Institut des Sciences et Techniques des Yvelines
CAMPUS DE MANTES EN YVELINES
CAMPUS DE SAINT-QUENTIN-EN-YVELINES

10-12 Av. de l'Europe, 78140
Vélizy-Villacoublay, France

18 Quai du Point du Jour, 92100
Boulogne-Billancourt, France

Prof. Soraya ZERTAL

Dr. Chong LI

Master internship: March 2023 - August 2023

DNN TRAINING ACCELERATION BY SPARSIFYING AND PARALLELIZING COMPUTATION

Eng. Slim KHIARI - Computer Engineering Cycle *"IATIC"*

Acknowledgments

I would like to thank all the staff of the Huawei Paris Research Lab for the support they were able to give me as well as their good humor.

I thank, in particular, Dr. Chong LI, my professional internship supervisor, for allowing me to do this internship in the Huawei Paris laboratory. He was always there to help me to solve the various problems encountered, by making adjustments weekly, while leaving me a great deal of work autonomy.

I thank Prof. Soraya Zertal for being my internship supervisor at ISTY and for helping me find my 4th year internship.

I also thank, with all my heart, Prof. Nahid Emad and PhD. candidate Quentin Petit for accompanying me during this internship.

Finally, I thank Prof. Denis Barthou for accepting to be the co-supervisor of my CIFRE thesis within Huawei.

Contents

Introduction

Training effectively deep learning models requires several conditions. One of these conditions is a solid choice of hyper-parameters. Hyperparameters in machine learning are variables that are established before the learning process starts and regulate certain facets of the model training procedure. These parameters are manually modified or optimized to obtain the optimal configuration for a specific problem, as opposed to learning them from the data like model weights do.

In order to make an appropriate hyper-parameters decision, we need to compare, for each training, the results of many deep learning model training by taking into consideration some metrics like the accuracy or the loss for example. This process could be really time consuming and may lead to an incorrect choice of some hyper-parameters.

The embedding dimension is one of the hyper-parameters of a given machine learning model. It may be described as a dense, low-dimensional vector representation of an object with higher dimensions. The uses for embeddings are numerous. For instance, embeddings are used to represent users and items in recommender systems. In order to provide more relevant recommendations, recommender systems like Wide and Deep can find related users and items. Because they enhance calculations by lowering dimensionality, embeddings are crucial.

The choice of embedding dimensions may be particularly delicate since they can either limit the expressiveness of the information or result in a lengthy training period for machine learning models. Therefore, choosing the suitable embedding dimension is a difficult problem for deep learning.

The work I did during this internship consists of an innovative approach allowing us to choose the right embedding dimension by applying the numerical method, MIRAMns (Multiple Implicitly Restarted Arnoldi Method with nested subspaces) to the recommendation system, Wide & Deep learning. For this, we will first describe the necessary tools that I used to set up this proposal, explain some generalities on neural networks, then explain how the Wide & Deep recommendation system works, then explain the Arnoldi method and its variants (IRAM & MIRAMns), then explain in detail our approach, to finish with some experiments carried out in order to validate our idea on our Wide & Deep learning case study.

Chapter 2 : Company presentation

Huawei announced the creation of a mathematics research center in France during its fourth European Innovation Day in 2016. This is Huawei's second mathematics research center; the first is in Russia. Huawei has established five research centers in France, with a particular emphasis on applied research in fields including wireless communications, artificial intelligence, design, image processing, and sensors.

With the help of the center, Huawei hopes to strengthen its position in fundamental research, notably in the development of mathematical algorithms for ICT (Information and Communications Technology) advancements.

The center is a component of Huawei's strategy to collaborate closely with French academics in order to establish a dynamic academic community centered on ICT growth. It brings France's skills and innovations to the international ICT industry by fusing France's competence in mathematical research with Huawei's prowess in translating research into commerce.

This research center, which is based in Boulogne-Billancourt, France, is devoted to exploring the country's fundamental mathematical resources and carrying out fundamental investigations into the physical and network layers of communications, distributed and parallel computing, data compression and storage, and other fundamental algorithms.

The research center is also concentrating on short-term products and strategic initiatives, as well as creating a general framework for distributed algorithms.

Huawei has built up 16 research centers around the world, counting the France Research Center, beneath which there are four groups separately concentrating on design, digital imaging, mathematics, and home devices.

Chapter 3 : The necessary tools

In this chapter, I will present the different tools that I have chosen to use in order to implement our approach and the usefulness of each in relation to our project. For this, I will present Huawei's AI framework, MindSpore, then the linear algebra library, Eigen, and the distributed computing library, Open MPI.

3.1) MindSpore & Model Zoo for MindSpore

- MindSpore (https://gitee.com/mindspore)

MindSpore, created entirely by Huawei, is an advanced AI computing framework designed to enable seamless collaboration between cloud, edge, and device environments. It offers unified APIs and comprehensive AI functionalities for model development, execution, and deployment in various scenarios.

Employing a distributed architecture, MindSpore utilizes a native differentiable programming paradigm and innovative AI native execution modes to enhance resource efficiency, security, and reliability. Furthermore, it optimizes the utilization of Ascend AI processors, making AI development more accessible to industries and accelerating the realization of inclusive AI.

- Model Zoo for MindSpore (https://gitee.com/mindspore/models)

The MindSpore models repository provides different task domains, classic SOTA model implementations and end-to-end solutions. The purpose is to make it easier for MindSpore users to use MindSpore for research and product development . In order to facilitate developers to enjoy the benefits of MindSpore framework, typical networks and some of the related pre-trained models will be added.

I used MindSpore and Model Zoo from MindSpore so I could train the Wide & Deep network on the Ascend processors. For more information on the MindSpore installation procedure, please refer to the Appendix section.

3.2) Eigen library

(https://eigen.tuxfamily.org/index.php?title=Main_Page)

Eigen is a comprehensive C++ library consisting of template headers designed for performing a wide range of mathematical operations, including linear algebra, matrix and vector manipulations, geometrical transformations, numerical solvers, and related algorithms.

One of Eigen's notable features is its implementation using the expression templates metaprogramming technique. This approach enables Eigen to build expression trees at compile time and generate specialized code to efficiently evaluate these expressions. By utilizing expression templates and considering the cost of floating-point operations, the library optimizes its performance through loop unrolling and vectorization. Additionally, Eigen can provide interfaces to BLAS (Basic Linear Algebra Subprograms) and a subset of LAPACK (Linear Algebra PACKage) functionality.

I used the Eigen library in order to be able to develop the MIRAMns method of our approach.

3.3) Open MPI

(https://www.open-mpi.org/)

An alliance of academic, scientific, and business partners created and maintains the Open MPI Project, an open source implementation of the Message Passing Interface. Therefore, in order to create the finest MPI library possible, Open MPI is able to integrate the knowledge, tools, and resources from throughout the High Performance Computing community. For system and software providers, application developers, and computer science academics, Open MPI offers benefits.

I used Open MPI in order to be able to integrate my work with the part of PhD. candidate Quentin Petit (who works on the matrix build part of this project).

Chapter 4 : Some generalities about neural networks

Before moving on to the case study and the training improvement proposal, it is important to understand a few concepts. In this chapter, we will therefore define neural networks, then we will understand the components of a neural network in a little more detail to end up explaining the concept of embedding which is the subject of this internship report.

4.1) Neural Networks

An algorithm known as a neural network imitates the structure and operation of the human brain. They are composed of linked "neurons" and were created to mimic biological brain networks. In general, neural networks are designed to provide an artificial system that can process and evaluate data in a manner akin to that of the human brain.

Neural networks come in a variety of forms, each with unique properties and uses. The sort of neurons that make them up and how information travels through the network are their primary differences.

In general, we may categorize them into three groups:

- Regular neural networks with complete connectivity
- Convolutional neural networks
- Recurrent neural networks

The different types of neural networks have elements in common such as; artificial neurons and layers. I will therefore detail these main components in the part below..

4.2) Components of Neural Networks

- Artificial Neurons

A neural network's fundamental building pieces are artificial neurons. They are structures based on biological neurons. Every artificial neuron takes inputs and sends a single output to a network of other neurons.

The majority of the time, inputs are outputs from other neurons, but they can also be numerical values from a sample of outside data. In essence, a neuron gets an input value, does a little computation on it, and sends the answer to the neurons in front of it.

The weighted sum is the easiest way to mathematically represent an artificial neuron :

$$z = w_1 x_1 + \dots + w_k x_k + b,$$
$$with \quad w_i \ are\ weights,\ x_i\ are\ inputs\ and\ b\ bias.$$

Next, a weighted sum z representing the neuron's final output is subjected to an activation function f as shown in the following diagram :

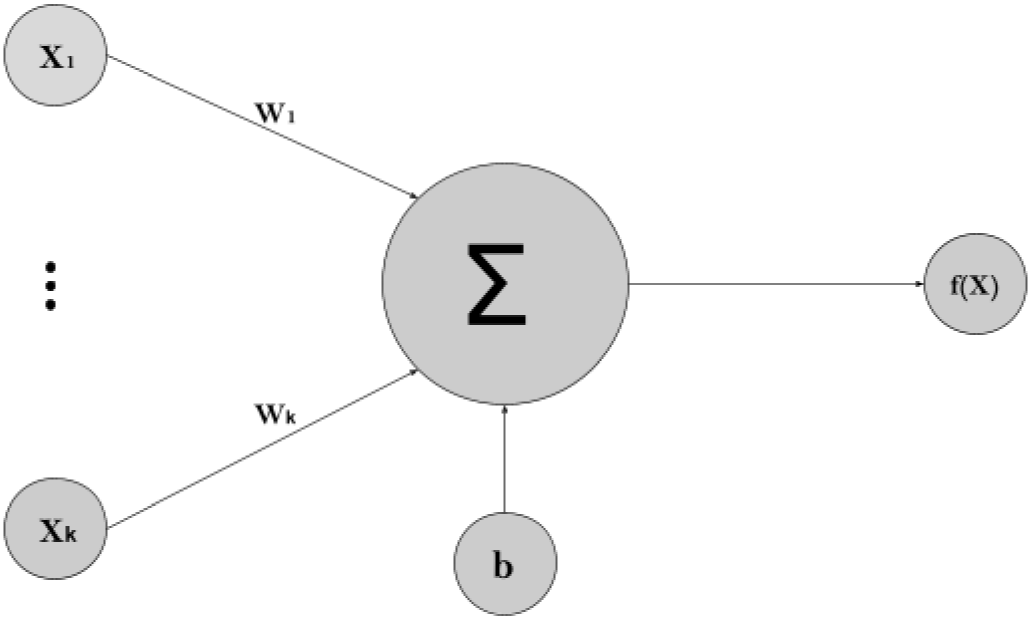


Figure 1 - The functioning of an artificial neuron

- Layers

A layer in a neural network is a cluster of neurons that work together to complete a single job. Each layer of linked neurons in a neural network carries out a specific function.

There are three different sorts of layers depending on where they are in a neural network:

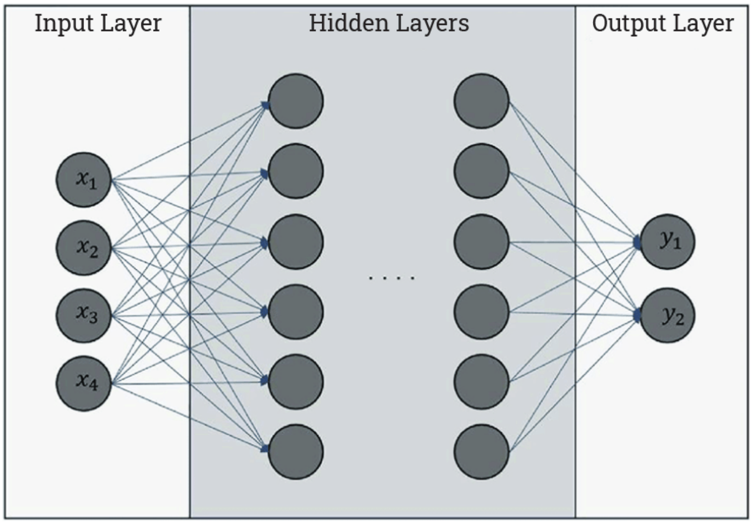


Figure 2 - The architecture of a neural network
(https://www.baeldung.com/cs/neural-nets-embedding-layers)

1. The **input layer** (first layer in a neural network) is in charge of obtaining input data and transmitting it to the following layer.

2. All but a few single-layer neural network types, such as the perceptron, have **hidden layers**. A neural network may have several hidden layers. The difficulty of the issue being solved will determine how many hidden layers there are and how many neurons there are in each layer.

3. The final layer of a neural network that generates the final output (or prediction) is called the **output layer**.

4.3) Embedding layers

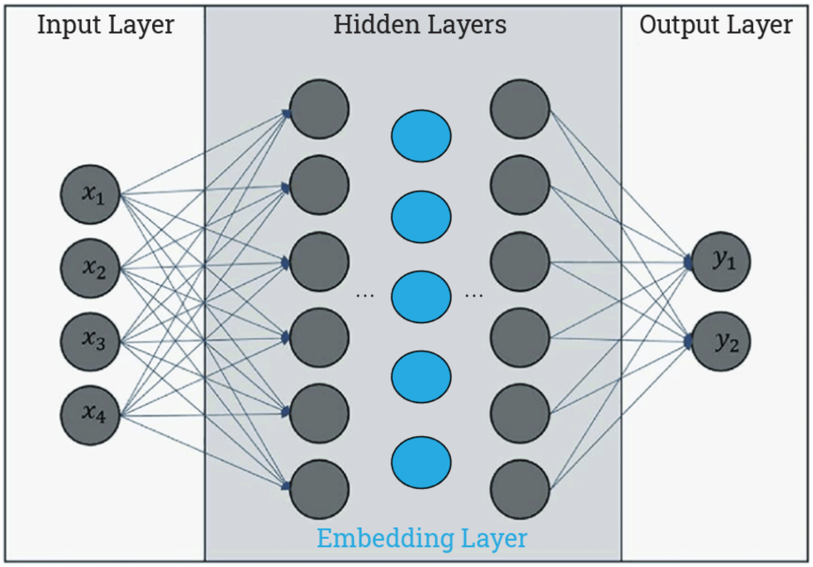


Figure 3 - The architecture of a neural network with an embedding layer
(https://www.baeldung.com/cs/neural-nets-embedding-layers)

In a neural network, an embedding layer, as shown in the diagram above, is a particular kind of hidden layer. This layer converts input data from a high-dimensional to a lower-dimensional space, enabling the network to better understand how inputs relate to one another and process data more quickly.

As an illustration, in natural language processing (NLP), words and phrases are frequently represented as one-hot vectors, where each dimension is a separate word from the lexicon. These vectors are sparse and high-dimensional, which makes it challenging to manipulate them.

The embedding layer might convert each word to a low-dimensional vector instead of utilizing these high-dimensional vectors, where each dimension reflects a different aspect of the word.

4.4) How does embedding work ?

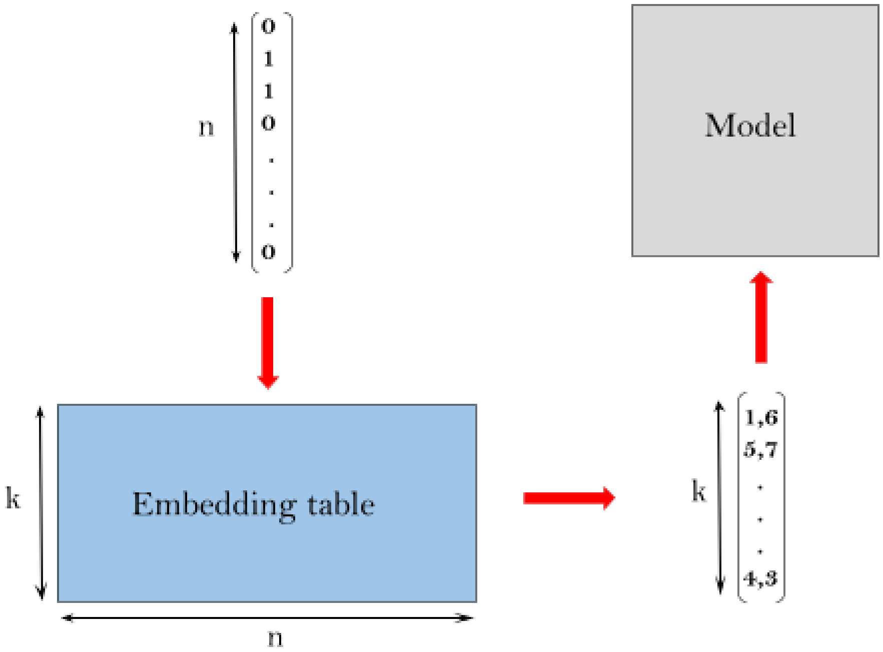


Figure 4 - Example of embedding : from dimension n to k with k < n

As shown in the diagram below, embedding works by using an embedding table to transform data. This reduces the size of a vector of size n to a vector of size k by multiplying the vector of size n by the embedding table. Note that this embedding table is initialized randomly.

4.5) Types of Embedding layers

The neural network and the embedding procedure determine the kind of embedding layer. There are various different kinds of embedding:

- Text embedding

  The most popular kind of embedding is undoubtedly text embedding. This is a result of the widespread use of language models and transformers. A language model built by the OpenAI team is called ChatGPT. It features GPT-3 architecture and is made up mostly of an encoder and a decoder.

The encoder component converts the input text into an embedding vector, which is then used by the decoder section to produce the response and convert it back to the original input text. It's important to note that models built with transformers provide contextual embeddings. It implies that if a word appears in a different context, it will probably receive a different embedding vector. Contrarily, certain well-liked non-contextual architectures exist ; GloVe and word2vec.

- Image embedding

A method for encoding pictures as dense embedding vectors is image embedding. These vectors record part of the picture's visual elements, and we may utilize them for operations like object recognition and image categorization.

We can create picture embeddings using a few well-known convolutional neural networks (CNNs) that have already been trained. NFNets, EfficientNets, ResNets, and other networks are a few of them. Visual transformer models have lately grown in prominence in addition to CNNs.

- Graph embedding

We employ graph embedding to represent graphs as dense embedding vectors in a lower-dimensional space, much as how pictures are represented. Furthermore, node embedding, where the objective is to build an embedding vector for each individual node rather than for the entire network, is frequently connected to graph embedding.
Popular graph embedding algorithms include the following:
DeepWalk Node2vec, Locally Linear Embedding (LLE), and others.

Chapter 5 : Wide and deep learning for recommender systems

In order to apply our approach, we have chosen a deep learning case study; wide and deep learning. We chose this model for several reasons, in particular because it is relatively easy and quick to train, and above all because it is finalized in terms of code on the MindSpore Huawei's framework. In addition, we chose this recommendation system because it is the reference for this type of system in the community. It is therefore important in this chapter to explain this model in order to have a general understanding of how it works.

In order to better understand how this type of recommendation system works, we will take the example of the following application: the only thing an app user needs to do is speak out loud the type of cuisine they are desiring (the query). The dish (the item) is mysteriously delivered to the user's front door by the app after it guesses which meal they would enjoy the most. Consumption rate is our primary statistic; if a dish was consumed by the user, the score is 1; otherwise, the score is 0 (the label).

5.1) Memorization (the wide component)

Learning the frequent co-occurrence of things or qualities and making use of the connection offered by the past data might be a rough definition of memory. A large variety of cross-product feature transformations may be used to effectively and comprehensively memorize feature interactions. Memorization-based recommendations are often more current and pertinent to the objects on which users have already taken action. Cross-product transformations over sparse features can be used to efficiently memorize data. This illustrates how a feature pair's co-occurrence and the target label's correlation. The inability of cross-product transformations to generalize to query-item feature pairings that are absent from the training set is one of its drawbacks. Utilizing cross-product feature transformations, wide linear models may efficiently memorize sparse feature interactions.

For our example, as shown in Figure 5 below, the model discovers that while the character match is greater for *AND(query="fried chicken", item="chicken and waffles")*, it doesn't receive as much love as it does for *AND(query="fried chicken", item="chicken fried rice")*. In other words, the application starts to gain more popularity because it performs an excellent job of remembering what users enjoy.

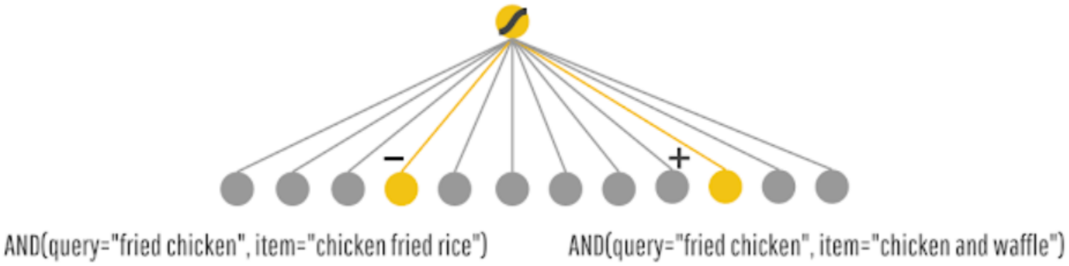

−
+
AND(query="fried chicken", item="chicken fried rice")
AND(query="fried chicken", item="chicken and waffle")

Figure 5 - Diagram showing an example of how memorization works
(*https://ai.googleblog.com/2016/06/wide-deep-learning-better-together-with.html*)

A generalized linear model using the formula $y = w \cdot T \cdot x + b$ with

$$\begin{cases} \text{y : the prediction} \\ \text{x = [x1, x2, …, xd] : the vector of d features} \\ \text{w = [w1, w2, …, wd] the model parameters} \\ \text{b : the bias} \end{cases}$$

describes the wide component.

The feature set consists of both transformed and raw input information. The cross-product transition, which is outlined as follows, is one of the most significant transformations :

$$\Phi_k = \prod_{i=1}^{d} x_i^{c_{ki}} \quad , \quad c_{ki} \in \{0, 1\}$$

5.2) Generalization (the deep component)

The deep component is a feed-forward neural network. The initial inputs for categorical features are feature strings. Each of these sparse, high-dimensional categorical characteristics is first transformed into an embedding vector, which is a low-dimensional, dense real-valued vector. The embeddings typically have a dimensionality between O(10) and O(100). The embedding vectors are given a random initialization before being trained to minimize the overall loss function.

The hidden layers of a neural network are then given these low-dimensional dense embedding vectors in the forward pass. Each hidden layer specifically carries out the following calculation:

$$a^{(l+1)} = f(W^{(l)} \cdot a^{(l)} + b^{(l)}) \quad \text{with}$$

l : the layer number
f : the activation function
a(l) : the activations at l-th layer
W(l) : the model weights at l-th layer
b(l) : the bias at l-th layer

Low dimensional embeddings enable deep neural networks to generalize to previously unknown feature interactions.

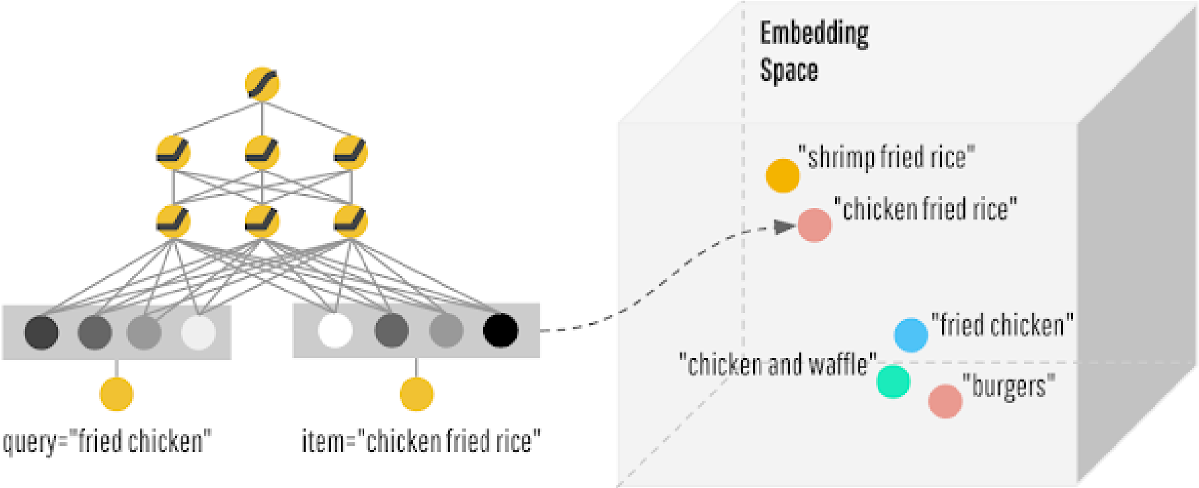


Figure 6 - Diagram showing an example of how generalization works
(*https://ai.googleblog.com/2016/06/wide-deep-learning-better-together-with.html*)

You can see from the example below that individuals who request "fried chicken" don't mind getting "burgers".

5.3) Joint training of the wide and deep component

By accounting for the wide and deep parts as well as the weights of their total during training, joint training concurrently optimizes all characteristics.

A weighted sum of the output log odds from the wide and deep components is used to combine them, and this prediction is then given to a single logistic loss function for joint training. Backpropagating the gradients from the output to both the wide and deep component of the model at the same time using mini-batch stochastic optimization is how a Wide & Deep Model is trained together.

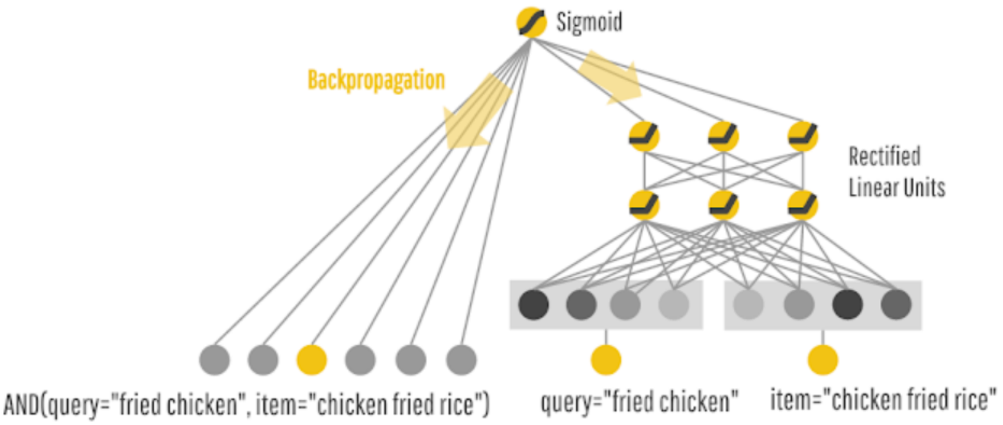

Sigmoid
Backpropagation
Rectified
Linear Units
AND(query="fried chicken", item="chicken fried rice")
query="fried chicken"
item="chicken fried rice"

Figure 7 - Diagram showing an example of how Combining Wide and Deep models works (*https://ai.googleblog.com/2016/06/wide-deep-learning-better-together-with.html*)

As can be seen in the figure 7, the wide and deep parts both employ sparse features like query="fried chicken" and item="chicken fried rice". The model parameters are trained by backpropagating the prediction errors to both sides during training. The deep model component may use embeddings to generalize to comparable things whereas the wide model component can memorize all those sparse, specialized rules.

Chapter 6 : Arnoldi's method and its varieties

In this chapter, we will justify our choice of using MIRAMns. We will explain it theoretically, then we will explain our 2 versions of its algorithms (sequential and distributed). Note that the distribution version in this case is only in order to be able to run our MIRAMns implementation because of the large initial matrix and not to optimize it. The optimization of our implementations is therefore a future work to be done. We will indicate how we approached it in order to test and validate our IRAM as well as MIRAMns.

6.1) What is the best version of Arnoldi for our proposal ?

6.1.1) Why are the power method and deflation bad?

The power method will only converge to one eigenvalue of a given matrix. Throughout the repetition, this method discards spectral data that may be valuable. After the k^{th} iteration, we change the vector $A^{k-1} \times v_0$ to $A^k \times v_0$ where v_0 is the algorithm's starting vector. However, keeping the preceding vector rather than overwriting it, and hence keeping the whole collection of prior vectors, proves to be advantageous { v_0, $A \times v_0$, $A^2 \times v_0$, ..., $A^{k-1} \times v_0$ }. We call the subspace $K_k(A, v_0) = span\{v_0, A \times v_0, A^2 \times v_0, ..., A^{k-1} \times v_0\}$ the k^{th} Krylov subspace corresponding to A and v_0. Krylov subspace or projection methods are techniques that employ linear combinations of vectors in this space to extract spectral information. The fundamental concept is to build approximations of the eigenvectors in the Krylov subspaces $K_k(A, v_0)$.

The best method to find the k dominating eigenvalues of a given matrix may be improved in a number of ways.

One method, called deflation, is fairly simple. After computing the first eigenpair, a transformation is applied to the given matrix to shift the first eigenvalue to the interior of the spectrum, making the second eigenvalue the dominant eigenvalue of the transformed matrix. Up until the k dominating eigenvalues are identified, this process is repeated.

Since finding one eigenpair at a time is not very efficient, deflation can be improved by applying a given matrix on a collection of k beginning vectors at each iteration, we expand the power method. The iterates will all converge to the dominating eigenvector when handled independently. The set will eventually become the basis for an invariant subspace of the given matrix corresponding to the k eigenvectors with the biggest magnitude if we orthonormalize the k vectors at each step. This technique is referred to as **subspace iteration**.

We will apply **the Arnoldi Method**, a powerful extension of subspace iteration, to simultaneously identify all k eigenpairs of the given matrix.

6.1.2) Why is MIRAMns the best Arnoldi method for our situation ?

Having shown that the Arnoldi method is better than deflation, there is however a set of variations of this method. It is important to look for the best.

Let's start with Arnoldi's explicit restart method. Indeed, according to the paper "MULTIPLE EXPLICITLY RESTARTED ARNOLDI METHOD FOR SOLVING LARGE EIGENPROBLEM" published by NAHID EMAD , SERGE PETITON, AND GUY EDJLALI, MERAM converges faster than ERAM. MERAM is therefore better than ERAM. Still, according to the same paper, IRAM is a more robust method than ERAM. This is because of the powerful restarting strategy used in IRAM.

Now according to the paper "A KEY TO CHOOSE SUBSPACE SIZE IN IMPLICITLY RESTARTED ARNOLDI METHOD" published by S. A. SHAHZADEH FAZELI, N. EMAD , AND Z. LIU, MIRAMns converges faster than IRAM. We will therefore use MIRAMns for our solution.

Since MIRAMns is based on IRAM, we will explain in detail how IRAM works then we'll move on to how MIRAMns works.

6.2) IRAM and its components

6.2.1) The Arnoldi factorization

The Arnoldi factorization's main goal is to create a large matrix A's eigenpairs from those of the small matrix H. A connection of the type $A \cdot V = V \cdot H + f \cdot e_k^T$ is what we refer to as a k-step Arnoldi factorization of $A \in C^{nxn}$ where,

$V \in C^{nxk}$: has orthonormal columns,

$V^H \cdot f = 0$, and

$H \in C^{kxk}$: upper Hessenberg with a non-negative subdiagonal

Using the following algorithm, we can extend a k-step Arnoldi factorization to a (k+p)-step Arnoldi factorization:

Algorithm 1 : The Arnoldi factorization allowing to manage both k-step and (k+p)-step factorization

Inputs

A: the large initial square matrix

startStep : the starting position of the factorization

numVector : the subspace size

Outputs

f: the residual vector

V : the matrix of orthogonal basis

H : the projected matrix

Local variables

β, α, v, w and h : local variables of this algorithm

```
If (startStep = 1) then
    β ← norm of vector f
    $v = \frac{f}{\beta}$
    $w = A \cdot v$
    α = adjoint of v · $w$
    $f = w - v \cdot \alpha$
    Update the first column of the matrix V by the vector v
    Update the 1st value of the matrix H with α

For j=startStep to numVectors do
    β ← norm of vector f
    v = $\frac{f}{\beta}$
    $H_{j-1, j-2} = \beta$
    $V_{j-1} = v$
    $w = A \cdot v$
    h =adjoint of $V_{:, 1:j} \times w$
    f = w − $V_{:, 1:j} \cdot$ h
    $H_{0:j, j-1}$ = h
```

Remember that our main problem is to apply MIRAMns on a very large square matrix of size 200,000 (the matrix named A). This therefore represents good memory management because we cannot simply declare a square array of dimension 200,000 directly. Since MIRAMns is based on Algorithm 1, Arnoldi factorization, it is therefore essential to distribute calculations while minimizing communications. Let us also remember that having redundant calculations is better than having a lot of communications. In this case, the line that poses a problem for us in algorithm 1 is the following: $w = A \times v$**.**

In order to remedy this problem, we will take pieces of matrix A. Each piece will be assigned to a process. Then, we will apply the Arnoldi factorization on these pieces (i.e. each process will apply the Arnoldi factorization). In Arnoldi's factorization algorithm, each time we have a calculation that involves the matrix A, each process will partially do the calculation with its piece of the matrix A. When each process finishes its calculation with the piece of the matrix A, it sends its partial result to the other processes.

Thus, all the processes will have in memory the final calculation resulting from the intervention of the matrix A without storing it entirely. Our algorithm 1 in distributed mode becomes (the parts that change are in yellow):

Algorithm 2 : The distributed Arnoldi factorization allowing to manage both k-step and (k+p)-step factorization

Inputs

submatrix_A: The part of matrix A

startStep : the starting position of the factorization

numVector : the subspace size

Outputs

f: the residual vector

V : the matrix of orthogonal basis

H : the projected matrix

Local variables

tmp, β, α, v, w and h : local variables of this algorithm

```
If (startStep = 1) then
    β ← norm of vector f
    $v = \frac{f}{\beta}$
    tmp = submatrix_A × v
    MPI_Allgather(tmp, size of tmp vector, MPI_DOUBLE_COMPLEX, w, size of tmp vector, MPI_DOUBLE_COMPLEX, MPI_COMM_WORLD)
    α = adjoint of v · $w$
    $f = w - v \cdot \alpha$
    Update the first column of the matrix V by the vector v
    Update the 1st value of the matrix H with α

For j=startStep to numVectors do
    β ← norm of vector f
    $v = \frac{f}{\beta}$
    $H_{j-1, j-2} = \beta$
    $V_{j-1} = v$
    tmp = submatrix_A × v
    MPI_Allgather(tmp, size of tmp vector, MPI_DOUBLE_COMPLEX, w, size of tmp vector, MPI_DOUBLE_COMPLEX, MPI_COMM_WORLD)
    h =adjoint of $V_{:, 1:j} \times w$
    f = w − $V_{:, 1:j} \cdot$ h
    $H_{0:j, j-1} = $ h
```

6.2.2) The Ritz peers

Ritz estimate is used to calculate the error of the Arnoldi projection (algorithm 1/ algorithm 2) with only an elementary mathematical operation.

Remember that H is the projected matrix, and V is the matrix of orthogonal basis, both issued from the Arnoldi factorization (algorithm 1/ algorithm 2). If (y, θ) is an eigenpair of H, then x = V·y satisfies the relation :

$$\begin{aligned} ||A \cdot x - x \cdot \theta|| &= ||A \cdot V \cdot y - V \cdot y \cdot \theta|| \\ &= ||(A \cdot V - V \cdot H) \cdot y|| \\ &= ||f \cdot e_k^T \cdot y|| \\ &= \beta \cdot |e_k^T \cdot y| \end{aligned}$$

where $\beta = ||f||$ and e_k^T the transpose of the canonical vector of size k . Thus, (x, θ) is called the Ritz peers with θ being the Ritz value and x the Ritz vector. The term $\beta \cdot |e_k^T \cdot y|$ is the Ritz estimate of the pair (x, θ) . It explains how well the eigenpair approximation works. The Ritz values and vectors are precisely the eigenvalues and eigenvectors of A, and V is an invariant subspace of A, when f = 0.

Here is our Ritz peer calculation algorithm:

Algorithm 3 : The Ritz Peer Calculation Algorithm

Inputs

nrmfr : norm of matrix A
k : number of desired eigen elements
eigenvalues_Hmm : the vector that contains the eigenvalues of H issued from the Arnoldi factorization
eigenvectors_Hmm : the matrix that contains the eigenvectors of H issued from the Arnoldi factorization
V : the matrix of orthogonal basis issued from the Arnoldi factorization

Outputs

U_desired : the matrix that contains the ritz eigen vectors of the A matrix
desired_eigenvalues_Hmm : the vector that contains the desired eigen values of Hmm
residuals : the residual vector

```
j=0

For  i = size(eigenvalues_Hmm)  - 1; i > size(eigenvalues_Hmm)-1-k; i -- do
    desired_eigenvalues_Hmm(j) = eigenvalues_Hmm(i)
    desired_eigenvectors_Hmm.col(j) = eigenvectors_Hmm.col(i)
    j++

U_desired = V × desired_eigenvectors_Hmm;

For  i = 0; i < k; i ++ do
    residuals(i) = norm of f  × |desired_eigenvectors_Hmm(the number of rows of desired_eigenvectors_Hmm - 1, i)|
```

6.2.3) The QR Method

The QR method is a version of the iteration subspace. The diagonal entries of H become a more accurate representation of the eigenvalues of the matrix A as the QR method advances. The diagonal elements of H are better approximations to the eigenvalues of A as the magnitude of the subdiagonal entries of H decreases. If the product of the orthogonal matrices Q_i is gathered and saved at each step, the eigenvectors can be retrieved afterwards.

Here is the algorithm of the QR method:

Algorithm 4 : The QR method

Inputs

A : the initial matrix

V : the matrix of orthogonal basis issued from the Arnoldi factorization

H : the projected matrix issued from the Arnoldi factorization

With $A \cdot V = V \cdot H$ and $V^H \cdot V = I$

Outputs

The V and H matrices such that $A \cdot V = V \cdot H$, $V^H \cdot V = I$ and H is upper triangular

```
For i = 1, 2, 3, ..., until convergence do
    Factor $H_i = Q_i R_i$
    $H_{i+1} = R_i Q_i$
    $V_{i+1} = V_i Q_i$
```

However, there is another version of the QR method, the explicitly shifted QR method. It introduces a shift σ_i at each iteration and it is more useful. It is this version that we will use for our implementation of IRAM and MIRAMns.

Algorithm 5 : The explicitly shifted QR method

Inputs

A : the initial matrix

V : the matrix of orthogonal basis issued from the Arnoldi factorization

H : the projected matrix issued from the Arnoldi factorization

With $A \cdot V = V \cdot H$ and $V^H \cdot V = I$

Outputs

The V and H matrices such that $A \cdot V = V \cdot H$, $V^H \cdot V = I$ and H is upper triangular

```
For i = 1, 2, 3, ..., until each subdiagonal element of $H$ do
    Select a shift $\sigma_i$
    Factor $H_i - \sigma_i \times I = Q_i R_i$
    $H_{i+1} = Q_i^H \times H_i \times Q_i$
    $V_{i+1} = V_i \times Q_i^H$
```

However, the QR method is not appropriate for finding k selected eigenpairs of a large sparse matrix because it has a set of drawbacks :

- The sparsity and structure of the matrix are destroyed by the QR method's transformation.
- Some of the eigenvalues can't be calculated using the QR technique, but not all of them.
- The original matrix or the QR-transformations must be retained if eigenvector information is also requested.

6.2.4) The final algorithm

In general, we like the beginning vector v_0 used to start the Arnoldi factorization to have extremely tiny components in the direction of the other eigenvectors and to be rich in the subspace covered by the desired eigenvectors. As we gain a clearer understanding of the required eigenvectors, we'd like to adaptively refine v_0 to be a linear combination of those approximations and retry the Arnoldi factorization using this updated vector. The implicitly restarted Arnoldi approach, based on the implicitly shifted QR factorization, provides a simple and reliable way to accomplish this without explicitly computing a new Arnoldi factorization.

Let be an m-step Arnoldi factorization of the form to which we aim to apply the shift σ :

$$AV = VH + fe_m^T$$

Since $H \in C^{m\times m}$ is small, we can factor $H - \sigma I = QR$.

Based on the 2 previous equations, we have the following equivalent statements:

$$(A - \sigma I)V - V(H - \sigma I) = fe_m^T$$

$$(A - \sigma I)V - VQR = fe_m^T$$

$$(A - \sigma I)V - VQRQ = fe_m^TQ$$

$$A(VQ) - (VQ)(RQ + \sigma I) = fe_m^TQ$$

$$AV_+ = V_+H_+ + fe_m^TQ$$

V_+ has orthonormal columns since it is the product of V and an orthogonal matrix Q. H_+ turns out to be upper Hessenberg as well. So we can say that shifting by σ does not change the Arnoldi factorization's structure. The result of these operations is that the first column of V_+ is $(A - \sigma I) \times v_1$ where $v_1$is the first column of V .

The idea of IRAM is to extend a k-step Arnoldi factorization $AV_k = V_kH_k + f_ke_k^T$, to a (k+p)-step Arnoldi factorization $AV_{k+p} = V_{k+p}H_{k+p} + f_{k+p}e_{k+p}^T$.

Then p implicit shifts are applied to the factorization, resulting in the new factorization : $AV_+ = V_+H_+ + f_{k+p}e_{k+p}^TQ$, where $V_+ = V_{k+p}Q$, $H_+ = Q^HH_{k+p}Q$, and $Q = Q_1Q_2... Q_p$ where Q_i is associated with factoring $(H - \sigma_iI) = Q_iR_i$.

It turns out that the first $k - 1$ entries of e_{k+p}Q are zero. So, a new k-step Arnoldi factorization can be obtained by equating the first columns on each side k:

$$AV_k^+ = V_k^+H_k^+ + f_k^+e_k^T.$$

Applying shifts, we repeatedly expand this new k-step factorization to a (k+p)-step factorization.

Here is the final IRAM algorithm:

Algorithm 6 : The implicitly Restarted Arnoldi Method - IRAM

Inputs

A : the initial matrix

V_m : the matrix of orthogonal basis issued from the Arnoldi factorization

H_m : the projected matrix issued from the Arnoldi factorization

f_m : the residual vector

With $A \cdot V_m = V_m \cdot H_m + f_m e_m^T$ an m-step Arnoldi factorization

Outputs

Approximated k wanted eigen-elements of A

```
For i = 1, 2, 3, …, until convergence do
    Compute $\sigma(H_m)$ the eigen-elements of $H_m$
    Compute residual norm, if convergence stop
    Select set of $p = m - k$ shifts $(\mu_1^{(m)}, \ldots, \mu_p^{(m)})$, based upon $\sigma(H_m)$ and set $q^T = e_m^T$
    For j = 1, 2, 3, …, p do
        Factor $[Q_j, R_j] = \mathrm{qr}(H_m - \mu_j^{(m)} I)$
        $H_m = Q_j^H H_m Q_j$
        $V_m = V_m Q_j$
        $q^H = q^H Q_j$
    $f_k = v_{k+1}\beta_k + f_m\sigma_k$
    $V_k = V_m(1{:}n,\ 1{:}k)$
    $H_k = H_m(1{:}k,\ 1{:}k)$
    Beginning with the k-step Arnoldi factorization $A \cdot V_k = V_k \cdot H_k + f_k e_k^T$, apply $p$ additional steps of the
        Arnoldi process to obtain a new m-step Arnoldi factorization $A \cdot V_m = V_m \cdot H_m + f_m e_m^T$.
```

The diagram above summarizes the main steps of the above algorithm:

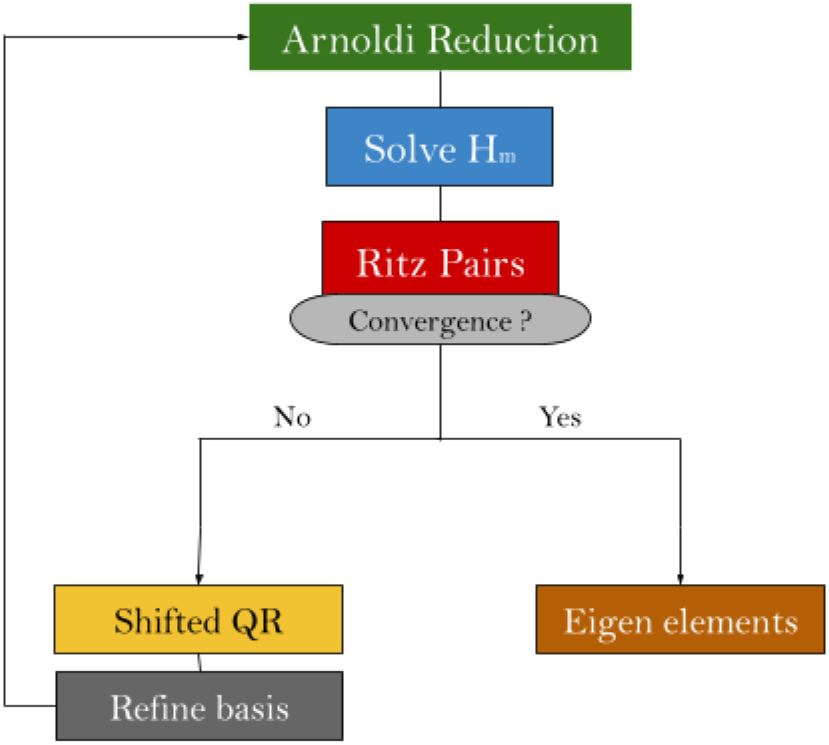


Figure 8 - Diagram summarizing the main steps of IRAM

However, as mentioned before, a convergence improvement has been proposed by the following paper: A KEY TO CHOOSE SUBSPACE SIZE IN IMPLICITLY RESTARTED ARNOLDI METHOD by S. A. SHAHZADEH FAZELI , N. EMAD , AND Z. LIU.

6.3) MIRAMns - Multiple Restarted Implicitly Restarted Arnoldi Method with nested subspaces

The improvement that MIRAMns brings to IRAM is at the level of the blue square in the diagram below. Indeed, with MIRAMns, we calculate the eigen-elements with different sizes from H, in other words with nested sizes. Then, according to the error, we take the best size of H with the variables that go with it.

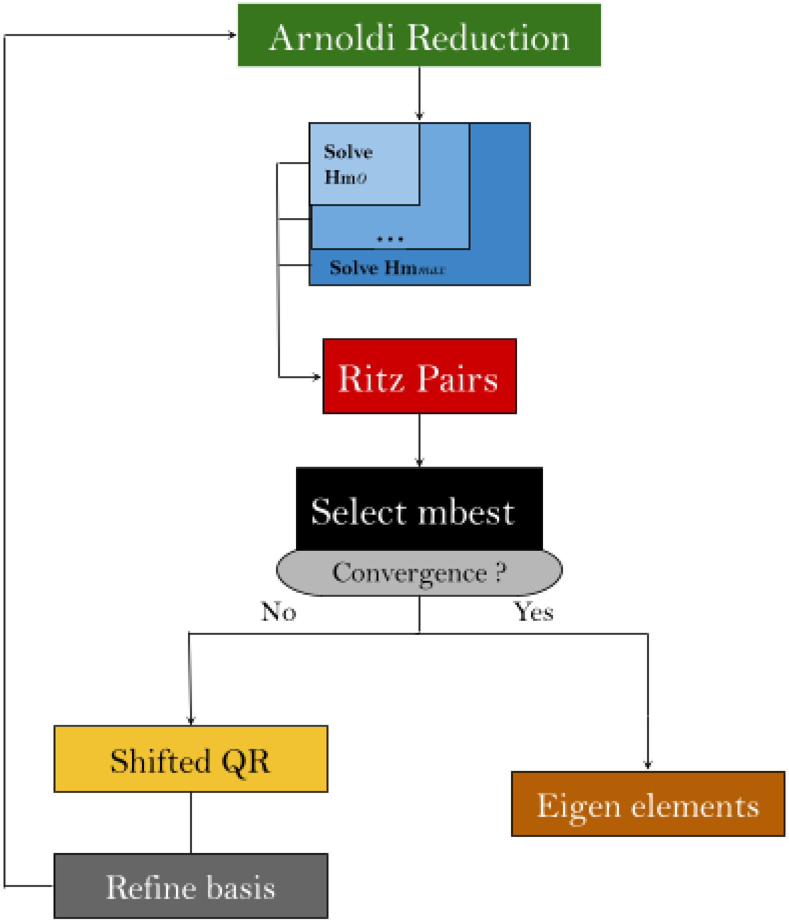


Figure 9 - Diagram summarizing the main steps of MIRAMns

Here is the MIRAMns algorithm:

Algorithm 6 : The Multiple implicitly Restarted Arnoldi Method with nested subspaces

Inputs

A : the initial matrix

V_{m_i} : the matrix of orthogonal basis issued from the Arnoldi factorization

H_{m_i} : the projected matrix issued from the Arnoldi factorization

f_{m_i} : the residual vector

With $A \cdot V_{m_i} = V_{m_i} \cdot H_{m_i} + f_{m_i} e_{m_i}^T$ an m_i-step Arnoldi factorization

Outputs

Approximated k wanted eigen-elements of A

```
For i = 1, 2, 3, …, until convergence do
    ●  Compute $\sigma(H_{m_t})$ the eigen-elements of $H_{m_t}$ for t = (1, …, l)
    ●  Compute residual norm, if convergence in one of subspaces then stop
    Select the best results in these subspaces and the associated best subspace size $m_{best}$, $H_m = H_{m_{best}}$, $V_m = V_{m_{best}}$, $f_m = f_{m_{best}}$
    Select set of $p = m - k$ shifts $(\mu_1^{(m)}, \dots, \mu_p^{(m)})$, based upon $\sigma(H_m)$ and set $q^T = e_m^T$
    For j = 1, 2, 3, …, p do
        Factor $[Q_j, R_j] = \mathrm{qr}(H_m - \mu_j^{(m)} I)$
        $H_m = Q_j^H H_m Q_j$
        $V_m = V_m Q_j$
        $q^H = q^H Q_j$
    $f_k = v_{k+1}\beta_k + f_m \sigma_k$
    $V_k = V_m(1{:}n,\ 1{:}k)$
    $H_k = H_m(1{:}k,\ 1{:}k)$
    Beginning with the k-step Arnoldi factorization $A \cdot V_k = V_k \cdot H_k + f_k e_k^T$, apply $p$ additional steps of the
        Arnoldi process to obtain a new m-step Arnoldi factorization $A \cdot V_m = V_m \cdot H_m + f_m e_m^T$.
```

6.4) Implementations and tests

All the algorithms mentioned before have been programmed and are currently in a private github repository. It will be public soon. In order to verify that our algorithms give us the correct results, we checked their convergence with a Matlab implementation of the paper A KEY TO CHOOSE A SUBSPACE SIZE IN IMPLICITLY RESTARTED ARNOLDI METHOD.

In the appendix 3, *Test document of IRAM and MIRAMns*, several tests with different types of matrices and by varying the parameters of IRAM and MIRAMns.

Chapter 7 : Proposal for improving the case of wide and deep learning

After explaining how MIRAMns and Wide & Deep learning work, we will now explain and justify in this chapter how MIRAMns can be a good tool for efficient embedding dimension selection for this kind of models. We will first demonstrate our innovation analogically, then we will explain how we implement our solution and on which platforms, then we will discuss some results of our approach in the case of Wide & Deep. Finally, we will see if we can generalize our proposal on transformers like GPT for example. We will also see the tracks on which it is absolutely necessary to continue working in order to have a relatively complete work.

7.1) Proof by analogy

Let’s assume that we have a dataset D (square for simplicity) with n rows and n columns, representing n samples that are each defined by n attributes. Let's say that we wish to shrink from D to s, or $ED\ (Embedding\ Dimension) = s$. As a result, we should limit the number of features to s rather than n. To do this reduction, we compute the s major axes that best reflect all n columns in our dataset using one of the dimension reduction techniques.

IRAM makes it possible to calculate these s main axes (s eigenvectors associated with the s dominant eigenvalues of D). To do this, it projects the problem of size n into a subspace of size m (m large enough to be able to contain the s eigenvalues sought; for example $m \geq 2 \times s$). Be aware that the s vectors must be calculated in order to achieve the goal.

MIRAMns improves the performance of IRAM by searching for the s eigenvectors in several nested subspaces instead of just one whose dimension of the smallest of them is at least $2 \times s$ (for example $m_1 = 2 \times s \prec m_2 \prec \ldots \prec m_{max}$). This improvement consists in detecting what is the best size of subspace in which one can find these dominant eigenvectors.

As the conclusion to the preceding point, let $n = 10^6$, $s = 5$, $m_1 = 30$, $m_2 = 40$, and $m_3 = 80$ The shortened dataset (with $ED = 5$) is represented by these five vectors of length 10^6 if the five dominant eigenvectors and five major axes are calculated using MIRAMns.

In order to better understand this analogy, the table below summarizes this connection between the MIRAMns numerical method and Embedding.

	IRAM/MIRAMns	**Embedding**
Initial size of the vector to reduce	Initial size of square matrix D m	Size of vector n before multiplication by the embedding table
Target size	Number of desired eigen elements to calculate s	Size of vector k after multiplication
Data to reduce	A large initial matrix D to another smaller matrix with the target size (m, s)	A set of large embedding vectors of size "n" to the target size k
Reduction technique used	Convergence of eigenelements using the Arnoldi method	Multiplication with embedding table
Where are we reducing?	Krylov subspace	Embedding space

7.2) Implementation of the solution

In order to properly implement our solution, it is first important to specify the platforms on which we will launch our project. Indeed, wide & deep training is done on Huawei hardware (the AI processor Ascend) using Huawei's AI framework (MindSpore). Concerning the MIRAMns numerical method, it is launched on the Paris-Saclay University cluster, Ruche (https://mesocentre.pages.centralesupelec.fr/user_doc/ruche/01_cluster_overview/).

Separating the execution of the MIRAMns numerical method and the training of Wide & Deep makes it possible not to load Ascend.

For the implementation of MIRAMns, I opted for the Eigen library because this library is both simple to use and powerful in terms of execution and precision of calculations.

Our solution consists of several steps. Regarding the internship missions, I carried out steps 1, 3, 4, 5 and 6. Step 2 (the construction of the co-occurrence matrix) was carried out by PhD. candidate Quentin Petit.

Step 1

This step consists of extracting data from the dataset used for Wide & Deep training. The extracted data are booleans. The extraction was done thanks to a method of the MindSpore framework: *create_dict_iterator*. This method allows us to create an iterator over the dataset. A dictionary data type will be the data that is returned.

Step 2

Since IRAM/MIRAMns accepts squared A matrices, PhD. candidate Quentin sought to transform the starting dataset (non-square) into a square dataset (i.e. matrix). For this, he opted for a matrix of co-occurrence. This step will not be covered in this internship report since it essentially concerns Quentin's part.

Step 3

After having transformed the matrix from the starting dataset to the co-occurrence matrix, it will be necessary to implement the MIRAMns method on this new matrix in order to tell us if the chosen embedding dimension could be a good solution or not for our Wide & Deep case.

Step 4

This step mainly consists of choosing and justifying the right parameter in MIRAMns that best represents the embedding dimension in machine learning. To justify our choice, I demonstrated this using the demonstration by analogy.

Step 5 & Step 6

Once all the previous steps have been carried out correctly, it is now possible to change the hyperparameter of the embedding dimension of the targeted machine learning model, and launch its training with this new embedding dimension value.

The diagram below effectively summarizes these different steps.

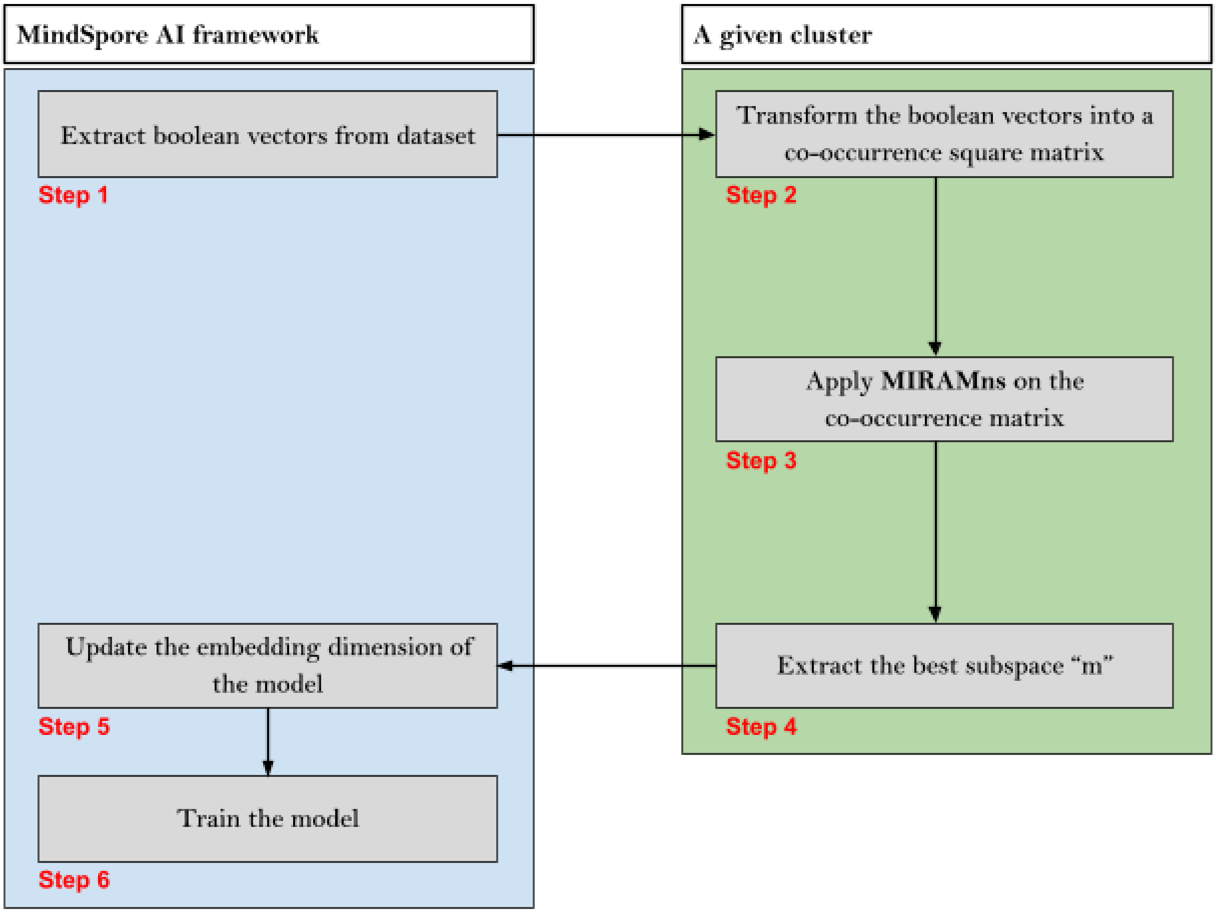

MindSpore AI framework
A given cluster
Extract boolean vectors from dataset
Step 1
Transform the boolean vectors into a co-occurrence square matrix
Step 2
Apply MIRAMns on the co-occurrence matrix
Step 3
Extract the best subspace "m"
Step 4
Update the embedding dimension of the model
Step 5
Train the model
Step 6

Figure 10 - Summary diagram of the different steps for the implementation of our solution

As mentioned in step 4, it is absolutely necessary to make a judicious choice for the parameter of the numerical method MIRAMns in order to represent the embedding dimension. For this, I chose reasoning by analogy because of its obviousness. This demonstration is detailed below.

7.3) Experiments and discussions

In this section, we will see some experiments carried out by applying our dimension reduction innovation. First of all, we will see the results of MIRAMns in order to reduce the initial dataset and see what MIRAMns can suggest as subspaces (i.e. embedding dimension). Then, we will train our Wide & Deep case study by varying each time the embedding dimensions (i.e. the subspaces of MIRAMns) in order to see if MIRAMns has proposed the right subspaces or not.

Regarding the Wide & Deep training, since the choice of the embedding dimension hyperparameter may influence the choice of the other hyperparameters, it is very important to note that we will keep exactly the same hyperparameters of the Wide & Deep model (which are suggested by MindSpore) except the embedding dimension (since our goal is to find the best one).

Here are the training hyperparameters of the Wide & Deep models proposed by MindSpore with an embedding dimension of 80 ("--emb_dim") :

Hyperparameter	Description
--device_target {Ascend,GPU}	device where the code will be implemented. (Default:Ascend)
--data_path DATA_PATH	This should be set to the same directory given to the data_download's data_dir argument
--epochs EPOCHS	Total train epochs. (Default:15)
--full_batch FULL_BATCH	Enable loading the full batch. (Default:False)
--batch_size BATCH_SIZE	Training batch size.(Default:16000)
--eval_batch_size	Eval batch size.(Default:16000)
--field_size	The number of features.(Default:39)
--vocab_size	The total features of dataset.(Default:200000)
--emb_dim	The dense embedding dimension of sparse feature.(Default:80)
--deep_layer_dim	The dimension of all deep layers.(Default:[1024,512,256,128])
--deep_layer_act	The activation function of all deep layers.(Default:'relu')
--keep_prob	The keep rate in dropout layer.(Default:1.0)
--dropout_flag	Enable dropout.(Default:0)

--output_path	Deprecated
--ckpt_path	The location of the checkpoint file. If the checkpoint file is a slice of weight, multiple checkpoint files need to be transferred. Use ';' to separate them and sort them in sequence Like "./checkpoints/0.ckpt;./checkpoints/1.ckpt". (Default:./checkpoints/)
--eval_file_name	Eval output file.(Default:eval.og)
--loss_file_name	Loss output file.(Default:loss.log)
--host_device_mix	Enable host device mode or not.(Default:0)
--dataset_type	The data type of the training files, chosen from tfrecord/mindrecord/hd5.(Default:tfrecord)
--parameter_server	Open parameter server of not.(Default:0)
--vocab_cache_size	Enable cache mode.(Default:0)

Our goal is to find a better embedding dimension (or even validate) the embedding dimension proposed by MindSpore by applying MIRAMns (80). We will therefore start by applying MIRAMns on the dataset with which we will train Wide & Deep and see what this dimension reduction method can offer us.

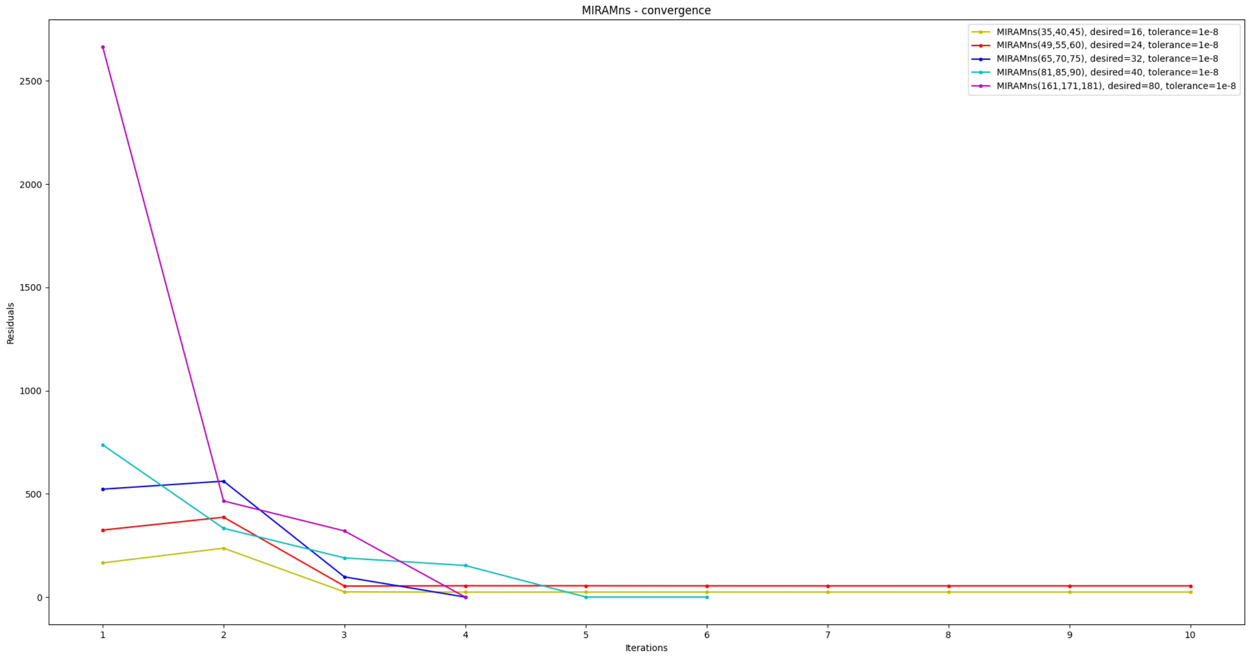


Figure 11 - Convergence graphs of MIRAMns with embedding dimensions less than embedding dimension proposed by default by MindSpore

The graph below represents the evolution of the convergence of a set of "desired" values representing the number of dominant eigen-elements to be calculated. A variation of convergence is observed according to the "desired" value taken.

For example, for "desired" equal to 16 or "desired" equal to 24, there is no convergence with the first 10 iterations. We even almost have stability at the residue level; a residual equal to 24 for "desired"=16 and a residual equal to 54 for "desired" to 24 . On the other hand, we have a convergence for the other "desired" values ; 32, 40 and 80. We can see that MIRAMns has made us a kind of filtering of the bad dimensions and the good dimensions in which we wish to reduce our large dataset.

Let us now concentrate on the good "desired" with which we have a convergence. Indeed, with "desired" equal to 80, MIRAMns converges towards the 4th iteration with a residual of 1.2907e-16. For 40 and 32, MIRAMns converges respectively towards iteration 6 with a residual of 6.68019e-11 and towards iteration 4 with a residual of 1.249e-15. If we take the speed of convergence of MIRAMns as the criterion for choosing the best embedding dimension, we can say the embedding dimension 80 is the best among the 5 values of "desired".

Note that MindSpore suggested 80 as the default embedding dimension values (as shown in the above table of default hyper-parameters). We thus have a first correspondence between MIRAMns and the choice of embedding dimension.

Now, let's analyze the training results of Wide & Deep training with the dimension embeddings with which we applied MIRAMns. The dataset used for training is Criteo Kaggle Display Advertising Challenge Dataset. The hardware used for training is Huawei's AI processor: Ascend 910. Here is some information about this machine:

OS	EulerOS 2.7
Architecture	aarch64
Byte Order	Little Endian
CPU(s)	192
Thread per core	1
CPU max MHz	2600,0000
CPU min MHz	200,0000

We will be particularly interested in the last epoch, the epoch with which the training ends, which is epoch 15. Remember that the number of epochs is a default hyperparameter selected by MindSpore. We are not going to change it so as not to increase it so as not to fall into overfitting, and not to decrease it so as to have improvements in terms of accuracy and losses. We will consider that this default epoch number is the best in our situation.

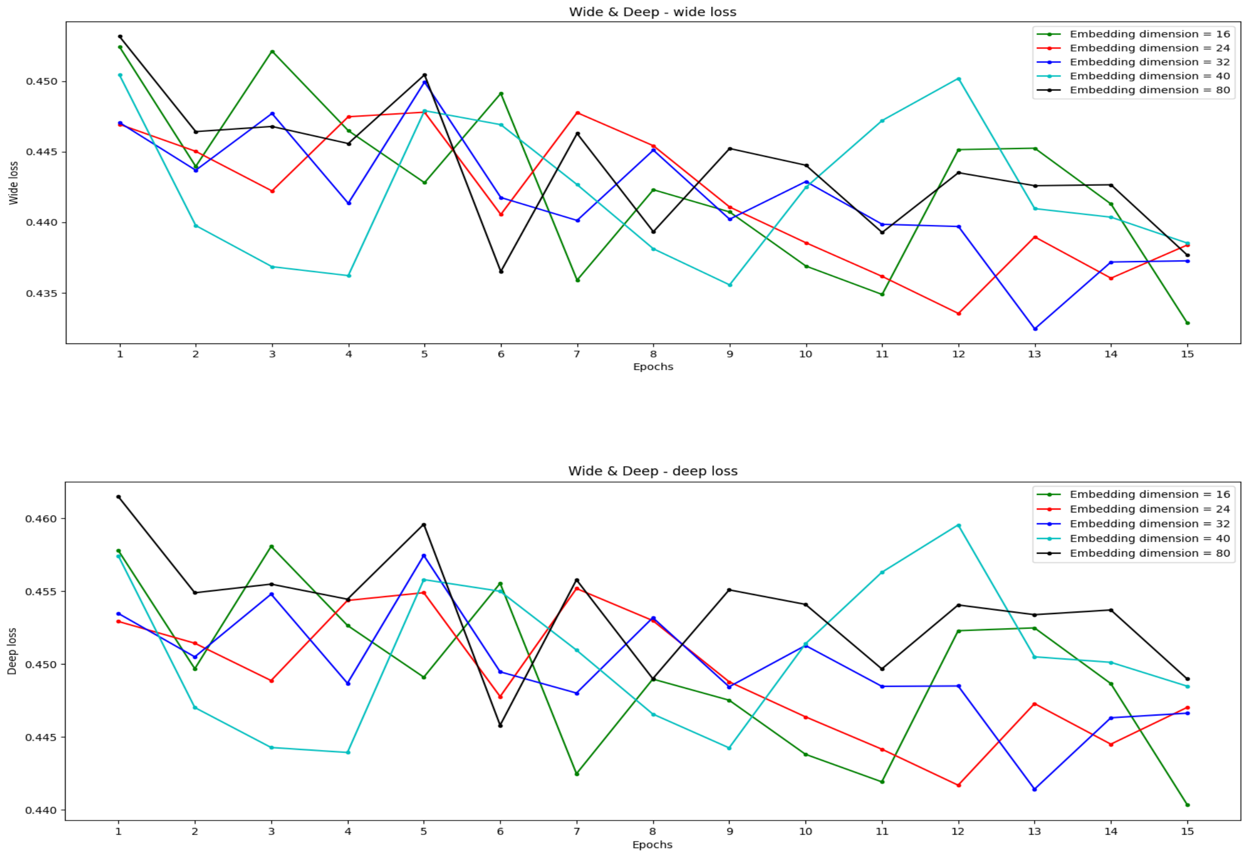


Figure 12 - Wide & Deep losses according to embedding dimensions less than or equal to 80

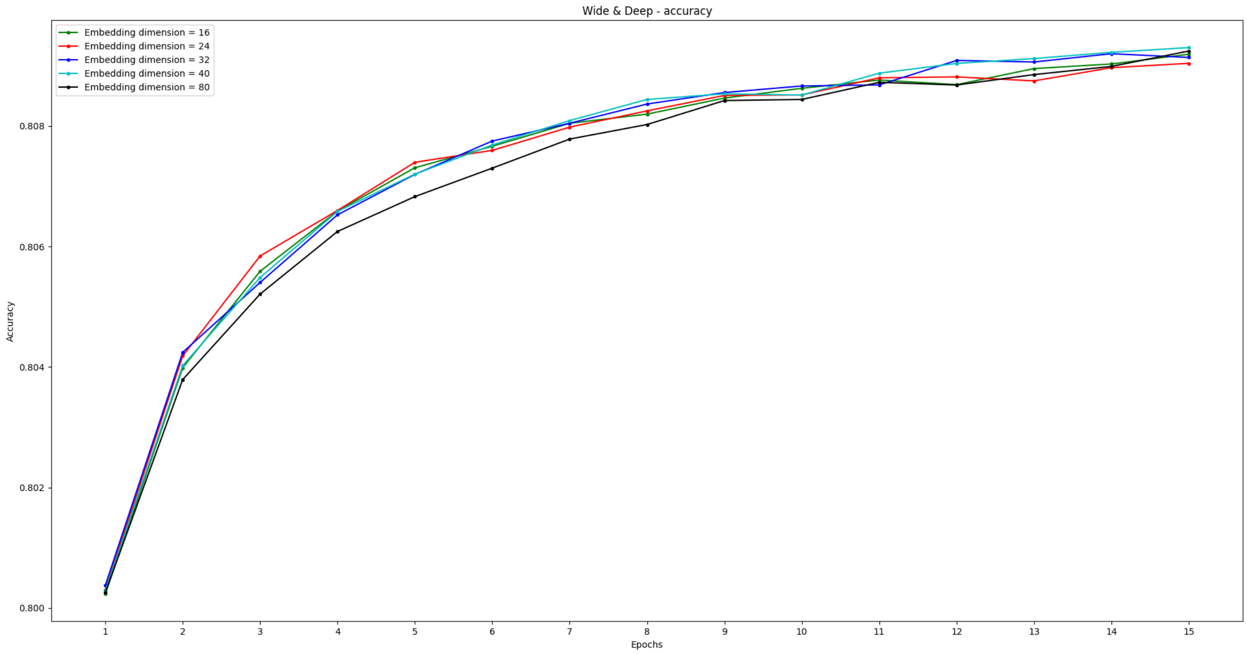


Figure 13 - Wide & Deep accuracy according to embedding dimensions less than or equal to 80

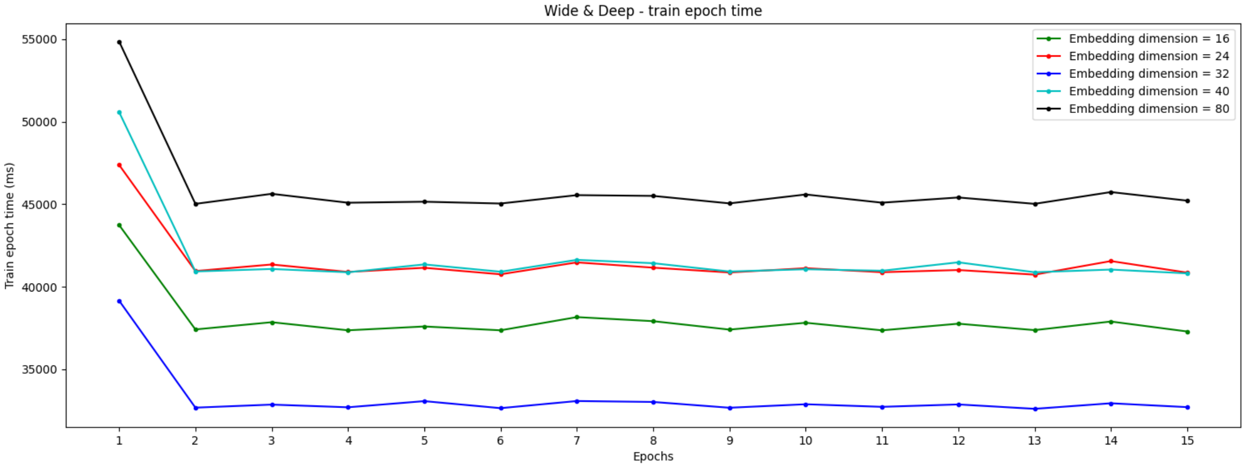


Figure 14 - Wide & Deep train epoch time according to embedding dimensions less than or equal to 80

As we mentioned, MIRAMns does not converge with "desired" (i.e. embedding dimension) equal to 16 and 24. According to figure 11, the precision of these 2 embedding dimensions (16 and 24) at epoch 15 are the 2 smallest precisions among the other embedding dimensions (32, 40 and 80). Based on this observation, we can say that MIRAMns limited us in terms of choice of embedding dimensions by eliminating those with the lowest precision. We now have the choice between 32, 40 and 80.

Training with less data for a dimension embedding is a gain in terms of execution time for an epoch. From Figure 12, an embedding dimension of 32 is the lowest in terms of execution time for an epoch.

Regarding the losses (figure 10), with an embedding dimension equal to 32, we have the smallest losses among the other embedding dimensions. This may confirm our proposal to replace 80 by 32 while keeping the other hyper-parameters fixed.

According to figure 11 and 12, an embedding dimension equal to 40 could be a very good alternative also to 80 because with 40 we obtain the best accuracy (even if a very slight difference compared to 32 and 80) and an execution time lower than 80 with 5000 ms and higher than 32 with 8000 ms approximately.

To sum up, MIRAMns allowed us to eliminate the worst embedding dimensions and keep only the best ones. Regarding the best, each dimension of embedding has its pros and cons.

7.4) Future work

However, there is still work to be done on this proposal. In order to solidify our innovation, 2 challenges remain to be accomplished. Since the execution of our MIRAMns takes a lot of time, it will therefore require a whole study concerning its optimization in terms of execution time via a complete profiling.

It will also be necessary to test, for our case study, with embedding dimensions greater than 80 such as 120 or 200. This requires efficient memory optimization. Our MIRAMns implementation does not yet work with "desired" equal to these values.

Also, it would be interesting to do other tests with other models other than recommendation systems such as transformers (NLP); GPT, BERT ... . For this, it will first be necessary to accomplish the 2 previous points. We trained GPT 2 by varying the embedding dimensions in order to have an idea of what our new improved MIRAMns will be able to offer us.

Note that a statistical indicator of how reliably a language model predicts a text sample is called perplexity. It measures how "surprised" the model is when it encounters fresh data. The model predicts the text more accurately, the lower the perplexity

Embedding size	192	516	1024
Perplexity	6210.462626976933	811.3101094340896	237.0283377988297

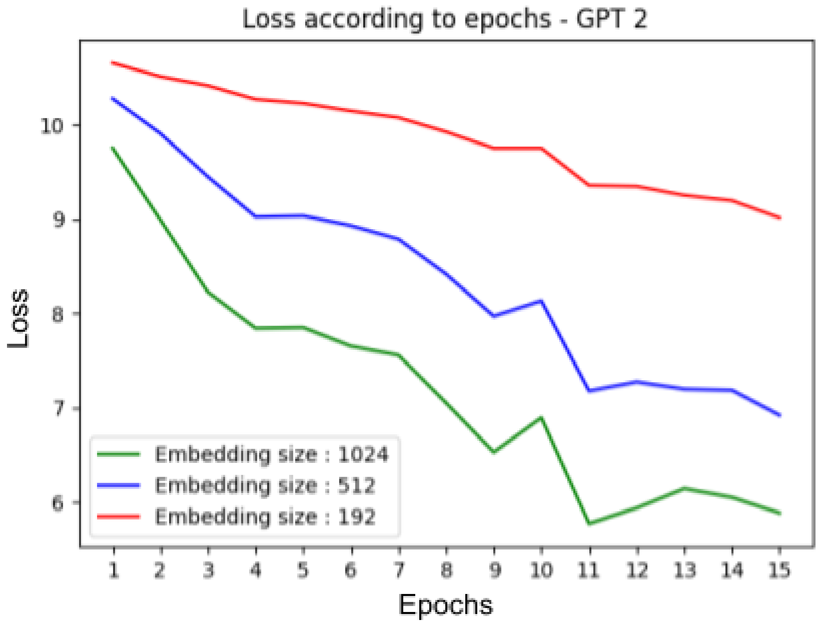


Figure 15 - The loss according to epochs for different embedding dimensions for the case of GPT 2

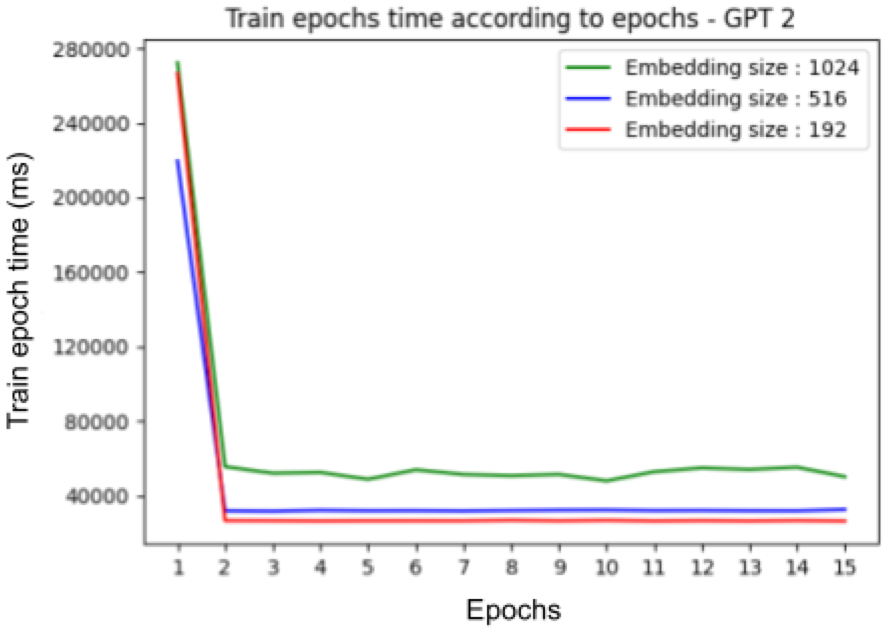


Figure 16 - The train epoch time according to epochs for different embedding dimensions for the case of GPT 2

According to Figure 13, the highest dimensional embedding (1024) represents the smallest loss. This is normal since the higher the embedding size, the more training information we have. This is visible from the table of perplexities. With a dimension embedding equal to 1024, we have the least perplexity. On the other hand, with 1024 of embedding size, we almost double the training time for an epoch. compared to embedding size 516 and 192. A good compromise between perplexity, loss and execution time, is to take an embedding size between 1024 and 516. This will be one of the challenges to aim for in the future : verify this hypothesis using MIRAMns.

Conclusion

Ultimately, the interest of our study is based on two main aspects. The first, in the field of linear algebra, concerns the numerical method MIRAMns. The second, in the field of AI, which concerns the case study of the Wide & Deep model.

The goal of our study is to accelerate the training of a deep learning model based on the right choice of its hyperparameters. For this, we focused on the reduction of dimension. In order to have an innovative idea, we took the numerical method MIRAMns, which is not very used in AI, in order to see its behavior at the level of dimension reduction.

To meet this goal, we have explained how Wide & Deep works. Then, we justified the choice of using MIRAMns and explained how it works. Then, we presented the results of our implementation of the MIRAMns method with the selected libraries. Finally, we applied MIRAMns on the Wide & Deep training dataset to see the result of the embedding dimension that this method can offer.

Despite the good results for the Wide & Deep case, work remains to be done. Indeed, we now wish to generalize our application on models other than Wide & Deep, such as transformers (NLP), GPT, BERT, etc. This track requires a good optimization of our MIRAMns implementation both in time and in space.

References

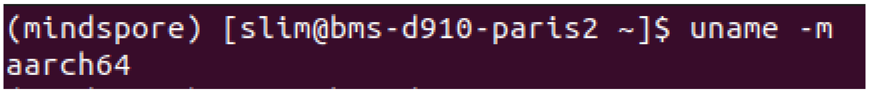
- https://www.huawei.com/en/news/2016/6/mathematics-research-center-in-france
- https://www.chinadaily.com.cn/a/202010/10/WS5f8126dea31024ad0ba7dd9a.html
- https://e.huawei.com/ar-SA/products/cloud-computing-dc/atlas/mindspore
- https://gitee.com/mindspore/models
- https://eigen.tuxfamily.org/index.php?title=Main_Page
- https://www.open-mpi.org/
- https://www.baeldung.com/cs/neural-nets-embedding-layers
- https://ai.googleblog.com/2016/06/wide-deep-learning-better-together-with.html
- https://medium.com/analytics-vidhya/wide-deep-learning-for-recommender-systems-dc99094fc291
- https://www.mindspore.cn/docs/en/r2.0/api_python/dataset/dataset_method/iterator/mindspore.dataset.Dataset.create_dict_iterator.html

- "A Matlab Implementation of the Implicitly Restarted Arnoldi Method for Solving Large-Scale Eigenvalue Problems" - Richard J. Radke
- "Wide & Deep Learning for Recommender Systems" - Google's paper
- "A KEY TO CHOOSE SUBSPACE SIZE IN IMPLICITLY RESTARTED ARNOLDI METHOD" by S. A. SHAHZADEH FAZELI , N. EMAD , AND Z. LIU
- AutoShard: Automated Embedding Table Sharding for Recommender Systems https://arxiv.org/pdf/2208.06399.pdf

Appendix

1) Installing MindSpore

Step 1 - check the architecture



```
(mindspore) [slim@bms-d910-paris2 ~]$ uname -m
aarch64
```

Step 2 - proxy activation

export http_proxy=http://127.0.0.1:3128

export https_proxy=https://127.0.0.1:3128

Step 3 - conda environment

wget https://repo.anaconda.com/archive/Anaconda3-2022.10-Linux-aarch64.sh

chmod +x Anaconda*.sh

bash Anaconda*.sh

source .bashrc #to use conda in the current session

Disable “base” default environment :

conda config –set auto_activate_base false

To have a clean starting environment follow the following commands:

conda create -n mindspore python=3.7.5 –channel conda-forge (mindspore = ENV_NAME)

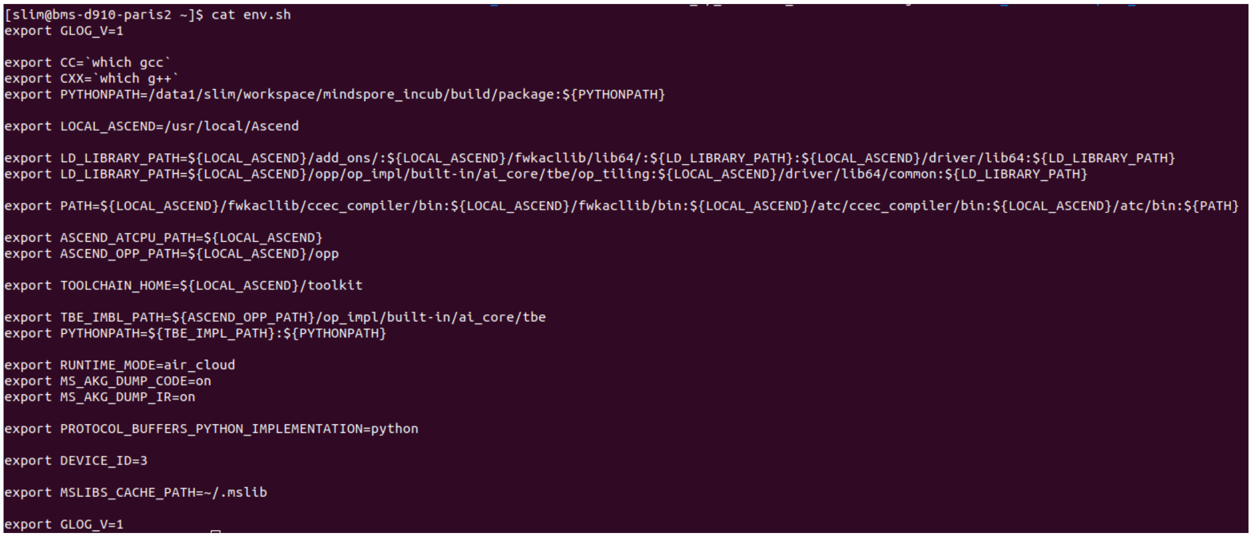
conda activate mindspore (mindspore = ENV_NAME)

Step 4 - some useful packages

conda install pandas pytest numpy

Step 5 - driver environment variables

Create an env.sh file and insert the following :



```
[slim@bms-d910-paris2 ~]$ cat env.sh
export GLOG_V=1

export CC=`which gcc`
export CXX=`which g++`
export PYTHONPATH=/data1/slim/workspace/mindspore_incub/build/package:${PYTHONPATH}

export LOCAL_ASCEND=/usr/local/Ascend

export LD_LIBRARY_PATH=${LOCAL_ASCEND}/add_ons/:${LOCAL_ASCEND}/fwkacllib/lib64/:${LD_LIBRARY_PATH}:${LOCAL_ASCEND}/driver/lib64:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${LOCAL_ASCEND}/opp/op_impl/built-in/ai_core/tbe/op_tiling:${LOCAL_ASCEND}/driver/lib64/common:${LD_LIBRARY_PATH}

export PATH=${LOCAL_ASCEND}/fwkacllib/ccec_compiler/bin:${LOCAL_ASCEND}/fwkacllib/bin:${LOCAL_ASCEND}/atc/ccec_compiler/bin:${LOCAL_ASCEND}/atc/bin:${PATH}

export ASCEND_ATCPU_PATH=${LOCAL_ASCEND}
export ASCEND_OPP_PATH=${LOCAL_ASCEND}/opp

export TOOLCHAIN_HOME=${LOCAL_ASCEND}/toolkit

export TBE_IMBL_PATH=${ASCEND_OPP_PATH}/op_impl/built-in/ai_core/tbe
export PYTHONPATH=${TBE_IMPL_PATH}:${PYTHONPATH}

export RUNTIME_MODE=air_cloud
export MS_AKG_DUMP_CODE=on
export MS_AKG_DUMP_IR=on

export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=python

export DEVICE_ID=3

export MSLIBS_CACHE_PATH=~/.mslib

export GLOG_V=1
```

source ./env.sh

Step 6 - driver python package

cd /usr/local/Ascend/latest/lib64

pip install te-0.4.0-py3-none-any.whl

pip install topi-0.4.0-py3-none-any.whl

pip install hccl-0.1.0-py3-none-any.whl

Step 7 - clone the repository

git clone -b pipeline https://gitee.com/ch-l/mindspore_incub.git –depth 1 (clone branch pipeline)

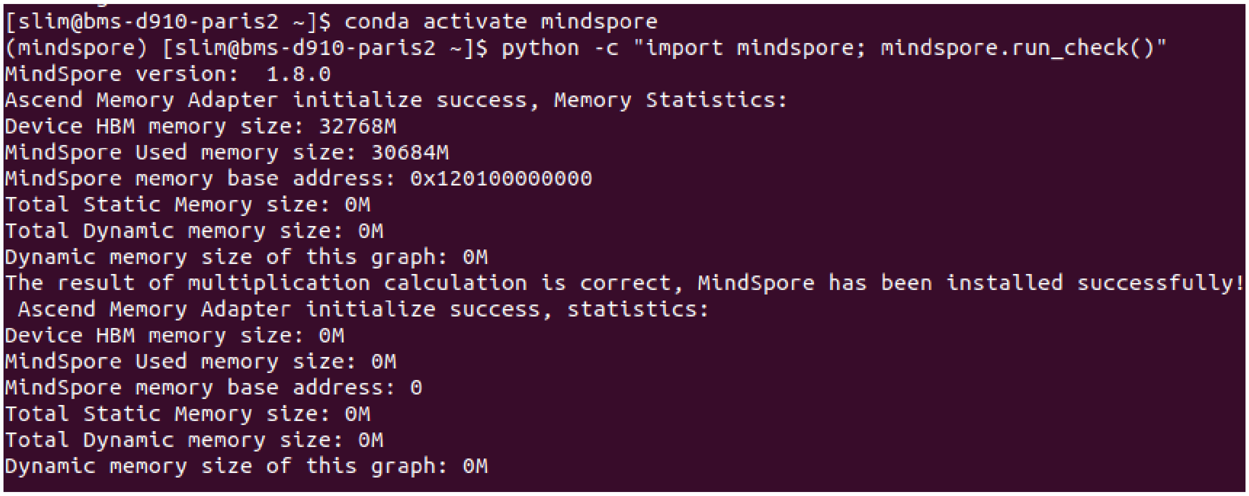
cd mindspore

bash build.sh -e ascend -128

pip install build/package/mindspore*.whl

Step 8 - test installation

We run the following commands to verify if MindSpore has been well installed and is supported:



```
[slim@bms-d910-paris2 ~]$ conda activate mindspore
(mindspore) [slim@bms-d910-paris2 ~]$ python -c "import mindspore; mindspore.run_check()"
MindSpore version:  1.8.0
Ascend Memory Adapter initialize success, Memory Statistics:
Device HBM memory size: 32768M
MindSpore Used memory size: 30684M
MindSpore memory base address: 0x120100000000
Total Static Memory size: 0M
Total Dynamic memory size: 0M
Dynamic memory size of this graph: 0M
The result of multiplication calculation is correct, MindSpore has been installed successfully!
 Ascend Memory Adapter initialize success, statistics:
Device HBM memory size: 0M
MindSpore Used memory size: 0M
MindSpore memory base address: 0
Total Static Memory size: 0M
Total Dynamic memory size: 0M
Dynamic memory size of this graph: 0M
```

Step 9 (final step) - Models Zoo

git clone https://gitee.com/mindspore/models.git

2) Data regarding Wide & Deep training with MindSpore in Ascend

- ED = 16

Epochs	Accuracy	Wide loss	Deep loss	Train epoch time (ms)
1	0.8002334656470003	0.45243815	0.457794	43737.533
2	0.8039880534768308	0.44395757	0.44969097	37414.696
3	0.8055890100341039	0.4521265	0.45808792	37851.650
4	0.8065894167930213	0.44649523	0.45266372	37361.461
5	0.8073077568382313	0.44280824	0.44911182	37593.691
6	0.8076650265437493	0.44913048	0.45555124	37361.279
7	0.8080462449431487	0.4359067	0.44248706	38160.401
8	0.8081986317490297	0.4423076	0.44898158	37915.460
9	0.8084645478636591	0.44073412	0.44753253	37404.224
10	0.8086270100141928	0.4369031	0.44382012	37819.020
11	0.8087641490378437	0.43489015	0.4419304	37359.598
12	0.8086875389995439	0.4451487	0.45230132	37762.082
13	0.8089541695742397	0.4452511	0.45249367	37372.432
14	0.8090307277497247	0.44129863	0.4486697	37893.798
15	0.8091922253116767	0.43287364	0.44036445	37290.198

- ED = 24

Epochs	Accuracy	Wide loss	Deep loss	Train epoch time (ms)
1	0.8003048766026182	0.44694078	0.45293447	47378.353
2	0.8041836925797409	0.44502726	0.45144254	40950.354
3	0.8058428473519665	0.4422233	0.44888088	41348.118
4	0.8065988016086041	0.44747728	0.45437622	40896.158
5	0.80739869704327	0.44779998	0.45490816	41147.983
6	0.8075966169594895	0.44055524	0.44778192	40753.736
7	0.8079822557075815	0.44776648	0.4551951	41474.715
8	0.8082517827215502	0.44542485	0.45299515	41155.443
9	0.8085084220295659	0.44108558	0.44879442	40863.825
10	0.8085171490358485	0.43855426	0.4463914	41124.444
11	0.8088015330332791	0.4361697	0.44417268	40878.979
12	0.8088159747009488	0.43354574	0.4417003	41012.606
13	0.8087498088252442	0.4389682	0.44729775	40732.802
14	0.8089691834532168	0.4360448	0.4445161	41554.733
15	0.8090413058374943	0.43839425	0.44703972	40855.205

- ED = 32

Epochs	Accuracy	Wide loss	Deep loss	Train epoch time (ms)
1	0.8003740894561854	0.44707084	0.45348155	39158.386
2	0.8042424633622053	0.44368905	0.4505067	32677.635
3	0.8054032578869861	0.4477097	0.4548087	32862.417
4	0.8065264929953647	0.44134864	0.44869414	32699.065
5	0.8071975246446202	0.44993687	0.4574642	33071.929
6	0.8077512820639929	0.4417568	0.44947946	32647.216
7	0.8080468055938591	0.44013423	0.4480228	33078.353
8	0.808365696866479	0.4451113	0.45319536	33022.376
9	0.8085565055181099	0.44022185	0.44845542	32672.096
10	0.8086654714711377	0.44289052	0.45128706	32883.975
11	0.808683029890049	0.4398619	0.4484849	32726.218
12	0.8090894009975557	0.43970603	0.44850844	32872.273
13	0.8090636369832824	0.4324558	0.44142887	32606.107
14	0.8092012672172791	0.43718898	0.4463255	32942.076
15	0.809138697032476	0.43727434	0.44664136	32708.647

- ED = 40

Epochs	Accuracy	Wide loss	Deep loss	Train epoch time (ms)
1	0.8002864469890296	0.4504385	0.45742443	50579.157
2	0.8040227778925486	0.43977335	0.44702983	40924.835
3	0.8054858729337968	0.4368594	0.44428706	41076.014
4	0.8065825928593522	0.43622664	0.44394603	40875.556
5	0.8072018441812142	0.44790864	0.45580763	41351.932
6	0.8076862867619594	0.4469167	0.45500737	40909.165
7	0.8080911861636055	0.44267142	0.45096993	41628.747
8	0.8084415129445748	0.4381262	0.4465817	41425.374
9	0.808535164141255	0.435571	0.44426104	40921.845
10	0.8085147696900197	0.44248056	0.4514059	41060.205
11	0.8088777915816503	0.44721118	0.4563096	40967.895
12	0.8090396393629189	0.45019844	0.4595624	41478.950
13	0.8091205262329879	0.44097078	0.45051023	40877.559
14	0.8092237530486981	0.440366	0.45013192	41040.680
15	0.8093027861448135	0.4385314	0.44849464	40810.230

- ED = 80

Epochs	Accuracy	Wide loss	Deep loss	Train epoch time (ms)
1	0.8002548345618531	0.45316774	0.46149144	54853.595
2	0.8037916617934354	0.4464254	0.45490396	45019.418
3	0.8052106311646563	0.44678956	0.45549685	45628.018
4	0.8062500584500175	0.4455805	0.4544628	45087.884
5	0.8068290188458301	0.45044813	0.4596096	45149.358
6	0.8073014926482075	0.43651414	0.44581372	45039.733
7	0.8077842024891903	0.44629076	0.45579612	45547.495
8	0.8080253934346572	0.43933585	0.44900677	45501.858
9	0.8084239100926363	0.44522816	0.45510408	45047.624
10	0.8084417568117545	0.44404247	0.45410666	45587.871
11	0.808726149647429	0.43928322	0.44967988	45090.193
12	0.8086815429373618	0.44351515	0.4540668	45405.727
13	0.808854387238755	0.44259214	0.4534003	45020.757
14	0.8089899902108675	0.44265974	0.45372313	45734.083
15	0.8092454785463628	0.43768018	0.44899312	45214.088

- ED = 120

Epochs	Accuracy	Wide loss	Deep loss	Train epoch time (ms)
1	0.7999290744901967	0.45221236	0.4612357	70902.350
2	0.8033196375068336	0.44979978	0.45907128	52284.421
3	0.8049597007013992	0.441763	0.4511037	52845.113
4	0.8060729216862548	0.450885	0.46054688	52279.486
5	0.8066483550976077	0.4496848	0.4595067	52313.404
6	0.8069589343540664	0.44080877	0.45081708	52257.515
7	0.8074054071679612	0.43273282	0.4429399	53224.951
8	0.8079222014068914	0.44289678	0.45333436	53039.518
9	0.8078842903272903	0.44829533	0.45885438	52445.340
10	0.8083937692780495	0.42971784	0.4404702	52997.663
11	0.8083985039520118	0.44409126	0.45501646	52316.307
12	0.8086351982561613	0.43475536	0.44598716	52530.819
13	0.8088134574622289	0.44237047	0.45381427	52346.858
14	0.8086557476888131	0.4414701	0.4530592	52675.195
15	0.8088379340736527	0.44276032	0.4547304	52328.534

- ED = 200

Epochs	Accuracy	Wide loss	Deep loss	Train epoch time (ms)
1	0.7997842309139266	0.44706762	0.45705423	83899.300
2	0.8031822997439177	0.4416085	0.4517657	64474.556
3	0.8046054338698678	0.44239187	0.4526662	64731.068
4	0.8055074440484421	0.45287538	0.46346262	64350.901
5	0.8062462223519479	0.44800916	0.4586957	64579.589
6	0.8069727062528632	0.42980456	0.44069633	64406.014
7	0.8072364493746809	0.4416076	0.4527016	64976.234
8	0.8075531630526911	0.4410252	0.45236188	65005.003
9	0.8080012805217544	0.4475862	0.4589138	64633.434
10	0.8081875163814088	0.4445421	0.45616448	64952.141
11	0.8084014291849505	0.43807065	0.44984537	64572.738
12	0.8086397959495807	0.43729457	0.4492399	65101.433
13	0.8085991935968484	0.43592805	0.44804543	64590.672
14	0.8085314738338888	0.44412795	0.4565987	64938.561
15	0.8087927829264143	0.44587874	0.4585424	64479.778

3) Test document of IRAM and MIRAMns

This document is divided into 2 parts :

- “ Testing & validation of IRAM - Implicitly Restarted Arnoldi Method ”
- “ Testing & validation of MIRAMns - Multiple Implicitly Restarted Arnoldi Method with Nested Subspaces ”

As the titles of the 2 parts indicate, the purpose of part 1 is to test our IRAM implementation in C++ by comparing the convergence with the implementation of IRAM in matlab. Part 2 is to compare the convergence of our MIRAMns implementation in C++ with the matlab implementation. The tests were carried out on different types of matrices.

Testing & validation of
IRAM - Implicitly Restarted Arnoldi Method

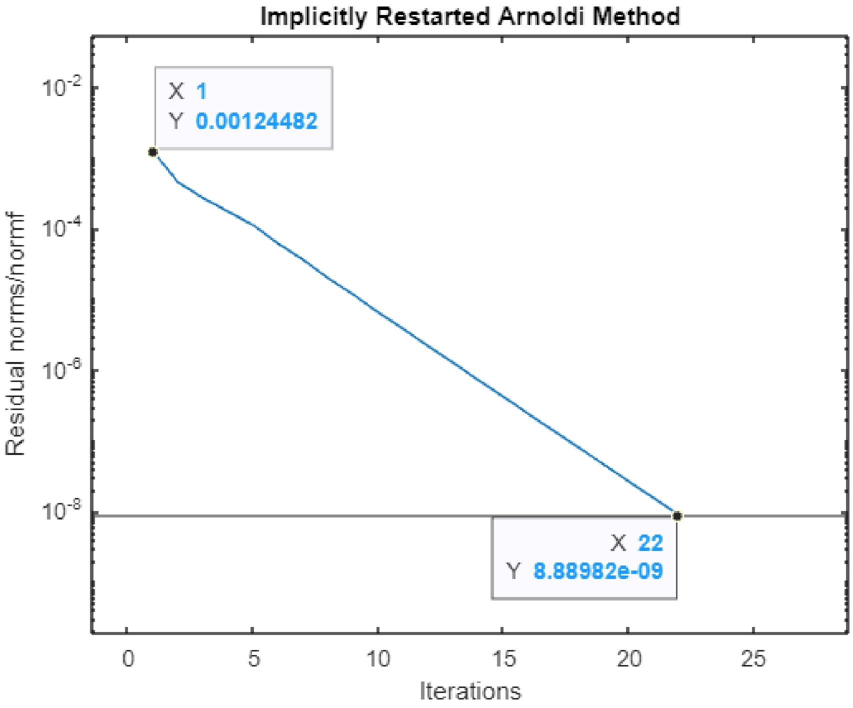
In order to validate our *IRAM* implementation, we compared the convergence of our IRAM implementation (in c++) with the IRAM implementation (in matlab) of the paper "A KEY TO CHOOSE SUBSPACE SIZE IN IMPLICITLY RESTARTED ARNOLDI METHOD" published by PhD. Fazeli, Prof. Emad, and Dr. Liu.

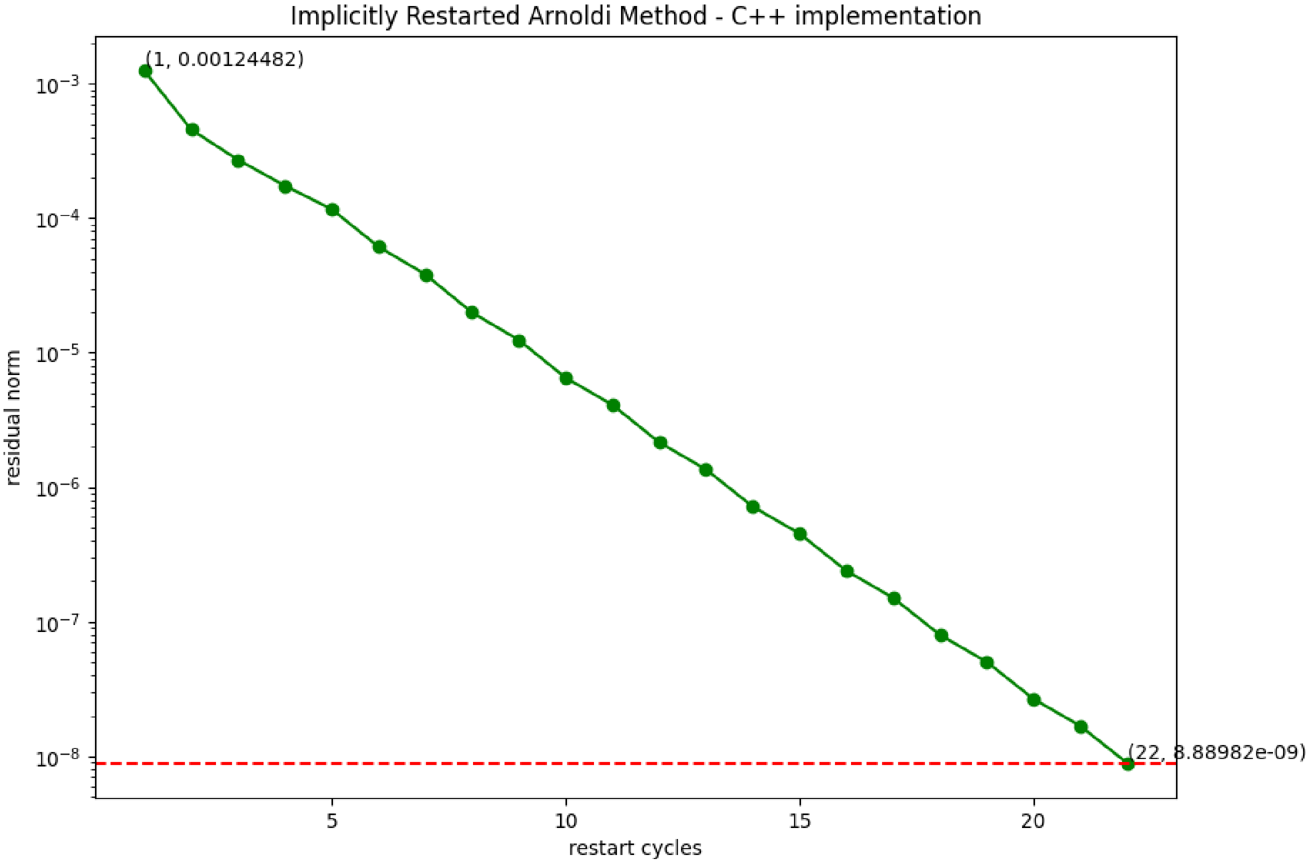
In this document, we tested our implementation with different types of matrices (real & unsymmetric, real & symmetric, real & symmetric & indefinite, real & symmetric & positive & definite). To have a valid comparison, we took the same initial conditions and the same matrices for each of the tests.

Each main part of this document represents a test with a given matrix. The tests with different initial conditions therefore represent the sub-parts. Each time, the coordinates of the 1st and the last points of the graphs are given in order to have an order of magnitude of the results of the 2 implementations (Matlab & C++). In the graphs of this document, the last point of each graph is highlighted with a horizontal line.

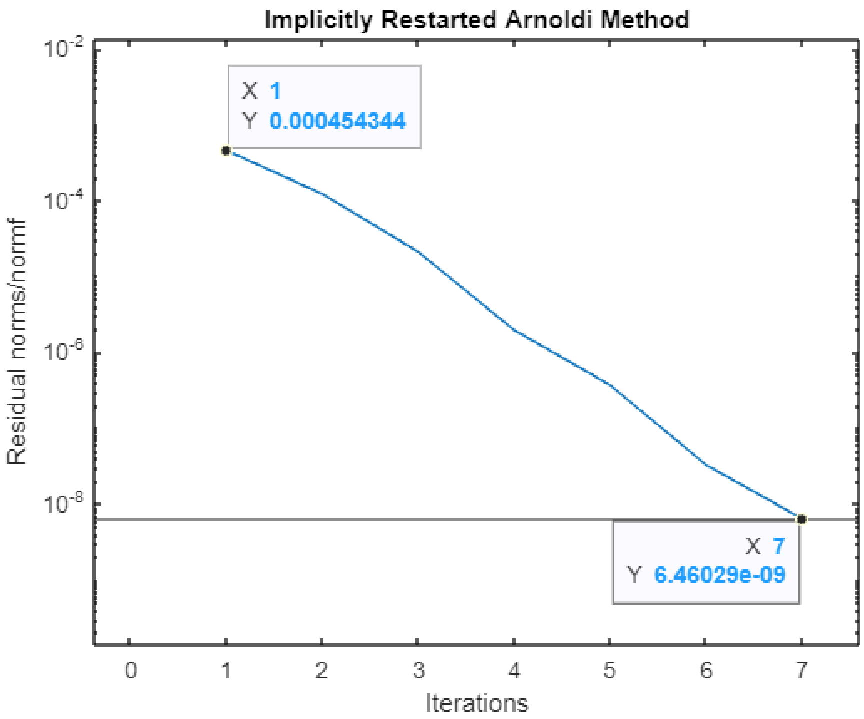
Tests with the “*AM_1000*” matrix : real & unsymmetric matrix

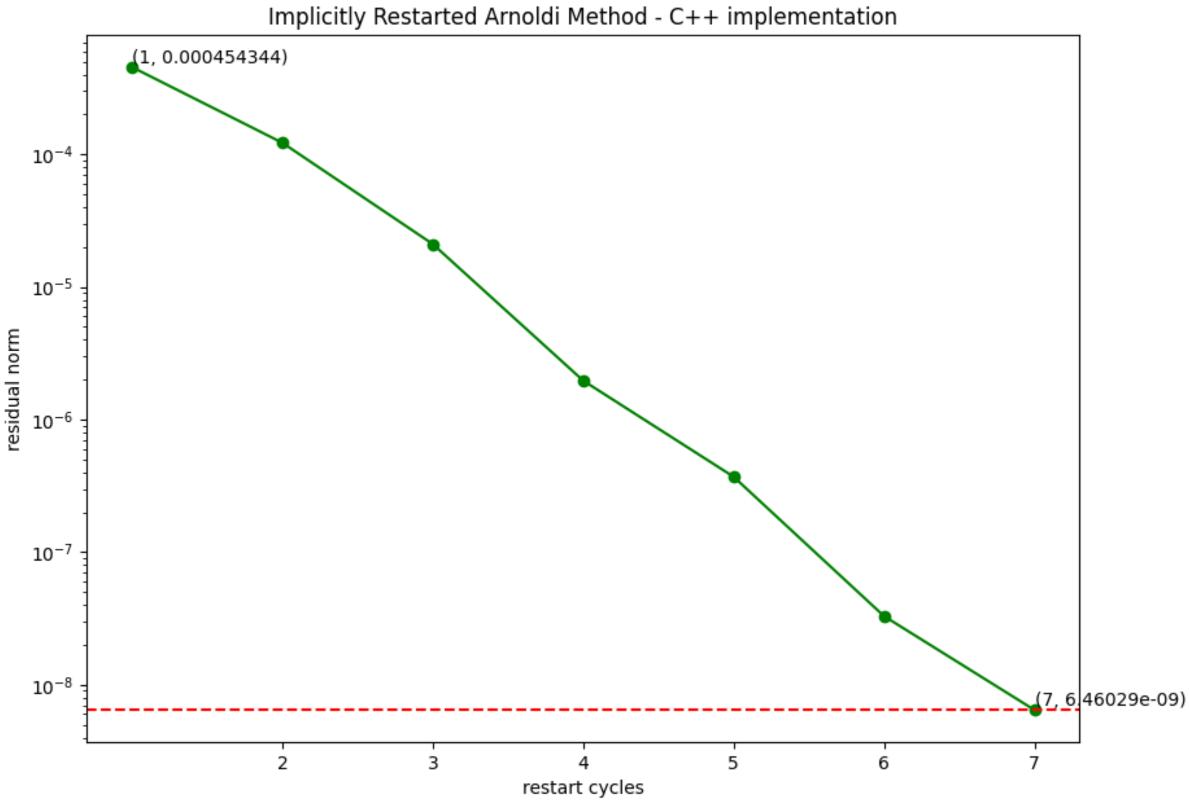
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Subspace	20
Tolerance	1e-8
Size of the matrice	1000 (with 2998 nonzero elements)


Implicitly Restarted Arnoldi Method
X 1
Y 0.00124482
X 22
Y 8.88982e-09
Residual norms/normf
10^{-2}
10^{-4}
10^{-6}
10^{-8}
0
5
10
15
20
25
Iterations

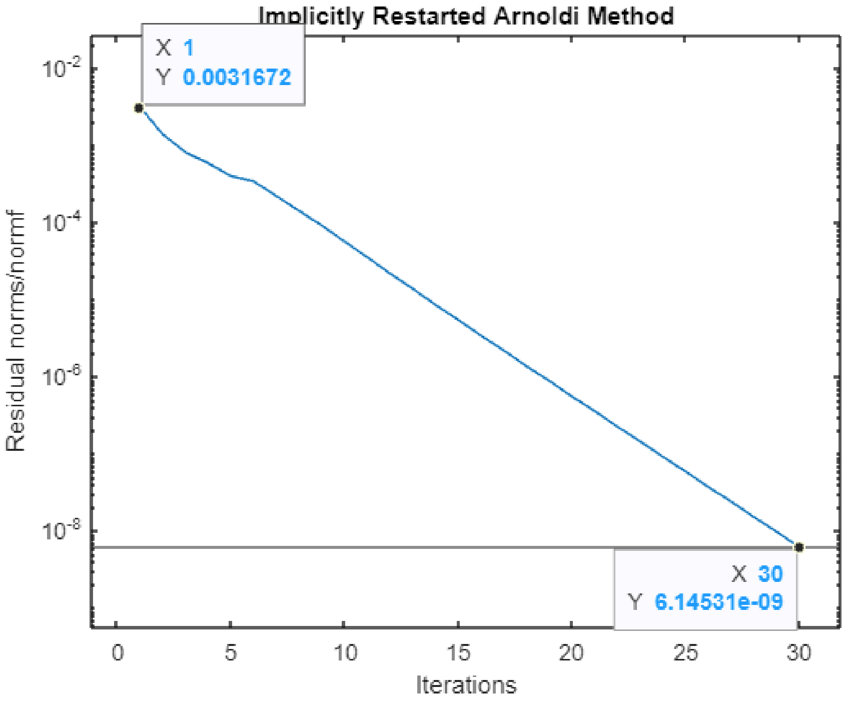

Implicitly Restarted Arnoldi Method - C++ implementation
(1, 0.00124482)
(22, 8.88982e-09)
residual norm
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
5
10
15
20
restart cycles

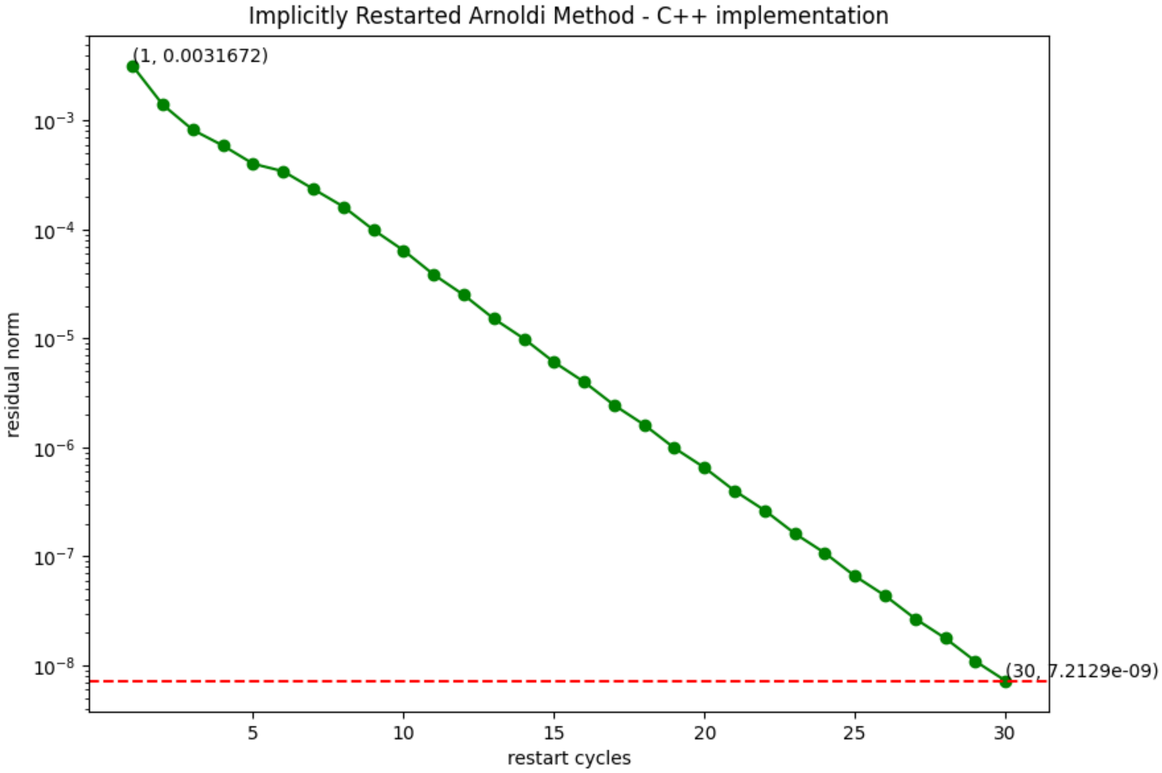
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Subspace	40
Tolerance	1e-8
Size of the matrice	1000 (with 2998 nonzero elements)


Implicitly Restarted Arnoldi Method
X 1
Y 0.000454344
X 7
Y 6.46029e-09
Residual norms/normf
10^{-2}
10^{-4}
10^{-6}
10^{-8}
0
1
2
3
4
5
6
7
Iterations

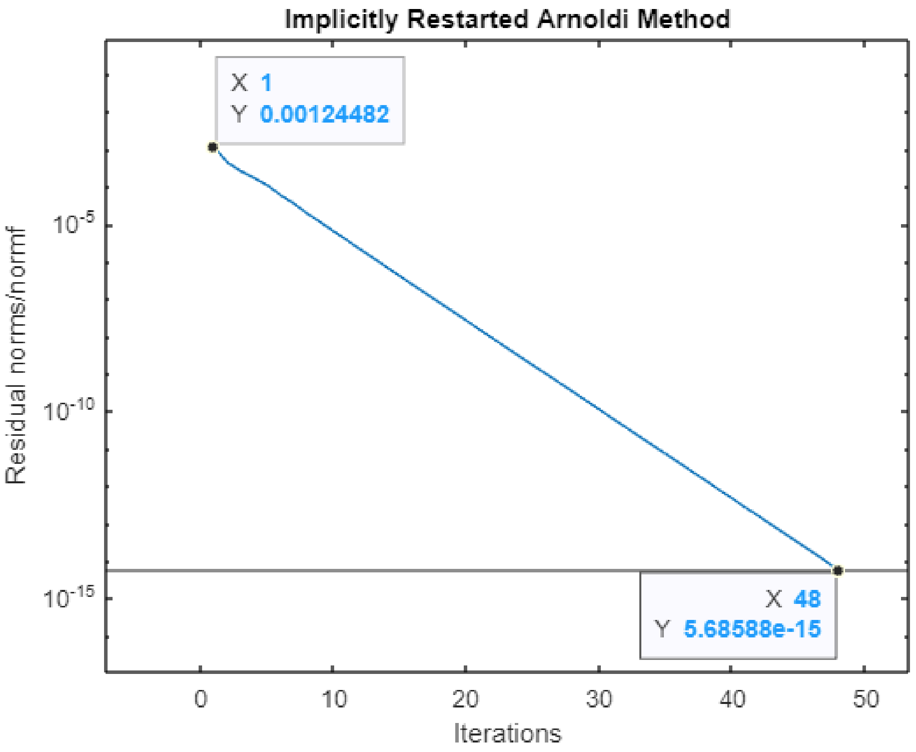

Implicitly Restarted Arnoldi Method - C++ implementation
(1, 0.000454344)
(7, 6.46029e-09)
residual norm
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
2
3
4
5
6
7
restart cycles

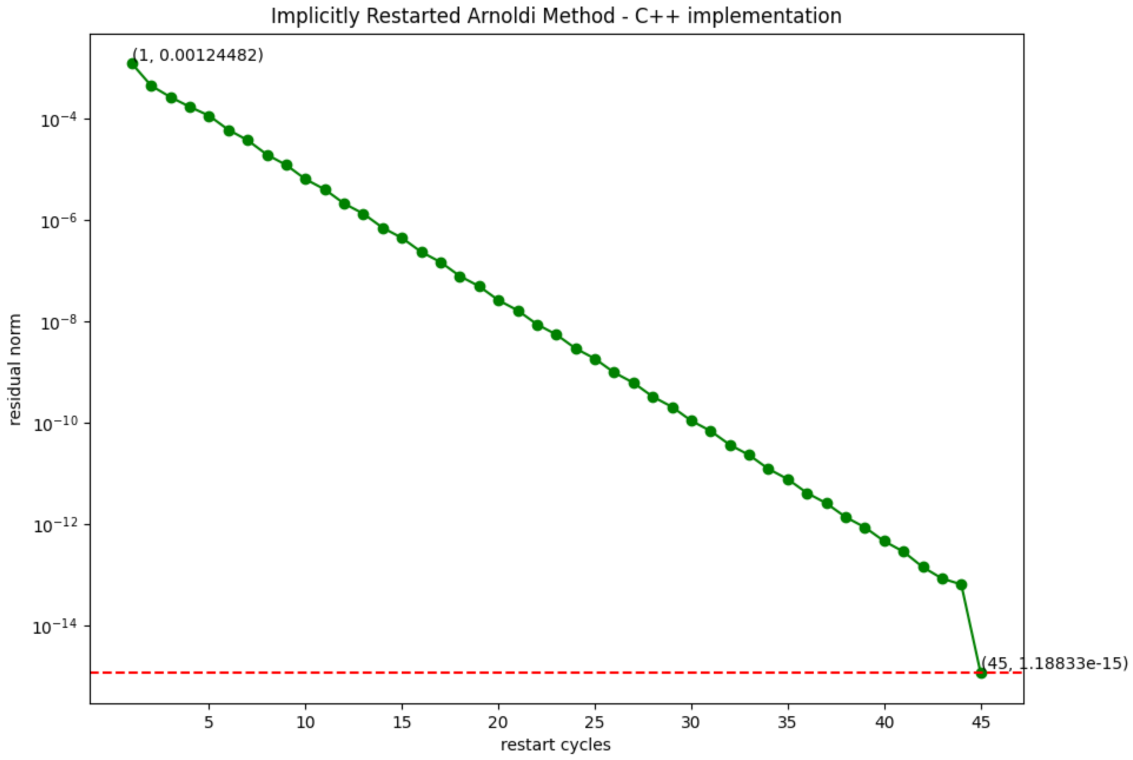
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	4
Subspace	20
Tolerance	1e-8
Size of the matrice	1000 (with 2998 nonzero elements)


Implicitly Restarted Arnoldi Method
X 1
Y 0.0031672
X 30
Y 6.14531e-09
Residual norms/normf
10^{-2}
10^{-4}
10^{-6}
10^{-8}
0
5
10
15
20
25
30
Iterations


Implicitly Restarted Arnoldi Method - C++ implementation
(1, 0.0031672)
(30, 7.2129e-09)
residual norm
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
5
10
15
20
25
30
restart cycles

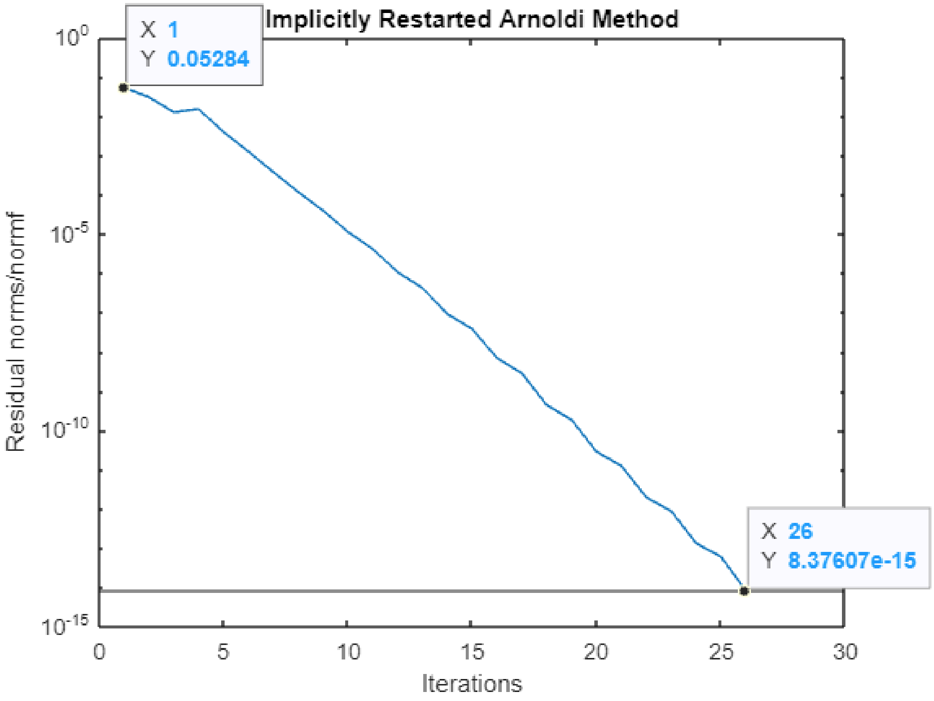
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Subspace	20
Tolerance	1e-14
Size of the matrice	1000 (with 2998 nonzero elements)

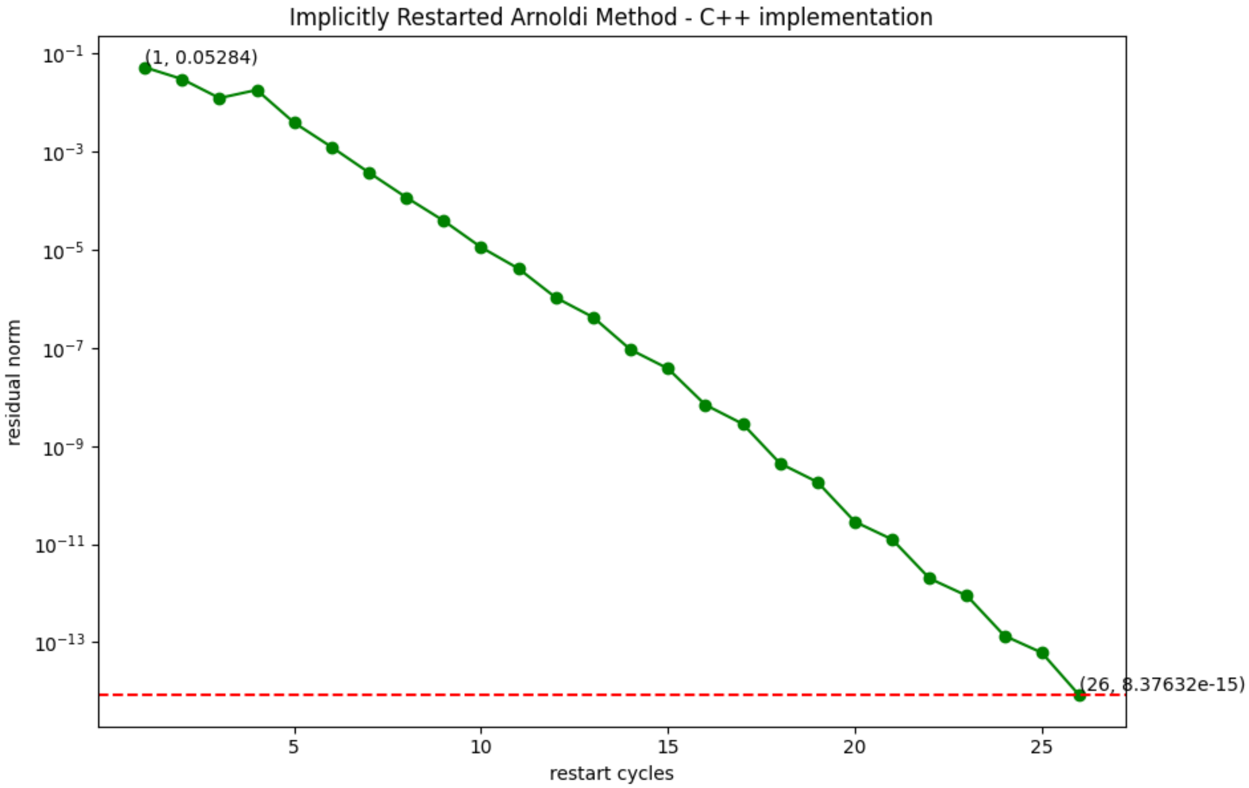

Implicitly Restarted Arnoldi Method
X 1
Y 0.00124482
X 48
Y 5.68588e-15
Residual norms/normf
10^{-5}
10^{-10}
10^{-15}
0
10
20
30
40
50
Iterations


Implicitly Restarted Arnoldi Method - C++ implementation
(1, 0.00124482)
(45, 1.18833e-15)
residual norm
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
5
10
15
20
25
30
35
40
45
restart cycles

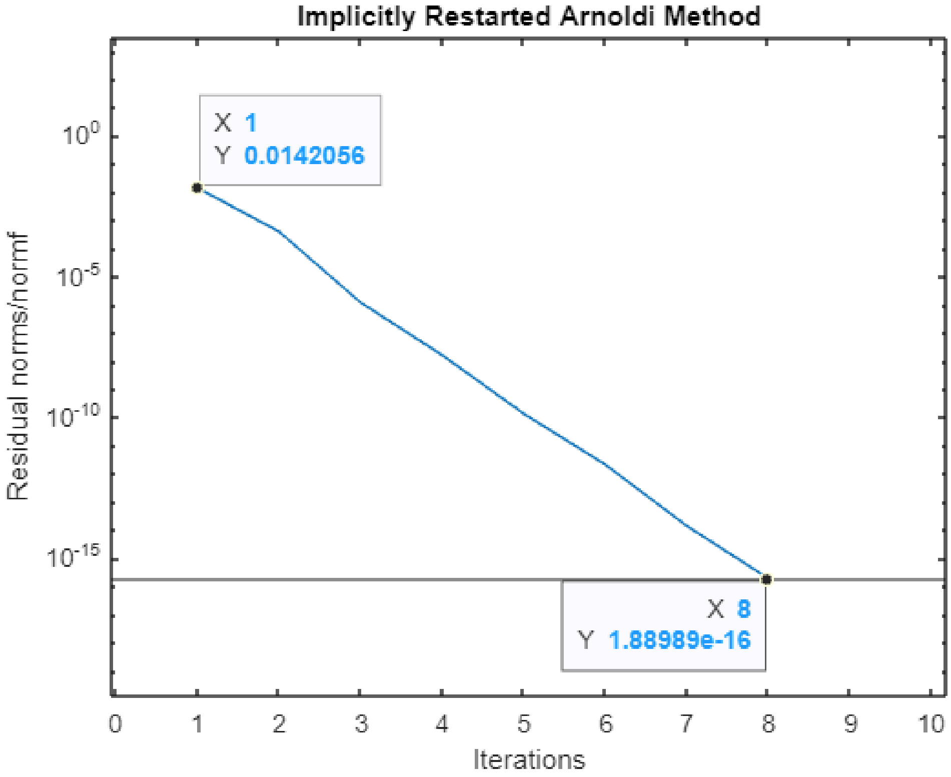
Tests with the “*bfw62a*” matrix : real & unsymmetric matrix

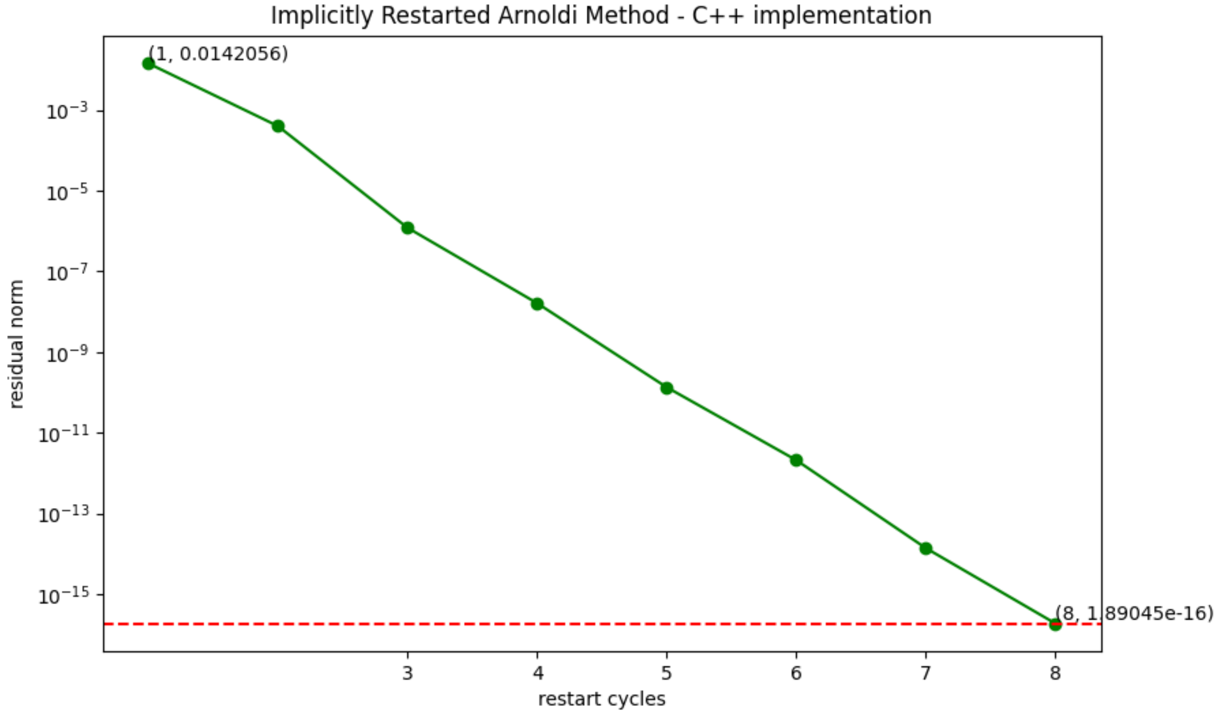
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Subspace	5
Tolerance	1e-14
Size of the matrice	62 (with 450 nonzero elements)


Implicitly Restarted Arnoldi Method
X 1
Y 0.05284
X 26
Y 8.37607e-15
10^{0}
10^{-5}
10^{-10}
10^{-15}
Residual norms/normf
0
5
10
15
20
25
30
Iterations

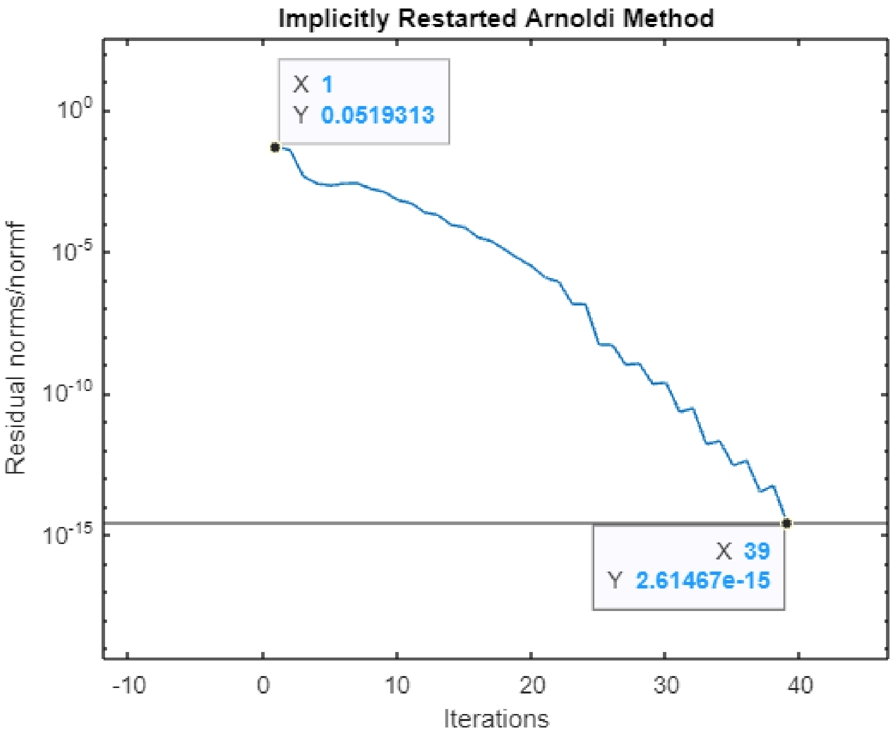

Implicitly Restarted Arnoldi Method - C++ implementation
(1, 0.05284)
(26, 8.37632e-15)
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
residual norm
5
10
15
20
25
restart cycles

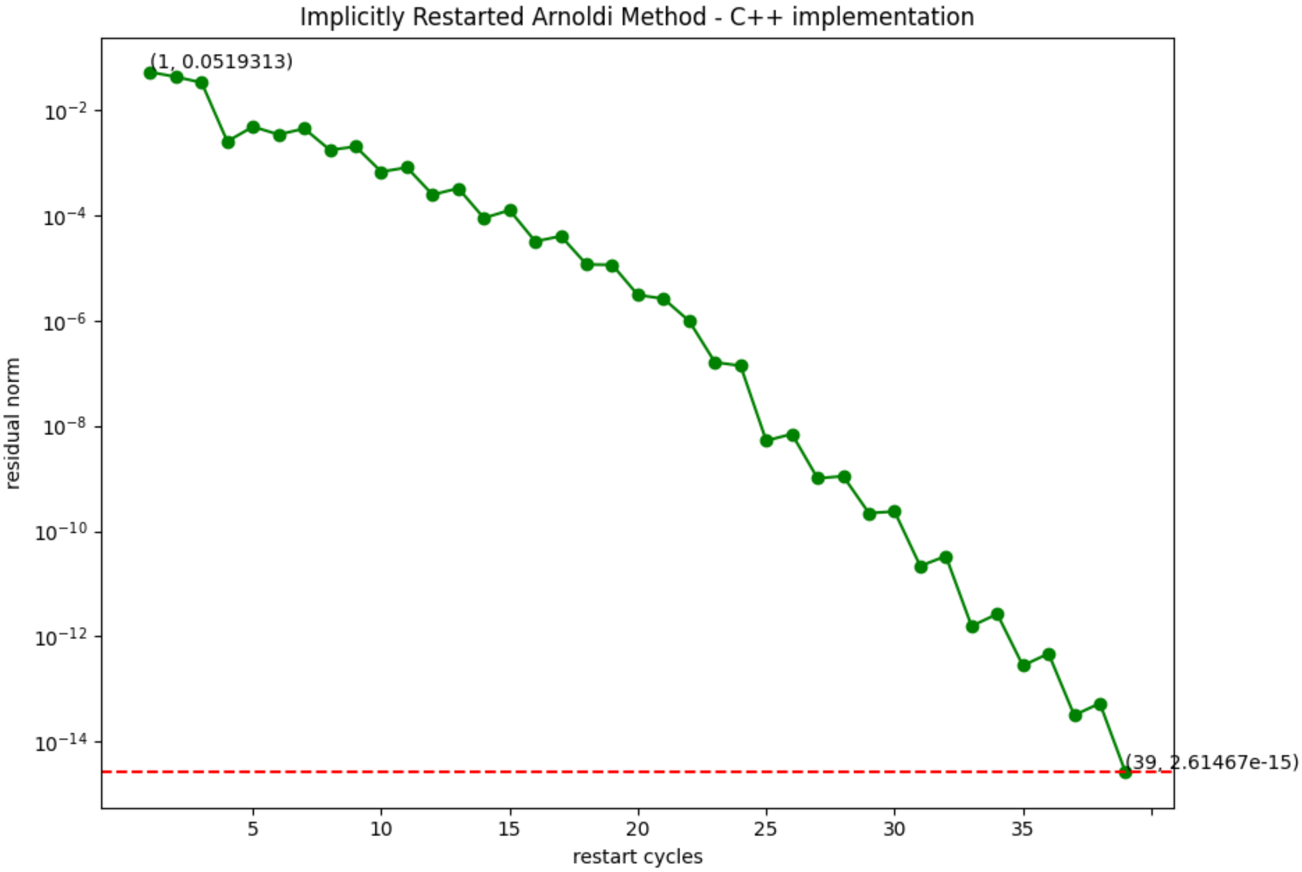
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Subspace	10
Tolerance	1e-14
Size of the matrice	62 (with 450 nonzero elements)


Implicitly Restarted Arnoldi Method
Residual norms/normf
10^{0}
10^{-5}
10^{-10}
10^{-15}
X 1
Y 0.0142056
X 8
Y 1.88989e-16
0
1
2
3
4
5
6
7
8
9
10
Iterations

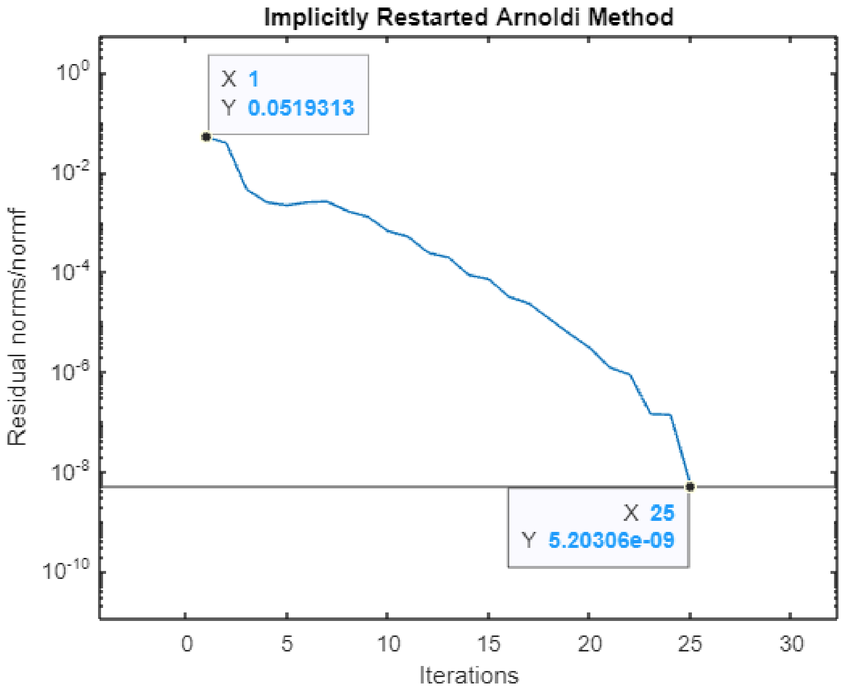

Implicitly Restarted Arnoldi Method - C++ implementation
residual norm
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
(1, 0.0142056)
(8, 1.89045e-16)
3
4
5
6
7
8
restart cycles

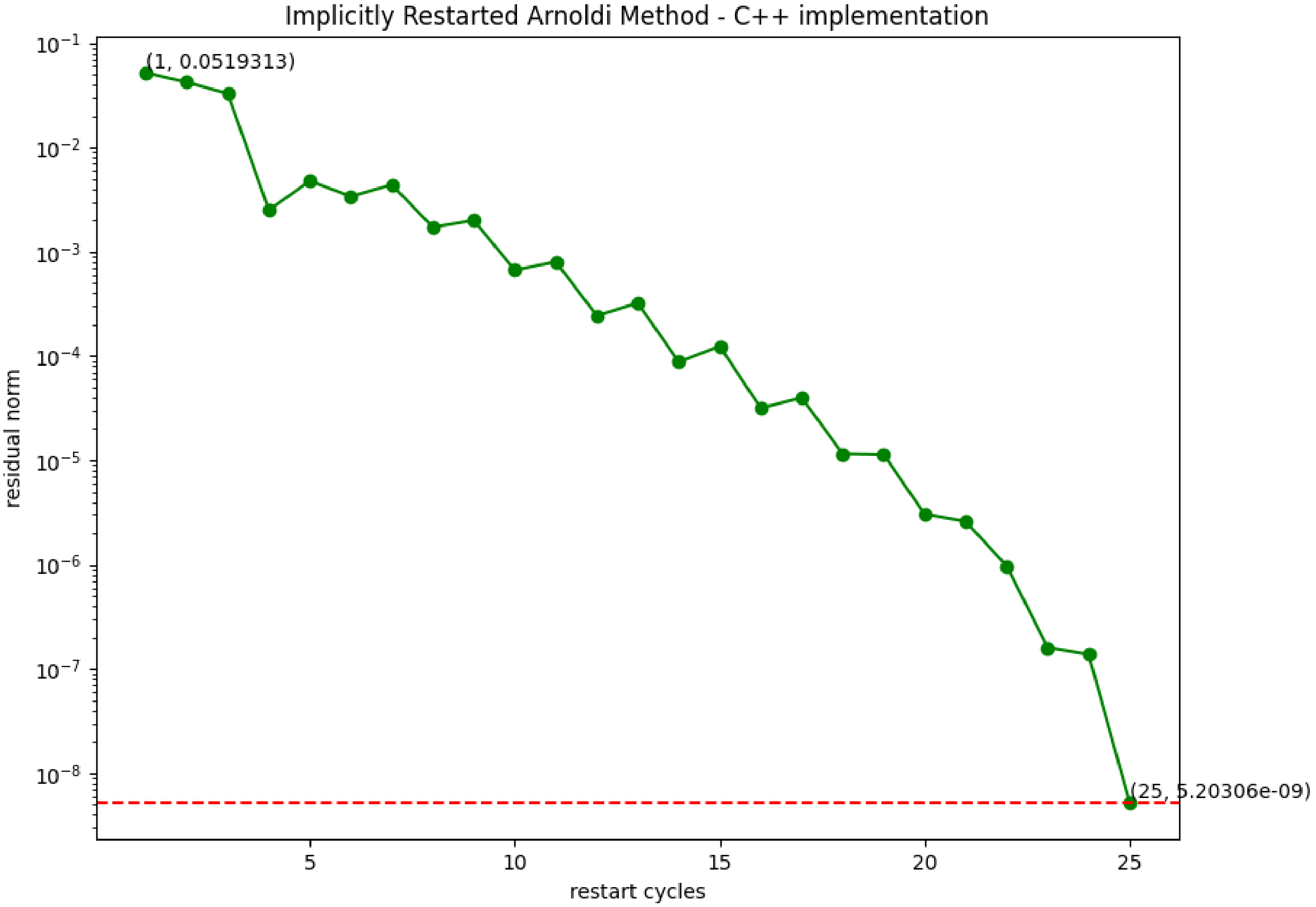
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	5
Subspace	10
Tolerance	1e-14
Size of the matrice	62 (with 450 nonzero elements)


Implicitly Restarted Arnoldi Method
X 1
Y 0.0519313
X 39
Y 2.61467e-15
Residual norms/normf
10^{0}
10^{-5}
10^{-10}
10^{-15}
-10
0
10
20
30
40
Iterations


Implicitly Restarted Arnoldi Method - C++ implementation
(1, 0.0519313)
(39, 2.61467e-15)
residual norm
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
5
10
15
20
25
30
35
restart cycles

Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	5
Subspace	10
Tolerance	1e-8
Size of the matrice	62 (with 450 nonzero elements)


Implicitly Restarted Arnoldi Method
Residual norms/normf
Iterations
X 1
Y 0.0519313
X 25
Y 5.20306e-09
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
0
5
10
15
20
25
30

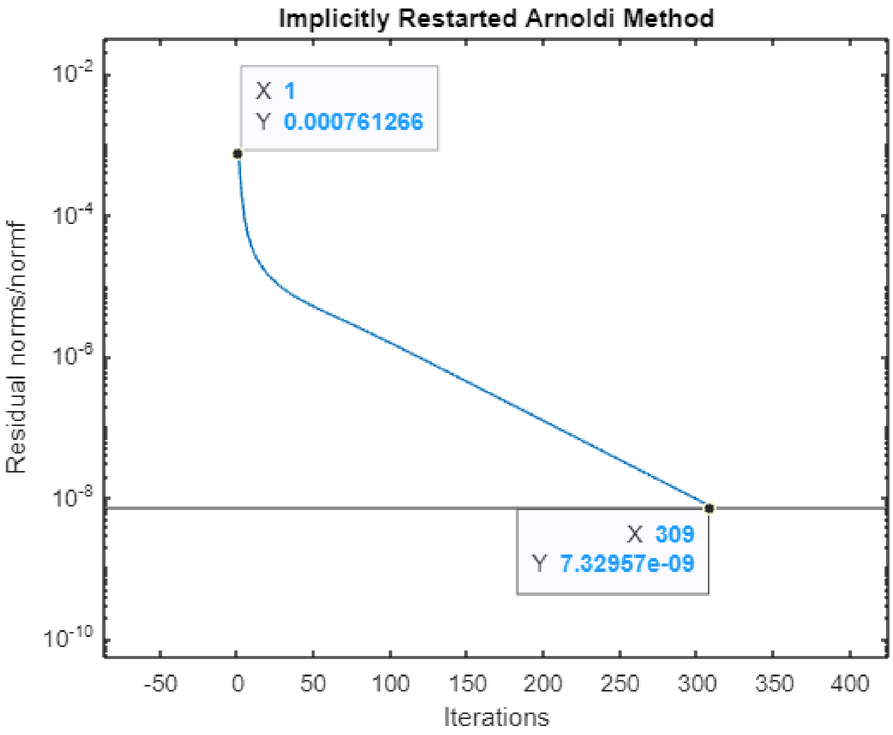

Implicitly Restarted Arnoldi Method - C++ implementation
residual norm
restart cycles
(1, 0.0519313)
(25, 5.20306e-09)
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
5
10
15
20
25

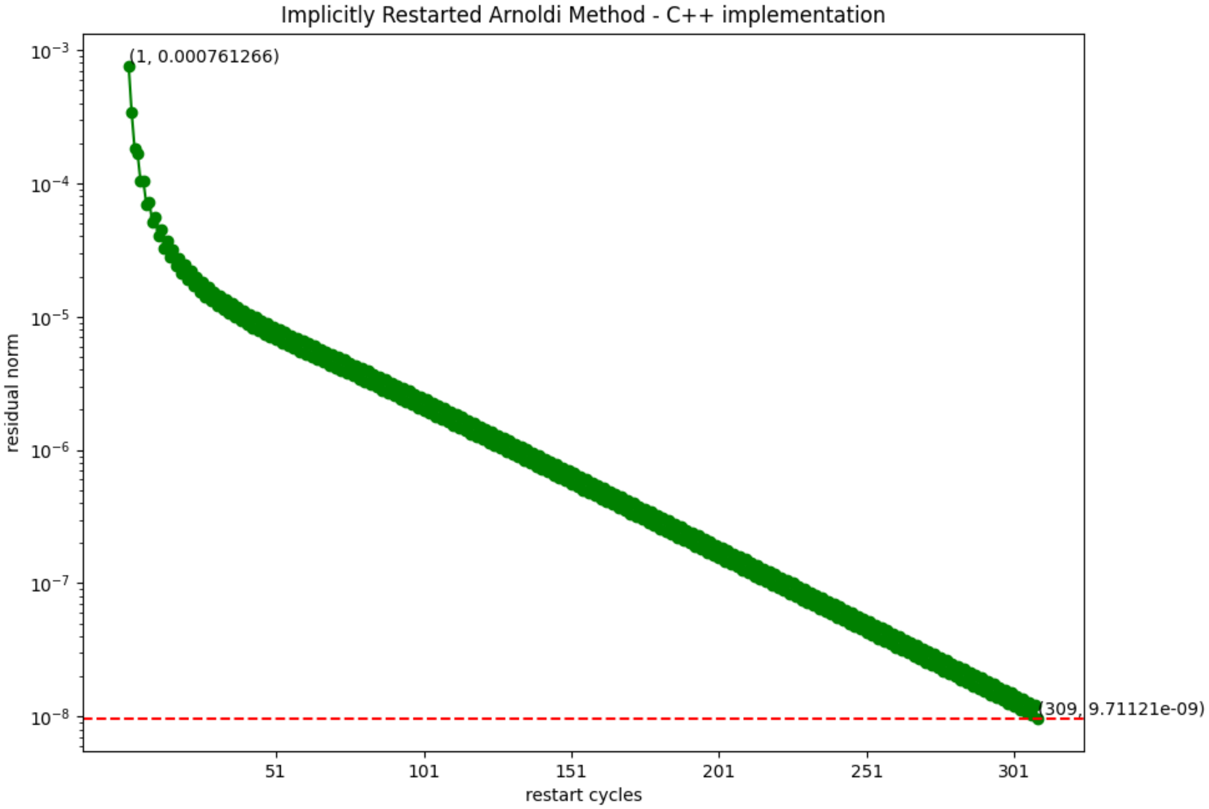
Note : *In order to symmetrize the matrix A, we applied the following formula:*

A = A + transpose(A) - Matrix(diagonal(A)).

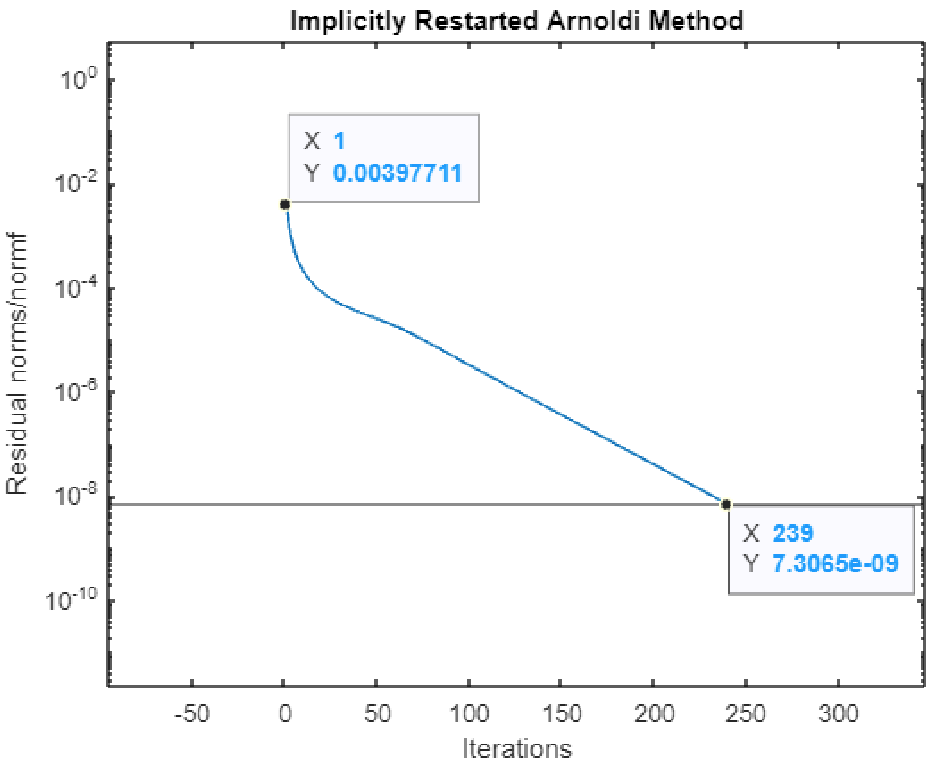
Tests with the "*A9_1000*" matrix : real & symmetric

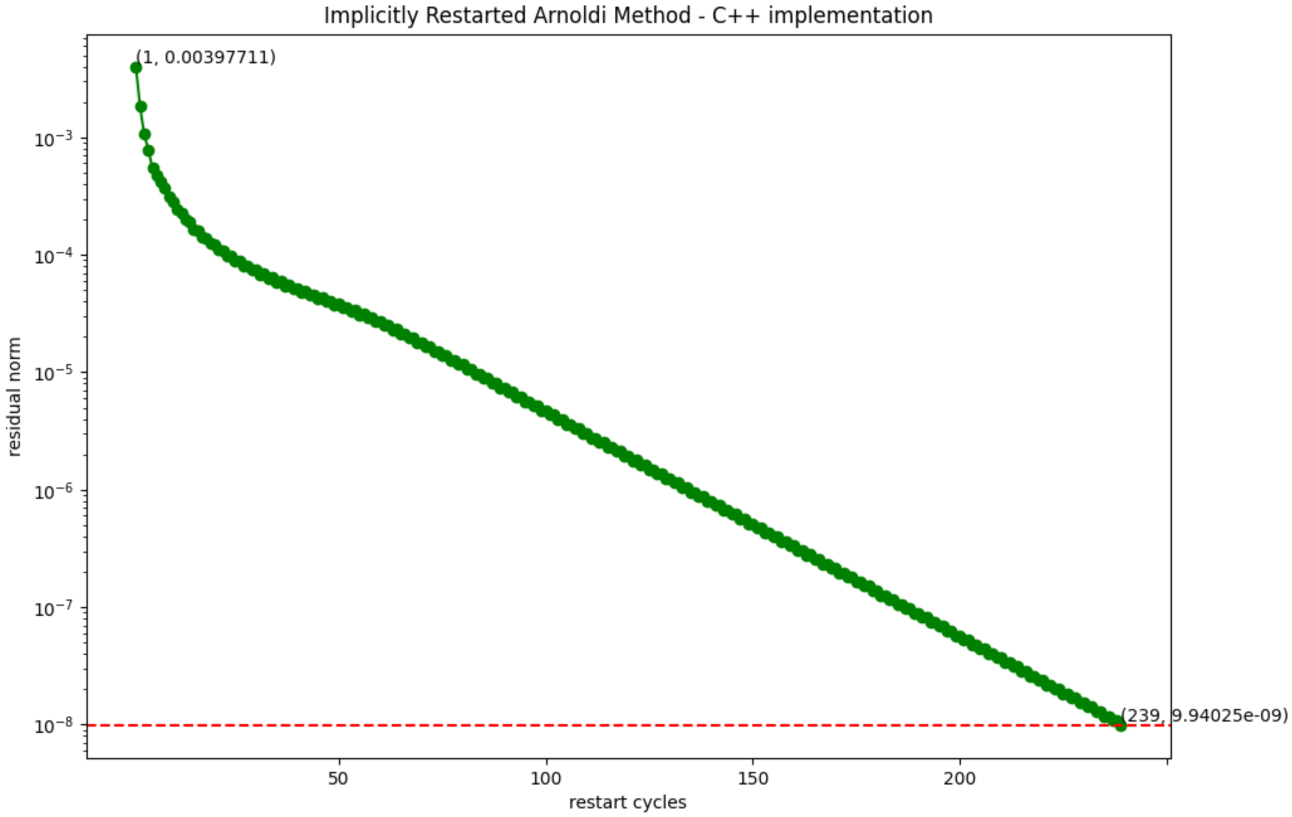
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Subspace	20
Tolerance	1e-8
Size of the matrice	1000 (with 2998 nonzero elements)


Implicitly Restarted Arnoldi Method
X 1
Y 0.000761266
X 309
Y 7.32957e-09
Residual norms/normf
Iterations
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
-50
0
50
100
150
200
250
300
350
400

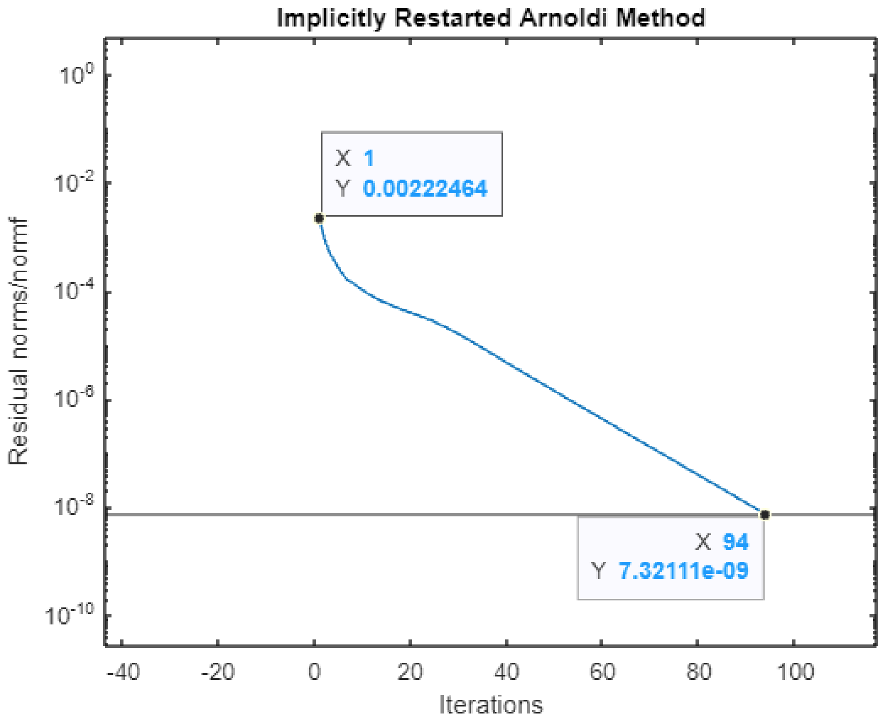

Implicitly Restarted Arnoldi Method - C++ implementation
(1, 0.000761266)
(309, 9.71121e-09)
residual norm
restart cycles
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
51
101
151
201
251
301

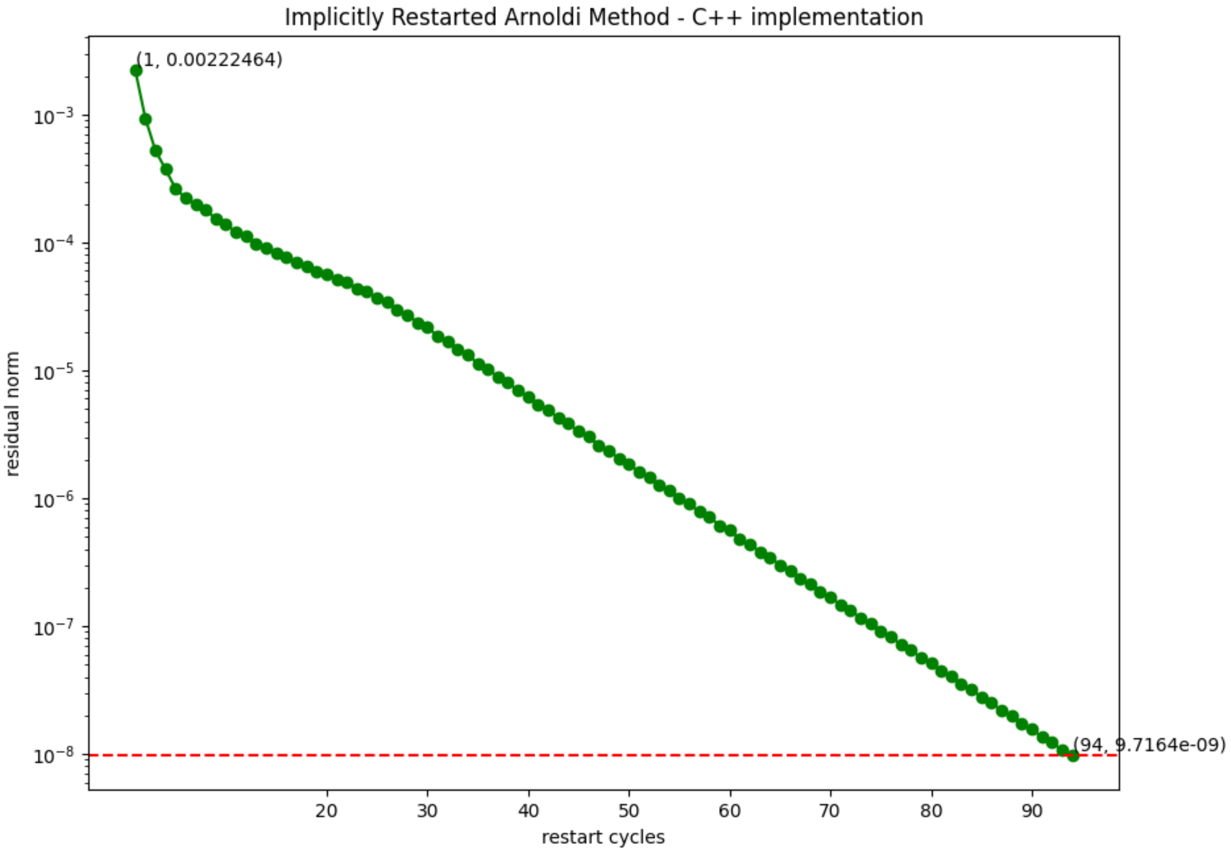
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	5
Subspace	20
Tolerance	1e-8
Size of the matrice	1000 (with 2998 nonzero elements)


Implicitly Restarted Arnoldi Method
Residual norms/normf
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
X 1
Y 0.00397711
X 239
Y 7.3065e-09
-50
0
50
100
150
200
250
300
Iterations


Implicitly Restarted Arnoldi Method - C++ implementation
residual norm
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
(1, 0.00397711)
(239, 9.94025e-09)
50
100
150
200
restart cycles

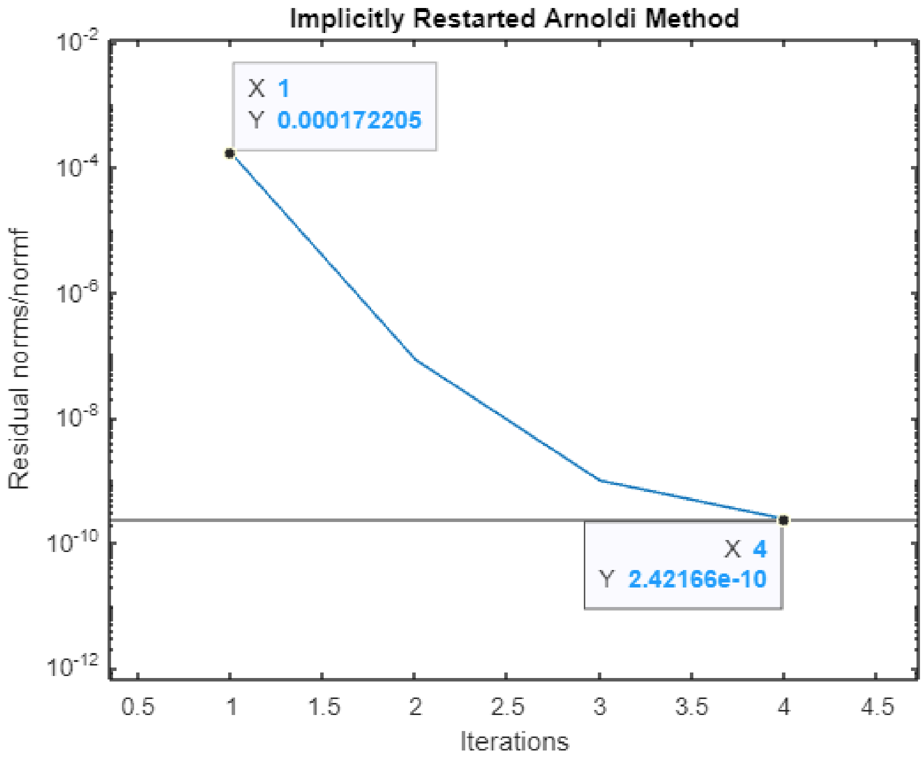
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	5
Subspace	30
Tolerance	1e-8
Size of the matrice	1000 (with 2998 nonzero elements)

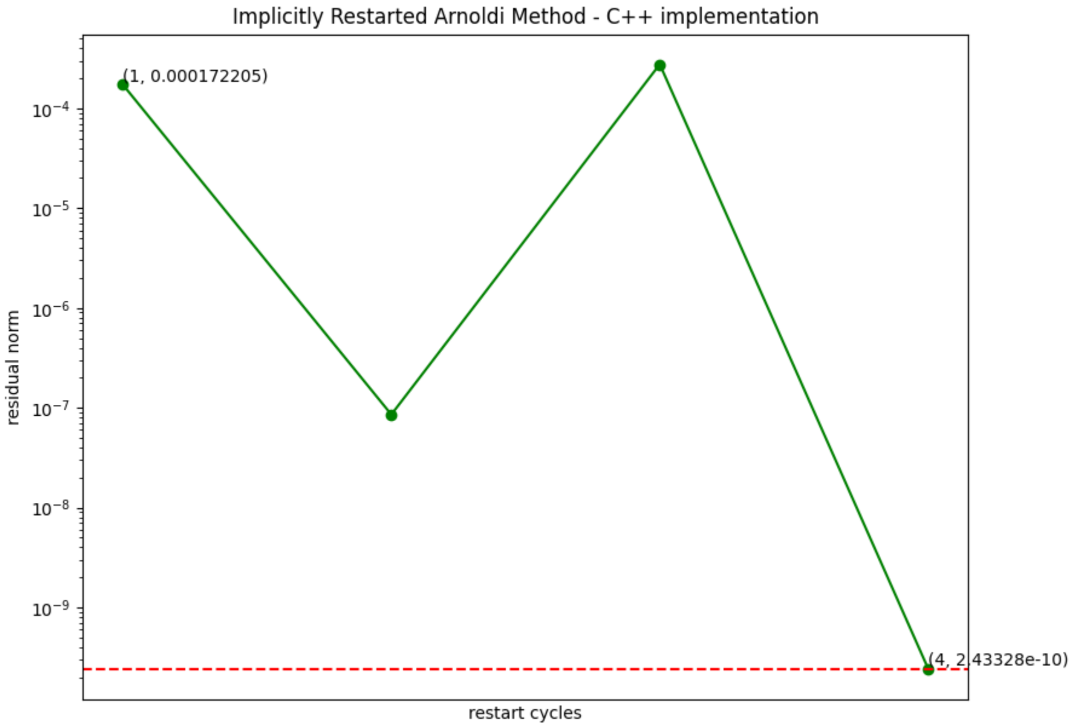

Implicitly Restarted Arnoldi Method


Implicitly Restarted Arnoldi Method - C++ implementation

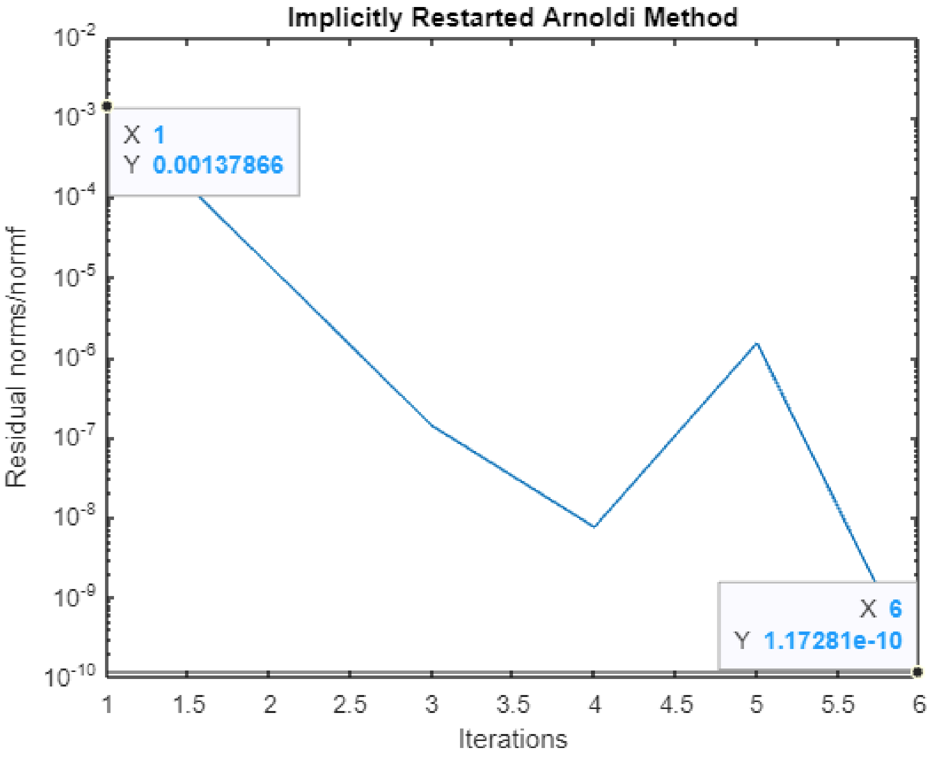
Tests with the “plat362” matrix : real & symmetric & indefinite

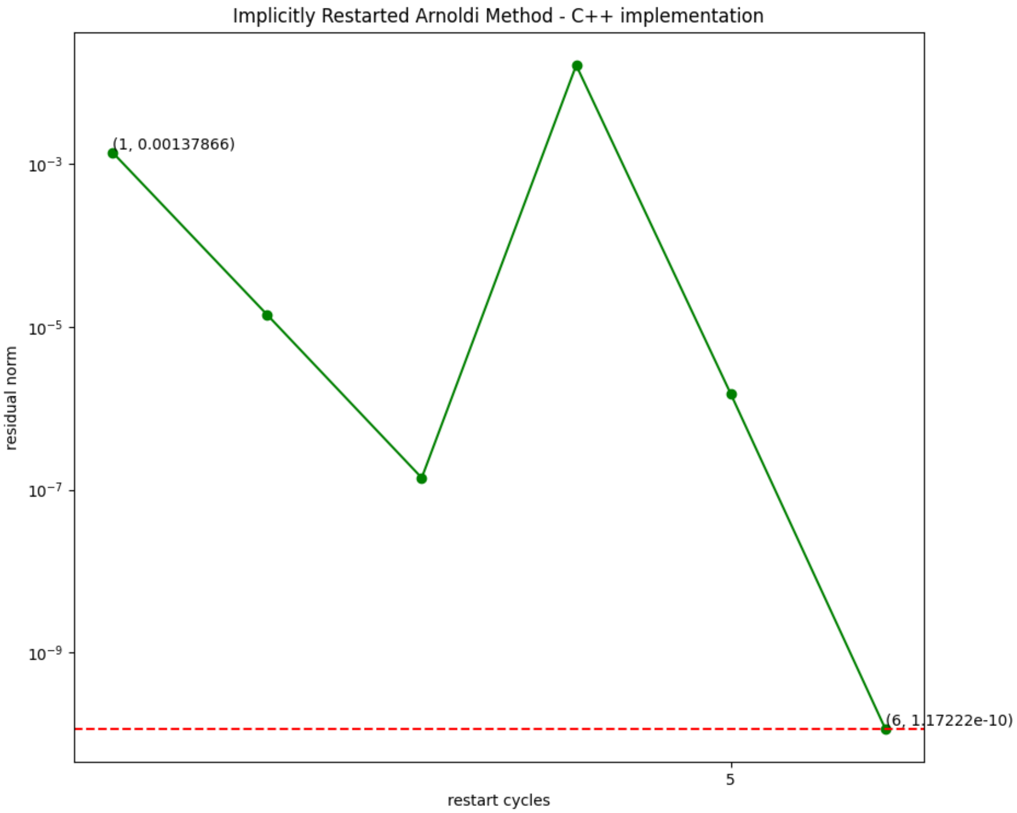
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Subspace	20
Tolerance	1e-8
Size of the matrice	362 (with 3074 nonzero elements)


Implicitly Restarted Arnoldi Method
X 1
Y 0.000172205
X 4
Y 2.42166e-10
Residual norms/normf
10^-2
10^-4
10^-6
10^-8
10^-10
10^-12
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Iterations

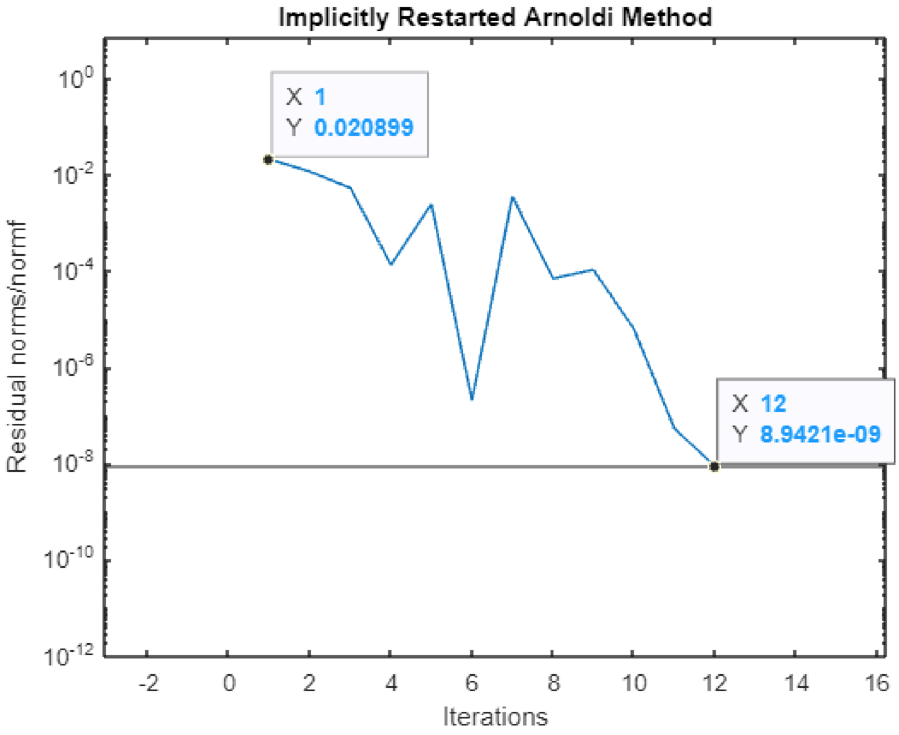

Implicitly Restarted Arnoldi Method - C++ implementation
(1, 0.000172205)
(4, 2.43328e-10)
residual norm
10^-4
10^-5
10^-6
10^-7
10^-8
10^-9
restart cycles

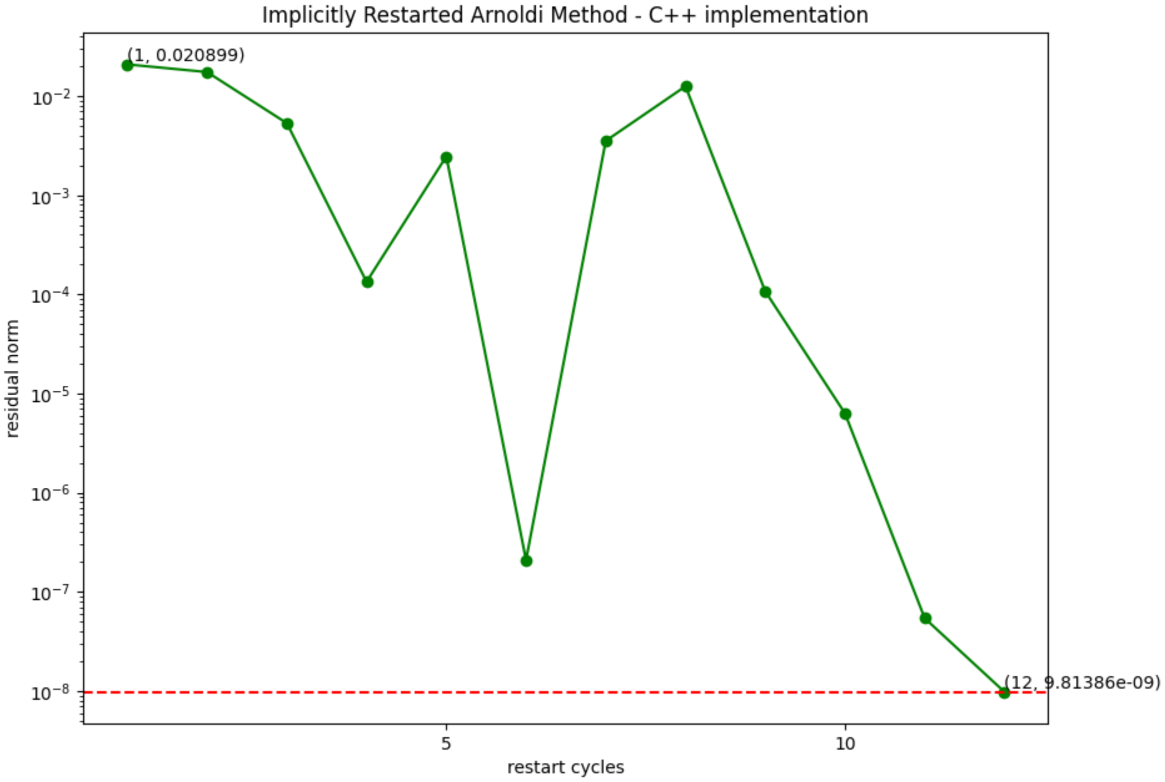
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Subspace	15
Tolerance	1e-8
Size of the matrice	362 (with 3074 nonzero elements)


Implicitly Restarted Arnoldi Method
Residual norms/normf
Iterations
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
1
1.5
2
2.5
3
3.5
4
4.5
5
5.5
6
X 1
Y 0.00137866
X 6
Y 1.17281e-10


Implicitly Restarted Arnoldi Method - C++ implementation
residual norm
restart cycles
10^{-3}
10^{-5}
10^{-7}
10^{-9}
5
(1, 0.00137866)
(6, 1.17222e-10)

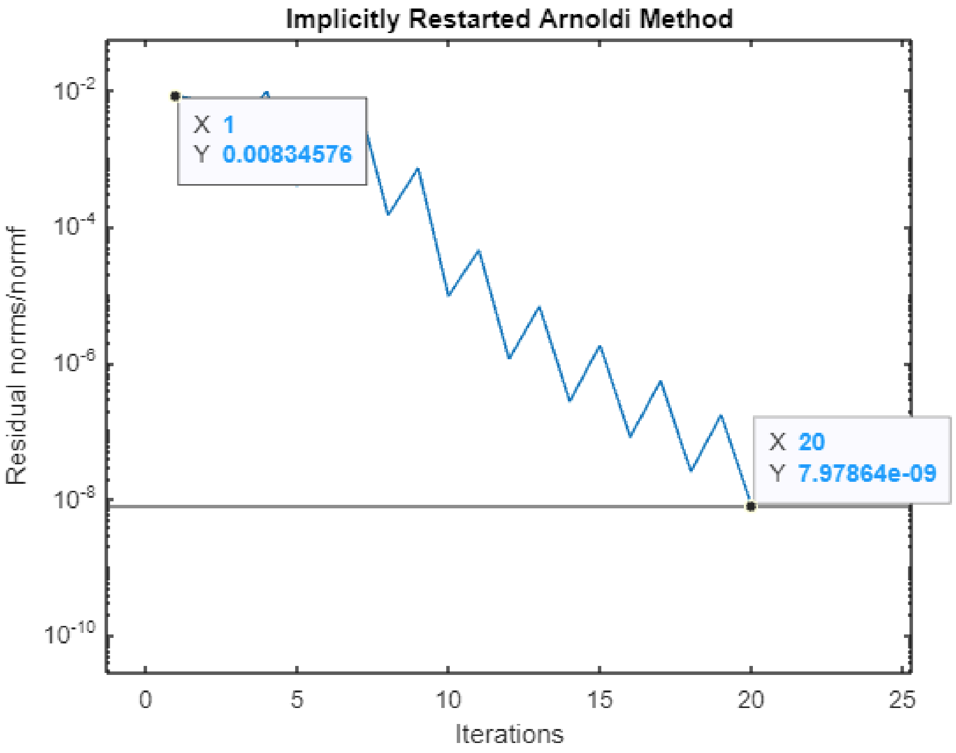
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	7
Subspace	15
Tolerance	1e-8
Size of the matrice	362 (with 3074 nonzero elements)

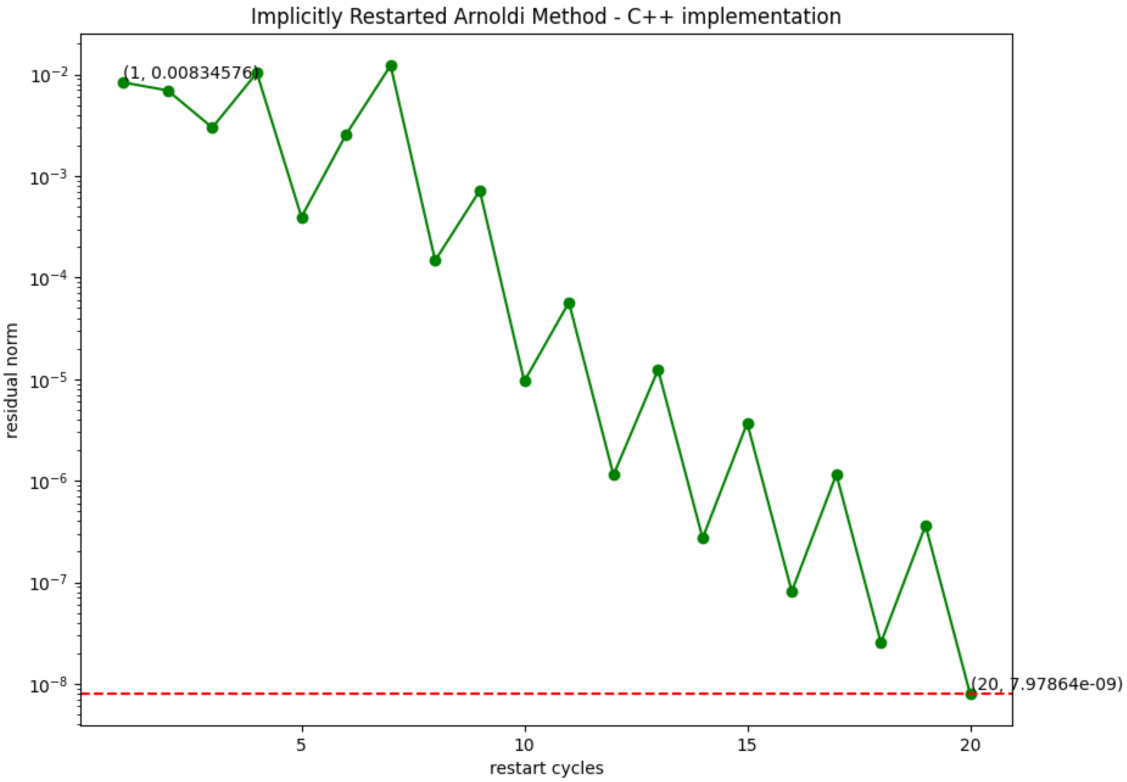

Implicitly Restarted Arnoldi Method
Residual norms/normf
Iterations
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
-2
0
2
4
6
8
10
12
14
16
X 1
Y 0.020899
X 12
Y 8.9421e-09


Implicitly Restarted Arnoldi Method - C++ implementation
residual norm
restart cycles
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
5
10
(1, 0.020899)
(12, 9.81386e-09)

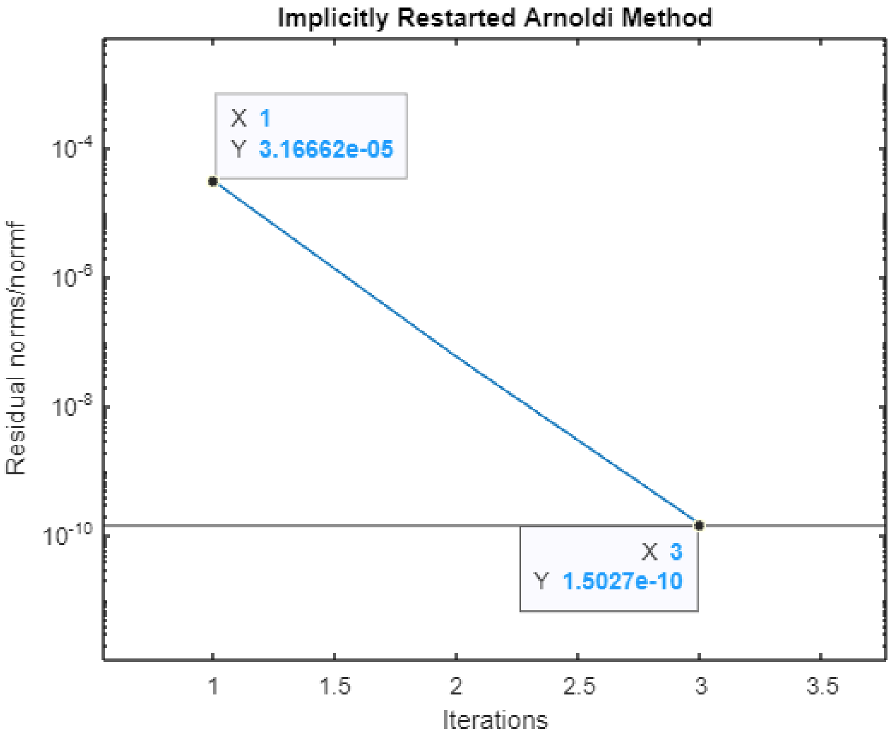
Tests with the "*bcsstk06*" matrix : real & symmetric & positive & definite

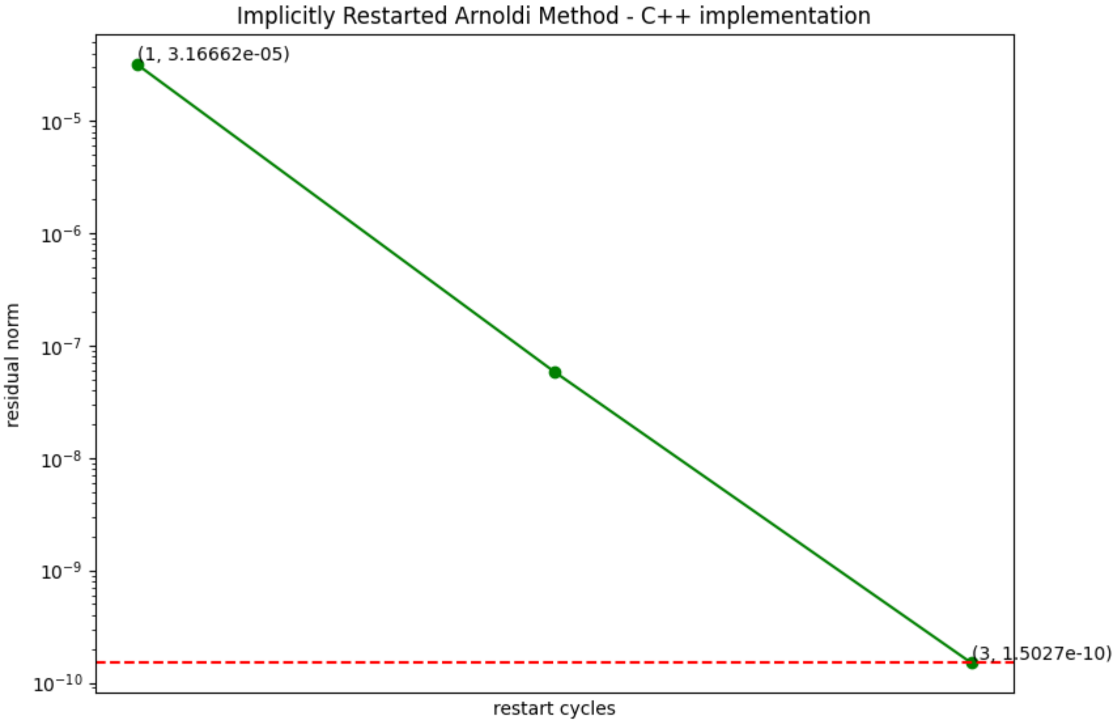
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	7
Subspace	15
Tolerance	1e-8
Size of the matrice	420 (with 4140 nonzero elements)


Implicitly Restarted Arnoldi Method
Residual norms/normf
Iterations
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
0
5
10
15
20
25
X 1
Y 0.00834576
X 20
Y 7.97864e-09


Implicitly Restarted Arnoldi Method - C++ implementation
residual norm
restart cycles
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
5
10
15
20
(1, 0.00834576)
(20, 7.97864e-09)

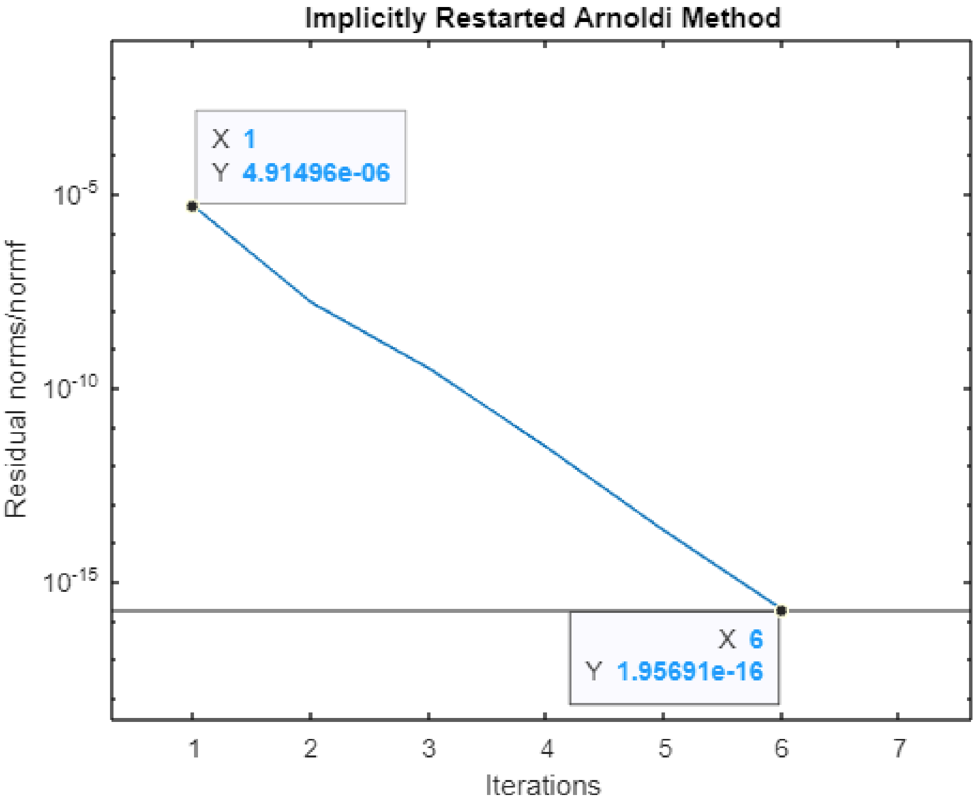
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Subspace	50
Tolerance	1e-8
Size of the matrice	420 (with 4140 nonzero elements)

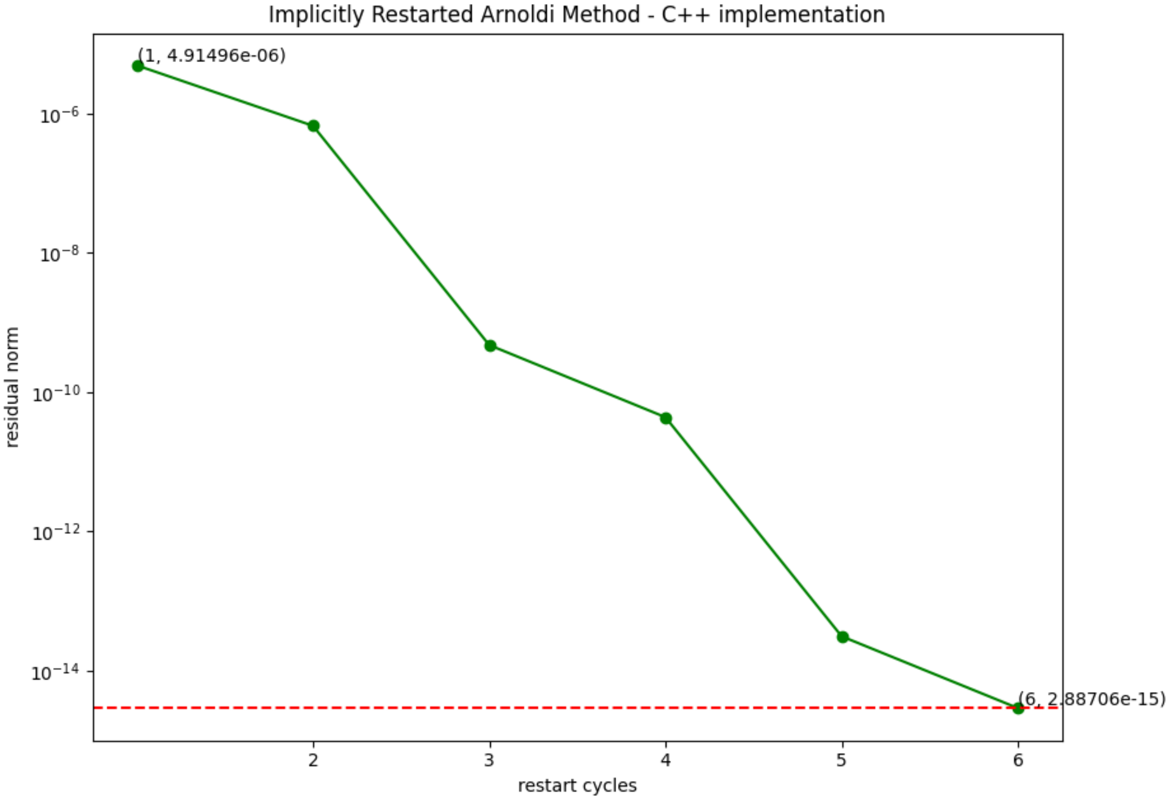

Implicitly Restarted Arnoldi Method
Residual norms/normf
Iterations
10^{-4}
10^{-6}
10^{-8}
10^{-10}
1
1.5
2
2.5
3
3.5
X 1
Y 3.16662e-05
X 3
Y 1.5027e-10


Implicitly Restarted Arnoldi Method - C++ implementation
residual norm
restart cycles
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
(1, 3.16662e-05)
(3, 1.5027e-10)

Tests with the "*mhd4800b*" matrix : real & symmetric & indefinite

Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	5
Subspace	15
Tolerance	1e-14
Size of the matrice	4800 (with 27520 nonzero elements)


Implicitly Restarted Arnoldi Method
X 1
Y 4.91496e-06
X 6
Y 1.95691e-16
10^-5
10^-10
10^-15
Residual norms/normf
1
2
3
4
5
6
7
Iterations


Implicitly Restarted Arnoldi Method - C++ implementation
(1, 4.91496e-06)
(6, 2.88706e-15)
10^-6
10^-8
10^-10
10^-12
10^-14
residual norm
2
3
4
5
6
restart cycles

Testing & validation of MIRAMns - Multiple Implicitly Restarted Arnoldi Method with Nested Subspaces

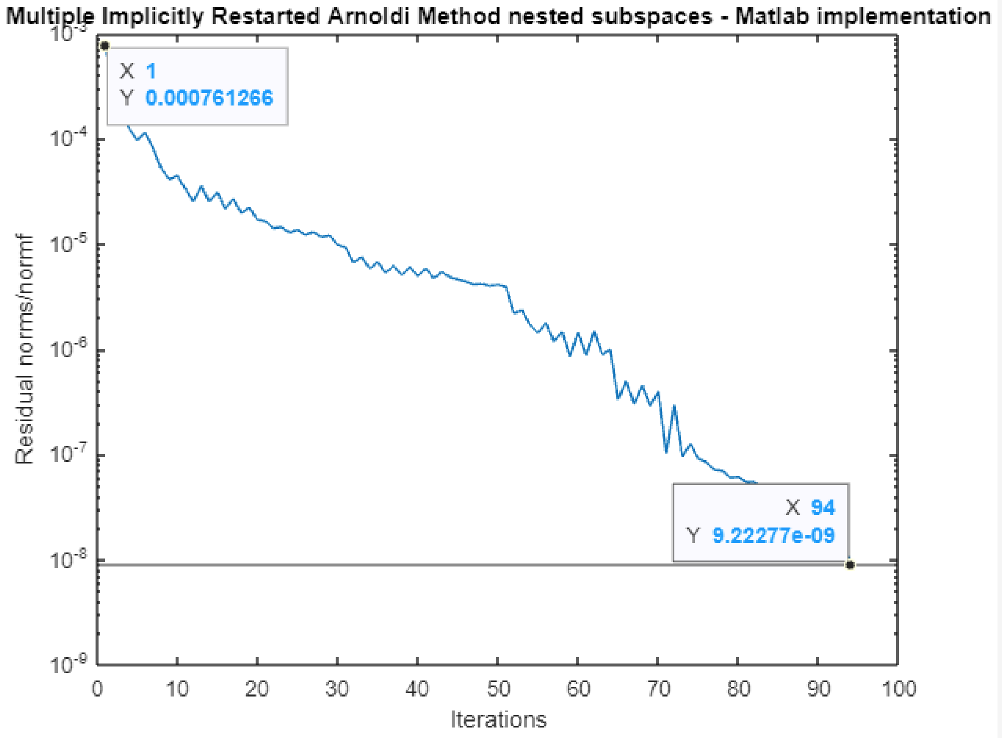
In order to validate our M*IRAM* implementation, we compared the convergence of our MIRAM implementation (in c++) with the MIRAM implementation (in matlab) of the paper "A KEY TO CHOOSE SUBSPACE SIZE IN IMPLICITLY RESTARTED ARNOLDI METHOD" published by PhD. Fazeli, Prof. Emad, and Dr. Liu.

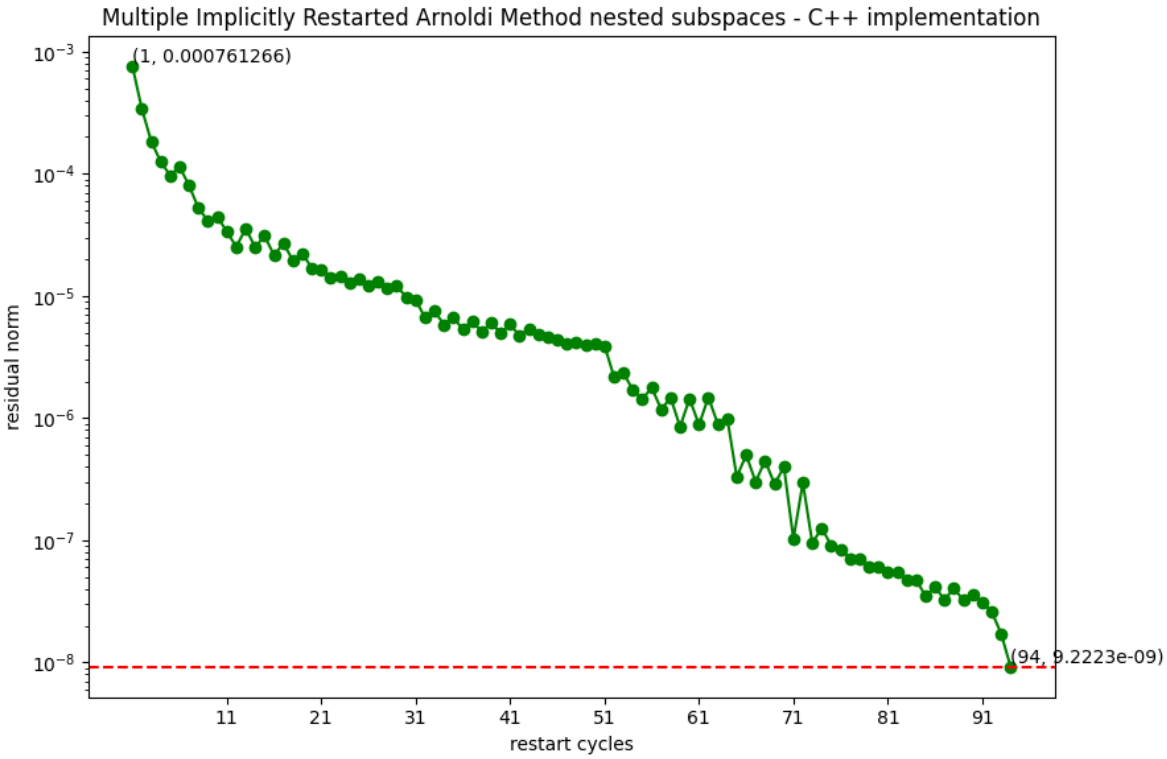
In this document, we tested our implementation with 2 types of matrices : **real & unsymmetric** and **real & symmetric**. To have a valid comparison, we took the same initial conditions and the same matrices for each of the tests.

Each main part of this document represents a test with a given matrix. The tests with different initial conditions therefore represent the sub-parts. Each time, the coordinates of the 1st and the last points of the graphs are given in order to have an order of magnitude of the results of the 2 implementations (Matlab & C++). In the graphs of this document, the last point of each graph is highlighted with a horizontal line.

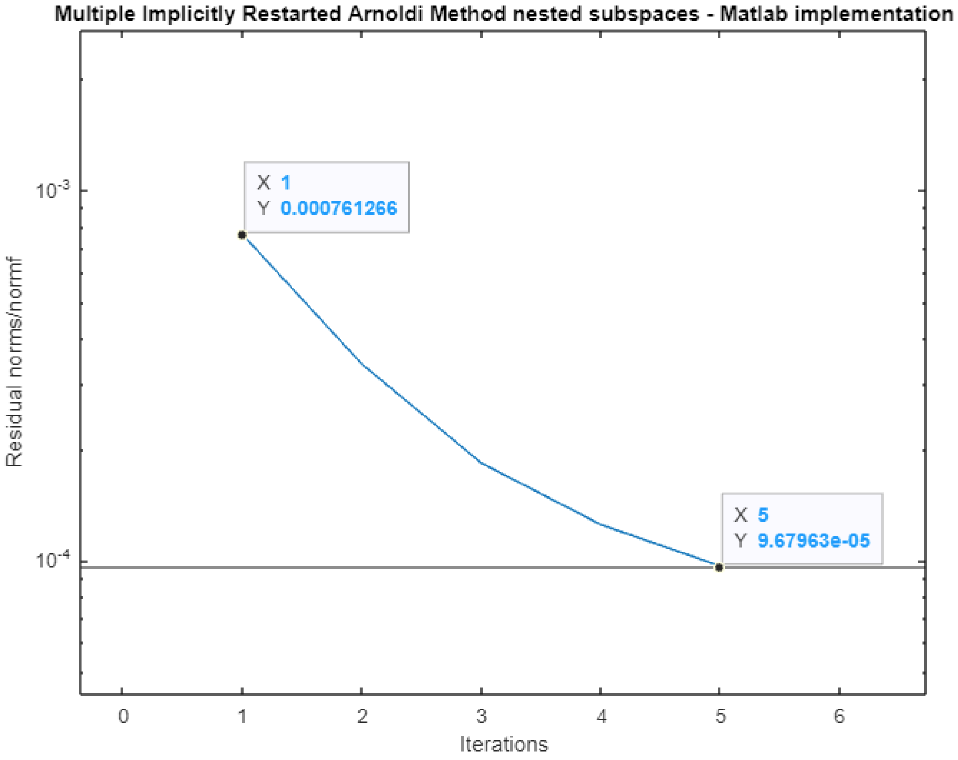
Tests with the “*A9_1000*” matrix : real & symmetric

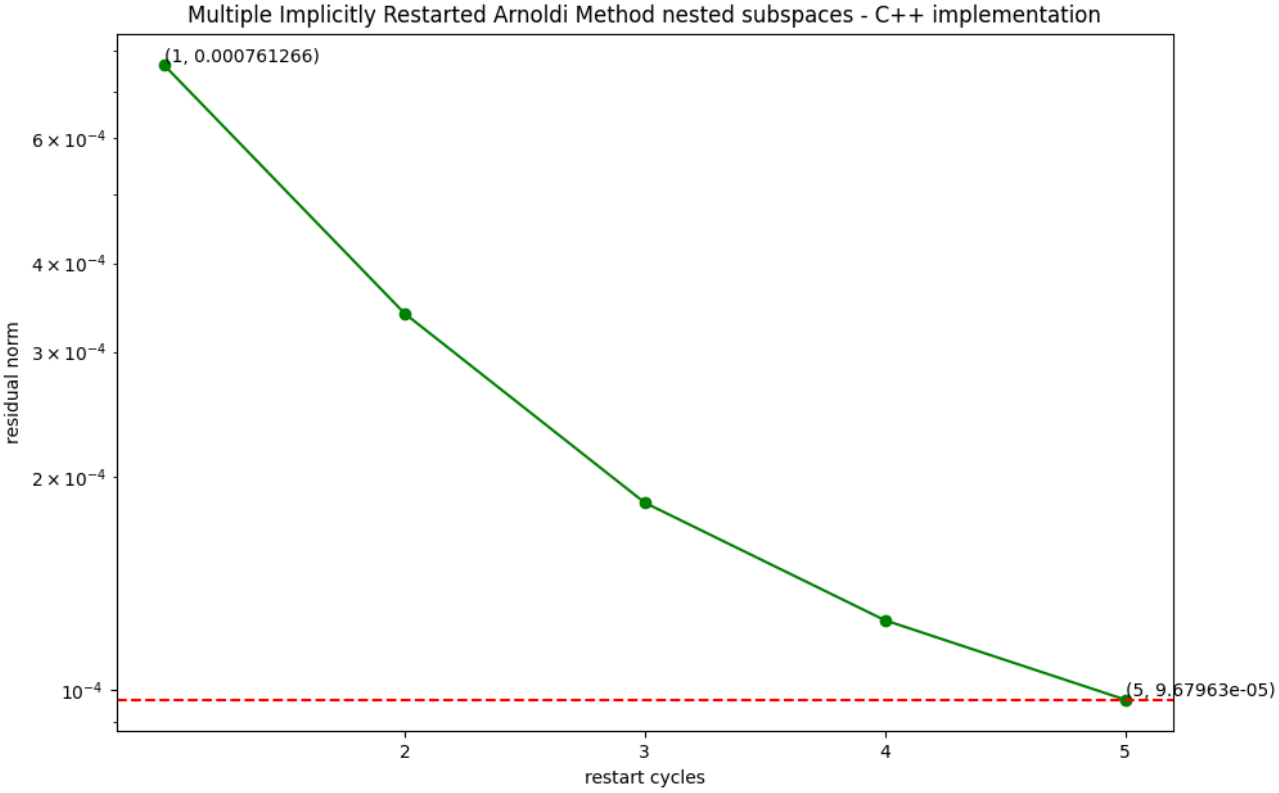
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Nested subspaces	10, 15, 20
Tolerance	1e-8
Size of the matrice	1000 (with 2998 nonzero elements)


Multiple Implicitly Restarted Arnoldi Method nested subspaces - Matlab implementation
Residual norms/normf
Iterations
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
0
10
20
30
40
50
60
70
80
90
100
X 1
Y 0.000761266
X 94
Y 9.22277e-09


Multiple Implicitly Restarted Arnoldi Method nested subspaces - C++ implementation
residual norm
restart cycles
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
11
21
31
41
51
61
71
81
91
(1, 0.000761266)
(94, 9.2223e-09)

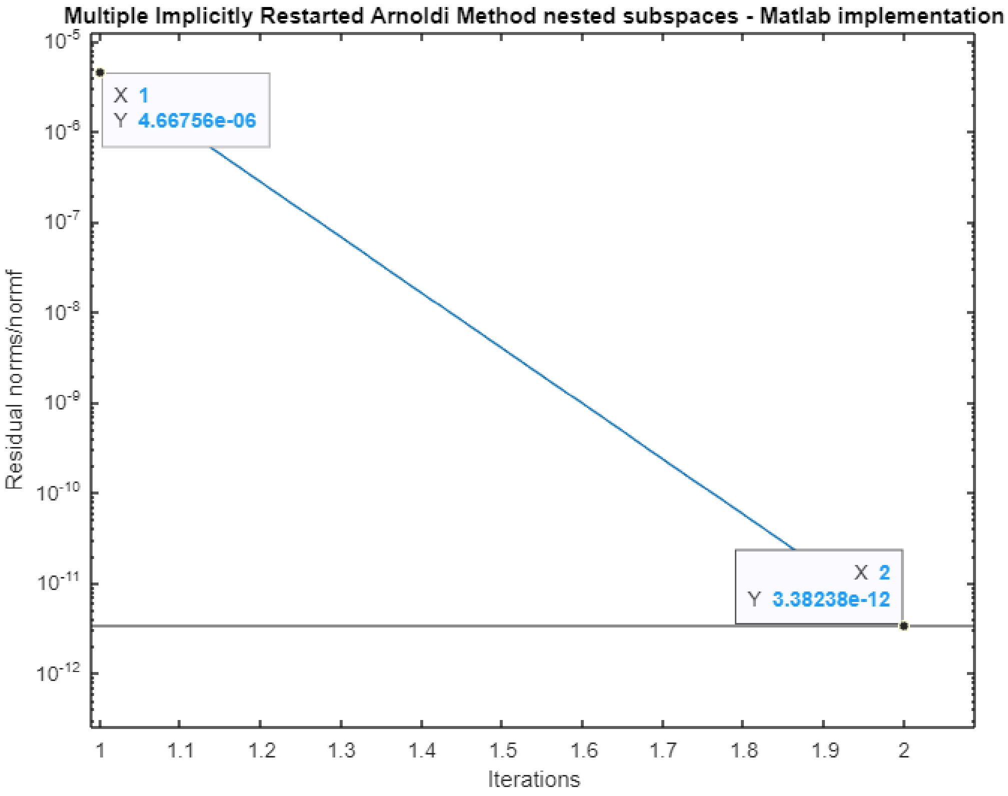
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Nested subspaces	10, 15, 20
Tolerance	1e-4
Size of the matrice	1000 (with 2998 nonzero elements)

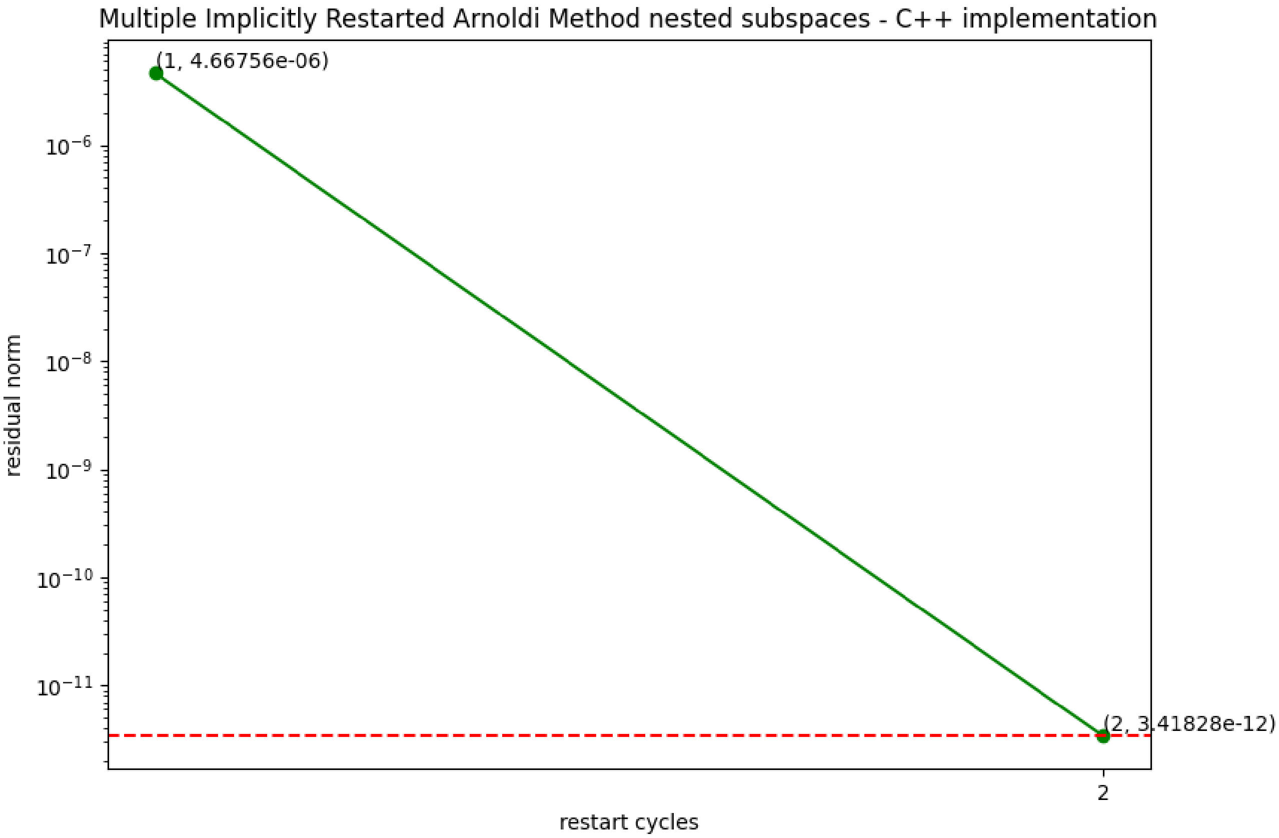

Multiple Implicitly Restarted Arnoldi Method nested subspaces - Matlab implementation
X 1
Y 0.000761266
X 5
Y 9.67963e-05
10^{-3}
10^{-4}
Residual norms/normf
0
1
2
3
4
5
6
Iterations


Multiple Implicitly Restarted Arnoldi Method nested subspaces - C++ implementation
(1, 0.000761266)
(5, 9.67963e-05)
6×10^{-4}
4×10^{-4}
3×10^{-4}
2×10^{-4}
10^{-4}
residual norm
2
3
4
5
restart cycles

Tests with the “*sherman3*” matrix : real & unsymmetric

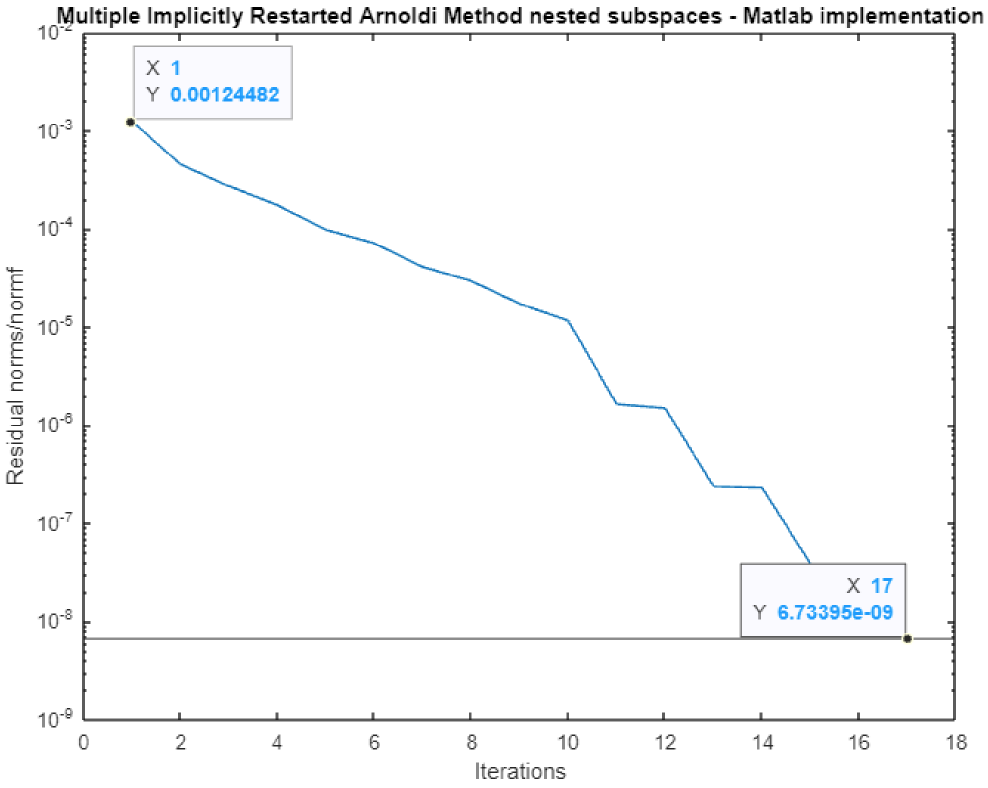
Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Nested subspaces	5, 8, 10
Tolerance	1e-8
Size of the matrice	5005 (with 20033 nonzero elements)


Multiple Implicitly Restarted Arnoldi Method nested subspaces - Matlab implementation
Residual norms/normf
Iterations
X 1
Y 4.66756e-06
X 2
Y 3.38238e-12
10^-5
10^-6
10^-7
10^-8
10^-9
10^-10
10^-11
10^-12
1
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2

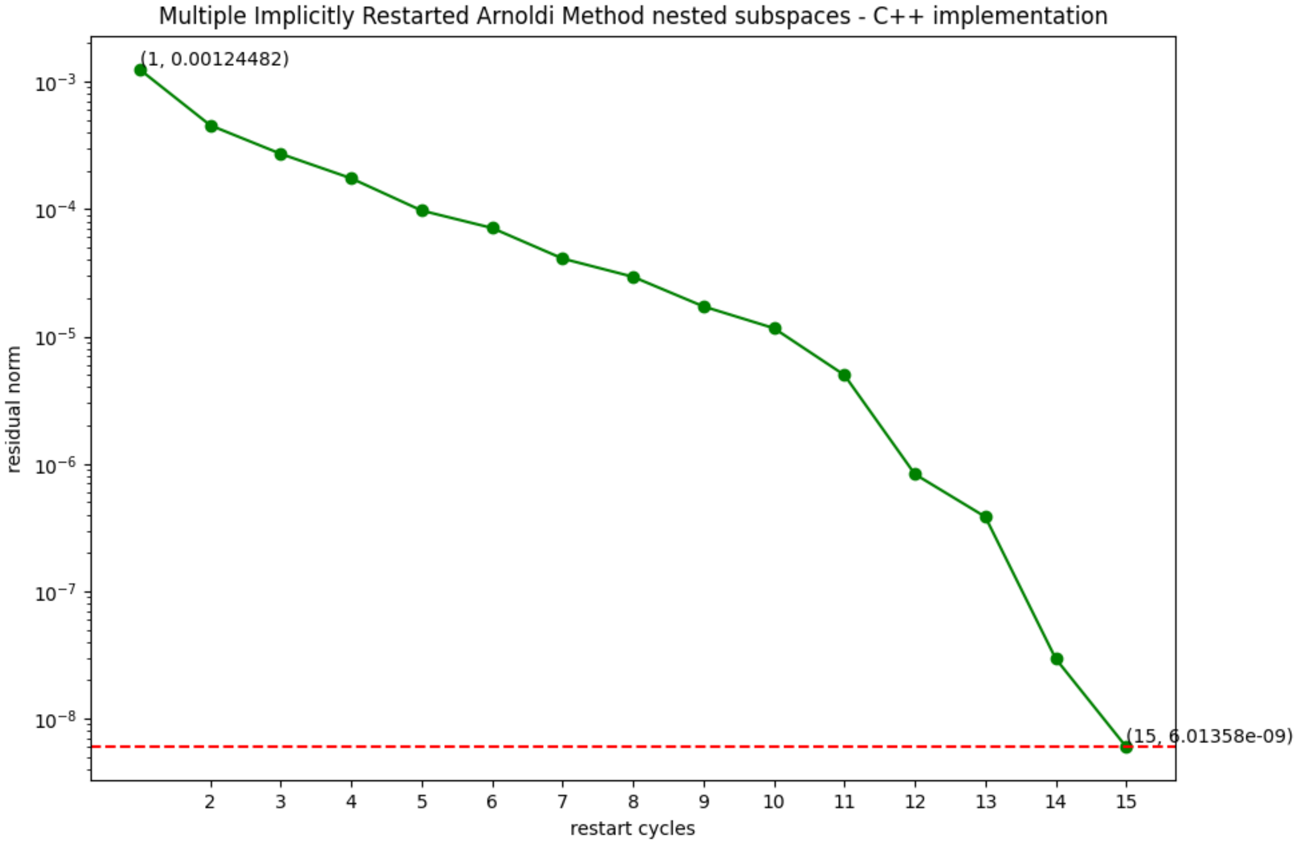

Multiple Implicitly Restarted Arnoldi Method nested subspaces - C++ implementation
residual norm
restart cycles
(1, 4.66756e-06)
(2, 3.41828e-12)
10^-6
10^-7
10^-8
10^-9
10^-10
10^-11
2

Tests with the "*AM_1000*" matrix : real & unsymmetric

Initial vector (residual vector)	1/nrmfr (with nrmfr = A matrix norm)
Number of eigenelements	2
Nested subspaces	13, 17, 20
Tolerance	1e-8
Size of the matrice	5005 (with 20033 nonzero elements)



Multiple Implicitly Restarted Arnoldi Method nested subspaces - Matlab implementation



Multiple Implicitly Restarted Arnoldi Method nested subspaces - C++ implementation

Abstract

This report is produced as part of a 6-month internship at Huawei Paris from March 6, 2023 to August 31, 2023.

On a technical level, the general theme of the internship is the interaction between HPC and AI. The goal is to propose an improvement in the choice of hyperparameters of a given deep learning model so that its learning is faster while not significantly losing precision.

The internship is mainly divided into 2 parts. The first part is to take an interest in the MIRAMns numerical method. The second part is its application to the field of AI in order to meet our objective. For this, we took the recommendation system, the reference in the research community, wide and deep learning.

On a personal level, this internship is a very good experience. In fact, this experience not only gave me another step in the field of scientific research, but also learned a lot of technological tools to develop a POC in the field of HPC & AI.

Regarding the improvements to be made, there are many tracks. At this stage, I propose to optimize in time and space our MIRAMns implementation so that it is applicable with much larger inputs, of NLP deep learning models for example like GPT, in order to confirm our innovation. Thus, MIRAMns can be democratized in the field of dimension reduction and can be an alternative to PCA for example.

Résumé

Ce rapport est réalisé dans le cadre d'un stage de 6 mois chez Huawei Paris du 6 Mars 2023 au 31 Août 2023.

Sur le plan technique, le thème général du stage est l'interaction entre HPC et IA. L’objectif est de proposer une amélioration du choix des hyperparamètres d’un modèle de deep learning donné afin que son apprentissage soit plus rapide tout en ne pas perdre en précision de manière significative.

Le stage est principalement divisé en 2 parties. La première partie est de s'intéresser à la méthode numérique MIRAMns. La deuxième partie est son application au domaine de l’IA afin de répondre à notre objectif. Pour cela, nous avons pris le système de recommandation, la référence dans la communauté des chercheurs, le Wide et le Deep Learning.

Sur le plan personnel, ce stage est une très bonne expérience. Cette expérience m'a non seulement permis de franchir une autre étape dans le domaine de la recherche scientifique, mais m'a également permis d'apprendre de nombreux outils technologiques pour développer un POC dans le domaine du HPC & IA.

Concernant les améliorations à apporter, les pistes sont nombreuses. Pour le moment, je propose d'optimiser en temps et espace notre implémentation de MIRAMns pour qu'elle soit applicable avec des inputs beaucoup plus larges, de modèles de deep learning NLP par exemple comme GPT, dans le but de confirmer notre innovation. Ainsi, MIRAMns peuvent être démocratisés dans le domaine de la réduction de dimensions et pourrait être une alternative au PCA par exemple.