

trackdem: Automated particle tracking to obtain population counts and size distributions from videos in R

Marjolein Bruijning, Marco D. Visser, Caspar A. Hallmann, Eelke Jongejans

October 24, 2018

The aim of *trackdem* is to obtain unbiased, automated estimates of population densities and body size distributions, using video material or image sequences as input. It is meant to assist in evolutionary and ecological studies, which often rely on accurate estimates of population size, population structure and/or individual behaviour. The package *trackdem* includes a set of functions to convert a short video into an image sequence, background detection, particle identification and linking, and the training of an artificial neural network for noise filtering.

This vignette provides a step-by-step introduction on the usage of all functions to analyse image sequences of moving organisms. No movie files are required as all functions are illustrated by simulated image sequences.

Contents

| | | |
|----|---|----|
| 1 | Simulate image sequences | 2 |
| 2 | Preparing and loading movie files into R | 2 |
| 3 | Identification of moving particles | 3 |
| 4 | Particle tracking | 7 |
| 5 | Choice of maximum cost (L) and frame search range (R) | 10 |
| 6 | Noise removal using artificial neural networks | 12 |
| 7 | Robustness checks | 14 |
| 8 | Batch analysis | 15 |
| 9 | Merge track objects | 17 |
| 10 | Creation of image sequence | 17 |

1 Simulate image sequences

The package *trackdem* contains functions to simulate an image sequence, which makes it straightforward to illustrate the functionality of the package or test its limitations. We use this for the first part of the tutorial, Section 10 discusses working with actual video files.

The following code simulates movement trajectories of particles we will later identify and track. The code simulates 15 individuals across 30 frames, with a displacement speed of 0.01 (in pixels, parameter *h*), and directional correlation of 0.9 (parameter *rho*), which sets the correlation parameter for the angle of displacement. Particle sizes are set with *sizes*, and are randomly drawn by default. By default, particles are allowed to move in a rectangle domain. Use *domain="circle"* to use a circular domain of radius 1. Images are saved as .png files in the path specified with argument *path*.

```
> require(trackdem)
> dir.create("images")
> set.seed(1000)
> ## Create image sequences
> traj <- simulTrajec(path="images",
+                   nframes=30,nIndividuals=15,domain="square",
+                   h=0.01,rho=0.9,
+                   sizes=runif(15,0.004,0.006))
```

Random static or moving noise can be added, using *staticNoise=TRUE*, or *movingNoise=TRUE*. Static noise creates non-moving grey spots; moving noise results in the creation of randomly moving particles, that remain for a random number of frames, and then disappear. Static noise is filtered out by background detection; moving noise can be filtered out with machine learning (see Section 6). Parameters used to generate this static and moving noise can be set as a list in *parsStatic* and *parsMoving*. See ?simulTrajec for more information. The above created image sequences can now be viewed in the working folder (see Fig. 1), and analyzed as described below. When using video material, image sequences can be created from movie files using *trackdem*, or using own software. See Section 10 for more detailed information.

2 Preparing and loading movie files into R

Next, we load the sequence into R, continuing with our simulated sequence:

```
> dir <- "images"
> allFullImages <- loadImages (dirPictures=dir,nImages=1:30)
> allFullImages

Trackdem color image
Images with size: 480 x 480 pixels.
Total of 30 images.

> class(allFullImages)

[1] "TrDm"          "colorimage" "array"
```

With argument *nImages* you can choose which color images to load. By default, images 1-30 (in the specified *dirPictures*) will be loaded. If you want to specify which images should be loaded, you can also use argument *filenames*, to provide a character string containing all image names you are interested in. By default, the full images are loaded. If you want a specific region of interest, specify the x and y coordinates, using argument *xranges* (e.g. *xranges=100:500*) and *yranges*. The function returns a four dimensional array, with a size of the of N_y rows \times N_x columns \times 3 colors \times *nImages*. All functions in *trackdem* return objects of class 'TrDm'. Use *plot()* to see the images:

```
> plot(allFullImages,frame=1)
```

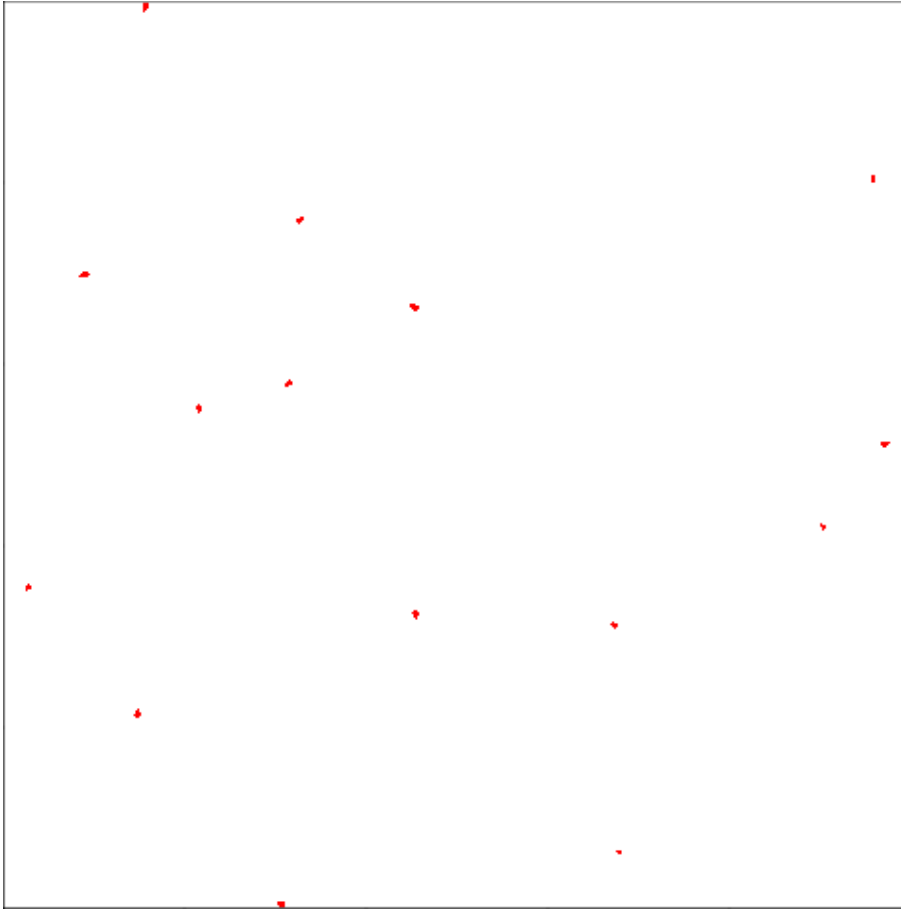


Figure 1: First frame of loaded (simulated) image sequence.

3 Identification of moving particles

We can now proceed with motion detection. The first step is to identify the background containing all motionless pixels. The package *trackdem* provides three different methods for background detection. By default, the mean pixel values over all frames, per color, is calculated (*method="mean"*). Alternatives are *method="powerroot"* and *method="filter"* (see `?createBackground`).

```
> ## Detect background
> stillBack <- createBackground(allFullImages,method="mean")

> class(stillBack)

[1] "TrDm"      "colorimage" "array"

> stillBack

Trackdem color image
Images with size: 480 x 480 pixels.
Total of 1 image.
```

`createBackground` returns an array with size N_y rows \times N_x columns \times 3 colors. We can now assess the created background and whether the function succeeded by plotting the object *stillBack*.

```
> plot(stillBack)
```



Figure 2: Example of background obtained using function *createBackground*, and *method="mean"*

Above, we did not simulate any static noise, resulting in the detected white background. When we simulate trajectories containing static noise and a circular domain, we can confirm that the background image successfully detects the static noise and circle:

```
> dir.create('imagesNoise')
> trajNoise <- simulTrajec(path="imagesNoise",
+                           nframes=30,nIndividuals=20,domain='circle',
+                           h=0.01,rho=0.9,staticNoise=TRUE,
+                           sizes=runif(20,0.004,0.006))
> dir <- "imagesNoise"
> allFullImagesNoise <- loadImages (dirPictures=dir,nImages=1:30)
> stillBackNoise <- createBackground(allFullImagesNoise,method="mean")
> plot(stillBackNoise)
```

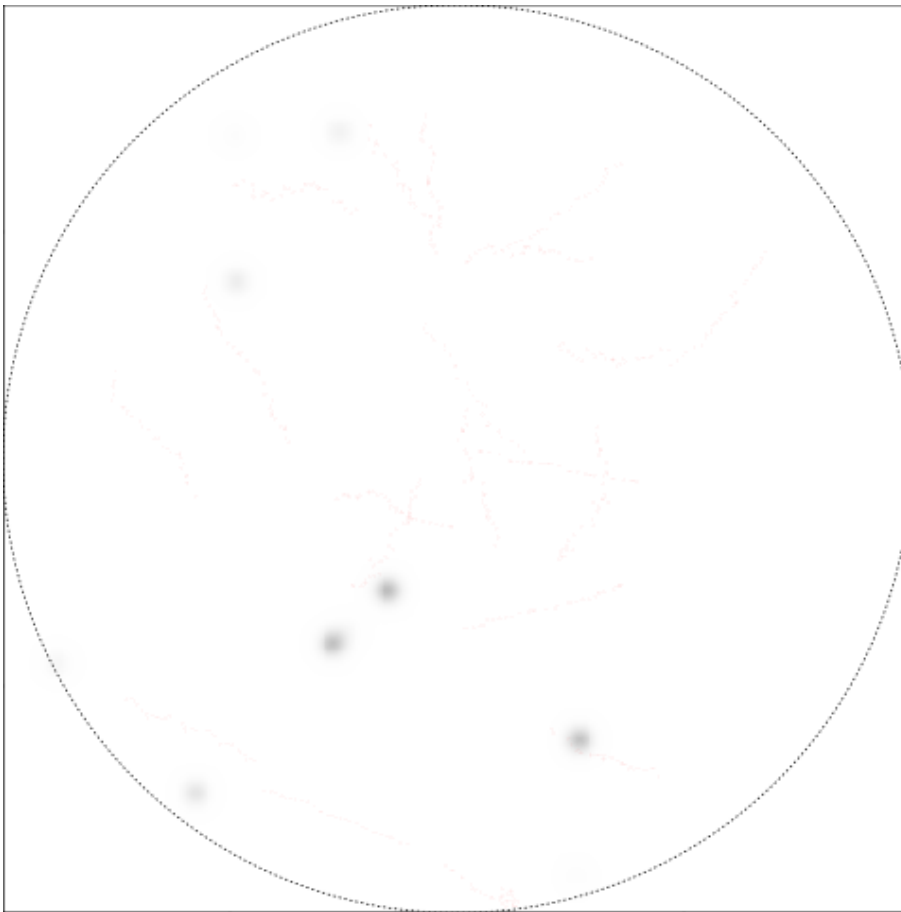


Figure 3: Example of background obtained using function `createBackground`, and `method="mean"`, including static noise and a circular domain.

Continuing with the sequence without noise, we now subtract the background from all images, which will identify all moving particles. By default, the original color images are used from the global environment. The argument `colorimages` can be used to specify a different set of images.

```
> allImages <- subtractBackground(bg=stillBack)
```

The procedure produces a four dimensional array (N_y rows \times N_x columns \times 3 colors \times $nImages$), containing the difference in color intensity between all original frames and the background. As explained in the manuscript, it is important that the background is stable and that only focal particles move. If this is not the case, for instance in more complex field conditions, more advanced background detection methods may be required. The function `subtractBackground` allows the input of objects not created by `createBackground`, for instance when the user wishes to supply a custom background. When doing so, make sure that the dimensions of the created background object equal either N_y rows \times N_x columns \times 3 colors, or N_y rows \times N_x columns \times 3 colors \times $nImages$, otherwise an error is returned. Using a four dimensional background implies that a dynamic background model can be used, that is different for each frame. When using short videos with a stable background, one of the three available background detection methods in `createBackground` will suffice.

Next, we identify particles, where the first step is thresholding. All pixel values larger than a threshold (or smaller, depending on the particle color compared to the background color) will be labeled as particles. To find an appropriate threshold value, use the function `findThreshold` (e.g. `findThreshold(allImages)`). This function opens an interactive interface, making use of the R-package `shiny` (Fig. 4). The graph on the left shows the original image. The graph on the right

shows the binary image in which all pixels are either labeled as 0 (white) or 1 (black), depending on a threshold. Set whether particles are darker or lighter than the background and use the slider to test a threshold. Note that when using simulated trajectories, in which focal particles are red, the first color channel does not distinguish between background and particles. Select another color channel to find appropriate threshold values.

Find threshold for particle detection

The graph on the left shows the original image. The graph on the right shows a binary image in which all particles are either labeled as zero (white), or as one (black). Set whether particles are light or dark compared to the background. Use the slider to find the best threshold, that results in all focal particles being labeled as one, and all background pixels labeled as zero. When finished, click on "Done".

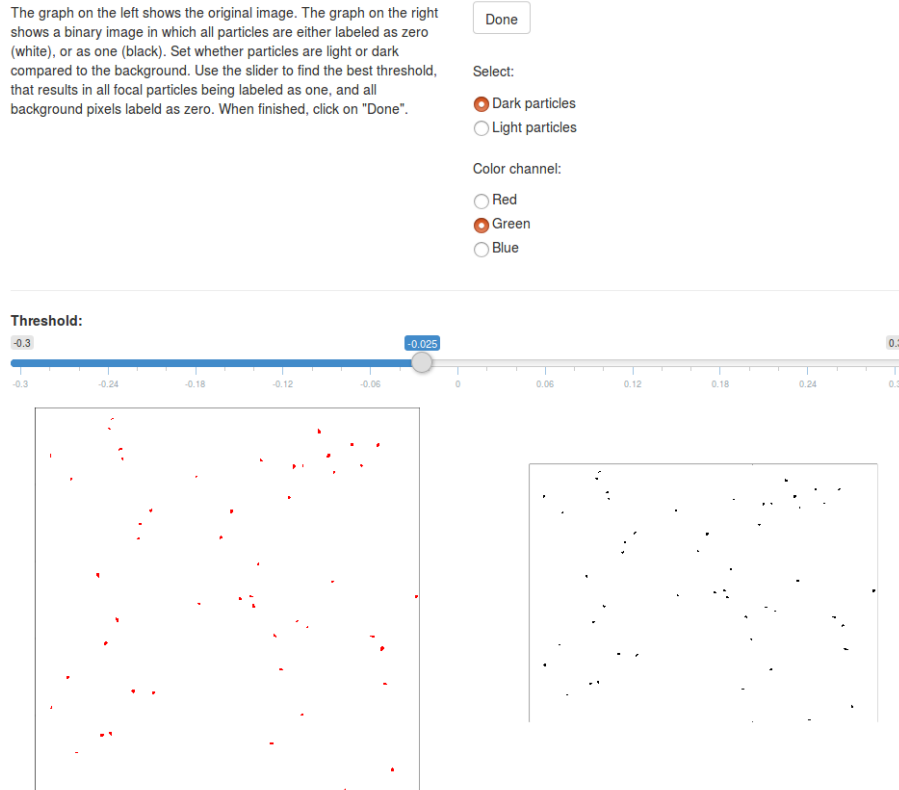


Figure 4: Graphical user interface to help users find the most appropriate threshold value.

When an appropriate threshold value is found, click on 'Done' to go back to R. Continue with *identifyParticles*, and use the chosen threshold value. Set the argument *select="dark"* when particles are darker compared to the background. Light particles are selected with *select="light"*, and *select="both"* selects both light and dark particles. In addition to supplying a threshold value, two other options for thresholding exist: automatic thresholding and thresholding based on a quantile. See Section ?? for more details.

Subsequently, all connecting pixels are labeled. Finally, pixels larger or smaller than user-defined values are removed (set with argument *pixelRange*). This can be used as a first noise filter when users know the size range of their study organisms (in pixels).

```
> ## Identify moving particles
> partIden <- identifyParticles(sbg=allImages,threshold=-0.1,
+                               pixelRange=c(1,500),
+                               autoThres=FALSE)
> summary(partIden)

Summary of identified particles (unlinked):

Average number of identified particles: 14.87 ( sd = 0.35 )
```

```
Coefficient of variation: 0.02
Range of particles for each frame ( 1- 30 ): 14 - 15
Threshold values: -0.1 -0.1 -0.1
```

identifyParticles() returns a dataframe with identified particles for each frame, use *summary()* to obtain some summary statistics. Identified particles can be plotted as open circles on the original image with:

```
> plot(partIden, frame=10)
```

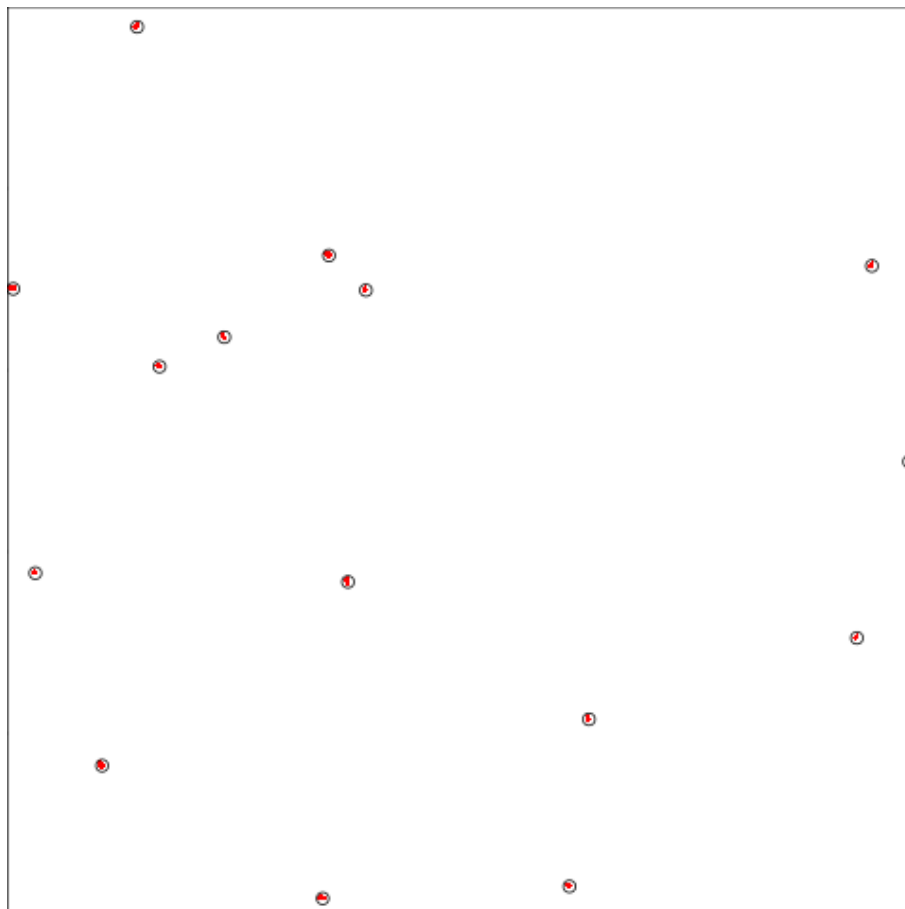


Figure 5: Identified moving particles shown as open circles using *identifyParticles*

4 Particle tracking

The next step is to reconstruct trajectories for each particle. In short, the tracking consists of two steps. First, particles in subsequent frames are linked, based on their size and distance. Second, trajectories that are not overlapping in time and likely involve the same particle, are merged, to fill gaps.

In this function, you can set the maximum "cost" for a particle or trajectory to link to another particle/trajectory in the next frame. The cost function depends on the size difference, the distance, and the difference between the predicted and observed location (based on the velocity), and returns the cost for linking particle i to particle j . If costs are larger than used-specified L , a particle is not linked to a particle, but to a dummy particle. This results in the start or end of a track segment. With argument R , the maximum number of frames is set to which particles after frame N can be

linked. This means that if a particle cannot be linked to a particle in frame $N + 1$ (because there are no particles, or costs are too high), algorithm tries to find a match in the subsequent frames, up to R frames.

To extract the outcomes, `summary()` can be used:

```
> summary(records, incThres=1)

Summary of Linked particles:

   ID Size  SD size Total movement Red Green  Blue
[1,]  1  12 1.2410600    176.6328  1 1e-06 1e-06
[2,]  2  15 1.1351237    161.3262  1 1e-06 1e-06
[3,]  3  12 0.8840866    178.8258  1 1e-06 1e-06
[4,]  4   8 1.0288929    151.4242  1 1e-06 1e-06
[5,]  5  11 1.2415230    137.5649  1 1e-06 1e-06
[6,]  6  10 2.0083160    173.7386  1 1e-06 1e-06
[7,]  7   7 0.9498941    150.4324  1 1e-06 1e-06
[8,]  8   7 0.9371024    201.2269  1 1e-06 1e-06
[9,]  9  12 0.9994251    152.1046  1 1e-06 1e-06
[10,] 10  12 0.9965458    185.1116  1 1e-06 1e-06
[11,] 11   9 0.7849153    191.7548  1 1e-06 1e-06
[12,] 12   7 1.2015316    173.8985  1 1e-06 1e-06
[13,] 13   9 1.1126973    156.3268  1 1e-06 1e-06
[14,] 14  14 1.9182813    189.2680  1 1e-06 1e-06
[15,] 15  10 1.9895045    153.7332  1 1e-06 1e-06

Total of 15 Identified particles.
Particles have a total area  of 0.0673 %
Minimum presence is set at 1 frames
```

The number of tracked segments is an estimate for the number of moving particles. A particle is only considered to be of interest when it is present in at least z frames. Users can define the value for z using the argument `incThres`, or use the default automatic algorithm which is based on k-means clustering. This sets the minimum number of frames that a tracked particle should be present. The logic behind this is that short tracks are more likely to represent noise.

Calculated statistics are stored in `summary(object)`. For example, to extract some basic statistics:

```
> # population count
> summary(records, incThres=1)$N

[1] 15

> # median body sizes
> summary(records, incThres=1)$particles[, "Size"]

[1] 12 15 12  8 11 10  7  7 12 12  9  7  9 14 10

> # total displacement (in pixels)
> summary(records, incThres=1)$particles[, "Total movement"]

[1] 176.6328 161.3262 178.8258 151.4242 137.5649 173.7386 150.4324 201.2269
[9] 152.1046 185.1116 191.7548 173.8985 156.3268 189.2680 153.7332

> # area covered by particles as a fraction of the total area
> summary(records, incThres=1)$area

[1] 0.06727431
```



```
> # minimum number of frames that a tracked particle
> # should be present (value z)
> summary(records,incThres=1)$presence
```

```
[1] 1
```

More detailed information can be extracted by looking at the reconstructed trajectories directly. The created object contains three arrays: 1) the x and y coordinates of each created trajectory for each frame, 2) the particle sizes of each trajectory for each frame, and 3) the RGB color values of each trajectory for each frame.

```
> ## x and y coordinates for each trajectory (rows), per frame (columns)
> dim(records$trackRecord)
```

```
[1] 15 30  2
```

```
> ## particle sizes (in pixels) for each trajectory (rows), per frame (columns)
> dim(records$sizeRecord)
```

```
[1] 15 30
```

```
> ## RGB color values for each trajectory (rows), per frame (columns)
> dim(records$colorRecord)
```

```
[1] 15 30  3
```

Using `plot()`, these results can be visualized in four different ways (argument `type`). By default, size distributions and track segments are plotted (`type = NULL`). Alternatively, you can plot only size distributions (`type="sizes"`) or only trajectories on the original colorimages (`type="trajectories"`). Here, use argument `frame` to choose which frame should be plotted (by default, the first frame is used, showing the start location for each path). Finally, an AVI animation can be created (`type="animation"`). When doing so, a temporary directory (called 'images') is made to save .png plots of each image. Using `libav`, these are combined into an AVI file and saved in `path`. To use this option, as with using `createImageSeq`, `libav` is required (see Section 10).

The following code compares the simulated trajectories with the estimated trajectories:

```
> ## Plot reconstructed trajectories
> plot(records,type="trajectories",incThres=1)
> ## Add simulated trajectories
> for (i in 1:length(unique(traj$id))) {
+   lines(traj$x[traj$id==i],traj$y[traj$id==i],col="grey",
+         lty=2,lwd=2)
+ }
```

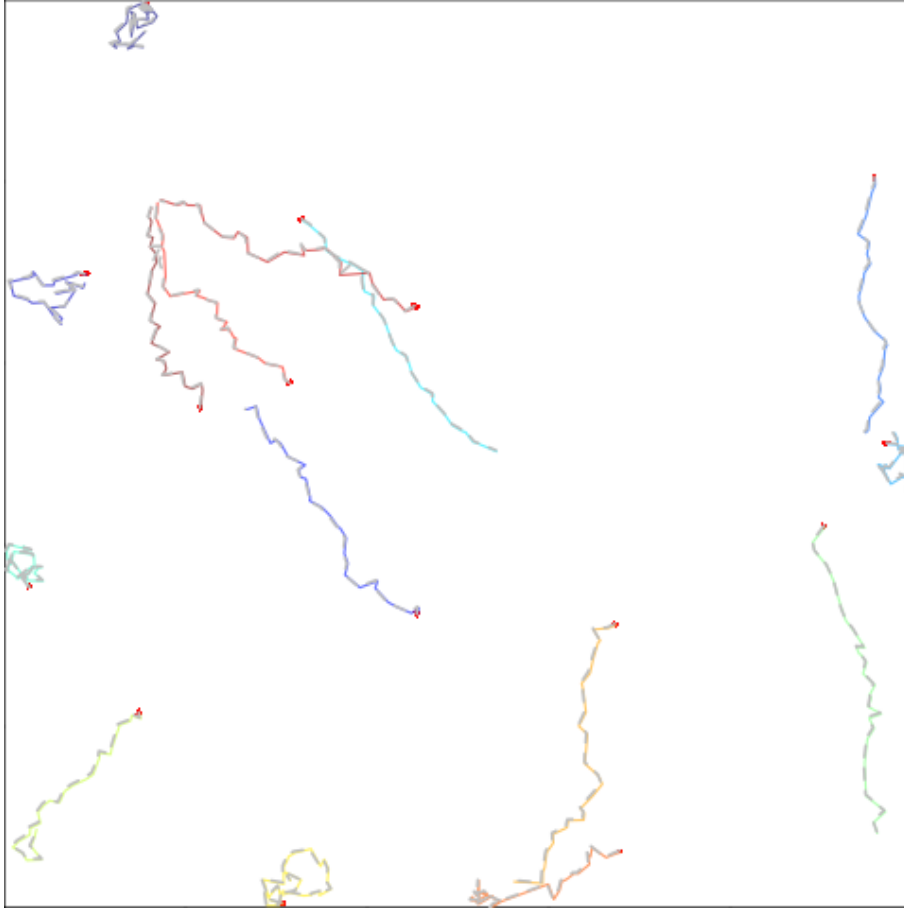


Figure 6: Comparison of particle movement according to simulated trajectories (grey dashed lines) and as obtained using *trackdem* (colored lines).

5 Choice of maximum cost (L) and frame search range (R)

To reconstruct trajectories using the function *trackParticles*, users need to specify a maximum "cost" L . If costs are larger than L , a particle is not linked to a particle, but to a dummy particle. This results in the start or end of a track segment. The choice of L will depend on the particle density and the quality of the movie. If L is set too low, particles are too often linked to dummy particles instead of the correct particle. This will result in an increased number of shorter track segments, possibly leading to an overestimation in number of particles. In contrast, with L values that are too high, particles will be wrongly linked to other particles. In the case of good quality movies, with not much false positives and overlapping particles, the risk of choosing an L value that is too high is not very large: if all particles are visible in each frame, they will all successfully be linked to the right particles in the next frame, even if L is unrealistically high. However, if there is more noise and disappearing particles, an L value too high will result in wrong links.

This is illustrated in Fig. 7. Here we simulated two image sequences, that differ in particle density. In the case of low densities (and no disappearing particles, as in the simulated sequence), the choice of L and R does not have a large effect on the outcome, as long as L is not set too low. At high densities, in contrast, particles will overlap more. This will result in an underestimation, without merging track segments (by increasing R).

In the case of applications to recorded videos that may contain noise and disappearances, L values too high can result in the reconstruction of wrong trajectories. See Fig. 8 for an example where we set L unrealistically high, at 200. To find the best choices for L and R , we recommend

users to plot reconstructed trajectories.

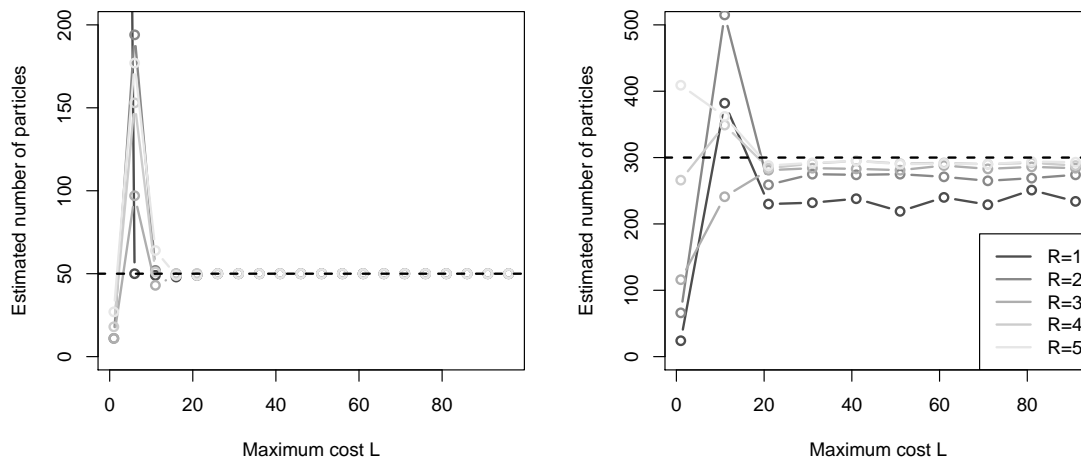


Figure 7: Effect of user-defined L and R . Two image sequences were simulated; one with a low density (left) and one with a high density (right). The number of particles was estimated for different L values, and for different R values (indicated by the different colors). Dotted horizontal lines show the simulated number of particles.

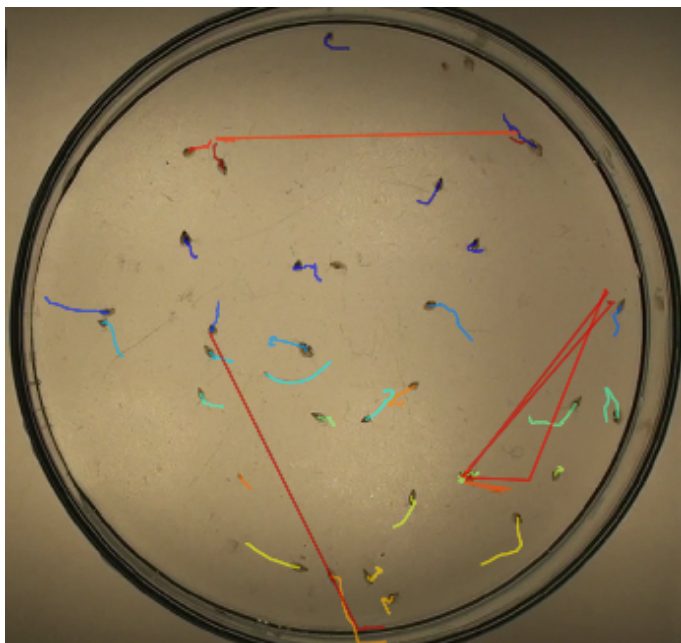


Figure 8: Effect of user-defined L that is set unrealistically high. Whenever particles are not detected in one frame, they are wrongly linked to another particle, that is clearly not the correct particle.

6 Noise removal using artificial neural networks

In the case of no noise and 100% detection, the obtained tracked object using *trackParticles()* gives the complete and accurate trajectories of all particles (Fig. 6). However, when false positives are present (due to e.g. debris, air bubbles, or reflection), a trained artificial neural network can be used as a filter. Also, in the case of a multi-species system, machine learning may be used to distinguish between species. We explain the use of this artificial neural network by simulating an image sequence containing randomly moving noise. Use the following code to simulate the image sequence, and identify all moving particles (including noise):

```
> dir.create("images")
> traj <- simulTrajec(path="images",
+                   nframes=30,nIndividuals=20,domain="square",
+                   h=0.01,rho=0.9,staticNoise=FALSE,movingNoise=TRUE,
+                   parsMoving = list(density = 20, duration = 10, size =1,
+                                     speed = 10, colRange = c(0, 1)),
+                   sizes=runif(20,0.004,0.006))
> ## Run trackdem
> allFullImages <- loadImages (dirPictures="images/",
+                             nImages=1:30)
> stillBack <- createBackground(allFullImages)
> allImages <- subtractBackground(bg=stillBack)
> partIden <- identifyParticles(sbg=allImages,
+                              threshold=-0.1,select="dark",
+                              pixelRange=c(1,500))
```

In the created image sequence, all focal particles are red, and all noise particles have a random color. Use *plot()* and *summary()* to see the output, as described above. Such an image sequence will require further noise filtering as described below.

A neural net requires data, which is then divided into training, validation and test data. To create the required datasets, the following function allows users to manually select false and true positives:

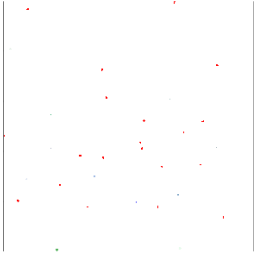
```
> mId <- manuallySelect(particles=partIden,frame=10)
```

Create a training dataset for frame 10

Manually select false and true positives to create a training data set. To do so, select "True positives" (correctly identified particles) or "False positives" (noise). Subsequently, click on the correctly or wrongly identified particles in the right graph. Use "Remove" to delete particles from the selection. Click on "Update image" to reload the image and selected particles, or select "Update automatically" to update after each click. The left graph can be used to zoom; draw a polygon and click on "Update image". Try to select as many particles as possible (> 20 if possible), and when done, click on "Done".

Choose region of interest and click on update.

Update image



Select:

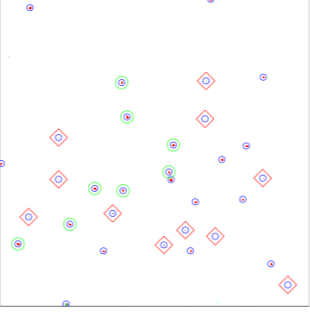
True positives

False positives

Remove

Upload automatically. Slower for larger images.

Done



Selected true positives

| Id | x | y | size |
|----|--------|--------|------|
| 5 | 190.00 | 132.54 | 13 |
| 6 | 198.36 | 186.00 | 14 |
| 18 | 192.17 | 300.58 | 12 |

Selected false positives

| Id | x | y | size |
|----|--------|--------|------|
| 7 | 319.50 | 189.00 | 2 |
| 4 | 321.00 | 130.00 | 1 |
| 8 | 92.00 | 218.00 | 1 |

Figure 9: Graphical user interface to manually select false and true positives using function *manuallySelect*. Blue circles indicate identified particles. In this case, most identified particles are correct (red particles, in this example), but there are several false positives (represented by other colors).

This will open a graphical user interface in which all identified particles are shown in blue, for a chosen frame (argument *frame*) (Fig. 9). Users can select false and true positives. To do so, click on 'True positives' (correctly identified particles) or 'False positives' (noise). Subsequently, click on the correctly or wrongly identified particles. Try to select as many particles as possible, and when done, click on 'Done'. It returns a list of all true and false positives, and a data frame combining the true and false positives with the previously identified particles.

One frame is probably insufficient to create a dataset big enough to train an artificial neural net; the following code can be used to select the three frames that have most identified particles, and create a dataset based on manual identification in those three frames:

```
> # select the nframes with the most identified particles
> nframes <- 3
> frames <- order(tapply(partIden$patchID,partIden$frame,length),
+                 decreasing=TRUE)[1:nframes]
> mId <- manuallySelect(particles=partIden,frames=frames)
```

Now, an artificial neural network can be trained:

```
> finalNN <- testNN(dat=mId,repetitions=10,maxH=4,prop=c(6,2,2))
```

With *maxH* you can set the maximum number of hidden neurons in each layer, and by *repetitions*, you can set the number of repetitions for the training. By default, the input dataset is randomly divided in a training, validation and test dataset, with ratio 8:1:1, respectively. Use *prop*, to change this, by providing a vector containing the ratios. By default, a predefined set of explanatory variables is used, including variables as particle size, perimeter-area ratio, coordinates

of the central pixel, color intensities of the central pixels and neighbors, average R, G, B values and associated variances. Use *predictors* to use a different set, by supplying a character string containing the names of these predictors (note that they must correspond to column names in the object obtained with *manuallySelect*). A principal component analysis can be performed (*pca=TRUE* by default) on the predictors, which ensures that only the variables explaining a high proportion of the total variance are used (by default 0.95, change with argument *thr*). The results can be evaluated:

```
> finalNN

      Trained neural network with 1 layer(s).

> summary(finalNN)

      Summary of trained neural network:
      Confusion table:
      Predicted
Actual  0  1
      0  6  0
      1  0  8
      F-score: 1
      10 repetitions.
      1 hidden layer(s).
```

Now we update the identified particle object, and re-track our particles, this time using the trained neural net to filter noise.

```
> partIdenNN <- update(partIden,finalNN) # update with neural net
> recordsNN <- trackParticles(partIdenNN,L=60,R=3)
> summary(recordsNN)
```

7 Robustness checks

We advise to always check the output of *trackdem* to verify that the right settings have been used and results can be trusted. In our experience, the following set of robustness checks will reveal most commonly encountered problems, if any, and we recommend users always perform these checks:

- Plot the detected background image using *plot(backgroundimage)*. If particles of interest are still visible in the background image, this could suggest that there is too little movement. Try if other background methods work better, and/or increase the number of frames.
- Look at the identified particles per frame, after running *identifyParticles()*. This will help users to diagnose problems with the chosen threshold and size range. If focal particles have not been detected, try a less conservative threshold (use *findThreshold()* to find the optimal threshold). If for instance small particles are overlooked, lower the minimum particle size.
- Look at reconstructed trajectories after running *trackParticles()*. Plot the output using *plot(recordsObject,type='trajectories',incThres=z)*, and visualize the trajectories for different values of *z*. Particularly ensure that there are no wrong links, indicating that *L* is set too high (see Section ??). Create an animation (*plot(recordsObject,type='animation',incThres=z)*) as a final check.
- Compare estimates to manual counts. Always obtain manual counts on a (random) subset of the videos, especially when applying it to a new video set up, and compare those to the automated estimates.

- When applying *trackdem* to many videos at once using the same settings (see Section 8), we recommend that users do the above checks for a subset of the videos. This could be done by creating the suggested plots for all videos by adding relevant lines of code to the loop. These can subsequently be inspected manually. See Section 8 for an example on how this could be done.

8 Batch analysis

After a user is satisfied with the output of a single image sequence, the settings from this single image sequence can be supplied to conduct a batch analysis. This will apply the same settings for all directories containing image sequences in *path*. For instance, we create three images sequences in three folders in a new directory we call "batchTest":

```
> # simulate 3 image sequences
> wd <- getwd()
> folders <- paste0(rep("images",3),1:3)
> populations <- c(15,25,50)
> dir.create("./batchTest")
> setwd("./batchTest")
> for(i in 1:length(folders)){
+   dir.create(folders[i])
+   set.seed(i)
+   traj <- simulTrajec(path=folders[i],
+                       nframes=30,nIndividuals=populations[i],
+                       h=0.01,rho=0.9,sizes=runif(populations[i],0.004,0.006))
+ }
```

We can then do a batch analysis on these sequences with our previous setup stored in "records" and can be viewed as:

```
> attributes(records)$settings

$nImages
 [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30

$BgMethod
 [1] "mean"

$threshold
 [1] -0.1 -0.1 -0.1

$pixelRange
 [1] 1 500

$qthreshold
NULL

$select
 [1] "dark"

$autoThres
 [1] FALSE

$perFrame
```

```
[1] FALSE
```

```
$frames  
NULL
```

```
$R  
[1] 3
```

```
$L  
[1] 60
```

```
$weight  
[1] 1 1 1
```

Alternatively, settings can be provided manually (see `?runBatch`). To run the batch analysis, use:

```
> setwd(wd)  
> batchpath <- "./batchTest"  
> results <- runBatch(path=batchpath,settings=records,incThres=10)
```

```
Directory Count  
1  images1    15  
2  images2    25  
3  images3    50
```

By default, this creates a data frame containing the image sequence names, and the number of estimated particles. To return all information gathered in records, use `saveAll`. This however may be memory hungry and slow down the analysis. Another method to run a batch analysis is to create your own loop. This gives you more flexibility, in the section of code below, any statistic of interest can be calculated or saved to the hard drive. This can especially be useful on systems with low memory or when non-standard statistics are being calculated from for instance the returned coordinates.

```
> ## Manual batch analysis  
> results <- data.frame(movie=folders,n=NA)  
> setwd("./batchTest")  
> for (i in 1:length(folders)) {  
+   direc <- folders[i]  
+   allFullImages <- loadImages (dirPictures=direc,nImages=1:30)  
+   stillBack <- createBackground(allFullImages,method="mean")  
+   allImages <- subtractBackground(bg=stillBack)  
+   partIden <- identifyParticles(sbg=allImages,  
+                               threshold=-0.1, # chosen threshold  
+                               pixelRange=c(1,500)) # min and max size  
+   records <- trackParticles(partIden,L=60,R=3)  
+   ## save the population count to view in R after the run  
+   results$n[i] <- summary(records,incThres=10)$N  
+  
+   ## Check output  
+   pdf(paste0(direc,'.pdf'))  
+   plot(partIden,frame=1)  
+   plot(records,type='trajectories',incThres=10)  
+   dev.off()  
+  
+   ## save the recorded size per particle per frame to the HD
```



```

+ ## access in R with readRDS later
+ saveRDS(records$sizeRecord,file=paste0("sizeRecord_",folders[i],".rds"))
+
+ ## clean previous large objects - this may speed up the run if these
+ ## are large
+ rm("allFullImages","stillBack","allImages")
+ }

```

9 Merge track objects

Whenever longer image sequences need to be analyzed which require too much memory to load into R, it is possible to cut the sequence into shorter fragments. They can subsequently be linked with the function *mergeTracks*. This function requires two objects as returned by *trackParticles*. Particles in the last frame of the first object are linked to particles in the first frame of the second object, based on locations and sizes. Note that to plot the results, users now need to define which colorimages to use.

```

> dir.create("images")
> set.seed(100)
> ## Create image sequence
> traj <- simulTrajec(path="images",
+                     nframes=60,nIndividuals=20,domain="square",
+                     h=0.01,rho=0.9,movingNoise=FALSE,
+                     sizes=runif(20,0.004,0.006))
> dir <- "images"
> ## Load and analyze images 1-30
> allFullImages1 <- loadImages (dirPictures=dir,nImages=1:30)
> stillBack1 <- createBackground(allFullImages1)
> allImages1 <- subtractBackground(bg=stillBack1)
> partIden1 <- identifyParticles(sbg=allImages1,
+                               pixelRange=c(1,500),
+                               threshold=-0.1)
> records1 <- trackParticles(partIden1,L=20,R=2)
> rm(list = c("allFullImages1", "stillBack1", "allImages1",
+            "partIden1"))
> gc()
> ## Load and analyze images 31-60
> allFullImages2 <- loadImages (dirPictures=dir,nImages=31:60)
> stillBack2 <- createBackground(allFullImages2)
> allImages2 <- subtractBackground(bg=stillBack2)
> partIden2 <- identifyParticles(sbg=allImages2,
+                               pixelRange=c(1,500),
+                               threshold=-0.1)
> records2 <- trackParticles(partIden2,L=20,R=2)
> ## Merge tracked objects
> records <- mergeTracks(records1,records2)
> plot(records,colorimages=allFullImages2,type="trajectories")
>

```

10 Creation of image sequence

In our example, we have used a simulated image sequence. However, when interested in live organisms, a few additional steps will be required. The first step is to record a short movie

on the population. Note that since *trackdem* is based on movement detection, it is crucial that particles of interest move. Furthermore, it is recommended to use a stable background, and strong (background) illumination. A video of approximately four seconds will generally suffice, and can be of any extension supported by libav (e.g. .MTS, .MP4 or .AVI; see below).

Second, the movie must be converted into an image sequence. This can be done using the function *createImageSeq*, as explained below. This requires the installation of Python 2.7 (not 3.6), libav and ExifTool. Note that installing Python 2.7 and ExifTool are required *only* if you want to use *trackdem* to create image sequences. Libav, in contrast, is also required to create animations, as explained in Section 4. All other functions run without the need to download additional software.

Ubuntu users can paste the following commands in a terminal to install libav and ExifTool (Python 2.7 should be included by default):

```
> # sudo apt-get update
> # sudo apt-get install libav-tools
> # sudo apt-get install libimage-exiftool-perl
```

This has been tested on Ubuntu 16.04.

Mac users can paste the following commands in a terminal to install libav:

```
> ## Make sure that homebrew is installed, see: https://brew.sh/
>
> ## Install libav
> # brew install libav
```

ExifTool can be downloaded from <http://www.sno.phy.queensu.ca/~phil/exiftool/install.html>; follow the installation instructions for the OS X Package. The newest Python 2.7 release, if not installed yet, can be downloaded from <https://www.python.org/downloads/mac-osx/>. This has been tested on MacOS 10.12.5 Sierra.

Windows users can download libav from <http://builds.libav.org/windows/>. Download the latest nightly-gpl release (<http://builds.libav.org/windows/nightly-gpl/?C=M&O=D>), and extract all files to a chosen location. Next, download the file named libgcc_s_sjlj-1.dll from <http://builds.libav.org/windows/>, and place it within the libav directory, in '/usr/bin'. ExifTool can be downloaded from <http://www.sno.phy.queensu.ca/~phil/exiftool/install.html>. For ExifTool, download the stand-alone executable and place the exiftool(-k).exe file in a chosen directory. For convenience, you can change the name to exiftool.exe, as described in the installation instructions. Finally, Python 2.7 can be downloaded from <https://www.python.org/downloads/windows/>. Follow the instructions for installation. This procedure was tested on Windows 7 and Windows 10.

Once all required software is installed, the function *createImageSeq* automatically converts each video into an image sequence. First, place all movies that need to be converted in one directory. Second, create a directory in which the image sequences should be saved. By default, directory names 'Movies' and 'ImageSequences' are used, and can be set with arguments *moviepath* and *imagepath*. If the directories containing the movies and image sequences are not located in the working directory (which can be set with *setwd()*), provide the full path to the locations. The function creates a temporary Python file (tmp.py) in *path* (which is by default set to the working directory). Per video, a directory within *imagepath* is created, containing .png images. The function will create directories that contain both the movie name and the recording date (if stored in the raw movie file). Videos that are already converted and saved in *imagepath* will be skipped. To convert a movie again to an image sequence, first manually delete the directory containing the images.

```
> createImageSeq(imagepath="ImageSequences", moviepath="Movies",  
+               fps=15, nsec=2, ext="MTS", libavpath=NULL,  
+               exiftoolpath=NULL, pythonpath=NULL)
```

With arguments *fps* the number of images per second can be set, and *nsec* defines the time frame at the center of each movie file that is used. For instance, in a 10 seconds movie file, a *nsec* of 2, will convert the portion of the movie after second 4 to second 6 into an image sequence. Make sure that the extension set with *ext* matches the extension of the videos (e.g. AVI or MP4).

With arguments *libavpath*, *exiftoolpath* and *pythonpath* users can set the location of the required executable files. Linux and Mac users can leave these arguments empty. Windows users will need to specify the location, or add the path variables to the Environmental Variables. For example, use *libavpath='C:/Users/USERNAME/libav/usr/bin/avconv.exe'*, where USERNAME is your username, or if libav is located elsewhere, supply the directory where you placed it. For ExifTool, use for instance *exiftoolpath='exiftool(-k).exe'* (if saved in the working directory), and for Python use *pythonpath='C:/Python27/python.exe'*. How to add the path variables to the Environmental Variables will depend on the version of Windows. See for instance [this webpage](#) for an explanation.

Alternatively, users can skip this step, and convert a video into a image sequence using other software. Once this is done, you can continue with *loadImages*, as described in Section 2.