PanthRBase Documentation

PanthR Team

December 16, 2014

Contents

1 LinAlg

LinAlg

Linear Algebra module offers a framework for Linear Algebra computations with a goal to making those operations reasonably efficient for large sizes. If you will only be using small matrices and/or vectors, but require a huge number of them, you might find this library unsuitable.

LinAlg.Matrix

Implementation of 2-dimensional matrices.

LinAlg.Vector

Implementation of fixed-length vectors.

2 Matrix

StructuredM.LowerTriM

Subclass of `Matrix` representing diagonal matrices. Users should not need to access this subclass directly.

Matrix(arr, options)

The `Matrix` class is a representation of 2-dimensional algebraic matrices with real entries. Their values are internally represented as `Vectors`. One can access the matrix dimensions via the properties `nrow` and `ncol`.

New `Matrix` objects are created via the `Matrix` constructor, which accepts a number of options for its first argument, `arr`:

Examples:

```
\texttt{// All these create:
//        0 1 1
//        2 0 1
//
new Matrix([0, 2, 1, 0, 1, 1], { nrow : 2 }); // by column default
new Matrix([0, 1, 1, 2, 0, 1], { nrow : 2, byRow: true });
new Matrix([[0, 1, 1], [2, 0, 1]], { byRow : true });
new Matrix([[0, 2], [1, 0], [1, 1]]);
// Sparse matrix:
new Matrix({ 1: { 2: 1, 3: 1}, 2: { 1: 2, 3: 1 }}, { nrow : 2, ncol: 3 });

// The following produces in rows: [[1, 2, 3], [2, 4, 6]]
new Matrix(function(i, j) { return i * j; }, { nrow: 2, ncol: 3 });}
```

Matrix.CDiagM

Subclass of `Matrix` representing matrices that are constant multiples of the identity. The constructor expects two arguments: `val` with the value to be used, and `nrow`, which is either a number indicating the number of rows or an object with an `nrow` property.

CDiagM matrices are immutable.

Matrix.DenseM

Subclass of `Matrix` representing "dense" matrices. Dense matrices are internally stored simply as Javascript Arrays. Users should not need to access this subclass directly.

Matrix.DiagM

Subclass of `Matrix` representing diagonal matrices. Users should not need to access this subclass directly. Use `Matrix.diag` instead.

One can only set values on the diagonal of a DiagM matrix. Trying to set outside the diagonal will result in error. In order to set values outside the diagonal, would need to "unstructure" the matrix.

Using rowView/colView on diagonal matrices may be quite inefficient, as it does not recognize the sparse nature of those vectors.

Matrix.LowerTriM

Subclass of `StructuredM` representing "Lower triangular" matrices.

The constructor expects two arguments:

- The first argument,`values`, cam be:

Matrix.OuterM

Subclass of `Matrix` representing outer products of vectors (i.e., rank-1 matrices). Users should not need to access this subclass directly.

Matrix.PermM

Subclass of `Matrix` representing permutation matrices. The constructor expects two arguments, a `perm` object that determines a `Permutation`, and an `nrow` number/object specifying the matrix dimensions.

Multiplying a non-permutation matrix m by a permutation matrix p returns an appropriate view (`Matrix.ViewM`) into m. Multiplying two permutation matrices returns the matrix for the composed permutation (`Matrix.PermM`).

Matrix.ProdM

Subclass of `Matrix` representing products of matrices. Users should not need to access this subclass directly.

Matrix.Solver

Class containing solvers for various linear systems. TODO: Add Solver module docs

Matrix.SparseM

Subclass of `Matrix` representing "sparse" matrices. Sparse matrices are stored as objects, whose keys represent the indices that have non-zero values. Users should not need to access this subclass directly.

Matrix.StructuredM

Subclass of `Matrix` acting as a superclass for classes of matrices with extra structure. Users should not need to access this subclass directly.

Matrix.SumM

Subclass of `Matrix` representing sums (A + k * B) of matrices. Users should not need to access this subclass directly.

Matrix.SymmetricM

Subclass of `StructuredM` representing symmetric matrices.

A symmetric matrix behaves exactly like a `Matrix.LowerTriM` matrix reflected across the main diagonal.

Matrix.TabularM

Subclass of `Matrix` representing matrices whose values are specified via a function `f(i)` of the index. The values of the matrix are computed lazily, only when they are accessed. Users should not need to access this subclass directly.

Matrix.UpperTriM

Subclass of `StructuredM` representing "Upper triangular" matrices.

See `Matrix.LowerTriM` for the constructor parameters. See `Matrix.prototype.upper` for obtaining the upper triangle of a given square matrix.

Matrix.Vector.colBind

See `Matrix.colBind`

Matrix.Vector.rowBind

See `Matrix.rowBind`

Matrix.ViewM

Subclass of `Matrix` representing submatrix views into another matrix. Changes to the view are reflected on the original matrix and vice-versa. Use `Matrix.prototype.view` to create these.

See also: `Matrix.prototype.rowView`, `Matrix.prototype.colView`, `Matrix.prototype.diagView`.

Matrix.ViewMV

Subclass of Vector that is used internally by `Matrix` for representing the rows/columns/diagonals of a matrix as vectors.

For creating these, see: `Matrix.prototype.rowView`, `Matrix.prototype.colView`, `Matrix.prototype.diagView`.

Matrix.Vector.prototype.colBind

See `Matrix.colBind`

Matrix.Vector.prototype.mult(other)

TODO: Find a way to add to Vector docs

Matrix.Vector.prototype.rowBind

See `Matrix.rowBind`

Matrix.colBind(matrices)

Bind the arguments column-wise into a matrix. The arguments may be a mixture of matrices and vectors, but their `nrow/length` must all be the same.

Matrix.commonConstr(A, B)

Return a common constructor for `A` and `B`, from the lists provided by `Matrix.prototype.classes`.

Matrix.compatibleDims(A, B)

Return whether `A` and `B` have compatible dimensions for forming the product `A * B`. If `A` and; `B` are not both matrices, then one of them is a matrix and the other is a vector.

Matrix.const(val, nrow)

Return a constant multiple of the identity matrix. These matrices cannot become mutable. They should be treated as constants. The second argument, `nrow` can be the number of rows, or an object with an `nrow` argument. For instance to create an identity matrix with size same as the matrix `A` one would do:

```
\texttt{Matrix.const(1, A);   // Identity matrix with dimension same as A.}
```

Matrix.diag(diagonal, len)

Return a square diagonal matrix with values given by `diagonal`. The argument `diagonal` may be an array, a `Vector`, or a function `f(i)`. In the latter case, a second argument `len` is required to provide the length of the resulting diagonal. `len` may also be an object with an `nrow` property.

This method takes ownership of the `diagonal` vector and may change its values when it itself is changed. Clone the array/vector before passing it to avoid this.

To obtain a diagonal of an arbitrary matrix, see `Matrix.prototype.diagView`.

Matrix.ensureSameDims(A, B)

Throw error if `A`, `B` don't have same dimensions.

Matrix.perm(perm, nrow)

Return a permutation matrix based on the permutation indicated by `perm`. `perm` can be a `Permutation` object, or anything that can be turned to one (see `Permutation`).

Matrix.rowBind(matrices)

Bind the arguments row-wise into a matrix. The arguments may be a mixture of matrices and vectors, but their `ncol/length` must all be the same.

Matrix.sameDims(A, B)

Return whether the matrix `A` has the same dimensions as the matrix `B`.

Matrix.prototype._get(i, j)

Internally used by `Matrix.prototype.get`. May be used in place of `Matrix.prototype.get` if both arguments are always present.

Matrix.prototype._set(i, j, val)

Internally used by `Matrix.prototype.set`. *Internal method.* May be used instead of `Matrix.prototype._set` if all three arguments are always present.

Matrix.prototype.all(pred)

Return true, if the predicate `pred(val, i, j)` is true for all entries, false otherwise.

Matrix.prototype.any(pred)

Return true, if the predicate `pred(val, i, j)` is true for at least one entry, false otherwise.

Matrix.prototype.change(i, j, val)

Internal method used by `Matrix.prototype._set` to change the value of the matrix at a particular location. *Internal method.* This method bypasses various checks and should only be used with extreme care.

Matrix.prototype.classes

The array of constructors for this type and its supertypes, in order from most specific to most general.

Matrix.prototype.clone(faithful)

Create a clone of the matrix. The clone inherits the values that the matrix has at the time of cloning. If `faithful` is `true` (default), then the clone also inherits any structure (e.g. being diagonal) when possible.

Unfaithful clones are useful if you want to set values of a structured matrix outside of the structure (e.g. setting off-diagonal elements on a diagonal matrix). In general, `Matrix.prototype.set` respects any imposed structure the matrix has on its creation.

Matrix.prototype.colBind(matrices)

See `Matrix.colBind`

Matrix.prototype.colPermute(perm)

Permute the columns of the matrix.

Matrix.prototype.colView(j)

Return a `Vector` view of the `j`-th column of the matrix.

Matrix.prototype.cols()

Return an array of all matrix columns as colViews

Matrix.prototype.compute(i, j)

Computes the value at the (i, j) location. *Internal method*. Use `Matrix.prototype.get` instead.

Matrix.prototype.constr()

Return the constructor method to be used for creating new objects of this type.

Each of these constructors will accept the parameter list `(f, obj)` where `f(i, j)` is a function for generating matrix values, and `obj` has properties `nrow` and `ncol`.

Matrix.prototype.diagView(offset)

Return a `Vector` view of the diagonal of the matrix specified by the given `offset` (defaults to 0). The main diagonal has offset 0, the diagonal above it has offset 1, while the one below the main diagonal has offset -1. Asking for a diagonal beyond the matrix bounds results in an error.

```
\texttt{var A1 = new Matrix([2, 3, 4, 5, 6, 7], { nrow: 2 });
A1.diagView();      // [2, 5];
A1.diagView(-1);    // [3];
A1.diagView(1);     // [4, 7];
A1.diagView(2);     // [6];
A1.diagView(3);     // Error;}
```

Matrix.prototype.each(f)

Apply the given function to each entry in the matrix. The signature of the function is `f(val, i, j)`.

`Each` respects the "structure" of the matrix. For instance on a `SparseM` matrix, it will only be called on the non-zero entries, on a `DiagM` matrix it will only be called on the diagonal entries, on a `SymmetricM` matrix it will be called on only roughly one half of the entries and so on.

If you really need the function to be called on *each* matrix entry, regardless of structure, then you should use `Matrix.prototype.clone` first to create an "unfaithful clone".

Matrix.prototype.eachCol(f)

Apply the function `f` to each column in the matrix. The signature of `f` is `f(col, j)` where `col` is a `Vector` object representing the `j`-th col.

Matrix.prototype.eachPair(other, f)

Apply function `f(val1, val2, i, j)` to all pairwise entries of `this` and `other`. The matrices must have the same dimensions. No promises are made about the order of iteration.

Matrix.prototype.eachRow(f)

Apply the function `f` to each row in the matrix. The signature of `f` is `f(row, i)` where `row` is a `Vector` object representing the `i`-th row.

Matrix.prototype.equals(m2, tolerance)

Test if `this` pointwise equals `m2`, within a given pointwise `tolerance` (defaults to `Vector.tolerance`).

Matrix.prototype.forEach(f)

Alias for `Matrix.prototype.each`

Matrix.prototype.force()

Force unresolved computations for the matrix.

Matrix.prototype.get(i, j)

Return the value at location `(i, j)`. Returns `0` if accessing a location out of bounds.

Called with 0 or 1 arguments, it is an alias for `Matrix.prototype.toArray`.

Matrix.prototype.getSolver()

Internally used to obtain a solver for systems.

Matrix.prototype.inverse()

Return the inverse of `this`, if `this` is a square non-singular matrix.

Matrix.prototype.isA(constr)

Return whether `constr` is in the list of class constructors produced by `Matrix.prototype.classes`.

Matrix.prototype.lower()

Return a lower-triangular matrix created by the lower triangle of `this`.

Matrix.prototype.lvMult(vec)

Multiply the matrix on the left with a vector `vec`. `vec.length` must equal `this.nrow`. Returns a vector of length `this.ncol`. This is an *internal method* and bypasses certain tests.

Matrix.prototype.map(f)

Apply the function `f(val, i, j)` to every entry of the matrix, and assemble the returned values into a new matrix. Just like `Matrix.prototype.each`, this method respects the structure of the input matrix, and will return a matrix with the same structure, only applying `f` on the values pertinent to the structure.

If you really need the function to be called on *each* matrix entry, regardless of structure, then you should use `Matrix.prototype.clone` first to create an "unfaithful clone".

```
\texttt{// Create a matrix containing the absolute values of the values in A.
A.map(Math.abs);}
```

Matrix.prototype.mapCol(f)

Similar to `Matrix.prototype.mapRow`, but operating on the columns of the matrix instead.

Matrix.prototype.mapPair(other, f)

Create a new matrix by applying the function `f(val1, val2, i, j)` to all pairwise entries of `this` and `other`. No matrix structure is preserved. The matrices must have the same dimensions.

Matrix.prototype.mapRow(f)

Apply the function `f(row, i)` to each row in the matrix, and assemble the resulting values.

If the return values of `f` are numbers, they are assembled into a `Vector`. If they are arrays or `Vectors`, then they must be of the same length, and they are assembled into a matrix with `nrow` equal to the original matrix's `nrow`, and `ncol` equal to the value's length.

```
\texttt{// Create an n x 3 array of the index, 1-norm and 2-norm of each row.
A.mapRow(function(row, i) { return [i, row.norm(1), row.norm(2) ]; });}
```

Matrix.prototype.mult(other)

Return the matrix product `this * other`, where `this` and `other` have compatible dimensions.

Matrix.prototype.mutable(newSetting)

With no arguments, returns the mutable state of the matrix.

With a boolean argument, sets the mutable state of the matrix and returns the matrix.

Matrix.prototype.pAdd(other, k)

Return `this + k * other`, where `this` and `other` are matrices of the same dimensions, and `k` is a scalar.

Matrix.prototype.reduce(f, initial)

Return the accumulated value of the calls of `f(acc, val, i, j)` over the entries of the matrix, with `acc` starting with value `initial`.

`Matrix.prototype.reduce` is similar to `Matrix.prototype.each` in how it deals with structured matrices.

Compare with `Vector.prototype.reduce`.

```
\texttt{var A = new Matrix(Math.random, { nrow: 3, ncol: 2 });
// Counts the number of entries in A which exceed 0.5
A.reduce(function(acc, val, i, j) {
  return acc + (val > 0.5 ? 1 : 0);
}, 0);}
```

Matrix.prototype.reduceCol(f, initial)

Return the accumulated value of the calls of `f(acc, col, i, j)` over the columns of the matrix, with `acc` starting with value `initial`.

Matrix.prototype.reducePair(other, f, initial)

Reduce on the pair of matrices `this` and `other` using the function `f(acc, val1, val2, i, j)`, with an `initial` value. The matrices must have the same dimensions. No promises are made about the order of iteration.

Matrix.prototype.reduceRow(f, initial)

Return the accumulated value of the calls of `f(acc, row, i, j)` over the rows of the matrix, with `acc` starting with value `initial`.

```
\texttt{// Add the rows in A with 2-norm >= 1
A.reduce(function(acc, row, i, j) {
  if (row.norm() >= 1) { return acc.pAdd(row); }
  return acc;
}, Vector.const(0, A.ncol));}
```

Matrix.prototype.rowBind(matrices)

See `Matrix.rowBind`

Matrix.prototype.rowPermute(perm)

Permute the rows of the matrix.

Matrix.prototype.rowView(i)

Return a `Vector` view of the `i`-th row of the matrix.

Matrix.prototype.rows()

Return an array of all matrix rows as rowViews

Matrix.prototype.rvMult(vec)

Multiply on the right with a vector `vec`. `vec.length` must equal `this.ncol`. Returns a vector of length `this.nrow`. This is an *internal method* and bypasses certain tests.

Matrix.prototype.sMult(k)

Return `k * this`, where `k` is a scalar (required numerical argument).

Matrix.prototype.set(i, j, val)

Set the value of the matrix at the `(i, j)` location to `val`. Requires that the matrix be set to be mutable.

If called with only one argument, then that argument may be a function `f(i, j)`, or a single value, or a `Matrix` with the same dimensions. That argument will then be used to set all the values of the Matrix.

```
\texttt{var A1 = new Matrix([1, 2, 3, 4, 5, 6], { nrow: 2, byRow: true });
A1.set(1, 1, 42);     // Throws an exception
A1.mutable(true);     // Set matrix to mutable
A1.set(2, 2, 42);     // Changes 5 to 42
A1.set(Math.random); // Fills A1 with random values
A1.set(5);            // Sets all entries to 5
var A2 = new Matrix([1, 2, 3, 4, 5, 6], { nrow: 2, byRow: true });
A1.set(A2);           // Sets all values of A1 based on those from A2
A1.set(1, 1, 42);     // Only changes A1, not A2}
```

Trying to set at an out-of-bounds location results in an exception. If the matrix is "structured", trying to set at a location outside the structure (e.g. an off-diagonal entry of a diagonal matrix) also results in an exception.

: In order to avoid unnecessary computations, many matrix operations avoid computing their values until those values are called for. If you have used a matrix or vector in the construction of other matrices/vectors, then you should avoid changing that matrice's values, as the effects of those changes on the dependent objects are unpredictable. In general, you should treat a matrix that has been used in the creation of other matrices as an immutable object, unless `Matrix.prototype.force` has been called on those other matrices.

Matrix.prototype.solve(b)

Return the solution to `Ax = b`, where `A` is `this` and `b` is a `Matrix` or `Vector`. Only works for square non-singular matrices `A` at the moment.

Matrix.prototype.toArray(byRow)

Return an array of arrays representing the matrix. This representation is as an array of columns (or an array of rows if `byRow` is `true`).

```
\texttt{var A = new Matrix([1, 2, 3, 4, 5, 6], { byRow: true, nrow: 3 });
A.toArray(true);  // [[1, 2], [3, 4], [5, 6]]
A.toArray(false); // [[1, 3, 5], [2, 4, 6]]}
```

Matrix.prototype.toVector(byRow)

Return a flat vector of the matrix values by concatenating its columns (or its rows if `byRow` is true). This is not a view into the matrix, and cannot be used to change the matrix values.

Matrix.prototype.transpose()

Return the transpose of the matrix, preserving any appropriate structure.

Matrix.prototype.upper()

Return a upper-triangular matrix created by the upper triangle of `this`.

Matrix.prototype.validIndices(i, j)

Return whether the (i, j) pair is within the matrix's bounds. Matrices with extra extra structure do further checks via `Matrix.prototype.validate`.

Matrix.prototype.validate(i, j, val)

Overriden by subclasses that need special index/value validation.

This method will be called from `Matrix.prototype._get` with two arguments `(i, j)`. It should return whether the pair `(i, j)` is valid for that array's structure, without worrying about being out of bounds (which is checked separately).

This method is also called from `Matrix.prototype._set` with three arguments `(i, j, val)`, where `val` is the value that is to be set in those coordinates. It should either return `false` or throw an error if the assignment should not happen, and return true if it should be allowed to happen.

Matrix.prototype.view(rowIndex, colIndex, dims)

Return a view into a submatrix of `this`.

The parameters `rowIndex`, `colIndex` may be either arrays or functions `f(i)` used to obtain the indices. In the latter case, a third argument `dims` is required.

`dims` is an object with properties `nrow` or `ncol` as needed, specifying the dimensions of the resulting matrix.

```
\texttt{// A 2x3 matrix
var A1 = new Matrix([2, 3, 4, 5, 6, 7], { nrow: 2 });
// Both return a view into the 2nd \& 3rd columns as a 2x2 matrix
var A2 = A1.view([1, 2], [2, 3]);
var A2 = A1.view([1, 2], function(j) { return 1 + j; }, { ncol: 2 });}
```

The View matrix (vector) is linked to the original matrix. The mutable state of the view is that of the original matrix. Changing the values in the view also changes the values in the matrix, and vice versa. Use `Matrix.prototype.clone` on the view matrix to break the link.

3 Vector

Vector.ConstV

Subclass of `Vector` efficiently representing vectors all of whose values are meant to be the same number. Users should not need to access this subclass directly. Use `Vector.const` or `Vector.ones` instead.

Vector.DenseV

Subclass of `Vector` representing "dense" vectors. Dense vectors are internally stored simply as Javascript Arrays. Users should not need to access this subclass directly.

Vector.SparseV

Subclass of `Vector` representing "sparse" vectors. Sparse vectors are stored as objects, whose keys represent the indices that have non-zero values. Users should not need to access this subclass directly.

Vector.TabularV

Subclass of `Vector` representing vectors whose values are specified via a function `f(i)` of the index. The values of the vector are computed lazily, only when they are accessed. Users should not need to access this subclass directly.

Vector.ViewV

Subclass of `Vector` representing vectors that provide a "view" into another object, e.g. a row or column of a `Matrix`. Changes to a view vector cause changes to the corresponding "viewed" object and vice versa. Users should not need to access this subclass directly. Use `Vector.prototype.view` instead.

Matrix.Vector.prototype.outer(v2, f)

Return the outer product matrix of two vectors. If a function `f(val1, val2, i, j)` is provided as the second argument, it will be used. If no second argument is provided, the usual multiplication of numbers is used resulting in the standard outer product.

TODO: Include helpful examples.

TODO: Find a way to add this the Vector docs

Vector(arr, len)

`Vector` objects are Javascript representations of real-valued vectors. They are constructed in one of three ways depending on the type of the first parameter `arr`:

When `arr` is a `Vector`, it is simply returned unchanged.

`Vector` objects are 1-indexed. By default, they are immutable structures, they cannot be edited once created. See `Vector.MutableV` for a description of mutable vectors.

Every vector has a fixed `length`, accessed as a property. Vectors of length `0` are allowed, though there is not much one can do with them.

```
\texttt{// A length-4 vector
var v1 = new Vector([3, 5, 1, 2]);
// A length-10 sparse vector
var v2 = new Vector({ 4: 10, 2: 12 }, 10);
// A length-3 vector with values exp(1), exp(2), exp(3)
var v3 = new Vector(Math.exp, 3);
v3.length === 3  // true
// A length-5 vector with all values equal to 4
var v4 = new Vector(4, 5);}
```

Vector.concat(vectors)

Returns the concatenation of its arguments. The arguments may be vectors, arrays or plain numbers.

Vector.const(val, len)

Generate a constant vector of length `len`, with all entries having value `val`. *Constant vectors are immutable.* Use `Vector.fill` if you want to initialize a vector with some value(s).

Vector.ones(len)

Generate a constant vector of length `len`, with all entries having value 1. *Constant vectors are immutable.*

```
\texttt{// Sums all elements of v1
Vector.ones(v1.length).dot(v1)}
```

Vector.seq(a, b, step)

Create a vector that follows a linear progression starting from `a` increasing by `step` amount, and ending the moment `b` is exceeded.

If `step` is omitted, it defaults to 1 or -1 depending on the relation between `a` and `b`. If `b` is also omitted, then the vector generated is `1,2,...,a`.

```
\texttt{Vector.seq(1, 6, 2)  // Produces [1, 3, 5]
Vector.seq(5, 1)     // Produces [5, 4, 3, 2, 1]
Vector.seq(3)        // Produces [1, 2, 3]}
```

Vector.tolerance

The tolerance used in equality tests. You may set a different value. Defaults to `1e-8`.

SparseV.prototype.resize(length, fill)

Return a new resized version of `this` with a new `length`.`fill` may be:

- `true`: we then recycle the Vector's values to the new length.
- `false` or omitted: we then fill in with zeros.
- a function `f(i)`: It is then used to fill in the *new* values.

Vector.prototype._get(i)

Same as `Vector.prototype.get`, but only works with an integer argument.

Vector.prototype._set(i, val)

Set the entry at index `i` of the vector to `val`. Can only be used on a vector that is currently mutable.

Vector.prototype.all(pred)

Return true, if the predicate `pred(val, i)` is true for all entries, false otherwise.

Vector.prototype.any(pred)

Return true, if the predicate `pred(val, i)` is true for at least one entry, false otherwise.

Vector.prototype.change(i, val)

Method meant to be used internally for setting the value at index `i` of the vector to `val`. Bypasses the checks made by `Vector.prototype._set`, including whether the vector has been set to be mutable. *Avoid using this method unless you are really certain of what you are doing!*

Vector.prototype.clone()

Return a clone of the vector.

Vector.prototype.compute(i)

Compute the entry at index `i` of the vector. This method is used internally by `Vector.prototype.get` and `Vector.prototype._get` to obtain the correct value in cases where the vector values are stored *lazily*. Users should not call it directly. Use `Vector.prototype.get` or `Vector.prototype._get` instead.

Vector.prototype.concat(vectors)

See `Vector.concat`.

Vector.prototype.cumMax()

Create a new vector from the partial maxima in the vector.

```
\texttt{v1.cumMax(); // Produces [3, 5, 5, 5]}
```

Vector.prototype.cumMin()

Create a new vector from the partial minima in the vector.

```
\texttt{v1.cumMin(); // Produces [3, 3, 1, 1]}
```

Vector.prototype.cumProd()

Create a new vector from the partial products in the vector.

```
\texttt{v1.cumProd(); // [3, 15, 15, 30]}
```

Vector.prototype.cumSum()

Create a new vector from the partial sums in the vector.

```
\texttt{v1.cumSum();  // [3, 8, 9, 11]}
```

Vector.prototype.cumulative(f, initial)

Create a new vector by accumulating one by one the results `f(acc, val, i)` as `val` ranges over the values of the vector, starting with the value `initial` (defaults to `0`). This is effectively a version of `Vector.prototype.reduce` where each intermediate step is stored.

```
\texttt{var v1 = new Vector([3, 5, 1, 2]);
function f(acc, val) { return acc + val * val; }
v1.cumulative(f, 2);  // [11, 36, 37, 41]}
```

Vector.prototype.diff()

Compute the successive differences of the values in the vector, "`this[i+1] - this[i]`."

```
\texttt{v1.diff(); // Produces: [5 - 3, 1 - 5, 2 - 1]
v1.diff().length === v1.length - 1 // true}
```

Vector.prototype.dot(v)

Compute the dot product of `this` with `v`.

```
\texttt{// Returns 3 * 3 + 5 * 5 + 1 * 1 + 2 * 2
v1.dot(v1);}
```

Vector.prototype.each(f, skipZeros)

Execute the function `f` for each entry of the vector, starting with the entry with index 1. `f` will be called as `f(value, index)`. If `skipZeros` is `true`, then the system *may* skip the execution of `f` for zero entries.

```
\texttt{var v1 = new Vector([3, 5, 1, 2]);
// Prints: 3 1, 5 2, 1 3, 2 4
v1.each(console.log);}
```

Vector.prototype.eachPair(v2, f, skipZeros)

Execute the function `f` for each pair of corresponding entries from the vector and `v2`, starting with the entries with index 1. `f` will be called as `f(val1, val2, index)`, where `val1`, `val2` are the entries of the vectors `this`, `v2` at index `i`. If `skipZeros` is `true`, then the system *may* skip the execution of `f` when one of the values is `0`.

```
\texttt{// Prints 3 3 1, 5 5 2, 1 1 3, 2 2 4
v1.eachPair(v1, console.log);}
```

Vector.prototype.equals(v2, tolerance)

Test if `this` pointwise equals `v2`, within a given pointwise `tolerance` (defaults to `Vector.tolerance`).

Vector.prototype.fill(val, start, end)

Fill in the segment of the vector's values from `start` to `end` with `val`. If `start` is an array or vector, use its values as the indices to fill. Only usable on vectors that are currently mutable.

Vector.prototype.foldl

Alias for `Vector.prototype.reduce`.

Vector.prototype.forEach(f, skipZeros)

Alias for `Vector.prototype.each`.

Vector.prototype.force()

Force a vector to be evaluated. This resolves any deferred calculations needed for the computation of the vector's elements.

Many vector methods, notably `Vector.prototype.map`, delay the required computations until the point where they need to be computed. `Vector.prototype.force` is one way to force that computation.

Vector.prototype.get(i)

Generic accessor method to obtain the values in the vector. The argument `i` can take a number of different forms:

Users should always go through this method, or `Vector.prototype._get`, when accessing values of the vector unless they really know what they're doing. You may use `Vector.prototype._get` for slightly more efficient access if you will always be accessing values via an integer.

```
\texttt{v1.get() === [3, 5, 1, 2];
v1.get([2, 3]) === [5, 1];
v1.get(1) === 3;
v1.get(2) === 5;
// Out of range defaults to 0
v1.get(0) === 0;
v1.get(5) === 0;}
```

Vector.prototype.isSparse()

Return whether the vector is stored as a sparse vector.

Vector.prototype.map(f, skipZeros)

Create a new vector by applying the function `f` to all elements of `this`. The function `f` has the signature `f(val, i)`. If `skipZeros` is `true`, the operation may assume that `f(0, i)=0` and may choose to skip those computations.

`Vector#map` only returns a "promise" to compute the resulting vector. The implementation may choose to not call `f` until its values are actually needed. Users should not rely on side-effects of `f`.

```
\texttt{// Results in [3 + 1, 5 + 2, 1 + 3, 2 + 4];
v1.map(function(val, i) { return val + i; });}
```

Vector.prototype.mapPair(v2, f, skipZeros)

Like `Vector.prototype.map`, but the function `f` acts on two vectors, with signature `f(val1, val2, i)`. If `skipZeros` is `true`, the implementation may assume that `f` will return `0` as long as one of the values is `0`.

Vector.prototype.mutable(isMutable)

Called with no arguments (or with undefined/null argument), return the mutable state of the vector.

Called with a boolean argument `isMutable`, set the mutable state to that value and return the vector.

Vector.prototype.norm(p)

Compute the p-norm of the vector. `p` should be a positive real number or `Infinity`. Defaults to the 2-norm.

```
\texttt{v.norm(1)           // 1−norm (sum of absolute values)
v.norm()            // 2−norm (Euclidean formula)
v.norm(Infinity) // Infinity (max) norm}
```

Vector.prototype.order(desc)

order takes a parameter `desc` which defaults to `false`. If `desc` is `true` then the order is given in descending order. Example: If `this` has values [3, 1, 8, 10, 2] then order(false) returns [2, 5, 1, 3, 4]. `desc` can also be a comparator function. The default ordering functions only work for numeric vectors. Provide custom function otherwise.

Vector.prototype.pAdd(v)

Pointwise add two vectors. Returns a new vector.

```
\texttt{// Returns: [3 + 1, 5 + 1, 1 + 1, 2 + 1]
v1.pAdd(Vector.ones(4));}
```

Vector.prototype.pDiv(v)

Pointwise divide two vectors. Returns a new vector.

Vector.prototype.pMult(v)

Pointwise multiply two vectors. Returns a new vector.

Vector.prototype.pPow(n)

Raise each entry in `this` to the `n`-th power. Returns a new vector.

Vector.prototype.pSub(v)

Pointwise subtract two vectors. Returns a new vector.

Vector.prototype.permute(perm)

Permute the vector entries according to `perm`

Vector.prototype.reduce(f, initial, skipZeros)

Similar to `Array.prototype.reduce`. Given a function `f(acc, val, i)` and an `initial` value, it successively calls the function on the vector's entries, storing each result in the variable `acc`, then feeding that value back. If `skipZeros` is `true`, this operation *may* skip any zero entries. `initial` and `acc` do not have to be numbers, but they do need to have the same type, and `f` should return that same type.

```
\texttt{function add(acc, val) { return acc + val; };
// Equivalent to ((((4 + 3) + 5) + 1) + 2)
v1.reduce(add, 4);}
```

Vector.prototype.reducePair(v2, f, initial, skipZeros)

Similar to `Vector.prototype.reduce` but acts on a pair of vectors `this`, `v2`. The signature of the function `f` would be `f(acc, val1, val2, i)` where `acc` is the accumulated value, `i` is the index, and `val1`, `val2` are the `i`-indexed values from `this`, `v2`. If `skipZeros` is `true`, the implementation *may* avoid calling `f` for an index `i` if one of the values is `0`.

The vectors `this`, `v2` need to have the same length.

```
\texttt{function f(acc, val1, val2) = { return acc + val1 * val2; };
// Computes the dot product of v1, v2.
v1.reducePair(v2, f, 0)}
```

Vector.prototype.rep(times)

Return a new vector with the values of `this` repeated according to `times`.

- If `times` is a number, recycle that many times.
- If `times` is a vector or array of the same length, use its values as frequencies for the corresponding entries.
- If `times` is an object with a `length` property, cycle the values until that length is filled.
- If `times` is an object with an `each` property, repeat each value that many times.

Vector.prototype.resize(length, fill)

Return a new resized version of `this` with a new `length`.`fill` may be:

- `true`: we then recycle the Vector's values to the new length.
- `false` or omitted: we then fill in with zeros.
- a function `f(i)`: It is then used to fill in the *new* values.

Vector.prototype.sMult(a)

Multiply the vector `v` by the constant `a`. Returns a new vector.

Vector.prototype.sameLength(other)

Return whether the vector has the same length as the vector `other`.

Vector.prototype.set(i, vals)

Set the entries of the vector that are specified by the parameter `i` to the value(s) specified by the parameter `vals`. *Can only be used on a vector that is set to be mutable.* The parameters can take two forms:

In order to set more than one of a vector's values at the same time, create a `Vector.ViewV` and use `Vector.prototype.set` on that.

You may use `Vector.prototype._set` if efficiency is an issue and you are certain that you are in the single-index case.

Vector.prototype.sort(desc)

Vector.prototype.toArray()

Return a Javascript array of the vector's values. Returns a new Array object every time.

Vector.prototype.view(arr, len)

Return a view vector on the `arr` indices. View vectors reflect the values on their target, but allow one to access those locations via a different indexing. Changing the values of a view vector actually changes the values of their target.

The indices to view may also be specified via a function `f(i)` as the first argument. In that case, a second argument is needed with the desired length for the resulting vector.

```
\texttt{var v1 = new Vector([3, 5, 1, 2]);
var view = v1.view([2, 3]);
view.get(1) === 5;
view.get(2) === 1;
var view2 = v1.view(function(i) { return 5 - i; }, 3); // [2, 1, 5]}
```

4 Permutation

CholeskyS(A)

Solves the system Ax = b for a symmetric positive definite A by computing a Cholesky decomposition. `A` is a square symmetric positive definite matrix. "isSingular" returning true would indicate the matrix is not positive definite.

DiagS(diag)

diag is meant to be the diagonal of a diagonal matrix D. Solves the system Dx = b.

LowerS(A)

Expects a `LowerTriM` matrix for `A` (or full square matrix and it will use its lower triangle). Solves by forward substitution.

PLUS(A, strategy)

Solves the system Ax = b by computing a PLU decomposition. `A` is a square matrix and `strategy` specifies the pivoting strategy for the PLU solver ('partial' or 'complete').

Permutation(relation)

A class representing permutations on sets `1, 2, ..., n`. Permutations are represented via a "function object" whose key-value pairs are non-fixed points of the permutation. In other words, if `s(i) = j` and `j` is not equal to `i`, then the object contains a key `i` with value `j`. If a key `i` does not appear in the object then `s(i) = i`. For example the cycle `(2 4)` can be represented as the object ` 2: 4, 4: 2 `.

The size `n` is only implicit; permutations can be thought of as functions on any set big enough to contain their non-fixed points. The first argument, `relation`, needs to be either an array representing a cycle, an array of arrays representing a product of cycles, or a "function object".

It can also be called with another `Permutation` as first argument, in which case it returns that permutation.

Permutation.cycleToObject(cycles)

Return the function object represented by the cycle. Helper method. It also recognizes an array of cycles, or a "function object".

Solver(A)

Top level class for solving linear systems

UpperS(A)

Expects a `UpperTriM` matrix for `A` (or full square matrix and it will use its Upper triangle). Solves by back substitution.

DiagS.prototype._solve(b)

Expects b to be a vector. Returns a vector

LowerS.prototype._solve(b)

Expects b to be a vector. Returns a vector

Permutation.prototype.compose(other)

Return the composed permutation of `this` followed by `other`. `other` may be a `Permutation`, a cycle, array of cycles, or a "function object".

Permutation.prototype.inverse()

Return the inverse permutation

Permutation.prototype.toCycles()

Return an array representing the cycle representation of the permutation.

Solver.prototype.isSingular()

Return whether the system that the solver solves is "singular". Overridden in subclasses. When `isSingular` returns true, you should not call `solve`.

Solver.prototype.solve(b)

Expects b to be a Vector or Matrix (maybe array also?)

UpperS.prototype._solve(b)

Expects b to be a vector. Returns a vector

5 utils

utils.op

Arithmetic operators

utils.op.add(a, b)

The function that adds two numbers. Also available as `utils.op['+']`.

utils.op.div(a, b)

The function that divides two numbers. Also available as `utils.op['/']`.

utils.op.mult(a, b)

The function that multiplies two numbers. Also available as `utils.op['*']`.

utils.op.sub(a, b)

The function that subtracts two numbers. Also available as `utils.op['-']`.

utils.veryClose(a, b, tol)

Return whether the numbers `a`, `b` are within `tol` of each other.