Polar

JZ

2025-02-07

In this (rough) document, we start by running the polar slice sampler on $e^{-|x|}$ for different dimensions. $n = 1000$ time-steps are used.

Example 1

```
library(rootSolve)
library(ggplot2)
library(gridExtra)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following object is masked from 'package:gridExtra':
##
##     combine
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
#functions for first example
norm = function(x){ #norm function
  return(sqrt(sum(x^2)))
}

pi1 = function(x, filler){ #pi(x) = exp(-|x|)
  return(exp(-1*norm(x)))
}
pi1_rstar = function(y, d, filler){ #filler used for function abstraction
  return(log(y*d^(-2*d+2))/((d-1)/d^2-1))
}

pi1_init = function(d){
  return(rep(sqrt(d), d))
}
```

```
polar = function(init, fn, n, fn_rstar, emp_dens){
  #init is the initialization
  #fn is the target distirbution
```

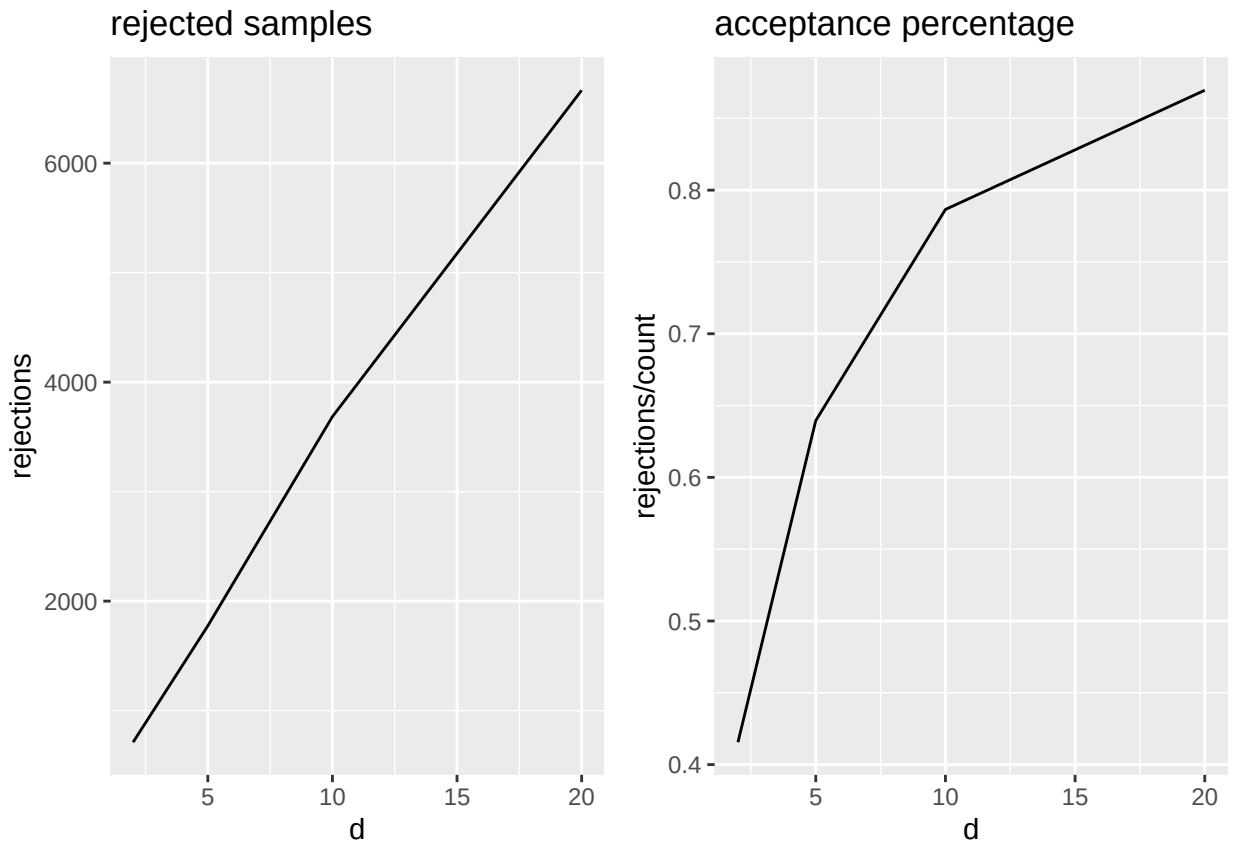
```
  #n is number of time steps
  #fn_rstar is the r* function used
  #emp_dens is extra data used for empirical examples
  start_time = proc.time() #track cpu run time
  d = length(init) #dimension from length of inital point
  chain = data.frame(matrix(data=0, nrow = n+1, ncol = d))
  chain[1,] = init
  rejections = 0 #number of times proposal point rejected
  count = 0 #total iterations including rejected points
  for(i in 1:n){
    x = chain[i,] #current chain value
    y = runif(1, 0, norm(x)^(d-1)*fn(x, emp_dens)) #sample y uniformly
    #print(paste(fn(x, emp_dens), y))
    r_star = fn_rstar(y, d, emp_dens)
    accept = 0
    while(accept == 0){ #iterate until a new point is accepted
      #here, we use fact pi(x) is completely defined by norm
      count = count + 1
      r = runif(1, 0, r_star) #sample r and theta
      theta = rnorm(d)
      x_prop = theta/norm(theta)*r
     # print(paste(y, fn(x_prop, emp_dens), r_star, r))
      if(norm(x_prop)^(d-1)*fn(x_prop, emp_dens) >= y){ #check if it satisfied bound
        accept = 1
        chain[i+1,] = x_prop
      }
      else{
        rejections = rejections + 1
      }

    }
  }
  end_time = proc.time()
  return(list(chain, rejections, count, end_time - start_time))
}
```

```
#generic function to print plots needed for output for all examples and both polar and uniform samplers
sampling_plots = function(d, n, fn, fn_rstar, type, fn_sampling, fn_init, emp_dens){
  #fn_init is the initialization
  #fn is the target distirbution
  #n is number of time steps
  #d is the dimension
  #fn_sampling is either polar or uniform function
  #fn_rstar is the r* function used (or null for uniform)
  #emp_dens is extra data used for empirical examples
  plots = list() #norm plots over time
  acf_plots = list() #acf plots
  rejections = c() #number of rejections by dimension
  count = c() #total iterations by dimension
  for(i in 1:length(d)){
    ret = fn_sampling(fn_init(d[i]), fn, n, fn_rstar, emp_dens) #run sampler and get chain
    x = ret[[1]]
    rejections = c(rejections, ret[[2]])
    count = c(count, ret[[3]])
```

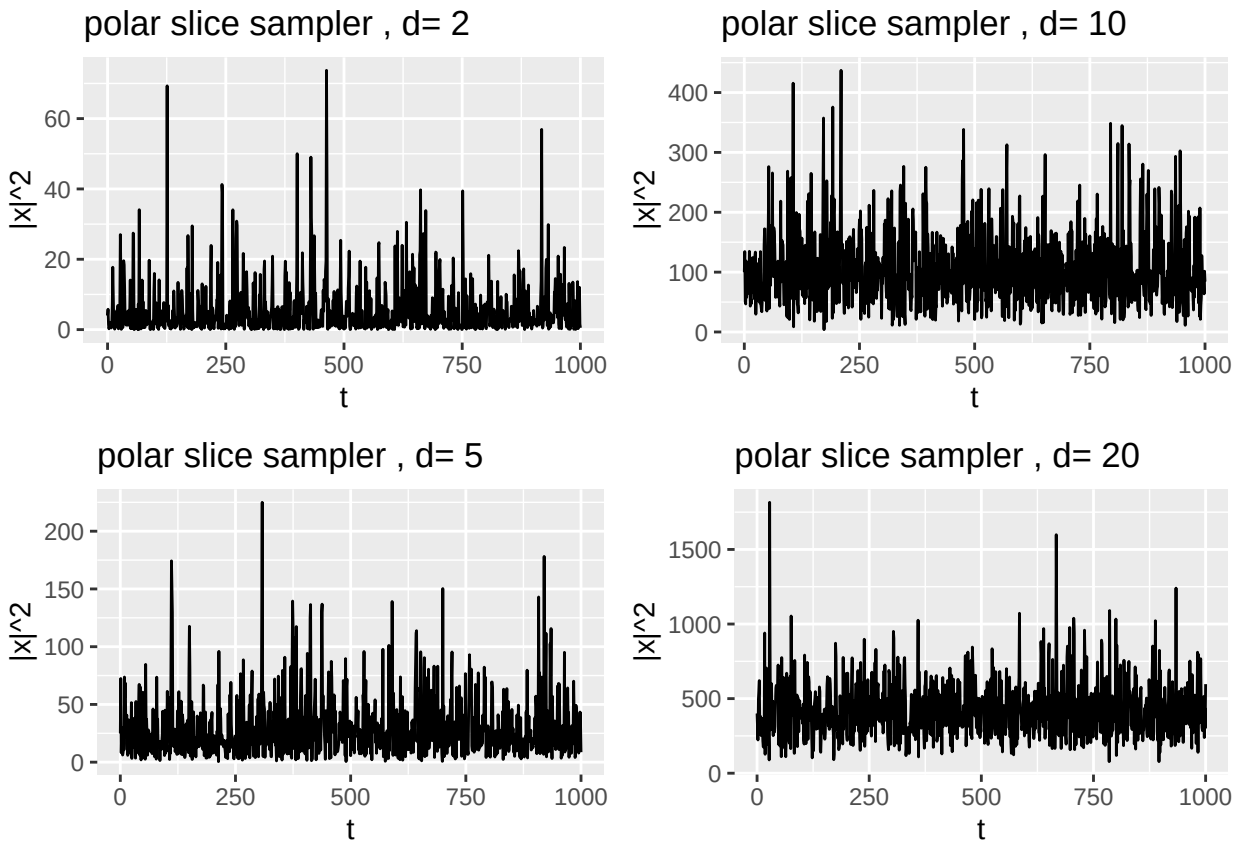
```
    x['norm'] = rowSums(x^2)
    plots[[i]] = ggplot(data = x, aes(x=seq(0,n, by=1), y = norm)) +
      geom_path() + labs(x = 't', y = '|x|^2',
                         title = paste(type, ', d=', d[i]))

    acf_df = data.frame(autocorrelation = acf(x['norm'], plot=F, lag.max=20)$acf, x=seq(0, 20, by=1))
    acf_plots[[i]] = ggplot(acf_df, aes(x=x, y = autocorrelation)) +
      geom_bar(stat = "identity", width=0.1) +
      labs(x = 'lag', y = 'autocorrelation', title = paste('ACF plot, d = ',d[i]) )
  }
  reject_plot = ggplot(NULL,aes(x = d, y = rejections)) + geom_path() +
    labs(title = 'rejected samples')
  accept_per = ggplot(NULL,aes(x = d, y = rejections/count)) + geom_path() +
    labs(title = 'acceptance percentage')
  #return plots
  return(list(plots, acf_plots, reject_plot, accept_per))
}
```

```
set.seed(2001)
polar1 = sampling_plots(c(2, 5, 10, 20), 1000, pi1, pi1_rstar, 'polar slice sampler', polar, pi1_init,
```

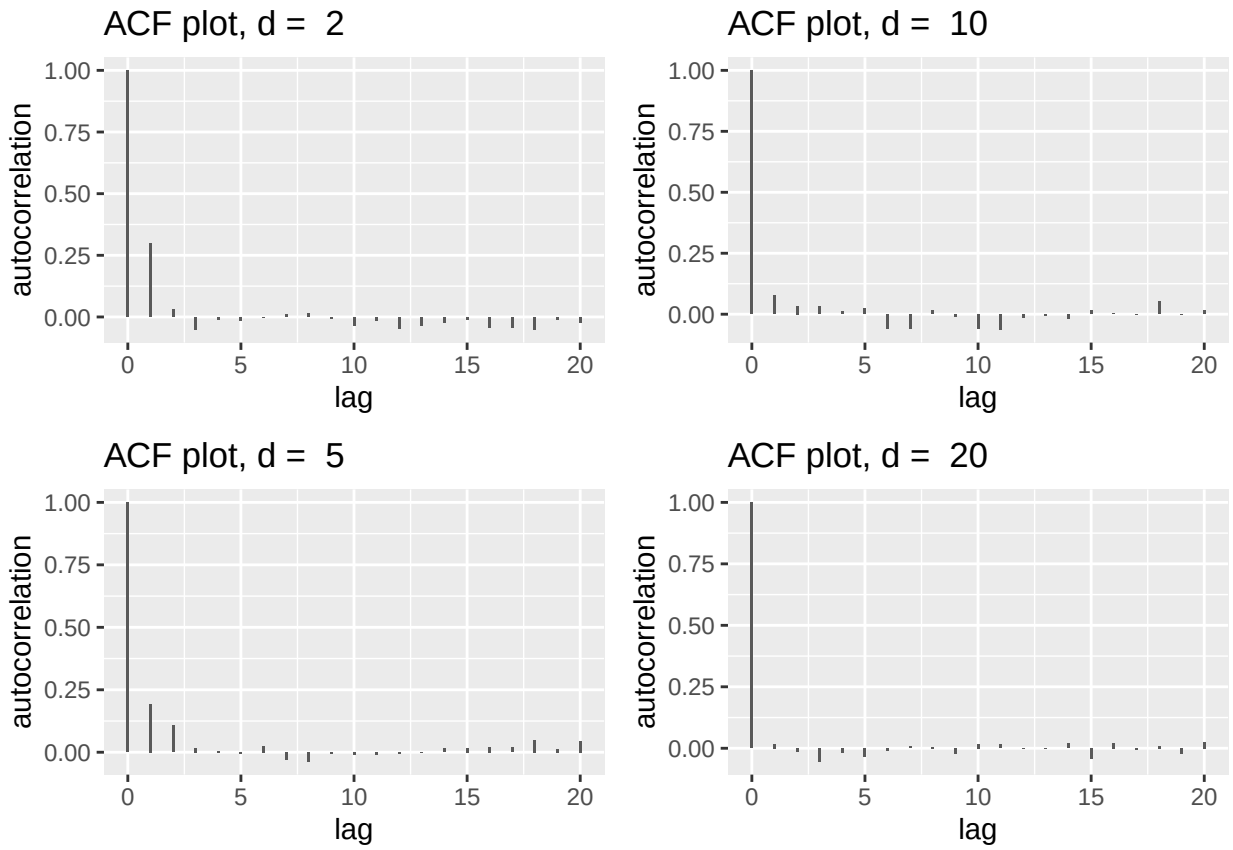
```
grid.arrange(grobs = list(polar1[[3]], polar1[[4]]) , ncol = 2, as.table = FALSE)
```



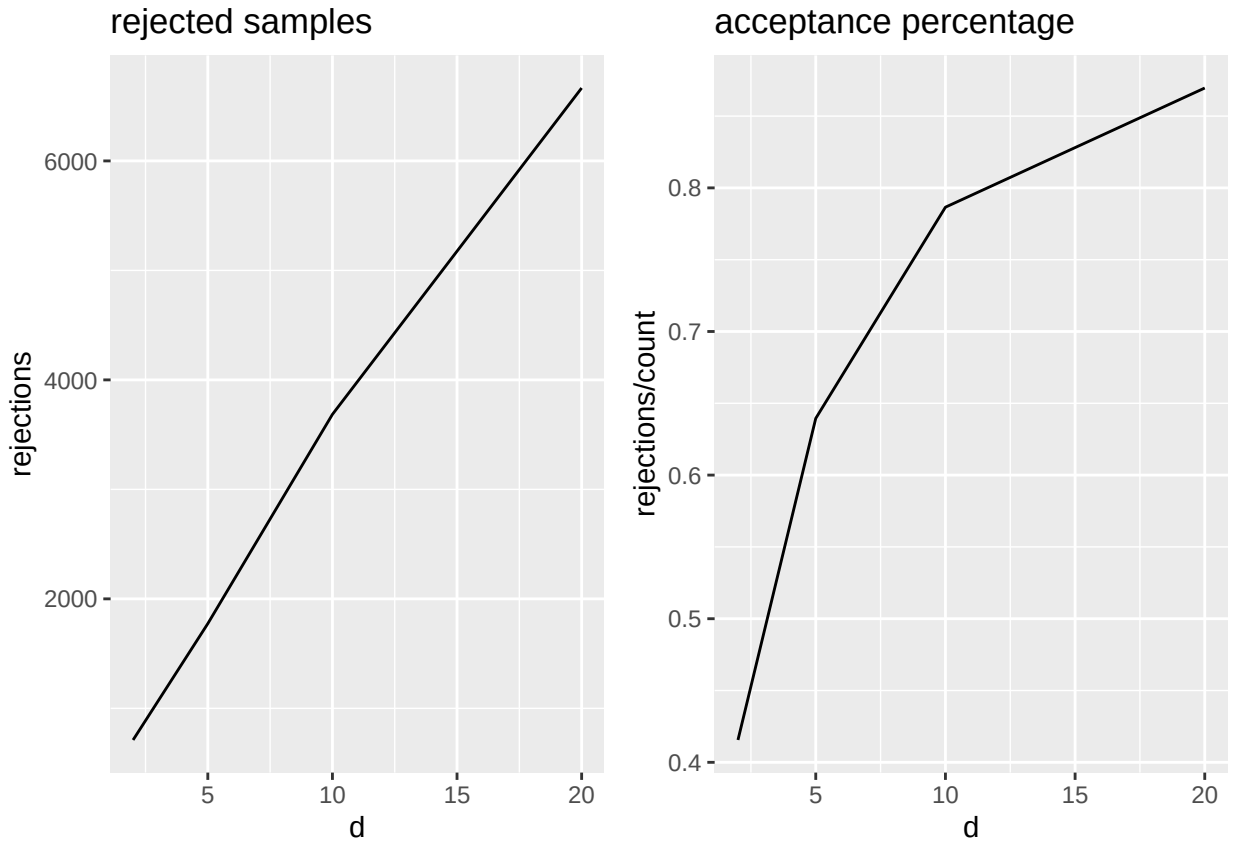

```
grid.arrange(grobs = polar1[[1]] , ncol = 2, as.table = FALSE)
```




```
grid.arrange(grobs = polar1[[2]] , ncol = 2, as.table = FALSE)
```




```
grid.arrange(grobs = list(polar1[[3]], polar1[[4]]) , ncol = 2, as.table = FALSE)
```



Here, we can see that the autocorrelation is low for all dimensions. Note that the plots use $n = 1000$ step chain, but I tested for up to $n = 25000$ with similar findings. We can also see that the number of rejected samples increases linearly with dimension, and is not very high. This preserves efficiency of the algorithm in higher dimensions. Of course, since the number of accepted samples is fixed at 1000, the rejection ratio increases rather quickly. In fact, here we used a rather naive (loose) choice of r^*, chosen through a numerical computation (using a 'trick' with δ being a free variable we choose)

$$y \leq \delta^{-(d-1)}(\delta|x|)^{d-1}e^{-|x|} \leq \delta^{-(d-1)}e^{(d-1)\delta|x|}e^{-|x|} = \delta^{-(d-1)}e^{((d-1)\delta-1)|x|}$$
$$\implies |x| \geq \frac{\log y\delta^{d-1}}{(d-1)\delta - 1} = r^*$$

We need $(d-1)\delta - 1 < 0$ for the RHS to converge to 0. It turns out that $\delta = \frac{1}{d^2}$ gives the tightest bound of $|x|^{d-1}e^{-|x|}$. In this simple example, we could of actually found the roots of $f_1(x) - y$ and used that as a tighter r^*, which would decrease number of rejections. From a rejection sampling lens, the ball of radius r^* is our scaled 'proposal density' $g(x)$ scaled by factor K. As d increases, r^* rapidly increases, and so the volume in the ball increases (can compute how fast). This means that our theoretical K value increases rapidly (since the proposal density always integrates to 1). This is evidenced by our rejection ratio increasing to 1. We proceed to compare polar slice sampling with the uniform slice sampler. We immediately see, even at lower dimensions, that autocorrelation spikes, meaning that the uniform slice sampler is not going through the entire support very well (see below plots).

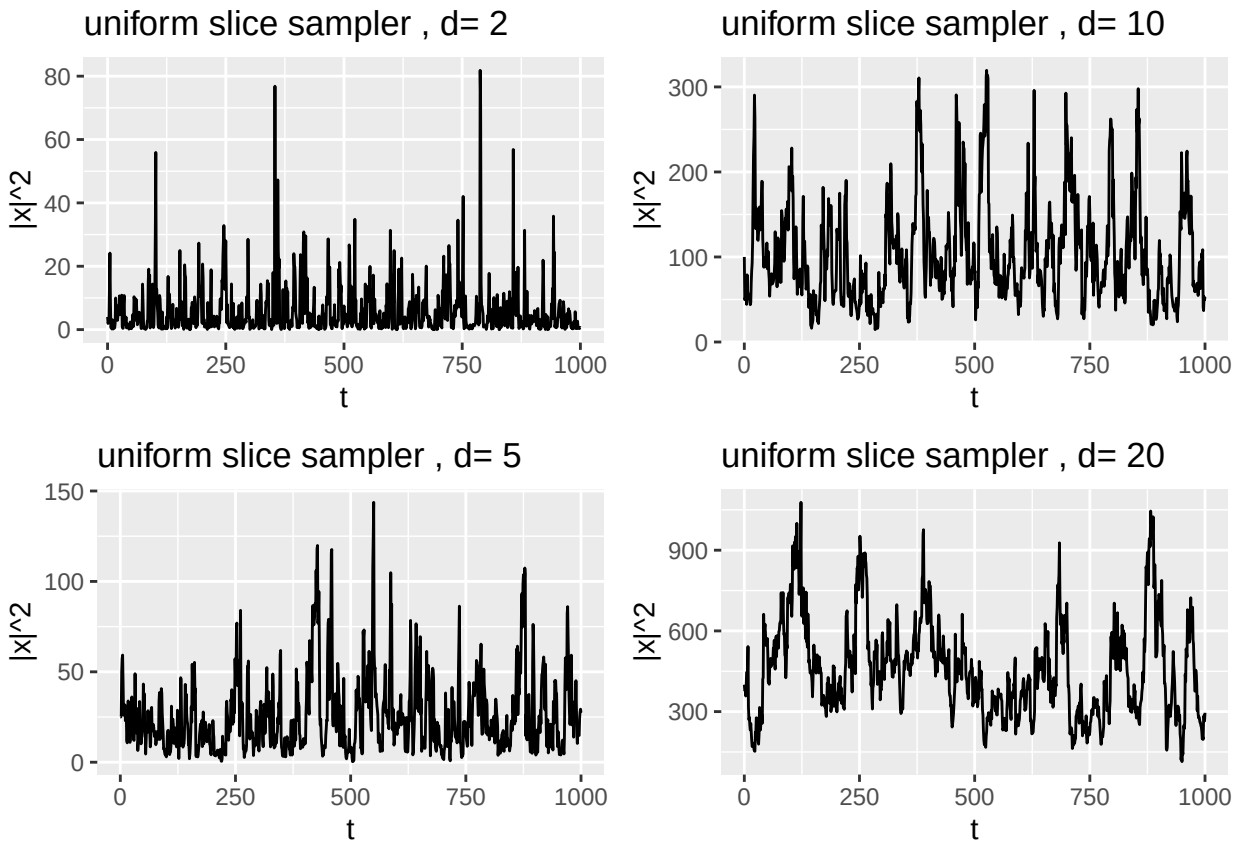
In our toy example, $e^{-|x|}$ is spherically symmetrical since it is a function of the norm. Each slice at y will partition the points $\pi(x) \geq y$ as being inside the ball of radius y. Intuitively, uniform slice sampler is poor because in high dimensions, a large proportion of points will be close to the boundary of the ball of radius r (which is the value $|x|$ so that $f(x) = y$). For example the ratio of points within a ball of ratio $\frac{r}{2}$ and ball of ratio r is $\frac{1}{2^d}$. So in high dimensions, uniformly sampling in this ball drastically oversamples points away from the mode, i.e. the origin (from below image there are no the entire range $(-r, r)$, leading to poor efficiency and convergence results. On the other hand, the polar slice sampler will sample a radius uniformly within

the ball, and so each distance from origin has equal chance of being chosen (in polar sampling, we actually consider $f_1(x) = |x|^{d-1}\pi(x)$, which is different than $\pi(x)$ but the same idea holds). This means the chain will mix faster leading to better convergence properties.

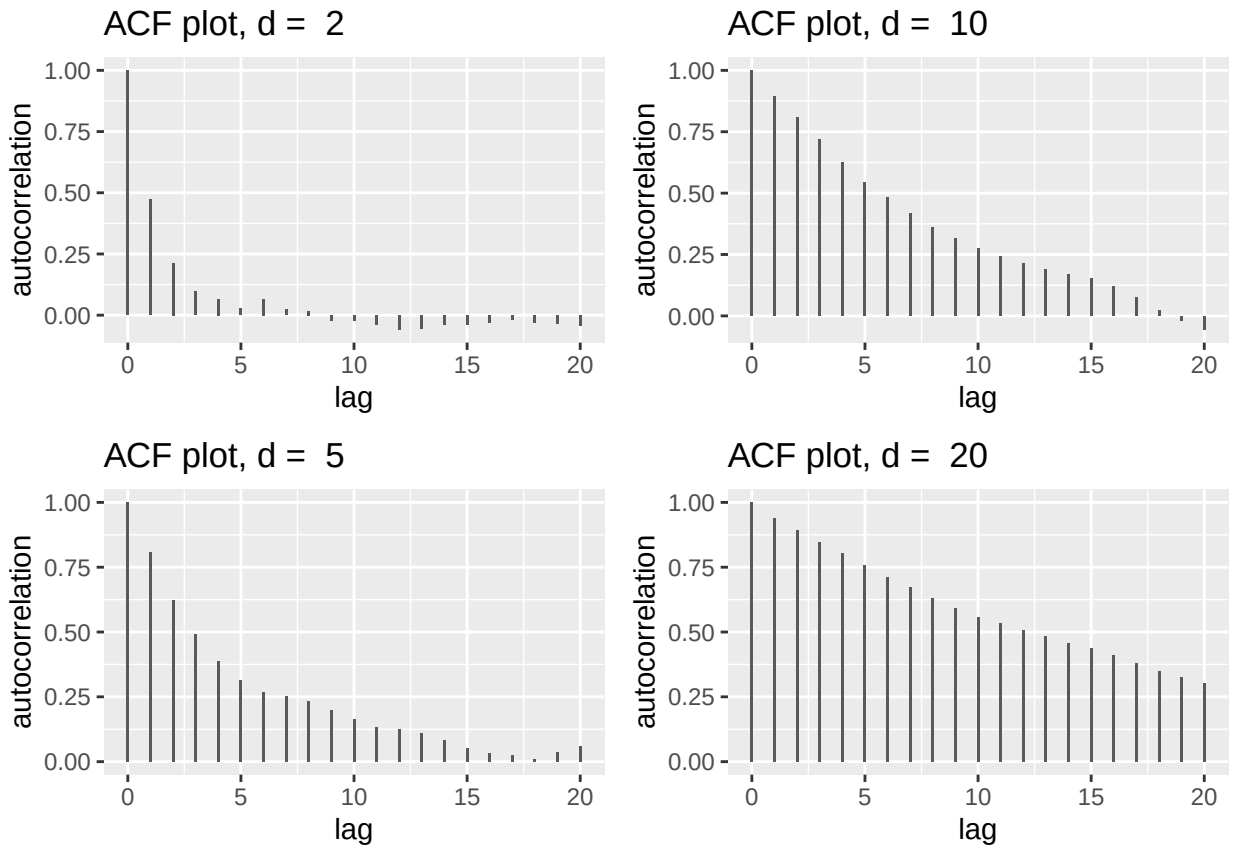
```
uniform_slice = function(init, fn, n, fn_rstar, fn_lim){
  #init is the initialization
  #fn is the target distirbution
  #n is number of time steps
  #fn_rstar is null
  #fn_lim are the bounds for second step of uniform slice sampler
  start_time = proc.time()
  d = length(init) #dimension from length of initial point
  chain = data.frame(matrix(data=0, nrow = n+1, ncol = d))
  chain[1,] = init
  for(i in 1:n){
    x = chain[i,] #current chain value
    y = runif(1, 0, fn(x, d)) #sample y uniformly
    lim = fn_lim(y, d)
    #we assume that the functions are 'simple' so that that the limits are a single interval in R^d
    x_new = rnorm(d)
    x_new_norm = norm(x_new)
    x_new = x_new/x_new_norm*lim
    u = runif(1)^(1/d)
    chain[i+1,] = x_new * u
  }
  end_time = proc.time()
  #for abstraction, we return dummy 0's where we would have rejection plots in polar
  return(list(chain, 0, 0, end_time - start_time))
}
```

```
pi1_lim = function(y, d){
  return(rep(-log(y), d))
}
set.seed(2001)
uni1 = sampling_plots(c(2, 5, 10, 20), 1000, pi1, pi1_rstar, 'uniform slice sampler', uniform_slice, pi
```

```
grid.arrange(grobs = uni1[[1]] , ncol = 2, as.table = FALSE)
```




```
grid.arrange(grobs = uni1[[2]] , ncol = 2, as.table = FALSE)
```




```
cpuact = data.frame(row.names = c(2,5,10,20))

#run polar and uniform for various dimensions and compare CPU and ACT times
set.seed(100)
for(i in c(2,5,10,20)){
  u = uniform_slice(rep(sqrt(i), i), pi1, 1000, pi1_rstar, pi1_lim)
  p = polar(rep(sqrt(i), i), pi1, 1000, pi1_rstar, 0)
  cpuact[paste(i), 'act.p'] = sum(acf(rowSums(p[[1]]^2), plot=F)[[1]])
  cpuact[paste(i), 'act.u'] = sum(acf(rowSums(u[[1]]^2), plot=F)[[1]])
  cpuact[paste(i), 'cpu.p'] = p[[4]][['elapsed']]
  cpuact[paste(i), 'cpu.u'] = u[[4]][['elapsed']]
}
cpuact$ac.p = cpuact$act.p * cpuact$cpu.p
cpuact$ac.u = cpuact$act.u * cpuact$cpu.u
```

```
grid.table(round(cpuact, 2))
```

	act.p	act.u	cpu.p	cpu.u	ac.p	ac.u
2	1.44	4.19	0.3	0.14	0.43	0.6
5	1.19	4.36	0.42	0.25	0.5	1.09
10	0.99	9.17	0.65	0.37	0.65	3.37
20	0.94	19.07	1.14	0.62	1.07	11.81

Example 2

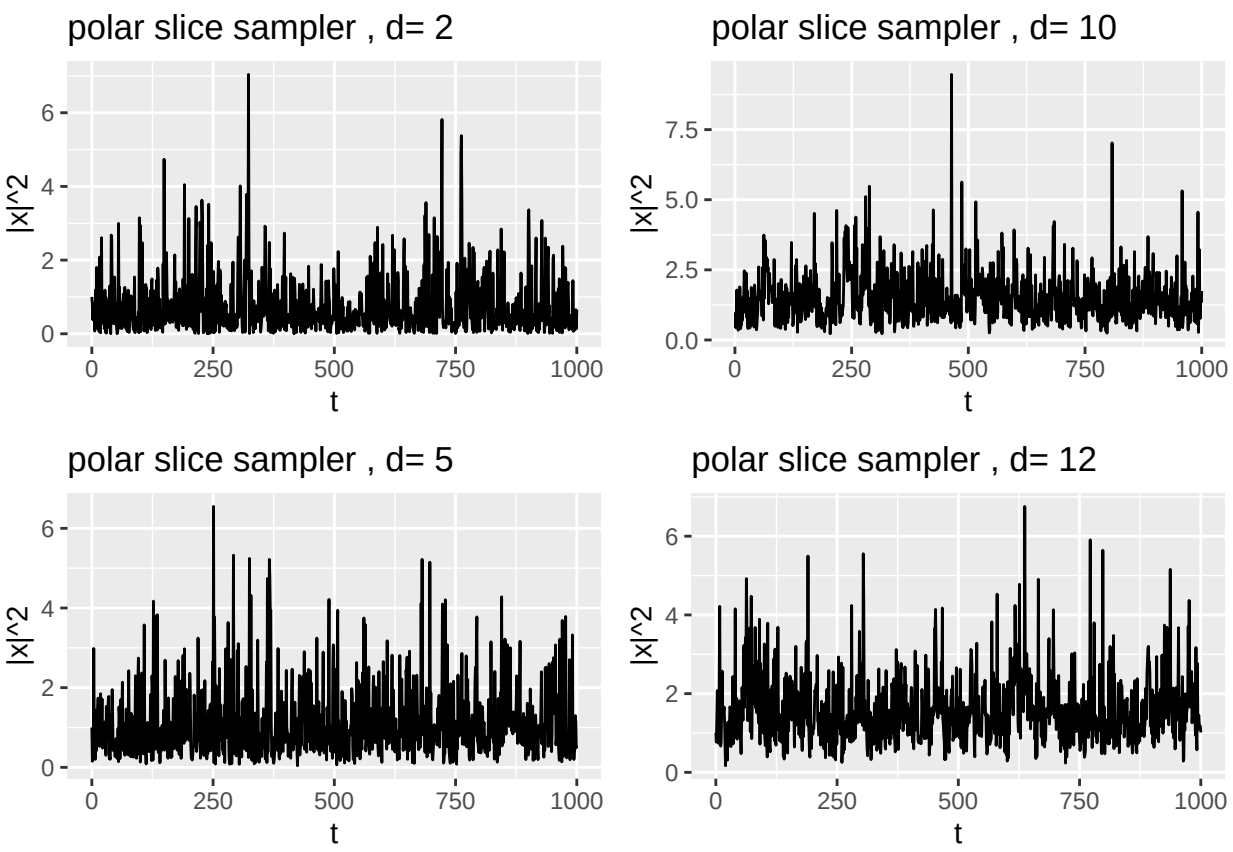
Next, we will consider another example, with the function $\pi_2(x) = e^{-\sum_{i=1}^{d} i x_i^2}$. In this example (similar to previous), we have

$$y \leq |x|^{d-1} e^{-\sum_{i=1}^{d} i x_i^2} \leq |x|^{d-1} e^{-|x|^2} \leq \delta^{-\frac{d-1}{2}} e^{(\frac{d-1}{2}\delta - 1)|x|^2} \implies |x| \geq \sqrt{\frac{\log y \delta^{\frac{d-1}{2}}}{\frac{d-1}{2}\delta - 1}} = r^*$$

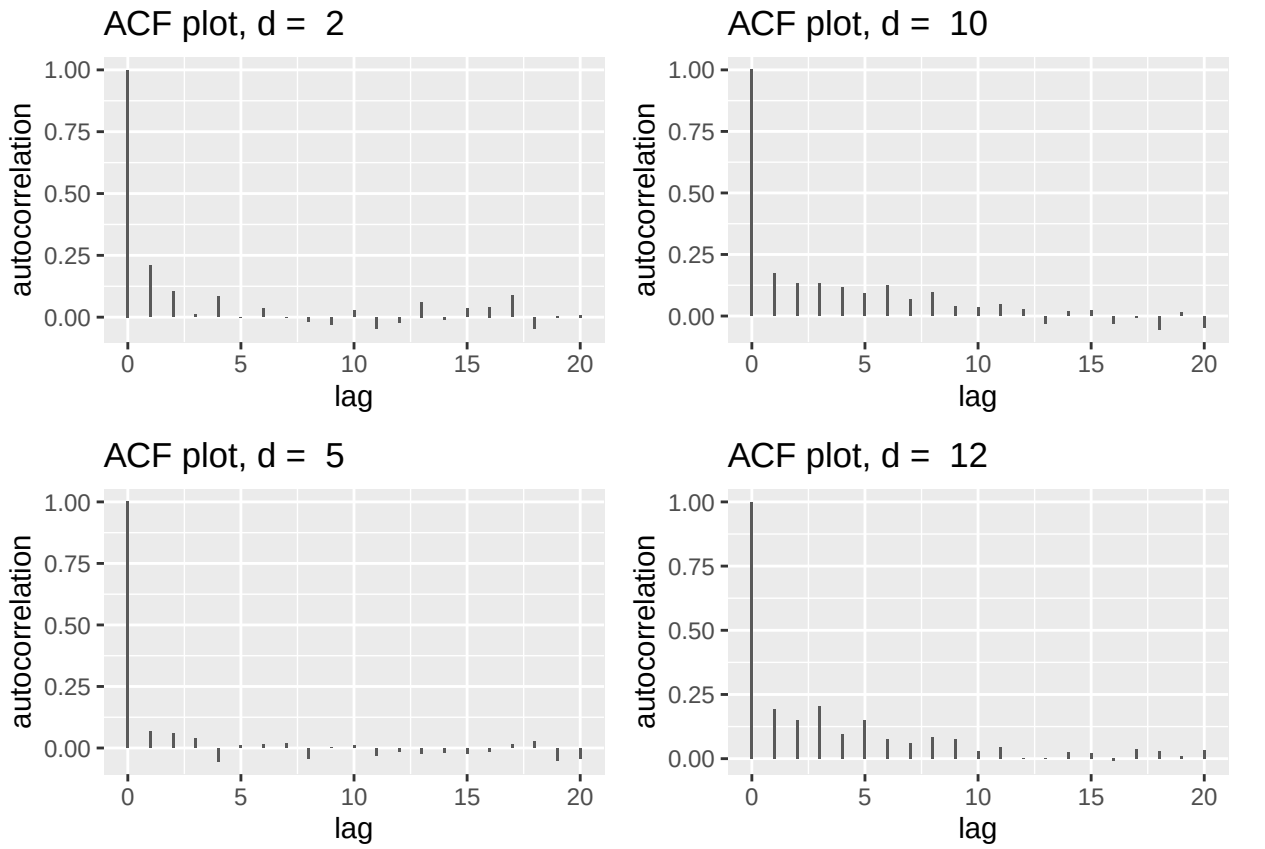
Again, we want $\frac{d-1}{2}\delta - 1 < 0$ for convergence, and we choose $\delta = 2d^{-\frac{3}{2}}$

```
#necessary functions and running of ex. 2
pi2 = function(x, filler){ #pi(x) = exp(-|x|)
  return(exp(-1*sum(seq(1, length(x), by=1)*x^2)))
}
pi2_rstar = function(y, d, filler){
  num = log(y*(2/d^1.5)^((d-1)/2))
  denom = (d-1)/d^1.5 - 1
  return(sqrt(num/denom))
}
pi2_init = function(d){
  return(rep(sqrt(1/d),d))
}

set.seed(2101)
polar2 = sampling_plots(c(2, 5, 10,12), 1000, pi2, pi2_rstar, 'polar slice sampler', polar, pi2_init, 0
```

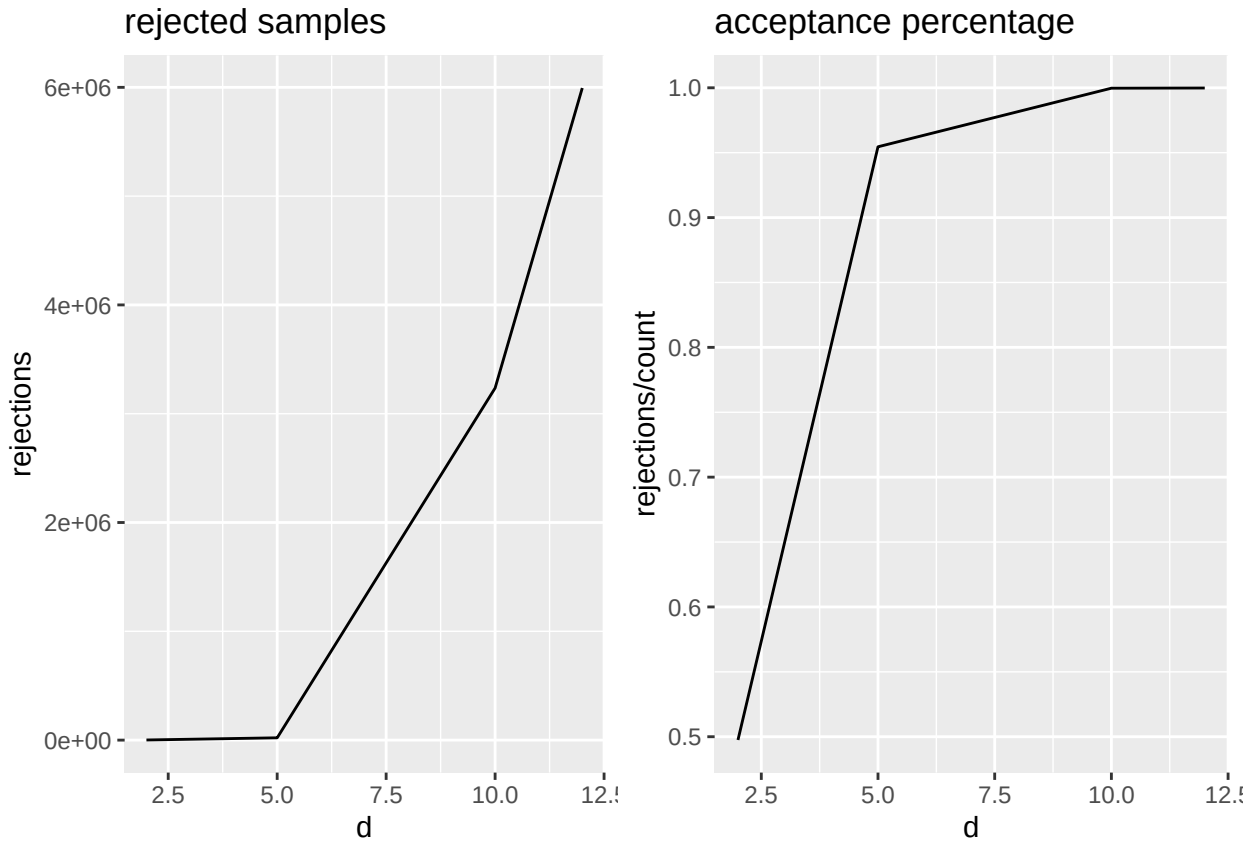
```
grid.arrange(grobs = polar2[[1]] , ncol = 2, as.table = FALSE)
```



```
grid.arrange(grobs = polar2[[2]] , ncol = 2, as.table = FALSE)
```



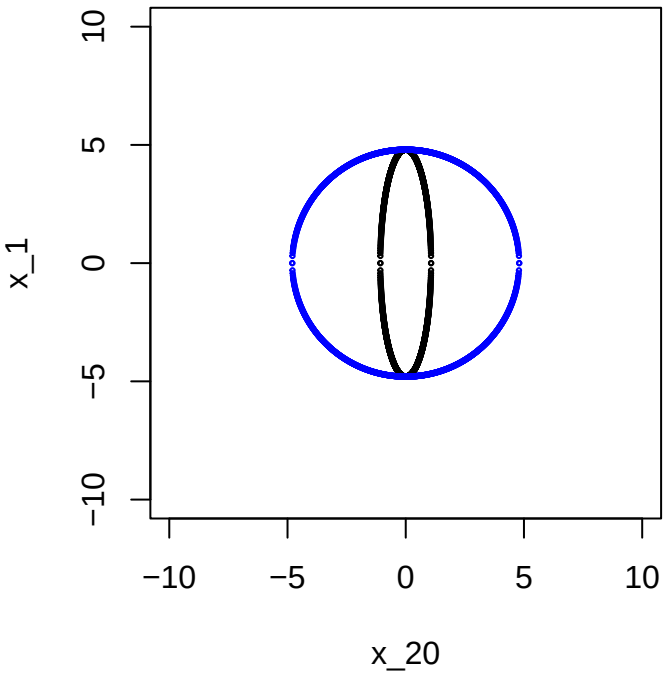

```
grid.arrange(grobs = list(polar2[[3]], polar2[[4]]) , ncol = 2, as.table = FALSE)
```



We can immediately see that the number of rejections has drastically increased for all d, and increases exponentially as a function of d. The rejection percentage immediately shoots up to around 1. In fact, for $d = 15$ the algorithm failed to run and got stuck at around 61 million rejections and 150 iterations (for a single run). Intuitively, this is because π_2 is not spherically symmetrical and our chosen r^* value is dependent on the norm of x (which of course is spherically symmetrical). This means that for large choices of y, r^* will increase (as a function of y). However, for large $d, |x|$, we have $\sum_{i=1}^{d} ix_i^2 > |x|^2 \implies e^{-\sum_{i=1}^{d} ix_i^2} << e^{-|x|^2}$. Thus most values of $r \sim Uniform(0, r^*)$ will be too large and not satisfy constraint $|x|^{d-1}\pi_2(x) \geq y$. For a visualization, consider π_2 as having 'elliptical' cross-sections of the form $\sum_{i=1}^{m} ix_i^2 = -\log \frac{y}{(r^*)^{d-1}}$. Taking 2 dimensional cross-sections for $d = 20$, since each x_i is independent, we can model the (1,20) marginal as proportional to the below ellipse ($r^* = |x_{20}|$)

$$x_1^2 + 20x_{20}^2 = -\log k \implies x_1 = \pm\sqrt{-\log y - 20x_{20}^2}$$

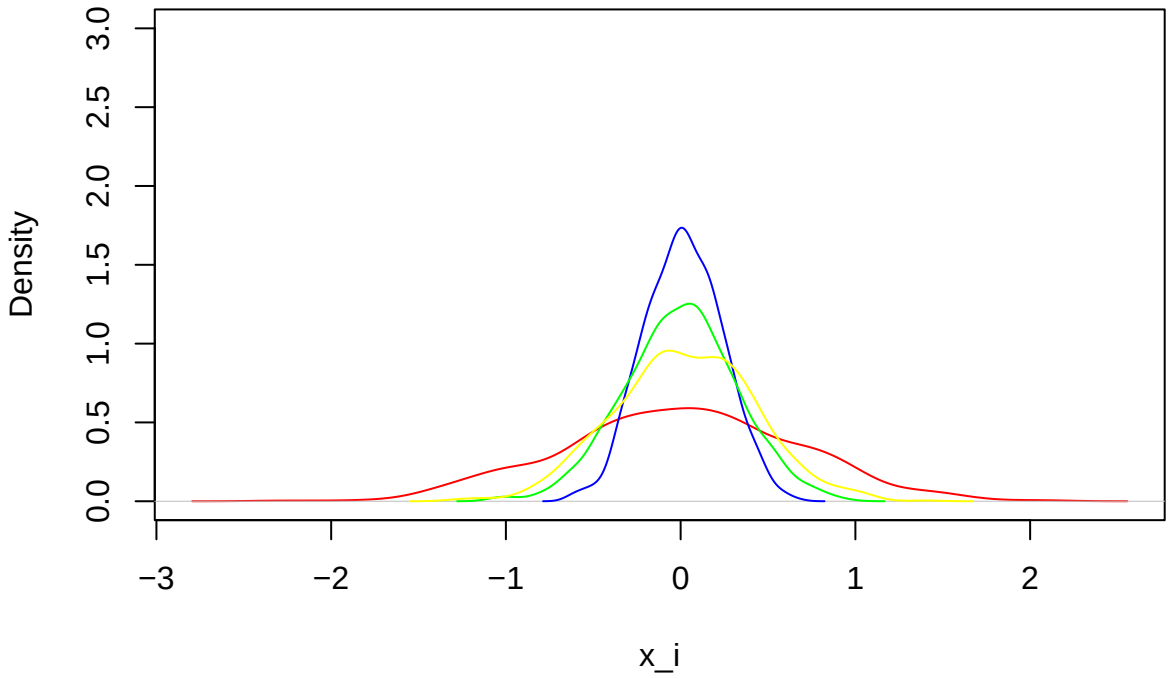
```
#ellipse and circle representing marginals of ex. 1 and ex. 2 (using above forumla)
a = sqrt(-log(10^-10)/20)
ac = sqrt(-log(10^-10))
x = seq(-a, a, length.out = 1000)
xc = seq(-ac, ac, length.out = 1000)

par(pty="s")
plot(x, sqrt(-log(10^-10) - 20*x^2), xlim = c(-10, 10), ylim = c(-10 ,10), cex = 0.3,
     xlab = 'x_20', ylab = 'x_1')
points(x, -sqrt(-log(10^-10) - 20*x^2), cex=0.3)
points(xc, sqrt(-log(10^-10) - xc^2), cex=0.3, col='blue')
points(xc, -sqrt(-log(10^-10) - xc^2), cex=0.3, col='blue')
```



Here, we use $\frac{y}{(r^*)^{d-1}} = 10^{-10}$, which is reasonable for large y and corresponding r^*. We are examining the function π_2, but $\pi_2|x|^{d-1}$ will also be elliptical since $|x|^{d-1}$ is itself spherical. From these cross-sections, we can see that the norm is maximized on the axes (i.e. when $x_{20} = 0$) for both the ellipse and circle. Extrapolating to the function π_2, the norm will be maximized when $x_i = 0$ for $i = 2, 3, ..., d$ for π_2 but the norm value is actually the same for both 'shapes'. In our first example, any radius within the ball radius r^* satisfied $|x|^{d-1}\pi(x) \geq y$, but for this example many choices for r will not satisfy $|x|^{d-1}\pi_2(x) \geq y$, as the area of the ellipse around 3 times smaller (and in d dimensions, the ratio of volumes is taken to dth power). Note that $e^{-|x|^2}$ is a poor approximation of $e^{\sum_{i=1}^{d} ix_i^2}$, which we then approximate again (w.r.t. the norm) to determine r^*, multiplying the issue. In the language of rejection sampling, the constant K is very large in order to have $g(x) \geq |x|^{d-1}\pi_2(x)$ where g is the surface of a d dimensional ball (scaled to have density 1), leading to a small acceptance probability. For this reason, we also see higher autocorrelation in the Markov chains.

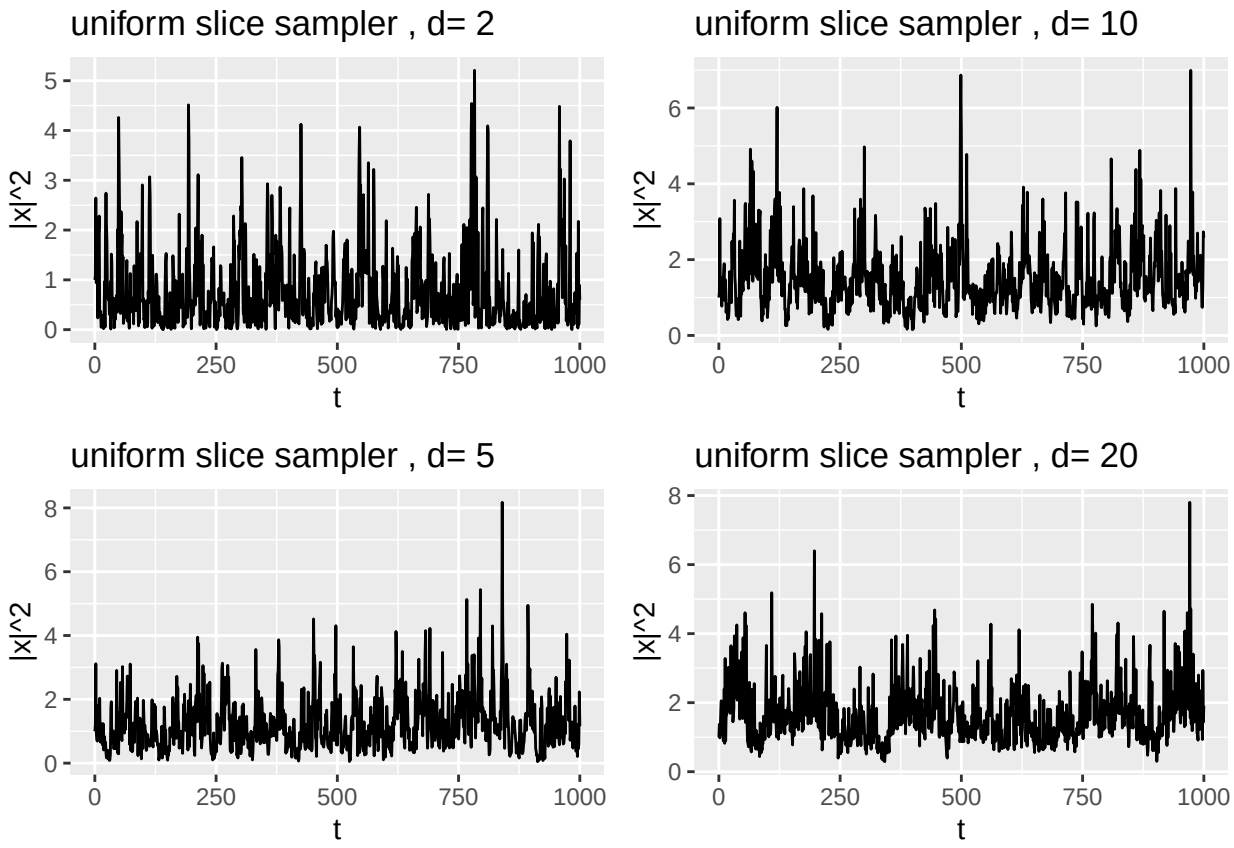
We try again with a tighter r^*, using an exact root finding method for $|x|^{d-1}\pi_2(x) - y = 0$ (function is below but results not printed), but the algorithm still fails to converge for high dimensional d, showing the issue is in the lack of spherical symmetry. Of course, computing $\sup\{|x| | |x|^{d-1}\pi_2(x) \geq y\}$ exactly will help, but there will still be far higher rejections than for $\pi(x)$. For visualization, and to see the polar sampler does work, we will show the $1, 3, 5, 10$ marginals of π_2 for $d = 10$. We also again compare with the uniform slice sampler and see that it also exhibits high autocorrelation and does a poor job of convergence.

```
#printing marginal distributions of resulting chain
set.seed(2102)
test = polar(rep(sqrt(1/10),10), pi2, 1000, pi2_rstar)

plot(density(test[[1]][,1]), ylim = c(0, 3), col = 'red', xlab = 'x_i', main='Marginal Densities')
lines(density(test[[1]][,10]),col = 'blue')
lines(density(test[[1]][,5]),col = 'green')
lines(density(test[[1]][,3]),col = 'yellow')
```

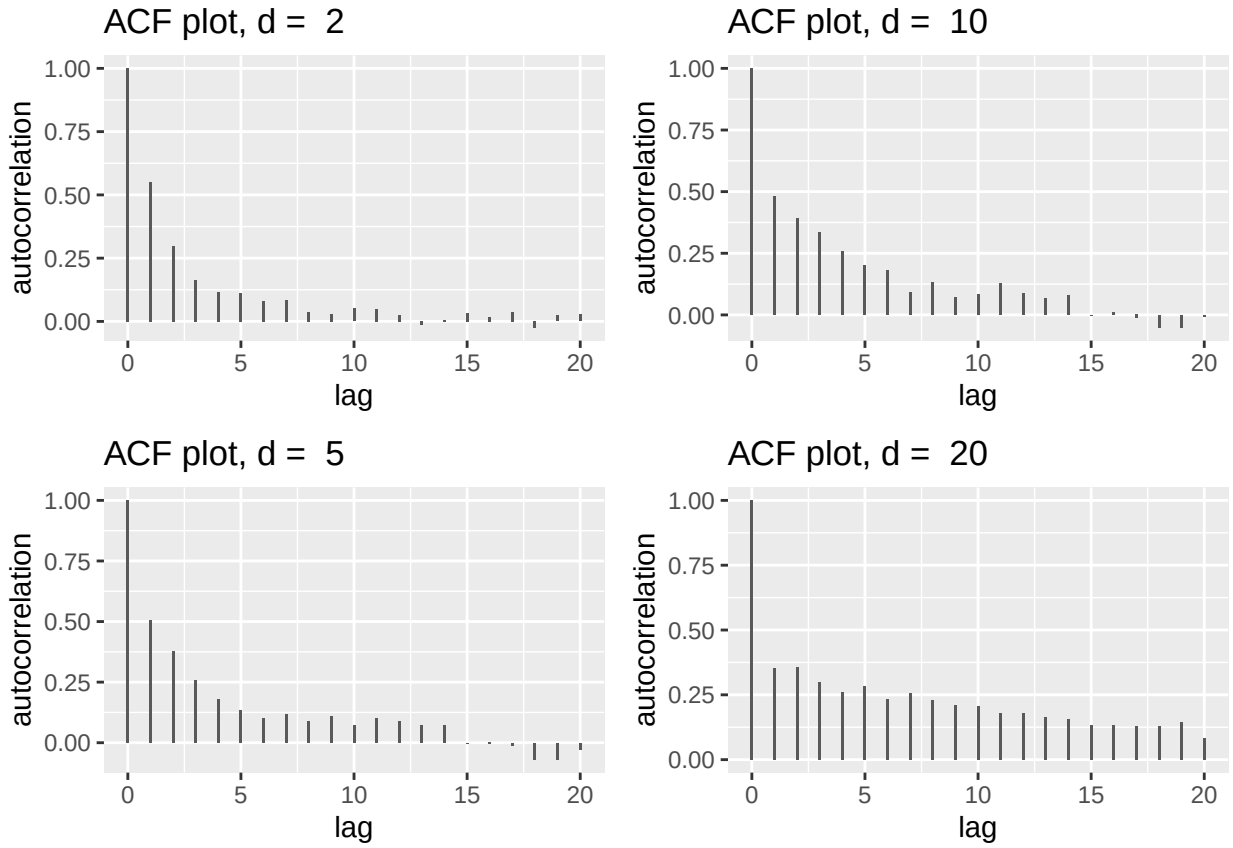



```
#alternative r* function; results omitted
pi2_rstar2 = function(y, d, filler){
  roots = uniroot.all(function(x){return(abs(x)^(d-1)*exp(-x^2) - y)}, c(0, 10))
  if(length(roots) == 0){
    print('hasd')
  }
  r_star = ifelse(length(roots) > 0 &&  max(roots) > 10^(-7), max(roots), 10)
  return(r_star)
}
pi2_lim = function(y, d){
  return(sqrt(-log(y))/sqrt(seq(1, d, by=1)))
}

#run uniform
set.seed(21)
uni2 = sampling_plots(c(2, 5, 10, 20), 1000, pi2, pi2_rstar, 'uniform slice sampler', uniform_slice, pi
```

```
grid.arrange(grobs = uni2[[1]] , ncol = 2, as.table = FALSE)
```




```
grid.arrange(grobs = uni2[[2]] , ncol = 2, as.table = FALSE)
```




```
#get CPU and ACT times for various functions (levels of symmetry), comparing uniforma and polar
ac = function(d){
  cpuact_t = data.frame(row.names = c(0,0.25, 0.5, 0.75, 1))
  for(i in c(0,0.25, 0.5, 0.75, 1)){
    pi2_t = function(x, d){
      return(exp(-1*sum(seq(1, d, by=1)^i*x^2)))
    }
    pi2_lim_t = function(y, d){
      return(sqrt(-log(y))/sqrt(seq(1, d, by=1)^i))
    }

    u_t = uniform_slice(rep(d^(0.5-i),d), pi2_t, 1000, pi2_rstar, pi2_lim_t)
    p_t = polar(rep(d^(0.5-i),d), pi2_t, 1000, pi2_rstar, d)
    cpuact_t[paste(i), 'act.p'] = sum(acf(rowSums(p_t[[1]]^2), plot=F)[[1]])
    cpuact_t[paste(i), 'act.u'] = sum(acf(rowSums(u_t[[1]]^2), plot=F)[[1]])
    cpuact_t[paste(i), 'cpu.p'] = p_t[[4]][['elapsed']]
    cpuact_t[paste(i), 'cpu.u'] = u_t[[4]][['elapsed']]
  }
  cpuact_t$ac.p = cpuact_t$act.p * cpuact_t$cpu.p
  cpuact_t$ac.u = cpuact_t$act.u * cpuact_t$cpu.u
  return(cpuact_t)
}

#run for dimensions 5 and 10
cpuact5_avg = ac(5)
cpuact10_avg = ac(10)
set.seed(8)
```

```
for(i in 1:5){
  print(i)
  cpuact5_avg = cpuact5_avg + ac(5)
  cpuact10_avg = cpuact10_avg + ac(10)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
grid.table(round(cpuact5_avg/6, 2))
grid.table(round(cpuact10_avg/6, 2))
```

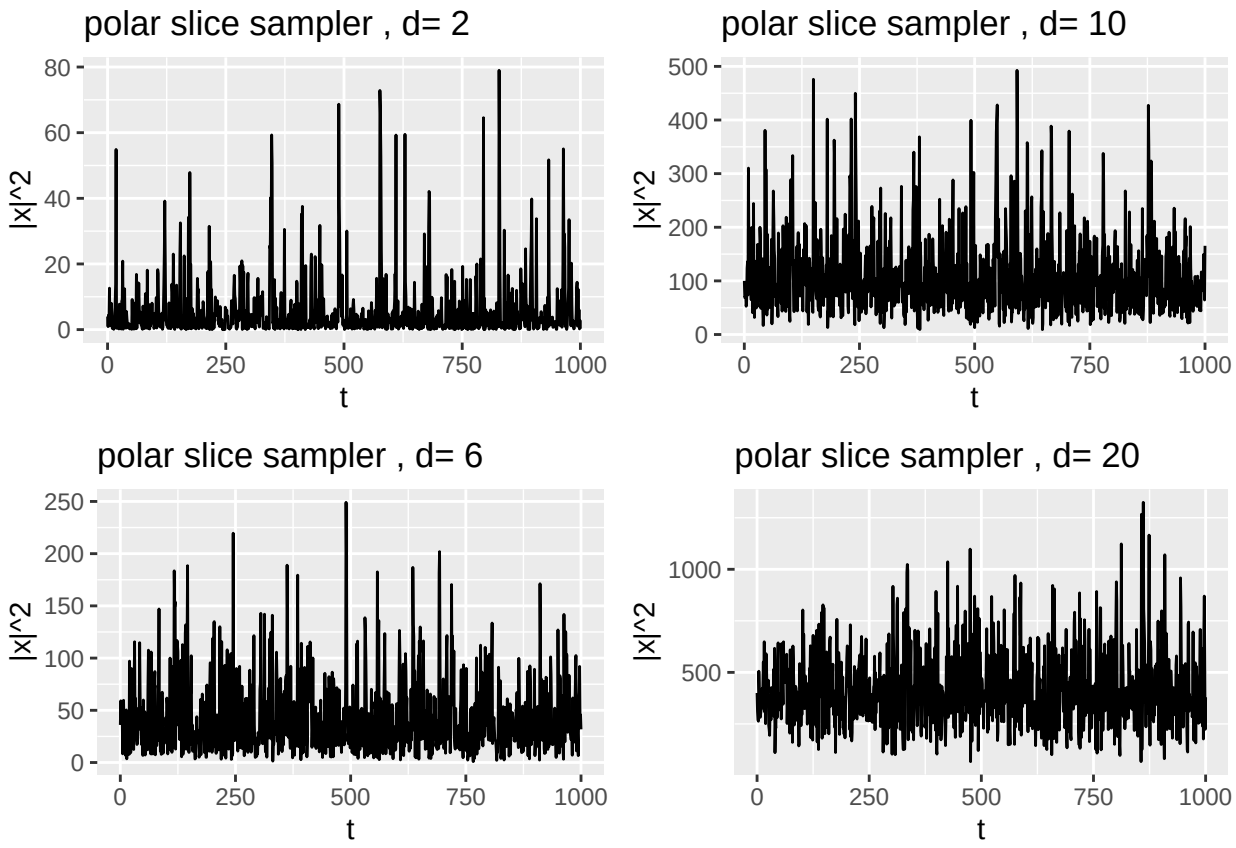
	act.p	act.u	cpu.p	cpu.u	ac.p	ac.u
0	1.1	6.06	0.96	0.64	1.06	3.88
0.25	1.03	4.72	1.13	0.64	1.16	3.03
0.5	1.02	4.54	7.85	0.66	9.16	3.02
0.75	1.22	4.23	5.89	0.65	7.27	2.76
1	2.04	3.43	16.65	0.64	35.38	2.18

Example 3

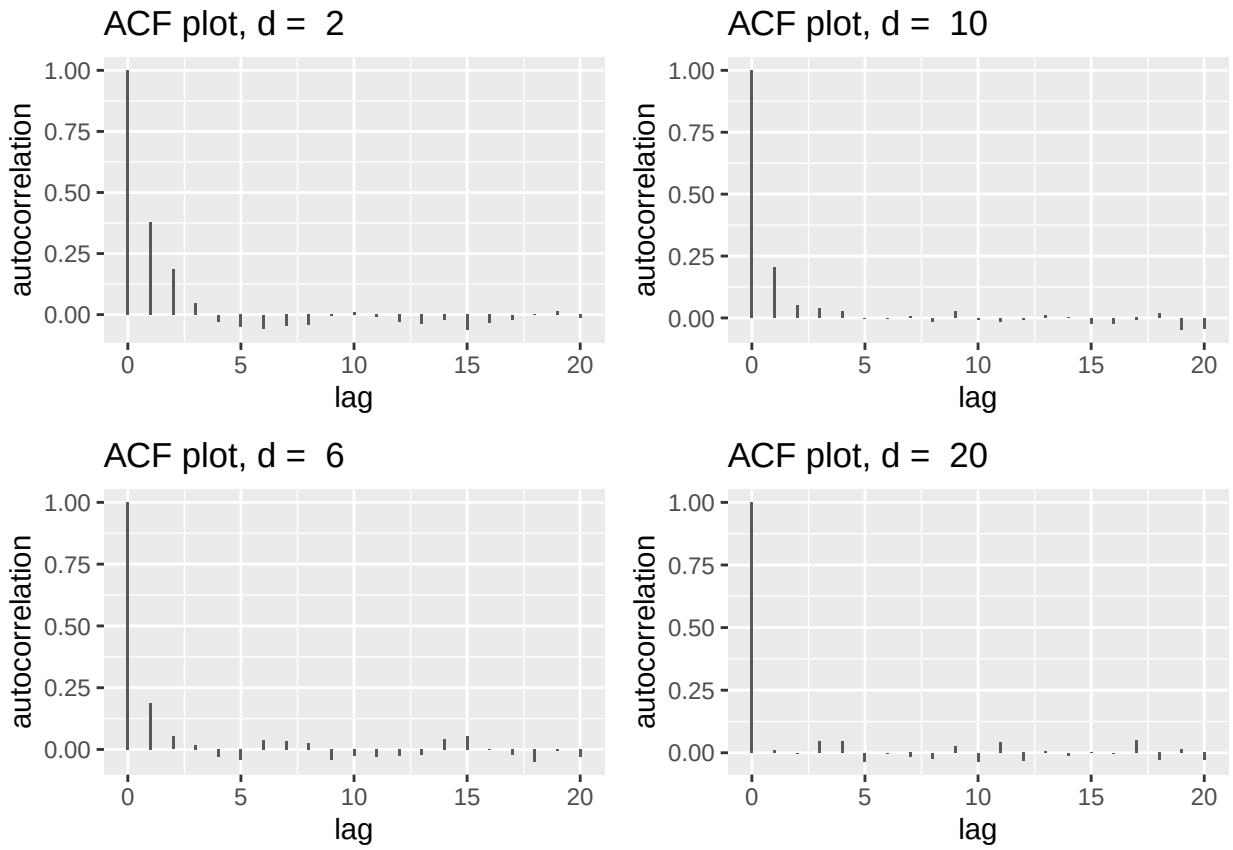
As another example, we will use $\pi_3(x) = e^{-|x|}]\mathbf{1}_{\mathbf{x_1},...,\mathbf{x_{\frac{d}{2}}} \geq \mathbf{0}}$. That is, the function $\pi(x)$ from our first example, restricted to the first $\frac{d}{2}$ components being positive. For simplicity, we will assume d to be odd. Clearly, this is a non-symmetric function. We will use the same r^* function.
The polar sampler still mixes very well, but we notice the number of rejected samples is significantly higher than the first example (which would be worse if increased chain length or dimension). In fact, the rate of increase is increasing so that for higher values of d, the computational power needed drastically increases. Of course, the chain still converges nicely. This shows it could be benfeicial to sample the 'angle' in our polar-rejection sample not completely at random. In this case, it would be easy to simply ensure a positive θ_i value for each $i = 1, ..., \frac{1}{2}$. But oftentimes, it is not so easy to sample r, θ_i in a more efficient manner. Marginal distribution plots are provided below to show that the first $\frac{d}{2}$ indices indeed need to be positive.

```
pi3 = function(x, filler){ #pi(x) = exp(-|x|)
  d = length(x)
  if(sum(x[1:(d/2)] >= 0) == d/2){
    return(exp(-1*norm(x)))
  }
  else{
    return(0)
  }
}

set.seed(8)
polar3 = sampling_plots(c(2,6, 10, 20), 1000, pi3, pi1_rstar, 'polar slice sampler', polar, pi1_init, 0
```

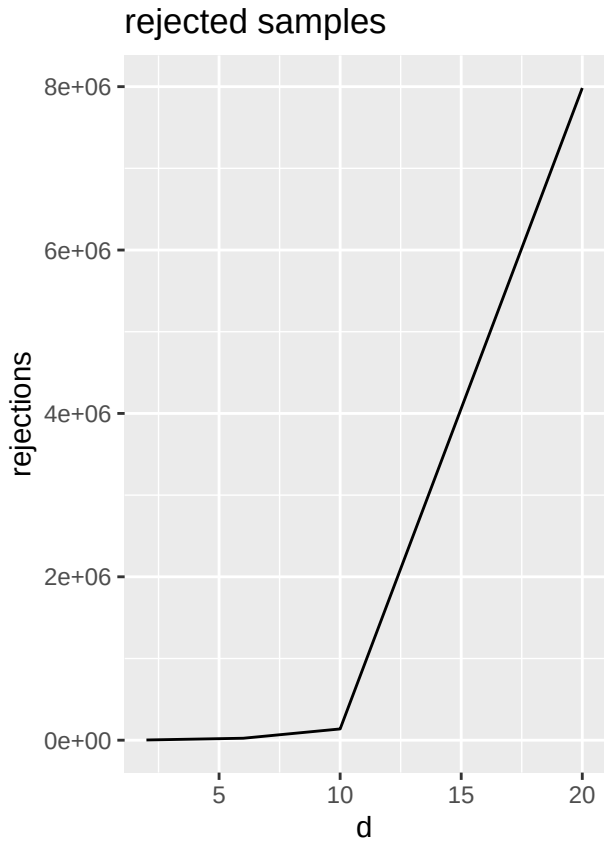
```
grid.arrange(grobs = polar3[[1]] , ncol = 2, as.table = FALSE)
```

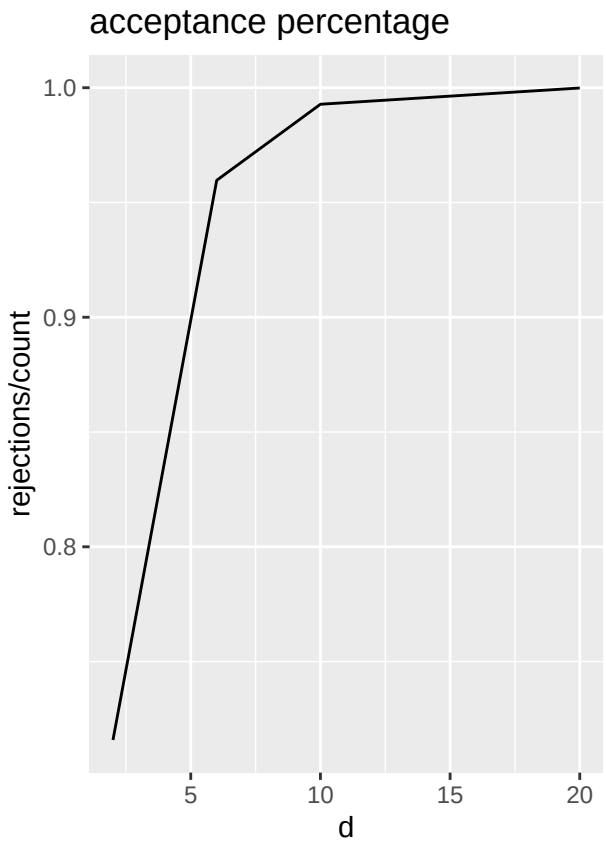



```
grid.arrange(grobs = polar3[[2]] , ncol = 2, as.table = FALSE)
```

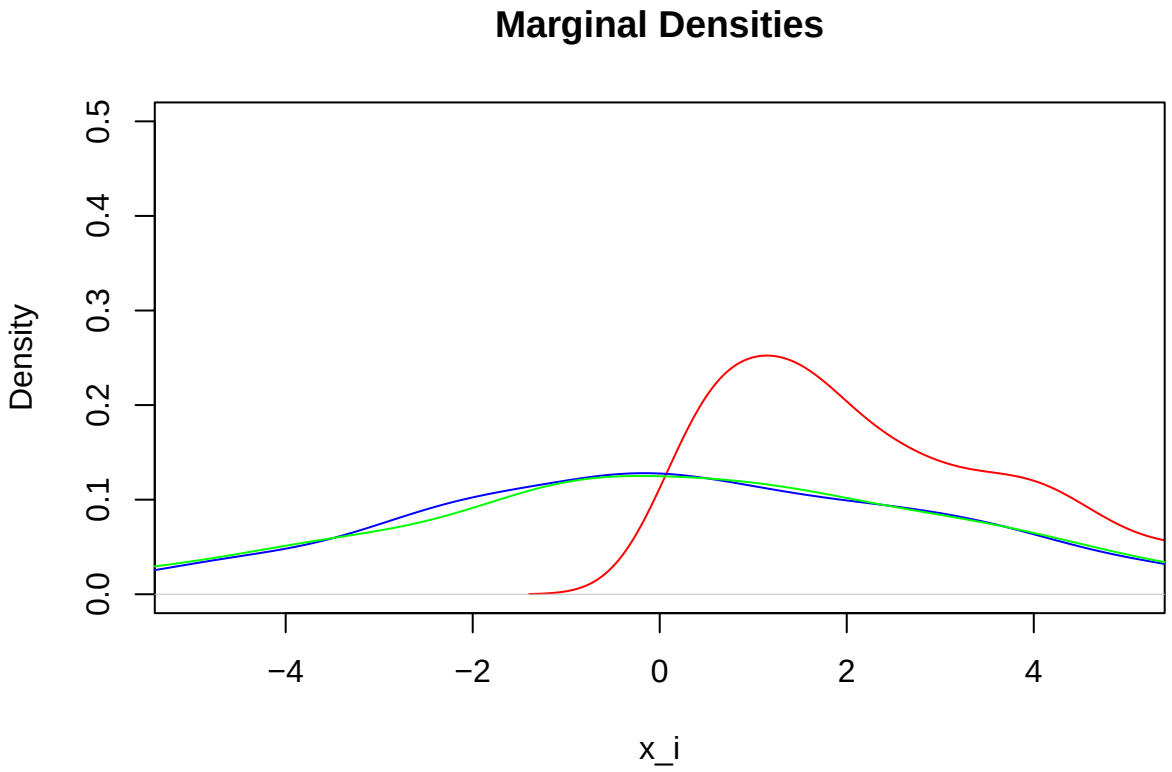



```
grid.arrange(grobs = list(polar3[[3]], polar3[[4]]) , ncol = 2, as.table = FALSE)
```






```r
#printing marginals
test = polar(rep(sqrt(10),10), pi3, 1000, pi1_rstar)

plot(density(test[[1]][,1]), xlim = c(-5, 5), ylim = c(0, 0.5), col = 'red',
     xlab = 'x_i', main='Marginal Densities')
lines(density(test[[1]][,10]),col = 'blue')
lines(density(test[[1]][,7]),col = 'green')
```




```
cpuact3 = data.frame(row.names = c(2,6,10,20))

set.seed(100)
for(i in c(2,6,10,20)){
  u = uniform_slice(rep(sqrt(i), i), pi3, 1000, pi1_rstar, pi1_lim)
  p = polar(rep(sqrt(i), i), pi3, 1000, pi1_rstar, 0)
  cpuact3[paste(i), 'act.p'] = sum(acf(rowSums(p[[1]]^2), plot=F)[[1]])
  cpuact3[paste(i), 'act.u'] = sum(acf(rowSums(u[[1]]^2), plot=F)[[1]])
  cpuact3[paste(i), 'cpu.p'] = p[[4]][['elapsed']]
  cpuact3[paste(i), 'cpu.u'] = u[[4]][['elapsed']]
}
cpuact3$ac.p = cpuact3$act.p * cpuact3$cpu.p
cpuact3$ac.u = cpuact3$act.u * cpuact3$cpu.u
```

```
cpuact3
```

```
##       act.p act.u  cpu.p cpu.u       ac.p ac.u
## 2  1.843903   NaN  0.274 0.114  0.5052294  NaN
## 6  1.077753   NaN  0.581 0.104  0.6261744  NaN
## 10 1.147657   NaN  1.322 0.109  1.5172023  NaN
## 20 1.152748   NaN 38.441 0.163 44.3127980  NaN
```

Example 4

The next goal is to run the polar slice sampler for real datasets. To do that, it is helpful to first create functions that allow us to work with empirical density functions. Here we write these functions in R and test it with toy examples (based on realizations of actual distributions).
Let $y_1, ..., y_n \in \mathbb{R}^d$ be observations. We wish to determine $f_1(x)$ for $x = (x_1, ..., x_d)^T$. We assume that the different features are independent so that the joint distribution can be calculated as the product of the

marginals. We consider each marginal $y_{1i}, ..., y_{ni} \in \mathbb{R}$ for $i = 1, ..., d$. Formally, the empirical density is $f(x) = \frac{1}{n}\sum_{j=1}^{n} \mathbf{1}_{y_{ji}=x}$. However, for computational ease, in our examples, we approximate the marginal with the function π_i that is the R built-in *density()* function applied to the observations $y_{1i}, ..., y_{ni} \in \mathbb{R}$. This results in a smooth interpolation on a grid $z_{i1}, ..., z_{ik}$ of k evenly-spaced points that covers the range of $y_{1i}, ..., y_{ni}$ (the default is $k = 512$). Then, we approximate $f_1(x_i) \approx \pi_i(z_{ij})$ where $i = argmin_{i=1,...,k}|x_i - z_{ij}|$. If the point x_i is not in the range of $z_{i1}, ..., z_{ik}$, i.e. $min_{i=1,...,k}|x_i - z_{ij}| > \delta_i$ for tolerance $\delta_i = 2(z_{i2} - z_{i1})$, then we return 0. We then compute

$$\pi_e(x) = \prod_{i=1}^{d} \pi_i(x_i)$$

We compute r^* based on our interpolated grid of points $z_j = (z_{1j}, ..., z_{dj}),\ j = 1, ..., k$

$$r^* = \max_{j=1,...,k}\{|z_j||z_j|^{d-1}\pi_e(z_j) \geq y\}$$

This is simple to compute as we can easily determine the values of $|z_j|^{d-1}\pi_e(z_j)$ on our grid.
In our first toy example, we simulate $y_i \in \mathbb{R}^4$ for $i = 1, ..., 1000$. The marginals are independent with $y_{i1} \sim N(0, 1), y_{i2} \sim N(2, 25), y_{i3} \sim U(-3, 3), y_{i4} \sim Exp(2)$. Below, we can see the histogram and approximated density of each marginal of the polar slice sampling chain (1000 iterations). Compared to the expected shape of these distributions, we see that the empirical normal distributions are best recovered. However, there are slight issues with the exponential and uniform empirical distributions around there boundaries since the uniform density is only non-zero on $(-3, 3)$ and the exponential density is only non-zero for positive values (part of the issue in the plots are the bin values). One issue we notice is the the number of rejections is around 12 million across the 10,000 length chain, which is a huge number and means this algorithm may not generalize well to more features and longer chains. This is due to the the range of an empirical distribution being finite, and in this case bounded as well. Thus when our proposed x_{prop} is outside the range for any margin, the value of $f_1(x_{prop}) = 0$ leading to us rejecting this point. Moreover, in this specific example, we also have the second and fourth margins being non-symmetrical, leading to immediate rejection of almost $\frac{3}{4}$ of proposed points (depending on quadrant).
We also output the chain of sample norms and the autocorrelation plot. We see that for this somewhat trivial example, the autocorrelation is low, so that the sample space is relatively well traversed, with some issues near the boundary.

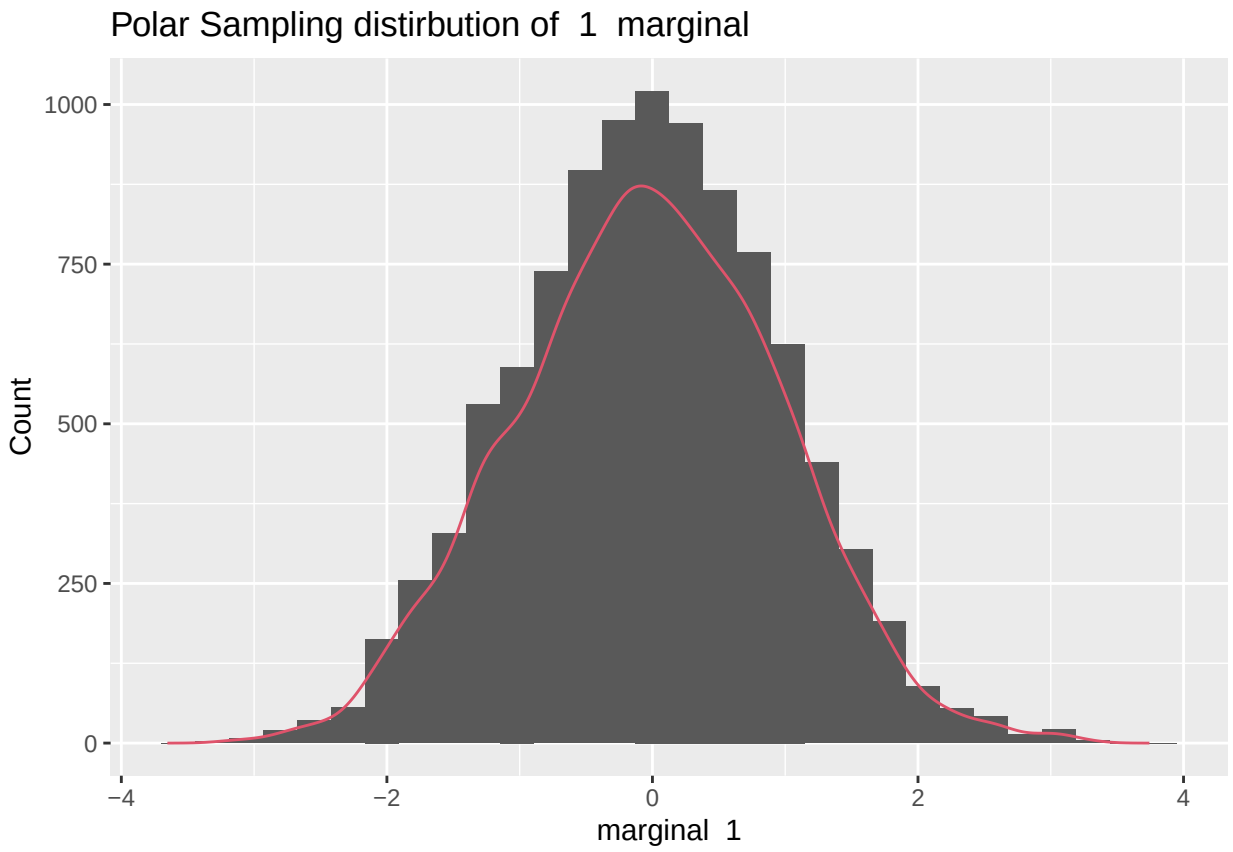
```
#empirical functions - trivial example
fn_empirical= function(x, d){
  #empirically compute the density for each x
  n = dim(d)[2]/2
  delta = (d[dim(d)[1],seq(1, 2*n-1, by=2)] - d[1,seq(1, 2*n-1, by=2)])/dim(d)[1]
  fx = 1
  for(i in 1:n){
    if(min(abs(d[,2*i - 1] - as.numeric(x[i]))) > delta[i]){
      return(0)
    }
    ind = which.min(abs(d[,2*i - 1] - as.numeric(x[i])))
    fx = fx * d[ind,2*i]
  }
  return(fx)
}

rstar_empirical = function(y, d, dens){
  xs = seq(1, 2*d-1, by=2)
  ys = seq(2, 2*d, by=2)
  dsq = rowSums(dens[,xs]^2)
  d2 = dsq^((d-1)/2) * apply(dens[,seq(2, 2*d, by=2)], 1, prod)
  g = which(d2>=y)
```

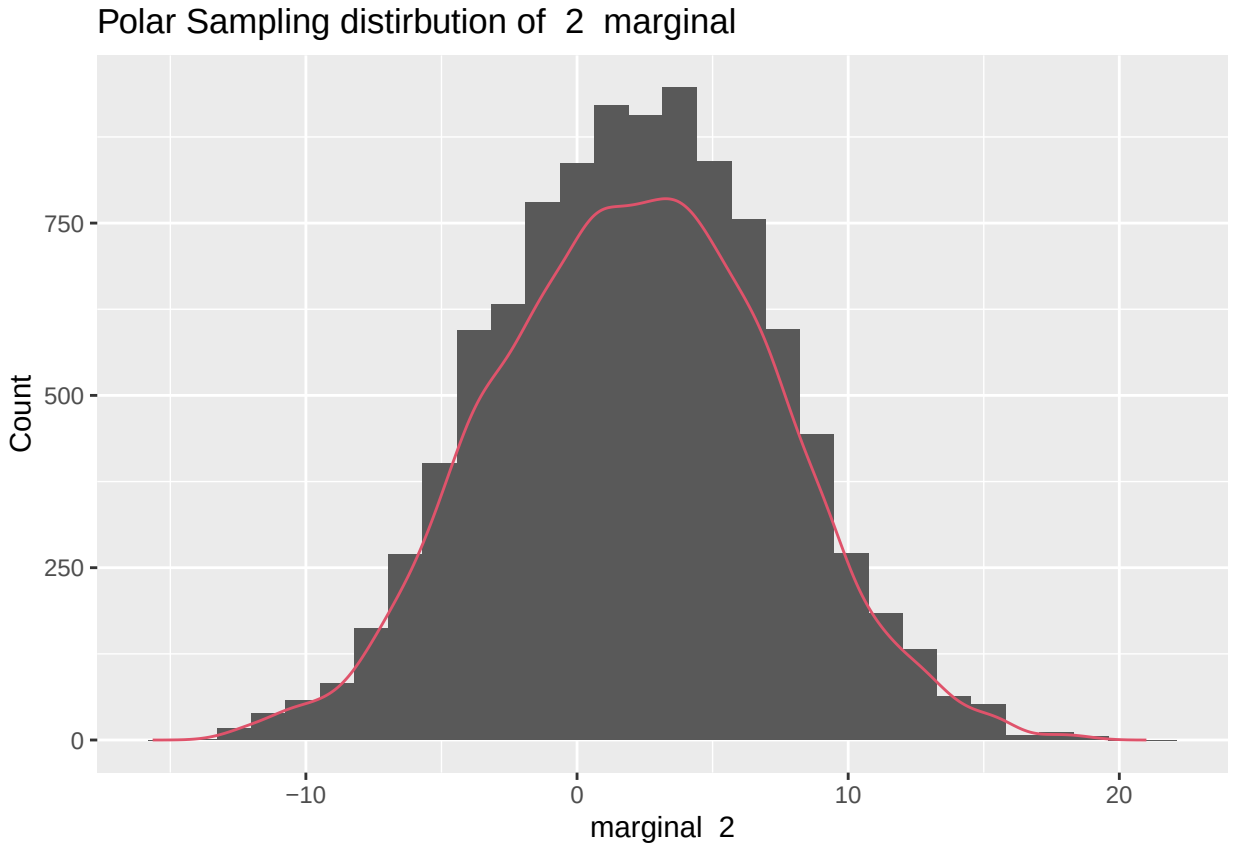
```
  return(ifelse(length(g) > 0, sqrt(max(dsq[g])), sqrt(max(dsq))))
}

set.seed(999)
n1 = density(rnorm(1000))
n2 = density(rnorm(1000, 2, 5))
n3 = density(runif(1000, -3, 3), from = -3, to = 3)
n4 = density(rexp(1000, 2), from = 0)
data = cbind(n1$x, n1$y, n2$x, n2$y, n3$x, n3$y, n4$x, n4$y)

ps = polar(c(0, 0, 0, 0.5), fn_empirical, 10000, rstar_empirical, data)
```

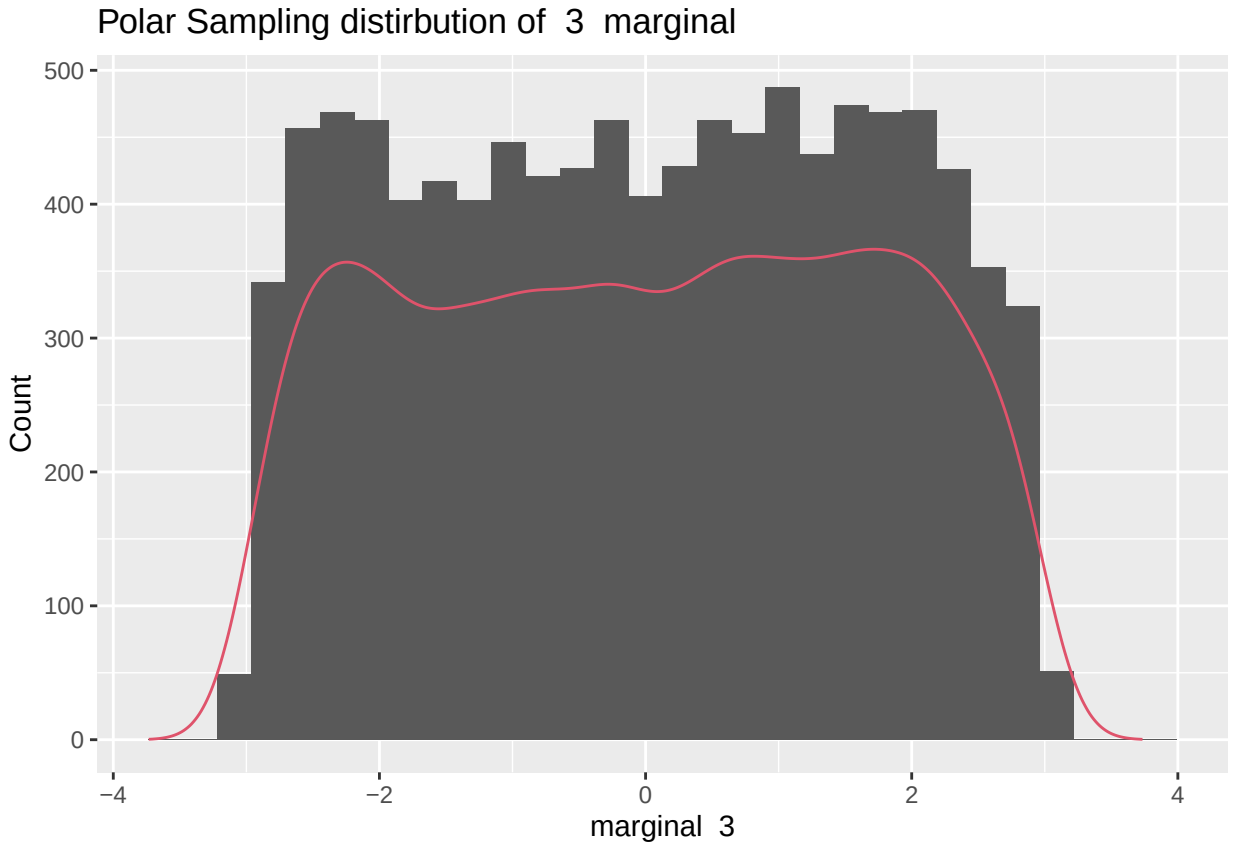
```
print(ps[[2]])
```

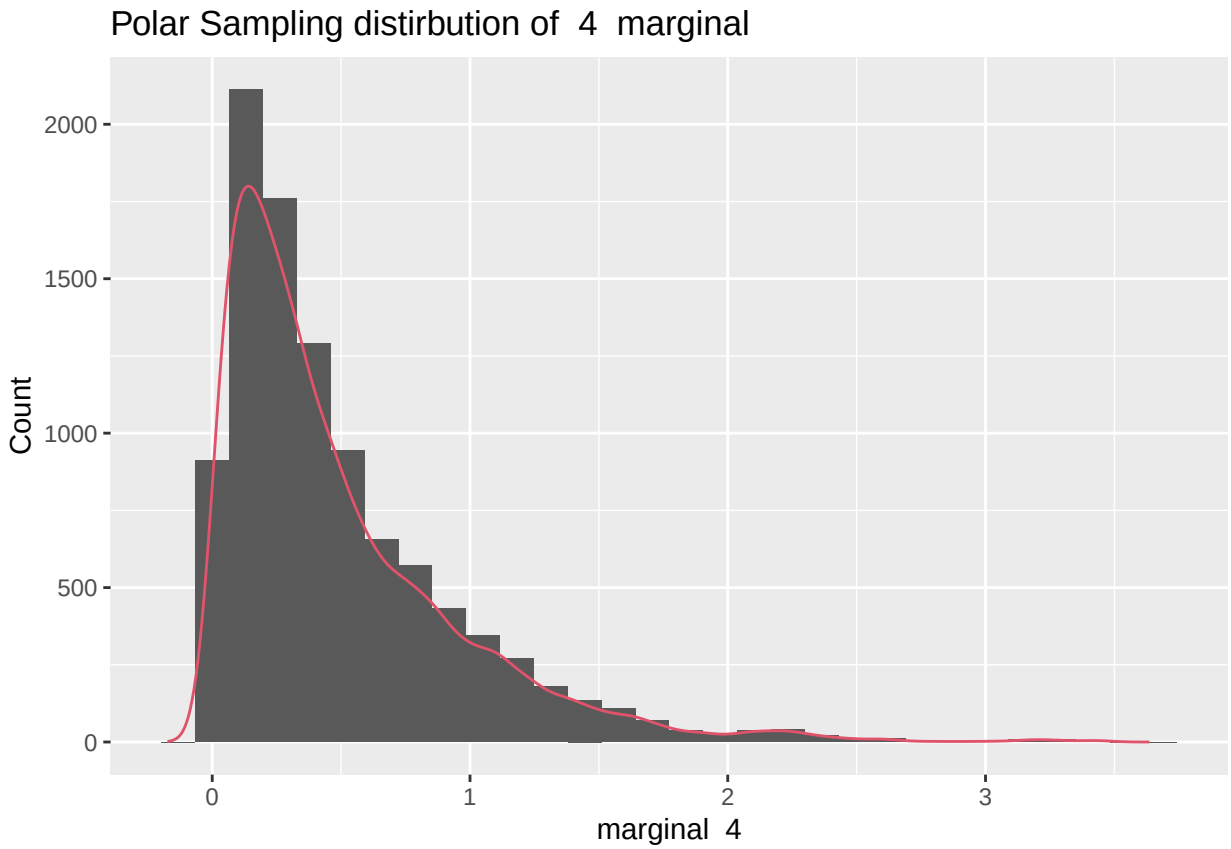
```
## [1] 12067300
```

```
for(i in 1:4){
  d = density(ps[[1]][,i])
  plt = ggplot(NULL, aes(x=ps[[1]][,i])) + geom_histogram(bins=30)  +
    geom_path(aes(x = d$x, y = d$y*(max(ps[[1]][,i]) - min(ps[[1]][,i])) / 30 * 10000), col=2) +
    labs(x = paste('marginal ', i), y = 'Count',
         title = paste('Polar Sampling distirbution of ', i, ' marginal'))
  print(plt)
}
```

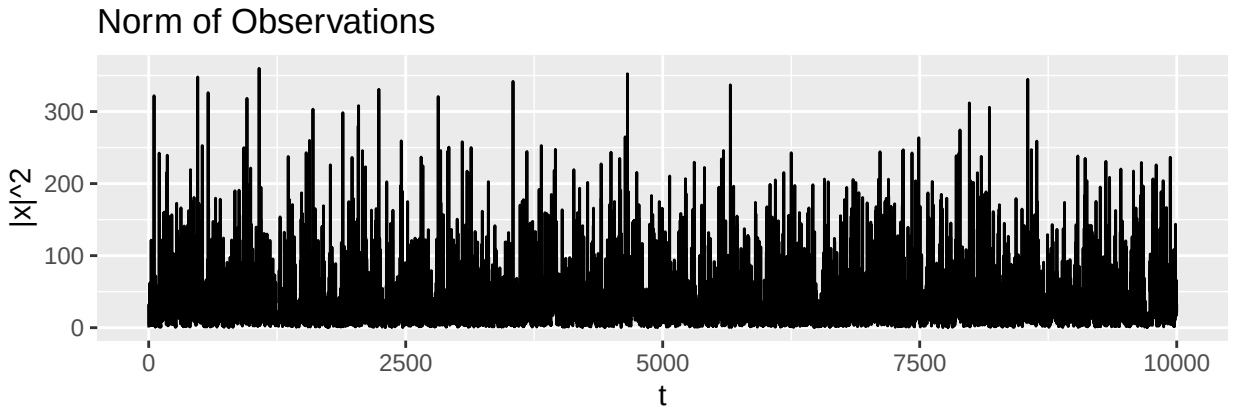


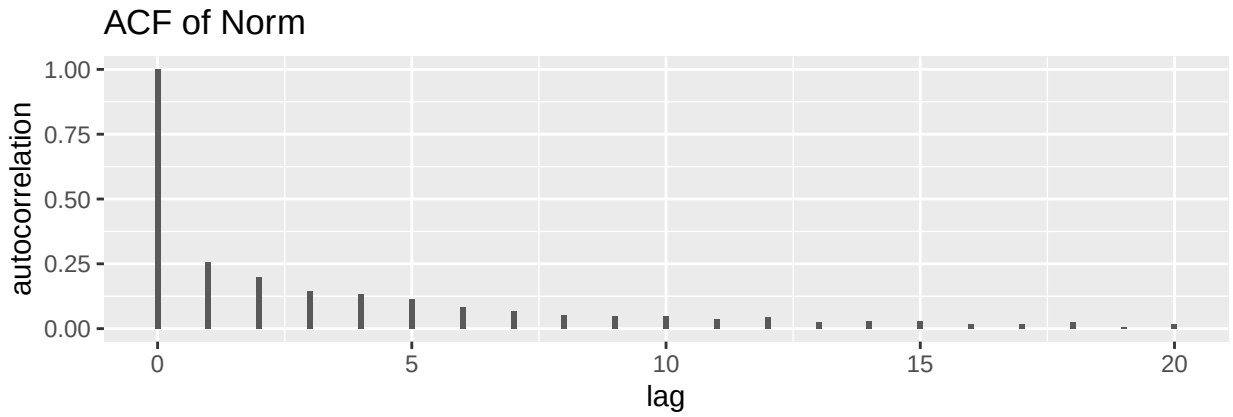

Polar Sampling distirbution of 2 marginal
Count
750
500
250
0
−10
0
10
20
marginal 2


Polar Sampling distirbution of 3 marginal
Count
500
400
300
200
100
0
−4
−2
0
2
4
marginal 3




```
plots = list()
np = rowSums(ps[[1]]^2)
plots[[1]] = ggplot(NULL, aes(x=seq(0,10000, by=1), y = np)) +
  geom_path() + labs(x = 't', y = '|x|^2',
                     title = 'Norm of Observations')
plots[[2]] = ggplot(NULL, aes(x=seq(0, 20, by=1),
                              y = acf(np, plot=F, lag.max=20)$acf)) +
  geom_bar(stat = "identity", width=0.1) +
  labs(x = 'lag', y = 'autocorrelation',
       title = 'ACF of Norm' )

grid.arrange(grobs =plots, ncol = 1, as.table = FALSE) ## display plot
```





Example 5

In this example, we run the algorithm for polar slice sample for empirical distributions, using a real dataset. The dataset of choice contains NBA stats from the last 25 years. Here we consider player age, height, weight, points, rebounds, assists and true shooting percentage. Note that there are repeated measurements per player, but for our purposes that's okay. Before sampling, we first standardize the features so that they are centred around 0, which will allow the polar slice sampler to reject less.

```
real_data = read.csv('all_seasons.csv')
```

```
labels = c('player_name', 'gp', 'net_rating', 'pts', 'reb', 'ast', 'ts_pct')
names = unique(real_data$player_name)
names_vet = c()
for(i in names){
  if(length(real_data[real_data$player_name == i,'player_name'])>= 8){
    names_vet = c(names_vet, i)
  }
}
#cutoff for player average is 8 to ease computation
data_vets = real_data[real_data$player_name %in% names_vet,labels]
points_vets = real_data[real_data$player_name %in% names_vet,c('player_name', 'pts')]
points_vets = points_vets[order(points_vets$player_name),]
points_vets$pts = points_vets$pts/10 #scale down point values
```

We consider a random effects model for average points scored in a season. For player $i = 1, ..., 645$, let J_i be the number of seasons they played (so far). Then we have

$$Y_{ij} \sim N(\theta_i, \sigma^2) \quad j = 1, ..., J_i, \quad \theta_i \sim N(\mu, \tau^2)$$

That is, each player has mean points θ_i and the player mean points are distributed around the global mean μ. So our parameter of interest is $\theta = (\theta_1, ..., \theta_n, \mu, \sigma^2, \tau^2)$ for $n = 645$. Let the prior distributions be inverse gamma

$$\sigma^2 \sim IG(5, 0.5) \quad \tau^2 \sim IG(5, 1) \quad \mu \sim IG(6, 7)$$

The choice of inverse gamma distributions is good because we expect all the quantities to be right skewed and in the range $[0, \infty)$. Moreover, we know that for $Z \sim IG(a, b)$, we have

$$E(Z) = \frac{b}{a-1}, a > 1 \quad Var(Z) = \frac{b^2}{(a-1)^2(a-2)} \quad Mode(Z) = \frac{b}{a+1}$$

For average points across all players, μ, we naively want the mean/mode to be around 1 with standard deviation 0.5 since the average scoring output per team is around 100 points per game, playing a rotation of around 10 players. Accordingly, we expect standard deviation between players to have mean 0.5 with small variance. Standard deviation within players should have mean 0.3 since player output is generally consistent with swings due to longevity. Note that the parameters are independent and so we can simplify the Bayesian model for to estimate the parameters

$$\begin{aligned}
\pi(\theta|Y) &\propto \mathcal{L}(Y|\theta)\pi(\theta) \\
&= \pi(\mu)\pi(\tau^2)\pi(\sigma^2)\left(\prod_{i=1}^{645}\pi(\theta_i|\mu,\tau^2)\right)\left(\prod_{i=1}^{645}\prod_{j=1}^{J_i} f(y_{ij}|\theta)\right) \\
&= \left(\frac{7^6}{\Gamma(6)}e^{\frac{-7}{\mu}}\mu^{-6-1}\right)\left(\frac{1^5}{\Gamma(5)}e^{\frac{-1}{\tau^2}}(\tau^2)^{-5-1}\right)\left(\frac{0.5^5}{\Gamma(5)}e^{\frac{-0.5}{\sigma^2}}(\sigma^2)^{-5-1}\right) \\
&\quad \left(\prod_{i=1}^{645}\frac{1}{\sqrt{2\pi\tau^2}}e^{-\frac{1}{2\tau^2}(\theta_i-\mu)^2}\right)\left(\prod_{i=1}^{645}\prod_{j=1}^{J_i}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2\sigma^2}(\theta_i-Y_{ij})^2}\right) \\
\log\pi(\theta|Y) &\propto -7\log\mu - (6+\frac{645}{2})\log\tau^2 - (6+\frac{1}{2}\sum_{i=1}^{645}J_i)\log\sigma^2 - \frac{7}{\mu} - \frac{1}{\tau^2} - \frac{1}{2\sigma^2} \\
&\quad - \frac{1}{2\tau^2}\sum_{i=1}^{645}(\theta_i-\mu)^2 - \frac{1}{2\sigma^2}\sum_{i=1}^{645}\sum_{j=1}^{J_i}(\theta_i - Y_{ij})^2
\end{aligned}$$

```
#theta[1] = mu, theta[2] = tau^2, theta[3] = sigma^2, theta[4] onwards theta_i alphabetically
mean_pts = points_vets %>% group_by(player_name) %>% summarise(avg = mean(pts))
theta = c(1, 0.1, 0.25, mean_pts$avg)
#pi_b(theta, points_vets)
#rstar_b(1, dim(points_vets)[1], 0)
ps = points_vets %>% group_by(player_name) %>% summarise(meany = mean(pts), vary = var(pts))
var(as.matrix(ps['meany']))
```

```
##           meany
## meany 0.2275582
```

```
var(as.matrix(points_vets['pts']))
```

```
##           [,1]
## [1,] 0.3836063
```

$$
\begin{aligned}
\pi(\theta|Y) &\propto \mathcal{L}(Y|\theta)\pi(\theta) \\
&= \pi(\mu)\left(\prod_{i=1}^{645}\pi(\theta_i|\mu)\right)\left(\prod_{i=1}^{645}\prod_{j=1}^{J_i} f(y_{ij}|\theta)\right) \\
&= \left(\frac{7^6}{\Gamma(6)}e^{\frac{-7}{\mu}}\mu^{-6-1}\right)\left(\prod_{i=1}^{645}\frac{1}{\sqrt{2\pi\tau^2}}e^{-\frac{1}{2\tau^2}(\theta_i-\mu)^2}\right)\left(\prod_{i=1}^{645}\prod_{j=1}^{J_i}\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2\sigma^2}(\theta_i-Y_{ij})^2}\right) \\
\log\pi(\theta|Y) &\propto -7\log\mu - \frac{645}{2}\log\tau^2 - \frac{1}{2}\sum_{i=1}^{645}J_i\log\sigma^2 - \frac{7}{\mu} - \frac{1}{2\tau^2}\sum_{i=1}^{645}(\theta_i-\mu)^2 - \frac{1}{2\sigma^2}\sum_{i=1}^{645}\sum_{j=1}^{J_i}(\theta_i-Y_{ij})^2 \\
&\leq \frac{7}{\mu} - \frac{d-1}{2}\log\tau^2 - \frac{n}{2}\log\sigma^2 - \frac{7}{\mu} - \frac{n+d-1}{2}\log 2\pi - \frac{1}{2\sigma^2}\sum(Y_{ij}-\bar{Y}_i)^2 \\
&\leq -\frac{d-1}{2}\log\tau^2 - \frac{n}{2}\log\sigma^2 - \frac{1}{2\sigma^2}\sum(Y_{ij}-\bar{Y}_i)^2 \\
\log y &\leq (d-1)\log|\theta| + \log\pi(\theta|Y) \\
\log|\theta| &\geq \frac{\log y + \frac{d-1}{2}\log\tau^2 + \frac{n}{2}\log\sigma^2}{d-1}
\end{aligned}
$$

```
#implement functions from above
pi_b2 = function(theta, data){
  d = length(theta)
  n = dim(data)[1]
  t = theta[2:d]
  xnorm = sqrt(sum(theta^2))

  y_sum = data %>% group_by(player_name) %>% summarise(y = sum(pts), ysq = sum(pts^2), count = n(), avg
  tau2 = as.numeric(var(as.matrix(y_sum[,'avg'])))
  sigma2 = as.numeric(var(as.matrix(data[,'pts'])))
  const = log(theta[1]^(-7) * exp(-7/theta[1]) * 7^6/ gamma(6)) - d/2*log(2*pi*tau2) - n/2*log(2*pi*sig

  ex1 = -(t - theta[1])^2/2/tau2
  ex2 = -1/2/sigma2*(pull(y_sum, 'count')*t^2 - 2*pull(y_sum, 'y')*t + pull(y_sum, 'ysq'))
  ex = exp(ex1 + ex2)

# tot = 1
# for(i in 1:(d-1)){
#   tot = tot*ex[i]*xnorm
#   print(paste(i, tot))
# }
  print(paste(const, 'a'))
  print(paste(sum(ex1), 'b'))
  print(paste(sum(ex2), 'c'))
  return(const + sum(ex1) + sum(ex2))
  #return(prod(ex*xnorm)*const)
}

rstar_b2 = function(y, d, data){
  ps = data %>% group_by(player_name) %>% summarise(avg = mean(pts))
  tau2 = as.numeric(var(as.matrix(ps[,'avg'])))
  sigma2 = as.numeric(var(as.matrix(data[,'pts'])))
  min_y = left_join(data, ps, by='player_name') %>% group_by(player_name) %>% summarise(res = sum((pts-a
```

```
  n = dim(data)[1]

  #r = (log(y) + sum(min_y[,'res'])/sigma2/2)/(d-1)
  print(n)
  print((n+d-1)/2*log(2*pi))
  #r = (log(y) + (d-1)/2*log(tau2) + n/2*log(sigma2) + (n+d-1)/2*log(2*pi) + sum(min_y[,'res']))/(d-1)
  r = (log(y) + (d-1)/2*log(tau2) + n/2*log(sigma2) + (n+d-1)/2*log(2*pi))/(d-1)
  return(exp(r))
}
```

```
#beta = 40, alpha = 5
#theta[1] = mu, theta[4] onwards theta_i alphabetically

mu = mean(points_vets$pts)
theta = c(mu, mean_pts$avg)

pi_b2(theta, points_vets)
```

```
## [1] "-3315.20311747989 a"
## [1] "-324.321213676706 b"
## [1] "-1355.06666637972 c"
```

```
## [1] -4994.591
```

```
#modified polar function
polar2 = function(init, fn, n, fn_rstar, emp_dens){
  d = length(init) #dimension from length of inital point
  chain = data.frame(matrix(data=0, nrow = n+1, ncol = d))
  chain[1,] = init
  rejections = 0
  count = 0
  for(i in 1:n){
    print(i)
    x = chain[i,] #current chain value

    y = runif(1, 0, exp(fn(as.vector(t(x)), emp_dens) + (d-1)/2*log(sum(x^2)))) #sample y uniformly
    print(paste('y', fn(as.vector(t(x)), emp_dens) + (d-1)/2*log(sum(x^2)), y))
    r_star = fn_rstar(y, d, emp_dens)
    test = rep(r_star/sqrt(d), times = d)

    print(paste(fn(test, emp_dens) + (d-1)/2*log(sum(test^2)), log(y)))
    accept = 0
    while(accept == 0){ #iterate until a new point is accepted
      #here, we use fact pi(x) is completely defined by norm
      count = count + 1
      r = runif(1, 0, r_star) #sample r and theta

      angle = rnorm(d)
      x_prop = abs(angle)/sqrt(sum(angle^2))*r
      print(paste(r, r_star, fn(x_prop, emp_dens) + (d-1)/2*log(sum(x_prop^2)), fn(x_prop, emp_dens), l
      #print(x_prop)
      if(fn(x_prop, emp_dens) + (d-1)/2*log(sum(x_prop^2)) >= log(y)){ #check if it satisfied bound
        accept = 1
        chain[i+1,] = x_prop
      }
```

```
      else{
        rejections = rejections + 1
      }

    }
  }
  return(list(chain, rejections, count))
}
```

```
#polar2(theta, pi_b2, 3, rstar_b2, points_vets)
```