Numerical Computing Cheatsheet

Edoardo Riggio

January 18, 2022

Numerical Computing - SA. 2021
Computer Science
Università della Svizzera Italiana, Lugano

Contents

1 PageRank Algorithm

The PageRank algorithm is entirely determined by the link structure of the World Wide Web. A page will thus have a high rank if other pages with high ranks link to it.

1.1 Random Surfer Model

This algorithm is based on the **Random Surfer Model**. Here we imagine a user going from one page to the other by randomly choosing an outgoing link from one page to the next. This process is also called **exploitation**.

The problem with exploitation is that it could lead to dead-ends – i.e. a page with no outgoing links, or cycles around cliques of interconnected pages. For this reason, we sometimes choose a random page from the web to navigate to. This process is called **exploration**.

1.2 Markov Chains

The random walk generated by the combination of exploitation and exploration is known as a **Markov Chain**. A Markov Chain – or Markov Process – is a stochastic process. Differently from other stochastic processes, it has the property of being memory-less. This means that the probability of future states are not dependent upon the steps that led up to the present state.

1.2.1 Markov Matrix

Let W be the set of webpages that can be reached by a Markov Chain of hyperlinks, n the number of pages in W, and G the $n \times n$ connectivity matrix of a portion of the Web. The matrix G will be composed as follows:

$$g_{ij} = \begin{cases} 1 & \text{if there is a hyperlink from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

From matrix G we can determine the in-degree and out-degree of a page j. This can be computed as follows:

$$r_i = \sum_j g_{ij}$$

$$c_j = \sum_i g_{ij}$$

Where r_i is the in-degree and c_j is the out-degree. Let now p be the probability that the random walk follows a link – i.e. performs exploitation. A typical value for p is 0.85. Let δ be the probability that a particular random page is chosen,

and $1 - p$ is the probability that some random page is chosen – i.e. perform exploration. Then δ will have the following formulation:

$$\delta = \frac{1-p}{n}$$

Now let A be a matrix that comes from scaling G by its column sums. The elements of matrix A will be:

$$a_{ij} = \begin{cases} p \cdot \dfrac{g_{ij}}{c_j} + \delta & \text{if } c_j \neq 0 \\ \dfrac{1}{n} & \text{if } c_j = 0 \end{cases}$$

Matrix A is the transition probability matrix for the Markov Chain – for this reason this matrix is also known as the Markov matrix. All of its elements are strictly between 1 and 0, and its column sums are all equal to 1.

1.3 Eigenvectors and Eigenvalues

1.3.1 Eigenvector

An **eigenvector** is a non-zero vector that changes at most by a scalar factor when a linear transformation is applied to it.

1.3.2 Eigenvalue

An **eigenvalue** is the factor by which the eigenvector is scaled. Thus, this relation holds true:

$$A\overrightarrow{v} = \lambda\overrightarrow{v}$$

Where A is the transformation matrix, $\overrightarrow{v}$ is the eigenvector, and λ is the eigenvalue.

1.3.3 Eigenbasis

An **eigenbasis** is the basis vector space which consists entirely of eigenvectors. Mathematically, this is represented by a diagonal matrix in which all of its columns are eigenvectors. This basis is not always computable.

1.3.4 Rayleigh Quotient

The **Rayleigh Quotient** is a simple way to find – given an eigenvector – its corresponding eigenvalue. The following is the quotient:

$$\mu(v) = \frac{v^T A v}{v^T v}$$

Where v is the vector for which we need to compute the eigenvalue, and A is the transformation matrix. One thing to note, is that if the vector v is an eigenvector, then $\mu(v)$ will be the corresponding eigenvalue. On the other hand, if v is not an eigenvector, then $\mu(v)$ will be the eigenvalue of the closest eigenvector of v.

This quotient is useful in the computation of the dominant eigenvalue. This is done by the power iteration and by the inverse iteration.

1.4 Power Iteration

The **power iteration** is used in order to compute the dominant eigenvector of a matrix A. Before we can compute it we need to make some assumptions:

- The matrix A has one eigenvalue that is strictly greater than all of the other eigenvalues
- The starting vector needs to have a non-zero component in the direction of the dominant eigenvector

In order to find the dominant eigenvalue, we need to follow this algorithm:

1. Start with an initial guess v_0, which is a vector
2. Compute w, s.t. $w = Av_k$
3. Set $v_{k+1} = ||w||$
4. Find the eigenvalue of v_{k+1} by using the Rayleigh quotient
5. Repeat steps 2 to 4 until the difference between the previous and current eigenvectors is under a certain threshold. Increment k by 1

1.5 Inverse Iteration

The **inverse iteration** is very similar to the power method. In the case of the inverse iteration, we can use it to find a particular eigenvalue – which is not necessarily the dominant one.

This can be done by applying the power method to $(A - \alpha I)^{-1}$ instead of A. By doing so, the α constant will contribute in making the non-dominant eigenvector the dominant one.

This algorithm converges much faster than the previous one. On the other hand, it is also more computationally expensive than the previous. In this case we will need to compute a system of equations for every step of the algorithm.

$$(A - \alpha I)^{-1}\overrightarrow{v} = \frac{1}{\lambda - \alpha}\overrightarrow{v} \quad \Longleftrightarrow \quad (A - \alpha I)x_{n+1} = x_n$$

2 Social Networks

In order to find a partition in a graph, we need to use the second smallest eigenvector – i.e. eigenvector v_2. This eigenvector is also known as the **Fiedler eigenvector**, and plays a very important role in graph partitioning. This is because all of the indices corresponding to vector entries can be divided as follows:

$$v_{2i} \in \begin{cases} \text{Set 1} & \text{if } v_{2i} > 0 \\ \text{Set 2} & \text{if } v_{2i} < 0 \end{cases}$$

The arising partition minimizes the number of edges between the two sets.

Furthermore, λ_2 of the graph Laplacian matrix is the algebraic connectivity of the graph G. The magnitude of λ_2 measures the connectivity of the graph. In particular, $\lambda_2 \neq 0$ iff G is a connected graph.

2.1 Matrix Decompositions and Permutations

2.1.1 Cholesky Decomposition

By using the **Cholesky Decomposition**, any symmetric positive definite matrix can be decomposed into the product:

$$A = LL^T$$

Where L is a lower triangular matrix with positive diagonal elements – also known as the Cholesky factor.

2.1.2 Reverse Cuthill McKee Ordering

The **Reverse Cuthill McKee Ordering** is the permutation of a sparse matrix A – which has a symmetric sparsity pattern – into a banded matrix with a small bandwidth. The resulting matrix will have its non-zero elements closer to the diagonal of the matrix.

This permutation can be very useful when dealing with large sparse matrices. If we wanted to perform a Cholesky Decomposition on such matrix, this would result in a very high number of fill-ins. This would make the computation of the Cholseky factor very inefficient. But, if we first use the Reverse Cuthill McKee Ordering, we would drastically reduce the number of fill-ins. This is because the permutation would move all of the elements closer to the diagonal of the matrix.

2.2 Centrality

2.2.1 Degree Centrality

The **degree centrality** of a matrix A is defined as the number of links incident upon a node. This means that it represents the number of vertices each node has.

2.2.2 Eigenvector Centrality

Eigenvector centrality is a measure of the influence of a node in a network. All relationships originating from high-scoring nodes contribute more to the score of the node than connections from low-scoring nodes.

A high eigenvector score means that a node is connected to many nodes who themselves have high scores.

3 Graph Partitioning

The graph partitioning problem is defined in the form of a graph G, such that it is possible to partition G into smaller components with specific properties.

There are several partitioning algorithms, and they divide into two main categories:

- **Local Approaches**
- **Global Approaches**

3.1 Graph Laplacian Matrix

The **Graph Laplacian Matrix** is the main tool for spectral partitioning. This is the matrix representation of a graph G. A graph Laplacian matrix is computed in two ways:

$$L = D - A$$
$$L = D - W$$

Where D is the degree matrix, A is the adjacency matrix, and W is the weight matrix. In the first case, we consider an **undirected and unweighted** graph, while in the second case, we consider an **undirected and weighted** graph.

Some properties of the graph Laplacian matrix $L(G)$ are:

- It is symmetric, meaning that the eigenvalues of $L(G)$ are real and its eigenvectors are real and orthogonal
- The eigenvalues of $L(G)$ are non-negative

3.1.1 Degree Matrix

A **degree matrix** is a diagonal matrix which contains the degree of each vertex. In the case of an **unweighted** graph, we consider as the diagonal entries the number of nodes that node is connected to. For example:

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Means that the first node is connected to 1 other node, the second node is connected to 2 other nodes, and the third node is connected to 1 other node.

In the case of a **weighted** graph, the non-zero values of the diagonal matrix represent the sum of all the weights of the connected edges. For example:

$$D = \begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 1.2 & 0 \\ 0 & 0 & 1.7 \end{bmatrix}$$

Means that the first node has n connected edges with total weight of 0.3, the second node has n connected edges with total weight of 1.2, and the third node has n connected edges with total weight of 1.7.

3.1.2 Adjacency Matrix

An **adjacency matrix** is a square matrix used to represent the connections of an **unweigthed** graph. If the graph is **undirected**, then its entries will be:

$$a_{ij} = \begin{cases} 1 & \text{if } i \text{ is connected to } j \text{ or } j \text{ is connected to } i \\ 0 & \text{otherwise} \end{cases}$$

3.1.3 Weight Matrix

A **weight matrix** is a square matrix used to represent the connections of a **weighted** graph. If the graph is **undirected**, then its entries will be:

$$w_{ij} = \begin{cases} w & \text{if } i \text{ is connected to } j \text{ or } j \text{ is connected to } i \\ 0 & \text{otherwise} \end{cases}$$

Where w is the weight of the edge of the graph.

3.2 Local Approaches

The major drawback of this approach is the arbitrary initial partitioning of the vertex set, which can affect the final solution quality.

Two algorithms that have a local approach are **Kerighan-Lin algorithm** and the **Fiduccia-Mattheyses algorithms**. These where the first two effective 2-way cuts by local strategy.

3.3 Global Approaches

Global approaches rely on the properties of the entire graph, rather than on an arbitrary initial partition.

3.3.1 Spectral Bisection

This algorithm is one of the most common in graph partitioning. Here the partitions are derived from the graph Laplacian matrix.

In order to perform a spectral bisection, we need to follow these steps:

1. **Pre-Processing**

   Here we compute the graph Laplacian matrix.

2. **Decomposition**

   Here we compute the second smallest eigenvalue and corresponding eigenvector – i.e. Fiedler eigenvector.

3. **Grouping**

   Here we divide the nodes based on the sign of the second eigenvector. More specifically:

$$v_i \in \begin{cases} G_1 & \text{if } x_{2i} < 0 \\ G_2 & \text{if } x_{2i} \geq 0 \end{cases}$$

   Where v_i is the i-th vertex of the graph, G_1 is the first partition, G_2 is the second partition, and x_{2i} is the i-th entry of the second smallest eigenvector.

By doing so, we will obtain two roughly balanced partitions with minimum edgecut.

3.3.2 Inertial Bisection

This algorithm relies on the geometric layout of the graph. The main idea is to find an hyperplane that runs through the centre of mass of the points. In order to find the partitions, we follow this algorithm:

1. **Pre-Processing**

   Here we compute the center of mass of the points

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad\qquad \overline{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

And matrix M

$$S_{xx} = \sum_{i=1}^{n} (x_i - \overline{x})^2$$
$$S_{yy} = \sum_{i=1}^{n} (y_i - \overline{y})^2$$
$$S_{xy} = \sum_{i=1}^{n} ((x_i - \overline{x})(y_i - \overline{y}))$$

$$M = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix}$$

2. **Decomposition**

   Here we compute the smallest eigenvalue and corresponding eigenvector of matrix M. This eigenvector is used in order to minimize the distance of the nodes to the line.

   $$u^T M u$$

   Where u is the smallest eigenvector.

3. **Grouping**

   We project each point to the line, compute the median and use this median in order to partition the nodes. We will thus have a line such that half nodes are on one side of the line, and the other half on the other side.

3.4 Recursive Bisection

The **recursive bisection** algorithm is highly dependent on the decisions made during the earlier stages of the partitioning process. Furthermore, it also suffers from the lack of global information. Thus, it may result in suboptimal partitions.

3.5 K-Way Partitioning

The graph G is initially coarsened down to a small number of vertices. A **k-way partitioning** of this much smaller graph is computed. Then, this partitioning is projected bask towards the original finer graph by successfully refining the partitioning at each intermediate step.

The main difference between this method and the previous one, is that here global information about the graph is stored.

4 Graph Clustering

The main reason as to why we use clustering is the need to extract information from a set of data.

Given a set of data points and some notion of similarity between two pairs of nodes, the goal of clustering is to divide the data points into several groups such that points in the same group are similar and points in different groups are dissimilar to each other.

4.1 Trees

A tree is an undirected graph in which any two vertices are connected by exactly one path.

4.1.1 Minimum Spanning Tree

A spanning tree is a subgraph that is a tree which includes all of the vertices of G, with the minimum possible number of edges and the minimum possible total edge weight.

4.2 Graphs

4.2.1 Similarity Graphs

In order to build such similarity graphs, we can use one of the following methods:

- **ϵ-neighbourhood graph**

  Here we connect all of the points whose pairwise distances are smaller than a threshold value ϵ. ϵ is chosen such that the resulting graph is safely connected.

- **k-nearest neighbour graph**

  Here we connect vertex v_i to vertex v_j, if v_j is among the k-nearest neighbours of v_i. For large graphs, we choose $k \sim \log(n)$.

- **Fully connected graph**

  Here we connect all points with one edge based on a similarity function s_{ij}. This will result in a full graph. In this case σ controls the size of the neighbourhood.

4.2.2 Graph Construction

In order to construct the matrix W, we do the following:

$$W = S \odot G$$

Where the $\odot$ operator performs element-wise multiplication, S represents the full similarity matrix constructed using the Gaussian similarity function, and G represents an ϵ-similarity graph matrix.

4.3 Clustering Algorithms

4.3.1 Spectral Clustering

In order to perform **spectral clustering** on a graph, we need to follow this algorithm:

1. Compute the minimum spanning tree of the graph in order to determine the ϵ value for the ϵ similarity graph
2. Generate the ϵ-neighbourhood graph
3. Create the adjacency matrix for the ϵ-neighbourhood graph – matrix $W = S \odot G$
4. Create the graph Laplacian Matrix
5. Compute the k-smallest eigenvectors based on L
6. Use k-means in order to cluster the nodes. Use the eigenvector matrix in order to obtain such clusters

The number of nodes inside of each cluster is very balanced.

4.3.2 K-Means Clustering

The **k-means clustering** algorithm makes use of centroids. **Centroids** are imaginary or real nodes in a graph which represent the center of a cluster.

The following is the algorithm for the k-means clustering:

1. Start with some initial random centroids
2. Assign each node to the nearest cluster based on its centroid

3. Recompute the centroids as the mean of the points in the cluster
4. Repeat steps 2 and 3 until no further improvement can be made

This algorithm is not guaranteed to find the optimal solution.

There are also some drawbacks with k-means clustering, namely:

- Does not work well with convex clustering scenarios
- Since we use the Euclidean distance in order to compute the distance between a point and a centroid, the data space is treated as isotropic – i.e. the distances remain unchanged by translations and rotations
- Small changes in the data will result in proportionally small changes to the position of the centroids of the clusters. For this reason this method does not deal well with outliers.
- It does not take into account the different densities of each cluster

5 Image Deblurring

Here we used the Conjugate Gradient in order to deblur an image given the exact blurred image and the original transformation matrix.

Matrix B represents the blurred image, vector b is the vectorized form of matrix B, matrix X represents the original grayscale image – where every entry corresponds to a pixel, the vector x is the vectorized form of matrix X, and matrix A indicates the transformation matrix coming from the repeated application of the **image kernel** – which is the bulr effect in this case. The sizes of the presented matrices and vectors are:

- $A \to n^2 \times n^2$
- $B \to n \times n$
- $X \to n \times n$
- $\overrightarrow{b} \to n^2$
- $\overrightarrow{x} \to n^2$

With all of these matrices and vectors defined, we can write the following system of equations:

$$Ax = b$$

By solving this system, we can recover the original image, getting rid of the

blur effect. The blurred effect in the image is given by a weighted average of the surrounding pixels to a pixel. These weights are defined by the kernel matrix K. From this matrix, we can obtain the matrix A such that the non-zero elements of each row of A correspond to the values of K.

A is a d^2 banded matrix – where $d \ll n$. This means that A is a sparse matrix.

5.1 Conjugate Gradient Method

In order to compute the solution of a linear system of equations, there are two possible methods:

- **Direct Method**

  Used to compute the exact solution in n steps. This method has a very high memory consumption due to additional fill-ins. Moreover, it uses Gaussian Elimination which – for large linear systems of equations – is too complex and computationally expensive.

- **Iterative Method**

  This method uses an arbitrary initial starting point in order to compute an approximate arbitrary solution. This method does not need any additional memory and it will converge after a few iterations. On the other side, this method is very often less robust and not as general as a direct method.

5.1.1 Definitions

The **error** in step m is the deviation of the computed point from the exact solution. This can be written as:

$$e^{(m)} = x - x^{(m)}$$

Where $x^{(m)}$ is the computed point and x is the exact solution. This error is not known during the iterations – otherwise we would know the solution.

The **residual** provides us with a measure of the real error. This is computed as:

$$r^{(m)} = b - Ax^{(m)}$$

Where $x^{(m)}$ is the computed point.

5.1.2 Steepest Descent

The **steepest descent** algorithm is a precursor to the Conjugate Gradient algorithm. Here we do the following operations:

1. Start with a random initial guess
2. Take the gradient – which is the direction of the deepest descent
3. Compute the minimum of the gradient
4. Take the new gradient s.t. it is orthogonal to the previous gradient
5. Repeat steps 2 to 4 until a certain convergence criterion is met

This algorithm has a very low convergence rate, but it does not take optimal steps in order to find the next approximate solution.

5.1.3 Conjugate Gradient

The speed of convergence of the Conjugate Gradient is determined by the condition number $\kappa(A)$ of matrix A. The larger $\kappa(A)$, the slower the improvement.

If matrix A were not to be positive-definite, we can apply this trick in order to obtain a positive-definite matrix back:

$$A^T Ax = A^T b \rightarrow \tilde{A}x = \tilde{b}$$

The premultiplication of A with A^T will result in the positive-definite matrix $\tilde{A}$.

In order to carry out the actual method, we can use a slightly modified version of the steepest descent. The difference between the two methods is that in the case of the Conjugate Gradient Method, the two gradients need to be A-orthogonal. This means that:

$$d_i^T A d_j = 0$$

Where d_i and d_j are two vectors. By doing so, this would result in the new residual being orthogonal to the old residual. Furthermore, it would mean that every new step will never redo or undo optimizations of previous steps.

5.2 Condition Number

If a small modification in b results in a big change in x, then the system is said to be **ill-conditioned**, and the condition number $\kappa(A)$ will be big. This condition number can be computed as:

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}}$$

Where σ indicates the singular values of A. For a real symmetric matrix, the singular values are equal to the absolute value of the eigenvalues.

$$\sigma = |\lambda|$$

5.3 Preconditioner

A symmetric positive preconditioner P is selected such that P^{-1} approximates to $\tilde{A}^{-1}$. We can decompose P into LL^T, where L is the Cholesky factor.

$$\begin{aligned} P^{-1}\tilde{A}x &= P^{-1}\tilde{b} \\ (L^{-1}\tilde{A}L^{-1})(Lx) &= L^{-1}\tilde{b} \\ \bar{\tilde{A}}\bar{x} &= \bar{\tilde{b}} \end{aligned}$$

Where

$$\kappa(\bar{\tilde{A}}) \ll \kappa(\tilde{A})$$

This is done in order to both decrease the condition number and to decrease the range of the eigenvalues. The computation of the preconditioner should be relatively inexpensive to find. This is why we use the **Incomplete Cholesky factorization** in order to compute the preconditioner. By using this method, we only compute the Cholesky factors for the non-zero elements of $\tilde{A}$. This will give us the preconditioner:

$$P = F^T F$$

Where F is the sparse IC factor. This routine can fail since the existence of F is not guaranteed. In order to amend to this issue, we can apply a diagonal shift to P. By doing so, we enforce positive-definiteness, thus making F computable.

6 Linear Programming

Linear programming is an optimization method which aims to maximize or minimize a linear objective function subject to linear equality or inequality constraints. Such an example is:

$$
\begin{aligned}
\max \quad & \sum_{i=1}^{n} c_i x_i \\
\text{s.t.} \quad & \sum_{j=1}^{n} a_{1,j} x_j \leq h_1 \\
& \vdots \\
& \sum_{j=1}^{n} a_{m,j} x_j \leq h_m
\end{aligned}
$$

In this case we have the presence of linear constrains. This means that the values of the vector x_i cannot assume any arbitrary value. The following needs to be respected:

- Satisfy a set of relationships
- Satisfy the non-negativity condition
  - Only positive values are accepted
  - Values need to belong to the feasible region

The previous linear problem can be re-written by using the following simplified notation:

$$
\begin{array}{rlrl}
\max & z = c^T x & \min & z = c^T x \\
\text{s.t.} & Ax \leq h & \text{s.t.} & Ax \geq h \\
& x \geq 0 & & x \geq 0
\end{array}
$$

Where $z = f(x)$ represents the value of the objective function, c is a vector of coefficients, x is a vector of unknowns, A is the coefficient matrix, and h is the vector coefficients of the constraints. The sizes of these matrices are vectors are:

- $A \rightarrow m \times n$
- $\overrightarrow{c} \rightarrow n$
- $\overrightarrow{x} \rightarrow n$
- $\overrightarrow{h} \rightarrow m$

This is known as the **standard form** of a linear program. After writing the problem in a standard form, we need to graphically plot the feasible region which satisfies all of the constraints.

6.1 Fundamental Theorem of Linear Programming

In order to find the optimal value, we can make use of the **Fundamental Theorem of Linear Programming**.

> If a linear programming problem has a solution, it must occur at a vertex of the set of feasible solutions. If the problem has more than one solution, then at least one of them must occur at a vertex of the set of feasible solutions. In either case, the value of the objective function is unique.

6.2 Simplex Method

The **simplex method** is an algorithm used for solving linear programming problems that have two or more variables. It has an exponential worst-case complexity.

In order to solve a linear programming problem with the simplex method, we need to do the following:

1. Write the problem in standard form
2. Transform inequalities into equalities by adding:
   (a) **Slack Variables** – for maximization
   (b) **Surplus Variables** – for minimization
3. Apply the iterative rule by exchanging the basic and nonbasic variable
4. Repeat step 3 until the optimality criterion is not met

By introducing **slack** and **surplus** variables to the standard form of the linear programming problem we could obtain:

$$\begin{aligned} \max \quad & z = 3x + 2y \\ \text{s.t.} \quad & x + 2y + s_1 = 4 \\ & x - y + s_2 = 1 \\ & x, y \geq 0;\ s_1, s_2 \geq 0 \end{aligned} \qquad \begin{aligned} \min \quad & z = 3x + 2y \\ \text{s.t.} \quad & x + 2y - s_1 = 4 \\ & x - y - s_2 = 1 \\ & x, y \geq 0;\ s_1, s_2 \geq 0 \end{aligned}$$

6.2.1 Basic and Nonbasic Variables

The following statements are true:

- We can freely swap the rows of A, as long as we also swap the elements of vector h
- We can freely swap the columns of A, as long as we also swap the elements of vector x

Matrix A can be split into two sub matrices:

$$A = \begin{bmatrix} B & D \end{bmatrix}$$

Where B is the matrix formed by m linearly independent columns of A, and D is the matrix containing the remaining columns. In the same way, we can split x and c:

$$x = \begin{bmatrix} x_B \\ x_D \end{bmatrix} \qquad c = \begin{bmatrix} c_B \\ c_D \end{bmatrix}$$

Now, we can rewrite the system $Ax = h$ to:

$$\begin{aligned} Bx_B + Dx_D &= h \\ Bx_B &= h - Dx_D \\ x_B &= B^{-1}h - B^{-1}Dx_D \end{aligned}$$

Where x_B are the **basic variables**, and x_D are the **nonbasic variables**. This equation represents a general solution of the linear programming problem.

6.2.2 Basic Solution

If we set $x_D = 0$ in the above equation, we obtain:

$$x_B = B^{-1}h$$

If the non-negativity condition for x_B is satisfied, then this **basic solution** is feasible. In the case that one or more components of x_B have value 0, then the solution is said to be **degenerate**.

The basic solution corresponds to the vertices of the polytope – which represents the feasible region of the linear programming problem.

6.2.3 Optimal Basic Solution

The existence of a feasible solution implies also the existence of an optimal basic solution. Moreover, the existence of an optimal solution implies the existence of an optimal basic solution.

A downside is the fact that the number of possible basic solutions grows exponentially with the number of unknowns and constraints. The maximum possible number of iterations of the simplex method is:

$$N = \frac{(m+n)!}{m!n!}$$

6.3 Optimality Condition

The **optimality condition** that needs to be checked at every iteration of the simplex method is based on the reduced cost coefficients.

$$r_D = c_D - c_B B^{-1} D$$

Where c_B represents the basic coefficients of the objective function, c_D represents the nonbasic coefficients of the objective function, B is the basic matrix, and D is the nonbasic matrix. The optimality conditions are given by $r_D \leq 0$ for a **maximization** problem, and $r_D \geq 0$ for a **minimization** problem.

6.4 Iterative Rule

If the stopping criterion is not met, we select the highest reduced cost coefficient (in the case of maximization), or the lowest reduced cost coefficient (in the case of minimization). This needs to be brought inside of the basis.

For all of the columns of the entering variable, we need to compute the following ratio:

$$\frac{B^{-1}h}{B^{-1}D}$$

Where B^{-1} represents the inverse of the basic matrix, h is the vector coefficients of the constraints, and D represents the non-basic matrix. Among all of the obtained ratios, we need to select the one that has the smallest positive value. This will be the **departing variable**.

6.5 Auxiliary Minimization Problem

By solving an initial **auxiliary minimization problem**, we find a feasible initial basic solution. It can be defined as follows – by introducing artificial variables $u_1, ..., u_m$:

$$\begin{aligned}
\max \quad & z_{\text{aux}} = \sum_{i=1}^{n} u_i \\
\text{s.t.} \quad & \sum_{j=1}^{n} a_{1,j} x_j + s_1 + u_1 = h_1 \\
& \vdots \\
& \sum_{j=1}^{n} a_{m,j} x_j + s_m + u_m = h_m \\
& x_i, ..., x_n \geq 0; \;\; s_1, ..., s_m \geq 0; \;\; u_1, ..., u_m \geq 0
\end{aligned}$$

The auxiliary problem aims at minimizing the sum of the artificial variables. The optimal solution of the auxiliary problem would be achieved when all of the artificial variables have a value of 0.

If z_{aux} does not reach the value 0, then both the auxiliary problem and the original linear programming problem do not admit a feasible solution.