Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 1

You should try to your best to do the core problems. If time permits, please try to do the rest as well.

Problem 1. Arrow matrix-vector multiplication (core problem)

Consider the multiplication of the two "arrow matrices" $\mathbf{A}$ with a vector $\mathbf{x}$, implemented as a function `arrowmatvec(d,a,x)` in the following MATLAB script

Listing 5: multiplying a vector with the product of two "arrow matrices"

```
1 function y = arrowmatvec(d,a,x)
2 % Multiplying a vector with the product of two ``arrow
     matrices''
3 % Arrow matrix is specified by passing two column
     vectors a and d
4 if (length(d) ~= length(a)), error ('size mismatch'); end
5 % Build arrow matrix using the MATLAB function diag()
6 A = [diag(d(1:end-1)),a(1:end-1);(a(1:end-1))',d(end)];
7 y = A*A*x;
```

(1a) ⊡ For general vectors $d = (d_1, \ldots, d_n)^\top$ and $a = (a_1, \ldots, a_n)^\top$, sketch the matrix $\mathbf{A}$ created in line 6 of Listing 5.

HINT: This MATLAB script is provided as file `arrowmatvec.m`.

Solution: $\mathbf{A} = \begin{pmatrix} d_1 & & & & a_1 \\ & d_2 & & & a_2 \\ & & \ddots & & \vdots \\ & & & d_{n-1} & a_{n-1} \\ a_1 & a_2 & \ldots & a_{n-1} & d_n \end{pmatrix}$

(1b) ⚂ The `tic-toc` timing results for `arrowmatvec.m` are available in Figure 1. Give a detailed explanation of the results.

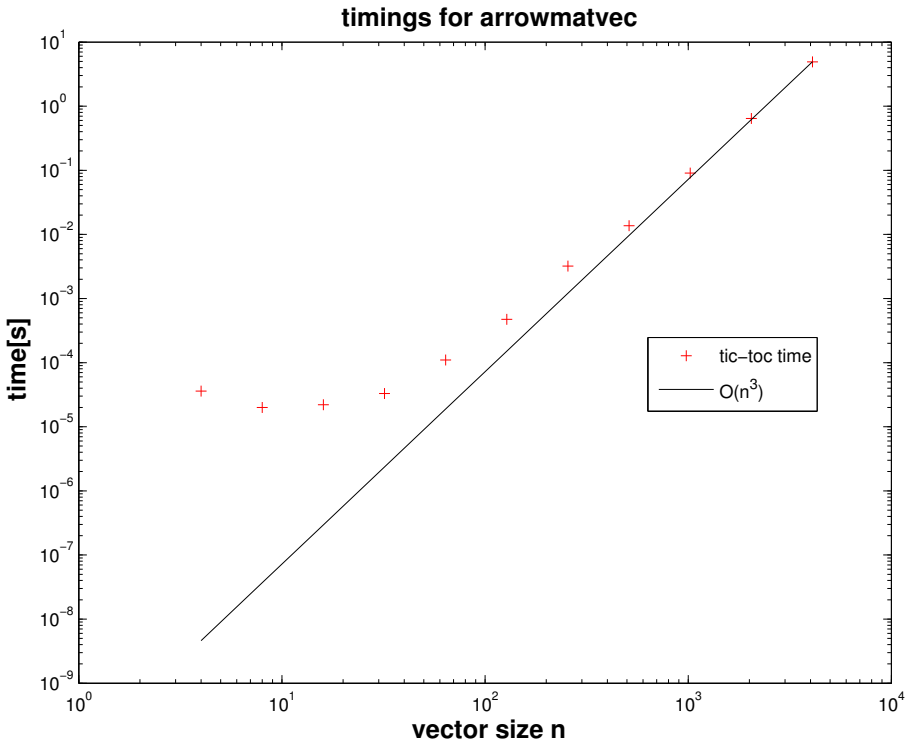

Figure 1: timings for `arrowmatvec(d,a,x)`

HINT: This MATLAB created figure is provided as file `arrowmatvectiming.{eps,jpg}`.

Solution: The standard matrix-matrix multiplication has runtimes growing with $O(n^3)$ and the standard matrix-vector multiplication has runtimes growing with $O(n^2)$. Hence, the overall computational complexity is dominated by $O(n^3)$.

(1c) ⚂ Write an *efficient* MATLAB function

```
function y = arrowmatvec2(d,a,x)
```

that computes the same multiplication as in code 5 but with optimal asymptotic complexity with respect to n. Here `d` passes the vector $(d_1,\ldots,d_n)^T$ and `a` passes the vector $(a_1,\ldots,a_n)^T$.

Solution: Due to the sparsity and special structure of the matrix, it is possible to write a more efficient implementation than the standard matrix-vector multiplication. See code listing 6

Listing 6: implementation of the function `arrowmatvec2`

```
function y = arrowmatvec2(d,a,x)
if (length(d) ~= length(a)), error ('size mismatch'); end
% Recursive matrix-vector multiplication to obtain A*A*x
   = A*(A*x)
y = A(d,a,A(d,a,x));
end

% Efficient multiplication of a vector with the ``arrow
   matrix''
function Ax = A(d,a,x)
Ax = d.*x;
Ax(1:end-1) = Ax(1:end-1) + a(1:end-1)*x(end);
Ax(end) = Ax(end) + a(1:end-1)'*x(1:end-1);
end
```

(1d) ⊡ What is the complexity of your algorithm from sub-problem (1c) (with respect to problem size n)?

Solution: The efficient implementation only needs two vector-vector element-wise multiplications and one vector-scalar multiplication. Therefore the complexity is $O(n)$.

(1e) ⚂ Compare the runtime of your implementation and the implementation given in code 5 for $n = 2^{5,6,\ldots,12}$. Use the routines `tic` and `toc` as explained in example [1, Ex. 1.4.10] of the Lecture Slides.

Solution: The standard matrix multiplication has runtimes growing with $O(n^3)$. The runtimes of the more efficient implementation are growing with $O(n)$. See Listing 7 and Figure 2.

Listing 7: Execution and timings of `arrowmatvec` and `arrowmatvec2`

```
nruns = 3; res = [];
```

```
ns = 2.^(2:12);
for n = ns
  a  = rand(n,1); d = rand(n,1); x = rand(n,1);
  t  = realmax;
  t2 = realmax;
  for k=1:nruns
    tic; y  = arrowmatvec(d,a,x);  t  = min(toc,t);
    tic; y2 = arrowmatvec2(d,a,x); t2 = min(toc,t2);
  end;
  res = [res; t t2];
end
figure('name','timings arrowmatvec and arrowmatvec2');
c1 = sum(res(:,1))/sum(ns.^3);
c2 = sum(res(:,2))/sum(ns);
loglog(ns, res(:,1),'r+', ns, res(:,2),'bo',...
       ns, c1*ns.^3, 'k-', ns, c2*ns, 'g-');
xlabel('{\bf vector size n}','fontsize',14);
ylabel('{\bf time[s]}','fontsize',14);
title('{\bf timings for arrowmatvec and
   arrowmatvec2}','fontsize',14);
legend('arrowmatvec','arrowmatvec2','O(n^3)','O(n)',...
       'location','best');
print -depsc2 '../PICTURES/arrowmatvec2timing.eps';
```

(1f) ⊡ Write the EIGEN codes corresponding to the functions `arrowmatvec` and `arrowmatvec2`.

Solution: See Listing 8 and Listing 9.

Listing 8: Implementation of `arrowmatvec` in EIGEN

```
#include <Eigen/Dense>
#include <iostream>
#include <ctime>

using namespace Eigen;

template <class Matrix>
```

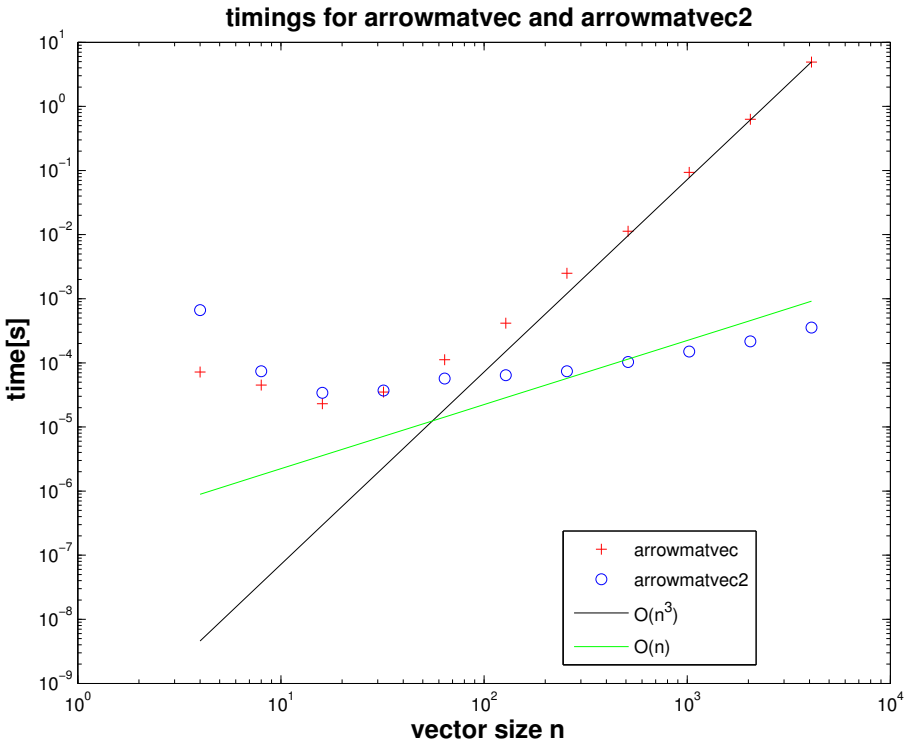

Figure 2: timings for `arrowmatvec2(d,a,x)`

```
8
9  // The function arrowmatvec computes the product the
       desired product directly as A*A*x
10 void arrowmatvec(const Matrix & d, const Matrix & a,
       const Matrix & x, Matrix & y)
11 {
12     // Here, we choose a MATLAB style implementation
           using block construction
13     // you can also use loops
14     // If you are interested you can compare both
           implementation and see if and how they differ
15     int n=d.size();
16     VectorXd dcut= d.head(n-1);
17     VectorXd acut = a.head(n-1);
18     MatrixXd ddiag=dcut.asDiagonal();
```

```
19      MatrixXd A(n,n);
20      MatrixXd D = dcut.asDiagonal();
21      // If you do not create the temporary matrix D, you
           will get an error: D must be casted to MatrixXd
22      A << D, a.head(n-1), acut.transpose(), d(n-1);
23
24      y=A*A*x;
25  }
26
27  // We test the function arrowmatvec with 5 dimensional
       random vectors.
28
29  int main(void)
30  {
31  //      srand((unsigned int) time(0));
32      VectorXd a=VectorXd::Random(5);
33      VectorXd d=VectorXd::Random(5);
34      VectorXd x=VectorXd::Random(5);
35      VectorXd y;
36
37      arrowmatvec(d,a,x,y);
38      std::cout << "A*A*x = " << y << std::endl;
39  }
```

Listing 9: Implementation of `arrowmatvec2` in EIGEN

```
1  #include <Eigen/Dense>
2  #include <iostream>
3
4  using namespace Eigen;
5
6  template <class Matrix>
7
8  // The auxiliary function Atimesx computes the function
       A*x in a smart way, using the particular structure of
       the matrix A.
9
10 void Atimesx(const Matrix & d, const Matrix & a, const
```

```
    Matrix & x, Matrix & Ax)
{
    int n=d.size();
    Ax=(d.array()*x.array()).matrix();
    VectorXd Axcut=Ax.head(n-1);
    VectorXd acut = a.head(n-1);
    VectorXd xcut = x.head(n-1);

    Ax << Axcut + x(n-1)*acut, Ax(n-1)+
       acut.transpose()*xcut;
}

// We compute A*A*x by using the function Atimesx twice
   with 5 dimensional random vectors.

int main(void)
{
    VectorXd a=VectorXd::Random(5);
    VectorXd d=VectorXd::Random(5);
    VectorXd x=VectorXd::Random(5);
    VectorXd Ax(5);

    Atimesx(d,a,x,Ax);
    VectorXd AAx(5);
    Atimesx(d,a,Ax,AAx);
    std::cout << "A*A*x = " << AAx << std::endl;
}
```

Problem 2. Avoiding cancellation (core problem)

In [1, Section 1.5.4] we saw that the so-called *cancellation phenomenon* is a major cause of numerical instability, *cf.* [1, § 1.5.38]. Cancellation is the massive amplification of *relative errors* when subtracting two real numbers of about the same value.

Fortunately, expressions vulnerable to cancellation can often be recast in a mathematically equivalent form that is no longer affected by cancellation, see [1, § 1.5.46]. There we studied several examples, and this problem gives some more.

(2a) ⊡ We consider the function

$$f_1(x_0, h) := \sin(x_0 + h) - \sin(x_0)\ . \tag{1}$$

It can the transformed into another form, $f_2(x_0, h)$, using the trigonometric identity

$$\sin(\varphi) - \sin(\psi) = 2\cos\left(\frac{\varphi + \psi}{2}\right)\sin\left(\frac{\varphi - \psi}{2}\right).$$

Thus, f_1 and f_2 give the same values, in exact arithmetic, for any given argument values x_0 and h.

1. Derive $f_2(x_0, h)$, which does no longer involve the difference of return values of trigonometric functions.
2. Suggest a formula that avoids cancellation errors for computing the approximation $(f(x_0 + h) - f(x_0))/h)$ of the derivative of $f(x) := \sin(x)$ at $x = x_0$. Write a MATLAB program tha implements your formula and computes an approximation of $f'(1.2)$, for $h = 1 \cdot 10^{-20}, 1 \cdot 10^{-19}, \cdots, 1$.

   HINT: For background information refer to [1, Ex. 1.5.40].
3. Plot the error (in doubly logarithmic scale using MATLAB's `loglog` plotting function) of the derivative computed with the suggested formula and with the naive implementation using f_1.
4. Explain the observed behaviour of the error.

Solution: Check the MATLABimplementation in Listing 10 and the plot in Fig. 3. We can clearly observe that the computation using f_1 leads to a big error as $h \to 0$. This is due to the cancellation error given by the subtraction of two number of approximately same magnitude. The second implementation using f_2 is very stable and does not display round-off errors.

Listing 10: MATLABscript for Problem 2.

```
%% Cancellation
h = 10.^(-20:0);
x = 1.2;

% Derivative
```

```
g1 = (sin(x+h) - sin(x)) ./ h; % Naive
g2 = 2 .* cos(x + h * 0.5) .* sin(h * 0.5) ./ h; % Better
ex = cos(x); % Exact

% Plot
loglog(h, abs(g1-ex), 'r',h, abs(g2-ex), 'b', h, h,
    'k--');
title('Error of the approximation of f''(x_0)');
legend('g_1','g_2', 'O(h)');
xlabel('h');
ylabel('| f''(x_0) - g_i(x_0,h) |');
grid on
```

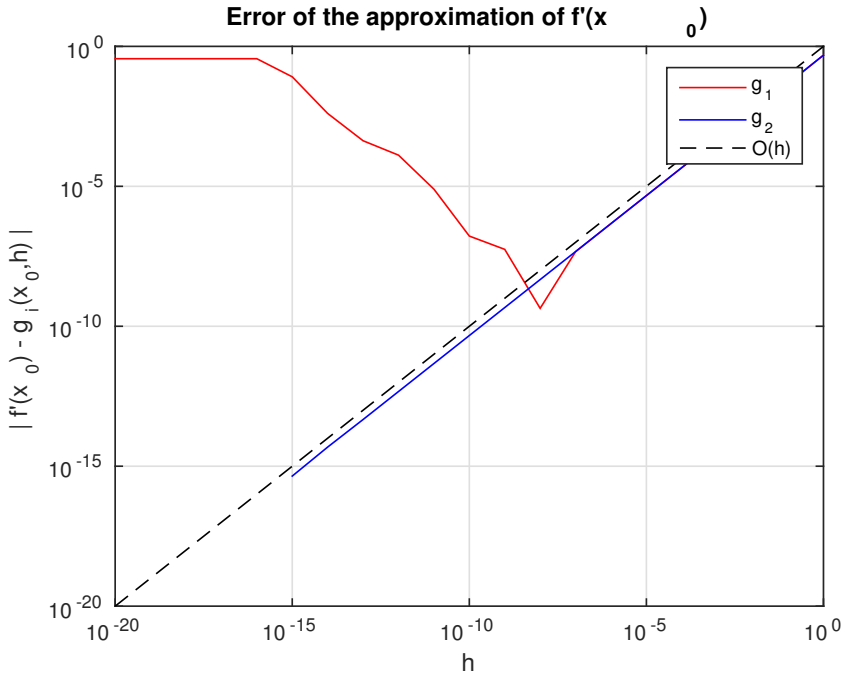

Figure 3: Timings for Problem 2.

(2b) ⚀ Using a trick applied in [1, Ex. 1.5.51] show that

$$\ln(x - \sqrt{x^2 - 1}) = -\ln(x + \sqrt{x^2 - 1}) \, .$$

Which of the two formulas is more suitable for numerical computation? Explain why, and provide a numerical example in which the difference in accuracy is evident.

Solution: $\ln(x-\sqrt{x^2-1})+\ln(x-\sqrt{x^2-1}) = \log(x^2-(x^2-1)) = 0$. As $x \to \infty$ the left $\log$ consists of subtraction of two numbers of equal magnitude, whilst the right $\log$ consists on the addition of two numbers of approximately the same magnitude. Therefore, in the first case there may be cancellation for large values of x, making it worse for numerical computation. Try, in MATLAB, with $x = 10^8$.

(2c) ⚀ For the following expressions, state the numerical difficulties that may occur, and rewrite the formulas in a way that is more suitable for numerical computation.

1. $\sqrt{x+\frac{1}{x}} - \sqrt{x-\frac{1}{x}}$, where $x \gg 1$.
2. $\sqrt{\frac{1}{a^2}+\frac{1}{b^2}}$, where $a \approx 0, b \approx 1$.

Solution:

1. Inside the square roots we have the addition (rest. subtraction) of a small number to a big number. The difference of the square roots incur in cancellation, since they have the same, large magnitude. $A := x+\frac{1}{x}, B := x-\frac{1}{x}$ then $(A-B)(A+B)/(A+B) = \frac{2/x}{\sqrt{x+\frac{1}{x}}+\sqrt{x-\frac{1}{x}}} = \frac{2}{\sqrt{x}(\sqrt{x^2+1}+\sqrt{x^2-1})}$
2. $\frac{1}{a^2}$ becomes very large as a approaches 0, whilst $\frac{1}{b^2} \to 1$ as $b \to 1$. Therefore, the relative size of $\frac{1}{a^2}$ and $\frac{1}{b^2}$ becomes so big, that, in computer arithmetic, $\frac{1}{a^2} + \frac{1}{b^2} = \frac{1}{b^2}$. On the other hand $\frac{1}{a}\sqrt{1+\left(\frac{a}{b}\right)^2}$ avoids this problem by performing a division between two numbers with very different magnitude, instead of a summation.

Problem 3. Kronecker product

In [1, Def. 1.4.16] we learned about the so-called Kronecker product, available in MATLAB through the command `kron`. In this problem we revisit the discussion of [1, Ex. 1.4.17]. Please refresh yourself on this example and study [1, Code 1.4.18] again.

As in [1, Ex. 1.4.17], the starting point is the line of MATLAB code

$$\mathbf{y} = \texttt{kron(A,B)} * \mathbf{x}, \tag{2}$$

where the arguments are $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,n}, \mathbf{x} \in \mathbb{R}^{n \cdot n}$.

(3a) ⊡ Obtain further information about the `kron` command from MATLAB help issuing `doc kron` in the MATLAB command window.

Solution: See MATLAB help.

(3b) ⊡ Explicitly write Eq. (2) in the form $\mathbf{y} = \mathbf{M}\mathbf{x}$ (i.e. write down $\mathbf{M}$), for $\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and $\mathbf{B} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$.

Solution: $\mathbf{y} = \begin{pmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{pmatrix} \mathbf{x}$.

(3c) ⊡ What is the asymptotic complexity (→ [1, Def. 1.4.3]) of the MATLAB code (2)? Use the Landau symbol from [1, Def. 1.4.4] to state your answer.

Solution: `kron(A,B)` results in a matrix of size $n^2 \times n^2$ and x has length n^2. So the complexity is the same as a matrix-vector multiplication for the resulting sizes. In total this is $O(n^2 * n^2) = O(n^4)$.

(3d) ⊡ Measure the runtime of (2) for $n = 2^{3,4,5,6}$ and random matrices. Use the MATLAB functions `tic` and `toc` as explained in example [1, Ex. 1.4.10] of the Lecture Slides.

Solution: Since `kron(A,B)` creates a large matrix consisting of smaller blocks with size n, i.e. B multiplied with $A(i,j)$, we can split the problem up in n matrix-vector multiplications of size n. This results in a routine with complexity $n * O(n^2) = O(n^3)$ The implementation is listed in 11. The runtimes are shown in Figure 4.

Listing 11: An efficient implementation for Problem 3.

```
function y = Kron_B(A,B,x)
% Return y = kron(A,B)*x  (smart version)
% Input:   A,B:     2 n x n matrices.
%          x:       Vector of length n*n.
% Output:  y:       Result vector of length n*n.

% check size of A
[n,m] = size(A);
assert(n == m, 'expected quadratic matrix')
```

```
10
11 % kron gives a matrix with n x n blocks
12 % block i,j is A(i,j)*B
13 % => y = M*x can be done block-wise so that we reuse
     B*x(...)
14
15 % init
16 y = zeros(n*n,1);
17 % loop first over columns and then (!) over rows
18 for j = 1:n
19     % reuse B*x(...) part (constant in given column) =>
         O(n^2)
20     z = B*x((j-1)*n+1:j*n);
21     % add to result vector (need to go through full
         vector) => O(n^2)
22     for i = 1:n
23         y((i-1)*n+1:i*n) = y((i-1)*n+1:i*n) + A(i,j)*z;
24     end
25 end
26 % Note: complexity is O(n^3)
27 end
```

(3e) ⚀ Explain in detail, why (2) can be replaced with the single line of MATLAB code

```
y = reshape( B * reshape(x,n,n) * A', n*n, 1);        (3)
```

and compare the execution times of (2) and (3) for random matrices of size $n = 2^{3,4,5,6}$.

Solution:

Listing 12: A second efficient implementation for Problem 3. using `reshape`.

```
1 function y = Kron_C (A,B,x)
2 % Return y = kron(A,B)*x  (smart version with reshapes)
3 % Input:   A,B:       2 n x n matrices.
4 %            x:       Vector of length n*n.
5 % Output:    y:       Result vector of length n*n.
6
7 % check size of A
```

```
[n,m] = size(A);  assert(n == m, 'expected quadratic
   matrix')

% init
yy = zeros(n,n);
xx = reshape(x,n,n);

% precompute the multiplications of B with the parts of
   vector x
Z = B*xx;
for j=1:n
    yy = yy + Z(:,j) * A(:,j)';
end
y = reshape(yy,n^2,1);
end
% Notes: complexity is O(n^3)
```

Listing 13: Main routine for runtime measurements of Problem 3.

```
%% Kron (runtimes)
clear all; close all;
nruns = 3;  % we average over a few runs
N = 2.^(2:10);
T = zeros(length(N),5);
for i = 1:length(N)
    n = N(i)     % problem size n
    % % initial matrices A,B and vector x
    % A = reshape(1:n*n, n, n)';
    % B = reshape(n*n+1:2*n*n, n, n)';
    % x = (1:n*n)';
    % alternative choice:
    A=rand(n);     B=rand(n);     x=rand(n^2,1);

    tic;     % smart implementation #1
    for ir = 1:nruns
        yB = Kron_B(A,B,x);
    end
    tb = toc/nruns;
```

```
20
21      tic;   % smart implementation #2: with reshapes
22      for ir = 1:nruns
23          yC = Kron_C(A,B,x);
24      end
25      tc = toc/nruns;
26      fprintf('Error B-vs-C:   %g\n', norm(yB-yC))
27
28      tic;   % implementation with kron and matrix*vector
29      if(N(i)<128)
30          for ir = 1:nruns
31              yA = kron(A,B)*x;
32          end
33          ta = toc/nruns;
34          fprintf('Error A-vs-B:   %g\n', norm(yA-yB))
35      else
36          ta=0;
37      end
38
39      tic;  % implementation with 1-line command!
40            % inspired by the solution of Kevin Bocksrocker
41      for ir = 1:nruns
42          yD = reshape(B*reshape(x,n,n)*A',n^2,1);
43      end
44      td = toc/nruns;
45      fprintf('Error D-vs-B:   %g\n', norm(yD-yB));
46
47      fprintf('Timings: Matrix: %g\n         Smart:
          %g\n',ta,tb)
48      fprintf('         Smart+Reshape:  %g\n
          1-line Smart:  %g\n',tc,td)
49      T(i,:) = [n, ta, tb, tc, td];
50  end
51
52  % log-scale plot for investigation of asymptotic
      complexity
53  a2 = sum(T(:,3)) / sum(T(:,1).^2);
54  a3 = sum(T(:,3)) / sum(T(:,1).^3);
```

```
a4 = sum(T(1:5,2)) / sum(T(1:5,1).^4);
figure('name','kron timings');
loglog(T(:,1),T(:,2),'m+', T(:,1),T(:,3),'ro',...
       T(:,1),T(:,4),'bd', T(:,1),T(:,5),'gp',...
       T(:,1),(T(:,1).^2)*a2,'k-',
          T(:,1),(T(:,1).^3)*a3,'k--',...
       T(1:5,1),(T(1:5,1).^4)*a4,'k-.', 'linewidth', 2);
xlabel('{\bf problem size n}','fontsize',14);
ylabel('{\bf average runtime (s)}','fontsize',14);
title(sprintf('tic-toc timing averaged over %d runs',
   nruns),'fontsize',14);
legend('slow evaluation','efficient evaluation',...
    'efficient ev. with reshape','Kevin 1-line',...
    'O(n^2)','O(n^3)','O(n^4)','location','northwest');
print -depsc2 '../PICTURES/kron_timings.eps';
```

(3f) ⚀ Based on the EIGEN numerical library (→ [1, Section 1.2.2]) implement a C++ function

```
template <class Matrix>
void kron(const Matrix & A, const Matrix & B, Matrix &
   C) {
  // Your code here
}
```

returns the Kronecker product of the argument matrices `A` and `B` in the matrix `C`.

HINT: Feel free (but not forced) to use the partial codes provided in `kron.cpp` as well as the CMake file `CMakeLists.txt` (including `cmake-modules`) and the timing header file `timer.h`.

Solution: See `kron.cpp` or Listing 14.

(3g) ⚁ Devise an implementation of the MATLAB code (2) in C++according to the function definition

```
template <class Matrix, class Vector>
void kron_mv(const Matrix & A, const Matrix & B, const
```

```
    Vector & x, Vector & y);
```

The meaning of the arguments should be self-explanatory.

Solution: See `kron.cpp` or Listing 14.

(3h) ⚄ Now, using a function definition similar to that of the previous sub-problem, implement the C++ equivalent of (3) in the function `kron_mv_fast`.

HINT: Study [1, Rem. 1.2.18] about "reshaping" matrices in EIGEN.

(3i) ⚄ Compare the runtimes of your two implementations as you did for the MATLAB implementations in sub-problem (3e).

Solution:

Listing 14: Main routine for runtime measurements of Problem 3.

```
#include <Eigen/Dense>
#include <iostream>
#include <vector>

#include "timer.h"

//! \brief Compute the Kronecker product C = A ⊗ B.
//! \param[in] A Matrix n × n
//! \param[in] B Matrix n × n
//! \param[out] C Kronecker product of A and B of dim
    n^2 × n^2
template <class Matrix>
void kron(const Matrix & A, const Matrix & B, Matrix & C)
{
    C = Matrix(A.rows()*B.rows(), A.cols()*B.cols());
    for(unsigned int i = 0; i < A.rows(); ++i) {
        for(unsigned int j = 0; j < A.cols(); ++j) {
            C.block(i*B.rows(),j*B.cols(), B.rows(),
               B.cols()) = A(i,j)*B;
        }
    }
}

```

```
22 //! \brief Compute the Kronecker product C = A⊗B.
     Exploit matrix-vector product.
23 //! A,B and x must have dimension n \times n resp. n
24 //! \param[in] A Matrix n×n
25 //! \param[in] B Matrix n×n
26 //! \param[in] x Vector of dim n
27 //! \param[out] y Vector y = kron(A,B)*x
28 template <class Matrix, class Vector>
29 void kron_fast(const Matrix & A, const Matrix & B, const
     Vector & x, Vector & y)
30 {
31     y = Vector::Zero(A.rows()*B.rows());
32
33     unsigned int n = A.rows();
34     for(unsigned int j = 0; j < A.cols(); ++j) {
35         Vector z = B * x.segment(j*n, n);
36         for(unsigned int i = 0; i < A.rows(); ++i) {
37             y.segment(i*n,n) += A(i,j)*z;
38         }
39     }
40 }
41
42 //! \brief Compute the Kronecker product C=A⊗B. Uses
     fast remapping tecniques (similar to Matlab reshape)
43 //! A,B and x must have dimension n \times n resp. n*n
44 //! Elegant way using reshape
45 //! WARNING: using Matrix::Map we assume the matrix is
     in ColMajor format, *beware* you may incur in bugs if
     matrix is in RowMajor isntead
46 //! \param[in] A Matrix n×n
47 //! \param[in] B Matrix n×n
48 //! \param[in] x Vector of dim n
49 //! \param[out] y Vector y = kron(A,B)*x
50 template <class Matrix, class Vector>
51 void kron_super_fast(const Matrix & A, const Matrix & B,
     const Vector & x, Vector & y)
52 {
53     unsigned int n = A.rows();
```

```
54      Matrix t = B * Matrix::Map(x.data(),n,n) *
            A.transpose();
55      y = Matrix::Map(t.data(), n*n, 1);
56  }
57
58  int main(void) {
59
60      // Check if kron works, cf.
61      Eigen::MatrixXd A(2,2);
62      A << 1, 2, 3, 4;
63      Eigen::MatrixXd B(2,2);
64      B << 5, 6, 7, 8;
65      Eigen::MatrixXd C;
66
67      Eigen::VectorXd x = Eigen::VectorXd::Random(4);
68      Eigen::VectorXd y;
69      kron(A,B,C);
70      y = C*x;
71      std::cout << "kron(A,B)=" << std::endl << C <<
            std::endl;
72      std::cout << "Using kron: y=       " << std::endl <<
            y << std::endl;
73
74      kron_fast(A,B,x,y);
75      std::cout << "Using kron_fast: y=  " << std::endl <<
            y << std::endl;
76      kron_super_fast(A,B,x,y);
77      std::cout << "Using kron_super_fast: y=  " <<
            std::endl << y << std::endl;
78
79      // Compute runtime of different implementations of
            kron
80      unsigned int repeats = 10;
81      timer<> tm_kron, tm_kron_fast, tm_kron_super_fast;
82      std::vector<int> times_kron, times_kron_fast,
            times_kron_super_fast;
83
84      for(unsigned int p = 2; p <= 10; p++) {
```

```
85          tm_kron.reset();
86          tm_kron_fast.reset();
87          tm_kron_super_fast.reset();
88          for(unsigned int r = 0; r < repeats; ++r) {
89              unsigned int M = pow(2,p);
90              A = Eigen::MatrixXd::Random(M,M);
91              B = Eigen::MatrixXd::Random(M,M);
92              x = Eigen::VectorXd::Random(M*M);
93
94              // May be too slow for large p, comment if so
95              tm_kron.start();
96  //              kron(A,B,C);
97  //              y = C*x;
98              tm_kron.stop();
99
100             tm_kron_fast.start();
101             kron_fast(A,B,x,y);
102             tm_kron_fast.stop();
103
104             tm_kron_super_fast.start();
105             kron_super_fast(A,B,x,y);
106             tm_kron_super_fast.stop();
107         }
108
109         std::cout << "Lazy Kron took:        " <<
              tm_kron.min().count() / 1000000. << " ms" <<
              std::endl;
110         std::cout << "Kron fast took:        " <<
              tm_kron_fast.min().count() / 1000000. << "
              ms" << std::endl;
111         std::cout << "Kron super fast took: " <<
              tm_kron_super_fast.min().count() / 1000000.
              << " ms" << std::endl;
112         times_kron.push_back( tm_kron.min().count() );
113         times_kron_fast.push_back(
              tm_kron_fast.min().count() );
114         times_kron_super_fast.push_back(
              tm_kron_super_fast.min().count() );
```

```
115     }
116
117     for(auto it = times_kron.begin(); it !=
          times_kron.end(); ++it) {
118       std::cout << *it << " ";
119     }
120     std::cout << std::endl;
121     for(auto it = times_kron_fast.begin(); it !=
          times_kron_fast.end(); ++it) {
122       std::cout << *it << " ";
123     }
124     std::cout << std::endl;
125     for(auto it = times_kron_super_fast.begin(); it !=
          times_kron_super_fast.end(); ++it) {
126       std::cout << *it << " ";
127     }
128     std::cout << std::endl;
129 }
```

Problem 4. Structured matrix–vector product

In [1, Ex. 1.4.14] we saw how the particular structure of a matrix can be exploited to compute a matrix-vector product with substantially reduced computational effort. This problem presents a similar case.

Consider the real $n \times n$ matrix $\mathbf{A}$ defined by $(\mathbf{A})_{i,j} = a_{i,j} = \min\{i, j\}$, for $i, j = 1, \ldots, n$. The matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ can be implemented in MATLAB as

$$\texttt{y = min(ones(n,1) * (1:n), (1:n)' * ones(1,n)) * x;} \tag{4}$$

(4a) ⚀ What is the asymptotic complexity (for $n \to \infty$) of the evaluation of the MATLAB command displayed above, with respect to the problem size parameter n?

Solution: Matrix–vector multiplication: quadratic dependence $O(n^2)$.

(4b) ⚅ Write an *efficient* MATLAB function

```
function y = multAmin(x)
```

that computes the same multiplication as (4) but with a better asymptotic complexity with respect to n.

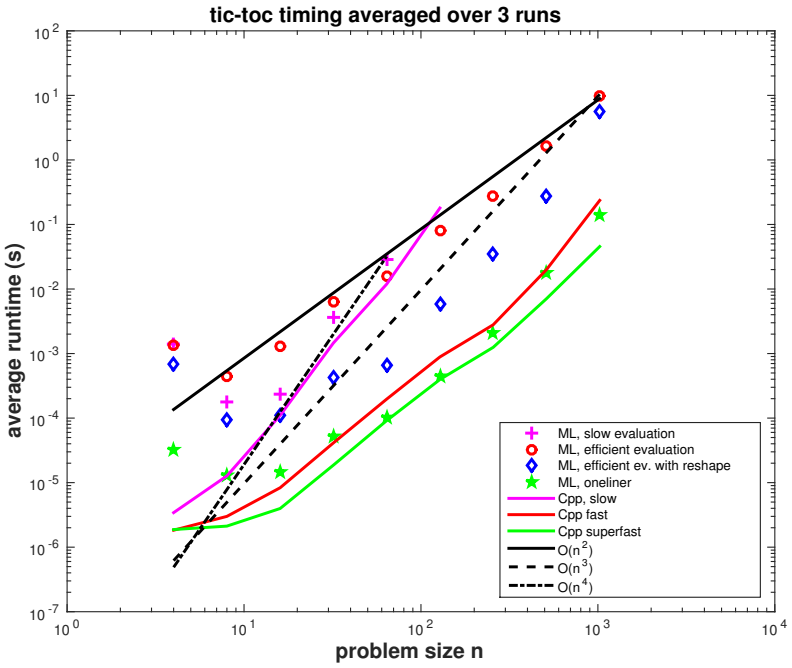

Figure 4: Timings for Problem 3. with MATLABand C++implementations.

HINT: you can test your implementation by comparing the returned values with the ones obtained with code (4).

Solution: For every j we have $y_j = \sum_{k=1}^{j} kx_k + j \sum_{k=j+1}^{n} x_k$, so we pre-compute the two terms for every j only once.

Listing 15: implementation for the function multAmin

```
function y = multAmin(x)
% O(n), slow version
n = length(x);
y = zeros(n,1);
v = zeros(n,1);
w = zeros(n,1);

v(1) = x(n);
w(1) = x(1);
```

```
for j = 2:n
    v(j) = v(j-1)+x(n+1-j);
    w(j) = w(j-1)+j*x(j);
end
for j = 1:n-1
    y(j) = w(j) + v(n-j)*j;
end
y(n) = w(n);

% To check the code, run:
% n=500; x=randn(n,1); y = multAmin(x);
% norm(y - min(ones(n,1)*(1:n), (1:n)'*ones(1,n)) * x)
```

(4c) ⚀ What is the asymptotic complexity (in terms of problem size parameter n) of your function `multAmin`?

Solution: Linear dependence: $O(n)$.

(4d) ⚁ Compare the runtime of your implementation and the implementation given in (4) for $n = 2^{5,6,\ldots,12}$. Use the routines `tic` and `toc` as explained in example [1, Ex. 1.4.10] of the Lecture Slides.

Solution:

The matrix multiplication in (4) has runtimes growing with $O(n^2)$. The runtimes of the more efficient implementation with hand-coded loops, or using the MATLABfunction `cumsum` are growing with $O(n)$.

Listing 16: comparison of execution timings

```
ps = 4:12;
ns = 2.^ps;
ts = 1e6*ones(length(ns),3);     % timings
nruns = 10;                      % to average the runtimes

% loop over different Problem Sizes
for j=1:length(ns)
    n = ns(j);
    fprintf('Vector length: %d \n', n);
    x = rand(n,1);
```

```
11
12      % timing for naive multiplication
13      tic;
14      for k=1:nruns
15        y = min(ones(n,1)*(1:n), (1:n)'*ones(1,n)) * x;
16      end
17      ts(j,1) = toc/nruns;
18
19      % timing multAmin
20      tic;
21      for k=1:nruns
22        y = multAmin(x);
23      end
24      ts(j,2) = toc/nruns;
25
26      % timing multAmin2
27      tic;
28      for k=1:nruns
29        y = multAmin2(x);
30      end
31      ts(j,3) = toc/nruns;
32  end
33
34  c1 = sum(ts(:,2)) / sum(ns);
35  c2 = sum(ts(:,1)) / sum(ns.^2);
36
37  loglog(ns, ts(:,1), '-k', ns, ts(:,2), '-og', ns,
      ts(:,3), '-xr',...
38          ns, c1*ns, '-.b', ns, c2*ns.^2, '--k',
              'linewidth', 2);
39  legend('naive','multAmin','multAmin2','O(n)','O(n^2)',...
40          'Location','NorthWest')
41  title(sprintf('tic-toc timing averaged over %d runs',
      nruns),'fontsize',14);
42  xlabel('{\bf problem size n}','fontsize',14);
43  ylabel('{\bf runtime (s)}','fontsize',14);
44
45  print -depsc2 '../PICTURES/multAmin_timings.eps';
```

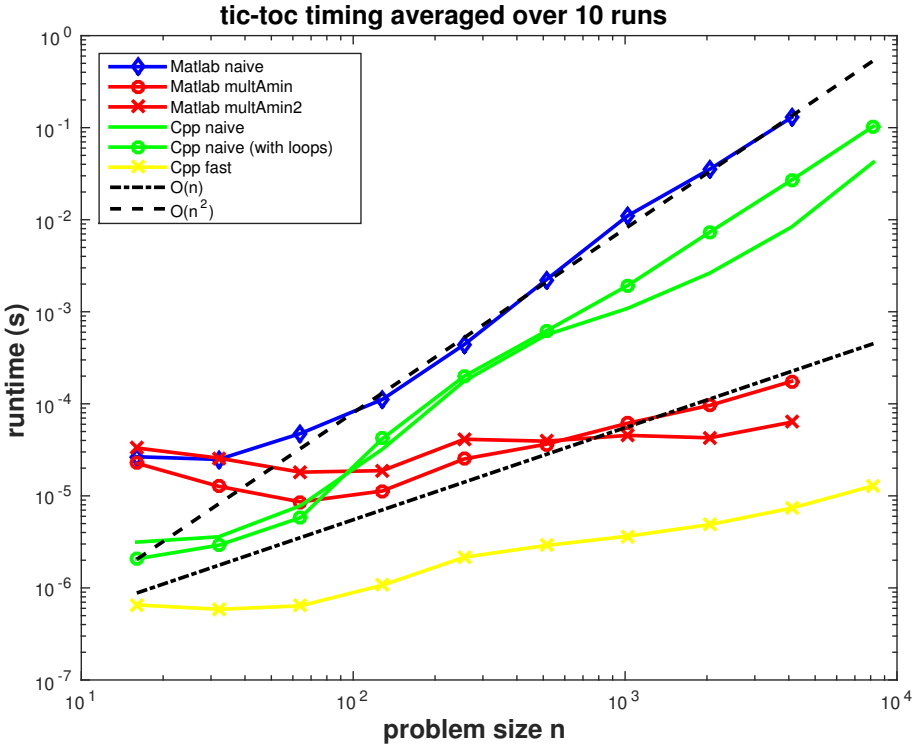

Figure 5: Timings for different implementations of $\mathbf{y} = \mathbf{A}\mathbf{x}$ with both MATLABand C++.

(4e) ⚂ Can you solve task (4b) without using any `for`- or `while`-loop? Implement it in the function

```
function y = multAmin2(x)
```

HINT: you may use the MATLABbuilt-in command `cumsum`.

Solution: Using `cumsum` to avoid the `for` loops:

Listing 17: implementation for the function multAmin without loops

```
function y = multAmin2(x)
% O(n), no-for version
n = length(x);
v = cumsum(x(end:-1:1));
w = cumsum(x.*(1:n)');
y = w + (1:n)'.*[v(n-1:-1:1);0];
```

(4f) ⚀ Consider the following MATLABscript `multAB.m`:

Listing 18: MATLABscript calling `multAmin`

```
1 n = 10;
2 B = diag(-ones(n-1,1),-1)+diag([2*ones(n-1,1);1],0)...
3     + diag(-ones(n-1,1),1);
4 x = rand(n,1);
5 fprintf('|x-y| = %d\n',norm(multAmin(B*x)-x));
```

Sketch the matrix $\mathbf{B}$ created in line 3 of `multAB.m`.

HINT: this MATLABscript is provided as file `multAB.m`.

Solution:

$$\mathbf{B} := \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 1 \end{pmatrix}$$

Notice the value 1 in the entry (n, n).

(4g) ⚁ Run the code of Listing 18 several times and conjecture a relationship between the matrices $\mathbf{A}$ and $\mathbf{B}$ from the output. Prove your conjecture.

HINT: You must take into account that computers inevitably commit round-off errors, see [1, Section 1.5].

Solution: It is easy to verify with MATLAB(or to prove) that $\mathbf{B} = \mathbf{A}^{-1}$.
For $2 \le j \le n-1$, we obtain:

$$(\mathbf{AB})_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} = a_{i,j-1} b_{j-1,j} + 2a_{i,j} b_{j,j} + a_{i,j+1} b_{j+1,j}$$

$$= -\min(i, j-1) + 2\min(i, j) - \min(i, j+1) = \begin{cases} -i + 2i - i = 0 & \text{if } i < j, \\ -(i-1) + 2i - i = 1 & \text{if } i = j, \\ -(j-1) + 2j - (j+1) = 0 & \text{if } i > j. \end{cases}$$

Furthermore, $(\mathbf{AB})_{i,1} = \delta_{i,1}$ and $(\mathbf{AB})_{i,n} = \delta_{i,n}$, hence $\mathbf{AB} = \mathbf{I}$.
The last line of `multAB.m` prints the value of $\|\mathbf{A}\,\mathbf{B}\,\mathbf{x} - \mathbf{x}\| = \|\mathbf{x} - \mathbf{x}\| = 0$.
The returned values are not exactly zero due to round-off errors.

(4h) ⚀ Implement a C++ function with declaration

```
template <class Vector>
void minmatmv(const Vector &x,Vector &y);
```

that realizes the efficient version of the MATLAB line of code (4). Test your function by comparing with output from the equivalent MATLAB functions.

Solution:

Listing 19: C++script implementing `multAmin`

```
#include <Eigen/Dense>

#include <iostream>
#include <vector>

#include "timer.h"

//! \brief build A*x using array of ranges and of ones.
//! \param[in] x vector x for A*x = y
//! \param[out] y y = A*x
void multAminSlow(const Eigen::VectorXd & x,
    Eigen::VectorXd & y) {
     unsigned int n = x.size();

     Eigen::VectorXd one = Eigen::VectorXd::Ones(n);
     Eigen::VectorXd linsp =
        Eigen::VectorXd::LinSpaced(n,1,n);
     y = ( (one * linsp.transpose()).cwiseMin(linsp *
        one.transpose()) ) *x;
}

//! \brief build A*x using a simple for loop
//! \param[in] x vector x for A*x = y
//! \param[out] y y = A*x
void multAminLoops(const Eigen::VectorXd & x,
    Eigen::VectorXd & y) {
```

```
23      unsigned int n = x.size();
24
25      Eigen::MatrixXd A(n,n);
26
27      for(unsigned int i = 0; i < n; ++i) {
28          for(unsigned int j = 0; j < n; ++j) {
29              A(i,j) = std::min(i+1,j+1);
30          }
31      }
32      y = A * x;
33  }
34
35  //! \brief build A*x using a clever representation
36  //! \param[in] x vector x for A*x = y
37  //! \param[out] y y = A*x
38  void multAmin(const Eigen::VectorXd & x, Eigen::VectorXd
        & y) {
39      unsigned int n = x.size();
40      y = Eigen::VectorXd::Zero(n);
41      Eigen::VectorXd v = Eigen::VectorXd::Zero(n);
42      Eigen::VectorXd w = Eigen::VectorXd::Zero(n);
43
44      v(0) = x(n-1);
45      w(0) = x(0);
46
47      for(unsigned int j = 1; j < n; ++j) {
48          v(j) = v(j-1) + x(n-j-1);
49          w(j) = w(j-1) + (j+1)*x(j);
50      }
51      for(unsigned int j = 0; j < n-1; ++j) {
52          y(j) = w(j) + v(n-j-2)*(j+1);
53      }
54      y(n-1) = w(n-1);
55  }
56
57  int main(void) {
58      // Build Matrix B with 10x10 dimensions such that B
           = inv(A)
```

```
59      unsigned int n = 10;
60      Eigen::MatrixXd B = Eigen::MatrixXd::Zero(n,n);
61      for(unsigned int i = 0; i < n; ++i) {
62          B(i,i) = 2;
63          if(i < n-1) B(i+1,i) = -1;
64          if(i > 0) B(i-1,i) = -1;
65      }
66      B(n-1,n-1) = 1;
67      std::cout << "B = " << B << std::endl;
68
69      // Check that B = inv(A) (up to machine precision)
70      Eigen::VectorXd x = Eigen::VectorXd::Random(n), y;
71      multAmin(B*x, y);
72      std::cout << "|y-x| = " << (y - x).norm() <<
           std::endl;
73      multAminSlow(B*x, y);
74      std::cout << "|y-x| = " << (y - x).norm() <<
           std::endl;
75      multAminLoops(B*x, y);
76      std::cout << "|y-x| = " << (y - x).norm() <<
           std::endl;
77
78      // Timing from 2^4 to 2^13 repeating nruns times
79      timer<> tm_slow, tm_slow_loops, tm_fast;
80      std::vector<int> times_slow, times_slow_loops,
           times_fast;
81      unsigned int nruns = 10;
82      for(unsigned int p = 4; p ≤ 13; ++p) {
83          tm_slow.reset();
84          tm_slow_loops.reset();
85          tm_fast.reset();
86          for(unsigned int r = 0; r < nruns; ++r) {
87              x = Eigen::VectorXd::Random(pow(2,p));
88
89              tm_slow.start();
90              multAminSlow(x, y);
91              tm_slow.stop();
92
```

```
93              tm_slow_loops.start();
94              multAminLoops(x, y);
95              tm_slow_loops.stop();
96
97              tm_fast.start();
98              multAmin(x, y);
99              tm_fast.stop();
100         }
101         times_slow.push_back( tm_slow.avg().count() );
102         times_slow_loops.push_back(
               tm_slow_loops.avg().count() );
103         times_fast.push_back( tm_fast.avg().count() );
104     }
105
106     for(auto it = times_slow.begin(); it !=
          times_slow.end(); ++it) {
107         std::cout << *it << " ";
108     }
109     std::cout << std::endl;
110     for(auto it = times_slow_loops.begin(); it !=
          times_slow_loops.end(); ++it) {
111         std::cout << *it << " ";
112     }
113     std::cout << std::endl;
114     for(auto it = times_fast.begin(); it !=
          times_fast.end(); ++it) {
115         std::cout << *it << " ";
116     }
117     std::cout << std::endl;
118 }
```

Problem 5. Matrix powers

(5a) ⚀ Implement a MATLAB function

$$\texttt{Pow(A,k)}$$

that, using only basic linear algebra operations (including matrix-vector or matrix-matrix multiplications), computes efficiently the k^{th} power of the $n \times n$ matrix $\mathbf{A}$.

HINT: use the MATLAB operator $\wedge$ to test your implementation on random matrices $\mathbf{A}$.

HINT: use the MATLAB functions `de2bi` to extract the "binary digits" of an integer.

Solution: Write k in binary format: $k = \sum_{j=0}^{M} b_j\, 2^j,\ b_j \in \{0,1\}$. Then

$$\mathbf{A}^k = \prod_{j=0}^{M} \mathbf{A}^{2^j\, b_j} = \prod_{j\ s.t.\ b_j=1} \mathbf{A}^{2^j}.$$

We compute $\mathbf{A},\ \mathbf{A}^2,\ \mathbf{A}^4,\ \ldots,\mathbf{A}^{2^M}$ (one matrix-matrix multiplication each) and we multiply only the matrices $\mathbf{A}^{2^j}$ such that $b_j \neq 0$.

Listing 20: An efficient implementation for Problem 5.

```
function X = Pow (A, k)
% Pow - Return A^k for square matrix A and integer k
% Input:    A:    n*n matrix
%           k:    positive integer
% Output:   X:    n*n matrix X = A^k

% transform k in basis 2
bin_k = de2bi(k) ;
M = length(bin_k);
X = eye(size(A));

for j = 1:M
    if bin_k(j) == 1
        X = X*A;
    end
    A = A*A;     % now A{new} = A{initial} ^(2^j)
end
```

(5b) ⊡ Find the asymptotic complexity in k (and n) taking into account that in MATLAB a matrix-matrix multiplication requires a $O(n^3)$ effort.

Solution: Using the simplest implementation:

$$A^k = \underbrace{\Big(\ldots\big(\big(A\cdot A\big)\cdot A\big)\ldots\cdot A\Big)\cdot A}_{k} \qquad \rightarrow \qquad O\big((k-1)n^3\big).$$

Using the efficient implementation from Listing 20, for each $j \in \{1, 2, \ldots, \log_2(k)\}$ we have to perform at most two multiplications ($X * A$ and $A * A$):

$$\text{complexity} \quad \leq \quad 2 * M * \text{matrix-matrix mult.} \quad \approx \quad 2 * \lceil \log_2 k \rceil * n^3.$$

($\lceil a \rceil = \texttt{ceil}(a) = \inf\{b \in \mathbb{Z},\ a \leq b\}$).

(5c) ⊡ Plot the runtime of the built-in MATLAB power ($\wedge$) function and find out the complexity. Compare it with the function `Pow` from (5a).
Use the matrix

$$A_{j,k} = \frac{1}{\sqrt{n}} \exp\left(\frac{2\pi i\, jk}{n}\right)$$

to test the two functions.

Solution:

Listing 21: Timing plots for Problem 5.

```
clear all; close all;
nruns = 10;                    % we average over a few runs
nn = 30:3:60;                  % dimensions used
kk = (2:50);                   % powers used
tt1 = zeros(length(nn), length(kk));   % times for Pow
tt2 = zeros(length(nn), length(kk));   % times for Matlab
    power

for i = 1:length(nn)
    n = nn(i);

    % matrix with |eigenvalues| =1:
    % A = vander([exp(1i * 2 * pi * [1:n]/n)])/sqrt(n);
    A = exp(1i * 2 * pi * [1:n]'*[1:n]/n)/sqrt(n);

    for j=1:length(kk)
        k = kk(j);
        tic
        for run = 1:nruns
            X = Pow(A, k);
        end
        tt1(i,j) = toc;
```

```
22
23          tic
24          for run = 1:nruns
25              XX = A^k;
26          end
27          tt2(i,j) = toc;
28          n_k_err = [n, k, max(max(abs(X-XX)))]
29      end
30
31  end
32
33  figure('name','Pow timings');
34  subplot(2,1,1)
35  n_sel=6;              %plot in k only for a selected n
36  % expected logarithmic dep. on k, semilogX used:
37  semilogx(kk,tt1(n_sel,:),'m+',
      kk,tt2(n_sel,:),'ro',...
38          kk,sum(tt1(n_sel,:))*log(kk)/(length(kk)*log(k)),
              'linewidth', 2);
39  xlabel('{\bf power k}','fontsize',14);
40  ylabel('{\bf average runtime (s)}','fontsize',14);
41  title(sprintf('tic-toc timing averaged over %d runs,
      matrix size = %d',...
42        nruns, nn(n_sel)),'fontsize',14);
43  legend('our implementation','Matlab built-in',...
44         'O(C log(k))','location','northwest');
45
46  subplot(2,1,2)
47  k_sel = 35;       %plot in n only for a selected k
48  loglog(nn, tt1(:,k_sel),'m+',      nn,
      tt2(:,k_sel),'ro',...
49          nn, sum(tt1(:,k_sel))* nn.^3/sum(kk.^3),
              'linewidth', 2);
50  xlabel('{\bf dimension n}','fontsize',14);
51  ylabel('{\bf average runtime (s)}','fontsize',14);
52  title(sprintf('tic-toc timing averaged over %d runs,
      power = %d',...
53        nruns, kk(k_sel)),'fontsize',14);
```

```
54 legend('our implementation','Matlab built-in',...
55         'O(n^3)','location','northwest');
56 print -depsc2 'Pow_timings.eps';
```

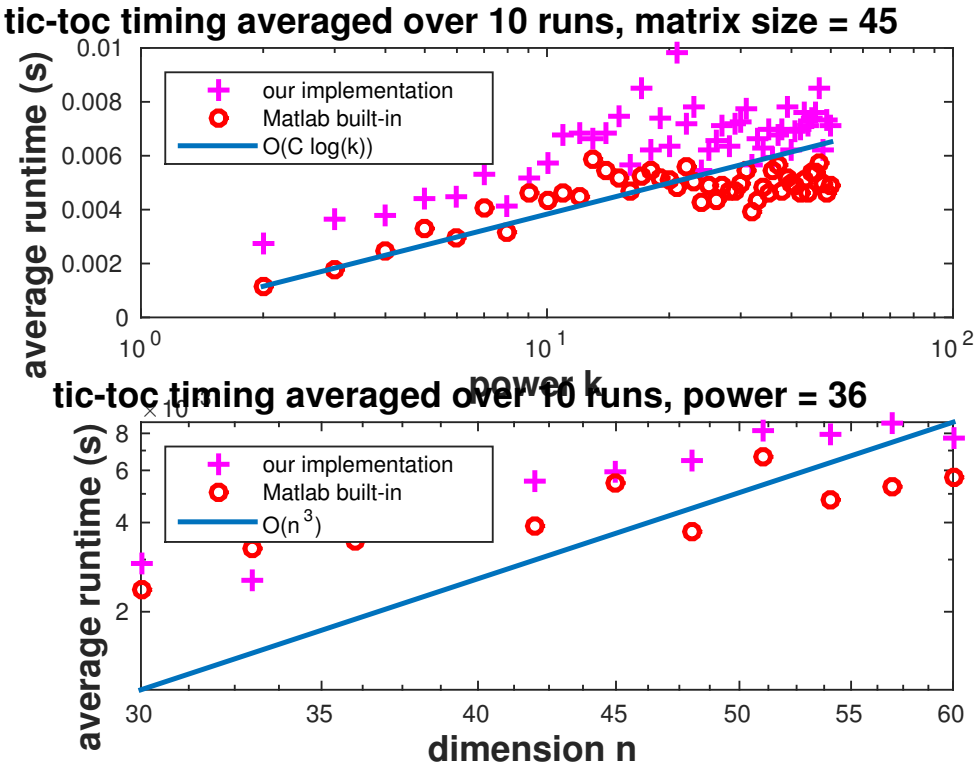

Figure 6: Timings for Problem 5.

The MATLAB∧-function has (at most) logarithmic complexity in k but the timing is slightly better than our implementation.

All the eigenvalues of the Vandermonde matrix A have absolute value 1, so the powers A^k are "stable": the eigenvalues of A^k are not approaching neither 0 nor ∞ when k grows.

(5d) ⚀ Using EIGEN, devise a C++ function with the calling sequence

```
1 template <class Matrix>
2 void matPow(const Matrix &A,unsigned int k);
```

that computes the k^{th} power of the square matrix $\mathbf{A}$ (passed in the argument `A`). Of course, your implementation should be as efficient as the MATLAB version from sub-problem (5a).

HINT: matrix multiplication suffers no aliasing issues (you can safely write `A = A*A`).

HINT: feel free to use the provided `matPow.cpp`.

HINT: you may want to use `log` and `ceil`.

HINT: EIGEN implementation of power (`A.pow(k)`) can be found in:

```
#include <unsupported/Eigen/MatrixFunctions>
```

Solution:

Listing 22: Implementation of `matPow`

```
1  #include <Eigen/Dense>
2  #include <unsupported/Eigen/MatrixFunctions>
3  #include <iostream>
4  #include <math.h>
5
6  //! \brief Compute powers of a square matrix using smart
       binary representation
7  //! \param[in,out] A matrix for which you want to
       compute A^k. A^k is stored in A
8  //! \param[out] k integer for A^k
9  template <class Matrix>
10 void matPow(Matrix & A, unsigned int k) {
11     Matrix X = Matrix::Identity(A.rows(), A.cols());
12
13     // p is used as binary mask to check wether k = sum_{i=0}^{M} b_i 2^i
          has 1 in the i-th binary digit
14     // obtaining the binay representation of p can be
          done in many ways, here we use ¬k & p to check
          i-th binary is 1
15     unsigned int p = 1;
16     // Cycle all the way up to the length of the binary
          representation of k
17     for(unsigned int j = 1; j ≤ ceil(log2(k)); ++j) {
```

```
18          if( ( ¬k & p ) == 0 ) {
19              X = X*A;
20          }
21
22          A = A*A;
23          p = p << 1;
24      }
25      A = X;
26  }
27
28  int main(void) {
29      // Check/Test with provided, complex, matrix
30      unsigned int n = 3; // size of matrix
31      unsigned int k = 9; // power
32
33      double PI = M_PI; // from math.h
34      std::complex<double> I = std::complex<double>(0,1);
            // imaginary unit
35
36      Eigen::MatrixXcd A(n,n);
37
38      for(unsigned int i = 0; i < n; ++i) {
39          for(unsigned int j = 0; j < n; ++j) {
40              A(i,j) = exp(2. * PI * I * (double) i *
                    (double) j / (double) n) / sqrt((double)
                    n);
41          }
42      }
43
44      // Test with simple matrix/simple power
45  //      Eigen::MatrixXd A(2,2);
46  //      k = 3;
47  //      A << 1,2,3,4;
48
49      // Output results
50      std::cout << "A = " << A << std::endl;
51      std::cout << "Eigen:" << std::endl << "A^" << k << "
            = " << A.pow(k) << std::endl;
```

```
52      matPow(A, k);
53      std::cout << "Ours:" << std::endl << "A^" << k << "
           = " << A <<std::endl;
54  }
```

Problem 6. Complexity of a MATLAB function

In this problem we recall a concept from linear algebra, the diagonalization of a square matrix. Unless you can still define what this means, please look up the chapter on "eigenvalues" in your linear algebra lecture notes. This problem also has a subtle relationship with Problem 5.

We consider the MATLAB function defined in `getit.m` (cf. Listing 23)

Listing 23: MATLABimplementation of `getit` for Problem 6..

```
1  function y = getit(A, x, k)
2      [S,D] = eig(A);
3      y = S*diag(diag(D).^k)* (S\x);
4  end
```

HINT: Give the command `doc eig` in MATLAB to understand what `eig` does.

HINT: You may use that `eig` applied to an $n \times n$-matrix requires an asymptotic computational effort of $O(n^3)$ for $n \to \infty$.

HINT: in MATLAB, the function `diag(x)` for $\mathbf{x} \in \mathbb{R}^n$, builds a diagonal, $n \times n$ matrix with $\mathbf{x}$ as diagonal. If $\mathbf{M}$ is a $n \times n$ matrix, `diag(M)` returns (extracts) the diagonal of $\mathbf{M}$ as a vector in $\mathbb{R}^n$.

HINT: the operator `v.^k` for $v \in \mathbb{R}^n$ and $k \in \mathbb{N} \setminus \{0\}$ returns the vector with components v_i^k (i.e. component-wise exponent)

(6a) ⚀ What is the output of `getit`, when A is a diagonalizable $n \times n$ matrix, $x \in \mathbb{R}^n$ and $k \in \mathbb{N}$?

Solution: The output is $\mathbf{y}$ s.t. $\mathbf{y} = \mathbf{A}^k\mathbf{x}$. The eigenvalue decomposition of the matrix $\mathbf{A} = \mathbf{S}\mathbf{D}\mathbf{S}^{-1}$ (where $\mathbf{D}$ is diagonal and $\mathbf{S}$ invertible), allows us to write:

$$\mathbf{A}^k = (\mathbf{S}\mathbf{D}\mathbf{S}^{-1})^k = \mathbf{S}\mathbf{D}^k\mathbf{S}^{-1},$$

and $\mathbf{D}^k$ can be computed efficiently (component-wise) for diagonal matrices.

(6b) ⊡ Fix $k \in \mathbb{N}$. Discuss (in detail) the asymptotic complexity of `getit` $n \to \infty$.

Solution: The algorithm comprises the following operations:

- diagonalization of a full-rank matrix $\mathbf{A}$ is $O(n^3)$;
- matrix-vector multiplication is $O(n^2)$;
- raising a vector in $\mathbb{R}^n$ for the power k has complexity $O(n)$;
- solve a fully determined linear system: $O(n^3)$.

The complexity of the algorithm is dominated by the operations with higher exponent. Therefore the total complexity of the algorithm is $O(n^3)$ for $n \to \infty$.

Issue date: 17.09.2015
Hand-in: 24.09.2015 (in the boxes in front of HG G 53/54).

Version compiled on: September 24, 2015 (v. 1.0).