Vous : Pouvez vous générer l'Analyse MERISE correspondant à ce fichier ? svp

Gemini a dit

Bonjour !

Pour être tout à fait transparent avec vous, la **méthode MERISE** est traditionnellement conçue pour la modélisation de Systèmes d'Information (bases de données persistantes, flux de gestion complexes, etc.). Votre fichier valeurs_propres.py étant un algorithme purement mathématique qui s'exécute en mémoire sans base de données, une analyse MERISE stricte (avec tables, clés primaires et étrangères) n'est pas son cas d'usage naturel.

Cependant, il est tout à fait possible d'**adapter les concepts de MERISE** (modélisation des données et des traitements) pour cartographier la logique et l'architecture de votre script. Voici l'analyse conceptuelle correspondante.

---

1. Dictionnaire des Données

Le dictionnaire recense les variables clés manipulées en mémoire par le programme.

- **n** : Entier (Dimension de la matrice carrée).
- **A** : Tableau 2D (Coefficients de la matrice, réels ou complexes).
- **w (eigenvalues)** : Tableau 1D (Valeurs propres calculées).
- **V (eigenvectors)** : Tableau 2D (Vecteurs propres calculés).
- **err (residual_rel)** : Réel (Résidu relatif pour évaluer la précision).
- **CV (cond_vectors)** : Réel (Conditionnement de la matrice des vecteurs propres).
- **status** : Chaîne de caractères (Indicateur de fiabilité : "ok" ou "warning").
- **schur_T / schur_Z** : Tableaux 2D (Matrices issues de la décomposition de Schur, si instabilité).

---

2. Modèle Conceptuel des Données (MCD) - *Adapté*

Si nous devions représenter les objets manipulés par votre code sous forme d'entités conceptuelles :

[MATRICE_SAISIE]

- Dimension (n)
- Coefficients (A)

[RESULTAT_ANALYSE]

- Valeurs_Propres
- Vecteurs_Propres
- Residu_Relatif
- Conditionnement
- Statut_Fiabilite
- Message_Diagnostic

[DECOMPOSITION_SCHUR] *(Entité conditionnelle)*

- Matrice_Triangulaire (T)
- Matrice_Unitaire (Z)

Relations :

- Une **[MATRICE_SAISIE]** *génère* (1,1) un **[RESULTAT_ANALYSE]**.
- Un **[RESULTAT_ANALYSE]** *peut contenir* (0,1) une **[DECOMPOSITION_SCHUR]** (uniquement déclenchée si le statut est "warning").

---

3. Modèle Conceptuel des Traitements (MCT)

Le MCT décrit la chronologie des opérations algorithmiques de votre script.

Événement déclencheur : Lancement du script main() par l'utilisateur.

Opération 1 : Saisie et Initialisation (read_matrix_from_user**)**

- **Actions :** * Demander la dimension *n*.
    - Boucler sur la saisie des lignes pour récupérer les coefficients.
    - Vérifier si la matrice est purement réelle pour optimiser le typage.
- **Résultat produit :** Matrice *A* validée et formatée.

Opération 2 : Calcul Mathématique et Diagnostic (robust_eigen**)**

- **Actions :**
    - Calcul des valeurs et vecteurs propres via les bibliothèques scipy ou numpy.
    - Calcul du résidu relatif (residual_rel) et du conditionnement.
- **Règles de gestion (Synchronisation) :**
    - *Si* résidu < 1e-8 *ET* conditionnement < 1e12 : Le statut est défini sur "ok".
    - *Sinon* : Le statut est défini sur "warning" et le programme calcule la décomposition de Schur (si SciPy est disponible).
- **Résultat produit :** Dictionnaire de données contenant les métriques et le diagnostic.

Opération 3 : Restitution et Visualisation (main **et** plot_eigenvectors**)**

- **Actions :**
    - Affichage dans la console de la matrice, du statut, des valeurs/vecteurs propres et des métriques de conditionnement.
- **Règles de gestion (Tracé graphique) :**
    - *Si* statut = "ok" *ET* dimension $\in\{2,3\}$ *ET* valeurs purement réelles : Tracé du graphique 2D/3D via matplotlib.
    - *Sinon* : Affichage d'un message d'impossibilité de tracé.
- **Résultat produit :** Affichage final (Console + Fenêtre graphique éventuelle). Fin du processus.

Souhaitez-vous que je vous aide à générer un diagramme de flux (Flowchart) au format Mermaid.js pour visualiser graphiquement cet algorithme, ou avez-vous besoin d'aide pour optimiser une fonction spécifique de ce code ?

Vous avez dit

Souhaitez-vous que je vous aide à générer un diagramme de flux (Flowchart) au format Mermaid.js pour visualiser graphiquement cet algorithme

Gemini a dit

Voici le diagramme de flux (Flowchart) représentant la logique complète de votre algorithme valeurs_propres.py.

Ce schéma retrace le cheminement des données, depuis la saisie utilisateur jusqu'à l'affichage du graphique, en passant par les contrôles de stabilité mathématique.

Extrait de code

```
graph TD
    Start([Début du programme]) --> SaisieN[Saisie de la dimension n]
    SaisieN --> CheckN{n > 0 ?}
    CheckN -- Non --> SaisieN
    CheckN -- Oui --> SaisieLignes[Saisie des coefficients par ligne]

    SaisieLignes --> CheckReel{Matrice quasi-réelle ?}
    CheckReel -- Oui --> CastFloat[Optimisation : Conversion en type float]
    CheckReel -- Non --> KeepComplex[Conservation en type complex]

    CastFloat --> CalcEig[Calcul valeurs w et vecteurs propres V]
    KeepComplex --> CalcEig
```

CalcEig --> CalcMetrics[Calcul du résidu relatif et du conditionnement]

CalcMetrics --> CheckStab{Résidu < 1e-8 ET
Conditionnement < 1e12 ?}

CheckStab -- Oui --> StatutOK[Statut : OK]

CheckStab -- Non --> StatutWarn[Statut : WARNING]

StatutWarn --> CheckScipy{SciPy disponible ?}

CheckScipy -- Oui --> CalcSchur[Calcul décomposition de Schur T, Z]

CheckScipy -- Non --> WarnScipy[Message : SciPy manquant pour Schur]

StatutOK --> Affichage[Affichage Console : Matrice, w, V, Métriques]

CalcSchur --> Affichage

WarnScipy --> Affichage

Affichage --> CheckPlot1{Statut == OK ?}

CheckPlot1 -- Non --> Fin([Fin du programme])

CheckPlot1 -- Oui --> CheckPlot2{Valeurs purement réelles
ET n ∈ {2, 3} ?}

CheckPlot2 -- Non --> PasDeTrace[Affichage : Tracé ignoré/impossible] --> Fin

CheckPlot2 -- Oui --> TraceGraph[Génération du Graphique 2D / 3D via Matplotlib] --> Fin

Comment utiliser ce diagramme ?

- **Si vous utilisez un éditeur Markdown (comme Obsidian, Notion ou VS Code) :** Ce code s'affichera automatiquement sous forme de graphique s'il prend en charge mermaid.
- **Si vous souhaitez le visualiser dans votre navigateur :** Vous pouvez copier-coller le bloc de code ci-dessus dans l'outil officiel en ligne **Mermaid Live Editor**.

Aimeriez-vous que nous ajoutions des commentaires détaillés directement dans votre fichier Python, ou souhaitez-vous explorer comment optimiser le calcul du conditionnement avec NumPy ?

Vous avez dit

Error: On line 30

MermaidDetailedError: Parse error on line 31:

...t réelles
ET n ∈ {2, 3} ?} CheckP

-----------------------^

Expecting 'SQE', 'DOUBLECIRCLEEND', 'PE', '-)', 'STADIUMEND', 'SUBROUTINEEND', 'PIPE', 'CYLINDEREND', 'DIAMOND_STOP', 'TAGEND', 'TRAPEND', 'INVTRAPEND', 'UNICODE_TEXT', 'TEXT', 'TAGSTART', got 'DIAMOND_START'

Gemini a dit

Ah, c'est de ma faute ! L'erreur vient du fait que Mermaid utilise les accolades {} pour dessiner les losanges de décision. En utilisant le symbole mathématique {2, 3} à l'intérieur du texte, l'analyseur de syntaxe s'est emmêlé les pinceaux et a cru que la forme se terminait prématurément.

Voici le code corrigé, où j'ai remplacé n ∈ {2, 3} par n = 2 ou 3 pour éviter ce conflit de syntaxe :

Extrait de code

```
graph TD
    Start([Début du programme]) --> SaisieN[Saisie de la dimension n]
    SaisieN --> CheckN{n > 0 ?}
    CheckN -- Non --> SaisieN
```

CheckN -- Oui --> SaisieLignes[Saisie des coefficients par ligne]

SaisieLignes --> CheckReel{Matrice quasi-réelle ?}

CheckReel -- Oui --> CastFloat[Optimisation : Conversion en type float]

CheckReel -- Non --> KeepComplex[Conservation en type complex]

CastFloat --> CalcEig[Calcul valeurs w et vecteurs propres V]

KeepComplex --> CalcEig

CalcEig --> CalcMetrics[Calcul du résidu relatif et du conditionnement]

CalcMetrics --> CheckStab{Résidu < 1e-8 ET
Conditionnement < 1e12 ?}

CheckStab -- Oui --> StatutOK[Statut : OK]

CheckStab -- Non --> StatutWarn[Statut : WARNING]

StatutWarn --> CheckScipy{SciPy disponible ?}

CheckScipy -- Oui --> CalcSchur[Calcul décomposition de Schur T, Z]

CheckScipy -- Non --> WarnScipy[Message : SciPy manquant pour Schur]

StatutOK --> Affichage[Affichage Console : Matrice, w, V, Métriques]

CalcSchur --> Affichage

```
    WarnScipy --> Affichage


    Affichage --> CheckPlot1{Statut == OK ?}
    CheckPlot1 -- Non --> Fin([Fin du programme])


    CheckPlot1 -- Oui --> CheckPlot2{Valeurs purement réelles<br>ET n = 2 ou 3 ?}
    CheckPlot2 -- Non --> PasDeTrace[Affichage : Tracé ignoré/impossible]
    PasDeTrace --> Fin
    CheckPlot2 -- Oui --> TraceGraph[Génération du Graphique 2D / 3D via Matplotlib]
    TraceGraph --> Fin
```

Ce code devrait s'afficher parfaitement dans votre éditeur Markdown !

Souhaitez-vous que l'on passe à l'ajout de commentaires détaillés dans votre fichier Python pour expliquer la logique métier, ou que l'on analyse comment optimiser les calculs de conditionnement ?

Vous avez dit

Oui je voudrais que vous passiez à l'ajout de commentaires détaillés dans mon fichier Python pour expliquer la logique métier. Voici le code que j'utilise dans mon fichier python : import numpy as np

```
import matplotlib.pyplot as plt


def read_matrix_from_user() -> np.ndarray:
    """Saisie d'une matrice carrée (réelle ou complexe) via console."""
    print("=== Saisie d'une matrice ===")
    while True:
```

```
    try:
        n = int(input("Entrez la dimension n (matrice n x n) : ").strip())
        if n <= 0:
            print(" -> Erreur : n doit être un entier strictement positif.")
            continue
        break
    except ValueError:
        print(" -> Erreur : Veuillez entrer un nombre entier valide.")

print("\nEntrez les coefficients ligne par ligne (séparés par des espaces).")
print("Réels : 3 -1.2 0 4.5")
print("Complexes : 1+2j 3 -0.5j 2\n")

A = np.zeros((n, n), dtype=complex)

for i in range(n):
    while True:
        parts = input(f"Ligne {i+1} : ").strip().split()
        if len(parts) != n:
            print(f" -> Il faut exactement {n} valeurs.")
            continue
```

```
        try:
            A[i, :] = [complex(x) for x in parts]
            break
        except ValueError:
            print(" -> Valeur invalide. Exemple : 2, -1.5, 1+2j")

    # Si la matrice est (quasi) réelle, convertir en float pour optimiser
    if np.allclose(A.imag, 0.0, atol=1e-14):
        return A.real.astype(float)
    return A


def residual_rel(A: np.ndarray, w: np.ndarray, V: np.ndarray) -> float:
    """
    Calcule le résidu relatif standard :
      ||A V - V diag(w)||_F / (||A||_F * ||V||_F)
    """
    A = np.asarray(A)
    w = np.asarray(w)
    V = np.asarray(V)
```

```python
    R = A @ V - V @ np.diag(w)
    num = np.linalg.norm(R, ord="fro")
    den = (np.linalg.norm(A, ord="fro") * np.linalg.norm(V, ord="fro")) + 1e-30
    return float(num / den)


def robust_eigen(A: np.ndarray, quality_tol: float = 1e-8, cond_tol: float = 1e12) -> dict:
    """
    Calcule valeurs/vecteurs propres. Si instable, fournit Schur (si SciPy dispo).
    """
    A = np.asarray(A)
    if A.ndim != 2 or A.shape[0] != A.shape[1]:
        raise ValueError("La matrice doit être carrée.")

    out = {"n": int(A.shape[0])}

    # 1) eig (SciPy si dispo, sinon NumPy)
    scipy_available = False
    try:
        from scipy.linalg import eig as scipy_eig, schur
        w, V = scipy_eig(A)
```

```
    scipy_available = True
except Exception:
    w, V = np.linalg.eig(A)

out["eigenvalues"] = w
out["eigenvectors"] = V

# 2) Qualité : résidu + conditionnement
err = residual_rel(A, w, V)
out["residual_rel"] = float(err)

try:
    CV = np.linalg.cond(V)
except np.linalg.LinAlgError:
    CV = np.inf
out["cond_vectors"] = float(CV)

# 3) Décision
if err < quality_tol and CV < cond_tol:
    out["status"] = "ok"
    out["message"] = "Vecteurs propres calculés et jugés fiables."
```

```
        return out

    out["status"] = "warning"
    out["message"] = (
        "Valeurs propres OK. Vecteurs propres potentiellement instables "
        "(matrice défectueuse ou proche). "
    )

    # 4) Schur si possible
    if scipy_available:
        T, Z = schur(A, output="complex")   # A = Z T Z*
        out["schur_T"] = T
        out["schur_Z"] = Z
        out["message"] += "Utilisez la décomposition de Schur (T, Z) fournie."
    else:
        out["message"] += "Installez SciPy pour obtenir la décomposition de Schur."

    return out


def plot_eigenvectors(w: np.ndarray, V: np.ndarray):
```

```
"""
Trace les vecteurs propres en 2D ou 3D s'ils sont purement réels.
"""
n = V.shape[0]

# 1. Vérification : On ne trace que du réel
if np.iscomplexobj(w) or np.iscomplexobj(V):
    # On tolère les petits résidus imaginaires (ex: 1e-16j) dus aux erreurs d'arrondi
    if np.allclose(w.imag, 0) and np.allclose(V.imag, 0):
        w = w.real
        V = V.real
    else:
        print("\n[Graphique] Impossible de tracer : les valeurs/vecteurs propres sont complexes.")
        return

# 2. Vérification : On ne trace que pour des dimensions 2x2 ou 3x3
if n not in [2, 3]:
    print(f"\n[Graphique] Tracé non pris en charge pour la dimension {n} (uniquement 2D ou 3D).")
    return

fig = plt.figure(figsize=(8, 8))
```

```
# --- TRACÉ 2D ---
if n == 2:
    ax = fig.add_subplot(111)
    ax.axhline(0, color='grey', lw=1, zorder=0)
    ax.axvline(0, color='grey', lw=1, zorder=0)

    colors = ['red', 'blue']
    for i in range(2):
        v = V[:, i]
        val = w[i]
        ax.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1,
                  color=colors[i], width=0.015, label=f'v{i+1} (λ={val:.2f})')

    ax.set_xlim(-1.5, 1.5)
    ax.set_ylim(-1.5, 1.5)
    ax.set_aspect('equal')
    ax.set_title("Vecteurs propres en 2D (Normalisés)")
    ax.grid(True, linestyle='--')
    ax.legend()
```

```
# --- TRACÉ 3D ---
elif n == 3:
    ax = fig.add_subplot(111, projection='3d')
    ax.plot([-1.5, 1.5], [0, 0], [0, 0], color='grey', lw=1)
    ax.plot([0, 0], [-1.5, 1.5], [0, 0], color='grey', lw=1)
    ax.plot([0, 0], [0, 0], [-1.5, 1.5], color='grey', lw=1)

    colors = ['red', 'blue', 'green']
    for i in range(3):
        v = V[:, i]
        val = w[i]
        ax.quiver(0, 0, 0, v[0], v[1], v[2], color=colors[i],
                  length=1.0, normalize=False, arrow_length_ratio=0.1,
                  label=f'v{i+1} (λ={val:.2f})')

    ax.set_xlim([-1.5, 1.5])
    ax.set_ylim([-1.5, 1.5])
    ax.set_zlim([-1.5, 1.5])
    ax.set_title("Vecteurs propres en 3D (Normalisés)")
    ax.legend()
```

```
    plt.show()


def main():
    A = read_matrix_from_user()
    print("\n=== Matrice A ===")
    with np.printoptions(precision=4, suppress=True):
        print(A)

    result = robust_eigen(A)

    print("\n=== Résultats ===")
    print(f"Status  : {result['status'].upper()}")
    print(f"Message : {result['message']}")

    with np.printoptions(precision=4, suppress=True):
        print("\nValeurs propres :")
        print(result["eigenvalues"])

        if "eigenvectors" in result:
            print("\nVecteurs propres (en colonnes) :")
```

```python
        print(result["eigenvectors"])

      print(f"\nErreur relative : {result.get('residual_rel', 'N/A'):.2e}")
      print(f"Conditionnement cond(V) : {result.get('cond_vectors', 'N/A'):.2e}")

      if "schur_T" in result:
        print("\n--- (Info robuste) Décomposition de Schur A = Z T Z* ---")
        print("Matrice triangulaire supérieure T :")
        print(result["schur_T"])

    # Appel du tracé graphique si les vecteurs sont fiables et réels
    if "eigenvectors" in result and result["status"] == "ok":
      plot_eigenvectors(result["eigenvalues"], result["eigenvectors"])

if __name__ == "__main__":
    main()
```

Gemini a dit

Voici votre code Python enrichi avec des commentaires détaillés. J'ai pris soin d'expliquer non seulement *ce que* fait le code, mais surtout *pourquoi* il le fait (la "logique métier" et mathématique derrière chaque étape).

Python

```python
import numpy as np
```

```
import matplotlib.pyplot as plt


def read_matrix_from_user() -> np.ndarray:
    """
    Saisie d'une matrice carrée (réelle ou complexe) via console.
    Logique métier : L'utilisateur peut faire des erreurs de frappe. On utilise
    des boucles 'while True' pour bloquer le programme tant que l'entrée
    n'est pas mathématiquement valide (dimension cohérente, type correct).
    """
    print("=== Saisie d'une matrice ===")


    # Étape 1 : Définir la dimension de l'espace (matrice carrée n x n)
    while True:
        try:
            n = int(input("Entrez la dimension n (matrice n x n) : ").strip())
            if n <= 0:
                print(" -> Erreur : n doit être un entier strictement positif.")
                continue
            break # On sort de la boucle si l'entier est valide
        except ValueError:
            print(" -> Erreur : Veuillez entrer un nombre entier valide.")
```

```
print("\nEntrez les coefficients ligne par ligne (séparés par des espaces).")
print("Réels : 3 -1.2 0 4.5")
print("Complexes : 1+2j 3 -0.5j 2\n")

# On initialise une matrice vide de nombres complexes par précaution,
# car on ne sait pas encore ce que l'utilisateur va saisir.
A = np.zeros((n, n), dtype=complex)

# Étape 2 : Remplissage ligne par ligne
for i in range(n):
    while True:
        parts = input(f"Ligne {i+1} : ").strip().split()

        # Vérification métier : Une ligne d'une matrice n x n doit avoir exactement n éléments
        if len(parts) != n:
            print(f" -> Il faut exactement {n} valeurs.")
            continue

        try:
            # On tente de convertir chaque élément de la ligne en type 'complex'
```

```
            A[i, :] = [complex(x) for x in parts]
            break
        except ValueError:
            print(" -> Valeur invalide. Exemple : 2, -1.5, 1+2j")

    # Étape 3 : Optimisation métier
    # Si la matrice ne contient que des réels (la partie imaginaire est proche de 0),
    # on la convertit en type 'float' classique. Cela accélère les calculs et
    # évite d'afficher des "+ 0j" inutiles à l'écran.
    if np.allclose(A.imag, 0.0, atol=1e-14):
        return A.real.astype(float)

    return A



def residual_rel(A: np.ndarray, w: np.ndarray, V: np.ndarray) -> float:
    """
    Calcule le résidu relatif standard : ||A V - V diag(w)||_F / (||A||_F * ||V||_F)
    Logique métier : C'est le "juge de paix". Mathématiquement, on doit avoir A*V = V*w.
    Si on soustrait les deux (A*V - V*w), on devrait obtenir 0.
    Ce calcul vérifie si le résultat s'éloigne de 0 à cause d'erreurs d'arrondi ou d'instabilités.
```

```
    """
    A = np.asarray(A)
    w = np.asarray(w)
    V = np.asarray(V)

    # R est la matrice des écarts (résidus). Si le calcul est parfait, R est remplie de zéros.
    R = A @ V - V @ np.diag(w)

    # On utilise la norme de Frobenius (ord="fro") qui est standard pour mesurer "la taille" d'une matrice
    num = np.linalg.norm(R, ord="fro")

    # On divise par la norme de A et V pour obtenir une erreur "relative" (en pourcentage global).
    # Le "+ 1e-30" est une astuce de sécurité pour éviter une division par zéro si A ou V sont nuls.
    den = (np.linalg.norm(A, ord="fro") * np.linalg.norm(V, ord="fro")) + 1e-30

    return float(num / den)



def robust_eigen(A: np.ndarray, quality_tol: float = 1e-8, cond_tol: float = 1e12) -> dict:
    """
    Calcule valeurs/vecteurs propres et évalue leur fiabilité.
```

```
"""
A = np.asarray(A)
if A.ndim != 2 or A.shape[0] != A.shape[1]:
    raise ValueError("La matrice doit être carrée.")


out = {"n": int(A.shape[0])}


# 1) Calcul principal : SciPy est préféré car ses algorithmes de bas niveau (LAPACK)
# sont souvent plus robustes que ceux de base de NumPy pour ce type de tâche.
scipy_available = False
try:
    from scipy.linalg import eig as scipy_eig, schur
    w, V = scipy_eig(A)
    scipy_available = True
except Exception:
    # Fallback (Plan B) si SciPy n'est pas installé sur la machine
    w, V = np.linalg.eig(A)


out["eigenvalues"] = w
out["eigenvectors"] = V
```

```
# 2) Contrôle Qualité (Logique de diagnostic)
err = residual_rel(A, w, V)
out["residual_rel"] = float(err)

# Le conditionnement (cond) indique la sensibilité aux erreurs.
# Un conditionnement trop élevé (> 1e12) signifie que les vecteurs propres sont
# presque parallèles (matrice "défectueuse"). Les résultats deviennent alors douteux.
try:
    CV = np.linalg.cond(V)
except np.linalg.LinAlgError:
    CV = np.inf # Si le calcul plante, c'est que la matrice est singulière (infinie)
out["cond_vectors"] = float(CV)

# 3) Arbre de décision métier
if err < quality_tol and CV < cond_tol:
    # Tout va bien, les résultats sont exploitables en ingénierie/maths.
    out["status"] = "ok"
    out["message"] = "Vecteurs propres calculés et jugés fiables."
    return out

# Si on arrive ici, les résultats normaux sont dangereux à utiliser.
```

```
    out["status"] = "warning"
    out["message"] = (
        "Valeurs propres OK. Vecteurs propres potentiellement instables "
        "(matrice défectueuse ou proche). "
    )

    # 4) Plan de secours : Décomposition de Schur
    # Au lieu d'échouer, on propose une alternative mathématiquement toujours stable,
    # même pour des matrices impossibles à diagonaliser proprement.
    if scipy_available:
        T, Z = schur(A, output="complex")   # La matrice T sera triangulaire, Z sera unitaire
        out["schur_T"] = T
        out["schur_Z"] = Z
        out["message"] += "Utilisez la décomposition de Schur (T, Z) fournie."
    else:
        out["message"] += "Installez SciPy pour obtenir la décomposition de Schur."

    return out


def plot_eigenvectors(w: np.ndarray, V: np.ndarray):
```

```
"""
Logique métier : La visualisation géométrique des vecteurs propres n'a de sens
physique (pour un humain) que dans l'espace 2D ou 3D réel.
"""
n = V.shape[0]

# 1. Nettoyage et vérification des données complexes
if np.iscomplexobj(w) or np.iscomplexobj(V):
    # L'ordinateur produit souvent des résidus imaginaires minuscules (ex: 2 + 1e-16j).
    # On tolère ces résidus en les ignorant s'ils sont infimes (allclose).
    if np.allclose(w.imag, 0) and np.allclose(V.imag, 0):
        w = w.real
        V = V.real
    else:
        # Si les valeurs sont vraiment complexes (ex: rotations), on annule le tracé.
        print("\n[Graphique] Impossible de tracer : les valeurs/vecteurs propres sont complexes.")
        return

# 2. On bloque les dimensions impossibles à représenter à l'écran (>3D) ou triviales (1D)
if n not in [2, 3]:
    print(f"\n[Graphique] Tracé non pris en charge pour la dimension {n} (uniquement 2D ou 3D).")
```

```
    return


fig = plt.figure(figsize=(8, 8))


# --- TRACÉ 2D ---
if n == 2:
    ax = fig.add_subplot(111)
    # Tracé des axes X et Y passant par zéro
    ax.axhline(0, color='grey', lw=1, zorder=0)
    ax.axvline(0, color='grey', lw=1, zorder=0)


    colors = ['red', 'blue']
    for i in range(2):
        v = V[:, i] # Extraction de la colonne i (qui est le vecteur propre)
        val = w[i]  # Valeur propre associée
        # Tracé du vecteur sous forme de flèche depuis l'origine (0,0)
        ax.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1,
                  color=colors[i], width=0.015, label=f'v{i+1} (λ={val:.2f})')


    ax.set_xlim(-1.5, 1.5)
    ax.set_ylim(-1.5, 1.5)
```

```
    ax.set_aspect('equal') # Pour ne pas déformer les flèches (un repère orthonormé)
    ax.set_title("Vecteurs propres en 2D (Normalisés)")
    ax.grid(True, linestyle='--')
    ax.legend()

# --- TRACÉ 3D ---
elif n == 3:
    ax = fig.add_subplot(111, projection='3d')
    # Tracé des axes X, Y, Z
    ax.plot([-1.5, 1.5], [0, 0], [0, 0], color='grey', lw=1)
    ax.plot([0, 0], [-1.5, 1.5], [0, 0], color='grey', lw=1)
    ax.plot([0, 0], [0, 0], [-1.5, 1.5], color='grey', lw=1)

    colors = ['red', 'blue', 'green']
    for i in range(3):
        v = V[:, i]
        val = w[i]
        # En 3D, quiver fonctionne différemment, on précise point de départ et coordonnées
        ax.quiver(0, 0, 0, v[0], v[1], v[2], color=colors[i],
                  length=1.0, normalize=False, arrow_length_ratio=0.1,
                  label=f'v{i+1} (λ={val:.2f})')
```

```
        ax.set_xlim([-1.5, 1.5])
        ax.set_ylim([-1.5, 1.5])
        ax.set_zlim([-1.5, 1.5])
        ax.set_title("Vecteurs propres en 3D (Normalisés)")
        ax.legend()

    plt.show()


def main():
    """
    Fonction chef d'orchestre : coordonne les entrées, les traitements et les sorties.
    """
    A = read_matrix_from_user()
    print("\n=== Matrice A ===")

    # 'suppress=True' empêche l'affichage en notation scientifique pour les petits nombres
    # 'precision=4' limite l'affichage à 4 décimales pour plus de lisibilité
    with np.printoptions(precision=4, suppress=True):
        print(A)
```

```python
# Exécution du cœur mathématique
result = robust_eigen(A)

print("\n=== Résultats ===")
print(f"Status  : {result['status'].upper()}")
print(f"Message : {result['message']}")

# Affichage formaté des résultats
with np.printoptions(precision=4, suppress=True):
    print("\nValeurs propres :")
    print(result["eigenvalues"])

    if "eigenvectors" in result:
        print("\nVecteurs propres (en colonnes) :")
        print(result["eigenvectors"])

    # get() est utilisé pour éviter une erreur si les clés n'existent pas dans le dict
    print(f"\nErreur relative : {result.get('residual_rel', 'N/A'):.2e}")
    print(f"Conditionnement cond(V) : {result.get('cond_vectors', 'N/A'):.2e}")
```

```
        # Si le diagnostic a activé la sécurité de Schur, on l'affiche
        if "schur_T" in result:
            print("\n--- (Info robuste) Décomposition de Schur A = Z T Z* ---")
            print("Matrice triangulaire supérieure T :")
            print(result["schur_T"])

    # Étape finale : On affiche le graphique uniquement si les mathématiques
    # derrière sont solides (status = 'ok').
    if "eigenvectors" in result and result["status"] == "ok":
        plot_eigenvectors(result["eigenvalues"], result["eigenvectors"])


# Point d'entrée standard en Python
if __name__ == "__main__":
    main()
```

J'ai détaillé les raisons pour lesquelles allclose, scipy plutôt que numpy, et le conditionnement sont utilisés.

Souhaitez-vous que nous passions en revue ensemble les bonnes pratiques pour écrire des tests unitaires (avec pytest) pour sécuriser ces fonctions mathématiques ?