Boost.uBlas: The Boost uBlas Library

Table of Contents

Basic Linear and Multilinear Algebra Library

uBLAS is a C++ template class library that provides BLAS level 1, 2, 3 functionality for dense, packed and sparse matrices. The design and implementation unify mathematical notation via operator overloading and efficient code generation via expression templates. Since 2018, uBLAS also supports basic operations for multilinear algebra operations such as the outer and inner product of higher-order tensors. The goal of the new tensor extension is to provide basic operations that simplify the implementation of e.g. machine learning and quantum computing algorithms.

Functionality

uBLAS provides templated C++ classes for dense, unit and sparse vectors, dense, identity, triangular, banded, symmetric, hermitian and sparse matrices. Views into vectors and matrices can be constructed via ranges, slices, adaptor classes and indirect arrays. The library covers the usual basic linear and multilinear algebra operations on vectors, matrices and tensors: reductions like different norms, addition and subtraction of vectors and matrices and multiplication with a scalar, inner and outer products of vectors, matrix vector and matrix matrix products and triangular solver. Similar operations are also provided for the tensor type. The glue between containers, views and expression templated operations is a mostly STL conforming iterator interface.

Known limitations

- The implementation assumes a linear memory address model.
- Tuning was focussed on dense matrices.

Further Information

- uBLAS mailing list
- uBLAS wiki
- Code

Authors and Credits

uBLAS initially was written by Joerg Walter and Mathias Koch. We would like thank all contributors who has supported this library. Amongst many contributors around the world, David Abrahams, Ed Brey, Fernando Cacciola, Juan Jose Gomez Cadenas, Beman Dawes, Matt Davies, Bob Fletcher, Kresimir Fresl, Joachim Kessel, Patrick Kowalzick, Toon Knapen, Hendrik Kueck, John Maddock, Jens Maurer, Alexei Novakov, Gary Powell, Joachim Pyras, Peter Schmitteckert, Jeremy Siek, Markus Steffl, Michael Stevens, Benedikt Weber, Martin Weiser, Gunter Winkler, Marc Zimmermann, Marco Guazzone, Nasos Iliopoulus and the members of Boost had a great impact and contribution helping the library to grow and mature.

Starting with the GSoC 2018 project, uBlas has been extended by a flexible tensor data type and basic tensor operations supporting general tensor contractions and the Einstein notation. The goal of the new tensor extension is to support the implementation of algorithms for e.g. machine learning and quantum computing applications. The initial implementation of this extension is written by Cem Bassoy. Contributors of the uBLAS extension are Amit Singh, Ashar Khan, Stefan Seefeld, Cem Bassoy and the members of Boost.

This library is currently maintained by David Bellot, Cem Bassoy and Stefan Seefeld.

Frequently Asked Questions

Q: Should I use uBLAS for new projects?
A: At the time of writing (09/2012) there are a lot of good matrix libraries available, e.g., MTL4, armadillo, eigen. uBLAS offers a stable, well tested set of vector and matrix classes, the typical operations for linear algebra and solvers for triangular systems of equations. uBLAS offers dense, structured and sparse matrices - all using similar interfaces. And finally uBLAS offers good (but not outstanding) performance. On the other side, the last major improvement of uBLAS was in 2008 and no significant change was committed since 2009. So one should ask himself some questions to aid the decision: *Availability?* uBLAS is part of boost and thus available in many environments. *Easy to use?* uBLAS is easy to use for simple things, but needs decent C knowledge when you leave the path. _Performance?_ There are faster alternatives. _Cutting edge?_ uBLAS is more than 10 years old and missed all new stuff from C11.

Q: I'm running the uBLAS dense vector and matrix benchmarks. Why do I see a significant performance difference between the native C and library implementations?
A: uBLAS distinguishes debug mode (size and type conformance checks enabled, expression

templates disabled) and release mode (size and type conformance checks disabled, expression templates enabled). Please check, if the preprocessor symbol `NDEBUG` of `cassert` is defined. `NDEBUG` enables release mode, which in turn uses expression templates. You can optionally define `BOOST_UBLAS_NDEBUG` to disable all bounds, structure and similar checks of uBLAS.

Q: I've written some uBLAS tests, which try to incorrectly assign different matrix types or overrun vector and matrix dimensions. Why don't I get a compile time or runtime diagnostic?
A: uBLAS distinguishes debug mode (size and type conformance checks enabled, expression templates disabled) and release mode (size and type conformance checks disabled, expression templates enabled). Please check, if the preprocessor symbol `NDEBUG` of `cassert` is defined. `NDEBUG` disables debug mode, which is needed to get size and type conformance checks.

Q: I've written some uBLAS benchmarks to measure the performance of matrix chain multiplications like `prod (A, prod (B, C))` and see a significant performance penalty due to the use of expression templates. How can I disable expression templates?
A: You do not need to disable expression templates. Please try reintroducing temporaries using either `prod (A, matrix_type (prod (B, C)))` or `prod (A, prod<``matrix_type > (B, C))`.

Overview

Rationale

It would be nice if every kind of numeric software could be written in C without loss of efficiency, but unless something can be found that achieves this without compromising the C type system it may be preferable to rely on Fortran, assembler or architecture-specific extensions (Bjarne Stroustrup).

This C++ library is directed towards scientific computing on the level of basic linear algebra constructions with matrices and vectors and their corresponding abstract operations. The primary design goals were:

- mathematical notation
- efficiency
- functionality
- compatibility

Another intention was to evaluate, if the abstraction penalty resulting from the use of such matrix and vector classes is acceptable.

Resources

The development of this library was guided by a couple of similar efforts:

- BLAS by Jack Dongarra et al.
- Blitz++ by Todd Veldhuizen
- POOMA by Scott Haney et al.

- MTL by Jeremy Siek et al.

BLAS seems to be the most widely used library for basic linear algebra constructions, so it could be called a de-facto standard. Its interface is procedural, the individual functions are somewhat abstracted from simple linear algebra operations. Due to the fact that is has been implemented using Fortran and its optimizations, it also seems to be one of the fastest libraries available. As we decided to design and implement our library in an object-oriented way, the technical approaches are distinct. However anyone should be able to express BLAS abstractions in terms of our library operators and to compare the efficiency of the implementations.

Blitz is an impressive library implemented in C. Its main design seems to be oriented towards multidimensional arrays and their associated operators including tensors. The author of Blitz states, that the library achieves performance on par or better than corresponding Fortran code due to his implementation technique using expression templates and template metaprograms. However we see some reasons, to develop an own design and implementation approach. We do not know whether anybody tries to implement traditional linear algebra and other numerical algorithms using Blitz. We also presume that even today Blitz needs the most advanced C compiler technology due to its implementation idioms. On the other hand, Blitz++ convinced us, that the use of expression templates is mandatory to reduce the abstraction penalty to an acceptable limit.

POOMA's design goals seem to parallel Blitz's in many parts . It extends Blitz's concepts with classes from the domains of partial differential equations and theoretical physics. The implementation supports even parallel architectures.

MTL is another approach supporting basic linear algebra operations in C. Its design mainly seems to be influenced by BLAS and the C Standard Template Library. We share the insight that a linear algebra library has to provide functionality comparable to BLAS. On the other hand we think, that the concepts of the C++ standard library have not yet been proven to support numerical computations as needed. As another difference MTL currently does not seem to use expression templates. This may result in one of two consequences: a possible loss of expressiveness or a possible loss of performance.

Concepts

Mathematical Notation

The usage of mathematical notation may ease the development of scientific algorithms. So a C library implementing basic linear algebra concepts carefully should overload selected C operators on matrix and vector classes.

We decided to use operator overloading for the following primitives:

Description	Operator
Indexing of vectors and matrices	`vector::operator(size_t i);` `matrix::operator(size_t i, size_t j);`

Description	Operator
Assignment of vectors and matrices	`vector::operator = (const vector_expression &); vector::operator += (const vector_expression &); vector::operator -= (const vector_expression &); vector::operator *= (const scalar_expression &); matrix::operator = (const matrix_expression &); matrix::operator += (const matrix_expression &); matrix::operator -= (const matrix_expression &); matrix::operator *= (const scalar_expression &);`
Unary operations on vectors and matrices	`vector_expression operator - (const vector_expression &); matrix_expression operator - (const matrix_expression &);`
Binary operations on vectors and matrices	`vector_expression operator + (const vector_expression &, const vector_expression &); vector_expression operator - (const vector_expression &, const vector_expression &); matrix_expression operator + (const matrix_expression &, const matrix_expression &); matrix_expression operator - (const matrix_expression &, const matrix_expression &);`
Multiplication of vectors and matrices with a scalar	`vector_expression operator * (const scalar_expression &, const vector_expression &); vector_expression operator * (const vector_expression &, const scalar_expression &); matrix_expression operator * (const scalar_expression &, const matrix_expression &); matrix_expression operator * (const matrix_expression &, const scalar_expression &);`

We decided to use no operator overloading for the following other primitives:

Description	Function
Left multiplication of vectors with a matrix	`vector_expression prod<vector_type > (const matrix_expression &, const vector_expression &); vector_expression prod (const matrix_expression &, const vector_expression &);`
Right multiplication of vectors with a matrix	`vector_expression prod<vector_type > (const vector_expression &, const matrix_expression &); vector_expression prod (const vector_expression &, const matrix_expression &);`
Multiplication of matrices	`matrix_expression prod<matrix_type > (const matrix_expression &, const matrix_expression &); matrix_expression prod (const matrix_expression &, const matrix_expression &);`

Description	Function
Inner product of vectors	`scalar_expression inner_prod (const vector_expression &, const vector_expression &);`
Outer product of vectors	`matrix_expression outer_prod (const vector_expression &, const vector_expression &);`
Transpose of a matrix	`matrix_expression trans (const matrix_expression &);`

Efficiency

To achieve the goal of efficiency for numerical computing, one has to overcome two difficulties in formulating abstractions with C++, namely temporaries and virtual function calls. Expression templates solve these problems, but tend to slow down compilation times.

Eliminating Temporaries

Abstract formulas on vectors and matrices normally compose a couple of unary and binary operations. The conventional way of evaluating such a formula is first to evaluate every leaf operation of a composition into a temporary and next to evaluate the composite resulting in another temporary. This method is expensive in terms of time especially for small and space especially for large vectors and matrices. The approach to solve this problem is to use lazy evaluation as known from modern functional programming languages. The principle of this approach is to evaluate a complex expression element wise and to assign it directly to the target.

Two interesting and dangerous facts result:

Aliases

One may get serious side effects using element wise evaluation on vectors or matrices. Consider the matrix vector product $x = A\ x$. Evaluation of $A_1 x$ and assignment to x_1 changes the right hand side, so that the evaluation of $A_2 x$ returns a wrong result. In this case there are **aliases** of the elements x_n on both the left and right hand side of the assignment.

Our solution for this problem is to evaluate the right hand side of an assignment into a temporary and then to assign this temporary to the left hand side. To allow further optimizations, we provide a corresponding member function for every assignment operator and also a `noalias` syntax. By using this syntax a programmer can confirm, that the left and right hand sides of an assignment are independent, so that element wise evaluation and direct assignment to the target is safe.

Complexity

The computational complexity may be unexpectedly large under certain cirumstances. Consider the chained matrix vector product $A\ (B\ x)$. Conventional evaluation of $A\ (B\ x)$ is quadratic. Deferred evaluation of $B\ x_i$ is linear. As every element $B\ x_i$ is needed linearly depending of the size, a completely deferred evaluation of the chained matrix vector product $A\ (B\ x)$ is cubic. In such cases one needs to reintroduce temporaries in the expression.

Eliminating Virtual Function Calls

Lazy expression evaluation normally leads to the definition of a class hierarchy of terms. This results in the usage of dynamic polymorphism to access single elements of vectors and matrices, which is also known to be expensive in terms of time. A solution was found a couple of years ago independently by David Vandervoorde and Todd Veldhuizen and is commonly called expression templates. Expression templates contain lazy evaluation and replace dynamic polymorphism with static, i.e. compile time polymorphism. Expression templates heavily depend on the famous Barton-Nackman trick, also coined 'curiously defined recursive templates' by Jim Coplien.

Expression templates form the base of our implementation.

Compilation times

It is also a well known fact, that expression templates challenge currently available compilers. We were able to significantly reduce the amount of needed expression templates using the Barton-Nackman trick consequently.

We also decided to support a dual conventional implementation (i.e. not using expression templates) with extensive bounds and type checking of vector and matrix operations to support the development cycle. Switching from debug mode to release mode is controlled by the `NDEBUG` preprocessor symbol of `<cassert>`.

Functionality

Every C++ library supporting linear algebra will be measured against the long-standing Fortran package BLAS. We now describe how BLAS calls may be mapped onto our classes.

The page Overview of Matrix and Vector Operations gives a short summary of the most used operations on vectors and matrices.

Blas Level 1

BLAS Call	Mapped Library Expression	Mathematical Description	Comment
`sasum` OR `dasum`	`norm_1 (x)`	$\texttt{sum(mod}(x_i))$	Computes the l_1 (sum) norm of a real vector.
`scasum` OR `dzasum`	`real (sum (v)) + imag (sum (v))`	$\texttt{sum re}(x_i) + \texttt{sum im}(x_i)$	Computes the sum of elements of a complex vector.
`_nrm2`	`norm_2 (x)`	$\texttt{sqrt sum(mod}(x_i)^2)$	Computes the l_2 (euclidean) norm of a vector.

BLAS Call	Mapped Library Expression	Mathematical Description	Comment
`i_amax`	`norm_inf (x)` `index_norm_inf (x)`	`max(mod(x_i))`	Computes the l_{inf} (maximum) norm of a vector. BLAS computes the index of the first element having this value.
`_dot _dotu _dotc`	`inner_prod (x, y)` or `inner_prod (conj (x), y)`	$x^T y$ or $x^H y$	Computes the inner product of two vectors. BLAS implements certain loop unrollment.
`dsdot sdsdot`	`a + prec_inner_prod (x, y)`	$a + x^T y$	Computes the inner product in double precision.
`_copy`	`x = y y.assign (x)`	$x \leftarrow y$	Copies one vector to another. BLAS implements certain loop unrollment.
`_swap`	`swap (x, y)`	$x \leftarrow \rightarrow y$	Swaps two vectors. BLAS implements certain loop unrollment.
`_scal csscal zdscal`	`x *= a`	$x \leftarrow a\,x$	Scales a vector. BLAS implements certain loop unrollment.
`_axpy`	`y += a * x`	$y \leftarrow a\,x + y$	Adds a scaled vector. BLAS implements certain loop unrollment.
`_rot _rotm csrot zdrot`	`t.assign (a * x + b * y), y.assign (- b * x + a * y), x.assign (t)`	$(x, y) \leftarrow (a\,x + b\,y, -b\,x + a\,y)$	Applies a plane rotation.
`_rotg _rotmg`		$(a, b) \leftarrow$ $(? a / sqrt (a^2 + b^2),$ $? b / sqrt (a^2 + b^2))$ or $(1, 0) \leftarrow (0, 0)$	Constructs a plane rotation.

Blas Level 2

BLAS Call	Mapped Library Expression	Mathematical Description	Comment
`_t_mv`	`x = prod (A, x)` or `x = prod (trans (A), x)` or `x = prod (herm (A), x)`	$x \leftarrow A\,x$ or + $x \leftarrow A^T\,x$ or + $x \leftarrow A^H\,x$	Computes the product of a matrix with a vector.
`_t_sv`	`y = solve (A, x, tag)` or `inplace_solve (A, x, tag)` or `y = solve (trans (A), x, tag)` or `inplace_solve (trans (A), x, tag)` or `y = solve (herm (A), x, tag)`\`or \`inplace_solve (herm (A), x, tag)	$y \leftarrow A^{-1}\,x$ or $x \leftarrow A^{-1}\,x$ or $y \leftarrow A^{T-1}\,x$ *or* x ← $A^{T-1}\,x$ or $y \leftarrow A^{H-1}\,x$ *or* x ← $A^{H-1}\,x$	Solves a system of linear equations with triangular form, i.e. A is triangular.
`_g_mv _s_mv _h_mv`	`y = a * prod (A, x) + b * y` or ` y = a * prod (trans (A), x) + b * y ` or ` y = a * prod (herm (A), x) + b * y `	$y \leftarrow a\,A\,x + b\,y$ or $y \leftarrow a\,A^T\,x + b\,y$ $y \leftarrow a\,A^H\,x + b\,y$	Adds the scaled product of a matrix with a vector.
`_g_r _g_ru _g_rc`	`A += a * outer_prod (x, y)` or ` A += a * outer_prod (x, conj (y)) `	$A \leftarrow a\,x\,y^T + A$ or $A \leftarrow a\,x\,y^H + A$	Performs a rank *1* update.
`_s_r _h_r`	`A += a * outer_prod (x, x)` or ` A += a * outer_prod (x, conj (x)) `	$A \leftarrow a\,x\,x^T + A$ or $A \leftarrow a\,x\,x^H + A$	Performs a symmetric or hermitian rank *1* update.
`_s_r2 _h_r2`	`A += a * outer_prod (x, y) + a * outer_prod (y, x))` or ` A += a * outer_prod (x, conj (y)) + conj (a) * outer_prod (y, conj (x))) `	$A \leftarrow a\,x\,y^T + a\,y\,x^T + A$ or $A \leftarrow a\,x\,y^H + \bar{a}\,y\,x^H + A$	Performs a symmetric or hermitian rank *2* update.

Blas Level 3

BLAS Call	Mapped Library Expression	Mathematical Description	Comment
`_t_mm`	`B = a * prod (A, B)` or `B = a * prod (trans (A), B)` or `B = a * prod (A, trans (B))` or `B = a * prod (trans (A), trans (B))` or `B = a * prod (herm (A), B)` or `B = a * prod (A, herm (B))` or `B = a * prod (herm (A), trans (B))` or `B = a * prod (trans (A), herm (B))` or `B = a * prod (herm (A), herm (B))`	$B \leftarrow a\ op\ (A)\ op\ (B)$ with $op\ (X) = X$ or $op\ (X) = X^T$ or $op\ (X) = X^H$	Computes the scaled product of two matrices.
`_t_sm`	`C = solve (A, B, tag)` or `inplace_solve (A, B, tag)` or `C = solve (trans (A), B, tag)` or `inplace_solve (trans (A), B, tag)` or `C = solve (herm (A), B, tag)` or `inplace_solve (herm (A), B, tag)`	$C \leftarrow A^{-1}\ B$ or $B \leftarrow A^{-1}\ B$ or $C \leftarrow A^{T-1}\ B\ or$ $B \leftarrow A^{-1}\ B$ or $C \leftarrow A^{H-1}\ B\ or$ $B \leftarrow A^{H-1}\ B$	Solves a system of linear equations with triangular form, i.e. A is triangular.

BLAS Call	Mapped Library Expression	Mathematical Description	Comment
`_g_mm _s_mm _h_mm`	`C = a * prod (A, B) + b * C` or `C = a * prod (trans (A), B) + b * C` or `C = a * prod (A, trans (B)) + b * C` or `C = a * prod (trans (A), trans (B)) + b * C` or `C = a * prod (herm (A), B) + b * C` or `C = a * prod (A, herm (B)) + b * C` or `C = a * prod (herm (A), trans (B)) + b * C` or `C = a * prod (trans (A), herm (B)) + b * C` or `C = a * prod (herm (A), herm (B)) + b * C`	$C \leftarrow a\ op\ (A)\ op\ (B) + b\ C$ with $op\ (X) = X$ or $op\ (X) = X^T$ or $op\ (X) = X^H$	Adds the scaled product of two matrices.
`_s_rk _h_rk`	`B = a * prod (A, trans (A)) + b * B` or `B = a * prod (trans (A), A) + b * B` or `B = a * prod (A, herm (A)) + b * B` or `B = a * prod (herm (A), A) + b * B`	$B \leftarrow a\ A\ A^T + b\ B$ or $B \leftarrow a\ A^T\ A + b\ B$ or $B \leftarrow a\ A\ A^H + b\ B$ or $B \leftarrow a\ A^H\ A + b\ B$	Performs a symmetric or hermitian rank k update.
`_s_r2k _h_r2k`	`C = a * prod (A, trans (B)) + a * prod (B, trans (A)) + b * C` or `C = a * prod (trans (A), B) + a * prod (trans (B), A) + b * C` or `C = a * prod (A, herm (B)) + conj (a) * prod (B, herm (A)) + b * C` or `C = a * prod (herm (A), B) + conj (a) * prod (herm (B), A) + b * C`	$C \leftarrow a\ A\ B^T + a\ B\ A^T + b\ C$ or $C \leftarrow a\ A^T\ B + a\ B^T\ A + b\ C$ or $C \leftarrow a\ A\ B^H + a^-\ B\ A^H + b\ C$ or $C \leftarrow a\ A^H\ B + a^-\ B^H\ A + b\ C$	Performs a symmetric or hermitian rank $2\ k$ update.

Storage Layout

uBLAS supports many different storage layouts. The full details can be found at the Overview of Types. Most types like `vector<double>` and `matrix<double>` are by default compatible to C arrays, but can also be configured to contain FORTAN compatible data.

Compatibility

For compatibility reasons we provide array like indexing for vectors and matrices. For some types (hermitian, sparse etc) this can be expensive for matrices due to the needed temporary proxy objects.

uBLAS uses STL compatible allocators for the allocation of the storage required for it's containers.

Benchmark Results

The following tables contain results of one of our benchmarks. This benchmark compares a native C implementation ('C array') and some library based implementations. The safe variants based on the library assume aliasing, the fast variants do not use temporaries and are functionally equivalent to the native C implementation. Besides the generic vector and matrix classes the benchmark utilizes special classes `c_vector` and `c_matrix`, which are intended to avoid every overhead through genericity.

The benchmark program **bench1** was compiled with GCC 4.0 and run on an Athlon 64 3000+. Times are scales for reasonable precision by running **bench1 100**.

First we comment the results for double vectors and matrices of dimension 3 and 3 x 3, respectively.

Comment				
inner_prod	C array	0.61	782	Some abstraction penalty
	c_vector		0.86	554
	vector<unbounded_array>		1.02	467
vector + vector	C array	0.51	1122	Abstraction penalty: factor 2
	c_vector fast		1.17	489
	vector<unbounded_array> fast		1.32	433
	c_vector safe		2.02	283
	vector<unbounded_array> safe		6.95	82
outer_prod	C array	0.59	872	Some abstraction penalty

Comment				
	c_matrix, c_vector fast		0.88	585
	matrix<unbounded_array>, vector<unbounded_array> fast		0.90	572
	c_matrix, c_vector safe		1.66	310
	matrix<unbounded_array>, vector<unbounded_array> safe		2.95	175
prod (matrix, vector)	C array	0.64	671	No significant abstraction penalty
	c_matrix, c_vector fast		0.70	613
	matrix<unbounded_array>, vector<unbounded_array> fast		0.79	543
	c_matrix, c_vector safe		0.95	452
	matrix<unbounded_array>, vector<unbounded_array> safe		2.61	164
matrix + matrix	C array	0.75	686	No significant abstraction penalty
	c_matrix fast		0.99	520
	matrix<unbounded_array> fast		1.29	399
	c_matrix safe		1.7	303
	matrix<unbounded_array> safe		3.14	164
prod (matrix, matrix)	C array	0.94	457	No significant abstraction penalty
	c_matrix fast		1.17	367

Comment				
	matrix<unbounded_array> fast		1.34	320
	c_matrix safe		1.56	275
	matrix<unbounded_array> safe		2.06	208

We notice a two fold performance loss for small vectors and matrices: first the general abstraction penalty for using classes, and then a small loss when using the generic vector and matrix classes. The difference w.r.t. alias assumptions is also significant.

Next we comment the results for double vectors and matrices of dimension 100 and 100 x 100, respectively.

Operation	Implementation	Elapsed [s]	MFLOP/s	Comment
inner_prod	C array	0.64	889	No significant abstraction penalty
	c_vector		0.66	862
	vector<unbounded_array>		0.66	862
vector + vector	C array	0.64	894	No significant abstraction penalty
	c_vector fast		0.66	867
	vector<unbounded_array> fast		0.66	867
	c_vector safe		1.14	501
	vector<unbounded_array> safe		1.23	465
outer_prod	C array	0.50	1144	No significant abstraction penalty
	c_matrix, c_vector fast		0.71	806
	matrix<unbounded_array>, vector<unbounded_array> fast		0.57	1004
	c_matrix, c_vector safe		1.91	300

Operation	Implementation	Elapsed [s]	MFLOP/s	Comment
	matrix<unbounded_array>, vector<unbounded_array> safe		0.89	643
prod (matrix, vector)	C array	0.65	876	No significant abstraction penalty
	c_matrix, c_vector fast		0.65	876
	matrix<unbounded_array>, vector<unbounded_array> fast		0.66	863
	c_matrix, c_vector safe		0.66	863
	matrix<unbounded_array>, vector<unbounded_array> safe		0.66	863
matrix + matrix	C array	0.96	596	No significant abstraction penalty
	c_matrix fast		1.21	473
	matrix<unbounded_array> fast		1.00	572
	c_matrix safe		2.44	235
	matrix<unbounded_array> safe		1.30	440
prod (matrix, matrix)	C array	0.70	813	No significant abstraction penalty
	c_matrix fast		0.73	780
	matrix<unbounded_array> fast		0.76	749
	c_matrix safe		0.75	759
	matrix<unbounded_array> safe		0.76	749

For larger vectors and matrices the general abstraction penalty for using classes seems to decrease, the small loss when using generic vector and matrix classes seems to remain. The difference w.r.t.

alias assumptions remains visible, too.

Vector

Vector Abstract

Description

The templated class `vector<T, A>` is the base container adaptor for dense vectors. For a n -dimensional vector and $0 <= i < n$ every element v_i is mapped to the i-th element of the container.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v (3);
    for (unsigned i = 0; i < v.size (); ++ i)
        v (i) = i;
    std::cout << v << std::endl;
}
```

Definition

Defined in the header vector.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the vector.	
`A`	The type of the Storage array. [1]	`unbounded_array<T>`

Model of

Vector, RandomAccessContainer

Type requirements

None, except for those imposed by the requirements of Vector and RandomAccessContainer.

Public base classes

`vector_container<vector<T, A> >`

Members

Member	Where defined	Description
`value_type`	VectorExpression	
`reference`	VectorExpression	
`const_reference`	VectorExpression	
`size_type`	VectorExpression	
`difference_type`	VectorExpression	
`const_iterator`	VectorExpression	
`iterator`	VectorExpression	
`const_reverse_iterator`	VectorExpression	
`reverse_iterator`	VectorExpression	
`array_type`	Vector	
`vector ()`	VectorExpression	Allocates an uninitialized `vector` that holds zero elements.
`vector (size_type size)`	Vector	Allocates an uninitialized `vector` that holds `size` elements.
`vector (const vector &v)`		The copy constructor.
`template<class AE> vector (const vector_expression<AE> &ae)`		The extended copy constructor.
`void resize (size_type size, bool preserve = true)`	Vector	Reallocates a `vector` to hold `size` elements. The existing elements of the `vector` are preseved when specified.
`size_type size () const`	VectorExpression	Returns the size of the `vector`.
`size_type max_size () const`	RandomAccessContainer	Returns the upper bound on the size of the `vector`.
`bool empty () const`	RandomAccessContainer	Equivilent to `size () == 0`.
`const array_type& data () const`	Vector	
`array_type& data ()`	Vector	
`const_reference operator () (size_type i) const`	VectorExpression	Returns a `const` reference of the `i` -th element.
`reference operator () (size_type i)`	VectorExpression	Returns a reference of the `i`-th element.
`const_reference operator [] (size_type i) const`	Vector	Returns a `const` reference of the `i` -th element.

Member	Where defined	Description
`reference operator [] (size_type i)`	Vector	Returns a reference of the `i`-th element.
`vector &operator = (const vector &v)`	VectorExpression	The assignment operator.
`vector &assign_temporary (vector &v)`	VectorExpression	Assigns a temporary. May change the vector `v`.
`template<class AE> vector &operator = (const vector_expression<AE> &ae)`	VectorExpression	The extended assignment operator.
`template<class AE> vector &assign (const vector_expression<AE> &ae)`	VectorExpression	Assigns a vector expression to the vector. Left and right hand side of the assignment should be independent.
`template<class AE> vector &operator += (const vector_expression<AE> &ae)`	VectorExpression	A computed assignment operator. Adds the vector expression to the vector.
`template<class AE> vector &plus_assign (const vector_expression<AE> &ae)`	VectorExpression	Adds a vector expression to the vector. Left and right hand side of the assignment should be independent.
`template<class AE> vector &operator -= (const vector_expression<AE> &ae)`	VectorExpression	A computed assignment operator. Subtracts the vector expression from the vector.
`template<class AE> vector &minus_assign (const vector_expression<AE> &ae)`	VectorExpression	Subtracts a vector expression from the vector. Left and right hand side of the assignment should be independent.
`template<class AT> vector &operator *= (const AT &at)`	VectorExpression	A computed assignment operator. Multiplies the vector with a scalar.
`template<class AT> vector &operator /= (const AT &at)`	VectorExpression	A computed assignment operator. Divides the vector through a scalar.
`void swap (vector &v)`	VectorExpression	Swaps the contents of the vectors.
`void insert_element (size_type i, const_reference t)`	Vector	Inserts the value `t` at the `i`-th element.
`void erase_element (size_type i)`	Vector	Erases the value at the `i`-th element.
`void clear ()`	Vector	Clears the vector.

Member	Where defined	Description
`const_iterator begin () const`	VectorExpression	Returns a `const_iterator` pointing to the beginning of the `vector`.
`const_iterator end () const`	VectorExpression	Returns a `const_iterator` pointing to the end of the `vector`.
`iterator begin ()`	VectorExpression	Returns a `iterator` pointing to the beginning of the `vector`.
`iterator end ()`	VectorExpression	Returns a `iterator` pointing to the end of the `vector`.
`const_reverse_iterator rbegin () const`	VectorExpression	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `vector`.
`const_reverse_iterator rend () const`	VectorExpression	Returns a `const_reverse_iterator` pointing to the end of the reversed `vector`.
`reverse_iterator rbegin ()`	VectorExpression	Returns a `reverse_iterator` pointing to the beginning of the reversed `vector`.
`reverse_iterator rend ()`	VectorExpression	Returns a `reverse_iterator` pointing to the end of the reversed `vector`.

Notes

[1] Common parameters for the Storage array are `unbounded_array<T>` , `bounded_array<T>` and `std::vector<T>` .

Unit Vector

Description

The templated class `unit_vector<T, ALLOC>` represents canonical unit vectors. For the *k*-th *n* -dimensional canonical unit vector and $0 \Leftarrow i < n$ holds $u^k_i = 0$, if $i <> k$, and $u^k_i = 1$.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    for (int i = 0; i < 3; ++ i) {
        unit_vector<double> v (3, i);
        std::cout << v << std::endl;
    }
}
```

Definition

Defined in the header vector.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the vector.	`int`
`ALLOC`	An STL Allocator for size_type and difference_type.	std::allocator

Model of

Vector .

Type requirements

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

Public base classes

`vector_container<unit_vector<T> >`

Members

Member	Description
`unit_vector ()`	Constructs an `unit_vector` that holds zero elements.
`unit_vector (size_type size, size_type index)`	Constructs the `index`-th `unit_vector` that holds `size` elements.
`unit_vector (const unit_vector &v)`	The copy constructor.
`void resize (size_type size, bool preserve = true)`	Resizes a `unit_vector` to hold `size` elements. Therefore the existing elements of the `unit_vector` are always preseved.

Member	Description
`size_type size () const`	Returns the size of the `unit_vector`.
`size_type index () const`	Returns the index of the `unit_vector`.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`const_reference operator [] (size_type i) const`	Returns the value of the `i`-th element.
`unit_vector &operator = (const unit_vector &v)`	The assignment operator.
`unit_vector &assign_temporary (unit_vector &v)`	Assigns a temporary. May change the unit vector `v` .
`void swap (unit_vector &v)`	Swaps the contents of the unit vectors.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `unit_vector`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `unit_vector`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `unit_vector`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `unit_vector`.

Zero Vector

Description

The templated class `zero_vector<T, ALLOC>` represents zero vectors. For a n-dimensional zero vector and $0 \leqslant i < n$ holds $z_i = 0$.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    zero_vector<double> v (3);
    std::cout << v << std::endl;
}
```

Definition

Defined in the header vector.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the vector.	`int`
`ALLOC`	An STL Allocator for size_type and difference_type.	std::allocator

Model of

Vector .

Type requirements

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

Public base classes

`vector_container<zero_vector<T> >`

Members

Member	Description
`zero_vector ()`	Constructs a `zero_vector` that holds zero elements.
`zero_vector (size_type size)`	Constructs a `zero_vector` that holds `size` elements.
`zero_vector (const zero_vector &v)`	The copy constructor.
`void resize (size_type size, bool preserve = true)`	Resizes a `zero_vector` to hold `size` elements. Therefore the existing elements of the `zero_vector` are always preseved.
`size_type size () const`	Returns the size of the `zero_vector`.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`const_reference operator [] (size_type i) const`	Returns the value of the `i`-th element.
`zero_vector &operator = (const zero_vector &v)`	The assignment operator.
`zero_vector &assign_temporary (zero_vector &v)`	Assigns a temporary. May change the zero vector `v` .
`void swap (zero_vector &v)`	Swaps the contents of the zero vectors.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `zero_vector`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `zero_vector`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `zero_vector`.

Member	Description
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `zero_vector`.

Scalar Vector

Description

The templated class `scalar_vector<T, ALLOC>` represents scalar vectors. For a n-dimensional scalar vector and $0 \Leftarrow i < n$ holds $z_i = s$.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    scalar_vector<double> v (3);
    std::cout << v << std::endl;
}
```

Definition

Defined in the header vector.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the vector.	`int`
`ALLOC`	An STL Allocator for size_type and difference_type.	std::allocator

Model of

Vector .

Type requirements

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

Public base classes

`vector_container<scalar_vector<T> >`

Members

Member	Description
`scalar_vector ()`	Constructs a `scalar_vector` that holds zero elements.
`scalar_vector (size_type size, const value_type &value)`	Constructs a `scalar_vector` that holds `size` elements each of the specified value.
`scalar_vector (const scalar_vector &v)`	The copy constructor.
`void resize (size_type size, bool preserve = true)`	Resizes a `scalar_vector` to hold `size` elements. Therefore the existing elements of the `scalar_vector` are always preseved.
`size_type size () const`	Returns the size of the `scalar_vector`.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`const_reference operator [] (size_type i) const`	Returns the value of the `i`-th element.
`scalar_vector &operator = (const scalar_vector &v)`	The assignment operator.
`scalar_vector &assign_temporary (scalar_vector &v)`	Assigns a temporary. May change the scalar vector `v` .
`void swap (scalar_vector &v)`	Swaps the contents of the scalar vectors.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `scalar_vector`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `scalar_vector`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `scalar_vector`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `scalar_vector`.

Sparse Vector

Mapped Vector

Description

The templated class `mapped_vector<T, A>` is the base container adaptor for sparse vectors using element maps. For a n-dimensional sparse vector and $0 <= i < n$ the non-zero elements v_i are mapped to consecutive elements of the associative container, i.e. for elements $k = v_{i1}$ and $k + 1 = \mathrm{v}_{i2}$ *of the container holds* $\mathrm{i}_1 < i_2$.

Example

```
#include <boost/numeric/ublas/vector_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    mapped_vector<double> v (3, 3);
    for (unsigned i = 0; i < v.size (); ++ i)
        v (i) = i;
    std::cout << v << std::endl;
}
```

Definition

Defined in the header vector_sparse.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the mapped vector.	
`A`	The type of the adapted array. [1]	`map_std<std::size_t, T>`

Model of

Vector .

Type requirements

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

Public base classes

`vector_container<mapped_vector<T, A> >`

Members

Member	Description
`mapped_vector ()`	Allocates a `mapped_vector` that holds zero elements.
`mapped_vector (size_type size, size_type non_zeros = 0)`	Allocates a `mapped_vector` that holds at most `size` elements.
`mapped_vector (const mapped_vector &v)`	The copy constructor.
`template<class AE> mapped_vector (size_type non_zeros, const vector_expression<AE> &ae)`	The extended copy constructor.

Member	Description
`void resize (size_type size, bool preserve = true)`	Reallocates a `mapped_vector` to hold at most `size` elements. The existing elements of the `mapped_vector` are preseved when specified.
`size_type size () const`	Returns the size of the `mapped_vector`.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`reference operator () (size_type i)`	Returns a reference of the `i`-th element.
`const_reference operator [] (size_type i) const`	Returns the value of the `i`-th element.
`reference operator [] (size_type i)`	Returns a reference of the `i`-th element.
`mapped_vector &operator = (const mapped_vector &v)`	The assignment operator.
`mapped_vector &assign_temporary (mapped_vector &v)`	Assigns a temporary. May change the mapped vector `v` .
`template<class AE> mapped_vector &operator = (const vector_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> mapped_vector &assign (const vector_expression<AE> &ae)`	Assigns a vector expression to the mapped vector. Left and right hand side of the assignment should be independent.
`template<class AE> mapped_vector &operator += (const vector_expression<AE> &ae)`	A computed assignment operator. Adds the vector expression to the mapped vector.
`template<class AE> mapped_vector &plus_assign (const vector_expression<AE> &ae)`	Adds a vector expression to the mapped vector. Left and right hand side of the assignment should be independent.
`template<class AE> mapped_vector &operator -= (const vector_expression<AE> &ae)`	A computed assignment operator. Subtracts the vector expression from the mapped vector.
`template<class AE> mapped_vector &minus_assign (const vector_expression<AE> &ae)`	Subtracts a vector expression from the mapped vector. Left and right hand side of the assignment should be independent.
`template<class AT> mapped_vector &operator *= (const AT &at)`	A computed assignment operator. Multiplies the mapped vector with a scalar.
`template<class AT> mapped_vector &operator /= (const AT &at)`	A computed assignment operator. Divides the mapped vector through a scalar.
`void swap (mapped_vector &v)`	Swaps the contents of the mapped vectors.
`true_reference insert_element (size_type i, const_reference t)`	Inserts the value `t` at the `i`-th element. Duplicates elements are not allowed.
`void erase_element (size_type i)`	Erases the value at the `i`-th element.
`void clear ()`	Clears the mapped vector.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `mapped_vector`.

Member	Description
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `mapped_vector`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `mapped_vector`.
`iterator end ()`	Returns a `iterator` pointing to the end of the `mapped_vector`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `mapped_vector`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `mapped_vector`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `mapped_vector`.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `mapped_vector`.

Notes

[1] Supported parameters for the adapted array are `map_array<std::size_t, T>` and `map_std<std::size_t, T>`. The latter is equivalent to `std::map<std::size_t, T>`.

Compressed Vector

Description

The templated class `compressed_vector<T, IB, IA, TA>` is the base container adaptor for compressed vectors. For a *n*-dimensional compressed vector and $0 \Leftarrow i < n$ the non-zero elements v_i are mapped to consecutive elements of the index and value container, i.e. for elements $k = v_{i1}$ and $k + 1 = v_{i2}$ *of these containers holds* $i_1 < i_2$.

Example

```
#include <boost/numeric/ublas/vector_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    compressed_vector<double> v (3, 3);
    for (unsigned i = 0; i < v.size (); ++ i)
        v (i) = i;
    std::cout << v << std::endl;
}
```

Definition

Defined in the header vector_sparse.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the compressed vector.	
`IB`	The index base of the compressed vector. [1]	`0`
`IA`	The type of the adapted array for indices. [2]	`unbounded_array<std::size_t>`
`TA`	The type of the adapted array for values. [2]	`unbounded_array<T>`

Model of

Vector .

Type requirements

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

Public base classes

`vector_container<compressed_vector<T, IB, IA, TA> >`

Members

Member	Description
`compressed_vector ()`	Allocates a `compressed_vector` that holds zero elements.
`compressed_vector (size_type size, size_type non_zeros)`	Allocates a `compressed_vector` that holds at most `size` elements.
`compressed_vector (const compressed_vector &v)`	The copy constructor.
`template<class AE> compressed_vector (size_type non_zeros, const vector_expression<AE> &ae)`	The extended copy constructor.
`void resize (size_type size, bool preserve = true)`	Reallocates a `compressed_vector` to hold at most `size` elements. The existing elements of the `compress_vector` are preseved when specified.
`size_type size () const`	Returns the size of the `compressed_vector`.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`reference operator () (size_type i)`	Returns a reference of the `i`-th element.
`const_reference operator [] (size_type i) const`	Returns the value of the `i`-th element.
`reference operator [] (size_type i)`	Returns a reference of the `i`-th element.

Member	Description
`compressed_vector &operator = (const compressed_vector &v)`	The assignment operator.
`compressed_vector &assign_temporary (compressed_vector &v)`	Assigns a temporary. May change the compressed vector `v`.
`template<class AE> compressed_vector &operator = (const vector_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> compressed_vector &assign (const vector_expression<AE> &ae)`	Assigns a vector expression to the compressed vector. Left and right hand side of the assignment should be independent.
`template<class AE> compressed_vector &operator += (const vector_expression<AE> &ae)`	A computed assignment operator. Adds the vector expression to the compressed vector.
`template<class AE> compressed_vector &plus_assign (const vector_expression<AE> &ae)`	Adds a vector expression to the compressed vector. Left and right hand side of the assignment should be independent.
`template<class AE> compressed_vector &operator -= (const vector_expression<AE> &ae)`	A computed assignment operator. Subtracts the vector expression from the compressed vector.
`template<class AE> compressed_vector &minus_assign (const vector_expression<AE> &ae)`	Subtracts a vector expression from the compressed vector. Left and right hand side of the assignment should be independent.
`template<class AT> compressed_vector &operator *= (const AT &at)`	A computed assignment operator. Multiplies the compressed vector with a scalar.
`template<class AT> compressed_vector &operator /= (const AT &at)`	A computed assignment operator. Divides the compressed vector through a scalar.
`void swap (compressed_vector &v)`	Swaps the contents of the compressed vectors.
`true_reference insert_element (size_type i, const_reference t)`	Inserts the value `t` at the `i`-th element. Duplicates elements are not allowed.
`void erase_element (size_type i)`	Erases the value at the `i`-th element.
`void clear ()`	Clears the compressed vector.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `compressed_vector`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `compressed_vector`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `compressed_vector`.
`iterator end ()`	Returns a `iterator` pointing to the end of the `compressed_vector`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `compressed_vector`.

Member	Description
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `compressed_vector`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `compressed_vector`.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `compressed_vector`.

Notes

[1] Supported parameters for the index base are `0` and `1` at least.

[2] Supported parameters for the adapted array are `unbounded_array<>` , `bounded_array<>` and `std::vector<>` .

Coordinate Vector

Description

The templated class `coordinate_vector<T, IB, IA, TA>` is the base container adaptor for compressed vectors. For a *n*-dimensional sorted coordinate vector and $0 \Leftarrow i < n$ the non-zero elements v_i are mapped to consecutive elements of the index and value container, i.e. for elements $k = v_{i1}$ and $k + 1 = v_{i2}$ *of these containers holds* $i_1 < i_2$.

Example

```
#include <boost/numeric/ublas/vector_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    coordinate_vector<double> v (3, 3);
    for (unsigned i = 0; i < v.size (); ++ i)
        v (i) = i;
    std::cout << v << std::endl;
}
```

Definition

Defined in the header vector_sparse.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the coordinate vector.	

Parameter	Description	Default
IB	The index base of the coordinate vector. [1]	0
IA	The type of the adapted array for indices. [2]	unbounded_array<std::size_t>
TA	The type of the adapted array for values. [2]	unbounded_array<T>

Model of

Vector .

Type requirements

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

Public base classes

vector_container<coordinate_vector<T, IB, IA, TA> >

Members

Member	Description
coordinate_vector ()	Allocates a coordinate_vector that holds zero elements.
coordinate_vector (size_type size, size_type non_zeros)	Allocates a coordinate_vector that holds at most size elements.
coordinate_vector (const coordinate_vector &v)	The copy constructor.
template<class AE> coordinate_vector (size_type non_zeros, const vector_expression<AE> &ae)	The extended copy constructor.
void resize (size_type size, bool preserve = true)	Reallocates a coordinate_vector to hold at most size elements. The existing elements of the coordinate_vector are preseved when specified.
size_type size () const	Returns the size of the coordinate_vector.
const_reference operator () (size_type i) const	Returns the value of the i-th element.
reference operator () (size_type i)	Returns a reference of the i-th element.
const_reference operator [] (size_type i) const	Returns the value of the i-th element.
reference operator [] (size_type i)	Returns a reference of the i-th element.
coordinate_vector &operator = (const coordinate_vector &v)	The assignment operator.
coordinate_vector &assign_temporary (coordinate_vector &v)	Assigns a temporary. May change the coordinate vector v.

Member	Description
`template<class AE> coordinate_vector &operator = (const vector_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> coordinate_vector &assign (const vector_expression<AE> &ae)`	Assigns a vector expression to the coordinate vector. Left and right hand side of the assignment should be independent.
`template<class AE> coordinate_vector &operator += (const vector_expression<AE> &ae)`	A computed assignment operator. Adds the vector expression to the coordinate vector.
`template<class AE> coordinate_vector &plus_assign (const vector_expression<AE> &ae)`	Adds a vector expression to the coordinate vector. Left and right hand side of the assignment should be independent.
`template<class AE> coordinate_vector &operator -= (const vector_expression<AE> &ae)`	A computed assignment operator. Subtracts the vector expression from the coordinate vector.
`template<class AE> coordinate_vector &minus_assign (const vector_expression<AE> &ae)`	Subtracts a vector expression from the coordinate vector. Left and right hand side of the assignment should be independent.
`template<class AT> coordinate_vector &operator *= (const AT &at)`	A computed assignment operator. Multiplies the coordinate vector with a scalar.
`template<class AT> coordinate_vector &operator /= (const AT &at)`	A computed assignment operator. Divides the coordinate vector through a scalar.
`void swap (coordinate_vector &v)`	Swaps the contents of the coordinate vectors.
`true_reference insert_element (size_type i, const_reference t)`	Inserts the value `t` at the `i`-th element. Duplicates elements are not allowed.
`void append_element (size_type i, size_type j, const_reference t)`	Appends the value `t` at the `i`-th element. Duplicate elements can be appended to a `coordinate_vector`. They are merged into a single arithmetically summed element by the `sort` function.
`void erase_element (size_type i)`	Erases the value at the `i`-th element.
`void clear ()`	Clears the coordinate vector.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `coordinate_vector`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `coordinate_vector`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `coordinate_vector`.
`iterator end ()`	Returns a `iterator` pointing to the end of the `coordinate_vector`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `coordinate_vector`.

Member	Description
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `coordinate_vector`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `coordinate_vector`.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `coordinate_vector`.

Notes

[1] Supported parameters for the index base are `0` and `1` at least.

[2] Supported parameters for the adapted array are `unbounded_array<>` , `bounded_array<>` and `std::vector<>` .

Vector Proxies

Vector Range

Description

The templated class `vector_range<V>` allows addressing a sub-range of a vector's element.

Example

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v (3);
    vector_range<vector<double> > vr (v, range (0, 3));
    for (unsigned i = 0; i < vr.size (); ++ i)
        vr (i) = i;
    std::cout << vr << std::endl;
}
```

Definition

Defined in the header vector_proxy.hpp.

Template parameters

Parameter	Description	Default
`V`	The type of vector referenced.	

Model of

Vector Expression .

If the specified range falls outside that of the index range of the vector, then the `vector_range` is not a well formed Vector Expression. That is, access to an element which is outside of index range of the vector is *undefined*.

Type requirements

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

Public base classes

`vector_expression<vector_range<V> >`

Members

Member	Description
`vector_range (vector_type &data, const range &r)`	Constructs a sub vector.
`size_type start () const`	Returns the start of the sub vector.
`size_type size () const`	Returns the size of the sub vector.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`reference operator () (size_type i)`	Returns a reference of the `i`-th element.
`const_reference operator [] (size_type i) const`	Returns the value of the `i`-th element.
`reference operator [] (size_type i)`	Returns a reference of the `i`-th element.
`vector_range &operator = (const vector_range &vr)`	The assignment operator.
`vector_range &assign_temporary (vector_range &vr)`	Assigns a temporary. May change the vector range `vr` .
`template<class AE> vector_range &operator = (const vector_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> vector_range &assign (const vector_expression<AE> &ae)`	Assigns a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> vector_range &operator += (const vector_expression<AE> &ae)`	A computed assignment operator. Adds the vector expression to the sub vector.
`template<class AE> vector_range &plus_assign (const vector_expression<AE> &ae)`	Adds a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> vector_range &operator -= (const vector_expression<AE> &ae)`	A computed assignment operator. Subtracts the vector expression from the sub vector.

Member	Description
`template<class AE> vector_range &minus_assign (const vector_expression<AE> &ae)`	Subtracts a vector expression from the sub vector. Left and right hand side of the assignment should be independent.
`template<class AT> vector_range &operator *= (const AT &at)`	A computed assignment operator. Multiplies the sub vector with a scalar.
`template<class AT> vector_range &operator /= (const AT &at)`	A computed assignment operator. Divides the sub vector through a scalar.
`void swap (vector_range &vr)`	Swaps the contents of the sub vectors.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `vector_range`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `vector_range`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `vector_range`.
`iterator end ()`	Returns a `iterator` pointing to the end of the `vector_range`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `vector_range`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `vector_range`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `vector_range`.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `vector_range`.

Simple Projections

Description

The free `subrange` functions support the construction of vector ranges.

Prototypes

```
template<class V>
vector_range<V> subrange (V &data,
   V::size_type start, V::size_type stop);
template<class V>
const vector_range<const V> subrange (const V &data,
   V::size_type start, V::size_type stop);
```

Generic Projections

Description

The free `project` functions support the construction of vector ranges. Existing `matrix_range` 's can be composed with a further range. The resulting range is computed using this existing range's `compose` function.

Prototypes

```
    template<class V>
    vector_range<V> project (V &data, const range &r);
    template<class V>
    const vector_range<const V> project (const V &data, const range &r);
    template<class V>
    vector_range<V> project (vector_range<V> &data, const range &r);
    template<class V>
    const vector_range<V> project (const vector_range<V> &data, const range &r);
```

Definition

Defined in the header vector_proxy.hpp.

Type requirements

- `V` is a model of Vector Expression .

Complexity

Linear depending from the size of the range.

Examples

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v (3);
    for (int i = 0; i < 3; ++ i)
        project (v, range (0, 3)) (i) = i;
    std::cout << project (v, range (0, 3)) << std::endl;
}
```

Vector Slice

Description

The templated class `vector_slice<V>` allows addressing a slice of a vector.

Example

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v (3);
    vector_slice<vector<double> > vs (v, slice (0, 1, 3));
    for (unsigned i = 0; i < vs.size (); ++ i)
        vs (i) = i;
    std::cout << vs << std::endl;
}
```

Definition

Defined in the header vector_proxy.hpp.

Template parameters

Parameter	Description	Default
`V`	The type of vector referenced.	

Model of

Vector Expression .

If the specified slice falls outside that of the index range of the vector, then the `vector_slice` is not a well formed Vector Expression. That is, access to an element which is outside of index range of the vector is *undefined*.

Type requirements

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

Public base classes

`vector_expression<vector_slice<V> >`

Members

Member	Description
`vector_slice (vector_type &data, const slice &s)`	Constructs a sub vector.
`size_type size () const`	Returns the size of the sub vector.

Member	Description
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`reference operator () (size_type i)`	Returns a reference of the `i`-th element.
`const_reference operator [] (size_type i) const`	Returns the value of the `i`-th element.
`reference operator [] (size_type i)`	Returns a reference of the `i`-th element.
`vector_slice &operator = (const vector_slice &vs)`	The assignment operator.
`vector_slice &assign_temporary (vector_slice &vs)`	Assigns a temporary. May change the vector slice `vs` .
`template<class AE> vector_slice &operator = (const vector_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> vector_slice &assign (const vector_expression<AE> &ae)`	Assigns a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> vector_slice &operator += (const vector_expression<AE> &ae)`	A computed assignment operator. Adds the vector expression to the sub vector.
`template<class AE> vector_slice &plus_assign (const vector_expression<AE> &ae)`	Adds a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> vector_slice &operator -= (const vector_expression<AE> &ae)`	A computed assignment operator. Subtracts the vector expression from the sub vector.
`template<class AE> vector_slice &minus_assign (const vector_expression<AE> &ae)`	Subtracts a vector expression from the sub vector. Left and right hand side of the assignment should be independent.
`template<class AT> vector_slice &operator *= (const AT &at)`	A computed assignment operator. Multiplies the sub vector with a scalar.
`template<class AT> vector_slice &operator /= (const AT &at)`	A computed assignment operator. Divides the sub vector through a scalar.
`void swap (vector_slice &vs)`	Swaps the contents of the sub vectors.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `vector_slice`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `vector_slice`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `vector_slice`.
`iterator end ()`	Returns a `iterator` pointing to the end of the `vector_slice`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `vector_slice`.

Member	Description
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `vector_slice`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `vector_slice`.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `vector_slice`.

Simple Projections

Description

The free `subslice` functions support the construction of vector slices.

Prototypes

```
template<class V>
vector_slice<V> subslice (V &data,
   V::size_type start, V::difference_type stride, V::size_type size);
template<class V>
const vector_slice<const V> subslice (const V &data,
   V::size_type start, V::difference_type stride, V::size_type size);
```

Generic Projections

Description

The free `project` functions support the construction of vector slices. Existing `vector_slice` 's can be composed with a further range or slices. The resulting slice is computed using this existing slices's `compose` function.

Prototypes

```
template<class V>
vector_slice<V> project (V &data, const slice &s);
template<class V>
const vector_slice<const V> project (const V &data, const slice &s);
template<class V>
vector_slice<V> project (vector_slice<V> &data, const range &r);
template<class V>
const vector_slice<V> project (const vector_slice<V> &data, const range &r);
template<class V>
vector_slice<V> project (vector_slice<V> &data, const slice &s);
template<class V>
const vector_slice<V> project (const vector_slice<V> &data, const slice &s);
```

Definition

Defined in the header vector_proxy.hpp.

Type requirements

- `V` is a model of Vector Expression .

Complexity

Linear depending from the size of the slice.

Examples

```
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/vector_proxy.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v (3);
    for (int i = 0; i < 3; ++ i)
        project (v, slice (0, 1, 3)) (i) = i;
    std::cout << project (v, slice (0, 1, 3)) << std::endl;
}
```

Vector Expressions

Description

The templated class `vector_expression<E>` is required to be a public base of all classes which model the Vector Expression concept.

Definition

Defined in the header expression_types.hpp.

Template parameters

Parameter	Description	Default
`E`	The type of the vector expression.	

Model of

None. Not a Vector Expression!

Type requirements

None.

Public base classes

None.

Members

Member	Description
`const expression_type &operator () () const`	Returns a `const` reference of the expression.
`expression_type &operator () ()`	Returns a reference of the expression.

Notes

The `range`, `slice` and `project` functions have been removed. Use the free functions defined in vector proxy instead.

Vector Container

Description

The templated class `vector_container<C>` is required to be a public base of all classes which model the Vector concept. This includes the class `vector` itself.

Definition

Defined in the header expression_types.hpp.

Template parameters

Parameter	Description	Default
`C`	The type of the vector container.	

Model of

None. Not a Vector Expression OR Vector!

Type requirements

None.

Public base classes

`vector_expression<C>`

Members

Member	Description
`const container_type &operator () () const`	Returns a `const` reference of the container.
`container_type &operator () ()`	Returns a reference of the container.

Vector References

Reference

Description

The templated class `vector_reference<E>` contains a reference to a vector expression.

Definition

Defined in the header vector_expression.hpp.

Template parameters

Parameter	Description	Default
`E`	The type of the vector expression.	

Model of

Vector Expression .

Type requirements

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

Public base classes

`vector_expression<vector_reference<E> >`

Members

Member	Description
`vector_reference (expression_type &e)`	Constructs a reference of the expression.
`void resize (size_type size)`	Resizes the expression to hold at most `size` elements.
`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.
`reference operator () (size_type i)`	Returns a reference 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.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the expression.

Member	Description
`iterator end ()`	Returns a `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.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed expression.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed expression.

Vector Operations

Unary Operation Description

Description

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

Definition

Defined in the header vector_expression.hpp.

Template parameters

Parameter	Description	Default
`E`	The type of the 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_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.

Unary Operations

Prototypes

```
template<class E, class F>
    struct vector_unary_traits {
        typedef vector_unary<typename E::const_closure_type, F> expression_type;
        typedef expression_type result_type;
     };

    // (- v) [i] = - v [i]
    template<class E>
     typename vector_unary_traits<E, scalar_negate<typename E::value_type>
>::result_type
    operator - (const vector_expression<E> &e);

    // (conj v) [i] = conj (v [i])
    template<class E>
     typename vector_unary_traits<E, scalar_conj<typename E::value_type>
>::result_type
    conj (const vector_expression<E> &e);

    // (real v) [i] = real (v [i])
    template<class E>
     typename vector_unary_traits<E, scalar_real<typename E::value_type>
>::result_type
    real (const vector_expression<E> &e);

    // (imag v) [i] = imag (v [i])
    template<class E>
     typename vector_unary_traits<E, scalar_imag<typename E::value_type>
>::result_type
    imag (const vector_expression<E> &e);

    // (trans v) [i] = v [i]
    template<class E>
     typename vector_unary_traits<E, scalar_identity<typename E::value_type>
>::result_type
    trans (const vector_expression<E> &e);

    // (herm v) [i] = conj (v [i])
    template<class E>
     typename vector_unary_traits<E, scalar_conj<typename E::value_type>
>::result_type
    herm (const vector_expression<E> &e);
```

Description

`operator -` computes the additive inverse of a vector expression. `conj` computes the complex conjugate of a vector expression. `real` and `imag` compute the real and imaginary parts of a vector expression. `trans` computes the transpose of a vector expression. `herm` computes the hermitian, i.e. the complex conjugate of the transpose of a vector expression.

Definition

Defined in the header vector_expression.hpp.

Type requirements

- `E` is a model of Vector Expression .

Preconditions

None.

Complexity

Linear depending from the size of the vector expression.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    vector<std::complex<double> > v (3);
    for (unsigned i = 0; i < v.size (); ++ i)
        v (i) = std::complex<double> (i, i);

    std::cout << - v << std::endl;
    std::cout << conj (v) << std::endl;
    std::cout << real (v) << std::endl;
    std::cout << imag (v) << std::endl;
    std::cout << trans (v) << std::endl;
    std::cout << herm (v) << std::endl;
}
```

Binary Operation Description

Description

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

Definition

Defined in the header vector_expression.hpp.

Template parameters

Parameter	Description	Default

`E1`	The type of the first vector expression.	
`E2`	The type of the 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.

Binary Operations

Prototypes

```
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;
     };

    // (v1 + v2) [i] = v1 [i] + v2 [i]
    template<class E1, class E2>
    typename vector_binary_traits<E1, E2, scalar_plus<typename E1::value_type,
                                                      typename E2::value_type>
>::result_type
    operator + (const vector_expression<E1> &e1,
                const vector_expression<E2> &e2);

    // (v1 - v2) [i] = v1 [i] - v2 [i]
    template<class E1, class E2>
    typename vector_binary_traits<E1, E2, scalar_minus<typename E1::value_type,
                                                       typename E2::value_type>
>::result_type
    operator - (const vector_expression<E1> &e1,
                const vector_expression<E2> &e2);
```

Description

`operator +` computes the sum of two vector expressions. `operator -` computes the difference of two vector expressions.

Definition

Defined in the header vector_expression.hpp.

Type requirements

- `E1` is a model of Vector Expression .
- `E2` is a model of Vector Expression .

Preconditions

- `e1 ().size () == e2 ().size ()`

Complexity

Linear depending from the size of the vector expressions.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v1 (3), v2 (3);
    for (unsigned i = 0; i < std::min (v1.size (), v2.size ()); ++ i)
        v1 (i) = v2 (i) = i;

    std::cout << v1 + v2 << std::endl;
    std::cout << v1 - v2 << std::endl;
}
```

Binary Outer Operation Description

Description

The templated class `vector_matrix_binary<E1, E2, F>` describes a binary outer vector operation.

Definition

Defined in the header matrix_expression.hpp.

Template parameters

Parameter	Description	Default
`E1`	The type of the first vector expression.	
`E2`	The type of the second vector 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<vector_matrix_binary<E1, E2, F> >`

Members

Member	Description
`vector_matrix_binary (const expression1_type &e1, const expression2_type &e2)`	Constructs a description of the expression.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the expression.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the expression.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the expression.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the expression.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed expression.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed expression.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed expression.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed expression.

Binary Outer Operations

Prototypes

```
template<class E1, class E2, class F>
    struct vector_matrix_binary_traits {
        typedef vector_matrix_binary<typename E1::const_closure_type,
                                     typename E2::const_closure_type, F>
expression_type;
        typedef expression_type result_type;
     };

    // (outer_prod (v1, v2)) [i] [j] = v1 [i] * v2 [j]
    template<class E1, class E2>
    typename vector_matrix_binary_traits<E1, E2, scalar_multiplies<typename E1
::value_type, typename E2::value_type> >::result_type
    outer_prod (const vector_expression<E1> &e1,
                const vector_expression<E2> &e2);
```

Description

`outer_prod` computes the outer product of two vector expressions.

Definition

Defined in the header matrix_expression.hpp.

Type requirements

- `E1` is a model of Vector Expression .
- `E2` is a model of Vector Expression .

Preconditions

None.

Complexity

Quadratic depending from the size of the vector expressions.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v1 (3), v2 (3);
    for (unsigned i = 0; i < std::min (v1.size (), v2.size ()); ++ i)
        v1 (i) = v2 (i) = i;

    std::cout << outer_prod (v1, v2) << std::endl;
}
```

Scalar Vector Operation Description

Description

The templated classes `vector_binary_scalar1<E1, E2, F>` and `vector_binary_scalar2<E1, E2, F>` describe binary operations between a scalar and a vector.

Definition

Defined in the header vector_expression.hpp.

Template parameters

Parameter	Description	Default

`E1/E2`	The type of the scalar expression.	
`E2/E1`	The type of the 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_scalar1<E1, E2, F> >` and `vector_expression<vector_binary_scalar2<E1, E2, F> >` resp.

Members

Member	Description
`vector_binary_scalar1 (const expression1_type &e1, const expression2_type &e2)`	Constructs a description of the expression.
`vector_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.

Scalar Vector Operations

Prototypes

```
template<class T1, class E2, class F>
    struct vector_binary_scalar1_traits {
        typedef vector_binary_scalar1<scalar_const_reference<T1>,
                                      typename E2::const_closure_type, F>
expression_type;
        typedef expression_type result_type;
    };

    // (t * v) [i] = t * v [i]
    template<class T1, class E2>
    typename vector_binary_scalar1_traits<T1, E2, scalar_multiplies<T1, typename E2
::value_type> >::result_type
    operator * (const T1 &e1,
                const vector_expression<E2> &e2);

    template<class E1, class T2, class F>
    struct vector_binary_scalar2_traits {
        typedef vector_binary_scalar2<typename E1::const_closure_type,
                                      scalar_const_reference<T2>, F> expression_type;
        typedef expression_type result_type;
    };

    // (v * t) [i] = v [i] * t
    template<class E1, class T2>
    typename vector_binary_scalar2_traits<E1, T2, scalar_multiplies<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>
    typename vector_binary_scalar2_traits<E1, T2, scalar_divides<typename E1
::value_type, T2> >::result_type
    operator / (const vector_expression<E1> &e1,
                const T2 &e2);
```

Description

`operator *` computes the product of a scalar and a vector expression. `operator /` multiplies the vector with the reciprocal of the scalar.

Definition

Defined in the header vector_expression.hpp.

Type requirements

- `T1/T2` is a model of Scalar Expression .
- `E2/E1` is a model of Vector Expression .

Preconditions

None.

Complexity

Linear depending from the size of the vector expression.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v (3);
    for (unsigned i = 0; i < v.size (); ++ i)
        v (i) = i;

    std::cout << 2.0 * v << std::endl;
    std::cout << v * 2.0 << std::endl;
}
```

Vector Reductions

Unary Reductions

Prototypes

```
template<class E, class F>
    struct vector_scalar_unary_traits {
         typedef typename F::result_type result_type;
    };

    // sum v = sum (v [i])
    template<class E>
    typename vector_scalar_unary_traits<E, vector_sum<typename E::value_type>
>::result_type
    sum (const vector_expression<E> &e);

    // norm_1 v = sum (abs (v [i]))
    template<class E>
    typename vector_scalar_unary_traits<E, vector_norm_1<typename E::value_type>
>::result_type
    norm_1 (const vector_expression<E> &e);

    // norm_2 v = sqrt (sum (v [i] * v [i]))
    template<class E>
    typename vector_scalar_unary_traits<E, vector_norm_2<typename E::value_type>
>::result_type
    norm_2 (const vector_expression<E> &e);

    // norm_2_square v = sum (v [i] * v [i])
    template<class E>
    typename vector_scalar_unary_traits<E, vector_norm_2_square<typename E::
value_type> >::result_type
    norm_2_square (const vector_expression<E> &e);

    // norm_inf v = max (abs (v [i]))
    template<class E>
    typename vector_scalar_unary_traits<E, vector_norm_inf<typename E::value_type>
>::result_type
    norm_inf (const vector_expression<E> &e);

    // index_norm_inf v = min (i: abs (v [i]) == max (abs (v [i])))
    template<class E>
    typename vector_scalar_unary_traits<E, vector_index_norm_inf<typename E
::value_type> >::result_type
    index_norm_inf (const vector_expression<E> &e);
```

Description

`sum` computes the sum of the vector expression's elements. `norm_1`, `norm_2` and `norm_inf` compute the corresponding $||.||_1$, $||.||_2$ and $||.||_{inf}$ vector norms. `index_norm_1` computes the index of the vector expression's first element having maximal absolute value.

Definition

Defined in the header vector_expression.hpp.

Type requirements

- `E` is a model of Vector Expression .

Preconditions

None.

Complexity

Linear depending from the size of the vector expression.

Examples

```
#include <boost/numeric/ublas/vector.hpp>

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v (3);
    for (unsigned i = 0; i < v.size (); ++ i)
        v (i) = i;

    std::cout << sum (v) << std::endl;
    std::cout << norm_1 (v) << std::endl;
    std::cout << norm_2 (v) << std::endl;
    std::cout << norm_inf (v) << std::endl;
    std::cout << index_norm_inf (v) << std::endl;
}
```

Binary Reductions

Prototypes

```
template<class E1, class E2, class F>
    struct vector_scalar_binary_traits {
        typedef typename F::result_type result_type;
    };

    // inner_prod (v1, v2) = sum (v1 [i] * v2 [i])
    template<class E1, class E2>
    typename vector_scalar_binary_traits<E1, E2, vector_inner_prod<typename E1
::value_type,
                                                                   typename E2
::value_type,
                                                                   typename
promote_traits<typename E1::value_type,

typename E2::value_type>::promote_type> >::result_type
    inner_prod (const vector_expression<E1> &e1,
                const vector_expression<E2> &e2);

    template<class E1, class E2>
    typename vector_scalar_binary_traits<E1, E2, vector_inner_prod<typename E1
::value_type,
                                                                   typename E2
::value_type,
                                                                   typename
type_traits<typename promote_traits<typename E1::value_type,

typename E2::value_type>::promote_type>::precision_type> >::result_type
    prec_inner_prod (const vector_expression<E1> &e1,
                     const vector_expression<E2> &e2);
```

Description

`inner_prod` computes the inner product of the vector expressions. `prec_inner_prod` computes the double precision inner product of the vector expressions`.`

Definition

Defined in the header vector_expression.hpp.

Type requirements

- `E1` is a model of Vector Expression .
- `E2` is a model of Vector Expression .

Preconditions

- `e1 ().size () == e2 ().size ()`

Complexity

Linear depending from the size of the vector expressions.

Examples

```
#include <boost/numeric/ublas/vector.hpp>

int main () {
    using namespace boost::numeric::ublas;
    vector<double> v1 (3), v2 (3);
    for (unsigned i = 0; i < std::min (v1.size (), v2.size ()); ++ i)
        v1 (i) = v2 (i) = i;

    std::cout << inner_prod (v1, v2) << std::endl;
}
```

Matrix

Matrix Abstract

Description

The templated class `matrix<T, F, A>` is the base container adaptor for dense matrices. For a *(m x n)*-dimensional matrix and $0 \leq i < m$, $0 \leq j < n$ every element $m_{i,j}$ is mapped to the *(i x n j)*-th element of the container for row major orientation or the *(i+ j x m)*-th element of the container for column major orientation.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

Definition

Defined in the header matrix.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the matrix.	
`F`	Functor describing the storage organization. [1]	`row_major`
`A`	The type of the Storage array. [2]	`unbounded_array<T>`

Model of

Matrix .

Type requirements

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

Public base classes

`matrix_container<matrix<T, F, A> >`

Members

Member	Description
`matrix ()`	Allocates an uninitialized `matrix` that holds zero rows of zero elements.
`matrix (size_type size1, size_type size2)`	Allocates an uninitialized `matrix` that holds `size1` rows of `size2` elements.
`matrix (const matrix &m)`	The copy constructor.
`template<class AE> matrix (const matrix_expression<AE> &ae)`	The extended copy constructor.
`void resize (size_type size1, size_type size2, bool preserve = true)`	Reallocates a `matrix` to hold `size1` rows of `size2` elements. The existing elements of the `matrix` are preseved when specified.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const array_type& data () const`	
`array_type& data ()`	
`const_reference operator () (size_type i, size_type j) const`	Returns a `const` reference of the `j` -th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`matrix &operator = (const matrix &m)`	The assignment operator.

Member	Description
`matrix &assign_temporary (matrix &m)`	Assigns a temporary. May change the matrix `m`.
`template<class AE> matrix &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> matrix &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the matrix. Left and right hand side of the assignment should be independent.
`template<class AE> matrix &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the matrix.
`template<class AE> matrix &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the matrix. Left and right hand side of the assignment should be independent.
`template<class AE> matrix &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the matrix.
`template<class AE> matrix &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the matrix. Left and right hand side of the assignment should be independent.
`template<class AT> matrix &operator *= (const AT &at)`	A computed assignment operator. Multiplies the matrix with a scalar.
`template<class AT> matrix &operator /= (const AT &at)`	A computed assignment operator. Divides the matrix through a scalar.
`void swap (matrix &m)`	Swaps the contents of the matrices.
`void insert_element (size_type i, size_type j, const_reference t)`	Inserts the value `t` at the `j`-th element of the `i`-th row.
`void erase_element (size_type i, size_type j)`	Erases the value at the `j`-th element of the `i`-th row.
`void clear ()`	Clears the matrix.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `matrix`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `matrix`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `matrix`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `matrix`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `matrix`.

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

Notes

[1] Supported parameters for the storage organization are `row_major` and `column_major`.

[2] Common parameters for the storage array are `unbounded_array<T>` , `bounded_array<T>` and `std::vector<T>` .

Identity Matrix

Description

The templated class `identity_matrix<T, ALLOC>` represents identity matrices. For a *(m x n)*-dimensional identity matrix and $0 \Leftarrow i < m, 0 \Leftarrow j < n$ holds $id_{i,j} = 0$, if $i <> j$, and $id_{i,i}$.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    identity_matrix<double> m (3);
    std::cout << m << std::endl;
}
```

Definition

Defined in the header matrix.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the matrix.	`int`
`ALLOC`	An STL Allocator for size_type and difference_type.	std::allocator

Model of

Matrix .

Type requirements

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

Public base classes

`matrix_container<identity_matrix<T> >`

Members

Member	Description
`identity_matrix ()`	Constructs an `identity_matrix` that holds zero rows of zero elements.
`identity_matrix (size_type size)`	Constructs an `identity_matrix` that holds `size` rows of `size` elements.
`identity_matrix (const identity_matrix &m)`	The copy constructor.
`void resize (size_type size, bool preserve = true)`	Resizes a `identity_matrix` to hold `size` rows of `size` elements. Therefore the existing elements of the `itendity_matrix` are always preseved.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.

Member	Description
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`identity_matrix &operator = (const identity_matrix &m)`	The assignment operator.
`identity_matrix &assign_temporary (identity_matrix &m)`	Assigns a temporary. May change the identity matrix `m`.
`void swap (identity_matrix &m)`	Swaps the contents of the identity matrices.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `identity_matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `identity_matrix`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `identity_matrix`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `identity_matrix`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `identity_matrix`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `identity_matrix`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `identity_matrix`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `identity_matrix`.

Zero Matrix

Description

The templated class `zero_matrix<T, ALLOC>` represents zero matrices. For a *(m x n)*-dimensional zero matrix and $0 \Leftarrow i < m$, $0 \Leftarrow j < n$ holds $z_{i,j} = 0$.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    zero_matrix<double> m (3, 3);
    std::cout << m << std::endl;
}
```

Definition

Defined in the header matrix.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the matrix.	`int`
`ALLOC`	An STL Allocator for size_type and difference_type.	std::allocator

Model of

Matrix .

Type requirements

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

Public base classes

`matrix_container<zero_matrix<T> >`

Members

Member	Description
`zero_matrix ()`	Constructs a `zero_matrix` that holds zero rows of zero elements.
`zero_matrix (size_type size1, size_type size2)`	Constructs a `zero_matrix` that holds `size1` rows of `size2` elements.
`zero_matrix (const zero_matrix &m)`	The copy constructor.
`void resize (size_type size1, size_type size2, bool preserve = true)`	Resizes a `zero_matrix` to hold `size1` rows of `size2` elements. Therefore the existing elements of the `zero_matrix` are always preseved.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`zero_matrix &operator = (const zero_matrix &m)`	The assignment operator.
`zero_matrix &assign_temporary (zero_matrix &m)`	Assigns a temporary. May change the zero matrix `m` .
`void swap (zero_matrix &m)`	Swaps the contents of the zero matrices.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `zero_matrix`.

Member	Description
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `zero_matrix`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `zero_matrix`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `zero_matrix`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `zero_matrix`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `zero_matrix`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `zero_matrix`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `zero_matrix`.

Scalar Matrix

Description

The templated class `scalar_matrix<T, ALLOC>` represents scalar matrices. For a *(m x n)*-dimensional scalar matrix and $0 \Leftarrow i < m, 0 \Leftarrow j < n$ holds $z_{i,j}$ = s.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    scalar_matrix<double> m (3, 3);
    std::cout << m << std::endl;
}
```

Definition

Defined in the header matrix.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the matrix.	`int`

Parameter	Description	Default
ALLOC	An STL Allocator for size_type and difference_type.	std::allocator

Model of

Matrix .

Type requirements

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

Public base classes

`matrix_container<scalar_matrix<T> >`

Members

Member	Description
`scalar_matrix ()`	Constructs a `scalar_matrix` that holds scalar rows of zero elements.
`scalar_matrix (size_type size1, size_type size2, const value_type &value)`	Constructs a `scalar_matrix` that holds `size1` rows of `size2` elements each of the specified value.
`scalar_matrix (const scalar_matrix &m)`	The copy constructor.
`void resize (size_type size1, size_type size2, bool preserve = true)`	Resizes a `scalar_matrix` to hold `size1` rows of `size2` elements. Therefore the existing elements of the `scalar_matrix` are always preseved.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`scalar_matrix &operator = (const scalar_matrix &m)`	The assignment operator.
`scalar_matrix &assign_temporary (scalar_matrix &m)`	Assigns a temporary. May change the scalar matrix `m` .
`void swap (scalar_matrix &m)`	Swaps the contents of the scalar matrices.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `scalar_matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `scalar_matrix`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `scalar_matrix`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `scalar_matrix`.

Member	Description
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `scalar_matrix`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `scalar_matrix`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `scalar_matrix`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `scalar_matrix`.

Triangular Matrix

Description

The templated class `triangular_matrix<T, F1, F2, A>` is the base container adaptor for triangular matrices. For a *(n x n)*-dimensional lower triangular matrix and $0 <= i < n$, $0 <= j < n$ holds $t_{i,j} = 0$, if $i > j$. If furthermore holds $t_{i,i} = 1$ the matrix is called unit lower triangular. For a *(n x n)*-dimensional lower triangular matrix and $0 <= i < n$, $0 <= j < n$ holds $t_{i,j} = 0$, if $i < j$. If furthermore holds $t_{i,i} = 1$ the matrix is called unit lower triangular. The storage of triangular matrices is packed.

Example

```
#include <boost/numeric/ublas/triangular.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    triangular_matrix<double, lower> ml (3, 3);
    for (unsigned i = 0; i < ml.size1 (); ++ i)
        for (unsigned j = 0; j <= i; ++ j)
            ml (i, j) = 3 * i + j;
    std::cout << ml << std::endl;
    triangular_matrix<double, upper> mu (3, 3);
    for (unsigned i = 0; i < mu.size1 (); ++ i)
        for (unsigned j = i; j < mu.size2 (); ++ j)
            mu (i, j) = 3 * i + j;
    std::cout << mu << std::endl;
}
```

Please read the full triangular example for more details.

Definition

Defined in the header triangular.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the matrix.	
`F1`	Functor describing the type of the triangular matrix. [1]	`lower`
`F2`	Functor describing the storage organization. [2]	`row_major`
`A`	The type of the adapted array. [3]	`unbounded_array<T>`

Model of

Matrix .

Type requirements

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

Public base classes

`matrix_container<triangular_matrix<T, F1, F2, A> >`

Members

Member	Description
`triangular_matrix ()`	Allocates an uninitialized `triangular_matrix` that holds zero rows of zero elements.
`triangular_matrix (size_type size1, size_type size2)`	Allocates an uninitialized `triangular_matrix` that holds `size1` rows of `size2` elements.
`triangular_matrix (const triangular_matrix &m)`	The copy constructor.
`template<class AE> triangular_matrix (const matrix_expression<AE> &ae)`	The extended copy constructor.
`void resize (size_type size1, size_type size2, bool preserve = true)`	Reallocates a `triangular_matrix` to hold `size1` rows of `size2` elements. The existing elements of the `triangular_matrix` are preseved when specified.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns a `const` reference of the `j` -th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.

Member	Description
`triangular_matrix &operator = (const triangular_matrix &m)`	The assignment operator.
`triangular_matrix &assign_temporary (triangular_matrix &m)`	Assigns a temporary. May change the triangular matrix `m`.
`template<class AE> triangular_matrix &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> triangular_matrix &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the triangular matrix. Left and right hand side of the assignment should be independent.
`template<class AE> triangular_matrix &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the triangular matrix.
`template<class AE> triangular_matrix &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the triangular matrix. Left and right hand side of the assignment should be independent.
`template<class AE> triangular_matrix &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the triangular matrix.
`template<class AE> triangular_matrix &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the triangular matrix. Left and right hand side of the assignment should be independent.
`template<class AT> triangular_matrix &operator *= (const AT &at)`	A computed assignment operator. Multiplies the triangular matrix with a scalar.
`template<class AT> triangular_matrix &operator /= (const AT &at)`	A computed assignment operator. Divides the triangular matrix through a scalar.
`void swap (triangular_matrix &m)`	Swaps the contents of the triangular matrices.
`void insert (size_type i, size_type j, const_reference t)`	Inserts the value `t` at the `j`-th element of the `i`-th row.
`void erase (size_type i, size_type j)`	Erases the value at the `j`-th elemenst of the `i`-th row.
`void clear ()`	Clears the matrix.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `triangular_matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `triangular_matrix`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `triangular_matrix`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `triangular_matrix`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `triangular_matrix`.

Member	Description
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `triangular_matrix`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `triangular_matrix`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `triangular_matrix`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `triangular_matrix`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `triangular_matrix`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `triangular_matrix`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `triangular_matrix`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `triangular_matrix`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `triangular_matrix`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `triangular_matrix`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `triangular_matrix`.

Notes

[1] Supported parameters for the type of the triangular matrix are `lower` , `unit_lower`, `upper` and `unit_upper` .

[2] Supported parameters for the storage organization are `row_major` and `column_major`.

[3] Supported parameters for the adapted array are `unbounded_array<T>` , `bounded_array<T>` and `std::vector<T>` .

Triangular Adaptor

Description

The templated class `triangular_adaptor<M, F>` is a triangular matrix adaptor for other matrices.

Example

```
#include <boost/numeric/ublas/triangular.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    triangular_adaptor<matrix<double>, lower> tal (m);
    for (unsigned i = 0; i < tal.size1 (); ++ i)
        for (unsigned j = 0; j <= i; ++ j)
            tal (i, j) = 3 * i + j;
    std::cout << tal << std::endl;
    triangular_adaptor<matrix<double>, upper> tau (m);
    for (unsigned i = 0; i < tau.size1 (); ++ i)
        for (unsigned j = i; j < tau.size2 (); ++ j)
            tau (i, j) = 3 * i + j;
    std::cout << tau << std::endl;
}
```

Please read the full triangular example for more details.

Definition

Defined in the header triangular.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of the adapted matrix.	
`F`	Functor describing the type of the triangular adaptor. [1]	`lower`

Model of

Matrix Expression .

Type requirements

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

Public base classes

`matrix_expression<triangular_adaptor<M, F> >`

Members

Member	Description
`triangular_adaptor (matrix_type &data)`	Constructs a `triangular_adaptor` of a matrix.
`triangular_adaptor (const triangular_adaptor &m)`	The copy constructor.

Member	Description
`template<class AE> triangular_adaptor (const matrix_expression<AE> &ae)`	The extended copy constructor.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns a `const` reference of the `j` -th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`triangular_adaptor &operator = (const triangular_adaptor &m)`	The assignment operator.
`triangular_adaptor &assign_temporary (triangular_adaptor &m)`	Assigns a temporary. May change the triangular adaptor `m`.
`template<class AE> triangular_adaptor &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> triangular_adaptor &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the triangular adaptor. Left and right hand side of the assignment should be independent.
`template<class AE> triangular_adaptor &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the triangular adaptor.
`template<class AE> triangular_adaptor &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the triangular adaptor. Left and right hand side of the assignment should be independent.
`template<class AE> triangular_adaptor &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the triangular adaptor.
`template<class AE> triangular_adaptor &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the triangular adaptor. Left and right hand side of the assignment should be independent.
`template<class AT> triangular_adaptor &operator *= (const AT &at)`	A computed assignment operator. Multiplies the triangular adaptor with a scalar.
`template<class AT> triangular_adaptor &operator /= (const AT &at)`	A computed assignment operator. Divides the triangular adaptor through a scalar.
`void swap (triangular_adaptor &m)`	Swaps the contents of the triangular adaptors.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `triangular_adaptor`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `triangular_adaptor`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `triangular_adaptor`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `triangular_adaptor`.

Member	Description
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `triangular_adaptor`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `triangular_adaptor`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `triangular_adaptor`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `triangular_adaptor`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `triangular_adaptor`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `triangular_adaptor`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `triangular_adaptor`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `triangular_adaptor`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `triangular_adaptor`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `triangular_adaptor`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `triangular_adaptor`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `triangular_adaptor`.

Notes

[1] Supported parameters for the type of the triangular adaptor are `lower` , `unit_lower`, `upper` and `unit_upper` .

Symmetric Matrix

Description

The templated class `symmetric_matrix<T, F1, F2, A>` is the base container adaptor for symmetric matrices. For a *(n x n*)-dimensional symmetric matrix and *0* < = *i* < *n*, *0* < = *j* < *n* holds $s_{i,j} = s_{j,i}$. The storage of symmetric matrices is packed.

Example

```
#include <boost/numeric/ublas/symmetric.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    symmetric_matrix<double, lower> ml (3, 3);
    for (unsigned i = 0; i < ml.size1 (); ++ i)
        for (unsigned j = 0; j <= i; ++ j)
            ml (i, j) = 3 * i + j;
    std::cout << ml << std::endl;
    symmetric_matrix<double, upper> mu (3, 3);
    for (unsigned i = 0; i < mu.size1 (); ++ i)
        for (unsigned j = i; j < mu.size2 (); ++ j)
            mu (i, j) = 3 * i + j;
    std::cout << mu << std::endl;
}
```

Definition

Defined in the header symmetric.hpp.

Template parameters

Parameter	Description	Default
T	The type of object stored in the matrix.	
F1	Functor describing the type of the symmetric matrix. [1]	lower
F2	Functor describing the storage organization. [2]	row_major
A	The type of the adapted array. [3]	unbounded_array<T>

Model of

Matrix .

Type requirements

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

Public base classes

matrix_container<symmetric_matrix<T, F1, F2, A> >

Members

Member	Description
`symmetric_matrix (size_type size)`	Allocates an uninitialized `symmetric_matrix` that holds `size` rows of `size` elements.
`symmetric_matrix (const symmetric_matrix &m)`	The copy constructor.
`template<class AE> symmetric_matrix (const matrix_expression<AE> &ae)`	The extended copy constructor.
`void resize (size_type size, bool preserve = true)`	Reallocates a `symmetric_matrix` to hold `size` rows of `size` elements. The existing elements of the `symmetric_matrix` are preseved when specified.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns a `const` reference of the `j` -th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`symmetric_matrix &operator = (const symmetric_matrix &m)`	The assignment operator.
`symmetric_matrix &assign_temporary (symmetric_matrix &m)`	Assigns a temporary. May change the symmetric matrix `m` .
`template<class AE> symmetric_matrix &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> symmetric_matrix &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the symmetric matrix. Left and right hand side of the assignment should be independent.
`template<class AE> symmetric_matrix &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the symmetric matrix.
`template<class AE> symmetric_matrix &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the symmetric matrix. Left and right hand side of the assignment should be independent.
`template<class AE> symmetric_matrix &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the symmetric matrix.
`template<class AE> symmetric_matrix &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the symmetric matrix. Left and right hand side of the assignment should be independent.
`template<class AT> symmetric_matrix &operator *= (const AT &at)`	A computed assignment operator. Multiplies the symmetric matrix with a scalar.
`template<class AT> symmetric_matrix &operator /= (const AT &at)`	A computed assignment operator. Divides the symmetric matrix through a scalar.
`void swap (symmetric_matrix &m)`	Swaps the contents of the symmetric matrices.
`void insert (size_type i, size_type j, const_reference t)`	Inserts the value `t` at the `j`-th element of the `i`-th row.

Member	Description
`void erase (size_type i, size_type j)`	Erases the value at the `j`-th elemenst of the `i`-th row.
`void clear ()`	Clears the matrix.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `symmetric_matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `symmetric_matrix`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `symmetric_matrix`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `symmetric_matrix`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `symmetric_matrix`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `symmetric_matrix`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `symmetric_matrix`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `symmetric_matrix`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `symmetric_matrix`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `symmetric_matrix`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `symmetric_matrix`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `symmetric_matrix`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `symmetric_matrix`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `symmetric_matrix`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `symmetric_matrix`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `symmetric_matrix`.

Notes

[1] Supported parameters for the type of the symmetric matrix are `lower` and `upper`.

[2] Supported parameters for the storage organization are `row_major` and `column_major`.

[3] Supported parameters for the adapted array are `unbounded_array<T>` , `bounded_array<T>` and `std::vector<T>` .

Symmetric Adaptor

Description

The templated class `symmetric_adaptor<M, F>` is a symmetric matrix adaptor for other matrices.

Example

```
#include <boost/numeric/ublas/symmetric.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    symmetric_adaptor<matrix<double>, lower> sal (m);
    for (unsigned i = 0; i < sal.size1 (); ++ i)
        for (unsigned j = 0; j <= i; ++ j)
            sal (i, j) = 3 * i + j;
    std::cout << sal << std::endl;
    symmetric_adaptor<matrix<double>, upper> sau (m);
    for (unsigned i = 0; i < sau.size1 (); ++ i)
        for (unsigned j = i; j < sau.size2 (); ++ j)
            sau (i, j) = 3 * i + j;
    std::cout << sau << std::endl;
}
```

Definition

Defined in the header symmetric.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of the adapted matrix.	
`F`	Functor describing the type of the symmetric adaptor. [1]	`lower`

Model of

Matrix Expression .

Type requirements

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

Public base classes

`matrix_expression<symmetric_adaptor<M, F> >`

Members

Member	Description
`symmetric_adaptor ()`	Constructs a `symmetric_adaptor` that holds zero rows of zero elements.
`symmetric_adaptor (matrix_type &data)`	Constructs a `symmetric_adaptor` of a matrix.
`symmetric_adaptor (const symmetric_adaptor &m)`	The copy constructor.
`template<class AE> symmetric_adaptor (const matrix_expression<AE> &ae)`	The extended copy constructor.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns a `const` reference of the `j` -th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`symmetric_adaptor &operator = (const symmetric_adaptor &m)`	The assignment operator.
`symmetric_adaptor &assign_temporary (symmetric_adaptor &m)`	Assigns a temporary. May change the symmetric adaptor `m`.
`template<class AE> symmetric_adaptor &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> symmetric_adaptor &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the symmetric adaptor. Left and right hand side of the assignment should be independent.
`template<class AE> symmetric_adaptor &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the symmetric adaptor.
`template<class AE> symmetric_adaptor &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the symmetric adaptor. Left and right hand side of the assignment should be independent.
`template<class AE> symmetric_adaptor &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the symmetric adaptor.
`template<class AE> symmetric_adaptor &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the symmetric adaptor. Left and right hand side of the assignment should be independent.
`template<class AT> symmetric_adaptor &operator *= (const AT &at)`	A computed assignment operator. Multiplies the symmetric adaptor with a scalar.
`template<class AT> symmetric_adaptor &operator /= (const AT &at)`	A computed assignment operator. Divides the symmetric adaptor through a scalar.

Member	Description
`void swap (symmetric_adaptor &m)`	Swaps the contents of the symmetric adaptors.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `symmetric_adaptor`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `symmetric_adaptor`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `symmetric_adaptor`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `symmetric_adaptor`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `symmetric_adaptor`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `symmetric_adaptor`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `symmetric_adaptor`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `symmetric_adaptor`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `symmetric_adaptor`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `symmetric_adaptor`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `symmetric_adaptor`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `symmetric_adaptor`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `symmetric_adaptor`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `symmetric_adaptor`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `symmetric_adaptor`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `symmetric_adaptor`.

Notes

[1] Supported parameters for the type of the symmetric adaptor are `lower` and `upper`.

Hermitian Matrix

Description

The templated class `hermitian_matrix<T, F1, F2, A>` is the base container adaptor for hermitian matrices. For a *(n x n)*-dimensional hermitian matrix and $0 <= i < n$, $0 <= j < n$ holds $h_{i,j} = \bar{h_{j,i}}$. The storage of hermitian matrices is packed.

Example

```
#include <boost/numeric/ublas/hermitian.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    hermitian_matrix<std::complex<double>, lower> ml (3, 3);
    for (unsigned i = 0; i < ml.size1 (); ++ i) {
        for (unsigned j = 0; j < i; ++ j)
            ml (i, j) = std::complex<double> (3 * i + j, 3 * i + j);
        ml (i, i) = std::complex<double> (4 * i, 0);
    }
    std::cout << ml << std::endl;
    hermitian_matrix<std::complex<double>, upper> mu (3, 3);
    for (unsigned i = 0; i < mu.size1 (); ++ i) {
        mu (i, i) = std::complex<double> (4 * i, 0);
        for (unsigned j = i + 1; j < mu.size2 (); ++ j)
            mu (i, j) = std::complex<double> (3 * i + j, 3 * i + j);
    }
    std::cout << mu << std::endl;
}
```

Definition

Defined in the header hermitian.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the matrix.	
`F1`	Functor describing the type of the hermitian matrix. [1]	`lower`
`F2`	Functor describing the storage organization. [2]	`row_major`
`A`	The type of the adapted array. [3]	`unbounded_array<T>`

Model of

Matrix .

Type requirements

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

Public base classes

`matrix_container<hermitian_matrix<T, F1, F2, A> >`

Members

Member	Description
`hermitian_matrix ()`	Allocates an uninitialized `hermitian_matrix` that holds zero rows of zero elements.
`hermitian_matrix (size_type size)`	Allocates an uninitialized `hermitian_matrix` that holds `size` rows of `size` elements.
`hermitian_matrix (const hermitian_matrix &m)`	The copy constructor.
`template<class AE> hermitian_matrix (const matrix_expression<AE> &ae)`	The extended copy constructor.
`void resize (size_type size, bool preserve = true)`	Reallocates a `hermitian_matrix` to hold `size` rows of `size` elements. The existing elements of the `hermitian_matrix` are preseved when specified.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns a `const` reference of the `j` -th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`hermitian_matrix &operator = (const hermitian_matrix &m)`	The assignment operator.
`hermitian_matrix &assign_temporary (hermitian_matrix &m)`	Assigns a temporary. May change the hermitian matrix `m` .
`template<class AE> hermitian_matrix &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> hermitian_matrix &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the hermitian matrix. Left and right hand side of the assignment should be independent.
`template<class AE> hermitian_matrix &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the hermitian matrix.
`template<class AE> hermitian_matrix &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the hermitian matrix. Left and right hand side of the assignment should be independent.

Member	Description
`template<class AE> hermitian_matrix &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the hermitian matrix.
`template<class AE> hermitian_matrix &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the hermitian matrix. Left and right hand side of the assignment should be independent.
`template<class AT> hermitian_matrix &operator *= (const AT &at)`	A computed assignment operator. Multiplies the hermitian matrix with a scalar.
`template<class AT> hermitian_matrix &operator /= (const AT &at)`	A computed assignment operator. Divides the hermitian matrix through a scalar.
`void swap (hermitian_matrix &m)`	Swaps the contents of the hermitian matrices.
`void insert (size_type i, size_type j, const_reference t)`	Inserts the value `t` at the `j`-th element of the `i`-th row.
`void erase (size_type i, size_type j)`	Erases the value at the `j`-th elemenst of the `i`-th row.
`void clear ()`	Clears the matrix.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `hermitian_matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `hermitian_matrix`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `hermitian_matrix`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `hermitian_matrix`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `hermitian_matrix`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `hermitian_matrix`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `hermitian_matrix`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `hermitian_matrix`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `hermitian_matrix`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `hermitian_matrix`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `hermitian_matrix`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `hermitian_matrix`.

Member	Description
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `hermitian_matrix`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `hermitian_matrix`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `hermitian_matrix`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `hermitian_matrix`.

Notes

[1] Supported parameters for the type of the hermitian matrix are `lower` and `upper`.

[2] Supported parameters for the storage organization are `row_major` and `column_major`.

[3] Supported parameters for the adapted array are `unbounded_array<T>` , `bounded_array<T>` and `std::vector<T>` .

Hermitian Adaptor

Description

The templated class `hermitian_adaptor<M, F>` is a hermitian matrix adaptor for other matrices.

Example

```
#include <boost/numeric/ublas/hermitian.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    matrix<std::complex<double> > m (3, 3);
    hermitian_adaptor<matrix<std::complex<double> >, lower> hal (m);
    for (unsigned i = 0; i < hal.size1 (); ++ i) {
        for (unsigned j = 0; j < i; ++ j)
            hal (i, j) = std::complex<double> (3 * i + j, 3 * i + j);
        hal (i, i) = std::complex<double> (4 * i, 0);
    }
    std::cout << hal << std::endl;
    hermitian_adaptor<matrix<std::complex<double> >, upper> hau (m);
    for (unsigned i = 0; i < hau.size1 (); ++ i) {
        hau (i, i) = std::complex<double> (4 * i, 0);
        for (unsigned j = i + 1; j < hau.size2 (); ++ j)
            hau (i, j) = std::complex<double> (3 * i + j, 3 * i + j);
    }
    std::cout << hau << std::endl;
}
```

Definition

Defined in the header hermitian.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of the adapted matrix.	
`F`	Functor describing the type of the hermitian adaptor. [1]	`lower`

Model of

Matrix Expression .

Type requirements

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

Public base classes

`matrix_expression<hermitian_adaptor<M, F> >`

Members

Member	Description
`hermitian_adaptor (matrix_type &data)`	Constructs a `hermitian_adaptor` of a matrix.
`hermitian_adaptor (const hermitian_adaptor &m)`	The copy constructor.
`template<class AE> hermitian_adaptor (const matrix_expression<AE> &ae)`	The extended copy constructor.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns a `const` reference of the `j` -th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`hermitian_adaptor &operator = (const hermitian_adaptor &m)`	The assignment operator.
`hermitian_adaptor &assign_temporary (hermitian_adaptor &m)`	Assigns a temporary. May change the hermitian adaptor `m`.
`template<class AE> hermitian_adaptor &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> hermitian_adaptor &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the hermitian adaptor. Left and right hand side of the assignment should be independent.

Member	Description
`template<class AE> hermitian_adaptor &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the hermitian adaptor.
`template<class AE> hermitian_adaptor &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the hermitian adaptor. Left and right hand side of the assignment should be independent.
`template<class AE> hermitian_adaptor &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the hermitian adaptor.
`template<class AE> hermitian_adaptor &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the hermitian adaptor. Left and right hand side of the assignment should be independent.
`template<class AT> hermitian_adaptor &operator *= (const AT &at)`	A computed assignment operator. Multiplies the hermitian adaptor with a scalar.
`template<class AT> hermitian_adaptor &operator /= (const AT &at)`	A computed assignment operator. Divides the hermitian adaptor through a scalar.
`void swap (hermitian_adaptor &m)`	Swaps the contents of the hermitian adaptors.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `hermitian_adaptor`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `hermitian_adaptor`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `hermitian_adaptor`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `hermitian_adaptor`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `hermitian_adaptor`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `hermitian_adaptor`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `hermitian_adaptor`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `hermitian_adaptor`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `hermitian_adaptor`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `hermitian_adaptor`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `hermitian_adaptor`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `hermitian_adaptor`.

Member	Description
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `hermitian_adaptor`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `hermitian_adaptor`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `hermitian_adaptor`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `hermitian_adaptor`.

Notes

[1] Supported parameters for the type of the hermitian adaptor are `lower` and `upper`.

Banded Matrix

Description

The templated class `banded_matrix<T, F, A>` is the base container adaptor for banded matrices. For a *(m x n)*-dimensional banded matrix with *l* lower and *u* upper diagonals and $0 <= i < m$, $0 <= j < n$ holds $b_{i,j} = 0$, if $i > j + l$ or $i < j - u$. The storage of banded matrices is packed.

Example

```
#include <boost/numeric/ublas/banded.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    banded_matrix<double> m (3, 3, 1, 1);
    for (signed i = 0; i < signed (m.size1 ()); ++ i)
        for (signed j = std::max (i - 1, 0); j < std::min (i + 2, signed (m.size2
())); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

Definition

Defined in the header banded.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the matrix.	

Parameter	Description	Default
`F`	Functor describing the storage organization. [1]	`row_major`
`A`	The type of the adapted array. [2]	`unbounded_array<T>`

Model of

Matrix .

Type requirements

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

Public base classes

`matrix_container<banded_matrix<T, F, A> >`

Members

Member	Description
`banded_matrix ()`	Allocates an uninitialized `banded_matrix` that holds zero rows of zero elements.
`banded_matrix (size_type size1, size_type size2, size_type lower = 0, size_type upper = 0)`	Allocates an uninitialized `banded_matrix` that holds `(lower + 1 + upper)` diagonals around the main diagonal of a matrix with `size1` rows of `size2` elements.
`banded_matrix (const banded_matrix &m)`	The copy constructor.
`template<class AE> banded_matrix (const matrix_expression<AE> &ae)`	The extended copy constructor.
`void resize (size_type size1, size_type size2, size_type lower = 0, size_type upper = 0, bool preserve = true)`	Reallocates a `banded_matrix` to hold `(lower + 1 + upper)` diagonals around the main diagonal of a matrix with `size1` rows of `size2` elements. The existing elements of the `banded_matrix` are preseved when specified.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`size_type lower () const`	Returns the number of diagonals below the main diagonal.
`size_type upper () const`	Returns the number of diagonals above the main diagonal.
`const_reference operator () (size_type i, size_type j) const`	Returns a `const` reference of the `j` -th element in the `i`-th row.

Member	Description
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i`-th row.
`banded_matrix &operator = (const banded_matrix &m)`	The assignment operator.
`banded_matrix &assign_temporary (banded_matrix &m)`	Assigns a temporary. May change the banded matrix `m`.
`template<class AE> banded_matrix &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> banded_matrix &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the banded matrix. Left and right hand side of the assignment should be independent.
`template<class AE> banded_matrix &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the banded matrix.
`template<class AE> banded_matrix &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the banded matrix. Left and right hand side of the assignment should be independent.
`template<class AE> banded_matrix &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the banded matrix.
`template<class AE> banded_matrix &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the banded matrix. Left and right hand side of the assignment should be independent.
`template<class AT> banded_matrix &operator *= (const AT &at)`	A computed assignment operator. Multiplies the banded matrix with a scalar.
`template<class AT> banded_matrix &operator /= (const AT &at)`	A computed assignment operator. Divides the banded matrix through a scalar.
`void swap (banded_matrix &m)`	Swaps the contents of the banded matrices.
`void insert (size_type i, size_type j, const_reference t)`	Inserts the value `t` at the `j`-th element of the `i`-th row.
`void erase (size_type i, size_type j)`	Erases the value at the `j`-th elemenst of the `i`-th row.
`void clear ()`	Clears the matrix.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `banded_matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `banded_matrix`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `banded_matrix`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `banded_matrix`.

Member	Description
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `banded_matrix`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `banded_matrix`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `banded_matrix`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `banded_matrix`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `banded_matrix`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `banded_matrix`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `banded_matrix`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `banded_matrix`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `banded_matrix`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `banded_matrix`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `banded_matrix`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `banded_matrix`.

Notes

[1] Supported parameters for the storage organization are `row_major` and `column_major`.

[2] Supported parameters for the adapted array are `unbounded_array<T>` , `bounded_array<T>` and `std::vector<T>` .

Banded Adaptor

Description

The templated class `banded_adaptor<M>` is a banded matrix adaptor for other matrices.

Example

```
#include <boost/numeric/ublas/banded.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    banded_adaptor<matrix<double> > ba (m, 1, 1);
    for (signed i = 0; i < signed (ba.size1 ()); ++ i)
        for (signed j = std::max (i - 1, 0); j < std::min (i + 2, signed (ba.size2
())); ++ j)
            ba (i, j) = 3 * i + j;
    std::cout << ba << std::endl;
}
```

Definition

Defined in the header banded.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of the adapted matrix.	

Model of

Matrix Expression .

Type requirements

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

Public base classes

`matrix_expression<banded_adaptor<M> >`

Members

Member	Description
`banded_adaptor (matrix_type &data, size_type lower = 0, size_type upper = 0)`	Constructs a `banded_adaptor` that holds `(lower + 1 + upper)` diagonals around the main diagonal of a matrix.
`banded_adaptor (const banded_adaptor &m)`	The copy constructor.
`template<class AE> banded_adaptor (const matrix_expression<AE> &ae)`	The extended copy constructor.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.

Member	Description
`size_type lower () const`	Returns the number of diagonals below the main diagonal.
`size_type upper () const`	Returns the number of diagonals above the main diagonal.
`const_reference operator () (size_type i, size_type j) const`	Returns a `const` reference of the `j` -th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`banded_adaptor &operator = (const banded_adaptor &m)`	The assignment operator.
`banded_adaptor &assign_temporary (banded_adaptor &m)`	Assigns a temporary. May change the banded adaptor `m` .
`template<class AE> banded_adaptor &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> banded_adaptor &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the banded adaptor. Left and right hand side of the assignment should be independent.
`template<class AE> banded_adaptor &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the banded adaptor.
`template<class AE> banded_adaptor &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the banded adaptor. Left and right hand side of the assignment should be independent.
`template<class AE> banded_adaptor &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the banded adaptor.
`template<class AE> banded_adaptor &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the banded adaptor. Left and right hand side of the assignment should be independent.
`template<class AT> banded_adaptor &operator *= (const AT &at)`	A computed assignment operator. Multiplies the banded adaptor with a scalar.
`template<class AT> banded_adaptor &operator /= (const AT &at)`	A computed assignment operator. Divides the banded adaptor through a scalar.
`void swap (banded_adaptor &m)`	Swaps the contents of the banded adaptors.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `banded_adaptor`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `banded_adaptor`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `banded_adaptor`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `banded_adaptor`.

Member	Description
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `banded_adaptor`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `banded_adaptor`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `banded_adaptor`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `banded_adaptor`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `banded_adaptor`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `banded_adaptor`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `banded_adaptor`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `banded_adaptor`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `banded_adaptor`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `banded_adaptor`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `banded_adaptor`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `banded_adaptor`.

Sparse Matricies

Mapped Matrix

Description

The templated class `mapped_matrix<T, F, A>` is the base container adaptor for sparse matricies using element maps. For a *(m xn)*-dimensional sparse matrix and $0 <= i < m$, $0 <= j < n$ the non-zero elements $h_{i,j}$ are mapped via *(i x n + j)* for row major orientation or via *(i + j x m)* for column major orientation to consecutive elements of the associative container, i.e. for elements $k = m_{i1,j1}$ and $k + 1 = m_{i2,j2}$ *of the container holds* $i_1 < i_2$ *or* ($i_1 = i_2$ *and* $j_1 < j_2$*) with row major orientation or* $j_1 < j_2$ *or* ($j_1 = j_2$ *and* $i_1 < i_2$) with column major orientation.

Example

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    mapped_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

Definition

Defined in the header matrix_sparse.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the mapped matrix.	
`F`	Functor describing the storage organization. [1]	`row_major`
`A`	The type of the adapted array. [2]	`map_std<std::size_t, T>`

Model of

Matrix .

Type requirements

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

Public base classes

`matrix_container<mapped_matrix<T, F, A> >`

Members

Member	Description
`mapped_matrix ()`	Allocates a `mapped_matrix` that holds at most zero rows of zero elements.
`mapped_matrix (size_type size1, size_type2, size_type non_zeros = 0)`	Allocates a `mapped_matrix` that holds at most `size1` rows of `size2` elements.
`mapped_matrix (const mapped_matrix &m)`	The copy constructor.

Member	Description
`template<class AE> mapped_matrix (size_type non_zeros, const matrix_expression<AE> &ae)`	The extended copy constructor.
`void resize (size_type size1, size_type size2, bool preserve = true)`	Reallocates a `mapped_matrix` to hold at most `size1` rows of `size2` elements. The existing elements of the `mapped_matrix` are preseved when specified.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`mapped_matrix &operator = (const mapped_matrix &m)`	The assignment operator.
`mapped_matrix &assign_temporary (mapped_matrix &m)`	Assigns a temporary. May change the mapped matrix `m` .
`template<class AE> mapped_matrix &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> mapped_matrix &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the mapped matrix. Left and right hand side of the assignment should be independent.
`template<class AE> mapped_matrix &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the mapped matrix.
`template<class AE> mapped_matrix &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the mapped matrix. Left and right hand side of the assignment should be independent.
`template<class AE> mapped_matrix &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the mapped matrix.
`template<class AE> mapped_matrix &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the mapped matrix. Left and right hand side of the assignment should be independent.
`template<class AT> mapped_matrix &operator *= (const AT &at)`	A computed assignment operator. Multiplies the mapped matrix with a scalar.
`template<class AT> mapped_matrix &operator /= (const AT &at)`	A computed assignment operator. Divides the mapped matrix through a scalar.
`void swap (mapped_matrix &m)`	Swaps the contents of the mapped matrices.
`true_refrence insert_element (size_type i, size_type j, const_reference t)`	Inserts the value `t` at the `j`-th element of the `i`-th row. Duplicates elements are not allowed.
`void erase_element (size_type i, size_type j)`	Erases the value at the `j`-th element of the `i`-th row.
`void clear ()`	Clears the mapped matrix.

Member	Description
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `mapped_matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `mapped_matrix`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `mapped_matrix`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `mapped_matrix`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `mapped_matrix`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `mapped_matrix`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `mapped_matrix`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `mapped_matrix`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `mapped_matrix`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `mapped_matrix`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `mapped_matrix`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `mapped_matrix`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `mapped_matrix`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `mapped_matrix`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `mapped_matrix`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `mapped_matrix`.

Notes

[1] Supported parameters for the storage organization are `row_major` and `column_major`.

[2] Supported parameters for the adapted array are `map_array<std::size_t, T>` and `map_std<std::size_t, T>`. The latter is equivalent to `std::map<std::size_t, T>`.

Compressed Matrix

Description

The templated class `compressed_matrix<T, F, IB, IA, TA>` is the base container adaptor for compressed matrices. For a *(m x n*)-dimensional compressed matrix and *0 < = i < m*, *0 < = j < n* the non-zero elements $m_{i,j}$ are mapped via *(i x n + j)* for row major orientation or via *(i + j x m)* for column major orientation to consecutive elements of the index and value containers, i.e. for elements $k = m_{i1,j1}$and $k + 1 = m_{i2,j2}$ *of the container holds* $i_1 < i_2$ *or* ($i_1 = i_2$ *and* $j_1 < j_2$*) with row major orientation or* $j_1 < j_2$ *or* ($j_1 = j_2$ *and* $i_1 < i\_\_{}_2$*)* with column major orientation.

Example

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    compressed_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

Definition

Defined in the header matrix_sparse.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the compressed matrix.	
`F`	Functor describing the storage organization. [1]	`row_major`
`IB`	The index base of the compressed vector. [2]	`0`
`IA`	The type of the adapted array for indices. [3]	`unbounded_array<std::size_t>`
`TA`	The type of the adapted array for values. [3]	`unbounded_array<T>`

Model of

Matrix .

Type requirements

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

Public base classes

`matrix_container<compressed_matrix<T, F, IB, IA, TA> >`

Members

Member	Description
`compressed_matrix ()`	Allocates a `compressed_matrix` that holds at most zero rows of zero elements.
`compressed_matrix (size_type size1, size_type2, size_type non_zeros = 0)`	Allocates a `compressed_matrix` that holds at most `size1` rows of `size2` elements.
`compressed_matrix (const compressed_matrix &m)`	The copy constructor.
`template<class AE> compressed_matrix (size_type non_zeros, const matrix_expression<AE> &ae)`	The extended copy constructor.
`void resize (size_type size1, size_type size2, bool preserve = true)`	Reallocates a `compressed_matrix` to hold at most `size1` rows of `size2` elements. The existing elements of the `compressed_matrix` are preseved when specified.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`compressed_matrix &operator = (const compressed_matrix &m)`	The assignment operator.
`compressed_matrix &assign_temporary (compressed_matrix &m)`	Assigns a temporary. May change the compressed matrix `m`.
`template<class AE> compressed_matrix &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> compressed_matrix &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the compressed matrix. Left and right hand side of the assignment should be independent.
`template<class AE> compressed_matrix &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the compressed matrix.
`template<class AE> compressed_matrix &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the compressed matrix. Left and right hand side of the assignment should be independent.

Member	Description
`template<class AE> compressed_matrix &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the compressed matrix.
`template<class AE> compressed_matrix &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the compressed matrix. Left and right hand side of the assignment should be independent.
`template<class AT> compressed_matrix &operator *= (const AT &at)`	A computed assignment operator. Multiplies the compressed matrix with a scalar.
`template<class AT> compressed_matrix &operator /= (const AT &at)`	A computed assignment operator. Divides the compressed matrix through a scalar.
`void swap (compressed_matrix &m)`	Swaps the contents of the compressed matrices.
`true_reference insert_element (size_type i, size_type j, const_reference t)`	Inserts the value `t` at the `j`-th element of the `i`-th row. Duplicates elements are not allowed.
`void erase_element (size_type i, size_type j)`	Erases the value at the `j`-th element of the `i`-th row.
`void clear ()`	Clears the compressed matrix.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `compressed_matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `compressed_matrix`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `compressed_matrix`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `compressed_matrix`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `compressed_matrix`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `compressed_matrix`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `compressed_matrix`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `compressed_matrix`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `compressed_matrix`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `compressed_matrix`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `compressed_matrix`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `compressed_matrix`.

Member	Description
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `compressed_matrix`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `compressed_matrix`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `compressed_matrix`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `compressed_matrix`.

Notes

[1] Supported parameters for the storage organization are `row_major` and `column_major`.

[2] Supported parameters for the index base are `0` and `1` at least.

[3] Supported parameters for the adapted array are `unbounded_array<>` , `bounded_array<>` and `std::vector<>` .

Coordinate Matrix

Description

The templated class `coordinate_matrix<T, F, IB, IA, TA>` is the base container adaptor for compressed matrices. For a *(m x n*)-dimensional sorted coordinate matrix and $0 <= i < m$, $0 <= j < n$ the non-zero elements $m_{i,j}$ are mapped via *(i x n + j)* for row major orientation or via *(i + j x m)* for column major orientation to consecutive elements of the index and value containers, i.e. for elements $k = m_{i1,j1}$ and $k + 1 = m_{i2,j2}$ *of the container holds* $i_1 < i_2$ *or* ($i_1 = i_2$ *and* $j_1 < j_2$*) with row major orientation or* $j_1 < j_2$ *or* ($j_1 = j_2$ *and* $i_1 < i\_\_2$*)* with column major orientation.

Example

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    coordinate_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

Definition

Defined in the header matrix_sparse.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the coordinate matrix.	
`F`	Functor describing the storage organization. [1]	`row_major`
`IB`	The index base of the coordinate vector. [2]	`0`
`IA`	The type of the adapted array for indices. [3]	`unbounded_array<std::size_t>`
`TA`	The type of the adapted array for values. [3]	`unbounded_array<T>`

Model of

Matrix .

Type requirements

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

Public base classes

```
matrix_container<coordinate_matrix<T, F, IB, IA, TA> >
```

Members

Member	Description
`coordinate_matrix ()`	Allocates a `coordinate_matrix` that holds at most zero rows of zero elements.
`coordinate_matrix (size_type size1, size_type2, size_type non_zeros = 0)`	Allocates a `coordinate_matrix` that holds at most `size1` rows of `size2` elements.
`coordinate_matrix (const coordinate_matrix &m)`	The copy constructor.
`template<class AE> coordinate_matrix (size_type non_zeros, const matrix_expression<AE> &ae)`	The extended copy constructor.
`void resize (size_type size1, size_type size2, bool preserve = true)`	Reallocates a `coordinate_matrix` to hold at most `size1` rows of `size2` elements. The existing elements of the `coordinate_matrix` are preseved when specified.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.

Member	Description
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i`-th row.
`coordinate_matrix &operator = (const coordinate_matrix &m)`	The assignment operator.
`coordinate_matrix &assign_temporary (coordinate_matrix &m)`	Assigns a temporary. May change the coordinate matrix `m`.
`template<class AE> coordinate_matrix &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> coordinate_matrix &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the coordinate matrix. Left and right hand side of the assignment should be independent.
`template<class AE> coordinate_matrix &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the coordinate matrix.
`template<class AE> coordinate_matrix &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the coordinate matrix. Left and right hand side of the assignment should be independent.
`template<class AE> coordinate_matrix &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the coordinate matrix.
`template<class AE> coordinate_matrix &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the coordinate matrix. Left and right hand side of the assignment should be independent.
`template<class AT> coordinate_matrix &operator *= (const AT &at)`	A computed assignment operator. Multiplies the coordinate matrix with a scalar.
`template<class AT> coordinate_matrix &operator /= (const AT &at)`	A computed assignment operator. Divides the coordinate matrix through a scalar.
`void swap (coordinate_matrix &m)`	Swaps the contents of the coordinate matrices.
`true_reference insert_element (size_type i, size_type j, const_reference t)`	Inserts the value `t` at the `j`-th element of the `i`-th row. Duplicates elements are not allowed.
`void append_element (size_type i, size_type j, const_reference t)`	Appends the value `t` at the `j`-th element of the `i`-th row. Duplicate elements can be appended to a `coordinate_matrix`. They are merged into a single arithmetically summed element by the `sort` function.
`void erase_element (size_type i, size_type j)`	Erases the value at the `j`-th element of the `i`-th row.
`void clear ()`	Clears the coordinate matrix.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `coordinate_matrix`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `coordinate_matrix`.

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

Notes

[1] Supported parameters for the storage organization are `row_major` and `column_major`.

[2] Supported parameters for the index base are `0` and `1` at least.

[3] Supported parameters for the adapted array are `unbounded_array<>` , `bounded_array<>` and `std::vector<>` .

Matrix Proxies

Matrix Row

Description

The templated class `matrix_row<M>` allows addressing a row of a matrix.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i) {
        matrix_row<matrix<double> > mr (m, i);
        for (unsigned j = 0; j < mr.size (); ++ j)
            mr (j) = 3 * i + j;
        std::cout << mr << std::endl;
    }
}
```

Definition

Defined in the header matrix_proxy.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of matrix referenced.	

Model of

Vector Expression .

If the specified row falls outside that of the row index range of the matrix, then the `matrix_row` is not a well formed Vector Expression. That is, access to an element which is outside of the matrix is *undefined*.

Type requirements

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

Public base classes

`vector_expression<matrix_row<M> >`

Members

Member	Description
`matrix_row (matrix_type &data, size_type i)`	Constructs a sub vector.
`size_type size () const`	Returns the size of the sub vector.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`reference operator () (size_type i)`	Returns a reference of the `i`-th element.
`matrix_row &operator = (const matrix_row &mr)`	The assignment operator.
`matrix_row &assign_temporary (matrix_row &mr)`	Assigns a temporary. May change the matrix row `mr` .
`template<class AE> matrix_row &operator = (const vector_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> matrix_row &assign (const vector_expression<AE> &ae)`	Assigns a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_row &operator += (const vector_expression<AE> &ae)`	A computed assignment operator. Adds the vector expression to the sub vector.
`template<class AE> matrix_row &plus_assign (const vector_expression<AE> &ae)`	Adds a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_row &operator -= (const vector_expression<AE> &ae)`	A computed assignment operator. Subtracts the vector expression from the sub vector.
`template<class AE> matrix_row &minus_assign (const vector_expression<AE> &ae)`	Subtracts a vector expression from the sub vector. Left and right hand side of the assignment should be independent.
`template<class AT> matrix_row &operator *= (const AT &at)`	A computed assignment operator. Multiplies the sub vector with a scalar.
`template<class AT> matrix_row &operator /= (const AT &at)`	A computed assignment operator. Divides the sub vector through a scalar.
`void swap (matrix_row &mr)`	Swaps the contents of the sub vectors.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `matrix_row`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `matrix_row`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `matrix_row`.
`iterator end ()`	Returns a `iterator` pointing to the end of the `matrix_row`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `matrix_row`.

Member	Description
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `matrix_row`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `matrix_row`.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `matrix_row`.

Projections

Description

The free `row` functions support the construction of matrix rows.

Prototypes

```
template<class M>
matrix_row<M> row (M &data, std::size_t i);
template<class M>
const matrix_row<const M> row (const M &data, std::size_t i);
```

Definition

Defined in the header matrix_proxy.hpp.

Type requirements

- `M` is a model of Matrix Expression .

Complexity

Linear depending from the size of the row.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i) {
        for (unsigned j = 0; j < m.size2 (); ++ j)
            row (m, i) (j) = 3 * i + j;
        std::cout << row (m, i) << std::endl;
    }
}
```

Matrix Column

Description

The templated class `matrix_column<M>` allows addressing a column of a matrix.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned j = 0; j < m.size2 (); ++ j) {
        matrix_column<matrix<double> > mc (m, j);
        for (unsigned i = 0; i < mc.size (); ++ i)
            mc (i) = 3 * i + j;
        std::cout << mc << std::endl;
    }
}
```

Definition

Defined in the header matrix_proxy.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of matrix referenced.	

Model of

Vector Expression .

If the specified column falls outside that of the column index range of the matrix, then the `matrix_column` is not a well formed Vector Expression. That is, access to an element which is outside of the matrix is *undefined*.

Type requirements

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

Public base classes

`vector_expression<matrix_column<M> >`

Members

Member	Description
`matrix_column (matrix_type &data, size_type j)`	Constructs a sub vector.
`size_type size () const`	Returns the size of the sub vector.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`reference operator () (size_type i)`	Returns a reference of the `i`-th element.
`matrix_column &operator = (const matrix_column &mc)`	The assignment operator.
`matrix_column &assign_temporary (matrix_column &mc)`	Assigns a temporary. May change the matrix column `mc` .
`template<class AE> matrix_column &operator = (const vector_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> matrix_column &assign (const vector_expression<AE> &ae)`	Assigns a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_column &operator += (const vector_expression<AE> &ae)`	A computed assignment operator. Adds the vector expression to the sub vector.
`template<class AE> matrix_column &plus_assign (const vector_expression<AE> &ae)`	Adds a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_column &operator -= (const vector_expression<AE> &ae)`	A computed assignment operator. Subtracts the vector expression from the sub vector.
`template<class AE> matrix_column &minus_assign (const vector_expression<AE> &ae)`	Subtracts a vector expression from the sub vector. Left and right hand side of the assignment should be independent.
`template<class AT> matrix_column &operator *= (const AT &at)`	A computed assignment operator. Multiplies the sub vector with a scalar.

Member	Description
`template<class AT> matrix_column &operator /= (const AT &at)`	A computed assignment operator. Divides the sub vector through a scalar.
`void swap (matrix_column &mc)`	Swaps the contents of the sub vectors.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `matrix_column`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `matrix_column`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `matrix_column`.
`iterator end ()`	Returns a `iterator` pointing to the end of the `matrix_column`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `matrix_column`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `matrix_column`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `matrix_column`.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `matrix_column`.

Projections

Description

The free `column` functions support the construction of matrix columns.

Prototypes

```
template<class M>
matrix_column<M> column (M &data, std::size_t j);
template<class M>
const matrix_column<const M> column (const M &data, std::size_t j);
```

Definition

Defined in the header matrix_proxy.hpp.

Type requirements

- `M` is a model of Matrix Expression .

Complexity

Linear depending from the size of the column.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned j = 0; j < m.size2 (); ++ j) {
        for (unsigned i = 0; i < m.size1 (); ++ i)
            column (m, j) (i) = 3 * i + j;
        std::cout << column (m, j) << std::endl;
    }
}
```

Vector Range

Description

The templated class `matrix_vector_range<M>` allows addressing a sub vector of a matrix.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;

    matrix_vector_range<matrix<double> > mvr (m, range (0, 3), range (0, 3));
    std::cout << mvr << std::endl;
}
```

Definition

Defined in the header matrix_proxy.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of matrix referenced.	

Model of

Vector Expression .

If the specified ranges fall outside that of the index range of the matrix, then the `matrix_vector_range` is not a well formed Vector Expression. That is, access to an element which is outside of the matrix is *undefined*.

Type requirements

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

Public base classes

`vector_expression<matrix_vector_range<M> >`

Members

Member	Description
`matrix_vector_range (matrix_type &data, const range &r1, const range &r2)`	Constructs a sub vector.
`size_type size () const`	Returns the size of the sub vector.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.
`reference operator () (size_type i)`	Returns a reference of the `i`-th element.
`matrix_vector_range &operator = (const matrix_vector_range &mvr)`	The assignment operator.
`matrix_vector_range &assign_temporary (matrix_vector_range &mvr)`	Assigns a temporary. May change the matrix vector range `mvr`.
`template<class AE> matrix_vector_range &operator = (const vector_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> matrix_vector_range &assign (const vector_expression<AE> &ae)`	Assigns a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_vector_range &operator += (const vector_expression<AE> &ae)`	A computed assignment operator. Adds the vector expression to the sub vector.
`template<class AE> matrix_vector_range &plus_assign (const vector_expression<AE> &ae)`	Adds a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_vector_range &operator -= (const vector_expression<AE> &ae)`	A computed assignment operator. Subtracts the vector expression from the sub vector.

Member	Description
`template<class AE> matrix_vector_range &minus_assign (const vector_expression<AE> &ae)`	Subtracts a vector expression from the sub vector. Left and right hand side of the assignment should be independent.
`template<class AT> matrix_vector_range &operator *= (const AT &at)`	A computed assignment operator. Multiplies the sub vector with a scalar.
`template<class AT> matrix_vector_range &operator /= (const AT &at)`	A computed assignment operator. Divides the sub vector through a scalar.
`void swap (matrix_vector_range &mvr)`	Swaps the contents of the sub vectors.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `matrix_vector_range`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `matrix_vector_range`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `matrix_vector_range`.
`iterator end ()`	Returns a `iterator` pointing to the end of the `matrix_vector_range`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the `matrix_vector_range`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `matrix_vector_range`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `matrix_vector_range`.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `matrix_vector_range`.

Vector Slice

Description

The templated class `matrix_vector_slice<M>` allows addressing a sliced sub vector of a matrix.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;

    matrix_vector_slice<matrix<double> > mvs (m, slice (0, 1, 3), slice (0, 1, 3));
    std::cout << mvs << std::endl;
}
```

Definition

Defined in the header matrix_proxy.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of matrix referenced.	

Model of

Vector Expression .

If the specified slices fall outside that of the index range of the matrix, then the `matrix_vector_slice` is not a well formed Vector Expression. That is, access to an element which is outside of the matrix is *undefined*.

Type requirements

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

Public base classes

`vector_expression<matrix_vector_slice<M> >`

Members

Member	Description
`matrix_vector_slice (matrix_type &data, const slice &s1, const slice &s2)`	Constructs a sub vector.
`size_type size () const`	Returns the size of the sub vector.
`const_reference operator () (size_type i) const`	Returns the value of the `i`-th element.

Member	Description
`reference operator () (size_type i)`	Returns a reference of the `i`-th element.
`matrix_vector_slice &operator = (const matrix_vector_slice &mvs)`	The assignment operator.
`matrix_vector_slice &assign_temporary (matrix_vector_slice &mvs)`	Assigns a temporary. May change the matrix vector slice `vs`.
`template<class AE> matrix_vector_slice &operator = (const vector_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> matrix_vector_slice &assign (const vector_expression<AE> &ae)`	Assigns a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_vector_slice &operator += (const vector_expression<AE> &ae)`	A computed assignment operator. Adds the vector expression to the sub vector.
`template<class AE> matrix_vector_slice &plus_assign (const vector_expression<AE> &ae)`	Adds a vector expression to the sub vector. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_vector_slice &operator -= (const vector_expression<AE> &ae)`	A computed assignment operator. Subtracts the vector expression from the sub vector.
`template<class AE> matrix_vector_slice &minus_assign (const vector_expression<AE> &ae)`	Subtracts a vector expression from the sub vector. Left and right hand side of the assignment should be independent.
`template<class AT> matrix_vector_slice &operator *= (const AT &at)`	A computed assignment operator. Multiplies the sub vector with a scalar.
`template<class AT> matrix_vector_slice &operator /= (const AT &at)`	A computed assignment operator. Divides the sub vector through a scalar.
`void swap (matrix_vector_slice &mvs)`	Swaps the contents of the sub vectors.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `matrix_vector_slice`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `matrix_vector_slice`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `matrix_vector_slice`.
`iterator end ()`	Returns a `iterator` pointing to the end of the `matrix_vector_slice`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `matrix_vector_slice`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `matrix_vector_slice`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `matrix_vector_slice`.

Member	Description
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `matrix_vector_slice`.

Matrix Range

Description

The templated class `matrix_range<M>` allows addressing a sub matrix of a matrix.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    matrix_range<matrix<double> > mr (m, range (0, 3), range (0, 3));
    for (unsigned i = 0; i < mr.size1 (); ++ i)
        for (unsigned j = 0; j < mr.size2 (); ++ j)
            mr (i, j) = 3 * i + j;
    std::cout << mr << std::endl;
}
```

Definition

Defined in the header matrix_proxy.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of matrix referenced.	

Model of

Matrix Expression .

If the specified ranges fall outside that of the index range of the matrix, then the `matrix_range` is not a well formed Matrix Expression. That is, access to an element which is outside of the matrix is *undefined*.

Type requirements

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

Public base classes

`matrix_expression<matrix_range<M> >`

Members

Member	Description
`matrix_range (matrix_type &data, const range &r1, const range &r2)`	Constructs a sub matrix.
`size_type start1 () const`	Returns the index of the first row.
`size_type size1 () const`	Returns the number of rows.
`size_type start2 () const`	Returns the index of the first column.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`matrix_range &operator = (const matrix_range &mr)`	The assignment operator.
`matrix_range &assign_temporary (matrix_range &mr)`	Assigns a temporary. May change the matrix range `mr` .
`template<class AE> matrix_range &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> matrix_range &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the sub matrix. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_range &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the sub matrix.
`template<class AE> matrix_range &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the sub matrix. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_range &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the sub matrix.
`template<class AE> matrix_range &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the sub matrix. Left and right hand side of the assignment should be independent.
`template<class AT> matrix_range &operator *= (const AT &at)`	A computed assignment operator. Multiplies the sub matrix with a scalar.
`template<class AT> matrix_range &operator /= (const AT &at)`	A computed assignment operator. Divides the sub matrix through a scalar.
`void swap (matrix_range &mr)`	Swaps the contents of the sub matrices.

Member	Description
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `matrix_range`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `matrix_range`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `matrix_range`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `matrix_range`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `matrix_range`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `matrix_range`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `matrix_range`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `matrix_range`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `matrix_range`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `matrix_range`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `matrix_range`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `matrix_range`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `matrix_range`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `matrix_range`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `matrix_range`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of reversed the `matrix_range`.

Simple Projections

Description

The free `subrange` functions support the construction of matrix ranges.

Prototypes

```
    template<class M>
    matrix_range<M> subrange (M &data,
       M::size_type start1, M::size_type stop1, M::size_type start2, M::size_type,
stop2);
    template<class M>
    const matrix_range<const M> subrange (const M &data,
       M::size_type start1, M::size_type stop1, M::size_type start2, M::size_type,
stop2);
```

Generic Projections

Description

The free `project` functions support the construction of matrix ranges. Existing `matrix_range`s can be composed with further ranges. The resulting ranges are computed using this existing ranges' `compose` function.

Prototypes

```
    template<class M>
    matrix_range<M> project (M &data, const range &r1, const range &r2);
    template<class M>
    const matrix_range<const M> project (const M &data, const range &r1, const range
&r2);
    template<class M>
    matrix_range<M> project (matrix_range<M> &data, const range &r1, const range &r2);
    template<class M>
    const matrix_range<M> project (const matrix_range<M> &data, const range &r1, const
range &r2);
```

Definition

Defined in the header matrix_proxy.hpp.

Type requirements

- `M` is a model of Matrix Expression .

Complexity

Quadratic depending from the size of the ranges.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            project (m, range (0, 3), range (0, 3)) (i, j) = 3 * i + j;
    std::cout << project (m, range (0, 3), range (0, 3)) << std::endl;
}
```

Matrix Slice

Description

The templated class `matrix_slice<M>` allows addressing a sliced sub matrix of a matrix.

Example

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    matrix_slice<matrix<double> > ms (m, slice (0, 1, 3), slice (0, 1, 3));
    for (unsigned i = 0; i < ms.size1 (); ++ i)
        for (unsigned j = 0; j < ms.size2 (); ++ j)
            ms (i, j) = 3 * i + j;
    std::cout << ms << std::endl;
}
```

Definition

Defined in the header matrix_proxy.hpp.

Template parameters

Parameter	Description	Default
`M`	The type of matrix referenced.	

Model of

Matrix Expression .

If the specified slices fall outside that of the index range of the matrix, then the `matrix_slice` is not a well formed Matrix Expression. That is, access to an element which is outside of the matrix is *undefined*.

Type requirements

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

Public base classes

`matrix_expression<matrix_slice<M> >`

Members

Member	Description
`matrix_slice (matrix_type &data, const slice &s1, const slice &s2)`	Constructs a sub matrix.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`matrix_slice &operator = (const matrix_slice &ms)`	The assignment operator.
`matrix_slice &assign_temporary (matrix_slice &ms)`	Assigns a temporary. May change the matrix slice `ms` .
`template<class AE> matrix_slice &operator = (const matrix_expression<AE> &ae)`	The extended assignment operator.
`template<class AE> matrix_slice &assign (const matrix_expression<AE> &ae)`	Assigns a matrix expression to the sub matrix. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_slice &operator += (const matrix_expression<AE> &ae)`	A computed assignment operator. Adds the matrix expression to the sub matrix.
`template<class AE> matrix_slice &plus_assign (const matrix_expression<AE> &ae)`	Adds a matrix expression to the sub matrix. Left and right hand side of the assignment should be independent.
`template<class AE> matrix_slice &operator -= (const matrix_expression<AE> &ae)`	A computed assignment operator. Subtracts the matrix expression from the sub matrix.
`template<class AE> matrix_slice &minus_assign (const matrix_expression<AE> &ae)`	Subtracts a matrix expression from the sub matrix. Left and right hand side of the assignment should be independent.
`template<class AT> matrix_slice &operator *= (const AT &at)`	A computed assignment operator. Multiplies the sub matrix with a scalar.

Member	Description
`template<class AT> matrix_slice &operator /= (const AT &at)`	A computed assignment operator. Multiplies the sub matrix through a scalar.
`void swap (matrix_slice &ms)`	Swaps the contents of the sub matrices.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the `matrix_slice`.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the `matrix_slice`.
`iterator1 begin1 ()`	Returns a `iterator1` pointing to the beginning of the `matrix_slice`.
`iterator1 end1 ()`	Returns a `iterator1` pointing to the end of the `matrix_slice`.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the `matrix_slice`.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the `matrix_slice`.
`iterator2 begin2 ()`	Returns a `iterator2` pointing to the beginning of the `matrix_slice`.
`iterator2 end2 ()`	Returns a `iterator2` pointing to the end of the `matrix_slice`.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed `matrix_slice`.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed `matrix_slice`.
`reverse_iterator1 rbegin1 ()`	Returns a `reverse_iterator1` pointing to the beginning of the reversed `matrix_slice`.
`reverse_iterator1 rend1 ()`	Returns a `reverse_iterator1` pointing to the end of the reversed `matrix_slice`.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed `matrix_slice`.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed `matrix_slice`.
`reverse_iterator2 rbegin2 ()`	Returns a `reverse_iterator2` pointing to the beginning of the reversed `matrix_slice`.
`reverse_iterator2 rend2 ()`	Returns a `reverse_iterator2` pointing to the end of the reversed `matrix_slice`.

Simple Projections

Description

The free `subslice` functions support the construction of matrix slices.

Prototypes

```
    template<class M>
    matrix_slice<M> subslice (M &data,
       M::size_type start1, M::difference_type stride1, M::size_type size1,
       M::size_type start2, M::difference_type stride2, M::size_type size2);
    template<class M>
    const matrix_slice<const M> subslice (const M &data,
       M::size_type start1, M::difference_type stride1, M::size_type size1,
       M::size_type start2, M::difference_type stride2, M::size_type size2);
```

Generic Projections

Description

The free `project` functions support the construction of matrix slices. Existing `matrix_slice` 's can be composed with further ranges or slices. The resulting slices are computed using this existing slices' `compose` function.

Prototypes

```
    template<class M>
    matrix_slice<M> project (M &data, const slice &s1, const slice &s2);
    template<class M>
    const matrix_slice<const M> project (const M &data, const slice &s1, const slice
&s2);
    template<class M>
    matrix_slice<M> project (matrix_slice<M> &data, const range &r1, const range &r2);
    template<class M>
    const matrix_slice<M> project (const matrix_slice<M> &data, const range &r1, const
range &r2);
    template<class M>
    matrix_slice<M> project (matrix_slice<M> &data, const slice &s1, const slice &s2);
    template<class M>
    const matrix_slice<M> project (const matrix_slice<M> &data, const slice &s1, const
slice &s2);
```

Definition

Defined in the header matrix_proxy.hpp.

Type requirements

- `M` is a model of Matrix Expression .

Complexity

Quadratic depending from the size of the slices.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            project (m, slice (0, 1, 3), slice (0, 1, 3)) (i, j) = 3 * i + j;
    std::cout << project (m, slice (0, 1, 3), slice (0, 1, 3)) << std::endl;
}
```

Matrix Expressions

Description

The templated class `matrix_expression<E>` is required to be a public base of all classes which model the Matrix Expression concept.

Definition

Defined in the header expression_types.hpp.

Template parameters

Parameter	Description	Default
`E`	The type of the matrix expression.	

Model of

None. Not a Matrix Expression

Type requirements

None.

Public base classes

None.

Members

Member	Description
`const expression_type &operator () () const`	Returns a `const` reference of the expression.
`expression_type &operator () ()`	Returns a reference of the expression.

Notes

The `operator[]`, `row`, `column`, `range`, `slice` and `project` functions have been removed. Use the free functions defined in matrix proxy instead.

Matrix Container

Description

The templated class `matrix_container<C>` is required to be a public base of all classes which model the Matrix concept. This includes the class `matrix` itself.

Definition

Defined in the header expression_types.hpp.

Template parameters

Parameter	Description	Default
`E`	The type of the matrix expression.	

Model of

None. Not a Matrix Expression OR Matrix

Type requirements

None.

Public base classes

`matrix_expression<C>`

Members

Member	Description
`const container_type &operator () () const`	Returns a `const` reference of the container.
`container_type &operator () ()`	Returns a reference of the container.

Matrix References

Reference

Description

The templated class `matrix_reference<E>` contains a reference to a matrix expression.

Definition

Defined in the header matrix_expression.hpp.

Template parameters

Parameter	Description	Default
`E`	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_reference<E> >`

Members

Member	Description
`matrix_reference (expression_type &e)`	Constructs a constant reference of the expression.
`void resize (size_type size1, size2)`	Resizes the expression to hold at most `size1` rows of `size2` elements.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`reference operator () (size_type i, size_type j)`	Returns a reference of the `j`-th element in the `i` -th row.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the expression.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the expression.

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

Matrix Operations

Unary Operation Description

Description

The templated classes `matrix_unary1<E, F>` and `matrix_unary2<E, F>` describe unary matrix operations.

Definition

Defined in the header matrix_expression.hpp.

Template parameters

Parameter	Description	Default
`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> >` resp.

Members

Member	Description
`matrix_unary1 (const expression_type &e)`	Constructs a description of the expression.
`matrix_unary2 (const expression_type &e)`	Constructs a description of the expression.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the expression.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the expression.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the expression.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the expression.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed expression.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed expression.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed expression.

Member	Description
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed expression.

Unary Operations

Prototypes

```
template<class E, class F>
    struct matrix_unary1_traits {
        typedef matrix_unary1<typename E::const_closure_type, F> expression_type;
        typedef expression_type result_type;
     };

    // (- m) [i] [j] = - m [i] [j]
    template<class E>
     typename matrix_unary1_traits<E, scalar_negate<typename E::value_type>
>::result_type
    operator - (const matrix_expression<E> &e);

    // (conj m) [i] [j] = conj (m [i] [j])
    template<class E>
     typename matrix_unary1_traits<E, scalar_conj<typename E::value_type>
>::result_type
    conj (const matrix_expression<E> &e);

    // (real m) [i] [j] = real (m [i] [j])
    template<class E>
     typename matrix_unary1_traits<E, scalar_real<typename E::value_type>
>::result_type
    real (const matrix_expression<E> &e);

    // (imag m) [i] [j] = imag (m [i] [j])
    template<class E>
     typename matrix_unary1_traits<E, scalar_imag<typename E::value_type>
>::result_type
    imag (const matrix_expression<E> &e);

    template<class E, class F>
    struct matrix_unary2_traits {
        typedef matrix_unary2<typename E::const_closure_type, F> expression_type;
        typedef expression_type result_type;
     };

    // (trans m) [i] [j] = m [j] [i]
    template<class E>
     typename matrix_unary2_traits<E, scalar_identity<typename E::value_type>
>::result_type
    trans (const matrix_expression<E> &e);

    // (herm m) [i] [j] = conj (m [j] [i])
    template<class E>
     typename matrix_unary2_traits<E, scalar_conj<typename E::value_type>
>::result_type
    herm (const matrix_expression<E> &e);
```

Description

`operator -` computes the additive inverse of a matrix expression. `conj` computes the complex conjugate of a matrix expression. `real` and `imag` compute the real and imaginary parts of a matrix expression. `trans` computes the transpose of a matrix expression. `herm` computes the hermitian, i.e. the complex conjugate of the transpose of a matrix expression.

Definition

Defined in the header matrix_expression.hpp.

Type requirements

- `E` is a model of Matrix Expression .

Preconditions

None.

Complexity

Quadratic depending from the size of the matrix expression.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<std::complex<double> > m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = std::complex<double> (3 * i + j, 3 * i + j);

    std::cout << - m << std::endl;
    std::cout << conj (m) << std::endl;
    std::cout << real (m) << std::endl;
    std::cout << imag (m) << std::endl;
    std::cout << trans (m) << std::endl;
    std::cout << herm (m) << std::endl;
}
```

Binary Operation Description

Description

The templated class `matrix_binary<E1, E2, F>` describes a binary matrix operation.

Definition

Defined in the header matrix_expression.hpp.

Template parameters

Parameter	Description	Default
`E1`	The type of the first matrix expression.	
`E2`	The type of the 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> >`.

Members

Member	Description
`matrix_binary (const expression1_type &e1, const expression2_type &e2)`	Constructs a description of the expression.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the expression.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the expression.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the expression.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the expression.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed expression.

Member	Description
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed expression.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed expression.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed expression.

Binary Operations

Prototypes

```
template<class E1, class E2, class F>
    struct matrix_binary_traits {
        typedef matrix_binary<typename E1::const_closure_type,
                              typename E2::const_closure_type, F> expression_type;
        typedef expression_type result_type;
     };

    // (m1 + m2) [i] [j] = m1 [i] [j] + m2 [i] [j]
    template<class E1, class E2>
    typename matrix_binary_traits<E1, E2, scalar_plus<typename E1::value_type,
                                                      typename E2::value_type>
>::result_type
    operator + (const matrix_expression<E1> &e1,
                const matrix_expression<E2> &e2);

    // (m1 - m2) [i] [j] = m1 [i] [j] - m2 [i] [j]
    template<class E1, class E2>
    typename matrix_binary_traits<E1, E2, scalar_minus<typename E1::value_type,
                                                       typename E2::value_type>
>::result_type
    operator - (const matrix_expression<E1> &e1,
                const matrix_expression<E2> &e2);
```

Description

`operator +` computes the sum of two matrix expressions. `operator -` computes the difference of two matrix expressions.

Definition

Defined in the header matrix_expression.hpp.

Type requirements

- `E1` is a model of Matrix Expression .

- `E2` is a model of Matrix Expression .

Preconditions

- `e1 ().size1 () == e2 ().size1 ()`
- `e1 ().size2 () == e2 ().size2 ()`

Complexity

Quadratic depending from the size of the matrix expressions.

Examples

```
#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);
    for (unsigned i = 0; i < std::min (m1.size1 (), m2.size1 ()); ++ i)
        for (unsigned j = 0; j < std::min (m1.size2 (), m2.size2 ()); ++ j)
            m1 (i, j) = m2 (i, j) = 3 * i + j;

    std::cout << m1 + m2 << std::endl;
    std::cout << m1 - m2 << std::endl;
}
```

Scalar Matrix Operation Description

Description

The templated classes `matrix_binary_scalar1<E1, E2, F>` and `matrix_binary_scalar2<E1, E2, F>` describe binary operations between a scalar and a matrix.

Definition

Defined in the header matrix_expression.hpp.

Template parameters

Parameter	Description	Default
`E1/E2`	The type of the scalar expression.	
`E2/E1`	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_binary_scalar1<E1, E2, F> >` and `matrix_expression<matrix_binary_scalar2<E1, E2, F> >` resp.

Members

Member	Description
`matrix_binary_scalar1 (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.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the expression.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the expression.
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the expression.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the expression.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed expression.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed expression.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed expression.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed expression.

Scalar Matrix Operations

Prototypes

```
template<class T1, class E2, class F>
    struct matrix_binary_scalar1_traits {
        typedef matrix_binary_scalar1<scalar_const_reference<T1>,
                                      typename E2::const_closure_type, F>
expression_type;
        typedef expression_type result_type;
     };

    // (t * m) [i] [j] = t * m [i] [j]
    template<class T1, class E2>
    typename matrix_binary_scalar1_traits<T1, E2, scalar_multiplies<T1, typename E2
::value_type> >::result_type
    operator * (const T1 &e1,
                const matrix_expression<E2> &e2);

    template<class E1, class T2, class F>
    struct matrix_binary_scalar2_traits {
        typedef matrix_binary_scalar2<typename E1::const_closure_type,
                                      scalar_const_reference<T2>, F> expression_type;
        typedef expression_type result_type;
     };

    // (m * t) [i] [j] = m [i] [j] * t
    template<class E1, class T2>
    typename matrix_binary_scalar2_traits<E1, T2, scalar_multiplies<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>
    typename matrix_binary_scalar2_traits<E1, T2, scalar_divides<typename E1
::value_type, T2> >::result_type
    operator / (const matrix_expression<E1> &e1,
                const T2 &e2);
```

Description

`operator *` computes the product of a scalar and a matrix expression. `operator /` multiplies the matrix with the reciprocal of the scalar.

Definition

Defined in the header matrix_expression.hpp.

Type requirements

- `T1`/`T2` is a model of Scalar Expression .

- `E2/E1` is a model of Matrix Expression .

Preconditions

None.

Complexity

Quadratic depending from the size of the matrix expression.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;

    std::cout << 2.0 * m << std::endl;
    std::cout << m * 2.0 << std::endl;
}
```

Matrix Vector Operations

Binary Operation Description

Description

The templated classes `matrix_vector_binary1<E1, E2, F>` and `matrix_vector_binary2<E1, E2, F>` describe binary matrix vector operations.

Definition

Defined in the header matrix_expression.hpp.

Template parameters

Parameter	Description	Default
`E1`	The type of the matrix or vector expression.	
`E2`	The type of the vector or matrix 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<matrix_vector_binary1<E1, E2, F> >` and `vector_expression<matrix_vector_binary2<E1, E2, F> >` resp.

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.

Binary Operations

Prototypes

```
template<class T1, class E1, class T2, class E2>
    struct matrix_vector_binary1_traits {
        typedef row_major_tag dispatch_category;
        typedef typename promote_traits<T1, T2>::promote_type promote_type;
        typedef matrix_vector_binary1<typename E1::const_closure_type,
                                      typename E2::const_closure_type,
                                      matrix_vector_prod1<T1, T2, promote_type> >
expression_type;
        typedef expression_type result_type;
     };
```

```
    template<class E1, class E2>
    typename matrix_vector_binary1_traits<typename E1::value_type, E1,
                                          typename E2::value_type, E2>::result_type
    prod (const matrix_expression<E1> &e1,
          const vector_expression<E2> &e2,
          row_major_tag);

    // Dispatcher
    template<class E1, class E2>
    typename matrix_vector_binary1_traits<typename E1::value_type, E1,
                                          typename E2::value_type, E2>::result_type
    prod (const matrix_expression<E1> &e1,
          const vector_expression<E2> &e2);

    template<class E1, class E2>
    typename matrix_vector_binary1_traits<typename type_traits<typename E1::
value_type>::precision_type, E1,
                                          typename type_traits<typename E2
::value_type>::precision_type, E2>::result_type
    prec_prod (const matrix_expression<E1> &e1,
               const vector_expression<E2> &e2,
               row_major_tag);

    // Dispatcher
    template<class E1, class E2>
    typename matrix_vector_binary1_traits<typename type_traits<typename E1::
value_type>::precision_type, E1,
                                          typename type_traits<typename E2
::value_type>::precision_type, E2>::result_type
    prec_prod (const matrix_expression<E1> &e1,
               const vector_expression<E2> &e2);

    template<class V, class E1, class E2>
    V
    prod (const matrix_expression<E1> &e1,
          const vector_expression<E2> &e2);

    template<class V, class E1, class E2>
    V
    prec_prod (const matrix_expression<E1> &e1,
               const vector_expression<E2> &e2);

    template<class T1, class E1, class T2, class E2>
    struct matrix_vector_binary2_traits {
        typedef column_major_tag dispatch_category;
        typedef typename promote_traits<T1, T2>::promote_type promote_type;
        typedef matrix_vector_binary2<typename E1::const_closure_type,
                                      typename E2::const_closure_type,
                                      matrix_vector_prod2<T1, T2, promote_type> >
expression_type;
        typedef expression_type result_type;
```

```
    };

    template<class E1, class E2>
    typename matrix_vector_binary2_traits<typename E1::value_type, E1,
                                          typename E2::value_type, E2>::result_type
    prod (const vector_expression<E1> &e1,
          const matrix_expression<E2> &e2,
          column_major_tag);

    // Dispatcher
    template<class E1, class E2>
    typename matrix_vector_binary2_traits<typename E1::value_type, E1,
                                          typename E2::value_type, E2>::result_type
    prod (const vector_expression<E1> &e1,
          const matrix_expression<E2> &e2);

    template<class E1, class E2>
    typename matrix_vector_binary2_traits<typename type_traits<typename E1::
value_type>::precision_type, E1,
                                          typename type_traits<typename E2
::value_type>::precision_type, E2>::result_type
    prec_prod (const vector_expression<E1> &e1,
               const matrix_expression<E2> &e2,
               column_major_tag);

    // Dispatcher
    template<class E1, class E2>
    typename matrix_vector_binary2_traits<typename type_traits<typename E1::
value_type>::precision_type, E1,
                                          typename type_traits<typename E2
::value_type>::precision_type, E2>::result_type
    prec_prod (const vector_expression<E1> &e1,
               const matrix_expression<E2> &e2);

    template<class V, class E1, class E2>
    V
    prod (const vector_expression<E1> &e1,
          const matrix_expression<E2> &e2);

    template<class V, class E1, class E2>
    V
    prec_prod (const vector_expression<E1> &e1,
               const matrix_expression<E2> &e2);
```

Description

`prod` computes the product of the matrix and the vector expression. `prec_prod` computes the double precision product of the matrix and the vector expression.

Definition

Defined in the header matrix_expression.hpp.

Type requirements

- `E1` is a model of Matrix Expression or Vector Expression .
- `E2` is a model of Vector Expression or Matrix Expression .

Preconditions

- `e1 ().size2 () == e2 ().size ()`
- `e1 ().size () == e2 ().size1 ()`

Complexity

Quadratic depending from the size of the matrix expression.

Examples

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    vector<double> v (3);
    for (unsigned i = 0; i < std::min (m.size1 (), v.size ()); ++ i) {
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
        v (i) = i;
    }

    std::cout << prod (m, v) << std::endl;
    std::cout << prod (v, m) << std::endl;
}
```

Triangular Solver

Prototypes

```
template<class E1, class E2>
    struct matrix_vector_solve_traits {
        typedef typename promote_traits<typename E1::value_type, typename E2
::value_type>::promote_type promote_type;
        typedef vector<promote_type> result_type;
    };

    template<class E1, class E2>
```

```
    void inplace_solve (const matrix_expression<E1> &e1,
                        E2 &e2,
                       lower_tag,
                       vector_tag);
    template<class E1, class E2>
    void inplace_solve (const matrix_expression<E1> &e1,
                        E2 &e2,
                       upper_tag,
                       vector_tag);
    template<class E1, class E2>
    void inplace_solve (const matrix_expression<E1> &e1,
                        E2 &e2,
                       unit_lower_tag,
                       vector_tag);
    template<class E1, class E2>
    void inplace_solve (const matrix_expression<E1> &e1,
                        E2 &e2,
                       unit_upper_tag,
                       vector_tag);

    template<class E1, class E2, class C>
    typename matrix_vector_solve_traits<E1, E2>::result_type
    solve (const matrix_expression<E1> &e1,
            const vector_expression<E2> &e2,
           C);

    template<class E1, class E2>
    void inplace_solve (E1 &e1,
                        const matrix_expression<E2> &e2,
                         vector_tag,
                         lower_tag);
    template<class E1, class E2>
    void inplace_solve (E1 &e1,
                        const matrix_expression<E2> &e2,
                         vector_tag,
                         upper_tag);
    template<class E1, class E2>
    void inplace_solve (E1 &e1,
                        const matrix_expression<E2> &e2,
                         vector_tag,
                         unit_lower_tag);
    template<class E1, class E2>
    void inplace_solve (E1 &e1,
                        const matrix_expression<E2> &e2,
                         vector_tag,
                         unit_upper_tag);

    template<class E1, class E2, class C>
    typename matrix_vector_solve_traits<E1, E2>::result_type
    solve (const vector_expression<E1> &e1,
            const matrix_expression<E2> &e2,
```

```
          C);
```

Description

`solve` solves a linear equation for lower or upper (unit) triangular matrices.

Definition

Defined in the header triangular.hpp.

Type requirements

- `E1` is a model of Matrix Expression or Vector Expression .
- `E2` is a model of Vector Expression or Matrix Expression .

Preconditions

- `e1 ().size1 () == e1 ().size2 ()`
- `e1 ().size2 () == e2 ().size ()`
- `e1 ().size () == e2 ().size1 ()`
- `e2 ().size1 () == e2 ().size2 ()`

Complexity

Quadratic depending from the size of the matrix expression.

Examples

```
#include <boost/numeric/ublas/triangular.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    vector<double> v (3);
    for (unsigned i = 0; i < std::min (m.size1 (), v.size ()); ++ i) {
        for (unsigned j = 0; j <= i; ++ j)
            m (i, j) = 3 * i + j + 1;
        v (i) = i;
    }

    std::cout << solve (m, v, lower_tag ()) << std::endl;
    std::cout << solve (v, m, lower_tag ()) << std::endl;
}
```

Matrix Matrix Operations

Binary Operation Description

Description

The templated class `matrix_matrix_binary<E1, E2, F>` describes a binary matrix operation.

Definition

Defined in the header matrix_expression.hpp.

Template parameters

Parameter	Description	Default
`E1`	The type of the first matrix expression.	
`E2`	The type of the 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_matrix_binary<E1, E2, F> >`.

Members

Member	Description
`matrix_matrix_binary (const expression1_type &e1, const expression2_type &e2)`	Constructs a description of the expression.
`size_type size1 () const`	Returns the number of rows.
`size_type size2 () const`	Returns the number of columns.
`const_reference operator () (size_type i, size_type j) const`	Returns the value of the `j`-th element in the `i`-th row.
`const_iterator1 begin1 () const`	Returns a `const_iterator1` pointing to the beginning of the expression.
`const_iterator1 end1 () const`	Returns a `const_iterator1` pointing to the end of the expression.

Member	Description
`const_iterator2 begin2 () const`	Returns a `const_iterator2` pointing to the beginning of the expression.
`const_iterator2 end2 () const`	Returns a `const_iterator2` pointing to the end of the expression.
`const_reverse_iterator1 rbegin1 () const`	Returns a `const_reverse_iterator1` pointing to the beginning of the reversed expression.
`const_reverse_iterator1 rend1 () const`	Returns a `const_reverse_iterator1` pointing to the end of the reversed expression.
`const_reverse_iterator2 rbegin2 () const`	Returns a `const_reverse_iterator2` pointing to the beginning of the reversed expression.
`const_reverse_iterator2 rend2 () const`	Returns a `const_reverse_iterator2` pointing to the end of the reversed expression.

Binary Operations

Prototypes

```
template<class T1, class E1, class T2, class E2>
    struct matrix_matrix_binary_traits {
        typedef unknown_orientation_tag dispatch_category;
        typedef typename promote_traits<T1, T2>::promote_type promote_type;
        typedef matrix_matrix_binary<typename E1::const_closure_type,
                                     typename E2::const_closure_type,
                                     matrix_matrix_prod<T1, T2, promote_type> >
expression_type;
        typedef expression_type result_type;
    };

    template<class E1, class E2>
    typename matrix_matrix_binary_traits<typename E1::value_type, E1,
                                         typename E2::value_type, E2>::result_type
    prod (const matrix_expression<E1> &e1,
          const matrix_expression<E2> &e2,
          unknown_orientation_tag);

    // Dispatcher
    template<class E1, class E2>
    typename matrix_matrix_binary_traits<typename E1::value_type, E1,
                                         typename E2::value_type, E2>::result_type
    prod (const matrix_expression<E1> &e1,
          const matrix_expression<E2> &e2);

    template<class E1, class E2>
    typename matrix_matrix_binary_traits<typename type_traits<typename E1::
value_type>::precision_type, E1,
                                         typename type_traits<typename E2::
```

```
value_type>::precision_type, E2>::result_type
    prec_prod (const matrix_expression<E1> &e1,
               const matrix_expression<E2> &e2,
               unknown_orientation_tag);

    // Dispatcher
    template<class E1, class E2>
    typename matrix_matrix_binary_traits<typename type_traits<typename E1::
value_type>::precision_type, E1,
                                         typename type_traits<typename E2::
value_type>::precision_type, E2>::result_type
    prec_prod (const matrix_expression<E1> &e1,
               const matrix_expression<E2> &e2);

    template<class M, class E1, class E2>
    M
    prod (const matrix_expression<E1> &e1,
          const matrix_expression<E2> &e2);

    template<class M, class E1, class E2>
    M
    prec_prod (const matrix_expression<E1> &e1,
               const matrix_expression<E2> &e2);
```

Description

`prod` computes the product of the matrix expressions. `prec_prod` computes the double precision product of the matrix expressions.

Definition

Defined in the header matrix_expression.hpp.

Type requirements

- `E1` is a model of Matrix Expression .
- `E2` is a model of Matrix Expression .

Preconditions

- `e1 ().size2 () == e2 ().size1 ()`

Complexity

Cubic depending from the size of the matrix expression.

Examples

```
#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);
    for (unsigned i = 0; i < std::min (m1.size1 (), m2.size1 ()); ++ i)
        for (unsigned j = 0; j < std::min (m1.size2 (), m2.size2 ()); ++ j)
            m1 (i, j) = m2 (i, j) = 3 * i + j;

    std::cout << prod (m1, m2) << std::endl;
}
```

Triangular Solvers

Prototypes

```
template<class E1, class E2>
    struct matrix_matrix_solve_traits {
        typedef typename promote_traits<typename E1::value_type, typename E2
::value_type>::promote_type promote_type;
        typedef matrix<promote_type> result_type;
    };

    template<class E1, class E2>
    void inplace_solve (const matrix_expression<E1> &e1,
                        E2 &e2,
                        lower_tag,
                        matrix_tag);
    template<class E1, class E2>
    void inplace_solve (const matrix_expression<E1> &e1,
                        E2 &e2,
                        upper_tag,
                        matrix_tag);
    template<class E1, class E2>
    void inplace_solve (const matrix_expression<E1> &e1,
                        E2 &e2,
                        unit_lower_tag,
                        matrix_tag);
    template<class E1, class E2>
    void inplace_solve (const matrix_expression<E1> &e1,
                        E2 &e2,
                        unit_upper_tag,
                        matrix_tag);

    template<class E1, class E2, class C>
    typename matrix_matrix_solve_traits<E1, E2>::result_type
    solve (const matrix_expression<E1> &e1,
           const matrix_expression<E2> &e2,
           C);
```

Description

`solve` solves a linear equation for lower or upper (unit) triangular matrices.

Definition

Defined in the header triangular.hpp.

Type requirements

- `E1` is a model of Matrix Expression .
- `E2` is a model of Matrix Expression .

Preconditions

- `e1 ().size1 () == e1 ().size2 ()`
- `e1 ().size2 () == e2 ().size1 ()`

Complexity

Cubic depending from the size of the matrix expressions.

Examples

```
#include <boost/numeric/ublas/triangular.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m1 (3, 3), m2 (3, 3);
    for (unsigned i = 0; i < std::min (m1.size1 (), m2.size1 ()); ++ i)
        for (unsigned j = 0; j <= i; ++ j)
            m1 (i, j) = m2 (i, j) = 3 * i + j + 1;

    std::cout << solve (m1, m2, lower_tag ()) << std::endl;
}
```

Tensor

Tensor Idealogy

tensor_index<value_t, storage_t, array_t, N>

Description

The template class `tensor_index` decorates the tensor template class with indices for tensor contraction.

Example

```
#include <boost/numeric/ublas/tensor/einstein.hpp>

int main () {
    using namespace boost::numeric::ublas;
    shape s{4,3,2};
    for (auto i = 0u; i < s.size(); ++i) {
        std::cout << s.at(i) << std::endl;
    }
}
```

Definition

Defined in the header tensor/tensor_einstein.hpp.

Public base classes

None.

Template parameters

Parameter	Description	
`value_t`	The type of object stored in the tensor.	
`format_t`	Storage organization of the tensor.	
`storage_t`	The type of the storage array of the tensor.	
`N`	Number of indices provided.	

Tensor Abstract

a. Description

The templated class `tensor<value_t,format_t,storage_t>` is the base container adaptor for dense tensors. Every element $(t_/ i_1, i_2, \dots, i_p)$ of a p-order $(n_1 \times n_2 \times \cdots \times n_p)$-dimensional tensor T is mapped to j-th element of a one-dimensional container where $(j = \sum_ /{r=1}^p i_r \cdot w_r)$ with $1 \leq i_r \leq n_r $ for $1 \leq r \leq p$. For the first-order orientation $w_1 = 1$ and $(w_k = n_{k-1} \cdot w_{k-1})$ for $k > 1$. For last-order orientation $w_p = 1$ and $(w_k = n_/{k+1} \cdot w_/{k+1})$ for $k < p$.

b. Example

```
#include <boost/numeric/ublas/tensor.hpp>

int main () {
  using namespace boost::numeric::ublas;
  tensor<double> t{4,2,3};
  for (auto k = 0ul; k < t.size (2); ++ k)
    for (auto j = 0ul; j < t.size (1); ++ j)
      for (auto i = 0ul; i < t.size (0); ++ i)
        t.at(i,j,k) = 3*i + 2*j + 5*k;

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

c. Definition

Defined in the header file `tensor/tensor.hpp`.

d. Model of

Tensor

e. Type requirements

None, except for those imposed by the requirements of Tensor .

f. Public base classes

`tensor_container<tensor<value_t,format_t,storage_t> >`

g. Template parameters

Parameter	Description	Default
`value_t`	The type of object stored in the tensor.	
`format_t`	Storage organization. [1]	`first_order`
`storage_t`	The type of the Storage array. [2]	`std::vector<value_t>`

h. Member types

Member type	Description
`value_type`	Type `value_t` of the tensor elements.
`layout_type`	Format of the tensor which is either `first_order` or `last_order`.
`array_type`	Sequence container type that stores all tensor elements and is accessible with a single index.
`strides_type`	Type of the strides vector `basic_strides<std::size_t,layout_type>` that stores all tensor elements and is accessible with a single index.
`extents_type`	Type of the dimension extents vector `shape` that stores all tensor elements and is accessible with a single index.
`size_type`	Unsigned integer which is usually `std::size_t`.
`difference_type`	Unsigned integer which is usually `std::ptrdiff_t`.
`reference`	Reference type `storage_type::reference` which is in most cases `value_type&`.
`const_reference`	Constant reference type `storage_type::const_reference` which is in most cases `const value_type&`.
`pointer`	Pointer type `storage_type::pointer` which is in most cases `value_type*`.
`const_pointer`	Constant reference type `storage_type::const_reference` which is in most cases `const value_type*`.
`iterator`	RandomAccessIterator `storage_type::iterator`.
`const_iterator`	Constant RandomAccessIterator `storage_type::const_iterator`.
`reverse_iterator`	Reverse RandomAccessIterator `storage_type::reverse_iterator`.
`const_reverse_iterator`	Reverse RandomAccessIterator `storage_type::const_reverse_iterator`.
`matrix_type`	Type of the matrix `matrix<value_type,layout_type,array_type>` with which the tensor type interacts.
`vector_type`	Type of the vector `matrix<value_type,layout_type,array_type>` with which the tensor type interacts.

i. Alias templates

Alias template	Description
`template<class derived_type> using tensor_expression_type = detail::tensor_expression<self_type,derived_type>`	Type of tensor_expression where `self_type` is the `tensor` type.
`template<class derived_type> using matrix_expression_type = matrix_expression<derived_type>`	Type of matrix_expression.
`template<class derived_type> using vector_expression_type = vector_expression<derived_type>`	Type of vector_expression.

j. Member Functions

i. Construction

Member function	Description
`tensor ()`	Constructs an uninitialized `tensor` that holds zero elements.
`tensor (std::initializer_list<size_type> list)`	Constructs an uninitialized `tensor` where `list` specifies the dimension extents.
`tensor (extents_type const& s)`	Constructs an uninitialized `tensor` where `s` specifies the dimension extents.
`tensor (extents_type const& e, array_type const& a)`	Constructs an uninitialized `tensor` where `e` specifies the dimension extents and `a` the data elements of the tensor.
<code>tensor (tensor<value_type,other_layout&rt; const& other)</code>	Constructs tensor by copying elements from `other` where the layout is different from this layout type.
`tensor (tensor const& other)`	Constructs tensor by copying elements from `other`.
`tensor (tensor && other)`	Constructs tensor by moving elements from `other`.
`tensor (matrix_type const& other)`	Constructs tensor by copying elements from `other` matrix. The tensor will have the order 2.
`tensor (matrix_type && other)`	Constructs tensor by moving elements from `other` matrix. The tensor will have the order 2.
`tensor (vector_type const& other)`	Constructs tensor by copying elements from `other` vector. The tensor will have the order 1.
`tensor (vector_type && other)`	Constructs tensor by moving elements from `other` vector. The tensor will have the order 1.

Member function	Description
`tensor (tensor_expression_type<derived_type> const& expr)`	Constructs tensor by evaluating the tensor expression `expr` and copying all elements of the result.
`tensor (matrix_expression_type<derived_type> const& expr)`	Constructs tensor by evaluating the matrix expression `expr` and copying all elements of the result.
`tensor (vector_expression_type<derived_type> const& expr)`	Constructs tensor by evaluating the vector expression `expr` and copying all elements of the result.

ii. Assignment

Member function	Description
`tensor& operator=(tensor_expression_type<derived_type> const& expr)`	Evaluates the tensor expression `expr` and copyies all elements of the result.
`tensor& operator=(tensor other)`	Copies or moves elements of `other`.
`tensor& operator=(const_reference v)`	Initialiates all elements of a tensor with `v`.

iii. Capacity

Member function	Description
`bool empty() const`	Returns true if a tensor has zero elements.
`size_type size() const`	Returns the number of elements of the tensor.
`size_type rank() const`	Returns the number of dimensions of the tensor.
`size_type order() const`	Returns the number of dimensions of the tensor.
`strides_type const& strides() const`	Returns a constant reference to the strides of the tensor.
`extents_type const& extents() const`	Returns a constant reference to the extents of the tensor.

iv. Element access

Member function	Description
`pointer data()`	Returns a `pointer` the first element of the tensor.
`const_pointer data() const`	Returns a `const_pointer` the first element of the tensor.
`reference operator[](size_type j)`	Returns a `reference` to the `j`-th element of the storage array of the tensor. Corresponds to the function call `tensor::data()+j`

Member function	Description
`const_reference operator[](size_type j) const`	Returns a `const_reference` to the `j`-th element of the storage array of the tensor. Corresponds to the function call `tensor::data()+j`.
`template<class ⋯ size_types> reference at(size_type i, size_types ⋯ is)`	Returns a `reference` to the `(i,is⋯)`-th element of the tensor where ` (i,is...)` denotes a multi-index with `tensor::order()` elements. If `sizeof⋯(is)==0`, `tensor::operator[i]` is called.
`template<class ⋯ size_types> const_reference at(size_type i, size_types ⋯ is)`	Returns a `const_reference` to the `(i,is⋯)`-th element of the tensor where ` (i,is...)` denotes a multi-index with `tensor::order()` elements. If `sizeof⋯(is)==0`, `tensor::operator[i]` is called.

v. Proxy Generation

Member function	Description
`template<std::size_t I, class ⋯ index_types> tensor_index operator()(indices::Index<I> p, index_types ⋯ ps)`	Returns a tensor index instance with index objects `(p,ps⋯)` for a tensor contraction where `sizeof⋯(ps)+1` must be equal to `tensor::order()`.

vi. Iterators

Member function	Description
`const_iterator begin() const`	Returns a const_iterator pointing to the first element of the tensor.
`const_iterator cbegin() const`	Returns a const_iterator pointing to the first element of the tensor.
`iterator begin()`	Returns an iterator pointing to the first element of the tensor.
`const_iterator end() const`	Returns a const_iterator pointing to the position after the last element of the tensor.
`const_iterator cend() const`	Returns a const_iterator pointing to the position after the last element of the tensor.
`iterator begin()`	Returns an iterator pointing to the position after the last element of the tensor.

vii. Modifiers

Member function	Description
`void reshape(extents_type const& e, value_type v = value_type{})`	Reshapes the tensor according to the extents `e`. If `e.product()` is greater than `tensor::size()`, the tensor is resized with `v`.

viii. Notes

[1] Supported parameters for the storage organization are `first_order` and `last_order`.

[2] Common parameters for the storage array are `std::array<N,T>` and `std::vector<T>`.

Tensor Expressions

Description

The templated class `tensor_expression<T,E>` is required to be a public base of all classes. There is no Tensor Expression concept defined.

Definition

Defined in the header tensor/expression.hpp.

Model of

None. Not a Tensor Expression!

Type requirements

None.

Public base classes

`ublas_expression<E>`.

Template parameters

Parameter	Description
`T`	The type of the tensor.
`E`	The type of the tensor expression.

Member types

Member type	Description
`expression_type`	Type of the derived expression which is `E`.
`type_category`	Tag for categorization which is `tensor_tag`.
`tensor_type`	Reference type which is `T`.

Public Member Functions

Member	Description
`const expression_type &operator()() const`	Returns a `const` reference to the derived expression.

Entrywise Tensor Operations

Binary Tensor Expression

Description

The templated class `binary_tensor_expression<T,EL,ER,OP>` contains a constant reference to a left and right expression that can be evaluated by using the access operator.

Definition

Defined in the header tensor/expression.hpp.

Model of

Tensor Expression

Type requirements

None.

Public base classes

`tensor_expression<T,binary_tensor_expression<T,EL,ER,OP>>`

Template parameters

Parameter	Description
`T`	Type of the tensor.
`EL`	Type of the left binary tensor expression.
`ER`	Type of the right binary tensor expression.
`OP`	Type of the binary operation.

Member types

Member type	Description
`expression_type_left`	Type of the left expression which is `EL`.
`expression_type_right`	Type of the right expression which is `ER`.
`tensor_type`	Reference type which is `T`.
`binary_operation`	Type of the binary operation which is `OP`.

Public Member Functions

Member	Description
`decltype(auto) operator()(std::size_t i) const`	Returns a `const` reference to the i-th element of the expression.

Unary Tensor Expression

Description

The templated class `unary_tensor_expression<T,E,OP>` contains a constant reference to an expression that can be evaluated by using the access operator.

Definition

Defined in the header tensor/expression.hpp.

Model of

Tensor Expression

Type requirements

None.

Public base classes

`tensor_expression<T,unary_tensor_expression<T,E,OP>>`

Template parameters

Parameter	Description
`T`	Type of the tensor.
`E`	Type of the unary tensor expression.
`OP`	Type of the unary operation.

Member types

Member type	Description
`expression_type`	Type of the expression which is `E`.
`tensor_type`	Reference type which is `T`.
`unary_operation`	Type of the unary operation which is `OP`.

Public Member Functions

Member	**Description**
`decltype(auto) operator()(std::size_t i) const`	Returns a `const` reference to the i-th element of the expression.

Tensor Extents

basic_extents<size_type>

Description

The template class `basic_extents` specifies dimension extents of a tensor instance.

Example

```
#include <boost/numeric/ublas/tensor/extents.hpp>

int main () {
    using namespace boost::numeric::ublas;
    shape s{4,3,2};
    for (auto i = 0u; i < s.size(); ++i) {
        std::cout << s.at(i) << std::endl;
    }
}
```

Definition

Defined in the header tensor/extents.hpp.

Public base classes

None.

Specialization

`using shape = basic_extents<std::size_t>`

Template parameters

Parameter	Description
`size_type`	Unsigned integer type.

Member types

Member type	Description
`value_type`	Type `size_type` of the extents.
`size_type`	Unsigned integer such as `std::size_t`.
`reference`	Reference type which is `value_type&`.
`const_reference`	Constant reference type which is `const value_type&`.
`pointer`	Pointer type which is `value_type*`.
`const_pointer`	Constant reference type which is `const value_type*`.

Member Functions

Member Function	Description
`basic_extents ()`	Constructs an empty instance of `basic_extents`.
`basic_extents (std::vector<value_type> const& vector)`	Constructs an instance copying the content of `vector`.
`basic_extents (std::vector<value_type> && vector)`	Constructs an instance moving the content of `vector`.
`basic_extents (std::initializer_list<value_type> list)`	Constructs an instance from `list`.
`basic_extents (const_iterator first, const_iterator last)`	Constructs an instance from the range specified by `[first,last)`.
`basic_extents (basic_extents const& other)`	Constructs an instance from `other` copying its elements.
`basic_extents (basic_extents && other)`	Constructs an instance from `other` by moving its elements.
`basic_extents& operator= (basic_extents other)`	Assigns the elements of `other` to this instance.
`bool is_scalar() const`	Returns true if the elements are `(1,1,[1,⋯,1])`.
`bool is_vector() const`	Returns true if the elements are `(n,1,[1,⋯,1])` or `(1,n,[1,⋯,1])` where `n>1`.
`bool is_matrix() const`	Returns true if the elements are `(m,n,[1,⋯,1])` where `m>1` and `n>1`.
`bool is_tensor() const`	Returns true if it is not a scalar, vector or matrix.
`const_pointer data() const`	Returns a `const_pointer` to the first element.
`pointer data()`	Returns a `pointer` to the first element.
`reference operator[](size_type i)`	Returns a `reference` to the `i`-th element.
`const_reference operator[](size_type i) const`	Returns a `const_reference` to the `i`-th element.
`reference at(size_type i)`	Returns a `reference` to the `i`-th element.
`const_reference at(size_type i) const`	Returns a `const_reference` to the `i`-th element.
`bool empty()`	Returns true if the container has no elements.
`size_type size() const`	Returns the number of elements.
`bool valid() const`	Returns true if size()>1 and all elements are greater than one.
`size_type product() const`	Returns the multiplication of all entries.
`basic_extents squeeze() const`	Returns a new instance where entries equal to one are eliminated.
`bool operator==(basic_extents const& b) const`	Returns true if all elements are equal.
`bool operator!=(basic_extents const& b) const`	Returns true if some elements are not equal.

Member Function	Description
`const_iterator begin() const`	Returns an `const_iterator pointing to the first element.`
`const_iterator end() const`	Returns a `const_iterator` pointing to an element passed the last element.
`std::vector<size_type> base() const`	Returns a const reference to the private member sequence container holding all elements.

Tensor Strides

basic_strides<size_type, format_type>

Description

The template class `basic_strides` contains weights for a given storage format in order to map multi-indices to scalar memory indices for tensor instances.

Example

```
#include <boost/numeric/ublas/tensor/strides.hpp>

int main () {
    using namespace boost::numeric::ublas;
    auto wf = strides<first_order>(shape{4,3,2});
    for (auto i = 0u; i < wf.size(); ++i)
        std::cout << wf.at(i) << std::endl;
        // 1,4,12

    auto wl = strides<first_order>(shape{4,3,2});
    for (auto i = 0u; i < wl.size(); ++i)
        std::cout << wl.at(i) << std::endl;
        // 6,2,1
}
```

Definition

Defined in the header tensor/strides.hpp.

Public base classes

None.

Specialization

`template<class format_t>using strides = basic_strides<std::size_t,format_t>`

Template parameters

Parameter	Description
`size_type`	Unsigned integer type.

Member types

Member type	Description
`value_type`	Type `size_type` of the strides.
`size_type`	Unsigned integer such as `std::size_t`.
`reference`	Reference type which is `value_type&`.
`const_reference`	Constant reference type which is `const value_type&`.
`pointer`	Pointer type which is `value_type*`.
`const_pointer`	Constant pointer type which is `const value_type*`.
`layout_type`	Layout type which can be either `boost::numeric::ublas::first_order` or `boost::numeric::ublas::last_order`.

Member Functions

Member Function	Description
`basic_strides ()`	Constructs an empty instance of `basic_strides`.
`basic_strides (basic_extents<value_type> const& be)`	Constructs an instance based on the tensor extents specified by `be`.
`basic_strides (std::vector<value_type> const& v)`	Constructs an instance copying the content of `v`.
`basic_strides (std::vector<value_type> && v)`	Constructs an instance moving the content of `v`.
`basic_strides (basic_strides const& other)`	Constructs an instance from `other` copying its elements.
`basic_strides (basic_strides && other)`	Constructs an instance from `other` by moving its elements.
`basic_strides& operator= (basic_strides other)`	Assigns the elements of `other` to this instance.
`const_pointer data() const`	Returns a `const_pointer` to the first element.
`const_reference operator[](size_type i) const`	Returns a `const_reference` to the `i`-th element.
`const_reference at(size_type i) const`	Returns a `const_reference` to the `i`-th element.
`bool empty()`	Returns true if the container has no elements.
`size_type size() const`	Returns the number of elements.
`void clear()`	Erases all elements.
`bool operator==(basic_strides const& b) const`	Returns true if all elements are equal.

Member Function	Description
`bool operator!=(basic_strides const& b) const`	Returns true if some elements are not equal.
`const_iterator begin() const`	Returns an `const_iterator` pointing to the first element.
`const_iterator end() const`	Returns a `const_iterator` pointing to an element passed the last element.
`std::vector<size_type> base() const`	Returns the private member sequence container holding all elements.

Non-Member Functions

Function	Description
`access(std::vector<size_type> const& i, strides<layout_type> w)`	Returns relative memory location depending on the multi-index vector `i` and strides `w`.
`access(size_type sum, strides<layout_type> w, size_type i, size_types ⋯ is)`	Returns relative memory location depending on the indices `i`, `is ⋯` and stride vector `w` (recursive function).

Tensor Expressions

Description

The templated class `tensor_expression<T,E>` is required to be a public base of all classes. There is no Tensor Expression concept defined.

Definition

Defined in the header tensor/expression.hpp.

Model of

None. Not a Tensor Expression!

Type requirements

None.

Public base classes

`ublas_expression<E>`.

Template parameters

Parameter	Description
`T`	The type of the tensor.
`E`	The type of the tensor expression.

Member types

Member type	Description
`expression_type`	Type of the derived expression which is `E`.
`type_category`	Tag for categorization which is `tensor_tag`.
`tensor_type`	Reference type which is `T`.

Public Member Functions

Member	Description
`const expression_type &operator()() const`	Returns a `const` reference to the derived expression.

Entrywise Tensor Operations

Binary Tensor Expression

Description

The templated class `binary_tensor_expression<T,EL,ER,OP>` contains a constant reference to a left and right expression that can be evaluated by using the access operator.

Definition

Defined in the header tensor/expression.hpp.

Model of

Tensor Expression

Type requirements

None.

Public base classes

`tensor_expression<T,binary_tensor_expression<T,EL,ER,OP>>`

Template parameters

Parameter	Description
`T`	Type of the tensor.
`EL`	Type of the left binary tensor expression.
`ER`	Type of the right binary tensor expression.
`OP`	Type of the binary operation.

Member types

Member type	Description
`expression_type_left`	Type of the left expression which is `EL`.
`expression_type_right`	Type of the right expression which is `ER`.
`tensor_type`	Reference type which is `T`.
`binary_operation`	Type of the binary operation which is `OP`.

Public Member Functions

Member	Description
`decltype(auto) operator()(std::size_t i) const`	Returns a `const` reference to the i-th element of the expression.

Unary Tensor Expression

Description

The templated class `unary_tensor_expression<T,E,OP>` contains a constant reference to an expression that can be evaluated by using the access operator.

Definition

Defined in the header tensor/expression.hpp.

Model of

Tensor Expression

Type requirements

None.

Public base classes

`tensor_expression<T,unary_tensor_expression<T,E,OP>>`

Template parameters

Parameter	Description
`T`	Type of the tensor.
`E`	Type of the unary tensor expression.
`OP`	Type of the unary operation.

Member types

Member type	Description

`expression_type`	Type of the expression which is `E`.
`tensor_type`	Reference type which is `T`.
`unary_operation`	Type of the unary operation which is `OP`.

Public Member Functions

Member	Description
`decltype(auto) operator()(std::size_t i) const`	Returns a `const` reference to the i-th element of the expression.

Miscellaneous

Unbounded Array Storage

Description

The templated class `unbounded_array<T, ALLOC>` implements a unbounded storage array using an allocator. The unbounded array is similar to a `std::vector` in that in can grow in size beyond any fixed bound. However `unbounded_array` is aimed at optimal performance. Therefore `unbounded_array` does not model a `Sequence` like `std::vector` does.

When resized `unbounded_array` will reallocate it's storage even if the new size requirement is smaller. It is therefore inefficient to resize a `unbounded_array`

Example

```
#include <boost/numeric/ublas/storage.hpp>

int main () {
    using namespace boost::numeric::ublas;
    unbounded_array<double> a (3);
    for (unsigned i = 0; i < a.size (); ++ i) {
        a [i] = i;
        std::cout << a [i] << std::endl;
    }
}
```

Definition

Defined in the header storage.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the array.	

`ALLOC`	An STL Allocator	std::allocator

Model of

Storage

Type requirements

None, except for those imposed by the requirements of Storage.

Public base classes

None.

Members

- The description does not describe what the member actually does, this can be looked up in the corresponding concept documentation, but instead contains a remark on the implementation of the member inside this model of the concept.
- Typography:
  - Members that are not part of the implemented concepts are in blue.

Member	Where defined	Description
`value_type`	Container	
`pointer`	Container	Defined as `value_type*`
`const_pointer`	Container	Defined as `const value_type*`
`reference`	Container	Defined as `value_type&`
`const_reference`	Container	Defined as `const value_type&`
`size_type`	Container	Defined as `Alloc::size_type`
`difference_type`	Container	Defined as `Alloc::difference_type`
`iterator`	Container	Defined as `pointer`
`const_iterator`	Container	Defined as `const_pointer`
`revere_iterator`	Container	Defined as `std::reverse_iterator<iterator>`
`const_revere_iterator`	Container	Defined as `std::reverse_iterator<const_iterator>`
allocator_type		Defined as ALLOC
`explicit unbounded_array (ALLOC &a = ALLOC())`	Storage	Creates an `unbounded_array` that holds zero elements, using a specified allocator.

Member	Where defined	Description
`explicit unbounded_array (size_type size, ALLOC &a = ALLOC())`	Storage	Creates a uninitialized `unbounded_array` that holds `size` elements, using a specified allocator. All the elements are default constructed.
`unbounded_array (size_type size, const T& init, ALLOC& a = ALLOC())`	Storage	Creates an initialized `unbounded_array` that holds `size` elements,using a specified allocator. All the elements are constructed from the `init` value.
`unbounded_array (const unbounded_array &a)`	Container	The copy constructor.
`~unbounded_array ()`	Container	Deallocates the `unbounded_array` itself.
`void resize (size_type n)`	Storage	Reallocates an `unbounded_array` to hold `n` elements. Values are uninitialised.
`void resize(size_type n, const T& t)`	Storage	Reallocates an `unbounded_array` to hold `n` elements. Values are copies of `t`
`size_type size () const`	Container	Returns the size of the `unbounded_array`.
`const_reference operator [] (size_type i) const`	Container	Returns a `const` reference of the `i` -th element.
`reference operator [] (size_type i)`	Container	Returns a reference of the `i`-th element.
`unbounded_array &operator = (const unbounded_array &a)`	Container	The assignment operator.
`unbounded_array &assign_temporary (unbounded_array &a)`		Assigns a temporary. May change the array `a`.
`void swap (unbounded_array &a)`	Container	Swaps the contents of the arrays.
`const_iterator begin () const`	Container	Returns a `const_iterator` pointing to the beginning of the `unbounded_array`.
`const_iterator end () const`	Container	Returns a `const_iterator` pointing to the end of the `unbounded_array`.

Member	Where defined	Description
`iterator begin ()`	Container	Returns a `iterator` pointing to the beginning of the `unbounded_array`.
`iterator end ()`	Container	Returns a `iterator` pointing to the end of the `unbounded_array`.
`const_reverse_iterator rbegin () const`	ReversibleContainer	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `unbounded_array`.
`const_reverse_iterator rend () const`	ReversibleContainer	Returns a `const_reverse_iterator` pointing to the end of the reversed `unbounded_array`.
`reverse_iterator rbegin ()`	ReversibleContainer	Returns a `reverse_iterator` pointing to the beginning of the reversed `unbounded_array`.
`reverse_iterator rend ()`	ReversibleContainer	Returns a `reverse_iterator` pointing to the end of the reversed `unbounded_array`.

Bounded Array Storage

Description

The templated class `bounded_array<T, N, ALLOC>` implements a bounded storage array. The bounded array is similar to a C++ array type in that its maximum size is bounded by N and is allocated on the stack instead of the heap. Similarly a `bounded_array` requires no secondary storage and ALLOC is only used to specify `size_type` and `difference_type`.

When resized `bounded_array` never reallocated the storage. It is therefore always efficient to resize a `bounded_array` but the size bound N must not be exceeded.

Example

```
#include <boost/numeric/ublas/storage.hpp>

int main () {
    using namespace boost::numeric::ublas;
    bounded_array<double, 3> a (3);
    for (unsigned i = 0; i < a.size (); ++ i) {
        a [i] = i;
        std::cout << a [i] << std::endl;
    }
}
```

Definition

Defined in the header storage.hpp.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the array.	
`N`	The allocation size of the array.	
`ALLOC`	An STL Allocator	std::allocator

Model of

Storage

Type requirements

None, except for those imposed by the requirements of Storage.

Public base classes

None.

Members

- The description does not describe what the member actually does, this can be looked up in the corresponding concept documentation, but instead contains a remark on the implementation of the member inside this model of the concept.
- Typography:
  - Members that are not part of the implemented concepts are in blue.

Member	Where defined	Description
`value_type`	Container	
`pointer`	Container	Defined as `value_type*`
`const_pointer`	Container	Defined as `const value_type*`
`reference`	Container	Defined as `value_type&`
`const_reference`	Container	Defined as `const value_type&`
`size_type`	Container	Defined as `Alloc::size_type`
`difference_type`	Container	Defined as `Alloc::difference_type`
`iterator`	Container	Defined as `pointer`
`const_iterator`	Container	Defined as `const_pointer`

Member	Where defined	Description
`revere_iterator`	Container	Defined as `std::reverse_iterator<iterator>`
`const_revere_iterator`	Container	Defined as `std::reverse_iterator<const_iterator>`
`bounded_array ()`	Storage	Creates an `unbounded_array` that holds **zero** elements.
`bounded_array (size_type size)`	Storage	Creates a uninitialized `bounded_array` that holds `size` elements. All the elements are default constructed.
`bounded_array (size_type size, const T& init)`	Storage	Creates an initialized `bounded_array` that holds `size` elements. All the elements are constructed from the `init` value.
`bounded_array (const bounded_array &c)`	Container	The copy constructor.
`~bounded_array ()`	Container	Deallocates the `bounded_array` itself.
`void resize (size_type size)`	Storage	Reallocates a `bounded_array` to hold `size` elements.
`void resize (size_type size, const T& t)`	Storage	Reallocates a `bounded_array` to hold `size` elements.
`size_type size () const`	Container	Returns the size of the `bounded_array`.
`const_reference operator [] (size_type i) const`	Container	Returns a `const` reference of the `i` -th element.
`reference operator [] (size_type i)`	Container	Returns a reference of the `i`-th element.
`bounded_array &operator = (const bounded_array &a)`	Container	The assignment operator.
`bounded_array &assign_temporary (bounded_array &a)`		Assigns a temporary. May change the array `a`.
`void swap (bounded_array &a)`	Container	Swaps the contents of the arrays.
`const_iterator begin () const`	Container	Returns a `const_iterator` pointing to the beginning of the `bounded_array`.

Member	Where defined	Description
`const_iterator end () const`	Container	Returns a `const_iterator` pointing to the end of the `bounded_array`.
`iterator begin ()`	Container	Returns a `iterator` pointing to the beginning of the `bounded_array`.
`iterator end ()`	Container	Returns a `iterator` pointing to the end of the `bounded_array`.
`const_reverse_iterator rbegin () const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `bounded_array`.
`const_reverse_iterator rend () const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the end of the reversed `bounded_array`.
`reverse_iterator rbegin ()`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed `bounded_array`.
`reverse_iterator rend ()`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed `bounded_array`.

Overview of Tensor, Matrix- and Vector Types

Contents:

Notation

`T`	is the data type. For general linear algebra operations this will be a real type e.g. `double`, ...
`F`	is the orientation type, either `row_major` or `column_major` for matrices and `first_order` or `last_order` for tensors
`A, IA, TA`	is an array storage type, e.g. `std::vector`, `bounded_array`, `unbounded_array`, ⋯

`TRI`	is a triangular functor: `lower, unit_lower, strict_lower, upper, unit_upper, strict_upper`
`M, N, K`	are unsigned integer sizes (`std::size_t`)
`IB`	is an index base (`std::size_t`)
`VEC`	is any vector type
`MAT`	is any matrix type
`TEN`	is any tensor type
`[…]`	denote optional arguments - for more details look at the section "storage layout".

Vectors

Definition	Description
`vector<T [, A]>  v(size);`	a dense vector of values of type `T` of variable size. A storage type `A` can be specified which defaults to `unbounded_array`. Elements are constructed by `A`, which need not initialise their value.
`bounded_vector<T, N>  v;`	a dense vector of values of type `T` of variable size but with maximum `N`. The default constructor creates `v` with size `N`. Elements are constructed by the storage type `bounded_array`, which need not initialise their value.
`c_vector<T, M>  v(size);`	a dense vector of values of type `T` with the given size. The data is stored as an ordinary C++ array `T data_[M]`
`zero_vector<T>  v(size);`	the zero vector of type `T` with the given size.
`unit_vector<T>  v(size, index);`	the unit vector of type `T` with the given size. The vector is zero other then a single specified element. `index` should be less than `size`.
`mapped_vector<T [, S]>  v(size);`	a sparse vector of values of type `T` of variable size. The sparse storage type `S` can be `std::map<size_t, T>` or `map_array<size_t, T>`.
`compressed_vector<T [,IB, IA, TA]>  v(size);`	a sparse vector of values of type `T` of variable size. The non zero values are stored as two seperate arrays - an index array and a value array. The index array is always sorted and there is at most one entry for each index.

Definition	Description
`coordinate_vector<T [,IB, IA, TA]>  v(size);`	a sparse vector of values of type `T` of variable size. The non zero values are stored as two seperate arrays - an index array and a value array. The arrays may be out of order with multiple entries for each vector element. If there are multiple values for the same index the sum of these values is the real value.

Note: the default types are defined in `boost/numeric/ublas/fwd.hpp`.

Vector Proxies

Definition	Description
`vector_range<VEC>  vr(v, range);`	a vector referencing a continuous subvector of elements of vector `v` containing all elements specified by `range`.
`vector_slice<VEC>  vs(v, slice);`	a vector referencing a non continuous subvector of elements of vector `v` containing all elements specified by `slice`.
`matrix_row<MAT>  vr(m, index);`	a vector referencing the `index`-th row of matrix `m`
`matrix_column<MAT>  vc(m, index);`	a vector referencing the `index`-th column of matrix `m`

Matrices

Definition	Description
`matrix<T [, F, A]>  m(size1, size2);`	a dense matrix of values of type `T` of variable size. A storage type `A` can be specified which defaults to `unbounded_array`. The orientation functor `F` defaults to `row_major`. Elements are constructed by `A`, which need not initialise their value.
`bounded_matrix<T, M, N [, F]>  m;`	a dense matrix of type `T` with variable size with maximum `M`-by-`N`. The orientation functor `F` defaults to `row_major`. The default constructor creates `m` with size `M`-by-`N`. Elements are constructed by the storage type `bounded_array`, which need not initialise their value.
`c_matrix<T, M, N>  m(size1, size2);`	a dense matrix of values of type `T` with the given size. The data is stored as an ordinary C++ array `T data_[N][M]`

Definition	Description
`vector_of_vector<T [, F, A]>  m(size1, size2);`	a dense matrix of values of type `T` with the given size. The data is stored as a vector of vectors. The orientation `F` defaults to `row_major`. The storage type `S` defaults to `unbounded_array<unbounded_array<T> >`
`zero_matrix<T>  m(size1, size2);`	a zero matrix of type `T` with the given size.
`identity_matrix<T>  m(size1, size2);`	an identity matrix of type `T` with the given size. The values are `v(i,j) = (i==j)?T(1):T()`.
`scalar_matrix<T>  m(size1, size2, value);`	a matrix of type `T` with the given size that has the value `value` everywhere.
`triangular_matrix<T [, TRI, F, A]>  m(size);`	a triangular matrix of values of type `T` of variable size. Only the nonzero elements are stored in the given order `F`. ("triangular packed storage") The triangular type `F` defaults to `lower`, the orientation type `F` defaults to `row_major`.
`banded_matrix<T [, F, A]>  m(size1, size2, n_lower, n_upper);`	a banded matrix of values of type `T` of variable size with `n_lower` sub diagonals and `n_upper` super diagonals. Only the nonzero elements are stored in the given order `F`. ("packed storage")
`symmetric_matrix<T [, TRI, F, A]>  m(size);`	a symmetric matrix of values of type `T` of variable size. Only the given triangular matrix is stored in the given order `F`.
`hermitian_matrix<T [, TRI, F, A]>  m(size);`	a hermitian matrix of values of type `T` of variable size. Only the given triangular matrix is stored using the order `F`.
`mapped_matrix<T, [F, S]>  m(size1, size2 [, non_zeros]);`	a sparse matrix of values of type `T` of variable size. The sparse storage type `S` can be either `std::map<size_t, std::map<size_t, T> >` or `map_array<size_t, map_array<size_t, T> >`.
`sparse_vector_of_sparse_vector<T, [F, C]> m(size1, size2 [, non_zeros]);`	a sparse matrix of values of type `T` of variable size.
`compressed_matrix<T, [F, IB, IA, TA]> m(size1, size2 [, non_zeros]);`	a sparse matrix of values of type `T` of variable size. The values are stored in compressed row/column storage.
`coordinate_matrix<T, [F, IB, IA, TA]> m(size1, size2 [, non_zeros]);`	a sparse matrix of values of type `T` of variable size. The values are stored in 3 parallel array as triples (i, j, value). More than one value for each pair of indices is possible, the real value is the sum of all.

Definition	Description
`generalized_vector_of_vector<T, F, A> m(size1, size2 [, non_zeros]);`	a sparse matrix of values of type `T` of variable size. The values are stored as a vector of sparse vectors, e.g. `generalized_vector_of_vector<double, row_major, unbounded_array<coordinate_vector<double> > >`

Note: the default types are defined in `boost/numeric/ublas/fwd.hpp`.

Matrix Proxies

Definition	Description
`triangular_adaptor<MAT, TRI>  ta(m);`	a triangular matrix referencing a selection of elements of the matrix `m`.
`symmetric_adaptor<MAT, TRI>  sa(m);`	a symmetric matrix referencing a selection of elements of the matrix `m`.
`hermitian_adaptor<MAT, TRI>  ha(m);`	a hermitian matrix referencing a selection of elements of the matrix `m`.
`banded_adaptor<MAT>  ba(m, n_lower, n_upper);`	a banded matrix referencing a selection of elements of the matrix `m`.
`matrix_range<MAT, TRI>  mr(m, range1, range2);`	a matrix referencing a submatrix of elements in the matrix `m`.
`matrix_slice<MAT, TRI>  ms(m, slice1, slice2);`	a matrix referencing a non continues submatrix of elements in the matrix `m`.

Tensors

Definition	Description
`tensor<T [, F, A]>  t(size1, size2, ⋯ );`	a dense matrix of values of type `T` of variable size. A storage type `A` can be specified which defaults to `std::vector<T>`. The orientation type `F` defaults to `first_order`. Elements are constructed by `A`, which need not initialise their value.

Special Storage Layouts

The library supports conventional dense, packed and basic sparse vector and matrix storage layouts. The description of the most common constructions of vectors and matrices comes next.

Construction	Comment
`vector<T,  std::vector<T> >  v (size)`	a dense vector, storage is provided by a standard vector. The storage layout usually is BLAS compliant.

Construction	Comment
`vector<T, unbounded_array<T> > v (size)`	a dense vector, storage is provided by a heap-based array. The storage layout usually is BLAS compliant.
`vector<T, bounded_array<T, N> > v (size)`	a dense vector, storage is provided by a stack-based array. The storage layout usually is BLAS compliant.
`mapped_vector<T, std::map<std::size_t, T> > v (size, non_zeros)`	a sparse vector, storage is provided by a standard map.
`mapped_vector<T, map_array<std::size_t, T> > v (size, non_zeros)`	a sparse vector, storage is provided by a map array.
`matrix<T, row_major, std::vector<T> > m (size1, size2)`	a dense matrix, orientation is row major, storage is provided by a standard vector.
`matrix<T, column_major, std::vector<T> > m (size1, size2)`	a dense matrix, orientation is column major, storage is provided by a standard vector. The storage layout usually is BLAS compliant.
`matrix<T, row_major, unbounded_array<T> > m (size1, size2)`	a dense matrix, orientation is row major, storage is provided by a heap-based array.
`matrix<T, column_major, unbounded_array<T> > m (size1, size2)`	a dense matrix, orientation is column major, storage is provided by a heap-based array. The storage layout usually is BLAS compliant.
`matrix<T, row_major, bounded_array<T, N1 * N2> > m (size1, size2)`	a dense matrix, orientation is row major, storage is provided by a stack-based array.
`matrix<T, column_major, bounded_array<T, N1 * N2> > m (size1, size2)`	a dense matrix, orientation is column major, storage is provided by a stack-based array. The storage layout usually is BLAS compliant.
`triangular_matrix<T, row_major, F, A> m (size)`	a packed triangular matrix, orientation is row major.
`triangular_matrix<T, column_major, F, A> m (size)`	a packed triangular matrix, orientation is column major. The storage layout usually is BLAS compliant.
`banded_matrix<T, row_major, A> m (size1, size2, lower, upper)`	a packed banded matrix, orientation is row major.
`banded_matrix<T, column_major, A> m (size1, size2, lower, upper)`	a packed banded matrix, orientation is column major. The storage layout usually is BLAS compliant.
`symmetric_matrix<T, row_major, F, A> m (size)`	a packed symmetric matrix, orientation is row major.
`symmetric_matrix<T, column_major, F, A> m (size)`	a packed symmetric matrix, orientation is column major. The storage layout usually is BLAS compliant.

Construction	Comment
`hermitian_matrix<T, row_major, F, A> m (size)`	a packed hermitian matrix, orientation is row major.
`hermitian_matrix<T, column_major, F, A> m (size)`	a packed hermitian matrix, orientation is column major. The storage layout usually is BLAS compliant.
`mapped_matrix<T, row_major, std::map<std::size_t, T> > m (size1, size2, non_zeros)`	a sparse matrix, orientation is row major, storage is provided by a standard map.
`mapped_matrix<T, column_major, std::map<std::size_t, T> > m (size1, size2, non_zeros)`	a sparse matrix, orientation is column major, storage is provided by a standard map.
`mapped_matrix<T, row_major, map_array<std::size_t, T> > m (size1, size2, non_zeros)`	a sparse matrix, orientation is row major, storage is provided by a map array.
`mapped_matrix<T, column_major, map_array<std::size_t, T> > m (size1, size2, non_zeros)`	a sparse matrix, orientation is column major, storage is provided by a map array.
`compressed_matrix<T, row_major> m (size1, size2, non_zeros)`	a compressed matrix, orientation is row major. The storage layout usually is BLAS compliant.
`compressed_matrix<T, column_major> m (size1, size2, non_zeros)`	a compressed matrix, orientation is column major. The storage layout usually is BLAS compliant.
`coordinate_matrix<T, row_major> m (size1, size2, non_zeros)`	a coordinate matrix, orientation is row major. The storage layout usually is BLAS compliant.
`coordinate_matrix<T, column_major> m (size1, size2, non_zeros)`	a coordinate matrix, orientation is column major. The storage layout usually is BLAS compliant.

Range and Slice Storage

Range<SizeType,DistanceType>

Description

The class `range` specifies a range of indicies. The range is a sequence of indices from a start value to stop value. The indices increase by one and exlude the stop value. `range` can therefore be used to specify ranges of elements from vectors and matrices.

Example

```
#include <boost/numeric/ublas/storage.hpp>

int main () {
    using namespace boost::numeric::ublas;
    range r (0, 3);
    for (unsigned i = 0; i < r.size (); ++ i) {
        std::cout << r (i) << std::endl;
    }
}
```

Definition

Defined in the header storage.hpp.

Model of

Reversible Container.

Type requirements

None, except for those imposed by the requirements of Reversible Container.

Public base classes

None.

Members

Member	Description
`range (size_type start, size_type stop)`	Constructs a range of indicies from `start` to `stop (excluded)` .
`size_type start () const`	Returns the beginning of the `range`.
`size_type size () const`	Returns the size of the `range`.
`const_reference operator [] (size_type i) const`	Returns the value `start + i` of the `i` -th element.
`range compose (const range &r) const`	Returns the composite range from `start + r.start ()` to `start + r.start () + r.size ()`.
`bool operator == (const range &r) const`	Tests two ranges for equality.
`bool operator != (const range &r) const`	Tests two ranges for inequality.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `range`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `range`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `range`.

Member	Description
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `range`.

Preconditions

- `start () < = stop ()`

Slice<SizeType,DistanceType>

Description

The class `slice` specifies a 'slice' of indicies. Slices are more general then ranges, the stride allows the sequence of indicies to increase and decrease by the specified amount between element. `slice` can therefore be used to specify slices of element from vectors and matrices.

Example

```
#include <boost/numeric/ublas/storage.hpp>

int main () {
    using namespace boost::numeric::ublas;
    slice s (0, 1, 3);
    for (unsigned i = 0; i < s.size (); ++ i) {
        std::cout << s (i) << std::endl;
    }
}
```

Definition

Defined in the header storage.hpp.

Model of

Reversible Container.

Type requirements

None, except for those imposed by the requirements of Reversible Container.

Public base classes

None.

Members

Member	Description
`slice (size_type start, size_type stride, size_type size)`	Constructs a slice `start,start+stride,start+2*stride⋯` with `size` elements.
`size_type start () const`	Returns the beginning of the `slice`.
`size_type stride () const`	Returns the stride of the `slice`.
`size_type size () const`	Returns the size of the `slice`.
`const_reference operator [] (size_type i) const`	Returns the value `start + i * stride` of the `i`-th element.
`slice compose (const range &r) const`	Returns the composite slice from `start + stride * r.start ()` to `start + stride * (r.start () + r.size ())` with stride `stride`.
`slice compose (const slice &s) const`	Returns the composite slice from `start + stride * s.start ()` to `start + stride * s.stride () * (s.start () + s.size ())` with stride `stride * s.stride ()`.
`bool operator == (const slice &s) const`	Tests two slices for equality.
`bool operator != (const slice &s) const`	Tests two slices for inequality.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `slice`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `slice`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `slice`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `slice`.

Preconditions

- None all strides are vaild. However when an index is returned or an iterator is dereferenced its value must be representable as the size_type.

Boost Basic Linear Algebra - Configuration Options

NDEBUG

Make sure you define NDEBUG The only way uBLAS knows you want a release configuration is to check if you have defined NDEBUG. If you don't it assumes you want a debug configuration and adds a lot of very useful runtime check. However these are very slow!

BOOST_UBLAS_MOVE_SEMANTICS

The patch and description was provided by Nasos Iliopoulos.

An immediate effect of this option is the elimination of the need for noalias in types vector<T> and matrix<T>, when assigned to the same type. This option doesn't have an effect on bounded and c types. Although it is rare, not all compilers support copy elision (that allows for move semantics), so a test must be performed to make sure that there is a benefit when it is enabled. A small demonstration and test can be found in test_move_semantics.cpp

In the test example two tests are defined, one for vectors and one for matrices. The aim of this example is to print the pointers of the storage of each of the containers, before and after the assignment to a temporary object. When move semantics are enabled, the vector<T> and matrix<T> storage is moved from the temporary and no copy is performed.

If move semantics are supported by your compiler you will get an output like the following:

```
matrix<double> ----------------------------------------------------------------
Temporary pointer r: 0x94790c0
Pointer (must be equal to temp. pointer if move semantics are enabled) : 0x94790c0
```

Notes:

- It should be no surprise to see matrices and vectors been passed by VALUE, the compiler takes care and either moves (if the underlying code does not modify the object), or copies (if the underlying code modifies the object).
- There might be some space for some improvements (like clearing the data, before swaping)
- Move semantics don't eliminate temporaries. They rather move their storage around so no copies are performed.
- MSVC does no implement Named Return Value Optimization in debug mode. So if you build in debug with this compiler you might get different behaviour than a release build.
- Enabling move semantics is done via #define BOOST_UBLAS_MOVE_SEMANTICS.
- There is plenty of room for optimizations when c++0x standard is out, taking advantage of rvalue references. (I have a sweet vector implementation using that).
- If you enable move semantics and your compiler does not support them, the operation will just be as passing by const reference.

Interesting links

- Want Speed? Pass by Value.
- Rvalue References: C++0x Features in VC10, Part 2
- Move It With Rvalue References

BOOST_UBLAS_CHECK_ENABLE

When BOOST_UBLAS_CHECK_ENABLE is defined then all index and parameter checks are enabled. This is enabled in debug mode and disabled in release mode.

BOOST_UBLAS_TYPE_CHECK

When BOOST_UBLAS_TYPE_CHECK is enabled then all possibly expensive structure checks are enabled. If this is not desireable then use #define BOOST_UBLAS_TYPE_CHECK 0 before including any uBLAS header. The define BOOST_UBLAS_TYPE_CHECK_EPSILON can be used to control the acceptable tolerance, see detail/matrix_assign.hpp for implementation details of this check.

BOOST_UBLAS_USE_LONG_DOUBLE

Enable uBLAS expressions that involve containers of 'long double'

BOOST_UBLAS_USE_INTERVAL

Enable uBLAS expressions that involve containers of 'boost::numeric::interval' types

Configuring uBLAS with Macros

Many macro's appear in ublas/config.hpp and elsewhere. Hopefully in the future some of these will disappear! They fall into 4 groups:

- Automatically set by 'boost/numeric/ublas/config.hpp' based on NDEBUG. Makes the distinction between debug (safe) and release (fast) mode. Similar to STLport
  - *Release* mode (NDEBUG defined)
    - BOOST_UBLAS_INLINE *Compiler dependant definition to control function inlining.*
    - BOOST_UBLAS_USE_FAST_SAME
  - *Debug* mode
    - BOOST_UBLAS_CHECK_ENABLE *Enable checking of indexs, iterators and parameters. Prevents out of bound access etc.*
    - BOOST_UBLAS_TYPE_CHECK *Enable additional checks for the results of expressions using non dense types. Picks up runtime error such as the assignment of a numerically non-symmetric matrix to symmertic_matrix. Use #define BOOST_UBLAS_TYPE_CHECK 0 to disable expensive numeric type checks.* (Note: "structure check" would be a much better name.)
    - BOOST_UBLAS_TYPE_CHECK_EPSILON *default: sqrt(epsilon), controls how large the difference between the expected result and the computed result may become. Increase this value if you are going to use near singular or badly scaled matrices. Please, refer to detail/matrix_assign.hpp for implementation of these type checks.*
- Automatically set by 'boost/numeric/ublas/config.hpp' based on compiler and boost/config.hpp macro's. Augments the compiler deficiency workarounds already supplied by boost/config.hpp
  - BOOST_UBLAS_NO_NESTED_CLASS_RELATION *A particularly nasty problem with VC7.1 Requires that uBLAS and the user use begin(it) rather then it.begin()*
  - BOOST_UBLAS_NO_SMART_PROXIES *Disable the automatic propagation of 'constantness' to proxies. Smart proxies automatically determine if the underling container they reference is constant or not. They adjust there definition of iterators and container access to reflect this constantness.*

- For use by uBLAS authors to test implementation methods. Preset in config.hpp
  - BOOST_UBLAS_USE_INVARIANT_HOISTING
  - BOOST_UBLAS_USE_INDEXING
  - BOOST_UBLAS_USE_INDEXED_ITERATOR
  - BOOST_UBLAS_NON_CONFORMANT_PROXIES *Gappy containers may be non-conformant, that is contain elements at different indices. Assigning between proxies (vector ranges for example) of these containers is difficult as the LHS may need insert new elements. This is slow.*
  - BOOST_UBLAS_USE_DUFF_DEVICE *Near useless on all platforms (see GCC's -funroll-loops)*
- User options. Can be predefined by user before including any uBLAS headers. They may also be automatically defined for some compilers to work around compile bugs.
  - BOOST_UBLAS_USE_LONG_DOUBLE *Enable uBLAS expressions that involve containers of 'long double'*
  - BOOST_UBLAS_USE_INTERVAL *Enable uBLAS expressions that involve containers of 'boost::numeric::interval' types*
  - BOOST_UBLAS_SIMPLE_ET_DEBUG *In order to simplify debugging is is possible to simplify expression templateso they are restricted to a single operation*
  - BOOST_UBLAS_ENABLE_PROXY_SHORTCUTS *enable automatic conversion from proxy class to matrix expression*
  - BOOST_UBLAS_NO_ELEMENT_PROXIES *Disables the use of element proxies for gappy types.*
  - *The Gappy types (sparse, coordinate, compressed) store non-zero elements in their own containers. When new non-zero elements are assigned they must rearrange these containers. This invalidates references, iterators or pointers to these elements. This can happen at some surprising times such as the expression "a [1] = a [0] = 1;". Element proxies guarantee all such expressions will work as expected. However they bring their own restrictions and efficiency problems. For example as of Boost 1.30.0 they prevent the assignment of elements between different types.*
  - BOOST_UBLAS_REFERENCE_CONST_MEMBER *Enable to allow refernces to be returned to fixed (zero or one) elements of triangular or banded matrices*
  - BOOST_UBLAS_NO_EXCEPTIONS *Disable the use exceptions of uBLAS internal checks and error conditions. BOOST_NO_EXCEPTIONS has same effect.*
  - BOOST_UBLAS_SINGULAR_CHECK *Check the for singularity in triangular solve() functions*

Last modified: Wed Sep 16 23:16:45 CEST 2009

Sparse Storage

Default Standard Map

Description

The templated class `map_std<I, T, ALLOC>` provides a wrapper for the standard library associative container `std::map`. The wrapper has one simple purpose. It allows the definition of a default template parameter (for the adapted array) when declaring the sparse container types.

Example

```
#include <boost/numeric/ublas/storage_sparse.hpp>

int main () {
    using namespace boost::numeric::ublas;
    map_std<int, double> a (3);
    for (unsigned i = 0; i < a.size (); ++ i) {
        a [i] = i;
        std::cout << a [i] << std::endl;
    }
}
```

Definition

Defined in the header storage_sparse.hpp.

Template parameters

Parameter	Description	Default
`I`	The type of index stored in the array.	
`T`	The type of object stored in the array.	
`ALLOC`	An STL Allocator	std::allocator

Model of

Reversible Container.

Type requirements

None, except for those imposed by the requirements of Reversible Container.

Public base classes

std::map

Map Array

Description

The templated class `map_array<I, T, ALLOC>` implements a `std::map` like associative container as a sorted array. It therefore some of the Associative Container interface without having the same semantics as an std::map.

At any time the `map_array` has a capacity up to which new element can be inserted. If `insert` would cause the size of the `map_array` to exceeds its capactity then it is **reallocated**. Iterators and reference are invalidated. The capacity can be directly control using the `reserve` member function.

Example

```
#include <boost/numeric/ublas/storage_sparse.hpp>

int main () {
    using namespace boost::numeric::ublas;
    map_array<int, double> a (3);
    for (unsigned i = 0; i < a.size (); ++ i) {
        a [i] = i;
        std::cout << a [i] << std::endl;
    }
}
```

Definition

Defined in the header storage_sparse.hpp.

Template parameters

Parameter	Description	Default
I	The type of index stored in the array.	
T	The type of object stored in the array.	
ALLOC	An STL Allocator	std::allocator

Model of

Reversible Container.

Type requirements

None, except for those imposed by the requirements of Reversible Container.

Public base classes

None.

Members

Member	Description
map_array (ALLOC &a = ALLOC())	Allocates a map_array that holds at most zero elements.
map_array (const map_array &c)	The copy constructor.
~map_array ()	Deallocates the map_array itself.

Member	Description
`void reserve (size_type capacity)`	Changes the `map_array` capacity. It can hold at most `capacity` elements without reallocation. The capacity can be reduced such that `capacity >= size()`. The content of the `map_array` is preserved.
`size_type size () const`	Returns the size of the `map_array`.
`size_type size () const`	Returns the capacity of the `map_array`.
`data_reference operator [] (index_type i)`	Returns a reference of the element that is associated with a particular index. If the `map_array` does not already contain such an element, `operator[]` inserts the default `T ()`.
`map_array &operator = (const map_array &a)`	The assignment operator.
`map_array &assign_temporary (map_array &a)`	Assigns a temporary. May change the array `a`.
`void swap (map_array &a)`	Swaps the contents of the arrays.
`std::pair insert (const value_type &p)`	Inserts `p` into the array. The second part of the return value is `true` if `p` was inserted and `false` if was not inserted because it was aleady present.
`iterator insert (iterator it, const value_type &p)`	Inserts `p` into the array, using `it` as a hint to where it will be inserted.
`void erase (iterator it)`	Erases the value at `it`.
`void clear ()`	Clears the array.
`const_iterator find (index_type i) const`	Finds an element whose index is `i`.
`iterator find (index_type i)`	Finds an element whose index is `i`.
`const_iterator lower_bound (index_type i) const`	Finds the first element whose index is not less than `i` .
`iterator lower_bound (index_type i)`	Finds the first element whose index is not less than `i` .
`const_iterator upper_bound (index_type i) const`	Finds the first element whose index is greater than `i` .
`iterator upper_bound (index_type i)`	Finds the first element whose index is greater than `i` .
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `map_array`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `map_array`.
`iterator begin ()`	Returns a `iterator` pointing to the beginning of the `map_array`.

Member	Description
`iterator end ()`	Returns a `iterator` pointing to the end of the `map_array`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `map_array`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `map_array`.
`reverse_iterator rbegin ()`	Returns a `reverse_iterator` pointing to the beginning of the reversed `map_array`.
`reverse_iterator rend ()`	Returns a `reverse_iterator` pointing to the end of the reversed `map_array`.

Range and Slice Storage

Range<SizeType,DistanceType>

Description

The class `range` specifies a range of indicies. The range is a sequence of indices from a start value to stop value. The indices increase by one and exlude the stop value. `range` can therefore be used to specify ranges of elements from vectors and matrices.

Example

```
#include <boost/numeric/ublas/storage.hpp>

int main () {
    using namespace boost::numeric::ublas;
    range r (0, 3);
    for (unsigned i = 0; i < r.size (); ++ i) {
        std::cout << r (i) << std::endl;
    }
}
```

Definition

Defined in the header storage.hpp.

Model of

Reversible Container.

Type requirements

None, except for those imposed by the requirements of Reversible Container.

Public base classes

None.

Members

Member	Description
`range (size_type start, size_type stop)`	Constructs a range of indicies from `start` to `stop (excluded)` .
`size_type start () const`	Returns the beginning of the `range`.
`size_type size () const`	Returns the size of the `range`.
`const_reference operator [] (size_type i) const`	Returns the value `start + i` of the `i` -th element.
`range compose (const range &r) const`	Returns the composite range from `start + r.start ()` to `start + r.start () + r.size ()`.
`bool operator == (const range &r) const`	Tests two ranges for equality.
`bool operator != (const range &r) const`	Tests two ranges for inequality.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `range`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `range`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `range`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `range`.

Preconditions

- `start () < = stop ()`

Slice<SizeType,DistanceType>

Description

The class `slice` specifies a 'slice' of indicies. Slices are more general then ranges, the stride allows the sequence of indicies to increase and decrease by the specified amount between element. `slice` can therefore be used to specify slices of element from vectors and matrices.

Example

```
#include <boost/numeric/ublas/storage.hpp>

int main () {
    using namespace boost::numeric::ublas;
    slice s (0, 1, 3);
    for (unsigned i = 0; i < s.size (); ++ i) {
        std::cout << s (i) << std::endl;
    }
}
```

Definition

Defined in the header storage.hpp.

Model of

Reversible Container.

Type requirements

None, except for those imposed by the requirements of Reversible Container.

Public base classes

None.

Members

Member	Description
`slice (size_type start, size_type stride, size_type size)`	Constructs a slice `start,start+stride,start+2*stride···` with `size` elements.
`size_type start () const`	Returns the beginning of the `slice`.
`size_type stride () const`	Returns the stride of the `slice`.
`size_type size () const`	Returns the size of the `slice`.
`const_reference operator [] (size_type i) const`	Returns the value `start + i * stride` of the `i`-th element.
`slice compose (const range &r) const`	Returns the composite slice from `start + stride * r.start ()` to `start + stride * (r.start () + r.size ())` with stride `stride`.
`slice compose (const slice &s) const`	Returns the composite slice from `start + stride * s.start ()` to `start + stride * s.stride () * (s.start () + s.size ())` with stride `stride * s.stride ()` .
`bool operator == (const slice &s) const`	Tests two slices for equality.

Member	Description
`bool operator != (const slice &s) const`	Tests two slices for inequality.
`const_iterator begin () const`	Returns a `const_iterator` pointing to the beginning of the `slice`.
`const_iterator end () const`	Returns a `const_iterator` pointing to the end of the `slice`.
`const_reverse_iterator rbegin () const`	Returns a `const_reverse_iterator` pointing to the beginning of the reversed `slice`.
`const_reverse_iterator rend () const`	Returns a `const_reverse_iterator` pointing to the end of the reversed `slice`.

Preconditions

- None all strides are vaild. However when an index is returned or an iterator is dereferenced its value must be representable as the size_type.

Special Products

Functions

template<class V, class E1, class E2> BOOST_UBLAS_INLINE V &	axpy_prod (const matrix_expression< E1 > &e1, const vector_expression< E2 > &e2, V &v, bool init=true) computes `v += A x` or `v = A x` in an optimized fashion.
template<class V, class E1, class E2> BOOST_UBLAS_INLINE V &	axpy_prod (const vector_expression< E1 > &e1, const matrix_expression< E2 > &e2, V &v, bool init=true) computes `v += AT x` or `v = AT x` in an optimized fashion.
template<class M, class E1, class E2> BOOST_UBLAS_INLINE M &	axpy_prod (const matrix_expression< E1 > &e1, const matrix_expression< E2 > &e2, M &m, bool init=true) computes `M += A X` or `M = A X` in an optimized fashion.
template<class M, class E1, class E2> BOOST_UBLAS_INLINE M &	opb_prod (const matrix_expression< E1 > &e1, const matrix_expression< E2 > &e2, M &m, bool init=true) computes `M += A X` or `M = A X` in an optimized fashion.

1.

BOOST_UBLAS_INLINE V& axpy_prod	(	const matrix_expression< E1 > &	*e1*,
		const vector_expression< E2 > &	*e2*,
		V &	*v*,
		bool	*init* = `true`
	)		

	computes `v += A x` or `v = A x` in an optimized fashion. **Parameters:** *e1* the matrix expression `A`, *e2* the vector expression `x`, *v* the result vector `v`, *init* a boolean parameter `axpy_prod(A, x, v, init)` implements the well known axpy-product. Setting *init* to `true` is equivalent to call `v.clear()` before `axpy_prod`. Currently *init* defaults to `true`, but this may change in the future. Up to now there are some specialisation for compressed matrices that give a large speed up compared to prod.

2.

BOOST_UBLAS_INLINE V& axpy_prod	(	const vector_expression< E1 > &	*e1*,
		const matrix_expression< E2 > &	*e2*,
		V &	*v*,
		bool	*init* = `true`
	)		

	computes `v += AT x` or `v = AT x` in an optimized fashion. **Parameters:** *e1* the vector expression `x`, *e2* the matrix expression `A`, *v* the result vector `v`, *init* a boolean parameter `axpy_prod(x, A, v, init)` implements the well known axpy-product. Setting *init* to `true` is equivalent to call `v.clear()` before `axpy_prod`. Currently *init* defaults to `true`, but this may change in the future. Up to now there are some specialisation for compressed matrices that give a large speed up compared to prod.

3.

BOOST_UBLAS_INLINE M& axpy_prod	(	const matrix_expression< E1 > &	*e1*,
		const matrix_expression< E2 > &	*e2*,
		M &	*m*,
		bool	*init* = `true`
	)		

	computes `M += A X` or `M = A X` in an optimized fashion. **Parameters:** *e1* !the matrix expression `A`, *e2* !the matrix expression `X`, *m* !the result matrix `M`, *init* !a boolean parameter `axpy_prod(A, X, M, init)` implements the well known axpy-product. Setting *init* to `true` is equivalent to call `M.clear()` before `axpy_prod`. Currently *init* defaults to `true`, but this may change in the future. Up to now there are no specialisations.

4.

BOOST_UBLAS_INLINE M& opb_prod	(	const matrix_expression< E1 > &	*e1*,
		const matrix_expression< E2 > &	*e2*,
		M &	*m*,
		bool	*init* = `true`
	)		

	computes `M += A X` or `M = A X` in an optimized fashion. **Parameters:** *e1* the matrix expression `A` *e2* the matrix expression `X` *m* the result matrix `M` *init* a boolean parameter `opb_prod(A, X, M, init)` implements the well known axpy-product. Setting *init* to `true` is equivalent to call `M.clear()` before `opb_prod`. Currently *init* defaults to `true`, but this may change in the future. This function may give a speedup if `A` has less columns than rows, because the product is computed as a sum of outer products.

Level 3 BLAS

Functions

template<class M1, class T, class M2, class M3> M1 & boost::numeric::ublas::blas_3::tmm (M1 &m1, const T &t, const M2 &m2, const M3 &m3) triangular matrix multiplication
template<class M1, class T, class M2, class C> M1 & boost::numeric::ublas::blas_3::tsm (M1 &m1, const T &t, const M2 &m2, C) triangular solve $m2 * x = t * m1$ in place, *m2* is a triangular matrix
template<class M1, class T1, class T2, class M2, class M3> M1 & boost::numeric::ublas::blas_3::gmm (M1 &m1, const T1 &t1, const T2 &t2, const M2 &m2, const M3 &m3) general matrix multiplication
template<class M1, class T1, class T2, class M2> M1 & boost::numeric::ublas::blas_3::srk (M1 &m1, const T1 &t1, const T2 &t2, const M2 &m2) symmetric rank k update: $m1 = t * m1 + t2 * (m2 * m2^T)$
template<class M1, class T1, class T2, class M2> M1 & boost::numeric::ublas::blas_3::hrk (M1 &m1, const T1 &t1, const T2 &t2, const M2 &m2) hermitian rank k update: $m1 = t * m1 + t2 * (m2 * m2^H)$
template<class M1, class T1, class T2, class M2, class M3> M1 & boost::numeric::ublas::blas_3::sr2k (M1 &m1, const T1 &t1, const T2 &t2, const M2 &m2, const M3 &m3) generalized symmetric rank k update: $m1 = t1 * m1 + t2 * (m2 * m3^T) + t2 * (m3 * m2^T)$
template<class M1, class T1, class T2, class M2, class M3> M1 & boost::numeric::ublas::blas_3::hr2k (M1 &m1, const T1 &t1, const T2 &t2, const M2 &m2, const M3 &m3) generalized hermitian rank k update: $m1 = t1 * m1 + t2 * (m2 * m3^H) + (m3 * (t2 * m2)^H)$
template<class M, class E1, class E2> BOOST_UBLAS_INLINE M & boost::numeric::ublas::axpy_prod (const matrix_expression< E1 > &e1, const matrix_expression< E2 > &e2, M &m, bool init=true) computes `M += A X` or `M = A X` in an optimized fashion.

template<class M, class E1, class E2> BOOST_UBLAS_INLINE M & boost::numeric::ublas::opb_prod (const matrix_expression< E1 > &e1, const matrix_expression< E2 > &e2, M &m, bool init=true) computes `M += A X` or `M = A X` in an optimized fashion.

Function Documentation

1.

M1& tmm	(	M1 &	*m1*,
		const T &	*t*,
		const M2 &	*m2*,
		const M3 &	*m3*
	)		

	triangular matrix multiplication

2.

M1& tsm	(	M1 &	*m1*,
		const T &	*t*,
		const M2 &	*m2*,
		C	
	)		

	triangular solve *m2* * *x* = *t* * *m1* in place, *m2* is a triangular matrix

3.

M1& gmm	(	M1 &	*m1*,
		const T1 &	*t1*,
		const T2 &	*t2*,
		const M2 &	*m2*,
		const M3 &	*m3*
	)		

	general matrix multiplication

4.

M1& srk	(	M1 &	*m1*,
		const T1 &	*t1*,
		const T2 &	*t2*,
		const M2 &	*m2*
	)		

	symmetric rank k update: $m1 = t * m1 + t2 * (m2 * m2^T)$ **Todo:** use opb_prod()

5.

M1& hrk	(	M1 &	*m1*,
		const T1 &	*t1*,
		const T2 &	*t2*,
		const M2 &	*m2*
	)		

	hermitian rank k update: $m1 = t * m1 + t2 * (m2 * m2^H)$ **Todo:** use opb_prod()

6.

M1& sr2k	(	M1 &	*m1*,
		const T1 &	*t1*,
		const T2 &	*t2*,
		const M2 &	*m2*,
		const M3 &	*m3*
	)		

	generalized symmetric rank k update: $m1 = t1 * m1 + t2 * (m2 * m3^T) + t2 * (m3 * m2^T)$ **Todo:** use opb_prod()

7.

M1& hr2k	(	M1 &	*m1*,
		const T1 &	*t1*,
		const T2 &	*t2*,
		const M2 &	*m2*,
		const M3 &	*m3*
	)		

	generalized hermitian rank k update: $m1 = t1 * m1 + t2 * (m2 * m3^H) + (m3 * (t2 * m2)^H)$ **Todo:** use opb_prod()

Container Concepts

Vector

Description

A Vector describes common aspects of dense, packed and sparse vectors.

Refinement of

DefaultConstructible, Vector Expression [1].

Associated types

In addition to the types defined by Vector Expression

Public base	vector_container<V>	V must be derived from this public base type.
Storage array	V::array_type	Dense Vector ONLY. The type of underlying storage array used to store the elements. The array_type must model the **Storage** concept.

Notation

`V`	A type that is a model of Vector
`v`	Objects of type `V`
`n, i`	Objects of a type convertible to `size_type`
`t`	Object of a type convertible to `value_type`
`p`	Object of a type convertible to `bool`

Definitions

Valid expressions

In addition to the expressions defined in DefaultConstructible, Vector Expression the following expressions must be valid.

Name	Expression	Type requirements	Return type
Sizing constructor	`V v (n)`		`V`
Insert	`v.insert_element (i, t)`	`v` is mutable.	`void`
Erase	`v.erase_element (i)`	`v` is mutable.	`void`
Clear	`v.clear ()`	`v` is mutable.	`void`
Resize	`v.resize (n)` `v.resize (n, p)`	`v` is mutable.	`void`
Storage	`v.data()`	`v` is mutable and Dense.	`array_type&` if `v` is mutable, `const array_type&` otherwise

Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in Vector Expression .

Name	Expression	Precondition	Semantics	Postcondition
Sizing constructor	`V v (n)`	`n >= 0`	Allocates a vector of `n` elements.	`v.size () == n`.
Element access [2]	`v[n]`	`0<n>v.size()`	returns the n-th element in v	
Insert	`v.insert_element (i, t)`	`0 < = i < v.size ()`.	Inserts an element at `v (i)` with value `t`. The storage requirement of the Vector may be increased.	`v (i)` is equal to `t`.

Name	Expression	Precondition	Semantics	Postcondition
Erase	`v.erase_element (i)`	`0 < = i < v.size ()`	Destroys the element as `v (i)` and replaces it with the default `value_type ()`. The storage requirement of the Vector may be decreased.	`v (i)` is equal to `value_type ()`.
Clear	`v.clear ()`		Equivalent to `for (i = 0; i < v.size (); ++ i) v.erase_element (i);`	
Resize	`v.resize (n)` `v.resize (n, p)`		Reallocates the vector so that it can hold `n` elements. Erases or appends elements in order to bring the vector to the prescribed size. Appended elements copies of `value_type()`. When `p == false` then existing elements are not preserved and elements will not appended as normal. Instead the vector is in the same state as that after an equivalent sizing constructor.	`v.size () == n`.
Storage	`v.data()`		Returns a reference to the underlying dense storage.	

Complexity guarantees

The run-time complexity of the sizing constructor is linear in the vector's size.

The run-time complexity of insert_element and erase_element is specific for the Vector model and it depends on increases/decreases in storage requirements.

The run-time complexity of resize is linear in the vector's size.

Invariants

Models

- `vector`, `bounded_vector`, `c_vector`
- `unit_vector`, `zero_vector`, `scalar_vector`
- `mapped_vector;`, `compressed_vector`, `coordinate_vector`

Notes

[1] As a user you need not care about Vector being a refinement of the VectorExpression. Being a refinement of the VectorExpression is only important for the template-expression engine but not the user.

[2] The `operator[]` is added purely for convenience and compatibility with the `std::vector`. In uBLAS however, generally `operator()` is used for indexing because this can be used for both vectors and matrices.

Matrix

Description

A Matrix describes common aspects of dense, packed and sparse matrices.

Refinement of

DefaultConstructible, Matrix Expression [1] .

Associated types

In addition to the types defined by Matrix Expression

Public base	matrix_container<M>	M must be derived from this public base type.
Storage array	M::array_type	Dense Matrix ONLY. The type of underlying storage array used to store the elements. The array_type must model the **Storage** concept.

Notation

`M`	A type that is a model of Matrix

`m`	Objects of type `M`
`n1, n2, i, j`	Objects of a type convertible to `size_type`
`t`	Object of a type convertible to `value_type`
`p`	Object of a type convertible to `bool`

Definitions

Valid expressions

In addition to the expressions defined in Matrix Expression the following expressions must be valid.

Name	Expression	Type requirements	Return type
Sizing constructor	`M m (n1, n2)`		`M`
Insert	`m.insert_element (i, j, t)`	`m` is mutable.	`void`
Erase	`m.erase_element (i, j)`	`m` is mutable.	`void`
Clear	`m.clear ()`	`m` is mutable.	`void`
Resize	`m.resize (n1, n2)` `m.resize (n1, n2, p)`	`m` is mutable.	`void`
Storage	`m.data()`	`m` is mutable and Dense.	`array_type&` if `m` is mutable, `const array_type&` otherwise

Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in Matrix Expression .

Name	Expression	Precondition	Semantics	Postcondition
Sizing constructor	`M m (n1, n2)`	`n1 >= 0` and `n2 >= 0`	Allocates a matrix of `n1` rows and `n2` columns.	`m.size1 () == n1` and `m.size2 () == n2`.
Insert	`m.insert_element (i, j, t)`	`0 < = i < m.size1 ()`, + `0 < = j < m.size2 ()`.	Inserts an element at `m (i, j)` with value `t`. The storage requirement of the Matrix may be increased. `m (i, j)` is equal to `t`.	`m(i,j)` is equal to `t`.

Name	Expression	Precondition	Semantics	Postcondition
Erase	`m.erase_element (i, j)`	`0 <= i < m.size1 ()`and ` 0 <= j < m.size2`	Destroys the element as `m (i, j)` and replaces it with the default `value_type ()`. The storage requirement of the Matrix may be decreased.	`m (i, j)` is equal to `value_type ()`.

Clear	`m.clear ()`		Equivalent to
			`for (i = 0; i < m.size1 (); i) + for (j = 0; j < m.size2 (); j)`
			
			`m.erase_element (i, j);`	

Resize	`m.resize (n1, n2)` `m.resize (n1, n2, p)`		Reallocate the matrix so that it can hold `n1` rows and `n2` columns. Erases or appends elements in order to bring the matrix to the prescribed size. Appended elements are `value_type()` copies. When `p == false` then existing elements are not preserved and elements will not appended as normal. Instead the matrix is in the same state as that after an equivalent sizing constructor.	`m.size1 () == n1` and `m.size2 () == n2`.
Storage	`m.data()`		Returns a reference to the underlying dense storage.	

Complexity guarantees

The run-time complexity of the sizing constructor is quadratic in the matrix's size.

The run-time complexity of insert_element and erase_element is specific for the Matrix model and it depends on increases/decreases in storage requirements.

The run-time complexity of resize is quadratic in the matrix's size.

Invariants

Models

- `matrix`, `bounded_matrix`, `c_matrix`
- `identity_matrix` , `zero_matrix` , `scalar_matrix`
- `triangular_matrix` , `symmetric_matrix` , `banded_matrix`
- `mapped_matrix` , `compressed_matrix` , `coordinate_matrix`

Notes

[1] As a user you need not care about Matrix being a refinement of the MatrixExpression. Being a refinement of the MatrixExpression is only important for the template-expression engine but not the user.

Tensor

Description

A Tensor describes common aspects of dense multidimensional arrays.

Refinement of

DefaultConstructible, Tensor Expression [1] .

Associated types

In addition to the types defined by Tensor Expression

Public base	`tensor_container<tensor_t>`	`tensor_t` must be derived from this public base type.
Storage array	`tensor_t::array_type`	Dense tensor ONLY. The type of underlying storage array used to store the elements. The array_type must model the **Storage** concept.

Notation

`tensor_t`	**A type that is a model of Tensor**
`t`	Objects of type `tensor_t`
`n1, n2, np, m1, m2, mq `	Dimension objects of a type convertible to `size_type`
`i1, i2, ip, j, k `	Index objects of a type convertible to `size_type`
`v`	Object of a type convertible to `value_type`

Definitions

Valid expressions

In addition to the expressions defined in Tensor Expression the following expressions must be valid.

Name	**Expression**	**Type requirements**	**Return type**
Sizing constructor	`T t(n1, n2, ⋯, np)`		`T`
Write	`t.at(i1, i2, ⋯, ip)`	`t` is mutable.	`void`
Read	`t.at(i1, i2, ⋯, ip)`	`t` is mutable.	`v`
Clear	`t.clear ()`	`t` is mutable.	`void`
Resize	`t.resize(m1, m2, ⋯ , mq)`	`t` is mutable.	`void`
Storage	`t.data()`	`t` is mutable and dense.	`pointer` if `t` is mutable, `const_pointer` otherwise

Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in Tensor Expression .

Name	**Expression**	**Precondition**	**Semantics**	**Postcondition**
Sizing constructor	`T t(n1, n2, ⋯, np)`	n_r > =1 for 1< = 1 < = p	Allocates a `p` -order tensor with dimension extents n_1, n_2, ⋯. n_p.	`t.size(r)==nr` for 1 < = r < = p.
Write	`t.at(i1,i2,⋯ ,ip)=v`	0 < = i_r < n_r for 1 < = r < = p.	Writes an element at multi-index position i_1, i_2, ⋯ i_p with value `v`.	`t(i1,i2,⋯,ip)` is equal to `v`.

Name	Expression	Precondition	Semantics	Postcondition
Read	`v=t.at(i1,i2,⋯,ip)`	$0 <= i_r < n_r$ for $1 <= r <= p$.	Reads the element at multi-index position $i_1, i_2, \cdots i_p$ and returns a value `v`.	`t(i1,i2,⋯,ip)` is equal to `v`.
Clear	`t.clear()`		Removes all elements from the container.	
Resize	`t.resize(m1, m2, ⋯, mq)`	$m_r >= 1$ for $1 <= 1 <= q$	Reallocate the matrix so that it can hold $m_1 \times m_2 \times \cdots m_q$ elements. + Erases or appends elements in order to bring the matrix to the prescribed size. Appended elements are `value_type()` copies.	`t.size(r) == mr` for $1 <= r <= q$.
Storage	`m.data()`		Returns a reference to the underlying dense storage.	

Complexity guarantees

The run-time complexity of contructor is linear in the tensor's size $n_1 \times n_2 \times \cdots n_p$.

The run-time complexity of `write()` and `read()` is linear in the order of the tensor.

The run-time complexity of resize is at most linear in the tensor's size $m_1 \times m_2 \times \cdots n_q$.

Invariants

Models

- `tensor`

Notes

[1] As a user you need not care about Tensor being a refinement of the TensorExpression. Being a refinement of the TensorExpression is only important for the template-expression engine but not the user.

Expression Concepts

Scalar Expression

Description

A Scalar Expression is an expression convertible to a scalar type.

Refinement of

Default Constructible.

Associated types

Public base	scaler_expression<S>	S must be derived from this public base type.
Value type	`value_type`	The type of the scalar expression.

Notation

`S`	A type that is a model of Scalar Expression

Definitions

Valid expressions

In addition to the expressions defined in Default Constructible the following expressions must be valid.

Name	Expression	Type requirements	Return type
Evaluation	`operator value_type () const`		`value_type`

Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in Default Constructible.

Name	Expression	Precondition	Semantics	Postcondition
Evaluation	`operator value_type () const`		Evaluates the scalar expression.	

Complexity guarantees

The run-time complexity of the evaluation is specific for the evaluated scalar expression.

Invariants

Models

- `vector_scalar_unary`
- `vector_scalar_binary`

Vector Expression

Description

A Vector Expression is an expression evaluatable to a vector. Vector Expression provides an Indexed Bidirectional Iterator or an Indexed Random Access Iterator .

Refinement of

Default Constructible.

Associated types

Public base	vector_expression<V>	V must be derived from this public base type.
Value type	`value_type`	The element type of the vector expression.
Reference type	`reference`	The return type when accessing an element of a vector expression. Convertable to a `value_type`.
Const reference type	`const_reference`	The return type when accessing an element of a constant vector expression. Convertable to a `value_type`.
Size type	`size_type`	The index type of the vector expression. Am unsigned integral type used to represent size and index values. Can represent any nonnegative value of `difference_type`.
Distance type	`difference_type`	A signed integral type used to represent the distance between two of the vector expression's iterators.
Const iterator type	`const_iterator`	A type of iterator that may be used to examine a vector expression's elements.
Iterator type	`iterator`	A type of iterator that may be used to modify a vector expression's elements.

Const reverse iterator type	`const_reverse_iterator`	A Reverse Iterator adaptor whose base iterator type is the vector expression's const iterator type.
Reverse iterator type	`reverse_iterator`	A Reverse Iterator adaptor whose base iterator type is the vector expression's iterator type.

Notation

`V`	A type that is a model of Vector Expression
`v, v1, v2`	Object of type `V`
`i`	Object of a type convertible to `size_type`
`t`	Object of a type convertible to `value_type`

Definitions

Valid expressions

In addition to the expressions defined in Default Constructible the following expressions must be valid.

Name	Expression	Type requirements	Return type
Beginning of range	`v.begin ()`		`const_iterator`
	`v.begin ()`	`v` is mutable.	`iterator`
End of range	`v.end ()`		`const_iterator`
	`v.end ()`	`v` is mutable.	`iterator`
Size	`v.size ()`		`size_type`
Swap	`v1.swap (v2)`	`v1` and `v2` are mutable.	`void`
Beginning of reverse range	`v.rbegin ()`		`const_reverse_iterator`
	`v.rbegin ()`	`v` is mutable.	`reverse_iterator`
End of reverse range	`v.rend ()`		`const_reverse_iterator`
	`v.rend ()`	`v` is mutable.	`reverse_iterator`
Element access	`v (i)`	`i` is convertible to `size_type`.	Convertible to `value_type`.
Assignment	`v2 = v1`	`v2` is mutable and `v1` is convertible to `V`.	`V &`
	`v2.assign (v1)`	`v2` is mutable and `v1` is convertible to `V`.	`V &`

Name	Expression	Type requirements	Return type
Computed assignment	`v2 += v1`	`v2` is mutable and `v1` is convertible to `V`.	`V &`
	`v2.plus_assign (v1)`	`v2` is mutable and `v1` is convertible to `V`.	`V &`
	`v2 -= v1`	`v2` is mutable and `v1` is convertible to `V`.	`V &`
	`v2.minus_assign (v1)`	`v2` is mutable and `v1` is convertible to `V`.	`V &`
	`v *= t`	`v` is mutable and `t` is convertible to `value_type`.	`V &`

Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in Default Constructible.

Name	Expression	Precondition	Semantics	Postcondition
Beginning of range	`v.begin ()`		Returns an iterator pointing to the first element in the vector expression.	`v.begin ()` is either dereferenceable or past-the-end. It is past-the-end if and only if `v.size () == 0`.
End of range	`v.end ()`		Returns an iterator pointing one past the last element in the vector expression.	`v.end ()` is past-the-end.
Size	`v.size ()`		Returns the size of the vector expression, that is, its number of elements.	`v.size () >= 0`
Swap	`v1.swap (v2)`		Equivalent to `swap (v1, v2)`.	

Name	Expression	Precondition	Semantics	Postcondition
Beginning of reverse range	`v.rbegin ()`		Equivalent to `reverse_iterator (v.end ())`.	`v.rbegin ()` is either dereferenceable or past-the-end. It is past-the-end if and only if `v.size () == 0`.
End of reverse range	`v.rend ()`		Equivalent to `reverse_iterator (v.begin ())`.	`v.rend ()` is past-the-end.
Element access	`v (i)`	`0 ⇐ i < v.size ()`	Returns the `i`-th element of the vector expression.	
Assignment	`v2 = v1`	`v1.size () == v2.size ()`	Assigns every element of the evaluated vector expression `v1` to the corresponding element of `v2` .	
	`v2.assign (v1)`	`v1.size () == v2.size ()`	Assigns every element of `v1` to the corresponding element of `v2`.	
Computed assignment	`v2 += v1`	`v1.size () == v2.size ()`	Adds every element of the evaluated vector expression `v1` to the corresponding element of `v2`.	
	`v2.plus_assign (v1)`	`v1.size () == v2.size ()`	Adds every element of `v1` to the corresponding element of `v2`.	
	`v2 -= v1`	`v1.size () == v2.size ()`	Subtracts every element of the evaluated vector expression `v1` from the corresponding element of `v2` .	

Name	Expression	Precondition	Semantics	Postcondition
	`v2.minus_assign (v1)`	`v1.size () == v2.size ()`	Subtracts every element of `v1` from the corresponding element of `v2`.	
	`v *= t`		Multiplies every element of `v` with `t` .	

Complexity guarantees

The run-time complexity of `begin ()` and `end ()` is specific for the evaluated vector expression, typically amortized constant time.

The run-time complexity of `size ()` is constant time.

The run-time complexity of `swap ()` is specific for the evaluated vector expression, typically constant time.

The run-time complexity of `rbegin ()` and `rend ()` is specific for the evaluated vector expression, typically amortized constant time.

The run-time complexity of the element access is specific for the evaluated vector expression, typically amortized constant time for the dense and logarithmic for the sparse case.

The run-time complexity of the arithmetic operations is specific for the evaluated vector expressions, typically linear in the size of the expressions.

Invariants

Valid range	For any vector expression `v`, `[v.begin (), v.end ())` is a valid range.
Completeness	An algorithm that iterates through the range `[v.begin (), v.end ())` will pass through every element of `v` .
Valid reverse range	`[v.rbegin (), v.rend ())` is a valid range.
Equivalence of ranges	The distance from `v.begin ()` to `v.end ()` is the same as the distance from `v.rbegin ()` to `v.rend ()`.

Models

- `vector_range;`
- `vector_slice`
- `matrix_row`
- `matrix_column`

- `matrix_vector_range`
- `matrix_vector_slice`
- `vector_unary`
- `vector_binary`
- `vector_binary_scalar1`
- `vector_binary_scalar2`
- `matrix_vector_unary1`
- `matrix_vector_unary2`
- `matrix_vector_binary1`
- `matrix_vector_binary2`

Matrix Expression

Description

A Matrix Expression is an expression evaluatable to a matrix. Matrix Expression provides an Indexed Bidirectional Column/Row Iterator or an Indexed Random Access Column/Row Iterator .

Refinement of

Default Constructible.

Associated types

immutable types

Public base	`matrix_expression<M>`	M must be derived from this public base type.
Value type	`value_type`	The element type of the matrix expression.
Const reference type	`const_reference`	The return type when accessing an element of a constant matrix expression. Convertable to a `value_type`.
Size type	`size_type`	The index type of the matrix expression. Am unsigned integral type used to represent size and index values. Can represent any nonnegative value of `difference_type`.
Distance type	`difference_type`	A signed integral type used to represent the distance between two of the matrix expression's iterators.

Const iterator types	`const_iterator1`	A type of column iterator that may be used to examine a matrix expression's elements.
	`const_iterator2`	A type of row iterator that may be used to examine a matrix expression's elements.
Const reverse iterator types	`const_reverse_iterator1`	A Reverse Iterator adaptor whose base iterator type is the matrix expression's const column iterator type.
	`const_reverse_iterator2`	A Reverse Iterator adaptor whose base iterator type is the matrix expression's const row iterator type.

mutable types

Reference type	`reference`	The return type when accessing an element of a matrix expression. Convertable to a `value_type`.
Iterator types	`iterator1`	A type of column iterator that may be used to modify a matrix expression's elements.
	`iterator2`	A type of row iterator that may be used to modify a matrix expression's elements.
Reverse iterator types	`reverse_iterator1`	A Reverse Iterator adaptor whose base iterator type is the matrix expression's column iterator type.
	`reverse_iterator2`	A Reverse Iterator adaptor whose base iterator type is the matrix expression's row iterator type.

Notation

`M`	A type that is a model of Matrix Expression
`m, m1, m2`	Object of type `M`
`i, j`	Objects of a type convertible to `size_type`
`t`	Object of a type convertible to `value_type`

Definitions

Valid expressions

In addition to the expressions defined in Default Constructible the following expressions must be valid.

immutable expressions

Name	Expression	Type requirements	Return type
Size	`m.size1 ()`		`size_type`
	`m.size2 ()`		`size_type`

possibly mutable expressions

Name	Expression	Type requirements	Return type
Beginning of range	`m.begin1 ()`		`const_iterator1`
	`m.begin2 ()`		`const_iterator2`
	`m.begin1 ()`	`m` is mutable.	`iterator1`
	`m.begin2 ()`	`m` is mutable.	`iterator2`
End of range	`m.end1 ()`		`const_iterator1`
	`m.end2 ()`		`const_iterator2`
	`m.end1 ()`	`m` is mutable.	`iterator1`
	`m.end2 ()`	`m` is mutable.	`iterator2`
Swap	`m1.swap (m2)`	`m1` and `m2` are mutable.	`void`
Beginning of reverse range	`m.rbegin1 ()`		`const_reverse_iterator1`
	`m.rbegin2 ()`		`const_reverse_iterator2`
	`m.rbegin1 ()`	`m` is mutable.	`reverse_iterator1`
	`m.rbegin2 ()`	`m` is mutable.	`reverse_iterator2`
End of reverse range	`m.rend1 ()`		`const_reverse_iterator1`
	`m.rend2 ()`		`const_reverse_iterator2`
	`m.rend1 ()`	`m` is mutable.	`reverse_iterator1`
	`m.rend2 ()`	`m` is mutable.	`reverse_iterator2`
Element access	`m (i, j)`	`i` and `j` are convertible to `size_type` .	Convertible to `value_type`.
Assignment	`m2 = m1`	`m2` is mutable and `m1` is convertible to `M`.	`M &`

Name	Expression	Type requirements	Return type
	`m2.assign (m1)`	`m2` is mutable and `m1` is convertible to `M`.	`M &`
Computed assignment	`m2 += m1`	`m2` is mutable and `m1` is convertible to `M`.	`M &`
	`m2.plus_assign (m1)`	`m2` is mutable and `m1` is convertible to `M`.	`M &`
	`m2 -= m1`	`m2` is mutable and `m1` is convertible to `M`.	`M &`
	`m2.minus_assign (m1)`	`m2` is mutable and `m1` is convertible to `M`.	`M &`
	`m *= t`	`m` is mutable and `t` is convertible to `value_type`.	`M &`

Expression semantics

Semantics of an expression is defined only where it differs from, or is not defined in Default Constructible.

Name	Expression	Precondition	Semantics	Postcondition
Beginning of range	`m.begin1 ()`		Returns an iterator pointing to the first element in the first column of a matrix expression.	`m.begin1 ()` is either dereferenceable or past-the-end. It is past-the-end if and only if `m.size1 () == 0`.
	`m.begin2 ()`		Returns an iterator pointing to the first element in the first row of a matrix expression.	`m.begin2 ()` is either dereferenceable or past-the-end. It is past-the-end if and only if `m.size2 () == 0`.
End of range	`m.end1 ()`		Returns an iterator pointing one past the last element in the matrix expression.	`m.end1 ()` is past-the-end.

Name	Expression	Precondition	Semantics	Postcondition
	`m.end2 ()`		Returns an iterator pointing one past the last element in the matrix expression.	`m.end2 ()` is past-the-end.
Size	`m.size1 ()`		Returns the number of rows of the matrix expression.	`m.size1 () >= 0`
	`m.size2 ()`		Returns the number of columns of the matrix expression.	`m.size2 () >= 0`
Swap	`m1.swap (m2)`		Equivalent to `swap (m1, m2)`.	
Beginning of reverse range	`m.rbegin1 ()`		Equivalent to `reverse_iterator1 (m.end1 ())`.	`m.rbegin1 ()` is either dereferenceable or past-the-end. It is past-the-end if and only if `m.size1 () == 0`.
	`m.rbegin2 ()`		Equivalent to `reverse_iterator2 (m.end2 ())`.	`m.rbegin2 ()` is either dereferenceable or past-the-end. It is past-the-end if and only if `m.size2 () == 0`.
End of reverse range	`m.rend1 ()`		Equivalent to `reverse_iterator1 (m.begin1 ())`.	`m.rend1 ()` is past-the-end.
	`m.rend2 ()`		Equivalent to `reverse_iterator2 (m.begin2 ())`.	`m.rend2 ()` is past-the-end.
Element access	`m (i, j)`	`0 ⇐ i < m.size1 ()` and `0 ⇐ j < m.size2 ()`	Returns the `j`-th element of the `i`-th row of the matrix expression.	

Name	Expression	Precondition	Semantics	Postcondition
Assignment	`m2 = m1`	`m1.size1 () == m2.size1 ()` and ` m1.size2 () == m2.size2 ()`	Assigns every element of the evaluated matrix expression `m1` to the corresponding element of `m2` .	
	`m2.assign (m1)`	`m1.size1 () == m2.size1 ()` and ` m1.size2 () == m2.size2 ()`	Assigns every element of `m1` to the corresponding element of `m2`.	
Computed assignment	`m2 += m1`	`m1.size1 () == m2.size1 ()` and ` m1.size2 () == m2.size2 ()`	Adds every element of the evaluated matrix expression `m1` to the corresponding element of `m2`.	
	`m2.plus_assign (m1)`	`m1.size1 () == m2.size1 ()` and ` m1.size2 () == m2.size2 ()`	Adds every element of `m1` to the corresponding element of `m2`.	
	`m2 -= m1`	`m1.size1 () == m2.size1 ()` and ` m1.size2 () == m2.size2 ()`	Subtracts every element of the evaluated matrix expression `m1` from the corresponding element of `m2` .	
	`m2.minus_assign (m1)`	`m1.size1 () == m2.size1 ()` and ` m1.size2 () == m2.size2 ()`	Subtracts every element of `m1` from the corresponding element of `m2`.	
	`m *= t`		Multiplies every element of `m` with `t` .	

Complexity guarantees

The run-time complexity of `begin1 ()`, `begin2 ()` , `end1 ()` and `end2 ()` is specific for the evaluated matrix expression.

The run-time complexity of `size1 ()` and `size2 ()` is constant time.

The run-time complexity of `swap ()` is specific for the evaluated matrix expression, typically constant time.

The run-time complexity of `rbegin1 ()`, `rbegin2 ()` , `rend1 ()` and `rend2 ()` is specific for the evaluated matrix expression.

The run-time complexity of the element access is specific for the evaluated matrix expression, typically amortized constant time for the dense and logarithmic for the sparse case.

The run-time complexity of the arithmetic operations is specific for the evaluated matrix expressions, typically quadratic in the size of the proxies.

Invariants

Valid range	For any matrix expression `m`, `[m.begin1 (), m.end1 ())` and `[m.begin2 (), m.end2 ())` are valid ranges.
Completeness	An algorithm that iterates through the range `[m.begin1 (), m.end1 ())` will pass through every row of `m` , an algorithm that iterates through the range `[m.begin2 (), m.end2 ())` will pass through every column of `m` .
Valid reverse range	`[m.rbegin1 (), m.rend1 ())` and `[m.rbegin2 (), m.rend2 ())` are valid ranges.
Equivalence of ranges	The distance from `m.begin1 ()` to `m.end1 ()` is the same as the distance from `m.rbegin1 ()` to `m.rend1 ()` and the distance from `m.begin2 ()` to `m.end2 ()` is the same as the distance from `m.rbegin2 ()` to `m.rend2 ()`.

Models

- `matrix_range`
- `matrix_slice;`
- `triangular_adaptor`
- `symmetric_adaptor`
- `banded_adaptor`
- `vector_matrix_binary`
- `matrix_unary1`
- `matrix_unary2`
- `matrix_binary`
- `matrix_binary_scalar1`
- `matrix_binary_scalar2`
- `matrix_matrix_binary`

Storage concept

Description

Storage is a variable-size container whose elements are arranged in a strict linear order.

Storage extends the STL Container concept with some STL Sequence-like functionality. The main difference with the Sequence concept however is that the Storage concept does not require default-initialisation of its elements.

Refinement of

Random Access Container and Default Constructible

Associated types

No additional types beyond those defined by Random Access Container

Notation

X	A type that is model of Storage
T	The value_type of X
t	An object of type T
n	object of type convertible to X::size_type

Definitions

Valid expressions

In addition to the expressions defined in Random Access Container, and Default Constructible the following expressions must be valid:

Name	Expression	Type requirements	Return type
Size constructor	X(n)	T is DefaultConstructible	X
Fill constructor	X(n,t)		X
Range constructor	X(i, j)	i and j are Input Iterators whose value type is convertible to T	X
Resize	a.resize(n, t)	a is mutable	void
Resize	a.resize(n)	a is mutable	void

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Default-constructor	X()		Creates 0 elements	size()==0
Size-constructor	X(n)	n>=0	Creates n elements. Elements are constructed without an initializer. That is if T is a (possibly cv-qualified) non-POD class type (or array thereof), the object is default initialized. Otherwise, the object created has indeterminate value. See the sentance "If new initializer is omitted" in section 5.3.4 paragraph 15 of the ISO C++ standard.	size()==n
Fill-constructor	X(n,t)	n>=0	Creates n initialised element with copies of `t`	size()==n
Range constructor	X(i, j)	[i,j) is a valid range.	copies the range [i,j) to the storage	size() is equal to the distance from i to j. Each element is a copy of the corresponding element in the range [i,j).

Name	Expression	Precondition	Semantics	Postcondition
Resize	a.resize(n, t)	n ⇐ a.max_size()	Modified the container so that it has exactly n elements. + The container may be reallocated if its size changes. Existing element values are preserved, additional elements are copies of t.	a.size() == n
Resize	a.resize(n)	n ⇐ a.max_size()	Modified the container so that it has exactly n elements. + The container may be reallocated if its size changes. Element values are uninitialised. That is, each element value may be a previously assigned value or default construced value for T.	a.size() == n

Complexity guarantees

Invariants

Models

- unbounded_array
- bounded_array

Notes

Iterator Concepts

An Iterator is a restricted pointer-like object pointing into a vector or matrix container.

Indexed Bidirectional Iterator

Description

An Indexed Bidirectional Iterator is an iterator of a container that can be dereferenced, incremented, decremented and carries index information.

Refinement of

Assignable, Equality Comparable, Default Constructible.

Associated types

Value type	The type of the value obtained by dereferencing a Indexed Bidirectional Iterator
Container type	The type of the container a Indexed Bidirectional Iterator points into.

Notation

`I`	A type that is a model of Indexed Bidirectional Iterator
`T`	The value type of `I`
`C`	The container type of `I`
`it`, `itt`, `it1`, `it2`	Objects of type `I`
`t`	Object of type `T`
`c`	Object of type `C`

Definitions

A Indexed Bidirectional Iterator may be *mutable*, meaning that the values referred to by objects of that type may be modified, or *constant* , meaning that they may not. If an iterator type is mutable, this implies that its value type is a model of Assignable; the converse, though, is not necessarily true.

A Indexed Bidirectional Iterator may have a *singular* value, meaning that the results of most operations, including comparison for equality, are undefined. The only operation that is guaranteed to be supported is assigning a nonsingular iterator to a singular iterator.

A Indexed Bidirectional Iterator may have a *dereferenceable* value, meaning that dereferencing it yields a well-defined value. Dereferenceable iterators are always nonsingular, but the converse is not true.

An Indexed Bidirectional Iterator is *past-the-end* if it points beyond the last element of a container. Past-the-end values are nonsingular and nondereferenceable.

Valid expressions

In addition to the expressions defined for Assignable, Equality Comparable and Default Constructible, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Default constructor	`I it`		
Dereference	`*it`		Convertible to `T`.
Dereference assignment	`*it = t`	`I` is mutable.	
Member access	`it->m`	`T` is a type for which `t.m` is defined.	
Preincrement	`++ it`		`I &`
Postincrement	`it ++`		`I`
Predecrement	`-- it`		`I &`
Postdecrement	`it --`		`I`
Index	`it.index ()`		`C::size_type`

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in, Assignable, Equality Comparable and Default Constructible.

Name	**Expression**	**Precondition**	**Semantics**	**Postcondition**
Default constructor	`I it`			`it` is singular.
Dereference	`*it`	`it` is dereferenceable.		
Dereference assignment	`*it = t`	Same as for `*it`.		`*it` is a copy of t.
Member access	`it->m`	`it` is dereferenceable.	Equivalent to `(*it).m`	
Preincrement	`++ it`	`it` is dereferenceable.	`it` is modified to point to the next element.	`it` is dereferenceable or past-the-end. ` &it == & it `. + If `it1 == it2 `, + then ` it1 == ++ it2 `.
Postincrement	`it ++`	Same as for `++ it`.	Equivalent to `{ I itt = it; ++ it; return itt; }`	`it` is dereferenceable or past-the-end.

Name	Expression	Precondition	Semantics	Postcondition
Predecrement	`-- it`	`it` is dereferenceable or past-the-end. There exists a dereferenceable iterator `itt` such that `it == ++ itt`.	`it` is modified to point to the previous element.	`it` is dereferenceable. `&it = &-- it`. If `it1 == it2`, then `-- it1 ==  it2`. If `it2` is dereferenceable and `it1 == ++it2`, then `--it1 == it2`.
Postdecrement	`it --`	Same as for — `it`.	Equivalent to `{  I itt = it;  -- it;  return itt; }`	`it` is dereferenceable.
Index	`it.index ()`	`it` is dereferenceable.	`it.index () >= 0` and `it.index () < it ().size ()`	If `it1 == it2`, then `it1.index () == it2.index ()`. If `it1 == it2`, then `it1.index () < (++ it2).index ()`. If `it1 == it2`, then `it1.index () > (-- it2).index ()`.

Complexity guarantees

The complexity of operations on indexed bidirectional iterators is guaranteed to be amortized constant time.

Invariants

Identity	`it1 == it2` **if and only if** `&*it1 == &*it2`.
Symmetry of increment and decrement	If `it` is dereferenceable, then `it; --it;` is a null operation. Similarly, `-- it; it;` is a null operation.
Relation between iterator index and container element operator	If `it` is dereferenceable, `*it == it () (it.index ())`.

Models

- `sparse_vector::iterator`

Indexed Random Access Iterator

Description

An Indexed Random Access Iterator is an iterator of a container that can be dereferenced, moved forward, moved backward and carries index information.

Refinement of

LessThanComparable, Indexed Bidirectional Iterator .

Associated types

Value type	The type of the value obtained by dereferencing a Indexed Random Access Iterator
Container type	The type of the container a Indexed Random Access Iterator points into.

Notation

`I`	A type that is a model of Indexed Random Access Iterator
`T`	The value type of `I`
`C`	The container type of `I`
`it`, `itt`, `it1`, `it2`	Objects of type `I`
`t`	Object of type `T`
`n`	Object of type `C::difference_type`

Definitions

An Indexed Random Access Iterator `it1` is *reachable* from an Indexed Random Access Iterator `it2` if, after applying `operator ++` to `it2` a finite number of times, `it1 == it2`.

Valid expressions

In addition to the expressions defined for Indexed Bidirectional Iterator , the following expressions must be valid.

Name	Expression	Type requirements	Return type
Forward motion	`it += n`		`I &`
Iterator addition	`it + n`		`I`
Backward motion	`i -= n`		`I &`
Iterator subtraction	`it - n`		`I`
Difference	`it1 - it2`		`C::difference_type`
Element operator	`it [n]`		Convertible to `T`.
Element assignment	`it [n] = t`	`I` is mutable	Convertible to `T`.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in, Indexed Bidirectional Iterator .

Name	Expression	Precondition	Semantics	Postcondition
Forward motion	`it += n`	Including `it` itself, there must be `n` dereferenceable or past-the-end iterators following or preceding `it`, depending on whether `n` is positive or negative.	If `n > 0`, equivalent to executing `++ it n` times. If `n < 0`, equivalent to executing `-- it n` times. If `n == 0`, this is a null operation.	`it` is dereferenceable or past-the-end.
Iterator addition	`it + n`	Same as for `i += n`.	Equivalent to `{ I itt = it; return itt += n; }`	Result is dereferenceable or past-the-end.
Backward motion	`it -= n`	Including `it` itself, there must be `n` dereferenceable or past-the-end iterators preceding or following `it`, depending on whether `n` is positive or negative.	Equivalent to `it += (-n)`.	`it` is dereferenceable or past-the-end.
Iterator subtraction	`it - n`	Same as for `i -= n`.	Equivalent to `{ I itt = it; return itt -= n; }`	Result is dereferenceable or past-the-end.
Difference	`it1 - it2`	Either `it1` is reachable from `it2` or `it2` is reachable from `it1`, or both.	Returns a number `n` such that `it1 == it2 + n`	
Element operator	`it [n]`	`it + n` exists and is dereferenceable.	Equivalent to `*(it + n)`	
Element assignment	`i[n] = t`	Same as for `it [n]`.	Equivalent to `*(it + n) = t`	

Complexity guarantees

The complexity of operations on indexed random access iterators is guaranteed to be amortized constant time.

Invariants

Symmetry of addition and subtraction	If `it + n` is well-defined, then `it += n; it -= n;` and `(it + n) - n` are null operations. Similarly, if `it - n` is well-defined, then `it -= n; it += n;` and `(it - n) + n` are null operations.
Relation between distance and addition	If `it1 - it2` is well-defined, then `it1 == it2 + (it1 - it2)`.
Reachability and distance	If `it1` is reachable from `it2`, then `it1 - it2 >= 0`.

Models

- `vector::iterator`

Indexed Bidirectional Column/Row Iterator

Description

An Indexed Bidirectional Column/Row Iterator is an iterator of a container that can be dereferenced, incremented, decremented and carries index information.

Refinement of

Assignable, Equality Comparable, Default Constructible.

Associated types

Value type	The type of the value obtained by dereferencing a Indexed Bidirectional Column/Row Iterator
Container type	The type of the container a Indexed Bidirectional Column/Row Iterator points into.

Notation

`I1`	A type that is a model of Indexed Bidirectional Column/Row Iterator
`I2`	A type that is a model of Indexed Bidirectional Row/Column Iterator
`T`	The value type of `I1` and `I2`
`C`	The container type of `I1` and `I2`
`it1`, `it1t`, `it11`, `it12`	Objects of type `I1`
`it2`, `it2t`	Objects of type `I2`

`t`	Object of type `T`
`c`	Object of type `C`

Definitions

Valid expressions

In addition to the expressions defined for Assignable, Equality Comparable and Default Constructible, the following expressions must be valid.

Name	Expression	Type requirements	Return type
Default constructor	`I1 it`		
Dereference	`*it`		Convertible to `T`.
Dereference assignment	`*it = t`	`I1` is mutable.	
Member access	`it->m`	`T` is a type for which `t.m` is defined.	
Preincrement	`++ it`		`I1 &`
Postincrement	`it ++`		`I1`
Predecrement	`-- it`		`I1 &`
Postdecrement	`it --`		`I1`
Row Index	`it.index1 ()`		`C::size_type`
Column Index	`it.index2 ()`		`C::size_type`
Row/Column Begin	`it.begin ()`		`I2`
Row/Column End	`it.end ()`		`I2`
Reverse Row/Column Begin	`it.rbegin ()`		`reverse_iterator<I2>`
Reverse Row/Column End	`it.rend ()`		`reverse_iterator<I2>`

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in, Assignable, Equality Comparable and Default Constructible.

Name	**Expression**	**Precondition**	**Semantics**	**Postcondition**
Default constructor	`I1 it`			`it` is singular.
Dereference	`*it`	`it` is dereferenceable.		

Name	Expression	Precondition	Semantics	Postcondition
Dereference assignment	`*it = t`	Same as for `*it`.		`*it` is a copy of t.
Member access	`it→m`	`it` is dereferenceable.	Equivalent to `(*it).m`	
Preincrement	`++ it`	`it` is dereferenceable.	`it` is modified to point to the next element of the column/row, i.e. for column iterators holds `it.index1 () < ( it).index1 ()` and + `it.index2 () == ( it).index2 ()`, for row iterators holds `it.index1 () == ( it).index1 ()` and + `it.index2 () < ( it).index2 ()`.	`it` is dereferenceable or past-the-end. ` &it == & it `. + If `it1 == it2 `, + then ` it1 == ++ it2 `.
Postincrement	`it ++`	Same as for `++ it`.	Equivalent to `{ I1 itt = it; ++ it; return itt; }`	`it` is dereferenceable or past-the-end.
Predecrement	`-- it`	`it` is dereferenceable or past-the-end. There exists a dereferenceable iterator `itt` such that `it == ++ itt`.	`it` is modified to point to the previous element of the column/row, i.e. for column iterators holds `it.index1 () > (-- it).index1 ()` and `it.index2 () == (-- it).index2 ()`, for row iterators holds `it.index1 () == (-- it).index1 ()` and `it.index2 () > (-- it).index2 ()`.	`it` is dereferenceable. `&it = &-- it`. If `it1 == it2`, then `-- it1 ==⍰–⍰it2`.
Postdecrement	`it --`	Same as for — `it`.	Equivalent to `{ I1 itt = it; -- it; return itt; }`	`it` is dereferenceable.

Name	Expression	Precondition	Semantics	Postcondition
Row Index	`it.index1 ()`	If `it` is a Row iterator then `it` must be dereferenceable.	`it.index1 () >= 0` and `it.index1 () < it () .size1 ()`	If `it1 == it2`, then `it1.index1() == 12.index1 ()`. If `it1`, `it2` are Row Iterators with `it1 == it2`, then `it1.index1 () < (++ it2).index1 ()`. and `it1.index1 () > (-- it2).index1 ()`.
Column Index	`it.index2 ()`	If `it` is a Column iterator then `it` must be dereferenceable.	`it.index2 () >= 0` and `it.index2 () < it () .size2 ()`	If `it1 == it2`, then `it1.index2 () == it2.index2 ()` . If `it1`, `it2` are Column Iterators with `it1 == i12`, then `it1.index2 () < (++ it2).index2 ()`. end `it1.index2 () > (-- it2).index2 ()`.
Row/Column Begin	`it.begin ()`	`it` is dereferenceable.	If `it` is a Column Iterator, then `it2 = it.begin ()` is a Row Iterator with `it2.index1 () == it.index1 ()`. If `it` is a Row Iterator, then `it2 = it.begin ()` is a Column Iterator with `it2.index2 () == it.index2 ()`.	

Name	Expression	Precondition	Semantics	Postcondition
Row/Column End	`it.end ()`	`it` is dereferenceable.	If `it` is a Column Iterator, then `it2 = it.end ()` is a Row Iterator with `it2.index1 () == it.index1 ()`. If `it` is a Row Iterator, then `it2 = it.end ()` is a Column Iterator with `it2.index2 () == it.index2 ()`.	
Reverse Row/Column Begin	`it.rbegin ()`	`it` is dereferenceable.	Equivalent to `reverse_iterator<I2> (it.end ())`.	
Reverse Row/Column End	`it.rend ()`	`it` is dereferenceable.	Equivalent to `reverse_iterator<I2> (it.begin ())`.	

Complexity guarantees

The complexity of operations on indexed bidirectional column/row iterators is guaranteed to be logarithmic depending on the size of the container. The complexity of one iterator (depending on the storage layout) can be lifted to be amortized constant time. The complexity of the other iterator (depending on the storage layout and the container) can be lifted to be amortized constant time for the first row/first column respectively.

Invariants

Identity	`it1 == it2` **if and only if** `&*it1 == &*it2`.
Symmetry of increment and decrement	If `it` is dereferenceable, then `++it; --it;` and `--it; ++it;` is a null operation. Similarly, `--it; it++;` is a null operation.
Relation between iterator index and container element operator	If `it` is dereferenceable, `*it == it () (it.index1 (), it.index2 ())`

Identity	`it1 == it2` **if and only if** `&*it1 == &*it2`**.**
Relation between iterator column/row begin and iterator index	If `it` is a Column Iterator and `it2 = it.begin ()` then `it2.index2 () < it2t.index2 ()` for all `it2t` with `it2t () == it2 ()` and `it2t ().index1 () == it2 ().index1 ()`. If `it` is a Row Iterator and `it2 = it.begin ()` then `it2.index1 () < it2t.index1 ()` for all `it2t` with `it2t () == it2 ()` and `it2t ().index2 () == it2 ().index2 ()`.
Relation between iterator column/row end and iterator index	If `it` is a Column Iterator and `it2 = it.end ()` then `it2.index2 () > it2t.index2 ()` for all `it2t` with `it2t () == it2 ()` and `it2t ().index1 () == it2 ().index1 ()`. If `it` is a Row Iterator and `it2 = it.end ()` then `it2.index1 () > it2t.index1 ()` for all `it2t` with `it2t () == it2 ()` and `it2t ().index2 () == it2 ().index2 ()`.

Models

- `sparse_matrix::iterator1`
- `sparse_matrix::iterator2`

Indexed Random Access Column/Row Iterator

Description

An Indexed Random Access Column/Row Iterator is an iterator of a container that can be dereferenced, incremented, decremented and carries index information.

Refinement of

Indexed Bidirectional Column/Row Iterator .

Associated types

Value type	The type of the value obtained by dereferencing a Indexed Random Access Column/Row Iterator
Container type	The type of the container a Indexed Random Access Column/Row Iterator points into.

Notation

`I`	A type that is a model of Indexed Random Access Column/Row Iterator
`T`	The value type of `I`
`C`	The container type of `I`
`it`, `itt`, `it1`, `it2`	Objects of type `I`
`t`	Object of type `T`
`c`	Object of type `C`

Definitions

Valid expressions

In addition to the expressions defined for Indexed Bidirectional Column/Row Iterator , the following expressions must be valid.

Name	Expression	Type requirements	Return type
Forward motion	`it += n`		`I &`
Iterator addition	`it + n`		`I`
Backward motion	`i -= n`		`I &`
Iterator subtraction	`it - n`		`I`
Difference	`it1 - it2`		`C::difference_type`
Element operator	`it [n]`		Convertible to `T`.
Element assignment	`it [n] = t`	`I` is mutable	Convertible to `T`.

Expression Semantics

Semantics of an expression is defined only where it differs from, or is not defined in, Indexed Bidirectional Column/Row Iterator .

Name	Expression	Precondition	Semantics	Postcondition
Forward motion	`it += n`	Including `it` itself, there must be `n` dereferenceable or past-the-end iterators following or preceding `it`, depending on whether `n` is positive or negative.	If `n > 0`, equivalent to executing `++ it n` times. If `n < 0`, equivalent to executing `-- it n` times. If `n == 0`, this is a null operation.	`it` is dereferenceable or past-the-end.

Name	Expression	Precondition	Semantics	Postcondition
Iterator addition	`it + n`	Same as for `i += n`.	Equivalent to `{ I itt = it; return itt += n; }`	Result is dereferenceable or past-the-end.
Backward motion	`it -= n`	Including `it` itself, there must be `n` dereferenceable or past-the-end iterators preceding or following `it`, depending on whether `n` is positive or negative.	Equivalent to `it += (-n)`.	`it` is dereferenceable or past-the-end.
Iterator subtraction	`it - n`	Same as for `i -= n`.	Equivalent to `{ I itt = it; return itt -= n; }`	Result is dereferenceable or past-the-end.
Difference	`it1 - it2`	Either `it1` is reachable from `it2` or `it2` is reachable from `it1`, or both.	Returns a number `n` such that `it1 == it2 + n`	
Element operator	`it [n]`	`it + n` exists and is dereferenceable.	Equivalent to `*(it + n)`	
Element assignment	`i[n] = t`	Same as for `it [n]`.	Equivalent to `*(it + n) = t`	

Complexity guarantees

The complexity of operations on indexed random access Column/Row iterators is guaranteed to be amortized constant time.

Invariants

Symmetry of addition and subtraction	If `it + n` is well-defined, then `it += n; it -= n;` and `(it + n) - n` are null operations. Similarly, if `it - n` is well-defined, then `it -= n; it += n;` and `(it - n) + n` are null operations.
Relation between distance and addition	If `it1 - it2` is well-defined, then `it1 == it2 + (it1 - it2)`.
Reachability and distance	If `it1` is reachable from `it2`, then `it1 - it2 >= 0`.

Models

- `matrix::iterator1`
- `matrix::iterator2`

Boost.uBlas Tutorial

For the official Boost.uBlas documentation visit here

You can find a set of tutorials and educational materials by the Boost community at Boost.uBlas Tutorials. The goal of this page is to provide high-quality resources by the Boost.uBlas project, both for self-learning and for teaching classes with, in the format of Godbolt. If you're interested in adding your own content, check the Boost.uBlas repository on GitHub.

Boost.uBlas Vector

What is a vector ?

Vectors are sequence containers representing arrays that can change in size. Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container. Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container. Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with amortized constant time complexity (see push_back). Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way. Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists.

Why a vector ?

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array. The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle

future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The complexity (efficiency) of common operations on vectors is as follows:

- Random access - constant □(1)
- Insertion or removal of elements at the end - amortized constant □(1)
- Insertion or removal of elements - linear in the distance to the end of the vector □(n)

How to create a vector in Boost.uBlas ?

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

int main () {
    using namespace boost::numeric::ublas;
    vector<int> v (3);
    for (unsigned i = 0; i < v.size (); ++ i) {
        v (i) = i;
    }
    std::cout << v << '\n';
}
```

Boost.uBlas Matrix

What is a matrix ?

A matrix is a specialized 2-D array that retains its 2-D nature through operations. It has certain special operators, such as **(matrix multiplication)** and **(matrix power)**. Just like vectors, matrix use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in vectors. But unlike vectors, their size can change dynamically, with their storage being handled automatically by the container. Internally, matrix use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container. Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with amortized constant time complexity (see push_back). Therefore, compared to vectors, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way. Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like vectors) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they

perform worse than the others, and have less consistent iterators and references than lists and forward_lists.

Why a matrix ?

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array. The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The complexity (efficiency) of common operations on vectors is as follows:

- Random access - constant 𝒪(1)
- Insertion or removal of elements at the end - amortized constant 𝒪(1)
- Insertion or removal of elements - linear in the distance to the end of the vector 𝒪(n)

How to create a matrix in Boost.uBlas ?

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<int> m (3, 3);
    for (unsigned i = 0; i < m.size1(); ++ i){
        for (unsigned j = 0; j < m.size2(); ++ j){
            m (i, j) = 3 * i + j;
        }
     }
    std::cout << m << std::endl;
}
```

Boost.uBlas Matrix

What is a matrix ?

In mathematics, a matrix (plural matrices) is a rectangular array or table of numbers, symbols, or expressions, arranged in rows and columns, which is used to represent a mathematical object or a property of such an object. For example,

is a matrix with two rows and three columns; one say often a "two by three matrix", a "2×3-matrix", or a matrix of dimension 2×3.

Without further specifications, matrices represent linear maps, and allow explicit computations in linear algebra. Therefore, the study of matrices is a large part of linear algebra, and most properties

and operations of abstract linear algebra can be expressed in terms of matrices. For example, matrix multiplication represents composition of linear maps.

Why a matrix ?

Working with matrices is a way to deal with a lot of numbers at the same time in a reduced space and practical way. Suppose we have a system of equations:

We can see that it has a repeating pattern, that for every equation, we have values multiplied by x, y and z. With matrices we have a way to write this system without repeating these and in a much more elegant way:

With this, adding one more equation to the system is a matter of adding numbers to the first and the last matrix, without touching the [x y z] matrix, since they keep repeating. (Those who already did endless exercise lists of equation systems knows how boring is to keep writing x, y, z over and over and over hehe) Neural networks often use too many values and operations, and write down every one of them would be impractical, so we need a way to compact things as much as possible to make them easier. Matrices help us with it.

Of course, these are not the only reasons to use function notations and matrices: With these we can write theorems and equations that generalize to any rule or any quantity of values. Another nice reason is that matrices are cpu/gpu friendly; computers take advantage of matrices to speed up processing their expressions.

How to create a matrix in Boost.uBlas ?

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

int main () {
    using namespace boost::numeric::ublas;
    matrix<double> m (3, 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

Boost.uBlas Tensor

In mathematics, a tensor is an algebraic object that describes a multilinear relationship between sets of algebraic objects related to a vector space. Objects that tensors may map between include vectors and scalars, and even other tensors. There are many types of tensors, including scalars and vectors, dual vectors, multilinear maps between vector spaces, and even some operations such as the dot product. Tensors are defined independent of any basis, although they are often referred to by their components in a basis related to a particular coordinate system. Tensors have become important in physics because they provide a concise mathematical framework for formulating and solving physics problems in areas such as mechanics, electrodynamics, or general relativity and

others. In applications, it is common to study situations in which a different tensor can occur at each point of an object;

Why a Tensor ?

One way to understand the importance of tensor calculus is to consider geometric complications when drawing right angles. If you are developing a system that uses the flat-earth model, you can draw right angles using the Pythagorean Theorem. The limits of the Pythagorean Theorem become clear when you try to draw a right angle on a spherical surface. In this case, the Pythagorean Theorem no longer works. It's here that the metric tensor comes to the rescue. It generalizes coordinates and geometries so that distance can be measured in any given space. The magic of tensors comes from their special transformational properties that enable them to describe the same physics in all reference frames.

Think of a tensor as a multi-linear map. Given a set of coordinates (or expand out to functions or other objects), each of these coordinates can be transformed according to a set of rules (linear transformations) into a new set of coordinates. The key here is that each coordinate can have a unique transformation. For example, you can stretch or distort different coordinates in different ways. If we take a rectangular piece of bubble gum with edges on the x, y, and z-axes, and then squeeze the bubble gum on the x-axis (one-dimension input), the x dimension will compress a certain amount, while the y and z dimensions will expand a given amount. This results in output changes in three dimensions while maintaining a constant volume. Assuming a linearity of the squeezing reaction, the behavior can also be calculated using a metric tensor if the gum is squeezed off-axis.

How to create a tensor in Boost.uBlas ?

```
#include <boost/numeric/ublas/tensor.hpp>

int main () {
  using namespace boost::numeric::ublas;
  tensor<double> t{4,2,3};
  for (auto k = 0ul; k < t.size (2); ++ k)
    for (auto j = 0ul; j < t.size (1); ++ j)
      for (auto i = 0ul; i < t.size (0); ++ i)
        t.at(i,j,k) = 3*i + 2*j + 5*k;

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

Release Notes

Boost Basic Linear Algebra

Release 1.70.0

improvements

- Add support for GPU-accelerated operations via Boost.Compute
- Add support for a new (arbitrary-rank) tensor type and associated operations.

Release 1.52.0

improvements

- [4024] improve performance of inplace_solve
- [6511] Division by scalar should use enable_if<>
- [7297] Make the free functions 'num_columns' and 'num_rows' support the uBLAS traits system and better work with expression types

bug fixes

- [7296] fixes and improvements to test utility functions
- [7363] fixed coordinate_matrix::sort() for gcc 4.7 and others

Release 1.43.0

bug fixes

- [3968] fixed coordinate_matrix sort problem on MSVC10
- [3539] changed computation of `norm_inf` for complex types to match mathematical definition. **Note:** This might cause a performance drop because now `std::abs(z)` is called for each vector element. The old implementation used `std::max(std::abs(real(z)),std::abs(imag(z))`. Further `norm_inf` and `norm_1` will now return the same values for complex vector.
- [3501] Moved free functions in `concepts.hpp` into anonymous namespace.

Release 1.41.1

new features

- Move semantics of vector/matrix container assignments have been implemented. They can be enabled by setting BOOST_UBLAS_MOVE_SEMANTICS. More details are on the preprocessor options page.
- Introduce new free functions. See [3449], the new tests in libs/numeric/ublas/test and the inline documentation of the files in boost/numeric/ublas/operation/.

bug fixes

- [3293] Fix resizing problem in `identity_matrix`
- [3499] Add DefaultConstructible to concept checks

Release 1.40.0 and before

- Release notes were not available in this form.

CONTRIBUTORS

Appendix A: Copyright and License