때 바뀐 e_k에 따라 함수 e와 g가 달라지고 확률밀도함수가 g인 분포에서 Y를 뽑는 함수 icdf가 달라진다.

icdf 함수는 g에 대한 Inverse Cumulative Distribution Function이다. g를 정의역의 최소부터 x까지 적분한 함수 F(x)라 하면 표준균등분포의 원소 U에 대응되는 F의 inverse (U)를 찾는 함수이다. icdm을 구하는 방법은 다음과 같다. 이를 r 코드로 만든 것은 아래와 같다.

(1) x_1, ... , x_m은 크기 순서대로 나열된 f의 support 상의 원소들이다. 이 때 u_i=F(x_i)가 되는 u_1, ... , u_m을 잡으면 이들도 크기 순서대로 나열된다.

(2) 0과 1 사이에 있는 U에 대하여 U가 u_i와 u_(i+1) 사이에 존재한다면 F의 함수 그래프에 하여 (x_i, u_i)와 (x_(i+1), u_(i+1))를 잇는 선분의 내분점의 x좌표로 변수 X를 반환한다.

```
icdf = function(u){
          integF=function(x){...
            }

          Fseq=seq(lower.bound,upper.bound,length=51)
          uniseq=NULL
          for(j in 1:51){
            uniseq=c(uniseq,integF(Fseq[j]))
          }
          for(j in 1:50){
            if(uniseq[j]<=u &uniseq[j+1]>=u){
              return ((uniseq[j+1]-u)/(uniseq[j+1]-uniseq[j])*Fseq[j]
+(u-uniseq[j])/(uniseq[j+1]-uniseq[j])*Fseq[j+1])
            }
```

```
    }
  }
    }
```

icdf 함수를 실행하기 위해서는 integF 함수가 필요하다. integF 함수는 g의 lower.bound부터 x까지의 적분 값을 출력하는 함수이다. 실제로는 e의 정의역 전체 적분 값에서 x까지의 적분 값을 나눈 함수로 구하였다. 이 때 integrate 함수를 이용하면 코드의 길이를 단축시킬 수 있지만 10번의 iteration을 도는 데 3분 이상의 시간이 걸릴 정도로 긴 시간이 소요된다. g는 piecewise하게 적분함수이므로 integF는 지수함수의 적분을 이용하여 계산하면 짧은 시간으로 sample을 뽑을 수 있다. 코드의 길이 때문에 integF를 생략하였다.

ars 함수의 method에 'ars'를 지정한 예시는 다음과 같다. testf 함수에 대한 크기 20짜리 표본이다. sample을 살펴보면 이 표본이 정규분포를 따른다는 사실을 확인할 수 있다. 표본 20개를 뽑는 데에 22번의 iteration이 이루어진다. 즉 2번만 reject된다는 것이다. 이는 앞서 'rs' method에서 확인한 reject보다 훨씬 감소한 것이다. 정규분포나 지수분포로 임의로 만든 envelope 함수보다 자체적으로 생성한 envelope 함수의 효율이 더 좋다는 것을 알 수 있다. 또한 최종적으로 Tk의 원소는 6개이다. update가 일어난 경우가 initial 이후로 3번 더 있었다는 의미이다. 따라서 upd가 TRUE가 되어 e_k 함수와 s_k 함수를 재 정의하는 작업이 22번 중 3번 존재하기 때문에 upd가 시간을 단축하는 데에 큰 공헌을 했음을 밝힐 수 있다. upd가 없었다면 매 iteration마다 s_k와 e_k를 재정의 했을 것이기 때문이다.

```
set.seed(51)
result=ars(testf,nsample=20,method='ars',derivative=testdf)
result[1:3]
```

```
## $sample
##  [1]  1.3195075  1.1321933  3.6813971 -0.3773698 -2.0457432 -4.2868421
##  [7] -2.3043794  0.6815601 -1.9245356  3.4662456  3.5046726 -2.6455077
## [13] -2.1957662  0.9366515  2.3658628 -0.4577834 -1.7920588  0.4731762
## [19] -1.6158812 -2.8425692
##
## $iteration_number
## [1] 22
##
## $Tk
## [1] -4.0000000  1.0000000  4.0000000 -0.3773698 -4.2868421 -2.6455077
```

testf의 sample 개수를 1000개로 키우면 1000회의 iteration을 모두 실행한 후 817개 크기의 sample이 만들어진다. 183회의 rejection이 일어났고, Tk에 원소가 추가되어 s와 e 함수의 update가 일어난 것은 Tk의 길이인 31에서 초기 값 3개를 제외한 28번이다. iteration 이 표본을 히스토그램을 그리면 정규분포와 유사한 분포가 나온다. 따라서 ars로 뽑은 표본이 표적함수의 분포에서 뽑은 것으로 볼 수 있다.

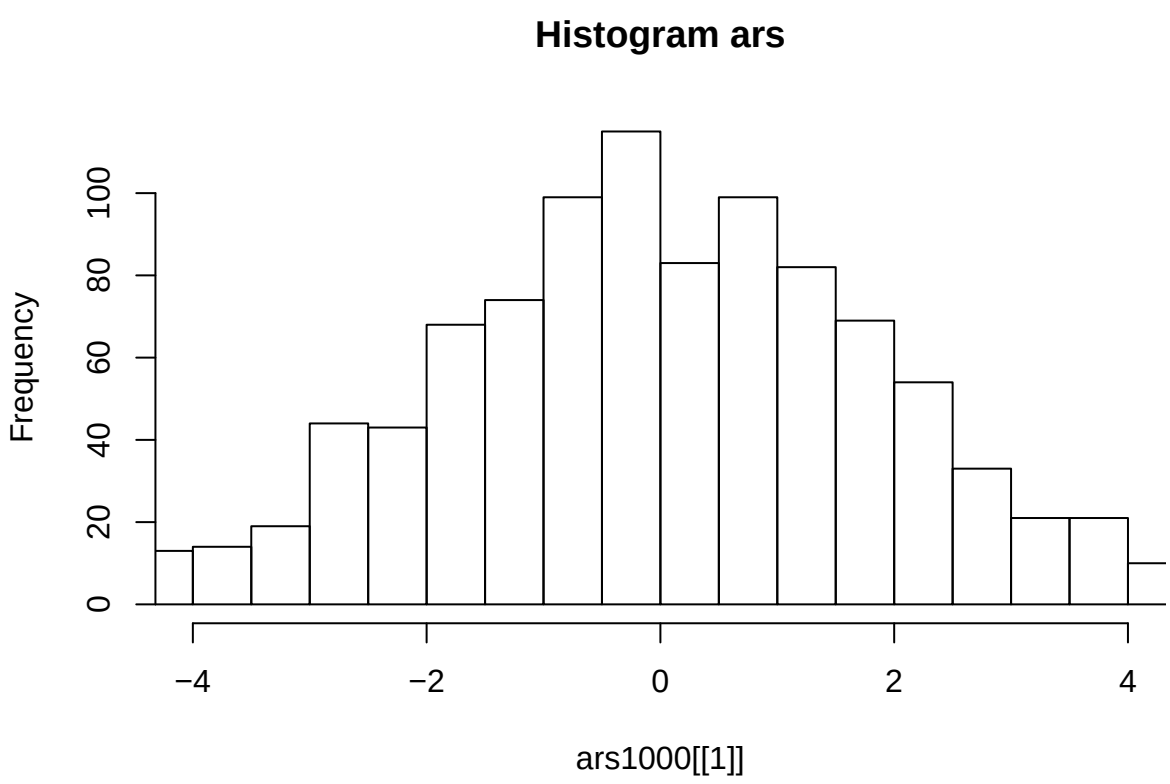
```
set.seed(51)
ars1000=ars(testf,nsample=1000,method='ars',derivative=testdf)
ars1000[2:3]
```

```
## $iteration_number
## [1] 1000
##
## $Tk
##  [1] -4.00000000  1.00000000  4.00000000 -0.37736976 -4.28684213
##  [6] -2.64550773 -1.00556728  1.59868546  4.02985588  4.66227857
## [11] -1.77447641 -5.57899337  2.32634444  0.74335449  0.28153790
## [16] -6.03404291  3.05054928 -0.19438547  2.14694758 -1.17336677
## [21] -0.85833721 -2.82248955  5.62053749 -2.21922622  5.74645304
## [26] -6.10131902 -0.03585256 -3.50637860  1.69764780  2.54371411
## [31]  6.37298371
```

```
length(ars1000[[1]])
```

```
## [1] 990
```

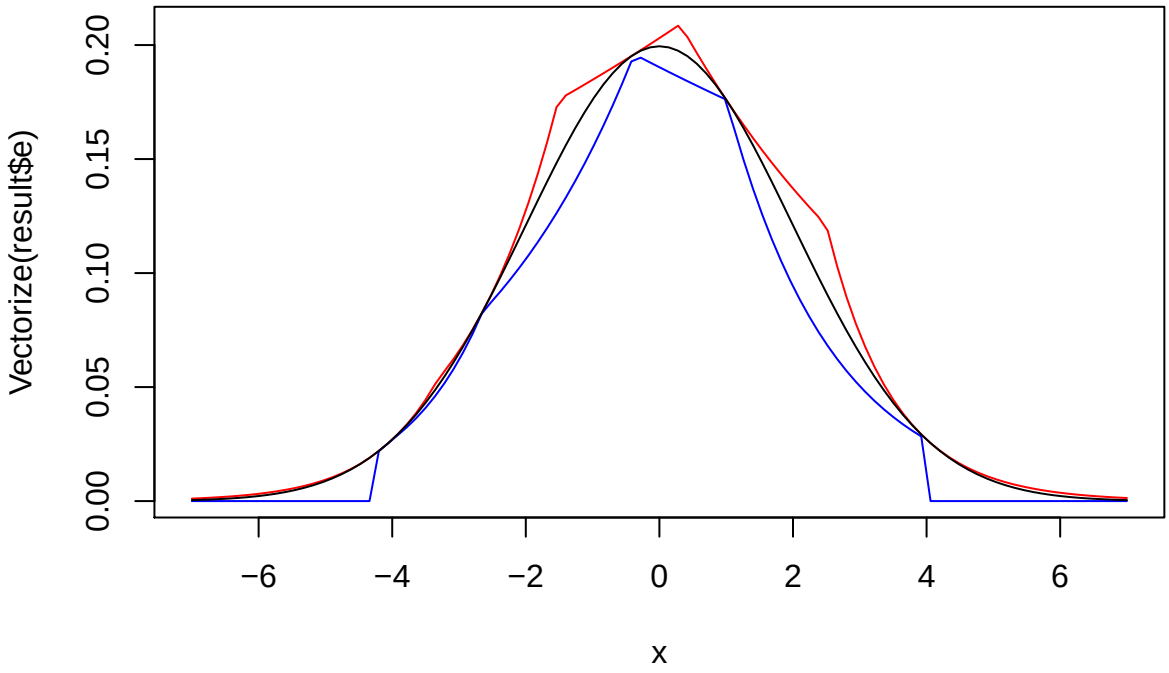
```
hist(ars1000[[1]],breaks=30,xlim=c(-4,4),main="Histogram ars")
```



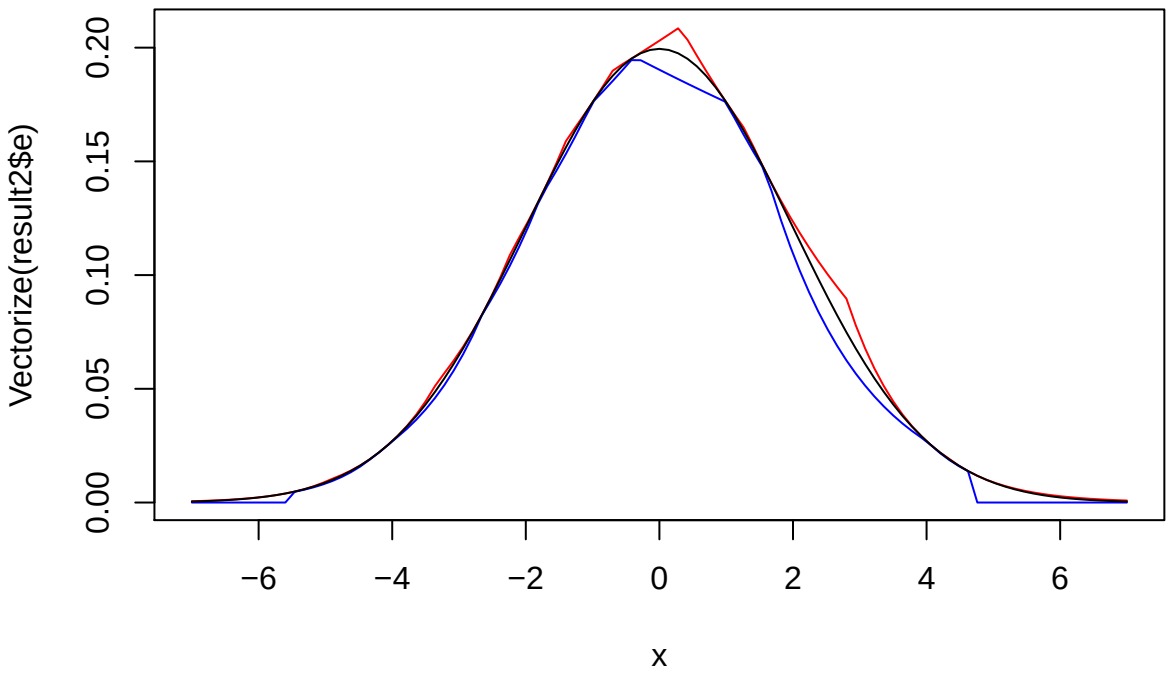
다음은 이 결과를 그래프로 나타낸 것이다. 빨간색은 envelope 함수 e, 파란색은 sqeezing 함수 s, 검정색은 target 함수 realf의 그래프이다. nsample, 즉 표본의 원소의 개수를 늘릴 때 envelope 함수와 squeezing 함수가 target 함수로 다가간다는 것을 시각적으로 확인할 수 있다. 실제로 원소 100개를 뽑은 result2에서 Tk 집합의 원소의 개수는 12개이다. 앞서 원소를 20개 뽑은 result에서 Tk 집합의 원소의 개수가 6개였던 것에 비해 증가했기 때

문에 두 함수가 realf와 가까워진 것이다. 또한 Tk의 원소의 개수는 nsample이 증가하는 것에 비해 느린 속도로 증가한다는 것도 추가적으로 알 수 있다.

```
plot(Vectorize(result$e),xlim=c(-7,7),col='red')
plot(Vectorize(result$s),xlim=c(-7,7),add=TRUE, col='blue')
plot(Vectorize(function(x) exp(testf(x))),xlim=c(-7,7),add=TRUE,col='black')
```




```
set.seed(51)
result2=ars(testf,nsample=100,method='ars',derivative=testdf)
plot(Vectorize(result2$e),xlim=c(-7,7),col='red')
plot(Vectorize(result2$s),xlim=c(-7,7),add=TRUE,col='blue')
plot(Vectorize(function(x) exp(testf(x))),xlim=c(-7,7),add=TRUE,col='black')
```




```
result2$Tk
```

```
##  [1] -4.0000000  1.0000000  4.0000000 -0.3773698 -4.2868421 -2.6455077
##  [7] -1.0055673  1.5986855  4.0298559  4.6622786 -1.7744764 -5.5789934
```

2.6 dfars method

method 옵션에 'dfars'를 넣은 경우는 본질적으로 'ars'와 동일하다. 달라지는 유일한 것은 envelope 함수이다. ars일 때 envelope 함수에 쓰이는 벡터 z는 T_k의 이웃한 두 점에서 접선의 교점의 x좌표였다. dfars에서 z는 T_k의 이웃한 원소 사이에 존재하는 점의 x 좌표를 의미한다. 수식으로 설명하자면 log f 그래프 위의 이웃한 점을 잇는 선분의 방정식 L_i(x)에 대하여 L_(j-1)(x)=L_(j+1)(x)인 T_k의 j, j+1번째 원소 사이의 점이다. estar의 반환 값은 벡터이다. 벡터의 첫 번째 원소에는 함수 값이 들어가고, 두 번째 원소에는 그 원소가 속한 선분의 번호가 들어간다. 선분의 번호는 lower.bound를 지나는 선분부터 1로 시작한다. 홀수 번호는 왼쪽에 T_k의 원소가 들어가는 선분이고, 짝수 번호는 오른쪽에 T_k의 원소가 들어가는 선분이다.

```
estar=function(x){
  for(j in 0:(n-2)){
    if(x<=sortv[j+1] & x>=z[j+1]){
     return(c(((x-sortv[j+1])*f(sortv[j+2])+f(sortv[j+1])*(sortv[j+2]-x))/(sortv[j+2]-sortv[j+1]),2*j+1
    }
```

```
    if(x>=sortv[j+2] & x<=z[j+3]){
      return(c(((x-sortv[j+1])*f(sortv[j+2])+f(sortv[j+1])*(sortv[j+2]-x))/(sortv[j+2]-sortv[j+1]),2*j+
    }
  }
  return(c(-Inf,0))
}
```

이 선분 번호를 이용하여 integF 함수에서 적분을 할 때 오류 없이 효율적으로 적분이 가능해진다. 그러나 이 선분들 중에는 x축과 수직인 선, 즉 기울기가 무한대인 선분들도 존재하는데, 이 때문에 T_k의 원소들의 e 함숫값이 불연속인 점들이 발견된다. 이 때 좌극한 우극한을 표시하기 위해 각각 아주 작은 h에 대하여 e(x-h)와 e(x+h)로 둔다. 이로 인해 결과적으로 약간의 오차가 발생한다. 따라서 'ars'에서 integF 함수의 내용과 동일하게 integFraw 함수를 만들고 이를 보정하여 integF를 최종적으로 생성한다.

```
for(j in 0:((k-2)/2)){
  if(f(sortv[j+2])!=f(sortv[j+1]) & sortv[j+1]!=z[j+1]){
    intx=intx+(e(sortv[j+1]-h)-e(z[j+1]+h))/(f(sortv[j+2])-f(sortv[j+1]))*(sortv[j+2]-sortv[j+1])
  } else{
    intx=intx+e(z[j+1]+h)*(sortv[j+1]-z[j+1])
  }
}
```

dfars method의 결과는 ars일 때와 유사하게 얻어진다. 그러나 dfars는 initial 원소가 적은 경우에 iteration을 거치며 sample이 얻어지지 않는 문제가 발생할 수 있다. 그 예시는 아래와 같다. iteration_number가 지정된 최댓값인 1000이 나오지만 sample은 NULL이다. 즉 모든 경우에 rejection이 벌어졌다는 것이다. 어떤 문제가 있는지 e의 그래프를 살펴보자. e의 함숫값은 –4와 -3부근에서 dnorm에 비하여 매우 큰 값을 가진다. s_k의 선분들의 외삽으로 그래프가 이루어지기 때문이다. 정리하자면 특수한 f에 대해 initial의 원소의 개수가 3개 정도로 적은 경우 이러한 에러가 발생할 수 있다는 것이다.

```
set.seed(51)
pung=ars(ldnorm,nsample=30,method='dfars')
pung$sample
```

```
## NULL
```

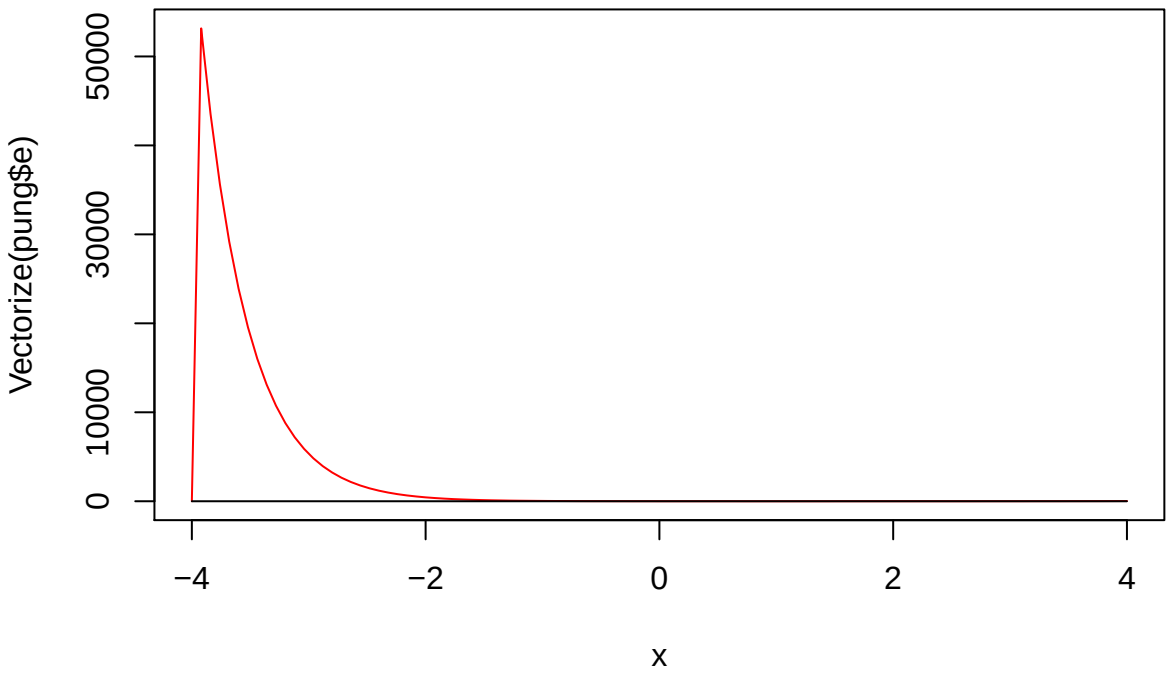
```
pung$iteration_number
```

```
## [1] 1000
```

```
pung$Tk
```

```
## [1] -4  1  4
```

```
plot(Vectorize(pung$e),xlim=c(-4,4),col='red')
plot(Vectorize(function(x) dnorm(x)),xlim=c(-4,4),add=TRUE,col='black')
```



dfars에서 testf의 sample 개수를 100개로 키우면 110회의 iteration을 실행한 후 100개 크기의 sample이 만들어진다. 이 표본은 0부근에서 많이 뽑히는 분포지만 완벽히 정규분포라고 보기 어렵다. 이는 e 함수의그래프에서 근거를 찾을 수 있다. iteration 과정 중에 0 부근에서 e 함수가 표적 함수와 간격이 크므로 rejection 비율이 높기 때문에 발생한다.

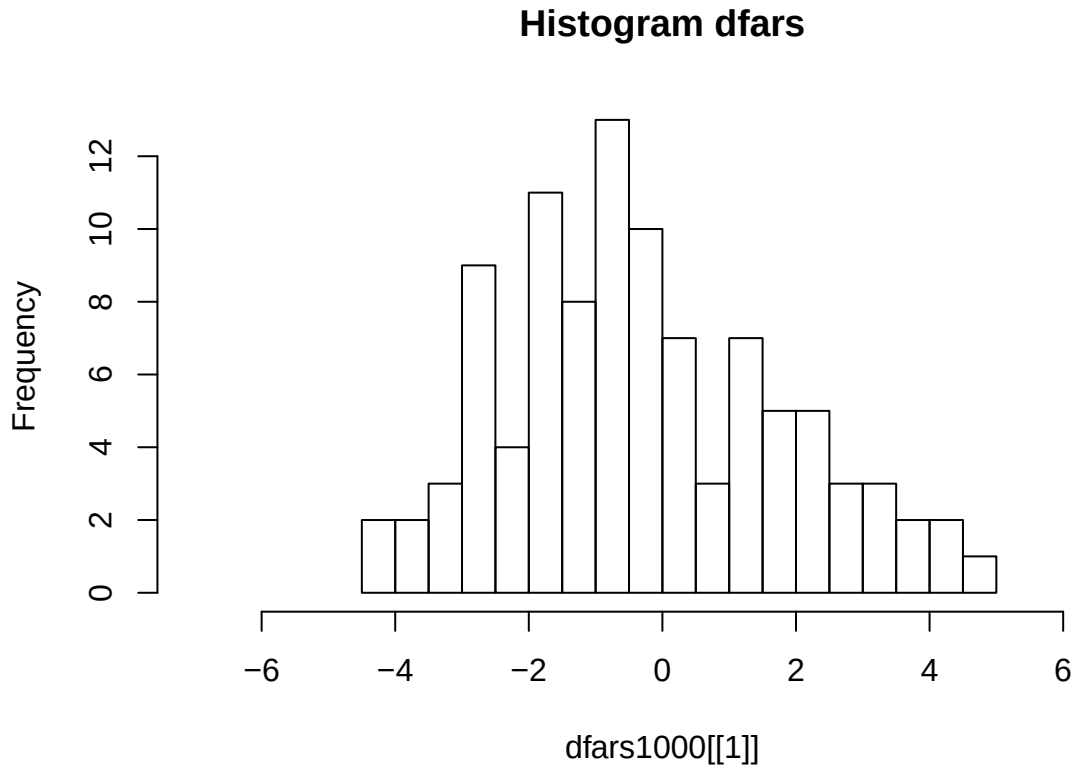
```
set.seed(51)
dfars1000=ars(testf,nsample=100,method='dfars')
dfars1000[2:3]
```

```
## $iteration_number
## [1] 110
##
## $Tk
##  [1] -4.0000000  1.0000000  4.0000000 -0.8124673  4.0356104 -4.0433234
##  [7] -2.5464301  1.5773593 -1.6499427 -3.3703410  4.6333225 -4.1127897
## [13] -2.9156886
```

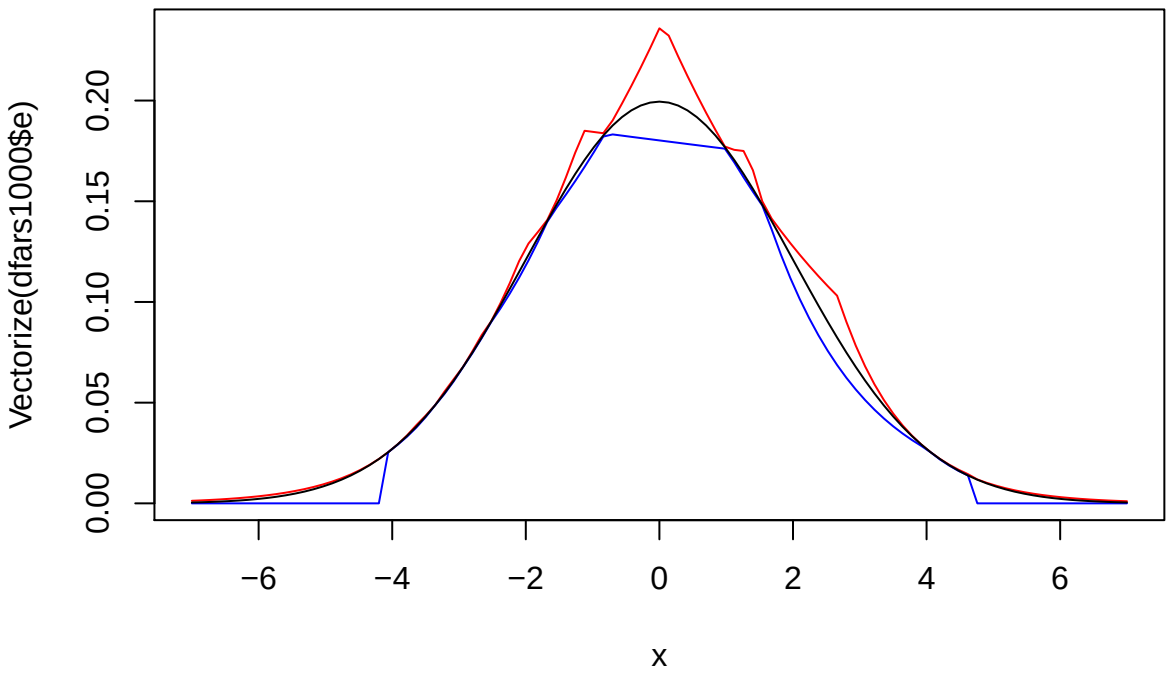
```
length(dfars1000[[1]])
```

```
## [1] 100
```

```r
hist(dfars1000[[1]],breaks=30,xlim=c(-7,7),main="Histogram dfars")
```

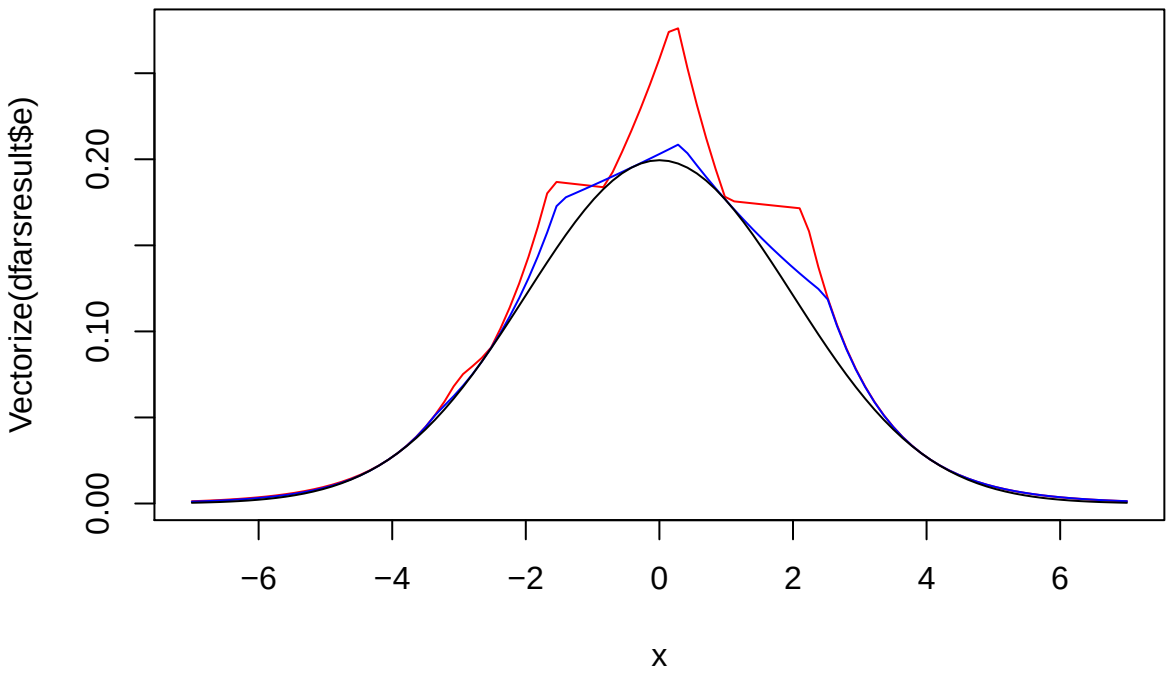



```r
plot(Vectorize(dfars1000$e),xlim=c(-7,7),col='red')
plot(Vectorize(dfars1000$s),xlim=c(-7,7),add=TRUE,col='blue')
plot(Vectorize(function(x) exp(testf(x))),xlim=c(-7,7),add=TRUE,col='black')
```



dfars 방법의 장점은 함수 f의 derivative를 계산하지 않아도 된다는 것이다. 그러나 dfars 방법이 가지는 단점도 많다. 첫째로 e의 함숫값의 크기가 ars일 때 e에 비해 크다는 것이다. 즉 alpha가 작고 rejection의 확률이 증가한다는 것이다. 예를 들어 다음 그래프에서 빨간 색으로 표시된 e 함수는 위에서 살펴보았던 result의 파란색으로 표시된 e 함수보다 대체로 위에 있음을 확인할 수 있다.

```
set.seed(51)
dfarsresult=ars(testf,nsample=20,method='dfars')
plot(Vectorize(dfarsresult$e),xlim=c(-7,7),col='red')
plot(Vectorize(result$e),xlim=c(-7,7),add=TRUE,col='blue')
plot(Vectorize(function(x) exp(testf(x))),xlim=c(-7,7),add=TRUE,col='black')
```



두 번째 단점은 dfars가 시간이 오래 걸린다는 것이다. 다음 코드를 통해 시간을 비교할 수 있다. rs 코드에서는 0.03초만에 sampling이 된 반면 ars 코드에서는 0.3초가 걸렸고, dfars 코드에서는 1.3초가 걸린다. 이 시간 데이터는 매 실행마다 약간의 오차가 발생하여 Rmarkdown으로 작성되어 실행될 때의 수치와 일치하지 않을 수 있다. rs 코드에서는 473번의 iteration을 거치지만 시간이 짧게 걸리고 나머지에서는 22번의 iteration에서도 시간이 길게 걸리는 이유는 적분과 관련된 함수 실행이 시간을 길게 소모하기 때문이다. ars 코드와 dfars 코드에서 유일한 다른 점은 z 벡터를 만드는 방식과 적분 방식이다. 적분 과정에서 dfars 코드는 arsbo다 두 배 이상의 사칙연산과 두 배 이상의 함수 대입을 하기 때문에 dfars 코드의 실행 시간이 길다.

```
beginTime<-Sys.time()
set.seed(51)
exrs=ars(testf, nsample=20,method='rs',envelope='n')
Sys.time()-beginTime
```

```
## Time difference of 0.020015 secs
```

```
beginTime<-Sys.time()
set.seed(51)
exars=ars(testf,nsample=20,method='ars',derivative=testdf)
Sys.time()-beginTime
```

```
## Time difference of 0.302248 secs
```

```r
beginTime<-Sys.time()
set.seed(51)
exdfars=ars(testf,nsample=20,method='dfars')
Sys.time()-beginTime
```

```
## Time difference of 1.201838 secs
```

제 3 절 결론

본 보고서는 log-concave 함수 f에 대하여 확률밀도함수가 정확히 f인 분포를 따르는 표본을 만드는 세 가지 방법을 이론적으로 분석하고, 실제로 R 코드를 작성하여 특징을 연구하였다. 표준정규분포와 표준지수분포를 이용한 rs는 빠르지만 많은 iteration 횟수를 요한다. ars는 rs보다 더 적은 iteration 횟수로, dfars보다 빠른 시간 내에 sampling에 성공하지만, 도함수를 계산하기 까다로운 경우에 유용하게 쓰일 수 없다. 마지막으로 dfars는 derivative-free란 이름답게 도함수를 필요로 하지 않지만, envelope function의 함숫값이 ars에 비해 커 rejection 확률이 높고 ars보다 매우 긴 시간이 소요한다는 단점을 가지고 있다.

제 4 절 Reference

[1] Computational Statistics by Givens and Hoeting, Wiley.