2025 年度 科学技術計算

学生番号 36714143
情報工学科知能情報分野
和田小百合

2025 年 12 月 11 日

目次

1　課題 05-1

1.1　課題の内容

小規模な密行列のすべての固有値を同時に反復的に求める方法として QR 法があります。QR 法は行列 A を QR 分解し，QR 変換を繰り返し適用することで固有値を求める手法です。一般の n 次対称行列に対する QR 分解の計算量は $O(n^3)$ であるため、QR 法の反復において一般の QR 分解を行ってしまうと、非常に計算量が大きくなってしまいます。また、QR 法の収束は 1 次であるため、反復回数も多くなることがあります。これを改善するための工夫として、三重対角行列に対する QR 法の反復では三重対角は保たれる性質から、まずハウスホルダー変換で元の対称行列を三重対角行列に変換します。この計算量は $O(n^3)$ ですが，前処理として一度だけで済みます。QR 法の反復における QR 分解は、ギブンス変換を用いて、三重対角行列を上三角行列へ変換すます。この計算量は $O(n^2)$ で済みます。

それを実装した関数が授業資料の eigh_by_qr_method() です。この関数の QR 法の収束条件として「主対角成分 (diagonals) が変化しなくなるまで」と設定していたが、この条件では収束が不十分であり、非対角成分が完全には 0 に収束しません。そこで、QR 法の収束の精度を上げるために「優対角成分 (superdiagonals) も変化しなくなるまで」という条件を追加しているが、これにより QR 法の反復回数が増加してしまいます。なので、今回の課題では収束条件を主対角成分 (diagonals) のみとし、その時点で得られる近似固有値（主対角成分）と近似固有ベクトルを初期値としてレイリー商反復法を適用し、より精度の高い固有値と固有ベクトルを得る方法を実装し、計算量をはやくしました。

1.2　ハウスホルダー変換について

ハウスホルダー法は、鏡映行列とベクトルの積を使って、結果のベクトルが 1 つの要素を除いて 0 になるような鏡映行列（ハウスホルダー行列）を利用し、$n-1$ 回の変換で n 次対称行列を三重対角行列に変換する方法です。以下がハウスホルダー変換を行う関数です。

```
 #ハウスホルダー法
def householder_step( A: np.ndarray, k: int) -> Tuple[np.ndarray, np.ndarray]:
    assert A.ndim == 2  #次元であるかチェック2
    assert A.shape[0] == A.shape[1] #正方行列かどうかチェック

    n = A.shape[0]
    assert 0 <= k < n - 2

    x = A[k + 1:, k]  # の列目のAkk行目以降の部分を取得縦の部分+1()
    y = np.zeros_like(x) #全部ゼロの配列
    y[0] = - np.sign(x[0]) * norm(x)  # ベクトルの先頭要素を設定y

    u = (x - y) / norm(x - y)  # 単位ベクトルを計算u
    H = np.eye(len(x)) - 2 * np.outer(u, u)  # ハウスホルダー行列を計算H

    Q = np.eye(n)    #単位行列
    Q[k + 1:, k + 1:] = H  # にを組み込むQH

    QAQ = Q @ A @ Q  # 変換後の行列を計算
    return QAQ, Q

def householder_transform(A: np.ndarray,) -> Tuple[np.ndarray, np.ndarray]:
    assert A.ndim == 2
```

```
    assert A.shape[0] == A.shape[1]
    n = A.shape[0]

    Atri = A.copy()
    Qhh = np.eye(n)  #サイズの単位行列n

    for k in range(n - 2):
        Atri, Q = householder_step(Atri, k)
        Qhh = Qhh @ Q  # 直交行列を更新
    return Atri, Qhh
```

1.3 ギブンス変換について

ギブンス変換はギブンス回転（Givens rotation）を利用して、指定した 1 つの要素を 0 にするような変換を行います。1 回の変換で特定の要素を消去し、それを繰り返して他の要素も消していきます。これを繰り返すことによって、このギブンス変換では、三重対角対称行列を上三角行列へと変換します。目標とするのは、劣対角成分 $a_{i+1,i}$ をすべて 0 にすることです。なので、これらの要素をすべて 0 にし、かつ下三角成分を 0 に保つことで、ギブンス回転行列を用いて上三角行列 R を得ます、つまり QR 分解を実現でききます。
以下にギブンス変換を行う関数を示す。

```
  #ギブンス変換
def givens_r_matrix(xp_xq: np.ndarray) -> np.ndarray:
    assert len(xp_xq) == 2

    xp, xq = xp_xq[0], xp_xq[1]

    denominator = norm(xp_xq)
    c = xp / denominator
    s = -xq / denominator

    R = np.array([
        [c, s],
        [-s, c]
    ])
    return R

def qr_decom_by_givens(Atri_org: np.ndarray,) -> Tuple[np.ndarray, np.ndarray]:
    assert len(Atri_org.shape) == 2
    assert Atri_org.shape[0] == Atri_org.shape[1]
    n = Atri_org.shape[0]

    Atri = Atri_org.copy()
    Q = np.eye(n)

    for k in range(n - 1):
        p, q = k, k + 1
        Rot2x2 = givens_r_matrix(Atri[[p, q]][:, p])
        Atri[[p, q]] = Rot2x2.T @ Atri[[p, q]]
        Q[:, [p, q]] = Q[:, [p, q]] @ Rot2x2

    R = Atri
    return Q, R
```

1.4 固有値固有ベクトルの導出について

ハウスホルダー変換を行い行列 A を三重行列にして、QR 分解にギブンス変換を行うことによって近似的な固有値と固有ベクトルを導出することができた。この近似値を正確な固有値、固有ベクトルにするためにレイリー商反復を行い実質的な値に近づけていく。
以下がレイリー商反復を行う関数です。

```
  #レイリー商反復について
#関数
def rayleigh_quotient_iteration(A: np.ndarray,mu: float, u, tol = 1e-12,n = np.
    ndarray, maxiter: int = 5000,) -> Tuple[float, np.ndarray]:
    #入力 行列が対称かどうかA
    assert A.ndim == 2
    assert A.shape[0] == A.shape[1]
    n = A.shape[0]
    assert len(u) == n

    u_pre = u.copy()

    #レイリー商反復
    for i in range(maxiter):
        try:
            u = solve(A - mu * np.eye(n), u)     #シフト値を用いて連立方程式を解くmu
        except:
            break

        u /= norm(u)     #正規化
        mu = u.T @ A @ u

        if np.allclose(u, u_pre) or np.allclose(u, -u_pre):
        #の値がめっちゃちかくなった→近似u
            break
        u_pre = u.copy()
    return mu, u
```

レイリー商反復では $A - \mu I$ の連立方程式を解く部分があり、μ が A の固有値に近づくにつれて得意行列になりやすくなるため、その例外処理として u = solve(A - mu * np.eye(n), u) の部分で try catch を行っています。
また、これらの操作をすべて一つで行う関数を作成しました。以下がその関数です。また、QR 法関数も以下にしめす。

```
  #法関数QR
def eigh_by_qr_method(A0: np.ndarray, maxiter: int = 1000,) -> Tuple[np.ndarray
    , np.ndarray]:
    assert A0.ndim == 2
    assert A0.shape[0] == A0.shape[1]
    n = A0.shape[0]
    A = A0.copy()

    Qall = np.eye(n)
    diagonals_pre = diag(A)
    #ハウスホルダー変換初めの一回で計算量を削減()
```

```
    Atri, Qhh = householder_transform(A)
    Qall = Qhh
    Atri_org = Atri.copy()

    for i in range(maxiter):
        #ギブンス変換
        Q, R = qr_decom_by_givens(Atri_org)
        Atri_org = R @ Q
        Qall = Qall @ Q

        diagonals = diag(Atri_org)
        if (np.allclose(diagonals, diagonals_pre)):
            break

        diagonals_pre = diagonals.copy()

    return diagonals, Qall


#一括で固有値固有ベクトルを求める関数
def eigh_via_qr_rayleigh( A: np.ndarray,) -> Tuple[np.ndarray, np.ndarray]:
    assert A.ndim == 2
    assert A.shape[0] == A.shape[1]
    n = A.shape[0]

    #法QR
    eigenvalues_approx, Qqr  = eigh_by_qr_method(A)
    eigenvalues = np.zeros(n)
    eigenvectors = np.zeros((n, n))

    #レイリー商反復
    for k in range(n):
        mu0 = eigenvalues_approx[k]
        u0 = Qqr[k, :]

        mu_refined, u_refined = rayleigh_quotient_iteration(A, mu0, u0)

        eigenvalues[k] = mu_refined
        eigenvectors[:, k] = u_refined

    return eigenvalues, eigenvectors
```

このように QR 法を行う前にハウスホルダー変換を行うことによって
三重対角化:1 回だけ $\Rightarrow\ O(n^3)$
QR iteration: 一回あたり $\Rightarrow\ O(n^2)$(for 文で回す)
合計 $\Rightarrow\ O(n^3) + O(n^2)$
となり、授業資料よりも計算量を削減することができました。

1.5 実行結果

numpy と自作の固有値固有ベクトルの比較はこのように検証した。

```
  #main
```

```
#実行
for i in tqdm(range(10)):
    n = rng.integers(low = 2, high = 10)
    A = rng.random(size=(n, n))

    #対称行列
    A = A.T @ A
    print("行列 A:")
    print(A)

    d, U = eigh_via_qr_rayleigh(A)
    eigvals_np, eigvecs_np = np.linalg.eigh(A)


    with np.printoptions(formatter={'float': '{: 12.8f}'.format}, suppress=True
        , linewidth=100):
        d_np, U_np = np.linalg.eigh(A)
        U_np = U_np[:, ::-1]
        d_np = d_np[::-1]

        # print("d_np", d_np)
        # print("d   ", d)
        print("d == d_np", np.allclose(d, d_np))

        print("U_np\n", U_np, sep='')
        print("U\n", U, sep='')

        for i in range(len(U)):
            print(
                f"column {i}: U[{i}] == U_np[{i}] ?",
                np.allclose(U[:, i], U_np[:, i]) or np.allclose(U[:, i], -U_np
                    [:, i])
            )
```

結果の一部を記す。

```
行列 A:
[[1.80393177 1.09552884 0.87745122 1.2803063 ]
 [1.09552884 0.75242837 0.80159591 0.94802328]
 [0.87745122 0.80159591 1.25722598 1.13293044]
 [1.2803063  0.94802328 1.13293044 1.46912807]]
d == d_np True
column 0: U[0] == U_np[0] ? True
column 1: U[1] == U_np[1] ? True
column 2: U[2] == U_np[2] ? True
column 3: U[3] == U_np[3] ? True
```

このように numpy と値が同じであることがわかりました。

2 課題 05-3

2.1 課題の内容

固有値分解（対角化）を用いた thin SVD の単純な計算する関数と QR 分解を用いた full/reduced QR 分解を利用した thin SVD の計算をする関数を作成し、numpy 関数の np.linalg.svd と一致するか検証しましました。

2.2 QR 分解を利用する関数 full/reduced QR 分解を利用した thin SVD の計算方法

以下が作成した関数です。

```
 #分解を利用する関数QR full/reduced 分解を利用したQRthin の計算方法SVD
def svd_via_qr(A: np.ndarray,) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    #行列のサイズ計算
    m, n = A.shape
    print("m =",m)
    print("n =",n)

    if(m >= n):
        #A.T @ A対称行列()の対角化->分解QR固有値計算->
        ATA = A.T @ A
        sigma, V = np.linalg.eigh(ATA)
        sigma = np.sqrt(np.maximum(sigma, 0))

        V = V[:,::-1]     #と行列の順番が違うため変換numpy
        sigma = sigma[::-1]
        a = A @ V
        U, c = np.linalg.qr(a)

    elif(m < n):
        AAT = A @ A.T
        sigma , U =  np.linalg.eigh(AAT)
        sigma = np.sqrt(np.maximum(sigma, 0))
        U = U[:,::-1]     #関数と行列の順番が違うため変換numpy
        sigma = sigma[::-1]
        a = A.T @ U
        V ,c = np.linalg.qr(a)

    VT = V.T  #最後に転置

    return U, sigma, VT
```

基本的には授業資料に説明されていた通りの計算方法でプログラムを作成したが、numpy で実装されている行列と自分の関数で実装した初めの固有値分解の時に出てくる行列の順番が違っていたので順番を入れ替える操作をし、そのあと、QR 分解をし二つめの行列を作成しました。

2.3 固有値分解を利用した単純な thinSVD の計算方法

以下が作成した関数である。

```
 #固有値分解を利用した単純なの計算方法thinSVD
```

```python
def svd_via_eigh( A: np.ndarray, ) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    #行列のサイズ計算
    m, n = A.shape
    print("m =",m)
    print("n =",n)

    if(m >= n):
            #A.T @ A対称行列()の対角化->分解QR固有値計算->
        ATA = A.T @ A
        sigma, V = np.linalg.eigh(ATA)
        V = V[:,::-1]  #関数と行列の順番が違うため変換numpy
        sigma = np.sqrt(np.maximum(sigma, 0))
        sigma = sigma [::-1]
        Sigma_inv = np.diag(1.0 / sigma) # 各対角成分を逆数にするだけ
        U = A @ V @ Sigma_inv

    elif(m < n):
        AAT = A @ A.T
        sigma , U =  np.linalg.eigh(AAT)
        sigma = np.sqrt(np.maximum(sigma, 0))
        sigma = sigma [::-1]
        U = U[:,::-1]  #関数と行列の順番が違うため変換numpy
        Sigma_inv = np.diag(1.0 / sigma) # 各対角成分を逆数にするだけ
        Sigma_inv = np.diag(1.0 / sigma)
        V = A.T @ U @ Sigma_inv
    VT = V.T
    return U ,sigma, VT
```

基本的には授業資料に説明されていた通りの計算方法でプログラムを作成したが、numpy で実装されている行列と自分の関数で実装した初めの固有値分解の時に出てくる行列の順番が違っていたので順番を入れ替える操作をしました。

2.4 実行結果

以下のように自作関数と np.linalg.svd を比較しました。

```python
for i in tqdm(range(10)):
    m = rng.integers(low = 2, high = 10)
    n = rng.integers(low = 2, high = 10)
    A = rng.random(size=(m, n))

    #分解を利用する関数のほうQR
    U, sigma, VT = svd_via_qr(A)

    with np.printoptions(formatter={'float': '{: 12.8f}'.format}, suppress=True
        , linewidth=100):
        U_np, sigma_np, VT_np = np.linalg.svd(A)

        print("sigma == sigma_np", np.allclose(sigma, sigma_np))

        for i in range(U.shape[1]):
            print(
```

```
                f"column {i}: U[{i}] == U_np[{i}] ?",
                np.allclose(U[:, i], U_np[:, i]) or np.allclose(U[:, i], -U_np[:, i])
            )
        for i in range(len(VT)):
            print(
                f"column {i}: VT[{i}] == VT_np[{i}] ?",
                np.allclose(VT[i, :], VT_np[i, :]) or np.allclose(VT[i, :], -VT_np[i, :])
            )


for i in tqdm(range(10)):
    m = rng.integers(low = 2, high = 10)
    n = rng.integers(low = 2, high = 10)
    A = rng.random(size=(m, n))

    U, sigma, VT = svd_via_eigh(A)


    with np.printoptions(formatter={'float': '{: 12.8f}'.format}, suppress=True, linewidth=100):
        U_np, sigma_np, VT_np = np.linalg.svd(A)

        print("sigma == sigma_np", np.allclose(sigma, sigma_np))

        for i in range(U.shape[1]):
            print(
                f"column {i}: U[{i}] == U_np[{i}] ?",
                np.allclose(U[:, i], U_np[:, i]) or np.allclose(U[:, i], -U_np[:, i])
            )
        for i in range(len(VT)):
            print(
                f"column {i}: VT[{i}] == VT_np[{i}] ?",
                np.allclose(VT[i, :], VT_np[i, :]) or np.allclose(VT[i, :], -VT_np[i, :])
            )
```

行列は符号が違うことがあるのでそれを考慮して allclose で値自身一致しているか確認を行いました。また、自作関数は、thin 関数なので固有値分解したときに出てくる行列は numpy 関数と同じ行列のサイズが期待されるが、そのあとに出てくるもう一方の行列は実際の SVD と同じサイズで出力されない。なので for 文を自作関数のサイズに合わせてあっているかどうか判定した。
結果としてはそれぞれの関数を 10 回回して False が出ることはなかった。よって自作関数が numpy 関数とほぼ一致していることがわかりました。