Sparse linear systems

Sparse matrices

Techniques of storage

Creating sparse matrices in python

Algorithms for sparse matrices in python

- elementary matrix operations

- decomposition and other methods

Arnoldi iteration

If most of the elements in a matrix are zero, then it makes no sense to store them.
In addition, a lot of the matrix operations will be elementary.

Dense Matrix

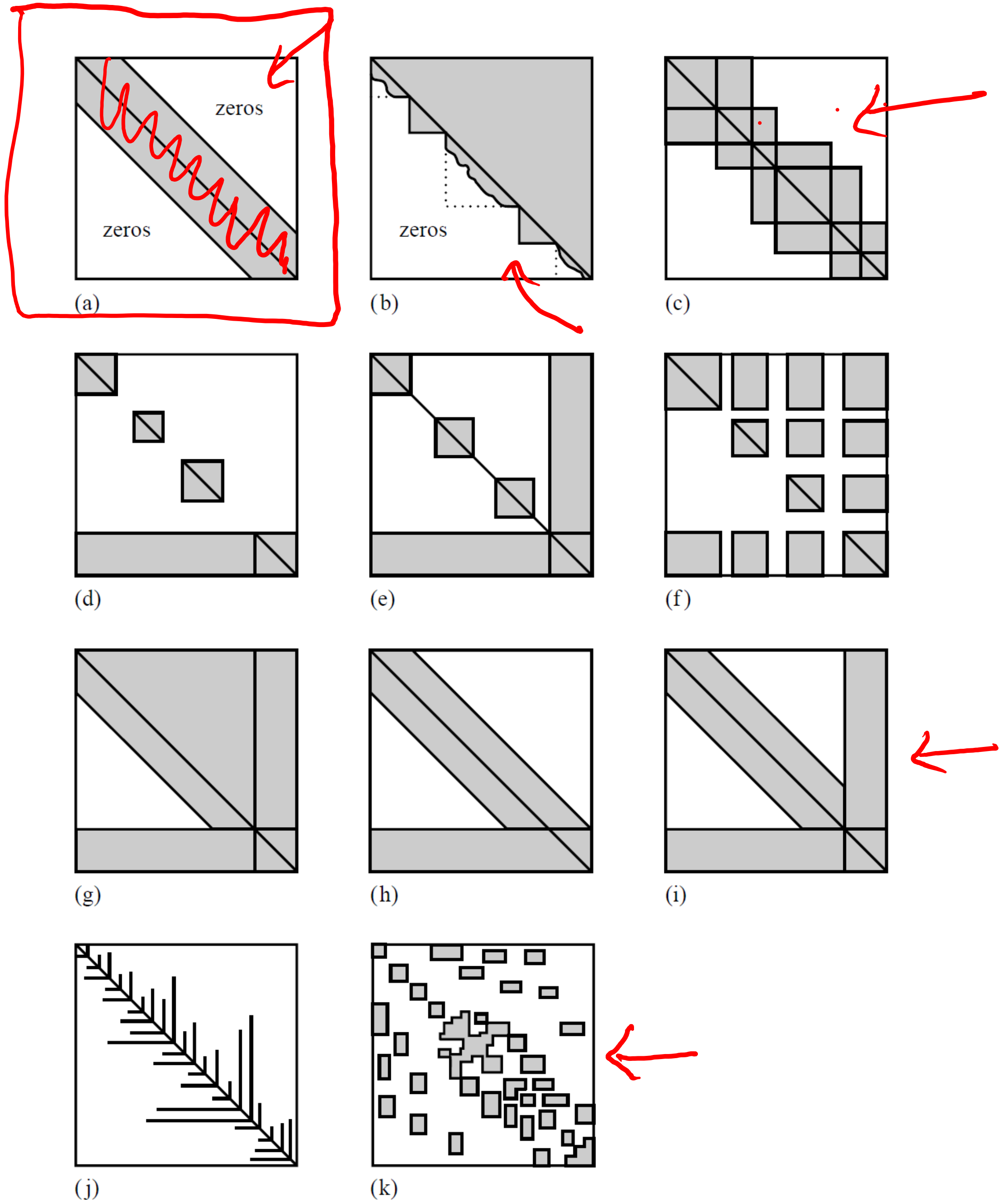
1	2	31	2	9	7	34	22	11	5
11	92	4	3	2	2	3	3	2	1
3	9	13	8	21	17	4	2	1	4
8	32	1	2	34	18	7	78	10	7
9	22	3	9	8	71	12	22	17	3
13	21	21	9	2	47	1	81	21	9
21	12	53	12	91	24	81	8	91	2
61	8	33	82	19	87	16	3	1	55
54	4	78	24	18	11	4	2	99	5
13	22	32	42	9	15	9	22	1	21

Sparse Matrix

1	0	3	0	9	0	3	0	0	0
11	0	4	0	0	0	0	0	2	1
0	0	1	0	0	0	4	0	1	0
8	0	0	0	3	1	0	0	0	0
0	0	0	9	0	0	1	0	17	0
13	21	.	9	2	47	1	81	21	9
.	.	.	.	.	.	.	.	.	.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11	.	.	.	.
.	.	2	.	.	.	.	22	.	21

Instead of storing the *full* matrix, we store only the *non-zero elements.*

From https://cmdlinetips.com/2018/03/sparse-matrices-in-python-with-scipy/


zeros
zeros
zeros
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)

Sparse matrices can be stored using different protocols:

coo_matrix: **COO**rdinate format matrix

csc_matrix: **C**ompressed **S**parse **C**olumn matrix

csr_matrix: **C**ompressed **S**parse **R**ow matrix

bsr_matrix: **B**lock **S**parse **R**ow matrix

dia_matrix: Sparse matrix with **DIA**gonal storage

dok_matrix: **D**ictionary **O**f **K**eys based sparse matrix.

lil_matrix: Row-based linked list sparse matrix

Different protocols are more efficient for different algorithms.
We will discuss two of these: the COO format and the CSC format

The Coordinate format matrix format (COO)

In this storage protocol, the indices are stored as a double-entry list,
And the elements are stored as a list of the same length. E.g.

0th

Matrix

0th

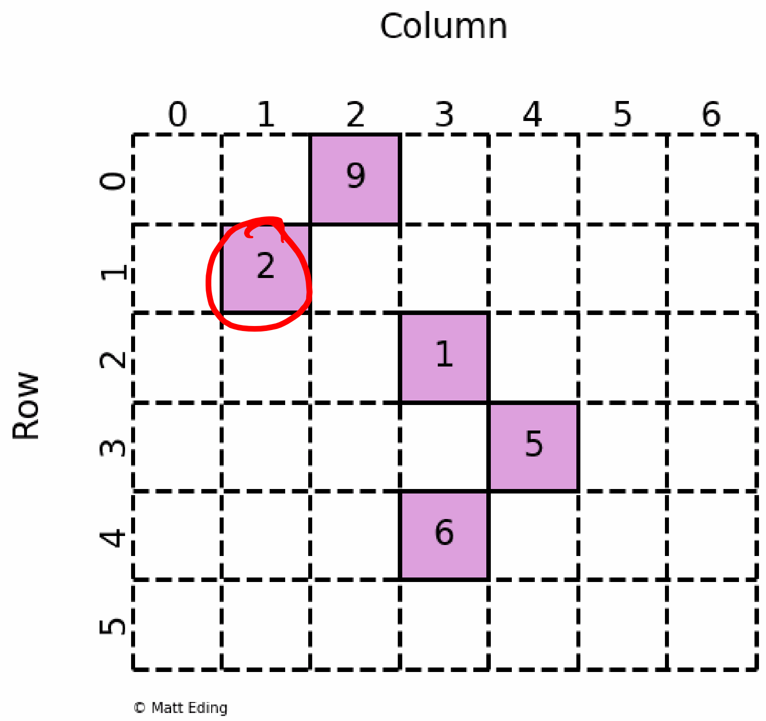
1	.	3	.	9	.	3	.	.	.
11	.	4	.	.	.	.	.	2	1
.	.	1	.	.	.	4	.	1	.
8	.	.	.	3	1	.	.	.	.
.	.	.	9	.	.	1	.	17	.
13	21	.	9	2	47	1	81	21	9
.	.	.	.	.	.	.	.	.	.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11	.	.	.	.
.	.	2	.	.	.	.	22	.	21

How this is stored

row column

(0,0) 1
(0,2) 3
(0,4) 9
…

The COO is a natural way to think about sparse matrices, but is not efficient for matrix operations.



COO

Row

1	3	0	2	4

Column

1	4	2	3	3

Data

2	5	9	1	6

https://matteding.github.io/2019/04/25/sparse-matrices/

Compressed **S**parse **C**olumn matrix (CSC) format

In this protocol the non-zero entries are stored in the following way:

Store the Data in an array going down the columns and removing the zeros
Store the Row index of each element in the Data array
Create an array where adjacent pairs give *slices* into the Data array.

Matrix

0th	**1**	**0**	**2**
1st	0	0	3
2nd	4	5	6

How this is stored

1st column   2nd   3rd column.

Data: [1 , 4 , 5 , 2 , 3 , 6]

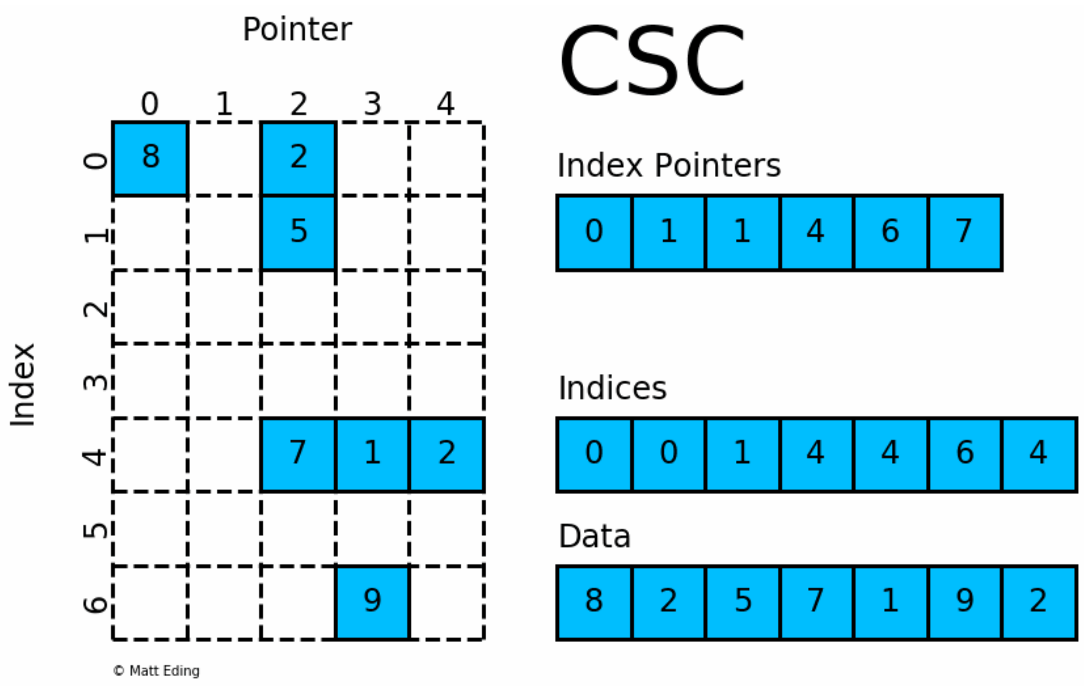
Row indices: [0 , 2 , 2 , 0 , 1 , 2]

Index pointers: [0 , 2 , 3 , 6]

2:3 → 2,2

0:2   3:6 → 3,4,5

0,1



row

https://matteding.github.io/2019/04/25/sparse-matrices/

Which format to use? Two things to keep in mind:

1. Some formats are more efficient than others for certain operations.

<u>We *recommend sticking to CSC or CSR formats for linear algebra*</u>

2. It really doesn't matter – python will let you know if you start using an inefficient matrix format.

You can convert between formats using

A.tocsc()

A.tocoo()

.etc.

The Sparse Matrix modules in python

We will be relying on the *sparse* module from scipy, which itself contains a *sparse linear algebra* module

```
import numpy as np
from scipy import sparse as sp
from scipy.sparse import linalg as sla
```

Note: the sparse "linalg" module is *different* from the regular scipy linalg module.

Sparse matrices in python are stored in their own data type, according to the compression protocol (COO, CSC, etc)

```
   ...: import numpy as np
   ...: from scipy import sparse as sp
   ...: from scipy.sparse import linalg as sla
   ...:
   ...: row = np.array([0,0,1,1])
   ...: col = np.array([0,1,0,1])
   ...:
   ...: entries = np.array([2.,2.,2.,2])
   ...:
   ...: A = sp.csc_matrix((entries,(row,col)))

In [3]: A
Out[3]:
<2x2 sparse matrix of type '<class 'numpy.float64'>'
    with 4 stored elements in Compressed Sparse Column format>

In [4]: print(A)
  (0, 0)    2.0
  (1, 0)    2.0
  (0, 1)    2.0
  (1, 1)    2.0
```

Building basic sparse matrices in python

Sparse matrices can be constructed directly using

- the *type* of sparse matrix you want
- a list of row indices
- a list of column indices
- a list of the data entries

A = sp.csc_matrix((entries,(row,col)))

create a csc matrix

brackets

```
   ...: import numpy as np
   ...: from scipy import sparse as sp
   ...: from scipy.sparse import linalg as sla
   ...:
   ...: row = np.array([0,0,1,1])
   ...: col = np.array([0,1,0,1])
   ...:
   ...: entries = np.array([2.,2.,2.,2])
   ...:
   ...: A = sp.csc_matrix((entries,(row,col)))

In [3]: A
Out[3]:
<2x2 sparse matrix of type '<class 'numpy.float64'>'
	with 4 stored elements in Compressed Sparse Column format>

In [4]: print(A)
  (0, 0)	2.0
  (1, 0)	2.0
  (0, 1)	2.0
  (1, 1)	2.0
```

$$\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

Larger sparse matrices can be defined using the “shape” option

A = sp.csc_matrix((entries,(row,col)),shape= (M,N))

```
   ...: A3 = sp.csc_matrix((entries,(row,col)),shape=(4,4))

In [24]: A3
Out[24]:
<4x4 sparse matrix of type '<class 'numpy.float64'>'
    with 4 stored elements in Compressed Sparse Column format>

In [25]: A3.A
Out[25]:
array([[2., 2., 0., 0.],
       [2., 2., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

Matrices of different data types can be defined using the “dtype” option

```
   ...: A3 = sp.csc_matrix((entries,(row,col)),dtype = int,shape=(4,4))

In [27]: A3
Out[27]:
<4x4 sparse matrix of type '<class 'numpy.intc'>'
    with 4 stored elements in Compressed Sparse Column format>

In [28]: A3.A
Out[28]:
array([[2, 2, 0, 0],
       [2, 2, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int32)
```

Sparse matrices can be converted to dense matrices in two ways:

1. Using the .todense() or .A methods

2. Automatically as the result of a matrix operation that produces a dense matrix

```
In [18]: print(A)
  (0, 0)    2.0
  (1, 0)    2.0
  (0, 1)    2.0
  (1, 1)    2.0

In [19]: print(A.todense())
[[2. 2.]
 [2. 2.]]

In [20]: print(A.A)
[[2. 2.]
 [2. 2.]]
```

```
In [30]: b = np.array([[1],[2]])

In [31]: A @ b
Out[31]:
array([[6.],
       [6.]])
```

NB: when this happens be careful! Some of your Sparse routines will not work on regular matrices.

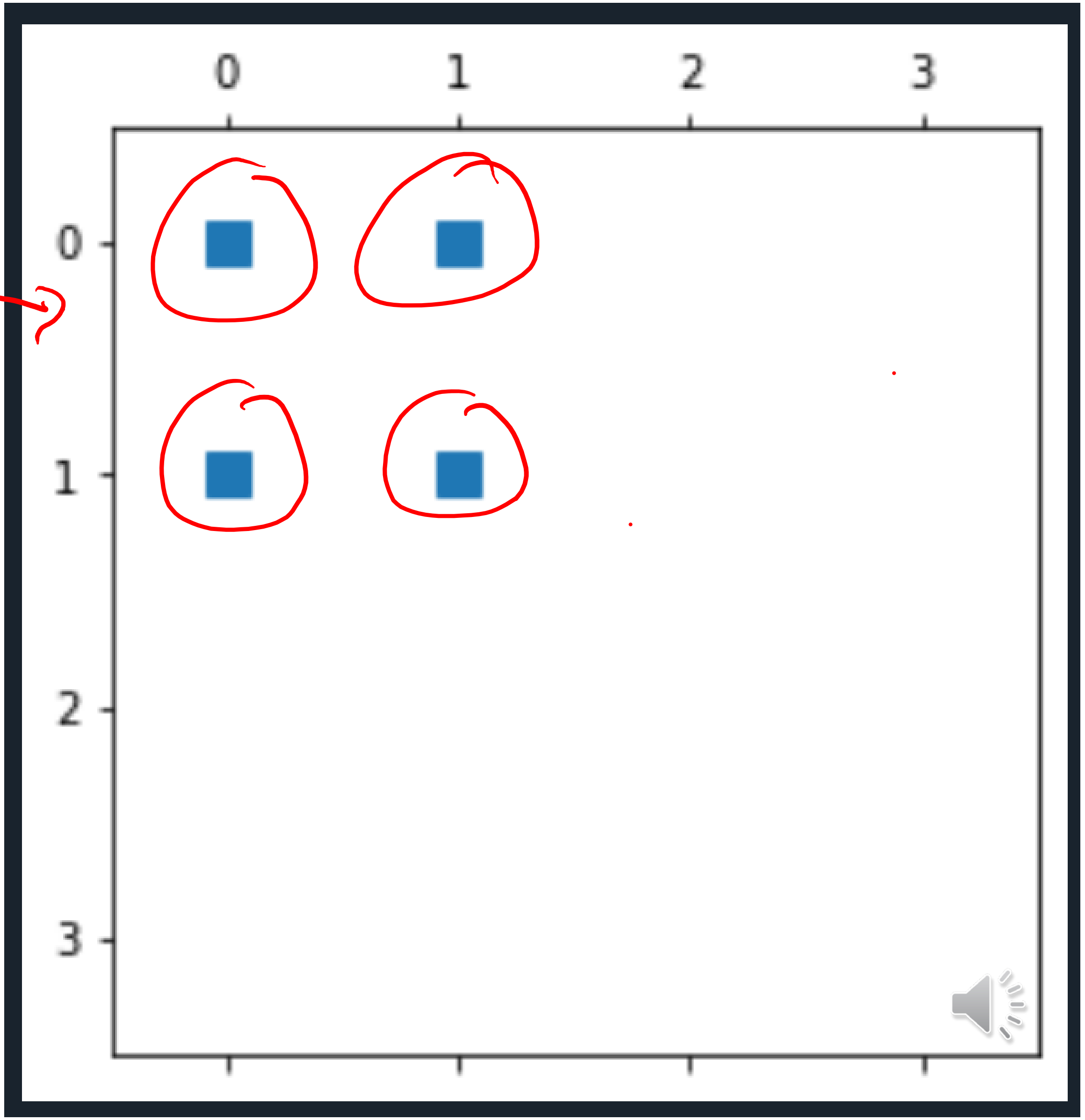
You can convert a dense matrix to a sparse matrix using
The sp.[…]_matrix() function

csc_matrix

```
In [65]: AD = np.diag([1,1,1,2,2,2])

In [66]: print(AD)
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 2 0 0]
 [0 0 0 0 2 0]
 [0 0 0 0 0 2]]

In [67]: AS = sp.csc_matrix(AD)

In [68]: print(AS)
  (0, 0)    1
  (1, 1)    1
  (2, 2)    1
  (3, 3)    2
  (4, 4)    2
  (5, 5)    2
```

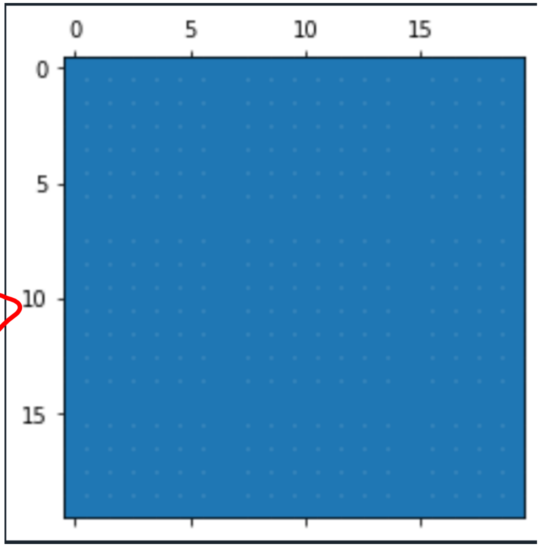
Sparse matrices can be visualised using the "spy()" function in conjunction with matplotlib.

```
In [37]: A3.A
Out[37]:
array([[2, 2, 0, 0],
       [2, 2, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int32)

In [38]: plt.spy(A3)
Out[38]: <matplotlib.lines.Line2D at 0x1c75337da00>
```



Sparsify-ing matrices

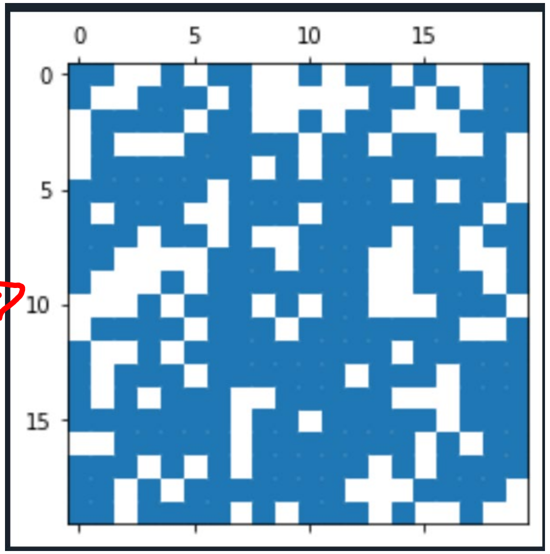
Often in a real situation a matrix will have elements which are close to zero without being sparse. A useful Technique is to "sparsify" the matrix by setting all the small elements to exactly zero.



Python offers a simple way to do this:

condition

```
In [60]: AD[abs(AD)<0.1] = 0.

In [61]: A = sp.csc_matrix(AD)
```



<u>Building more complicated sparse matrices in python</u> – the diags function

When building a sparse matrix yourself you usually have to specify the elements either along the main diagonal, or along the upper or lower diagonals.

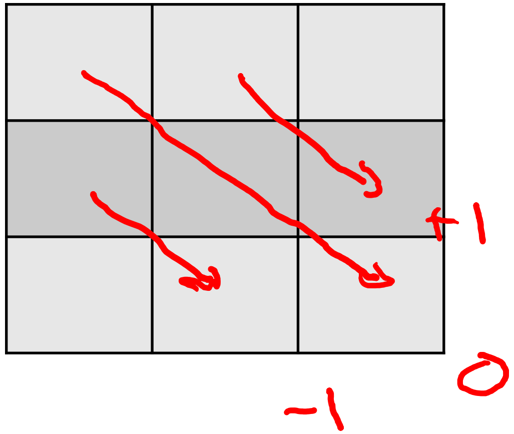
This can be done using the diags function. There are two approaches:

1. Direct definition:

[1,2,1] [3,4] 0 1

A = sp.diags([[elements on 1st diagonal],[elements on 2nd diagonal]],…],[index of first diagonal, index of 2nd diagonal,…])

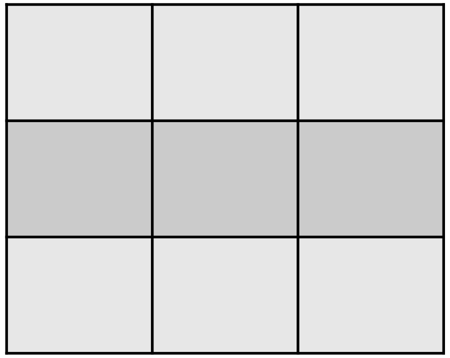
```
In [113]: G = sp.diags([[1,2,1],[3,4]],[0,1])

In [114]: G.A
Out[114]:
array([[1., 3., 0.],
       [0., 2., 4.],
       [0., 0., 1.]])
```



2. Broadcasting (assign all elements to a single number)

A = sp.diags([element 1,element 2, ...],[0,1,2,...],shape=(N,N))

```
In [117]: G = sp.diags([1,2],[0,1],shape=(3,3))

In [118]: G.A
Out[118]:
array([[1., 2., 0.],
       [0., 1., 2.],
       [0., 0., 1.]])
```



Operations with sparse matrices

All the "usual" matrix operations work with sparse matrices.

E.g. Matrix multiplication:

```
In [83]: AD = np.diag([1,1,1,2,2,2])

In [84]: BD = np.array([[1],[0],[0],[-1],[2],[0]])

In [85]: AS = sp.csc_matrix(AD)

In [86]: bs = sp.csc_matrix(BD)

In [87]: cs = AS @ bs

In [88]: print(cs)
  (4, 0)    4
  (3, 0)    -2
  (0, 0)    1
```

```
In [89]: cs2 = AS.dot(bs)

In [90]: print(cs2)
  (4, 0)    4
  (3, 0)    -2
  (0, 0)    1
```

Note that multiplication will convert a sparse matrix to a normal one if it becomes non-sparse.

Also Note: The norm function in scipy only computes the L_1 and L_∞ norm!

Decomposition of Sparse matrices

The LU decomposition can be done using the sparse.splu() function
In the scipy.sparse.linalg module, and returns an L and U packaged together:

```
In [109]: LU = sla.splu(AS)

In [110]: LU.L.A
Out[110]:
array([[ 1.        ,  0.        ,  0.        ,  0.        ],
       [-0.15129968,  1.        ,  0.        ,  0.        ],
       [-0.18137319,  0.60793442,  1.        ,  0.        ],
       [-0.33503656, -0.43754454,  0.52032285,  1.        ]])

In [111]: LU.U.A
Out[111]:
array([[ 1.54042516, -0.63969771, -0.27747472,  0.54167411],
       [ 0.        ,  1.23102453, -0.56414286, -0.50376781],
       [ 0.        ,  0.        ,  0.83797449,  1.17092558],
       [ 0.        ,  0.        ,  0.        , -0.5762109 ]])
```

L = LU.L

U = LU.U

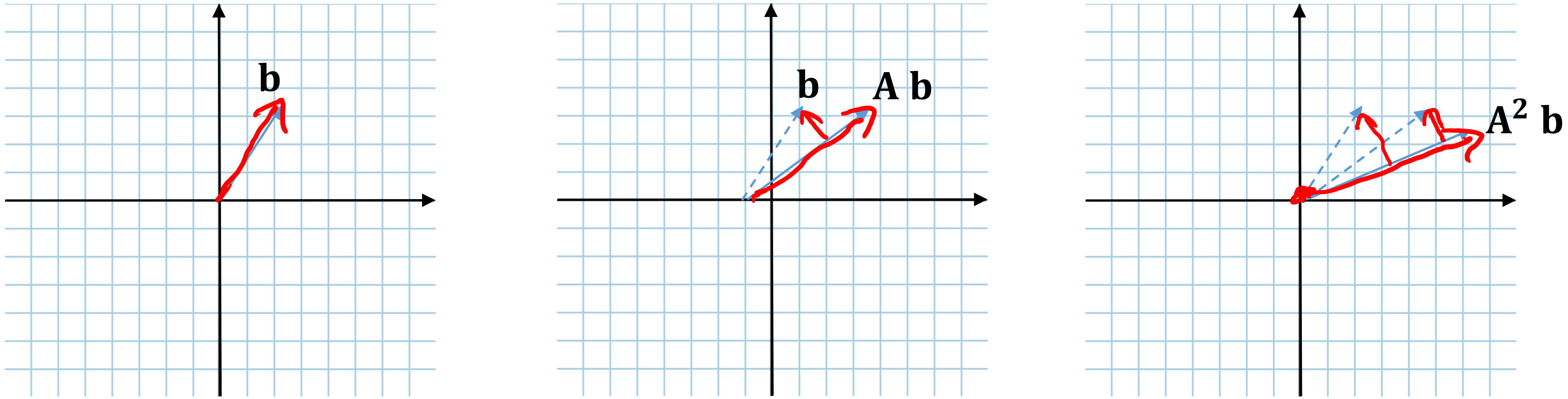
Note: There is currently no
official sparse QR decomposition for python!

<u>Arnoldi iteration</u>

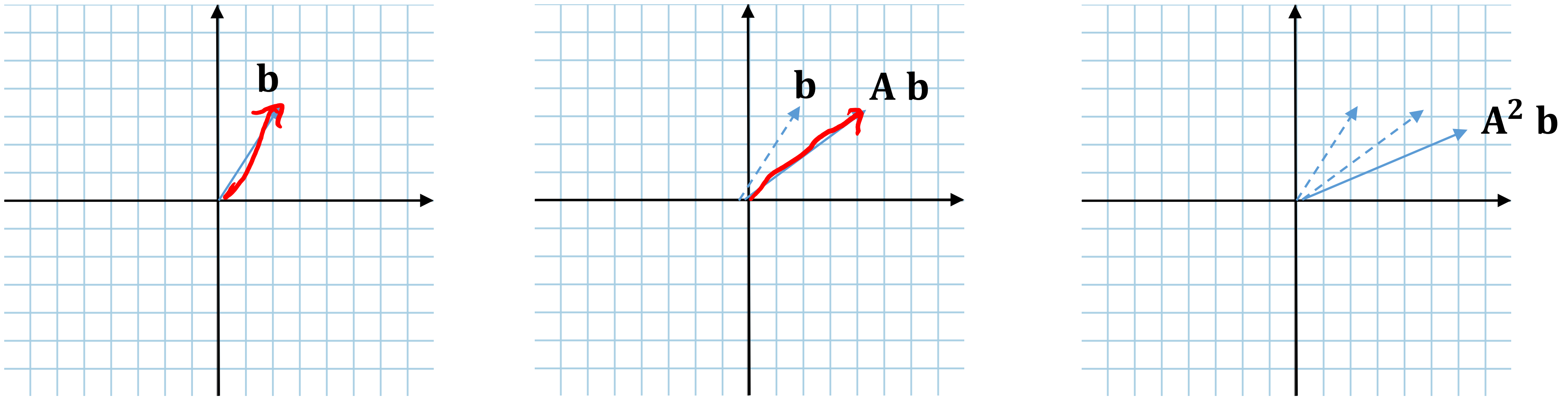
Arnoldi iteration is a powerful and stable method for computing the eigenvalues of large matrices.
It is especially suitable for sparse systems.

$$\boxed{\mathbf{A}\mathbf{v} = \lambda \mathbf{v}}$$

Recall in *Power Iteration,* successive applications of A bring any vector toward the eigenvector.



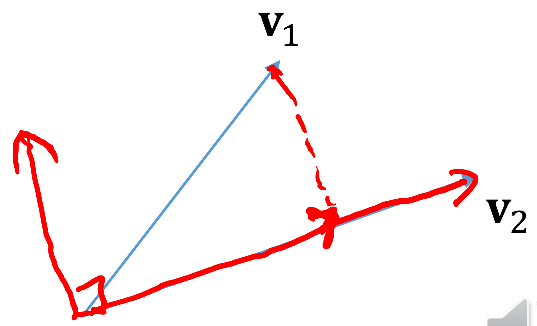
The idea of Arnoldi iteration is to *keep* the information from each iteration,
and use it to construct an orthogonal basis that can be used to compute the eigenvalues.



For each iteration we keep the resulting vector into a matrix, known as the Krylov matrix

$$K_n = \begin{bmatrix} \vdots & \vdots & \vdots & & \vdots \\ b & Ab & A^2 b & \cdots & A^{n-1} b \\ \vdots & \vdots & \vdots & & \vdots \end{bmatrix}$$

q_0 q_1



From each column we construct an *orthonormal set of vectors* $\mathbf{q}_j$ using Gramm-Schmidt orthogonalization. This is known as the Krylov subspace.

We then re-write the matrix A in terms of these new basis vectors $\mathbf{q}_j$

$$H = Q^* A\, Q \qquad \begin{bmatrix} | & | & | & \\ q_0 & q_1 & q_2 & \cdots \\ | & | & | & \end{bmatrix}$$

Because the $\mathbf{q}_j$s are orthogonal, the matrix H is *almost upper triangular, and will also have the same eigenvalues as A.*

We can then use QR-decomposition to compute the eigenvalues extremely quickly and efficiently.

Arnoldi Iteration

1. Start with a vector q_0 (preferably randomized)

2. Compute

$$q_k = Aq_{k-1}$$

3. Loop over j from 1 to k-1

Compute $\quad q_k := q_k - (q_j^* q_k) q_k$

Normalise $\quad q_{k+1} := \dfrac{q_k}{||q_k||_\infty}$

4. Repeat from step 2 as many times as you like

5. Do QR decomposition on the matrix ~~A~~ H to find the eigenvalues

Other Important Sparse approaches for solving the eigenvalue problem:

- Lanczos algorithm:
  This is the Arnoldi iteration applied to symmetric matrices

- Generalised minimal Residual method (GMRES)
  Uses Arnoldi iteration to compute the eigenvalues of an arbitrary non-symmetric matrix

- Conjugate gradient squared algorithm
  Applies conjugate gradients to solve linear systems