10. Financial Mathematics for Engineers

Phase 4: Math for Finance & Decision Making
⏱ ~45 minutes | 🎯 Practical | 💰 Essential for tech leadership

---

What Problem This Solves

You're evaluating:

- Should we build vs buy?
- Is this infrastructure investment worth it?
- How do we price our SaaS subscription?
- Should we pay off technical debt or ship features?
- What's the real cost of a 3-year cloud commitment?

Without financial math intuition, you rely on gut feeling for decisions involving time and money. You might underestimate long-term costs, overpay for "discounts," or miscompare options with different timelines.

With financial math, you quantify tradeoffs across time, compare apples-to-apples even when cash flows happen at different moments, and make engineering decisions that align with business value.

Intuition & Mental Model

The Core Insight: Money Has a Time Dimension

```
$100 today ≠ $100 next year
     ↓
Why? Because $100 today can:
- Earn interest in a bank (grows to $105 in a year at 5%)
- Generate revenue if invested in infrastructure
- Have less purchasing power next year (inflation)

Financial math = translating money across time
```

Mental Model: The Money Time Machine

```
           Compounding (Forward)
         ────────────────────────>
$100 today                      $??? future


         <─────────────────────────
           Discounting (Backward)
$??? today                      $100 future
```

- **Compounding** = "If I have $X today, what will it become?"
- **Discounting** = "If I need $X in the future, what's it worth today?"

Core Concepts

1. Simple vs Compound Interest

Simple Interest: Interest only on principal

```
function simpleInterest(principal, rate, years) {
  return principal * (1 + rate * years);
}

// Example: $1000 at 5% simple interest for 3 years
simpleInterest(1000, 0.05, 3);  // $1150
// Year 1: +$50, Year 2: +$50, Year 3: +$50
```

Compound Interest: Interest on interest (exponential)

```
function compoundInterest(principal, rate, years) {
  return principal * Math.pow(1 + rate, years);
}

// Same setup with compounding
compoundInterest(1000, 0.05, 3);  // $1157.63
// Year 1: +$50, Year 2: +$52.50, Year 3: +$55.13
```

Why It Matters:

```
Simple:   Linear growth    (Y = X)
Compound: Exponential growth (Y = X^n)

At 5% over 30 years:
Simple:    $1000 → $2500     (2.5x)
Compound:  $1000 → $4321     (4.3x)
```

Real-world example: **AWS Reserved Instances**

```
// Pay upfront (simple cost) vs pay monthly (compound opportunity cost)
function compareReservation(monthlyPrice, upfrontPrice, months, opportunityCostRate)
{
  // Option 1: Pay upfront (money locked in)
  const upfrontCost = upfrontPrice;

  // Option 2: Pay monthly, invest the difference
  let payAsYouGo = 0;
  let investmentValue = upfrontPrice;  // Start with same amount

  for (let i = 0; i < months; i++) {
    payAsYouGo += monthlyPrice;
    investmentValue = investmentValue * (1 + opportunityCostRate / 12);
  }

  const netOpportunityCost = investmentValue - payAsYouGo;

  return {
    upfrontCost,
    payAsYouGoTotal: payAsYouGo,
    investmentValue,
    netOpportunityCost,
    winner: netOpportunityCost > 0 ? 'Pay monthly + invest' : 'Pay upfront'
  };
}

// Should you pay $10k upfront or $400/month for 3 years?
// If you could invest that $10k at 8% annual return:
compareReservation(400, 10000, 36, 0.08);
/* {
  upfrontCost: 10000,
  payAsYouGoTotal: 14400,
  investmentValue: 12682,
  netOpportunityCost: -1718,  // Negative! Pay upfront is better
  winner: 'Pay upfront'
} */
```

2. Present Value & Future Value

Future Value (FV): What will money become?

```
function futureValue(present, rate, years) {
  return present * Math.pow(1 + rate, years);
}

// If I invest $5000 in infrastructure automation today,
// and it saves 10 hours/month at $100/hour (120% annual return):
futureValue(5000, 1.2, 3);  // $52,998 in savings over 3 years
```

Present Value (PV): What is future money worth today?

```
function presentValue(future, rate, years) {
  return future / Math.pow(1 + rate, years);
}

// Client offers $100k contract paid in 2 years
// What's it worth to you today? (Discount rate = 10%)
presentValue(100000, 0.10, 2);  // $82,645

// Interpretation: $82,645 today = $100k in 2 years
```

Why Discounting Matters: The Time Machine Analogy

```
Question: Would you rather have $100 today or $100 in a year?

Obviously today! Because:
1. You could invest it and have >$100 in a year
2. Inflation erodes purchasing power
3. Uncertainty (bird in hand vs bush)
```

```
So future money is "discounted" when valuing it today.
```

3. Net Present Value (NPV)

The Decision-Making Tool: Should you invest in a project?

```
function calculateNPV(initialInvestment, cashFlows, discountRate) {
  let npv = -initialInvestment;  // Cost is negative

  cashFlows.forEach((cashFlow, year) => {
    npv += presentValue(cashFlow, discountRate, year + 1);
  });

  return npv;
}

// Example: Should we build a custom CMS?
const buildCost = 50000;           // Initial investment
const yearlySavings = [15000, 20000, 25000, 25000];  // Save on SaaS fees
const discountRate = 0.12;         // Our cost of capital

const npv = calculateNPV(buildCost, yearlySavings, discountRate);
console.log(npv);  // $13,855

// NPV > 0 → Good investment! (breakeven is ~2.5 years)
// NPV < 0 → Don't do it
// NPV = 0 → Indifferent
```

Real Example: Paying Down Technical Debt

```
function shouldWeRefactor(refactorCost, currentMonthlyWaste, improvementPercent,
years) {
  const monthlySavings = currentMonthlyWaste * improvementPercent;
  const annualSavings = monthlySavings * 12;

  // Generate cash flows
  const cashFlows = Array(years).fill(annualSavings);

  const npv = calculateNPV(refactorCost, cashFlows, 0.15);  // 15% discount rate

  return {
    npv,
    decision: npv > 0 ? 'Refactor' : 'Keep status quo',
    breakEvenYears: refactorCost / annualSavings,
    roi: (npv / refactorCost * 100).toFixed(1) + '%'
  };
}

// We have a slow API endpoint costing 5 hours/week debugging (=$2000/month)
// Refactor costs $15k, improves performance by 70%
```

```
shouldWeRefactor(15000, 2000, 0.70, 3);
/* {
  npv: 11,241,
  decision: 'Refactor',
  breakEvenYears: 0.89,  // Pays for itself in ~11 months
  roi: '74.9%'
} */
```

4. Annuities: Regular Payments

Problem: Subscription pricing, loan payments, recurring costs

```
// Present value of an annuity (series of equal payments)
function presentValueAnnuity(payment, rate, periods) {
  if (rate === 0) return payment * periods;
  return payment * (1 - Math.pow(1 + rate, -periods)) / rate;
}

// Customer wants to pay $100/month for 2 years
// What's that worth as a lump sum today? (10% annual discount = 0.83% monthly)
presentValueAnnuity(100, 0.0083, 24);  // $2,178

// So $2,200 today ≈ $100/month for 2 years
```

Monthly Subscription Calculator

```
function calculateSubscriptionValue(monthlyPrice, expectedLifetimeMonths, churnRate,
discountRate) {
  let pv = 0;
  const monthlyRate = discountRate / 12;

  for (let month = 1; month <= expectedLifetimeMonths; month++) {
    // Each month, fewer customers remain (churn)
    const retentionFactor = Math.pow(1 - churnRate, month);
    const expectedPayment = monthlyPrice * retentionFactor;
    pv += expectedPayment / Math.pow(1 + monthlyRate, month);
  }

  return pv;
}

// SaaS pricing: $50/month, 5% monthly churn, 10% annual discount rate
calculateSubscriptionValue(50, 36, 0.05, 0.10);  // $637 LTV

// This is your Customer Lifetime Value (LTV)
```

5. Internal Rate of Return (IRR)

Question: What rate of return does an investment give?

```
// Find the discount rate where NPV = 0
function calculateIRR(initialInvestment, cashFlows) {
  // Binary search for the rate
  let low = -0.5, high = 1.0;
  const tolerance = 0.0001;

  while (high - low > tolerance) {
    const rate = (low + high) / 2;
    const npv = calculateNPV(initialInvestment, cashFlows, rate);

    if (npv > 0) {
      low = rate;  // Rate too low, NPV positive
    } else {
      high = rate;  // Rate too high, NPV negative
    }
  }

  return (low + high) / 2;
}

// We invest $20k in CI/CD automation
// It saves us $8k, $10k, $12k over 3 years
const irr = calculateIRR(20000, [8000, 10000, 12000]);
console.log((irr * 100).toFixed(1) + '%');  // ~23% return

// Compare to alternatives:
// - Stock market: ~10%
// - Corporate bonds: ~5%
// - This project: ~23% → Great investment!
```

6. Inflation & Real vs Nominal Rates

Nominal Rate: What the number says
Real Rate: Adjusted for inflation

```
function realRate(nominalRate, inflationRate) {
  // Fisher equation: (1 + real) = (1 + nominal) / (1 + inflation)
  return (1 + nominalRate) / (1 + inflationRate) - 1;
}

// Bank offers 7% savings account, but inflation is 3%
realRate(0.07, 0.03);  // 3.88% real return

// Your salary increases 5% but inflation is 4%
realRate(0.05, 0.04);  // 0.96% real raise (barely keeping up!)
```

Software Cost Example:

```
function adjustForInflation(costs, inflationRate) {
  return costs.map((cost, year) => ({
    year,
    nominalCost: cost,
    realCost: cost / Math.pow(1 + inflationRate, year)
  }));
}

// AWS pricing stays flat at $1000/month for 5 years
// With 3% inflation, what's the real cost trend?
const flatPricing = Array(5).fill(12000);  // $12k/year
adjustForInflation(flatPricing, 0.03);
/* [
  { year: 0, nominalCost: 12000, realCost: 12000 },
  { year: 1, nominalCost: 12000, realCost: 11650 },
  { year: 2, nominalCost: 12000, realCost: 11311 },
  { year: 3, nominalCost: 12000, realCost: 10981 },
  { year: 4, nominalCost: 12000, realCost: 10660 }
]
// Flat pricing = getting cheaper in real terms! */
```

7. Risk & Expected Return

Higher Risk → Require Higher Return

```
function requiredReturn(riskFreeRate, beta, marketReturn) {
  // CAPM formula: Expected Return = Risk-Free + Beta × (Market - Risk-Free)
  return riskFreeRate + beta * (marketReturn - riskFreeRate);
}

// Treasury bonds: 3% (risk-free)
// Stock market: 10% average
// High-risk startup: Beta = 2.0 (twice as volatile as market)
requiredReturn(0.03, 2.0, 0.10);  // 17% expected return to justify risk
```

Engineering Decision Example:

```
function compareTechOptions(options) {
  return options.map(option => {
    const { name, upfrontCost, yearlySavings, riskFactor } = option;

    // Higher risk → require higher discount rate
    const discountRate = 0.10 + (riskFactor * 0.05);
    const cashFlows = Array(3).fill(yearlySavings);
    const npv = calculateNPV(upfrontCost, cashFlows, discountRate);

    return { name, npv, discountRate, decision: npv > 0 ? '✅' : '❌' };
  });
}

compareTechOptions([
  { name: 'Boring SQL optimization', upfrontCost: 10000, yearlySavings: 8000,
riskFactor: 0 },
  { name: 'New NoSQL DB migration', upfrontCost: 50000, yearlySavings: 30000,
riskFactor: 2 },
  { name: 'GraphQL API rewrite', upfrontCost: 30000, yearlySavings: 15000,
riskFactor: 1 }
]);
/* [
  { name: 'Boring SQL', npv: 9,895, discountRate: 0.10, decision: '✅' },
  { name: 'NoSQL migration', npv: 23,841, discountRate: 0.20, decision: '✅' },
  { name: 'GraphQL rewrite', npv: 5,211, discountRate: 0.15, decision: '✅' }
]
// All positive NPV, but SQL optimization has best risk-adjusted return */
```

Software Engineering Connections

1. Build vs Buy Decisions

```
function buildVsBuy(buildCost, buySaasYearlyCost, years, maintenancePercent,
discountRate) {
  // Build option
  const buildMaintenance = Array.from({ length: years }, (_, i) =>
    buildCost * maintenancePercent * (i + 1)  // Maintenance grows over time
  );
  const buildNPV = calculateNPV(buildCost, buildMaintenance.map(c => -c),
discountRate);

  // Buy option (SaaS)
  const buyNPV = -presentValueAnnuity(buySaasYearlyCost, discountRate, years);

  return {
    buildTotalCost: -buildNPV,
    buyTotalCost: -buyNPV,
    winner: buildNPV > buyNPV ? 'Build' : 'Buy',
    savings: Math.abs(buildNPV - buyNPV)
  };
}

buildVsBuy(100000, 25000, 5, 0.15, 0.12);
// Factors: Initial cost, ongoing SaaS price, timeframe, maintenance burden,
discount rate
```

2. Technical Debt Interest Rate

```
// Technical debt accumulates "interest" = increasing maintenance cost
function technicalDebtGrowth(initialTechDebt, growthRate, years) {
  return Array.from({ length: years }, (_, year) => ({
    year,
    debtCost: initialTechDebt * Math.pow(1 + growthRate, year)
  }));
}
```

```
// Starting with 10 hours/week wasted, growing 20% per year
technicalDebtGrowth(10, 0.20, 5);
/* Year 0: 10 hrs
   Year 1: 12 hrs
   Year 2: 14.4 hrs
   Year 3: 17.3 hrs
   Year 4: 20.7 hrs
   Year 5: 24.9 hrs

Technical debt has a compound "interest rate"! */
```

3. SaaS Pricing Strategy

```
function optimizeSaasPricing(targetLTV, churnRate, years) {
  // Work backwards: What monthly price gives us target LTV?
  const monthlyRate = 0.10 / 12;  // 10% annual discount

  let low = 1, high = 10000;

  while (high - low > 0.01) {
    const price = (low + high) / 2;
    const ltv = calculateSubscriptionValue(price, years, churnRate, 0.10);

    if (ltv < targetLTV) {
      low = price;
    } else {
      high = price;
    }
  }

  return (low + high) / 2;
}

// We want $1000 LTV, 3% monthly churn, 3-year window
optimizeSaasPricing(1000, 0.03, 36);  // ~$56/month needed
```

4. Infrastructure ROI

```
function infrastructureROI(cost, timesSaved, hourlyRate, improvementYears) {
  const annualSavings = timesSaved * hourlyRate * 12;  // Monthly savings
  const cashFlows = Array(improvementYears).fill(annualSavings);

  const npv = calculateNPV(cost, cashFlows, 0.15);
  const paybackPeriod = cost / annualSavings;
  const irr = calculateIRR(cost, cashFlows);

  return {
    npv: npv.toFixed(0),
    paybackYears: paybackPeriod.toFixed(2),
    irr: (irr * 100).toFixed(1) + '%',
    decision: npv > 0 ? 'Invest' : 'Skip'
  };
}

// Kubernetes setup: $30k, saves 40 hours/month, $100/hr engineers
infrastructureROI(30000, 40, 100, 3);
/* {
  npv: '79,144',
  paybackYears: '0.63',  // 7.5 months to break even
  irr: '156.8%',
  decision: 'Invest'
} */
```

5. Cloud Commitment Discounts

```
function evaluateCloudCommitment(onDemandMonthly, commitmentDiscount,
commitmentYears, actualUsagePercent) {
  const committedMonthly = onDemandMonthly * (1 - commitmentDiscount);
  const months = commitmentYears * 12;

  // Total paid under commitment
  const committedTotal = committedMonthly * months;

  // Actual value received (if usage drops)
  const actualTotal = onDemandMonthly * actualUsagePercent * months;

  const savings = actualTotal - committedTotal;

  return {
    committedTotal,
    onDemandTotal: actualTotal,
    netSavings: savings,
    percentSaved: (savings / actualTotal * 100).toFixed(1) + '%',
    worthIt: savings > 0
  };
}

// $10k/month on-demand, 30% discount for 3-year commit, but usage drops to 80%
evaluateCloudCommitment(10000, 0.30, 3, 0.80);
/* {
  committedTotal: 252,000,
  onDemandTotal: 288,000,
  netSavings: 36,000,
  percentSaved: '12.5%',
  worthIt: true
}
// Even with 20% usage drop, commitment saves money */
```

Common Misconceptions

❌ "5% interest = money doubles in 20 years"

Wrong: That's simple interest.

```
// Simple: 5% × 20 years = 100% → 2x
simpleInterest(1000, 0.05, 20);  // $2000

// Compound: (1.05)^20 = 2.65x
compoundInterest(1000, 0.05, 20);  // $2653
```

Use the **Rule of 72**: Years to double ≈ 72 / interest rate

- 5% → 72/5 = 14.4 years
- 10% → 7.2 years
- 8% → 9 years

❌ "Discount rate doesn't matter much"

```
function showDiscountImpact(futureCashFlow, years) {
  return [0.05, 0.10, 0.15, 0.20].map(rate => ({
    rate: (rate * 100) + '%',
    presentValue: presentValue(futureCashFlow, rate, years).toFixed(0)
  }));
}

showDiscountImpact(100000, 5);
/* [
  { rate: '5%', presentValue: '78353' },
  { rate: '10%', presentValue: '62092' },
  { rate: '15%', presentValue: '49718' },
  { rate: '20%', presentValue: '40188' }
]
// 5% vs 20% = 2x difference in valuation! */
```

❌ "Pay upfront discounts are always worth it"

Not if you have better uses for the money:

```
// AWS offers 40% off for 3-year upfront: $60k vs $100k over 3 years
// But you could invest that $60k at 15% return

const upfrontCost = 60000;
const paygoCost = 100000 / 36;  // Monthly

let invested = upfrontCost;
for (let month = 0; month < 36; month++) {
  invested = invested * (1 + 0.15 / 12) - paygoCost;
}
```

```
console.log(invested);  // Negative! Upfront was better.

// But if your investment returns 25%:
invested = upfrontCost;
for (let month = 0; month < 36; month++) {
  invested = invested * (1 + 0.25 / 12) - paygoCost;
}
console.log(invested);  // Positive! Pay monthly + invest wins.
```

❌ "NPV and IRR always agree"

They can disagree on mutually exclusive projects:

```
const projectA = { cost: 1000, cashFlows: [1200], npv: calculateNPV(1000, [1200],
0.10) };  // NPV: $91
const projectB = { cost: 10000, cashFlows: [11500], npv: calculateNPV(10000,
[11500], 0.10) };  // NPV: $455

projectA.irr = calculateIRR(1000, [1200]);  // 20%
projectB.irr = calculateIRR(10000, [11500]);  // 15%

// IRR says A (20% > 15%)
// NPV says B ($455 > $91)
// NPV is correct! It measures absolute value creation.
```

Practical Mini-Exercises

- ▶ **Exercise 1: Compound Interest** (Click to expand)
- ▶ **Exercise 2: Build vs Buy** (Click to expand)
- ▶ **Exercise 3: Technical Debt** (Click to expand)
- ▶ **Exercise 4: SaaS Lifetime Value** (Click to expand)

Summary Cheat Sheet

```
// CORE FORMULAS

// 1. Compound Interest
FV = PV × (1 + r)^n

// 2. Present Value
PV = FV / (1 + r)^n

// 3. NPV
NPV = -Initial Investment + Σ(Cash Flow_t / (1 + r)^t)
Decision: NPV > 0 → Invest

// 4. Annuity (Equal Payments)
PV = Payment × [(1 - (1 + r)^-n) / r]
```

```
// 5. Real Rate (Inflation-Adjusted)
Real = (1 + Nominal) / (1 + Inflation) - 1

// 6. Rule of 72
Years to Double ≈ 72 / Interest Rate

// DECISION HEURISTICS
- Compare projects → Use NPV (absolute value)
- Compare returns → Use IRR (percentage)
- Always discount future cash flows
- Higher risk → Higher required return
- Technical debt = Compound interest on wasted time
```

Key Takeaways:

1. Money has a time dimension (compounding & discounting)
2. NPV is the universal decision tool
3. Always adjust for risk and inflation
4. Payback period ≠ Profitability
5. Long-term costs compound exponentially

Next Steps

✅ **You've completed**: Financial math for engineering decisions
➡️ **Up next**: 11. Exponential Growth & Decay - Compounding effects, viral growth, technical debt as exponential decay

Before moving on, try this:

```
// Real-world challenge: Should your startup pay engineers $150k salary
// or $120k + $30k in stock options that vest over 4 years?
// (Assume stock grows 30%/year, 10% discount rate, and stock has 2x risk)

function compensationComparison() {
  // Your solution here
}
```
