Analyse Numérique : Devoir 4
GMRES préconditionné

1 Algorithme

L'algorithme *GMRES* trouve la solution du système $Ax = b$ de manière itérative en minimisant le résidu dans κ_n.

Data: Le problème matriciel $Ax = b$.
Result: Une solution approchée, x_n.
begin
  $e_1 \leftarrow (1, 0, \ldots, 0)$
  $r_0 \leftarrow M^{-1}(b - Au_0)$
  $q_1 \leftarrow r_0 / \|r_0\|_2$
  for $n = 1, 2, 3, \ldots$ **do**
    Trouver $\tilde{H}_n$ et Q_n par Arnoldi
    Trouver y qui minimise $\left\|\tilde{H}_n y - \|r_0\|_2 e_1\right\|_2$
    $x_n \leftarrow Q_n y$
  end
  return x_n
end

Algorithm 1: Generalized minimal residual method

A l'itération n, on approxime la solution finale par $x_n \in \kappa_n$ qui minimise la norme du résidu $r_n = b - Ax_n$ comme:

$$min \|r_n\| \tag{1}$$
$$\|b - AQ_n y\| \tag{2}$$
$$\left\|\tilde{H}_n y - \|r_0\|_2 e_1\right\|_2 \tag{3}$$

A cette itération, les dimensions des différentes matrices sont de :
- $A \in \mathbb{C}^{m \times m}$, $b \in \mathbb{C}^m$, $x_n \in \mathbb{C}^m$
- $Q_n \in \mathbb{C}^{m \times n+1}$: Q_n étant la base orthonormée pour les sous-espaces de Krylov successifs du système $Ax = b$

$$\kappa_n = \langle b, Ab, \ldots, A^{n-1}b \rangle = \langle q_1, \ldots, q_n \rangle \subseteq \mathbb{C}^n$$

- $\tilde{H}_n \in \mathbb{C}^{n+1 \times m}$: $\tilde{H}_n$ étant la matrice triangulaire supérieure d'Hessenberg avec une ligne de plus sous la diagonale.
- $e_1 \in \mathbb{N}^{n+1}$: e_1 valant $(1, 0, \ldots, 0)$ puisqu'on a choisi $q_1 = \dfrac{r_0}{\|r_0\|_2}$ pour initialiser Arnoldi et que tous les autres q_k sont perpendiculaires à q_1 avec $k > 1$ (Espace de Krylov).

2 Préconditionnement

Étant donné que GMRES est une méthode itérative, il serait intéressant de jouer sur la convergence de cet algorithme. Pour ce faire, il est possible d'utiliser un préconditonneur (à gauche avec *ILU0* par exemple), le système à résoudre devient donc $M^{-1}Ax = M^{-1}b$. Un bon préconditionneur vient condenser le spectre des valeurs propres de la matrice $M^{-1}A$ autour de 0. On peut voir sur la figure 1 ci-dessous que *ILU0* est un bon préconditionneur car il condense bien les valeurs propres autour de 0.

On peut également voir dans le tableau 3 que la convergence de GMRES avec le préconditonneur *ILU0* converge radicalement plus vite, la convergence dépendant du spectre de la matrice.

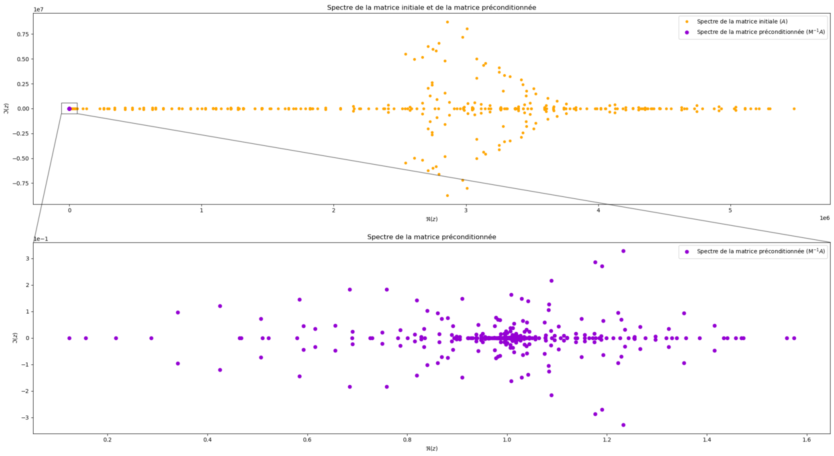

Figure 1: Spectre de A et $M^{-1}A$

3 Convergence

Comme représenté sur la figure 2, on peut remarquer que le raffinement joue un rôle essentiel sur la rapidité de la convergence, le système étant plus grand, il met plus d'itérations pour converger. De plus la symétrie de la matrice (pas de vélocité) joue un rôle important sur la convergence de la norme du résidu puisque le nombre de conditionnement se voit plus petit que lorsque la matrice n'est pas symétrique, la convergence dépendant directement du nombre de conditionnement.

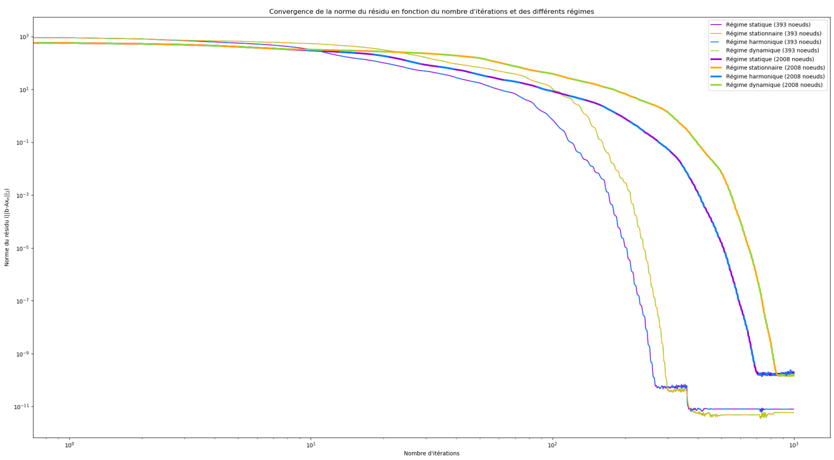

Figure 2: Convergence de GMRES en fonction de la taille du système et du régime

Itérations : n	1	10	25	50	100	250
$\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_n\|_2$	1035.66	373	71.18	18.31	7.72×10^{-1}	2.25×10^{-9}
$\|\boldsymbol{M}^{-1}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_n)\|_2$	2.13×10^{-3}	6.32×10^{-5}	5.10×10^{-8}	8.07×10^{-18}	1.09×10^{-17}	1.35×10^{-17}

Figure 3: Convergence avec et sans préconditionneur

4 Code

```python
def csrGMRES(sA,iA,jA,b,rtol, prec=False, x0=None, res_history=[], max_iterations=1e10):
    n = len(b)
    H = np.zeros_like(sA, shape=(n+1,n)) # Hessenberg matrix
    Q = np.zeros_like(sA, shape=(n,n))   # Espace de Krylov
    e1beta = np.zeros_like(sA, shape=(n+1,)) # (beta, 0, ..., 0)
    if x0 is not None: # [1.] Definir le premier residu
        r0 = np.array(b - csr_dot(sA, iA, jA, x0))
    else:
        x0 = np.zeros(n)
        r0 = np.array(b)
    if prec: # Preconditionnement
        sM, iM, jM = csrILU0(sA, iA, jA) # ILU factorisation
        r0 = csr_tridot(sM, iM, jM, r0)
    beta = np.linalg.norm(r0, ord=2) # [2.] Definir la norme du residu
    e1beta[0] = beta
    Q[:,0] = np.divide(r0, beta) # [3.] Premier V0 = r0 / beta
    for j in range(n):
        w = csr_dot(sA, iA, jA, Q[:,j]) # [4.] Definition de w = A@Vj
        if prec : w = csr_tridot(sM, iM, jM, w)
        for i in range(j+1):
            H[i,j] = np.dot(Q[:,i], w) # [5.] H[i,j] = Vi @ w
            w = w - np.dot(H[i,j],Q[:,i]) # [6.] w -= H[i,j] @ Vi
        H[j+1,j] = np.linalg.norm(w, ord=2) # [7.] H[j+1,j] = ||w||2
        if H[j+1, j] != 0 and j < n-1:
            Q[:, j+1] = np.divide(w, H[j+1, j]) # [8.]
        y = QRSolve(H, e1beta) # [9.] Resous aux moindres carrees
        x = x0 + np.dot(Q, y) # Solution plus precise (x_n)
        valres = b - csr_dot(sA, iA, jA, x) # Valeur du residu
        if prec: valres = csr_tridot(sM, iM, jM, valres)
        res_history.append(np.linalg.norm(valres, ord=2)) # Append la norme du nouveau residu
        if res_history[-1] < rtol or len(res_history)==max_iterations: # Condition d'arret atteinte
            return x, np.asarray(res_history)
    return csrGMRES(sA, iA, jA, b, rtol=rtol, prec=prec, x0=x, res_history=res_history, max_iterations=max_iterations)
```

```python
def csrILU0(sA,iA,jA):
    sM = np.array(sA)
    n = len(iA) - 1
    def get(i,j):
        return sM[iA[i]:iA[i+1]][jA[iA[i]:iA[i+1]]==j]
    for i in range(1,n):
        icol     = jA[iA[i]: iA[i+1]]
        i_line   = sM[iA[i]: iA[i+1]]
        k_ranger = icol[icol<i]
        for k in k_ranger:
            kcol   = jA[iA[k]:iA[k+1]]
            k_line = sM[iA[k]:iA[k+1]]
            kpivot = get(k,k)
            if len(kpivot) == 0:
                print('ATTENTION : Division par zero pour pivot (%d,%d)'%(k,k), file=sys.stderr)
            else:
                sM[iA[i]:iA[i+1]][icol==k] /= kpivot
            j_line = icol[icol>k]
            Aik = get(i,k)
            for j in j_line:
                Akj = get(k,j)
                Aij = get(i,j)
                if len(Aik) != 0 and len(Akj) != 0 and len(Aij) != 0:
                    sM[iA[i]:iA[i+1]][icol==j] -= Aik*Akj
    return sM, iA, jA
```