FINAL REPORT

Extending Boost uBLAS Library

GENERIC PROGRAMMING, CPSC-622

KARTHICK MANIVANNAN AND RAGHAVENDER BOORGAPALLY

12/12/2008

USER GUIDE

OVERVIEW

RATIONALE

This extension of Boost uBLAS (u-Basic Linear Algebra Subprograms) library is directed towards those in need of a library for developing signal processing or data analysis applications that makes heavy use of matrix and vector manipulation. The primary design goals are:

- ***Provide MATLAB like interface to uBLAS.***
- ***Adding richer interface but staying close to the design style of Boost uBLAS.***
- ***Non intrusive extension of uBLAS while preserving efficiency and maximizing code reuse.***
- ***Aid faster porting of MATLAB code to C++.***

RESOURCES

This project for extending uBLAS draws inspiration from:

- uBLAS ***by Joerg Walter, Mathias Koch.***
- MATLAB***, a commercially available prototyping software by Mathworks.***

This document has similar style as that of Boost uBLAS documentation

FUNCTIONALITIES

Mathematical Operations

> ***"The usage of mathematical notation may ease the development of scientific algorithms. Hence, uBLAS carefully overloads selected C++ operators on matrix and vector classes"****(Joerg Walter, Mathias Koch)*

We decided to use operator overloading wherever possible without introducing ambiguities. The following notations will be used to summarize the operations that were added other than those that are currently available in uBLAS ***:***

Legend	
Notation	Type
A, B, C	Matrices or Matrix expressions *(Note: matrix expressions cannot be a L values)*
u, v, w	Vectors or Vector expressions *(Note: vector expressions cannot be a L values)*
I, j, k	Integer values
ui, vi, wi	Integer vectors (for indices)
t, t1, t2	Scalar values

Binary Operations	
Operation	Valid Expression
Scalar add	A + t, t + A u + t, t + u
Scalar subtraction	A - t, t - A u - t, t - u
Element wise multiply	elem_mult(A, B) elem_mult(u, v)
Element wise divide	elem_divide(A, B) elem_divide(u, v)
Element wise integer power	A ^ j u ^ j
Element wise real power	power(A, t) power(u, t)

Self Assign	
Operation	Valid Expression
Plus assign scalar	A += t u += t
Minus assign scalar	A -= t u -= t

Unary Operations (element wise)	
Operation	Valid Expression
Absolute value	abs(A) abs(u)
Exponential	exp(A) exp(u)
Natural log	log(A) log(u)
Log base 10	log10(A) log10(u)
Sine	sin(A) sin(u)
cosine	cos(A) cos(u)
Square root	sqrt(A) sqrt(u)

Unary Operations (returning scalar)	
Operation (on vectors)	Valid Expression
L1 Norm	t = norm_L1(u)
L2 Norm	t = norm_L2(u)
Infinity Norm	t = norm_inf(u)
Min	t = min(u)
Max	t = max(u)
Absolute min	t = abs_min(u)
Absolute max	t = abs_max(u)

Unary Operations (returning scalar)	
Operation (on vectors)	Valid Expression
Sum	t = sum(u)
Mean	t = mean(u)
Absolute mean	t = abs_mean(u)
RMS	t = RMS(u)

Unary Operations (returning vectors)	
Operation (Column wise)	Valid Expression
L1 Norm	u = norm_L1(A)
L2 Norm	u = norm_L2(A)
Infinity Norm	u = norm_inf(A)
Min	u = min(A)
Max	u = max(A)
Absolute min	u = abs_min(A)
Absolute max	u = abs_max(A)
Sum	u = sum(A)
Mean	u = mean(A)
Absolute mean	u = abs_mean(A)

Unary Operations (returning indices)	
Operation (on vectors)	Valid Expression
Infinity Norm index	i= norm_inf_ind(u)
Min index	i = min_ind (u)
Max index	i= max_ind (u)
Absolute min index	i = abs_min_ind (u)
Absolute max index	i = abs_max_ind (u)

Unary Operations (returning vector of indices)	
Operation (on colums)	Valid Expression
Infinity Norm index	ui= norm_inf_ind(A)
Min index	ui = min_ind (A)
Max index	ui= max_ind (A)
Absolute min index	ui = abs_min_ind (A)
Absolute max index	ui = abs_max_ind (A)

Comparison Operations (returning boolean expression)	
Operation (element wise)	Valid Expression
Greater than	A > B, A > t, t > A u > v, u > t, t > u
Greater than or equal	A >= B, A >= t, t >= A u >= v, u >= t, t >= u
Less than	A < B, A < t, t < A u < v, u < t, t < u
Less than or equal	A <= B, A <= t, t <= A u <= v, u <= t, t <= u
Equal to	A = B, A = t, t = A u = v, u = t, t = u
Not equal to	A == B, A == t, t == A u == v, u == t, t == u

Logical Operations (on boolean expressions)	
Operation	Valid Expression
AND	A && B, A && t, t && A u && v, u &&t, t && u
OR	A || B, A || t, t || A u || v, u || t, t || u
NOT	!A !u

Indexing

Boost uBLAS already provides a rich set of indexing/subscripting operations that allow both element access and access to sub matrices and vectors, however MATLAB provides a even more intuitive interface that allow directly accessing a subset of elements from a vector or matrix using subscript () operators. For example to access a range of elements with indices n to m of a vector V, uBLAS requires the following convention project(V, range (n, (m - n + 1)). While, MATLAB will allow the following V(n:m).The later is more compact and intuitive. MATLAB also provides some ways to access the last element V(end), all elements V(:), similarly for matrices last row can be accessed by M(end,:). Similar interface can be created by overloading the operator () of uBLAS vectors and matrices. The following is the (non-exhaustive) list of new subscripted access to vectors and matrices.

Example Subscript Operations on Vectors		
Operation	Valid Expressions	MATLAB Equivalent
Vector Range	u(I,j) u(R(I,j)) u(ALL) u(reverse()) u(I, END)	u(I:j) u(I:j) u(:) u(end:-1:1) u(i:end)

Example Subscript Operations on Vectors		
Operation	Valid Expressions	MATLAB Equivalent
Vector Slice	u(I,j, k) u(S(I,j,k)) u(I, j, END) u(END, I, j)	u(I:j:k) u(I:j:k) u(i:j:end) u(end:i:j)
Vector index-vector	u(vi) //Elements at indices in vi	u(vi)

Example Subscript Operations on Matrices		
Operation	Valid Expressions	MATLAB Equivalent

Matrix range	A(R(I,j), R(k,l))
	A(I, ALL) //(ith column)
	A(ALL, i) //(ith row)
	A(END, :)//(last row)
		A(i:j, k:l)
		A(I, :)
		A(:, i)
		A(end, :)

Matrix slice	A(S(I,j,k), S(l,m,n)) A(S(I,j,k), l) A(l, S(I,j,k),)	N/A A([i:j:k], l) A(l, [i:j:k])
Matrix index-vector	A(vi, ui) A(vi, j)	N/A A(vi, j)

Not all the combinations for subscript access are listed above but the following summarizes the possibilities for vectors u(Range/Slice/Index-vector/index/end/all/reverse) and for matrices A(Range/Slice/Index-vector/index/end/all/reverse, Range/Slice/Index-vector/index/end/all/reverse).

Adapters

The tiling operations described above require creation of new matrices by repeatedly copying data. However, if the data will not be modified, this is an unnecessary overhead. Such scenarios do arise in signal processing domain. It will be beneficial to use an adapter that makes a matrix or a vector look like a larger matrix created by tiling. There are scenarios where a vector may be required to look like a matrix or a mxn matrix may be required to look like a lxk matrix (mxn ==lxk). Reshaping adapters will be useful in these scenarios without having to copy or create new vectors or matrices. The following adapters are provided.

Adapters		
Adapter	Valid Expressions	MATLAB Equivalent
vector_tile(u, m, n)	May be used as a matrix expression	N/A
vector_reshape(u, m, n)	May be used as a matrix expression and L value	N/A
matrix_tile(A, m, n)	May be used as a matrix expression	N/A
matrix_reshape(A, m, n)	May be used as a matrix expression and L value	N/A

DESIGN DECISIONS

Efficiency through Expression Templates

Boost uBLAS is a very good example of how expression templates and inlining can help improve efficiency by reducing temporaries and to some extent alleviating the cost of abstraction. Staying close to this idea, all mathematical, logical and comparison operations make use of expression templates. Indexing operations give rise to vector or matrix expressions. The tiling and reshape adapters can also be used in expressions. Thus, all efforts were made to extend the library without compromising efficiency.

Remaining Nonintrusive Vs Remaining Compatible

One of the major and challenging design decisions was to be nonintrusive. All efforts were made to provide this extended set of functionalities without having to modify the existing uBLAS library.This was indeed achieved. The additional functionality can be achieved simply by including the relevant header files. All new operators were implemented as global functions. Adapters or by definition nonintrusive. Adapter classes were derived from uBLAS vector and matrix expression classes and hence could be used seamlessly with existing uBLAS classes. Implementing the indexing extentions was particularly challenging since the function call operator () had to be overloaded retroactively. This was achieved by specializing existing uBLAS vector and matrix expression classes and injecting the new indexing operators.

TEST STRATEGY

Unit test

The various extensions (operations, adaptors, indexing, functions) were grouped and several test files were created to test the implementation. Due to the complex nature of the extension and the various possible scenarios that may arise, it is not possible to exhaustively test the library extensions. This is left as a future task for want of time. A make file is also provided to run the tests. Here is the list of test files that were used to test the implementation:

Test_Matrix_Vector_Find.cpp

Test_Vector_Unary_Returning_Vector.cpp

Test_Vector_Unary_Returning_Scalar.cpp

Test_Vector_Binary_Operations.cpp

Test_Matrix_Unary_Operations.cpp

Test_Matrix_Binary_Operations.cpp

Test_Vector_Tile.cpp

Test_Vector_Reshape.cpp

Test_Matrix_Tile.cpp

Test_Matrix_Reshape.cpp

Test_End_Tag.cpp

Test_Easy_Range_Slice.cpp

The make file has a rule that generates the object files for all the above files except Test_Vector_Unary_Returning_Scalar.cpp. This file tests the unary operations on vectors that return a scalar. The operations are tested to work well on VC++ platform. There is a bug that prevents it from compiling on UNIX. This is the known issue encountered during testing.

If you wish to generate an object file for the above files individually, run the make utility with the target as filename with extension changed to ".out".

TUTORIAL

VECTOR OPERATIONS

This extension of Boost uBLAS (u-Basic Linear Algebra Subprograms) library facilitates the usage of a wide variety of operations on vectors. Suppose we have two vectors. We wish to increment the corresponding components of the first vector whose sine or cosine values are greater than those of the second vector. The following example illustrates how to accomplish this.

```
#include "functional_ext.hpp"
#include "vector expression ext.hpp"
#include <iostream>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>

int main(){

  using namespace boost::numeric::ublas;
  vector<double> v1 (10), v2(10);
  vector<bool> bv1 (10), bv2(10);
  for (unsigned i = 0; i < v1.size (); ++ i){
       v1 (i) = (i+1);
       v2 (i) = (i*i);
  }
  bv1 = (cos(v1) > cos(v2));
  bv2 = (sin(v1) > sin(v2));
  std::cout << v1 << std::endl;
  std::cout << v2 << std::endl;
  std::cout <<cos(v1) << std::endl;
  std::cout <<cos(v2) << std::endl;
  std::cout << bv1 <<std::endl;
  std::cout << sin(v1) << std::endl;
  std::cout << sin(v2) << std::endl;
  std::cout << bv2 <<std::endl;
  bv1 = (bv1 || bv2);
  std::cout << bv1 << std::endl;
  for (unsigned i = 0; i < v1.size(); ++ i)
      if(bv1(i)) v1(i) = v1(i) + 4;
  std::cout << v1 << std::endl;
}
```

In the above example we declare two vectors v1 and v2. We apply unary operation cosine which returns a vector, to both the vectors and use the unary comparison operator ">" which returns a vector to determine which components have greater cosine values. The result is stored in the Boolean vector bv1. We repeat the same for sine and store the result in bv2. Then the binary operator logical OR is performed on the Boolean vectors bv1 and bv2. The result gives the components of v1 whose cosine or sine values are greater than those of v2. Then we increment the corresponding components by 4.

We can perform numerous similar operations on a vector by using a combination of unary and binary operations that are added to UBLAS. For a detailed list of operations that can be performed with the vector, please see the reference manual.

MATRIX OPERATIONS

This extension of Boost uBLAS (u-Basic Linear Algebra Subprograms) library facilitates the usage of a wide variety of operations on matrices. Suppose we have two matrices. We wish to increment the corresponding components of the first matrix whose sine or cosine values are greater than those of the second matrix. The following example illustrates how to accomplish this.

```
#include "functional_ext.hpp"
#include "matrix expression ext.hpp"
#include <iostream>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>

int main(){

  using namespace boost::numeric::ublas;
  matrix<double> m1 (3,3), m2(3,3);
  matrix<bool> bm1 (3,3), bm2(3,3);
  for (unsigned i = 0; i < m1.size1 (); ++ i)
      for(unsigned j = 0; j < m1.size2 (); ++ j){
       m1 (i,j) = (i+j)+3;
       m2 (i,j) = ((i+1)*(j+1));
  }
  bm1 = (cos(m1) > cos(m2));
  bm2 = (sin(m1) > sin(m2));
  std::cout << m1 << std::endl;
  std::cout << m2 << std::endl;
  std::cout <<cos(m1) << std::endl;
  std::cout <<cos(m2) << std::endl;
  std::cout << bm1 <<std::endl;
  std::cout << sin(m1) << std::endl;
  std::cout << sin(m2) << std::endl;
  std::cout << bm2 <<std::endl;
  bm1 = (bm1 || bm2);
  std::cout << bm1 << std::endl;
  for (unsigned i = 0; i < m1.size1 (); ++ i)
    for (unsigned j =0; j < m1.size2 (); ++j)
      if(bm1(i,j)) m1(i,j) = m1(i,j) + 4;
  std::cout << m1 << std::endl;
}
```

In the above example we declare two matrices m1 and m2. We apply unary operation cosine which returns a matrix, to both the matrices and use the unary comparison operator ">" which returns a matrix to determine which components have greater cosine values. The result is stored in the Boolean matrix bm1. We repeat the same for sine and store the result in bm2. Then the binary operator logical OR is performed on the Boolean matrices bm1 and bm2. The result gives the components of m1 whose cosine or sine values are greater than those of m2. Then we increment the corresponding components of m1 by 4.

We can perform numerous similar operations on a matrix by using a combination of unary and binary operations that are added to UBLAS. For a detailed list of operations that can be performed with the vector, please see the reference manual.

TILING ADAPTORS

The tiling adaptors can be used to construct a larger matrix which has similar sub matrices. The following example illustrates the usage of tiling adaptors. Suppose we have a 2*5 matrix. We wish to construct a 10*10 matrix out of it. We can accomplish this by tiling the matrix 5*2 times.

```
#include "functional_ext.hpp"
#include "matrix expression ext.hpp"
#include <iostream>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
#include "tile_matrix.hpp"

int main(){

  using namespace boost::numeric::ublas;
  typedef matrix<double,column_major> matrix;
  matrix m1 (2,5), m2(10,10);
  matrix_tile < matrix > mta(m1,5,2);

  for (unsigned i = 0; i < m1.size1 (); ++ i)
      for(unsigned j = 0; j < m1.size2 (); ++ j)
       m1 (i,j) = (i+j)+3;

     typedef matrix_tile < matrix > :: iterator1 i11_t;
   typedef matrix_tile < matrix > :: iterator2 i22_t;

   for (i11 t i1 = mta.begin1(); i1 != mta.end1(); ++i1) {
         for (i22_t i2 = i1.begin(); i2 != i1.end()(); ++i2)
                std::cout << "(" << i2.index1() << "," << i2.index2()
                          << ":" << *i2 << ")  ";
    std::cout << std::endl;
  }

   m2.assign(prod <matrix> (mta ,matmat(m1,5,2)));

   std::cout << m2 << std::endl;
}
```

In the above example we declare a 2 *5 matrix m1. We construct an adaptor by tiling this matrix 5*2 times to get a 10*10 matrix. Then we take the product of the resultant 10*10 matrix with itself and assign it to another 10*10 matrix.

The tiling adaptors facilitate the construction of interesting matrices. The reference manual contains a detailed account of matrix and vector tiling adaptors.

RESHAPING ADAPTORS

The reshaping adaptors can be used to construct a different matrix with the same elements as a matrix or vector. The following example illustrates the usage of reshaping adaptors. Suppose we have a 2*5 matrix. We wish to construct a 5*2 matrix out of it. We can accomplish this by reshaping the original matrix.

```
#include "functional ext.hpp"
#include "matrix_expression_ext.hpp"
#include <iostream>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
#include "matrix reshape.hpp"

int main(){

  using namespace boost::numeric::ublas;
  typedef matrix<double,column_major> matrix;
  matrix m1 (2,5), m2(2,2);
  matrix_reshape < matrix > mta(m1,5,2);

  for (unsigned i = 0; i < m1.size1 (); ++ i)
      for(unsigned j = 0; j < m1.size2 (); ++ j)
       m1 (i,j) = (i+j)+3;
```

```
  std::cout << m1 << std::endl;

  typedef matrix reshape < matrix > :: iterator1 i11 t;
  typedef matrix_reshape < matrix > :: iterator2 i22_t;

  for (i11_t i1 = mta.begin1(); i1 != mta.end1(); ++i1) {
         for (i22 t i2 = i1.begin(); i2 != i1.end()(); ++i2)
                std::cout << "(" << i2.index1() << "," << i2.index2()
                          << ":" << *i2 << ")   ";
    std::cout << std::endl;
  }

  m2.assign(prod <matrix> (m1,mta));

  std::cout << m2 << std::endl;
}
```

In the above example we declare a 2 *5 matrix m1. We construct an adaptor by that reshapes this matrix into a 5*2 matrix. Then we take the product of the reshaped matrix with original matrix and assign it to another 2*2 matrix.

The reshaping adaptors facilitate the construction of interesting matrices. The reference manual contains a detailed account of matrix and vector reshaping adaptors.

INDEXING OPERATIONS

A rich set of indexing operations are provided. These can be made to be automatically available by including the header "expression_types_ext.hpp". This makes available a set of Matlab like tag sthat allow accessing elements relative to last element (END). Easy slice and range access to vectors and matrices are also provided. A detailed description can be found in the reference manual section. The following code snippet shows how easily tags can be used to access vector and matrix elements in an intuitive fashion. This example shows how a range of elements can be accessed using the RG() macro and a slice of elements be accessed using SL() macro. It can also be seen that the slice and ranges may be used with either index values or tags(END, ALL, REV_ALL). Matrices may be accessed either as sub-blocks or as columns or rows.

```
        #include "expression types ext.hpp"                    //  Includes indexing extensions
        #include <boost\numeric\ublas\vector.hpp>
        #include <boost\numeric\ublas\matrix.hpp>
        #include <boost/numeric/ublas/io.hpp>
        int main(){
                using namespace boost::numeric::ublas;
                vector<int> examp(10, 1);
                for(std::size_t ii = 0; ii < 10; ii++){
                        examp[ii] = ii;
                }
                std::cout << "Same as all elements: " << examp(RG(0, END)) << std::endl;
                std::cout << "Elements at odd indices: " << examp(SL(1, 2, END)) << std::endl;
                std::cout << "elements at even indices: " << examp(SL(0, 2, END)) << std::endl;

                matrix<int> matExamp(10, 10, 10);
                for(std::size_t ii = 0; ii < 10; ii++)
                        for(std::size t jj = 0; jj < 10; jj++)
                                matExamp(ii, jj) = ii*10 + jj;



                std::cout << "Odd index elments from third row:" <<
                          matExamp(2, SL(1, 2, END))<< std::endl;
```

```
        std::cout << "Odd indexed elements for enitre matrix in reverse:" <<
                  matExamp(SL(END, -2, 0), SL(END, -2, 0))<< std::endl;
        return 0;
}
```

VECTOR AND MATRIX FIND FUNCTION

A generalized form of find() function for vector and matrices is available and may be used by including the header "ublas_find.hpp". The find function may be used to get the indices of all elements in a vector or a matrix that satisfy a certain Boolean condition. Any complex Boolean expression can be composed easily using the overloaded comparison and logical operations that are made available by the extended set of operators defined in previous sections. The find function may optionally be passed an additional vector or matrix (search) expression. In this case the search expression is indexed using the indices where the Boolean expression returns a true. This will be helpful if we need find() to return all elements in an expression satisfying a certain condition. The following example shows how the find function may be used. The reference manual has more detailed description.

```
#include "expression types ext.hpp"                    //  Includes indexing extensions
#include "matrix_expression_ext.hpp"
#include "vector expression ext.hpp"
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include "ublas_find.hpp"
#include <boost/numeric/ublas/io.hpp>
int main(){
        using namespace boost::numeric::ublas;
        // Initialize sample vectors
        vector<int> examp(10, 1), examp2(10, 1);
        for(std::size_t ii = 0; ii < 10; ii++){
                examp[ii] = ii;
                examp2[ii] = 10 - ii;
        }

        // Initialize sample matrix
        matrix<int> matExamp(10, 10, 10);
        for(std::size_t ii = 0; ii < 10; ii++)
                for(std::size t jj = 0; jj < 10; jj++)
                        matExamp(ii, jj) = ii*10 + jj;

        // Find elements of matrix matExamp satisfying a condition
        std::cout << find(((matExamp > 1) && (matExamp < 10)), matExamp) << std::endl;
        // Find elements of vector expression  (examp + 1) satisfying a boolean conditionm
        std::cout << find(((examp < 1) && (examp2 >= 5) ), examp + 1) << std::endl;
        return 0;

}
```

REFERENCE MANUAL

VECTOR OPERATIONS

UNARY OPERATIONS

Description

The template class vector_unary<E, F> describes a unary vector operation.

Definition

Defined in the header vector_expression.hpp

Template parameters

Parameter	Description
E	The type of the expression
F	The type of the operation

Model of

Vector Expression

Type requirements

None, except for those imposed by the requirements of Vector Expression

Public base classes

vector_expression <vector_unary < E, F> >

Members

Member	Description
vector_unary (const expression_type &e)	Constructs a description of the expression.
size_type size () const	Returns the size of the expression.
const_reference operator () (size_type i) const	Returns the value of the `i`-th element
const_iterator begin () const	Returns a `const_iterator` pointing to the beginning of the expression.

const_iterator end () const	Returns a `const_iterator` pointing to the end of the expression.
const_reverse_iterator rbegin () const	Returns a `const_reverse_iterator` pointing to the beginning of the reversed expression.
const_reverse_iterator rend () const	Returns a `const_reverse_iterator` pointing to the end of the reversed expression.

Prototypes of unary operations returning scalar

```
// Unary operations on vectors that return scalars
template <class E, class F>
struct vector_scalar_unary_traits {
        typedef vector_scalar_unary< E, F> expression_type;
        typedef expression_type result_type;
};
// Log Minimum
template<class E>
BOOST_UBLAS_INLINE
typename vector_scalar_unary_traits<E,
general_vector_scalar_unary<E,
scalar_log_minimum <typename E::value_type> > >::result_type
log_min (const vector_expression<E> &e);
// Log Maximum
template<class E>
BOOST_UBLAS_INLINE
typename vector_scalar_unary_traits<E,
general_vector_scalar_unary< E,
scalar_log_maximum <typename E::value_type> > >::result_type
log max (const vector expression<E> &e);
// Minimum
template<class E>
BOOST_UBLAS_INLINE
typename vector_scalar_unary_traits<E,
general vector scalar unary< E,
scalar_minimum<typename E::value_type> > >::result_type
min (const vector expression<E> &e);
// Maximum
template<class E>
BOOST_UBLAS_INLINE
typename vector scalar unary traits<E,
general_vector_scalar_unary< E,
scalar maximum<typename E::value type> > >::result type
max (const vector_expression<E> &e);
// Sum
template<class E>
BOOST UBLAS INLINE
typename vector_scalar_unary_traits<E,
general vector scalar unary< E,
sum_vector <typename E::value_type> > >::result_type
sum(const vector_expression<E> &e);
// Absolute minimum
template<class E>
BOOST UBLAS INLINE
typename vector_scalar_unary_traits<E,
general_vector_scalar_unary< E,
abs_min_vector<typename E::value_type> > >::result_type
abs min(const vector expression<E> &e);
// Absolute maximum
template<class E>
BOOST_UBLAS_INLINE
typename vector_scalar_unary_traits<E,
general_vector_scalar_unary< E,
abs max vector<typename E::value type> > >::result type
abs_max(const vector_expression<E> &e);
```

```
        // Mean
        template<class E>
        BOOST_UBLAS_INLINE
        typename vector scalar unary traits<E,
        general_vector_scalar_unary< E,
        mean_vector<typename E::value_type> > >::result_type
        mean(const vector_expression<E> &e);
        // Absolute minimum
        template<class E>
        BOOST UBLAS INLINE
        typename vector_scalar_unary_traits<E,
        general_vector_scalar_unary< E,
        abs_mean_vector<typename E::value_type> > >::result_type
        abs_mean(const vector_expression<E> &e);
        //Root Mean Square (RMS)
        template<class E>
        BOOST_UBLAS_INLINE
        typename vector_scalar_unary_traits<E,
        general_vector_scalar_unary< E,
        rms_vector<typename E::value_type> > >::result_type
```

Prototypes of unary operations returning vector

```
// (abs v) [i] = abs (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename vector unary traits<E, scalar abs<typename
E::value_type> >::result_type
abs (const vector expression<E> &e)
// (exp v) [i] = exp (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename vector unary traits<E, scalar exp<typename
E::value_type> >::result_type
exp (const vector expression<E> &e)
// (log v) [i] = log (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename vector unary traits<E, scalar log<typename
E::value_type> >::result_type
log (const vector expression<E> &e)
// (log10 v) [i] = log10 (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename vector_unary_traits<E, scalar_log10<typename
E::value_type> >::result_type
log10 (const vector_expression<E> &e)
// (sqrt v) [i] = sqrt (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename vector_unary_traits<E, scalar_sqrt<typename
E::value type> >::result type
sqrt  (const vector_expression<E> &e)
// (sin v) [i] = sin (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename vector_unary_traits<E, scalar_sin<typename
E::value type> >::result type
sin  (const vector_expression<E> &e)
// (cos v) [i] = cos (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename vector_unary_traits<E, scalar_cos<typename
E::value type> >::result type
cos  (const vector_expression<E> &e)
rms(const vector_expression<E> &e);
```

Examples: Unary operations

```
#include "vector_expression_ext.hpp"
#include <boost/numeric/ublas/vector.hpp>
int main(){
  using namespace boost::numeric::ublas;
  vector<double>  v(10);
  for (unsigned i = 0; i < v.size (); ++ i)
          v (i) = (i+1);

  std::cout << abs(v) << std::endl;
  std::cout << exp(v) << std::endl;
  std::cout << log(v) << std::endl;
  std::cout << log10(v) << std::endl;
  std::cout << sqrt(v) << std::endl;
  std::cout << sin(v)  << std::endl;
  std::cout << cos(v)  << std::endl;
  std::cout << log_min (v) << std::endl;
  std::cout << log max (v) << std::endl;
  std::cout << min(v) << std::endl;
  std::cout << max(v) << std::endl;
  std::cout << sum(v) << std::endl;
```

```
    std::cout << abs_min(v) << std::endl;
    std::cout << abs_max(v) << std::endl;
    std::cout << mean(v) << std::endl;
    std::cout << abs_mean(v) << std::endl;
    std::cout << rms(v) << std::endl;
}
```

BINARY OPERATIONS

Description

The template class vector_binary<E1, E2, F> describes a binary vector operation.

Definition

Defined in the header vector_expression.hpp

Template parameters

Parameter	Description
E1	The type of first vector expression
E2	The type of second vector expression
F	The type of the operation

Model of

Vector Expression

Type requirements

None, except for those imposed by the requirements of Vector Expression

Public base classes

vector_expression <vector_binary < E1, E2, F> >

Members

Member	Description
vector_binary (const expression1_type &e1, const_expression2_type &e2)	Constructs a description of the expression.
size_type size () const	Returns the size of the expression.
const_reference operator () (size_type i) const	Returns the value of the `i`-th element
const_iterator begin () const	Returns a `const_iterator` pointing to the beginning of the expression.
const_iterator end () const	Returns a `const_iterator` pointing to the end of the expression.

const_reverse_iterator rbegin () const	Returns a `const_reverse_iterator` pointing to the beginning of the reversed expression.
const_reverse_iterator rend () const	Returns a `const_reverse_iterator` pointing to the end of the reversed expression.

Prototypes of binary operations returning vector

```
template<class E1, class E2, class F>
struct vector_binary_traits {
typedef vector_binary<typename E1::const_closure_type,
typename E2::const_closure_type, F>  expression_type;
typedef expression type result type;
}
// (t + v) [i] = t + v [i]
template<class T1, class E2>
BOOST_UBLAS_INLINE
typename vector_binary_scalar1_traits<const T1, E2, scalar_plus<T1,
typename E2::value type> >::result type
operator + (const T1 &e1, const vector_expression<E2> &e2);
// (t - v) [i] = t - v [i]
template<class T1, class E2>
BOOST_UBLAS_INLINE
typename vector binary scalar1 traits<const T1, E2, scalar minus<T1,
typename E2::value_type> >::result_type
operator - (const T1 &e1, const vector expression<E2> &e2)
// (v + t) [i] = v [i] + t
template<class E1, class T2>
BOOST_UBLAS_INLINE
typename vector binary scalar2 traits<E1, const T2, scalar plus<typename
E1::value_type, T2> >::result_type
operator + (const vector expression<E1> &e1, const T2 &e2)
// (v - t) [i] = v [i] - t
template<class E1, class T2>
BOOST_UBLAS_INLINE
typename vector binary scalar2 traits<E1, const T2, scalar minus<typename
E1::value_type, T2> >::result_type
operator - (const vector expression<E1> &e1, const T2 &e2)
// (v ^ t) [i] = v [i] ^ t
template<class E1, class T2>
BOOST_UBLAS_INLINE
typename vector binary scalar2 traits<E1, const T2, scalar power<typename
E1::value_type, T2> >::result_type
operator ^ (const vector_expression<E1> &e1, const T2 &e2)
```

Prototypes of Binary comparison operations returning Boolean vector

```
// (v > t) [i] = v [i] > t
template<class E1>
BOOST_UBLAS_INLINE
typename vector_binary_scalar2_traits<E1, typename E1::value_type,
scalar greater<typename E1::value type, typename E1::value type>
>::result_type
operator > (const vector expression<E1> &e1, const typename E1::value type& e2)
// (t > v) [i] = t > v [i]
template<class E2>
BOOST_UBLAS_INLINE
typename vector binary scalar1 traits<typename E2::value type, E2,
scalar_greater<typename E2::value_type, typename E2::value_type>
>::result type
operator > (const typename E2::value_type& e1, const vector_expression<E2> &e2)
// (v1 > v2) [i] = v1[i] > v2[i]
template <class E1, class E2>
BOOST UBLAS INLINE
typename vector_binary_traits <E1, E2, scalar_greater <typename
```

```
E1::value_type, typename E2::value_type> >::result_type
operator > (const vector expression<E1> &e1,
const vector_expression<E2> &e2)
// (v >= t) [i] = v [i] >= t
template<class E1>
BOOST_UBLAS_INLINE
typename vector_binary_scalar2_traits<E1, typename E1::value_type,
scalar greater equal<typename E1::value type, typename E1::value type>
>::result_type
operator >= (const vector expression<E1> &e1, const typename E1::value type& e2)
// (t >= v) [i] = t >= v [i]
template<class E2>
BOOST_UBLAS_INLINE
typename vector_binary_scalar1_traits<typename E2::value_type, E2,
scalar_greater_equal<typename E2::value_type, typename E2::value_type>
>::result type
operator >= (const typename E2::value_type& e1, const vector_expression<E2> &e2)
// (v1 >= v2) [i] = v1[i] >= v2[i]
template <class E1, class E2>
BOOST_UBLAS_INLINE
typename vector_binary_traits <E1, E2, scalar_greater_equal <typename
E1::value type , typename E2::value type > >::result type
operator >= (const vector_expression<E1> &e1,
const vector_expression<E2> &e2)
// (v < t) [i] = v [i] < t
template<class E1>
BOOST UBLAS INLINE
typename vector_binary_scalar2_traits<E1, typename E1::value_type,
scalar lesser<typename E1::value type, typename E1::value type>
>::result_type
operator < (const vector_expression<E1> &e1, const typename E1::value_type& e2)
// (t < v) [i] = t < v [i]
template<class E2>
BOOST_UBLAS_INLINE
typename vector binary scalar1 traits<typename E2::value type, E2,
scalar_lesser<typename E2::value_type, typename E2::value_type>
>::result_type
operator < (const typename E2::value_type& e1, const vector_expression<E2> &e2)
// (v1 < v2) [i] = v1[i] < v2[i]
template <class E1, class E2>
BOOST UBLAS INLINE
typename vector_binary_traits <E1, E2, scalar_lesser <typename
E1::value_type, typename E2::value_type> >::result_type
operator < (const vector_expression<E1> &e1,
const vector expression<E2> &e2)
// (v <= t) [i] = v [i] <= t
template<class E1>
BOOST_UBLAS_INLINE
typename vector_binary_scalar2_traits<E1, typename E1::value_type,
scalar lesser equal<typename E1::value type, typename E1::value type>
>::result_type
operator <= (const vector expression<E1> &e1, const typename E1::value type& e2)
// (t <= v) [i] = t <= v [i]
template<class E2>
BOOST_UBLAS_INLINE
typename vector binary scalar1 traits<typename E2::value type, E2,
scalar_lesser_equal<typename E2::value_type, typename E2::value_type>
>::result type
operator <= (const typename E2::value_type& e1, const vector_expression<E2> &e2)
// (v1 <= v2) [i] = v1[i] <= v2[i]
template <class E1, class E2>
BOOST_UBLAS_INLINE
typename vector_binary_traits <E1, E2, scalar_lesser_equal <typename
E1::value type, typename E2::value type> >::result type
operator <= (const vector_expression<E1> &e1,
const vector_expression<E2> &e2)
// (v != t) [i] = v [i] != t
template<class E1>
BOOST_UBLAS_INLINE
typename vector binary scalar2 traits<E1, typename E1::value type,
scalar_inequality<typename E1::value_type, typename E1::value_type>
```

```
>::result_type
operator != (const vector expression<E1> &e1, const typename E1::value type& e2)
// (t != v) [i] = t != v [i]
template<class E2>
BOOST_UBLAS_INLINE
typename vector_binary_scalar1_traits<typename E2::value_type, E2,
scalar_inequality<typename E2::value_type, typename E2::value_type>
>::result type
operator != (const typename E2::value_type& e1, const vector_expression<E2> &e2)
// (v1 != v2) [i] = v1[i] != v2[i]
template <class E1, class E2>
BOOST_UBLAS_INLINE
typename vector_binary_traits <E1, E2, scalar_inequality <typename
E1::value_type, typename E2::value_type> >::result_type
operator != (const vector_expression<E1> &e1,
const vector_expression<E2> &e2)
```

Prototypes of binary logical operations returning Boolean vector

```
// (v && t) [i] = v [i] && t
template<class E1>
BOOST_UBLAS_INLINE
typename vector_binary_scalar2_traits<E1, const bool,
logical_and<typename E1::value_type, bool> >::result_type
operator && (const vector_expression<E1> &e1, const bool &e2)
// (t && v) [i] = t && v [i]
template< class E2>
BOOST_UBLAS_INLINE
typename vector_binary_scalar1_traits<const bool, E2, logical_and<bool,
typename E2::value_type> >::result_type
operator && (const bool &e1, const vector_expression<E2> &e2)
// (v1 && v2) [i] = v1[i] || v2[i]
template <class E1, class E2>
BOOST UBLAS INLINE
typename vector_binary_traits <E1, E2, logical_and <bool, bool>
>::result_type
operator && (const vector_expression<E1> &e1,
const vector expression<E2> &e2)
// (v || t) [i] = v [i] || t
template<class E1>
BOOST_UBLAS_INLINE
typename vector_binary_scalar2_traits<E1, const bool, logical_or<
typename E1::value_type, bool> >::result_type
operator || (const vector expression<E1> &e1, const bool &e2)
// (t || v) [i] = t || v [i]
template<class E2>
BOOST_UBLAS_INLINE
typename vector_binary_scalar1_traits<const bool, E2, logical_or<bool,
typename E2::value_type> >::result_type
operator || (const bool &e1, const vector expression<E2> &e2)
// (v1 || v2) [i] = v1[i] || v2[i]
template <class E1, class E2>
BOOST_UBLAS_INLINE
typename vector_binary_traits <E1, E2, logical_or <bool, bool>
>::result type
operator || (const vector expression<E1> &e1,
const vector expression<E2> &e2)
// (!v)[i] = ! (v[i])
template <class E>
BOOST_UBLAS_INLINE
typename vector unary traits <E, logical not < typename E::value type> >
:: result_type
operator ! (const vector_expression <E> &e)
```

Examples: Binary operations on vectors

```
#include "vector_expression_ext.hpp"
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>
int main(){
  using namespace boost::numeric::ublas;
  vector<double> v1(10), v2(10);
  vector<bool> b1(10), b2(10);
  for (unsigned i = 0; i < v1.size (); ++ i) {
        v1(i) = 1;
        v2(i) = 1;
        b1 (i) = true;
        b2 (i) = false;
  }
  std::cout << (2 + v1) << std::endl;
  std::cout << (v1 + 4) << std::endl;
  std::cout << (2 - v1) << std::endl;
  std::cout << v1 - 4 << std:: endl;
  std::cout << (v1 ^ 2) << std:: endl;
  std::cout << (v1 > 1) << std:: endl;
  std::cout << (1 > v1) << std::endl;
  std::cout << (v1 > v2) << std::endl;
  std::cout << (v1 < 1) << std::endl;
  std::cout << (1 < v1) << std::endl;
  std::cout << (v1 < v2) << std::endl;
  std::cout << (v1 >= 1) << std::endl;
  std::cout << (1 >= v1) << std::endl;
  std::cout << (v1 >= v2) << std::endl;
  std::cout << (v1 <= 1) << std::endl;
  std::cout << (1 <= v1) << std::endl;
  std::cout << (v1 <= v1) << std::endl;
  std::cout << (v1 != 1) << std::endl;
  std::cout << (1 != v1) << std::endl;
  std::cout << (v2 != v1) << std::endl;
  std::cout << (b1 && true) << std::endl;
  std::cout << (true && b1) << std::endl;
  std::cout << (b1 && b2) << std::endl;
  std::cout << (b2 || true) << std::endl;
  std::cout << (true || b2) << std::endl;
  std::cout << (b1 || b2) << std::endl;
  std::cout << (!b2) << std::endl;
}
```

MATRIX OPERATIONS

UNARY OPERATIONS

Description

The template class matrix_unary1 <E, F> and matrix_unary2 <E, F>describes unary matrix operations.

Definition

Defined in the header matrix_expression.hpp

Template parameters

Parameter	Description
E	The type of the matrix expression
F	The type of the operation

Model of

Matrix Expression

Type requirements

None, except for those imposed by the requirements of Matrix Expression

Public base classes

matrix_expression <matrix_unary1 < E, F> > and matrix_expression <matrix_unary2 <E,F> > respectively.

Members

Member	Description
matrix_vector_binary1 (const expression1_type &e1, const expression2_type &e2)	Constructs a description of the expression.
matrix_vector_binary2 (const expression1_type &e1, const expression2_type &e2)	Constructs a description of the expression.
size_type size () const	Returns the size of the expression.
const_reference operator () (size_type i) const	Returns the value of the `i`-th element
const_iterator begin () const	Returns a `const_iterator` pointing to the beginning of the expression.
const_iterator end () const	Returns a `const_iterator` pointing to the end of the expression.
const_reverse_iterator rbegin () const	Returns a `const_reverse_iterator` pointing to the beginning of the reversed expression.
const_reverse_iterator rend () const	Returns a `const_reverse_iterator` pointing to the end of the reversed expression.

Prototypes of unary operations on matrices that return a vector

```
template <class E1, class F>
struct matrix_vector_unary_traits {
typedef unknown storage tag storage category;
typedef row_major_tag orientation_category;
typedef typename E1::value_type value_type;
```

```
        typedef matrix_vector_unary<E1, F> expression_type;
        #ifndef BOOST UBLAS SIMPLE ET DEBUG
        typedef expression_type result_type;
        #else
        typedef typename E1::vector_temporary_type result_type;
        #endif
        };
        //sum m[j] = sum (m[i][j])
        template<class E>
        BOOST UBLAS INLINE
        typename matrix_vector_unary_traits<E, matrix_column_sum< E> >::result_type
        sum (const matrix_expression<E> &e)
        //norm_L1 m[j] = norm_L1 (m[i][j])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix vector unary traits<E, matrix vector norm 1 <E>
        >::result_type
        norm_L1 (const matrix_expression<E> &e)
        //norm_L2 m[j] = norm_L2 (m[i][j])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix vector unary traits<E, matrix vector norm 2 <E>
        >::result_type
        norm_L2 (const matrix_expression<E> &e)
        //norm_inf m[j] = norm_inf (m[i][j])
        template<class E>
        BOOST UBLAS INLINE
        typename matrix_vector_unary_traits<E, matrix_vector_norm_inf <E> >::result_type
        col norm inf (const matrix expression<E> &e)
        //max m[j] = max (m[i][j])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix vector unary traits<E, matrix vector max <E> >::result type
        max (const matrix_expression<E> &e)
        //min m[j] = min (m[i][j])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix_vector_unary_traits<E, matrix_vector_min <E> >::result_type
        min (co //abs min m[j] = abs min (m[i][j])
        //abs_min m[j] = abs_min (m[i][j])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix_vector_unary_traits<E, matrix_vector_abs_min <E>
        >::result_type
        abs min (const matrix expression<E> &e)nst matrix expression<E> &e)
        //abs_max m[j] = abs_max (m[i][j])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix_vector_unary_traits<E, matrix_vector_abs_max <E>
         >::result type
        abs_max (const matrix_expression<E> &e)
        //mean m[j] = mean (m[i][j])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix_vector_unary_traits<E, matrix_vector_mean <E> >::result_type
        mean (const matrix expression<E> &e)
        //abs_mean m[j] = abs_mean (m[i][j])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix_vector_unary_traits<E, matrix_vector_abs_mean <E> >::result_type
        abs_mean (const matrix_expression<E> &e)
        //rms m[j] = rms (m[i][j])
        template<class E>
        BOOST UBLAS INLINE
        typename matrix_vector_unary_traits<E, matrix_vector_rms <E> >::result_type
        rms (const matrix_expression<E> &e)
```

Prototypes of binary operations on matrices that return a matrix

```
// (t + m) [i] [j] = t + m [i] [j]
template<class T1, class E2>
BOOST_UBLAS_INLINE
typename matrix binary scalar1 traits<const T1,     E2,
scalar_plus<T1, typename E2::value_type> >::result_type
operator + (const T1 &e1, const matrix expression<E2> &e2);
// (m + t) [i] [j] = m [i] [j] + t
template<class E1, class T2>
BOOST_UBLAS_INLINE
typename matrix binary scalar2 traits<E1, const T2,
scalar_plus<typename E1::value_type, T2> >::result_type
operator + (const matrix expression<E1> &e1, const T2 &e2);
// (t - m) [i] [j] = t - m [i] [j]
template<class T1, class E2>
BOOST_UBLAS_INLINE
typename matrix binary scalar1 traits<const T1, E2,
scalar_minus<T1, typename E2::value_type> >::result_type
operator - (const T1 &e1, const matrix expression<E2> &e2);
// (m - t) [i] [j] = m [i] [j] - t
template<class E1, class T2>
BOOST_UBLAS_INLINE
typename matrix_binary_scalar2_traits<E1, const T2,
scalar_minus<typename E1::value_type, T2> >::result_type
operator - (const matrix_expression<E1> &e1, const T2 &e2);
// (m ^ t) [i] [j] = m [i] [j] ^ t
template<class E1, class T2>
BOOST_UBLAS_INLINE
typename matrix_binary_scalar2_traits<E1, const T2,
scalar power<typename E1::value type, T2> >::result type
operator ^ (const matrix_expression<E1> &e1, const T2 &e2);
```

Prototypes of unary operations on matrices that return a matrix

```
// (log10 v) [i] = log10 (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename matrix_unary1_traits<E, scalar_log10<typename E::value_type>
>::result_type
log10 (const matrix_expression<E> &e)
// (abs v) [i] = abs (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename matrix_unary1_traits<E, scalar_abs<typename E::value_type>
 >::result_type
abs (const matrix_expression<E> &e)
// (exp v) [i] = exp (v [i])
template<class E>
BOOST UBLAS INLINE
typename matrix_unary1_traits<E, scalar_exp<typename E::value_type>
>::result_type
exp (const matrix_expression<E> &e)
// (log v) [i] = log (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename matrix_unary1_traits<E, scalar_log<typename E::value_type>
>::result_type
log (const matrix expression<E> &e)
// (sqrt v) [i] = sqrt (v [i])
template<class E>
BOOST_UBLAS_INLINE
typename matrix_unary1_traits<E, scalar_sqrt<typename E::value_type>
>::result_type
sqrt  (const matrix_expression<E> &e)
```

```
        // (sin v) [i] = sin (v [i])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix unary1 traits<E, scalar sin<typename E::value type>
        >::result_type
        sin  (const matrix_expression<E> &e)
        // (cos v) [i] = cos (v [i])
        template<class E>
        BOOST_UBLAS_INLINE
        typename matrix unary1 traits<E, scalar cos<typename E::value type>
        >::result_type
        cos (const matrix_expression<E> &e)
```

Examples: Unary operations on matrices

```
        #include "matrix_expression_ext.hpp"
        #include <boost/numeric/ublas/matrix.hpp>
        #include <boost/numeric/ublas/io.hpp>
        int main(){
                using namespace boost::numeric::ublas;
                matrix<double> m (3, 3), mm(3,3);
                for (unsigned i = 0; i < m.size1 (); ++ i){
                        for (unsigned j = 0; j < m.size2 (); ++ j){
                                m (i, j) = (pow(-1.0f,(int)j))*(i+j);
                                mm(i, j) = (i+j + 1);
                        }
                }
                std::cout << abs(m) << std::endl;
                std::cout << log10(mm) << std::endl;
                std::cout << exp(m) << std::endl;
                std::cout << log(mm) << std::endl;
                std::cout << sin(m) << std::endl;
                std::cout << cos(m) << std::endl;
                std::cout << sqrt(mm) << std::endl;
                std::cout << sum(m) << std::endl;
                std::cout << norm_L1(m) << std::endl;
                std::cout << norm_L2(m) << std::endl;
                std::cout << col norm inf(m) << std::endl;
                std::cout << max(m) << std::endl;
                std::cout << min(m) << std::endl;
                std::cout << abs_max(m) << std::endl;
                std::cout << abs_min(m) << std::endl;
                std::cout << mean(m) << std::endl;
                std::cout << abs mean(m) << std::endl;
                std::cout << rms (m) << std::endl;
}
```

BINARY OPERATIONS

Description

The template class matrix_binary <E1, E2, F>, matrix_binary_scalar1 <E1, E2, F> and matrix_binary_scalar2 <E1, E2, F> describe binary matrix operations.

Definition

Defined in the header matrix_expression.hpp

Template parameters

Parameter	Description
E1	The type of first matrix expression
E2	The type of second matrix expression
F	The type of the operation

Model of

Matrix Expression

Type requirements

None, except for those imposed by the requirements of Matrix Expression

Public base classes

matrix_expression <matrix_binary < E1, E2, F>, matrix_expression < matrix_binary_scalar1 <E1, E2, F> > and matrix_expression <matrix_binary_scalar2 <E1, E2, F> > respectively.

Members

Member	Description
matrix_binary (const expression1_type &e1, const expression2_type &e2)	Constructs a description of the expression.
matrix_binary_scalar1 (const expression1_type &e1, const expression2_type &e2)	Constructs a description of the expression.
matrix_binary_scalar2 (const expression1_type &e1, const expression2_type &e2)	Constructs a description of the expression.
size_type size () const	Returns the size of the expression.
const_reference operator () (size_type i) const	Returns the value of the `i`-th element
const_iterator begin () const	Returns a `const_iterator` pointing to the beginning of the expression.
const_iterator end () const	Returns a `const_iterator` pointing to the end of the expression.
const_reverse_iterator rbegin () const	Returns a `const_reverse_iterator` pointing to the beginning of the reversed expression.
const_reverse_iterator rend () const	Returns a `const_reverse_iterator` pointing to the end of the reversed expression.

Prototypes of logical operations on matrices that return Boolean matrix

```
// (t && m) [i] [j] = t && m [i] [j]
template<class E2>
BOOST_UBLAS_INLINE
typename matrix binary scalar1 traits< bool, E2, logical and<bool,
typename E2::value_type> >::result_type
operator && (bool e1, const matrix expression<E2> &e2)
// (m && t) [i] [j] = m [i] [j] && t
template<class E1>
BOOST_UBLAS_INLINE
typename matrix binary scalar2 traits<E1, bool, logical and<typename
E1::value_type, bool> >::result_type
operator && (const matrix expression<E1> &e1,  bool e2)
//(m1 && m2) [i] [j] = m1 [i] [j] && m2 [i] [j]
template<class E1, class E2>
BOOST_UBLAS_INLINE
typename matrix binary traits<E1, E2, logical and
<typename E1::value_type, typename E2::value_type> >::result_type
operator && (const matrix expression<E1> &e1,
const matrix_expression<E2> &e2)
// (t || m) [i] [j] = t || m [i] [j]
template<class E2>
BOOST_UBLAS_INLINE
typename matrix_binary_scalar1_traits<bool, E2,
logical_or<bool, typename E2::value_type> >::result_type
operator || (bool e1, const matrix_expression<E2> &e2)
// (m || t) [i] [j] = m [i] [j] || t
template<class E1>
BOOST_UBLAS_INLINE
typename matrix binary scalar2 traits<E1, bool,
logical_or<typename E1::value_type, bool> >::result_type
operator || (const matrix expression<E1> &e1, bool e2)
//(m1 || m2) [i] [j] = m1 [i] [j] || m2 [i] [j]
template<class E1, class E2>
BOOST_UBLAS_INLINE
typename matrix binary traits<E1, E2,
logical_or<typename E1::value_type,
typename E2::value type> >::result type
operator || (const matrix_expression<E1> &e1,
const matrix_expression<E2> &e2)
//(!m)[i] = !(m[i])
template <class E>
BOOST_UBLAS_INLINE
typename matrix unary1 traits <E, logical not <
typename E::value_type> > :: result_type
operator ! (const matrix_expression <E> &e)
```

Prototypes of comparison operations on matrices that return Boolean matrix

```
// ((m > t) [i][j]  =  m [i][j] > t
template<class E1>
BOOST_UBLAS_INLINE
typename matrix_binary_scalar2_traits<E1,typename
E1::value type, scalar greater<typename
E1::value_type, typename E1::value_type> >::result_type
operator > (const matrix_expression<E1> &e1,
const typename E1::value_type& e2)
// (t > m) [i][j]  = t > m [i][j]
template<class E2>
BOOST UBLAS INLINE
typename matrix_binary_scalar1_traits<typename E2::value_type,
E2, scalar_greater<typename E2::value_type,
typename E2::value_type> >::result_type
operator > (const typename E2::value type& e1,
const matrix_expression<E2> &e2)
// (m1 > m2) [i][j] = m1[i][j] > m2[i][j]
template <class E1, class E2>
```

```
        BOOST_UBLAS_INLINE
        typename matrix binary traits <E1, E2,
        scalar_greater <typename E1::value_type,
        typename E2::value type> >::result type
        operator > (const matrix_expression<E1> &e1,
        const matrix_expression<E2> &e2)
        // (m >= t) [i][j] = m [i][j] >= t
        template<class E1>
        BOOST_UBLAS_INLINE
        typename matrix binary scalar2 traits<E1,
        typename E1::value_type, scalar_greater_equal<typename
        E1::value_type, typename E1::value_type> >::result_type
        operator >= (const matrix_expression<E1> &e1,
        const typename E1::value_type& e2)
        // (t >= m) [i][j] = t >= v [i][j]
        template<class E2>
        BOOST_UBLAS_INLINE
        typename matrix_binary_scalar1_traits<typename
        E2::value_type, E2, scalar_greater_equal<typename
        E2::value_type, typename E2::value_type> >::result_type
        operator >= (const typename E2::value_type& e1,
        const matrix expression<E2> &e2)
        // (m1 >= m2) [i] = m1[i] >= m2[i]
        template <class E1, class E2>
        BOOST_UBLAS_INLINE
        typename matrix_binary_traits <E1, E2, scalar_greater_equal
        <typename E1::value type , typename E2::value type > >::result type
        operator >= (const matrix_expression<E1> &e1,
        const matrix expression<E2> &e2)
        // (m < t) [i][j] = m [i][j] < t
        template<class E1>
        BOOST_UBLAS_INLINE
        typename matrix binary scalar2 traits<E1,
        typename E1::value_type, scalar_lesser<typename E1::value_type,
        typename E1::value type> >::result type
        operator < (const matrix_expression<E1> &e1,
        const typename E1::value_type& e2)
        // (t < m) [i][j] = t < m [i][j]
        template<class E2>
        BOOST_UBLAS_INLINE
        typename matrix binary scalar1 traits<typename E2::value type,
        E2, scalar_lesser<typename E2::value_type,
        typename E2::value_type> >::result_type
        operator < (const typename E2::value_type& e1,
        const matrix expression<E2> &e2)
        // (m1 < m2) [i] = m1[i] < m2[i]
        template <class E1, class E2>
        BOOST_UBLAS_INLINE
        typename matrix_binary_traits <E1, E2,
        scalar lesser <typename E1::value type ,
        typename E2::value_type > >::result_type
        operator < (const matrix expression<E1> &e1,
        const matrix_expression<E2> &e2)
        // (m <= t) [i][j] = m [i][j] <= t
        template<class E1>
        BOOST UBLAS INLINE
        typename matrix_binary_scalar2_traits<E1,
        typename E1::value type, scalar lesser equal<typename
        E1::value_type, typename E1::value_type> >::result_type
        operator <= (const matrix_expression<E1> &e1,
        const typename E1::value_type& e2)
        // (t <= m) [i][j] = t <= m [i][j]
        template<class E2>
        BOOST UBLAS INLINE
        typename matrix_binary_scalar1_traits<typename
        E2::value_type, E2, scalar_lesser_equal<typename
        E2::value_type, typename E2::value_type> >::result_type
        operator <= (const typename E2::value_type& e1,
        const matrix_expression<E2> &e2)
        // (m1 <= m2) [i] = m1[i] <= m2[i]
        template <class E1, class E2>
```

```
BOOST_UBLAS_INLINE
typename matrix binary traits <E1, E2,
scalar_lesser_equal <typename E1::value_type ,
typename E2::value type > >::result type
operator <= (const matrix_expression<E1> &e1,
const matrix_expression<E2> &e2)
// (m != t) [i][j] = m [i][j] != t
template<class E1>
BOOST_UBLAS_INLINE
typename matrix binary scalar2 traits<E1,
typename E1::value_type, scalar_inequality<typename
E1::value_type, typename E1::value_type> >::result_type
operator != (const matrix_expression<E1> &e1,
const typename E1::value_type& e2)
// (t != m) [i][j] = t != m [i][j]
template<class E2>
BOOST_UBLAS_INLINE
typename matrix_binary_scalar1_traits<typename
E2::value_type, E2, scalar_inequality<typename
E2::value_type, typename E2::value_type> >::result_type
operator != (const typename E2::value_type& e1,
const matrix expression<E2> &e2)
// (m1 != m2) [i] = m1[i] != m2[i]
template <class E1, class E2>
BOOST_UBLAS_INLINE
typename matrix_binary_traits <E1,
E2, scalar inequality <typename E1::value type ,
typename E2::value_type > >::result_type
operator != (const matrix expression<E1> &e1,
const matrix_expression<E2> &e2)
```

Examples: Binary operations on matrices

```
#include "matrix_expression_ext.hpp"
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
int main () {
        using namespace boost::numeric::ublas;
        matrix<double> m (3, 3), mm(3,3);
        matrix<bool> mb1(3,3), mb2(3,3);
        for (unsigned i = 0; i < m.size1 (); ++ i){
                for (unsigned j = 0; j < m.size2 (); ++ j){
                        m (i, j) = (pow(-1.0,(int)j))*(i+j);
                        mm(i, j) = (i+j);
                        mb1 (i,j)= true;
                        mb2 (i,j)= false;
                }
        }
        std::cout << 1 + m << std::endl;
        std::cout << m + 1 << std::endl;
        std::cout << 1 - m << std::endl;
        std::cout << m - 1 << std::endl;
        std::cout << (m ^ 2) << std::endl;
        std::cout << (true && mb1) << std::endl;
        std::cout << (mb1 && true) << std::endl;
        std::cout << (mb1 && mb2) << std::endl;
        std::cout << (true || mb2) << std::endl;
        std::cout << (mb2 || true) << std::endl;
        std::cout << (mb1 || mb2) << std::endl;
        std::cout << (!mb1) << std::endl;
        std::cout << (m > 3) << std::endl;
        std::cout << (m >= 3) << std::endl;
        std::cout << (m > mm) << std::endl;
        std::cout << (5 > m) << std::endl;
        std::cout << (5 >= m) << std::endl;
        std::cout << (m >= mm) << std::endl;
        std::cout << (m < 3) << std::endl;
        std::cout << (3 < m) << std::endl;
```

```
                std::cout << (m < mm) << std::endl;
                std::cout << (3 <= m) << std::endl;
                std::cout << (m <= 3) << std::endl;
                std::cout << (m <= mm) << std::endl;
                std::cout << (3 != m) << std::endl;
                std::cout << (m != 3) << std::endl;
                std::cout << (m != mm) << std::endl;
                std::cout << ((m <= 3) && (m > mm)) << std::endl;
}
```

TILING ADAPTORS

VECTOR TILING

Description

The template class vector_tile<M>describes tiling of a vector to form a matrix.

Definition

Defined in the header tile_vector.hpp

Template parameters

Parameter	Description
M	The type of the vector

Model of

Matrix Expression

Type requirements

None, except for those imposed by the requirements of Matrix Expression

Public base classes

matrix_expression <vector_tile <M> >

Members

Member	Description
vector_tile(matrix_type &data)	Constructs a description of the expression.
vector_tile (const vector_tile &m)	Constructs a description of the expression.
vector_tile (matrix_type &data, size_type rows, size_type cols)	Returns the size of the expression.
size_type size1 () const	Returns the value of the `i`-th element
size_type size2 () const	Returns a `const_iterator` pointing to the beginning of the expression.
const_iterator1 begin1 () const	Returns a const_iterator1 pointing to the beginning of the vector_tile.
const_iterator1 end1 () const	Returns a const_iterator1 pointing to the end of the vector_tile
iterator1 begin1 ()	Returns a iterator1 pointing to the beginning of the vector_tile
iterator1 end1 ()	Returns a iterator1 pointing to the end of the vector_tile.
const_iterator2 begin2 () cons	Returns a const_iterator2 pointing to the beginning of the vector_tile.
const_iterator2 end2 () const	Returns a const_iterator2 pointing to the end of the vector_tile
iterator2 begin2 ()	Returns a iterator2 pointing to the beginning of the vector_tile.
iterator2 end2 ()	Returns a iterator2 pointing to the end of the vector_tile
const_reverse_iterator1 rbegin1 () const	Returns a const_reverse_iterator1 pointing to the beginning of the reversed vector_tile.
const_reverse_iterator1 rend1 () const	Returns a const_reverse_iterator1 pointing to the end of the reversed vector_tile.
reverse_iterator1 rbegin1 ()	Returns a reverse_iterator1 pointing to the beginning of the reversed vector_tile.
reverse_iterator1 rend1 ()	Returns a reverse_iterator1 pointing to the end of the reversed vector_tile.
const_reverse_iterator2 rbegin2 () const	Returns a const_reverse_iterator2 pointing to the beginning of the reversed vector_tile
const_reverse_iterator2 rend2 () const	Returns a const_reverse_iterator2 pointing to the end of the reversed vector_tile.
reverse_iterator2 rbegin2 ()	Returns a reverse_iterator2 pointing to the beginning of the reversed vector_tile.
reverse_iterator2 rend2 ()	Returns a reverse_iterator2 pointing to the end of the reversed vector_tile.

Examples: Vector tile

```
#include "tile_vector.hpp"
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/io.hpp>

int main(){
  using namespace boost::numeric::ublas;
  matrix<double,column_major> matprod (1,10);
  vector<double> v (10);

  // Create 10 x 100 matrix tiling a 1x10 vector
  vector tile < vector <double> > add1(v, 10,10);

  for (std::size_t i = 0; i < v.size (); ++ i)    v (i) = (i+1);
  // Multiplies a ( 10 x 100 ) matrix by (100 x 10) matrix
  matprod.assign(prod <matrix<double,column major> > (matvec(v,1,10),matvec(v,100,1)));

  std::cout << "vector" << v << std::endl;
  std::cout << "matprod" << matprod << std::endl;

  // Access every element of a tile using iterators
  typedef vector_tile <vector <double> > :: iterator1 i1_t;
  typedef vector_tile <vector <double> > :: iterator2 i2_t;
  for (i1_t i1 = add1.begin1(); i1 != add1.end1(); ++i1) {
       for (i2_t i2 = i1.begin(); i2 != i1.end()(); ++i2)
                 std::cout << "(" << i2.index1() << "," << i2.index2()
                                    << ":" << *i2 << ")   ";
             std::cout << std::endl;
  }
}
```

MATRIX TILING

Description

The template class matrix_tile<M> describes tiling of a matrix into matrix.

Definition

Defined in the header tile_matrix.hpp

Template parameters

Parameter	Description
M	The type of the matrix expression

Model of

Matrix Expression

Type requirements

None, except for those imposed by the requirements of Matrix Expression

Public base classes

matrix_expression <matrix_tile < M> >

Members

Member	Description
matrix_tile(matrix_type &data)	Constructs a description of the expression.
matrix_tile (const matrix_tile &m)	Constructs a description of the expression.
matrix_tile (matrix_type &data, size_type rows, size_type cols)	Returns the size of the expression.
size_type size1 () const	Returns the value of the `i`-th element
size_type size2 () const	Returns a `const_iterator` pointing to the beginning of the expression.
const_iterator1 begin1 () const	Returns a const_iterator1 pointing to the beginning of the matrix_tile.
const_iterator1 end1 () const	Returns a const_iterator1 pointing to the end of the matrix_tile
iterator1 begin1 ()	Returns a iterator1 pointing to the beginning of the matrix_tile
iterator1 end1 ()	Returns a iterator1 pointing to the end of the matrix_tile.
const_iterator2 begin2 () cons	Returns a const_iterator2 pointing to the beginning of the matrix_tile.
const_iterator2 end2 () const	Returns a const_iterator2 pointing to the end of the matrix_tile
iterator2 begin2 ()	Returns a iterator2 pointing to the beginning of the matrix_tile.
iterator2 end2 ()	Returns a iterator2 pointing to the end of the matrix_tile
const_reverse_iterator1 rbegin1 () const	Returns a const_reverse_iterator1 pointing to the beginning of the reversed matrix_tile.
const_reverse_iterator1 rend1 () const	Returns a const_reverse_iterator1 pointing to the end of the reversed matrix_tile.
reverse_iterator1 rbegin1 ()	Returns a reverse_iterator1 pointing to the beginning of the reversed matrix_tile.
reverse_iterator1 rend1 ()	Returns a reverse_iterator1 pointing to the end of the reversed matrix_tile.

const_reverse_iterator2 rbegin2 () const	Returns a const_reverse_iterator2 pointing to the beginning of the reversed matrix_tile
const_reverse_iterator2 rend2 () const	Returns a const_reverse_iterator2 pointing to the end of the reversed matrix_tile.
reverse_iterator2 rbegin2 ()	Returns a reverse_iterator2 pointing to the beginning of the reversed matrix_tile.
reverse_iterator2 rend2 ()	Returns a reverse_iterator2 pointing to the end of the reversed matrix_tile.

Examples: Matrix tile

```
#include "tile_matrix.hpp"
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>

int main(){
  using namespace boost::numeric::ublas;
  typedef matrix<double,column_major> matrix_type;

  matrix type m(10,10);
  matrix_type matprod (10,10);
  //Tile a 10 x 10 matrix to form a 40 x 40 matrix
  matrix_tile < matrix_type > add1(m, 4, 4);

  // No data was copied during tiling so it is
  // OK to initialize the matrix being tiled here!
  for (std::size_t i = 0; i < m. size1 () ; ++i)
         for (std::size_t j =0; j < m.size2 () ; ++j)
                m (i,j) = i + j;

  // Multiply 10 x 20 matrix by 20 x 10 matrix
  matprod.assign(prod <matrix type> (matmat(m,1,2),matmat(m,2,1)));

  std::cout << "matrix" << m << std::endl;
  std::cout << "matprod" << matprod << std::endl;

  //Access each element of the tile by iterators
  typedef matrix tile < matrix type > :: iterator1 i1 t;
  typedef matrix_tile < matrix_type > :: iterator2 i2_t;
       for (i1_t i1 = add1.begin1(); i1 != add1.end1(); ++i1) {
          for (i2_t i2 = i1.begin(); i2 != i1.end()(); ++i2)
                std::cout << "(" << i2.index1() << "," << i2.index2()
                                          << ":" << *i2 << ")";
       std::cout << std::endl;
  }
}
```

RESHAPING ADAPTORS

VECTOR RESHAPE

Description

The template class vector_reshape<M>describes reshapeing of a vector into matrix.

Definition

Defined in the header vector_reshape.hpp

Template parameters

Parameter	Description
M	The type of the vector

Model of

Matrix Expression

Type requirements

None, except for those imposed by the requirements of Matrix Expression

Public base classes

matrix_expression <vector_reshape <M> >

Members

Member	Description
vector_reshape(matrix_type &data)	Constructs a description of the expression.
vector_reshape (const vector_reshape &m)	Constructs a description of the expression.
vector_reshape (matrix_type &data, size_type rows, size_type cols)	Returns the size of the expression.
size_type size1 () const	Returns the value of the `i`-th element
size_type size2 () const	Returns a `const_iterator` pointing to the beginning of the expression.
const_iterator1 begin1 () const	Returns a const_iterator1 pointing to the beginning of the vector_reshape.
const_iterator1 end1 () const	Returns a const_iterator1 pointing to the end of the vector_reshape
iterator1 begin1 ()	Returns a iterator1 pointing to the beginning of the vector_reshape
iterator1 end1 ()	Returns a iterator1 pointing to the end of the vector_reshape.
const_iterator2 begin2 () cons	Returns a const_iterator2 pointing to the beginning of the vector_reshape.
const_iterator2 end2 () const	Returns a const_iterator2 pointing to the end of the vector_reshape

iterator2 begin2 ()	Returns a iterator2 pointing to the beginning of the vector_reshape.
iterator2 end2 ()	Returns a iterator2 pointing to the end of the vector_reshape
const_reverse_iterator1 rbegin1 () const	Returns a const_reverse_iterator1 pointing to the beginning of the reversed vector_reshape.
const_reverse_iterator1 rend1 () const	Returns a const_reverse_iterator1 pointing to the end of the reversed vector_reshape.
reverse_iterator1 rbegin1 ()	Returns a reverse_iterator1 pointing to the beginning of the reversed vector_reshape.
reverse_iterator1 rend1 ()	Returns a reverse_iterator1 pointing to the end of the reversed vector_reshape.
const_reverse_iterator2 rbegin2 () const	Returns a const_reverse_iterator2 pointing to the beginning of the reversed vector_reshape
const_reverse_iterator2 rend2 () const	Returns a const_reverse_iterator2 pointing to the end of the reversed vector_reshape.
reverse_iterator2 rbegin2 ()	Returns a reverse_iterator2 pointing to the beginning of the reversed vector_reshape.
reverse_iterator2 rend2 ()	Returns a reverse_iterator2 pointing to the end of the reversed vector_reshape.

Examples: Vector reshape

```
        #include "vector_reshape.hpp"
        #include <boost/numeric/ublas/matrix.hpp>
        #include <boost/numeric/ublas/vector.hpp>
        #include <boost/numeric/ublas/io.hpp>
        int main(){
          using namespace boost::numeric::ublas;
          typedef matrix<double,column_major> matrix_type;

          vector<double> v (10);
          matrix_type matprod (2,2);

          // Adapt a 1 x 10 vector as 2 x 5 matrix
          vector_reshape < vector <double> > add1(v, 2, 5);

          // It is OK to initialize here since the adaptor
          // doees not copy data
          for (std::size_t i = 0; i < v.size (); ++ i)   v (i) = (i+1)+0.00;

          // Multiply 2 x 5 matrix with 5 x 2 matrix
          matprod.assign(prod <matrix_type> ( vec_resh (v,2,5), vec_resh (v,5,2)));

          std::cout << "vector" << v << std::endl;
          std::cout << "matprod" << matprod << std::endl;

          // Acess each element using iterators
          typedef vector_reshape <vector <double> > :: iterator1 i1_t;
          typedef vector_reshape <vector <double> > :: iterator2 i2_t;
          for (i1_t i1 = add1.begin1(); i1 != add1.end1(); ++i1) {
                  for (i2_t i2 = i1.begin(); i2 != i1.end(); ++i2)
                                  std::cout << "(" << i2.index1() << "," << i2.index2()
                                                                  << ":" << *i2 << ")   ";
                      std::cout << std::endl;
          }
}
```

MATRIX RESHAPE

Description

The template class matrix_reshape<M> describes reshaping of a matrix into another matrix.

Definition

Defined in the header matrix_reshape.hpp

Template parameters

Parameter	Description
M	The type of the matrix expression

Model of

Matrix Expression

Type requirements

None, except for those imposed by the requirements of Matrix Expression

Public base classes

matrix_expression <matrix_reshape < M>

Members

Member	Description
matrix_reshape(matrix_type &data)	Constructs a description of the expression.
matrix_reshape (const matrix_reshape &m)	Constructs a description of the expression.
matrix_reshape (matrix_type &data, size_type rows, size_type cols)	Returns the size of the expression.
size_type size1 () const	Returns the value of the `i`-th element
size_type size2 () const	Returns a `const_iterator` pointing to the beginning of the expression.
const_iterator1 begin1 () const	Returns a const_iterator1 pointing to the beginning of the matrix_reshape.
const_iterator1 end1 () const	Returns a const_iterator1 pointing to the end of the matrix_reshape

iterator1 begin1 ()	Returns a iterator1 pointing to the beginning of the matrix_reshape
iterator1 end1 ()	Returns a iterator1 pointing to the end of the matrix_reshape.
const_iterator2 begin2 () cons	Returns a const_iterator2 pointing to the beginning of the matrix_reshape.
const_iterator2 end2 () const	Returns a const_iterator2 pointing to the end of the matrix_reshape
iterator2 begin2 ()	Returns a iterator2 pointing to the beginning of the matrix_reshape.
iterator2 end2 ()	Returns a iterator2 pointing to the end of the matrix_reshape
const_reverse_iterator1 rbegin1 () const	Returns a const_reverse_iterator1 pointing to the beginning of the reversed matrix_reshape.
const_reverse_iterator1 rend1 () const	Returns a const_reverse_iterator1 pointing to the end of the reversed matrix_reshape.
reverse_iterator1 rbegin1 ()	Returns a reverse_iterator1 pointing to the beginning of the reversed matrix_reshape.
reverse_iterator1 rend1 ()	Returns a reverse_iterator1 pointing to the end of the reversed matrix_reshape.
const_reverse_iterator2 rbegin2 () const	Returns a const_reverse_iterator2 pointing to the beginning of the reversed matrix_reshape
const_reverse_iterator2 rend2 () const	Returns a const_reverse_iterator2 pointing to the end of the reversed matrix_reshape.
reverse_iterator2 rbegin2 ()	Returns a reverse_iterator2 pointing to the beginning of the reversed matrix_reshape.
reverse_iterator2 rend2 ()	Returns a reverse_iterator2 pointing to the end of the reversed matrix_reshape.

Examples: Matrix reshape

```
#include "matrix_reshape.hpp"
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
int main(){
  using namespace boost::numeric::ublas;
  typedef matrix<double,column_major> matrix_type;

  matrix_type test (4,5);
  matrix type matprod (2,2);

  // Reshape a 4 x 5 matrix and acess it as 10 x 2 matrix
  matrix_reshape < matrix_type > add1(test, 10, 2);

  // It is OK to initialize here since no data was copied
  // in previous step
  for (std::size_t i = 0; i < test.size1 (); ++ i)
       for(std::size t j =0; j < test.size2 (); ++j) test (i,j) = (i+1);

  // Multiply 2 x 10 matrix by 10 x 2 matrix
  matprod.assign(prod <matrix_type> ( matrix_reshape<matrix_type> (test,2,10),
                                matrix_reshape<matrix_type> (test,10,2)));

  std::cout << "matrix" << test << std::endl;
  std::cout << "matprod" << matprod << std::endl;
```

```
        // Acess each element of the reshaped matrix with iterators
        typedef matrix_reshape <matrix_type > :: iterator1 i1_t;
        typedef matrix reshape <matrix type > :: iterator2 i2 t;
        for (i1_t i1 = add1.begin1(); i1 != add1.end1(); ++i1) {
                for (i2_t i2 = i1.begin(); i2 != i1.end(); ++i2)
                                std::cout << "(" << i2.index1() << "," << i2.index2()
                                                                << ":" << *i2 << ")  ";
                    std::cout << std::endl;
        }
}
```

INDEXING EXTENSIONS

MATLAB TAG: LAST_ELEM

Description

The template class last_elem<D = std::ptrdiff> describes a tag that allows indexing of vectors and matrices relative to the last element.

Definition

Defined in the header matlab_tags.hpp

Template parameters

Parameter	Description
D	Indexing difference type

Model of

None

Type requirements

D needs to be an integral type

Public base classes

None

Members

Member	Description
`last_elem()`	Represents an index with zero offset from last element
`last_elem(difference_type offset)`	Represents an index with an offset of 'offset' from last element
`last_elem(const last_elem<D>& other)`	Copy constructor
`difference_type get_offset()`	Gets the offset relative to last element

Global Operators

Prototypes	Description
`last_elem<T> operator + (const last_elem<T>& elem, T offset);` `last_elem<T> operator + (T offset, const last_elem<T>& elem)`	Equvalent to indices: last + offset, offset + last
`last_elem<T> operator + (const last_elem<T>& elem, T offset);` `last_elem<T> operator + (T offset, const last_elem<T>& elem)`	Equvalent to indices: last - offset, offset - last

Macros

Declaration	Description
`#define END last_elem<>()`	end tag

Examples: End element

```
#include "expression_types_ext.hpp"                    //  Includes indexing extensions
#include <boost\numeric\ublas\vector.hpp>
#include <boost\numeric\ublas\matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
int main(){
        using namespace boost::numeric::ublas;
        vector<int> examp(10, 1);
        for(std::size_t ii = 0; ii < 10; ii++){
                examp[ii] = ii;
        }
        std::cout << "Last vector element: " << examp(END) << std::endl;
        std::cout << "Last but one element: " << examp(END - 1) << std::endl;
        std::cout << "Last but one element: " << examp(-1 + END) << std::endl;
        matrix<int> matExamp(10, 10, 10);
        for(std::size_t ii = 0; ii < 10; ii++)
                for(std::size_t jj = 0; jj < 10; jj++)
                        matExamp(ii, jj) = ii*10 + jj;

        std::cout << "Element from last row last column:" <<
                  matExamp(END, END)<< std::endl;
        std::cout << "Element from first row first column:" <<
                   matExamp(END - 9, END - 9)<< std::endl;
        std::cout << "Element from first row first column:" <<
                   matExamp(-9 + END, -9 + END)<< std::endl;
        return 0;
}
```

MATLAB TAG: ALL, REV_ALL

Description

The classes all_elem and rev_all describe tag that allows acess to all elements or all elements in reverse order

Definition

Defined in the header matlab_tags.hpp

Model of

None

Type requirements

None

Public base classes

None

Members

None

Macros

Declaration	Description
#define ALL all_elem<>()	All elements tag
#define REV_ALL rev_all<>()	All elements in reverse order

Examples: End element

```
#include "expression types ext.hpp"                    //  Includes indexing extensions
#include <boost\numeric\ublas\vector.hpp>
#include <boost\numeric\ublas\matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
int main(){
        using namespace boost::numeric::ublas;
        vector<int> examp(10, 1);
        for(std::size_t ii = 0; ii < 10; ii++){
                examp[ii] = ii;
        }
        std::cout << "All vector elements: " << examp(ALL) << std::endl;
        std::cout << "All vector elements in reverse: " << examp(REV_ALL) << std::endl;
```

```
                matrix<int> matExamp(10, 10, 10);
                for(std::size_t ii = 0; ii < 10; ii++)
                        for(std::size_t jj = 0; jj < 10; jj++)
                                matExamp(ii, jj) = ii*10 + jj;

                std::cout << "All Elements from last row:" <<
                          matExamp(END, ALL)<< std::endl;
                std::cout << "Revrse the whole matrix:" <<
                           matExamp(REV_ALL, REV_ALL)<< std::endl;
                return 0;
        }
```

SIMPLIFIED RANGE AND SLICE EXTENSIONS

Description

The template class basic_slice_ext <class Z = std::size_t, class D = std::ptrdiff_t> describes a simple template class that allows easy Matlab like range and slice access to vector and matrices

Definition

Defined in the header matlab_tags.hpp

Template parameters

Parameter	Description
Z	Indexing size type
D	Indexing difference type

Model of

None

Type requirements

Z, D need to be an integral type

Public base classes

None

Members

Member	Description
`basic_slice_ext(const last_elem<D>& start, difference_type stride, const last_elem<D>& end);`	Represents a slice starting with start index ‘start’ skipping ‘stride’ elements till ‘end’ element. This extension allows both zero

`basic_slice_ext(const last_elem<D>& start, difference_type stride, size_type end);`
`basic_slice_ext(size_type start, difference_type stride, const last_elem<D>& end);`
`basic_slice_ext(const last_elem<D>& start, const last_elem<D>& end);`
`basic_slice_ext(size_type start, difference_type stride, size_type end);`
	based indexing and indexing relative to last element

`basic_slice_ext(const last_elem<D>& start, size_type end);`
`basic_slice_ext(size_type start, const last_elem<D>& end);`
`basic_slice_ext(size_type start, size_type end);`
	Represents a range starting with start index 'start' till 'end' element. This extension allows both zero based indexing and indexing relative to last element

Macros

Declaration	Description
`#define RG(x, y)` `(boost::numeric::ublas::basic_slice_ext<>((x),(y)))`	Easy range
`#define SL(x, y, z)` `(boost::numeric::ublas::basic_slice_ext<>((x),(y),(z)))`	Easy slice

Examples: Easy range and slice

```
#include "expression_types_ext.hpp"                    //  Includes indexing extensions
#include <boost\numeric\ublas\vector.hpp>
#include <boost\numeric\ublas\matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
int main(){
        using namespace boost::numeric::ublas;
        vector<int> examp(10, 1);
        for(std::size_t ii = 0; ii < 10; ii++){
                examp[ii] = ii;
        }
        std::cout << "Same as all elements: " << examp(RG(0, END)) << std::endl;
        std::cout << "Elements at odd indices: " << examp(SL(1, 2, END)) << std::endl;
        std::cout << "elements at even indices: " << examp(SL(0, 2, END)) << std::endl;

        matrix<int> matExamp(10, 10, 10);
        for(std::size_t ii = 0; ii < 10; ii++)
                for(std::size_t jj = 0; jj < 10; jj++)
                        matExamp(ii, jj) = ii*10 + jj;



        std::cout << "Odd index elments from third row:" <<
                  matExamp(2, SL(1, 2, END))<< std::endl;
        std::cout << "Odd indexed elements for enitre matrix in reverse:" <<
                  matExamp(SL(END, -2, 0), SL(END, -2, 0))<< std::endl;
        return 0;
}
```

VECTOR AND MATRIX FIND FUNCTION

Description

Global template function find() that can be used to search vector and matrix expressions for elements satisfying a Boolean expression.

Definition

Defined in the header ublas_find.hpp

Usage

Calling convention	Description
find(vectBoolExpression, vectExpressiontoSearch)	find(boolean vector expression exprCond, vector expression for return values exprSearch) , returns exprSearch(elements where exprCond == true)
find(vectBoolExpression, vectExpressiontoSearch , indexVect)	find(boolean vector expression exprCond, vector expression for return values exprSearch, indices where boolean expression evaluates to true), returns exprSearch(elements where exprCond == true)
find(vectBoolExpression , indexVect)	find(boolean vector expression exprCond, indices where boolean expression evaluates to true), returns an indirect array that can be used to index
find(matBoolExpression, matExpressiontoSearch)	find(boolean matrix expression exprCond, matrix expression for return values exprSearch) , returns exprSearch(elements where exprCond == true)
find(matBoolExpression, matExpressiontoSearch, indexVect1, indexVect2)	find(boolean matrix expression exprCond, matrix expression for return values exprSearch, row indices where boolean expression evaluates to true, column indices where boolean expression evaluates to true), returns exprSearch(elements where exprCond == true) - linear indexed matrix
find(matBoolExpression, indexVect1, indexVect2)	find(boolean matrix expression exprCond, row indices where boolean expression evaluates to true, column indices where boolean expression evaluates to true)

Type requirements

First argument needs to be a matrix or vector expression with a value_type bool

Preconditions

Size of the Boolean expression and the size of the expression to searh must be the same

Complexity

Linear at most

Prototypes

```
        // find(boolean vector expression exprCond,
        //             vector expression for return values exprSearch)
        // returns exprSearch(elements where exprCond == true)
        template<class E, class E1>
        vector indirect<const E1, indirect array<unbounded array<typename E1::size type> > >
        find(const vector_expression<E>& exprCond,
                const vector_expression<E1>& exprSearch,
               typename boost::enable_if<
                  typename boost::is_same<typename E::value_type, bool>::type>::type *dummy = 0);
        // find(boolean vector expression exprCond,
        //             vector expression for return values exprSearch,
        //             indices where boolean expression evaluates to true)
        // returns exprSearch(elements where exprCond == true)
        template<class E, class E1>
        vector_indirect<const E1, indirect_array<unbounded_array<typename E1::size_type> > >
        find(const vector expression<E>& exprCond,
               const vector_expression<E1>& exprSearch,
               vector<typename E1::size_type>& index,
               typename boost::enable_if<
                typename boost::is_same<typename E::value_type, bool>::type>::type *dummy = 0);
        // find(boolean vector expression exprCond,
        //             indices where boolean expression evaluates to true)
        // returns an indirect array that can be used to index
        template<class E, class S>
        indirect_array<unbounded_array<typename E::size_type> >
        find(const vector_expression<E>& exprCond,
               vector<S>& index,
               typename boost::enable if<
                typename boost::is_same<typename E::value_type, bool>::type>::type *dummy = 0);
        // find(boolean matrix expression exprCond,
        //             matrix expression for return values exprSearch)
        // returns exprSearch(elements where exprCond == true)
        template<class E, class E1>
        matrix_vector_indirect<const E1,
              indirect array<unbounded array<typename E1::size type> > >
        find(const matrix_expression<E>& exprCond,
                const matrix_expression<E1>& exprSearch,
               typename boost::enable_if<
                typename boost::is same<typename E::value type, bool>::type>::type *dummy = 0);
        // find(boolean matrix expression exprCond,
        //             matrix expression for return values exprSearch,
        //             row indices where boolean expression evaluates to true,
        //             column indices where boolean expression evaluates to true)
        // returns exprSearch(elements where exprCond == true) - linear indexed matrix
        template<class E, class E1>
        matrix_vector_indirect<const E1,
              indirect array<unbounded array<typename E1::size type> > >
        find(const matrix_expression<E>& exprCond,
                const matrix_expression<E1>& exprSearch,
                vector<typename E1::size_type>& index1,
                vector<typename E1::size type>& index2,
               typename boost::enable_if<
                typename boost::is_same<typename E::value_type, bool>::type>::type *dummy = 0);
```

```
// find(boolean matrix expression exprCond,
//             row indices where boolean expression evaluates to true,
//             column indices where boolean expression evaluates to true)
template<class E, class S>
void find(const matrix_expression<E>& expr,
         vector<S>& index1,
         vector<S>& index2,
         typename boost::enable_if<
          typename boost::is_same<typename E::value_type, bool>::type>::type *dummy = 0);
```

Examples: Using find

```
#include "expression_types_ext.hpp"                    //  Includes indexing extensions
#include "matrix expression ext.hpp"
#include "vector_expression_ext.hpp"
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include "ublas find.hpp"
#include <boost/numeric/ublas/io.hpp>
int main(){
        using namespace boost::numeric::ublas;
        // Initialize sample vectors
        vector<int> examp(10, 1), examp2(10, 1);
        for(std::size_t ii = 0; ii < 10; ii++){
                examp[ii] = ii;
                examp2[ii] = 10 - ii;
        }

        // Initialize sample matrix
        matrix<int> matExamp(10, 10, 10);
        for(std::size t ii = 0; ii < 10; ii++)
                for(std::size_t jj = 0; jj < 10; jj++)
                        matExamp(ii, jj) = ii*10 + jj;

        // Find elements of matrix matExamp satisfying a condition
        std::cout << find(((matExamp > 1) && (matExamp < 10)), matExamp) << std::endl;
        // Find elements of vector expression  (examp + 1) satisfying a boolean conditionm
        std::cout << find(((examp < 1) && (examp2 >= 5) ), examp + 1) << std::endl;
        return 0;

}
```