3.3: Modular Programming

Introduction to Computer Science · Robert Sedgewick and Kevin Wayne · Copyright © 2005 · http://www.cs.Princeton.EDU/IntroCS

Review

Data type: set of values and operations on those values.

A Java `class` allows us to define a data type by:
- Specifying a set of variables.
- Defining operations on those values.

Modular programming: break up a larger program into smaller independent pieces.
- Class = program that defines a data type.
- Client = program that uses a data type.

Procedural vs. Object Oriented Programming

Procedural programming. [verb-oriented]
- Tell the computer to do this.
- Tell the computer to do that.

Alan Kay's philosophy. Software is a simulation of the real world.
- We know (approximately) how the real world works.
- Design software to model the real world.

Objected oriented programming (OOP). [noun-oriented]
- Programming paradigm based on data types.
- Identify things that are part of the problem domain or solution.
- Things in the world know things: instance variables.
- Things in the world do things: methods.

Alan Kay

Alan Kay. [Xerox PARC 1970s]
- Inventor of Smalltalk programming language.
- Conceiver of Dynabook portable computer.
- Ideas led to: laptop, modern GUI, OOP.

> "The computer revolution hasn't started yet."
>
> "The best way to predict the future is to invent it."
>
> "If you don't fail at least 90 percent of the time, you're not aiming high enough."

Turing Award (2003)
Kyoto Prize (2005)

OOP Advantages

OOP enables:

- Data abstraction: manage complexity of large programs.
- Modular programming: divide large program into smaller independent pieces.
  Encapsulation: hide information to make programs robust.
- Inheritance: reuse code.

Religious wars ongoing.

9.8: Monte Carlo Simulation

"Doing physics by tossing dice."

Ferromagnetism

Ferromagnetism. Phenomenon by which a material exhibits spontaneous magnetization.

- Explains most of magnetization that arises in everyday life.
- Quantum mechanical cause: spin and Pauli exclusion principle.

refrigerator magnet

Phase transition.

- Curie temperature: ferromagnetism occurs if temperature < T_c; does not occur if temperature > T_c
- Ex: for iron T_c = 1043° K; for nickel T_c = 627° K.
- Study of phase transitions showed defect in mean field theory.

via computational approaches (Ising spin model)

later replaced by renormalization group theory

Ising Spin Model

Ising spin model. [Lenz 1920, Ising 1924] Simple classical model of ferromagnetism that provides insight into phase transitions.

- N-by-N lattice S of cells.
- Cell (i, j) has spin $s_{ij} = \pm 1$.
- Interactions occur only between a cell and its 4 nearest neighbors.

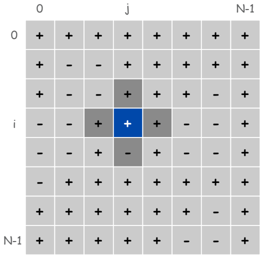
Physical quantities.

- Magnetization: $M(S) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} s_{ij}$
- Cell energy: $e_{ij} = -s_{ij} \left(s_{i(j+1)} + s_{i(j-1)} + s_{(i+1)j} + s_{(i-1)j} \right)$

nearest neighbors

Energy: $E(S) = \frac{1}{2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} e_{ij}$

we double-counted



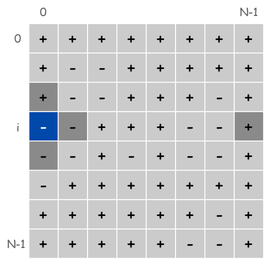
	0			j				N-1
0	+	+	+	+	+	+	+	+
	+	-	-	+	+	+	+	+
	+	-	-	+	+	+	-	+
i	-	-	+	+	+	-	-	+
	-	-	+	-	+	-	-	+
	-	+	+	+	+	+	+	+
	+	+	+	+	+	+	-	+
N-1	+	+	+	+	+	-	-	+

8-by-8 lattice S
s_{ij} = +1, e_{ij} = -2

Lattice

N-by-N lattice.
- Typical lab system has N = 10^9.
- Approximate with smaller systems. Why?

Periodic boundary condition.
- Mitigates finite size effect.
- Cell (i, 0) and (i, N-1) are neighbors.
- Cell (0, j) and (N-1, j) are neighbors.
- Topologically equivalent to a torus.



	0							N-1
0	+	+	+	+	+	+	+	+
	+	-	-	+	+	+	+	+
	+	-	-	+	+	+	-	+
i	-	-	+	+	+	-	-	+
	-	-	+	-	+	-	-	+
	-	+	+	+	+	+	+	+
	+	+	+	+	+	+	-	+
N-1	+	+	+	+	+	-	-	+

8-by-8 lattice S
$s_{i0} = -1$, $e_{i0} = 0$

Cell.java

```java
public class Cell {
   private boolean spin;

   public Cell(double p) {
      spin = (Math.random() < p);
   }

   public void flip() { spin = !spin; }

   public double magnetization() {
      if (spin) return +1.0;
      else      return -1.0;
   }

   public void draw() {
      if (spin) StdDraw.setColor(StdDraw.WHITE);
      else      StdDraw.setColor(StdDraw.BLUE);
      StdDraw.spot(1, 1);
   }
}
```

+1 with probability p

State.java

```java
public class State {
   private int N;
   private Cell[][] lattice;

   public State(int N, double p) {
      this.N       = N;
      this.lattice = new Cell[N][N];
      for (int i = 0; i < N; i++)
         for (int j = 0; j < N; j++)
            lattice[i][j] = new Cell(p);
   }

   public void draw() {
      for (int i = 0; i < N; i++) {
         for (int j = 0; j < N; j++) {
            StdDraw.moveTo(i + 0.5, j + 0.5);
            lattice[i][j].draw();
         }
      }
   }
```

State.java (cont)

$$M(S) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} s_{ij}$$

```java
public double magnetization() {
      double M = 0.0;
      for (int i = 0; i < N; i++)
         for (int j = 0; j < N; j++)
               M += lattice[i][j].magnetization()
      return M;
}
```

$$e_{ij} = -s_{ij}\left(s_{i(j+1)} + s_{i(j-1)} + s_{(i+1)j} + s_{(i-1)j}\right)$$

```java
private double energy(int i, int j) {
    double E = 0.0;
    E += lattice[(i+1)%N][j].magnetization();
    E += lattice[i][(j+1)%N].magnetization();
    E += lattice[(i-1+N)%N][j].magnetization();
    E += lattice[i][(j-1+N)%N].magnetization();
    E *= lattice[i][j].magnetization();
    return -E;
}
```

use % N for periodic boundary conditions

$$E(S) = \frac{1}{2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} e_{ij}$$

```java
public double energy() {
   double E = 0.0;
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         E += energy(i, j);
   return 0.5 * E;
}
```

Boltzmann Energy Distribution

Boltzmann distribution. The probability of finding system in state S is:

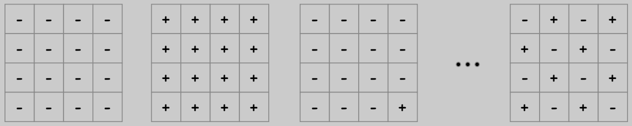
$$Bz(S) = \frac{e^{-E(S)\,/\,kT}}{Z}$$

kT = temperature

between 0 and 4

normalization constant Z chosen so that Σ Bz(S) = 1

Mean magnetization. $$< M > \; = \sum_{\text{states } S} M(S)\; Bz(S)$$



E(S) = -32
Bz(S) = 0.073

$\frac{e^{32/3}}{585{,}976}$

E(S) = -32
Bz(S) = 0.073

E(S) = -24
Bz(S) = 0.0051

E(S) = +32
Bz(S) = $4 \cdot 10^{-11}$

N = 4, kT = 3 ⇒ Z ≈ 585,976

Boltzmann Energy Distribution

Boltzmann distribution. The probability of finding system in state S is:

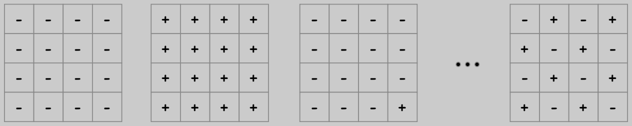
$$Bz(S) = \frac{e^{-E(S)\,/\,kT}}{Z}$$

kT = temperature

between 0 and 4

normalization constant Z chosen so that Σ Bz(S) = 1

Mean magnetization. $$< M > \; = \sum_{\text{states } S} M(S)\; Bz(S)$$



E(S) = -32
Bz(S) = 0.33

E(S) = -32
Bz(S) = 0.33

E(S) = -24
Bz(S) = 0.0061

E(S) = +32
Bz(S) = $4 \cdot 10^{-15}$

N = 4, kT = 2 ⇒ Z ≈ 26,839,014

Boltzmann Energy Distribution

Boltzmann distribution. The probability of finding system in state S is:

$$Bz(S) = \frac{e^{-E(S)\,/\,kT}}{Z}$$

kT = temperature

between 0 and 4

normalization constant Z chosen so that Σ Bz(S) = 1

Mean magnetization. $$< M > \; = \sum_{\text{states } S} M(S)\; Bz(S)$$

Computational approaches.

- Direct computation: not practical since Z is sum of 2^{N*N} values.
- Random sampling: take average of a random sampling of states.
  - uniformly random states won't approximate distribution (also won't contribute much to sum)
- Importance sampling: choose state S with probability Bz(S).

Metropolis Algorithm

Goal. Dynamic process to create states with desired probability.

Single-flip dynamics.

- Cell (i, j) randomly flips spin.
- Physical interpretation: result of vibrations in lattice.

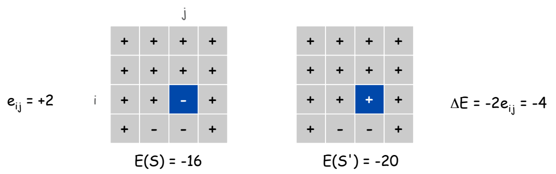
Metropolis algorithm. [Metropolis-Rosenbruth-Rosenbruth-Teller-Teller 1953]

- Current state S.
- Randomly flip spin of one cell to get state S'.
- Let ΔE = E(S') - E(S) be change in energy.
- If ΔE < 0, update state to S'.
  Else, update state to S' with probability $e^{-\Delta E/kT}$.

Theorem. Long run fraction of time process is in state S = Bz(S).

State.java

```
public void metropolis(int i, int j, double kT) {
   double deltaE = -2 * energy(i, j);
   if (deltaE < 0 || (Math.random() <= Math.exp(-deltaE / kT))
      cell[i][j].flip();
}

public void phase(double kT) {
   for (int steps = 0; steps < N*N; steps++) {
      int i = (int) (Math.random() * N);
      int j = (int) (Math.random() * N);
      metropolis(i, j, kT);
   }
}
```



+	+	+	+
+	+	+	+
+	+	-	+
+	-	-	+

E(S) = -16

+	+	+	+
+	+	+	+
+	+	+	+
+	-	-	+

E(S') = -20

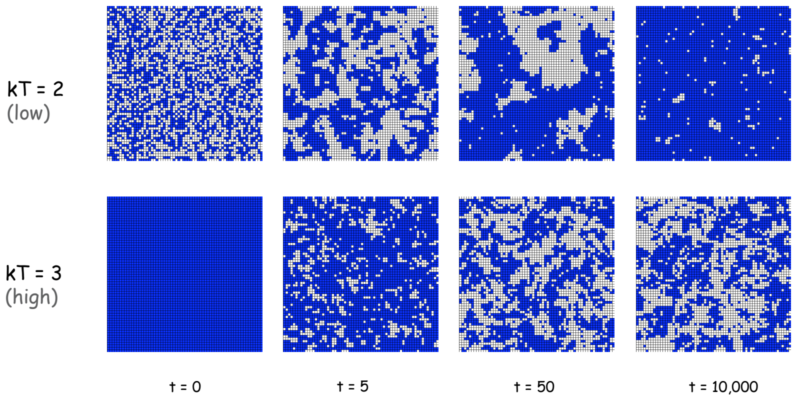
Metropolis Visualization

Animate Metropolis algorithm by plotting results after each phase.

```
public class Metropolis {
   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      double kT = Double.parseDouble(args[1]);
      StdDraw.create(512, 512);
      StdDraw.setScale(0, 0, N, N);
      State state = new State(N, 0.5);
      while (true) {
         state.phase(kT);
         state.draw();
         StdDraw.pause(50);
      }
   }
}
```

Spin Over Time

Low kT: strong tendency for all spins to align.
High kT: strong tendency for spins to cancel out.



Metropolis Algorithm: Properties

Estimate physical quantities. Let S_t be Metropolis state after t phases.

- Estimate mean magnetization by: $$< M > \approx \frac{M(S_1) + \ldots + M(S_t)}{t}$$
- Estimate mean energy by: $$< E > \approx \frac{E(S_1) + \ldots + E(S_t)}{t}$$
- Estimate mean squared energy by: $$< E^2 > \approx \frac{E(S_1)^2 + \ldots + E(S_t)^2}{t}$$
- Estimate heat capacity by: $$C \approx \frac{1}{kT^2}\left(< E^2 > - < E >^2\right)$$

Consequence. Easy to estimate relevant physical quantities.

Mean Magnetization

Approach. Perform phases; record average value of desired quantity.

```
// mean magentization per cell in N-by-N grid, at
// temperature kT, after given number of phases
public static double meanM(int N, double kT, int PHASES) {
   State state = new State(N, 0.5);
   double sumM = 0.0;
   for (int t = 0; t < PHASES; t++) {
      state.phase(kT);
      sumM += state.magnetization() / (N*N);
   }
   return sumM / PHASES;
}
```

Computational Experiments

Implementation details.

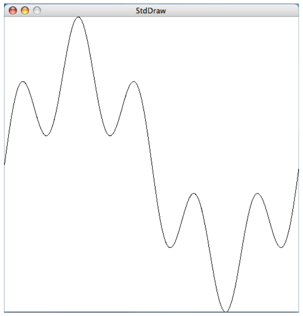
- How to choose N?
- How many phases is enough?
- Burn-in: run for B phases before accumulating statistics.

Goal. Run a computational experiment and visualize results.

- Run for various values of temperature.
- Plot mean magnetization vs. temperature.

Plotting

Diversion. Plot an array.



$a(t) = \sin(t) + \frac{1}{2}\sin(5t), \quad t = 0..2\pi$

Scientific Experiment

Diversion. Plot an array.

```
public static void main(String[] args) {
   StdDraw.create(512, 512);
   int N = Integer.parseInt(args[0]);
   double[] a = new double[N];
   for (int i = 0; i < N; i++) {
      double t = 2 * Math.PI * i / N;
      a[i] = Math.sin(t) + 0.5 * Math.sin(5*t);
   }
→  Plot.lines(a);
   StdDraw.show();
}
```

$a(t) = \sin(t) + \frac{1}{2}\sin(5t), \quad t = 0..2\pi$

Plotting Library

Plot library. Simple library for plotting points and lines.

```
public class Plot {
   public static void lines(double[] a) {
      double ymin = ProbStat.min(a);
      double ymax = ProbStat.max(a);
      StdDraw.setScale(0, ymin, a.length - 1, ymax);
      for (int i = 1; i < a.length; i++) {
         StdDraw.moveTo(i-1, a[i-1]);
         StdDraw.lineTo(i, a[i]);
      }
   }

   public static void spots(double[] a) {
      ...
      for (int i = 0; i < a.length; i++)
         StdDraw.moveTo(i, a[i]);
         StdDraw.spot();
      }
   }
}
```

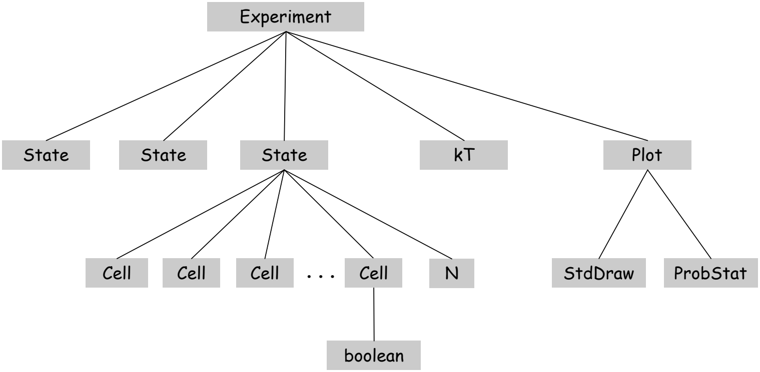
Probability and Statistics Library

ProbStat library. Library for probability and statistics functions.

```
public class ProbStat {

   public static double min(double[] a) {
      double min = Double.POSITIVE_INFINITY;
      for (int i = 0; i < a.length; i++)
         if (a[i] < min) min = a[i];
      return min;
   }

   public static double max(double[] a) { }

   public static double mean(double[] a) {
      double sum = 0.0;
      for (int i = 0; i < a.length; i++)
         sum = sum + a[i];
      return sum / a.length;
   }
}
```

Layers of Abstraction

Relationships among data types.



Computational Experiment

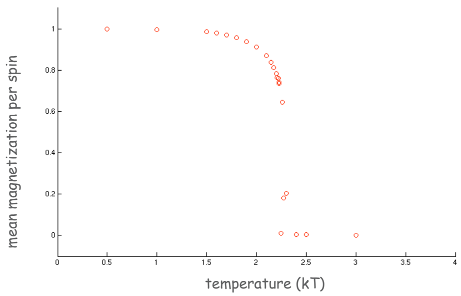
Approach. Perform phases; record average value of desired quantity.

```
public class Experiment {
   public static void main(String[] args) {
      int N      = Integer.parseInt(args[0]);
      int PHASES = Integer.parseInt(args[1]);

      double[] m = new double[N];
      for (int i = 0; i < N; i++) {
         kT = 4.0 * i / N;
         m[i] = meanM(N, kT, PHASES);
         m[i] /= (N * N);

      }
      Plot.lines(m);
   }
}
```

plot mean magnetization per spin vs. temperature

Computational Experiments

Plot. N = 64; kT = 0 to 4; 100,000 phases.



Hypothesis. Phase transition near kT = 2.2, above which material loses its magnetization.

Analytic Solution

2D lattice exact solution. [Onsager 1944]

- Critical temperature:

$$\sinh\left(\frac{2}{kT}\right) = 1$$

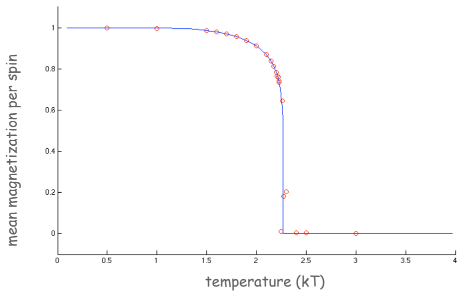
$$\Rightarrow kT_c \approx 2.269185$$

- Mean magnetization per spin:

$$= \begin{cases} \left[1 - \sinh^{-4}\left(\frac{2}{kT}\right)\right]^{1/8} & \text{if } T < T_c \\ 0 & \text{if } T \geq T_c \end{cases}$$

Computational Experiment vs. Theory

Compare experiment and theory. Plot experimental data points and predicted values.

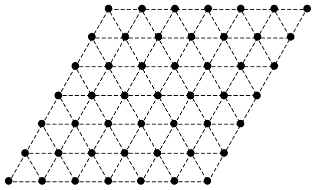


Conclusion. Computational experiments agree closely with theory.
(and both predict phase transition observed in physical experiments)

Lattices

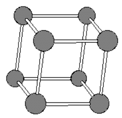
Triangular lattice.

- Each cell has 6 neighbors.
- Ex: surface of graphite.
- Curie temperature: $kT_c \approx 3.641$.



3D cube lattice.

- Each cell has 6 neighbors.
- Curie temperature: $kT_c \approx 4.511$.
- Value only known through Monte Carlo simulation.
- No known analytic solution, and unlikely to exist!



stay tuned (NP-completeness)

Design Snafu

Current design. Does not extend to other lattice structures.

↖ State data type implementation depends on 2D lattice

OOP design.

- Each `Cell` knows its spin and its neighbors.
- Each `Cell` calculates its energy by communicating with its neighbors.
- A `State` is a list of `Cell` objects (in any lattice structure).

↖ or "graph"
(stay tuned)

Bottom line. OOP design makes programs easier to extend.

Other Applications of Ising Model

Applications.

- Spin glasses.
- Protein folds.
- Flocking birds.
- Social behavior.
- Neural networks.
- Beating heart cells.
- Phase separation in a binary alloy.

Variants.

- External magnetic field.
- Different lattice topologies.
- Potts model: spin is one of k discrete values.

Bottom line. With modular design, can reuse code to solve new problems.

Summary

Modular programming.

- Break a large program into smaller independent modules.
- Ex: `Cell`, `State`, `Experiment`, `Plot`, `ProbStat`, `StdDraw`.

Debug and test each piece independently. [unit testing]

- Each class can have its own `main`.
- Spend less overall time debugging.

Divide work for multiple programmers.

- Software architect specifies data types.
- Each programmer writes, debugs, and tests one.

Reuse code.

- Ex: reuse `StdDraw` and `ProbStat` in all kinds of programs.
- Ex: reuse `Experiment` or `Plot` with any scientific experiment.
- Ex: reuse `Cell` for other lattice structures.