16. Numerical Methods: When Computers Lie

Phase 6: Real-World Applications & Limitations
⏱ ~40 minutes | 💻 Practical Limitations | ⚠️ Why Math Breaks in Computers

---

What Problem This Solves

You observe:

- 0.1 + 0.2 !== 0.3 in JavaScript (it's 0.30000000000000004)
- Your ML algorithm diverges or produces NaN unpredictably
- Financial calculations are off by pennies (then compound to dollars)
- Integration of a smooth function gives garbage results
- Newton's method crashes instead of converging
- GPU simulations diverge differently than CPU versions

Without numerical methods intuition, you treat computers as mathematical oracles. Numbers come out wrong, and you have no mental model for why or how to fix it.

With numerical methods, you understand floating-point representation, stability of algorithms, approximation trade-offs, and when to trust (or distrust) computed results.

Intuition & Mental Model

The Core Insight: Computers Are Approximate Machines

```
Mathematics:     Exact, infinite precision
                1/3 = 0.333333... (repeating)

Computers:       Finite precision, rounding errors
                1/3 ≈ 0.333333... (stops)
                   error = actual - computed
```

Mental Model: The Precision Budget

```
64-bit IEEE 754 float:
- 53 bits for significand (number)
- 11 bits for exponent (scale)
- 1 bit for sign

Result: ~15-17 decimal digits of precision
       Can't represent everything exactly
```

Core Concepts

1. Floating-Point Representation

```
function explainFloatingPoint() {
  // IEEE 754: ±1.mantissa × 2^exponent
```

```
  // Example: 0.1 in binary
  // 0.1 (decimal) → 0.0001100110011... (binary, repeating)
  // Rounded to 53 bits → Approximation
  // Error: 0.1 - computed ≈ 5.5×10^-18

  const x = 0.1;
  const y = 0.2;
  const sum = x + y;

  return {
    x,
    y,
    expectedSum: 0.3,
    actualSum: sum,
    error: Math.abs(sum - 0.3),
    why: 'Both 0.1 and 0.2 are inexact in binary',
    solution: 'Round before comparison or use decimal arithmetic'
  };
}

console.log(explainFloatingPoint());
/* {
  x: 0.1,
  y: 0.2,
  expectedSum: 0.3,
  actualSum: 0.30000000000000004,
  error: 4.4408920985006262e-17
} */

// How to fix it
function checkEqual(a, b, tolerance = 1e-10) {
  return Math.abs(a - b) < tolerance;
}

checkEqual(0.1 + 0.2, 0.3);  // true
```

Why This Happens:

```
// Not all decimal numbers are exactly representable in binary

// 0.1 in binary: 0.0001100110011... (repeating)
// Stored with limited precision: gets rounded
// When you add more rounded numbers, errors compound

// Safe rule: Use relative comparison for floats
function almostEqual(a, b, relTolerance = 1e-9) {
  return Math.abs((a - b) / Math.max(Math.abs(a), Math.abs(b))) < relTolerance;
}
```

```
almostEqual(0.1 + 0.2, 0.3);  // true
almostEqual(1000000.1 + 0.2, 1000000.3);  // true (scales with magnitude)
```

2. Numerical Stability

Problem: Some algorithms amplify small errors

```
function unstableVsStable() {
  // Quadratic formula: x = (-b ± √(b² - 4ac)) / 2a
  // When b > 0 and b² >> 4ac: cancellation error in (-b + √(...))

  function quadraticNaive(a, b, c) {
    const disc = Math.sqrt(b * b - 4 * a * c);
    const x1 = (-b + disc) / (2 * a);
    const x2 = (-b - disc) / (2 * a);
    return [x1, x2];
  }

  function quadraticStable(a, b, c) {
    // Use different form to avoid cancellation
    const disc = Math.sqrt(b * b - 4 * a * c);
    const q = b > 0 ? -(b + disc) / 2 : (-b + disc) / 2;
    const x1 = q / a;
    const x2 = c / q;
    return [x1, x2];  // More numerically stable
  }

  // Example: a=1, b=1e9, c=1
  // Discriminant ≈ b² (since 4ac is tiny)

  return {
    naive: quadraticNaive(1, 1e9, 1),
    stable: quadraticStable(1, 1e9, 1),
    difference: 'Naive loses precision, stable is robust'
  };
}

// Result: Different libraries use different formulations
// to maintain stability across all domains
```

Real Example: Summing Large and Small Numbers

```
function sumOrders() {
  // Financial scenario: Add tiny fees to large account balance

  const balance = 1e8;  // $100 million
  const fees = [0.01, 0.02, 0.01];  // Pennies

  // Naive (wrong order)
  let result1 = balance;
  for (const fee of fees) {
```

```
    result1 += fee;  // Adding tiny to huge → loses precision
  }

  // Stable (accumulate small first)
  let result2 = fees.reduce((sum, f) => sum + f, 0);
  result2 += balance;

  return {
    naive: result1,
    stable: result2,
    difference: result1 - result2,
    lesson: 'Order matters in floating point!'
  };
}

// Naive: Might lose some fees entirely
// Stable: Preserves all pennies
```

3. Root Finding: Approximating Solutions

```
function findRoot(f, fPrime, guess, iterations = 20) {
  // Newton's method: x_new = x_old - f(x) / f'(x)
  // Converges quadratically (fast!) but needs derivative

  let x = guess;
  const history = [{ iter: 0, x, fx: f(x) }];

  for (let i = 1; i <= iterations; i++) {
    const fx = f(x);
    const derivative = fPrime(x);

    if (Math.abs(derivative) < 1e-10) {
      // Derivative too small: algorithm fails
      return { failed: true, reason: 'Flat slope' };
    }

    x = x - fx / derivative;

    if (i % 5 === 0) {
      history.push({ iter: i, x: x.toFixed(10), fx: f(x).toFixed(2e-10) });
    }
  }

  return { root: x, history };
}

// Find root of f(x) = x² - 2 (answer: √2 ≈ 1.414)
const f = x => x * x - 2;
const fPrime = x => 2 * x;

findRoot(f, fPrime, 1.0);
```

```
/* {
  root: 1.4142135623730951,
  history: [
    { iter: 0, x: '1.0000000000', fx: '-1.00' },
    { iter: 5, x: '1.4142135624', fx: '0.00' },
    ...
  ]
}
// Converges to √2 in ~5 iterations */
```

Stability Issues:

```
// Newton fails if:
// 1. Derivative is zero (flat slope)
// 2. Initial guess too far from root
// 3. Multiple roots close together (confusing)

// Better: Use bisection (slower but more stable)
function bisection(f, a, b, tolerance = 1e-10) {
  // Guarantee: f(a) and f(b) have opposite signs
  while (b - a > tolerance) {
    const mid = (a + b) / 2;
    if (f(mid) * f(a) < 0) {
      b = mid;  // Root in left half
    } else {
      a = mid;  // Root in right half
    }
  }
  return (a + b) / 2;
}

bisection(x => x * x - 2, 1, 2);  // √2 ≈ 1.414
```

4. Numerical Integration: Approximating Area

```
function numericalIntegration(f, a, b, n = 1000) {
  // Approximate ∫f(x)dx from a to b

  // Trapezoidal rule: Connect points with line segments
  const h = (b - a) / n;
  let sum = (f(a) + f(b)) / 2;

  for (let i = 1; i < n; i++) {
    sum += f(a + i * h);
  }

  return sum * h;
}

// Integrate sin(x) from 0 to π
```

```
// True answer: 2.0
const sine = x => Math.sin(x);
console.log(numericalIntegration(sine, 0, Math.PI, 100));   // ≈ 1.9835 (100
segments)
console.log(numericalIntegration(sine, 0, Math.PI, 1000));  // ≈ 1.9998 (1000
segments)

// More segments → higher accuracy, but:
// - Takes longer
// - Eventually: rounding errors dominate!
// - Can't just use infinity (must choose optimal n)
```

Simpson's Rule (more accurate):

```
function simpsonIntegration(f, a, b, n = 100) {
  // n must be even
  if (n % 2 !== 0) n++;

  const h = (b - a) / n;
  let sum = f(a) + f(b);

  for (let i = 1; i < n; i++) {
    const coeff = i % 2 === 0 ? 2 : 4;
    sum += coeff * f(a + i * h);
  }

  return (sum * h) / 3;
}

// Same integral
console.log(simpsonIntegration(sine, 0, Math.PI, 100));  // ≈ 2.00000 (much better!)
```

5. Linear System Solving: When Errors Grow

```
function solveLinearSystem(A, b) {
  // Solve Ax = b using Gaussian elimination

  const n = A.length;
  const aug = A.map((row, i) => [...row, b[i]]);

  // Forward elimination with partial pivoting (more stable)
  for (let col = 0; col < n; col++) {
    // Find pivot (largest element for stability)
    let maxRow = col;
    for (let row = col + 1; row < n; row++) {
      if (Math.abs(aug[row][col]) > Math.abs(aug[maxRow][col])) {
        maxRow = row;
      }
    }
```

```
    if (Math.abs(aug[maxRow][col]) < 1e-10) {
      return { error: 'Singular matrix (no unique solution)' };
    }

    // Swap rows
    [aug[col], aug[maxRow]] = [aug[maxRow], aug[col]];

    // Eliminate below
    for (let row = col + 1; row < n; row++) {
      const factor = aug[row][col] / aug[col][col];
      for (let j = col; j <= n; j++) {
        aug[row][j] -= factor * aug[col][j];
      }
    }
  }

  // Back substitution
  const x = Array(n);
  for (let i = n - 1; i >= 0; i--) {
    x[i] = aug[i][n];
    for (let j = i + 1; j < n; j++) {
      x[i] -= aug[i][j] * x[j];
    }
    x[i] /= aug[i][i];
  }

  return x;
}

// Solve: 2x + y = 5, x + 3y = 5
// Answer: x=2, y=1
const A = [[2, 1], [1, 3]];
const b = [5, 5];
console.log(solveLinearSystem(A, b));  // [2, 1]
```

6. Condition Number: Sensitivity to Input Errors

```
function conditionNumber(matrix) {
  // How much output error grows from input error
  // High condition number = sensitive = unstable

  // Simplified (real calculation is more complex)
  // For matrix A: κ(A) ≈ ||A|| × ||A⁻¹||

  return {
    wellConditioned: 'Small errors → small output errors (κ ~ 1)',
    poorlyConditioned: 'Small errors → large output errors (κ >> 1)',
    example: 'Hilbert matrices are notoriously poorly conditioned'
  };
}
```

```
// Hilbert matrix (ill-conditioned)
const hilbert2 = [
  [1, 1/2],
  [1/2, 1/3]
];

// Even though it looks innocent, solving with it is numerically dangerous
// κ(H₂) ≈ 33, κ(H₄) ≈ 15,000, κ(H₁₀) ≈ 10¹¹

// Lesson: Avoid ill-conditioned matrices
//         Preconditioning can help
//         Use specialized libraries (they know the tricks)
```

Software Engineering Connections

1. Financial Calculations

```
function financialAccuracy() {
  // Rule: Store money as integers (cents/pence), not floats

  function naiveApproach() {
    let balance = 1000.00;  // $1000 in float
    balance -= 0.1;
    balance -= 0.1;
    balance -= 0.1;
    balance -= 0.1;
    balance -= 0.1;
    return balance;  // 999.4999999999999 (incorrect!)
  }

  function correctApproach() {
    let balanceCents = 100000;  // $1000 = 100,000 cents
    balanceCents -= 10;
    balanceCents -= 10;
    balanceCents -= 10;
    balanceCents -= 10;
    balanceCents -= 10;
    return balanceCents / 100;  // 999.50 (correct!)
  }

  return {
    naiveResult: naiveApproach(),
    correctResult: correctApproach(),
    lesson: 'Never use float for money. Use decimal or integers.'
  };
}
```

2. Scientific Computing

```
function simulatePhysics() {
  // Example: Iterate velocity and position
  // If step size too large → accumulates errors
  // If step size too small → rounding errors dominate

  function simulate(stepSize, steps) {
    let x = 0, v = 1;  // position, velocity
    let energy = [];

    for (let i = 0; i < steps; i++) {
      // Euler method (simple but unstable for large steps)
      const dx = v * stepSize;
      const dv = -x * stepSize;  // spring force
      x += dx;
      v += dv;

      // Total energy (should be conserved)
      const E = 0.5 * v * v + 0.5 * x * x;
      energy.push(E);
    }

    return energy;
  }

  // Stable: small step size
  const stable = simulate(0.01, 1000);

  // Unstable: large step size (diverges)
  const unstable = simulate(0.5, 100);

  return {
    stable_energy: stable.slice(-1),  // Stays ~1.0
    unstable_energy: unstable.slice(-1),  // Blows up to 10+
    lesson: 'Step size matters. Use adaptive/implicit methods for stiff systems.'
  };
}
```

3. Machine Learning Training

```
function neuralNetworkStability() {
  // Common issues:

  return {
    vanishingGradient: {
      problem: 'Gradients → 0, learning stops',
      cause: 'Deep networks + certain activations',
      solution: 'ReLU, skip connections, batch norm'
    },
    explodingGradient: {
      problem: 'Gradients → ∞, NaN/loss diverges',
```

```
      cause: 'Large learning rate or weight initialization',
      solution: 'Gradient clipping, lower learning rate, Xavier init'
    },
    numericalUnderflow: {
      problem: 'exp(-1000) → 0, log(0) → -∞',
      cause: 'Computing log(softmax(x)) naively',
      solution: 'Log-sum-exp trick, numerically stable implementations'
    }
  };
}

// Numerically stable softmax
function softmaxStable(logits) {
  const maxLogit = Math.max(...logits);
  const shifted = logits.map(x => x - maxLogit);  // Avoid overflow/underflow
  const exp = shifted.map(x => Math.exp(x));
  const sum = exp.reduce((a, b) => a + b, 0);
  return exp.map(x => x / sum);
}

softmaxStable([1000, 1001, 1002]);  // Works! (naive version gives [NaN, NaN, NaN])
```

Common Misconceptions

❌ "Computers do exact arithmetic"

They don't: Limited precision by design

```
// Not all numbers are representable
0.1 + 0.2 === 0.3;  // false!
1e20 + 1 === 1e20;   // true (1 gets lost!)
```

❌ "More precision is always better"

Trade-offs exist:

```
// Higher precision:
// ✓ More accuracy
// ✗ Slower
// ✗ More memory
// ✗ Sometimes introduces MORE error (paradox!)

// Optimal precision = problem-dependent
// Usually: 32-bit float fine, 64-bit for safety
```

❌ "If math says it's true, code proves it"

Counterexample: Series that should converge:

```
// Grandi's series: 1 - 1 + 1 - 1 + 1 - ...
// Math: Diverges (no sum)
// Naive code:
let sum = 0;
for (let i = 1; i <= 1000000; i++) {
  sum += Math.pow(-1, i + 1);
}
console.log(sum);  // Outputs 1 or 0 (wrong!)

// The series numerically "oscillates" but math disagrees
```

Practical Mini-Exercises

▶ **Exercise 1: Comparing Floats** (Click to expand)

▶ **Exercise 2: Stable Summation** (Click to expand)

Summary Cheat Sheet

```
// FLOATING-POINT BASICS
- 64-bit float: ~15-17 decimal digits precision
- Not all decimals are exactly representable
- Error compounds with operations

// STABILITY RULES
1. Don't add very large + very small numbers
2. Avoid subtracting nearly equal numbers
3. Use abs() comparison for floats (with tolerance)
4. Use specialized libraries (they know tricks)
5. If something seems wrong, suspect numerical issues first

// RED FLAGS
- NaN/Infinity appearing mysteriously
- Result doesn't match mathematical expectation
- Small changes in input cause large changes in output
- Algorithm oscillates instead of converges
```

Next Steps

✅ **You've completed**: Numerical methods & floating-point

➡️ **Up next**: 17. Randomized Algorithms - Why randomness helps, hashing, load balancing, shuffling

Before moving on:

```
// Challenge: Implement numerically stable variance calculation
function varianceStable(values) {
  // Implement without intermediate large numbers
```

```
  // Handle both large values and small deviations
}
```
