Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 09

```
begin
    using PlutoUI
    using PlutoVista
    using BenchmarkTools
    using LinearAlgebra
end
```

Linear System Solution
Matrix & vector norms
Vector norms
Matrix norms
Condition number and error propagation
Complexity: "big O notation"
A Direct method: Cramer's rule
LU decomposition
Gaussian elimination
A first LU decomposition
Doolittles method
Pivoting
LU Factorization from Julia library
LU vs. inv
Some performance tests.

Linear System Solution

- Let A be a real $N \times N$ matrix, $b \in \mathbb{R}^N$.
- Solve $Ax = b$

Direct methods:

- Exact up to machine precision
- Sometimes expensive, sometimes not

Iterative methods:

- "Only" approximate
- With good convergence and proper accuracy control, results may be not worse than for direct methods
- Sometimes expensive, sometimes not
- Convergence guarantee is problem dependent and can be tricky

Matrix & vector norms

Vector norms

let $x = (x_i) \in \mathbb{R}^n$

v = [0.2, 0.3, 1.0]

```
v=[0.2, 0.3,  1.0]
```

$||x||_1 = \sum_{i=1}^n |x_i|$: sum norm, l_1-norm

1.5

```
norm(v,1)
```

$||x||_2 = \sqrt{\sum_{i=1}^n x_i^2}$: Euclidean norm, l_2-norm

```
(1.06301, 1.06301)
```

```
norm(v,2),norm(v)
```

$||x||_\infty = \max_{i=1}^n |x_i|$: maximum norm, l_∞-norm

```
1.0
```

```
norm(v,Inf)
```

Matrix norms

Given a matrix $A = (a_{ij}) \in \mathbb{R}^n \times \mathbb{R}^n$

- A matrix is a representation of a linear operator in the basis defined by the unit vectors: $\mathcal{A} : \mathbb{R}^n \to \mathbb{R}^n$ is defined via $\mathcal{A} : x \mapsto y = Ax$ with $y_i = \sum_{j=1}^n a_{ij}x_j$
- Vector norm $||\cdot||_p$ induces corresponding matrix norm:

$$
\begin{aligned}
||A||_p &:= \max_{x\in\mathbb{R}^n, x\neq 0} \frac{||Ax||_p}{||x||_p} \\
&= \max_{x\in\mathbb{R}^n, ||x||_p=1} \frac{||Ax||_p}{||x||_p}
\end{aligned}
$$

```
A0 = 3×3 Matrix{Float64}:
      3.0  2.0  3.0
      0.1  0.3  0.5
      0.6  2.0  3.0
```

```
A0=[3.0 2.0 3.0;
    0.1 0.3 0.5;
    0.6 2.0 3.0]
```

$||A||_1 = \max_{j=1}^n \sum_{i=1}^n |a_{ij}|$: maximum of column sums of absolute values of entries

```
6.5
```

```
opnorm(A0,1)
```

$||A||_2 = \sqrt{\lambda_{max}}$ with λ_{max}: largest eigenvalue of A^TA.

```
(5.78018, 5.78018, 5.78018)
```

```
opnorm(A0,2), opnorm(A0),sqrt(maximum(eigvals(A0'*A0)))
```

$||A||_\infty = \max_{i=1}^n \sum_{j=1}^n |a_{ij}| = ||A^T||_1$: maximum of row sums of absolute values of entries

```
(8.0, 8.0)
```

```
opnorm(A0,Inf),opnorm(A0',1)
```

Condition number and error propagation

- Solve $Ax = b$, where b is inexact
- Let Δb be the error in b and Δx be the resulting error in x such that

$$A(x + \Delta x) = b + \Delta b.$$

- Since $Ax = b$, we get $A\Delta x = \Delta b$ and $\Delta x = A^{-1}\Delta b$

$$\begin{aligned} Ax &= b \\ ||A|| \cdot ||x|| &\geq ||b|| \\ ||\Delta x|| &\leq ||A^{-1}|| \cdot ||\Delta b|| \\ \Rightarrow \frac{||\Delta x||}{||x||} &\leq \kappa(A)\frac{||\Delta b||}{||b||} \end{aligned}$$

where $\kappa(A) = ||A|| \cdot ||A^{-1}||$ is the *condition number* of A.

This means that the relative error in the solution is proportional to the relative error of the right hand side. The proportionality factor $\kappa(A)$ is usually larger (and in many relevant cases significantly larger) than one.

Just remark that this estimates does not assume inexact arithmetics.

Let us have an example. We use alternatively rational or floating point arithmetic:

Rational: ☑

```
T_test=use_rational ? Rational{Int64} : Float64;
```

```
a = 1000000//1
```

```
a=T_test(1_000_000)
```

```
pert_b = 1//1000000000
```

```
pert_b=T_test(1//1_000_000_000)
```

```
A = 2×2 Matrix{Rational{Int64}}:
          1//1       -1//1
     1000000//1  1000000//1
```

```
A=[ 1 -1;
    a  a]
```

```
2×2 Matrix{Rational{Int64}}:
  1//2  1//2000000
 -1//2  1//2000000
```

```
inv(A)
```

Assume a solution vector:

```
x =  [1, 1]
```

```
x=[1,1]
```

Create corresponding right hand side:

```
b =  [0//1, 2000000//1]
```

```
b=A*x
```

Define a perturbation of the right hand side:

```
Δb =  [1//1000000000, 1//1000000000]
```

```
Δb=[pert_b, pert_b]
```

Calculate the error with respect to the unperturbed solution:

```
Δx =  [1000001//2000000000000000, -999999//2000000000000000]
```

```
Δx=inv(A)*(b+Δb)-x
```

Relative error of right hand side:

```
δb = 7.071067811865475e-16
δb=norm(Δb)/norm(b)
```

Relative error of solution:

```
δx = 5.000000000002499e-10
δx=norm(Δx)/norm(x)
```

Comparison with condition number based estimate:

```
κ = 1.0e6
κ=opnorm(A)*opnorm(inv(A))
```

```
(1.0e6, 7.07107e5)
κ,δx/δb
```

Complexity: "big O notation"

Let $f, g : \mathbb{V} \to \mathbb{R}^+$ be some functions, where $\mathbb{V} = \mathbb{N}$ or $\mathbb{V} = \mathbb{R}$.

Write

$$f(x) = O(g(x)) \quad (x \to \infty)$$

if there exists a constant $C > 0$ and $x_0 \in \mathbb{V}$ such that $\forall x > x_0, \quad |f(x)| \leq C|g(x)|$

Often, one skips the part "$(x \to \infty)$"

Examples:

- Addition of two vectors: $O(N)$
- Straightforward matrix-vector multiplication (for matrix where all entries are assumed to be nonzero): $O(N^2)$

A Direct method: Cramer's rule

Solve $Ax = b$ by Cramer's rule:

$$x_i = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1i-1} & b_1 & a_{1i+1} & \dots & a_{1N} \\ a_{21} & & \dots & & b_2 & & \dots & a_{2N} \\ \vdots & & & & \vdots & & & \vdots \\ a_{N1} & & \dots & & b_N & & \dots & a_{NN} \end{vmatrix} / |A| \quad (i = 1 \dots N)$$

This takes $O(N!)$ operations...

LU decomposition

Gaussian elimination

So let us have a look at Gaussian elimination to solve $Ax = b$. The elementary matrix manipulation step in Gaussian elimination ist the multiplication of row k by $-a_{jk}/a_{kk}$ and its addition to row j such that element $_{jk}$ in the resulting matrix becomes zero. If this is done at once for all $j > k$, we can express this operation as the left multiplication of A by a lower triangular Gauss transformation matrix $M(A, k)$.

```
function M(A,k)
    n=size(A,1)
    m=UnitLowerTriangular(Matrix{eltype(A)}(one(eltype(A))I,n,n))
    for j=k+1:n
        m[j,k]=-A[j,k]/A[k,k]
    end
    m
end;
```

Use rational numbers for the following examples: ☑

```
T_lu = Rational{Int64}
```

```
T_lu= lu_use_rational ? Rational{Int64} : Float64
```

Define a test matrix:

```
A1 = 4×4 Matrix{Rational{Int64}}:
      2//1  1//1  3//1  4//1
      5//1  6//1  7//1  8//1
      7//1  6//1  8//1  5//1
      3//1  4//1  2//1  2//1
```

```
A1=T_lu[2 1 3 4;
        5 6 7 8;
        7 6 8 5;
        3 4 2 2;]
```

This is the Gauss transform for first column:

```
4×4 UnitLowerTriangular{Rational{Int64}, Matrix{Rational{Int64}}}:
  1//1   ⋅     ⋅     ⋅
 -5//2  1//1   ⋅     ⋅
 -7//2  0//1  1//1   ⋅
 -3//2  0//1  0//1  1//1
```

```
M(A1,1)
```

Applying it then sets all elements in the first column to zero besides of the main diagonal element:

```
U1 = 4×4 Matrix{Rational{Int64}}:
      2//1  1//1   3//1   4//1
      0//1  7//2  -1//2  -2//1
      0//1  5//2  -5//2  -9//1
      0//1  5//2  -5//2  -4//1
```

```
U1=M(A1,1)*A1
```

We can repeat this with the second column:

```
U2 = 4×4 Matrix{Rational{Int64}}:
      2//1  1//1    3//1    4//1
      0//1  7//2   -1//2   -2//1
      0//1  0//1  -15//7  -53//7
      0//1  0//1  -15//7  -18//7
```

```
U2=M(U1,2)*U1
```

And the third column:

```
U3 = 4×4 Matrix{Rational{Int64}}:
      2//1  1//1    3//1    4//1
      0//1  7//2   -1//2   -2//1
      0//1  0//1  -15//7  -53//7
      0//1  0//1    0//1    5//1
```

```
U3=M(U2,3)*U2
```

And here, we arrived at a triangular matrix. In the standard Gaussian elimination we would have manipulated the right hand side accordingly.

From here on we would start the backsubstitution which in fact is the solution of a triangular system of equations.

Thus, $A1 = LU$ with

```
L = 4×4 UnitLowerTriangular{Rational{Int64}, Matrix{Rational{Int64}}}:
     1//1   ⋅     ⋅     ⋅
     5//2  1//1   ⋅     ⋅
     7//2  5//7  1//1   ⋅
     3//2  5//7  1//1  1//1
```

```
L=inv(M(A1,1))*inv(M(U1,2))*inv(M(U2,3))
```

```
U = 4×4 Matrix{Rational{Int64}}:
     2//1  1//1    3//1    4//1
     0//1  7//2   -1//2   -2//1
     0//1  0//1  -15//7  -53//7
     0//1  0//1    0//1    5//1
```

```
U=U3
```

```
0.0
```

```
norm(L*U-A1)
```

```
true
```

```
iszero(L*U-A1)
```

A first LU decomposition

We can put this together into a function:

```
function my_first_lu_decomposition(A)
    n=size(A,1)
    L=Matrix(one(eltype(A))I,n,n) # L=I
    U=A
    for k=1:n-1
       Mtran=M(U,k)
       L=L*inv(Mtran)
       U=Mtran*U
    end
    # Check if matrices are triangular
    Ltri=UnitLowerTriangular(L)
    @assert iszero(Ltri-L)
    Utri=UpperTriangular(U)
    @assert iszero(Utri-U)
    Ltri,Utri
end;
```

```
 (
   1:  4×4 UnitLowerTriangular{Rational{Int64}, Matrix{Rational{Int64}}}:
        1//1   ⋅     ⋅     ⋅
        5//2  1//1   ⋅     ⋅
        7//2  5//7  1//1   ⋅
        3//2  5//7  1//1  1//1
   2:  4×4 UpperTriangular{Rational{Int64}, Matrix{Rational{Int64}}}:
        2//1  1//1    3//1    4//1
         ⋅    7//2   -1//2   -2//1
         ⋅     ⋅    -15//7  -53//7
         ⋅     ⋅       ⋅      5//1
)
```

```
Lx,Ux=my_first_lu_decomposition(A1)
```

Check for correctness:

```
0.0
```

```
norm(Lx*Ux-A1)
```

```
true
```

```
iszero(Lx*Ux-A1)
```

Solving $LUx = b$ then amounts to solve two triangular systems:

$$Ly = b$$
$$Ux = y$$

```
my_first_lu_solve (generic function with 1 method)
```

```julia
function my_first_lu_solve(L,U,b)
   y=inv(L)*b
   x=inv(U)*y
end
```

```
b1 =  [1, 2, 3, 4]
```

```julia
b1=[1,2,3,4]
```

```
x1 =  [182//75, -7//75, -154//75, 3//5]
```

```julia
x1=my_first_lu_solve(Lx,Ux,b1)
```

Check...

```
[0//1, 0//1, 0//1, 0//1]
```

```julia
A1*x1-b1
```

... in order to be didactical, in this example, we made a very inefficient implementation by creating matrices in each step. We even cheated by using `inv` in order to solve a triangular system.

Doolittles method

- Doolittles method (Adapted from **wikipedia: LU_decomposition**)
- This allows to perform LU decomposition in-place.

```
doolittle_lu_decomposition! (generic function with 1 method)
```

```julia
function doolittle_lu_decomposition!(LU)
    n = size(LU,1)
    # decomposition of matrix, Doolittle's Method
    for i = 1:n
        for j = 1:(i - 1)
            alpha = LU[i,j];
            for k = 1:(j - 1)
                alpha = alpha - LU[i,k]*LU[k,j];
            end
            LU[i,j] = alpha/LU[j,j];
        end
        for j = i:n
            alpha = LU[i,j];
            for k = 1:(i - 1)
                alpha = alpha - LU[i,k]*LU[k,j];
            end
            LU[i,j] = alpha;
        end
    end

end
```

```
doolittle_lu_solve (generic function with 1 method)
```

```julia
function doolittle_lu_solve(LU,b)
    n = length(b);
    x = zeros(eltype(LU),n);
    y = zeros(eltype(LU),n);
    # LU= L+U-I
    # find solution of Ly = b
    for i = 1:n
        alpha = zero(eltype(LU));
        for k = 1:i
            alpha = alpha + LU[i,k]*y[k];
        end
        y[i] = b[i] - alpha;
    end
    # find solution of Ux = y
    for i = n:-1:1
        alpha = zero(eltype(LU));
        for k = (i + 1):n
            alpha = alpha + LU[i,k]*x[k];
        end
        x[i] = (y[i] - alpha)/LU[i, i];
    end
    x
end
```

```
end
```

We can then implement a method for linear system solution:

```
doolittle_solve (generic function with 1 method)
```

```
function doolittle_solve(A,b)
    LU=copy(A)
    doolittle_lu_decomposition!(LU)
    doolittle_lu_solve(LU,b)
end
```

```
x2 =  [182//75, -7//75, -154//75, 3//5]
```

```
x2=doolittle_solve(A1,b1)
```

```
[0//1, 0//1, 0//1, 0//1]
```

```
A1*x2-b1
```

Pivoting

So far, we ignored the possibility that a diagonal element becomes zero during the LU factorization procedure.

Pivoting tries to remedy the problem that during the algorithm, diagonal elements can become zero. Before undertaking the next Gauss transformation step, we can exchange rows such that we always divide by the largest of the remaining diagonal elements. This would then in fact result in a decompositon

$$PA = LU$$

where P is a permutation matrix which can be stored in an integer vector. This approach is called "partial pivoting". Full pivoting in addition would perform column permutations. This would result in another permutation matrix Q and the decomposition

$$PAQ = LU$$

Almost all practically used LU decomposition implementations use partial pivoting.

LU Factorization from Julia library

Julia implements a pivoting LU factorization

```
lu1 = LU{Rational{Int64}, Matrix{Rational{Int64}}}
      L factor:
      4×4 Matrix{Rational{Int64}}:
       1//1   0//1    0//1  0//1
       5//7   1//1    0//1  0//1
       3//7   5//6    1//1  0//1
       2//7  -5//12  -1//2  1//1
      U factor:
      4×4 Matrix{Rational{Int64}}:
       7//1   6//1   8//1    5//1
       0//1  12//7   9//7   31//7
       0//1   0//1  -5//2  -23//6
       0//1   0//1   0//1    5//2
```

Like in matlab, the backslash opertor "solves", in this case it solves the LU factorization:

```
[182//75, -7//75, -154//75, 3//5]
```

```
lu1\b1
```

Of course we can apply `\` directly to a matrix. However, behind this always LU decomposition and LU solve are called:

```
x3 =  [182//75, -7//75, -154//75, 3//5]
```

```
x3=A1\b1
```

LU vs. inv

In principle we could work with the inverse matrix as well:

```
A1inv = 4×4 Matrix{Rational{Int64}}:
          68//75  -2//3   2//25   49//75
         -43//75   1//3  -2//25    1//75
         -46//75   1//3   6//25  -53//75
           2//5    0//1  -1//5     1//5
```

```
A1inv=inv(A1)
```

```
[182//75, -7//75, -154//75, 3//5]
```

```
A1inv*b1
```

However, inversion is more complex than the LU factorization.

Some performance tests.

```
rand_Ab (generic function with 1 method)
```

```
rand_Ab(n)=(100.0I(n)+rand(n,n),rand(n))
```

```
function perftest_lu(n)
    A,b=rand_Ab(n)
    @elapsed A\b
end;
```

```
perftest_inv (generic function with 1 method)
```

```
function perftest_inv(n)
    A,b=rand_Ab(n)
    @elapsed inv(A)*b
end
```

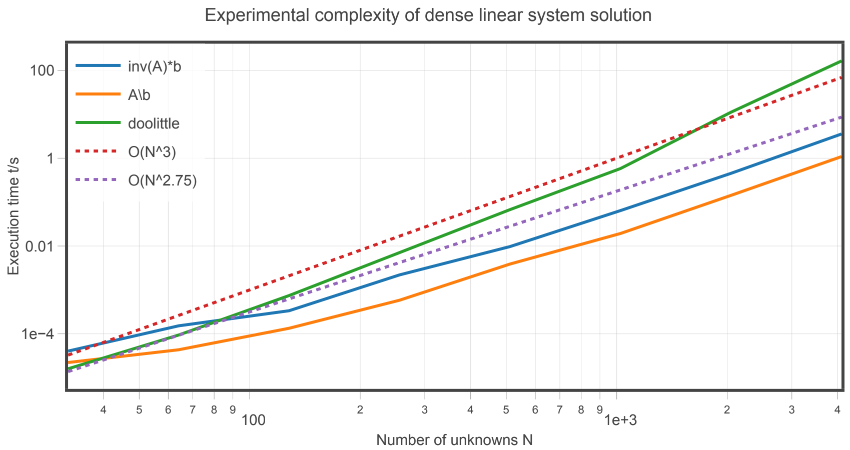
```
perftest_doolittle (generic function with 1 method)
```

```
function perftest_doolittle(n)
    A,b=rand_Ab(n)
    @elapsed doolittle_solve(A,b)
end
```

```
test_and_plot (generic function with 1 method)
```

```
function test_and_plot(powmax)
    N= 2 .^collect(5:powmax)
    t_inv=perftest_inv.(N)
    t_lu=perftest_lu.(N)
    t_doo=perftest_doolittle.(N)

    p=plot(dim=1,resolution=(600,300),
           xscale=:log, yscale=:log,
           title="Experimental complexity of dense linear system solution",
           xlabel="Number of unknowns N",
           ylabel="Execution time t/s",
           legend=:lt)
    plot!(p,N,t_inv,label="inv(A)*b")
    plot!(p,N,t_lu,label="A\\b")
    plot!(p,N,t_doo,label="doolittle")
    plot!(p,N,1.0e-9*N.^3,label="O(N^3)",linestyle=:dot)
    plot!(p,N,1.0e-9*N.^2.75,label="O(N^2.75)",linestyle=:dot)
    p
end
```



```
test_and_plot(12)
```

- The complexity for the built-in factorization is around $O(N^{2.75})$ which is in the region of some theoretical estimates.
- For standard floating point types, Julia uses highly optimized versions of **LINPACK** and **BLAS**
  - Same for python/numpy and many other coding environments
- A good implementation is hard to get right, straightforward code performs worse than the system implementation
- Using inversion instead of factorization is significantly slower (log scale in the plot!)