Aamer Jalan - Python Code

```python
class Complex:
    def __init__(self, real, img):
        self.real = real
        self.img = img

    def __add__(self, y):
        return Complex(self.real + y.real, self.img + y.img)

    def __sub__(self, y):
        return Complex(self.real - y.real, self.img - y.img)

    def __mul__(self, y):
        return Complex(self.real * y.real - self.img * y.img, self.real * y.img + self.img * y.real)

    def __truediv__(self, y):
        denominator = y.real**2 + y.img**2
        if denominator == 0:
            raise ValueError("Cannot divide by 0")
        return Complex((self.real * y.real + self.img * y.img) / denominator, (self.img * y.real - self.real * y.img) / denominator)

    def __abs__(self):
        return (self.real**2 + self.img**2)**0.5

    def conjugate(self):
        return Complex(self.real, -self.img)

class Vector:
    def __init__(self, field, n, *coordinates):
        if field not in ["real", "complex"]:
            raise ValueError("Field must be real or complex")

        self.field = field
        self.n = n

        if len(coordinates) != n:
            raise ValueError(f"There should be {n} coordinates")

        if field == "real":
            for coord in coordinates:
                if not isinstance(coord, (int, float)):
                    raise TypeError("coordinates must be real numbers i.e. integers or floats"
```

```python
                    )
        elif field == "complex":
            for coord in coordinates:
                if not isinstance(coord, Complex):
                    raise TypeError("coordinates must be complex numbers i.e. instances of the class Complex)")

        self.coordinates = list(coordinates)

    def inner_product(self, other):
        if self.n != other.n:
            raise ValueError("Vectors must have the same dimension")
        if self.field != other.field:
            raise ValueError("Vectors must belong to the same field")

        if self.field == "complex":
            return sum(a * b.conjugate() for a, b in zip(self.coordinates, other.coordinates))
        else:
            return sum(a * b for a, b in zip(self.coordinates, other.coordinates))

    def is_orthogonal_to(self, other):
        ip = self.inner_product(other)
        if self.field == "complex":
            modulus = abs(ip)
        else:
            modulus = abs(ip)
        return modulus < 1e-10

    @staticmethod
    def gram_schmidt(vectors):
        orthogonal_set = []
        for v in vectors:
            w_coords = v.coordinates.copy()
            for u in orthogonal_set:
                scalar = v.inner_product(u) / u.inner_product(u)
                w_coords = [a - scalar * b for a, b in zip(w_coords, u.coordinates)]
            v = Vector(v.field, v.n, *w_coords)
            orthogonal_set.append(v)
        return orthogonal_set

class Matrix:
    def __init__(self, field, n=None, m=None, *values_or_vectors):
```

```python
        if field not in ["real", "complex"]:
            raise ValueError("Field must be real or complex")

        self.field = field

        if all(isinstance(v, Vector) for v in values_or_vectors):
            vectors = values_or_vectors
            m = len(vectors)
            if m == 0:
                raise ValueError("At least one vector is required to form a matrix")
            n = len(vectors[0].coordinates)

            for vec in vectors:
                if vec.field != field:
                    raise TypeError("All vectors must match the specified field")
                if len(vec.coordinates) != n:
                    raise ValueError("All vectors must have the same length")

            self.entries = [[vec.coordinates[i] for vec in vectors] for i in range(n)]
            self.n = n
            self.m = m

        else:
            if n is None or m is None:
                raise ValueError("n and m must be specified")
            if len(values_or_vectors) != n * m:
                raise ValueError(f"There should be {n * m} values")

            if field == "real":
                for val in values_or_vectors:
                    if not isinstance(val, (int, float)):
                        raise TypeError("values must be real numbers")
            elif field == "complex":
                for val in values_or_vectors:
                    if not isinstance(val, Complex):
                        raise TypeError("values must be complex numbers")

            self.entries = []
            for i in range(n):
```

```python
                row = list(values_or_vectors[i * m: (i + 1) * m])
                self.entries.append(row)
            self.n = n
            self.m = m

    def __add__(self, other):
        if not isinstance(other, Matrix):
            raise TypeError("Add with Matrix only")
        if self.field != other.field:
            raise TypeError("Field mismatch")
        if self.n != other.n or self.m != other.m:
            raise ValueError("Dimensions must match")

        result_entries = []
        for i in range(self.n):
            row = []
            for j in range(self.m):
                row.append(self.entries[i][j] + other.entries[i][j])
            result_entries.extend(row)
        return Matrix(self.field, self.n, self.m, *result_entries)

    def __mul__(self, other):
        if not isinstance(other, Matrix):
            raise TypeError("Multiply with Matrix only")
        if self.field != other.field:
            raise TypeError("Fields do not match")
        if self.m != other.n:
            raise ValueError("Dimensions not compatible")

        result_entries = []
        for i in range(self.n):
            row = []
            for j in range(other.m):
                sum_product = self.entries[i][0] * other.entries[0][j]
                for k in range(1, self.m):
                    sum_product += self.entries[i][k] * other.entries[k][j]
                row.append(sum_product)
            result_entries.extend(row)
        return Matrix(self.field, self.n, other.m, *result_entries)

    def get_row(self, i):
        if i < 0 or i >= self.n:
            raise IndexError("Row out of range")
```

```python
        return Matrix(self.field, 1, self.m, *self.entries[i])

    def get_column(self, j):
        if j < 0 or j >= self.m:
            raise IndexError("Column out of range")
        return Matrix(self.field, self.n, 1, *[self.entries[i][j] for i in range(self.n)])

    def transpose(self):
        transposed_entries = [self.entries[j][i] for i in range(self.m) for j in range(self.n)]
        return Matrix(self.field, self.m, self.n, *transposed_entries)

    def conjugate(self):
        conjugated_entries = []
        for row in self.entries:
            conjugated_entries.extend([elem.conjugate() if isinstance(elem, Complex) else elem for elem in row])
        return Matrix(self.field, self.n, self.m, *conjugated_entries)

    def transpose_conjugate(self):
        return self.transpose().conjugate()

    def is_zero(self):
        return all(all(entry == 0 for entry in row) for row in self.entries)

    def is_symmetric(self):
        if self.n != self.m:
            return False
        return all(self.entries[i][j] == self.entries[j][i] for i in range(self.n) for j in range(i, self.m))

    def is_hermitian(self):
        if self.n != self.m or self.field != "complex":
            return False
        return all(self.entries[i][j] == self.entries[j][i].conjugate() for i in range(self.n) for j in range(i, self.m))

    def is_square(self):
        return self.n == self.m

    def is_orthogonal(self):
        if not self.is_square():
            return False
```

```python
        identity_matrix = self.identity_matrix(self.field, self.n)
        return self * self.transpose() == identity_matrix

    def identity_matrix(self, field, size):
        if field not in ["real", "complex"]:
            raise ValueError("Field must be real or complex")

        entries = [1 if i == j else 0 for i in range(size) for j in range(size)]
        return Matrix(field, size, size, *entries)

    def is_unitary(self):
        if not self.is_square() or self.field != "complex":
            return False
        identity_matrix = self.identity_matrix(self.field, self.n)
        return self.transpose_conjugate() * self == identity_matrix

    def is_scalar(self):
        if not self.is_square():
            return False
        diagonal_value = self.entries[0][0]
        return all(self.entries[i][i] == diagonal_value for i in range(self.n)) and \
            all(self.entries[i][j] == 0 for i in range(self.n) for j in range(self.m) if i != j)

    def rank(self):
        matrix = [row[:] for row in self.entries]
        rank = 0

        for col in range(self.m):
            for row in range(rank, self.n):
                if matrix[row][col] != 0:
                    matrix[rank], matrix[row] = matrix[row], matrix[rank]
                    break
            else:
                continue

            for i in range(rank + 1, self.n):
                if matrix[i][col] != 0:
                    factor = matrix[i][col] / matrix[rank][col]
                    matrix[i] = [matrix[i][j] - factor * matrix[rank][j] for j in range(self.m)]
```

```python
            rank += 1
        return rank

    def is_singular(self):
        if not self.is_square():
            return False
        return self.rank() < self.n

    def is_invertible(self):
        return self.is_square and not self.is_singular()

    def is_identity(self):
        if not self.is_square():
            return False
        return all(self.entries[i][i] == 1 for i in range(
            self.n)) and \
            all(self.entries[i][j] == 0 for i in range(self.
                n) for j in range(self.m) if i != j)

    def is_nilpotent(self):
        if not self.is_square():
            return False

        power = self
        for _ in range(1, self.n + 1):
            power = power * self
            if power.is_zero():
                return True
        return False

    def is_diagonalizable(self):
        if not self.is_square():
            return False

        if self.field == "real" and self.is_symmetric():
            return True

    def has_lu_decomposition(self):
        if not self.is_square():
            return False

        matrix = [row[:] for row in self.entries]
        n = self.n

        for k in range(n):
            if matrix[k][k] == 0:
                return False

            for i in range(k + 1, n):
```

```
                if matrix[i][k] != 0:
                    factor = matrix[i][k] / matrix[k][k]
                    for j in range(k, n):
                        matrix[i][j] -= factor * matrix[k][j]
        return True

    def vector_length(self, vector):
        if vector.field == "real":
            return sum(coord**2 for coord in vector.coordinates) ** 0.5
        elif vector.field == "complex":
            return sum(abs(coord)**2 for coord in vector.coordinates) ** 0.5

    def size(self):
        return self.n, self.m

    def nullity(self):
        return self.m - self.rank()

    def rref(self, show_steps=False):
        matrix = [row[:] for row in self.entries]
        n, m = self.n, self.m
        row_operations = []
        elementary_matrices = []

        def create_identity(size):
            identity = [[1 if i == j else 0 for j in range(size)] for i in range(size)]
            return identity

        for i in range(min(n, m)):
            pivot_row = None
            for row in range(i, n):
                if matrix[row][i] != 0:
                    pivot_row = row
                    break
            if pivot_row is None:
                continue

            if pivot_row != i:
                matrix[i], matrix[pivot_row] = matrix[pivot_row], matrix[i]
                if show_steps:
                    row_operations.append(f"Swap row {i} with row {pivot_row}")
                    elementary_matrices.append(Matrix(self.field, n, n, *create_identity(n)))

```

```python
            pivot = matrix[i][i]
            matrix[i] = [val / pivot for val in matrix[i]]
            if show_steps:
                row_operations.append(f"Normalize row {i}")
                elementary_matrices.append(Matrix(self.field, n, n, *create_identity(n)))

            for row in range(n):
                if row != i and matrix[row][i] != 0:
                    factor = matrix[row][i]
                    matrix[row] = [matrix[row][j] - factor * matrix[i][j] for j in range(m)]
                    if show_steps:
                        row_operations.append(f"Eliminate row {row} using row {i}")
                        elementary_matrices.append(Matrix(self.field, n, n, *create_identity(n)))

        if show_steps:
            return Matrix(self.field, n, m, *[elem for row in matrix for elem in row]), row_operations, elementary_matrices
        return Matrix(self.field, n, m, *[elem for row in matrix for elem in row])

    def are_linearly_independent(self, vectors):
        matrix = Matrix(self.field, len(vectors[0].coordinates), len(vectors), *[v.coordinates[i] for v in vectors for i in range(v.n)])
        return matrix.rank() == len(vectors)

    def dimension_of_span(self, vectors):
        matrix = Matrix(self.field, len(vectors[0].coordinates), len(vectors), *[v.coordinates[i] for v in vectors for i in range(v.n)])
        return matrix.rank()

    def basis_for_span(self, vectors):
        rref_matrix = self.rref()
        basis_vectors = []
        for i in range(rref_matrix.n):
            if any(rref_matrix.entries[i][j] != 0 for j in range(rref_matrix.m)):
                basis_vectors.append(Vector(self.field, rref_matrix.m, *rref_matrix.entries[i]))
        return basis_vectors

    def rank_factorization(self):
        if not self.is_square():
```

```
            raise ValueError("Rank factorization is only
                defined for square matrices")

        U = self.rref()
        non_zero_rows = [row for row in U.entries if any(val
             != 0 for val in row)]
        R = Matrix(self.field, len(non_zero_rows), self.m,
            *[val for row in non_zero_rows for val in row])
        C = Matrix(self.field, self.n, len(non_zero_rows),
            *[self.get_column(i).entries for i in range(len(
            non_zero_rows))])
        return R, C

    def lu_decompose(self):
        if not self.is_square():
            raise ValueError("LU decomposition is only
                defined for square matrices")

        if not self.has_lu_decomposition():
            raise ValueError("LU decomposition is not
                possible due to a zero pivot")

        L = [[0] * self.n for _ in range(self.n)]
        U = [[0] * self.n for _ in range(self.n)]
        matrix = [row[:] for row in self.entries]

        for i in range(self.n):
            for j in range(i, self.n):
                U[i][j] = matrix[i][j] - sum(L[i][k] * U[k][
                    j] for k in range(i))

            for j in range(i, self.n):
                if i == j:
                    L[i][i] = 1
                else:
                    L[j][i] = (matrix[j][i] - sum(L[j][k] *
                        U[k][i] for k in range(i))) / U[i][i]

        L_matrix = Matrix(self.field, self.n, self.n, *[val
            for row in L for val in row])
        U_matrix = Matrix(self.field, self.n, self.n, *[val
            for row in U for val in row])

        return L_matrix, U_matrix

    def plu_decompose(self):
        if not self.is_square():
            raise ValueError("PLU decomposition is only
                defined for square matrices")

```

```python
        n = self.n
        P = [[1 if i == j else 0 for j in range(n)] for i in range(n)]
        matrix = [row[:] for row in self.entries]

        for i in range(n):
            max_row = max(range(i, n), key=lambda r: abs(matrix[r][i]))
            if matrix[max_row][i] == 0:
                raise ValueError("Matrix is singular and cannot be decomposed")

            P[i], P[max_row] = P[max_row], P[i]

            matrix[i], matrix[max_row] = matrix[max_row], matrix[i]

        permuted_matrix = Matrix(self.field, n, n, *[val for row in matrix for val in row])

        L, U = permuted_matrix.lu_decompose()

        P_matrix = Matrix(self.field, n, n, *[val for row in P for val in row])

        return P_matrix, L, U

    def inverse_by_row_reduction(self):
        if not self.is_square():
            raise ValueError("Only square matrices have an inverse")
        if not self.is_invertible():
            raise ValueError("Matrix is not invertible")

        identity = Matrix(self.field, self.n, self.n, *[1 if i == j else 0 for i in range(self.n) for j in range(self.n)])
        augmented = Matrix(self.field, self.n, self.n * 2,
                           *[val for row in self.entries for val in row] +
                           [val for row in identity.entries for val in row])

        rref_augmented = augmented.rref()

        inverse_entries = [rref_augmented.entries[i][self.n:] for i in range(self.n)]
        inverse_flat = [val for row in inverse_entries for val in row]

```

```
        return Matrix(self.field, self.n, self.n, *inverse_flat)

    def inverse_by_adjoint(self):
        if not self.is_square():
            raise ValueError("Only square matrices have an inverse")
        if not self.is_invertible():
            raise ValueError("Matrix is not invertible")

        cofactor_entries = []
        for i in range(self.n):
            for j in range(self.n):
                minor_matrix = self.minor_matrix(i, j)
                cofactor = ((-1) ** (i + j)) * minor_matrix.determinant_by_rref()
                cofactor_entries.append(cofactor)

        adjugate = Matrix(self.field, self.n, self.n, *cofactor_entries).transpose()

        identity = Matrix(self.field, self.n, self.n, *[1 if i == j else 0 for i in range(self.n) for j in range(self.n)])
        augmented = Matrix(self.field, self.n, self.n * 2,
                           *[val for row in self.entries for val in row] +
                           [val for row in adjugate.entries for val in row])
        rref_augmented = augmented.rref()

        inverse_entries = [rref_augmented.entries[i][self.n:] for i in range(self.n)]
        inverse_flat = [val for row in inverse_entries for val in row]

        return Matrix(self.field, self.n, self.n, *inverse_flat)

    def determinant_by_rref(self):
        if not self.is_square():
            raise ValueError("Determinant is only defined for square matrices")

        matrix = [row[:] for row in self.entries]
        n = self.n
        determinant = 1

        for i in range(n):
            pivot_row = None
```

```
            for j in range(i, n):
                if matrix[j][i] != 0:
                    pivot_row = j
                    break
            if pivot_row is None:
                return 0

            if pivot_row != i:
                matrix[i], matrix[pivot_row] = matrix[pivot_row], matrix[i]
                determinant *= -1

            pivot = matrix[i][i]
            determinant *= pivot
            matrix[i] = [val / pivot for val in matrix[i]]

            for j in range(i + 1, n):
                factor = matrix[j][i]
                matrix[j] = [matrix[j][k] - factor * matrix[i][k] for k in range(n)]

        return determinant

    def minor_matrix(self, row, col):
        if row < 0 or row >= self.n or col < 0 or col >= self.m:
            raise IndexError("Row or column index out of range")
        minor_entries = [
            [self.entries[i][j] for j in range(self.m) if j != col]
            for i in range(self.n) if i != row
        ]
        return Matrix(self.field, self.n - 1, self.m - 1, *[val for sublist in minor_entries for val in sublist])

    def is_in_span(self, S, v):
        if not isinstance(v, Vector):
            raise TypeError("v must be a Vector")
        if not all(isinstance(vec, Vector) for vec in S):
            raise TypeError("S must contain Vector objects")
        if any(vec.field != self.field for vec in S) or v.field != self.field:
            raise ValueError("All vectors must belong to the same field")
        if any(len(vec.coordinates) != v.n for vec in S):
            raise ValueError("All vectors must have the same dimension as v")

```

```
        matrix_S = Matrix(self.field, S[0].n, len(S), *[vec.coordinates[i] for vec in S for i in range(vec.n)])

        coefficients, representation = matrix_S.linear_combination(v)
        return representation != "0"

    def linear_combination(self, S, v):
        if not self.is_in_span(S, v):
            raise ValueError("v is not in the span of S")

        matrix_S = Matrix(self.field, S[0].n, len(S), *[vec.coordinates[i] for vec in S for i in range(vec.n)])

        augmented = Matrix(self.field, v.n, len(S) + 1,
                           *[elem for row in matrix_S.entries for elem in row] + v.coordinates)
        rref_matrix = augmented.rref()

        coefficients = [rref_matrix.entries[i][-1] for i in range(len(S))]
        representation = " + ".join(f"({coeff})*{S[i].coordinates}" for i, coeff in enumerate(coefficients))

        return coefficients, representation

    def do_sets_span_the_same_space(self, S1, S2):
        if not all(isinstance(v, Vector) and v.field == self.field for v in S1 + S2):
            raise ValueError("All vectors must be instances of Vector and belong to the same field")

        for v in S2:
            if not self.is_in_span(S1, v):
                return False

        for v in S1:
            if not self.is_in_span(S2, v):
                return False

        return True

    def compute_coordinates(self, B, v):
        if not self.is_in_span(B, v):
            raise ValueError("v is not in the span of B")

```

```
        coefficients, representation = self.linear_combination(B, v)
        return coefficients, representation

    def vector_from_coordinates(self, B, coordinates):
        if len(B) != len(coordinates):
            raise ValueError("Number of basis vectors must match number of coordinates")
        if any(b.field != self.field for b in B):
            raise ValueError("All basis vectors must belong to the same field as the matrix")

        reconstructed = [0] * B[0].n
        for coeff, basis_vector in zip(coordinates, B):
            for i in range(len(reconstructed)):
                reconstructed[i] += coeff * basis_vector.coordinates[i]

        return Vector(self.field, len(reconstructed), *reconstructed)

    def change_of_basis_matrix(self, B1, B2):
        if len(B1) != len(B2):
            raise ValueError("B1 and B2 must have the same number of vectors")

        matrix_B1 = Matrix(self.field, B1[0].n, len(B1), *[b.coordinates[i] for b in B1 for i in range(b.n)])
        matrix_B2 = Matrix(self.field, B2[0].n, len(B2), *[b.coordinates[i] for b in B2 for i in range(b.n)])

        matrix_B1_inv = matrix_B1.inverse_by_row_reduction()
        return matrix_B2 * matrix_B1_inv

    def coordinates_in_new_basis(self, B1, B2, coordinates_B1):
        change_matrix = self.change_of_basis_matrix(B1, B2)
        coord_vector_B1 = Vector(self.field, len(coordinates_B1), *coordinates_B1)

        result_matrix = change_matrix * Matrix(self.field, len(coordinates_B1), 1, *coord_vector_B1.coordinates)
        return [result_matrix.entries[i][0] for i in range(result_matrix.n)]

    def determinant_by_cofactor(self):
        if not self.is_square():
            raise ValueError("Determinant is only defined for square matrices")
```

```python
        if self.n == 1:
            return self.entries[0][0]
        determinant = 0
        for col in range(self.m):
            minor = self.minor_matrix(0, col)
            cofactor = ((-1) ** col) * minor.determinant_by_cofactor()
            determinant += self.entries[0][col] * cofactor
        return determinant

    def determinant_by_plu(self):
        if not self.is_square():
            raise ValueError("Determinant is only defined for square matrices")
        P, _, U = self.plu_decompose()
        determinant = 1
        for i in range(self.n):
            determinant *= U.entries[i][i]
        return determinant * (-1) ** (sum(row.index(1) for row in P.entries))

    def determinant_by_rref_method(self):
        if not self.is_square():
            raise ValueError("Determinant is only defined for square matrices")
        return self.determinant_by_rref()

    def qr_factorization(matrix):
        Q = gram_schmidt([Vector(matrix.field, matrix.n, *matrix.get_column(i).coordinates) for i in range(matrix.m)])
        R = Matrix(matrix.field, len(Q), len(Q))
        for i, q in enumerate(Q):
            for j in range(i, len(Q)):
                R.entries[i][j] = inner_product(q, Vector(matrix.field, matrix.n, *matrix.get_column(j).coordinates))
        return Matrix(matrix.field, len(Q), matrix.n, *[elem for q in Q for elem in q.coordinates]), R

    def pseudoinverse(self):
        Q, R = self.qr_factorization()
        return R.inverse_by_row_reduction() * Q.transpose_conjugate()

    def least_squares_solution(self, b):
        pseudo_inv = self.pseudoinverse()
        return pseudo_inv * b

    def cholesky_decomposition(self):
```

```
        if not self.is_square() or not self.is_hermitian():
            raise ValueError("Cholesky decomposition requires a square, Hermitian positive-definite matrix")
        L = [[0] * self.n for _ in range(self.n)]
        for i in range(self.n):
            for j in range(i + 1):
                if i == j:
                    L[i][j] = (self.entries[i][i] - sum(L[i][k] ** 2 for k in range(j))) ** 0.5
                else:
                    L[i][j] = (self.entries[i][j] - sum(L[i][k] * L[j][k] for k in range(j))) / L[j][j]
        return Matrix(self.field, self.n, self.n, *[val for row in L for val in row])

class SystemOfEquations:
    def __init__(self, A, b):
        if not isinstance(A, Matrix):
            raise TypeError("A must be a Matrix")
        if not isinstance(b, Vector):
            raise TypeError("b must be a Vector")
        if A.n != b.n:
            raise ValueError("Matrix A and vector b dimensions do not match")
        self.A = A
        self.b = b

    def is_consistent(self):
        augmented_matrix = Matrix(self.A.field, self.A.n, self.A.m + 1,
                                  *[elem for row in self.A.entries for elem in row] + self.b.coordinates)
        return self.A.rank() == augmented_matrix.rank()

    def solve(self):
        if not self.is_consistent():
            raise ValueError("System is inconsistent and has no solution")

        augmented_matrix = Matrix(self.A.field, self.A.n, self.A.m + 1,
                                  *[elem for row in self.A.entries for elem in row] + self.b.coordinates)
        rref_matrix = augmented_matrix.rref()

        solution = []
```

```
        for i in range(self.A.m):
            if i < rref_matrix.n and rref_matrix.entries[i][i] != 0:
                solution.append(rref_matrix.entries[i][-1])
            else:
                solution.append(0)

        return Vector(self.A.field, len(solution), *solution)

    def is_subspace(S1, S2):
        combined = S2 + S1
        matrix = Matrix(S1[0].field, len(combined[0].coordinates), len(combined),
                        *[v.coordinates[i] for v in combined for i in range(v.n)])
        return matrix.rank() == len(S2)

    def solution_set(self):
        if not self.is_consistent():
            raise ValueError("System is inconsistent and has no solution")

        augmented_matrix = Matrix(self.A.field, self.A.n, self.A.m + 1,
                                  *[elem for row in self.A.entries for elem in row] + self.b.coordinates)
        rref_matrix = augmented_matrix.rref()

        free_vars = []
        basic_vars = []
        for i in range(self.A.m):
            if i < rref_matrix.n and rref_matrix.entries[i][i] != 0:
                basic_vars.append(i)
            else:
                free_vars.append(i)

        solutions = {}
        for var in basic_vars:
            solutions[f"x{var}"] = rref_matrix.entries[var][-1]
        for var in free_vars:
            solutions[f"x{var}"] = "Free"

        return solutions

    def solve_with_plu(self):
        if not self.A.is_square():
```

```
            raise ValueError("PLU decomposition requires a square matrix")
        if not self.is_consistent():
            raise ValueError("System is inconsistent and has no solution")

        P, L, U = self.A.plu_decompose()
        Pb = P * Matrix(self.A.field, self.b.n, 1, *self.b.coordinates)

        Y = [0] * self.A.n
        for i in range(self.A.n):
            Y[i] = Pb.entries[i][0] - sum(L.entries[i][j] * Y[j] for j in range(i))

        X = [0] * self.A.n
        for i in range(self.A.n - 1, -1, -1):
            X[i] = (Y[i] - sum(U.entries[i][j] * X[j] for j in range(i + 1, self.A.n))) / U.entries[i][i]

        return Vector(self.A.field, self.A.m, *X)
```