13. Linear Algebra for Engineers

Phase 5: Modern Software Math
⏱ ~50 minutes | 🔢 Foundational for ML/Graphics/Search | 📐 Geometric Intuition

---

What Problem This Solves

You're encountering:

- Machine learning tutorials that throw around "matrix multiplication" and "dot products"
- 3D graphics transformations (rotation, scaling, translation)
- Recommendation engines ("compute similarity between users")
- Database query optimization ("vector operations")
- Principal Component Analysis (PCA) for data compression
- Graph algorithms ("eigenvector centrality")

Without linear algebra, these concepts are black boxes. You copy-paste TensorFlow code, tweak CSS transforms blindly, or avoid ML entirely because "it's too math-heavy."

With linear algebra intuition, you understand why ML works, how graphics transformations compose, what "embedding spaces" really are, and when vector operations make sense for your problem.

Intuition & Mental Model

The Core Insight: Vectors as Arrows & Data Rows

Two Ways to Think About Vectors:

```
GEOMETRIC VIEW:              DATA VIEW:
    ↑                       user = [age, income, clicks]
   /                        product = [price, rating, sales]
  /  → Vector as arrow      row in a database table
 /    (direction + magnitude)
•
```

Mental Model: Vectors = Lists of Numbers with Meaning

```
// A vector is just an array with structure
const user = [28, 75000, 142];  // Age, income, clicks
const product = [49.99, 4.5, 1523];  // Price, rating, sales

// Operations on vectors = operations on features
```

Matrices: Transformations & Relationships

```
Matrix = A grid of numbers
       = A transformation (rotate, scale, skew)
       = A relationship map (user-item interactions)
```

```
[a b]   ← row 1
[c d]   ← row 2
```

The "Why" of Linear Algebra: Almost everything in computers is lists of numbers. Linear algebra is the math of transforming, comparing, and combining these lists efficiently.

Core Concepts

1. Vectors: The Building Block

```
// Vector as array
class Vector {
  constructor(components) {
    this.components = components;
  }

  // Magnitude (length)
  magnitude() {
    return Math.sqrt(
      this.components.reduce((sum, val) => sum + val * val, 0)
    );
  }

  // Add two vectors
  add(other) {
    return new Vector(
      this.components.map((val, i) => val + other.components[i])
    );
  }

  // Scalar multiplication
  scale(scalar) {
    return new Vector(
      this.components.map(val => val * scalar)
    );
  }

  // Dot product
  dot(other) {
    return this.components.reduce(
      (sum, val, i) => sum + val * other.components[i],
      0
    );
  }
}

// Usage
const v1 = new Vector([3, 4]);
console.log(v1.magnitude());  // 5 (Pythagorean theorem: √(3² + 4²))

const v2 = new Vector([1, 2]);
```

```
console.log(v1.add(v2).components);  // [4, 6]

console.log(v1.dot(v2));  // 3×1 + 4×2 = 11
```

Geometric Intuition:

```
Vector [3, 4]:
  ↑ 4
  |  /
  | /  → 3
  |/
  •

Magnitude = Length of arrow = √(3² + 4²) = 5
```

2. Dot Product: Similarity & Projection

What It Means:

```
v1 · v2 = |v1| × |v2| × cos(θ)

θ = angle between vectors

If θ = 0°   → cos = 1  → Vectors aligned (same direction)
If θ = 90°  → cos = 0  → Vectors perpendicular (orthogonal)
If θ = 180° → cos = -1 → Vectors opposite
```

Software Use Case: Measuring Similarity

```
function cosineSimilarity(vec1, vec2) {
  const v1 = new Vector(vec1);
  const v2 = new Vector(vec2);

  const dotProduct = v1.dot(v2);
  const magnitudes = v1.magnitude() * v2.magnitude();

  return dotProduct / magnitudes;
}

// User preferences: [likes sci-fi, likes romance, likes action]
const user1 = [5, 1, 4];  // Loves sci-fi & action
const user2 = [4, 2, 5];  // Similar taste
const user3 = [1, 5, 1];  // Loves romance (different)

cosineSimilarity(user1, user2);  // 0.98 (very similar)
cosineSimilarity(user1, user3);  // 0.29 (dissimilar)

// This is how recommendation engines work!
```

Real Example: Document Similarity

```
function documentSimilarity(doc1, doc2, vocabulary) {
  // Convert docs to "bag of words" vectors
  const vec1 = vocabulary.map(word =>
    (doc1.match(new RegExp(word, 'gi')) || []).length
  );
  const vec2 = vocabulary.map(word =>
    (doc2.match(new RegExp(word, 'gi')) || []).length
  );

  return cosineSimilarity(vec1, vec2);
}

const vocab = ['react', 'vue', 'angular', 'frontend', 'backend'];
const docA = 'react frontend react components';
const docB = 'vue frontend vue components';
const docC = 'backend node express';

documentSimilarity(docA, docB, vocab);  // ~0.87 (similar: both frontend frameworks)
documentSimilarity(docA, docC, vocab);  // ~0.31 (different: frontend vs backend)
```

3. Matrices: Transformations & Data Tables

Matrix as Data Table:

```
// User-item ratings matrix
const ratings = [
  //  Item1  Item2  Item3
  [5,    1,     4],    // User 1
  [4,    0,     5],    // User 2
  [0,    5,     2]     // User 3
];

// Access: ratings[userIndex][itemIndex]
```

Matrix as Transformation:

```
function transformPoint(matrix, point) {
  // Matrix × Vector = Transformed Vector
  return matrix.map(row =>
    row.reduce((sum, val, i) => sum + val * point[i], 0)
  );
}

// Rotation matrix (90° counterclockwise)
const rotation90 = [
  [0, -1],
  [1,  0]
];

const point = [3, 0];  // Point at (3, 0)
```

```
const rotated = transformPoint(rotation90, point);
console.log(rotated);  // [0, 3] - now pointing up!

/*
Visual:
Before: (3, 0) →          After: (0, 3) ↑
                                        |
                                        |
*/
```

Scaling Matrix:

```
// Scale by 2x horizontally, 3x vertically
const scaleMatrix = [
  [2, 0],
  [0, 3]
];

transformPoint(scaleMatrix, [1, 1]);  // [2, 3]
transformPoint(scaleMatrix, [2, 1]);  // [4, 3]

// This is how CSS transform: scale(2, 3) works!
```

4. Matrix Multiplication: Composing Transformations

```
function matrixMultiply(A, B) {
  const rows = A.length;
  const cols = B[0].length;
  const inner = B.length;

  const result = Array(rows).fill().map(() => Array(cols).fill(0));

  for (let i = 0; i < rows; i++) {
    for (let j = 0; j < cols; j++) {
      for (let k = 0; k < inner; k++) {
        result[i][j] += A[i][k] * B[k][j];
      }
    }
  }

  return result;
}

// Compose transformations: First scale, then rotate
const scale = [[2, 0], [0, 2]];  // 2x scaling
const rotate = [[0, -1], [1, 0]];  // 90° rotation

const combined = matrixMultiply(rotate, scale);
// Apply both transformations at once
```

```
transformPoint(combined, [1, 0]);  // [0, 2]
// First scaled to (2, 0), then rotated to (0, 2)
```

Why This Matters:

```
In graphics: Chaining transforms (translate, rotate, scale)
In ML: Layer compositions (neural networks)
In databases: Join operations
```

5. Identity Matrix & Inverse

Identity Matrix = "Do nothing" transformation

```
const identity = [
  [1, 0, 0],
  [0, 1, 0],
  [0, 0, 1]
];

// Any matrix × Identity = Same matrix
// Any vector × Identity = Same vector
transformPoint([[1, 0], [0, 1]], [3, 4]);  // [3, 4] - unchanged
```

Inverse Matrix = "Undo" transformation

```
// Rotation 90° clockwise
const rotateCW = [[0, 1], [-1, 0]];

// Inverse = Rotation 90° counterclockwise
const rotateCCW = [[0, -1], [1, 0]];

// Rotating clockwise then counterclockwise = Identity
const shouldBeIdentity = matrixMultiply(rotateCW, rotateCCW);
// [[1, 0], [0, 1]] ✓

// Use case: Undoing transformations in graphics/game engines
```

6. Eigenvalues & Eigenvectors (Intuition)

Question: Are there special vectors that a matrix only *scales*, not *rotates*?

```
function demonstrateEigenvector() {
  // Matrix that scales x by 2, y by 3
  const matrix = [[2, 0], [0, 3]];

  const xAxis = [1, 0];  // Along x-axis
  const yAxis = [0, 1];  // Along y-axis

  const transformedX = transformPoint(matrix, xAxis);  // [2, 0] - scaled by 2
  const transformedY = transformPoint(matrix, yAxis);  // [0, 3] - scaled by 3
```

```
  return {
    xEigenvector: xAxis,
    xEigenvalue: 2,  // Scaling factor
    yEigenvector: yAxis,
    yEigenvalue: 3
  };
}

// Eigenvectors = "special directions" unchanged by transformation (except scaling)
// Eigenvalues = How much they're scaled
```

Real-World Use Case: PageRank

```
// Simplified PageRank: Eigenvector of link matrix
function pageRank(linkMatrix, iterations = 20) {
  const n = linkMatrix.length;
  let ranks = Array(n).fill(1 / n);  // Start with equal ranks

  for (let iter = 0; iter < iterations; iter++) {
    const newRanks = Array(n).fill(0);

    for (let i = 0; i < n; i++) {
      for (let j = 0; j < n; j++) {
        newRanks[i] += linkMatrix[i][j] * ranks[j];
      }
    }

    ranks = newRanks;
  }

  return ranks;
}

// Link matrix: links[i][j] = probability of going from page j to page i
const links = [
  [0,    0.5,  0.5],  // Page 0 links to pages 1, 2
  [0.33, 0,    0.5],  // Page 1 links to pages 0, 2
  [0.33, 0.5,  0]     // Page 2 links to pages 0, 1
];

pageRank(links);
// Converges to eigenvector = "steady-state" page importance
// [0.36, 0.32, 0.32] - Page 0 most important
```

Software Engineering Connections

1. Recommendation Systems

```
class CollaborativeFiltering {
  constructor(ratingsMatrix) {
    // ratingsMatrix[user][item] = rating (0 if not rated)
    this.ratings = ratingsMatrix;
    this.users = ratingsMatrix.length;
    this.items = ratingsMatrix[0].length;
  }

  // Find similar users using cosine similarity
  similarUsers(userIndex) {
    const userVector = this.ratings[userIndex];

    const similarities = this.ratings.map((otherUser, i) => {
      if (i === userIndex) return { userIndex: i, similarity: 0 };

      const similarity = cosineSimilarity(userVector, otherUser);
      return { userIndex: i, similarity };
    });

    return similarities.sort((a, b) => b.similarity - a.similarity);
  }

  // Recommend items based on similar users
  recommendItems(userIndex, topN = 3) {
    const similarUsers = this.similarUsers(userIndex).slice(0, 5);  // Top 5 similar
    const scores = Array(this.items).fill(0);

    // Weighted average of similar users' ratings
    similarUsers.forEach(({ userIndex: simUser, similarity }) => {
      this.ratings[simUser].forEach((rating, itemIndex) => {
        if (this.ratings[userIndex][itemIndex] === 0 && rating > 0) {
          scores[itemIndex] += rating * similarity;
        }
      });
    });

    return scores
      .map((score, itemIndex) => ({ itemIndex, score }))
      .sort((a, b) => b.score - a.score)
      .slice(0, topN);
  }
}

// Example: Movie ratings
const movieRatings = [
  //  M1  M2  M3  M4  M5
  [5,  4,  0,  0,  1],  // User 0: Likes M1, M2, dislikes M5
  [5,  5,  0,  0,  2],  // User 1: Similar to User 0
  [0,  0,  5,  5,  5],  // User 2: Different taste
  [4,  4,  0,  0,  0]   // User 3: Similar to Users 0, 1
];
```

```
const cf = new CollaborativeFiltering(movieRatings);

// Recommend movies for User 0
cf.recommendItems(0);
/* [
  { itemIndex: 3, score: 5.98 },  // M4 - similar users liked it
  { itemIndex: 2, score: 5.00 },  // M3
  { itemIndex: 4, score: 2.95 }   // M5 (low score - they disliked similar movies)
]
// Linear algebra powers Netflix, Spotify, Amazon recommendations! */
```

2. Image Transformations (CSS/Canvas)

```
function applyTransformation(image, transformMatrix) {
  // Image = array of [x, y, r, g, b] pixels
  return image.map(pixel => {
    const [x, y, r, g, b] = pixel;
    const [newX, newY] = transformPoint(transformMatrix, [x, y]);
    return [newX, newY, r, g, b];
  });
}

// Transformation matrices
const transforms = {
  rotate45: [
    [Math.cos(Math.PI/4), -Math.sin(Math.PI/4)],
    [Math.sin(Math.PI/4),  Math.cos(Math.PI/4)]
  ],
  flipHorizontal: [
    [-1, 0],
    [0,  1]
  ],
  shear: [
    [1, 0.5],  // Shear: x' = x + 0.5y
    [0, 1]
  ]
};

// This is exactly what CSS transform does behind the scenes
// transform: rotate(45deg) → multiply by rotation matrix
// transform: scaleX(-1) → multiply by flip matrix
```

3. Machine Learning: Neural Network Layer

```
function neuralNetworkLayer(input, weights, biases, activation) {
  // input: [x1, x2, ..., xn]
  // weights: matrix [[w11, w12, ...], [w21, w22, ...], ...]
  // biases: [b1, b2, ...]
```

```
  // activation: function (ReLU, sigmoid, etc.)

  // Matrix multiplication: weights × input
  const weighted = weights.map((row, i) => {
    const sum = row.reduce((acc, weight, j) => acc + weight * input[j], 0);
    return sum + biases[i];
  });

  // Apply activation function
  return weighted.map(activation);
}

// ReLU activation
const relu = x => Math.max(0, x);

// Simple 2-layer network
const input = [1.0, 0.5];
const layer1Weights = [[0.2, 0.8], [0.5, 0.1], [0.9, 0.3]];
const layer1Biases = [0.1, 0.2, 0.3];

const hidden = neuralNetworkLayer(input, layer1Weights, layer1Biases, relu);
console.log(hidden);  // Output of first layer

// All of deep learning = chained matrix multiplications + nonlinearities
// Linear algebra enables GPUs to train massive networks efficiently
```

4. Search: TF-IDF & Vector Space Model

```
function tfidf(documents) {
  // Term Frequency-Inverse Document Frequency
  // Converts documents into vectors for similarity search

  const vocabulary = new Set();
  documents.forEach(doc =>
    doc.toLowerCase().split(/\s+/).forEach(word => vocabulary.add(word))
  );

  const vocabArray = Array.from(vocabulary);
  const docCount = documents.length;

  // Document frequency: how many docs contain each word
  const df = vocabArray.map(word =>
    documents.filter(doc => doc.toLowerCase().includes(word)).length
  );

  // Convert each doc to TF-IDF vector
  const vectors = documents.map(doc => {
    const words = doc.toLowerCase().split(/\s+/);
    const tf = vocabArray.map(word =>
      words.filter(w => w === word).length / words.length
    );
```

```
    return tf.map((termFreq, i) =>
      termFreq * Math.log(docCount / (df[i] || 1))  // TF × IDF
    );
  });

  return { vocabulary: vocabArray, vectors };
}

const docs = [
  'react frontend components props',
  'vue frontend components reactive',
  'backend nodejs express api',
  'react hooks useState useEffect'
];

const { vocabulary, vectors } = tfidf(docs);

// Search: Query = "react components"
const query = 'react components';
const queryVector = vocabulary.map(word =>
  query.includes(word) ? 1 : 0
);

// Find most similar document
const similarities = vectors.map((docVec, i) => ({
  docIndex: i,
  similarity: cosineSimilarity(queryVector, docVec)
})).sort((a, b) => b.similarity - a.similarity);

console.log(similarities[0]);  // Most relevant document
// This is how search engines rank results!
```

5. Principal Component Analysis (PCA) - Dimensionality Reduction

```
function simplePCA(data, components = 2) {
  // Reduce high-dimensional data to lower dimensions
  // Use case: Visualizing high-dimensional user data, compressing features

  const n = data.length;
  const dims = data[0].length;

  // Center the data (subtract mean)
  const means = Array(dims).fill(0);
  data.forEach(row => row.forEach((val, i) => means[i] += val / n));

  const centered = data.map(row =>
    row.map((val, i) => val - means[i])
  );

  // In real PCA: compute covariance matrix, find eigenvectors
```

```
  // Here: simplified projection onto first 2 dimensions (demo only)
  return centered.map(row => row.slice(0, components));
}

// Example: User features (100+ dimensions) → 2D for visualization
const highDimUsers = [
  [5, 2, 8, 1, 9, 3, 7, 4, 6, 2],  // User 1
  [5, 3, 8, 2, 9, 4, 7, 3, 6, 1],  // User 2 (similar)
  [1, 8, 2, 9, 1, 8, 3, 9, 2, 8]   // User 3 (different)
];

simplePCA(highDimUsers, 2);
// Reduces to 2D while preserving most variance
// Enables 2D scatter plot visualization of user clusters
```

Common Misconceptions

❌ "Linear algebra is just for graphics and ML"

Wrong: It's everywhere in software:

- **Databases**: Query optimization (matrix operations on tables)
- **Search**: Ranking algorithms (PageRank = eigenvector)
- **Compression**: JPEG, MP3 (frequency transforms = matrix operations)
- **Networks**: Graph centrality (adjacency matrix analysis)
- **Cryptography**: Encryption algorithms (matrix transformations)

❌ "Matrices are just 2D arrays"

They represent:

- Transformations (graphics)
- Relationships (user-item, graph adjacency)
- Systems of equations
- Data tables (but with mathematical operations)

```
// Matrix as relationship
const friends = [
  //  Alice  Bob  Carol
  [0,     1,   1],     // Alice friends with Bob, Carol
  [1,     0,   0],     // Bob friends with Alice
  [1,     0,   0]      // Carol friends with Alice
];

// Matrix multiplication = "friends of friends"
const friendsOfFriends = matrixMultiply(friends, friends);
// Shows 2-hop connections (mutual friends)
```

❌ "Dot product is just multiplying numbers"

It measures:

- **Angle** between vectors (cosine similarity)
- **Projection** (how much of one vector lies along another)
- **Work** in physics (force · displacement)
- **Correlation** (how aligned are two variables)

```
// High dot product = vectors point same direction = similar
const vec1 = [1, 0];
const vec2 = [1, 0];
new Vector(vec1).dot(new Vector(vec2));  // 1 (perfectly aligned)

// Zero dot product = perpendicular = uncorrelated
const vec3 = [0, 1];
new Vector(vec1).dot(new Vector(vec3));  // 0 (orthogonal)
```

❌ "Linear algebra requires manual calculations"

Libraries do the heavy lifting:

```
// In practice, use libraries
// JavaScript: math.js, ml-matrix
// Python: NumPy, scipy
// Just understand the concepts!

const math = require('mathjs');

const A = math.matrix([[1, 2], [3, 4]]);
const B = math.matrix([[5, 6], [7, 8]]);

math.multiply(A, B);  // Matrix multiplication
math.inv(A);          // Inverse
math.eigs(A);         // Eigenvalues/eigenvectors

// You don't need to implement from scratch (unless learning)
```

Practical Mini-Exercises

▶ **Exercise 1: User Similarity** (Click to expand)
▶ **Exercise 2: Image Rotation** (Click to expand)
▶ **Exercise 3: Document Search** (Click to expand)

Summary Cheat Sheet

```
// VECTORS
- Array of numbers: [x, y, z]
- Magnitude: √(x² + y² + z²)
- Addition: [a, b] + [c, d] = [a+c, b+d]
- Scalar: k × [a, b] = [k×a, k×b]

// DOT PRODUCT
```

```
v1 · v2 = Σ(v1[i] × v2[i])
        = |v1| × |v2| × cos(θ)
Use: Measuring similarity, projection

// MATRICES
- Grid of numbers
- Represent: transformations, relationships, data tables
- Multiplication: Composition of transformations

// KEY OPERATIONS
- Matrix × Vector = Transformed vector
- Matrix × Matrix = Combined transformation
- Identity matrix = "Do nothing"
- Inverse matrix = "Undo"
- Eigenvalue/Eigenvector = "Special directions"
```

When to Use Linear Algebra:

- Comparing high-dimensional data → Dot product/cosine similarity
- Transforming graphics/geometry → Matrix transformations
- Recommendation systems → Matrix factorization
- ML model layers → Matrix multiplication
- Search/ranking → Vector space models, PageRank

Next Steps

✅ **You've completed**: Linear algebra foundations
➡️ **Up next**: 14. Optimization - What optimization means, constraints, local/global optima

Before moving on, implement:

```
// Challenge: Build a simple content-based recommendation system
// Given user likes/dislikes, recommend from a pool of items
function recommendContent(userProfile, itemPool) {
  // userProfile: [feature1, feature2, ...]
  // itemPool: [[item1Features], [item2Features], ...]
  // Return: Top 3 most similar items

  // Your solution using cosine similarity
}
```
