



Computational Physics (PHYS6350)

Lecture 14: Linear algebra and matrices: Part II

- Matrix inversion
- Tri- and band-diagonal systems
- QR decomposition
- Eigenvalue problem

March 7, 2023

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: <https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

Matrix inversion

Inverse of matrix \mathbf{A} , if it exists, satisfies

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}.$$

Let us denote the columns of the inverse matrix \mathbf{A}^{-1} by \mathbf{x}_k , i.e. $\mathbf{A}^{-1} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$

The vectors \mathbf{x}_k satisfy the following N systems of non-linear equations

$$\mathbf{A}\mathbf{x}_k = \mathbf{v}_k, \quad k = 1 \dots N$$

where $\mathbf{v}_{k,j} = \delta_{kj}$

Since the matrix \mathbf{A} is always the same, these systems can be efficiently solved with LU-decomposition to find all \mathbf{x}_k and thus the inverse matrix \mathbf{A}^{-1}

Complexity: 1 LU-decomposition $O(N^3)$ + N backsubstitutions [each $O(N^2)$] $\sim O(N^3)$

Matrix inversion

```
def matrix_inverse_with_ludecomp(A):  
    # First step: LU decomposition of matrix A  
    L, U, row_map = lu_decomp_partialpivot(A)  
    N = len(row_map)  
  
    Ainv = A.copy()  
    for c in range(N):  
        v = np.zeros(N, float)  
        v[c] = 1.  
        x = solve_using_lu_partialpivot(L,U,row_map,v)  
        Ainv[:,c] = x  
  
    return Ainv
```

```
A = np.array([[ 0,  1,  4,  1 ],  
              [ 3,  4, -1, -1 ],  
              [ 1, -4,  1,  5 ],  
              [ 2, -2,  1,  3 ]],float)
```

```
Ainv = matrix_inverse_with_ludecomp(A)  
print("A*A^{-1} = \n", tabulate(A.dot(Ainv)))
```

A*A⁻¹ =

-----	-----	-----	-----
1	-5.55112e-17	1.11022e-16	-1.11022e-16
-2.77556e-17	1	-1.11022e-16	3.33067e-16
2.77556e-17	5.55112e-17	1	4.44089e-16
-2.77556e-17	-5.55112e-17	-5.55112e-16	1
-----	-----	-----	-----

Tridiagonal systems

Tridiagonal system of equation is a special case when the matrix \mathbf{A} is tridiagonal

$$\mathbf{A} = \begin{pmatrix} d_1 & u_1 & & & \\ l_2 & d_2 & u_2 & & \\ & l_3 & \ddots & \ddots & \\ & & \ddots & \ddots & u_{n-1} \\ & & & l_n & d_n \end{pmatrix}$$

Tridiagonal systems of linear equations often appear in physics, e.g.

- Nearest-neighbor interaction (linear chain of springs)
- Finite differences applied to partial differential equations (heat equation)

$$\frac{\partial u(t, x)}{\partial t} = \alpha \frac{\partial^2 u(t, x)}{\partial x^2} \quad \longrightarrow \quad \begin{pmatrix} \frac{\partial u_1(t)}{\partial t} \\ \frac{\partial u_2(t)}{\partial t} \\ \vdots \\ \frac{\partial u_N(t)}{\partial t} \end{pmatrix} = \frac{\alpha}{\Delta x} \begin{pmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & 1 & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_N(t) \end{pmatrix}$$

Solving tridiagonal systems

$$\mathbf{A} = \begin{pmatrix} d_1 & u_1 & & & \\ l_2 & d_2 & u_2 & & \\ & l_3 & \ddots & \ddots & \\ & & \ddots & \ddots & u_{n-1} \\ & & & l_n & d_n \end{pmatrix}$$

Tridiagonal system can be solved in linear time $[O(N)]$ with Gaussian elimination

- At each step need to subtract only one row below the current one, and at most two elements
- Backsubstitution step: subtract only single element from the upper superdiagonal

$$\begin{aligned} x_n &= \tilde{v}_n, \\ x_k &= \tilde{v}_k - \tilde{u}_k x_{k+1}, \quad k = 1 \dots N - 1. \end{aligned}$$

Solving tridiagonal systems

```
# Solve tridiagonal system of linear equations
# d: vector of diagonal elements
# l: vector of elements on the lower subdiagonal
# u: vector of elements on the upper superdiagonal
# v0: right-hand-side vector
def linsolve_tridiagonal(d, l, u, v0):
    # Initialization
    N = len(v0)
    a = d.copy() # Current diagonal elements
    b = u.copy() # Current upper diagonal elements
    v = v0.copy()

    # Gaussian elimination
    for r in range(N):
        if (a[r] == 0.):
            print("Diagonal element is zero! Cannot solve")
            return None
        b[r] /= a[r]
        v[r] /= a[r]
        a[r] = 1.
        if (r < N - 1):
            a[r + 1] -= l[r+1] * b[r]
            v[r + 1] -= l[r+1] * v[r]

    # Backsubstitution
    x = np.empty(N, float)
    x[N - 1] = v[N - 1]
    for r in range(N-2, -1, -1):
        x[r] = v[r] - b[r] * x[r + 1]

    return x
```

Test for a random tridiagonal matrix

```
def random_tridiagonal(n):
    A = np.random.rand(n, n)
    for r in range(n):
        for c in range(0, r-1):
            A[r][c] = 0.
        for c in range(r + 2, n):
            A[r][c] = 0.
    return A
```

```
n = 8
A = random_tridiagonal(n)
print("A =\n", tabulate(A))
v = np.random.rand(n)
x = linsolve_tridiagonal(*(get_tridiagonal(A)), v)
print(" x = ", x)
print("Ax = ", A.dot(x))
print(" v = ", v)
```

```
A =
-----
0.258085  0.365256  0          0          0          0          0          0
0.598828  0.60432  0.0777664  0          0          0          0          0
0          0.839862  0.209281  0.418171  0          0          0          0
0          0          0.730628  0.406732  0.789413  0          0          0
0          0          0          0.597044  0.627719  0.575418  0          0
0          0          0          0          0.639515  0.293871  0.737697  0
0          0          0          0          0          0.74846  0.477541  0.704847
0          0          0          0          0          0          0.00826684  0.109495
-----
x = [ 2.2288744 -0.6670926 -5.72667763  5.03759801  3.51116171 -7.41798625
      1.22249979  8.40485586]
Ax = [0.33158055 0.48623086 0.34782961 0.63663712 0.94324165 0.96734828
      0.9558695  0.93039484]
v = [0.33158055 0.48623086 0.34782961 0.63663712 0.94324165 0.96734828
      0.9558695  0.93039484]
```

Band-diagonal systems

Band-diagonal system: in each row

- at most m_{lower} non-zero elements to the left of the main diagonal
- at most m_{upper} non-zero elements to the right of the main diagonal

$$\mathbf{A} = \begin{pmatrix} d_1 & u_{1,1} & u_{2,1} & & & & \\ l_{1,2} & d_2 & u_{1,2} & u_{2,2} & & & \\ l_{2,3} & l_{1,3} & d_3 & u_{1,3} & u_{2,3} & & \\ & l_{2,4} & l_{1,4} & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots & u_{1,n-2} & u_{2,n-2} \\ & & & \ddots & \ddots & \ddots & u_{1,n-1} \\ & & & & l_{2,n} & l_{1,n} & d_n \end{pmatrix}$$

Generalization of tridiagonal systems ($m_{\text{lower}} = m_{\text{upper}} = 1$)

- k -nearest-neighbor interaction (linear chain of springs)
- High-order finite difference applied to partial differential equations (heat equation)

Solving band-diagonal systems

$$\mathbf{A} = \begin{pmatrix} d_1 & u_{1,1} & u_{2,1} & & & & \\ l_{1,2} & d_2 & u_{1,2} & u_{2,2} & & & \\ l_{2,3} & l_{1,3} & d_3 & u_{1,3} & u_{2,3} & & \\ & l_{2,4} & l_{1,4} & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \ddots & u_{1,n-2} & u_{2,n-2} \\ & & & \ddots & \ddots & \ddots & u_{1,n-1} \\ & & & & l_{2,n} & l_{1,n} & d_n \end{pmatrix}$$

Solving band-diagonal system proceeds also through Gaussian elimination

- At each step one has to normalize $m_{\text{upper}} + 1$ elements in the current row, then subtract the current row from at most m_{lower} rows below it. $O(N \times m_{\text{upper}} \times m_{\text{lower}})$
- Backsubstitution step: in each row subtract up to m_{upper} elements to the right from the main diagonal. $O(N \times m_{\text{upper}})$

Overall complexity: $O(N \times m_{\text{upper}} \times m_{\text{lower}})$

Solving band-diagonal systems

```
# Solving linear system of banded equations
# d: diagonal elements of the banded matrix
# l: lower non-zero diagonals of the banded matrix
# u: upper non-zero diagonals of the banded matrix
# v0: r.h.s. vector
def linsolve_banded(d, l, u, v0):
    # Initialization
    v = v0.copy()
    N = len(v)
    mlower = len(l)
    mupper = len(u)

    a = d.copy() # Diagonal elements
    dup = u.copy() # Upper diagonal elements
    dlow = l.copy() # Lower diagonal elements

    # Gaussian elimination
    for r in range(N):
        # Divide row r by diagonal element
        div = a[r]
        if (div == 0.):
            print("Diagonal element is zero! Cannot solve the system with simple")
            return None
        for c in range(r+1, min(r + mupper + 1, N)):
            dup[c-r-1, r] /= div
        a[r] /= div
        v[r] /= div

        # Now subtract this row from the lower rows
        # We do not need to go through all the rows
        # but only up to a certain row depending
        # on the number of non-zero elements to right and left of the main diagonal
        max_row = min(r + mlower, N - 1)

        # First the vector
        for r2 in range(r+1, max_row + 1):
            v[r2] -= dlow[r2 - r - 1, r2] * v[r]

        # Then the matrix rows
        for c in range(r + 1, min(r + mupper + 1, N)):
            for r2 in range(r+1, max_row + 1):
                if (c == r2):
                    a[r2] -= dlow[r2 - r - 1, r2] * dup[c-r-1, r]
                elif (c < r2 and r2 - c - 1 < mlower):
                    dlow[r2 - c - 1, r2] -= dlow[r2 - r - 1, r2] * dup[c-r-1, r]
                elif (c > r2 and c - r2 - 1 < mupper):
                    dup[c - r2 - 1, r2] -= dlow[r2 - r - 1, r2] * dup[c-r-1, r]

    # Backsubstitution
    x = np.empty(N, float)
    for r in range(N-1, -1, -1):
        x[r] = v[r]
        for c in range(r+1, min(r + mupper + 1, N)):
            x[r] -= dup[c - r - 1, r] * x[c]

    return x
```

```
def random_banded(n, mlower, mupper):
    A = np.random.rand(n, n)
    for r in range(n):
        for c in range(0, r-mlower):
            A[r][c] = 0.
        for c in range(r + mupper + 1, n):
            A[r][c] = 0.
    return A
```

```
n = 10
mupper = 4
mlower = 3
A = random_banded(n, mlower, mupper)
print("A =\n", tabulate(A))
v = np.random.rand(n)
x = linsolve_banded(*get_banded(A, mlower, mupper), v)
print(" x = ", x)
print("Ax = ", A.dot(x))
print(" v = ", v)
```

```
A =
-----
0.398214  0.972929  0.249665  0.0841269  0.667612  0          0          0          0          0
0.668918  0.27899  0.98081  0.311536  0.395787  0.250914  0          0          0          0
0.103901  0.955985  0.591015  0.178139  0.194452  0.565519  0.207813  0          0          0
0.094764  0.417524  0.401467  0.507737  0.198962  0.872389  0.999096  0.0251162  0          0
0          0.624273  0.727381  0.409819  0.859236  0.00593723  0.68357  0.707848  0.427377  0
0          0          0.304125  0.768458  0.03066  0.322861  0.123774  0.0664194  0.775458  0.117592
0          0          0          0.294081  0.698817  0.82127  0.43601  0.743528  0.107041  0.917906
0          0          0          0          0.642317  0.763591  0.0719641  0.752717  0.223502  0.607639
0          0          0          0          0          0.18334  0.0233041  0.0637123  0.95644  0.420615
0          0          0          0          0          0          0.698929  0.919066  0.824737  0.521482
-----
x = [-0.28014041  0.17110358  1.10225327 -1.21247032  0.14011191  0.26755524
      0.024373  -0.04051962  0.83538165  0.223171 ]
Ax = [0.32164853  0.68630553  0.75354435  0.15641687  0.87865905  0.1685409
      0.23585233  0.58787029  0.93990131  0.78514492]
v = [0.32164853  0.68630553  0.75354435  0.15641687  0.87865905  0.1685409
      0.23585233  0.58787029  0.93990131  0.78514492]
```

QR decomposition*

Any real square matrix **A** permits a decomposition

$$\mathbf{A} = \mathbf{QR}$$

where

- **Q** is orthogonal, $\mathbf{Q}^{-1} = \mathbf{Q}^T$, and thus, $\mathbf{Q}^T \mathbf{Q} = \mathbf{I}$
- **R** is upper diagonal

There are many algorithms for constructing the QR decomposition, from simple Gram-Schmidt process to more involved methods using Householder transformation and Givens rotations. These are readily implemented in linear algebra packages.

QR decomposition and systems of linear equations

System of linear equations

$$\mathbf{Ax} = \mathbf{v}.$$

If $\mathbf{A} = \mathbf{QR}$ we can rewrite the system as

$$\mathbf{QRx} = \mathbf{v}.$$

Multiplying each side of the equation by \mathbf{Q}^T , we have

$$\mathbf{Rx} = \mathbf{Q}^T \mathbf{v}.$$

The matrix \mathbf{R} is upper triangular, thus, the system can be solved using backsubstitution.

```
def linsolve_using_qr(Q,R,v):  
    # Initialization  
    N = len(v)  
    # Calculate y = Q^T v  
    y = np.zeros(N,float)  
    for r in range(N):  
        for c in range(N):  
            y[r] += Q[c][r] * v[c]  
  
    # Backsubstitution for R*x = y  
    x = np.empty(N,float)  
    for r in range(N-1,-1,-1):  
        x[r] = y[r]  
        for c in range(r+1,N):  
            x[r] -= R[r][c] * x[c]  
  
        x[r] /= R[r][r]  
    return x
```

```
A = np.array([[ 0,  1,  4,  1 ],  
              [ 3,  4, -1, -1 ],  
              [ 1, -4,  1,  5 ],  
              [ 2, -2,  1,  3 ]],float)  
  
Q, R = np.linalg.qr(A)  
v = np.array([ -4,  3,  9,  7 ],float)  
x = linsolve_using_qr(Q,R,v)  
print('x  =', x)  
print('Ax =', A.dot(x))  
print('v  =', v)
```

Eigenvalue problem

A common matrix problem in physics is the calculation of eigenvalues and eigenvectors of a matrix (e.g. classical and quantum mechanics).

The eigenvalue problem corresponds to the equation

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}.$$

Here λ are the eigenvalues and \mathbf{v} are the eigenvectors

General approach (not used in practice):

- Eigenvalues are roots of the characteristic polynomial

$$\det(\lambda I - A) = 0$$

- Once λ are found, the eigenvectors can be computed by solving linear system

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0$$

Eigenvalue problem

In most cases the matrix \mathbf{A} is either real symmetric or Hermitian (complex numbers).

In this case, for a $N \times N$ matrix there are N eigenvectors $\mathbf{v}_1, \dots, \mathbf{v}_N$ with real eigenvalues $\lambda_1, \dots, \lambda_N$. The eigenvectors are orthogonal, i.e. $\mathbf{v}_i \mathbf{v}_j = \delta_{ij}$ with the appropriate normalization.

The eigenvalue problem can thus be cast as a matrix equation

$$\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{D}.$$

Here \mathbf{V} is the matrix of eigenvectors, i.e. column k corresponds to the eigenvector \mathbf{v}_k , and \mathbf{D} is a diagonal matrix with entries corresponding to eigenvalues, $\mathbf{D} = \text{diag}(\lambda_1, \dots, \lambda_N)$.

How to solve the matrix equation $\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{D}$. ?

QR algorithm

QR algorithm

QR algorithm is a method of finding eigenvalues and/or eigenvectors. It is based on the following iterative procedure.

1. One starts with matrix $\mathbf{A}_1 = \mathbf{A}$ and calculates its QR decomposition $\mathbf{A}_1 = \mathbf{Q}_1 \mathbf{R}_1$.
2. The next matrix is computed as $\mathbf{A}_2 = \mathbf{R}_1 \mathbf{Q}_1$. This is a similarity transform. Multiplying both sides by $\mathbf{I} = \mathbf{Q}_1^T \mathbf{Q}_1$ and taking into account $\mathbf{A}_1 = \mathbf{Q}_1 \mathbf{R}_1$ one gets $\mathbf{A}_2 = \mathbf{Q}_1^T \mathbf{A}_1 \mathbf{Q}_1$.
3. The process is repeated as follows, $\mathbf{A}_{n+1} = \mathbf{R}_n \mathbf{Q}_n = \mathbf{Q}_n^T \mathbf{A}_n \mathbf{Q}_n$.
4. Matrix \mathbf{A}_n converges to diagonal form in the limit $n \rightarrow \infty$ (for real symmetric \mathbf{A}). It can be written as

$$\mathbf{A}_\infty = \mathbf{Q}^T \mathbf{A} \mathbf{Q},$$

where

$$\mathbf{Q} = \prod_{k=1}^{\infty} \mathbf{Q}_k$$

is an orthogonal matrix.

Multiplying \mathbf{A}_∞ by \mathbf{Q} from the right we get an equation

$$\mathbf{A} \mathbf{Q} = \mathbf{Q} \mathbf{A}_\infty.$$

Comparing to the matrix eigenvalue equation $\mathbf{A} \mathbf{V} = \mathbf{V} \mathbf{D}$ one recognizes that $\mathbf{Q} \equiv \mathbf{V}$ is the matrix of eigenvectors and $\mathbf{A}_\infty = \mathbf{D}$ is the eigenvector matrix.

In practice, the algorithm stops once non-diagonal elements of \mathbf{A}_n are below a certain threshold ϵ .

QR algorithm

```
def eigen_qr_simple(A, iterations=1000):
    Ak = np.copy(A)
    n = len(A[0])
    QQ = np.eye(n)
    for k in range(iterations):
        Q, R = np.linalg.qr(Ak)
        Ak = np.dot(R, Q)
        QQ = np.dot(QQ, Q)
        if k%100 == 0:
            print("A", k, "=")
            print(tabulate(Ak))
            print("\n")
    return Ak, QQ
```

```
n = 5
A = np.random.rand(n, n)
# Make symmetric
for r in range(n):
    for c in range(r):
        A[r][c] = A[c][r]
```

```
# We call the function
AQ = eigen_qr_simple(A)

# Print A' = D
print("A' =\n", tabulate(AQ[0]))
print("Q =\n", tabulate(AQ[1]))

# We compare our results with the official numpy algorithm
print(np.linalg.eig(A))

# Check orthogonality
print("v1*v2 = ", P[:,0].dot(P[:,1]))
```

```
A' =
-----
2.74844 -3.02815e-16 1.6284e-16 -5.32145e-17 2.4959e-16
0 -0.746594 1.15176e-17 4.62374e-17 -1.69369e-17
0 9.02192e-67 -0.643849 -9.85419e-17 1.54078e-16
0 -4.94066e-324 -4.5242e-287 0.331455 -2.97135e-16
0 -4.94066e-324 -7.90505e-323 1.77529e-36 -0.305375
-----
Q =
-----
-0.537155 -0.550458 0.199338 0.418962 -0.439541
-0.445186 -0.37513 0.179416 -0.561737 0.559775
-0.483201 0.62303 0.264306 -0.365474 -0.418235
-0.457385 0.126918 -0.871099 0.0886914 0.0894995
-0.265687 0.38987 0.315282 0.606205 0.557247
-----
v1*v2 = -2.513426757818653e-16
```