**Modified version of MTrack2 plugin:**

**“MTrack2 kt” outputs the results in a single column for later analysis with the Ibidi chemotaxis tool.**

To install this modified MTrack2 plugin, download a free Java compiler program such as “JD-GUI”.

Copy the following text into the window and save as a .java file.

Drag the .java file into the ImageJ program folder in the “tools” or “plugins” folders.

Re-start the ImageJ program.

**import** ij.plugin.filter.PlugInFilter;

**import** java.awt.Color;

**import** java.util.\*;

**import** java.io.\*;

**import** java.lang.Float;

**import** ij.\*;

**import** ij.gui.\*;

**import** ij.io.\*;

**import** ij.process.\*;

**import** ij.plugin.filter.ParticleAnalyzer;

**import** ij.plugin.filter.Analyzer;

**import** ij.measure.\*;

/\*\*

Uses ImageJ's particle analyzer to track the movement of

multiple objects through a stack.

Based on the Object Tracker plugin filter by Wayne Rasband

Based on Multitracker, but should be quite a bit more intelligent

Nico Stuurman, Vale Lab, UCSF/HHMI, May,June 2003

Added single track column result format. KT, Magdeburg, June 2016

\*/

**public** **class** MTrack2\_kt **implements** PlugInFilter, Measurements {

ImagePlus imp;

**int** nParticles;

**float**[][] ssx;

**float**[][] ssy;

String directory,filename;

**static** **int** minSize = 1;

**static** **int** maxSize = 999999;

**static** **int** minTrackLength = 2;

**static** **boolean** bSaveResultsFile = **false**;

**static** TrackFormat saveTrackFormat = TrackFormat.SINGLE\_COLUMN\_SET;

**static** **boolean** bShowLabels = **false**;

**static** **boolean** bShowPositions = **false**;

**static** **boolean** bShowPaths = **false**;

**static** **boolean** bShowPathLengths = **false**;

**static** **float** maxVelocity = 10;

**static** **int** maxColumns=75;

**static** **boolean** skipDialogue = **false**;

**public** **class** particle {

**float** x;

**float** y;

**int** z;

**int** trackNr;

**boolean** inTrack=**false**;

**boolean** flag=**false**;

**public** **void** copy(particle source) {

**this**.x=source.x;

**this**.y=source.y;

**this**.z=source.z;

**this**.inTrack=source.inTrack;

**this**.flag=source.flag;

}

**public** **float** distance (particle p) {

**return** (**float**) Math.sqrt(sqr(**this**.x-p.x) + sqr(**this**.y-p.y));

}

}

/\*\*

\* Track format for results file: Multiple column sets, or single column set.

\*/

**public** **enum** TrackFormat {

/\*\*

\* Original result data format, with multiple column sets for multiple tracks.

\* - Grid: frame per row, track per column set.

\* - Each set of columns (x, y, flag) is a track. (The number of columns depends on the number of tracks.)

\* - Each row starts with a frame number followed by column sets for each track. (A frame is an image in the stack sequence).

\* - If a track has no point in a frame, then its (x, y, flag) cells are blank in that row.

\* - If many tracks are found in the stack, then the rowset of all frames is broken into multiple rowsets to fit in maxColumns.

\*/

MULTI\_COLUMN\_SETS,

/\*\*

\* Result data format with single column set for all tracks.

\* - The columns are (line in file, track number, slice in track, source frame in stack, x, y, flag).

\* - All slices of one track appear in frame order before the slices of the next track.

\* - If a track has no point in a frame, then no row appears with that track number and source frame in stack.

\*/

SINGLE\_COLUMN\_SET;

**static** String[] getAllLabels() {

**return** **new** String[]{MULTI\_COLUMN\_SETS.name(), SINGLE\_COLUMN\_SET.name()};

}

}

**public** **int** setup(String arg, ImagePlus imp) {

**this**.imp = imp;

**if** (IJ.versionLessThan("1.17y"))

**return** DONE;

**else**

**return** DOES\_8G+NO\_CHANGES;

}

**public** **static** **void** setProperty (String arg1, String arg2) {

**if** (arg1.equals("minSize"))

minSize = Integer.parseInt(arg2);

**else** **if** (arg1.equals("maxSize"))

maxSize = Integer.parseInt(arg2);

**else** **if** (arg1.equals("minTrackLength"))

minTrackLength = Integer.parseInt(arg2);

**else** **if** (arg1.equals("maxVelocity"))

maxVelocity = Float.valueOf(arg2).floatValue();

**else** **if** (arg1.equals("saveResultsFile"))

bSaveResultsFile = Boolean.valueOf(arg2);

**else** **if** (arg1.equals("saveTrackFormat"))

saveTrackFormat = TrackFormat.valueOf(arg2);

**else** **if** (arg1.equals("showPathLengths"))

bShowPathLengths = Boolean.valueOf(arg2);

**else** **if** (arg1.equals("showLabels"))

bShowLabels = Boolean.valueOf(arg2);

**else** **if** (arg1.equals("showPositions"))

bShowPositions = Boolean.valueOf(arg2);

**else** **if** (arg1.equals("showPaths"))

bShowPaths = Boolean.valueOf(arg2);

**else** **if** (arg1.equals("skipDialogue"))

skipDialogue = Boolean.valueOf(arg2);

}

**public** **void** run(ImageProcessor ip) {

**if** (!skipDialogue) {

GenericDialog gd = **new** GenericDialog("Object Tracker");

gd.addNumericField("Minimum Object Size (pixels): ", minSize, 0);

gd.addNumericField("Maximum Object Size (pixels): ", maxSize, 0);

gd.addNumericField("Maximum\_ Velocity:", maxVelocity, 0);

gd.addNumericField("Minimum\_ track length (frames)", minTrackLength, 0);

gd.addChoice("Result format", TrackFormat.getAllLabels(), saveTrackFormat.name());

gd.addCheckbox("Save Results File", bSaveResultsFile);

gd.addCheckbox("Display Path Lengths", bShowPathLengths);

gd.addCheckbox("Show Labels", bShowLabels);

gd.addCheckbox("Show Positions", bShowPositions);

gd.addCheckbox("Show Paths", bShowPaths);

gd.showDialog();

**if** (gd.wasCanceled())

**return**;

minSize = (**int**)gd.getNextNumber();

maxSize = (**int**)gd.getNextNumber();

maxVelocity = (**float**)gd.getNextNumber();

minTrackLength = (**int**)gd.getNextNumber();

bSaveResultsFile = gd.getNextBoolean();

**this**.saveTrackFormat = TrackFormat.valueOf(gd.getNextChoice());

bShowPathLengths = gd.getNextBoolean();

bShowLabels = gd.getNextBoolean();

bShowPositions = gd.getNextBoolean();

bShowPaths = gd.getNextBoolean();

**if** (bShowPositions)

bShowLabels =**true**;

}

**if** (bSaveResultsFile) {

SaveDialog sd=**new** SaveDialog("Save Track Results","trackresults",".txt");

directory=sd.getDirectory();

filename=sd.getFileName();

}

track(imp, minSize, maxSize, maxVelocity, directory, filename);

}

**public** **void** track(ImagePlus imp, **int** minSize, **int** maxSize, **float** maxVelocity, String directory, String filename) {

**int** nFrames = imp.getStackSize();

**if** (nFrames<2) {

IJ.showMessage("Tracker", "Stack required");

**return**;

}

ImageStack stack = imp.getStack();

**int** options = 0; // set all PA options false

**int** measurements = CENTROID;

// Initialize results table

ResultsTable rt = **new** ResultsTable();

rt.reset();

// create storage for particle positions

List[] theParticles = **new** ArrayList[nFrames];

// record particle positions for each frame in an ArrayList

**for** (**int** iFrame=1; iFrame<=nFrames; iFrame++) {

theParticles[iFrame-1]=**new** ArrayList();

rt.reset();

ParticleAnalyzer pa = **new** ParticleAnalyzer(options, measurements, rt, minSize, maxSize);

pa.analyze(imp, stack.getProcessor(iFrame));

**float**[] sxRes = rt.getColumn(ResultsTable.X\_CENTROID);

**float**[] syRes = rt.getColumn(ResultsTable.Y\_CENTROID);

**if** (sxRes==**null**)

**continue**;

**for** (**int** iPart=0; iPart<sxRes.length; iPart++) {

particle aParticle = **new** particle();

aParticle.x=sxRes[iPart];

aParticle.y=syRes[iPart];

aParticle.z=iFrame-1;

theParticles[iFrame-1].add(aParticle);

}

IJ.showProgress((**double**)iFrame/nFrames);

}

// now assemble tracks out of the particle lists

// Also record to which track a particle belongs in ArrayLists

List theTracks = **new** ArrayList();

**int** trackCount=0;

**for** (**int** i=0; i<=(nFrames-1); i++) {

IJ.showProgress((**double**)i/nFrames);

**for** (ListIterator j=theParticles[i].listIterator();j.hasNext();) {

particle aParticle=(particle) j.next();

**if** (!aParticle.inTrack) {

// This must be the beginning of a new track

List aTrack = **new** ArrayList();

trackCount++;

aParticle.inTrack=**true**;

aParticle.trackNr=trackCount;

aTrack.add(aParticle);

// search in next frames for more particles to be added to track

**boolean** searchOn=**true**;

particle oldParticle=**new** particle();

particle tmpParticle=**new** particle();

oldParticle.copy(aParticle);

**for** (**int** iF=i+1; iF<=(nFrames-1);iF++) {

**boolean** foundOne=**false**;

particle newParticle=**new** particle();

**for** (ListIterator jF=theParticles[iF].listIterator();jF.hasNext() && searchOn;) {

particle testParticle =(particle) jF.next();

**float** distance = testParticle.distance(oldParticle);

// record a particle when it is within the search radius, and when it had not yet been claimed by another track

**if** ( (distance < maxVelocity) && !testParticle.inTrack) {

// if we had not found a particle before, it is easy

**if** (!foundOne) {

tmpParticle=testParticle;

testParticle.inTrack=**true**;

testParticle.trackNr=trackCount;

newParticle.copy(testParticle);

foundOne=**true**;

}

**else** {

// if we had one before, we'll take this one if it is closer. In any case, flag these particles

testParticle.flag=**true**;

**if** (distance < newParticle.distance(oldParticle)) {

testParticle.inTrack=**true**;

testParticle.trackNr=trackCount;

newParticle.copy(testParticle);

tmpParticle.inTrack=**false**;

tmpParticle.trackNr=0;

tmpParticle=testParticle;

}

**else** {

newParticle.flag=**true**;

}

}

}

**else** **if** (distance < maxVelocity) {

// this particle is already in another track but could have been part of this one

// We have a number of choices here:

// 1. Sort out to which track this particle really belongs (but how?)

// 2. Stop this track

// 3. Stop this track, and also delete the remainder of the other one

// 4. Stop this track and flag this particle:

testParticle.flag=**true**;

}

}

**if** (foundOne)

aTrack.add(newParticle);

**else**

searchOn=**false**;

oldParticle.copy(newParticle);

}

theTracks.add(aTrack);

}

}

}

**boolean** writefile=**false**;

**if** (filename != **null**) {

File outputfile=**new** File (directory,filename);

**if** (!outputfile.canWrite()) {

**try** {

outputfile.createNewFile();

}

**catch** (IOException e) {

IJ.showMessage ("Error", "Could not create "+directory+filename);

}

}

**if** (outputfile.canWrite())

writefile=**true**;

**else**

IJ.showMessage ("Error", "Could not write to " + directory + filename);

}

// display the table with particle positions

// first when we only write to the screen

**if** (!writefile) {

**try** {

StringWriter sw = **new** StringWriter();

BufferedWriter bw = **new** BufferedWriter(sw);

**if** (saveTrackFormat == TrackFormat.MULTI\_COLUMN\_SETS) {

writeTracksMultiColumns(bw, theTracks, theParticles);

} **else** **if** (saveTrackFormat == TrackFormat.SINGLE\_COLUMN\_SET) {

writeTracksSingleColumn(bw, theTracks, theParticles);

}

**if** (bShowPathLengths) {

writePathLengths(bw, theTracks);

}

bw.close();

sw.close();

BufferedReader br = **new** BufferedReader(**new** StringReader(sw.toString()));

IJ.setColumnHeadings(br.readLine());

**for** (String line; (line = br.readLine()) != **null**; ) {

IJ.write(line);

}

} **catch** (IOException never) {

**throw** **new** AssertionError(never);

}

}

// and now when we write to file

**if** (writefile) {

**try** {

File outputfile=**new** File (directory,filename);

BufferedWriter dos= **new** BufferedWriter (**new** FileWriter (outputfile));

**if** (saveTrackFormat == TrackFormat.MULTI\_COLUMN\_SETS) {

writeTracksMultiColumns(dos, theTracks, theParticles);

} **else** **if** (saveTrackFormat == TrackFormat.SINGLE\_COLUMN\_SET) {

writeTracksSingleColumn(dos, theTracks, theParticles);

}

**if** (bShowPathLengths) {

writePathLengths(dos, theTracks);

}

dos.close();

}

**catch** (IOException e) {

**if** (filename != **null**)

IJ.error ("An error occurred writing the file. \n \n " + e);

}

}

// Now do the fancy stuff when requested:

// makes a new stack with objects labeled with track nr

// optionally also displays centroid position

**if** (bShowLabels) {

String strPart;

ImageStack newstack = imp.createEmptyStack();

**int** xHeight=newstack.getHeight();

**int** yWidth=newstack.getWidth();

**for** (**int** i=0; i<=(nFrames-1); i++) {

**int** iFrame=i+1;

String strLine = "" + i;

ImageProcessor ip = stack.getProcessor(iFrame);

newstack.addSlice(stack.getSliceLabel(iFrame),ip.crop());

ImageProcessor nip = newstack.getProcessor(iFrame);

nip.setColor(Color.black);

// hack to only show tracks longerthan minTrackLength

**int** trackNr=0;

**int** displayTrackNr=0;

**for** (ListIterator iT=theTracks.listIterator(); iT.hasNext();) {

trackNr++;

List bTrack=(ArrayList) iT.next();

**if** (bTrack.size() >= minTrackLength) {

displayTrackNr++;

**for** (ListIterator k=theParticles[i].listIterator();k.hasNext();) {

particle aParticle=(particle) k.next();

**if** (aParticle.trackNr==trackNr) {

strPart=""+displayTrackNr;

**if** (bShowPositions) {

strPart+="="+(**int**)aParticle.x+","+(**int**)aParticle.y;

}

// we could do someboundary testing here to place the labels better when we are close to the edge

nip.moveTo((**int**)aParticle.x+5,doOffset((**int**)aParticle.y,yWidth,5) );

//nip.moveTo(doOffset((int)aParticle.x,xHeight,5),doOffset((int)aParticle.y,yWidth,5) );

nip.drawString(strPart);

}

}

}

}

IJ.showProgress((**double**)iFrame/nFrames);

}

ImagePlus nimp = **new** ImagePlus(imp.getTitle() + " labels",newstack);

nimp.show();

imp.show();

nimp.updateAndDraw();

}

// 'map' of tracks

**if** (bShowPaths) {

**if** (imp.getCalibration().scaled()) {

IJ.showMessage("MultiTracker", "Cannot display paths if image is spatially calibrated");

**return**;

}

ImageProcessor ip = **new** ByteProcessor(imp.getWidth(), imp.getHeight());

ip.setColor(Color.white);

ip.fill();

trackCount=0;

**int** color;

**for** (ListIterator iT=theTracks.listIterator();iT.hasNext();) {

trackCount++;

List bTrack=(ArrayList) iT.next();

**if** (bTrack.size() >= minTrackLength) {

ListIterator jT=bTrack.listIterator();

particle oldParticle=(particle) jT.next();

**for** (;jT.hasNext();) {

particle newParticle=(particle) jT.next();

color =Math.min(trackCount+1,254);

ip.setValue(color);

ip.moveTo((**int**)oldParticle.x, (**int**)oldParticle.y);

ip.lineTo((**int**)newParticle.x, (**int**)newParticle.y);

oldParticle=newParticle;

}

}

}

**new** ImagePlus("Paths", ip).show();

}

}

// Utility functions

**double** sqr(**double** n) {**return** n\*n;}

**int** doOffset (**int** center, **int** maxSize, **int** displacement) {

**if** ((center - displacement) < 2\*displacement) {

**return** (center + 4\*displacement);

}

**else** {

**return** (center - displacement);

}

}

**double** s2d(String s) {

Double d;

**try** {d = **new** Double(s);}

**catch** (NumberFormatException e) {d = **null**;}

**if** (d!=**null**)

**return**(d.doubleValue());

**else**

**return**(0.0);

}

/\*\*

\* Write track data using a row for each frame

\* and a set of columns (x, y, flag) for each track.

\* Each row starts with its frame number.

\* Cells are blank for a frame where a track has no point.

\* Wrap rowsets at maxColumns, with a new title and heading

\* for each subset of rows.

\*/

**void** writeTracksMultiColumns(BufferedWriter dos, List<?> theTracks, List<?>[] theParticles) **throws** IOException {

// Create the column headings based on the number of tracks

// with length greater than minTrackLength

// since the number of tracks can be larger than can be accomodated by Excell, we deliver the tracks in chunks of maxColumns

// As a side-effect, this makes the code quite complicated

String strHeadings = "Frame";

**int** trackCount=1;

**for** (ListIterator iT=theTracks.listIterator(); iT.hasNext();) {

List bTrack=(ArrayList) iT.next();

**if** (bTrack.size() >= minTrackLength) {

**if** (trackCount <= maxColumns)

strHeadings += "\tX" + trackCount + "\tY" + trackCount +"\tFlag" + trackCount;

trackCount++;

}

}

**int** repeat=(**int**) ( (trackCount/maxColumns) );

**float** reTest = (**float**) trackCount/ (**float**) maxColumns;

**if** (reTest > repeat)

repeat++;

dos.write(strHeadings);

dos.newLine();

**for** (**int** j=1; j<=repeat;j++) {

**int** to=j\*maxColumns;

**if** (to > trackCount-1)

to=trackCount-1;

String stLine="Tracks " + ((j-1)\*maxColumns+1) +" to " +to;

dos.write(stLine);

dos.newLine();

**for** (**int** i=0; i<theParticles.length; i++) {

String strLine = "" + (i+1);

**int** trackNr=0;

**int** listTrackNr=0;

**for** (ListIterator iT=theTracks.listIterator(); iT.hasNext();) {

trackNr++;

List bTrack=(ArrayList) iT.next();

**boolean** particleFound=**false**;

**if** (bTrack.size() >= minTrackLength) {

listTrackNr++;

**if** ( (listTrackNr>((j-1)\*maxColumns)) && (listTrackNr<=(j\*maxColumns))) {

**for** (ListIterator k=theParticles[i].listIterator();k.hasNext() && !particleFound;) {

particle aParticle=(particle) k.next();

**if** (aParticle.trackNr==trackNr) {

particleFound=**true**;

String flag;

**if** (aParticle.flag)

flag="\*";

**else**

flag=" ";

strLine+="\t" + aParticle.x + "\t" + aParticle.y + "\t" + flag;

}

}

**if** (!particleFound)

strLine+="\t \t \t ";

}

}

}

dos.write(strLine);

dos.newLine();

}

}

}

/\*\*

\* Write data for every track, using a row for each point in the track.

\* The row columns are: line in file, track number, slice in track, source frame in stack, x, y, flag.

\* Omit track rows with no data, where no point was found for the track in a frame of the image stack.

\*/

**void** writeTracksSingleColumn(BufferedWriter writer, List<?> theTracks, List<?>[] eachFramesParticles) **throws** IOException {

writer.write("line"+'\t'+"TrackNr"+'\t'+"TrSlice"+'\t'+"Frame"+'\t'+"X"+'\t'+"Y"+'\t'+"Flag");

writer.newLine();

**int** lineCount = 0, outputTrackCount = 0;

**for** (**int** trackIndex = 0; trackIndex < theTracks.size(); trackIndex++) {

List<?> track = (List<?>) theTracks.get(trackIndex);

**int** trackNr = trackIndex + 1;

**int** trackSliceNr = 0;

**if** (track.size() >= **this**.minTrackLength) {

++outputTrackCount;

**for** (**int** frameIndex = 0; frameIndex < eachFramesParticles.length; frameIndex++) {

List<?> frameParticles = (List<?>) eachFramesParticles[frameIndex];

**int** frameNr = frameIndex + 1;

particle trackFrameParticle = **null**;

**for** (Iterator<?> particleIter = frameParticles.iterator(); particleIter.hasNext(); ) {

particle aFrameParticle = (particle) particleIter.next();

**if** (aFrameParticle.trackNr == trackNr) {

trackFrameParticle = aFrameParticle;

**break**;

}

}

**if** (trackFrameParticle != **null**) {

particle p = trackFrameParticle;

++lineCount;

++trackSliceNr;

writer.write(lineCount+"\t"+outputTrackCount+"\t"+trackSliceNr+"\t"+frameNr+"\t");

writer.write(p.x+"\t"+p.y+"\t"+(p.flag ? "\*" : ""));

writer.newLine();

}

}

}

}

}

/\*\*

\* Summarize each track with length (sum of step distances), distance between first and last position, and number of frames.

\*/

**void** writePathLengths(BufferedWriter dos, List<?> theTracks) **throws** IOException {

**int** trackCount = theTracks.size();

**double**[] lengths = **new** **double**[trackCount];

**double**[] distances = **new** **double**[trackCount];

**int**[] frames = **new** **int**[trackCount];

**double** x1, y1, x2, y2;

**int** trackNr=0;

**int** displayTrackNr=0;

**for** (ListIterator iT=theTracks.listIterator(); iT.hasNext();) {

trackNr++;

List bTrack=(ArrayList) iT.next();

**if** (bTrack.size() >= minTrackLength) {

displayTrackNr++;

ListIterator jT=bTrack.listIterator();

particle oldParticle=(particle) jT.next();

particle firstParticle=**new** particle();

firstParticle.copy(oldParticle);

frames[displayTrackNr-1]=bTrack.size();

**for** (;jT.hasNext();) {

particle newParticle=(particle) jT.next();

lengths[displayTrackNr-1]+=Math.sqrt(sqr(oldParticle.x-newParticle.x)+sqr(oldParticle.y-newParticle.y));

oldParticle=newParticle;

}

distances[displayTrackNr-1]=Math.sqrt(sqr(oldParticle.x-firstParticle.x)+sqr(oldParticle.y-firstParticle.y));

}

}

dos.newLine();

dos.write("Track \tLength\tDistance traveled\tNr of Frames");

dos.newLine();

**for** (**int** i=0; i<displayTrackNr; i++) {

String str = "" + (i+1) + "\t" + (**float**)lengths[i] + "\t" + (**float**)distances[i] + "\t" + (**int**)frames[i];

dos.write(str);

dos.newLine();

}

}

}