

Matrices

Random matrices can either be explicitly or implicitly distributed. If they are explicitly distributed, their entries have a specific distribution. Otherwise, the entries are generated by the generative algorithm the matrix uses.

Explicitly Distributed

For explicitly distributed matrices, we can use a “function factory” method to be concise. The actual implementation is more verbose for the purposes of argument documentation, but the following code is minimal and fully functional.

```
# ... represents all the arguments taken in by the rdist function
RM_explicit <- function(rdist){
  function(N, ..., symm = FALSE){
    # Create an [N x N] matrix sampling the rows from rdist, passing ... to rdist
    P <- matrix(rdist(N^2, ...), nrow = N)
    # Make symmetric if prompted
    if(symm){P <- .makeHermitian(P)}
    # Return P
    P
  }
}

# A version where we add an imaginary component
RM_explicit_cplx <- function(rdist){
  RM_dist <- function(N, ..., symm = FALSE, cplx = FALSE, herm = FALSE){
    # Create an [N x N] matrix sampling the rows from rdist, passing ... to rdist
    P <- matrix(rdist(N^2, ...), nrow = N)
    # Make symmetric/hermitian if prompted
    if(symm || herm){P <- .makeHermitian(P)}
    # Returns a matrix with complex (and hermitian) entries if prompted
    if(cplx){
      # Recursively add imaginary components as 1i * instance of real-valued matrix.
      Im_P <- (1i * RM_dist(N, ...))
      # Make imaginary part hermitian if prompted
      if(herm){P <- P + .makeHermitian(Im_P)}
      else{P <- P + Im_P}
    }
    P # Return the matrix
  }
}
```

With our function factories set up, we can quickly generate all the random matrix functions for all the distributions our hearts could desire.

```
RM_unif <- RM_explicit_cplx(runif)
RM_norm <- RM_explicit_cplx(rnorm)
```

Implicitly Distributed

In the case of implicitly distributed matrices, we have two types: stochastic matrices and the beta ensemble matrices.

Beta Matrices

For the β -ensemble matrices, we simply use the algorithm provided in Dimitriu's paper. Doing so, we get the function:

```
RM_beta <- function(N, beta){  
  # Set the diagonal as a N(0,2) distributed row.  
  P <- diag(rnorm(N, mean = 0, sd = sqrt(2)))  
  # Set the off-1 diagonals as chi squared variables with df(beta), as given in Dumitriu's model  
  df_seq <- beta*(N - seq(1, N-1)) # Get degrees of freedom sequence for offdiagonal  
  P[row(P) - col(P) == 1] <- P[row(P) - col(P) == -1] <- sqrt(rchisq(N-1, df_seq)) # Generate tridiagonal  
  P <- P/sqrt(2) # Rescale the entries by 1/sqrt(2)  
  P # Return the matrix  
}
```

Stochastic Matrices

For stochastic matrices, we require slightly more setup. First, we setup the row functions to sample probability vectors:

```
# Generates stochastic rows of size N  
.stoch_row <- function(N){  
  row <- runif(N,0,1) # Sample probability distribution  
  row/sum(row) # Return normalized row  
}
```

For random introduced sparsity, we define the following row function.

```
# Generates same rows as in r_stoch(N), but with introduced random sparsity  
.stoch_row_zeros <- function(N){  
  row <- runif(N,0,1)  
  degree_vertex <- sample(1:(N-1), size = 1) # Sample a degree of at least 1, as to ensure row is stochastic  
  row[sample(1:N, size = degree_vertex)] <- 0 # Choose edges to sever and sever them  
  row/sum(row) # Return normalized row  
}
```

Once this is done, we can use this function iteratively. With some magic, we can incorporate an option to make the matrix symmetric, and we get the following function.

```
RM_stoch <- function(N, symm = F, sparsity = F){  
  if(sparsity){row_fxn <- .stoch_row_zeros} else {row_fxn <- .stoch_row} # Choose row function  
  # Generate the [N x N] stochastic matrix stacking N stochastic rows (using the chosen function)  
  P <- do.call("rbind", lapply(X = rep(N, N), FUN = row_fxn))  
  if(symm){ # Make symmetric (if prompted)  
    P <- .makeHermitian(P) # Make lower and upper triangles equal to each other's conjugate transpose  
    diag(P) <- rep(0, N) # Nullify diagonal  
    for(i in 1:N){P[i, ] <- P[i, ]/sum(P[i, ])} # Normalize rows  
    # Set diagonal to the diff. between 1 and the non-diagonal entry sums such that rows sum to 1  
    diag <- vector("numeric", N)  
    for(i in 1:N){diag[i] <- (1 - sum(.offdiagonalEntries(row = P[i, ], row_index = i)))}  
    diag(P) <- diag  
  }  
  P # Return the matrix  
}
```

Erdos-Renyi Matrices

For the Erdos-Renyi walks, we do something similar by defining a parameterized row function.

```

# Generates a stochastic row with parameterized sparsity of p
.stoch_row_erdos <- function(N, p){
  row <- runif(N,0,1) # Generate a uniform row of probabilities
  degree_vertex <- rbinom(1, N, 1-p) # Sample number of zeros so that degree of row/vertex i ~ Bin(n,p)
  row[sample(1:N, degree_vertex)] <- 0 # Choose edges to sever and sever them
  if(sum(row) != 0){row/sum(row)} else{row} # Return normalized row only if non-zero (cannot divide by 0)
}

```

And we again use the row function iteratively to get the following function.

```

RM_erdos <- function(N, p, stoch = T){
  # Generate an [N x N] Erdos-Renyi walk stochastic matrix by stacking N p-stochastic rows
  P <- do.call("rbind", lapply(X = rep(N, N), FUN = .stoch_row_erdos, p = p))
  # If the matrix is to be truly stochastic, map rows with all zeros to have diagonal entry 1
  if(stoch){
    # Set diagonal to ensure that rows sum to 1
    diag <- rep(0, N)
    for(i in 1:N){diag[i] <- (1 - sum(.offdiagonalEntries(row = P[i, ], row_index = i)))}
    diag(P) <- diag
  }
  P # Return the matrix
}

```

And as such, we have minimal, functional implementations of functions that sample random matrices! In total, we only needed two helper functions. The `.offdiagonalEntries` function was used to normalize the probabilities in `RM_stoch` and `RM_erdos`.

```

# Manually make equate the entries in the upper triangle to the conjugate of those in the lower triangle
.makeHermitian <- function(P){
  # Run over entry of the matrix
  for(i in 1:nrow(P)){
    for(j in 1:ncol(P)){
      # Restrict view to one of the triangles (i < j): Lower Triangle
      if(i < j){P[i,j] <- Conj(P[j,i])} # Equalize lower and upper triangles, making conjugate if complex
    }
  }
  P # Return Hermitian Matrix
}

# Return the off-diagonal entries of row i
.offdiagonalEntries <- function(row, row_index){row[which(1:length(row) != row_index)]}

```

Ensemble Extensions

Lastly, we have the ensemble extensions. These functions are quite simple to implement using a “function factory”. Again, the actual implementations are more verbose due to the argument descriptions, but otherwise, are exactly the same.

```

RME_extender <- function(RM_dist){
  # Function returns a list of replicates of the RM_dist function with ... as its arguments
  function(N, ..., size){
    lapply(X = rep(N, size), FUN = RM_dist, ...)
  }
}

```

Now, we extend the functions as follows, and we are done with the matrix module!

```
RME_unif <- RME_extender(RM_unif)
RME_norm <- RME_extender(RM_norm)
RME_beta <- RME_extender(RM_beta)
RME_stoch <- RME_extender(RM_stoch)
RME_erdos <- RME_extender(RM_erdos)
```