

Advanced Topics from Scientific Computing

TU Berlin Winter 2023/24

Notebook 07



Jürgen Fuhrmann

```
1 begin
2     using LinearAlgebra
3     using BenchmarkTools
4     BenchmarkTools.DEFAULT_PARAMETERS.seconds=1
5     using CairoMakie
6     using LaTeXStrings
7     using Colors
8     CairoMakie.activate!(type="svg")
9 end
```

Direct solution of linear systems of equations

LU Factorization for dense matrices

LU Factorization for sparse matrices

1D PDE problems

2D PDE problems

Complexity estimate for sparse direct solvers

One more thing: how to create a sparse matrix

Final remarks

Direct solution of linear systems of equations

LU Factorization for dense matrices

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

```
n1 = 1000
```

```
1 n1=1000
```

```
A1 = 1000×1000 Matrix{Float64}:
```

```
 0.753802  0.01799  0.339953  0.737393  ...  0.392214  0.0694217  0.189724
 0.291844  0.106554  0.530366  0.0987676  ...  0.0269888  0.620554  0.431681
 0.938774  0.605128  0.33763  0.103028  ...  0.291447  0.333717  0.178977
 0.32829  0.852711  0.660378  0.254808  ...  0.178027  0.510128  0.145813
 0.990202  0.16697  0.408879  0.888151  ...  0.124851  0.15294  0.820325
 0.33153  0.93014  0.365096  0.722669  ...  0.564907  0.569961  0.381232
 0.592735  0.890489  0.735259  0.722941  ...  0.627676  0.659641  0.897708
  ⋮
 0.526348  0.107154  0.671321  0.158944  ...  0.0898608  0.874968  0.650386
 0.271917  0.988568  0.0920855  0.962191  ...  0.129458  0.542647  0.444973
 0.278775  0.583207  0.0895019  0.669835  ...  0.197964  0.432388  0.066795
 0.0150554  0.175874  0.38333  0.402243  ...  0.120967  0.725175  0.172001
 0.349289  0.810497  0.116466  0.556378  ...  0.545595  0.497446  0.306455
 0.391132  0.333933  0.443134  0.174298  ...  0.375859  0.137281  0.902679
```

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

```
x1 =
```

```
[0.360065, 0.578613, 0.727061, 0.0467842, 0.541898, 0.124776, 0.0511791, 0.935183, 0.2153]
```

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

```
b1 =
```

```
[236.874, 254.933, 240.221, 240.907, 255.777, 247.494, 249.773, 242.91, 252.609, 253.607,
```

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

```
6.97031321550412e-12
```

```
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:

$$\begin{aligned} A &= LU \\ Ly &= b \\ Ux &= y \end{aligned}$$

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
 0.0283386 1.0      0.0      0.0      0.0      0.0      0.0      0.0
 0.00987099 0.25725  1.0      0.0      0.0      0.0      0.0      0.0
 0.916381  -0.269127 -0.514119 0.0      0.0      0.0      0.0      0.0
 0.139038  -0.0333605 0.685978 0.0      0.0      0.0      0.0      0.0
 0.272005  0.919505  -0.127025 ...  0.0      0.0      0.0      0.0
 0.710902  0.733238  0.0441187 0.0      0.0      0.0      0.0      0.0
  ⋮
 0.205081  0.123499  0.656733  ...  0.0      0.0      0.0      0.0
 0.66871  0.514942  0.0302527 ...  0.0      0.0      0.0      0.0
 0.100733  0.0961482 0.264704  1.0      0.0      0.0      0.0      0.0
 0.795891  -0.0610501 -0.153349 -0.44157  1.0      0.0      0.0      0.0
 0.533658  0.102248  -0.103927 0.491829 -0.216504 1.0      0.0      0.0
 0.469023  0.525383  0.105465 0.507627 -0.644981 0.0556798 1.0
```

```
U factor:
```

```
1000×1000 Matrix{Float64}:
```

```
 0.999674  0.32375  0.621629  0.347337 ...  0.481253  0.599062  0.719596
 0.0      0.979339  0.0508501 0.484454 ...  0.917681  0.443676  0.165126
 0.0      0.0      0.974278  0.486101 ...  0.0363039 0.680242  0.626513
 0.0      0.0      0.0      1.05933 ... -0.121089 0.119526 -0.120348
 0.0      0.0      0.0      0.0      ...  0.279084 -0.0435752 0.283245
 0.0      0.0      0.0      0.0      ... -0.841017 0.0122298 0.175448
 0.0      0.0      0.0      0.0      ... -0.303459 -0.617823 0.290945
  ⋮
 0.0      0.0      0.0      0.0      ...  5.1432  -13.921  1.43865
 0.0      0.0      0.0      0.0      ...  3.70442  20.74  1.10267
 0.0      0.0      0.0      0.0      ... -7.87448  13.6371  4.65952
 0.0      0.0      0.0      0.0      ... -4.88745  8.06421 -1.9466
 0.0      0.0      0.0      0.0      ...  0.0     -10.0649  2.00505
 0.0      0.0      0.0      0.0      ...  0.0     -1.5934
```

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

Extract the pivoting permutation:

```
p =
```

```
[632, 162, 830, 202, 422, 996, 359, 543, 595, 161, 33, 623, 23, 776, 854, 626, 178, 394, 8
```

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

Permute the right hand side vector:

```
b1_permuted =
```

```
[250.077, 247.297, 263.256, 249.969, 252.688, 250.756, 259.816, 252.093, 259.357, 250.962
```

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

Solve the triangular systems with L and U

```
y =
```

```
[250.077, 240.21, 198.994, 187.756, 134.896, -125.57, 71.4162, -32.0096, 46.2753, -71.256
```

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

```
x1_lu =
```

```
[0.360065, 0.578613, 0.727061, 0.0467842, 0.541898, 0.124776, 0.0511791, 0.935183, 0.2155
```

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

```
6.97031321550412e-12
```

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

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

BenchmarkTools.Trial: 24 samples with 1 evaluation.

| | | | |
|----------------------|-------------------------|-------------------|-----------------|
| Range (min ... max): | 41.525 ms ... 43.739 ms | GC (min ... max): | 0.00% ... 2.38% |
| Time (median): | 41.862 ms | GC (median): | 0.00% |
| Time (mean ± σ): | 42.233 ms ± 699.312 μs | GC (mean ± σ): | 0.43% ± 0.96% |



Memory estimate: 7.64 MiB, allocs estimate: 4.

```
1 @benchmark A1\b1
```

BenchmarkTools.Trial: 9 samples with 1 evaluation.

| | | | |
|----------------------|---------------------------|-------------------|-----------------|
| Range (min ... max): | 122.229 ms ... 124.424 ms | GC (min ... max): | 0.00% ... 0.89% |
| Time (median): | 123.160 ms | GC (median): | 0.00% |
| Time (mean ± σ): | 123.176 ms ± 759.469 μs | GC (mean ± σ): | 0.10% ± 0.30% |

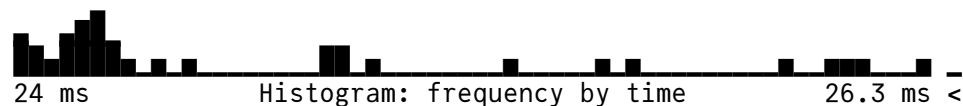


Memory estimate: 8.13 MiB, allocs estimate: 6.

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

BenchmarkTools.Trial: 41 samples with 1 evaluation.

| | | | |
|----------------------|-------------------------|-------------------|-----------------|
| Range (min ... max): | 23.977 ms ... 26.276 ms | GC (min ... max): | 0.00% ... 4.48% |
| Time (median): | 24.191 ms | GC (median): | 0.00% |
| Time (mean ± σ): | 24.553 ms ± 690.611 μs | GC (mean ± σ): | 0.71% ± 1.65% |



Memory estimate: 8.13 MiB, allocs estimate: 7.

```
1 @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 to 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

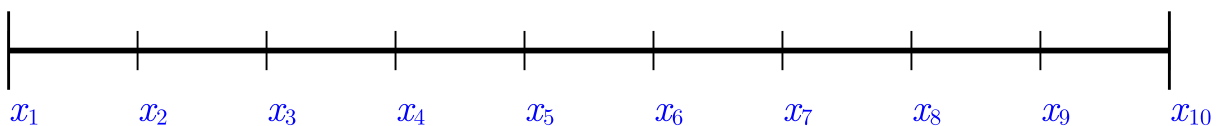
Assume a one-dimensional rod of length 1

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

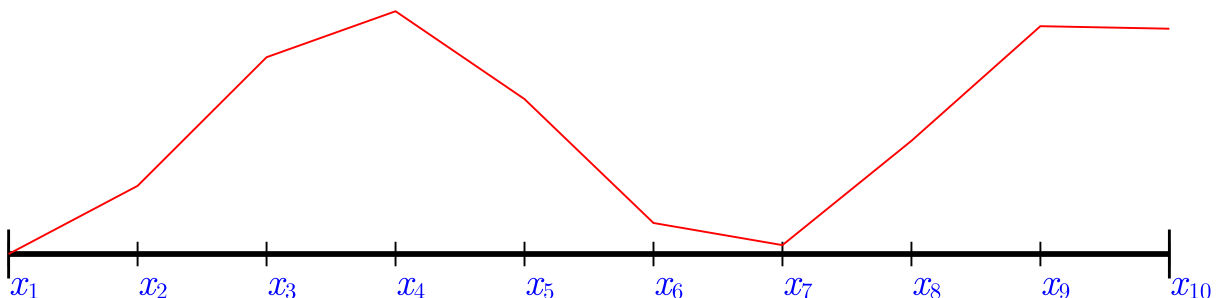
$$\begin{cases} -u''(x) & = f(x) \text{ in } \Omega \\ -u'(0) + \alpha(u(0) - v_L) & = 0 \\ u'(1) + \alpha(u(1) - v_R) & = 0 \end{cases}$$

- The solution u describes the equilibrium temperature distribution. Behind the second derivative is Fourier's 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

```
1 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:



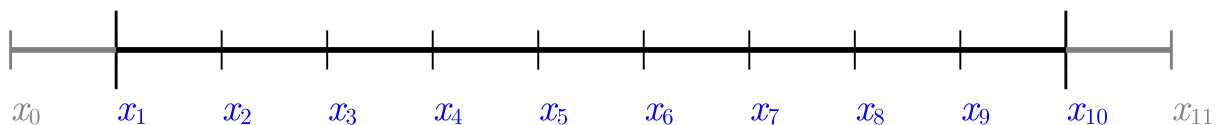
```
1 plotgrid(N0, func=x->0.5*sin(5*x)^2, height=200)
```

- 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:

$$-u'(0) + \alpha(u(0) - v_L) \approx \frac{1}{2h}(u_0 - u_2) + \alpha(u_1 - v_L) = 0$$

$$-u''(x_i) - f(x_i) \approx \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} - f_i = 0 \quad (i = 1 \dots N) (*)$$

$$u'(1) + \alpha(u(1) - v_R) \approx \frac{1}{2h}(u_{N+1} - u_{N-1}) + \alpha(u_N - v_R) = 0$$



- 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:

$$u_0 = u_2 - 2h\alpha(u_1 - v_L)$$

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

- 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:

$$f = \begin{pmatrix} \frac{h}{2} f_1 + \alpha v_L \\ hf_2 \\ hf_3 \\ \vdots \\ hf_{N-2} \\ hf_{N-1} \\ \frac{h}{2} f_N + \alpha v_R \end{pmatrix}$$

We want to solve $Au = f$

Let us define functions assembling these:

heatmatrix1d! (generic function with 1 method)

```

1 function heatmatrix1d!(A,N;α=1)
2     h=1/(N-1)
3     A[1,1]=1/h+α
4     for i=2:N-1
5         A[i,i]=2/h
6     end
7     for i=1:N-1
8         A[i,i+1]=-1/h
9     end
10    for i=2:N
11        A[i,i-1]=-1/h
12    end
13    A[N,N]=1/h+α
14    A
15 end
16
```

heatrhs1d (generic function with 1 method)

```

1 function heatrhs1d(N;vl=0,vr=0,func=x->0,α=1)
2     h=1/(N-1)
3     F=zeros(N)
4     F[1]=h/2*func(0)+α*vl
5     for i=2:N-1
6         F[i]=h*func((i-1)*h)
7     end
8     F[N]=h/2*func(1)+α*vr
9     F
10 end
11
```

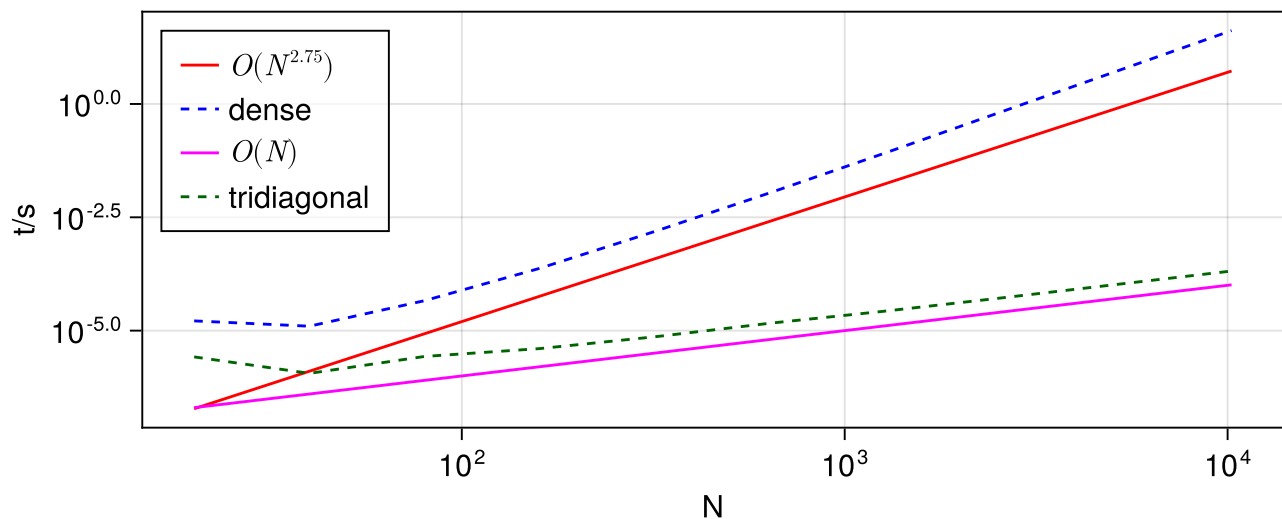
$\alpha = 100$

```
1 α=100
```



```
times_full =
 [1.6401e-5, 1.2519e-5, 4.5872e-5, 0.000237633, 0.00155232, 0.0110792, 0.0838737, 0.66664:
```

```
1 times_full= let
2   times=zeros(0)
3   for N=1:N
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

```
[2.632e-6, 1.141e-6, 2.695e-6, 4.019e-6, 7.331e-6, 1.4605e-5, 2.7036e-5, 5.2549e-5, 0.000
```

```
1 if try_tridiagonal
2   times_tridiagonal= let
3     times=zeros(0)
4     for N=1:N
5       A=Tridiagonal(zeros(N-1),zeros(N),zeros(N-1))
6       A=heatmatrix1d!(A,N,α=α)
7       b=heatrhs1d(N,α=α)
8       t=@elapsed u=A\b
9       push!(times,t)
10    end
11    times
12  end
13 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
```

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

2D PDE problems

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

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

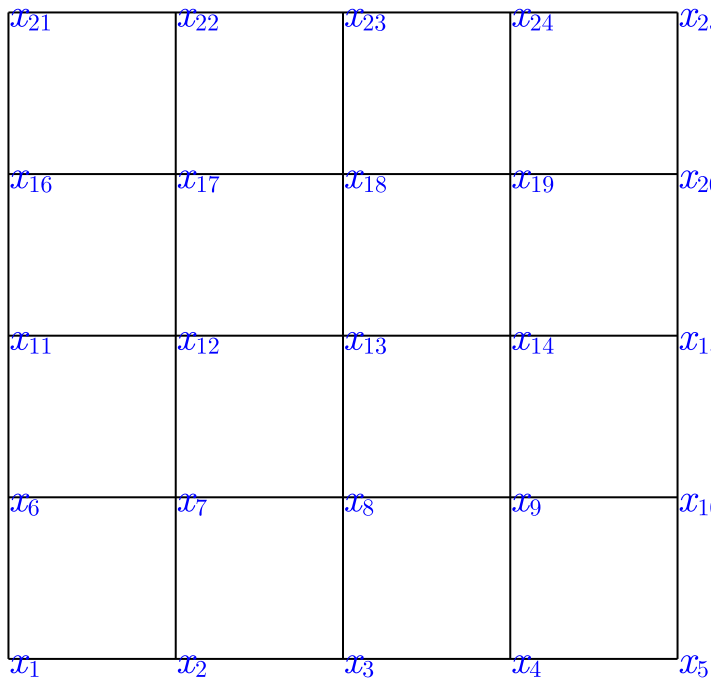
We use 2D regular discretization $n \times n$ grid with grid points $x_{ij} = ((i - 1)h, (j - 1)h)$. The finite difference approximation yields:

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

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)

```
1 function heatmatrix2d!(A,n;α=1)
2     function update_pair(A,v,i,j)
3         A[i,j]+=-v
4         A[j,i]+=-v
5         A[i,i]+=v
6         A[j,j]+=v
7     end
8     N=n^2
9     h=1.0/(n-1)
10    l=1
11    for j=1:n
12        for i=1:n
13            if i<n
14                update_pair(A,1.0,l,l+1)
15            end
16            if i==1 || i==n
17                A[l,l]+=α
18            end
19            if j<n
20                update_pair(A,1,l,l+n)
21            end
22            if j==1 || j==n
23                A[l,l]+=α
24            end
25            l=l+1
26        end
27    end
28    A
29 end
```

heatrhs2d (generic function with 1 method)

```

1 function heatrhs2d(n; rhs=(x,y)->0,bc=(x,y)->0,α=1.0)
2     h=1.0/(n-1)
3     x=collect(0:h:1)
4     y=collect(0:h:1)
5     N=n^2
6     f=zeros(N)
7     for i=1:n-1
8         for j=1:n-1
9             ij=(j-1)*n+i
10            f[ij]+=h^2/4*rhs(x[i],y[j])
11            f[ij+1]+=h^2/4*rhs(x[i+1],y[j])
12            f[ij+n]+=h^2/4*rhs(x[i],y[j+1])
13            f[ij+n+1]+=h^2/4*rhs(x[i+1],y[j+1])
14        end
15    end
16
17    for i=1:n
18        ij=i
19        fac=h
20        if i==1 || i==n
21            fac=h/2
22        end
23        f[ij]+=fac*α*bc(x[i],0)
24        ij=i+(n-1)*n
25        f[ij]+=fac*α*bc(x[i],1)
26    end
27    for j=1:n
28        fac=h
29        if j==1 || j==n
30            fac=h/2
31        end
32        ij=1+(j-1)*n
33        f[ij]+=fac*α*bc(0,y[j])
34        ij=n+(j-1)*n
35        f[ij]+=fac*α*bc(1,y[j])
36    end
37    f
38 end

```

n = 20

```
1 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,
```

```
1 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

```

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

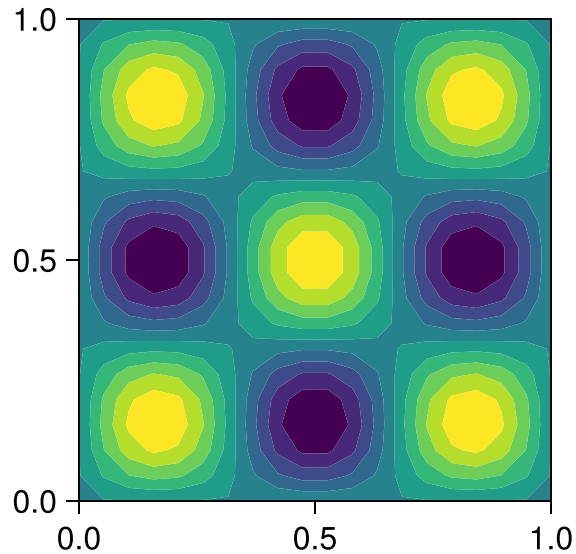
```
1 using Tables
```

| | Column1 | Column2 | Column3 | Column4 | Column5 | Column6 | Column7 | Column8 | Co |
|-----|---------|---------|---------|---------|---------|---------|---------|---------|-------|
| 1 | 202.0 | -1.0 | 0.0 | 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 | 0.0 |
| 3 | 0.0 | -1.0 | 103.0 | -1.0 | 0.0 | 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 | 0.0 |
| 5 | 0.0 | 0.0 | 0.0 | -1.0 | 103.0 | -1.0 | 0.0 | 0.0 | 0.0 |
| 6 | 0.0 | 0.0 | 0.0 | 0.0 | -1.0 | 103.0 | -1.0 | 0.0 | 0.0 |
| 7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -1.0 | 103.0 | -1.0 | 0.0 |
| 8 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -1.0 | 103.0 | -1.0 |
| 9 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -1.0 | 103.0 |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -1.0 |
| 11 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 12 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 13 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 14 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

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(resolution=(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),cmap="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]
```

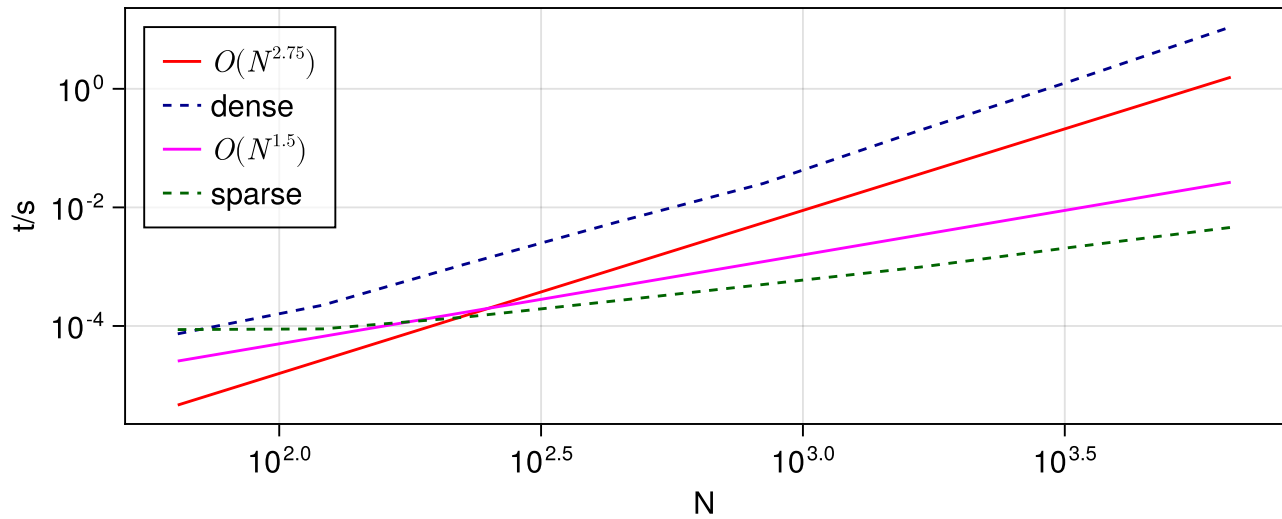
```
times_full2d =
```

```
[7.3705e-5, 0.000224311, 0.00108602, 0.00563992, 0.025404, 0.202459, 1.3514, 11.066]
```

```

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

```
1 using SparseArrays
```

```
[8.6963e-5, 8.9272e-5, 0.000140251, 0.000267167, 0.000501449, 0.000987738, 0.00211547, 0
```

```
1 if try_sparse
2   times_sparse2d= let
3     times=zeros(0)
4     for n in 2d
5       A0=spzeros(n^2,n^2)
6       A=heatmatrix2d!(A0,n,α=α)
7       b=heatrhs2d(n,α=α)
8       t=@elapsed u=A\b
9       push!(times,t)
10    end
11    times
12 end
13 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
```

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

```
1 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]
```

```
1 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:

```
Vector{Float64}[
 1: [1.0, 2.0, 3.0]
 2: [4.0]
 3: [5.0, 6.0, 7.0]
 4: [8.0, 9.0, 10.0]
 5: [11.0, 12.0]
]
```

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

```
Vector{Int64}[
  1: [1, 2, 3]
  2: [2]
  3: [1, 3, 4]
  4: [1, 2, 4]
  5: [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

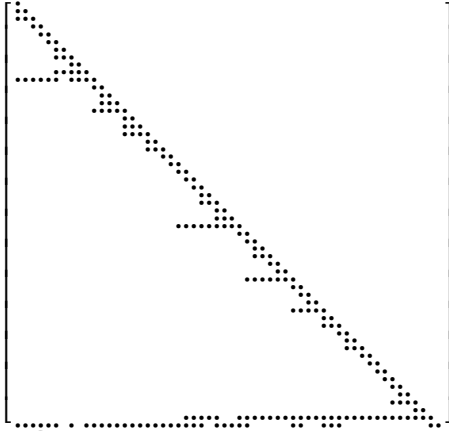
```
1 np=Int(ceil(Np^(1/dim)))
```

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

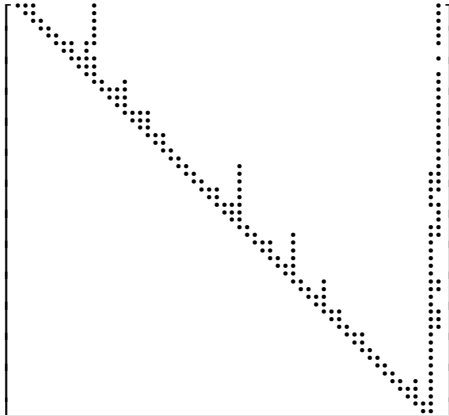


```
1 if dim==1
2   Ap=heatmatrix1d!(spzeros(Np,Np),np)
3 elseif dim==2
4   Ap=heatmatrix2d!(spzeros(np^2,np^2),np)
5 end
```

```
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 reasonable 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

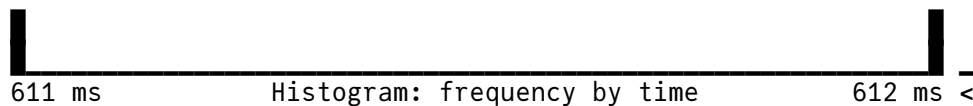
`N3 = 200`

```
1 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): 610.735 ms ... 612.232 ms | GC (min ... max): 0.00% ... 0.00%
Time (median):      611.484 ms                | GC (median):      0.00%
Time (mean ± σ):    611.484 ms ± 1.059 ms    | GC (mean ± σ):   0.00% ± 0.00%
```



Memory estimate: 6.22 MiB, allocs estimate: 30.

```
1 @benchmark heatmap2d!(spzeros(N3^2, N3^2), N3)
```

BenchmarkTools.Trial: 31 samples with 1 evaluation.

| | | | |
|--------------------------|-------------------------|------------------------|-----------------|
| Range (min ... max): | 31.644 ms ... 34.935 ms | GC (min ... max): | 0.00% ... 7.25% |
| Time (median): | 31.942 ms | GC (median): | 0.00% |
| Time (mean ± σ): | 32.899 ms ± 1.359 ms | GC (mean ± σ): | 2.61% ± 3.40% |



Memory estimate: 34.30 MiB, allocs estimate: 66.

```

1 let
2   A=heatmap2d!(spzeros(N3^2,N3^2),N3)
3   b=rand(N3^2)
4   @benchmark $A\b
5 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)

```

1 function heatmatrix2d_coo(n;α=1)
2     I=Int[]
3     J=Int[]
4     A=Float64[]
5     function addentry(i,j,v)
6         push!(I,i)
7         push!(J,j)
8         push!(A,v)
9     end
10    function update_pair(i,j,v)
11        addentry(i,j,-v)
12        addentry(j,i,-v)
13        addentry(i,i,v)
14        addentry(j,j,v)
15    end
16    N=n^2
17    h=1.0/(n-1)
18    l=1
19    for j=1:n
20        for i=1:n
21            if i<n
22                update_pair(l,l+1,1)
23            end
24            if i==1 || i==n
25                addentry(l,l,α)
26            end
27            if j<n
28                update_pair(l,l+n,1)
29            end
30            if j==1 || j==n
31                addentry(l,l,α)
32            end
33            l=l+1
34        end
35    end
36    sparse(I,J,A)
37 end

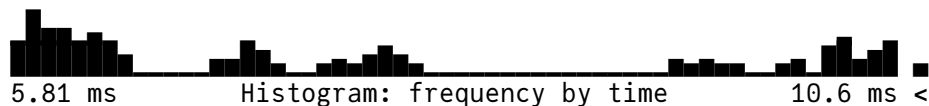
```

true

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

BenchmarkTools.Trial: 133 samples with 1 evaluation.

| | | | |
|----------------------|------------------------|-------------------|------------------|
| Range (min ... max): | 5.810 ms ... 10.734 ms | GC (min ... max): | 0.00% ... 24.61% |
| Time (median): | 6.946 ms | GC (median): | 0.00% |
| Time (mean ± σ): | 7.525 ms ± 1.758 ms | GC (mean ± σ): | 10.12% ± 12.34% |



Memory estimate: 24.51 MiB, allocs estimate: 57.

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

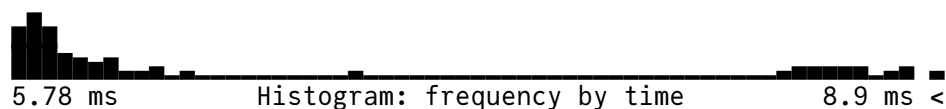
This approach requires to modify the structure of the assembly loop. If we run through this loop once this is ok. If one wants to update the nonzero entries, one needs to implement this loop twice. Moreover, one loses the intuitive way of writing into a matrix.

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.

```
1 using ExtendableSparse
```

BenchmarkTools.Trial: 160 samples with 1 evaluation.

```
Range (min ... max): 5.781 ms ... 8.940 ms | GC (min ... max): 0.00% ... 29.26%
Time (median):      5.903 ms                | GC (median):      0.00%
Time (mean ± σ):    6.268 ms ± 923.082 μs | GC (mean ± σ):  5.14% ± 9.85%
```



Memory estimate: 11.54 MiB, allocs estimate: 30.

```
1 @benchmark begin
2   Aext=heatmatrix2d!(ExtendableSparseMatrix(N3^2,N3^2),N3)
3   flush!(Aext)
4 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)

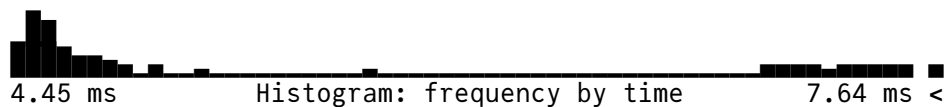
```

1 function heatmatrix2d_update!(A,n;α=1)
2     function update_pair(A,v,i,j)
3         updateindex!(A,+,-v,i,j)
4         updateindex!(A,+,-v,j,i)
5         updateindex!(A+,v,i,i)
6         updateindex!(A+,v,j,j)
7     end
8     N=n^2
9     h=1.0/(n-1)
10    l=1
11    for j=1:n
12        for i=1:n
13            if i<n
14                update_pair(A,1.0,l,l+1)
15            end
16            if i==1 || i==n
17                updateindex!(A+,α,l,l)
18            end
19            if j<n
20                update_pair(A,1.0,l,l+n)
21            end
22            if j==1 || j==n
23                updateindex!(A+,α,l,l)
24            end
25            l=l+1
26        end
27    end
28    A
29 end

```

BenchmarkTools.Trial: 201 samples with 1 evaluation.

| | | | |
|----------------------|-----------------------|-------------------|------------------|
| Range (min ... max): | 4.454 ms ... 7.869 ms | GC (min ... max): | 0.00% ... 33.69% |
| Time (median): | 4.588 ms | GC (median): | 0.00% |
| Time (mean ± σ): | 4.972 ms ± 944.552 μs | GC (mean ± σ): | 6.73% ± 11.78% |



Memory estimate: 11.54 MiB, allocs estimate: 30.

```

1 @benchmark begin
2     Aext=heatmatrix2d_update!(ExtendableSparseMatrix(N3^2,N3^2),N3)
3     flush!(Aext)
4 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 nonzero 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.
-

```
1 begin
2     using PlutoUI
3     using HypertextLiteral: @html, @html_str
4 end;
```