Final Project "Tips & Tricks"

COS126 - Spring 2017
No universal acceptance of the atomic nature of matter

Botanist Robert Brown notices erratic motion of pollen grains in water. This motion is later called: Brownian motion.
Einstein publishes a revolutionary paper:

- Brownian motion is caused by smaller moving particles colliding with the larger pollen grains.
- Density of particles affects displacement in Brownian motion.
Jean Baptist Perrin experimentally validated Einstein’s theory and equations.

Your Task: Redo Perrin’s experiments!

Not so difficult with computers and your COS126 skills!
Experiment Overview

1. Record a microscopic video of particles undergoing Brownian motion
Experiment Overview

Convert the video into a set of frames
Experiment Overview

video

Frames

Detect Beads in every frame
Experiment Overview

video → Frames → Beads

4

Compare positions of beads in every two consecutive frames

frame i  frame i+1
Experiment Overview

Record all bead displacements across consecutive frames.
Experiment Overview

video

Frames

Beads

Avogadro’s Number

Displacements

Frames

frame i

frame i+1

7.1833
4.7932
2.1693
5.5287
5.4292
2.1893
5.7294
......

……..
Project Overview

**Given as input**
- Video
- Frames
- Beads

**BeadFinder.java**
Detects all the "Beads" in a given frame.

**BeadTracker.java**
Outputs displacements of beads over successive frames.

**Avogadro.java**
Computes Avogadro's number from a given set of displacements.

**Blob.java**
Represents a set of adjacent pixels.

**readme.txt**
Shows performance analysis.
Project Requirements

Implement the following:

(1) **Blob.java**  
Represents a set of adjacent pixels.

(2) **BeadFinder.java**  
Detects all the “Beads” in a given picture.

(3) **BeadTracker.java**  
Outputs displacements of beads over consecutive frames.

(4) **Avogadro.java**  
Computes Avogadro’s number from a given set of displacements.

(5) **Readme File**  
Shows performance analysis.
(1) **Blob.java**
Represents a set of adjacent pixels.

(2) **BeadFinder.java**
Detects all the “Beads” in a given picture.

(3) **BeadTracker.java**
Outputs displacements of beads over consecutive frames.

(4) **Avogadro.java**
Computes Avogadro’s number from a given set of displacements.

(5) **Readme File**
Shows performance analysis.
**Blob**: Any group of adjacent light pixels.

Adjacency is based on not

How many blobs are there?

Assume each block to be a pixel
**Blob**: Any group of adjacent light pixels. Adjacency is based on not

How many blobs are there?

Assume each block to be a pixel
**Blob**: Any group of adjacent light pixels. Adjacency is based on not.

How many blobs are there?

**Bead**: A blob with a number of pixels that is at least $min$.

How many beads are there? (assume $min=5$)
**Blob**: Any group of adjacent light pixels. Adjacency is based on not.

How many blobs are there?

**Bead**: A blob with a number of pixels that is at least $\textit{min}$.

How many beads are there? (assume $\textit{min}=5$)
public class Blob {
    ...
    public void add(int x, int y) { ... }
    public int mass() { ... }
    public double distanceTo(Blob that) { ... }
    ...
}

- **add** a point to the blob
- **mass** returns the number of points in the blob
- **distanceTo** measures the distance between this and that blob

Distance is measured between the centers of mass (avgX, avgY)
public class Blob {
    public Blob() {
    }
    public int mass() {
    }
    public void add(int x, int y) {
    }
    public double distanceTo(Blob that) {
    }
    public String toString() {
    }
    public static void main(String[] args) {
    }
}

- **Do not store every** added point. We are only interested in the **center of mass** of the points.

- **Checklist** has tips for implementing `toString()` and for handling corner cases.
Project Requirements

Implement the following:

1. **Blob.java**
   Represents a set of adjacent pixels.

2. **BeadFinder.java**
   Detects all the "Beads" in a given picture.

3. **BeadTracker.java**
   Outputs displacements of beads over consecutive frames.

4. **Avogadro.java**
   Computes Avogadro’s number from a given set of displacements.

5. **Readme File**
   Shows performance analysis.
Input:

An Image

A Luminance Threshold $\tau$

BeadFinder.java
After Applying a Luminance Threshold $\tau$

threshold (monochrome luminance $> 180$)
BeadFinder.java

**Input:**
An Image

**A Luminance Threshold** $\tau$

---

**BeadFinder.java**

Constructor stores *all* blobs in the image

Returns through

```java
Blob [] getBeads(int min)
```

all blobs that have at least $min$ points
Detecting All Blobs

For each pixel p:
  If it is white enough
    Create a blob from all its adjacent light pixels

For each pixel p
  If p is white enough AND p is not visited
    Create a new Blob
    Start DFS from p
  Mark p as visited
Detecting All Blobs

For each pixel $p$

If $p$ is white enough
AND $p$ is not visited

Create a new Blob
Start DFS from $p$
Mark $p$ as visited
Detecting All Blobs

For each pixel $p$

If $p$ is white enough
AND $p$ is not visited

Create a new Blob
Start DFS from $p$
Mark $p$ as visited

First pixel is dark
Detecting All Blobs

For each pixel $p$

If $p$ is white enough
AND $p$ is not visited

Create a new Blob

Start DFS from $p$

Mark $p$ as visited

Skip dark pixels and mark them as visited
For each pixel $p$

If $p$ is white enough AND $p$ is not visited

Create a new Blob

Start DFS from $p$

Mark $p$ as visited

Skip dark pixels and mark them as visited
Detecting All Blobs

For each pixel \( p \)

If \( p \) is white enough

AND \( p \) is not visited

Create a new Blob

Start DFS from \( p \)

Mark \( p \) as visited

Skip dark pixels and mark them as visited
Detecting All Blobs

For each pixel $p$

- If $p$ is white enough AND $p$ is not visited
  - Create a new Blob
  - Start DFS from $p$
  - Mark $p$ as visited

This pixel is light and has not been visited before. Iteration in the for loop is suspended and DFS starts.
Detecting All Blobs

For each pixel \( p \)

If \( p \) is white enough AND \( p \) is not visited

Create a new Blob

Start DFS from \( p \)

Mark \( p \) as visited

DFS adds all ‘light’ pixels adjacent to this pixel to a blob. All pixels visited by the DFS are marked as visited.
Detecting All Blobs

For each pixel $p$
- If $p$ is white enough AND $p$ is not visited
  - Create a new Blob
  - Start DFS from $p$
- Mark $p$ as visited

For loop proceeds and ignores this pixel because it is marked as visited

Detected Blobs
Detecting All Blobs

For each pixel $p$

If $p$ is white enough
   AND $p$ is not visited
   Create a new Blob
   Start DFS from $p$
Mark $p$ as visited

light and not visited!
Create a new blob and start a new DFS
Detecting All Blobs

For each pixel $p$
  If $p$ is white enough AND $p$ is not visited
    Create a new Blob
    Start DFS from $p$
    Mark $p$ as visited

A new blob with only one pixel!
Detecting All Blobs

For each pixel $p$

If $p$ is white enough AND $p$ is not visited

Create a new Blob

Start DFS from $p$

Mark $p$ as visited

light and not visited!
Create a new blob and start a new DFS
Detecting All Blobs

For each pixel \( p \)

- If \( p \) is white enough
  - AND \( p \) is not visited
    - Create a new Blob
    - Start DFS from \( p \)
- Mark \( p \) as visited

DFS adds all ‘light’ pixels adjacent to this pixel to a blob. All pixels visited by the DFS are marked as visited.
Detecting All Blobs

For each pixel $p$

If $p$ is white enough
AND $p$ is not visited
Create a new Blob
Start DFS from $p$
Mark $p$ as visited

Algorithm ends when all pixels have been marked as visited
Depth-First Search

DFS starting at \( p \)
Base cases?

Mark \( p \) as visited
Add \( p \) to the blob
Depth-First Search

**DFS starting at** \( p \)

**Base cases?**

- Mark \( p \) as *visited*
- Add \( p \) to the *blob*

- **Pixel out of bounds**
- **Pixel is dark**
- **Pixel is visited**
Number of Blobs is not known ahead of time. What data structure will you use to store them?

- **Array of Blobs?**
  What should the size of the array be?

- **Linked List of Blobs?**
  More implementation work!
  Be careful not to traverse the whole list to add a blob!

- **java.util?**
  Not allowed!

- **Others?**
  You can assume access to Stack.java, Queue.java and ST.java. BUT, make sure to make a choice that is efficient and makes sense!
Images are 640 x 480
Don’t hardwire! Your code should work for any image size.

Private helper methods?
You will definitely need at least one! You can’t do recursion in a constructor!
Implement the following:

(1) **Blob.java**
Represents a set of adjacent pixels.

(2) **BeadFinder.java**
Detects all the “Beads” in a given picture.

(3) **BeadTracker.java**
Outputs displacements of beads over consecutive frames.

(4) **Avogadro.java**
Computes Avogadro’s number from a given set of displacements.

(5) **Readme File**
Shows performance analysis.
Input:
- An Image
- A sequence of images
- A distance Threshold \( \delta \)
- A Luminance Threshold \( \tau \)

Output:
- A list of bead displacements

BeadTracker.java
For every pair of consecutive images $img1$ and $img2$,
Output how much every bead in $img2$ has moved from its position in $img1$.
For every pair of consecutive images $img_1$ and $img_2$:

Output how much every bead in $img_2$ has moved from its position in $img_1$

Displacements:

<table>
<thead>
<tr>
<th>Displacements</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1833</td>
</tr>
<tr>
<td>4.7932</td>
</tr>
<tr>
<td>2.1693</td>
</tr>
<tr>
<td>5.5287</td>
</tr>
<tr>
<td>5.4292</td>
</tr>
<tr>
<td>7.1833</td>
</tr>
<tr>
<td>4.7932</td>
</tr>
<tr>
<td>2.1693</td>
</tr>
<tr>
<td>7.1833</td>
</tr>
<tr>
<td>4.7932</td>
</tr>
</tbody>
</table>
For every pair of consecutive images \texttt{img1} and \texttt{img2} output how much every bead in \texttt{img2} has moved from its position in \texttt{img1}
BeadTracker.java

img1 — input to —> BeadFinder — produces —> Blob [] beads1
img2 — input to —> BeadFinder — produces —> Blob [] beads2
For each bead $b$ in $\text{beads2}$

Find closest bead in $\text{beads1}$

output distance between $b$ and closest
For each pair of images img1 and img2

- img1 —input to—> BeadFinder —produces—> Blob [] beads1
- img2 —input to—> BeadFinder —produces—> Blob [] beads2

For each bead b in beads2

- **Find** closest bead in beads1
- **output** distance between b and closest
For each pair of images **img1** and **img2**

**img1** — input to —> **BeadFinder** — produces —> Blob [] **beads1**
**img2** — input to —> **BeadFinder** — produces —> Blob [] **beads2**

For each bead **b** in **beads2**

Find closest bead in **beads1**
output distance between **b** and closest
For each pair of images \texttt{img1} and \texttt{img2}

\texttt{img1} \rightarrow \texttt{BeadFinder} \rightarrow \texttt{Blob [] beads1}

\texttt{img2} \rightarrow \texttt{BeadFinder} \rightarrow \texttt{Blob [] beads2}

For each bead \( b \) in \texttt{beads2}

Find closest bead in \texttt{beads1}

output distance between \( b \) and closest
For each pair of images img1 and img2

img1 — input to —> BeadFinder — produces —> Blob [ ]
img2 — input to —> BeadFinder — produces —> Blob [ ]

For each bead $b$ in beads2

Find closest bead in beads1

output distance between $b$ and closest

Avoid storing all the images!

Avoid finding beads for the same image more than one

Avoid finding beads for the same image more than one
Project Requirements

Implement the following:

(1) Blob.java
Represents a set of adjacent pixels.

(2) BeadFinder.java
Detects all the “Beads” in a given picture.

(3) BeadTracker.java
Outputs displacements of beads over consecutive frames.

(4) Avogadro.java
Computes Avogadro’s number from a given set of displacements.

(5) Readme File
Shows performance analysis.
Receives as input a sequence of displacements.

Avogadro’s Number is \( N_A = \frac{R}{k} \)

Where \( R \) is given and \( K \) can be computed using:

\[ D = \frac{kT}{6\pi\eta\rho} \]

Where \( T, \pi, \eta \) and \( \rho \) are given and \( D \) can be computed using:

\[ \sigma^2 = 2D\Delta t \]

Where \( \Delta t \) is given and \( \sigma^2 \) is your job to compute!
Final Tips

• Be careful about units. **Convert** every read displacement from **pixels to meters** before using it in any formula.

• **Avogadro** can be implemented and tested **independently**.

• **Constants!** No magic numbers + No cryptic names!

• **Timing Tests!** Read Checklist + Use StopWatch.java + Redirect output to a file.
Image Sources

- Slide 2:

- Slide 3:
  - https://upload.wikimedia.org/wikipedia/commons/d/d3/Albert_Einstein_Head.jpg

- Slide 4: