Data in Brief 22 (2019) 605–619

Contents lists available at ScienceDirect

Data in Brief

journal homepage: www.elsevier.com/locate/dib

Data Article

# Image and data processing algorithms for identifying cell-bound membrane vesicle trajectories and movement information

Ye Xu [a], Wendiao Zhang [a], Yong Chen [a,*], Wenzhe Shan [a,b,**]

[a] Nanoscale Science and Technology Laboratory, Institute for Advanced Study, Nanchang University, Nanchang, Jiangxi 330031, PR China
[b] Department of Civil Engineering, Nanchang University, Nanchang, Jiangxi 330031, PR China

ARTICLE INFO

*Article history:*
Received 24 November 2018
Received in revised form
20 December 2018
Accepted 21 December 2018
Available online 28 December 2018

ABSTRACT

This DIB article provides details about the trajectory identification and data processing algorithms used in the article "Dynamic single-vesicle tracking of cell-bound membrane vesicles on resting, activated, and cytoskeleton-disrupted cells" (Zhang et al.) [1]. The algorithm identifies vesicles on cell membranes from series of undyed grayscale images captured by the confocal microscope based on contrast differences and then trajectories of vesicles are obtained by analyzing their positions in consecutive images. Once the trajectories have been obtained, other quantitative movement information, such as moving speed, direction and acceleration, are derived by standard dynamic relations.



DOI of original article: https://doi.org/10.1016/j.bbamem.2018.10.013
* Corresponding author.
** Corresponding author at: Nanoscale Science and Technology Laboratory, Institute for Advanced Study, Nanchang University, 999 Xuefu Ave. Honggutan District, Nanchang, Jiangxi 330031, PR China.
*E-mail addresses:* tychen@ncu.edu.cn (Y. Chen), shan@ncu.edu.cn (W. Shan).

https://doi.org/10.1016/j.dib.2018.12.076

**Specifications table**

| | |
|---|---|
| Subject area | *Biology, Image processing* |
| More specific subject area | *Point-like feature identification and trajectory tracing* |
| Type of data | *Images, Computer algorithms, Source code* |
| How data was acquired | *Confocal Microscope, In-house MATLAB program* |
| Data format | *Analyzed* |
| Experimental factors | *The image processing algorithms introduced are implemented in MATLAB. The raw images processed by the program are undyed grayscale images captured by the Confocal microscope.* |
| Experimental features | *In-house developed image processing algorithm for identifying positions and trajectories of undyed point-like features from series of raw images, as well as obtaining their moving status (velocity, moving range, etc.).* |
| Data source location | *Nanchang, Jiangxi Province, China* |
| Data accessibility | *Algorithms, equations and codes are presented in this article.* |
| Related research article | *W. Zhang, Y. Xu, G. Chen, K. Wang, W. Shan, Y. Chen, Dynamic single-vesicle tracking of cell-bound membrane vesicles on resting, activated, and cytoskeleton-disrupted cells, BBA-Biomembranes, 1861(1):26–33* [1]. |

**Value of the data**

- Algorithms and source code are provided and can be used for identifying multiple point-like features on grayscale raw images (dyeing is not required for the experiment).
- Algorithms and source code are provided and can be used for identifying the moving trajectories of multiple point-like features from series of grayscale raw images (dyeing is not required for the experiment).
- Algorithms and source code are provided and can be used to calculate the movement information, e.g. velocity and moving range, from identified trajectories of moving point-like features.

## 1. Data

Three types of data are used in the article [1] for analyzing the moving status of vesicles on cell membranes: mean velocity, moving range (displacement) and population of vesicles on different cell samples. These three types of data are obtained from their positions and trajectories (**TRJ**) extracted from series of raw images captured by the confocal microscope (**CM**). This report presents algorithmic and implementation details about our **in-house developed programs** using MATLAB, which is used for extracting the TRJs from unmarked raw images, as well as obtaining the abovementioned data from the TRJs. The following key algorithms are explained in the next section: vesicle-identification, trajectory-tracing and postprocess for additional movement information. Source codes of corresponding algorithms are provided (code 1–code 14), as well.

## 2. Experimental design, materials and methods

### *2.1. Algorithm for the vesicle-identification*

First, the pixel data are extracted from raw images that are in the RGB format, where each pixel contains three color-data [red, green, blue], ranging from 0 to 254. Since images captured by the CM are stored in the grayscale-style, the three color-data are identical for each pixel and we can take an arbitrary one for the image processing. The source code for this step is given in Code 1.

**Code 1: Function for reading pixel data from raw images**

```
function group = read_img(filenames, Lx, Ly)
////////////////////////////////////////////////////////////////
// inputs:
// filenames: filenames of raw images
// Lx, Ly: image size (pixels); all images must be of the same size Lx X Ly
// output:
// group: a 3D matrix storing the pixel data of raw images
////////////////////////////////////////////////////////////////
       // identify number of images
       nimg = length(filenames);
       // allocate memory for image data
       group = zeros(Lx, Ly, nimg, 'uint8');
       for i = 1:nimg
              // read pixel data from a single image (RGB)
              pic = imread(filenames{i});
              // storing the pixel data in group
              group(:, :, i) = pic(:, :, 1);
       end
end
Special MATLAB Functions:
imread(): Image Processing Toolbox.
```

After the raw data extraction, the so-called Laplacian of Gaussian (**LOG**) algorithm, which is a standard image-processing algorithm for boundary identification, is used to isolate pixel groups corresponding to vesicles. The raw image data corresponds to redness (brightness in red) of each pixel and the LOG values of pixels corresponding to vesicles will be significantly higher than the rest of the image. Therefore, a lower bound can be used to isolate the pixels corresponding to vesicles from the rest of the image. Such lower bound is obtained by comparing the filtered results with human-eye observations in a try-and-error manner. The filtered image data is than normalized back into the grayscale format, so that each pixel will be assigned a value ranging from 0 to 1.After the normalization, the pixel data is then converted into the black-and-white style so that those corresponding to vesicles will have the value of 1 and the rest pixels will have the value of 0, ready for the next step. The additional lower-bound filter mentioned above also guarantees the boundary for vesicles which are close to each other will have their boundaries set to zero. The source code for this step is given in Code 2.

**Code 2: Function for applying the LOG filter to pixel data of a single image**

```
functiondat = LOG(dat, n, Lmin)
////////////////////////////////////////////////////////////////////////
// inputs:
// dat: pixel data of a single raw image
// n: standard deviation for the Gaussian filter, which approximately
translates into the radius of vesicles in pixels
// Lmin: lower bound for isolating vesicles
// output:
// dat: filtered pixel data
////////////////////////////////////////////////////////////////////////
       // apply the Gaussian filter to the raw pixel data
       dat = double(imgaussfilt(dat, n));
       // compute the brightness gradient of the filtered data
       [gx, gy] = gradient(dat);
       // compute the divergence of the gradient (Laplacian)
       dat = divergence(gx, gy);
       // applying the lower bound to isolate vesicle pixels
       dat(dat > -10) = 0;
       // normalize the LOG values into grayscale style
       dat = abs(dat);
       dat = dat./max(dat(:));
       // convert the LOG values into back-and-white style
       dat = im2bw(dat, 0.01);
end
Special MATLAB functions:
imgaussfilt(): Image Processing Toolbox
gradient(): MATLAB Built-in Function
divergence(): MATLAB Built-in Function
```

After applying the LOG filter and converting the pixel data into the black-and-white style, the filtered pixel data will look like a group of white blobs with approximately the same size

corresponding to the vesicles in a black background, a recursive algorithm which checks the neighboring pixels is then applied to identify and store them in a more organized format and their centers are then computed and used as the positions of vesicles which will be used later for the trajectory-identification and postprocess. In addition to center coordinates, the size of each blob (number of pixels) will also be counted so that and additional lower bound can be defined to filter out noise blobs whose size are significantly smaller than actual vesicles. Such lower bound should be approximately the number of pixels of the smallest vesicle, which is identified by human-eye observations. The source code for this step is given in Code 3.

**Code 3: Function for identifying vesicles from LOG filtered pixel data of a single image**

```
function cen = find_blob(dat)
////////////////////////////////////////////////////////////////////////////
// inputs:
// dat: LOG filtered pixel data
// output:
// cen: centers of blobs (vesicles)
////////////////////////////////////////////////////////////////////////////
      // identify image size
      [m, n] = size(dat)
      // initialize counter for vesicles
      count = 0;
      kind  = zeros(m, n);
      // find positions of white pixels
      [x, y] = find(dat > 0);
      // number of white pixels
      np = size(x, 1);
      // recursive function for identifying a group of white pixels
      function f(a, b)
            // check the pixel coordinates
            if (a>0 && a<=m)&&(b>0 && b<=n)
                  // check if the current pixel is white
                  if dat(a, b) > 0
                        // tag the current pixel
                        kind(a, b) = count;
                        // set the pixel to black
                        dat(a, b) = 0;
                        // recursively check the neighbors of pixels
                        f(a-1, b);
                        f(a+1, b);
                        f(a, b-1);
                        f(a, b+1);
                  end
            end
      end
      if np > 0
            for i = 1:np
            // loop over all white pixels
                  if dat(x(i), y(i)) > 0
                        // store neighboring white pixels into groups, tagged
                        //by count
                        count = count + 1;
                        f(x(i), y(i));
                  end
            end
      end
      // allocate memory for vesicle centers
      cen = zeros(count, 3);
      if count > 0
      // compute the center for each group of white pixels
            for i = 1:count
                  // extract coordinates of white pixels belonging to the
                  //same group
                  [x, y] = find(kind == i);
                  // compute the center and size of each group
                  cen(i, :) = [mean(x), mean(y), length(x)]
            end
      end
end
Special MATLAB functions: None
```

With all the steps above, the vesicle positions in all the images are then ready to be identified, or more specifically, by applying step 2 and step 3 to the pixel data read from all raw images in step 1, whose details are given in Code 4.

**Code 4: Function for identifying vesicle positions from a series of raw images**

```
function[point, np] = find_point(filenames, Lx, Ly, r_bub, LOG_min, np_bub)
////////////////////////////////////////////////////////////////////////////
// inputs:
// filenames: file names of raw images
// Lx, Ly: size of raw images in pixels
// r_bub: approximate mean radius of vesicles in pixels
// LOG_min: minimum value for the LOG filter
// np_bub: approximate minimum size of vesicles in pixels
// output:
// point: a cell of matrices storingthe coordinates of vesicles of each image
// np: number of vesicles identified for each image
////////////////////////////////////////////////////////////////////////////
      // extract pixel data from raw images (STEP 1)
      group = read_img(filenames, Lx, Ly)
      // number of images
      nimg = size(group, 3)
      // allocate memory for the outputs
      point = cell(nimg, 1);
      np = zeros(nimg, 1);
      // loop over images
      for i = 1:nimg
            // apply the LOG filter (STEP 2)
            tmp = LOG(group(:, :, i), r_bub, LOG_min);
            // identify centers and sizes (STEP 3)
            cen = find_blob(tmp);
            // find vesicles larger than the given minimum value
            isvalid = cen(:, 3) > np_bub;
            // store their center coordinates
            point{i} = cen(isvalid, 1:2);
            // store the number of valid vesicles for the current image
            np(i)    = sum(isvalid);
      end
end
Special MATLAB functions:None
```

### 2.2. Algorithm for trajectory tracing

From human-eye observations, the number of vesicles in each image can be different and their movements can be categorized into the following five types:

1. Move and oscillate within a small region;
2. Move in a very large region with large speed;
3. Some vesicles vanish;
4. Multiple vesicles merge into a single one;
5. Some new vesicles emerge.

In accordance, four types of trajectories must be allowed for the tracing algorithm:

- **Complete trajectories**: traced through the entire series of images
- **Incomplete trajectories**: only traced in a portion of consecutive images
- **Branched trajectories**: a single trajectory expands into multiple branches at certain step
- **Merged trajectories**: multiple trajectories merge into one at certain step.

Therefore, the number of trajectories is most likely NOT equal to the number of vesicles and a relatively complicated data structure is defined for storing and manipulating the trajectory data. In this DIB, it is given the name **TRJ**, which is a compound data structure containing the following members:

- **TRJ.counter**: the number of trajectories stored;
- **TRJ.pos**: array of pointers that access the data of each trajectory;
- **TRJ.nframe**: number of images used for tracing trajectories;
- **TRJ.dat**: group of trajectory data. The information of each trajectory in stored in a three-column matrix as: [**iframe, pid1, pid2**], where **iframe** is the index of the current image, **pid1** is the index of vesicle in the previous image and **pid2** is the index of vesicle in the current image. Such type of

data structure is used because vesicles in each image are identified by the same algorithm mentioned above and therefore vesicles with same indices in different images are most likely not the same.

The schematic of the TRJ structure is illustrated in Fig. 1.
**For the tracing process**, the following methods are defined for TRJ:

1. **Initialization**: initialize the data structure and allocate memory.

**Code 5: Method for initialize the TRJ data structure**

```
function trj = trj_init(nimg, ntrj_max)
////////////////////////////////////////////////////////////////////////////
// inputs:
// nimg: number of images
// ntrj_max: initial guess of the maximum number of trajectories (only affects
//the efficiency of the program
// output:
// trj: initialized TRJ structure for storing trajectory information
////////////////////////////////////////////////////////////////////////////
       // define the structure in MATLAB
       trj = struct('dat', [], 'counter', 0, 'nframe', nimg, 'pos', []);
       // allocate memory for members
       trj.dat = cell(ntrj_max, 1);
       trj.pos = zeros(ntrj_max, 1);
end
Special MATLAB functions:None
```

2. **New**: add a new trajectory.

**Code 6: Method for adding a new trajectory to the TRJ data structure**

```
function trj = trj_new(trj, iframe, ibub)
////////////////////////////////////////////////////////////////////////////
// inputs:
// trj: the old TRJ
// iframe: image index
// ibub: vesicle index
// output:
// trj: updated TRJ with a new trajectory starting with vesicle of index ibub
in //the image of index iframe.
////////////////////////////////////////////////////////////////////////////
       // increase the counter
       trj.counter = trj.counter + 1;
       // allocate memory for the new trajectory data
       trj.dat{trj.counter} = zeros(trj.nframe, 3);
       // set the head of the new trajectory in its data
       trj.dat{trj.counter}(1, :) = [iframe, 0, ibub];
       // set the local pointer of the new trajectory
       trj.pos(trj.counter) = 1;
end
Special MATLAB functions:None
```

3. **Tails**: get the tails of all live trajectories, which are equal to the locations of their local pointers during the tracing process. A live trajectory means its end has not been identified yet, and the judging criteria is (pid2 $>$ 0 in Fig. 1) for its current segment, so that it is considered as 'alive' for the tracing process. Within this method, trajectories with identical tails will be identified, which corresponds to the merging scenario mentioned above, and those with the same tail will be compared by their lengths, where only the longest will be kept alive while the rest will be killed (considered as merged), as illustrated in Fig. 2.

The source code for the 'Tails' method is given in Code 7.

**Code 7: Method for returning the tails of all live trajectories**

```
function [trj_ids, img_ids, bub_ids] = trj_tails(trj)
////////////////////////////////////////////////////////////////////////////////
// inputs:
// trj: the TRJ data
// output:
// trj_ids: indices of all live trajectories.
// img_ids: indices of images for their last segments
// bub_ids: indices of vesicles in corresponding images for their last segments
////////////////////////////////////////////////////////////////////////////////
      // find trajectories with non-zero length
      trj_ids = find(trj.pos > 0);
      // extracting local pointers
      pos = trj.pos(trj_ids);
      // allocate memory for outputs
      img_ids = zeros(length(trj_ids), 1);
      bub_ids = zeros(length(trj_ids), 1);
      // extracting image indices and vesicle indices for the last segments
      for i = 1:length(trj_ids);
            dat = trj.dat{trj_ids(i)}(pos(i), :);
            bub_ids(i) = dat(3); // pid2
            img_ids(i) = dat(1);
      end
      // identify live trajectories
      isvalid = bub_ids > 0;
      trj_ids = trj_ids(isvalid);
      img_ids = img_ids(isvalid);
      bub_ids = bub_ids(isvalid);
      pos     = pos(isvalid);
      // identify merged trajectories and kill shorter ones
      [tmp, I] = sortrows(bub_ids);
      tokill   = false(length(I));
      id_max   = 1;
      pos_max  = pos(I(id_max));
      bub_now  = tmp(id_max);
      for i = 2:length(I)
            Ii = I(i);
            if bub_now == tmp(i)// check for duplicated tails
            //compare lengths of trajectories with the same tail
                  if pos(Ii) > pos_max
                  // current is longer, update the longest and kill the
                  //previous longest one
                        tokill(id_max) = true;
                        pos_max        = pos(Ii);
                        id_max         = i;
                  else
                  // current is shorter and kill it
                        tokill(i) = true;
                  end
            else
            // next tail (different than the previous one)
                  bub_now = tmp(i);
                  pos_max = pos(Ii);
                  id_max  = i;
            end
      end
      // kill merged but shorter trajectories identified above
      if any(tokill)
            kill_ids = I(tokill);
            for i = 1:length(kill_ids)
                  i1     = kill_ids(i);
                  trj_id = trj_ids(i1);
                  pos1   = pos(i1)+1;
                  trj.dat{tid}(pos1, :) = [img_ids(i1), bub_ids(i1), 0];
                  trj.pos(tid)          = pos1;
            end
            // remove killed trajectories from the outputs
            idleft = I(~tokill);
            trj_ids = trj_ids(idleft);
            img_ids = img_ids(idleft);
            bub_ids = bub_ids(idleft);
      end
end
Special MATLAB functions:
sortrows(): Built-in MATLAB Function
```

4. **Insertion**: append new segments to live trajectories. If no corresponding live trajectories are found for certain new segments, new trajectories will be created, using them as the head segments. The criteria for identifying segments of new trajectories is: pid1 $==$ 0 and pid2 $>$ 0, as shown in Fig. 1, and the source code for this method is given in Code 8.

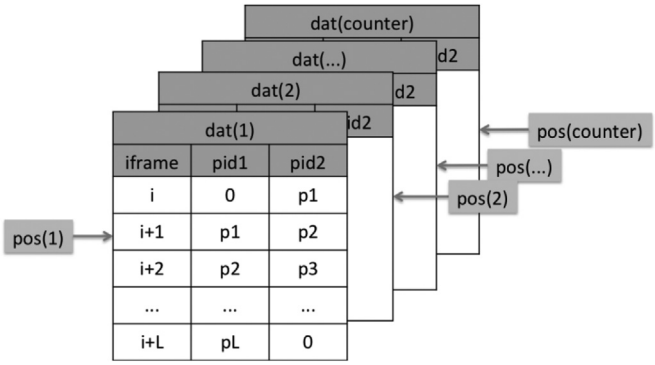


**Fig. 1.** Schematic illustration of the compound data structure storing trajectories (TRJ).

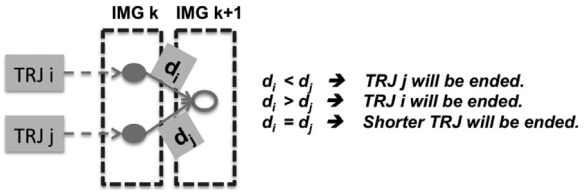


**Fig. 2.** Merging of trajectories.

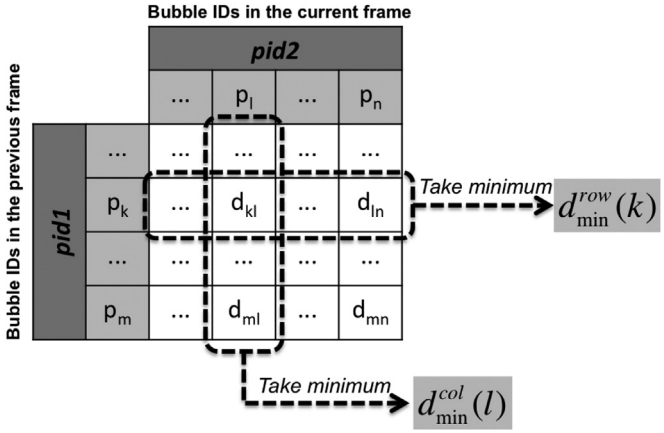


**Fig. 3.** Distance matrix used for determining closest vesicles/bubbles in two consecutive images.

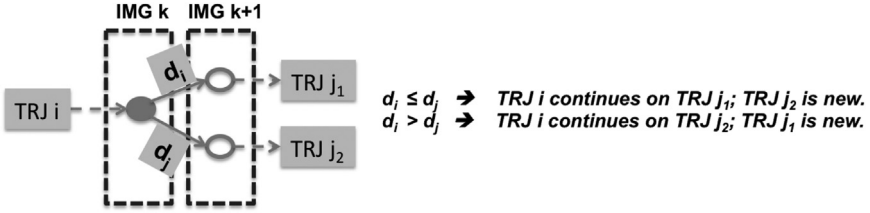


**Fig. 4.** Branching of a trajectory.

**Code 8: Method for appending (or creating) segments to live trajectories**

```
function trj = trj_insert(trj, trj_ids, img_id, bub_ids1, bub_ids2)
////////////////////////////////////////////////////////////////////////////
// trj: the old TRJ
// trj_ids: indices of live trajectories
// img_id: image index corresponding to new segments
// bub_ids1: vesicle index in the previous image (img_id-1) (pid1 in Figure 1)
// bub_ids2: vesicle index in the current image (img_id) (pid2 in Figure 1)
// output:
// trj: updated TRJ.
////////////////////////////////////////////////////////////////////////////
      // number of segments
      nseg = length(trj_ids);
      for i = 1:nseg
            pid1 = bub_ids1(i);
            pid2 = bub_ids2(i);
            tid  = trj_ids(i);
            if pid1 == 0
                  if pid2 > 0
                  // add new trajectory (refer to the method 'New')
                        trj = trj_new(trj, img_id, pid2);
                  end
            else
                  // append a new segment to an existing trajectory and
                  //update its local pointer
                  pos = trj.pos(tid)+1;
                  trj.dat{tid}(pos, :) = [img_id, pid1, pid2];
                  trj.pos(tid)         = pos;
            end
      end
end
Special MATLAB functions:None
```

5. **Coordinates Extraction**: extract coordinates by vesicle indices and image indices. Since only vesicle indices and image indices are stored in the TRJ data structure, such coordinate extraction method must be defined to get the actual coordinates of vesicles along their trajectories, using the results obtained from the vesicle-identification process. The source code for this method is given in Code 9.

**Code 9: Extracting vesicle coordinates by vesicle indices and image indices**

```
function coords = trj_gen_coords(img_ids, bub_ids, point)
////////////////////////////////////////////////////////////////////////////
// inputs:
// img_ids: image indices
// bub_ids: vesicle indices
// point: vesicle coordinates for all images (identified by Code 4)
// output:
// coords: extracted coordinates of specified vesicles at specified images.
////////////////////////////////////////////////////////////////////////////
      // number of vesicles
      nbub = length(bub_ids);
      // allocate memory for the output
      coords = zeros(nbub, 2);
      for i = 1:nbub
            coords(i, :) = points{img_ids(i)}(bub_ids(i), :);
      end
end
Special MATLAB functions:None
```

Limited by experiment conditions, vesicles cannot be individually tagged and traced by techniques other than visually comparing their positions in consecutive images. Therefore, the **criteria for identifying two vesicles in two consecutive images belonging to the same trajectory** are:

- These two vesicles are closest to each other than the rest;
- Their distance should be smaller than a given cutoff distance that corresponds to the maximum moving distance of a vesicle for the time interval between two consecutive images.

The closest vesicles in two consecutive images are determined by using the **distance matrix** shown in Fig. 3, where distances between all pairs of vesicles in two images are computed and stored. Minimum values are taken for each row and each column and the corresponding column index and row index are stored as well. Taking the **row minimum** is equivalent to finding the closest vesicles in the current frame to the previous frame, while taking the **column minimum** determines the closest vesicles in the previous

frame to the current frame. If the obtained minimum distance is less than the cutoff distance, then it is considered as a valid value. The results can be categorized into the following five types:

1. **Regular movement:** A vesicle in the current frame is the closest neighbor to a *unique* vesicle in the previous frame and vise versa. In the distance matrix, a unique row minimum can be found at the corresponding column;
2. **Merging**: A vesicle in the current frame is the closest neighbor to *more than one vesicles* in the previous frame (as illustrated in Fig. 2); In the distance matrix, more than one row minimums will be found at the corresponding column;
3. **Branching**: A vesicle in the previous frame is the closest neighbor to *more than one* vesicles in the current frame (as illustrated in Fig. 4); In the distance matrix, more than one column minimums will be found at the corresponding row;

4. **End of a trajectory**: For a vesicle in the previous frame, no vesicle in the current frame can be considered as a valid neighbor (distance too large).
5. **Begin of a new trajectory**: For a vesicle in the current frame, no vesicle in the previous frame can be considered as a valid neighbor (distance too large).

The source code for identifying the connectivity of vesicles identified in two consecutive images is given in Code 10.

**Code 10: Identify connectivity of vesicles identified in consecutive images**

```
functionconnect = find_connect(bub1, bub2, rcut)
////////////////////////////////////////////////////////////////////////////
// inputs:
// pt1: coordinates of vesicles in the previous image
// pt2: coordinates of vesicles in the current image
// rcut: cutoff distance for valid neighbors
// output:
// connect: connectivity matrix in the form of [pid1, pid2], where the vesicles
//with indices pid2 in the current image correspond to vesicles with indices
//pid1 in the previous image. If (pid1 == 0), then the corresponding pid2 are
//considered as beginnings of new trajectories. If (pid2 == 0), then
//trajectories with tails corresponding to pid1 are considered to be ended.
////////////////////////////////////////////////////////////////////////////
      // number of vesicles in the previous image
      nbub1 = size(bub1, 1);
      // number of vesicles in the current image
      nbub2 = size(bub2, 1);
      // allocate memory for the output
      connect = zeros(nbub1+nbub2, 2);
      // initialize distance matrix (see Figure 3).
      dmat = -ones(nbub1, nbub2);
      // compute distance matrix
      for i = 1:nbub1
            d = repmat(bub1(i, :), nbub2, 1) - bub2;
            dmat(i, :) = sum(d.*d, 2)';
      end
      // initialize the connectivity matrix
      connect(1:nbub1, 1)       = (1:nbub1)';
      connect((nbub1+1):end, 2) = (1:nbub2)';
      // identifying row minimums
      for i = 1:nbub1
            [dmin_row, id] = min(dmat(i, :));
            // compare to the cutoff distance and keep the valid neighbor
            if dmin_row< rcut^2
                  connect(i, 2) = id;
            end
      end
      // identifying column minimums
      for i = 1:nbub2
            [dmin_col, id] = min(dmat(:, i));
            // compare to the cutoff distance and keep the valid neighbor
            if dmin_col < rcut^2
                  connect(i+nbub1, 1) = id;
            end
      end
      // remove duplicated rows in the connectivity matrix
      connect = unique(connect, 'rows');
end
Special MATLAB functions:
unique(…, 'rows'): Built-in MATLAB Function
```

Based on human-eye observations of raw images, the speed of vesicles can vary significantly. Some vesicles can move much faster than others, while the speed of the same vesicle can also sometimes change significantly during a time period. Therefore, a single cutoff distance for the connectivity search can be inconvenient. A small cutoff distance can improve the accuracy of the tracing algorithm for slow vesicles, but very likely miss many fast ones, while a large cutoff distance will decrease the accuracy of the algorithm for vesicles close to each other (mixing-up). Hence, an improved algorithm based on Code 10 is used for searching the connectivity between two groups of points. For the new algorithm, multiple cutoff distances ranging from a minimum value to a maximum value are used, and the implementation is given in Code 11.

**Code 11: Identify connectivity of vesicles with multiple cutoff distances**

```
function connect = find_connect_multi(bub1, bub2, rcut_min, rcut_max, nres)
////////////////////////////////////////////////////////////////////////////
// inputs:
// pt1: coordinates of vesicles in the previous image
// pt2: coordinates of vesicles in the current image
// rcut_min: minimum cutoff distance for valid neighbors
// rcut_max: maximum cutoff distance for valid neighbors
// nres: number of searching steps between rcut_min and rcut_max
// output:
// connect: connectivity matrix of two groups of vesicles, same as Code 10
////////////////////////////////////////////////////////////////////////////
      // evaluate cutoff distances for each search step
      rcut = linspace(rcut_min, rcut_max, nres);
      rid  = 1;
      // number of vesicles in the previous image
      nbub1 = size(bub1, 1);
      // number of vesicles in the current image
      nbub2 = size(bub2, 1);
      // allocate memory for the output
      connect = zeros(nbub1+nbub2, 2);
      ids1    = (1:nbub1)';
      ids2    = (1:nbub2)';
      counter = 0;
      // loop over cutoff distances and apply Code 10 for each one
      while rid <= nres
            // identify vesicle connectivity using Code 10
            connect1 = find_connect(bub1(ids1, :), bub2(ids2, :), rcut(rid));
            // indices of bub1 for the next search
            is_left1 = false(length(ids1), 1);
            // indices of bub2 for the next search
            is_left2 = false(length(ids2), 1);
            // look for isolated vesicles (without valid neighbors)
            for i = 1:size(connect1, 1)
                  pid1 = connect1(i, 1);
                  pid2 = connect1(i, 2);
                  if pid1*pid2 == 0
                  // mark isolated vesicles for the next search
                        if pid1 == 0
                              is_left2(pid2) = true;
                        else
                              is_left1(pid1) = true;
                        end
                  else
                  // store valid neighbors in the output
                        counter = counter + 1;
                        connect(counter, :) = [ids1(pid1), ids2(pid2)];
                  end
            end
            // prepare for the next search
            ids1 = ids1(is_left1);
            ids2 = ids2(is_left2);
            // check if the search pools are empty
            ifisempty(ids1) || isempty(ids2)
                  break;
            end
            rid = rid + 1;
      end
end
Special MATLAB functions:
linspace(): Built-in MATLAB Function
```

Trajectories of vesicles can then be traced using the methods introduced above (Code 5–Code 9), as well as the connectivity search algorithms (Code 10–Code 11), and stored in the TRJ data structure.

The implementation for the combined tracing algorithm is given in Code 12.

**Code 12: Trace trajectories of vesicles identified in a series of images**

```
function[trj, trj_mat] = trace_trj(point, rcut_min, rcut_max, nres)
////////////////////////////////////////////////////////////////////////////////
// inputs:
// point: vesicle coordinates for all images (identified by Code 4)
// rcut_min: minimum cutoff distance for valid neighbors
// rcut_max: maximum cutoff distance for valid neighbors
// nres: number of searching steps between rcut_min and rcut_max
// outputs:
// trj: trajectory data stored as the TRJ data structure (Figure 1)
// trj_mat: trajectory matrix used for postprocess (mentioned below)
////////////////////////////////////////////////////////////////////////////////
       // number of images
       nimg = length(point)
       // initialize number of vesicles
       nbub= size(point{1}, 1);
       // initialize the TRJ data (Code 5)
       trj = trj_init(nimg, nbub*10);
       // add all vesicles in the first image as new trajectories
       pids2   = (1:nbub)';
       pids1   = zeros(nbub, 1);
       trj_ids = (1:nbub)';
       trj     = trj_insert(trj, trj_ids, 1, pids1, pids2);// (Code 8)
       // loop over all images and trace trajectories
       for n = 2:nimg
              // get tails of live trajectories (also handles merging)
              [trj_ids, ~, pids1] =trj_tails(trj);// (Code 7)
              // extract coordinates of tails
              cen1 =trj_gen_coords(n-1, pids1, point);// (Code 9)
              // identify the connectivity matrix between vesicles in current
              //image and those in the previous image (Code 11)
              connect =find_connect_multi(cen1, point{n}, rcut_min, rcut_max, …
              rcut_nres);
              // based on the connectivity matrix to determine whether to create
              //new trajectories (branching) or extend current ones.
              isoccupied = false(length(trj_ids), 1);
              for i = 1:size(connect, 1)
              // loop over neighbor pairs
                     tid_loc = connect(i, 1);
                     pid2    = connect(i, 2);
                     if tid_loc > 0
                     // segments for existing trajectory
                            pid1 = pids1(tid_loc);
                            if isoccupied(tid_loc)
                            // if the trajectory is already appended, then a
                            //create new one (branching)
                                   trj =trj_new(n-1, pid1);// (Code 6)
                                   tid =trj.counter;
                            else
                            // append to existing trajectory
                                   tid = trj_ids(tid_loc);
                                   isoccupied(tid_loc) = true;
                            end
                            // insert segment
                            trj =trj_insert(tid, n, pid1, pid2);// (Code 8)
                     else
                     // segments for new trajectory
                            trj =trj_new(n, pid2);// (Code 6)
                     end
              end
       end
       // clear unused memory allocated for TRJ (Code 13)
       trj =trj_trim(trj);
       // generate trajectory matrix for postprocess (Code 14)
       trj_mat =trj_gen_trj_map(trj);
end
Special MATLAB functions:None
```

To improve the efficiency of the program, the TRJ data needs to be initialized with an over-estimated number of possible trajectories. Therefore, a trimming function should be used after the tracing process to free unused memory, whose implementation is given in Code 13.

**Code 13: Free unused memory allocated for the TRJ data**

```
function trj = trj_trim(trj)
////////////////////////////////////////////////////////////////////////////
// inputs:
// trj: trajectory data (TRJ) (Figure 1)
// outputs:
// trj: trajectory data (TRJ) after memory cleanup
////////////////////////////////////////////////////////////////////////////
      // free completely unused memory
      trj.dat = trj.dat(1:trj.counter);
      trj.pos = trj.pos(1:trj.counter);
      // remove trajectories corresponding to only one segment
      is_rm = trj.pos == 1;
      trj.dat(is_rm) = [];
      trj.pos(is_rm) = [];
      trj.counter    = trj.counter - sum(is_rm);
      // free unused memory allocated for each trajectory
      for i = 1:trj.counter
            trj.dat{i} = trj.dat{i}(1:trj.pos(i), :);
      end
end
Special MATLAB functions:None
```

## 2.3. Postprocess for trajectories

The postprocess for trajectories includes visualization and calculation of extra information such as velocity, speed, moving range (displacement), etc. from the obtained vesicle positions and trajectory data. The design of the TRJ data structure illustrated in Fig. 1 is for the convenience of the tracing process, but its direct usage for the postprocess is not efficient. Instead, we designed a **trajectory matrix** that can be extracted from the TRJ data after the identification process for the postprocess. The schematic of the matrix structure is shown in Fig. 5. And the MATLAB code for extracting the matrix is given in Code 14.

**Code 14: Extract the trajectory matrix from the TRJ data for the postprocess**

```
function trj_mat = trj_gen_trj_map(trj)
////////////////////////////////////////////////////////////////////////////
// inputs:
// trj: trajectory data (TRJ) (Figure 1)
// outputs:
// trj: trajectory matrix for the postprocess (Figure 5)
////////////////////////////////////////////////////////////////////////////
      // allocate memory for the output and initialize
      trj_mat = zeros(trj.counter, trj.nframe);
      // loop over each trajectory data and extract the vesicle index
      for i = 1:trj.counter
            if trj.pos(i) > 0
            // ensure current trajectory data is not empty
                  tmp = trj.dat{i}(1:trj.pos(i), [1, 3]);
                  // image indices
                  img_ids = tmp(:, 1);
                  // vesicle indices
                  bub_ids = tmp(:, 3);
                  if bub_ids(end) == 0
                        img_ids = img_ids(end-1);
                        bub_ids = bub_ids(end-1);
                  end
                  // fill in the trajectory with vesicle indices
                  trj_mat(i, img_ids) = bub_ids';
            end
      end
end
Special MATLAB functions:None
```

The sequential coordinates of vesicles corresponding to each trajectory can be easily obtained by using the trajectory matrix, together with the coordinate generation code given by Code 9. Then the velocity, speed and moving range (displacement), as well as other kinds of movement information can be readily derived. In addition, by counting the number of non-zero elements in each row of the trajectory matrix, the life (how many images a trajectory lasts) of each trajectory can be obtained, which can be used for controlling the quality of trajectories used for calculating statistical

| | | Column number = Image ID | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | ... | N_img |
| Row number = Trajectory ID | 1 | 0 | P1 | p3 | 0 | ... | 0 |
| | 2 | P2 | p2 | p3 | p5 | ... | p3 |
| | ... | ... | ... | ... | ... | ... | ... |
| | N_trj | 0 | 0 | p3 | p4 | ... | 0 |

**Fig. 5.** Trajectory matrix used for the postprocess, where elements are indices of identified vesicles in each image whose index is the column number, while the row number is the index of trajectories. A trajectory started with the first non-zero vesicle index and ended with the last non-zero index. In the above example, the 1st and 2nd trajectories merged at the 3rd frame, while the 2nd trajectory branched to the last trajectory at the 4th frame.

information. For example, adding a lower-bound filter to the trajectory life can allow us to exclude short-lived trajectories most likely caused by image noises.

The **velocity** of a vesicle along a trajectory is computed by

$$\vec{v}(n) = \begin{cases} \frac{\vec{x}(n+1) - \vec{x}(n)}{\Delta t}, & n = 1 \\ \frac{\vec{x}(n+1) - \vec{x}(n-1)}{2\Delta t}, & 1 < n < N\_img \\ \frac{\vec{x}(n) - \vec{x}(n-1)}{\Delta t}, & n = N\_img \end{cases}$$

where $\vec{v}(n)$ is the velocity at the time corresponding to the image of index $n$, $\vec{x}(n)$ is the vesicle position, $\Delta t$ is the time interval between two consecutive images and *N_img* is the number of images. The **speed** is by definition the magnitude of the velocity, namely

$$v(n) = \left\| \vec{v}(n) \right\|$$

The **traveling distance** of a vesicle along its trajectory is computed by

$$s(n) = \begin{cases} 0, & n = 1 \\ \sum_{i=2}^{n} \| \vec{x}(i) - \vec{x}(i-1) \|, & n \le N\_img \end{cases}$$

The **moving range (displacement)** of a vesicle along its trajectory is defined by

$$R(n) = \begin{cases} 0, & n = 1 \\ \max\{ \| \vec{x}(i) - \vec{x}(1) \| | 1 < i \le n, & n \le N\_img \end{cases}$$

Implementations for calculating the above variables are quite straightforward and are therefore omitted here.

## Acknowledgements

This study was supported by the National Natural Science Foundation of China (31760320), the Natural Science Foundation of Jiangxi Province of China (20161ACB20001, 20171BAB211014).

## Transparency document. Supplementary material

Transparency data associated with this article can be found in the online version at https://doi.org/10.1016/j.dib.2018.12.076.

## Reference

[1] W. Zhang, Y. Xu, G. Chen, K. Wang, W. Shan, Y. Chen, Dynamic single-vesicle tracking of cell-bound membrane vesicles on resting, activated, and cytoskeleton-disrupted cells, BBA-Biomembr., vol. 1861(1), pp. 26–33.