Elementary volesti tutorial

Vissarion Fisikopoulos

1 February 2025

Abstract

This tutorial has been designed for the presentation at FOSDEM 2025.

Contents

The volesti package

volesti is a C++ package (with an R interface) for sampling from high dimensional distributions and computing estimations of volume of polytopes given by a set of points or linear inequalities or Minkowski sum of segments (zonotopes).

There are various algorithms for volume estimation, sampling and rounding polytopes.

We can download the R package from https://CRAN.R-project.org/package=volesti or build the development version from https://github.com/GeomScale/Rvolesti

A gentle intro to R interface of volesti (version `1.1.2-2`) is https://journal.r-project.org/archive/2021/RJ-2021-077/RJ-2021-077.pdf

In this tutorial we are using volesti `1.2.0`

```
# first load the volesti library
#install.packages('volesti')
library(volesti)
```

```
## Loading required package: Rcpp
```

```
packageVersion("volesti")
```

```
## [1] '1.2.0'
```

You have access to the documentation of volesti functions like volume computation and sampling.

```
help("volume")
help("sample_points")
```

Full documentation for the CRAN version here: https://cran.r-project.org/web/packages/volesti/volesti.pdf

Let's try our first volesti command to estimate the volume of a 3-dimensional cube.

```
P <- gen_cube(4,'H')
print(volume(P))
```

```
## $log_volume
## [1] 2.825736
##
## $volume
## [1] 16.87336
```

What is the exact volume of P? Did we obtain a good estimation?

Sampling

Sampling uniformly in the square.

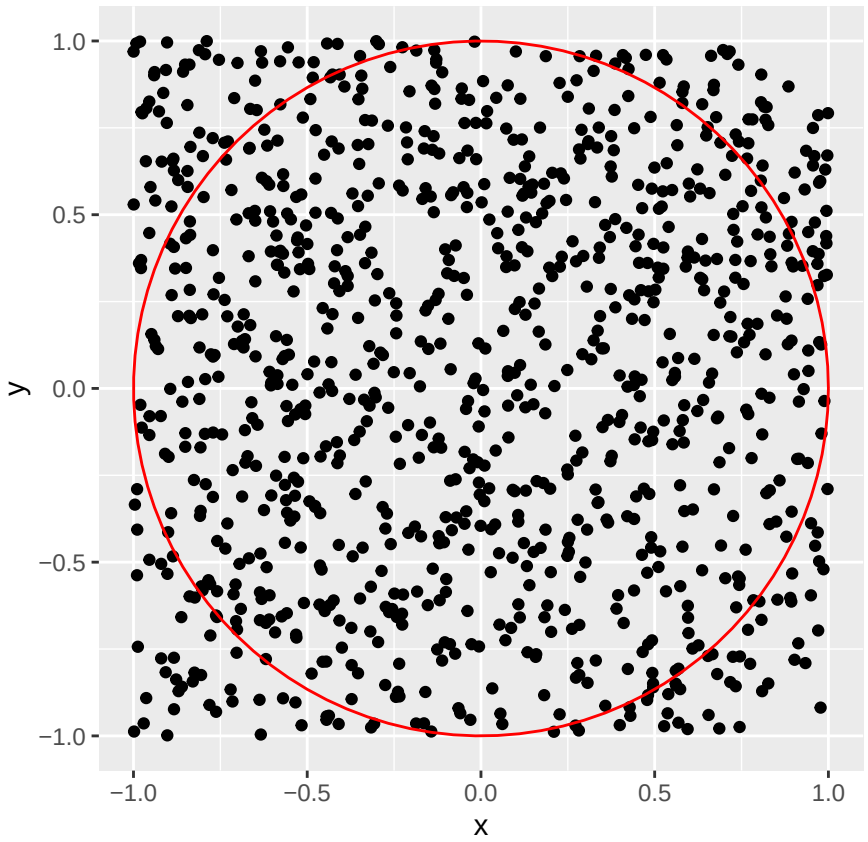
```
library(ggplot2)

x1<-runif(1000, min = -1, max = 1)
x2<-runif(1000, min = -1, max = 1)

g<-ggplot(data.frame( x=x1, y=x2 )) + geom_point( aes(x=x, y=y))

g<-g+annotate("path",
   x=cos(seq(0,2*pi,length.out=100)),
   y=sin(seq(0,2*pi,length.out=100)),color="red")+coord_fixed()

plot(g)
```



Can we estimate the volume of the red ball via sampling? Solution: rejection sampling. The following computation illustrates that this will fail in (not so) high dimensions.

```
for (d in 2:20) {
  num_of_points <- 10000
  count_inside <- 0

  points1 <- matrix(nrow=d, ncol=num_of_points)
  for (i in 1:d) {
    x <- runif(num_of_points, min = -1, max = 1)
    for (j in 1:num_of_points) {
      points1[i,j] <- x[j]
    }
  }

  for (i in 1:num_of_points) {
    if (norm(points1[,i], type="2") < 1) {
      count_inside <- count_inside + 1
    }
  }
  vol_estimation <- count_inside*2^d/num_of_points
  vol_exact <- pi^(d/2)/gamma(d/2+1)

  cat(d, vol_estimation, vol_exact, abs(vol_estimation- vol_exact)/
vol_exact, "\n")
}
```

```
## 2 3.1228 3.141593 0.005981887
```

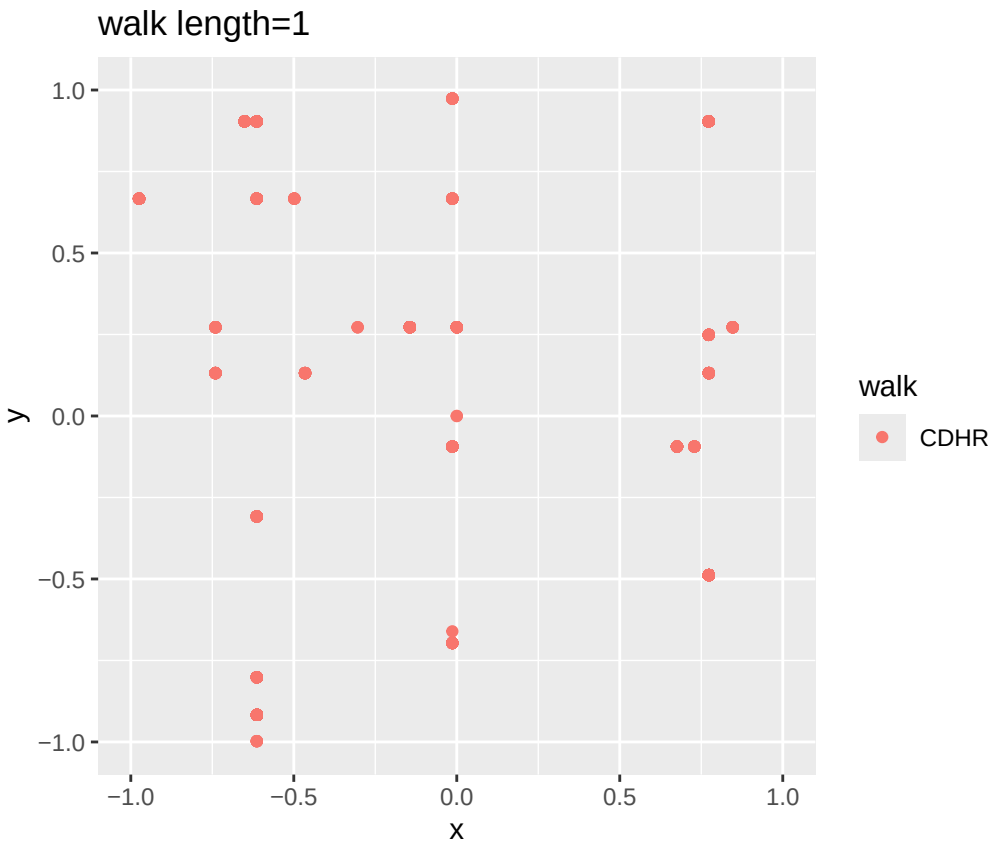
```
## 3 4.1848 4.18879 0.0009525912
## 4 4.8944 4.934802 0.008187198
## 5 5.2032 5.263789 0.01151053
## 6 5.0496 5.167713 0.02285591
## 7 4.224 4.724766 0.1059875
## 8 3.9424 4.058712 0.0286574
## 9 3.1744 3.298509 0.03762576
## 10 2.3552 2.550164 0.07645157
## 11 1.8432 1.884104 0.02170999
## 12 0.8192 1.335263 0.3864878
## 13 0 0.9106288 1
## 14 0 0.5992645 1
## 15 0 0.3814433 1
## 16 0 0.2353306 1
## 17 0 0.1409811 1
## 18 0 0.08214589 1
## 19 0 0.0466216 1
## 20 0 0.02580689 1
```

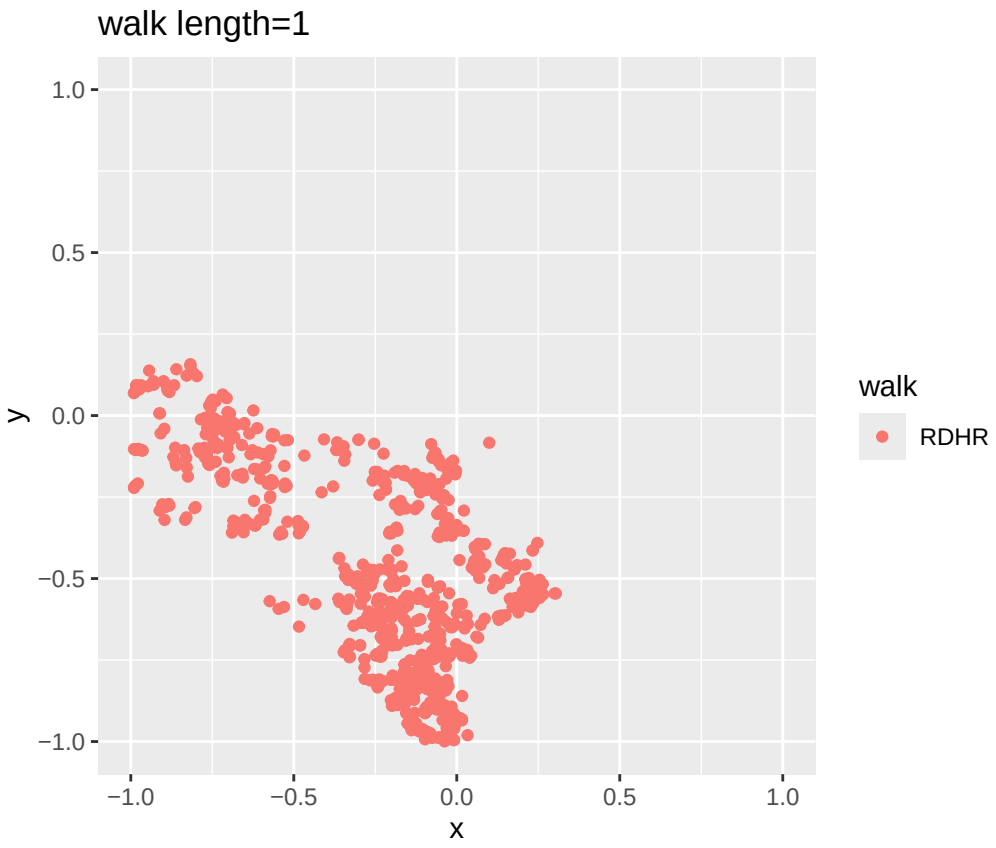
Sampling via random walks

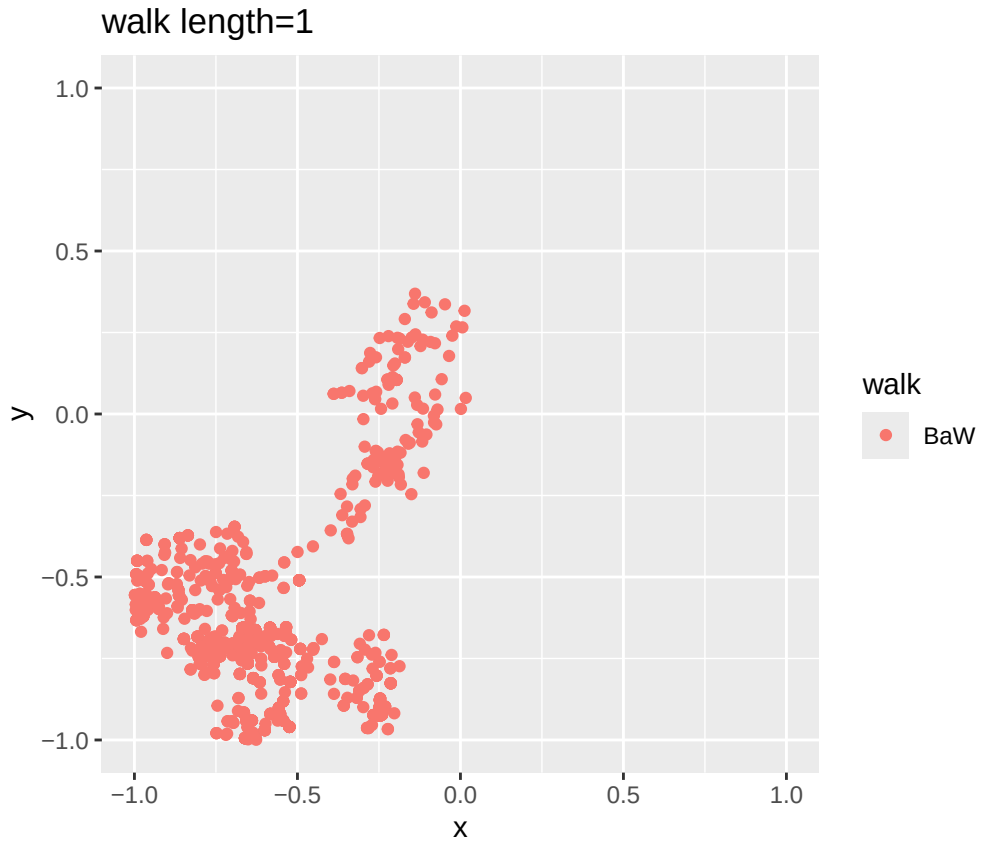
volesti supports various types of random walks

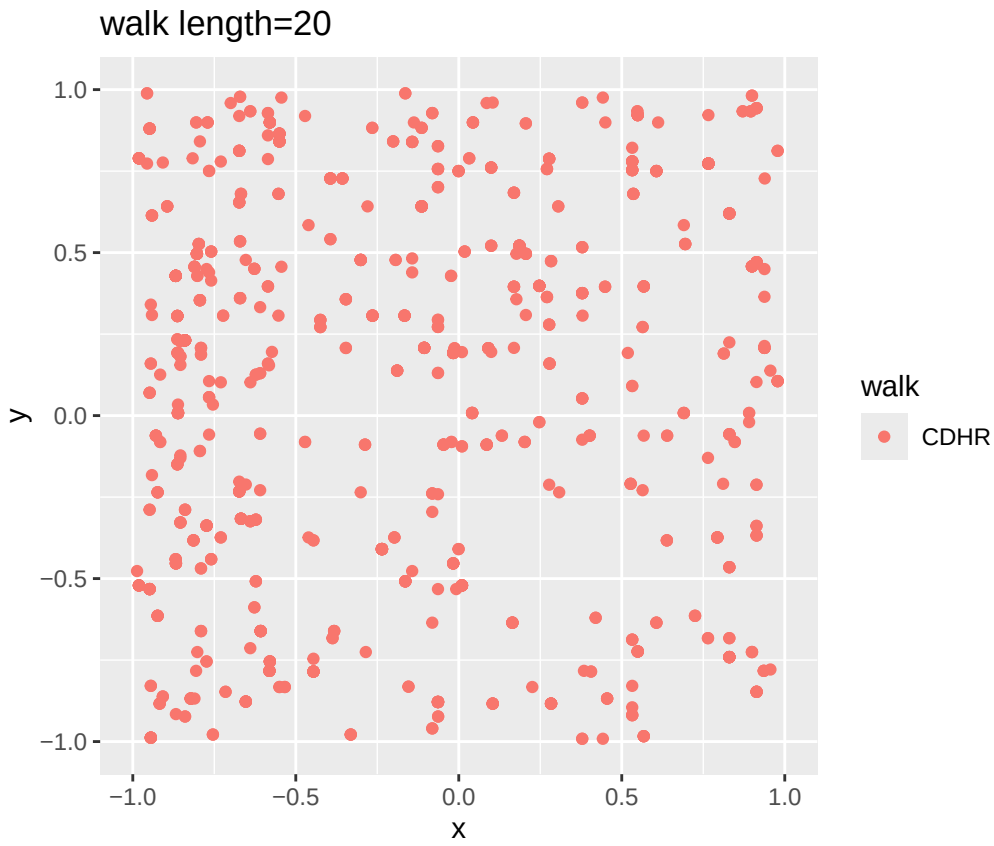
There are two important parameters cost per step and mixing time that affects the accuracy and performance of the walks. Below we illustrate this by choosing different walk steps for each walk while sampling on the 100-dimensional cube.

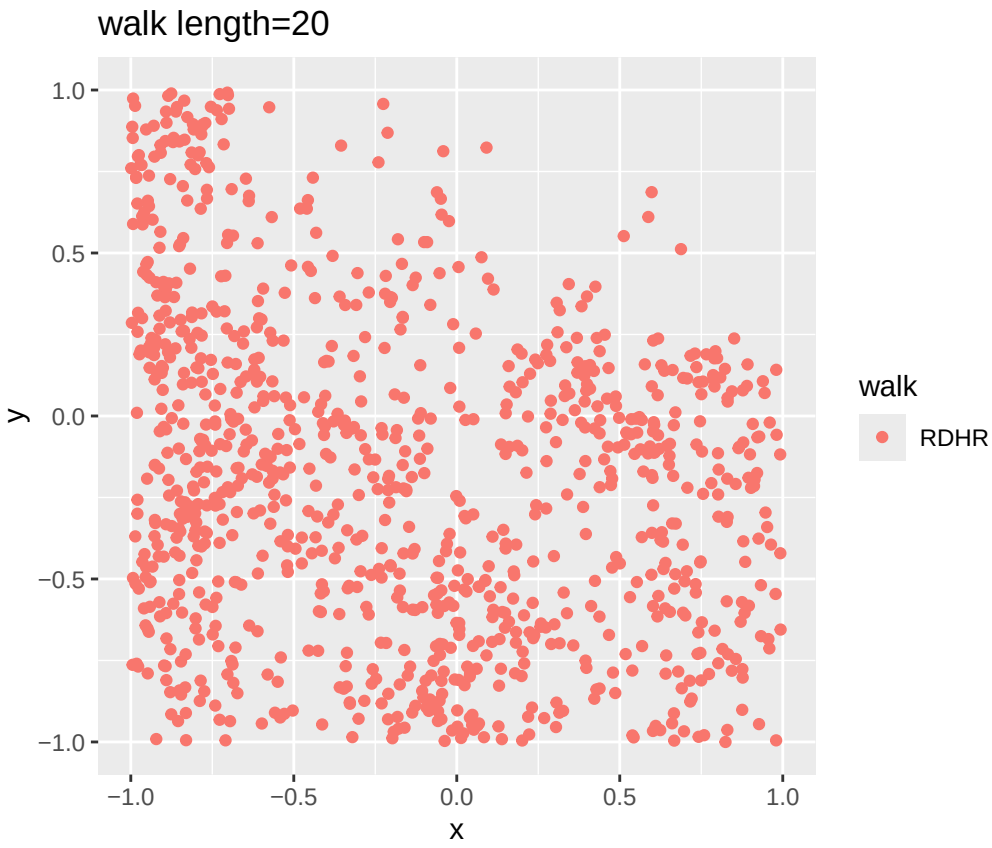
```
knitr::opts_chunk$set(echo = TRUE, fig.align = "center")
#run in few secs
library(ggplot2)
library(volesti)
for (step in c(1,20,100,150)){
  for (walk in c("CDHR", "RDHR", "BaW")){
    P <- gen_cube(100, 'H')
    points1 = sample_points(P, n = 1000, random_walk = list("walk" = walk, "walk_length" = step))
    g<-plot(ggplot(data.frame( x=points1[1,], y=points1[2,] )) +
geom_point( aes(x=x, y=y, color=walk)) + coord_fixed(xlim = c(-1,1),
ylim = c(-1,1)) + ggtitle(sprintf("walk length=%s", step)))
  }
}
```

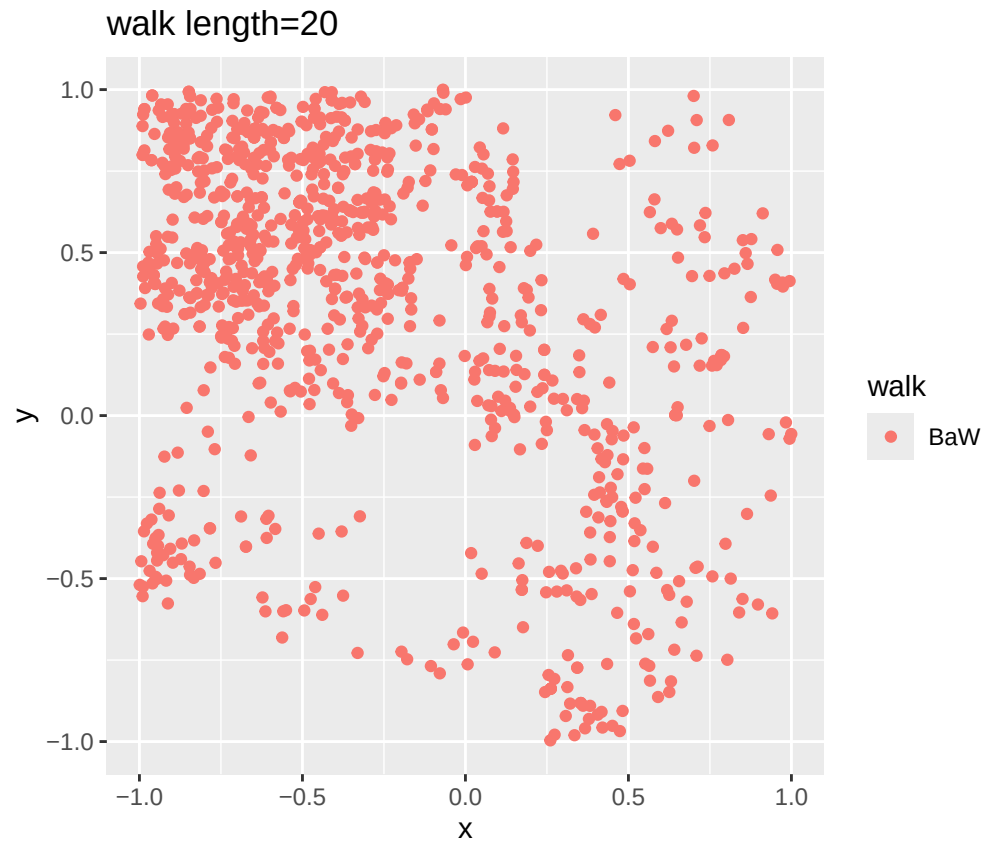

walk length=1
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
CDHR

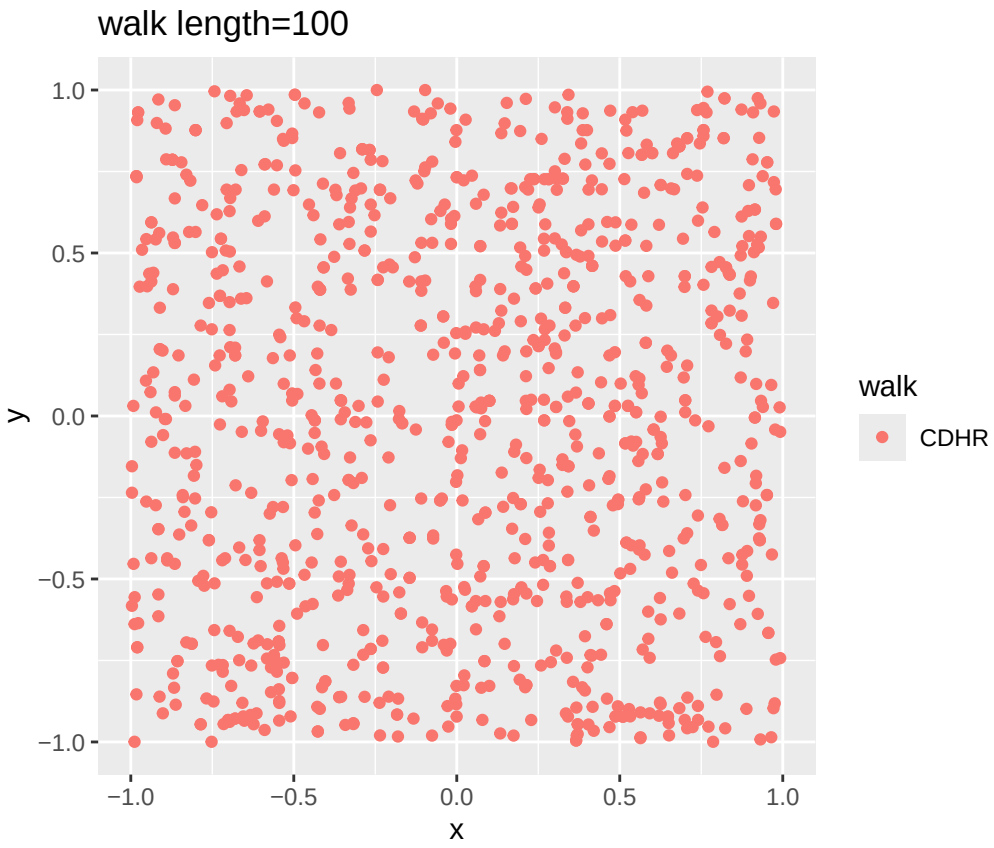

walk length=1
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
RDHR

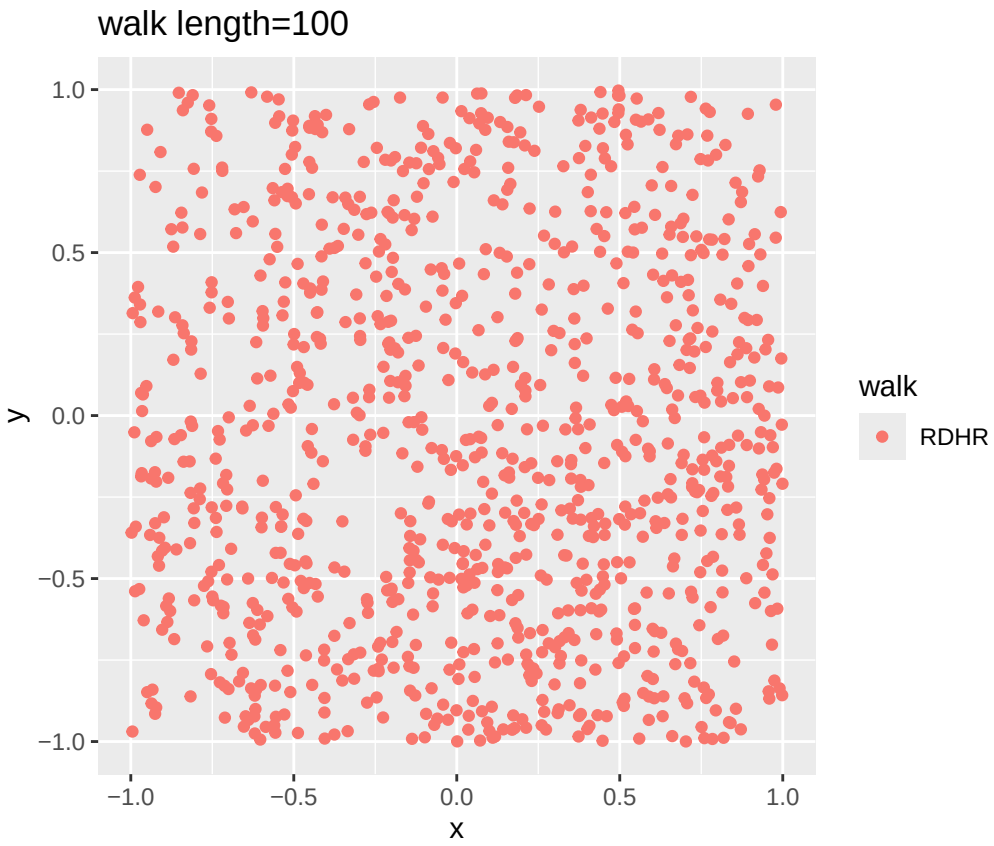

walk length=1
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
BaW


walk length=20
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
CDHR

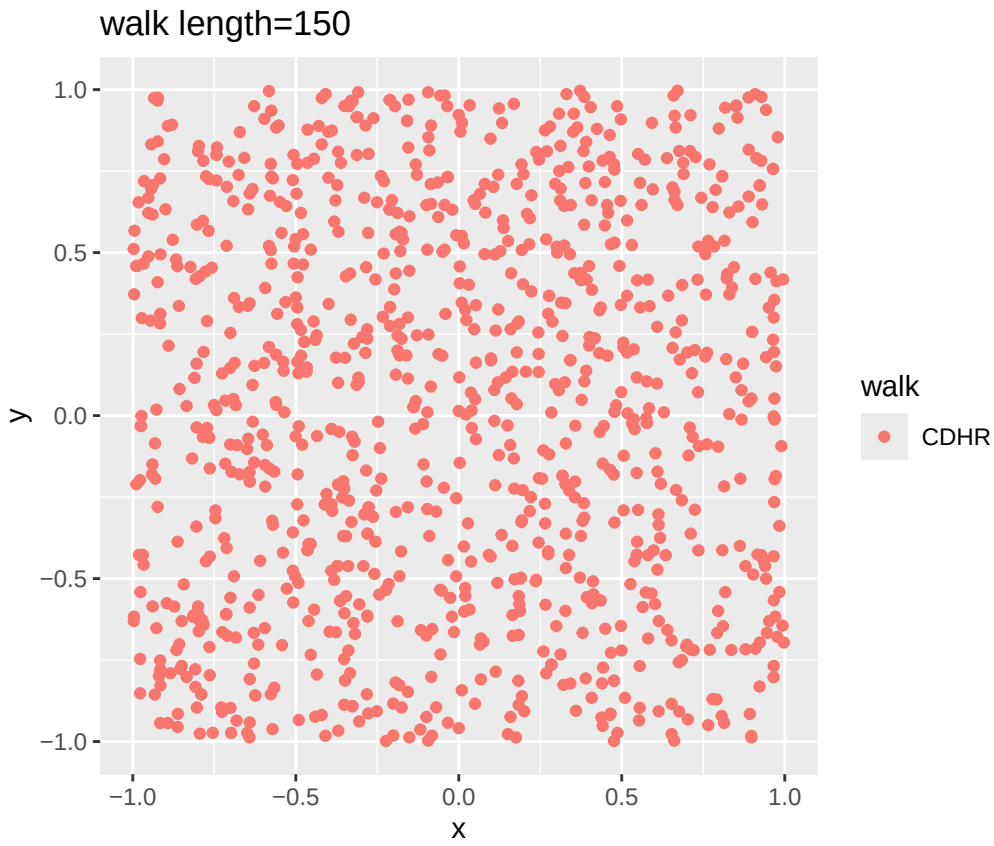

walk length=20
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
RDHR

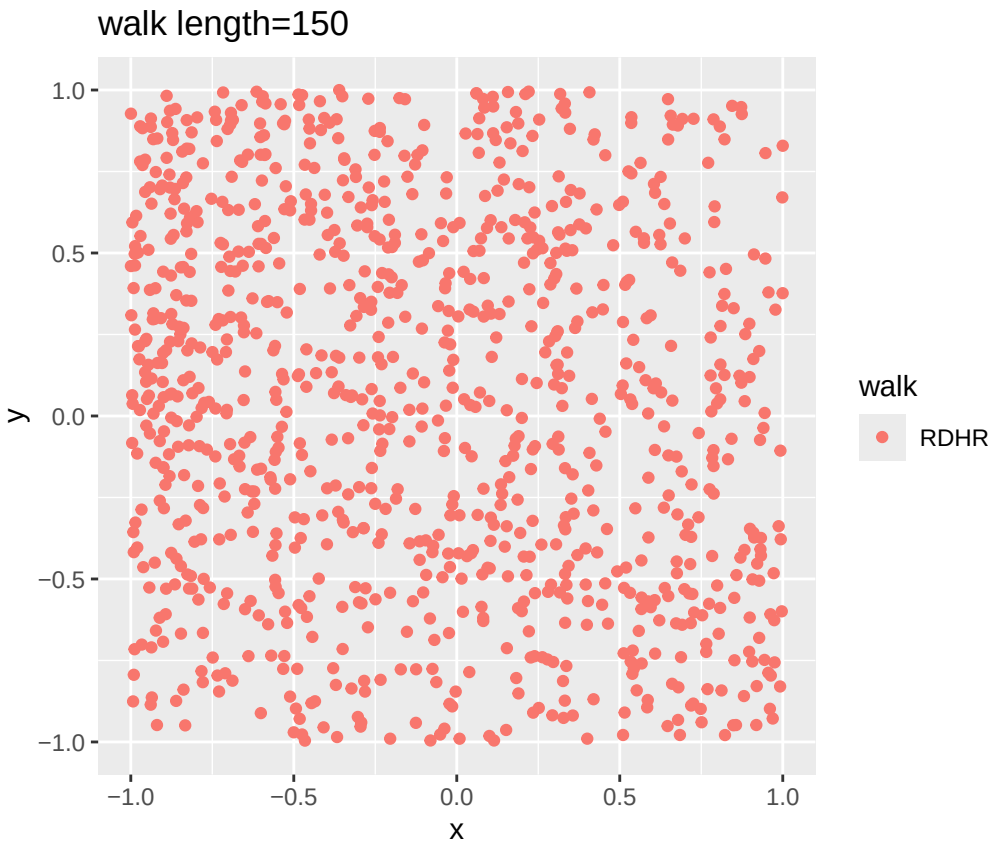

walk length=20
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
BaW

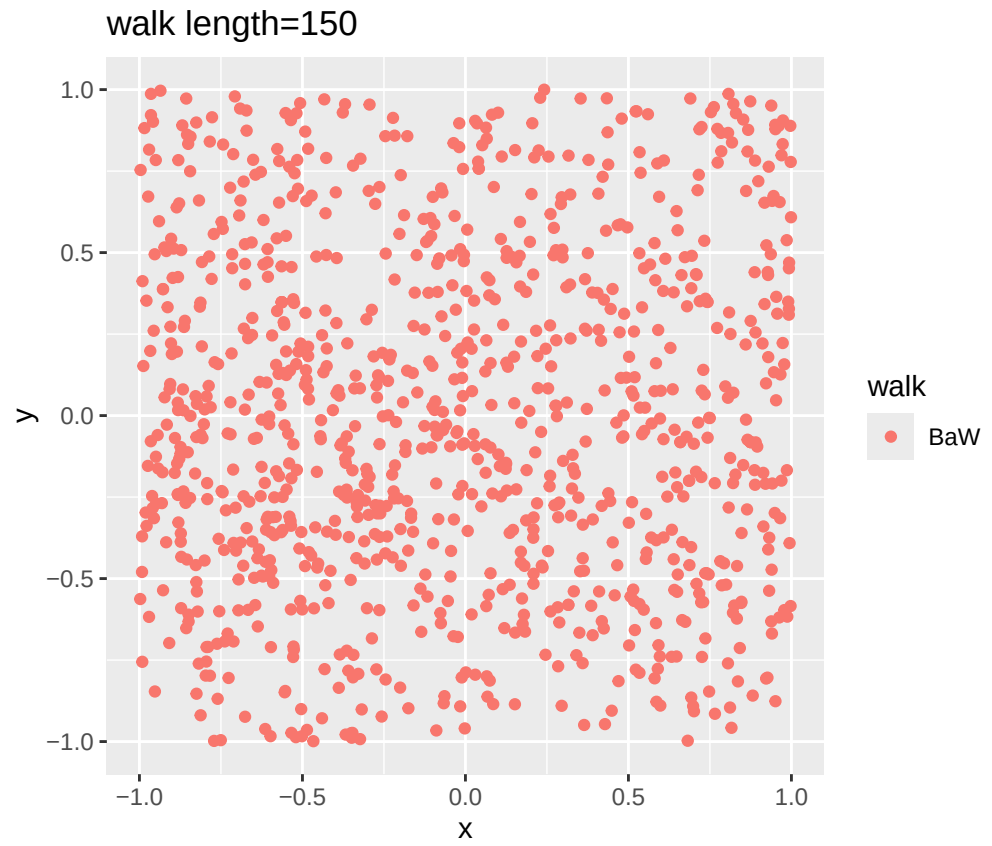

walk length=100
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
CDHR


walk length=100
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
RDHR

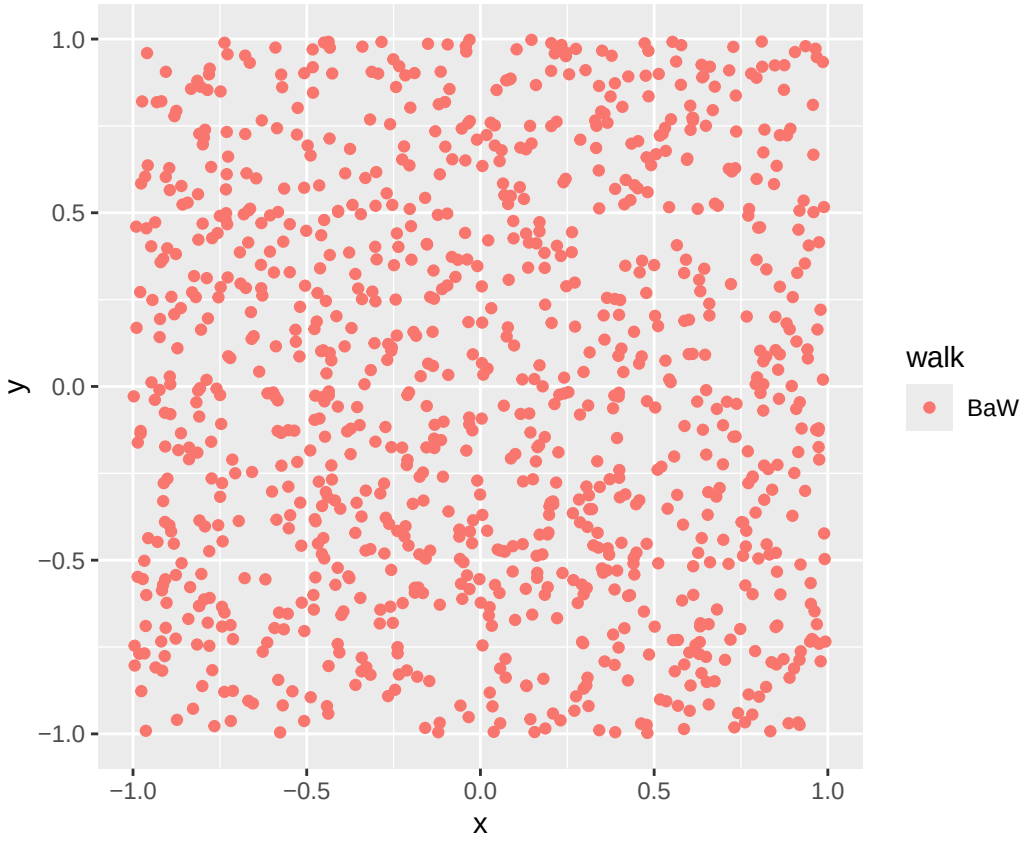

walk length=100
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
BaW


walk length=150
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
CDHR


walk length=150
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
RDHR


walk length=150
1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
walk
BaW

```
P <- gen_cube(100, 'H')
points1 = sample_points(P, n = 1000, random_walk = list("walk" = "aBiW", "walk_length" = 1))
g<-plot(ggplot(data.frame( x=points1[1,], y=points1[2,] )) +
geom_point( aes(x=x, y=y, color=walk)) + coord_fixed(xlim = c(-1,1),
ylim = c(-1,1)))
```



Another way to test the quality of the sample is to use some statistical tests, e.g. the effective sample size (ess)

```
P <- gen_cube(100, 'H')
for (walk in c("CDHR", "RDHR", "BaW", "aBiW")){
  points1 = sample_points(P, n = 1000, random_walk = list("walk" = walk, "walk_length" = 10))
  cat(walk, min(ess(points1)),"\n")
}
```

```
## CDHR 24.97454
## RDHR 3.280214
## BaW 3.046283
## aBiW 595.9506
```

Volume computation

Now let's compute our first example. The volume of the 3-dimensional cube.

We want to compare with an exact volume computation software ***geometry*** *(R wrapper for qhull) so we focus on V-polytopes.*

```r
#install.packages("geometry")
library(geometry)

PV <- gen_cube(3,'V')
str(PV)
```

```
## Formal class 'Vpolytope' [package "volesti"] with 3 slots
##   ..@ V     : num [1:8, 1:3] -1 1 -1 1 -1 1 -1 1 -1 -1 ...
##   ..@ volume: num 8
##   ..@ type  : chr "Vpolytope"
```

```r
geom_values <- convhulln(PV@V, options = 'FA')
vol_approx <- volume(PV)

cat(sprintf("exact vol = %f\napprx vol = %f\nrelative error = %f\n",
            geom_values$vol, vol_approx$volume, abs(geom_values$vol-vol_approx$volume)/geom_values$vol))
```

```
## exact vol = 8.000000
## apprx vol = 8.607548
## relative error = 0.075944
```

Now try a higher dimensional example. By setting the error parameter we can control the approximation of the algorithm.

```r
PH = gen_cube(10,'H')
volumes <- list()
for (i in 1:10) {
  volumes[[i]] <- volume(PH, settings = list("error" = 1))$volume # default parameters
}
options(digits=10)
summary(as.numeric(volumes))
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##  908.3939 1020.8826 1053.8389 1040.6910 1068.8263 1171.9247
```

```r
volumes <- list()
for (i in 1:10) {
  volumes[[i]] <- volume(PH, settings = list("error" = 0.01))$volume
}
summary(as.numeric(volumes))
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##  992.1002 1011.7640 1020.8991 1019.6855 1027.4431 1046.7015
```

Deterministic algorithms for volume are limited to low dimensions (e.g. less than 15)

```r
library(geometry)

P = gen_rand_vpoly(15, 20)
# this will return an error about memory allocation, i.e. the dimension is too high for qhull
#tim1 <- system.time({ geom_values = convhulln(P$V, options = 'FA') })

time <- system.time({ vol <- volume(P)$volume })
print(vol)
```

```
## [1] 3.267534618e-12
```

```
print(time)
```

```
##    user  system elapsed
##  12.669   1.128  13.798
```

Volume of Birkhoff polytopes

We now continue with a more interesting example, the 10-th Birkhoff polytope (dimension=81). It is known from https://arxiv.org/pdf/math/0305332.pdf that its volume equals

$vol(B_10) = \frac{7272912840167864209775084579901218625488232600525573333866078 89}{828160860106766855125676318796872729344622463533089422677980721388055739956270293750883504892820848640000000}$
obtained via massive parallel computation.

```
library(volesti)
exact <- 7272912840167864209775084579901218625488232600525573333866078 89/828160860106766855125676318796
exact
```

```
## [1] 8.782005031e-46
```

```
# warning the following will take around a minute
B <- gen_birkhoff(10)
time <- system.time({ vol <- volume(B)$volume })
print(vol)
```

```
## [1] 7.438175104e-55
```

```
print(time)
```

```
##    user  system elapsed
##  16.516   0.000  16.518
```

Compare our computed estimation with the "normalized" floating point version of $vol(B_10)$

```
n <- 10
vol_B10 <- 7272912840167864209775084579901218625488232600525573333866078 89/828160860106766855125676318
print(vol_B10/(n^(n-1)))
```

```
## [1] 8.782005031e-55
```

Rounding

We generate skinny polytopes, in particular skinny cubes of the form $\{x = x_1, \ldots, x_d) | x_1 \leq 100, x_1 \geq -100, x_i \leq 1, x_i \geq -1, x_i \in \mathbb{R}, \text{for } i = 2, \ldots, d\}$ Random walks perform poorly on those polytopes especially as the dimension increases.

```
library(ggplot2)

d <- 100

P = gen_skinny_cube(d)
P <- rotate_polytope(P)$P

for (walk in c("CDHR", "RDHR", "BaW")){
  points1 = sample_points(P, n = 1000, random_walk = list("walk" = walk, "walk_length" = 100))
  cat(walk, min(ess(points1)),"\n")
}
```

```
## CDHR 15.80258485
## RDHR 5.88882177
## BaW 9.177027206
```

```
points1 = sample_points(P, n = 1000, random_walk = list("walk" = "aBiW", "walk_length" = 1))
cat(walk, min(ess(points1)),"\n")
```

```
## BaW 7.918883067
```

Applying a rounding algorithm improves the convergence of walks.

```
Pr <- round_polytope(P)$P
for (walk in c("CDHR", "RDHR", "BaW")){
  points1 = sample_points(Pr, n = 1000, random_walk = list("walk" = walk, "walk_length" = 100))
  cat(walk, min(ess(points1)),"\n")
}
```

```
## CDHR 16.1418395
## RDHR 13.57237513
## BaW 4.818799866
```

```
points1 = sample_points(Pr, n = 1000, random_walk = list("walk" = "aBiW", "walk_length" = 1))
cat(walk, min(ess(points1)),"\n")
```

```
## BaW 273.7269229
```

Integration

We can use sampling and volume estimation to estimate integrals over polyhedral domains. Below there is an example with a degree 2 polynomial over a 3-dimensional cube.

```
library(cubature) # load the package "cubature"
f <- function(x) { 2/3 * (2 * x[1]^2 + x[2] + x[3]) + 10 }  # "x" is vector
adaptIntegrate(f, lowerLimit = c(-1, -1, -1), upperLimit = c(1, 1, 1))$integral
```

```
## [1] 83.55555556
```

```
# Simple Monte Carlo integration
# https://en.wikipedia.org/wiki/Monte_Carlo_integration
P = gen_cube(3, 'H')
num_of_points <- 10000
points1 <- sample_points(P, random_walk = list("Walk" = "aBiW", "walk_length" = 100), n=num_of_points)
int<-0
for (i in 1:num_of_points){
  int <- int + f(points1[,i])
}
V <- volume(P)$volume
print(int*V/num_of_points)
```

```
## [1] 84.95385984
```

Counting linear extensions

Let $G = (V, E)$ be an acyclic digraph with $V = [n] := \{1, 2, ..., n\}$. One might want to consider G as a representation of the partially ordered set (poset) $V : i \geq j$ if and only if there is a directed path from node i to node j.A permutation π of $[n]$ is called a linear extension of G (or the associated poset V) if $\pi^{-1}(i) \geq \pi^{-1}(j)$ for every edge $i \to j \in E$.

Let $P_{LE}(G)$ be the polytope in R^n defined by $P_{LE}(G) = \{x \in \mathbb{R}^n \mid 1 \geq x_i \geq 0 \text{ for all } i = 1, 2, \ldots, n, \text{ and } x_i \geq x_j \text{ for all directed edges } i \to j \in E$. A well known result is $\#_{LE}G = vol(P) * n!$.

The following example from this note has 9 linear extensions.

See also this paper for counting linear extensions in practice.

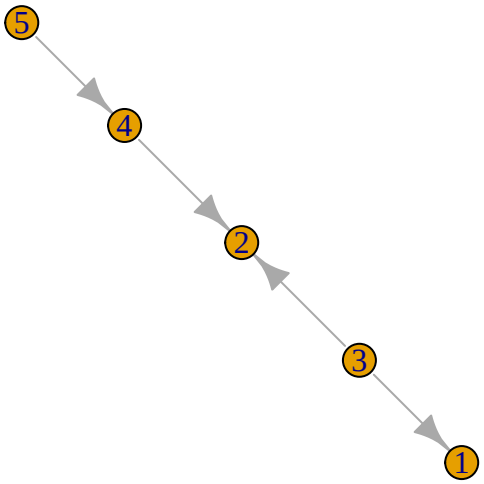
```
#install.packages("igraph")
library(igraph)
```

```
##
## Attaching package: 'igraph'
```

```
## The following objects are masked from 'package:stats':
##
##     decompose, spectrum
```

```
## The following object is masked from 'package:base':
##
##     union
```

```
# Specify edges for a directed graph
gd <- make_graph(c(5,4, 3,2, 4,2, 3,1), directed = TRUE)
plot(gd)
```




```
A = matrix(c(-1,0,1,0,0,0,-1,1,0,0,0,-1,0,1,0,0,0,0,-1,1,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,
ncol=5, nrow=14, byrow=TRUE)
b = c(0,0,0,0,0,0,0,0,0,1,1,1,1,1)
P = Hpolytope(A=A, b=b)
volume(P)$volume * factorial(5)
```

```
## [1] 9.028633337
```