Tensor Omics (TOX)

Analyze gene expression data and evolutionary relationships between genes

User's Guide

Akdi Mohamed[1], **Bass Vivian**[1], **Beketova Anna**[1], **Daba Jitu**[1], **Faensen Luka**[1], **Hallab Asis**[1], **Schröder Aaron**[1], **Schwarzpaul Alexander**[1], **Sill Franz-Eric**[1]

[1]University of Applied Sciences Bingen, Germany

Last revised October 20, 2025

Contents

1 Tensor Omics Method and Theory

The Tensor Omics (TOX) tool implements a streamlined pipeline for analyzing gene expression data and evolutionary relationships between genes. The pipeline integrates tools for data normalization, and geometric analysis of gene expression vectors. This document outlines the current implementation details and provides the mathematical formulation of the algorithms used in the prototype.

1.1 The TOX Data Table

The foundational data structure in Tensor Omics is the **Data Table**. Unlike a simple spreadsheet, a Data Table is a conceptual grouping of multiple, type-specific 2D arrays that represent datasets read from a source like a CSV file. Because Fortran is a strongly-typed language, data of different types (e.g., integers, reals, characters) cannot be stored in a single matrix. The TOX library solves this by parsing the source data into a collection of distinct, column-major arrays that share the same number of rows.

The process begins by reading the entire CSV file into a single, general-purpose 2D array of strings. This string array acts as an in-memory staging area. From this staging array, users can then extract specific columns into strongly-typed arrays, creating the components of the Data Table:

- `integer_data(:,:)`: An array for integer columns (e.g., gene IDs).
- `real_data(:,:)`: An array for real-valued columns (e.g., expression levels, scores).
- `logical_data(:,:)`: An array for boolean columns (e.g., status flags).
- `character_data(:,:)`: An array for string columns (e.g., gene names).
- `complex_data(:,:)`: An array for complex number columns.

All downstream analyses in TOX operate on these strongly-typed arrays.

1.2 From Raw Data to Expression Vectors

The starting point for all downstream analyses is a table of gene-wise expression measurements across a set of biological samples, such as tissues, developmental stages, or experimental conditions. These measurements may originate from transcriptomic, proteomic, or metabolomic experiments, typically normalized to account for sequencing depth and technical variation.

What is Gene Expression Data? For each gene, a real-valued quantity is recorded that reflects its biological activity in a given sample. In the case of transcriptomics, this is often the abundance of messenger RNA (mRNA) molecules transcribed from that gene, estimated via sequencing and mapped back to the gene's known sequence. The result is a non-negative real number proportional to the gene's transcriptional activity in that sample. After normalization and transformation, these values can be treated as coordinates in a real vector space.

Thus, each gene is represented as a vector $\vec{v} \in \mathbb{R}^n$, where n is the number of biological samples (e.g., tissues or time points), and each component $v_i \geq 0$ corresponds to the expression level in sample i. These vectors encode the distribution of gene activity across conditions and form the geometric basis for all downstream analyses.

Note on Stress Responses. In certain studies, a subset of the samples may correspond to stress conditions. Here, "stress" refers to any external or internal perturbation—such as mechanical damage, heat shock, pathogen infection, or nutrient deprivation—that triggers measurable changes in gene expression. These are often analyzed relative to baseline conditions using log fold-change transformations, such that the resulting vectors reflect differential expression patterns, including up- and downregulation across dimensions.

1.3 Normalization of Expression Values

Raw gene expression data is normalized in the following steps:

1. Division by gene-wise standard deviation for scale normalization.
2. Quantile normalization across samples to reduce global expression biases.
3. Log-transformation: $x \mapsto \log(1 + x)$ to stabilize variance and suppress outliers.
4. For stress response studies, log fold-changes are computed:

   $$\Delta_{\text{stress}} = \log(1 + x_{\text{stress}}) - \log(1 + x_{\text{control}})$$

   This effectively makes each stress related axis show the fold change in gene expression under stress, e.g. "drought in root divided by control root".

1.4 Calculation of family Centroids

To summarize the expression profile of each family, Tensor Omics computes a centroid vector for every group. The centroid represents the average expression pattern of all genes assigned to the family across all conditions or tissues.

There are two modes for selecting the set of genes to include in the centroid:

- **Full mode:** All genes assigned to the family are used to compute the centroid (mean vector).
- **Orthologs-only mode:** Only genes explicitly marked as orthologs are included in the centroid calculation. This mode is useful for focusing on conserved gene relationships across species.

1.5 Euclidean Distance Between Expression Vectors

The Euclidean distance is used throughout Tensor Omics to quantify the geometric difference between two gene expression vectors, or between a gene and its family centroid. Given two vectors $\vec{v}_1, \vec{v}_2 \in \mathbb{R}^n$, the Euclidean distance is defined as:

$$d(\vec{v}_1, \vec{v}_2) = \sqrt{\sum_{i=1}^{n} (v_{1,i} - v_{2,i})^2}$$

This metric measures the straight-line distance between two points in n-dimensional space. In the context of gene expression, it reflects the overall magnitude of difference in expression profiles across all samples or conditions.
For example, the distance between a paralog's expression vector and the centroid of its gene family is used to detect outlier genes and to quantify the degree of divergence.

1.6 Outlier Detection Based on Family-Scaled Distances

To robustly identify genes with divergent expression profiles within their families, Tensor Omics implements an outlier detection procedure based on family-scaled Euclidean distances. The process consists of the following steps:

1. **Compute Euclidean Distances:** For each gene, calculate the Euclidean distance d_i between its expression vector and the centroid of its assigned family (see Section 1.5).

2. **Estimate Family-Specific Scaling Factors:** For each family f, we compute a family-specific scaling factor. This can be the maximum distance between any ortholog and the ortholog centroid. For families lacking sufficient orthologs, a LOESS smoothing strategy is used to estimate an appropriate scaling factor based on the family's median intra-member distance.

3. **Relative Distance Index (RDI):** For each gene i in family f, compute the Relative Distance Index (RDI):
$$\mathrm{RDI}_i = \frac{d_i}{d_{scale}}$$
This normalizes the gene's distance by the expected intra-family variability, making the metric comparable across families of different sizes and variances.

4. **Outlier Detection:** Genes are flagged as outliers if their RDI exceeds a chosen percentile threshold (typically the 95th percentile) of the empirical RDI distribution. That is, gene i is an outlier if $\mathrm{RDI}_i \geq T$, where T is the threshold value corresponding to the desired percentile.

This approach ensures that outlier detection is adaptive to the structure and variability of each gene family, and is robust to differences in family size and dispersion. The LOESS-based scaling is particularly important for families with few members.

1.7 Tissue Versatility and Specificity

Tissue Versatility quantifies how uniformly a gene is expressed across all tissues (or, more generally, axes of the expression space). It is defined as the cosine of the angle between a gene's expression vector $\vec{v}$ and the space diagonal $\vec{d} = (1, 1, \ldots, 1)$. The smaller this angle (the larger the cosine), the greater the tissue (axis) versatility of the gene's expression.

Interpretation:

- A lower angle (cosine closer to 1) means the gene is more evenly expressed across all tissues (high versatility).
- A larger angle (cosine closer to 0) means the gene is active in only a few tissues (high specificity).

Dimensionality Dependence

The maximum possible angle to the diagonal depends on the number of tissues n. The cosine of this angle is minimal when a gene is expressed in only one tissue:

$$\cos(\varphi_{\mathrm{max\_angle}}) = \frac{1}{\sqrt{n}}$$

Normalized Tissue Versatility
To make versatility comparable across datasets with different numbers of tissues, we define a normalized version that scales values into the interval $[0, 1]$:

$$\text{Tissue Versatility} = \frac{\cos(\varphi) - \frac{1}{\sqrt{n}}}{1 - \frac{1}{\sqrt{n}}}$$

This maps the cosine values into the interval $[0, 1]$:

- 1: perfectly uniform expression (maximum versatility)
- 0: expression in only one tissue (minimum versatility)

Tissue Specificity
Tissue specificity is defined as the complement of normalized tissue versatility:

$$\text{Tissue Specificity} = 1 - \text{Tissue Versatility}$$

Values close to 1 indicate high tissue-specificity; values close to 0 indicate broad, uniform expression.

2 Error Handling

In case you encounter errors while using TOX, the following error codes may help you identify the issue. The error codes are grouped into categories based on their nature, such as I/O errors, format validation errors, memory errors, and internal errors.

Table 1: TOX Error Codes Reference

`Error Code`	**Description**
`0`	Success
1xx: I/O / File Reading Errors	
`101`	Could not open file.
`102`	Could not read magic number.
`103`	Could not read array type code.
`104`	Could not read number of dimensions.
`105`	Could not read array dimensions.
`106`	Could not read character length.
`107`	Could not read array data.
2xx: Format / Input Validation Errors	
`200`	Invalid format detected (e.g., magic mismatch or corrupt format).
`201`	Invalid input provided.
`202`	Empty input arrays provided (e.g., zero-length arrays).
`203`	Dimension mismatch detected (shape inconsistencies).
`204`	NaN or Inf found in input data where not allowed.
`205`	Unsupported data type encountered (e.g., complex types).

Continued on next page

Table 1 – continued from previous page

Error Code	Description
3xx: Memory Errors	
301	Memory allocation failed.
302	Null pointer reference encountered.
5xxx: Fortran Runtime / Unit State Errors	
5002	Fortran runtime error: unit not open or not connected.
9xxx: Internal / Unknown Errors	
9001	Internal error: unexpected state or invariant violated.
9999	Unknown error (fallback for unexpected errors).

3 Implementation Notes

This section details the practical pipeline for using the TOX library to read and process tabular data. The workflow is consistent across Fortran, Python, and R, starting with reading a CSV file into a staging area of strings and then converting columns to their proper data types.

3.1 Fortran Pipeline

1. **Normalization of Expression Values**

   The subroutine runs a complete normalization pipeline on gene expression data. It first applies standard deviation normalization, then quantile normalization, followed by replicate averaging, and finally a log2 transformation. The result is returned in the `buf_log` array, which contains the fully normalized expression values. Additionally intermediate results of the performed calculations are available in output buffer arrays. Call the `normalization_pipeline` subroutine with the following arguments:

   Input arguments:

   - `n_genes integer(int32)`: Total number of genes
   - `n_tissues integer(int32)`: Number of axes (tissues / dimensions)
   - `input_matrix real(real64)`: Expression vector matrix with size `n_genes` × `n_tissues`
   - `max_stack integer(int32)`: Stack size for quicksort
   - `group_s integer(int32)`: Start column index for each replicate group (`n_grps`)
   - `group_c integer(int32)`: Number of columns per replicate group (`n_grps`)
   - `n_grps integer(int32)`: Number of replicate groups

   Output arguments:

   - `buf_stddev real(real64)`: Buffer for std dev normalization (`n_genes` * `n_tissues`)
   - `buf_quant real(real64)`: Buffer for quantile normalization (`n_genes` * `n_tissues`)
   - `buf_avg real(real64)`: Buffer for replicate averaging (`n_genes` * `n_grps`)
   - `buf_log real(real64)`: Buffer for log2(x+1) transformation (`n_genes` * `n_grps`)
   - `temp_col real(real64)`: Temporary column vector for sorting (`n_genes`)

- `rank_means real(real64)`: Buffer for rank means (`n_genes`)
- `perm integer(int32)`: Permutation vector for sorting (`n_genes`)
- `stack_left integer(int32)`: Left stack for quicksort (`max_stack`)
- `stack_right integer(int32)`: Right stack for quicksort (`max_stack`)
- `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 1: Normalization Pipeline

```
call normalization_pipeline(n_genes, n_tissues, input_matrix,
   buf_stddev, buf_quant, buf_avg, buf_log, temp_col, rank_means, perm
   , stack_left, stack_right, max_stack, group_s, group_c, n_grps,
   ierr)
```

Alternatively you can calculate each step of the normalization pipeline manually by calling the following subroutines in sequence:

(a) **Normalize by Standard Deviation**

The subroutine normalizes each gene's expression vector using `sqrt(mean(x^2))` across tissues (not classical standard deviation). Call the `normalize_by_std_dev` subroutine with the following arguments:

Input arguments:

- `n_genes integer(int32)`: Number of genes (rows)
- `n_tissues integer(int32)`: Number of tissues (columns)
- `input_matrix real(real64)`: Flattened input matrix of gene expression values (column-major)

Output arguments:

- `output_matrix real(real64)`: Output normalized matrix (same shape as input)
- `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 2: Standard Deviation Normalization

```
call normalize_by_std_dev(n_genes, n_tissues, input_matrix,
   output_matrix, ierr)
```

(b) **Quantile Normalization**

The subroutine performs quantile normalization of a gene expression matrix (F42-compliant) and computes average expression per rank across tissues. Call the `quantile_normalization` subroutine with the following arguments:

Input arguments:

- `n_genes integer(int32)`: Number of genes (rows)
- `n_tissues integer(int32)`: Number of tissues (columns)
- `input_matrix real(real64)`: Flattened input matrix (column-major)
- `max_stack integer(int32)`: Stack size passed from R

Output arguments:

- `output_matrix real(real64)`: Output normalized matrix (same shape as input)
- `temp_col real(real64)`: Temporary vector for column sorting (size `n_genes`)
- `rank_means real(real64)`: Preallocated vector to store rank means (size `n_genes`)
- `perm integer(int32)`: Permutation vector (size `n_genes`)
- `stack_left integer(int32)`: Manual quicksort stack ($\geq log2($`n_genes`$) + 10$)
- `stack_right integer(int32)`: Manual quicksort stack (same size as stack_left)
- `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 3: Quantile Normalization

```
call quantile_normalization(n_genes, n_tissues, input_matrix,
   output_matrix, temp_col, rank_means, perm, stack_left,
   stack_right, max_stack, ierr)
```

(c) **Calculate Tissue Averages**

The subroutine calculates tissue averages by averaging replicates within each group. For each group of tissue replicates, it computes the average expression per gene. The input matrix is column-major, flattened as a 1D array. Call the `calc_tiss_avg` subroutine with the following arguments:

Input arguments:

- `n_gene integer(int32)`: Number of genes (rows)
- `n_grps integer(int32)`: Number of tissue groups
- `group_s integer(int32)`: Start column index for each group (length: `n_grps`)
- `group_c integer(int32)`: Number of columns per group (length: `n_grps`)
- `input_matrix real(real64)`: Flattened input matrix (length: `n_gene` * sum(`group_c`))

Output arguments:

- `output_matrix real(real64)`: Flattened output matrix (length: `n_gene` * `n_grps`)
- `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 4: Tissue Averages Calculation

```
call calc_tiss_avg(n_gene, n_grps, group_s, group_c, input_matrix,
    output_matrix, ierr)
```

(d) **Log2 Transformation**

The subroutine applies `log2(x + 1)` transformation to each element of the input matrix. This is computed via `log(x + 1) / log(2)` for compatibility with WebAssembly (WASM). Call the `log2_transformation` subroutine with the following arguments:

Input arguments:

- `n_genes integer(int32)`: Number of genes (rows)
- `n_tissues integer(int32)`: Number of tissues (columns)
- `input_matrix real(real64)`: Flattened input matrix (size: `n_genes` * `n_tissues`)

Output arguments:

- `output_matrix real(real64)`: Output matrix (same size as input)

- `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 5: Log2 Transformation

```fortran
call log2_transformation(n_genes, n_tissues, input_matrix,
   output_matrix, ierr)
```

2. **Calculate Fold Change**

   If fold change calculation is needed, this subroutine can be used after normalization. The subroutine calculates `log2` fold changes between condition and control columns. For each control-condition pair, this subroutine computes the `log2` fold change by subtracting the expression value in the control column from the corresponding value in the condition column, for all genes. Call the `calc_fchange` subroutine with the following arguments:

   Input arguments:

   - `n_genes integer(int32)`: Total number of genes
   - `n_cols integer(int32)`: Number of columns in the input matrix
   - `n_pairs integer(int32)`: Number of control-condition pairs
   - `control_cols integer(int32)`: Control column indices (length `n_pairs`)
   - `cond_cols integer(int32)`: Condition column indices (length `n_pairs`)
   - `i_matrix real(real64)`: Input matrix, flattened (length: `n_genes` $\times$ `n_cols`)

   Output arguments:

   - `o_matrix real(real64)`: Output matrix for fold changes (length: `n_genes` $\times$ `n_pairs`)
   - `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 6: Fold Change Calculation

```fortran
call calc_fchange(n_genes, n_cols, n_pairs, control_cols, cond_cols,
   i_matrix, o_matrix, ierr)
```

3. **Calculation of family Centroids**

   The subroutine computes the centroid (mean expression vector) for each gene family in a gene expression dataset. It iterates over all families, selects the relevant genes for each family and calculates the mean vector for each family. There are two possible modes to use. To use all genes for calculation, select `all` mode. If you want to specify relevant genes for calculation, select `orthologs` mode. It is important that you provide a logical `ortholog_set` array when using `orthologs` mode. Call the `group_centroid` subroutine with the following arguments:

   Input arguments:

   - `expression_vectors real(real64)`: Expression vector matrix with size `n_axes` $\times$ `n_genes`
   - `n_axes integer(int32)`: Number of axes (tissues / dimensions)
   - `n_genes integer(int32)`: Total number of genes

- `gene_to_family integer(int32)`: Gene-to-family mapping array with the length of `n_genes` (1-based indexing)
- `n_families integer(int32)`: Total number of gene families
- `mode character(len=*)`: Calculation mode (”all” or ”orthologs”)
- (Optional) `ortholog_set logical`: Logical array indicating if a gene is part of the calculation subset (only used in `orthologs` mode)

Output arguments:

- `centroid_matrix real(real64)`: Matrix of calculated family centroid vectors with size `n_axes` × `n_families`
- `selected_indices integer(int32)`: Array of gene indices used for the calculation with length of `n_genes`
- `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 7: Calculation of Family Centroids

```
call group_centroid(expression_vectors, n_axes, n_genes,
   gene_to_family, n_families, centroid_matrix, mode, selected_indices
   , ierr, ortholog_set)
```

(a) **Calculate Mean Vector**

Alternatively you can calculate a single mean vector for a given set of vectors by calling the `mean_vector` subroutine directly with the following arguments:

Input arguments:

- `expression_vectors real(real64)`: Expression vector matrix with size `n_axes` × `n_genes`
- `n_axes integer(int32)`: Number of axes (tissues / dimensions)
- `n_genes integer(int32)`: Total number of genes
- `gene_indices integer(int32)`: An array containing the column indices of the selected genes in `expression_vectors`
- `n_selected_genes integer(int32)`: Total number of genes in the current family to be calculated

Output arguments:

- `centroid real(real64)`: Calculated mean vector (centroid) with length of `n_axes`
- `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 8: Mean Vector Calculation

```
call mean_vector(expression_vectors, n_axes, n_genes, gene_indices
   , n_selected_genes, centroid, ierr)
```

4. **Calculate Distance to Centroid**

   For each gene in the expression vector matrix, the subroutine computes the Euclidean distance to its corresponding family centroid and returns a distances array containing the distance value for each expression vector to its family centroid. Call the `distance_to_centroid` subroutine with the following arguments:

   Input arguments:

   - `n_genes integer(int32)`: Total number of genes
   - `n_families integer(int32)`: Total number of gene families
   - `genes real(real64)`: Gene expression matrix (`d` × `n_genes`)
   - `centroids real(real64)`: Family centroid matrix (`d` × `n_genes`)
   - `gene_to_fam integer(int32)`: Gene-to-family mapping array with the length of `n_genes` (1-based indexing)
   - `d integer(int32)`: Expression vector dimension

   Output arguments:

   - `distances real(real64)`: Array of distances for each gene with length of `n_genes`

   Listing 9: Distance to Centroid Calculation

```
call distance_to_centroid(n_genes, n_families, genes, centroids,
   gene_to_fam, distances, d)
```

   (a) **Calculate Euclidean Distance**

   Alternatively you can calculate a single Euclidean distance between two vectors manually by calling `euclidean_distance` subroutine with the following arguments:

   Input arguments:

   - `vec1 real(real64):` First expression vector as array with length of dimension `d`
   - `vec2 real(real64):` Second expression vector as array with length of dimension `d`
   - `d integer(int32):` Dimension value

   Output arguments:

   - `result real(real64):` Euclidean distance value between `vec1` and `vec2`

   Listing 10: Euclidean distance calculation

```
call euclidean_distance(vec1, vec2, d, result)
```

5. **Detect Gene Outliers**

   The subroutine first computes family scaling, then calculates relative distance indices (RDI) and identifies outliers based on the specified percentile threshold. It returns a boolean array indicating whether each gene is an outlier. Call the `detect_outliers` subroutine with the following arguments:

   Input arguments:

   - `n_genes integer(int32)`: Total number of genes

- `n_families integer(int32)`: Total number of gene families
- `distances real(real64)`: Array of distances for each gene to its centroid with length of `n_genes`
- `gene_to_fam integer(int32)`: Gene-to-family mapping array with the length of `n_genes` (1-based indexing)
- **(Optional)** `percentile real(real64)`: Percentile threshold value for outlier detection (default is 95)

Input / Output arguments:

- `work_array real(real64)`: Work array for sorting with length of `n_genes`
- `perm integer(int32)`: Permutation array for sorting with length of `n_genes`
- `stack_left integer(int32)`: Stack array for left indices during sorting with length of `n_genes`
- `stack_right integer(int32)`: Stack array for right indices during sorting with length of `n_genes`
- `loess_x real(real64)`: Reference x-coordinates for loess smoothing with length of `n_families`
- `loess_y real(real64)`: Reference y-coordinates for loess smoothing with length of `n_families`
- `loess_n integer(int32)`: Indices of reference points used for smoothing with length of `n_families`

Output arguments:

- `is_outlier logical`: Boolean array with length of `n_genes` indicating whether each gene is an outlier
- `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 11: Gene Outlier Calculation

```
call detect_outliers(n_genes, n_families, distances, gene_to_fam,
    work_array, perm, stack_left, stack_right, is_outlier, loess_x,
    loess_y, loess_n, ierr, percentile)
```

Alternatively you can calculate each step of `detect_outliers` manually with the following subroutine:

(a) **Compute Family Scaling**

The subroutine calculates a scaling factor for each gene family by computing the median and standard deviation of gene-to-centroid distances within each family, then applies LOESS smoothing to these values. Call the `compute_family_scaling` subroutine with the following arguments:

Input arguments:

- `n_genes integer(int32)`: Total number of genes
- `n_families integer(int32)`: Total number of gene families

- `distances real(real64)`: Array of distances for each gene to its centroid with length of `n_genes`
- `gene_to_fam integer(int32)`: Gene-to-family mapping array with the length of `n_genes` (1-based indexing)

Input / Output arguments:

- `loess_x real(real64)`: Reference x-coordinates for loess smoothing with length of `n_families`
- `loess_y real(real64)`: Reference y-coordinates for loess smoothing with length of `n_families`
- `indices_used integer(int32)`: Indices of reference points used for smoothing with length of `n_families`
- `perm_tmp integer(int32)`: Permutation array for sorting with length of `n_genes`

Output arguments:

- `dscale real(real64)`: Array of scaling factors per family with length of `n_families`
- `family_distances real(real64)`: Temporary work array with length of `n_genes` to store distances of genes within each family during calculation
- `ierr integer(int32)`: Error code (for error details please check error handling section)

Listing 12: Compute Family Scaling

```
call compute_family_scaling(n_genes, n_families, distances,
    gene_to_fam, dscale, loess_x, loess_y, indices_used, perm_tmp,
    stack_left_tmp, stack_right_tmp, family_distances, ierr)
```

(b) **Compute Family Scaling Helper**

This helper subroutine allocates internal arrays for easier usage and calls `compute_family_scaling`. Call the `compute_family_scaling_alloc` subroutine with the following arguments:

Input arguments:

- `n_genes integer(int32)`: Total number of genes
- `n_families integer(int32)`: Total number of gene families
- `distances real(real64)`: Array of distances for each gene to its centroid with length of `n_genes`
- `gene_to_fam integer(int32)`: Gene-to-family mapping array with the length of `n_genes` (1-based indexing)

Input / Output arguments:

- `loess_x real(real64)`: Reference x-coordinates for loess smoothing with length of `n_families`
- `loess_y real(real64)`: Reference y-coordinates for loess smoothing with length of `n_families`
- `indices_used integer(int32)`: Indices of reference points used for smoothing with length of `n_families`

Output arguments:

- `dscale real(real64)`: Array of scaling factors per family with length of `n_families`

- ierr integer(int32): Error code (for error details please check error handling section)

Listing 13: Compute Family Scaling

```
call compute_family_scaling_alloc(n_genes, n_families, distances,
    gene_to_fam, dscale, loess_x, loess_y, indices_used, ierr)
```

(c) **Compute Relative Distance Index (RDI)**

The subroutine calculates the Relative Distance Index (RDI) for each gene by dividing its Eculidean distance to the family centroid by the scaling factor of its family. Call the compute_rdi subroutine with the following arguments:

Input arguments:

- n_genes integer(int32): Total number of genes
- distances real(real64): Array of distances for each gene to its centroid with length of n_genes
- gene_to_fam integer(int32): Gene-to-family mapping array with the length of n_genes (1-based indexing)
- dscale real(real64): Array of scaling factors per family with length of n_families

Input / Output arguments:

- sorted_rdi real(real64): Work array for sorting with length of n_genes
- perm integer(int32): Pre-initialized (1: n_genes) permutation array for sorting with length of n_genes
- stack_left integer(int32): Stack array for sorting with length of n_genes
- stack_right integer(int32): Stack array for sorting with length of n_genes

Output arguments:

- rdi real(real64: Array of RDI values for each gene with length of n_genes

Listing 14: Compute Relative Distance Index (RDI)

```
call compute_rdi(n_genes, distances, gene_to_fam, dscale, rdi,
    sorted_rdi, perm, stack_left, stack_right)
```

(d) **Identify Outliers**

The subroutine identifies outliers based on the top percentile of RDI values. Call the identify_outliers subroutine with the following arguments:

Input arguments:

- n_genes integer(int32): Total number of genes
- rdi real(real64: Array of RDI values for each gene with length of n_genes
- sorted_rdi real(real64: Array of sorted RDI values for each gene with legnth of n_genes - **this array must be filtered to remove negatives and be sorted in ascending order!**
- **(Optional)** percentile real(real64): Percentile threshold value for outlier detection (default is 95)

Output arguments:

- is_outlier logical: Boolean array with length of n_genes indicating whether each gene is an outlier

- `threshold`: Actual threshold value used for outlier detection

Listing 15: Identify Outliers

```
call identify_outliers(n_genes, rdi, sorted_rdi, is_outlier,
   threshold, percentile)
```

6. **Compute Normalized Tissue Versatility**

   The subroutine calculates a normalized tissue versatility score for selected gene expression vectors. It is based on the angle between each gene expression vector and the space diagonal. Versatility is normalized to [0, 1], where 0 means uniform expression 1 means expression in only one axis. Call the `compute_tissue_versatility` subroutine with the following parameters:

   Input arguments:

   - `n_genes integer(int32)`: Total number of genes
   - `n_vectors integer(int32)`: Total number of expression vectors
   - `n_selected_axes integer(int32)`: Total number of selected axes
   - `n_selected_vectors integer(int32)`: Total number of selected vectors
   - `expression_vectors real(real64)`: Expression vector matrix with size `n_axes` × `n_vectors`
   - `exp_vecs_selection_index logical`: Logical array with length of `n_vectors` to specify which vectors should be included in the calculation
   - `axes_selection logical`: Logical array with length of `n_axes` to specify which axes should be included in the calculation

   Output arguments:

   - `tissue_versatilities real(real64)`: Array with the length of `n_selected_vectors` containing the calculated tissue versatilities
   - `tissue_angles_deg real(real64)`: Array with the length of `n_selected_vectors` containing the calculated angles in degrees
   - `ierr integer(int32)`: Error code (for error details please check error handling section)

   Listing 16: Tissue Versatility Calculation

```
call compute_tissue_versatility(n_axes, n_vectors, expression_vectors,
    exp_vecs_selection_index, n_selected_vectors, axes_selection,
   n_selected_axes, tissue_versatilities, tissue_angles_deg, ierr)
```

7. **Binary Search Trees for 1D Range Queries**

   Binary Search Trees provide efficient indexing and range query capabilities for one-dimensional data using flat array implementations for optimal memory performance.

   Fortran BST Implementation

   (a) **Build BST Index**

   Constructs a Binary Search Tree index by sorting the input values and storing permutation indices. Uses an efficient stack-based sorting algorithm.

   Input:

- `values real(real64)`: 1D array of values to index
- `num_values integer(int32)`: Number of elements

Output:

- `sorted_indices integer(int32)`: Permutation indices after sorting (1-based)
- `left_stack, right_stack integer(int32)`: Sorting workspace
- `ierr integer(int32)`: Error code

```
call build_bst_index(values, num_values, sorted_indices,
   left_stack, right_stack, ierr)
```

(b) **Get Sorted Value**

Retrieves the value at a specific position in the sorted order defined by the BST index.

Input:

- `values real(real64)`: Original values array
- `sorted_indices integer(int32)`: BST index array (1-based)
- `position integer(int32)`: Position in sorted order (1-based)

Output:

- `sorted_value real(real64)`: Value at specified position
- `ierr integer(int32)`: Error code

```
sorted_value = get_sorted_value(values, sorted_indices, position,
   ierr)
```

(c) **BST Range Query**

Efficiently finds all values within a specified range using the precomputed BST index.

Input:

- `values real(real64)`: Original values array
- `sorted_indices integer(int32)`: BST index array (1-based)
- `num_values integer(int32)`: Number of elements
- `lower_bound, upper_bound real(real64)`: Range boundaries

Output:

- `output_indices integer(int32)`: Indices of matching values (1-based)
- `num_matches integer(int32)`: Number of matches found
- `ierr integer(int32)`: Error code

```
call bst_range_query(values, sorted_indices, num_values,
   lower_bound, upper_bound, &
                    output_indices, num_matches, ierr)
```

8. **k-d Trees for Multi-Dimensional Indexing**

k-d trees provide efficient space partitioning for multi-dimensional data with support for both Euclidean and spherical geometries.

Fortran k-d Tree Implementation

(a) **Build k-d Tree Index**

Constructs a k-d tree index using stack-based non-recursive partitioning along specified dimensions.

Input:

- `points real(real64)`: Data points matrix (dimensions × points)
- `num_dimensions integer(int32)`: Number of dimensions
- `num_points integer(int32)`: Number of points
- `dimension_order integer(int32)`: Dimension ordering for splitting (1-based)

Output:

- `kd_indices integer(int32)`: Output index array (1-based)
- `workspace integer(int32)`: Workspace array
- `value_buffer real(real64)`: Value buffer for sorting
- `permutation integer(int32)`: Permutation array
- `left_stack, right_stack integer(int32)`: Sorting stacks
- `recursion_stack integer(int32)`: Recursion stack
- `ierr integer(int32)`: Error code

```
call build_kd_index(points, num_dimensions, num_points, kd_indices
   , dimension_order, &
                    workspace, value_buffer, permutation, left_stack
                       , right_stack, recursion_stack, ierr)
```

(b) **Build Spherical k-d Tree**

Specialized k-d tree construction optimized for unit vectors on hyperspheres.

Input:

- `vectors real(real64)`: Unit vectors matrix (dimensions × vectors)
- `num_dimensions integer(int32)`: Number of dimensions
- `num_vectors integer(int32)`: Number of vectors
- `dimension_order integer(int32)`: Dimension ordering for splitting (1-based)

Output:

- `sphere_indices integer(int32)`: Output index array (1-based)
- Various workspace arrays (same as build_kd_index)
- `ierr integer(int32)`: Error code

```
call build_spherical_kd(vectors, num_dimensions, num_vectors,
   sphere_indices, dimension_order, &
                        workspace, value_buffer, permutation,
                           left_stack, right_stack, recursion_stack,
                            ierr)
```

(c) **Get Point from k-d Index**

Retrieves point coordinates from specified position in k-d tree ordering.

Input:

- `points real(real64)`: Original points matrix
- `kd_indices integer(int32)`: k-d tree index array (1-based)

- `position integer(int32)`: Position in index (1-based)

Output:

- `point_values real(real64)`: Retrieved point values
- `ierr integer(int32)`: Error code

```
call get_kd_point(points, kd_indices, position, point_values, ierr
   )
```

9. **Array Serialization and Deserialization Module**

   The array serialization and deserialization module provides comprehensive routines for efficiently storing and retrieving multi-dimensional arrays of various data types.

   (a) **General Architecture**

   The module supports three data types (integer, real, and character) across dimensions 1 through 5. All routines follow a consistent naming convention and parameter structure for ease of use.

   (b) **Supported Functions**

   - `serialize_int_Nd(arr, filename, ierr)` - Serialize N-dimensional integer arrays
   - `deserialize_int_Nd(arr, filename, ierr)` - Deserialize N-dimensional integer arrays
   - `serialize_real_Nd(arr, filename, ierr)` - Serialize N-dimensional real arrays
   - `deserialize_real_Nd(arr, filename, ierr)` - Deserialize N-dimensional real arrays
   - `serialize_char_Nd(arr, filename, ierr)` - Serialize N-dimensional character arrays
   - `deserialize_char_Nd(arr, filename, ierr)` - Deserialize N-dimensional character arrays
   - `get_array_metadata(filename, dims, max_dims, ndims, ierr, [clen])` - Retrieve array metadata from files

   Where N represents the array dimension (1-5).

   (c) **Detailed Parameter Specifications**

   Serialization Functions Parameters:

   - `arr` (input): The array to be serialized.
     Type: Allocatable array of type `integer(int32)`, `real(real64)`, or `character(len=clen)`
     Dimensions: 1D to 5D
     Intent: `in`
     Note: *clen* has to be the size of the arrays elements
   - `filename` (input): The name of the binary file to create.
     Type: `character(len=*)`
     Intent: `in`
   - `ierr` (output): Error code indicating operation status.
     Type: `integer(int32)`
     Intent: `out`
     Values: 0 indicates success, non-zero values indicate specific error conditions

Listing 17: Example usage of a serialize function

```
call serialize_int_2d(array, "test_file.bin", ierr)
```

(d) **Metadata Retrieval Function Parameters:**

- `filename` (input): The name of the binary file to read metadata from.
  Type: `character(len=*)`
  Intent: `in`
- `dims` (output): Array containing the dimensions of the stored array.
  Type: `integer(int32), dimension(max_dims)`
  Intent: `out` **Note:** Must be preallocated, recommended with size 5
- `max_dims` (input): Size of the dims array (typically 5 for maximum supported dimensions).
  Type: `integer(int32)`
  Intent: `in`
- `ndims` (output): Number of dimensions actually used by the array.
  Type: `integer(int32)`
  Intent: `out`
- `ierr` (output): Error code indicating operation status.
  Type: `integer(int32)`
  Intent: `out`
  Values: 0 indicates success, non-zero values indicate specific error conditions
- `clen` (output, optional): For character arrays, the length of character elements.
  Type: `integer(int32)`
  Intent: `out`
  Note: This parameter is only used and required for character arrays

Listing 18: Usage of metadata retrieval

```
call get_array_metadata(filename, dims, max_dims, ndims, ierr,
    clen)
```

Deserialization Functions Parameters:

- `arr` (output): The array to be populated with deserialized data.
  Type: Allocatable array of type `integer(int32)`, `real(real64)`, or `character(len=clen)`
  Dimensions: 1D to 5D
  Intent: `out`
  Note: The array must be allocated with correct dimensions and correct element length before calling deserialization routines
- `filename` (input): The name of the binary file to read.
  Type: `character(len=*)`
  Intent: `in`
- `ierr` (output): Error code indicating operation status.
  Type: `integer(int32)`
  Intent: `out`
  Values: 0 indicates success, non-zero values indicate specific error conditions

Listing 19: Example usage of a deserialize function

```
call deserialize_int_2d(array, "test_file.bin", ierr)
```

Listing 20: Full example

```fortran
integer(int32), allocatable :: kallisto_expression(:,:),
   kallisto_expression_2(:,:)
integer(int32) :: dims(5)
integer(int32) :: ndims, ierr, max_dims
character(len=64) :: fname

!Assume array is filled with data

call serialize_real_2d(kallisto_expression, "kallisto_data_v1.bin"
   , ierr)
if(.not. is_ok(ierr)) error stop ierr

!Reading the data again
call get_array_metadata("kallisto_data_v1.bin", dims, max_dims,
   ndims, ierr)
if(.not. is_ok(ierr)) error stop ierr
allocate(kallisto_expression_2(dims(1), dims(2))

call deserialize_real_2d(kallisto_expression_2, "kallisto_data_v1.
   bin", ierr)
if(.not. is_ok(ierr)) error stop ierr
```

(e) **Error Handling**

All functions return an error code through the `ierr` parameter. The error handling utility tox errors provides functions to check and interpret these codes.

(f) **Implementation Notes**

- All routines are implemented in standard Fortran for interoperability
- The module uses allocatable arrays for flexible memory management
- File operations use Fortran's native unformatted I/O for efficiency
- Error handling is consistent across all functions for predictable behavior

3.2 Python Pipeline

The Python workflow is exposed through the `tensoromics_functions.py` module. The functions provide a high-level interface that mirrors the Fortran pipeline.

1. **Normalization of Expression Values**

   The function runs a complete normalization pipeline on gene expression data. It first applies standard deviation normalization, then quantile normalization, followed by replicate averaging, and finally a log2 transformation. The result is returned in an array, which contains the fully normalized expression values. Call the `tox_normalization_pipeline` function with the following arguments:

   - `input_matrix`: Numeric expression vector matrix with size `genes` × `tissues`
   - `group_starts`: Integer array, start column index for each replicate group (1-based)
   - `group_counts`: Integer array, number of columns per replicate group

   Return:

- Numpy array with log2(x+1) normalized expression with size (genes × groups)

Listing 21: Python Normalization Pipeline

```
tox_normalization_pipeline(input_matrix, group_starts, group_counts)
```

Alternatively you can calculate each step of the normalization pipeline manually by calling the following functions in sequence:

(a) **Normalize by Standard Deviation**

The function normalizes each gene's expression vector using `sqrt(mean(x^2))` across tissues (not classical standard deviation). Call the `tox_normalize_by_std_dev` function with the following arguments:

- `input_matrix`: A numeric matrix with genes as rows and tissues as columns

Returns:

- Numpy array: Contains a normalized matrix with same dimensions as input

Listing 22: Python Standard Deviation Normalization

```
tox_normalize_by_std_dev(input_matrix)
```

(b) **Quantile Normalization**

The function performs quantile normalization of a gene expression matrix (F42-compliant) and computes average expression per rank across tissues. Call the `tox_quantile_normalization` function with the following arguments:

- `input_matrix`: A numeric matrix with genes as rows and tissues as columns

Returns:

- Numpy array: Quantile-normalized matrix with same dimensions as input

Listing 23: Python Quantile Normalization

```
tox_quantile_normalization(input_matrix)
```

(c) **Calculate Tissue Averages**

The function calculates tissue averages by averaging replicates within each group. For each group of tissue replicates, it computes the average expression per gene. The input matrix is column-major, flattened as a 1D array. Call the `tox_calculate_tissue_averages` function with the following arguments:

- `input_matrix`: A numeric matrix with genes as rows and tissue replicates as columns
- `group_starts`: Array of starting column indices for each group (1-based for Fortran)
- `group_counts`: Array of counts for each group

Returns:

- Numpy array: Matrix with genes as rows and averaged tissues as columns

Listing 24: Python Tissue Averages Calculation

```
tox_calculate_tissue_averages(input_matrix, group_starts,
    group_counts)
```

(d) **Log2 Transformation**

The function applies `log2(x + 1)` transformation to each element of the input matrix. This is computed via `log(x + 1) / log(2)` for compatibility with WebAssembly (WASM). Call the `tox_log2_transformation` function with the following arguments:

- `input_matrix`: A numeric matrix with genes as rows and tissues as columns

Returns:

- Numpy array: Log2-transformed matrix with same dimensions as input

Listing 25: Python Log2 Transformation

```
tox_log2_transformation(input_matrix)
```

2. **Calculate Fold Change**

If fold change calculation is needed, this function can be used after normalization. The function calculates `log2` fold changes between condition and control columns. For each control-condition pair, this function computes the `log2` fold change by subtracting the expression value in the control column from the corresponding value in the condition column, for all genes. Call the `tox_calculate_fold_changes` function with the following arguments:

- `input_matrix`: A numeric matrix with genes as rows and tissues/conditions as columns
- `control_cols`: Array of control column indices (1-based for Fortran)
- `condition_cols`: Array of condition column indices (1-based for Fortran)

Returns:

- Numpy array: Matrix with genes as rows and fold change values as columns

Listing 26: Python Fold Change Calculation

```
tox_calculate_fold_changes(input_matrix, control_cols, condition_cols)
```

3. **Calculation of family Centroids**

The function computes the centroid (mean expression vector) for each gene family in a gene expression dataset. It iterates over all families, selects the relevant genes for each family and calculates the mean vector for each family. There are two possible modes to use. To use all genes for calculation, select `all` mode. If you want to specify relevant genes for calculation, select `orthologs` mode. It is important that you provide a logical `ortholog_set` array when using `orthologs` mode. Call the `tox_group_centroid` function with the following arguments:

- `expression_vectors`: Expression vector matrix with size `n_axes` × `n_genes`
- `gene_to_family`: Gene-to-family mapping array with the length of `n_genes` (1-based indexing)
- `n_families`: Total number of gene families
- `mode`: Calculation mode ("all" or "orthologs")
- (Optional) `ortholog_set`: Logical array indicating if a gene is part of the calculation subset (only used in `orthologs` mode)

Return:

- `centroids_out`: Matrix of calculated family centroid vectors with size `n_axes` × `n_families` (read-only)

Listing 27: Python Calculation of Family Centroids

```
tox_group_centroid(expression_vectors, gene_to_family, n_families,
    mode, ortholog_set)
```

(a) **Calculate Mean Vector**

Alternatively you can calculate a single mean vector for a given set of vectors by calling the `mean_vector` function directly with the following arguments:

- `expression_vectors real(real64)`: Expression vector matrix with size `n_axes` × `n_genes`
- `gene_indices integer(int32)`: An array containing the column indices of the selected genes in `expression_vectors`

Return:

- `centroid`: Calculated mean vector (centroid) with length of `n_axes` (read-only)

Listing 28: Python Mean Vector Calculation

```
tox_mean_vector(expression_vectors, gene_indices)
```

4. **Calculate Distance to Centroid**

For each gene in the gene expression matrix, the function computes the Euclidean distance to its corresponding family centroid and returns a read-only distances array containing the distance value for each expression vector to its family centroid. Call the `tox_distance_to_centroid` function with the following arguments:

- `genes`: Gene expression matrix array with size of (`n_genes` × `d`)
- `centroids`: Family centroid matrix array with size of (`n_families` × `d`)
- `gene_to_fam`: Gene-to-family mapping array with the length of `n_genes` (0 = no family, >0 = family index)
- `d`: Gene expression matrix dimension as number

Listing 29: Python Distance to Centroid Calculation

```
tox_distance_to_centroid(genes, centroids, gene_to_fam, d)
```

(a) **Calculate Euclidean Distance**

Alternatively you can calculate a single Euclidean distance between two vectors manually by calling `tox_euclidean_distance` function with the following arguments:

- `vec1`: First vector as array
- `vec2`: Second vector as array

Both vectors must have same length!

Listing 30: Python Euclidean distance calculation

```
tox_euclidean_distance(vec1, vec2)
```

5. **Detect Gene Outliers**

   The function first computes family scaling, then calculates relative distance indices (RDI) and identifies outliers based on the specified percentile threshold. It returns a read-only dictionary containing the outlier boolean values and intermediate results. Call the `tox_detect_outliers` function with the following arguments:

   - `distances`: Array of distances for each gene to its centroid
   - `gene_to_fam`: Gene-to-family mapping array with the same length as the distances array (0 = no family, >0 = family index)
   - **(Optional)** `percentile`: Percentile threshold value for outlier detection (default is 95)

   Return dictionary:

   - **outliers**: Boolean array indicating whether each gene in an outlier
   - **loess_x**: Array of x coordinates used as reference points for LOESS smoothing (median distance)
   - **loess_y**: Array of y coordinates used as reference points for LOESS smoothing (standard deviation)
   - **loess_n**: Array with numbers of genes in each family used to compute the corresponding median and standard deviation

   Listing 31: Python Gene Outlier Calculation

   ```
   tox_detect_outliers(distances, gene_to_fam, percentile=95.0)
   ```

   Alternatively you can calculate each step of `tox_detect_outliers` manually with the following functions:

   (a) **Compute Family Scaling**

   The function calculates a scaling factor for each gene family by computing the median and standard deviation of gene-to-centroid distances within each family, then applies LOESS smoothing to these values. Call the `tox_compute_family_scaling` function with the following arguments:

   - `distances`: Array of distances for each gene to its centroid
   - `gene_to_fam`: Gene-to-family mapping array with the same length as the distances array (0 = no family, >0 = family index)

   Return dictionary:

   - **dscale**: Array of scaling factors per family
   - **loess_x**: Array of x coordinates used as reference points for LOESS smoothing (median distance)
   - **loess_y**: Array of y coordinates used as reference points for LOESS smoothing (standard deviation)
   - **indices_used**: Array of indices of families that have been included in the calculation

Listing 32: Python Compute Family Scaling

```
tox_compute_family_scaling(distances, gene_to_fam)
```

(b) **Compute Family Scaling (Expert Version)**

This function is the expert version of `tox_detect_outliers` and requires pre-allocated work arrays for maximum performance and control. It can be used when you need fine-grained control over memory allocation or you are calling this function many times in a tight loop. Call the `tox_compute_family_scaling_expert` function with the following arguments:

- `distances`: Array of distances for each gene to its centroid
- `gene_to_fam`: Gene-to-family mapping array with the same length as the distances array (0 = no family, >0 = family index)
- `perm_tmp`: Permutation array for sorting with the same length as the distances array
- `stack_left_tmp`: Stack array for sorting with the same length as the distances array
- `stack_right_tmp`: Stack array for sorting with the same length as the distances array
- `family_distances`: Work array for sorting with the same length as the distances array

Return dictionary:

- **dscale**: Array of scaling factors per family
- **loess_x**: Array of x coordinates used as reference points for LOESS smoothing (median distance)
- **loess_y**: Array of y coordinates used as reference points for LOESS smoothing (standard deviation)
- **indices_used**: Array of indices of families that have been included in the calculation
- **perm_tmp**: Permutation array for sorting
- **stack_left_tmp**: Stack array for sorting
- **stack_right_tmp**: Stack array for sorting
- **family_distances**: Work array for sorting

Listing 33: Python Compute Family Scaling (Expert Version)

```
tox_compute_family_scaling_expert(distances, gene_to_fam, perm_tmp
    , stack_left_tmp, stack_right_tmp, family_distances)
```

(c) **Compute Relative Distance Index (RDI)**

The function calculates the Relative Distance Index (RDI) for each gene by dividing its Euclidean distance to the family centroid by the scaling factor of its family. The function returns a numpy array with RDI values for each gene. Call the `tox_compute_rdi` function with the following arguments:

- `distances`: Array of distances for each gene to its centroid
- `gene_to_fam`: Gene-to-family mapping array with the same length as the distances array (0 = no family, >0 = family index)
- **dscale**: Array of scaling factors per family

Listing 34: Python Compute Relative Distance Index (RDI)

```
tox_compute_rdi(distances, gene_to_fam, dscale)
```

(d) **Identify Outliers**

The function identifies outliers based on the top percentile of RDI values. Call the `tox_identify_outliers` function with the following arguments:

- `rdi`: Array of RDI values for each gene
- **(Optional)** `threshold`: Fixed RDI threshold number (if None, percentile value is used)
- **(Optional)** `percentile`: Percentile threshold number for outlier detection (default is 95)

Return dictionary:

- `outliers`: Boolean array indicating whether each gene is an outlier
- `threshold`: Actual threshold value used for outlier detection

Listing 35: Python Identify Outliers

```
tox_identify_outliers(rdi, threshold=None, percentile=95.0)
```

6. **Compute Normalized Tissue Versatility:**

The function calculates a normalized tissue versatility score for selected gene expression vectors. It is based on the angle between each gene expression vector and the space diagonal. Versatility is normalized to [0, 1], where 0 means uniform expression 1 means expression in only one axis. Call the `tox_compute_tissue_versatility` function with the following parameters:

- `expression_vectors`: Expression vector matrix array with size `n_axes` $\times$ `n_vectors`
- `vector_selection`: Logical array with length of `n_vectors` to specify which vectors should be included in the calculation
- `axes_selection`: Logical array with length of `n_axes` to specify which axes should be included in the calculation

Return dictionary:

- `tissue_versatilities`: Array containing the calculated tissue versatilities for selected vectors
- `tissue_angles_deg`: Array containing the calculated angles in degrees for selected vectors
- `n_selected_vectors`: Number of vectors processed
- `n_selected_axes`: Number of axes used in calculation

Listing 36: Python Tissue Versatility Calculation

```
tox_calculate_tissue_versatility(expression_vectors, vector_selection,
    axis_selection)
```

7. **Python BST Interface**

   (a) **Build BST Index**
   Python wrapper for building BST index from NumPy arrays. Automatically converts from Fortran 1-based to Python 0-based indexing.
   Input:
   - `values numpy.ndarray`: 1D array of float64 values

   Output:
   - `np.array`: Permutation indices (0-based Python indices)

   ```
   indices = build_bst_index(values)
   ```

   (b) **BST Range Query**
   Performs range queries on BST index. Accepts and returns Python 0-based indices.
   Input:
   - `values numpy.ndarray`: Original values array
   - `indices numpy.ndarray`: BST index array from build_bst_index (0-based)
   - `lower_bound float`: Lower bound of range (inclusive)
   - `upper_bound float`: Upper bound of range (inclusive)

   Output:
   - `tuple`: (matching_indices, count) where indices are 0-based

   ```
   matching_indices, count = bst_range_query(values, indices,
       lower_bound, upper_bound)
   ```

   (c) **Get Sorted Value**
   Note: This function is not implemented as a separate Python function. Use direct array indexing instead:

   ```
   value = values[indices[position]]  # position is 0-based
   ```

8. **Python k-d Tree Interface**

   (a) **Build k-d Tree Index**
   Python interface for constructing k-d trees. Requires Fortran-ordered arrays and returns 0-based indices.
   Input:
   - `points numpy.ndarray`: 2D array of points (dimensions $\times$ n_points) in Fortran order
   - `dimension_order numpy.ndarray`: Dimension ordering (1-based Fortran indices, optional)

   Output:
   - `np.array`: Permutation indices (0-based Python indices)

   ```
   kd_indices = build_kd_index(points, dimension_order)
   ```

(b) **Build Spherical k-d Tree**
Alias for build_kd_index with the same parameters and behavior.
Input: Same as build_kd_index **Output:** Same as build_kd_index

```
sphere_indices = build_spherical_kd(vectors, dimension_order)
```

(c) **Get Point from k-d Index**
Note: This function is not implemented as a separate Python function. Use direct array indexing instead:

```
point = points[:, kd_indices[position]]  # position is 0-based
```

9. **Python Interface for Array Serialization and Deserialization**

The Python interface provides bindings to the Fortran serialization and deserialization routines, enabling efficient handling of multi-dimensional arrays.This interface supports NumPy arrays with native data type compatibility and Fortran-style column-major memory layout.

(a) **General Architecture**
The Python module `tensoromics_functions` provides six main functions for array handling:

- `tox_serialize_int_nd(array, filename)` - Serialize N-dimensional integer arrays
- `tox_deserialize_int_nd(filename)` - Deserialize N-dimensional integer arrays
- `tox_serialize_real_nd(array, filename)` - Serialize N-dimensional real arrays
- `tox_deserialize_real_nd(filename)` - Deserialize N-dimensional real arrays
- `tox_serialize_char_nd(array, filename)` - Serialize N-dimensional character arrays
- `tox_deserialize_char_nd(filename)` - Deserialize N-dimensional character arrays
- `get_array_metadata(filename)` - Retrieve array dimensions and metadata

(b) **Array Functions**
Serialization Functions:

- `array` (input): NumPy array to be serialized
  Type: `numpy.ndarray` with dtype `np.int32` or `np.float64`, for characters: `U5`
  Layout: Column-major (Fortran) order (`order='F'`)
  Dimensions: Since the array is being passed flat, theoretically all dimensions are supported, but practical use cases are typically 1D to 5D
- `filename` (input): Name of the binary file to create
  Type: `str`

Listing 37: Example usage of serialization function

```
tox_serialize_int_nd(int_array, filename)
tox_serialize_real_nd(real_array, filename)
```

Deserialization Functions:

- `filename` (input): Name of the binary file to read
  Type: `str`

- **Return Value:** Deserialized NumPy array
  Type: `numpy.ndarray` with original dtype and dimensions
  Layout: Column-major (Fortran) order

Listing 38: Example usage of deserialization function

```
restored_int = tox_deserialize_int_nd(filename)
restored_real = tox_deserialize_real_nd(filename)
```

(c) **Implementation Details**

- The Python interface uses Fortran-style column-major array ordering for optimal compatibility
- All arrays are converted to the appropriate Fortran data types before serialization
- The interface automatically handles metadata including dimensions and data types
- Error handling is implemented through Python exceptions

(d) **Usage Example**

Listing 39: Full example of serialization and deserialization

```
# Real array example
real_array = np.array([1.5, 2.3, 3.2, 4.0, 5.0], dtype=np.float64,
    order='F')
tox_serialize_real_nd(real_array, "test_real_1d.bin")
restored_real = tox_deserialize_real_nd("test_real_1d.bin")

# Character array example with metadata
char_array = np.asfortranarray(["hello", "world"], dtype='U5')
tox_serialize_char_nd(char_array, "test_char_1d.bin")

restored_char = tox_deserialize_char_nd("test_char_1d.bin")

# Multi-dimensional examples
int_2d = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int32, order='F
   ')
tox_serialize_int_nd(int_2d, "test_int_2d.bin")

restored_2d = tox_deserialize_int_nd("test_int_2d.bin")
```

(e) **Compatibility Notes**

- Requires NumPy installation
- Arrays must use Fortran-style column-major ordering
- Character arrays must use fixed-length string data types
- The interface supports N dimensions, but practical use cases are typically 1D to 5D
- Data types are restricted to int32, float64, and U5 for characters

3.3 R Pipeline

The R workflow is exposed through the `tensoromics_functions.R` module. The functions provide a high-level interface that mirrors the Fortran pipeline.

1. **Normalization of Expression Values**

   The function runs a complete normalization pipeline on gene expression data. It first applies standard deviation normalization, then quantile normalization, followed by replicate averaging, and finally a log2 transformation. The result is returned in an array, which contains the fully normalized expression values. Call the `tox_normalization_pipeline` function with the following arguments:

   - `input_matrix`: Numeric expression vector matrix with size `genes` × `tissues`
   - `group_s`: Integer vector: start column index for each replicate group (1-based)
   - `group_c`: Integer vector: number of columns per replicate group

   Return:

   - Numeric matrix: log2(x+1) normalized expression

   Listing 40: R Normalization Pipeline

   ```
   tox_normalization_pipeline(input_matrix, group_s, group_c)
   ```

   Alternatively you can calculate each step of the normalization pipeline manually by calling the following functions in sequence:

   (a) **Normalize by Standard Deviation**

   The function normalizes each gene's expression vector using `sqrt(mean(x^2))` across tissues (not classical standard deviation). Call the `tox_normalize_by_std_dev` function with the following arguments:

   - `input_matrix`: A numeric matrix with genes as rows and tissues as columns

   Returns:

   - A normalized numeric matrix with the same dimensions and names as the input

   Listing 41: R Standard Deviation Normalization

   ```
   tox_normalize_by_std_dev(input_matrix)
   ```

   (b) **Quantile Normalization**

   The function performs quantile normalization of a gene expression matrix (F42-compliant) and computes average expression per rank across tissues. Call the `tox_quantile_normalization` function with the following arguments:

   - `input_matrix`: A numeric matrix with genes as rows and tissues as columns

   Returns:

   - A quantile-normalized numeric matrix with the same dimensions and names as the input

   Listing 42: R Quantile Normalization

   ```
   tox_quantile_normalization(input_matrix)
   ```

(c) **Calculate Tissue Averages**

The function calculates tissue averages by averaging replicates within each group. For each group of tissue replicates, it computes the average expression per gene. The input matrix is column-major, flattened as a 1D array. Call the `tox_calculate_tissue_averages` function with the following arguments:

- `df`: A data frame or matrix with genes as rows and tissue replicates as columns

Returns:

- A data frame with genes as rows and averaged tissues as columns

Listing 43: R Tissue Averages Calculation

```
tox_calculate_tissue_averages(df)
```

(d) **Log2 Transformation**

The function applies `log2(x + 1)` transformation to each element of the input matrix. This is computed via `log(x + 1) / log(2)` for compatibility with WebAssembly (WASM). Call the `tox_log2_transformation` function with the following arguments:

- `input_matrix`: A numeric matrix with genes as rows and tissues as columns

Returns:

- A numeric matrix with log2-transformed expression values, preserving the same dimensions and names as the input.

Listing 44: R Log2 Transformation

```
tox_log2_transformation(input_matrix)
```

Additionally there are two helpfer functions to improve the data quality and prevent calculation errors:

(a) **Diagnose Data Quality**

The function checks your gene expression matrix for common data quality issues, such as missing values (NA), infinite values, NaNs, zeros, and negative values. It provides a summary of these problems, lists affected genes, and reports basic statistics like minimum, maximum, and mean values. Use this function to quickly assess the quality of your data before running normalization or analysis steps. Call the `tox_diagnose_data_quality` function with the following arguments:

- `input_matrix`: A numeric matrix with genes as rows and tissues as columns
- (Optional) `show_details`: Logical indicating whether to show detailed information (default is TRUE)

Returns:

- A list with diagnostic information about the data quality

Listing 45: R Diagnose Data Quality

```
tox_diagnose_data_quality(input_matrix, show_details)
```

(b) **Clean Data for Normalization**

The function prepares your gene expression matrix for normalization by handling problematic values. It can remove or impute NA, NaN, and Inf values, filter out genes with all zero values, and optionally convert very small values to zero. You can choose different strategies for dealing with missing data, ensuring your matrix is clean and ready for downstream analysis. Call the `tox_clean_data_for_normalization` function with the following arguments:

- `df_matrix`: A numeric matrix with genes as rows and tissues as columns
- (Optional) `remove_all_zero_genes`: Logical, whether to remove genes that are all zeros (default = TRUE)
- (Optional) `na_strategy`: Strategy for handling NA values: "remove_genes", "remove_samples", "impute_zero", "impute_mean" (default = "remove_genes"
- (Optional) `min_expression_threshold`: Minimum expression value to consider (values below this become 0 and default is 0)
- (Optional) `convert_small_to_zero`: Logical, wheter very small numbers should be converted to zero (default = FALSE)

Returns:

- A cleaned matrix ready for normalization

Listing 46: R Clean Data for Normalization

```
tox_clean_data_for_normalization(df_matrix, remove_all_zero_genes,
    na_strategy, min_expression_threshold, convert_small_to_zero)
```

2. **Calculate Fold Change**

If fold change calculation is needed, this function can be used after normalization. The function calculates `log2` fold changes between condition and control columns. For each control-condition pair, this function computes the `log2` fold change by subtracting the expression value in the control column from the corresponding value in the condition column, for all genes. Call the `tox_calculate_fold_changes` function with the following arguments:

- `df`: A data frame with genes as rows and tissues/conditions as columns
- `control_pattern`: A string pattern to detect control columns
- `condition_patterns`: A character vector with patterns to detect condition columns

Returns:

- A data frame with genes as rows and log2 fold change values as columns

Listing 47: R Fold Change Calculation

```
tox_calculate_fc_by_patterns(df, control_pattern, condition_patterns)
```

3. **Calculation of family Centroids**

The function computes the centroid (mean expression vector) for each gene family in a gene expression dataset. It iterates over all families, selects the relevant genes for each family and calculates the mean vector for each family. There are two possible modes to use. To use all genes for calculation, select `all` mode. If you want to specify relevant genes for calculation, select `orthologs` mode. It is important that you provide a logical `ortholog_set` array when using `orthologs` mode. Call the `tox_group_centroid` function with the following arguments:

- `expression_vectors`: Expression vector matrix with size `n_axes` × `n_genes`
- `gene_to_family`: Gene-to-family mapping array with the length of `n_genes` (1-based indexing)
- `n_families`: Total number of gene families
- `mode`: Calculation mode (”all” or ”orthologs”)
- (Optional) `ortholog_set`: Logical array indicating if a gene is part of the calculation subset (only used in `orthologs` mode)

Return:

- `centroid_matrix`: Matrix of calculated family centroid vectors with size `n_axes` × `n_families`

Listing 48: R Calculation of Family Centroids

```
tox_group_centroid(expression_vectors, gene_to_family, n_families,
  mode, ortholog_set)
```

(a) **Calculate Mean Vector**

Alternatively you can calculate a single mean vector for a given set of vectors by calling the `mean_vector` function directly with the following arguments:

- `expression_vectors real(real64)`: Expression vector matrix with size `n_axes` × `n_genes`
- `gene_indices integer(int32)`: An array containing the column indices of the selected genes in `expression_vectors`

Return:

- `centroid`: Calculated mean vector (centroid) with length of `n_axes`

Listing 49: R Mean Vector Calculation

```
tox_mean_vector(expression_vectors, gene_indices)
```

4. **Calculate Distance to Centroid**

For each gene in the gene expression matrix, the function computes the Euclidean distance to its corresponding family centroid and returns a numeric vector of distances from each gene to its family centroid. Call the `tox_distance_to_centroid` function with the following arguments:

- `genes`: Gene expression matrix with size of (`n_genes` × `d`)
- `centroids`: Family centroid matrix with size of (`n_families` × `d`)
- `gene_to_fam`: Gene-to-family mapping integer vector with the length of `n_genes` (0 = no family, >0 = family index)

- d: Gene expression matrix dimension as integer

Listing 50: R Distance to Centroid Calculation

```
tox_distance_to_centroid(genes, centroids, gene_to_fam, d)
```

(a) **Calculate Euclidean Distance**

Alternatively you can calculate a single Euclidean distance between two vectors manually by calling `tox_euclidean_distance` function with the following arguments:

- vec1: First vector
- vec2: Second vector

Both vectors must have same length!

Listing 51: R Euclidean distance calculation

```
tox_euclidean_distance(vec1, vec2)
```

5. **Detect Gene Outliers**

The function first computes family scaling, then calculates relative distance indices (RDI) and identifies outliers based on the specified percentile threshold. It returns a list of components containing the outlier boolean values and intermediate results. Call the `tox_detect_outliers` function with the following arguments:

- `distances`: Numeric vector of distances for each gene to its centroid
- `gene_to_fam`: Gene-to-family mapping integer vector mapping with the same length as the distances vector (0 = no family, >0 = family index)
- `n_families`: Integer number of families
- **(Optional)** `percentile`: Percentile threshold value for outlier detection (default is 95.0)

Return list:

- **is_outlier**: Logical vector indicating whether each gene in an outlier
- **loess_x**: Numeric vector of x coordinates used as reference points for LOESS smoothing (median distance)
- **loess_y**: Numeric vector of y coordinates used as reference points for LOESS smoothing (standard deviation)
- **loess_n**: Integer vector with numbers of genes in each family used to compute the corresponding median and standard deviation

Listing 52: R Gene Outlier Calculation

```
tox_detect_outliers(distances, gene_to_fam, n_families, percentile)
```

Alternatively you can calculate each step of `tox_detect_outliers` manually with the following functions:

(a) **Compute Family Scaling**

The function calculates a scaling factor for each gene family by computing the median and standard deviation of gene-to-centroid distances within each family, then applies LOESS smoothing to these values. Call the `tox_compute_family_scaling` function with the following arguments:

- `distances`: Numeric vector of distances for each gene to its centroid
- `gene_to_fam`: Gene-to-family mapping integer vector mapping with the same length as the distances vector (0 = no family, >0 = family index)
- `n_families`: Integer number of families

Return dictionary:

- **dscale**: Numeric vector of scaling factors per family
- **loess_x**: Numeric vector of x coordinates used as reference points for LOESS smoothing (median distance)
- **loess_y**: Numeric vector of y coordinates used as reference points for LOESS smoothing (standard deviation)
- **indices_used**: Integer vector of indices of families that have been included in the calculation

Listing 53: R Compute Family Scaling

```
tox_compute_family_scaling(distances, gene_to_fam, n_families)
```

(b) **Compute Family Scaling (Expert Version)**

This function is the expert version of `tox_detect_outliers` and requires pre-allocated work arrays for maximum performance and control. It can be used when you need fine-grained control over memory allocation or you are calling this function many times in a tight loop. Call the `tox_compute_family_scaling_expert` function with the following arguments:

- `distances`: Numeric vector of distances for each gene to its centroid
- `gene_to_fam`: Gene-to-family mapping integer vector mapping with the same length as the distances vector (0 = no family, >0 = family index)
- `n_families`: Integer number of families
- `perm_tmp`: Permutation integer vector for sorting with the same length as distances
- `stack_left_tmp`: Stack integer vector for sorting with the same length as distances
- `stack_right_tmp`: Stack integer vector for sorting with the same length as distances
- `family_distances`: Work integer vector for sorting with the same length as distances

Return list:

- **dscale**: Numeric vector of scaling factors per family
- **loess_x**: Numeric vector of x coordinates used as reference points for LOESS smoothing (median distance)
- **loess_y**: Numeric vector of y coordinates used as reference points for LOESS smoothing (standard deviation)
- **indices_used**: Integer vector of indices of families that have been included in the calculation

- **perm_tmp**: Permutation vector for sorting
- **stack_left_tmp**: Stack vector for sorting
- **stack_right_tmp**: Stack vector for sorting
- **family_distances**: Work vector for sorting

Listing 54: R Compute Family Scaling (Expert Version)

```
tox_compute_family_scaling_expert(distances, gene_to_fam, n_
    families, perm_tmp, stack_left_tmp, stack_right_tmp, family_
    distances)
```

(c) **Compute Relative Distance Index (RDI)**

The function calculates the Relative Distance Index (RDI) for each gene by dividing its Euclidean distance to the family centroid by the scaling factor of its family. Call the `tox_compute_rdi` function with the following arguments:

- `distances`: Numeric vector of distances for each gene to its centroid
- `gene_to_fam`: Gene-to-family mapping integer vector mapping with the same length as the distances vector (0 = no family, >0 = family index)
- **dscale**: Numeric vector of scaling factors per family

Return list:

- `rdi`: Numeric vector containing the RDI value for each gene
- `sorted_rdi`: Numeric vector containing the RDI values sorted in ascending order

Listing 55: R Compute Relative Distance Index (RDI)

```
tox_compute_rdi(distances, gene_to_fam, dscale)
```

(d) **Identify Outliers**

The function identifies outliers based on the top percentile of RDI values. Call the `tox_identify_outliers` function with the following arguments:

- `rdi`: Numeric vector containing the RDI value for each gene
- **(Optional)** `percentile`: Percentile threshold number for outlier detection (default is 95.0)

Return list:

- `is_outlier`: Logical vector indicating whether each gene is an outlier
- `threshold`: Actual threshold value used for outlier detection

Listing 56: R Identify Outliers

```
tox_identify_outliers(rdi, percentile)
```

6. **Compute Normalized Tissue Versatility:**

The function calculates a normalized tissue versatility score for selected gene expression vectors. It is based on the angle between each gene expression vector and the space diagonal. Versatility is normalized to [0, 1], where 0 means uniform expression 1 means expression in only one axis. Call the `tox_compute_tissue_versatility` function with the following parameters:

- `expression_vectors`: Numeric matrix array with size `n_axes` × `n_vectors`
- `vector_selection`: Logical vector with length of `n_vectors` to specify which vectors should be included in the calculation
- `axes_selection`: Logical vector with length of `n_axes` to specify which axes should be included in the calculation

Return list:

- `tissue_versatilities`: Numeric vector containing the calculated tissue versatilities for selected vectors
- `tissue_angles_deg`: Numeric vector containing the calculated angles in degrees for selected vectors
- `n_selected_vectors`: Integer number of vectors processed
- `n_selected_axes`: Integer number of axes used in calculation

Listing 57: R Tissue Versatility Calculation

```
tox_calculate_tissue_versatility(expression_vectors, vector_selection,
    axis_selection)
```

7. **R BST Interface**

(a) **Build BST Index**

R interface for building BST index. Uses R's native 1-based indexing.

Input:

- `x numeric`: Vector of real values

Output:

- `integer`: Permutation indices (1-based R indices)

```
ix <- build_bst_index(x)
```

(b) **BST Range Query**

Performs range queries on BST index. Returns a list with components.

Input:

- `x numeric`: Original values vector
- `ix integer`: BST index vector from build_bst_index (1-based)
- `lo numeric`: Lower bound of range (inclusive)
- `hi numeric`: Upper bound of range (inclusive)

Output:

- `list`: Contains indices and count components

```
result <- bst_range_query(x, ix, lo, hi)
indices <- result$indices  # 1-based indices
count <- result$count
```

(c) **Get Sorted Value**

Retrieves value from sorted position using R 1-based indexing.

Input:

- `x numeric`: Original values vector
- `ix integer`: BST index vector (1-based)
- `position integer`: Position in sorted order (1-based)

Output:

- `numeric`: Value at specified position

```
value <- get_sorted_value(x, ix, position)
```

8. **R k-d Tree Interface**

(a) **Build k-d Tree Index**

R interface for k-d tree construction. Input matrix dimensions are rows=dimensions, columns=points.

Input:

- `X matrix`: Matrix of points (nrow = dimensions, ncol = n_points)
- `dim_order integer`: Dimension ordering (1-based, optional)

Output:

- `integer`: Permutation indices (1-based R indices)

```
kd_ix <- build_kd_index(X, dim_order)
```

(b) **Build Spherical k-d Tree**

R interface for spherical k-d tree construction with separate implementation.

Input:

- `V matrix`: Matrix of vectors (nrow = dimensions, ncol = n_vectors)
- `dim_order integer`: Dimension ordering (1-based, optional)

Output:

- `integer`: Permutation indices (1-based R indices)

```
sphere_ix <- build_spherical_kd(V, dim_order)
```

(c) **Get Point from k-d Index**

Retrieves point from k-d tree using R matrix indexing.

Input:

- `X matrix`: Original points matrix
- `kd_ix integer`: k-d tree index vector (1-based)
- `position integer`: Position in k-d tree order (1-based)

Output:

- `numeric`: Point values as vector

```
point <- get_kd_point(X, kd_ix, position)
```

9. **R Interface for Array Serialization and Deserialization**

   The R interface provides bindings to the Fortran serialization and deserialization routines, enabling efficient handling of multi-dimensional arrays for statistical computing and bioinformatics applications. This interface supports R arrays, matrices, and vectors with automatic type conversion to compatible Fortran data types.

   (a) General Architecture

   The R package provides six main functions for array handling:

   - `tox_serialize_int_array(array, filename)` - Serialize N-dimensional integer arrays
   - `tox_deserialize_int_array(filename)` - Deserialize N-dimensional integer arrays
   - `tox_serialize_real_array(array, filename)` - Serialize N-dimensional real arrays
   - `tox_deserialize_real_array(filename)` - Deserialize N-dimensional real arrays
   - `tox_serialize_char_array(array, filename)` - Serialize N-dimensional character arrays
   - `tox_deserialize_char_array(filename)` - Deserialize N-dimensional character arrays

   (b) **Array Functions**

   Serialization Functions:

   - `array` (input): R array, matrix, or vector to be serialized
     Type: `integer` or `numeric` (double precision), or characters
     Dimensions: Theoretically are N dimensions supported, but typical use cases should not exceed 5 dimensions

   - `filename` (input): Name of the binary file to create
     Type: `character`

   Listing 58: R Array Serialization Example

```
tox_serialize_int_array(array, filename)
tox_serialize_real_array(array, filename)
tox_serialize_char_array(array, filename)
```

   Deserialization Functions:

   - `filename` (input): Name of the binary file to read
     Type: `character`
   - **Return Value:** Deserialized R array with original dimensions
     Type: `array`, `matrix`, or `vector`
     Data Type: Same as original

   Listing 59: R Array Deserialization Example

```
restored_int <- tox_deserialize_int_array(filename)
restored_real <- tox_deserialize_real_array(filename)
restored_char <- tox_deserialize_char_array(filename)
```

   (c) **Implementation Details**

- The R interface automatically converts between R data types and Fortran-compatible formats
- Integer vectors are converted to 32-bit integers
- Numeric vectors are converted to 64-bit floating point
- Character arrays handle variable-length strings with automatic padding
- All array dimensions are preserved during serialization/deserialization
- The interface handles both regular arrays and matrices seamlessly

Usage Example

Listing 60: Full example of serialization and deserialization in R

```
tox_serialize_int_array(arr1, fn("int1d.bin"))
restored <- tox_deserialize_int_array(fn("int1d.bin"))

# Real array example
arr2 <- matrix(runif(12), nrow = 3, ncol = 4)
tox_serialize_real_array(arr2, fn("real2d.bin"))
restored <- tox_deserialize_real_array(fn("real2d.bin"))

# Character array example
arr3 <- c("GENE1", "BRCA1", "TP53", "EGFR")
tox_serialize_char_array(arr3, fn("char1d.bin"))
restored <- tox_deserialize_char_array(fn("char1d.bin"))
```

Performance Characteristics

- Efficient memory mapping between R arrays and Fortran memory structures
- Minimal data copying between R and Fortran layers
- Optimized binary format for large scientific datasets
- Support for arrays exceeding available memory through file-based storage

(d) **Compatibility Notes**

- Requires R with appropriate Fortran bindings
- Supports standard R data types (integer, numeric, character)
- Preserves array attributes and dimensions during serialization
- Character arrays maintain content but may have padding removed upon deserialization