LECTURE 5

Vector

- Vectors in `boost` are templated containers
- The template parameter can be any type satisfying certain requirements
- To access the basic vector features, include

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

- the actual type is (for `double` values)

```
boost::numeric::ublas::vector<double>
```

- use `typedef` to define a shorter type name, e.g.

```
typedef boost::numeric::ublas::vector<double> BVector;
```

- for input/output operations, include

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

Vector - first examples

```
// constructing vectors
BVector vec1, vec2(2);
vec2[0]=1.3;
vec2[1]=2.5;
BVector vec3(vec2);
cout << "vec1=" << vec1 <<endl;
cout << "vec2=" << vec2 <<endl;
cout << "vec3=" << vec3 <<endl;
```

The output is:

```
vec1=[0]()
vec2=[2](1.3,2.5)
vec3=[2](1.3,2.5)
```

Vector - first examples

```
1  // constructing vectors
2  BVector vec1, vec2(2);
3  vec2[0]=1.3;
4  vec2[1]=2.5;
5  BVector vec3(vec2);
6  cout << "vec1=" << vec1 <<endl;
7  cout << "vec2=" << vec2 <<endl;
8  cout << "vec3=" << vec3 <<endl;
```

- `vec1`: the default constructor initialises an empty vector
- `vec2(2)`: initialises a vector of size 2
- `vec3(vec2)`: copy constructor

Vector - vector operations

- Operators +, +=, −, −= taking vectors are available
- For scalar operations the operators ⋆, ⋆=, / and /= are available (taking variables of the same type

```
1  // +, +=, etc. operator
2  BVector vec4=vec2+vec3;
3  cout << "vec4=vec2+vec3=" << vec4 <<endl;
4  cout << "(vec4⋆=3.0)=" << (vec4⋆=3.0) <<endl;
```

The output is

```
vec4=vec2+vec3=[2](2.6,5)
(vec4⋆=3.0)=[2](7.8,15)
```

- The members `size()`, `resize()`, `empty()`, `max_size()`, operator `[]`, `clear()`, `begin()`, `end()`, `swap()` behave similarly to the stl vector members of the same names
- Note that **no** `push_back()` member is provided. However there is a member `insert_element()` taking an index and a reference.

Vector - iterators

`boost` vectors support iterators, and stl algorithms can be used on these vectors, e.g.

```
1  // using stl algorithms on boost vectors
2  BVector vec5(vec2.size());
3  BVector::iterator bIter1 = vec2.begin();
4  BVector::iterator eIter1 = vec2.end();
5  BVector::iterator bIter2 = vec3.begin();
6  BVector::iterator iter = vec5.begin();
7  std::transform(bIter1,
8       eIter1,
9       bIter2,
10      iter,
11      std::plus<double>());
12 cout << "vec5 = " << vec5 << endl;
```

The output:

```
vec5 = [2](2.6,5)
```

Vector - further global functions

Some further global functions taking vectors:

```
1  // inner_prod, outer_prod, element_prod
2  cout << "inner_prod(vec4,vec3)="
3      << boost::numeric::ublas::inner_prod(vec4,vec3) << endl;
4  cout << "outer_prod(vec4,vec3)="
5      << boost::numeric::ublas::outer_prod(vec4,vec3) << endl;
6  cout << "element_prod(vec4,vec3)="
7      << boost::numeric::ublas::element_prod(vec4,vec3) << endl;
```

- `inner_prod()` - inner product, returns a scalar
- `outer_prod()` - outer product, returns a matrix
- `element_prod()` - element-wise product returns a vector

The output:

```
inner_prod(vec4,vec3)=47.64
outer_prod(vec4,vec3)=[2,2]((10.14,19.5),(19.5,37.5))
element_prod(vec4,vec3)=[2](10.14,37.5)
```

Matrix

- Matrices in `boost` are templated containers indexed by two indices
- To access the basic matrix features, include

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

- There are more advanced fatures available via

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

- The input/output operations are available via

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

- Documentation is available at `http://www.boost.org/doc/libs/1_40_0/libs/numeric/ublas/doc/matrix.htm`

Matrix - construction matrices

One possible way to construct a matrix is by determining the size. Then the elements can be specified using the operator `()` taking two indices.

```
typedef boost::numeric::ublas::matrix<double> BMatrix;
 //constructing matrices
 BMatrix mat1(3,4);
 for(unsigned int i=0; i<mat1.size1(); ++i) //defining entries
  for(unsigned int j=0; j<mat1.size2(); ++j)
    mat1(i,j)=i*j;
 cout << "mat1=" << mat1 << endl;
 cout << "mat1.size1()=" << mat1.size1() << endl;
 cout << "mat1.size2()=" << mat1.size2() << endl;
 BMatrix mat2(mat1); //copy constructor
 cout << "mat2=" << mat2 << endl;
 mat2.resize(3,3); //resizing the matrix
 cout << "mat2=" << mat2 << endl;
```

```
mat1=[3,4]((0,0,0,0),(0,1,2,3),(0,2,4,6))
mat1.size1()=3
mat1.size2()=4
mat2=[3,4]((0,0,0,0),(0,1,2,3),(0,2,4,6))
mat2=[3,3]((0,0,0),(0,1,2),(0,2,4))
```

Matrix - matrix operations

Special matrices (see examples later)

- `identity_matrix`
- `zero_matrix`
- `scalar_matrix`

Some important members

- `size1()` and `size2()`: take no argument, return dimensions
- `resize()`: takes two indices
- `insert_element()`: takes two indices and a reference to a value
- `erase_element()`: takes two indices
- `clear()`, `swap()`: usual clear and swap operations

Matrix - matrix operations

Algebraic operations

- `boost` provides operators +, +=, − and −= taking two matrix arguments and operators
- ⋆, ⋆=, / and /= taking one matrix one scalar as arguments.

```
1 BMatrix mat3(mat2*3);
2 cout << "mat3=" << mat3 << endl;
3 cout << "mat3+=mat2" << (mat3+=mat2) << endl;
4 cout << "mat3/=2.0" << (mat3/=2.0) << endl;
```

The output:

```
mat3=[3,3]((0,0,0),(0,3,6),(0,6,12))
mat3+=mat2[3,3]((0,0,0),(0,4,8),(0,8,16))
mat3/=2.0[3,3]((0,0,0),(0,2,4),(0,4,8))
```

Matrix - matrix operations

To multiply matrices and vectors use the `prod()` function:

```
1  BVector e1(boost::numeric::ublas::unit_vector<double>(3,0));
2  cout << "mat3=" << mat3 << endl;
3  cout << "e1=" << e1 << endl;
4  cout << "mat3*e1=" << prod(mat3,e1) << endl;
5  cout << "mat3*mat3=" << prod(mat3,mat3) << endl;
```

The output:

```
mat3=[3,3]((0,0,0),(0,2,4),(0,4,8))
e1=[3](1,0,0)
mat3*e1=[3](0,0,0)
mat3*mat3=[3,3]((0,0,0),(0,20,40),(0,40,80))
```

Matrix - iterators

Matrices support iterators, however the syntax is slightly different; in matrices iterators iterate either on a row or a column.

```
1  //iterators
2  cout << "mat3=" << mat3 << endl;
3  BMatrix::iterator1 iter1=mat3.begin1();
4  (*iter1)+=1.0;
5  cout << "mat3=" << mat3 << endl;
6  ++iter1;
7  (*iter1)+=1.0;
8  cout << "mat3=" << mat3 << endl;
9  BMatrix::iterator2 iter2(mat3.begin2());
10 (*iter2)+=1.0;
11 cout << "mat3=" << mat3 << endl;
12 ++iter2;
13 (*iter2)+=1.0;
14 cout << "mat3=" << mat3 << endl;
15 BMatrix::iterator1 iter12(iter2.begin());
16 (*iter12)+=1.0;
17 cout << "mat3=" << mat3 << endl;
18 ++iter12;
19 (*iter12)+=1.0;
20 cout << "mat3=" << mat3 << endl;
```

Matrix - iterators

The output of the previous piece of code:

```
mat3=[3,3]((0,0,0),(0,2,4),(0,4,8))
mat3=[3,3]((1,0,0),(0,2,4),(0,4,8))
mat3=[3,3]((1,0,0),(1,2,4),(0,4,8))
mat3=[3,3]((2,0,0),(1,2,4),(0,4,8))
mat3=[3,3]((2,1,0),(1,2,4),(0,4,8))
mat3=[3,3]((2,2,0),(1,2,4),(0,4,8))
mat3=[3,3]((2,2,0),(1,3,4),(0,4,8))
mat3=[3,3]((2,2,0),(1,3,4),(0,4,8))
```

- `begin1()` returns a column iterator (`iterator1`)
- `iter1` has type `iterator1` and it iterates in the first column
- `begin2()` returns a row iterator (`iterator2`)
- `iter2` has type `iterator2` and it iterates in the first row
- `iter12` is again a column iterator iterating in the second column

Matrix - proxies

Matrix proxies are used to access sub-ranges of matrices.

```
#include <boost/numeric/ublas/matrix_proxy.hpp>
  typedef boost::numeric::ublas::matrix_row<BMatrix> BMatrixRow;
  typedef boost::numeric::ublas::matrix_column<BMatrix> BMatrixCol;

  BMatrixRow matrow1(mat3,1);
  BMatrixCol matcol1(mat3,0);
  cout << "mat3=" << mat3 << endl;
  cout << "row 1 of mat3=" << matrow1 << endl;
  cout << "col 0 of mat3=" << matcol1 << endl;
  matrow1(1)=10;
  cout << "row 1 of mat3=" << matrow1 << endl;
  cout << "mat3=" << mat3 << endl;
  matrow1*=10;
  cout << "row 1 of mat3=" << matrow1 << endl;
  cout << "mat3=" << mat3 << endl;
  matcol1(1)=0.1;
  cout << "col 0 of mat3=" << matcol1 << endl;
  cout << "mat3=" << mat3 << endl;
  matcol1*=0.1;
  cout << "col 0 of mat3=" << matcol1 << endl;
  cout << "mat3=" << mat3 << endl;
  std::fill(matrow1.begin(),matrow1.end(),3.14);
  cout << "row 1 of mat3=" << matrow1 << endl;
  cout << "mat3=" << mat3 << endl;
```

Matrix - proxies

The output of the previous piece of code:

```
mat3=[3,3]((2,2,0),(1,3,4),(0,4,8))
row 1 of mat3=[3](1,3,4)
col 0 of mat3=[3](2,1,0)
row 1 of mat3=[3](1,10,4)
mat3=[3,3]((2,2,0),(1,10,4),(0,4,8))
row 1 of mat3=[3](10,100,40)
mat3=[3,3]((2,2,0),(10,100,40),(0,4,8))
col 0 of mat3=[3](2,0.1,0)
mat3=[3,3]((2,2,0),(0.1,100,40),(0,4,8))
col 0 of mat3=[3](0.2,0.01,0)
mat3=[3,3]((0.2,2,0),(0.01,100,40),(0,4,8))
row 1 of mat3=[3](3.14,3.14,3.14)
mat3=[3,3]((0.2,2,0),(3.14,3.14,3.14),(0,4,8))
```

- row and column proxies are constructed from a matrix an the index of the row/column respectively.
- any modification of a row or column proxy also modifies the matrix
- row and column proxies behave like vectors, one can call vector operations on them and in particular stl algorithms (e.g. `fill()`) using the vector iterators.

Linear algebra

The `boost` library contains some linear algebraic features. For detailed **documentation**, see
`http://www.boost.org/doc/libs/1_40_0/libs/numeric/ublas/doc/operations_overview.htm`

For optimised linear algebra operations the following **special containers** might be useful

- sparse vectors - `mapped_vector`:
  `http://www.boost.org/doc/libs/1_40_0/libs/numeric/ublas/doc/vector_sparse.htm`
- sparse matrices - `mapped_matrix`:
  `http://www.boost.org/doc/libs/1_40_0/libs/numeric/ublas/doc/matrix_sparse.htm`
- triangular matrices - `triangular_matrix`
  `http://www.boost.org/doc/libs/1_40_0/libs/numeric/ublas/doc/triangular.htm`

Norm functions provided by the library:

- vector norms: `norm_inf()`, `norm_1()`, `norm_2()`
- for matrices: `norm_inf()`, `norm_1()`, `norm_frobenius()`

Solving linear equation

LU factorisation is available in `boost`.

```
#include <boost/numeric/ublas/lu.hpp>
  BMatrix A(2,2);
  A(0,0)=1;
  A(0,1)=1;
  A(1,0)=1;
  A(1,1)=0;
  BVector b(2);
  b(0)=5;
  b(1)=2;
  cout << "A=" << A << endl;
  cout << "b=" << b << endl;
  //solving Ax=b
    boost::numeric::ublas::lu_factorize(A);
  cout << "A=" << A << endl;
  boost::numeric::ublas::lu_substitute<const BMatrix,BVector>(A,b);
  cout << "b=" << b << endl;
  //solving A*Ainv=Identity
    BMatrix Ainv(boost::numeric::ublas::identity_matrix<double>(2));
  cout << "Ainv=" << Ainv << endl;
  boost::numeric::ublas::lu_substitute<const BMatrix,BMatrix>(A,Ainv);
  cout << "Ainv=" << Ainv << endl;
```

Solving linear equation

The output of the previous piece of code:

```
A=[2,2]((1,1),(1,0))
b=[2](5,2)
A=[2,2]((1,1),(1,-1))
b=[2](2,3)
Ainv=[2,2]((1,0),(0,1))
Ainv=[2,2]((0,1),(1,-1))
```

- `lu_factorize(A)` writes the factors into `A`
- `lu_substitute<const BMatrix,BVector>(A,b)` solves the equation $Ax = b$ (where A is the original matrix) writing the result into `b`.
- `Ainv` is initialized to be the identity matrix (note: the type of `Ainv` is a `BMatrix`, however it is constructed from a special type matrix)
- `lu_substitute<const BMatrix,BMatrix>(A,Ainv)` solves $A \cdot Ainv = I$, i.e. it inverts `A`.

Summary

`boost` vector	operations, iterators
`inner_prod()`	`element_prod()`
`boost` matrix	operations, iterators
matrix proxies	`prod()`
`lu_factorize()`	`lu_substitute()`