University of Cape Town

# Multiple Particle Tracking in PEPT using Voronoi Tessellations

Dylan Blakemore

*Supervisors:*
Dr Indresan Govender
Dr Andrew McBride

*Date:*
June 2016

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

**Abstract**

An algorithm is presented which makes use of three-dimensional Voronoi tessellations to track up to 20 tracers using a PET scanner. The lines of response generated by the PET scanner are discretized into sets of equidistant points, and these are used as the input seeds to the Voronoi tessellation. For each line of response, the point with the smallest Voronoi region is located; this point is assumed to be the origin of the corresponding line of response. Once these origin points have been determined, any outliers are removed, and the remaining points are clustered using the DBSCAN algorithm. The centroid of each cluster is classified as a tracer location.

Once the tracer locations are determined for each time frame in the experimental data set, a custom multiple target tracking algorithm is used to associate identical tracers from frame to frame. Since there are no physical properties to distinguish the tracers from one another, the tracking algorithm uses velocity and position to extrapolate the locations of existing tracers and match the next frame's tracers to the trajectories.

A series of experiments were conducted in order to test the robustness, accuracy and computational performance of the algorithm. A measure of robustness is the chance of track loss, which occurs when the algorithm fails to match a tracer location with its trajectory, and the track is terminated. The chance of track loss increases with the number of tracers; the acceleration of the tracers; the time interval between successive frames; and the proximity of tracers to each other. In the case of two tracers colliding, the two tracks merge for a short period of time, before separating and become distinguishable again. Track loss also occurs when a tracer leaves the field of view of the scanner; on return it is treated as a new object.

The accuracy of location of the algorithm was found to be slightly affected by tracer velocity, but is much more dependent on the distance between consecutive points on a line of response, and the number of lines of response used per time frame. A single tracer was located to within 1.26mm. This was compared to the widely accepted Birmingham algorithm, which located the same tracer to within 0.92mm. Precisions of between 1.5 and 2.0mm were easily achieved for multiple tracers.

The memory usage and processing time of the algorithm are dependent on the number of tracers used in the experiment. It was found that the processing time per frame for 20 tracers was about 15s, and the memory usage was 400MB. Because of the high processing times, the algorithm as is is not feasible for practical use. However, the location phase of the algorithm is massively parallel, so the code can be adapted to significantly increase the efficiency.

# Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This these/dissertation has been submitted to the Turnitin module (or equivalent similarity and originality checking software) and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

Signed: Dylan Murphy Blakemore

# Acknowledgements

I would like to thank all the people who helped and supported me throughout the research, with special thanks to Dr. Indresan Govender and Dr. Andrew McBride. Their advice and support helped me throughout the project, and they allowed me to explore my own ideas and take the project in my own direction. I'd also like to thank the staff at iThemba Labs, especially Mike van Heerden and Liu Cong, who took the time to prepare and help me conduct the necessary experiments.

Thanks also go out to the students and staff involved with the Center for Minerals Research. The people were amazing, and provided great opportunities to brainstorm and refine ideas. The multidisciplinary aspect of the group allowed for new ideas from multiple different fields, and helped ground the project and provide some much-needed practical context.

Finally, thanks to my family, who were supportive throughout my University career; and my friends and roommates, who provided distractions and companionship whenever the going got tough.

# Contents

**1 Introduction** **8**

**2 Background** **10**
2.1 Positron Emission Tomography . . . 10
2.1.1 History . . . 10
2.1.2 Data Acquisition and Image Reconstruction . . . 10
2.2 Positron Emission Particle Tracking . . . 13
2.2.1 Multiple Particle Tracking in PEPT . . . 14
2.3 Voronoi Diagrams . . . 15
2.3.1 Applications . . . 16
2.3.2 Variations . . . 19
2.3.3 Algorithms . . . 21
2.4 Cluster Analysis . . . 21
2.4.1 Clustering Algorithms . . . 23
2.5 Multiple Target Tracking . . . 24

**3 Description of Algorithm** **27**
3.1 Part 1 - Location . . . 27
3.1.1 Input and Output . . . 27
3.1.2 Description . . . 28
3.1.3 Parameters . . . 33
3.1.4 Parallelization . . . 33
3.1.5 Pseudocode . . . 35
3.2 Part 2 - Tracking . . . 36
3.2.1 Input and Output . . . 36
3.2.2 Description . . . 36
3.2.3 Parameters . . . 39
3.2.4 Pseudocode . . . 41

**4 Experimental Setup** **43**
4.1 Apparatus . . . 43
4.1.1 PET Scanner . . . 43
4.1.2 Tracers . . . 44
4.1.3 Custom Rig . . . 46
4.2 Descriptions of Experiments . . . 47
4.2.1 Circular Rotation . . . 48
4.2.2 Erratic Rotation . . . 48
4.2.3 Z-axis Rotation . . . 50
4.2.4 Varying Speed . . . 50
4.2.5 Static Deflection . . . 54
4.2.6 Tracer Impact . . . 56

**5 Analysis** **58**
5.1 Curve Fitting . . . 58
5.2 Track Loss . . . 59
5.2.1 Tracer Exiting FOV . . . 60
5.2.2 Scattering and Attenuation . . . 64
5.2.3 High Acceleration . . . 65
5.2.4 Tracer Proximity . . . 67
5.2.5 Large Tracer Count . . . 69
5.3 Location Accuracy . . . 69
5.3.1 Velocity . . . 70
5.3.2 Tracer Count . . . 72
5.3.3 Line Count . . . 72
5.3.4 Discretisation Separation Distance . . . 73
5.4 Erratic Rotation . . . 75
5.5 Computational Performance . . . 77
5.5.1 Runtime . . . 77
5.5.2 Memory Usage . . . 79

**6 Alternative Applications** **82**
6.1 Solid Deformation . . . 82

**7 Conclusions** **86**
7.1 Summary of Findings . . . 86
7.1.1 Accuracy . . . 86
7.1.2 Robustness . . . 87
7.1.3 Computational Performance . . . 87
7.2 Applications of Multiple-PEPT . . . 88
7.3 Future Developments . . . 89

**Appendices** **95**

**A Matlab Code** **96**
A.1 Location Algorithm . . . 96
A.1.1 Main Method . . . 96
A.1.2 Functions . . . 99
A.2 Tracking Algorithm . . . 107
A.2.1 Main Method . . . 107
A.2.2 Classes . . . 110
A.2.3 Functions . . . 113

**B Ethics Form** **116**

# List of Figures

2.1 Sinogram formation in PET. (A) Lines of response generated by a single point of radioactivity. (B) Angles and displacements plotted to form a vertical sine graph. (C) Collection of multiple overlapping sinograms. (D) Image reconstructed from sinograms. (Image from [7]). . . . 12
2.2 Position and velocity profiles of a tracer within a tumbling mill. The scanner used is the Siemens EXACT3D, and the Birmingham algorithm was used for processing. . . . 13
2.3 Example of a 2D Voronoi diagram using 20 seed points [26]. . . . 17
2.4 Voronoi tessellations appearing in nature . . . . 18
2.5 Voronoi diagram (green) with Delaunay triangulation (red) overlaid. . . . 18
2.6 Various weighted Voronoi tessellations using a Euclidean distance metric. The size of the coloured halos indicate the size of the weights. Images from [35] . . . . 20
2.7 Illustration of Fortune's sweep-line algorithm. The red line is the sweep line, the black line is the beach line, the blue lines are the Voronoi edges and the blue dots are the seed points. . . . 22
2.8 K-means clustering of two data sets. . . . 24
2.9 DBSCAN clustering of two data sets. . . . 25
2.10 Basic elements of a multiple target tracking algorithm. Figure from [46] . . . . 26

3.1 Plot showing the LOR's for two artificially generated tracers in 2D. . . . 28
3.2 Voronoi diagram for seeds generated by two artificial tracers in 2D, with the colour of each polygon indicating the mean vertex distance of that polygon. The image on the right shows the logarithm of the volumes to more clearly indicate the differences. . . . 29
3.3 Scatter plots of real data for a single frame in an experiment using two tracers. 100 lines were used per tracer, with a separation distance between points of 5mm, for a total of 31 130 discrete seed points . . . . 30
3.4 Distributions of local outlier factors for real data using two tracers. Far outliers (with LOF values greater than the mean plus twice the standard deviation) are excluded from this data. . . . 31
3.5 Histogram showing the Mean Vertex Distances of polyhedra surrounding the remaining seed points after LOF cleaning. . . . 32
3.6 Scatter plots illustrating the association matrix for arbitrary 2-dimensional points. . . . 37
3.7 Scatter plots illustrating the association matrix for arbitrary 2-dimensional points, with fewer new entries than existing tracks. . . . 38

4.1 CTI/Siemens EXACT3D model 966 PET scanner. . . . 43
4.2 Two tracers inside the plastic tubing. White insulation tape was used to provide extra padding for a more snug fit inside the rig. Ruler for scale. . . . 44
4.3 Polyethylene discs used in PEPT experiments. . . . 46
4.4 Assembly showing polyethylene discs assembled onto aluminium shaft. . . . 46
4.5 Custom rig used in multiple-PEPT experiments. . . . 47
4.6 Tracers attached to rubber bands to provide semi-uncontrolled motion. . . . 50

4.7 Rig to provide rotation in the z-axis, in order to test the effect of tracers exiting and entering the field of view. . . . 52
4.8 Plot of motor reading vs measured rotational velocity. . . . 54
4.9 Steel rod setup used in static deflection tests. . . . 55

5.1 Illustration of best-fit sine curves to track data generated by the VMPT algorithm. . . . 59
5.2 Trajectories of experiment in which two tracers exit and re-enter the field of view, while three tracers remain within the field of view. . . . 60
5.3 Results of Monte-Carlo simulations used to illustrate the effect of the $z$-position on the number of lines detectable by the scanner. Red lines do not intersect the top and bottom within the field of view, and therefore cannot be detected. . . . 62
5.4 Scatter plot showing the fraction of detectable lines of response in a Monte-Carlo simulation. . . . 63
5.5 Scatter plot and fitted curve showing the number of data points per cluster versus the Z-position of the located tracer. . . . 64
5.6 Plots showing 15 s snapshots of tracks generated during an experiment using eight tracers. In each plot, the time interval used during the location stage of the algorithm was changed. . . . 66
5.7 Trajectories of two tracers before and after impact. . . . 67
5.8 Extended trajectories of tracers impacting multiple times. . . . 68
5.9 Comparison between tracers in the same position with different total tracer counts. . . . 69
5.10 Scatter plot showing tracer velocity and respective RMSE values in experiments using four tracers at varying velocities. . . . 71
5.11 Scatter plot showing tracer velocity and respective accuracies in experiments using four tracers at varying velocities. . . . 71
5.12 Plots showing the effect of the number of tracers on the accuracy of location. . . . 72
5.13 Plots showing effect on the number of lines of response used by the location algorithm on the accuracy of location. . . . 73
5.14 Plots showing effect of the discretization spacing used by the location algorithm on the accuracy of location. . . . 74
5.15 Comparison between tracks generated by CTRACK and VMPT algorithms. . . . 75
5.16 Plots showing motion of 8 tracers, 4 of which experience controlled circular motion, and 4 of which experienced semi-chaotic motion. . . . 76
5.17 Graphs showing the time taken to process a single frame of data based on the number of seed points used during tessellation. . . . 78
5.18 Graphs showing the time taken to process a single frame of data based on the number of seed points used during tessellation. . . . 79
5.19 Memory used by Matlab while processing frames of data generated by experiments using six tracers. . . . 80
5.20 Maximum memory used when performing VMPT processing versus number of tracers used in experiments. . . . 81

6.1 Deflection over time of 9 tracers placed along an end-loaded cantilever. . . . 83
6.2 Positions of tracers detected by the algorithm, both before and after loading. . . . 84

# List of Tables


4.1 Activities of tracers used, measured before the experiments were performed. . . . . . . . 45
4.2 List of experiments performed under controlled rotation conditions. Tracers were kept stationary for 30 s, after which they were rotated with a controlled circular motion. . . . 48
4.3 Experiments performed for erratic rotation. . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.4 Experiments performed for rotation in the $xz$ axis with tracers inside the field of view at all times. Tracer positions used in configuration lists are of the form $(x, \theta)$, where $x$ is a fraction of the maximum radius, and $\theta$ is the angular position in °C. . . . . . . . . . 51
4.5 Experiments performed for rotation in the $xz$ axis with tracers exiting and entering the field of view. Tracer positions used in configuration lists are of the form $(x, \theta)$, where $x$ is a fraction of the maximum radius, and $\theta$ is the angular position in °C. . . . . . . . . . 52
4.6 Velocities used for velocity-controlled rotation. Since the reading on the motor was in arbitrary units, a digital tachometer was used to measure the speed in RPM. All runs used 4 tracers. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.7 Physical specifications of static deflection setup. . . . . . . . . . . . . . . . . . . . . . . . 55
4.8 Setup configurations used in static deflection tests. . . . . . . . . . . . . . . . . . . . . . 56

5.1 Scatter plots and best fit curves of a single tracer experiencing controlled circular motion. 59
5.2 Rotational velocities and standard errors for each dimension $x, y, z$. . . . . . . . . . . . . 59
5.3 Equation of best fit coefficients for VMPT tracking. . . . . . . . . . . . . . . . . . . . . . 75
5.4 Equation of best fit coefficients for CTRACK tracking. . . . . . . . . . . . . . . . . . . . . 75

6.1 Tabel showing Young's Moduli calculated at positions along a cantilever defined by the location of the tracers. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

# Chapter 1

# Introduction

Positron emission tomography (PET) is a nuclear imaging technique commonly used in nuclear medicine to produce three-dimensional images of functional processes within the body. In PET, a positron-emitting radionuclide is introduced into the system of interest. The radionuclide undergoes $\beta^+$ decay, during which a positron and a neutrino are produced. When the positron comes into the neighbourhood of an electron in the surrounding medium, an annihilation event occurs, and the total mass of the positron and electron is converted into energy in the form of two photons. To conserve the energy of the system, the two photons each have an energy of 511 keV, which is equivalent to the rest mass of an electron (or positron). To conserve momentum, the photons move in anti-parallel directions; that is, they travel back-to-back along a straight line.

PET scanners are used to detect these photons. They make use of detector cells composed of scintillator crystals which emit flashes of light when struck by a photon. If two photons are detected within some window of time, they are assumed to have originated from the same source. A line joining the positions of the two detections defines a *line of response* (LOR) along which the radioactive source is assumed to lie. Theoretically, the intersection of two LOR's would perfectly locate the radionuclide. However, since radioactive decay is a random process, and the radionuclide may be moving, the chance of two LOR's intersecting is close to zero. As such, algorithms have been developed to reconstruct images from sets of LOR's taken over a relatively large time frame. The image generated by these algorithms is commonly a static, three-dimensional image, showing the tracer concentration over some time period.

In recent years, PET scanners and algorithms have been adapted to be used in industrial systems, where one wishes to study the internal structure or mechanics *in situ*. PEPT (Positron Emission Particle Tracking) is a technique developed at the University of Birmingham which allows for a single tracer, which has been tagged with a radionuclide, to be tracked dynamically. PEPT allows one to measure the velocities and accelerations within a dynamic system, and has been used to successfully study granular fluids mechanics, flotation mechanics, and high-speed hydrocyclone dynamics.

During this thesis, an algorithm, dubbed the VMPT (Voronoi-based Multiple Particle Tracking) algorithm, is developed to track multiple tracers in a PEPT environment by making use of geometric structures called Voronoi tessellations. A Voronoi tessellation is a partition of space into so-called Voronoi cells, based on a set of $n$-dimensional discrete seed points. The cells are constructed such that each cell contains exactly one seed, with every point within that cell being closer to the interior seed than to any other seed in the set. Voronoi tessellations are well-understood mathematical structures, and have been used in fields as diverse as pure mathematics, astronomy and epidemiology.

A background is provided in this thesis, which covers the history and principles of both PET and PEPT. Voronoi tessellations are then described, along with their variations, and computer algorithms commonly used to generate these tessellations are discussed. Data clustering, another important facet of the VMPT algorithm, is also discussed, and two clustering techniques are described along with their strengths and weaknesses.

The VMPT algorithm is presented in two stages - the Location stage, and the Tracking stage.

The Location stage makes use of Voronoi tessellations, outlier-removal techniques, and data clustering to locate the positions of an arbitrary number of tracers at each time frame within a data set. The Tracking stage implements concepts and ideas used in popular multiple target tracking algorithms to track these located tracers. This stage is important because the tracers themselves cannot be uniquely identified by any physical properties.

A number of experiments, used to properly analyse various aspects of the algorithm, are summarised. These aspects include the performance and precision of the algorithm, as well as the robustness. To quantify the precision of the algorithm, the tracers were forced to rotate with a controlled circular motion. To test the robustness, multiple aspects were investigated, including tracers exiting and re-entering the field of view of the scanner, and two or more tracers colliding with each other.

The algorithm is then deconstructed and analysed, especially with respect to the accuracy of location, computational performance, and track loss. Track loss occurs when after some time a tracer's position is lost by the algorithm. This can occur for one of multiple reasons, including high acceleration and attenuation effects, as well as the tracer physically leaving the field of view of the PET scanner. A metric for accuracy of location is defined, and a method for calculating this accuracy is outlined. Various natural and artificial parameters, such as tracer velocity and discretization size, are identified as possibly having an effect on the accuracy of location, and the outlined method is used to quantify this effect. The accuracy of the VMPT algorithm for one tracer is then compared to the accuracy of the most commonly used Birmingham algorithm for single-PEPT.

Innovative applications of the VMPT algorithm, as well as multiple-PEPT in general, are discussed. A proof-of-concept experiment is described, in which a number of tracers are used to track the deflection of a cantilever with an end load.

The conclusion to the thesis highlights the advantages and disadvantages of the VMPT algorithm compared to existing multiple particle tracking techniques, and summarises the results of the algorithm analyses. It then discusses possible future improvements to the algorithm, focusing on parallelizing the location stage in order to decrease the processing time.

# Chapter 2

# Background

## 2.1 Positron Emission Tomography

### 2.1.1 History

Positron-emitting isotopes and annihilation events have been used in the medical field since the early 1950s. Sweet [1] and Wrenn *et al.* [2] published two independent studies, both of which used NaI(Tl) crystals to detect annihilation events in order to locate and treat brain tumours. Throughout the 1950s and 1960s PET was still in the experimental and theoretical stages.

The basics of emission and transmission tomography were developed in the 1960s by D. Kuhl and R. Edwards [3]. These techniques, then referred to as Positron Emission Transaxial Tomography, were further improved upon by Ter-Pogossian *et al.* [4] at the Washington University School of Medicine. The developments and improvements led to the first tomographs being built at the University of Pennsylvania. The construction of the first true PET cameras was limited by two main factors: the limited physical space within the camera, and the number of photomultiplier tubes needed. Previously, the tomographs used as many photomultiplier tubes as there were scintillator crystals. Since the photomultiplier tubes were relatively large and expensive, this limited the number and size of scintillator crystals, and therefore detector cells. In 1972, Burnham and Brownell showed that a single photomultiplier tube could be used for multiple crystals [5], which lead to improvements to resolution in both ring and cylindrical PET cameras.

Early PET cameras were only capable of detecting lines of response in a transaxial plane, since only detectors in the same ring were linked. The generated images were therefore two dimensional slices over the length of the field of view. These slices were then stacked to generate three dimensional images; however, to make up for the high signal-to-noise ratio in these images, data smoothing methods were implemented to create interpretable images. The smoothing caused a loss of spatial resolution in the final results. Modern PET cameras can create fully three dimensional images by allowing any two detectors in the array to detect lines of response. This means that all of the lines of response can be detected, which leads to a larger storage space requirement.

The greatest obstacle to the advancement and practical use of PET was the difficulty involved in producing suitable radioisotopes. In 1978, Ido *et al.* [6] developed the means to produce fluorodeoxyglucose (FDG), showing that $^{18}$F can be used as the $\beta^+$ positron emitter. Since $^{18}$F has a convenient half-life for use in PET, and FDG is metabolised similarly to many metabolic processes in the body, this lead to increased viability in the usage of PET as a medical imaging tool.

### 2.1.2 Data Acquisition and Image Reconstruction

PET cameras work by detecting $\gamma$-rays with an energy of 511 keV, which are created by positron annihilation events generated by $\beta^+$ decay in certain radioisotopes. $\beta^+$ decay involves an unstable nucleus stabilising by converting a proton into a neutron, a positron, and an electron neutrino. Since

the positron is the antimatter version of an electron, an annihilation event occurs when the positron comes into contact with an electron in the surrounding medium. When this positron annihilation event occurs, the mass of a positron and electron are converted into energy in the form of photons. The mass of an electron is equivalent to 511 keV of energy; therefore the total energy of the photons emitted must be 1.22 MeV to conserve energy. Since annihilation events normally occur when the electron and positron are moving slowly, the total linear momentum of the two particles is approximately zero. In order to conserve this momentum, the 1.22 MeV must manifest as two photons travelling along a collinear axis in opposite directions. However, in reality the positron and electron have some small amount of momentum, so the photons may not be perfectly collinear.

The detecting elements in the scanner are scintillator crystals; multiple compounds can be used for these crystals, but they all function in a similar manner. When a scintillator crystal is excited by ionizing radiation (in this case the $\gamma$-rays) the crystal emits the energy in the form of a flash of light. Photomultiplier tubes are set up to detect these flashes of light, with the most common number being one tube per four crystals[5]. The tubes make use of the photoelectric effect to eject electrons when incident photons strike the detectors. The electrons are then multiplied via a secondary emission process, which creates a measurable current. Since the final current is proportional to the energy of the original $\gamma$-rays, the machine can determine whether the ionizing radiation was in fact a 511 keV photon, as opposed to one from a different energy level. In the majority of PET scanners the scintillator crystals are arranged in consecutive rings along the length of the cylindrical field of view of the camera, although there are some exceptions to this. For instance, parallel scanners have the crystals arranged in two parallel planes.

When a scintillator crystal detects a $\gamma$-ray, the system opens an electronic window for a period of time on the order of nanoseconds. If another $\gamma$-ray is detected by a different crystal within this window, the two detections are considered to have originated from the same source. The position of the detection is defined as the centre of the appropriate detector cell; a line joining the centres of two paired detector cells defines an LOR along which the source of radiation is assumed to lie. Note that a single LOR cannot provide enough information to determine *where* along that line the tracer lies.

Spurious lines of response may be recorded due to a number of situations. The first is when one or both of the photons undergoes Compton scattering in the surrounding medium. This happens when a photon's path is deflected by a neighbouring charged particle, most commonly an electron. The camera may still detect both photons, but the connecting them will not pass through the annihilation position. Spurious detections may also occur when two annihilations happen almost simultaneously. After the time window is opened due to the detection of one photon, the camera may then detect a photon created in the other annihilation event. This would appear to the camera as a single LOR, but it would in fact pass through neither of the two true events.

The camera may also fail to detect some events. If only a single photon is detected within the window period, the detection is discarded. Similarly, the detection is discarded if more than two photons are detected within a single window period.

Depending on the software used in the camera, a LOR can either be stored in binned sinograms [7] or in list mode format. When saved in list mode format, the detected pairs are stored along with the time-stamp of the event. The raw data in a list mode file is stored in sequential words (2-byte data types) to provide efficient compression. A word can represent one of three pieces of information:

- A false detection. Used when the camera fails to detect an event properly, as described above.
- A successful detection. The information in this case is a pair of numbers, each representing a detector cell. The actual positions of the cells are later reconstructed using the dimensions and shape of the camera.
- A time increment. The time at the initiation of the experiment is assumed to be zero. When a word representing a time increment is found, the time at detection is incremented by one. Since the 'time' at detection is an integer, the actual time since the beginning of the experiment is found by multiplying the time at detection by the time-resolution of the camera. In the case of the EXACT-3D camera, this time-resolution is one 1 ms.

Sinograms are used in PET to generate 2D cross-sectional images, which are then stacked to create 3D images. A LOR is defined by its angular orientation with respect to the y-axis of the field of view, as well as the distance of that line from the centre of the gantry. Because the lines of response are stored this way, only those lines that are detected by cells within the same ring are used to reconstruct the image, hence the 2D nature of the images. Considering a theoretical point of $\beta^+$ decay within the field of view of the camera, the angle and displacement for each LOR generated by that point can be plotted, with the angle on the y-axis and the displacement from the centre on the x-axis. The subsequent plot is a vertical sine graph. The location of the point can be found using this sinogram; the amplitude of the sine wave represents the distance of the point from the centre of the field of view, while the phase represents the angular location about the central axis. The set of sinograms generated by the PET camera over the experimental time takes the form of many overlapping sine waves. Using one of several common Fourier filters this data can be smoothed and separated into individual sine waves, which can then be used to generate the final image. This image reconstruction method using sinograms is referred to as the filtered back-projection technique.

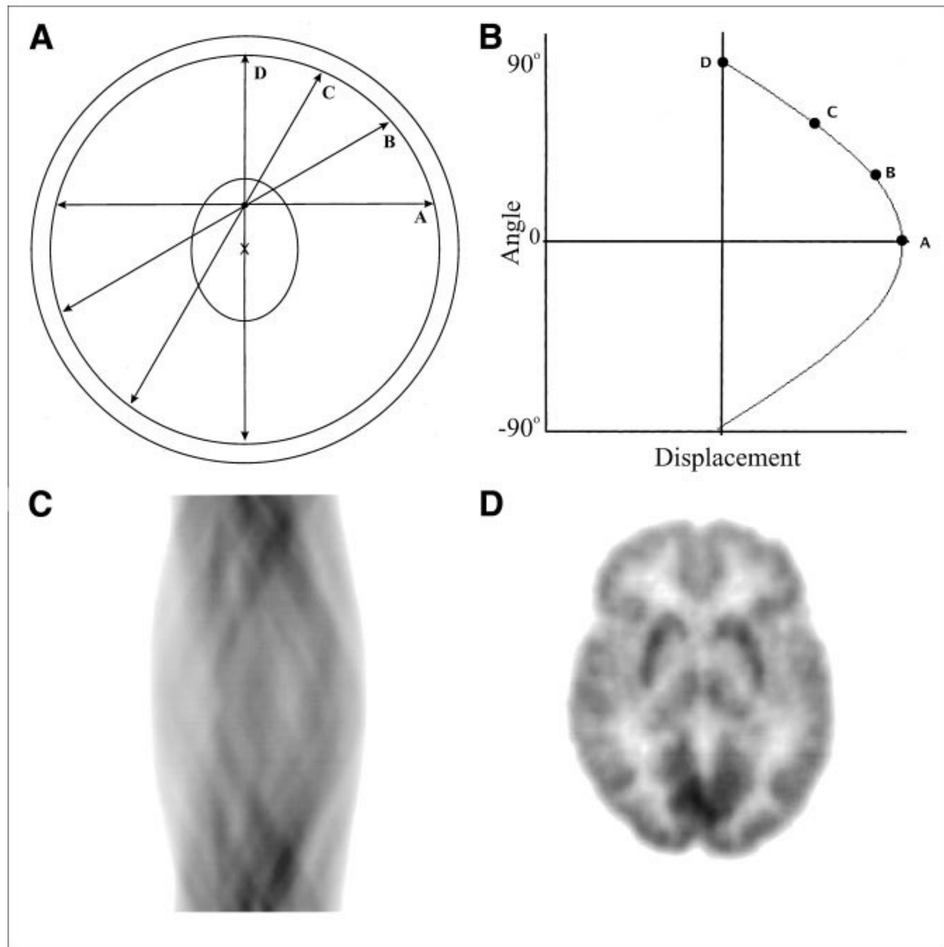



Figure 2.1: Sinogram formation in PET. (A) Lines of response generated by a single point of radioactivity. (B) Angles and displacements plotted to form a vertical sine graph. (C) Collection of multiple overlapping sinograms. (D) Image reconstructed from sinograms. (Image from [7]).

In recent years the filtered back-projection technique has fallen out of favour, with iterative techniques being used instead. Iterative algorithms work by making an initial guess at the distribution of the activity; the theoretical projections due to this distribution are then generated and compared to

the actual experimental data. The difference between the generated and measured projections are used to correct the initial guess; this process is repeated iteratively until the difference converges to some criterion [8].

Originally, the problem with iterative reconstruction techniques was the computational run time. A prominent group of algorithms used in iterative techniques today is the group of expectation-maximisation algorithms, of which the first was described by Dempster *et al.* in 1977 [9]. Specifically, the most commonly used algorithm is the ordered-subset expectation maximisation (OSEM) algorithm developed by Hudson and Larkin in 1994 [10]. This algorithm was the first iterative algorithm to be used practically in medical imaging because of its advanced computational acceleration techniques. Today, iterative techniques are becoming easier to implement efficiently with the increasing speed of available personal computers.

## 2.2 Positron Emission Particle Tracking

Positron Emission Particle Tracking, or PEPT, is a measurement tool used to track a single radioactive particle in motion. PEPT uses PET cameras, but where PET generates a single image showing the distribution of the radiation, PEPT generates a position-time history for the radioactive particle. The particles used are referred to as tracers, and are often made from glass beads labelled with a radioisotope that undergoes $\beta^+$ decay. The most common radioisotopes used in PEPT are $^{18}$F, $^{22}$Na and $^{68}$Ga. The size of the tracer itself can range from a few hundred micrometers to a few millimetres in diameter. In 2012, Cole *et al.* [11] managed to successfully track a 50 μm tracer.

Hawkesworth *et al.* [12] was the first to use a PET camera for non-medical applications. Hawkesworth and his group were based at the Positron Imaging Centre at the University of Birmingham, where PEPT research has continued since 1984, when the group acquired their first PET camera. The algorithm developed by the Birmingham group is known as the Birmingham algorithm, and was improved upon in 1993 by Parker *et al.* [13]. Theoretically, given two consecutive lines of response from a data set, the location of the tracer will be at the intersection of the two lines. However, the lines will always have some noise, and there will most likely be false lines in the set. This means that the likelihood of two LOR's actually intersecting in three-dimensional space is almost zero. The Birmingham algorithm works by selecting $N$ consecutive lines of response from the data set. It then triangulates the location of the tracer from this set by finding the point $\mathbf{x}$ which minimises the sum of the perpendicular distances from $\mathbf{x}$ to each of the $N$ lines of response. The algorithm then discards a number of lines of response until a fraction $f$ of the number of lines are remaining. The location and discard steps are repeated iteratively until a predetermined number of lines remains or the distance metric is exceeded by the RMS error of the triangulation. Another centre dedicated to PEPT research is the positron tracking facility at iThemba LABS in Cape Town, South Africa[14]. The Cape Town group owns two scanners: the Siemens ECAT HR++ EXACT3D scanner, which is one of the most sensitive PET scanners ever built [15]; and the ADAC Epic Vertex Gamma camera, a parallel-plate scanner commissioned by Marconi.

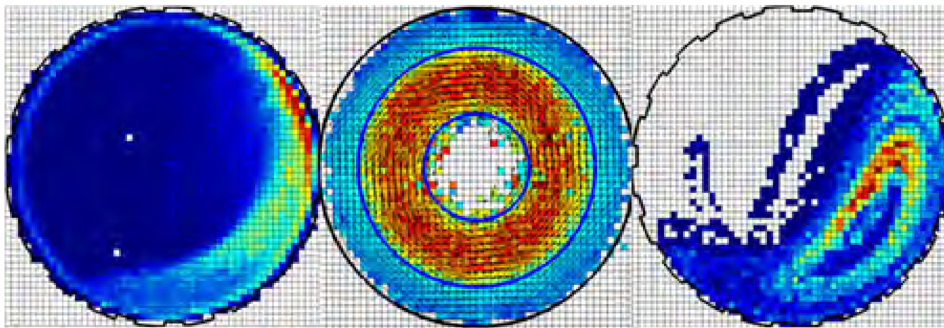


Figure 2.2: Position and velocity profiles of a tracer within a tumbling mill. The scanner used is the Siemens EXACT3D, and the Birmingham algorithm was used for processing.

Since its inception in 1984, PEPT has proved itself a valuable measurement tool, with many applications. Parker *et al.* [16] used PEPT to observe the motion of particles moving freely within a horizontal rotating drum. The drum simulates a tumbling mill used in many mines to crush ore. In 2000, Fangary *et al* made use of PEPT to measure the flow fields of non-newtonian fluids in a stirred mill. Chang *et al.* made use of PEPT to observe the motion of a particle within a hydrocyclone [17]. PEPT is also commonly used in conjunction with numerical methods, such as the discrete element method (DEM) and smooth particle hydrodynamics (SPH), as an experimental validation tool [18].

Tracers used in PEPT experiments can be manufactured in a number of ways. For larger tracers, with a diameter larger than a millimetre, a drop of radioactive fluid or a small amount of radioactive salt can be placed inside a small casing. The casing is often a glass bead of some size and shape (most commonly spherical or cylindrical) with a small hole drilled into it. The hole is then sealed with a suitable resin or epoxy. Whether a salt or fluid is used is dependent on the radioisotope: for instance, $^{22}$Na is manufactured as a salt. Alternatively, a tracer can be made with a non-radioactive isotope of the desired element. The tracer is then bombarded with a charged particle beam, such as $^{3}$He. When manufacturing smaller tracers (less than one millimetre), different methods must be used. The preferred method makes use of anion exchange resins [19], which are immersed in a solution of radioactive ions. The ions are absorbed by the resin, and the resin itself is used as the tracers once it hardens. Fan *et al.* [20] improved the ion uptake by introducing metallic ions to the surface of the resin, and Cole *et al.* [11] took advantage of the ion-exchange technique to manufacture tracers with a diameter of 50 micrometers.

### 2.2.1 Multiple Particle Tracking in PEPT

In 2004, Gundogdu [21] published a paper on a new algorithm used to track multiple particles using a PET camera. The algorithm was based on the idea of intersections between pairs of LOR's. Actual intersections in 3 dimensions happen rarely due to the random nature of positron ejection, so consequently the probability of two or more annihilation events occurring at the same point in space is close to zero. This probability is further reduced when the tracer is in motion. Gundogdu therefore defined an intersection as two lines coming within some user-defined maximum distance of each other. The actual point of intersection was the midpoint of a line perpendicular to and joining two "intersecting" lines of response. A k-means clustering method [22] was then used to cluster the set of intersections, and the location of a tracer was found by finding the geometrical centroid of the cluster.

In 2006, Yang *et al.* created an algorithm to track three tracers with different activity levels. The algorithm was an extension of the original PEPT algorithm; to find the first tracer (that with the largest activity) the original iterative algorithm was run. Once the tracer location was determined, all of the discarded lines of response were used as the next set to find the location of the tracer with the median activity. Finally, the lines discarded from this second run were used to locate the third tracer. In order for the tracers to be unique under the location algorithm, their relative activities had to increase exponentially. That is,

$$A_1 = 2A_2 = 4A_3$$

where $A_i$ is the activity of the $i$-th tracer. As such, the total activity level inside the field of view of the camera increases quickly with the number of tracers, and the saturation level of the detectors is reached at fairly low tracer counts. This puts a physical limit on the total number of tracers that could be used in an experiment.

In 2008, Yang *et al.* investigated using multiple PEPT for making innovative measurements previously impossible with only single PEPT [23]. With one tracer, measurement is limited to translational motion. Using three tracers attached to a solid object, Yang was able to measure and quantify the rotational motion of the object, providing further insight as to the total energy of the object.

A major limitation of the initial method was that the tracers had to remain roughly in the centre of the field of view of the camera; when a tracer approached the edges, its apparent activity level (according to the camera) would decrease, and the algorithm would fail to uniquely locate the tracer. To solve this problem, in 2007 Yang *et al.* [24]. mapped the relative detection levels across the length

of the field of view of the camera. Once the drop-off was quantified, a transformation could be applied to the location algorithm based on its 3D location. This allowed for more accurate tracking over a larger percentage of the field of view of the camera.

In 2012 Bickell *et al.* investigated a new method of multiple particle tracking [25], called the line density method. To find the initial locations of the tracers, the algorithm first divided the field of view of the camera into voxels of a fixed size. Each voxel was then assigned a density value proportional to the number of LOR's that passed through that voxel. Given that $n_t$ tracers were used in the experiment, the $n_t$ voxels with the highest densities were considered as containing a tracer. The location of the tracer was assumed to be in the centre of the voxel. The initial locations of all of the tracers was then confirmed by the user, who would have a rough idea of the placements of the tracers within the camera. The accuracy of this method was highly dependent on the size of the voxels; although the tracer location was assigned to the centre of a voxel, in reality it could have appeared anywhere inside the voxel. However, the voxels could not be made too small, as the smaller the voxel, the lower the probability of sufficient LOR's passing through that voxel. To find successive tracer locations a different method was used. A scalar search distance was used , which was defined by the user and took a default value assigned by the authors. For a reliable tracer location, the search distance had to be chosen such that the magnitude of the tracer acceleration was less than twice the search distance divided by the time interval between frames. That is

$$|\mathbf{a}| \leq 2d_s/(\Delta t)^2$$

where $\mathbf{a}$ is the acceleration of the tracer, $d_s$ is the search distance and $\Delta t$ is the time between successive frames. Each LOR passing within the search distance of the previous tracer location was then considered as associated with that tracer. Next, a method similar to the original PEPT algorithm was implemented, in which an iterative method was used to find the point that minimised the sum of the distances to all the valid LOR's. This line density method was also applied to PET; the density map across voxels was smoothed using Fourier filters and the resulting images showed the concentrations of the radioisotope throughout the volume. By breaking the entire set of LOR's into frames of predetermined line counts, the algorithm was shown to be capable of generating semi-dynamic PET images.

A common limitation to all the methods discussed is that the exact number of tracers in the field of view has to be constant and explicitly defined by the user. In Gundogdu's algorithm, the k-means clustering method used requires the number of clusters as an input. Furthermore, k-means has no concept of noise; every data point is considered and each of them contributes to the cluster. Yang's algorithm searches each time for the number of uniquely labelled tracers defined by the user. In Bickell's algorithm the locations in each frame after the first depend on the locations in the previous frame. In all of the above examples, any tracer exiting or entering the field of view of the camera requires manual intervention; the program pauses and the user redefines the number of tracers for which to search.

## 2.3 Voronoi Diagrams

A Voronoi diagram is a geometric data structure which divides a plane populated with a set of points (seeds) according to the nearest neighbour rule. Formally, we denote a set of $n$ seeds in a plane with $S$, and the corresponding Voronoi diagram is denoted by $V(S)$. Now for two distinct seeds $p, q \in S$ we define the *dominance* of $p$ over $q$ by

$$\text{dom}(p,q) = \{x \in \mathbb{R}^2 | \delta(x,p) \leq \delta(x,q)\},$$

which is the subset of the plane which is closer to $p$ than to $q$. Note that normally $\delta$ denotes the Euclidean distance function, but other distance functions may be used as well; of particular importance is the Manhattan distance function. Now the *region* of a seed $p$ is that part of the plane which contains all points closer to $p$ than to any other seed, or the portion containing all the dominances of $p$ over every other seed in the set $S$. This can be shown formally with

$$\text{reg}(p) = \bigcap_{q \in S-p} \text{dom}(p,q).$$

This creates a distinct region encapsulating each seed, with every point bound by a region being closer to the enclosed seed than to any other seed. The Voronoi $V(S)$ will always contain exactly $n$ regions, since a region cannot be empty. However, the plane which contains the seeds may be bound or unbound; when it is unbound, there will exist some regions with an infinite area. This is due to the existence of points which, while arbitrarily far from a seed $p$, are still closer to $p$ than to any other point $q$. Except for the arbitrarily large regions, each region forms a convex polygon, with each point on an edge of a polygon being equidistant from exactly two sites. The concept of a two-dimensional Voronoi diagram can be extended to higher-dimensional (most importantly three dimensional) Voronoi tessellations, where each region is a polytope surrounding the corresponding seed point.

It is interesting to note that Voronoi diagrams behave linearly with respect to the number of seeds. That is, the number of edges increases linearly with the number of seeds. This may seem counter-intuitive, since if there are $n$ seeds there should be $\binom{n}{2} = O(n^2)$ combinations and therefore $O(n^2)$ separators between seeds. However, it can be shown that the number of separators which contribute to an edge in $V(S)$ is linearly with respect to $n$ by making use of Euler's relation $n + v - e \geq 2$. Since each of the $v$ vertices is the junction of at least 3 edges and each of the $e$ edges has exactly 2 vertices (except in some degenerate and non-bound cases), we can say that $2e \geq 3v$. Combining this with Euler's relation gives

$$e \leq 3n - 6$$
$$v \leq 2n - 4$$

which show that both the number of edges and the number of vertices are linear in $n$.

Voronoi diagrams are named after George Voronoy, who defined the general $n$-dimensional case in 1908, and are also referred to as Voronoi tessellations, Voronoi decompositions or Voronoi partitions. However, they have been studied from as early as 1840, when Carl Friedrich Gauss showed that Voronoi diagrams can be used to interpret some quadratic forms. Later, in 1850, Peter Gustav Lejeune Dirichlet used this idea to prove the unique reducibility of quadratic forms. In 1854, John Snow made informal use of Voronoi diagrams to pinpoint the source of a cholera outbreak in London [27]. He used the locations of various water pumps as the seed points to generate nearest-neighbour cells on a map. The cell with the highest concentration of cholera cases (the Broad Street pump) was then identified as the source of the outbreak.

The theory and mathematics behind Voronoi diagrams have been studied extensively, with Franz Aurenhammer [28] performing a comprehensive survey in 1990. Voronoi tessellations are similar to fractals and Fibonacci spirals in that they frequently appear in nature. The spots on giraffes exhibit Voronoi patterns, as do segmented wings on insects such as dragonflies, as can be seen in Figure 2.4. Voronoi tessellations intrinsically describe least-energy configurations; this means that packed configurations tend to take the shape of approximate Voronoi tessellations. This is illustrated in Figure 2.4b, which shows bubbles packing together in a confined space. Biological cells being packed exhibit similar geometric characteristics to these bubbles. Crystal growth also display Voronoi tessellation properties. If multiple crystal seeds start growing at the same time and at the same rate, the borders of the growth stop when they meet other borders, leading to an overall Voronoi tessellation.

The Voronoi diagram is widely considered to be one of the most mathematically fundamental constructs defined by a set of discrete points. It is linked to other geometrical structures, in particular the Delaunay triangulation, of which it is the mathematical dual. A Delaunay triangulation is one in which two seeds are joined by a straight line if and only if the corresponding Voronoi polygons share a common edge. This divides the region of space containing the seed points into a mesh of triangles in two dimensions, and tetrahedrons in three dimensions. If a circumcircle is drawn around each of the triangles, it will contain no interior seed points, and the centre of this circle will correspond to a vertex of the Voronoi diagram.

### 2.3.1 Applications

Voronoi tessellations have applications in many fields of study, including theoretical fields such as pure mathematics, as well as more practical fields such as computer science, biology and even city planning.

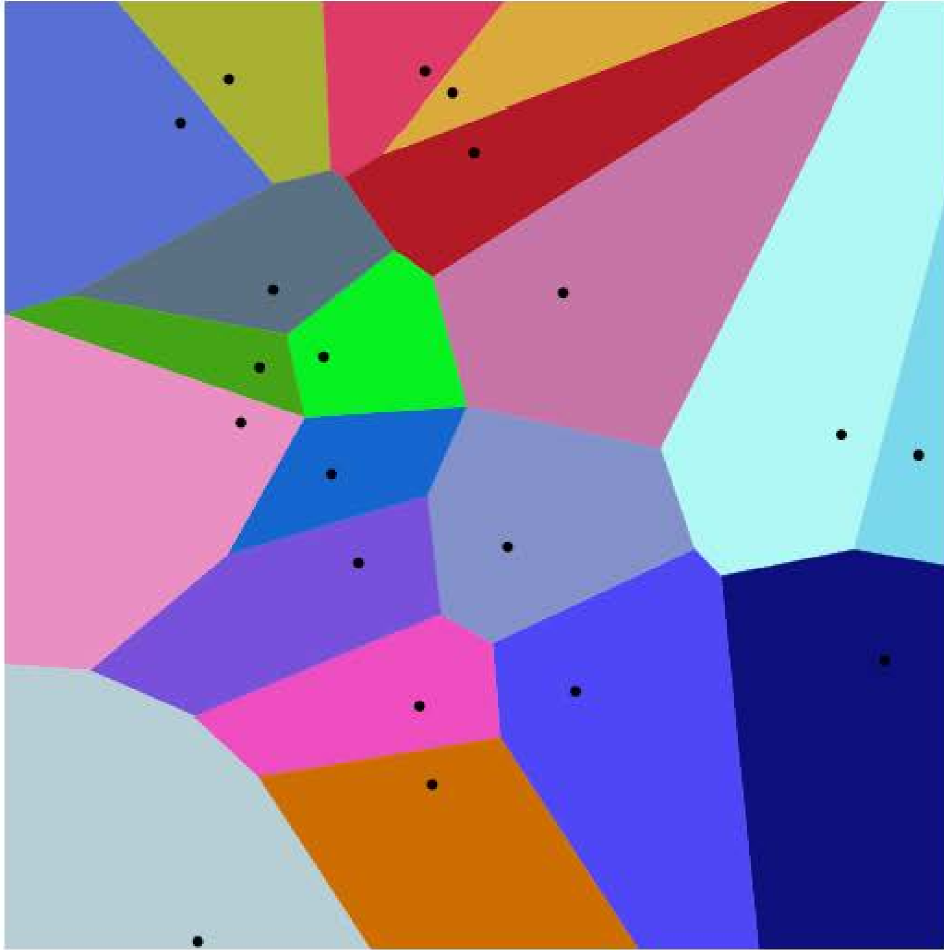
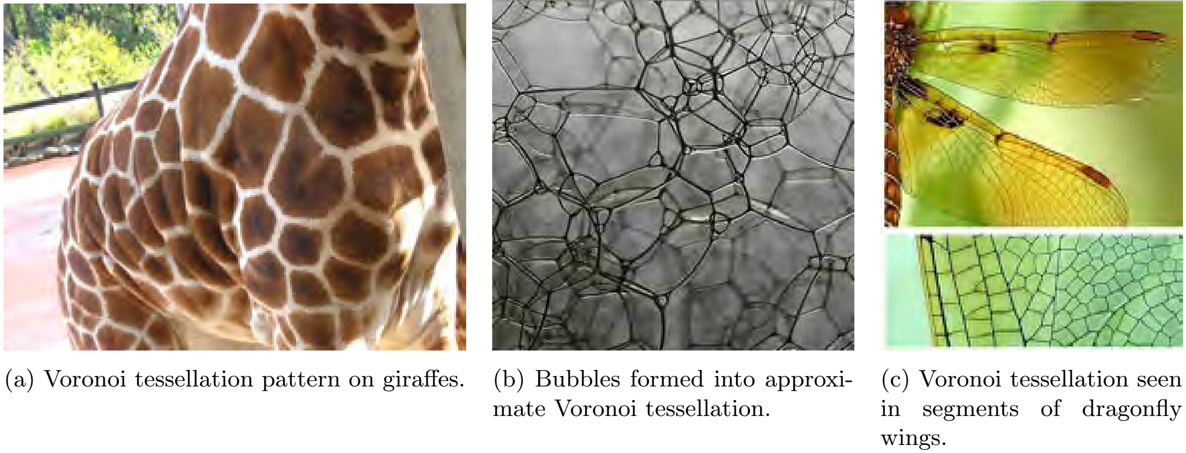


Figure 2.3: Example of a 2D Voronoi diagram using 20 seed points [26].

In computer science, Voronoi diagrams can be used for associative file searching. This problem is most commonly known as the post-office problem: given a set $S$ of $n$ seeds (or post offices) on a plane (or map), what is the quickest way to find the closest post office to some point $p$. A brute-force method to find the closest entry would be a simple comprehensive search, and would be of $O(n)$ complexity given $n$ entries. However, if it is known that the data set will be accessed multiple time, a Voronoi tessellation $V(S)$ can be used to reduce the complexity of the search, which is reduced to finding the region that contains $p$. Kirkpatrick [29] and Edelsbrunner *et al.* showed in 1983 and 1986 respectively that this problem can be solved in $O(\log n))$ time and requires $O(n)$ memory usage. This shows that with no increase in the memory usage the query time can be significantly reduced. This nearest-neighbour method itself has applications outside of computer science. The same concept can be used in aviation to find the closest airfield in the case of an emergency diversion. It can also be used in epidemiology, as discussed above, to locate the source of an epidemic.

Voronoi diagrams have been shown to be useful in data clustering. Clustering involves partitioning a data set into subsets such that elements of a subset are similar, while elements across subsets are dissimilar. If the similarity metric can be reflected by points on a plane (or higher dimensional volume when the number of attributes is greater than two) then a Voronoi tessellation can be calculated using the data points as seeds. Certain properties of the Voronoi diagram can then be used to determine proximity (or similarity) of seeds. For denser clusters, the Voronoi regions will have smaller areas; where the seeds are more spread out, the regions will have relatively large areas. Furthermore, the



(a) Voronoi tessellation pattern on giraffes.

(b) Bubbles formed into approximate Voronoi tessellation.

(c) Voronoi tessellation seen in segments of dragonfly wings.

Figure 2.4: Voronoi tessellations appearing in nature

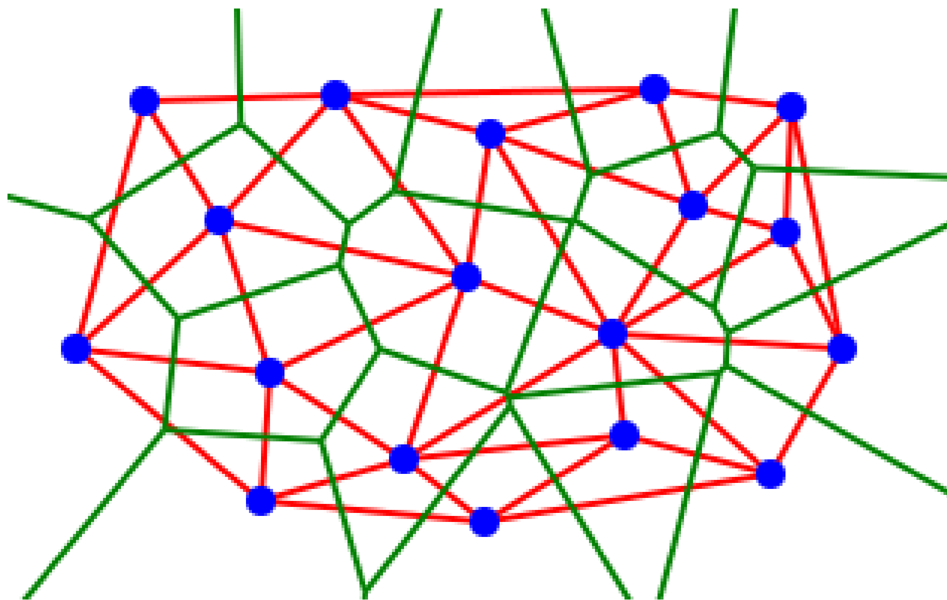


Figure 2.5: Voronoi diagram (green) with Delaunay triangulation (red) overlaid.

popular K-means clustering algorithm [22][30] naturally divides the data set into Voronoi regions.

Interestingly, Voronoi diagrams can be used to control the hardware of read-write systems. Using an $L_1$, or Manhattan, distance metric, Voronoi diagrams can be used to approximate the minimum time it takes for a read-write head to retrieve a set of requested records, as well as the corresponding path. Although an exact solution to the problem was shown to be NO-complete [31], an approximate solution can be found by creating a Voronoi diagram using the requested record locations as the seed points. The time taken to compute the tessellation is of the order $O(n\log n)$, and thereafter the minimum-$L_1$-length tree can be constructed. Since the tree construction is quicker than the Voronoi tessellation, the total time taken is of order $O(n\log n)$.

Voronoi diagrams can be used in the field of robotics to control translational motion. A generalisation of the Voronoi diagram is the weighted Voronoi diagram, in which each seed has a different weight; this weight may be visualised as a circle instead of a point. If obstacles within the field of view of the robot are modelled as circles, a weighted Voronoi diagram may be constructed using the radii of the circles as the weights. Intuitively, the edges of the power diagram are the "safest" paths, because they will always be farthest from any two obstacles.

Because Voronoi tessellations have natural-looking configurations, they are often used in computer graphics. They are specifically used to generate organic-looking textures, for instance lava flows. They are also used in computer graphics to calculate three-dimensional shattering effects.

Voronoi tessellations have been used in biology to model biological structures. Bock *et al.* [32] used Voronoi tessellations to model the arrangement and growth of cells in cancerous tissue. The interactions between cells were simulated using adhesive and repelling forces along the edges of the Voronoi regions. The feasibility of the method was shown by simulating the differential equations for the position and velocity of the cell centres. Li *et al.* [33] modelled bone micro-architecture using Voronoi tessellations. The model was then used to analyse the bone strength and bone density.

Voronoi tessellations are not just used in scientific fields. Structures and art which make use of Voronoi shapes tend to be visually attractive. An example of this is the winning entry for the redevelopment of the Gold Coast Cultural Precinct, which made extensive use of Voronoi diagrams in not just the structures but also the landscape [34].

### 2.3.2 Variations

Besides the standard Voronoi tessellation, multiple variants exist.

The distance metric used may not always be Euclidean; other commonly used metrics are the Manhattan and Mahalanobis metrics. The Manhattan or taxicab distance between two vectors $\mathbf{p}, \mathbf{q}$ in $n$ dimensions is denoted by $d_1$, since it uses the $L_1$ norm. With a fixed Cartesian coordinate system, this is found by projecting the line segment joining $\mathbf{p}$ and $\mathbf{q}$ onto the coordinate axes. The Manhattan distance is then the sum of the lengths of these projections. Mathematically, we have

$$D_1(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{n} |p_i - q_i|,$$

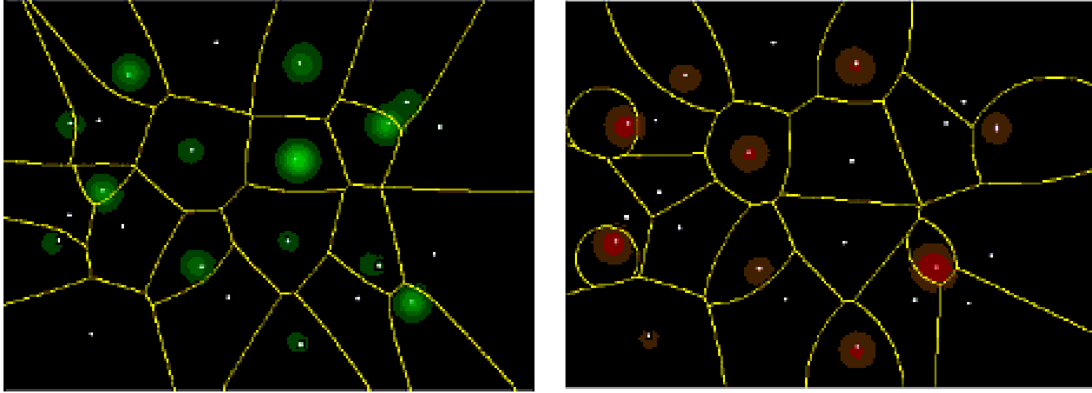
where $\mathbf{p}$ and $\mathbf{q}$ are $n$-dimensional vectors.

The Mahalanobis distance on the other hand, is the distance from a point $\mathbf{p}$ to a distribution $D$. It is defined by

$$D_m(\mathbf{p}) = \sqrt{(\mathbf{p} - \boldsymbol{\mu})^T \mathbf{S}^{-1} (\mathbf{p} - \boldsymbol{\mu})},$$

where $\mathbf{p} = (p_1, p_2, ..., p_n)^T$ is the n-dimensional point of interest; $\boldsymbol{\mu} = (\mu_1, \mu_2, ..., \mu_n)^T$ is the mean of a set of observations; and $\mathbf{S}$ is the covariance matrix of that set of observations. The Mahalanobis distance measures the number of standard deviations between $\mathbf{p}$ and the mean of the data set $D$, generalised to multiple dimensions.

The boundaries of the Voronoi cells using other distance metrics may be more complicated than in the simple Euclidean case. In the case of the Euclidean distance metric, the boundary between two seeds $\mathbf{p}, \mathbf{q} \in \mathbb{R}^n$ will always be of dimension $n - 1$; that is, the boundary for two points is a subspace



(a) Additively weighted Voronoi diagram. (b) Multiplicatively weighted Voronoi diagram.

Figure 2.6: Various weighted Voronoi tessellations using a Euclidean distance metric. The size of the coloured halos indicate the size of the weights. Images from [35]

with codimension 1. However, this may not hold true for other distance metrics, even in the simple 2-D case.

As discussed previously, a weighted Voronoi diagram assigns an additive or multiplicative weight to each seed point. Multiplicative weighted Voronoi tessellations are created when the distance between two points is multiplied by a positive weight. These diagrams are referred to as circular Dirichlet tessellations [36]. The boundaries of circular Dirichlet tessellations are not necessarily straight; they may be straight line segments or circular arcs. Furthermore, the cells may be disconnected, non-convex and even have holes, unlike the strictly convex cells found in standard Voronoi diagrams. An example of a circular Dirichlet tessellation found in nature is crystal growth in which each crystal grows at a different rate.

Alternatively, additively weighted Voronoi diagrams are generated when positive weights are subtracted from the distances between two points, and are referred to as hyperbolic Dirichlet tessellations or Apollonius diagrams. The boundaries of hyperbolic Dirichlet tessellations may be either straight line segments or hyperbolic arcs.

Power diagrams are Voronoi diagrams defined for a set of circles instead of points. They may be thought of as Voronoi diagrams with weights equal to the radii of the circles and seeds in the centre of the circles. The weights are then added to the squared distances between points.

Approximate Voronoi diagrams, or AVD's [37], are used to decrease both the memory and time needed to compute a Voronoi diagram, especially in higher dimensions. An AVD accepts two constants $t$ and $\epsilon$ as parameters. If we have a set $S$ of $n$ seeds in $\mathbb{R}^d$, the Voronoi space is divided into cells of constant complexity. Each cell $c$ is associated with $t$ seeds, and any point within $c$ is an approximate nearest neighbour to one of the associated seeds. An approximate nearest neighbour is one that is not much further than the nearest neighbour. Formally, given a query point $\mathbf{q}$ then $\mathbf{p}$ is an approximate nearest neighbour to $\mathbf{q}$ if

$$D(\mathbf{p}, \mathbf{q}) \leq (1 + \epsilon)D(\mathbf{p}^*, \mathbf{q})$$

where $D$ is some distance metric, $\epsilon > 0, \in \mathbb{R}$ and $\mathbf{p}^*$ is the true nearest neighbour to $\mathbf{q}$. Arya *et al.* [37] showed that the storage needed was of order $O(n/\epsilon^d)$ for $t = 1$, and the query time needed for a single seed was $O(\log(n/\epsilon^d))$ for a total construction time of $O(n\log(n/\epsilon^d))$. In two dimensions the AVD does not significantly reduce the computational resources needed, but for higher dimensions, where an exact Voronoi tessellation may not be feasible, it provides a more efficient alternative.

### 2.3.3 Algorithms

#### Lloyd's Algorithm

Lloyd's algorithm is used to compute centroidal Voronoi diagrams. A centroidal Voronoi diagram is one which has the seed points as the centroids of their respective Voronoi regions.

#### Fortune's Algorithm

Fortune's algorithm, also referred to as the sweep-line algorithm [38] for Voronoi tessellations, was originally designed to generate Voronoi diagrams on a plane. The algorithm makes use of a 'sweep line' and a 'beach line'. The sweep line is by convention a straight vertical line which sweeps across the data set from left to right. During the sweep, any points to the left of the line will have been incorporated correctly into the diagram; those to the right have yet not been considered. The beach line is a piecewise set of sections of parabolas. For each seed point to the left of the sweep line, a parabola can be defined which contains all points equidistant from the seed point and the sweep line. The beach line is the union of these parabolas (see Figure 2.7).

As the sweep line moves across the plane, the beach line is continuously updated. While this happens, the edges of the Voronoi regions are generated. These edges follow the vertices of the beach line, which are defined by the intersections of two parabolas.

To increase efficiency, a binary search tree and a priority queue are used to maintain the beach line structure. Without the use of the binary search tree, the algorithm would have a complexity of O($n$log$n$). However, since there are O($n$) events to process, and a binary search tree provides O(log$n$) search time, the total complexity of Fortune's algorithm is O($n$log$n$).

Fortune's algorithm can also be extended to 3D with some changes. Instead of a sweep line, a sweep plane angled at 45° to the direction of the sweep. The points are modelled as cones with the tip at the point location, and a cone angle of 45°. The intersection of the sweep plane with the cones is analogous to the beach line.

#### Bowyer-Watson Algorithm

The Bowyer-Watson algorithm is an algorithm used to compute the Delaunay triangulation of a set of seed points. First, three points are randomly selected; this is the minimum needed to generate a Delaunay triangulation. With these three points, a triangulation is generated, which is trivial since it is simply a triangle. From there, each of the remaining seed points is added one by one. After each point is added, the Delaunay triangulation is re-calculated by deleting all triangles whose circumcircle includes the new point (since a Delaunay triangulation consists of circumcircles with no interior points).

Once a Delaunay triangulation has been generated, the corresponding Voronoi tessellation can be computed in O($n$) time, since the Voronoi diagram is the mathematical dual of the Delaunay triangulation. The connectivity graph of the existing triangulation at each step can be used to efficiently locate any triangles that must be deleted at each step. This leads to a complexity of O($n$log$n$) on average; however some degenerate cases exist which lead to a complexity of O($n^2$).

Once again, the Bowyer-Watson algorithm can be extended to 3D by first computing the 3D Delaunay triangulation.

## 2.4 Cluster Analysis

Cluster analysis involves the separation of data objects into groups, or clusters. A data object in a cluster will be more similar to other objects in that group than to objects in other groups. It is used as a statistical tool in fields such as machine learning, image analysis, pattern recognition, bioinformatics, market research and social network analysis. There is no strict definition for what constitutes a cluster, and as such there are many distinct clustering algorithms, each defining a cluster differently. The idea of similarity is also somewhat ill-defined; if the objects are simply points in space, various distance

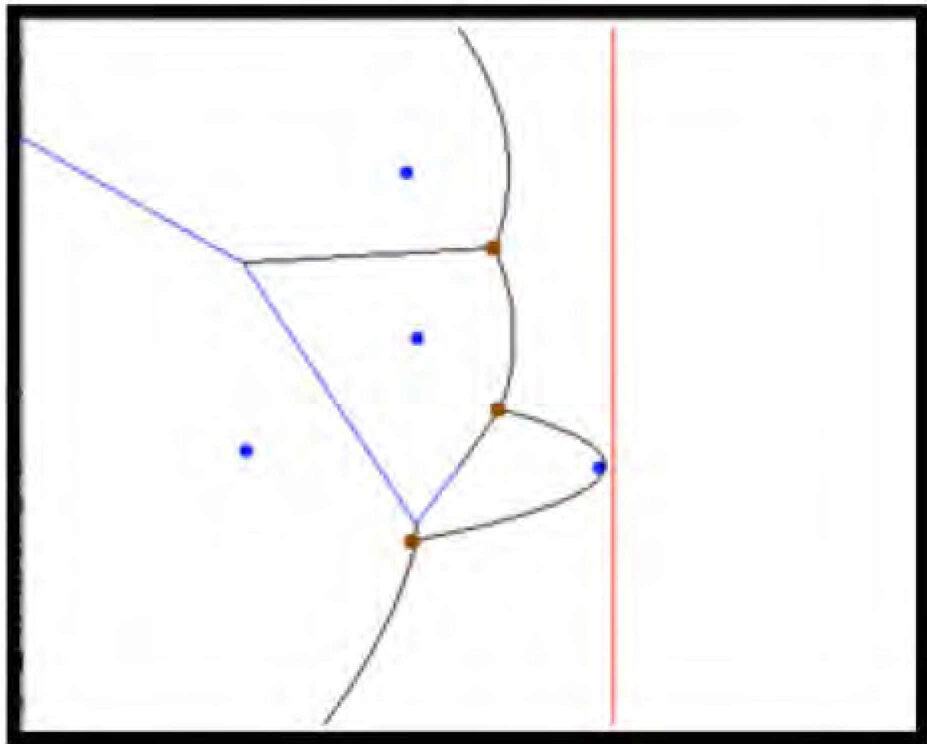


Figure 2.7: Illustration of Fortune's sweep-line algorithm. The red line is the sweep line, the black line is the beach line, the blue lines are the Voronoi edges and the blue dots are the seed points.

metrics can be used to define similarity. However, if the objects are more complex, the analyst will have to formally define the similarity based on the properties of the object. If the objects have large numbers of properties clustering may prove difficult, as many clustering methods are affected by the "curse of dimensionality".

In order to properly perform a cluster analysis, the analyst should have some prior knowledge about the data. The difficulty with clustering lies in choosing an appropriate clustering method, as well as optimising the parameter settings used in that method. Choosing the method and parameters will often involve many iterations of trial and error, with modifications to both the parameter settings and the method itself occurring at each iteration.

### 2.4.1 Clustering Algorithms

Although there are many different clustering methods and algorithms, two of the most common and widely used are discussed here: k-means and DBSCAN.

#### K-means Clustering

K-means clustering implements the idea of centroid-based clustering. The term was first used by MacQueen in 1967 [22], but the idea was first discussed in 1957 by Hugo Steinhaus. Centroid-based clustering methods represent a cluster with a central vector, which is not necessarily an object of the data set. Generally, the number of clusters $k$ must be known beforehand, and entered as a parameter. The problem can be approached as an optimisation problem; the $k$ cluster centres must be found such that the squared distances from a centre to all the objects in that cluster is minimised.

The most well-known k-means algorithm is Lloyd's algorithm [30]. Since the optimisation is NP-hard, Lloyd's algorithm takes an iterative approach. It starts by assigning the $k$ cluster centres randomly; the squared distances are then calculated and the cluster centres are shifted. This is repeated until the sum of the squared distances converges to some value. The iterative approach means that the solution will only be an approximation; the desired accuracy can be changed via some parameter. Furthermore, the algorithm can only converge to a local optimum, as opposed to a global optimum. The algorithm is therefore commonly run multiple times, and it is up to the analyst to choose the most likely clustering.

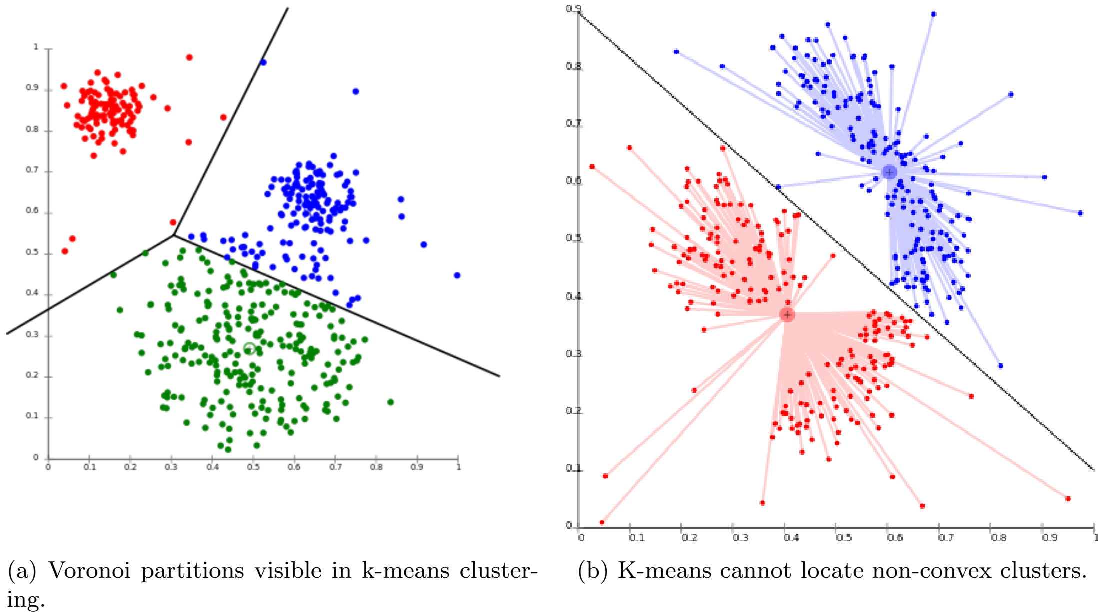
Without using Lloyd's algorithm, the problem can be solved exactly in $O(n^{dk+1}\log n)$ time, where $d$ is the dimension of the problem. With Lloyd's algorithm, however, the time is given as $O(nkdi)$, where $i$ is the number of iterations to convergence. Although the number of iterations is difficult to determine beforehand, the runtime is still linear in $n$ providing significant improvements over the exact solution.

Accurate k-means clustering requires clusters of similar size, as it assigns an object to the closest centroid. The method also clusters the entire data set, so there is no concept of noise, and there can be no erroneous data points. Interestingly, k-means clustering divides the space into a Voronoi diagram, which means that it can only distinguish convex clusters. It can also be seen as a variation of the Expectation-maximisation algorithm.

#### DBSCAN

DBSCAN (**D**ensity **B**ased **S**patial **C**lustering of **A**pplications with **N**oise) is a clustering method which uses the concept of density to distinguish clusters [39]. It defines a cluster as an area of higher data object density than its immediate surroundings. Two value are used to parametrise the clustering, namely $k \in \mathbb{Z}$ and $\epsilon \in \mathbb{R}$, where $k$ is the minimum number of objects in a cluster and $\epsilon$ is the maximum search distance. Together, these two values define a threshold density for clusters. Because a threshold density is defined, DBSCAN can detect outliers; they are objects which lie in regions with a lower object density than the threshold.

DBSCAN begins with a random object $\mathbf{p}$. All of the objects within a distance $\epsilon$ of $\mathbf{p}$ are determined, and this set is called the $\epsilon$-neighbourhood. If the $\epsilon$-neighbourhood has at least $k$ entries, $\mathbf{p}$ is labelled



(a) Voronoi partitions visible in k-means clustering.

(b) K-means cannot locate non-convex clusters.

Figure 2.8: K-means clustering of two data sets.

as a core point. If not, it is labelled as noise. Once labelled as noise, an object may later be found to be part of a cluster.

If $\mathbf{p}$ is found to be a core point, a new cluster is created, and each object within the $\epsilon$-neighbourhood of $\mathbf{p}$ is added to the cluster. After this, each object within a distance $\epsilon$ of an object in the cluster is added to that cluster. Once no new objects are found to be part of the cluster, an unvisited object is retrieved and the process begins again.

A brute force approach to a DBSCAN algorithm would yield a complexity of $O(n^2)$. This is due mainly to the neighbourhood query: each object must be compared to each other object to determine whether they are within a distance $\epsilon$ of each other. However, if an indexing structure is used to execute the neighbourhood query this can be optimised for an average runtime of $O(n \log n)$. An example of an indexing structure that can be used is a distance matrix. However, since the distance matrix is of size $(n^2 - n)/2$, the memory required is $O(n^2)$ as opposed to the $O(n)$ memory needed for an implementation without a distance matrix.

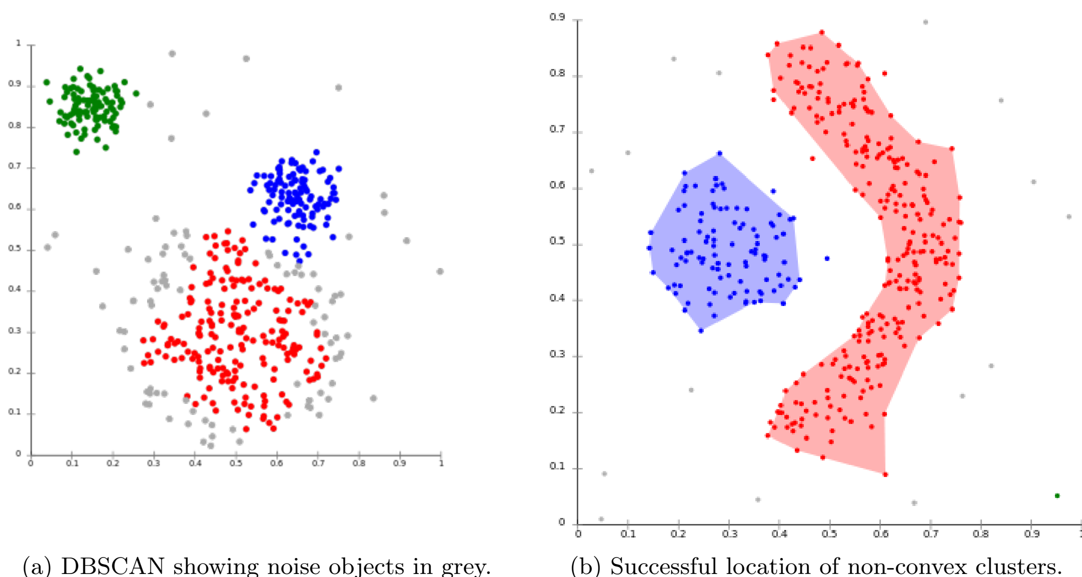
As opposed to k-means clustering, DBSCAN does not require any knowledge about the number of clusters in the data set. It can also find non-convex clusters, as well as clusters completely surrounded by other clusters. If the data is well understood, the $k$ and $\epsilon$ values can be somewhat easily determined. However, if the data is less well understood, a trial and error process is used to determine the parameters.

A disadvantage to DBSCAN is that it cannot cluster data sets in which there are large differences in the densities of the clusters. It is also highly affected by the curse of dimensionality, especially since it becomes difficult to choose an appropriate value for $\epsilon$ when using high-dimensional objects.

Although DBSCAN is mostly deterministic, there are borderline cases in which the ordering of the data objects can affect the final clustering. However, these cases are rare, and the effect will be small; the core points will be preserved regardless of ordering.

## 2.5 Multiple Target Tracking

Multiple target tracking (MTT) involves the tracking of multiple discrete targets, especially when the targets are indistinguishable from each other. The input data is often a set $S$ of points $\mathbf{p}_i = (\mathbf{x}, t)$,



(a) DBSCAN showing noise objects in grey. (b) Successful location of non-convex clusters.

Figure 2.9: DBSCAN clustering of two data sets.

where $\mathbf{x}$ is the spatial position of $\mathbf{p}$ and $t$ is the temporal position. The data is split up into time frames, with each frame $F(t)$ containing each point with the same time $t$. The desired output is a set of trajectories, with each trajectory describing the position-time history of a single target.

In general, the detections are noisy, with a possibility of false and missed detections occurring. Furthermore, the time intervals may not necessarily be constant. MTT algorithms should be able to track an unknown number of targets, and allow for targets appearing and disappearing (entering and exiting the field of view).

Figure 2.10 shows the basic elements for any multiple target tracking algorithm. The input data is a set of observations from one or more sensors. The observations are associated with existing tracks; this association method varies widely between algorithms. The next step is track maintenance: tracks are initiated, deleted and confirmed based upon some criteria. Finally the next positions for each track are predicted, and a search area is calculated for each track.

Three major methods are used for solving the multiple target tracking problem: probability-based, Monte Carlo-based and multiple hypothesis based methods. An example of a probability-based algorithm is the Joint Probabilistic Data Association Filter (JPDA) method [40]. The JPDA approximates the distributions of targets as Gaussian distributions. The result is that every observation is used to update each track, with weights being assigned to each observation based on the distance from the track in question.

The most popular method for solving MTT problems is the Multiple Hypothesis Tracking method developed by Blackman[41]. Where most methods perform the association step by finding the most likely configuration of track-observation associations for a given time frame, MHT takes a different approach. MHT creates multiple different hypotheses for the association; that is, it saves all possible track-observation pair configurations for each frame. This generates a tree with multiple configuration possibilities. The tree can then be traversed backwards to find the least likely configuration at a previous frame. This means that the MHT method uses multiple frames to find the correct associations. The probabilistic multiple hypothesis tracking (PMHT) model is a modification of MHT. PMHT reduces the complexity of MHT by assuming independent associations over the tracks.

A downside of the MHT algorithm is that with large numbers of targets, the number of possible hypotheses increases exponentially, leading to large trees that could take up exorbitant amounts of memory. This is dealt with by pruning the tree regularly; highly improbable associations are ignored

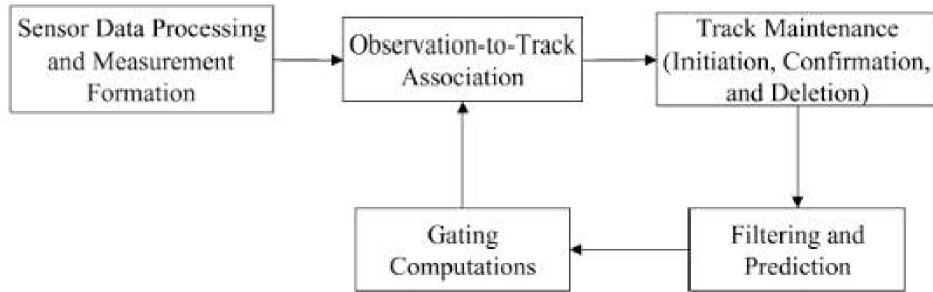



Figure 2.10: Basic elements of a multiple target tracking algorithm. Figure from [46]

or deleted. Some algorithms have been used which tackle the problem with large numbers of targets. Oh *et al.* [42] used a sampling method called the Markov Chain Monte Carlo (MCMC) association. The association is still based on past time frames, however it does not use an exhaustive tree as is found in MHT. The MCMC technique is in fact an approximation of an optimal Bayesian filter, and was shown to outperform MHT in extreme conditions. Extreme conditions may include large numbers of targets and high noise levels and false detection rates. Sarkka *et al.* [43] used Rao-Blackwellization [44] to further improve the efficiency of Monte Carlo tracking methods.

Many of the MTT algorithms make use of a Kalman filter [45] during the prediction step. A Kalman filter uses inaccurate or noisy measurements taken over time to estimate the state of unknown variables, or variables at some as yet undetermined time. Kalman filters are widely used, especially for automatic guidance for vehicles, but are also applied to time series analysis in signal processing and econometrics. The filter works by first estimating the current variables and their uncertainties based on previous data. It then uses one or more measurements to update the variables using a weighted average between the estimations and the measurements. The weights are based on the uncertainties in the measurements and estimations. Essentially, this allows for a prediction based on multiple previous observations without having to keep track of those observations.

# Chapter 3

# Description of Algorithm

The Voronoi-based Multiple Particle Tracking (VMPT) algorithm consists of two main parts: the first part locates the positions of the tracers at each time step, while the second part uses these locations to define position-time tracks.

## 3.1 Part 1 - Location

### 3.1.1 Input and Output

The output files from the PET camera are in list mode format. Each 8-byte word represents either a pair of physical detector cells in the camera which have detected an LOR; a time increment; or an invalid detection. The data is highly compressed to minimise the size of the list mode files, so the first step in using it is to decompress the files. When a word represents a detector cell pair, the word is transformed into a physical coordinate pair based on the dimensions of the camera. These coordinates are measured in mm. When a time increment is detected, the current timeframe is increased by one millisecond. Invalid detections occur when a cell detects a photon, but no other photons are detected within the open window. These invalid detections are discarded.

The output of the decompression is a $n_l$x7 matrix, which can be stored as a comma separated value (.csv) file. Here, $n_l$ is the total number of lines of response (LOR's) detected by the camera within the time interval specified. Each row of the matrix takes the form $[a_i, a_2, a_3, b_1, b_2, b_3, t]$, where $a_i$ are the $x, y$ and $z$ coordinates of the first detector cell, and $b_i$ are the coordinates of the second detector cell. The time at which the LOR was detected is given by $t$. The decompressed files are significantly larger than the compressed files, and for this reason the data is split across multiple files.

The code used for this decompression was not developed in this project. Rather, the code from the original Birmingham C-Track algorithm was used to generate the files containing the line of response and time data.

When the VMPT program initialises, the user is prompted for three sets of input:

- Input folder. This is the folder containing the files generated by the decompression algorithm. The files should be named sequentially for proper ordering.
- Number of tracers. This is the total number of tracers used in the experiment. This is used to initially define how many lines of response are used per frame of data, since more lines are needed to accurately locate larger numbers of tracers. The actual number of tracers detected by the algorithm may still vary.
- Output folder. The output file from the processing is written to this folder, as well as a log file.

The output of the location algorithm is a single .csv file. Each row of the file takes the form $[x, y, z, t]$ with $x, y, z$ being the spatial coordinates of a tracer in millimetres, and $t$ the time at which the tracer

was located, in milliseconds. This output file is used as input for the tracking algorithm described in section 3.2.

### 3.1.2 Description

The Voronoi Location algorithm starts by loading the parameters (as discussed in section 3.1.3) from a settings file. It then prompts the user for the name of the folder containing the input data, the number of tracers used in the experiment, and the output folder location. The first of the input files is loaded into memory, and stored as a matrix.

Once the initialization is complete, the algorithm enters the processing loop. This loop is continued until there is no more data to process, or the user forcefully quits the program. Within the loop, the next $n_l$ rows of the input matrix are extracted; these represent the LOR's under current consideration. This set of rows will be referred to as the active frame.

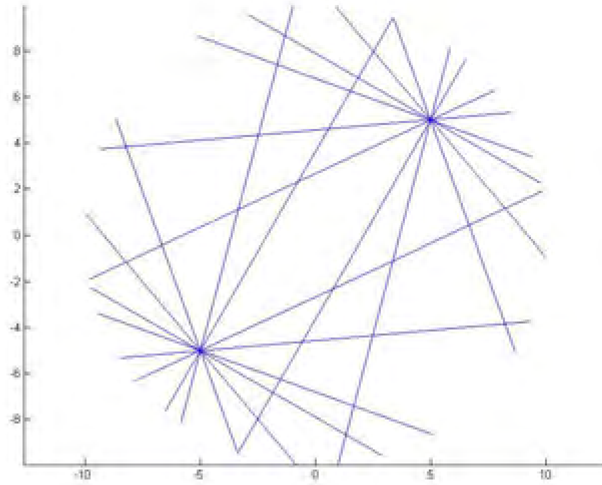



Figure 3.1: Plot showing the LOR's for two artificially generated tracers in 2D.

Each line of response (LOR) is now broken into multiple discrete points with a constant separation distance. Mathematically, let $\mathbf{l} = \{\mathbf{a}, \mathbf{b}\}$ where $\mathbf{a}$,$\mathbf{b}$ are 3-dimensional spatial vectors describing the two end points of the LOR, $\mathbf{l}$. The end points $\mathbf{a}$ and $\mathbf{b}$ are extracted from the input data as discussed in section 3.1.1.

Now a point on the LOR can be described by the equation

$$\mathbf{x} = \mathbf{a} + \lambda(\mathbf{b} - \mathbf{a}) \tag{3.1}$$

where $\mathbf{x}$ is the point along the line and $\lambda$ is a real number between 0 and 1.

Next, the number of points along the line (given a fixed distance between successive points) is calculated using

$$n_p = round\left(\frac{|\mathbf{a} - \mathbf{b}|}{\delta_s}\right) \tag{3.2}$$

where $\delta_s$ is the separation distance and $n_p$ is the number of points generated.

Finally, the set of discrete points along the line can be determined with

$$\mathbf{p}_i = \mathbf{a} + \frac{i\delta_s}{|\mathbf{a} - \mathbf{b}|}(\mathbf{b} - \mathbf{a}) \qquad i = \{1, 2, \ldots, n_p\} \tag{3.3}$$

with $\mathbf{p}_i$ being the $i^{th}$ point along the line.

It must be noted that since the number of points was rounded, the final point $\mathbf{p}_{n_p}$ will not coincide exactly with the end-point $\mathbf{b}$. This difference is negligible to the outcome of the algorithm. Another option would have been to discretize each line into an equal number of points, which would simplify the algorithm because it would not require the data structures used to keep track of corresponding lines and points. However, this would mean that lines of different length would generate points with different separation distances. It was found that fixed separation distances led to more reliable clustering and therefore better tracer location.

Once all of the lines in the set have been discretized, the points used as seeds in a Voronoi tessellation. The tessellation generates a polyhedron around each of the discrete points, defined by $n_v$ spatial vectors representing the vertices of the polyhedron. Because the number of vertices can vary from seed to seed, data structures are used to keep track of the vertices surrounding each seed point. After the tessellation the "volume" of each polyhedron is determined. To calculate the true volume, a convex hull algorithm would be used to divide the polyhedron into tetrahedrons. However, convex hull calculations are computationally expensive, and since the number of seed points is often of the order of 100 000, finding the true volume is highly inefficient. Instead, a new metric for volume was defined; it will hereafter be referred to as the mean vertex distance, or MVD. The MVD is simply the average distance from the centre of the polyhedron to each of its vertices. Formally,

$$\mathbf{c} = \frac{\sum_{i=1}^{n_v} \mathbf{x}_{v,i}}{n_v} \tag{3.4}$$

$$V_p = \frac{\sum_{i=1}^{n_v} |\mathbf{x}_{v,i} - \mathbf{c}|}{n_v} \tag{3.5}$$

Here, $\mathbf{c}$ is the centre of the polyhedron under consideration, $\mathbf{x}_{v,i}$ is the $i^{th}$ vertex of the polyhedron, and $V_p$ is the mean vertex distance.

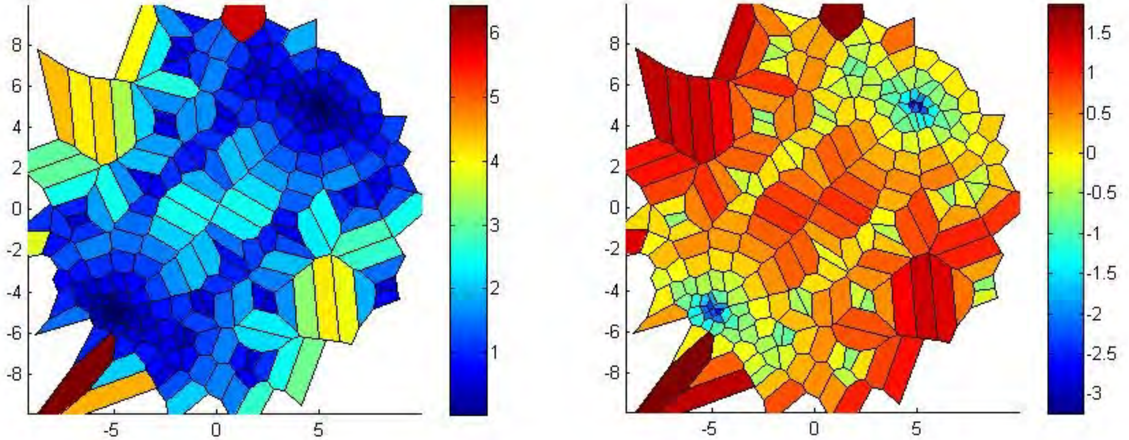
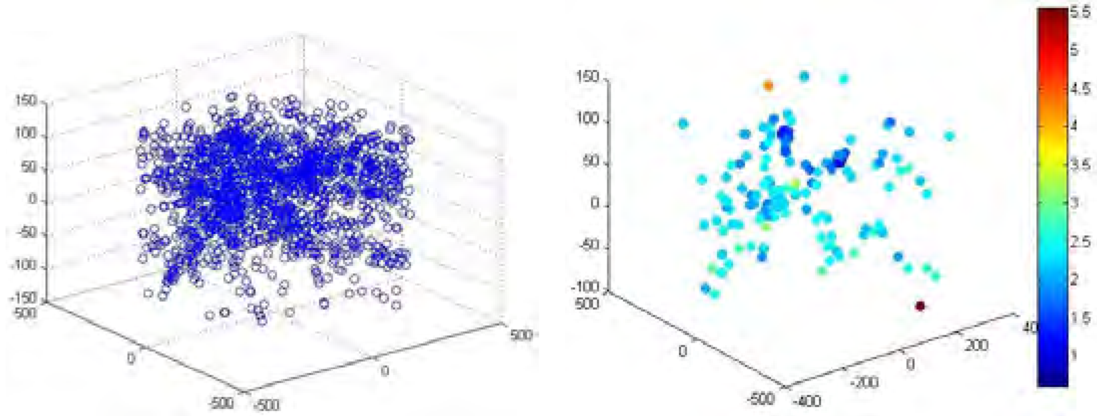



Figure 3.2: Voronoi diagram for seeds generated by two artificial tracers in 2D, with the colour of each polygon indicating the mean vertex distance of that polygon. The image on the right shows the logarithm of the volumes to more clearly indicate the differences.

It can be seen in figure 3.2 that the volumes of the polyhedra increase with the distance of the seed point from the tracer location. Therefore it can be assumed that the tracer location relative to some LOR is in the vicinity of the seed point on that line with the smallest volume (or mean vertex distance). Under this assumption, the seeds for each LOR are searched, and the point with the




(a) Discretization of the 200 LOR's. Five percent of the total 31 130 points are shown.

(b) Seed points along each LOR with the smallest surrounding polyhedra. The colours of the markers indicate the logarithm of the volume of the corresponding polyhedra.

Figure 3.3: Scatter plots of real data for a single frame in an experiment using two tracers. 100 lines were used per tracer, with a separation distance between points of 5mm, for a total of 31 130 discrete seed points

smallest surrounding volume per line is chosen and stored in a data structure. All other seed points are discarded. Once the smallest seeds have been found, the size of the data set decreases by multiple orders of magnitude, making further processing of the set less computationally expensive.

Ideally, the next step would be to use a clustering algorithm to group the remaining points, and thereby locate the tracers. However, Figure 3.3b shows that after the smallest volume per line has been found, the remaining points are noisy to a degree that makes it difficult for most clustering algorithms to sufficiently group the data points. To increase the reliability of the clustering, two data cleaning methods are used.

Firstly, the algorithm makes use of a local outlier factor [47], or LOF, which assigns a scalar real value to each data point. To calculate the LOF, some variables must first be defined: the k-distance($A$) of an object A to its $k$-th nearest neighbour. The set of these $k$ nearest neighbours to $A$ is denoted by $N_k(A)$. Now we define the *reachability distance* of an object $A$ from $B$ with

$$\text{reachability-distance}_k(A,B) = \max\left\{\text{k-distance}(B), d(A,B)\right\} \tag{3.6}$$

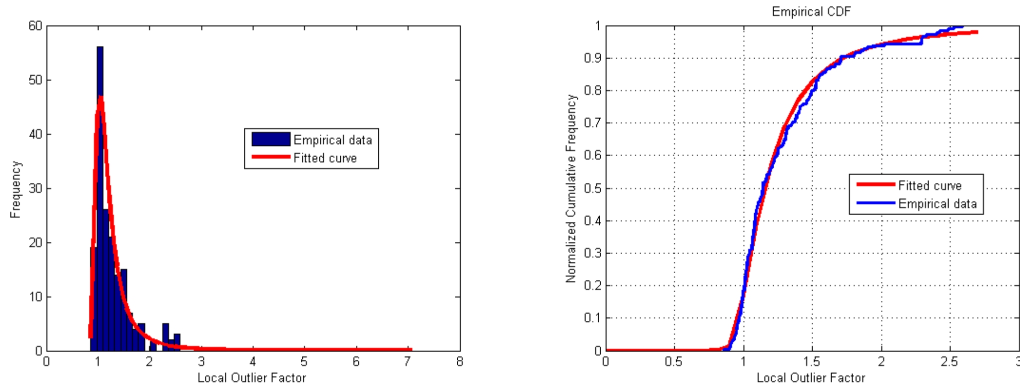
where $d(A,B)$ is the distance from object $A$ to object $B$. Note that any distance metric can be used to find the local outlier factor, although the most commonly used metric is the Euclidean distance. Using the reachability distance, a *local reachability density* of $A$ can be found using

$$\text{lrd}(A) := 1/\left(\frac{\sum_{B\in N_k(A)} \text{reachability-distance}_k(A,B)}{|N_k(A)|}\right) \tag{3.7}$$

Intuitively, the local reachability density is the inverse of the average reachability distance of $A$ from its neighbours. Finally, the LOF of object $A$ can be found with

$$\text{LOF}_k(A) := \frac{\sum_{B\in N_k(A)} \frac{\text{lrd}(B)}{\text{lrd}(A)}}{|N_k(A)|} \tag{3.8}$$

The local outlier factor algorithm was first published in 2000 by Breunig *et al.* [47]; it is used to find outliers in data sets with clusters of varying densities. The scalar LOF value assigned to a point is a measure of the "outlier-ness" of that data point, and does not directly indicate whether the point




(a) Histogram of LOF values, with a fitted generalised extreme value probability density function.

(b) Corresponding normalized cumulative frequency plots.

Figure 3.4: Distributions of local outlier factors for real data using two tracers. Far outliers (with LOF values greater than the mean plus twice the standard deviation) are excluded from this data.

is classed as an outlier. Some attempts have been made to correctly determine whether an LOF score indicates an outlier: Kriegel *et al.* [48] used a probabilistic method in which the outlier score took a value between zero and one, inclusive, which is the absolute probability that a point is an outlier. In 2011, the same authors investigated a way to unify outlier scores from various outlier detection methods [49], including the LOF method. Once again, the result was a probability score between zero and one.

As can be seen in Figure 3.3b, the density of points varies widely within the field of view. With more tracers, and especially when some of the tracers are close to the edge of the field of view, this density difference is exacerbated, and different tracers may have different densities of corresponding data points. This means that typical density measures for outliers are not sufficient. The LOF is used because it makes use of the *local*, rather than global, density.

Because the LOF does not determine whether a point is an outlier or not, it is up to the user to decide which values describe outliers. Rather than using the methods designed by Kriegel *et al.* [48] [49], a custom method was used in which the probability density function (PDF) of the set of LOF scores was examined, and a percentage cutoff based on the cumulative density was defined. After fitting multiple different PDFs to the frequency plots, it was found that a generalised extreme value PDF fits best. A generalised extreme value PDF is, in general, a distribution of the normalised maxima of a sequence. It therefore makes sense that a set of LOF's would follow this distribution, since the LOF makes use of the *reachability distance* of the data points, which is defined as a maximum of a set of values.

Observing Figure 3.4b, the default cutoff percentage used in the algorithm is 65%, that is, LOF values corresponding to a cumulative frequency value of less than or equal to 65% are seen as useful data points, and all those outside this range are defined as outliers. The 65% value is somewhat arbitrary; it was arrived at through trial and error. If it does not give reasonable results, this value may be changed by the user according to their needs.

The LOF cleaning sufficiently separates clusters which are close enough to start merging. For instance, when two tracers come close to each other, the clusters of data points surrounding each will begin to merge. Because the LOF is based on local density, it isolates merged clusters. However, the LOF cleaning process fails to detect points around the edges as outliers. This is because all points near the edges of the field of view have similar local densities, even though the density values themselves may be low. To further remove outliers from the data, the volumes of the Voronoi polyhedra surrounding the remaining points can be examined. It can be seen in Figure 3.5 that the volumes are arranged in a bimodal distribution. Unfortunately, it is not immediately clear what type of distributions the two modes represent, making analysis of the data difficult. Instead, a solution was found through trial and

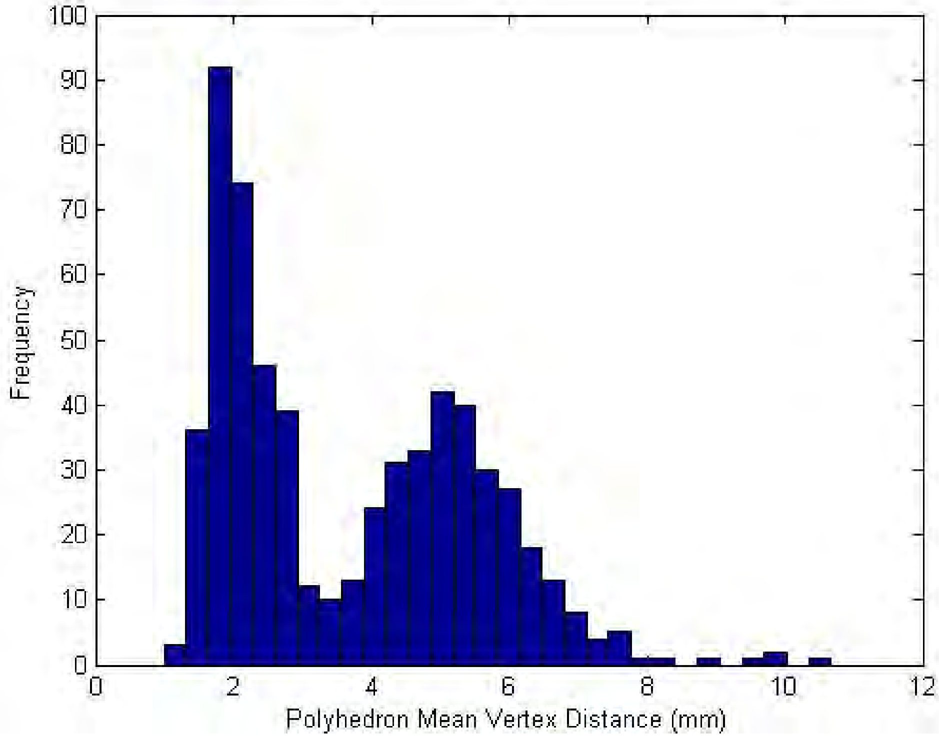



Figure 3.5: Histogram showing the Mean Vertex Distances of polyhedra surrounding the remaining seed points after LOF cleaning.

error. From the set of points still remaining after the LOF cleaning, those which have a volume of greater than the mean plus twice the standard deviation are discarded. This removes the far outliers which skew the data significantly. After this, a new mean and standard deviation are determined, and points with a volume larger than the mean plus 0.2 times the standard deviation are discarded. Although this method was found through trial and error, it has shown to give consistent results.

After the two filtering methods have been used, the remaining points must be clustered. The DBSCAN algorithm discussed in 2.4.1 was used for clustering purposes. The reason for this is twofold:

- the data will still have some erroneous entries (noise) and DBSCAN is well equipped to cluster data with low levels of noise.
- DBSCAN is a density-based clustering method, so a minimum density threshold for clusters can be defined.

As discussed, the input parameters to DBSCAN (besides the data points themselves) are $k \in \mathbb{Z}$ and $\epsilon \in \mathbb{R}$. Together, these two values define the minimum density threshold needed for a data point to belong to a cluster. The value for $k$ is the same as that used by the LOF filtering algorithm, where it defines the number of nearest neighbours used in calculating the reachability distance. The value of $\epsilon$ is the same as the separation distance $\delta_s$. The separation distance is on the order of the radius of the tracer, so that there is a high chance of a discretized point on an LOR lying within the physical bounds of the tracer, but a low chance of multiple points on a single LOR lying within the tracer. Similarly, $\epsilon$ is of the order of the radius so that points from multiple LOR's will be within a radius of each other, given that they belong to the same cluster.

### 3.1.3 Parameters

There are a number of parameters, both natural and artificial, that affect the performance and efficiency of the location algorithm. Except for the number of tracers, these parameters are stored in an .ini text file which can be easily edited. This file is loaded by the program at start up.

#### Number of Tracers

Denoted by $n_t$, this is simply the number of tracers used in the experiment. Note that although the actual number of tracers within the field of view of the camera may change, this value remains constant throughout the runtime of the algorithm.

#### Number of Lines of Response per Tracer

As the number of tracers used increase, the number of lines needed to successfully locate the tracers also increases. Taking this into account, the total number of lines is calculated as a multiple of the number of tracers used, and this multiple is denoted by $n_l$. The number of lines selected per frame is therefore $n_{lt} \times n_t$.

Increasing $n_l$ increases the accuracy and reliability of the triangulation. However, as the number of lines increases, the processing time increases with $n_l^{\frac{3}{2}}$ ; this is discussed further in Section 5.5. The tessellation scheme is already computationally expensive, so a compromise must be found between accuracy and processing time. In most of the experiments, a value of $n_{lt} = 100$ was used.

#### Separation distance

As discussed in 3.1.2, the separation distance $\delta_s$ is the distance between consecutive points along a discretized line, measured in millimetres. This value is similar to $n_{lt}$ in that decreasing the separation distance increases the accuracy as well as the processing runtime. Once again, a compromise must be found. However, unlike $n_{lt}$, the separation distance has a lower bound, after which the accuracy decreases quickly. Again, this is discussed in Section 5.5. It was found that a separation distance on the order of the radius of the tracers performed close to optimally, i.e. $\delta_s = 5$.

#### Minimum Points for Clusters

This value, denoted by $k$, is the number which specifies how many data points qualify as a cluster. It is used in both the DBSCAN and LOF algorithms to find the k-nearest neighbours for each point. The final value settled on was $k = 4$. The reasoning behind the selection of $k$ is discussed in Section 5.2.1, and is based on the fact that tracers towards the edges of the field of view of the scanner generate fewer photon pairs that are likely to be detected by the scanner.

### 3.1.4 Parallelization

Performing a Voronoi tessellation with a large number of seed points is computationally expensive (at best $O(n\log n$ complexity); in fact, the tessellation takes up about 99% of the total processing time. When tracking large numbers of tracers, the processing time can be so long that it makes the technique impractical. However, the algorithm has been designed so that each frame can be processed independently of every other frame. On a machine with multiple cores, the program can be altered so that each core works simultaneously on a separate frame; once each core has finished processing the data, the locations are combined and added to the output matrix.

Unfortunately, the author did not have access to the parallel processing Matlab toolkit. The solution to this was to use C++ to write the code with a low-level language. C++ has built-in parallel computing capabilities, but a third-party library was used to perform the Voronoi tessellation. *Voro++* is a 3D Voronoi tessellation library developed by Chris Rycroft [50].

The Voro++ library constructs tessellations cell-by-cell; that is, each data point is added one after the other and the tessellation updates with each insertion. It is robust, has relatively high performance, and can perform tessellations on large numbers of particles. However, it is not easily parallelizable. The library uses some static members, which leads to variables in memory that must be shared across cores. This means that running the program on multiple cores actually slows down the processing speed, since each core has to wait in order to access the shared memory.

A possible final solution to the parallel problem is to use the Python scripting language. The scientific and mathematical libraries developed for Python use the same Voronoi tessellation algorithms as Matlab (namely Qhull), but unlike Matlab, Python is free to use. Since Python is a scripting language, some of the methods may be slow. In order to speed up processing, these methods can be written in C/C++ and linked to the Python script.

### 3.1.5 Pseudocode

**Algorithm 1** Location Algorithm - Main

```
 1: procedure Location
 2:     lmInput ← load data from lm file
 3:     while lmInput has more entries do
 4:         frame ← extract next N_l entries from lmInput
 5:         frameTime ← mean(frame(:,7))
 6:         lines ← frame(:,1:6)
 7:         seeds ← discretize all entries in lines
 8:         voronoiCells ← perform Voronoi tessellation on seeds
 9:         for each Line iLine in lines do
10:             smallSeeds(i) ← find seeds entry along iLine with smallest corresponding Voronoi cell
11:             increment i by 1
12:         end for
13:         lof ← calculate LOF for smallSeeds
14:         fit GEV cumulative density function to lof
15:         lofCutoff ← LOF value corresponding to 65% cutoff in CDF
16:         lofSeeds ← empty array
17:         for seedNum := 1 to length(smallSeeds) do
18:             if lof(seedNum) < lofCutoff then
19:                 append smallSeeds(seedNum) to lofSeeds
20:             end if
21:         end for
22:         volumes ← volumes Voronoi cells of remaining seed points
23:         volMean ← meanvolumes
24:         volStd ← standard-deviation(volumes)
25:         volSeeds ← empty array
26:         for seedNum := 1 to length(lofSeeds) do
27:             if volumes(seedNum) < (volMean + 0.2volStd) then
28:                 append lofSeeds(seedNum) to volSeeds
29:             end if
30:         end for
31:         clusters ← apply DBSCAN algorithm to volSeeds
32:         locations ← empty array
33:         for clusterNum := 1 to max(clusters) do
34:             p ← [0, 0, 0]
35:             nEntries ← 0
36:             for seedNum := 1 to length(volSeeds) do
37:                 if clusters(seedNum) == clusterNum then
38:                     p+ = volSeeds(seedNum)
39:                     nEntries+ = 1
40:                 end if
41:             end for
42:             x ← [p/nEntries , frameTime]
43:             append x to locations
44:         end for
45:         write locations to file
46:     end while
47: end procedure
```

## 3.2 Part 2 - Tracking

### 3.2.1 Input and Output

The input data to the tracking algorithm is the cluster file generated by the location algorithm. The user is prompted for the file path of this file, as well as the output folder to which the tracks will be written.

The output of the tracking algorithm is a series of track files. Each track file represents the trajectory of a single tracer, and the data is stored in an $n_e \times 4$ matrix, with $n_e$ being the number of entries in the trajectory. Each row of a track file takes the form $[x, y, z, t]$, where $[x, y, z]$ is the location of the tracer and $t$ is the time at which that location occurred.

The output data is raw, in that no smoothing or post processing is performed. The locations in the track files correspond to those in the input file.

### 3.2.2 Description

The tracking algorithm uses the output from the location algorithm to determine the paths taken by the tracers. It starts by asking the user for two sets of input: the path of the output file generated by the location algorithm, and the path of the folder to which the tracks will be written. If the output folder does not exist, it will be created.

Next, the rows of the input matrix are sorted by time. This sorting will cause the rows in the matrix to be grouped according to their times; these groups will hereafter be referred to as time frames. Each group should have a number of entries (rows) roughly equal to the number of tracers used in the experiment. A group may have fewer entries if a tracer exited the field of view, or if the algorithm was unable to locate a tracer. Similarly, a group may have more entries in the event of a false location.

After the sorting stage, the tracking begins. The first time frame is extracted, and each entry (location) is assigned to the beginning of a new track object. A track object is one representing the path of a single tracer. It stores all of the entries in the track, as well as multiple methods, discussed below. After initialising the tracks, the next time frame is loaded.

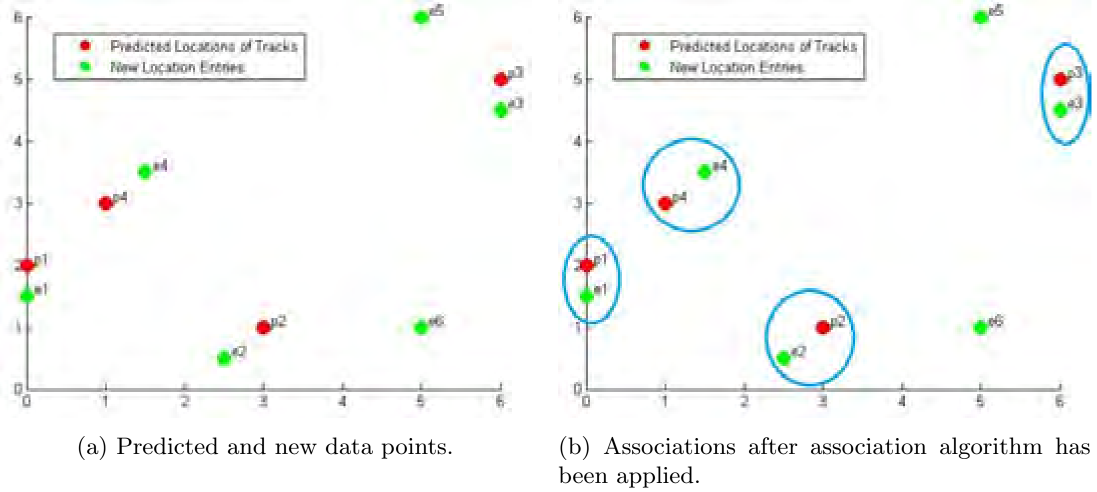
The next step is to predict the next position of each of the currently existing tracks. For most entries, this prediction uses the last $n_{pp}$ entries in the track. The value of $n_{pp}$ can vary, as discussed in Section 3.2.3, and is the total number of entries in the track, but at most a maximum value defined by the user. Using these values, a line of best fit is found for each of $xvst$, $yvst$, and $zvst$. The best fit line may be linear or quadratic, depending on the expected time increments. For instances where the time increments are small compared to the acceleration a linear fit is recommended, since it will be more robust to noise. If a quadratic fit is used, a larger value of $n_{pp}$ is recommended to more robustly determine the fit. In general, a linear fit can be reliably used if the time interval is small enough such that the motion is approximately linear over that interval. Using the equations of these lines of best fit, the locations at the time of the newly-loaded time frame can be found. For the first two entries with a linear fit and the first three entries with a quadratic, this prediction is not possible. Instead, for the first $n_{pp}$ entries, the entry itself is used as the predicted location. Later in the algorithm, adjustments are made to accommodate for this.

Now, the predicted values are matched to the entries in the loaded time frame. The first step in this association is creating a distance matrix $\mathbf{D}$. The entries in this matrix are given by

$$D_{ij} = |\mathbf{p}_i - \mathbf{e}_j|_2 \tag{3.9}$$

where $\mathbf{p}_i$ is the predicted value of the $i^{th}$ existing track, and $\mathbf{e}_j$ is the $j^{th}$ entry of the new time frame.

Once the distance matrix is defined, an association matrix $\mathbf{A}$ can be generated. An entry $A_{ij}$ in the association matrix is 1 if the corresponding entry $D_{ij}$ in the distance matrix is the minimum value in the row $i$ and the minimum in the column $j$; otherwise it takes the value 0. Mathematically, this can be shown with




(a) Predicted and new data points.

(b) Associations after association algorithm has been applied.

Figure 3.6: Scatter plots illustrating the association matrix for arbitrary 2-dimensional points.

$$A_{ij} = \begin{cases} 1, & \text{if } D_{ij} = \min(D_{i,*}) \ \wedge \ D_{ij} = \min(D_{*,j}) \\ 0, & \text{otherwise} \end{cases} \tag{3.10}$$

Now if $A_{ij} = 1$, the entry $\mathbf{e}_j$ is associated with the $i^{th}$ track. Note that it is not sufficient to simply find the closest point to each existing track. In the case where there are fewer new entries than existing tracks, this simplistic approach would lead to an entry being associated with multiple tracks. The double-minimum check is needed to ensure uniqueness.

This association step can be illustrated with simple 2-dimensional examples, shown in Figures 3.6 and 3.7. The distance matrix for these data points is calculated as

$$\mathbf{D} = \begin{bmatrix} 0.5 & 2.9 & 6.5 & 2.1 & 6.4 & 5.1 \\ 3.0 & 0.7 & 4.6 & 2.9 & 5.4 & 2.0 \\ 6.9 & 5.7 & 0.5 & 4.7 & 1.4 & 4.1 \\ 1.8 & 2.9 & 5.2 & 0.7 & 5.0 & 4.5 \end{bmatrix}$$

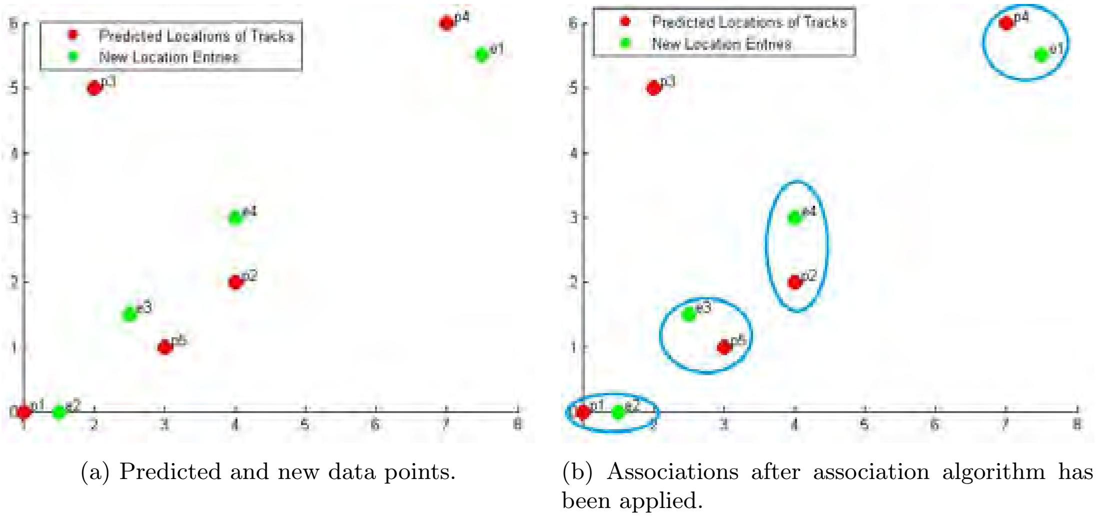
and, applying equation 3.10, we find the association matrix takes the values

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

It can be clearly seen from $\mathbf{A}$ which entry is associated with each track. A more interesting example to observe is one in which there are fewer detected entries than there are existing tracks. Figure 3.7 shows a case with five tracks but only four new detections. This example is also slightly more complicated since corresponding numbers do not necessarily match up.

The corresponding distance and association matrices for Figure 3.7 are

$$\mathbf{D} = \begin{bmatrix} 8.5 & 0.5 & 2.1 & 4.2 \\ 4.9 & 3.2 & 1.6 & 1.0 \\ 5.5 & 5.0 & 3.5 & 2.8 \\ 0.7 & 8.1 & 6.4 & 4.2 \\ 6.4 & 1.8 & 0.7 & 2.2 \end{bmatrix}, \ \mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$




(a) Predicted and new data points.

(b) Associations after association algorithm has been applied.

Figure 3.7: Scatter plots illustrating the association matrix for arbitrary 2-dimensional points, with fewer new entries than existing tracks.

and matrix $\mathbf{A}$ shows that track number 3 does not have an associated entry. If the simplistic case was used, wherein the closest point to a track prediction is used, we would have

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

which associates entry $\mathbf{e}_4$ with tracks $\mathbf{p}_3$ and $\mathbf{p}_2$. Judging the results by eye, it is easy to see that this is incorrect.

The final step in the association is to validate whether the matches make sense based on the kinematic properties of the system. To do this, a value called the gate distance is calculated. For a valid association, the new entry must be within a gate distance from the predicted location. The gate distance changes with the kinematics of the system; specifically, it is a function of both the velocity and the acceleration of the associated track. To calculate the gate distance, some definitions are required. First, the velocity of the most recent point in the track is required. Since the data is noisy, the velocity must be the average over the $n_{pp}$ most recent points in the track. The set of these points for a track will be referred to as $\mathbf{r}_j$, with $j = 1, \ldots, n_{pp}$ Now we define the velocity for point $j$ as

$$\mathbf{v}_j = \frac{\mathbf{r}_j - \mathbf{r}_{j-1}}{t_j - t_{j-1}} \tag{3.11}$$

where $t_j$ is the time corresponding to $\mathbf{r}_j$. Remember that $n_{pp}$ is defined as the number of entries in the track used for prediction and $n_{pp,max}$ is the maximum number of entries to be used for prediction. So for instances where $n_{pp} \leq n_{pp,max}$, the first entry $\mathbf{r}_1$ in the track will not have a velocity assigned this way. Instead, $\mathbf{v}_1$ is assigned a value of zero.

Once the individual velocities have been calculated, it is trivial to find the average velocity $\bar{\mathbf{v}}_j$ for point $\mathbf{r}_j$ with

$$\bar{\mathbf{v}}_j = \left( \sum_{j=1}^{n_{pp}} \mathbf{v}_j \right) / n_{pp} \tag{3.12}$$

We now define the *gate distance*, denoted by $\delta_g$. If a location is associated with a track via the association matrix, and the location falls within a gate distance of that track, then that location is

appended to the list of entries in the track. Through trial and error, it was found that a gate distance of 20mm was optimal, and sufficed in most cases. However, when a tracer experiences high accelerations, this value may not suffice. The most basic equation of kinematic motion tells us that

$$\mathbf{x}_f = \mathbf{x}_i + \mathbf{v}\Delta t + \frac{1}{2}\mathbf{a}(\Delta t)^2. \tag{3.13}$$

Since a linear prediction method is used, and acceleration of zero is assumed, and the tracer's motion is modelled according to

$$\mathbf{x}_f = \mathbf{x}_i + \mathbf{v}\Delta t. \tag{3.14}$$

It can be seen that the term $\frac{1}{2}\mathbf{a}(\Delta t)^2$ describes the error in the prediction (assuming further time derivatives have less influence than acceleration). The gate distance describes the maximum error allowed during association, or more formally

$$\delta_g \geq \frac{1}{2}|\mathbf{a}|(\Delta t)^2 \tag{3.15}$$

$$|\mathbf{a}| \leq \frac{2\delta_g}{(\Delta t)^2}. \tag{3.16}$$

In words, if the magnitude of the acceleration vector is larger than twice the gate distance divided by the time interval squared, the association will fail. In order to more reliably track accelerating tracers, either the gate distance must be increased or the time interval must be decreased. This is discussed further in Section 5.2.

When a new entry is added in this way to a track, all the velocity and acceleration values are updated in the track object. If a track has no associated location, a null entry is added and the number of consecutive entry skips for that track is incremented by one (see section 3.2.3). The null entry is necessary for calculating the track density. If a location is not associated with any track, a new track object is created with that location as the initial entry. This new track is added to the array of existing tracks.

Finally all of the existing tracks must be checked to determine whether they have come to an end. The number of consecutive skips is inspected for each track; if it exceeds the maximum allowable value, the track is marked as closed. Once the closed tracks have been determined, they must either be saved to disk or discarded. A track is discarded if:

- it has fewer than the minimum number of track entries $n_{e,min}$, or;
- it has a track density less than the minimum track density $\rho_{t,min}$.

Otherwise it is saved to disk. (For more information on $\rho_{t,min}$ and $n_{e,min}$, see section 3.2.3.)

Once the relevant tracks have been saved and discarded the next set of locations is loaded and the process is repeated until there are no new entries.

### 3.2.3 Parameters

#### Maximum Number of Predictive Points

This number is an integer given by $n_{pp,max}$, with a default maximum value of 10. It defines the maximum number of points used to predict the next track location. If the total number of entries in the track is less than $n_{pp,max}$, then $n_{pp}$ is simply the total number of entries in the track. A larger $n_{pp,max}$ should provide a more accurate result, under the assumption that the motion of the tracer is purely linear (for a linear fit) or quadratic (for a quadratic fit). However, this is very rarely the case in practice, since the acceleration will most likely not be constant.

### Minimum Track Entries

The minimum number of track entries is the minimum number of valid entries in the track for it to be saved to disk, and is given by $n_{e,min}$, with a default value of 50. However, in most cases it may be more convenient to define a minimum time interval over which the track must exist to be considered valid. The settings can be easily changed to allow for this.

### Search Gate Distance

To be associated with a track, an entry must be within the search gate distance $\delta_g$ of a that track. The default value for the gate distance is 20mm; this can be increased if large accelerations are expected.

### Maximum Number of Consecutive of Skips

This integer $n_{s,max}$ defines the number of consecutive null (empty) entries in the track, and has a default value of 15. Once this number is reached, the track is terminated. From there, it is either discarded (based on the minimum track entries and minimum density) or saved to disk.

### Minimum Track Density

The minimum track density, given by $\rho_{t,min}$ is the ratio of non-empty entries in a track to the total number of entries. Tracks with densities lower than the specified value are discarded on termination. The default value is $\rho_{t,min} = 0.7$.

### 3.2.4 Pseudocode

**Algorithm 2** Tracking Algorithm - Main

1: **procedure** TRACKING
2:     *inputData* ← load output from location algorithm
3:     sort rows of inputData according to 4th column (time)
4:     *tracks* ← empty array of Track objects
5:     *outputData* ← empty array of Track objects
6:     **for** each unique entry $t \in \mathbb{N}$ in 4th column of inputData **do**
7:         *frame* ← all rows $\mathbf{r} \in \mathbb{R}^4$ with $\mathbf{r}(4) = t$
8:         *locations* ← *frame*(:,1:3)
9:         **for** each Track *iTrack* in tracks **do**
10:             predict next position of *iTrack*
11:         **end for**
12:         associate *tracks* predictions with *locations*
13:         **for** each row $\mathbf{x} \in \mathbb{R}^3$ in *locations* with an associated track *aTrack* **do**
14:             append $\mathbf{x}$ at time $t$ to *aTrack*
15:             increment *aTrack*(numEntries) by 1
16:         **end for**
17:         **for** each track with no associated location *jTrack* **do**
18:             append NULL entry to *jTrack*
19:             increment *jTrack*(numEntries) by 1
20:             increment *jTrack*(numSkips) by 1
21:         **end for**
22:         **for** each location with no associated track *kLocation* **do**
23:             create new Track object with initial entry *kLocation* and initial time $t$
24:             append new Track object to *tracks*
25:         **end for**
26:         **for** each Track *lTrack* in *tracks* **do**
27:             **if** *lTrack*(numSkips) $> n_{s,max}$ **then**
28:                 **if** *lTrack*(trackDensity) $< \rho_{t,min}$ OR *lTrack*(numEntries) $< n_{e,min}$ **then**
29:                     delete *lTrack* from *tracks*
30:                 **else**
31:                     append *lTrack* to *outputData*
32:                     delete *lTrack* from *tracks*
33:                 **end if**
34:             **end if**
35:         **end for**
36:     **end for**
37:     **for** each Track *iOut* in *outputData* **do**
38:         write *iOut*(trackHistory) to a new file
39:     **end for**
40: **end procedure**

**Algorithm 3** Track Class

```
1:  class Track
2:      properties
3:          numEntries ← 0
4:          numSkips ← 0
5:          trackHistory ← matrix with rows [x, y, z, t]
6:      end properties
7:      methods
8:          getPredictedLocation()
9:          addEntry()
10:         addSkip()
11:         getTrackDensity()
12:     end methods
13: end class
```

# Chapter 4

# Experimental Setup

The experiments used to gather PET data to test the VMPT algorithm were performed at iThemba LABS in Cape Town, South Africa. This author designed the experiments and created the experimental matrices, and assisted Mike van Heerden in setting up the equipment. For the majority of the tests, Mike van Heerden or Liu Cong ran the scanner, while this author measured the rotational velocity of the rotating shaft using a digital tachometer. For the tests using high tracer counts, Mr. van Heerden measured the velocity, since he is a registered radiation worker.

## 4.1 Apparatus

### 4.1.1 PET Scanner

Figure 4.1: CTI/Siemens EXACT3D model 966 PET scanner.

The PET scanner used in the following experiments was a CTI/Siemens 966 model, dubbed the EXACT3D scanner. The scanner was designed with the goal of achieving the highest possible resolution and sensitivity using the technology available at the time. There are a total of 48 rings along the length of the scanner, each made up of 72 crystal blocks. Each block contains 64 detector cells ($8 \times 8$), for 576 detector cells per ring and a total of 27648 in the scanner. Each of these detector blocks is mounted on

four photomultiplier tubes, which multiply the effect of the scintillation during detection. The position of an event is determined to be the weighted average of the detected scintillations on that block.

The ring diameter of the scanner is 830 mm, with each detector cell measuring $4.39 \times 4.05 \times 30$ mm, and the axial depth is 234 mm. The scanner has a spatial resolution of $4.8 \pm 0.2$ mm transaxially, and $5.6 \pm 0.5$ mm axially. The maximum detection rate allowed by the I/O hardware is about four million detections per second. This is restricted by the data transfer rate from the detectors to memory, at 17 MB/s. To provide some added visibility during experiments the scanner was lined with multi-colour LED strips, and for extra precision a laser alignment system was used.

The EXACT3D scanner is able to generate output in both list-mode and sinogram forms. For the experiments discussed here, only list-mode files were used.

### 4.1.2 Tracers

A total of twenty tracers were used throughout the experiments. They were manufactured at iThemba LABS in Cape Town, South Africa by Mr. van Heerden. The tracers were made by drilling a 1 mm diameter hole halfway through a 10 mm radius glass bead. A small amount of sodium-22 ($^{22}$Na) radioisotope was then inserted into the hole, and resin was used to fill it in. To make the tracers easier to handle, a 20 mm length of clear, flexible 8 mm inside-diameter tubing was cut for each tracer. Each tracer was then inserted into the end of a piece of tubing by warming the end of the tube until it was soft enough to easily expand.

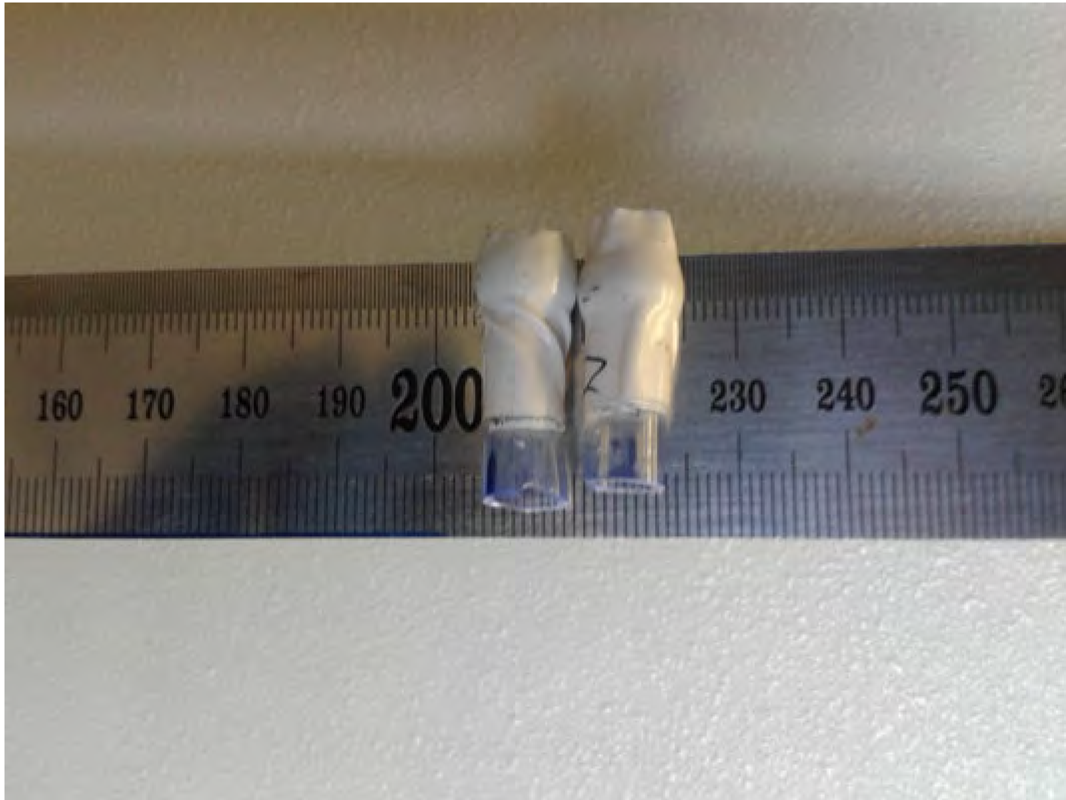



Figure 4.2: Two tracers inside the plastic tubing. White insulation tape was used to provide extra padding for a more snug fit inside the rig. Ruler for scale.

Sodium-22 was chosen as the tracer material because it has a half life of 2.6 years; this longer half-life makes it usable over a longer period of time. It can be used for weeks, and possibly months, while still providing sufficient activity for tracking. The other commonly used radioisotopes in PEPT

are Fluorine-18 and Gallium-68, which have half-lives of 110 and 68 minutes respectively. If $^{18}F$ or $^{68}Ga$ were to be used, new tracers would have to be manufactured for each day of experiments. The tracers were also manufactured to have lower activities, or decay rates, than usual. This is because PET scanners have a saturation limit, which is the maximum number of LOR's that the scanner can detect per second. A higher tracer count means that there is a higher overall activity within the field of view of the scanner, and the activity is directly proportional to the number of lines of response generated per second. Near saturation levels of activity, the scanner starts to detect *fewer* LOR's per tracer because some will be lost. Using tracers of lower activity means that a larger number of tracers can be used before reaching this saturation limit.

A total of twenty tracers were used; the algorithm may be able to track more tracers, but iThemba LABS only had sufficient amounts of $^{22}Na$ to make twenty.

Table 4.1: Activities of tracers used, measured before the experiments were performed.

| **Tracer Number** | **Activity (µCi)** |
|---:|---:|
| 1 | 72 |
| 2 | 72 |
| 3 | 66 |
| 4 | 63 |
| 5 | 63 |
| 6 | 62 |
| 7 | 62 |
| 8 | 59 |
| 9 | 59 |
| 10 | 58 |
| 11 | 58 |
| 12 | 54 |
| 13 | 85 |
| 14 | 83 |
| 15 | 66 |
| 16 | 65 |
| 17 | 58 |
| 18 | 49 |
| 19 | 74 |
| 20 | 60 |

Table 4.1 shows the activities of all of the tracers used in the experiments. The activity readings were taken before the experiments began, shortly after they were manufactured. These same twenty tracers were used over the course of the experiments, which lasted four weeks.

$$A(t) = A_0 e^{-\lambda t} \tag{4.1}$$

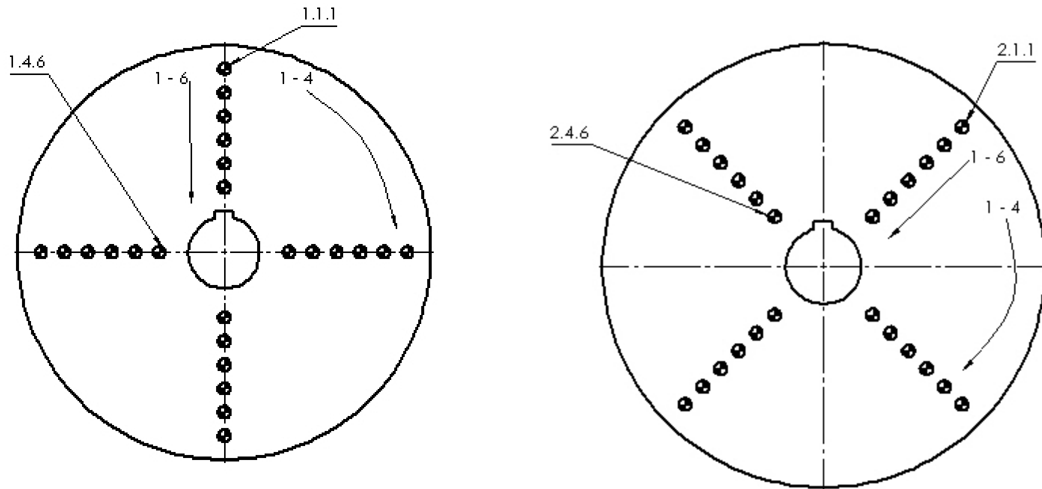
$$A(t) = A_0 2^{-t/T_{1/2}} \tag{4.2}$$

Equation 4.2 can be used to find the activity $A(t)$ of a quantity of radioactive isotope after a time $t$, given the initial activity $A_0$, which is the activity at time $t = 0$; and the half-life of the isotope $T_{1/2}$. Since the half-life of $^{22}Na$ is 2.6 years, the percentage of activity remaining after the four weeks of experiments is

$$\begin{aligned} A(0.0767) &= 100 \times 2^{-0.0767/2.6} \\ &= 97.98\% \end{aligned}$$

where 0.0767 is four weeks converted to years. A difference in activity of two percent should not be enough to make a noticeable difference in the results of the experiments or the processing of the data.

### 4.1.3 Custom Rig




(a) Front view of the first disc with notes showing numbering conventions.

(b) Front view of second disc; columns of holes are rotated by 45 deg.

Figure 4.3: Polyethylene discs used in PEPT experiments.

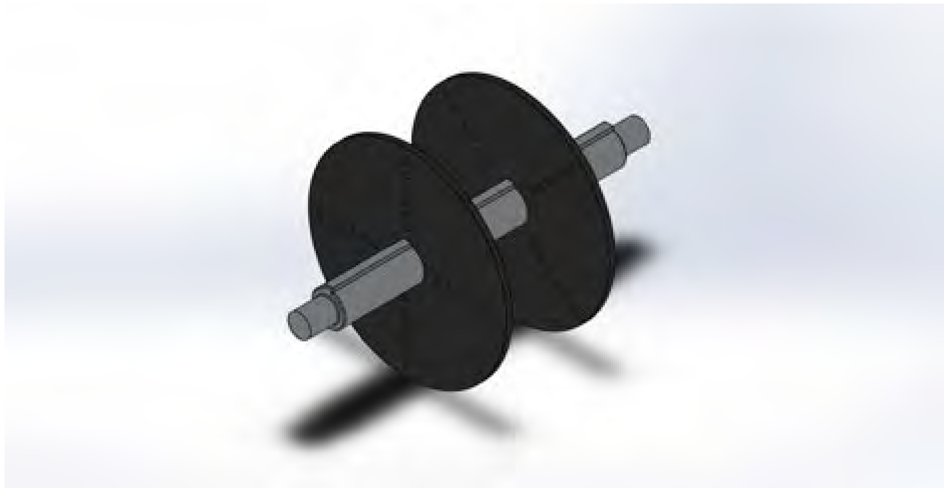


Figure 4.4: Assembly showing polyethylene discs assembled onto aluminium shaft.

A rig was built for the experiments to ensure controlled circular motion. It consisted of an aluminium shaft powered by an electric motor (shown in Figure 4.5a), with three high-density polyethylene discs with holes in which to place the tracers. These discs are shown in Figure 4.3. Each disc had a series of holes in which tracers could be placed. Two of the discs had the holes in the configuration shown in Figure 4.3a, with the first column of holes in line with the keyway. The third disc had the columns of holes rotated through an angle of 45deg.

Initially all three discs were used to run the experiments. However, during analysis it was found that tracers in the discs closer to the edges of the field of view of the scanner were less likely to be detected, due to the number of LOR's being concentrated towards the centre of the field of view. After

this realisation, subsequent experiments were performed using only two discs, and some were repeated in order to provide more consistent data.

The holes in the discs were numbered to keep track of the tracer configurations. The numbering system took the format $a.b.c$ where:

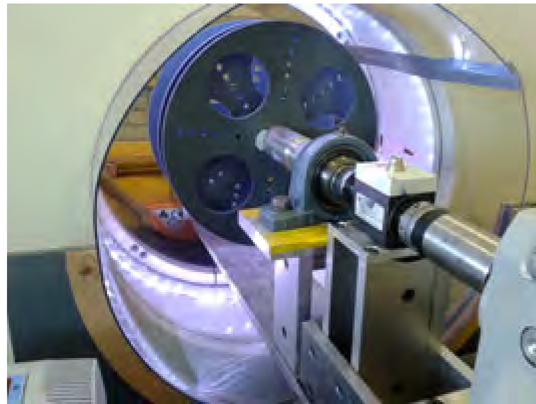
- $a$ is the number of the disc, and goes from 1 to 3.
- $b$ is the column number, and runs from 1 to 4 clockwise, viewed from the front of the scanner.
- $c$ is the number of the hole in the column, and runs from 1 to 6 radially inward.

Figure 4.3 illustrates this numbering system.

The discs fit on to the shaft using keys to hold them stationary relative to the shaft. Meanwhile, the shaft was attached to the motor through the use of a ball bearing, and supported at the other end using a similar bearing contained within a housing. The whole rig was inserted through the field of view of the scanner and supported at both ends by two hydraulic trolleys; this is shown in Figure 4.5b. Figure 4.4 shows two discs assembled to the shaft.

The motor speed was changed using a digital control panel on the top of the motor. The value displayed on the control panel does not indicate the speed of the motor; rather, the value is a function of the speed and the torque of the motor. However, for all the experiments a constant torque was used, so the readings were proportional to the rotational velocity of the shaft. The actual rotational velocity was measured with a digital tachometer. In Section 4.2.4 the relationship between display reading and rotational velocity is fully quantified.

(a) Side view of the driving motor.



(b) View of the assembled rig inside the EXACT3D scanner.

Figure 4.5: Custom rig used in multiple-PEPT experiments.

## 4.2 Descriptions of Experiments

Multiple different experiments were performed, each one designed to test a different aspect of the tracking and to identify weaknesses and limitations of the VMPT algorithm.

### 4.2.1 Circular Rotation

Table 4.2: List of experiments performed under controlled rotation conditions. Tracers were kept stationary for 30 s, after which they were rotated with a controlled circular motion.

| Tracer Count | Number of Runs | Run time (s) |
|---:|---:|---:|
| 1 | 13 | 120 |
| 2 | 10 | 120 |
| 4 | 6 | 120 |
| 6 | 10 | 120 |
| 8 | 10 | 120 |
| 12 | 13 | 120 |
| 14 | 6 | 120 |
| 16 | 6 | 120 |
| 17 | 6 | 120 |
| 18 | 6 | 120 |
| 19 | 4 | 120 |
| 20 | 4 | 120 |

This group of experiments was the most simple, designed for proof of concept, accuracy quantification, and to find the upper limit to the number of possible tracers that the algorithm could successfully track. The tracers were placed into the holes in the plastic discs and rotated at constant velocity for two minutes. Two different rotational velocities were used throughout this series of tests, with the controlled variable being the number of tracers, which ranged from one to twenty. Since the motion of the particles was known, the data from this experiment was used to determine the degree of precision to which the algorithm could locate the tracers.

For each velocity and tracer count, different placement configurations were used. For low tracer counts, many different configurations were used: as the number of tracers increased, the number of possible configurations decreased, especially because having multiple tracers in close proximity to each other led to large numbers of LOR's interfering with each other.

It can be seen in Table 4.2 that there are more runs for tracer counts of 2,6,10,8 and 12. The initial setup used three discs evenly spaced along the length of the rotating shaft. However, during the analysis, it was found that the two outer discs were close enough to the edge of the field of view that the tracers on these discs were seldom tracked properly. The configuration was later changed to only two discs, evenly spaced, and further experiments were performed using the listed tracer counts.

The runs using only a single tracer were used mainly to compare the VMPT algorithm for one tracer to the existing Birmingham algorithm. They were also used as location markers, to locate the positions of the discs along the shaft.

### 4.2.2 Erratic Rotation

These experiments were designed to test the tracking capabilities when the tracers experienced irregular, unpredictable motion, with larger accelerations than in the controlled rotation experiments. To achieve the semi-uncontrolled rotation, lengths of rubber band were tied to the end of the tracers. The rubber bands were then attached to the discs; during rotation, the tracers would swing back and forth and jump erratically along with the original rotation.

While the circular rotation experiments were used for quantitative analysis of the VMPT algorithm, the erratic experiments were used more as a proof of concept. The motion during these experiments provided higher velocities and accelerations, as well as a chance of tracers impacting (or coming close enough to each other such that the algorithm fails to distinguish between the two).

Table 4.3 shows the tracer count in each erratic rotation run, as well as the number of tracers that experienced this described erratic rotation. In each run, a number of tracers were placed firmly in the

Table 4.3: Experiments performed for erratic rotation.

| Run ID | Tracer Count | Free Tracers |
|---|---:|---:|
| umpt_001 | 02 | 1 |
| umpt_002 | 02 | 1 |
| umpt_003 | 02 | 1 |
| umpt_004 | 04 | 3 |
| umpt_005 | 04 | 3 |
| umpt_006 | 04 | 3 |
| umpt_007 | 04 | 3 |
| umpt_008 | 08 | 7 |
| umpt_009 | 08 | 7 |
| umpt_010 | 08 | 4 |
| umpt_011 | 08 | 4 |
| umpt_012 | 10 | 5 |
| umpt_013 | 12 | 6 |
| umpt_014 | 10 | 9 |
| umpt_015 | 12 | 11 |
| umpt_016 | 14 | 7 |
| umpt_017 | 14 | 7 |
| umpt_018 | 16 | 8 |
| umpt_019 | 17 | 9 |
| umpt_020 | 17 | 9 |
| umpt_021 | 18 | 10 |
| umpt_022 | 14 | 11 |
| umpt_023 | 14 | 11 |
| umpt_024 | 16 | 13 |
| umpt_025 | 17 | 14 |
| umpt_026 | 18 | 15 |
| umpt_027 | 10 | 5 |
| umpt_028 | 12 | 6 |
| umpt_029 | 10 | 9 |
| umpt_030 | 12 | 11 |
| umpt_031 | 16 | 8 |
| umpt_032 | 17 | 9 |
| umpt_033 | 18 | 10 |
| umpt_034 | 19 | 11 |
| umpt_035 | 20 | 12 |

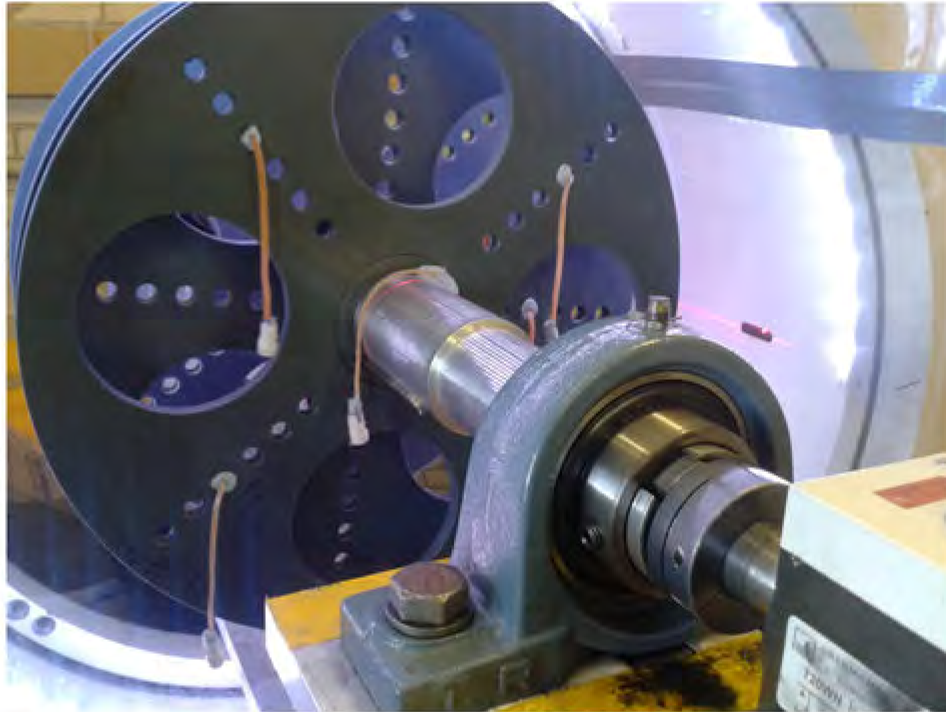


Figure 4.6: Tracers attached to rubber bands to provide semi-uncontrolled motion.

holes in the discs to ensure controlled circular rotation. There were two main reasons for having both erratic and controlled rotation within the same experiment. Firstly, the controlled tracers could be used as a check to verify that the tracking was working properly. Secondly, the erratic motion could be compared to the controlled motion in order to more clearly observe the effect that the erratic motion had on the tracking algorithm.

### 4.2.3 Z-axis Rotation

To test how the algorithm responded to tracers exiting and re-entering the field of view of the scanner, a separate rig was used. This rig consisted of a smaller disc attached to a less powerful motor, which rested on a wooden board going through the field of view of the scanner. The disc rotated in the $xz$ plane, and was aligned off-centre in the $z$-axis so that tracers could exit and re-enter the field of view in a controlled manner. These experiments were run at roughly 100 RPM; however, this speed was difficult to keep constant due to a poor control system.

Unlike the other tests, the rig did not have holes in which to place the tracers. The positions were therefore defined with $(r, \theta)$ pairs, where $r$ is the distance from the centre of rotation, and $\theta$ is the angle clockwise from the positive $x$-axis. A total of 21 tests were performed, with 16 of those being completely within the field of view of the camera, and the other five involving tracers exiting and entering the field of view.

### 4.2.4 Varying Speed

These tests were performed to examine the effect tracer velocity on the accuracy of the location algorithm. They were similar to the controlled rotation experiments, except that number of tracers and the tracer configurations were kept constant, with only the rotational velocity being changed.

A useful consequence of the velocity experiments was that a relationship between the reading on the motor and the actual speed of the shaft could be determined. The readings and speeds from Table

Table 4.4: Experiments performed for rotation in the $xz$ axis with tracers inside the field of view at all times. Tracer positions used in configuration lists are of the form $(x, \theta)$, where $x$ is a fraction of the maximum radius, and $\theta$ is the angular position in °C.

| Run ID | Tracer Count | Tracer Configuration $(x \times R, \theta)$ |
|---|---|---|
| zmpt_001 | 1 | (0,0) |
| zmpt_002 | 3 | (0,0),(1,0),(1,180) |
| zmpt_003 | 3 | (0,0),(1,0),(0.5,0) |
| zmpt_004 | 3 | (0,0),(1,0),(0.5,90) |
| zmpt_005 | 5 | (0,0),(1,180),(0.5,90),<br>(1,0),(0.5,270) |
| zmpt_006 | 5 | (0,0),(1,180),(1,90),<br>(1,0),(1,270) |
| zmpt_007 | 5 | (0,0),(1,180),(1,90),<br>(0.5,180),(0.5,90) |
| zmpt_008 | 5 | (0,0),(1,0),(1,180),<br>(0.5,0),(0.5,180) |
| zmpt_009 | 9 | (0,0),(1,0),(0.5,0),<br>(1,180),(0.5,180),(1,90),<br>(0.5,90),(1,270),(0.5,270) |
| zmpt_010 | 9 | (0,0),(1,0),(1,45),<br>(1,90),(1,135),(1,180),<br>(1,225),(1,270),(1,315) |
| zmpt_011 | 9 | (0,0),(1,0),(1,90),<br>(1,180),(0.5,180),(0.5,225),<br>(1,270),(0.67,0),(0.33,0) |
| zmpt_012 | 9 | (0,0),(1,0),(1,90),<br>(1,180),(0.5,180),(0.5,225),<br>(1,270),(0.67,0),(0.33,0) |
| zmpt_013 | 12 | (0,0),(1,0),(1,90),<br>(1,180),(1,270),(0.5,0),<br>(0.5,90),(0.5,180),(0.5,270),<br>(1,45),(1,135),(1,225) |
| zmpt_014 | 12 | (0,0),(1,0),(1,90),<br>(1,180),(1,270),(0.5,0),<br>(0.5,90),(0.5,180),(0.5,270),<br>(0.5,45),(0.5,135),(0.5,225) |
| zmpt_015 | 12 | (0,0),(1,0),(1,90),<br>(1,180),(1,270),(0.67,0),<br>(0.33,0),(0.67,90),(0.33,90),<br>(0.67,180),(0.33,180),(0.5,270) |

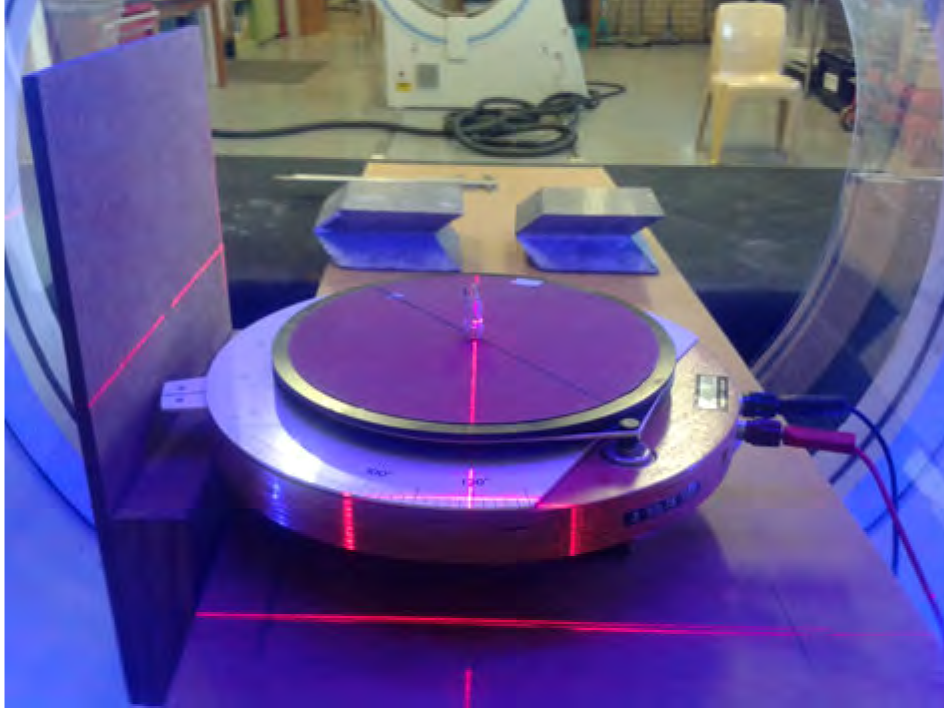


Figure 4.7: Rig to provide rotation in the z-axis, in order to test the effect of tracers exiting and entering the field of view.

4.6 were plotted in Figure 4.8, and a line of best fit was calculated. Since each measurement had some uncertainty associated with it, a method developed by York *et al.* [51] was used to implement these uncertainties in the fit. The result is an equation of the form

$$y = (a \pm \mathrm{u}(a))x + (b \pm \mathrm{u}(b)) \tag{4.3}$$

where $a, b$ are constants and $\mathrm{u}(a), \mathrm{u}(b)$ are the uncertainties associated with those constants. The variables $y$ and $x$ represent the rotational velocity and the motor reading, respectively. Assuming the

Table 4.5: Experiments performed for rotation in the $xz$ axis with tracers exiting and entering the field of view. Tracer positions used in configuration lists are of the form $(x, \theta)$, where $x$ is a fraction of the maximum radius, and $\theta$ is the angular position in °C.

| Run ID | Tracer Count | Tracer Configuration $(x \times R, \theta)$ |
|---|---|---|
| zmpt_017 | 2 | (0,0),(1,0) |
| zmpt_018 | 3 | (0,0),(1,270),(0.5,270) |
| zmpt_019 | 3 | (0,0),(1,270),(1,180) |
| zmpt_020 | 5 | (0,0),(1,270),(0.5,0),<br>(0.5,90),(0.5,180) |
| zmpt_021 | 5 | (0,0),(1,270),(1,180),<br>(0.5,0),(0.5,90) |

motor readings have no units, we arrive at

$$\begin{aligned} a &= 0.0804 \text{ rpm} \\ \text{u}(a) &= 0.0011 \text{ rpm} \\ b &= 0.29 \text{ rpm} \\ \text{u}(b) &= 0.59 \text{ rpm} \\ \omega_s &= (0.0804 \pm 0.0011)\omega_r + (0.29 \pm 0.59) \end{aligned}$$

where $\omega_s$ is the rotational velocity of the shaft, in revolutions per minute, and $\omega_r$ is the reading on the motor display.

Table 4.6: Velocities used for velocity-controlled rotation. Since the reading on the motor was in arbitrary units, a digital tachometer was used to measure the speed in RPM. All runs used 4 tracers.

| Run ID | Motor Display | u(Display) | Motor Speed (RPM) | u(Speed) |
|---|---|---|---|---|
| rmpt_001 | 170.1 | 1.7 | 13.80 | 0.69 |
| rmpt_002 | 219.9 | 2.2 | 18.00 | 0.90 |
| rmpt_003 | 270.0 | 2.7 | 22.1 | 1.1 |
| rmpt_004 | 320.1 | 3.2 | 26.1 | 1.3 |
| rmpt_005 | 369.9 | 3.7 | 30.2 | 1.5 |
| rmpt_006 | 420.0 | 4.2 | 34.2 | 1.7 |
| rmpt_007 | 470.1 | 4.7 | 38.2 | 1.9 |
| rmpt_008 | 519.9 | 5.2 | 42.4 | 2.1 |
| rmpt_009 | 570.0 | 5.7 | 46.3 | 2.3 |
| rmpt_010 | 620.1 | 6.2 | 50.2 | 2.5 |
| rmpt_011 | 669.9 | 6.7 | 54.1 | 2.7 |
| rmpt_012 | 720.0 | 7.2 | 58.4 | 2.9 |
| rmpt_013 | 770.1 | 7.7 | 62.4 | 3.1 |
| rmpt_014 | 819.9 | 8.2 | 66.2 | 3.3 |
| rmpt_015 | 870.0 | 8.7 | 70.2 | 3.5 |
| rmpt_016 | 920.1 | 9.2 | 74.1 | 3.7 |
| rmpt_017 | 969.9 | 9.7 | 78.2 | 3.9 |
| rmpt_018 | 1020 | 10 | 82.0 | 4.1 |
| rmpt_019 | 1070 | 11 | 86.3 | 4.3 |
| rmpt_020 | 1120 | 11 | 90.4 | 4.5 |
| rmpt_021 | 1170 | 12 | 94.3 | 4.7 |
| rmpt_022 | 1220 | 12 | 98.4 | 4.9 |
| rmpt_023 | 1270 | 13 | 102.4 | 5.1 |
| rmpt_024 | 1320 | 13 | 106.3 | 5.3 |
| rmpt_025 | 1370 | 14 | 110.3 | 5.5 |
| rmpt_026 | 1420 | 14 | 114.3 | 5.7 |
| rmpt_027 | 1470 | 15 | 118.3 | 5.9 |
| rmpt_028 | 1520 | 15 | 122.3 | 6.1 |
| rmpt_029 | 1570 | 16 | 126.3 | 6.3 |
| rmpt_030 | 1620 | 16 | 130.3 | 6.5 |
| rmpt_031 | 1670 | 17 | 134.2 | 6.7 |
| rmpt_032 | 1720 | 17 | 138.3 | 6.9 |
| rmpt_033 | 1770 | 18 | 142.2 | 7.1 |

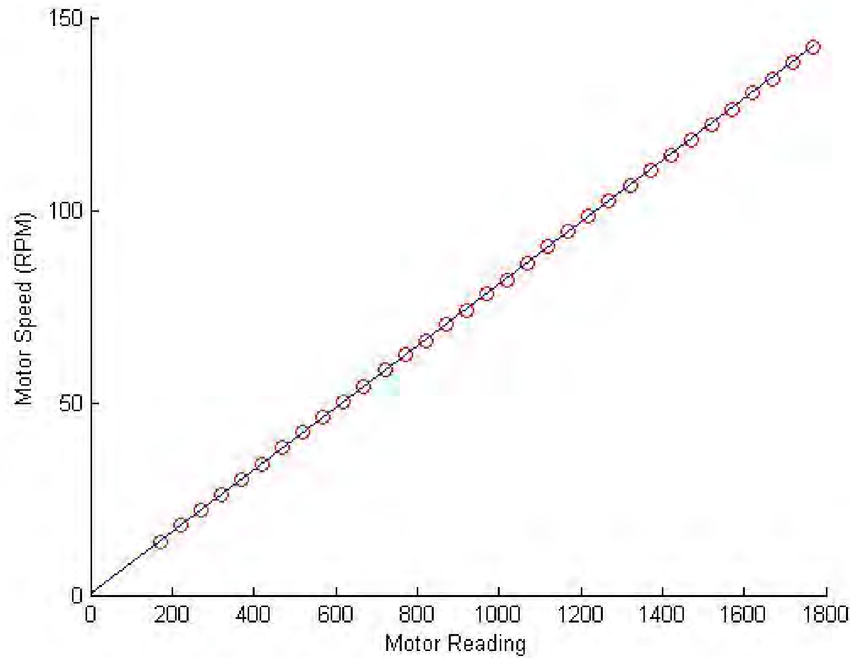



Figure 4.8: Plot of motor reading vs measured rotational velocity.

In Table 4.6, two measures of uncertainty were used: the uncertainty in the reading of the instruments ($u_{read}$), and the rating of the instruments ($u_{rate}$). These two values are combined as follows to give the total uncertainty:

$$u_{read}(x) = (x_{inc})/2\sqrt{6} \tag{4.4}$$

$$u_{rate}(x) = x \times rating \tag{4.5}$$

$$u(x) = \sqrt{u_{rate}(x)^2 + u_{read}(x)^2}, \tag{4.6}$$

where *rating* is the rating of the instrument used to take the reading and $x_{inc}$ is the smallest increment of the reading.

### 4.2.5 Static Deflection

Some very simple tests were performed to test the viability of using multiple-PEPT to measure deformation of solid materials. Multiple tracers were placed along the length of a steel rod of known mechanical properties. The rod was held in place with a retort clamp (see Figure 4.9) and then slowly deformed, using a mass tied to the end of the rod while it was inside the FOV of the scanner. The rod could then be modelled as a simple cantilever beam with a point force at a known location. This type of deformation was chosen because analytical solutions to static deflections in a cantilever are well known, so the experimental data could be compared to a theoretical deflection.

In Table 4.7. the Total Length is the length of the physical rod, while the Deflection Length is the distance from the stationary point at the clamp to the tracer placed furthest from the clamp. The material of the rod was not immediately available, so an educated guess had to be made. The manufacturer of the rod was known, so their inventory was scoured to find a product that closely matched the dimensions. There were multiple options, but Sanmac Steel 4435 seemed the most likely. Sanmac steel 4435 was assumed to be a reasonable guess, since the yield strength and Young's modulus are typical of most stainless steels, at 215 MPa and 180 GPa respectively.

(a) Front view of the rod configuration with mass attached to show deflection.

(b) Top view of the rod without the mass attached.

Figure 4.9: Steel rod setup used in static deflection tests.

Table 4.7: Physical specifications of static deflection setup.

| **Specification** | **Symbol** | **Value** | **Units** |
|---:|---:|---:|---:|
| Total Length | $L_{rod}$ | 0.5 | m |
| Deflection Length | $L_{def}$ | 0.4 | m |
| Outer Diameter | $D_{rod}$ | 3.175 | mm |
| Inner Diameter | $d_{rod}$ | 1.1 | mm |
| Material | - | Sanmac Steel 4435 | - |
| Yield Strength | $\sigma_y$ | 220 | MPa |
| Young's Modulus | $E_{rod}$ | 200 | GPa |
| Mass 1 | $m_1$ | 101.35 | g |
| Mass 2 | $m_2$ | 120.15 | g |

For the sake of the experiment, it was important that the rod only experienced stresses within the elastic deformation range. There were two reasons for this:

- For repeatability of the experiment. Once the rod undergoes plastic deformation, it cannot return to its original state.
- For simplicity. Beam calculations within the elastic range are fairly simple; since this experiment was used as a proof of concept, the simplest approach was used.

To ensure a safe loading configuration, the equation for a beam under stress was used:

$$\frac{\sigma}{n_s} = \frac{M\bar{y}}{I}. \tag{4.7}$$

Here, $\sigma$ is the stress at some point along the rod; $M$ is the bending moment at that point; $\bar{y}$ is the distance from the neutral axis to the point; and I is the second moment of area of the rod about the neutral axis. The variable $n_s$ is simply a safety factor, which was set to a value of 1.2.

Given this equation, we can define the variables in terms of the dimensions of the rod:

$$M = m_1 g \times L_{def} \tag{4.8}$$

$$\bar{y} = D_{rod}/2 \tag{4.9}$$

$$I = \pi \times \frac{D_{rod}^4 - d_{rod}^4}{64}. \tag{4.10}$$

The safe length can then be found by substituting the variables in 4.8 into Equation 4.7 and solving for $L_{def}$ to give

$$L_{def} = \frac{\sigma I}{m g \bar{y} n_s}, \tag{4.11}$$

and substituting the yield stress $\sigma_y$ for $\sigma$ yields a safe deflection length of 0.482m.

Markers were placed along the rod at regular intervals of 25mm, for a total of 16 nodes along the length of rod. Depending on the configuration used, tracers were placed at some of these nodes to track the deflection due to the load being applied. The two tracer configurations used are shown in Table 4.8.

Table 4.8: Setup configurations used in static deflection tests.

| Run ID | Tracer Count | Mass Used (g) |
|---|---|---|
| dmpt_001 | 2 | 101.35 |
| dmpt_002 | 2 | 120.15 |
| dmpt_003 | 3 | 101.35 |
| dmpt_004 | 3 | 101.35 |
| dmpt_005 | 4 | 101.35 |
| dmpt_006 | 5 | 101.35 |
| dmpt_007 | 9 | 101.35 |
| dmpt_008 | 3 | 120.15 |
| dmpt_009 | 5 | 120.15 |
| dmpt_010 | 9 | 120.15 |

### 4.2.6 Tracer Impact

The purposes of these tests were to observe how the algorithm handled two tracers coming into contact with each other. Theoretically, when two tracers come within some distance of each other, the clusters generated by the location stage of the VMPT algorithm would overlap and become indistinguishable.

It was initially assumed that this distance would be of the order of the $\epsilon$ variable used in the DBSCAN clustering algorithm, since a point must be within a distance $\epsilon$ of another point to be considered part of the same cluster. Note that this distance is not a centre-to-centre distance, but rather boundary-to-boundary. This is because $\epsilon$ is chosen such that there is a high probability of data points belonging to a cluster falling within the bounds of the physical tracer.

The experimental setup was fairly simple. A horizontal beam was supported by two retort stands and clamps, and aligned such that the length of the beam lay along the $x$-axis of the field of view of the scanner. Two tracers were tied to this beam, with one tracer hanging slightly lower than the other. During the experiment, the lower-hanging tracer was pulled to the side and allowed to swing sideways and impact with the other tracer. Once both tracers stopped moving, they were made to impact again. This was repeated multiple times over the course of the experiment.

# Chapter 5

# Analysis

## 5.1 Curve Fitting

During the majority of the experiments, the tracers were rotating in a controlled circular path. As such, plotting the $x, y, z$ positions versus time gives three sinusoidal graphs. Knowing this, we can fit a simple sine function to each dimension of a data set of the form

$$x(t) = A_x + B_x \sin(\omega_x t - \phi_x), \tag{5.1}$$

where $A_x$ is the offset, $B_x$ is the amplitude of the curve, $\omega_x$ is the rotational velocity, $t$ is the time, and $\phi_x$ is the initial angular offset of the curve fit to dimensions $x$.

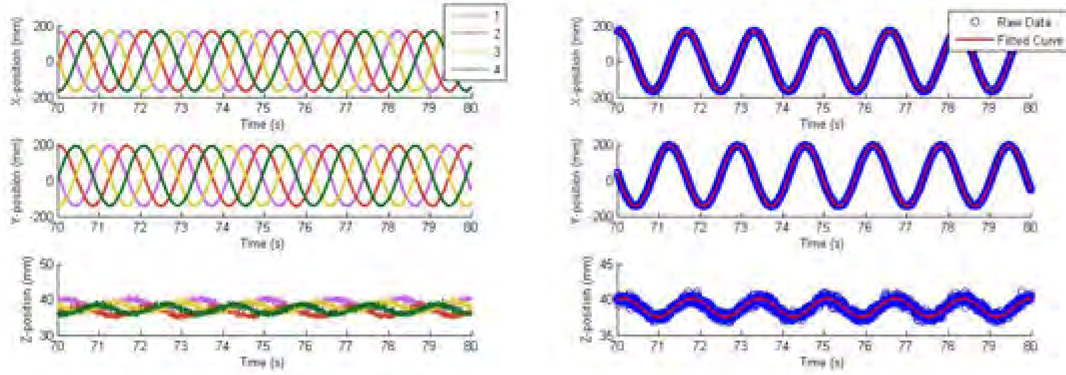
The MatLab function *NonLinearModel.fit(* x , y , model , beta0 *)* was used to fit the data. The input parameters to the function are defined as follows:

- **x**: a vector of real numbers, this stores the x-values of the measured data to be fitted.
- **y**: a vector of real numbers, this stores the y-values of the measured data to be fitted.
- *model*: a handle to the model function to which the data must be fitted.
- *beta0*: an initial approximation of the coefficients in the model.

The *NonLinearModel.fit()* function makes use of the Levenberg-Marquardt nonlinear least squares algorithm [52] for non-robust estimation of parameters, and an iterative reweighted least squares algorithm for robust estimation. The relevant values returned by the function are listed below:

- coefficients
  - estimates: the estimated values of the coefficients.
  - SE: standard error of the estimates.
- fitted: a vector containing predicted values based on the input data.
- residuals: a table of the residuals of the fit (observed minus fitted values).
- RMSE: the root mean square error, analogous to the standard deviation.

Once the best-fit function has been calculated, it can be plotted alongside the raw position-time data as shown in Figure 5.1. Figure 5.1a shows the position-time scatter plots for an experiment in which four tracers were moving with controlled circular motion. The plot shows four distinct tracks, each taking the form of sine waves in each of the $x$,$y$ and $z$ dimensions. A curve is fitted to tracer 1 in Figure 5.1b, and upon visual inspection this curve shows a close correlation to the data.




(a) Raw position-time data for four tracers moving with a controlled circular motion.

(b) Sine curve fitted to tracer number 1.

Figure 5.1: Illustration of best-fit sine curves to track data generated by the VMPT algorithm.

Note that in Equation 5.1, $\omega_x, \omega_y, \omega_z$ should be equal (within experimental error). In other words, the angular velocities in each dimension should agree, since the tracer is moving as part of a rigid body. An example with a single tracer is used to verify this assumption in Table 5.2, which lists the rotational velocities and their errors in each dimension of a single tracer. The table shows that, within standard error, all of the rotational velocities agree with each other.

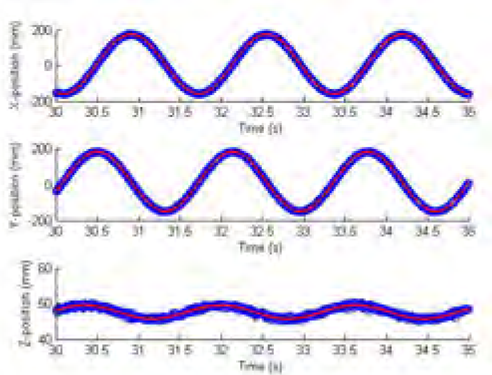



Table 5.1: Scatter plots and best fit curves of a single tracer experiencing controlled circular motion.

| | Value (rad/s) | Standard Error (rad/s) |
|---|---|---|
| $\omega_x$ | 3.8328 | 0.0002 |
| $\omega_y$ | 3.8325 | 0.0002 |
| $\omega_z$ | 3.8276 | 0.0062 |

Table 5.2: Rotational velocities and standard errors for each dimension $x, y, z$.

## 5.2 Track Loss

Track loss occurs when a tracer appears to have disappeared from the field of view of the scanner. During the tracking part of the algorithm (see Section 3.2), if at some time a track does not have a matching entry, that track object gets assigned a null entry and the skip counter (which stores the number of consecutive null entries) is incremented by 1. If the skip counter becomes larger than some predefined value, the track is terminated, and either saved or discarded accordingly.

Track loss can happen because of one of several reasons:

- The tracer has left the field of view of the scanner.

- The tracer is within the field of view of the scanner, but is otherwise undetectable due to being in the vicinity of a high density object which causes Compton scattering or attenuation.
- A tracer experiences large accelerations, and large time increments are used in the tracking algorithm.
- Two tracers move close together or come into contact with one another.
- A large number of tracers can increase the chance of track loss.

After a track has been lost, the tracer corresponding to that track ceases to exist according to the algorithm. When it reappears, it is detected as an entirely new, separate tracer. This is because the only distinguishing feature of a tracer is its current position and velocity. The article published in 2006 by Yang *et al.* [53] distinguished tracers according to their relative activities. That is, each successive tracer was labelled with an activity level twice that of the preceding one. This is a reasonable practice when tracking low numbers of tracers. However, as the number of tracers increases, two major problems arise:

- The activity difference between the tracer with the lowest and that with the highest activity becomes large, reducing the chance that the lower-activity tracers can be detected at all.
- The activity needed to label each successive tracer increases exponentially, quickly leading to saturation of the detector cells.

Due to these restrictions, using activity as an identifier is not feasible for large numbers of tracers.

### 5.2.1 Tracer Exiting FOV

It is obvious that when a tracer leaves the field of view of the scanner that track loss will occur. As described in the previous section, once the tracer re-enters the field of view of the scanner, it is treated as a new object. If some physical constraint were to be placed on the system which only allowed for one tracer to exit the field of view at a time, the algorithm could be adapted to match a terminated track with a newly created one. However, this would mean that the experiment would be unnaturally controlled, and could lead to poor data.

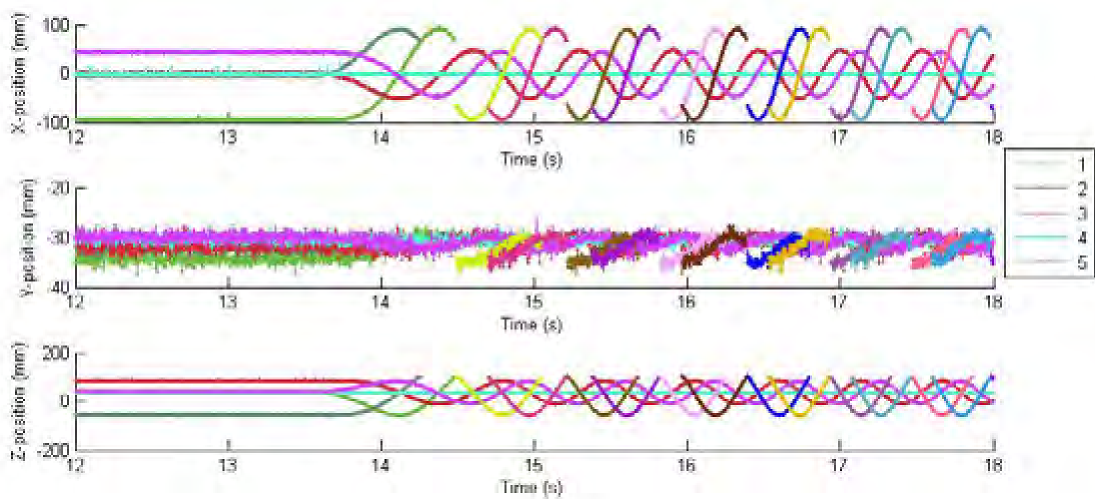
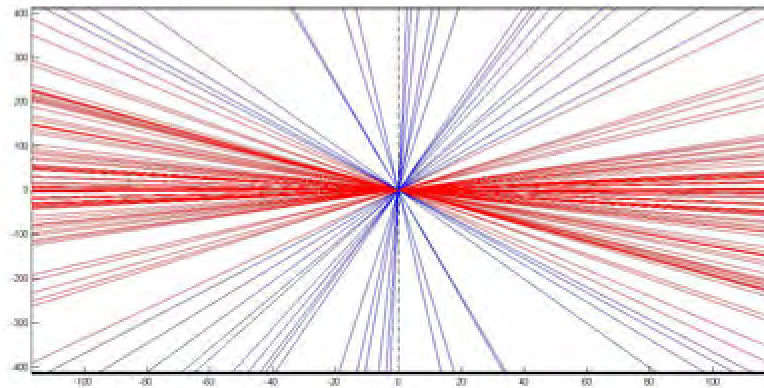


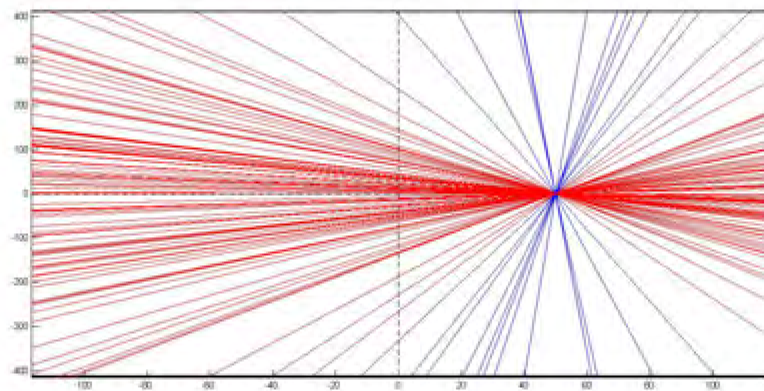

Figure 5.2: Trajectories of experiment in which two tracers exit and re-enter the field of view, while three tracers remain within the field of view.

Figure 5.2 shows the trajectories of 5 tracers. It can be seen that tracers 1 and 2 are those that leave the field of view; their tracks are lost at approximately 14.4 s and 14.2 s, respectively. Once they re-enter the field of view, they are discovered as completely separate tracers, which is made clear by the fact that the colours of the tracks are different to the original colours. This illustrates the idea that the algorithm is unable to uniquely identify a tracer with physical properties, and so cannot track an exit and re-entry.

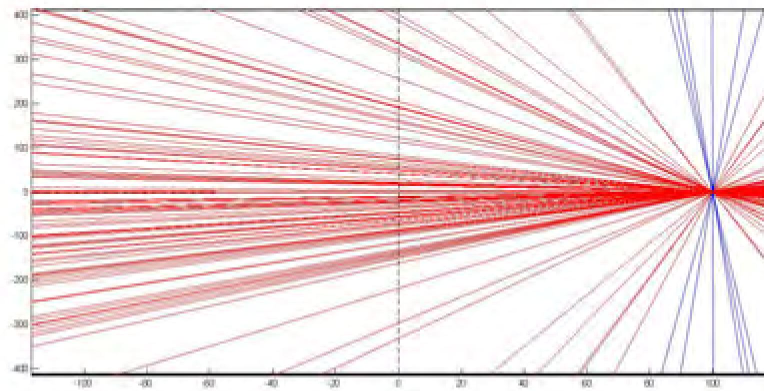
Due to the geometry of the scanner, tracers near the edge of the field of view generate fewer *detectable* lines of response, regardless of the actual activity of the tracer in question. A Monte-Carlo simulation was written to illustrate this principle, and the results are shown in Figure 5.3 For simplicity, the simulation only looks at a 2D longitudinal cross section of the scanner's field of view. The 'tracer' is the source of the lines of response; each line is generated with a random angle, and passes through the tracer location. The 'detector cells' are at the top and bottom of the graph, indicated by thick black lines. If an LOR intersects both the top and bottom line of detector cells within the field of view, it can be detected by the scanner.



(a) Tracer at $z = 0$



(b) Tracer at $z = 50$



(c) Tracer at $z = 100$

Figure 5.3: Results of Monte-Carlo simulations used to illustrate the effect of the $z$-position on the number of lines detectable by the scanner. Red lines do not intersect the top and bottom within the field of view, and therefore cannot be detected.

During the simulation, the tracer was moved along the $z$-axis in increments of 1 mm. At each

location, 1000 lines of response were generated. The fraction of lines which the scanner could detect was then found by dividing the total number of detectable lines by the number of lines generated (1000). This was repeated 100 times for each location, so that a mean and standard deviation of the fraction of detectable lines for each position could be determined. The results of this simulation are plotted in Figure 5.4; it can be seen that the number of detectable lines decreases linearly with the distance from the centre of the field of view.

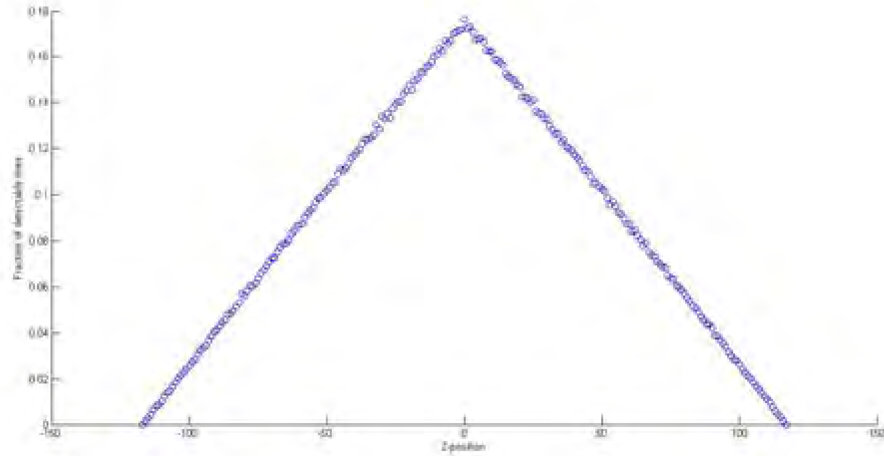


Figure 5.4: Scatter plot showing the fraction of detectable lines of response in a Monte-Carlo simulation.

What the Monte-Carlo simulation shows is that the *perceived* field of view may not correspond to the *physical* field of view. This effect is more noticeable when higher tracer counts are used, since the clustering method is biased towards detecting tracers with a higher perceived activity.

The Monte-Carlo simulation was verified using a Z-axis Rotation experiment in which the tracers were made to exit the field of view of the scanner. For each tracer, and at each time step, the number of data points identified as contributing to the cluster locating that tracer were recorded. Each data point represents one line of response generated by that tracer and detected by the scanner.

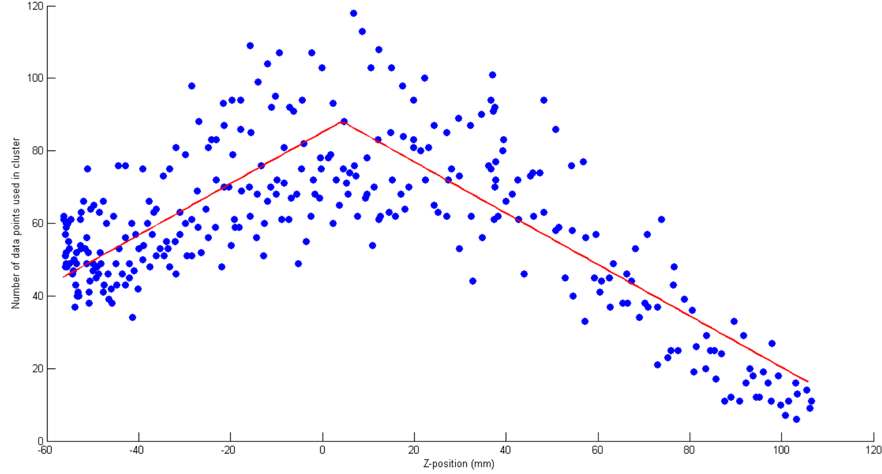



Figure 5.5: Scatter plot and fitted curve showing the number of data points per cluster versus the Z-position of the located tracer.

Figure 5.5 shows the number of data points per cluster plotted against the position of the tracer in the Z-dimension. As per the Monte-Carlo simulation, an absolute value equation of the form

$$f(x) = m|x - a| + c$$

was fitted to the data. Although the scattered data has a high variance, it is clear that the number of data points per cluster reaches a maximum somewhere near the centre of the Z-axis, and decreases linearly to either side.

With this in mind, the value for $k$ used as the nearest neighbour search in the LOF algorithm as well as the minimum points per cluster used in DBSCAN had to be chosen such that the algorithm could locate tracers towards the ends of the scanner. Through trial and error, $k = 4$ was determined to be a reasonable value. However, further research may show that the optimal value for $k$ is not 4, and may even be dependent on other factors, such as the number of tracers used in the experiment.

### 5.2.2 Scattering and Attenuation

As described in Section 2.1.2, Compton scattering occurs when a photon produced by the $\beta^+$ decay is deflected by an electron in the medium surrounding the tracer. The attenuation of some material describes the extent to which that material can be penetrated by light, sound, or some other form of energy.

The material properties that govern the degree of attenuation and scattering are called the mass attenuation, absorption and scattering coefficients. These coefficients are denoted respectively by $\frac{\mu}{\rho_m}$, $\frac{\mu_a}{\rho_m}$, and $\frac{\mu_s}{\rho_m}$, where $\rho_m$ is the density of the material. The attenuation coefficient is simply the sum of the absorption and scattering coefficients. A large attenuation coefficient implies that radiation can easily penetrate the material, and vice versa. Since the attenuation coefficient is inversely proportional to the density of the material, high-density materials are more resistant to penetration than low-density materials.

When a tracer is in the vicinity of a high-density object, some fraction of the photons will be absorbed or scattered. The anti-parallel photons corresponding to these will be detected by the detector cells, but the attenuated partners will not, and the incident will be discarded. This can lead to track loss, especially when using a high tracer count.

### 5.2.3 High Acceleration

During the prediction step of the tracking algorithm, a linear extrapolation method is used to predict the location of a track at the current time and match it with a located tracer. A linear extrapolation method assumes a constant velocity, or zero acceleration. The equation for kinematic motion is

$$\mathbf{x}_f = \mathbf{x}_i + \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}\Delta t + \frac{1}{2}\frac{\mathrm{d}^2\mathbf{x}}{\mathrm{d}t^2}(\Delta t)^2 + \frac{1}{6}\frac{\mathrm{d}^3\mathbf{x}}{\mathrm{d}t^3}(\Delta t)^3 + ... + \frac{1}{n!}\frac{\mathrm{d}^n\mathbf{x}}{\mathrm{d}t^n} \tag{5.2}$$

$$\mathbf{x}_f = \mathbf{x}_i + \mathbf{v}\Delta t + \frac{1}{2}\mathbf{a}(\Delta t)^2 + ... \tag{5.3}$$

It can be seen from the equations that, when a linear extrapolation method is used, any time derivatives of order two or greater (acceleration, jerk, etc.) will contribute to an error in the extrapolation; the magnitude of this combined error term will be denoted $\epsilon_x$. The most important term in the combined error term is the second derivative, or the acceleration. In most practical experiments acceleration is easy to measure, and can have a large influence on the outcome. The higher order derivatives have less of an impact, and are often ignored for simplicity.

The linear extrapolation method suffices when the $\epsilon_x$ term is small. Specifically, we want

$$\epsilon_x \leq \delta_g, \tag{5.4}$$

where $\delta_g$ is the search gate distance used by the algorithm, since $\delta_g$ defines the bounds for the matching algorithm. However, when Equation 5.4 does not hold true, a located tracer may not be assigned to an existing track. While this is not an issue if the acceleration is brief, it can lead to track loss when the acceleration continues for an extended amount of time. Combined with the fact that the accuracy of location can be affected in multiple ways, large accelerations can cause disjointed and therefore inaccurate tracking.

A naïve approach to satisfy the condition in Equation 5.4 is simply to increase $\delta_g$. However, this solution poses some problems:

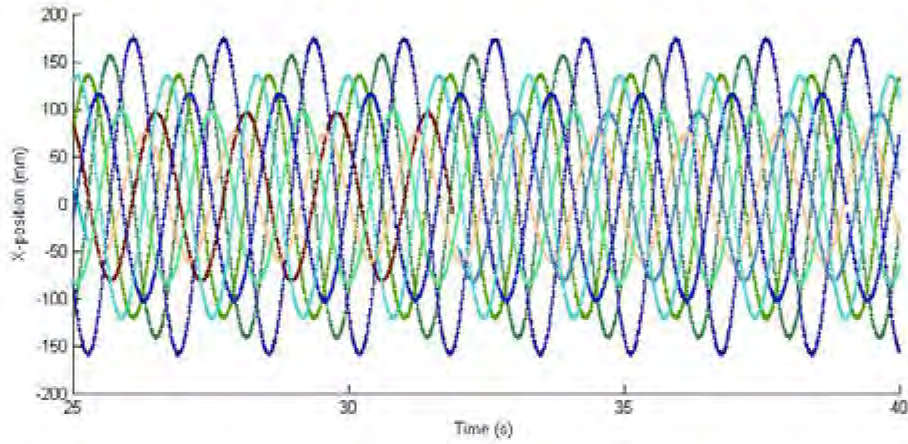
- $\delta_g$ must increase linearly with acceleration, while in a simple circular motion experiment, acceleration (and therefore $\epsilon_x$) is proportional to the square of the velocity. So the search gate distance will have to be large when tracers experience large accelerations.
- The larger the search gate distance becomes, the less useful it is as a tool used to narrow down possibilities during the track-tracer matching step. In the extreme case, an arbitrarily large $\delta_g$ (or one with a value similar to the diameter of the field of view) will serve no purpose.

The other option to decrease the effect of acceleration on track loss is to decrease the time increment $\Delta t$ used during the location algorithm. The benefits of changing the time increment relate directly to the difficulties with increasing the gate distance:

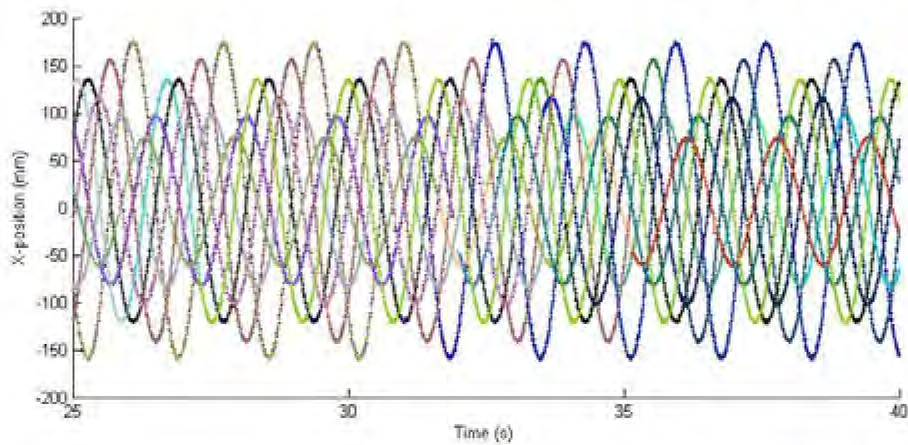
- $\epsilon_x$ decreases *quadratically* with a linear decrease in $\Delta t$. This means that a smaller decrease in the time increment can have a significant effect on the tracking.
- The search gate distance can be kept small, while still being larger than the combined error term.

It must be noted that decreasing the time increment comes with an increase in processing time and memory usage, both during the location and tracking algorithms.

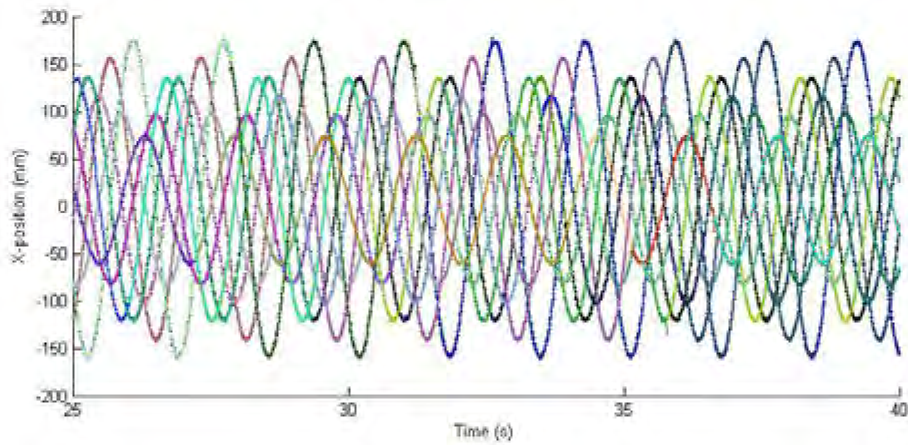
To illustrate the effect of decreasing the time increment, Figure 5.6 compares the tracks generated by location data using three different time intervals. For the sake of clarity, only the $x$-positions are plotted. It can be seen that as the time interval $\Delta t$ increases, the total number of tracks detected also increases. This increase in track count is due to track splitting because of a larger error term $\epsilon_x$.




(a) $\Delta t = 7.21 \pm 0.46$ s. Total number of tracks = 11




(b) $\Delta t = 14.42 \pm 0.58$ s. Total number of tracks = 27




(c) $\Delta t = 21.63 \pm 0.68$ s. Total number of tracks = 44

Figure 5.6: Plots showing 15 s snapshots of tracks generated during an experiment using eight tracers. In each plot, the time interval used during the location stage of the algorithm was changed.

### 5.2.4 Tracer Proximity

When two or more tracers come close enough together, they can be detected as a single tracer. During the DBSCAN clustering algorithm, a point is defined as belonging to a cluster if it comes within a distance $\epsilon$ of another point in that cluster. Therefore, if points from two clusters are within $\epsilon$ of each other, the two clusters will be merged into one. The minimum distance between two tracers which ensures unique identification is reliant on a number of factors:

- The value used for $\epsilon$. Note that this is the same value used for the discretization separation distance between points on an LOR.
- Number of tracers. The number of tracers used in an experiment (and therefore the total number of lines of response within a given frame) affects the distribution of seed points within the field of view. A low tracer count gives rise to a high density differential (clusters are much more dense than the surrounding noise), while a large tracer count gives rise to a lower density differential.
- Velocity of tracers. Although the velocity has a minimal effect on location accuracy (as described in Section 5.3) it does have an effect on the size of the clusters. Higher velocities lead to higher cluster sizes, since the lines of response generated by an individual tracer are spread over a larger distance. Larger clusters lead to a higher chance of those clusters merging.

The effect of tracer proximity on the tracks can be different to other forms of track loss. When two clusters merge, the algorithm detects this combined track for the time period over which the tracers are in close proximity. This can lead to only one track being lost, since the combined track is calculated as belonging to one of the existing tracks. Once the tracers separate sufficiently to allow them to be individually located, a 'new' track is formed, while the combined track reverts to its original path.

Track combination due to proximity is an inevitable result of the algorithm, since tracers are not identified uniquely by any physical properties. During analysis of the tracks, this can be corrected manually by simply deleting the time period over which the tracers merged.

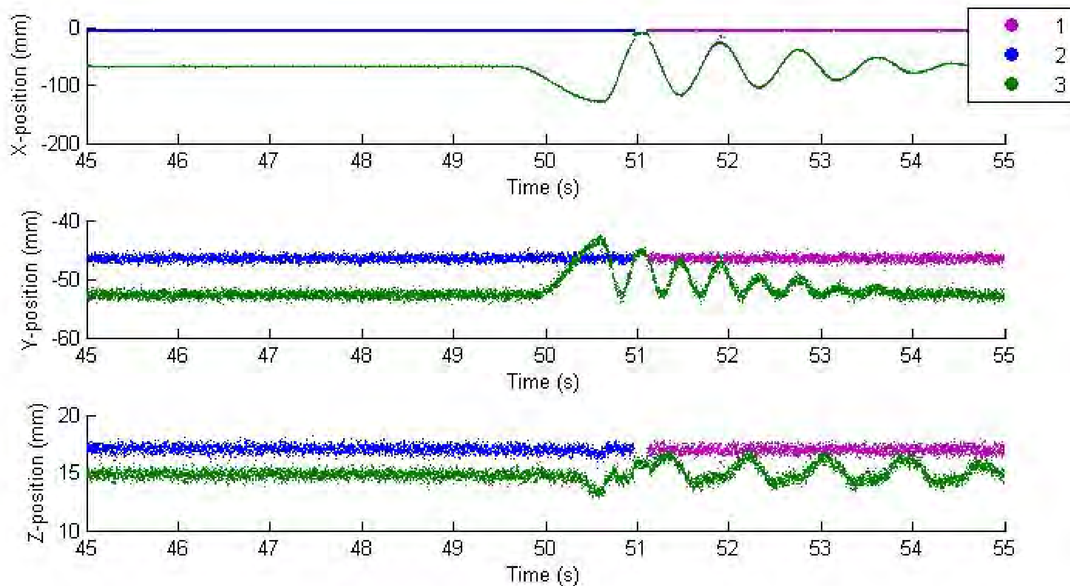



Figure 5.7: Trajectories of two tracers before and after impact.

Figure 5.7 illustrates an example of tracer impact. We will refer to the tracers hereafter as T1, T2, and T3, with the numbers corresponding to the tracks in the figure. Here, T2 is held steady while T3

is pulled to the side and allowed to swing towards T2. At the 51 s mark, T3 collides with T2. It can be seen that the track generated by T2 is lost, because T2 and T3 become indistinguishable from each other. Once T3 moves a sufficient distance away from T2, a 'new' tracer, T1, is located. To the human eye, T1 and T2 are obviously the same tracer. However, the algorithm cannot join the two tracks based on its parameters.

Note that it is difficult, if not impossible, to determine which track will persist and which will be lost. In Figure 5.8, at the 40 s mark, the track generated by the moving tracer is lost, while the stationary track persists. As mentioned previously, this is reversed at the 51 s mark. There may also be cases in which *both* tracks are lost, as observed at the 82 s mark.

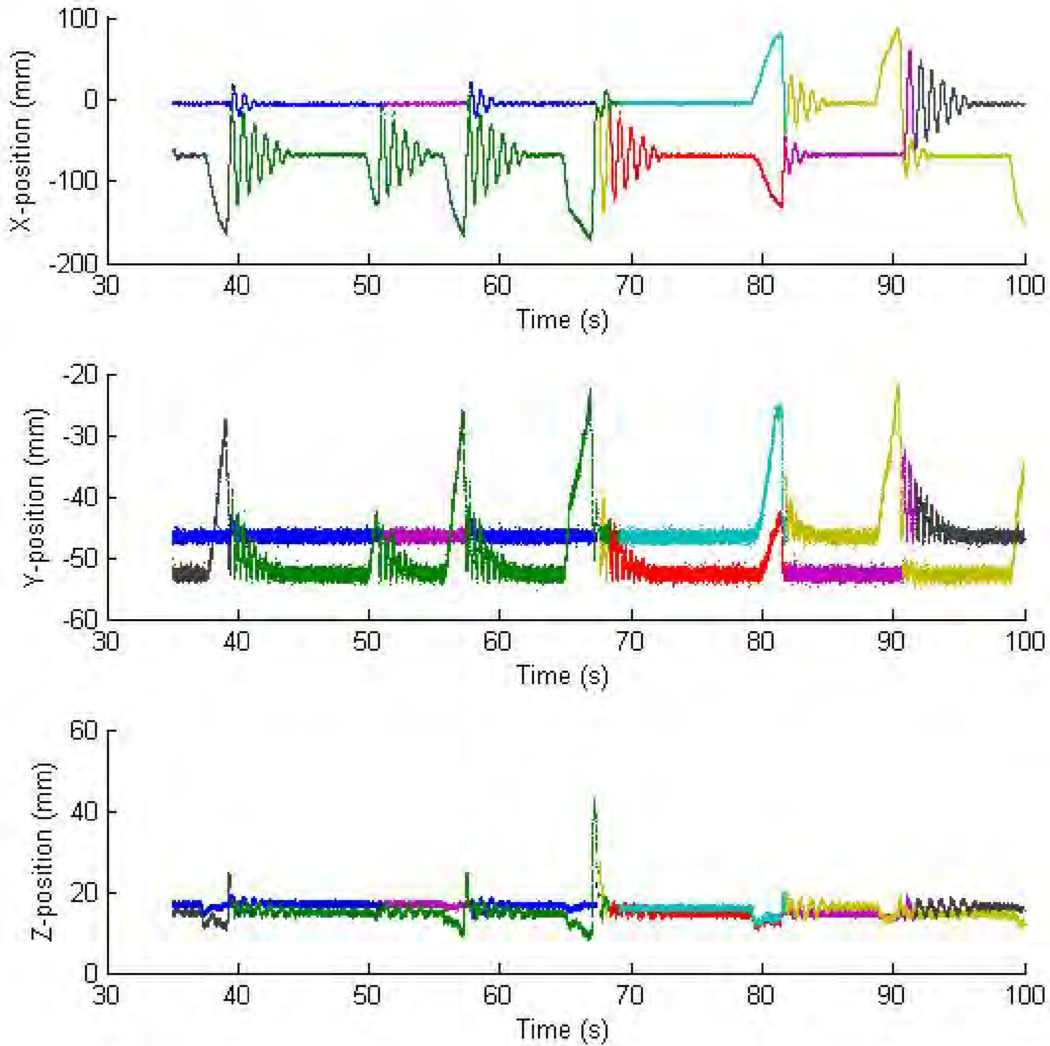



Figure 5.8: Extended trajectories of tracers impacting multiple times.

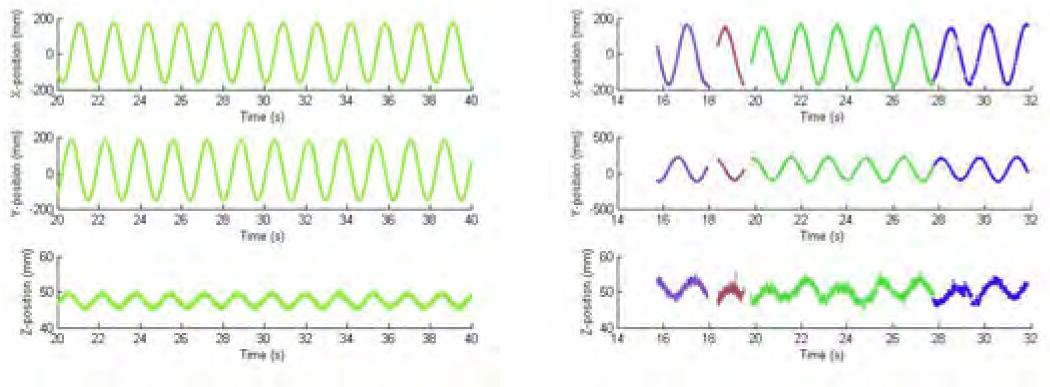
Whether a track persists or is lost depends on the clustering step of the algorithm. The track which is closest to the centroid of the merged cluster will persist, while the other track will be lost. If the two clusters are particularly large (as may happen when both tracers have relatively high velocities), then

the centroid of the merged clusters may not correspond to wither pre-existing track. This causes both tracks to be lost.

### 5.2.5 Large Tracer Count

A high tracer count does not cause track loss itself. Rather, it increases the chance of track loss given the other conditions described above. During the LOF smoothing and clustering steps of the location algorithm (see Section 3.1.2), the chance of locating a tracer is biased towards tracers with higher relative perceived activities. A tracer with a lower activity will be less likely to be clustered since some of the data points may be cleaned away.

The relative perceived activity of a tracer is not equivalent to the actual activity; it is the number of lines of response detected by the scanner that were generated by that tracer relative to the number of detected lines generated by the other tracers within the field of view. Practically, an increased tracer count leads to a smaller viable field of view; a higher chance of track loss due to absorption and scattering; and an increased separation at which two tracers can be uniquely located.




(a) Trajectory of a tracer in a controlled rotation experiment using a total of two tracers.

(b) Trajectory of a tracer in a controlled rotation experiment using a total of 20 tracers.

Figure 5.9: Comparison between tracers in the same position with different total tracer counts.

Figure 5.9 compares the trajectories of a single tracer from two different experiments. In both experiments, the tracer was in the same position and had the same velocity (and therefore acceleration). In neither case did the tracer come near the edge of the field of view of the scanner, and there was little interference due to attenuation or scattering effects. Figure **??** shows one continuous track; in fact, the tracer was successfully tracked over the entirety of the experiment without track loss. However, 5.9b shows four separate tracks over a period of about 16 seconds. To a human observer it is clear that each of these tracks was generated by the same tracer, but they are separated enough such that the VMPT algorithm cannot make the connection. In the controlled rotation experiments, it would be fairly simple to match separated tracks based on the best fit curves, but in real experiments the motion is not necessarily known beforehand.

It is important to note that tracer count does not have a large effect on the precision to which an individual tracer can be located (this is discussed in Section 5.3), but rather affects the chance of track loss and track splitting.

## 5.3 Location Accuracy

To quantify the accuracy of the location, the root mean square error (RMSE) of the sinusoidal fit was used. The RMSE is a measure of how well data fits a model, and is analogous to the R-squared

value used for linear fits, or the standard deviation for a probability distribution. The RMSE (in one dimension) is calculated with

$$\text{RMSE}_x = \sqrt{\frac{\sum_{j=1}^{N}(x(t) - \hat{x}(t))^2}{N}} \tag{5.5}$$

where $N$ is the number of data points, $\hat{x}(t)$ is the measured value of the position at time $t$, and $x(t)$ is the position at time $t$ according to the fitted function. The RMSE is in fact used to find the equation of best fit; an iterative process is used to change the parameters of the equation (namely $A, B, \omega, \text{and}\phi$) in order to minimise the RMSE. If the RMSE in each dimension is known, the three-dimensional RMSE can be found with

$$\text{RMSE} = \sqrt{\text{RMSE}_x^2 + \text{RMSE}_y^2 + \text{RMSE}_z^2}.$$

Note that the curves fitted to the $x, y, z$ data sets should have the same $\omega$ parameter. In practice, this may not be the case, due to the fact that the curves are fitted independently. However, the different rotational velocities should agree within experimental uncertainty.

The RMSE only provides a measure of the relative accuracy, as opposed to the absolute accuracy. Given the experimental setup, it was difficult to determine the absolute error. The exact centre of the field of view is itself difficult to determine, and from there precise placement of a tracer proves problematic as well. The RMSE relative error was deemed acceptable, since in practice the data is smoothed during the post-processing.

While the RMSE is a good measure of the error in location, it is inversely proportional to the 'accuracy' of location. As such, a metric for the accuracy of location $\alpha_l$ is defined as

$$\alpha_l = \frac{1}{RMSE}. \tag{5.6}$$

which is simply the inverse of the RMSE.

### 5.3.1 Velocity

The velocity of the tracers being tracked may have some effect on the accuracy of location. Referring to Section 4.2.4, the experiments used to asses the affect of velocity all used a total of four tracers per run. The tracers were made to rotate in a circle, with the rotational velocity being controlled using a digital tachometer. Although the rotational velocity was controlled, the the linear velocity is in fact the variable of interest. Equation 5.1 can be differentiated with respect to time to find an equation to describe the velocity:

$$v_x(t) = \frac{dx}{dt} = \omega B \cos(\omega t - \phi). \tag{5.7}$$

Here, $v_x$ is the velocity in the $x$-direction. Given $v_x, v_y, v_z$, the magnitude of the linear velocity can be found using

$$v = |\mathbf{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}.$$

Although the velocity in each direction is a function of time, the magnitude of the 3D velocity vector $\mathbf{v}$ is constant, since the tracers rotate at constant rotational velocity and radius. As such, we can choose a time $t_1$, and use it to find $v_x(t_1), v_y(t_1), v_z(t_1)$; the velocity at this time should be the same as at any other time. To keep it as simple as possible, $t_1 = 0$ was chosen for all of the analyses.

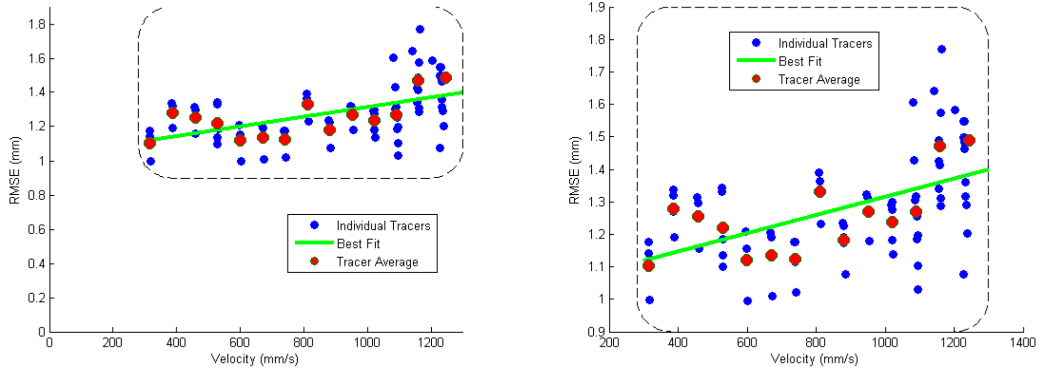



Figure 5.10: Scatter plot showing tracer velocity and respective RMSE values in experiments using four tracers at varying velocities.

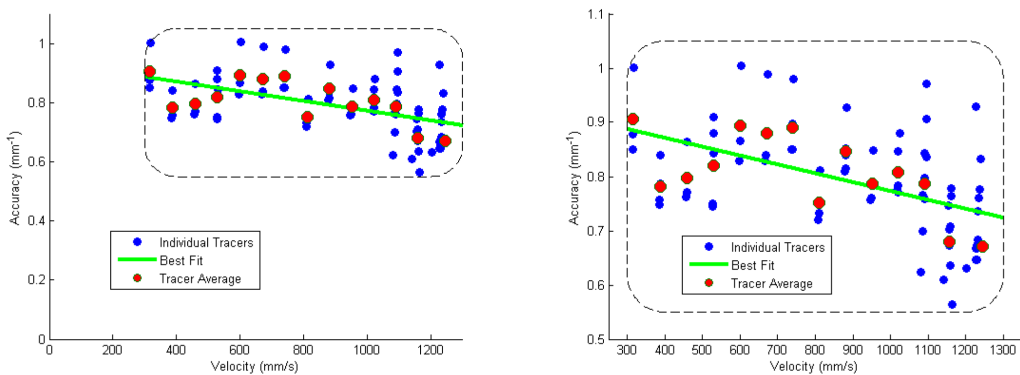



Figure 5.11: Scatter plot showing tracer velocity and respective accuracies in experiments using four tracers at varying velocities.

Figures 5.10 and 5.11 show the results of the velocity tests. Each blue dot represents a single tracer, and the red dots are the averages of the tracers during one run. For each run below 1000 mm/s, there are four blue tracers (blue dots); however, for the faster runs, it appears that there are more tracers. This is due to the higher acceleration, and is discussed in Section 5.2.

It can be seen from the graph that the velocity of a tracer does not have a strong impact on the accuracy of location. Although there is a slight correlation, the effect will not be noticed except at velocities higher than those normally found in PEPT experiments. It should be noted that a linear fit for each of the plots was used simply to show a general increasing or decreasing trend. Further experiments and analyses are needed to determine the for of the relationship between velocity and RMSE.

The RMSE ranges from 1.1mm to 1.8mm over the range of velocities, even though the resolution of the scanner is $4.8 \pm 0.2$ mm transaxially, and $5.6 \pm 0.5$ mm axially. Due to a combination of the fact that multiple LOR's are used per tracer and various smoothing and interpolating techniques, the tracers can in fact be located to a greater precision than the resolution of the scanner.

### 5.3.2 Tracer Count

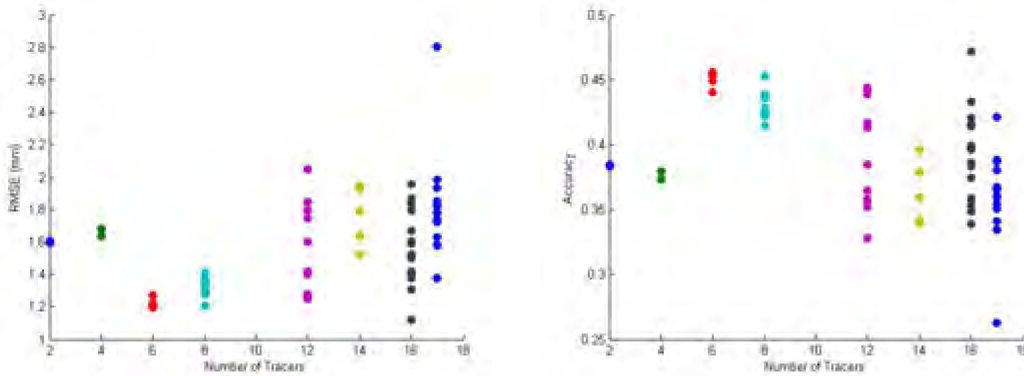



Figure 5.12: Plots showing the effect of the number of tracers on the accuracy of location.

It was originally assumed that an increase in the tracer count would cause a noticeable decrease in the accuracy of location of the VMPT algorithm. However, Figure 5.12 shows that this is not necessarily the case. The assumption was that more tracers would generate more LOR's, which could interfere with each other and cause the clustering process to be less precise. A possible explanation is that the interference may be somewhat symmetrical, and would average out over the field of view.

Although there is no clear relationship between tracer count and average accuracy, the number of tracers does seem to affect the *spread* in the accuracies of the tracks. There are several explanations for this:

- Since many tracers are used, there is a large variation in tracer velocities due to a constant rotational velocity with varying distances from the centre of rotation. It was shown previously that tracer velocity has some effect on the accuracy of location, and this could manifest as a large range of accuracies.
- With more tracers, there is a higher chance of track loss. This implies that, in general, continuous tracks may be shorter for larger tracer counts. Since the RMSE is analogous to a standard deviation for a fit, the more data points there are, the smaller the RMSE will be, and therefore the accuracy will be increased.

Although the accuracy does not seem to be affected much for tracer counts up to 20, this may not be the case for much higher counts, especially when the total activity within the field of view of the scanner approaches or exceeds the saturation level. When the scanner is saturated, the number of LOR's detected per tracer should theoretically decrease dramatically, leading to either a decrease in accuracy or an inability to track the tracers at all. However, for most experiments, the tracer count should not be enough for the scanner to become saturated.

### 5.3.3 Line Count

The number of lines of response used per frame during process is an important variable which can be changed by the user, but takes a default value of 100. It is not the total number of lines used per frame, but rather the number of lines per tracer; the total number of lines will then be $n_{lt} \times n_t$, where $n_{lt}$ is the number of lines per tracer, and $n_t$ is the number of tracers used in the experiment. Increasing the number of lines of response increases both the processing time as well as the amount of noise in the frame. However, if the number of lines of response per tracer is too low, the algorithm may not be able to detect the tracers.

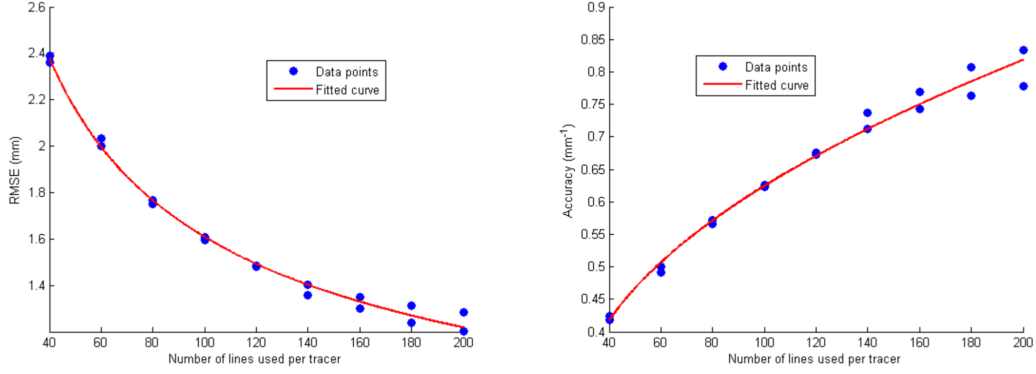



Figure 5.13: Plots showing effect on the number of lines of response used by the location algorithm on the accuracy of location.

Figure 5.13 shows the effect that the number of lines of response has on the accuracy of location. The sample experiments used two tracers for simplicity; using more tracers can lead to other variables affecting the accuracy. The spacing between discretized points along each LOR was set to $\delta_s = 5$ mm. In each instance of the location algorithm, an equal time interval between frames was used to eliminate the affect of acceleration on location. Specifically, the interval between frames was set to 100 lines of response (or $3.4 \pm 0.5$ms), regardless of the number of lines used per tracer. This technique is similar to the sliding windows scheme used to track fast-moving particles in a hydrocyclone **hydrocyclone** It leads to consecutive frames sharing some number of lines of response, but allows the user to decrease (or increase) the time interval between frames.

According to the curves of best fit, it appears that the error in location (RMSE) decreases with $1/\sqrt{n}$, where $n$ is the number of lines of response used per tracer. This is in agreement with findings from previous single-PEPT analyses. However, it is assumed that the accuracy would start to decrease if the number of lines were to be increased indefinitely. This is because, as $n_{lt}$ increases, the time interval over which those lines were generated becomes larger, and therefore so does the distance. The assumption is then that there is an inherent uncertainty associated with the position of the tracer. In fact, $n_{lt} = 40$ is the minimum number of lines per tracer that can be used to successfully track two tracers, given a spacing value of $\delta_s = 5$ mm.

### 5.3.4 Discretisation Separation Distance

The separation distance $\delta_s$ used in the algorithm (see Section 3.1.2, Equation 3.2), like the line count, could affect both the accuracy of location as well as the computational performance.

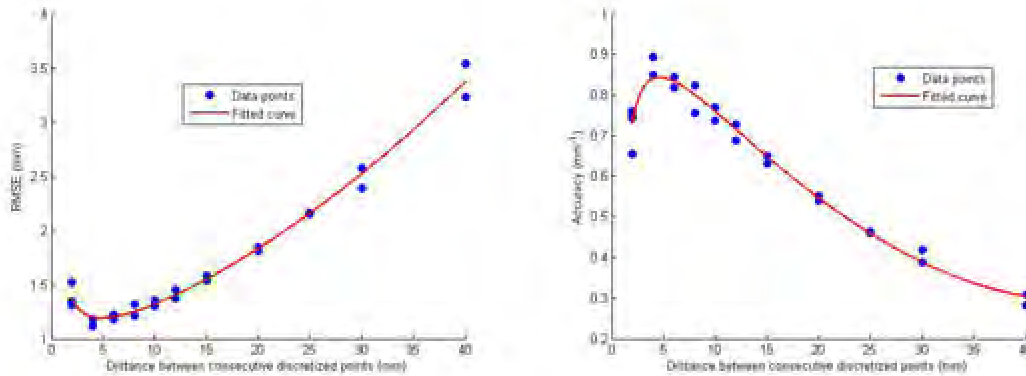



Figure 5.14: Plots showing effect of the discretization spacing used by the location algorithm on the accuracy of location.

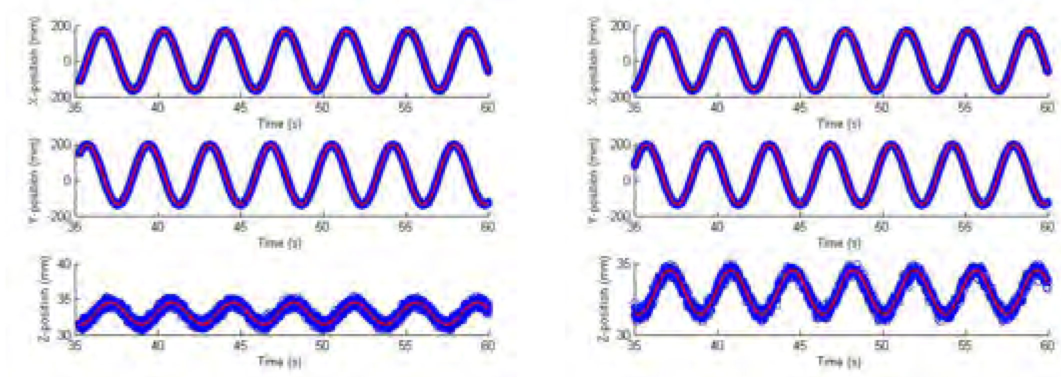
Figure 5.14 shows the effect of the discretization spacing on the accuracy of location and RMSE. A run using two tracers was used to analyse this effect, with 100 lines of response per tracer, for a total of 200.

It can be seen that the RMSE increases at a rate faster than linearly with the spacing; however, when the spacing is reduced to 2mm the RMSE sharply increases, and therefore the accuracy decreases. The line of best fit is of the form $f(\delta_s) \propto \delta_s^2 + 1/\delta_s$. This means that when $\delta_s$ (the separation distance) is small, the $1/\delta_s$ term dominates, and the RMSE increases quickly with decreasing $\delta_s$. When $\delta_s$ is larger than the turning point, the RMSE increases with *increasing* $\delta_s$, specifically with $\delta_s^2$. It must however be noted that the form of the fitted curve was estimated. Further investigation may be needed for a more precise form to be determined.

When the spacing was set to 2mm, some track loss occurred, and four tracks were detected when only two tracers were used. In fact, using any spacing below a value of 2mm renders the algorithm unable to locate the tracers at all. For two tracers with 100 lines of response per tracer, the optimum spacing is about 4mm.

It should be noted that the effect of separation distance and that of line count per tracer are intimately related: they both contribute to the overall number of seed points used in the Voronoi tessellation.

### Comparison to Birmingham Algorithm



(a) Best fit found using tracks generated by VMPT algorithm.

(b) Best fit found using tracks generated by CTRACK algorithm.

Figure 5.15: Comparison between tracks generated by CTRACK and VMPT algorithms.

The Birmingham CTRACK algorithm is the most commonly used algorithm to track a single tracer with PEPT. To compare the VMPT algorithm to CTRACK, a single tracer moving in a controlled circular motion was tracked. A best fit curve was found for both cases, and the RMSE calculated.

| | A | B | omega | theta |
|---|---|---|---|---|
| x | 2.78 | 166.6 | 1.6996 | 4.19 |
| y | 29.05 | 166.7 | 1.6995 | 2.62 |
| z | 32.96 | -1.53 | 1.6992 | 1.82 |
| RMSE | | 1.2615 | | |

Table 5.3: Equation of best fit coefficients for VMPT tracking.

| | A | B | omega | theta |
|---|---|---|---|---|
| x | 2.77 | 166.6 | 1.6996 | 4.19 |
| y | 29.09 | 166.8 | 1.6995 | 2.62 |
| z | 32.96 | -1.54 | 1.6994 | 1.84 |
| RMSE | | 0.92012 | | |

Table 5.4: Equation of best fit coefficients for CTRACK tracking.

Figure 5.15 compares the tracks generated by the VMPT algorithm to the CTRACK algorithm. The curves of best fit are defined in Tables 5.3 and 5.4, where the coefficients $A, B, \omega$ and $\phi$ correspond to those in the best fit equation

$$x = A_x + B_x \sin(\omega t - \phi_x) \tag{5.8}$$

for each dimension $x, y, z$.

As can be seen, the coefficients match closely between the curves fit to the CTRACK and VMPT algorithms. Comparing the two RMSE values, we have 0.92 and 1.26 mm for the CTRACK and VMPT algorithms respectively. This indicates that the CTRACK algorithm has a higher degree of precision than the VMPT algorithm. This is to be expected, since CTRACK has been refined since its inception, and it was designed specifically for single tracer tracking. It is however encouraging to find that the VMPT RMSE is of a similar order of magnitude to the CTRACK RMSE.

## 5.4 Erratic Rotation

The RMSE, and therefore accuracy, could not be determined for these data sets since the tracers were not moving according to a defined function. However, as a proof of concept the uncontrolled rotation experiments were necessary to show that the tracking algorithm could still function and reliably track tracers experiencing erratic speeds and accelerations.

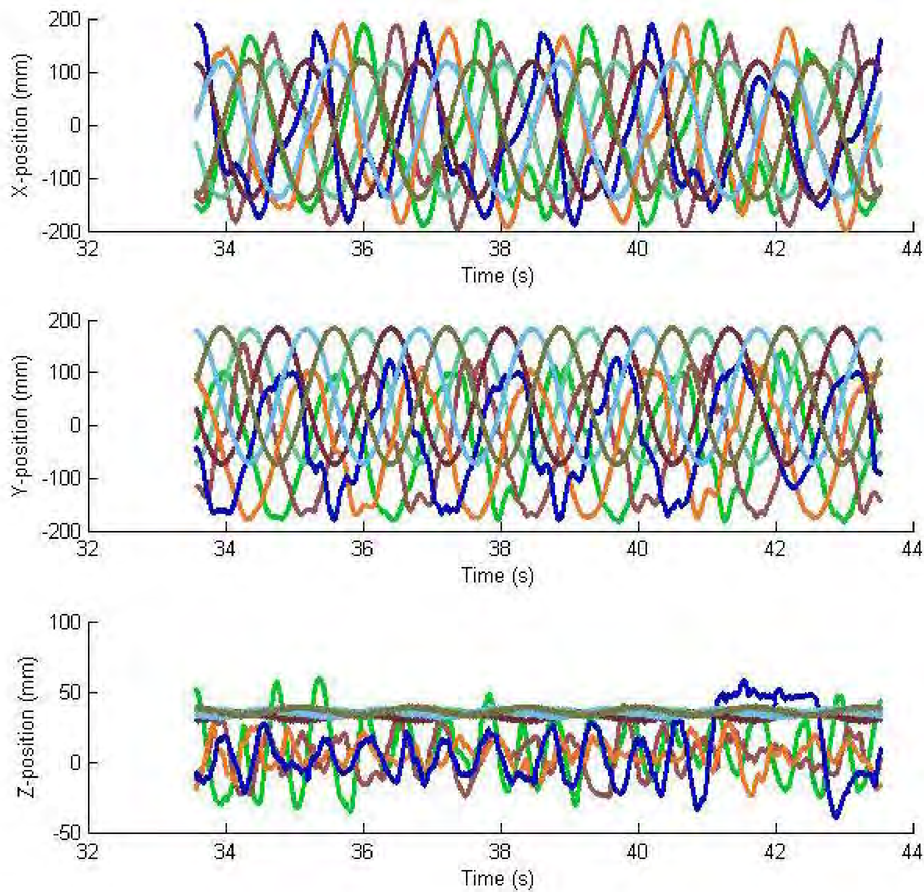



Figure 5.16: Plots showing motion of 8 tracers, 4 of which experience controlled circular motion, and 4 of which experienced semi-chaotic motion.

Figure 5.16 shows the paths of eight tracers. It is clear from the graphs that four tracers are moving in a controlled circular path. These four tracers are indicated by the tracks which follow well-defined sine curves. The four controlled tracers were used to compare the uncontrolled motion to a known path, and to verify that the algorithm was working properly.

The other four tracks exhibit erratic and uncontrolled behaviour; these tracks belong to four tracers attached to the disc via a rubber band. These tracks indicate this motion clearly: the overarching motion is a sine wave similar to the controlled tracers. This is due to the fact that the tracers are connected to the rigid disc, which is moving in a circle. On top of this, the tracer swings from sided to side, which appears in the track as noise along the curve, but which may be considered a second sine curve. The total motion is therefore compounded of multiple sine curves. Another major difference between the controlled and uncontrolled tracers is that the uncontrolled tracers show pronounced motion in the $z$-direction. The discs were aligned such that motion in the $z$-direction was minimised, so that the majority of the displacement in controlled tracers occurred in the $xy$-plane. The uncontrolled tracers did not have this constraint, which allows for significant displacement in the $z$-direction.

## 5.5 Computational Performance

Computational performance is an important metric to determine whether the tracking technique is practical to use. The two most important aspects of performance are the memory required to process one frame of data, and the time taken to process that frame. Section 3.1.4 describes the possibility of parallelizing the location algorithm in order to decrease the processing time. Ideally, the VMPT algorithm could be run on as many cores and/or machines as possible, or the code could be re-structured to run on a massively parallel graphics processing unit (GPU). The limiting factor to this is the memory requirement. GPU's generally do not have large amounts of memory, and so running the algorithm on a GPU may prove to be infeasible.

All processing was performed on an Intel Core i7-4500U CPU 1.8-2.4 GHz with 5.72 GB usable RAM on a 64-bit, x64-based copy of Windows 10.

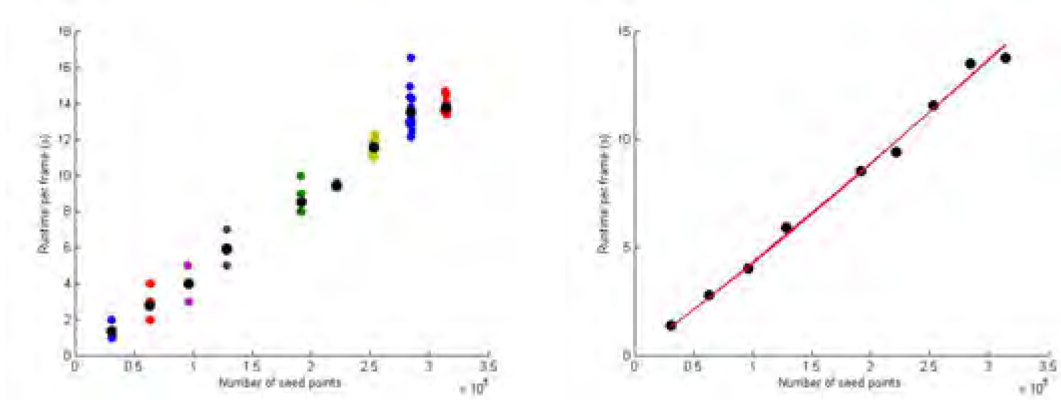
### 5.5.1 Runtime

The runtime per frame of the algorithm is defined as the total time to process some amount of data divided by the number of frames processed. The runtime of the algorithm is dominated by the Voronoi tessellation, which has a complexity of $O(n\log n)$ at best, where $n$ is the number of seed points. Since the number of seed points is often of the order of tens or hundreds of thousands, processing times can be fairly large.

Other parts of the algorithm have relatively small runtimes compared to the tessellation:

- The initial discretisation of the lines runs in $O(n)$ time, where $n$ is the number of lines used per frame. In this case, $n$ is usually no more than 2000.
- When finding the point along each line with the smallest surrounding Voronoi cell, the $n$ is the same as that used during the tessellation. However, since it is simply an issue of finding a minimum of a set, it runs in $O(n)$ time.
- Although the LOF-smoothing and DBSCAN-clustering algorithms run optimally in $O(n\log n)$ time, the number of data points used is $n \leq n_l$, where $n_l$ is the number of lines of response used in that frame.

The processing time was measured using the *tic toc* functionality in Matlab. In a Matlab script, the *tic* command starts a timer, which is stop when the *toc* command is encountered. The timer encapsulated all the code devoted to processing a single frame; the processing time for that frame was then stored in a vector. After a predetermined number of frames had been processed, the vector containing these times was written to file for later analysis.




(a) Scatter plot showing the relationship between the average number of seed points and the processing time per frame (using a line count per frame of 100, and a separation distance of 5)

(b) Function of form $n\log n$ fitted to scatter plot of average processing time and average number of seed points.

Figure 5.17: Graphs showing the time taken to process a single frame of data based on the number of seed points used during tessellation.

Figure 5.17 shows the processing time as a function of the number of seed points used in the Voronoi tessellation. From the description of common Voronoi algorithms described in Section 2.3.3, the average complexity of a Voronoi algorithm is $O(n\log n)$. To find a best fit function, the equation $f(n) = n\log n$ was first linearised, so that a straight line could be fit to a plot of $\frac{t}{\log n}$ vs $n$, where $t$ is the time taken to process one frame of data, and $n$ is the number of seed points used to generate the Voronoi tessellation for that frame. The best fit line takes the form

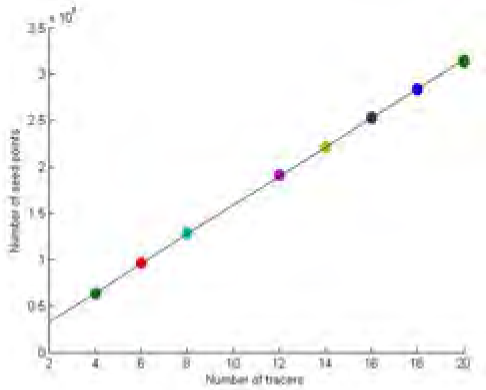
$$\frac{t}{\log n} = 0.3562 \times 10^{-5} n + 0.0137 \tag{5.9}$$

which can be rearranged to give

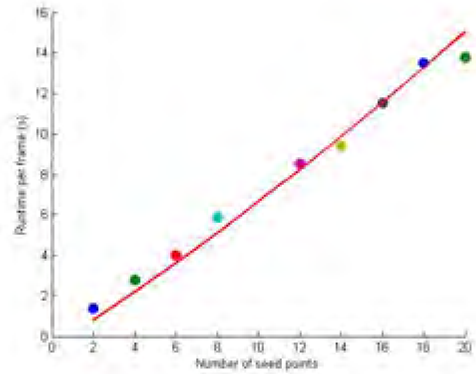
$$t = (0.3562 \times 10^{-5} n + 0.0137)\log n. \tag{5.10}$$

When Equation 5.10 is plotted, we get the graph shown in Figure 5.17b. It can be seen that the fit closely matches the measured data, verifying the theoretical assumption that the runtime increases with $n\log n$.

The number of seed points is however not an intuitive value to use when estimating the processing time of the algorithm. Instead, we can find a correlation between the number of tracers used in an experiment and the number of seed points generated by the tracers. Figure 5.18a shows the average number of seed points generated for various tracer counts. It can be seen that a linear plot fits this relationship very well Based on the relationship between the processing time and the number of seed points, we can assume that the relationship between *tracer count* and processing time is similarly of the order of $O(n_t\log n_t)$, where $n_t$ is the number of tracers used in the experiment. Figure 5.18b verifies this assumption; the best fit function shows a remarkably similar trend to that found in Figure 5.17b.




(a) Average number of seed points used in Voronoi tessellation




(b) Plot showing relationship between the number of tracers used in an experiment and the processing time per frame (using a line count per frame of 100, and a separation distance of 5)

Figure 5.18: Graphs showing the time taken to process a single frame of data based on the number of seed points used during tessellation.

### 5.5.2 Memory Usage

As mentioned previously, memory usage can be viewed as the limiting factor to the parallelization of the algorithm. It is therefore important to quantify the memory used in order to plan for further developments to the algorithm.

As with the processing time, the memory usage is dominated by the Voronoi tessellation itself. As described in Section 2.3.3, the theoretical memory usage of a tessellation is $O(n^{d/2})$, where $d$ is the dimensionality of the problem. Since this problem is inherently of a 3-dimensional nature, we should expect the memory used to increase with $O(n^{1.5})$.

Memory usage of a function is more difficult to track than the processing time. Matlab does not have an inbuilt function to track memory usage over time, and while there are tools available for download for Windows, they did not seem to provide the functionality necessary. To track the memory, a bash script was written, which was run in the *cygwin* command line for Windows. The script took a snapshot of the data shown in the Windows Task Manager at 1 s intervals.

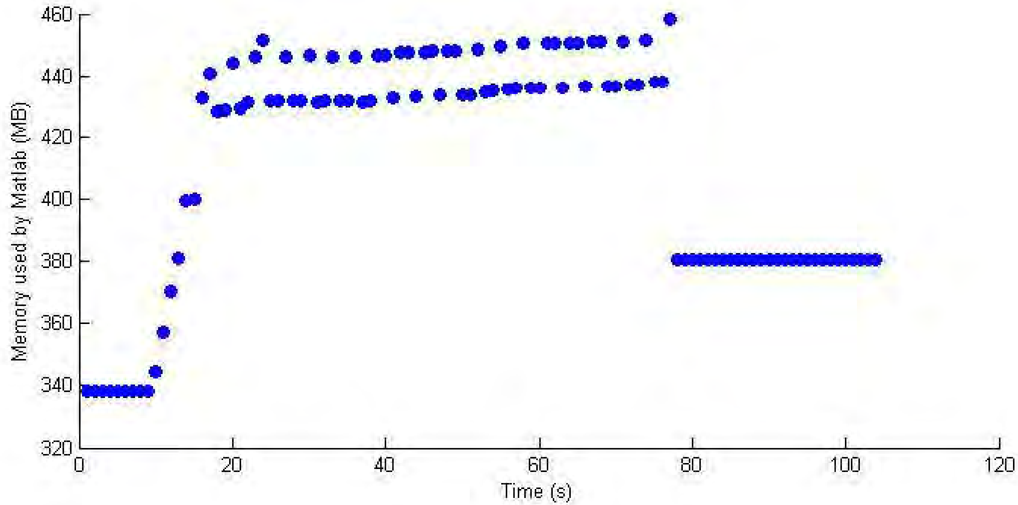
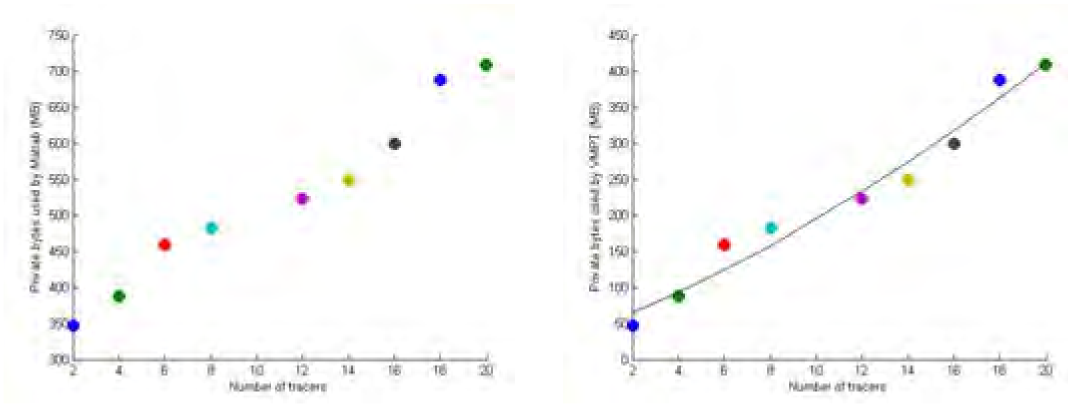



Figure 5.19: Memory used by Matlab while processing frames of data generated by experiments using six tracers.

Figure 5.19 shows this memory usage over time. The usage starts at roughly 340 MB, and increases sharply at around the 10 s mark. Between the 18 s and 75 s marks, the memory usage alternates between about 430 MB and 450 MB, all while slowly increasing. The highest usage level shown during this period, at about 450 MB, indicates the amount of memory used during the actual Voronoi tessellation, while the lower level indicates the periods after the tessellation has completed, but while all of the output from the tessellation is stored in memory. Finally, the usage drops sharply when the program terminates. After termination, the usage is higher than it was before initialisation. This is because Matlab retains some of the data used in the function. After a period of time, or once a new function or script is started, the data stored like this is removed from memory.

Since the memory is considered a limiting factor, the maximum usage is the most important aspect. Figure 5.20 shows the maximum memory used while processing data generated by experiments using varying numbers of tracers. The maximum memory usage of the VMPT algorithm is technically a function of the number of seed points used in the Voronoi tessellation. However, the number of seeds is difficult to control, so the relationship between the number of seeds and the number of tracers is used to infer the memory usage as a function of tracers count, which is indicated in Figure 5.20.

Note that Figures 5.20a and 5.20a show the memory used by Matlab, which is more than the memory used by the VMPT algorithm. Matlab itself has an overhead of roughly 300 MB, so Figure 5.20b shows the memory usage adjusted for this overhead. It also shows a curve of the form $M = n_t^{1.5}$, where $M$ is the memory usage and $n_t$ is the number of tracers. This curve was fit using a similar linearisation method to that used in Figure 5.17b. The data shows a close fit to the curve, thus verifying the theoretical memory usage.




(a) Memory used by Matlab while performing VMPT processing.

(b) Memory used by VMPT algorithm, adjusted for base Matlab usage. Fitted curve is of form $y = x^{3/2}$

Figure 5.20: Maximum memory used when performing VMPT processing versus number of tracers used in experiments.

# Chapter 6

# Alternative Applications

Because efficient, reliable multiple particle tracking algorithms have not been thoroughly explored, the range of possible applications of multiple particles tracking has similarly not been determined.

## 6.1 Solid Deformation

An unexplored application of multiple particle tracking in PEPT is the tracking of a deforming solid. Theoretically, if tracers are placed at points on a solid object, the positions of the tracers can be monitored over time. The tracers could represent nodes of a mesh in a finite element code, and using this, information about the deformation can be gleaned. In cases where the material properties are known, information such as stresses and strains, as well as strain rates can be calculated using a finite element method. Conversely, if the forces acting on the material are known, an inverse-FEM method could be used to determine the material properties of the deforming object. This could be particularly useful if the properties are non-linear or strain-rate dependent.

A simple experiment was performed to provide proof of concept for this application. Section 4.2.5 describes the experiment in detail. The goal of the experiments is to calculate the Young's modulus $E$ of the steel using the measurements from the PEPT experiments. Two different configurations were used: one in which the load was placed at the end of the rod, and the other in which it was placed between the end and the clamp.

The deflection at a point along a cantilever loaded at the end is given by

$$\delta_y(x) = \frac{Px^2}{6EI}(3L - x) \tag{6.1}$$

where $\delta_y(x)$ is the vertical deflection at a position $x$ along the cantilever; $P$ is the load; $E$ is the Young's modulus; $I$ is the second moment of area of the cantilever; and $L$ is the length of the cantilever.

Rearranging Equation 6.1 and substituting $P = mg$ gives us

$$E = \frac{mgx^2}{6\delta_y I}(3L - x). \tag{6.2}$$

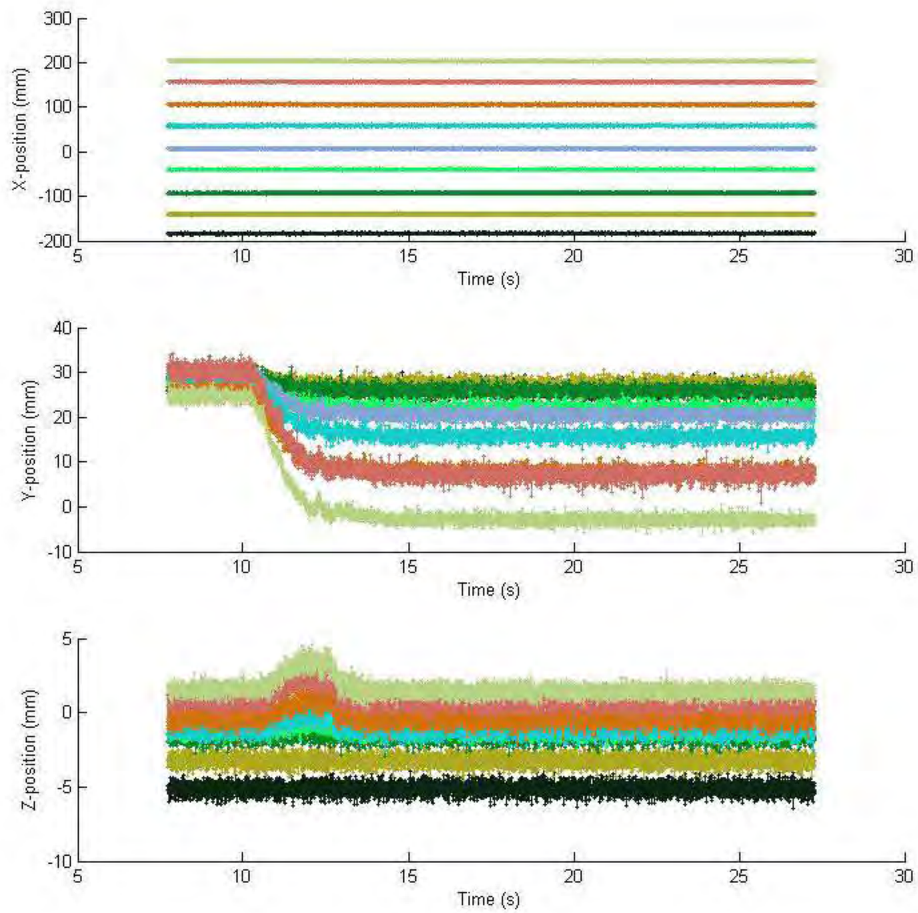



Figure 6.1: Deflection over time of 9 tracers placed along an end-loaded cantilever.

Figure 6.1 shows the motion of the tracers, or 'nodes' over the timeframe of the experiment. Nine tracers were used, placed at regular intervals along the cantilever; it can be seen that the load was gradually applied between the 10 and 15 second marks.

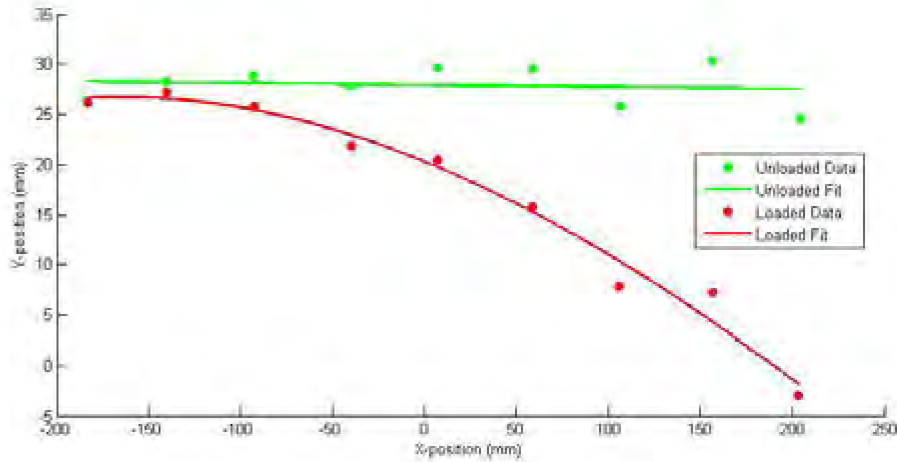



Figure 6.2: Positions of tracers detected by the algorithm, both before and after loading.

Figure 6.2 shows the unloaded and loaded positions of the cantilever, with curves fitted to the data. A linear fit was applied to the unloaded data, since the cantilever was modelled as straight, while a cubic fit was applied to the loaded data since the deflection is proportional to the cube of the position along the length of the cantilever. The two fits were used to find the Young's moduli at various values of $x$; the locations of the nodes (tracers) were used for these $x$-values. So let $y_u(x)$ be the linear fit equation for the unloaded data set, and $y_d(x)$ be the cubic fit equation for the loaded data set. Now we have

$$\delta_y(x_i) = y_u(x_i) - y_d(x_i) \tag{6.3}$$

for $i = 1...n_t$ where $n_t$ is the number of tracers used in the experiment. Note that the Young's modulus at the point of zero deflection (at the clamp) cannot be calculated, since $\delta_y = 0$ and Equation 6.2 becomes undefined.

The equations of best fit are calculated as follows:

$$y_u(x) = (-0.002 \pm 0.129)x + (27.95 \pm 1.64) \tag{6.4}$$

$$y_d(x) = (12.5 \pm 1.2) \times 10^{-6}x^3 - (1.9 \pm 1.3) \times 10^{-4}x^2 - (7.5 \pm 3.5)^{-2}x + (20 \pm 2.6). \tag{6.5}$$

These were used to calculate the $y$-positions of the tracers before and after loading.

Table 6.1 shows the Young's moduli calculated at eight different positions along the cantilever. These positions correspond to the locations the tracers placed along the cantilever. The table shows that most of the calculated Young's moduli fall within acceptable ranges for various steels (120 to 180 GPa); only the values calculated for the first and second nodes are unexpected. Looking at the graphs in Figure 6.2 we can see that the length of the cantilever between nodes 1 and 2 has an almost constant displacement of 1.6mm, while the displacement at node 1 *should* be 0mm. This is simply a result of imperfect data. For this reason, when analysing the Young's moduli, $x$-values between nodes 1 and 2 are disregarded.

Because the fitted function is continuous, we can sample it at any value for $x$. With enough samples, the mean and standard deviation of the Young's modulus can be found. Taking 1000 samples, calculating the Young's moduli for each, and finding the mean gives us

$$E = (161 \pm 15)\text{GPa}, \tag{6.6}$$

which is a reasonable value for the Young's modulus for stainless steel.

These results show that deformation tracking using multiple-PEPT is possible. Although the experiment was crude, a reasonable value for the Young's modulus of stainless steel was determined.

Table 6.1: Tabel showing Young's Moduli calculated at positions along a cantilever defined by the location of the tracers.

| Distance from clamp (mm) | Displacement (mm) | Young's Modulus (GPa) |
|---:|---:|---:|
| 0 | 1.6 | 0 |
| 43.2 | 1.6 | 51.6 |
| 90.7 | 2.7 | 132.0 |
| 144.0 | 5.1 | 166.1 |
| 190.8 | 8.2 | 172.7 |
| 242.3 | 12.6 | 171.8 |
| 290.5 | 17.5 | 168.1 |
| 341.2 | 23.5 | 162.9 |
| 388.0 | 29.6 | 157.6 |

# Chapter 7

# Conclusions

## 7.1 Summary of Findings

An algorithm, dubbed the Voronoi-based Multiple Particle Tracking (VMPT) algorithm, was developed. It used a combination of pre-existing algorithmic techniques, such as Voronoi tessellation, clustering, outlier-removal and multiple target tracking, to perform the task of tracking multiple tracers within a PEPT environment. Up to 20 $^{22}$Na tracers were successfully tracked. However, this may not be the upper limit, since there was a lack of $^{22}$Na with which to manufacture more tracers.

### 7.1.1 Accuracy

A series of experiments were performed to test the accuracy, robustness, and computational performance of the algorithm. To determine the accuracy to which a tracer can be located, tracers were made to rotate in a circle within the field of view of the PEPT scanner. When a plot is generated of position versus time for each of the $x, y, z$ directions, the circular motion means that we get three separate sine curves. Once a best-fit curve is fitted to the position-time data, the root mean square error (RMSE) of the data set can be calculated. The RMSE is analogous to the $R^2$ value of a linear best fit, or the standard deviation of a Gaussian plot, and in this thesis was used as a measure of the resolution of the VMPT algorithm.

It was found that a RMSE value between 1.2 and 2 mm was fairly easily achievable. The parameters of interest when considering the accuracy fall into two groups: natural and artificial parameters. Natural parameters are those that are a fundamental part of the physical system, and cannot be controlled artificially by the user of the algorithm. The two natural parameters investigated were tracer velocity and tracer count. Other parameters were considered, but these were considered the two (possibly) most influential. Interestingly, the RMSE was not found to be dependent on the number of tracers being tracked. It was however linearly dependent on the velocity of the tracer in question, but the gradient of the best fit line was small, so the effect of velocity on RMSE can be largely ignored for most experiments.

Artificial parameters are those that can be changed within the algorithm. The two most important artificial parameters identified were the number of LOR's used per tracer for each frame $n_{lt}$, and the spacing between discrete points on the LOR's $\delta_s$. Together, these two parameters describe the degree of discretisation of the space within the field of view. It was found that the RMSE is highly dependent on both the line count and the separation distance. As the line count per tracer increases, the RMSE decreases (and the accuracy therefore increases). Over the range observed the decreases proportionally to $1/\sqrt{n}$, where $n$ is the number of lines of response per tracer. This is in agreement with previous findings in single PEPT. However, this may not be the case if higher line counts were inspected. It is assumed that after a point the RMSE will start to increase due to the set of lines covering a large time interval. This is however as yet untested. The RMSE curve shows a very definite optimal value $\delta_{s,opt}$, which was found to be 4 mm in the case where two tracers are tracked with a line count of 100

lines per tracer. For $\delta_s > \delta_{s,opt}$ the RMSE increases with separation distance. The rate of increase is faster than linear, although it is unknown whether a logarithmic, exponential or polynomial rate best describes the relationship. However, for $\delta_s < \delta_{s,opt}$ the RMSE increases rapidly until the algorithm fails entirely to locate the any tracers.

As a comparison to the well-established Birmingham CTRACK algorithm, an experiment was performed using only one tracer. It was tracked using both the CTRACK and VMPT algorithms individually, and the RMSE's for the results were compared. An RMSE of 0.92 mm was determined for the CTRACK tracking method, while the RMSE from the VMPT method was 1.26 mm, which is an increase of 37 %. This is a positive result, since in terms of accuracy it means that the VMPT algorithm is a competitive method for low tracer counts, while also providing the means to track larger numbers of tracers.

### 7.1.2 Robustness

The factors which were identified as defining robustness are chance of track loss, need for human intervention with the algorithm, and the chance that the program will encounter situations which lead to crashes.

Track loss is when an existing track generated by a tracer is terminated for one or more of several reasons:

- tracers exiting the field of view will be lost.
- high tracer accelerations lead to the tracking algorithm failing to accurately predict the locations of tracers because a linear extrapolation technique is used.
- scattering and attenuation effects result in photons being undetectable by the detector cells. When a tracer moves near the vicinity of a high-density material its trajectory could therefore be lost.
- two or more tracers coming within a close proximity of each other will be detected as a single tracer. Once they move apart they will once more be detected individually. The 'merged' tracer may be added to any one of the pre-existing tracks, although it is impossible to determine which.

If track loss occurs the track is completely terminated and either saved to disc or deleted, depending on the number of position-time entries. Since each tracer is uniquely identified purely based on its existing trajectory, when a lost tracer reappears, it is detected as an entirely new object. Although a high tracer count does not directly cause track loss, it does increase the chance.

A major benefit of the VMPT algorithm over other multiple-PEPT techniques is that it requires very little knowledge of or restrictions on the physical system. Previous techniques require that all the tracers be within the field of view of the scanner at all times. If a tracer left the field of view, the program would have to be stopped and the number of tracers redefined. With the VMPT algorithm, only the maximum number of tracers within the field of view must be known. The only value that is dependent on the tracer count is the number of LOR's per frame. Once this has been defined, tracers may exit and enter the field of view freely with no need for human intervention.

Another constraint which most previous methods require is that no tracers collide with each other. This requires a physical constraint on the relative motion between tracers. For most experiments, this constraint decreases the chance that the experiment can accurately represent the true physical behaviour. The VMPT algorithm requires no such constraint. In the case of tracer collision, the actual point of collision can be removed manually in post-processing, but it has no effect on the running of the program.

### 7.1.3 Computational Performance

The computational performance of any algorithm determines its viability. Two aspects of the VMPT algorithm were analysed, namely the processing time and the memory usage. Both aspects were found

to be dominated by the Voronoi tessellation, and were dependent on the number of seed points. It was shown that the number of seed points was directly related to the number of tracers used in the experiment. A relationship could therefore easily be found between the number of tracers and the computational performance. This is a more useful relationship than that between the number of seed points and the performance, since the tracer count can be easily controlled.

It was found that the complexity of both processing time and memory usage closely matched the theoretically expected complexities. Specifically, the processing time had a complexity of $O(n\log n)$ while the memory usage had a complexity of $O(n^{3/2})$, where $n$ is the number of seeds points. Using $\delta_s = 4$ and $n_{lt} = 100$ the memory usage while tracking two tracers was about 60 MB, and each frame took just over 1 s to process. With the same parameters but tracking 20 tracers, the memory usage was 400 MB, and the processing time per frame was about 15 s.

This performance is prohibitive when tracking data from longer experiments. A reasonable time interval between frames is about 25 s. Assuming this 25 s time interval and the settings described above, it would take 40 hours to process 1 hour of experimental time, while tracking 20 tracers would take 600 hours.

Although these processing times are too long for the algorithm to be practical, the code was designed so that it is highly parallelisable. Each frame is independent of every other frame, and can therefore be processed simultaneously. Furthermore, the raw list mode data from the PET scanner can be segmented and distributed across multiple separate machines. Once these segments have been processed, the location data can be combined into one file. This is possible because the input to the tracking algorithm does not have to be sequential with regards to time stamps; the tracking algorithm itself sorts the locations according to time.

## 7.2 Applications of Multiple-PEPT

A novel application of multiple particle tracking, namely deformation tracking, was explored in this thesis. Tracers were placed at nodes along a deformable cantilever, and the deformation due to an applied force was tracked with the VMPT algorithm. Using deflecting beam theory, the Young's modulus of the cantilever was determined to be $(161 \pm 15)$GPa, which is typical of stainless steel. As a proof of concept, this indicated that tracking deformation of a solid object is viable using multiple-PEPT. Some other possible applications of deformation tracking are:

- using portable PEPT scanners to observe deformation inside machinery on site.
- placing tracers on the exterior of a body part during a PET scan. This allows for motion correction of the PET data.
- an inverse finite element method with tracers at the nodes can be coupled with computational fluid dynamics to determine non-linear or strain-rate dependent material properties.

Another obvious application of multiple particle tracking is simply to decrease experimental time. When performing a PEPT experiment, it is often necessary to have runs lasting upwards of two hours in order to gain enough data to assume that equilibrium has been reached. As more tracers are added to the experiment, the amount of position-time data points increases per time frame. This means that the total experimental time can be reduced while still generating a significant amount of data.

An application which has been touched on by Yang *et al.* [23] is tracking the rotation of a solid body in motion. In a standard PEPT experiment, the translational motion of a particle is studied. From this, inferences about energies, stresses, strains and forces can be made. However, no information can be gleaned about the rotational energy in a system. By placing three tracers at strategic points on a solid body, the three dimensional rotation can be tracked, providing a more complete picture of the kinematics and kinetics of the system.

In the field of granular fluid flow, complex interacting fluids can be studied using multiple-PEPT. If a system contains two or more distinct granular fluids or regions of flow, tracers can be manufactured

such that they have similar characteristics to the surrounding fluid. The interaction of these fluids can then be studied by observing the motion of the tracers.

## 7.3 Future Developments

The first step in further developing the VMPT algorithm is to fully parallelise the code. It is recommended that the new code be written in Python, for multiple reasons:

- Python is freeware and multi-platform, allowing the code to be run on any machine.
- Python uses the qhull Voronoi tessellation scheme, which is the same one that Matlab uses. It is light and compact as well as fast and efficient.
- Libraries such as NumPy and SciPy provide similar matrix functionality to Matlab.
- Python provides interfaces so that C/C++ functions may be implemented in the main script.

It is recommended that the function to find the smallest Voronoi cell per LOR be written in C. The function makes use of multiple loops, and since Python is an interpreted scripting language, meaning for- and while-loops are inefficient. C/C++ on the other hand is compiled, so loops are highly efficient. C can also be used for the LOF and DBSCAN functions (if no existing libraries exist) although it is less necessary since by this stage the number of points will have been decreased significantly. Another option is to once again attempt to rewrite the code completely in a low level language such as C++. This will mean developing a custom implementation for the program to interface with the qhull Voronoi tessellation libraries. In addition to this, the effectiveness of using the *mean vertex distance* described in Section 3.1.2 as a tool to compare relative polyhedron sizes can be more fully explored. This can be done by generating a large number of polyhedra, and calculating the normalized volumes and the normalized mean vertex distances for each of them. Although this would not be a substitute for the true volume, it could theoretically be used as a comparison tool.

Some changes can be made to various aspects of the algorithm itself. In the description of the location algorithm, after the LOF outlier-removal step, the 'large' Voronoi cells are removed by finding the mean volume of the cells and deleting those which have a volume larger than this mean. However, it is shown that a histogram of these volumes forms a clear bimodal distribution. If these two distributions can be identified and separated, the 'large' cells can be removed in a more robust, and possibly accurate, manner.

In the tracking stage of the algorithm, a possible major change would be to implement a quadratic extrapolation in order to predict the next location of each tracer. This would theoretically allow for more robust tracking of tracers experiencing higher accelerations. A tracking method that more closely matches commonly-used multiple target tracking algorithms can also be implemented. In particular, most multiple target tracking algorithms make use of Kalman filters, which eliminate the need to store a location history and use a weighting technique for prediction. Implementing a Kalman filter properly takes some investigation into the covariance of the system, which is a measure of the estimated uncertainty of the state prediction.

Further research can be used to more fully quantify the robustness and accuracy of the algorithm. Experiments could be performed to determine the distance between two tracers before they are detected as a single tracer. These could involve two (or more) tracers slowly being brought into contact with each other. It is assumed that this separation distance would be dependent on the total tracer count, so this would have to be taken into account when planning the experiments. The form of the curve describing the relationship between discretization spacing $\delta_s$ and the RMSE can also be determined using more experiments.

Other applications of the code can be more thoroughly explored. Rotation of solids is of particular interest, especially in the mining industry. A set of incremental experiments can be performed to test the viability of multiple-PEPT as a means to measure 3D rotation. Ideally the initial experiments would

start with large objects rotating on three axes in a controlled, predictable manner. The experiments could then evolve by decreasing the size of the object and letting it move in an erratic manner.

Finally the application of PEPT to the medical industry has been under-explored. As mentioned above, it can be used in conjunction with PET to provide motion correction during scanning. Multiple-PEPT can also theoretically be used to determine material properties of biological materials *in situ*. These properties have historically proven difficult to determine since most biological materials have highly non-linear and strain rate dependent properties. Furthermore, it would be ideal to take the necessary measurements on living tissue to ensure more accurate results. Multiple-PEPT could provide the means to do this successfully.

# Bibliography


[1] W. H. Sweet, "The use of nuclear disintegration in diagnosis and treatment of brain tumors.," *New England Journal of Medicine*, vol. 245, pp. 875–878, 1951.

[2] F. R. Wrenn, L. Myron, and P. Handler, "The use of positron-emitting radioisotopes for the localization of brain tumors.," *Science*, vol. 113, pp. 525–527, 2940 1951.

[3] D. E. Kuhl and R. Q. Edwards, "Image separation radioisotope scanning 1," *Radiology*, vol. 80, no. 4, pp. 653–662, 1963.

[4] M. M. Ter-Pogossian, M. E. Phelps, E. J. Hoffman, and N. A. Mullani, "A positron-emission transaxial tomograph for nuclear imaging (pett) 1," *Radiology*, vol. 114, no. 1, pp. 89–98, 1975.

[5] C. Burnham and G. Brownell, "A multi-crystal positron camera," *Nuclear Science, IEEE Transactions on*, vol. 19, no. 3, pp. 201–205, 1972.

[6] T. Ido, C.-N.Wan, V. Casella, J. S. Fowler, A. P.Wolf, M. Reivich, and D. E. Kuhl, "Labelled 2-deoxy-d-glucose analogs. 18f-labeled 2-deoxy-2-uoro-d-glucose, 2-deoxy-2-uoro-d-mannose and 14c-2-deoxy-2-uoro-d-glucose.," *Journal of Labelled Compounds and Radiopharmaceuticals,*, vol. 14, pp. 175–183, 2 1978.

[7] F. H. Fahey, "Data acquisition in pet imaging," *Journal of nuclear medicine technology*, vol. 30, no. 2, pp. 39–49, 2002.

[8] G. B. Saha, "Basics of pet imaging: Physics," *Chemistry, and Regulations. Springer Verlag*, 2005.

[9] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the royal statistical society. Series B (methodological)*, pp. 1–38, 1977.

[10] H. M. Hudson and R. S. Larkin, "Accelerated image reconstruction using ordered subsets of projection data," *Medical Imaging, IEEE Transactions on*, vol. 13, no. 4, pp. 601–609, 1994.

[11] K. Cole, A. Buffler, N. van der Meulen, J. Cilliers, J. Franzidis, I. Govender, C. Liu, and M. van Heerden, "Positron emission particle tracking measurements with 50 micron tracers," *Chemical Engineering Science*, vol. 75, pp. 235–242, 2012.

[12] M. Hawkesworth, D. Parker, P. Fowles, J. Crilly, N. Jefferies, and G. Jonkers, "Nonmedical applications of a positron camera," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 310, no. 1, pp. 423–434, 1991.

[13] D. Parker, C. Broadbent, P. Fowles, M. Hawkesworth, and P. McNeil, "Positron emission particle tracking-a technique for studying flow within engineering equipment," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 326, no. 3, pp. 592–607, 1993.

[14] A. Buffler, I. Govender, J. Cilliers, D. Parker, J. Franzidis, A. Mainza, R. Newman, M. Powell, and A. Van der Westhuizen, "Pept cape town: A new positron emission particle tracking facility at ithemba labs," in *Proceedings of International Topical Meeting on Nuclear Research Applications and Utilization of Accelerators*, vol. 4, 2009.

[15] T. Spinks, T. Jones, P. Bloomfield, D. Bailey, M. Miller, D. Hogg, W. Jones, K. Vaigneur, J. Reed, J. Young, *et al.*, “Physical characteristics of the ecat exact3d positron tomograph,” *Physics in medicine and biology*, vol. 45, no. 9, p. 2601, 2000.

[16] D. Parker, A. Dijkstra, T. Martin, and J. Seville, “Positron emission particle tracking studies of spherical particle motion in rotating drums,” *Chemical Engineering Science*, vol. 52, no. 13, pp. 2011–2022, 1997.

[17] Y.-F. Chang, C. Ilea, Ø. Aasen, and A. Hoffmann, “Particle flow in a hydrocyclone investigated by positron emission particle tracking,” *Chemical Engineering Science*, vol. 66, no. 18, pp. 4203–4211, 2011.

[18] B. Hoomans, J. Kuipers, M. M. Salleh, M. Stein, and J. Seville, “Experimental validation of granular dynamics simulations of gas-fluidised beds with homogenous in-flow conditions using positron emission particle tracking,” *Powder Technology*, vol. 116, no. 2, pp. 166–177, 2001.

[19] X. Fan, D. Parker, and M. Smith, “Labelling a single particle for positron emission particle tracking using direct activation and ion-exchange techniques,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 562, no. 1, pp. 345–350, 2006.

[20] ——, “Enhancing 18 f uptake in a single particle for positron emission particle tracking through modification of solid surface chemistry,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 558, no. 2, pp. 542–546, 2006.

[21] O. Gundogdu, “Positron emission tomography particle tracking using cluster analysis,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 534, no. 3, pp. 562–576, 2004.

[22] J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Oakland, CA, USA., vol. 1, 1967, pp. 281–297.

[23] Z. Yang, X. Fan, S. Bakalis, D. Parker, and P. Fryer, “A method for characterising solids translational and rotational motions using multiple-positron emission particle tracking (multiple-pept),” *International Journal of Multiphase Flow*, vol. 34, no. 12, pp. 1152–1160, 2008.

[24] Z. Yang, P. Fryer, S. Bakalis, X. Fan, D. Parker, and J. Seville, “An improved algorithm for tracking multiple, freely moving particles in a positron emission particle tracking system,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 577, no. 3, pp. 585–594, 2007.

[25] M. Bickell, A. Buffler, I. Govender, and D. Parker, “A new line density tracking algorithm for pept and its application to multiple tracers,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 682, pp. 36–41, 2012.

[26] Balu.ertl. (Feb. 22, 2015). 20 points and their voronoi cells, [Online]. Available: `https://en.wikipedia.org/wiki/Voronoi_diagram#/media/File:Euclidean_Voronoi_diagram.svg`.

[27] J. Borak, *The ghost map: The story of london’s most terrifying epidemic and how it changed science, cities, and the modern world*, 2007.

[28] F. Aurenhammer, “Voronoi diagrams - a survey of a fundamental geometric data structure,” *ACM Computing Surveys (CSUR)*, vol. 23, pp. 345–405, 3 1991.

[29] D. Kirkpatrick, “Optimal search in planar subdivisions,” *SIAM Journal on Computing*, vol. 12, no. 1, pp. 28–35, 1983.

[30] S. P. Lloyd, “Least squares quantization in pcm,” *Information Theory, IEEE Transactions on*, vol. 28, no. 2, pp. 129–137, 1982.

[31] D.-T. Lee and C. Wong, “Voronoui diagrams in $L_1(L_\infty)$ metrics with 2-dimensional storage applications,” *SIAM Journal on Computing*, vol. 9, no. 1, pp. 200–211, 1980.

[32] M. Bock, A. K. Tyagi, J.-U. Kreft, and W. Alt, “Generalized voronoi tessellation as a model of two-dimensional cell tissue dynamics,” *Bulletin of mathematical biology*, vol. 72, no. 7, pp. 1696–1731, 2010.

[33] H. Li, K. Li, T. Kim, A. Zhang, and M. Ramanathan, “Spatial modeling of bone microarchitecture,” in *IS&T/SPIE Electronic Imaging*, International Society for Optics and Photonics, 2012, pp. 82 900–82 900.

[34] (Nov. 5, 2015). Gold coast cultural precinct, ARM Architecture, [Online]. Available: `http://www.a-r-m.com.au/projects_GoldCoastCP.html`.

[35] (). Robust multiscale 2d skeletonization, University of Groningen, [Online]. Available: `http://www.cs.rug.nl/svcg/Shapes/Skel`.

[36] P. F. Ash and E. D. Bolker, “Generalized dirichlet tessellations,” *Geometriae Dedicata*, vol. 20, no. 2, pp. 209–243, 1986.

[37] S. Arya and T. Malamatos, “Linear-size approximate voronoi diagrams,” in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 2002, pp. 147–155.

[38] S. Fortune, “A sweepline algorithm for voronoi diagrams,” *Algorithmica*, vol. 2, no. 1-4, pp. 153–174, 1987.

[39] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise.,” in *Kdd*, vol. 96, 1996, pp. 226–231.

[40] T. E. Fortmann, Y. Bar-Shalom, and M. Scheffe, “Sonar tracking of multiple targets using joint probabilistic data association,” *Oceanic Engineering, IEEE Journal of*, vol. 8, no. 3, pp. 173–184, 1983.

[41] S. S. Blackman, “Multiple hypothesis tracking for multiple target tracking,” *Aerospace and Electronic Systems Magazine, IEEE*, vol. 19, no. 1, pp. 5–18, 2004.

[42] S. Oh, S. Russell, and S. Sastry, “Markov chain monte carlo data association for general multiple-target tracking problems,” in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, IEEE, vol. 1, 2004, pp. 735–742.

[43] S. Särkkä, A. Vehtari, and J. Lampinen, “Rao-blackwellized particle filter for multiple target tracking,” *Information Fusion*, vol. 8, no. 1, pp. 2–15, 2007.

[44] D. Blackwell, “Conditional expectation and unbiased sequential estimation,” *The Annals of Mathematical Statistics*, pp. 105–110, 1947.

[45] R. E. Kalman, “A new approach to linear filtering and prediction problems,” *Journal of Fluids Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[46] S. Blackrnan and A. House, “Design and analysis of modern tracking systems,” *Boston, MA: Artech House*, 1999.

[47] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, “Lof: Identifying density-based local outliers,” in *ACM sigmod record*, ACM, vol. 29, 2000, pp. 93–104.

[48] H.-P. Kriegel, P. Kröger, E. Schubert, and A. Zimek, “Loop: Local outlier probabilities,” in *Proceedings of the 18th ACM conference on Information and knowledge management*, ACM, 2009, pp. 1649–1652.

[49] H.-P. K. P. K. Erich and S. A. Zimek, “Interpreting and unifying outlier scores,” in *11th SIAM International Conference on Data Mining (SDM), Mesa, AZ*, SIAM, vol. 42, 2011.

[50] C. Rycroft. (2008). Voro++, [Online]. Available: `http://math.lbl.gov/voro++/`.

[51] D. York, N. M. Evensen, M. L. Martınez, and J. D. B. Delgado, "Unified equations for the slope, intercept, and standard errors of the best straight line," *American Journal of Physics*, vol. 72, no. 3, pp. 367–375, 2004.

[52] G. Saber and C. Wild, *Nonlinear Regression.* Hoboken, NJ: Wiley-Interscience, 2003.

[53] Z. Yang, D. Parker, P. Fryer, S. Bakalis, and X. Fan, "Multiple-particle tracking - an improvement for positron particle tracking," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 564, no. 1, pp. 332–338, 2006.

# Appendices

# Appendix A

# Matlab Code

## A.1 Location Algorithm

### A.1.1 Main Method

```
function VMPT_Location
tic
cleanupObj = onCleanup(@cleanupfun);
warning off all;
global outputfolder;
%% -------Filename and folder data-------%
[input_folder, out_folder, num_tracers] = getUserInputs();
outputfolder = out_folder;
input_files = dir(strcat(input_folder,'/*.dat'));
num_files = length(input_files);
disp('Number of files found:');
disp(num_files);
outputpath = strcat(outputfolder,'/cluster_data.csv');
dlmwrite(outputpath,[]);
% --------------------------------------%
%% -------Constant Parameters-------%
[num_lines, eps, k, interval] = getParams(num_tracers);
start_frame = 1;                 %timeframe at which to start the algorithm
start_file = 1;                  %file at which to start the algorithm
end_file = num_files;
% ---------------------------------%
%%
writeToLog(outputfolder,'Starting run...\n----------------------------',1);
disp(strcat({'Start time:'},{' '},datestr(now)));

total_frames = 0;
for file_num = start_file:end_file% Loop through each file
    %% Extracts the data from the next file.
    % Additionally, this section adds the remaining data from the previous
    % file to the beginning of the new data. If this is the first file, it
    % detects what type of camera was used
    filename_i = strcat(input_folder,'/',input_files(file_num).name);
    data = dlmread(filename_i);
    if file_num == start_file
        camera = getCameraType(size(data,2));
    end
    line_count = size(data,1);
    disp(strcat('Now reading file #',num2str(file_num)));
    disp('--------------------------------------------');
    num_frames = ceil(size(data,1)/interval);
    output_cell = cell(num_frames,1);
```

```
42      %%
43      % Loop through frames while there are at least NUMLINES lines in the current frame
44      for frame = start_frame:num_frames
45          %% Set up the current frame.
46          % If the algorithm finds that there are fewer than NUMLINES lines
47          % left in the data array, it exits the while-loop and adds the
48          % remaining lines to the beginning of the next file
49          total_frames = total_frames + 1;
50
51          t_start = (frame - 1) * interval + 1;
52          t_end = t_start + num_lines - 1;
53
54          if t_end > line_count
55              t_start = line_count - num_lines + 1;
56              t_end = line_count;
57          end
58          [frame_data,times] = getLines(data,t_start,t_end,camera);
59          frame_time = mean(times);
60          %% Main part of algorithm goes here
61          [discretized,line_inds] = getSeeds(frame_data,eps);
62
63          try
64              [tesselated_inds,volumes] = performTessellation(discretized,line_inds);
65          catch exc
66              disp(exc);
67          end
68          tesselated = discretized(tesselated_inds,:);
69
70          [inlierclass,¬] = getOutliers(tesselated, k);
71
72          [cluster_data,final_vols] = getSmallVolumes(tesselated(inlierclass == 1,:),...
                volumes(inlierclass == 1));
73
74          [cluster_class,¬] = dbscan(cluster_data,k,eps);
75          numclusters = max(cluster_class);
76  %         scatter_clusters(cluster_data,cluster_class);
77          unordered = zeros(numclusters,3);
78          for i = 1:numclusters
79              cluster = cluster_data(cluster_class == i,:);
80              cluster_vols = 1./(final_vols(cluster_class == i));
81              unordered(i,:) = getCentroid(cluster,cluster_vols);
82          end
83          %% Write Position Data To File
84          output = zeros(numclusters,6);
85          output(:,1:3) = unordered;
86          output(:,4) = frame_time;
87          output(:,5) = frame;
88          output(:,6) = file_num;
89
90          output_cell{frame} = output;
91
92          disp(strcat({'Frame #'},{int2str(frame)},{':'},{' '},{int2str(numclusters)}));
93          %pause(0.001);
94      end
95      writeOutput(outputpath,output_cell);
96  end
97  disp(toc);
98  disp('Total number of frames processed: ');
99  disp(total_frames);
100 end
101
102 function cleanupfun
103     global outputfolder
104     write_to_log(outputfolder,'Ending run...\n----------------------------',0);
105     disp('-------------------------------------');
106     disp(strcat({'End time:'},{' '},datestr(now)));
```

```
107  end
```

### A.1.2 Functions

```
1
2
3 function [camera] = getCameraType(num_columns)
4 if num_columns == 7
5     camera = 'c';
6 elseif num_columns == 5
7     camera = 'p';
8 else
9     camera = 'n';
10 end
11 end
```

```
1 function c = getCentroid(P,W)
2     if nargin == 1
3        W = ones(size(P,1),1);
4     end
5     W = W/sum(W);    %normalise the weights
6     c = sum(P .* repmat(W,[1 size(P,2)]));
7 end
```

```
1 function [lines, times] = getLines(data, t_start, t_end, camera)
2     numlines = t_end - t_start + 1;
3     if camera == 'c'
4         lines = data(t_start:t_end,1:6);
5         times = data(t_start:t_end,7);
6     elseif camera == 'p'
7         lines = [data(t_start:t_end,1) data(t_start:t_end,2)...
8             zeros(numlines,1) data(t_start:t_end,3) data(t_start:t_end,4)...
9             799*ones(numlines,1)];
10        times = data(t_start:t_end,5);
11    else
12        lines = 0;
13        times = 0;
14    end
15 end
```

```
1 % Finds the outliers in a data set by using the Local Outlier Factor metric.
2 % Since the LOF assigns a value to each entry, it is up to the user to decide
3 % what value constitutes an outlier. In this case, anything larger than the mean
4 % is considered an outlier.
5 function [ outlier_class, LOF] = getOutliers(x, k)
6
7     knn = knnsearch(x,x,'K',k+1);
8     knn = knn(:,2:end);
9     [kdist,dist] = get_kdistance(x,knn);
10    LRD = get_lrd(x, knn, kdist, dist);
11    LOF = get_lof(LRD, knn);
12
13
14    lof_mean = mean(LOF);
15    lof_std = std(LOF);
16
17    outlier_class = LOF < (lof_mean);%outliers will have class = 0
18 end
19
20 function [kdist,dist] = get_kdistance(x,knn)
21    kdist = zeros(size(knn,1),1);
```

```
22      dist = zeros(size(knn));
23
24      numpoints = size(x,1);
25
26      for i = 1:numpoints
27          knn_i = knn(i,:);
28          kdist_i = pdist2(x(i,:),x(knn_i,:));
29          dist(i,:) = kdist_i;
30          kdist(i) = max(kdist_i);
31      end
32  end
33
34  function [LRD] = get_lrd(x,knn,kdist,dist)
35      LRD = zeros(size(kdist));
36      for A_ind = 1:size(x,1)
37
38          B_inds = knn(A_ind,:);
39
40          rdist_sum = 0;
41
42          for j = 1:size(B_inds,2);
43              kdist_B = kdist(B_inds(j));
44              dist_AB = dist(A_ind,j);
45              rdist = max(kdist_B, dist_AB);
46              rdist_sum = rdist_sum + rdist;
47          end
48
49          LRD(A_ind) =  size(B_inds,2) / rdist_sum;
50      end
51  end
52
53  function [LOF] = get_lof(LRD, knn)
54      LOF = zeros(size(LRD));
55      numpoints = size(LRD,1);
56      for p = 1:numpoints
57          knn_p = knn(p,:);
58          LRD_p = LRD(p);
59          k = size(knn,2);
60
61          LRD_o_sum = 0;
62
63          for o = 1:k
64              LRD_o = LRD(knn_p(o));
65              LRD_o_sum = LRD_o_sum + LRD_o;
66          end
67
68          LOF(p) = (LRD_o_sum / LRD_p)/k;
69      end
70  end
```

```
1   %% Gets the Constant Parameters used in the Tesselation
2   % This function takes in two arguments and returns the parameters used in
3   % discretizing the lines of response.
4   %   Input Arguments:
5   %       - numtracers:   The number of tracers to be tracked.
6   %       - type:         The type of discretization. Can take one of two
7   %                       values: 'spacing' for constant spacing along the
8   %                       lines; or 'numpoints' for equal numbers of points
9   %                       per line.
10  %   Output Arguments:
11  %       - param1:       The number of lines used per frame.
12  %       - param2:       Either the spacing distance or the number of points
13  %                       used per line.
14  function [num_lines, eps, k, interval] = getParams(num_tracers)
```

```
    num_lines = 100 * num_tracers;
    eps = 5;
    k = 4;
    interval = num_lines / 2;
end
```

```
% This function discretizes the data for one timeframe. Depending on the
% method used for the discretization, the function parameters and return
% values will possibly change.
% Specifically, this version splits each LOR into a number of points using
% a constant spacing between points
function [outpoints, outindices] = getSeeds(LD,spacing)
    numlines = size(LD,1);

    R = LD(:,4:6) - LD(:,1:3);%vectors describing lines

    numpoints = floor(sum(sqrt(R(:,1).^2 + R(:,2).^2 + R(:,3).^2))/spacing);
    points = zeros(numpoints,3);
    indices = zeros(numpoints,1);

    count = 1;
    for i = 1:numlines
        len = norm(R(i,:));
        numpoints = floor(norm(R(i,:))/spacing);
        indices(count:numpoints + count) = i;

        points(count:(count + numpoints),:) = repmat(LD(i,1:3),[numpoints+1 1]) + ...
            spacing * repmat((0:numpoints)',[1 3]) .* repmat(R(i,:)/len,[numpoints...
                +1,1]);

        count = count + numpoints;
    end

    [outpoints,ia,¬] = unique(points(any(points,2),:),'rows');

    clear points;

    indices = indices(any(indices,2));
    outindices = indices(ia);
end
```

```
/*
 * test_mex.c
 */
#include <math.h>
#include "mex.h"
#include "matrix.h"
/*
 * Declare functions
 */
double get_3D_dist(const double* p1_in, const double* p2_in);
double get_vol(const double* V,const int M, const double* center, const mxArray* ...
    inds_in);
void check_inputs(int nlhs, mxArray *plhs[],int nrhs, const mxArray *prhs[]);
/*
 * Entry point of the mex file
 */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    #define X_in        prhs[0]     //matrix of 3D data points
    #define LINES_in    prhs[1]     //cell array; each element is the indices into ...
        X_in of the corresponding line number
```

```
21      #define V_in        prhs[2]     //array of voronoi vertices in 3D space
22      #define C_in        prhs[3]     //cell array; each element is the indices into ...
            V_in of the corresponding voronoi point
23
24      #define INDS_out    plhs[0]     //vector; each element is an index into X_in of ...
            the smallest point on the corresponding line
25      #define VOL_out     plhs[1]     //vector; contains volumes of corresponding points
26
27      /* Declare variables */
28      int i,line,xM,numlines,minvol_ind,p_counter,vM;
29      double p_vol,minvol,*x,*inds_pr,*vol_pr,*V_pr;
30      mxArray *vertexcell;
31      /* Validate input data */
32      check_inputs(nlhs,plhs,nrhs,prhs);
33      /* Initialise variables */
34      numlines = (int)mxGetM(LINES_in);
35      vM = (int)mxGetM(V_in);
36      xM = (int)mxGetM(X_in);
37      V_pr = mxGetPr(V_in);
38      x = mxGetPr(X_in);
39      if(V_pr == 0)
40          mexErrMsgIdAndTxt("MATLAB:nullPointerAccessViolation","Pointer to V_in data is...
                 a null pointer.");
41      if(x == 0)
42          mexErrMsgIdAndTxt("MATLAB:nullPointerAccessViolation","Pointer to X_in data is...
                 anull pointer.");
43      /* The output variables */
44      INDS_out = mxCreateDoubleMatrix(numlines,1,mxREAL);
45      VOL_out = mxCreateDoubleMatrix(numlines,1,mxREAL);
46      inds_pr = mxGetPr(INDS_out);
47      vol_pr = mxGetPr(VOL_out);
48      /* Loop over each line */
49      for(line = 0; line < numlines; line++)
50      {
51          mxArray *linecell = mxGetCell(LINES_in,line);
52          const int n_points = (int)mxGetNumberOfElements(linecell);
53          int p;
54          double* linepoints = mxGetPr(linecell);
55          if(linepoints == 0)
56          {
57              vol_pr[line] = 0;
58              inds_pr[line] = -1;
59              continue;
60          }
61          minvol = 9999999;
62          minvol_ind = 0;
63          /* Loop over each point on the line */
64          for(p = 0; p < n_points; p++)
65          {
66              int p_index = (int)(linepoints[p]-1);
67              double center[3];
68              if(p_index ≥ xM)
69              {
70                  mexWarnMsgIdAndTxt("MATLAB:IndexOutOfBounds","Index into X_in exceeds ...
                     size of X_in.");
71                  continue;
72              }
73              center[0] = x[p_index + xM*0];
74              center[1] = x[p_index + xM*1];
75              center[2] = x[p_index + xM*2];
76              vertexcell = mxGetCell(C_in,p_index);
77              p_vol = get_vol(V_pr,vM,center,vertexcell);
78              if(p_vol ≤ 0) //p_vol = -1 when there is a polygon of infinite size
79              {
80                  continue;
81              }
```

```
82              else
83              {
84                  if(p_vol < minvol)
85                  {
86                      minvol = p_vol;
87                      minvol_ind = p;
88                  }
89              }
90          }
91          vol_pr[line] = minvol;
92          inds_pr[line] = (int)linepoints[minvol_ind];
93      }
94      return;
95  }
96  /*
97   * Check that all inputs are valid
98   */
99  void check_inputs(int nlhs, mxArray *plhs[],
100                  int nrhs, const mxArray *prhs[])
101 {
102     if(prhs[0] == 0 || prhs[1] == 0 || prhs[2] == 0 || prhs[3] == 0)
103         mexErrMsgIdAndTxt("MATLAB:nullPointerAccessViolation","A null pointer was sent...
             as an input argument.");
104     if(nrhs != 4)
105         mexErrMsgIdAndTxt("MATLAB:arrayFillGetPr:rhs","This function takes 4 input ...
            arguments.");
106     if(nlhs != 2)
107         mexErrMsgIdAndTxt("MATLAB:arrayFillGetPr:lhs","This function returns 2 output ...
            arguments.");
108     if(!mxIsCell(prhs[1]) || (mxGetN(prhs[1]) != 1))
109         mexErrMsgIdAndTxt("MATLAB:odearguments:InconsistentDataType","Second input ...
            argument should be a cell array of size M-by-1");
110     if(!mxIsCell(prhs[3]) || (mxGetN(prhs[3]) != 1))
111         mexErrMsgIdAndTxt("MATLAB:odearguments:InconsistentDataType","Fourth input ...
            argument should be a cell array of size X-by-1");
112     if(mxGetM(prhs[3]) != mxGetM(prhs[0]))
113         mexErrMsgIdAndTxt("MATLAB:odearguments:InconsistentDataLength","The first and ...
            fourth input arguments should have the same numberofrows.");
114     if(mxGetN(prhs[0]) != 3)
115         mexErrMsgIdAndTxt("MATLAB:odearguments:InconsistentDataLength","The first ...
            input argument should be an X-by-3 matrix.");
116     return;
117 }
118 
119 /*
120  * Returns the 'simple volume' of a shape in 3D.
121  * The simple volume has been defined as the maximum distance from the center
122  * of the volume to each of its vertices.
123  */
124 double get_vol(const double* V,const int M, const double* center, const mxArray* ...
        inds_in)
125 {
126     double vol,tempvol,*inds;
127     int i,inds_size,j;
128     double vertex[3];
129     ////////////////////////////////
130     vol = 0;
131     tempvol = 0;
132     inds_size = (int)mxGetNumberOfElements(inds_in);
133     inds = mxGetPr(inds_in);
134     if(inds == 0)
135         mexErrMsgIdAndTxt("MATLAB:nullPointerAccessViolation","Pointer to mxGetPr(...
            inds_in) in get_vol is a null pointer.");
136     /* Loop over each vertex */
137     for(i = 0; i < inds_size; i++)
138     {
```

```
139         int m = (int)(inds[i]) - 1;
140         for(j = 0; j < 3; j++)
141         {
142             if(mxIsInf(V[m + M*j]))
143             {
144                 return (-1);
145             }
146             else
147             {
148                 vertex[j] = V[m + M*j];
149             }
150         }
151 
152         tempvol = get_3D_dist(center, vertex);
153 
154         if(tempvol > vol)
155         {
156             vol = tempvol;
157         }
158     }
159     if(vol == 0) //If the  volume has been discoveed to be 0, somethnig has gone wrong...
         and a large volume is assigned
160     {
161         vol = 99999999;
162     }
163     return vol;
164 }
165 /*
166  * Returns the Euclidian distance between two points in 3D
167  */
168 double get_3D_dist(const double* p1_in, const double* p2_in)
169 {
170     int i;
171     double dist = 0;
172 
173     for(i = 0; i < 3; i++)
174     {
175         dist += (p1_in[i] - p2_in[i]) * (p1_in[i] - p2_in[i]);
176     }
177     dist = sqrt(dist);
178     return dist;
179 }
```

```
1  function [final_data,final_vols] = getSmallVolumes(data,vols)
2  init_mean = mean(vols);
3  init_std = std(vols);
4  
5  valid_inds = find(vols ≤ (init_mean + 2*init_std));
6  valid_data = data(valid_inds,:);
7  valid_vols = vols(valid_inds);
8  
9  new_mean = mean(valid_vols);
10 new_std = std(valid_vols);
11 
12 max_vol = new_mean + 0.2 * new_std;
13 
14 final_data = valid_data(valid_vols ≤ max_vol,:);
15 final_vols = valid_vols(valid_vols ≤ max_vol);
16 end
```

```
1  %% Get Initial User Inputs
2  % Returns the user inputs and initiates the output folder as well as the
3  % output files.
```

```
4  % Output arguments:
5  %   - input_path:      The path of the folder containing the input files,
6  %                      i.e. the files with the line data.
7  %   - output_path:     The folder to which the tracer data is written.
8  %                      NOTE: this should be a single name, not a full path.
9  %                      The folder name is appended to '[current directory]/output/'.
10 %   - numtracers:      The number of tracers expected in the camera view.
11 function [input_path, output_path, numtracers] = getUserInputs()
12 %% Gets the Path Containing the Input Data
13 input_path = input('Enter the folder in which the input files are stored:   ','s');
14 
15 %% Creates the Output Folder
16 % Gets the name of the output folder. If the folder already exists, asks
17 % whether to overwrite the existing folder or choose a different name.
18 % NOTE: overwriting the folder will overwrite the files contained within.
19 CREATEFOLDER = 0;
20 while ¬CREATEFOLDER
21     output_path = input('Enter the folder to which the tracking data will be written: ...
          ','s');
22     if exist(output_path,'dir')
23         overwrite = input('Folder already exists. Overwrite (y/n)?   ','s');
24         if strcmpi(overwrite,'y')
25             CREATEFOLDER = 1;
26         end
27     else
28         CREATEFOLDER = 1;
29     end
30 end
31 mkdir(output_path);
32 %% Gets thenumber of tracers
33 numtracers = input('Enter the number of tracers:   ');
34 end
```

```
1  % Uses voronoi tesselation to refine the discretized lines.
2  %
3  % It does so by performing a Voronoi tesselation on the whole set of
4  % points. Then, for each line, it finds the point on that line which has
5  % the smallest Voronoi volume surrounding it. This 'smallest' point is
6  % added to the output matrix. The volume is also returned as an entry in an
7  % array.
8  function [out_points, out_vols] = performTessellation(data,line_numbers)
9      num_lines = max(line_numbers);
10 
11     [V,C] = voronoiDiagram(delaunayTriangulation(data));
12     % Create cell structure for lines. Each entry in the cell structure
13     % contains an array of indices into data, with each data point
14     % belonging to that line
15     lines = cell(num_lines,1);
16     for i = 1:num_lines
17         lines{i} = find(line_numbers == i);
18     end
19     %---------Find smallest polyhedron per line----------------%
20     try
21         [smallest,vols] = getSmallestMex(data,lines,V,C);
22 
23         clear V C lines;
24 
25         valid = find(vols > 0);
26 
27         [¬,ia,¬] = unique(data(smallest(valid)),'rows');
28         out_vols = vols(valid(ia));
29         out_points = smallest(valid(ia));
30     catch exc
31         throw(exc);
```

```
32      end
33  end
```

```
1   function writeOutput(filename, frames)
2       for i = 1:size(frames,1)
3           dlmwrite(filename,frames{i},'-append');
4       end
5   end
```

```
1   %% Write Message to Log.
2   % Appends a message at the end of a log file. Also adds the current date
3   % and time as a header to the message.
4   %   Input Arguments:
5   %       - folder:       The folder which contains the log file t be edited.
6   %       - entry:        A string to be appended to the log file.
7   function writeToLog(folder,entry,overwrite)
8   if nargin == 2
9       overwrite = 0;
10  end
11  try
12      % creates the folder if it doesn't exist
13      if ¬exist(folder,'dir')
14          mkdir(folder);
15      end
16      date = datestr(now());
17      format = '----------------------------\n%s\n%s\n';
18      formatstring = sprintf(format,date,entry);
19      logfile = strcat(folder,'/log.txt');
20      if overwrite
21          fid = fopen(logfile,'w');
22      else
23          fid = fopen(logfile,'a');
24      end
25      fprintf(fid,formatstring);
26      fclose(fid);
27  catch exc
28      disp(exc.message);
29  end
30  end
```

## A.2 Tracking Algorithm

### A.2.1 Main Method

```
function VMPT_Tracking()
% --------------------------------------------------------------
        disp('Enter the name of the folder containing the clustered data:  ');
        folder = input('      ','s');
        filename = strcat(folder,'/cluster_data.csv');
        disp('Enter the name of the output files: ');
        basename = input('      ','s');
        mkdir(strcat(folder,'/tracks'));
        stitchedname = strcat(folder,'/tracks/',basename);
%---------------------------------------------------------------
    trajectories = {};
    outputs = {};
    outputcount = 0;
    ID_counter = 1;

    global maxdist;
    maxdist = 20;

    interval_size = 3;
    disp('Reading cluster data...');
    cluster_data = dlmread(filename);
    disp('Ordering cluster data...');
    frame_cell = convertToCell(cluster_data);
    num_frames = length(frame_cell);

    disp('Calculating trajectories...');
    % First frame - place each centroid into an individual trajectory
    for clust = 1:size(frame_cell{1},1)
        frame_data = frame_cell{1}(clust,:);
        trajectories{end+1} = Trajectory(frame_data(1:3),frame_data(4),ID_counter);
        ID_counter = ID_counter + 1;
    end
    %Subsequent frames
    for i = 1+interval_size:interval_size:num_frames
        %Get data for the current frame
%         disp(size(frame_cell{i}));
        centroids = frame_cell{i}(:,1:3);
        time = frame_cell{i}(1,4);
        %See which (if any) clusters match up to a trajectory that
        %currently exists
        match = getMatchMatrix(trajectories, centroids, time);
        %Loop over each trajectory:
        end_flags = zeros(length(trajectories),1);
        for t = 1:length(trajectories)
            %If no clusters match the current trajectory, skip the frame.
            %Note: if more than max_skips (defined in Trajectory) skips
            %occur in a row, the trajectory is ended and either discarded
            %or written to file.
            match_arr = match(t,:);
            if ¬any(match_arr)
                end_flags(t) = trajectories{t}.skipFrame();%Add in what happens to the...
                     trajectory...
                if end_flags(t) == 1
                    outputcount = outputcount + 1;
                    outputs{outputcount,1} = trajectories{t}.getTrajectoryOutput;
                    outputs{outputcount,2} = trajectories{t}.getID();
                end
            %Add the matching centroids to their relevant trajectories
            else
                trajectories{t}.addPosition(centroids(match_arr == 1,:),time);
```

```
60              end
61          end
62          trajectories(end_flags > 0) = [];
63          %Now for each cluster that does not belong to a trajectory, create
64          %a new trajectory with the cluster as the initial value.
65          for c = 1:size(centroids,1)
66              match_arr = match(:,c);
67              if ¬any(match_arr)
68                  trajectories{end+1} = Trajectory(centroids(c,:),time,ID_counter);
69                  ID_counter = ID_counter + 1;
70              end
71          end
72      end
73
74      for t = 1:length(trajectories)
75          if trajectories{t}.doWrite()
76              outputcount = outputcount + 1;
77              outputs{outputcount,1} = trajectories{t}.getTrajectoryOutput;
78              outputs{outputcount,2} = trajectories{t}.getID();
79          end
80      end
81      disp(size(outputs,1));
82      outputs = sortrows(outputs,2);
83
84      disp('Padding trajectories...');
85      padded = padTrajectories(outputs(:,1));
86      disp('Stitching trajectories...');
87      stitched = stitchTrajectories(padded);
88      disp(length(stitched));
89      disp('Writing trajectories to file...');
90
91      for t = 1:length(stitched)
92          stitchedpath = strcat(stitchedname,'_trajectory_',padNumber(t,length(stitched)...
            ),'.csv');
93          dlmwrite(stitchedpath,stitched{t});
94      end
95
96      disp('Done.');
97  end
98  % Convert the data file created by the tesselation scheme into a cell
99  % array, with each entry containing the clusters located at one time frame
100
101 function [converted] = convertToCell(x)
102     times = unique(x(:,4));
103     numentries = size(times,1);
104     converted = cell(numentries,1);
105
106     for i = 1:numentries
107         time = times(i);
108         converted{i} = x(x(:,4) == time,1:4);
109     end
110 end
111
112 % Determine which centroids match up to which trajectories
113 function [match] = getMatchMatrix(trajectories, centroids, time)
114     global maxdist;
115
116     num_t = length(trajectories);
117     num_c = size(centroids,1);
118     inferred = zeros(num_t,3);
119     match = zeros(num_t,num_c);
120     match_c = zeros(size(match));
121     match_t = zeros(size(match));
122     distances = zeros(size(match));
123
124     for i = 1:num_t
```

```
125         inferred(i,:) = trajectories{i}.getInferredPosition(time);
126     end
127 
128     for t = 1:num_t
129         traj = inferred(t,:);
130         for c = 1:num_c
131             cent = centroids(c,:);
132             distances(t,c) = pdist2(traj,cent);
133         end
134     end
135     probs = 1./distances;
136     [¬,i_c] = max(probs,[],1);
137     [¬,i_t] = max(probs,[],2);
138 
139     for c = 1:num_c
140         match_c(i_c(c),c) = 1;
141     end
142     for t = 1:num_t
143         match_t(t,i_t(t)) = 1;
144     end
145 
146     match = match_c .* match_t;
147 
148     matched = find(match == 1);
149 
150     for i = 1:length(matched)
151         if distances(matched(i)) > maxdist
152             match(matched(i)) = 0;
153         end
154     end
155 end
```

### A.2.2 Classes

```
1  classdef Trajectory < handle
2  
3      properties (SetAccess = private, GetAccess = private)
4          all_data = [];
5          number_previous = 10;
6          positions = [];
7          velocities = [];
8          accelerations = [];
9          times = [];
10         valid_error_limit = 10;
11         ID = 0;
12         skipped_frames = 0;
13         max_skip = 10;
14         min_entries = 100;
15         num_entries = 0;
16     end
17 
18     methods
19         %Constructor
20         function trajectory = Trajectory(initial_position,initial_time, t_ID)
21             trajectory.positions = initial_position;
22             trajectory.times = initial_time;
23             trajectory.velocities = zeros(size(initial_position));
24             trajectory.ID = t_ID;
25             trajectory.all_data = [initial_position,initial_time];
26         end
27         %Function adds new position to trajectory data. If the 'positions'
28         %matrix is full (ie has number_previous non-zero entries) the first
29         %row is deleted, everything is shifted up and the new entry is
30         %added at the end.
31         function [valid] = addPosition(trajectory,new_position,new_time)
32             valid = 1;%trajectory.validatePosition(new_position,new_time);
33             if valid == 0
34                 return;
35             end
36             if size(trajectory.positions,1) < trajectory.number_previous
37                 trajectory.positions = [trajectory.positions;new_position];
38                 trajectory.times = [trajectory.times;new_time];
39             else
40                 trajectory.positions(1:trajectory.number_previous-1,:) = ...
41                     trajectory.positions(2:trajectory.number_previous,:);
42                 trajectory.positions(trajectory.number_previous,:) = new_position;
43 
44                 trajectory.times(1:trajectory.number_previous-1,:) = ...
45                     trajectory.times(2:trajectory.number_previous,:);
46                 trajectory.times(trajectory.number_previous,:) = new_time;
47             end
48             trajectory.all_data(end+1,:) = [new_position,new_time];
49             trajectory.updateVelocities();
50             trajectory.num_entries = trajectory.num_entries + 1;
51             trajectory.skipped_frames = 0;
52         end
53         %function interpolates next position, based on the previous
54         %positions of the particle
55         function [next_position] = getInferredPosition(trajectory,new_t)
56             x_avg = mean(trajectory.positions,1);
57             v_avg = mean(trajectory.velocities,1);
58             t_avg = mean(trajectory.times,1);
59             if norm(v_avg) == 0
60                 next_position = trajectory.positions(end,:);
61                 return;
62             end
```

```
63
64              xn_prime = x_avg + v_avg * (trajectory.times(end) - t_avg);
65
66              next_position = xn_prime + v_avg * (new_t - trajectory.times(end));
67          end
68          %returns the average velocity over the last n positions
69          function [v] = getVelocity(trajectory)
70              if size(trajectory.velocities,1) > 0
71                  v = mean(trajectory.velocities,1);
72              else
73                  v = 0;
74              end
75          end
76          %returns the most recent position
77          function [x] = getLastPosition(trajectory)
78              x = trajectory.positions(end,:);
79          end
80
81          %returns the difference between an input time and the latest time
82          %in the array
83          function [Δ_t] = getDeltaT(trajectory,new_t)
84              Δ_t = new_t - trajectory.times(end);
85          end
86          function [prev_position] = previousPosition(trajectory)
87              prev_position = trajectory.positions(end,:);
88          end
89
90          function [END_FLAG] = skipFrame(trajectory) %0 = continue; 1 = write data to ...
                file; 2 = discard trajectory
91              trajectory.skipped_frames = trajectory.skipped_frames + 1;
92              if trajectory.skipped_frames ≥ trajectory.max_skip
93                  if trajectory.num_entries < trajectory.min_entries
94                      END_FLAG = 2;
95                  else
96                      END_FLAG = 1;
97                  end
98              else
99                  END_FLAG = 0;
100             end
101         end
102
103         function [output] = getTrajectoryOutput(trajectory)
104             output = trajectory.all_data;
105         end
106
107         function [id] = getID(trajectory)
108             id = trajectory.ID;
109         end
110
111         function [WRITE_FLAG] = doWrite(trajectory)
112             if trajectory.num_entries < trajectory.min_entries
113                 WRITE_FLAG = 0;
114             else
115                 WRITE_FLAG = 1;
116             end
117         end
118     end
119
120     methods(Access = private)
121         %udpates all velocities when a new position is added
122         function updateVelocities(trajectory)
123             new_v = (trajectory.positions(end,:) - trajectory.positions(end-1,:))/...
124                 (trajectory.times(end) - trajectory.times(end-1));
125
126             if size(trajectory.velocities,1) < trajectory.number_previous
127                  trajectory.velocities = [trajectory.velocities;new_v];
```

```
128             else
129                 trajectory.velocities(1:end-1,:) = ...
130                     trajectory.velocities(2:end,:);
131                 trajectory.velocities(end,:) = new_v;
132             end
133         end
134         %checks to see if a newly-added position is valid, i.e. within a
135         %sphere of a certain size around the inferred position
136         function [valid] = validatePosition(trajectory,test_position,new_time)
137             if new_time ≤ trajectory.times(end)
138                 valid = 0;
139                 return;
140             end
141 
142             inferred = trajectory.getInferredPosition(new_time);
143             difference = inferred - test_position;
144             distance = sqrt(sum(difference .^ 2));
145 
146             max_distance = trajectory.valid_error_limit + ...
147                 norm(trajectory.getVelocity()) * (new_time - trajectory.times(end));
148 
149             if distance ≤ max_distance
150                 valid = 1;
151             else
152                 valid = 0;
153             end
154         end
155     end
156 
157 end
```

### A.2.3 Functions

```
1  function [padded] = padTrajectories(trajectories)
2      numtraj = length(trajectories);
3
4      times = getUniqueTimes(trajectories);
5      numtimes = length(times);
6
7      padded = cell(numtraj,1);
8
9      for i = 1:numtraj
10         output = NaN(numtimes,4);
11         output(:,4) = times;
12
13         frame = trajectories{i};
14         for j = 1:size(frame,1)
15             output(output(:,4) == frame(j,4),1:3) = frame(j,1:3);
16         end
17         padded{i} = output;
18     end
19 end
20
21 function [times] = getUniqueTimes(frames)
22     all_times = [];
23
24     for i = 1:length(frames)
25         all_times = [all_times;frames{i}(:,4)];
26     end
27
28     times = unique(all_times,'sorted');
29 end
```

```
1  function [stitched] = stitchTrajectories(input_traj)
2      % Parameter defines how close chronologically two trajectories must be
3      % to be stitched together
4      global CHRONOMAX;
5      global EXTRAP_LENGTH;
6      CHRONOMAX = 30;
7      EXTRAP_LENGTH = 18;
8      distmax = 20;
9
10     trajectories = input_traj;
11
12     stitched = {};
13     % While the trajectory cell array still contains entries...
14     while ¬isempty(trajectories)
15         traj1 = trajectories{1};
16
17         possibles = [];
18         % Find trajectories which could possibly intersect based on their
19         % chronological orientation
20         for i = 2:size(trajectories,1)
21             if chronoIntersect(traj1, trajectories{i})
22                 possibles(end+1) = i;
23             end
24         end
25         % If no trajectories can be stitched to traj1, save it to the
26         % output array (stitched) and remove it from the current array
27         if isempty(possibles)
28             stitched{end+1} = traj1;
29             trajectories(1) = [];
30         else
31             dists = zeros(length(possibles),1);
```

```
32              for j = 1:length(possibles)
33                  dists(j) = getMatchability(traj1,trajectories{possibles(j)});
34              end
35              [val,ind] = min(dists);
36              if val < distmax
37                  joined = joinTrajectories(traj1,trajectories{possibles(ind)});
38                  trajectories([1,possibles(ind)]) = [];
39                  trajectories{end+1} = joined;
40              else
41                  stitched{end+1} = traj1;
42                  trajectories(1) = [];
43              end
44          end
45      end
46  end
47
48  function [joined] = joinTrajectories(traj1, traj2)
49      inds1 = find(¬isnan(traj1(:,1)));
50      inds2 = find(¬isnan(traj2(:,1)));
51
52      data1 = traj1(min(inds1):max(inds1),:);
53      data2 = traj2(min(inds2):max(inds2),:);
54
55      if data1(end,4) < data2(end,4)
56          joined = traj1;
57          joined(min(inds2):max(inds2),:) = data2;
58      else
59          joined = traj2;
60          joined(min(inds1):max(inds1),:) = data1;
61      end
62  end
63  % Check whether two trajectories match up chronologically - that is, the
64  % beginning of one is near (in time) to the end of the other
65  function [intersect] = chronoIntersect(traj1, traj2)
66      global CHRONOMAX;
67      intersect = 0;
68
69      inds1 = find(¬isnan(traj1(:,1)));
70      inds2 = find(¬isnan(traj2(:,1)));
71
72      if (abs(max(inds1) - min(inds2)) < CHRONOMAX)...
73              || (abs(max(inds2) - min(inds1)) < CHRONOMAX)
74          intersect = 1;
75      end
76  end
77
78  function [match_level] = getMatchability(traj1, traj2)
79      inds1 = find(¬isnan(traj1(:,1)));
80      inds2 = find(¬isnan(traj2(:,1)));
81
82      data1 = traj1(min(inds1):max(inds1),:);
83      data2 = traj2(min(inds2):max(inds2),:);
84
85      if data1(end,4) < data2(end,4)
86          first = interpolateNans(data1);
87          second = interpolateNans(data2);
88      else
89          first = interpolateNans(data2);
90          second = interpolateNans(data1);
91      end
92
93      mid_time = (first(end,4) + second(1,4))/2;
94
95      extrap_first = extrapolatePath(first,mid_time);
96      extrap_second = extrapolatePath(second,mid_time);
97
```

```
98      match_level = norm(extrap_first - extrap_second);
99  end
100
101 function [interpolated] = interpolateNans(broken)
102     smoothed = Smoovie(broken,1);
103     interpolated = nan(size(broken));
104     interpolated(:,end) = broken(:,end);
105
106     non_nan = find(¬isnan(smoothed(:,1)));
107
108     for i = min(non_nan):max(non_nan)
109         if isnan(smoothed(i,1))
110             ti = smoothed(i,4);
111             i1 = i - 1;
112             i2 = find(¬isnan(smoothed(i:end,1)),1,'first');
113             i2 = i2 + (size(smoothed,1) - size(smoothed(i:end,:),1));
114             t1 = smoothed(i1,4);
115             t2 = smoothed(i2,4);
116
117             interpolated(i,1:3) = smoothed(i1,1:3) + ...
118                 (smoothed(i2,1:3) - smoothed(i1,1:3)) * (ti - t1)/(t2 - t1);
119             smoothed(i,1:3) = interpolated(i,1:3);
120         else
121             interpolated(i,1:3) = smoothed(i,1:3);
122         end
123     end
124 end
125
126 function [extrap] = extrapolatePath(data, new_t)
127     global EXTRAP_LENGTH;
128
129     times = data(:,4);
130
131     [¬,ind] = min(abs(times - new_t));
132
133     if (ind + EXTRAP_LENGTH/2) > length(times)
134         end_i = length(times);
135         start_i = length(times) - EXTRAP_LENGTH + 1;
136     elseif (ind - EXTRAP_LENGTH) < 1;
137         end_i = EXTRAP_LENGTH;
138         start_i = 1;
139     else
140         start_i = ceil(ind - EXTRAP_LENGTH/2);
141         end_i = floor(ind + EXTRAP_LENGTH/2);
142     end
143
144     extrap_x = data(start_i:end_i,1:3);
145     extrap_t = data(start_i:end_i,4);
146
147     vel = zeros(size(extrap_x,1)-1,3);
148     for i = 1:(size(vel,1))
149         vel(i,:) = (extrap_x(i+1,:) - extrap_x(i,:))/(extrap_t(i+1) - extrap_t(i));
150     end
151
152     x_avg = mean(extrap_x,1);
153     v_avg = mean(vel,1);
154     t_avg = mean(extrap_t);
155
156     xn_prime = x_avg + v_avg * (extrap_t(end) - t_avg);
157
158     extrap = xn_prime + v_avg * (new_t - extrap_t(end));
159 end
```

# Appendix B

# Ethics Form

Application for Approval of Ethics in Research (EiR) Projects
Faculty of Engineering and the Built Environment, University of Cape Town

## APPLICATION FORM

**Please Note:**
Any person planning to undertake research in the Faculty of Engineering and the Built Environment (EBE) at the University of Cape Town is required to complete this form **before** collecting or analysing data.The objective of submitting this application *prior* to embarking on research is to ensure that the highest ethical standards in research, conducted under the auspices of the EBE Faculty, are met. Please ensure that you have read, and understood the **EBE Ethics in Research Handbook**(available from the UCT EBE, Research Ethics website) prior to completing this application form: http://www.ebe.uct.ac.za/usr/ebe/research/ethics.pdf

| **APPLICANT'S DETAILS** | | |
|---|---|---|
| Name of principal researcher, student or external applicant | | [illegible] |
| Department | | Mechanical Engineering |
| Preferred email address of applicant | | [illegible] |
| If a Student | Your Degree: e.g., MSc, PhD, etc., | MSc |
| | Name of Supervisor (if supervised): | [illegible] |
| If this is a researchcontract, indicate the source of funding/sponsorship | | Click here to enter text. |
| Project Title | | [illegible] |

**I hereby undertake to carry out my research in such a way that:**

- there is no apparent legal objection to the nature or the method of research; and
- the research will not compromise staff or students or the other responsibilities of the University;
- the stated objective will be achieved, and the findings will have a high degree of validity;
- limitations and alternative interpretations will be considered;
- the findings could be subject to peer review and publicly available; and
- I will comply with the conventions of copyright and avoid any practice that would constitute plagiarism.

| SIGNED BY | Full name | Signature | Date |
|---|---|---|---|
| **Principal Researcher/ Student/External applicant** | [illegible] | | 26/05/2016 |

| APPLICATION APPROVED BY | Full name | Signature | Date |
|---|---|---|---|
| **Supervisor** (where applicable) | [illegible] | | 26/05/2016 |
| **HOD (or delegated nominee)** Final authority for all applicants who have answered NO to all questions in Section1; and for all Undergraduateresearch (Including Honours). | Prof. T. Bello-Ochende | | 5/8/2016 Click here to enter a date. |
| **Chair : Faculty EIR Committee** For applicants other than undergraduate students who have answered YES to any of the above questions. | Click here to enter text. | | Click here to enter a date. |