ETH Lecture 401-0663-00L Numerical Methods for CSE

Numerical Methods for Computational Science and Engineering

Prof. R. Hiptmair, SAM, ETH Zurich

(with contributions from Prof. P. Arbenz and Dr. V. Gradinaru)

Autumn Term 2015

URL: http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE/NumCSE15.pdf

Introduction

- Use the study center HG E 41, Mon from 18:00
- A toolbox course → many different topics, that are only loosely related

[toolbox contains hammer, screwdriver, duct tape]

I. Computing with Matrices and Vectors

(some aspects of $\underbrace{\text{numerical linear algebra}}_{\text{foundation of all numerical codes}}$)

I.1. Fundamentals

I.1.1. Notations:

Vector $\underline{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{K}^n$, $\mathbb{K} = \mathbb{R}, \mathbb{C}$ (default: column vector)

Matrix $A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \in \mathbb{K}^{m,n}$

row vector $v^T = [v_1 \cdots v_n]$

↳ just special matrices $\in \mathbb{K}^{1,n}$

Components/entries: $(v)_i := v_i$, $(A)_{i,j} := a_{ij}$

sub-vector/matrix blocks: $(v)_{k:\ell} = \begin{bmatrix} v_k \\ \vdots \\ v_\ell \end{bmatrix}$, $1 \leq k \leq \ell \leq n$

$(A)_{k:\ell, r:s} \in \mathbb{K}^{\ell-k+1, s-r+1}$

$\mathbf{A} := \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}$

→ entry $(\mathbf{A})_{i,j} = a_{ij}$, $1 \leq i \leq n, 1 \leq j \leq m$,

→ i-th row, $1 \leq i \leq n$: $a_{i,:} = (\mathbf{A})_{i,:}$,

→ j-th column, $1 \leq j \leq m$: $a_{i,:} = (\mathbf{A})_{:,j}$,

→ matrix block (sub-matrix) $(a_{ij})_{\substack{i=k,\dots,l \\ j=r,\dots,s}} = (\mathbf{A})_{k:l,r:s}$, $1 \leq k \leq l \leq n$, $1 \leq r \leq s \leq m$.

1.2. Software and Libraries

→ Always rely on them for numerical linear algebra

1.2.1. MATLAB

↳ IDE for numerical computation (numerical engine, libraries, editor, debugger, profiler, help)

"In MATLAB everything is a matrix"

(Fundamental "data type" in MATLAB = matrix of complex numbers)

↳ non-typed language

[m,n] = size(A) → request dimensions of matrix

A(i,j), v(i) → entry access

Initialization of matrix:

, ⟺ horizontal
; ⟺ vertical
} concatenation

Example:

```
% Caution: matrices are dynamically
  expanded when
% out of range entries are accessed
M = [1,2,3;4,5,6]; M(4,6) = 1.0; M,
```

Output: M=

1	2	3	0	0	0
4	5	6	0	0	0
0	0	0	0	0	0
0	0	0	0	0	1

"loop index vectors" $v = (a:s:b)$

```
>> v = (3:-0.5:-0.3)
v =   3.0000    2.5000    2.0000    1.5000    1.0000    0.5000         0
>> v = (1:2.5:-13)
v = Empty matrix: 1-by-0
```

⇒ loop: for $i = (a:s:b)$

```
% MATLAB loop over columns of a matrix
M = [1,2,3;4,5,6];
for i = M; i, end
```

Output:

```
i = 1    i = 2    i = 3
    4        5        6
```

1.2.2. Eigen

= Header-only C++ library for numerical algebra,
→ template metaprogramming
→ expression templates

Fundamental data type : matrix

```
Matrix<typename Scalar, int RowsAtCompileTime, int
    ColsAtCompileTime>
```

for specifying small fixed size matrices

Special data types : MatrixX*, $* \in \{i, f, d, cd\}$

```
#include <Eigen/Dense >

template<typename Scalar>
void eigenTypeDemo(unsigned int dim)
{
  using dynMat_t =
     Eigen::Matrix<Scalar,Eigen::Dynamic,Eigen::Dynamic>;
  using dynColVec_t = Eigen::Matrix<Scalar,Eigen::Dynamic,1>;
  using dynRowVec_t = Eigen::Matrix<Scalar,1,Eigen::Dynamic>;
  using index_t = typename dynMat_t::Index;
  using entry_t = typename dynMat_t::Scalar;

  dynColVec_t colvec(dim);
  dynRowVec_t rowvec(dim);
  for(index_t i=0; i< colvec.size(); ++i) colvec(i) = (Scalar)i;
  for(index_t i=0; i< rowvec.size(); ++i) rowvec(i) =
     (Scalar)1/(i+1);
  dynMat_t vecprod = colvec*rowvec;  ← matrix product
  const int nrows = vecprod.rows();
  const int ncols = vecprod.cols();
}
```

Initialization of matrices :

```
#include <Eigen/Dense >
// Just allocate space for matrix, no initialisation
Eigen::MatrixXd A(rows,cols);  ← reserve space
// Zero matrix.  Similar to matlab command zeros(rows, cols);
Eigen::MatrixXd B = MatrixXd::Zero(rows, cols);
// Ones matrix.  Similar to matlab command ones(rows, cols);
Eigen::MatrixXd C = MatrixXd::Ones(rows, cols);
// Matrix with all entries same as value.
Eigen::MatrixXd D = MatrixXd::Constant(rows, cols, value);
// Random matrix, entries uniformly distributed in [0,1]
Eigen::MatrixXd E = MatrixXd::Random(rows, cols);
// (Generalized) identity matrix, 1 on main diagonal
Eigen::MatrixXd I = MatrixXd::Identity(rows,cols);
std::cout << "size of A = (" << A.rows() << ',' << A.cols() << ')'
   << std::endl;
```

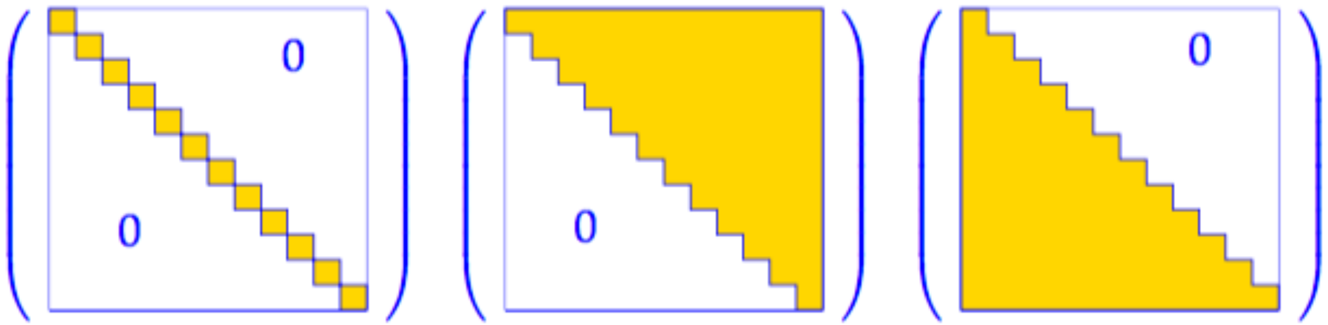
Addressing sub-matrices :

```
template<typename MatType> void
  blockAccess(Eigen::MatrixBase<MatType> &M)
{
  using index_t = typename Eigen::MatrixBase<MatType>::Index;
  using entry_t = typename Eigen::MatrixBase<MatType>::Scalar;
  const index_t nrows(M.rows()); % No. of rows
  const index_t ncols(M.cols()); % No. of columns

  cout << "Matrix M = " endl << M << endl; // Print matrix
  // Block size half the size of the matrix
  index_t p = nrows/2,q = ncols/2;
  // Output submatrix with left upper entry at position (i,i)
  for(index_t i=0; i < min(p,q); i++)
    cout << "Block (" << i << ',' << i << ',' << p << ',' << q
         << ") = " << M.block(i,i,p,q) << endl;
  // l-value access: modify sub-matrix by adding a constant
  M.block(1,1,p,q) += MatrixXd::Constant(p,q,1.0);
  cout << "M = " endl << M << endl;
  // r-value access: extract sub-matrix
  MatrixXd B(M.block(1,1,p,q));
  cout << "Isolated modified block = " endl << B << endl;
  // Special sub-matrices
  cout << p << " top rows of m = " << M.topRows(p) << endl;
  cout << p << " bottom rows of m = " << M.bottomRows(p) << endl;
  cout << q << " left cols of m = " << M.leftCols(q) << endl;
  cout << q << " right cols of m = " << M.rightCols(p) << endl;
  // r-value access to upper triangular part
  const MatrixXd T = M.template triangularView<Upper>(); //
  cout << "Upper triangular part = " << endl << T << endl;
  // l-value access to upper triangular part
  M.template triangularView<Lower>() *= -1.5; //
  cout << "Matrix M = " << endl << M << endl;
}
```

In Eigen : indexing from 0 !

Triangular matrices :



diagonal matrix   upper triangular   lower triangular

1.2.3. Matrix storage formats

→ Matrix stored in linearized form (as a vector)

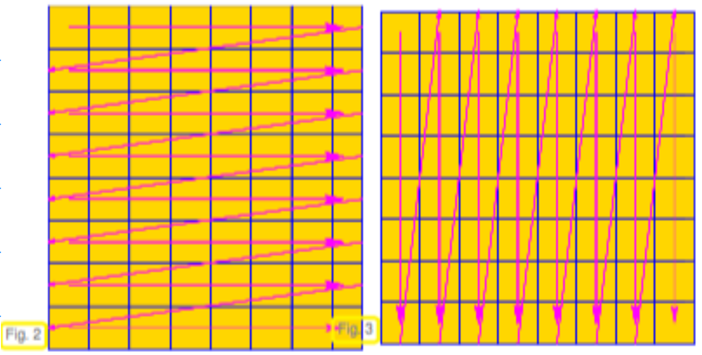
$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Row major (C-arrays, bitmaps, Python):

A_arr	1	2	3	4	5	6	7	8	9

Column major (Fortran, MATLAB, EIGEN):

A_arr	1	4	7	2	5	8	3	6	9



row major   column major

Index mappings $A \in \mathbb{K}^{n,m}$

r.m. : $(i,j) \leftrightarrow m(i-1)+j$

```
// Template parameter ColMajor selects column major data layout
Matrix<double,Dynamic,Dynamic,ColMajor> mcm(nrows,ncols);
```
→ default

```
// Template parameter RowMajor selects row major data layout
Matrix<double,Dynamic,Dynamic,RowMajor> mrm(nrows,ncols);
```

Why care ? → Calling library functions

Exp.: Impact of storage format on runtime

1.2.20

Matlab : Column major

```
A = randn(n,n);
for j = 1:n-1,
  A(:,j+1) = A(:,j+1) - A(:,j);
end
```
column oriented access

↓

faster, because contiguous access to memory

```
A = randn(n,n);
for i = 1:n-1,
  A(i+1,:) = A(i+1,:) - A(i,:);
end
```
row oriented access

↓

many cache misses