Motivation

I wrote this library when there weren't that many viable matrix libraries available. This library has stood the test of running in mission critical production for many years. It has been optimized for correctness and performance.
Most of the math routines are straight translation from Fortran libraries,. The arithmetic operators are implemented by expression template techniques.

Code structure

This is strictly a header only library. All the Matrix codes are in *include* directory. The *src* directory contains the test source file and make files

Build Instructions

Using plain make and make-files

Go to the root of the repository, where license file is, and execute *build_all.sh*. This will build test executables for Linux flavors.

Using cmake

Please see README file. Thanks to @justinjk007, you should be able to build this in Linux, Windows, Mac, and more

Example

These are some example code:

```
DDMatrix dmat (8, 3);
int      count = 0;

for (DDMatrix::size_type i = 0; i < 8; ++i)
    for (DDMatrix::size_type j = 0; j < 3; ++j)
        dmat (i, j) = ++count;

dmat.dump (std::cout);

const   DDMatrix::ColumnVector   cv = dmat.get_column (1);
const   DDMatrix::RowVector      rv = dmat.get_row (4);

std::cout << "Column 1:\n";
for (DDMatrix::ColumnVector::const_iterator citer = cv.begin ();
     citer != cv.end (); ++citer)
    std::cout << *citer << "\n";

std::cout << "\nRow 4:\n";
for (DDMatrix::RowVector::const_iterator citer = rv.begin ();
     citer != rv.end (); ++citer)
    std::cout << *citer << "   ";
std::cout << std::endl;

auto            dfut = dmat.determinant_async ();
const double    deter = dfut.get();

std::cout << "Determinant:  " << deter << std::endl;
std::cout << "Condition#:   " << dmat.condition () << std::endl;
```

```
std::cout << "Is singular?  " << dmat.is_singular () << std::endl;
```

For more examples see file *matrix_tester.cc*

Types

The exception that Matrix may throw

```
struct  MatrixException { };
struct  NotSquare : public MatrixException  { };
struct  Singular : public MatrixException  { };
struct  NotSolvable : public MatrixException  { };
```

Enum to represent I/O format

```
enum class io_format : unsigned char  {
   csv = 1,
   binary = 2
};
```

These are various types used throughout

```
using size_type =              unsigned int;
using value_type               T;
using reference =              value_type &;
using const_reference =        const value_type &;
using pointer =                value_type *;
using const_pointer =          const value_type *;
using ColumnVector =           VectorRange<value_type>;
using RowVector =              StepVectorRange<value_type>;
```

Methods

Default constructor that creates a matrix with zero rows and columns

```
inline Matrix() = default;
```

Constructor to create a matrix with row rows and col columns

row: Number of rows to be created
col: Number of columns to be created
def_value: The default value (of T) to be filled in.

```
inline Matrix (size_type row,
               size_type col,
               const_reference def_value = value_type ()) noexcept;
```

const and none-const versions of operator () to access a value at a given row and column in the matrix. It returns a [const] reference to the value.

r: Row number
c: column number
inline reference operator() (size_type r, size_type c) noexcept;
inline const_reference operator() (size_type r, size_type c) const noexcept;
inline reference at (size_type r, size_type c) noexcept;
inline const_reference at (size_type r, size_type c) const noexcept;

Transpose self and return a reference to self
inline Matrix &transpose () noexcept;
Assign a transposed copy of self to that and return a reference to that. Leave self unchanged
inline Matrix &transpose (Matrix &that) const noexcept;

Inverse(A) * A = A * Inverse(A) = I
Also:

$$\text{Inverse}(A) = \frac{1}{\text{Determinant}(A)} * \text{Adjoint}(A)$$

Invert self and return a reference to self
inline Matrix &invert(); // throw (NotSquare, Singular);
Assign an inverted copy of self to that and return a reference to that. Leave self unchanged
inline Matrix &inverse(Matrix &that) const; // throw (NotSquare, Singular);

Row Reduced Echelon Form:
A matrix that has undergone Gaussian elimination is said to be in row reduced echelon form. Such a matrix has the following characteristics:
1. All zero rows are at the bottom of the matrix
2. The leading entry of each nonzero row occurs to the right of the leading entry of the previous row.
3. The leading entry in any nonzero row is 1.
4. All entries in the column above and below a leading 1 are zero.

RREF is an important technique in solving simultaneous equations and finding the rank of a matrix.

Apply rref to self and return a reference to self

rank: a reference to contain the rank after return
inline Matrix &rref (size_type &rank) noexcept;
Assign an applied rref copy of self to that and return a reference to that. Leave self unchanged
inline Matrix &rref (Matrix &that, size_type &rank) const noexcept;

Return an identity matrix
inline Matrix &identity (); // throw (NotSquare);
static Matrix &identity (Matrix &that); // throw (NotSquare);

The column rank of a matrix is the maximal number of linearly independent columns of the matrix. Likewise, the row rank is the maximal number of linearly independent rows of the matrix. Since the column rank and the row rank are always equal, they are simply called the rank of the matrix;
for the proofs, see, Murase (1960), Andrea & Wong (1960), Williams & Cater (1968), Mackiw (1995).
inline size_type rank () const noexcept;

This is the best determinant definition I could find:
In 2-D, when you talk about the point (2, 4), you can think of the 2 and 4 as directions to get from the origin to the point (move 2 units in the x direction and 4 in the y direction). In a 3-D system, the same idea holds - (1, 3, 7) means start at the origin (0, 0, 0), go 1 unit in the x direction, 3 in the y direction, and 7 in the z direction.
The determinant of a 1x1 matrix is the signed length of the line from the origin to the point. It's positive if the point is in the positive x direction, negative if in the other direction. In 2-D, look at the matrix as two 2-dimensional points on the plane, and complete the parallelogram that includes those two points and the origin. The (signed) area of this parallelogram is the determinant. If you sweep clockwise from the first to the second, the determinant is negative, otherwise, positive. In 3-D, look at the matrix as 3 3-dimensional points in space. Complete the parallel-pipe that includes these points and the origin, and the determinant is the (signed) volume of the parallel-pipe. The same idea works in any number of dimensions. The determinant is just the (signed) volume of the n-dimensional parallel-pipe. Note that length, area, volume are the _volumes_ in 1, 2, and 3-dimensional spaces. A similar concept of volume exists for Euclidean space of any dimensionality.

[HM]: A matrix that has determinant equals zero is singular. It means it has at least 2 vectors that are linearly dependent, therefore they cannot enclose an area or a space.

For how determinant of N X N matrices are calculated see:
http://mathworld.wolfram.com/Determinant.html
For Laplace Expansion see: http://en.wikipedia.org/wiki/Cofactor_expansion

NOTE: This is a relatively _expensive_ calculation. Its complexity is O(n!), where n is the number of rows or columns.
inline value_type determinant () const; // throw (NotSquare);
Same determinant but an async version
inline std::future<value_type>
determinant_async () const;

Minor of a matrix is the same matrix with the specified row and column taken out.

NOTE: Linear algebraically speaking, a minor of a matrix is the determinant_ of some smaller square matrix, cut down from the original matrix.

mmatrix: The matrix to contain the minor of self after return
drow: Row to take out
dcol: Columns to take out
It return a reference to mmatrix

```
inline Matrix &
get_minor (Matrix &mmatrix, size_type drow, size_type dcol) const noexcept;
```

A Cofactor of a matrix is the determinant of a minor of the matrix. You can also say cofactor is the signed minor of the matrix.

row: Rows to take out for getting the minor
col: columns to take out for getting the minor
It returns the co-factor

```
inline value_type
cofactor (size_type row, size_type column) const; // throw (NotSquare);
```

The Adjoint of a matrix is formed by taking the transpose of the cofactors matrix of the original matrix.

amatrix: A matrix to contain the adjoint
It returns a reference to amatrix

```
inline Matrix &
adjoint (Matrix &amatrix) const; // throw (NotSquare);
```

Variance/Covariance matrix.
The columns of the matrix are assumed to be observations of some random variable. So the covariance matrix is a square matrix containing the covariances of the above random variables.

If it is unbiased the estimate is divided by n - 1, otherwise by n, n being the number of rows.

The reason for dividing by n - 1 is because we are dividing by degrees of freedom (i.e. number of independent observations). For example, if we have two observations, when calculating the mean we have two independent observations; however, when calculating the variance, we have only one independent observation, since the two observations are equally distant from the mean.
For a nXm matrix, you will get a mXm covariance matrix

is_unbiased: If unbiased, take one less row

It return the covariance matrix
inline Matrix
covariance (bool is_unbiased = true) const; // throw (NotSolvable);

The Pearson product-moment correlation coefficient:

$$\frac{Cov(x, y)}{Std(x) * Std(y)}$$

For a nXm matrix, you will get a mXm correlation matrix

It returns the correlation matrix
inline Matrix correlation () const; // throw (NotSolvable);

Solve the simultaneous equation Ax = rhs by Gaussian elimination.

It returns the x vector.
inline Matrix
solve_se (const Matrix &rhs) const; // throw(NotSolvable, Singular);

Frobenius Norm:
The Frobenius norm of a matrix is the square root of the sum of the squares of the values of the elements of the matrix.

It returns the norm
inline value_type norm () const noexcept;

The Max (Euclidian) norm of a matrix is the maximum absolute singular value of the matrix.

It returns the norm
inline value_type max_norm () const noexcept;

The Column norm of a matrix is the absolute sum of the column with the maximum value in the matrix.

It return the column norm
inline value_type col_norm () const noexcept;

The Row norm of a matrix is the absolute sum of the row with the maximum value in the matrix.

It returns the row norm
inline value_type row_norm () const noexcept;

In numerical analysis, the condition number associated with a problem is a measure of that problem's amenability to digital computation, that is, how numerically well-posed the problem is. A problem with a low condition number is said to be well-conditioned, while a problem with a high condition number is said to be ill-conditioned.

The condition number of a matrix:
For example, the condition number associated with the linear equation Ax = b gives a bound on how inaccurate the solution x will be after approximate solution. Note that this is before the effects of round-off error are taken into account; conditioning is a property of the matrix, not the algorithm or floating-point accuracy of the computer used to solve the corresponding system. In particular, one should think of the condition number as being (very roughly) the rate at which the solution, x, will change with respect to a change in b. Thus, if the condition number is large, even a small error in b may cause a large error in x. On the other hand, if the condition number is small then the error in x will not be much bigger than the error in b.
The condition number is defined more precisely to be the maximum ratio of the relative error in x divided by the relative error in b.

It returns the condition
inline value_type condition () const; // throw (NotSquare, Singular);

Sum of the values on the main diagonal or equivalently sum of the
matrix's eigenvalues.

It returns the trace
inline value_type trace () const; // throw (NotSquare);

Let A be an nXn matrix. The number l is an eigenvalue of A if there exists a non-zero vector v such that
Av = lv
In this case, vector v is called an eigenvector of A corresponding to l. For each eigenvalue l, the set of all vectors v satisfying Av = lv is called the eigenspace of A corresponding to l.

Putting it in more laymen's term:
Place a coin on its side and set it spinning. Take some butter and spread it over a slice of bread. Pick up an elastic band and stretch it. In each of these cases we do something that affects the shape or orientation of the object in question. The elastic band and butter have been deformed and the coin has been rotated. When we investigate transformations mathematically we find that there are directions that remain the same after the deformation has occurred. In the case of the stretching of the elastic band, if you had

drawn an arrow on the band before you stretched it, would it point in the same direction afterwards? The answer depends on how you drew the original arrow. The transformation preserves the direction in which you stretch the band, but some other arrows have their direction changed. The preserved direction is called an EIGENVECTOR of the transformation and the associated amount by which it has been stretched is an EIGENVALUE. Eigenvalues are multipliers. They are numbers that represent how much stretching has taken place or, in other words, how much something has been scaled up by. In the sentence 'I am 3.2 times taller than when I was born' the number 3.2 is acting as an eigenvalue. To make sense of an eigenvalue it must have an associated 'operation' (the transformation that has occurred) and an associated 'direction' (the eigenvector). It doesn't mean anything by itself to say that 3.2 is an eigenvalue. You need to know the operation 'enlarged' and the direction 'up' to say that I am now 3.2 times bigger in the up-direction! If you rotate a coin by 360 degrees you preserve all directions and so each direction is an eigenvector. Because no stretching has occurred, all of these eigenvectors have eigenvalue 1. Rotating the coin by 60 degrees destroys all directions and this transformation has no eigenvectors or eigenvalues at all. Careful spreading of butter on bread, by moving the knife in one direction only, is a transformation with an eigenvector in the direction of spreading. The associated eigenvalue depends on how far the butter is spread. The elastic band has an eigenvector in the left-right direction because that arrow still points in that direction after the transformation. The band has been stretched by approximately the same amount that I have grown since I was born - so the eigenvector has eigenvalue 3.2!
The idea of eigenvalues and eigenvectors can be extended to any operator H. (H might be 'rotate by 360 degrees' or 'stretch in direction of y-axis' or operators in Quantum theory or elsewhere). We write H(x) to mean 'the action of H on x'. (So x might be a particular vector that we are rotating or stretching, or it might be a quantum state or some other object). If we can find an object x and a number k so that the following equation is true:

    H(x) = k * x

Then we know that x has been preserved by H apart from a scalar multiplier k. It is k times bigger than what it was before H acted upon it. Therefore we call x an eigenvector of H with eigenvalue k.

This method finds all the eigenvalues and eigenvectors.
If matrix is symmetric:
    first tri-diagonalize, then diagonalize.
else:
    reduce to Hessenberg form, then reduce to real Schur form.

eigenvalue: A matrix to contain the eigen values
eigenvectors: A matrix to contain the eigen vectors
sort_values: If true, it sorts the values in ascending order

```
template<class MAT>
inline void
eigen_space (MAT &eigenvalues,
             MAT &eigenvectors,
             bool sort_values = false) const; // throw (NotSolvable);
```

The async version of eigen_space

```
template<class MAT>
```

```
inline std::future<void>
eigen_space_async (MAT &eigenvalues,
                   MAT &eigenvectors,
                   bool sort_values = false) const;
```

The n-th root of a diagonal matrix is another diagonal matrix with each element being the n-th root of the corresponding element in the
original matrix. A non-diagonal nXn matrix A could be written as A = Uλ(U^-1). Where U is the matrix of Eigen vectors and λ is the diagonal matrix of Eigen values. If λ is positive then:
A^n = U*(λ^n)*(U^-1)

result: A matrix to contain the root result
n: The n-th root
is_diag: If true, it indicates self is diagonal
It returns a reference to result

```
inline Matrix &power (Matrix &result,
                      value_type n,
                      bool is_diag) const; // throw (NotSolvable);
```

Same as above, but it raise self to the n-th power
It returns a reference to self

```
inline Matrix &power (value_type n, bool is_diag); // throw (NotSolvable);
```

In linear algebra, the Singular Value Decomposition (SVD) is an important factorization of a rectangular real or complex matrix, with several applications in signal processing and statistics. Applications which employ the SVD include computing the pseudoinverse, matrix approximation, and determining the rank, range and null space of a matrix.

Suppose M is an mXn matrix whose entries come from the field K, which is either the field of real numbers or the field of complex numbers. Then there exists a factorization of the form

M = U*Σ*~V

where U is an mXm unitary matrix over K, the matrix Σ is mXn with nonnegative numbers on the diagonal (as defined for a rectangular matrix) and zeros off the diagonal, and ~V denotes the conjugate transpose of V (transpose of V in case of real matrices), an nXn unitary matrix over K. Such a factorization is called a Singular Value Decomposition of M.

-- The matrix V thus contains a set of orthonormal "input" or "analyzing" basis vector directions for M
-- The matrix U contains a set of orthonormal "output" basis vector directions for M
-- The matrix Σ contains the singular values, which can be thought of as scalar "gain controls" by which each corresponding input is multiplied to give a corresponding output.

A common convention is to order the values Σi,i in non-increasing
fashion. In this case, the diagonal matrix Σ is uniquely determined

by M (though the matrices U and V are not).

U: The U matrix, see above
S: The S matrix, see above
V: The V matrix, see above
inline void svd (Matrix &U,
Matrix &S,
Matrix &V,
bool full_size_S = true) const; // throw (NotSolvable);

In linear algebra, the QR decomposition (also called the QR factorization) of a matrix is a decomposition of the matrix into an orthogonal and a triangular matrix. The QR decomposition could be used to solve the linear least squares problem. The QR decomposition is also the basis for a particular eigenvalue algorithm, the QR algorithm. QR decomposition of a real square matrix A is a decomposition of A as

A = QR

where Q is an orthogonal matrix (meaning that ~Q * Q = I) and R is an upper triangular matrix (also called right triangular matrix). Analogously, we can define the QL, RQ, and LQ decompositions of A (with L being a left triangular matrix in this case).

Q: The Q matrix, see above
R: The R matrix, see above
inline void qrd (Matrix &Q, Matrix &R) const noexcept;

In linear algebra, the LU decomposition is a matrix decomposition which writes a matrix as the product of a lower and upper triangular matrices. This decomposition is used in numerical analysis to solve systems of linear equations or calculate the determinant. LU decomposition of a square matrix A is a decomposition of A as

A = LU

where L and U are lower and upper triangular matrices (of the same size), respectively. This means that L has only zeros above the diagonal and U has only zeros below the diagonal. The LU decomposition is twice as efficient as the QR decomposition.

NOTE: The LU decomposition with pivoting always exists, even if the matrix is singular. The primary use of the LU decomposition is in the solution of square systems of simultaneous linear equations. This will fail if is_singular() returns false.

L: The L matrix, see above
U: The U matrix, see above
inline void lud (Matrix &L, Matrix &U) const; // throw (NotSquare);

Cholesky decomposition:
In mathematics, the Cholesky decomposition is named after André-Louis Cholesky, who found that a symmetric positive-definite matrix can be decomposed into a lower triangular matrix and the transpose of the lower triangular matrix. The lower triangular

matrix is the Cholesky triangle of the original, positive-definite matrix. Cholesky's result has since been extended to matrices with complex entries.
Any square matrix A with non-zero pivots can be written as the product of a lower triangular matrix L and an upper triangular matrix U; this is called the LU decomposition. However, if A is symmetric and positive definite, we can choose the factors such that U is the transpose of L, and this is called the Cholesky decomposition. Both the LU and the Cholesky decomposition are used to solve systems of linear equations. When it is applicable, the Cholesky decomposition is twice as efficient as the LU decomposition.

NOTE: if right == true it is a right Cholesky decomposition, meaning for a symmetric positive-definite matrix A, you get

A = ~R * R

else it is a left Cholesky decomposition, meaning you get

A = R * ~R

R: The R matrix, see above
right: See above

```
template<class MAT>
inline void chod (MAT &R, bool right = true) const; // throw (NotSolvable);
```

Boolean Methods

The followings return the obvious results

```
inline bool is_square () const noexcept;
inline bool is_singular () const noexcept;
inline bool is_diagonal () const noexcept;
```

A diagonal matrix with non-zero diagonal

```
inline bool is_scalar () const noexcept;
inline bool is_identity () const noexcept;
inline bool is_null () const noexcept;
```

For real matrices symmetric and Hermitian are equivalent

```
inline bool is_symmetric () const noexcept;
inline bool is_skew_symmetric () const noexcept;
inline bool is_upper_triangular () const noexcept;
inline bool is_lower_triangular () const noexcept;
```

A * ~A == ~A * A

```
inline bool is_normal () const noexcept;
```

An orthogonal matrix is a square matrix whose transpose is its inverse:

A * ~A == ~A * A == I
A complex orthogonal matrix is called unitary
inline bool is_orthogonal () const; // throw (Singular);

It is orthogonal and its determinant == 1
inline bool is_special_orthogonal () const; // throw (Singular);

Miscellaneous Methods

Clears the matrix and sets rows and columns to zero
void clear () noexcept;

Returns true, if self has zero rows and columns
inline bool empty () const noexcept;

They returns the number of rows and columns respectively
inline size_type rows () const noexcept;
inline size_type columns () const noexcept;

This writes self, in special csv format, to the stream
template<typename STRM>
bool write (STRM &stream, io_format iof = io_format::csv) const;

This reads self from the given file
*bool read (const char *file_name, io_format iof = io_format::csv);*

Return the given row or column in vector format
inline ColumnVector get_column (size_type c) noexcept;
inline ColumnVector get_column (size_type c) const noexcept;
inline RowVector get_row (size_type r) noexcept;
inline RowVector get_row (size_type r) const noexcept;

Row and column operations run a binary operator on the row/column and the given iterator.
template<class OPT, class ITER>
inline void column_operation (OPT opt, ITER col_data, size_type col);
template<class OPT, class ITER>
inline void row_operation (OPT opt, ITER row_data, size_type row);

Row and column scale run a binary operator on the row/column and the given expression.

```
template<class OPT, class EXPR>
inline void scale_column (OPT opt, const EXPR &e, size_type col);
template<class OPT, class EXPR>
inline void scale_row (OPT opt, const EXPR &e, size_type row);
```

Scale the matrix by the given operator and scalar

```
template<class OPT, class EXPR>
inline void scale (OPT opt, const EXPR &e) noexcept;
```

There are row and columns iterators. See matrix_tester.cc