Image Processing and Particle Tracking for Biological Applications

Semester thesis at ICoS¹

Ingo Oppermann
ingo@student.ethz.ch
Rechnergestützte Wissenschaften

July 24, 2003

Advisors: Ivo F. Sbalzarini
Prof. Petros Koumoutsakos

¹Institute of Computational Science, Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zürich. http://www.icos.ethz.ch/
Contents

1 Introduction 1

2 Foundations 1

3 Particle Tracker Software 2
  3.1 Programming language 2
  3.2 Server 2
  3.3 Client 3
  3.4 Protocol 4
  3.5 Supported file types 4

4 Algorithms 6
  4.1 Image restoration 6
  4.2 Trajectory linking 7

5 Benchmarks 10
  5.1 Direct evaluation vs. FFT 10
  5.2 MATLAB vs. particle tracker 12
  5.3 Real world data 14

6 Conclusion 16

7 Outlook 16

8 References 17

A Users manual 18
  A.1 Server 18
    A.1.1 Compiling 18
    A.1.2 Running 18
    A.1.3 Stopping 19
  A.2 Client 19
    A.2.1 Compiling 19
    A.2.2 Running 19
    A.2.3 Input file 20
    A.2.4 Result file 21

B Technical 22
  B.1 Naming conventions 22
  B.2 Libraries 23
    B.2.1 libtiff 23
    B.2.2 mpeglib 23
    B.2.3 FFT 23
  B.3 Particle tracker API 23
  B.4 Server 26
  B.5 Communication protocol 28
1 Introduction

In this semester project the task was to develop and to implement a stand-alone image processing core for virus particle tracking from micrographs and movies.

On the basis of an existing MATLAB\textsuperscript{2} code for virus particle tracking I developed a stand-alone software that run under both the Linux/UNIX environment and the Windows\textsuperscript{3} environment consisting of a computational core (server) that run in the background and a lightweight client application to communicate with the server. The interface to the outside consists of a simple communication protocol based on TCP/IP.

The client will upload a series of images or movies of moving particles to the server where the particles will be detected and their paths tracked over time. Additionally the client can specify some parameter based on the quality of the images to gain better results. The resulting trajectories will be returned to the client in a human-readable form for further processing by hand or by other software.

The server has to read the uploaded images and applies filters to increase the visual quality. After that for each image the possible particles will be detected. In the final step the particles of each image will be linked to one of the next image resulting in a set of trajectories for this sequence of images. In every step it is possible to tweak some parameter to gain better results.

2 Foundations

The whole project is based on “A MATLAB toolbox for virus particle tracking” [1]. The corresponding technical report [2] explains the usage and the used algorithms in this toolbox in great detail. Further more I was provided with additional papers ([3] through [7]) to get better knowledge and in-depth information about the topics covering image restoration, particle detection and particle tracking. Some of these papers gave me hints for possible points of improvements for the implementation of the currently used algorithms.

On the other hand I looked on the internet for further information on image restoration and found some very interesting concepts using mathematical morphology [8] or optimized cubic filter [9] for an improved noise reduction.

\textsuperscript{2}MATLAB is a trademark of The Mathworks, Inc.
\textsuperscript{3}Windows is a trademark of Microsoft, Inc.
3 Particle Tracker Software

The particle tracker software is based on the client-server model, i.e. the server is running on some machine and is waiting for connecting clients. The communication between server and client is controlled by a simple packet-based, human-readable protocol. This protocol is directly build upon TCP/IP. This makes it possible to access the server from any remote host that provides access to the internet.

3.1 Programming language

The server application as well as the client application are written in C. Compiler specific extensions are avoided. The code is designed to be highly portable (the built-in libraries excluded) thus it will run on a great variety of operating systems with a standard C compiler. The choice fell on C because it will generate fast code and as a secondary goal of this semester project was to outperform the MATLAB code. Please refer to the benchmark section. Another advantage of C is that there are a lot of free and open-source libraries available for all kind of purposes.

3.2 Server

The server application consists essentially of two parts. The communication part and the particle tracker part. The particle tracker provides an API that is integrated by the communication part. This API provides a set of functions for setting the tracking parameters, for submitting a list of images to be analyzed and for getting the results (i.e. the found trajectories) and so on. The exact API specification will be described in appendix B.

The communication part connects the particle tracker API with the client and it is designed in a way such that several clients can access the particle tracker API at the same time. This is realized by multi-threading under the Windows environment and multi-processing under the Linux/UNIX environment. The pros and cons of multi-threading and multi-processing respectively will be discussed in appendix B.

As soon as a client tries to connect to the server it will check if the maximum allowed number of clients is reached. If so the client will be disconnected with an appropriate error message. The maximum allowed clients per server can be set at startup time of the server. Same for the port the server has to listen on. If a client gets accepted a new process will be forked (under Linux/UNIX) or a new thread will be created (under Windows) and the server is again ready for accepting new connections.

At this point the server is ready to receive requests by the client. The client can set some parameter, upload files, execute the image processing and request the results of the image processing. Without any images uploaded the particle tracker will refuse to start the calculation. At least two images have to be uploaded. After setting the parameters and uploading the images to be analyzed one can send an execute calculation request and the server will initialize the particle tracker. If the initialization was successful the server executes the particle tracking procedure.
The particle tracker now checks if the uploaded files are of an valid format. For a description of valid file formats please refer to 'Supported file types'. Additional to the file the client uploads what color channel of the current file should be taken for the image processing. This color channel will be saved in a temporary file for later use. Temporary files are used to minimize the amount of needed memory in contrast to keep them all in memory.

Now every image pass the particle detection facility where it runs through four steps. First the image needs to be restored, i.e. filtering long-wavelength modulations of the background intensity and noise from the digital camera and the frame grabber. Second the particle positions are estimated. After the estimation comes the position refinement and the non-particle discrimination.

After all particles in every frame have been found they are passed to the trajectory linking. There it will be calculated what particle in the next frame is, according to a metric, the one in the current frame with the highest possibility. The metric is based on the current particle position and the intensity momenta. The linking is not restricted to the next frame. The so-called linkrange can be stretched to any value greater than one. Though the possibility of finding the right particle will go to zero for frames that are far away. This feature is considered to compensate gaps in the final trajectory of the particle.

If all trajectories are successfully linked the server sends an 'OK' as an answer of the execute calculation request the client sent before. The server is now ready to accept the request for sending the results. As soon as the results are sent the server is ready for a new job. Even in between these steps of setting the parameter, uploading images and so on one is able to reset all made changes and to start from the beginning. Alternatively the client can disconnect and reconnect on any time it wants.

3.3 Client

The client application provided with the server application is a lightweight implementation of the communication protocol and is considered as a starting point for a more sophisticated implementation. The client is a console application, thus it doesn’t provide a graphical user interface. Because the communication protocol is based on TCP/IP a client can be written or ported to any platform or programming language that support TCP/IP communication. The behaviour of the current client can be controlled by a simple input file.

A first step is to create an input file containing all needed information for the client such that it can connect to the server and finally receive the results. Inside the input file one can define or tweak a bunch of settings depending on the data to be analyzed. Available setting are the radius of the particles, the cutoff radius for the non-particle discrimination, the percentile that defines how sensitive the particle detection should be, the maximum displacement of one particle between two subsequent frames and the linkrange that defines how many frames the trajectory linking should take in consideration. Further more it is necessary to define what files should be uploaded for image processing and what color channel should be taken. Finally the IP or the hostname and the port of the server and the name of the result file have to be provided. All these parameter and settings are described in appendix A in more detail.

As soon as the input file is ready the client application can be executed with the name of the input file as parameter. The client then reads the file
and extracts the information. If there are syntax errors the client produces an appropriate error message and stops execution. If files provided in the file list can’t be found the client generates an error message but proceeds execution. If everything is in order the client connects to the server and starts setting the parameter and uploading the files. After that it executes the image processing on the server and waits until the server has finished. Then the result file will be downloaded and the client disconnects from the server.

Inside the result file are again information about the parameter set and other information gathered by the server. The rest of this file are the found trajectories themselves. They can be used for further processing by another software tool or even by hand. All information in the result file is in human-readable form. A description of a result file is provided in appendix A.

3.4 Protocol

The protocol for the client-server communication is packet-based. Every packet has a fixed length header and a limited length body, thus every packet has a maximum length. The minimum length is the length of the header. Every header contains a code to identify what the packet is for and the length of the packet body. Every packet a client sends to the server will be acknowledged by one or more return packets. The server is only reacting on request by the client and not vice versa. The first request sent by the client is actually the attempt to connect to the server.

Since every packet is of a limited size it is necessary to split up a file for upload. The upload is divided in a sequence of request. First the client sends a upload request that the server can accept or deny. If the upload is accepted it is initialized. Now the client can begin to send actual data packets, so the server knows where they belong to. Each data packet will be acknowledged. In between there can be packets for setting a parameter for example. This doesn’t influence the upload at all. If the upload is done the client has to send a upload finish packet in order to upload more files.

Another special handling concerns the result file. After the client requests the result file the server begins to send result file packets and finishes the transaction with an result file finish packet. Then the client knows that the complete results are downloaded.

The last packet the client sends is the request for disconnect. Even this request will be acknowledged. Then the server closes the connection to the client and frees the used system resources.

The exact protocol specifications can be found in appendix B.

3.5 Supported file types

Only TIFF images and MPEG-1 movie streams are currently supported. The support of TIFF images is provided by the freely available and open-source libtiff [10], but not fully. There are a some sub-formats of the TIFF image format that are not integrated in the libtiff (e.g. TIFF images that are compressed with JPEG). Occasionally the user has to convert the images to another TIFF sub-format to use the particle tracker.

The support of MPEG video streams is provided by the free and open-source mpeglib [11], but there are some restrictions too. Only MPEG-1 video streams
are considered valid. MPEG streams with audio will be rejected as unsupported. Actually MPEG-2 support was planned but unfortunately there isn’t any free and open-source decoder API for this format.

To make the software not relying on pre-installed components on the system it will be running on, the libraries are built-in. Therefore it is not needed to install any of these libraries separately. Even if they are already available on the system only the built-in ones will be used.
4 Algorithms

The algorithms used for particle detection and trajectory linking in this particle tracker software are based on the ones used in the corresponding MATLAB toolbox [2]. Because it is not possible to transfer MATLAB code directly into C code there is room for other methods doing the same. Thus improvements in speed or efficiency can take place along with extensions to the existing algorithms or even completely new things. In the following subsections I will discuss the changes, extensions and new additions that are currently implemented in the particle tracker software.

4.1 Image restoration

The uploaded micrographs or movies have imperfections on it that have to be removed (i.e. filtered out) in order to be able to do a better particle detection. There are two kinds of noise that have to be taken out. First the long-wavelength modulations of the background intensity and second the discretization noise from the digital camera and the frame grabber. These two noises can be removed by a combination of a boxcar average filter and a Gaussian filter both over a square region of extend $2w+1$ where $w$ is a user-defined parameter called kernel radius. This filter is a convolution of the original image with a kernel of support $2w+1$:

$$A_f(x, y) = \sum_{i=-w}^{w} \sum_{j=-w}^{w} A(x + i, y + j)K(i, j)$$

where $A$ is the original image with the coordinates $x = 0, \ldots, X$ and $y = 0, \ldots, Y$. $K$ is the filter kernel and $A_f$ is the resulting filtered image. The exact appearance of $K$ is described in [2]. The convolution can be done either with the direct evaluation of the convolution sum or by a Fast Fourier transform to do the convolution in frequency space where it is a simple multiplication.

The drawback of doing the convolution in frequency space is the FFT. Both the image and the kernel have to be transformed into frequency space and element-wise complex multiplied (real values are transformed into complex numbers). The result then needs to be re-transformed back into spatial space for further processing. There the FFT produces quite a large overhead in computation time and in memory. Fortunately the kernel is not image dependent, except the image size and kernel radius, but these values stays the same for one sequence of images. Thus the FFT of the kernel can be computed once in advance and reused for every image. Therefore only the image needs to the transformed and re-transformed.

To do the convolution the image and the kernel need to be of the same dimension, i.e. $(X + w) \times (Y + w)$. The original kernel is placed in the top-left corner of the resized kernel and the leftover space is filled with zeros. The same for the image but there the leftover space is padded by the last column and the last row of the image. After the re-transform of the convoluted image it needs to be extracted to have again the same dimension as the original image (the filtered image is now in the lower right corner). If the leftover space is padded with zeros as in the kernel, there will be artifacts at the boundary (of thickness $w$) in the resulting image. These artifacts can lead to wrong results in particle detection and position refinement.
The FFT itself is done by applying twice a 1D-FFT on the 2-dimensional data. The used FFT algorithm is by R. C. Singleton [12] ported to C by Javier Soley [13] that I found here [14]. It uses the split radix method. Unfortunately this algorithm only works correct if the length of the data to transform is a power of 2. Therefore the already enlarged image and kernel need to be resized again to an appropriate size (the method for avoiding artifacts in the resulting image stays the same though). But this gives another impact on computation time and memory usage because more data has to be crunched than needed.

For small kernel radii it is more efficient to do the convolution by directly evaluating the sum. There are only \((2w + 1)^2\) multiplications per resulting pixel to be done and this can be heavily optimized for a given radius. For the particle tracker server I wrote optimized functions for the kernel radii from 1 to 5. Further more I wrote an optimized generic function that can be applied to any radius but it is not so fast and will be outperformed by the FFT for certain radii.

The choice of the best method (FFT or direct evaluation) according to the image dimensions and the kernel radius is individually done by the server at computation time. The underlying model is based on the benchmark results in the 'Benchmarks' section.

4.2 Trajectory linking

The described algorithm for trajectory linking in [2] tries to find a correlation between the particles in the current frame and the ones in the following frame. This is done by computing a relation matrix filled with values that represent the costs for associating a particle in the current frame with a particle in the next frame. The cost is a metric based on the spatial coordinates of two particles and their intensity momenta. The maximum allowed cost is given by the square of the user-defined parameter called displacement. This value defines how far a particle may move at the most from one frame to the next to get identified as the one in the current frame. This relation matrix will be optimized to find the best (in the sense of the costs) correlations.

Since this algorithm only takes the current and the next frame into consideration it can happen that for a particle that disappears for one frame (e.g. due to a variation in the intensity and a too sensitive particle detection or non-particle
discrimination) and reappears in the next frame the trajectory will be broken. Thus in the result there will be two trajectories for one particle.

To avoid these broken trajectories I extended the trajectory linking algorithm such that it can take the particles of \( n \) future frames into consideration where \( n \) is an integer between 1 and \( \infty \) (in principle). This is done by sequentially applying the linking algorithm to the \( n \) following frames. The current frame stays always the same. Thus if one particle is missing in one frame it can be identified in the following frame. Though the linking will get more inaccurate for each frame in the future because the maximum allowed displacement will be increased too. Therefore it will have more possible candidates and maybe the wrong will be taken. It is also assumed that the currently invisible particles moves in the same manner as before, i.e. if the particle reappears somewhere it couldn’t be expected, the linking will fail. The value of \( n \) can be set by the user and is called \textit{linkrange}. If \( n \) is bigger than the actual available number of future frames it will be set to that value.

With this extension, not all gaps get closed, though. Consider two particles on trajectories that are crossing such that in the point where they cross, only one particle will be detected. With a linkrange of 1 there will be three trajectories found. A long and two shorter ones, because the trajectory of the second particle will be ‘cut through’ by the first one. A linkrange of 2 or more won’t necessarily close the gap because the second particle of frame \( i \) could be linked to the first particle in frame \( i + n \). But this position is already occupied, thus the trajectory of the second particle will stop in frame \( i \) and a new one will be started in frame \( i + 2 \).

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{broken_closed_trajectories.png}
\caption{Possible trajectories for a bigger linkrange. (Image intensities are inverted for printing purposes.)}
\end{figure}

Another possibility is that the two particles switch their trajectories. There, the trajectory of the second particle can be closed but with the one of the first particle. The resulting two trajectories will maybe not reflect the desired results.

The algorithm is more accurate for trajectories that are not crossing. A gap of 3 or more frames can be closed without any problems. This will be demonstrated in figure 3. It shows a series of 4 frames with two particles on it. The upper particle is only present in the first and the last image. This would require a minimum linkrange of 3 to complete the trajectory for this particle. The second particle is present in every frame and moves around. The rightmost image shows the final trajectories found by the particle tracker. As one
can see, the gap of two frames has been closed and the particles have been linked correctly.

But there, the question will show up if it makes sense closing a gap of more frames, since it could really be the starting point of a newly appeared particle’s trajectory. It strongly depends on the data to be analyzed if tweaking the linkrange will result in better and correct trajectories.
5 Benchmarks

In this section the performance of the particle tracker software and the MATLAB toolbox will be compared. Additionally I will take a look at the memory usage of these two programs and how this does influence the system. Further more the two methods of computing the convolution for the image restoration will be compared in dependence of the image size.

System:
CPU: AMD Athlon XP 1800+
Memory: 256MB
Hardisk: IBM 18GB SCSI
OS: Microsoft Windows 2000 Professional

The version of the “MATLAB toolbox for virus particle tracking” is the one dated on February 12, 2003. The used MATLAB version is Release 12.1. To measure the time for the computation I inserted to = clock before the images will be read in and zeit = etime(clock, t0) after the call of link_trajectories(...) into the file tracker.m. For the particle tracker server of this project I used the version that is on the enclosed CD. It was compiled with Microsoft Visual Studio 6 and with release settings. The time measurement is done by the server and is returned to the client after the computation.

All input files and obtained data can be found in the Benchmark directory on the CD.

5.1 Direct evaluation vs. FFT

The described formula for the convolution for image restoration can be computed either with the direct evaluation of the sum or by a Fast Fourier transform. For small kernel radii it is faster to directly evaluate the sum and for larger ones it is faster to do it by FFT. But where is the transition between the two methods? The following values are only taken from the particle tracking server to find a connection between the kernel radius and the used computation method and the image size. The time values are obtained by computing 30 frames each. To 'disable' the trajectory linking the parameter percentile has been set to a very low value. This avoids that the particle tracker will find any particles. To force the particle tracker server to use the wished method for a given kernel radius the code had to be slightly modified.

As one can see the Fast Fourier transform is always slower than the direct evaluation. Due to limitations by the used FFT algorithm it is only possible to calculate the transform of sizes of the next greater power of 2. Therefore for an image of the dimensions of 256x256 pixels the FFT has to be evaluated over a region of 512x512 pixels. The same applies to 512x512 pixels with the actual computation region of 1024x1024 pixels.

The following table 2 will show a slightly different situation. The times for the direct sum are almost equal to the one in the previous table for an image size of 512x512 pixels. However the measured times for the FFT are more equal to the ones for an image of size 256x256 pixels. This is due the fact that the FFT computes for both the smaller image (256x256) and the larger one (500x500) over the same region of 512x512 pixels. The differences are given by the the
<table>
<thead>
<tr>
<th>Radius</th>
<th>Direct sum 256x256</th>
<th>Direct sum 512x512</th>
<th>FFT 256x256</th>
<th>FFT 512x512</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>13</td>
<td>52</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>13</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
<td>14</td>
<td>53</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>13</td>
<td>54</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>12</td>
<td>14</td>
<td>57</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>18</td>
<td>15</td>
<td>61</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>23</td>
<td>16</td>
<td>63</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>28</td>
<td>16</td>
<td>65</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>33</td>
<td>17</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>39</td>
<td>18</td>
<td>73</td>
</tr>
</tbody>
</table>

Table 1: Comparison of the computation time for calculating the convolution sum either by evaluating the sum directly or by Fast Fourier transform.

dilation operation that depends on the image size. For a kernel radius of 10 both methods are equally fast and for bigger radii the FFT gets faster.

<table>
<thead>
<tr>
<th>Radius</th>
<th>500x500</th>
<th>500x500</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>21</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>25</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>30</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>35</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>40</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>47</td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Comparison of the computation times of images of a size slightly smaller than a power of 2.

In table 3 it is shown how the computation time depends on the image size for a given kernel radius of 12. There one can see that somewhere between the image size of 400x400 and 450x450 pixels the FFT gets faster than the direct evaluation. For choosing the approximately fastest method for a given kernel radius and image size one has to take these two information into account.

When the kernel radius gets bigger the point where the FFT gets faster than the direct evaluation is moving towards the lower image size. E.g. for a kernel radius of 20 and an image size of 256x256 pixels the direct sum does take slightly more time with 34 seconds than using the FFT with 33 seconds. Therefore if the kernel radius is big enough the FFT has to be preferred.

These gathered information are used to improve the choice of the best method in the particle tracker server. The rules are the following:

11
<table>
<thead>
<tr>
<th>Size</th>
<th>Direct sum</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>300</td>
<td>17</td>
<td>23</td>
</tr>
<tr>
<td>350</td>
<td>23</td>
<td>27</td>
</tr>
<tr>
<td>400</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>450</td>
<td>38</td>
<td>36</td>
</tr>
<tr>
<td>500</td>
<td>47</td>
<td>42</td>
</tr>
</tbody>
</table>

Table 3: Comparison of the computation time of different image sizes for a given kernel radius of 12.

If the kernel radius is smaller than 12 the direct sum method is taken, no matter what the image dimension is.

If the kernel radius is between 12 and 20 the chosen method depends on the width of the image. The middle between the lower and upper power of 2 of the width is taken and if the width is smaller than this value the direct sum method is taken and the FFT method otherwise. E.g. if the width is 400 pixels the FFT method will be taken because it is bigger than $384 = 256 + (256/2)$.

If the kernel radius is equal to or bigger than 20 the FFT method is always taken no matter what the image dimension is.

These rules are a very simplified model of the measured values but they are accurate enough to provide the convolution with an appropriate computing method. In the second rule only the width of the image is taken into consideration because I assume that most of the images that will be analyzed has more or less square dimensions.

5.2 MATLAB vs. particle tracker

A secondary goal for the implementation of the MATLAB toolbox into a stand-alone software was to outperform the MATLAB code. This little benchmark will show if this goal is reached. Besides the time measurements I take a look at the memory footprint of the MATLAB application and of the particle tracker software. Additionally the accuracy of the resulting trajectories will be compared. For the accuracy test, a set of synthetic data is needed where the particle positions are known. This test data has been generated by the trackertest.m file. Two kinds of test sequences are used. First a set of 40 frames with a horizontal movement of the particles. The images are 512x512 pixel of size and have 24 particles on it. Second a set of 40 frames with 10 randomly moving particles. This set is only used for an accuracy test because the computing time and the memory consumption will not differ from the first set. For sequences longer than 40 frames, MATLAB ran out of memory and began to swap to the hard disk. This would make the comparison unfair. For both sets the kernel radius is set to 6 and the maximum displacement to 20. The values for the upper $p$-th percentile used are different for the MATLAB code (0.1) and for the particle tracker software (0.15) due to different implementation of the histogram function. But both values are chosen such that all particles will be detected and the correct trajectories will be found. The tasks are to read in the images, to filter the images, to detect the particles and to refine the positions and finally to link the trajectories.
Linear trajectories
The MATLAB code did the job in 79 seconds and the particle tracker server did the same in 25 seconds. The server is around three times faster than MATLAB. The memory consumption during the computation of the 40 images is for the particle tracker around 8MB whereas MATLAB uses 200MB and more because it keeps all images in memory. To get information how accurate the computation of the both programs is, I calculated the mean of the standard deviation of the found particle positions in the 40 frames. For the standard deviation the reference value is not the mean but the position that should have been found. The gathered results are summarized in the following table.

<table>
<thead>
<tr>
<th>Program</th>
<th>Time</th>
<th>Memory consumption</th>
<th>Mean std. dev. x-axis</th>
<th>Mean std. dev. y-axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td>79 secs</td>
<td>&gt; 200MB</td>
<td>0.145</td>
<td>0.148</td>
</tr>
<tr>
<td>Particle tracker</td>
<td>25 secs</td>
<td>around 8MB</td>
<td>0.170</td>
<td>0.174</td>
</tr>
</tbody>
</table>

Table 4: Summarized results of the comparison of the MATLAB toolbox and the particle tracker software on linear moving particles.

The particle tracker is faster and uses less memory, but the results aren’t as accurate as computed with MATLAB. Maybe the regularity of the positioning of the particles plays a role, since every particle is placed exactly on a pixel in the image and MATLAB can take advantage of this. The results of the random trajectories will show another situation.

Random trajectories
Here, no time or memory measurements are taken, only the accuracy of the found trajectories will be discussed. Both programs found all particles and linked them correctly. Besides the comparison of the mean standard deviation of the trajectories, I was also interested how good the found trajectories of the two programs fit on each other. For this I computed the square root of the mean square distances of each trajectory. There the mean deviation in x-direction is
Table 5: Summarized results of the comparison of the MATLAB toolbox and the particle tracker software on randomly moving particles.

<table>
<thead>
<tr>
<th>Program</th>
<th>Mean std. dev. x-direction</th>
<th>Mean std. dev. y-direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td>0.222</td>
<td>0.205</td>
</tr>
<tr>
<td>Particle tracker</td>
<td>0.185</td>
<td>0.173</td>
</tr>
</tbody>
</table>

0.144 and 0.132 in y-direction. As one can see, the found trajectories differ quite a lot from each other. But which one is closer to the rated values?

Table 5 shows the mean standard deviation of the positions where they should be in the two spatial coordinates of the trajectories. Compared to the values of the previous test set, the trajectories computed by the particle tracker server now fits better than the ones by MATLAB. That’s not because the values are more accurate than before, but the ones computed by MATLAB are more inaccurate. The particle tracker server stays in the same range of accuracy. Since the situation where all particles move in parallel on an exact grid does not correspond very well with reality, the latter results should be considered more meaningful.

5.3 Real world data

For testing the particle tracker on a real world example I used a series of 330 micrographs showing three stationary beads with a diameter of 24 nanometer each. One bead is almost in the middle of the image, the second one on the bottom and the third one on the left border. Each frame is 200x200 pixel of size or covers a region of 56x56 nanometer. The kernel radius is set to 4 pixel or 11.36 nm respectively.

![Sample image of the real world test data](image.png)

Figure 5: Sample image of the real world test data. (Image intensities are inverted for printing purposes.)

The calculation was done in 14 seconds and generated an output file containing the trajectories of the three beads. The length of these trajectories is over all 330 frames. Additionally there are other shorter trajectories of moving particles that appear and disappear. The trajectories of the three stationary beads are used to calculate the accuracy of the particle detection algorithm. As these particles don’t move over the period of 330 frames they are an ideal object to compute the standard deviation in the x-coordinate and the y-coordinate.

The results can be found in the following table. The x-coordinate is top-down
from 0 to 199 and the y-coordinate is left-right from 0 to 199. The deviation of
e.g. the x-coordinate of particle 1 of 0.125 pixel corresponds to a deviation of
0.35 nm compared to the diameter of about 24 nm.

The shorter trajectories of the detected moving particles can be used to
extract the diffusion constant. This is done via the mean square displacement
and it’s relation to the diffusion constant

$$< r(t)^2 > \approx 4Dt$$

where $D$ is the diffusion constant and $t$ the diffusion time.

<table>
<thead>
<tr>
<th>Particle</th>
<th>x-coordinate Mean</th>
<th>x-coordinate Std. dev.</th>
<th>y-coordinate Mean</th>
<th>y-coordinate Std. dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99.183</td>
<td>0.125</td>
<td>61.182</td>
<td>0.113</td>
</tr>
<tr>
<td>2</td>
<td>112.274</td>
<td>0.214</td>
<td>1.344</td>
<td>0.117</td>
</tr>
<tr>
<td>3</td>
<td>189.844</td>
<td>0.091</td>
<td>32.557</td>
<td>0.105</td>
</tr>
</tbody>
</table>

Table 6: The mean and standard deviation in the two spatial coordinates for
the three stationary beads.

For calculating the diffusion constant I took 4 trajectories of moving beads
and plotted their mean square displacement over time. The time is given in
frames that have been taken with a frequency of 31 Hz. Then I averaged the
mean square displacements and did a least square fit and got the solid line shown
in the left image of figure 6. The slope of this line is $32.12 \text{ pixel}^2/\text{frame} = 4D$,
or $D = 8.03 \text{ pixel}^2/\text{frame}$. This will give a diffusion constant of $208.87 \text{ nm}^2/\text{s}$
with $1 \text{ pixel}^2 = 8.07 \text{ nm}^2$ and $1 \text{ frame} = \frac{1}{31} \text{s}$. The right-hand image shows the
averaged logarithm of the mean square displacements plotted over the logarithm
of the time (frame numbers). For these points I did a least square fit too and
got the solid line with a slope of 1.48. Since the slope should be near 1, the
consistency of the data is not ensured. With a slope of 1, the covered range
on the ordinate axis gives the logarithm of $4D$. The actual data covers a range
of 3.99 and it follows a value of $13.52 \text{ pixel}^2/\text{frame}$ for $D$. That is a diffusion
constant of $3382.29 \text{ nm}^2/\text{s}$. These differences in the values are due to the small
ensemble and to the inaccuracy in particle detection.
6 Conclusion

The software developed in this semester project is a stand-alone application based on “A MATLAB toolbox for virus particle tracking”. It implements the same algorithms but can offer more speed, less memory consumption and extensions to the algorithms. This software is designed to be a client-server application, where the server runs on a host and waits for connecting clients that will upload a series of images to be analyzed. Thus many people can use the particle tracker anywhere they are and they don’t need to learn how to use MATLAB. But it is still necessary to know what the available tweakable parameter do in order to get good results.

The work on this project gave me a great insight into topics like image restoration, particle detection, trajectory linking and networking. During the time of this work I gained a lot skills in programming in general and in image processing by porting the MATLAB code to C. I had a lot of fun even though it was plenty of work.

7 Outlook

This first implementation of a particle tracker as a stand-alone client-server application is surely not perfect an will not cover all needs. The used algorithms for image restoration, particle detection and trajectory linking still offer points for more improvements, not only in speed but in more accuracy too. Or one can think of replacing them with completely other methods that will do an even better job.

Another point is supporting more image formats even though TIFF is the most common file format used for micrographs. Since the MPEG-1 support is considered as experimental it should be improved or completely removed and replaced by a good MPEG-2 library. A major point there is to get rid of global variables for a better usability under a multi-threading environment and a better code design.

To improve speed in image restoration (this consumes the most time of the whole calculation) it would be useful to replace the currently used FFT algorithm by the splendid FFTW library [15] (or only the needed parts of it).

Maybe there are flaws in the code design that can be fixed, or the communication protocol needs a review to support more features that are needed to implement more powerful client applications. The software seems to very stable but it has never been tested under high loads like hundreds clients at the same time that are uploading, calculating and downloading.
8 References

A Users manual

This section describes how to compile and to use the particle tracker server as well as the client. Both applications don’t have any graphical interface, thus they run on any console. Both the server and client have been successfully compiled and tested on the following systems:

- Microsoft Windows 2000 Professional
- Linux Debian/GNU
- FreeBSD 4.1

Additionally to the source code, the pre-compiled executables of the server and the client for the three mentioned systems are provided. They can be found in the Binaries-directory on the CD. These binaries are only for the i386 microprocessor architecture and are compiled with double precision.

A.1 Server

A.1.1 Compiling

Before compiling three compile-time parameter can be set. The first parameter defines if the tracker should use single or double precision variables during calculation. The other two defines the default values for the server port and the maximum allowed connections. They can be found in config.h in the Server-directory.

Windows:
In order to compile the server application under Windows you need Microsoft Visual Studio 6 or later and your system has to support the Winsock2 API. Copy the contents of the Server-directory on the CD including all subdirectories into a directory on your harddisk. Change to that directory and double-click on server.dsw. Then select ’Build → Build server.exe’ or simply press F7. The compiled server.exe can be found in the bin-directory. If you want to change the active configuration from Debug to Release select ’Build → Set Active Configuration ...’ and choose the appropriate setting.

Linux/UNIX:
Copy the contents of the Server-directory on the CD including all subdirectories into a directory on your harddisk. Change to that directory and type in make and press ENTER. The server will be compiled into the bin-directory.

A.1.2 Running

In order to run the server you can use the appropriate pre-compiled binary for your system or you can compile your own executable as described in the previous section. Once you have a suitable server executable you can let it run. Open a console (Windows: DOS console) and change to the directory where the server executable is. Then just type in server and press ENTER. Something like this will show up:

Starting server ...
Using default*:
   Listening on port 1138
Max. number of connections: 128

* To specify your own values for the port and the maximum number of connections please start the server as follows:
  
  server [port max_connections]

Server is running on laptop:1138 (192.168.1.33:1138)

The values for the port and the maximum number of connections may vary if you compiled the server with different defaults. To specify your own values for the port and the maximum number of connections you can type in e.g. 'server 1234 42' to make the server listen on port 1234 and allow maximally 42 connected clients. Under Linux/UNIX you can append an ampersand at the end of the command line to make the server running in the background.

Once the server runs, it tries to create a temp directory at the location where it is, thus the server needs to have writing rights on this directory. Otherwise it will quit with an appropriate error message.

A.1.3 Stopping

You can stop the server by pressing CTRL-c in the console window the server is running in. Under Linux/UNIX you can also use the kill command with the PID of the server. Be sure to kill the parent process in order to kill the whole server, otherwise only one child process will be killed and the parent is still accepting connections.

A.2 Client

A.2.1 Compiling

Windows:

In order to compile the client application under Windows you need Microsoft Visual Studio 6 or later and your system has to support the Winsock2 API. Copy the contents of the client-directory on the CD including all subdirectories into a directory on your harddisk. Change to that directory and double-click on client.dsw. Then select 'Build' → 'Build client.exe' or simply press F7. The compiled client.exe can be found in the bin-directory. If you want to change the active configuration from Debug to Release select 'Build' → 'Set Active Configuration ...' and choose the appropriate setting.

Linux/UNIX:

Copy the contents of the client-directory on the CD including all subdirectories into a directory on your harddisk. Change to that directory and type in make and press ENTER. The client will be compiled into the bin-directory.

A.2.2 Running

In order to run the client you can use the appropriate pre-compiled binary for your system or you can compile your own executable as described in the previous section. Once you have a suitable client executable you can let it run. Open a console (Windows: DOS console) and change to the directory where the client executable is and type in client [input file] and press ENTER, where [input
file] has to be replaced by the path to your input file. How to create an input file is described in the following subsection. The client reads the provided input file, connects to the specified server, uploads the specified files, downloads the results to the specified location and finally quits.

A.2.3 Input file

This section will explain the various parameter that can be set in the input file for the client application. Every parameter will be set by a command followed by an equality sign and the value (e.g. host = localhost). The command itself is case-insensitive. Only one command per line is allowed and a line shouldn't be longer than 1024 characters. One can add comments into the input file by adding a # (sharp) in front of the part that shall be commented out. There it doesn't matter if it is on the beginning of the line or somewhere in between. Below is a table of all allowed commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>The hostname. This can be either the name or the IP of the host the server is running on</td>
<td>localhost</td>
</tr>
<tr>
<td>port</td>
<td>The port the server is listening on</td>
<td>1138</td>
</tr>
<tr>
<td>radius</td>
<td>Particle radius. This value should be greater than the particle radius and smaller than the interparticle spacing. Unit is pixel.</td>
<td>3</td>
</tr>
<tr>
<td>cutoff</td>
<td>Cutoff radius. Probability cutoff for non-particle discrimination.</td>
<td>3.0</td>
</tr>
<tr>
<td>percentile</td>
<td>Intensity percentile in which to accept particles. Unit is percent.</td>
<td>0.1</td>
</tr>
<tr>
<td>displacement</td>
<td>Maximum allowed particle displacement between two subsequent frames.</td>
<td>10.0</td>
</tr>
<tr>
<td>linkrange</td>
<td>Maximum number of frames to use for trajectory linking.</td>
<td>1</td>
</tr>
<tr>
<td>results</td>
<td>The path of the file where to save the resulting trajectories.</td>
<td>results.txt</td>
</tr>
<tr>
<td>color</td>
<td>For each image to be read in, there can be specified what color channel should be taken for particle tracking. Once a color channel is set it will be used for all following images as long as no other color channel is set. Valid color channels are: r (red), g (green), b (blue) and i (intensity). If an invalid color channel is set it will be ignored.</td>
<td>i</td>
</tr>
<tr>
<td>file</td>
<td>To specify the files to upload. This command has to be used for each file that shall be uploaded.</td>
<td>-</td>
</tr>
<tr>
<td>type</td>
<td>Defines the type of the following files. Once a type is set it will be used for all following files as long as no other type is set. Valid types are: TIFF for TIFF images and MPEG for MPEG movie streams.</td>
<td>TIFF</td>
</tr>
<tr>
<td>list</td>
<td>A for-loop like command to specify a numbered sequence.</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>verbose</td>
<td>Verbose output in the result file. 0 for verbose mode off, 1 for verbose mode on.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: A list of all allowed commands in an input file

The commands file and list need a special explanation. If you have a numbered series of 20 frames with one file per frame you would write something like this in the input file:

```plaintext
file = frame1.tif
file = frame2.tif
file = frame3.tif
...
file = frame20.tif
```

Because this can get very annoying for more files there is the list command that makes life way easier. Using list, the above will look like this:

```plaintext
list = 1, 20, 1
file = frame%d.tif
```

where the value of list is defined as [start], [stop], [stepsize]. The place-holder %d will be replaced by the appropriate number (if the filename contains a percent sign one has to write %%). It is following the definition for formatted output of the function printf(...) in C. Only the file command right after list will be considered as a part of the construct. The list and the file command can be mixed like this:

```plaintext
file = frame1.tif
list = 2, 9, 1
    file = frame%d.tif
file = frame10.tif
file = frame11.tif
list = 12, 20, 1
    file = frame%d.tif
```

An example of an input file can be found in the Client-directory on the CD.

### A.2.4 Result file

The result file contains the found trajectories. They are ordered in three columns. The first column indicates the frame number. The numbering starts at zero. The second and the third column contain the x-coordinate and the y-coordinate of the particle where the x-axis is top-down and the y-axis is left-right on the image. Each trajectory is separated by a blank line. A trajectory covers two frames in minimum. Particles that appear only in one frame (i.e. trajectories of length 1) are not listed. Besides that, there are some general information on the top such as the time the analysis took place, the used parameter and information about the analyzed images.
B Technical

This section explains the used strategies, structures and API’s of the particle tracker software. This will not be a complete description of the source code. A description for every function is given in the source code explaining what it does, what parameter it needs, what the output is and what other functions it uses. Besides that there are more comments on what happens inside the functions. For in-depth information please refer to the source code.

The source code of the particle tracker application consists of the following files that can be found in the Server-directory on the enclosed CD.

- `config.h` Configuration file for compile-time parameter.
- `convolve.c` The different kernel radius dependend subroutines for the convolution.
- `dilate.c` The different kernel radius dependend subroutines for the gray scale dilation.
- `filelist.c` A little helper API for the server for handling filelists.
- `import.c` Routines for reading images and movies.
- `messages.c` The messages used by the server.
- `server.c` The server code. Contains the `main()` function.
- `sing.c` Subroutines for calculating the FFT.
- `tracker.c` The particle tracker API.
- `mpeg/` This directory contains the files of the mpeglib.
- `tiff/` This directory contains the files of the libtiff.

To every of the mentioned .c file exists a corresponding .h file that contains the needed type definitions and function prototypes.

The source code of the client consists only of the file `client.c` that can be found in the Client-directory on the enclosed CD.

B.1 Naming conventions

To distinct clearly the different parts of the code they have an appropriate prefix. All constants and function have the following prefixes:

- `SERVER_` For the server part
- `FL_` For the filelist API
- `PT_` For the particle tracker API
- `IMPORT_` For the image and movie import
- `SM_` For the server messages
- `CLIENT_` For the client part

Variables and the defined structures do not follow these conventions. The functions of the used libraries have their own naming conventions and are leaved untouched.
B.2 Libraries

B.2.1 libtiff

The used library for importing TIFF images is the libtiff v3.5.7 [10]. The support for JPEG compressed images is not given because this needed to include the LIBjpeg so I disabled it in the tiff/tiffconf.h file. All other common TIFF formats are supported.

To coop better with the particle tracker software the two global variables of the libtiff, `_TIFFErrorHandler` and `_TIFFWarningHandler`, are set to NULL. This avoids any unwished output of the library routines.

B.2.2 mjpeg

The used library for importing MPEG streams is the mjpeglib v1.3.1 [11]. It only supports MPEG-1 video streams, thus streams with audio in it will be rejected as invalid.

This library is not perfect. It makes heavily use of global variables. This cause some problems in an multi-threading environment like Windows. Threads do share the same global variables what makes a semaphore necessary to control the access to the decoder such that only one client at a time can use it. As soon as the import routines get called they look whether the semaphore is set or not. If the semaphore is already set, the routine will wait until it will be released.

Another problem is that in the code are `exit()` calls that will shut down the whole server in an multi-threading environment. In an multi-processing environment it will only quit the actual process that called the `exit()` function. I hopefully eliminated all of these calls but the MPEG support should be considered as experimental. Thus it is more safe to run the server under Linux/UNIX or any other operating system that support the `fork()` system call.

B.2.3 FFT

For calculating the FFT I took an split radix algorithm [12]. It is a port of an FORTRAN code to C thus it is not very structured, but it works. Unfortunately it uses global variables too, but this problem is solved in the same manner as for the mjpeglib.

B.3 Particle tracker API

The particle tracker itself is constructed as an API. This allows it to use the functionality in other programs other than the server application. The main data structure for the particle tracker is the `PTSequence`. It is defined as follows:

```c
typedef struct PTSequence
{
    int radius;  /* Kernel radius */
    real cutoff; /* Cutoff radius */
    real percentile; /*Percentile */
    real displacement; /* Maximum displacement */
    int verbose; /* verbose mode */
    real lambda; /* Correlation lenght */
};
```
/* Image sequence parameter */
int width, height;
real min, max;
int number_of_frames;
real *frame;
PTFrame *framelist; /* List of all frames */

int linkrange;
ParticleList *particlelist; /* List of all particles */

/* Function pointer to the appropriate routine */
real *((*Convolve)(struct PTSequence *pts, real *filtered, \]
real *input));
real *((*Dilate)(struct PTSequence *pts, real *dilated, \]
real *input));

real *kernel; /* Holds the kernel */
int kernel_width;

int *mask; /* Holds the dilation mask */

real *filtered; /* Holds the filtered image */
real *dilated; /* Holds the dilated image */

/* Structure for the FFT calculation */
FFT fft;

/* Result handling */
int result;
char result_file[256];
FILE *result_fp;

/* Error handling */
int error;
char error_msg[128];
}

PTSequence;

The variable real is defined either as float or as double according how it
was defined at compile-time. The definitions of the three new types PTFrame,
ParticleList and FFT can be found in tracker.h. They are helper types to
make the structure more clear.

The particle tracker API functions that uses the above structure are as follows:
PTSequence *PT_CreateSequence(void)
This functions creates a PTSequence structure. It allocates memory for it and
provides all variables within the structure with defaults and finally returns a
pointer to it. This function should be called before all others.

int PT_SetParameter(PTSequence *pts, const char *params)
This function allows to set the user-defined parameter of the PTSequence. The
variable *params* contains a string with a code defining what parameter to set and the value the parameter should have, i.e. if *params* contains "1 4" the kernel radius will be set to 4. The codes are defined like this:

```c
#define PT_PARAM_KERNELRADIUS 1
#define PT_PARAM_CUTOFF 2
#define PT_PARAM_PERCENTILE 3
#define PT_PARAM_DISPLACEMENT 4
#define PT_PARAM_LINKRANGE 5
#define PT_PARAM_VERBOSE 6
```

The return value is either 1 for success and 0 on failure and the error variable will be set with an appropriate error message.

```c
int PT_InitSequence(PTSequence *pts, FileList *filelist)
```
This function takes a pointer to a PTSequence structure and a pointer to a FileList structure. The FileList structure contains the list of images or movies that has to be analyzed by the particle tracker. First all images in the file list will be checked if they are valid and they are read in. While reading in all images (for MPEG streams every frame will be extracted) the user-defined color channel is saved to a temporary file, i.e. if one image is 200x200 pixel of size the temporary file will be 40000 bytes with every byte containing the intensity value of one pixel. These temporary files are used in order to save memory. After that the image parameter of the PTSequence can be set. According to all parameter the memory for holding the information that come up during the calculation will be reserved. In dependence of the image dimension and the kernel radius the FFT structure will be used or not and the function pointer for the convolution and the dilation will point to the appropriate routine. Anything that can be pre-computed is also done here such as the kernel, the dilation mask and the Fast Fourier transform of the kernel (if needed).

The return value is either 1 for success and 0 on failure and the error variable will be set with an appropriate error message. If something went wrong the PTSequence will be reset as if it were newly created by PT_CreateSequence().

```c
int PT_FindTrajectories(PTSequence *pts)
```
This function takes a pointer to a PTSequence as the only parameter and is the real core of the particle tracker. Here every image will be read in from the temporary file created by the PT_InitSequence(…) function and the particle detection and trajectory linking algorithms takes place as described in [2] and in the previous section 'Algorithms'. After a successful computation a result file has been written than can be requested by PT_GetResults(…).

The return value is either 1 for success and 0 on failure and the error variable will be set with an appropriate error message. If something went wrong the PTSequence will be reset as if it were newly created by PT_CreateSequence().

```c
void PT_DestroySequence(PTSequence *pts)
```
This function frees all used memory by the PTSequence pts and should be called to release all system resources such as deleting all created temporary files.

```c
void PT_ResetSequence(PTSequence *pts)
```
This function resets the PTSequence pts as if it were newly created by PT_CreateSequence().
char *PT_GetResults(PTSequence *pts)
This function returns a string containing the filename of the result file created by a successful run of PT_FindTrajectories(...), if available. Otherwise it will return a NULL pointer.

char *PT_GetError(PTSequence *pts)
This function returns a string containing an error message if available. Otherwise it will return a NULL pointer.

Some of the above functions use other PT_ and IMPORT_ functions but they are not intended to be used directly. A sample sequence of function calls is:

PT_CreateSequence
PT_InitSequence
PT_FindTrajectories
PT_GetResults
PT_DestroySequence

Once a PTSequence has been created it can be used more than one time. It is possible to do something like this:

PT_CreateSequence
do
    PT_SetParameter
    PT_InitSequence
    PT_FindTrajectories
    PT_GetResults
    PT_ResetSequence
while(...)  
PT_DestroySequence

This strategy is followed by the server and it depends on the client if one can take advantage of it.

B.4 Server
The server part of the particle tracker application uses the following structures:

typedef struct Client
{
    unsigned int clientsocket;

    FILE *fp;
    FileList *filelist;
    FileListEntry *current_file;

    int fileupload_ack;
    Packet *in, *out;

    PTSequence *pts;
} Client;
Since the communication with the client application uses UNIX TCP sockets only the socket of the client needs to be stored. Besides that a pointer to a PkSequence will be stored for the usage of the particle tracker API and a file list for handling the uploaded images or movies that will be passed to the tracker and two Packets for incoming and the outgoing data.

A FileList is defined through these structures:

typedef struct FileListEntry
{
    char path[256];
    int type;
    int special;
    struct FileListEntry *prev, *next;
} FileListEntry;

typedef struct FileList
{
    int number_of_files;
    FileListEntry *root;
} FileList;

This is a linked list with pointers to the previous and the next file. Access to the filelist is provided through this set of functions:

FileList *FL_CreateFileList(void);
void FL_DestroyFileList(FileList *filelist);
FileListEntry *FL_AddFile(FileList *filelist, int type, \!
    int special);
void FL_RemoveFile(FileList *filelist, FileListEntry *entry);
void FL_RemoveAllFiles(FileList *filelist);
int FL_FileExists(const char *filename);

Only *FL_AddFile(FileList *filelist, int type, int special) needs a more detailed description. All other functions should be self-explanatory.

*FL_AddFile(FileList *filelist, int type, int special)
This function has as parameter the filelist, the type and a special purpose variable. The file type, which can be any number, is for use in the particle tracker either 1 for TIFF images or 2 for MPEG streams. These two constants are defined in tracker.h:

#define PT_FILE_TIFF 1
#define PT_FILE_MPEG 2

The third parameter, special, is used to set what color channel of this file should be taken. These are 0 for intensity, 1 for red, 2 for green and 3 for blue. This function returns a FileListEntry that contains the path of the file the data can be written to, thus the file has to be opened separately, but it has been created. If something went wrong the return value would be a NULL pointer. The uploaded files will be stored in a temp directory in the same location where the server is.
To allow more than one client at a time the server is designed to be multi-threading under Windows and multi-processing under Linux/UNIX. Running the server in multi-processing mode (i.e. not under Windows) will be more preferable because it will cause less trouble with the usage of global variables and the MPEG library as explained in a previous subsection. This is because every process has its own globals that are not shared with other processes. Another advantage of multi-processing is the fact that if in one process something happens that causes the process to exit (segmentation fault or else) only this process will be killed. The main process is still accepting connections. Whereas under a multi-threading environment the whole server gets killed and no one can connect to the server. (The argument that splitting a new process creates much more overhead than creating a new thread each time a client connects to the server will not be applicable because there won’t be that much connections per second, or even per minute, than for a highly frequented webservice.)

What happens when a client has successfully connected to the server is explained in section 3. The communication between the server and the client and the explanation of the Packet structure will follow in the next subsection.

### B.5 Communication protocol

The communication between the client and the server uses a packet based protocol using TCP/IP for transportation. The data packets are handled with the following structure:

```c
typedef struct Packet {
    int type;  
    int len;   
    int curlen; 
    int buflen; 
    unsigned char *body;  
    unsigned char *buffer; 
} Packet;
```

Every packet consists of an header and a body. A packet has a maximum length of 4096 bytes of which 7 bytes are reserved for the packet header. The remaining 4089 bytes are the payload. Not every packet needs to be that big. The minimum packet length is 7 bytes, thus only the header. The header is split up in 3 + 4 bytes where the first 3 bytes defines the type of the packet. The type is a number between 100 and 999. The other 4 bytes contains the length of the body of the packet. The body itself can contain arbitrary data depending on what type the packet has. The protocol is defined to be human-readable, so a sample packet of type 200 and length 2 will look and transferred like this:

```
20000020K
```

As one can see the packet length field have to padded with zeroes. The type and the length are coded in the decimal system. If the body contains more than 2 characters they will be considered as the header and the body of the next packet.

The server knows only 9 packet types that can be received from client. These types are called *request types*. As a reply to these request types there are 26
response types of which are 7 acknowledge types, 3 special types and 16 error types that indicates that something went wrong. It is not necessary to implement all error types in the client. It would be sufficient to check whether the requested action has been acknowledged or not. Along with every error packet there can be an appropriate error description in the body.

All packet types and names are defined in server.h and have the prefix SERVER_CODE_. Here are all packet types with a short description except the request types. The request types will be explained later on the basis of these response types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
<td>Everything is fine</td>
</tr>
<tr>
<td>201</td>
<td>PARAMSET</td>
<td>Parameter is set</td>
</tr>
<tr>
<td>202</td>
<td>UPLOADINITACK</td>
<td>Upload acknowledged</td>
</tr>
<tr>
<td>203</td>
<td>UPLOADOK</td>
<td>Upload successful</td>
</tr>
<tr>
<td>204</td>
<td>ABORTEDUPLOAD</td>
<td>Upload aborted</td>
</tr>
<tr>
<td>205</td>
<td>FILESDELETED</td>
<td>Files successfully deleted</td>
</tr>
<tr>
<td>206</td>
<td>CALCDONE</td>
<td>Calculation done</td>
</tr>
<tr>
<td>300</td>
<td>RESULT</td>
<td>Result of the calculation</td>
</tr>
<tr>
<td>301</td>
<td>RESULTFINISH</td>
<td>All results has been sent</td>
</tr>
<tr>
<td>302</td>
<td>STATUS</td>
<td>current Status (unimplemented)</td>
</tr>
<tr>
<td>400</td>
<td>ERROR</td>
<td>Something went wrong</td>
</tr>
<tr>
<td>401</td>
<td>NOPENFILE</td>
<td>No open file (file upload)</td>
</tr>
<tr>
<td>402</td>
<td>CALCFAILED</td>
<td>Calculation failed</td>
</tr>
<tr>
<td>403</td>
<td>UPINPROGRESS</td>
<td>Upload in progress</td>
</tr>
<tr>
<td>404</td>
<td>NOUPINPROGRESS</td>
<td>No upload in progress</td>
</tr>
<tr>
<td>405</td>
<td>UPTYPENOTSUP</td>
<td>Upload type is not supported</td>
</tr>
<tr>
<td>406</td>
<td>EXTNOTSUP</td>
<td>Extension not supported</td>
</tr>
<tr>
<td>407</td>
<td>FILELISTERROR</td>
<td>Couldn’t create a new filist entry</td>
</tr>
<tr>
<td>408</td>
<td>FILEOPENFAILED</td>
<td>Couldn’t open the file</td>
</tr>
<tr>
<td>500</td>
<td>UNKNOWNERROR</td>
<td>Unknown error</td>
</tr>
<tr>
<td>501</td>
<td>UNKNOWNCODE</td>
<td>Unknown packet type</td>
</tr>
<tr>
<td>502</td>
<td>OUTOFMEMORY</td>
<td>Server is out of memory</td>
</tr>
<tr>
<td>503</td>
<td>THREADERROR</td>
<td>Server couldn’t create a new thread</td>
</tr>
<tr>
<td>504</td>
<td>TOO MANYCLIENTS</td>
<td>Clientlist is full</td>
</tr>
<tr>
<td>505</td>
<td>FORKERROR</td>
<td>Server couldn’t fork a new process</td>
</tr>
<tr>
<td>506</td>
<td>INVALIDLENGTH</td>
<td>Invalid length of a packet</td>
</tr>
</tbody>
</table>

Table 8: All possible response types by the server

All listed response type except the RESULT packet have an optional body content, e.g. they can only consists of the header. But the server sends always a body with an description of the response. In the following the 7 request type definitions will be explained.

100 SETPARAM
This request sets the parameter and the value given in the body of the packet. The whole body of this request will be passed to the PT_SetParam(…) function
unmodified. Please refer to the according section to see the description of the syntax.

The response type is either PARAMSET if the parameter has been successfully set, or ERROR if something went wrong. An appropriate error message is provided by the particle tracker.

101 UPLOADINIT uploadtype filetype color
This request asks for an file upload. Three parameter have to be provided in the packet body. Every parameter is separated by blanks.

uploadtype is always 1. filetype is either 1 for a TIFF file or 2 for a MPEG stream. color is 0 for intensity, 1 for red, 2 for green or 3 for blue. For the last two parameter see the definition of FLAddFile(...) in the latter subsection.

The response type is UPLOADINITACK if the client may begin to upload the data, or one of the following if something went wrong: FILEOPENFAILED returns if the server couldn't open the file. FILELISTERROR returns if something went wrong with the filelist. This is an server error. EXTNOTSUP returns if the filetype is unknown. UPTYPENOTSUP returns if the uploadtype is unknown. UPINPROGRESS will indicates that there is still an upload in progress, therefore it can't be requested a new one. The current upload has first to be finished or aborted.

102 UPLOADDATA
This request isn't really a request. It contains the contents of the uploading file in the body. The upload is binary safe and there are no special escape sequences needed.

The response type is OK if the data has been successfully written, or NOOPENFILE if no file is open to write the data to. This is a server error. NOUPINPROGRESS comes back if no upload has been initialized with UPLOADINIT.

103 UPLOADFINISH
This request tells the server that the current upload is finished and no more data will follow for this file. The contents of the body of this request will not be written to the file.

The response type is UPLOADOK if the file has been closed successfully, or NOUPINPROGRESS if no upload has been initialized with UPLOADINIT.

104 UPLOADABORT
This request tells the server to abort the current upload. The file will be closed and deleted.

The response type is ABORTEUPLOAD if the file has been successfully deleted, or NOUPINPROGRESS if no upload has been initialized with UPLOADINIT.

105 DELETEFILES
This request tells the server to delete all uploaded files so far. If an upload is still in progress it has first to be finished or aborted.

The response type is FILEDELETED if all files have been successfully deleted, or UPINPROGRESS if an upload is still in progress.

106 EXECCALC
This request asks the server to initialize the particle tracker with the uploaded files and to execute the FTFindTrajectories(...) function.
The response type is **CALCDONE** if the particle tracker successfully finished the calculation. The body contains the calculation time in seconds. **UPINPROGRESS** will be returned if an upload is still in progress. **CALCFAILED** will be returned if either the initialization or the calculation failed. The body contains an appropriate error message generated by the particle tracker.

107 **SENDRESULTS**
This request asks the server to send the result from the calculation. Because the result file can be bigger as a packet, it will be split up in more packets. Therefore the response type is **RESULT** as long there are more result packets available. If the whole result file has been sent, the server will send a **RESULTFINISH** to indicate the end of the result file transfer. This packet contains no data belonging to the result file. If the function call of `PT_SetResults(...)` failed, the response is **ERROR**. This can happen e.g. when there is no result file to download. Another possible response is **FILEOPENFAILED** when the server couldn’t open the result file.

108 **DISCONNECT**
This request tells the server to disconnect from the client. The response will be **OK** and the socket will be closed. After that the used system resources by the client will be released and the current thread / process will be terminated.