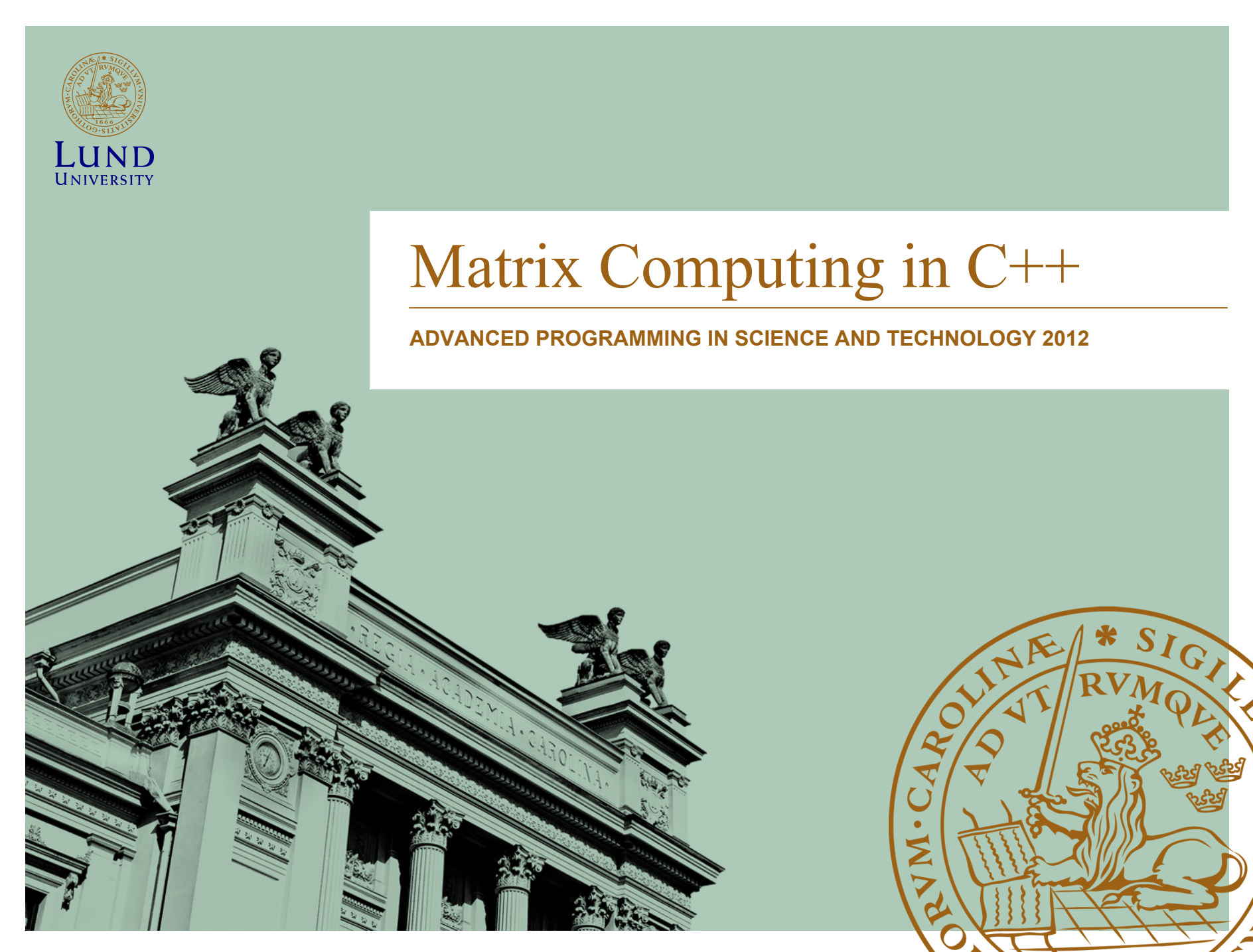


LUND UNIVERSITY

Matrix Computing in C++

ADVANCED PROGRAMMING IN SCIENCE AND TECHNOLOGY 2012

Matrix Computing with C++

- Matrices not native to C++
- Basic Linear Algebra Subprograms – BLAS
  - Defacto standard for matrix computing
  - Interface exists for C
  - Not so easy to use
  - You should use it

```
void sgemm(const char *transa, const char *transb,
   int *l, int *n, int *m, float *alpha,
   const void *a, int *lda, void *b, int *ldb,
   float *beta, void *c, int *ldc);
```

This is not MATLAB 

Solution

- Use a C++ Matrix library
- Should have bindings to MKL/ACML/ATLAS to be efficient
- It should be possible to get to “raw” pointers
  - Interfacing with other routines
- Make sure there is continous development
  - You are going to base your application on it
  - You don’t want to replace it later
- Platform support
  - Where are your code going to ported?

Available libraries

- Eigen
- uBLAS
  - No matrix inversion. Verbose syntax
- Newmat11
  - No BLAS/LAPACK binding. Licensing?
- Armadillo C++
  - Provides a lot of MATLAB functionality
  - Liberal licensing

Available libraries

- Blitz++
  - Strict GPL license
  - Supports Matrix, Tensors and Vectors
- IT++
  - Strict GPL license
  - No lazy evaluation

Armadillo C++ - Feature Criteria

- An easy to use syntax, similar to Matlab and Octave.
- Support numerical element types useful in linear algebra
- Use of the LAPACK library as a building block for providing various matrix decompositions. As a side-effect, this also allows processor-specific high-performance LAPACK replacement libraries to be used.
- An efficient delayed-operations framework, based on template meta-programming and recursive templates, allowing fast matrix manipulations.

From - Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. (http://arma.sourceforge.net/armadillo_nicta_2010.pdf)

LUND UNIVERSITY

Armadillo C++ - Feature Criteria

- An efficient mechanism to provide read and write access to sub-matrices.
- Ability to load and save matrices from files.
- Allow easy interfacing with other libraries, by providing access to elements via STL-iterators as well as matrix initialisation from an existing memory block.
- Open source development, allowing contributions from outside parties. The contributions include new features as well as bug reports.

From - Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. (http://arma.sourceforge.net/armadillo_nicta_2010.pdf)

Armadillo C++ - Feature Criteria

- Distribution of the library source code under a license that is appropriate for the development of both open-source and closed source (proprietary) software. We have chosen the LGPL license (which is different to the GPL)…
- Cross-platform, usable on Unix-like systems (e.g. Linux, Solaris, Mac OS X, BSD), as well as lower grade systems such as MS Windows.

From - Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. (http://arma.sourceforge.net/armadillo_nicta_2010.pdf)

Armadillo C++ - Usability

- Submatrix access important
- Make conversion of code from MATLAB/Octave easier

From - Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. (http://arma.sourceforge.net/armadillo_nicta_2010.pdf)

LUND UNIVERSITY

Example of a Armadillo C++ program

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main()
{
    mat A = randu<mat>(4,5);
    mat B = randu<mat>(4,5);

    cout << A*B.t() << endl;
}
```

```
   1.3761   1.3042   1.9442   1.3841
   0.6430   0.8686   0.9816   0.9922
   0.7714   1.1052   1.2178   1.1837
   1.1633   1.1126   1.2615   1.4649
```

Matrix types

- Root matrix class is **Mat<type>**
  - Type can be **char, int, float, double, std::complex<double>** etc.
- Convenience typedefs:
  - **mat = Mat<double>**
  - **fmat = Mat<float>**
  - **cx_mat = Mat<cx_double>**
  - **cx_fmat = Mat<cx_float>**
  - **umat = Mat<uword>**
  - **imat = Mat<sword>**

LUND UNIVERSITY

Matrix constructors

- Available constructors
  - **mat()**
  - **mat(n_rows, n_cols)**
  - **mat(mat)**
  - **mat(vec)**
  - **mat(rowvec)**
  - **mat(string)**
  - **mat(std::vector)** (treated as a column vector)
  - **mat(initialiser_list)** (C++11 only)
  - **cx_mat(mat,mat)** (for constructing a complex matrix out of two real matrices)

Special constructors

- **mat(aux_mem*, n_rows, n_cols, copy_aux_mem = true, strict = true)**
  - Creates matrix from external data (aux_mem*)
  - if **copy_aux_mem = false** no copying will be done (DANGEROUS)
  - **strict** controls if the matrix should be bound to external memory during its lifetime

Special constructors

- **mat(const aux_mem*, n_rows, n_cols)**
  - Creates matrix from external data (aux_mem*) by copying
- **mat::fixed<n_rows, n_cols>**
  - Fixed size matrix. Memory is allocated at compile time
  - Cannot change size
  - square typedefs defined from 2x2 to 9x9 (mat22, mat33 …)

Example

```
mat A = randu<mat>(5,5);
double x = A(1,2);

mat B = A + A;
mat C = A * B;
mat D = A % B;

cx_mat X(A,B);

B.zeros();
B.set_size(10,10);
B.zeros(5,6);
```

```
//
// fixed size matrices:

mat::fixed<5,6> F;
F.ones();

mat44 G;
G.randn();

cout << mat22().randu() << endl;

//
// constructing matrices from
// auxiliary (external) memory:

double aux_mem[24];
mat H(aux_mem, 4, 6, false);
```

LUND UNIVERSITY

Treatment of Scalars (Warning)

- Scalars are treated as 1x1 matrices during initialisation.
  - mat A(5,5); A = 123.0;
  - Will not generate a 5x5 matrix with all elements set to 123.0
  - Instead:
  - mat A(5,5); A.fill(123.0);

Column vectors

- Classes for vectors (matrices with one column)
- Base class **Col<type>** inherits most members from **Mat<type>**
- Convenience typedefs
  - vec, colvec = Col<double>
  - fvec, fcolvec = Col<float>
  - cx_vec, cx_colvec = Col<cx_double>
  - cx_fvec, cx_fcolvec = Col<cx_float>
  - uvec, ucolvec = Col<uword>
  - ivec, icolvec = Col<sword>

Column vectors

- vec = colvec
- Functions that take **Mat** as input can take column vectors as input as well.

Column vector constructors

- **vec(n_elem=0)**
- **vec(vec)**
- **vec(mat)** (a std::logic_error exception is thrown if the given matrix has more than one column)
- **vec(string)** (elements separated by spaces)
- **vec(std::vector)**
- **vec(initialiser_list)** (C++11 only)

Advanced constructors (same as Mat)

- **vec(aux_mem*, number_of_elements, copy_aux_mem = true, strict = true)**
- **vec(const aux_mem*, number_of_elements)**
- **vec::fixed<number_of_elements>**
- **vec::fixed<number_of_elements>**

Example

```
vec x(10);
vec y = zeros<vec>(10,1);

mat A = randu<mat>(10,10);
vec z = A.col(5); // extract a column vector
```

LUND UNIVERSITY

Row vectors

- Classes for vectors (matrices with one row)
- Base class **Row<type>** inherits most members from **Mat<type>**
- Convenience typedefs
  - **rowvec = Row<double>**
  - **frowvec = Row<float>**
  - **cx_rowvec = Row<cx_double>**
  - **cx_frowvec = Row<cx_float>**
  - **urowvec = Row<uword>**
  - **irowvec = Row<sword>**

Example

```
rowvec x(10);
rowvec y = zeros<mat>(1,10);

mat A = randu<mat>(10,10);
rowvec z = A.row(5); // extract a row vector
```

LUND UNIVERSITY

Sparse Matrix Types

- Root matrix class is **SpMat<type>**
  - Type can be **char, int, float, double, std::complex<double>** etc.
- Convenience typedefs:
  - **sp_mat = SpMat<double>**
  - **sp_fmat = SpMat<float>**
  - **sp_cx_mat = SpMat<cx_double>**
  - **sp_cx_fmat = SpMat<cx_float>**
  - **sp_umat = SpMat<uword>**
  - **sp_imat = SpMat<sword>**

LUND UNIVERSITY

Sparse Matrix notes

- Elements stored in CSC format

val	10	3	3	0	7	8	4	8	8 ··· 0	2	3	13	-1
row_ind	1	2	4	2	3	5	6	3	4 ··· 5	6	2	5	6

col_ptr	1	4	8	10	13	17	20

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 0 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 0 & 0 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}$$

- All elements are zero by default
- Functions fill all element have been omitted
  - fill(), zeros(), ones()

LUND UNIVERSITY

Sparse Matrix constructors

- Available constructors
  - **sp_mat()**
  - **sp_mat(n_rows, n_cols)**
  - **sp_mat(sp_mat)**
  - **sp_mat(string)**
  - **sp_cx_mat(sp_mat,sp_mat)** (for constructing a complex matrix out of two real matrices)

Example

```
sp_mat A(5,6);
sp_mat B(6,5);

A(0,0) = 1;
A(1,0) = 2;

B(0,0) = 3;
B(0,1) = 4;

sp_mat C = 2*B;

sp_mat D = A*C;
```

Notes on Sparse Matrices

- Support in Armadillo is preliminary
- Not fully optmised
- Supported operations
  - element access
  - fundamental arithmetic operations (such as addition and multiplication)
  - submatrix views
  - **accu(), as_scalar(), dot(), min(), max(), print(), speye(), sprandu()/sprandn(), square(), sum(), trace(), trans()**

Member attributes

- **.n_rows** (number of rows)
- **.n_cols** (number of columns)
- **.n_elem** (total number of elements)
- **.n_slices** (number of slices)
- **.n_nonzero** (number of nonzero elements)
- Member variables are read-only
- Use methods to change size.

Changing size

- **.set_size(n_elem)**
  - (member function of Col, Row, and field)
- **.set_size(n_rows, n_cols)**
  - (member function of Mat and field)
- If the number of elements are the same existing elements are reused
- Memory is not initialised
  - Use **.zeros()** or **.fill()** or similar to initialise memory

Resizing a matrix

- **.resize(n_elem)** (member function of Col, Row)
- **.resize(n_rows, n_cols)** (member function of Mat)
- Recreate the object according to given size specifications, while preserving the elements as well as the layout of the elements
- Can be used for growing or shrinking an object (ie. adding/removing rows, and/or columns, and/or slices)
- Caveat: **.resize()** is slower than **.set_size()**, which doesn't preserve data

Reshaping a matrix

- **.reshape(n_rows, n_cols, dim=0)** (member function of Mat, Col, Row)
- Recreate the object according to given size specifications, with the elements taken from the previous version of the object, either column-wise (dim=0) or row-wise (dim=1);
- Layout of elements will be different
- Use for vectorising a matrix
- Extra elements are set to zero.
- Extra elements are truncated

Setting matrix to zero

- **.zeros()**
- **.zeros(n_elem)**
- **.zeros(n_rows, n_cols)**
- Sets the elements to zero optionally resizing the martrix

Assigning specific values to all elements

- .fill(value)
- Sets all elements to a specific value. Value must match the matrix type.

Creating an identity matrix

- .eye()
- .eye(n_rows, n_cols)
- Sets all elements on the main diagonal to 1 and all off elements to 0
- Optionally resizes the matrix.

Initialising elements

- Instances of Mat, Col, Row and field classes can be initialised via repeated use of the << operator
- Special element endr indicates "end of row" (conceptually similar to std::endl)
- Setting elements via << is a bit slower than directly accessing the elements, but code using << is generally more readable as well as being easier to write

```
mat A;

A << 1 << 2 << 3 << endr
  << 4 << 5 << 6 << endr;
```

LUND UNIVERSITY

Submatrix views

- X.col(col_number)
- X.row(row_number)
- X.cols(first_col, last_col)
- X.rows(first_row, last_row)
- X(span::all, col_number)
- X(span(first_row, last_row), col_number)

LUND UNIVERSITY

Submatrix views - Contiguous

- **X(row_number, span::all)**
- **X(row_number, span(first_col, last_col))**
- **X.submat(first_row, first_col, last_row, last_col)**
- **X.submat(span(first_row, last_row), span(first_col, last_col))**
- **X(span(first_row, last_row), span(first_col, last_col))**
- **V.subvec(first_index, last_index)** (for vectors only)
- **V(span(first_index, last_index))** (for vectors only)

LUND UNIVERSITY

Submatrix views

- Instances of **span::all**, to indicate an entire range, can be replaced by **span()**, where no number is specified
- In the function X.elem(vector_of_indices), elements specified in vector_of_indices are accessed. X is interpreted as one long vector, with column-by-column ordering of the elements of X. The vector_of_indices must evaluate to be a vector of type uvec (eg., generated by find()).

LUND UNIVERSITY

Example of submatrix views

```
mat A = zeros<mat>(5,10);

A.submat(0,1,2,3) = randu<mat>(3,3);

// the following three statements
// access the same part of A
mat B = A.submat(0,1,2,3);
mat C = A.submat( span(0,2), span(1,3) );
mat D = A( span(0,2), span(1,3) );

// the following two statements
// access the same part of A
A.col(1)        = randu<mat>(5,1);
A(span::all, 1) = randu<mat>(5,1);

mat X = randu<mat>(5,5);

// get all elements of X that are greater than 0.5
vec q = X.elem( find(X > 0.5) );

// set four specific elements of X to 1
uvec indices;
indices << 2 << 3 << 6 << 8;

X.elem(indices) = ones<vec>(4);
```

LUND UNIVERSITY

Operate on diagonals

- **.diag(k=0)**
- Member function of Mat
- Read/write access to the k-th diagonal in a matrix
- The argument k is optional -- by default the main diagonal is accessed (k=0)
- For k > 0, the k-th super-diagonal is accessed (top-right corner)
- For k < 0, the k-th sub-diagonal is accessed (bottom-left corner)
- An extracted diagonal is interpreted as a column vector

LUND UNIVERSITY

Example of diagonal access

```
mat X = randu<mat>(5,5);

vec a = X.diag();
vec b = X.diag(1);
vec c = X.diag(-2);

X.diag()  = randu<vec>(5);
X.diag() += 6;
```

Matrix operators

+	Addition of two objects
-	Subtraction of one object from another or negation of an object
/	Element-wise division of an object by another object or a scalar
*	Matrix multiplication of two objects; not applicable to the cube class unless multiplying a cube by a scalar
%	Schur product: element-wise multiplication of two objects
==	Element-wise equality evaluation of two objects; generates a matrix of type umat with entries that indicate whether at a given position the two elements from the two objects are equal (1) or not equal (0)
!=	Element-wise non-equality evaluation of two objects
>=	As for ==, but the check is for "greater than or equal to"
<=	As for ==, but the check is for "less than or equal to"
>	As for ==, but the check is for "greater than"
<	As for ==, but the check is for "less than"

Matrix operators

- If the * operator is chained, Armadillo will try to find an efficient ordering of the matrix multiplications
- If the +, - and % operators are chained, Armadillo will try to avoid the generation of temporaries; no temporaries are generated if all given objects are of the same type and size
- Caveat: operators involving an equality comparison (ie., ==, !=, >=, <=) may not work as expected for floating point element types (ie., float, double) due to the necessarily limited precision of these types; in other words, these operators are (in general) not recommended for matrices of type mat or fmat

LUND UNIVERSITY

Example of matrix operators

```
mat A = randu<mat>(5,10);
mat B = randu<mat>(5,10);
mat C = randu<mat>(10,5);

mat P = A + B;
mat Q = A - B;
mat R = -B;
mat S = A / 123.0;
mat T = A % B;
mat U = A * C;

// V is constructed without temporaries
mat V = A + B + A + B;

imat AA = "1 2 3; 4 5 6; 7 8 9;";
imat BB = "3 2 1; 6 5 4; 9 8 7;";

// compare elements
umat ZZ = (AA >= BB);
```

Transposing matrices

- **.t()**
- **.t()** provides a transposed copy of the object; if a given object has complex elements, a Hermitian transpose is done (ie. the conjugate of the elements is taken during the transpose operation)
- The function **trans(…)** can also be used to return a transpose of a matrix

Inverse of matrix

- **.i()**
- Provides an inverse of the matrix expression
- If the matrix expression is not square, a std::logic_error exception is thrown
- If the matrix expression appears to be singular, the output matrix is reset and a std::runtime_error exception is thrown
- For matrix sizes ≤ 4x4, a fast inverse algorithm is used by default. In rare instances, the fast algorithm might be less precise than the standard algorithm. To force the use of the standard algorithm, set the slow argument to true

Example of the inverse

```
mat A = randu<mat>(4,4);

mat X = A.i();
mat Y = (A+A).i();

mat B = randu<mat>(4,1);
mat Z = A.i() * B; // automatically converted to Z=solve(A,B)
```

Solving equation systems

- **X = solve(A, B, slow=false)**
- **solve(X, A, B, slow=false)**
- Solve a system of linear equations, ie., A*X = B, where X is unknown
- The slow argument is optional
- For a square matrix A, this function is conceptually the same as X = inv(A)*B, but is more efficient
- Similar functionality to the "\" (left division operator) operator in Matlab/Octave, ie. X = A \ B

LUND UNIVERSITY

Solving equation systems

- The number of rows in A and B must be the same
- If A is known to be a triangular matrix, the solution can be computed faster by explicitly marking the matrix as triangular through trimatu() or trimatl()
- If A is non-square (and hence also non-triangular), solve() will also try to provide approximate solutions to under-determined as well as over-determined systems
- If no solution is found, X is reset and:
- solve(A,B) throws a std::runtime_error exception

Solving equation systems

- solve(X,A,B) returns a bool set to false
- For matrix sizes ≤ 4x4, a fast algorithm is used by default. In rare instances, the fast algorithm might be less precise than the standard algorithm. To force the use of the standard algorithm, set the slow argument to true

Example – Solving equation systems

```
mat A = randu<mat>(5,5);
vec b = randu<vec>(5);
mat B = randu<mat>(5,5);

vec x = solve(A, b);
mat X = solve(A, B);

vec x2;
solve(x2, A, b);

// tell solve() to look only at the upper triangular part of A
mat Y = solve( trimatu(A), B );


mat44 C = randu<mat>(4,4);
mat44 D = randu<mat>(4,4);

mat E = solve(C, D);       // use fast algorithm by default
mat F = solve(C, D, true); // use slow algorithm
```

LUND UNIVERSITY

Function applied to elements

- Most normal math functions can be applied elementwise
- abs(), eps(), exp, exp2, exp10, trunc_exp, log, log2, log10, trunc_log, pow, sqrt, square, floor, ceil
- Trigonometric functions:
  - cos family: cos, acos, cosh, acosh
  - sin family: sin, asin, sinh, asinh
  - tan family: tan, atan, tanh, atanh

Scalar valued functions for vectors and matrices

- **accu(mat)**
  - Accumulate (sum) all elements
- **det(A)**
  - Determinant of square matrix A
- **as_scalar(expression)**
  - evaluate an expression that results in a 1x1 matrix.
- **dot(A,B)**
  - dot product of A and B. A and B should be vectors with the same number of elements

Scalar valued functions for vectors and matrices

- norm_dot(A, B)
  - Normalised version of dot
- rank(X)
  - Return the rank of matrix X
- trace(mat)
  - Sum of the diagonal element of a square matrix

Scalar / vector valued functions

- **diagvec(A, k=0)**
  - Extract the k-th diagonal from matrix A
  - columnvector
- **min(mat, dim=0), min(rowvec), min(colvec), max(mat, dim=0), max(rowvec), max(colvec)**
  - For matrix arguments return max / min for each column
  - For vector arguments return max / min

LUND UNIVERSITY

Scalar / vector valued functions

- **prod(mat, dim=0), prod(rowvec), prod(colvec)**
  - For a matrix argument, return the product of elements in each column (dim=0), or each row (dim=1)
  - For a vector argument, return the product of all elements
- **sum(mat, dim=0), sum(rowvec), sum(colvec)**
  - For a matrix argument, return the sum of elements in each column (dim=0), or each row (dim=1)
  - For a vector argument, return the sum of all elements
  - To get a sum of all the elements regardless of the argument type (ie. matrix or vector), you may wish to use accu() instead

Generated matrices

- Random numbers
- **randu(n_elem), randu(n_rows, n_cols), randn(n_elem), randn(n_rows, n_cols)**
- Generate a vector, matrix or cube with the elements set to random values
- **randu()** uses a uniform distribution in the [0,1] interval
- **randn()** uses a normal/Gaussian distribution with zero mean and unit variance
- To change the seed, use the **std::srand()** function.

Exampe – random numbers

```
vec  v = randu<vec>(5);
mat  A = randu<mat>(5,6);
cube Q = randu<cube>(5,6,7);
```

Reading and writing matrices

- The Armadillo library supports storing matrices in many different file formats.
  - binary, ascii, arma_ascii, arma_binary, csv_ascii, hdf5_binary, pgm_binary, ppm_binary
- Writing to file
  - .save(name, file_type = arma_binary)
  - .save(stream, file_type = arma_binary)
- Reading from file
  - .load(name, file_type = auto_detect)
  - .load(stream, file_type = auto_detect)

Example – Reading and writing

```
mat A = randu<mat>(5,5);

A.save("A1.mat");  // default save format is arma_binary
A.save("A2.mat", arma_ascii);

mat B;
// automatically detect format type
B.load("A1.mat");

mat C;
// force loading in the arma_ascii format
C.load("A2.mat", arma_ascii);


// example of saving/loading using a stream
std::stringstream s;
A.save(s);

mat D;
D.load(s);


// example of testing for success
mat E;
bool status = E.load("A2.mat");

if(status == true)
{
    cout << "loaded okay" << endl;
    }
    else
    {
        cout << "problem with loading" << endl;
    }
```

LUND UNIVERSITY

Accessing raw pointer to storage

- **.memptr()**
- Obtain a raw pointer to the memory used for storing elements. Not recommended for use unless you know what you're doing!
- The function can be used for *interfacing with libraries* such as FFTW/MPI
- As soon as the size of the matrix/vector/cube is changed, the pointer is no longer valid
- Data for matrices is stored in a column-by-column order

Example - raw pointer

```
mat A = randu<mat>(5,5);
const mat B = randu<mat>(5,5);

double* A_mem = A.memptr();
const double* B_mem = B.memptr();
```

A real example

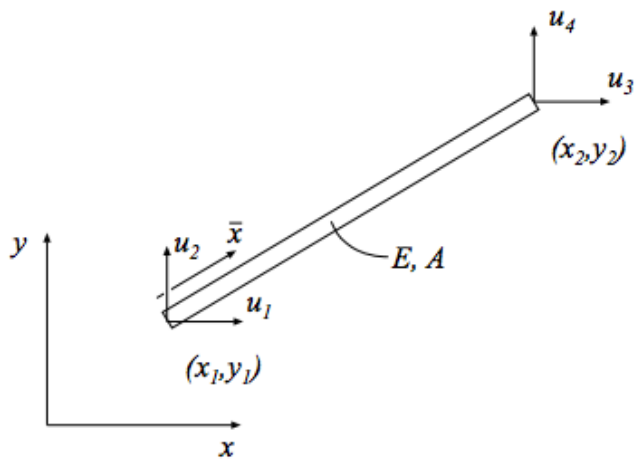
Example from the CALFEM toolbox

- CALFEM is a MATLAB toolbox
- Function-oriented Finite Element library
- Implementing simple element routines
- Implementing a larger example problem

Implementing bar2e/s

Purpose:

Compute element stiffness matrix for a two dimensional bar element.



Syntax:

Ke=bar2e(ex,ey,ep)

Description:

bar2e provides the global element stiffness matrix Ke for a two dimensional bar element.

The input variables

$$\begin{matrix} \mathsf{ex} = [\ x_1 & x_2\ ] \\ \mathsf{ey} = [\ y_1 & y_2\ ] \end{matrix} \qquad \mathsf{ep} = [\ E\ A\ ]$$

supply the element nodal coordinates x_1, y_1, x_2, and y_2, the modulus of elasticity E, and the cross section area A.

Theory:

The element stiffness matrix $\mathbf{K}^e$, stored in Ke, is computed according to

$$\mathbf{K}^e = \mathbf{G}^T \, \bar{\mathbf{K}}^e \, \mathbf{G}$$

where

$$\bar{\mathbf{K}}^e = \frac{EA}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad \mathbf{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & 0 & 0 \\ 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} \end{bmatrix}$$

The transformation matrix $\mathbf{G}$ contains the direction cosines

$$n_{x\bar{x}} = \frac{x_2 - x_1}{L} \qquad n_{y\bar{x}} = \frac{y_2 - y_1}{L}$$

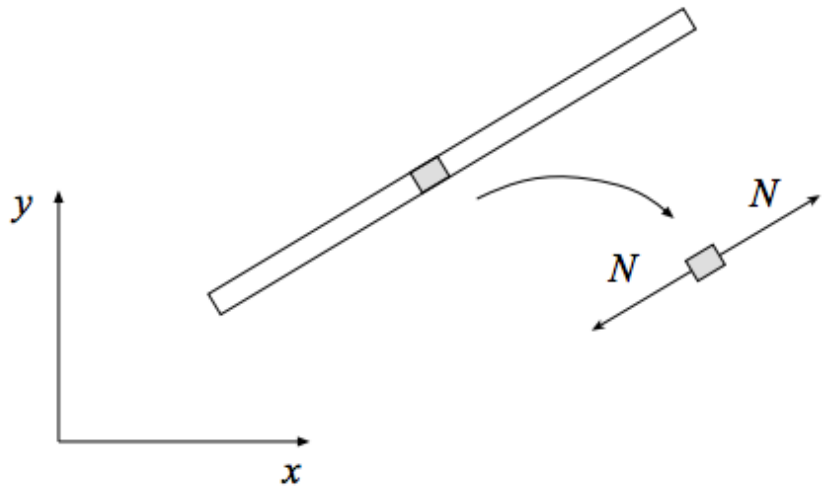
where the length

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
arma::mat bar2e(arma::rowvec ex, arma::rowvec ey, arma::rowvec ep)
{
    using namespace arma;

    double E = ep(0);
    double A = ep(1);
    double L = sqrt(pow(ex(1)-ex(0),2)+pow(ey(1)-ey(0),2));
    double C = E*A/L;

    mat Ke_loc(2,2);

    Ke_loc <<  C << -C << endr
           << -C <<  C << endr;

    double nxx = (ex(1)-ex(0))/L;
    double nyx = (ey(1)-ey(0))/L;

    mat G(2,4);

    G << nxx << nyx << 0.0 << 0.0 << endr
      << 0.0 << 0.0 << nxx << nyx << endr;

    mat Ke = G.t()*Ke_loc*G;
    return Ke;
}
```

Purpose:

Compute normal force in a two dimensional bar element.



Syntax:

es=bar2s(ex,ey,ep,ed)

Description:

bar2s computes the normal force in the two dimensional bar elements bar2e and bar2g.

The input variables ex, ey, and ep are defined in bar2e and the element nodal displacements, stored in ed, are obtained by the function extract.

The output variable

$$\text{es} = [\ N\ ]$$

contains the normal force N.

Theory:

The normal force N is computed from

$$N = \frac{EA}{L}\begin{bmatrix} -1 & 1 \end{bmatrix} \mathbf{G}\,\mathbf{a}^e$$

where E, A, L, and the transformation matrix $\mathbf{G}$ are defined in **bar2e**. The nodal displacements in global coordinates

$$\mathbf{a}^e = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 \end{bmatrix}^T$$

are also shown in **bar2e**. Note that the transpose of $\mathbf{a}^e$ is stored in **ed**.

LUND
UNIVERSITY

```
double bar2s(arma::rowvec ex, arma::rowvec ey, arma::rowvec ep,
arma::rowvec ed)
{
    using namespace arma;

    double E = ep(0);
    double A = ep(1);
    double L = sqrt(pow(ex(1)-ex(0),2)+pow(ey(1)-ey(0),2));
    double C = E*A/L;

    double nxx = (ex(1)-ex(0))/L;
    double nyx = (ey(1)-ey(0))/L;

    mat G(2,4);

    G << nxx << nyx << 0.0 << 0.0 << endr
      << 0.0 << 0.0 << nxx << nyx << endr;

    rowvec temp;
    temp << -C << C;

    return as_scalar(temp * G * ed.t());
}
```

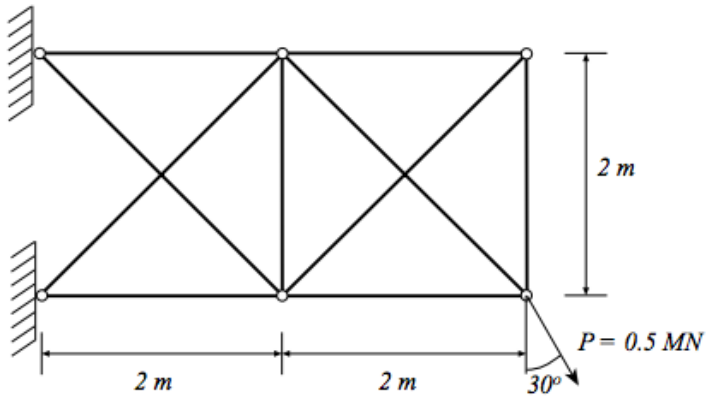
A sample problem exs4

Purpose:

Analysis of a plane truss.

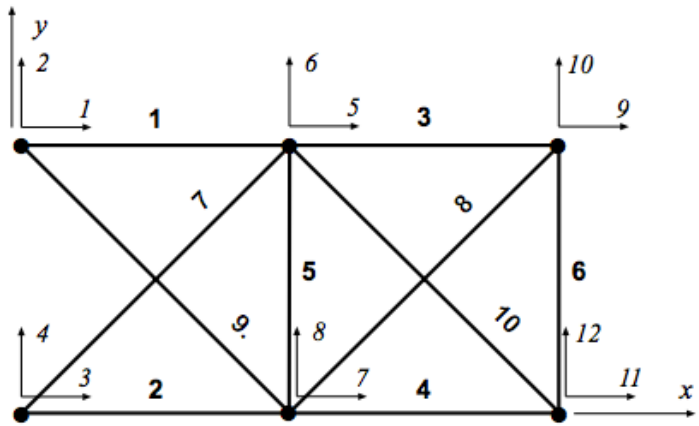
Description:

Consider a plane truss, loaded by a single force $P = 0.5$ MN.



$A = 25.0 \cdot 10^{-4}$ m^2
$E = 2.10 \cdot 10^{5}$ MPa

The corresponding finite element model consists of ten elements and twelve degrees of freedom.



MATLAB code:

```
Edof=[1   1  2  5  6;
      2   3  4  7  8;
      3   5  6  9 10;
      4   7  8 11 12;
      5   7  8  5  6;
      6  11 12  9 10;
      7   3  4  5  6;
      8   7  8  9 10;
      9   1  2  7  8;
     10   5  6 11 12];
```

C++ code:

```
imat edof(10,4);

edof << 1 << 2 << 5 << 6 << endr
     << 3 << 4 << 7 << 8 << endr
     << 5 << 6 << 9 << 10 << endr
     << 7 << 8 << 11 << 12<< endr
     << 7 << 8 << 5 << 6 << endr
     << 11 << 12 << 9 << 10 << endr
     << 3 << 4 << 5 << 6 << endr
     << 7 << 8 << 9 << 10 << endr
     << 1 << 2 << 7 << 8 << endr
     << 5 << 6 << 11 << 12 << endr;

edof -= 1;
```

Indices in Armadillo starts at 0 (zero).
We apply the compound operator -= to decrease all elements with 1.

LUND UNIVERSITY

MATLAB code:

```
K=zeros(12);
f=zeros(12,1);  f(11)=0.5e6*sin(pi/6);  f(12)=-0.5e6*cos(pi/6);
```

C++ code:

```
// Stiffness matrix

mat K(12,12);
K.zeros();

// Force vector

mat f(12,1);
f.zeros();
f(10,0) = 0.5e6*sin(M_PI/6);
f(11,0) = -0.5e6*cos(M_PI/6);
```

LUND UNIVERSITY

MATLAB code:

```
A=25.0e-4;    E=2.1e11;    ep=[E A];

Ex=[0 2;
    0 2;
    2 4;
    2 4;
    2 2;
    4 4;
    0 2;
    2 4;
    0 2;
    2 4];

Ey=[2 2;
    0 0;
    2 2;
    0 0;
    0 2;
    0 2;
    0 2;
    0 2;
    2 0;
    2 0];
```

C++ code:

```
// Material properties

double A = 25.0e-4;
double E = 2.1e11;

rowvec ep(2);

ep << E << A;

// Element coordinates

mat ex(10,2);
mat ey(10,2);

ex << 0 << 2 << endr
   << 0 << 2 << endr
   << 2 << 4 << endr
   << 2 << 4 << endr
   << 2 << 2 << endr
   << 4 << 4 << endr
   << 0 << 2 << endr
   << 2 << 4 << endr
   << 0 << 2 << endr
   << 2 << 4 << endr;

ey << 2 << 2 << endr
   << 0 << 0 << endr
   << 2 << 2 << endr
   << 0 << 0 << endr
   << 0 << 2 << endr
   << 0 << 2 << endr
   << 0 << 2 << endr
   << 0 << 2 << endr
   << 2 << 0 << endr
   << 2 << 0 << endr;
```

LUND UNIVERSITY

MATLAB code:

```
for i=1:10
   Ke=bar2e(Ex(i,:),Ey(i,:),ep);
   K=assem(Edof(i,:),K,Ke);
end;
```

C++ code:

```
for (int i=0; i<ex.n_rows; i++)
{
    mat Ke = bar2e(ex.row(i), ey.row(i), ep);
    assem(edof.row(i), K, Ke);
}
```

Pass all larger matrices by reference to avoid copying

```
void assem(arma::Mat<int>& topo, arma::mat& K, arma::mat& Ke)
{
    for (int row=0; row<Ke.n_rows; row++)
        for (int col=0; col<Ke.n_cols; col++)
            K(topo(row), topo(col)) += Ke(row,col);
}
```

Increase efficiency by using compount operators

MATLAB code:

```
bc=[1 0;2 0;3 0;4 0];
[a,r]=solveq(K,f,bc)
```

C++ code:

```
// Boundary conditions

irowvec bcDofs(4);
bcDofs << 0 << 1 << 2 << 3;

rowvec bcValues(4);
bcValues << 0.0 << 0.0 << 0.0 << 0.0;

// Solution displacment and reaction vector

mat a(K.n_rows, 1);
mat r(K.n_rows, 1);

// Solve equation system

solveq(K, f, bcDofs, bcValues, a, r);
```

LUND UNIVERSITY

```
void solveq(arma::mat& K, arma::mat&f, arma::irowvec& bcDofs, arma::rowvec& bcValues,
arma::mat& a, arma::mat& r)
```