Stat 243: Problem Set 8

Eugene Yedvabny

12/03/2014

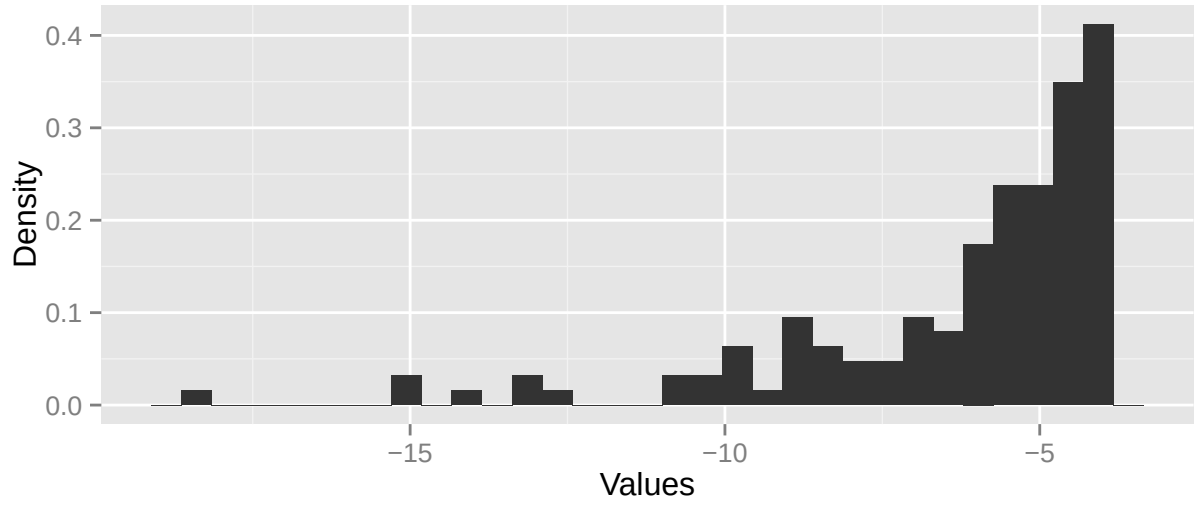
Q1

Here we are trying to estimate the mean of a t-distribution with 3 degrees of freedom and truncated to X < -4. By definition that means the mean should be < -4.

First, the "stupid" method:

```
x <- rt(10000,df=3)
x <- x[x < -4]
sprintf("Mean: %f; Variance: %f; Samples: %d (%f)",
        mean(x), var(x)/length(x), length(x), length(x)/10000)

ggplot(data.frame(x),aes(x=x)) +
  geom_histogram(aes(y=..density..)) +
  xlab("Values") +
  ylab("Density")
```

```
## [1] "Mean: -6.275468; Variance: 0.052734; Samples: 132 (0.013200)"
```



The single-sample variance isn't actually that bad but as we can see only 1.32% of the sampled points are used for the estimation. This makes sense since the probability of drawing a value < -4 from a t-distribution is very small. This can be improved in importance sampling by sampling from a distribution that's a lot more likely to yield an X < -4 and subsequently re-weighing the samples.

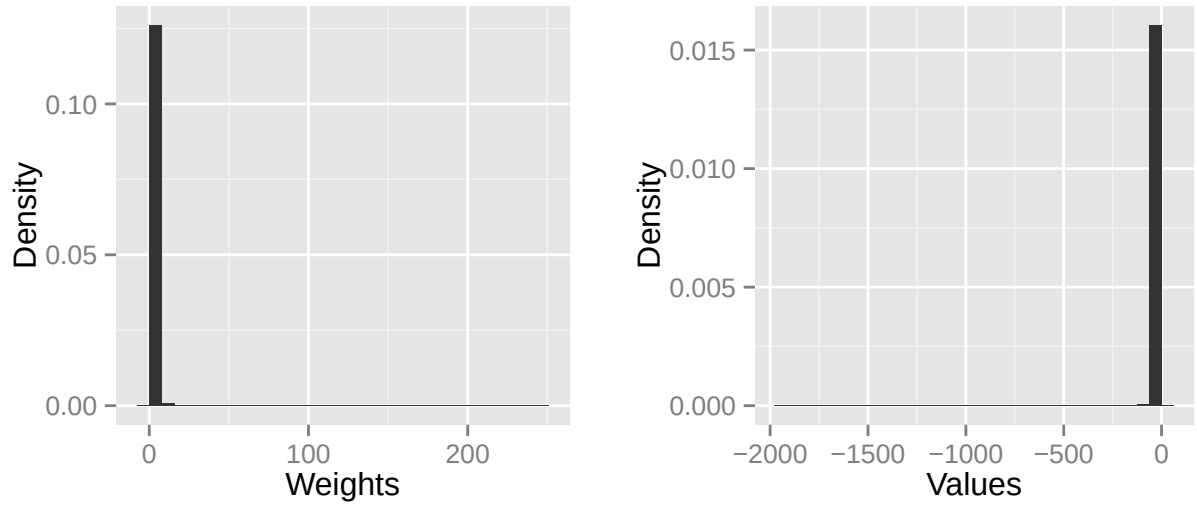
The sampling distribution should be a half-normal distribution centered at -4. Since the distribution is symmetric, rather than discarding values we can take the "negative absolute value" of a sample and thus use all 10000 draws. The `-4` just shifts the center of the distribution to the desired location.

Per our nomenclature:

- f(x) is the truncated t distribution with df=3.
- g(x) is above half-normal distribution.

```
x <- -1*abs(rnorm(10000))-4
f <- dt(x,df=3)/pt(-4,df=3) # Truncated distribution f(x|x<A) = PDF(x)/CDF(A) for x < A.
g <- dnorm(x+4)
w <- f/g

sprintf("Mean: %f; Variance: %f; Samples: %d (%f)",
        mean(x*w), var(x*w)/length(x), length(x), length(x)/10000)

dat <- data.frame(wts = w, vals = x*w)
h1 <- ggplot(dat,aes(x=wts)) +
  geom_histogram(aes(y=..density..)) +
  xlab("Weights") +
  ylab("Density")

h2 <- ggplot(dat,aes(x=vals)) +
  geom_histogram(aes(y=..density..)) +
  xlab("Values") +
  ylab("Density")

grid.arrange(h1,h2,nrow=1)
```

```
## [1] "Mean: -8.728876; Variance: 0.058732; Samples: 10000 (1.000000)"
```

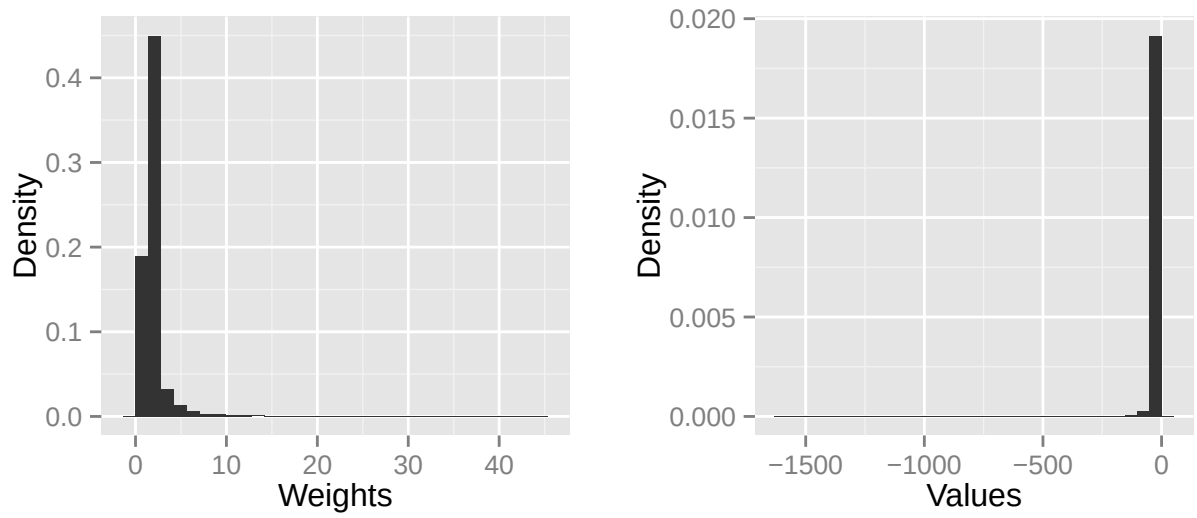


As the above figures show, we drew some rather improbable values from the half-normal, resulting in very small g(x) and thus *huge* weights. This is expected since the normal distribution has 'lighter tails' than the t distribution. While most weights are ~ 1, there are outliers as far as 236.5683414 that have a lot of influence. Even though we're using all 10000 samples, our variance estimate is better than the naive approach but not great.

Now we attemp the same but now sampling from a t distribution:

```
x <- -1*abs(rt(10000,df=3))-4
f <- dt(x,df=3)/pt(-4,df=3)
g <- dt(x+4,df=3)
w <- f/g

sprintf("Mean: %f; Variance: %f; Samples: %d (%f)",
        mean(x*w), var(x*w)/length(x), length(x), length(x)/10000)

dat <- data.frame(wts = w, vals = x*w)
h1 <- ggplot(dat,aes(x=wts)) +
  geom_histogram(aes(y=..density..)) +
  xlab("Weights") +
  ylab("Density")

h2 <- ggplot(dat,aes(x=vals)) +
  geom_histogram(aes(y=..density..)) +
  xlab("Values") +
  ylab("Density")

grid.arrange(h1,h2,nrow=1)
```

```
## [1] "Mean: -12.461366; Variance: 0.105497; Samples: 10000 (1.000000)"
```



Now the variance should be less (dependent on the RNG draw, current max is 43.946491) and correspondingly we don't have too many influential weights on the mean estimator.

Disclaimer: Per Wiki on truncated distrubutions, the PDF of a truncated distribution is the PDF of the original distribution / CDF value at the truncated value. In the case of the t-distribution that translates to dt(x,df=3)/pt(-4,df=3). If this is an erroneous formula, that would invalidate the above simulation results. Sorry if that's the case.

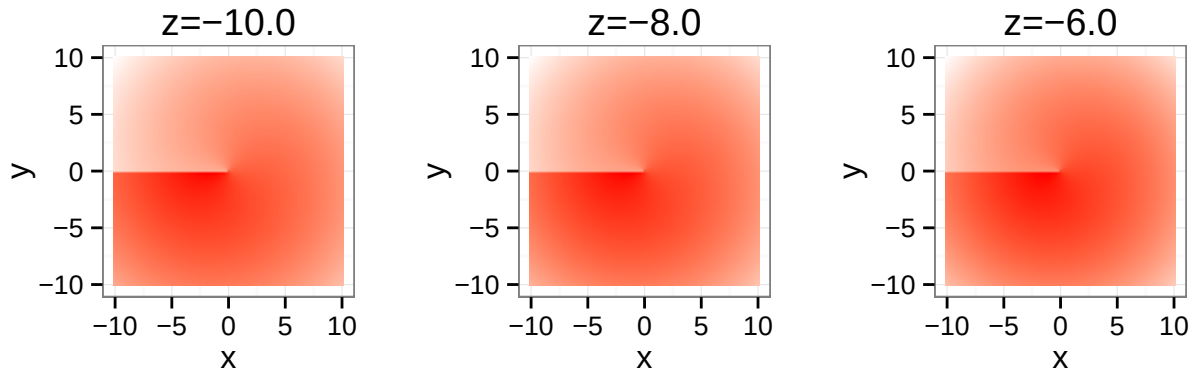
Q2

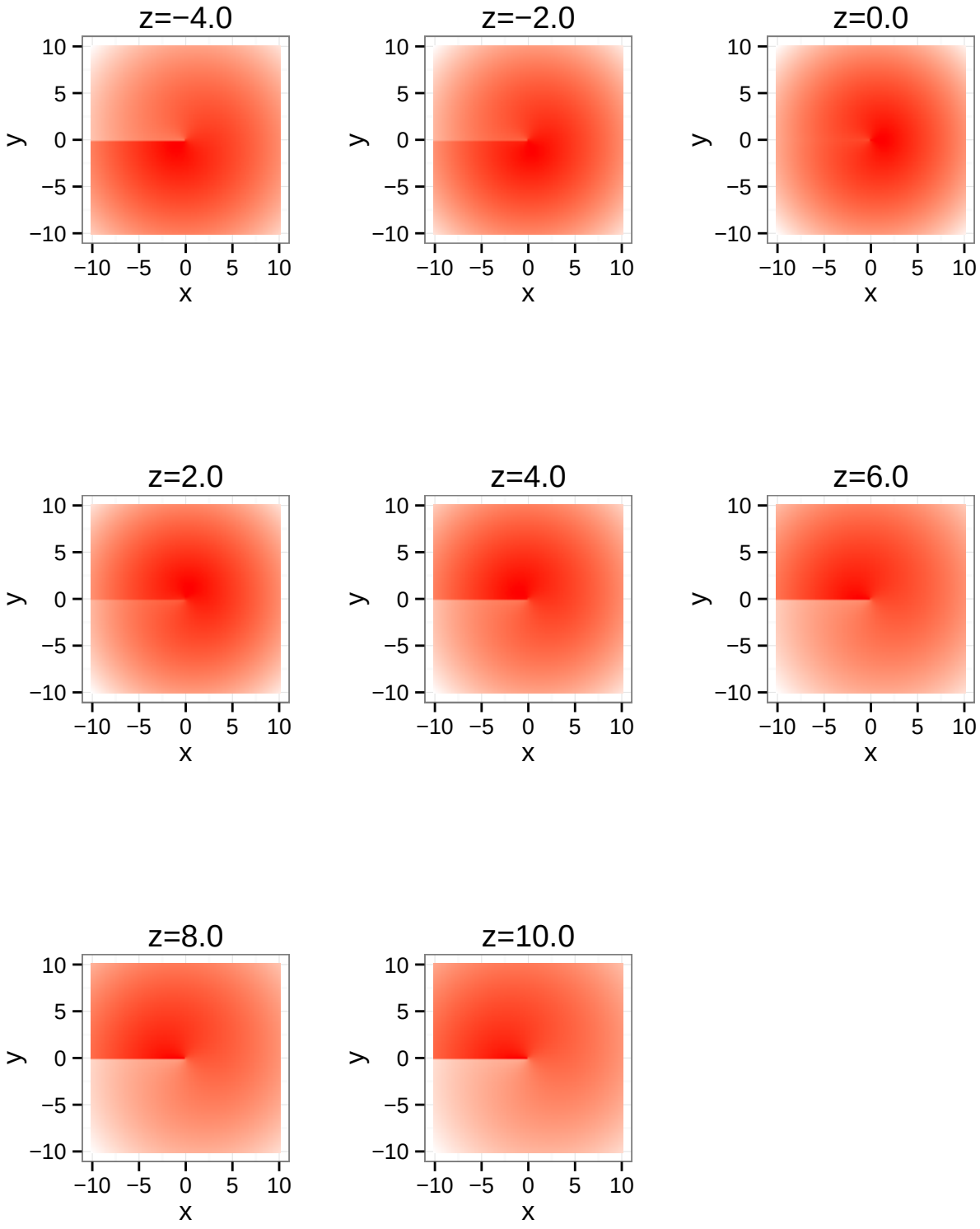
The provided "helical valley" function is reproduced below:

```
theta <- function(x1,x2) atan2(x2, x1)/(2*pi)

heli <- function(x,y,z) {
  f1 <- 10*(z - 10*theta(x,y))
  f2 <- 10*(sqrt(x^2+y^2)-1)
  f3 <- z
  return(f1^2+f2^2+f3^2)
}
```

The function takes in three arguments, so a complete representation would have to be four dimensional. To simplify this we'll take slices through z and plot it for x and y.

```
z_range <- seq(-10,10,length.out=11)
val_range <- seq(-10,10,length.out=101)

slices <- lapply(z_range, function(x){
            slice <- outer(val_range,val_range,heli,x)
            dimnames(slice)<-list(val_range,val_range)
            slice <- melt(slice, varnames=c("x","y"), value.name="z")
            ggplot(slice,aes(x=x,y=y,fill=z)) +
              geom_raster() +
              scale_fill_gradient(low="red", high="white", guide=F) +
              coord_fixed()+
              ggtitle(sprintf("z=%0.1f",x))+
              theme_bw()
          })

grid.arrange(slices[[1]],slices[[2]],slices[[3]],ncol=3)
grid.arrange(slices[[4]],slices[[5]],slices[[6]],ncol=3)
grid.arrange(slices[[7]],slices[[8]],slices[[9]],ncol=3)
grid.arrange(slices[[10]],slices[[11]],ncol=3)
```





Not the most intuitive visualization of the function but it appears to behave like a cork-screw with the

minimum somewhere around (0,0,0). Let's see if the optimization functions can find the right minimum:

```
func <- function(x){heli(x[1],x[2],x[3])}

optim(c(0,0,0),func)$par
```

```
## [1] 0.999978292 0.002730698 0.004284640
```

```
optim(c(10,10,10),func,method = 'BFGS')$par
```

```
## [1] 1.000000e+00 1.220429e-08 1.943689e-08
```

```
optim(c(-10,-10,-10),func,method = 'CG',control = list(maxit = 1000))$par
```

```
## [1]  1.000000e+00 -4.699776e-07 -7.466096e-07
```

```
optim(c(100,-55,12),func,method = 'SANN')$par
```

```
## [1] 1.00433286 0.02444664 0.04054838
```

It appears that the minimum is 0 at (1,0,0), which is not too far off from the prediction based off the visual slices. All algorithms were able to find this minimum from varying starting points, so it doesn't appear like there is an local minimum (also evident from the slices). The Nelder-Meade method is least accurate which BFGS gives the most precise minimum. Conjugate Gradient needed extra iterations to converge, so it's the slowest method. Simulated Annealing is pretty inaccurate but this is likely due to default cooling parameters being used (and the function is smooth, so numerical gradients work well).

The NLM method works pretty well too:

```
nlm(func,c(10,10,10))$estimate
```

```
## [1] 1.000000e+00 2.418588e-10 3.472303e-10
```

```
nlm(func,c(-10,-10,-10))$estimate
```

```
## [1]  1.000000e+00 -2.788853e-10 -4.093534e-10
```

```
nlm(func,c(100,-55,12))$estimate
```

```
## [1] 1.000000e+00 1.917499e-10 1.482544e-09
```

... except it chokes at (0,0,0):

```
nlm(func,c(0,0,0))
```

```
## $minimum
## [1] 100
##
## $estimate
## [1] 0 0 0
##
## $gradient
## [1] -12500000         0         0
##
## $code
## [1] 3
##
## $iterations
## [1] 1
```

here I assume the algorithm is trying to multiply the value by its gradient, and anything multiplied by 0 is 0, so the iterations do not advance anywhere.