

Aamer Jalan - Python Code

```
1 class Complex:
2     def __init__(self, real, img):
3         self.real = real
4         self.img = img
5
6     def __add__(self, y):
7         return Complex(self.real + y.real, self.img + y.img)
8
9     def __sub__(self, y):
10        return Complex(self.real - y.real, self.img - y.img)
11
12    def __mul__(self, y):
13        return Complex(self.real * y.real - self.img * y.img
14                        , self.real * y.img + self.img * y.real)
15
16    def __truediv__(self, y):
17        denominator = y.real**2 + y.img**2
18        if denominator == 0:
19            raise ValueError("Cannot divide by 0")
20        return Complex((self.real * y.real + self.img * y.
21                        img) / denominator, (self.img * y.real - self.
22                        real * y.img) / denominator)
23
24    def __abs__(self):
25        return (self.real**2 + self.img**2)**0.5
26
27    def conjugate(self):
28        return Complex(self.real, -self.img)
29
30 class Vector:
31     def __init__(self, field, n, *coordinates):
32         if field not in ["real", "complex"]:
33             raise ValueError("Field must be real or complex")
34
35         self.field = field
36         self.n = n
37
38         if len(coordinates) != n:
39             raise ValueError(f"There should be {n}
40                             coordinates")
41
42         if field == "real":
43             for coord in coordinates:
44                 if not isinstance(coord, (int, float)):
45                     raise TypeError("coordinates must be
46                                     real numbers i.e. integers or floats")
```

```

    )
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):

```

```

82     if field not in ["real", "complex"]:
83         raise ValueError("Field must be real or complex"
84                             )
85
86     self.field = field
87
88     if all(isinstance(v, Vector) for v in
89             values_or_vectors):
90         vectors = values_or_vectors
91         m = len(vectors)
92         if m == 0:
93             raise ValueError("At least one vector is
94                             required to form a matrix")
95         n = len(vectors[0].coordinates)
96
97         for vec in vectors:
98             if vec.field != field:
99                 raise TypeError("All vectors must match
100                                the specified field")
101             if len(vec.coordinates) != n:
102                 raise ValueError("All vectors must have
103                                the same length")
104
105         self.entries = [[vec.coordinates[i] for vec in
106                           vectors] for i in range(n)]
107         self.n = n
108         self.m = m
109
110     else:
111         if n is None or m is None:
112             raise ValueError("n and m must be specified"
113                             )
114         if len(values_or_vectors) != n * m:
115             raise ValueError(f"There should be {n * m}
116                             values")
117
118         if field == "real":
119             for val in values_or_vectors:
120                 if not isinstance(val, (int, float)):
121                     raise TypeError("values must be real
122                                     numbers")
123         elif field == "complex":
124             for val in values_or_vectors:
125                 if not isinstance(val, Complex):
126                     raise TypeError("values must be
127                                     complex numbers")
128
129         self.entries = []
130         for i in range(n):

```

```

121         row = list(values_or_vectors[i * m: (i + 1)
122                     * m])
123         self.entries.append(row)
124     self.n = n
125     self.m = m
126
127     def __add__(self, other):
128         if not isinstance(other, Matrix):
129             raise TypeError("Add with Matrix only")
130         if self.field != other.field:
131             raise TypeError("Field mismatch")
132         if self.n != other.n or self.m != other.m:
133             raise ValueError("Dimensions must match")
134
135         result_entries = []
136         for i in range(self.n):
137             row = []
138             for j in range(self.m):
139                 row.append(self.entries[i][j] + other.
140                             entries[i][j])
141             result_entries.extend(row)
142         return Matrix(self.field, self.n, self.m, *
143                       result_entries)
144
145     def __mul__(self, other):
146         if not isinstance(other, Matrix):
147             raise TypeError("Multiply with Matrix only")
148         if self.field != other.field:
149             raise TypeError("Fields do not match")
150         if self.m != other.n:
151             raise ValueError("Dimensions not compatible")
152
153         result_entries = []
154         for i in range(self.n):
155             row = []
156             for j in range(other.m):
157                 sum_product = self.entries[i][0] * other.
158                             entries[0][j]
159                 for k in range(1, self.m):
160                     sum_product += self.entries[i][k] *
161                                 other.entries[k][j]
162                 row.append(sum_product)
163             result_entries.extend(row)
164         return Matrix(self.field, self.n, other.m, *
165                       result_entries)
166
167     def get_row(self, i):
168         if i < 0 or i >= self.n:
169             raise IndexError("Row out of range")

```

```

164         return Matrix(self.field, 1, self.m, *self.entries[i
165                        ])
166     def get_column(self, j):
167         if j < 0 or j >= self.m:
168             raise IndexError("Column out of range")
169         return Matrix(self.field, self.n, 1, *[self.entries[
170            i][j] for i in range(self.n)])
171     def transpose(self):
172         transposed_entries = [self.entries[j][i] for i in
173            range(self.m) for j in range(self.n)]
174         return Matrix(self.field, self.m, self.n, *
175            transposed_entries)
176     def conjugate(self):
177         conjugated_entries = []
178         for row in self.entries:
179             conjugated_entries.extend([elem.conjugate() if
180                isinstance(elem, Complex) else elem for elem
181                in row])
182         return Matrix(self.field, self.n, self.m, *
183            conjugated_entries)
184     def transpose_conjugate(self):
185         return self.transpose().conjugate()
186     def is_zero(self):
187         return all(all(entry == 0 for entry in row) for row
188            in self.entries)
189     def is_symmetric(self):
190         if self.n != self.m:
191             return False
192         return all(self.entries[i][j] == self.entries[j][i]
193            for i in range(self.n) for j in range(i, self.m))
194     def is_hermitian(self):
195         if self.n != self.m or self.field != "complex":
196             return False
197         return all(self.entries[i][j] == self.entries[j][i].
198            conjugate() for i in range(self.n) for j in range
199            (i, self.m))
200     def is_square(self):
201         return self.n == self.m
202     def is_orthogonal(self):
203         if not self.is_square():
204             return False

```

```

203         identity_matrix = self.identity_matrix(self.field,
204         self.n)
205         return self * self.transpose() == identity_matrix
206
207     def identity_matrix(self, field, size):
208         if field not in ["real", "complex"]:
209             raise ValueError("Field must be real or complex"
210             )
211
212         entries = [1 if i == j else 0 for i in range(size)
213         for j in range(size)]
214         return Matrix(field, size, size, *entries)
215
216     def is_unitary(self):
217         if not self.is_square() or self.field != "complex":
218             return False
219         identity_matrix = self.identity_matrix(self.field,
220         self.n)
221         return self.transpose_conjugate() * self ==
222         identity_matrix
223
224     def is_scalar(self):
225         if not self.is_square():
226             return False
227         diagonal_value = self.entries[0][0]
228         return all(self.entries[i][i] == diagonal_value for
229         i in range(self.n)) and \
230         all(self.entries[i][j] == 0 for i in range(self.
231         n) for j in range(self.m) if i != j)
232
233     def rank(self):
234         matrix = [row[:] for row in self.entries]
235         rank = 0
236
237         for col in range(self.m):
238             for row in range(rank, self.n):
239                 if matrix[row][col] != 0:
240                     matrix[rank], matrix[row] = matrix[row],
241                     matrix[rank]
242                     break
243             else:
244                 continue
245
246         for i in range(rank + 1, self.n):
247             if matrix[i][col] != 0:
248                 factor = matrix[i][col] / matrix[rank][
249                 col]
250                 matrix[i] = [matrix[i][j] - factor *
251                 matrix[rank][j] for j in range(self.m
252                 )]
```

```

242         rank += 1
243     return rank
244
245
246 def is_singular(self):
247     if not self.is_square():
248         return False
249     return self.rank() < self.n
250
251 def is_invertible(self):
252     return self.is_square and not self.is_singular()
253
254 def is_identity(self):
255     if not self.is_square():
256         return False
257     return all(self.entries[i][i] == 1 for i in range(
258         self.n)) and \
259         all(self.entries[i][j] == 0 for i in range(self.
260             n) for j in range(self.m) if i != j)
261
262 def is_nilpotent(self):
263     if not self.is_square():
264         return False
265
266     power = self
267     for _ in range(1, self.n + 1):
268         power = power * self
269         if power.is_zero():
270             return True
271     return False
272
273 def is_diagonalizable(self):
274     if not self.is_square():
275         return False
276
277     if self.field == "real" and self.is_symmetric():
278         return True
279
280 def has_lu_decomposition(self):
281     if not self.is_square():
282         return False
283
284     matrix = [row[:] for row in self.entries]
285     n = self.n
286
287     for k in range(n):
288         if matrix[k][k] == 0:
289             return False
290
291     for i in range(k + 1, n):

```

```

290         if matrix[i][k] != 0:
291             factor = matrix[i][k] / matrix[k][k]
292             for j in range(k, n):
293                 matrix[i][j] -= factor * matrix[k][j]
294         ]
295     return True
296
297 def vector_length(self, vector):
298     if vector.field == "real":
299         return sum(coord**2 for coord in vector.
300                     coordinates) ** 0.5
301     elif vector.field == "complex":
302         return sum(abs(coord)**2 for coord in vector.
303                     coordinates) ** 0.5
304
305 def size(self):
306     return self.n, self.m
307
308 def nullity(self):
309     return self.m - self.rank()
310
311 def rref(self, show_steps=False):
312     matrix = [row[:] for row in self.entries]
313     n, m = self.n, self.m
314     row_operations = []
315     elementary_matrices = []
316
317     def create_identity(size):
318         identity = [[1 if i == j else 0 for j in range(
319                     size)] for i in range(size)]
320         return identity
321
322     for i in range(min(n, m)):
323         pivot_row = None
324         for row in range(i, n):
325             if matrix[row][i] != 0:
326                 pivot_row = row
327                 break
328         if pivot_row is None:
329             continue
330
331         if pivot_row != i:
332             matrix[i], matrix[pivot_row] = matrix[
333                 pivot_row], matrix[i]
334             if show_steps:
335                 row_operations.append(f"Swap row {i}
336                                     with row {pivot_row}")
337             elementary_matrices.append(Matrix(self.
338                 field, n, n, *create_identity(n)))

```



```

333     pivot = matrix[i][i]
334     matrix[i] = [val / pivot for val in matrix[i]]
335     if show_steps:
336         row_operations.append(f"Normalize row {i}")
337         elementary_matrices.append(Matrix(self.field
338             , n, n, *create_identity(n)))
339
340     for row in range(n):
341         if row != i and matrix[row][i] != 0:
342             factor = matrix[row][i]
343             matrix[row] = [matrix[row][j] - factor *
344                 matrix[i][j] for j in range(m)]
345             if show_steps:
346                 row_operations.append(f"Eliminate
347                     row {row} using row {i}")
348                 elementary_matrices.append(Matrix(
349                     self.field, n, n, *
350                     create_identity(n)))
351
352     if show_steps:
353         return Matrix(self.field, n, m, *[elem for row
354             in matrix for elem in row]), row_operations,
355             elementary_matrices
356     return Matrix(self.field, n, m, *[elem for row in
357         matrix for elem in row])
358
359 def are_linearly_independent(self, vectors):
360     matrix = Matrix(self.field, len(vectors[0].
361         coordinates), len(vectors), *[v.coordinates[i]
362             for v in vectors for i in range(v.n)])
363     return matrix.rank() == len(vectors)
364
365 def dimension_of_span(self, vectors):
366     matrix = Matrix(self.field, len(vectors[0].
367         coordinates), len(vectors), *[v.coordinates[i]
368             for v in vectors for i in range(v.n)])
369     return matrix.rank()
370
371 def basis_for_span(self, vectors):
372     rref_matrix = self.rref()
373     basis_vectors = []
374     for i in range(rref_matrix.n):
375         if any(rref_matrix.entries[i][j] != 0 for j in
376             range(rref_matrix.m)):
377             basis_vectors.append(Vector(self.field,
378                 rref_matrix.m, *rref_matrix.entries[i]))
379     return basis_vectors
380
381 def rank_factorization(self):
382     if not self.is_square():

```

```

369         raise ValueError("Rank factorization is only
370                             defined for square matrices")
371
372     U = self.rref()
373     non_zero_rows = [row for row in U.entries if any(val
374                                                         != 0 for val in row)]
375     R = Matrix(self.field, len(non_zero_rows), self.m,
376               *[val for row in non_zero_rows for val in row])
377     C = Matrix(self.field, self.n, len(non_zero_rows),
378               *[self.get_column(i).entries for i in range(len(
379                   non_zero_rows))])
380     return R, C
381
382 def lu_decompose(self):
383     if not self.is_square():
384         raise ValueError("LU decomposition is only
385                             defined for square matrices")
386
387     if not self.has_lu_decomposition():
388         raise ValueError("LU decomposition is not
389                             possible due to a zero pivot")
390
391     L = [[0] * self.n for _ in range(self.n)]
392     U = [[0] * self.n for _ in range(self.n)]
393     matrix = [row[:] for row in self.entries]
394
395     for i in range(self.n):
396         for j in range(i, self.n):
397             U[i][j] = matrix[i][j] - sum(L[i][k] * U[k][
398                 j] for k in range(i))
399
400         for j in range(i, self.n):
401             if i == j:
402                 L[i][i] = 1
403             else:
404                 L[j][i] = (matrix[j][i] - sum(L[j][k] *
405                     U[k][i] for k in range(i))) / U[i][i]
406
407     L_matrix = Matrix(self.field, self.n, self.n, *[val
408                                                         for row in L for val in row])
409     U_matrix = Matrix(self.field, self.n, self.n, *[val
410                                                         for row in U for val in row])
411
412     return L_matrix, U_matrix
413
414 def plu_decompose(self):
415     if not self.is_square():
416         raise ValueError("PLU decomposition is only
417                             defined for square matrices")

```

```

407     n = self.n
408     P = [[1 if i == j else 0 for j in range(n)] for i in
           range(n)]
409     matrix = [row[:] for row in self.entries]
410
411     for i in range(n):
412         max_row = max(range(i, n), key=lambda r: abs(
            matrix[r][i]))
413         if matrix[max_row][i] == 0:
414             raise ValueError("Matrix is singular and
                                cannot be decomposed")
415
416         P[i], P[max_row] = P[max_row], P[i]
417
418         matrix[i], matrix[max_row] = matrix[max_row],
            matrix[i]
419
420     permuted_matrix = Matrix(self.field, n, n, *[val for
            row in matrix for val in row])
421
422     L, U = permuted_matrix.lu_decompose()
423
424     P_matrix = Matrix(self.field, n, n, *[val for row in
            P for val in row])
425
426     return P_matrix, L, U
427
428 def inverse_by_row_reduction(self):
429     if not self.is_square():
430         raise ValueError("Only square matrices have an
                                inverse")
431     if not self.is_invertible():
432         raise ValueError("Matrix is not invertible")
433
434     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)])
435     augmented = Matrix(self.field, self.n, self.n * 2,
436         *[val for row in self.entries for
            val in row] +
437         [val for row in identity.entries for
            val in row])
438
439     rref_augmented = augmented.rref()
440
441     inverse_entries = [rref_augmented.entries[i][self.n
            :]] for i in range(self.n)]
442     inverse_flat = [val for row in inverse_entries for
            val in row]
443

```

```

444         return Matrix(self.field, self.n, self.n, *
445                        inverse_flat)
446
447     def inverse_by_adjoint(self):
448         if not self.is_square():
449             raise ValueError("Only square matrices have an
450                               inverse")
451         if not self.is_invertible():
452             raise ValueError("Matrix is not invertible")
453
454         cofactor_entries = []
455         for i in range(self.n):
456             for j in range(self.n):
457                 minor_matrix = self.minor_matrix(i, j)
458                 cofactor = ((-1) ** (i + j)) * minor_matrix.
459                     determinant_by_rref()
460                 cofactor_entries.append(cofactor)
461
462         adjugate = Matrix(self.field, self.n, self.n, *
463                          cofactor_entries).transpose()
464
465         identity = Matrix(self.field, self.n, self.n, *[1 if
466                 i == j else 0 for i in range(self.n) for j in
467                 range(self.n)])
468         augmented = Matrix(self.field, self.n, self.n * 2,
469                             *[val for row in self.entries for
470                                 val in row] +
471                             [val for row in adjugate.entries for
472                                 val in row])
473         rref_augmented = augmented.rref()
474
475         inverse_entries = [rref_augmented.entries[i][self.n
476                 :]] for i in range(self.n)
477         inverse_flat = [val for row in inverse_entries for
478                         val in row]
479
480         return Matrix(self.field, self.n, self.n, *
481                        inverse_flat)
482
483     def determinant_by_rref(self):
484         if not self.is_square():
485             raise ValueError("Determinant is only defined
486                               for square matrices")
487
488         matrix = [row[:] for row in self.entries]
489         n = self.n
490         determinant = 1
491
492         for i in range(n):
493             pivot_row = None

```

```

482         for j in range(i, n):
483             if matrix[j][i] != 0:
484                 pivot_row = j
485                 break
486         if pivot_row is None:
487             return 0
488
489         if pivot_row != i:
490             matrix[i], matrix[pivot_row] = matrix[
491                 pivot_row], matrix[i]
492             determinant *= -1
493
494         pivot = matrix[i][i]
495         determinant *= pivot
496         matrix[i] = [val / pivot for val in matrix[i]]
497
498         for j in range(i + 1, n):
499             factor = matrix[j][i]
500             matrix[j] = [matrix[j][k] - factor * matrix[
501                 i][k] for k in range(n)]
502
503         return determinant
504
505     def minor_matrix(self, row, col):
506         if row < 0 or row >= self.n or col < 0 or col >=
507             self.m:
508             raise IndexError("Row or column index out of
509                 range")
510         minor_entries = [
511             [self.entries[i][j] for j in range(self.m) if j
512                 != col]
513             for i in range(self.n) if i != row
514         ]
515         return Matrix(self.field, self.n - 1, self.m - 1, *[
516             val for sublist in minor_entries for val in
517                 sublist])
518
519     def is_in_span(self, S, v):
520         if not isinstance(v, Vector):
521             raise TypeError("v must be a Vector")
522         if not all(isinstance(vec, Vector) for vec in S):
523             raise TypeError("S must contain Vector objects")
524         if any(vec.field != self.field for vec in S) or v.
525             field != self.field:
526             raise ValueError("All vectors must belong to the
527                 same field")
528         if any(len(vec.coordinates) != v.n for vec in S):
529             raise ValueError("All vectors must have the same
530                 dimension as v")

```

```

522     matrix_S = Matrix(self.field, S[0].n, len(S), *[vec.
        coordinates[i] for vec in S for i in range(vec.n)
        ])
523
524     coefficients, representation = matrix_S.
        linear_combination(v)
525     return representation != "0"
526
527 def linear_combination(self, S, v):
528     if not self.is_in_span(S, v):
529         raise ValueError("v is not in the span of S")
530
531     matrix_S = Matrix(self.field, S[0].n, len(S), *[vec.
        coordinates[i] for vec in S for i in range(vec.n)
        ])
532
533     augmented = Matrix(self.field, v.n, len(S) + 1,
534         *[elem for row in matrix_S.entries
            for elem in row] + v.coordinates)
535     rref_matrix = augmented.rref()
536
537     coefficients = [rref_matrix.entries[i][-1] for i in
        range(len(S))]
538     representation = " + ".join(f"({coeff})*{S[i].
        coordinates}" for i, coeff in enumerate(
        coefficients))
539
540     return coefficients, representation
541
542 def do_sets_span_the_same_space(self, S1, S2):
543     if not all(isinstance(v, Vector) and v.field == self
        .field for v in S1 + S2):
544         raise ValueError("All vectors must be instances
        of Vector and belong to the same field")
545
546     for v in S2:
547         if not self.is_in_span(S1, v):
548             return False
549
550     for v in S1:
551         if not self.is_in_span(S2, v):
552             return False
553
554     return True
555
556 def compute_coordinates(self, B, v):
557     if not self.is_in_span(B, v):
558         raise ValueError("v is not in the span of B")
559

```

```

560         coefficients, representation = self.
561             linear_combination(B, v)
562         return coefficients, representation
563
564     def vector_from_coordinates(self, B, coordinates):
565         if len(B) != len(coordinates):
566             raise ValueError("Number of basis vectors must
567                 match number of coordinates")
568         if any(b.field != self.field for b in B):
569             raise ValueError("All basis vectors must belong
570                 to the same field as the matrix")
571
572         reconstructed = [0] * B[0].n
573         for coeff, basis_vector in zip(coordinates, B):
574             for i in range(len(reconstructed)):
575                 reconstructed[i] += coeff * basis_vector.
576                     coordinates[i]
577
578         return Vector(self.field, len(reconstructed), *
579             reconstructed)
580
581     def change_of_basis_matrix(self, B1, B2):
582         if len(B1) != len(B2):
583             raise ValueError("B1 and B2 must have the same
584                 number of vectors")
585
586         matrix_B1 = Matrix(self.field, B1[0].n, len(B1), *[b
587             .coordinates[i] for b in B1 for i in range(b.n)])
588         matrix_B2 = Matrix(self.field, B2[0].n, len(B2), *[b
589             .coordinates[i] for b in B2 for i in range(b.n)])
590
591         matrix_B1_inv = matrix_B1.inverse_by_row_reduction()
592         return matrix_B2 * matrix_B1_inv
593
594     def coordinates_in_new_basis(self, B1, B2,
595         coordinates_B1):
596         change_matrix = self.change_of_basis_matrix(B1, B2)
597         coord_vector_B1 = Vector(self.field, len(
598             coordinates_B1), *coordinates_B1)
599
600         result_matrix = change_matrix * Matrix(self.field,
601             len(coordinates_B1), 1, *coord_vector_B1.
602                 coordinates)
603         return [result_matrix.entries[i][0] for i in range(
604             result_matrix.n)]
605
606     def determinant_by_cofactor(self):
607         if not self.is_square():
608             raise ValueError("Determinant is only defined
609                 for square matrices")

```

```

596         if self.n == 1:
597             return self.entries[0][0]
598         determinant = 0
599         for col in range(self.m):
600             minor = self.minor_matrix(0, col)
601             cofactor = ((-1) ** col) * minor.
602                 determinant_by_cofactor()
603             determinant += self.entries[0][col] * cofactor
604         return determinant
605
606     def determinant_by_plu(self):
607         if not self.is_square():
608             raise ValueError("Determinant is only defined
609                 for square matrices")
610         P, _, U = self.plu_decompose()
611         determinant = 1
612         for i in range(self.n):
613             determinant *= U.entries[i][i]
614         return determinant * (-1) ** (sum(row.index(1) for
615             row in P.entries))
616
617     def determinant_by_rref_method(self):
618         if not self.is_square():
619             raise ValueError("Determinant is only defined
620                 for square matrices")
621         return self.determinant_by_rref()
622
623     def qr_factorization(matrix):
624         Q = gram_schmidt([Vector(matrix.field, matrix.n, *
625             matrix.get_column(i).coordinates) for i in range(
626                 matrix.m)])
627         R = Matrix(matrix.field, len(Q), len(Q))
628         for i, q in enumerate(Q):
629             for j in range(i, len(Q)):
630                 R.entries[i][j] = inner_product(q, Vector(
631                     matrix.field, matrix.n, *matrix.
632                         get_column(j).coordinates))
633         return Matrix(matrix.field, len(Q), matrix.n, *[elem
634             for q in Q for elem in q.coordinates]), R
635
636     def pseudoinverse(self):
637         Q, R = self.qr_factorization()
638         return R.inverse_by_row_reduction() * Q.
639             transpose_conjugate()
640
641     def least_squares_solution(self, b):
642         pseudo_inv = self.pseudoinverse()
643         return pseudo_inv * b
644
645     def cholesky_decomposition(self):

```



```

636         if not self.is_square() or not self.is_hermitian():
637             raise ValueError("Cholesky decomposition
                                requires a square, Hermitian positive-
                                definite matrix")
638         L = [[0] * self.n for _ in range(self.n)]
639         for i in range(self.n):
640             for j in range(i + 1):
641                 if i == j:
642                     L[i][j] = (self.entries[i][i] - sum(L[i]
                                                             ][k] ** 2 for k in range(j))) ** 0.5
643                 else:
644                     L[i][j] = (self.entries[i][j] - sum(L[i]
                                                             ][k] * L[j][k] for k in range(j))) /
645                             L[j][j]
646         return Matrix(self.field, self.n, self.n, *[val for
647             row in L for val in row])
648
649 class SystemOfEquations:
650     def __init__(self, A, b):
651         if not isinstance(A, Matrix):
652             raise TypeError("A must be a Matrix")
653         if not isinstance(b, Vector):
654             raise TypeError("b must be a Vector")
655         if A.n != b.n:
656             raise ValueError("Matrix A and vector b
657                 dimensions do not match")
658         self.A = A
659         self.b = b
660
661     def is_consistent(self):
662         augmented_matrix = Matrix(self.A.field, self.A.n,
663             self.A.m + 1,
664             *[elem for row in self.A.
665                 entries for elem in row]
666             + self.b.coordinates)
667         return self.A.rank() == augmented_matrix.rank()
668
669     def solve(self):
670         if not self.is_consistent():
671             raise ValueError("System is inconsistent and has
672                 no solution")
673
674         augmented_matrix = Matrix(self.A.field, self.A.n,
675             self.A.m + 1,
676             *[elem for row in self.A.
677                 entries for elem in row]
678             + self.b.coordinates)
679         rref_matrix = augmented_matrix.rref()
680
681         solution = []

```

```

672     for i in range(self.A.m):
673         if i < rref_matrix.n and rref_matrix.entries[i][
            i] != 0:
674             solution.append(rref_matrix.entries[i][-1])
675         else:
676             solution.append(0)
677
678     return Vector(self.A.field, len(solution), *solution
        )
679
680 def is_subspace(S1, S2):
681     combined = S2 + S1
682     matrix = Matrix(S1[0].field, len(combined[0].
        coordinates), len(combined),
683         *[v.coordinates[i] for v in combined
            for i in range(v.n)])
684     return matrix.rank() == len(S2)
685
686 def solution_set(self):
687     if not self.is_consistent():
688         raise ValueError("System is inconsistent and has
            no solution")
689
690     augmented_matrix = Matrix(self.A.field, self.A.n,
        self.A.m + 1,
691         *[elem for row in self.A.
            entries for elem in row]
            + self.b.coordinates)
692     rref_matrix = augmented_matrix.rref()
693
694     free_vars = []
695     basic_vars = []
696     for i in range(self.A.m):
697         if i < rref_matrix.n and rref_matrix.entries[i][
            i] != 0:
698             basic_vars.append(i)
699         else:
700             free_vars.append(i)
701
702     solutions = {}
703     for var in basic_vars:
704         solutions[f"x{var}"] = rref_matrix.entries[var
            ][-1]
705     for var in free_vars:
706         solutions[f"x{var}"] = "Free"
707
708     return solutions
709
710 def solve_with_plu(self):
711     if not self.A.is_square():

```

```

712         raise ValueError("PLU decomposition requires a
713                             square matrix")
714     if not self.is_consistent():
715         raise ValueError("System is inconsistent and has
716                             no solution")
717
718     P, L, U = self.A.plu_decompose()
719     Pb = P * Matrix(self.A.field, self.b.n, 1, *self.b.
720                     coordinates)
721
722     Y = [0] * self.A.n
723     for i in range(self.A.n):
724         Y[i] = Pb.entries[i][0] - sum(L.entries[i][j] *
725                                     Y[j] for j in range(i))
726
727     X = [0] * self.A.n
728     for i in range(self.A.n - 1, -1, -1):
729         X[i] = (Y[i] - sum(U.entries[i][j] * X[j] for j
730                           in range(i + 1, self.A.n))) / U.entries[i][i]
731
732     return Vector(self.A.field, self.A.m, *X)

```