Scientific Programming (2MMN20)
Exercise set 2
Sparse matrices and Graphs

dr. Joseph M. Maubach

November 4, 2020

0. Hand in one report (file) which for all questions contains

1. Example input
2. Script
3. Outputs: Output data and figures.

Figures must have *labeled axes and title*. Copy-paste [inputs/script/outputs/figures] into LaTeX, MSWord or Matlab reporter. When needed use 2MMN20 `xformlatex()` or `xformmatlab()`.

1. Make sure your scripts can be copy-pasted from your `.pdf` into Matlab without problems except for with single quotes (').

2. Unless mentioned otherwise these control instructions are not allowed:

```
if x == 1; y = 1, else y = 0, end;
for x=[0,1]; y = x, end;
while ~isempty(x); y = x(end), x(end) = []; end;
```

3. For $> 16 \times 16$ matrices red colored instructions of Table **??** are not allowed.

4. "1 statement" stands for 1 nested **non-control** statement such as:

`1 + 2`	`1 + (2 + 3)`
`ones(3, 1 + 2)`	`ones(3, 1 + 2).^2`
`v = (1+3./(1:8)).^(1:8)`	`f = @(x) x.^2 * min(x)`
`reshape(repmat(1:4,[4,1]),[1,16])`	

Functions which consist of one statement such as

```
function y = square(x)
y = x.^2;
end
```

also count as 1 statement. If an exercise mentions "max k statements" you may use maximally k such (nested) statements, seperated by comma or semi-colon. `X % -> Y` abbreviates: Statement(s) `X` produces output `Y`.

5. To "measure your script's execution time" , first switch off/pause other running programs such as music/video and chat programs. "time" your scripts at least 3 times and report **the average** time.

6. When asked to provide a 1–3–5 statement script:

1. It is possible to find a 1 statement script, but not necessarily with optimal performance;
2. It is possible to find a 3 statement script with optimal performance;
3. You may use up to and including 5 statements.

7. All non-bonus questions: You can earn 0 (Fail), 1 (Sufficient) or 2 (Good) credit points. When asked to provide a 1–3–5 statement k-statement script:

1. > 5 statements: ≤ 1 credit point
2. Use of not permitted control instruction(s): 0 credit points
3. Use of non-efficient Matlab instructions: ≤ 1 credit points
4. Creating an $n \times n$ sparse format matrix from a full $n \times n$ format matrix: ≤ 0.5 credit points
5. Not copy-pasteable scripts from your `.pdf`: 0 credit points
6. 1–5 statement(s) correct answers **all** score maximal

Bonus questions: Feedback and 0 credits.

8. Example question and the expected motivated 2PTS answer:

1. Q: Write a 1 statement script that for input $\mathbf{v} = [v_1, \ldots, v_n]$ returns $\mathbf{v} = [v_1^2, \ldots, v_n^2]$:
2. A: Your test input(s), your script, and the resulting output(s):

```
>> v = [1,4,3,2]      % your test input data -- does not count as statements
v =

   1   4   3   2

>> v = v.^2           % your script -- all statements count
v =

    1   16    9    4  % your results -- does not count as statements
```

```
>>
```

9. Example question and an incorrect 0.5PTS answer:

1. Q: Write a 5 − 7 statement script that for inputs n and d returns an $n \times n$ lower triangular sparse format matrix with approximately $d\%$ entries
2. A: This answer first creates a full format matrix with `randi()` and next forces it sparse format

   ```
   L = tril(randi([-k,k],n),-1); % creates a full format matrix ... 0.5PTS
   nzrs = find(L);
   L(nzrs(randperm(length(nzrs),((n-1)*n/2)-d))) = 0; % sets entries to zero
   L = sparse(L) % converts into sparse matrix
   ```

   which implies that this script involves $O(n^2)$ OPS for $n \to \infty$.

Exercise 1. **Sparse matrices**. Let

$$\mathbf{A} = \begin{bmatrix} -2 & -8 & -7 & 0 & 0 & 1 \\ 1 & 2 & 0 & -8 & 0 & 0 \\ -5 & 0 & 1 & -7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 8 & -8 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{6\times 6}$$

be created by

```
A = [ ...
  -2,  -8,  -7,   0,   0,   1; ...
   1,   2,   0,  -8,   0,   0; ...
  -5,   0,   1,  -7,   0,   0; ...
   0,   0,   0,   0,   0,   0; ...
   0,   5,   0,   0,   8,  -8; ...
   0,   0,   0,   0,   0,   0; ...
]
```

a. Write a script which outputs `n, m, I, J, V` for the COO format of $\mathbf{A}$

b. Write a script which outputs `n, m, I, N, V` for the CCS format of $\mathbf{A}$

c. Let $k, l \in \mathbb{Z}$, $n, m \in \mathbb{N}$ and $d \in (0, 1)$. Write a 2–5 statement function

```
function A = sprandi(interval, n, m, d)
...
...
...
```

such that input `sprandi([k,l], n, m, d)` returns a $n \times m$ sparse format matrix with approximately $d \cdot m \cdot n$ uniformly distributed random integer entries in range $[k, k+1, \ldots, l-1, l]$.

Exercise 2. Altering entries of a matrix: Let $\mathbf{A}$ be generated by `matlab`:

```
A = randi([0,1],[8,8])
```

and `python`:

```
A = np.random.randint(*[0,2], (8,8))
```

Use commands such as `find()`, `sparse()` and `spones()` to:

a. Write a 2 statements script which alters **all non-zero** entries a_{ij} into $10 \cdot i + j$

b. Write a 2 statements script which alters **all** entries a_{ij} into $10 \cdot i + j$

c. Write a 1–2 statement(s) script which alter **all non-zero** entries a_{ij} in columns 2 and 6 into number 7. You are not allowed to use that columns 2 and 6 starts at linear index 8 resp. 40

Exercise 3. On sparse format matrix creation. Let $n, k, l \in \mathbb{N}_+$.

a. Write a 3 – 4–6 statements linear-time script which for inputs n, k, l returns a sparse format $n \times n$ unit lower triangular matrix $\mathbf{L}$ with approximately l off-diagonal entries in $\{-k, \ldots, -1, 0, 1, \ldots, k\}$. You are not allowed to use the command `spdiags()`.

b. Let $\mathbf{L} \in \mathbb{R}^{n\times n}$ be unit lower triangular and $\mathbf{D} \in \mathbb{R}^{n\times n}$ be a diagonal matrix with positive diagonal elements. Show that $\mathbf{LDL}^{\mathrm{T}}$ is positive definite.

c. Let $\mathbf{L}$ be defined as in the first question and let

```
D = diag(randi([1,k],[n,1]));
```

Write a 1 statement script which for inputs $\mathbf{L}$ and $\mathbf{D}$ returns a symmetric positive definite matrix $\mathbf{A} \in \mathbb{Z}^{n\times n}$, with only integer entries.

d. Show that for all matrices $\mathbf{B} \in \mathbb{R}^{n\times n}$ and all $\mathbf{x} \in \mathbb{R}^n$

$$((\mathbf{B} - \mathbf{B}^{\mathrm{T}})\mathbf{x}, \mathbf{x})_2 = 0.$$

e. Let $\mathbf{L} \in \mathbb{Z}^{n\times n}$ be defined as in the first question and let

```
D = diag(randi([1,k],[n,1]));
```

Write a 1–2 statement script which for inputs $\mathbf{L} \in \mathbb{Z}^{n\times n}$ and $\mathbf{D} \in \mathbb{Z}^{n\times n}$ returns a non-symmetric positive definite matrix $\mathbf{A} \in \mathbb{Z}^{n\times n}$, with only integer entries. Hint: First create a symmetric positive definite matrix $\mathbf{LDL}^{\mathrm{T}}$ as before. Then create a skew-symmetric matrix (for instance use $\mathbf{L}$) and add it to $\mathbf{LDL}^{\mathrm{T}}$.

Exercise 4. **Sparse matrices**. Not allowed are instructions which cause your scrip to be inefficient: The use of `mod()`, `floor()`, `fix()` and other rounding operators; Full format matrices; The use of set commands such as

ismember(), intersect().
Let $G(V, A)$ be a graph with $n = 8$ vertices $V = v_1, \ldots, v_n$ have connectivity matrix

$$\mathbf{C} = \begin{array}{c} v/w \in V \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{bmatrix} 1 & & 1 & & & & & \\ & & & & 1 & & & 1 \\ 1 & & 1 & & & 1 & 1 & \\ 1 & & 1 & & 1 & 1 & 1 & 1 \\ 1 & & 1 & 1 & 1 & & & \\ & & 1 & & & 1 & 1 & \\ 1 & & & & 1 & & & \\ & 1 & & & & 1 & 1 & \end{bmatrix} \end{array} \in \mathbb{N}^{8 \times 8}$$

`matlab`creation – the coordinates of vertex k are in the k-th row of `XY`:

```
n = 8
V = 1:n
C = sparse([ ...
   1,   0,   1,   0,   0,   0,   0,   0; ...
   0,   0,   0,   0,   1,   0,   0,   1; ...
   1,   0,   1,   0,   0,   1,   1,   0; ...
   1,   0,   1,   0,   1,   1,   1,   1; ...
   1,   0,   1,   1,   1,   0,   0,   0; ...
   0,   0,   1,   0,   0,   1,   1,   0; ...
   1,   0,   0,   0,   1,   0,   0,   0; ...
   0,   1,   0,   0,   0,   1,   1,   0; ...
]); full(C)

XY = [[1;2;3;1;2;3;1;2],[1;1;1;2;2;2;3;3]];
```

and `python`creation:

```
n = 8
V = np.arange(n)
C = sparse.dok_matrix([[1, 0, 1, 0, 0, 0, 0, 0],
                       [0, 0, 0, 0, 1, 0, 0, 1],
                       [1, 0, 1, 0, 0, 1, 1, 0],
                       [1, 0, 1, 0, 1, 1, 1, 1],
                       [1, 0, 1, 1, 1, 0, 0, 0],
                       [0, 0, 1, 0, 0, 1, 1, 0],
                       [1, 0, 0, 0, 1, 0, 0, 0],
                       [0, 1, 0, 0, 0, 1, 1, 0]])
XY = np.array([[1,2,3,1,2,3,1,2],[1,1,1,2,2,2,3,3]])

# only if plot_graph was in your 2mmn20.zip download:
plot_graph(C,XY)
```

An alternative `python` answer:

```
from scipy import sparse as sp
import numpy as np
import networkx as nx
```

```
from network2tikz import plot
import itertools

n = 8
C = sp.csc_matrix(np.random.randint(0, 2, np.power(n, 2)).reshape(n, n))
G = nx.Graph(C)
XY = list(itertools.product(range(np.ceil(np.sqrt(n)).astype(int)), repeat=2))
# ceil also still works for dimension n = 3 (but no more for n = 4)
print(C.todense())

[C.nonzero()] = range(1, C.nonzero()[0].size + 1)
print(C.todense())

np.random.shuffle(XY)
plot(G,
 filename='../src/images/set2/4a.pdf',
 layout=dict(enumerate(XY)),
 canvas=(n,n),
 margin=1)
```

a. Write a script (for arbitrary inputs **C** and `XY`) which plots the undirected graph related to **C**. `matlab`Hint: Use `gplot()`. `python`Hint: Must make your own `plot_graph` or ask lector.

b. Write a script (for arbitrary inputs **C** and `XY`) which labels (writes numbers next to) the vertices and edges. Hint: Use `text()`.

The neighbors of vertex k (the other vertices to which it is connected) have an entry with value 1 in row k: For instance:

- vertex v_3 is connected to v_1, v_3, v_6 and v_7 (`3` has connections `1,3,6,7`)
- vertex v_3 has neighbors v_1, v_6 and v_7 (`3` has neighbors `1,6,7`).

Vertices v_3, v_1, v_7

- Are connected to (in order) $[1, 3, 6, 7; 1, 3; 1, 5]$
- Have neighbors (in order) $[1, 6, 7; 3; 1, 5]$.

The connections of v_3 can be found with `find(C(3,:)) %-> 1, 3, 6, 7`.

Let $n = |V|$. Define sequence of "unprocessed" vertices $\mathbf{U} \in \mathbb{N}^n$ by:

$$U_k = \begin{cases} k & \text{if vertex } k \text{ is unprocessed} \\ 0 & \text{if vertex } k \text{ is processed.} \end{cases}$$

Initially, before processing any of the vertices, $\mathbf{U} = [1, 2, \ldots, n]$. I.e., $\mathbf{U}$ serves as indicator, but using value $k > 0$ instead of $1 > 0$.

For a given sequence of processed vertices we want to obtain the sequence of their unprocessed neighbor vertices, i.e., all their neigbors k for which $U(k) = k > 0$.

c. Assume that the only unprocessed vertices are $[5, 7, 3]$. The vector $\mathbf{U}$ which reflects this state is:

```
n = 8, U = 1:n, U([1,2,4,6,8]) = 0
```

Write a 1 statement script which filters out the unprocessed vertices, leaving:

$$V = [5; 3; 7],$$

using the sequence `U`.

d. Write an efficient 2–4 statement function `degr(C,VL)` which determines the degree for a sequence of vertices: for instance for $VL = [3, 1, 7]$ function `degr(C,VL)` should return $[\text{degree}(v_3), \text{degree}(v_1), \text{degree}(v_7)]$.

e. Write a $1 - 2$ statement script which for input the sparse format matrix $\mathbf{C}$ outputs a *sparse connection matrix* $\mathbf{N}$ with **each column** k now related to all connections of vertex v_k:

$$\mathbf{N} = \begin{bmatrix} 1 & & 1 & 1 & 1 & & 1 & \\ & & & & & & & 2 \\ 3 & & 3 & 3 & 3 & 3 & & \\ & & & & 4 & & & \\ & 5 & & 5 & 5 & & 5 & \\ & & 6 & 6 & & 6 & & 6 \\ & & 7 & 7 & & 7 & & 7 \\ & 8 & & 8 & & & & \end{bmatrix}$$

You may assume that $\mathbf{C} = \mathbf{C}^{\mathrm{T}}$. Hint: One of the statements is `find()`.

f. Based on $\mathbf{N}$ write a 1-statement script to for all vertices sort their connections, to obtain

$$\begin{bmatrix} & & & & & & & \\ & & & & & & & \\ & & & 1 & & & & \\ & & & 3 & & & & \\ & & 1 & 5 & 1 & & & \\ & & 3 & 6 & 3 & 3 & & 2 \\ 1 & 5 & 6 & 7 & 4 & 6 & 1 & 6 \\ 3 & 8 & 7 & 8 & 5 & 7 & 5 & 7 \end{bmatrix}$$

The simultaneous sort of all neighbors is more efficient than processing the neighbors of one vertex at a time.

g. Based on **N**: Apply the statement to vertices $[2, 4, 1]$ (i.e., to `N(:,[2,4,1])` to obtain

$$\begin{bmatrix} & & \\ & & \\ & 1 & \\ & 3 & \\ & 5 & \\ & 6 & \\ 5 & 7 & 1 \\ 8 & 8 & 3 \end{bmatrix}$$

h. Write a 2–4 statements script to collect all neighbors of vertices $[2, 4, 1]$ into one sequence, eliminating the doubles **and** keeping the *column order*, to obtain the **full** neighbor vertex vector:

$$V = [\ \underbrace{5; 8}_{\text{from 1-st column}}\ ;\ \underbrace{1; 3; 6; 7}_{\text{from 2-nd column}}\ ].$$

`matlab`: Hint: These are a first possible statements of a 3 statement solution:

```
[~,J,V] = find(N(:,[2, 4, 1]));
[~,Q] = sortrows([V,J],[2,1]);
V = V(Q)
...
```

but there is an even shorter (faster) one.

i. Write an efficient 1 statement script which for inputs $n \in \mathbb{N}_+$ and $d \in [0, 1]$ returns an $n \times n$ sparse directed connectivity matrix, with approximately $n^2 \cdot d$ non-zero entries.

j. Assume that $n = 8$, $V = [2, 4, 1]$ and that the unprocessed vertices are $[5, 3, 7]$. Test the combination of your scripts on an $n \times n$ matrix created in the previous question:

- Determine the unique unprocessed neighbors of vertices $[2, 4, 1]$
- Ensure you print your matrix $\mathbf{C}$
- And provide your script.

EXERCISE 5. **Graphs**. **Python users: Skip this question**.

All your connectivity, adjacency and incidence matrices should be sparse format. You are not allowed to use the R2015b novel commands `adjacency()` and `incidence()`.

a. Make a 2D-directed graph $G(V, A)$ of ≥ 10 points (vertices), which are partially connected by arcs. Do not take any 2MMN20 graph. (Ensure it differs from the other groups' choices). Write a Matlab script which defines:

(a) The point coordinate list: `XYZ`

(b) The graphs' arc list: `IJV`

(c) The graph's dimension: `nb_dimension = 2`

(d) The graph's amount of points: `nb_points`

(e) Determine the graph's sparse format connectivity matrix $\mathbf{C}$

(f) Plot your grid with `gplot3(C, XYZ, 'b-*')`, label your vertices with `text()` and include the figure in your report.

b. Write a 1–2 statement script which returns the sparse format connectivity matrix $\mathbf{C}$ for an arbitrary directed graph specified by its arclist and vertex coordinates. Maximally 0 control-structures are permitted.
Hint: One statement should contain `sparse(...)`

c. Write a 2–3 statement script which for input the grid's connectivity matrix returns the related sparse format adjacency matrix **C**. Hint: One statement should contain `sparse(...)`

d. Write a 1–5 statement script which calculates the graph's sparse format incidence matrix **C**.

EXERCISE 6. **Grids**. *All connectivity, adjacency and incidence matrices should be sparse.*

a. `matlab`: Run `GD_info` to determine how many grid examples are provided.
`python`: Skip this question.

b. Draw a grid of ≥ 10 connected 2-simplices, different from the example below (and ensure it differs from the other groups' choices).
Specify the following entities:

(a) The point coordinate list: `XYZ`

(b) The cells (to points) *Cell List*: `KLW`

(c) The cells *Neighbor List*: `NB` (**not 2018–2019 and later**)

(d) The grid's dimension 2; `nb_dimension = size(XYZ,2)`

(e) The grid's amount of points: `nb_points = size(XYZ,1)`

(f) The grid's amount of cells `nb_cells = size(KLW,1)`

c. Why are the connectivity, adjacency, and incidence matrix not uniquely defined?

For the remainder we choose to represent the grid by a directed graph $G(V, A)$ representation and for each cell $[p_1, p_2, p_3]$ we choose arcs (p_1, p_2), (p_2, p_3) and (p_3, p_1). Furthermore we define

```
nb_arcs = nb_cells*3 % or
nb_arcs = numel(KLW)
```

d. For your example grid write down the its connectivity matrix **C** based on input `KLW`

e. Using your connectivity matrix `C` plot your grid with `trimesh()` (for a 3-d grid use `tetramesh()`):

```
figure();
[KLW, NB, XYZ] = GD0();
trimesh(KLW,XYZ(:,1),XYZ(:,2))
figure();
[KLW, NB, XYZ] = GD1();
tetramesh(KLW,XYZ)
```

f. Write a 1–5 statement script which determines a connectivity matrix **C** for an arbitrary triangular grid. No for loops are permitted.

g. Write a 1 statement script which for input the grid's cell list calculates the related adjacency matrix **C**

h. Write a 1–2 statement script which for the input grid's cell list calculates an incidence matrix **C** for your grid.

i. Explain why for a triangular grid with many elements a sparse format **C** is to be preferred above a full format **C**.

Exercise 7. We now focus on tetrahedral example grid `GD1`:

a. Load and plot `GD1` with `VGD1`, include the print in your report (`VGD1` uses `tetramesh()`.
On `python` – ensure that `./2mmn20/gallery/grids/` is on the `python` path:

```
import numpy as np
# read and make cell list KLW (matrix: nb_cells x 4 vertices)
KLW = np.fromfile("GD1points.txt",dtype=int,sep=' ')
KLW = KLW.reshape((int(np.size(KLW)/4),4))
print(KLW)
# read and make XYZ coordinates (matrix: nb_vertices x 3 coordinates)
XYZ = np.fromfile("GD1XYZ.txt",dtype=float,sep=' ')
XYZ = XYZ.reshape((int(np.size(XYZ)/3),3))
print(XYZ)
```

Note that dimension information is lost . . . .

On `python` more specifically: Write to and read from file: Methods `fromfile()` and `tofile()` write and read full precision – but **does not write the matrix dimensions or write multiple columns (!?)**: write to and read from ascii file "by hand":

```
import numpy as np
IJVW = np.array([[1,2,3.0],[2,2,-1.0],[3,2,0.1],[4,2,1/3]]);
IJVW.tofile("IJV.txt",sep=" ",format="%s");
# well, tofile() does not save dimensions ....
IJVR = np.fromfile("IJV.txt",dtype=float,sep=' ')
IJVR = IJVR.reshape((4,3))
print(np.linalg.norm(IJVW-IJVR,2))
```

Write in computer platform independent manner (and read back): Methods `save()` and `load()` write and read full precision – and also save the dimensions – **but without dictionary only first variable saved**

```
import numpy as np
IJVW  =   np.array([[1,2,3.0],[2,2,-1.0],[3,2,0.1],[4,2,1/3]]);
IJVW2 = 2*np.array([[1,2,3.0],[2,2,-1.0],[3,2,0.1],[4,2,1/3]]);
# only the first matrix is saved ...
np.save('IJV.npy', IJVW, IJVW2)
IJVR = np.load('IJV.npy')
print(np.linalg.norm(IJVW-IJVR,2))
```

b. Write a 3 – 8 statement script which only plots those edges of the grid `GD1` which coincide with plane $x = 1/2$. Hint: One statement should contain `find()` and the other `gplot3()`

EXERCISE 8. **Complexity**. **WARNING: Timing for $k = 4$ in (c) can take several minutes per trial**. Assume that $n-1$ is an amount of intervals (along the x-axis). We consider the complexity of various reordering methods. we apply these methods to a uniform grid of $2 \cdot n^2$ triangles on $[0, 1]^2$ such as is shown for $n = 8$ in Figure 1, created with

```
% create uniform grid and define quantities
[XYZ, KLW, NB] = square_triangular_grid(8);
nb_cells = size(KLW,1)
nb_arcs = nb_cells*size(KLW,2)
nb_points = size(XYZ,1)

% create undirected connectivity matrix
IJV(:,1) = reshape(KLW,              [nb_arcs,1]);
IJV(:,2) = reshape(KLW(:, [2:end,1]),[nb_arcs,1]);
C = sparse(IJV(:,1),IJV(:,2),1,nb_points,nb_points);
C = spones(C+C');

% plot the graph related to the grid
trimesh(KLW,XYZ(:,1),XYZ(:,2))

% plot the graph related to the grid
gplot(C,XYZ,'*-b');
print('uniform_triangles_8.eps','-depsc2');
```

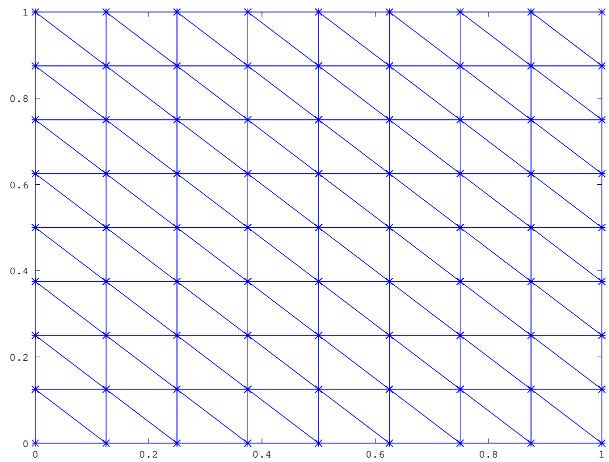

Figure 1: An $n \times n$ uniform grid of triangles for n

Note that for this example $n = 8+1 = 9$ is the amount of grid points situated at $y = 0$. Denote the maximum number of neighbors of a grid point with B.

a. Determine the maximum amount of connections respectively the maximum amount B of neighbors of a grid point.

To see the subsequent levelsets execute `bfs_levelset_demo(C,XYZ,1)`. We want to determine the amount of OPS (the complexity) $O(n^d)$ of `bfs_levelset()` as a function of n. You may assume that n is even.

b. Determine the exponent d of the complexity $O(n^d)$ of `bfs_levelset()` as a function of n: For each command determine the amount of associated OPS and calculate (estimate) the total amount to find d.

c. Determine the complexity $O(n^d)$ of `bfs_queue()` as a function of n.

We now verify the theoretical estimates in a practical manner: We time all methods:

d. For the sequence of grids for $k = 1, 2, 3, 4$, created by `square_triangular_grid(2^(2*k))`, $k = 1, \ldots, 4$, run the reorderings methods

```
bfs_levelset();
bfs_queue();
cm_levelset();
cm_queue();
```

```
cm_levelset_ic(); % levelset on column oriented C
cm_levelset_icg(); % levelset on column oriented C and Greedy
cmp_levelset();
cmp_levelset_icg();
```

(a `for`-loop is permitted). Visualize the 4×8 times in a multi-bar plot with `bar()` as in `graphs/timing.m`. Use a legend, lable the axes, etc.

e. Are there differences between the theoretical (predicted amount $O(\texttt{OPS})$ and the practice?

For those who want to experiment: This very fast script creates a list of random points in $[0, 1]^2$ connected by a triangular grid:

```
nb_points = 40; mx = 20;
XYZ = unique([randi([0,mx],[nb_points,1]),randi([0,mx],[nb_points,1])],'rows')/mx;
KLW = delaunay(XYZ(:,1),XYZ(:,2))
triplot(KLW,XYZ(:,1),XYZ(:,2));
```

Exercise 9. **Graphs**. **WARNING: `bfs_queue()` applied to ST5 can take up to two hours**. Goal: Write a very fast version of `bfs_levelset()`, called `bfs_greedy()`, which processes lists (level-sets) of un-processes new neighbors simultaneously. Make use of the statements you wrote in Exercise 4.

`matlab`: The new fast script should look as follows:

```
function [P] = bfs_levelset_greedy(C, V)
U = 1:size(C,1); P = []; U(V) = 0;
while ~isempty(V)
 P = [P, V]; L = V; V = [];
 % REPLACE FOR LOOP: for v = L
  ...
  ...
  ...
 % REPLACE FOR LOOP: end
end
function w = connected(C, v) % w connected v
% REPLACE COMMAND: w = find(C(v,:));
 ...
% REPLACE COMMAND: end
```

`python`: The file `reorder.py` contains examples `bfs_levelset()`, `bfs_queue()`. Your new fast script should look as follows:

```
def bfs_levelset_greedy(C, V):
 U = np.arange(0,np.size(C,1))
 P = np.array([],dtype=int)
 U[V] = -1;
```

```
while np.size(V) > 0:
 P = np.append(P,V);
 L = V;
 V = np.array([],dtype=int);
 % NO FOR LOOP: for v in L:
 ...
 ...
 ...
return P
```

a. Time `bfs_levelset()` vs `bfs_levelset_greedy()` for the connectivity matrices related to matrices **X** from `ST4` and `ST5`. Put all results together into one timing bar-chart.

   `matlab`: `load('ST4.mat')` and `load('ST5.mat')` both load matrix `X`.

   `python`: Load **X** for `ST4` and `ST5` from `ST4_IJV.txt`, `ST5_IJV.txt`:

```
import numpy as np
from scipy import sparse
X = np.fromfile('ST4_IJV.txt',dtype=float,sep=' ')
X = X.reshape((int(np.size(X,0)/3),3))
X = sparse.csc_matrix((X[:,2], (np.array(X[:,0],dtype=int), np.array(X[:,1],dtype=int))))
print(X)
```

b. For both matrices **X** from `ST4` and `ST5`, plot `spy(X(P1,P1))` next to `spy(X(P2,P2))`, where `P1` and `P2` are obtained by application of `bfs_queue()` resp. `bfs_greedy()`. What do you expect and are the results what you expect?

Matlab commands: `sparse()`, `size()`, `numel()` and `nnz()`:

```
A = sparse(3,4); A(3,3) = 1  % a sparse format 3 (rows) x 4 (columns) matrix
size(A)   % -> [3, 4]        % its dimensions n x m
size(A,2) % ->  4            % its 2-nd dim   m
numel(A)  % -> 12            % max. potentially nonzeros
nnz(A)    % -> 1             % actual amount of nonzeros
A(:)      % -> "(9,1) -> 1"  % returns       reshape(A,[numel(A),1])
v = 1:4   % -> v = [1,2,3,4] % a full format 1 x 4 matrix

A = diag(1:6,-1) % ->
A =
   0   0   0   0   0   0   0
   1   0   0   0   0   0   0
   0   2   0   0   0   0   0
   0   0   3   0   0   0   0
   0   0   0   4   0   0   0
   0   0   0   0   5   0   0
   0   0   0   0   0   6   0
A = spdiags((1:7)',-1,7,7); full(A) % ->
A =
   0   0   0   0   0   0   0
   1   0   0   0   0   0   0
   0   2   0   0   0   0   0
   0   0   3   0   0   0   0
   0   0   0   4   0   0   0
   0   0   0   0   5   0   0
   0   0   0   0   0   6   0
A = spones(A); full(A) % ->
   0   0   0   0   0   0   0
   1   0   0   0   0   0   0
   0   1   0   0   0   0   0
   0   0   1   0   0   0   0
   0   0   0   1   0   0   0
   0   0   0   0   1   0   0
   0   0   0   0   0   6   0
A = zeros(3,4) % ->
A =
   0   0   0   0
   0   0   0   0
   0   0   0   0
A = ones(3,4) % ->
A =
   1   1   1   1
   1   1   1   1
   1   1   1   1
```

`repmat()`, `repelem()`

```
A = repmat([4,3],[2,2]) % ->
A =
   4   3   4   3
   4   3   4   3
L = repelem([3,1,5],[2,3,4]) % ->
L =
   3   3   1   1   1   5   5   5   5
```

`ndgrid()`

```
[X,Y] = ndgrid(1:3,1:4) % ->
X =
   1   1   1   1
   2   2   2   2
   3   3   3   3
Y =
   1   2   3   4
   1   2   3   4
   1   2   3   4
```

`reshape()` – alter matrix dimensions while preserving memory layout:

```
reshape([1,2,3;4,5,6],[1,6]) % ->  [1, 4, 2,   5, 3, 6]  % 2x3 -> 1x6
reshape([1,2,3;4,5,6],[3,2]) % -> [[1; 4; 2], [5; 3; 6]] % 2x3 -> 3x2
```

`nonzeros(A)` – returns `V` obtained by `[I,J,V] = find(A)`:

`find()` – obtain nonzero entry information – for $\mathbf{A} = \begin{bmatrix} 0 & -1 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 4 \end{bmatrix}$:

1. `K = find(A)` returns their single-index `k` locations:

   ```
   K = [2;  4;  5;  9]
   ```

2. `[I,J,V] = find(A)` in addition returns their nonzero values:

   ```
   I = [2;   1;  2;  3]
   J = [1;   2;  2;  3]
   V = [2;  -1;  1;  4]
   ```

`all(v)` – returns boolean 1 if all entries of vector `v` are nonzero:

```
all([1,0,2,-1]) % -> 0
all([1,1,2,-1]) % -> 1
all([1,0;2,-1]) % -> [0, 1] % applied to n x matrix returns [all(A(:,1),all(A(:,2)),...,all(A(:,end))]
all([1,1;2,-1]) % -> [1, 1]
```

`accumarray()` – its inputs *must* be column-vector(s):

`accumarray`$([3;4;1;4;5;3;1;4],1)\%->[\underbrace{2}_{"\times 1}\underbrace{0}_{"\times 2};\underbrace{2}_{"\times 3};\underbrace{3}_{"\times 4};\underbrace{1}_{"\times 5}]$

`accumarray`$([1;2;3;2;4;1;3;1;4;2;4],[-1;0;1;2;3;4;5;6;7;8;9])\%->[\underbrace{-1+4+6}_{"1"};\underbrace{0+2+8}_{"2"};\underbrace{1+5}_{"3"};\underbrace{3+7+9}_{"4"}]$

`sort()` – sort vector entries/column entries for all matrix columns:

```
v  = [4,  1,  3,  1,  5,  3,  6,  2]; [s,iv] = sort(v); s, iv % ->
s  =  1   1   2   3   3   4   5   6 % sorted values default: ascending
iv =  2   4   8   3   6   1   5   7 % s == v(iv)
v  = [4,  1,  3,  1,  5,  3,  6,  2]; [s,iv] = sort(v,'descend'); s, iv % ->
s =   6   5   4   3   3   2   1   1 % sorted values: descending
iv =  7   5   1   3   6   8   2   4 % s == v(iv)
```

$$\mathrm{sort}(\begin{bmatrix} -2 & -8 & -7 & & & 1 \\ 1 & 2 & & -8 & & \\ -5 & & 1 & -7 & & \\ & & & & & \\ & 5 & & & 8 & -8 \\ & & & & & \end{bmatrix}) = \begin{bmatrix} -5 & -8 & -7 & -8 & & -8 \\ -2 & & & -7 & & \\ & & & & & \\ & & & & & \\ & 2 & & & & \\ 1 & 5 & 1 & & 8 & 1 \end{bmatrix} \quad \texttt{\%sort each column}$$

$$\mathrm{sortrows}(\begin{bmatrix} 2 & 4 \\ 1 & 8 \\ 2 & 2 \\ 1 & 2 \\ 3 & 4 \\ 2 & 7 \end{bmatrix}, [1,2]) = \begin{bmatrix} 1 & 2 \\ 1 & 8 \\ 2 & 2 \\ 2 & 4 \\ 2 & 7 \\ 3 & 4 \end{bmatrix}, \qquad \mathrm{sortrows}(\begin{bmatrix} 2 & 4 \\ 1 & 8 \\ 2 & 2 \\ 1 & 2 \\ 3 & 4 \\ 2 & 7 \end{bmatrix}, [2,1]) = \begin{bmatrix} 1 & 2 \\ 2 & 2 \\ 2 & 4 \\ 3 & 4 \\ 2 & 7 \\ 1 & 8 \end{bmatrix} \quad \texttt{\%sort inside columns } [1,2] \texttt{ resp. } [2,1]$$

`unique()` – returns a vector with the distinct entries of its input:

```
v = [6,8,3,6,7,8,6]; [u,iv,iu] = unique(v);          u,iv',iu' % ->
   u  = 3   6   7   8                    % the k=4 sorted unique entries
   iv = 3   1   5   2                    % their single-index locations
   iu = 2   4   1   2   3   4   2        % relabel all entries 1-to-1 into {1,....,k}
v(iv) % -> 3   6   7   8                 % u == v(iv)
u(iu) % -> 6   8   3   6   7   8   6     % u(iu) == v

v = [6,8,3,6,7,8,6]; [u,iv,iu] = unique(v,'stable'); u,iv',iu' % ->
  u  = 6   8   3   7                     % the k=4 FIRST ENCOUNTER unique entries
  iv = 1   2   3   5                     % their single-index locations
  iu = 1   2   3   1   4   2   1         % relabel all entries 1-to-1 into {1,....,k}
v(iv) % -> 6   8   3   7                 % u == v(iv)
u(iu) % -> 6   8   3   6   7   8   6     % u(iu) == v


v = [6,8,3,6,7,8,6]; [u,iv,iu] = unique(v,'first'); u,iv,iu % ->
  u  = 3   6   7   8                     % the k=4 sorted unique entries
  iv = 3   1   5   2                     % their single-index locations
  iu = 2   4   1   2   3   4   2         % relabel all entries 1-to-1 into {1,....,k}
v(iv) % -> 6   8   3   7                 % u == v(sort(iv))
u(iu) % -> 6   8   3   6   7   8   6     % u(iu) == v
v = [6,8,3,6,7,8,6]; [u,iv,iu] = unique(v,'last'); u,iv,iu % ->
  u  = 3   6   7   8                     % the k=4 sorted unique entries
  iv = 3   7   5   6                     % their single-index locations
  iu = 2   4   1   2   3   4   2         % relabel all entries 1-to-1 into {1,....,k}
v(iv) % -> 6   8   3   7                 %
u(iu) % -> 6   8   3   6   7   8   6     %
```

$$\mathrm{unique}(\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 1 & 2 \\ 5 & 6 \\ 2 & 4 \\ 1 & 2 \\ 3 & 4 \end{bmatrix}, '\mathrm{rows}') = \begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \qquad \texttt{\%return unique rows}$$

`rand()`, `randi()` and `randperm()` and inverse permutation:

```
A = rand(2, 4)           % -> full 2 x 4 matrix with entries stand. unif. distr. in [0,1]
A = sprand(2, 4, 15/100) % -> sparse 2 x 4 matrix with 15% nonzeros,
                         %    which are stand. unif. distr. in [0,1]
A = randi([-1,3],[2,4])  % -> 2 x 4 matrix with random integers from {-1,0,1,2,3}
```

```
% -> [3, 0, 1, -1; -1, 1, -1, 2]
p = randperm(8)          % -> 1 x 8 vector with a random permutation of {1,2,...,7,8}
% -> [5, 2, 6, 7, 1, 3, 8, 4]
p = randperm(8,5)        % -> 1 x 5 vector with 5 entries from randperm(8)
% -> [5, 3, 4, 2, 1]

p = [3,2,1,4]
pinv(p) = 1:4
p(pinv)              % [1, 2, 3, 4]
pinv(p)              % [1, 2, 3, 4]
```

`sum()`, `cumsum()`, `prod()`, `cumprod()` and `diff()`:

```
v = 1:6              % -> v = [1, 2, 3, 4, 5, 6]
sum(v)               % -> 21
cumsum(v)            % -> [1, 3, 6, 10, 15, 21]
prod(v)              % -> 720
cumprod(v)           % -> [1, 2, 6, 24, 120, 720]
diff(v)              % -> [1, 1, 1, 1, 1]
diff(cumsum([1, v])) % -> [1, 2, 3, 4, 5, 6]
```

`min()`, `max()`:

```
[mn,loc] = min(v)    % -> mn = 1; loc = 1
[mn,loc] = max(v)    % -> mn = 6; loc = 6
w = 6:-1:1           % -> w = [6, 5, 4, 3, 2, 1]
u = min(v,w)         % -> u = [1, 2, 3, 3, 2, 1]
```

`char()`, `double()`:

```
double('a')          % -> 97
char(97)             % -> 'a'
'c' - 'a'            % -> 2
```

Python commands: `sparse()`, `size()`, `shape()` and `count_nonzero()`:

```
import numpy as np
from numpy import matlib
from scipy import sparse
np.set_printoptions(threshold=np.inf) # print all entries

A = sparse.csc_matrix((3,4)); A[2,2] = 1        # sparse 3 x 4 matrix
np.shape(A)     # -> (3, 4)                     # its dimensions n x m
np.size(A,1)    # ->  4                         # its 2-nd dim   m
np.size(A)      # ->  1                         # actual amount of nonzeros
sparse.csc_matrix.count_nonzero(A) # -> 1       # actual amount of nonzeros
A = sparse.random(8, 8, 0.7, format='csr');     # sparse random

# replace all non-zeros by 1
A.data.fill(1)

A = np.random.randint(low=1,high=8, size=(8,8)) # full 8 x 8 matrix
np.reshape(A,(8*8))                             # reshaped to array
np.reshape(A,(2,32))                            # reshaped to matrix
v = np.arange(1,5) # -> v = [1,2,3,4]           # 1 x 4 array

A = sparse.spdiags(np.arange(1,8),-1,7,7)       # sparse 7 x 7 sub-diagonal
A = np.zeros(shape=(3,4))                       # full zeros
A = np.ones(shape=(3,4))                        # full ones
```

`repmat()`, `repelem()`:

```
matlib.repmat(np.array([4, 3]), 2, 2)          # -> array([[4, 3, 4, 3], [4, 3, 4, 3]])
np.repeat(np.array([3,1,5]),np.array([2,3,4])) # -> array([3, 3, 1, 1, 1, 5, 5, 5, 5])
```

`meshgrid()`:

```
np.meshgrid(np.array([1,2,3]),np.array([1,2,3,4])) # ->
#  [array([[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]),
#   array([[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]])]
```

`reshape()`:

```
np.reshape(np.array([[1,2,3],[4,5,6]]),(1,6)) # -> array([[1, 2, 3, 4, 5, 6]])
np.reshape(np.array([[1,2,3],[4,5,6]]),(3,2)) # -> array([[1, 2], [3, 4], [5, 6]])
```

`nonzero()` and `find()`:

```
A = sparse.csc_matrix(np.array([[0,-1,0],[2,1,0],[0,0,4]]))
I,J,V = sparse.find(A)                                     # -> array I, J, V
V = A[sparse.csc_matrix.nonzero(A)]                        # -> full V
V = np.squeeze(np.array(A[sparse.csc_matrix.nonzero(A)])) # -> array V

I,J,V = sparse.find(sparse.csc_matrix(np.array([[0,-1,0],[2,1,0],[0,0,4]])))
# -> I = [0 1 1 2]
# -> J = [1 0 1 2]
# -> V = [-1  2  1  4]

# find example: restrict to lower triangular part:
n = 9; k = 4
A = sparse.random(n, n, 0.7, format='csr'); print(A.toarray())
I,J,V = sparse.find(A)
S = J - I <= 0
Y = sparse.csc_matrix((V[S],(I[S],J[S])), shape=(np.size(A,0), np.size(A,1)))
```

`all()`:

```
np.all(np.array( [1,0,  2,-1]))  # -> False
np.all(np.array( [1,1,  2,-1]))  # -> True
np.all(np.array([[1,0],[2,-1]])) # -> False
np.all(np.array([[1,1],[2,-1]])) # -> True
np.all(np.array([[1,0],[2,-1]]),axis=0) # -> [False,False]
np.all(np.array([[1,1],[2,-1]]),axis=0) # -> [True,True]
```

`accumarray()`:

```
def accumarray(I,V,sze=None,fmt='full'):
 if sze is None: sze=max(I)+1
 if fmt == 'full':
  A = np.squeeze(np.array(sparse.csc_matrix.todense(sparse.csc_matrix(sparse.coo_matrix((V, (np.zeros(np.size(I),dtype=
 else:
  A = sparse.csc_matrix(sparse.coo_matrix((V, (np.zeros(np.size(I),dtype=int),I)), shape=(1,sze)))
 return A

I = np.array([0, 2, 1, 2, 1, 1])
V = np.array([1, 1, 1, 1, 2, 2])
A = accumarray(I,V,fmt='sparse')        # sparse [1 5 2]
```

```
A = accumarray(I,V)                     # full   [1 5 2]
A = accumarray(I,V,sze=7,fmt='sparse')  # sparse [1 5 2 0 0 0 0]
A = accumarray(I,V,sze=7)               # full   [1 5 2 0 0 0 0]
A = accumarray(np.array([3,4,1,4,5,3,1,4])-1,np.array([1,1,1,1,1,1,1,1])) # -> full [2, 0, 2, 3, 1]
A = accumarray(np.array([1,2,3,2,4,1,3,1,4,2,4])-1,np.array([-1,0,1,2,3,4,5,6,7,8,9])) # -> full [ 9, 10,  6, 19]
```

sort():

```
# sort offers 'stable' (useless), option 'ascend' is not implemented, permutation needs 2-nd call
v  = np.array([4,  1,  3,  1,  5,  3,  6,  2]); # -> array([4, 1, 3, 1, 5, 3, 6, 2])
s = np.sort(v,kind='heapsort')                  # -> array([1, 1, 2, 3, 3, 4, 5, 6])
iis = np.argsort(v,kind='heapsort')             # -> array([3, 1, 7, 2, 5, 0, 4, 6])

# sort on a sparse matrix: is not implemented
A = sparse.csc_matrix((6,6))
A[0,0] = -2; A[0,1] = -8; A[0,3] = -7; A[0,5] = 1; A[1,0] = 1; A[1,1] = 2; A[1,3] = -8;
A[2,0] = -5; A[2,2] = 1; A[2,3] = -7; A[4,1] = 5; A[4,4] = 8; A[4,5] = -8;

# row sort DOES NOT EXIST:
A = np.array([[2,4],[1,8],[2,2],[1,2],[3,4],[2,7]])
V = np.sort(A,kind='heapsort',axis=0) # ->
#  array([[1, 2], [1, 2], [2, 4], [2, 4], [2, 7], [3, 8]]) % not row sort
V = np.sort(A,kind='heapsort',axis=1) # ->
#  array([[2, 4], [1, 8], [2, 2], [1, 2], [3, 4], [2, 7]]) % not row sort
```

unique():

```
# option 'stable' (first in u are the first encounts) not implemented:
v = np.array([6,8,3,6,7,8,6]);
u,iv,iu = np.unique(v,return_index=True,return_inverse=True) # ->
#  u = array([3, 6, 7, 8])
#  iv = array([2, 0, 4, 1])
#  iu = array([1, 3, 0, 1, 2, 3, 1])
v[iv] # -> array([3, 6, 7, 8])
u[iu] # -> array([6, 8, 3, 6, 7, 8, 6])

# this function provides option 'stable':
def uniquestable(v) :
 p= np.argsort(iv)
 us= u[p]; ivs = iv[p]
 pinv = np.empty(np.size(p),dtype=int)
 pinv[p] = np.arange(0,np.size(p))
 ius = pinv[iu]
 return us, ivs, ius

# test stable unique:
v = np.array([6,8,3,6,7,8,6])
u,iv,iu = uniquestable(v)
#  u = array([6, 8, 3, 7])
#  iv = array([0, 1, 2, 4])
#  iu = array([0, 1, 2, 0, 3, 1, 0])
v[iv] # -> array([3, 6, 7, 8])
u[iu] # -> array([6, 8, 3, 6, 7, 8, 6])

# unique for rows DOES EXIST -- but of course deletes the doubles:
A = np.array([[2,4],[1,8],[2,2],[1,2],[3,4],[2,7]])
U = np.unique(A,axis=0) # ->
#  array([[1, 2], [1, 8], [2, 2], [2, 4], [2, 7], [3, 4]])
```

random(), inverse permutation:

```
# random vector, matrix
A = np.random.randint(low=-10,high=10,size=(8, 8))  # random full 8 x 8
A = np.random.randint(low=-10,high=10,size=10)      # random permutation/array
A = np.random.randint(low=-10,high=10,size=10)[0:6] # random permutation/array part
A = A[0:5]                                          # 5 random out of 10
A = sparse.random(8, 8, 0.7, format='csr');         # sparse random

# inverse permutation
p    = np.random.permutation(10)
pinv = np.empty(10,dtype=int)
pinv[p] = np.arange(0,10);
p[pinv] # -> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
pinv[p] # -> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

stack(), hstack(), vstack():

```
# stack compatible matrices
A = np.arange(1,4)
B = np.arange(4,7)
np.stack((A, B))         # -> A   on top of B   (matlab [A,B])
np.stack((A, B), axis=1) # -> A^T left   of B^T (matlab [A';B'])
np.hstack((A, B))        # -> A   left   of B   (matlab [A;B])
np.vstack((A, B))        # -> A   on top of B   (matlab [A,B])
```

sum(), cumsum(), prod(), cumprod(), diff():

```
# sum, cumsum(), prod(), cumprod() and diff()
v = np.arange(1,7)                               # v = [1, 2, 3, 4, 5, 6]
np.sum(v)                                        # 21
np.cumsum(v)                                     # [1, 3, 6, 10, 15, 21]
np.prod(v)                                       # 720
np.cumprod(v)                                    # [1, 2, 6, 24, 120, 720]
np.diff(v)                                       # [1, 1, 1, 1, 1]
np.diff(np.cumsum(np.hstack((np.array([1]),v)))) # [1, 2, 3, 4, 5, 6]
```

min(), minimum():

```
w = 7 - np.arange(1,7)                           # [6, 5, 4, 3, 2, 1]
mn = min(w)                                      # 1     # no location information
u = np.minimum(v,w)                              # [1, 2, 3, 3, 2, 1]
```

char(), ord():

```
ord('a')                                         # -> 97
char(97)                                         # -> 'a'
ord('c') - ord('a')                              # -> 2
```

Linear index:

```
A = np.zeros([3,3]); K = [0,4,8]; A[np.unravel_index(K, A.shape)] = K;
 # -> A = [[0,0,0],[0,4,0],[0,0,8]]
```

auxiliary:

```
# matrix parts: let k be the number of the diagonal: works:
# for full A:
A = np.random.randint(low=-10,high=10,size=(8, 8)) # random full 8 x 8
np.tril(A,k)
np.triu(A,k)
np.diag(A,k)
# for sparse A:
A = sparse.random(8, 8, 0.7, format='csr');     # sparse random
sparse.tril(A,k)
sparse.triu(A,k)
#sparse.diag(A,k) # is not implemented
# for sparse A:
A.diagonal() # selects main diagonal, can not select other diagonals
sparse.triu(sparse.tril(A,k),k) # selects MATRIX with only diagonal k but is slow
```

Enjoy and Good luck!