Particle tracking using R

Ian Williams[1]

[1]*H.H. Wills Physics Laboratory, Tyndall Ave., Bristol, BS8 1TL, UK*

(Dated: September 2, 2014)

An introduction to performing particle tracking analysis on two-dimensional data using R. The routines described were developed by the author starting in September 2013 and are based on those developed by Crocker and Grier [1] and published online by Eric Weeks [2]. The original IDL routines have been translated into R and tweaked for use with brightfield images, although they should still function for confocal data.

I. GETTING STARTED

A. Software

The first thing you need to do is ensure R is installed on your computer. I believe that this is automatic on the managed Windows PCs. If, however, you find that R is not installed then you can download it from the R Project website [3]. Additionally I recommend using an integrated development environment such as RStudio [4], which will make the whole process much friendlier.

Once R and RStudio are installed, open up RStudio. Welcome to R. This is your new home. The particle tracking code described below relies on the EBImage [5] package from Bioconductor [6] to handle the reading, writing and manipulation of .tiff images. Initially, you need to manually install this package. This only needs to be done once. Paste the following into the R command line (most likely the bottom left panel in RStudio):

```
source("http://bioconductor.org/biocLite.R")
biocLite()
```

You can then load the EBImage library using

```
library(EBImage)
```

If all goes well you will receive no error messages and EBImage will be loaded. You will need to load EBImage every time you start RStudio to do particle tracking. If at any point in this process you receive error messages it is likely an issue with permissions and filepaths and you may need to contact the IT service desk for advice on how to proceed. They sorted me out when my packages and ability to install new packages strangely disappeared in August.

Everything else you need to track your particles and perform some common analyses is included in the TrackPack folder. You will most likely need to write additional data analysis routines to extract the quantities of interest from particle trajectories, and for that you may well need to become competent in R (or some other language).

B. R Basics

Okay, so now you have RStudio open and have learned how to install a package using `library()`. I like to have RStudio set up with four panels open. Firstly I have the command line in the bottom left, into which I can type commands and tell R what to do. Above the command line I have a panel showing any R source code I have written. This is where the routines I have written for you will go. Let's give this a go now with a simple test function that generates some points on a sine curve. Locate `gensin.r` in the TrackPack folder and open it in RStudio. You should see the source code appear in the top left panel. This code defines a new function called `gensin`(). However, this function is not defined until we run `source()` on it. Within RStudio you can run `source()` by clicking the Source button at the top right of the active code panel. If you do this, you will define the new function `gensin()`.

The next panel to be concerned with is the Workspace and History panel on the top right. If you don't see this then you can enable it by selecting View and Workspace. The workspace summarises any variables and functions *etc.* you may have defined. Hopefully you can see the function `gensin()` listed there. The history tab contains a list of all the commands you have entered at the command line. Both of these are very useful. If we now define a new variable called `test` by running `gensin()` you will see a matrix named `test` appear in the Workspace tab. Do this now by typing the following into the command line:

```
test <- gensin()
```

Now you should see an object called `test` listed in the Workspace tab. Next to the name you will see that `test` is a 100×2 double matrix. What this means is that `test` is a matrix object consisting of 100 rows and 2 columns. What the function `gensin()` has done is generate this matrix with equally spaced x values in the first column and $y = \sin x$ in the second column. You should note in this command that the arrow, `<-`, is used instead of an equals sign to put the output of `gensin()` into the variable `test`.

This brings us to the fourth panel in RStudio, which has a number of tabs. The ones I use most frequently are Plots and Help. You can see the Plots tab in action if you plot `test` using

```
plot(test)
```

Now a plot of your sine function should have appeared in the Plots tab in the lower right panel. I'll discuss plotting in more depth in Section I B 2. The lower right panel is also where you can read the help files on many R commands and concepts. To access the help for, for instance, trigonometric functions type

```
help(sin)
```

This works for pretty much anything you may have problems with, except the code I have written for which you will need to refer to this document or send me an email.

1. Reading and writing text files

The way I store data that I have generated in R is in tab-spaced text files. Please note that, while everything I am writing in this document *should* work, it may not actually be the best way to do things in R. I learned R hastily after having used IDL for a long time, so my approach to using R is quite IDL-like. As such, I like to use two-dimensional array objects rather than data frames.

Let's first write `test` to a text file. You do this using `write()`:

```
write(t(test),file="C:\\Desktop\\test.dat",ncolumns=2,sep="\t")
```

This creates a text file on your desktop called `test.dat` that contains the contents of `test`. The first argument of this command, `t(test)`, specifies that we want to write out `test` (or, more correctly, the tranpose of `test`). The second argument is the filepath of the file we want to write into. Note that you must use double backslashes in the filepath. You can change this path to be wherever you want to save a file. The third argument is the number of columns in the data. For `test` this is 2. You must always specify the number of columns when writing out data this way, and if you specify the wrong number of columns your file will be garbled. Lastly, `sep="\t"` specifies that we want to space out the data with tabs in the output file. You should be able to adapt this `write()` command to write out any data you create using R.

Reading in data from text files such as that which you have just created is a two step process which first reads the data in as a table and then converts that table to a matrix (since I like to work with matrices). You can read your test data back in using:

```
test2 <- read.table("C:\\Desktop\\test.dat",sep="\t")
test2 <- data.matrix(test2)
```

Running this will read the test data back in and put it into a matrix named `test2`, which will now hold the exact same information as `test`.

Congratulations, you now know how to read and write text files from R. This is main way I get numerical data into and out of R.

2. Plotting

While it is possible to use R to make beautiful plots I tend to use Origin for this purpose, so my knowledge of R plotting is limited to basic quick-and-dirty plotting in order to see how something looks. By all means go ahead and learn how to make publication quality plots in R (you'll probably want to look into using ggplot2 for this [7]), but all I am going to teach you here are the basics of R plotting.

By now you should have already plotted `test` using `plot(test)`. This only works because `test` contains two columns representing x and y. In general, the syntax is `plot(x,y)`. So you could get the same result using the command

```
plot(test[,1],test[,2])
```

Here we are explicitly telling plot that the first column of `test` contains our x values and the second column contains the corresponding y values. Previously R managed to figure that out by itself. Note that this form of using square brackets to specify specific elements of an array is very common in my code and you will probably want to familiarise yourself with it. Let's say I want to to see what number is stored in the 5th row, 2nd column of the matrix `test`, then I could type `test[5,2]` at the command line. The row is always specified first and the column second. In the above plotting example I used `test[,1]`, I did not specify a specific row, only a specific column, which means I told `plot()` to use the whole of the first column of `test`. This is one way to select a single column from a larger array.

By default R plots graphs with points. But what if I want to use a line instead? Well then we nead to pass an additional argument to the `plot()` command:

```
plot(test[,1],test[,2],type="l")
```

Here `type="l"` specifies that we want lines instead of points. You could also use `type="b"` to obtain both points and lines.

Lastly, you can add additional lines or points to an existing plot using `lines()` and `points()` respectively. Let's try now by plotting another, different sinusoid on the same axes as we have plotted `test`. We can do this by doing some maths to `test` within the `lines()` command:

```
lines(test[,1],test[,2]/2,col="red")
```

This new line has half the amplitude of the original line and is drawn in red thanks to the `col="red"` argument in the command. By this point your plot window should look something like Figure 1.

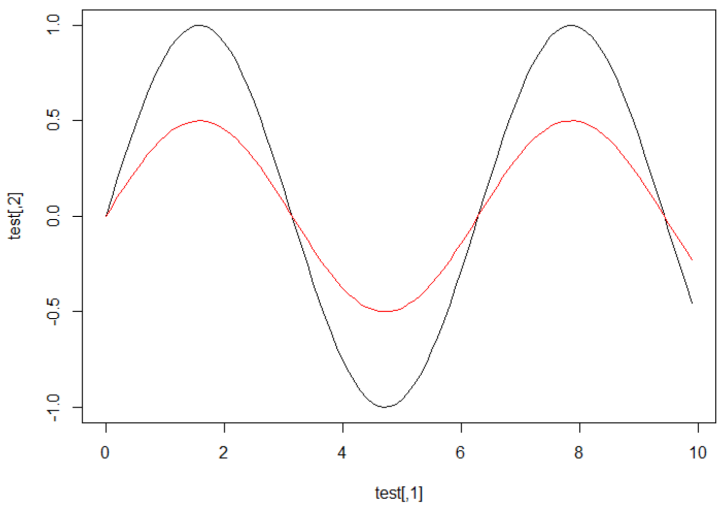

FIG. 1: The results of the quick plotting tutorial.

I exported the image shown in Figure 1 using the Export button in the Plot panel in RStudio and selecting Export Plot as Image. This is pretty useful for getting plots out of R if you feel inclined to do so.

3. The 'which' command

The last part of this introduction to using R is concerned with the `which()` command, which I think is one of the most useful little things you can learn. It behaves much like `where` in IDL and allows to select a subset of your data. For instance, what if I want to work with `test` only where its second column is positive. Well, I can do this by defining a variable I will call `w` thusly:

```
w <- which(test[,2] > 0)
```

Now `w` is a list of all the places where the second column of `test` is positive. You can now plot this subset of your data using

```
plot(test[w,])
```

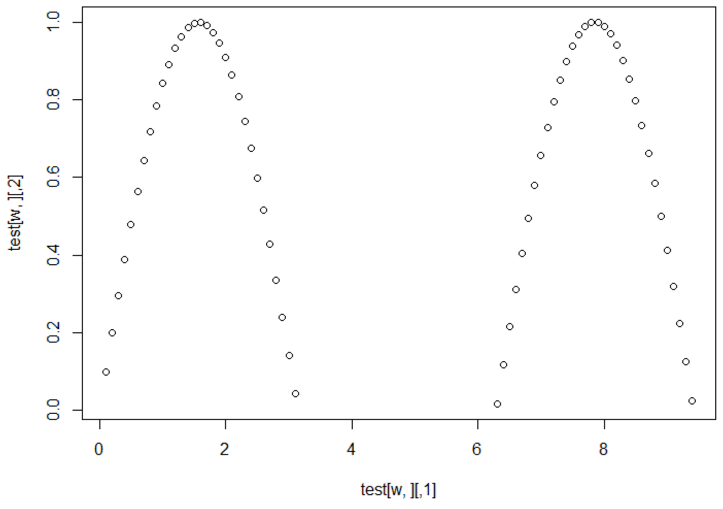

FIG. 2: Plotting only the positive values of the test data.

Figure 2 shows the result of this. What you have told `plot()` to do here is to only consider those rows of `test` for which the second column is positive. You could also define a whole new variable called, *e.g.* `testpos` containing only these values of `test` by using `testpos <- test[w,]`. The `which()` command allows you do all sorts of useful data subsetting that you will most likely find useful in your adventures with R.

II. PARTICLE TRACKING

Now you should know enough about R to start tracking some of your data. A good tracking tutorial is available on Eric Weeks website [2], and since my R code is adapted from his IDL code, much of the same advice applies here, although the language syntax is slightly different.

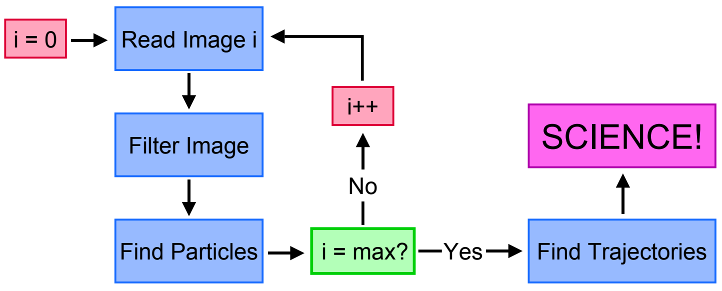

FIG. 3: Flow diagram showing overview of particle tracking procedure.

A schematic overview of the basic particle tracking procedure is shown in Figure 3. The software considers each image in a sequence separately. Each image is, in turn, read into RStudio, then subjected to some image processing filters to improve the image quality. The image processing methods employed may vary depending on the microscope from which images are obtained, but in general some sort of bandpass filter is applied in order to remove short wavelength noise and long wavelength variations in brightness. After filtering the particles in the image are identified based on local maxima in brightness. This procedure repeats for all images in the sequences and then finally the particle locations in each frame are linked together to form trajectories. Your final task, of course, is to use these particle trajectories to do some exciting science!

A. Pre-Pretracking

Before you can batch process a whole sequence of images you need to figure out the best values for a number of parameters. To do this I normally manually perform a number of operations on the first image in my sequence and assume that if it works for image zero then it should work for the whole sequence. This may not be a good assumption for confocal data if your particles bleach significantly through the experiment. In that case you may to write yourself some slightly cleverer code (or some how convince me to do the same). Throughout the tutorial I will illustrate the process using data from both brightfield and confocal so you know the sort of thing you are aiming for. These test images are included in the TrackPack folder so you can follow the examples. I am assuming that you have put this folder on your Desktop. If you have placed this folder elsewhere you will need to modify the filepaths in the examples.

Firstly you must make sure that the EBImage package is loaded using `library(EBImage)`. This will allow you to read in images using `readImage()`. The code is designed to work with a greyscale image and so you should combine `readImage()` with `channel()` which either extracts a single channel from the input (red, green or blue) or merges all three channels into a single greyscale image. The commands to read in the two test images are:

```
imgbf <- channel(readImage("C:\\Desktop\\TrackPack\\brightfield.tif"),"grey")
imgcn <- channel(readImage("C:\\Desktop\\TrackPack\\confocal.tif"),"green")
```

This creates two image objects named `imgbf` for the brightfield image and `imgcn` for the confocal image. Note that for the brightfield image you should use the argument `"grey"` in `channel()` but when using confocal data you should use the correct channel for your image (in this case `"green"` but also sometimes `"red"`).

This images can be viewed using the `display()` command: `display(imgbf,method="raster")`, where the first argument is the name of the image object (in this case the brightfield image) and the second argument tells it to display the image inside of RStudio rather than opening an external image viewer. Figure 4 shows the two images as displayed in RStudio.

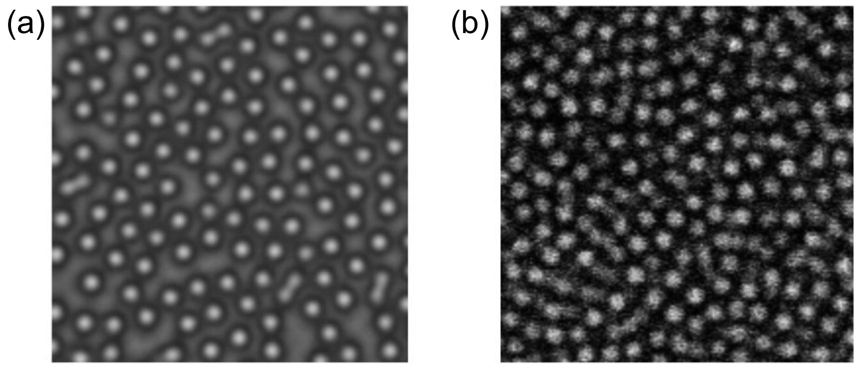

FIG. 4: Test images from (a) brightfield microscopy and (b) confocal microscopy as displayed in RStudio.

For good particle tracking you ideally want your particles to appear as bright circular objects on a black background with the maximum in brightness near the particle centre. To do this you should use the function `lowpass.r` from the TrackPack folder. This is the main image processing routine that I use. Firstly you will need to open it in RStudio and compile it using the Source button to create the function `lowpass()`. The commands for running `lowpass()` on your images are:

```
lpbf <- lowpass(imgbf,lobject=5,bgavg=5)
lpcn <- lowpass(imgcn,lobject=11,bgavg=11)
```

This creates two new processed images called `lpbf` and `lpcn` for the brightfield and confocal data respectively. You can display these images using the `display()` command, such as is shown in Figure 5.

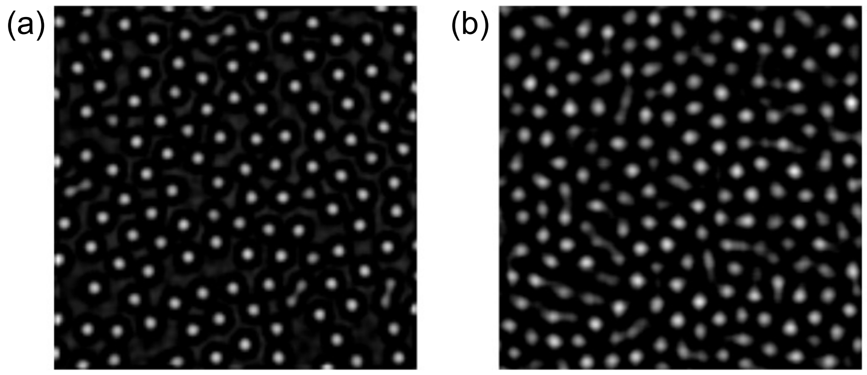

FIG. 5: Processed images from (a) brightfield microscopy and (b) confocal microscopy as displayed in RStudio.

There are two arguments that you need to worry about, `lobject` and `bgavg`. By changing these numbers you can alter the quality of the image processing. Note that because of reasons, both of these numbers must be integer and odd. Best results are often obtained when both `lobject` and `bgavg` are approximately equal to the diameter of your particles in pixels. If you choose correctly you should end up with a fairly nice image consisting of bright particles on a black background. You should make a note of the parameters that you deem best for your data as you will need them later when it comes to batch processing your entire image series.

Next you must attempt to find particles in your processed image. This is done using the `feature()` function, found in the TrackPack folder. As before you must compile this function before it will work. `feature()` is run on the test images using:

```
fbf <- feature(lpbf,diameter=9,masscut=3,minimum=0.3)
fcn <- feature(lpcn,diameter=11,masscut=5,minimum=0.25)
```

which creates arrays named `fbf` and `fcn` for the brightfield and confocal data respectively. On running `feature()` you see it tell you how many particles the software has found. The resulting arrays have five columns and one row for each identified particle. I will describe the contents of all these columns later in this document, but for now it is sufficient to know that the Cartesian co-ordinates of each particle are stored in the first two columns of these arrays.

The way `feature()` works is to initially look for locally brightest pixels and take these as candidate particle positions. Here locality is defined by the argument `diameter`. If no pixel within a circle of the specified diameter is brighter than pixel i then pixel i is considered a candidate particle location. In practise, `diameter` should be set to approximately the particle diameter in pixel. Additionally, `diameter` must be an odd integer.

It is useful now to take a moment to understand exactly how the image data is handled by R. An image is a two-dimensional array of numerical values encoding the brightness of each pixel. Each pixel is represented by a number between 0 and 1, where 1 represents white and 0 represents black. So the locally brightest pixel is that element of the local image having the largest numerical value. The problem with using the locally brightest pixels as particle co-ordinates is that it is comparative measure rather than an absolute measure. A dark grey pixel surrounded by black pixels is locally brightest and thus is identified as a particle position, but in reality it is likely only some noise in the image background. Clearly you do not want to be identifying parts of the image background as particles. This is where the final two arguements of `feature()` come in. If you try running `feature()` with both `masscut=0` and `minimum=0` you will see that the software identifies many more particles than are really there. These are spurious features that need to be removed.

Firstly, `minimum` specifies the minimum value a pixel must have to be considered a particle location. Since this is a single pixel brightness value it can be any value between 0 and 1, although I find values between 0.2 and 0.5 are usually suitable. `feature()` rejects any locally brightest pixel whose brightness is less than `minimum`. This throws away many of the mistakenly identified background features.

In addition to `minimum`, `feature()` also takes the `masscut` argument. This is a second cut-off type parameter which, instead of looking at only the brightness of the central pixel of a particle, considers the sum of the brightness of all pixels within a circle of size `diameter` of the identified locally brightest pixel. This sum is also referred to as the integrated brightness of a particle. `feature()` rejects any particles with an integrated brightness less than `masscut`.

You will likely need to run `feature()` with a number of different values for all the parameters before you are happy with the particle identification. In order to compare the identified particle locations to the original image, or the processed image, you should use the `overcirc()` function (including in TrackPack) run using:

```
overcirc(lpbf,fbf,rad=3)
overcirc(lpcn,fcn,rad=4)
```

This draws circles centred on the identified particle locations as an overlay to the processed images. The circle radius can be changed by altering the argument `rad`. Figure 6 shows these overlays for the test data using the parameters specified above.

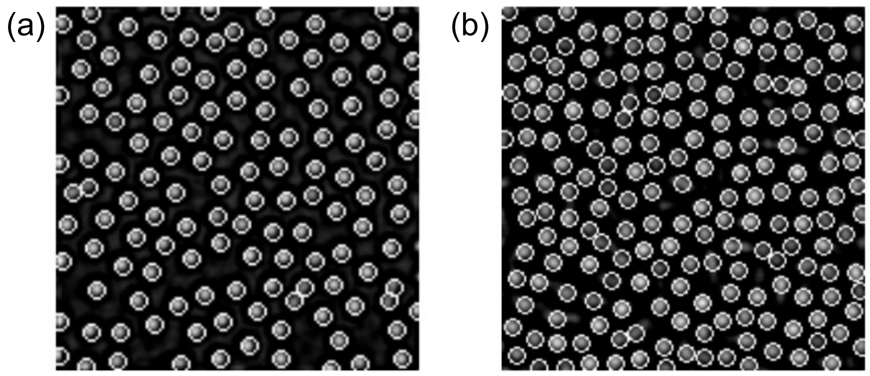

FIG. 6: Overlay of identified features on processed images from (a) brightfield and (b) confocal microscopy.

While it is fairly easy to find good values for `minimum` and `masscut` through trial and error and using `overcirc()`, it can be trickier to judge the best value for `diameter`. In theory, `feature()` can locate particles with accuracy down to $1/\sigma$ pixels, where σ is the diameter of the particle image in pixels. So far I have only discussed finding the locally brightest pixel. Subpixel accuracy is obtained by taking these initial integer pixel position estimates and refining them by performing a brightness weighted centroiding over all pixels within a circle of size `diameter` of the locally brightest pixel. In this way you can obtained subpixel positional information. One useful way to assess your choice of `diameter` is to consider the subpixel part of the position — *i.e.* testing for pixel biasing. Ideally you would expect the distribution of the fractional parts of the particle x and y positions to be flat. However, if your choice of `diameter` is too small, particle positions tend to be biased towards the nearest whole pixel. You can check for pixel biasing by plotting the histogram of the fractional part of particle x and y positions using

```
hist(fbf[,1] - round(fbf[,1]))
hist(fbf[,2] - round(fbf[,2]))
```

where `fbf[,1]` contains all the x co-ordinates and `fbf[,2]` contains all the y co-ordinates of particles in the brightfield image. If these distributions show a peak around 0 then you should try increasing `diameter` until it is approximately flat. Of course you should also not make `diameter` too large, as `feature()` also imposes a minimum separation between identified particles using this parameter.

Through repeated use of `feature()`, `overcirc()` and checking for pixel biasing you should be able to find a set of parameters that produce particle locations that you are happy with. If you are having trouble obtaining satisfactory results you may need to take a step backwards and try different image processing parameter in `lowpass()`. Be aware that the software is easily confused near the edges of the images, so it may be difficult to get good data in these regions. You can always ignore particles too close to the edge later in your analysis.

If, however, you find it difficult to remove a few spurious particles from your set of identifications don't worry, there are still a few more tools at our disposal. I previously stated that `feature()` outputs an array consisting of 5 rows and one column for each identified particle. You may have realised by now that the first two columns contain the particle positions. The remaining three columns are also useful — they contain, the integrated particle brightness (column 3), an estimate of the particle radius of gyration (column 4) and the particle eccentricity (column 5). The integrated brightness has already been introduced when discussing `masscut`, the radius of gyration gives an estimate of the particle size and the eccentricity characterises how circular the particle image appears with more circular features having eccentricity close to 0 and elongated objects taking larger values up to 1. You can plot histograms of these particle characteristics using *e.g* `hist(fbf[,5])` to see the distribution of eccentricities in the brightfield data. See the help file for `hist()` (`help(hist)`) to learn more about R's histogram function including how to manually specify the binning to use. You can also plot these quantities against one another using *e.g.* `plot(fbf[,3],fbf[,5])` to see the brightfield data in the brightness–eccentricity plane. This sort of analysis is described more fully in Eric Weeks' particle tracking tutorial [2]. By combining these sorts of visualisations with the `which()` command (see Section I B 3) and maybe some of R's logical operators (ask Google) you can selectively manually discard spurious features. Analysis of this sort is also useful for distinguishing large and small particles in binary samples.

The good news is that you have now done most of the hard work. You should have successfully found a set of parameters that produces good particle co-ordinates in the first image of your sequence. Remember these parameters as you will need them in following section for batch processing. When I particle track a video I usually save my chosen parameters in a text document in the folder with the data so I can always remember my choices and maybe save time in the future when tracking similar images.

B. Pretracking

After the previous section you should have a set of pretracking parameters which successfully identify particles in the first frame of your video. The parameters I have chosen for the test brightfield and confocal images are summarised in Table I. Additionally you may have decided on some manual cut-offs in total brightness, radius of gyration and

Parameter	Brightfield	Confocal	Meaning
`lobject`	5	11	Lengthscale for filtering
`bgavg`	5	11	Lengthscale for background smoothing
`diameter`	9	11	Approximate particle size
`masscut`	3	5	Minimum integrated particle brightness
`minimum`	0.3	0.25	Minimum brightness of central pixel

TABLE I: Pretracking parameters determined in Section II A for brightfield and confocal data

eccentricity which are not applied automatically but must be imposed manually.

To perform the pretracking analysis to an entire image sequence and thus obtain particle co-ordinates for a whole video you need to use the `pretrack()` function which automatically applies everything from the previous Section to each image in a sequence. To do this, `pretrack()` requires that you have a folder containing all your images. The filenames of these images can be anything you like as long as they are all identical and end in sequential 5 digit numbers starting with 00000 and have the file extension `.tif`. For example, if you have a sequence of 2000 images, the first might be named `colloid_image_00000.tif`, and the last would then need to be called `colloid_image_01999.tif` with the intervening images number sequentially.

The `pretrack()` function takes a lot of arguments as it contains many options for different circumstances. Some of these arguments are set by default to certain values that I have previously used regularly. If you do not specify an argument its default value will be used, which may lead to unexpected results. To run `pretrack()` on an image sequence you should use:

```
pt <- pretrack(filename="C:\\Data\\colloid_image_",images=2000,diameter=9,filter=5,bgavg=5,
masscut=3,minimum=0.25,chan="grey")
```

Firstly we define the filename stub for the images including the whole filepath but leaving off the number and `.tif` from the end of the image names. Next we specify the number of images in the sequence with the `images` argument. `diameter`, `bgavg`, `masscut` and `minimum` have the same meanings as previously and you should replace the above values of these arguments with the parameters you have chosen. Confusingly, for some reason, the parameter that was previously named `lobject` is renamed `filter` here, so you should replace the above value for `filter` with your value for `lobject`. The last argument, `chan`, determines which channel you will use when you read in each image. It defaults to `"grey"` as I tend to work with brightfield images. For confocal analysis you will want to replace this with (probably) `"green"`. If you paste the above into RStudio now it will not work, as this is just an example of the command and no such data set exists.

In addition to the above parameters you can also optionally specify cropping and rotation to be applied to the images. If you need to use this functionality you should investigate the comments in the `pretrack.r` source code. The default behaviour is no cropping and no rotation.

Depending on the length of your video, the number of particles in your images and the size of your images `pretrack()` can take a while to run. On completion it will output an array, named `pt` in the above example command, which consists of 6 columns and, most likely, many many rows. Each row represents a single particle in a single frame. The first five columns are the same as the output from `feature()`, while the final column contains the image number. Inside the code the image or frame number is the natural unit of time, so a particle location in the first image is considered to be at time $t = 0$. Similarly, the natural unit of length is the pixel. Of course, later in your analysis you will need to convert these fairly meaningless units into scientifically relevant units.

The contents of the output of `pretrack()` are summarised in Table II.

Column	1	2	3	4	5	6
Contents	x position	y position	Total brightness	Radius of gyration	Eccentricity	Frame number

TABLE II: The contents of a pretrack array. Each row correspsonds to an identified feature in a single frame.

You will likely want to write your pretrack array to a file in the manner discussed in Section I B 1. This is done with the command:

```
write(t(pt),file="C:\\Desktop\\TrackPack\\pretrack.dat",ncolumns=6,sep="\t")
```

where you must remember to specify the number of columns in order to correctly store the data.

Now you know the locations of all your particles in all the frames of your video. If you are only interested in structural measures such as $g(r)$ or the Voronoi tesselation then this information is sufficient. Dynamic quantities, however, require you to know particle trajectories — that is, you need to know which particle is which in every frame so that you can follow a given particle in space and time. This is described in the next Section.

C. Tracking

The final stage in particle tracking is to link the particle locations in each frame into trajectories. This is done using the `iantrack()` routine. This is perhaps the most fragile part of the analysis as, rather than translate John Crocker's tracking routine from IDL into R, I wrote my own. The tracking routine assigns a unique particle identification number to each particle. It works simply by comparing the particle locations in frame $i + 1$ to those in frame i and assuming that a particle in frame $i + 1$ is the same particle as that in frame i that is nearest to its new location (within some cut-off radius). Essentially `iantrack()` takes a guess at which particle is which by minimising the particle displacements between frames. The smaller the distance moved by particles between frames the better this code performs, and therefore it is important to use a sensible frame-rate when acquiring videos such that particles do not move too far between images. A good rule of thumb is that "too far" is of the order of a particle radius. As long as your particles move less than their radius between frames then trajectories should be found reliably.

To run `iantrack()`, use the command

```
tr <- iantrack(pretrack=pt,maxdisp=5,imgsize=c(512,512),goodenough=10)
```

where `pt` is your pretracked data. The argument `maxdisp` is the range over which the code will look for the next particle location, *i.e.* it is the maximum displacement of particles between frames in pixels. The code will run more quickly if `maxdisp` is smaller, but decreasing it too far may result in poor tracking if particles are moving further than `maxdisp`. `imgsize` is simply a two element vector containing the x and y dimensions of your original source images, in my example 512×512 pixels. The final argument, `goodenough`, is the minimum length of an acceptable trajectory in frames. Any trajectories that persist for less than `goodenough` frames are ignored. This allows you to throw away any remaining spurious features that may appear for only one frame. Please note that `iantrack()` does not currently have some of the functionality included in John Crocker's IDL tracking routine such as a memory, allowing for brief disappearances of particles.

The resulting output from `iantrack()` contains all the data previously included in your pretrack data plus an additional seventh column which contains the ID number of each particle in each frame. So now you know which particle is which. You can, for instance, plot the trajectory of particle number 5 in this manner:

```
w < - which(tr[,7]==5)
plot(tr[w,1],tr[w,2],type="l")
```

And hopefully by now you can figure out how this process works. Note that I use `==` for the comparison of values in the `which()` function. In R, a single equals sign is used to assign values to arguments when calling a function and a double equals sign is used for comparisons.

So, the last thing to do is to write your tracked data to a file:

```
write(t(tr),file="C:\\Desktop\\TrackPack\\track.dat",ncolumns=7,sep="\t")
```

where again it is important to specify the number of columns in your data (in this case 7).

Congratulations, you have extracted particle trajectories from two-dimensional microscope data. Now you have to do some science with this information.

III. ADDITIONAL ROUTINES FOR COMMON ANALYSES

Finally, I have included in TrackPack some routines for very common analyses, such that you don't have to figure out how to write these pieces of code. I will not describe in depth how the code for these works, but you are welcome to delve into the source code and figure it out for yourself. All of the functions describes are saved in individual R source files in the TrackPack folder and need to be compiled before use.

A. Radial distribution function, $g(r)$

The function `gr2d()` calculates $g(r)$ for two-dimensional data from either pretrack or track data obtained from microscope images which have a finite field of view and no periodic boundary conditions (*i.e.* this routine is only suitable for the analysis of experimental data, not simulated data). The bulk of the code is concerned with the normalisation when the field of view is finite and is translated from Eric Weeks' IDL code. Calculating $g(r)$ can be quite slow when you have a large number of particles as it requires calculating the distance between all pairs of particles. Therefore, when working with many particles I tend to calculate $g(r)$ using data from only a few frames of the total video in order to save time. To find $g(r)$ use:

```
gr <- gr2d(tr,nbins=100,deltar=0.25,imgsize=c(512,512))
```

Here I am using tracked data, `tr`, but you can just as easily use pretrack data as $g(r)$ is a purely structural quantity and does not care which particle is which. The arguments `nbins` and `deltar` set the number of bins and the binwidth (in pixels) respectively, so in the above examples I am considering $g(r)$ in 100 bins, each with a width of 0.25 pixels, which results in calculating $g(r)$ from $r = 0$ up to $r = 25$ pixels. You will need to adjust these quantities in order to obtain good results. `imgsize` is again the dimensions of your original source images in pixels.

The output for `gr2d()` is a two column matrix with r in the first column and $g(r)$ in the second column. You can plot it in R, or you can save it as a text file and import it into Excel or Origin or your favourite plotting program.

B. Hexagonal bond-orientational order parameter, ψ_6

ψ_6 is a local bond-orientational order parameter defined for each particle based on the hexagonality of its local environment. For more information on its definition I recommend reading [8] or [9]. It is a complex number, the magnitude of which indicates the degree of hexagonal ordering and the argument of which indicates the local orientation of the hexagonal lattice. The calculation of ψ_6 relies on functions in the package `tripack` which also includes tools for Voronoi analysis. You can find information about `tripack` online. All you need to do is download the package within RStudio by clicking on Tools in the menu bar and then Install Packages. Start typing 'tripack' into the middle Packages box and you should see it autocomplete. Then click Okay to install the package. You only need to do this once. However, much like with `EBImage`, you need to run `library(tripack)` to load the library and access its functionality.

In order to compute ψ_6 for a series of data you need two functions from TrackPack. These are `psi6loc()` and `psi6series()`. `psi6loc()` computes ψ_6 for a single frame and `psi6series()` runs through all the frames in your data calling `psi6loc()`. To compute ψ_6 use the command

```
p6 <- psi6series(tr)
```

The output of this will be a matrix with six columns and one row for each particle in each frame. The columns contain, in order, x position, y position, real part of ψ_6, imaginary part of ψ_6, modulus of ψ_6 and the frame number.

C. Mean-squared displacement

To compute the mean squared displacement you need the function `msd()` and the function `shift()`. `msd()` requires tracked data as it needs to know which particle is which. By default the mean squared displacement is calculated for all time intervals up to the maximum time interval in your data, *i.e.* the total number of frames. To run:

```
meansq <- msd(tr)
```

The output has seven columns which are: time interval in frames, mean δx, mean δy, mean δx^2, mean δy^2, mean δr^2 (the MSD) and the number of displacements averaged over at each time interval.

D. Self intermediate scattering function, $F_s(q,t)$

The self part of the intermediate scattering function is computing using `isf()`, which also relies on `shift()`. It is called in much the same way as `msd()`:

```
fsqt <- isf(tr,length=15)
```

where we also have to specify a lengthscale over which to compute the ISF. This lengthscale is usually the particle diameter in pixels. The output of `isf()` has 5 columns: time interval in frames, real part of ISF, imaginary part of ISF, modulus of ISF and the number of displacements averaged over at each time interval.

[1] J. C. Crocker and D. G. Grier, Journal of Colloid and Interface Science **179**, 298 (1996).
[2] E. R. Weeks and J. C. Crocker, *Particle tracking using IDL*, URL `http://www.physics.emory.edu/faculty/weeks//idl/`.
[3] *The R project for statistical computing*, URL `http://www.r-project.org/`.
[4] *Rstudio*, URL `http://www.rstudio.com`.
[5] G. Pau, A. Oles, M. Smith, O. Sklyar, and W. Huber, *EBImage: Image processing toolbox for r*, r package version 4.4.0.
[6] *Bioconductor: Open source software for bioinformatics*, URL `http://www.bioconductor.org`.
[7] *ggplot2*, URL `http://ggplot2.org/`.
[8] I. Williams, E. C. Oğuz, P. Bartlett, H. Löwen, and C. P. Royall, Nature Commun. **4**, 2555 (2013).
[9] I. Williams, E. C. Oğuz, R. L. Jack, P. Bartlett, H. Löwen, and C. P. Royall, J. Chem. Phys. **140**, 104907 (2014).