APPLICATION OF PARALLEL COMPUTING IN PERFORMING AN UNSUPERVISED FLUOROSCOPIC ANALYSIS OF KNEE JOINT KINEMATICS

by

Renu Ramanatha

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

December 2009

APPROVAL TO SUBMIT THESIS

This thesis presented by Renu Ramanatha entitled *Application of Parallel Computing in Performing an Unsupervised Fluoroscopic Analysis of Knee Joint Kinematics* is hereby approved:

Dr. Amit Jain Advisor and Committee Co-Chair	Date
Dr. Elisa Barney Smith Committee Co-Chair	Date
Dr. Timothy Anderson Committee Member	Date
John R. Pelton Dean of the Graduate College	Date

dedicated to my husband, Prashant and my parents, Geetha and Ramanatha

ACKNOWLEDGMENTS

I wish to express my earnest gratefulness to Dr. Amit Jain and Dr. Elisa Barney Smith for guiding me towards achieving my M.S. degree. They were exceptional mentors to me. I thank Dr. Jain for introducing me to this incredible field of Parallel Computing. He has been guiding and motivating me from my very first day at the Boise State University. He has helped me immensely in understanding the concepts of parallel computing, better programming skills and general computer science skills. Dr. Barney Smith has helped me understand the fluoroscopic principles and the complexity of medical imaging algorithms. Her lucid explanations using 3-D models not only helped me understand the work required for my thesis, but also created a huge interest in this subject. I also wish to express my gratitude to Dr. Tim Andersen for providing me different ideas to implement and test my thesis work. I extend my heartfelt gratitude to all three of them for their invaluable support throughout my research.

I wish to express my deepest thankfulness to Dr. Teresa Cole and Dr. Jain for providing me a teaching assistantship. It helped me cover my tuition and other expenses during my first semester at BSU. I also like to thank all my faculty members for helping me learn the key computer science skills required for this thesis and the rest of my career.

My husband and my parents have been a tremendous support to me all along my M.S studies. Without their encouragement, understanding and sacrifices, this work would not have been possible. I owe my success to all my teachers, right from kindergarten to graduate school.

ABSTRACT

Fluoroscopic analysis of knee joint kinematics involves accurately determining the position and orientation of bones in the knee joint. This data can be derived using the static 3-D CT scan images and 2-D video fluoroscopy images together. This involves generating hypothetical digitally reconstructed radiographs (DRR) from the CT scan image with known position and orientation and comparing them to the original fluoroscopic frame. This represents a search problem in which, among all the DRRs possible from a CT image, the image that matches the target fluoroscopy frame of the knee joint has to be searched.
Each image in the search space differs from another by the set of position and orientation values. Position is defined by the x, y and z co-ordinates in the Cartesian coordinate system. Orientation is defined by the values of azimuth, elevation and rotation. Therefore this constitutes a six dimensional search problem. Using a brute force method to search for the target image which is associated with six independent values, each of which has a large range of possible values, can take a tremendous amount of time. The fact that the set of six values cannot be ordered or categorized in any way adds to the complexity of the search. Further, previous research conducted by Charles Scott et al. [1] suggests that even good sequential search algorithms such as the sequential Monte Carlo method can be very time-consuming. Therefore, a search solution which is suitable for this kind of 6-D search and one that provides a significant speedup has to be used. This thesis involves using Swarm Intelligence (SI) techniques in a parallel computing environment to achieve faster results. Parallel programs are developed using two SI techniques, Bees Algorithm and Particle Swarm Optimization technique.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CT	– Computer Tomography
DRR	– Digital Reconstructed Radiograph
SI	– Swarm Intelligence
PSO	– Particle Swarm Optimization
ACO	– Ant Colony Optimization
SDS	– Stochastic Diffusion Search

CHAPTER 1

INTRODUCTION

1.1 Fluoroscopic Analysis

A good understanding of knee joint kinematics is essential for evaluating and researching the problems associated with the joint. To understand knee joint kinematics, determination of the position of the bones in the joint during motion in three dimensions is required. Fluoroscopy is a medical imaging technique that produces real-time, two dimensional images of an object, whereas, computer tomography (CT) is an imaging technique that produces static three dimensional image of the scanned object. The two dimensional fluoroscopy frames and the three dimensional CT scan images are not individually sufficient to provide complete information about the position of the bones in the joint in motion as a function of time. Three dimensional position data over time can be derived using the static CT scan images and time sequence information from video fluoroscopy images together.

A digitally reconstructed radiograph (DRR) is a simulation of a conventional 2-D X-ray image or a fluoroscopy frame created from CT data. Once the CT scan and a fluoroscopic video of the joint in motion are generated, a DRR image can be generated using the CT scan image. This requires a simulation of the fluoroscope with the X-ray source and the X-ray film positioned at equal radial distances on opposite sides of the origin. The CT cube is placed at the origin in between the X-ray source and the film. The distance between the C-ray source and the film is fixed and known. The cube in the center can translate in three dimensions: x, y, and z. The film can rotate while the X-ray source can move in azimuth and elevation dimensions. This simulation is illustrated in Figure 1.1. The projection of the X-rays from the simulated source onto the film through the CT cube produces a DRR image. Different values of X-ray source's azimuth and elevation, the film's rotation and the cube's x, y, and z co-ordinates generate different DRR images. If a generated DRR image matches the fluoroscopy frame, the position of the cube, and the orientation of the X-ray source and the film are known. Using this information, a three dimensional rendering of the joint in motion can be constructed.

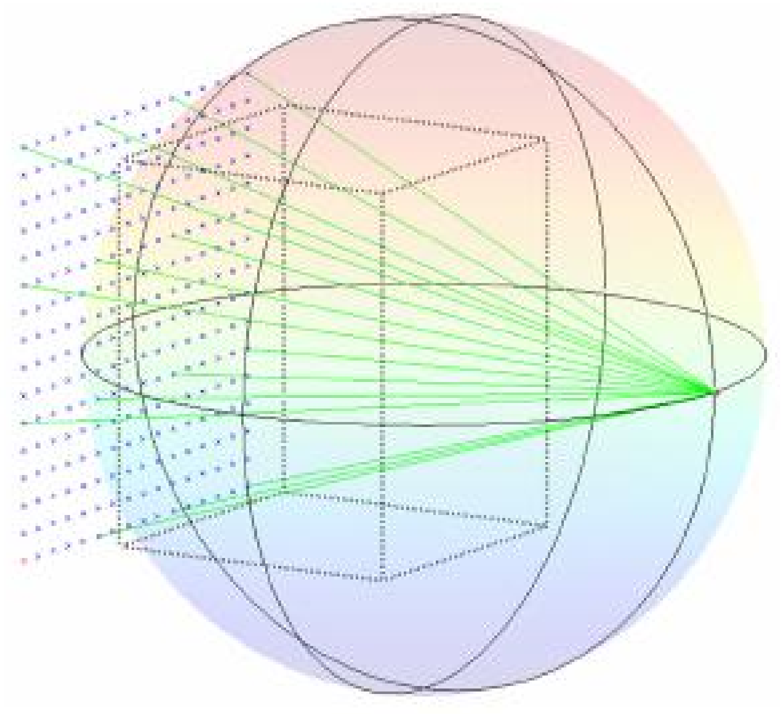


Figure 1.1: Geometric configuration of the simulated fluoroscope [1]

In order to accomplish this, several algorithms are used for different operations including generating DRR images, extracting the edges of DRR images, and comparing the edge images. These algorithms are based on the work of several researchers [2, 3]. Previous experiments used a Monte Carlo search technique to search for the matching DRR image, given an X-ray fluoroscopy image [1]. These experiments were conducted to search for azimuth and elevation of the camera and the rotation of the film. They have demonstrated that this can be achieved to an accuracy of 0.5 degree rotation, azimuth and elevation. The process, however, is very time consuming and calls for advance search algorithms to reduce the time taken for finding the orientation of the X-ray source, the X-ray film and the position of the bones. This thesis will involve using swarm intelligence techniques in a parallel computing environment in the attempt to achieve faster results.

1.2 Swarm Intelligence in a Parallel Computing Environment

Swarm intelligence is artificial intelligence based on the collective behavior of decentralized, self-organized systems. Swarm intelligent systems are made up of a population of entities, which together perform an activity and converge to a solution by interacting with one another. Even though there is no central control, their instinctive behavior and communication among one another leads to a collective behavior. Swarms move through the search space and their inherent intelligence leads to the solution set. Ant colony optimization (ACO), bees algorithm, particle swarm optimization (PSO), and stochastic diffusion search (SDS) are some of the techniques which exhibit swarm intelligence [5, 9, 11, 13]. These techniques have been used in many optimization problems like the traveling salesman problem, telecommunications routing, factory scheduling, etc. Swarm intelligence techniques are known to be cost-effective, robust and simple.

Based on a review of some of the swarm intelligence techniques, bees algorithm and

PSO have been chosen to solve the search problem in fluoroscopic analysis of knee joint kinematics [5, 9]. Bees algorithm is a biologically inspired technique based on the food foraging behavior of a swarm of honey bees. The bees in the group are divided into scout bees and worker bees. Scout bees go on a random search for flower patches and come back to the bee hive to report their findings. The quality and distance of each of the flower patches found is evaluated. Finally, the best ones are chosen and worker bees are sent to them to collect nectar. The second algorithm used in this thesis, the PSO technique, is influenced by the flocking behaviour in birds. It consists of several particles organised into close knit groups or neighborhoods. The particles move in the search space searching for the destination by interacting with one another and they collectively arrive at the destination. It is based on the communication and collaboration among particles which adjust their position and velocity accordingly and move through the search space to find the solution.

Bees algorithm and PSO are suitable methods to solve the six dimensional search problem because the entities in the system can begin the search independently and with periodic communication, they can guide each other to go in the right direction toward the destination. They can be easily mapped to the search problem discussed in Section 1.1. It is also easy to implement them in parallel. Each entity, a bee or a particle, can map to a process in the parallel computing system. Both these methods depend on random number generators just like the Monte Carlo method. But they are more easily parallelizable than the Monte Carlo method which assumes a very sequential approach. This thesis entails implementing the solution using both these techniques in a parallel computing environment.

1.3 Prior Research

This thesis is an extension to the research done by Scott et al [1]. The goal of the research conducted by Scott et al., was to develop a method for collecting accurate, real time three dimensional kinematic data of bones and joints using two dimensional X-ray images resulting from a video fluoroscopy technique combined with the static three dimensional Computer Tomography data [1]. The process used in their research involved creating virtual X-rays from the CT image through the digitally reconstructed radiograph projections. Traditional methods used to achieve kinematic data of the joints required human intervention to initialize the starting position. The method they proposed was minimally invasive and used Monte Carlo search technique with a variable search range. The search range decreased as matching between the target image and the image created with known position values improved. This was an optimization search. This is a very efficient method for small ranges of postion values.

Considering the extent of the allowed movement of the knee joint within the fluoroscope and the allowed width of the angular movements of the X-ray source and the X-ray film, the potential range of the position values is enormous. Even an extremely efficient and well-suited search algorithm would take a tremendous amount of time to compute the required data if processed sequentially. Therefore, this calls for a suitable search algorithm in a parallel computing environment. The purpose of the research in this undertaking is to find suitable parallelized search mechanisms to speed up the process of determining the three dimensional position data of the bones and joints using the same image processing techniques that were used by Scott et al.

1.4 Thesis Statement

There are five objectives to this project. They are:

1. Find the DRR X-ray generated at arbitrary point for three degrees of freedom, using sequential search algorithm.
2. Find the DRR X-ray generated at arbitrary point for three degrees of freedom, using parallel search algorithm.
3. Find the DRR X-ray generated at arbitrary point for three degrees of freedom, using parallel search with separated bones.
4. Find the DRR X-ray generated at arbitrary point for six degrees of freedom, using parallel search algorithm.
5. Attempt to find the DRR X-ray of the two bones of a joint separately in real fluoroscopy X-ray.

The objectives begin with providing solution to a simple problem first and then progress toward solving a more complex problem involving real fluoroscopy X-rays instead of DRR images generated from known position values. A good test bed should be created to accomplish the objectives, in such a way that this could be used to test alternate joints to analyse other ailments. It should also be possible to use alternate fluoroscopic analysis methods and use the same test bed to verify the results.

CHAPTER 2

ALGORITHMS USED FOR FLUOROSCOPIC ANALYSIS

In this chapter, the meaning of fluoroscopic analysis and the algorithms used in the analysis are discussed.

2.1 Fluoroscopy

Fluoroscopy is an imaging technique that uses a fluoroscope to obtain real-time X-ray images of a moving body part such as bones, muscles, and organs such as the heart, lung, or kidneys. A fluoroscope consists of an X-ray source and a fluorescent screen between which a patient is placed. Figure 2.1 shows a modern day fluoroscope. Modern fluoroscopes couple the screen to an x-ray image intensifier and video camera allowing the images to be recorded and played on a monitor. Thus it is possible to view an X-ray movie of the body part being analyzed. The X-ray video is displayed using a continuous series of images produced at a rate of around 25-30 complete images per second. This is similar to the way conventional television transmits images. The fluoroscopic technique has been used in several medical procedures such as diagnosing problems in the digestive system, image guided injections into the spine or joints, analyzing cardiac functioning and guiding orthopedic surgery in placements of metalwork or implants.

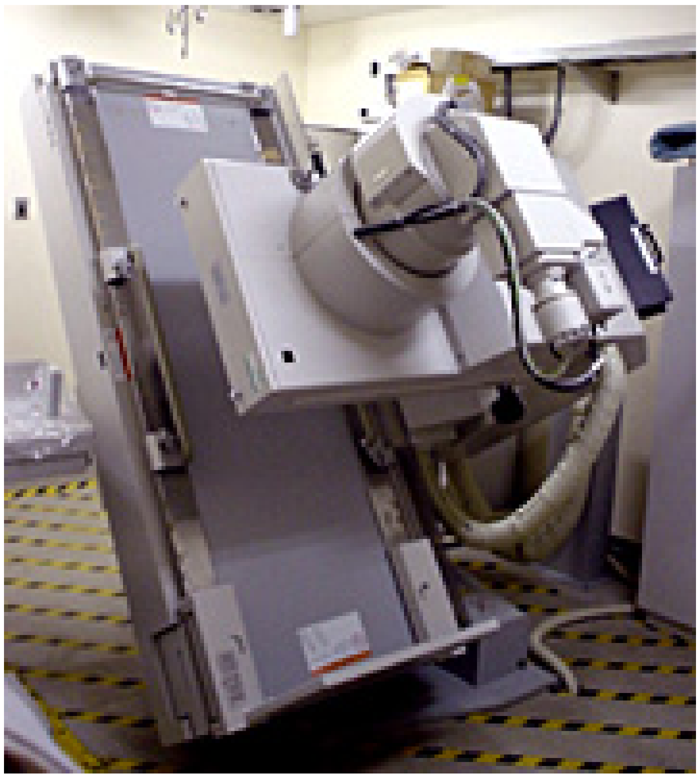


Figure 2.1: Fluoroscope [14]

2.2 Computer Tomography

Computed Tomography (CT) is a specialized medical imaging technique that uses X-rays. It is based on the concept of tomography. The word *tomography* is derived from two Greek words *tomos* which means ’slice’, and *graphein* which means ’to write’. This means a series of slices or cross sectional images are stacked together to construct a three dimensional model of the object being scanned. The X-ray source is rotated around a single axis to take a series of two-dimensional X-ray images of the object. These images are fed to a computer which performs digital geometry processing to create several individual images called slices. A three dimensional image is numerically reconstructed using these slices. Thus a CT scan is used to produce detailed pictures of the internal structure of a body part. Figure 2.2 shows a modern CT scanning equipment. The patient is placed on the horizontal surface and the scanner takes cross sectional images and computes a CT scan image of the required body part.

A CT scan can be used to study all parts of the body, such as the chest, belly, pelvis, or an arm or leg. It can take detailed pictures of several organs like liver, intestines, lungs, etc., as well as help in the study of bones, joints and spinal cord [16]. CT is often used to image complex fractures, especially ones around joints, because of its ability to reconstruct the area of interest in multiple planes. A CT scan image of knee joint can be helpful in diagnosing the problems in the kinematics of the knee joint and to measure an implant for the knee joint, if required. An example of a CT scan is shown in Figure 2.3 which is a CT scan image of a knee joint.

Figure 2.2: CT Scanner [15]

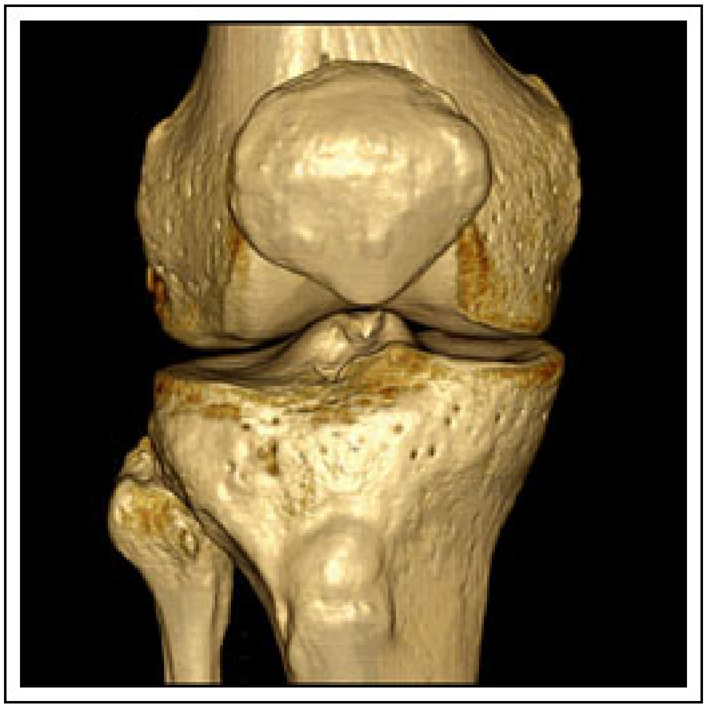


Figure 2.3: CT Scan of a Knee Joint

2.3 Knee Joint Kinematics

A knee joint connects the thigh with the leg. It joins the two bones: *femur*, the thigh bone and *tibia*, the leg bone. Figure 2.3 shows the anatomy of a knee. A knee joint is one of the largest and most complicated joint in the human body. It is a mobile joint which allows different kinds of movements between the bones. Flexion and extension movements occur between the *femur* and *tibia*. Medial and lateral rotation of the joint are also possible. The knee joint is the key to the movement of human beings from one place to another.

Knee disorders or injuries could make the movement of the knee very difficult and painful. Medical science offers knee replacement surgery to overcome very difficult knee problems which cannot be treated by other means. Surgery or any kind of knee treatments require accurate diagnosis and measurements. X-rays and CT scans are used to study the structure

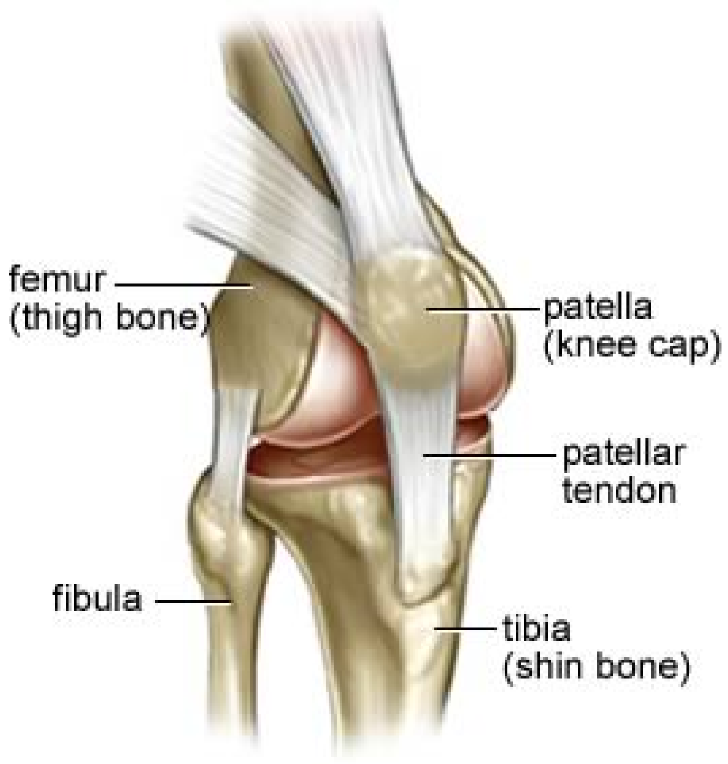


Figure 2.4: Anatomy of a Knee [17]

of the knee and take appropriate measurements. However, sometimes, diagnosis requires more than just static images and measurement of the joint structure. The different kinds of movements possible with the knee joint, the friction between bones connected during motion, the impact of the surrounding ligaments and the knee cap and other biological factors increases the complexity of diagnosis. Therefore the structure of the knee joint has to be studied in vivo. Study of the knee joint kinematics is crucial to diagnosing knee problems. The positions of the bones and the joint in motion have to be determined. In addition to the diagnosing knee joint conditions, this kind of information assists in the diagnosis of any other joint, several medical procedures like tracking of surgical instruments during a surgery, analyzing the wear and tear of the implants already fitted into the joint, and other joint related injuries and surgeries.

A CT scan and an X-ray are sufficient to determine the position of the bones in a stationary postion. However, doing the same while the joint is in motion is a very difficult task. In the past, invasive position capture techniques were used. Research conducted by Scott et al. developed a method to determine three dimensional real-time kinematic data of the knee joint without using invasive methods [1]. The subsequent sections in this chapter will describe how this was achieved. The goal of the research involved in this thesis is to decrease the time taken to conduct this operation.

2.4 Fluoroscopic Analysis

The process of determining three dimensional position data over time can be achieved using 3-D CT data and 2-D fluoroscopy images. This process is called fluoroscopic analysis. There are several steps required in this process. Figure 2.5 provides a block diagram of the major steps involved.

Consider a fluoroscopy environment in which the knee joint is placed. Refer to Figure 1.1 in Chapter 1. The knee joint is placed in between the X-ray source and the X-ray film. The position of the knee joint is affected by the lateral movements of the joint in any of the three directions (x, y, and z) about the origin. Secondly, the position is influenced by the azimuth and elevation of the X-ray source and thirdly, by the rotation of the X-ray film about a central axis. The knee joint can also change its azimuth, elevation and rotation values, but in this experiment, those values for the joint are kept constant, while the film and the X-ray source are rotated instead. Thus, values of the six dimensions have to be measured over time in order to obtain the real time positional information of the knee joint.

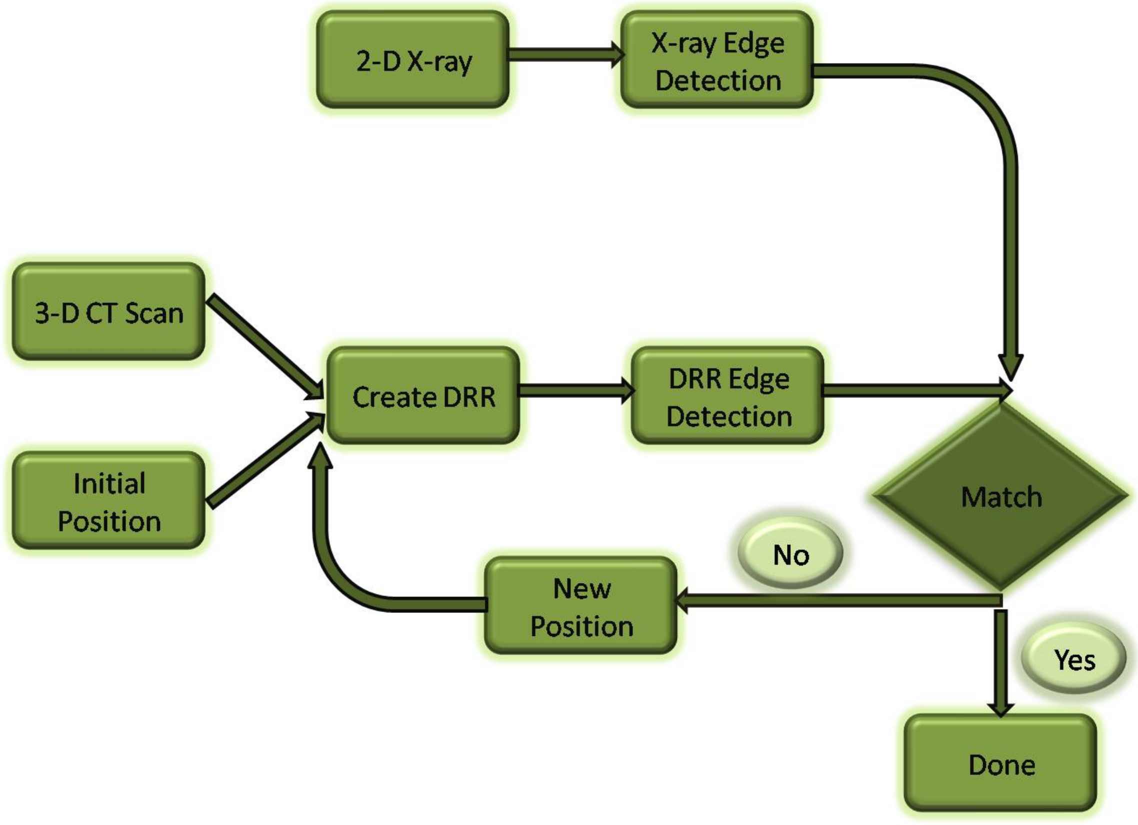


Figure 2.5: Procedure for Fluoroscopic Analysis

Using the static CT scan image as the input, the 3-D real time position data of the knee joint has to be constructed. In the first step, virtual X-ray images of the joint at different positions are created using the CT data. Then, the essential features of the image which are useful to determine the position are extracted. The essential features of the target X-ray frame from the fluoroscopy video images are also extracted. In the next step, the virtual image is compared with the target image. This process is continued until there is a match. A match indicates the position of the joint for the first target fluoroscopy image. The whole process should be repeated for each fluoroscopy image in the video.
Thus this process provides a rendering of the 3-D positional data over time. Sections 2.4.1 to 2.4.4 provided details for each of the above steps.

2.4.1 DRR Image Creation

A Digitally Reconstructed Radiograph (DRR) is a simulation of a 2D X-ray image, created from CT data. Ray tracing involves constructing a DRR from a CT image by calculating the radiological path through a three-dimensional CT array. Siddon's algorithm [2] is used for this purpose. This method considers the CT data as consisting of the intersection volumes of three orthogonal sets of equally spaced, parallel planes. The intersection of the ray with the volume elements (voxels), are obtained as a subset of the intersections of the ray with the planes. For each voxel intersection length, the corresponding voxel indices are obtained and the products of the intersected length and voxel density summed over all the intersections yields the radiological path and attenuation. The ray tracing mechanism employed here is similar to the ray tracing used in computer graphics to generate 3-D images. But in computer graphics, the transmittance or the path of the light rays through the pixels is used to generate the image, whereas, in fluoroscopic analysis, the reflection of the X-rays on the film is used to generate the DRR image.

2.4.2 Edge Detection

Edge detection of an image is used to identify points in the image at which the brightness of the image changes sharply or where there are discontinuities in surface orientation. By applying an edge detector to an image, the amount of information contained in the image can be significantly reduced by filtering out the less relevant information and retaining only the structural properties of the image. Using edges for comparison increases the probability of finding a match between a DRR image and a fluoroscopy image, as opposed to the comparison of the images themselves whose X-ray intensities may not match. This also decreases the amount of data to be processed and thus significantly simplifies the computations that involve interpreting the information contained in the image. In our implementation, Sobel's edge detection method is used [4]. This method finds the edges in an image using the gradient filter method.

In Sobel's edge detection method, a pixel location is declared an edge location if the value of the gradient exceeds some threshold. Edges will have a higher intensity gradient than uniform regions. Whenever the gradient value exceeds the threshold value, an edge is detected. Sobel's edge detector uses two 3x3 convolution masks. The two masks are applied to each pixel, one with a horizontal trend and one with a vertical trend. The convolution mask, which is much smaller than the actual image, is slid over the image, manipulating a square of pixels at a time. The Sobel masks Gx and Gy for the two gradients are shown in Figure 2.4.2. The Gx mask highlights the edges in the horizontal direction and the Gy mask in the vertical direction. After taking the magnitude of both, the resulting output detects the edges in both directions.

-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1

Figure 2.6: Sobel Gx and Gy Masks

The magnitude of the gradient is calculated using

$$|G| = \sqrt{Gx^2 + Gy^2}. \tag{2.1}$$

An approximate magnitude can be calculated using

$$|G| = |Gx| + |Gy|. \tag{2.2}$$

2.4.3 Image matching

The target X-ray fluoroscopic frame has to be compared with a DRR image generated using arbitrary camera and film orientation, and cube displacement values. Both the images are converted to edge images using the Sobel filter and then compared. Our implementations involve several such comparisons with images generated using different camera and film orientation and cube displacement values. A contour matching algorithm is used to evaluate the overlap of two edge images [3]. The matching score resulting from a comparison is a value between 0 and 1. A value of 1 indicates a perfect match. The matching score is obtained by multiplying the two images together pixel by pixel and summing the result and then normalizing by the sum of the predicted image. If J(x,y) is the input contour image and K(x,y) is the predicted contour image, then contour match score (CMS) is calculated using

$$CMS = \frac{\Sigma_{(x,y)} J(x,y) K(x,y)}{\Sigma_{(x,y)} K(x,y)}. \tag{2.3}$$

2.4.4 Searching

The position of the knee joint is defined by six values, viz., the x, y, and z co-ordinate values and the azimuth, elevation and rotation values. Since each of these values have a large range, a simple brute force method of searching will be highly inefficient. Each iteration of the search is a time-consuming process, because it involves creating a virtual image with known position values using ray tracing, extracting the edges and comparing it to the target image. Therefore, the goal should be to reduce the number of iterations of the search as well as the range of search with each iteration.

The measurement of the actual fluoroscope gives the range of values possible for each of the six positional parameters. The azimuth values of the X-ray source can range from -180 to +180 degrees and the elevation can be any where between -90 to +90 degrees. The film can rotate from -90 to +90 degrees. The knee joint or the hypothetical CT cube that is used in our experiments has a wide range of translation possible. Typically the object being scanned is placed close to the center and hence, for the tests conducted in this project, -5mm to +5mm translations from the center, on each axis, are considered.

Efficient search algorithms, like binary search cannot be effectively applied to a six dimensional problem of this proportion. With a binary search algorithm, we can only divide the values for one of the six dimensions into half. Even though this can potentially cut down the number of permutations of the six values to half of them, the number of remaining permutations is still dramatically large. And moreover, there is simply no way to sort the values for any of the six parameters due to the nature of the comparisons performed in the search. All search algorithms involve comparisons of the elements being searched in one way or another. In this case, we compare two images and the matching algorithm used for this comparison produces a match score between 0 and 1 for each comparison. The values of the six positional parameters are known when the score is 1. The images are being compared and not the six values. The images cannot be sorted in any manner. Thus this indicates that the usage of any search algorithm which depends on sorted elements is ruled out.

Some search algorithms that use hashing methods, require that the elements are organizable into some form of groups. The search problem presented in this project does not fit this category either. It is not practical to use deterministic search algorithms for this problem. Monte Carlo search methods that depend on random sampling and a few Swarm Intelligence algorithms are useful for this kind of search problems. The details for the Monte Carlo and the Swarm Intelligence algorithms are presented in chapter 3.

CHAPTER 3

SEARCH ALGORITHMS

In Chapter 2 the necessity of an efficient search algorithm in fluorosocpic analysis was explained. This chapter provides details of the algorithms adopted in this thesis. In addition to the Monte Carlo method that was used in prior research, the two SI techniques used in this thesis, the Bees Algorithm and Particle Swarm Optimization are presented here.

3.1 Monte Carlo Search Technique

The Monte Carlo search method is a non-deterministic method which depends on random sampling. Monte Carlo method is generally used to solve problems that are difficult to solve using deterministic methods. This method provides approximate solutions to a variety of mathematical problems by performing statistical sampling experiments on a computer. This method solves problems by generating random numbers and using the fraction of the numbers obeying certain properties to predict if the solution has been reached or not.

Stanislaw Ulam, a Polish born mathematician, is given the credit for inventing the Monte Carlo method. The first paper on the Monte Carlo method was written by Ulam and Metropolis [18]. This method is named after the Monte Carlo casino in Monaco. Although this method became popular during the life of Ulam and Metropolis since 1944, it is said to have been used a century ago. Monte Carlo methods have been heavily used in a wide

```
1. Initialize the current parameter value to zero.
   // This current value is evaluated based on the expected
   // target value in each iteration.
2. While (stopping criterion not met)
3.    Generate random values and add it to the initial value.
4.    Evaluate the fitness of the value obtained in step 3.
5.    Negate the random values and add it to the current value.
6.    Evaluate the fitness of the value obtained in step 5.
7.    Select the better of the two values. (This becomes the current value
8. End While.
```

Figure 3.1: The pseudocode for the Monte Carlo search algorithm.

variety of fields ranging from economics to nuclear physics. It can be used for risk analysis, financial predictions, fluid dynamics, and video games, to name a few.

In the research conducted by Scott et al., the Monte Carlo method is used to search for the DRR image that matches the target image [1]. In order to search, a DRR image should be created with different values of camera's azimuth and elevation, and the film's rotation angles in each iteration. The DRR image thus created is compared to the target frame to find a match. Initially, the camera azimuth, elevation, and film rotation angles are set to arbitrary values generated using a random number generator. Random step values with a range of 0 to 10^o are generated for each iteration of the search. Two sets of orientation values are obtained by adding and subtracting the step values to the initial values. DRR images are then generated using both these sets. A contour match score is calculated for the comparison of each of these images with the edge image of the target fluoroscopy frame. The parameter values which produce a better score is retained and the path that is more promising is chosen. Figure 3.1 gives the pseudocode for the Monte Carlo search algorithm.

The search continues by producing two new sets of random steps which are added and subtracted to the current values obtained from the previous iteration. This cycle is repeated until a match is found and the score of the match is greater than 0.8. Thus in each iteration of the search, the search space decreases and the search gets closer to the solution. This results in values that are approximately equal to the desired values.

Once the preliminary search is completed, a fine search is performed with smaller step values. The fine search is used to search closer to the parameter values obtained from the previous round of search. It is assumed that the first stage of the search resulted in a value that is close to the solution and hence the next stage of the search is expected to obtain a more accurate solution by searching in smaller steps. After both the stages are completed, the resulting parameter values should be fairly accurate. The sequential implementation of this method has been able to generate values within 0.5^o of the desired parameter values.

3.2 Swarm Intelligence

The word *Swarm* means a group of insects which are close to one another and engaged in a common activity. In general, *swarm* is a term used to describe the behavior of a group or an aggregate of animals of similar size and body orientation. Swarming behavior can be observed in many animals such as bees, ants, other insects, fish, bacteria and even people. This phenomenon can be called by different names such as flocking in birds, schooling in fishes, societal behavior in human beings and growth of colonies in bacteria.

Swarm intelligence is a phenomenon, based on self-organization, where the group of insects or other beings, solve complex problems together. There is no single individual of the group that is the leader or in charge of the outcome of the entire operation. The swarm arrives at the solution collectively. Constant and consistent communication and feedback among the group, and adherence to common rules are essential to the success of swarm intelligence. Errors and randomness are the pillars of this system. Even if a few individuals fail to achieve their objective, the group as a whole can succeed.

Examples of this behavior can be observed in the food foraging behavior in ants and bees, and migration in birds and fishes. The swarming of bees and ants can also refer to the reproductive action of an entire colony, as opposed to the reproduction of individual members of the colony. Reproduction in mosquitoes involve a swarm activity. The males form a large swarm at sunset, and the females fly into the swarms to mate. Schools or shoals of fish swim together in a polarized manner to find food, mates or to migrate to different regions of the water body. Swarm intelligence has a strongly noticeable benefit for all the members of the swarm. Chances of foraging food, finding mates, avoiding predators, overcoming obstacles, migrating to a better place are made faster and easier by swarm intelligence.

The biologically inspired swarm intelligence concept, when applied to computing, can help solve plenty of computing problems. This is a branch of artificial intelligence where collaboration and co-operation among entities in a system is the key to solving a problem. The expression *Swarm Intelligence (SI)* was introduced by Gerardo Beni and Jing Wang in 1989 in the context of cellular robotic systems [19]. There are several SI techniques such as the Ants optimization algorithm, Bees algorithm, Particle Swarm Optimization, and Stochastic Diffusion Search. SI techniques can be applied in problems involving finding the shortest path, swarm-based data analysis, robotics, financial forecasting, search, network management, internet server optimization, etc. SI methods generally have simple rules and are easy to implement.

Though the Monte Carlo method is suited for the search problem in fluoroscopic analysis, it is very sequential in nature. It is difficult to implement it in parallel and hence not chosen for implementation in this thesis. The two SI techniques, Bees Algorithm and Particle

Swarm Optimization have been chosen for implementing the solution to solve the search problem. The group of entities in both the methods can be easily mapped to the group of processes in the parallel computing system. In addition to being easily parallelizable, they are also easy to implement. Sections 3.2.1 and 3.2.2 describe these two methods in depth.

3.2.1 Bees Algorithm

The Bees Algorithm is inspired by the behavior of a honey bee colony in nectar collection. It is a population based search algorithm [5]. This approach has been used to solve many optimization problems.

A honey bee colony has many worker bees who have specific roles. The main functions of the colony is to identify food sources and collect and store nectar in the bee hives. The task of finding good quality flower patches is taken up by a population of the bees called the 'scout bees'. The scout bees set out in search of flower patches in random directions. When they come back to the hive, they deposit the nectar collected in a cell and go onto the dance floor to perform waggle dance. *Waggle Dance* [7] is the figure-eight dance preformed by the scout bee to share the information about the distance, direction and quality of the flower patch. While dancing the bees move in the form of figure-eight. If a line passing through the sun and the designated dance area is considered as a reference line, the angle between that line and the direction of the waggle dance indicates the direction of the food source. It could in the direction of the sun, to the right or left of the sun. The duration of the waggle dance indicates the distance of the flower patch. Figure 3.2 depicts the bee waggle dance. All the scout bees perform the waggle dance after returning from the flower patches. After learning about the distance and quality of all the flower patches through the waggle dances, the flower patches are evaluated based on some criteria. A few best sites are chosen from these if their fitness value is above a certain threshold. A large number of worker bees are recruited to collect nectar from the most promising sites and the remaining worker bees are sent to the other flower patches. Sites that are very far and those that have very few flowers with nectar and pollen are discarded. The scout bees are sent again in search of new flower patches for the next round.

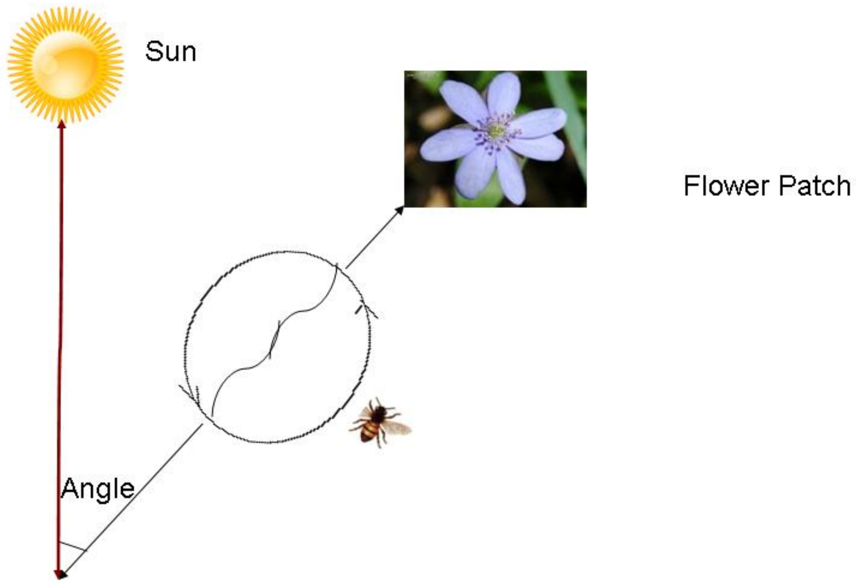


Figure 3.2: Waggle Dance

The bees algorithm in its simplest form requires a number of parameters [6], namely:

1. number of scout bees (n)

2. number of sites selected out of n visited sites (m)

3. number of best sites out of m selected sites (e)

```
1. Initialize the population with random solutions.
2. Evaluate fitness of the population.
3. While (stopping criterion not met) //Forming new population.
4.   Select sites for neighborhoodod search.
5.   Recruit bees for selected sites (more bees for best e sites)
     and evaluate fitnesses.
6.   Select the fittest bee from each patch.
7.   Assign remaining bees to search randomly and evaluate their fitnesse
8. End While.
```

Figure 3.3: The pseudocode for the bees algorithm in its simplest form [6].

4. number of bees recruited for best e sites (nep)
5. number of bees recruited for the other (m-e) selected sites (nsp)
6. initial size of the patches (ngh)
7. stopping criterion

The pseudocode is presented in figure 3.3.

The bees algorithm is illustrated using Figures 3.4 and 3.5. In Figure 3.4, the scout bees set out in random directions away from the main bee hive. Some flower patches are big while some are small. Some are very far from the hive while some are very close. There can also be more than one way to reach a flower patch. In figure 3.5 the returning scout bees perform the waggle dance in the bee hive and as a result of this, the majority of the worker bees fly to the larger and closer sites via the shortest route. Scout bees go in search of new sites.

3.2.2 Particle Swarm Optimization

Particle swarm optimization (PSO) [9, 10] is a population based stochastic optimization technique. This method is inspired by the social behaviour of a flock of birds or a school

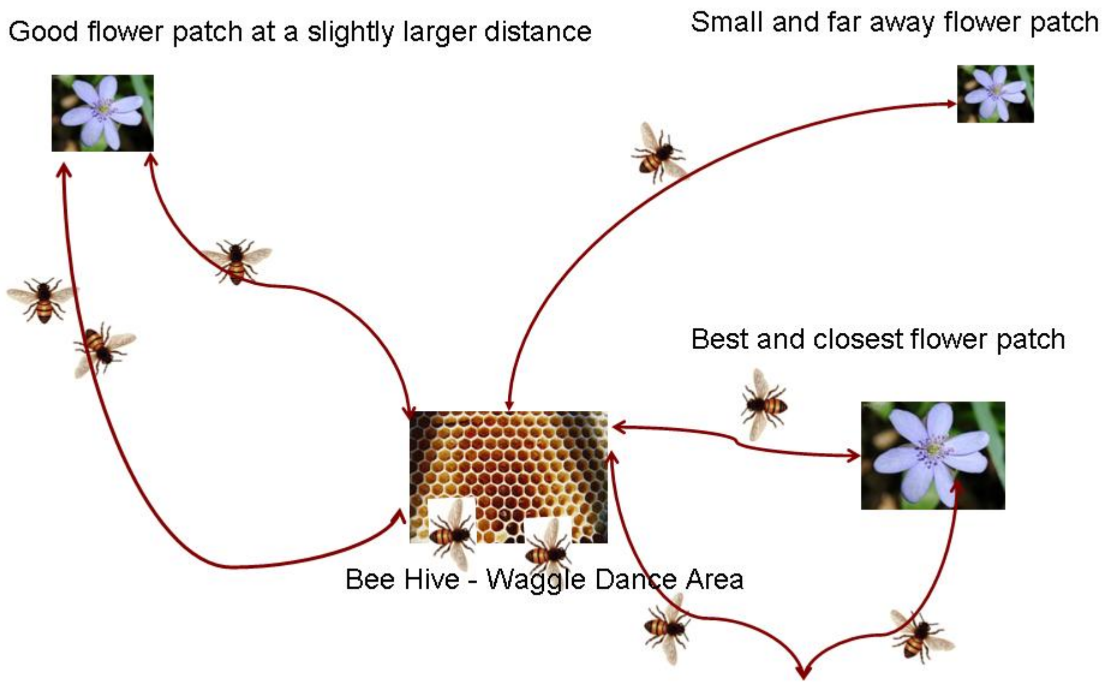

Good flower patch at a slightly larger distance
Small and far away flower patch
Best and closest flower patch
Bee Hive - Waggle Dance Area

Figure 3.4: Food foraging in bees - Scout bees are going in search of flower patches

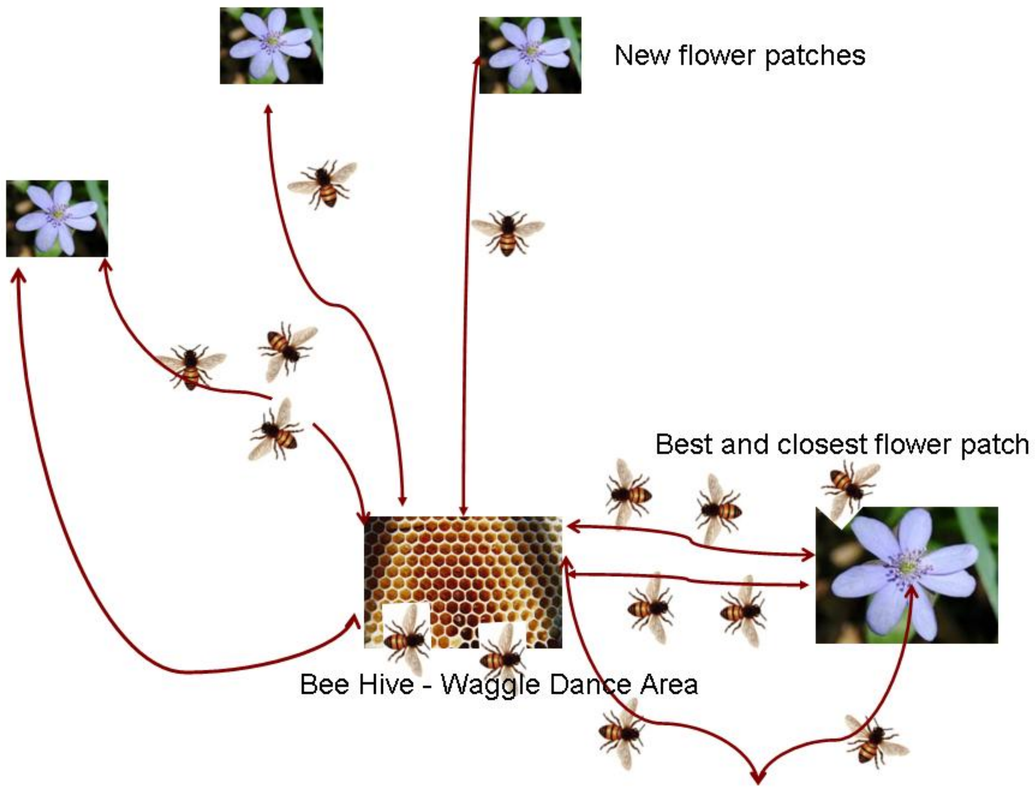


Figure 3.5: Food foraging in bees - Worker bees are going to the best flower patches

of fish. This concept was first explained by James Kennedy and Russell C. Eberhart [8]. A particle swarm system is similar to a social network and depends on a communication structure in the network. Just as a person who benefits from his interaction with the society, a particle in the swarm enhances its position in the system due to its interaction with other particles. People in a society talk to other people to find out their experiences, and try to solve their problems using the inputs given by others. Other people's views and attitudes might affect an individual's own opinion and future decisions. Some people are more influential than the others. There are various factors that can affect the social dynamics such as who the neighbors are, how they influence the direction an individual takes in his journey to a destination and his own progress at any moment.

Particle swarms are used to simulate social networks as well as several engineering applications. Particle swarm optimization technique can be used in search problems, where the particles in the swarm comminicate with one another and reach the solution faster than they would have if they were on their own. This technique has also been used to address other complex optimization problems, pattern recognition, data mining and image processing.

The swarm is typically modeled by particles in multi-dimensional space. Each particle has a position and a velocity with which it moves. All particles have a starting point and they move randomly in the search space. As the particles move from one position to another, they have to evaluate the fitness of that postion based on some criteria which determines how close they are to the solution. Navigation of the particles in the correct direction depends significantly on the accuracy of the fitness evaluation function. Periodically, a particle finds the best position it has held among the previous set of positions. It resets its position to the best position and continues the search. Each particle has a set of designated neighbors with which it should interact at regular intervals. The particles should not only keep track of the local best value, but also find out the best value in the neighborhood by

```
1. Initialize the particle positions and velocities.
2. While (global best value does not match solution)
4.   Move particle to a random position from the current position
     within a certain threshold from the current position.
5.   Store the local best value.
5.   Talk to neighbors and exchange neighborhood search information.
6.   Store neighborhood best value.
7.   Store global best value.
9.   Update position and velocity to the neighborhood best.
8. End While.
```

Figure 3.6: The pseudocode for the general particle swarm optimization.

communicating with other particles in the neighborhood. There is a particle designated in every neighborhood to keep track of the global best value. The particles adjust their postions at regular intervals based on the local, neighborhood and global best values. Thus they move towards those particles in their neighborhood, which have better fitness values. Over time they converge toward good positions and move about in the search space in close proximity to the global best and stop exploring the rest of search space. The rest of the search space which has a very low likelihood of containing the solution is discarded. The pseuodocode which explains this technique is given in Figure 3.6.

Figures 3.7, 3.8 and 3.9 illustrate this concept. Figure 3.7 shows a swarm of fifteen particles organized into three groups of five particles. The particles in the three groups are denoted by circles, squares and triangles respectively. In this figure, all the particles are initialized to some random values. In Figure 3.8, after fitness evaluation is conducted by all the particles, they are communicated to all particles in the neighborhood. These values help the particles to decide if they can continue from their current position or if they should follow a neighbor. This communication results in the movement of the particles toward the neighbor with the best fitness value. In ideal cases, after several iterations of this exercise, the particles converge toward the best global fitness value and thus converge to the solution. This stage looks similar to what is shown in Figure 3.9. All the particles then begin to search intensively around the best value until they find the solution.

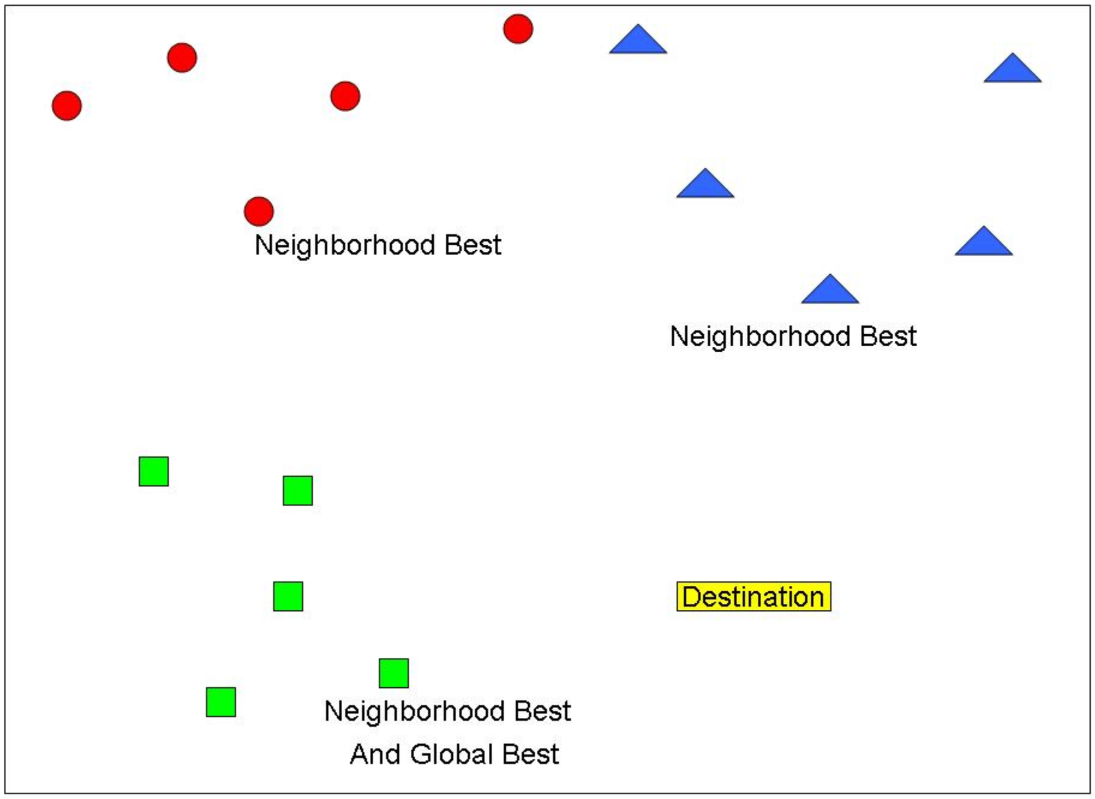


Figure 3.7: Particles in the swarm are moving randomly toward their destination

The application of the bees algorithm and the particle swarm optimization to the image search part of fluorocopic analysis has been described in detail in chapter 4.

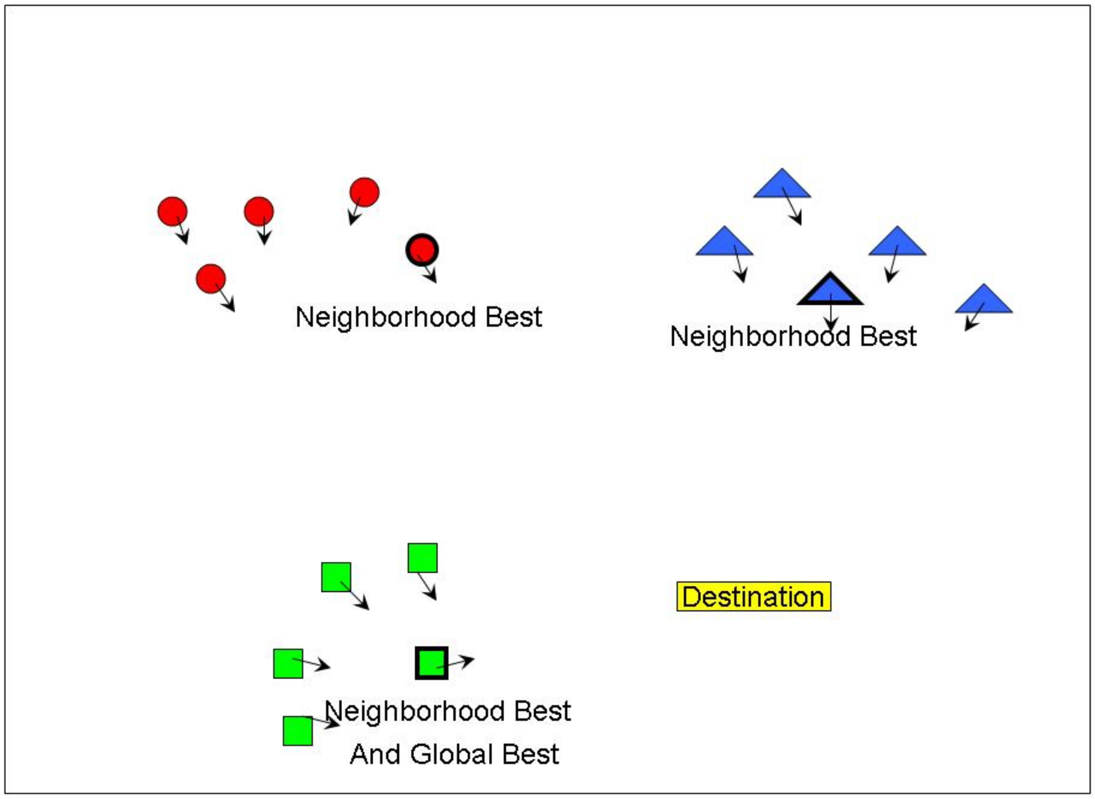


Figure 3.8: Particles are getting closer to the best position of the neighorhood

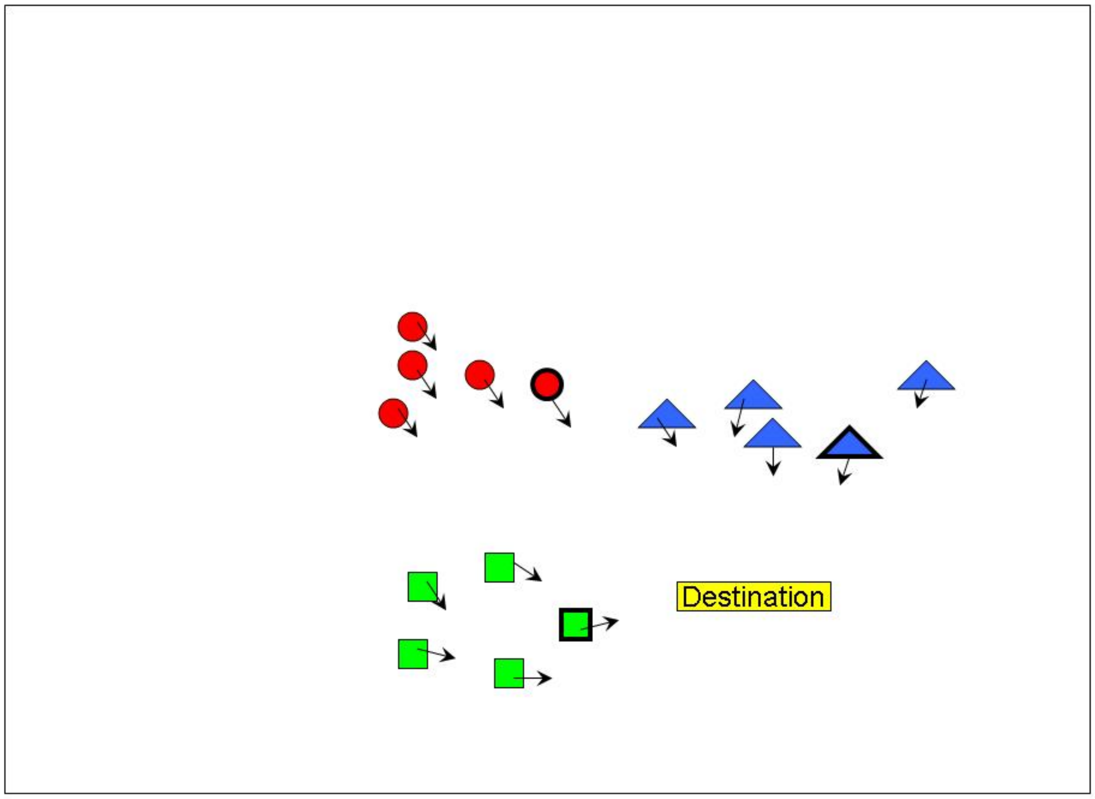


Figure 3.9: Particles are getting closer to the global best position

CHAPTER 4

DESIGN AND IMPLEMENTATION

The search problem involved in the fluorscopic analysis process has been decribed in Chapter 1. While chapter 2 provided an introduction to the world of fluoroscopic analysis and presented the computing problems associated with the process, Chapter 3 presented different methods to solve the search problem in flurosocpic analysis. In this chapter the design and implementation details as well as the pseudocode for the Bees algorithm and PSO method used for this thesis are provided.

4.1 Design Approach

This thesis consists of providing solutions to a search problem in a parallel computing environment. After finding the suitable algorithms for the search problem, a pseudocode for implementing it in the parallel environment was developed. Additionally, pseudocodes for the sequential version of the algorithm was also developed. In order to measure the speed and effectiveness of the parallel implementation, comparison with its sequential counterpart is required. After a detailed analysis of the fluoroscopic analysis application, and exploration of different search strategies, two swarm intelligence algorithms, Bees and PSO were chosen. Developing pseudocodes for these two algorithms for parallel and sequential implementation was the first step toward the design of this project.

The second step consisted of developing a good organization structure and an activity flow-chart which are very essential to a good design. All the steps involved in the process were studied. Each step required an separate component. For fluoroscopic analysis, different components are required for specific tasks, such as, generating DRR images from a given postion and a CT scan image, image filtering for extracting the edges of an X-ray or DRR image, image matching, and fluoroscopic searching. Figure 2.5 in Chapter 2 provides an activity flow-chart for fluoroscopic analysis that illutrates the different stages in the fluoroscopic analysis and the steps to transition from one stage to another.

The third step involved identifying the different entities in the system and designing data structures for each of them. The main entities are:

1. CT Volume - used to represent a CT scan image.

2. DRR Image - used to represent a DRR X-ray image.

3. Bee - used to represent a bee in Bees Algorithm.

4. Particle - used to represent a particle in PSO method.

5. Sobel filter kernels - used in Filtering method to extract edges of an X-ray image

These data structures are presented in Appendix B.

Figure 4.1 provides the pseudocode for the general fluoroscopic analysis. Figures 4.2 and 4.3 provide the pseudocode for the sequential Bees algorithm and PSO method respectively. The pseudocode for the parallel Bees algorithm is shown in Figures 4.4 and 4.5, while Figures 4.6 and 4.7 provide the pseudocode for the parallel PSO method.

```
\textbf{Fluoroscopic Analysis(CTCube, TargetXrayImage)}
1. TargetEdges = SobelFilter(TargetXrayImage)
2. score = 0
3. While(score < GOOD_MATCH_SCORE)
5.      x = Random()
6.      y = Random()
7.      z = Random()
8.      azimuth = Random()
9.      elevation = Random()
10.     rotation = Random()
4.      TestDRRImage = GenerateDRR(CTCube, x,y,z,azimuth,elevation,rotation)
5.      TestEdges = SobelFilter(TestDRRImage)
6.      score = ContourMatching(TestEdges, TargetEdges)
7. Match found.
   Position values are x, y, and z.
   Orientation values are azimuth, elevation, and rotation
```

Figure 4.1: Pseudocode for Fluoroscopic Analysis in the simplest form.

4.2 Build Infrastructure

Build infrastructure provides the capability to add new code to the project, compile and maintain the code. The project files are organized in such a manner that it facilitates the ease of compiling and executing the application. All the code and the input files are placed in a repository controlled by a version control system called Subversion. The main Makefile is placed in the main directory of the project. This makefile facilitates all the build actions required for the project. The makefile targets and the functions performed by them are given in Appendix A.

The directories are organized in such a way that it becomes convenient to maintain different libraries, input images, results, and configuration individually without affecting other parts of the project. The directory structure of the project files is provided in Appendix A.

```
\textbf{Sequential_Bees(CTCube, TargetXrayImage)}

n = Number of bees
m = Number of best sites
p = Number of bees chosen for the best sites (best bees)
q = n-p = Number of bees chosen as scout bees for the next iteration
r = Number of individual bee iterations

1. TargetEdges = EdgeFilter(TargetXrayImage)
2. While(BestScore < GOOD_MATCH_SCORE):
3.      Foreach Bee:
4.            For i = 0  to p:
5.            if(scout bee)
                    //In the first iteration of #1, all bees are scout bees
6.                  Generate random values for position(x,y,z)
                    and orientation(az,el,rt)
7.            else if(worker bee)
8.                  Generate random step values and
                    add them to the previous values.
9.            TestDRRImage = GenerateDRR(x,y,z,az,el,rt, CTCube)
10.           TestEdges = EdgeFilter(TestDRRImage)
11.           BeeScore = ContourMatching(TestEdges,TargetEdges)
12.     BeeScore = Calculate best of all the p scores

//Waggle dance section
13.     Rank the BeeScore of all the individual bees
14.     Choose the m best sites from the list based on scores.
15.     Assign p bees to m sites equally (worker bees)
16.     Assign q bees for random search (scout bees)
17.     Save the BestScore and the values associated with it
18. End While
19. [x,y,z,az,el,rt] associated with BestScore
    are the required position and orientation values.
20. End
```

Figure 4.2: Pseudocode for Sequential Bees Algorithm.

```
\textbf{Sequential_PSO(CTCube, TargetXrayImage)}

n = Number of particles
m = Number of neighborhoods
p = Number of particles in a neighborhood
r = Number of individual particle iterations

1. TargetEdges = EdgeFilter(TargetXrayImage)
2. Foreach neighborhood m:
3.      NeighborhoodBest = 0.0
4. GlobalBest = 0.0
2. While(BestScore < GOOD_MATCH_SCORE):
3.      Foreach Neighborhood m:
3.            if NeighborhoodBest < 0.1:
                  //This means, search randomly
4.                RANDOMSTEP = 0
5.            else if NeighborhoodBest < 0.4
                  //Add large random steps to previous value
5.                RANDOMSTEP =  LARGESTEP
5.            else if NeighborhoodBest < 0.6
                  //Add medium random steps to previous value
5.                RANDOMSTEP =  MEDIUMSTEP
5.            else if NeighborhoodBest < 0.8
                  //Add small random steps to previous value
5.                RANDOMSTEP =  SMALLSTEP
              Foreach neighbor p:
4.                For i = 0  to r:
5.                     if(RANDOMSTEP == 0):
                            //In the 1st iteration all particles search ra
6.                          Generate random values for position(x,y,z)
                            and orientation(az,el,rt)
7.                     else:
8.                          Generate random step values and
                            add them to the previous neighborhood best val
9.                TestDRRImage = GenerateDRR(x,y,z,az,el,rt, CTCube)
10.               TestEdges = EdgeFilter(TestDRRImage)
11.               ParticleScore = ContourMatching(TestEdges,TargetEdges)
12.               ParticleBestScore = the best of all the r particle scores
13.       NeighborhoodBest = the best of all scores in the neighborhood
14.     GlobalBest = the best of all scores
15. End While
18. [x,y,z,az,el,rt] associated with GlobalBest
    are the required position and orientation values.
19. End
```

Figure 4.3: Pseudocode for Sequential PSO.

```
Parallel_Bees(CTCube, TargetXrayImage)

n = Number of bees/processes
m = Number of best sites
p = Number of bees chosen for the best sites (best bees)
q = n-p = Number of bees chosen as scout bees for the next iteration
r = Number of individual bee iterations

1. TargetEdges = EdgeFilter(TargetXrayImage)
2. Barrier()  //Wait for all processes to arrive at this point.
3. While(BestScore < GOOD_MATCH_SCORE):
4.      Foreach Bee:
5.            For i = 0  to p:
6.            if(scout bee)
                  //In the first iteration of #1, all bees are scout bees
7.                Generate random values for position(x,y,z)
                  and orientation(az,el,rt)
8.            else if(worker bee)
9.                Generate random step values and
                  add them to the previous values.
10.           TestDRRImage = GenerateDRR(x,y,z,az,el,rt, CTCube)
11.           TestEdges = EdgeFilter(TestDRRImage)
12.           BeeScore = ContourMatching(TestEdges,TargetEdges)
13.     BeeScore = Calculate best of all the p scores
```

Figure 4.4: Pseudocode for Parallel Bees Algorithm.

```
//Waggle Dance section
14.     if( process_id != 0):
        //All processes other than process 0,
        //send their scores and 6 value to process 0
15.           Send(BeeScore,x,y,z,az,el,rt) to Process 0
16.     else if(process_id == 0):
17.           for(i =1 to n):
18.                 Recv(BeeScore,x,y,z,az,el,rt) from Process i

19.           Rank the BeeScore of all the individual bees
20.           Choose the m best sites from the list based on scores.
21.           Save the BestScore and the values associated with it

22.           for(i=1 to n):
                     //Send information about best sites to the the best bees
                     //Make some bees random.
23.                  Send(BeeScore, x,y,z,az,el,rt,ScoutOrWorkerFlag) to Proc
24.     if(process_id != 0)
25.           Recv(BeeScore, x,y,z,az,el,rt,ScoutOrWorkerFlag) from Process 0
26. End While
27. [x,y,z,az,el,rt] associated with BestScore
    are the required position and orientation values.
28. End
```

Figure 4.5: Pseudocode for Parallel Bees Algorithm, Ctd.

```
Parallel_PSO(CTCube, TargetXrayImage)

n = Number of particles/processes
    Every particle in the system has a particle_id
m = Number of neighborhoods/process groups
p = Number of particles in a neighborhood
    Every neighbor in a neighborhood has a neighbor_id
r = Number of individual particle iterations

1. TargetEdges = EdgeFilter(TargetXrayImage)
2. Create m ProcessGroups //Neighborhoods
3. Initialize all neighbors in a group/neighborhood
4. GlobalBest = 0.0
5. Barrier() //Wait for all particles to get initialized
6. While(BestScore < GOOD_MATCH_SCORE):
        //This is done by all neighborhoods.
7.      if NeighborhoodBest < 0.1:
8.            RANDOMSTEP = 0 //This means, search randomly
9.      else if NeighborhoodBest < 0.4
10.            RANDOMSTEP =  LARGESTEP  //Add large random steps to previous
11.     else if NeighborhoodBest < 0.6
12.            RANDOMSTEP =  MEDIUMSTEP  //Add medium random steps to previou
13.     else if NeighborhoodBest < 0.8
14.            RANDOMSTEP =  SMALLSTEP  //Add
small random steps to previous value
15.     For i = 0  to r:
16.            if(RANDOMSTEP == 0): //In the first iteration all particles se
17.                 Generate random values for position(x,y,z)
                    and orientation(az,el,rt)
18.            else:
19                  Generate random step values and
                    add them to the previous neighborhood best values
20.            TestDRRImage = GenerateDRR(x,y,z,az,el,rt, CTCube)
21.            TestEdges = EdgeFilter(TestDRRImage)
22.            ParticleScore = ContourMatching(TestEdges,TargetEdges)
23.            ParticleBestScore = the best of all the r particle scores
```

Figure 4.6: Pseudocode for Parallel PSO.

```
24.      if(neighbor_id != 0):
               //All neighbors send their score
               //and values to neighbor with id=0
25.            Send(ParticleBestScore, x,y,z,az,el,rt) to neighbor 0
26.      else:
               //neighbor 0 receives all the scores
27.            for( i=1 to p):
28.                 Recv(ParticleBestScore, x,y,z,az,el,rt) from neighbor i
29.            Store the NeighborhoodBest score
               //Send the neighborhood best score and values back to others
30.            for( i=1 to p):
31.                 Send(NeighborhoodBest, x,y,z,az,el,rt) to neighbor i

32.      if(neighbor_id !=0):
33.            Recv(NeighborhoodBest, x,y,z,az,el,rt) from neighbor 0

34.      if(particle_id != 0):
35.            Send(NeighborhoodBest, x,y,z,az,el,rt) to particle 0
36.      else:
37.            for( i=1 to n):
38.                 Recv(NeighborhoodBest, x,y,z,az,el,rt) from particle i
39.            Store the GlobalBest score
40. End While
41. [x,y,z,az,el,rt] associated with GlobalBest
    are the required position and orientation values.
42. End
```

Figure 4.7: Pseudocode for Parallel PSO, Ctd.

4.3 Implementation Details

Implementation of this application consists of several files of C code that implement all the algorithms required for each step in fluoroscopic analysis. The inputs to this application are DRR images of pre-determined postions and orientation or an actual 2-D X.ray fluoroscopic frame as the target images, CT scan image, and configuration files that describe the parameters required to define the algorithms. The program outputs the position and orientation of the Best matching score. By configuring the makefile to generate the debug version of the program, the program can be made to print out more information about each iteration of the search.

The algorithms are implemented as library plugins that can be dynamically linked to the main program. The main program performs all the main steps of extracting the CT volume information from the input CT image, and making dynamic calls to the required libraries for each of the other functions such as searching, generating DRR, etc. By implementing the algorithms as dynamically linked plugins, the flexibility for using any algorithm for a specific task is provided. This makes it possible to implement a new algorithm for any step and compile it independently. No changes are needed to any other part of the project to use the new plugin. The new plugin has to only be specified as a parameter while executing the application. This is done to allow further research and experimentation for the fluoroscopic analysis and to make it easy to explore different possibilities of solving this problem.

The environment details required to develop and execute this project is provided in Table 4.3. The instructions to execute this project are provided in Figure 4.8.

Operating System	Linux - CentOS release 5.2
Version Control System	Subversion 1.4.2
Programming Language	C
Libraries	mpich2, nrutils
Sequential Compiler	gcc
Parallel Compiler	mpicc
Parallel Computing platform	Beowulf cluster
Scripting Language	Korn Shell
Processor	..
Debugger	gdb
Memory Profiler	Valgrind

Table 4.1: Environment Details

```
Execution of sequential program:
    fluoro_s <path to search plugin(.so lib file)> <path to the input config

Execution of parallel program:
    mpiexec -l -n <num of processes> fluoro_p <path to search plugin(.so lib
            <path to the input config file>
```

Figure 4.8: Program Execution

An example config files for the Bees algorithm and the PSO method is provided in Figures 4.9 and 4.10 respectively.

4.4 Test Infrastructure

The configuration driven program execution adopted in this project makes it very easy to set up tests for this project. In this project, the specification of a particular algorithm and a set of parameters constitute a test case. By varying the choice of algorithm and the parameters required to define it, plenty of test cases can be generated. To generate a test case, the creation of a config file with the required details is necessary. This can be given as an input to the program executable to run the test case. This structure, makes it easy to

```
#Input CT image
CT_IMAGE=<path>/images/really_small_cube.txt

#Target DRR with known postion values
DRR_IMAGE=<path>/images/drr_image10_10_10_8_8_8.txt

#DRR generation algorithm
GENERATE_DRR_LIB=<path>/lib/libp_generate_drr_raytracing.so

#Filtering algorithm
FILTER_LIB=<path>/lib/libp_filter_sobel_bw.so

#Matching algorithm
MATCH_LIB=<path>/lib/libp_match_contour_bw.so

MAX_ITER=3

BEE_ITER=1

NUM_BEES=8

NUM_BEST_BEES=6

NUM_BEST_SITES=3

RANDOM_SEED=123

AZIMUTH_RANGE=180  #In degrees

ELEVATION_RANGE=90 #In degrees

ROTATION_RANGE=90  #In degrees

TRANSLATION_RANGE=20 #In pixels

TRANSLATION_FINE_STEP=3 #In pixels

CAMERA_FINE_STEP=5 #In degrees
```

Figure 4.9: Config file for Bees Algorithm

```
#Input CT image
CT_IMAGE=<path>/images/really_small_cube.txt

#Target DRR with known postion values
DRR_IMAGE=<path>/images/drr_image10_10_10_8_8_8.txt

#DRR generation algorithm
GENERATE_DRR_LIB=<path>/lib/libp_generate_drr_raytracing.so

#Filtering algorithm
FILTER_LIB=<path>/lib/libp_filter_sobel_bw.so

#Matching algorithm
MATCH_LIB=<path>/lib/libp_match_contour_bw.so

MAX_ITER=3

PARTICLE_ITER=1

NUM_PARTICLES=8

NUM_NEIGHBORHOODS=4

RANDOM_SEED=123

AZIMUTH_RANGE=180  #In degrees

ELEVATION_RANGE=90 #In degrees

ROTATION_RANGE=90  #In degrees

TRANSLATION_RANGE=20  #In pixels

TRANSLATION_FINE_STEP_LEVEL1=10  #In pixels

TRANSLATION_FINE_STEP_LEVEL2=5   #In pixels

TRANSLATION_FINE_STEP_LEVEL3=3   #In pixels

CAMERA_FINE_STEP_LEVEL3=10  #In degrees

CAMERA_FINE_STEP_LEVEL3=5   #In degrees

CAMERA_FINE_STEP_LEVEL3=1   #In degrees
```

Figure 4.10: Config file for PSO

write different configuration files and write a script to run the application with each config file and thus easily automate the test execution process. The output from each application can be redirected to a results folder and used to compare with subsequent test executions for the same test case in regression testing.

CHAPTER 5

TESTING AND RESULTS

The implementation details for the fluoroscopic analysis process and the test infrastructure developed are provided in Chapter 4. This chapter lists the efforts to test this implementation.

5.1 Test Cases

Software implementations require thorough testing for verifying the code and also validating the results obtained for the given inputs. Testing the implementation of fluoroscopic analysis process is required for the following purposes:

- To check the correctness of code by running it against various inputs and checking if the program runs to completion successfully.
- To verify if the parallel implementation does generate the required output in shorter time compared to the sequential implementation.
- To find the better of the two SI methods, Bees algorithm and PSO, for solving the search problem in fluoroscopic analysis.

The implementation for both the sequential and parallel versions provide the time taken for the search operation. This helps in comparing the differences in the time taken for the two versions. Several test cases are designed to test the implementation to enable the verification of all the the points mentioned above.

The implementation for Bees algorithm and the Particle Swarm Optimization algorithm require several parameters to be set. The Bees algorithm requires values for the number of bees, number of best sites to choose, the number of bees to allocate to each best site, etc. In the case of PSO, the parameters required are, number of particles, number of neighborhoods, number of members in a neighborhood and other details required for the algorithm to be successfully initialized. These parameter values are placed in a configuration file and given as input to the main program. Examples of the configuration files for Bees and PSO algorithms are provided in Figures 4.9 and 4.10 respectively. In this project, different test cases are derived by specifying different parameters for each algorithm. Therefore, by creating several configuration files, one for each set of test configuration, many test cases are constructed.

An attempt has been made to diversify the parameters in such a way that the program will be tested for different sizes of the groups (bees or particles), different sizes of the sub-groups, different position and orientation values of the target image, and so on. This would help in the determination of an optimized combination of parameters. For example, a particle population consisting of 36 particles grouped into 6 neighborhoods of 6 members each, might give faster results than a population grouped into 12 neighborhoods of 3 members each. In case of Bees algorithm, if 3 best sites are chosen and 10 bees are selected for each of the 3 sites from a population of 36 bees, while allowing the remaining 6 bees to search randomly, the grouping might provide better results than any other grouping of the bees.

Bees algorithm and PSO method are both swarm intelligence algorithms that use different mechanisms of grouping and sharing information among one another within their colony. Which method is more suitable for fluoroscopic analysis? How many entities are required for the search to provide optimal results without costing any severe communication overhead? What is an ideal grouping of the entities involved? Do smaller number of groups with large sizes work better than larger number of groups with small sizes? What kind of target images result in good solutions? Does the implementation provide good results only for a small subset of the target images or does it hold good for any range? The execution of tests attempts to answer these question that help determine the limitations and strengths of each algorithm by configuring several permutations and combinations of the parameter values. Tables 5.1 and 5.2 provide some of the test cases for the Bees algorithm and the PSO method respectively. Additionally, these combinations are executed multiple times with different target DRR images and for several number of maximum iterations.

Configuration/ Testcase	1	2	3	4	5	6
Number of Bees	12	24	24	24	36	36
Number of Best Bees	9	20	18	16	27	30
Number of Scout Bees	3	4	6	8	9	6
Number of Best Sites	3	2	6	4	9	6

Table 5.1: Test Cases for the Bees Algorithm

Configuration/ Testcase	1	2	3	4	5	6
Number of Particles	12	24	24	24	36	36
Number of Neighbors	4	6	4	3	4	6
Number of Neighborhoods	3	4	6	8	9	6

Table 5.2: Test Cases for the Particle Swarm Optimization method

In order to study the communication overhead caused in the parallel implementations of the two SI algorithms, a snapshot of the inter-process communication has been taken by adding extra logging to the program and Jumpshot 4.0 software has been used to visualize the communication in terms of send and receive operations between the processes by reading those logs. Figure 5.1 provides the visualization of the inter-process communication during the execution of Bees algorithm. The left pane of the picture lists the 12 processes with id 0 to 11. The timeline is seen at the bottom pane. It can be noted that for each iteration, all the bees or processes send their values to the master process (id = 0) and the master process sends a value after comparison and evaluation of all sent values, back to all the bees. This cycle repeats for every iteration of the search. The inter-process communication for PSO algorithm is quite different from this. Figure 5.2 provides the visualization for this communication. In this, it can be seen that in each iteration, subgroups communicate among themselves first, where all the subgroup members send message to and receive from the subgroup leader. The leaders then communicate with the master process. The communication overhead load on the master process is much lesser in the case of PSO then Bees algorithm. Considering the time taken for each iteration in generating a DRR image and comparing that to a target DRR image, the communication overhead caused by sending and receving matching score values at the end of each iteration is not very significant.

5.2 Results

Search algorithm implementations require several iterations of DRR image generation and their comparison to target image before finding the required position and orientation values. In sequential execution each iteration of the search is conducted sequentially and hence takes roughly *n.x* amount of time for *n* iterations, if x is the amount of time taken for one operation of DRR image generation and comparison. In both the parallel implementations, the parallel processes share the number of iterations of the search. In ideal cases, if *p* is the number of processes, then the time taken for n iterations reduces by a factor of *n/p*. Of course, the actual value will be less than this if we account for the process communication overhead.

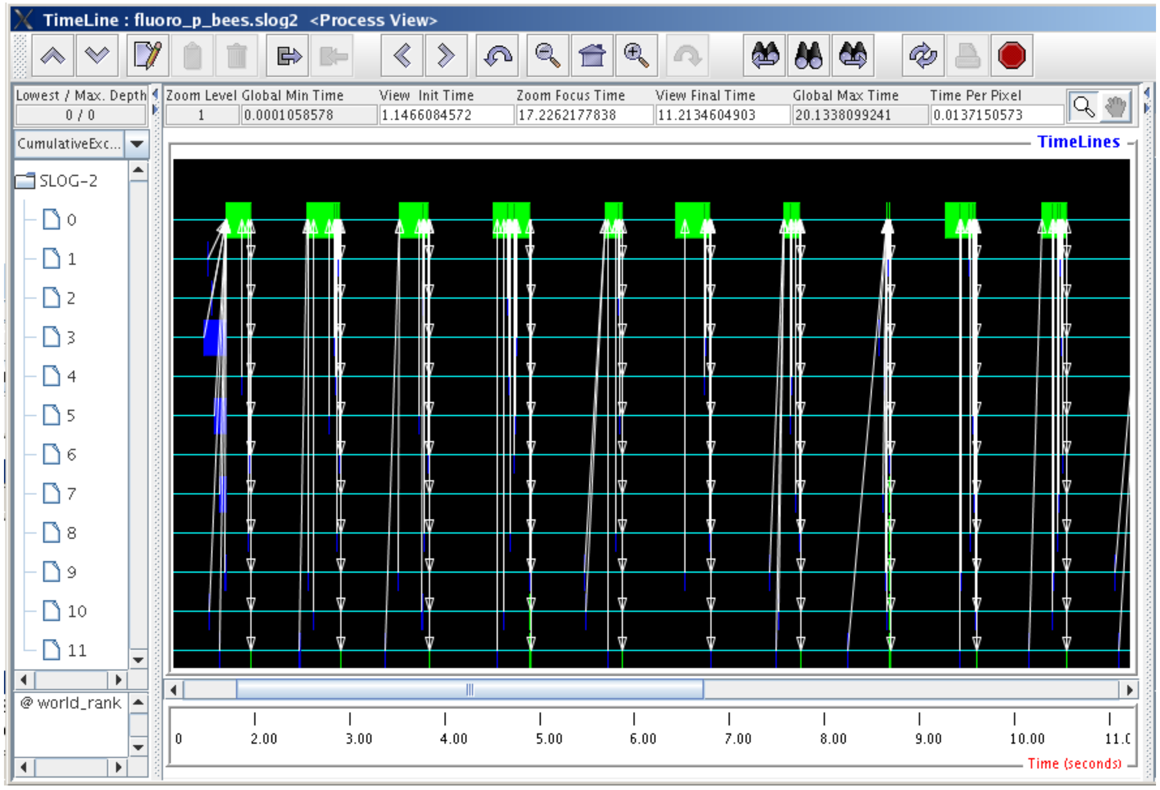
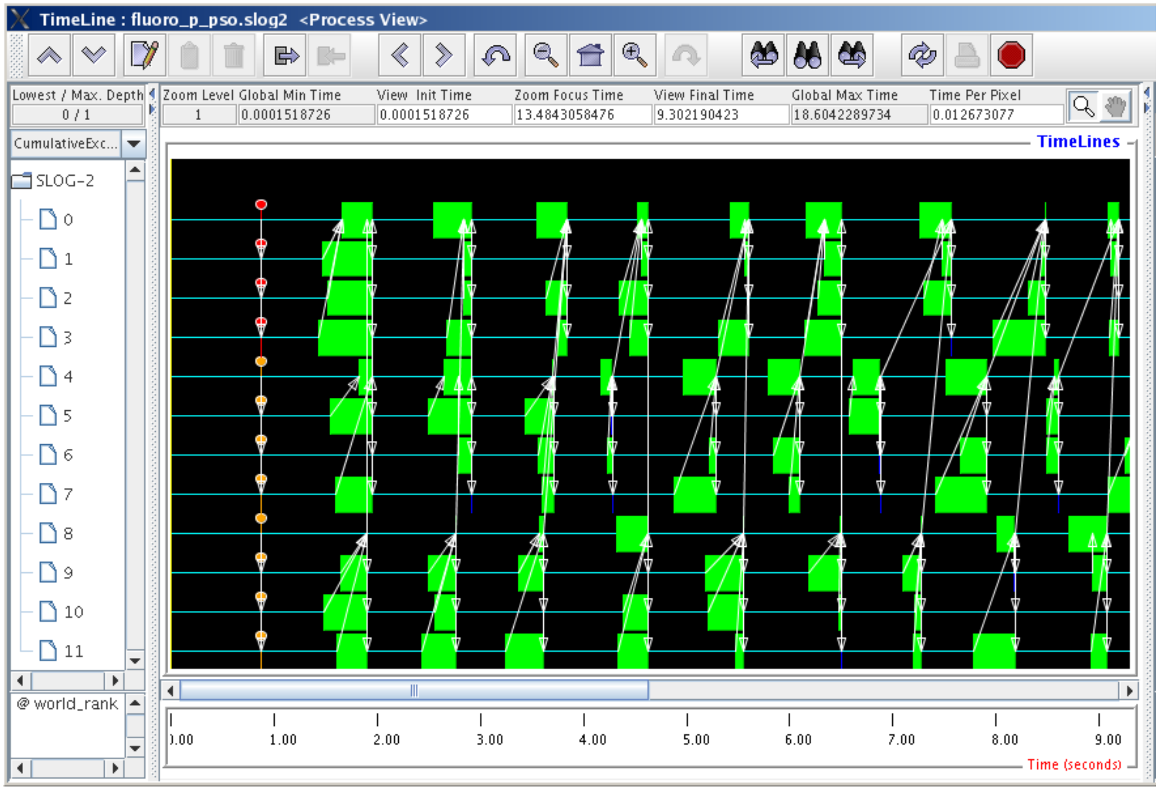


Figure 5.1: Communication Visualization for the Parallel Bees algorithm


TimeLine : fluoro_p_pso.slog2 <Process View>

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 1	1	0.0001518726	0.0001518726	13.4843058476	9.302190423	18.6042289734	0.012673077

Figure 5.2: Communication Visualization for the Parallel PSO algorithm

Mode of Execution	Algorithm	Max Iteration	Number of Entities	Time Taken(in secs)
Sequential	Bees	20	12	176
Parallel	Bees	20	12	173
Sequential	PSO	20	12	19
Parallel	PSO	20	12	18

Table 5.3: Timing Results

All the experiments confirmed that the the time taken by the parallel programs were less than the time taken by their sequential counterparts by a significant amount. One such instance is provided in Table 5.3.

These results help in verifying the first test objective and part of the second objective. The sequential and parallel programs have run without any crashes or execution errors for all the configuration values for large number of iterations. In addition, for the same number of iterations, the parallel programs are considerably faster than the sequential programs.

In order to get the results, each component of the fluoroscopic analysis process have to function accurately. One of the most crucial roles in affecting the results is played by the DRR X-ray image matching algorithm. The image matching algorithm chosen in this project provided unexpected matching score values in some cases. This has made it very difficult to obtain the expected position and orientation values from the search operation. Hence this prevents us from effectively comparing the sequential and parallel implementations and also comparing different methods of searching to find out which method takes us to the result in the shortest amount of time.

Owing to the difficulties in extracting the essential information from the complex X-ray images, and inaccuracy of the matching operation due to the presence of unsuitable information given by prior steps, the second and the third test objectives cannot be fully analyzed. The other reason could also be that the random position and orientation values used to generate the reference DRR images might not represent valid X-ray images. Due to this, all the subsequent processes of filtering and matching are bound to go wrong. There is scope for further research to improve the results and this is discussed in Chapter 6.

CHAPTER 6

CONCLUSIONS

6.1 Swarm Intelligence for Solving the Search Problem

Fluoroscopic analysis of knee joint kinematics consists of determining the position and orientation of the bones in the knee joint. In the experiments conducted in this thesis, a hypothetical fluoroscope with a CT cube placed between an X-ray camera and an X-ray film, is used. The objective of the thesis is to find the position of the CT cube and azimuth and elevation angles of the camera and the rotation of the film for a given CT scan image and a target fluoroscopy X-ray frame.

The focus of this thesis has been to develop faster search algorithms for finding the position and orientation information. This is done by implementing the two swarm intelligence techniques, the Bees algorithm and the PSO method for sequential and parallel execution. Both these methods are biologically inspired algorithms. While the Bees algorithm mimics the food foraging behavior of the bees, PSO method is inspired by the social behavior of a flock of birds or a school of fish. By adopting the communication and collaboration among a population of entities to solve a common problem, it is easier to reach a better solution faster.

Parallel computing is a suitable method for solving problems that are enormously time consuming. Fluoroscopic analysis is one such problem and the usage of parallel computing for solving this is well warranted. The Bees algorithm and the PSO method are easily parallelizable and are easy to apply to a problem of this nature. The experiments conducted to test the implementation provide sufficient information to indicate that parallel computing can significantly decrease the time required to search the position and orientation values in the fluoroscopic process. Several configurations of the Bees and the PSO algorithms have been successfully executed.

6.2 Future Scope

There is scope for further research in several components of the fluoroscopic analysis process. In order to derive accurate results, it is essential to have very efficient and effective methods for each of the analysis operations such as the DRR image generation, filtering of essential information from DRR or X-ray images, matching two DRR or X-ray images as well as searching for the position and orientation values from a given CT image and a target X-ray image. While the Bees and PSO algorithms appear to be promising solutions to the search problem, more analysis has to be carried out to find the best configuration to get optimum results out of these two methods, and this would be possible with accurate results from the other steps in the fluoroscopic process.

The intent of this fluoroscopic analysis process is to provide dynamic real-time information of the knee joint to assist in several diagnostic or surgical processes. This research can be extended to include Magnetic Resonance imaging (MRI) process to find alternate solutions. Additionally, the fluoroscopic analysis process can be applied to any joint in the human body to help diagnose and treat ailments of the joints. There is a huge potential for this process to truly help the medical community in the treatment of joint related problems.

BIBLIOGRAPHY

[1] Charles Scott and Elisa H. Barney Smith. *An Unsupervised Fluoroscopic Analysis of Knee Joint Kinematics.* IEEE Computer Society Washington, DC, USA, 2006.

[2] Siddon, R. L. *Fast calculation of the exact radiological path for a three-dimensional CT array.* Medical Physics 12(2), Mar/Apr 1985, pp. 252-255.

[3] Sarojak, M., W. Hoff, R. Komistek, and D. Dennis. *An Interactive System for Kinematics Analysis of Artificial Joint Implants.* Proc. of 36th Rocky Mountain Bioengineering Symposium, Copper Mountain, Colorado, 16-18 April 1999.

[4] Gonzalez, R. C., Woods, R. E. *Digital Image Processing.* Addison-Wesley Publishing Company, 1992.

[5] Pham D. T., Ghanbarzadeh A, Koc E, Otri S, Rahim S., and Zaidi M. *The Bees Algorithm.* Technical Note, Manufacturing Engineering Centre, Cardiff University, UK, 2005.

[6] Pham D. T., Ghanbarzadeh A, Koc E, Otri S, Rahim S., and Zaidi M. *The Bees Algorithm - A Novel Tool for Complex Optimisation Problems.* Proceedings of IPROMS 2006 Conference, pp.454-461.

[7] Von Frisch K. *Bees: Their Vision, Chemical Senses and Language.* (Revised edn) Cornell University Press, N.Y., Ithaca, 1976.

[8] J. Kennedy and R. Eberhart. *Particle Swarm Optimization.* Proceedings of the IEEE International Conference on Neural Networks, Perth, Australia, IEEE Service Center, Vol. 4. Piscataway, NJ, 1995, pp. 1942C1948.

[9] Parsopoulos, K.E., Vrahatis, M.N. *Recent Approaches to Global Optimization Problems Through Particle Swarm Optimization.* Natural Computing, 1 (2-3), pp. 235-306, 2002.

[10] Maurice Clerc. *Particle Swarm Optimization.* ISTE, ISBN 1-905209-04-5, 2006.

[11] Dorigo, M., Maniezzo, V., Colorni, A. *Ant System: Optimization by a Colony of Cooperating Agents.* IEEE Transactions on Systems, Man, and Cybernetics Part B,26(1): 29-46, 1996.

[12] Dorigo, M., Stützle, T. *Ant Colony Optimization.* MIT Press. ISBN 0-262-04219-3, 2004.

[13] Bishop, J. M. *Stochastic Searching Networks.* Proc. 1st IEEE Conf. on Artificial Neural Networks, pp 329-331, London, 1989.

[14] University of Virginia Health System. *World Wide Web.* http://www.healthsystem.virginia.edu/uvahealth/adult_radiology/fluoros.cfm

[15] Toshiba Medical Systems - Aquilion ONE. *World Wide Web.* http://www.toshiba-medical.eu/en/Our-Product-Range/CT/Systems/Aquilion-ONE

[16] WebMD. *World Wide Web.* http://www.webmd.com/a-to-z-guides/computed-tomography-ct-scan-of-the-body

[17] ACL Solutions - Anatomy of the Knee. *World Wide Web.* http://www.aclsolutions.com/
anatomy.php

[18] Metropolis, N., Ulam, S. *The Monte Carlo Method.* Journal of the American Statistical Association 44 (247): 335341, 1949.

[19] Beni,G., wang,J. *Swarm Intelligence in Cellular Robotic Systems.* NATO Advanced Workshop on Robots and Bialogical Systems, Tuscany, Italy, 1989.

APPENDIX A

README FILE

```
-------------------------------------------------------
                     Readme file
                  Fluoroscopy Project
-------------------------------------------------------

Sections:
    1.  Description
    2.  Steps Invloved in Executing the Project.
    3.  How to Compile the Project.
    4.  Makefile Features.
    5.  How to Run the Project.
    6.  Directory Structure of the Project.
    7.  How to Add a Plugin.
    8.  Code Documentation.
    9.  Naming Conventions.
    10. Scripts.
    11. Config Files.
    12. Results.
    13. Examples for Compiling and Executing the Project.

--------------------------------------------------------
1. Description
--------------------------------------------------------
This project performs an unsupervized fluoroscopic
analysis of knee-joint kinematics. The operations
involved include
    1. reading a CT scan image,
    2. generating DRR images,
    3. filtering the edges from a DRR image,
```

4. matching the edges of two different DRR images and having a score associated with it,
5. and searching for the right DRR image for a CT image, by using different combinations of the film and camera orientation and CT cube offsets.

```
-------------------------------------------------------
2. Steps Involved in Executing the Project
-------------------------------------------------------
```

The main program of this project relies on the availability of shared libraries for the following operations : generating DRR, filtering, matching and searching Below are a list of steps required to execute this project. Refer to relevant sections for details in each step. In order to use the Makefile provided as is, please follow the directory structure and the other requirements as mentioned for each step. Deviation from them would require custom makefiles or modification to paths in the makefile or whatever is required for the specific change.

1. Place the CT image file in the ’images’ directory.
2. Place a DRR image for that CT image in the ’images’ directory. The DRR image file for a CT image, can also be created by running the program create_target_drr with the required camera and film orientation and cube offsets. Refer section 3.A and 5.A.
3. Create an input config file which lists all the parameters required for running the program, such as number of iterations to run the search, path to the CT image, path to the DRR image, and any other parameters required for the algorithm chosen. Place this file in the ’config’ directory. Refer section 3.E
4. Compile all the plugins required for the instance of the program. Each plugin should be in a separate directory in the ’plugins’ directory. Place libraries (*.so files) in the lib directory and turn off the selinux security restrictions for those libraries. Refer section 3.B and 3.C.

```
    5. Compile the main program by specifying the
       compiler. Refer section 3.D.
    6. Run the program by providing the input config
       file. Refer section 5.B.

--------------------------------------------------------
3. How to Compile the Project
--------------------------------------------------------
$(FLUOROSCOPY) represents the path to the fluoroscopy
project source, the outermost directory of the trunk,
which consists of main makefile and all other
directories.

    A. To compile the create_drr_image program:
       cd /$(FLUOROSCOPY)
       make clean_create_drr
       make create_drr

    B. To compile all the plugins in the 'plugins'
       directory:
       cd /$(FLUOROSCOPY)
       make clean_plugins
       make plugins

       Note: This creates a .so file for sequential
       and parallel, for DOF=3 and 6, and all the
       permutations required.
       DOF = Degrees of Freedom

    C. To compile a single plugin:
       For any plugin, you can run 'make clean' followed
       by 'make all' to make all libraries that is possible
       for that plugin.  To only make a particular library,
       run make with required parameters.

    D. To compile the main program:
       cd /$(FLUOROSCOPY)
       make clean
       make main CC=<compiler>

       NOTE: Compiler should be 'gcc' or 'mpicc'.
    E. To create input config files:
       Run the relevant scripts from the 'scripts'
       directory or run the makefile with the right
       target to create the config files.

       make configure_bees #For Bees
       make configure_pso #For PSO
       make configure_montecarlo #For MonteCarlo


    F. To create lib config file[obsolete]:
       make configure_libs

       NOTE: This feature is preserved for any future
       use that might necessitate the need for a lib
       config file.

--------------------------------------------------------
4. Makefile Features
--------------------------------------------------------
The Makefile present in the main directory of the
project trunk, facilitates many operations.  Below is a
list of all the targets accepted by the makefile.
    make clean_all
        executes clean, clean_create_drr, clean_libs,
        clean_plugins, and clean_swapfiles
    make clean
        cleans the current directory and source directory
    make clean_create_drr
        cleans files associated with create_target_drr
        program from the source directory
    make clean_libs
        deletes all the libraries (*.so) files from the
        'lib' directory
    make clean_plugins
        cleans all the plugin directories
    make clean_swapfiles
        cleans all the swap files (*~) from all the
        directories in the project recursively.
    make create_drr
```

```
        compiles the create-target_drr program
    make plugins
        compiles all the plugins in the 'plugins'
        directory and copies all the libs to the lib
        folder
    make main CC=<compiler>
        compiles the main program. CC= <gcc|mpicc>
    make help
        prints all the targets accepted by make
    make configure_bees
        runs a script that creates an input config file
        for the Bees algorithm by taking user inputs and
        copies it to the config folder
    make configure_pso
        runs a script that creates an input config file
        for the PSO algorithm by taking user inputs and
        copies it to the config folder
    make configure_montecarlo
        runs a script that creates an input config file
        for the MonteCarlo algorithm by taking user
        inputs and copies it to the config folder
    make configure_libs
        runs a script that creates an lib config file
        and copies it to the config folder
    make documentation
        compiles and creates the code documentation
        and copies them to the doc folder.
```

Makefile for the main program:
The makefile for the main program is named Makefile_main and is present in the 'source' directory.

Makefile for the create_target_drr program:
The makefile for the create_target_drr program is named Makefile_drr and is present in the 'source' directory.

Makefiles for plugins:
Each plugin has a directory associated with it in the 'plugins' directory. Each plugin has its own makefile named 'Makefile'.

```
--------------------------------------------------------
```

```
5. How to Run the project
---------------------------------------------------------
    A. To run the create_target_drr program:
       create_target_drr
       <path to generate_drr plugin(.so lib file)>
       <CT image file>
       <resolution>
       <path to output drr file>
       [<azimuth> <elevation> <rotation>
       [<x offset> <y offset> <z offset>] ]

       NOTE: Example for resolution is 128, 256, 512 etc.
       Resolution for really_small_cube.txt is 128.

    B. To run the main program
       Sequential:
       fluoro_s
       <path to search plugin(.so lib file)>
       <path to the input config file>

       Parallel:
       mpiexec -l -n <num of processes> fluoro_p
       <path to search plugin(.so lib file)>
       <path to the input config file>

---------------------------------------------------------
6. Directory Structure of the Project
---------------------------------------------------------
$(FLUOROSCOPY) represents the main truck directory of
the project.

Create an environment variable for this as follows:
export FLUOROSCOPY=<fluoroscopy trunk absolute path>

$(FLUOROSCOPY)
        |
        |__matlab
        |
        |__config
        |
        |__doc
        |
        |__include
        |
        |__images
        |
        |__lib
        |
        |__results
        |
        |__scripts
        |
        |__source
        |        |__util
        |
        |__plugins
                |__filter_sobel_bw
                |__filter_sobel_gs
                |__match_contour_bw
                |__match_contour_gs
                |__search_parallel_bees
                |__search_parallel_pso
                |__search_sequential_bees
                |__search_sequential_pso
                |__search_sequential_montecarlo
                |__generate_drr_raytracing
                        |__sort_insertion
                        |__sort_qsort
                        |__sort_quick
```

Place the files in the relevant directories. The directory names are self-explanatory and clearly indicate what kind of files are expected to be present within them.

NOTE: The Makefile in the trunk directory does not compile the source in 'matlab' directory. 'matlab' directory contains the source for fluoroscopy in matlab language. All the other folders and files are associated with the C implementation for fluoroscopy.

```
---------------------------------------------------------
7. How to Add a Plugin
---------------------------------------------------------
Following are the steps to create a new plugin:
    1. Create a folder in the 'plugins' directory.
       Follow the naming conventions provided in section
       9.
    2. Place all the source files in the created
       directory.
    3. If the plugin depends on other libraries or other
       set of source files, or different sort techniques
       etc, make separate directories for each of them
       within the directory for the plugin. For example,
       generate_drr_raytracing has 3 directories for sort
       function. One of them is used to compile the
       library.
    4. Create a makefile for the plugin.
    5. In the makefile, create a target 'all' which
       generates multiple libraries files for all the
       required combinations.
       Refer to other plugin makefiles as examples.
    6. Create an input config file, as required for your
       libraries and execute the main program with the
       appropriate parameters.

---------------------------------------------------------
8. Code Documentation
---------------------------------------------------------
The open source tool Doxygen is used for generating the
code reference manual.  The configuration file Doxyfile
is present in the main truck directory. Currently, it is
configured to provide html, latex-pdf and perlmod-pdf
documentations. Edit this file to customize the
documentation to your needs. Please refer to the Doxygen
tutorial for details on installing, configuring and
running the tool.  Some basic steps are:

    To generate documentation, run the commands:
        make documentation

    To view the documentation, run any of the following
    commands:
        firefox docs/index.html
        acroread docs/CodeDocumentation.pdf
        acroread docs/CodeReferenceManual.pdf

    Alternate way to generate documentation:
        doxygen Doxyfile

    Alternate way to view documentation:
        cd html; firefox index.html
        cd latex; make; acroread refman.pdf
        cd perlmod; make; acroread doxylatex.pdf
```

Currently, the documentation includes all the plugins. If desired, separate documentation can be created for each plugin, by providing a Doxygen config file for each plugin and following the steps similar to the above instructions.

----
9. Naming Conventions

The names chosen for files or directories should be self-explanatory and easy to understand.

The names of the plugin directories should have a first name that indicates the operation performed or the interface implemented, e.g., filter, match, search etc. The second name should indicate if it is parallel or sequential if applicable. the next name should indicate the algorithm used, e.g., sobel, contour, bees etc. Any other variations should be next, e.g., bw for black and white, or gs for grayscale, etc. Separate the names by underscores.

The lib names should begin with ’lib’, followed by ’s’ or ’p’ to indicate if they are sequential or parallel. After this, separate each name with underscore and follow a pattern as mentioned above.

```
10. Scripts
--------------------------------------------------------
Below is the list of existing script files and its
functions:
    configure_libs.sh
        Used to create a lib config file [obsolete]
    configure_input_bees.sh
        Used to create an input config file for Bees
        algorithm
    configure_input_montecarlo.sh
        Used to create an input config file for Monte
        Carlo algorithm
    configure_input_pso.sh
        Used to create an input config file for PSO
        algorithm
    run_s_sanity.sh
        Used to run all the sequential programs for
        sanity checking. Basically to check if the
        program does not crash for the minimum number
        of iterations.
    run_s_limited.sh
        Used to run all sequential programs for
        testing the limited range for search parameters
    run_s_full.sh
        Used to run all sequential programs for
        testing the full range for search parameters
    run_p_sanity.pbs
        Used to run all the parallel programs for
        sanity checking. Basically to check if the
        program doesnt crash for the minimum
        number of iterations.
    run_p_limited.pbs
        Used to run all parallel programs for testing
        the limited range for search parameters
    run_p_full.pbs
        Used to run all parallel programs for testing
        the full range for search parameters
    run_s.sh
        Used by the run_s_* scripts internally.
    run_p.sh
        Used by the run_p_* scripts internally.
    helper_s.sh
        Used by the run_s.sh script internally.
    helper_p.sh
        Used by the run_p.sh script internally.

    NOTE: Check to see if any paths need to be changed
    in the scripts.  In some places, the path is set
    to the current user's working directory.

--------------------------------------------------------
11. Config Files
--------------------------------------------------------
    Input configuration files can be created using the
    steps specified in 3E.  The main program requires a
    config file as an input. It consists of all the
    algorithm parameters and also the target DRR image.
    In this project, a naming convention is used to name
    the config files for ease of use.

    $(FLUOROSCOPY)/config directory consists of all the
    config files.  The config files required for
    sequential tests are placed in 's_tests' folder, and
    files required for parallel tests are placed in
    'p_tests' folder.  Within these two folders, there
    can be many folders with separate config files, one
    for each test case.

    Note: Naming convention for config and result
    directories:

    Directory used for tests to just check if the
    programs run without error:
    Name = sanity

    Directory used for tests that limit the search
    space:
    Underscore delimited name contains:
    1. "limited"
    2. num of processors
    3. [num of neighbors]-[num of neighborhoods]  (for PSO)
       Use the second number for [num of best sites] (for Bees)
    5. num_bee_iter(for Bees) or num_particle_iter (for PSO)
    6. drr_image : "img" followed by az,el,rt
                    orientation
            and x,y,z translation values

    Directory used for tests that search the entire
    search space also follow the above naming
    convention except that the name starts with "full"
    instead of "limited".

--------------------------------------------------------
12. Results
--------------------------------------------------------
    $(FLUOROSCOPY)/results directory is created to
    contain all the output redirected from test
    executions.  The result files required for
    sequential tests are placed in 's_tests' folder, and
    files required for parallel tests are placed in
    'p_tests' folder.  Within these two folders, there
    can be many folders, one for each test case.

    The same naming convention used for naming config
    files may be used for result files too to easier
    maintenance and management of tests.

--------------------------------------------------------
13. Examples for Compiling and Executing the Project
--------------------------------------------------------
    A. Steps for the sequential program:
        1.  cd $(FLUOROSCOPY)
        2.  make clean_all
        3.  make plugins
        4.  make create_drr
        5.  cd $(FLUOROSCOPY)/source
        6.  ./create_target_drr
            $(FLUOROSCOPY)/lib/libs_generate_drr_raytracing.so
            $(FLUOROSCOPY)/images/really_small_cube.txt
            128
            $(FLUOROSCOPY)/images/drr_image.txt
            60 30 45 8 17 22
        7.  cd $(FLUOROSCOPY)
        8.  make clean_create_drr
        9.  make main CC=gcc
        10. fluoro_s $(FLUOROSCOPY)/lib/libs_search_bees_6d.so
            $(FLUOROSCOPY)/config/<config file>

    B. Steps for the parallel program:
        1. make clean_all
        2. make plugins
        3. make create_drr
        4. cd $(FLUOROSCOPY)/source
        5. ./create_target_drr
            $(FLUOROSCOPY)/lib/libp_generate_drr_raytracing.so
            $(FLUOROSCOPY)/images/really_small_cube.txt
            128
            $(FLUOROSCOPY)/images/drr_image.txt
            60 30 45 8 17 22
        6. cd $(FLUOROSCOPY)
        7. make clean_create_drr
        8. make main CC=mpicc
        9. fluoro_p $(FLUOROSCOPY)/lib/libp_search_bees_6d.so
            $(FLUOROSCOPY)/config/<config file>

    Note: The main differences between sequential and
    parallel executions are in the .so library names
    and the executable names.
--------------------------------------------------------
```

APPENDIX B

CODE DOCUMENTATION

This appendix chapter consists of the source code documentation for the essential components of the code. This documentation is extracted from the code with the help of a tool called *Doxygen*.

B.1 Data Structure Documentation

The following sections from Section B.2 to B.9 provide the main data structures used in the implementation of this thesis. Sections B.2 to B.6 are used in the fluoroscopic analysis process, Section B.7 is used in the configuration file reader program, and Section B.8 and B.9 are used in the SI search algorithms namely Bees algorithm and PSO method respectively.

B.2 _ctVolume Struct Reference

B.2.1 Detailed Description

This struct represents a 3-D Computer Tomography volume information used in the experimental fluoroscope.

Data Fields

- double ∗ **volume**
- **int size**
- **double width**
- **double height**
- **double length**
- **float cam_to_table**
- **double cube_adjust [3]**
- **double cube_offset [3]**
- **int origin [3]**
- **double far_corner [3]**
- **double dimension [3]**

B.2.2 Field Documentation

B.2.2.1 float _ctVolume::cam_to_table

Distance between the camera and the fluoroscope table

B.2.2.2 double _ctVolume::cube_adjust[3]

CT cube adjustment buffer

B.2.2.3 double _ctVolume::cube_offset[3]

CT cube offset (x,y,z translations from the origin)

B.2.2.4 double _ctVolume::dimension[3]

CT volume dimensions

B.2.2.5 double _ctVolume::far_corner[3]

Coordinates of far corner of the CT cube

B.2.2.6 double _ctVolume::height

CT volume height in cm

B.2.2.7 double _ctVolume::length

CT volume length in cm

B.2.2.8 int _ctVolume::origin[3]

CT cube origin coordinates(x,y,z), in pixels

B.2.2.9 int _ctVolume::size

number of elements in volume

B.2.2.10 double∗ _ctVolume::volume

3-D CT volume data

B.2.2.11 double _ctVolume::width

CT volume width in cm

The documentation for this struct was generated from the following file:

- **fluoro.h**

B.3 _drrImage Struct Reference

B.3.1 Detailed Description

This struct represents the DRR image.

Data Fields

- int **n**
- **int m**
- **double** $**$ **image**

B.3.2 Field Documentation

B.3.2.1 double$**$ _drrImage::image

2-D image

B.3.2.2 int _drrImage::m

Number of rows

B.3.2.3 int _drrImage::n

Number of columns

The documentation for this struct was generated from the following file:

- **fluoro.h**

B.4 _film Struct Reference

B.4.1 Detailed Description

This struct represents the film in the experimental fluoroscope.

Data Fields

- double **size**
- **double resolution**
- **double dist**
- **double table_to_film**
- **double pixel_size**
- **double offset [3]**
- **double center [3]**
- **double rotation**
- **double azimuth**
- **double elevation**
- **double distance**

B.4.2 Field Documentation

B.4.2.1 double _film::azimuth

Film azimuth

B.4.2.2 double _film::center[3]

B.4.2.3 double _film::dist

Distance

B.4.2.4 double _film::distance

Distance

B.4.2.5 double _film::elevation

Film elevation

B.4.2.6 double _film::offset[3]

Film offset (x,y,z)

B.4.2.7 double _film::pixel_size

Pixel size

B.4.2.8 double _film::resolution

Pixel resolutions

B.4.2.9 double _film::rotation

Film rotation

B.4.2.10 double _film::size

Size of the film

B.4.2.11 double _film::table_to_film

Distance between the film and the fluoroscopic table

The documentation for this struct was generated from the following file:

- **fluoro.h**

B.5 _camera Struct Reference

B.5.1 Detailed Description

This struct represents the X-ray camera in the experimental fluoroscope.

Data Fields

- int **camera_to_table**
- **double azimuth**
- **double elevation**
- **double dist**
- **double offset [3]**
- **double coord [3]**

B.5.2 Field Documentation

B.5.2.1 double _camera::azimuth

Camera azimuth

B.5.2.2 int _camera::camera_to_table

Distance between the camera and the fluoroscopic table

B.5.2.3 double _camera::coord[3]

Camera coordinates

B.5.2.4 double _camera::dist

Camera distance

B.5.2.5 double _camera::elevation

Camera elevation

B.5.2.6 double _camera::offset[3]

Camera offset

The documentation for this struct was generated from the following file:

- **fluoro.h**

B.6 _edges Struct Reference

B.6.1 Detailed Description

This struct represents the filtered DRR image that consists of edges of the image.

Data Fields

- int **n**
- **int m**
- **double** $**$ **image**

B.6.2 Field Documentation

B.6.2.1 double$**$ _edges::image

2-D image

B.6.2.2 int _edges::m

Number of rows

B.6.2.3 int _edges::n

Number of columns

The documentation for this struct was generated from the following file:

- **fluoro.h**

B.7 _dict Struct Reference

B.7.1 Detailed Description

This struct defines a dictionary to hold the key-value pairs of a configuration item present in a config file.

Data Fields

- char **key [MAX_VALUE_LEN]**
- **char value [MAX_VALUE_LEN]**

B.7.2 Field Documentation

B.7.2.1 char _dict::key[MAX_VALUE_LEN]

Configuration item key

B.7.2.2 char _dict::value[MAX_VALUE_LEN]

Configuration item value

The documentation for this struct was generated from the following file:

- **config_reader.h**

B.8 _bee Struct Reference

B.8.1 Detailed Description

This data structure represent a Bee in the Bees Algorithm. Each bee that is involved in the search for the right drr image, is associated with the values for orientation of camera and film, offset of the cube and score of the resulting DRR image, at a given time.

Data Fields

- double **orientation [3]**
- **double offset [3]**
- **double score**
- **int finesearch**
- **int rank**

B.8.2 Field Documentation

B.8.2.1 double _bee::orientation

Represents the orientation of the camera)(az and el) and the film(rt

B.8.2.2 int _bee::finesearch

Indicates whether the bee will next search around the previous values or in random. Value is either 1 or 0

B.8.2.3 double _bee::offset

Represents the cube offset values

B.8.2.4 int _bee::rank

Represents the ranking of the bee based on the score value

B.8.2.5 double _bee::score

Represents the score of the resulting DRR image

The documentation for this struct was generated from the following files:

- **search_parallel_bees.c**
- **search_sequential_bees.c**

B.9 _particle Struct Reference

B.9.1 Detailed Description

This data structure represent a particle in the Particle Swarm Optimization methos. Each particle that is involved in the search for the right drr image, is associated with the values for orientation of camera and film, offset of the cube and score of the resulting DRR image, at a given time.

Data Fields

- double **orientation [3]**
- **double offset [3]**
- **double score**

B.9.2 Field Documentation

B.9.2.1 double _particle::orientation

Represents the orientation of the camera(az and el) and film(rt)

B.9.2.2 double _particle::offset

Represents the cube offset values

B.9.2.3 double _particle::score

Represents the score of the resulting DRR image

The documentation for this struct was generated from the following files:

- **search_parallel_pso.c**

- **search_sequential_pso.c**

B.10 Plugin Interface Documentation

The following sections from Section B.11 to B.13 provide the details of the header files used as interfaces to the plugins to the fluoroscopic project. Implementation of fluoroscopic analysis requires algorithms to be implemented using these interfaces. The algorithm implementation will define all the functions declared in the corresponding header file. This makes it possible for any algorithm that implements this interface to fit into main fluoroscopic analysis program. Provisions have been made to specify which algorithm to use during run time of the main program.

B.11 generate_drr.h File Reference

B.11.1 Detailed Description

This header file provides an interface for implementing an algorithm for generating Digitally Reconstructed Radiograph images from a given CT image.

Typedefs

- typedef void ∗ **CT_Volume_Ptr**
- **typedef void ∗ Drr_Image**

Functions

- void **generate_drr (double camera_azimuth, double camera_elevation, double film_rotation, double cube_offset[ ], CT_Volume_Ptr cube, Drr_Image out_drr_-image)**

B.11.2 Typedef Documentation

B.11.2.1 typedef void ∗ CT_Volume_Ptr

This typedef is used to define a CT volume.

B.11.2.2 typedef void ∗ Drr_Image

This typedef is used to define a DRR image.

B.11.3 Function Documentation

B.11.3.1 void generate_drr (double *azimuth*, double *elevation*, double *rotation*, double *offset*[ ], void ∗ *_cube*, void ∗ *_drr_image*)

This function creates a DRR model.

Parameters:

azimuth Azimuth angle of the camera

elevation Elevation angle of the camera

rotation Rotation angle of the film

offset x,y,z offsets of the CT cube

_cube 3-D CT volume data

_drr_image Pointer to the generated output DRR image [output parameter]

Returns:

Void

B.12 filter.h File Reference

B.12.1 Detailed Description

This header file provides an interface for implementing an algorithm for filtering X-ray images to extract required information from an image.

Typedefs

- typedef void ∗ **Filter_Image**

Functions

- **Filter_Image filter (Filter_Image image)**

B.12.2 Typedef Documentation

B.12.2.1 typedef void ∗ Filter_Image

This typedef is used to define an image.

B.12.3 Function Documentation

B.12.3.1 Filter_Image filter (void ∗ *image*)

Filters the input image and provides an image with only the requires features of the input image.

Parameters:

image Image to be filtered

Returns:

Filtered image

B.13 match.h File Reference

B.13.1 Detailed Description

This header file provides an interface for implementing an algorithm for matching or comparing two images.

Typedefs

- typedef void ∗ **Match_Image**

Functions

- double **match (Match_Image ref_image, Match_Image target_image)**

B.13.2 Typedef Documentation

B.13.2.1 typedef void ∗ Match_Image

This typedef is used to define an image.

B.13.3 Function Documentation

B.13.3.1 double match (void ∗ *ref_image*, void ∗ *target_image*)

Computes the score of an image. This routine is intended to be used after an image has been filtered using a sobel operator for black and white images.

Parameters:

ref_image Reference Contour image

target_image Target Contour image

Returns:

Score value between 0 and 1

B.14 search.h File Reference

B.14.1 Detailed Description

This header file provides an interface for implementing a search algorithm for searching the position and orientation information with a given CT image and configuration information required for the algorithm.

Typedefs

- typedef void ∗ **Search_CT_Volume_Ptr**

Functions

- void **search (Search_CT_Volume_Ptr cube, FILE ∗search_config_filename)**

B.14.2 Typedef Documentation

B.14.2.1 typedef void∗ Search_CT_Volume_Ptr

B.14.3 Function Documentation

B.14.3.1 void search (void ∗ *_cube*, FILE ∗ *config_file*)

This function attempts to search for an fluoroscopy image. The search algorithm is based on a Bees algorithm, where random camera positions are tried until a close match is found.

Parameters:

_cube Pointer to the CT_VOLUME

config_file Pointer to the open file containing all the configuration values required for this algorithm

Returns:

Void

B.15 Source File Documentation

The following sections from Section B.16 to B.31 provide the details of the source files included in this thesis.

Section B.16 contains information about the program that is used to create a DRR image from a given CT image along with position and orientation data. This program is kept separate from the rest of the fluoroscopic analysis code, so that, required DRR images can be created beforehand and used as input to the main fluoroscopy program. Image once created and stored, need not be recreated during each program execution.

Section B.17 contains information about the program which is the main entry point to the entire fluoroscopic analysis process. Sections B.21 to B.31 provide information about the plugins that implement the interfaces defined in Sections B.11 to B.13

B.16 create_target_drr.c File Reference

B.16.1 Detailed Description

This program is used to generate a DRR image from the specified orientation and offset configuration inputs for a given CT image. USAGE: <ProgramName> <PathToGenerate-DrrPlugin> <PathToCTImageFile> <Resolution> <PathToOutputFile> [<Azimuth> <Elevation> <Rotation> [<X_Offset> <Y_Offset> <Z_Offset>]]

Functions

- static void **print_usage (const char ∗appname)**
- **int main (int argc, char ∗argv[ ])**

Variables

- void ∗ **handle**
- **void(∗) generate_drr_function (double azimuth, double elevation, double rotation, double offset[ ], void ∗_cube, void ∗_drr_image)**
- **const char ∗ dlError**

B.16.2 Function Documentation

B.16.2.1 int main (int *argc*, char ∗ *argv*[ ])

This is the main function for creating DRR images from a given CT image. It is responsible for creating a drr image file from the specified orientation and position offset configuration inputs for a given CT image.

Parameters:

> ***argc*** Number of command line parameters
>
> ***argv*** Array of command line parameters

Returns:

> 0 on success, 1 on failure

B.16.2.2 static void print_usage (const char $*$ *appname*) `[static]`

Displays the command line parameters required for this program

Parameters:

> ***appname*** Name of the program

Returns:

> Void

B.16.3 Variable Documentation

B.16.3.1 const char$*$ dlError

B.16.3.2 void($*$) generate_drr_function(double azimuth, double elevation, double rotation, double offset[ ], void $*$_cube, void $*$_drr_image)

B.16.3.3 void$*$ handle

B.17 main.c File Reference

B.17.1 Detailed Description

This is the main entry point for the fluoroscopy program. This program is dependent on shared libraries for filtering and searching DRR images USAGE: <ProgramName> <PathToSearchPlugin> <PathToInputConfigFile>.

Functions

- static void **print_usage_exit (const char ∗appname)**
- **int main (int argc, char ∗argv[ ])**

Variables

- void ∗ **search_handle**
- **void(∗) search_function (void ∗, FILE ∗)**
- **const char ∗ dlError**

B.17.2 Function Documentation

B.17.2.1 int main (int *argc*, char ∗ *argv*[ ])

This is the main function for the fluoroscopy program.

- The input configuration file is parsed to extract the CT image file and the DRR image file.
- CT volume and DRR image are constructed from these two files.
- Following this the edges of the drr image are extracted.

- Using this as a target, a search for the orientation and the offset of the CT cube is begun.
- This function relies on shared libraries for filtering and searching which have to be provided while compiling this program.

Parameters:

argc Number of command line parameters

argv Array of command line parameters

Returns:

0 on success, 1 on failure

B.17.2.2 static void print_usage_exit (const char ∗ *appname*) `[static]`

Displays the command line parameters required for this program

Parameters:

appname Name of the program

Returns:

Void

B.17.3 Variable Documentation

B.17.3.1 const char∗ dlError

B.17.3.2 void(∗) search_function(void ∗, FILE ∗)

B.17.3.3 void∗ search_handle

B.18 write_image.c File Reference

B.18.1 Detailed Description

This file contains routines for writing matrices representing a CT cube and a DRR image to a text file.

Functions

- void **write_image (double **image, int m, int n, const char *filename)**
- **void write_array (double *array, int n, const char *filename)**
- **void write_drr (DRR_IMAGE *drr_image, const char *filename)**

B.18.2 Function Documentation

B.18.2.1 void write_array (double * *array*, int *n*, const char * *filename*)

Writes an array to a text file. Each value is delimited by a space

Parameters:

array Array of doubles

n Number of elements

filename File name to create and write into

Returns:

Void

B.18.2.2 void write_drr (DRR_IMAGE ∗ *drr_image*, const char ∗ *filename*)

Writes a DRR image to a text file. First two lines contain the number of rows and number of columns respectively. The image values are inserted from the third line. Each value is delimited by a space.

Parameters:

drr_image DRR image

filename File name to create and write into

Returns:

Void

B.18.2.3 void write_image (double ∗∗ *image*, int *m*, int *n*, const char ∗ *filename*)

Writes an image array to a text file. Each value is delimited by a space

Parameters:

image Image array (output parameter)

m Number of rows

n Number of columns

filename File name to create and write into

Returns:

Void

B.19 util.c File Reference

B.19.1 Detailed Description

This file contains general application utility functions.

Functions

- static void **set_volume_defaults (CT_VOLUME ∗volume)**
- **FILM ∗ create_film ()**
- **void free_film (FILM ∗film)**
- **CAMERA ∗ create_camera ()**
- **void free_camera (CAMERA ∗cam)**
- **DRR_IMAGE ∗ alloc_drr (int m, int n)**
- **DRR_IMAGE ∗ create_drr (const char ∗filename)**
- **void free_drr (DRR_IMAGE ∗drr)**
- **CT_VOLUME ∗ create_volume (const char ∗filename)**
- **void free_volume (CT_VOLUME ∗volume)**
- **double ∗∗∗ alloc_dcube (int m, int n, int p)**
- **void free_dcube (double ∗∗∗cube)**
- **double ∗∗ alloc_dmat (int m, int n)**
- **void free_dmat (double ∗∗dmat)**
- **void ∗ xmalloc (size_t bytes)**
- **double xrandom ()**
- **double yrandom ()**
- **int zrandom (double limit)**

- **double d_abs (double val)**

B.19.2 Function Documentation

B.19.2.1 double $***$ alloc_dcube (int *m*, int *n*, int *p*)

Allocates memory for a 3 dimensional array of doubles as a contiguous block of memory.

Parameters:

m Number of rows

n Number of columns

p Number of planes

Returns:

A 3-D array of doubles

B.19.2.2 double $**$ alloc_dmat (int *m*, int *n*)

Allocates memory for a 2 dimensional array of doubles as a contiguous block of memory.

ARGS: Number of rows Number of columns

Returns:

Data array(double)

B.19.2.3 DRR_IMAGE $*$ alloc_drr (int *m*, int *n*)

Allocates memory for a DRR image.

Parameters:

m Number of rows

n Number of columns

Returns:

Pointer to a DRR_IMAGE object

B.19.2.4 CAMERA $*$ create_camera ()

Creates a new CAMERA object.

Returns:

: Pointer to a new CAMERA object

B.19.2.5 DRR_IMAGE $*$ create_drr (const char $*$ ***filename***)

Creates an DRR image initialized by the data contained in the specified file.

Parameters:

filename Name of the file containing the DRR_IMAGE data

Returns:

Pointer to a DRR_IMAGE object

B.19.2.6 FILM $*$ create_film ()

Creates a FILM object.

Returns:

Pointer to a new FILM object

B.19.2.7 CT_VOLUME $*$ create_volume (const char $*$ *filename*)

Creates an volume object initialized by the data contained in the specified file.

Parameters:

filename Name of the file containing the CT volume data

Returns:

Pointer to a CT_VOLUME object

B.19.2.8 double d_abs (double *val*)

Returns the absolute value.

Parameters:

val Input value

Returns:

Absolute value

B.19.2.9 void free_camera (CAMERA $*$ *cam*)

Frees up the CAMERA object.

Parameters:

cam Pointer to a CAMERA object

Returns:

Void

B.19.2.10 void free_dcube (double $***$ *cube*)

Frees a dcube object.

Parameters:

cube Pointer to a dcube

Returns:

Void

B.19.2.11 void free_dmat (double $**$ *dmat*)

Frees a dmat object.

Parameters:

dmat Pointer to a dmat object to free

Returns:

Void

B.19.2.12 void free_drr (DRR_IMAGE $*$ *drr*)

Frees memory allocated for a DRR_IMAGE object.

Parameters:

drr Pointer to a DRR_IMAGE

Returns:

Void

B.19.2.13 void free_film (FILM ∗ *film*)

Frees a FILM object.

Parameters:

film Pointer to a FILM object

Returns:

void

B.19.2.14 void free_volume (CT_VOLUME ∗ *volume*)

Frees memory for a CT_VOLUME object.

Parameters:

volume Pointer to a CT_VOLUME object

Returns:

Void

B.19.2.15 static void set_volume_defaults (CT_VOLUME ∗ *volume*) `[static]`

Initializes the default values for the volume object.

Parameters:

volume Pointer to the CT volume

Returns:

void

B.19.2.16 void $*$ xmalloc (size_t *bytes*)

Allocates memory. If memory allocation fails, the application will abort.

Parameters:

bytes Number of bytes to allocate

Returns:

Pointer to a chunk of memory

B.19.2.17 double xrandom ()

Generates a random number (double) between 0 and 1. It is always positive

Returns:

Random number (double)

B.19.2.18 double yrandom ()

Generates a random number (double) between 0 and 1. It can be a positive or a negative number. The positiveness or negativeness is also generated randomly.

Returns:

Random number (double)

B.19.2.19 int zrandom (double *limit*)

Generates a random number (int) that lies within the specified limit. It can be a positive or a negative number. The positiveness or negativeness is also generated randomly.

Returns:

Random number (int)

B.20 config_reader.c File Reference

B.20.1 Detailed Description

This file contains functions to extract the configuration information from a file consisting of key-value pairs in a suitable format. Example for a configuration item: 'MAX_ITER=500', where MAX_ITER is the key and 500 is the value.

Functions

- void **read_config (FILE ∗config)**
- **void print_dict ()**
- **int get_config_value (char ∗key, char ∗∗value)**

B.20.2 Function Documentation

B.20.2.1 void get_config_value (char ∗ *key*, char ∗∗ *value*)

Searches for the given key in the dictionary of configuration items and returns 1 or 0 to indicate the success of the search. It also returns the value through the output parameter.

Parameters:

key String that denotes the key to be searched for in the configuration.

value This will hold the value associated with the key as a result of this operation.

Returns:

1 if key-value pair is found, 0 otherwise.

B.20.2.2 void print_dict ()

Prints the configuration dictionary contents in the form of key-value pairs.

Returns:

void

B.20.2.3 void read_config (FILE $*$ *config*)

Reads the configuration file and constructs a dictionary of key-value pairs for each configuration item.

Parameters:

config FilePointer to the configuration file

Returns:

void

B.21 generate_drr.c File Reference

B.21.1 Detailed Description

This file implements the interface **generate_drr.h** (p. 90). This file contains utility routines for creating Digitally Reconstructed Radiograph (DRR) images. It contains the routine for creating a DRR model which includes setting up the point on a film array and uses ray tracing to create a DRR image for the given orientation and offset values.

Functions

- static void **sph2cart (double theta, double phi, double rho, double cart[ ])**
- **void generate_drr_helper (CAMERA ∗cam, CT_VOLUME ∗cube, double ∗∗∗film_array, int m, int n, int p, DRR_IMAGE ∗drr_image)**
- **void generate_drr (double azimuth, double elevation, double rotation, double offset[ ], void ∗_cube, void ∗_drr_image)**

B.21.2 Function Documentation

B.21.2.1 void generate_drr (double *azimuth*, double *elevation*, double *rotation*, double *offset*[ ], void ∗ *_cube*, void ∗ *_drr_image*)

This function creates a DRR model.

Parameters:

azimuth Azimuth angle of the camera

elevation Elevation angle of the camera

rotation Rotation angle of the film

offset x,y,z offsets of the CT cube

_cube 3-D CT volume data

_drr_image Pointer to the generated output DRR image [output parameter]

Returns:

Void

B.21.2.2 void generate_drr_helper (CAMERA ∗ *cam*, CT_VOLUME ∗ *cube*, double ∗∗∗ *film_array*, int *m*, int *n*, int *p*, DRR_IMAGE ∗ *drr_image*)

This is a helper function that calls the ray tracing routine to create a DRR image from the given information.

Parameters:

cam Pointer to camera orientation

cube Pointer to a CT volume

film_array Pointer to a 3D array containing points on a film

m Number of rows

n Number of columns

p Depth of the array

drr_image Pointer to the DRR image [output parameter]

Returns:

Void

B.21.2.3 static void sph2cart (double *theta*, double *phi*, double *rho*, double *cart*[ ]) `[static]`

This function converts spherical coordinates to Cartesian coordinates.

Parameters:

theta Value of theta

phi Value of phi

rho Value of rho

cart Array for 3-D Cartesian coordinates [output parameter]

Returns:

Void

B.22 ray_tracing.c File Reference

B.22.1 Detailed Description

This file implements Siddon's ray tracing algorithm". Ray tracing involves constructing a DRR from a CT image by calculating the radiological path through a three-dimensional CT array.

Functions

- void **trace_CT (double ∗cam_pt, double ∗film_array, int ∗film_dims, double ∗CT_image, double ∗N_vox, double ∗CT_size, int ∗CT_orig, double ∗DRR_out_-im)**

Variables

- const int ∗ **film_dims**

B.22.2 Function Documentation

B.22.2.1 void trace_CT (double ∗ *cam_pt*, double ∗ *film_array*, int ∗ *film_dims*, double ∗ *CT_image*, double ∗ *N_vox*, double ∗ *CT_size*, int ∗ *CT_orig*, double ∗ *DRR_out_im*)

B.22.3 Variable Documentation

B.22.3.1 const int∗ film_dims

B.23 filter_sobel_bw.c File Reference

B.23.1 Detailed Description

This file implements the interface "filter.h". It is the plugin for filtering using Sobel algorithm for black and white values.

Functions

- static int ∗∗ **conv2d (double ∗∗image, int num_rows, int num_cols, const int filter_kernal[ ][3], int fk_row, int fk_col)**
- **void ∗ filter (void ∗image)**

Variables

- static const int **x_filt [ ][3]**
- **static const int y_filt [ ][3]**
- **static const int d1_filt [ ][3]**
- **static const int d2_filt [ ][3]**

B.23.2 Function Documentation

B.23.2.1 static int ∗∗ conv2d (double ∗∗ *image*, int *num_rows*, int *num_cols*, const int *filter_kernal*[ ][3], int *fk_row*, int *fk_col*) [static]

Function for two dimensional convolution: g(x,y) = SUM{ SUM{ w(s,t)f(x+s,y+t), t=-b..b}, s=-a..a}

Parameters:

image 2-D image array

num_rows Number of rows in image array

num_cols Number of cols in image array

filter_kernal 2-D filter kernal

fk_row Number of rows in filter kernal

fk_col Number of cols in filter kernal

Returns:

Weighted average value

B.23.2.2 Filter_Image filter (void ∗ *image*)

Filters the input image and provides an image with only the requires features of the input image.

Parameters:

image Image to be filtered

Returns:

Filtered image

B.23.3 Variable Documentation

B.23.3.1 const int d1_filt[ ][3] `[static]`

B.24 filter_sobel_gs.c File Reference

B.24.1 Detailed Description

This file implements the interface ”filter.h”. It is the plugin for filtering using Sobel algorithm for grayscale values.

Functions

- static int ∗∗ **conv2d (double ∗∗image, int num_rows, int num_cols, const int filter_kernal[ ][3], int fk_row, int fk_col)**
- **void ∗ filter (void ∗image)**

Variables

- static const int **x_filt [ ][3]**
- **static const int y_filt [ ][3]**
- **static const int d1_filt [ ][3]**
- **static const int d2_filt [ ][3]**

B.24.2 Function Documentation

B.24.2.1 static int ∗∗ conv2d (double ∗∗ *image*, int *num_rows*, int *num_cols*, const int *filter_kernal*[ ][3], int *fk_row*, int *fk_col*) `[static]`

Function for two dimensional convolution: g(x,y) = SUM{ SUM{ w(s,t)f(x+s,y+t), t=-b..b}, s=-a..a}

Parameters:

image 2-D image array

num_rows Number of rows in image array

num_cols Number of cols in image array

filter_kernal 2-D filter kernal

fk_row Number of rows in filter kernal

fk_col Number of cols in filter kernal

Returns:

Weighted average value

B.24.2.2 void∗ filter (void ∗ *image*)

Filters the input image and provides an image with only the requires features of the input image.

Parameters:

image Image to be filtered

Returns:

Filtered image

B.24.3 Variable Documentation

B.24.3.1 const int d1_filt[ ][3] `[static]`

B.25 match_contour_bw.c File Reference

B.25.1 Detailed Description

Provides contour matching for black and white images. This file implements the interface ”match.h”. It uses contour matching algorithm for comparing two black and white contour images.

Functions

- double **match (void ∗image, void ∗target_image)**

B.25.2 Function Documentation

B.25.2.1 double match (void ∗ *image*, void ∗ *target_image*)

Computes the score of an image. This routine is intended to be used after an image has been filtered using a sobel operator for black and white images.

Parameters:

image Reference Contour image

target_image Target Contour image

Returns:

Score value between 0 and 1

B.26 match_contour_gs.c File Reference

B.26.1 Detailed Description

Provides contour matching for grayscale images. This file implements the interface "match.h". It uses contour matching algorithm for comparing two grayscale contour images.

Functions

- double **match (void ∗image, void ∗target_image)**

B.26.2 Function Documentation

B.26.2.1 double match (void ∗ *image*, void ∗ *target_image*)

Computes the score of an image. This routine is intended to be used after an image has been filtered using a sobel operator for black and white images.

Parameters:

image Reference Contour image

target_image Target Contour image

Returns:

Score value between 0 and 1

B.27 search_sequential_montecarlo.c File Reference

B.27.1 Detailed Description

This file implements the interface 'search.h using Monte Carlo search technique. This is a sequential program that contains routines for attempting to find a fluoroscopy image frame by deriving a matching DRR image from a CT volume. Monte Carlo method is used to achieve this. This program depends on libraries for filtering and matching a DRR image.

Functions

- static double **get_score (double cam[ ], double offset[ ], double step[ ], EDGES *t_edges, CT_VOLUME *cube)**
- **void search (void *_cube, FILE *config_file)**

Variables

- void * **generate_drr_handle**
- **void * filter_handle**
- **void * match_handle**
- **void(*) generate_drr_function (double, double, double, double[ ], void *, void *)**
- **void *(*) filter_function (void *)**
- **double(*) match_function (void *, void *)**
- **const char * dlError**
- **double offset [3] = {0,0,0}**

B.27.2 Function Documentation

B.27.2.1 static double get_score (double *cam*[ ], double *offset*[ ], double *step*[ ], EDGES ∗ *t_edges*, CT_VOLUME ∗ *cube*) `[static]`

Creates a DRR image for the given orientation and offset and returns its score as compared to the target DRR image.

Parameters:

cam Camera coordinates for the required DRR image

offset Cube offsets for the required DRR image

step Camera offset step values for the required DRR image

t_edges Pointer to the edges of the target DRR image

cube Pointer to the CT volume data

Returns:

Image score

B.27.2.2 void search (void ∗ *_cube*, FILE ∗ *config_file*)

This function attempts to search for an fluoroscopy image. The search algorithm is based on a Bees algorithm, where random camera positions are tried until a close match is found.

Parameters:

_cube Pointer to the CT_VOLUME

config_file Pointer to the open file containing all the configuration values required for this algorithm

Returns:

Void

B.27.3 Variable Documentation

B.27.3.1 const char∗ dlError

B.27.3.2 void∗(∗) filter_function(void ∗)

B.27.3.3 void ∗ filter_handle

B.27.3.4 void(∗) generate_drr_function(double, double, double, double[ ], void ∗, void ∗)

B.27.3.5 void∗ generate_drr_handle

B.27.3.6 double(∗) match_function(void ∗, void ∗)

B.27.3.7 void ∗ match_handle

B.27.3.8 double offset[3] = {0,0,0}

B.28 search_sequential_bees.c File Reference

B.28.1 Detailed Description

This file implements the interface 'search.h using Bees Algorithm. This is a sequential program that contains routines for attempting to find a fluoroscopy image frame by deriving a matching DRR image from a CT volume. Bees algorithm is used to achieve this. This program depends on libraries for filtering and matching a DRR image.

Data Structures

- struct **_bee**

Typedefs

- typedef **_bee bee**
- **typedef _bee * BeePtr**

Functions

- static **BeePtr create_bee ()**
- **static void copy_bee (BeePtr from_bee, BeePtr to_bee)**
- **static void free_bee (BeePtr the_bee)**
- **static double get_score (BeePtr the_bee, EDGES *t_edges, CT_VOLUME *cube)**
- **static void print_info (long iteration, int bee_id, BeePtr the_bee, char *msg)**
- **void search (void *_cube, FILE *config_file)**

Variables

- void ∗ **generate_drr_handle**
- **void ∗ filter_handle**
- **void ∗ match_handle**
- **void(∗) generate_drr_function (double, double, double, double[ ], void ∗, void ∗)**
- **void ∗(∗) filter_function (void ∗)**
- **double(∗) match_function (void ∗, void ∗)**
- **const char ∗ dlError**

B.28.2 Typedef Documentation

B.28.2.1 typedef struct _bee bee

B.28.2.2 typedef struct _bee∗ BeePtr

B.28.3 Function Documentation

B.28.3.1 static void copy_bee (BeePtr *from_bee*, BeePtr *to_bee*) `[static]`

Copies the values of one bee to another bee.

Parameters:

from_bee Pointer to the bee whose values are to be copied

to_bee Pointer to the bee whose values should be copied from the from_bee

/returns Void

B.28.3.2 static BeePtr create_bee () `[static]`

Creates a new bee and allocates memory to it. It also initializes all its members to 0. /returns Pointer to a bee

B.28.3.3 static void free_bee (BeePtr *the_bee*) `[static]`

Deallocate the memory allocated to the bee.

Parameters:

the_bee Pointer to the bee

Returns:

Void

B.28.3.4 static double get_score (BeePtr *the_bee*, EDGES ∗ *t_edges*, CT_VOLUME ∗ *cube*) `[static]`

Creates a DRR image for the given orientation and offset and returns its score as compared to the target DRR image.

Parameters:

the_bee Pointer to the bee which contains the values of camera and film orientation, cube position offsets and score

t_edges Pointer to the edges of the target DRR image

cube Pointer to the CT volume data

Returns:

Image score

B.28.3.5 void print_info (long *iteration*, int *bee_id*, BeePtr *the_bee*, char ∗ *msg*) `[static]`

This function is used to print debug information. It is not a general purpose logger, but very specific to printing orientation and offset values for a specific bee at a specific iteration.

Parameters:

iteration Iteration of the search

bee_id Id of the bee whose info is to be printed

msg String that displays the message associated with the information

B.28.3.6 void search (void ∗ *_cube*, FILE ∗ *config_file*)

This function attempts to search for an fluoroscopy image. The search algorithm is based on a Bees algorithm, where random camera and film orientation and CT cube positions are tried until a close match is found.

Parameters:

_cube Pointer to the CT_VOLUME

config_file Pointer to the open file containing all the configuration values required for this algorithm

Returns:

Void

B.28.4 Variable Documentation

B.28.4.1 const char∗ dlError

B.28.4.2 void∗(∗) filter_function(void ∗)

B.28.4.3 void ∗ filter_handle

B.28.4.4 void(∗) generate_drr_function(double, double, double, double[ ], void ∗, void ∗)

B.28.4.5 void∗ generate_drr_handle

B.28.4.6 double(∗) match_function(void ∗, void ∗)

B.28.4.7 void ∗ match_handle

B.29 search_sequential_pso.c File Reference

B.29.1 Detailed Description

This file implements the interface 'search.h using Particle Swarm Optimization method. This is a sequential program that contains routines for attempting to find a fluoroscopy image frame by deriving a matching DRR image from a CT volume. Particle Swarm Optimization algorithm is used to achieve this. This program depends on libraries for filtering and matching a DRR image.

Data Structures

- struct **_particle**

Typedefs

- typedef **_particle particle**
- **typedef _particle * ParticlePtr**

Functions

- static **ParticlePtr create_particle ()**
- **static void copy_particle (ParticlePtr from_particle, ParticlePtr to_particle)**
- **static void free_particle (ParticlePtr the_particle)**
- **static double get_score (ParticlePtr the_particle, EDGES *t_edges, CT_-VOLUME *cube)**
- **static void print_info (long iteration, int neighborhood_id, int particle_id, ParticlePtr the_particle, char *msg)**
- **void search (void *_cube, FILE *config_file)**

Variables

- void ∗ **generate_drr_handle**
- **void ∗ filter_handle**
- **void ∗ match_handle**
- **void(∗) generate_drr_function (double, double, double, double[ ], void ∗, void ∗)**
- **void ∗(∗) filter_function (void ∗)**
- **double(∗) match_function (void ∗, void ∗)**
- **const char ∗ dlError**

B.29.2 Typedef Documentation

B.29.2.1 typedef struct _particle particle

B.29.2.2 typedef struct _particle∗ ParticlePtr

B.29.3 Function Documentation

B.29.3.1 static void copy_particle (ParticlePtr *from_particle*, ParticlePtr *to_particle*) `[static]`

Copies the values of one particle to another particle.

Parameters:

from_particle Pointer to the particle whose values are to be copied

to_particle Pointer to the particle whose values should be copied from the from_-particle

/returns Void

B.29.3.2 static ParticlePtr create_particle () `[static]`

Creates a new particle and allocates memory to it. It also initializes all its members to 0. /returns Pointer to a particle

B.29.3.3 static void free_particle (ParticlePtr *the_particle*) `[static]`

Deallocate the memory allocated to the particle.

Parameters:

the_particle Pointer to the particle

Returns:

Void

B.29.3.4 static double get_score (ParticlePtr *the_particle*, EDGES ∗ *t_edges*, CT_VOLUME ∗ *cube*) `[static]`

Creates a DRR image for the given orientation and offset and returns its score as compared to the target DRR image.

Parameters:

the_particle Pointer to the particle which contains the values of camera and film orientation,cube position offsets and score

t_edges Pointer to the edges of the target DRR image

cube Pointer to the CT volume data

Returns:

Image score

B.29.3.5 void print_info (long *iteration*, int *neighborhood_id*, int *particle_id*, ParticlePtr *the_particle*, char ∗ *msg*) `[static]`

This function is used to print debug information. It is not a general purpose logger, but very specific to printing orientation and offset values for a specific bee at a specific iteration.

Parameters:

iteration Iteration of the search

neighborhood_id ID of the particle's neighborhood

particle_id ID of the particle whose info is to be printed

the_particle Pointer to the particle

msg String that displays the message associated with the information

B.29.3.6 void search (void ∗ *_cube*, FILE ∗ *config_file*)

This function attempts to search for an fluoroscopy image. The search algorithm is based on a Bees algorithm, where random camera and film orientation and CT cube positions are tried until a close match is found.

Parameters:

_cube Pointer to the CT_VOLUME

config_file Pointer to the open file containing all the configuration values required for this algorithm

Returns:

Void

B.29.4 Variable Documentation

B.29.4.1 const char∗ dlError

B.29.4.2 void∗(∗) filter_function(void ∗)

B.29.4.3 void ∗ filter_handle

B.29.4.4 void(∗) generate_drr_function(double, double, double, double[ ], void ∗, void ∗)

B.29.4.5 void∗ generate_drr_handle

B.29.4.6 double(∗) match_function(void ∗, void ∗)

B.29.4.7 void ∗ match_handle

B.30 search_parallel_bees.c File Reference

B.30.1 Detailed Description

This file implements the interface 'search.h using Bees Algorithm. This is parallel program that contains routines for attempting to find a fluoroscopy image frame by deriving a matching DRR image from a CT volume. Bees algorithm is used to achieve this. This program depends on libraries for filtering and matching a DRR image.

Data Structures

- struct **_bee**

Typedefs

- typedef **_bee bee**
- **typedef _bee * BeePtr**

Functions

- static **BeePtr create_bee ()**
- **static void copy_bee (BeePtr from_bee, BeePtr to_bee)**
- **static void free_bee (BeePtr the_bee)**
- **static double get_score (BeePtr the_bee, EDGES *t_edges, CT_VOLUME *cube)**
- **static void print_info (long iteration, int bee_id, BeePtr the_bee, char *msg)**
- **void search (void *_cube, FILE *config_file)**

Variables

- void ∗ **generate_drr_handle**
- **void ∗ filter_handle**
- **void ∗ match_handle**
- **void(∗) generate_drr_function (double, double, double, double[ ], void ∗, void ∗)**
- **void ∗(∗) filter_function (void ∗)**
- **double(∗) match_function (void ∗, void ∗)**
- **const char ∗ dlError**

B.30.2 Typedef Documentation

B.30.2.1 typedef struct _bee bee

B.30.2.2 typedef struct _bee∗ BeePtr

B.30.3 Function Documentation

B.30.3.1 static void copy_bee (BeePtr *from_bee*, BeePtr *to_bee*) `[static]`

Copies the values of one bee to another bee.

Parameters:

from_bee Pointer to the bee whose values are to be copied

to_bee Pointer to the bee whose values should be copied from the from_bee

/returns Void

B.30.3.2 static BeePtr create_bee () `[static]`

Creates a new bee and allocates memory to it. It also initializes all its members to 0. /returns Pointer to a bee

B.30.3.3 static void free_bee (BeePtr *the_bee*) `[static]`

Deallocate the memory allocated to the bee.

Parameters:

the_bee Pointer to the bee

Returns:

Void

B.30.3.4 static double get_score (BeePtr *the_bee*, EDGES $*$ *t_edges*, CT_VOLUME $*$ *cube*) `[static]`

Creates a DRR image for the given orientation and offset and returns its score as compared to the target DRR image.

Parameters:

the_bee Pointer to the bee which contains the values of camera and film orientation,cube position offsets and score

t_edges Pointer to the edges of the target DRR image

cube Pointer to the CT volume data

Returns:

Image score

B.30.3.5 void print_info (long *iteration*, int *bee_id*, BeePtr *the_bee*, char $*$ *msg*)

`[static]`

This function is used to print debug information. It is not a general purpose logger, but very specific to printing orientation and offset values for a specific bee at a specific iteration.

Parameters:

iteration Iteration of the search

bee_id Id of the bee whose info is to be printed

msg String that displays the message associated with the information

B.30.3.6 void search (void $*$ *_cube*, FILE $*$ *config_file*)

This function attempts to search for an fluoroscopy image. The search algorithm is based on a Bees algorithm, where random camera and film orientation and CT cube positions are tried until a close match is found.

Parameters:

_cube Pointer to the CT_VOLUME

config_file Pointer to the open file containing all the configuration values required for this algorithm

Returns:

Void

B.30.4 Variable Documentation

B.30.4.1 const char∗ dlError

B.30.4.2 void∗(∗) filter_function(void ∗)

B.30.4.3 void ∗ filter_handle

B.30.4.4 void(∗) generate_drr_function(double, double, double, double[ ], void ∗, void ∗)

B.30.4.5 void∗ generate_drr_handle

B.30.4.6 double(∗) match_function(void ∗, void ∗)

B.30.4.7 void ∗ match_handle

B.31 search_parallel_pso.c File Reference

B.31.1 Detailed Description

This file implements the interface 'search.h using Particle Swarm Optimization method. This is a parallel program that contains routines for attempting to find a fluoroscopy image frame by deriving a matching DRR image from a CT volume. Particle Swarm Optimization algorithm is used to achieve this. This program depends on libraries for filtering and matching a DRR image.

Data Structures

- struct **_particle**

Typedefs

- typedef **_particle particle**
- **typedef _particle * ParticlePtr**

Functions

- static **ParticlePtr create_particle ()**
- **static void copy_particle (ParticlePtr from_particle, ParticlePtr to_particle)**
- **static void free_particle (ParticlePtr the_particle)**
- **static double get_score (ParticlePtr the_particle, EDGES *t_edges, CT_-VOLUME *cube)**
- **static void print_info (long iteration, int neighborhood_id, int particle_id, ParticlePtr the_particle, char *msg)**
- **void search (void *_cube, FILE *config_file)**

Variables

- void ∗ **generate_drr_handle**
- **void ∗ filter_handle**
- **void ∗ match_handle**
- **void(∗) generate_drr_function (double, double, double, double[ ], void ∗, void ∗)**
- **void ∗(∗) filter_function (void ∗)**
- **double(∗) match_function (void ∗, void ∗)**
- **const char ∗ dlError**

B.31.2 Typedef Documentation

B.31.2.1 typedef struct _particle particle

B.31.2.2 typedef struct _particle∗ ParticlePtr

B.31.3 Function Documentation

B.31.3.1 static void copy_particle (ParticlePtr *from_particle*, ParticlePtr *to_particle*) `[static]`

Copies the values of one particle to another particle.

Parameters:

from_particle Pointer to the particle whose values are to be copied

to_particle Pointer to the particle whose values should be copied from the from_-particle

/returns Void

B.31.3.2 static ParticlePtr create_particle () `[static]`

Creates a new particle and allocates memory to it. It also initializes all its members to 0. /returns Pointer to a particle

B.31.3.3 static void free_particle (ParticlePtr *the_particle*) `[static]`

Deallocate the memory allocated to the particle.

Parameters:

the_particle Pointer to the particle

Returns:

Void

B.31.3.4 static double get_score (ParticlePtr *the_particle*, EDGES $*$ *t_edges*, CT_VOLUME $*$ *cube*) `[static]`

Creates a DRR image for the given orientation and offset and returns its score as compared to the target DRR image.

Parameters:

the_particle Pointer to the particle which contains the values of camera and film orientation, cube position offsets and score

t_edges Pointer to the edges of the target DRR image

cube Pointer to the CT volume data

Returns:

Image score

B.31.3.5 void print_info (long *iteration*, int *neighborhood_id*, int *particle_id*, ParticlePtr *the_particle*, char ∗ *msg*) [static]

This function is used to print debug information. It is not a general purpose logger, but very specific to printing camera and film orientation and CT cube offset values for a specific bee at a specific iteration.

Parameters:

iteration Iteration of the search

neighborhood_id ID of the particle's neighborhood

particle_id ID of the particle whose info is to be printed

the_particle Pointer to the particle

msg String that displays the message associated with the information

B.31.3.6 void search (void ∗ *_cube*, FILE ∗ *config_file*)

This function attempts to search for an fluoroscopy image. The search algorithm is based on a Bees algorithm, where random camera and film orientation and CT cube positions are tried until a close match is found.

Parameters:

_cube Pointer to the CT_VOLUME

config_file Pointer to the open file containing all the configuration values required for this algorithm

Returns:

Void

B.31.4 Variable Documentation

B.31.4.1 const char$*$ dlError

B.31.4.2 void$*$($*$) filter_function(void $*$)

B.31.4.3 void $*$ filter_handle

B.31.4.4 void($*$) generate_drr_function(double, double, double, double[ ], void $*$, void $*$)

B.31.4.5 void$*$ generate_drr_handle

B.31.4.6 double($*$) match_function(void $*$, void $*$)

B.31.4.7 void $*$ match_handle