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 p×δa to the stack
- .append_derivative_dependence(a,p): Append +p×δa 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(a): 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_threadunsafe(): Global Stack object is not thread-local?
Dense dynamic array constructors

Vector, Matrix, Array3D, Array4D... Array7D
intVector, intMatrix, intArray3D... intArray7D
boolVector, boolMatrix, boolArray3D... boolArray7D
floatVector, floatMatrix, floatArray3D... floatArray7D
aVector, aMatrix, aArray3D... aArray7D

Define new dynamic array types as follows:
- Define new fixed array types as follows:
  - Dense fixed-size array types
    - Vector2, Vector3, Vector4
    - Matrix22, Matrix33, Matrix44
    - aVector2, aVector3, aVector4
    - aMatrix22, aMatrix33, aMatrix44

Define new fixed array types as follows:

- Special square matrix types
  - SymmMatrix, aSymmMatrix
  - DiagMatrix, aDiagMatrix
  - TridiagMatrix, aTridiagMatrix
  - PentadiagMatrix, aPentadiagMatrix
  - LowerMatrix, aLowerMatrix
  - UpperMatrix, aUpperMatrix

- Dense dynamic array constructors

  Matrix M;
  Matrix N(M);
  Matrix N = M;
  Matrix N(3,4);
  Matrix N(dimensions(3,4));
  Matrix N(M.dimensions());
  Matrix N(ptr, dimensions(3,4));
  Matrix N = log(M);
  Matrix N = {{1.0,2.0},{3.0,4.0}};

  Arrays of type Real
  Arrays of type int
  Arrays of type bool
  Arrays of type float
  Active arrays of type Real

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 constant 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(1,1)). 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:

- .range(ibeg,iend) All elements of indexed dimension
- .stride(ibeg,iend,istride) Strided range (istride can be negative but not zero)

If any of the arguments is an 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`, `asin`, `acos`, `atan`, `expm1`, `erf`, `erfc`, `exp`, `exp2`, `log`, `round`, `trunc`, `rint`, `nearbyint` and `fastexp`.

Binary functions: `pow`, `atan2`, `min`, `max`, `fmin` and `fmax`.

Binary 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 ith dimension (0 based)

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

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

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

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
- `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”.

The `adept::` namespace contains all the above functions and capabilities.
## Comparison of Array Syntax between Fortran 90 (and later), Matlab and the C++ Libraries Adept and Eigen

<table>
<thead>
<tr>
<th>Feature</th>
<th>Fortran 90+</th>
<th>Matlab</th>
<th>C++ Adept (with C++11 features)</th>
<th>C++ Eigen</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maximum dimensions</strong></td>
<td>7 (15 from Fortran 2008)</td>
<td>Unlimited</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td><strong>Vector declaration</strong></td>
<td>real, dimension(:)</td>
<td>Vector</td>
<td>VectorXd</td>
<td>ArrayXd</td>
</tr>
<tr>
<td><strong>Matrix declaration</strong></td>
<td>real, dimension(:, :)</td>
<td>Matrix</td>
<td>MatrixXd, ArrayXd</td>
<td>MatrixXd&lt; double,M,N&gt;</td>
</tr>
<tr>
<td><strong>3D array declaration</strong></td>
<td>real, dimension(:, :, :)</td>
<td>Array3D</td>
<td></td>
<td>DiagonalMatrix&lt; double,DYNAMIC&gt;</td>
</tr>
<tr>
<td><strong>Fixed matrix declaration</strong></td>
<td>real, dimension(M,N)</td>
<td>FixedMatrix&lt;double,false,M,N&gt;</td>
<td>DiagMatrix</td>
<td>SymmMatrix</td>
</tr>
<tr>
<td><strong>Diagonal matrix declaration</strong></td>
<td></td>
<td></td>
<td></td>
<td>SparseMatrix&lt; double&gt;</td>
</tr>
<tr>
<td><strong>Symmetric matrix decl.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sparse matrix declaration</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Get rank</strong></td>
<td>rank(A)</td>
<td>ndims(A)</td>
<td>A::rank</td>
<td>A.size()</td>
</tr>
<tr>
<td><strong>Get total size</strong></td>
<td>size(A)</td>
<td>numel(A)</td>
<td>A.size()</td>
<td>A. rows(), A.cols()</td>
</tr>
<tr>
<td><strong>Get size of dimension</strong></td>
<td>size(A,i)</td>
<td>size(A,i)</td>
<td>A.size(i)</td>
<td></td>
</tr>
<tr>
<td><strong>Get all dimensions</strong></td>
<td>shape(A)</td>
<td>size(A)</td>
<td>A.dimensions()</td>
<td></td>
</tr>
<tr>
<td><strong>Resize</strong></td>
<td>allocate(A(m,n))</td>
<td>A = zeros(m,n)</td>
<td>A.resize(m,n)</td>
<td>A.resize(m,n)</td>
</tr>
<tr>
<td><strong>Clear</strong></td>
<td>deallocate(A)</td>
<td>A = []</td>
<td>A.clear()</td>
<td>A.resize(0,0)</td>
</tr>
<tr>
<td><strong>Link/associate</strong></td>
<td>A =&gt; B</td>
<td>A = []</td>
<td>A &gt;&gt; = B</td>
<td>(Complicated)</td>
</tr>
<tr>
<td><strong>Set elements to constant</strong></td>
<td>A = x</td>
<td>A(:) = x</td>
<td>A = x</td>
<td>A.fill(x)</td>
</tr>
<tr>
<td><strong>Fill vector with data</strong></td>
<td>v = [0,1]</td>
<td>v = [0,1]</td>
<td>v &lt;= 0.1</td>
<td>v &lt;= 0.1</td>
</tr>
<tr>
<td><strong>Fill matrix with data</strong></td>
<td>A = reshape([0,1,2,3],[2,2])</td>
<td>A = [1 2; 3 4]</td>
<td>A &lt;= 1,2,3,4 or A = {{1,2},{3,4}}</td>
<td>A &lt;= 1,2,3,4</td>
</tr>
<tr>
<td><strong>Vector literal</strong></td>
<td>v(1:12)</td>
<td>v(1:12)</td>
<td>v.subset(1,12)</td>
<td>v.segment(11,m)</td>
</tr>
<tr>
<td><strong>Vector end indexing</strong></td>
<td>v(1:end)</td>
<td>v.subset(1:end)</td>
<td>v.subset(1, end)</td>
<td>(Complicated)</td>
</tr>
<tr>
<td><strong>Index relative to end</strong></td>
<td>v(end-1)</td>
<td>v(end-1)</td>
<td>v.end-1</td>
<td>v.tail(n)</td>
</tr>
<tr>
<td><strong>Index by int vector</strong></td>
<td>v(index)</td>
<td>v(index)</td>
<td>v(index)</td>
<td></td>
</tr>
<tr>
<td><strong>Matrix subset</strong></td>
<td>A(1:12,1:2)</td>
<td>A.subset(1:12,1:2)</td>
<td>A.block(1:12,1:2)</td>
<td>A.block(1:12,1:2)</td>
</tr>
<tr>
<td><strong>Extract row</strong></td>
<td>A(1,:)</td>
<td>A(1,:)</td>
<td>A.row(1)</td>
<td>A.row(i)</td>
</tr>
<tr>
<td><strong>Matrix end block</strong></td>
<td>M(:,1:)</td>
<td>M(:,1:)</td>
<td>M.subset(:,1)</td>
<td>M.bottomRightCorner(m,n)</td>
</tr>
<tr>
<td><strong>Diagonal matrix from vector</strong></td>
<td>diag(v)</td>
<td>v.diag_matrix()</td>
<td>v.asDiagonal()</td>
<td>v.asDiagonal()</td>
</tr>
<tr>
<td><strong>Matrix diagonals as vector</strong></td>
<td>diag(A)</td>
<td>A.diag_vector()</td>
<td>A.diag_vector()</td>
<td>A.diag_vector()</td>
</tr>
<tr>
<td><strong>Matrix off-diagonals</strong></td>
<td>diag(A,i)</td>
<td>A.diag_vector(i)</td>
<td>A.diag_vector(i)</td>
<td>A.diag_vector(i)</td>
</tr>
<tr>
<td><strong>Elementwise multiplication</strong></td>
<td>A * B</td>
<td>A * B</td>
<td>A * B</td>
<td>A.array() * B.array()</td>
</tr>
<tr>
<td><strong>Elemental function</strong></td>
<td>sqrt(A)</td>
<td>sqrt(A)</td>
<td>sqrt(A)</td>
<td>A.sqrt()</td>
</tr>
<tr>
<td><strong>Addition assignment</strong></td>
<td>A = A + B</td>
<td>A = A + B</td>
<td>A = A + B</td>
<td>A.array() += B</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>A ** B</td>
<td>A ** B</td>
<td>A ** B</td>
<td>A.array().pow(B)</td>
</tr>
<tr>
<td><strong>Matrix multiplication</strong></td>
<td>matmul(A,B)</td>
<td>A * B</td>
<td>A * B</td>
<td>A.array().matrix()</td>
</tr>
<tr>
<td><strong>Dot product</strong></td>
<td>dot_product(v,w)</td>
<td>dot(v,w)</td>
<td>dot_product(v,w)</td>
<td>v.dot(w)</td>
</tr>
<tr>
<td><strong>Matrix transpose</strong></td>
<td>transpose(A)</td>
<td>A.T()</td>
<td>A.transpose()</td>
<td>A.transpose()</td>
</tr>
<tr>
<td><strong>In-place transpose</strong></td>
<td>A.in_place_transpose()</td>
<td>A.in_place_transpose()</td>
<td>A.in_place_transpose()</td>
<td>A.inplaceTranspose()</td>
</tr>
<tr>
<td><strong>Matrix solve</strong></td>
<td>A \ b</td>
<td>solve(A,b)</td>
<td>solve(A,b)</td>
<td>A.colPivHouseholderQr().solve(b)</td>
</tr>
<tr>
<td><strong>Matrix inverse</strong></td>
<td>inv(A)</td>
<td>inv(A)</td>
<td>inv(A)</td>
<td>A.inverse()</td>
</tr>
<tr>
<td><strong>“Find” conditional assign</strong></td>
<td>v(find(w&lt;0)) = 0</td>
<td>v(find(w&lt;0)) = 0</td>
<td>v(find(w&lt;0)) = 0</td>
<td>v = (w&lt;0).select(0,v)</td>
</tr>
<tr>
<td><strong>“Where” conditional assign</strong></td>
<td>v.where(w&lt;0) = 0</td>
<td>v.where(w&lt;0) = 0</td>
<td>v.where(w&lt;0)=either_or(0,1)</td>
<td>v = (w&lt;0).select(0,1)</td>
</tr>
<tr>
<td><strong>Where” with both cases</strong></td>
<td>...elsewhere v = 1</td>
<td>v.where(w&lt;0)=either_or(0,1)</td>
<td>v.where(w&lt;0)=either_or(0,1)</td>
<td>v = (w&lt;0).select(0,1)</td>
</tr>
<tr>
<td><strong>Average all elements</strong></td>
<td>mean(A)</td>
<td>mean(A)</td>
<td>mean(A)</td>
<td>A.mean()</td>
</tr>
<tr>
<td><strong>Average along dimension</strong></td>
<td>mean(A,1)</td>
<td>mean(A,1)</td>
<td>mean(A,1)</td>
<td>A.colwise().mean()</td>
</tr>
<tr>
<td><strong>Maximum of all elements</strong></td>
<td>maxv1(A)</td>
<td>maxv1(A)</td>
<td>maxv1(A)</td>
<td>A.maxCoeff()</td>
</tr>
<tr>
<td><strong>Maximum of two arrays</strong></td>
<td>max(A,B)</td>
<td>maxv1(A,B)</td>
<td>max(A,B)</td>
<td>A.max(B)</td>
</tr>
<tr>
<td><strong>Spread along new dimension</strong></td>
<td>spread(A,dim,n)</td>
<td>(Complicated)</td>
<td>spread&lt;dim&gt;(A,n)</td>
<td></td>
</tr>
</tbody>
</table>