Linear Algebra

July 4, 2025

Linear Algebra with SageMath

Sagemath provides standard constructions from linear algebra, e.g., the characteristic polynomial, echelon form, trace, decomposition, etc., of a matrix.

[48]:
```
M = Matrix([[1, 2, 3], [4, 5, 6]])
print(M)
# Creating a 3x3 identity matrix
I = identity_matrix(3)
print(I)
# Creating a random matrix
R = random_matrix(ZZ, 3, 3, x=-5, y=5)
print(R)
```

```
[1 2 3]
[4 5 6]
[1 0 0]
[0 1 0]
[0 0 1]
[ 0 -4 -2]
[-1 -3  0]
[ 0 -3 -3]
```

[49]:
```
M = Matrix([[1, 2, 3], [4, 5, 6]]) ; M
```

[49]:
```
[1 2 3]
[4 5 6]
```

[50]:
```
identity_matrix(3)
```

[50]:
```
[1 0 0]
[0 1 0]
[0 0 1]
```

[51]:
```
R=random_matrix(ZZ, 3, 3, x=-5, y=5) ; R
```

[51]:
```
[-5 -3 -5]
[ 1 -3 -1]
[-2  4 -1]
```

[53]:
```
A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
w = vector([1,1,-4])
```

[54]:
```
w*A
```

[54]:
```
(0, 0, 0)
```

[55]:
```
A*w
```

[55]:
```
(-9, 1, -2)
```

Solving matrix equations is easy, using the method solve_right. Evaluating A.solve_right(Y) returns a matrix (or vector) X so that $AX = Y$. Similarly, use A.solve_left(Y) to solve for X in $XA = Y$

[56]:
```
Y = vector([0, -4, -1])
X = A.solve_right(Y)
```

[57]:
```
X
```

[57]:
```
(-2, 1, 0)
```

[58]:
```
A * X
```

[58]:
```
(0, -4, -1)
```

[60]:
```
A.solve_right(Y)
```

[60]:
```
(-2, 1, 0)
```

[61]:
```
A = matrix([[1, 2], [3, 4]])
B = matrix([[5, 6], [7, 8]])
# Transpose
A_T = A.transpose()
print(f"A^T = \n{A_T}")

# Determinant
det_A = A.determinant()
print(f"det(A) = {det_A}")

# Trace
tr_A = A.trace()
print(f"tr(A) = {tr_A}")
```

```
A^T =
[1 3]
[2 4]
det(A) = -2
tr(A) = 5
```

[62]:
```
A.transpose()
```

[62]:
```
[1 3]
[2 4]
```

[63]:
```
A.determinant()
```

[63]:
```
-2
```

[66]:
```
A.trace()
```

[66]:
```
5
```

[67]:
```
# Creating an invertible matrix
A = matrix([[2, 1], [1, 1]])
print("Matrix A:")
print(A)

# Check if matrix is invertible
det_A = A.determinant()
print("det(A) =", det_A)

if det_A != 0:
    A_inv = A.inverse()
    print("\nA^(-1) =")
    print(A_inv)

    # Verify: A * A^(-1) = I
    product = A * A_inv
    print("\nA * A^(-1) =")
    print(product)
else:
    print("Matrix is not invertible (singular)")
    print(A.determinant())
    print(A.rank())
```

```
Matrix A:
[2 1]
[1 1]
det(A) = 1

A^(-1) =
[ 1 -1]
[-1  2]

A * A^(-1) =
[1 0]
[0 1]
```

[68]:
```
A = matrix([[2, 1], [1, 1]]) ; A
```

[68]:
```
[2 1]
[1 1]
```

[69]:
```
A.determinant()
```

[69]:
```
1
```

[70]:
```
A.inverse()
```

[70]:
```
[ 1 -1]
[-1  2]
```

[71]:
```
A*A.inverse()
```

[71]:
```
[1 0]
[0 1]
```

[ ]:
```
A.rank()
```

[ ]:
```
# Properties of determinants
B3 = matrix(QQ, [[2, 0, 1], [1, 3, 2], [0, 1, 1]])
print("\nMatrix B:")
print(B3)
print("det(B) =", B3.determinant())

# det(AB) = det(A)det(B)
product = A3 * B3
print("det(A)det(B) =", A3.determinant() * B3.determinant())
print("det(AB) =", product.determinant())
```

[72]:
```
# Define the coefficient matrix and constant vector
A = matrix([[2, 1], [1, 3]])
b = vector([5, 7])

# Solve Ax = b
x = A.solve_right(b)
print(f"Solution: x = {x}")

# Verify the solution
verification = A * x
print(f"Verification: A * x = {verification}")
print(f"b = {b}")
```

```
Solution: x = (8/5, 9/5)
Verification: A * x = (5, 7)
b = (5, 7)
```

[73]:
```
A.solve_right(b)
```

[73]: (8/5, 9/5)

[74]:
```
# Define the system: Ax = b
# Example: 2x + 3y = 7, x - y = 1
A = matrix([[2, 3], [1, -1]])
b = vector([7, 1])

print("Coefficient matrix A:")
print(A)
print("\nConstant vector b:")
print(b)

# Solve the system
x = A.solve_right(b)
print("\nSolution x =", x)

# Verify the solution
verification = A * x
print("Verification: A * x =", verification)
print("Should equal b =", b)
```

```
Coefficient matrix A:
[ 2  3]
[ 1 -1]

Constant vector b:
(7, 1)

Solution x = (2, 1)
Verification: A * x = (7, 1)
Should equal b = (7, 1)
```

[75]:
```
 A.solve_right(b)
```

[75]: (2, 1)

[ ]:
```
Exercise : Solve the system of linear equations:
2x + 3y - z = 1
x - y + 2z  = 4
3x + y + z = 2
```

[ ]:
```
```

[76]:
```
# Define coefficient matrix A and constant vector b
A = matrix([[2, 3, -1], [1, -1, 2], [3, 1, 1]])
b = vector([1, 4, 2])
```

```
print("Coefficient matrix A:")
print(A)
print("\nConstant vector b:")
print(b)

# Check if system has unique solution by computing determinant
det_A = A.determinant()
print(f"\nDeterminant of A: {det_A}")

if det_A != 0:
    # Solve the system
    x = A.solve_right(b)
    print(f"\nSolution: x = {x[0]}, y = {x[1]}, z = {x[2]}")

    # Verify the solution
    verification = A * x
    print(f"Verification: A * x = {verification}")
    print(f"Should equal b = {b}")
    print(f"Check: A * x == b? {verification == b}")
else:
    print("System does not have a unique solution")
```

```
Coefficient matrix A:
[ 2  3 -1]
[ 1 -1  2]
[ 3  1  1]

Constant vector b:
(1, 4, 2)

Determinant of A: 5

Solution: x = -9/5, y = 3, z = 22/5
Verification: A * x = (1, 4, 2)
Should equal b = (1, 4, 2)
Check: A * x == b? True
```

[80]:
```
A = matrix([[2, 3, -1], [1, -1, 2], [3, 1, 1]])
b = vector([1, 4, 2])
```

[81]:
```
A.determinant()
```

[81]: 5

[82]:
```
A.solve_right(b)
```

[82]: (-9/5, 3, 22/5)

The syntax for the output of eigenvectors_left is a list of triples: (eigenvalue, eigenvector, multiplicity).) Eigenvalues and eigenvectors over QQ or RR can also be computed using Maxima . As noted in Basic Rings, the ring over which a matrix is defined affects some of its properties. In the following, the first argument to the matrix command tells Sage to view the matrix as a matrix of integers (the ZZ case), a matrix of rational numbers (QQ), or a matrix of reals (RR)

[83]:
```
# Eigenvalues and eigenvectors
A = matrix(QQ, [[4, -2], [1, 1]])
print("Matrix A:")
print(A)
# Characteristic polynomial
x = var('x')
char_poly = A.characteristic_polynomial()
print("Characteristic polynomial:", char_poly)
# Find eigenvalues
eigenvalues = A.eigenvalues()
print("Eigenvalues:", eigenvalues)
```

```
Matrix A:
[ 4 -2]
[ 1  1]
Characteristic polynomial: x^2 - 5*x + 6
Eigenvalues: [3, 2]
```

[84]:
```
A
```

[84]:
```
[ 4 -2]
[ 1  1]
```

[85]:
```
A.characteristic_polynomial()
```

[85]:
```
x^2 - 5*x + 6
```

[86]:
```
 A.eigenvalues ()
```

[86]:
```
[3, 2]
```

[87]:
```
# Find eigenvectors
eigenvectors_right = A.eigenvectors_right()
print("Right eigenvectors:")
for eigenval, eigenvecs, mult in eigenvectors_right:
    print(f"  = {eigenval}, multiplicity = {mult}")
    for vec in eigenvecs:
        print(f"  Eigenvector: {vec}")
        # Verify Av =  v
        print(f"  Verification: A*v = {A*vec},  *v = {eigenval*vec}")
```

```
Right eigenvectors:
  = 3, multiplicity = 1
```

```
 Eigenvector: (1, 1/2)
 Verification: A*v = (3, 3/2),  *v = (3, 3/2)
= 2, multiplicity = 1
 Eigenvector: (1, 1)
 Verification: A*v = (2, 2),  *v = (2, 2)
```

[88]:
```
A.eigenvectors_right()
```

[88]:
```
[(3, [(1, 1/2)], 1), (2, [(1, 1)], 1)]
```

[89]:
```
A.eigenvectors_left()
```

[89]:
```
[(3, [(1, -1)], 1), (2, [(1, -2)], 1)]
```

Find the eigenvalues and eigenvectors of the matrix:

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 3 & 0 \\ 2 & 1 & 1 \end{pmatrix}$$

[90]:
```
# Define the matrix
A = matrix([[1, 2, 0], [0, 3, 0], [2, 1, 1]])
print("Matrix A:")
print(A)

# Compute eigenvalues
eigenvals = A.eigenvalues()
print(f"\nEigenvalues: {eigenvals}")

# Compute eigenvectors
eigenvects = A.eigenvectors_right()
print("\nEigenvalues and corresponding eigenvectors:")
for eigenval, eigenvects_list, mult in eigenvects:
    print(f"\nEigenvalue   = {eigenval} (algebraic multiplicity: {mult})")
    for i, v in enumerate(eigenvects_list):
        print(f"  Eigenvector {i+1}: {v}")
        # Verify: A * v =   * v
        Av = A * v
        lambda_v = eigenval * v
        print(f"  Verification: A*v = {Av}")
        print(f"   *v = {lambda_v}")
        print(f"  Equal? {Av == lambda_v}")

# Characteristic polynomial
char_poly = A.characteristic_polynomial()
print(f"\nCharacteristic polynomial: {char_poly}")
```

```
Matrix A:
[1 2 0]
```

```
[0 3 0]
[2 1 1]

Eigenvalues: [3, 1, 1]

Eigenvalues and corresponding eigenvectors:

Eigenvalue   = 3 (algebraic multiplicity: 1)
  Eigenvector 1: (1, 1, 3/2)
  Verification: A*v = (3, 3, 9/2)
   *v = (3, 3, 9/2)
  Equal? True

Eigenvalue   = 1 (algebraic multiplicity: 2)
  Eigenvector 1: (0, 0, 1)
  Verification: A*v = (0, 0, 1)
   *v = (0, 0, 1)
  Equal? True

Characteristic polynomial: x^3 - 5*x^2 + 7*x - 3
```

[91]:
```
A.eigenvalues()
```

[91]: [3, 1, 1]

[92]:
```
A.eigenvectors_right()
```

[92]: [(3, [(1, 1, 3/2)], 1), (1, [(0, 0, 1)], 2)]

[93]:
```
A.characteristic_polynomial()
```

[93]: x^3 - 5*x^2 + 7*x - 3

[94]:
```
# Example of matrix diagonalization
A = matrix([[3, 1], [0, 2]])
print("Matrix A:")
print(A)

# Check if diagonalizable and find diagonalization
try:
    D, P = A.jordan_form(transformation=True)
    print("\nDiagonal form D:")
    print(D)
    print("\nTransformation matrix P:")
    print(P)

    # Verify: P^(-1) * A * P = D
    P_inv = P.inverse()
```

```
    result = P_inv * A * P
    print("\nVerification P^(-1) * A * P:")
    print(result)

except:
    print("Matrix is not diagonalizable over the rationals")
```

```
Matrix A:
[3 1]
[0 2]

Diagonal form D:
[3|0]
[-+-]
[0|2]

Transformation matrix P:
[ 1  1]
[ 0 -1]

Verification P^(-1) * A * P:
[3 0]
[0 2]
```

[95]:
```
A = matrix([[3, 1], [0, 2]])
A.jordan_form()
```

[95]:
```
[3|0]
[-+-]
[0|2]
```

[99]:
```
D, P = A.jordan_form(transformation=True)
```

[96]:
```
H = P.inverse() ; H
```

[96]:
```
[ 1  1]
[ 0 -1]
```

[97]:
```
H*A*P
```

[97]:
```
[3 0]
[0 2]
```

[98]:
```
A.jordan_form(transformation=True)
```

[98]:
```
(
[3|0]
[-+-]  [ 1  1]
```

```
[0|2], [ 0 -1]
)
```

[108]:
```
AZ = matrix(ZZ, [[2,0], [0,1]])
AQ = matrix(QQ, [[2,0], [0,1]])
AR = matrix(RR, [[2,0], [0,1]])
```

[109]:
```
AZ.echelon_form()
```

[109]:
```
[2 0]
[0 1]
```

[101]:
```
AQ.echelon_form()
```

[101]:
```
[1 0]
[0 1]
```

[104]:
```
AR.echelon_form()
```

[104]:
```
[2 0]
[0 1]
```

For computing eigenvalues and eigenvectors of matrices over floating point real or complex numbers, the matrix should be defined over RDF (Real Double Field) or CDF (Complex Double Field), respectively. If no ring is specified and floating point real or complex numbers are used then by default the matrix is defined over the RR or CC fields, respectively, which do not support these computations for all the cases:

[111]:
```
ARDF = matrix(RDF,[[1.2, 2], [2, 3]])
ARDF.eigenvalues()  # rel tol 8e-16
```

[111]:
```
[-0.09317121994613098, 4.293171219946131]
```

[106]:
```
ACDF = matrix(CDF, [[1.2, 1], [2, 3]])
ACDF.eigenvectors_right()  # rel tol 3e-15
```

[106]:
```
[(0.42369453857597894, [(0.789915495116472, -0.6132157129223768)], 1),
 (3.776305461424021, [(0.361850067513069, 0.9322363051505703)], 1)]
```

[ ]:
```
# Check if matrix is invertible
A = matrix([[1, 2], [3, 4]])
if A.is_invertible():
    A_inv = A.inverse()
    print(f"A^(-1) = \n{A_inv}")

    # Verify A * A^(-1) = I
    product = A * A_inv
    print(f"A * A^(-1) = \n{product}")
```

```
else:
    print("Matrix is not invertible")
```

[112]:
```
A = matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Row space
row_space = A.row_space()
print(f"Row space dimension: {row_space.dimension()}")
print(f"Row space basis: {row_space.basis()}")
# Null space (kernel)
null_space = A.kernel()
print(f"Null space dimension: {null_space.dimension()}")
print(f"Null space basis: {null_space.basis()}")
# Column space
col_space = A.column_space()
print(f"Column space dimension: {col_space.dimension()}")
print(f"Column space basis: {col_space.basis()}")
# Null space (kernel)
null_space = A.kernel()
print(f"Null space dimension: {null_space.dimension()}")
print(f"Null space basis: {null_space.basis()}")
# Rank
rank = A.rank()
print(f"Rank of A: {rank}")
```

```
Row space dimension: 2
Row space basis: [
(1, 2, 3),
(0, 3, 6)
]
Null space dimension: 1
Null space basis: [
(1, -2, 1)
]
Column space dimension: 2
Column space basis: [
(1, 1, 1),
(0, 3, 6)
]
Null space dimension: 1
Null space basis: [
(1, -2, 1)
]
Rank of A: 2
```

[113]:
```
A = matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

[114]:
```
A.row_space()
```

[114]:
```
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
```

[115]:
```
A.kernel()
```

[115]:
```
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -2  1]
```

[116]:
```
A.column_space()
```

[116]:
```
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 1 1]
[0 3 6]
```

[117]:
```
A.rank()
```

[117]:
```
2
```

MATRIX SPACE To specify the space of 3 by 4 matrices, you would use MatrixSpace(QQ,3,4). If the number of columns is omitted, it defaults to the number of rows, so MatrixSpace(QQ,3) is a synonym for MatrixSpace(QQ,3,3).) The space of matrices is equipped with its canonical basis:

[131]:
```
M = MatrixSpace(QQ,4)
```

[132]:
```
B = M.basis()
len(B)
```

[132]:
```
16
```

[133]:
```
B[0,1]
```

[133]:
```
[0 1 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

[134]:
```
A = M(range(16)); A
```

[134]:
```
[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]
[12 13 14 15]
```

[125]:
```
 A.echelon_form()
```

[125]:
```
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
```

[126]:
```
A.kernel()
```

[126]:
```
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

[ ]:
```
# Creating vectors
v = vector([1, 2, 3])
w = vector(QQ, [4, 5, 6])  # over rationals
print(v + w)
```

[135]:
```
# Vector operations
u = vector([1, 0, 0])
v = vector([0, 1, 0])
print(u + v)          # vector addition
print(3 * u)          # scalar multiplication
print(u.dot_product(v)) # dot product
print(u.cross_product(v)) # cross product
```

```
(1, 1, 0)
(3, 0, 0)
0
(0, 0, 1)
```

[136]:
```
u.dot_product(v)
```

[136]:
```
0
```

[137]:
```
u.cross_product(v)
```

[137]:
```
(0, 0, 1)
```

[138]:
```
# Define 3D points
P1 = vector([1, 2, 3])
P2 = vector([4, 5, 6])
P3 = vector([2, 1, 4])

# Vector operations
v1 = P2 - P1  # Vector from P1 to P2
v2 = P3 - P1  # Vector from P1 to P3

print(f"Vector v1: {v1}")
print(f"Vector v2: {v2}")
```

```
# Dot product and cross product
dot_product = v1.dot_product(v2)
cross_product = v1.cross_product(v2)

print(f"Dot product: {dot_product}")
print(f"Cross product: {cross_product}")

# Magnitude of vectors
mag_v1 = v1.norm()
mag_v2 = v2.norm()
print(f"Magnitude of v1: {mag_v1}")
print(f"Magnitude of v2: {mag_v2}")
```

```
Vector v1: (3, 3, 3)
Vector v2: (1, -1, 1)
Dot product: 3
Cross product: (6, 0, -6)
Magnitude of v1: 3*sqrt(3)
Magnitude of v2: sqrt(3)
```

[139]:
```
# Define 3D points
P1 = vector([1, 2, 3])
P2 = vector([4, 5, 6])
P3 = vector([2, 1, 4])

# Vector operations
v1 = P2 - P1  # Vector from P1 to P2
v2 = P3 - P1  # Vector from P1 to P3
v1.dot_product(v2)
```

[139]:
```
3
```

[140]:
```
v1
```

[140]:
```
(3, 3, 3)
```

[141]:
```
v2
```

[141]:
```
(1, -1, 1)
```

[142]:
```
v1.cross_product(v2)
```

[142]:
```
(6, 0, -6)
```

[143]:
```
v1.norm()
```

[143]:
```
3*sqrt(3)
```

[144]:
```
show(v1.norm())
```

$3\sqrt{3}$

[145]:
```
show(v2.norm())
```

$\sqrt{3}$

Let illustrate computation of matrices defined over finite fields:

[1]:
```
 M = MatrixSpace(GF(2),4,8)
```

[2]:
```
A = M([1,1,0,0, 1,1,1,1, 0,1,0,0, 1,0,1,1,
       0,0,1,0, 1,1,0,1, 0,0,1,1, 1,1,1,0])
```

[3]:
```
A
```

[3]:
```
[1 1 0 0 1 1 1 1]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
[0 0 1 1 1 1 1 0]
```

[4]:
```
rows = A.rows()
```

[5]:
```
A.columns()
```

[5]:
```
[(1, 0, 0, 0),
 (1, 1, 0, 0),
 (0, 0, 1, 1),
 (0, 0, 0, 1),
 (1, 1, 1, 1),
 (1, 0, 1, 1),
 (1, 1, 0, 1),
 (1, 1, 1, 0)]
```

[ ]:
```
rows
```

We make the subspace over F_2 spanned by the above rows.

[6]:
```
 V = VectorSpace(GF(2),8)
```

[7]:
```
S = V.subspace(rows)
```

[8]:
```
S
```

[8]:
```
Vector space of degree 8 and dimension 4 over Finite Field of size 2
Basis matrix:
[1 0 0 0 0 1 0 0]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 1]
```

```
[0 0 0 1 0 0 1 1]
```

[ ]:
```
A.echelon_form()
```

The basis of S used by Sage is obtained from the non-zero rows of the reduced row echelon form of the matrix of generators of S

[9]:
```
# Practical Application of Markov Chains : Transition matrix for a simple␣
↪weather model
P = matrix(RDF, [[0.7, 0.3], [0.4, 0.6]])

# Initial state
state = vector(RDF, [1, 0])  # Start with sunny day

print("Day 0:", state)

# Simulate 10 days
for day in range(1, 11):
    state = state * P
    print(f"Day {day}: {state}")

# Find steady state (eigenvector with eigenvalue 1)
eigenvalues = P.transpose().eigenvalues()
eigenvectors = P.transpose().eigenvectors_right()

for eigenval, eigenvecs, mult in eigenvectors:
    if abs(eigenval - 1) < 1e-10:
        steady_state = eigenvecs[0]
        # Normalize
        steady_state = steady_state / sum(steady_state)
        print(f"Steady state: {steady_state}")
```

```
Day 0: (1.0, 0.0)
Day 1: (0.7, 0.3)
Day 2: (0.6099999999999999, 0.39)
Day 3: (0.583, 0.4169999999999999)
Day 4: (0.5749, 0.4250999999999999)
Day 5: (0.5724699999999999, 0.42752999999999997)
Day 6: (0.5717409999999999, 0.42825899999999995)
Day 7: (0.5715222999999999, 0.42847769999999996)
Day 8: (0.5714566899999999, 0.42854330999999996)
Day 9: (0.5714370069999999, 0.4285629929999999)
Day 10: (0.5714311020999998, 0.42856889789999986)
Steady state: (0.5714285714285714, 0.4285714285714286)
```

[10]:
```
# Kronecker product
A = matrix([[1, 2], [3, 4]])
B = matrix([[0, 1], [1, 0]])
```

```
A_kron_B = A.tensor_product(B)
print(f"A   B = \n{A_kron_B}")
```

```
A   B =
[0 1|0 2]
[1 0|2 0]
[---+---]
[0 3|0 4]
[3 0|4 0]
```

[11]:
```
A.tensor_product(B)
```

[11]:
```
[0 1|0 2]
[1 0|2 0]
[---+---]
[0 3|0 4]
[3 0|4 0]
```

[ ]:
```

```