①

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 2016

URL: http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE/NumCSE16.pdf

I. Computing with Matrices and Vectors

I.1. Fundamentals

I.1.1. Notation

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

column

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

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

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

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

I. Libraries

↳ Always use them for numerical linear algebra computations

1.2.3. Eigen

↳ Header-only C++ template library for numerical LA

Fundamental data type = matrix

Eigen::Matrix< typename Scalar, int nrows, int ncols >

fixed size matrices — dynamic matrices

C++11 code 1.2.12: Vector type and their use in EIGEN

```
#include <Eigen/Dense >

template<typename Scalar>
void eigenTypeDemo(unsigned int dim)
{
  // General dynamic (variable size) matrices
  using dynMat_t =
      Eigen::Matrix<Scalar,Eigen::Dynamic,Eigen::Dynamic>;
  // Dynamic (variable size) column vectors
  using dynColVec_t = Eigen::Matrix<Scalar,Eigen::Dynamic,1>;
  // Dynamic (variable size) row vectors
  using dynRowVec_t = Eigen::Matrix<Scalar,1,Eigen::Dynamic>;
  using index_t = typename dynMat_t::Index;
  using entry_t = typename dynMat_t::Scalar;

  // Declare vectors of size 'dim'; not yet initialized
  dynColVec_t colvec(dim);
  dynRowVec_t rowvec(dim);
  // Initialisation through component access
  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);
  colvec[0] = (Scalar)3.14; rowvec[dim-1] = (Scalar)2.718;
  // Form tensor product, a matrix, see Section 1.3.1
  dynMat_t vecprod = colvec*rowvec;
  const int nrows = vecprod.rows();
  const int ncols = vecprod.cols();
}
```

} create vector objects

[Matrix product]

②

Component Access operator (int), (int, int), [Both Lvalue Rvalue]

→ Indices from 0!

Line 23 :

tensor product

$$\begin{bmatrix} \\ \\ \end{bmatrix} \cdot \begin{bmatrix} \quad \end{bmatrix} = \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

$\underline{a}, \underline{b} \in \mathbb{R}^n$: $(\underline{a}\,\underline{b}^T)_{i,j} = a_i b_j$

Convenience data types : $\underbrace{\text{MatrixX}*}_{\text{dynamic matrix}}$, $* \in \{i, f, d, cd\}$

VectorX* ≙ dynamic column vector

C++11 code 1.2.14: Initializing special matrices in EIGEN

```
#include <Eigen/Dense>
// Just allocate space for matrix, no initialisation
Eigen::MatrixXd A(rows,cols);
// 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;
```

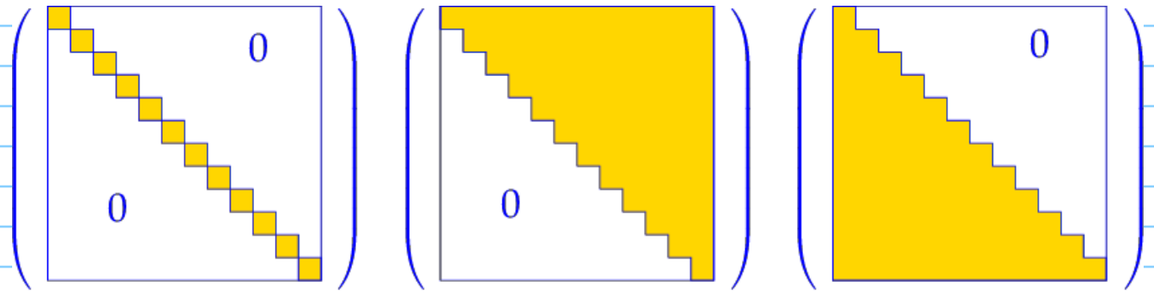
C++11 code 1.2.16: Demonstration code for access to matrix blocks in EIGEN

```
2  template<typename MatType>
3  void blockAccess(Eigen::MatrixBase<MatType> &M)
4  {
5    using index_t = typename Eigen::MatrixBase<MatType>::Index;
6    using entry_t = typename Eigen::MatrixBase<MatType>::Scalar;
7    const index_t nrows(M.rows()); // No. of rows
8    const index_t ncols(M.cols()); // No. of columns
9
10   cout << "Matrix M = " << endl << M << endl; // Print matrix
11   // Block size half the size of the matrix
12   index_t p = nrows/2, q = ncols/2;
13   // Output submatrix with left upper entry at position (i,i)
14   for(index_t i=0; i < min(p,q); i++)
15     cout << "Block (" << i << ',' << i << ',' << p << ',' << q
16 [read]   << ") = " << M.block(i,i,p,q) << endl;
17   // l-value access: modify sub-matrix by adding a constant
[write] M.block(1,1,p,q) += Eigen::MatrixBase<MatType>::Constant(p,q,1.0);
19   cout << "M = " << endl << M << endl;
20   // r-value access: extract sub-matrix
21   MatrixXd B = M.block(1,1,p,q);
22   cout << "Isolated modified block = " << endl << B << endl;
23   // Special sub-matrices
24   cout << p << " top rows of m = " << M.topRows(p) << endl;
25   cout << p << " bottom rows of m = " << M.bottomRows(p) << endl;
26   cout << q << " left cols of m = " << M.leftCols(q) << endl;
27   cout << q << " right cols of m = " << M.rightCols(p) << endl;
28   // r-value access to upper triangular part
29   const MatrixXd T = M.template triangularView<Upper>(); //
30   cout << "Upper triangular part = " << endl << T << endl;
31   // l-value access to upper triangular part
32   M.template triangularView<Lower>() *= -1.5; //
33   cout << "Matrix M = " << endl << M << endl;
34 }
```

Submatrix : upper left corner at (i,i) & size p×q

MATLAB triu

③ Triangular matrices :



diagonal matrix | upper triangular | lower triangular

1.2.4. (Dense) Matrix Storage Formats

→ stored in one contiguous array

$A \in \mathbb{K}^{m,n} \Leftrightarrow$ Array of length $m \cdot n$

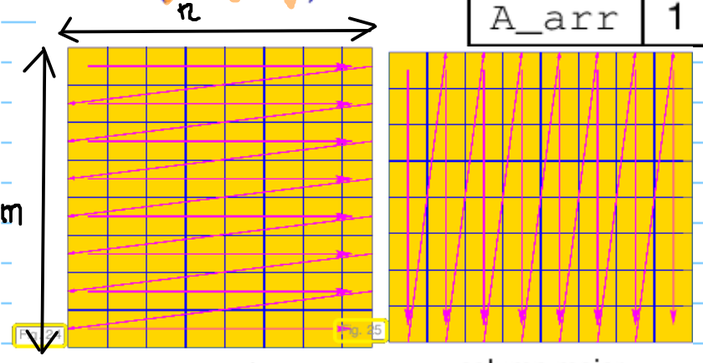
$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

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 :

row major : [indexing from 0]

$(i,j) \longrightarrow i \cdot n + j$

for $A \in \mathbb{R}^{m,n}$

C++11 code 1.2.22: Single index access of matrix entries in EIGEN

```
void storageOrder(int nrows=6,int ncols=7)
{
  cout << "Different matrix storage layouts in Eigen" << endl;
  // Template parameter ColMajor selects column major data layout
  Matrix<double,Dynamic,Dynamic,ColMajor> mcm(nrows,ncols);
  // Template parameter RowMajor selects row major data layout
  Matrix<double,Dynamic,Dynamic,RowMajor> mrm(nrows,ncols);
  // Direct initialization; lazy option:  use int as index type
  for (int l=1,i= 0; i< nrows; i++)
    for (int j= 0; j< ncols; j++,l++)
      mcm(i,j) = mrm(i,j) = l;

  cout << "Matrix mrm = " << endl << mrm << endl;
  cout << "mcm linear = ";
  for (int l=0;l < mcm.size(); l++) cout << mcm(l) << ',';
  cout << endl;
                                        Matrix-as-array access
  cout << "mrm linear = ";
  for (int l=0;l < mrm.size(); l++) cout << mrm(l) << ',';
  cout << endl;
}
```

Application : "reshaping" ⇔ reinterpretation of matrix array data

Ex : Vectorization

$$A \in \mathbb{R}^{m,n} : \quad \mathrm{vec}(A) = \begin{bmatrix} (A)_{:,1} \\ (A)_{:,2} \\ \vdots \\ (A)_{:,n} \end{bmatrix}$$

C++11 code 1.2.28: Demonstration on how reshape a matrix in EIGEN

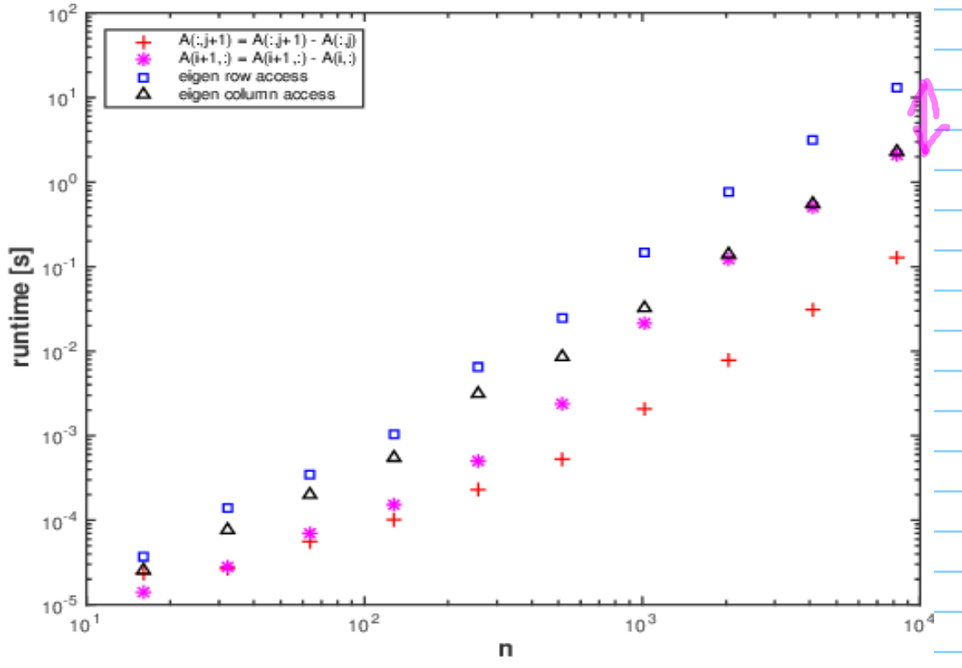
```
template<typename MatType>
void reshapetest(MatType &M)
{
  using index_t = typename MatType::Index;
  using entry_t = typename MatType::Scalar;
  const index_t nsize(M.size());

 // reshaping possible only for matrices with non-prime dimensions
  if ((nsize %2) == 0) {
    entry_t *Mdat = M.data(); // raw data array for M
    // Reinterpretation of data of M
    Map<Eigen::Matrix<entry_t,Dynamic,Dynamic>> R(Mdat,2,nsize/2);
    // (Deep) copy data of M into matrix of different size
    Eigen::Matrix<entry_t,Dynamic,Dynamic> S =
      Map<Eigen::Matrix<entry_t,Dynamic,Dynamic>>(Mdat,2,nsize/2);

    cout << "Matrix M = " << endl << M << endl;
    cout << "reshaped to " << R.rows() << 'x' << R.cols()
      << " = " << endl << R << endl;
    // Modifying R affects M, because they share the data space !
    R *= -1.5;
    cout << "Scaled (!) matrix M = " << endl << M << endl;
    // Matrix S is not affected, because of deep copy
    cout << "Matrix S = " << endl << S << endl;
  }
```

← C-pointer

```
Matrix M =
 0 -1 -2 -3 -4 -5 -6
 1  0 -1 -2 -3 -4 -5
 2  1  0 -1 -2 -3 -4
 3  2  1  0 -1 -2 -3
 4  3  2  1  0 -1 -2
 5  4  3  2  1  0 -1
reshaped to 2x21 =
 0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1 -6 -4 -2
 1  3  5  0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1
Scaled (!) matrix M =
   -0  1.5    3  4.5    6  7.5    9
 -1.5   -0  1.5    3  4.5    6  7.5
   -3 -1.5   -0  1.5    3  4.5    6
 -4.5   -3 -1.5   -0  1.5    3  4.5
   -6 -4.5   -3 -1.5   -0  1.5    3
 -7.5   -6 -4.5   -3 -1.5   -0  1.5
Matrix S =
 0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1 -6 -4 -2
 1  3  5  0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1
```

Data layout important for runtime performance!

Ex: $A.\overset{row}{col}(j+1) -= A.\overset{row}{col}(j)$



Contiguous memory column access (for Colum Major storage layout) much faster: cache misses for row access.

1.4. Computational Effort

↳ No. of elementary operations '+', '-', '*', '/'

"Computational effort $\not\sim$ runtime"

The computational effort involved in a run of a numerical code is only loosely related to overall execution time on modern computers.

↓

Mainly determined by memory access patterns

1.4.1. Asymptotic Complexity

Definition 1.4.4. (Asymptotic) complexity

The asymptotic complexity of an algorithm characterises the worst-case dependence of its computational effort on one or more problem size parameter(s) when these tend to ∞.

↑ vector length, matrix dimensions

Notation : "Landau - O", e.g.

(*) $\quad \text{Cost}(n) = O(n^\alpha) \quad , \alpha > 0$

Strictly : $\Rightarrow \exists C > 0, n_0 \in \mathbb{N} : \quad \text{Cost}(n) \leq C n^\alpha \ \forall n \geq n_0$

tacit assumption of "sharpness" :

(*) also means : $\text{Cost}(n) \neq O(n^\beta) \ \forall \beta < \alpha$

Asymptotic complexity $\not\Rightarrow$ runtime

$\Downarrow$

dependence of runtime on problem size for large problems

e.g. : $\text{cost}(n) = O(n^2)$

$\Rightarrow \quad n \to n*2 \quad : \quad \text{cost} \times 4$

Telling (polynomial) asymptotic complexity from runtime measurements

Conjecture : $\quad t_i \approx C n_i^\alpha \quad , i = 1, \dots, N$

↑ runtimes ↑ problem sizes

$$\underbrace{\log t_i \approx \log C + \alpha \log n_i}$$

Data points (n_i, t_i) roughly on a straight line with slope α in doubly logarithmic plot.

1.4.2. Cost of basic operations

operation	description	#mul/div	#add/sub	asymp. complexity
dot product	$(\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}^H\mathbf{y}$	n	$n-1$	$O(n)$
tensor product	$(\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}\mathbf{y}^H$	nm	0	$O(mn)$
matrix product(*)	$(\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{B} \in \mathbb{R}^{n,k}) \mapsto \mathbf{AB}$	mnk	$mk(n-1)$	$O(mnk)$

↳ three loop implementation

1.4.3 Some tricks to reduce complexity

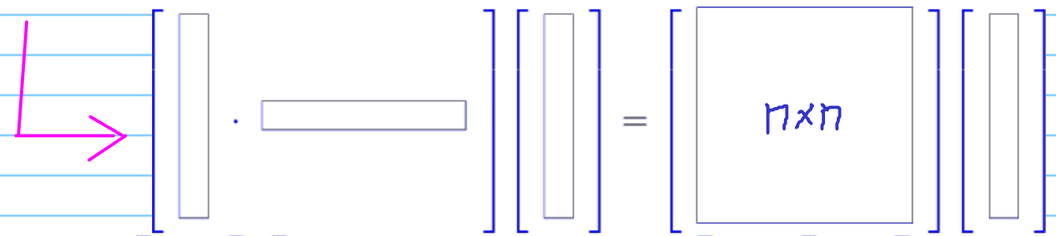
Ex: Exploit associativity

$\underline{a} \in \mathbb{R}^n$, $\underline{b} \in \mathbb{R}^n$, $\underline{x} \in \mathbb{R}^n$

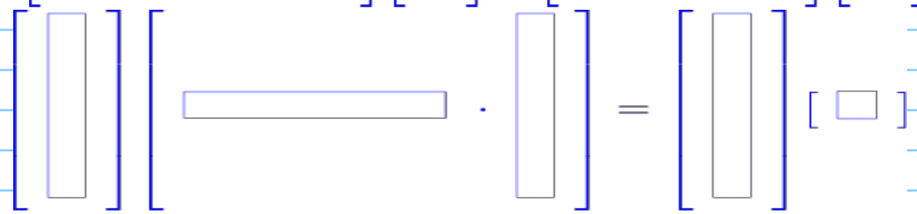
$$\mathbf{y} = \left(\mathbf{a}\mathbf{b}^\top\right)\mathbf{x}\,. \qquad (1.4.12) \qquad\qquad \mathbf{y} = \mathbf{a}\left(\mathbf{b}^\top\mathbf{x}\right)\,. \qquad (1.4.13)$$

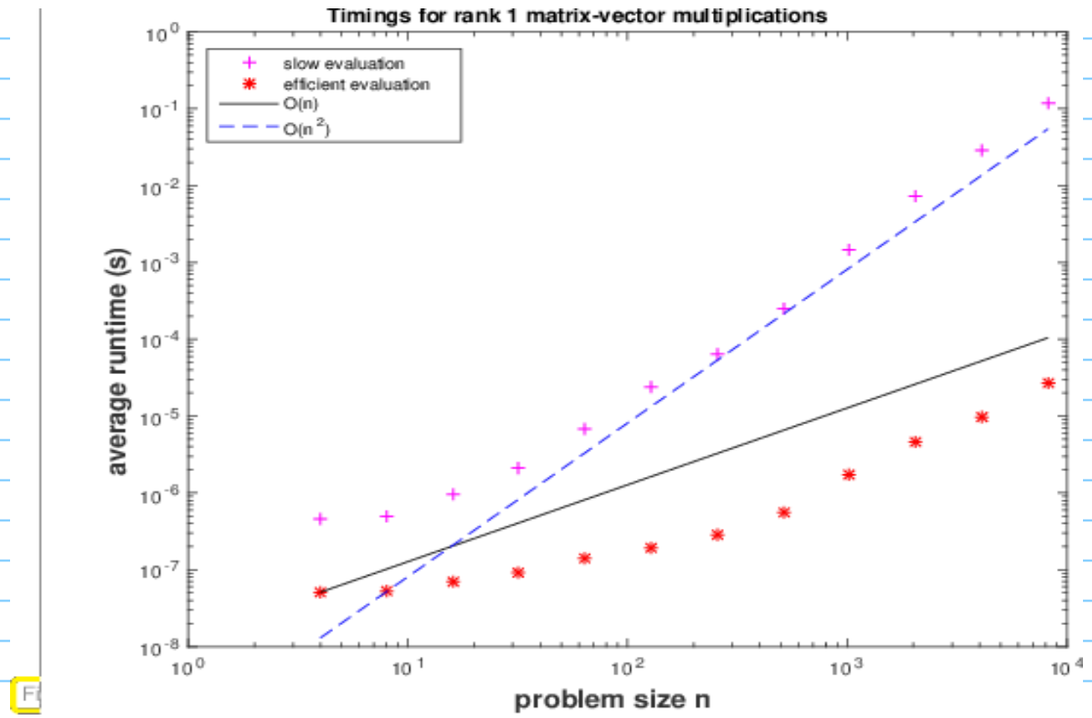
```
T = (a*b.transpose())*x;
```

```
t = a*b.dot(x);
```



(1.4.13):





Ex: "Hidden summation"

$A \in \mathbb{R}^{n,p}$, $B \in \mathbb{R}^{n,p}$, $p \ll n$

```
Eigen::MatrixXd AB = A*B.transpose();
y = AB.triangularView<Eigen::Upper>()*x;
```

$\underline{y} = \mathrm{triu}(A*B^T)$

$\Rightarrow$ Cost $= O(n^2)$ for $n \to \infty$, p fixed

$p = 1$:

$$\mathbf{y} = \mathrm{triu}(\mathbf{a}\mathbf{b}^T)\mathbf{x} = \begin{bmatrix} a_1b_1 & a_1b_2 & \dots & & \dots & a_1b_n \\ 0 & a_2b_2 & a_2b_3 & \dots & \dots & a_2b_n \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \dots & & \dots & 0 & a_nb_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \\ \\ \vdots \\ x_n \end{bmatrix}$$

$$= \underbrace{\underbrace{\begin{bmatrix} a_1 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & a_n \end{bmatrix} \underbrace{\begin{bmatrix} 1 & 1 & \dots & & \dots & 1 \\ 0 & 1 & 1 & \dots & \dots & 1 \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \dots & & \dots & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \underbrace{\begin{bmatrix} b_1 & & \\ & \ddots & \\ & & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \\ \\ \vdots \\ x_n \end{bmatrix}}_{\text{componentwise mult.}}}_{\text{cumulative summation}}}_{\text{componentwise mult.}}$$

Cost $= O(n)$ $\Downarrow$ Cost $= O(n)$ $[\cdot p]$

$p > 1$: $AB^T = \sum_{\ell=1}^{p} (A)_{:,\ell} (B)_{:,\ell}^T$

$y = \mathrm{triu}(AB^T)\underline{x} = \sum_{\ell=1}^{p} \mathrm{triu}((A)_{:,\ell}(B)_{:,\ell}^T)\underline{x}$

Ex : Kronecker product

Definition 1.4.17. Kronecker product

The Kronecker product $\mathbf{A} \otimes \mathbf{B}$ of two matrices $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{l,k}$, $m, n, l, k \in \mathbb{N}$, is the $(ml) \times (nk)$-matrix

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} (\mathbf{A})_{11}\mathbf{B} & (\mathbf{A})_{1,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{1,n}\mathbf{B} \\ (\mathbf{A})_{2,1}\mathbf{B} & (\mathbf{A})_{2,2}\mathbf{B} & & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ (\mathbf{A})_{m,1}\mathbf{B} & (\mathbf{A})_{m,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{m,n}\mathbf{B} \end{bmatrix} \in \mathbb{K}^{ml,nk}.$$

Compute : $\underline{y} = (A \otimes B)\underline{x}$

$\underline{A}, \underline{B} \in \mathbb{R}^{n,n}, x \in \mathbb{R}^{n^2} \Rightarrow$ cost$(n) = O(n^4)$

Smarter: $\underline{x} = \begin{bmatrix} \underline{x}^1 \\ \vdots \\ \underline{x}^n \end{bmatrix}$, $\underline{x}^j \in \mathbb{R}^n$

$$(\mathbf{A} \otimes \mathbf{B})\mathbf{x} = \begin{bmatrix} (\mathbf{A})_{1,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{1,2}\mathbf{B}\mathbf{x}^2 + \dots + (\mathbf{A})_{1,n}\mathbf{B}\mathbf{x}^n \\ (\mathbf{A})_{2,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{2,2}\mathbf{B}\mathbf{x}^2 + \dots + (\mathbf{A})_{2,n}\mathbf{B}\mathbf{x}^n \\ \vdots \\ \vdots \\ (\mathbf{A})_{m,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{m,2}\mathbf{B}\mathbf{x}^2 + \dots + (\mathbf{A})_{m,n}\mathbf{B}\mathbf{x}^n \end{bmatrix}.$$

8

9

1.5. Machine Arithmetic

1.5.1. Loss of Orthogonalization

LA: Gram-Schmidt orthogonalization

Input: $\{\mathbf{a}^1,\dots,\mathbf{a}^k\} \subset \mathbb{K}^n$

1: $\mathbf{q}^1 := \frac{\mathbf{a}^1}{\|\mathbf{a}^1\|_2}$ % 1st output vector
2: **for** $j = 2,\dots,k$ **do**
{ % Orthogonal projection
3: $\mathbf{q}^j := \mathbf{a}^j$
4: **for** $\ell = 1,2,\dots,j-1$ **do**
5: { $\mathbf{q}^j \leftarrow \mathbf{q}^j - \langle \mathbf{a}^j, \mathbf{q}^\ell \rangle \mathbf{q}^\ell$ }
6: **if** ($\mathbf{q}^j = \mathbf{0}$) **then** STOP
7: **else** { $\mathbf{q}^j \leftarrow \frac{\mathbf{q}^j}{\|\mathbf{q}^j\|_2}$ }
8: }

(GS)

Output: $\{\mathbf{q}^1,\dots,\mathbf{q}^j\}$

If G.S. does not terminate prematurely*

- $(\underline{q}^j)^T(\underline{q}^i) = \delta_{ij}$ [orthonormal vectors]
- $\mathrm{Span}\{\underline{q}^1,\dots,\underline{q}^\ell\} = \mathrm{Span}\{\underline{a}^1,\dots,\underline{a}^\ell\}$

Proof → LA

[* if $\{\underline{a}^1,\dots,\underline{a}^r\}$ linearly indep.]

Exp.: $A \in \mathbb{R}^{10,10}$, $(A)_{i,j} = \frac{1}{i+j-1}$, $1 \le i,j \le 10$ ▷
"Hilbert matrix"

Test orthonormality of output vectors: $Q^TQ == I$

C++11 code 1.5.3: Gram-Schmidt orthogonalisation in EIGEN

```
2  template <class Matrix>
3  Matrix gramschmidt( const Matrix & A ) {
4      Matrix Q = A;
5      // First vector just gets normalized, Line 1 of (GS)
6      Q.col(0).normalize();
7      for(unsigned int j = 1; j < A.cols(); ++j) {
8          // Replace inner loop over each previous vector in Q with fast
9          // matrix-vector multiplication (Lines 4, 5 of (GS))
10         Q.col(j) -= Q.leftCols(j) * (Q.leftCols(j).transpose() *
               A.col(j));//
11         // Normalize vector, if possible.
12         // Otherwise colums of A must have been linearly dependent
13         if( Q.col(j).norm() <= 10e-9 * A.col(j).norm() ) { //
14             std::cerr << "Gram-Schmidt failed: A has lin. dep
                   columns." << std::endl;
15             break;
16         } else { Q.col(j).normalize(); } // Line 7 of (GS)
17     }
18     return Q;
19 }
```

C++11 code 1.5.7: Wrong result from Gram-Schmidt orthogonalisation EIGEN

```
2  void gsroundoff(MatrixXd& A){
3    // Gram-Schmidt orthogonalization of columns of A, see Code 1.5.3
4    MatrixXd Q = gramschmidt(A);
5    // Test orthonormality of columns of Q, which should be an
6    // orthogonal matrix according to theory
7    cout << setprecision(4) << fixed << "I = "
8         << endl << Q.transpose()*Q << endl;
9    // EIGEN's stable internal Gram-Schmidt orthogonalization by
10   // QR-decomposition, see Rem. 1.5.9 below
11   HouseholderQR<MatrixXd> qr(A.rows(),A.cols()); //
12   qr.compute(A); MatrixXd Q1 = qr.householderQ(); //
13   // Test orthonormality
14   cout << "I1 = " << endl << Q1.transpose()*Q1 << endl;
15   // Check orthonormality and span property (1.5.2)
16   MatrixXd R1 = qr.matrixQR().triangularView<Upper>();
17   cout << scientific << "A-Q1*R1 = " << endl << A-Q1*R1 << endl;
18 }
```

} Eigen's G.S.

```
I =
    1.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
    0.0000    1.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
   -0.0000   -0.0000    1.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
    0.0000    0.0000    0.0000    1.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
   -0.0000   -0.0000   -0.0000   -0.0000    1.0000    0.0000   -0.0008   -0.0007   -0.0007   -0.0006
    0.0000    0.0000    0.0000    0.0000    0.0000    1.0000   -0.0540   -0.0430   -0.0360   -0.0289
   -0.0000   -0.0000   -0.0000   -0.0000   -0.0008   -0.0540    1.0000    0.9999    0.9998    0.9996
   -0.0000   -0.0000   -0.0000   -0.0000   -0.0007   -0.0430    0.9999    1.0000    1.0000    0.9999
   -0.0000   -0.0000   -0.0000   -0.0000   -0.0007   -0.0360    0.9998    1.0000    1.0000    1.0000
   -0.0000   -0.0000   -0.0000   -0.0000   -0.0006   -0.0289    0.9996    0.9999    1.0000    1.0000
```

No orthonormality

▷ Computers cannot compute properly (in $\mathbb{R}$)!

↳ Inevitably so, because they can handle only finitely many numbers

→ Machine numbers (set $\mathbb{M}$)

$\mathbb{M} \subset \mathbb{R}$: finite & discrete

∃ largest/smallest (in modulus) machine numbers

Defined in Standard IEEE754/IEC559 [for double/float] → implementation

underflow \ overflow can happen

C++11 code 1.5.21: Querying characteristics of double numbers

```
#include <limits>
#include <iostream>
#include <iomanip>

using namespace std;

int main(){
    cout << std::numeric_limits<double>::is_iec559 << endl
    << std::defaultfloat << numeric_limits<double>::min() << endl
    << std::hexfloat << numeric_limits<double>::min() << endl
    << std::defaultfloat << numeric_limits<double>::max() << endl
    << std::hexfloat << numeric_limits<double>::max() << endl;
}
```

Output:

```
true
2.22507e-308
0010000000000000
1.79769e+308
7fefffffffffffff
```

$op : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ $\quad op \in \{'+', '-', '*', '/'\}$

$\mathbb{M} \times \mathbb{M} \not\longrightarrow \mathbb{M}$

[$\mathbb{M}$ is not closed under elementary arithmetic operations]

On computer $\widetilde{op} : \mathbb{M} \times \mathbb{M} \longrightarrow \mathbb{M}$

implementation : $\widetilde{op} = rd \circ op$

Definition 1.5.27. Correct rounding [to nearest machine number]

Correct rounding ("rounding up") is given by the function

$$rd : \begin{cases} \mathbb{R} & \to & \mathbb{M} \\ x & \mapsto & \max \operatorname{argmin}_{\tilde{x} \in \mathbb{M}} |x - \tilde{x}| \,. \end{cases}$$

C++11 code 1.5.23: Demonstration of roundoff errors

```
#include <iostream>
int main(){
    std::cout.precision(15);
    double a = 4.0/3.0, b = a-1, c = 3*b, e = 1-c;
    std::cout << e << std::endl;
    a = 1012.0/113.0; b = a-9; c = 113*b; e = 5+c;
    std::cout << e << std::endl;
    a = 83810206.0/6789.0; b = a-12345; c = 6789*b; e = c-1;
    std::cout << e << std::endl;
}
```

Exact: $e = 0$

◁ Computing in $\mathbb{M}$

Output:

```
2.22044604925031e-16
6.75015598972095e-14
-1.60798663273454e-09
```

Controlling roundoff error:

Maximal relative error of rounding:

$$\text{EPS} = \max_{x \in \mathbb{R}} \frac{|\,\text{rd}(x) - x\,|}{|x|}$$

[overflow/underflow ignored]

[rel. error of $\tilde{x}$ w.r.t. x $= \frac{|\tilde{x}-x|}{|x|} = \delta$: $\tilde{x} = x(1+\varepsilon)$, $|\varepsilon| < \delta$]

machine precision

Assumption 1.5.32. "Axiom" of roundoff analysis

There is a small positive number EPS, the machine precision, such that for the elementary arithmetic operations $\star \in \{+,-,\cdot,/\}$ and "hard-wired" functions* $f \in \{\exp, \sin, \cos, \log, \ldots\}$ holds

$$x \tilde{\star} y = (x \star y)(1+\delta) \quad , \quad \tilde{f}(x) = f(x)(1+\delta) \quad \forall x, y \in \mathbb{M}\,,$$

with $|\delta| < \text{EPS}$.

worst case estimate

C++11-code 1.5.34: Finding out EPS in C++

```
#include <iostream>
#include <limits> // get various properties of arithmetic types
int main() {
  std::cout.precision(15);
  std::cout << std::numeric_limits<double>::epsilon() << std::endl;
}
```

Output:

```
2.22044604925031e-16
```

```
cout.precision(25);
double eps =
   numeric_limits<double>::epsilon();
cout << fixed << 1.0 + 0.5*eps << endl
     << 1.0 - 0.5*eps << endl
     << (1.0 + 2/eps) - 2/eps << endl;
```

Output:

```
1.0000000000000000000000000
0.9999999999999998889776975
0.0000000000000000000000000
```

$1 \tilde{+} \underbrace{\tfrac{1}{2}\text{EPS}}_{=:\,\varepsilon > 0} = 1$ compatible with "Axiom"?

$1 = (1+\varepsilon)(1+\delta)$

$\Rightarrow |\delta| = \left|\frac{\varepsilon}{1+\varepsilon}\right| < \varepsilon = \tfrac{1}{2}\text{EPS}$

ok ✓

Remark: Testing == 0.0

C++11 code 1.5.3: Gram-Schmidt orthogonalisation in EIGEN

```
template <class Matrix>
Matrix gramschmidt( const Matrix & A ) {
    Matrix Q = A;
    // First vector just gets normalized, Line 1 of (GS)
    Q.col(0).normalize();
    for(unsigned int j = 1; j < A.cols(); ++j) {
        // Replace inner loop over each previous vector in Q with fast
        // matrix-vector multiplication (Lines 4, 5 of (GS))
        Q.col(j) -= Q.leftCols(j) * (Q.leftCols(j).transpose() *
            A.col(j));//
        // Normalize vector, if possible.
        // Otherwise colums of A must have been linearly dependent
        if( Q.col(j).norm() <= 10e-9 * A.col(j).norm() ) { //
            std::cerr << "Gram-Schmidt failed: A has lin. dep
                columns." << std::endl;
            break;
        } else { Q.col(j).normalize(); } // Line 7 of (GS)
    }
    return Q;
}
```

?

a numerical crime for results of floating point computations

replace with test of "relative smallness"

1.5.4. Cancellation

Ex: Computing the roots of a quadratic polynomial

C++11-code 1.5.41: Discriminant formula for the real roots of $p(\xi) = \xi^2 + \alpha\xi + \beta$

```
//!  C++ function computing the zeros of a quadratic polynomial
//!  ξ → ξ^2 + αξ + β by means of the familiar discriminant
//!  formula ξ_{1,2} = 1/2(-α ± √(α^2 - 4β)).  However
//!  this implementation is vulnerable to round-off!  The zeros are
//!  returned in a column vector
Vector2d zerosquadpol(double alpha, double beta){
  Vector2d z;
  double D = std::pow(alpha,2) -4*beta; // discriminant
  if(D < 0) throw "no real zeros";
  else{
    // The famous discriminant formula
    double wD = std::sqrt(D);
    z << (-alpha-wD)/2, (-alpha+wD)/2; //
  }
  return z;
}
```

Apply to $p(\xi) = (\xi - y)(\xi - 1/y) = \xi^2 - (y + 1/y) + 1$

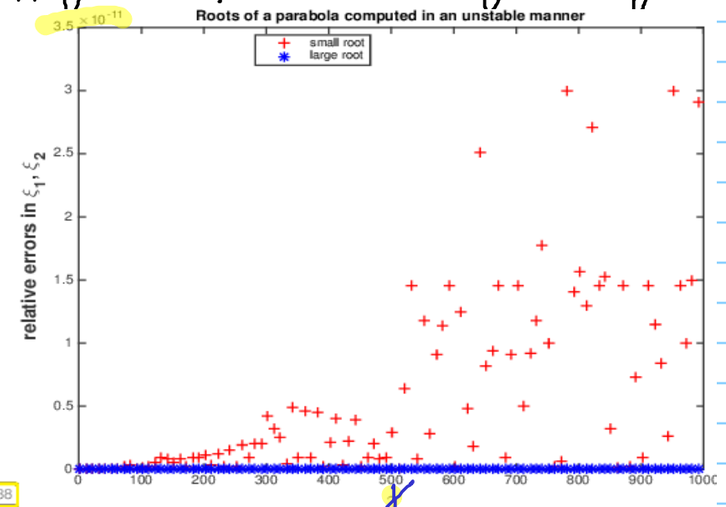


Fig. 38

◁ Huge* relative error for small root $1/y$

* w.r.t. EPS

(14)

We witness cancellation:

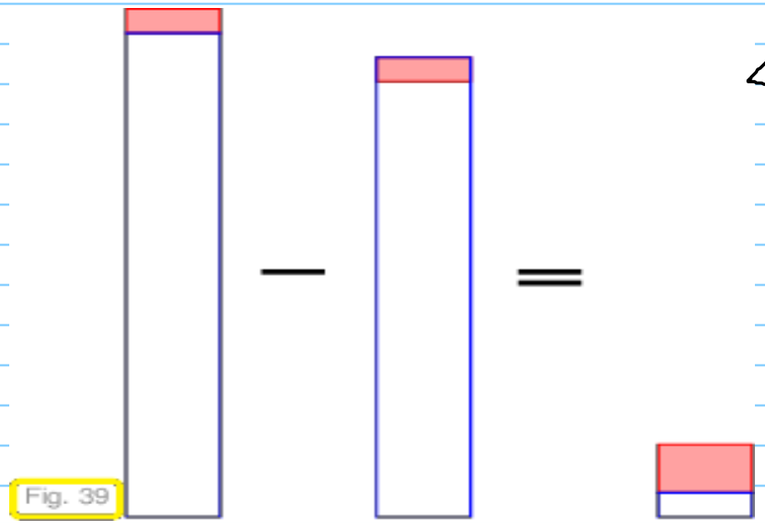


Fig. 39

◁ Huge error amplification when substracting two numbers of about the same size.

C++11-code 1.5.41: Discriminant formula for the real roots of $p(\xi) = \xi^2 + \alpha\xi + \beta$

```
//!  C++ function computing the zeros of a quadratic polynomial
//!  ξ → ξ^2 + αξ + β by means of the familiar discriminant
//!  formula ξ_{1,2} = 1/2(−α ± √(α^2 − 4β)).  However
//!  this implementation is vulnerable to round-off!  The zeros are
//!  returned in a column vector
Vector2d zerosquadpol(double alpha, double beta){
  Vector2d z;
  double D = std::pow(alpha,2) −4*beta; // discriminant
  if(D < 0) throw "no real zeros";
  else{
    // The famous discriminant formula
    double wD = std::sqrt(D);
    z << (−alpha−wD)/2, (−alpha+wD)/2; //
  }
  return z;
}
```

cancellation here

$\gamma \gg 1$

$\Rightarrow \alpha \gg 1, \beta = 1$

$\Rightarrow D \approx \alpha$

Ex: Difference quotients

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, \quad h \ll 1$$

cancellation will happen here

Analysis: approximation error $\longrightarrow 0$ as $h \longrightarrow 0$

C++11-code 1.5.48: Difference quotient approximation of the derivative of exp $\Rightarrow f(x) = e^x$

```
//!  Difference quotient approximation
//!  of the derivative of exp
void diffq(){
  double h = 0.1, x = 0.0;
  for(int i = 1; i <= 16; ++i){
    double df = (exp(x+h)−exp(x))/h;
    cout << setprecision(14) << fixed;
    cout  << setw(5) << −i
      << setw(20) << df−1 << endl;
    h /= 10;
  }
}
```

Measured relative errors ▷

$\log_{10}(h)$	relative error
-1	0.05170918075648
-2	0.00501670841679
-3	0.00050016670838
-4	0.00005000166714
-5	0.00000500000696
-6	0.00000049996218
-7	0.00000004943368
-8	-0.00000000607747
-9	0.00000008274037
-10	0.00000008274037
-11	0.00000008274037
-12	0.00008890058234
-13	-0.00079927783736
-14	-0.00079927783736
-15	0.11022302462516
-16	-1.00000000000000

effect of cancellation

We observe an initial decrease of the relative approximation error followed by a steep increase when h drops below 10^{-8}.

$$h_{opt} \approx \sqrt{EPS}$$

Rem: Cancellation & orthogonalization

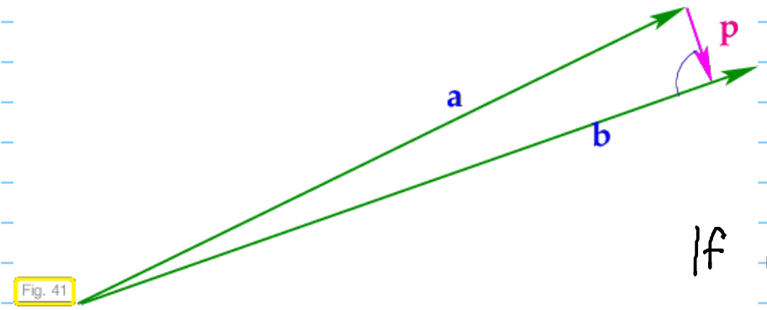


Fig. 41

◁ orth. projection

$$p = a - \frac{a^T b}{b^T b} b$$

If a, b nearly parallel: $p \approx 0$

→ cancellation here

Avoiding cancellation:

- Computing the roots of a quadratic polynomial:

$p(\xi) = \xi^2 + \alpha\xi + \beta$ w/ zeros ξ_1, ξ_2

$\Rightarrow \quad \xi_1 \cdot \xi_2 = \beta$

Idea: Compute large root ξ_2 (no problem)

$\xi_1 = \beta / \xi_2$

C++11-code 1.5.54: Stable computation of real root of a quadratic polynomial

NCSE15

```
//!  C++ function computing the zeros of a quadratic polynomial
//!  ξ → ξ^2 + αξ + β by means of the familiar discriminant
//!  formula ξ_{1,2} = ½(−α ± √(α^2 − 4β)).
//!  This is a stable implementation based on Vieta's theorem.
//!  The zeros are returned in a column vector
VectorXd zerosquadpolstab(double alpha, double beta){
  Vector2d z(2);
  double D = std::pow(alpha,2) -4*beta; // discriminant
  if(D < 0) throw "no real zeros";
  else{
    double wD = std::sqrt(D);
    // Use discriminant formula only for zero far away from 0
    // in order to avoid cancellation. For the other zero
    // use Vieta's formula.
    if(alpha >= 0){
      double t = 0.5*(-alpha-wD); //
      z << t, beta/t;
    }
    else{
      double t = 0.5*(-alpha+wD); //
      z << beta/t, t;
    }
  }
  return z;
}
```

→ no cancellation

- Avoiding cancellation by math. identities

$$\int_0^x \sin t \, dt = 1 - \cos x = 2 \sin^2(x/2)$$

↑ cancellation for $x \approx 0$

- Summation of exp-series

C++11-code 1.5.61: Summation of exponential series

```
double expeval(double x,
               double tol=1e-8){
  // Initialization
  double y = 1.0, term = 1.0;
  long int k = 1;
  // Termination criterion
  while(abs(term) > tol*y) {
    term *= x/k;     // next summand
    y += term; // Summation
    ++k;
  }
  return y;
}
```

$$e^x = \sum_{\ell=0}^{\infty} \frac{1}{\ell!} x^\ell$$

← termination criterion

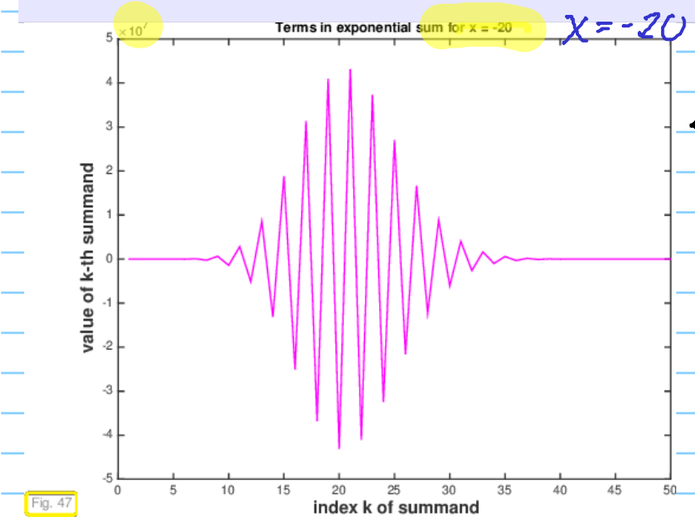


Fig. 47

◁ Size of summands: huge (in modulus)

x	Approximation $\widetilde{\exp}(x)$	$\exp(x)$	$\frac{\lvert\exp(x)-\widetilde{\exp}(x)\rvert}{\exp(x)}$
-20	6.1475618242e-09	2.0611536224e-09	1.982583033727893
-18	1.5983720359e-08	1.5229979745e-08	0.049490585500089
-16	1.1247503300e-07	1.1253517472e-07	0.000534425951530
-14	8.3154417874e-07	8.3152871910e-07	0.000018591829627
-12	6.1442105142e-06	6.1442123533e-06	0.000000299321453
-10	4.5399929604e-05	4.5399929762e-05	0.000000003501044
-8	3.3546262812e-04	3.3546262790e-04	0.000000000662004
-6	2.4787521758e-03	2.4787521767e-03	0.000000000332519
-4	1.8315638879e-02	1.8315638889e-02	0.000000000530724
-2	1.3533528320e-01	1.3533528324e-01	0.000000000273603
0	1.0000000000e+00	1.0000000000e+00	0.000000000000000
2	7.3890560954e+00	7.3890560989e+00	0.000000000479969
4	5.4598149928e+01	5.4598150033e+01	0.000000001923058
6	4.0342879295e+02	4.0342879349e+02	0.000000001344248
8	2.9809579808e+03	2.9809579870e+03	0.000000002102584
10	2.2026465748e+04	2.2026465795e+04	0.000000002143799
12	1.6275479114e+05	1.6275479142e+05	0.000000001723845
14	1.2026042798e+06	1.2026042842e+06	0.000000003634135
16	8.8861105010e+06	8.8861105205e+06	0.000000002197990
18	6.5659968911e+07	6.5659969137e+07	0.000000003450972
20	4.8516519307e+08	4.8516519541e+08	0.000000004828737

8 digits

Remedy : $e^{-x} = \frac{1}{e^x}$

Ex: Defeat cancellation by approximation

$$I(a) := \int_0^1 e^{at}\,dt = \frac{e^a - 1}{a} \quad : \text{ cancellation for } a \ll 1$$

$a > 0$

Idea: $$I(a) = \underbrace{\sum_{k=0}^{m} \frac{1}{(k+1)!} a^k}_{\text{Taylor sum}} + \frac{1}{(m+1)!} a^{\xi} a^m \qquad [0 < \xi < a]$$

↳ no cancellation

Choosing m large enough $\Rightarrow$ Taylor sum $\approx I(a)$

1.5.5. Numerical Stability

"Problem" $\quad F : X \subset \mathbb{R}^n \longrightarrow Y \subset \mathbb{R}^m$

(normed) $\quad$ data space $\quad$ result space

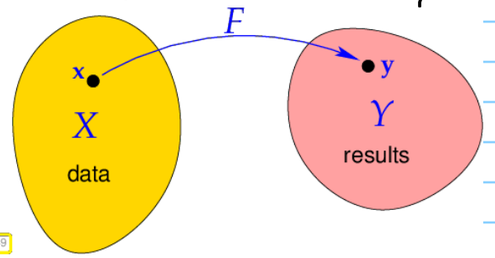


Fig. 49

Example: Problem of solving a $n \times n$ (square) linear system of equations (LSE)

$$F : \begin{cases} \mathbb{R}^{n,n}_{*} \times \mathbb{R}^n \longrightarrow \mathbb{R}^n \\ (A, \underline{b}) \longrightarrow \underline{x} := A^{-1}\underline{b} \end{cases}$$

$\mathbb{R}^{n,n}_{*} \hat{=}$ regular $n \times n$ matrices

Algorithm (work $\mathcal{M}$): $\quad \tilde{F} : \tilde{X} \subset X \longrightarrow \tilde{Y} \subset \mathcal{M}^m$

Definition 1.5.82. Stable algorithm = (Good algorithm)

An algorithm $\tilde{F}$ for solving a problem $F : X \mapsto Y$ is numerically stable, if for all $\mathbf{x} \in X$ its result $\tilde{F}(\mathbf{x})$ (possibly affected by roundoff) is the exact result for "slightly perturbed" data:

$$\exists C \approx 1: \quad \forall \mathbf{x} \in X: \quad \exists \tilde{\mathbf{x}} \in X: \quad \|\mathbf{x} - \tilde{\mathbf{x}}\|_X \le C w(\mathbf{x})\, \mathrm{EPS} \|\mathbf{x}\|_X \quad \wedge \quad \tilde{F}(\mathbf{x}) = F(\tilde{\mathbf{x}}) .$$

elementary ops. $\qquad$ machine precision

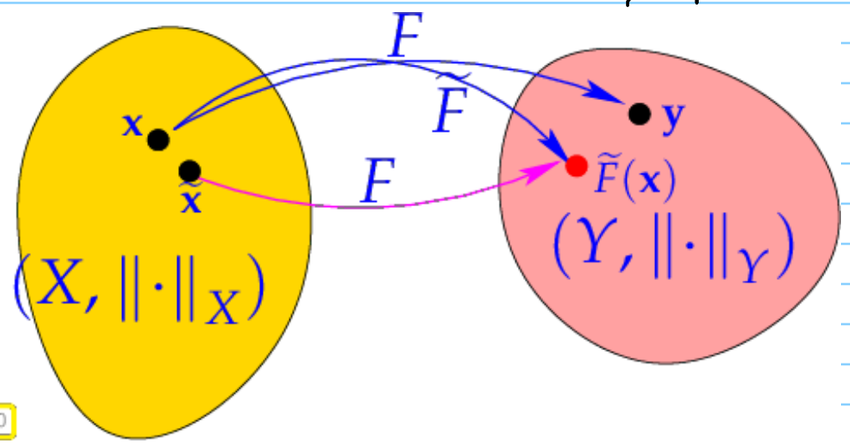


Fig. 50

Sloppily speaking, the impact of roundoff $(*)$ on a *stable algorithm* is of the same order of magnitude as the effect of the inevitable perturbations due to rounding the input data.

Stability $\not\Longleftarrow$ $\| F(x) - \tilde{F}(x) \| \approx EPS \, \| F(x) \|$

Fig. 51

Example : LSE

$\tilde{x} \in \mathbb{R}^n$: perturbed result of solver function

[i.e. $\hat{x} = \hat{F}(A,b)$]

$A\underline{x} = \underline{b}$ [$x = F(A,b)$]

to find: $\tilde{A}, \tilde{b}$: $\tilde{A}\tilde{x} = \tilde{b}$ [$\Leftrightarrow \tilde{x} = F(\tilde{A}, \tilde{b})$]

(i) $\tilde{A} = A$, $\tilde{b} = A\tilde{x}$

$$\| \tilde{b} - b \| = \underbrace{\| b - A\tilde{x} \| \overset{!}{\leq} C \cdot EPS \cdot \| b \|}_{\text{stable, if this holds for all } A,b}$$

$\rightarrow$ prove based on "Axiom"

(ii) $\tilde{A} = A + \frac{r \hat{x}^T}{\| \hat{x} \|^2}$, $r := b - A\tilde{x}$, $\tilde{b} = b$