Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 12

```
begin
    using BenchmarkTools
    using SparseArrays
    using LinearAlgebra
end;
```

Sparse matrices

Sparse matrices

In the previous lectures we found examples of matrices from partial differential equations which have only 3 of 5 nonzero diagonals. For 3D computations this would be 7 diagonals. One can make use of this diagonal structure, e.g. when coding the progonka method in the case of tridiagonal matrices.

Matrices from unstructured meshes for finite element or finite volume methods have a more irregular pattern, but as a rule only a few entries per row compared to the number of unknowns. In this case storing the diagonals becomes unfeasible.

> **Definition**: We call a matrix *sparse* if regardless of the number of unknowns N, the number of non-zero entries per row and per column remains limited by a constant n_s independent of N.

- If we find a scheme which allows to store only the non-zero matrix entries, we would need not more than $Nn_s = O(N)$ storage locations instead of N^2
- The same would be true for the matrix-vector multiplication if we program it in such a way that we use every nonzero element just once: matrix-vector multiplication would use $O(N)$ instead of $O(N^2)$ operations

Storage formats

- What is a good storage format for sparse matrices?
- Is there a way to implement Gaussian elimination for general sparse matrices which allows for linear system solution with $O(N)$ operation ?
- Is there a way to implement Gaussian elimination *with pivoting* for general sparse matrices which allows for linear system solution with $O(N)$ operations?
- Is there *any algorithm* for sparse linear system solution with $O(N)$ operations?

Triplet storage format

- Store all nonzero elements along with their row and column indices
- One real, two integer arrays, length = nnz - number of nonzero elements

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$

AA	12.	9.	7.	5.	1.	2.	11.	3.	6.	4.	8.	10.
JR	5	3	3	2	1	1	4	2	3	2	3	4
JC	5	5	3	4	1	4	4	1	1	2	4	3

(Y.Saad, Iterative Methods, p.92)

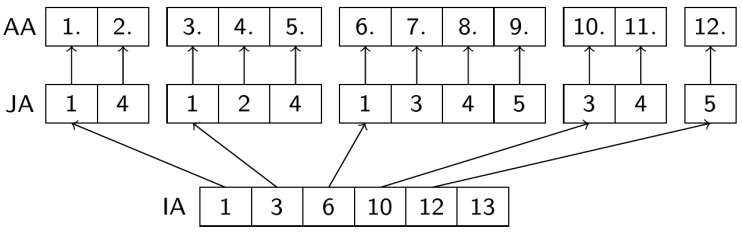
- Also known as Coordinate (COO) format
- This format often is used as an intermediate format for matrix construction

Compressed Sparse Row (CSR) format

(aka Compressed Sparse Row (CSR) or IA-JA etc.)

- float array `AA`, length nnz, containing all nonzero elements row by row
- integer array `JA`, length nnz, containing the column indices of the elements of `AA`
- integer array `IA`, length N+1, containing the start indizes of each row in the arrays `IA` and `JA` and `IA[N+1]=nnz+1`

$$A = \begin{pmatrix} 1. & 0. & 0. & 2. & 0. \\ 3. & 4. & 0. & 5. & 0. \\ 6. & 0. & 7. & 8. & 9. \\ 0. & 0. & 10. & 11. & 0. \\ 0. & 0. & 0. & 0. & 12. \end{pmatrix}$$



- Used in many sparse matrix solver packages

Compressed Sparse Column (CSC) format

- Uses similar principle but stores the matrix column-wise.
- Used in Julia

Sparse matrices in Julia

To explain the CSC format used in Julia we create first a sparse matrix from a full matrix:

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

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

```
As = 5×5 SparseMatrixCSC{Float64, Int64} with 12 stored entries:
 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=sparse(A)
```

```
column 1: 1:3
    rows: 1 2 3
  values: 1.0 3.0 6.0

column 2: 4:4
    rows: 2
  values: 4.0

column 3: 5:6
    rows: 3 4
  values: 7.0 10.0

column 4: 7:10
    rows: 1 2 3 4
  values: 2.0 5.0 8.0 11.0

column 5: 11:12
    rows: 3 5
  values: 9.0 12.0
```

```
showsparse(As)
```

```
[1, 4, 5, 7, 11, 13]
```

```
As.colptr
```

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

```
As.rowval
```

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

```
As.nzval
```

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

Sparse matrix creation

Create a sparse matrix from a dense matrix

```
3×3 SparseMatrixCSC{Int64, Int64} with 1 stored entry:
 ⋅  ⋅  ⋅
 1  ⋅  ⋅
 ⋅  ⋅  ⋅
```

```
sparse([0 0 0 ;
        1 0 0 ;
        0 0 0 ])
```

Create a random sparse matrix

```
N = 100
```

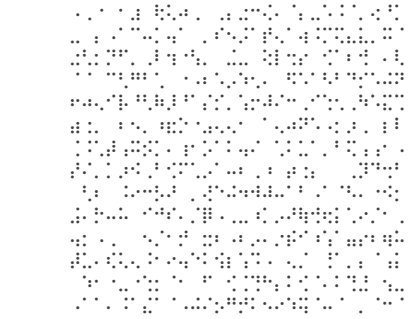
```
N=100
```

```
p = 0.1
```

```
p=0.1
```

Random sparse matrix with probability p=0.1 that A_{ij} is nonzero:

```
A2 = 100×100 SparseMatrixCSC{Float64, Int64} with 947 stored entries:
```



```
A2=sprand(N,N,p)
```

Creation of a sparse matrix from given data

This is the most important question when it comes to finite volume and finite element methods. We demonstrate the method with a tridiagonal matrix.

```
N1 = 10000
```

```
N1=10000
```

```
a =
```

```
[-0.0327213, -0.766082, -0.1912, -0.276464, -0.991639, -0.24643, -0.530243, -0.102243,
```

```
a=-rand(N1-1)
```

```
b =
```

```
[2.84474, 2.00138, 2.78509, 2.09571, 2.81323, 2.2436, 2.89416, 2.14314, 2.82984, 2.7401
```

```
b=rand(N1).+2
```

```
c =
```

```
[-0.901338, -0.274691, -0.776477, -0.596193, -0.576889, -0.160763, -0.421551, -0.82552
```

```
c=-rand(N1-1)
```

Special case: use the Julia tridiagonal matrix constructor

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

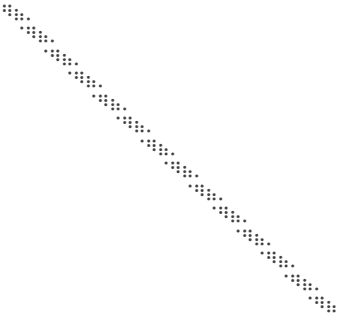
```
sptri_special(a,b,c)=sparse(Tridiagonal(a,b,c))
```

Create an empty Julia sparse matrix and fill it incrementally

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

```
function sptri_incremental(a,b,c)
```

```
10000×10000 SparseMatrixCSC{Float64, Int64} with 29998 stored entries:
```



```
sptri_incremental(a,b,c)
```

Coordinate (COO) format as intermediate storage

This is the recommended default way.

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

```
function sptri_coo(a,b,c)
```

ExtendableSparse.jl package

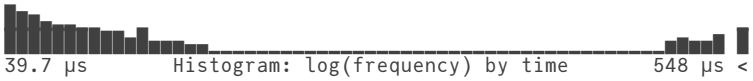
This implicitely uses the so-called linked list format for intermediate storage of new entries. Note the flush!() method which needs to be called in order to transfer them to the Julia sparse matrix structure.

```
using ExtendableSparse
```

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

```
function sptri_ext(a,b,c)
```

```
BechmarkTools.Trial: 10000 samples with 1 evaluations.
 Range (min … max):  39.691 μs … 968.349 μs  ┊ GC (min … max):  0.00% … 87.51%
 Time  (median):     46.334 μs               ┊ GC (median):     0.00%
 Time  (mean ± σ):   56.917 μs ±  62.757 μs  ┊ GC (mean ± σ):  11.45% ±  9.72%
```



```
 39.7 μs       Histogram: log(frequency) by time       548 μs <

 Memory estimate: 547.27 KiB, allocs estimate: 8.
```

```
@benchmark sptri_special(a,b,c)
```

```
BechmarkTools.Trial: 259 samples with 1 evaluations.
 Range (min … max):  18.668 ms …  21.114 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     19.215 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   19.315 ms ± 399.091 μs  ┊ GC (mean ± σ):  0.08% ± 0.48%
```

```
 18.7 ms         Histogram: frequency by time         20.5 ms <

 Memory estimate: 1.08 MiB, allocs estimate: 33.
```

```
@benchmark sptri_incremental(a,b,c)
```

```
BechmarkTools.Trial: 6048 samples with 1 evaluations.
 Range (min … max):  699.097 μs …   2.428 ms  ┊ GC (min … max): 0.00% … 38.58%
 Time  (median):     729.398 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   822.275 μs ± 224.632 μs  ┊ GC (mean ± σ):  5.22% ± 10.82%
```

```
 699 μs        Histogram: log(frequency) by time       1.67 ms <

 Memory estimate: 2.65 MiB, allocs estimate: 66.
```

```
@benchmark sptri_coo(a,b,c)
```

```
BechmarkTools.Trial: 5452 samples with 1 evaluations.
 Range (min … max):  781.280 μs …   2.727 ms  ┊ GC (min … max): 0.00% … 26.89%
 Time  (median):     834.764 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   913.933 μs ± 205.775 μs  ┊ GC (mean ± σ):  2.91% ±  7.98%
```

```
 781 μs        Histogram: log(frequency) by time       1.95 ms <

 Memory estimate: 1.53 MiB, allocs estimate: 25.
```

```
@benchmark sptri_ext(a,b,c)
```

- **Caution!** The incremental creation of a SparseMartrixCSC from an initial state with no nonzero entries is slow because of the data shifts and reallocations necessary during the construction.
- Use the COO intermediate format as an alternative way.
- ExtendableSparse package may be more convenient.

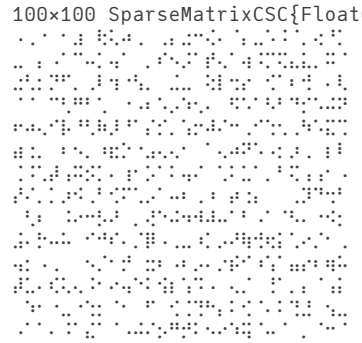
Sparse direct solvers

- Sparse direct solvers implement LU factorization with different pivoting strategies. Some examples:
  - UMFPACK: e.g. used in Julia
  - Pardiso (omp + MPI parallel)
  - SuperLU (omp parallel)
  - MUMPS (MPI parallel)
  - Pastix
- Quite efficient for 1D/2D problems - we will discuss this more deeply
- Essentially they implement the LU factorization algorithm
- They suffer from *fill-in*, especially for 3D problems:

Let $A = LU$ be an LU-Factorization. Then, as a rule, $nnz(L+U) >> nnz(A)$.

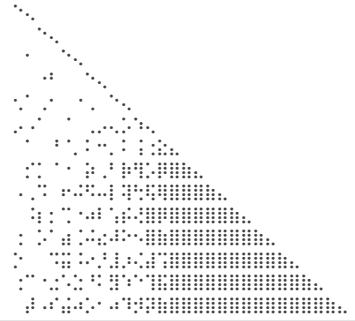
- increased memory usage to store L,U
- high operation count

```
100×100 SparseMatrixCSC{Float64, Int64} with 947 stored entries:
```



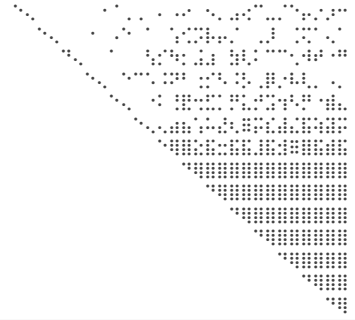
```
A2
```

```
100×100 SparseMatrixCSC{Float64, Int64} with 1919 stored entries:
```



```
lu(A2).L
```

```
100×100 SparseMatrixCSC{Float64, Int64} with 1900 stored entries:
```



```
lu(A2).U
```

```
(947, 3819)
```

```
nnz(A2), nnz(lu(A2))
```

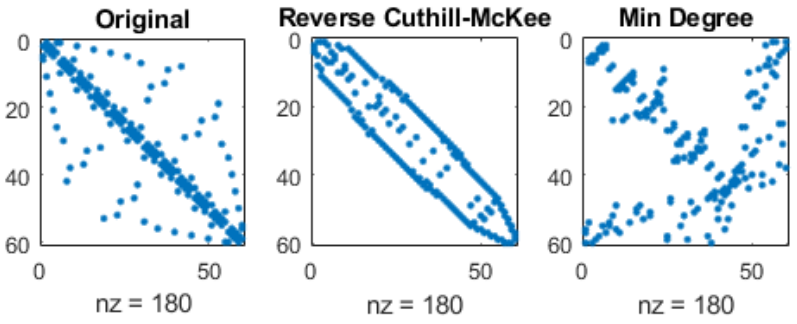
Solution steps with sparse direct solvers

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

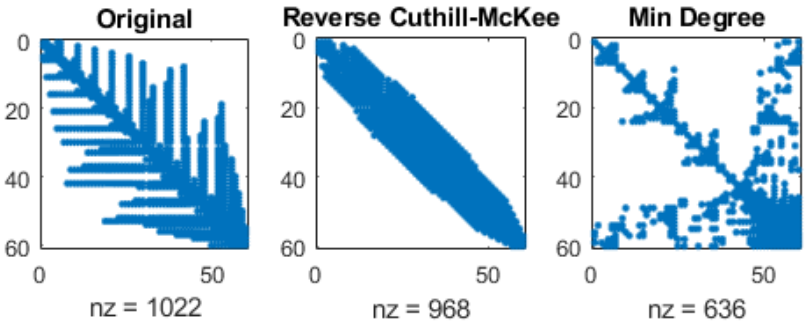
Influence of reordering

- Sparsity patterns for original matrix with three different orderings of unknowns
  - number of nonzero elements (of course) independent of ordering:



(mathworks.com)

- Sparsity patterns for corresponding LU factorizations
  - number of nonzero elements depend original ordering!



(mathworks.com)

Complexity estimate

- 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

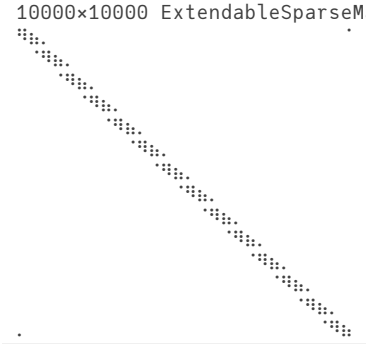
d	$work$
1	$O(N) \mid O(n)$
2	$O(N \log N) \mid O(n^2 \log n)$
3	$O(N^{\frac{4}{3}}) \mid O(n^4)$

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

Practical use

- For a sparse matrix, the julia `\` operator is realized via sparse LU factorization.
- When solving repeated problems with the same sparse matrix, consider reusing the LU factorization

```
10000×10000 ExtendableSparseMatrix{Float64, Int64} with 30000 stored entries:
```



```
begin
```

```
f =
```

```
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.
```

```
f=ones(N1)
```

```
[0.551143, 0.63002, 0.729658, 0.707706, 0.576385, 0.738183, 0.526362, 0.81002, 0.553449
```

```
A3\f
```

```
LU3=lu(A3);
```

```
0.0
```

```
norm(LU3\f-A3\f)
```

```
BechmarkTools.Trial: 774 samples with 1 evaluations.
 Range (min … max):  5.628 ms …   8.530 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     6.221 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   6.286 ms ± 446.859 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
 Memory estimate: 11.00 MiB, allocs estimate: 73.
```

```
@benchmark A3\f
```

```
40790
```

```
nnz(LU3.fact)
```

```
30000
```

```
nnz(A3)
```

```
BechmarkTools.Trial: 749 samples with 1 evaluations.
 Range (min … max):  5.606 ms …  11.276 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     6.546 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   6.660 ms ± 660.076 μs  ┊ GC (mean ± σ):  2.21% ± 3.81%
```


```
 Memory estimate: 10.32 MiB, allocs estimate: 66.
```

```
@benchmark lu(A3)
```

```
BechmarkTools.Trial: 10000 samples with 1 evaluations.
 Range (min … max):  262.172 μs … 579.673 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     276.753 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   283.725 μs ±  26.440 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
 Memory estimate: 546.98 KiB, allocs estimate: 4.
```

```
@benchmark LU3\f
```

> The LU factorization is the most expensive operation with sparse direct solvers. Whenever possible, try to reuse it.

```
begin
```

```
begin
```