MAA507, Power method, QR-method and sparse matrix representation.

Christopher Engström

February 11, 2014

Lecture 7: Overview

Today we will look at:

- The Power method.
- QR-method.
- If time: A look at sparse matrix representation and "fill in".

Why do we need numerical methods?

I think everyone have seen how time consuming it is to find the eigenvalues and eigenvectors analytically by hand.

- In the same way using Gauss-elimination or the characteristic polynomial is computationally slow and not very useful in many real-world systems where we often work with very large matrices.
- We will look at some different methods for different kinds of matrices or depending on if you need all eigenvalues or if it's enough with one.

Why do we need numerical methods?

If we need to find eigenvalues or eigenvectors of a matrix we should first ask ourselves:

- Do we already know something about the eigenvalues or eigenvectors we can use? Maybe we know some eigenvalues or have a good approximation already.
- Do we seek only eigenvalues or eigenvectors? Do we need all or only one or a few eigenvalues/eigenvectors?
- Do we have a small or moderately sized matrix or do we have a large or very large matrix requiring specialized methods?
- Do we have a full matrix (few zero elements) or sparse matrix with mostly zeros.
- Do our matrix have any useful property or structure we could use? such as being positive definite or a bandmatrix.

Power method

The Power method is a very simple algorithm for finding the eigenvalue λ_1 with the largest absolute value and it's corresponding eigenvector.

- The method works by iterating $\mathbf{x}_{k+1} = \frac{A\mathbf{x}_k}{||A\mathbf{x}_k||}$.
- Where we assume there is only 1 eigenvalue on the spectral radius and that A is diagonalizable.
- If the eigenvalue with largest absolute value $\lambda_1 \neq 0$ then $\frac{A\mathbf{x}_k}{||A\mathbf{x}_k||}$ approaches a vector parallel to the eigenvector of λ_1.
- We get the eigenvalue λ_1 from the relation $A^k\mathbf{x} \approx \lambda_1 A^{k-1}\mathbf{x}$ when k is large.

Power method

We give a short example looking at the following matrix:

- $$A = \begin{bmatrix} -2 & 1 & 1 \\ 3 & -2 & 0 \\ 1 & 3 & 1 \end{bmatrix}$$
- We start with $\mathbf{x} = [1\ 1\ 1]^\top$ and iterate:

Power method

We give a short example looking at the following matrix:

- $$A\mathbf{x} = \begin{bmatrix} -2 & 1 & 1 \\ 3 & -2 & 0 \\ 1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 5 \end{bmatrix}$$
- $||A\mathbf{x}|| = \sqrt{26}$
- $\mathbf{x}_1 = \dfrac{A\mathbf{x}}{||A\mathbf{x}||} = [0, 1, 5]/\sqrt{26}$

Power method

Continuing for the next iteration we get:

- $$A\mathbf{x}_1 = \begin{bmatrix} -2 & 1 & 1 \\ 3 & -2 & 0 \\ 1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 5 \end{bmatrix} \frac{1}{\sqrt{26}} = \begin{bmatrix} 6 \\ -2 \\ 8 \end{bmatrix} \frac{1}{\sqrt{26}}$$
- $\mathbf{x}_2 = \dfrac{A\mathbf{x}_1}{||A\mathbf{x}_1||} = [6, -2, 8]/\sqrt{104}$

Power method

If we continue for a couple of more iterations we get:

- $\mathbf{x}_2 = [0.59, -0.17, 0.78]^\top$.
- $\mathbf{x}_3 = [-0.25, 0.91, 0.33]^\top$.
- $\mathbf{x}_4 = [0.41, -0.61, 0.67]^\top$.
- $\mathbf{x}_7 = [-0.28, 0.85, -0.44]^\top$.
- $\mathbf{x}_{10} = [0.28, -0.79, 0.55]^\top$.
- We see that it seems to converge, but note that it could take some time.
- For reference the true eigenvector is $\mathbf{x}_{true} = [0.267, -0.802, 0.525]$.

Power method: convergence speed

To show why and how fast the method converge we assume A is diagonalizable with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_m$ such that $|\lambda_1| > |\lambda_j|, \quad j > 1$ and corresponding eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m$

- We write the initial vector $\mathbf{x}_0$ as a linear combination of the eigenvectors:

$$\mathbf{x}_0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \ldots + c_m\mathbf{v}_m$$

- Assuming $c_1 \neq 0$ (which is true if we choose $\mathbf{x}_0$ randomly with a high probability) after k iterations we get:

$$\mathrm{A}^k\mathbf{x}_0 = c_1\mathrm{A}^k\mathbf{v}_1 + c_2\mathrm{A}^k\mathbf{v}_2 + \ldots + c_m\mathrm{A}^k\mathbf{v}_m$$

Power method: convergence speed

$$A^k \mathbf{x}_0 = c_1 A^k \mathbf{v}_1 + c_2 A^k \mathbf{v}_2 + \ldots + c_m A^k \mathbf{v}_m$$

- Since $\lambda_j \mathbf{v}_j = A\mathbf{v}_j$ we can rewrite this:

$$A^k \mathbf{x}_0 = c_1 \lambda_1^k \mathbf{v}_1 + c_2 \lambda_2^k \mathbf{v}_2 + \ldots + c_m \lambda_m^k \mathbf{v}_m$$

- Which can be rewritten by:

$$A^k \mathbf{x}_0 = c_1 \lambda_1^k \left(\mathbf{v}_1 + \frac{c_2}{c_1} \left(\frac{\lambda_2}{\lambda_1} \right)^k \mathbf{v}_2 + \ldots + \frac{c_m}{c_1} \left(\frac{\lambda_m}{\lambda_1} \right)^k \mathbf{v}_m \right)$$

- Since $|\lambda_1| > |\lambda_j|, \quad j > 1$ the expression on the right side will converge to $\mathbf{v}_1$.

Power method: convergence speed

For the power method we had

$$\mathbf{x}_{k+1} = \frac{A\mathbf{x}_k}{||A\mathbf{x}_k||}$$

Which gives:

$$\mathbf{x}_k = \frac{A^k\mathbf{x}_0}{||A^k\mathbf{x}_0||}$$

Since the scaling doesn't effect the convergence speed it's obvious from the previous slide that $\mathbf{x}_k$ converge to some multiple of the eigenvector $\mathbf{v}_1$. We can also see that the convergence speed depend on the relative size between the largest and second largest eigenvalue $\left|\frac{\lambda_2}{\lambda_1}\right|$.

Power method: Time complexity

We have seen that the method converge geometrically but what is the work needed in every iteration?

In every iteration we need to do one *matrix* $\times$ *vector* multiplication which can be done in $O(n^2)$ time, where n is the length of the vector.

Calculating the normalization constant is a summation of n elements ($O(n)$) and normalizing the vector with this constant is done by n multiplications (and one division).

Thus we find that in every iteration we need to do $O(n^2)$ operations. Unless λ_2 is very close to λ_1 we also get $O(n^2)$ for the whole algorithm itself.

Power method: Time complexity

If the matrix is sparse with on average p non zero elements on every row the *matrix* $\times$ *vector* multiplication can instead be done in $O(pn)$ time.

In practice in many applications p does not increase with n (such as for many real life networks), thise means we essentially have a linear algorithm $O(n)$, at least if we know that λ_2 is unlikely to be very close to λ_1

Power method

There are a couple of things to consider when using the Power method:

- If there is more than one eigenvalue on the spectral radius or if it's multiplicity is larger than one, the method won't converge.
- Since the error is of the order $|\lambda_2/\lambda_1|^k$, if λ_2 is close to λ_1 the convergence can be very slow.
- We only get the (in absolute value) largest eigenvalue and corresponding eigenvector.
- There are however ways to handle some of these weaknesses as we will see later.

Power method

While the method does have a couple of negatives, we found that it does have a couple of very strong points as well.

- In every step we make only a single vector-matrix multiplication.
- If A is a sparse matrix the iterations will be fast even for extremely large matrices.
- Since we do no matrix decomposition (as we do in most other methods) we do not "destroy" the sparcity of the matrix when calculating the matrix decomposition.

Power method

But how can we handle the problems found earlier?

- We will start by looking at something called the inverse power method.

Inverse power method

We consider the matrix invertible matrix A with one eigenpair λ, x.

- We then have $\mathsf{A}\mathbf{x} = \lambda\mathbf{x}$ from the definition. This gives:
- $$\mathsf{A}\mathbf{x} = \lambda\mathbf{x} \Leftrightarrow (\mathsf{A}-\mu\mathsf{I})\mathbf{x} = (\lambda-\mu)\mathbf{x} \Leftrightarrow \mathbf{x} = (\mathsf{A}-\mu\mathsf{I})^{-1}(\lambda-\mu)\mathbf{x}$$
$$\Leftrightarrow (\lambda-\mu)^{-1}\mathbf{x} = (\mathsf{A}-\mu\mathsf{I})^{-1}\mathbf{x}$$
- We see that $(\lambda-\mu)^{-1}, \mathbf{x}$ is an eigenpair of $(\mathsf{A}-\mu\mathsf{I})^{-1}$.

Inverse power method

If we instead iterate using this new matrix $(\mathsf{A} - \mu\mathsf{I})^{-1}$.

- The method becomes:

$$\mathbf{x}_{k+1} = \frac{(\mathsf{A} - \mu\mathsf{I})^{-1}\mathbf{x}_k}{||(\mathsf{A} - \mu\mathsf{I})^{-1}\mathbf{x}_k||}$$

  .

- This will converge to the eigenvector of the dominant eigenvalue of $(\mathsf{A} - \mu\mathsf{I})^{-1}$.
- We get the dominant eigenvalue $(\lambda - \mu)^{-1}$ for the λ closest to μ.

Inverse power method

If we look at the error we get:

- $$\left| \frac{(\lambda_{(\text{second closest to } \mu)} - \mu)^{-1}}{(\lambda_{(\text{closest to } \mu)} - \mu)^{-1}} \right|^k$$
- $$\Leftrightarrow \left| \frac{(\lambda_{(\text{closest to } \mu)} - \mu)}{(\lambda_{(\text{second closest to } \mu)} - \mu)} \right|^k$$

Inverse power method

We can easy see that if μ is a good approximation of an eigenvalue of A the error:

$$\left| \frac{(\lambda_{(\text{closest to } \mu)} - \mu)}{(\lambda_{(\text{second closest to } \mu)} - \mu)} \right|^k$$

will be very close to zero and the method converge very fast. Often in only one or a couple iteration.

Inverse power method

By using different μ we can find different eigenvectors of A, not only the dominant one.

- However we still have the problem with eigenvalues with multiplicity larger than one, or more than one with the same absolute value.
- The Inverse power method is often used when you have a good approximation of an eigenvalue and need to find an approximation of the corresponding eigenvector.
- The inverse power method requires us to either solve a linear system or calculate the inverse of the matrix $(\mathrm{A} - \mu)$. This makes it unsuitable for huge sparse matrices where the basic power algorithm works better.

Inverse power method: time complexity

So we found that using the inverse method we could correct some problems with the method, but this also comes with a price in that we must be able to calculate the inverse of the matrix $(A - \mu)$.

- Inverting a $n \times n$ matrix using for example Gauss-Jordan requires $O(n^3)$ operations.
- Inverting a matrix typically destroys the sparcity of the matrix, such that we can no longer use the sparcity of A during our iterations anymore.
- Even if we find an effective way to calculate the inverse matrix, the iteration steps typically require much more time than the basic algorithm for sparse matrices.

QR-method

Often it's not enough to find only one eigenvalue or eigenvector but you need all of them.

- In general, when you use a program to find eigenvalues and eigenvectors it's nearly always the QR-method you use.
- But before we look at the method we will take a short look at QR-factorization and why this is relevant.

QR-decomposition

Theorem

Every $n \times m$ matrix A have a matrix decomposition

$$A = QR$$

. *where*

- R *is a* $n \times m$ *upper triangular matrix..*
- Q *is a* $n \times n$ *unitary matrix.*

QR-decomposition

A unitary matrix is a square matrix satisfying:

- $U^H U = U U^H = I$
- In other words $U^{-1} = U^H$.
- Where U^H is the conjugate transpose $(A^H)_{ij} = \overline{(A)_{ji}}$.

QR-decomposition

We let U be a unitary matrix, some important properties are:

- The condition number $\kappa(\mathrm{U}) = 1$, making for very well-conditioned and robust systems or algorithms.
- Since $\mathrm{U}^{-1} = \mathrm{U}^{H}$ we can very easily find the inverse of U.
- This means similarity transforms $\mathrm{U}^{H}\mathrm{AU}$ involving U is both easy to compute and robust.

QR-decomposition

If A is a square real matrix then Q is an orthogonal matrix $Q^\top = Q^{-1}$.

- QR-factorization plays an important role in the QR-method commonly used to calculate eigenvalues and eigenvectors.
- The QR-factorization can be computed using for example Householder transformations or the Gram-Schmidt process.

QR-method

The QR-method is an iterative algorithm for computing the eigenvalues and eigenvectors of a matrix A using QR-factorization.

- We start by assigning $A_0 := A$
- In every step we then compute the QR-decomposition $A_k = Q_k R_k$.
- We then form $A_{k+1} = R_k Q_k = Q_k^H Q_k R_k Q_k = Q_k^H A_k Q_k = Q_k^{-1} A_k Q_k$
- So all A_k are similar and therefor have the same eigenvalues.
- When the method converge A_k will be a triangular matrix and therefor have it's eigenvalues on the diagonal.

QR-method

The main advantage of the QR-method is that it's not only fast when implemented with some necessary modifications.

- It's also very stable in that a small change in A only gives a small change in the computed eigenvalues.
- This is one of the main reason it's used over other similar methods such as for example methods using the LUP decomposition.

QR-method

The QR-method can be seen as a variation of the Power method but instead of only one vector, we work with a complete basis of vectors, using QR-decomposition to normalize and orthogonalize in every step.

- As with the Power method, the QR-method in it's basic form have a couple of things to consider.
- Computing the QR-decomposition of a matrix is expensive, and we need to do it in every step.
- The convergence can be very slow.
- Like the Power method we have problems with complex and non simple eigenvalues.

QR-method

- To solve the first problem the matrix A is nearly always first brought to upper Hessenberg form.
- This will both decrease the cost of the QR-factorizations as well as improve the convergence

Hessenberg matrix

A matrix A is a (upper) Hessenberg matrix if all elements $a_{ij} = 0, i > j + 1$. In other words A is of the form:

$$A = \begin{bmatrix} \bigstar & \bigstar & \bigstar & \cdots & \bigstar & \bigstar & \bigstar \\ \bigstar & \bigstar & \bigstar & \cdots & \bigstar & \bigstar & \bigstar \\ 0 & \bigstar & \bigstar & \cdots & \bigstar & \bigstar & \bigstar \\ 0 & 0 & \bigstar & \cdots & \bigstar & \bigstar & \bigstar \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \bigstar & \bigstar & \bigstar \\ 0 & 0 & 0 & \cdots & 0 & \bigstar & \bigstar \end{bmatrix}$$

Hessenberg matrix

Theorem

Every square matrix A have a matrix decomposition

$$A = U^H H U$$

where

- U *is a unitary matrix.*
- H *is a Hessenberg matrix.*

Hessenberg matrix

Since U is a unitary matrix, A and H are similar and we can work with H rather than A in the QR-method.

- This is useful for a couple of reasons.
- Calculating the QR-factorization of a Hessenberg matrix is much faster than for a general matrix.
- Since multiplication of a Hessenberg matrix with a triangular matrix (both in the same direction) is itself a Hessenberg matrix. We need only do the Hessenberg factorization once.
- Since a Hessenberg matrix is nearly triangular, its convergence towards the triangular matrix will be improved.
- Since U is unitary, it's condition number is 1, numerical erros when finding H is not amplified.

QR-method

However the convergece can still be very slow even when working with the Hessenberg form.

- We remember how we could solve this in the Power method by shifting the eigenvalues of the matrix.
- We can do the same in the QR-method using $(A_k - \sigma I)$.
- Rewriting as before we get $A_{k+1} = Q_k^H(A_k - \sigma I)Q_k + \sigma I = Q_k^H(A_k)Q_k$.
- We see that A_k, A_{k+1} is still similar and therefor will have the same eigenvalues.
- As in the Power method we want σ to be as close as possible to one of the eigenvalues.
- The diagonal elements of the hessenberg matrix prove to give a good approximation.

QR-method

- Since the diagonal elements of A_k should give increasingly accurate approximation of the eigenvalues we can change our shift to another eigenvalue or improve the current one between iterations as well.
- As fast as a subdiagonal element is close to zero, we have either found an eigenvalue if it's one of the outer elements.
- Or we can divide the problem in two smaller ones since we would have:
- $A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix}$

QR-method

- In practice an implicit method using multiple shifts at the same time is usually used.
- This also solves the problems with complex eigenvalues in real matrices.
- The Implicit version is also called the Francis algorithm.

QR-method

To find the eigenvectors

- The eigenvectors can either be found by multiplying $Q = Q_0 Q_1 \dots Q_k$ with the initial unitary matrix to bring the problem to Hessenberg from.
- They can also be found through the use of a form of inverse iteration.
- It's also possible to use for example Gauss elimination using the eigenvalues on our original matrix.

QR-method

We give a summary of the modified method using a single shift:

1. Compute the Hessenberg form ($A = U^H H U$) using for example Householder transformations.
2. Assign $A_0 = H$. Then in every step, choose a shift σ as one of the diagonal elements of A_k.
3. Compute the $Q_k R_k$ factorization of $(A_k - \sigma I)$.
4. Compute $A_{k+1} = R_k Q_k + \sigma I$
5. Iterate $2 - 4$ until A_k is a triangular matrix, optionally dividing the problem in two whenever a subdiagonal element becomes zero.

QR-method: time complexity

We said that the QR-method is fast, but how fast is it? The most expensive steps are:

- Bringing the matrix to Hessenberg form can be done in $O(n^3)$ time.
- Finding the QR decomposition of a upper Hessenberg matrix can be done in $O(n^2)$ time (compared to $O(n^3)$ for a full matrix).
- If we assume we need $O(n)$ iterations we end up with $O(n^3)$ if we bring it to Hessenberg form, and $O(n^4)$ if not.

Representing a general sparse matrix

- To store all elements in a $n \times m$ matrix we would need to be able to store nm elements.
- But what if most of these elements was zero? We would like to store only the non-zero elements.
- Sometimes we have a general sparse matrix and sometimes it might also have some useful structures we can use to store it more efficiently.

Representing a general sparse matrix

The Yale format is a format for storing a general sparse matrix.

- Yale is very similar to compressed sparse row (CSR) and compressed sparse column (CSC).
- Yale stores a sparse matrix M using 3 vectors.
- A containing the non-zero elements.
- IA containing the index in A of the first non-zero element on every row of M.
- JA contains the column number for all the elements in A.
- For both IA and JA the first row/column is considered row/column zero.

Representing a general sparse matrix

Consider the matrix M:

$$\begin{bmatrix} 11 & 22 & 0 & 0 & 33 & 0 \\ 0 & 0 & 44 & 0 & 0 & 0 \\ 55 & 66 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 77 & 0 & 88 \end{bmatrix}$$

- We get:

$$A = \begin{bmatrix} 11 & 22 & 33 & 44 & 55 & 66 & 77 & 88 \end{bmatrix}$$

- $$IA = \begin{bmatrix} 0 & 3 & 4 & 6 & 8 \end{bmatrix}$$

- $$JA = \begin{bmatrix} 0 & 1 & 4 & 2 & 0 & 1 & 3 & 5 \end{bmatrix}$$

Representing a general sparse matrix

The Yale format is constructed to allow not only efficient storage, but also efficient calculations as well.

- Think for a moment how you would implement addition of two matrices in Yale format.
- What about multiplication with a vector?

Representing sparse band matrices

One common type of sparse matrices are band matrices where all non-zero elements lieas close to the diagonal. Given a matrix A:

- The lower (left) half-bandwidth is the smallest k_1 such that:
$$a_{ij} = 0, \quad i > j + k_1$$
- The upper (right) half-bandwidth is the smallest k_2 such that:
$$a_{ij} = 0, \quad i < j - k_2$$
- The matrix bandwidth b is then:
$$b = k_1 + k_2 + 1$$

Representing sparse band matrices

We consider the matrix below.

$$\begin{bmatrix} 11 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 34 & 0 & 0 \\ 0 & 0 & 43 & 44 & 0 & 46 \\ 0 & 0 & 0 & 54 & 0 & 56 \\ 0 & 0 & 0 & 0 & 65 & 66 \end{bmatrix}$$

- All elements below the sub-diagonal is zero so we get lower bandwidth $k_1 = 1$.
- Similarily we get upper bandwidth $k_2 = 2$.
- And bandwidth $b = 4$.

Representing sparse band matrices

A $n \times m$ matrix A with bandwidth b can be represented by a matrix B of size $n \times b$.

- We let every row of our new matrix represent the elements starting k_1 elements to the left of the diagonal and ending k_2 elements to the right.
- This way every column of B represent one of the diagonals of A.
- We also need to store which column of B contains the diagonal of A.

Representing sparse band matrices

The mattrix to the left can be represented by the matrix to the right:

$$A = \begin{bmatrix} 11 & 0 & 0 & 0 & 0 & 0 \\ 21 & 22 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 34 & 0 & 0 \\ 0 & 0 & 43 & 44 & 0 & 46 \\ 0 & 0 & 0 & 54 & 0 & 56 \\ 0 & 0 & 0 & 0 & 65 & 66 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 11 & 0 & 0 \\ 21 & 22 & 0 & 24 \\ 0 & 0 & 34 & 0 \\ 43 & 44 & 0 & 46 \\ 54 & 0 & 56 & 0 \\ 65 & 66 & 0 & 0 \end{bmatrix}$$

Cholesky factorization and fill-in

Cholesky factorization is used to decompose a matrix A such that $A = GG^H$, where A is *Hermitian* and *positive-definite* and G is lower triangular.

- Doing a matrix factorization of a sparse matrix often destroys the sparcity of the matrix by adding many new non-zero elements (so called fill in).
- Permutating rows and columns can greatly reduce this fill in.

Cholesky factorization and fill-in

We consider a real matrix matrix A with Cholesky factorization $A = GG^\top$.

$$A = \begin{bmatrix} \bigstar & \bigstar & \bigstar & 0 & 0 \\ \bigstar & \bigstar & \bigstar & \bigstar & 0 \\ \bigstar & \bigstar & \bigstar & \bigstar & \bigstar \\ 0 & \bigstar & \bigstar & \bigstar & \bigstar \\ 0 & 0 & \bigstar & \bigstar & \bigstar \end{bmatrix} \quad G = \begin{bmatrix} \bigstar & 0 & 0 & 0 & 0 \\ \bigstar & \bigstar & 0 & 0 & 0 \\ \bigstar & \bigstar & \bigstar & 0 & 0 \\ X & \bigstar & \bigstar & \bigstar & 0 \\ X & X & \bigstar & \bigstar & \bigstar \end{bmatrix}$$

Where $\bigstar$ represent a non-zero element. What happens if any of the X elements in G would be non-zero?

Cholesky factorization and fill-in

$$GG^\top = \begin{bmatrix} \star & 0 & 0 & 0 & 0 \\ \star & \star & 0 & 0 & 0 \\ \star & \star & \star & 0 & 0 \\ X_1 & \star & \star & \star & 0 \\ X_2 & X_3 & \star & \star & \star \end{bmatrix} \begin{bmatrix} \star & \star & \star & X_1 & X_2 \\ 0 & \star & \star & \star & X_3 \\ 0 & 0 & \star & \star & \star \\ 0 & 0 & 0 & \star & \star \\ 0 & 0 & 0 & 0 & \star \end{bmatrix}$$

$$= \begin{bmatrix} \star & \star & \star & Y_1 & Y_2 \\ \star & \star & \star & \star & Y_3 \\ \star & \star & \star & \star & \star \\ Y_1 & \star & \star & \star & \star \\ Y_2 & Y_3 & \star & \star & \star \end{bmatrix}$$

- If $X_1 \neq 0$ then $Y_1 \neq 0$ and if $X_2 \neq 0$ then $Y_2 \neq 0$.
- If $X_1 = X_2 = 0$ then if $X_3 \neq 0$ then $Y_3 \neq 0$.

Cholesky factorization and fill-in

- If we reduce the bandwidth of A, we also increase the number of guaranteed zero elements of G.
- If we could find a suitible permutation matrix we could potentially greatly reduce the number of non-zero elements after the Cholesky factorization.
- Storing it as a bandmatrix we could also allocate space for it beforehand.

Cuthill-McKee

The Cuthill-McKee algorithm is used to permutate a sparse symmetric matrix A into a bandmatrix with a short bandwidth.

- It works by considering A as the adjacency matrix of a graph.
- The new matrix is then created using a breadth-first algorithm starting in one of the "peripheral" vertices.

Cuthill-McKee

Given a $n \times n$ sparse symmetric matrix A. At the end we want to have a set R with n elements containing the order in which we want the n vertices.

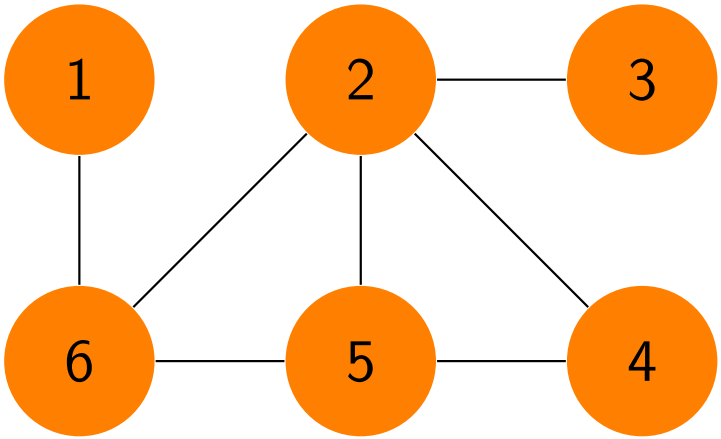
- Start by selecting one of the vertices with the lowest degree (peripheral vertex) x and add it to R: $R = \{x\}$.
- Then for $i = 1, 2, \ldots$ we iterate until $|R| = n$:
  - Find the set of vertices adjacent to vertex i in R but not in R called A_i.

$$\mathrm{A}_i := \mathrm{Adj}(R_i) \setminus R$$

  - Sort A_i in ascending order using the vertex degree.
  - Add A_i at the end of R.
- In other words we add vertices using breadth first (first add all neighbors to the previously added vertex before going to the next) where the vertices are added from lowest to highest degree.

Cuthill-McKee

Let us consider the following matrix and corresponding graph(ignoring self loops):



$$A = \begin{bmatrix} \bigstar & 0 & 0 & 0 & 0 & \bigstar \\ 0 & \bigstar & \bigstar & \bigstar & \bigstar & \bigstar \\ 0 & \bigstar & \bigstar & 0 & 0 & 0 \\ 0 & \bigstar & 0 & \bigstar & \bigstar & 0 \\ 0 & \bigstar & 0 & \bigstar & \bigstar & \bigstar \\ \bigstar & \bigstar & 0 & 0 & \bigstar & \bigstar \end{bmatrix}$$

Doing a cholesky factorization we could end up with a full upper triangular matrix G! Instead we want to reduce the potential fill-in.

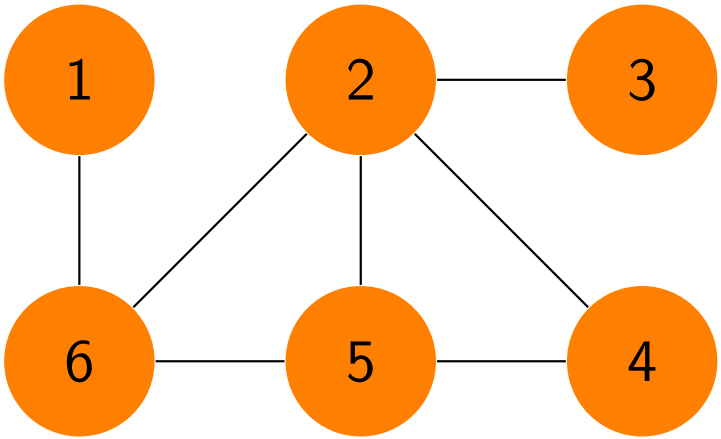
Cuthill-McKee

We use Cuthil-McKee to create a band matrix:

- We begin by selecting any of the vertices with minimum degree (vertex 1 or 3). We choose to begin with $R = \{1\}$.
- $(i = 1)$ The only neighbor to vertex 1 is vertex 6 so it is added next. $R = \{1, 6\}$.
- $(i = 2)$ vertex 2 and 5 are the neighbors to node 6 not already in R: $A_2 = \{2, 5\}$. We sort them in ascending order acording to degree and add them to R: $R = \{1, 6, 5, 2\}$.
- $(i = 3)$ node 4 is the only vertex adjacent to vertex 5 not already in R so we add it next. $R = \{1, 6, 5, 2, 4\}$
- $(i = 4)$ node 3 is the only vertex adjacent to vertex 2 not already in R so we add it next. $R = \{1, 6, 5, 2, 4, 3\}$
- Since R contains 6 elements we are finished.

Cuthill-McKee

Creating the new matrix by ordering them $\{1, 6, 5, 2, 4, 3\}$ gives:



$$A = \begin{bmatrix} \bigstar & \bigstar & 0 & 0 & 0 & 0 \\ \bigstar & \bigstar & \bigstar & \bigstar & 0 & 0 \\ 0 & \bigstar & \bigstar & \bigstar & \bigstar & 0 \\ 0 & \bigstar & \bigstar & \bigstar & \bigstar & \bigstar \\ 0 & 0 & \bigstar & \bigstar & \bigstar & 0 \\ 0 & 0 & 0 & \bigstar & 0 & \bigstar \end{bmatrix}$$

Doing a cholesky factorization we would now probably end up with many more zero elements in G than before, saving both storage and computation time.

Cuthill-McKee

Some things to note about the Cuthill-McKee algorithm and fill-in for the Cholesky factorization:

- While the algorithm generally finds a "good" ordering to reform a matrix as a bandmatrix, it generally does not find the "best" ordering.
- Finding the ordering which gives the least possible fill-in can actually be shown to be NP-hard. In other words it probably doesn't even have a polynomial time algorithm.