uBLAS 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);`
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 &);

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)`	*sum* $\|x_i\|$	Computes the l_1 (sum) norm of a real vector.
`scasum` OR `dzasum`	*`real (sum (v)) + imag (sum (v))`*	*sum re*(x_i) + *sum im*(x_i)	Computes the sum of elements of a complex vector.
`_nrm2`	`norm_2 (x)`	*sqrt (sum* $\|x_i\|^2$ *)*	Computes the l_2 (euclidean) norm of a vector.
`i_amax`	`norm_inf (x)` `index_norm_inf (x)`	*max* $\|x_i\|$	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 <- y	Copies one vector to another. BLAS implements certain loop unrollment.
`_swap`	`swap (x, y)`	x <-> y	Swaps two vectors. BLAS implements certain loop unrollment.
`_scal` `csscal` `zdscal`	`x *= a`	x <- $a\ x$	Scales a vector. BLAS implements certain loop unrollment.
`_axpy`	`y += a * x`	y <- $a\ x + y$	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) <- $(a\ x + b\ y, -b\ x + a\ y)$	Applies a plane rotation.
`_rotg` `_rotmg`		(a, b) <- $($? $a / sqrt (a^2 + b^2),$	Constructs a plane rotation.

		? b / sqrt (a^2 *+* b^2*))* or *(1, 0) <- (0, 0)*	

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)`	$x <- A\, x$ or $x <- A^T x$ or $x <- A^H x$	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)`	$y <- A^{-1} x$ or $x <- A^{-1} x$ or $y <- A^{T^{-1}} x$ or $x <- A^{T^{-1}} x$ or $y <- A^{H^{-1}} x$ 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`
		$y <- a\, A\, x + b\, y$
		or
		$y <- a\, A^T x + b\, y$
		$y <- a\, A^H x + b\, y$
			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))`
		$A <- a\, x\, y^T + A$
		or
		$A <- a\, x\, y^H + A$
			Performs a rank *1* update.

`_s_r` `_h_r`	`A += a * outer_prod (x, x)` or `A += a * outer_prod (x, conj (x))`	$A <- a\, x\, x^T + A$ or $A <- a\, x\, x^H + A$	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)))`	$A <- a\, x\, y^T + a\, y\, x^T + A$ or $A <- a\, x\, y^H + a^{-} y\, x^H + A$	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 <- 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 <-* A^{-1} *B* or *B <-* A^{-1} *B* or *C <-* $A^{T^{-1}}$ *B* or *B <-* A^{-1} *B* or *C <-* $A^{H^{-1}}$ *B* or *B <-* $A^{H^{-1}}$ *B*	Solves a system of linear equations with triangular form, i.e. *A* is triangular.

`_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 <- 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`	$B <- a A A^T + b B$ or $B <- a A^T A + b B$ or $B <- a A A^H + b B$ or $B <- a A^H A + b B$	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`
		$C <- a A B^T + a B A^T + b C$ or
		$C <- a A^T B + a B^T A + b C$ or
		$C <- a A B^H + a^- B A^H + b C$ or
		$C <- a A^H B + a^- B^H A + b C$
			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
	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	
	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	
	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.
