Problem Sheet 1

Numerical Methods for CSE

David Bimmler

1. Arrow matrix-vector multiplication

1.a. Matrix A

$$A = \begin{bmatrix} d_1 & 0 & 0 & \dots & a_1 \\ 0 & d_2 & 0 & \dots & a_2 \\ 0 & 0 & d_3 & \dots & a_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & a_3 & \dots & d_n \end{bmatrix}$$

1.b. `tic-toc` Results

We can clearly see that the asymptotic complexity of the function `arrowmatvec` lies in $O(n^3)$. In the first few measurements, the actual multiplication is being dominated by other factors, such as the calls to length and the creation of the matrix A. With increasing input size n, though, these factors become have a lesser impact, and most of the time is spend calculating the product, which is why the runtime approaches the graphed line.

From the code we can see why $O(n^3)$ is the expected asymptotic complexity: the matrix A gets multiplied with itself, which is an operation in $O(n^3)$ as the matrices are all square and of size n.

1.c. Efficient Reimplementation

We can avoid creating the matrix A entirely. Furthermore, we first simulate the multiplication of A with x and then simulate the second multiplication of A with the result. The multiplication with A is simulated by exploiting the arrow structure.

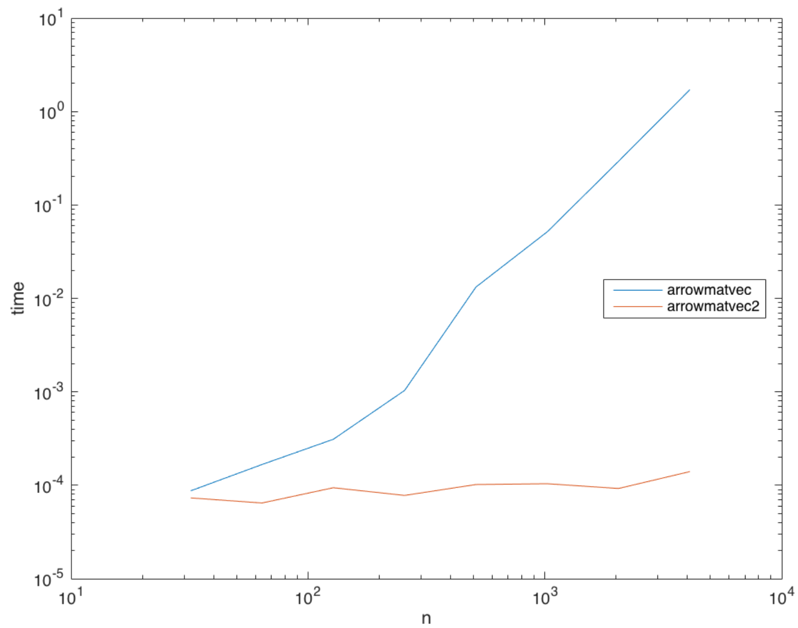
```
function y = arrowmatvec2(d,a,x)
if (length(d) ~= length(a)), error ('size mismatch'); end
a(end) = 0;
yy = d .* x + x(end) * a;
```

```
yy(end) = yy(end) + dot(a(1:end-1),x(1:end-1));
y = d .* yy + yy(end) * a;
y(end) = y(end) + dot(a(1:end-1),yy(1:end-1));
```

1.d. Complexity of Reimplementation

The more efficient version runs in $O(n)$.

1.e. Runtime Comparison



The above plot was generated using the following code:

```
scale = 2 .^ (5:12);
c = length(scale);
tries = 3;
res = zeros(c, 2);
tt = zeros(2,tries);
for i = 1:c
    n = scale(i);
    a = rand(n,1);
    d = rand(n,1);
    x = rand(n,1);
    for t = 1:tries
        tic
```

```
        y = arrowmatvec(d,a,x);
        tt(1,t) = toc;
        tic
        yy = arrowmatvec2(d,a,x);
        tt(2,t) = toc;
        if (y ~= yy)
            error ('mismatch');
        end
    end
    res(i,1) = min(tt(1,:));
    res(i,2) = min(tt(2,:));
end

loglog(scale, res(:,1));
hold on;
loglog(scale, res(:,2));
xlabel('n');
ylabel('time');
legend('arrowmatvec', 'arrowmatvec2', 'Location', 'east');
```

1.f. Eigen implementations

1.f.1. arrowmatvec

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

using Eigen::MatrixXd;
using Eigen::VectorXd;

template <class Vector>
Vector arrowmatvec(const Vector & d, const Vector & a, const Vector & x) {
    int d_s = d.size();
    if (d.size() != a.size()) {
        std::cerr << "size mismatch";
        throw 1;
    }
    MatrixXd A(d_s, d_s);
    A.rightCols(1) = a;
    A.bottomRows(1) = a.transpose();
    A.diagonal() = d;
    return A*A*x;
}
```

1.f.2. arrowmatvec2

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

using Eigen::MatrixXd;
using Eigen::VectorXd;

template <class Vector>
Vector arrowmatvec2(const Vector & d, const Vector & a, const Vector & x) {
    long n = d.size();
    if (n != a.size()) {
        std::cerr << "size mismatch";
        throw 1;
    }
    Vector a_ = a;
    a_(n-1) = 0;
    Vector yy = (d.array() * x.array()).matrix() + x(n-1) * a_;
    yy(n-1) += a_.topRows(n).dot(x.topRows(n));
    Vector y = (d.array() * yy.array()).matrix() + yy(n-1) * a_;
    y(n-1) += a_.topRows(n).dot(yy.topRows(n));
    return y;
}
```

2. Avoiding Cancellation

2.a. Behaviour of the Error

2.a.1. Derivation of f_2

$$f_1(x_0, h) \coloneqq \sin(x_0 + h) - \sin(x_0) \tag{1}$$

$$\sin(x_0 + h) - \sin(x_0) = 2\cos(x_0 + \frac{h}{2})\sin(\frac{h}{2}) \eqqcolon f_2(x_0, h) \tag{2}$$

2.a.2. Approximation for $f'(x)$

$$f'(x) \approx \frac{f(x_0 + h) - f(x_0)}{h} = \frac{f_2(x_0, h)}{h} \tag{3}$$

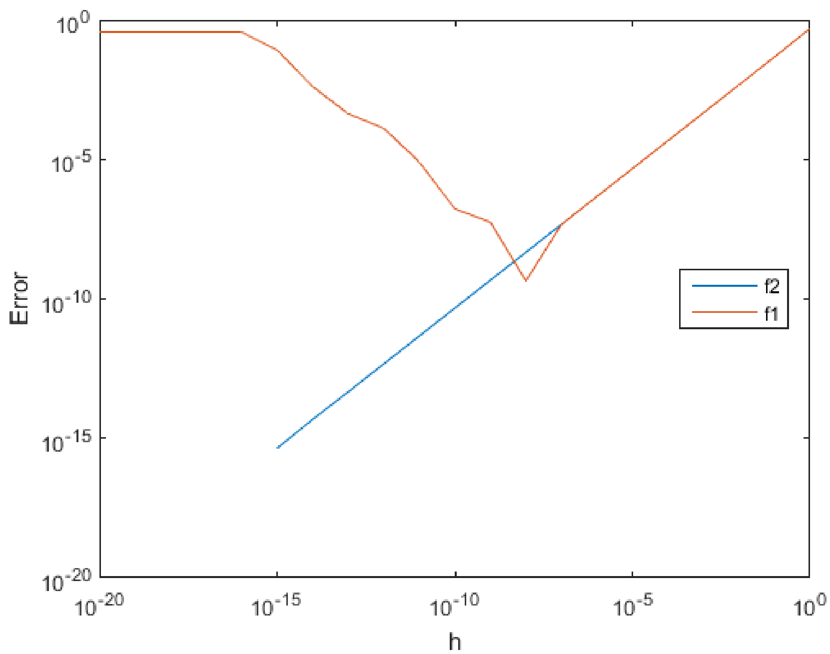
The following code implements this formula:

```
function res = nme1p2a
    x = 1.2;
    res = zeros(21,2);
```

```
    for i = -20:0
        h = 10^i;
        res(i+21,:) = [f2(x,h)/h,f1(x,h)/h];
    end

function y = f1(x0,h)
y = sin(x0 + h) - sin(x0);

function y = f2(x0, h)
y = 2 * cos(x0 + (h/2)) * sin(h/2);
```

`f1` is only included here to make the plotting easier afterwards.

2.a.3. Error plot



The plot is generated using the following code:

```
res = nme1p2a';
scale = 10 .^ (-20:0);
loglog(scale, abs(res(1,:) - cos(1.2)))
hold on;
loglog(scale, abs(res(2,:) - cos(1.2)))
legend('f2','f1','Location','east')
xlabel('h')
ylabel('Error')
```

2.a.4. Explanation of Error Behaviour

The catastrophic effects of cancellation are visible in the plot of the error in `f1`. When h gets too small, the cancellation error dominates the approximation error. In `f2`, no cancellation error is present, and the approximation error decreases with h.

2.b. Suitability for Numerical Computation

$$\ln\left(x - \sqrt{x^2 - 1}\right) = \ln\left(x - \sqrt{x^2 - 1} * \frac{x + \sqrt{x^2 - 1}}{x + \sqrt{x^2 - 1}}\right) \tag{4}$$

$$\ln\left(x - \sqrt{x^2 - 1} * \frac{x + \sqrt{x^2 - 1}}{x + \sqrt{x^2 - 1}}\right) = \ln\left(\frac{1}{x + \sqrt{x^2 - 1}}\right) = -\ln\left(x + \sqrt{x^2 - 1}\right) \tag{5}$$

The second formula is much more suitable for numerical computation, since the first one will suffer from cancellation, as the term $x - \sqrt{x^2 - 1}$ matches the criteria of a cancellation-error prone calculation. For $x = 60000000$ we already observe an error of 0.112, which is significant.

2.c. Numerical Difficulties

2.c.1. $\sqrt{x + \frac{1}{x}} - \sqrt{x - \frac{1}{x}}$ with $x \gg 1$

In this expression cancellation will occur, since the two terms that get subtracted are very close to the same size. We can apply the same trick as before:

$$\sqrt{x + \frac{1}{x}} - \sqrt{x - \frac{1}{x}} = \frac{2}{x * \left(\sqrt{x + \frac{1}{x}} + \sqrt{x - \frac{1}{x}}\right)} \tag{6}$$

This expression does not suffer from cancellation.

2.c.2. $\sqrt{\frac{1}{a^2} + \frac{1}{b^2}}$ with $a \approx 0, b \approx 1$

The problem in this expression is that $\frac{1}{a^2}$ blows up the error in a. Furthermore since $\frac{1}{a^2}$ becomes much bigger than $\frac{1}{b^2}$, truncation occurs. The following rewrite offers an improvement:

$$\sqrt{\frac{1}{a^2} + \frac{1}{b^2}} = \sqrt{\frac{a^2 + b^2}{a^2 b^2}} = \frac{\sqrt{a^2 + b^2}}{ab} \tag{7}$$

3. Kronecker Product

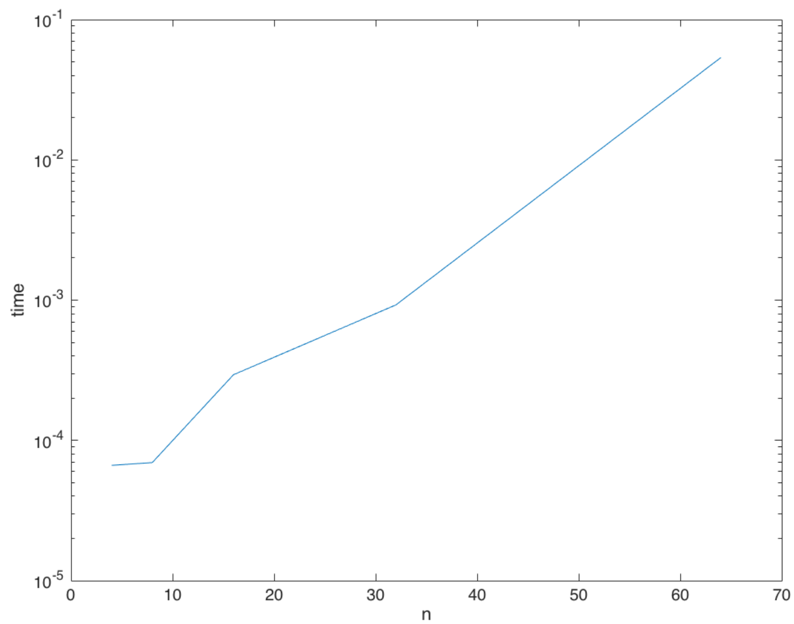
3.b. Matrix M

$$M = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{bmatrix}$$

3.c. Asymptotic Complexity

The asymptotic complexity of `y = kron(A,B) * x` is $O(n^4)$.

3.d. Measured Runtimes



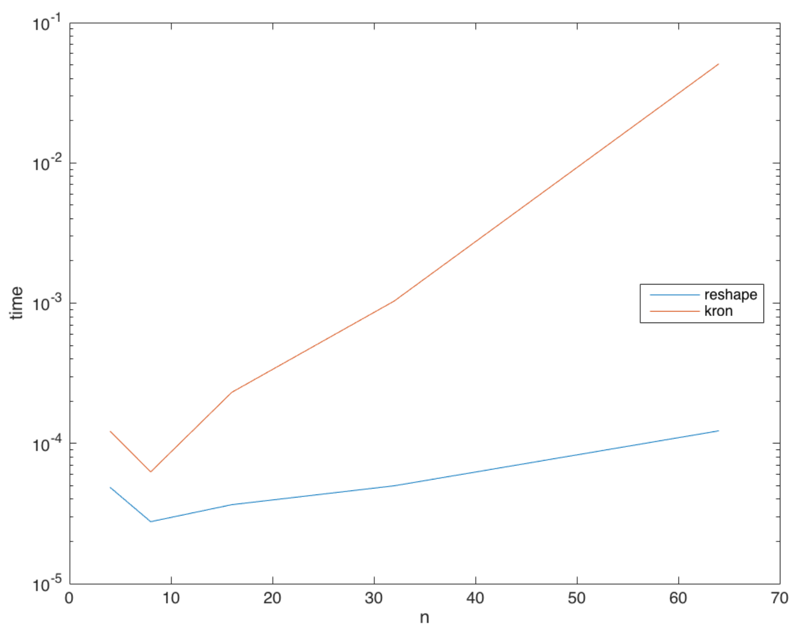
The above plot was generated using the following code:

```
res = zeros(1,5);
tt = zeros(3,5);
scale = 2 .^(2:6);
for i = 1:5
    n = 2^(i+1);
    A = rand(n);
    B = rand(n);
    x = rand(n^2,1);
```

```
    for t = 1:3
        tic
        y = kron(A,B) * x;
        tt(t, i) = toc;
    end
    res(i) = min(tt(:,i));
end

semilogy(scale, res);
xlabel('n');
ylabel('time');
```

3.e. Discussion of Reshape Implementation

The outer call to `reshape` simply reformats the n by n matrix as an n^2 by 1 column vector, and the inner call to `reshape` creates a square matrix of size n from the contents of the vector x. The interesting part is the multiplication $B \times x' \times A^T$ where x' is a n by n matrix formed from the elements of x, but I am unable to explain the equivalence properly.



The above plot was generated using the following code:

```
tt = zeros(3,5);
tt2 = zeros(3,5);
res = zeros(1,5);
```

```
res2 = zeros(1,5);
scale = 2 .^(2:6);
for i = 1:5
    n = 2^(i+1);
    A = rand(n);
    B = rand(n);
    x = rand(n^2,1);
    for t = 1:3
        tic
        y = reshape( B * reshape(x,n,n) * A', n*n, 1 );
        tt(t, i) = toc;
        tic
        y = kron(A,B) * x;
        tt2(t, i) = toc;
    end
    res(i) = min(tt(:,i));
    res2(i) = min(tt2(:,i));
end

semilogy(scale, res);
hold on;
semilogy(scale, res2);
xlabel('n');
ylabel('time');
legend('reshape', 'kron', 'Location', 'east');
```

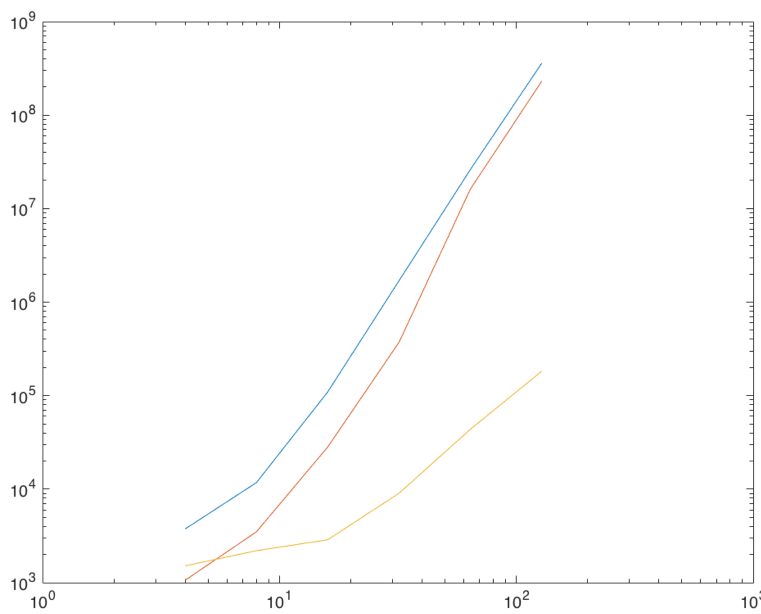
3.f. Eigen Implementation of kron, kron_fast and kron_super_fast

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

#include "timer.h"

//! \brief Compute the Kronecker product $C = A \otimes B$.
//! \param[in] A Matrix $n \times n$
//! \param[in] B Matrix $n \times n$
//! \param[out] C Kronecker product of A and B of dim $n^2 \times n^2$
template <class Matrix>
void kron(const Matrix & A, const Matrix & B, Matrix & C)
{
    long n = A.rows();
    if (A.rows() != B.rows()) {
        std::cerr << "size mismatch";
        throw 1;
```

```
    }
    Matrix res(n * n, n * n);

    int col = 0;
    for (size_t i = 0, size = A.size(); i < size; i++)
    {
        if (i > 0 && i % n == 0) {
            col++;
        }
        double a = (*(A.data() + i));
        res.block((i%n)*n,col*n,n,n) = a * B;
    }
    C = res;
}

//! \brief Compute the Kronecker product C = $A \otimes B$. Exploit matrix-vector product.
//! A,B and x must have dimension n \times n resp. n
//! \param[in] A Matrix $n \times n$
//! \param[in] B Matrix $n \times n$
//! \param[in] x Vector of dim $n$
//! \param[out] y Vector y = kron(A,B)*x
template <class Matrix, class Vector>
void kron_fast(const Matrix & A, const Matrix & B, const Vector & x, Vector & y)
{
    long n = A.rows();
    if (n != B.rows()) {
        std::cerr << "size mismatch";
        throw 1;
    }

    Matrix res(n * n, n * n);

    int col = 0;
    for (size_t i = 0, size = A.size(); i < size; i++)
    {
        if (i > 0 && i % n == 0) {
            col++;
        }
        double a = (*(A.data() + i));
        res.block((i%n)*n,col*n,n,n) = a * B;
    }
    y = res * x;
}

//! \brief Compute the Kronecker product $C = A \otimes B$. Uses fast remapping tecniques (s
//! A,B and x must have dimension n \times n resp. n*n
```

```
//! \param[in] A Matrix $n \times n$
//! \param[in] B Matrix $n \times n$
//! \param[in] x Vector of dim $n$
//! \param[out] y Vector y = kron(A,B)*x
template <class Matrix, class Vector>
void kron_super_fast(const Matrix & A, const Matrix & B, const Vector & x, Vector & y)
{
    long n = A.rows();
    if (n != B.rows()) {
        std::cerr << "size mismatch";
        throw 1;
    }

    Matrix X(n,n);
    for (int i = 0; i < n; i++) {
        X.block(0,i,n,1) = x.segment(i*n,n);
    }
    Matrix Y = (B * X * A.adjoint());
    y = Eigen::Map<Vector>(Y.data(), n*n);
}
```

3.i. Runtime comparisons of Eigen implementations



```
data = [3754 11751 109742 1697966 26194731 358813933;
1060 3500 28289 372684 16207242 230059816;
1512 2197 2875 9050 43683 182527];

scale = 2 .^(2:7);

loglog(scale, data(1,:), scale, data(2,:), scale, data(3,:));
```