Adept Quick Reference

All functions and types are placed in the `adept` namespace.

Header files

`adept.h`	Include if only scalar automatic differentiation is required
`adept_arrays.h`	Include if array capabilities are needed as well
`adept_fortran.h`	Interface to Fortran 2018 array descriptors
`adept_optimize.h`	Minimization algorithms, e.g. Levenberg-Marquardt
`adept_source.h`	Include entire Adept library, so linking to library not required

Scalar types

`Real`	Passive scalar type used for differentiation (usually `double`)
`aReal`	Active scalar of underlying type `Real`
`adouble`, `afloat`	Active scalars of underlying type `double` and `float`

Basic reverse-mode workflow

`Stack stack;`	Object to store derivative information
`aVector x = {1.0, 2.0};`	Initialize independent (input) variables (C++11)
`stack.new_recording();`	Start a new recording
`aReal J = algorithm(x);`	Any complicated algorithm here
`J.set_gradient(1.0);`	Seed adjoint of cost function
`stack.reverse();`	Perform reverse-mode differentiation
`Vector dJ_dx = x.get_gradient();`	Return gradients of output with respect to inputs

Basic Jacobian workflow

`Stack stack;`	Object to store derivative information
`aVector x = {1.0, 2.0};`	Initialize independent (input) variables (C++11)
`stack.new_recording();`	Start a new recording
`aVector y = algorithm(x);`	Algorithm with vector output
`stack.independent(x);`	Declare independent variables
`stack.dependent(y);`	Declare dependent variables
`Matrix dy_dx = stack.jacobian();`	Compute Jacobian matrix

`aReal` member functions

The first three functions below also work with active array arguments, where g would be of the equivalent passive array type:

`.set_gradient(g)`	Initialize gradient to g
`.get_gradient()`	After forward or reverse pass, return gradient
`.get_gradient(g)`	As above, but writing gradient to g
`.add_derivative_dependence(a,p)`	Add $\mathrm{p}\times\delta\mathrm{a}$ to the stack
`.append_derivative_dependence(a,p)`	Append $+\mathrm{p}\times\delta\mathrm{a}$ to the stack

`Stack` member functions

Constructors:

`Stack stack;`	Construct and activate immediately
`Stack stack(false);`	Construct in inactive state

Member functions:

`.new_recording()`	Clear any existing differential statements
`.pause_recording()`	Pause recording (`ADEPT_PAUSABLE_RECORDING` needed)
`.continue_recording()`	Continue recording
`.is_recording()`	Is Adept currently recording?
`.forward()`	Perform forward-mode differentiation
`.compute_tangent_linear()`	...as above
`.reverse()`	Perform reverse-mode differentiation
`.compute_adjoint()`	...as above
`.independent(x)`	Declare an independent variable (active scalar or array)
`.independent(xptr,n)`	Declare `n` independent scalar variables starting at `xptr`
`.dependent(y)`	Declare a dependent variable (active scalar or array)
`.dependent(yptr,n)`	Declare `n` dependent scalar variables starting at `yptr`
`.jacobian()`	Return Jacobian matrix
`.jacobian(jacptr)`	Place Jacobian matrix into `jacptr` (column major)
`.jacobian(jacptr,false)`	Place Jacobian matrix into `jacptr` (row major)
`.clear_gradients()`	Clear gradients set with `set_gradient` function
`.clear_independents()`	Clear independent variables
`.clear_dependents()`	Clear dependent variables
`.n_independents()`	Number of independent variables declared
`.n_dependents()`	Number of dependent variables declared
`.print_status()`	Print status of `Stack` to standard output
`.print_statements()`	Print list of differential statements
`.print_gradients()`	Print current values of gradients
`.activate()`	Activate the stack
`.deactivate()`	Deactivate the stack
`.is_active()`	Is the stack currently active?
`.memory()`	Return number of bytes currently used
`.preallocate_statements(n)`	Preallocate space for `n` statements
`.preallocate_operations(n)`	Preallocate space for `n` operations

Query functions in `adept` namespace

`active_stack()`	Return pointer to currently active `Stack` object
`version()`	Return `std::string` with Adept version number
`configuration()`	Return `std::string` describing Adept configuration
`have_matrix_multiplication()`	Adept compiled with matrix multiplication (BLAS)?
`have_linear_algebra()`	Adept compiled with linear-algebra (LAPACK)?
`set_max_blas_threads(n)`	Set maximum threads for matrix operations
`max_blas_threads()`	Get maximum threads for matrix operations
`is_thread_unsafe()`	Global `Stack` object is *not* thread-local?

Dense dynamic array types

`Vector`, `Matrix`, `Array3D`, `Array4D`... `Array7D`	Arrays of type `Real`
`intVector`, `intMatrix`, `intArray3D`... `intArray7D`	Arrays of type `int`
`boolVector`, `boolMatrix`, `boolArray3D`... `boolArray7D`	Arrays of type `bool`
`floatVector`, `floatMatrix`, `floatArray3D`... `floatARray7D`	Arrays of type `float`
`aVector`, `aMatrix`, `aArray3D`... `aArray7D`	Active arrays of type `Real`

Define new dynamic array types as follows:

```
typedef Array<short,2,false> shortMatrix;
typedef Array<float,3,true> afloatArray3D;
```

Dense fixed-size array types

`Vector2`, `Vector3`, `Vector4`	Passive vectors of fixed length 2–4
`Matrix22`, `Matrix33`, `Matrix44`	Passive matrices of fixed size 2×2, 3×3, 4×4
`aVector2`, `aVector3`, `aVector4`	Active vectors of fixed length 2–4
`aMatrix22`, `aMatrix33`, `aMatrix44`	Active matrices of fixed size 2×2, 3×3, 4×4

Define new fixed array types as follows:

```
typedef FixedArray<short,false,2,4> shortMatrix24;
typedef FixedArray<Real,true,3,3,3> aArray333;
```

Special square matrix types

`SymmMatrix`, `aSymmMatrix`	Symmetric matrix
`DiagMatrix`, `aDiagMatrix`	Diagonal matrix
`TridiagMatrix`, `aTridiagMatrix`	Tridiagonal matrix
`PentadiagMatrix`, `aPentadiagMatrix`	Pentadiagonal matrix
`LowerMatrix`, `aLowerMatrix`	Lower-triangular matrix
`UpperMatrix`, `aUpperMatrix`	Upper-triangular matrix

Dense dynamic array constructors

`Matrix M;`	Create an empty matrix of type `Real`
`Matrix N(M);`	Create matrix sharing data with existing matrix
`Matrix N = M;`	...as above
`Matrix N(3,4);`	Create matrix with size 3×4
`Matrix N(dimensions(3,4));`	...as above
`Matrix N(M.dimensions());`	Create matrix with the same size as `M`
`Matrix N(ptr,dimensions(3,4));`	Create 3×4 matrix sharing data from pointer `ptr`
`Matrix N = log(M);`	Create matrix containing copy of right-hand-side
`Matrix N = {{1.0,2.0},{3.0,4.0}};`	Create 2×2 matrix from initializer list (C++11)

Array resize and link member functions

`.clear()`	Return array to original empty state
`.resize(3,4)`	Resize array discarding data
`.resize(dimensions(3,4))`	...as above
`.resize_row_major(3,4)`	Resize with row-major storage (default)
`.resize_column_major(3,4)`	Resize with column-major storage
`.resize(M.dimensions())`	Resize to same as `M`
`.resize_contiguous(...)`	Resize guaranteeing contiguous storage
`N >>= M;`	Discard existing data and link to array on right-hand-side

Array query member functions

`::rank`	Number of array dimensions
`.empty()`	Return `true` if array is empty, `false` otherwise
`.dimensions()`	Return an object that can be used to resize other arrays
`.dimension(i)`	Return length of dimension `i` (0 based)
`.size()`	Return total number of elements
`.data()`	Return pointer to underlying passive data
`.const_data()`	Return `const` pointer to underlying data

Array filling

`M = 1.0;`	Fill all elements of array with the same number
`M << 1.0, 2.0, 3.0, 4.0;`	Fill first four elements of array
`M = {{1.0,2.0},{3.0,4.0}};`	Fill 2×2 matrix (C++11)

Array indexing and slicing

Dense arrays can be indexed/sliced using the function-call operator with as many arguments as there are dimensions (e.g. index a matrix with `M(i,j)`). In all cases a slice can be used as an lvalue or rvalue. If all arguments are scalars then a single element of the array is extracted. The following special values are available:

`end`	The last element of the dimension being indexed
`end-1`	Penultimate element of indexed dimension (any integer arithmetic possible)

If one or more argument is a *regular index range* then the return type will be an `Array` pointing to part of the original array. For every scalar argument, its rank will be reduced by one compared to the original array. The available ranges are:

`__`	All elements of indexed dimension
`range(ibeg,iend)`	Contiguous range from `ibeg` to `iend`
`stride(ibeg,iend,istride)`	Strided range (`istride` can be negative but not zero)

If any of the arguments is a *irregular index range* (such as an `intVector` containing an arbitrary list of indices) then the return type will be an `IndexedArray`. If used as an lvalue, it will modify the original array, but if passed into a function receiving an `Array` type then any modifications inside the function will not affect the original array.

Passing arrays to and from functions

There are three ways an array can be received as an argument to a function:

`Matrix&`	For an array that might be resized in the function
`Matrix`	For an array or array slice to be modified inside the function
`const Matrix&`	For a read-only array, array slice or array expression

Member functions returning lvalue

The functions in this section return an `Array` that links to the original data and can be used on the left- or right-hand-side of an assignment. The following only work on dynamic or fixed-size dense arrays:

`.subset(ibeg0,iend0,ibeg1,iend1,...)`	Contiguous subset
`.permute(i0,i1,...)`	Permute dimensions
`.diag_matrix()`	For vector, return `DiagMatrix`
`.soft_link()`	

The following works on any matrix:

`.T()`	Transpose of matrix

The following work only with square matrices, including special square matrices

`.diag_vector()`	Return vector linked to its diagonals
`.diag_vector(i)`	Return vector linked to offdiagonal `i`
`.submatrix_on_diagonal(ibeg,iend)`	Return square matrix lying on diagonal

Elemental mathematical functions

Return passive part of active object: `value(x)`
Binary operators: `+`, `-`, `*` and `/`.
Assignment operators: `+=`, `-=`, `*=` and `/=`.
Unary functions: `sqrt`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `abs`, `asinh`, `acosh`, `atanh`, `expm1`, `log1p`, `cbrt`, `erf`, `erfc`, `exp2`, `log2`, `round`, `trunc`, `rint`, `nearbyint` and `fastexp`.
Binary functions: `pow`, `atan2`, `min`, `max`, `fmin` and `fmax`.
Unary functions returning `bool` expressions: `isfinite`, `isinf` and `isnan`.
Binary operators returning `bool` expressions: `==`, `!=`, `>`, `<`, `>=` and `<=`.

Alias-related functions

`eval(E)`	Avoid aliasing by evaluating expression `E` into an array
`noalias(E)`	Turn off alias checking for expression `E`

Reduction functions

`sum(M)`	Return the sum of all elements in `M`
`sum(M,i)`	Return array of rank one less than `M` containing sum along `i`th dimension (0 based)

Other reduction functions working in the same way: `mean`, `product`, `minval`, `maxval`, `norm2`.

`dot_product(x,y)`	The same as `sum(a*b)` for rank-1 arguments

Expansion functions

`spread<d>(M,n)`	Replicate `M` array expression `n` times along dimension `d`
`outer_product(x,y)`	Return rank-2 outer product from two rank-1 arguments

Matrix multiplication and linear algebra

`transpose(M)`	Transpose matrix or 2D matrix expression
`matmul(M,N)`	Matrix multiply, where at least one argument must be a matrix, and orientation of any vector arguments is inferred
`M ** N`	Shortcut for `matmul`; precedence is the same as normal multiply
`inv(M)`	Inverse of square matrix
`solve(A,x)`	Solve system of linear equations

Preprocessor variables

The following can be defined to change the behaviour of your code:

`ADEPT_STACK_THREAD_UNSAFE`	Thread-unsafe `Stack` (faster)
`ADEPT_RECORDING_PAUSABLE`	Recording can be paused (slower)
`ADEPT_NO_AUTOMATIC_DIFFERENTIATION`	Turn off differentiation
`ADEPT_TRACK_NON_FINITE_GRADIENTS`	Exception thrown if derivative non-finite
`ADEPT_BOUNDS_CHECKING`	Check array bounds (slower)
`ADEPT_NO_ALIAS_CHECKING`	Turn off alias checking (faster)
`ADEPT_NO_DIMENSION_CHECKING`	Turn off dimension checking (faster)
`ADEPT_INIT_REAL_SNAN`	Initialize real numbers to signaling NaN The
`ADEPT_INIT_REAL_ZERO`	Initialize real numbers to zero
`ADEPT_FAST_EXPONENTIAL`	Use faster vectorizable exponential
`ADEPT_FAST_SCALAR_EXPONENTIAL`	Provide faster `adept::exp` for scalars
`ADEPT_FAST`	Enable bit-reproducible options
`ADEPT_STORAGE_THREAD_SAFE`	Thread-safe array storage (slower)
`ADEPT_SUPPORT_HUGE_ARRAYS`	Use `std::size_t` for array dimensions
`ADEPT_REAL_TYPE_SIZE`	Size of `Real`: 4 or 8 (default 8)

`ADEPT_VERSION` variable contains version number as an integer, e.g. 20108, while `ADEPT_VERSION_STR` contains it as a string, e.g. "2.0.8".

Comparison of array syntax between Fortran 90 (and later), Matlab and the C++ libraries Adept and Eigen

	Fortran 90+	Matlab	C++ Adept (with C++11 features)	C++ Eigen
Maximum dimensions	7 (15 from Fortran 2008)	Unlimited	7	2
Vector declaration	real,dimension(:)		Vector	VectorXd
Matrix declaration	real,dimension(:,:)		Matrix	MatrixXd, ArrayXd
3D array declaration	real,dimension(:,:,:)		Array3D	
Fixed matrix declaration	real,dimension(M,N)		FixedMatrix<double,false,M,N>	Matrix<double,M,N>
Diagonal matrix declaration			DiagMatrix	DiagonalMatrix<double,Dynamic>
Symmetric matrix decl.			SymmMatrix	
Sparse matrix declaration				SparseMatrix<double>
Get rank	rank(A)	ndims(A)	A::rank	
Get total size	size(A)	numel(A)	A.size()	A.size()
Get size of dimension	size(A,i)	size(A,i)	A.size(i)	A.rows(), A.cols()
Get all dimensions	shape(A)	size(A)	A.dimensions()	
Resize	allocate(A(m,n))	A = zeros(m,n)	A.resize(m,n)	A.resize(m,n)
Clear	deallocate(A)	A = []	A.clear()	A.resize(0,0)
Link/associate	A => B		A >>= B	(Complicated)
Set elements to constant	A = x	A(:) = x	A = x	A.fill(x)
Fill vector with data	v = [0,1]	v = [0,1]	v << 0,1	v << 0,1
Fill matrix with data	A=reshape([0,1,2,3],[2,2])	A = [1 2; 3 4]	A << 1,2,3,4 or A = {{1,2},{3,4}}	A << 1,2,3,4
Vector literal	[1.0, 2.0]	[1.0 2.0]	Vector{1.0, 2.0}	
Vector subset	v(i1:i2)	v(i1:i2)	v.subset(i1,i2)	v.segment(i1,m)
Strided indexing	v(i1:i2:s)	v(i1:s:i2)	v(stride(i1,i2,s))	(Complicated)
Vector end indexing	v(i:)	v(i:end)	v.subset(i,end)	v.tail(n)
Index relative to end		v(end-1)	v(end-1)	
Index by int vector	v(index)	v(index)	v(index)	
Matrix subset	A(i1:i2,j1:j2)	A(i1:i2,j1:j2)	A.subset(i1,i2,j1,j2)	A.block(i1,j1,m,n)
Extract row	A(i,:)	A(i,:)	A(i,__), A[i]	A.row(i)
Matrix end block	M(i:,j:)	M(i:end,j:end)	M.subset(i,end,j,end)	M.bottomRightCorner(m,n)
Diagonal matrix from vector		diag(v)	v.diag_matrix()	v.asDiagonal()
Matrix diagonals as vector		diag(A)	A.diag_vector()	A.diagonal()
Matrix off-diagonals		diag(A,i)	A.diag_vector(i)	A.diagonal(i)
Elementwise multiplication	A * B	A .* B	A * B	A.array() * B.array()
Elemental function	sqrt(A)	sqrt(A)	sqrt(A)	A.array().sqrt()
Addition assignment	A = A + B	A = A + B	A += B	A.array() += B
Power	A ** B	A .^ C	pow(A,B)	A.array().pow(B)
Matrix multiplication	matmul(A,B)	A * B	A ** B	A * B
Dot product	dot_product(v,w)	dot(v,w)	dot_product(v,w)	v.dot(w)
Matrix transpose	transpose(A)	A'	A.T()	A.transpose()
In-place transpose			A.in_place_transpose()	A.transposeInPlace()
Matrix solve		A \ b	solve(A,b)	A.colPivHouseholderQr().solve(b)
Matrix inverse		inv(A)	inv(A)	A.inverse()
"Find" conditional assign		v(find(w<0)) = 0	v(find(w<0)) = 0	
"Where" conditional assign	where(w<0) v = 0		v.where(w<0) = 0	v = (w<0).select(0,v)
"Where" with both cases	...elsewhere v = 1		v.where(w<0)=either_or(0,1)	v = (w<0).select(0,1)
Average all elements	mean(A)	mean(A(:)	mean(A)	A.mean()
Average along dimension	mean(A,i)	mean(A,i)	mean(A,i)	A.colwise().mean()
Maximum of all elements	maxval(A)	max(A(:))	maxval(A)	A.maxCoeff()
Maximum of two arrays	max(A,B)	(Complicated)	max(A,B), fmax(A,B)	A.max(B)
Spread along new dimension	spread(A,dim,n)		spread<dim>(A,n)	