Direct solution of linear systems of equations

Advanced Topics from Scientific Computing
TU Berlin Winter 2024/25
Notebook 07
 Jürgen Fuhrmann

```
begin
    using LinearAlgebra
    using BenchmarkTools
    BenchmarkTools.DEFAULT_PARAMETERS.seconds=1
    using CairoMakie
    using LaTeXStrings
    using Colors
    CairoMakie.activate!(type="svg")
end
```

LU Factorization for dense matrices

Let aus create a matrix and solve the corresponding linear system:

```
n1 = 1000
```

```
n1=1000
```

```
A1 = 1000×1000 Matrix{Float64}:
 0.0187078  0.188446   0.056845   0.328211   …  0.871847  0.996718   0.483872
 0.238439   0.163153   0.0569715  0.171423      0.251316  0.62393    0.443677
 0.418107   0.208094   0.424371   0.702195      0.014016  0.603652   0.758104
 0.706897   0.526035   0.882269   0.51316       0.355237  0.276658   0.79914
 0.79588    0.272369   0.124635   0.556243      0.212103  0.380853   0.327854
 0.218654   0.38597    0.910232   0.464832   …  0.518781  0.844682   0.0495958
 0.607581   0.880196   0.580824   0.390077      0.552918  0.0958148  0.313097
 ⋮                                          ⋱
 0.329904   0.129021   0.839118   0.534181      0.592694  0.574137   0.508975
 0.809565   0.541011   0.0791187  0.936945   …  0.533528  0.163605   0.0329636
 0.651893   0.770524   0.950013   0.25946       0.368895  0.494187   0.00233496
 0.900179   0.425672   0.227911   0.138358      0.119497  0.309069   0.674051
 0.280946   0.634689   0.820704   0.0385803     0.862712  0.61552    0.105805
 0.536608   0.248844   0.457615   0.7537        0.308882  0.177129   0.390003
```

```
A1=rand(n1,n1)
```

```
x1 =
▸ [0.984984, 0.651061, 0.962979, 0.800093, 0.51045, 0.1374, 0.731817, 0.364274, 0.316247, (
```

```
x1=rand(n1)
```

```
b1 =
▸[253.482, 261.27, 254.493, 250.716, 252.925, 251.646, 255.41, 246.143, 258.692, 255.815,
```

```
1 b1=A1*x1
```

```
8.766598558196392e-13
```

```
1 norm(A1\b1-x1, Inf)
```

The "\" operator provides a default solver for linear systems of equations based on the **LU factorization** of the matrix into an upper and a lower triangular matrix, and the subsequent solution of the triangular systems:

$$A = LU$$
$$Ly = b$$
$$Ux = y$$

This approach is equivalent to the Gaussian elimination process. The algorithm is improved by stability enhancing **pivoting** - reordering of the system of equations such that divisions by small main diagonal elements is avoided.

We can demonstrate this process:

```
lu1 = LU{Float64, Matrix{Float64}, Vector{Int64}}
     L factor:
     1000×1000 Matrix{Float64}:
      1.0          0.0         0.0          …   0.0        0.0        0.0       0.0
      0.00383949   1.0         0.0              0.0        0.0        0.0       0.0
      0.112664    -0.067023    1.0              0.0        0.0        0.0       0.0
      0.991586    -0.612582    0.367514         0.0        0.0        0.0       0.0
      0.693816     0.407883   -0.451443         0.0        0.0        0.0       0.0
      0.0851052    0.596835   -0.395128    …    0.0        0.0        0.0       0.0
      0.0376346    0.138246   -0.00152919       0.0        0.0        0.0       0.0
      ⋮                                    ⋱
      0.568917     0.294731   -0.157776         0.0        0.0        0.0       0.0
      0.75586      0.464616   -0.214669    …    0.0        0.0        0.0       0.0
      0.554549     0.528174   -0.0315308       1.0        0.0        0.0       0.0
      0.838586    -0.0716095  -0.122622        0.30364    1.0        0.0       0.0
      0.820054    -0.0947504   0.466606       -0.142745   0.654561   1.0       0.0
      0.961111    -0.346829    0.776097       -0.260066  -0.911346  -0.553184  1.0
     U factor:
     1000×1000 Matrix{Float64}:
      0.999977  0.620705  0.436929  0.159444  …   0.547293     0.359917    0.194705
      0.0       0.992278  0.589948  0.379974      0.60991      0.0620284   0.688592
      0.0       0.0       0.960552  0.221624      0.121772     0.914632    0.188875
      0.0       0.0       0.0       0.991237      0.255211    -0.164363    0.638524
      0.0       0.0       0.0       0.0           0.0466011    0.678258   -0.237994
      0.0       0.0       0.0       0.0       …  -0.143928     0.880002   -0.168435
      0.0       0.0       0.0       0.0          -0.206481    -0.290818    0.343534
      ⋮                                      ⋱
      0.0       0.0       0.0       0.0           4.63777     -3.27656    -1.639
      0.0       0.0       0.0       0.0       …   2.35711      4.01344     2.84879
      0.0       0.0       0.0       0.0           0.653449    -8.1761     -2.03969
      0.0       0.0       0.0       0.0          -4.51071      5.71619     0.712088
      0.0       0.0       0.0       0.0           0.0        -13.7588     -9.74364
      0.0       0.0       0.0       0.0           0.0          0.0        -1.58958
```

```
1 lu1=lu(A1)
```

Extract the pivoting permutation:

```
p =
▸[914, 885, 522, 266, 589, 335, 347, 98, 624, 764, 584, 576, 513, 124, 797, 69, 53, 890, 375,
```

```
1 p=lu1.p
```

Permute the right hand side vector:

```
b1_permuted =
▸[244.273, 263.136, 242.998, 257.758, 250.099, 251.636, 256.213, 248.482, 254.959, 244.58
```

```
1 b1_permuted=b1[p]
```

Solve the triangular systems with L and U

```
y =
▸[244.273, 262.198, 233.05, 90.5087, 75.9184, 126.988, 64.3686, -111.019, 144.512, -78.17
```

```
y=lu1.L\b1_permuted
```

```
x1_lu =
▸[0.984984, 0.651061, 0.962979, 0.800093, 0.51045, 0.1374, 0.731817, 0.364274, 0.316247, (
```

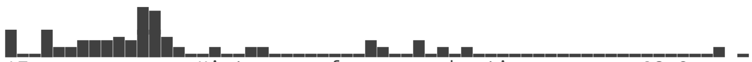
```
x1_lu=lu1.U\y
```

```
8.766598558196392e-13
```

```
norm(x1-x1_lu, Inf)
```

These steps are combined in the "\" operator for LU factorizations

```
BenchmarkTools.Trial: 56 samples with 1 evaluation.
 Range (min … max):  17.002 ms …  22.097 ms  ┊ GC (min … max): 0.00% … 18.49%
 Time  (median):     17.752 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   17.960 ms ± 903.892 μs  ┊ GC (mean ± σ):  1.64% ±  3.73%
```

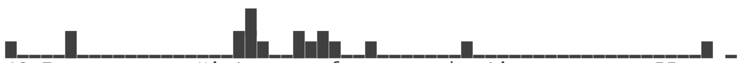


```
 17 ms           Histogram: frequency by time          20.8 ms <

 Memory estimate: 7.64 MiB, allocs estimate: 9.
```

```
@benchmark A1\b1
```

```
BenchmarkTools.Trial: 20 samples with 1 evaluation.
 Range (min … max):  49.329 ms … 54.987 ms  ┊ GC (min … max): 0.00% … 1.98%
 Time  (median):     51.366 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   51.517 ms ±  1.173 ms  ┊ GC (mean ± σ):  0.11% ± 0.44%
```



```
 49.3 ms         Histogram: frequency by time           55 ms <

 Memory estimate: 8.13 MiB, allocs estimate: 13.
```

```
@benchmark inv(A1)*b1
```

```
BenchmarkTools.Trial: 79 samples with 1 evaluation.
 Range (min … max):  12.022 ms …  17.274 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     12.322 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   12.759 ms ± 979.936 μs  ┊ GC (mean ± σ):  1.44% ± 2.86%
```

```
 12 ms           Histogram: frequency by time          17.1 ms <

 Memory estimate: 8.13 MiB, allocs estimate: 15.
```

```
@benchmark Symmetric(A1)\b1
```

- LU factorization takes the most time in this approach, triangular solves are fast
- LU factorization + triangular solve is significantly faster than matrix inversion
- Symmetry and positive definiteness of a matrix can be utilized to speed up the method.
- For standard floating point types, Julia uses highly optimized versions of LAPACK and BLAS
    - Same for python/numpy, matlab and many other coding environments

LU Factorization for sparse matrices

As we focus in this course on partial differential equations, we need discuss matrices which evolve from the discretization of PDEs.

- Are there any structural or numerical patterns in these matrices we can take advantage of with regard to memory and time complexity when solving linear systems ?

In this lecture we introduce a relatively simple "drosophila" problem which we will use do discuss these issues.

For the start we use simple structured discretization grids and a finite difference approach to the discretization. Later, this will be generalized to more general grids and to finite element and finite volume discretization methods.

1D PDE problems

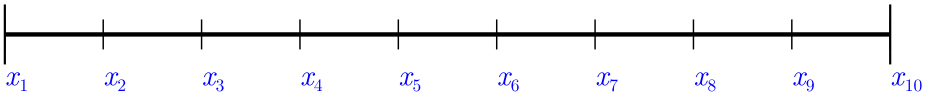
Assume a one-dimensional rod of length 1

- Heat source $f(x)$
- v_L, v_R: ambient temperatures
- α: boundary heat transfer coefficient
- Second order boundary value problem in $\Omega = [0,1]$:

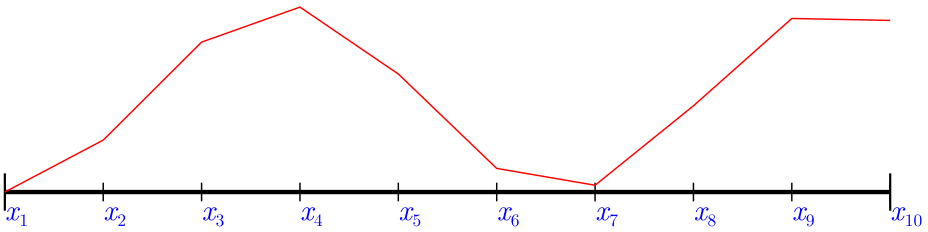
$$\begin{cases} -u''(x) & = f(x) \text{ in } \Omega \\ -u'(0) + \alpha(u(0) - v_L) & = 0 \\ u'(1) + \alpha(u(1) - v_R) & = 0 \end{cases}$$

- The solution u describes the equilibrium temperature distribution. Behind the second derivative is Fouriers law and the continuity equation
- In math, the boundary conditions are called "Robin" or "third kind". They describe a heat in/outflux proportional to the difference between rod end temperature and ambient temperature
- Fix a number of discretization points N
- Let $h = \frac{1}{N-1}$
- Let $x_i = (i-1)h \;\; i = 1 \dots N$ be discretization points

```
N0=10;
```



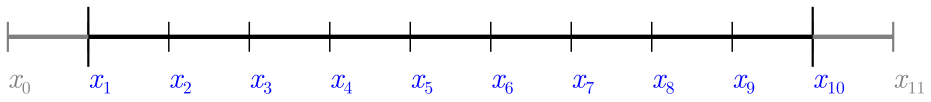
We can approximate continuous functions f by piecewise linear functions defined by the values $f_i = f(x_i)$. Using more points yields a better approximation:




```
plotgrid(N0,func=x->0.5*sin(5*x)^2,height=200)
```

- Let u_i approximations for $u(x_i)$ and $f_i = f(x_i)$
- Use a finite difference approximation to approximate $u'(x_{i+\frac{1}{2}}) \approx \frac{u_{i+1} - u_i}{h}$
- Same approach for second derivative: $u''(x_i) \approx \frac{u'(x_{i+\frac{1}{2}}) - u'(x_{i-\frac{1}{2}})}{h}$
- Finite difference approximation of the PDE:

$$
\begin{aligned}
-u'(0) + \alpha(u(0) - v_L) &\approx \frac{1}{2h}(u_0 - u_2) + \alpha(u_1 - v_L) = 0 \\
-u''(x_i) - f(x_i) &\approx \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} - f_i = 0 \qquad (i = 1 \dots N)\ (*) \\
u'(1) + \alpha(u(1) - v_R) &\approx \frac{1}{2h}(u_{N+1} - u_{N-1}) + \alpha(u_N - v_R) = 0
\end{aligned}
$$



- Here, we introduced "mirror values" u_0 and u_{N+1} in order to approximate the boundary conditions accurately, such that the finite difference formulas used to approximate $u'(0)$ or $u'(1)$ are centered around these values.
- After rearranging, these values can be expressed via the boundary conditions:

$$
\begin{aligned}
u_0 &= u_2 - 2h\alpha(u_1 - v_L) \\
u_{N+1} &= u_{N-1} - 2h\alpha(u_N - v_R)
\end{aligned}
$$

- Finally, they can be replaced in $(*)$:

$$
\begin{aligned}
0 &= \frac{-u_2 + 2u_1 - u_0}{h} - hf_1 \\
&= \frac{-u_2 + 2u_1 - u_2 + 2h\alpha(u_1 - v_L)}{h} - hf_1 \\
&= \frac{2u_1 - 2u_2 + 2h\alpha(u_1 - v_L)}{h} - hf_1 \\
0 &= \frac{u_1 - u_2 + h\alpha(u_1 - v_L)}{h} - 2hf_1 \\
&= \left(\frac{1}{h} + \alpha\right)u_1 - \frac{1}{h}u_2 - 2hf_1 - \alpha v_L
\end{aligned}
$$

Similarly,

$$
\left(\frac{1}{h} + \alpha\right)u_N - \frac{1}{h}u_{N-1} = 2hf_N - \alpha v_R
$$

For $i = 2 \dots N-1$, rewrite $(*)$ as

$$
-\frac{1}{h}u_{i+1} + \frac{2}{h}u_i - \frac{1}{h}u_{i-1} = hf_i
$$

and

The resulting discretization matrix is

$$
A = \begin{pmatrix}
\alpha + \frac{1}{h} & -\frac{1}{h} & & & & \\
-\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\
 & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\
 & \ddots & \ddots & \ddots & \ddots & \\
 & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\
 & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} \\
 & & & & -\frac{1}{h} & \frac{1}{h} + \alpha
\end{pmatrix}
$$

Outside of the three diagonals, the entries are zero.

The right hand side is:

$$f = \begin{pmatrix} \frac{h}{2} f_1 + \alpha v_L \\ h f_2 \\ h f_3 \\ \vdots \\ h f_{N-2} \\ h f_{N-1} \\ \frac{h}{2} f_N + \alpha v_R \end{pmatrix}$$

We want to solve $Au = f$

Let us define functions assembling these:

heatmatrix1d! (generic function with 1 method)

```
function heatmatrix1d!(A,N;α=1)
    h=1/(N-1)
    A[1,1]=1/h+α
    for i=2:N-1
        A[i,i]=2/h
    end
    for i=1:N-1
        A[i,i+1]=-1/h
    end
    for i=2:N
        A[i,i-1]=-1/h
    end
    A[N,N]=1/h+α
    A
end

```

heatrhs1d (generic function with 1 method)

```
function heatrhs1d(N;vl=0,vr=0,func=x->0,α=1)
    h=1/(N-1)
    F=zeros(N)
    F[1]=h/2*func(0)+α*vl
    for i=2:N-1
        F[i]=h*func((i-1)*h)
    end
    F[N]=h/2*func(1)+α*vr
    F
end

```

α = 100

```
α=100
```

N1 = 100

```
N1=100
```

A1_heat =

```
100×100 Matrix{Float64}:
 199.0  -99.0    0.0    0.0    0.0    0.0  …    0.0    0.0    0.0    0.0    0.0
 -99.0  198.0  -99.0    0.0    0.0    0.0       0.0    0.0    0.0    0.0    0.0
   0.0  -99.0  198.0  -99.0    0.0    0.0       0.0    0.0    0.0    0.0    0.0
   0.0    0.0  -99.0  198.0  -99.0    0.0       0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0  -99.0  198.0  -99.0       0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0  -99.0  198.0  …    0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0  -99.0       0.0    0.0    0.0    0.0    0.0
   ⋮                                   ⋮    ⋱    ⋮
   0.0    0.0    0.0    0.0    0.0    0.0    -99.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0    0.0  …  198.0  -99.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0    0.0    -99.0  198.0  -99.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0    0.0      0.0  -99.0  198.0  -99.0    0.0
   0.0    0.0    0.0    0.0    0.0    0.0      0.0    0.0  -99.0  198.0  -99.0
   0.0    0.0    0.0    0.0    0.0    0.0      0.0    0.0    0.0  -99.0  199.0
```

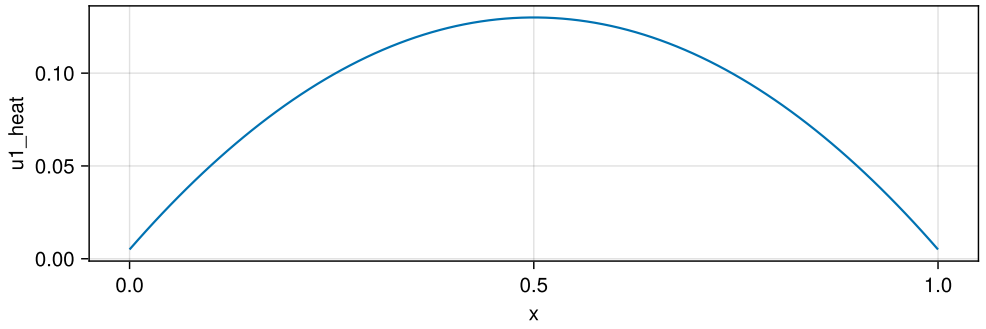
```
A1_heat=heatmatrix1d!(zeros(N1,N1),N1,α=α)
```

```
b1_heat =
▸[0.00505051, 0.010101, 0.010101, 0.010101, 0.010101, 0.010101, 0.010101, 0.010101, 0.010
```

```
1 b1_heat=heatrhs1d(N1,func=x->1,α=α)
```

```
u1_heat =
▸[0.005, 0.00999949, 0.0148969, 0.0196924, 0.0243858, 0.0289771, 0.0334665, 0.0378538, 0.
```

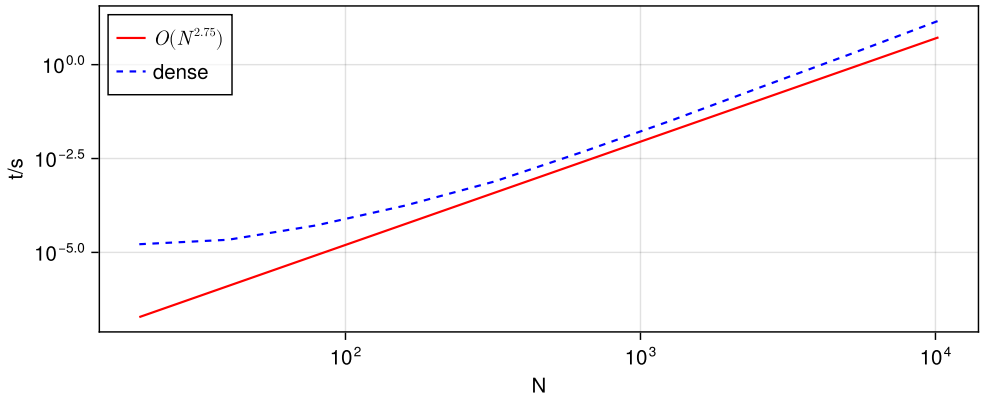
```
1 u1_heat=A1_heat\b1_heat
```




```
allN = ▸[20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240]
```

```
times_full =
▸[1.651e-5, 2.1687e-5, 5.2795e-5, 0.00017796, 0.000774045, 0.00481406, 0.0331157, 0.25205
```

```
1  times_full= let
2      times=zeros(0)
3    for N∈allN
4        A=heatmatrix1d!(zeros(N,N),N,α=α)
5        b=heatrhs1d(N,α=α)
6        t=@elapsed u=A\b
7        push!(times,t)
8    end
9      times
10 end
```



The complexity of the LU factorization of a dense matrix is estimated with $O(N^{2.75})$

However, we can store only three diagonals of the matrix and apply methods tailored to this case:

Try tridiagonal ☐

Tridiagonal matrix storage

```
if try_tridiagonal
   times_tridiagonal= let
    times=zeros(0)
  for N∈allN
       A=Tridiagonal(zeros(N-1),zeros(N),zeros(N-1))
       A=heatmatrix1d!(A,N,α=α)
       b=heatrhs1d(N,α=α)
       t=@elapsed u=A\b
       push!(times,t)
  end
    times
  end
end
```

We learn, that in this case, solution time is O(N), much better.

One caveat: Never calculate the inverse of a matrix from a PDE, as it neither tridiagonal nor sparse anymore!

```
10×10 Matrix{Float64}:
 0.00990196  0.00881264  0.00772331  …  0.00227669  0.00118736  9.80392e-5
 0.00881264  0.106731    0.0935379      0.0275732   0.0143803   0.00118736
 0.00772331  0.0935379   0.179352       0.0528698   0.0275732   0.00227669
 0.00663399  0.080345    0.154056       0.0781663   0.0407662   0.00336601
 0.00554466  0.067152    0.128759       0.103463    0.0539591   0.00445534
 0.00445534  0.0539591   0.103463    …  0.128759    0.067152    0.00554466
 0.00336601  0.0407662   0.0781663      0.154056    0.080345    0.00663399
 0.00227669  0.0275732   0.0528698      0.179352    0.0935379   0.00772331
 0.00118736  0.0143803   0.0275732      0.0935379   0.106731    0.00881264
 9.80392e-5  0.00118736  0.00227669     0.00772331  0.00881264  0.00990196
```

```
let
    A=Tridiagonal(zeros(N0-1),zeros(N0),zeros(N0-1))
    A=heatmatrix1d!(A,N0,α=α)
    inv(A)
end
```

2D PDE problems

Just pose the heat problem in a 2D domain $\Omega = (0,1) \times (0,1)$:

$$\begin{cases} -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} & = f(x,y) \text{ in } \Omega \\ \frac{\partial u}{\partial n} + \alpha(u - v) & = 0 \text{ on } \partial\Omega \end{cases}$$

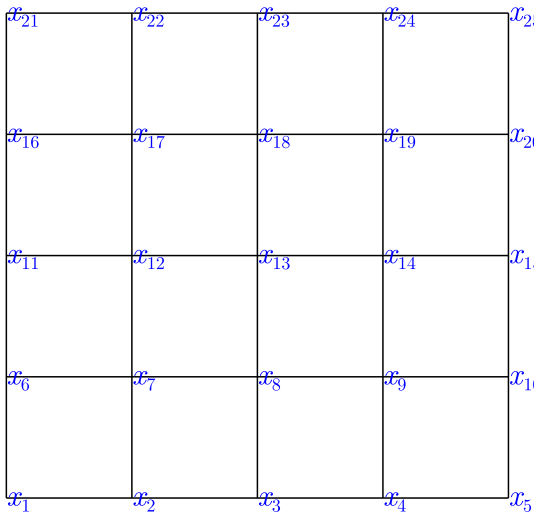
We use 2D regular discretization $n \times n$ grid with grid points $x_{ij} = ((i-1)h, (j-1)h)$. The finite difference approximation yields:

$$\frac{-u_{i-1,j} - u_{i,j-1} + 4u_{ij} - u_{i+1,j} - u_{i,j+1}}{h^2} = f_{ij}$$

This just comes from summing up the 1D finite difference formula for the x and y directions.

We do not discuss the boundary conditions here.

The $n \times n$ grid leads to an $n^2 \times n^2$ matrix!



Matrix and right hand side assembly inspired by the finite volume method which will be covered later in the course. The result is the same as for the finite difference method with the mirror trick for the boundary condition.

```
heatmatrix2d! (generic function with 1 method)
```

```
function heatmatrix2d!(A,n;α=1)
    function update_pair(A,v,i,j)
        A[i,j]+=-v
        A[j,i]+=-v
        A[i,i]+=v
        A[j,j]+=v
    end
    N=n^2
    h=1.0/(n-1)
    l=1
        for j=1:n
            for i=1:n
                if i<n
                    update_pair(A,1.0,l,l+1)
                end
                if i==1|| i==n
                    A[l,l]+=α
                end
                if j<n
                    update_pair(A,1,l,l+n)
                end
                if j==1|| j==n
                    A[l,l]+=α
                end
                l=l+1
            end
        end
    A
end
```

```
heatrhs2d (generic function with 1 method)
```

```
function heatrhs2d(n; rhs=(x,y)->0,bc=(x,y)->0,α=1.0)
    h=1.0/(n-1)
    x=collect(0:h:1)
    y=collect(0:h:1)
    N=n^2
    f=zeros(N)
    for i=1:n-1
        for j=1:n-1
            ij=(j-1)*n+i
            f[ij]+=h^2/4*rhs(x[i],y[j])
            f[ij+1]+=h^2/4*rhs(x[i+1],y[j])
            f[ij+n]+=h^2/4*rhs(x[i],y[j+1])
            f[ij+n+1]+=h^2/4*rhs(x[i+1],y[j+1])
        end
    end

    for i=1:n
        ij=i
        fac=h
        if i==1 || i==n
            fac=h/2
        end
        f[ij]+=fac*α*bc(x[i],0)
        ij=i+(n-1)*n
        f[ij]+=fac*α*bc(x[i],1)
    end
    for j=1:n
        fac=h
        if j==1 || j==n
            fac=h/2
        end
        ij=1+(j-1)*n
        f[ij]+=fac*α*bc(0,y[j])
        ij=n+(j-1)*n
        f[ij]+=fac*α*bc(1,y[j])
    end
    f
end
```

```
n = 20
```

```
n=20
```

```
b2 =
```

```
▸ [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
```

```
b2=heatrhs2d(n,rhs=(x,y)->sin(3*π*x)*sin(3*π*y),α=α)
```

```
A2 = 400×400 Matrix{Float64}:
 202.0   -1.0    0.0    0.0    0.0    0.0  …    0.0    0.0    0.0    0.0    0.0
  -1.0  103.0   -1.0    0.0    0.0    0.0       0.0    0.0    0.0    0.0    0.0
   0.0   -1.0  103.0   -1.0    0.0    0.0       0.0    0.0    0.0    0.0    0.0
   0.0    0.0   -1.0  103.0   -1.0    0.0       0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0   -1.0  103.0   -1.0       0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0   -1.0  103.0  …    0.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0   -1.0       0.0    0.0    0.0    0.0    0.0
   ⋮                                  ⋮    ⋱    ⋮
   0.0    0.0    0.0    0.0    0.0    0.0      -1.0    0.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0    0.0  …  103.0   -1.0    0.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0    0.0      -1.0  103.0   -1.0    0.0    0.0
   0.0    0.0    0.0    0.0    0.0    0.0       0.0   -1.0  103.0   -1.0    0.0
   0.0    0.0    0.0    0.0    0.0    0.0       0.0    0.0   -1.0  103.0   -1.0
   0.0    0.0    0.0    0.0    0.0    0.0       0.0    0.0    0.0   -1.0  202.0
```

```
A2=heatmatrix2d!(zeros(n^2,n^2),n,α=α)
```

```
using Tables
```

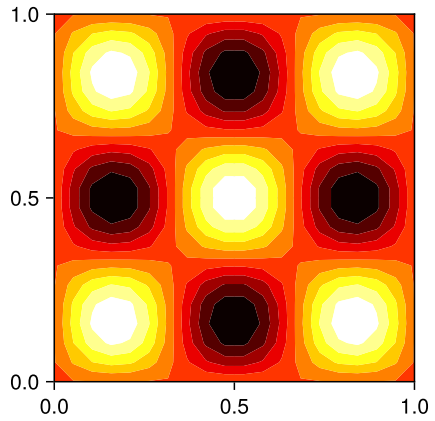
	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
1	202.0	-1.0	0.0	0.0	0.0	0.0	0.0	0.0
2	-1.0	103.0	-1.0	0.0	0.0	0.0	0.0	0.0
3	0.0	-1.0	103.0	-1.0	0.0	0.0	0.0	0.0
4	0.0	0.0	-1.0	103.0	-1.0	0.0	0.0	0.0
5	0.0	0.0	0.0	-1.0	103.0	-1.0	0.0	0.0
6	0.0	0.0	0.0	0.0	-1.0	103.0	-1.0	0.0
7	0.0	0.0	0.0	0.0	0.0	-1.0	103.0	-1.0
8	0.0	0.0	0.0	0.0	0.0	0.0	-1.0	103.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-1.0
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

⋮ more

u2 =

▸[1.28855e-7, 1.30143e-5, 2.28006e-5, 2.71143e-5, 2.49065e-5, 1.6704e-5, 4.48045e-6, -8.8

```
1 u2=A2\b2
```




```
1  let
2      fig=Figure(size=(300,300))
3      axis=Axis(fig[1,1])
4      h=1.0/(n-1)
5      x=collect(0:h:1)
6      y=collect(0:h:1)
7
8      contourf!(axis,x,y,reshape(u2,n,n),colormap="hot")
9      fig
10 end
11
```

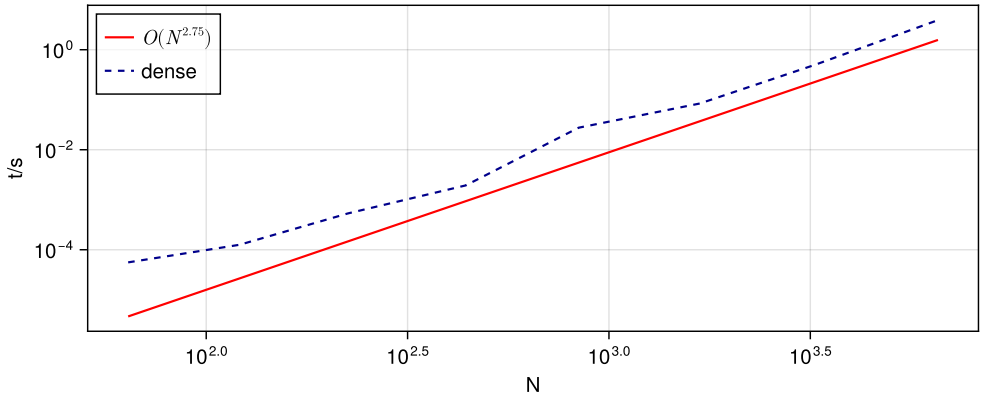
n2d = ▸[8, 11, 15, 21, 29, 41, 57, 81]

```
1 n2d=Int[ceil(5*sqrt(2)^i) for i=1:8]
```

times_full2d =

▸[5.5891e-5, 0.000125675, 0.000533385, 0.00193508, 0.027586, 0.0840268, 0.499638, 3.9104]

```
1  times_full2d= let
2      times=zeros(0)
3    for n∈n2d
4        A=heatmatrix2d!(zeros(n^2,n^2),n,α=α)
5        b=heatrhs2d(n,α=α)
6        t=@elapsed u=A\b
7        push!(times,t)
8    end
9      times
10 end
```



The matrix of this system has 5 nonzero diagonals. While it is possible to store just these five diagonals, there is not much software around which handles this case. But we can use the possibility to store it as a **sparse** matrix.

Try sparse ☐

Sparse matrix storage

```
using SparseArrays
```

```
if try_sparse
times_sparse2d= let
    times=zeros(0)
  for n∈n2d
       A0=spzeros(n^2,n^2)
       A=heatmatrix2d!(A0,n,α=α)
       b=heatrhs2d(n,α=α)
       t=@elapsed u=A\b
       push!(times,t)
  end
    times
end
end
```

In Julia, sparse matrices are stored in Compressed Column Storage (CSC) format.

```
A = 5×5 Matrix{Float64}:
 1.0  0.0  5.0   8.0   0.0
 2.0  4.0  0.0   9.0   0.0
 3.0  0.0  6.0   0.0  11.0
 0.0  0.0  7.0  10.0   0.0
 0.0  0.0  0.0   0.0  12.0
```

```
A=Float64[1 0  5  8 0;
          2 4  0  9  0;
          3 0  6  0  11;
          0 0  7 10 0;
          0 0  0  0  12]
```

```
As = 5×5 SparseMatrixCSC{Float64, Int64} with 12 stored entries:
 1.0   ⋅   5.0   8.0    ⋅
 2.0  4.0   ⋅    9.0    ⋅
 3.0   ⋅   6.0    ⋅    11.0
  ⋅    ⋅   7.0  10.0    ⋅
  ⋅    ⋅    ⋅     ⋅    12.0
```

```
As=sparse(A)
```

The matrix is stored using three vectors:

`nzval` contains the values of the nonzero entries, stored contiguously column by column:

```
▸ [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0]
```

```
As.nzval
```

`rowval` contains the row indices of the nonzero entries

▸ [1, 2, 3, 2, 1, 3, 4, 1, 2, 4, 3, 5]

```
1 As.rowval
```

`colptr` points to the start of a column in `rowval` and `nzval`

▸ [1, 4, 5, 8, 11, 13]

```
1 As.colptr
```

So we can access the nonzeros and the rowvals column by column:

▸ [[1.0, 2.0, 3.0], [4.0], [5.0, 6.0, 7.0], [8.0, 9.0, 10.0], [11.0, 12.0]]

```
1 [As.nzval[As.colptr[i]:As.colptr[i+1]-1] for i=1:size(A,1)]
```

▸ [[1, 2, 3], [2], [1, 3, 4], [1, 2, 4], [3, 5]]

```
1 [As.rowval[As.colptr[i]:As.colptr[i+1]-1] for i=1:size(A,1)]
```

Sparse direct solvers allow to handle LU factorizations for sparse matrices. For 2D problems, the complexity of the LU factorization is $O(N^{1.5})$

Np = 100000

```
1 Np=100_000
```

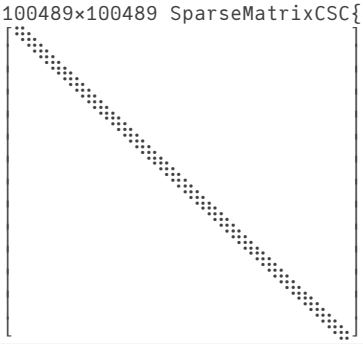
dim = 2

```
1 dim=2
```

np = 317

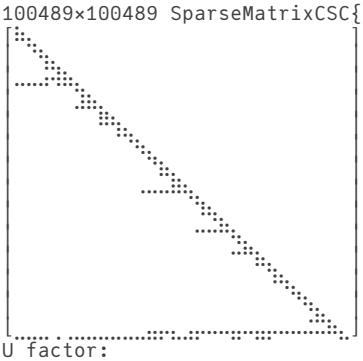
```
1 np=Int(ceil(Np^(1/dim)))
```

```
100489×100489 SparseMatrixCSC{Float64, Int64} with 501177 stored entries:
```



```
1 if dim==1
2     Ap=heatmatrix1d!(spzeros(Np,Np),np)
3 elseif dim==2
4     Ap=heatmatrix2d!(spzeros(np^2,np^2),np)
5 end
```

```
LUp = SparseArrays.UMFPACK.UmfpackLU{Float64, Int64}
      L factor:
      100489×100489 SparseMatrixCSC{Float64, Int64} with 3089774 stored entries:
```



```
      U factor:
      100489×100489 SparseMatrixCSC{Float64, Int64} with 3089774 stored entries:
```

```
1 LUp=lu(Ap)
```

```
12.330071012835784
```

```
1 nnz(LUp)/nnz(Ap)
```

We observe that for space dimension>1 the number of nonzero entries of a sparse LU factorization is significantly larger than the number of nonzeros of the original matrix and depends on the ordering of the unknowns. This phenomenon is called **fill-in**.

Solution steps with sparse direct solvers

1. Pre-ordering
   - Decrease amount of non-zero elements generated by fill-in by re-ordering of the matrix
   - Several, graph theory based heuristic algorithms exist
   - Julia uses resonable defaults with UMFPACK
2. Symbolic factorization
   - If pivoting is ignored, the indices of the non-zero elements are calculated and stored
   - Most expensive step wrt. computation time
3. Numerical factorization
   - Calculation of the numerical values of the nonzero entries
   - Moderately expensive, once the symbolic factors are available
4. Upper/lower triangular system solution
   - Fairly quick in comparison to the other steps

- Separation of steps 2 and 3 allows to save computational costs for problems where the sparsity structure remains unchanged, e.g. time dependent problems on fixed computational grids
- With pivoting, steps 2 and 3 have to be performed together, and pivoting can increase fill-in
- Instead of pivoting, *iterative refinement* may be used in order to maintain accuracy of the solution

Complexity estimate for sparse direct solvers

- Complexity estimates depend on storage scheme, reordering etc.
- Sparse matrix - vector multiplication has complexity $O(N)$
- Some estimates can be given from graph theory for discretizations of heat equation with $N = n^d$ unknowns on close to cubic grids in space dimension d
- sparse LU factorization:

d	$work$	$storage$
1	$O(N) \mid O(n)$	$O(N) \mid O(n)$
2	$O(N^{\frac{3}{2}}) \mid O(n^3)$	$O(N \log N) \mid O(n^2 \log n)$
3	$O(N^2) \mid O(n^6)$	$O(N^{\frac{4}{3}}) \mid O(n^4)$

- triangular solve: work dominated by storage complexity

d	$work$
1	$O(N) \mid O(n)$
2	$O(N \log N) \mid O(n^2 \log n)$
3	$O(N^{\frac{4}{3}}) \mid O(n^4)$

Source: J. Poulson, Fast parallel solution of heterogeneous 3D time-harmonic wave equations (PhD thesis, UT Austin, 2012)

One more thing: how to create a sparse matrix

```
N3 = 200
```

```julia
N3=200
```

A sparse matrix `A` in Julia can be updated just by writing into `A[i,j]`, updating the nonzero entries is done automatically. So start with a sparse matrix with no nonzero entries and just write into it…

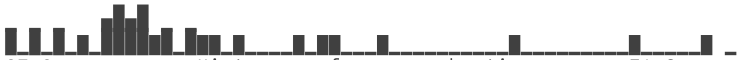
```
BenchmarkTools.Trial: 2 samples with 1 evaluation.
 Range (min … max):  617.911 ms … 623.219 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     620.565 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   620.565 ms ±   3.754 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  618 ms           Histogram: frequency by time          623 ms <

 Memory estimate: 8.29 MiB, allocs estimate: 57.
```

```julia
@benchmark heatmatrix2d!(spzeros(N3^2,N3^2),N3)
```

```
BenchmarkTools.Trial: 36 samples with 1 evaluation.
 Range (min … max):  27.154 ms … 31.868 ms  ┊ GC (min … max): 0.00% … 3.68%
 Time  (median):     28.039 ms              ┊ GC (median):    4.12%
 Time  (mean ± σ):   28.371 ms ±  1.068 ms  ┊ GC (mean ± σ):  2.56% ± 2.07%
```



```
  27.2 ms         Histogram: frequency by time         31.9 ms <

 Memory estimate: 34.30 MiB, allocs estimate: 72.
```

```julia
let
    A=heatmatrix2d!(spzeros(N3^2,N3^2),N3)
    b=rand(N3^2)
    @benchmark $A\$b
end
```

Matrix build-up is much more expensive than solution of the linear system.

… Re-arranging the internal structure is connected to shifting and re-allocating th the arrays many times.

A frequent recommendation ist to use the "coordinate" or "triplet" format as an intermediate: Just collect in three arrays I, J, A all updates of matrix entries and pass them to `sparse`:

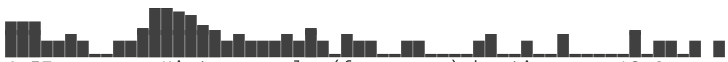
```
heatmatrix2d_coo (generic function with 1 method)
```

```
function heatmatrix2d_coo(n;α=1)
    I=Int[]
    J=Int[]
    A=Float64[]
    function addentry(i,j,v)
        push!(I,i)
        push!(J,j)
        push!(A,v)
    end
    function update_pair(i,j,v)
        addentry(i,j,-v)
        addentry(j,i,-v)
        addentry(i,i,v)
        addentry(j,j,v)
    end
    N=n^2
    h=1.0/(n-1)
    l=1
        for j=1:n
            for i=1:n
                if i<n
                    update_pair(l,l+1,1)
                end
                if i==1|| i==n
                    addentry(l,l,α)
                end
                if j<n
                    update_pair(l,l+n,1)
                end
                if j==1|| j==n
                    addentry(l,l,α)
                end
                l=l+1
            end
        end
    sparse(I,J,A)
end
```

```
true
```

```
heatmatrix2d!(spzeros(N3^2,N3^2),N3)==heatmatrix2d_coo(N3)
```

```
BenchmarkTools.Trial: 162 samples with 1 evaluation.
 Range (min … max):  4.567 ms … 14.068 ms  ┊ GC (min … max):  0.00% … 20.60%
 Time  (median):     5.914 ms              ┊ GC (median):    19.51%
 Time  (mean ± σ):   6.195 ms ±  1.389 ms  ┊ GC (mean ± σ):  14.49% ±  9.56%
```



```
  4.57 ms       Histogram: log(frequency) by time      10.2 ms <

 Memory estimate: 34.10 MiB, allocs estimate: 91.
```

```
@benchmark heatmatrix2d_coo(N3)
```

This approach requires to modify the structure of the assembly loop. If we run through this loop once this is ok. If one wants to update the nonzero entries, one needs to implement this loop twice. Moreover, one loses the intuitive way of writing into a matrix.

The ExtendableSparse.jl packages provides a remedy. It uses a linked list internal representation to build up the matrix and hides it behind the intuitive way of writing into a matrix. A `flush!` method sets up a SparseMatrixCSC structure after assembly.

```
using ExtendableSparse
```

```
BenchmarkTools.Trial: 107 samples with 1 evaluation.
 Range (min … max):  6.877 ms … 19.682 ms  ┊ GC (min … max):  0.00% … 29.93%
 Time  (median):     9.143 ms              ┊ GC (median):     0.00%
 Time  (mean ± σ):   9.368 ms ±  2.433 ms  ┊ GC (mean ± σ):  14.34% ± 15.87%
```

```
 6.88 ms         Histogram: frequency by time        16.1 ms <

 Memory estimate: 25.41 MiB, allocs estimate: 199252.
```

```
@benchmark begin
    Aext=heatmatrix2d!(ExtendableSparseMatrix(N3^2,N3^2),N3)
    flush!(Aext)
end
```

This approach can be made faster by using `updateindex!` instead of `+=`. In Julia issue 15630 it is proposed that the compiler should detect this situation.

```
heatmatrix2d_update! (generic function with 1 method)
```

```
function heatmatrix2d_update!(A,n;α=1)
    function update_pair(A,v,i,j)
        updateindex!(A,+,-v,i,j)
        updateindex!(A,+,-v,j,i)
        updateindex!(A,+,v,i,i)
        updateindex!(A,+,v,j,j)
    end
    N=n^2
    h=1.0/(n-1)
    l=1
        for j=1:n
            for i=1:n
                if i<n
                    update_pair(A,1.0,l,l+1)
                end
                if i==1|| i==n
                    updateindex!(A,+,α,l,l)
                end
                if j<n
                    update_pair(A,1.0,l,l+n)
                end
                if j==1|| j==n
                    updateindex!(A,+,α,l,l)
                end
                l=l+1
            end
        end
    A
end
```

```
BenchmarkTools.Trial: 126 samples with 1 evaluation.
 Range (min … max):  5.478 ms … 20.856 ms  ┊ GC (min … max):  0.00% … 40.12%
 Time  (median):     8.507 ms              ┊ GC (median):     0.00%
 Time  (mean ± σ):   7.977 ms ±  2.373 ms  ┊ GC (mean ± σ):  18.05% ± 17.99%
```

```
 5.48 ms         Histogram: frequency by time        15.5 ms <

 Memory estimate: 25.41 MiB, allocs estimate: 199252.
```

```
@benchmark begin
    Aext=heatmatrix2d_update!(ExtendableSparseMatrix(N3^2,N3^2),N3)
    flush!(Aext)
end
```

Final remarks

- As a rule, direct solution of linear systems of equations is implemented via LU factorization
- Matrices from finite difference methods for PDEs are sparse. True also for finite elements and finite volume methods.

- LU factorizations from sparse matrices suffer from fill-in: LU factors tend to have more nonzeor entries than the original matrices. In 3D significantly so.
- Inverses of matrices from PDEs tend to be full matrices
- The Julia `\` operator by default maps to the UMFPACK sparse direct solver from the Suitesparse collection by T. Davis.
- Other sparse direct solvers (e.g. the thread-parallel Pardiso) are available as Julia packages.

```
begin
    using PlutoUI
    using HypertextLiteral: @htl, @htl_str
end;
```