Armadillo

C++ library for linear algebra & scientific computing

API Documentation for Armadillo 9.900

Preamble

- For converting Matlab/Octave programs, see the syntax conversion table
- First time users: please see the short example program
- If you discover any bugs or regressions, please report them
- History of API additions

- Please cite the following papers if you use Armadillo in your research and/or software.
  Citations are useful for the continued development and maintenance of the library.

  Conrad Sanderson and Ryan Curtin.
  Armadillo: a template-based C++ library for linear algebra.
  Journal of Open Source Software, Vol. 1, pp. 26, 2016.

  Conrad Sanderson and Ryan Curtin.
  A User-Friendly Hybrid Sparse Matrix Class in C++.
  Lecture Notes in Computer Science (LNCS),
  Vol. 10931, pp. 422-430, 2018.

Overview

- matrix, vector, cube and field classes
- member functions & variables

- generated vectors / matrices / cubes
- functions of vectors / matrices / cubes

- decompositions, factorisations, inverses and equation solvers (dense matrices)
- decompositions, factorisations, and equation solvers (sparse matrices)

- signal & image processing
- statistics and clustering
- miscellaneous (constants, configuration)

Matrix, Vector, Cube and Field Classes

Mat<*type*>, mat, cx_mat	dense matrix class
Col<*type*>, colvec, vec	dense column vector class
Row<*type*>, rowvec	dense row vector class
Cube<*type*>, cube, cx_cube	dense cube class ("3D matrix")
field<*object type*>	class for storing arbitrary objects in matrix-like or cube-like layouts
SpMat<*type*>, sp_mat, sp_cx_mat	sparse matrix class
operators	`+ - * / % == != <= >= < >`

Member Functions & Variables

attributes	.n_rows, .n_cols, .n_elem, .n_slices, ...
element access	element/object access via (), [] and .at()
element initialisation	set elements via << operator or initialiser list
.zeros	set all elements to zero
.ones	set all elements to one
.eye	set elements along main diagonal to one and off-diagonal elements to zero
.randu / .randn	set all elements to random values
.fill	set all elements to specified value
.imbue	imbue (fill) with values provided by functor or lambda function
.clean	replace elements below a threshold with zeros
.replace	replace specific elements with a new value
.transform	transform each element via functor or lambda function
.for_each	apply a functor or lambda function to each element
.set_size	change size without keeping elements (fast)
.reshape	change size while keeping elements
.resize	change size while keeping elements and preserving layout
.copy_size	change size to be same as given object
.reset	change size to empty
submatrix views	read/write access to contiguous and non-contiguous submatrices
subcube views	read/write access to contiguous and non-contiguous subcubes
subfield views	read/write access to contiguous subfields
.diag	read/write access to matrix diagonals
.each_col / .each_row	repeated operations on each column or row of matrix
.each_slice	repeated operations on each slice of cube
.set_imag / .set_real	set imaginary/real part
.insert_rows/cols/slices	insert vector/matrix/cube at specified row/column/slice
.shed_rows/cols/slices	remove specified rows/columns/slices

.swap_rows/cols	swap specified rows or columns
.swap	swap contents with given object
.memptr	raw pointer to memory
.colptr	raw pointer to memory used by specified column
iterators (matrices)	iterators and associated member functions for dense matrices and vectors
iterators (cubes)	iterators and associated member functions for cubes
iterators (sparse matrices)	iterators and associated member functions for sparse matrices
iterators (submatrices)	iterators and associated member functions for submatrices & subcubes
compat. container functions	compatibility container functions
.as_col / .as_row	return flattened matrix as column or row vector
.t / .st	return matrix transpose
.i	return inverse of square matrix
.min / .max	return extremum value
.index_min / .index_max	return index of extremum value
.eval	force evaluation of delayed expression
.in_range	check whether given location or span is valid
.is_empty	check whether object is empty
.is_vec	check whether matrix is a vector
.is_sorted	check whether vector or matrix is sorted
.is_trimatu / .is_trimatl	check whether matrix is upper/lower triangular
.is_diagmat	check whether matrix is diagonal
.is_square	check whether matrix is square sized
.is_symmetric	check whether matrix is symmetric
.is_hermitian	check whether matrix is hermitian
.is_sympd	check whether matrix is symmetric/hermitian positive definite
.is_zero	check whether all elements are zero
.is_finite	check whether all elements are finite
.has_inf	check whether any element is +-Inf
.has_nan	check whether any element is NaN
.print	print object to *std::cout* or user specified stream
.raw_print	print object without formatting
.save/.load (matrices & cubes)	save/load matrices and cubes in files or streams
.save/.load (fields)	save/load fields in files or streams

Generated Vectors/Matrices/Cubes

linspace	generate vector with linearly spaced elements
logspace	generate vector with logarithmically spaced elements
regspace	generate vector with regularly spaced elements
randperm	generate vector with random permutation of a sequence of integers

eye	generate identity matrix
ones	generate object filled with ones
zeros	generate object filled with zeros
randu / randn	generate object with random values (uniform and normal distributions)
randg	generate object with random values (gamma distribution)
randi	generate object with random integer values in specified interval
speye	generate sparse identity matrix
spones	generate sparse matrix with non-zero elements set to one
sprandu / sprandn	generate sparse matrix with non-zero elements set to random values
toeplitz	generate Toeplitz matrix

Functions of Vectors/Matrices/Cubes

abs	obtain magnitude of each element
accu	accumulate (sum) all elements
affmul	affine matrix multiplication
all	check whether all elements are non-zero, or satisfy a relational condition
any	check whether any element is non-zero, or satisfies a relational condition
approx_equal	approximate equality
arg	phase angle of each element
as_scalar	convert 1x1 matrix to pure scalar
clamp	obtain clamped elements according to given limits
cond	condition number of matrix
conj	obtain complex conjugate of each element
conv_to	convert between matrix types
cross	cross product
cumsum	cumulative sum
cumprod	cumulative product
det	determinant
diagmat	generate diagonal matrix from given matrix or vector
diagvec	extract specified diagonal
diff	differences between adjacent elements
dot / cdot / norm_dot	dot product
eps	obtain distance of each element to next largest floating point representation
expmat	matrix exponential
expmat_sym	matrix exponential of symmetric matrix
find	find indices of non-zero elements, or elements satisfying a relational condition
find_finite	find indices of finite elements
find_nonfinite	find indices of non-finite elements
find_unique	find indices of unique elements
fliplr / flipud	flip matrix left to right or upside down
imag / real	extract imaginary/real part
ind2sub	convert linear index to subscripts
index_min / index_max	indices of extremum values
inplace_trans	in-place transpose
intersect	find common elements in two vectors/matrices
join_rows / join_cols	concatenation of matrices
join_slices	concatenation of cubes
kron	Kronecker tensor product

log_det	log determinant
logmat	matrix logarithm
logmat_sympd	matrix logarithm of symmetric matrix
min / max	return extremum values
nonzeros	return non-zero values
norm	various norms of vectors and matrices
normalise	normalise vectors to unit *p*-norm
prod	product of elements
powmat	matrix power
rank	rank of matrix
rcond	reciprocal of condition number
repelem	replicate elements
repmat	replicate matrix in block-like fashion
reshape	change size while keeping elements
resize	change size while keeping elements and preserving layout
reverse	reverse order of elements
roots	roots of polynomial
shift	shift elements
shuffle	randomly shuffle elements
size	obtain dimensions of given object
sort	sort elements
sort_index	vector describing sorted order of elements
sqrtmat	square root of matrix
sqrtmat_sympd	square root of symmetric matrix
sum	sum of elements
sub2ind	convert subscripts to linear index
symmatu / symmatl	generate symmetric matrix from given matrix
trace	sum of diagonal elements
trans	transpose of matrix
trapz	trapezoidal numerical integration
trimatu / trimatl	copy upper/lower triangular part
trimatu_ind / trimatl_ind	obtain indices of upper/lower triangular part
unique	return unique elements
vectorise	flatten matrix into vector
misc functions	miscellaneous element-wise functions: exp, log, pow, sqrt, round, sign, ...
trig functions	trigonometric element-wise functions: cos, sin, ...

Decompositions, Factorisations, Inverses and Equation Solvers (Dense Matrices)

chol	Cholesky decomposition
eig_sym	eigen decomposition of dense symmetric/hermitian matrix
eig_gen	eigen decomposition of dense general square matrix
eig_pair	eigen decomposition for pair of general dense square matrices
hess	upper Hessenberg decomposition
inv	inverse of general square matrix
inv_sympd	inverse of symmetric positive definite matrix
lu	lower-upper decomposition
null	orthonormal basis of null space
orth	orthonormal basis of range space

pinv	pseudo-inverse
qr	QR decomposition
qr_econ	economical QR decomposition
qz	generalised Schur decomposition
schur	Schur decomposition
solve	solve systems of linear equations
svd	singular value decomposition
svd_econ	economical singular value decomposition
syl	Sylvester equation solver

Decompositions, Factorisations and Equation Solvers (Sparse Matrices)

eigs_sym	limited number of eigenvalues & eigenvectors of sparse symmetric real matrix
eigs_gen	limited number of eigenvalues & eigenvectors of sparse general square matrix
spsolve	solve sparse systems of linear equations
svds	truncated svd: limited number of singular values & singular vectors of sparse matrix

Signal & Image Processing

conv	1D convolution
conv2	2D convolution
fft / ifft	1D fast Fourier transform and its inverse
fft2 / ifft2	2D fast Fourier transform and its inverse
interp1	1D interpolation
interp2	2D interpolation
polyfit	find polynomial coefficients for data fitting
polyval	evaluate polynomial

Statistics & Clustering

stats functions	mean, median, standard deviation, variance
cov	covariance
cor	correlation
hist	histogram of counts
histc	histogram of counts with user specified edges
quantile	quantiles of a dataset
princomp	principal component analysis (PCA)
normpdf	probability density function of normal distribution
log_normpdf	logarithm version of probability density function of normal distribution
normcdf	cumulative distribution function of normal distribution
mvnrnd	random vectors from multivariate normal distribution
chi2rnd	random numbers from chi-squared distribution
wishrnd	random matrix from Wishart distribution
iwishrnd	random matrix from inverse Wishart distribution
running_stat	running statistics of scalars (one dimensional process/signal)
running_stat_vec	running statistics of vectors (multi-dimensional process/signal)
kmeans	cluster data into disjoint sets
gmm_diag/gmm_full	model and evaluate data using Gaussian Mixture Models (GMMs)

Miscellaneous

constants	pi, inf, NaN, speed of light, ...
wall_clock	timer for measuring number of elapsed seconds
logging of errors/warnings	how to change the streams for displaying warnings and errors
uword / sword	shorthand for unsigned and signed integers
cx_double / cx_float	shorthand for std::complex<double> and std::complex<float>
Matlab/Armadillo syntax differences	examples of Matlab syntax and conceptually corresponding Armadillo syntax
example program	short example program
config.hpp	configuration options
API additions	API stability and list of API additions

Matrix, Vector, Cube and Field Classes

Mat<*type*>
mat
cx_mat

- Classes for dense matrices, with elements stored in column-major ordering (ie. column by column)
- The root matrix class is **Mat<*type*>**, where *type* is one of:
  - *float*, *double*, *std::complex<float>*, *std::complex<double>*, *short*, *int*, *long*, and unsigned versions of *short*, *int*, *long*
- For convenience the following typedefs have been defined:

```
mat = Mat<double>

dmat = Mat<double>
fmat = Mat<float>
cx_mat = Mat<cx_double>
cx_dmat = Mat<cx_double>
cx_fmat = Mat<cx_float>
umat = Mat<uword>
imat = Mat<sword>
```

- In this documentation the *mat* type is used for convenience; it is possible to use other types instead, eg. *fmat*
- Functions which use LAPACK or ATLAS (generally matrix decompositions) are only valid for the following types: *mat*, *dmat*, *fmat*, *cx_mat*, *cx_dmat*, *cx_fmat*
- Constructors:

```
mat()
mat(n_rows, n_cols)
```
(memory is not initialised)

`mat(n_rows, n_cols, fill_type)`	(memory is initialised)
`mat(size(X))`	(memory is not initialised)
`mat(size(X), fill_type)`	(memory is initialised)
`mat(mat)`	
`mat(vec)`	
`mat(rowvec)`	
`mat(initializer_list)`	
`mat(string)`	
`mat(std::vector)`	(treated as a column vector)
`mat(sp_mat)`	(for converting a sparse matrix to a dense matrix)
`cx_mat(mat,mat)`	(for constructing a complex matrix out of two real matrices)

- When using the *mat(n_rows, n_cols)* or *mat(size(X))* constructors, by default the memory is uninitialised (ie. may contain garbage); memory can be explicitly initialised by specifying the *fill_type*, which is one of: *fill::zeros*, *fill::ones*, *fill::eye*, *fill::randu*, *fill::randn*, *fill::none*, with the following meanings:

`fill::zeros`	= set all elements to 0
`fill::ones`	= set all elements to 1
`fill::eye`	= set the elements along the main diagonal to 1 and off-diagonal elements to 0
`fill::randu`	= set each element to a random value from a uniform distribution in the [0,1] interval
`fill::randn`	= set each element to a random value from a normal/Gaussian distribution with zero mean and unit variance
`fill::none`	= do not modify the elements

- When using the *mat(string)* constructor, the format is elements separated by spaces, and rows denoted by semicolons; for example, the 2x2 identity matrix can be created using `"1 0; 0 1"`.
  Caveat: string based initialisation is slower than directly setting the elements or using element initialisation.

- Advanced constructors:

```
mat(ptr_aux_mem, n_rows, n_cols, copy_aux_mem = true, strict = false)
```

Create a matrix using data from writable auxiliary (external) memory, where *ptr_aux_mem* is a pointer to the memory. By default the matrix allocates its own memory and copies data from the auxiliary memory (for safety). However, if *copy_aux_mem* is set to *false*, the matrix will instead directly use the auxiliary memory (ie. no copying); this is faster, but can be dangerous unless you know what you are doing!

The *strict* parameter comes into effect only when *copy_aux_mem* is set to *false* (ie. the matrix is directly using auxiliary memory)

- when *strict* is set to *false*, the matrix will use the auxiliary memory until a size change
- when *strict* is set to *true*, the matrix will be bound to the auxiliary memory for its lifetime; the number of elements in the matrix can't be changed
- the default setting of *strict* in versions 6.000+ is *false*
- the default setting of *strict* in versions 5.600 and earlier is *true*

```
mat(const ptr_aux_mem, n_rows, n_cols)
```

Create a matrix by copying data from read-only auxiliary memory, where *ptr_aux_mem* is a pointer to the memory

```
mat::fixed<n_rows, n_cols>
```

Create a fixed size matrix, with the size specified via template arguments. Memory for the matrix is reserved at compile time. This is generally faster than dynamic memory allocation, but the size of the matrix can't be changed afterwards (directly or indirectly).

For convenience, there are several pre-defined typedefs for each matrix type (where the types are: *umat*, *imat*, *fmat*, *mat*, *cx_fmat*, *cx_mat*). The typedefs specify a square matrix size, ranging from 2x2 to 9x9. The typedefs were defined by simply appending a two digit form of the size to the matrix type -- for example, *mat33* is equivalent to *mat::fixed<3,3>*, while *cx_mat44* is equivalent to *cx_mat::fixed<4,4>*.

```
mat::fixed<n_rows, n_cols>(const ptr_aux_mem)
```

Create a fixed size matrix, with the size specified via template arguments; data is copied from auxiliary memory, where *ptr_aux_mem* is a pointer to the memory

- Examples:

```
mat A(5, 5, fill::randu);
double x = A(1,2);

mat B = A + A;
mat C = A * B;
mat D = A % B;

cx_mat X(A,B);

B.zeros();
B.set_size(10,10);
B.ones(5,6);

B.print("B:");

mat::fixed<5,6> F;

double aux_mem[24];
mat H(&aux_mem[0], 4, 6, false);  // use auxiliary memory
```

- **Caveat:** For mathematical correctness, scalars are treated as 1x1 matrices during initialisation. As such, the code below **will not** generate a 5x5 matrix with every element equal to 123.0:

```
mat A(5,5);  A = 123.0;
```

Use the following code instead:

```
mat A(5,5);  A.fill(123.0);
```

- See also:
  - matrix attributes
  - accessing elements
  - initialising elements
  - math & relational operators
  - submatrix views
  - saving & loading matrices
  - printing matrices
  - element iterators

- .eval()
- conv_to() (convert between matrix types)
- explanation of *typedef* (cplusplus.com)
- Col class
- Row class
- Cube class
- SpMat class (sparse matrix with compressed sparse column format)
- config.hpp

Col<*type*>
vec
cx_vec

- Classes for column vectors (dense matrices with one column)
- The **Col<*type*>** class is derived from the **Mat<*type*>** class and inherits most of the member functions
- For convenience the following typedefs have been defined:

```
   vec =     colvec = Col<double>
  dvec =    dcolvec = Col<double>
  fvec =    fcolvec = Col<float>
cx_vec =  cx_colvec = Col<cx_double>
cx_dvec = cx_dcolvec = Col<cx_double>
cx_fvec = cx_fcolvec = Col<cx_float>
  uvec =    ucolvec = Col<uword>
  ivec =    icolvec = Col<sword>
```

- In this documentation, the ***vec*** and ***colvec*** types have the **same meaning** and are used **interchangeably**
- In this documentation, the types *vec* or *colvec* are used for convenience; it is possible to use other types instead, eg. *fvec*, *fcolvec*
- Functions which take *Mat* as input can generally also take *Col* as input; main exceptions are functions which require square matrices
- Constructors:

`vec()`	
`vec(n_elem)`	(memory is not initialised)
`vec(n_elem, fill_type)`	(memory is initialised)
`vec(size(X))`	(memory is not initialised)
`vec(size(X), fill_type)`	(memory is initialised)
`vec(vec)`	
`vec(mat)`	(*std::logic_error* exception is thrown if the given matrix has more than one column)
`vec(initializer_list)`	
`vec(string)`	(elements separated by spaces)
`vec(std::vector)`	

`cx_vec(vec,vec)` (for constructing a complex vector out of two real vectors)

- When using the *vec(n_elem)* or *vec(size(X))* constructors, by default the memory is uninitialised (ie. may contain garbage); memory can be explicitly initialised by specifying the *fill_type*, as per the Mat class
- Advanced constructors:

```
vec(ptr_aux_mem, number_of_elements, copy_aux_mem = true, strict = false)
```

Create a column vector using data from writable auxiliary (external) memory, where *ptr_aux_mem* is a pointer to the memory. By default the vector allocates its own memory and copies data from the auxiliary memory (for safety). However, if *copy_aux_mem* is set to *false*, the vector will instead directly use the auxiliary memory (ie. no copying); this is faster, but can be dangerous unless you know what you are doing!

The *strict* parameter comes into effect only when *copy_aux_mem* is set to *false* (ie. the vector is directly using auxiliary memory)

- when *strict* is set to *false*, the vector will use the auxiliary memory until a size change
- when *strict* is set to *true*, the vector will be bound to the auxiliary memory for its lifetime; the number of elements in the vector can't be changed
- the default setting of *strict* in versions 6.000+ is *false*
- the default setting of *strict* in versions 5.600 and earlier is *true*

```
vec(const ptr_aux_mem, number_of_elements)
```

Create a column vector by copying data from read-only auxiliary memory, where *ptr_aux_mem* is a pointer to the memory

```
vec::fixed<number_of_elements>
```

Create a fixed size column vector, with the size specified via the template argument. Memory for the vector is reserved at compile time. This is generally faster than dynamic memory allocation, but the size of the vector can't be changed afterwards (directly or indirectly).

For convenience, there are several pre-defined typedefs for each vector type (where the types are: *uvec*, *ivec*, *fvec*, *vec*, *cx_fvec*, *cx_vec* as well as the corresponding *colvec* versions). The pre-defined typedefs specify vector sizes ranging from 2 to 9. The typedefs were defined by simply appending a single digit form of the size to the vector type -- for example, *vec3* is equivalent to *vec::fixed<3>*, while *cx_vec4* is equivalent to *cx_vec::fixed<4>*.

```
vec::fixed<number_of_elements>(const ptr_aux_mem)
```

Create a fixed size column vector, with the size specified via the template argument; data is copied from auxiliary memory, where *ptr_aux_mem* is a pointer to the memory

- Examples:

```
vec x(10);
vec y = zeros<vec>(10);

mat A = randu<mat>(10,10);
vec z = A.col(5); // extract a column vector
```

- **Caveat:** For mathematical correctness, scalars are treated as 1x1 matrices during initialisation. As such, the code below **will not** generate a column vector with every element equal to 123.0:

```
vec a(5);  a = 123.0;
```

  Use the following code instead:

```
vec a(5);  a.fill(123.0);
```

- See also:
  - element initialisation
  - Mat class
  - Row class

Row<*type*>
rowvec
cx_rowvec

- Classes for row vectors (dense matrices with one row)
- The template **Row<*type*>** class is derived from the **Mat<*type*>** class and inherits most of the member functions
- For convenience the following typedefs have been defined:

```
   rowvec = Row<double>
  drowvec = Row<double>
  frowvec = Row<float>
cx_rowvec = Row<cx_double>
cx_drowvec = Row<cx_double>
cx_frowvec = Row<cx_float>
  urowvec = Row<uword>
  irowvec = Row<sword>
```

- In this documentation, the *rowvec* type is used for convenience; it is possible to use other types instead, eg. *frowvec*
- Functions which take *Mat* as input can generally also take *Row* as input. Main exceptions are functions which require square matrices
- Constructors:

`rowvec()`	
`rowvec(n_elem)`	(memory is not initialised)
`rowvec(n_elem, fill_type)`	(memory is initialised)
`rowvec(size(X))`	(memory is not initialised)
`rowvec(size(X), fill_type)`	(memory is initialised)
`rowvec(rowvec)`	
`rowvec(mat)`	(*std::logic_error* exception is thrown if the given matrix has more than one row)

```
rowvec(initializer_list)
rowvec(string)              (elements separated by spaces)
rowvec(std::vector)
cx_rowvec(rowvec,rowvec)    (for constructing a complex row vector out of two real row vectors)
```

- When using the *rowvec(n_elem)* or *rowvec(size(X))* constructors, by default the memory is uninitialised (ie. may contain garbage); memory can be explicitly initialised by specifying the *fill_type*, as per the Mat class
- Advanced constructors:

```
rowvec(ptr_aux_mem, number_of_elements, copy_aux_mem = true, strict = false)
```

Create a row vector using data from writable auxiliary (external) memory, where *ptr_aux_mem* is a pointer to the memory. By default the vector allocates its own memory and copies data from the auxiliary memory (for safety). However, if *copy_aux_mem* is set to *false*, the vector will instead directly use the auxiliary memory (ie. no copying); this is faster, but can be dangerous unless you know what you are doing!

The *strict* parameter comes into effect only when *copy_aux_mem* is set to *false* (ie. the vector is directly using auxiliary memory)

- when *strict* is set to *false*, the vector will use the auxiliary memory until a size change
- when *strict* is set to *true*, the vector will be bound to the auxiliary memory for its lifetime; the number of elements in the vector can't be changed
- the default setting of *strict* in versions 6.000+ is *false*
- the default setting of *strict* in versions 5.600 and earlier is *true*

```
rowvec(const ptr_aux_mem, number_of_elements)
```

Create a row vector by copying data from read-only auxiliary memory, where *ptr_aux_mem* is a pointer to the memory

```
rowvec::fixed<number_of_elements>
```

Create a fixed size row vector, with the size specified via the template argument. Memory for the vector is reserved at compile time. This is generally faster than dynamic memory allocation, but the size of the vector can't be changed afterwards (directly or indirectly).

For convenience, there are several pre-defined typedefs for each vector type (where the types are: *urowvec*, *irowvec*, *frowvec*, *rowvec*, *cx_frowvec*, *cx_rowvec*). The pre-defined typedefs specify vector sizes ranging from 2 to 9. The typedefs were defined by simply appending a single digit form of the size to the vector type -- for example, *rowvec3* is equivalent to *rowvec::fixed<3>*, while *cx_rowvec4* is equivalent to *cx_rowvec::fixed<4>*.

```
rowvec::fixed<number_of_elements>(const ptr_aux_mem)
```

Create a fixed size row vector, with the size specified via the template argument; data is copied from auxiliary memory, where *ptr_aux_mem* is a pointer to the memory

- Examples:

```
rowvec x(10);
rowvec y = zeros<rowvec>(10);
```

```
mat    A = randu<mat>(10,10);
rowvec z = A.row(5); // extract a row vector
```

- **Caveat:** For mathematical correctness, scalars are treated as 1x1 matrices during initialisation. As such, the code below **will not** generate a row vector with every element equal to 123.0:

  ```
  rowvec r(5);  r = 123.0;
  ```

  Use the following code instead:

  ```
  rowvec r(5);  r.fill(123.0);
  ```

- See also:
  - element initialisation
  - Mat class
  - Col class

Cube<*type*>
cube
cx_cube

- Classes for cubes, also known as "3D matrices" or 3rd order tensors
- The cube class is **Cube<*type*>**, where *type* is one of:
  - *float*, *double*, *std::complex<float>*, *std::complex<double>*, *short*, *int*, *long* and unsigned versions of *short*, *int*, *long*
- For convenience the following typedefs have been defined:

  ```
     cube = Cube<double>
    dcube = Cube<double>
    fcube = Cube<float>
  cx_cube = Cube<cx_double>
 cx_dcube = Cube<cx_double>
 cx_fcube = Cube<cx_float>
    ucube = Cube<uword>
    icube = Cube<sword>
  ```

- In this documentation the *cube* type is used for convenience; it is possible to use other types instead, eg. *fcube*
- Cube data is stored as a set of slices (matrices) stored contiguously within memory. Within each slice, elements are stored with column-major ordering (ie. column by column)
- Each slice can be interpreted as a matrix, hence functions which take *Mat* as input can generally also take cube slices as input
- Constructors:

  ```
  cube()
  cube(n_rows, n_cols, n_slices)              (memory is not initialised)
  cube(n_rows, n_cols, n_slices, fill_type)   (memory is initialised)
  ```

`cube(size(X))`	(memory is not initialised)
`cube(size(X), fill_type)`	(memory is initialised)
`cube(cube)`	
`cx_cube(cube, cube)`	(for constructing a complex cube out of two real cubes)

- When using the *cube(n_rows, n_cols, n_slices)* or *cube(size(X))* constructors, by default the memory is uninitialised (ie. may contain garbage); memory can be explicitly initialised by specifying the *fill_type*, as per the Mat class (except for *fill::eye*)

- Advanced constructors:

  `cube::fixed<n_rows, n_cols, n_slices>`

  Create a fixed size cube, with the size specified via template arguments. Memory for the cube is reserved at compile time. This is generally faster than dynamic memory allocation, but the size of the cube can't be changed afterwards (directly or indirectly).

  `cube(ptr_aux_mem, n_rows, n_cols, n_slices, copy_aux_mem = true, strict = false)`

  Create a cube using data from writable auxiliary (external) memory, where *ptr_aux_mem* is a pointer to the memory. By default the cube allocates its own memory and copies data from the auxiliary memory (for safety). However, if *copy_aux_mem* is set to *false*, the cube will instead directly use the auxiliary memory (ie. no copying); this is faster, but can be dangerous unless you know what you are doing!

  The *strict* parameter comes into effect only when *copy_aux_mem* is set to *false* (ie. the cube is directly using auxiliary memory)
  - when *strict* is set to *false*, the cube will use the auxiliary memory until a size change
  - when *strict* is set to *true*, the cube will be bound to the auxiliary memory for its lifetime; the number of elements in the cube can't be changed
  - the default setting of *strict* in versions 6.000+ is *false*
  - the default setting of *strict* in versions 5.600 and earlier is *true*

  `cube(const ptr_aux_mem, n_rows, n_cols, n_slices)`

  Create a cube by copying data from read-only auxiliary memory, where *ptr_aux_mem* is a pointer to the memory

- Examples:

```
cube x(1,2,3);
cube y = randu<cube>(4,5,6);

mat A = y.slice(1);  // extract a slice from the cube
                     // (each slice is a matrix)

mat B = randu<mat>(4,5);
y.slice(2) = B;     // set a slice in the cube

cube q = y + y;     // cube addition
cube r = y % y;     // element-wise cube multiplication

cube::fixed<4,5,6> f;
f.ones();
```

- **Caveats:**
  - The size of individual slices can't be changed. For example, the following **will not** work:

```
cube c(5,6,7);
c.slice(0) = randu<mat>(10,20); // wrong size
```

  - For mathematical correctness, scalars are treated as 1x1x1 cubes during initialisation. As such, the code below **will not** generate a cube with every element equal to 123.0:

```
cube c(5,6,7);  c = 123.0;
```

    Use the following code instead:

```
cube c(5,6,7);  c.fill(123.0);
```

- See also:
  - cube attributes
  - accessing elements
  - math & relational operators
  - subcube views and slices
  - saving & loading cubes
  - element iterators
  - field class
  - Mat class

---

field<*object_type***>**

- Class for storing arbitrary objects in matrix-like or cube-like layouts
- Somewhat similar to a matrix or cube, but instead of each element being a scalar, each element can be a vector, or matrix, or cube
- Each element can have an arbitrary size (eg. in a field of matrices, each matrix can have a different size)
- Constructors, where *object_type* is another class, eg. *vec*, *mat*, *std::string*, etc:

```
field<object_type>()
field<object_type>(n_elem)
field<object_type>(n_rows, n_cols)
field<object_type>(n_rows, n_cols, n_slices)
field<object_type>(size(X))
field<object_type>(field<object_type>)
```

- **Caveat**: to store a set of matrices of the same size, the Cube class is more efficient
- Examples:

```
mat A = randn(2,3);
mat B = randn(4,5);

field<mat> F(2,1);
F(0,0) = A;
```

```
F(1,0) = B;

F.print("F:");

F.save("mat_field");
```

- See also:
  - field attributes
  - subfield views
  - saving/loading fields
  - Cube class

SpMat<*type*>
sp_mat
sp_cx_mat

- Classes for sparse matrices; intended for storing very large matrices, where the vast majority of elements are zero
- The root sparse matrix class is **SpMat<*type*>**, where *type* is one of:
  - *float*, *double*, *std::complex<float>*, *std::complex<double>*, *short*, *int*, *long* and unsigned versions of *short*, *int*, *long*
- For convenience the following typedefs have been defined:

```
   sp_mat = SpMat<double>
  sp_dmat = SpMat<double>
  sp_fmat = SpMat<float>
 sp_cx_mat = SpMat<cx_double>
sp_cx_dmat = SpMat<cx_double>
sp_cx_fmat = SpMat<cx_float>
  sp_umat = SpMat<uword>
  sp_imat = SpMat<sword>
```

- In this documentation the *sp_mat* type is used for convenience; it is possible to use other types instead, eg. *sp_fmat*
- Constructors:

```
sp_mat()
sp_mat(n_rows, n_cols)
sp_mat(size(X))
sp_mat(sp_mat)
sp_mat(mat)                 (for converting a dense matrix to a sparse matrix)
sp_cx_mat(sp_mat,sp_mat)    (for constructing a complex matrix out of two real matrices)
```

- All elements are treated as zero by default (ie. the matrix is initialised to contain zeros)
- Non-zero elements are stored in compressed sparse column (CSC) format (ie. column-major ordering); zero-valued elements are never stored

- This class behaves in a similar manner to the Mat class; however, member functions which set all elements to non-zero values (and hence do not make sense for sparse matrices) have been deliberately omitted; examples of omitted functions: .fill(), .ones(), += scalar, etc.

- Batch insertion constructors:
  - form 1: `sp_mat(locations, values, sort_locations = true)`
  - form 2: `sp_mat(locations, values, n_rows, n_cols, sort_locations = true, check_for_zeros = true)`
  - form 3: `sp_mat(add_values, locations, values, n_rows, n_cols, sort_locations = true, check_for_zeros = true)`
  - form 4: `sp_mat(rowind, colptr, values, n_rows, n_cols)`

    - For forms 1, 2, 3, *locations* is a dense matrix of type *umat*, with a size of *2* x *N*, where *N* is the number of values to be inserted; the location of the *i*-th element is specified by the contents of the *i*-th column of the *locations* matrix, where the row is in *locations(0,i)*, and the column is in *locations(1,i)*

    - For form 4, *rowind* is a dense column vector of type *uvec* containing the row indices of the values to be inserted, and *colptr* is a dense column vector of type *uvec* (with length *n_cols + 1*) containing indices of *values* corresponding to the start of new columns; the vectors correspond to the arrays used by the compressed sparse column format; this form is useful for copying data from other CSC sparse matrix containers

    - For all forms, *values* is a dense column vector containing the values to be inserted; it must have the same element type as the sparse matrix. For forms 1 and 2, the value in *values[i]* will be inserted at the location specified by the *i*-th column of the *locations* matrix.

    - For form 3, *add_values* is either *true* or *false*; when set to *true*, identical locations are allowed, and the values at identical locations are added

    - The size of the constructed matrix is either automatically determined from the maximal locations in the *locations* matrix (form 1), or manually specified via *n_rows* and *n_cols* (forms 2, 3, 4)

    - If *sort_locations* is set to *false*, the *locations* matrix is assumed to contain locations that are already sorted according to column-major ordering; do not set this to *false* unless you know what you are doing!

    - If *check_for_zeros* is set to *false*, the *values* vector is assumed to contain no zero values; do not set this to *false* unless you know what you are doing!

- The following subset of operations & functions is available for sparse matrices:
  - fundamental arithmetic operations (such as addition and multiplication)
  - submatrix views (contiguous forms only)
  - diagonal views
  - saving and loading (using *arma_binary*, *coord_ascii*, and *csv_ascii* formats)
  - element-wise functions: abs(), ceil(), conj(), floor(), imag(), real(), round(), sign(), sqrt(), square(), trunc()
  - scalar functions of matrices: accu(), as_scalar(), dot(), norm(), trace()
  - vector valued functions of matrices: diagvec(), min(), max(), nonzeros(), sum(), mean(), var(), vectorise()
  - matrix valued functions of matrices: clamp(), diagmat(), flipud()/fliplr(), join_rows(), join_cols(), kron(), normalise(), repelem(), repmat(), reshape(), resize(), reverse(), symmatu()/symmatl(), trimatu()/trimatl(), .t(), trans()
  - generated matrices: speye(), spones(), sprandu(), sprandn(), zeros()
  - eigen and svd decomposition: eigs_sym(), eigs_gen(), svds()
  - solution of sparse linear systems: spsolve()

- miscellaneous: approx_equal(), element access, element iterators, .as_col() / .as_row(), .for_each(), .print(), .clean(), .replace(), .transform(), .is_finite(), .is_symmetric(), .is_hermitian(), .is_trimatu(), .is_trimatl(), .is_diagmat()

- **Caveats:**
  - Armadillo 9.600 is the minimum recommended version; it has considerably improved support for sparse matrices compared to earlier versions
  - In old versions of Armadillo (7.x and earlier), incrementally populating sparse matrices via element access operators is inefficient; use batch insertion constructors instead

- Examples:

```
sp_mat A = sprandu(1000, 2000, 0.01);
sp_mat B = sprandu(2000, 1000, 0.01);

sp_mat C = 2*B;
sp_mat D = A*C;

sp_mat E(1000,1000);
E(1,2) = 123;  // element access (Armadillo 8.x and later)


// batch insertion of two values at (5, 6) and (9, 9)
umat locations;
locations << 5 << 9 << endr
          << 6 << 9 << endr;

vec values;
values << 1.5 << 3.2 << endr;

sp_mat X(locations, values);
```

- See also:
  - element access
  - element iterators (sparse matrices)
  - printing matrices
  - Sparse Matrix in Wikipedia
  - Mat class (dense matrix)

---

operators: `+ - * / % == != <= >= < >`

- Overloaded operators for *Mat*, *Col*, *Row* and *Cube* classes

- Meanings:

`+` Addition of two objects

`-` Subtraction of one object from another or negation of an object

`/` Element-wise division of an object by another object or a scalar

`*` Matrix multiplication of two objects; not applicable to the *Cube* class unless multiplying a cube by a scalar

`%` Schur product: element-wise multiplication of two objects

`==` Element-wise equality evaluation of two objects; generates a matrix of type *umat* with entries that indicate whether at a given position the two elements from the two objects are equal (1) or not equal (0)

`!=` Element-wise non-equality evaluation of two objects

`>=` As for `==`, but the check is for "greater than or equal to"

`<=` As for `==`, but the check is for "less than or equal to"

`>` As for `==`, but the check is for "greater than"

`<` As for `==`, but the check is for "less than"

- **Caveat:** operators involving an equality comparison (ie. `==`, `!=`, `>=`, `<=`) are not recommended for matrices of type *mat* or *fmat*, due to the necessarily limited precision of the underlying element types; consider using approx_equal() instead
- A *std::logic_error* exception is thrown if incompatible object sizes are used
- If the `+`, `-` and `%` operators are chained, Armadillo will try to avoid the generation of temporaries; no temporaries are generated if all given objects are of the same type and size
- If the `*` operator is chained, Armadillo will try to find an efficient ordering of the matrix multiplications
- Examples:

```
mat A = randu<mat>(5,10);
mat B = randu<mat>(5,10);
mat C = randu<mat>(10,5);

mat P = A + B;
mat Q = A - B;
mat R = -B;
mat S = A / 123.0;
mat T = A % B;
mat U = A * C;

// V is constructed without temporaries
mat V = A + B + A + B;

imat AA = "1 2 3; 4 5 6; 7 8 9;";
imat BB = "3 2 1; 6 5 4; 9 8 7;";

// compare elements
umat ZZ = (AA >= BB);
```

- See also:
  - approx_equal()
  - powmat()
  - any()
  - all()
  - affmul()
  - accu()
  - as_scalar()

- find()
- .replace()
- .transform()
- .each_col() & .each_row()
- miscellaneous element-wise functions (exp, log, pow, sqrt, square, round, ...)
- floating point representation in Wikipedia
- floating point representation in MathWorld

Member Functions & Variables

---

attributes

`.n_rows`	number of rows; present in *Mat*, *Col*, *Row*, *Cube*, *field* and *SpMat*
`.n_cols`	number of columns; present in *Mat*, *Col*, *Row*, *Cube*, *field* and *SpMat*
`.n_elem`	total number of elements; present in *Mat*, *Col*, *Row*, *Cube*, *field* and *SpMat*
`.n_slices`	number of slices; present in *Cube* and *field*
`.n_nonzero`	number of non-zero elements; present in *SpMat*

- The variables are of type uword
- The variables are read-only; to change the size, use .set_size(), .copy_size(), .zeros(), .ones(), or .reset()
- For the *Col* and *Row* classes, *n_elem* also indicates vector length
- Examples:

```
mat X(4,5);
cout << "X has " << X.n_cols << " columns" << endl;
```

- See also:
  - .set_size()
  - .copy_size()
  - .zeros()
  - .ones()
  - .reset()
  - size()

---

element/object access via (), [] and .at()

- Provide access to individual elements or objects stored in a container object (ie. *Mat*, *Col*, *Row*, *Cube*, *field*)

`(n)`	For *vec* and *rowvec*, access the *n*-th element. For *mat*, *cube* and *field*, access the *n*-th element/object under the assumption of a flat layout, with column-major ordering of data (ie. column by column). A *std::logic_error* exception is thrown if the requested element is out of bounds. The bounds check can be optionally disabled at compile-time to get more speed.
`.at(n)` or `[n]`	As for `(n)`, but without a bounds check. Not recommended for use unless your code has been thoroughly debugged.
`(i,j)`	For *mat* and *2D field* classes, access the element/object stored at the *i*-th row and *j*-th column. A *std::logic_error* exception is thrown if the requested element is out of bounds. The bounds check can be optionally disabled at compile-time to get more speed.
`.at(i,j)`	As for `(i,j)`, but without a bounds check. Not recommended for use unless your code has been thoroughly debugged.
`(i,j,k)`	For *cube* and *3D field* classes, access the element/object stored at the *i*-th row, *j*-th column and *k*-th slice. A *std::logic_error* exception is thrown if the requested element is out of bounds. The bounds check can be optionally disabled at compile-time to get more speed.
`.at(i,j,k)`	As for `(i,j,k)`, but without a bounds check. Not recommended for use unless your code has been thoroughly debugged.

- The bounds checks used by the `(n)`, `(i,j)` and `(i,j,k)` element access forms can be disabled by defining the ARMA_NO_DEBUG macro before including the *armadillo* header file (eg. `#define ARMA_NO_DEBUG`). Alternatively, the `.at(n)`, `.at(i,j)` and `.at(i,j,k)` element access forms can be used, which do not have bounds checks. Either way, **disabling the bounds checks is not recommended** until your code has been thoroughly tested and debugged -- it's better to write correct code first, and then maximise its speed.
- The indices of elements are specified via the uword type, which is a typedef for an unsigned integer type. When using loops to access elements, it's best to use *uword* instead of *int*. For example:
  `for(uword i=0; i<X.n_elem; ++i) { X(i) = ... }`
- **Caveat:** accessing elements via `[i,j]` and `[i,j,k]` does not work correctly in C++; instead use `(i,j)` and `(i,j,k)`
- Examples:

```
mat A = randu<mat>(10,10);
A(9,9) = 123.0;
double x = A.at(9,9);
double y = A[99];

vec p = randu<vec>(10,1);
p(9) = 123.0;
double z = p[9];
```

- See also:
  - .in_range()
  - element initialisation

- ind2sub()
- sub2ind()
- .index_min() / .index_max()
- submatrix views
- .memptr()
- .transform()
- .for_each()
- iterators (dense matrices)
- iterators (cubes)
- iterators (sparse matrices)
- config.hpp

---

element initialisation

- When using the C++11 standard, elements in *Mat*, *Col*, *Row* can be set via initialiser lists
- When using the old C++98 standard, elements can be set via the << operator; special element *endr* indicates "end of row" (conceptually similar to *std::endl*)
- **Caveat:** using the << operator is slower than using initialiser lists
- Examples:

```
// C++11

vec v = { 1, 2, 3 };

mat A = { {1, 3, 5},
          {2, 4, 6} };


// C++98

mat B;

B << 1 << 3 << 5 << endr
  << 2 << 4 << 6 << endr;
```

- See also:
  - element access
  - .reshape()
  - .print()
  - saving & loading matrices
  - advanced constructors (matrices)

.zeros()	(member function of *Mat*, *Col*, *Row*, *SpMat*, *Cube*)
.zeros(n_elem)	(member function of *Col* and *Row*)
.zeros(n_rows, n_cols)	(member function of *Mat* and *SpMat*)
.zeros(n_rows, n_cols, n_slices)	(member function of *Cube*)

.zeros(size(X)) (member function of *Mat*, *Col*, *Row*, *Cube*, *SpMat*)

- Set the elements of an object to zero, optionally first changing the size to specified dimensions
- Examples:

```
mat A(5,10);  A.zeros();   // or:  mat A(5,10,fill::zeros);

mat B;  B.zeros(10,20);

mat C;  C.zeros( size(B) );
```

- See also:
  - zeros() (standalone function)
  - .ones()
  - .clean()
  - .is_zero()
  - .randu()
  - .fill()
  - .imbue()
  - .reset()
  - .set_size()
  - size()

.ones() (member function of *Mat*, *Col*, *Row*, *Cube*)
.ones(n_elem) (member function of *Col* and *Row*)
.ones(n_rows, n_cols) (member function of *Mat*)
.ones(n_rows, n_cols, n_slices) (member function of *Cube*)
.ones(size(X)) (member function of *Mat*, *Col*, *Row*, *Cube*)

- Set all the elements of an object to one, optionally first changing the size to specified dimensions
- Examples:

```
mat A(5,10);  A.ones();   // or:  mat A(5,10,fill::ones);

mat B;  B.ones(10,20);

mat C;  C.ones( size(B) );
```

- See also:
  - ones() (standalone function)
  - .eye()
  - .zeros()
  - .fill()
  - .imbue()
  - .randu()
  - size()

.eye()
.eye(n_rows, n_cols)
.eye(size(X))

- Member functions of *Mat* and *SpMat*
- Set the elements along the main diagonal to one and off-diagonal elements to zero, optionally first changing the size to specified dimensions
- An identity matrix is generated when *n_rows* = *n_cols*
- Examples:

```
mat A(5,5);  A.eye();  // or:  mat A(5,5,fill::eye);

mat B;  B.eye(5,5);

mat C;  C.eye( size(B) );
```

- See also:
  - .ones()
  - .diag()
  - diagmat()
  - diagvec()
  - eye() (standalone function)
  - size()

.randu()	(member function of *Mat*, *Col*, *Row*, *Cube*)
.randu(n_elem)	(member function of *Col* and *Row*)
.randu(n_rows, n_cols)	(member function of *Mat*)
.randu(n_rows, n_cols, n_slices)	(member function of *Cube*)
.randu(size(X))	(member function of *Mat*, *Col*, *Row*, *Cube*)
.randn()	(member function of *Mat*, *Col*, *Row*, *Cube*)
.randn(n_elem)	(member function of *Col* and *Row*)
.randn(n_rows, n_cols)	(member function of *Mat*)
.randn(n_rows, n_cols, n_slices)	(member function of *Cube*)
.randn(size(X))	(member function of *Mat*, *Col*, *Row*, *Cube*)

- Set all the elements to random values, optionally first changing the size to specified dimensions
- *.randu()* uses a uniform distribution in the [0,1] interval
- *.randn()* uses a normal/Gaussian distribution with zero mean and unit variance
- To change the RNG seed, use *arma_rng::set_seed(value)* or *arma_rng::set_seed_random()* functions
- Examples:

```
mat A(5,10);  A.randu();   // or:  mat A(5,10,fill::randu);
```

```
mat B;  B.randu(10,20);

mat C;  C.randu( size(B) );

arma_rng::set_seed_random();  // set the seed to a random value
```

- See also:
  - randu() & randn() (standalone functions)
  - .fill()
  - .imbue()
  - .ones()
  - .zeros()
  - size()
  - uniform distribution in Wikipedia
  - normal distribution in Wikipedia

.fill(value)

- Member function of *Mat*, *Col*, *Row*, *Cube*, *field*
- Sets the elements to a specified value
- The type of value must match the type of elements used by the container object (eg. for *mat* the type is *double*)
- Examples:

```
mat A(5, 6);
A.fill(123.0);
```

- **Note:** to explicitly set all elements to zero during object construction, use the following more compact form:

```
mat A(5, 6, fill::zeros);
```

- See also:
  - .imbue()
  - .ones()
  - .zeros()
  - .randu() & .randn()
  - .replace()
  - constants (pi, nan, inf, ...)

.imbue(functor **)**
.imbue(lambda_function **)** *(C++11 only)*

- Member functions of *Mat*, *Col*, *Row* and *Cube*
- Imbue (fill) with values provided by a functor or lambda function

- For matrices, filling is done column-by-column (ie. column 0 is filled, then column 1, ...)
- For cubes, filling is done slice-by-slice, with each slice treated as a matrix
- Examples:

```
// C++11 only example
// need to include <random>

std::mt19937 engine;  // Mersenne twister random number engine

std::uniform_real_distribution<double> distr(0.0, 1.0);

mat A(4,5);

A.imbue( [&]() { return distr(engine); } );
```

- See also:
  - .fill()
  - .transform()
  - element access
  - function object at Wikipedia
  - C++11 lambda functions at Wikipedia
  - lambda function at cprogramming.com

.clean(threshold **)**

- Member function of *Mat*, *Col*, *Row*, *Cube* and *SpMat*
- For objects with non-complex elements: each element with an absolute value ≤ *threshold* is replaced by zero
- For objects with complex elements: for each element, each component (real and imaginary) with an absolute value ≤ *threshold* is replaced by zero
- Can be used to sparsify a matrix, in the sense of zeroing values with small magnitudes
- **Caveat:** to explicitly convert from dense storage to sparse storage, use the SpMat class
- Examples:

```
sp_mat A;

A.sprandu(1000, 1000, 0.01);

A(12,34) =  datum::eps;
A(56,78) = -datum::eps;

A.clean(datum::eps);
```

- See also:
  - .replace()
  - .transform()
  - .is_zero()

- .zeros()
- clamp()
- datum::eps

.replace(old_value, new_value)

- Member function of *Mat*, *Col*, *Row*, *Cube* and *SpMat*
- For all elements equal to *old_value*, set them to *new_value*
- The type of *old_value* and *new_value* must match the type of elements used by the container object (eg. for *mat* the type is *double*)
- **Caveats:**
  - floating point numbers (*float* and *double*) are approximations due to their necessarily limited precision
  - for sparse matrices (*SpMat*), replacement is not done when *old_value = 0*
- Examples:

```
mat A(5,6,fill::randu);

A.diag().fill(datum::nan);

A.replace(datum::nan, 0);  // replace each NaN with 0
```

- See also:
  - .transform()
  - .for_each()
  - .clean()
  - .fill()
  - .has_nan()
  - .has_inf()
  - find()
  - clamp()
  - relational operators
  - constants (pi, nan, inf, ...)

.transform(functor **)**
.transform(lambda_function **)** *(C++11 only)*

- Member functions of *Mat*, *Col*, *Row*, *Cube* and *SpMat*
- Transform each element using a functor or lambda function
- For dense matrices, transformation is done column-by-column for all elements
- For sparse matrices, transformation is done column-by-column for non-zero elements
- For cubes, transformation is done slice-by-slice, with each slice treated as a matrix

- Examples:

```
// C++11 only example

mat A = ones<mat>(4,5);

// add 123 to every element
A.transform( [](double val) { return (val + 123.0); } );
```

- See also:
  - .for_each()
  - .replace()
  - .imbue()
  - .clean()
  - .fill()
  - element access
  - overloaded operators
  - miscellaneous element-wise functions (exp, log, pow, sqrt, square, round, ...)
  - function object at Wikipedia
  - C++11 lambda functions at Wikipedia
  - lambda function at cprogramming.com

.for_each(functor **)**
.for_each(lambda_function **)** *(C++11 only)*

- Member functions of *Mat*, *Col*, *Row*, *Cube*, *SpMat* and *field*
- For each element, pass its reference to a functor or lambda function
- For dense matrices and fields, the processing is done column-by-column for all elements
- For sparse matrices, the processing is done column-by-column for non-zero elements
- For cubes, processing is done slice-by-slice, with each slice treated as a matrix
- Examples:

```
// C++11 only examples

// add 123 to each element in a dense matrix

mat A = ones<mat>(4,5);

A.for_each( [](mat::elem_type& val) { val += 123.0; } );  // NOTE: the '&' is crucial!


// add 123 to each non-zero element in a sparse matrix

sp_mat S; S.sprandu(1000, 2000, 0.1);

S.for_each( [](sp_mat::elem_type& val) { val += 123.0; } );  // NOTE: the '&' is crucial!


// set the size of all matrices in field F
```

```
field<mat> F(2,3);

F.for_each( [](mat& X) { X.zeros(4,5); } );  // NOTE: the '&' is crucial!
```

- See also:
  - .transform()
  - .replace()
  - .each_col() & .each_row()
  - .each_slice()
  - element access
  - miscellaneous element-wise functions (exp, log, pow, sqrt, square, round, ...)
  - function object at Wikipedia
  - C++11 lambda functions at Wikipedia
  - lambda function at cprogramming.com

.set_size(n_elem) (member function of *Col*, *Row*, *field*)
.set_size(n_rows, n_cols) (member function of *Mat*, *SpMat*, *field*)
.set_size(n_rows, n_cols, n_slices) (member function of *Cube* and *field*)
.set_size(size(X)) (member function of *Mat*, *Col*, *Row*, *Cube*, *SpMat*, *field*)

- Change the size of an object, without explicitly preserving data and without initialising the elements
- If you need to initialise the elements to zero while changing the size, use .zeros() instead
- If you need to explicitly preserve data while changing the size, use .reshape() or .resize() instead; **caveat**: *.reshape()* and *.resize()* are considerably slower than *.set_size()*
- Examples:

```
mat A;  A.set_size(5,10);       // or:  mat A(5,10);

mat B;  B.set_size( size(A) );  // or:  mat B( size(A) );

vec v;  v.set_size(100);        // or:  vec v(100);
```

- See also:
  - .reset()
  - .copy_size()
  - .reshape()
  - .resize()
  - .zeros()
  - size()

.reshape(n_rows, n_cols) (member function of *Mat* and *SpMat*)
.reshape(n_rows, n_cols, n_slices) (member function of *Cube*)
.reshape(size(X)) (member function of *Mat*, *Cube*, *SpMat*)

- Recreate the object according to given size specifications, with the elements taken from the previous version of the object in a column-wise manner; the elements in the generated object are placed column-wise (ie. the first column is filled up before filling the second column)
- The layout of the elements in the recreated object will be different to the layout in the previous version of the object
- If the total number of elements in the previous version of the object is less than the specified size, the extra elements in the recreated object are set to zero
- If the total number of elements in the previous version of the object is greater than the specified size, only a subset of the elements is taken
- **Caveats:**
  - do not use *.reshape()* if you simply want to change the size without preserving data; use .set_size() instead, which is much faster
  - to grow/shrink the object while preserving the elements **as well as** the layout of the elements, use .resize() instead
  - to flatten a matrix into a vector, use vectorise() or .as_col() / .as_row() instead
- Examples:

```
mat A = randu<mat>(4,5);

A.reshape(5,4);
```

- See also:
  - .resize()
  - .set_size()
  - .copy_size()
  - .zeros()
  - .reset()
  - .as_col() / .as_row()
  - reshape() (standalone function)
  - vectorise()
  - size()

.resize(n_elem) (member function of *Col*, *Row*)
.resize(n_rows, n_cols) (member function of *Mat* and *SpMat*)
.resize(n_rows, n_cols, n_slices) (member function of *Cube*)
.resize(size(X)) (member function of *Mat*, *Col*, *Row*, *Cube*, *SpMat*)

- Recreate the object according to given size specifications, while preserving the elements as well as the layout of the elements
- Can be used for growing or shrinking an object (ie. adding/removing rows, and/or columns, and/or slices)
- **Caveat:** do not use *.resize()* if you simply want to change the size without preserving data; use .set_size() instead, which is much faster

- Examples:

```
mat A = randu<mat>(4,5);
A.resize(7,6);
```

- See also:
  - .reshape()
  - .set_size()
  - .copy_size()
  - .zeros()
  - .reset()
  - .insert_rows/cols/slices
  - .shed_rows/cols/slices
  - resize() (standalone function)
  - vectorise()
  - size()

.copy_size(A)

- Set the size to be the same as object *A*
- Object *A* must be of the same root type as the object being modified (eg. you can't set the size of a matrix by providing a cube)
- Examples:

```
mat A = randu<mat>(5,6);
mat B;
B.copy_size(A);

cout << B.n_rows << endl;
cout << B.n_cols << endl;
```

- See also:
  - .reset()
  - .set_size()
  - .reshape()
  - .resize()
  - .zeros()
  - size()

.reset()

- Reset the size to zero (the object will have no elements)
- Examples:

```
mat A = randu<mat>(5, 5);
A.reset();
```

- See also:
  - .set_size()
  - .is_empty()
  - .zeros()

---

submatrix views

- A collection of member functions of *Mat*, *Col* and *Row* classes that provide read/write access to submatrix views

- contiguous views for matrix X:

  X.**col(** col_number **)**
  X.**row(** row_number **)**

  X.**cols(** first_col**,** last_col **)**
  X.**rows(** first_row**,** last_row **)**

  X.**submat(** first_row**,** first_col**,** last_row**,** last_col **)**

  X**(span(**first_row**,** last_row**), span(**first_col**,** last_col**))**

  X**(** first_row**,** first_col**, size(**n_rows**,** n_cols**))**
  X**(** first_row**,** first_col**, size(**Y**))** *[Y is a matrix]*

  X**(span(**first_row**,** last_row**),** col_number **)**
  X**(** row_number**, span(**first_col**,** last_col**))**

  X.**head_cols(** number_of_cols **)**
  X.**head_rows(** number_of_rows **)**

  X.**tail_cols(** number_of_cols **)**
  X.**tail_rows(** number_of_rows **)**

  X.**unsafe_col(** col_number **)** *[use with caution]*

- contiguous views for vector V:

  V**(span(**first_index**,** last_index**))**
  V.**subvec(** first_index**,** last_index **)**

  V.**subvec(** first_index**, size(**W**))** *[W is a vector]*

  V.**head(** number_of_elements **)**
  V.**tail(** number_of_elements **)**

- non-contiguous views for matrix or vector X:

  X.**elem(** vector_of_indices **)**
  X**(** vector_of_indices **)**

  X.**cols(** vector_of_column_indices **)**
  X.**rows(** vector_of_row_indices **)**

  X.**submat(** vector_of_row_indices**,** vector
  X**(** vector_of_row_indices**,** vector_of_colu

- related matrix views (documented separately)

  X.diag()
  X.each_row()
  X.each_col()

- Instances of *span(start,end)* can be replaced by *span::all* to indicate the entire range

- For functions requiring one or more vector of indices, eg. *X.submat(vector_of_row_indices, vector_of_column_indices)*, each vector of indices must be of type *uvec*

- In the function *X.elem(vector_of_indices)*, elements specified in *vector_of_indices* are accessed. *X* is interpreted as one long vector, with column-by-column ordering of the elements of *X*. The *vector_of_indices* must evaluate to a vector of type uvec (eg. generated by the find() function). The aggregate set of the specified elements is treated as a column vector (ie. the output of *X.elem()* is always a column vector).

- The function *.unsafe_col()* is provided for speed reasons and should be used only if you know what you are doing. It creates a seemingly independent *Col* vector object (eg. *vec*), but uses memory from the existing matrix object. As such, the created vector is not alias safe, and does not take into account that the underlying matrix memory could be freed (eg. due to any operation involving a size change of the matrix).

- Submatrix views of sparse matrices are only useful with Armadillo 9.600 and later versions; in earlier versions they are inefficient

- Examples:

```
mat A = zeros<mat>(5,10);

A.submat( 0,1, 2,3 )      = randu<mat>(3,3);
A( span(0,2), span(1,3) ) = randu<mat>(3,3);
A( 0,1, size(3,3) )       = randu<mat>(3,3);

mat B = A.submat( 0,1, 2,3 );
mat C = A( span(0,2), span(1,3) );
mat D = A( 0,1, size(3,3) );

A.col(1)        = randu<mat>(5,1);
A(span::all, 1) = randu<mat>(5,1);

mat X = randu<mat>(5,5);

// get all elements of X that are greater than 0.5
vec q = X.elem( find(X > 0.5) );

// add 123 to all elements of X greater than 0.5
X.elem( find(X > 0.5) ) += 123.0;

// set four specific elements of X to 1
uvec indices;
indices << 2 << 3 << 6 << 8;

X.elem(indices) = ones<vec>(4);

// add 123 to the last 5 elements of vector a
vec a(10, fill::randu);
a.tail(5) += 123.0;

// add 123 to the first 3 elements of column 2 of X
X.col(2).head(3) += 123;
```

- See also:
  - diagonal views
  - .each_col() & .each_row() (vector operations repeated on each column or row)
  - .colptr()
  - .in_range()
  - find()
  - join_rows/columns/slices
  - shed_rows/columns/slices

- .insert_rows/cols/slices
- size()
- subcube views

---

subcube views and slices

- A collection of member functions of the *Cube* class that provide subcube views
- contiguous views for cube Q:

  Q.**row(** row_number **)**
  Q.**rows(** first_row**,** last_row **)**

  Q.**col(** col_number **)**
  Q.**cols(** first_col**,** last_col **)**

  Q.**slice(** slice_number **)**
  Q.**slices(** first_slice**,** last_slice **)**

  Q.**subcube(** first_row**,** first_col**,** first_slice**,** last_row**,** last_col**,** last_slice **)**

  Q**(span(**first_row**,** last_row**), span(**first_col**,** last_col**), span(**first_slice**,** last_slice**))**

  Q**(** first_row**,** first_col**,** first_slice**, size(**n_rows**,** n_cols**,** n_slices**))**
  Q**(** first_row**,** first_col**,** first_slice**, size(**R**))** *[R is a cube]*

  Q.**head_slices(** number_of_slices **)**
  Q.**tail_slices(** number_of_slices **)**

  Q.**tube(** row**,** col **)**
  Q.**tube(** first_row**,** first_col**,** last_row**,** last_col **)**
  Q.**tube(span(**first_row**,** last_row**), span(**first_col**,** last_col**))**
  Q.**tube(** first_row**,** first_col**, size(**n_rows**,** n_cols**))**

- non-contiguous views for cube Q:

  Q.**elem(** vector_of_indices **)**
  Q**(** vector_of_indices **)**

  Q.**slices(** vector_of_slice_indices **)**

- related cube views (documented separately)

  Q.each_slice()

- Instances of *span(a,b)* can be replaced by:
  - *span()* or *span::all*, to indicate the entire range
  - *span(a)*, to indicate a particular row, column or slice
- An individual slice, accessed via *.slice()*, is an instance of the *Mat* class (a reference to a matrix is provided)
- All *.tube()* forms are variants of *.subcube()*, using *first_slice = 0* and *last_slice = Q.n_slices-1*
- The *.tube(row,col)* form uses *row = first_row = last_row*, and *col = first_col = last_col*
- In the function *Q.elem(vector_of_indices)*, elements specified in *vector_of_indices* are accessed. *Q* is interpreted as one long vector, with slice-by-slice and column-by-column ordering of the elements of *Q*. The *vector_of_indices* must evaluate to a vector of type uvec (eg. generated by the find() function). The aggregate set of the specified elements is treated as a column vector (ie. the output of *Q.elem()* is always a column

vector).

- In the function *Q.slices(vector_of_slice_indices)*, slices specified in *vector_of_slice_indices* are accessed. The *vector_of_slice_indices* must evaluate to a vector of type uvec.

- Examples:

```
cube A = randu<cube>(2,3,4);

mat  B = A.slice(1); // each slice is a matrix

A.slice(0) = randu<mat>(2,3);
A.slice(0)(1,2) = 99.0;

A.subcube(0,0,1,  1,1,2)             = randu<cube>(2,2,2);
A( span(0,1), span(0,1), span(1,2) ) = randu<cube>(2,2,2);
A( 0,0,1, size(2,2,2) )              = randu<cube>(2,2,2);

// add 123 to all elements of A greater than 0.5
A.elem( find(A > 0.5) ) += 123.0;

cube C = A.head_slices(2);  // get first two slices

A.head_slices(2) += 123.0;
```

- See also:
  - .in_range()
  - .each_slice()
  - join_slices()
  - shed_slices()
  - insert_slices()
  - size()
  - submatrix views

---

subfield views

- A collection of member functions of the *field* class that provide subfield views

- For a 2D field *F*, the subfields are accessed as:

  F.**row(** row_number **)**
  F.**col(** col_number **)**

  F.**rows(** first_row**,** last_row **)**
  F.**cols(** first_col**,** last_col **)**

  F.**subfield(** first_row**,** first_col**,** last_row**,** last_col **)**

  F**(span(**first_row**,** last_row**), span(**first_col**,** last_col**))**

  F**(** first_row**,** first_col**, size(**G**))** *[G is a 2D field]*
  F**(** first_row**,** first_col**, size(**n_rows**,** n_cols**))**

- For a 3D field *F*, the subfields are accessed as:

  F.**slice(** slice_number **)**

  F.**slices(** first_slice**,** last_slice **)**

  F.**subfield(** first_row**,** first_col**,** first_slice**,** last_row**,** last_col**,** last_slice **)**

  F**(span(**first_row**,** last_row**), span(**first_col**,** last_col**), span(**first_slice**,** last_slice**))**

  F**(** first_row**,** first_col**,** first_slice**, size(**G**))** *[G is a 3D field]*
  F**(** first_row**,** first_col**,** first_slice**, size(**n_rows**,** n_cols**,** n_slices**))**

- Instances of *span(a,b)* can be replaced by:
  - *span()* or *span::all*, to indicate the entire range
  - *span(a)*, to indicate a particular row or column

- See also:
  - .in_range()
  - size()
  - submatrix views
  - subcube views

.diag()
.diag(k)

- Member function of *Mat* and *SpMat*

- Read/write access to a diagonal in a matrix

- The argument *k* is optional; by default the main diagonal is accessed (*k=0*)

- For *k > 0*, the *k*-th super-diagonal is accessed (top-right corner)

- For *k < 0*, the *k*-th sub-diagonal is accessed (bottom-left corner)

- The diagonal is interpreted as a column vector within expressions

- Examples:

```
mat X = randu<mat>(5,5);

vec a = X.diag();
vec b = X.diag(1);
vec c = X.diag(-2);

X.diag()  = randu<vec>(5);
X.diag() += 6;
X.diag().ones();
```

- **NOTE:** handling of sparse matrix diagonals has changed slightly between Armadillo 7.x and 8.x; to copy sparse diagonal to dense vector, use:

```
sp_mat S = sprandu<sp_mat>(10,10,0.1);

vec v(S.diag());  // copy sparse diagonal to dense vector
```

- See also:
  - .eye()
  - diagvec()
  - diagmat()
  - submatrix views
  - .each_col() & .each_row()
  - trace()

.each_col()
.each_row()

.each_col(vector_of_indices **)**
.each_row(vector_of_indices **)**

.each_col(lambda_function **)** *(C++11 only)*
.each_row(lambda_function **)** *(C++11 only)*

- Member functions of *Mat*
- Apply a vector operation to each column or row of a matrix
- Supported operations for *.each_col() / .each_row()* and *.each_col(vector_of_indices) / .each_row(vector_of_indices)* forms:

`+`	addition	`+=`	in-place addition
`-`	subtraction	`-=`	in-place subtraction
`%`	element-wise multiplication	`%=`	in-place element-wise multiplication
`/`	element-wise division	`/=`	in-place element-wise division
`=`	assignment (copy)		

- The argument *vector_of_indices* is optional; by default all columns or rows are used
- If the argument *vector_of_indices* is specified, it must evaluate to a vector of type *uvec*; the vector contains a list of indices of the columns or rows to be used
- If the *lambda_function* is specified, the function must accept a reference to a Col or Row object with the same element type as the underlying matrix
- Examples:

```
mat X = ones<mat>(6,5);
vec v = linspace<vec>(10,15,6);

X.each_col() += v;         // in-place addition of v to each column vector of X

mat Y = X.each_col() + v;  // generate Y by adding v to each column vector of X

// subtract v from columns 0 through to 3 in X
```

```
X.cols(0,3).each_col() -= v;


uvec indices(2);
indices(0) = 2;
indices(1) = 4;

X.each_col(indices) = v;   // copy v to columns 2 and 4 in X


X.each_col( [](vec& a){ a.print(); } );     // lambda function with non-const vector

const mat& XX = X;
XX.each_col( [](const vec& b){ b.print(); } );  // lambda function with const vector
```

- See also:
  - submatrix views
  - diagonal views
  - repmat()
  - .for_each()
  - .each_slice()

.each_slice() (form 1)
.each_slice(vector_of_indices **)** (form 2)
.each_slice(lambda_function **)** (form 3) *(C++11 only)*
.each_slice(lambda_function, use_mp **)** (form 4) *(C++11 only)*

- Member function of *Cube*
- Apply a matrix operation to each slice of a cube, with each slice treated as a matrix
- Supported operations for form 1:

`+`	addition	`+=`	in-place addition
`-`	subtraction	`-=`	in-place subtraction
`%`	element-wise multiplication	`%=`	in-place element-wise multiplication
`/`	element-wise division	`/=`	in-place element-wise division
`*`	matrix multiplication	`*=`	in-place matrix multiplication
`=`	assignment (copy)		

- For form 2:
  - the argument *vector_of_indices* contains a list of indices of the slices to be used; it must evaluate to a vector of type *uvec*
  - arithmetic operations as per form 1 are supported, except for `*` and `*=` (ie. matrix multiplication)
- For form 3:
  - apply the given *lambda_function* to each slice; the function must accept a reference to a Mat object with the same element type as the underlying cube
- For form 4:
  - apply the given *lambda_function* to each slice, as per form 3
  - the argument *use_mp* is a bool which enables the use of OpenMP for multi-threaded execution of

lambda_function on multiple slices at the same time
- the order of processing the slices is not deterministic (eg. slice 2 can be processed before slice 1)
- *lambda_function* must be thread-safe, ie. it must not write to variables outside of its scope

- Examples:

```
cube C(4,5,6, fill::randu);

mat M = repmat(linspace<vec>(1,4,4), 1, 5);

C.each_slice() += M;          // in-place addition of M to each slice of C

cube D = C.each_slice() + M;  // generate D by adding M to each slice of C


uvec indices(2);
indices(0) = 2;
indices(1) = 4;

C.each_slice(indices) = M;    // copy M to slices 2 and 4 in C


C.each_slice( [](mat& X){ X.print(); } );     // lambda function with non-const matrix

const cube& CC = C;
CC.each_slice( [](const mat& X){ X.print(); } );  // lambda function with const matrix
```

- See also:
  - subcube views
  - .for_each()
  - .each_col() & .each_row()
  - lambda function at cprogramming.com

.set_imag(X)
.set_real(X)

- Set the imaginary/real part of an object
- *X* must have the same size as the recipient object
- Examples:

```
   mat A = randu<mat>(4,5);
   mat B = randu<mat>(4,5);

cx_mat C = zeros<cx_mat>(4,5);

C.set_real(A);
C.set_imag(B);
```

- **Caveat:** to directly construct a complex matrix out of two real matrices, the following code is faster:

```
   mat A = randu<mat>(4,5);
   mat B = randu<mat>(4,5);

cx_mat C = cx_mat(A,B);
```

- See also:
  - matrix constructors
  - cube constructors
  - imag() / real()

.insert_rows(row_number, X **)** (member functions of *Mat*, *Col* and *Cube*)
.insert_rows(row_number, number_of_rows **)**
.insert_rows(row_number, number_of_rows, set_to_zero **)**

.insert_cols(col_number**,** X **)** (member functions of *Mat*, *Row* and *Cube*)
.insert_cols(col_number**,** number_of_cols **)**
.insert_cols(col_number**,** number_of_cols**,** set_to_zero **)**

.insert_slices(slice_number**,** X **)** (member functions of *Cube*)
.insert_slices(slice_number**,** number_of_slices **)**
.insert_slices(slice_number**,** number_of_slices**,** set_to_zero **)**

- Functions with the *X* argument: insert a copy of *X* at the specified row/column/slice
  - if inserting rows, *X* must have the same number of columns (and slices) as the recipient object
  - if inserting columns, *X* must have the same number of rows (and slices) as the recipient object
  - if inserting slices, *X* must have the same number of rows and columns as the recipient object (ie. all slices must have the same size)

- Functions with the *number_of_...* argument:
  - expand the object by creating new rows/columns/slices
  - by default, the new rows/columns/slices are set to zero
  - if *set_to_zero* is *false*, the memory used by the new rows/columns/slices will not be initialised

- Examples:

```
mat A = randu<mat>(5,10);
mat B = ones<mat>(5,2);

// at column 2, insert a copy of B;
// A will now have 12 columns
A.insert_cols(2, B);

// at column 1, insert 5 zeroed columns;
// B will now have 7 columns
B.insert_cols(1, 5);
```

- See also:
  - shed_rows/columns/slices
  - join_rows/columns/slices
  - .resize()
  - submatrix views
  - subcube views

.shed_row(row_number **)** (member function of *Mat*, *Col*, *SpMat*, *Cube*)
.shed_rows(first_row, last_row **)** (member function of *Mat*, *Col*, *SpMat*, *Cube*)
.shed_rows(vector_of_indices **)** (member function of *Mat*, *Col*)

.shed_col(column_number **)** (member function of *Mat*, *Row*, *SpMat*, *Cube*)
.shed_cols(first_column, last_column **)** (member function of *Mat*, *Row*, *SpMat*, *Cube*)
.shed_cols(vector_of_indices **)** (member function of *Mat*, *Row*)

.shed_slice(slice_number **)** (member functions of *Cube*)
.shed_slices(first_slice, last_slice **)**
.shed_slices(vector_of_indices **)**

- Functions with single scalar argument: remove the specified row/column/slice
- Functions with two scalar arguments: remove the specified range of rows/columns/slices
- The *vector_of_indices* must evaluate to a vector of type uvec; it contains the indices of rows/columns/slices to remove
- Examples:

```
mat A = randu<mat>(5,10);
mat B = randu<mat>(5,10);

A.shed_row(2);
A.shed_cols(2,4);

uvec indices = {4, 6, 8};
B.shed_cols(indices);
```

- See also:
  - insert_rows/columns/slices
  - join_rows/columns/slices
  - .resize()
  - submatrix views
  - subcube views
  - *shed* in thesaurus.com

.swap_rows(row1, row2)
.swap_cols(col1, col2)

- Member functions of *Mat*, *Col*, *Row* and *SpMat*
- Swap the contents of specified rows or columns
- Examples:

```
mat X = randu<mat>(5,5);
X.swap_rows(0,4);
```

- See also:

- fliplr() & flipud()
- .swap()

.swap(X)

- Member function of *Mat*, *Col*, *Row* and *Cube*
- Swap contents with object *X*
- Examples:

```
mat A = zeros<mat>(4,5);
mat B =  ones<mat>(6,7);

A.swap(B);
```

- See also:
  - .swap_rows() & .swap_cols()

.memptr()

- Member function of *Mat*, *Col*, *Row* and *Cube*
- Obtain a raw pointer to the memory used for storing elements
- The function can be used for interfacing with libraries such as FFTW
- Data for matrices is stored in a column-by-column order
- Data for cubes is stored in a slice-by-slice (matrix-by-matrix) order
- **Caveat:** the pointer becomes invalid after any operation involving a size change or aliasing
- **Caveat:** this function is not recommended for use unless you know what you are doing!
- Examples:

```
      mat A = randu<mat>(5,5);
const mat B = randu<mat>(5,5);

      double* A_mem = A.memptr();
const double* B_mem = B.memptr();
```

- See also:
  - .colptr()
  - element access
  - iterators (dense matrices)
  - iterators (cubes)
  - advanced constructors (matrices)
  - advanced constructors (cubes)

.colptr(col_number)

- Member function of *Mat*
- Obtain a raw pointer to the memory used by the specified column
- **Caveat:** the pointer becomes invalid after any operation involving a size change or aliasing
- **Caveat:** this function is not recommended for use unless you know what you are doing -- it is safer to use submatrix views instead
- Examples:

```
mat A = randu<mat>(5,5);

double* mem = A.colptr(2);
```

- See also:
  - .memptr()
  - submatrix views
  - element access
  - iterators (dense matrices)
  - advanced constructors (matrices)

---

iterators (dense matrices & vectors)

- Iterators and associated member functions of *Mat*, *Col*, *Row*
- Iterators for dense matrices and vectors traverse over all elements within the specified range
- Member functions:

.begin()	iterator referring to the first element
.end()	iterator referring to the *past-the-end* element
.begin_col(col_number)	iterator referring to the first element of the specified column
.end_col(col_number)	iterator referring to the *past-the-end* element of the specified column
.begin_row(row_number)	iterator referring to the first element of the specified row
.end_row(row_number)	iterator referring to the *past-the-end* element of the specified row

- Iterator types:

mat::iterator **vec::iterator** **rowvec::iterator**	random access iterators, for read/write access to elements (which are stored column by column)

mat::const_iterator **vec::const_iterator** **rowvec::const_iterator**	random access iterators, for read-only access to elements (which are stored column by column)
mat::col_iterator **vec::col_iterator** **rowvec::col_iterator**	random access iterators, for read/write access to the elements of specified columns
mat::const_col_iterator **vec::const_col_iterator** **rowvec::const_col_iterator**	random access iterators, for read-only access to the elements of specified columns
mat::row_iterator	bidirectional iterator, for read/write access to the elements of specified rows
mat::const_row_iterator	bidirectional iterator, for read-only access to the elements of specified rows
vec::row_iterator **rowvec::row_iterator**	random access iterators, for read/write access to the elements of specified rows
vec::const_row_iterator **rowvec::const_row_iterator**	random access iterators, for read-only access to the elements of specified rows

- Examples:

```
mat X(5, 6, fill::randu);

mat::iterator it     = X.begin();
mat::iterator it_end = X.end();

for(; it != it_end; ++it)
  {
  cout << (*it) << endl;
  }

mat::col_iterator col_it     = X.begin_col(1);  // start of column 1
mat::col_iterator col_it_end = X.end_col(3);    //   end of column 3

for(; col_it != col_it_end; ++col_it)
  {
  cout << (*col_it) << endl;
  (*col_it) = 123.0;
  }
```

- See also:
  - Mat class
  - element access
  - .for_each()
  - .memptr()
  - .colptr()
  - submatrix views
  - iterators (submatrices)

- iterators (cubes)
- iterators (sparse matrices)
- iterator at cplusplus.com

---

iterators (cubes)

- Iterators and associated member functions of *Cube*
- Iterators for cubes traverse over all elements within the specified range
- Member functions:

.begin()	iterator referring to the first element
.end()	iterator referring to the *past-the-end* element
.begin_slice(slice_number)	iterator referring to the first element of the specified slice
.end_slice(slice_number)	iterator referring to the *past-the-end* element of the specified slice

- Iterator types:

cube::iterator	random access iterator, for read/write access to elements; the elements are ordered slice by slice; the elements within each slice are ordered column by column
cube::const_iterator	random access iterator, for read-only access to elements
cube::slice_iterator	random access iterator, for read/write access to the elements of a particular slice; the elements are ordered column by column
cube::const_slice_iterator	random access iterator, for read-only access to the elements of a particular slice

- Examples:

```
cube X = randu<cube>(2,3,4);

cube::iterator it     = X.begin();
cube::iterator it_end = X.end();

for(; it != it_end; ++it)
  {
  cout << (*it) << endl;
  }

cube::slice_iterator s_it     = X.begin_slice(1);  // start of slice 1
cube::slice_iterator s_it_end = X.end_slice(2);    // end of slice 2

for(; s_it != s_it_end; ++s_it)
  {
  cout << (*s_it) << endl;
  (*s_it) = 123.0;
  }
```

- See also:
  - Cube class
  - element access
  - .for_each()
  - .memptr()
  - subcube views
  - iterators (subcubes)
  - iterators (dense matrices)
  - iterator at cplusplus.com

---

iterators (sparse matrices)

- Iterators and associated member functions of *SpMat*
- Iterators for sparse matrices traverse over non-zero elements within the specified range
- **Caveats:**
  - to modify the non-zero elements in a safer manner, use .transform() or .for_each() instead of iterators; writing a zero value into a sparse matrix through an iterator will invalidate all current iterators associated with the sparse matrix
  - row iterators for sparse matrices are only useful with Armadillo 8.500 and later versions; in earlier versions they are inefficient
- Member functions:

.begin()	iterator referring to the first element
.end()	iterator referring to the *past-the-end* element
.begin_col(col_number)	iterator referring to the first element of the specified column
.end_col(col_number)	iterator referring to the *past-the-end* element of the specified column
.begin_row(row_number)	iterator referring to the first element of the specified row
.end_row(row_number)	iterator referring to the *past-the-end* element of the specified row

- Iterator types:

sp_mat::iterator	bidirectional iterator, for read/write access to elements (which are stored column by column)
sp_mat::const_iterator	bidirectional iterator, for read-only access to elements (which are stored column by column)
sp_mat::col_iterator	bidirectional iterator, for read/write access to the elements of a specific column
sp_mat::const_col_iterator	bidirectional iterator, for read-only access to the elements of a specific column

sp_mat::row_iterator	bidirectional iterator, for read/write access to the elements of a specific row
sp_mat::const_row_iterator	bidirectional iterator, for read-only access to the elements of a specific row

- The iterators have *.row()* and *.col()* functions which return the row and column of the current element; the returned values are of type uword
- Examples:

```
sp_mat X = sprandu<sp_mat>(1000, 2000, 0.1);

sp_mat::const_iterator it     = X.begin();
sp_mat::const_iterator it_end = X.end();

for(; it != it_end; ++it)
  {
  cout << "val: " << (*it)    << endl;
  cout << "row: " << it.row() << endl;
  cout << "col: " << it.col() << endl;
  }
```

- See also:
  - SpMat class
  - element access
  - .transform()
  - .for_each()
  - .replace()
  - submatrix views
  - iterators (dense matrices)
  - iterator at cplusplus.com

---

iterators (dense submatrices & subcubes) *(C++11 only)*

- iterators for dense submatrix and subcube views, allowing range-based for loops
- **Caveat:** These iterators are intended **only** to be used with range-based for loops. Any other use is not supported. For example, the direct use of the *begin()* and *end()* functions, as well as the underlying iterators types is not supported. The implementation of submatrices and subcubes uses short-lived temporary objects that are subject to automatic deletion, and as such are error-prone to handle manually.
- Examples:

```
mat X(100, 200, fill::randu);

for( double& val : X(span(40,60), span(50,100)) )
  {
  cout << val << endl;
  val = 123.0;
  }
```

- See also:
  - submatrix views

- .for_each()
- iterators (dense matrices)
- iterators (cubes)
- range-based for (cppreference.com)

---

compatibility container functions

- Member functions to mimic the functionality of containers in the C++ standard library:

.front()	access the first element in a vector
.back()	access the last element in a vector
.clear()	causes an object to have no elements
.empty()	returns *true* if the object has no elements; returns *false* if the object has one or more elements
.size()	returns the total number of elements

- Examples:

```
mat A = randu<mat>(5,5);
cout << A.size() << endl;

A.clear();
cout << A.empty() << endl;
```

- See also:
  - iterators (dense matrices)
  - iterators (cubes)
  - iterators (sparse matrices)
  - matrix and vector attributes
  - .is_empty()
  - .reset()

.as_col()
.as_row()

- Member functions of any matrix expression
- *.as_col()*: return a flattened version of the matrix as a column vector; flattening is done by concatenating all columns
- *.as_row()*: return a flattened version of the matrix as a row vector; flattening is done by concatenating all rows
- **Caveat:** concatenating columns is faster than concatenating rows
- Examples:

```
mat X = randu<mat>(4,5);
vec v = X.as_col();
```

- See also:
  - .reshape()
  - .t() / .st()
  - as_scalar()
  - vectorise()

.t()
.st()

- Member functions of any matrix or vector expression
- For real (non-complex) matrix:
  - *.t()* provides a transposed copy of the matrix
  - *.st()* not applicable
- For complex matrix:
  - *.t()* provides a Hermitian transpose (ie. the conjugate of the elements is taken during the transpose)
  - *.st()* provides a transposed copy without taking the conjugate of the elements
- Examples:

```
mat A = randu<mat>(4,5);
mat B = A.t();
```

- See also:
  - trans()
  - fliplr() & flipud()
  - reverse()
  - .as_col() / .as_row()

.i()

- Member function of any matrix expression
- Provides an inverse of the matrix expression
- If the matrix expression is not square sized, a *std::logic_error* exception is thrown
- If the matrix expression appears to be singular, the output matrix is reset and a *std::runtime_error* exception is thrown
- **Caveats:**
  - if matrix *A* is know to be symmetric positive definite, it's faster to use inv_sympd() instead
  - to solve a system of linear equations, such as *Z = inv(X)*Y*, using solve() can be faster and/or more accurate
- Examples:

```
mat A = randu<mat>(4,4);
```

```
mat B = randu<mat>(4,1);

mat X = A.i();

mat Y = (A+A).i();
```

- See also:
  - inv()
  - rcond()
  - pinv()
  - solve()
  - spsolve()

.min()
.max()

- Return the extremum value of any matrix or cube expression
- For objects with complex numbers, absolute values are used for comparison
- Examples:

```
mat A = randu<mat>(5,5);

double max_val = A.max();
```

- See also:
  - .index_min() & .index_max()
  - min() & max() (standalone functions with extended functionality)
  - clamp()
  - running_stat
  - running_stat_vec

.index_min()
.index_max()

- Return the linear index of the extremum value of any matrix or cube expression
- For objects with complex numbers, absolute values are used for comparison
- The returned index is of type uword
- Examples:

```
mat A = randu<mat>(5,5);

uword i = A.index_max();

double max_val = A(i);
```

- See also:

- .min() & .max()
- index_min() & index_max() (standalone functions with extended functionality)
- ind2sub()
- sort_index()
- find()
- element access

.eval()

- Member function of any matrix or vector expression
- Explicitly forces the evaluation of a delayed expression and outputs a matrix
- This function should be used sparingly and only in cases where it is absolutely necessary; indiscriminate use can cause performance degradations
- Examples:

```
cx_mat A( randu<mat>(4,4), randu<mat>(4,4) );

real(A).eval().save("A_real.dat", raw_ascii);
imag(A).eval().save("A_imag.dat", raw_ascii);
```

- See also:
  - Mat class

.in_range(i **)**	(member of *Mat*, *Col*, *Row*, *Cube*, *SpMat*, *field*)
.in_range(span(start**,** end**))**	(member of *Mat*, *Col*, *Row*, *Cube*, *SpMat*, *field*)
.in_range(row**,** col **)**	(member of *Mat*, *Col*, *Row*, *SpMat*, *field*)
.in_range(span(start_row**,** end_row**), span(**start_col**,** end_col**))**	(member of *Mat*, *Col*, *Row*, *SpMat*, *field*)
.in_range(row**,** col**,** slice **)**	(member of *Cube* and *field*)
.in_range(span(start_row**,** end_row**), span(**start_col**,** end_col**), span(**start_slice**,** end_slice**))**	(member of *Cube* and *field*)
.in_range(first_row**,** first_col**, size(**X**))** *(X is a matrix or field)*	(member of *Mat*, *Col*, *Row*, *SpMat*, *field*)
.in_range(first_row**,** first_col**, size(**n_rows**,** n_cols**))**	(member of *Mat*, *Col*, *Row*, *SpMat*, *field*)
.in_range(first_row**,** first_col**,** first_slice**, size(**Q**))** *(Q is a cube or field)*	(member of *Cube* and *field*)
.in_range(first_row**,** first_col**,** first_slice**, size(**n_rows**,** n_cols n_slices**))**	(member of *Cube* and *field*)

- Returns *true* if the given location or span is currently valid

- Returns *false* if the object is empty, the location is out of bounds, or the span is out of bounds
- Instances of *span(a,b)* can be replaced by:
  - *span()* or *span::all*, to indicate the entire range
  - *span(a)*, to indicate a particular row, column or slice
- Examples:

```
mat A = randu<mat>(4,5);

cout << A.in_range(0,0) << endl;  // true
cout << A.in_range(3,4) << endl;  // true
cout << A.in_range(4,5) << endl;  // false
```

- See also:
  - element access
  - submatrix views
  - subcube views
  - subfield views
  - .set_size()

.is_empty()

- Returns *true* if the object has no elements
- Returns *false* if the object has one or more elements
- Examples:

```
mat A = randu<mat>(5,5);
cout << A.is_empty() << endl;

A.reset();
cout << A.is_empty() << endl;
```

- See also:
  - .is_square()
  - .is_vec()
  - .is_finite()
  - .reset()

.is_vec()
.is_colvec()
.is_rowvec()

- Member functions of *Mat* and *SpMat*
- .is_vec():
  - returns *true* if the matrix can be interpreted as a vector (either column or row vector)
  - returns *false* if the matrix does not have exactly one column or one row

- .is_colvec():
  - returns *true* if the matrix can be interpreted as a column vector
  - returns *false* if the matrix does not have exactly one column
- .is_rowvec():
  - returns *true* if the matrix can be interpreted as a row vector
  - returns *false* if the matrix does not have exactly one row
- **Caveat:** do not assume that the vector has elements if these functions return *true*; it is possible to have an empty vector (eg. 0x1)
- Examples:

```
mat A = randu<mat>(1,5);
mat B = randu<mat>(5,1);
mat C = randu<mat>(5,5);

cout << A.is_vec() << endl;
cout << B.is_vec() << endl;
cout << C.is_vec() << endl;
```

- See also:
  - .is_empty()
  - .is_square()
  - .is_finite()

.is_sorted()
.is_sorted(sort_direction)
.is_sorted(sort_direction, dim)

- Member function of *Mat*, *Row* and *Col*
- If the object is a vector, return a *bool* indicating whether the elements are sorted
- If the object is a matrix, return a *bool* indicating whether the elements are sorted in each column (*dim=0*), or each row (*dim=1*)
- The *sort_direction* argument is optional; *sort_direction* is one of:

`"ascend"`	↦ elements are ascending; consecutive elements can be equal; this is the **default operation**
`"descend"`	↦ elements are descending; consecutive elements can be equal
`"strictascend"`	↦ elements are strictly ascending; consecutive elements cannot be equal
`"strictdescend"`	↦ elements are strictly descending; consecutive elements cannot be equal

- The *dim* argument is optional; by default *dim=0* is used
- For matrices and vectors with complex numbers, order is checked via absolute values
- Examples:

```
vec a = randu<vec>(10);
```

```
vec b = sort(a);

bool check1 = a.is_sorted();
bool check2 = b.is_sorted();


mat A = randu<mat>(10,10);

// check whether each column is sorted in descending manner
cout << A.is_sorted("descend") << endl;

// check whether each row is sorted in ascending manner
cout << A.is_sorted("ascend", 1) << endl;
```

- See also:
  - sort()
  - sort_index()

.is_trimatu()
.is_trimatl()

- Member functions of *Mat* and *SpMat*

- *.is_trimatu()*:
  - return *true* if the matrix is upper triangular, ie. the matrix is square sized and all elements below the main diagonal are zero; return *false* otherwise
  - **caveat:** if this function returns *true*, do not assume that the matrix contains non-zero elements on or above the main diagonal

- *.is_trimatl()*:
  - return *true* if the matrix is lower triangular, ie. the matrix is square sized and all elements above the main diagonal are zero; return *false* otherwise
  - **caveat:** if this function returns *true*, do not assume that the matrix contains non-zero elements on or below the main diagonal

- Examples:

```
mat A = randu<mat>(5,5);
mat B = trimatl(A);

cout << A.is_trimatu() << endl;
cout << B.is_trimatl() << endl;
```

- See also:
  - trimatu() / trimatl()
  - .is_symmetric()
  - .is_diagmat()
  - Triangular matrix in MathWorld
  - Triangular matrix in Wikipedia

.is_diagmat()

- Member function of *Mat* and *SpMat*
- Return *true* if the matrix is diagonal, ie. all elements outside of the main diagonal are zero
- Return *false* otherwise
- **Caveat:** if this function returns *true*, do not assume that the matrix contains non-zero elements on the main diagonal
- Examples:

```
mat A = randu<mat>(5,5);
mat B = diagmat(A);

cout << A.is_diagmat() << endl;
cout << B.is_diagmat() << endl;
```

- See also:
  - diagmat()
  - .is_trimatu() / .is_trimatl()
  - .is_symmetric()
  - Diagonal matrix in MathWorld
  - Diagonal matrix in Wikipedia

.is_square()

- Member function of *Mat* and *SpMat*
- Returns *true* if the matrix is square, ie. number of rows is equal to the number of columns
- Returns *false* if the matrix is not square
- Examples:

```
mat A = randu<mat>(5,5);
mat B = randu<mat>(6,7);

cout << A.is_square() << endl;
cout << B.is_square() << endl;
```

- See also:
  - .is_symmetric()
  - .is_hermitian()
  - .is_empty()
  - .is_vec()
  - .is_finite()

.is_symmetric()
.is_symmetric(tol)

- Member function of *Mat* and *SpMat*
- Returns *true* if the matrix is symmetric
- Returns *false* if the matrix is not symmetric
- The *tol* argument is optional; if *tol* is specified, the given matrix *X* is considered symmetric if *norm(X - X.st(), "inf") / norm (X, "inf") ≤ tol*
- Examples:

```
mat A = randu<mat>(5,5);
mat B = A.t() * A;

cout << A.is_symmetric() << endl;
cout << B.is_symmetric() << endl;
```

- See also:
  - .is_hermitian()
  - .is_sympd()
  - .is_square()
  - .is_trimatu() / .is_trimatl()
  - Symmetric matrix in Wikipedia
  - Symmetric matrix in MathWorld

.is_hermitian()
.is_hermitian(tol)

- Member function of *Mat* and *SpMat*
- Returns *true* if the matrix is hermitian (self-adjoint)
- Returns *false* if the matrix is not hermitian
- The *tol* argument is optional; if *tol* is specified, the given matrix *X* is considered hermitian if *norm(X - X.t(), "inf") / norm (X, "inf") ≤ tol*
- Examples:

```
cx_mat A = randu<cx_mat>(5,5);
cx_mat B = A.t() * A;

cout << A.is_hermitian() << endl;
cout << B.is_hermitian() << endl;
```

- See also:
  - .is_symmetric()
  - .is_sympd()
  - .is_square()
  - Hermitian matrix in Wikipedia
  - Hermitian matrix in MathWorld

.is_sympd()
.is_sympd(tol)

- Member function of *Mat* and any dense matrix expression
- Returns *true* if the matrix is symmetric/hermitian positive definite within the tolerance given by *tol*
- Returns *false* otherwise
- The *tol* argument is optional; if *tol* is not specified, by default *tol = 100 * datum::eps * norm(X, "fro")*
- Examples:

```
mat A(5,5,fill::randu);

mat B = A.t() * A;

cout << A.is_sympd() << endl;
cout << B.is_sympd() << endl;
```

- See also:
  - .is_symmetric()
  - .is_hermitian()
  - datum::eps

.is_zero()
.is_zero(tolerance **)**

- For objects with non-complex elements: return *true* if each element has an absolute value ≤ *tolerance*; return *false* otherwise
- For objects with complex elements: return *true* if for each element, each component (real and imaginary) has an absolute value ≤ *tolerance*; return *false* otherwise
- The argument *tolerance* is optional; by default *tolerance = 0*
- Examples:

```
mat A = zeros<mat>(5,5);

A(0,0) = datum::eps;

cout << A.is_zero()           << endl;
cout << A.is_zero(datum::eps) << endl;
```

- See also:
  - .clean()
  - all()
  - datum::eps
  - approx_equal()

.is_finite()

- Member function of *Mat*, *Col*, *Row*, *Cube*, *SpMat*
- Returns *true* if all elements of the object are finite
- Returns *false* if at least one of the elements of the object is non-finite (±infinity or NaN)
- Examples:

```
mat A = randu<mat>(5,5);
mat B = randu<mat>(5,5);

B(1,1) = datum::inf;

cout << A.is_finite() << endl;
cout << B.is_finite() << endl;
```

- See also:
  - .has_inf()
  - .has_nan()
  - find_finite()
  - find_nonfinite()
  - constants (pi, nan, inf, ...)

.has_inf()

- Member function of *Mat*, *Col*, *Row*, *Cube*, *SpMat*
- Returns *true* if at least one of the elements of the object is ±infinity
- Returns *false* otherwise
- Examples:

```
mat A = randu<mat>(5,5);
mat B = randu<mat>(5,5);

B(1,1) = datum::inf;

cout << A.has_inf() << endl;
cout << B.has_inf() << endl;
```

- See also:
  - .has_nan()
  - .replace()
  - .is_finite()
  - find_nonfinite()
  - constants (pi, nan, inf, ...)

.has_nan()

- Member function of *Mat*, *Col*, *Row*, *Cube*, *SpMat*
- Returns *true* if at least one of the elements of the object is NaN (not-a-number)
- Returns *false* otherwise
- **Caveat:** `NaN` is not equal to anything, even itself
- Examples:

```
mat A = randu<mat>(5,5);
mat B = randu<mat>(5,5);

B(1,1) = datum::nan;

cout << A.has_nan() << endl;
cout << B.has_nan() << endl;
```

- See also:
  - .has_inf()
  - .replace()
  - .is_finite()
  - find_nonfinite()
  - constants (pi, nan, inf, ...)

.print()
.print(header)

.print(stream)
.print(stream, header)

- Member functions of *Mat*, *Col*, *Row*, *SpMat*, *Cube* and *field*
- Print the contents of an object to the *std::cout* stream (default), or a user specified stream, with an optional header string
- Objects can also be printed using the << stream operator
- Elements of a field can only be printed if there is an associated *operator<<* function defined
- Examples:

```
mat A = randu<mat>(5,5);
mat B = randu<mat>(6,6);

A.print();

// print a transposed version of A
A.t().print();

// "B:" is the optional header line
```

```
B.print("B:");

cout << A << endl;

cout << "B:" << endl;
cout << B << endl;
```

- See also:
  - .raw_print()
  - saving & loading matrices
  - initialising elements
  - logging of errors and warnings

.raw_print()
.raw_print(header)

.raw_print(stream)
.raw_print(stream, header)

- Member functions of *Mat*, *Col*, *Row*, *SpMat* and *Cube*

- Similar to the .print() member function, with the difference that no formatting of the output is done; the stream's parameters such as precision, cell width, etc. can be set manually

- If the cell width is set to zero, a space is printed between the elements

- Examples:

```
mat A = randu<mat>(5,5);

cout.precision(11);
cout.setf(ios::fixed);

A.raw_print(cout, "A:");
```

- See also:
  - .print()
  - std::ios_base::fmtflags (cppreference.com)
  - std::ios_base::fmtflags (cplusplus.com)

---

saving/loading matrices & cubes

.save(filename **)**	**.load(** filename **)**
.save(filename**,** file_type **)**	**.load(** filename**,** file_type **)**
.save(stream **)**	**.load(** stream **)**
.save(stream**,** file_type **)**	**.load(** stream**,** file_type **)**
.save(hdf5_name(filename**,** dataset**))**	**.load(hdf5_name(**filename**,** dataset**))**
.save(hdf5_name(filename**,** dataset**,** settings**))**	**.load(hdf5_name(**filename**,** dataset**,** settings**))**
.save(csv_name(filename**,** header**))**	**.load(csv_name(**filename**,** header**))**
.save(csv_name(filename**,** header**,** settings**))**	**.load(csv_name(**filename**,** header**,** settings**))**

- Member functions of *Mat*, *Col*, *Row*, *Cube* and *SpMat*
- Store/retrieve data in a file or stream (**caveat:** the stream must be opened in binary mode)
- On success, *.save()* and *.load()* return a *bool* set to *true*
- On failure, *.save()* and *.load()* return a *bool* set to *false*; additionally, *.load()* resets the object so that it has no elements
- *file_type* can be one of the following:

auto_detect Used only by *.load()* only: attempt to automatically detect the file type as one of the formats described below;
[default operation for *.load()*]

arma_binary Numerical data stored in machine dependent binary format, with a simple header to speed up loading. The header indicates the type and size of matrix/cube.
[default operation for *.save()*]

arma_ascii Numerical data stored in human readable text format, with a simple header to speed up loading. The header indicates the type and size of matrix/cube.

raw_binary Numerical data stored in machine dependent raw binary format, without a header. Matrices are loaded to have one column, while cubes are loaded to have one slice with one column. The .reshape() function can be used to alter the size of the loaded matrix/cube without losing data.

raw_ascii Numerical data stored in raw ASCII format, without a header. The numbers are separated by whitespace. The number of columns must be the same in each row. Cubes are loaded as one slice. Data which was saved in Matlab/Octave using the *-ascii* option can be read in Armadillo, except for complex numbers. Complex numbers are stored in standard C++ notation, which is a tuple surrounded by brackets: eg. (1.23,4.56) indicates 1.24 + 4.56i.

csv_ascii Numerical data stored in comma separated value (CSV) text format, **without** a header. To save/load **with** a header, use the **csv_name(**filename,header**)** specification instead (more details below). Applicable to *Mat* and *SpMat*.

coord_ascii Numerical data stored as a text file in coordinate list format, without a header. Only non-zero values are stored. Applicable only to sparse matrices (SpMat).
For real matrices, each line contains information in the following format: `row column value`
For complex matrices, each line contains information in the following format: `row column real_value imag_value`
The rows and columns start at zero.

pgm_binary Image data stored in Portable Gray Map (PGM) format. Applicable to *Mat* only. Saving *int*, *float* or *double* matrices is a lossy operation, as each element is copied and converted to an 8 bit representation. As such the matrix should have values in the [0,255] interval, otherwise the resulting image may not display correctly.

ppm_binary Image data stored in Portable Pixel Map (PPM) format. Applicable to *Cube* only. Saving *int*, *float* or *double* matrices is a lossy operation, as each element is copied and converted to an 8 bit representation. As such the cube/field should have values in the [0,255] interval, otherwise the resulting image may not display correctly.

hdf5_binary Numerical data stored in portable HDF5 binary format.
- for saving, the default dataset name within the HDF5 file is "dataset"
- for loading, the order of operations is: (1) try loading a dataset named "dataset", (2) try loading a dataset named "value", (3) try loading the first available dataset
- to explicitly control the dataset name, specify it via the **hdf5_name()** argument (more details below)

- By providing either **hdf5_name(**filename**,** dataset**)** or **hdf5_name(**filename**,** dataset**,** settings**)**, the *file_type* type is assumed to be *hdf5_binary*
  - the *dataset* argument specifies an HDF5 dataset name (eg. "my_dataset") that can include a full path (eg. "/group_name/my_dataset"); if a blank dataset name is specified (ie. ""), it's assumed to be "dataset"
  - the *settings* argument is optional; it is one of the following, or a combination thereof:

`hdf5_opts::trans`	save/load the data with columns transposed to rows (and vice versa)
`hdf5_opts::append`	instead of overwriting the file, append the specified dataset to the file; the specified dataset must not already exist in the file
`hdf5_opts::replace`	instead of overwriting the file, replace the specified dataset in the file **caveat:** HDF5 v1.8 may not automatically reclaim deleted space; use h5repack to clean HDF5 files

  the above settings can be combined using the `+` operator; for example:
  `hdf5_opts::trans + hdf5_opts::append`

- **Caveat**: for saving/loading HDF5 files, support for HDF5 must be enabled within Armadillo's configuration; the *hdf5.h* header file must be available on your system and you will need to link with the HDF5 library (eg. `-lhdf5`)

- By providing either **csv_name(**filename**,** header**)** or **csv_name(**filename**,** header**,** settings**)**, the file is assumed to have data in comma separated value (CSV) text format
  - the *header* argument specifies the object which stores the separate elements of the header line; it must have the type `field<std::string>`
  - the optional *settings* argument is one of the following, or a combination thereof:

`csv_opts::trans`	save/load the data with columns transposed to rows (and vice versa)
`csv_opts::no_header`	assume there is no header line; the *header* argument is not referenced

the above settings can be combined using the `+` operator; for example:
`csv_opts::trans + csv_opts::no_header`

- Examples:

```
mat A = randu<mat>(5,6);

// default save format is arma_binary
A.save("A.bin");

// save in raw_ascii format
A.save("A.txt", raw_ascii);

// save in CSV format without a header
A.save("A.csv", csv_ascii);

// save in CSV format with a header
field<std::string> header(A.n_cols);
header(0) = "foo";
header(1) = "bar";  // etc
A.save( csv_name("A.csv", header) );

// save in HDF5 format with internal dataset named as "my_data"
A.save(hdf5_name("A.h5", "my_data"));

// automatically detect format type while loading
mat B;
B.load("A.bin");

// force loading in arma_ascii format
mat C;
C.load("A.txt", arma_ascii);


// example of testing for success
mat D;
bool ok = D.load("A.bin");

if(ok == false)
  {
  cout << "problem with loading" << endl;
  }
```

- See also:
  - HDF in Wikipedia
  - CSV in Wikipedia
  - saving/loading fields

---

saving/loading fields

.save(name) **.load(name)**
.save(name, file_type) **.load(name, file_type)**

.save(stream) **.load(stream)**
.save(stream, file_type) **.load(stream, file_type)**

- Store/retrieve data in a file or stream (**caveat:** the stream must be opened in binary mode)
- On success, *.save()* and *.load()* return a *bool* set to *true*
- On failure, *.save()* and *.load()* return a *bool* set to *false*; additionally, *.load()* resets the object so that it has no elements
- Fields with objects of type *std::string* are saved and loaded as raw text files. The text files do not have a header. Each string is separated by a whitespace. *load()* will only accept text files that have the same number of strings on each line. The strings can have variable lengths.
- Other than storing string fields as text files, the following file formats are supported:

  auto_detect
  - *.load()*: attempt to automatically detect the field format type as one of the formats described below; this is the default operation

  arma_binary
  - objects are stored in machine dependent binary format
  - default type for fields of type *Mat*, *Col*, *Row* or *Cube*
  - only applicable to fields of type *Mat*, *Col*, *Row* or *Cube*

  ppm_binary
  - image data stored in Portable Pixmap Map (PPM) format
  - only applicable to fields of type *Mat*, *Col* or *Row*
  - *.load()*: loads the specified image and stores the red, green and blue components as three separate matrices; the resulting field is comprised of the three matrices, with the red, green and blue components in the first, second and third matrix, respectively
  - *.save()*: saves a field with exactly three matrices of equal size as an image; it is assumed that the red, green and blue components are stored in the first, second and third matrix, respectively; saving *int*, *float* or *double* matrices is a lossy operation, as each matrix element is copied and converted to an 8 bit representation

- See also:
  - saving/loading matrices and cubes

Generated Vectors/Matrices/Cubes

---

linspace(start, end)
linspace(start, end, N)

- Generate a vector with *N* elements; the values of the elements are linearly spaced from *start* to (and including) *end*
- The argument *N* is optional; by default *N = 100*
- Usage:
  - vec v = linspace(start, end, N)
  - *vector_type* v = linspace<*vector_type*>(start, end, N)
- **Caveat:** for *N = 1*, the generated vector will have a single element equal to *end*
- Examples:

```
   vec a = linspace(0, 5, 6);

rowvec b = linspace<rowvec>(5, 0, 6);
```

- See also:
  - regspace()
  - logspace()
  - randperm()
  - ones()
  - interp1()

---

logspace(A, B)
logspace(A, B, N)

- Generate a vector with *N* elements; the values of the elements are logarithmically spaced from 10^A to (and including) 10^B
- The argument *N* is optional; by default *N = 50*
- Usage:
  - vec v = logspace(A, B, N)
  - *vector_type* v = logspace<*vector_type*>(A, B, N)
- Examples:

```
   vec a = logspace(0, 5, 6);

rowvec b = logspace<rowvec>(5, 0, 6);
```

- See also:
  - linspace()
  - regspace()

---

regspace(start, end)
regspace(start, delta, end)

- Generate a vector with regularly spaced elements:
  [*(start + 0*delta)*, *(start + 1*delta)*, *(start + 2*delta)*, ⋯, *(start + M*delta)*]
  where *M = floor((end-start)/delta)*, so that *(start + M*delta) ≤ end*

- Similar in operation to the Matlab/Octave colon operator, ie. *start:end* and *start:delta:end*

- If *delta* is not specified:
  - *delta* = +1, if *start ≤ end*
  - *delta* = −1, if *start > end* (**caveat:** this is different to Matlab/Octave)

- An empty vector is generated when one of the following conditions is true:
  - *start < end*, and *delta < 0*
  - *start > end*, and *delta > 0*
  - *delta = 0*

- Usage:
  - vec v = regspace(start, end)
  - vec v = regspace(start, delta, end)
  - *vector_type* v = regspace<*vector_type*>(start, end)
  - *vector_type* v = regspace<*vector_type*>(start, delta, end)

- Examples:

```
 vec a = regspace(0,  9);             // 0,  1, ...,   9

uvec b = regspace<uvec>(2,  2,  10);  // 2,  4, ...,  10

ivec c = regspace<ivec>(0, -1, -10);  // 0, -1, ..., -10
```

- **Caveat:** do not use *regspace()* to specify ranges for contiguous submatrix views; use span() instead

- See also:
  - linspace()
  - logspace()

---

randperm(N)
randperm(N, M)

- Generate a vector with a random permutation of integers from *0* to *N-1*

- The optional argument *M* indicates the number of elements to return, sampled without replacement from *0* to *N-1*

- Examples:

```
uvec X = randperm(10);
uvec Y = randperm(10,2);
```

- See also:
  - randi()
  - linspace()

- regspace()

---

eye(n_rows, n_cols)
eye(size(X))

- Generate a matrix with the elements along the main diagonal set to one and off-diagonal elements set to zero
- An identity matrix is generated when *n_rows* = *n_cols*
- Usage:
  - mat X = eye(n_rows, n_cols)
  - *matrix_type* X = eye<*matrix_type*>(n_rows, n_cols)
  - *matrix_type* Y = eye<*matrix_type*>(size(X))
- Examples:

```
mat A = eye(5,5);  // or:  mat A(5,5,fill::eye);

fmat B = 123.0 * eye<fmat>(5,5);

cx_mat C = eye<cx_mat>( size(B) );
```

- See also:
  - .eye() (member function of Mat)
  - .diag()
  - ones()
  - diagmat()
  - diagvec()
  - speye()
  - size()

---

ones(n_elem)
ones(n_rows, n_cols)
ones(n_rows, n_cols, n_slices)
ones(size(X))

- Generate a vector, matrix or cube with all elements set to one
- Usage:
  - *vector_type* v = ones<*vector_type*>(n_elem)
  - *matrix_type* X = ones<*matrix_type*>(n_rows, n_cols)
  - *matrix_type* Y = ones<*matrix_type*>(size(X))
  - *cube_type* Q = ones<*cube_type*>(n_rows, n_cols, n_slices)
  - *cube_type* R = ones<*cube_type*>(size(Q))
- Examples:

```
vec  v = ones<vec>(10);
uvec u = ones<uvec>(11);
mat  A = ones<mat>(5,6);
```

```
cube Q = ones<cube>(5,6,7);

mat  B = 123.0 * ones<mat>(5,6);
```

- See also:
  - .ones() (member function of *Mat*, *Col*, *Row* and *Cube*)
  - eye()
  - linspace()
  - regspace()
  - randperm()
  - zeros()
  - randu() & randn()
  - spones()
  - size()

---

zeros(n_elem)
zeros(n_rows, n_cols)
zeros(n_rows, n_cols, n_slices)
zeros(size(X))

- Generate a vector, matrix or cube with the elements set to zero

- Usage:
  - *vector_type* v = zeros<*vector_type*>(n_elem)
  - *matrix_type* X = zeros<*matrix_type*>(n_rows, n_cols)
  - *matrix_type* Y = zeros<*matrix_type*>(size(X))
  - *cube_type* Q = zeros<*cube_type*>(n_rows, n_cols, n_slices)
  - *cube_type* R = zeros<*cube_type*>(size(Q))

- Examples:

```
vec  v = zeros<vec>(10);
uvec u = zeros<uvec>(11);
mat  A = zeros<mat>(5,6);
cube Q = zeros<cube>(5,6,7);
```

- See also:
  - .zeros() (member function of *Mat*, *Col*, *Row*, *SpMat* and *Cube*)
  - .ones() (member function of *Mat*, *Col*, *Row* and *Cube*)
  - ones()
  - randu() & randn()
  - size()

---

randu( )
randu(n_elem)
randu(n_rows, n_cols)
randu(n_rows, n_cols, n_slices)
randu(size(X))

randn( )
randn(n_elem)
randn(n_rows, n_cols)
randn(n_rows, n_cols, n_slices)
randn(size(X))

- Generate a scalar, vector, matrix or cube with the elements set to random **floating point** values
- *randu()* uses a uniform distribution in the [0,1] interval
- *randn()* uses a normal/Gaussian distribution with zero mean and unit variance
- Usage:
  - *scalar_type* s = randu<*scalar_type*>( ), where *scalar_type* ∈ { float, double, cx_float, cx_double }
  - *vector_type* v = randu<*vector_type*>(n_elem)
  - *matrix_type* X = randu<*matrix_type*>(n_rows, n_cols)
  - *matrix_type* Y = randu<*matrix_type*>(size(X))
  - *cube_type* Q = randu<*cube_type*>(n_rows, n_cols, n_slices)
  - *cube_type* R = randu<*cube_type*>(size(Q))
- To change the RNG seed, use *arma_rng::set_seed(value)* or *arma_rng::set_seed_random()* functions
- **Caveat:** to generate a matrix with random integer values instead of floating point values, use randi() instead
- Examples:

```
vec  v = randu<vec>(5);
mat  A = randu<mat>(5,6);
cube Q = randu<cube>(5,6,7);

arma_rng::set_seed_random();  // set the seed to a random value
```

- See also:
  - .randu() & .randn() (member functions)
  - randi()
  - randg()
  - mvnrnd()
  - normpdf()
  - .imbue()
  - ones()
  - zeros()
  - shuffle()
  - sprandu() & sprandn()
  - size()
  - uniform distribution in Wikipedia
  - normal distribution in Wikipedia

---

randg( )
randg(distr_param(a,b))

randg(n_elem)
randg(n_elem, distr_param(a,b))

randg(n_rows, n_cols)
randg(n_rows, n_cols, distr_param(a,b))

randg(n_rows, n_cols, n_slices)
randg(n_rows, n_cols, n_slices, distr_param(a,b))

randg(size(X))
randg(size(X), distr_param(a,b))

- Generate a scalar, vector, matrix or cube with the elements set to random values from a gamma distribution:

$$p(x|a,b) = \frac{x^{a-1}\ exp(-x/b)}{b^{a}\ \Gamma(a)}$$

  where *a* is the shape parameter and *b* is the scale parameter, with constraints $a > 0$ and $b > 0$

- The default distribution parameters are *a=1* and *b=1*

- Usage:
  - *scalar_type* s = randg<*scalar_type*>( ), where *scalar_type* is either *float* or *double*
  - *scalar_type* s = randg<*scalar_type*>(distr_param(a,b)), where *scalar_type* is either *float* or *double*

  - *vector_type* v = randg<*vector_type*>(n_elem)
  - *vector_type* v = randg<*vector_type*>(n_elem, distr_param(a,b))

  - *matrix_type* X = randg<*matrix_type*>(n_rows, n_cols)
  - *matrix_type* X = randg<*matrix_type*>(n_rows, n_cols, distr_param(a,b))
  - *matrix_type* Y = randg<*matrix_type*>(size(X))
  - *matrix_type* Y = randg<*matrix_type*>(size(X), distr_param(a,b))

  - *cube_type* Q = randg<*cube_type*>(n_rows, n_cols, n_slices)
  - *cube_type* Q = randg<*cube_type*>(n_rows, n_cols, n_slices, distr_param(a,b))
  - *cube_type* R = randg<*cube_type*>(size(Q))
  - *cube_type* R = randg<*cube_type*>(size(Q), distr_param(a,b))

- To change the RNG seed, use *arma_rng::set_seed(value)* or *arma_rng::set_seed_random()* functions

- Examples:

```
vec v = randg<vec>(100, distr_param(2,1));

mat X = randg<mat>(10, 10, distr_param(2,1));
```

- See also:
  - randu() & randn()
  - randi()
  - chi2rnd()
  - .imbue()
  - size()
  - gamma distribution in Wikipedia

randi( )
randi(distr_param(a,b))

randi(n_elem)
randi(n_elem, distr_param(a,b))

randi(n_rows, n_cols)
randi(n_rows, n_cols, distr_param(a,b))

randi(n_rows, n_cols, n_slices)
randi(n_rows, n_cols, n_slices, distr_param(a,b))

randi(size(X))
randi(size(X), distr_param(a,b))

- Generate a scalar, vector, matrix or cube with the elements set to random **integer** values in the [a,b] interval
- The default distribution parameters are *a=0* and *b=maximum_int*
- Usage:
  - *scalar_type* v = randi<*scalar_type*>( )
  - *scalar_type* v = randi<*scalar_type*>(distr_param(a,b))
  - *vector_type* v = randi<*vector_type*>(n_elem)
  - *vector_type* v = randi<*vector_type*>(n_elem, distr_param(a,b))
  - *matrix_type* X = randi<*matrix_type*>(n_rows, n_cols)
  - *matrix_type* X = randi<*matrix_type*>(n_rows, n_cols, distr_param(a,b))
  - *matrix_type* Y = randi<*matrix_type*>(size(X))
  - *matrix_type* Y = randi<*matrix_type*>(size(X), distr_param(a,b))
  - *cube_type* Q = randi<*cube_type*>(n_rows, n_cols, n_slices)
  - *cube_type* Q = randi<*cube_type*>(n_rows, n_cols, n_slices, distr_param(a,b))
  - *cube_type* R = randi<*cube_type*>(size(Q))
  - *cube_type* R = randi<*cube_type*>(size(Q), distr_param(a,b))
- To change the RNG seed, use *arma_rng::set_seed(value)* or *arma_rng::set_seed_random()* functions
- **Caveat:** to generate a continuous distribution with floating point values (ie. *float* or *double*), use randu() instead
- Examples:

```
imat A = randi<imat>(5, 6);

imat A = randi<imat>(6, 7, distr_param(-10, +20));

arma_rng::set_seed_random();  // set the seed to a random value
```

- See also:
  - randu() & randn()
  - randg()
  - randperm()
  - .imbue()
  - ones()
  - zeros()

- shuffle()
- size()

---

speye(n_rows, n_cols)
speye(size(X))

- Generate a sparse matrix with the elements along the main diagonal set to one and off-diagonal elements set to zero
- An identity matrix is generated when *n_rows* = *n_cols*
- Usage:
  - *sparse_matrix_type* X = speye<*sparse_matrix_type*>(n_rows, n_cols)
  - *sparse_matrix_type* Y = speye<*sparse_matrix_type*>(size(X))
- Examples:

```
sp_mat A = speye<sp_mat>(5,5);
```

- See also:
  - spones()
  - sprandu() & sprandn()
  - eye()
  - size()

---

spones(A)

- Generate a sparse matrix with the same structure as sparse matrix *A*, but with the non-zero elements set to one
- Examples:

```
sp_mat A = sprandu<sp_mat>(100, 200, 0.1);

sp_mat B = spones(A);
```

- See also:
  - speye()
  - sprandu() & sprandn()
  - ones()

---

sprandu(n_rows, n_cols, density)
sprandn(n_rows, n_cols, density)

sprandu(size(X), density)
sprandn(size(X), density)

- Generate a sparse matrix with the non-zero elements set to random values

- The *density* argument specifies the percentage of non-zero elements; it must be in the [0,1] interval
- *sprandu()* uses a uniform distribution in the [0,1] interval
- *sprandn()* uses a normal/Gaussian distribution with zero mean and unit variance
- Usage:
  - *sparse_matrix_type* X = sprandu<*sparse_matrix_type*>(n_rows, n_cols, density)
  - *sparse_matrix_type* Y = sprandu<*sparse_matrix_type*>(size(X), density)
- To change the RNG seed, use *arma_rng::set_seed(value)* or *arma_rng::set_seed_random()* functions
- Examples:

```
sp_mat A = sprandu<sp_mat>(100, 200, 0.1);
```

- See also:
  - speye()
  - spones()
  - randu() & randn()
  - size()
  - uniform distribution in Wikipedia
  - normal distribution in Wikipedia

---

toeplitz(A)
toeplitz(A, B)
circ_toeplitz(A)

- toeplitz(): generate a Toeplitz matrix, with the first column specified by *A*, and (optionally) the first row specified by *B*
- circ_toeplitz(): generate a circulant Toeplitz matrix
- A and B must be vectors
- Examples:

```
vec A = randu<vec>(5);
mat X = toeplitz(A);
mat Y = circ_toeplitz(A);
```

- See also:
  - Toeplitz matrix in MathWorld
  - Toeplitz matrix in Wikipedia
  - Circulant matrix in Wikipedia

Functions of Vectors/Matrices/Cubes

---

abs(X)

- Obtain the magnitude of each element
- Usage for non-complex X:
  - Y = abs(X)
  - X and Y must have the same matrix type or cube type, such as *mat* or *cube*
- Usage for complex X:
  - *real_object_type* Y = abs(X)
  - The type of X must be a complex matrix or complex cube, such as *cx_mat* or *cx_cube*
  - The type of Y must be the real counterpart to the type of X; if X has the type *cx_mat*, then the type of Y must be *mat*
- Examples:

```
mat A = randu<mat>(5,5);
mat B = abs(A);

cx_mat X = randu<cx_mat>(5,5);
   mat Y = abs(X);
```

- See also:
  - arg()
  - conj()
  - imag() / real()
  - conv_to()

---

accu(X)

- Accumulate (sum) all elements of a vector, matrix or cube
- Examples:

```
mat A(5, 6, fill::randu);
mat B(5, 6, fill::randu);

double x = accu(A);

double y = accu(A % B);

// accu(A % B) is a "multiply-and-accumulate" operation
// as operator % performs element-wise multiplication
```

- See also:
  - sum()
  - cumsum()
  - trace()
  - mean()
  - dot()
  - as_scalar()

---

affmul(A, B)

- Multiply matrix *A* by an augmented form of *B*, where a row with ones is appended to B;
  for example:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_0 \\ B_1 \\ 1 \end{bmatrix}$$

- *A* is typically an affine transformation matrix
- The number of columns in *A* must be equal to number of rows in the augmented form of *B* (ie. A.n_cols = B.n_rows+1)
- *B* can be a vector or matrix
- Examples:

```
mat44 A; A.randu();
vec3  B; B.randu();

vec4  C = affmul(A,B);
```

- See also:
  - math operators
  - Affine transformation in Wikipedia
  - Transformation matrix in Wikipedia

---

all(V)
all(X)
all(X, dim)

- For vector *V*, return *true* if all elements of the vector are non-zero or satisfy a relational condition
- For matrix *X* and
  - *dim=0*, return a row vector (of type urowvec or umat), with each element (0 or 1) indicating whether the corresponding column of *X* has all non-zero elements
  - *dim=1*, return a column vector (of type ucolvec or umat), with each element (0 or 1) indicating whether the corresponding row of *X* has all non-zero elements
- The *dim* argument is optional; by default *dim=0* is used
- Relational operators can be used instead of *V* or *X*, eg. *A > 0.5*
- Examples:

```
vec V = randu<vec>(10);
mat X = randu<mat>(5,5);
```

```
// status1 will be set to true if vector V has all non-zero elements
bool status1 = all(V);

// status2 will be set to true if vector V has all elements greater than 0.5
bool status2 = all(V > 0.5);

// status3 will be set to true if matrix X has all elements greater than 0.6;
// note the use of vectorise()
bool status3 = all(vectorise(X) > 0.6);

// generate a row vector indicating which columns of X have all elements greater than 0.7
umat A = all(X > 0.7);
```

- See also:
  - any()
  - approx_equal()
  - find()
  - .is_zero()
  - conv_to() (convert between matrix/vector types)
  - vectorise()

---

any(V)
any(X)
any(X, dim)

- For vector *V*, return *true* if any element of the vector is non-zero or satisfies a relational condition
- For matrix *X* and
  - *dim=0*, return a row vector (of type urowvec or umat), with each element (0 or 1) indicating whether the corresponding column of *X* has any non-zero elements
  - *dim=1*, return a column vector (of type ucolvec or umat), with each element (0 or 1) indicating whether the corresponding row of *X* has any non-zero elements
- The *dim* argument is optional; by default *dim=0* is used
- Relational operators can be used instead of *V* or *X*, eg. *A > 0.9*
- Examples:

```
vec V = randu<vec>(10);
mat X = randu<mat>(5,5);


// status1 will be set to true if vector V has any non-zero elements
bool status1 = any(V);

// status2 will be set to true if vector V has any elements greater than 0.5
bool status2 = any(V > 0.5);

// status3 will be set to true if matrix X has any elements greater than 0.6;
// note the use of vectorise()
bool status3 = any(vectorise(X) > 0.6);

// generate a row vector indicating which columns of X have elements greater than 0.7
```

```
umat A = any(X > 0.7);
```

- See also:
  - all()
  - approx_equal()
  - find()
  - conv_to() (convert between matrix/vector types)
  - vectorise()

---

approx_equal(A, B, method, tol)
approx_equal(A, B, method, abs_tol, rel_tol)

- Return *true* if all corresponding elements in *A* and *B* are approximately equal
- Return *false* if any of the corresponding elements in *A* and *B* are not approximately equal, or if *A* and *B* have different dimensions
- The argument *method* controls how the approximate equality is determined; it is one of:
  `"absdiff"` ↦ scalars *x* and *y* are considered equal if $|x - y| \leq tol$
  `"reldiff"` ↦ scalars *x* and *y* are considered equal if $|x - y| / max(|x|, |y|) \leq tol$
  `"both"` ↦ scalars *x* and *y* are considered equal if $|x - y| \leq abs\_tol$ **or** $|x - y| / max(|x|, |y|) \leq rel\_tol$
- Examples:

```
mat A = randu<mat>(5,5);
mat B = A + 0.001;

bool same1 = approx_equal(A, B, "absdiff", 0.002);


mat C = 1000 * randu<mat>(5,5);
mat D = C + 1;

bool same2 = approx_equal(C, D, "reldiff", 0.1);

bool same3 = approx_equal(C, D, "both", 2, 0.1);
```

- See also:
  - all()
  - any()
  - find()
  - .is_zero()
  - relational operators

---

arg(X)

- Obtain the phase angle (in radians) of each element
- Usage for non-complex X:
  - Y = arg(X)

- X and Y must have the same matrix type or cube type, such as *mat* or *cube*
- non-complex elements are treated as complex elements with zero imaginary component

- Usage for complex X:
  - *real_object_type* Y = arg(X)
  - The type of X must be a complex matrix or complex cube, such as *cx_mat* or *cx_cube*
  - The type of Y must be the real counterpart to the type of X; if X has the type *cx_mat*, then the type of Y must be *mat*

- Examples:

```
cx_mat A = randu<mat>(5,5);
   mat B = arg(A);
```

- See also:
  - abs()
  - atan2()
  - Argument (complex analysis) in Wikipedia
  - Complex Argument in MathWorld

---

as_scalar(expression)

- Evaluate an expression that results in a 1x1 matrix, followed by converting the 1x1 matrix to a pure scalar

- Optimised expression evaluations are automatically used when a binary or trinary expression is given (ie. 2 or 3 terms)

- Examples:

```
rowvec r = randu<rowvec>(5);
colvec q = randu<colvec>(5);
mat    X = randu<mat>(5,5);

// examples of expressions which have optimised implementations

double a = as_scalar(r*q);
double b = as_scalar(r*X*q);
double c = as_scalar(r*diagmat(X)*q);
double d = as_scalar(r*inv(diagmat(X))*q);
```

- See also:
  - .as_col() / .as_row()
  - vectorise()
  - accu()
  - trace()
  - dot()
  - norm()
  - conv_to()

---

clamp(X, min_val, max_val)

- Create a copy of *X* with each element clamped to the [*min_val*, *max_val*] interval;
  any value lower than *min_val* will be set to *min_val*, and any value higher than *max_val* will be set to *max_val*
- If *X* is a sparse matrix, clamping is applied only to the non-zero elements
- Examples:

```
mat A = randu<mat>(5,5);

mat B = clamp(A, 0.2,     0.8);

mat C = clamp(A, A.min(), 0.8);

mat D = clamp(A, 0.2, A.max());
```

- See also:
  - .min() & .max()
  - .clean()
  - .replace()
  - find()

---

cond(A)

- Return the condition number of matrix *A* (the ratio of the largest singular value to the smallest)
- Large condition numbers suggest that matrix *A* is nearly singular
- The computation is based on singular value decomposition; if the decomposition fails, a *std::runtime_error* exception is thrown
- **Caveat:** the rcond() function is faster for providing an estimate of the reciprocal of the condition number
- Examples:

```
mat    A = randu<mat>(5,5);
double c = cond(A);
```

- See also:
  - rcond()
  - rank()
  - inv()
  - solve()
  - condition number in MathWorld
  - condition number in Wikipedia

---

conj(X)

- Obtain the complex conjugate of each element in a complex matrix or cube
- Examples:

```
cx_mat X = randu<cx_mat>(5,5);
cx_mat Y = conj(X);
```

- See also:
  - abs()
  - imag() / real()
  - trans()

---

conv_to< *type* >::from(X)

- Convert from one matrix type to another (eg. *mat* to *imat*), or one cube type to another (eg. *cube* to *icube*)
- Conversion between *std::vector* and Armadillo matrices/vectors is also possible
- Conversion of a *mat* object into *colvec*, *rowvec* or *std::vector* is possible if the object can be interpreted as a vector
- Examples:

```
mat  A = randu<mat>(5,5);
fmat B = conv_to<fmat>::from(A);

typedef std::vector<double> stdvec;

stdvec x(3);
x[0] = 0.0; x[1] = 1.0;  x[2] = 2.0;

colvec y = conv_to< colvec >::from(x);
stdvec z = conv_to< stdvec >::from(y);
```

- See also:
  - as_scalar()
  - abs()
  - imag() / real()
  - advanced constructors (matrices)
  - advanced constructors (cubes)

---

cross(A, B)

- Calculate the cross product between A and B, under the assumption that A and B are 3 dimensional vectors
- Examples:

```
vec a = randu<vec>(3);
vec b = randu<vec>(3);

vec c = cross(a,b);
```

- See also:
  - dot()

- Cross product in Wikipedia
- Cross product in MathWorld

---

cumsum(V)
cumsum(X)
cumsum(X, dim)

- For vector *V*, return a vector of the same orientation, containing the cumulative sum of elements
- For matrix *X*, return a matrix containing the cumulative sum of elements in each column (*dim=0*), or each row (*dim=1*)
- The *dim* argument is optional; by default *dim=0* is used
- Examples:

```
mat A = randu<mat>(5,5);
mat B = cumsum(A);
mat C = cumsum(A, 1);

vec x = randu<vec>(10);
vec y = cumsum(x);
```

- See also:
  - cumprod()
  - accu()
  - sum()
  - diff()

---

cumprod(V)
cumprod(X)
cumprod(X, dim)

- For vector *V*, return a vector of the same orientation, containing the cumulative product of elements
- For matrix *X*, return a matrix containing the cumulative product of elements in each column (*dim=0*), or each row (*dim=1*)
- The *dim* argument is optional; by default *dim=0* is used
- Examples:

```
mat A = randu<mat>(5,5);
mat B = cumprod(A);
mat C = cumprod(A, 1);

vec x = randu<vec>(10);
vec y = cumprod(x);
```

- See also:

- cumsum()
- prod()

---

det(A)

- Determinant of square matrix *A*
- If *A* is not square sized, a *std::logic_error* exception is thrown
- **Caveat**: for large matrices *log_det()* is more precise than *det()*
- Examples:

```
mat A(5,5,fill::randu);

double x = det(A);
```

- See also:
  - log_det()
  - rcond()
  - determinant in MathWorld
  - determinant in Wikipedia

---

diagmat(V)
diagmat(V, k)

diagmat(X)
diagmat(X, k)

- Generate a diagonal matrix from vector *V* or matrix *X*
- Given vector *V*, generate a square matrix with the *k*-th diagonal containing a copy of the vector; all other elements are set to zero
- Given matrix *X*, generate a matrix with the *k*-th diagonal containing a copy of the *k*-th diagonal of *X*; all other elements are set to zero
- The argument *k* is optional; by default the main diagonal is used (*k=0*)
- For *k > 0*, the *k*-th super-diagonal is used (above main diagonal, towards top-right corner)
- For *k < 0*, the *k*-th sub-diagonal is used (below main diagonal, towards bottom-left corner)
- Examples:

```
mat A = randu<mat>(5,5);
mat B = diagmat(A);
mat C = diagmat(A,1);

vec v = randu<vec>(5);
```

```
mat D = diagmat(v);
mat E = diagmat(v,1);
```

- See also:
  - .diag()
  - .is_diagmat()
  - diagvec()
  - trimatu() / trimatl()
  - symmatu() / symmatl()
  - reshape()
  - vectorise()
  - diagonal matrix in Wikipedia

---

diagvec(A)
diagvec(A, k)

- Extract the *k*-th diagonal from matrix *A*
- The argument *k* is optional; by default the main diagonal is extracted (*k=0*)
- For *k > 0*, the *k*-th super-diagonal is extracted (top-right corner)
- For *k < 0*, the *k*-th sub-diagonal is extracted (bottom-left corner)
- The extracted diagonal is interpreted as a column vector
- Examples:

```
mat A = randu<mat>(5,5);

vec d = diagvec(A);
```

- See also:
  - .diag()
  - diagmat()
  - trace()
  - vectorise()

---

diff(V)
diff(V, k)

diff(X)
diff(X, k)
diff(X, k, dim)

- For vector *V*, return a vector of the same orientation, containing the differences between consecutive elements
- For matrix *X*, return a matrix containing the differences between consecutive elements in each column (*dim=0*),

or each row (*dim=1*)

- The optional argument *k* indicates that the differences are calculated recursively *k* times; by default *k=1* is used
- The resulting number of differences is *n* - *k*, where *n* is the number of elements; if $n \leq k$, the number of differences is zero (ie. an empty vector/matrix is returned)
- The argument *dim* is optional; by default *dim=0*
- Examples:

```
vec a = linspace<vec>(1,10,10);

vec b = diff(a);
```

- See also:
  - trapz()
  - numerical differentiation in Wikipedia
  - numerical differentiation in MathWorld

---

dot(A, B)
cdot(A, B)
norm_dot(A, B)

- *dot(A,B)*: dot product of *A* and *B*, treating *A* and *B* as vectors
- *cdot(A,B)*: as per *dot(A,B)*, but the complex conjugate of *A* is used
- *norm_dot(A,B)*: normalised dot product; equivalent to *dot(A,B) / (* $\|A\| \bullet \|B\|$ *)*
- **Caveat:** norm() is more robust for calculating the norm, as it handles underflows and overflows
- Examples:

```
vec a = randu<vec>(10);
vec b = randu<vec>(10);

double x = dot(a,b);
```

- See also:
  - norm()
  - as_scalar()
  - cross()
  - conj()

---

eps(X)

- Obtain the positive distance of the absolute value of each element of *X* to the next largest representable floating point number

- *X* can be a scalar (eg. *double*), vector or matrix
- Examples:

```
mat A = randu<mat>(4,5);
mat B = eps(A);
```

- See also:
  - datum::eps
  - Floating-Point Arithmetic in MathWorld
  - IEEE Standard for Floating-Point Arithmetic in Wikipedia

B = expmat(A)
expmat(B, A)

- Matrix exponential of general square matrix *A*
- If *A* is not square sized, a *std::logic_error* exception is thrown
- If the matrix exponential cannot be found:
  - *B = expmat(A)* resets *B* and throws a *std::runtime_error* exception
  - *expmat(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)
- **Caveat**: if matrix *A* is symmetric, using expmat_sym() is faster
- **Caveat**: the matrix exponential operation is generally **not** the same as applying the exp() function to each element
- Examples:

```
mat A = randu<mat>(5,5);

mat B = expmat(A);
```

- See also:
  - expmat_sym()
  - logmat()
  - sqrtmat()
  - miscellaneous element-wise functions
  - matrix exponential in Wikipedia
  - matrix exponential in MathWorld
  - Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later

---

B = expmat_sym(A)
expmat_sym(B, A)

- Matrix exponential of symmetric/hermitian matrix *A*

- The computation is based on eigen decomposition
- If *A* is not square sized, a *std::logic_error* exception is thrown
- If the matrix exponential cannot be found:
  - *B = expmat_sym(A)* resets *B* and throws a *std::runtime_error* exception
  - *expmat_sym(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)
- **Caveat**: the matrix exponential operation is generally **not** the same as applying the exp() function to each element
- Examples:

```
mat A = randu<mat>(5,5);

mat B = A*A.t();   // make symmetric matrix

mat C = expmat_sym(B);
```

- See also:
  - expmat()
  - logmat_sympd()
  - sqrtmat_sympd()
  - .is_symmetric()
  - .is_hermitian()
  - miscellaneous element-wise functions
  - matrix exponential in Wikipedia
  - matrix exponential in MathWorld
  - Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later

---

find(X)
find(X, k)
find(X, k, s)

- Return a column vector containing the indices of elements of *X* that are non-zero or satisfy a relational condition
- The output vector must have the type uvec (ie. the indices are stored as unsigned integers of type uword)
- *X* is interpreted as a vector, with column-by-column ordering of the elements of *X*
- Relational operators can be used instead of *X*, eg. *A > 0.5*
- If *k=0* (default), return the indices of all non-zero elements, otherwise return at most *k* of their indices
- If *s="first"* (default), return at most the first *k* indices of the non-zero elements
- If *s="last"*, return at most the last *k* indices of the non-zero elements
- **Caveats:**
  - to clamp values to an interval, clamp() is more efficient
  - to replace a specific value, .replace() is more efficient

- Examples:

```
mat  A  = randu<mat>(5,5);
mat  B  = randu<mat>(5,5);

uvec q1 = find(A > B);
uvec q2 = find(A > 0.5);
uvec q3 = find(A > 0.5, 3, "last");

// change elements of A greater than 0.5 to 1
A.elem( find(A > 0.5) ).ones();
```

- See also:
  - all()
  - any()
  - clamp()
  - .transform()
  - approx_equal()
  - find_finite()
  - find_nonfinite()
  - find_unique()
  - nonzeros()
  - unique()
  - sort_index()
  - ind2sub()
  - .index_min() & .index_max()
  - trimatu_ind() / trimatl_ind()
  - submatrix views
  - subcube views

---

find_finite(X)

- Return a column vector containing the indices of elements of *X* that are finite (ie. not ±Inf and not NaN)
- The output vector must have the type uvec (ie. the indices are stored as unsigned integers of type uword)
- *X* is interpreted as a vector, with column-by-column ordering of the elements of *X*
- Examples:

```
mat A = randu<mat>(5,5);

A(1,1) = datum::inf;

// accumulate only finite elements
double val = accu( A.elem( find_finite(A) ) );
```

- See also:
  - find()
  - find_nonfinite()
  - .is_finite()
  - .replace()

- .has_inf()
- .has_nan()
- submatrix views
- constants (pi, nan, inf, ...)

---

find_nonfinite(X)

- Return a column vector containing the indices of elements of *X* that are non-finite (ie. ±Inf or NaN)
- The output vector must have the type uvec (ie. the indices are stored as unsigned integers of type uword)
- *X* is interpreted as a vector, with column-by-column ordering of the elements of *X*
- Examples:

```
mat A = randu<mat>(5,5);

A(1,1) = datum::inf;
A(2,2) = datum::nan;

// change non-finite elements to zero
A.elem( find_nonfinite(A) ).zeros();
```

- **Caveat:** to replace instances of a specific non-finite value (eg. nan or inf), it's more efficient to use .replace()
- See also:
  - find()
  - find_finite()
  - .is_finite()
  - .replace()
  - .has_inf()
  - .has_nan()
  - submatrix views
  - constants (pi, nan, inf, ...)

---

find_unique(X)
find_unique(X, ascending_indices)

- Return a column vector containing the indices of unique elements of *X*
- The output vector must have the type uvec (ie. the indices are stored as unsigned integers of type uword)
- *X* is interpreted as a vector, with column-by-column ordering of the elements of *X*
- The *ascending_indices* argument is optional; it is one of:
  `true` = the returned indices are sorted to be ascending (**default setting**)
  `false` = the returned indices are in arbitrary order (faster operation)
- Examples:

```
mat A;

A << 2 << 2 << 4 << endr
  << 4 << 6 << 6 << endr;

uvec indices = find_unique(A);
```

- See also:
  - find()
  - unique()
  - intersect()
  - submatrix views

---

fliplr(X)
flipud(X)

- *fliplr():* generate a copy of matrix *X*, with the order of the columns reversed
- *flipud():* generate a copy of matrix *X*, with the order of the rows reversed
- Examples:

```
mat A = randu<mat>(5,5);

mat B = fliplr(A);
mat C = flipud(A);
```

- See also:
  - reverse()
  - shift()
  - .swap_rows() & .swap_cols()
  - .t()

---

imag(X)
real(X)

- Extract the imaginary/real part of a complex matrix or cube
- Examples:

```
cx_mat C = randu<cx_mat>(5,5);

mat    A = imag(C);
mat    B = real(C);
```

- **Caveat:** versions 4.4, 4.5 and 4.6 of the GCC C++ compiler have a bug when using the *-std=c++0x* compiler option (ie. experimental support for C++11); to work around this bug, preface Armadillo's *imag()* and *real()* with the *arma* namespace qualification, eg. *arma::imag(C)*
- See also:

- .set_imag() / .set_real()
- abs()
- conj()
- conv_to()

---

uvec sub = **ind2sub(size(**X**),** index **)** (form 1)
umat sub = **ind2sub(size(**X**),** vector_of_indices **)** (form 2)

- Convert a linear index, or a vector of indices, to subscript notation
- The argument ***size(X)*** can be replaced with ***size(****n_rows, n_cols****)*** or ***size(****n_rows, n_cols, n_slices****)***
- A *std::logic_error* exception is thrown if an index is out of range
- When only one index is given (form 1), the subscripts are returned in a vector of type uvec
- When a vector of indices (of type uvec) is given (form 2), the corresponding subscripts are returned in each column of an *m x n* matrix of type umat; *m=2* for matrix subscripts, while *m=3* for cube subscripts
- Examples:

```
mat M(4, 5, fill::randu);

uvec s = ind2sub( size(M), 6 );

cout << "row: " << s(0) << endl;
cout << "col: " << s(1) << endl;


uvec indices = find(M > 0.5);
umat t       = ind2sub( size(M), indices );


cube Q(2,3,4);

uvec u = ind2sub( size(Q), 8 );

cout << "row:   " << u(0) << endl;
cout << "col:   " << u(1) << endl;
cout << "slice: " << u(2) << endl;
```

- See also:
  - size()
  - sub2ind()
  - element access
  - find()

---

index_min(V) **index_max(V)**
index_min(M) **index_max(M)**
index_min(M, dim) **index_max(M, dim)**
index_min(Q) **index_max(Q)**
index_min(Q, dim) **index_max(Q, dim)**

- For vector *V*, return the linear index of the extremum value; the returned index is of type uword
- For matrix *M* and:
  - *dim=0*, return a row vector (of type urowvec or umat), with each column containing the index of the extremum value in the corresponding column of *M*
  - *dim=1*, return a column vector (of type uvec or umat), with each row containing the index of the extremum value in the corresponding row of *M*
- For cube *Q*, return a cube (of type ucube) containing the indices of extremum values of elements along dimension *dim*, where *dim* $\in$ { 0, 1, 2 }
- For each column, row, or slice, the index starts at zero
- The *dim* argument is optional; by default *dim=0* is used
- For objects with complex numbers, absolute values are used for comparison
- Examples:

```
vec   v = randu<vec>(10);
uword i = index_max(v);

double max_val_in_v = v(i);


mat     M  = randu<mat>(5,6);

urowvec ii = index_max(M);
ucolvec jj = index_max(M,1);

double max_val_in_col_2 = M( ii(2), 2 );

double max_val_in_row_4 = M( 4, jj(4) );
```

- See also:
  - min() & max()
  - .index_min() & .index_max() (member functions)
  - sort_index()
  - find()

---

inplace_trans(X)
inplace_trans(X, method)

inplace_strans(X)
inplace_strans(X, method)

- In-place / in-situ transpose of matrix *X*

- For real (non-complex) matrix:
  - *inplace_trans()* performs a normal transpose
  - *inplace_strans()* not applicable
- For complex matrix:
  - *inplace_trans()* performs a Hermitian transpose (ie. the conjugate of the elements is taken during the transpose)
  - *inplace_strans()* provides a transposed copy without taking the conjugate of the elements
- The argument *method* is optional
- By default, a greedy transposition algorithm is used; a low-memory algorithm can be used instead by explicitly setting *method* to `"lowmem"`
- The low-memory algorithm is considerably slower than the greedy algorithm; using the low-memory algorithm is only recommended for cases where *X* takes up more than half of available memory (ie. very large *X*)
- Examples:

```
mat X = randu<mat>(4,5);
mat Y = randu<mat>(20000,30000);

inplace_trans(X);            // use greedy algorithm by default

inplace_trans(Y, "lowmem");  // use low-memory (and slow) algorithm
```

- See also:
  - .t()
  - trans()
  - inplace matrix transpose in Wikipedia

C = intersect(A, B) (form 1)
intersect(C, iA, iB, A, B) (form 2)

- For form 1:
  - return the unique elements common to both *A* and *B*, sorted in ascending order
- For form 2:
  - store in *C* the unique elements common to both *A* and *B*, sorted in ascending order
  - store in *iA* and *iB* the indices of the unique elements, such that *C = A.elem(iA)* and *C = B.elem(iB)*
  - *iA* and *iB* must have the type uvec (ie. the indices are stored as unsigned integers of type uword)
- *C* is a column vector if either *A* or *B* is a matrix or column vector; *C* is a row vector if both *A* and *B* are row vectors
- For matrices and vectors with complex numbers, ordering is via absolute values
- Examples:

```
ivec A = regspace<ivec>(4, 1);  // 4, 3, 2, 1
ivec B = regspace<ivec>(3, 6);  // 3, 4, 5, 6
```

```
ivec C = intersect(A,B);       // 3, 4

ivec CC;
uvec iA;
uvec iB;

intersect(CC, iA, iB, A, B);
```

- See also:
  - unique()
  - find_unique()
  - submatrix views

---

join_rows(A, B)
join_rows(A, B, C)
join_rows(A, B, C, D)

join_horiz(A, B)
join_horiz(A, B, C)
join_horiz(A, B, C, D)

join_cols(A, B)
join_cols(A, B, C)
join_cols(A, B, C, D)

join_vert(A, B)
join_vert(A, B, C)
join_vert(A, B, C, D)

- *join_rows()* and *join_horiz()*: horizontal concatenation; join the corresponding rows of the given matrices; the given matrices must have the same number of rows

- *join_cols()* and *join_vert()*: vertical concatenation; join the corresponding columns of the given matrices; the given matrices must have the same number of columns

- Examples:

```
mat A = randu<mat>(4,5);
mat B = randu<mat>(4,6);
mat C = randu<mat>(6,5);

mat AB = join_rows(A,B);
mat AC = join_cols(A,C);
```

- See also:
  - .shed_rows/cols/slices
  - .insert_rows/cols/slices
  - submatrix views

---

join_slices(cube C, cube D)
join_slices(mat M, mat N)

join_slices(mat M, cube C)
join_slices(cube C, mat M)

- for two cubes *C* and *D*: join the slices of *C* with the slices of *D*; cubes *C* and *D* must have the same number of rows and columns (ie. all slices must have the same size)

- for two matrices *M* and *N*: treat *M* and *N* as cube slices and join them to form a cube with 2 slices; matrices *M* and *N* must have the same number of rows and columns
- for matrix *M* and cube *C*: treat *M* as a cube slice and join it with the slices of *C*; matrix *M* and cube *C* must have the same number of rows and columns
- Examples:

```
cube C(5, 10, 3, fill::randu);
cube D(5, 10, 4, fill::randu);

cube E = join_slices(C,D);

mat M(10, 20, fill::randu);
mat N(10, 20, fill::randu);

cube Q = join_slices(M,N);

cube R = join_slices(Q,M);

cube S = join_slices(M,Q);
```

- See also:
  - .shed_rows/cols/slices
  - .insert_rows/cols/slices
  - subcube views

---

kron(A, B)

- Kronecker tensor product
- Given matrix *A* (with *n* rows and *p* columns) and matrix *B* (with *m* rows and *q* columns), generate a matrix (with *nm* rows and *pq* columns) that denotes the tensor product of *A* and *B*
- Examples:

```
mat A = randu<mat>(4,5);
mat B = randu<mat>(5,4);

mat K = kron(A,B);
```

- See also:
  - repmat()
  - repelem()
  - Kronecker product in MathWorld
  - Kronecker product in Wikipedia

---

log_det(val, sign, A) (form 1)

complex result = log_det(A) (form 2)

- Log determinant of square matrix *A*
- If *A* is not square sized, a *std::logic_error* exception is thrown
- form 1: store the calculated log determinant in *val* and *sign*
  the determinant is equal to *exp(val)*sign*
- form 2: return the complex log determinant
  - if matrix *A* is real and the determinant is positive:
    - the real part of the result is the log determinant
    - the imaginary part is zero
  - if matrix *A* is real and the determinant is negative:
    - the real part of the result is the log of the absolute value of the determinant
    - the imaginary part is equal to *datum::pi*
- Examples:

```
mat A(5,5,fill::randu);

double val;
double sign;

log_det(val, sign, A);          // form 1

cx_double result = log_det(A);  // form 2
```

- See also:
  - det()
  - rcond()
  - cx_double
  - determinant in MathWorld
  - determinant in Wikipedia

B = logmat(A)
logmat(B, A)

- Complex matrix logarithm of general square matrix *A*
- If *A* is not square sized, a *std::logic_error* exception is thrown
- If the matrix logarithm cannot be found:
  - *B = logmat(A)* resets *B* and throws a *std::runtime_error* exception
  - *logmat(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)
- **Caveat**: if matrix *A* is symmetric positive definite, using logmat_sympd() is faster
- **Caveat**: the matrix logarithm operation is generally **not** the same as applying the log() function to each element
- Examples:

```
   mat A = randu<mat>(5,5);

cx_mat B = logmat(A);
```

- See also:
  - logmat_sympd()
  - expmat()
  - sqrtmat()
  - real()
  - miscellaneous element-wise functions
  - matrix logarithm in Wikipedia

B = logmat_sympd(A)
logmat_sympd(B, A)

- Matrix logarithm of symmetric/hermitian positive definite matrix *A*
- If *A* is not square sized, a *std::logic_error* exception is thrown
- If the matrix logarithm cannot be found:
  - *B = logmat_sympd(A)* resets *B* and throws a *std::runtime_error* exception
  - *logmat_sympd(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)
- **Caveat**: the matrix logarithm operation is generally **not** the same as applying the log() function to each element
- Examples:

```
mat A = randu<mat>(5,5);

mat B = A*A.t();   // make symmetric matrix

mat C = logmat_sympd(B);
```

- See also:
  - logmat()
  - expmat_sym()
  - sqrtmat_sympd()
  - .is_sympd()
  - miscellaneous element-wise functions
  - matrix logarithm in Wikipedia
  - positive definite matrix in Wikipedia
  - positive definite matrix in MathWorld

---

min(V)	**max(V)**
min(M)	**max(M)**
min(M, dim)	**max(M, dim)**
min(Q)	**max(Q)**
min(Q, dim)	**max(Q, dim)**
min(A, B)	**max(A, B)**

- For vector *V*, return the extremum value

- For matrix *M*, return the extremum value for each column (*dim=0*), or each row (*dim=1*)
- For cube *Q*, return the extremum values of elements along dimension *dim*, where *dim* $\in$ { 0, 1, 2 }
- The *dim* argument is optional; by default *dim=0* is used
- For two matrices/cubes *A* and *B*, return a matrix/cube containing element-wise extremum values
- For objects with complex numbers, absolute values are used for comparison
- Examples:

```
colvec v = randu<colvec>(10,1);
double x = max(v);

mat    M = randu<mat>(10,10);

rowvec a = max(M);
rowvec b = max(M,0);
colvec c = max(M,1);

// element-wise maximum
mat X = randu<mat>(5,6);
mat Y = randu<mat>(5,6);
mat Z = arma::max(X,Y);  // use arma:: prefix to distinguish from std::max()
```

- See also:
  - .min() & .max() (member functions)
  - index_min() & index_max()
  - clamp()
  - statistics functions
  - running_stat - class for running statistics of scalars
  - running_stat_vec - class for running statistics of vectors
  - ensmallen - library for finding minimum of arbitrary function

---

nonzeros(X)

- Return a column vector containing the non-zero values of *X*
- *X* can be a sparse or dense matrix
- **Caveat:** do not use *nonzeros()* if you only want to obtain the number of non-zero elements in a sparse matrix; use the *.n_nonzero* attribute instead, eg. *X.n_nonzero*
- Examples:

```
sp_mat A = sprandu<sp_mat>(100, 100, 0.1);
   vec a = nonzeros(A);

mat B(100, 100, fill::eye);
vec b = nonzeros(B);
```

- See also:

- find()
- unique()
- vectorise()
- .for_each()

---

norm(X)
norm(X, p)

- Compute the *p*-norm of *X*, where *X* can be a vector or matrix
- For vectors, *p* is an integer ≥1, or one of: `"-inf"`, `"inf"`, `"fro"`
- For matrices, *p* is one of: 1, 2, `"inf"`, `"fro"`; the calculated norm is the *induced norm* (not entrywise norm)
- `"-inf"` is the minimum norm, `"inf"` is the maximum norm, while `"fro"` is the Frobenius norm
- The argument *p* is optional; by default *p=2* is used
- For vector norm with *p=2* and matrix norm with *p="fro"*, a robust algorithm is used to reduce the likelihood of underflows and overflows
- To obtain the zero norm or Hamming norm (ie. the number of non-zero elements), use this expression:
  `accu(X != 0)`
- Examples:

```
vec    q = randu<vec>(5);

double x = norm(q, 2);
double y = norm(q, "inf");
```

- See also:
  - normalise()
  - vectorise()
  - dot()
  - vector norm in Wikipedia
  - vector norm in MathWorld
  - matrix norm in Wikipedia
  - matrix norm in MathWorld

---

normalise(V)
normalise(V, p)

normalise(X)
normalise(X, p)
normalise(X, p, dim)

- For vector *V*, return its normalised version (ie. having unit *p*-norm)

- For matrix *X*, return its normalised version, where each column (*dim=0*) or row (*dim=1*) has been normalised to have unit *p*-norm
- The *p* argument is optional; by default *p=2* is used
- The *dim* argument is optional; by default *dim=0* is used
- Examples:

```
vec A = randu<vec>(10);
vec B = normalise(A);
vec C = normalise(A, 1);

mat X = randu<mat>(5,6);
mat Y = normalise(X);
mat Z = normalise(X, 2, 1);
```

- See also:
  - norm()
  - norm_dot()
  - Normalised vector in MathWorld
  - Unit vector in Wikipedia

---

prod(V)
prod(M)
prod(M, dim)

- For vector *V*, return the product of all elements
- For matrix *M*, return the product of elements in each column (*dim=0*), or each row (*dim=1*)
- The *dim* argument is optional; by default *dim=0* is used
- Examples:

```
colvec v = randu<colvec>(10,1);
double x = prod(v);

mat    M = randu<mat>(10,10);

rowvec a = prod(M);
rowvec b = prod(M,0);
colvec c = prod(M,1);
```

- See also:
  - cumprod()
  - Schur product

B = powmat(A, n)
powmat(B, A, n)

- Matrix power operation: raise square matrix *A* to the power of *n*, where *n* has the type *int* or *double*
- If *n* has the type *double*, the resultant matrix *B* always has complex elements
- For *n = 0*, an identity matrix is generated
- If *A* is not square sized, a *std::logic_error* exception is thrown
- If the matrix power cannot be found:
  - *B = powmat(A)* resets *B* and throws a *std::runtime_error* exception
  - *powmat(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)
- **Caveats**:
  - to find the inverse of a matrix, use inv() instead
  - to solve a system of linear equations, use solve() instead
  - to find the matrix square root, use sqrtmat() instead
  - the matrix power operation is generally **not** the same as applying the pow() function to each element
- Examples:

```
mat A = randu<mat>(5,5);

mat B = powmat(A, 4);     //     integer exponent

cx_mat C = powmat(A, 4.56);  // non-integer exponent
```

- See also:
  - sqrtmat()
  - inv()
  - eye()
  - operators
  - miscellaneous element-wise functions

---

rank(X)
rank(X, tolerance)

- Return the rank of matrix *X*
- Any singular values less than *tolerance* are treated as zero
- The *tolerance* argument is optional; by default *tolerance* is *max(m,n)*max_sv*datum::eps*, where:
  - *m* = number of rows and *n* = number of columns in *X*
  - *max_sv* = maximal singular value of *X*
  - *datum::eps* = difference between 1 and the least value greater than 1 that is representable
- The computation is based on singular value decomposition; if the decomposition fails, a *std::runtime_error* exception is thrown
- **Caveat:** to confirm whether a matrix is singular, use rcond() or cond()
- Examples:

```
mat   A = randu<mat>(4,5);

uword r = rank(A);
```

- See also:
  - cond()
  - rcond()
  - svd()
  - orth()
  - datum::eps
  - Rank in MathWorld
  - Rank in Wikipedia

---

rcond(A)

- Return the 1-norm estimate of the reciprocal of the condition number of square matrix *A*
- Values close to 1 suggest that *A* is well-conditioned
- Values close to 0 suggest that *A* is badly conditioned
- If *A* is not square sized, a *std::logic_error* exception is thrown
- Examples:

```
mat    A = randu<mat>(5,5);
double r = rcond(A);
```

- See also:
  - cond()
  - rank()
  - det()
  - inv()
  - solve()

---

repelem(A, num_copies_per_row, num_copies_per_col)

- Generate a matrix by replicating each element of matrix *A*
- The generated matrix has the following size:
  n_rows = num_copies_per_row * A.n_rows
  n_cols = num_copies_per_col * A.n_cols
- Examples:

```
mat A = randu<mat>(2, 3);

mat B = repelem(A, 4, 5);
```

- See also:

- repmat()
- kron()
- reshape()
- resize()

---

repmat(A, num_copies_per_row, num_copies_per_col)

- Generate a matrix by replicating matrix *A* in a block-like fashion
- The generated matrix has the following size:
  n_rows = num_copies_per_row*A.n_rows
  n_cols = num_copies_per_col *A.n_cols
- **Caveat:** to apply a vector operation on each row or column of a matrix, it's generally more efficient to use .each_row() or .each_col()
- Examples:

```
mat A = randu<mat>(2, 3);

mat B = repmat(A, 4, 5);
```

- See also:
  - .each_col() & .each_row() (vector operations applied to each column or row)
  - repelem()
  - kron()
  - reshape()
  - resize()

---

reshape(X, n_rows, n_cols) (X is a vector or matrix)
reshape(X, size(Y))

reshape(Q, n_rows, n_cols, n_slices) (Q is a cube)
reshape(Q, size(R))

- Generate a vector/matrix/cube with given size specifications, whose elements are taken from the given object in a column-wise manner; the elements in the generated object are placed column-wise (ie. the first column is filled up before filling the second column)
- The layout of the elements in the generated object will be different to the layout in the given object
- If the total number of elements in the given object is less than the specified size, the remaining elements in the generated object are set to zero
- If the total number of elements in the given object is greater than the specified size, only a subset of elements is taken from the given object
- **Caveats:**
  - do not use *reshape()* if you simply want to change the size without preserving data; use .set_size() instead, which is much faster

- to grow/shrink a matrix while preserving the elements **as well as** the layout of the elements, use resize() instead
- to flatten a matrix into a vector, use vectorise() or .as_col() / .as_row() instead

- Examples:

```
mat A = randu<mat>(10, 5);
mat B = reshape(A, 5, 10);
```

- See also:
  - .reshape() (member function)
  - .set_size()
  - resize()
  - vectorise()
  - as_scalar()
  - conv_to()
  - diagmat()
  - repmat()
  - repelem()
  - size()
  - interp2()

---

resize(X, n_rows, n_cols) (X is a vector or matrix)
resize(X, size(Y))

resize(Q, n_rows, n_cols, n_slices) (Q is a cube)
resize(Q, size(R))

- Generate a vector/matrix/cube with given size specifications, whose elements as well as the layout of the elements are taken from the given object

- **Caveat:** do not use *resize()* if you simply want to change the size without preserving data; use .set_size() instead, which is much faster

- Examples:

```
mat A = randu<mat>(4, 5);
mat B = resize(A, 7, 6);
```

- See also:
  - .resize() (member function of Mat and Cube)
  - .set_size() (member function of Mat and Cube)
  - reshape()
  - vectorise()
  - as_scalar()
  - conv_to()
  - repmat()
  - repelem()
  - size()

---

reverse(V)
reverse(X)
reverse(X, dim)

- For vector *V*, generate a copy of the vector with the order of elements reversed
- For matrix *X*, generate a copy of the matrix with the order of elements reversed in each column (*dim=0*), or each row (*dim=1*)
- The *dim* argument is optional; by default *dim=0* is used
- Examples:

```
vec v(123, fill::randu);
vec y = reverse(v);

mat A(4, 5, fill::randu);
mat B = reverse(A);
mat C = reverse(A,1);
```

- See also:
  - fliplr() & flipud()
  - shift()
  - sort()
  - .swap_rows() & .swap_cols()
  - .t()

R = roots(P)
roots(R, P)

- Find the complex roots of a polynomial function represented via vector *P* and store them in column vector *R*
- The polynomial function is modelled as:
  $y = p_0 x^N + p_1 x^{N-1} + p_2 x^{N-2} + ... + p_{N-1} x^1 + p_N$
  where p_i is the i-th polynomial coefficient in vector *P*
- The computation is based on eigen decomposition; if the decomposition fails:
  - *R = roots(P)* resets *R* and throws a *std::runtime_error* exception
  - *roots(R,P)* resets *R* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
  vec P = randu<vec>(5);

cx_vec R = roots(P);
```

- See also:
  - polyval()
  - polyfit()
  - real()

- solve()
- zero of a function in Wikpedia

---

shift(V, N)
shift(X, N)
shift(X, N, dim)

- For vector *V*, generate a copy of the vector with the elements shifted by *N* positions in a circular manner
- For matrix *X*, generate a copy of the matrix with the elements shifted by *N* positions in each column (*dim=0*), or each row (*dim=1*)
- *N* can be positive or negative
- The *dim* argument is optional; by default *dim=0* is used
- Examples:

```
mat A = randu<mat>(4,5);
mat B = shift(A, -1);
mat C = shift(A, +1);
```

- See also:
  - shuffle()
  - fliplr() & flipud()
  - reverse()

---

shuffle(V)
shuffle(X)
shuffle(X, dim)

- For vector *V*, generate a copy of the vector with the elements shuffled
- For matrix *X*, generate a copy of the matrix with the elements shuffled in each column (*dim=0*), or each row (*dim=1*)
- The *dim* argument is optional; by default *dim=0* is used
- Examples:

```
mat A = randu<mat>(4,5);
mat B = shuffle(A);
```

- See also:
  - shift()
  - sort()
  - unique()
  - randu() / randn()

size(X)
size(n_rows, n_cols)
size(n_rows, n_cols, n_slices)

- Obtain the dimensions of object *X*, or explicitly specify the dimensions
- The dimensions can be used in conjunction with:
  - object constructors
  - functions for changing size
  - generator functions
  - submatrix views
  - subcube views
  - subfield views
  - ind2sub() and sub2ind()
- The dimensions support simple arithmetic operations; they can also be printed and compared for equality/inequality
- **Caveat:** to prevent interference from *std::size()* in C++17, preface Armadillo's *size()* with the *arma* namespace qualification, eg. *arma::size(X)*
- Examples:

```
mat A(5,6);

mat B(size(A), fill::zeros);

mat C; C.randu(size(A));

mat D = ones<mat>(size(A));

mat E(10,20, fill::ones);
E(3,4,size(C)) = C;    // access submatrix of E

mat F( size(A) + size(E) );

mat G( size(A) * 2 );

cout << "size of A: " << size(A) << endl;

bool is_same_size = (size(A) == size(E));
```

- See also:
  - attributes

---

sort(V)
sort(V, sort_direction)

sort(X)
sort(X, sort_direction)
sort(X, sort_direction, dim)

- For vector *V*, return a vector which is a sorted version of the input vector
- For matrix *X*, return a matrix with the elements of the input matrix sorted in each column (*dim=0*), or each row (*dim=1*)
- The *dim* argument is optional; by default *dim=0* is used
- The *sort_direction* argument is optional; *sort_direction* is either `"ascend"` or `"descend"`; by default `"ascend"` is used
- For matrices and vectors with complex numbers, sorting is via absolute values
- Examples:

```
mat A = randu<mat>(10,10);
mat B = sort(A);
```

- See also:
  - sort_index()
  - .is_sorted()
  - shuffle()
  - unique()
  - reverse()
  - randu() / randn()

---

sort_index(X)
sort_index(X, sort_direction)

stable_sort_index(X)
stable_sort_index(X, sort_direction)

- Return a vector which describes the sorted order of the elements of *X* (ie. it contains the indices of the elements of *X*)
- The output vector must have the type uvec (ie. the indices are stored as unsigned integers of type uword)
- *X* is interpreted as a vector, with column-by-column ordering of the elements of *X*
- The *sort_direction* argument is optional; *sort_direction* is either `"ascend"` or `"descend"`; by default `"ascend"` is used
- The *stable_sort_index()* variant preserves the relative order of elements with equivalent values
- For matrices and vectors with complex numbers, sorting is via absolute values
- Examples:

```
vec  q       = randu<vec>(10);
uvec indices = sort_index(q);
```

- See also:
  - sort()

- find()
- .is_sorted()
- .index_min() & .index_max()
- ind2sub()

B = sqrtmat(A)
sqrtmat(B, A)

- Complex square root of general square matrix *A*
- If *A* is not square sized, a *std::logic_error* exception is thrown
- If matrix *A* appears to be singular, an approximate square root is attempted; additionally, *sqrtmat(B,A)* returns a bool set to false
- **Caveat**: if matrix *A* is symmetric positive definite, using sqrtmat_sympd() is faster
- **Caveat**: the square root of a matrix is generally **not** the same as applying the sqrt() function to each element
- Examples:

```
mat A = randu<mat>(5,5);

cx_mat B = sqrtmat(A);
```

- See also:
  - sqrtmat_sympd()
  - powmat()
  - expmat()
  - logmat()
  - chol()
  - real()
  - miscellaneous element-wise functions
  - square root of a matrix in Wikipedia

B = sqrtmat_sympd(A)
sqrtmat_sympd(B, A)

- Square root of symmetric/hermitian positive definite matrix *A*
- If *A* is not square sized, a *std::logic_error* exception is thrown
- If the square root cannot be found:
  - *B = sqrtmat_sympd(A)* resets *B* and throws a *std::runtime_error* exception
  - *sqrtmat_sympd(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)
- **Caveat**: the matrix square root operation is generally **not** the same as applying the sqrt() function to each element

- Examples:

```
mat A = randu<mat>(5,5);

mat B = A*A.t();   // make symmetric matrix

mat C = sqrtmat_sympd(B);
```

- See also:
  - sqrtmat()
  - expmat_sym()
  - logmat_sympd()
  - chol()
  - .is_sympd()
  - miscellaneous element-wise functions
  - square root of a matrix in Wikipedia
  - positive definite matrix in Wikipedia
  - positive definite matrix in MathWorld

---

sum(V)
sum(M)
sum(M, dim)
sum(Q)
sum(Q, dim)

- For vector *V*, return the sum of all elements
- For matrix *M*, return the sum of elements in each column (*dim=0*), or each row (*dim=1*)
- For cube *Q*, return the sums of elements along dimension *dim*, where *dim* $\in$ { 0, 1, 2 }; for example, *dim=0* indicates the sum of elements in each column within each slice
- The *dim* argument is optional; by default *dim=0* is used
- **Caveat:** to get a sum of all the elements regardless of the object type (ie. vector, or matrix, or cube), use accu() instead
- Examples:

```
colvec v = randu<colvec>(10,1);
double x = sum(v);

mat    M = randu<mat>(10,10);

rowvec a = sum(M);
rowvec b = sum(M,0);
colvec c = sum(M,1);

double y = accu(M);   // find the overall sum regardless of object type
```

- See also:
  - accu()
  - cumsum()

- trace()
- trapz()
- mean()
- as_scalar()

---

uword index = **sub2ind(size(**M**),** row**,** col **)** (*M* is a matrix)
uvec indices = **sub2ind(size(**M**),** matrix_of_subscripts **)**

uword index = **sub2ind(size(**Q**),** row**,** col**,** slice **)** (*Q* is a cube)
uvec indices = **sub2ind(size(**Q**),** matrix_of_subscripts **)**

- Convert subscripts to a linear index
- The argument ***size(X)*** can be replaced with ***size(****n_rows****,*** *n_cols****)*** or ***size(****n_rows****,*** *n_cols****,*** *n_slices****)***
- For the *matrix_of_subscripts* argument, the subscripts must be stored in each column of an *m x n* matrix of type umat; *m=2* for matrix subscripts, while *m=3* for cube subscripts
- A *std::logic_error* exception is thrown if a subscript is out of range
- Examples:

```
mat  M(4,5);
cube Q(4,5,6);

uword i = sub2ind( size(M), 2, 3 );
uword j = sub2ind( size(Q), 2, 3, 4 );
```

- See also:
  - size()
  - ind2sub()
  - element access

---

symmatu(A)
symmatu(A, do_conj)

symmatl(A)
symmatl(A, do_conj)

- *symmatu(A)*: generate symmetric matrix from square matrix *A*, by reflecting the upper triangle to the lower triangle
- *symmatl(A)*: generate symmetric matrix from square matrix *A*, by reflecting the lower triangle to the upper triangle
- If *A* is a complex matrix, the reflection uses the complex conjugate of the elements; to disable the complex conjugate, set *do_conj* to *false*
- If *A* is non-square, a *std::logic_error* exception is thrown

- Examples:

```
mat A = randu<mat>(5,5);

mat B = symmatu(A);
mat C = symmatl(A);
```

- See also:
  - diagmat()
  - trimatu() / trimatl()
  - .is_symmetric()
  - .is_hermitian()
  - Symmetric matrix in Wikipedia

---

trace(X)

- Sum of the elements on the main diagonal of matrix *X*
- If *X* is an expression, the function will try to use optimised expression evaluations to calculate only the diagonal elements
- Examples:

```
mat    A = randu<mat>(5,5);

double x = trace(A);
```

- See also:
  - accu()
  - as_scalar()
  - .diag()
  - diagvec()
  - sum()

---

trans(A)
strans(A)

- For real (non-complex) matrix:
  - *trans()* provides a transposed copy of the matrix
  - *strans()* not applicable
- For complex matrix:
  - *trans()* provides a Hermitian transpose (ie. the conjugate of the elements is taken during the transpose)
  - *strans()* provides a transposed copy without taking the conjugate of the elements
- Examples:

```
mat A = randu<mat>(5,10);
```

```
mat B = trans(A);
mat C = A.t();    // equivalent to trans(A), but more compact
```

- See also:
  - .t()
  - inplace_trans()

---

trapz(X, Y)
trapz(X, Y, dim)

trapz(Y)
trapz(Y, dim)

- Compute the trapezoidal integral of *Y* with respect to spacing in *X*, in each column (*dim=0*) or each row (*dim=1*) of *Y*
- *X* must be a vector; its length must equal either the number of rows in *Y* (when *dim=0*), or the number of columns in *Y* (when *dim=1*)
- If *X* is not specified, unit spacing is used
- The *dim* argument is optional; by default *dim=0*
- Examples:

```
vec X = linspace<vec>(0, datum::pi, 1000);
vec Y = sin(X);

mat Z = trapz(X,Y);
```

- See also:
  - sum()
  - diff()
  - linspace()
  - numerical integration in Wikipedia
  - numerical integration in MathWorld
  - trapezoidal rule in Wikipedia

---

trimatu(A)
trimatu(A, k)

trimatl(A)
trimatl(A, k)

- Create a new matrix by copying either the upper or lower triangular part from square matrix *A*, and setting the remaining elements to zero
  - *trimatu()* copies the upper triangular part
  - *trimatl()* copies the lower triangular part

- The argument *k* specifies the diagonal which inclusively delineates the boundary of the triangular part
  - for *k > 0*, the *k*-th super-diagonal is used (above main diagonal, towards top-right corner)
  - for *k < 0*, the *k*-th sub-diagonal is used (below main diagonal, towards bottom-left corner)
- The argument *k* is optional; by default the main diagonal is used (*k=0*)
- If *A* is non-square, a *std::logic_error* exception is thrown
- Examples:

```
mat A = randu<mat>(5,5);

mat U  = trimatu(A);
mat L  = trimatl(A);

mat UU = trimatu(A,  1);  // omit the main diagonal
mat LL = trimatl(A, -1);  // omit the main diagonal
```

- See also:
  - .is_trimatu() / .is_trimatl()
  - trimatu_ind() / trimatl_ind()
  - symmatu() / symmatl()
  - diagmat()
  - nonzeros()
  - Triangular matrix in MathWorld
  - Triangular matrix in Wikipedia

---

trimatu_ind(size(A))
trimatu_ind(size(A), k)

trimatl_ind(size(A))
trimatl_ind(size(A), k)

- Return a column vector containing the indices of elements that form the upper or lower triangle part of matrix *A*
  - *trimatu_ind()* refers to the upper triangular part
  - *trimatl_ind()* refers to the lower triangular part
- The output vector must have the type uvec (ie. the indices are stored as unsigned integers of type uword)
- The argument *k* specifies the diagonal which inclusively delineates the boundary of the triangular part
  - for *k > 0*, the *k*-th super-diagonal is used (above main diagonal, towards top-right corner)
  - for *k < 0*, the *k*-th sub-diagonal is used (below main diagonal, towards bottom-left corner)
- The argument *k* is optional; by default the main diagonal is used (*k=0*)
- The argument ***size(A)*** can be replaced with ***size(****n_rows, n_cols****)***
- Examples:

```
mat A = randu<mat>(5,5);

uvec upper_indices = trimatu_ind( size(A) );
uvec lower_indices = trimatl_ind( size(A) );
```

```
// extract upper/lower triangle into vector
vec upper_part = A(upper_indices);
vec lower_part = A(lower_indices);

// obtain indices without the main diagonal
uvec alt_upper_indices = trimatu_ind( size(A),  1);
uvec alt_lower_indices = trimatl_ind( size(A), -1);
```

- See also:
  - trimatu() / trimatl()
  - find()
  - submatrix views

---

unique(A)

- Return the unique elements of *A*, sorted in ascending order
- If *A* is a vector, the output is also a vector with the same orientation (row or column) as *A*; if *A* is a matrix, the output is always a column vector
- Examples:

```
mat X;
X << 1 << 2 << endr
  << 2 << 3 << endr;

mat Y = unique(X);
```

- See also:
  - find()
  - find_unique()
  - sort()
  - shuffle()
  - nonzeros()
  - intersect()

---

vectorise(X)
vectorise(X, dim)
vectorise(Q)

- Generate a flattened version of matrix *X* or cube *Q*
- The argument *dim* is optional; by default *dim=0* is used
- For *dim=0*, the elements are copied from *X* column-wise, resulting in a column vector; equivalent to concatenating all the columns of *X*
- For *dim=1*, the elements are copied from *X* row-wise, resulting in a row vector; equivalent to concatenating all the rows of *X*

- **Caveats:**
  - column-wise vectorisation is faster than row-wise vectorisation
  - for sparse matrices, row-wise vectorisation is not recommended

- Examples:

```
mat X = randu<mat>(4, 5);

vec v = vectorise(X);
```

- See also:
  - .as_col() / .as_row()
  - nonzeros()
  - reshape()
  - resize()
  - diagvec()
  - as_scalar()

---

miscellaneous element-wise functions:

exp	**exp2**	**exp10**	**trunc_exp**	**expm1**
log	**log2**	**log10**	**trunc_log**	**log1p**
pow	**square**	**sqrt**		
floor	**ceil**	**round**	**trunc**	
erf	**erfc**			
lgamma				
sign				

- Apply a function to each element

- Usage:
  - B = fn(A)
  - A and B must have the same matrix type or cube type, such as *mat* or *cube*
  - fn(A) is one of:

`exp(A)`	base-e exponential: e^x
`exp2(A)`	base-2 exponential: 2^x
`exp10(A)`	base-10 exponential: 10^x
`trunc_exp(A)`	base-e exponential, truncated to avoid infinity (only for *float* and *double* elements)
`expm1(A)`	compute `exp(A)-1` accurately for values of A close to zero (only for *float* and *double* elements)
`log(A)`	natural log: $log_e\ x$
`log2(A)`	base-2 log: $log_2\ x$
`log10(A)`	base-10 log: $log_{10}\ x$
`trunc_log(A)`	natural log, truncated to avoid ±infinity (only for *float* and *double* elements)
`log1p(A)`	compute `log(1+A)` accurately for values of A close to zero (only for *float* and *double* elements)
`pow(A, p)`	raise to the power of p: x^p
`square(A)`	square: x^2

`sqrt(A)`	square root: $x^{½}$
`floor(A)`	largest integral value that is not greater than the input value
`ceil(A)`	smallest integral value that is not less than the input value
`round(A)`	round to nearest integer, with halfway cases rounded away from zero
`trunc(A)`	round to nearest integer, towards zero
`erf(A)`	error function (only for *float* and *double* elements)
`erfc(A)`	complementary error function (only for *float* and *double* elements)
`lgamma(A)`	natural log of the gamma function (only for *float* and *double* elements)
`sign(A)`	signum function; for each element *a* in *A*, the corresponding element *b* in *B* is: $b = \begin{cases} -1 & \text{if } a < 0 \\ 0 & \text{if } a = 0 \\ +1 & \text{if } a > 0 \end{cases}$ if *a* is complex and non-zero, then *b* = *a* / abs(*a*)

- Examples:

```
mat A = randu<mat>(5,5);
mat B = exp(A);
```

- See also:
  - abs()
  - clamp()
  - conj()
  - imag() / real()
  - .transform() (apply user-defined function to each element)
  - expmat()
  - logmat()
  - sqrtmat()
  - powmat()
  - trigonometric functions
  - statistics functions
  - miscellaneous constants

---

trigonometric element-wise functions (cos, sin, tan, ...)

- For single argument functions, *B = trig_fn(A)*, where *trig_fn* is applied to each element in *A*, with *trig_fn* as one of:
  - *cos*, *acos*, *cosh*, *acosh*
  - *sin*, *asin*, *sinh*, *asinh*
  - *tan*, *atan*, *tanh*, *atanh*
  - *sinc*, defined as sinc(x) = sin(πx) / (πx) for x ≠ 0, and sinc(x) = 1 for x = 0

- For dual argument functions, apply the function to each tuple of two corresponding elements in *X* and *Y*:
  - *Z = atan2(Y, X)*
  - *Z = hypot(X, Y)*

- Examples:

```
mat X = randu<mat>(5,5);
mat Y = cos(X);
```

- See also:
  - miscellaneous element-wise functions
  - trigonometric functions in Wikipedia
  - atan2 function in Wikipedia
  - hypot function in Wikipedia
  - sinc function in Wikipedia

Decompositions, Factorisations, Inverses and Equation Solvers (Dense Matrices)

R = chol(X)
R = chol(X, layout)

chol(R, X)
chol(R, X, layout)

- Cholesky decomposition of matrix *X*
- Matrix *X* must be symmetric/hermitian and positive-definite
- By default, *R* is upper triangular, such that *R.t()*R = X*
- The argument *layout* is optional; *layout* is either `"upper"` or `"lower"`, which specifies whether *R* is upper or lower triangular
- If the decomposition fails:
  - *R = chol(X)* resets *R* and throws a *std::runtime_error* exception
  - *chol(R,X)* resets *R* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
mat X = randu<mat>(5,5);
mat Y = X.t()*X;

mat R1 = chol(Y);
mat R2 = chol(Y, "lower");
```

- See also:
  - sqrtmat()
  - lu()
  - qr()
  - .is_sympd()
  - Cholesky decomposition in MathWorld
  - Cholesky decomposition in Wikipedia

---

vec eigval = eig_sym(X)

eig_sym(eigval, X)

eig_sym(eigval, eigvec, X)
eig_sym(eigval, eigvec, X, method)

- Eigen decomposition of **dense** symmetric/hermitian matrix *X*
- The eigenvalues and corresponding eigenvectors are stored in *eigval* and *eigvec*, respectively
- The eigenvalues are in ascending order
- The eigenvectors are stored as column vectors
- If *X* is not square sized, a *std::logic_error* exception is thrown
- The *method* argument is optional; *method* is either `"dc"` or `"std"`
  - `"dc"` indicates divide-and-conquer method (default setting)
  - `"std"` indicates standard method
  - the divide-and-conquer method provides slightly different results than the standard method, but is considerably faster for large matrices
- If the decomposition fails:
  - *eigval = eig_sym(X)* resets *eigval* and throws a *std::runtime_error* exception
  - *eig_sym(eigval,X)* resets *eigval* and returns a bool set to *false* (exception is not thrown)
  - *eig_sym(eigval,eigvec,X)* resets *eigval* & *eigvec* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
// for matrices with real elements

mat A = randu<mat>(50,50);
mat B = A.t()*A;  // generate a symmetric matrix

vec eigval;
mat eigvec;

eig_sym(eigval, eigvec, B);


// for matrices with complex elements

cx_mat C = randu<cx_mat>(50,50);
cx_mat D = C.t()*C;

   vec eigval2;
cx_mat eigvec2;

eig_sym(eigval2, eigvec2, D);
```

- See also:
  - eig_gen()
  - eig_pair()
  - svd()
  - svd_econ()
  - princomp()

- eigs_sym()
- .is_symmetric()
- .is_hermitian()
- eigen decomposition in MathWorld
- eigenvalues & eigenvectors in Wikipedia
- divide & conquer eigenvalue algorithm in Wikipedia

---

cx_vec eigval = eig_gen(X)
cx_vec eigval = eig_gen(X, bal)

eig_gen(eigval, X)
eig_gen(eigval, X, bal)

eig_gen(eigval, eigvec, X)
eig_gen(eigval, eigvec, X, bal)

eig_gen(eigval, leigvec, reigvec, X)
eig_gen(eigval, leigvec, reigvec, X, bal)

- Eigen decomposition of **dense** general (non-symmetric/non-hermitian) square matrix *X*
- The eigenvalues and corresponding right eigenvectors are stored in *eigval* and *eigvec*, respectively
- If both left and right eigenvectors are requested they are stored in *leigvec* and *reigvec*, respectively
- The eigenvectors are stored as column vectors
- The *bal* argument is optional; *bal* is one of:
  `"balance"` ↦ diagonally scale and permute *X* to improve conditioning of the eigenvalues
  `"nobalance"` ↦ do not balance *X*; this is the **default operation**
- If *X* is not square sized, a *std::logic_error* exception is thrown
- If the decomposition fails:
  - *eigval = eig_gen(X)* resets *eigval* and throws a *std::runtime_error* exception
  - *eig_gen(eigval,X)* resets *eigval* and returns a bool set to *false* (exception is not thrown)
  - *eig_gen(eigval,eigvec,X)* resets *eigval* & *eigvec* and returns a bool set to *false* (exception is not thrown)
  - *eig_gen(eigval,leigvec,reigvec,X)* resets *eigval*, *leigvec* & *reigvec* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
mat A = randu<mat>(10,10);

cx_vec eigval;
cx_mat eigvec;

eig_gen(eigval, eigvec, A);
eig_gen(eigval, eigvec, A, "balance");
```

- See also:

- eig_pair()
- eig_sym()
- svd()
- svd_econ()
- schur()
- eigs_gen()
- eigen decomposition in MathWorld
- eigenvalues & eigenvectors in Wikipedia

---

cx_vec eigval = eig_pair(A, B)

eig_pair(eigval, A, B)

eig_pair(eigval, eigvec, A, B)

eig_pair(eigval, leigvec, reigvec, A, B)

- Eigen decomposition for pair of general **dense** square matrices *A* and *B* of the same size, such that *A*eigvec = B*eigvec*diagmat(eigval)*
- The eigenvalues and corresponding right eigenvectors are stored in *eigval* and *eigvec*, respectively
- If both left and right eigenvectors are requested they are stored in *leigvec* and *reigvec*, respectively
- The eigenvectors are stored as column vectors
- If *A* or *B* is not square sized, a *std::logic_error* exception is thrown
- If the decomposition fails:
  - *eigval = eig_pair(A,B)* resets *eigval* and throws a *std::runtime_error* exception
  - *eig_pair(eigval,A,B)* resets *eigval* and returns a bool set to *false* (exception is not thrown)
  - *eig_pair(eigval,eigvec,A,B)* resets *eigval* & *eigvec* and returns a bool set to *false* (exception is not thrown)
  - *eig_pair(eigval,leigvec,reigvec,A,B)* resets *eigval*, *leigvec* & *reigvec* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
mat A = randu<mat>(10,10);
mat B = randu<mat>(10,10);

cx_vec eigval;
cx_mat eigvec;

eig_pair(eigval, eigvec, A, B);
```

- See also:
  - eig_gen()
  - eig_sym()
  - qz()
  - eigen decomposition in MathWorld
  - eigenvalues & eigenvectors in Wikipedia

H = hess(X)

hess(H, X)

hess(U, H, X)

- Upper Hessenberg decomposition of square matrix *X*, such that *X = U*H*U.t()*
- *U* is a unitary matrix containing the Hessenberg vectors
- *H* is a square matrix known as the upper Hessenberg matrix, with elements below the first subdiagonal set to zero
- If *X* is not square sized, a *std::logic_error* exception is thrown
- If the decomposition fails:
  - *H = hess(X)* resets *H* and throws a *std::runtime_error* exception
  - *hess(H,X)* resets *H* and returns a bool set to *false* (exception is not thrown)
  - *hess(U,H,X)* resets *U* & *H* and returns a bool set to *false* (exception is not thrown)
- **Caveat:** in general, upper Hessenberg decomposition is not unique
- Examples:

```
mat X(20,20, fill::randu);

mat U;
mat H;

hess(U, H, X);
```

- See also:
  - qz()
  - schur()
  - Hessenberg decomposition in MathWorld
  - Hessenberg matrix in Wikipedia

B = inv(A)
inv(B, A)

- Inverse of general square matrix *A*
- If *A* is not square sized, a *std::logic_error* exception is thrown
- If *A* appears to be singular:
  - *B = inv(A)* resets *B* and throws a *std::runtime_error* exception
  - *inv(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)
- **Caveats:**
  - if matrix *A* is know to be symmetric positive definite, using inv_sympd() is faster

  - if matrix *A* is know to be diagonal, use *inv(diagmat(A))*
  - if matrix *A* is know to be triangular, use *inv(trimatu(A))* or *inv(trimatl(A))*
  - to solve a system of linear equations, such as *Z = inv(X)*Y*, using solve() can be faster and/or more accurate

- Examples:

```
mat A = randu<mat>(5,5);

mat B = inv(A);
```

- See also:
  - .i()
  - inv_sympd()
  - rcond()
  - pinv()
  - solve()
  - spsolve()
  - diagmat()
  - trimatu() / trimatl()
  - powmat()
  - matrix inverse in MathWorld
  - invertible matrix in Wikipedia

B = inv_sympd(A)
inv_sympd(B, A)

- Inverse of symmetric/hermitian positive definite matrix *A*

- If *A* is not square sized, a *std::logic_error* exception is thrown

- If *A* appears to be singular or not positive definite:
  - *B = inv_sympd(A)* resets *B* and throws a *std::runtime_error* exception
  - *inv_sympd(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)

- **Caveat:** to solve a system of linear equations, such as *Z = inv(X)*Y*, using solve() can be faster and/or more accurate

- Examples:

```
mat A = randu<mat>(5,5);
mat B = A.t() * A;
mat C = inv_sympd(B);
```

- See also:
  - inv()
  - rcond()
  - pinv()
  - solve()
  - eig_sym()
  - .is_sympd()

- matrix inverse in MathWorld
- invertible matrix in Wikipedia
- positive definite matrix in MathWorld
- positive definite matrix in Wikipedia

---

lu(L, U, P, X)
lu(L, U, X)

- Lower-upper decomposition (with partial pivoting) of matrix *X*
- The first form provides a lower-triangular matrix *L*, an upper-triangular matrix *U*, and a permutation matrix *P*, such that *P.t()*L*U = X*
- The second form provides permuted *L* and *U*, such that *L*U = X*; note that in this case *L* is generally not lower-triangular
- If the decomposition fails:
  - *lu(L,U,P,X)* resets *L*, *U*, *P* and returns a bool set to *false* (exception is not thrown)
  - *lu(L,U,X)* resets *L*, *U* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
mat A = randu<mat>(5,5);

mat L, U, P;

lu(L, U, P, A);

mat B = P.t()*L*U;
```

- See also:
  - chol()
  - LU decomposition in Wikipedia
  - LU decomposition in MathWorld

B = null(A)
B = null(A, tolerance)

null(B, A)
null(B, A, tolerance)

- Find the orthonormal basis of the null space of matrix *A*
- The dimension of the range space is the number of singular values of *A* not greater than *tolerance*
- The *tolerance* argument is optional; by default *tolerance* is *max(m,n)*max_sv*datum::eps*, where:
  - *m* = number of rows and *n* = number of columns in *A*
  - *max_sv* = maximal singular value of *A*
  - *datum::eps* = difference between 1 and the least value greater than 1 that is representable

- The computation is based on singular value decomposition; if the decomposition fails:
  - *B = null(A)* resets *B* and throws a *std::runtime_error* exception
  - *null(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)

- Examples:

```
mat A = randu<mat>(5,6);

A.row(0).zeros();
A.col(0).zeros();

mat B = null(A);
```

- See also:
  - orth()
  - qr()
  - svd()
  - rank()
  - datum::eps
  - Orthonormal basis in Wikipedia

B = orth(A)
B = orth(A, tolerance)

orth(B, A)
orth(B, A, tolerance)

- Find the orthonormal basis of the range space of matrix *A*, so that *B.t()*B ≈ eye(r,r)*, where *r = rank(A)*

- The dimension of the range space is the number of singular values of *A* greater than *tolerance*

- The *tolerance* argument is optional; by default *tolerance* is *max(m,n)*max_sv*datum::eps*, where:
  - *m* = number of rows and *n* = number of columns in *A*
  - *max_sv* = maximal singular value of *A*
  - *datum::eps* = difference between 1 and the least value greater than 1 that is representable

- The computation is based on singular value decomposition; if the decomposition fails:
  - *B = orth(A)* resets *B* and throws a *std::runtime_error* exception
  - *orth(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)

- Examples:

```
mat A = randu<mat>(5,6);

mat B = orth(A);
```

- See also:
  - null()
  - qr()
  - svd()
  - rank()

- datum::eps
- Orthonormal basis in Wikipedia

B = pinv(A)
B = pinv(A, tolerance)
B = pinv(A, tolerance, method)

pinv(B, A)
pinv(B, A, tolerance)
pinv(B, A, tolerance, method)

- Moore-Penrose pseudo-inverse of matrix *A*
- The computation is based on singular value decomposition
- The *tolerance* argument is optional
- The default tolerance is *max(m,n)*max_sv*datum::eps*, where:
  - *m* = number of rows and *n* = number of columns in *A*
  - *max_sv* = maximal singular value of *A*
  - *datum::eps* = difference between 1 and the least value greater than 1 that is representable
- Any singular values less than *tolerance* are treated as zero
- The *method* argument is optional; *method* is either `"dc"` or `"std"`
  - `"dc"` indicates divide-and-conquer method (default setting)
  - `"std"` indicates standard method
  - the divide-and-conquer method provides slightly different results than the standard method, but is considerably faster for large matrices
- If the decomposition fails:
  - *B = pinv(A)* resets *B* and throws a *std::runtime_error* exception
  - *pinv(B,A)* resets *B* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
mat A = randu<mat>(4,5);

mat B = pinv(A);        // use default tolerance

mat C = pinv(A, 0.01);  // set tolerance to 0.01
```

- See also:
  - .i()
  - inv()
  - solve()
  - spsolve()
  - datum::eps
  - Pseudoinverse in MathWorld
  - Moore-Penrose Matrix Inverse in MathWorld
  - Moore-Penrose pseudoinverse in Wikipedia

---

qr(Q, R, X) (form 1)
qr(Q, R, P, X, "vector") (form 2)
qr(Q, R, P, X, "matrix") (form 3)

- Decomposition of *X* into an orthogonal matrix *Q* and a right triangular matrix *R*, with an optional permutation matrix/vector *P*
  - form 1: decomposition has the form *Q*R = X*
  - form 2: *P* is permutation vector with type uvec; decomposition has the form *Q*R = X.cols(P)*
  - form 3: *P* is permutation matrix with type umat; decomposition has the form *Q*R = X*P*
- If *P* is specified, a column pivoting decomposition is used; the diagonal entries of *R* are ordered from largest to smallest magnitude
- If the decomposition fails, *Q*, *R* and *P* are reset and the function returns a bool set to *false* (exception is not thrown)
- Examples:

```
mat X = randu<mat>(5,5);

mat Q;
mat R;

qr(Q, R, X);

uvec P_vec;
umat P_mat;

qr(Q, R, P_vec, X, "vector");
qr(Q, R, P_mat, X, "matrix");
```

- See also:
  - qr_econ()
  - chol()
  - orth()
  - orthogonal matrix in Wikipedia
  - QR decomposition in Wikipedia
  - QR decomposition in MathWorld

---

qr_econ(Q, R, X)

- Economical decomposition of *X* (with size *m* x *n*) into an orthogonal matrix *Q* and a right triangular matrix *R*, such that *Q*R = X*
- If *m* > *n*, only the first *n* rows of *R* and the first *n* columns of *Q* are calculated (ie. the zero rows of *R* and the corresponding columns of *Q* are omitted)
- If the decomposition fails, *Q* and *R* are reset and the function returns a bool set to *false* (exception is not thrown)

- Examples:

```
mat X = randu<mat>(6,5);

mat Q;
mat R;

qr_econ(Q, R, X);
```

- See also:
  - qr()
  - orthogonal matrix in Wikipedia
  - QR decomposition in Wikipedia
  - QR decomposition in Octave
  - QR decomposition in MathWorld

---

qz(AA, BB, Q, Z, A, B)
qz(AA, BB, Q, Z, A, B, select)

- Generalised Schur decomposition for pair of general square matrices *A* and *B* of the same size, such that *A = Q.t()*AA*Z.t()* and *B = Q.t()*BB*Z.t()*

- The *select* argument is optional and specifies the ordering of the top left of the Schur form; it is one of the following:

`"none"`	no ordering (default operation)
`"lhp"`	left-half-plane: eigenvalues with real part < 0
`"rhp"`	right-half-plane: eigenvalues with real part > 0
`"iuc"`	inside-unit-circle: eigenvalues with absolute value < 1
`"ouc"`	outside-unit-circle: eigenvalues with absolute value > 1

- The left and right Schur vectors are stored in *Q* and *Z*, respectively

- In the complex-valued problem, the generalised eigenvalues are found in *diagvec(AA) / diagvec(BB)*

- If *A* or *B* is not square sized, a *std::logic_error* exception is thrown

- If the decomposition fails, *AA*, *BB*, *Q* and *Z* are reset, and the function returns a bool set to *false* (exception is not thrown)

- Examples:

```
mat A = randu<mat>(10,10);
mat B = randu<mat>(10,10);

mat AA;
mat BB;
mat Q;
mat Z;

qz(AA, BB, Q, Z, A, B);
```

- See also:

- hess()
- schur()
- eig_pair()
- generalised Schur decomposition in Wikipedia

S = schur(X)

schur(S, X)

schur(U, S, X)

- Schur decomposition of square matrix *X*, such that *X = U*S*U.t()*
- *U* is a unitary matrix containing the Schur vectors
- *S* is an upper triangular matrix, called the Schur form of *X*
- If *X* is not square sized, a *std::logic_error* exception is thrown
- If the decomposition fails:
  - *S = schur(X)* resets *S* and throws a *std::runtime_error* exception
  - *schur(S,X)* resets *S* and returns a bool set to *false* (exception is not thrown)
  - *schur(U,S,X)* resets *U* & *S* and returns a bool set to *false* (exception is not thrown)
- **Caveat:** in general, Schur decomposition is not unique
- Examples:

```
mat X(20,20, fill::randu);

mat U;
mat S;

schur(U, S, X);
```

- See also:
  - hess()
  - qz()
  - eig_gen()
  - Schur decomposition in MathWorld
  - Schur decomposition in Wikipedia

X = solve(A, B)
X = solve(A, B, settings)

solve(X, A, B)
solve(X, A, B, settings)

- Solve a **dense** system of linear equations, *A*X = B*, where *X* is unknown; similar functionality to the \ operator in

Matlab/Octave, ie. *X = A \ B*

- By default, matrix *A* is analysed to automatically determine whether it is a general matrix, band matrix, diagonal matrix, or symmetric/hermitian positive definite (SPD) matrix; based on the detected matrix structure, a specialised solver is used for faster execution
- If *A* is known to be a triangular matrix, the solution can be computed faster by explicitly indicating that *A* is triangular through trimatu() or trimatl(); see examples below
- *A* can be square (critically determined system), or non-square (under/over-determined system)
- *B* can be a vector or matrix
- The number of rows in *A* and *B* must be the same
- The *settings* argument is optional; it is one of the following, or a combination thereof:

`solve_opts::fast`	fast mode: disable determining solution quality via rcond, disable iterative refinement, disable equilibration
`solve_opts::refine`	apply iterative refinement to improve solution quality (matrix *A* must be square)
`solve_opts::equilibrate`	equilibrate the system before solving (matrix *A* must be square)
`solve_opts::likely_sympd`	indicate that matrix *A* is likely symmetric/hermitian positive definite
`solve_opts::allow_ugly`	keep solutions of systems that are singular to working precision
`solve_opts::no_approx`	do not find approximate solutions for rank deficient systems
`solve_opts::no_band`	do not use specialised solver for band matrices or diagonal matrices
`solve_opts::no_trimat`	do not use specialised solver for triangular matrices
`solve_opts::no_sympd`	do not use specialised solver for symmetric/hermitian positive definite matrices

the above settings can be combined using the + operator; for example:
`solve_opts::fast + solve_opts::no_approx`

- **Caveat:** using *`solve_opts::fast`* will speed up finding the solution, but for poorly conditioned systems the solution may have lower quality
- **Caveat:** not all SPD matrices are automatically detected; to skip the analysis step and directly indicate that matrix *A* is likely SPD, use `solve_opts::likely_sympd`
- If no solution is found:
  - *X = solve(A,B)* resets *X* and throws a *std::runtime_error* exception
  - *solve(X,A,B)* resets *X* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
mat A = randu<mat>(5,5);
vec b = randu<vec>(5);
mat B = randu<mat>(5,5);

vec x1 = solve(A, b);

vec x2;
bool status = solve(x2, A, b);

mat X1 = solve(A, B);

mat X2 = solve(A, B, solve_opts::fast);  // enable fast mode
```

```
mat X3 = solve(trimatu(A), B);  // indicate that A is triangular
```

- See also:
  - inv()
  - pinv()
  - rcond()
  - roots()
  - syl()
  - spsolve()
  - linear system of equations in MathWorld
  - system of linear equations in Wikipedia
  - band matrix in Wikipedia
  - definiteness of a matrix in Wikipedia
  - positive definite matrix in MathWorld
  - iterative refinement

---

vec s = svd(X)

svd(vec s, X)

svd(mat U, vec s, mat V, mat X)
svd(mat U, vec s, mat V, mat X, method)

svd(cx_mat U, vec s, cx_mat V, cx_mat X)
svd(cx_mat U, vec s, cx_mat V, cx_mat X, method)

- Singular value decomposition of **dense** matrix *X*
- If *X* is square, it can be reconstructed using *X = U*diagmat(s)*V.t()*
- The singular values are in descending order
- The *method* argument is optional; *method* is either `"dc"` or `"std"`
  - `"dc"` indicates divide-and-conquer method (default setting)
  - `"std"` indicates standard method
  - the divide-and-conquer method provides slightly different results than the standard method, but is considerably faster for large matrices
- If the decomposition fails, the output objects are reset and:
  - *s = svd(X)* resets *s* and throws a *std::runtime_error* exception
  - *svd(s,X)* resets *s* and returns a bool set to *false* (exception is not thrown)
  - *svd(U,s,V,X)* resets *U*, *s*, *V* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
mat X = randu<mat>(5,5);

mat U;
vec s;
mat V;
```

```
svd(U,s,V,X);
```

- See also:
  - svd_econ()
  - eig_gen()
  - eig_sym()
  - princomp()
  - svds()
  - singular value decomposition in Wikipedia
  - singular value decomposition in MathWorld

---

svd_econ(mat U, vec s, mat V, mat X)
svd_econ(mat U, vec s, mat V, mat X, mode)
svd_econ(mat U, vec s, mat V, mat X, mode, method)

svd_econ(cx_mat U, vec s, cx_mat V, cx_mat X)
svd_econ(cx_mat U, vec s, cx_mat V, cx_mat X, mode)
svd_econ(cx_mat U, vec s, cx_mat V, cx_mat X, mode, method)

- Economical singular value decomposition of **dense** matrix *X*
- The singular values are in descending order
- The *mode* argument is optional; *mode* is one of:
  `"both"` = compute both left and right singular vectors (default operation)
  `"left"` = compute only left singular vectors
  `"right"` = compute only right singular vectors
- The *method* argument is optional; *method* is either `"dc"` or `"std"`
  - `"dc"` indicates divide-and-conquer method (default setting)
  - `"std"` indicates standard method
  - the divide-and-conquer method provides slightly different results than the standard method, but is considerably faster for large matrices
- If the decomposition fails, *U*, *s*, *V* are reset and a bool set to *false* is returned (exception is not thrown)
- Examples:

```
mat X = randu<mat>(4,5);

mat U;
vec s;
mat V;

svd_econ(U, s, V, X);
```

- See also:
  - svd()
  - eig_gen()
  - eig_sym()
  - princomp()

- svds()
- singular value decomposition in Wikipedia
- singular value decomposition in MathWorld

X = syl(A, B, C)
syl(X, A, B, C)

- Solve the Sylvester equation, ie. *AX + XB + C = 0*, where *X* is unknown
- Matrices *A*, *B* and *C* must be square sized
- If no solution is found:
  - *syl(A,B,C)* resets *X* and throws a *std::runtime_error* exception
  - *syl(X,A,B,C)* resets *X* and returns a bool set to *false* (exception is not thrown)
- Examples:

```
mat A = randu<mat>(5,5);
mat B = randu<mat>(5,5);
mat C = randu<mat>(5,5);

mat X1 = syl(A, B, C);

mat X2;
syl(X2, A, B, C);
```

- See also:
  - solve()
  - Sylvester equation in Wikipedia

Decompositions, Factorisations and Equation Solvers (Sparse Matrices)

---

vec eigval = eigs_sym(X, k)
vec eigval = eigs_sym(X, k, form)
vec eigval = eigs_sym(X, k, form, tol)

eigs_sym(eigval, X, k)
eigs_sym(eigval, X, k, form)
eigs_sym(eigval, X, k, form, tol)

eigs_sym(eigval, eigvec, X, k)
eigs_sym(eigval, eigvec, X, k, form)
eigs_sym(eigval, eigvec, X, k, form, tol)

- Obtain a limited number of eigenvalues and eigenvectors of **sparse** symmetric real matrix *X*

- *k* specifies the number of eigenvalues and eigenvectors
- The argument *form* is optional; *form* is one of:
  - `"lm"` = obtain eigenvalues with largest magnitude (default operation)
  - `"sm"` = obtain eigenvalues with smallest magnitude (see caveat below)
  - `"la"` = obtain eigenvalues with largest algebraic value
  - `"sa"` = obtain eigenvalues with smallest algebraic value
- The argument *tol* is optional; it specifies the tolerance for convergence
- The eigenvalues and corresponding eigenvectors are stored in *eigval* and *eigvec*, respectively
- If *X* is not square sized, a *std::logic_error* exception is thrown
- If the decomposition fails:
  - *eigval = eigs_sym(X,k)* resets *eigval* and throws a *std::runtime_error* exception
  - *eigs_sym(eigval,X,k)* resets *eigval* and returns a bool set to *false* (exception is not thrown)
  - *eigs_sym(eigval,eigvec,X,k)* resets *eigval* & *eigvec* and returns a bool set to *false* (exception is not thrown)
- **Caveats:**
  - the number of obtained eigenvalues/eigenvectors may be lower than requested, depending on the given data
  - it's more difficult to compute the smallest eigenvalues than the largest eigenvalues; if the decomposition fails, try increasing *k* (number of eigenvalues) and/or the tolerance
- Examples:

```
// generate sparse matrix
sp_mat A = sprandu<sp_mat>(1000, 1000, 0.1);
sp_mat B = A.t()*A;

vec eigval;
mat eigvec;

eigs_sym(eigval, eigvec, B, 5);  // find 5 eigenvalues/eigenvectors
```

- See also:
  - eigs_gen()
  - eig_sym()
  - svds()
  - .is_symmetric()
  - eigen decomposition in MathWorld
  - eigenvalues & eigenvectors in Wikipedia

---

cx_vec eigval = eigs_gen(X, k)
cx_vec eigval = eigs_gen(X, k, form)
cx_vec eigval = eigs_gen(X, k, form, tol)

eigs_gen(eigval, X, k)
eigs_gen(eigval, X, k, form)
eigs_gen(eigval, X, k, form, tol)

eigs_gen(eigval, eigvec, X, k)
eigs_gen(eigval, eigvec, X, k, form)
eigs_gen(eigval, eigvec, X, k, form, tol)

- Obtain a limited number of eigenvalues and eigenvectors of **sparse** general (non-symmetric/non-hermitian) square matrix *X*
- *k* specifies the number of eigenvalues and eigenvectors
- The argument *form* is optional; *form* is one of:
  - `"lm"` = obtain eigenvalues with largest magnitude (default operation)
  - `"sm"` = obtain eigenvalues with smallest magnitude (see caveat below)
  - `"lr"` = obtain eigenvalues with largest real part
  - `"sr"` = obtain eigenvalues with smallest real part
  - `"li"` = obtain eigenvalues with largest imaginary part
  - `"si"` = obtain eigenvalues with smallest imaginary part
- The argument *tol* is optional; it specifies the tolerance for convergence
- The eigenvalues and corresponding eigenvectors are stored in *eigval* and *eigvec*, respectively
- If *X* is not square sized, a *std::logic_error* exception is thrown
- If the decomposition fails:
  - *eigval = eigs_gen(X,k)* resets *eigval* and throws a *std::runtime_error* exception
  - *eigs_gen(eigval,X,k)* resets *eigval* and returns a bool set to *false* (exception is not thrown)
  - *eigs_gen(eigval,eigvec,X,k)* resets *eigval* & *eigvec* and returns a bool set to *false* (exception is not thrown)
- **Caveats:**
  - the number of obtained eigenvalues/eigenvectors may be lower than requested, depending on the given data
  - it's more difficult to compute the smallest eigenvalues than the largest eigenvalues; if the decomposition fails, try increasing *k* (number of eigenvalues) and/or the tolerance
- Examples:

```
// generate sparse matrix
sp_mat A = sprandu<sp_mat>(1000, 1000, 0.1);

cx_vec eigval;
cx_mat eigvec;

eigs_gen(eigval, eigvec, A, 5);  // find 5 eigenvalues/eigenvectors
```

- See also:
  - eigs_sym()
  - eig_gen()
  - svds()
  - eigen decomposition in MathWorld
  - eigenvalues & eigenvectors in Wikipedia

X = spsolve(A, B)
X = spsolve(A, B, solver)
X = spsolve(A, B, solver, opts)

spsolve(X, A, B)
spsolve(X, A, B, solver)
spsolve(X, A, B, solver, opts)

- Solve a **sparse** system of linear equations, *A*X = B*, where *A* is a sparse matrix, *B* is a dense matrix or vector, and *X* is unknown
- The number of rows in *A* and *B* must be the same
- If no solution is found:
  - *X = spsolve(A, B)* resets *X* and throws a *std::runtime_error* exception
  - *spsolve(X, A, B)* resets *X* and returns a bool set to *false* (no exception is thrown)
- The *solver* argument is optional; *solver* is either `"superlu"` or `"lapack"`; by default `"superlu"` is used
  - For `"superlu"`, *ARMA_USE_SUPERLU* must be enabled in config.hpp
  - For `"lapack"`, sparse matrix *A* is converted to a dense matrix before using the LAPACK solver; this considerably increases memory usage
- **Notes**:
  - The SuperLU solver is mainly useful for very large and/or very sparse matrices
  - If you have sufficient amount of memory to store a dense version of matrix *A*, the LAPACK solver can be faster
- The *opts* argument is optional and applicable to the SuperLU solver;
  opts is an instance of the *superlu_opts* structure:

```
struct superlu_opts
  {
  bool             allow_ugly;   // default: false
  bool             equilibrate;  // default: false
  bool             symmetric;    // default: false
  double           pivot_thresh; // default: 1.0
  permutation_type permutation;  // default: superlu_opts::COLAMD
  refine_type      refine;       // default: superlu_opts::REF_NONE
  };
```

  - *allow_ugly* is either *true* or *false*; indicates whether to keep solutions of systems singular to working precision
  - *equilibrate* is either *true* or *false*; indicates whether to equilibrate the system (scale the rows and columns of *A* to have unit norm)
  - *symmetric* is either *true* or *false*; indicates whether to use SuperLU symmetric mode, which gives preference to diagonal pivots
  - *pivot_threshold* is in the range [0.0, 1.0], used for determining whether a diagonal entry is an acceptable pivot (details in SuperLU documentation)
  - *permutation* specifies the type of column permutation; it is one of:
    `superlu_opts::NATURAL` natural ordering

`superlu_opts::MMD_ATA`	minimum degree ordering on structure of `A.t() * A`
`superlu_opts::MMD_AT_PLUS_A`	minimum degree ordering on structure of `A.t() + A`
`superlu_opts::COLAMD`	approximate minimum degree column ordering

- *refine* specifies the type of iterative refinement; it is one of:

`superlu_opts::REF_NONE`	no refinement
`superlu_opts::REF_SINGLE`	iterative refinement in single precision
`superlu_opts::REF_DOUBLE`	iterative refinement in double precision
`superlu_opts::REF_EXTRA`	iterative refinement in extra precision

- Examples:

```
sp_mat A = sprandu<sp_mat>(1000, 1000, 0.1);

vec b = randu<vec>(1000);
mat B = randu<mat>(1000, 5);

vec x = spsolve(A, b);  // solve one system
mat X = spsolve(A, B);  // solve several systems

bool status = spsolve(x, A, b);  // use default solver
if(status == false)  { cout << "no solution" << endl; }

spsolve(x, A, b, "lapack" );  // use LAPACK  solver
spsolve(x, A, b, "superlu");  // use SuperLU solver

superlu_opts opts;

opts.allow_ugly  = true;
opts.equilibrate = true;

spsolve(x, A, b, "superlu", opts);
```

- See also:
  - solve()
  - SuperLU home page
  - linear system of equations in MathWorld
  - system of linear equations in Wikipedia

---

vec s = svds(X, k)
vec s = svds(X, k, tol)

svds(vec s, X, k)
svds(vec s, X, k, tol)

svds(mat U, vec s, mat V, sp_mat X, k)
svds(mat U, vec s, mat V, sp_mat X, k, tol)

svds(cx_mat U, vec s, cx_mat V, sp_cx_mat X, k)
svds(cx_mat U, vec s, cx_mat V, sp_cx_mat X, k, tol)

- Obtain a limited number of singular values and singular vectors (truncated SVD) of **sparse** matrix *X*

- The singular values and vectors are calculated via sparse eigen decomposition of:
$$\begin{bmatrix} \texttt{zeros(X.n\_rows, X.n\_rows)} & \texttt{X} \\ \texttt{X.t()} & \texttt{zeros(X.n\_cols, X.n\_cols)} \end{bmatrix}$$
- *k* specifies the number of singular values and singular vectors
- The singular values are in descending order
- The argument *tol* is optional; it specifies the tolerance for convergence; it is passed as *(tol ÷ √2)* to eigs_sym()
- If the decomposition fails, the output objects are reset and:
  - *s = svds(X,k)* resets *s* and throws a *std::runtime_error* exception
  - *svds(s,X,k)* resets *s* and returns a bool set to *false* (exception is not thrown)
  - *svds(U,s,V,X,k)* resets *U*, *s*, *V* and returns a bool set to *false* (exception is not thrown)
- **Caveats:**
  - *svds()* is intended only for finding a few singular values from a large sparse matrix; to find all singular values, use svd() instead
  - depending on the given matrix, *svds()* may find fewer singular values than specified
- Examples:

```
sp_mat X = sprandu<sp_mat>(100, 200, 0.1);

mat U;
vec s;
mat V;

svds(U, s, V, X, 10);
```

- See also:
  - eigs_gen()
  - eigs_sym()
  - svd()
  - singular value decomposition in Wikipedia
  - singular value decomposition in MathWorld

Signal & Image Processing

---

conv(A, B)
conv(A, B, shape)

- 1D convolution of vectors *A* and *B*
- The orientation of the result vector is the same as the orientation of *A* (ie. either column or row vector)
- The *shape* argument is optional; it is one of:

`"full"` = return the full convolution (**default setting**), with the size equal to *A.n_elem + B.n_elem - 1*

`"same"` = return the central part of the convolution, with the same size as vector *A*

- The convolution operation is also equivalent to FIR filtering
- Examples:

```
vec A(256, fill::randu);

vec B(16, fill::randu);

vec C = conv(A, B);

vec D = conv(A, B, "same");
```

- See also:
  - conv2()
  - fft()
  - cor()
  - interp1()
  - Convolution in MathWorld
  - Convolution in Wikipedia
  - FIR filter in Wikipedia

---

conv2(A, B)
conv2(A, B, shape)

- 2D convolution of matrices *A* and *B*
- The *shape* argument is optional; it is one of:
  `"full"` = return the full convolution (**default setting**), with the size equal to *size(A) + size(B) - 1*
  `"same"` = return the central part of the convolution, with the same size as matrix *A*
- The implementation of 2D convolution in this version is preliminary; it is not yet fully optimised
- Examples:

```
mat A(256, 256, fill::randu);

mat B(16, 16, fill::randu);

mat C = conv2(A, B);

mat D = conv2(A, B, "same");
```

- See also:
  - conv()
  - fft2()
  - interp2()
  - Convolution in MathWorld
  - Convolution in Wikipedia
  - Kernel (image processing) in Wikipedia

cx_mat Y = fft(X)
cx_mat Y = fft(X, n)

cx_mat Z = ifft(cx_mat Y)
cx_mat Z = ifft(cx_mat Y, n)

- *fft():* fast Fourier transform of a vector or matrix (real or complex)
- *ifft():* inverse fast Fourier transform of a vector or matrix (complex only)
- If given a matrix, the transform is done on each column vector of the matrix
- The optional *n* argument specifies the transform length:
  - if *n* is larger than the length of the input vector, a zero-padded version of the vector is used
  - if *n* is smaller than the length of the input vector, only the first *n* elements of the vector are used
- If *n* is not specified, the transform length is the same as the length of the input vector
- **Caveat:** the transform is fastest when the transform length is a power of 2, eg. 64, 128, 256, 512, 1024, ...
- The implementation of the transform in this version is preliminary; it is not yet fully optimised
- Examples:

```
   vec X = randu<vec>(100);
cx_vec Y = fft(X, 128);
```

- See also:
  - fft2()
  - conv()
  - real()
  - fast Fourier transform in MathWorld
  - fast Fourier transform in Wikipedia

---

cx_mat Y = fft2(X)
cx_mat Y = fft2(X, n_rows, n_cols)

cx_mat Z = ifft2(cx_mat Y)
cx_mat Z = ifft2(cx_mat Y, n_rows, n_cols)

- *fft2():* 2D fast Fourier transform of a matrix (real or complex)
- *ifft2():* 2D inverse fast Fourier transform of a matrix (complex only)
- The optional arguments *n_rows* and *n_cols* specify the size of the transform; a truncated and/or zero-padded version of the input matrix is used
- **Caveat:** the transform is fastest when both *n_rows* and *n_cols* are a power of 2, eg. 64, 128, 256, 512, 1024, ...

- The implementation of the transform in this version is preliminary; it is not yet fully optimised
- Examples:

```
   mat A = randu<mat>(100,100);
cx_mat B = fft2(A);
cx_mat C = fft2(A, 128, 128);
```

- See also:
  - fft()
  - conv2()
  - real()
  - fast Fourier transform in MathWorld
  - fast Fourier transform in Wikipedia

---

interp1(X, Y, XI, YI)
interp1(X, Y, XI, YI, method)
interp1(X, Y, XI, YI, method, extrapolation_value)

- 1D data interpolation
- Given a 1D function specified in vectors *X* and *Y* (where *X* specifies locations and *Y* specifies the corresponding values),
  generate vector *YI* which contains interpolated values at locations *XI*
- The *method* argument is optional; it is one of:
  `"nearest"` = interpolate using single nearest neighbour
  `"linear"` = linear interpolation between two nearest neighbours (**default setting**)
  `"*nearest"` = as per `"nearest"`, but faster by assuming that *X* and *XI* are monotonically increasing
  `"*linear"` = as per `"linear"`, but faster by assuming that *X* and *XI* are monotonically increasing
- If a location in *XI* is outside the domain of *X*, the corresponding value in *YI* is set to *extrapolation_value*
- The *extrapolation_value* argument is optional; by default it is datum::nan (not-a-number)
- Examples:

```
vec x = linspace<vec>(0, 3, 20);
vec y = square(x);

vec xx = linspace<vec>(0, 3, 100);

vec yy;

interp1(x, y, xx, yy);  // use linear interpolation by default

interp1(x, y, xx, yy, "*linear");  // faster than "linear"

interp1(x, y, xx, yy, "nearest");
```

- See also:
  - interp2()

- polyval()
- linspace()
- regspace()
- conv()
- interpolation in Wikipedia

---

interp2(X, Y, Z, XI, YI, ZI)
interp2(X, Y, Z, XI, YI, ZI, method)
interp2(X, Y, Z, XI, YI, ZI, method, extrapolation_value)

- 2D data interpolation
- Given a 2D function specified by matrix *Z* with coordinates given by vectors *X* and *Y*,
  generate matrix *ZI* which contains interpolated values at the coordinates given by vectors *XI* and *YI*
- The vector pairs *(X, Y)* and *(XI, YI)* define 2D coordinates in a grid;
  for example, *X* defines the horizontal coordinates and *Y* defines the corresponding vertical coordinates,
  so that (*X(m)*, *Y(n)*) is the 2D coordinate of element *Z(n,m)*
- The length of vector *X* must be equal to the number of columns in matrix *Z*
- The length of vector *Y* must be equal to the number of rows in matrix *Z*
- Vectors *X*, *Y*, *XI*, *YI* must contain monotonically increasing values (eg. 0.1, 0.2, 0.3, ...)
- The *method* argument is optional; it is one of:
  `"nearest"` = interpolate using nearest neighbours
  `"linear"` = linear interpolation between nearest neighbours (**default setting**)
- If a coordinate in the 2D grid specified by *(XI, YI)* is outside the domain of the 2D grid specified by *(X, Y)*,
  the corresponding value in *ZI* is set to *extrapolation_value*
- The *extrapolation_value* argument is optional; by default it is datum::nan (not-a-number)
- Examples:

```
mat Z;

Z.load("input_image.pgm", pgm_binary);  // load an image in pgm format

vec X = regspace(1, Z.n_cols);  // X = horizontal spacing
vec Y = regspace(1, Z.n_rows);  // Y = vertical spacing

vec XI = regspace(X.min(), 1.0/2.0, X.max()); // magnify by approx 2
vec YI = regspace(Y.min(), 1.0/3.0, Y.max()); // magnify by approx 3

mat ZI;

interp2(X, Y, Z, XI, YI, ZI);  // use linear interpolation by default

ZI.save("output_image.pgm", pgm_binary);
```

- See also:
  - interp1()
  - regspace()
  - conv2()
  - reshape()
  - bilinear interpolation in Wikipedia

P = polyfit(X, Y, N)
polyfit(P, X, Y, N)

- Given a 1D function specified in vectors *X* and *Y* (where *X* holds independent values and *Y* holds the corresponding dependent values),
  model the function as a polynomial of order *N* and store the polynomial coefficients in column vector *P*

- The given function is modelled as:
  $y = p_0 x^N + p_1 x^{N-1} + p_2 x^{N-2} + ... + p_{N-1} x^1 + p_N$
  where p_i is the i-th polynomial coefficient; the coefficients are selected to minimise the overall error of the fit (least squares)

- The column vector *P* has *N+1* coefficients

- *N* must be smaller than the number of elements in *X*

- If the polynomial coefficients cannot be found:
  - *P = polyfit(X, Y, N)* resets *P* and throws a *std::runtime_error* exception
  - *polyfit(P, X, Y, N)* resets *P* and returns a bool set to *false* (exception is not thrown)

- Examples:

```
vec x = linspace<vec>(0,4*datum::pi,100);
vec y = cos(x);

vec p = polyfit(x,y,10);
```

- See also:
  - polyval()
  - roots()
  - interp1()
  - polynomial in Wikipedia
  - least squares in Wikipedia
  - curve fitting in Wikipedia
  - least squares fitting in MathWorld

Y = polyval(P, X)

- Given vector *P* of polynomial coefficients and vector *X* containing the independent values of a 1D function, generate vector *Y* which contains the corresponding dependent values

- For each *x* value in vector *X*, the corresponding *y* value in vector *Y* is generated using:
  $y = p_0 x^N + p_1 x^{N-1} + p_2 x^{N-2} + ... + p_{N-1} x^1 + p_N$
  where p_i is the i-th polynomial coefficient in vector *P*
- *P* must contain polynomial coefficients in descending powers (eg. generated by the polyfit() function)
- Examples:

```
vec x1 = linspace<vec>(0,4*datum::pi,100);
vec y1 = cos(x1);
vec p1 = polyfit(x1,y1,10);

vec y2 = polyval(p1,x1);
```

- See also:
  - polyfit()
  - roots()
  - interp1()
  - polynomial in Wikipedia

Statistics & Clustering

---

mean, median, stddev, var, range

mean(V) **mean(M)** **mean(M, dim)** **mean(Q)** **mean(Q, dim)**	mean (average value)
median(V) **median(M)** **median(M, dim)**	median
stddev(V) **stddev(V, norm_type)** **stddev(M)** **stddev(M, norm_type)** **stddev(M, norm_type, dim)**	standard deviation
var(V) **var(V, norm_type)** **var(M)** **var(M, norm_type)** **var(M, norm_type, dim)**	variance

range(V)
range(M)
range(M, dim)
} range (difference between max and min)

- For vector *V*, return the statistic calculated using all the elements of the vector
- For matrix *M*, find the statistic for each column (*dim=0*), or each row (*dim=1*)
- For cube *Q*, find the statistics of elements along dimension *dim*, where *dim* $\in$ { 0, 1, 2 }
- The *dim* argument is optional; by default *dim=0* is used
- The *norm_type* argument is optional; by default *norm_type=0* is used
- For the *var()* and *stddev()* functions:
  - the default *norm_type=0* performs normalisation using *N-1* (where *N* is the number of samples), providing the best unbiased estimator
  - using *norm_type=1* performs normalisation using *N*, which provides the second moment around the mean
- **Caveat:** to obtain statistics for integer matrices/vectors (eg. umat, imat, uvec, ivec), convert to a matrix/vector with floating point values (eg. mat, vec) using the conv_to() function
- Examples:

```
mat A    = randu<mat>(5,5);
mat B    = mean(A);
mat C    = var(A);
double m = mean(mean(A));

vec    v = randu<vec>(5);
double x = var(v);
```

- See also:
  - cov()
  - cor()
  - diff()
  - hist()
  - histc()
  - quantile()
  - min() & max()
  - running_stat - class for running statistics of scalars
  - running_stat_vec - class for running statistics of vectors
  - gmm_diag / gmm_full - model and evaluate data using Gaussian Mixture Models (GMMs)
  - kmeans()

---

cov(X, Y)
cov(X, Y, norm_type)

cov(X)
cov(X, norm_type)

- For two matrix arguments *X* and *Y*, if each row of *X* and *Y* is an observation and each column is a variable, the

(i,j)-th entry of *cov(X,Y)* is the covariance between the *i*-th variable in *X* and the *j*-th variable in *Y*

- For vector arguments, the type of vector is ignored and each element in the vector is treated as an observation
- For matrices, *X* and *Y* must have the same dimensions
- For vectors, *X* and *Y* must have the same number of elements
- *cov(X)* is equivalent to *cov(X, X)*
- The default *norm_type=0* performs normalisation using *N-1* (where *N* is the number of observations), providing the best unbiased estimation of the covariance matrix (if the observations are from a normal distribution). Using *norm_type=1* causes normalisation to be done using *N*, which provides the second moment matrix of the observations about their mean
- Examples:

```
mat X = randu<mat>(4,5);
mat Y = randu<mat>(4,5);

mat C = cov(X,Y);
mat D = cov(X,Y, 1);
```

- See also:
  - cor()
  - statistics functions
  - running_stat_vec
  - Covariance in MathWorld

---

cor(X, Y)
cor(X, Y, norm_type)

cor(X)
cor(X, norm_type)

- For two matrix arguments *X* and *Y*, if each row of *X* and *Y* is an observation and each column is a variable, the *(i,j)*-th entry of *cor(X,Y)* is the correlation coefficient between the *i*-th variable in *X* and the *j*-th variable in *Y*
- For vector arguments, the type of vector is ignored and each element in the vector is treated as an observation
- For matrices, *X* and *Y* must have the same dimensions
- For vectors, *X* and *Y* must have the same number of elements
- *cor(X)* is equivalent to *cor(X, X)*
- The default *norm_type=0* performs normalisation of the correlation matrix using *N-1* (where *N* is the number of observations). Using *norm_type=1* causes normalisation to be done using *N*
- Examples:

```
mat X = randu<mat>(4,5);
```

```
mat Y = randu<mat>(4,5);

mat R = cor(X,Y);
mat S = cor(X,Y, 1);
```

- See also:
  - cov()
  - conv()
  - statistics functions
  - running_stat_vec
  - Correlation in MathWorld
  - Autocorrelation in MathWorld

---

hist(V)
hist(V, n_bins)
hist(V, centers)

hist(X, centers)
hist(X, centers, dim)

- For vector *V*, produce an unsigned vector of the same orientation as *V* (ie. either uvec or urowvec) that represents a histogram of counts
- For matrix *X*, produce a umat matrix containing either column histogram counts (for *dim=0*, default), or row histogram counts (for *dim=1*)
- The bin centers can be automatically determined from the data, with the number of bins specified via *n_bins* (default is 10); the range of the bins is determined by the range of the data
- The bin centers can also be explicitly specified via the *centers* vector; the vector must contain monotonically increasing values (eg. 0.1, 0.2, 0.3, ...)
- Examples:

```
vec  v  = randn<vec>(1000); // Gaussian distribution

uvec h1 = hist(v, 11);
uvec h2 = hist(v, linspace<vec>(-2,2,11));
```

- See also:
  - histc()
  - quantile()
  - statistics functions
  - conv_to()

---

histc(V, edges)
histc(X, edges)
histc(X, edges, dim)

- For vector *V*, produce an unsigned vector of the same orientation as *V* (ie. either uvec or urowvec) that contains the counts of the number of values that fall between the elements in the *edges* vector
- For matrix *X*, produce a *umat* matrix containing either column histogram counts (for *dim=0*, default), or row histogram counts (for *dim=1*)
- The *edges* vector must contain monotonically increasing values (eg. 0.1, 0.2, 0.3, ...)
- Examples:

```
vec  v = randn<vec>(1000);  // Gaussian distribution

uvec h = histc(v, linspace<vec>(-2,2,11));
```

- See also:
  - hist()
  - statistics functions
  - conv_to()

---

quantile(V, P)
quantile(X, P)
quantile(X, P, dim)

- For a dataset stored in vector *V* or matrix *X*, calculate the quantiles corresponding to the cumulative probability values in the given *P* vector
- For vector *V*, produce a vector with the same orientation as *V* and the same length as *P*
- For matrix *X*, produce a matrix with the quantiles for each column vector (*dim=0*) or each row vector (*dim=0*)
- The *dim* argument is optional; by default *dim=0*
- The *P* vector must contain values in the [0,1] interval (eg. 0.00, 0.25, 0.50, 0.75, 1.00)
- The algorithm for calculating the quantiles is based on *Definition 5* in:
  Rob J. Hyndman and Yanan Fan. Sample Quantiles in Statistical Packages. The American Statistician, Vol. 50, No. 4, pp. 361-365, 1996. http://doi.org/10.2307/2684934
- Examples:

```
vec V = randn<vec>(1000);  // Gaussian distribution
vec P = { 0.25, 0.50, 0.75 };

vec Q = quantile(V, P);
```

- See also:
  - hist()
  - median()
  - normcdf()
  - quantile in Wikipedia

mat coeff = princomp(mat X)
cx_mat coeff = princomp(cx_mat X)

princomp(mat coeff, mat X)
princomp(cx_mat coeff, cx_mat X)

princomp(mat coeff, mat score, mat X)
princomp(cx_mat coeff, cx_mat score, cx_mat X)

princomp(mat coeff, mat score, vec latent, mat X)
princomp(cx_mat coeff, cx_mat score, vec latent, cx_mat X)

princomp(mat coeff, mat score, vec latent, vec tsquared, mat X)
princomp(cx_mat coeff, cx_mat score, vec latent, cx_vec tsquared, cx_mat X)

- Principal component analysis of matrix *X*
- Each row of *X* is an observation and each column is a variable
- output objects:
  - *coeff*: principal component coefficients
  - *score*: projected data
  - *latent*: eigenvalues of the covariance matrix of *X*
  - *tsquared*: Hotteling's statistic for each sample
- The computation is based on singular value decomposition
- If the decomposition fails:
  - *coeff = princomp(X)* resets *coeff* and throws a *std::runtime_error* exception
  - remaining forms of *princomp()* reset all output matrices and return a bool set to *false* (exception is not thrown)
- Examples:

```
mat A = randu<mat>(5,4);

mat coeff;
mat score;
vec latent;
vec tsquared;

princomp(coeff, score, latent, tsquared, A);
```

- See also:
  - eig_sym()
  - svd()
  - svd_econ()
  - principal components analysis in Wikipedia

---

normpdf(X)

normpdf(X, M, S)

- For each scalar *x* in *X*, compute its probability density function according to a Gaussian (normal) distribution using the corresponding mean value *m* in *M* and the corresponding standard deviation value *s* in *S*:

$$y = \frac{1}{s\sqrt{2\pi}} \exp\left[-0.5 \frac{(x-m)^2}{s^2}\right]$$

- *X* can be a scalar, vector, or matrix
- *M* and *S* can jointly be either scalars, vectors, or matrices
- If *M* and *S* are omitted, their values are assumed to be 0 and 1, respectively
- **Caveat:** to reduce the incidence of numerical underflows, consider using log_normpdf()
- Examples:

```
vec X(10, fill::randu);
vec M(10, fill::randu);
vec S(10, fill::randu);

   vec P1 = normpdf(X);
   vec P2 = normpdf(X,    M,    S   );
   vec P3 = normpdf(1.23, M,    S   );
   vec P4 = normpdf(X,    4.56, 7.89);
double P5 = normpdf(1.23, 4.56, 7.89);
```

- See also:
  - log_normpdf()
  - normcdf()
  - randn()
  - gmm_diag / gmm_full - model and evaluate data using Gaussian Mixture Models (GMMs)
  - normal distribution in Wikipedia

---

log_normpdf(X)
log_normpdf(X, M, S)

- For each scalar *x* in *X*, compute the logarithm version of probability density function according to a Gaussian (normal) distribution using the corresponding mean value *m* in *M* and the corresponding standard deviation value *s* in *S*:

$$y = \log\left[\frac{1}{s\sqrt{2\pi}} \exp\left[-0.5 \frac{(x-m)^2}{s^2}\right]\right]$$

$$= -\log(s\sqrt{2\pi}) + \left[-0.5 \frac{(x-m)^2}{s^2}\right]$$

- *X* can be a scalar, vector, or matrix

- *M* and *S* can jointly be either scalars, vectors, or matrices
- If *M* and *S* are omitted, their values are assumed to be 0 and 1, respectively
- Examples:

```
vec X(10, fill::randu);
vec M(10, fill::randu);
vec S(10, fill::randu);

   vec P1 = log_normpdf(X);
   vec P2 = log_normpdf(X,    M,    S   );
   vec P3 = log_normpdf(1.23, M,    S   );
   vec P4 = log_normpdf(X,    4.56, 7.89);
double P5 = log_normpdf(1.23, 4.56, 7.89);
```

- See also:
  - normpdf()
  - gmm_diag / gmm_full - model and evaluate data using Gaussian Mixture Models (GMMs)
  - normal distribution in Wikipedia

---

normcdf(X)
normcdf(X, M, S)

- For each scalar *x* in *X*, compute its cumulative distribution function according to a Gaussian (normal) distribution using the corresponding mean value *m* in *M* and the corresponding standard deviation value *s* in *S*
- *X* can be a scalar, vector, or matrix
- *M* and *S* can jointly be either scalars, vectors, or matrices
- If *M* and *S* are omitted, their values are assumed to be 0 and 1, respectively
- Examples:

```
vec X(10, fill::randu);
vec M(10, fill::randu);
vec S(10, fill::randu);

   vec P1 = normcdf(X);
   vec P2 = normcdf(X,    M,    S   );
   vec P3 = normcdf(1.23, M,    S   );
   vec P4 = normcdf(X,    4.56, 7.89);
double P5 = normcdf(1.23, 4.56, 7.89);
```

- See also:
  - normpdf()
  - quantile()
  - randn()
  - normal distribution in Wikipedia
  - cumulative distribution function in Wikipedia

X = mvnrnd(M, C)
X = mvnrnd(M, C, N)

mvnrnd(X, M, C)
mvnrnd(X, M, C, N)

- Generate a matrix with random column vectors from a multivariate Gaussian (normal) distribution with parameters *M* and *C*:
  - *M* is the mean; must be a column vector
  - *C* is the covariance matrix; must be symmetric positive semi-definite (preferably positive definite)
- *N* is the number of column vectors to generate; if *N* is omitted, it is assumed to be 1
- **Caveat:** repeated generation of one vector (or a small number of vectors) using the same *M* and *C* parameters can be inefficient;
  for repeated generation consider using the *generate()* function in the gmm_diag and gmm_full classes
- If generating the random vectors fails:
  - *X = mvnrnd(M, C)* and *X = mvnrnd(M, C, N)* reset *X* and throw a *std::runtime_error* exception
  - *mvnrnd(X, M, C)* and *mvnrnd(X, M, C, N)* reset *X* and return a bool set to *false* (exception is not thrown)
- Examples:

```
vec M(5, fill::randu);

mat B(5, 5, fill::randu);
mat C = B.t() * B;

mat X = mvnrnd(M, C, 100);
```

- See also:
  - randn()
  - chi2rnd()
  - wishrnd()
  - cov()
  - .is_sympd()
  - gmm_diag / gmm_full - model and evaluate data using Gaussian Mixture Models (GMMs)
  - multivariate normal distribution in Wikipedia

---

chi2rnd(DF)
chi2rnd(DF_scalar)
chi2rnd(DF_scalar, n_elem)
chi2rnd(DF_scalar, n_rows, n_cols)
chi2rnd(DF_scalar, size(X))

- Generate a random scalar, vector or matrix with elements sampled from a chi-squared distribution with the degrees of freedom specified by parameter *DF* or *DF_scalar*
- *DF* is a vector or matrix, while *DF_scalar* is a scalar
- Each value in *DF* and *DF_scalar* must be greater than zero

- For the *chi2rnd(DF)* form, the output vector/matrix has the same size and type as *DF*; each element in *DF* specifies a separate degree of freedom

- Usage:
  - *vector_type* v = chi2rnd(DF), where the type of DF is a real *vector_type*
  - *matrix_type* X = chi2rnd(DF), where the type of DF is a real *matrix_type*

  - *scalar_type* s = chi2rnd<*scalar_type*>(DF_scalar), where *scalar_type* is either *float* or *double*
  - *vector_type* v = chi2rnd<*vector_type*>(DF_scalar, n_elem)
  - *matrix_type* X = chi2rnd<*matrix_type*>(DF_scalar, n_rows, n_cols)
  - *matrix_type* Y = chi2rnd<*matrix_type*>(DF_scalar, size(X))

- Examples:

```
mat X = chi2rnd(2, 5, 6);

mat A = randi<mat>(5, 6, distr_param(1, 10));
mat B = chi2rnd(A);
```

- See also:
  - randg()
  - randn()
  - mvnrnd()
  - wishrnd()
  - size()
  - chi-squared distribution in Wikipedia

W = wishrnd(S, df)
W = wishrnd(S, df, D)

wishrnd(W, S, df)
wishrnd(W, S, df, D)

- Generate a random matrix sampled from the Wishart distribution with parameters *S* and *df*:
  - *S* is a symmetric positive definite matrix (eg. a covariance matrix)
  - *df* is a scalar specifying the degrees of freedom; it can be a non-integer value

- *D* is an optional argument; it specifies the Cholesky decomposition of *S*; if *D* is provided, *S* is ignored; using *D* is more efficient if you need to use *wishrnd()* many times for the same *S* matrix

- If generating the random matrix fails:
  - *W = wishrnd(S, df)* and *W = wishrnd(S, df, D)* reset *W* and throw a *std::runtime_error* exception
  - *wishrnd(W, S, df)* and *wishrnd(W, S, df, D)* reset *W* and return a bool set to *false* (exception is not thrown)

- Examples:

```
mat X(5, 5, fill::randu);

mat S = X.t() * X;

mat W = wishrnd(S, 6.7);
```

- See also:
  - iwishrnd()
  - chi2rnd()
  - mvnrnd()
  - chol()
  - cov()
  - randn()
  - .is_sympd()
  - Wishart distribution in Wikipedia

W = iwishrnd(T, df)
W = iwishrnd(T, df, Dinv)

iwishrnd(W, T, df)
iwishrnd(W, T, df, Dinv)

- Generate a random matrix sampled from the inverse Wishart distribution with parameters *T* and *df*:
  - *T* is a symmetric positive definite matrix
  - *df* is a scalar specifying the degrees of freedom; it can be a non-integer value

- *Dinv* is an optional argument; it specifies the Cholesky decomposition of the inverse of *T*; if *Dinv* is provided, *T* is ignored
  using *Dinv* is more efficient if you need to use *iwishrnd()* many times for the same *T* matrix

- If generating the random matrix fails:
  - *W = iwishrnd(T, df)* and *W = iwishrnd(T, df, Dinv)* reset *W* and throw a *std::runtime_error* exception
  - *iwishrnd(W, T, df)* and *iwishrnd(W, T, df, Dinv)* reset *W* and return a bool set to *false* (exception is not thrown)

- Examples:

```
mat X(5, 5, fill::randu);

mat T = X.t() * X;

mat W = iwishrnd(T, 6.7);
```

- See also:
  - wishrnd()
  - chol()
  - inv_sympd()
  - .is_sympd()
  - inverse Wishart distribution in Wikipedia

---

running_stat<*type*>

- Class for running statistics (online statistics) of scalars (one dimensional process/signal)

- Useful if the storage of all samples (scalars) is impractical, or if the number of samples is not known in advance
- *type* is one of: *float*, *double*, *cx_float*, *cx_double*
- For an instance of *running_stat* named as *X*, the member functions are:

X(scalar**)**	update the statistics using the given scalar
X.min()	current minimum value
X.max()	current maximum value
X.range()	current range
X.mean()	current mean or average value
X.var() and **X.var(**norm_type**)**	current variance
X.stddev() and **X.stddev(**norm_type**)**	current standard deviation
X.reset()	reset all statistics and set the number of samples to zero
X.count()	current number of samples

- The *norm_type* argument is optional; by default *norm_type=0* is used
- For the *.var()* and *.stddev()* functions, the default *norm_type=0* performs normalisation using *N-1* (where *N* is the number of samples so far), providing the best unbiased estimator; using *norm_type=1* causes normalisation to be done using *N*, which provides the second moment around the mean
- The return type of *.count()* depends on the underlying form of *type*: it is either *float* or *double*
- Examples:

```
running_stat<double> stats;

for(uword i=0; i<10000; ++i)
  {
  double sample = randn();
  stats(sample);
  }

cout << "mean = " << stats.mean() << endl;
cout << "var  = " << stats.var()  << endl;
cout << "min  = " << stats.min()  << endl;
cout << "max  = " << stats.max()  << endl;
```

- See also:
  - running_stat_vec (running statistics of vectors)
  - statistics functions
  - gmm_diag / gmm_full - model and evaluate data using Gaussian Mixture Models (GMMs)

---

running_stat_vec<*vec_type***>**
running_stat_vec<*vec_type***>(calc_cov)**

- Class for running statistics (online statistics) of vectors (multi-dimensional process/signal)
- Useful if the storage of all samples (vectors) is impractical, or if the number of samples is not known in advance

- This class is similar to *running_stat*, with the difference that vectors are processed instead of scalars
- *vec_type* is the vector type of the samples; for example: *vec*, *cx_vec*, *rowvec*, ...
- For an instance of *running_stat_vec* named as *X*, the member functions are:

X(vector**)**	update the statistics using the given vector
X.min()	vector of current minimum values
X.max()	vector of current maximum values
X.range()	vector of current ranges
X.mean()	vector of current means
X.var() and **X.var(**norm_type**)**	vector of current variances
X.stddev() and **X.stddev(**norm_type**)**	vector of current standard deviations
X.cov() and **X.cov(**norm_type**)**	matrix of current covariances; valid if *calc_cov=true* during construction of *running_stat_vec*
X.reset()	reset all statistics and set the number of samples to zero
X.count()	current number of samples

- The *calc_cov* argument is optional; by default *calc_cov=false*, indicating that the covariance matrix will not be calculated; to enable the covariance matrix, use *calc_cov=true* during construction; for example: *running_stat_vec<vec> X(true);*
- The *norm_type* argument is optional; by default *norm_type=0* is used
- For the *.var()* and *.stddev()* functions, the default *norm_type=0* performs normalisation using *N-1* (where *N* is the number of samples so far), providing the best unbiased estimator; using *norm_type=1* causes normalisation to be done using *N*, which provides the second moment around the mean
- The return type of *.count()* depends on the underlying form of *vec_type*: it is either *float* or *double*
- Examples:

```
running_stat_vec<vec> stats;

vec sample;

for(uword i=0; i<10000; ++i)
  {
  sample = randu<vec>(5);
  stats(sample);
  }

cout << "mean = " << endl << stats.mean() << endl;
cout << "var  = " << endl << stats.var()  << endl;
cout << "max  = " << endl << stats.max()  << endl;

//
//

running_stat_vec<rowvec> more_stats(true);

for(uword i=0; i<20; ++i)
  {
  sample = randu<rowvec>(3);
```

```
  sample(1) -= sample(0);
  sample(2) += sample(1);

  more_stats(sample);
  }

cout << "covariance matrix = " << endl;
cout << more_stats.cov() << endl;

rowvec sd = more_stats.stddev();

cout << "correlations = " << endl;
cout << more_stats.cov() / (sd.t() * sd);
```

- See also:
  - running_stat (running statistics of scalars)
  - statistics functions
  - cov()
  - cor()
  - gmm_diag / gmm_full - model and evaluate data using Gaussian Mixture Models (GMMs)

---

kmeans(means**,** data**,** k**,** seed_mode**,** n_iter**,** print_mode **)**

- Cluster given data into *k* disjoint sets
- The *means* parameter is the output matrix for storing the resulting centroids of the sets, with each centroid stored as a column vector
- The *data* parameter is the input data matrix, with each sample stored as a column vector
- The *k* parameter indicates the number of centroids; the number of samples in the *data* matrix should be much larger than *k*
- The *seed_mode* parameter specifies how the initial centroids are seeded; it is one of:

`keep_existing`	use the centroids specified in the *means* matrix as the starting point
`static_subset`	use a subset of the data vectors (repeatable)
`random_subset`	use a subset of the data vectors (random)
`static_spread`	use a maximally spread subset of data vectors (repeatable)
`random_spread`	use a maximally spread subset of data vectors (random start)

  caveat: seeding the initial centroids with `static_spread` and `random_spread` can be much more time consuming than with `static_subset` and `random_subset`
- The *n_iter* parameter specifies the number of clustering iterations; this is data dependent, but about 10 is typically sufficient
- The *print_mode* parameter is either *true* or *false*, indicating whether progress is printed during clustering
- If the clustering fails, the *means* matrix is reset and a bool set to *false* is returned
- The clustering will run faster on multi-core machines when OpenMP is enabled in your compiler (eg. *-fopenmp*

in GCC and clang)

- Examples:

```
uword d = 5;       // dimensionality
uword N = 10000;   // number of vectors

mat data(d, N, fill::randu);

mat means;

bool status = kmeans(means, data, 2, random_subset, 10, true);

if(status == false)
  {
  cout << "clustering failed" << endl;
  }

means.print("means:");
```

- See also:
  - gmm_diag / gmm_full - model and evaluate data using Gaussian Mixture Models (GMMs)
  - statistics functions
  - running_stat_vec
  - k-means clustering in Wikipedia
  - k-means clustering in MathWorld
  - OpenMP in Wikipedia

---

gmm_diag
gmm_full

- Classes for multivariate data modelling and evaluation via **Gaussian Mixture Models** (GMMs)
- The *gmm_diag* class is tailored for **diagonal covariance matrices** (ie. in each covariance matrix, all entries outside the main diagonal are assumed to be zero)
- The *gmm_full* class is tailored for **full covariance matrices**
- The *gmm_diag* class is typically much faster to train and use than the *gmm_full* class, at the potential cost of some reduction in modelling accuracy
- The *gmm_diag* and *gmm_full* classes include dedicated optimisation algorithms for learning (training) the model parameters from data:
  - k-means clustering, for quick initial estimates
  - Expectation-Maximisation (EM), for maximum-likelihood estimates

  The optimisation algorithms are multi-threaded and can run much quicker on multi-core machines when OpenMP is enabled in your compiler (eg. *-fopenmp* in GCC and clang)
- The classes can also be used for probabilistic clustering and vector quantisation (VQ)
- Data is modelled as:

$$p(x) = \sum_{g=0}^{n\_gaus-1} h_g \; \mathrm{N}(x \,|\, m_g, C_g)$$

  where:
  - *n_gaus* is the number of Gaussians; $n\_gaus \geq 1$
  - $\mathrm{N}(x \,|\, m_g, C_g)$ represents a Gaussian (normal) distribution
  - each Gaussian g has the following parameters:
    - h_g is the heft (weight), with constraints $h_g \geq 0$ and $\sum h_g = 1$
    - m_g is the mean vector (centroid) with dimensionality *n_dims*
    - C_g is the covariance matrix (either diagonal or full)

- Please cite the following paper if you use the *gmm_diag* or *gmm_full* classes in your research and/or software:

  Conrad Sanderson and Ryan Curtin.
  An Open Source C++ Implementation of Multi-Threaded Gaussian Mixture Models, k-Means and Expectation Maximisation.
  International Conference on Signal Processing and Communication Systems, 2017.

- For an instance of *gmm_diag* or *gmm_full* named as *M*, the member functions and variables are:

M.log_p(V**)**	return a scalar representing the log-likelihood of vector *V* (of type *vec*)
M.log_p(V, g**)**	return a scalar representing the log-likelihood of vector *V* (of type *vec*), according to Gaussian with index *g*

M.log_p(X**)**	return a row vector (of type `rowvec`) containing log-likelihoods of each column vector in matrix *X* (of type *mat*)
M.log_p(X, g**)**	return a row vector (of type `rowvec`) containing log-likelihoods of each column vector in matrix *X* (of type *mat*), according to Gaussian with index *g*
M.sum_log_p(X**)**	return a scalar representing the sum of log-likelihoods for all column vectors in matrix *X* (of type *mat*)
M.sum_log_p(X, g**)**	return a scalar representing the sum of log-likelihoods for all column vectors in matrix *X* (of type *mat*), according to Gaussian with index *g*
M.avg_log_p(X**)**	return a scalar representing the average log-likelihood of all column vectors in matrix *X* (of type *mat*)
M.avg_log_p(X, g**)**	return a scalar representing the average log-likelihood of all column vectors in matrix *X* (of type *mat*), according to Gaussian with index *g*
M.assign(V, dist_mode**)**	return the index of the closest mean (or Gaussian) to vector *V* (of type *vec*); parameter *dist_mode* is one of: `eucl_dist` Euclidean distance (takes only means into account) `prob_dist` probabilistic "distance", defined as the inverse likelihood (takes into account means, covariances and hefts)
M.assign(X, dist_mode**)**	return a row vector (of type *urowvec*) containing the indices of the closest means (or Gaussians) to each column vector in matrix *X* (of type *mat*); parameter *dist_mode* is either `eucl_dist` or `prob_dist` (as per the **.assign()** function above)
M.raw_hist(X, dist_mode**)**	return a row vector (of type `urowvec`) representing the raw histogram of counts; each entry is the number of counts corresponding to a Gaussian; each count is the number times the corresponding Gaussian was the closest to each column vector in matrix *X*; parameter *dist_mode* is either `eucl_dist` or `prob_dist` (as per the **.assign()** function above)
M.norm_hist(X, dist_mode**)**	similar to the **.raw_hist()** function above; return a row vector (of type `rowvec`) containing normalised counts; the vector sums to one; parameter *dist_mode* is either `eucl_dist` or `prob_dist` (as per the **.assign()** function above)
M.generate()	return a column vector (of type *vec*) representing a random sample generated according to the model's parameters
M.generate(N**)**	return a matrix (of type *mat*) containing *N* column vectors, with each vector representing a random sample generated according to the model's parameters
M.save(filename**)**	save the model to a file and return a *bool* indicating either success (*true*) or failure (*false*)

M.load(filename**)**	load the model from a file and return a *bool* indicating either success (*true*) or failure (*false*)
M.n_gaus()	return the number of means/Gaussians in the model
M.n_dims()	return the dimensionality of the means/Gaussians in the model
M.reset(n_dims, n_gaus**)**	set the model to have dimensionality *n_dims*, with *n_gaus* number of Gaussians; all the means are set to zero, all covariance matrix representations are equivalent to the identity matrix, and all the hefts (weights) are set to be uniform
M.hefts	read-only row vector (of type *rowvec*) containing the hefts (weights)
M.means	read-only matrix (of type *mat*) containing the means (centroids), stored as column vectors
M.dcovs *[only in gmm_diag]*	read-only matrix (of type *mat*) containing the representation of diagonal covariance matrices, with the set of diagonal covariances for each Gaussian stored as a column vector; applicable only to the *gmm_diag* class
M.fcovs *[only in gmm_full]*	read-only cube containing the full covariance matrices, with each covariance matrix stored as a slice within the cube; applicable only to the *gmm_full* class
M.set_hefts(V**)**	set the hefts (weights) of the model to be as specified in row vector *V* (of type *rowvec*); the number of hefts must match the existing model
M.set_means(X**)**	set the means to be as specified in matrix *X* (of type *mat*); the number of means and their dimensionality must match the existing model
M.set_dcovs(X**)** *[only in gmm_diag]*	set the diagonal covariances matrices to be as specified in matrix *X* (of type *mat*), with the set of diagonal covariances for each Gaussian stored as a column vector; the number of covariance matrices and their dimensionality must match the existing model; applicable only to the *gmm_diag* class
M.set_fcovs(X**)** *[only in gmm_full]*	set the full covariances matrices to be as specified in cube *X*, with each covariance matrix stored as a slice within the cube; the number of covariance matrices and their dimensionality must match the existing model; applicable only to the *gmm_full* class
M.set_params(means, covs, hefts**)**	set all the parameters at the same time; the type and layout of the parameters is as per the **.set_hefts()**, **.set_means()**, **.set_dcovs()** and **.set_fcovs()** functions above; the number of Gaussians and dimensionality can be different from the existing model

M.learn(data, n_gaus, dist_mode, seed_mode, km_iter, em_iter, var_floor, print_mode**)**
learn the model parameters via multi-threaded k-means and/or EM algorithms; return a `bool` value, with *true* indicating success, and *false* indicating failure; the parameters have the following meanings:

data
matrix (of type *mat*) containing training samples; each sample is stored as a column vector

n_gaus
set the number of Gaussians to *n_gaus*; to help convergence, it is recommended that the given *data* matrix (above) contains at least 10 samples for each Gaussian

dist_mode
specifies the distance used during the seeding of initial means and k-means clustering:

`eucl_dist`	Euclidean distance
`maha_dist`	Mahalanobis distance, which uses a global diagonal covariance matrix estimated from the training samples; this is recommended for probabilistic applications

seed_mode
specifies how the initial means are seeded prior to running k-means and/or EM algorithms:

`keep_existing`	keep the existing model (do not modify the means, covariances and hefts)
`static_subset`	a subset of the training samples (repeatable)
`random_subset`	a subset of the training samples (random)
`static_spread`	a maximally spread subset of training samples (repeatable)
`random_spread`	a maximally spread subset of training samples (random start)

caveat: seeding the initial means with `static_spread` and `random_spread` can be much more time consuming than with `static_subset` and `random_subset`

km_iter
the number of iterations of the k-means algorithm; this is data dependent, but typically 10 iterations are sufficient

em_iter
the number of iterations of the EM algorithm; this is data dependent, but typically 5 to 10 iterations are sufficient

var_floor
the variance floor (smallest allowed value) for the diagonal covariances; setting this to a small non-zero value can help with convergence and/or better quality parameter estimates

print_mode
either `true` or `false`; enable or disable printing of progress during the k-means and EM algorithms

- Examples:

```
// create synthetic data with 2 Gaussians

uword d = 5;       // dimensionality
uword N = 10000;   // number of vectors

mat data(d, N, fill::zeros);

vec mean0 = linspace<vec>(1,d,d);
vec mean1 = mean0 + 2;

uword i = 0;

while(i < N)
  {
  if(i < N)  { data.col(i) = mean0 + randn<vec>(d); ++i; }
  if(i < N)  { data.col(i) = mean0 + randn<vec>(d); ++i; }
  if(i < N)  { data.col(i) = mean1 + randn<vec>(d); ++i; }
  }


// model the data as a diagonal GMM with 2 Gaussians

gmm_diag model;

bool status = model.learn(data, 2, maha_dist, random_subset, 10, 5, 1e-10, true);

if(status == false)
  {
  cout << "learning failed" << endl;
  }

model.means.print("means:");

double  scalar_likelihood = model.log_p( data.col(0)    );
rowvec     set_likelihood = model.log_p( data.cols(0,9) );

double overall_likelihood = model.avg_log_p(data);

uword   gaus_id  = model.assign( data.col(0),    eucl_dist );
urowvec gaus_ids = model.assign( data.cols(0,9), prob_dist );

urowvec hist1 = model.raw_hist (data, prob_dist);
 rowvec hist2 = model.norm_hist(data, eucl_dist);

model.save("my_model.gmm");
```

- See also:
  - normpdf()
  - mvnrnd()
  - statistics functions
  - running_stat_vec
  - kmeans()
  - covariance matrix in Wikipedia
  - covariance matrix in MathWorld
  - Mahalanobis distance in Wikipedia
  - multivariate normal distribution in Wikipedia
  - mixture model in Wikipedia
  - k-means clustering in Wikipedia
  - k-means clustering in MathWorld
  - Expectation-Maximisation algorithm in Wikipedia

- maximum likelihood in MathWorld
- vector quantisation in Wikipedia
- OpenMP in Wikipedia

Miscellaneous

---

constants (pi, inf, speed of light, ...)

`datum::pi`	π, the ratio of any circle's circumference to its diameter
`datum::inf`	∞, infinity
`datum::nan`	“not a number” (NaN); **caveat:** NaN is not equal to anything, even itself
`datum::e`	base of the natural logarithm
`datum::sqrt2`	square root of 2
`datum::sqrt2pi`	square root of 2π
`datum::eps`	machine epsilon: the difference between 1 and the value least greater than 1 that is representable (type and machine dependent)
`datum::log_min`	log of minimum non-zero value (type and machine dependent)
`datum::log_max`	log of maximum value (type and machine dependent)
`datum::euler`	Euler's constant, aka Euler-Mascheroni constant
`datum::gratio`	golden ratio
`datum::m_u`	atomic mass constant (in kg)
`datum::N_A`	Avogadro constant
`datum::k`	Boltzmann constant (in joules per kelvin)
`datum::k_evk`	Boltzmann constant (in eV/K)
`datum::a_0`	Bohr radius (in meters)
`datum::mu_B`	Bohr magneton
`datum::Z_0`	characteristic impedance of vacuum (in ohms)
`datum::G_0`	conductance quantum (in siemens)
`datum::k_e`	Coulomb's constant (in meters per farad)
`datum::eps_0`	electric constant (in farads per meter)
`datum::m_e`	electron mass (in kg)
`datum::eV`	electron volt (in joules)
`datum::ec`	elementary charge (in coulombs)
`datum::F`	Faraday constant (in coulombs)

`datum::alpha`	fine-structure constant
`datum::alpha_inv`	inverse fine-structure constant
`datum::K_J`	Josephson constant
`datum::mu_0`	magnetic constant (in henries per meter)
`datum::phi_0`	magnetic flux quantum (in webers)
`datum::R`	molar gas constant (in joules per mole kelvin)
`datum::G`	Newtonian constant of gravitation (in newton square meters per kilogram squared)
`datum::h`	Planck constant (in joule seconds)
`datum::h_bar`	Planck constant over 2 pi, aka reduced Planck constant (in joule seconds)
`datum::m_p`	proton mass (in kg)
`datum::R_inf`	Rydberg constant (in reciprocal meters)
`datum::c_0`	speed of light in vacuum (in meters per second)
`datum::sigma`	Stefan-Boltzmann constant
`datum::R_k`	von Klitzing constant (in ohms)
`datum::b`	Wien wavelength displacement law constant

- The constants are stored in the *Datum<type>* class, where *type* is either *float* or *double*;
  for convenience, *Datum<double>* is typedefed as *datum*, and *Datum<float>* is typedefed as *fdatum*
- **Caveat:** `datum::nan` is not equal to anything, even itself; to check whether a scalar *x* is finite, use std::isfinite(*x*)
- The physical constants were mainly taken from NIST 2018 CODATA values, and some from WolframAlpha (as of 2009-06-23)
- Examples:

```
cout << "2.0 * pi = " << 2.0 * datum::pi << endl;

cout << "speed of light = " << datum::c_0 << endl;

cout << "log_max for floats = ";
cout << fdatum::log_max << endl;

cout << "log_max for doubles = ";
cout << datum::log_max << endl;
```

- See also:
  - .is_finite()
  - .fill()
  - NaN in Wikipedia
  - physical constant in Wikipedia
  - std::numeric_limits

---

wall_clock

- Simple wall clock timer class for measuring the number of elapsed seconds
- Examples:

```
wall_clock timer;

mat A = randu<mat>(100,100);
mat B = randu<mat>(100,100);
mat C;

timer.tic();

for(uword i=0; i<100000; ++i)
  {
  C = A + B + A + B;
  }

double n = timer.toc();

cout << "number of seconds: " << n << endl;
```

---

logging of warnings and errors

set_cerr_stream(user_stream)
set_cout_stream(user_stream)

std::ostream& x = get_cerr_stream()
std::ostream& x = get_cout_stream()

- In Armadillo 8.x and later versions, warnings and errors are printed by default to the *std::cerr* stream; in Armadillo 7.x and earlier, the *std::cout* stream is used
  - the printing can be disabled by placing *#define ARMA_DONT_PRINT_ERRORS* before *#include <armadillo>*
  - detailed information about errors is always reported via the base *std::exception* class
- **set_cerr_stream()**:
  - change the stream for printing warnings and errors involving out of bounds accesses, failed decompositions and out of memory conditions
  - the stream can also be changed via the ARMA_CERR_STREAM define; see config.hpp
- **set_cout_stream()**:
  - change the default stream for printing matrices and cubes with .print() and .raw_print()
  - the stream can also be changed via the ARMA_COUT_STREAM define; see config.hpp
- **get_cerr_stream()**:
  - get a reference to the stream for printing warnings and errors
- **get_cout_stream()**:
  - get a reference to the stream for printing matrices and cubes
- Examples:

```
// print error messages to a log file
ofstream f("my_log.txt");
```

```
set_cerr_stream(f);

// trying to invert a singular matrix
// will print an error message and throw an exception
mat X = zeros<mat>(5,5);
mat Y = inv(X);

// disable printing of error messages
std::ostream nullstream(0);
set_cerr_stream(nullstream);
```

- See also:
  - config.hpp
  - .print()
  - std::cout
  - std::ostream
  - std::exception
  - tutorial on exceptions

---

uword, **sword**

- *uword* is a typedef for an unsigned integer type; it is used for matrix indices as well as all internal counters and loops
- *sword* is a typedef for a signed integer type
- The minimum width of both *uword* and *sword* is either 32 or 64 bits:
  - when using the old C++98 / C++03 standards, the default width is 32 bits
  - when using the new C++11 / C++14 standards, the default width is 64 bits on 64-bit platforms
- The width can also be forcefully set to 64 bits by enabling *ARMA_64BIT_WORD* via editing *include/armadillo_bits/config.hpp*
- See also:
  - C++ variable types
  - explanation of *typedef*
  - imat & umat matrix types
  - ivec & uvec vector types

---

cx_double, **cx_float**

- *cx_double* is a shorthand / typedef for *std::complex<double>*
- *cx_float* is a shorthand / typedef for *std::complex<float>*
- Example:

```
cx_mat X = randu<cx_mat>(5,5);

X(1,2) = cx_double(2.0, 3.0);
```

- See also:
  - complex numbers in the standard C++ library
  - explanation of *typedef*
  - cx_mat matrix type
  - cx_vec vector type

Examples of Matlab/Octave syntax and conceptually corresponding Armadillo syntax

Matlab/Octave	Armadillo	Notes
A(1, 1)	A(0, 0)	indexing in Armadillo starts at 0
A(k, k)	A(k-1, k-1)	
size(A,1)	A.n_rows	read only
size(A,2)	A.n_cols	
size(Q,3)	Q.n_slices	Q is a cube (3D array)
numel(A)	A.n_elem	
A(:, k)	A.col(k)	this is a conceptual example only; exact conversion from Matlab/Octave to Armadillo syntax will require taking into account that indexing starts at 0
A(k, :)	A.row(k)	
A(:, p:q)	A.cols(p, q)	
A(p:q, :)	A.rows(p, q)	
A(p:q, r:s)	A(span(p,q), span(r,s))	A(span(first_row, last_row), span(first_col, last_col))
Q(:, :, k)	Q.slice(k)	Q is a cube (3D array)
Q(:, :, t:u)	Q.slices(t, u)	
Q(p:q, r:s, t:u)	Q(span(p,q), span(r,s), span(t,u))	
A'	A.t() or trans(A)	matrix transpose / Hermitian transpose (for complex matrices, the conjugate of each element is taken)
A = zeros(size(A))	A.zeros()	
A = ones(size(A))	A.ones()	
A = zeros(k)	A = zeros<mat>(k,k)	
A = ones(k)	A = ones<mat>(k,k)	
C = complex(A,B)	cx_mat C = cx_mat(A,B)	
A .* B	A % B	element-wise multiplication
A ./ B	A / B	element-wise division

A \ B	solve(A,B)	conceptually similar to inv(A)*B, but more efficient
A = A + 1;	A++	
A = A - 1;	A--	
A = [1 2; 3 4;]	A << 1 << 2 << endr << 3 << 4 << endr;	element initialisation, with special element *endr* indicating *end of row*
X = A(:)	X = vectorise(A)	
X = [A B]	X = join_horiz(A,B)	
X = [A; B]	X = join_vert(A,B)	
A	cout << A << endl; or A.print("A =");	
save -ascii 'A.dat' A	A.save("A.dat", raw_ascii);	Matlab/Octave matrices saved as ascii are readable by Armadillo (and vice-versa)
load -ascii 'A.dat'	A.load("A.dat", raw_ascii);	

A = randn(2,3);
B = randn(4,5);
F = { A; B }
	mat A = randn(2,3);
	mat B = randn(4,5);
	field<mat> F(2,1);
	F(0,0) = A;
	F(1,0) = B;
		fields store arbitrary objects, such as matrices

---

example program

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main()
  {
  mat A = randu<mat>(4,5);
  mat B = randu<mat>(4,5);

  cout << A*B.t() << endl;

  return 0;
  }
```

- If you save the above program as *example.cpp*, under Linux and Mac OS X it can be compiled using:
  `g++ example.cpp -o example -O2 -larmadillo`
- As Armadillo is a template library, we strongly recommend to enable optimisation when compiling programs (eg. when compiling with GCC or clang, use the -O2 or -O3 options)
- See also the example program that comes with the Armadillo archive

config.hpp

- Armadillo can be configured via editing the file *include/armadillo_bits/config.hpp*. Specific functionality can be enabled or disabled by uncommenting or commenting out a particular *#define*, listed below.

`ARMA_DONT_USE_WRAPPER`	Disable going through the run-time Armadillo wrapper library (*libarmadillo.so*) when calling LAPACK, BLAS, ARPACK, SuperLU and HDF5 functions. You will need to directly link with BLAS, LAPACK, etc (eg. `-lblas -llapack`)
`ARMA_USE_LAPACK`	Enable use of LAPACK, or a high-speed replacement for LAPACK (eg. Intel MKL, AMD ACML or the Accelerate framework). Armadillo requires LAPACK for functions such as svd(), inv(), eig_sym(), solve(), etc.
`ARMA_DONT_USE_LAPACK`	Disable use of LAPACK; overrides *ARMA_USE_LAPACK*
`ARMA_USE_BLAS`	Enable use of BLAS, or a high-speed replacement for BLAS (eg. OpenBLAS, Intel MKL, AMD ACML or the Accelerate framework). BLAS is used for matrix multiplication. Without BLAS, Armadillo will use a built-in matrix multiplication routine, which might be slower for large matrices.
`ARMA_DONT_USE_BLAS`	Disable use of BLAS; overrides *ARMA_USE_BLAS*
`ARMA_USE_NEWARP`	Enable use of the built-in reimplementation of ARPACK (Armadillo 7.x and later versions). This is used for the eigen decomposition of real (non-complex) sparse matrices, ie. eigs_gen(), eigs_sym() and svds(). Requires ARMA_USE_LAPACK to be enabled.
`ARMA_DONT_USE_NEWARP`	Disable use of the built-in reimplementation of ARPACK; overrides *ARMA_USE_NEWARP*
`ARMA_USE_ARPACK`	Enable use of ARPACK, or a high-speed replacement for ARPACK. Armadillo requires ARPACK for the eigen decomposition of complex sparse matrices, ie. eigs_gen(), eigs_sym() and svds()
`ARMA_DONT_USE_ARPACK`	Disable use of ARPACK; overrides *ARMA_USE_ARPACK*
`ARMA_USE_SUPERLU`	Enable use of SuperLU, which is used by spsolve() for finding the solutions of sparse systems; you will need to link with the superlu library, for example `-lsuperlu` **Caveat:** Armadillo 7.x and later versions require SuperLU 5.2
`ARMA_DONT_USE_SUPERLU`	Disable use of SuperLU; overrides *ARMA_USE_SUPERLU*

`ARMA_USE_HDF5`	Enable the ability to save and load matrices stored in the HDF5 format; the *hdf5.h* header file must be available on your system and you will need to link with the hdf5 library (eg. `-lhdf5`)
`ARMA_DONT_USE_HDF5`	Disable the use of the HDF5 library; overrides *ARMA_USE_HDF5*
`ARMA_USE_CXX11`	Use C++11 features, such as initialiser lists; automatically enabled when using a compiler in C++11 or C++14 mode, for example: `g++ -std=c++11`
`ARMA_DONT_USE_CXX11`	Disable use of all C++11 features; overrides *ARMA_USE_CXX11*
`ARMA_DONT_USE_CXX11_MUTEX`	Disable use of only *std::mutex* in C++11; applicable if your compiler and/or environment doesn't support *std::mutex*
`ARMA_DONT_OPTIMISE_BAND`	Disable automatically optimised handling of band matrices by solve() and chol()
`ARMA_DONT_OPTIMISE_SYMPD`	Disable automatically optimised handling of symmetric/hermitian positive definite matrices by solve(), inv(), expmat(), logmat(), sqrtmat(), powmat(), rcond()
`ARMA_USE_OPENMP`	Use OpenMP for parallelisation of computationally expensive element-wise operations (such as exp(), log(), cos(), etc). Automatically enabled when using a C++11/C++14 compiler which has OpenMP 3.1+ active (eg. the `-fopenmp` option for gcc and clang). **Caveat:** when using gcc, use of `-march=native` in conjunction with `-fopenmp` may lead to speed regressions on recent processors.
`ARMA_DONT_USE_OPENMP`	Disable use of OpenMP for parallelisation of element-wise operations; overrides *ARMA_USE_OPENMP*
`ARMA_OPENMP_THRESHOLD`	The minimum number of elements in a matrix to enable OpenMP based parallelisation of computationally expensive element-wise functions; default value is 240
`ARMA_OPENMP_THREADS`	The maximum number of threads for OpenMP based parallelisation of computationally expensive element-wise functions; default value is 10
`ARMA_BLAS_CAPITALS`	Use capitalised (uppercase) BLAS and LAPACK function names (eg. DGEMM vs dgemm)
`ARMA_BLAS_UNDERSCORE`	Append an underscore to BLAS and LAPACK function names (eg. dgemm_ vs dgemm). Enabled by default.
`ARMA_BLAS_LONG`	Use "long" instead of "int" when calling BLAS and LAPACK functions

`ARMA_BLAS_LONG_LONG`	Use "long long" instead of "int" when calling BLAS and LAPACK functions
`ARMA_USE_FORTRAN_HIDDEN_ARGS`	Use so-called "hidden arguments" when calling BLAS and LAPACK functions. Enabled by default. See Fortran argument passing conventions for more details.
`ARMA_DONT_USE_FORTRAN_HIDDEN_ARGS`	Disable use of so-called "hidden arguments" when calling BLAS and LAPACK functions. May be necessary when using Armadillo in conjunction with broken MKL headers (eg. if you have `#include "mkl_lapack.h"` in your code).
`ARMA_USE_TBB_ALLOC`	Use Intel TBB *scalable_malloc()* and *scalable_free()* instead of standard *malloc()* and *free()* for managing matrix memory
`ARMA_USE_MKL_ALLOC`	Use Intel MKL *mkl_malloc()* and *mkl_free()* instead of standard *malloc()* and *free()* for managing matrix memory
`ARMA_USE_MKL_TYPES`	Use Intel MKL types for complex numbers. You will need to include appropriate MKL headers before the Armadillo header. You may also need to enable one or more of the following options: `ARMA_BLAS_LONG`, `ARMA_BLAS_LONG_LONG`, `ARMA_DONT_USE_FORTRAN_HIDDEN_ARGS`
`ARMA_64BIT_WORD`	Use 64 bit integers. Automatically enabled when using a C++11 compiler on 64-bit platforms. Useful if you require matrices/vectors capable of holding more than 4 billion elements. Your machine and compiler must have support for 64 bit integers (eg. via "long" or "long long"). This can also be enabled by adding *#define ARMA_64BIT_WORD* before each instance of *#include <armadillo>*
`ARMA_NO_DEBUG`	Disable all run-time checks, such as bounds checking. This will result in faster code, but you first need to make sure that your code runs correctly! We strongly recommend to have the run-time checks enabled during development, as this greatly aids in finding mistakes in your code, and hence speeds up development. We recommend that run-time checks be disabled **only** for the shipped version of your program (ie. final release build).
`ARMA_EXTRA_DEBUG`	Print out the trace of internal functions used for evaluating expressions. Not recommended for normal use. This is mainly useful for debugging the library.
`ARMA_MAT_PREALLOC`	The number of pre-allocated elements used by matrices and vectors. Must be always enabled and set to an integer that is at least 1. By default set to 16. If you mainly use lots of very small vectors (eg. ≤ 4 elements), change the number to the size of your vectors.

ARMA_COUT_STREAM	The default stream used for printing matrices and cubes by .print(). Must be always enabled. By default defined to *std::cout*
ARMA_CERR_STREAM	The default stream used for printing error messages and warnings. Must be always enabled. By default defined to *std::cerr*
ARMA_PRINT_ERRORS	Print errors and warnings encountered during program execution
ARMA_DONT_PRINT_ERRORS	Do not print errors or warnings; overrides *ARMA_PRINT_ERRORS*

- See also:
  - logging of warnings and errors
  - element access
  - element initialisation
  - uword/sword

History of API Additions, Changes and Deprecations

- API Stability and Versioning:
  - Each release of Armadillo has its public API (functions, classes, constants) described in the accompanying API documentation specific to that release.
  - Each release of Armadillo has its full version specified as *A.B.C*, where *A* is a major version number, *B* is a minor version number, and *C* is a patch level (indicating bug fixes).
  - Within a major version (eg. 7), each minor version has a public API that strongly strives to be **backwards compatible** (at the source level) with the public API of preceding minor versions. For example, user code written for version 7.100 should work with version 7.200, 7.300, 7.400, etc. However, as later minor versions may have more features (API extensions) than preceding minor versions, user code specifically written for version 7.400 may not work with 7.300.
  - An increase in the patch level, while the major and minor versions are retained, indicates modifications to the code and/or documentation which aim to fix bugs without altering the public API.
  - We don't like changes to existing public API and strongly prefer not to break any user software. However, to allow evolution, we reserve the right to alter the public API in future major versions of Armadillo while remaining backwards compatible in as many cases as possible (eg. major version 8 may have slightly different public API than major version 7).
  - **Caveat:** any function, class, constant or other code not explicitly described in the public API documentation is considered as part of the underlying internal implementation details, and may be removed or changed without notice. (In other words, don't use internal functionality).
- List of additions and changes for each version:
  - Version 9.900:

- faster solve() for under/over-determined systems
- faster eig_gen() and eig_pair() for large matrices
- faster handling of matrix multiplication expressions by diagvec() and diagmat()
- faster handling of relational expressions by accu()
- faster handling of sympd matrices by expmat(), logmat(), sqrtmat()
- faster access to columns in sparse submatrix views
- added quantile()
- added powmat()
- added trimatu_ind() and trimatl_ind()
- added log_normpdf()
- added .is_zero()
- added `ARMA_DONT_USE_CXX11_MUTEX` configuration option to disable use of *std::mutex*
- expanded eig_gen() and eig_pair() to optionally provide left and right eigenvectors
- expanded qr() to optionally use pivoted decomposition
- expanded .save() and .load() to handle CSV files with headers via *csv_name(filename,header)* specification
- more consistent detection of sparse vector expressions
- updated physical constants to NIST 2018 CODATA values
- workaround for save/load issues with HDF5 v1.12

◦ Version 9.800:
- faster solve() in default operation; iterative refinement is no longer applied by default; use `solve_opts::refine` to explicitly enable refinement
- faster expmat()
- faster handling of triangular matrices by rcond()
- added .front() and .back()
- added .is_trimatu() and .is_trimatl()
- added .is_diagmat()

◦ Version 9.700:
- faster handling of cubes by vectorise()
- faster handling of sparse matrices by nonzeros()
- faster row-wise index_min() and index_max()
- expanded join_rows() and join_cols() to handle joining up to 4 matrices
- expanded .save() and .load() to allow storing sparse matrices in CSV format
- added randperm() to generate a vector with random permutation of a sequence of integers

◦ Version 9.600:
- faster handling of sparse submatrices
- faster handling of sparse diagonal views
- faster handling of sparse matrices by symmatu() and symmatl()
- faster handling of sparse matrices by join_cols()
- expanded clamp() to handle sparse matrices
- added .clean() to replace elements below a threshold with zeros

◦ Version 9.500:
- expanded solve() with `solve_opts::likely_sympd` to indicate that the given matrix is likely positive definite
- more robust automatic detection of positive definite matrices by solve() and inv()
- faster handling of sparse submatrices
- expanded eigs_sym() to print a warning if the given matrix is not symmetric
- extended LAPACK function prototypes to follow Fortran passing conventions for so-called "hidden arguments", in order to address GCC Bug 90329;

to use previous LAPACK function prototypes without the "hidden arguments", `#define ARMA_DONT_USE_FORTRAN_HIDDEN_ARGS` before `#include <armadillo>`

- Version 9.400:
  - faster cov() and cor()
  - added .as_col() and .as_row()
  - expanded .shed_rows() / .shed_cols() / .shed_slices() to remove rows/columns/slices specified in a vector
  - expanded vectorise() to handle sparse matrices
  - expanded element-wise versions of max() and min() to handle sparse matrices
  - optimised handling of sparse matrix expressions: sparse % (sparse +- scalar) and sparse / (sparse +- scalar)
  - expanded eig_sym(), chol(), expmat_sym(), logmat_sympd(), sqrtmat_sympd(), inv_sympd() to print a warning if the given matrix is not symmetric
  - more consistent detection of vector expressions

- Version 9.300:
  - faster handling of compound complex matrix expressions by trace()
  - more efficient handling of element access for inplace modifications in sparse matrices
  - added .is_sympd() to check whether a matrix is symmetric/hermitian positive definite
  - added interp2() for 2D data interpolation
  - added expm1() and log1p()
  - expanded .is_sorted() with options `"strictascend"` and `"strictdescend"`
  - expanded eig_gen() to optionally perform balancing prior to decomposition

- Version 9.200:
  - faster handling of symmetric positive definite matrices by rcond()
  - faster transpose of matrices with size ≥ 512x512
  - faster handling of compound sparse matrix expressions by accu(), diagmat(), trace()
  - faster handling of sparse matrices by join_rows()
  - added sinc()
  - expanded sign() to handle scalar arguments
  - expanded operators (*, %, +, –) to handle sparse matrices with differing element types (eg. multiplication of complex matrix by real matrix)
  - expanded conv_to() to allow conversion between sparse matrices with differing element types
  - expanded solve() to optionally allow keeping solutions of systems singular to working precision

- Version 9.100:
  - faster handling of symmetric/hermitian positive definite matrices by solve()
  - faster handling of inv_sympd() in compound expressions
  - added .is_symmetric()
  - added .is_hermitian()
  - expanded spsolve() to optionally allow keeping solutions of systems singular to working precision
  - new configuration options ARMA_OPTIMISE_SOLVE_BAND and ARMA_OPTIMISE_SOLVE_SYMPD
  - smarter use of the element cache in sparse matrices

- Version 8.600:
  - added hess() for Hessenberg decomposition
  - added .row(), .rows(), .col(), .cols() to subcube views
  - expanded .shed_rows() and .shed_cols() to handle cubes
  - expanded .insert_rows() and .insert_cols() to handle cubes
  - expanded subcube views to allow non-contiguous access to slices
  - improved tuning of sparse matrix element access operators

- faster handling of tridiagonal matrices by solve()
- faster multiplication of matrices with differing element types when using OpenMP

- Version 8.500:
  - faster handling of sparse matrices by kron() and repmat()
  - faster transpose of sparse matrices
  - faster element access in sparse matrices
  - faster row iterators for sparse matrices
  - faster handling of compound expressions by trace()
  - more efficient handling of aliasing in submatrix views
  - expanded normalise() to handle sparse matrices
  - expanded .transform() and .for_each() to handle sparse matrices
  - added reverse() for reversing order of elements
  - added repelem() for replicating elements
  - added roots() for finding the roots of a polynomial

- Version 8.400:
  - faster handling of sparse matrices by repmat()
  - faster loading of CSV files
  - expanded kron() to handle sparse matrices
  - expanded index_min() and index_max() to handle cubes
  - expanded randi(), randu(), randn(), randg() to output single scalars
  - added submatrix & subcube iterators
  - added normcdf()
  - added mvnrnd()
  - added chi2rnd()
  - added wishrnd() and iwishrnd()

- Version 8.300:
  - faster handling of band matrices by solve()
  - faster handling of band matrices by chol()
  - faster randg() when using OpenMP
  - added normpdf()
  - expanded .save() to allow appending new datasets to existing HDF5 files

- Version 8.200:
  - added intersect() for finding common elements in two vectors/matrices
  - expanded affmul() to handle non-square matrices

- Version 8.100:
  - faster incremental construction of sparse matrices via element access operators
  - faster diagonal views in sparse matrices
  - expanded SpMat to save/load sparse matrices in coord format
  - expanded .save()/.load() to allow specification of datasets within HDF5 files
  - added affmul() to simplify application of affine transformations
  - warnings and errors are now printed by default to the *std::cerr* stream
  - added set_cerr_stream() and get_cerr_stream() to replace *set_stream_err1()*, *set_stream_err2()*, *get_stream_err1()*, *get_stream_err2()*
  - new configuration options ARMA_COUT_STREAM and ARMA_CERR_STREAM

- Version 7.960:
  - faster randn() when using OpenMP
  - faster gmm_diag class, for Gaussian mixture models with diagonal covariance matrices

- added *.sum_log_p()* to the gmm_diag class
- added gmm_full class, for Gaussian mixture models with full covariance matrices
- expanded .each_slice() to optionally use OpenMP for multi-threaded execution

◦ Version 7.950:
- expanded accu() and sum() to use OpenMP for processing expressions with computationally expensive element-wise functions
- expanded trimatu() and trimatl() to allow specification of the diagonal which delineates the boundary of the triangular part

◦ Version 7.900:
- expanded clamp() to handle cubes
- computationally expensive element-wise functions (such as exp(), log(), cos(), etc) can now be automatically sped up via OpenMP; this requires a C++11/C++14 compiler with OpenMP 3.1+ support
  - for GCC and clang compilers use the following options to enable both C++11 and OpenMP: `-std=c++11 -fopenmp`
  - **Caveat:** when using GCC, use of `-march=native` in conjunction with `-fopenmp` may lead to speed regressions on recent processors

◦ Version 7.800:
- changed license to the permissive Apache License 2.0; see the Questions page for more info

◦ Version 7.700:
- added polyfit() and polyval()
- added second form of log_det() to directly return the result as a complex number
- added range() to statistics functions
- expanded trimatu()/trimatl() and symmatu()/symmatl() to handle sparse matrices

◦ Version 7.600:
- more accurate eigs_sym() and eigs_gen()
- expanded floor(), ceil(), round(), trunc(), sign() to handle sparse matrices
- added arg(), atan2(), hypot()

◦ Version 7.500:
- expanded qz() to optionally specify ordering of the Schur form
- expanded .each_slice() to support matrix multiplication

◦ Version 7.400:
- added expmat_sym(), logmat_sympd(), sqrtmat_sympd()
- added .replace()

◦ Version 7.300:
- added index_min() and index_max() standalone functions
- expanded .subvec() to accept size() arguments
- more robust handling of non-square matrices by lu()

◦ Version 7.200:
- added .index_min() and .index_max() member functions
- expanded ind2sub() to handle vectors of indices
- expanded sub2ind() to handle matrix of subscripts
- expanded expmat(), logmat() and sqrtmat() to optionally return a bool indicating success
- faster handling of compound expressions by vectorise()

- Version 7.100:
  - added erf(), erfc(), lgamma()
  - added .head_slices() and .tail_slices() to subcube views
  - spsolve() now requires SuperLU 5.2
  - eigs_sym(), eigs_gen() and svds() now use a built-in reimplementation of ARPACK for real (non-complex) matrices; contributed by Yixuan Qiu

- Version 6.700:
  - added trapz() for numerical integration
  - added logmat() for calculating the matrix logarithm
  - added regspace() for generating vectors with regularly spaced elements
  - added logspace() for generating vectors with logarithmically spaced elements
  - added approx_equal() for determining approximate equality

- Version 6.600:
  - expanded sum(), mean(), min(), max() to handle cubes
  - expanded Cube class to handle arbitrarily sized empty cubes (eg. 0x5x2)
  - added shift() for circular shifts of elements
  - added sqrtmat() for finding the square root of a matrix

- Version 6.500:
  - added conv2() for 2D convolution
  - added stand-alone kmeans() function for clustering data
  - added trunc()
  - extended conv() to optionally provide central convolution
  - faster handling of multiply-and-accumulate by accu() when using Intel MKL, ATLAS or OpenBLAS

- Version 6.400:
  - expanded each_col(), each_row() and each_slice() to handle C++11 lambda functions
  - added ind2sub() and sub2ind()

- Version 6.300:
  - expanded solve() to find approximate solutions for rank-deficient systems
  - faster handling of non-contiguous submatrix views in compound expressions
  - added .for_each() to *Mat*, *Row*, *Col*, *Cube* and *field* classes
  - added rcond() for estimating the reciprocal condition number

- Version 6.200:
  - expanded diagmat() to handle non-square matrices and arbitrary diagonals
  - expanded trace() to handle non-square matrices

- Version 6.100:
  - faster norm() and normalise() when using Intel MKL, ATLAS or OpenBLAS
  - added Schur decomposition: schur()
  - stricter handling of matrix objects by hist() and histc()
  - advanced constructors for using auxiliary memory by Mat, Col, Row and Cube now have the default of *strict = false*
  - Cube class now delays allocation of .slice() related structures until needed
  - expanded join_slices() to handle joining cubes with matrices

- Version 5.600:
  - added .each_slice() for repeated matrix operations on each slice of a cube

- expanded .each_col() and .each_row() to handle out-of-place operations

- Version 5.500:
  - expanded object constructors and generators to handle size() based specification of dimensions

- Version 5.400:
  - added find_unique() for finding indices of unique values
  - added diff() for calculating differences between consecutive elements
  - added cumprod() for calculating cumulative product
  - added null() for finding the orthonormal basis of null space
  - expanded interp1() to handle repeated locations
  - expanded unique() to handle complex numbers
  - faster flipud()
  - faster row-wise cumsum()

- Version 5.300:
  - added generalised Schur decomposition: qz()
  - added .has_inf() and .has_nan()
  - expanded interp1() to handle out-of-domain locations
  - expanded sparse matrix class with .set_imag() and .set_real()
  - expanded imag(), real() and conj() to handle sparse matrices
  - expanded diagmat(), reshape() and resize() to handle sparse matrices
  - faster sparse sum()
  - faster row-wise sum(), mean(), min(), max()
  - updated physical constants to NIST 2014 CODATA values

- Version 5.200:
  - added orth() for finding the orthonormal basis of the range space of a matrix
  - expanded element initialisation to handle nested initialiser lists (C++11)

- Version 5.100:
  - added interp1() for 1D interpolation
  - added .is_sorted() for checking whether a vector or matrix has sorted elements
  - updated physical constants to NIST 2010 CODATA values

- Version 5.000:
  - added spsolve() for solving sparse systems of linear equations
  - added svds() for singular value decomposition of sparse matrices
  - added nonzeros() for extracting non-zero values from matrices
  - added handling of diagonal views by sparse matrices
  - expanded repmat() to handle sparse matrices
  - expanded join_rows() and join_cols() to handle sparse matrices
  - sort_index() and stable_sort_index() have been placed in the delayed operations framework for increased efficiency
  - use of 64 bit integers is automatically enabled when using a C++11 compiler

- Version 4.650:
  - added randg() for generating random values from gamma distributions (C++11 only)
  - added *.head_rows()* and *.tail_rows()* to submatrix views
  - added *.head_cols()* and *.tail_cols()* to submatrix views
  - expanded eigs_sym() to optionally calculate eigenvalues with smallest/largest algebraic values

- Version 4.600:

- added *.head()* and *.tail()* to submatrix views
- faster matrix transposes within compound expressions
- faster in-place matrix multiplication
- faster accu() and norm() when compiling with *-O3 -ffast-math -march=native* (gcc and clang)

- Version 4.550:
  - added matrix exponential function: expmat()
  - faster *.log_p()* and *.avg_log_p()* functions in the gmm_diag class when compiling with OpenMP enabled
  - faster handling of in-place addition/subtraction of expressions with an outer product

- Version 4.500:
  - faster handling of complex vectors by norm()
  - expanded chol() to optionally specify output matrix as upper or lower triangular
  - better handling of non-finite values when saving matrices as text files

- Version 4.450:
  - faster handling of matrix transposes within compound expressions
  - expanded symmatu()/symmatl() to optionally disable taking the complex conjugate of elements
  - expanded sort_index() to handle complex vectors
  - expanded the gmm_diag class with functions to generate random samples

- Version 4.400:
  - faster handling of subvectors by dot()
  - faster handling of aliasing by submatrix views
  - added clamp() for clamping values to be between lower and upper limits
  - added gmm_diag class for statistical modelling of data using Gaussian Mixture Models
  - expanded batch insertion constructors for sparse matrices to add values at repeated locations

- Version 4.320:
  - expanded eigs_sym() and eigs_gen() to use an optional tolerance parameter
  - expanded eig_sym() to automatically fall back to standard decomposition method if divide-and-conquer fails
  - cmake-based installer enables use of C++11 random number generator when using gcc 4.8.3+ in C++11 mode

- Version 4.300:
  - added find_finite() and find_nonfinite()
  - expressions *X=inv(A)*B*C* and *X=A.i()*B*C* are automatically converted to *X=solve(A,B*C)*

- Version 4.200:
  - faster transpose of sparse matrices
  - more efficient handling of aliasing during matrix multiplication
  - faster inverse of matrices marked as diagonal

- Version 4.100:
  - added normalise() for normalising vectors to unit *p*-norm
  - extended the field class to handle 3D layout
  - extended eigs_sym() and eigs_gen() to obtain eigenvalues of various forms (eg. largest or smallest magnitude)
  - automatic SIMD vectorisation of elementary expressions (eg. matrix addition) when using Clang 3.4+ with -O3 optimisation
  - faster handling of sparse submatrix views

- Version 4.000:
  - added eigen decompositions of sparse matrices: eigs_sym() and eigs_gen()
  - added eigen decomposition for pair of matrices: eig_pair()
  - added simpler forms of eig_gen()
  - added condition number of matrices: cond()
  - expanded find() to handle cubes
  - expanded subcube views to access elements specified in a vector
  - template argument for running_stat_vec expanded to accept vector types
  - more robust fast inverse of 4x4 matrices
  - faster divide-and-conquer decompositions are now used by default for eig_sym(), pinv(), princomp(), rank(), svd(), svd_econ()
  - the form *inv(sympd(X))* no longer assumes that *X* is positive definite; use inv_sympd() instead

- Version 3.930:
  - added size() based specifications of submatrix view sizes
  - added element-wise variants of min() and max()
  - added divide-and-conquer variant of svd_econ()
  - added divide-and-conquer variant of pinv()
  - added randi() for generating matrices with random integer values
  - added inplace_trans() for memory efficient in-place transposes
  - added more intuitive specification of sort direction in sort() and sort_index()
  - added more intuitive specification of method in det(), .i(), inv() and solve()
  - more precise timer for the wall_clock class when using C++11

- Version 3.920:
  - faster .zeros()
  - faster round(), exp2() and log2() when using C++11
  - added signum function: sign()
  - added move constructors when using C++11
  - added 2D fast Fourier transform: fft2()
  - added .tube() for easier extraction of vectors and subcubes from cubes
  - added specification of a fill type during construction of Mat, Col, Row and Cube classes, eg. *mat X(4, 5, fill::zeros)*

- Version 3.910:
  - faster multiplication of a matrix with a transpose of itself, ie. *X*X.t()* and *X.t()*X*
  - added vectorise() for reshaping matrices into vectors
  - added all() and any() for indicating presence of elements satisfying a relational condition

- Version 3.900:
  - added automatic SSE2 vectorisation of elementary expressions (eg. matrix addition) when using GCC 4.7+ with -O3 optimisation
  - faster median()
  - faster handling of compound expressions with transposes of submatrix rows
  - faster handling of compound expressions with transposes of complex vectors
  - added support for saving & loading of cubes in HDF5 format

- Version 3.820:
  - faster as_scalar() for compound expressions
  - faster transpose of small vectors
  - faster matrix-vector product for small vectors
  - faster multiplication of small fixed size matrices

- Version 3.810:
  - added fast Fourier transform: fft()
  - added handling of .imbue() and .transform() by submatrices and subcubes
  - added batch insertion constructors for sparse matrices

- Version 3.800:
  - added .imbue() for filling a matrix/cube with values provided by a functor or lambda expression
  - added .swap() for swapping contents with another matrix
  - added .transform() for transforming a matrix/cube using a functor or lambda expression
  - added round() for rounding matrix elements towards nearest integer
  - faster find()
  - changed license to the Mozilla Public License 2.0

- Version 3.6:
  - faster handling of compound expressions with submatrices and subcubes
  - faster trace()
  - added support for loading matrices as text files with *NaN* and *Inf* elements
  - added stable_sort_index(), which preserves the relative order of elements with equivalent values
  - added handling of sparse matrices by mean(), var(), norm(), abs(), square(), sqrt()
  - added saving and loading of sparse matrices in *arma_binary* format

- Version 3.4:
  - added economical QR decomposition: qr_econ()
  - added .each_col() & .each_row() for vector operations repeated on each column or row of a matrix
  - added preliminary support for sparse matrices
  - added ability to save and load matrices in HDF5 format
  - faster singular value decomposition via optional use of divide-and-conquer algorithm
  - faster .randn()
  - faster dot() and cdot() for complex numbers

- Version 3.2:
  - added unique(), for finding unique elements of a matrix
  - added .eval(), for forcing the evaluation of delayed expressions
  - faster eigen decomposition via optional use of divide-and-conquer algorithm
  - faster transpose of vectors and compound expressions
  - faster handling of diagonal views
  - faster handling of tiny fixed size vectors (≤ 4 elements)

- Version 3.0:
  - added shorthand for inverse: .i()
  - added datum class
  - added hist() and histc()
  - added non-contiguous submatrix views
  - faster handling of submatrix views with a single row or column
  - faster element access in fixed size matrices
  - faster repmat()
  - expressions *X=inv(A)*B* and *X=A.i()*B* are automatically converted to *X=solve(A,B)*
  - better detection of vector expressions by sum(), cumsum(), prod(), min(), max(), mean(), median(), stddev(), var()
  - faster generation of random numbers (eg. randu() and randn()), via an algorithm that produces slightly different numbers than in 2.x
  - support for tying writable auxiliary (external) memory to fixed size matrices has been removed;

instead, you can use standard matrices with writable auxiliary memory, or initialise fixed size matrices by copying the memory; using auxiliary memory with standard matrices is unaffected
- *.print_trans()* and *.raw_print_trans()* have been removed; instead, you can chain *.t()* and *.print()* to achieve a similar result: *X.t().print()*

◦ Version 2.4:
- added shorter forms of transposes: .t() and .st()
- added .resize() and resize()
- added optional use of 64 bit indices (allowing matrices to have more than 4 billion elements), enabled via ARMA_64BIT_WORD in *include/armadillo_bits/config.hpp*
- added experimental support for C++11 initialiser lists, enabled via ARMA_USE_CXX11 in *include/armadillo_bits/config.hpp*
- refactored code to eliminate warnings when using the Clang C++ compiler
- umat, uvec, .min() and .max() have been changed to use the *uword* type instead of the *u32* type; by default the *uword* and *u32* types are equivalent (ie. unsigned integer type with a minimum width 32 bits); however, when the use of 64 bit indices is enabled via ARMA_64BIT_WORD in *include/armadillo_bits/config.hpp*, the *uword* type then has a minimum width of 64 bits

◦ Version 2.2:
- added svd_econ()
- added circ_toeplitz()
- added .is_colvec() and .is_rowvec()

◦ Version 2.0:
- det(), inv() and solve() can be forced to use more precise algorithms for tiny matrices (≤ 4x4)
- added syl(), for solving Sylvester's equation
- added strans(), for transposing a complex matrix without taking the complex conjugate
- added symmatu() and symmatl()
- added submatrices of submatrices
- faster inverse of symmetric positive definite matrices
- faster element access for fixed size matrices
- faster multiplication of tiny matrices (eg. 4x4)
- faster compound expressions containing submatrices
- added handling of arbitrarily sized empty matrices (eg. 5x0)
- added .count() member function in running_stat and running_stat_vec
- added loading & saving of matrices as CSV text files
- trans() now takes the complex conjugate when transposing a complex matrix
- forms of chol(), eig_sym(), eig_gen(), inv(), lu(), pinv(), princomp(), qr(), solve(), svd(), syl() that do not return a bool indicating success now throw *std::runtime_error* exceptions when failures are detected
- princomp_cov() has been removed; eig_sym() in conjunction with cov() can be used instead
- .is_vec() now outputs *true* for empty vectors (eg. 0x1)
- set_log_stream() & get_log_stream() have been replaced by set_stream_err1() & get_stream_err1()

◦ Version 1.2:
- added .min() & .max() member functions of Mat and Cube
- added floor() and ceil()
- added representation of “not a number”: math::nan()
- added representation of infinity: math::inf()
- .in_range() expanded to use **span()** arguments
- fixed size matrices and vectors can use auxiliary (external) memory
- submatrices and subfields can be accessed via ***X(span(a,b), span(c,d))***
- subcubes can be accessed via ***X(span(a,b), span(c,d), span(e,f))***

- the two argument version of ***span*** can be replaced by ***span::all*** or ***span()***, to indicate an entire range
- for cubes, the two argument version of ***span*** can be replaced by a single argument version, ***span(a)***, to indicate a single column, row or slice
- arbitrary "flat" subcubes can be interpreted as matrices; for example:

```
cube Q = randu<cube>(5,3,4);
mat  A = Q( span(1), span(1,2), span::all );
// A has a size of 2x4

vec v = ones<vec>(4);
Q( span(1), span(1), span::all ) = v;
```

- added interpretation of matrices as triangular through trimatu() / trimatl()
- added explicit handling of triangular matrices by solve() and inv()
- extended syntax for submatrices, including access to elements whose indices are specified in a vector
- added ability to change the stream used for logging of errors and warnings
- added ability to save/load matrices in raw binary format
- added cumulative sum function: cumsum()

◦ Changed in 1.0 (compared to earlier 0.x development versions):

- the 3 argument version of lu(), eg. lu(L,U,X), provides L and U which should be the same as produced by Octave 3.2 (this was not the case in versions prior to 0.9.90)

- rand() has been replaced by randu(); this has been done to avoid confusion with std::rand(), which generates random numbers in a different interval

- In versions earlier than 0.9.0, some multiplication operations directly converted result matrices with a size of 1x1 into scalars. This is no longer the case. If you know the result of an expression will be a 1x1 matrix and wish to treat it as a pure scalar, use the as_scalar() wrapping function

- Almost all functions have been placed in the delayed operations framework (for speed purposes). This may affect code which assumed that the output of some functions was a pure matrix. The solution is easy, as explained below.

  In general, Armadillo queues operations before executing them. As such, the direct output of an operation or function cannot be assumed to be a directly accessible matrix. The queued operations are executed when the output needs to be stored in a matrix, eg. *mat B = trans(A)* or *mat B(trans(A))*. If you need to force the execution of the delayed operations, place the operation or function inside the corresponding Mat constructor. For example, if your code assumed that the output of some functions was a pure matrix, eg. *chol(m).diag()*, change the code to *mat(chol(m)).diag()*. Similarly, if you need to pass the result of an operation such as *A+B* to one of your own functions, use *my_function(mat(A+B))*.

[top]