Cell Deformer Code Sorting Algorithm

Michael Scott
Dr. Amy Rowat's Lab
UCLA Department of Integrative Biology and Physiology

August 8, 2013

Contents

1 Introduction

This readme is intended to explain the sorting algorithm used in the cell deformer code. Although the new sorting algorithm was designed to be easier to follow than the old one, often a little explanation is helpful. The cell detection and image processing portions of the code are not addressed in this readme.

The cell deformer code was originally written by Dr. Bino Varghese starting in October 2011. Dr. David Hoelzle reviewed and updated the code in January 2013. The code was rewritten by Ajay Gopinath, Sam Bruce, and Mike Scott in July 2013 to improve the speed, reliability, and readability. The code was written in and intended for use in Dr. Amy Rowat's lab in the Department of Integrative Biology and Physiology at UCLA.

2 Overview

In the cell deformer code, the sorting and tracking of cell is accomplished in two different functions. The first function is *CellTracking*, which also calls the second function, *ProcessTrackingData*. The two functions have different aims, with *CellTracking* performing a coarse sorting, and *ProcessTracking-Data* accomplishing the tracking of the filtered data. An overview of the sorting and tracking process follows.

2.1 CellTracking

Sorting begins with processed frames from the function *CellDetection*. The frames are logical, and ideally only consist of cells and background. The tracking occurs at eight horizontal lines, the first line is located before the constrictions and is used to calculate the unconstricted area, while the last seven lines occur near the beginning of each constriction. These last seven lines are used to calculate the transit times. *CellTracking* fills the array **cellInfo** with information about the each cell object. The function also performs some filtering to eliminate redundant objects (objects that are detected multiple times at a single line) and objects that for other reasons could not be counted as cells.

2.2 ProcessTrackingData

The array **cellInfo** is passed to *ProcessTrackingData* to track the found cells. This function analyzes the found objects a single lane at a time, with the object of combining multiple cell objects found in different frames into a single cell transiting through the device. In addition to tracking the cells, *ProcessTrackingData* also determines if a cell was alone in the lane during its transit; in instances in which there are two or more cells simultaneously in the lane, all of the cells are marked as paired. The final data output is separated between cells that transited alone (so-called lonely cells) and paired cells.

3 Important Variables

This section identifies variables that are important to the sorting and gives a brief description of their purpose.

3.1 In CellTracking

- **cellInfo**: A (1×16) cell array. The 16 dimensions correspond to the 16 lanes. Each cell contains an array that holds information about each cell object found in that lane. These arrays have 6 columns, which hold the following information:

  1. Column 1: frame number
  2. Column 2: unique cell label number
  3. Column 3: number of the line that the cell intersects
  4. Column 4: cell area (measured in pixels)
  5. Column 5: length of the major axis (in pixels)
  6. Column 6: length of the minor axis (in pixels)

- **laneIndex**: A (1×16) numeric array. Stores the current index to write data to in **cellInfo** for each lane.

- **checkingArray**: An (8×16) numeric array. The 8 corresponds to the each line (1 for area and 1 at each constriction), and the 16 corresponds to the each lane. When a cell object is found at a line and lane, these are used as indexes for **checkingArray**. The frame number the object

is found is stored at the index $(line, lane)$. This is used to check for cells that remain at a line for multiple frames and eliminate redundant entries in **cellInfo**.

- **linesOneAndTwo**: A logical variable that is *true* if a cell is simultaneously touching the unconstricted area line and the line at the first constriction. Allows for counting of cells that do not touch line 1 while not touching line 2 (though the area may be off).

3.2 In ProcessTrackingData

- **contactCount**: An (8×1) numeric array. The 8 entries correspond to the 8 horizontal lines used in tracking, and the array is cleared between lanes. The array stores the number of cell objects that have touched each line in the lane currently being analyzed. **contactCount** is initialized to zeros, and the first entry is updated if an object is found at line 1. For other lines, the object is only counted as a cell if the entry above is greater than the entry at the current line, and the current entry is updated to match the entry above.

- **laneData**: An $(n \times 9 \times 4)$ numeric array. Stores information about cells moving through the device. The variable can be thought of as four separate data sets, as follows:

  1. Time data: Columns 1 through 8 contain the frame at which the cell object first touched each line, and column 9 stores whether or not the cell is paired.
  2. Area data: Columns 1-8 contain the area data for each line, and column 9 contains the lane the cell was found in (for debugging).
  3. Diameter data: Contains a crude "diameter", defined as the average of the major and minor axes of the cell object.
  4. Eccentricity data: contains the eccentricity of the cell object at each constriction.

- **unpairedTransitData**: An $(n \times 9 \times 4)$ numeric array that stores data in the same manner as **laneData**, but is only contains the unpaired (lonely) cells. The frame data in dimension 1 is also converted into Δt values for each lane. The first column stores the total transit time, the

second column stores the unconstricted area, and the rest store the Δt values. item **pairedTransitData**: An $(n \times 9 \times 4)$ numeric array that is exactly the same as **unpairedTransitData** except that it contains only paired cells.

4 Sorting in CellTracking

As previously mentioned, *CellTracking* primarily tries to sort the cells into lanes and reduce the number of redundant entries in **cellInfo**. Redundant entries are defined as those where a cell remains at a line for multiple frames, or times where a cell object (often a "blip") is found where it cannot possible be counted as a cell. For example, if a cell object is found at line 8, but not cells has been found at line 7 in that lane, it cannot be counted as a cell and is ignored.

4.1 Function Overview

This section is a skeleton code of sorts for *CellTracking*. Points which are likely to be confusing are expanded upon, and others are glossed over if the code is relatively simple or straightforward. The code goes through each frame sequentially.

1. Objects in the current frame are labeled, and their centroid as well as other geometric properties are calculated.

2. The line the object intersects with is found. This is done backwards (from 8 to 1), so the object is counted at the furthest line it is touching. If the object is found at line 2, the code checks if it also touches line 1 and sets **linesOneAndTwo** accordingly. Cells not touching any lines are ignored.

3. The lane the cell is in is calculated from the centroid

4. **checkingArray** determines if the cell is stored or not. If a cell was at the same line in the previous frame, the code assumes they are the same cell and ignores the current object. Otherwise the cell is stored. If the cell touches lines 1 and 2 simultaneously, it is stored twice, once at each line.

The code also checks if the array in **cellInfo** is filling, estimates the number of additional rows necessary, and increases the size of the array. This allows preallocation, and reduces the number of concatenation operations in the case that many cells are found in a lane. After all of the frames have been processed, **cellInfo** is passed to *ProcessTrackingData*.

5 Sorting in ProcessTrackingData

ProcessTrackingData takes **cellInfo** as an input, and tracks the cells through the device. This function goes through the data by lane, though the data is later compiled in the main body of the code.

5.1 Function Overview

As with **cellTracking**, a brief skeleton of the code follows.

1. First, **checkingArray** is checked to see if any entries in the current column (lane) are zero. If so, no cells transited through the device in this lane, and no analysis is carried out.

2. If the current object is at line 1, and the previous cell (if one exists) has passed to at least line 2, the cell is stored in **laneData** at the current index specified by **contactCount** as well as the lane it is in.

3. If the cell is not at line 1, and fewer cells have reached the current line than the previous line, store the cell in **laneData** at the current index specified by **contactCount** at the same row corresponding to the current line the cell is touching.

4. Repeat for all of the cells in the lane.

5. Eliminate any cells with a zero for the eighth line transit frame. Having a zero means they did not transit fully and will not be counted as a cell. This occurs near the end of the video, or when a cell is lost by the detection algorithm partly through the transit.

6. Check **laneData** for the frame at which each cell hits line 8. Then check sequential cells to see if the cell should be paired. If it should be paired, continue checking (up to the next 6 cells) and pairing as

necessary. For the pairing, 1 denotes a lonely cell, and 2 denotes a cell that did not transit alone (but does not tell how many other cells were concurrently in the lane—perhaps in the future).

7. Separate all the unpaired and paired cells in the lane into the first and second rows of the cell array **trackingData**, respectively.

8. Concatenate the unpaired and paired transit data from all lanes into **unpairedTransitData** and **pairedTransitData**, and then convert the frames in dimension 1 into Δt values.

6 Known Issues

This section describes some known issues with the tracking code, and describes why they arise as well as why they are an issue. Potential solutions may be mentioned. Few known issues are purely caused by the tracking algorithm, and many are a function of the detection and tracking algorithms combined.

- **Cell Blobbing**: The most common problem, in which cells are close together around line 1 and are counted as one cell. Several related issues arise, and are detailed below.

- **Splitting Blob**: Since a blob must only be detected in one lane, if it breaks apart and goes into two lanes sometimes one of the cells is not counted since it skipped line 1. This could be improved is the goal is to count the objects, however our most important statistic is the first transit time, so this issue remains unresolved. Perhaps if the cells could be segmented, the tracking could be improved.

- **Closely Spaced Cells**: This issue is often caused by a blob, but can also be arise when two cells are separated by less than one constriction. Cells that are closely spaced (the previous cell has not reached the next line) are not separated. This could be resolved by changing the restrictions on **checkingArray**, but would also add in redundant objects (essentially rendering **checkingArray** useless). Blips would also be more of an issue.

- **Switching lanes**: at the top of the constriction, sometimes cells switch from one lane to another. Rarely, and generally if the cells are very large, they can occupy both lanes at once until they hit line 2. These cells will not be counted.

- **Skipping Lines** Cells occasionally "miss a line". This is often because of motion blurring, and is largely a function of the video contrast and frame rate, as well as the detection algorithm. This issue can be alleviated by taking the videos at a higher frame rate, with shorter exposure, or by optimizing the image processing algorithms for the particular cell line and lighting conditions used.

- **Extremely Large Cells**: Extremely large cells are often very slow moving, and cause the next problem. Also, they can touch multiple lines simultaneously (though this issue has been largely resolved). They may not be paired correctly, as they are counted only at their last line (they can be marked fully transited because their leading edge has reached line 8, but their trailing edge remains in the channel).

- **Extremely Slow Cells**: Cells that are extremely slow can be eliminated by the background subtraction algorithm in *cellDetection*. These will not be counted. In our analysis, these cells rarely fully transit, so it has not been a huge issue.