Devoir 3 : Algorithme QR

LINMA1170

Prof. J.-F. Remacle

Dans ce devoir, vous allez découvrir un des algorithmes les plus importants de l'algèbre linéaire. Il est même repris comme l'un des **10 plus grand algorithme du 20e siècle** : il s'agit de l'algorithme QR pour calculer des valeurs propres. Les valeurs propres et vecteurs propres sont omniprésents dans l'ingénierie :

- En mécanique, les valeurs propres peuvent indiquer les fréquences naturelles de vibration d'une structure.
- En data science, la PCA (*Principal component analysis*) n'autre autre qu'une décomposition d'une matrice A en valeurs singulières, elles-mêmes valeurs propres de A^TA.
- Dans le calcul stochastique et les chaînes de Markov, les vecteurs propres peuvent désigner l'état d'équilibre d'un système.
- En dynamique des systèmes, les valeurs propres indiquent si ledit système est stable $\mathrm{Re}(\lambda_i) < 0 \; \forall i$, ou instable $\exists i$ t.q. $\mathrm{Re}(\lambda_i) > 0$.
- ...

Mais comment les calcule-t-on ? La résolution de ce problème est autrement plus compliqué que celle d'un système linéaire puisque ils sont de natures fondamentalement différentes.

- La résolution d'un système ne fait appel qu'à des opérations de base ($+$, $-$, $\times$,$\div$) et sa solution possède donc une expression explicte qu'on peut obtenir en un nombre fini d'opérations.
- Le calcul des valeur propre d'une matrice 2×2 nécessite déjà l'opération $\sqrt{}$. Lorsque l'on atteint des matrices 5×5, il n'existe plus de solution explicite que l'on pourrait calculer en un nombre fini d'opérations. On se retrouve donc réduits à devoir approximer les valeurs propres de la matrice avec un algorithme itératif.

L'approche naïve consisterait à calculer les racines du polynôme caractéristique $\det(A - \lambda I)$. Cette idée est impossible en pratique puisque les coefficients du polynôme ne peuvent pas être calculés de façon numériquement stable, et que le calcul des racines n'est pas stable non plus. Il existe toute une famille d'algorithmes qui surmontent ces obstacles. En pratique, plusieurs facteurs influenceront le choix de l'algorithme utilisé :

- A est réelle ou complexe ?
- A est symétrique, anti-symétrique, hermitienne, anti-hermitienne ?
- A est une matrice creuse ou bande ?
- veut-on juste les valeurs propres, ou bien également les vecteurs propres ?
- veut-on toutes les valeurs propres ? les plus grandes/petites en valeur absolue ? celle qui ont la plus grande/petite partie réelle ? ...

Algorithme QR en théorie... et en pratique

Dans notre cas, on va investiguer le cas le plus général d'une matrice $A \in \mathbb{C}^{n\times n}$ dont on souhaite calculer le spectre complet. Pour celà, l'algorithme QR est le plus indiqué. Cet algorithme construit une décomposition de Schur T de la matrice A, où T est une matrice triangulaire supérieure dont les entrées diagonales sont les valeurs propres

$$A = UTU^*$$

En théorie, l'algorithme tient en trois lignes

```
for k = 1, 2, ...
    Q R <- A
    A   <- R Q
```

Sous certaines conditions, l'algorithme converge et `A` contient la matrice T. Cependant, cet algorithme n'est jamais implémenté de cette manière en pratique. Il est plutôt implémenté en deux étapes : une transformation sous forme hesseberg suivie d'une séquence de transformations QR.

Transformation sous forme hesseberg

Tout d'abord, A est transformée sous sa forme de Hessenberg H, une matrice *presque* triangulaire supérieure. On vous demande d'écrire une fonction Python `hessenberg(A, P)` où

- `A` est un `numpy.ndarray` de type `complex`, de taille `n x n` qui contient la matrice A
- `P` est un `numpy.ndarray` de type `complex`, de taille `n x n` non-initialisé
- En sortie, le tableau A a été réécrit par H, et P contient la transformation unitaire.

Transformation QR

Ensuite, une transformation unitaire Q est appliquée à gauche et à droite. On vous demande ici d'écrire une fonction Python `step_qr(H, U, m)` où

- `H` est un `numpy.ndarray` de type `complex`, de taille `n x n` qui contient une matrice sous forme Hessenberg
- `U` est un `numpy.ndarray` de type `complex`, de taille `n x n` qui contient la matrice de transformation unitaire — $H = U^*AU$
- `m` est la dimension de la matrice *active*
- En sortie, le tableau `H` a été réécrit par RQ où le produit QR est une décomposition qr de H. `U` est également mis à jour.

Pour cette fonction, vous aurez probablement besoin de calculer la rotation de givens d'un vecteur $(a, b) \in \mathbb{C}^2$.

Transformation QR *with shifts*

Dans la majorité des cas, il est en possible d'itérer uniquement avec la fonction `step_qr(H, Q)` jusqu'à convergence à la forme de Schur. Néanmoins, il existe une manière simple d'accélérer fortement la convergence de l'algorithme en introduisant un *shift* σ; on utilisera ici le *Wilkinson shift*. On calcule alors la décomposition QR de $H - \sigma I$, au lieu de H.

On vous demande d'écrire une fonction Python `m_new = step_qr_shift(H, Q, m)` où

- `H`, `Q` et `m` sont définis comme ci-dessus
- `m_new` est la dimension active après avoir effectué l'itération
- En sortie, les tableaux `H` et `Q` sont mis à jours comme ci-dessus

Algorithme QR

Pour synthétiser, on vous demande également une fonction Python `U, k = solve_qr(A, use_shifts, eps, max_iter)` où

- `A` est un `numpy.ndarray` de type `complex`, de taille `n x n` qui contient la matrice A en entrée, et la matrice T en sortie
- `U` est un `numpy.ndarray` de type `complex`, de taille `n x n` qui contient la transformation unitaire U telle que $A = UTU^*$
- `use_shifts` est un booléen qui indique si on souhaite utiliser des *shifts*
- `eps` est un `float` qui détermine le critère d'arrêt : on considère que l'algorithme a "convergé" lorsque les entrées sous-diagonales de `A` sont inférieures en norme à `eps`
- `k` est un `int` qui indique le nombre d'itérations qr qui ont été nécessaires, ou bien `-1` si on a atteint `max_iter` itérations

Visualisation

Un programme de visualisation `app_qr.py` de votre algorithme vous a été gentiment fourni par les assistants. Il contient une classe `WindowQR` qui affiche en temps réel les entrées de la matrice ainsi que ses entrées diagonales dans le plan complexe.

Cette partie est **facultative** et requiert l'installation des packages `PyQt5` et `pyqtgraph`. Vous pouvez compléter cette classe avec vos 3 premières fonctions décrites ci-dessus. Malheureusement, il semble difficile d'utiliser Numba avec des classes. On vous invite donc à dupliquer vos fonctions en y apportant de légères modifications (c'est moche, on sait…):

- retirez le décorateur Numba
- ajoutez l'argument `self`
- remplacez vos `A[i,j] = ...` par `self.set_matrix(A, i, j, ...)`

Une fois le programme exécuté, un menu d'aide est disponible avec la touche `h`. Après quelques itérations, vous devriez observer ce genre de résultat

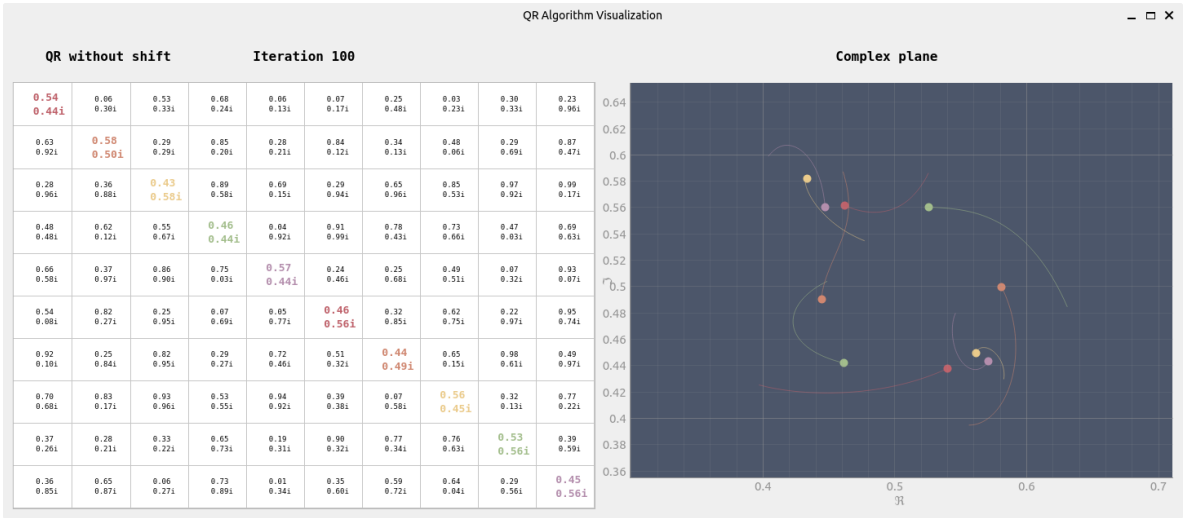

Figure 1: Interface graphique avec une solution incorrecte

Ce qui vous est demandé

1. Lire le document *Chapter 4: The QR Algorithm* (sections 4.1 à 4.4) où les algorithmes sont décrits en détail.
2. Répondre aux questions suivantes
   - Démontrez que les matrices normales ($A^*A = AA^*$) sont *exactement* les matrices diagonalisables par transformations unitaires. Pensez à la décomposition de Schur.
   - Lorsque la matrice est Hermitienne ou symétrique, quelles modifications pouvons-nous apporter à l'algorithme QR ?
   - Montrez la convergence linéaire et quadratique de l'algorithme sans/avec shift. Si

     $$H_k = \begin{bmatrix} a & b \\ \epsilon & d \end{bmatrix}$$

     montrez donc que l'élément sous-diagonal de H_{k+1} est en $\mathcal{O}(\epsilon)$ sans shift et en $\mathcal{O}(\epsilon^2)$ avec le Rayleigh-quotient shift. Vous pouvez vous limiter au cas réel.
   - (Bonus) Avec l'algorithme sans shift, les entrées diagonales ne convergent pas toujours vers un point fixe dans le plan complexe mais peuvent se retrouver coincées sur une courbe périodique dans le plan complexe. Quand cela arrive-t-il ? À quoi la période est liée ? Illustrez.
3. Écrire un module `devoir3.py` qui contient (au moins) les quatre fonctions `hessenberg(A, Q)`, `step_qr(H, Q)`, `step_qr_shift(H, Q)` et `solve_qr(A, use_shifts)`. Toute fonction de la librairie `numpy.linalg` est strictement interdite. Nous vous encourageons à utiliser `Numba` pour obtenir de meilleures performances. Les fonctions numpy de type `@` ou `dot` sont donc relativement découragées. Votre algorithme sera testé sur des matrices de type `complex`. Ces matrices ne seront pas nécessairement aléatoires : pensez aux cas limites !
4. Analysez la convergence de votre algorithme QR, avec ou sans shifts. Comparez la avec celle de `scipy`. Commentez.

Soumission Gradescope

Vous devez soumettre

- votre module python `devoir3.py`
- votre rapport en `.pdf`, ainsi que le `.tex` (maximum 4 pages A4)
- les scripts nécéssaires à la reproduction de vos figures

sur Gradescope pour le dimanche ~~14/04/2024 à 18h~~ **07/04/2024 à 22h**.

Le travail demandé est un travail individuel. Vos implémentations de l'algorithme et vos rapports seront soumis à un logiciel anti-plagiat.