# Linear Algebra Cheat Sheet

Addison Hu edited this page on 20 Jun · 27 revisions

## Core Concepts

Compared to other numerical computing environments, Breeze matrices default to column major ordering, like Matlab, but indexing is 0-based, like Numpy. Breeze has as its core concepts matrices and column vectors. Row vectors are normally stored as matrices with a single row. This allows for greater type safety with the downside that conversion of row vectors to column vectors is performed using a transpose-slice ( `a.t(::,0)` ) instead of a simple transpose ( `a.t` ).

UFuncs are very important in Breeze. Once you get a feel for the syntax (i.e. what's in this section), it might be worthwhile to read the first half of the UFunc wiki page. (You can skip the last half that involves implementing your own UFuncs...until you're ready to contribute to Breeze!)

## Quick Reference

The following table assumes that Numpy is used with `from numpy import *` and Breeze with

```
import breeze.linalg._
import breeze.numerics._
```

### Creation

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Zeroed matrix | `DenseMatrix.zeros[Double](n,m)` | `zeros(n,m)` | `zeros((n,m))` | `mat.or.vec(n, m)` |
| Zeroed vector | `DenseVector.zeros[Double](n)` | `zeros(n,1)` | `zeros(n)` | `mat.or.vec(n, 1)` |
| Vector of ones | `DenseVector.ones[Double](n)` | `ones(n,1)` | `ones(n)` | `mat.or.vec(n, 1) + 1` |
| Vector of particular number | `DenseVector.fill(n){5.0}` | `ones(n,1) * 5` | `ones(n) * 5` | `(mat.or.vec(5, 1) + 1) * 5` |
| range given stepsize | `DenseVector.range(start,stop,step)` or `Vector.rangeD(start,stop,step)` | | | `seq(start,stop,step)` |
| n element range | `linspace(start,stop,numvals)` | `linspace(0,20,15)` | | |
| Identity matrix | `DenseMatrix.eye[Double](n)` | `eye(n)` | `eye(n)` | `identity(n)` |
| Diagonal matrix | `diag(DenseVector(1.0,2.0,3.0))` | `diag([1 2 3])` | `diag((1,2,3))` | `diag(c(1,2,3))` |
| Matrix inline creation | `DenseMatrix((1.0,2.0), (3.0,4.0))` | `[1 2; 3 4]` | `array([ [1,2], [3,4] ])` | `matrix(c(1,2,3,4), nrow = 2, ncol = 2)` |
| Column vector inline creation | `DenseVector(1,2,3,4)` | `[1 2 3 4]` | `array([1,2,3,4])` | `c(1,2,3,4)` |
| Row vector | | | | |

| | | | | |
|---|---|---|---|---|
| inline creation | `DenseVector(1,2,3,4).t` | `[1 2 3 4]'` | `array([1,2,3]).reshape(-1,1)` | `t(c(1,2,3,4))` |
| Vector from function | `DenseVector.tabulate(3){i => 2*i}` | | | |
| Matrix from function | `DenseMatrix.tabulate(3, 2){case (i, j) => i+j}` | | | |
| Vector creation from array | `new DenseVector(Array(1, 2, 3, 4))` | | | |
| Matrix creation from array | `new DenseMatrix(2, 3, Array(11, 12, 13, 21, 22, 23))` | | | |
| Vector of random elements from 0 to 1 | `DenseVector.rand(4)` | | | `runif(4)` (requires stats library) |
| Matrix of random elements from 0 to 1 | `DenseMatrix.rand(2, 3)` | | | `matrix(runif(6),2)` (requires stats library) |

#### Reading and writing Matrices

Currently, Breeze supports IO for Matrices in two ways: Java serialization and csv. The latter comes from two functions: `breeze.linalg.csvread` and `breeze.linalg.csvwrite`. `csvread` takes a File, and optionally parameters for how the CSV file is delimited (e.g. if it is actually a tsv file, you can set tabs as the field delimiter.) and returns a DenseMatrix. Similarly, `csvwrite` takes a File and a DenseMatrix, and writes the contents of a matrix to a file.

## Indexing and Slicing

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Basic Indexing | `a(0,1)` | `a(1,2)` | `a[0,1]` | `a[1,2]` |
| Extract subset of vector | `a(1 to 4)` or `a(1 until 5)` or `a.slice(1,5)` | `a(2:5)` | `a[1:5]` | `a[2:5]` |
| (negative steps) | `a(5 to 0 by -1)` | `a(6:-1:1)` | `a[5::-1]` | |
| (tail) | `a(1 to -1)` | `a(2:end)` | `a[1:]` | `a[2:length(a)]` or `tail(a,n=length(a)-1)` |
| (last element) | `a( -1 )` | `a(end)` | `a[-1]` | `tail(a, n=1)` |
| Extract column of matrix | `a(::, 2)` | `a(:,3)` | `a[:,2]` | `a[,2]` |

## Other Manipulation

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Reshaping | `a.reshape(3, 2)` | `reshape(a, 3, 2)` | `a.reshape(3,2)` | `matrix(a,nrow=3,byrow=T)` |
| Flatten matrix | `a.toDenseVector` (Makes copy) | `a(:)` | `a.flatten()` | `as.vector(a)` |
| Copy lower triangle | `lowerTriangular(a)` | `tril(a)` | `tril(a)` | `a[upper.tri(a)] <- 0` |
| Copy upper triangle | `upperTriangular(a)` | `triu(a)` | `triu(a)` | `a[lower.tri(a)] <- 0` |
| Copy (note, no parens!!) | `a.copy` | | `np.copy(a)` | |
| Create view of matrix diagonal | `diag(a)` | NA | `diagonal(a)` (Numpy >= 1.9) | `diag(a)` |

| | | | | |
|---|---|---|---|---|
| Vector Assignment to subset | `a(1 to 4) := 5.0` | `a(2:5) = 5` | `a[1:5] = 5` | `a[2:5] = 5` |
| Vector Assignment to subset | `a(1 to 4) := DenseVector(1.0,2.0,3.0,4.0)` | `a(2:5) = [1 2 3 4]` | `a[1:5] = array([1,2,3,4])` | `a[2:5] = c(1,2,3,4)` |
| Matrix Assignment to subset | `a(1 to 3,1 to 3) := 5.0` | `a(2:4,2:4) = 5` | `a[1:4,1:4] = 5` | `a[2:4,2:4] = 5` |
| Matrix Assignment to column | `a(::, 2) := 5.0` | `a(:,3) = 5` | `a[:,2] = 5` | `a[,3] = 5` |
| Matrix vertical concatenate | `DenseMatrix.vertcat(a,b)` | `[a ; b]` | `vstack((a,b))` | `rbind(a, b)` |
| Matrix horizontal concatenate | `DenseMatrix.horzcat(d,e)` | `[d , e]` | `hstack((d,e))` | `cbind(d, e)` |
| Vector concatenate | `DenseVector.vertcat(a,b)` | `[a b]` | `concatenate((a,b))` | `c(a, b)` |

## Operations

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Elementwise addition | `a + b` | `a + b` | `a + b` | `a + b` |
| Shaped/Matrix multiplication | `a * b` | `a * b` | `dot(a, b)` | `a %*% b` |
| Elementwise multiplication | `a :* b` | `a .* b` | `a * b` | `a * b` |
| Elementwise division | `a :/ b` | `a ./ b` | `a / b` | `a / b` |
| Elementwise comparison | `a :< b` | `a < b` (gives matrix of 1/0 instead of true/false) | `a < b` | `a < b` |
| Elementwise equals | `a :== b` | `a == b` (gives matrix of 1/0 instead of true/false) | `a == b` | `a == b` |
| Inplace addition | `a :+= 1.0` | `a += 1` | `a += 1` | `a = a + 1` |
| Inplace elementwise multiplication | `a :*= 2.0` | `a *= 2` | `a *= 2` | `a = a * 2` |
| Vector dot product | `a dot b`, `a.t * b`[†] | `dot(a,b)` | `dot(a,b)` | `crossprod(a,b)` |
| Elementwise max | `max(a)` | `max(a)` | `a.max()` | `max(a)` |
| Elementwise argmax | `argmax(a)` | `[v i] = max(a); i` | `a.argmax()` | `which.max(a)` |

## Sum

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Elementwise sum | `sum(a)` | `sum(sum(a))` | `a.sum()` | `sum(a)` |
| Sum down each column (giving a row vector) | `sum(a, Axis._0)` or `sum(a(::, *))` | `sum(a)` | `sum(a,0)` | `apply(a,2,sum)` |
| Sum across each row (giving a column vector) | `sum(a, Axis._1)` or `sum(a(*, ::))` | `sum(a')` | `sum(a,1)` | `apply(a,1,sum)` |
| Trace (sum of diagonal elements) | `trace(a)` | `trace(a)` | `a.trace()` | `sum(diag(a))` |
| Cumulative sum | `accumulate(a)` | `cumsum(a)` | `a.cumsum()` | `apply(a,2,cumsum)` |

## Boolean Operators

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Elementwise and | `a :& b` | `a && b` | `a & b` | `a & b` |
| Elementwise or | `a :\| b` | `a \|\| b` | `a \| b` | `a \| b` |
| Elementwise xor | `a ^^ b` | | `a ^ b` | bitwXor(a, b) |

| | | | | |
|---|---|---|---|---|
| Elementwise not | `!a` | `~a` | `~a` | `!a` |
| True if any element is nonzero | `any(a)` | `any(a)` | any(a) | |
| True if all elements are nonzero | `all(a)` | `all(a)` | all(a) | |

### Linear Algebra Functions

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Linear solve | `a \ b` | `a \ b` | `linalg.solve(a,b)` | `solve(a,b)` |
| Transpose | `a.t` | `a'` | `a.conj.transpose()` | `t(a)` |
| Determinant | `det(a)` | `det(a)` | `linalg.det(a)` | `det(a)` |
| Inverse | `inv(a)` | `inv(a)` | `linalg.inv(a)` | `solve(a)` |
| Moore-Penrose Pseudoinverse | `pinv(a)` | `pinv(a)` | `linalg.pinv(a)` | |
| Vector Frobenius Norm | `norm(a)` | `norm(a)` | `norm(a)` | |
| Eigenvalues (Symmetric) | `eigSym(a)` | `[v,l] = eig(a)` | `linalg.eig(a)[0]` | |
| Eigenvalues | `val (er, ei, _) = eig(a)` (separate real & imaginary part) | `eig(a)` | `linalg.eig(a)[0]` | `eigen(a)$values` |
| Eigenvectors | `eig(a)._3` | `[v,l] = eig(a)` | `linalg.eig(a)[1]` | `eigen(a)$vectors` |
| Singular Value Decomposition | `val svd.SVD(u,s,v) = svd(a)` | `svd(a)` | `linalg.svd(a)` | `svd(a)$d` |
| Rank | `rank(a)` | `rank(a)` | `rank(a)` | `rank(a)` |
| Vector length | `a.length` | `size(a)` | `a.size` | `length(a)` |
| Matrix rows | `a.rows` | `size(a,1)` | `a.shape[0]` | `nrow(a)` |
| Matrix columns | `a.cols` | `size(a,2)` | `a.shape[1]` | `ncol(a)` |

### Rounding and Signs

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Round | `round(a)` | `round(a)` | `around(a)` | `round(a)` |
| Ceiling | `ceil(a)` | `ceil(a)` | `ceil(a)` | `ceiling(a)` |
| Floor | `floor(a)` | `floor(a)` | `floor(a)` | `floor(a)` |
| Sign | `signum(a)` | `sign(a)` | `sign(a)` | `sign(a)` |
| Absolute Value | `abs(a)` | `abs(a)` | `abs(a)` | `abs(a)` |

### Constants

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Not a Number | `NaN` or `nan` | `NaN` | `nan` | `NA` |
| Infinity | `Inf` or `inf` | `Inf` | `inf` | `Inf` |
| Pi | `Constants.Pi` | `pi` | `math.pi` | `pi` |
| e | `Constants.E` | `exp(1)` | `math.e` | `exp(1)` |

## Complex numbers

If you make use of complex numbers, you will want to include a `breeze.math._` import. This declares a `i` variable, and provides implicit conversions from Scala's basic types to complex types.

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Imaginary unit | `i` | `i` | `z = 1j` | `1i` |
| Complex numbers | `3 + 4 * i` or `Complex(3,4)` | `3 + 4i` | `z = 3 + 4j` | `3 + 4i` |
| Absolute Value | `abs(z)` or `z.abs` | `abs(z)` | `abs(z)` | `abs(z)` |
| Real Component | `z.real` | `real(z)` | `z.real` | `Re(z)` |
| Imaginary Component | `z.imag` | `imag(z)` | `z.imag()` | `Im(z)` |
| Imaginary Conjugate | `z.conjugate` | `conj(z)` | `z.conj()` or `z.conjugate()` | `Conj(z)` |

## Numeric functions

Breeze contains a fairly comprehensive set of special functions under the `breeze.numerics._` import. These functions can be applied to single elements, vectors or matrices of Doubles. This includes versions of the special functions from `scala.math` that can be applied to vectors and matrices. Any function acting on a basic numeric type can "vectorized", to a UFunc function, which can act elementwise on vectors and matrices:

```
val v = DenseVector(1.0,2.0,3.0)
exp(v) // == DenseVector(2.7182818284590455, 7.38905609893065, 20.085536923187668)
```

UFuncs can also be used in-place on Vectors and Matrices:

```
val v = DenseVector(1.0,2.0,3.0)
exp.inPlace(v) // == DenseVector(2.7182818284590455, 7.38905609893065, 20.085536923187668)
```

See Universal Functions for more information.

Here is a (non-exhaustive) list of UFuncs in Breeze:

### Trigonometry

- `sin`, `sinh`, `asin`, `asinh`
- `cos`, `cosh`, `acos`, `acosh`
- `tan`, `tanh`, `atan`, `atanh`
- `atan2`
- `sinc(x) == sin(x)/x`
- `sincpi(x) == sinc(x * Pi)`

### Logarithm, Roots, and Exponentials

- `log`, `exp` `log10`
- `log1p`, `expm1`
- `sqrt`, `sbrt`
- `pow`

### Gamma Function and its cousins

The gamma function is the extension of the factorial function to the reals. Numpy needs `from scipy.special import *` for this and subsequent sections.

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Gamma function | `exp(lgamma(a))` | `gamma(a)` | `gamma(a)` | `gamma(a)` |
| log Gamma function | `lgamma(a)` | `gammaln(a)` | `gammaln(a)` | `lgamma(a)` |
| Incomplete gamma function | `gammp(a, x)` | `gammainc(a, x)` | `gammainc(a, x)` | `pgamma(a, x)` (requires stats library) |
| Upper incomplete gamma function | `gammq(a, x)` | `gammainc(a, x, tail)` | `gammaincc(a, x)` | `pgamma(x, a, lower = FALSE) * gamma(a)` (requires stats library) |
| derivative of lgamma | `digamma(a)` | `psi(a)` | `polygamma(0, a)` | `digamma(a)` |

| | | | | |
|---|---|---|---|---|
| derivative of digamma | `trigamma(a)` | `psi(1, a)` | `polygamma(1, a)` | `trigama(a)` |
| nth derivative of digamma | na | `psi(n, a)` | `polygamma(n, a)` | `psigamma(a, deriv = n)` |
| Log Beta function | lbeta(a,b) | `betaln(a, b)` | `betaln(a,b)` | `lbeta(a, b)` |
| Generalized Log Beta function | lbeta(a) | na | na | |

### Error Function

The error function...

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| error function | `erf(a)` | `erf(a)` | `erf(a)` | `2 * pnorm(a * sqrt(2)) - 1` |
| 1 - erf(a) | `erfc(a)` | `erfc(a)` | `erfc(a)` | `2 * pnorm(a * sqrt(2), lower = FALSE)` |
| inverse error function | `erfinv(a)` | `erfinv(a)` | `erfinv(a)` | `qnorm((1 + a) / 2) / sqrt(2)` |
| inverse erfc | `erfcinv(a)` | `erfcinv(a)` | `erfcinv(a)` | `qnorm(a / 2, lower = FALSE) / sqrt(2)` |

### Other functions

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| logistic sigmoid | `sigmoid(a)` | na | `expit(a)` | `sigmoid(a)` (requires pracma library) |
| Indicator function | `I(a)` | not needed | `where(cond, 1, 0)` | `0 + (a > 0)` |
| Polynominal evaluation | `polyval(coef,x)` | | | |

### Map and Reduce

For most simple mapping tasks, one can simply use vectorized, or universal functions. Given a vector `v`, we can simply take the log of each element of a vector with `log(v)`. Sometimes, however, we want to apply a somewhat idiosyncratic function to each element of a vector. For this, we can use the map function:

```
val v = DenseVector(1.0,2.0,3.0)
v.map( xi => foobar(xi) )
```

Breeze provides a number of built in reduction functions such as sum, mean. You can implement a custom reduction using the higher order function `reduce`. For instance, we can sum the first 9 integers as follows:

```
val v = linspace(0,9,10)
val s = v.reduce( _ + _ )
```

## Broadcasting

Sometimes we want to apply an operation to every row or column of a matrix, as a unit. For instance, you might want to compute the mean of each row, or add a vector to every column. Adapting a matrix so that operations can be applied columnwise or rowwise is called **broadcasting**. Languages like R and numpy automatically and implicitly do broadcasting, meaning they won't stop you if you accidentally add a matrix and a vector. In Breeze, you have to signal your intent using the broadcasting operator `*`. The `*` is meant to evoke "foreach" visually. Here are some examples:

```
val dm = DenseMatrix((1.0,2.0,3.0),
                     (4.0,5.0,6.0))

val res = dm(::, *) + DenseVector(3.0, 4.0)
assert(res === DenseMatrix((4.0, 5.0, 6.0), (8.0, 9.0, 10.0)))

res(::, *) := DenseVector(3.0, 4.0)
assert(res === DenseMatrix((3.0, 3.0, 3.0), (4.0, 4.0, 4.0)))

val m = DenseMatrix((1.0, 3.0), (4.0, 4.0))
// unbroadcasted sums all elements
assert(sum(m) === 12.0)
```

```
assert(mean(m) === 3.0)

assert(sum(m(*, ::)) === DenseVector(4.0, 8.0))
assert(sum(m(::, *)) === DenseMatrix((5.0, 7.0)))

assert(mean(m(*, ::)) === DenseVector(2.0, 4.0))
assert(mean(m(::, *)) === DenseMatrix((2.5, 3.5)))
```

The UFunc trait is similar to numpy's ufunc. See Universal Functions for more information on Breeze UFuncs.

## Casting and type safety

Compared to Numpy and Matlab, Breeze requires you to be more explicit about the types of your variables. When you create a new vector for example, you must specify a type (such as in `DenseVector.zeros[Double](n)`) in cases where a type can not be inferred automatically. Automatic inference will occur when you create a vector by passing its initial values in (`DenseVector`). A common mistake is using integers for initialisation (e.g. `DenseVector`), which would give a matrix of integers instead of doubles. Both Numpy and Matlab would default to doubles instead.

Breeze will not convert integers to doubles for you in most expressions. Simple operations like `a :+ 3` when `a` is a `DenseVector[Double]` will not compile. Breeze provides a convert function, which can be used to explicitly cast. You can also use `v.mapValues(_.toDouble)`.

### Casting

| Operation | Breeze | Matlab | Numpy | R |
|---|---|---|---|---|
| Convert to Int | `convert(a, Int)` | `int(a)` | `a.astype(int)` | `as.integer(a)` |

## Performance

Breeze uses netlib-java for its core linear algebra routines. This includes all the cubic time operations, matrix-matrix and matrix-vector multiplication. Special efforts are taken to ensure that arrays are not copied.

Netlib-java will attempt to load system optimised BLAS/LAPACK if they are installed, falling back to the reference natives, falling back to pure Java. Set your logger settings to `ALL` for the `com.github.fommil.netlib` package to check the status, and to `com.github.fommil.jniloader` for a more detailed breakdown. Read the netlib-java project page for more details.

Currently vectors and matrices over types other than `Double`, `Float` and `Int` are boxed, so they will typically be a lot slower. If you find yourself needing other AnyVal types like `Long` or `Short`, please ask on the list about possibly adding support for them.