Matrices

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

Explicitly Distributed

For (homogenous) 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. Additionally, there are the beta matrices, which use the matrix model provided by the algorithm in Dimitriu's paper.

Homogenously Distributed

```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
  }
}
```

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

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

Beta Matrices

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

```r
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 offdigonal
  P[row(P) - col(P) == 1] <- P[row(P) - col(P) == -1] <- sqrt(rchisq(N-1, df_seq)) # Generate tridiagon
  P <- P/sqrt(2) # Rescale the entries by 1/sqrt(2)
  P # Return the matrix
}
```

Implicitly Distributed

In the case of implicitly distributed matrices, we have various types of stochastic matrices.

Stochastic Matrices

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

```r
# 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.

```r
# 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 stoch
  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.

```r
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.

```r
# 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 probabilites
  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
}
```

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

```r
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`.

```r
# Manually make equate the entries in the upper triangle to the conjugate of those in the lower triangl
.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 compl
    }
  }
  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 user 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)
```

Spectral Statistics

Spectrum

```r
spectrum <- 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"){
    purrr::map_dfr(array, .spectrum_matrix, components, sort_norms, singular, order, digits)
  }
}
```

```r
# Helper function returning tidied eigenvalue array for a matrix
.spectrum_matrix <- function(P, components, sort_norms, singular, order, digits = 4){
  # If prompted for singular values, then take the product of the matrix and its tranpose instead
  if(singular){P <- P %*% t(P)}
  # Get the sorted eigenvalue spectrum of the matrix
  eigenvalues <- eigen(P)$values # Compute the eigenvalues of P
  if(singular){eigenvalues <- sqrt(eigenvalues)} # Take the square root of the eigenvalues
  if(sort_norms){eigenvalues <- .sort_by_norm(eigenvalues)} # Order the eigenvalue spectrum by norm rat
  else{eigenvalues <- sort(eigenvalues, decreasing = TRUE)} # Else, sort by sign.
  # If uninitialized, get eigenvalues of all orders; Otherwise, concatenate so single inputs become vec
  if(class(order) == "logical"){order <- 1:nrow(P)} else{order <- c(order)}
  purrr::map_dfr(order, .resolve_eigenvalue, eigenvalues, components, digits) # Get the eigenvalues
}
```

```r
# Read and parse an eigenvalue from an eigen(P)$value array
.resolve_eigenvalue <- function(order, eigenvalues, components, digits){
  eigenvalue <- eigenvalues[order] # Read from eigen(P)$values
  # Get norm and order columns (will unconditionally be returned)
  norm_and_order <- data.frame(Norm = abs(eigenvalue), Order = order)
  # If components are requested, resolve parts into seperate columns and cbind to norm and order
  if(components){evalue <- cbind(data.frame(Re = Re(eigenvalue), Im = Im(eigenvalue)), norm_and_order)}
  else{evalue <- cbind(data.frame(Eigenvalue = eigenvalue), norm_and_order)}
  evalue <- round(evalue, digits) # Round entries
  evalue # Return resolved eigenvalue
}
```

Helper Functions

```r
# Sort an array of numbers by their norm (written for eigenvalue sorting)
.sort_by_norm <- function(eigenvalues){
  (data.frame(eigenvalue = eigenvalues, norm = abs(eigenvalues)) %>% arrange(desc(norm)))$eigenvalue
  }
```

Dispersions

```r
#=================================================================================#
#                                   DISPERSION
#=================================================================================#

dispersion <- function(array, pairs = NA, norm_order = TRUE, singular = FALSE, 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)
  # Array is an ensemble; recursively row binding each matrix's dispersions
  if(array_class == "ensemble"){
    disp <- purrr::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"){
    disp <- .dispersion_matrix(array, pairs, norm_order, singular, pow_norm, digits)
  }
  # Resolve column types; i.e. coerce real-valued eigenvalues to a numeric type if possible
  disp <- .resolveNumType(disp)
  # Return the dispersion
  disp
}

#=================================================================================#
# 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 the dispersion
  disp <- purrr::map2_dfr(pairs[["i"]], pairs[["j"]], .resolve_dispersion, eigenvalues, norm_fn, digits
  # Return the dispersion
  disp
}

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

```r
}

#==================================================================================#
#                           DISPERSION: HELPER FUNCTIONS
#==================================================================================#

# Parses a matrix spectrum array for the eigenvalue at a given order as cplx type (for arithmetic)
.read_eigenvalue <- function(order, mat_spectrum){
  # If the components are not resolved, return value in the first (Eigenvalue) column
  if(ncol(mat_spectrum) == 3){ mat_spectrum[order, "Eigenvalue"] }
  # Components are resolved; get components and make it a complex number for arithmetic prep
  else{ complex(real = mat_spectrum[order, "Re"], imaginary = mat_spectrum[order, "Im"]) }
}

#==================================================================================#
# Resolves the numerical types of the eigenvalue columns of the dispersion dataframe
.resolveNumType <- function(disp){
  # See if the eigenvalues are complex
  is_complex <- FALSE %in% c(Im(disp$eig_i) == 0, Im(disp$eig_j) == 0)
  # If it is real, coerce it into a numeric, removing the +0i
  if(!is_complex){
    disp$eig_i <- as.numeric(disp$eig_i)
    disp$eig_j <- as.numeric(disp$eig_j)
    disp$id_diff <- as.numeric(disp$id_diff)
  }
  disp # Return the resolved dispersion
}

#==================================================================================#
# Parse a string argument for which pairing scheme to utilize
.parsePairs <- function(pairs, array, array_class){
  valid_schemes <- c("largest", "lower", "upper", "consecutive", "all") # Valid schemes for printing if
  # Obtain the matrix by inferring array type; if ensemble take first matrix
  if(array_class == "ensemble"){
    P <- array[[1]]
  } else if(array_class == "matrix"){
    P <- array
  }
  if(class(pairs) == "logical"){pairs <- "consecutive"} # Set default value to be the consecutive pair
  # 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, ".", ""))
  }
  # 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)}
  pair_scheme # Return pair scheme
```

```r
}

#==================================================================================#
#                                 PAIRING SCHEMES
#==================================================================================#

# 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. No linear combiantions of dispersions exist.
.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)
  pairs <- do.call("rbind",purrr::map2(is, js, .f = function(i, j){if(i > j){c(i = i, j = j)}}))
  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)
  pairs <- do.call("rbind",purrr::map2(is, js, .f = function(i, j){if(i < j){c(i = i, j = j)}}))
  data.frame(pairs)
}
```

Pairing Schema

```r
# Parse a string argument for which pairing scheme to utilize
.parsePairs <- function(pairs, array){
  valid_schemes <- c("largest", "lower", "upper", "consecutive", "all") # Valid schemes for printing if
  # Obtain the matrix by inferring array type; if ensemble take first matrix
  if(class(array) == "list"){P <- array[[1]]} else{P <- array}
  if(class(pairs) == "logical"){pairs <- "consecutive"} # Set default value to be the consecutive pair
  # Stop function call if the argument is invalid
  if(!(pairs %in% valid_schemes)){stop(paste("Invalid pair scheme. Try one of the following: ",paste(va
  # 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)}
  pair_scheme # Return pair scheme
}
```

```r
# The antisymmetric pair scheme (for assymetric dispersion metrics); essentially all the permutations
.all_pairs <- function(N){
  purrr::map_dfr(1:N, function(i, N){data.frame(i = rep(i, N), j = 1:N)}, N)
}

# The consecutive-value scheme (Sufficient such that no linear combiantions of the diseprsion metric ex
.consecutive_pairs <- function(N){
  purrr::map_dfr(2:N, function(i){data.frame(i = i, j = i - 1)})
}

# The triangular pair schema (for symmetric dispersion metrics); essentially all the combinations
# (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)
  do.call("rbind",purrr::map2(is, js, .f = function(i, j){if(i > j){c(i = i, j = j)}}))
}
# The triangular pair schema (for symmetric dispersion metrics); essentially all the combinations
# (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)
  do.call("rbind",purrr::map2(is, js, .f = function(i, j){if(i < j){c(i = i, j = j)}}))
}
```

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)
  }
}
```

Visualization Functions

Spectrum Visualization

```r
spectrum.scatterplot <- function(array, ..., mat_str = ""){
  # Process spectrum of the matrix/ensemble
  if(class(array) == "list" || class(array) == "matrix"){
    array_spectrum <- spectrum(array, ...)
  }
  # Else, the array is a precomputed spectrum (avoid computational waste for multiple visualizations)
  else{
    array_spectrum <- array
  }
  # Infer plot title string from which type of array (matrix/ensemble)
  title_str <- .plot_title(class(array), prefix = "Spectrum", mat_str)
  # Plot parameters
  order <- array_spectrum[["Order"]]
  # Plot
  array_spectrum %>%
    ggplot() +
    geom_point(mapping = aes(x = Re, y = Im, color = order), alpha = 0.75) +
    scale_color_continuous(type = "viridis") +
    labs(x = "Re", y = "Im", title = paste(title_str,sep = "")) +
    coord_fixed()
}
spectrum.histogram <- function(array, ..., component = NA, bins = 100, mat_str = ""){
  # Process spectrum of the matrix/ensemble
  if(class(array) == "list" || class(array) == "matrix"){array_spectrum <- spectrum(array, ...)}
  else{array_spectrum <- array} # Else, the array is a precomputed spectrum (avoid computational waste
  # Infer plot title string from which type of array (matrix/ensemble)
  title_str <- .plot_title(class(array), prefix = "Spectrum", mat_str)
  # Plot parameters
  color0 <- "mediumpurple3"
  num_entries <- nrow(array) # Get number of entries to normalize
  if(class(component) == "logical"){component <- c("Re", "Im")} # Set default to both components
  # Plot lambda function
  component_plot <- function(component){
    # Plot parameters
    component_str <- paste(" (",component,")",collapse = "")
    # Plot
    array_spectrum %>%
      ggplot() +
      geom_histogram(mapping = aes_string(x = component), fill = color0, bins = bins) +
      labs(x = component, y = "Frequency", title = paste(title_str, component_str, sep = ""))
  }
  # Get list of plots
  plots <- purrr::map(component, component_plot)
  # If we have both components and patchwork is loaded, attach plots to each other
  if(length(plots) == 2){plots[[1]] / plots[[2]]} else if(length(plots) == 1){plots[[1]]}
  # Return the list of plots
  else{plots}
}

# Helper function returning appoporiate string for matrix/ensemble given a matrix type string and class
```

```r
.plot_title <- function(array_class, prefix, mat_str){
  if(mat_str != ""){pre_space <- " "} else{pre_space <- ""} # Format without given name
  # Infer plot title string from which type of array (matrix/ensemble)
  if(array_class == "matrix"){plot_str <- paste(pre_space, mat_str," Matrix", sep = "", collapse = "")}
  else if(array_class == "list"){plot_str <- paste(pre_space, mat_str," Matrix Ensemble", sep = "", coll
  else{plot_str <- paste(pre_space, mat_str," Matrix Ensemble", sep = "", collapse = "")}
  paste(prefix," of a",plot_str, sep = "")
}
```

```r
order.scatterplot <- function(spectrum, component){
  spectrum %>%
    ggplot(aes(x = Order, y = {{ component }}, color = Order)) +
    geom_point() +
    scale_color_viridis_c() +
    theme(legend.position = "bottom")
}
order.density <- function(spectrum, component){
  spectrum %>%
    ggplot(mapping = aes(group = Order, x = {{ component }}, color = Order)) +
    geom_density() +
    scale_color_viridis_c() +
    theme(legend.position = "bottom")
}
order.summary <- function(spectrum, component){
  spectrum %>%
    group_by(Order) %>%
    summarize(
      Mean_Re = mean(Re), Mean_Im = mean(Im), Mean_Norm = mean(Norm),
      Variance_Re = var(Re), Variance_Im = var(Im), Variance_Norm = var(Norm)) %>%
    ggplot(mapping = aes(y = {{ component }}, x = Order, color = Order)) +
    geom_point() +
    geom_line() +
    scale_color_viridis_c() +
    theme(legend.position = "bottom")
}
```

Dispersion Visualization

```r
dispersion.histogram <- function(array, metric = NA, ..., bins = 100){
  valid_schemes <- c("id_diff","id_diff_norm","abs_diff") # Valid schemes for printing if user is unawa
  if(class(metric) == "logical"){stop("Please input a valid dispersion metric. Try one of the following
  # Process spectrum of the matrix/ensemble
  if(class(array) == "list" || class(array) == "matrix"){ disps_df <- dispersion(array, ...) }
  else{disps_df <- array} # Otherwise, the array is a precomputed dispersion dataframe
  num_entries <- nrow(disps_df) # Get number of entries
  # Plot parameters
  color0 <- "darkorchid4"
  # Return plot
  ggplot(data = disps_df, mapping = aes_string(x = metric)) +
    geom_histogram(mapping = aes(y = stat(count / num_entries)), bins = bins) +
    labs(title = "Distribution of Eigenvalue Spacings", y = "Probability")
}
```

```r
dispersion.scatterplot <- function(array, metric = "id_diff_norm", pairs = NA, ...){
  # Process dispersion of the matrix/ensemble; if array is a dispersion data frame, copy it.
  if(class(array) == "list" || class(array) == "matrix"){disps_df <- dispersion(array, pairs, ...)}
  else{disps_df <- array} # Otherwise, the array is a precomputed dispersion dataframe
  # Parse plotting aesthetics from pairs. diff_ij is more useful unless pairs = "consecutive" or "large
  if(pairs %in% c("consecutive","largest")){order_stat <- "j"} else{order_stat <- "diff_ij"}

  # Plot parameters
  color0 <- "darkorchid4"
  real_valued <- T
  # Scatterplot of dispersion metric
  if(real_valued){
    disps_df %>%
      ggplot(mapping = aes_string(x = metric, y = order_stat, color = order_stat)) +
      geom_point() +
      scale_color_continuous(type = "viridis") +
      labs(title = "Distribution of Eigenvalue Spacings by Order Statistic")
  } else{
      resolved <- .resolve_eigenvalues(disps_df)
      resolved %>%
        ggplot() +
        geom_point()
    }
}
```