```r
# ... 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
  }
}
```

Beta Matrices

```r
# Generate a Hermite beta matrix using Dumitriu's Matrix Model
RM_beta <- function(N, beta){
  # Set the diagonal ~ N(0,2)
  P <- diag(rnorm(n = N, mean = 0, sd = sqrt(2)))
  # Get degrees of freedom sequence for offdigonal
  df_seq <- beta * (N - seq(1, N-1))
  # Set the off-1 diagonals as chi squared variables with df(beta_i)
  P[row(P) - col(P) == 1] <- P[row(P) - col(P) == -1] <- sqrt(rchisq(N-1, df_seq))
  # Rescale the entries by 1/sqrt(2)
  P <- P/sqrt(2)
  # Return the beta matrix
  P
}
```

Implicitly Distributed

Stochastic Matrices

```r
# Generates stochastic rows of length N
.stoch_row <- function(N){
  # Sample a vector of probabilities
  row <- runif(n = N, min = 0, max = 1)
  # Return the normalized row (sums to one)
  row / sum(row)
}
```

```r
# Generates same rows as in .stoch_row(N), but with randomly introduced sparsity
.stoch_row_zeros <- function(N){
  # Sample a vector of probabilities
  row <- runif(n = N, min = 0, max = 1)
  # Sample a vertex degree of at least one (as to ensure row is stochastic)
  degree_vertex <- sample(x = 1:(N-1), size = 1)
  # Sever a random selection of edges to set the vertex degree
  row[sample(1:N, size = N - degree_vertex)] <- 0
  # Return normalized row
  row / sum(row)
}
```

```r
RM_stoch <- function(N, symm = F, sparsity = F){
  # Choose row function depending on sparsity argument
  if(sparsity){row_fxn <- .stoch_row_zeros} else {row_fxn <- .stoch_row}
  # Generate the [N x N] stochastic matrix stacking N stochastic rows
  P <- do.call("rbind", lapply(X = rep(N, N), FUN = row_fxn))
  # Make symmetric (if prompted)
  if(symm){ P <- .makeStochSymm(P) }
  # Return the matrix
  P
}
```

Erdos-Renyi Matrices

```r
# Generates a stochastic row with parameterized sparsity of p
.stoch_row_erdos <- function(N, p){
  # Sample a vector of probabilities
  row <- runif(n = N, min = 0, max = 1)
  # Sample the vertex degree so that it is ~ Bin(n,p)
  degree_vertex <- rbinom(n = 1, size = N, prob = 1 - p)
  # Sever a random selection of edges to set the vertex degree
  row[sample(1:N, degree_vertex)] <- 0
  # Return normalized row only if non-zero (cannot divide by 0)
  if(sum(row) != 0){
    row / sum(row)
  } else{
    .stoch_row_erdos(N, p) # Otherwise, try again
  }
}
```

```r
RM_erdos <- function(N, p, stoch = T){
  # Generate an [N x N] Erdos-Renyi stochastic matrix by stacking N p-stochastic rows
  P <- do.call("rbind", lapply(X = rep(N, N), FUN = .stoch_row_erdos, p = p))
  # Return the Erdos-Renyi transition matrix
  P
}
```

```r
# Returns a Hermitian version of a matrix by manual assignment
.makeHermitian <- function(P){
  for(i in 1:nrow(P)){
    for(j in 1:ncol(P)){
      # Select the entries in the upper triangle (i < j)
      if(i < j){
        # Make the upper triangle equal to the conjugate transpose of the lower triangle
        P[i,j] <- Conj(P[j,i])
      }
    }
  }
  # Return the Hermitian Matrix
  P
}

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

Ensemble Extensions

```r
# Extends a RM_dist function to its RME_dist ensemble counterpart
RME_extender <- function(RM_dist){
  # Function returns a list of replicates of the RM_dist function with '...' as arguments
  function(N, ..., size){
    lapply(X = rep(N, size), FUN = RM_dist, ...)
  }
}
```

```r
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)
```

Spectral Statistics

Spectrum

```r
spectrum <- function(array, components = T, norm_order = T, singular = F, order = NA){
  # Digits to round values to
  digits <- 4
  # Get the type of array
  array_class <- .arrayClass(array)
  # For ensembles, iteratively rbind() each matrix's spectrum
  if(array_class == "ensemble"){
    map_dfr(array, .spectrum_matrix, components, norm_order, singular, order, digits)
  }
  # From matrices, call the function returning the ordered spectrum for a singleton matrix
  else if(array_class == "matrix"){
    .spectrum_matrix(array, components, norm_order, singular, order, digits)
  }
}
```

```r
# Helper function returning tidied eigenvalue array for a matrix
.spectrum_matrix <- function(P, components, norm_order, singular, order, digits = 4){
  # For singular values, take P as product of the itself and its tranpose
  if(singular){P <- P %*% t(P)}
  # Get the eigenvalues of P
  eigenvalues <- eigen(P, only.values = TRUE)$values
  # Take the square root of the eigenvalues to obtain singular values
  if(singular){eigenvalues <- sqrt(eigenvalues)}
  # Sort the eigenvalues to make it an ordered spectrum
  eigenvalues <- .sortValues(eigenvalues, norm_order)
  # If uninitialized, select all orders; otherwise, use c() so singletons => vectors
  if(class(order) == "logical"){order <- 1:nrow(P)} else{order <- c(order)}
  # Return the spectrum of the matrix
  purrr::map_dfr(order, .resolve_eigenvalue, eigenvalues, components, digits)
}
```

```r
# Read and parse an eigenvalue from a sorted eigenvalue array
.resolve_eigenvalue <- function(order, eigenvalues, components, digits){
  # Read from a sorted eigenvalue array at that order
  eigenvalue <- eigenvalues[order]
  # Get norm and order columns
  features <- data.frame(Norm = abs(eigenvalue), Order = order)
  if(components){
    # If components are sought, resolve the eigenvalue into seperate columns first
    res <- cbind(data.frame(Re = Re(eigenvalue), Im = Im(eigenvalue)), features)
  } else{
    # Otherwise, don't resolve the eigenvalue components
    res <- cbind(data.frame(Eigenvalue = eigenvalue), features)
  }
  # Round entries and return the resolved eigenvalue
  res <- round(res, digits)
  return(res)
}
```

Helper Functions

```r
# Parses an array to see classify it as a matrix or an ensemble of matrices.
.arrayClass <- function(array){
  # Sample an element from the array and get its class
  elem <- array[[1]]
  types <- class(elem)
  # Classify it by analyzing the element class
  if("numeric" %in% types || "complex" %in% types){
    return("matrix")
  }
  else if("matrix" %in% types){
    return("ensemble")
  }
}

# Sort an array of numbers by their norm (written for eigenvalue sorting)
.sortValues <- function(vals, norm_order){
  values <- data.frame(value = vals)
  # If asked to sort by norms, arrange by norm and return
  if(norm_order){
    values$norm <- abs(values$value)
    values <- values %>% arrange(desc(norm))
    # Return the norm-sorted values
    values$value
  }
  # Otherwise, sort by sign and return
  else{ sort(vals, decreasing = TRUE) }
}
```

Dispersions

```r
# Compute the dispersion of a matrix or matrix ensemble
dispersion <- function(array, pairs = NA, norm_order = T, singular = F, pow_norm = 1){
  # Digits to round values to
  digits <- 4
  # Get the type of array
  array_class <- .arrayClass(array)
  # Parse input and generate pair scheme (default NA), passing on array for dimension
  pairs <- .parsePairs(pairs, array, array_class)
  # For ensembles; iteratively rbind() each matrix's dispersion
  if(array_class == "ensemble"){
    map_dfr(array, .dispersion_matrix, pairs, norm_order, singular, pow_norm, digits)
  }
  # Array is a matrix; call function returning dispersion for singleton matrix
  else if(array_class == "matrix"){
    .dispersion_matrix(array, pairs, norm_order, singular, pow_norm, digits)
  }
}
```

```r
# Find the eigenvalue dispersions for a given matrix
.dispersion_matrix <- function(P, pairs, norm_order, singular, pow_norm, digits = 4){
  # Get the ordered spectrum of the matrix
  eigenvalues <- spectrum(P, norm_order = norm_order, singular = singular)
  # Generate norm function to pass along as argument (Euclidean or Beta norm)
  norm_fn <- function(x){ (abs(x))^pow_norm }
  # Compute and return the dispersion
  map2_dfr(pairs[["i"]], pairs[["j"]], .resolve_dispersion, eigenvalues, norm_fn, digits)
}
```

```r
# Read and parse a dispersion observation between eigenvalue i and j.
.resolve_dispersion <- function(i, j, eigenvalues, norm_fn, digits){
  # Initialize dispersion dataframe by adding order of eigenvalues compared
  disp <- data.frame(i = i, j = j)
  # Add the eigenvalues
  disp$eig_i <- .read_eigenvalue(i, eigenvalues)
  disp$eig_j <- .read_eigenvalue(j, eigenvalues)
  # Get the identity difference
  disp$id_diff <- disp$eig_j - disp$eig_i
  # Compute norm of the identity difference (standard norm metric)
  disp$id_diff_norm <- norm_fn(disp$id_diff)
  # Compute the difference of absolutes
  disp$abs_diff <- norm_fn(disp$eig_j) - norm_fn(disp$eig_i)
  # Round digits
  disp <- round(disp, digits)
  # Get the ranking difference
  disp$diff_ij <- disp$i - disp$j
  # Return the resolved dispersion observation
  disp
}
```

Helper Functions

```r
# Parse a string argument for which pairing scheme to utilize
.parsePairs <- function(pairs, array, array_class){
  # Valid schemes for printing if user is unaware of options
  valid_schemes <- c("largest", "lower", "upper", "consecutive", "all")
  # Set default to be the consecutive pair scheme
  if(class(pairs) == "logical"){pairs <- "consecutive"}
  # Stop function call if the argument is invalid
  if(!(pairs %in% valid_schemes)){
    scheme_list <- paste(valid_schemes, collapse = ", ")
    stop(paste("Invalid pair scheme. Try one of the following: ", scheme_list, ".", ""))
  }
  # // Once we verify that we have a valid pair scheme string, try to parse it.
  # First, obtain a matrix by inferring array type; if ensemble take first matrix
  if(array_class == "ensemble") { P <- array[[1]] }
  else if(array_class == "matrix") { P <- array }
  # Obtain the dimension of the matrix
  N <- nrow(P)
  # Parse the pair string and evaluate the pair scheme
  if(pairs == "largest"){pair_scheme <- data.frame(i = 2, j = 1)}
  else if(pairs == "consecutive"){pair_scheme <- .consecutive_pairs(N)}
  else if(pairs == "lower"){pair_scheme <- .unique_pairs_lower(N)}
  else if(pairs == "upper"){pair_scheme <- .unique_pairs_upper(N)}
  else if(pairs == "all"){pair_scheme <- .all_pairs(N)}
  # Return pair scheme
  return(pair_scheme)
}
```

Pairing Schema

```
#==================================================================================#
# The trivial pairing scheme:
# Enumerate all possible pairs.
.all_pairs <- function(N){
  purrr::map_dfr(1:N, function(i, N){data.frame(i = rep(i, N), j = 1:N)}, N)
}
#==================================================================================#
# The consecutive pairing scheme:
# Enumerate all possible consecutive/neighboring pairs. Ensures no linear combiantions.
.consecutive_pairs <- function(N){
  purrr::map_dfr(2:N, function(i){data.frame(i = i, j = as.integer(i - 1))})
}
#==================================================================================#
# The lower-triangular pairing scheme:
# Enumerate the pair combinations given N items with i > j.
.unique_pairs_lower <- function(N){
  is <- do.call("c", purrr::map(1:N, function(i){rep(i,N)}))
  js <- rep(1:N, N)
  # Helper function: selects elements only if they are lower triangular
  .isLowerTri <- function(i, j){if(i > j){ c(i = i, j = j) }}
  pairs <- do.call("rbind",purrr::map2(is, js, .f = .isLowerTri))
  data.frame(pairs)
}
#==================================================================================#
# The upper-triangular pairing scheme:
# Enumerate the pair combinations given N items with i < j.
.unique_pairs_upper <- function(N){
  is <- do.call("c", purrr::map(1:N, function(i){rep(i,N)}))
  js <- rep(1:N, N)
  # Helper function: selects elements only if they are lower triangular
  .isUpperTri <- function(i, j){if(i < j){ c(i = i, j = j) }}
  pairs <- do.call("rbind",purrr::map2(is, js, .f = .isUpperTri))
  data.frame(pairs)
}
```

Parallel Extensions

Spectrum

```
spectrum_parallel <- function(array, components = TRUE, sort_norms = TRUE, singular = FALSE, order = NA)
  digits <- 4 # Digits to round values to
  # Array is a matrix; call function returning eigenvalues for singleton matrix
  if(class(array) == "matrix"){
    .spectrum_matrix(array, components, sort_norms, singular, order, digits)
  }
  # Array is an ensemble; recursively row binding each matrix's eigenvalues
  else if(class(array) == "list"){
    furrr::future_map_dfr(array, .spectrum_matrix, components, sort_norms, singular, order, digits)
  }
}
```

Dispersion

```
dispersion_parallel <- function(array, pairs = NA, sort_norms = TRUE, singular = FALSE, norm_pow = 1){
  digits <- 4 # Digits to round values to
  pairs <- .parsePairs(pairs, array) # Parse input and generate pair scheme (default NA), passing on ar
  # Array is a matrix; call function returning dispersion for singleton matrix
  if(class(array) == "matrix"){
    .dispersion_matrix(array, pairs, sort_norms, singular, norm_pow, digits)
  }
  # Array is an ensemble; recursively row binding each matrix's dispersions
  else if(class(array) == "list"){
    furrr::future_map_dfr(array, .dispersion_matrix, pairs, sort_norms, singular, norm_pow, digits)
  }
}
```