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

```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
}
```

```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
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
# 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, ...)
  }
}
```

```
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
# 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) }
}
```

```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
# 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)
}
```

```r
#==========================================================================#
# 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)
  }
}
```