Overview of Tensor, Matrix and Vector Operations

Contents:

Definitions

`X, Y, Z`	are tensors
`A, B, C`	are matrices
`u, v, w`	are vectors
`i, j, k`	are integer values
`t, t1, t2`	are scalar values
`r, r1, r2`	are ranges, e.g. `range(0, 3)`
`s, s1, s2`	are slices, e.g. `slice(0, 1, 3)`

Basic Linear Algebra

standard operations: addition, subtraction, multiplication by a scalar

```
X = Y + Z; X = Y - Z; X = -Y;
C = A + B; C = A - B; C = -A;
w = u + v; w = u - v; w = -u;
X = t * Y; Y = X * t; X = Y / t;
C = t * A; C = A * t; C = A / t;
w = t * u; w = u * t; w = u / t;
```

computed assignments

```
X += Y; X -= Y;
C += A; C -= A;
w += u; w -= u;
X *= t; X /= t;
C *= t; C /= t;
w *= t; w /= t;
```

inner, outer and other products

```
t = inner_prod(u, v);
C = outer_prod(u, v);
w = prod(A, u); w = prod(u, A); w = prec_prod(A, u); w = prec_prod(u, A)
C = prod(A, B); C = prec_prod(A, B);
w = element_prod(u, v); w = element_div(u, v);
C = element_prod(A, B); C = element_div(A, B);
```

tensor products

```
Z = prod(X, v, t);
Z = prod(X, A, t);
Z = prod(X, Y, p);
Z = prod(X, Y, pa, pb);
t = inner_prod(X, Y);
Z = outer_prod(X, Y);
```

transformations

```
w = conj(u); w = real(u); w = imag(u);
C = trans(A); C = conj(A); C = herm(A); C = real(A); C = imag(A);
Z = trans(X); Z = conj(X); Z = real(X); Z = imag(X);
```

Advanced functions

norms

```
t = norm_inf(v); i = index_norm_inf(v);
t = norm_1(v);   t = norm_2(v);
t = norm_2_square(v);
t = norm_inf(A); i = index_norm_inf(A);
t = norm_1(A);   t = norm_frobenius(A);
t = norm(X);
```

products

```
axpy_prod(A, u, w, true);  // w = A * u
axpy_prod(A, u, w, false); // w += A * u
axpy_prod(u, A, w, true);  // w = trans(A) * u
axpy_prod(u, A, w, false); // w += trans(A) * u
axpy_prod(A, B, C, true);  // C = A * B
axpy_prod(A, B, C, false); // C += A * B
```

Note: The last argument (`bool init`) of `axpy_prod` is optional. Currently it defaults to `true`, but this may change in the future. Setting the `init` to `true` is equivalent to calling `w.clear()` before `axpy_prod`. There are some specialisation for products of compressed matrices that give a large speed up compared to `prod`.

```
w = block_prod<matrix_type, 64> (A, u); // w = A * u
w = block_prod<matrix_type, 64> (u, A); // w = trans(A) * u
C = block_prod<matrix_type, 64> (A, B); // C = A * B
```

Note: The blocksize can be any integer. However, the actual speed depends very significantly on the combination of blocksize, CPU and compiler. The function `block_prod` is designed for large dense matrices.

rank-k updates

```
opb_prod(A, B, C, true);  // C = A * B
opb_prod(A, B, C, false); // C += A * B
```

Note: The last argument (`bool init`) of `opb_prod` is optional. Currently it 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.

Submatrices, Subvectors

Accessing submatrices and subvectors via **proxies** using `project` functions:

```
w = project(u, r);         // the subvector of u specifed by the index r
w = project(u, s);         // the subvector of u specifed by the index s
C = project(A, r1, r2);    // the submatrix of A specified by the two ind
C = project(A, s1, s2);    // the submatrix of A specified by the two ind
w = row(A, i); w = column(A, j);    // a row or column of matrix as a ve
```

Assigning to submatrices and subvectors via **proxies** using `project` functions:

```
project(u, r) = w;         // assign the subvector of u specifed by the 
project(u, s) = w;         // assign the subvector of u specifed by the 
project(A, r1, r2) = C;    // assign the submatrix of A specified by the
project(A, s1, s2) = C;    // assign the submatrix of A specified by the
row(A, i) = w; column(A, j) = w;    // a row or column of matrix as a ve
```

Note: A range `r = range(start, stop)` contains all indices `i` with `start <= i < stop`. A slice is something more general. The slice `s = slice(start, stride, size)` contains the indices `start, start+stride, ..., start+(size-1)*stride`. The stride can be 0 or negative! If `start >= stop` for a range or `size == 0` for a slice then it contains no elements.

Sub-ranges and sub-slices of vectors and matrices can be created directly with the `subrange` and `sublice` functions:

```
w = subrange(u, 0, 2);          // the 2 element subvector of u
w = subslice(u, 0, 1, 2);       // the 2 element subvector of u
C = subrange(A, 0,2, 0,3);      // the 2x3 element submatrix of A
C = subslice(A, 0,1,2, 0,1,3); // the 2x3 element submatrix of A
```

```
subrange(u, 0, 2) = w;         // assign the 2 element subvector of u
subslice(u, 0, 1, 2) = w;      // assign the 2 element subvector of u
subrange(A, 0,2, 0,3) = C;     // assign the 2x3 element submatrix of A
subrange(A, 0,1,2, 0,1,3) = C; // assigne the 2x3 element submatrix of A
```

There are to more ways to access some matrix elements as a vector:

```
matrix_vector_range<matrix_type> (A, r1, r2);
matrix_vector_slice<matrix_type> (A, s1, s2);
```

Note: These matrix proxies take a sequence of elements of a matrix and allow you to access these as a vector. In particular `matrix_vector_slice` can do this in a very general way. `matrix_vector_range` is less useful as the elements must lie along a diagonal.

Example: To access the first two elements of a sub column of a matrix we access the row with a slice with stride 1 and the column with a slice with stride 0 thus:
`matrix_vector_slice<matrix_type> (A, slice(0,1,2), slice(0,0,2));`

Speed improvements

Matrix / Vector assignment

If you know for sure that the left hand expression and the right hand expression have no common storage, then assignment has no *aliasing*. A more efficient assignment can be specified in this case:

```
noalias(C) = prod(A, B);
```

This avoids the creation of a temporary matrix that is required in a normal assignment. 'noalias' assignment requires that the left and right hand side be size conformant.

Sparse element access

The matrix element access function `A(i1,i2)` or the equivalent vector element access functions (`v(i) or v[i]`) usually create 'sparse element proxies' when applied to a sparse matrix or vector. These *proxies* allow access to elements without having to worry about nasty C++ issues where references are invalidated.

These 'sparse element proxies' can be implemented more efficiently when applied to `const` objects. Sadly in C++ there is no way to distinguish between an element access on the left and right hand side of an assignment. Most often elements on the right hand side will not be changed and therefore it would be better to use the `const` proxies. We can do this by making the matrix or vector `const` before accessing it's elements. For example:

```
value = const_cast<const VEC>(v)[i];   // VEC is the type of V
```

If more then one element needs to be accessed `const_iterator`'s should be used in preference to `iterator`'s for the same reason. For the more daring 'sparse element proxies' can be completely turned off in uBLAS by defining the configuration macro `BOOST_UBLAS_NO_ELEMENT_PROXIES`.

Controlling the complexity of nested products

What is the complexity (the number of add and multiply operations) required to compute the following?

```
R = prod(A, prod(B,C));
```

Firstly the complexity depends on matrix size. Also since prod is transitive (not commutative) the bracket order affects the complexity.

uBLAS evaluates expressions without matrix or vector temporaries and honours the bracketing structure. However avoiding temporaries for nested product unnecessarly increases the complexity. Conversly by explictly using temporary matrices the complexity of a nested product can be reduced.

uBLAS provides 3 alternative syntaxes for this purpose:

```
temp_type T = prod(B,C); R = prod(A,T);   // Preferable if T is preallo
```

```
prod(A, temp_type(prod(B,C));
```

```
prod(A, prod<temp_type>(B,C));
```

The 'temp_type' is important. Given A,B,C are all of the same type. Say matrix<float>, the choice is easy. However if the value_type is mixed (int with float or double) or the matrix type is mixed (sparse with symmetric) the best solution is not so obvious. It is up to you! It depends on numerical properties of A and the result of the prod(B,C).
