Detection and Tracking of Multiple Circular Objects†

Dan Daniel[a‡] and Daniel Newman[a‡]

Accurate detection and tracking of multiple circular objects in images is a challenging computer vision problem encountered in many different contexts. Here, we propose using 1) a modified, gradient-based Circular Hough transform (CHT) to accurately detect multiple circles of different radii in a noisy image and using 2) a linear program (LP) in conjunction with the simplex method to track different circles from one frame to the next. For circle detection, we compare our implementation of CHT to MATLAB's native implementation, as well as a maximum likelihood estimation-based (MLE-based) method. For object tracking, we compare the performance of our implementation of the simplex algorithm to the GLPK solver as well as to both our own and another packaged implementation of the Hungarian algorithm. Combining the CHT and LP methods allows us to detect multiple circles of different radii in the presence of noise. To demonstrate our algorithms' effectiveness, we succesfully tracked the condensation and the subsequent movement of water droplets on a flat surface.

1 Introduction

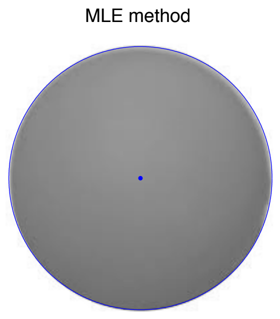


Fig. 1 A circular object detected using linear least-square method. Image taken from Google image.

The detection and subsequent tracking of multiple circular objects is an important computer vision problem which crops up in various different contexts, such as 1) tracking biological cells, 2) iris recognition for biometric identification, 3) particle tracking velocimetry to study fluid flow and 3) tracking water droplets in dew formation. In most cases, the images are noisy and the algorithm implemented must be able to detect relevant circular objects in the presence of extraneous image features (e.g. other facial features in iris recognition).

A digital image can be thought of as a matrix, with each pixel value represented by an entry in the 2D matrix array, which typically ranges from 0 to 255 for an 8-bit image. Therefore, many matrix operations can be adapted to perform various image manipulations: numerical differentiation, for example, can be used to detect edges, where there are large changes in pixel value. One approach to detecting circles is to apply a least-square fit to the edge points to find the center and the radius of the circle[1]. This is an example of a maximum likelihood estimation-based (MLE-based) method. Fig. 1 demonstrates how a linear least-square fit can be used to detect a circle following gradient-based edge detection, by noting that the equation of a circle can be written as $b_0 + b_1 x + b_2 y = x^2 + y^2$, such that $b_0 = r^2 - a^2 - b^2$, $b_1 = 2a$, and $b_2 = 2b$, where (a, b) and r are the center and radius of the circle. While MLE-based methods can be used succesfully to detect a single circle as demonstrated in Fig. 1, it is not clear how to extend this method for the detection of multiple circles and/or in the presence of spurious object features.

On the other hand, voting-based methods such as the Circular Hough Transform (CHT), can easily be used to detect multiple circles in a noisy image. There are many variants and extensions of CHT[2,3], but the main idea is the same: for a circle that can be described by the equation $(x-a)^2 + (y-b)^2 = r^2$, any arbitrary edge-point (x_i, y_i) will be transformed into a subspace in the (a, b, r) accummulator space. Some voting scheme is implemented in the accummulator space such that the maximum point corresponds to the parameters of the circle. We demonstrate the efficacy of CHT method by using it to detect locations of circular water droplets condensing on a surface for a series of images.

After the the locations and radii of the circles have been found, we then setup a linear program (LP) to help us track droplets from frame to frame. We do this by pairing one droplet from one frame to another droplet in the next frame, such that the total distance between paired objects is minimized. We implemented two methods to find the optimal solutions to the LPs: the "Big M" variant of the simplex method and the Hungarian algorithm. We found that both algorithms found optimal solutions in polynomial time but that the simplex method was slightly quicker overall.

There are many other methods that have been employed to

† Supplementary Information available.
[a] *SEAS, Harvard University, Cambridge, MA 02138,USA*
‡ These authors contributed equally to this work

track the motion of objects in a video. For example, the commonly used Lucas-Kanade[4] and Horn-Schunck[5] methods both use differential techniques to identify motion in videos. The combination of CHT and LP is well suited to the problem of tracking water droplets since these methods allow us to track many moving objects in a frame at once and because in our case, we can assume that droplets will typically have circular shapes.

2 The proposed algorithms

2.1 Circle detection

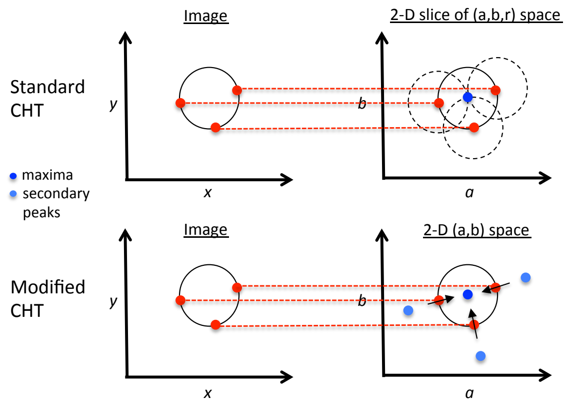


Fig. 2 Top, in the standard version, Circular Hough Transform (CHT) converts circles in the (x,y) space into circular cones in the (a,b,r) accummulator space. Bottom, in our implementation, the gradient of the image (black arrows) are used to limit 'voting' in the direction parallel and anti-parallel to the gradient vector in the (a,b) space.

In the standard version of CHT, every edge point (x_i,y_i) of a circle centered about (x_c,y_c) and radius r_c is transformed into right circular cones in the (a,b,r) accummulator space for $r>0$, with its apex at $(x_i,y_i,0)$. The different circular cones will then intersect at a single point (x_c,y_c,r_c) corresponding to the parameters of the circle. This is illustrated in Fig. 2 (top). The filled black line depicts the edge of the circle, while the dashed black lines are circular slices of the cones in the $r=r_c$ subspace of the three-dimensional accummulator space, centered about each corresponding edge point (x_i,y_i) (red dots). There is a maximum at (x_c,y_c,r_c) (blue dot) corresponding to the parameters of the circle.

Since, for a given $r=r_0$, the locus of the right circular cone on the (a,b) plane is simply a circle of radius r_0, it is sufficient to increment all the edge points for each value of radius on the (a,b) space alone. This is illustrated in Fig. 2 (bottom), where the accummulator space is the two-dimensional (a,b) space, with its third dimension r flattened out. In our implementation, the gradients at the edge points (black arrows) are used to limit the transformation in the parallel and anti-parallel directions of the gradient vectors. This method greatly reduces the memory requirements of the accummulation array.

```
Data: Image file in grayscale
Result: Centers and radii of circles with radius range
        $[r_{min}, r_{max}]$
1) Let $I(x,y)$ be the pixel value of the image;
2) Let $\tilde{I}(a,b)$ be the corresponding accummulator space,
reset to zero initially;
3) Calculate $\nabla I(x,y)$;
4) Define edge points $(x_i,y_i)$ for $|\nabla I| > val$;
5) Discretize the radius range into $r_1, r_2, ..., r_n$ ;
6) for i=1:end do
    for j=1:n do
        $a = x_i + \nabla I(x_i,y_i)_x r_j / |\nabla I(x_i,y_i)|$,
        $b = y_i + \nabla I(x_i,y_i)_y r_j / |\nabla I(x_i,y_i)|$;
        Increment $\tilde{I}(a,b)$ by $\nabla I(x,y)$;
        $a = x_i - \nabla I(x_i,y_i)_x r_j / |\nabla I(x_i,y_i)|$,
        $b = y_i - \nabla I(x_i,y_i)_y r_j / |\nabla I(x_i,y_i)|$;
        Increment $\tilde{I}(a,b)$ by $\nabla I(x,y)$;
    end
end
7) Smooth $\tilde{I}(a,b)$ and find local maxima in $\tilde{I}(a,b)$. Local
maxima correspond to circles' radii;
8) For each circle center, find the corresponding radii
$r = \{r_j\}$ for $j \in [1,2,...,n]$ that contribute to the maxima in
$\tilde{I}$ by using a look-up table method. The radius of the circle
is approximated by taking the mode of $r$;
```

Algorithm 1: Modified CHT

Algorithm 1 outlines our implementation of CHT in MATLAB. To speed up the computation times, step 6 was vectorized to avoid the costly FOR loop in MATLAB. For step 7, the smoothing was performed by using a 5×5 averaging matrix, and the multiple local maxima were found by either using a multi-start method or by a code developed by Aguilera[6] which look for the change in the parity of $\nabla\tilde{I}$ to identify positions of extrema. See Figs. S1 and S2 for more information.

2.2 Object tracking

Once circular objects have been detected, we tracked them from one frame to the next using optimization techniques that match up objects in two frames by minimizing the sum of distances between the paired objects. To do this, we first set up a square cost matrix containing the distances from each object in the first frame to each object in the second.

Next, we try to pick one assignment from each row of the cost matrix such that the overall "cost" is minimized. Using brute force methods, each combination must be tried, such that the time complexity is $O(n!)$. The processing time to find the minimum solution quickly scales up as the number of objects increases, so we implemented two optimization techniques to find an optimal solution more quickly: the Hungarian results and the simplex method. Both of these were implemented using Python 2.7.

Our implementation of the Hungarian algorithm is based on the technique first developed by Kuhn[7] and as outlined by Flood[8] and Munkres[9]. We also borrowed from ideas discussed by Srinivasan[10]. This method allows the assignment problem to be solved in polynomial time, which provides a vast speedup over brute force methods. Fig. 3 shows an example of the results of running the Hungarian algorithm on two sets of points. The first set of points were chosen randomly on intervals from 0 to 1000 on both axes. The second set of points are based off of the first set, but are offset on each axis by a random number between 0 and 75. As Fig. 3 shows, the algorithm matches up the two sets of points in a manner that minimizes the total length of the lines drawn.

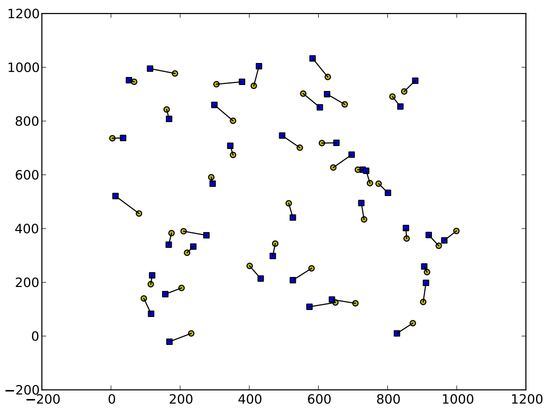


Fig. 3 Two sets of points are matched up by minimizing the total sum of distances between the corresponding points in the two sets.

We next implemented the simplex algorithm for our problem in order to compare its results with the Hungarian method. Our implementation of the simplex method utilizes the Big M method as described by Hillier and Lieberman[11].

To use the simplex method, we first formulated the problem as a linear program with the objective function:

$$minimize \sum_i \sum_j c_{ij} x_{ij} \tag{1}$$

And the following constraints which ensure that only one assignment is made in each row and each column of the assignment matrix:

$$\sum_{i=1}^{m} x_{ij}, \; for \; j = 1, ..., m \tag{2}$$

```
Data: An $m$-by-$n$ cost matrix, $A$ (where $m=n$)
Result: An $m$-by-$n$ assignment matrix (where $m=n$)
        containing a 1 in each row that represents object $m$
        being paired with object $n$
1) Subtract the minimum value in each row in $A$ from all
values in that row, such that each row contains at least one
zero;
2) Find a minimal set of lines which contain all the zeros
in $A$. if the number of lines equals $n$ then
    go to step 5
end
3) Let $h$ denote the smallest element of $A$ that is not
crossed out by any line in step 2. Add $h$ to all crossed out
elements (add twice if an element is crossed out twice).
Then, subtract the minimum value in $A$ from every value in
$A$;
4) Repeat steps 2 through 5 until the stopping condition in
step 2 is met;
5) Select one and only one 0 in each row such that each
column also has just one zero circled. Return an $m$-by-$n$
assignment matrix (where $m=n$) containing all 0's except
for a 1 in each column where there are 0's circled in A;
```
Algorithm 2: Hungarian Algorithm

$$\sum_{j=1}^{n} x_{ij}, \; for \; i = 1, ..., n \tag{3}$$

$$x_{ij} \geq 0 \tag{4}$$

Although the decision variables must be binary, it is not necessary to add this as a constraint, since the initial tableau only contains integers in our problem's formulation (outside of the row containing the objective function). Therefore the optimal solutions already assign non-negative integer values to the decision variables.

As discussed by Cormen, Leiserson, Rivest and Stein[12], the simplex algorithm runs in exponential time in the worst case scenario. However, the worst case scenario is rare and in most cases, the algorithm solves linear programs in polynomial time.

3 Results and discussions

3.1 Circle detection

For the initial test, we performed the CHT algorithm on the image file "coins.png" that ships with MATLAB. Fig. 4 shows the magnitude of the gradient $|\nabla I|$ and the accummulator array values $\tilde{I}$ as described in Algorithm 1. Fig. 5 shows the results of circle detection using our implementation of CHT and MATLAB's "imfindcircles" function.

Next, we ran the code to detect circles from a series of images of water droplets condensing on a surface. In this case,

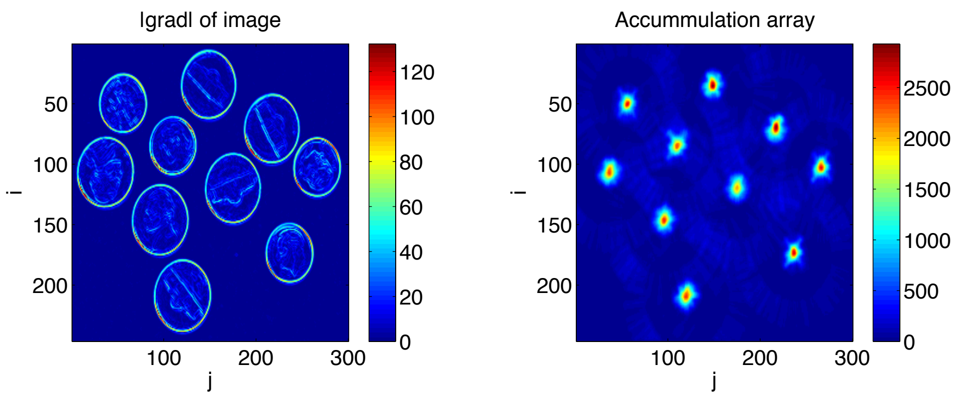


Fig. 4 Left, magnitude of the gradient of “coins.png”. Right, the corresponding accummulator array values

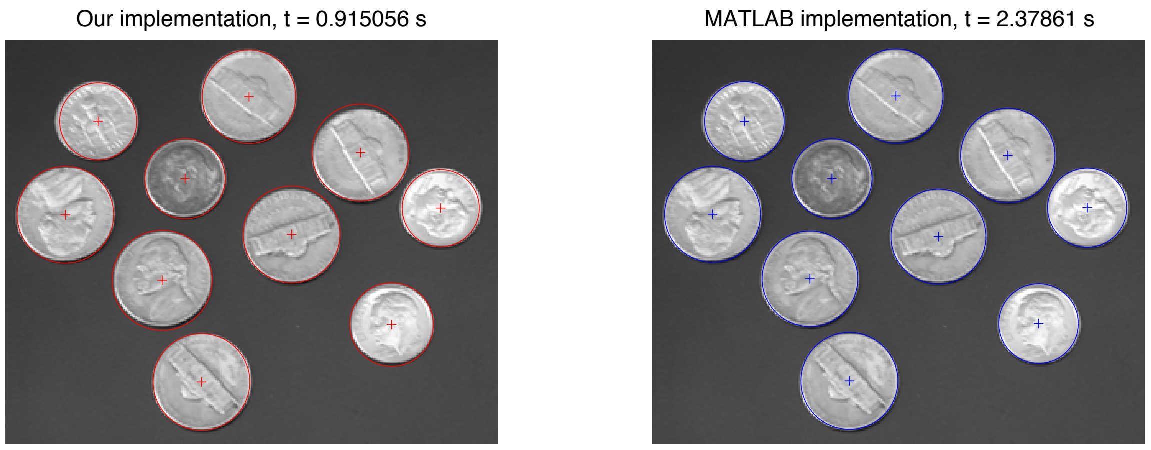
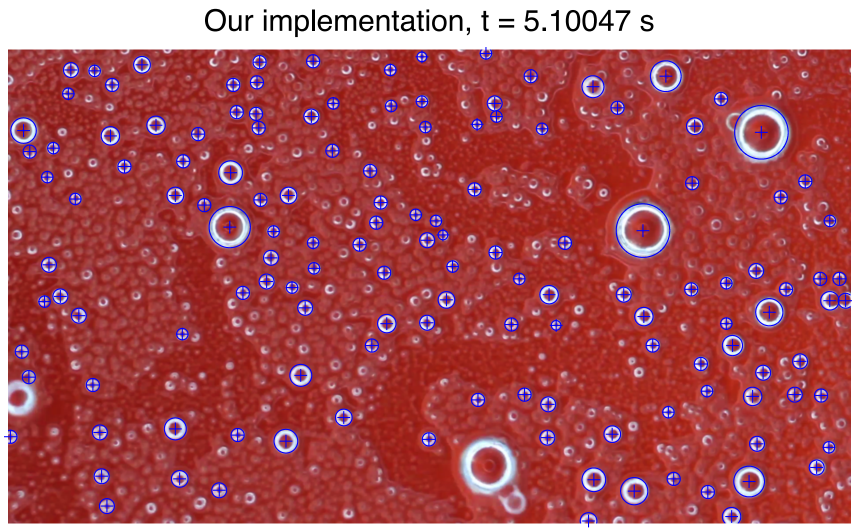

Our implementation, t = 0.915056 s
MATLAB implementation, t = 2.37861 s

Fig. 5 Compares the results of circle detection using our own implementation of the Hough transform and MATLAB’s native implementation “imfindcircles”, with the corresponding computation times of 0.9 and 2.4 s.



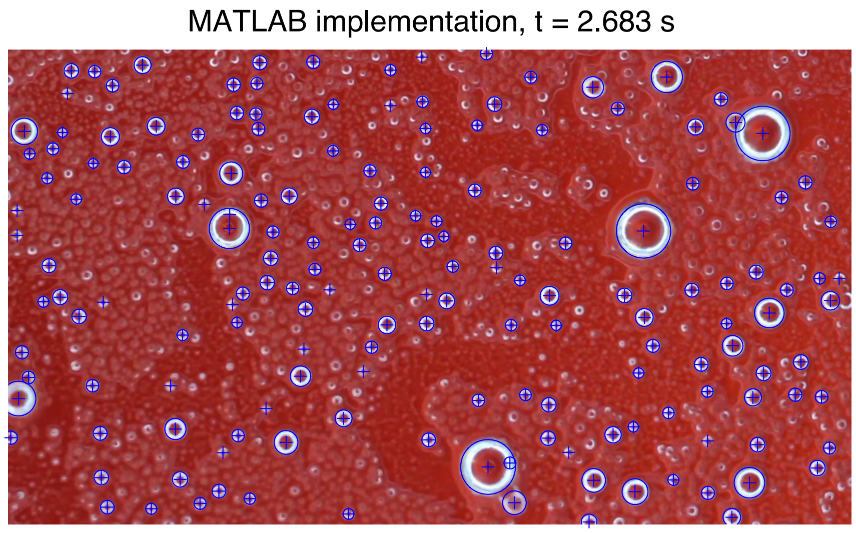


Fig. 6 Compares droplet detection using our implementation and MATLAB's native implementation of CHT.

there are many extraneous features on the images, but our CHT algorithm is still able to detect the droplets accurately. See Fig. 6 for a representative result. To detect only the outer ring and not the inner ring of each droplet, we can use the fact that the ∇I points in the opposite directions for the inner and outer edges. See Figs. S3 and S4 for more information.

To compare the computation times between the two implementations, we ran the codes for an $n \times n$ image, for n between 540 and 3780. Fig.7 summarizes the results. For large n, the computational complexities are $O(n^3)$ and $O(n^{2.5})$ for our implementation and MATLAB's.

For most medium-sized images ($n < 1000$), CHT will complete within a few seconds.

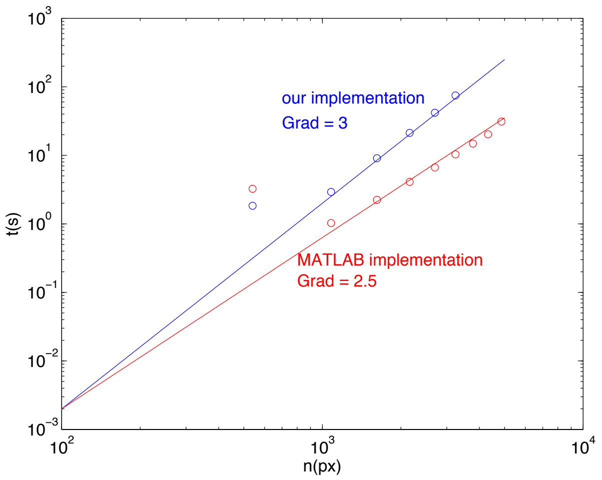


Fig. 7 log-log plot of the computational times for our and MATLAB's implementation of CHT for $n \times n$ image.

3.2 Object tracking

To test the efficiency of our optimization algorithms, we ran them on randomly generated $n \times n$ matrices and compared the timed results against open-source implementations of the algorithms. Fig. 8 shows the results of our implementation of the Hungarian algorithm against the Munkres package available for Python[13]. As the figure shows, our implementation takes a comparable amount of time to find an optimal solution for a given matrix.

Fig. 9 shows the results of our implementation of the simplex method against the implementation that incorporates the Pulp package for Python[14], which uses GLPK[15] as its default solver. As the matrix size increases, the Pulp performs significantly better than our implementation. This result is not surprising, since the GLPK solver is implemented in C, which generally runs much quicker than Python. Nonetheless, both implementations show polynomial time increases as the matrix

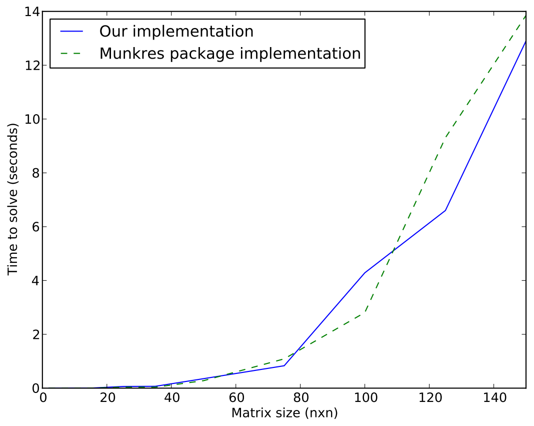


Fig. 8 Our implementation of the Hungarian algorithm versus the Munkres package for Python.

sizes grow larger. At smaller matrix sizes, our implementation actually runs a bit quicker, which is most likely due to the overhead of the Pulp package as it sets up the linear program. Since our version of the simplex algorithm is made to solve our specific linear program (unlike Pulp, which is a wrapper for general purpose LP/MIP solvers), our overhead in setting up the problem is likely to take up less time.

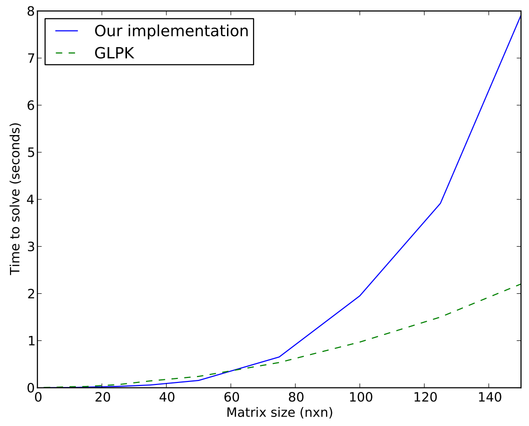


Fig. 9 Our implementation of the simplex method versus the GLPK solver implemented with the use of Python's Pulp package.

Both the Hungarian algorithm and the simplex method are guaranteed to find optimal solutions that minimize the total distance between the two sets of points. As shown in Fig. 10, our implementation of the simplex algorithm ran slightly quicker than our implementation of the Hungarian algorithm, so we decided to apply this method to the tracking of water droplets between frames.

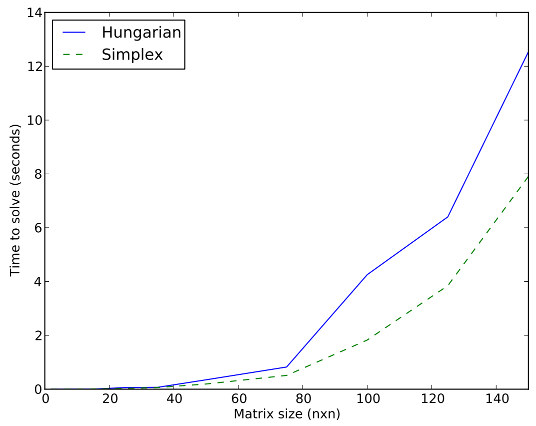


Fig. 10 Our implementation of the simplex method versus the GLPK solver implemented with the use of Python's Pulp package.

In order to effectively track the movements of water droplets, our program must often match up coordinate sets of different sizes. Differences in set sizes between two frames may result because a moving droplet enters or exits the frame; if two droplets merge together; or if the detection algorithm detects a new forming droplet or fails to detect a droplet that it previously detected. To handle these, we insert "dummy" points into the set that contains fewer elements. These dummy points are assigned arbitrarily high coordinate values such that they are located well outside the frame of the picture. Still, a coordinate at any finite location will be closer to some points in the frame than others, which can lead to the optimization algorithms incorrectly matching up droplets between frames. To handle this, we set a threshold for our cost matrix such that all points that are separated by a distance exceeding this threshold are given the same arbitrarily high corresponding value in the cost matrix. Then, when extracting assignments after the simplex algorithm is finished, we disregard all points that match up to other points that are separated by a distance above the threshold. By doing this, if two points in one set are near just one point in the other set, the closer of those points will be matched up to the point in the other set and the other point will simply not get a match.

Additionally, we decided to only track droplets whose radii exceed a certain threshold. A visual inspection of the images shows that only larger droplets tend to move around, so this criteria rarely affects our program's ability to track movements. Additionally, since there are fewer droplets to detect if we disregard smaller ones, this approach will tend to match up droplets more quickly and accurately.

Fig. 11 shows a screenshot from the object tracking program that we wrote. The trails indicated by the yellow paths show where droplets have moved up to this point in the animation.

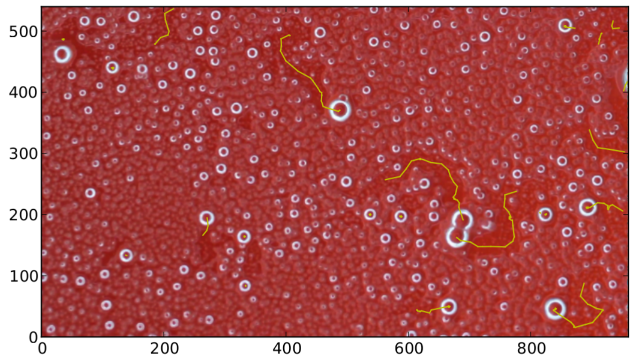


Fig. 11 A screenshot of our program's animation shows the detected paths of droplets. The yellow trails show the droplets' paths so far. More screenshots can be seen in the supplemental materials that we have included at the end of this report. To see the animation, run the script track_droplets.py.

When watching the animation, one can observe that this tracking method is more effective for droplets in the middle of the frame than for droplets around the edges of the image, since the detection algorithm is not made to detect semicircles from droplets that straddle the edges of the image, such as that belonging to the large droplet whose edge appears near the upper right corner of Fig. 11. Nonetheless, as a look at the animation generated by the script "track_droplets.py" shows, the optimization methods that we implemented are relatively effective overall for tracking circular objects between frames.

4 Conclusions

Using a combination of Circular Hough Transform and Hungarian/Simplex algorithm, we are able to detect and track the motion of circular objects even in the presence of noise. This method was generally successful for tracking the movement of water droplets, but is by no means the only method for doing so. One area where it falls short is with detecting and tracking droplets that are straddling the edge of the image and with detecting droplets as they temporarily lose their circular shapes when they are merging together. Future refinements of these algorithms might address these issues to make the detection and tracking processes more robust.

5 Running Our Code

The code for the CHT was written in MATLAB. The code for the optimization algorithms and for droplet tracking was written in Python (2.7). Most of the files are named after the figures in this report that they were used to generate. For more details and for instructions on running our code, see the readme.txt file included with our submission.

References

1 http://iacs-courses.seas.harvard.edu/courses/am205/assignment_1.pdf.

2 L. Pan, W.-S. Chu, J. M. Saragih, F. De la Torre and M. Xie, *Signal Processing Letters, IEEE*, 2011, **18**, 639–642.

3 H. K. Yuen, J. Princen, J. Illingworth and J. Kittler, Proc. 5th Alvey Vision Conf., Reading (31 Aug, 1989, pp. 169–174.

4 B. Lucas and T. Kanade, Proc. of Imaging Understanding Workshop, 1981, pp. 121–130.

5 B. Horn and B. Schunck, *Artificial Intelligence*, 1981, **17**, 185–203.

6 http://www.mathworks.com/matlabcentral/fileexchange/12275-extrema-m-extrema2-m.

7 H. Kuhn, *Naval Research Logistics Quarterly*, 1955, **2**, 83–97.

8 M. Flood, *Operations Research*, 1956, **4**, 61–75.

9 J. Munkres, *Journal of the Society for Industrial and Applied Mathematics*, 1957, **5**, 32–38.

10 G. Srinivasan, 2009, http://www.youtube.com/watch?v=BUGIhEecipE.

11 F. Hillier and G. Lieberman, *Introduction to Operations Research, 7th Edition*, Mcgraw Hill, 2001.

12 T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms, 3rd Edition*, MIT Press, Cambridge, 2009, pp. 843–897.

13 http://software.clapper.org/munkres/.

14 https://code.google.com/p/pulp-or/.

15 http://www.gnu.org/software/glpk/.