ENTROPIC GRAPHS FOR IMAGE REGISTRATION

by

Huzefa Firoz Neemuchwala

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Biomedical Engineering)
in The University of Michigan
2005

Doctoral Committee:

Professor Alfred O. Hero III, Co-chair
Professor Paul L. Carson, Co-chair
Professor Charles R. Meyer
Professor Jeffrey A. Fessler

ABSTRACT

ENTROPIC GRAPHS FOR IMAGE REGISTRATION

by
Huzefa Firoz Neemuchwala

Co-chairs: Alfred O. Hero III and Paul Carson

Given 2D or 3D images gathered via multiple sensors located at different positions, the multi-sensor image registration problem is to align the images so that they have an identical pose in a common coordinate system. Image registration methods depend crucially upon a robust image similarity measure to guide the image alignment. This thesis concerns itself with a new class of such similarity measures. The launching point of this thesis is the entropic graph based estimate of Rényi's α-entropy developed by Ma for image registration. This thesis extends this initial work to develop other entropic graph-based divergence measures to be used with advanced higher dimensional features. A detailed analysis of entropic graphs is followed by a demonstration of their performance advantages relative to conventional similarity measures. This thesis introduces techniques to extend image registration to higher dimension feature spaces using Rényi's generalized α-entropy. The α-entropy is estimated directly through continuous quasi-additive power-weighted graphs such as the minimal spanning tree (MST) and k-Nearest Neighbor graph (kNN). Entropic graph methods are further used to approximate similarity measures like the α-mutual informa-

tion, non-linear correlation coefficient, α-Jensen divergence, Henze-Penrose affinity and Geometric-Arithmetic mean affinity. Entropic-graph similarity measures are applied to problems in breast Ultrasound image registration for cancer management, geo-stationary satellite registration, feature clustering and classification and for atlas based multi-image registration. This last work is a novel and significant application of divergence estimation for registering several images simultaneously. These similarity measures offer robust registration benefits in a multisensor environment. Higher dimensional features used for this work include basis functions like multidimensional wavelets, independent component analysis (ICA) and discrete cosine transforms.

to my daughter Zainab
who puts a smile on my face every morning.

ACKNOWLEDGEMENTS

I have had two advisors, two mentors really, each with his own style, each with his own philosophy and each with his own unique contribution toward this thesis, toward my intellectual advancement and toward my outlook to life. While it is not my intention to compare their contributions, I would certainly like to convey my gratitude to them.

I would like to thank Professor Alfred Hero for giving me this opportunity, for his trust in me and his encouragement, for teaching me the value of time, the value of abstract thinking and a lot of other concepts in electrical engineering!

I would like to thank my co-advisor Professor Paul Carson, who had a major role in my coming to UM and my continued education here, even as I struggled. He has had me thinking on challenging practical issues in engineering, patiently compelling me to discuss relevancy to medical imaging even as I sometimes drifted into Utopian pursuits!

I would like to thank Professor Charles 'Chuck' Meyer for supporting me through this thesis, for encouraging my methods, for discussions on higher-order mutual information and for his generosity in extending me more than my share of lab resources.

I would like to thank Professor Jeffrey Fessler for his contribution in my work. Never have I walked out of his office without a new idea or a new approach toward my problems.

I would like to thank my wife Nadia (Nafisa) for her support in this long long period, sometimes enduring separation for our sake but never losing faith, never losing hope, always inspiring always smiling. And how can I forget my daughter, Zainab, a blessing straight from the Heavens. She has made the last four months seem so short!

I would like to thank my parents, Firoz and Saeeda Neemuchwala for their support and confidence in me as I pursued this thesis, for their contribution toward what I am and what I stand-for today. I would like to thank my brother Zoher and his family including my sister-in-law Shirin and their kids Sakina and Alifiya and my younger brother Moiz for their love and affection and constant support as I spent these years studying.

I would like to thank Sharon Karaghan, my office-mate with whom I could always have a discussion on any topic from Indian food to my job-hunt! I would like to thank Jochen Krucker and Jerry LeCarpentier, for their help as I started out in the lab, learning the ropes. Both of them are now close friends. I would like to thank Sakina who has been an exciting colleague to work with and has always been eager to participate in the research. I would like to thank my friends, colleagues, lab-mates (in no specific order): Desmond, Titaina, Bing, HyunJin, Ram, Roshni, Derek, Jose, Brian, Oliver, Rich, Aaron, Thyag, Narayan (PSLN) and many others. I would like to thank Tarry Goble, Carol Cribbins and Maria Steele for helping me navigate through all the red-tape.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF ALGORITHMS

Algorithm

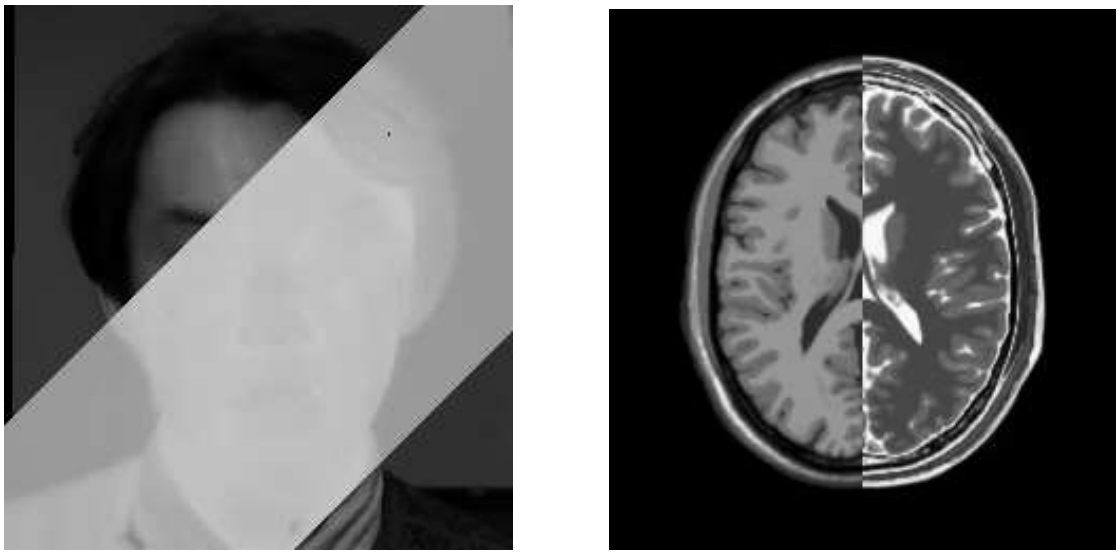
CHAPTER I

Introduction

1.1 Image Registration

Given 2D or 3D images gathered via multiple sensors located at different positions, the multi-sensor image registration problem is to align the images so that they have an identical pose in a common coordinate system. In medical imaging, cross sectional anatomic images are routinely acquired by magnetic induction (Magnetic Resonance Imaging, MRI), absorption of accelerated energized photons (X-Ray Computed Tomography, CT), ultra high frequency sound ultrasound (UL) waves and radionuclide imaging (emission computer tomography, (ECT), positron emission tomography (PET), single photon emission computed tomography (SPECT)). Each sensor or modality provides unique information about the object that is then combined with information from other sensors to arrive at a decision. For example before a surgical procedure in the brain, doctors gather soft-tissue information via MRI sensors and skull or hard-tissue information is acquired via CT. These modalities are then registered to identify locations of tumor or disease in the brain and perform surgical planning. Image fusion is defined as the task of extracting co-occurring information from multisensor images. Image fusion finds several applications in medical imaging where it is used to fuse anatomic and metabolic information [109, 87, 34], and build global anatomical atlases [120], among other applications. Figure 1.1.1 shows

examples of multimodal face matching and brain image registration.



(a) (b)

Figure 1.1.1: Image fusion: (a) Co-registered images of the face acquired via visible light and longwave senors. (b) Registered brain images acquired by time-weighted responses . Face and brain images courtesy ([33]) and ([25]) respectively.

Image registration is a challenging multi-sensor fusion problem due to the diversity of sensors capable of imaging objects and their intrinsic properties. Artifacts such as motion, occlusion, specular refraction, noise, inhomogeneities in the object and imperfections in the transducer compound the difficulty of image registration. Cost and other physical considerations can constrain the spatial or spectral resolution and the signal to noise ratio (SNR). Despite these hindrances, image registration is now commonplace in medical imaging, satellite imaging and stereo vision. Image registration also finds widespread usage in other pattern recognition and computer vision applications such as image segmentation, tracking and motion compensation. A comprehensive survey of the image registration problem, its applications, and implementable algorithms can be found in [86, 85]. In this thesis, we present extended entropic graph registration methods using accelerated

minimum spanning tree (MST) and k-nearest neighbor graph (kNNG) algorithms. Higher dimensional features extracted from images using independent component analysis and wavelets are used to represent images. This chapter describes past attempts to address the multimodal image registration problem. It also underlines the major contributions of this thesis and presents an outline of the work herein.

1.2 Previous approaches to image registration and their limitations

Early image registration methods were developed mostly often for images acquired via the same modality. Thus the pixel intensity maps of the reference and target image resembled each other closely except for intensity preserving deformation. Early researchers in image registration methods thus used measures like linear correlation to measure the similarity between images. Correlation was measured between the pixel intensity maps of the two images. Since images were captured using the same sensor or sensors with close physical characteristics, a linear relationship could be expected between pixel intensity maps of the images. Today, these early correlation methods find limited application in multisensor registration applications. Images acquired with multiple sensors often exhibit non-linear relationships amongst each other. In medical imaging, this is commonly seen in registration of images involving two different imaging modalities.

To overcome limitations of linear correlation, Viola and Wells [125] and Maes et. al. [84] devised a similarity measure based on an information divergence, specifically the Kullback-Liebler [74] divergence between the joint feature density and the product of the marginal densities. This is the mutual information (MI) measure and it quantifies the non-linear correlation between images as the amount of statistical dependency in the underlying joint probability distribution functions (pdf). They defined an estimate of MI measured as the divergence between the joint histograms of pixels and product of the

marginal histograms of pixel intensities. The histogram density estimator is substituted in the MI formula in place of the actual pdf's of the pixel features. Scalar pixel intensities enable histogram building in 1D to estimate marginal densities and in 2D to estimate joint density of pixel features. With the ability to capture non-linear relationships between pixels, registration methods received a boost. However, although the pixel-histogram method overcomes the nonlinear correlation problem, drawbacks abound due to the use of histogram density estimators. Histograms are efficient density estimators in low dimensions, but cannot be reliably constructed in higher dimensional feature spaces (> 4) thus limiting themselves to applications where dimensionality of feature space is very low. Several applications in multisensor fusion require the higher dimensional feature descriptors to effectively capture signal properties. Unfortunately, the pixel-histogram method cannot be directly extended to address these problems. While correlation can be measured in higher dimensional feature spaces, it cannot account for the nonlinear relationships between features.

Recently, Ma and Hero [82] proposed the use of entropic-graph methods for image registration. As contrasted to the previous approaches, entropic graphs estimate an information divergence without the need to compute histogram density estimates. The launching point of this thesis is the entropic graph based estimate of Rényi's α-entropy introduced by [56, 55, 54] and developed by Ma for image registration [82]. An entropic graph is any graph whose normalized total weight (sum of the edge lengths) is a consistent estimator of α-entropy. An example of an entropic graph is the minimal spanning tree and due to its low computational complexity it is an attractive entropic graph algorithm. This graph estimator can be viewed as a multidimensional generalization of the Vasicek-Shannon entropy estimator for one dimensional features [124, 12]. Graph methods such as the MST sidestep the issue of density estimation and have asymptotic convergence to the Rényi α-entropy

of the feature distribution. The α-entropy of a multivariate distribution is a generalization of the better known Shannon entropy. Alfred Rényi introduced the α-entropy in a 1961 paper [108] and since then many important properties of α-entropy have been established [9]. From Rényi's α-entropy the Rényi α-divergence and the Rényi α-mutual information (α-MI) can be defined in a straightforward manner. For $\alpha = 1$ these quantities reduce to the standard (Shannon) entropy, (Kullback-Liebler) divergence, and (Shannon) MI, respectively. Another useful quantity that can be derived from the α-entropy is the α-Jensen difference, which is a generalization of the standard (Shannon) Jensen difference. Ma [82] demonstrated that this generalization allows for an image matching algorithm that benefits from a simple estimation procedure and an extra degree of freedom (α).

Various forms of α-entropy have been exploited by others for applications including: reconstruction and registration of interferometric synthetic aperture radar (ISAR) images [49, 39]; blind deconvolution [35]; and time-frequency analysis [8, 128]. The innovation of Ma's work [82] and this thesis with respect to these other methods is the extension to high dimensional features via entropic graph estimation methods. On the other hand, the α-entropy approaches described in this thesis should not be confused with entropy-alpha classification in SAR processing [24]. A tutorial introduction to the use of entropic graphs to estimate multivariate α-entropy and other entropy quantities was published by in a recent survey article [55]. Generalized measures of dissimilarity were estimated from the features using MST and kNN graphs.

This discussion leads us to the next section where the chief contributions of this thesis are described with an emphasis on its attempts to overcome the deficiencies in previous methods as well as extensions.

1.3 Contributions of Thesis

Several new applications of entropic graphs in high dimensional feature spaces are presented in this thesis. These entropic graph estimates can be computed via a host of combinatorial optimization methods including the MST and the k-Nearest neighbor graph (kNNG). The computation and storage complexity of the MST and kNNG-based estimates increase linearly in feature dimension as opposed to the exponential rates of histogram-based estimates of entropy. Furthermore, as will be shown, entropic graphs can also be used to estimate more general similarity measures. Specific examples include the α-mutual information (α-MI), α-Jensen difference divergence, the Henze-Penrose (HP) affinity, which is a multidimensional approximation to the Wald-Wolfowitz test [127], the α-geometric-arithmetic (α-GA) mean divergence [118] and a new measure of nonlinear correlation called the nonlinear correlation coefficient, presented here for the first time. To our knowledge, the latter divergence measures have never been utilized in the context of image registration problems. We also explore variants of entropic graph methods that which exhibit faster convergence and reduced computational complexity.

A primary motivating problem for this thesis is the registration of UL images. Compared with other modalities, UL registration has not been studied extensively. This thesis thus extends the application of entropic graphs to new applications like Ultrasound breast image registration, multisensor satellite image registration, registration involving several images (simultaneous multi-image registration), MRI small volume registration and matching of human face images. Ultrasound imaging is a cheaply available and widely used modality to detect malignant breast lesions. However, compressibility of the breast tissue, high specular imaging noise, low resolution and small field-of-view have complicated past registration efforts. Through the use of graph-based registration methods and a

data-driven features extraction process, lower registration errors in test cases were seen.

Besides mono-modality UL registration, graph methods have been applied to various multimodal images. Aerial images of the earth taken by geo-stationary satellites in the thermal and visible bands of the electromagnetic spectrum are registered using graph methods and higher-dimensional features. In registering multiple images to form a representative atlas, high dimensional estimation of density is performed. With only 50 patients, the pixel-histogram method would require histogram construction in 100 dimensional space. Even if the computation technology would be available to build it, histogram estimation of a 100D joint density would be highly biased and non-smooth. Graph methods with linear complexity in feature dimensionality make a stronger case for a direct estimation of entropy and divergence without the need to estimate density in higher dimensions. Certain imaging applications involve detecting and tracking regions that have a small volume, such as micro-calcification in the breast and tumor in the brain. Histograms are poor estimators of density in the small sample regime i.e., when the number of features or voxels is low. Small volumes cannot be matched accurately with histogram methods due to the noisy density estimation process. This thesis overcomes such drawbacks with entropic graph methods which provide remarkable estimates of entropy and divergence even with a small number of features. A multimodal face matching examples is included to demonstrate the versatility of entropic graphs to perform other image matching tasks.

A requirement to accomplish accurate high dimensional image matching is a discriminating feature space adapted to the image characteristics. Different modalities and imaging characteristics demand different features. Higher dimensional features used for this work include those based on independent component analysis (ICA), multidimensional wavelet image analysis and discrete cosine transforms (DCT). ICA is a data-driven process of estimating statistically independent basis and is used to extract features from ultrasound

images. Local basis projection coefficients are implemented by projecting local 8 by 8 sub-images of the image onto the ICA basis. The high dimensionality (= 64 for local basis projections) of these feature spaces precludes the application of standard entropy-based pattern matching methods and provides a good illustration of the power of our approach. Satellite images are registered with entropic graph methods operating on features extracted using Meyer wavelets. The ability of the wavelet basis to capture spatial-frequency information in a hierarchical setting makes them an attractive choice for use in registration.

For demonstration purposes, primitive tag features that are local quantized pixel neighborhoods are used. Tags serve as a good introductory feature set before advancing to other feature extraction methods. In the past, pixel-pairs [110] were the only vector features used within the MI context of registration. It should be noted that local feature extraction via basis projection is a commonly used technique for image representation [112, 123]. Image registration methods that do not rely on information divergence often use wavelet features, e.g. [129, 117]. In [64] multiresolution wavelet analysis is used to perform pixel-histogram MI-based matching from a coarse to a high resolution. ICA features are somewhat less common but have been similarly applied by Olshausen, Hyvärinen and others [77, 61, 101].

Further, this thesis seeks to overcome issues associated with the algorithmic construction of the MST and kNN graph. It explores advanced algorithms to overcome the time-memory limitations of building the MST and kNN graph. A disc-based approach is used to pick candidate nodes and cull the number of edges used to build the MST. Similar approaches have enabled rapid MST construction for about 100,000 feature samples residing in 64 dimensional space. By comparison, a full search method would require several minutes to construct the MST over 10,000 points in 2D. It is virtually impossible to construct a MST for 100,000 points in 64D spaces using a traditional full search approach. This

acceleration has allowed evaluation of entropic graphs methods over several hundreds of images with a variety of feature spaces. This was not possible earlier due to computational limitations of MST construction methods. A kd-tree approach is used to rapidly construct the kNN graph and further accelerate the estimation process. We establish that computational complexity is less of a hurdle in registration with graph methods due to contributions made here. Functions written in the ANSI C and MATLAB programming language for each of our entropic graph methods is included in the appendix and can be downloaded from the website [1].

This thesis has thus developed several new techniques to address the image registration problem. Comprehensive simulation analysis of the performance of each of these metrics in rotational image registration scenarios described above is presented. Extensive comparisons between these methods and other existing methods, such as the pixel-histogram method and correlation coefficient based methods are also included. Testing has been performed on several images from the same modality, such as Ultrasound, and different modalities, such as time-weighted MRI, on imagery obtained from medical, satellite and face-matching applications. Numerical performance comparisons among the metrics and features are aimed at identifying the algorithms that best discriminate between rotationally aligned and misaligned images. The rotational deformations are local in nature since the thesis does not intend to focus on iterative optimization techniques. The discrimination ability for local rotational deformations provides a good comparison of the accuracy of registration for more general image deformations. Sensitivity and robustness to noise is also evaluated.

The next section provides an outline so as to help navigation of the thesis.

1.4 Outline of the thesis

Chapter II describes the multi-modality image registration problem, chief components of an image registration system, historical approaches to solve the registration problem and the main contributions of this thesis.

Chapter III presents a host of dis-similarity measures such as α-divergence, α-mutual information, α-Jensen difference divergence, α-geometric arithmetic mean divergence, non-linear correlation coefficient and Henze-Penrose affinity. We choose to focus on these measures due to our ability to estimate them reliably without density estimation.

Chapter IV illustrates entropic graph based entropy estimation, estimation of α-Jensen difference divergence and Henze-Penrose divergence. Chapter IV further introduces several new approximations that can be used to estimate α-MI, NLCC and other divergences using nearest neighbor graphs.

As a generalization of the pixel-level representation for registration of images, tag, ICA and wavelet feature representations are described in Chapter V. Chapter V also provides descriptions of how feature selection can be performed with tag, ICA and wavelet features.

A significant amount of time was spent on developing faster MST and kNN methods. Improvements over existing algorithms are described in detail in Chapter VI. Source code is provided freely on the Internet [1] and in the appendix of this thesis.

In Chapter VII we present experiments to validate and evaluate entropic graph methods in the context of other similar registration methods were performed on real images derived from ultrasound data of the breast of female patients undergoing chemotherapy and MRI images of the brain. Other multisensor satellite and face images were also used for evaluation of our methods.

1.5 Publication related to thesis

① Neemuchwala HF, Hero AO and Carson PL, “Pairwise and simultaneous multi-image registration using entropic graphs”, (in preparation for), International Journal of Imaging, 2005.

② Neemuchwala HF, Hero AO, and Carson PL,“Image matching using alpha-entropy measures and entropic graphs”,(in press, http://dx.doi.org/10.1016/j.sigpro.2004.10.002) European Journal on Signal Processing, Special Issue on: Content-based Visual Information Retrieval, 2005.

③ Neemuchwala HF and Hero AO, “Entropic graphs for registration”, (in press) ‘Multi-sensor image fusion and its applications’, Eds. R. S. Blum and Z. Liu, Marcel-Dekker, Inc 2005.

④ Neemuchwala HF, Hero AO, and Carson PL, “Algorithms for constructing Euclidean minimum spanning trees in fully-connected graphs”, Technical Report CSPL Communications and Signal Processing Laboratory, The University of Michigan Ann Arbor, MI 48109-2122, 2005

⑤ Neemuchwala HF and Hero AO, “Image registration in higher dimensional feature space”, Proceedings of the IS&T/SPIE 17th Annual Symposium on Electronic Imaging: Science and Technology, San Jose, California, January 2005.

⑥ Neemuchwala HF, Hero AO, Carson PL and Meyer CR, “Local feature matching using entropic graphs”, Proceedings of the 2004 IEEE Symposium on Biomedical Imaging; From nano to macro, Arlington VA, April 2004.

⑦ Neemuchwala HF, Hero AO, and Carson PL,“Image registration using entropic graph-matching criteria”, Proceedings of 36th Asilomar Conf. on Signals Systems and

Computers, Pacific Grove, CA, Nov. 2002.

⑧ Neemuchwala HF, Hero AO, and Carson PL,“Feature Coincidence Trees for Registration of Ultrasound Breast Images”, Proceedings of IEEE Int. Conf. on Image Proc., Thesaloniki, Greece, Oct. 2001.

⑨ Neemuchwala HF, Hero AO, and Carson PL,“Feature Coincidence Trees for Registration of Ultrasound Images”, AIUM 46th Annual Convention, Nashville, March 10-13, J. Ultras. Med., 21, S55, 2002.

⑩ Carson PL, Kruecker JF, Meyer CR, LeCarpentier GL, Fowlkes JB, Roubidoux, MA, Neemuchwala HF, Hero AO, “Image Registration: Breast Applications, Accuracy and Advanced Metrics”, in Carson PL, Parker KJ, et al., Ultrasound Image Registration, Categorical Course, AIUM 46th Annual Convention, Nashville, March 10-13, J. Ultras. Med., 21, S73, 2002.

CHAPTER II

Image fusion and registration

Sensors provide information about the environment. These days, multiple sensors are available to sample different properties of the environment or of an object of interest. Presuming that each sensor is capable of measuring a new object property, streams of sensor data are used to recreate object views in the computer or processing unit. Multisensor information fusion addresses the problem of combining information from multiple sensors to arrive at inferences about the object or its role in the environment. In recent years, multisensor data fusion has been extensively investigated by researchers in a variety of disciplines, such as artificial intelligence [78], pattern recognition [90], medical imaging [86], automated target recognition [3], speaker identification in video [36], remote sensing [111], monitoring of manufacturing processes [22] and robotics [62]. Sensor fusion typically involves different signal sources. The animal brain routinely integrates up to five different types of sensory information perceived by the eyes, ears, nose, tongue and skin to achieve a more accurate assessment of the surrounding environment and identification of threats, thereby improving its chances of survival. Humans fuse multiple information at every stimulation of their senses by appropriate signals (Figure 2.0.1). For example, consider a barking dog. Visual information is gathered by eye, audio information is provided by the ears. The brain is capable of fusing both sources of information to deduce that the

dog is barking. Humans identify each other through a fusion of sight and voice. Animals identify their cubs through a fusion of smell, voice and sight. However, as easy as it is for the brain to perform this task, training a computer to fuse this information is not trivial.

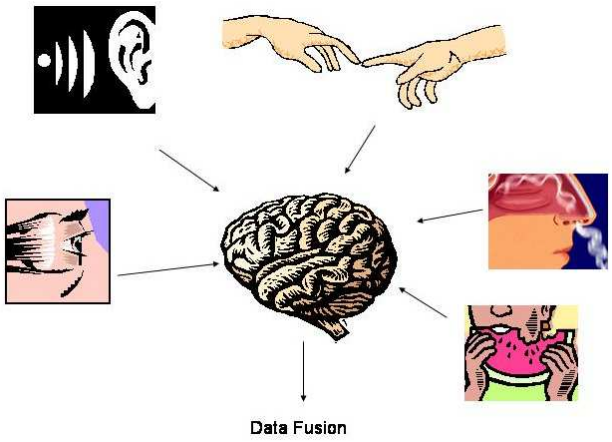


Figure 2.0.1: Data fusion: The human brain performs complex data fusion from up to five different sensors upon every sensory stimulation by an appropriate signal.

Information retrieved from an object in the real world has several dimensions, some are physical dimensions related to its position and appearance, some others are intrinsic properties such as its heat content and density. Consider the example of geostationary satellites that image the earth with several sensors monitoring properties of a region such as its visible geography, heat content, mineral content, water content and underlying geological activity among others. Sensors provide quantitative measurements which are dependent on the structure and configuration of the object under scrutiny. However, not all information is meaningful and even the meaningful data requires interpretation. To extract meaningful information from streams of sensory data, one has to devise features capable of capturing the structure and configuration of the object. Sensing is prone to information loss due to the sampling, physical limitations of the sensor, interference from nuisance objects and transients and projections. Inversion of the sensing process is ill-posed due to the information loss. One attempts to extract common information about the object

from multiple sensors to arrive at an inference. Multisensor data fusion addresses these problems by combining data not only from from multiple sensors but also from related information found in associated databases, to achieve improved accuracies and more specific inferences than could be achieved by the use of a single sensor alone. Multiple sensors overcome information loss by providing redundancy and reducing uncertainty in the measurements. Multisensor fusion critically depends upon multisensor registration. When the information from the sensors is not acquired simultaneously and from the same position and view angle, registration must be performed through software processing to correct for coordinate differences.

2.1 Motivations for Fusion in Imaging

Sensor fusion techniques are commonly adopted in applications where direct object perception is difficult, noise-prone or expensive. One example is medical image fusion, where organs inside the body are not directly accessible. In medical imaging, this inaccessibility mandates multiple sensor usage, each sensor providing measurements of a different tissue property. Information from a single sensor is fraught with common problems related to sensor noise, physical constraints, obstruction of view, shadowing, tissue movement and patient motion among others. Sensors providing complimentary tissue information or complimentary object views allow for a more complete observation of the anatomy, physiology (metabolism) or pathology (disease, tumor). Sensor fusion methods are then adopted to parse through the information from multiple sensors, perform complex deductions and provide consistent conclusions. Required margins of error from an automated fusion and inference system are required to be at least as low as that arrived at by the physician herself. An example of the use of fusion is in radiotherapy treatment, where CT and MRI are employed to provide complimentary soft-tissue hard-tissue information

in the brain and skull [121]. Mammography performed with X-Ray imaging is now fused with UL images of the breast to provide an additional perspective of the tumor or cyst and perform a better analysis of the case [65]. Treatment verification by comparison of pre- and post-intervention images, tumor growth monitoring or assessment of therapy can be performed using time series of UL data of breast tumors [71, 72], MRI scans on brain tumors [48] or CT scans on bones. In these applications, the sensor and sensing technology remains the same but the fusion occurs over data collected at two different time points.

Most automated artificial intelligence imaging systems rely on an information processing unit. In these systems multiple environment parameters are collected by sensors leading to an information overload from the glut of accumulated data. Automated systems are required to provide reliable decisions in a timely fashion. The amount of time needed to reach a reliable decision increases rapidly with the amount of information available. Sensor fusion is necessary to combine information in a way that removes inconsistencies and presents clearly the best interpretation of measurements input from many individual sources. Manual fusion requires a level of information processing that is extremely laborious and expensive and requires specialized training and knowledge. Automatic sensor fusion is a necessity to overcome time constraints and parse unimportant information. Through sensor fusion, we can combine measurements from several different sensors in order to combine different aspects of the environment into one coherent structure. When done properly, sensor fusion combines input from many independent sources of limited accuracy and reliability to give information of known accuracy and proven reliability.

Medical imaging applications use expensive sensors that are calibrated on a regular basis and are maintained to perform at high standards. Other applications require that sensors be placed in hostile environments where access is reduced or eliminated, such as in outer space, deep sea, forests, mountains or river beds. Due to this reason, applications

such as monitoring soil toxicity or water contamination can be addressed by distributing several hundreds of cheap sensors in the environment [70]. Such systems can be built with redundancy to reduce the impact of a single sensor failure on the outcome. Unforeseen, adverse circumstances and changing interfaces with the environment limit the ability of the some sensors to interact with their habitat. When numerous sensors gather data independently sensor fusion is performed to arrive at reliable conclusions and reduce the impact of sensor failure. Combining measurements from several different kinds of sensors can give a system more accurate information than otherwise possible. Combining several measurements from the same sensor makes a system less sensitive to noise because in the measurements of the same environment at different times the signal components are highly correlated while the noise components are independent.

Lastly, diversity in sensor types allows sensing of a variety of object or environment properties. Diversity could be achieved by harnessing sensors that focus on different bands in the electromagnetic spectrum. For example, visible and infrared sensors can be used in security systems, or visible light and audible sound can be used in a video camera. Satellites often image earth in up to 12 different bands of the electromagnetic spectrum. These observations are correlated with water vapor to predict weather patterns and with greenhouse gases to monitor environmental impact on food production. This diversity leads to a reduction in the probability of decision error and uncertainty encountered in the measurements thus making the sensing system more reliable which ultimately benefits the inference-making procedure. The probability of decision error in such a system would be expected to fall asymptomatically as the number of sensors providing new and useful object information is increased.

In summary, the advantages of sensor fusion over single sensor processing are due to the redundancy, diversity and complementarity among multiple sensors. When data from

multiple sensors is fused together the resultant observation is expected to have a higher signal to noise ratio, a reduction in overall measurement variance and a better and more sophisticated picture of the environment. Redundancy is caused by the use of multiple sensors to measure the same entity. It is well known that redundancy reduces uncertainty. This can be appreciated from the fact that for multiple sensors, the signal related to the measured quantity is often correlated, whereas the uncertainty associated with each individual sensor tends to be uncorrelated. If multiple sensors are different modalities, they measure the same scene with different laws of physics, and one obtains physical sensor diversity. Another diversity, spatial diversity, which offers different viewpoints of the sensed environment simply by having sensors in different locations, also plays a very important role in multisensor fusion. Multiple sensors observe a subset of the environment space, and the union of these subsets makes up broader environment observation. In this way, one achieves data complementarity.

2.2 The Role of Image Registration in Sensor Fusion

In many applications in medical imaging, satellite imaging and image and video processing, sensors acquire 2D cross-sectional and projection information. In volumetric imaging, several 2D images may be acquired to form a stack of cross-sectional views in 3D. A common problem related with such systems is the misalignment in the acquired images due to the coordinate differences in the images. This misalignment is further complicated by camera and object movement which change camera geometry relative to the object thus affecting object pose and view direction. For example, handheld UL transducers with a small field-of-view have a different coordinate system compared with large field-of-view X-Ray systems that are fixed on the floor. Further, the images are acquired at different resolutions, at different times and often with significant tissue change. Lastly, the

sensors acquire fundamentally different tissue properties thus the measurements differ in their units. For example, X-Ray sensors image object density, UL sensors image acoustic reflectivity, thermal sensors are sensitive to reflective heat energy, MRI sensors capture water content or and digital cameras capture visible light and thus the visual appearance of the object through a projection.

In systems that are not co-registered during image acquisition, the alignment of images is crucial and pivotal in the sensor fusion. Image registration is a precursor to sensor fusion and enables information extraction from multiple images. Recently, the distinction between fusion and registration has blurred in literature. Partly, this is due the evolution of sophisticated image registration algorithms that use fusion technology like pixel, feature or symbol level fusion. In this thesis registration and fusion terminology will be used interchangeably.

2.3 Chief Components of an Image Registration System

The three chief components of an effective image registration system (Figure 2.3.2) are: (1) definition of features that discriminate between different image poses; (2) adaptation of a matching criterion that quantifies feature similarity, is capable of resolving important differences between images, yet is robust to image artifacts; (3) implementation of optimization techniques which allow fast search over possible transformations. In this chapter, we shall be principally concerned with the first two components of the system. In a departure from conventional pixel-intensity features, we present techniques that use higher dimensional features extracted from images. We adapt traditional pixel matching methods that rely on entropy estimates to include higher dimensional features. We propose a general class of information theoretic feature similarity measures that are based on entropy and divergence and can be empirically estimated using entropic graphs, such as

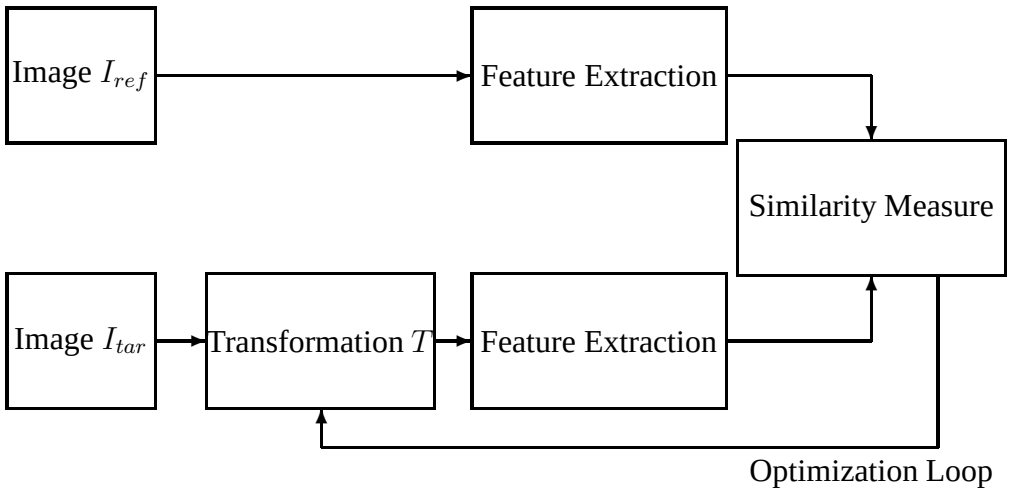


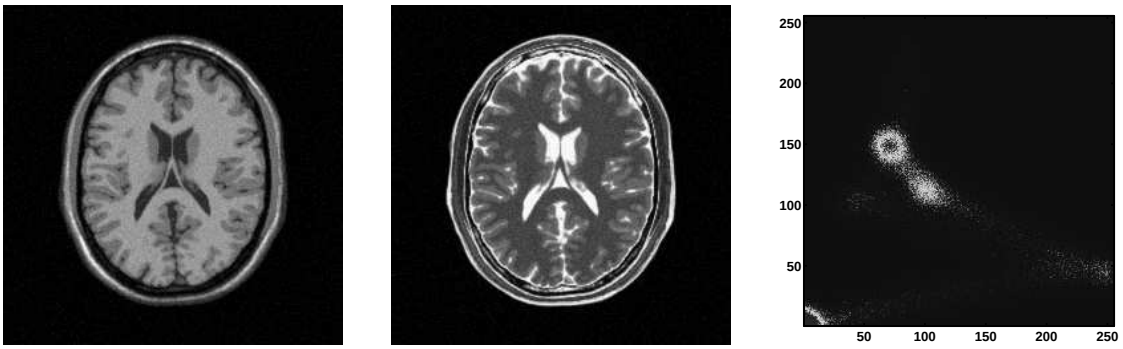
Figure 2.3.2: Block diagram of an image registration system

the minimal spanning tree (MST) or k-Nearest Neighbor (kNN) graph, and do not require density estimation or histograms.

2.4 Historical Perspective on Image Registration

Traditional approaches to image registration have included single pixel gray level features and correlation type matching functions. The correlation coefficient is a poor choice for the matching function in multi-sensor fusion problems. Multi-sensor images typically have intensity maps that are unique to the sensors used to acquire them and a direct linear correlation between intensity maps may not exist (Figure 2.4.3). Several other matching functions have been suggested in the literature [57, 63, 103]. Some of the most widespread techniques are: histogram matching [59]; texture matching [6]; intensity cross correlation [86]; optical flow matching [75]; kernel-based classification methods [27]; boosting classification methods [29, 68]; information divergence minimization [122, 116, 115, 49]; and mutual information (MI) maximization [125, 84, 45, 87, 17]. The last two methods can be called “entropic methods” since both use a matching criterion defined as a relative entropy between the feature distributions. The main advantage of entropic methods is that they

can capture non-linear relations between features in order to improve discrimination between poor and good image matches. When combined with a highly discriminatory feature set, and reliable prior information, entropic methods are very compelling and have been shown to be virtually unbeatable for some multimodality image registration applications [76, 87, 57]. However, due to the difficulty in estimating the relative entropy over high dimensional feature spaces, the application of entropic methods have been limited to one or two feature dimensions. The independent successes of relative entropy methods, e.g., MI image registration, and the use of high dimensional features, e.g., SVM's for handwriting recognition, suggest that an extension of entropic methods to high dimensions would be worthwhile. Encouraging initial studies on these methods have been conducted by these authors and can be found in [96, 94].



(a) T1 weighted MRI (b) T2 weighted MRI (c) Joint histogram

Figure 2.4.3: MRI images of the brain, with additive noise. (a) T1 weighted I_1, (b) T2 weighted I_2. Images courtesy [25]. Although acquired by a single sensor, the time weighting renders different intensity maps to identical structures in the brain. (c) Joint gray-level pixel coincidence histogram is clustered and does not exhibit a linear correlation between intensities.

2.5 Classification of Sensor Fusion Techniques

As documented above, sensor fusion techniques have several motivations depending upon the particular application. The actual process of sensor fusion can be conducted in

various ways. The particular method adopted determines the choice of parameters and affects the results of the fusion process. There are several criteria to categorize current sensor fusion techniques. These are broadly divided by: types of sensor data, levels of representation and choice of mathematical basis. In multimodal image registration applications, sensor data may come from a time-series of images, often with redundancy. Different modalities may provide complementary information about the object. Some applications such as atlas registration involve multimodal time-series registration. Thus types of sensor data may not provide the best framework for classification of the methods provided here. This work extends on the information theoretic framework to provide the mathematical foundation for registration. The choice of classification of fusion types covered in this thesis may be classified by examining the different levels of feature representation.

2.5.1 Classification by Levels of Representation

Applications of multisensor fusion may be characterized by the level of representation given to data during the fusion process. Observational data may be combined, or fused, at a variety of levels: signal, feature, and symbol levels. Note that these levels of fusion are only a rough classification of representation possibilities, and in no way can capture the subtlety of numerous applications. Image registration applications often involve temporal and spatial alignment of image signals. In the information-theoretic framework signals are modeled as random processes corrupted by uncorrelated noise and the fusion process is considered as an estimation procedure. Signal level fusion methodology refers to a combination of the signals in a group of sensors in order to arrive at a fusion decision. Signal level fusion methods such as weighted averaging of images are superseded by advanced metrics that correlate feature or symbolic relationships between images. However, in cases where signals represent different phenomenologies, signal-level fusion may be

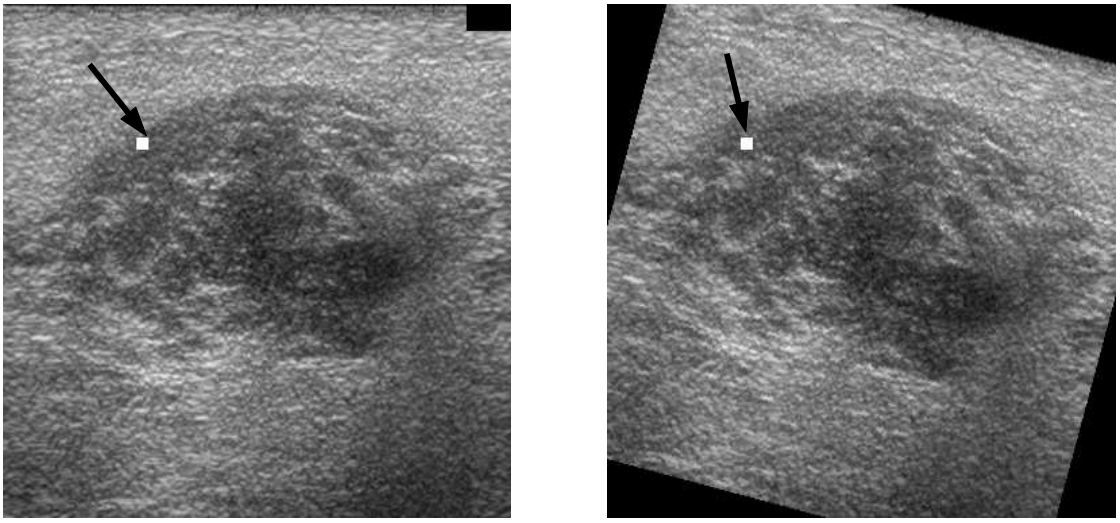
used as demonstrated in [37].

2.5.1.1 Feature-level fusion

In the multisensor registration scenario where each sensor observes a different object property, alignment must be accomplished through the use of features. Pixels are the most commonly used features for registration due to their low complexity and ease of use with minimum effort required in feature extraction. The downside of using primitive features is that they offer little or no spatial resolving capability of texture, image structure and spatial frequency. Figure 2.5.4 shows the coincidence measuring using pixel pairs from the images being considered for registration. When adequately represented, features have the ability to represent the sensory information, reduce the complexity of the processing procedure and increase the reliability of the processing results. Spatial features can be adapted for the purpose of image representation in multimodal image registration. Typical features extracted from an image and used for fusion include edges and regions of similar intensity. In Figure 2.5.5 a collection of quantized pixels (a pixel neighborhood) is used to extract edge information from image regions, called the method of tags in [4]. When multiple sensors have similar features at the same location, the likelihood that the features are actually present can be increased significantly and thus fusion improves the processing accuracy.

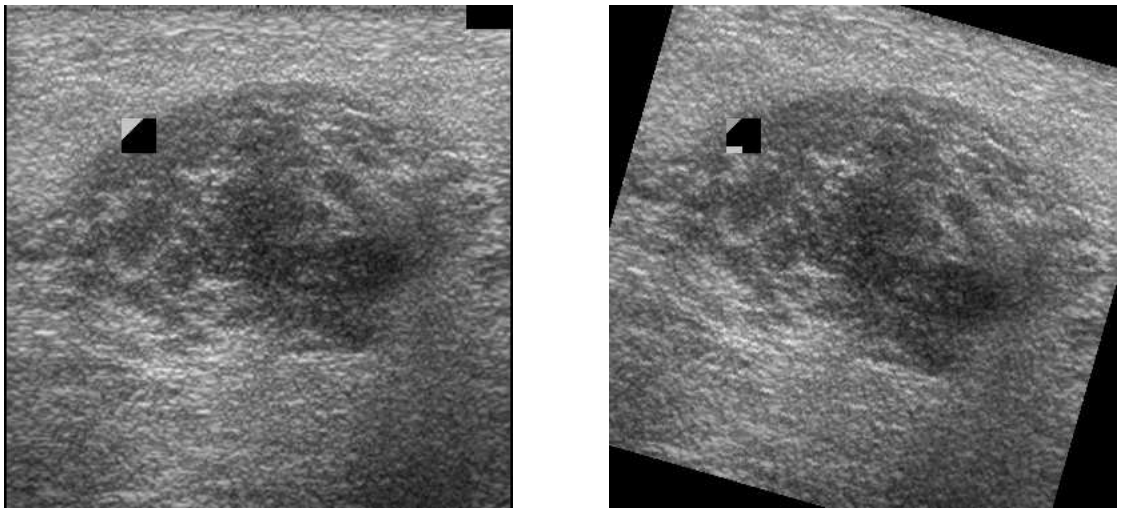
2.5.1.2 Symbol-level fusion

Symbolic representation of features strives to achieve a level of sophistication in the feature extraction process. Symbol-level fusion can effectively integrate the information from multiple sensors at the highest level of abstraction. The symbols can be derived through a symbolic reasoning processes that may make use of prior knowledge from a world model or sources external to the system. The most common type of symbol-level



(a) (b)

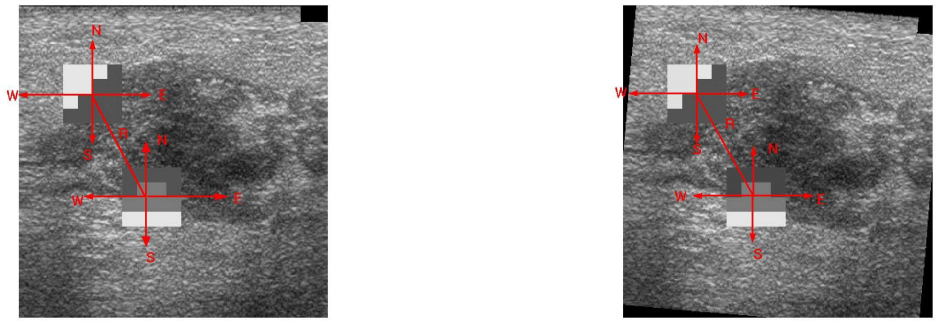
Figure 2.5.4: Single-pixel gray level coincidences are recorded by counting the number of co-occurrences of a pair of gray levels in the reference (a) and in the secondary (b) images at a pair of homologous pixel locations. Here the secondary image (b) is rotated by 15° relative to the reference image (a).



(a) (b)

Figure 2.5.5: Local tags features applied to image registration. Each pixel is labeled by a 8×8 tag type extracted using Geman's [44, 4] adaptive thresholding technique. Occurrences and coincidences of tag labels can be mapped to a coincidence histogram like Fig. 3.3.1

fusion application is pattern recognition. Feature information is extracted from sensor data, defining a point in the feature space. This point may be mapped to a symbolic interpretation of the world based on that symbol's neighborhood in the feature space. The ability to specify relations between features is used by humans for eclectic cognition and representation tasks. For example, humans identify voice modulation and facial expression and can relate the two to concur fight or flight decisions. This thesis attempts to identify primitive symbols through the relations between features. Figure 2.5.6 demonstrates feature extraction using symbolic directional relations between tag features at different spatial locations. Such spatial correspondence was introduced by [4]. Two lines running parallel can be identified as a feature and may identify a road or railway tracks on an aerial image or an artery in a medical image. Co-occurrences can then be evaluated between such features.



(a) Image I_1 (b) Image I_2

Figure 2.5.6: Symbol-level fusion: An abstraction of feature-level fusion where relations between features are expressed as features themselves. (a) Here the symbolic relation between two pixel neighborhoods is captured through the magnitude and direction of vector R. (b) The same vector R has now moved from its position in Ultrasound image I_1 due to the rotation of I_1. Tracking the change in position of R may help identify common features and enable fusion.

In conclusion, it should be noted that sensor fusion plays a significant role in the multi-

sensor imaging. Image registration is a precursor to the image fusion process since the extraction of common features can occur only after the images are spatially aligned. The development of multi-sensor image registration techniques has led to application of registration in cross modality applications. This thesis focuses on a feature-level representation of images where alignment between images is quantified by measuring similarity between features extracted from images. A critical component of image registration is the similarity measure used to estimate alignment between images. Features may be higher dimensional and hence require reliable and robust measures of similarity that can operate in higher dimensional space. The next chapter introduces entropic feature similarity and dis-similarity measures that enable image alignment through a higher dimensional feature representation of images.

CHAPTER III

Entropic Feature Similarity and Dissimilarity Measures

In this chapter we review entropy, relative entropy, and divergence as measures of dissimilarity between probability distributions. Let Y be a d-dimensional random vector and let $f(y)$ and $g(y)$ denote two possible densities for Y. Here Y will be a feature vector constructed from the reference image and the target image to be registered and f and g will be multidimensional feature densities. For example, information divergence methods of image retrieval [115, 31, 123] specify f as the estimated density of the reference image features and g as the estimated density of the target image features. When the features are discrete valued the densities f and g are interpreted as probability mass functions.

3.1 Statistical Framework

Let X_0 be a reference image, and consider a database of one or more target images $X_i, i = 1, \ldots, K$ of images to be registered to the reference image. Let $\mathcal{Z}_i$ be a feature vector extracted from X_i. Assume $\mathcal{Z}_i$ is a p-dimensional vector random variable. Specifically, assume that image X_i's feature vector $\mathcal{Z}_i$ is realization $\mathcal{Z}$ generated by a j.p.d.f. $f(\mathcal{Z}|\underline{\theta})$ which depends on a vector of unknown parameters $\underline{\theta}$ lying in a specified parameter space Θ. Under this probabilistic model the k-th observed image feature vector $\mathcal{Z}_k$ is assumed to have been generated from model $f(\mathcal{Z}_k|\underline{\theta_k})$, where $\underline{\theta_k}$ is called the "true pa-

rameter" underlying $\mathcal{Z}_k$. Under this statistical framework the similarity between images $X_0; X_1$ is reduced to similarity between feature probability models $f(\mathcal{Z}_0|\underline{\theta_0}); f(\mathcal{Z}_1|\underline{\theta_1})$.

3.2 Rényi Entropy and Divergence

The basis for entropic methods of image fusion is a measure of dissimilarity between densities f and g. Dis-similarity or divergence measures between f and g, denoted as $D(f\|g)$, should have the following properties:

- $D(f\|g) \geq 0$, when $f \neq g$,
- $D(f\|g) = 0$, when $f = g$ (a.e.),
- $D(f\|g)$ is smooth in f so that optimization can be readily accomplished over f,
- $D(f\|g)$ is easily estimated from data samples,
- $D(f\|g)$ is related to and is an accurate measure of mis-registration error,
- $D(f\|g)$ is generalizable to features in d-dimensional space,
- $D(f\|g)$ is robust to noise and sensitive to local and global perturbations in the images.

A very general entropic dis-similarity measure is the Rényi α-divergence, also called the Rényi α-relative entropy, between f and g of fractional order $\alpha \in (0, 1)$ [108, 28, 9] :

$$\begin{aligned} D_\alpha(f\|g) &= \frac{1}{\alpha - 1} \log \int g(z) \left(\frac{f(z)}{g(z)}\right)^\alpha dz \\ &= \frac{1}{\alpha - 1} \log \int f^\alpha(z) g^{1-\alpha}(z) dz. \end{aligned} \tag{3.1}$$

When the density f is supported on the d-dimensional unit cube and g is uniform over this domain the (negative) α-divergence reduces to the Rényi α-entropy of f:

$$H_\alpha(f) = \frac{1}{1 - \alpha} \log \int f^\alpha(z) dz. \tag{3.2}$$

When specialized to various values of α the α-divergence can be related to other well known divergence and affinity measures. Two of the most important examples are the Hellinger dissimilarity $-2\log\int\sqrt{f(z)g(z)}dz$ obtained when $\alpha = 1/2$, which is related to the Hellinger-Battacharya distance squared,

$$\begin{aligned} D_{Hellinger}(f\|g) &= \int\left(\sqrt{f(z)}-\sqrt{g(z)}\right)^2 dz \\ &= 2\left(1-\exp\left(\tfrac{1}{2}D_{\frac{1}{2}}(f\|g)\right)\right), \end{aligned} \tag{3.3}$$

and the Kullback-Liebler (KL) divergence [74], obtained in the limit as $\alpha \to 1$,

$$\lim_{\alpha\to 1} D_\alpha(f\|g) = \int g(z)\log\frac{g(z)}{f(z)}dz. \tag{3.4}$$

3.3 Mutual Information and α-Mutual Information

The mutual information (MI) can be interpreted as a similarity measure between the reference and target pixel intensities or as a dis-similarity measure between the joint density and the product of the marginals of these intensities. The MI was introduced for gray scale image registration [125, 84] and has since been applied to a variety of image matching problems [45, 76, 87, 106]. Let X_0 be a reference image and consider a transformation of the target image (X_1), defined as $X_T = T(X_1)$. We assume that the images are sampled on a grid of $M \times N$ pixels. Let (z_{0k}, z_{Tk}) be the pair of (scalar) gray levels extracted from the k-th pixel location in the reference and target images, respectively. The basic assumption underlying MI image matching is that $\{(z_{0k}, z_{Tk})\}_{k=1}^{MN}$ are independent identically distributed (i.i.d.) realizations of a pair $(\mathcal{Z}_0, \mathcal{Z}_T)$, ($\mathcal{Z}_T = T(\mathcal{Z}_1)$) of random variables having joint density $f_{0,1}(z_0, z_T)$. If the reference and the target images were perfectly correlated, e.g., identical images, then $\mathcal{Z}_0$ and $\mathcal{Z}_T$ would be dependent random variables. On the other hand, if the two images were statistically independent, the joint density of $\mathcal{Z}_0$

and $\mathcal{Z}_T$ would factor into the product of the marginals $f_{0,1}(z_0, z_T) = f_0(z_0)f_1(z_T)$. This suggests using the α-divergence $D_\alpha(f_{0,1}(z_0, z_T)\|f_0(z_0)f_1(z_T))$ between $f_{0,1}(z_0, z_T)$ and $f_0(z_0)f_1(z_T)$ as a similarity measure. For $\alpha \in (0, 1)$ we call this the α-mutual information (or α-MI) between $\mathcal{Z}_0$ and $\mathcal{Z}_T$ and it has the form

$$\begin{aligned} \alpha MI &= D_\alpha(f_{0,1}(\mathcal{Z}_0, \mathcal{Z}_T) \parallel f_0(\mathcal{Z}_0)f_1(\mathcal{Z}_T)) \\ &= \frac{1}{\alpha - 1} \log \int f_{0,1}^\alpha(z_0, z_T) f_0^{1-\alpha}(z_0) f_i^{1-\alpha}(z_T) dz_0 dz_T. \end{aligned} \tag{3.5}$$

When $\alpha \to 1$ the α-MI converges to the standard (Shannon) MI

$$\text{MI} = \int f_{0,1}(z_0, z_T) \log \left(\frac{f_{0,1}(z_0, z_T)}{f_0(z_0)f_1(z_T)} \right) dz_0 dz_T. \tag{3.6}$$

For registering two discrete $M \times N$ images, one searches over a set of transformations of the target image to find the one that maximizes the MI (3.6) between the reference and the transformed target. The MI is defined using features $(\mathcal{Z}_0, \mathcal{Z}_T) \in \{z_{0k}, z_{Tk}\}_{k=1}^{MN}$ equal to the discrete-valued intensity levels at common pixel locations (k, k) in the reference image and the rotated target image. We call this the "single pixel MI". In [125, 84], the authors empirically approximated the single pixel MI (3.6) by "histogram plug-in" estimates, which when extended to the α-MI gives the estimate

$$\widehat{\alpha MI} \stackrel{\text{def}}{=} \frac{1}{\alpha - 1} \log \sum_{z_0, z_T = 0}^{255} \hat{f}_{0,1}^\alpha(z_0, z_T) \left(\hat{f}_0(z_0)\hat{f}_1(z_T) \right)^{1-\alpha}. \tag{3.7}$$

In (3.7) we assume 8-bit gray level, $\hat{f}_{0,1}$ denotes the joint intensity level "coincidence histogram"

$$\hat{f}_{0,1}(z_0, z_T) = \frac{1}{MN} \sum_{k=1}^{MN} I_{z_{0k}, z_{Tk}}(z_0, z_T), \tag{3.8}$$

and $I_{z_{0k}, z_{Tk}}(z_0, z_T)$ is the indicator function equal to one when $(z_{0k}, z_{Tk}) = (z_0, z_T)$ and equal to zero otherwise. Other feature definitions have been proposed including gray level differences [17] and pixel pairs [110].

To illustrate the general procedure, the coincidence histogram is shown in Fig. 3.3.1 for the case of registration of US breast images X_0, X_1 (Fig. 3.3.2). Fig. 3.3.1 shows two cases. At top left is the coincidence histogram when the reference and secondary images are taken from the same two-dimensional slice of the US breast volume and are in perfect alignment ($X_0 = X_1$). At bottom left is the same histogram when the secondary image is rotated by 8°. The top right and bottom right panels in Fig. 3.3.1 are analogous except that the secondary images is extracted from a different two-dimensional slice separated from the reference (query) by 2mm which is approximately 4 times the distance of the minimum UL slice thickness . At this separation distance along the depth of the scan, the speckle in the images is decorrelated, but the anatomy in the images remains largely unchanged. In both cases the spread of the histogram is greater for the bottom panels (out of alignment) than for the top panels (in alignment) of the figure. The α-MI will take on greater values for the less spread top panels than for the more spread bottom panels.

Figure 3.3.3 illustrates the MI alignment procedure through a multisensor remote sensing example. Two modalities are illustrated, visible (a) and Infrared (b). Aligned images acquired by visible and thermally sensitive satellite sensors generate a joint gray level pixel coincidence histogram $f_{0,1}(z_0, z_1)$, shown in (c). Note, that the joint gray-level pixel coincidence histogram is not concentrated along the diagonal due to the mixed modalities of the images. When the thermal image is rotationally transformed, the corresponding joint gray-level pixel coincidence histogram $f_{0,1}(z_0, z_T)$ is dispersed, thus yielding a lower mutual information than in the case of aligned images. The higher dispersion of the gray-level pixel coincidence histogram for the mixed modality problem suggests that single-pixel features are inadequate.

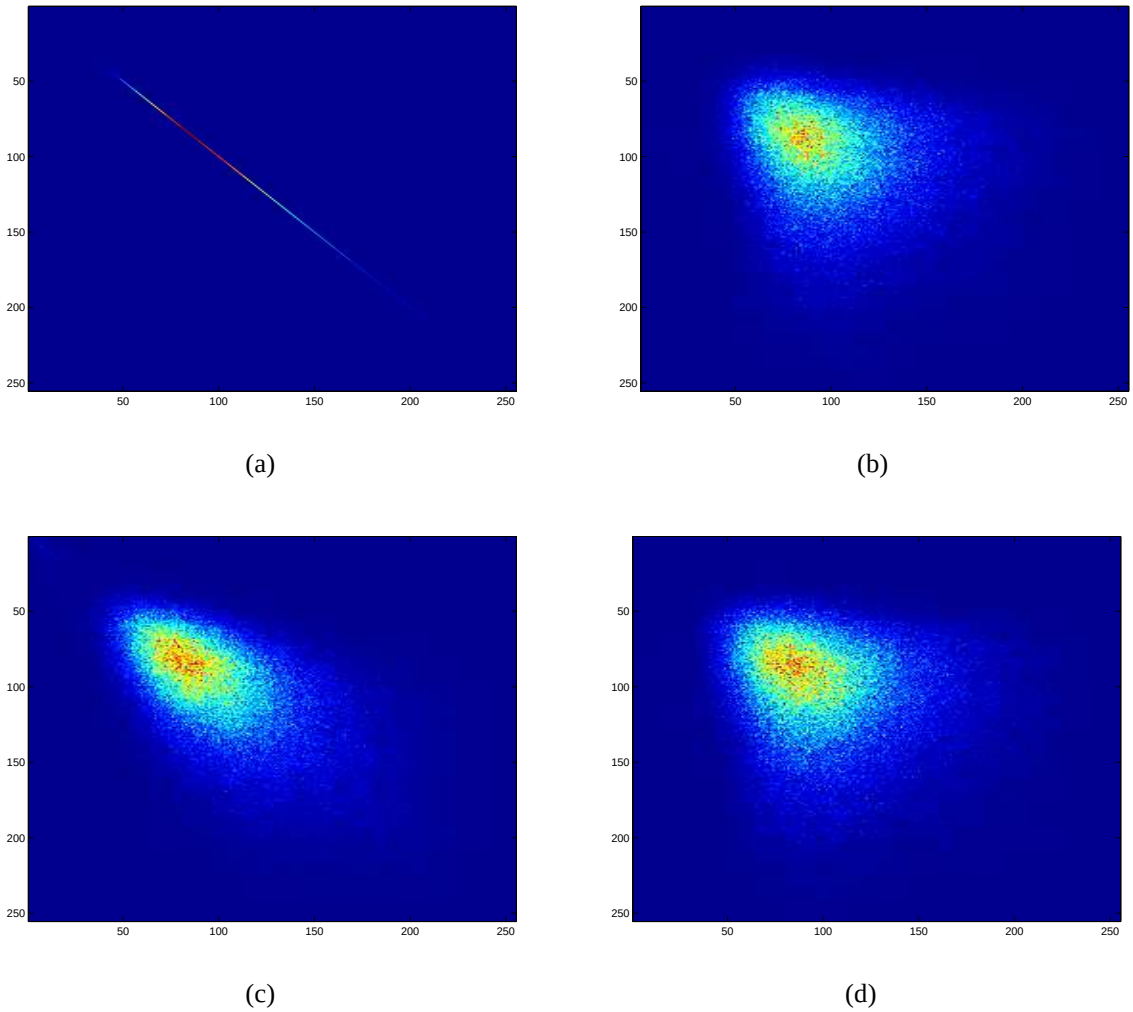
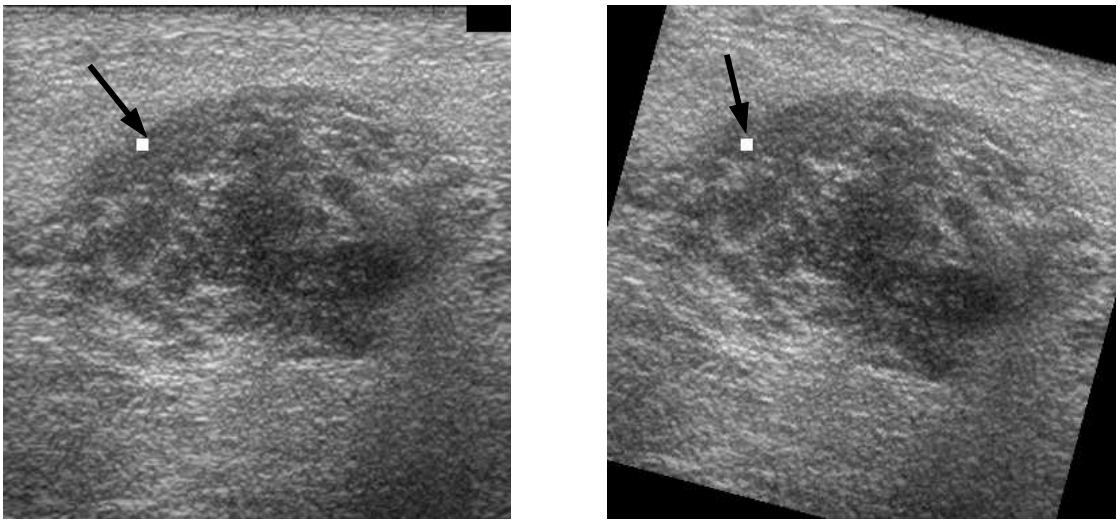


Figure 3.3.1: Joint coincidence histograms for single-pixel gray level features. Both horizontal and vertical axes of each panel are indexed over the gray level range of 0 to 255. (a): joint histogram scatter plot for the case that reference image (X_0) and secondary image (X_1) are the same slice of the US image volume (Case 142) at perfect 0° alignment ($X_1 = X_0$). (c): same as (a) except that reference and secondary are misaligned by 8° relative rotation as in Fig. 3.3.2. (b): same as (a) except that the reference and secondary images are from adjacent (2mm separation) slices of the image volume. (d): same as (c) except that images are misaligned by 8° relative rotation.



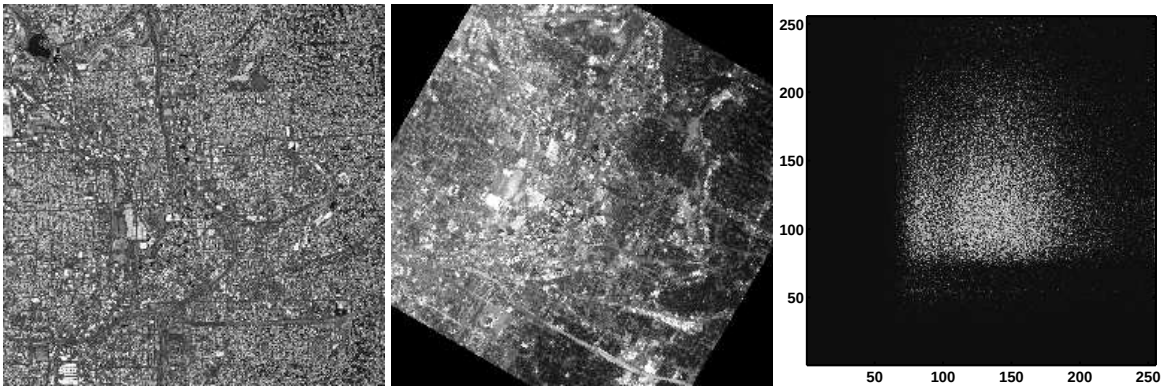
(a) (b)

Figure 3.3.2: Single-pixel gray level coincidences are recorded by counting number of co-occurrences of a pair of gray level in the reference (a) and in the secondary (b) images at a pair of homologous pixel locations. Here the secondary image (b) is rotated by 15° relative to the reference image (a).

(a) I_1: Urban Atlanta - thermal

(b) I_2: Urban Atlanta, visible

(c) Joint gray-level pixel coincidence histogram of registered I_1 and I_2



(d) I_1

(e) $T(I_2)$

(f) Joint gray-level pixel coincidence histogram of I_1 and $T(I_2)$

Figure 3.3.3: Mutual information based registration of multisensor, visible and thermal infrared, images of Atlanta acquired via satellite [105]. Top row (in-registration): (a) Visible light image I_1 (b) Thermal image I_2 (c) Joint gray-level pixel coincidence histogram $\hat{f}_{0,1}(z_0, z_1)$. Bottom row (out-of-registration): (d) Visible light image, unaltered I_1 (e) Rotationally transformed thermal image $T(I_2)$ (f) Joint gray-level pixel coincidence histogram shows wider dispersion $\hat{f}_{0,1}(z_0, z_T)$.

3.3.1 Relation of α-MI to Chernoff Bound

The α-MI (3.5) can be motivated as an appropriate registration function by large deviations theory through the Chernoff bound. Define the average probability of error $P_e(n)$ associated with a decision rule for deciding whether $\mathcal{Z}_T$ and $\mathcal{Z}_0$ are independent (hypothesis H_0) or dependent (hypothesis H_1) random variables based on a set of i.i.d. samples $\{z_{0k}, z_{Tk}\}_{k=1}^{n}$, where $n = MN$. For any decision rule, this error probability has the representation:

$$P_e(n) = \beta(n)P(H_1) + \alpha(n)P(H_0), \tag{3.9}$$

where $\beta(n)$ and $\alpha(n)$ are the probabilities of Type II (say H_0 when H_1 true) and Type I (say H_1 when H_0 true) errors, respectively, of the decision rule and $P(H_1) = 1 - P(H_0)$ is the prior probability of H_1. When the decision rule is the optimal minimum probability of error test the Chernoff bound implies that [30]:

$$\lim_{n\to\infty} \frac{1}{n} \log P_e(n) = - \sup_{\alpha\in[0,1]} \left\{(1-\alpha) D_\alpha(f_{0,1}(z_0, z_T) \| f_0(z_0) f_1(z_T)\right\}. \tag{3.10}$$

Thus the mutual α-information gives the asymptotically optimal rate of exponential decay of the error probability for testing H_0 vs H_1 as a function of the number $n = MN$ of samples. In particular, this implies that the α-MI can be used to select the optimal transformation T that maximizes the right side of (3.10). The appearance of the maximization over α implies the existence of an optimal parameter α ensuring the lowest possible registration error. When the optimal value α is not equal to 1 the MI criterion will be suboptimal in the sense of minimizing the asymptotic probability of error. For more discussion of the issue of optimal selection of α we refer the reader to [53].

3.4 α-Jensen Dissimilarity Measure

An alternative entropic dissimilarity measure between two distributions is the α-Jensen difference. This function was independently proposed by Ma [82, 52] and He *et al* [49] for image registration problems. It was also used by Michel *et al* in [89] for characterizing complexity of time-frequency images. For two densities f and g the α-Jensen difference is defined as [9]

$$\Delta H_\alpha(p, f, g) = H_\alpha(pf + qg) - [pH_\alpha(f) + qH_\alpha(g)], \tag{3.11}$$

where $\alpha \in (0, 1)$ and $p \in [0, 1]$ and $q = 1 - p$. As the α-entropy $H_\alpha(f)$ is strictly concave in f, Jensen's inequality implies that $\Delta H_\alpha(p, f, g) > 0$ when $f \neq g$ and $\Delta H_\alpha(p, f, g) = 0$ when $f = g$ (a.e.). Thus the α-Jensen difference is a bona-fide measure of dissimilarity between f and g.

The α-Jensen difference can be applied as a surrogate optimization criterion in place of the α-MI or the α-divergence. When applied as a surrogate for α-divergence One identifies $f = f_1(z_T)$ and $g = f_0(z_0)$ in (3.11). In this case an image match occurs when the α-Jensen difference is minimized over i. This is the approach taken by [49, 52] for image registration applications and discussed in more detail below.

On the other hand, the α-Jensen difference can also be used as a surrogate for the α-MI if one identifies $f = f_{0,1}(z_0, z_T)$ and $g = f_0(z_0)f_1(z_T)$ in (3.11). In this case to find a matching image to a query the α-Jensen difference is maximized over T. Asymptotic comparison between the α-MI and the α-Jensen difference can give useful insight [53]. It can be shown that when the features $\mathcal{Z}_0, \mathcal{Z}_T$ are nearly independent then the most discriminating value of α is 1/2 for the α-MI. For the α-Jensen difference the best value of α is 1 and the best value of p is 1/2. While use of α-Jensen as a surrogate for α-MI is certainly worthy of additional study, its computational requirements and its performance

appear similar to that of α-MI and therefore we do not consider it any further.

3.5 α-Geometric-Arithmetic Mean Divergence

The α-geometric-arithmetic (α-GA) mean divergence [118] is another measure of dissimilarity between probability distributions. Given continuous distributions f and g, the α-GA is defined as:

$$\begin{aligned} {}_{\alpha}D_{GA}(f,g) &= D_{\alpha}(pf+qg\|f^p g^q) \\ &= \frac{1}{\alpha-1}\log\int (pf(z)+qg(z))^{\alpha}(f^p(z)g^q(z))^{1-\alpha}dz \end{aligned} \tag{3.12}$$

The α-GA divergence is a measure of the discrepancy between the arithmetic mean and the geometric mean of f and g, respectively, with respect to weights p and $q = 1-p$, $p \in [0,1]$. The α-GA divergence can thus be interpreted as the dissimilarity between the weighted arithmetic mean $pf(x)+qg(x)$ and the weighted geometric mean $f^p(x)g^q(x)$. Similarly to the α-Jensen difference (3.11), the α-GA divergence is equal to zero if and only if $f = g$ (a.e.) and is otherwise greater than zero.

3.6 Henze-Penrose Affinity

While divergence is a measure of dissimilarity between distributions, similarity between distributions can be measured by affinity measures. One measure of affinity between probability distributions f and g is

$$A_{HP}(f,g) = 2pq\int \frac{f(z)g(z)}{pf(z)+qg(z)}dz, \tag{3.13}$$

with respect to weights p and $q = 1-p, p \in [0,1]$. This affinity measure was introduced by Henze and Penrose [50] as the limit of the Friedman-Rafsky statistic [42] and we shall call it the Henze-Penrose (HP) affinity. The HP affinity can be related to the divergence

measure:

$$D_{HP}(f\|g) = 1 - A_{FR}(f,g) = \int \frac{p^2 f^2(z) + q^2 g^2(z)}{pf(z) + qg(z)} dz \tag{3.14}$$

All of the above divergence measures can be obtained as special cases of the general class of f-divergences, e.g., as defined in [28, 9]. Two categories exist in the list of divergences presented above. Some, like the αJensen between two densities, αGA and Henze-Penrose affinity operate in the marginal spaces with half the dimensionality of some others like the αMI and Shannon MI that operate in the joint space of distributions.

3.7 Entropy Estimation and Divergence

Accurate estimation of divergence is related to accurate estimation of the entropy. Three general classes of entropy estimation methods can be identified: parametric estimators, non-parametric estimators based on density or function estimation, and non-parametric estimators based on direct estimation. The first two methods use density "plug-in" techniques where a parametric or non-parametric density estimate $\hat{f}$ is simply plugged into the divergence formula. When an accurate parametric model and good parameter estimates are available parametric plug-in estimates of divergence are attractive due to their $1/\sqrt{n}$ RMS convergence properties [82]. An analytical parametric form of the divergence can often be derived over the parametric class of densities considered and maximum likelihood can be used to estimate parameters in the divergence formula. Non-parametric plug-in divergence estimates do not benefit from closed form parametric expressions for divergence but avoid pitfalls of model dependent estimates. For example, when a non-parametric estimate of $\hat{f}$ is available the following plug-in estimates of α-entropy is natural

$$H_\alpha(\hat{f}) = \frac{1}{1-\alpha}\log\int \hat{f}^\alpha(z)dz. \tag{3.15}$$

For the special case of Shannon entropy $\lim_{\alpha\to 1} H_\alpha = -\int f(z)\log f(z)dz$ non-parametric estimation methods have included: histogram estimation plug-in, kernel density estimation plug-in and sample-spacing density estimation plug-in. The main difficulties with non-parametric plug-in methods are due to the infinite dimension of the spaces in which the unconstrained densities lie. Specifically: density estimator performance is poor without stringent smoothness conditions; no unbiased density estimators generally exist; density estimators have high variance and are sensitive to outliers; the high dimensional integration in Equation 3.15 might be difficult. Consider the α-entropy (Equation 3.2) which could be estimated by plugging in feature histogram estimates of the multivariate density f. A deterrent to this approach is the curse of dimensionality, which imposes prohibitive computational burden when attempting to construct histograms in large feature dimensions. For a fixed resolution per coordinate dimension the number of histogram bins increases geometrically in feature vector dimension. For example, for a 32 dimensional feature space even a coarse 10 cells per dimension would require keeping track of 10^{32} bins in the histogram, an unworkable and impractically large burden for any envisionable digital computer. As high dimensional feature spaces can be more discriminatory this creates a barrier to performing robust high resolution histogram-based entropic registration.

The problems with the above methods can be summarized by the basic observation: on the one hand parameterizing the divergence and entropy functionals with infinite dimensional density function models is a costly over parameterization, while on the other hand artificially enforcing lower dimensional density parametrization can produce significant bias in the estimates. This observation has motivated us to develop direct methods which

accurately estimate the entropy without the need for performing artificial low dimensional parameterizations or non-parametric density estimation. The next chapter describes the method for estimating the α-entropy via an entropic graph whose vertices are the locations of the feature vectors in feature space. This thesis focuses on those measures of divergence where an implementation using entropic graph methods is known and tractable.

CHAPTER IV

Entropic Graph Estimators

A principal focus of this thesis is the use of minimal graphs over feature vectors $\mathcal{Z}_n = \{z_1, \ldots, z_n\}$, and their associated minimal edge lengths, for estimation of entropy of the underlying feature density $f(z)$. For consistent estimates we require convergence of minimal graph length to an entropy related quantity. Such convergence issues have been studied for many years, beginning with Beardwood, Halton and Hammersley [11]. The monographs of Steele [114] and Yukich [132] cover the interesting developments in this area. In the general unified framework of Redmond and Yukich [107] a widely applicable convergence result can be invoked for graphs whose length functionals can be shown to be Euclidean, continuous and quasi additive. This result can often be applied to minimal graphs constructed by minimizing a graph length function L_γ of the form:

$$L_\gamma(\mathcal{Z}_n) = \min_{E \in \Omega} \sum_{e \in E} \|e(\mathcal{Z}_n)\|^\gamma,$$

where Ω is a set of graphs with specified properties, e.g., the class of acyclic or spanning graphs, e is an edge in Ω, $\|e\|$ is the Euclidean length of e, γ is called the edge exponent or the power weighting constant, and $0 < \gamma < d$. The determination of L_γ requires a combinatorial optimization over the set Ω.

If $\mathcal{Z}_n = \{z_1, \ldots, z_n\}$ is a random i.i.d. sample of d-dimensional vectors drawn from a Lebesgue multivariate density f and the length functional L_γ is continuous quasi additive

then the following limit holds [107]

$$\lim_{n\to\infty} L_\gamma(\mathcal{Z}_n)/n^\alpha = \beta_{d,\gamma} \int f^\alpha(z)dz, \quad (a.s.) \tag{4.1}$$

where $\alpha = (d-\gamma)/d$ and $\beta_{d,\gamma}$ is a constant independent of f. Comparing this to the expression (3.2) for the Rényi entropy it is obvious that an entropy estimator can be constructed as $(1-\alpha)^{-1}\log\left(L_\gamma(\mathcal{Z}_n)/n^\alpha\right) = H_\alpha(f) + c$, where $c = (1-\alpha)^{-1}\log\beta_{d,\gamma}$ is a removable bias. Furthermore, it is seen that one can estimate entropy for different values of $\alpha \in [0,1]$ by adjusting γ. In many cases the topology of the minimal graph is independent of γ and only a single combinatorial optimization is required to estimate H_α for all α.

A few words are in order concerning the sufficient conditions for the limit (4.1). Roughly speaking, continuous quasi additive functionals can be approximated closely by the sum of the weight functionals of minimal graphs constructed on a uniform partition of $[0,1]^d$. Examples of graphs with continuous quasi-additive length functional are the Euclidean minimal spanning tree, the traveling salesman tour solving the traveling salesman problem (TSP), the steiner tree, the Delaunay triangulation, and the k nearest neighbor graph. An example of a graph that does not have a continuous quasi additive length functional is the k-point MST (kMST) discussed in [56].

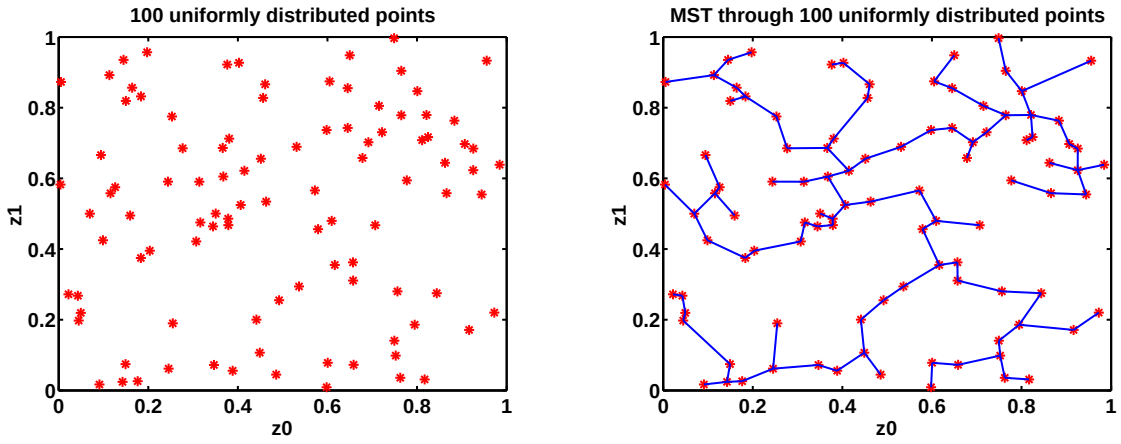
Even though any continuous quasi additive functional could in principle be used to estimate entropy via relation (4.1), only those that can be simply computed will be of interest to us here. An computationally intractable example is the TSP length functional $L_\gamma^{TSP}(\mathcal{Z}_n) = \min_{C\in c}\sum_{e\in C}\|e\|^\gamma$, where C is a cyclic graph that spans the points $\mathcal{Z}_n$ and visits each point exactly once. Construction of the TSP is NP hard and hence is not attractive for practical image registration applications. The following sections describe, in detail, the MST and kNN graph functionals.

4.1 Minimal Spanning Tree Entropy Estimator

A spanning tree is a connected acyclic graph which passes through all n feature vectors in $\{\mathcal{Z}_i\}_{i=1}^n$. The MST connect these points with $n-1$ edges, denoted $\{e_i\}$, in such a way as to minimize the total length:

$$L_\gamma(\mathcal{Z}_n) = \min_{e \in T} \sum_e \|e\|^\gamma, \tag{4.2}$$

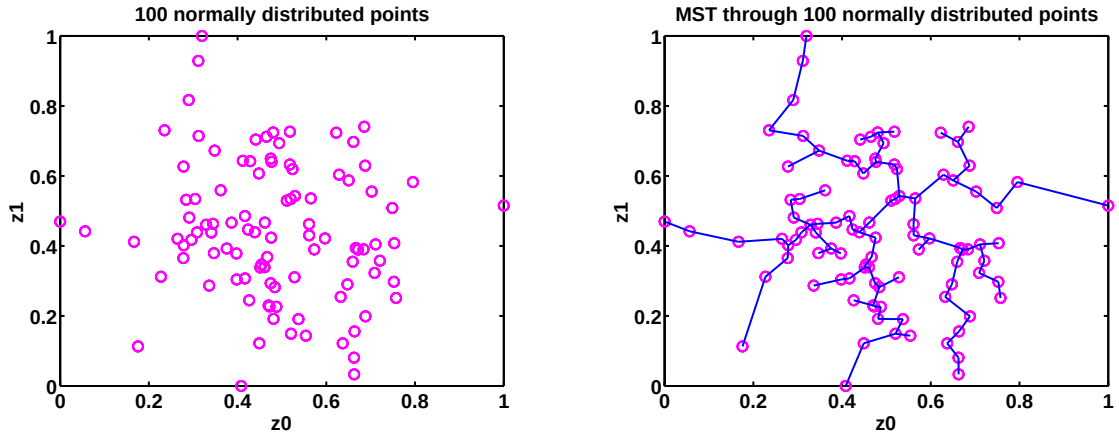
where T denotes the class of acyclic graphs (trees) that span $\mathcal{Z}_n$. While several spanning trees can be built over a set of points, the MST is unique if no two points overlap. However, the length of the MST, which has asymptotic convergence to αentropy, is always unique. is always unique See Figures 4.1.1 and 4.1.2 for an illustration when each $\mathcal{Z}_n$ is a set of n points in the unit square. We adopt $\gamma = 1$ for the following experiments.



(a) (b)

Figure 4.1.1: (a) A set of $n = 100$ uniformly distributed points $\{\mathcal{Z}_i\}_{i=1}^n$ in the unit square in $\mathbb{R}^2$ and (b) the corresponding Minimal Spanning Tree (MST).

The MST length $L_n = L(\mathcal{Z}_n)$ is plotted as a function of n in Figure 4.1.3 for the case of an i.i.d. uniform sample (right panel) and non-uniform sample (left panel) of $n = 100$ points in the plane. It is intuitive that the length of the MST spanning the more



(a) (b)

Figure 4.1.2: (a) A set of $n = 100$ normally distributed points $\{\mathcal{Z}_i\}_{i=1}^{n}$ in the unit square in $\mathbb{R}^2$ and (b) the corresponding Minimal Spanning Tree (MST).

concentrated non-uniform set of points increases at a slower rate in n than does the MST spanning the uniformly distributed points. This observation has motivated the MST as a way to test for randomness in the plane [58]. As shown in [132], the MST length is a continuous quasi additive functional and satisfies the limit (4.1). More precisely, with $\alpha \stackrel{\text{def}}{=} (d-\gamma)/d$ the log of the length function normalized by n^α converges (a.s.) within a constant factor to the α-entropy.

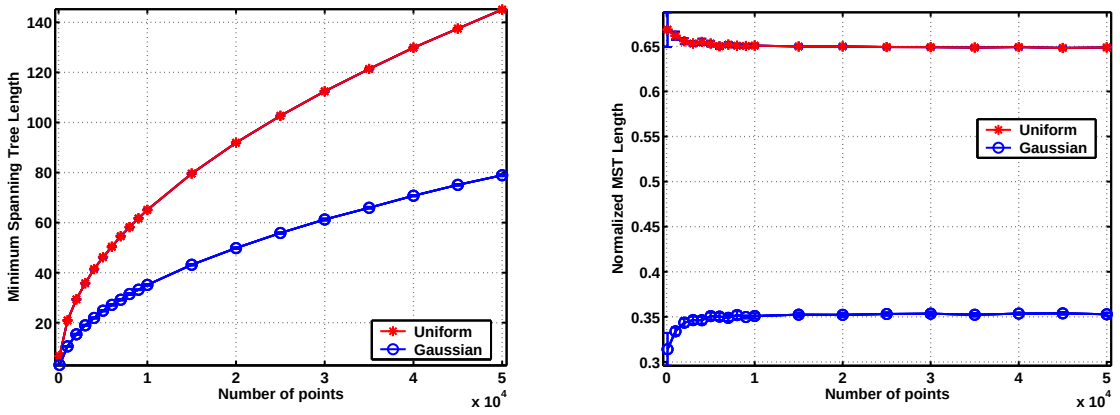
$$\lim_{n\to\infty} \log\left(\frac{L_\gamma(\mathcal{Z}_n)}{n^\alpha}\right) = H_\alpha(f) + c_{MST}, \quad \text{(a.s.)}, \tag{4.3}$$

Thus we can identify the difference between the asymptotes shown on the left Figure 4.1.3 as the difference between the α-entropies of the uniform and non-uniform densities ($\alpha = 1/2$). If f is the underlying density of $\mathcal{Z}_n$, the α-entropy estimator

$$\widehat{H}_\alpha(\mathcal{Z}_n) = 1/(1-\alpha)\left[\log L_\gamma(\mathcal{Z}_n)/n^\alpha - \log \beta_{d,\gamma}\right], \tag{4.4}$$

is an asymptotically unbiased and almost surely consistent estimator of the α-entropy of f where $\beta_{d,\gamma}$ is a constant which does not depend on the density f.

The constant $(c_{MST} = (1-\alpha)^{-1}\log\beta_{d,\gamma})$ in (4.3) is a bias term that can be estimated offline. The constant $\beta_{d,\gamma}$ is the limit of $L_\gamma(\mathcal{Z}_n)/n^\alpha$ as $n \to \infty$ for a uniform distribution $f(z) = 1$ on the unit cube $[0,1]^d$. This constant can be approximated by Monte Carlo simulation of mean MST length for a large number of d-dimensional random samples on the unit cube.



(a) (b)

Figure 4.1.3: (a) Mean Length functions L_n of MST implemented with $\gamma = 1$ and (b) $L_n/\sqrt{n}$ as a function of n for uniform and normal distributed points.

The MST approach to estimating the α-Jensen difference between the feature densities of two images can be implemented as follows. Assume two sets of feature vectors $\mathcal{Z}_0 = \{z_0^{(i)}\}_{i=1}^{n_0}$ and $\mathcal{Z}_1 = \{z_1^{(i)}\}_{i=1}^{n_1}$ are extracted from images X_0 and X_1 and are i.i.d. realizations from multivariate densities f_0 and f_1, respectively. In the applications explored in this paper $n_0 = n_1$ but it is worthwhile to maintain this level of generality. Define the set union $\mathcal{Z} = \mathcal{Z}_0 \cup \mathcal{Z}_1$ containing $n = n_0 + n_1$ unordered feature vectors. If n_0, n_1 increase at a constant rate as a function of n then any consistent entropy estimator constructed from the vectors $\{\mathcal{Z}^{(i)}\}_{i=1}^{n_0+n_1}$ will converge to $H_\alpha(pf_0 + qf_1)$ as $n \to \infty$ where $p = \lim_{n\to\infty} n_0/n$. This motivates the following finite sample entropic graph estimator of

α-Jensen difference

$$\Delta\widehat{H}_\alpha(p, f_0, f_1) = \widehat{H}_\alpha(\mathcal{Z}_0 \cup \mathcal{Z}_1) - [p\widehat{H}_\alpha(\mathcal{Z}_0) + q\widehat{H}_\alpha(\mathcal{Z}_1)], \tag{4.5}$$

where $p = n_0/n$, $\widehat{H}_\alpha(\mathcal{Z}_0 \cup \mathcal{Z}_1)$ is the MST entropy estimator constructed on the n point union of both sets of feature vectors and the marginal entropies $\widehat{H}_\alpha(\mathcal{Z}_0)$, $\widehat{H}_\alpha(\mathcal{Z}_1)$ are constructed on the individual sets of n_0 and n_1 feature vectors, respectively. We can similarly define a density-based estimator of α-Jensen difference. Observe that for affine image registration problems the marginal entropies $\{H_\alpha(f_i)\}_{i=1}^K$ over the set of image transformations will be identical, obviating the need to compute estimates of the marginal α-entropies.

As contrasted with histogram or density plug-in estimator of entropy or Jensen difference, the MST-based estimator enjoys the following properties [53, 51, 56]: it can easily be implemented in high dimensions; it completely bypasses the complication of choosing and fine tuning parameters such as histogram bin size, density kernel width, complexity, and adaptation speed; as the topology of the MST does not depend on the edge weight parameter γ, the MST α-entropy estimator can be generated for the entire range $\alpha \in (0, 1)$ once the MST for any given α is computed; the MST can be naturally robustified to outliers by methods of graph pruning. On the other hand the need for combinatorial optimization may be a bottleneck for a large number of feature samples for which accelerated MST algorithms are necessary.

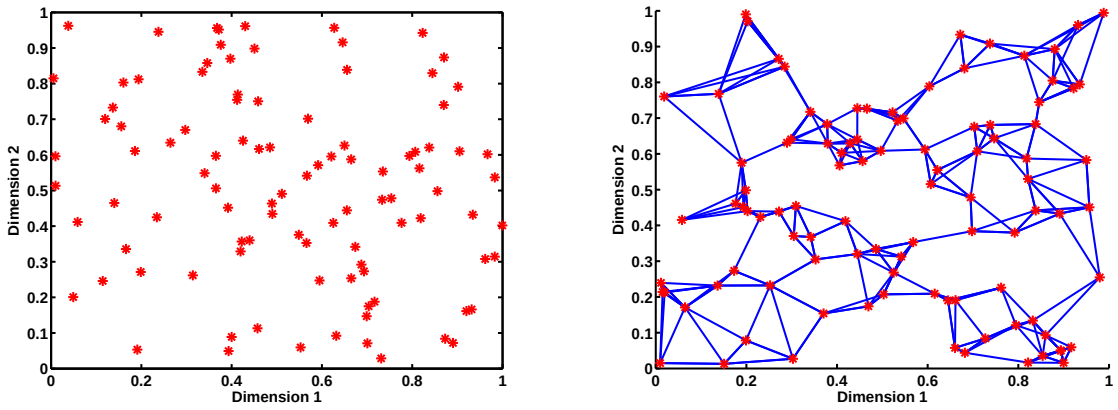
4.2 Nearest Neighbor Graph Entropy Estimator

The k-nearest neighbor graph is a continuous quasi-additive power-weighted graph that is a computationally attractive alternative to the MST. Given i.i.d vectors $\mathcal{Z}_n$ in $\mathbb{R}^d$,

the 1-nearest neighbor of z_i in $\mathcal{Z}_n$ is given by

$$\arg \min_{z \in \mathcal{Z}_n \setminus \{z_i\}} \|z - z_i\|, \tag{4.6}$$

where $\|z - z_i\|$ is the usual Euclidean (L_2) distance in $\mathbb{R}^d$. For general integer $k \geq 1$, the k-nearest neighbor of a point is defined in a similar way [13, 19, 98]. The kNN graph puts a single edge between each point in $\mathcal{Z}_n$ and its k-nearest neighbors. Let $\mathcal{N}_{k,i} = \mathcal{N}_{k,i}(\mathcal{Z}_n)$ be the set of k-nearest neighbors of z_i in $\mathcal{Z}_n$. The kNN problem consists of finding the set $N_{k,i}$ for each point z_i in the set $\mathcal{Z}_n - \{z\}$. As with the MST, the length of a kNN graph built over a set of random vectors is always unique. See Figures 4.2.4 and 4.2.5 for an illustration when $\mathcal{Z}_n$ are points in the unit square.



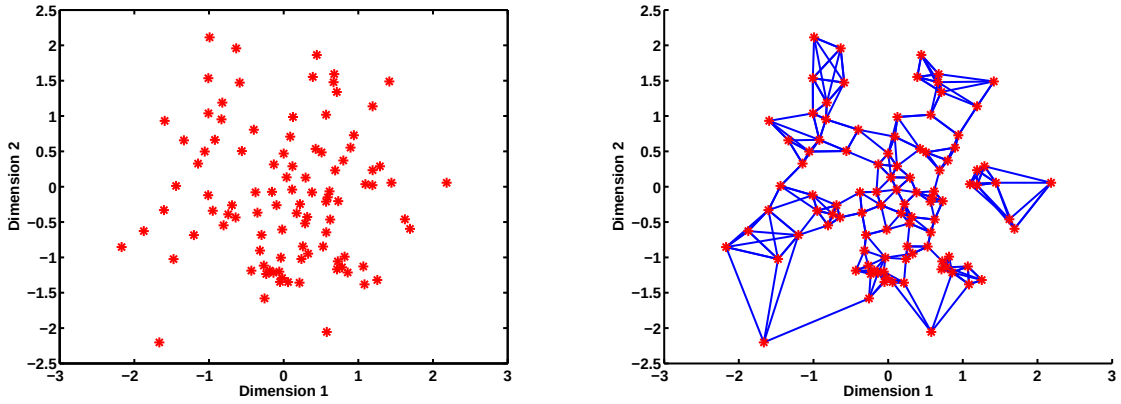
(a)

Figure 4.2.4: (a) A set of $n = 100$ uniformly distributed points $\{\mathcal{Z}_i\}_{i=1}^n$ in the unit square in $\mathbb{R}^2$ and (b) the corresponding k-Nearest Neighbor graph $(k = 4)$.

This problem has exact solutions that run in linear-log-linear time [15]. The total graph length is:

$$L_{\gamma,k}(\mathcal{Z}_n) = \sum_{i=1}^{N} \sum_{e \in N_{k,i}} \|e\|^{\gamma}. \tag{4.7}$$

In general, the kNN graph will count edges at least once, but sometimes count edges more



(a) (b)

Figure 4.2.5: (a) A set of $n = 100$ normally distributed points $\{\mathcal{Z}_i\}_{i=1}^n$ in the unit square in $\mathbb{R}^2$ and (b) the corresponding k-Nearest Neighbor graph $(k = 4)$.

than once. For example, if two points X_1 and X_2 are mutual k-nearest neighbors, then the same edge between X_1 and X_2 will be doubly counted.

Analogously to the MST, the log length of the kNN graph has limit

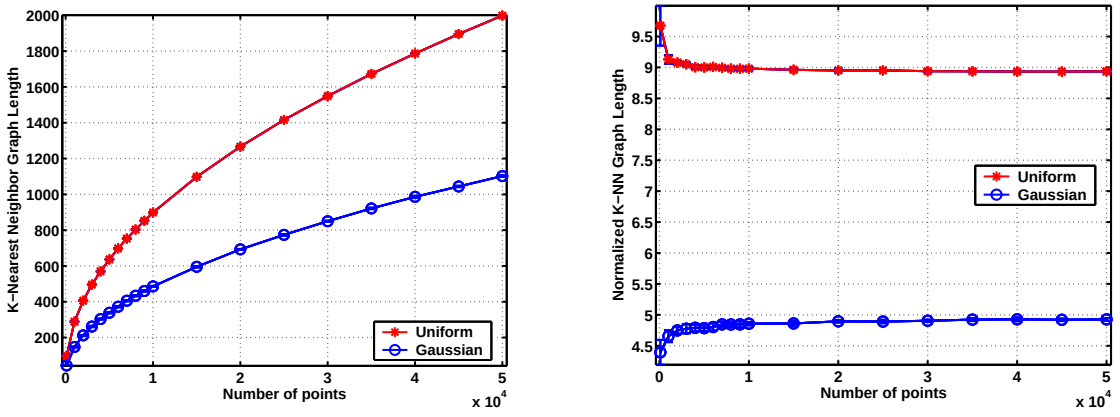
$$\lim_{n\to\infty} \log\left(\frac{L_{\gamma,k}(\mathcal{Z}_n)}{n^\alpha}\right) = H_\alpha(f) + c_{kNNG}, \quad \text{(a.s.)}. \tag{4.8}$$

Once again this suggests an estimator of the Renyi α-entropy

$$\widehat{H}_\alpha(\mathcal{Z}_n) = 1/(1-\alpha)\left[\log L_{\gamma,k}(\mathcal{Z}_n)/n^\alpha - \log \beta_{d,\gamma,k}\right], \tag{4.9}$$

As in the MST estimate of α-entropy, the constant $c_{kNNG} = (1-\alpha)^{-1}\log\beta_{d,\gamma,k}$ can be estimated off-line by Monte Carlo simulation of the kNNG on random samples drawn from the unit cube. The complexity of the kNNG algorithm is dominated by the nearest neighbor search, which can be done in $O(n\log n)$ time for n sample points. This contrasts with the MST that requires a $O(n^2\log n)$ implementation.

A related k-NN graph is the graph where edges connecting two points are counted only once. Such a graph eliminates one of the edges from each point pair that are mutual k-nearest neighbors. A kNN graph can be built by pruning such that every unique



(a) (b)

Figure 4.2.6: (a) Mean Length functions L_n of kNN graph implemented with $\gamma = 1$ and (b) $L_n/\sqrt{n}$ as a function of n for uniform and Gaussian distributed points.

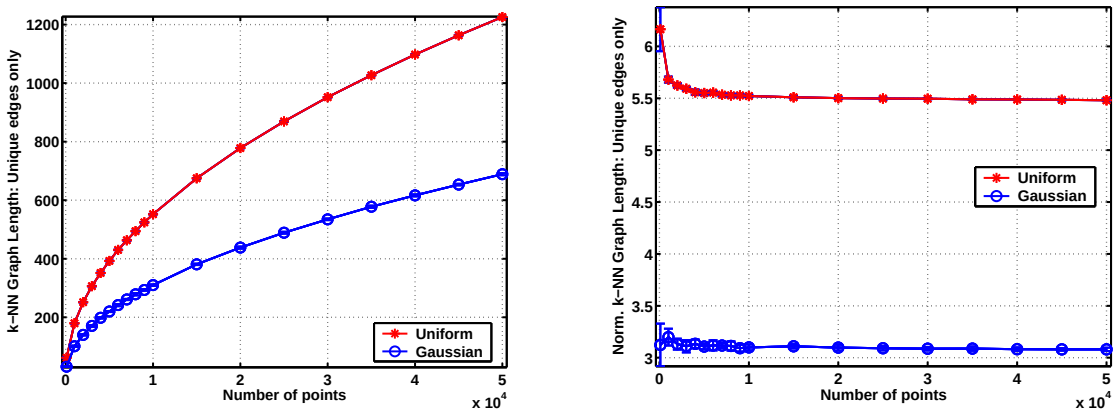
edge contributes only once to the total length. The resultant graph has the an identical appearance to the initial unpruned k-NN graph, when plotted on the page. However, the cumulative length of the edges in the graphs differ, and so does their β factor (Compare Figures 4.2.6 and 4.2.7). We call this special pruned k-NN graph, the "Single-Count k-NN graph".

4.3 Entropic Graph Estimate of Henze-Penrose Affinity

Friedman and Rafsky [42] presented a multivariate generalization of the Wald-Wolfowitz [127] runs statistic for the two sample problem. The Wald-Wolfowitz test statistic is used to decide between the following hypothesis based on a pair of samples $\mathcal{X}, O \in \mathbb{R}$ with densities f_x and f_o respectively:

$$H_0\colon f_x = f_o \tag{4.10}$$

$$H_1\colon f_x \neq f_o,$$



(a) (b)

Figure 4.2.7: (a) Mean Length functions L_n of Singe-Count kNN graph implemented with $\gamma = 1$ and (b) $L_n/\sqrt{n}$ as a function of n for uniform and normal distributed points.

The test statistic is applied to an i.i.d. random sample $\{\mathcal{X}_i\}_{i=1}^m, \{O_i\}_{i=1}^n$ from f_x and f_o. In the univariate Wald Wolfowitz test, the $n+m$ scalar observations $\{\mathcal{Z}_i\}_i = \{\mathcal{X}_i\}_i, \{O_i\}_i$ are ranked in ascending order. Each observation is then replaced by a class label $\mathcal{X}$ or O depending upon the sample to which it originally belonged, resulting in a rank ordered sequence. The Wald-Wolfowitz test statistic is the total number of runs (run-length) R_ℓ of $\mathcal{X}$'s or O's in the label sequence. As in run-length coding, R_ℓ, is the length of consecutive sequences of length ℓ of identical labels.

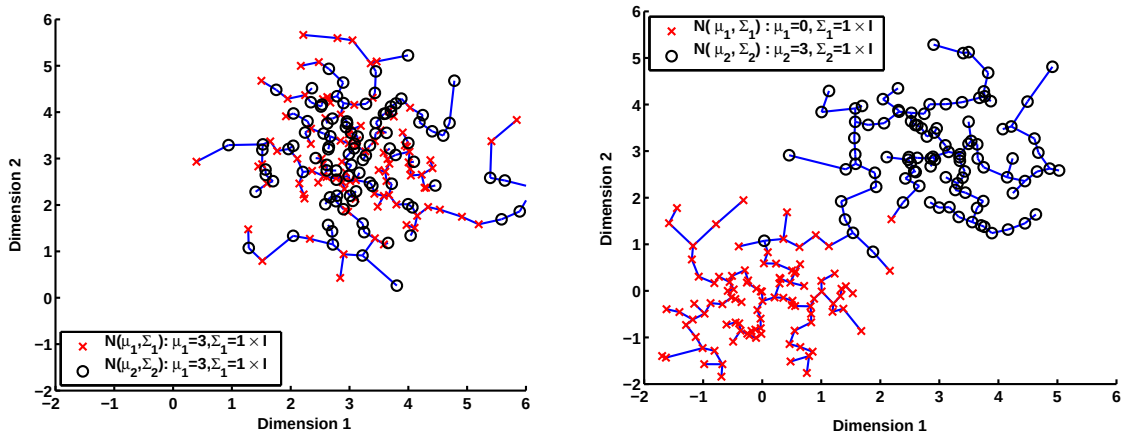
In Friedman and Rafsky's paper [42], the MST was used to obtain a multivariate generalization of the Wald-Wolfowitz test. This procedure is called the Friedman-Rafsky (FR) test and is similar to the MST for estimating the α-Jensen difference. It is constructed as follows:

1. construct the MST on the pooled multivariate sample points $\{\mathcal{X}_i\} \bigcup \{O_i\}$.

2. retain only those edges that connect an $\mathcal{X}$ labeled vertex to an O labeled vertex.

3. The FR test statistic, N, is defined as the number of edges retained.

The hypothesis H_1 is accepted for smaller values of the FR test statistic. As shown in [50], the FR test statistic N converges to the Henze-Penrose affinity (3.13) between the distributions f_x and f_o. The limit can be converted to the HP divergence by replacing N by the multivariate run length statistic $R_\ell^{FR} = n + m - 1 - N$.

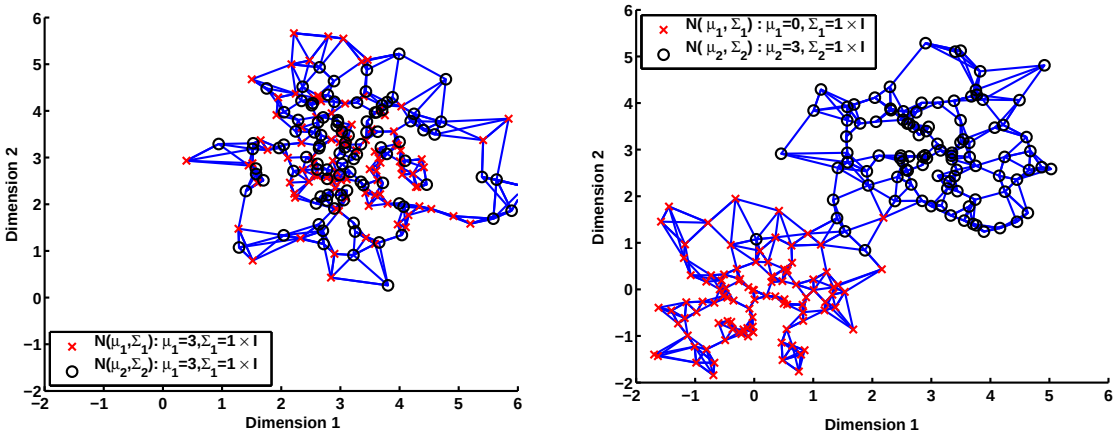


(a) MST $\mu_1 = \mu_2$ and $\Sigma_1 = \Sigma_2$

(b) MST $\mu_1 = \mu_2 - 3$ and $\Sigma_1 = \Sigma_2$

Figure 4.3.8: Illustration of MST for Gaussian case. Two bivariate normal distributions $\mathcal{N}(\mu_1, \Sigma_1)$ and $\mathcal{N}(\mu_1, \Sigma_1)$ are used. The 'x' labeled points are samples from $f_1(x) = \mathcal{N}(\mu_1, \Sigma_1)$, whereas the 'o' labeled points are samples from $f_2(o) = \mathcal{N}(\mu_2, \Sigma_2)$. (left) $\mu_1 = \mu_2$ and $\Sigma_1 = \Sigma_2$ and (right) $\mu_1 = \mu_2 - 3$ while $\Sigma_1 = \Sigma_2$.

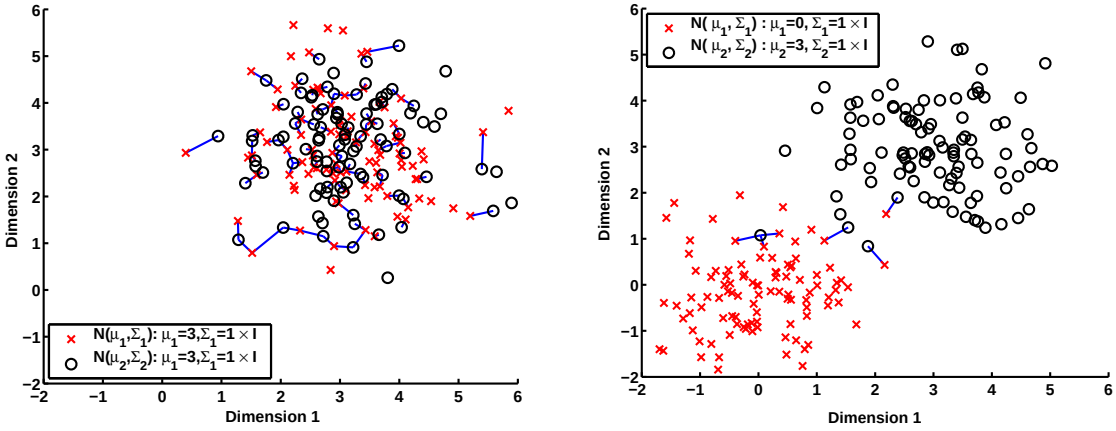
For illustration of these graph constructions we consider two bivariate normal distributions with density functions f_1 and f_2 parametrized by their mean and covariance $(\mu_1, \Sigma_1), (\mu_2, \Sigma_2)$. Graphs of the α-Jensen divergence calculated using MST (Figure 4.3.8), kNNG (Figure 4.3.9), and the Henze-Penrose affinity (Figure 4.3.10) are shown for the case where $\mu_1 = \mu_2, \Sigma_1 = \Sigma_2$. The 'x' labeled points are samples from $f_1(x) = \mathcal{N}(\mu_1, \Sigma_1)$, whereas the 'o' labeled points are samples from $f_2(o) = \mathcal{N}(\mu_2, \Sigma_2)$. μ_1 is then decreased so that $\mu_1 = \mu_2 - 3$. The resultant trends in the different divergence measures are seen in Figure 4.3.11. The α-Jensen divergence is minimized, as expected, when the two distributions are aligned and indistinguishable $(\mu_1 = \mu_2)$, whereas the Henze-Penrose



(a) kNN $\mu_1 = \mu_2$ and $\Sigma_1 = \Sigma_2$

(b) kNN $\mu_1 = \mu_2 + 3$ and $\Sigma_1 = \Sigma_2$

Figure 4.3.9: Illustration of kNN for Gaussian case. Two bivariate normal distributions $\mathcal{N}(\mu_1, \Sigma_1)$ and $\mathcal{N}(\mu_1, \Sigma_1)$ are used. The 'x' labeled points are samples from $f_1(x) = \mathcal{N}(\mu_1, \Sigma_1)$, whereas the 'o' labeled points are samples from $f_2(o) = \mathcal{N}(\mu_2, \Sigma_2)$. (left) $\mu_1 = \mu_2$ and $\Sigma_1 = \Sigma_2$ and (right) $\mu_1 = \mu_2 - 3$ while $\Sigma_1 = \Sigma_2$. $k = 4$

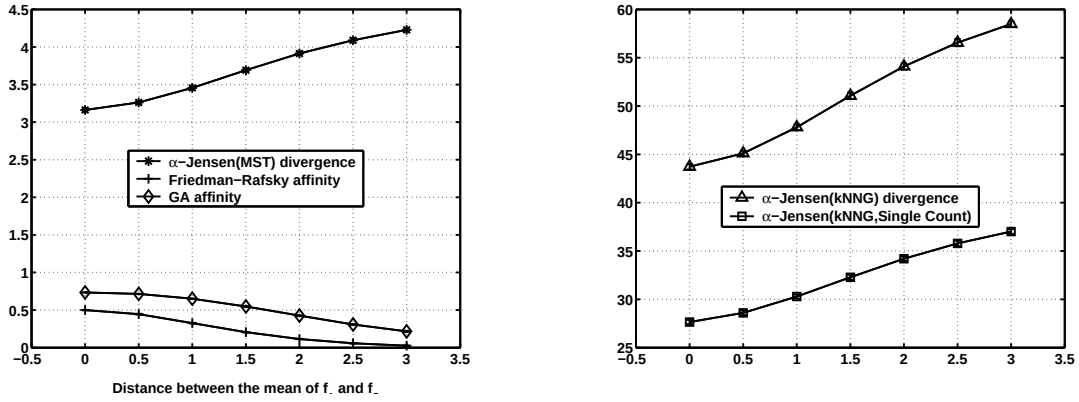


(a) Henze-Penrose $\mu_1 = \mu_2$ and $\Sigma_1 = \Sigma_2$

(b) Henze-Penrose $\mu_2 = \mu_1 + 3$ and $\Sigma_1 = \Sigma_2$

Figure 4.3.10: Illustration of Henze-Penrose affinity for Gaussian case. Two bivariate normal distributions $\mathcal{N}(\mu_1, \Sigma_1)$ and $\mathcal{N}(\mu_1, \Sigma_1)$ are used. The 'x' labeled points are samples from $f_1(x) = \mathcal{N}(\mu_1, \Sigma_1)$, whereas the 'o' labeled points are samples from $f_2(o) = \mathcal{N}(\mu_2, \Sigma_2)$. (left) $\mu_1 = \mu_2$ and $\Sigma_1 = \Sigma_2$ and (right) $\mu_1 = \mu_2 - 3$ while $\Sigma_1 = \Sigma_2$.

affinity is maximized.



(a) (b)

Figure 4.3.11: Illustration of divergence and affinity functions, as a function of the distance between the means of two bivariate normal distributions, f_1 and f_2. (a) αJensen divergence computed using MST, Friedman-Rafsky affinity and αGeometric-Arithmetic affinity. (b) αJensen divergence computed using kNNG and Single-count kNNG.

4.4 Entropic Graph Estimators of α-GA and α-MI

Assume for simplicity that the target and reference feature sets $O = \{o_i\}_i$ and $\mathcal{X} = \{x_i\}_i$ have the same cardinality $m = n$. Here i denotes the i^{th} pixel location in target and reference images. An entropic graph approximation to α-GA mean divergence (3.12) between target and reference is:

$$\widehat{\alpha D_{GA}} = \frac{1}{\alpha - 1} \log \frac{1}{2n} \sum_{i=1}^{2n} \min \left\{ \left(\frac{e_i(o)}{e_i(x)} \right)^{\gamma/2}, \left(\frac{e_i(x)}{e_i(o)} \right)^{\gamma/2} \right\}, \tag{4.11}$$

where $e_i(o)$ and $e_i(x)$ are the distances from a point $z_i \in \{\{o_i\}^i, \{x_i\}^i\} \in \mathbb{R}^d$ to its nearest neighbor in $\{O_i\}_i$ and $\{\mathcal{X}_i\}_i$, respectively. Here, as above $\alpha = (d - \gamma)/d$.

Likewise, an entropic graph approximation to the α-MI (3.5) between the target and the reference is:

$$\widehat{\alpha MI} = \frac{1}{\alpha - 1} \log \frac{1}{n^\alpha} \sum_{i=1}^{n} \left(\frac{e_i(o \times x)}{\sqrt{e_i(o) e_i(x)}} \right)^{2\gamma}, \tag{4.12}$$

where $e_i(o \times x)$ is the distance from the point $z_i = [o_i, x_i] \in \mathbb{R}^{2d}$ to its nearest neighbor in $\{Z_j\}_{j \neq i}$ and $e_i(o)$ $(e_i(x))$ is the distance from the point $o_i \in \mathbb{R}^d, (x_i \in \mathbb{R}^d)$ to its nearest neighbor in $\{O_j\}_{j \neq i} (\{\mathcal{X}_j\}_{j \neq i})$. See Figure 4.5.12 for illustration.

The estimators (4.11) and (4.12) can be derived from making a nearest neighbor approximation to the volume of the Voronoi cells constituting the kNN density estimator after plug-in to formulas (3.12) and (3.5), respectively. The details are given below. The theoretical convergence properties of these estimators are at present unknown.

Natural generalizations of (4.11) and (4.12) to multiple (> 2) images exist. The computational complexity of the α-MI estimator (4.12) grows only linearly in the number of images to be registered while that of the α-GA estimator (4.11) grows as linear log linear. Therefore, there is a significant complexity advantage to implementing α-MI via (4.12) for simultaneous registration of a large number of images.

Here we give a derivation of the entropic graph estimators of α-GA (4.11) and α-MI (4.12) estimators. The derivations are given for equal numbers m and n of features from the two images but are easily generalized to unequal m, n. The derivation is based on a heuristic and thus the convergence properties are, at present, unknown.

First consider estimating $\alpha D_{GA}(f, g) = (\alpha - 1)^{-1} \log I_{GA}(f, g)$, where $I_{GA}(f, g)$ is the integral in (3.12), by $\widehat{\alpha D_{GA}} = (\alpha - 1)^{-1} \log \widehat{I_{GA}}$ where:

$$\widehat{I_{GA}} = \frac{1}{2n} \sum_{i=1}^{2n} \left(\frac{\hat{f}^p(z_i) \hat{g}^q(z_i)}{\hat{h}(z_i)} \right)^{1-\alpha}. \tag{4.13}$$

Here $\hat{h}(z)$ is an estimate of the common pdf $pf(z) + qg(z)$ of the i.i.d. pooled unordered sample $\{Z_i\}_{i=1}^{2n} = \{O_i, \mathcal{X}_i\}_{i=1}^{n}$ and $\hat{f}$, $\hat{g}$ are estimates of the common densities f, g of the i.i.d. samples $\{O_i\}_{i=1}^{n}$ and $\{\mathcal{X}_i\}_{i=1}^{n}$, respectively. We assume that the support set of f, g, h is contained in a bounded region $\mathcal{S}$ of $\mathbb{R}^d$. If $\hat{f}$, $\hat{g}$, $\hat{h}$ are consistent, i.e they converge (a.s.)

as $n \to \infty$ to f, g, h, then by the strong law of large numbers $\widehat{I_{GA}}$ converges (a.s) to

$$\begin{aligned} E[\widehat{I}_{GA}] &= E\left[\left(\frac{f^p(z_i)g^q(z_i)}{h(z_i)}\right)^{1-\alpha}\right] \\ &= \int_{\mathcal{S}}\left(\frac{f^p(z)g^q(z)}{h(z)}\right)^{1-\alpha} h(z)dz, \end{aligned} \tag{4.14}$$

Taking the log of expression (4.14) and dividing by $\alpha - 1$, we obtain $\alpha D_{GA}(f, g)$ in (3.12) so that $\widehat{\alpha D_{GA}}$ is asymptotically unbiased and its variance goes to zero.

Next divide the samples $\{Z_i\}_{i=1}^{2n}$ into two disjoint sets: training samples Z_{train} and test samples Z_{test}. Using the training sample construct the Voronoi partition density estimators

$$\begin{aligned} \hat{h}(z) &= \frac{\mu(\Pi_z(z))}{\lambda(\Pi_z(z))} \\ \hat{f}(z) &= \frac{\mu(\Pi_o(z))}{\lambda(\Pi_o(z))} \\ \hat{g}(z) &= \frac{\mu(\Pi_x(z))}{\lambda(\Pi_x(z))}, \end{aligned} \tag{4.15}$$

where $\Pi_Z(z)$, $\Pi_O(z)$, $\Pi_{\mathcal{X}}(z)$ are the cells of the Voronoi partition of $\mathcal{S} \in \mathbb{R}^d$ containing the point $z \in \mathbb{R}^d$ and constructed from training samples $Z_{train} \equiv \{O_{train}, \mathcal{X}_{train}\}$, O_{train} and $\mathcal{X}_{train}$ respectively using K-means or other algorithm. Here μ and λ are the (normalized) counting measure and Lebesgue measure respectively, i.e $\mu(\Pi)$ is the number of points in the set Π divided by the total number of points and $\lambda(\Pi)$ is the volume of the set Π. Let $\{K_z, K_o, K_x\}$ be the number of cells in the partitions $\{\Pi_z, \Pi_o, \Pi_x\}$ respectively and let n_{train} be the number of training samples. The Voronoi partition density estimators are asymptotically consistent as $k, n_{train} \to \infty$ and $k/n_{train} \to 0$, for $k \in \{K_z, K_o, K_x\}$ [100].

Therefore, under these conditions and defining $\tilde{Z}_i = Z_{test}(i)$,

$$\widehat{\alpha D_{GA}} = \frac{1}{\alpha - 1}\log\left(\frac{1}{n_{test}}\sum_{i=1}^{n_{test}}\left(\frac{\hat{f}^p(\tilde{z}_i)\hat{g}^q(\tilde{z}_i)}{\hat{h}(\tilde{z}_i)}\right)^{1-\alpha}\right) \tag{4.16}$$

is an asymptotically consistent estimator as $k, n_{train}n_{test} \to \infty$ and $k/n_{train} \to 0$

Next consider the following plug-in estimator of α-MI: $\widehat{\alpha MI} = (\alpha-1)\log \widehat{I_{MI}}$, where

$$\widehat{I_{MI}} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\hat{f}_o(o_i)\hat{f}_x(x_i)}{\hat{f}_{ox}(o_i,x_i)}\right)^{1-\alpha}, \tag{4.17}$$

and $\hat{f}_{ox}$ is an estimate of the joint density of the $2d$ dimensional vector $(O_i, \mathcal{X}_i) \in \mathbb{R}^{2d}$. $\hat{f}_o$ and $\hat{f}_x$ are estimates of the density of O_i and $\mathcal{X}_i$, respectively. Again, if $\hat{f}_o$, $\hat{f}_x$ and $\hat{f}_{ox}$ are consistent then it is easily shown that $\widehat{I_{MI}}$ converges to the integral in the expression (3.5) for α-MI:

$$\int_{\mathcal{S}\times\mathcal{S}}\int\left(\frac{\hat{f}_o(u)\hat{f}_x(v)}{\hat{f}_{ox}(u,v)}\right)^{1-\alpha} f_{ox}(u,v)dudv, \tag{4.18}$$

where $\mathcal{S}$ is a bounded set containing the support of densities f_o and f_x. Similarly, separating $\{(O_i,\mathcal{X}_i)\}_{i=1}^{n}$ into training and test samples, we obtain an asymptotically consistent estimator:

$$\widehat{\alpha MI} = \frac{1}{\alpha-1}\log\left(\frac{1}{n_{test}}\sum_{i=1}^{n_{test}}\left(\frac{\hat{f}_o(\tilde{o}_i)\hat{f}_x(\tilde{x}_i)}{\hat{f}_{ox}(\tilde{o}_i,\tilde{x}_i)}\right)^{1-\alpha}\right) \tag{4.19}$$

The entropic graph estimators (4.11) and (4.12) are obtained by specializing to the case $n_{train} = 0$, in which case $\mu(\Pi_Z(z)) = (2n)^{-1}$, $\mu(\Pi_O(z)) = \mu(\Pi_{\mathcal{X}}(z)) = \mu(\Pi_{O\times\mathcal{X}}(z)) = n^{-1}$, and using the Voronoi cell volume approximations

$$\lambda(\Pi_Z(z_i)) \asymp e_i^d(z) = \min\{e_i^d(o), e_i^d(x)\} \tag{4.20}$$

$$\lambda(\Pi_{\mathcal{X}}(z_i)) \asymp e_i^d(x) \tag{4.21}$$

$$\lambda(\Pi_Y(z_i)) \asymp e_i^d(o) \tag{4.22}$$

$$\lambda(\Pi_{O\times\mathcal{X}}(z_i)) \asymp e_i^{2d}(o\times x) \tag{4.23}$$

where $\asymp$ denotes "proportional to" and $e_i(o)$, $e_i(x)$, $e_i(o\times x)$ are the NN distances defined in Section 4.2. In Equation 4.23 approximation was involved to represent the volume of a Voronoi cell by that of the minimum volume cell in O and $\mathcal{X}$, respectively.

4.4.1 Implementation Issue

The stable computation of the α-MI estimator (Equation 4.12) requires that $e_i(o)$ and $e_i(x)$ be non-zero whenever $e_i(o \times x)$ is non-zero (Figure 4.5.12). For continuously distributed features $\{O_i\}$ and $\{\mathcal{X}_i\}$ the probability of stable computation is one, since the probability that any two feature components be exactly equal is zero. However, for practical applications where the feature space is quantized to finite precision arithmetic, the probability of stable computation is strictly less than one. In fact, it can be shown that it rapidly goes to zero as the number of feature realizations gets large.

A remedy for this is randomization. To avoid zero values of $e_i(o)$ and $e_i(x)$, a small amount of uniform noise may be added to the feature coefficient. This randomization disperses points uniformly in an area around their discretized value. This process is consistent with the assumption that local distribution of continuously valued feature vectors is uniform around their discretized values. In simulations with discretized 8-bit pixel intensity features, univariate uniform noise with a variance $\sigma^2 = 0.02$ was added to each pixel intensity. This ensured that no two intensities were exactly the same and thus enabling stable computation of αMI.

4.5 A non-linear correlation measure

The simple form of Equation 4.12 is suggestive of a non-linear correlation measure between the features $\{O_i\}$ and $\{\mathcal{X}_i\}$ that eliminates the implementation issue discussed above. Indeed, if "e_i" in Equation 4.12 is redefined as the statistical expectation "E", then the α-MI estimator takes the appearance of a linear correlation coefficient between $\{O_i\}$ and $\{\mathcal{X}_i\}$. However, as explained above, the ratio $e_i(o \times x)/\sqrt{e_i(o)e_i(x)}$ is not bounded between 0 and 1, rather it can take values that are arbitrarily large. The following modification of Equation 4.12 can be used to ensure that the non-linear correlation measure

lie between 0 and 1. This new measure is called the non-linear correlation coefficient (NLCC).

Let $e_i(o \times x)$ be the distance from i-th feature pair (o_i, e_i) to its nearest neighbor as before. Instead of $e_i(o)$ and $e_i(x)$ being the coordinate-wise nearest neighbor distances along the feature coordinate axes $\mathcal{X}$ and O (See Figure 4.5.12) we define $\tilde{e}_i(o)$ and $\tilde{e}_i(x)$ the associated nearest neighbor distances in the plane (see Figure 4.5.13). The quantity $\tilde{e}_i(o \times x)/\sqrt{\tilde{e}_i(o)\tilde{e}_i(x)}$ is now bounded between 0 and 1. In particular, it is equal to one when the nearest neighbor to (o_i, x_i) is also the coordinate-wise nearest neighbor to (o_i, x_i) along the coordinate axes O and $\mathcal{X}$.

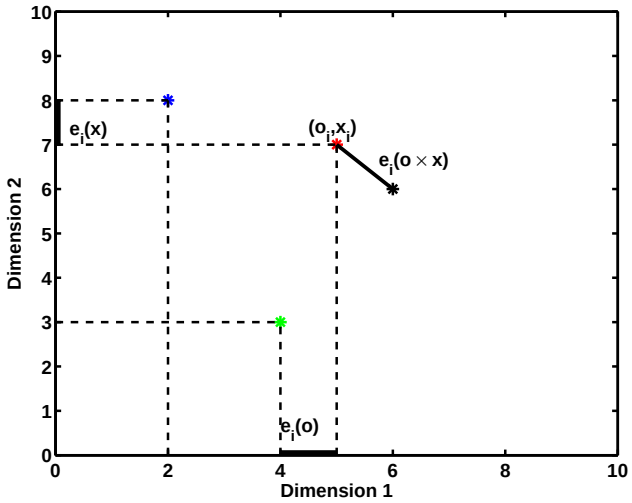


Figure 4.5.12: Illustration of the distances $e_i(o \times x)$, $e_i(o)$ and $e_i(x)$ used in the α-MI estimator (Equation 4.12)

In particular the quantity

$$\hat{\rho} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\tilde{e}_i(o \times x)}{\sqrt{\tilde{e}_i(o)\tilde{e}_i(x)}}\right) \tag{4.24}$$

is equal to one when the nearest neighbor graph is monotone (increasing or decreasing) piecewise linear curve in the plane 4.5.14. Thus if the features are realizations of the random vector $(O, \mathcal{X})$ which obeys the monotone model:

$$\Theta = g(\mathcal{X}), \tag{4.25}$$

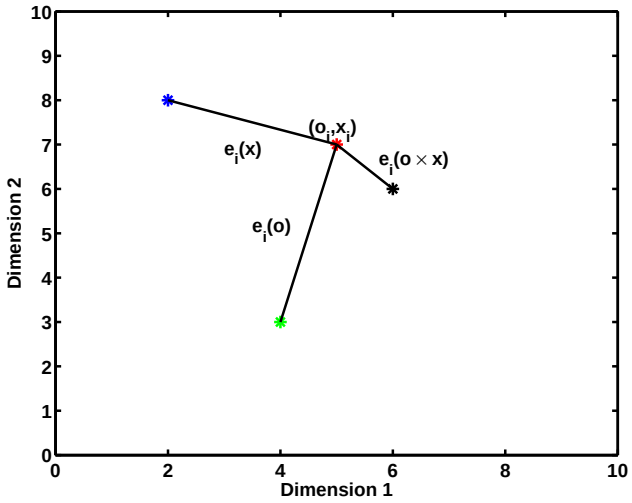


Figure 4.5.13: Illustration of modified distances $e_i(x)$ and $e_i(o)$ used to stabilize the estimator (Equation 4.12), defining the non-linear correlation coefficient (NLCC)

where $g(\cdot)$ is a monotonic increasing function, the NLCC $\hat{\rho}$ will equal 1 with probability one. This motivates the use of ρ as a measure of information between Θ and $\mathcal{X}$. Unfortunately, if the actual model is

$$\Theta = g(\mathcal{X}) + w \tag{4.26}$$

where w is additive noise, $\hat{\rho}$ will converge to zero as $n \to \infty$ for any continuous random variable w. It can be shown that the rate of convergence in this case is $n^{\frac{-\gamma}{2d}}$. This motivates the modification of the NLCC to:

$$\hat{\rho}_{NLCC} = \frac{1}{n^{1-\gamma/2d}} \sum_{i=1}^{n} \left(\frac{\tilde{e}_i(o \times x)}{\sqrt{\tilde{e}_i(o)\tilde{e}_i(x)}} \right). \tag{4.27}$$

This modified correlation now takes values between 0 and ∞. A normalized version can be defined as:

$$\hat{\rho} = \frac{\hat{\rho}_{NLCC}}{1 + \hat{\rho}_{NLCC}} \tag{4.28}$$

that is between zero and one.

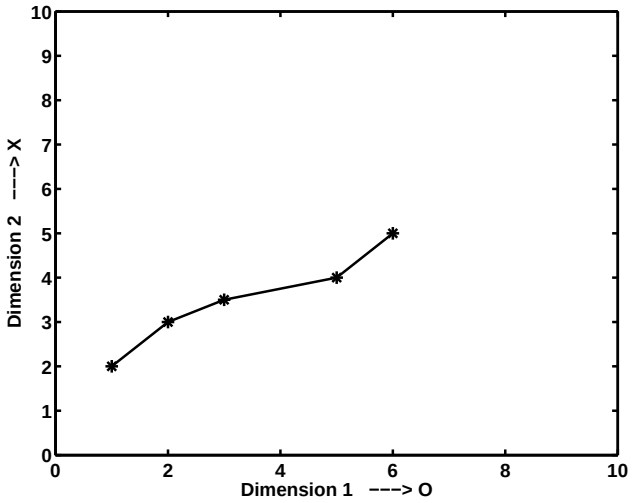


Figure 4.5.14: The Nearest Neighbor Graph over the realizations $\{(o_i \times x_i)\}_{i=1}^{N}$ of the paired features describes a monotone function in the plane. For this case, the NLCC $\hat{\rho} = 1$

We illustrate the NLCC by comparing it to the linear correlation coefficient 4.29 for two simple models. The linear correlation coefficient is defined as:

$$\hat{\rho}_{CC} = \frac{\frac{1}{n}\sum_{i=1}^{n}(o_i - \bar{o})(x_i - \bar{x})}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(o_i - \bar{o})^2 \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}} \tag{4.29}$$

where $\bar{o} = 1/n \sum_{i=1}^{n} o_i$ and $\bar{x} = 1/n \sum_{i=1}^{n} x_i$ are sample means.

4.5.1 Numerical experiments with NLCC

Consider the linear model $\Theta = a\mathcal{X} + w$, where $a^2 = \rho_{CC}^2/(\rho_{CC}^2 + 1)$. Figure 4.5.15 shows a plot of the linear (Equation 4.29) and nonlinear (Equation 4.27) correlation coefficients, $\hat{\rho}_{CC}$ and $\hat{\rho}_{NLCC}$ for this model as functions of the number of points N for various values of a. As a increases, the linear correlation increases but does not reach one due to the presence of additive noise w. In the limit as $N \to \infty$ the non-linear correlation coefficient converges to a constant.

Now consider the nonlinear model given by $\Theta = ag(\mathcal{X}) + w$; $g(\mathcal{X}) = b\mathcal{X}^3$. As shown in Figure 4.5.16, the linear correlation coefficient remains unchanged at the value

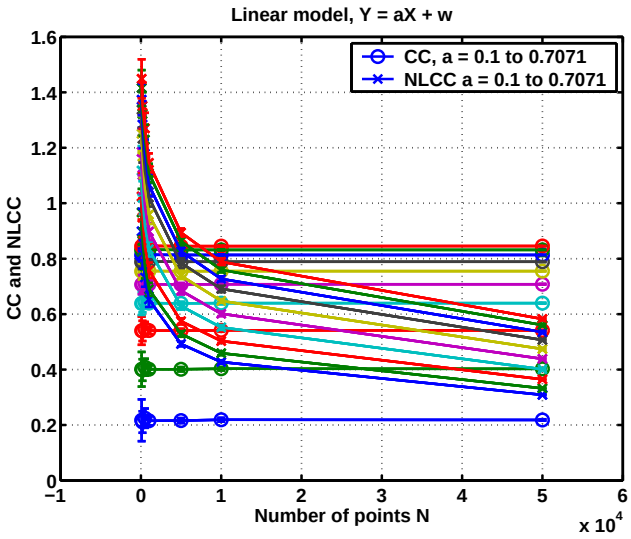


Figure 4.5.15: Comparison of Linear and non-linear correlation coefficient for a linear model

corresponding to the relation between Θ and $\mathcal{X}$. The non-linear correlation, however increases with a, showing that it responds to changes in the non-linear relation between Θ and $\mathcal{X}$.

Figure 4.5.17 confirms these findings. It illustrates the relation between the linear and non-linear correlation coefficients for both linear and non-linear models. The values are plotted for $N = 50000$ and a increases from 0.1 to 0.7071.

In conclusion this chapter laid out a direct entropy estimation method using graphs whose lengths converge asymptotically to the entropy of the underlying distributions. Such direct entropy estimations methods can be readily extended to higher dimensions without loss of consistency. MST and kNN graphs are used to estimate α-entropy and the αJensen difference between marginal distributions of feature densities. The MST is also used to estimate the Friedman-Rafsky statistic and the related Henze-Penrose affinity. Finally, entropic graph estimators for αMI and αGA and NLCC were presented.

As discussed earlier, this thesis focuses on feature-level representation of images. Unlike previous entropic matching methods however, features are not limited to scalar pixel

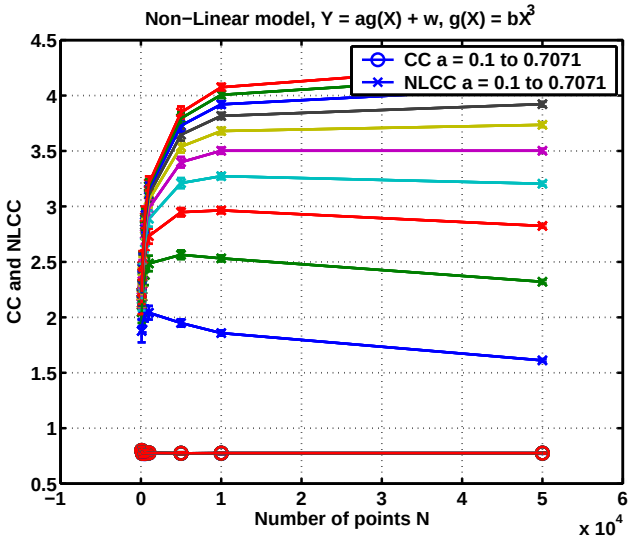


Figure 4.5.16: Comparison of Linear and non-linear correlation coefficient for a nonlinear model

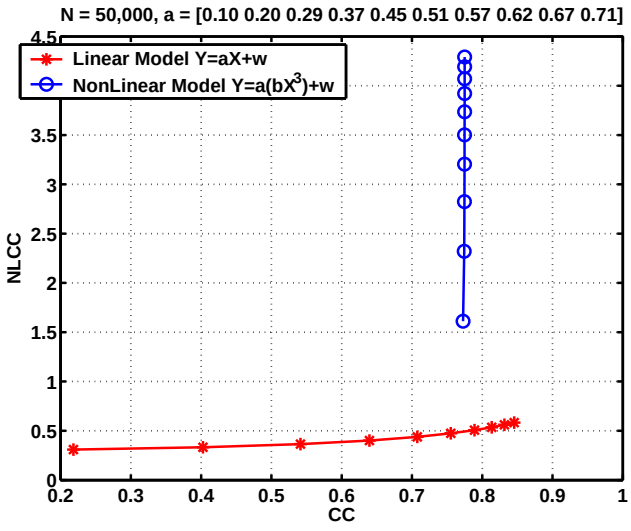


Figure 4.5.17: Plot of CC v/s NLCC for N = 50000 and a = 0.1 to 0.7071

intensity values. Higher dimensional features in continuum are used for image representation and registration through a matching process based on the entropic graph estimators discussed in this chapter.

CHAPTER V

Feature-based Matching

While scalar single pixel intensity level is the most widely used feature for MI registration, it is not the only possible feature. As pointed out by Leventon and Grimson [76], single pixel MI does not take into account joint spatial behavior of the coincidences and this can cause poor registration, especially in multi-modality situations. Alternative scalar valued features [17] and vector valued features [97, 110] have been investigated for mutual information based image registration. We will focus on local basis projection feature vectors which generalize pixel intensity levels. Mutual information based registration methodology using single pixel features is illustrated in Figure 3.3.1

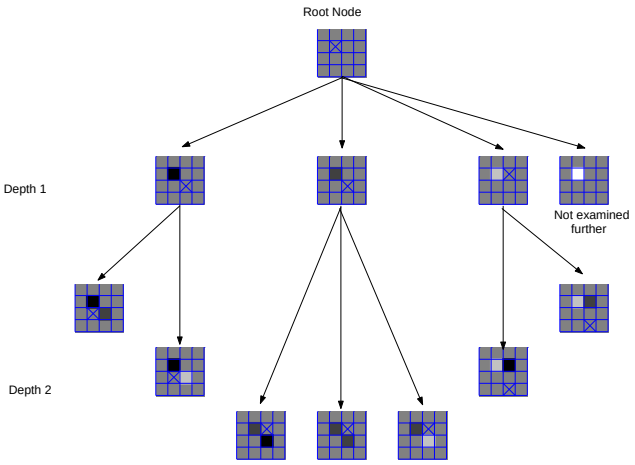
5.1 Local Tag Features

Tag features were introduced by Amit and Geman [4] and used for shape recognition. A set of primitive local features, called tags, are selected which provide a coarse description of the topography of the intensity surface in the vicinity of a pixel. Local image configurations, e.g. 8×8 pixel neighborhoods, are captured by coding each pixel with labels derived from the tags. For gray scale images, the number of different tag types can be extremely large. For example, if the image intensities are quantized to an 8-bit plane then there would exist $(256)^{64}$ different 8×8 tag types. Therefore, methods for prun-

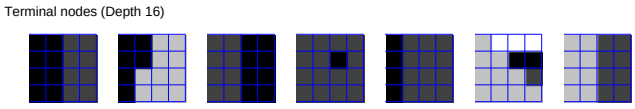
ing the tag types are essential for practical implementation. Randomized feature selection and adaptive thresholding are methods of pruning which were described by Geman and Koloydenko [44] and which we adapted [96, 97] to the UL image registration application described below.

Geman's adaptive quantization scheme is sensitive to local contrast in Ultrasound images. The quantized value assigned to a pixel within an 8×8 neighborhood depends on the gray values of its neighbors. Consider an 8×8 pixel neighborhood arbitrarily picked from an Ultrasound image. Let δ be a positive granularity parameter. The darkest pixel(s) are assigned quantized value 0, the next brightest pixel(s) are assigned 0 if the difference is less than δ and value 1 otherwise, the next brightest pixel(s) are assigned 2 if the difference is less than δ, and so on. Using this scheme on our ultrasound breast image database, tags associated with relatively uniform background areas (dark or bright) are eliminated. Tags with small spatial variances are classified as speckle and are also eliminated. Such tags are irrelevant to image matching and pruning achieves a reduction in tag types by almost 75%. From the remaining tag types, randomized selection is used to pick a set of 256 unique tag types as explained below.

Randomized selection using Geman's scheme [4] entails building a tree structure that is used to build histograms of tags. Figure 5.1.2(a) illustrates the tree based classification procedure for a 4×4 pixel tag. At the root node, the value of the pixel at (a randomly picked) position (2,2) in the tag is examined. The tag is classified amongst one of four different branches of the tree. A tree structure evolves when subsequent queries are used to further split the tag tree. Each node of the tree thus represents a query at a position in the tag. The tags at the bottom of this feature tree are identified as leaf nodes or bins. These nodes have been queried at all 16 positions and hence further splitting of the tree at the node is not possible. A section of the tree at the leaf depth is shown in Figure 5.1.2(b).



(a)



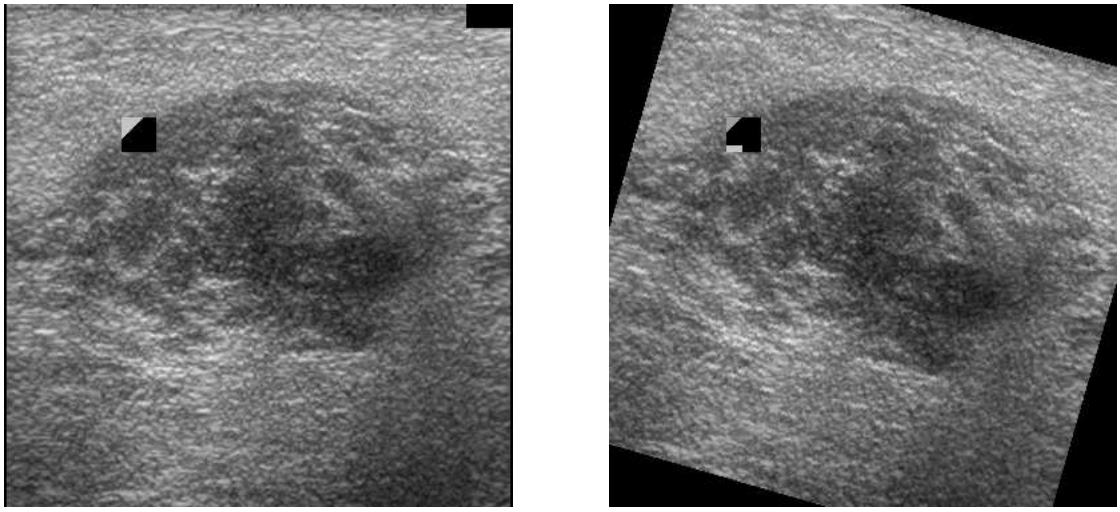
(b) b

Figure 5.1.1: (a) Feature tree structure used to pick tags for registration. (b) Feature tree at leaf level shows examples of tag types used for registration

To build a histogram in tag space an image is 'dropped' block-by-block, down the feature tree. To populate the bins in the feature tree, an image is subdivided into 4×4 blocks. The block size is identical to the size of the tags used. Blocks are also discretized in a manner identical to the discretization used for tags i.e. using adaptive thresholding. The image blocks are queried at specific positions determined by the feature tree described above. Based on their responses, the blocks traverse a path down the tree until they reach a bin at the leaf node. Several image blocks may accumulate in the bins at the leaf depth in the tree. These bins thus form a histogram of tag types and the histogram serves as an estimate of the tag pdf of the image. Some image blocks will obviously be discarded if the underlying tags have been discarded. This ensures that speckle or other structure deemed irrelevant to registration do not participate in the image registration process.

To illustrate we show in Fig. 5.1.2 tag features at a given pixel location for two US breast images in the same 2D slice but at two rotation angles. Coincidences of tag types are calculated by counting joint occurrences of feature types at identical spatial locations in the two images. The (amplified) tag pattern in the image on the left captures the edge of the tumor. A similar tag type will be observed in the secondary image on the right if it nearly aligned. These tags capture the local intensity pattern in the neighborhood of the pixel. The advantage of tags for matching US breast images is that they can more easily discriminate between speckle and tissue echos than can single pixel intensity values.

Basis projection features are extracted from an image by projecting local sub-images onto a basis of linearly independent sub-images of the same size. Such an approach is widely adopted in image matching applications, in particular with DCT or more general 2D wavelet bases [123, 31, 112, 83, 32]. Others have extracted a basis set adapted to image databases using principal components (PCA) or independent components analysis (ICA) [77, 61].



(a) (b)

Figure 5.1.2: Local tags features applied to image registration. Each pixel is labeled by a 8×8 tag type extracted using Geman's [44, 4] adaptive thresholding technique. Occurrences and coincidences of tag labels can be mapped to a coincidence histogram like Fig. 3.3.1

5.2 ICA Basis Projection Features

An ICA basis is especially well suited for registration purposes since it aims to obtain vector features which have statistically independent elements that can facilitate estimation of α-MI and other entropic measures. Specifically, in ICA an optimal basis is found which decomposes the image X_i into a small number of approximately statistically independent components (sub-images) $\{S_j\}$:

$$X_i = \sum_{j=1}^{p} a_{ij} S_j. \tag{5.1}$$

Basis elements $\{S_j\}$ are selected from an over-complete linearly dependent basis using randomized selection over the database. For image i the feature vectors Z_i are defined as the coefficients $\{a_{ij}\}$ in (5.1) obtained by projecting the image onto the basis (See Figure 5.2.3).

Figure 5.2.4 serves to illustrate the ICA basis selected for the MRI image database.

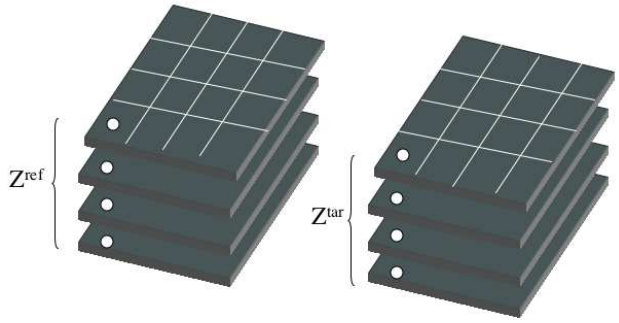


Figure 5.2.3: Subimages are projected onto the basis and the resultant coefficients $\{Z_{ref}\}$ and $\{Z_{tar}\}$ are used as features for registration.

ICA was implemented using Hyvarinen and Oja's [61] `FastICA` code (available from [60]) which uses a fixed-point algorithm to iteratively compute the maximum likelihood estimate of the basis elements in the ICA data model (5.1). An ICA basis of 8×8 sub-images was generated by randomized selection on the image volumes thus yielding a 64 dimensional feature set. Figure 5.2.4 shows a set of 64 16×16 basis vectors which were estimated from the 100,000 randomly selected 16×16 training sub-images picked from 5 consecutive image slices each of two MRI volume scans of the brain, one of the scans was T1 weighted whereas the other was T2 weighted. Given this ICA basis and a pair of to-be-registered $M \times N$ images, coefficient vectors were extracted by projecting each non-overlapping 16×16 neighborhood in the images onto the basis set. Asymptotic convergence of the entropic graph to the αentropy is under i.i.d condition for the underlying feature vectors. Non-overlapping blocks are chosen over overlapping blocks in an attempt to satisfy this constraint. For the 64 dimensional ICA basis shown in Figure 5.2.4 this yielded a set of MN vectors in a 64 dimensional vector space which will be used to define features. An ICA basis of 8×8 sub-images was also generated by an identical randomized selection on breast UL image volumes thus yielding a 64 dimensional feature set (Figure 5.2.5).

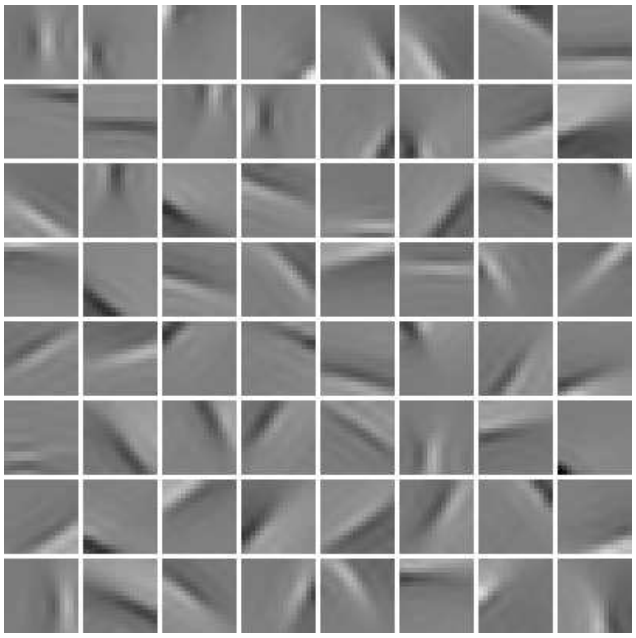


Figure 5.2.4: 16×16 ICA basis set obtained from training on randomly selected 16×16 blocks in 10 T1 and T2 time weighted MRI images. Only 64 of the 256 possible bases are shown. Features extracted from an image are the 64-dimensional vectors obtained by projecting 16×16 sub-images of the image on the ICA basis.

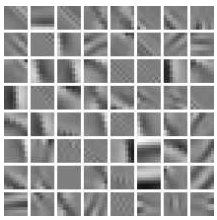


Figure 5.2.5: 8×8 ICA basis set obtained from training on randomly selected 8×8 blocks in 10 Ultrasound image volumes. Features extracted from an image are the 64-dimensional vectors obtained by projecting 8×8 sub-images of the image on the ICA basis.

5.2.1 Discrete vs. Continuous Features

While adaptive thresholding yields tag features that are discrete valued, ICA and other basis projection features are continuous valued. The potentially high dimension of the basis projection feature space makes estimation of the divergence measure problematic. A brute force method would be to discretize the vector of projection coefficients, e.g. using vector quantization [79] as in Ma's thesis [82], and generate histograms over the Voronoi cells. These histograms could then be used in the formula for the divergence to yield plug-in estimators of these quantities (Chapter IV). This presents difficulties for image matching applications since the reference and secondary images must both use the same cell partitioning in order to maintain consistency of bin indexes. For high dimensional feature space this brute force method also suffers from large bias unless one uses an impractically large number of cells. An alternative that can be applied to directly estimating divergence is through the use of entropic graphs as discussed in Chapter IV.

5.3 Multiresolution Wavelet basis features

As contrasted to ICA and tag features which require a training set of images to determine the feature mapping, multiresolution wavelet features can be extracted without any training sample required. Coarse-to-fine hierarchical wavelet basis functions describe a linear synthesis model for the image. The coarser basis functions have larger support than the finer basis; together they incorporate global and local spatial frequency information in the image. The multiresolution properties of the wavelet basis offer an alternative to the ICA basis, which is restricted to a single window size. Wavelet bases are commonly used to generate features for image registration [129, 117, 64] and are briefly reviewed here.

A multiresolution analysis of the space of Lebesgue measurable functions, $L^2(\mathbb{R})$, is a set of closed, nested subspaces V_j, $j \in \mathbb{Z}$. A wavelet expansion uses translations and

dilations of one fixed function, the wavelet $\psi \in L^2(\mathbb{R})$. ψ is a wavelet if the collection of functions $\{\psi(x-l)|l \in Z\}$ is a Riesz basis of V_0 and its orthogonal complement W_0. The continuous wavelet transform of a function $f(x) \in L^2(R)$ is given by:

$$\mathcal{W}f(a,b) =< f, \psi_{a,b} >; \psi_{a,b} = \frac{1}{\sqrt{|a|}}\psi(\frac{x-b}{a}), \tag{5.2}$$

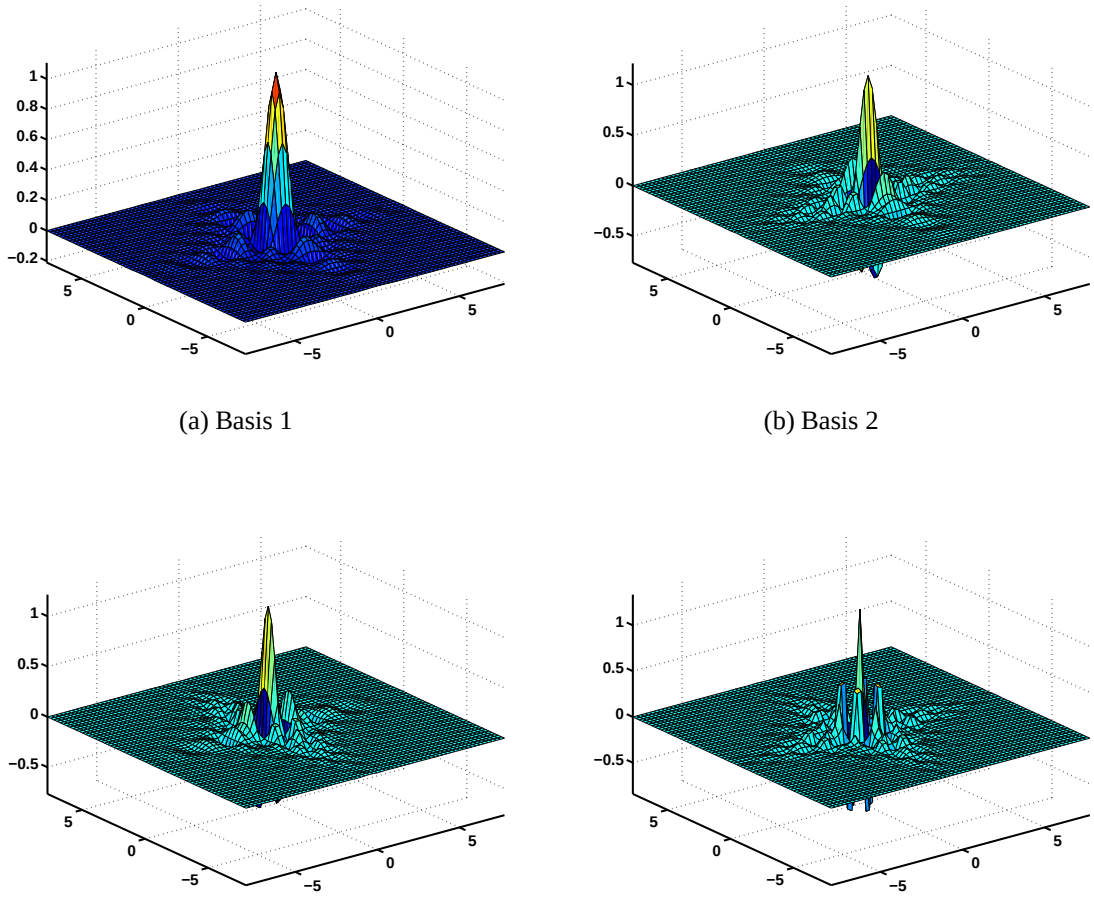
where $a, b \in \mathbb{R}, a \neq 0$.

For discrete wavelets, the dilation and translation parameters, b and a, are restricted to a discrete set, $a = 2^j, b = k$ where j and k are integers. The dyadic discrete wavelet transform is then given as:

$$\mathcal{W}f(j,k) =< f, \psi_{j,k} > \psi_{j,k} = 2^{-j/2}\psi(2^{-j}x - k) \tag{5.3}$$

where $j, k \in \mathbb{Z}$. Thus the wavelet coefficient of f at scale j and translation k is the inner product of f with the appropriate basis vector at scale j and translation k. The 2D discrete wavelet analysis is obtained by a tensor product of two multiresolution analysis of $L^2(\mathbb{R})$. At each scale, j, we have one scaling function subspace and three wavelet subspaces. The discrete wavelet transform of an image is the projection of the image onto the scaling function V_0 subspaces and the wavelet subspaces W_0. The corresponding coefficients are called the approximation and detail coefficients, implying the low and high pass characteristics of the basis filters. The process of projecting the image onto the successively coarser spaces continues to achieve the approximation desired. The difference information sensitive to vertical, horizontal and diagonal edges are treated as the three dimensions of each feature vector. Several members of the discrete Meyer basis used in this work are plotted below in Figure (5.3.6). They correspond to the Low-Low, Low-High, High-Low and High-High wavelet sub-bands respectively. The registration features are the wavelet coefficients obtained by projecting the images onto the Meyer wavelets. For the case of satellite images shown in Figure 3.3.3, the resultant coefficients seen in Figure 5.3.7 are

arranged in a 2D matrix format to resemble the images that were projected on the basis. Four wavelet sub-bands are used, each generating a quarter of the coefficient matrix seen in the figure. Use of wavelet features in registration in indexing has previously been limited to low dimensional basis representation for MI [64] and other divergence criteria or for correlation registration criteria. The adoption of entropic graph divergence approximation methods allows us to extend the use of wavelets to much higher dimensions.



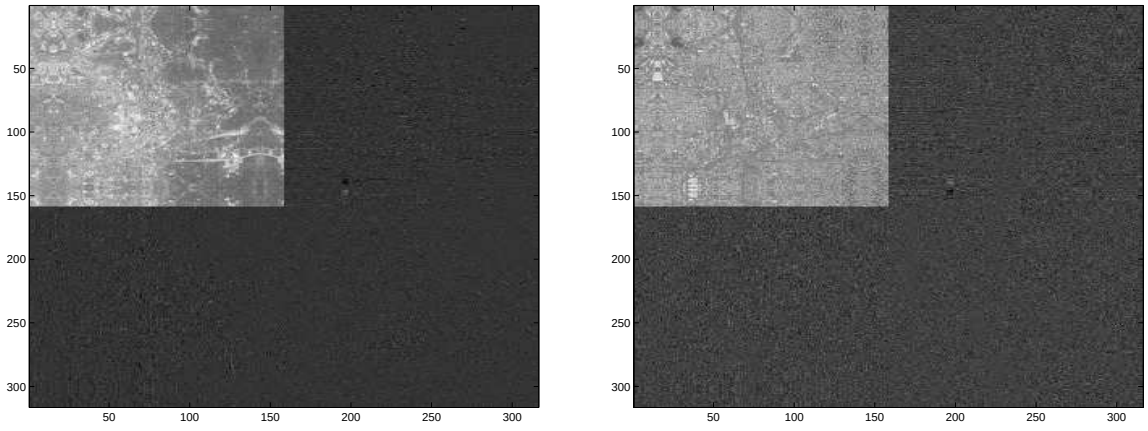
(a) Basis 1

(b) Basis 2

(c) Basis 3

(d) Basis 4

Figure 5.3.6: Wavelet decomposition: Discrete Meyer Wavelet Basis. (a) Scale subspace and (b-d) three wavelet subspaces at level 1 decomposition.



(a) (b)

Figure 5.3.7: Wavelet coefficients obtained by projecting visible and thermal satellite images from Figure 3.3.3(a) and (b) onto each of the Meyer wavelet basis shown in Figure 5.3.6.

5.3.1 Significance of spatial coordinates in feature definition

When a sub-block is extracted from position $\{x_i, y_i\}$ in an image, the spatial coordinates $\{x_i, y_i\}$ are appended to the corresponding ICA or wavelet feature coefficients. This ensures that invariance of divergence to shifting and existence of repeated pattern in the image. As stated before, some of the measures of divergence like αJensen difference, αGA and Henze-Penrose rely on marginal densities and do not have spatial discrimination capability. Unless the feature definition itself is very high dimensional, e.g. 8×8 ICA decomposition which is 64 dimensional, appending spatial coordinates is required. In practical image registration situations, features, like ICA in very high-dimensional spaces, tend to contain spatial tagging information and spatial coordinates may not provide extra benefit. However, in cases where repeated patterns exist in the image, spatial coordinates must be appended to these high-dimensional feature definitions. For measures such as αMI and NLCC, feature co-occurrences in the joint space provide the spatial information and do not need additional tagging with spatial coordinate information.

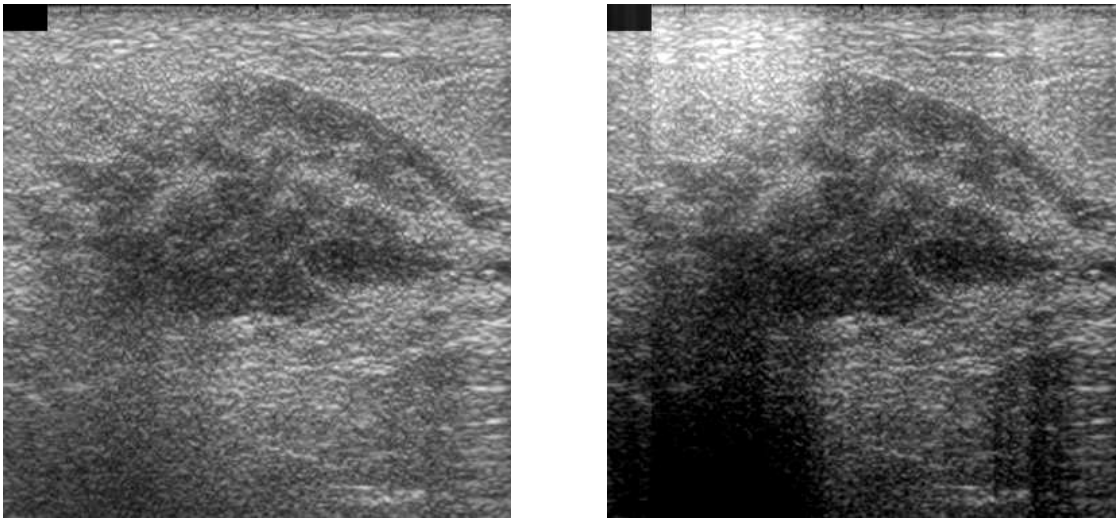
5.4 Sample image registration problem

In this simple example we demonstrate the graph-based estimates of α-Jensen difference and α-MI using basis projection features. Figure 5.4.8 shows two ultrasound images of the breast I_{orig} and I_{def}. I_{orig} is altered by partially suppressing the fundamental spatial frequency and boosting the first harmonic spatial Fourier frequency to create I_{def}. I_{def} shows degradation similar to that seen in ultrasound scans due to transducer noise and change in imaging direction relative to the tissue. It is our intention to demonstrate through this example that such change in frequency components can be captured by a information-measure like Jensen difference divergence or α-MI, through a wise choice of features. We shall consider two cases, one in which the severity of the deformation is high and this image I_{def} is shown along with the original undeformed image I_{orig} in Figure 5.4.8. We shall also consider the case where the severity of the deformation is not sufficient to to make any noticeable change in the image quality, although it could still affect registration. Let this other image be $I_{low-def}$. This image is not shown.

Ideally, the prior information about spatial frequency differences would suggest proclivity for Fourier basis as the features of choice. In this example we use a 2D feature vector composed of the fundamental and first spatial Fourier frequencies acquired through a 2D discrete cosine transform. Let features extracted from I_{orig} be $\{Z_{orig}\}$. Define $\{Z_{def}\}$ and $\{Z_{low-def}\}$ in a similar fashion for images I_{def} and $I_{low-def}$, respectively. When $\{Z_{orig}\}$ and $\{Z_{low-def}\}$ are pooled together as explained in Section 4.1, we obtain $\{Z_{orig} \bigcup Z_{low-def}\}$ and the resultant scatter-plot of pooled features is seen in Figure 5.4.9. On the other hand, when features from I_{orig} and I_{def} are pooled together $\{Z_{orig} \bigcup Z_{def}\}$, the resultant scatter-plot of features is seen in Figure 5.4.9. The increased dispersion of feature samples in the mismatched images, I_{orig} and I_{def}, will lead to higher values of

the pooled entropy of features. Thus the α-Jensen difference, as implemented by MST or kNN length, will discriminate matched versus mismatched images during the registration process.

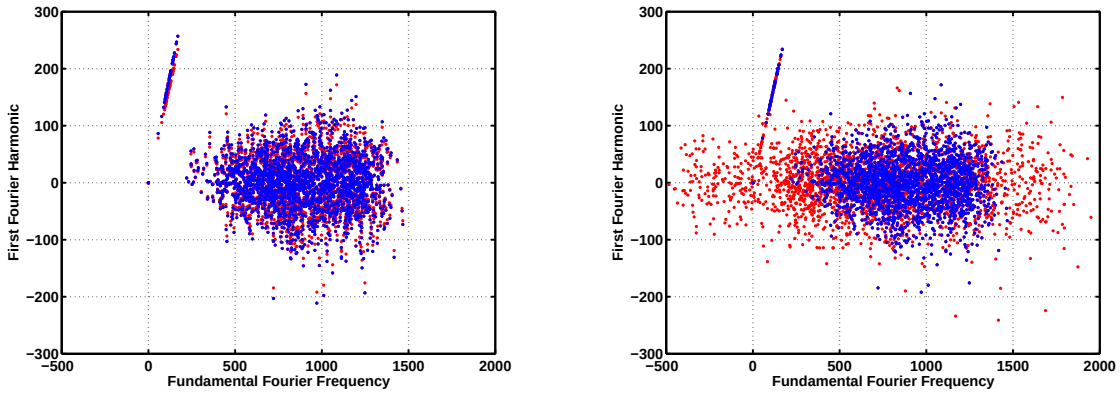
In Figure 5.4.10 the first harmonic frequency components of the DCT at I_{orig} and $I_{low-def}$ are shown in the joint histogram (scatter-plot) of co-occurrences. For the closely matched images, the scatter lies on a diagonal and yields a higher MI value than the case of the mismatched images. The plot of co-occurrences of fundamental frequency coefficients also yields similar plots and the resultant MI is higher for the matched image case.



(a) (b)

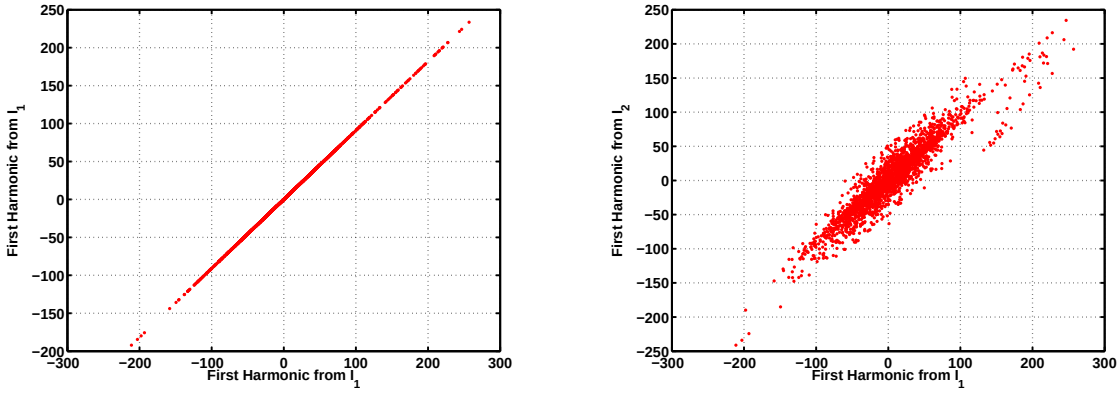
Figure 5.4.8: (a) Original image, I_{orig}, is an UL image of the breast (b) I_{orig} is deformed by selectively filtering spatial frequencies to give I_{def}. $I_{low-def}$ is not shown but it has an appearance similar to I_{orig}.

In conclusion, it is seen that while single-pixel feature representation of images is sufficient in some cases, generalization to higher dimensional features allows a more efficient and compact representation of images. Higher-dimensional features capture image attributes ignored by single-pixels. A roadblock in using higher dimensional features within the entropy framework of Shannon MI has been the unavailability of methods to reliably



(a) (b)

Figure 5.4.9: (a) Pooled feature sample of image with itself $\{Z_{ref} \bigcup Z_{ref}\}$. (b) Pooled feature sample with reference and target images $\{Z_{ref} \bigcup Z_{tar}\}$



(a) (b)

Figure 5.4.10: (a) Joint density of first harmonic DCT frequency from features $\{Z_{orig}, Z_{low-def}\}$ when images are matched and (b) Joint density of first harmonic DCT frequency from features $\{Z_{orig}, Z_{def}\}$ when images are mismatched.

estimate divergence in higher dimensional spaces. Feature-based representation of images can be adapted to the registration problem. The ICA feature basis are adapted to the UL and MRI image databases through a training process. Unlike the ICA bases, wavelets and DCT have no variability based on the image data and thus remain unaffected by noise in the training data.

Before proceeding to the experimental results, it is vital to examine algorithmic and computational considerations of entropic graph methods. The next chapter serves as an overview into the computational methods used for graph construction. It also presents some interesting insights into developing accelerated graph methods. Lastly, it compares different divergence measures based on their numerical complexity.

CHAPTER VI

Computational Considerations

A popular sentiment about graph methods, such as the MST and the kNN graph, is that they could be computationally taxing. However, graph theory algorithms have evolved and several variants with low time-memory complexity have been found. Henze-Penrose and the α-GA mean divergence metrics are based directly on the MST and kNNG and first require the solution of these combinatorial optimization problems. This chapter is devoted to providing insight into the formulation of these algorithms and the assumptions that lead to faster, lower complexity variants of these algorithms.

6.1 Complexity of the MST

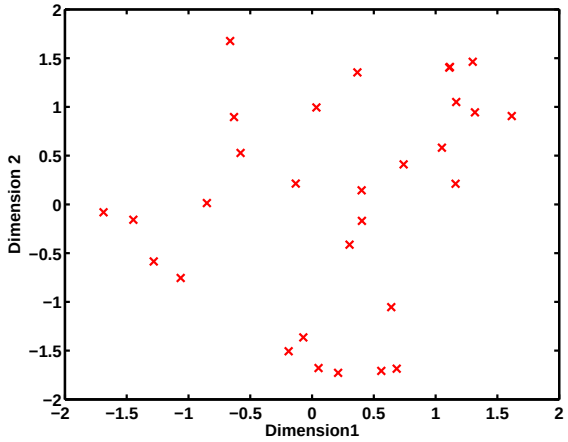
The MST problem has been studied since 1926 when the Boruvka MST algorithm [99] was formulated. Interest in the MST problem led to the discovery of two optimal algorithms, by Prim [104] and Kruskal [73]. The MST has several applications in computer science, pattern recognition and computer vision. Due to its widespread applicability, there have been and continue to be sporadic reductions in the time-memory complexity of the MST problem [10, 46, 91]. However, the two principal algorithms by Prim and Kruskal find wide applicability due to their optimality properties and ease of formulation.

A set of given edges, sorted by their weights, is maintained in a list and Kruskal's al-

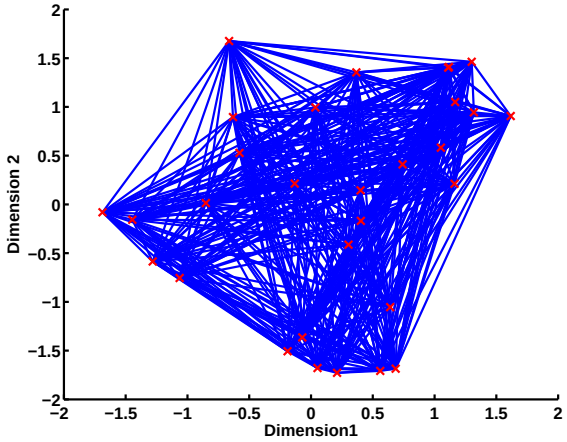
gorithm grows the tree an edge at a time. Cycles are avoided within the tree by discarding edges that connect two sub-trees already joined through a prior established path. The time complexity of the algorithm is of $O(ElogE)$ where E is the initial number of edges in the graph. Prims algorithm starts with an arbitrary vertex v_i and builds a tree by repeatedly finding the edge with the lowest weight that links a new vertex into the tree. Prim's algorithm has a running time of the $O(ElogN_v)$, where N_v is the initial number of vertices in the graph. It is widely held that the Kruskal algorithm has lower runtime for sparse graphs whereas Prim algorithm runs faster than Kruskal's for dense graphs [10]. Here we define the graph density parameter, $\rho = E/N_v$. Graph sparsity is defined as $1 - \rho$. The graph density gives us an estimate of the average number of edges incident on each vertex. As will be seen shortly, the complexity of most MST algorithms increase with the graph density. We shall sometimes refer to vertices of a graph v_i, as nodes or points, and edge weights as edge lengths.

Prim and Kruskal algorithms, although optimal, operate under the constraint that the initial graph is of low or moderate density, i.e. ρ is usually between 10 and 100. Additionally the complexity calculations assume that all edge weights are precomputed and available. In practice, researchers often find themselves starting with a set of random vertices of size N_v, from which they first construct a fully connected graph consisting of $N_v \times (N_v - 1)/2$ unique edges (Figure 6.1). Hence if G is the initial graph, then $G \equiv \{e_{i,j}; i = 1, \ldots, n, j = i, \ldots, n\}$ where $e_{i,j}$ is the edge that connects vertex v_i with v_j. This implies that a full edge matrix of order N_v^2 needs to be maintained in the computer system memory. The computational overhead of calculating edge weights can be significant especially for large N_v in higher dimensional Euclidean spaces ($d >> 2$).

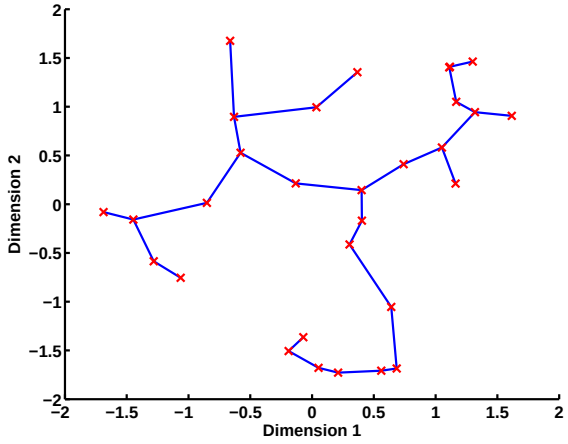
Building MST's from a dense graphs can be especially challenging due to the increased time-memory complexity that standard MST algorithms face when the sparsity constraint



(a) 30 clustered points in space



(b) Fully connected graph



(c) MST over 30 points

Figure 6.1.1: Complexity of generating fully connected tree

is removed. The runtime of both the algorithms increase with the number of edges E in the original graph. Full density graphs have density $\rho = N_v/2$, since $E = N_v \times (N_v - 1)/2$. Understandably, this is the worst case scenario for both Prim and Kruskal algorithms. For full density graphs, Kruskal's algorithm has a time requirement of $O(N_v^2 log N_v)$ and a memory requirement of $O(N_v^2)$. Prims algorithm may perform better than Kruskal for higher density graphs. However, the runtime is still of the $O({N_v}^2 log N_v)$. Note, that these complexity terms do not include the $O({N_v}^2 d)$ computations required to calculate

the $N_v \times (N_v - 1)/2$ edge weights. In the 2D image registration application discussed here, the number of vertices in the graph is of $O(10^5)$ in a 64 dimensional feature space. 3D registration would increase the number of vertices by over two orders of magnitude. Desktop processors cannot handle the very high memory requirements of the standard Kruskal algorithm in this case. Even with larger machines, the algorithm has a forbidding time requirement for tree construction.

Clearly, the problem here is two-fold. We need $O(dN_v^2)$ operations to build a fully connected graph. Additionally, fully connected graphs have $O(N_v^2)$ runtime. To address these issues we have implemented a method for sparsification that allows the MST to be constructed for several hundred thousand points in a few minutes of desktop computing time. This section aims to discuss strategies that reduce (1) the density of the initial graph and (2) to reduce the computational overhead involved in computing ${N_v}^2/2$ edges without sacrificing the minimality or spanning properties of the MST.

6.2 Previous efforts in making MST more efficient

Efficient algorithms for MST have existed since the latter half of the previous century. Surveys of MST algorithms currently in practice are presented in [10, 46, 91]. Kruskal, Prim and Boruvka all use the blue rule for greedy implementations of the MST. Good expositions on the blue rule and related algorithms are in [7, 26, 119]. Renewed interest in faster MST methods has led researchers to develop asymptotically faster algorithms [131, 21, 43, 38], however their practical impact has been limited. These algorithms use advanced data structures and appear to be more complicated to implement thus restricting their usage to advanced users. Most implementations for MST methods still utilize the simple Fibonacci heap data structures [38] within classical Prim or Kruskal algorithms. Karger, Klien and Tarjan [66] have proposed the linear expected-time algorithm that, theo-

retically, is the fastest with expected runtime $O(E)$. However, it was found to be unfeasible for practical use because it requires solution of the complicated subproblem of verifying an MST in linear time, which in turn requires the solution of the nearest common ancestor problem in linear time. Classical algorithms like Prim and Kruskal, which when used with a conventional binary heap, have fast runtime-low implementation complexity.

6.3 Modified Projection-decomposition algorithm (MPDA)

The algorithm developed and presented here is based on the classical projection-decomposition search paradigm first introduced by Friedman [40, 41]. Given a set of points $\mathcal{Z}_i$ in $\mathbf{R^d}$, the algorithm starts by processing points sequentially in the point set $\mathcal{Z}$. The point set $\mathcal{Z}$ is stored as a collection of d 1D arrays where the i^{th} array contains the i^{th} coordinate of the points. The point set $\mathcal{Z}$ is projected and ordered along any one of its d dimensions. However, we retain the forward mapping FM that maps the i^{th} point in the unordered set to the j^{th} point in the ordered set, i.e. $FM\{\mathcal{Z}_i\} = O_j$, where O is the ordered set. Now we have a set of 1D ordered coordinates for $\mathcal{Z}$. For every point p_j in the ordered set, we select m_ϵ neighbors such the projection distance from a neighbor q to p is less than ϵ (refer to Figure 6.3.2), where ϵ is the search radius. All such points are gathered in a list labeled as *candidate list*. Effectively this is the first step of the Friedman nearest neighbor algorithm. We do not continue ordering along all the d dimensions, since the idea here is not to find one nearest neighbor. The goal is to reduce the distance computation operations and sparsify the graph such that none of the edges left due to sparsification had a large probability of occurring in the final MST. This leads to the completion of the first stage of the algorithm, wherein we have built a list of candidate connections per point in $\mathcal{Z}$. For a large number of points and high dimensionality the performance of our method is at par with various other techniques that use ordering along multiple dimensions and complex data structures. The

appeal of our algorithm lies in the absence of any special data structure and in its intuitive simplicity and brevity.

The procedure explained above leads to a reduction in the number of vertices visited. This reduction is, of course, proportional to the search radius ϵ. Repeating the procedure for all the points in $\mathcal{Z}$ yields an entire set of potential edges. It should be noted that any of the data dimensions could be used for the above procedure. We would expect to see that the greatest reduction in edges visited would result if the dimension selected for ordering is the one where the data shows maximum variance. It should also be noted that only integer comparisons and memory lookup operations have been involved. However, there is an integrity check before the points can be finally accepted. From the point p, we visit each vertex q in the candidate list for p and compute the full Euclidean weight of the edge. If the weight of the edge is less than ϵ, the edge is inserted in the initial graph. Note that we have to extend our 1D search to ϵ, to ensure that no point that is within the ϵ radius is left out.

The pseudo-code for the algorithm is given below (Algorithm: 6.1). Algorithms 6.2 and 6.3 present pseudo-codes for the Disjoint-Set-Union-Find data structure and the standard Kruskal algorithm, respectively.

A significant acceleration can be obtained by this process of sparsification of the initial graph before tree construction. The selection criterion imposed on the edges, ensures that only those edges likely to occur in the final MST are included in the original graph G. As seen in Figure 6.3.2, only those edges with weights smaller than disc radius are accepted into the list. The edge-weight sort algorithm, within Kruskal's algorithm, now has to sort $O(N)$ number of edges. The C program for generating MST using the Modified Kruskal and MPDA is given in the Appendix.

The MPDA builds a candidate list from projections of points in a 1D space and hence

1: MSTL = FUNCTION MSTLENGTH(Points)
2: N_v : Number of vertices or features
3: P $\leftarrow \{v_i\}$ //Read point set in P
4: EC $\leftarrow \{\}$ //EC will contain edges of Candidate List
5: LEC $\leftarrow 0$ //Length of EC
6: MSTL $\leftarrow 0.0$ //Length of MST
7:
8: Q $\leftarrow$ FM(P,1) //Q $\equiv$ P sorted along its first dimension, FM $\equiv$ forward map
9: **for** $i = 1$ to N_v **do**
10: Pick current point $q_i \leftarrow$ Q(i)
11: **repeat**
12: Pick next point $q_j \leftarrow$ Q(j)
13: Measure $e^{L_1} \leftarrow (q_i - q_j)^2$ //L_1 distance
14: Measure $e^{L_2} \leftarrow \|q_i - q_j\|$
15: **if** $e^{L_2} \leq \epsilon$ **then**
16: EC $\leftarrow$ EC $\bigcup q_j$
17: LEC $\leftarrow$ LEC + 1 //Increment CANDIDATE LIST EC
18: **end if**
19: **until** ($e^{L_1} > \epsilon^2|$ Reached end of Q)
20: **end for**
21:
22: E $\leftarrow$ KRUSKAL(EC,Length(EC))
23: MSTL $\leftarrow \sum_{j=1,e_j \in E}^{LEC} e_j^{L_2}$

Algorithm 6.1: Modified Projection Decomposition Algorithm

```
SETS = FUNCTION DSUF-CREATE(N_v)
SETS ← N_v disjoint sets
SETS.setsize ← N_v
SETS.parent ← {}
SETS.rank ← {}

FUNCTION DSUF-MAKESET(v_i)
Given vertex v_i, Create a new set whose only member is pointed to by v_i
SETS.parent(i) = i
SETS.rank(i) = 0;

Found = FUNCTION DSUF-FIND(i)
j ← SETS.parent(i)
if i == j then
    Found ← i, STOP and RETURN
else
    for until loop broken do
        k ← SETS.parent(j)
        if j == k then
            BREAK
        else
            j = k
        end if
    end for
    while i ≠ k do
        j ← SETS.parent(i)
        SETS.parent(i) ← k
        i ← j
    end while
    Found ← k
end if

FUNCTION DSUF-UNITE(Set1Index,Set2Index)
rank1 ← SETS.rank(Set1Index)
rank2 ← SETS.rank(Set2Index)
if rank1 == rank2 then
    SETS.parent(Set1Index) ← Set2index
    rank2 ← rank2 +1
else if rank1 > rank2 then
    SETS.parent(Set2Index) ← Set1Index
else
    SETS.parent(Set1Index) ← Set2Index
end if
```

Algorithm 6.2: Disjoint Set Union Find Data structure

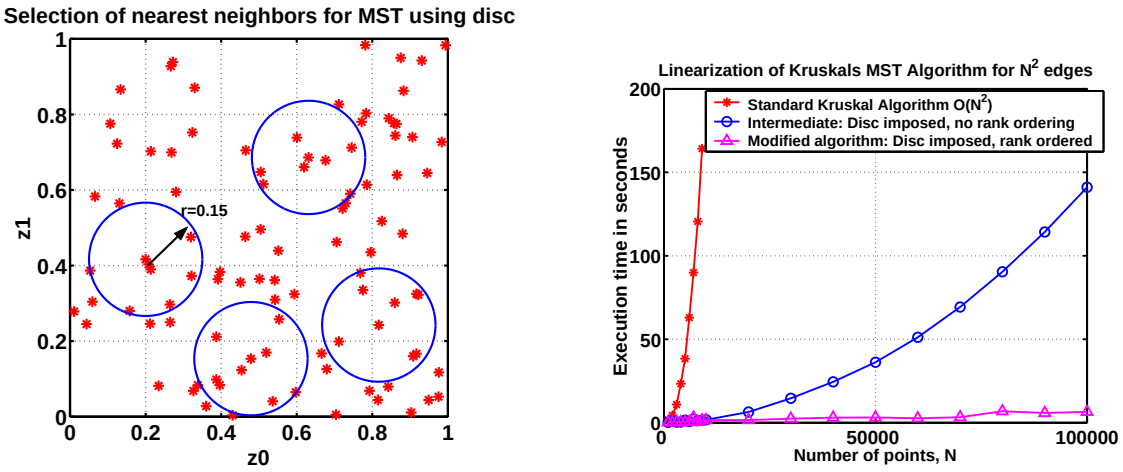
```
1: MSTL = FUNCTION KRUSKAL(EC)
2: E ← {} // E will contain the MST edges
3: SEL ← Sort(EC)
4: SETS = DSUF-CREATE(N_v)
5: for i = 1 to N_v do
6:    DSUF-MAKESET(v_i)
7: end for
8: for each edge e(q_1, q_2) in EC do
9:    root1 ← DSUF-FIND(q_1)
10:   root2 ← DSUF-FIND(q_2)
11:   if root1 ≠ root2 then
12:      E ← ⋃ e(q_1, q_2)
13:      DSUF-UNION(root1,root2)
14:   end if
15: end for
```

Algorithm 6.3: Kruskal MST Algorithm

requires an additional check before the points are admitted in the list. As mentioned before, one can conduct a full range search to pick candidate points. Obviously this is a modification of the k-Nearest neighbor algorithm, where the range search paradigm is used. The range-search kNN picks all NN within a specified ϵ of the query. We implemented a Prim MST algorithm with a Priority Heap data structure. A kNN range-search algorithm, operating in the full d-dimensions of the data space was used to build a candidate list. Although the algorithm is not described here, the C program for such an algorithm is included in the appendix.

6.4 Correctness

Before we proceed to offer a proof of optimality, we would like to provide insight into the rules of MST algorithms for non-cyclicity. To avoid cyclicity in the graph the Prim algorithm, examines only fringe edges. Fringe edges are those edges that connect some vertex in the tree to a vertex not in the tree. Kruskal's algorithm maintains a list of edges ordered by their weights. An edge in the list exists in the MST if it does not form cycles



(a) (b)

Figure 6.3.2: (a) Disc-based acceleration of Kruskal's MST algorithm from $n^2 \log n$ to $n \log n$ and (b) comparison of computation time for Kruskal's standard MST algorithm with respect to our accelerated algorithm.

by connecting components already connected in the tree. It uses a special Disjoint Set Union-Find data structure to determine, find and merge sets rapidly. These techniques are used to avoid cyclicity in the tree.

It is straightforward to prove that if the radius is suitably specified,the disc based tree construction described above is a minimum spanning tree. Recall that the Kruskal algorithm ensures construction of the exact MST [73].

(1) If point $p \in T$, then

$$\|e^{j}_{p \to T, non-cyclic}\| = \min_{i, i \in N_v} \|e^{i}_{p \to T, non-cyclic}\| \tag{6.1}$$

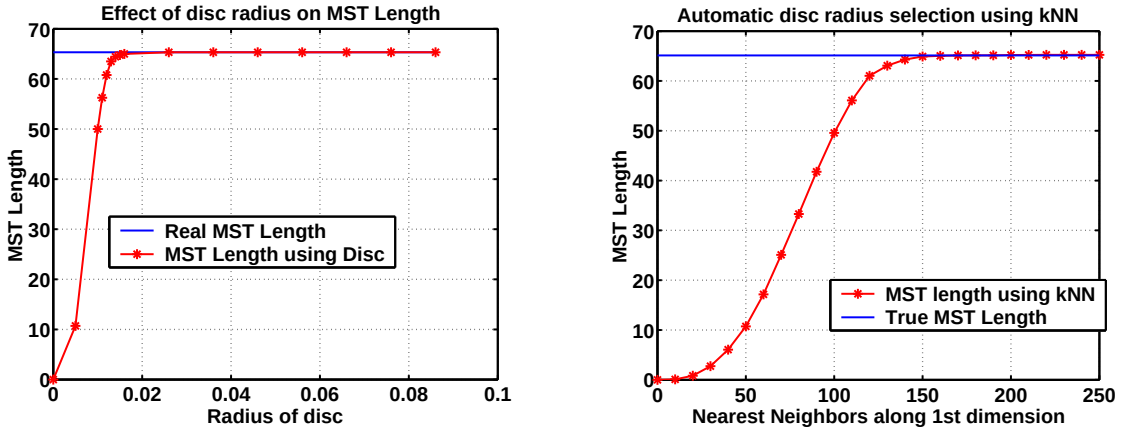
i.e. the edge that connects p to the tree T has the lowest weight amongst all possible non-cyclic paths. To prove this is trivial. The algorithm exhaustively examines all points within distance $\|\epsilon\|$, hence, if a non-cyclic edge $e_{p \to T, non-cyclic}$ is found by imposing the distance selection criterion such that the equation above is satisfied, then that edge is the smallest possible non-cyclic path that connects $p \to T$. The non-cyclicity of the path is ensured in

the Kruskal algorithm through the Union-Find dataset.

(2) If a point $p_i \ni T$,

$$\|e^i_{p\to T, non-cyclic}\|_{i\in N_v} > \|\epsilon\| \tag{6.2}$$

i.e. all the edges between p and its neighbors considered using the distance criterion have total edge weight greater than $\|\epsilon\|$ or have led to a cyclic path. Increasing $\|\epsilon\|$, repeatedly if necessary, will eventually find the path which is lowest in weight and non-cyclic. Since we start with fully connected initial graphs, we do not consider forests (where some points are left out of the tree) as end solutions. If during tree construction the algorithm runs out of edges, but the MST does not include all the points, it is necessary to consider more edges. To this end, increasing $\|\epsilon\|$ will reap additional edges. If the disc radius is underestimated the tree cannot be completed without first adding more edges to the list. If it is overestimated a surplus of edges will result in the edge list, however the final tree will have the required $N-1$ edges only 6.4.3.



(a) (b)

Figure 6.4.3: (a) Bias of the $n \log n$ MST algorithm as a function of radius parameter and (b) as a function of the number of nearest neighbors for uniform points in the unit square.

6.5 Complexity Analysis

Similar to the complexity analysis performed by [98], this section presents a complexity analysis of our algorithm in terms of storage and time requirements of the MST. We begin by examining the time complexity of the algorithm. Firstly, sorting the 1D edge list takes an average of $O(N_v log N_v)$ operations via the quick sort algorithm. Secondly, the number of vertices selected for the L_2 edge weight computation depends on ϵ and the underlying distribution of points in the hyperspace. Understandably, if ϵ is selected poorly, we end up selecting too many points and may start approaching the complexity of an exhaustive MST algorithm. Hence the analysis is presented for specific standard distributions. Let N_{c_i} be the number of edges contributed by the i^{th} point in $\mathcal{Z}$ toward the candidate list. Let $N_C = \sum_{i=0}^{N_v} N_{c_i}$. We are interested in $E(N_C)$. Hence,

$$\begin{aligned} E(N_C) &= E(\sum_{i=0}^{N_v} N_{c_i}) \\ &= \sum_{i=0}^{N_v} E(N_{c_i}). \end{aligned} \tag{6.3}$$

As shown by Nene and Nayar [98], the number of points $N_{c_i} = k$ contributed by any point toward the candidate list is a random variable with a binomial distribution given by

$$P(N_{c_i} = k) = P_c^k (1 - P_c^k)^{N_v - k} \binom{N_v}{k}, \tag{6.4}$$

where $P_c = P(D_c \leq \epsilon)$ and D_c is the 1D projection distance of any point from the i^{th} point in $\{\mathcal{Z}_i\}$. Also, now we can compute

$$\begin{aligned} E(N_{c_i}) &= \sum_{k=0}^{N_v} k P(N_{c_i} = k) \\ &= N_v P_c. \end{aligned} \tag{6.5}$$

The knowledge of the underlying distribution would enable us to determine D_c and hence P_c.

6.5.1 Uniformly Distributed Point Set

Examining the complexity terms for standard uniform and normal distributions should yield a better insight about algorithm performance. When $\{\mathcal{Z}_i\}$ is a uniformly distributed and i.i.d. we see that on the hypercube with length ℓ, we have

$$f_{\mathcal{Z}_i} = \begin{cases} 1/\ell; & \text{if } -\ell/2 \leq z \leq \ell/2 \\ 0; & \text{otherwise} \end{cases}$$

Now, the 1D projection distance $D_c = \mathcal{Z}_c - Q_c$, the density of D_c is written as:

$$f_{D_c|Q_c}\{z\} = \begin{cases} 1/\ell; & \text{if } -\ell/2 - Q_c \leq z \leq \ell/2 - Q_c; \\ 0; & \text{otherwise} \end{cases}$$

and

$$\begin{aligned} P_c &= P\{-\epsilon \leq D_c \leq \epsilon\} \\ &= \int_{-\epsilon}^{\epsilon} f_{D_c|Q_c} dz \\ &\leq \int_{-\epsilon}^{\epsilon} 1/\ell dz \\ &\leq \frac{2\epsilon}{\ell} \end{aligned} \tag{6.6}$$

Finally, we get an upper bound for

$$E[N_{c_i}] = \frac{2\epsilon N_v}{\ell}.$$

Also

$$\begin{aligned} E[N_C] &= \sum_{i=0}^{N_v-1} \frac{2\epsilon N_v}{\ell} \\ &= \frac{4\epsilon N_v}{\ell} \\ &= O(\frac{\epsilon N_v}{\ell}) \end{aligned} \tag{6.7}$$

We see that the algorithm complexity is independent of d.

6.5.2 Normally Distributed Point Set

If $\{\mathcal{Z}_i\}$ were a normally distributed point set, again under i.i.d. conditions with variance σ^2 mean μ:

$$f_{\mathcal{Z}_i} = \frac{1}{\sqrt{2\pi\sigma^2}} exp \frac{-(x-\mu)^2}{2\sigma^2}$$

Hence,

$$f_{D_c|Q_c} = \frac{1}{\sqrt{2\pi\sigma^2}} exp \frac{-(d-\mu-Q_c)^2}{2\sigma^2}$$

and

$$\begin{aligned} P_c &= P\{-\epsilon \leq D_c \leq \epsilon\} \\ &= \int_{-\epsilon}^{\epsilon} f_{D_c|Q_c} dz \\ &= \frac{1}{2}(erf(\frac{\epsilon-\mu-Q_c}{\sigma\sqrt{2}}) + erf(\frac{\epsilon+\mu-Q_c}{\sigma\sqrt{2}})) \end{aligned} \tag{6.8}$$

Figure 6.3.2 compares the performance of the standard Kruskal algorithm with our modified algorithm. The time and memory requirements are significantly reduced.

6.6 Discussion and Future Work: Predicting ϵ

One of the primary obstacles to automating the MST process is an coming up with optimal estimate of ϵ. As seen above, this is the primary parameter that affects the complexity of the algorithm. This section proposes some strategies to estimate ϵ. However, lacking verification, these methods are recommended only after rigorous testing.

6.6.1 Heuristic approach

Figure 6.3.2, shows a plot of the relation between ϵ and estimated length of the MST. In the event of underestimating ϵ the MST is not complete. The graph is an acyclic forest with unconnected trees, a subset of the MST, and hence its length is less than the length of the MST. If ϵ were selected small enough such that all but the nearest neighbor of each

point were pruned, we would get a k-NN graph $(k = 1)$, which is also a subset of the MST. The plot emphasizes the importance of a prudent selection of ϵ. ϵ plays a significant role in the complexity of the algorithm. To obtain the optimal trade-off between bias of the MST length and time-memory complexity, the disc radius should be selected at the knee of the curve seen in Fig 6.4.3.

For uniform distributions, a constant disc radius is optimal for all areas within the distribution. However, for non uniform distributions, ϵ may be forced to adapt according to the underlying distribution. This can be achieved by selecting the distance of the k^{th}NN *along the dimension of ordering* as ϵ for a given point. Let us call this parameter, ϵ^i_{kNN}. Not surprisingly, for uniform distributions, the distance of the k^{th}-NN along the dimension of ordering, remains roughly the same. The real advantage of the kNN technique lies in the ability to adapt ϵ^i_{kNN} on a point-by point basis for non-uniform distributions. We would like to emphasize here, that these are not the true nearest neighbors of the point. These points are the nearest neighbors in the dimension of ordering. Indeed, picking the true nearest neighbors is a valid and legitimate approach, one that we address in the next section.

It should be noted however, that using the above method we may introduce some error in the process. Consider 2 points p_1 that lives in a densely clustered region, and p_2 that lives in a sparsely clustered region. Now,

$$\epsilon^{p_1}_{kNN} < \epsilon^{p_2}_{kNN}.$$

This implies that p_2 will introduce edges in the candidate list, that would not have been included if

$$\epsilon = \epsilon^{p_1}_{kNN}.$$

Due to the weakening of the strict hypersphere condition imposed by ϵ in the previous

methods, we could now have spurious edges in the list. This impedes the rapid convergence of the forest to the MST, as seen in Figure 6.6.4.

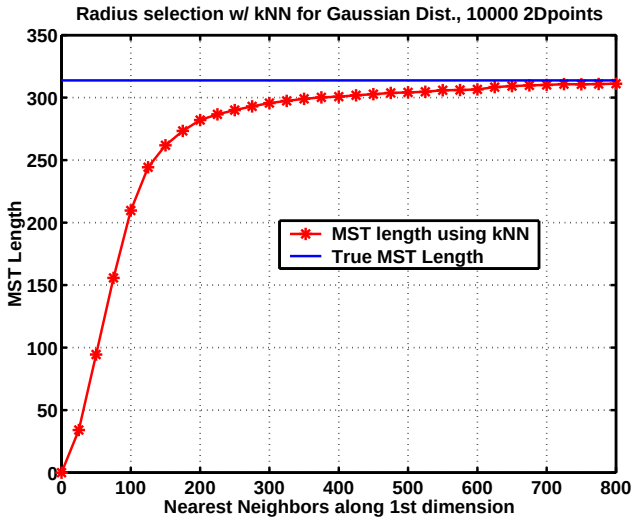


Figure 6.6.4: Constructing the MST based on kNN based disc radius estimate could be a problem in non-uniform distributions due to the slow convergence of the length function

6.6.2 MST construction using the kNN graph

Extending on the approach above, we could pick the kNN of a point on the assumption that the edge connecting nearest neighbors of a point have higher probabilities of participating in the final MST. Here, there are several options available to us. These include the conventional $k-d$ trees method [14, 13] and the list intersection method of Nene and Nayar [98]. We have implemented a version of the Prim algorithm with priority heap that uses the kNN algorithm of Bentley to build a candidate list. The program can be found in the appendix.

The method of list intersection as presented in Nene and Nayar [98] orders data coordinates in every dimension, not just one. Then candidate lists are drawn-up in each dimension according to some distance criterion ϵ. Finally, an intersection operation on the candidate list for the point, which yields points in a hypercube around the query, are

exhaustively searched for the kNN. In this technique, the coordinates are ordered in each dimension, as above. A list of neighbors is selected in the first dimension. Now, this list is intersected with a list of neighbors in the 2nd dimension. The resultant list is intersected with the neighbors in the third dimension. Finally after j intersections (note, the complexity of intersection reduces with j), the result is a list of true neighbors of point p (Figure 6.6.5).

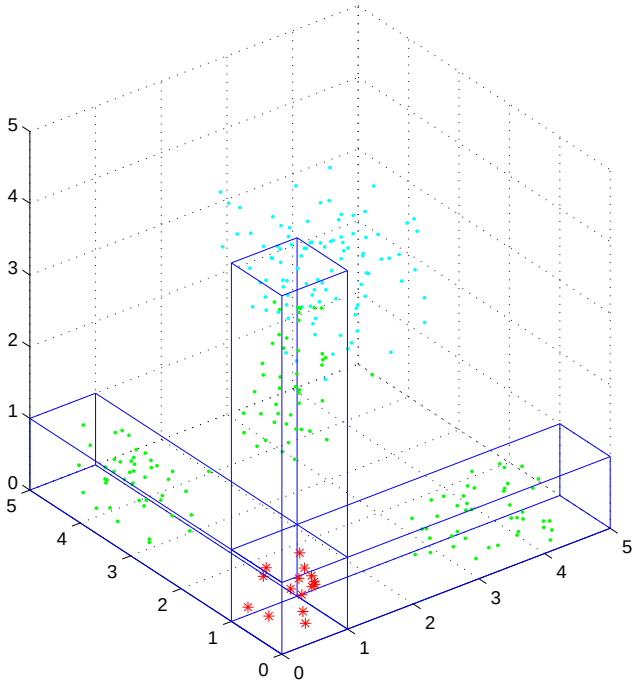


Figure 6.6.5: Picking k-Nearest neighbors within a range ϵ using intersection of lists of 1D ordered coordinates.

6.6.3 Relation to intrinsic dimensionality

An alternate approach to estimate ϵ is by estimating the intrinsic dimensionality of the given distribution. Given all inter-point distances for a distribution of random numbers, one can plot the histogram of such distances. By the central limit theorem, the distribution is approximately normal as $n \rightarrow \infty$. As the intrinsic dimensionality of the distribution increases, the mean of the distribution μ increases and the variance σ^2 decreases [19]. The

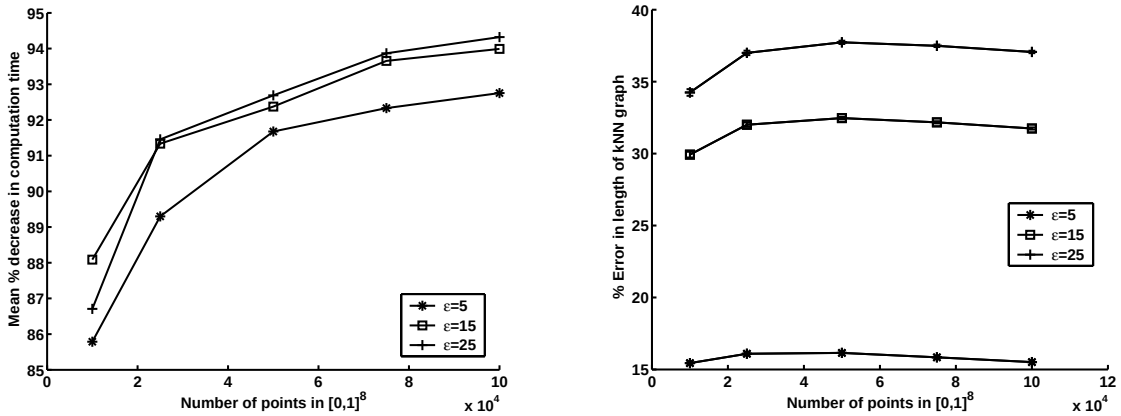
sharper the peak in this histogram, the lesser the number of points we can safely avoid visiting. Since we do not have all the inter-point distances, indeed the main idea here is to avoid calculating all the inter-point distances, we offer an alternative to fast determination of the intrinsic dimensionality of the distribution.

6.7 Acceleration of the kNN Graph construction

Time memory considerations in the nearest neighbor graph have prompted researchers to come up with various exact and approximate graph algorithms. With its wide-spread usage, it is not surprising that several fast methods exist for nearest neighbor graph constructions. Most of them are expandable to construct k-NN graphs. One of the first fast algorithms for constructing NN graphs was proposed by Bentley [14, 13]. In vector spaces with Bentleys kd trees [14, 13] are effective in selecting kNN for a given query point. A comprehensive survey of the latest methods for nearest neighbor searches in vector spaces is presented in [19]. A simple and intuitive method for nearest neighbor search in high dimensions is presented in [98].

Though compelling, the methods presented above focus on retrieving the exact nearest neighbors. One could hypothesize that for applications where the accuracy of the nearest neighbors is not critical, significant speed-up can be achieved by accepting a small bias in the nearest neighbor search. This is the principal argument presented in [5] and this is relevant to divergence estimation since only the total edge length of the kNN is important in entropic graph registration. We conducted our own experiments on the approximate NN method using the code provided in [92] (Figure 6.7.6). We conducted benchmarks on uniformly distributed points in 8 dimensional space. If the error incurred in picking the incorrect k-th nearest neighbor $\leq \epsilon$, the cumulative error in the length of the kNNG is plotted in Figure 6.7.6. Compared to an exact kNN search using k-d trees, a significant

reduction ($> 85\%$) in time can be obtained through approximate NN methods, incurring a 15% cumulative graph length error. The utility of approximate nearest neighbor code such as [5] for entropic graph registration depends on the sensitivity of graph length to small amount of registration error. Thus it is conceivable that a 15% error in graph length be negligible with respect to mean square registration error. This will be explained in the next chapter.



(a) (b)

Figure 6.7.6: Approximate k-NNG: (a) Decrease in computation time to build approximate kNNG for different ϵ, expressed as a percentage of time spent computing the exact kNNG over a uniformly distributed points in $[0, 1]^8$. An 85% reduction in computation time can be obtained by incurring a 15% error in cumulative graph length. (b) Corresponding error incurred in cumulative graph length.

6.8 Computation time

The MPDA and k-d tree algorithms described above are used to construct the MST and kNN graph respectively. The computation time for all the measures of divergence introduced here are tabulated below. Two images with dimensions 256×256 pixels were projected on to a 64D ICA basis and the resultant coefficients were used for divergence estimation.

Divergence (Graph)	Points	Dim	Trees (Points, Dim)	NN Queries	Time (s)
α-Jensen (MST)	2 N	d	1 (2N, d)	-	2.22
α-Jensen (kNNG)	2 N	d	1 (2N, d)	2Nk	4.75
Henze-Penrose (MST)	2 N	d	1 (2N, d)	-	2.54
α-GA (kNNG)	2 N	d	2 (N, d)	2N	4.80
α-MI (kNNG)	N	2 d	1 (N, 2d) + 2 (N, d)	3N	4.11
NLCC (kNNG)	N	2 d	1 (N, 2d) + 2 (N, d)	3N	4.10

Table 6.1: Average divergence computation time on two 256×256 pixel images, decomposed using 8×8 (64D) ICA. Computation times were tabulated using MATLAB ©C-mex program running under the Linux OS on a 2.4GHz 32-bit Intel Xeon Processor machine with 2GB RAM and 533MHz bus speed.

The MST computation time appears to be shorter than the computation time for the kNN. This is due to the careful selection of the search radius ϵ for constructing the MST. In practice, such a careful selection is difficult, if not impossible. The number of feature vectors from each image is $1024 = (256 \times 256)/(8 \times 8)$. For larger number of feature vectors, the MPDA shows slower convergence and the MST computation time will increase.

Further, it is seen that α-GA, α-MI and NLCC all show different computation time even as all of them depend on the kNN graph construction. α-GA required the construction of 2 k-d trees in 64D space at a computational complexity of $NlogN$, where $N = 1024$ for each tree. $2 \times N$ queries were executed in $O(\sqrt{N} \times k)$ time, where k is the number of nearest neighbors. α-MI requires the construction of a k-d trees in 128D space and 2 k-d trees in 64D space, but required only N queries to be executed in 64D space.

CHAPTER VII

Applications

This chapter describes three different registration applications and illustrates the entropic graph registration methods presented in this thesis.

7.1 Ultrasonic Breast Image Registration

An important application of the methods presented in this thesis is the co-registration of a pair of ultrasound (UL) images of the breast. Success in this application should lead to full 3D and 4D registration: the fourth dimension being time. Accurate registration of breast UL image volumes could lead to a breakthrough in the use of whole breast imaging. In particular, accurate UL breast imaging is essential to detect and quantify changes that can aid discrimination of malignant from benign lesions [16, 113]; can be used to detect multi-focal secondary masses [47] and can quantify response to chemotherapy or radiation therapy [67]. Currently,breast lesions are missed by UL community practitioners in up to 45% of women with dense breasts [69].

To understand the long term significance of image registration to breast cancer detection it should be noted that UL usage has progressed rapidly in diagnostic breast imaging; it has proven to be a reasonable task to scan a localized region in the breast and make a judgment regarding the likelihood of malignancy. It is a much more difficult task to assess

an entire breast volume and whole breast ultrasound imaging has not become standard practice. On one side, there are too many regions in many breasts that look abnormal, leading to increased biopsies or threat of legal action if a detected tumor was not biopsied. On the other side, numerous subtle lesions may be missed in whole breast scanning by all but the most dedicated, highly trained radiologists. To greatly improve this situation in breast cancer detection of occult masses new approaches are required. Precise registration of image volumes to allow rapid, localized comparison of ultrasound studies separated by months or years should greatly facilitate detection of changes and assessment of their potential malignant growth patterns. What is needed is a much more accurate and robust registration method to provide the needed confidence in the registration for studies with a wider range of image quality. Segmentation of artifacts, such as acoustic shadows and reverberations is a particularly important task not yet addressed, for the artifacts can be the dominant ”information” in some cases, and yet are affected strongly by subtle changes in the UL view. The same general techniques should work equally as well for 3D MRI.

Evaluation of breast tumor change during neoadjuvant chemotherapy using some of these ultrasound techniques will not only determine the contribution of this modality in chemotherapy, but may also contribute to the development of whole breast UL as a method for breast cancer detection in high risk women. As use of neoadjuvant chemotherapy is being explored for an increasing fraction of breast cancers, the need is apparent for a readily available surrogate end point to assess whether the treatment should be extended. UL, with or without contrast agent, shows promise similar to that of radionuclide and MRI techniques, but at considerably reduced cost (and worldwide availability). The advanced quantification techniques require a great deal of trained radiologist time to outline the tumor pre- and post-therapy and the tumor fraction in areas of incomplete tissue invasion is currently not estimated.

Image registration could also improve spatial compounding of UL images. In compounding, partly correlated views of the region of interest (ROI) are generated by scanning the ROI at different transducer tilt angles and then registering pairs of separate views. Compounding can result in an improvement in the signal-to-speckle ratio of UL images and lead to better delineation of specular reflectors [71]. The field of view of high resolution UL images is insufficient for full use of UL in detecting asymptomatic lesions in the breast and for tracking changes in response to treatment. To create an image of the entire breast or a large fraction of it, the small volume covered by a single scan can be extended by repeatedly scanning the breast in parallel, partially overlapping sweeps that can then be combined using registration [71]. Finally, registration of images collected from different isonation angles can also be used in Doppler imaging where the color flow acquisitions do not detect blood flow well when its direction of motion is normal to the direction of the ultrasound beam and where accurate triangulation can measure flow velocity.

Registration methods based on mutual information (MI), e.g. MIAMI Fuse ©(Mutual Information for Automatic Multimodality Image fusion) [87, 88], have been shown to outperform standard correlation and template-matching methods. Mutual information-based, linear and nonlinear registrations can be performed with a quality that will meet the needs of many applications in ultrasound, in a reasonable, but as yet undetermined, fraction of the cases evaluated. However, some cases have proven difficult to register with expected accuracy. For ultrasonic image volumes of the breast, the low signal-to-speckle noise ratio and abrupt deformations at tissue boundaries, as well as acoustic shadowing and refraction artifacts, are particular challenges that reduce the information shared by the reference and secondary image scans. Current methods leave much to be desired for ultrasound applications since they are each variously overly sensitive to some combination of spurious image components such as speckle, low contrast of key structures, anisotropic

backscatter, shear/compressive tissue deformation, and shadowing which are ubiquitous to UL breast imaging.

Accurate registration of breast ultrasound images is essential for whole breast imaging in order to efficiently detect asymptomatic breast lesions. This is a long-range goal. In the intermediate time range, the technique can aid discrimination of benign and malignant masses by facilitating retrieval of similar masses and visual or automatic comparison of a suspected mass, or even ultrasonically detected calcifications, with structures in the same region from a previous examination. The immediate contribution of these methods is the development of new registration techniques and comparisons of the registration accuracy and robustness with existing methods on a series of ultrasound volume sets of increasing realism and difficulty.

7.1.1 Feature driven entropic graph registration of ultrasound images

This thesis takes a novel approach to ultrasound image registration that gets around the disadvantages of single pixel- or voxel-based registration techniques. It deviates from previous approaches through the inclusion of highly specific independent component image features and use of a generalized information divergence-matching criterion implemented with entropic graphs. The matching methods presented here have the following advantages: 1) use of the generalized divergence enables examining and selecting the most stable objective function having optimal discrimination capability; and 2) use of higher order features captures non-local spatial information which is ignored in the standard single pixel MIAMI Fuse ©algorithm and which can lead to more accurate and robust image registration.

Local structural features tags and independent component analysis, introduced in Chapter IV are evaluated. The hypothesis that supplementation of single pixel or single voxel

matching by the matching of structural features specifically designed for registering ultrasound images and image volumes of the breast will significantly reduce the effect of speckle, shadowing, and non-linear distortions on registration accuracy will be evaluated. A rationale for supplementing single pixel features for UL image registration is that single pixel features cannot discriminate between speckle, shadowing and true structural characteristics of the image, leading to lack of robustness. Also, sharp features that might improve accuracy might not be given appropriate weighting. Prefiltering or precropping the UL images prior to registration is an ad hoc solution that requires human intervention. In contrast the multidimensional feature framework includes spatial discriminants directly into the feature space.

Use of generalized divergence as a registration criterion will improve the fine spatial resolution of the registration as compared with existing methods and facilitate function optimization. The validation of the technique is limited to tightly controlled quantitative comparisons on simulated and test object two-dimensional images. However, given the results presented here it is believed that more ambitious studies would be justified.

7.1.2 Database of breast UL images

In ultrasound of the breast, the reference and secondary images have genuine differences from each other due to biological changes and differences in the imaging, such as positioning of the tissues during compression and angle dependence of UL scattering from tissue boundaries. The tissues are distorted out of a given image plane as well as within it. Speckle noise, elastic deformations and shadows further complicate the registration process thus making Ultrasound breast images notoriously difficult to register. Comparison of entropy-based image discriminants such as Shannon MI and Rényi's α-MI to the MST-based α-Jensen difference divergence is presented.

The database used for this application is a set of 3D ultrasound scans of the left or right breast of 21 female subjects, aged 21-49 years, going to biopsy for possible breast cancer. Each volume scan is acquired at 10mm depth resolution yielding about 90 cross-sectional images at 5mm horizontal resolution. The lower age range was chosen to provide a sample of more complex breasts, which are also somewhat more difficult to diagnose than typical breasts of older women. Fig 7.1.1 shows slices of breast ultrasound image volumes representative of those found in clinical practice. The women were imaged on their backs with the transducer placed so as to image through the breast toward the chest wall. Twenty test cases chosen from the breast database are presented. The images exhibit connective tissue structure, malignant tumors in echogenic fibroglandular tissues and benign cysts. Tumors characteristically show discontinuous edges with a darker center and shadows below the borders. Area of enhancement below the cysts and some solid tissues are not uncommon. Some images also show uncommon degrees of degradation due to shadowing. The bottom two-thirds of these images include the chest wall and the dark shadow and reverberations behind the acoustically impenetrable boundary between the lung and chest wall. Some edge information is evident, however shadowy streaks are observed due to dense tissue absorbing the sound beam, refraction and phase correlation at oblique boundary or poor acoustic impedance match (air bubbles) between the transducer and the skin. For clarity of presentation and speed of processing we focus on registration of 2D slices. The extension of our methods to fully 3D voxel registration is straightforward but will not be presented here. The value of α used for all simulations is 0.5.

A 64D ICA bases was extracted from a training database of 6 volumetric breast ultrasound scans of patients. A separate test set of 15 patients also undergoing biopsy was also created. Each volumetric scan has a field of view of about 4cm3 (voxel dimensions are $0.1mm^2 \times 0.5mm$) and encompasses the tumor, cyst or other structure of interest. Tu-

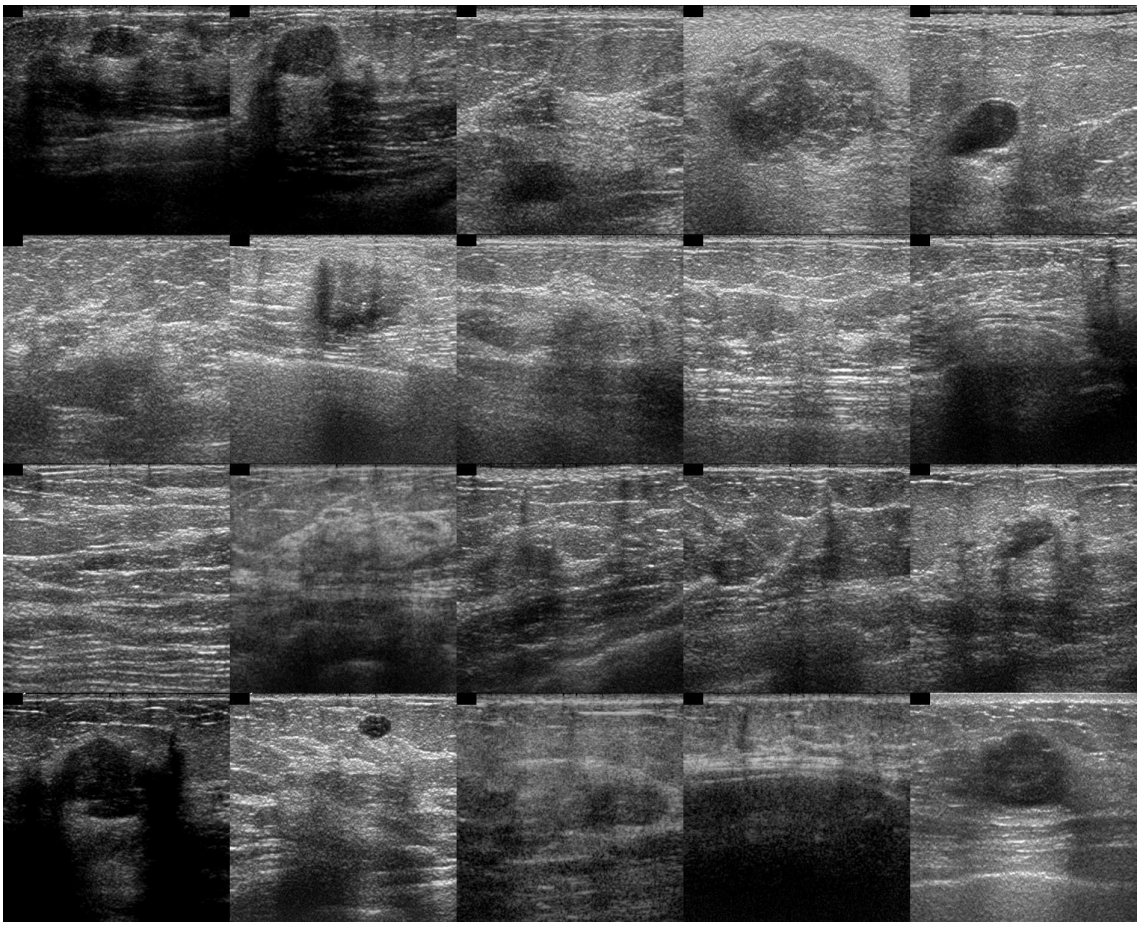


Figure 7.1.1: Ultrasound (UL) breast scans from twenty volume scans of patients undergoing chemotherapy.

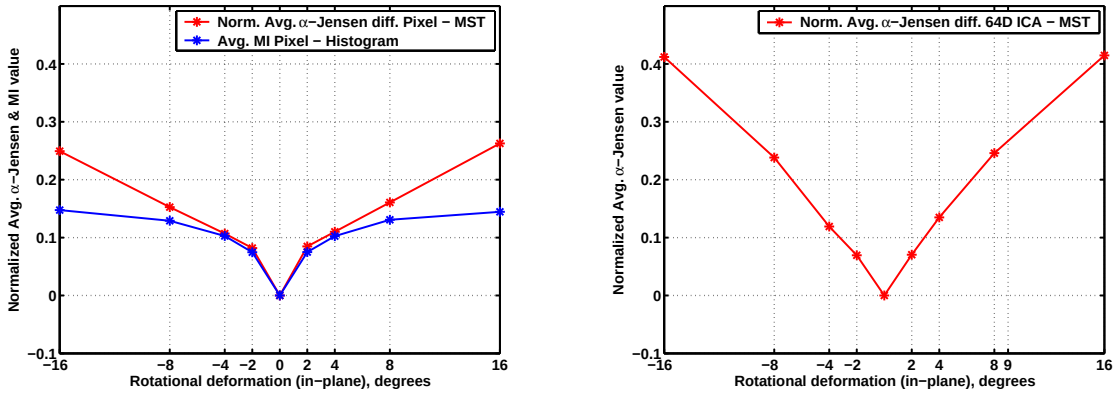
mor/Cyst dimensions vary and can range from 5 mm^3 to 1 cm^3 or higher. A slice from the UL breast image volume is registered to a slice 2mm deeper in the same image volume over 250 trials. At this separation distance, the speckle noise decorrelates. However the underlying anatomy remains approximately unchanged. As the aim of this study is to quantitatively compare different feature selection and registration methods we restricted our investigation to rotation deformations over $\pm 16^\circ$.

Two cases were investigated: a reduced dimension feature set consisting of only the 8 most discriminating of the 64 feature dimensions and the full 64 dimensional features. The FastICA algorithm provides independent components one by one based on the projection pursuit directions of the training data. Thus, the independent components are ranked according to closeness to Gaussian distribution of the coefficients [61]. Using this criterion, the first 8 of the 64 basis are selected and used for registration.

7.1.3 Experiments

Figure 7.1.2 shows average profiles of the registration objective functions. The panel on far left of Fig. 7.1.2 indicates that, for single-pixel features, entropic-graph (MST) estimates of α-Jensen difference and histogram plug-in estimates of MI give similarity functions with virtually identical profiles having a unique global minimum at the correct 0° rotation of the reference image. In the full 64 dimensional ICA feature space the MST-based Jensen difference criterion maintains a smooth profile (right panel) with a single global minimum at the correct location. In the 64 dimensional feature space the histogram plug-in estimates of MI or α-Jensen difference are not implementable.

To investigate the effect of small perturbations on small-angle registration performance varying amounts of truncated Gaussian noise were added to each pixel. Registration accuracy for single-pixels, tags, and discrete and continuous ICA features using the histogram

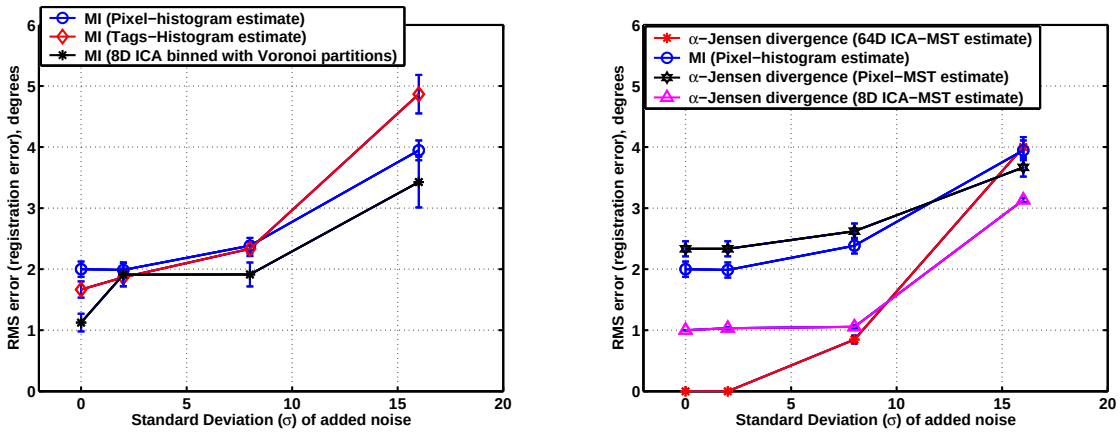


(a) (b)

Figure 7.1.2: Normalized average profiles of image matching criteria for registration of UL breast images taken from two slices of the image volume database: (a) MST-based α-Jensen and histogram-based α-MI for single pixel features and (b) MST-based α-Jensen for 64D ICA coefficient vector features.

and entropic-graph estimates of MI and α-Jensen difference divergence were compared under increasing noise conditions. Figure 7.1.3 shows plots of registration root mean square (RMS) error versus increasing levels of additive (truncated) Gaussian noise in the images. Shown on the plots are standard error bars. The resultant registration peak shifts from the perfect alignment position (0° relative rotation) by an amount depending on the SNR, the registration features used and entropy/MI estimation matching criteria adopted.

In the figure a comparison of MI versus α-Jensen ($\alpha = 0.5$) registration methods applied to single pixel, tag, and ICA features can be made. The left panel of the figure illustrates MI implemented with density plug-in estimates. For the single pixel and 8×8 tag features, the bivariate coincidence density for MI and entropy was estimated using the standard binned histogram. For single pixel features there were 256×256 bins and for tag features there were $(256^{64}) \times (256^{64})$ bins. The tag features were pruned down to 256×256 through adaptive thresholding [44, 4, 97]. For the ICA features the 8 most discriminating ICA dimensions were computed from training on 10000 breast samples from the 21 breast



(a) (b)

Figure 7.1.3: (a) Effect of additive Gaussian noise on the RMS error of the peak position of the Shannon-MI estimated using histograms on single-pixel intensity gray levels, 8×8 tag features extracted using Geman's [44, 4] adaptive thresholding method and histograms on 8D ICA features binned using Voronoi partitions. (b) RMS error for Shannon MI estimated using histograms on single-pixel intensity levels, α-Jensen difference divergence estimated directly with the MST on single-pixels, 8D ICA coefficient vector features and 64D ICA coefficient vector features. These plots are based on 250 repeated experiments from within the breast UL volumetric database of 21 breast cancer patients undergoing therapy. The two slices to be registered are spatially separated by 2mm. Search was restricted to a maximum rotation angle of $\pm 16^{\circ}$. The confidence intervals represent unit standard error in the computation of the mean of the interval.

volumes and the joint coincidence density (defined on $8 \times 8 = 64$ dimensional joint feature space) was estimated by partitioning the 8D features space using 256 Voronoi bins on the same training sample.

The registration MSE, for these three methods, is shown in the left panel of Figure 7.1.3 and the reader will notice a moderate improvement in MSE performance of MI registration with the tag and discretized ICA feature based density plug-in methods over the standard single pixel density plug-in method. We postulate that only modest improvement gains are attainable since histogram estimation becomes unstable in such a high (8) dimensional feature space.

In the 64 dimensional feature space the histogram plug-in estimates of MI or α-Jensen difference are not implementable. The right panel of Figure 7.1.3 illustrates how entropic graph based Jensen difference estimates significantly outperforms the MI registration methods. The α-Jensen difference divergence is calculated for single-pixel intensity gray levels, 8D ICA coefficient vector features and 64D ICA coefficient vector features using the MST entropic-graph estimate. It also shows the Shannon MI calculated using single-pixel intensities via the histogram plug-in estimator.

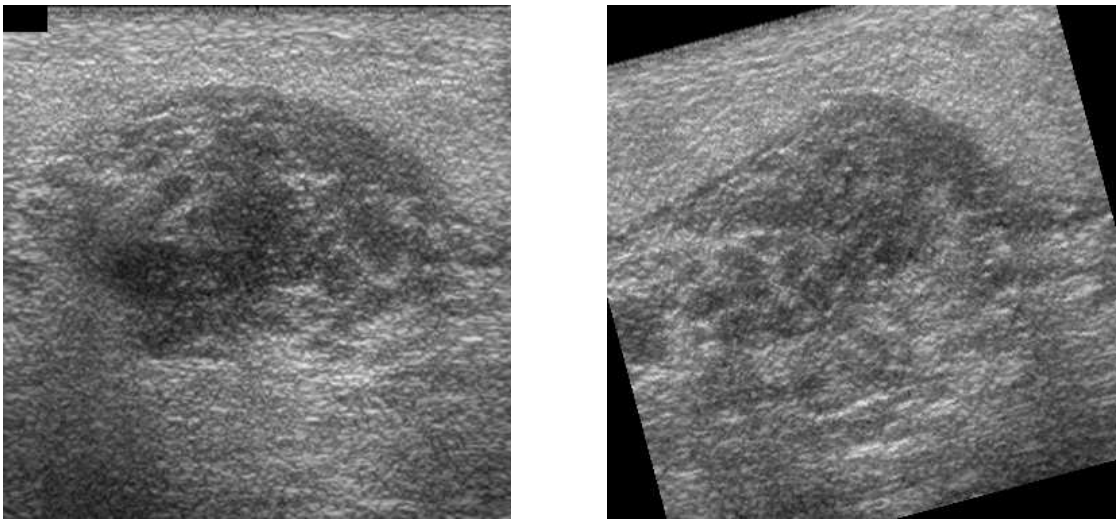
From Figure 7.1.3(b) observe that the performance of the α-Jensen difference and the standard Shannon MI are comparable when implemented with single-pixel features. When computed over a high dimensional feature set, such as the 8D and 64D ICA coefficients, the α-Jensen difference divergence performance improves. This improvement is derived from the use of a higher dimensional and more discriminatory feature set, and an entropic graph estimator of α-Jensen difference.

These preliminary results show that it may be beneficial to extensively investigate other divergence measures. However, due to the lack of a significant gain in registration MSE, histograms in higher dimensional spaces perform poorly as compared to entropic graph

methods. A second experiment was performed to investigate the entropic methods in more detail. Two image slices with tumor were extracted from each of the 15 test scans such that they showed the cross-section of the tumor. A representative example is shown in Figure 7.1.4. Slices have a separation distance of about 5mm, about twice the separation distance than the previous study, a situation that further broadens the registration peak in the matching function. The first cross sectional slice was picked such that it intersected with the ellipsoidal-shaped tumor through its center. The second slice was picked closer to the edge of the tumor. The selected slices thus show a natural decline in tumor size, as would be expected in time sampled scans of tumors responding to therapy.

Since view direction changes from one image scan to the next for the same patient over time, rotational deformation is often deployed to correct it during registration. We simulated this effect by registering a rotationally deformed image with its unrotated slice-separated counterpart, for each patient in the 15 test cases. Rotational deformation was in steps of 2 degrees such that the sequence of deformations ranged from -16 to $+16$ degrees. To remove any residual correlation, the images were offset (relatively translated) by 0.5mm (5 pixels) laterally since the correlated noise could bias the registration results. Since some displacement can be expected from the handheld UL imaging process and the relative tissue motion of the compressible breast tissue, this is not unreasonable deformation. For each deformation angle, divergence measures were calculated, where the 'registered state' is the one with 0 degree of relative deformation.

Figure 7.1.5 shows average objective function plots for the registration experiment discussed above. Thirty different noise realizations were added to the fifteen test images at every angle of rotational deformation to give $N = 400$ different images for calculation of the matching functions. In the figure, each graph plots the sample mean, $\hat{\mu}_\theta$, calculated over the N measurements at each angle, θ. The standard deviation of $\hat{\mu}_\theta$, also called the



(a) (b)

Figure 7.1.4: UL Images of the breast separated and rotationally deformed. (a) Cross-sectional image through center of tumor. (b) Rotated cross-sectional image acquired at a distance 5mm away from Image in (a).

standard error of the measurements, is given by $\sigma_{M_\theta} = \sigma_\theta/\sqrt{N}$ for $\theta \in \{-16^\circ, \ldots, +16^\circ\}$, where σ_θ is the standard deviation of the N measurements made at each rotational deformation. To normalize the images it is important to discount for the relative scaling between the matching functions. Hence, $\hat{\mu}_\theta$ of each matching function is normalized such that $\max(\sigma_{M_\theta})$ is unity. This restricts arbitrary scaling and also discounts for any scaling inherent in the computation of the matching function. In each row, the extent on the search space is identical. This facilitates comparison of two divergence estimates and also allows for comparison of a particular divergence as noise increases. It can readily be seen from the trends that at low levels of noise, all feature based estimates have sharper peaks than the Shannon MI estimate using pixel features. Further, as noise increases some divergence estimates, notably α GA and αMI divergence between the ICA features of the images, maintain sensitivity to rotational deformation.

Not surprisingly, these trends translate into improved MSE performance of the high-

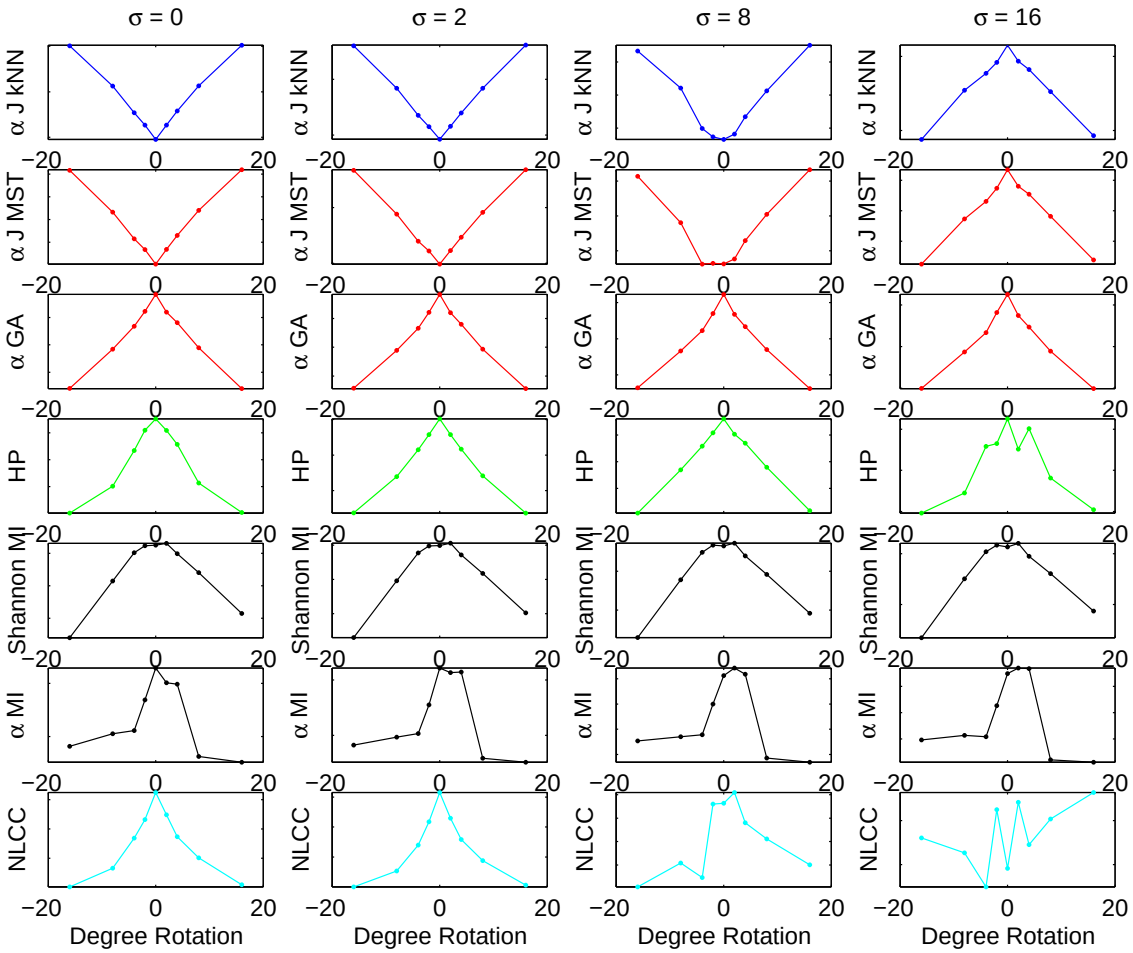


Figure 7.1.5: Normalized average profiles of image matching criteria for registration of UL breast images taken from two slices of the image volume database under decreasing SNR. All plots are normalized with respect to the maximum variance in the sampled observations.(row 1) kNN-based estimate of α-Jensen difference divergence between ICA features of the two images, (row 2) MST-based estimate of α-Jensen difference divergence between ICA features of the two images, (row 3) NN estimate of α Geometric-Arithmetic mean affinity between ICA features, (row 4) MST based estimate of Henze-Penrose affinity between ICA features, (row 5) Shannon Mutual Information estimated using pixel feature histogram method, (row 6) α Mutual Information estimated using NN graphs on ICA features and lastly, (row 7) NN estimate of the Non-linear correlation coefficient between the ICA feature vectors. Columns represent objective function under increasing additive noise. Column 1-4 represent additive truncated Gaussian noise of standard deviation, $\sigma = 0$, 2, 8 and 16. Rotational deformations were confined to $\pm$ 16 degrees.

dimensional entropic graph based estimates of divergence, as seen in Figure7.1.6. At low noise, all measures of divergence between ICA feature vectors show superior performance over the single pixel histogram estimate of Shannon MI.

In conclusion, it should be noted that investigation of UL image registration to aid detection and management of breast cancer is relatively new in and of itself, and not widespread. Only recently, with single pixels MI techniques, has such registration appeared to be very promising. The methods proposed here to investigate image registration have not been previously applied to UL image registration. Based on preliminary results these methods appear to offer significant improvement on single pixel based methods for registration of ultrasound breast images. The approach described here combines improvement of registration accuracy and robustness by automated selection of spatial features derived from independent components analysis and uses image feature matching criteria which extend the mutual information to a more stable criterion. These two ideas are combined into a computationally tractable registration algorithm. Future studies could explore 3D registration and further the ability of the new registration technique to improve discrimination between benign and malignant masses in follow-up studies or to improve visual or automated detection of cancers via serial studies in a high-risk population. One could also test the utility of our new registration technique for image volumes under pre and post chemotherapy conditions. For example, one could: 1) better estimate the final volume of tumors undergoing chemotherapy; and 2) better outline corresponding regions to test tumor shrinkage, reduction of vascularity and reduction of signal from targeted and non-targeted contrast agents as surrogate indicators of complete pathological response.

The registration methods presented above are applicable to other cross modality problems in image registration and matching. The next section presents applications of entropic graph based matching methods to image retrieval problems in a multimodal setting.

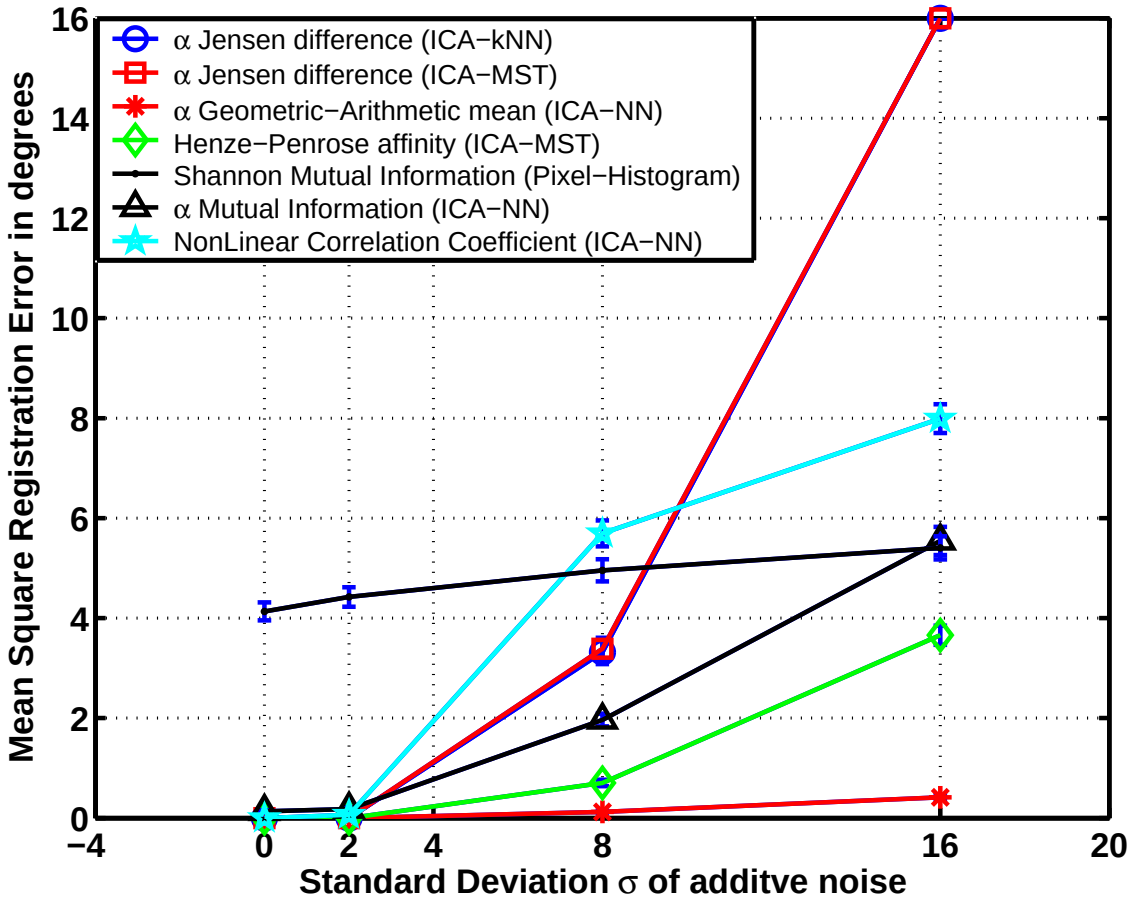


Figure 7.1.6: Rotational RMS error obtained from registration of UL imagery using seven different image similarity/dissimilarity criteria namely αJensen difference calculated using MST and kNN, Henze-Penrose affinity, αGeometric-Arithmetic mean divergence, αMutual Information, NLCC and Shannon MI. Shannon MI was computed using single-pixel intensity histograms with 1 bin per intensity level. 64D ICA feature vectors were used with MST or kNN graph to compute the other measures of divergence. These plots are based on 250 repeated experiments from within the breast UL volumetric database of 21 breast cancer patients undergoing therapy. The two slices to be registered are spatially separated by 5mm. Search was restricted to a maximum rotation angle of $\pm 16^{\circ}$. The confidence intervals represent unit standard error in the computation of the mean of the interval.

7.2 Multimodal Face Retrieval

Images databases are now commmonplace in large servers and desktops alike. Face retrieval from a database is a difficult problem due to the high variability of facial expressions, poses, and illuminations. The widespread availability of thermal cameras, such as those used at airports for detecting fever patients in the SARS outbreak of 2003, provides an opportunity to couple information from visible-light and infrared sources for face identification. Thermal face images lack the textural information typically found in visible images thus complicating the matching procedure and providing us with a challenging testbed for our methods. Our multisensor face image retrieval example intends to illustrate the flexibility of the entropic matching methods.

Many different approaches to this problem have been proposed [81, 20, 130]. Our objective in this paper is not to compete with these many fine tuned approaches. Rather we simply wish to illustrate the flexibility of the MST entropic matching method that we have presented in earlier sections. The Equinox visible/IR face database [33] consists of 7GB image data of 91 individuals photographed under 3 illumination conditions, poses, and facial expressions using a joint co-registered visible longwave infrared (V/LWIR) camera. Figure 7.2.7 shows a sampling of faces in this database. Given a V/LWIR image pair for a person the multimodal face retrieval problem is to extract a corresponding pair of images of the person from the database. Multimode retrieval using visible and thermal imagery is difficult due to the prominence of contour information as opposed to the textural details available in visible imagery. The lack of textural information typically leads investigators to use facial landmarks for indexing images. However, facial landmarks change positions and aspects with expressions and movement making them unreliable. This makes entropic methods of retrieval compelling for this problem due their ability to capture complex rela-

tions using high dimensional features and requiring no user intervention.

We implemented MST-based entropic retrieval as follows. We pulled queries at random from the database and used this query to test the image matching method between the query and the remaining images in the database. Two sample queries are illustrated in Figure 7.2.8. The remainder of the database was compared to the query image and rank-ordered with respect to their similarity with the query image. It should be noted that the images are used as seen in Figure 7.2.7, i.e. no segmentation was performed to delineate the background from the face. A perfect match was declared if the image with highest rank, as measured by estimated α-Jensen difference, matched the person in the query image. Rather than implement ICA, which has been reported to have deficiencies for face recognition [80], we used an 8x8-DCT basis set to extract the features. Matching is done in a 66-dimensional space, using 64 of the DCT coefficients dimensions plus 2 dimensions corresponding to spatial coordinates. The α-Jensen difference is computed by building the MST over this high-dimensional space for each image pair. The measured correct retrieval rate was a respectable 95.5% even using this relatively simple feature set and the simple MST α-Jensen image matching method.

7.3 Multimodal satellite image registration

Images acquired via geostationary satellites serve in research related to heat dissipation from urban centers, climactic changes and other ecological projects. In this section, we shall illustrate entropic graph based image registration for a remote sensing example. Images of sites on the earth are gathered by a variety of geostationary satellites. A satellite may carry more than one sensor and may acquire images throughout a period of time. These sensors gather information in distinct frequency bands in the electromagnetic spectrum and help predict daily weather patterns, environmental parameters influencing crop

Figure 7.2.7: Sampling of faces in the Equinox V/LWIR face database [33]. The database consists of 100 individual faces at various illumination, pose and facial expression configurations. Each visible-light image is co-registered to infrared counterpart by the camera.

Figure 7.2.8: Two examples of queries taken from the Equinox face database.

cycles such as soil composition, water and mineral levels deeper in the Earth's crust, and may also serve as surveillance sensors meant to monitor activity over hostile regions.

Changing weather conditions interfere with sensor signals. Images captured in a multisensor satellite imaging environment also show deformations due to the position of the sensors relative to the object. This deformation is often affine in nature and may manifest itself as relative translational, rotational or scaling between images. This provides a good setting to observe different divergence measures as a function of the relative deformation between images. Linear rotational deformation is simulated in order to reliably test the image matching functions. Affine and projective transformations do arise in this modality, but they are not addressed in this simulation. Thermal and visible light images captured for the Urban Heat Island [105] project form a part of the database used here. NASA's visible earth project [93] also provides images captured via different satellite sensors, and such multi-band images have been used here to provide a rich representative database of satellite images. Figures 7.3.9 and 7.3.10 show several visible aerial images and their corresponding thermal counterparts that make up the database of images used in this experiment. Thermal and visible-light sensors image different bands in the elec-

tromagnetic spectrum and thus have different intensity maps, making correlation-based registration methods difficult to apply. In particular, this situation requires mapping of the two modalities to a common feature set.

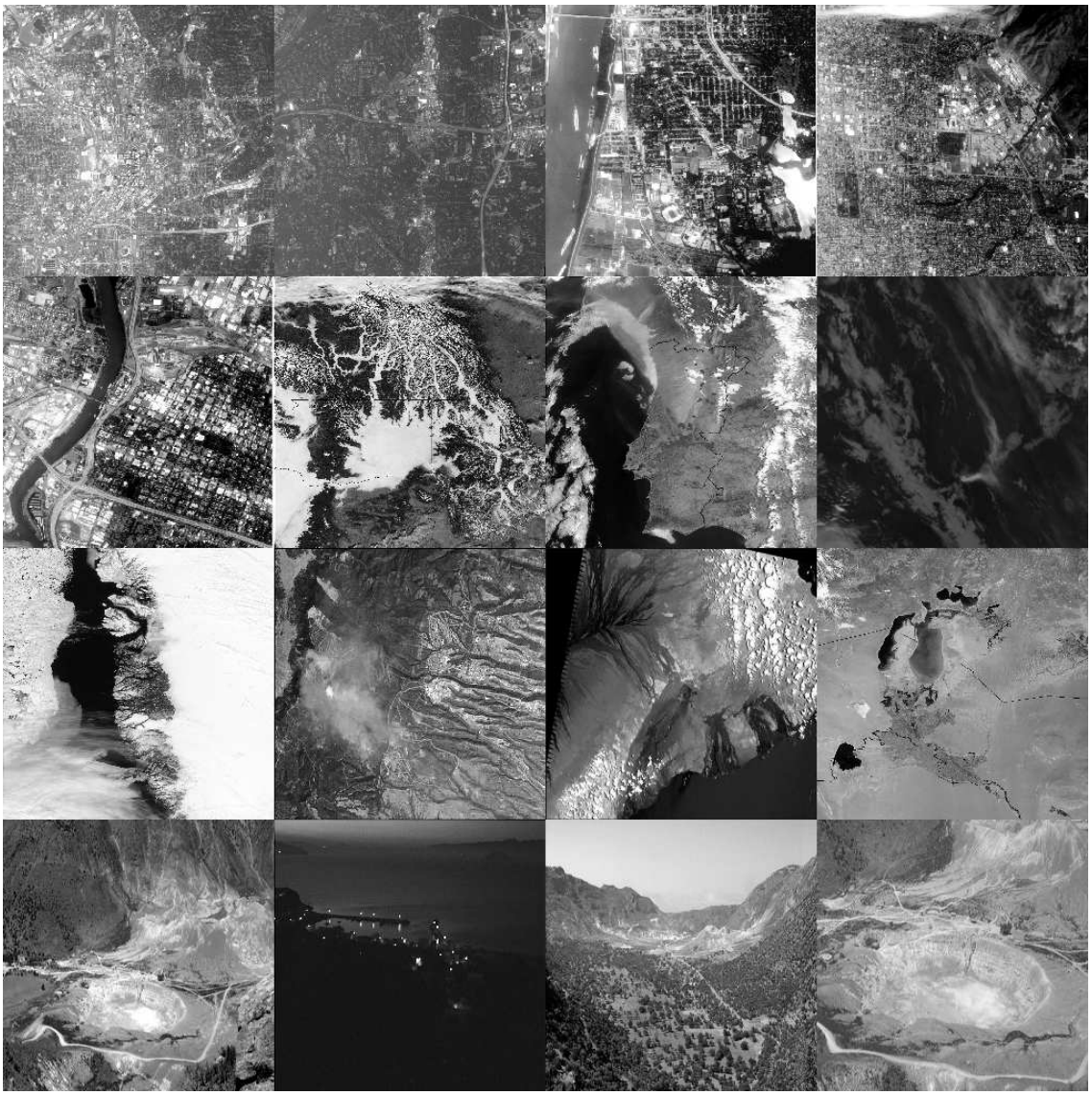


Figure 7.3.9: Visible-light image samples from multisensor satellite image database.

Figure 7.3.11 shows two images of downtown Atlanta, captured with visible and thermal sensors, as a part of the ‘Urban Heat Island’ project [105] that studies the creation of high heat spots in metropolitan areas across the USA. Pairs of visible light and thermal

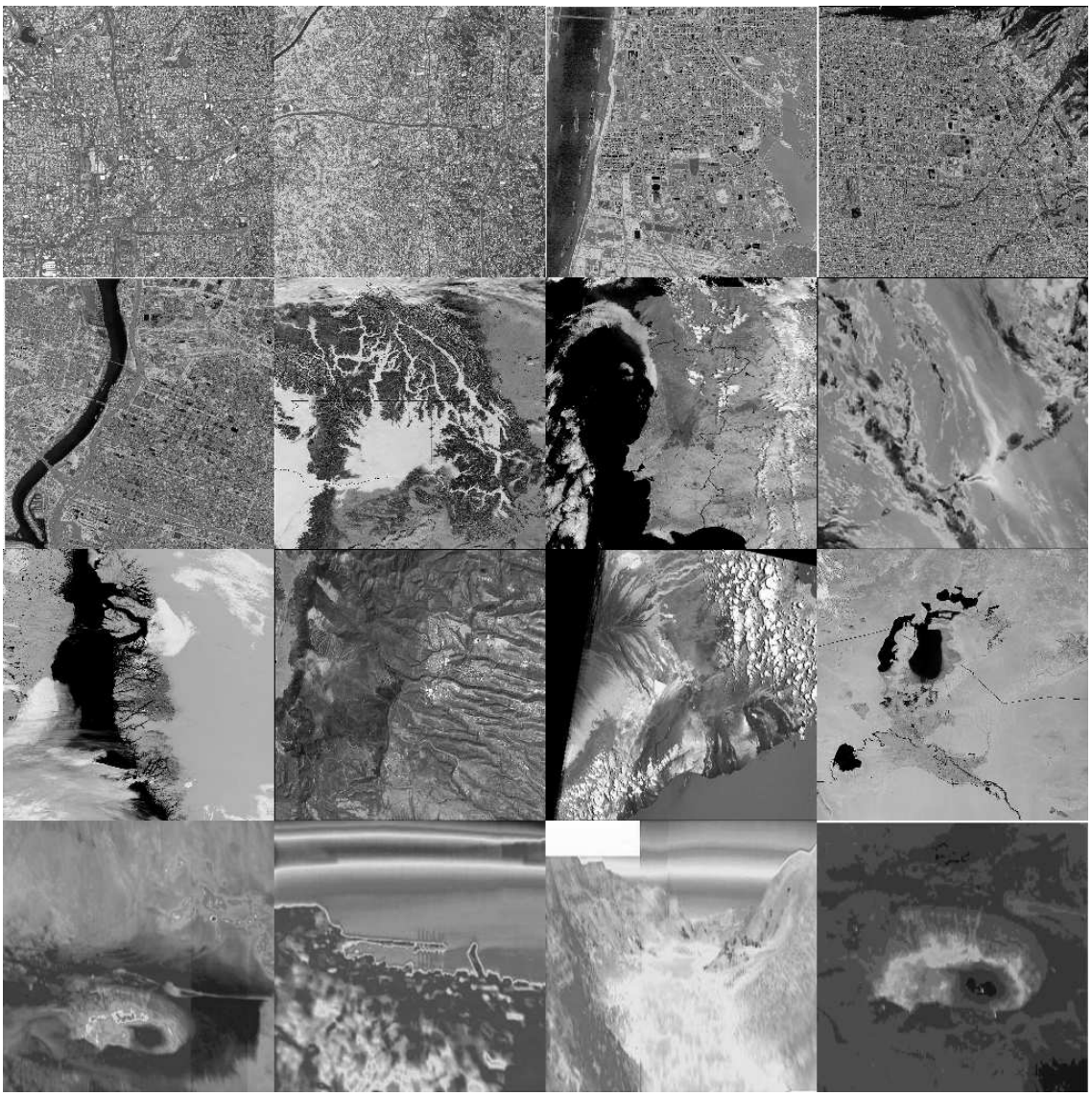
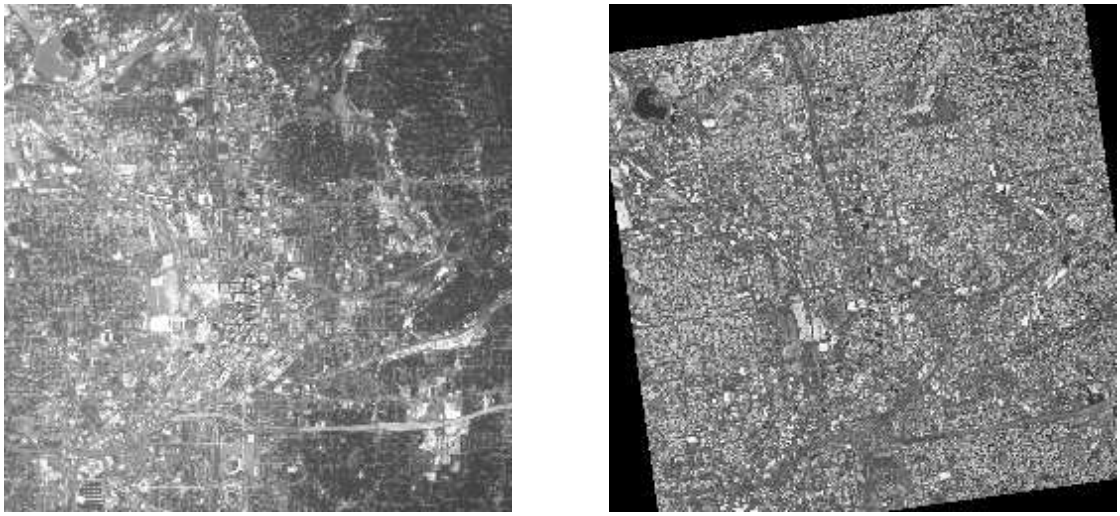


Figure 7.3.10: Thermal image samples corresponding to visible-light images from multi-sensor satellite image database shown in Figure 7.3.9.

satellite images were also obtained from NASA's Visible Earth website [93]. The variability in imagery arises due to the different specialized satellites used for imaging. These include weather satellites wherein the imagery shows heavy occlusion due to clouds and other atmospheric disturbances. Other satellites focus on urban areas with roads, bridges and high rise buildings. Still other images show entire countries or continents, oceans and large geographic landmarks such as volcanoes and active geologic features. Lastly, images contain different landscapes such as deserts, mountains and valleys with dense foliage.



(a) (b)

Figure 7.3.11: Images of downtown Atlanta obtained from Urban Heat Island project [105]. (a) Thermal image (b) Visible-light image under artificial rotational transformation

7.3.1 Feature definition for registration

Images are rotated through 0° to 32°, with a step size adjusted to allow a finer sampling of the objective function near 0°. The images are projected onto a Meyer wavelet basis, and the coefficients are used as features for registration. A feature sample from an image I in the database is represented as a 3-tuple consisting of the coefficient vector, and a two dimensional vector identifying the spatial coordinates of the origin of the image region

it represents. For example $\{\underline{W_{(i,j)}}, x_{(i,j)}, y_{(i,j)}\}$ represents the 3-tuple from position $\{i, j\}$ in the image. Now, $\underline{W_{(i,j)}} \equiv \{w_{(i,j)}^{Low-Low}, w_{(i,j)}^{Low-High}, w_{(i,j)}^{High-Low}, w_{(i,j)}^{High-High}\}$, where the super-script identifies the frequency band in the wavelet spectrum. Features from both the images $\{Z_1, Z_2\}$ are pooled together to form a joint sample pool $\{Z_1 \bigcup Z_2\}$. The MST and k-NN graph are individually constructed on this sample pool.

Figure 7.3.12 shows the rotational mean-squared registration error for the images in our database, in the presence of additive noise. Best performance under the presence of noise can be seen through the use of the α-MI estimated using wavelet features and kNN graph. Comparable performances are seen through the use of Henze-Penrose and αGeometric-Arithmetic mean divergences, both estimated using wavelet features. Interestingly, the single pixel Shannon MI has the poorest performance which may be attributed to its use of poorly discriminating scalar intensity features. Notice that the α-GA, Henze-Penrose affinity, and α-MI(Wavelet-kNN estimate), all implemented with wavelet features, have significantly lower MSE compared to the other methods.

Further insight into the performance of these wavelet-based divergence measures may be gained by considering the mean objective function over 750 independent trials. Figure 7.3.13.a shows the α-MI, HP affinity and the α-GA affinity and Fig. 7.3.13.b shows the α-Jensen difference divergence calculated using the kNN graph and the MST. The sensitivity and robustness of the dissimilarity measures can be evaluated by observing the divergence function near zero rotational deformation (Figure 7.3.13).

7.4 Local feature matching

This section aims to demonstrate the sensitivity of entropic-graph methods to local deformation. Local registration of images, of the human brain acquired under simulated reconstruction of dual modality (T1,T2 weighted) magnetic resonance imaging, is per-

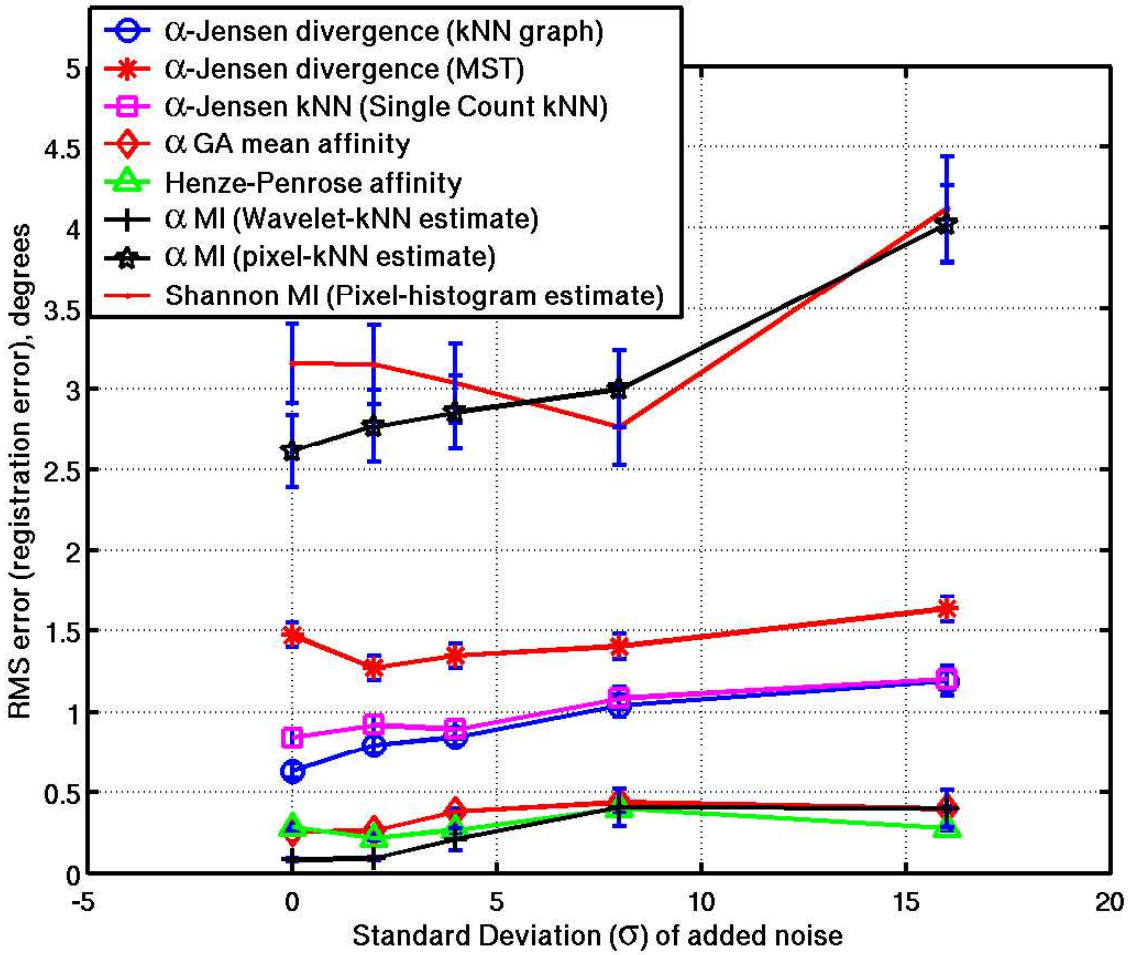
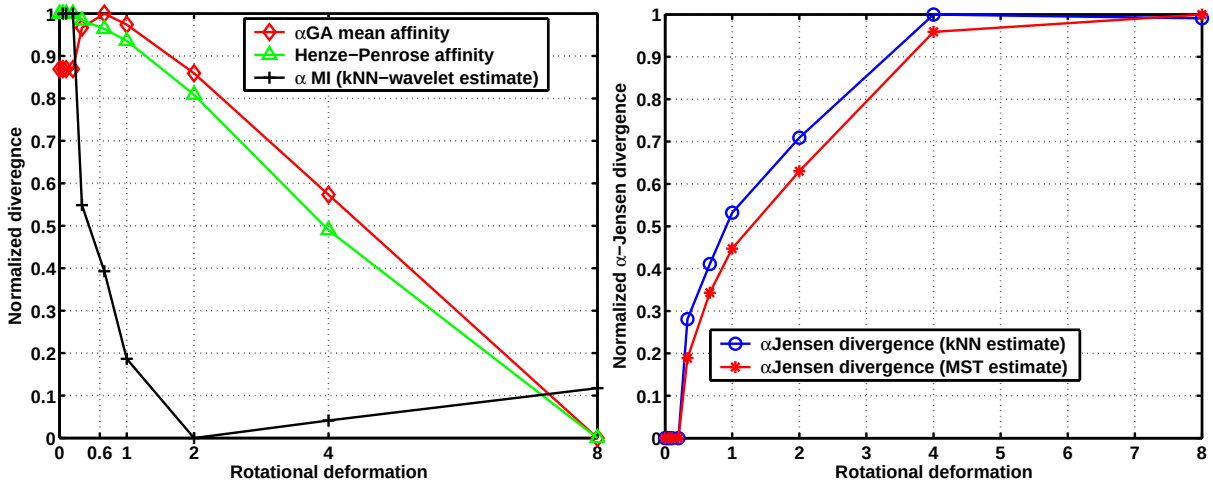


Figure 7.3.12: Rotational RMS error obtained from registration of multisensor V/IR Satellite imagery using six different image similarity/dissimilarity criteria namely αJensen difference calculated using MST, kNN and kNN 'single-count', Henze-Penrose affinity, αGeometric-Arithmetic mean divergence, αMutual Information, and Shannon MI. Shannon MI was computed using single-pixel intensity histograms with 1 bin per intensity level. Wavelet feature vectors were used with MST or kNN graph to compute the other measures of divergence. These plots are based on 250 repeated experiments from within the Satellite image database of 20 aerial images. Search was restricted to a maximum rotation angle of $\pm 16^{\circ}$. The confidence intervals represent unit standard error in the computation of the mean of the interval.



(a) Average α-GA affi nity, HP affi nity and α-MI (kNN-wavelet estimate. Rotation angle estimated by maximizing noisy versions of these objective functions.

(b) Average α-Jensen divergence (kNN and MST estimate on wavelet features). Rotation angle estimated by minimizing noisy versions of these objective functions.

Figure 7.3.13: Average affinity and divergence, over all images, in the vicinity of zero rotation error: (left) α-Jensen (kNN) and α-Jensen (MST), (right) α-GA mean affinity, HP affinity and α-MI estimated using wavelet features and kNN graph.

formed. Different areas in the brain (neural tissue, fat and water) have distinct magnetic resonance properties. Hence, they express different levels of excitation when appropriately time-weighted. This example qualifies as a multisensor fusion example due to the disparate intensity maps generated by the imaging sequence 2.4.3, commonly referred to as the T1 and T2 time weighted MRI sequences. An image matching technique for MRI images sensitive to local perturbations in the image is demonstrated.

The ability to discriminate differences between images with sensitivity to local differences is pivotal to any image matching algorithm. Previous work in these techniques has been limited to simple pixel based mutual information (MI) and pixel correlation techniques. In [102], local measures of MI outperform global MI in the context of adaptive grid refinement for automatic control point placement. However, the sensitivity of local MI deteriorates rapidly as the size of the image window decreases below 40×40 pixels in 2D.

The main constraints on these algorithms, when localizing differences, are (1) limited feature resolution of single pixel intensity based features, and (2) histogram estimators $h(X, Y)$ of joint probability density $f(X, Y)$ are noisy when computed with a small number of pixel features and are thus poor estimators of $f(X, Y)$ used by the algorithm to derive joint entropy $H(X, Y)$. Reliable identification of subtle local differences within images, is key to improving registration sensitivity and accuracy [95]. Stable unbiased estimates of local entropy are required to identify sites of local mismatch between images. These estimates play a vital role in successfully implementing local transformations.

7.4.1 Deformation localization

Iterative registration algorithms apply transformations to a sequence of images while minimizing some objective function. We demonstrate the sensitivity of our technique by

tracking deformations that correspond to small perturbations of the image. These perturbations are recorded by the change in the mismatch metric.

Global deformations reflect a change in imaging geometry and are modeled as global transformations on the images. However, global similarity metrics are ineffective in capturing local deformations in medical images that occur due to physiological or pathological changes in the specimen. Typical examples are: change in brain tumor size, appearance of micro-calcifications in breast, non-linear displacement of soft tissue due to disease and modality induced inhomogeneities such as in MRI and nonlinear breast compression in XRay mammograms. Most registration algorithms will not be reliable when the size of the mismatch site is insufficiently small, typically $(m \times n) \leq 40 \times 40$ [102]. With a combination of ICA and α-entropy we match sites having as few as 8×8 pixels. Due to the limited number samples in the feature space, the faster convergence properties of the MST are better suited to this problem. Although we do not estimate other divergence measures, α-Jensen calculated using the MST provides a benchmark for their performance.

In Figure 7.4.14, multimodal synthesized scan of T1 and T2 weighted brain MRI each of size 256×256 pixels [25] are seen. The original target images shall be deformed locally (see below) to generate a deformed target image.

7.4.1.1 Local deformation using B-Splines

B-spline deformations are cubic mapping functions that have local control and injective properties [23]. The 2D uniform tensor B-spline function F, is defined with a 4×4 control lattice ϕ in $\mathbb{R}^2$ as:

$$F(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} B_i(u) B_j(v) \phi_{ij}, \tag{7.1}$$

where $0 \leq u, v \leq 1$, ϕ_{ij} represents the spatial coordinates of the lattice and the B_i are the standard B-Spline basis functions. The uniform B-Spline basis functions used here are

quite common in computer graphics literature. They may be found in [23] and are defined as:

$$\begin{aligned} B_0(u) &= \frac{(1-u)^3}{6}, \\ B_1(u) &= \frac{3u^3 - 6u^2 + 4}{6}, \\ B_2(u) &= \frac{-3u^3 + 3u^2 + 3u + 1}{6}, \\ B_3(u) &= \frac{u^3}{6}. \end{aligned} \tag{7.2}$$

Given that the original images have 256×256 pixels, we impose a grid(Φ) of 10×10 control points on I_{tar}. Since the aim is to deform I_{tar} locally, not globally, we select a sub-grid (ϕ) of 4×4 control points in the center of I_{tar}. We then diagonally displace, by $\ell = 10$ mm, only one of the control points in ϕ, to generate deformed grid ϕ_{def}. I_{tar} is then reconstructed according to ϕ_{def}. The induced deformation is measured as $||\phi_{def} - \phi||$. Figure 7.4.14 shows the resultant warped image and difference image, $I_{tar} - T(I_{tar})$. For smaller deformations, Φ is a finer grid of 20×20 points, from which ϕ is picked. A control point in ϕ is then displaced diagonally by $\ell = 1, 2, \ldots 10$ to generate ϕ_{def}. When $\ell \leq 3$, noticeable deformation spans only 8×8 pixels.

7.4.2 Feature discrimination algorithm

We generate a d-dimensional feature set $\{Z_i\}_{i=1}^{m \times n}$, $m \times n \geq d$ by sequentially projecting sub-image block (window) $\{\Gamma_j\}_{j=1}^{M \times N}$ of size $m \times n$ onto a d-dimensional basis function set $\{S_k\}$ extracted from the MRI image, as discussed in Section 5.2. Raster scanning through I_{ref} we select sub-image blocks $\{\Gamma_i^{ref}\}_{i=1}^{M \times N}$. For this simulation exercise, we pick only the sub-image block Γ^{tar} from $T(I_{tar})$ corresponding to the particular pixel location $k = (128, 128)$. $\Gamma_{128,128}^{tar}$ corresponds to the area in I_{tar} where the B-Spline deformation has been applied.

The size of the ICA basis features is 8×8, i.e. the feature dimension is, $d = 64$. The MST is constructed over the joint feature set $\{Z_i^{ref}, Z_j^{tar}\}$. When suitably normalized with $1/n^{\alpha}, \alpha = 0.5$, the length of the MST becomes an estimate of $H_{\alpha}(Z_i^{ref}, Z_j^{tar})$. We score all the sub-image blocks $\{\Gamma_i^{ref}\}_{i=1}^{M \times N}$ with respect to the sub-image block $\Gamma_{128,128}^{tar}$. Let O_{ℓ} be the resultant $M \times N$ matrix of scores, at deformation ℓ. The objective function surface O_{ℓ} is a similarity map between $\{\Gamma^{ref}\}_{i=1}^{M \times N}$ and Γ^{tar}. When two sites are compared, the resulting joint probability distribution depends on the degree of mismatch. The best match is detected by searching for the region in I_{ref} that corresponds to Γ^{tar} as determined by the MST length. As opposed to the one-to-all block matching approach adopted here, one could also perform a block-by-block matching, where each block Γ_i^{ref} is compared with its corresponding block Γ_i^{tar}.

7.4.3 Local Feature matching Results

Figure 7.4.14 shows O_{10} for $m \times n = 8 \times 8$, 16×16 and 32×32. Similar maps can be generated for $\ell = \ell_1, \ell_2, \ldots \ell_p$. The gradient $\nabla(O) = O_{\ell_1} - O_{\ell_2}$ reflects the change in H_{α}, the objective function, when I_{tar} experiences an incremental change in deformation, from $\ell = \ell_1 \rightarrow \ell_2$. This gradient, at various sub-image block size is seen in Figure 7.4.14, where $\ell_1 = 0$ and $\ell_2 = 10$. For demonstration purposes in Figure 7.4.14, we imposed a large deformation to I_{tar}. Smaller deformations generated using a control grid spanning only 40×40 pixels are used to generate Figure 7.4.15. It shows the ratio of the gradient of the objective function:

$$R = \frac{\frac{1}{(m \times n)} \sum_{i=1}^{m \times n} |\nabla(O(i))|}{\frac{1}{(M \times N - m \times n)} \sum_{i=1}^{M \times N - m \times n} |\nabla(O(i))|}, \tag{7.3}$$

over the deformation site v/s background in the presence of additive Gaussian noise.

Figure 7.4.16 shows the similarity map O_{ℓ} when constructed using a histogram estimate of joint entropy calculated over sub-image size $m \times n$ (3.6). At lower sub-image

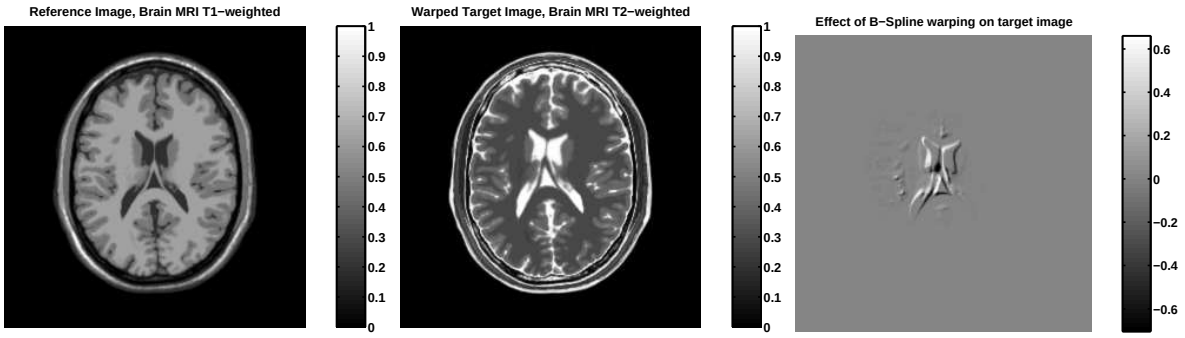
sizes, the estimate displays bias and several local minima even under noise free conditions. It is thus unsuitable for detection of local deformation of I_{tar}.

The framework presented here could be extended to (1) enhance registration performance by sensitizing it to local mismatch, (2) automatically track features of interest, such as tumors in brain or micro-calcifications in breast across temporal image sequences, (3) reliably match or register small images or image regions so as to improve disease diagnosis by locating and identifying small pathological changes in medical image volumes and (4) automate control point placement to initiate registration.

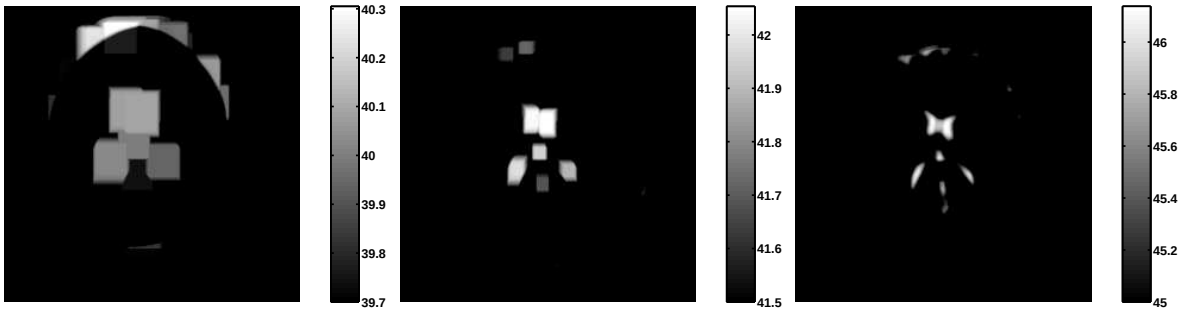
The next section presents the extensions of entropic graph methods to simultaneous multi-image registration. The aim is to demonstrate the utility of entropic graph methods for estimation of entropy and divergence such as α-GA, α-MI and NLCC between several images simultaneously. It is introductory in nature and serves to identify potentially exciting future directions for research in the area of entropic graph based entropy and divergence estimation.

7.5 Simultaneous multi-image registration

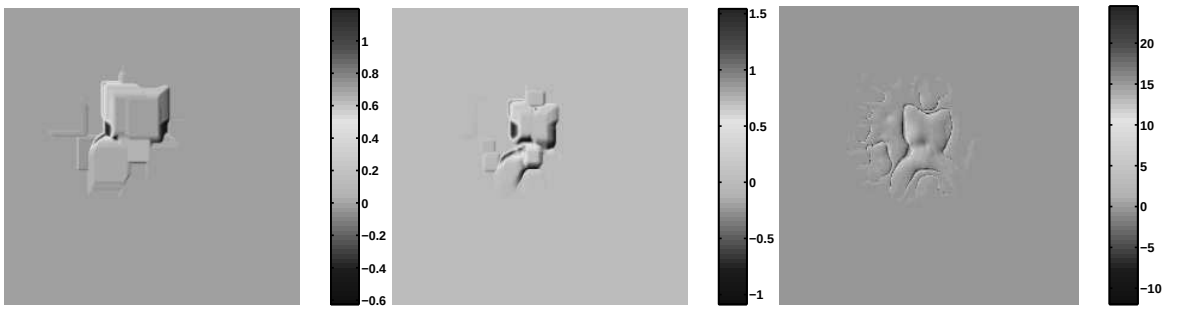
Multi-image registration deals with the problem of registering three or more images simultaneously. In breast cancer therapy patient progress is monitored by periodic UL scans of the breast. Radiologists often register breast images of a patient collected at periodic intervals to monitor tumor growth or recession. One approach is to sequentially register pairs of images from time A to time B, time B to time C and so on. Besides being cumbersome and expensive, this process may lead to the accumulation of registration errors. A less expensive solution that may be able to avoid error accumulation is to register all the sequential scans (A,B,C,...) simultaneously. Image registration methods play an important role in atlas construction [126]. Recently, simultaneous multi-image registration was



(a) I_{ref} (b) $T(I_{tar})$ (c) Difference Image $(I_{tar} - T(I_{tar}))$



(d) $O_{10} = H_\alpha(X, Y)$: 32×32 window (e) $O_{10} = H_\alpha(X, Y)$: 16×16 window (f) $O_{10} = H_\alpha(X, Y)$: 8×8 window



(g) Local $\nabla(H_\alpha) = O_{10} - O_0$: 32×32 window (h) Local $\nabla(H_\alpha) = O_{10} - O_0$: 16×16 window (i) Local $\nabla(H_\alpha) = O_{10} - O_0$: 8×8 window

Figure 7.4.14: B-Spline deformation on MRI images of the brain. (a) Reference image, (b) Warped target (c) True Deformation, (d) $O_{10} = H_\alpha$ as seen with a 32×32 window, (e) 16×16 window and (f) 8×8 window. (g) $\nabla(O) = \nabla(H_\alpha) = O_{10} - O_0$ as seen with a 32×32, (h) 16×16 and (i) 8×8 window.

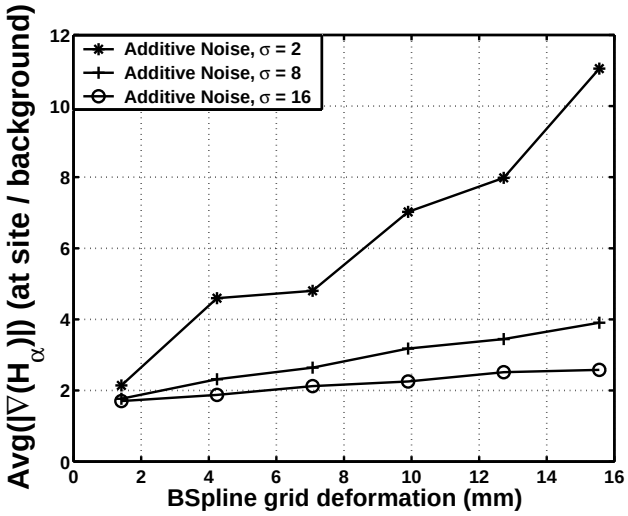
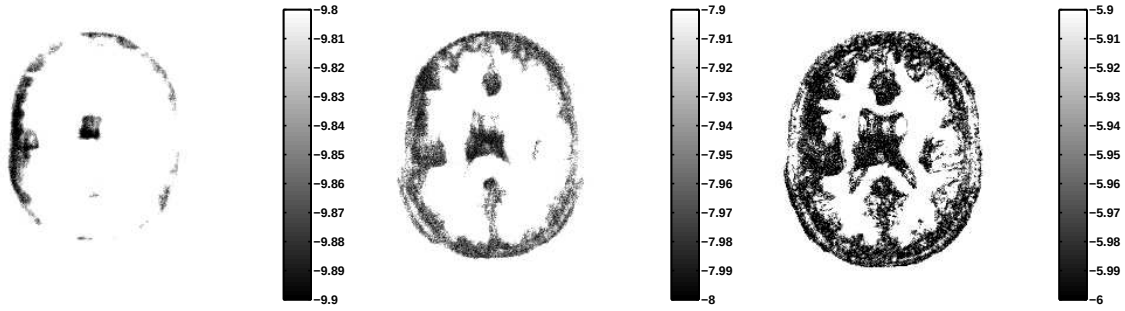


Figure 7.4.15: Ratio of $\nabla(H_\alpha) = \nabla O$ calculated over deformation site v/s background image for smaller deformation spanning $m \times n \geq 8 \times 8$.



(a) Shannon MI $O_{10} = MI(X,Y)$: 32×32

(b) Shannon MI $O_{10} = MI(X,Y)$: 16×16

(c) Shannon MI $O_{10} = MI(X,Y)$: 8×8

Figure 7.4.16: Performance of Shannon MI, computed using pixel intensity histograms, on deformed MRI images: (a) 32×32 window, (b) 16×16 window and (c) 8×8 window.

used to construct an atlas of a small patient database through the use of smart database techniques [18]. Compared to the complexity of their method, entropic graphs offer a relatively straightforward approach to multi-image divergence estimation, as explained here. This section demonstrates the utility of entropic graph methods to simultaneously register three or more images.

7.5.1 Divergence estimation for multi-image registration

Evaluation of divergence for multiple images is straightforward. The α-MI between d-dimensional features $\{\mathcal{X}_i\}_{i=1}^N$, $\{\mathcal{O}_i\}_{i=1}^N$, $\{\mathcal{Y}_i\}_{i=1}^N$ extracted from three images, I_1, I_2, I_3, respectively is an extension of Equation 4.12 as follows:

$$\widehat{\alpha MI} = \frac{1}{\alpha - 1} \log \frac{1}{n^\alpha} \sum_{i=1}^{n} \left(\frac{e_i(x \times o \times y)}{\sqrt{e_i(x) e_i(o) e_i(y)}} \right)^{3\gamma}, \tag{7.4}$$

where $e_i(x \times o \times y)$ is the distance from the point $z_i = [x_i, o_i, y_i] \in \mathbb{R}^{3d}$ to its nearest neighbor in $\{Z_j\}_{j \neq i}$ and $e_i(x)$ $(e_i(o))$ $(e_i(y))$ is the distance from the point $x_i \in \mathbb{R}^d, (o_i \in \mathbb{R}^d), (y_i \in \mathbb{R}^d)$ to its nearest neighbor in $\{X_j\}_{j \neq i} (\{\mathcal{O}_j\}_{j \neq i}) \{Y_j\}_{j \neq i}$ respectively.

Similarly, building on Equation 4.11 α-GA can be estimated between one reference and two target images as follows:

$$\begin{aligned} \widehat{\alpha D_{GA}} &= \frac{1}{\alpha - 1} \log \frac{1}{3n} \sum_{i=1}^{3n} \min\{r_j\}_{j=1}^3 \\ r_1 &= \min \left\{ \left(\frac{e_i(o)}{e_i(x)} \right)^{\gamma/2}, \left(\frac{e_i(x)}{e_i(o)} \right)^{\gamma/2} \right\}, \\ r_2 &= \min \left\{ \left(\frac{e_i(x)}{e_i(y)} \right)^{\gamma/2}, \left(\frac{e_i(y)}{e_i(x)} \right)^{\gamma/2} \right\}, \\ r_3 &= \min \left\{ \left(\frac{e_i(y)}{e_i(o)} \right)^{\gamma/2}, \left(\frac{e_i(o)}{e_i(y)} \right)^{\gamma/2} \right\}, \end{aligned} \tag{7.5}$$

where $e_i(x)$, $e_i(o)$ and $e_i(y)$ are the distances from a point $z_i \in \{\{x_i\}^i, \{o_i\}^i, \{y_i\}^i\} \in \mathbb{R}^d$ to its nearest neighbor in $\{\mathcal{X}_i\}_i$, $\{\mathcal{O}_i\}_i$ and $\{\mathcal{Y}_i\}_i$, respectively. Here, as above $\alpha = (d - \gamma)/d$.

Shannon MI can be estimated using pixel features by extending Equation 3.7 to histogram estimates of the joint pdf in three dimensional space as follows:

$$\widehat{\alpha MI} \stackrel{\text{def}}{=} \frac{1}{\alpha - 1} \log \sum_{x,o,y=0}^{255} \hat{f}_{0,1}^{\alpha}(x,o,y) \left(\hat{f}_x(x) \hat{f}_o(o) \hat{f}_y(y) \right)^{1-\alpha}. \tag{7.6}$$

In Equation 7.6 we assume 8-bit gray level, $\hat{f}_{x,o,y}$ denotes the joint intensity level "coincidence histogram"

$$\hat{f}_{x,o,y}(x,o,y) = \frac{1}{MN} \sum_{k=1}^{MN} I_{x_k,o_k y_k}(x,o,y), \tag{7.7}$$

and $I_{x_k,o_k y_k}(x,o,y)$ is the indicator function equal to one when $(x_k, o_k, y_k) = (x,o,y)$ and equal to zero otherwise.

Equation 7.7 requires building a histogram in the three dimensional joint space of the three images. Generalizing to N images, it can easily be seen that a N-dimensional histogram would be required t estimate Shannon MI using the histogram plug-in method. As discussed in Section 3.7, the curse of dimensionality restricts the estimation of Shannon MI in higher dimensions. On comparison with Equations 7.4 and 7.6 it is seen that estimation of α-MI and α-GA do not suffer from this curse-of-dimensionality since the complexity of the kNN graph grows only linearly in the dimension.

In the following section, the performance of entropic graph based divergence estimates of α-MI and α-GA is compared with traditional histogram estimation techniques of Shannon MI.

7.5.2 Quantitative performance evaluation in multi-image registration

The methods used to evaluate performance of divergence estimates for the two-image case are extended to three images. The database of UL images is divided, as before, into training and testing sets. 64D ICA are estimated on the training set and used as features for registration. Test images are extracted from each volumetric scan in the test

dataset. A $\pm$5mm depth directional distance separates the reference image I_{ref} from the two target images I_{tar_1} and I_{tar_2}. ICA basis coefficient features are extracted from the reference and target images using the standard sub-block projection technique, as before. Registration performance is evaluated over rotational deformation within the range $\pm 16^{\circ}$. Figure 7.5.17 shows an example registration scenario where the reference images is shown to be sandwiched between two target images that are rotated.

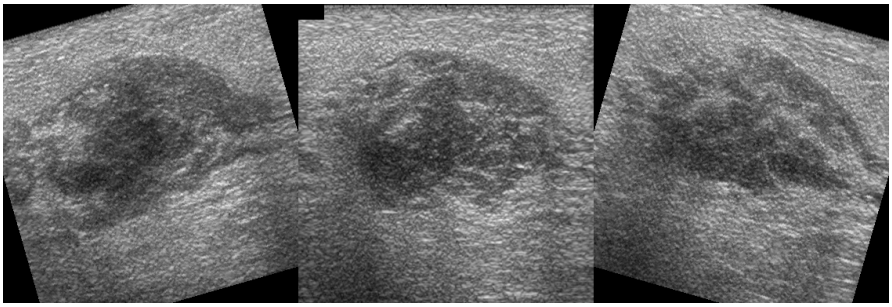


Figure 7.5.17: Multi-image registration example illustrated using three UL images of the breast where the reference image is sandwiched between two target images that are rotated $\pm 16^{\circ}$ respectively.

In Figure 7.5.18 shows the registration performance of the 16 test image sets. Mis-registration error is measured as the sum of mean-squared mis-registration errors for each of the target images, and can hence vary from 0° to 32°. The SNR in all the images is progressively decreased by adding truncated uncorrelated Gaussian noise. Mean mis-registration error is obtained by Monte-Carlo simulations over 30 different noise realizations on each of the 16 image. Thus, every point in the graph is the mean error over 480 measurements. Standard error bars are as shown.

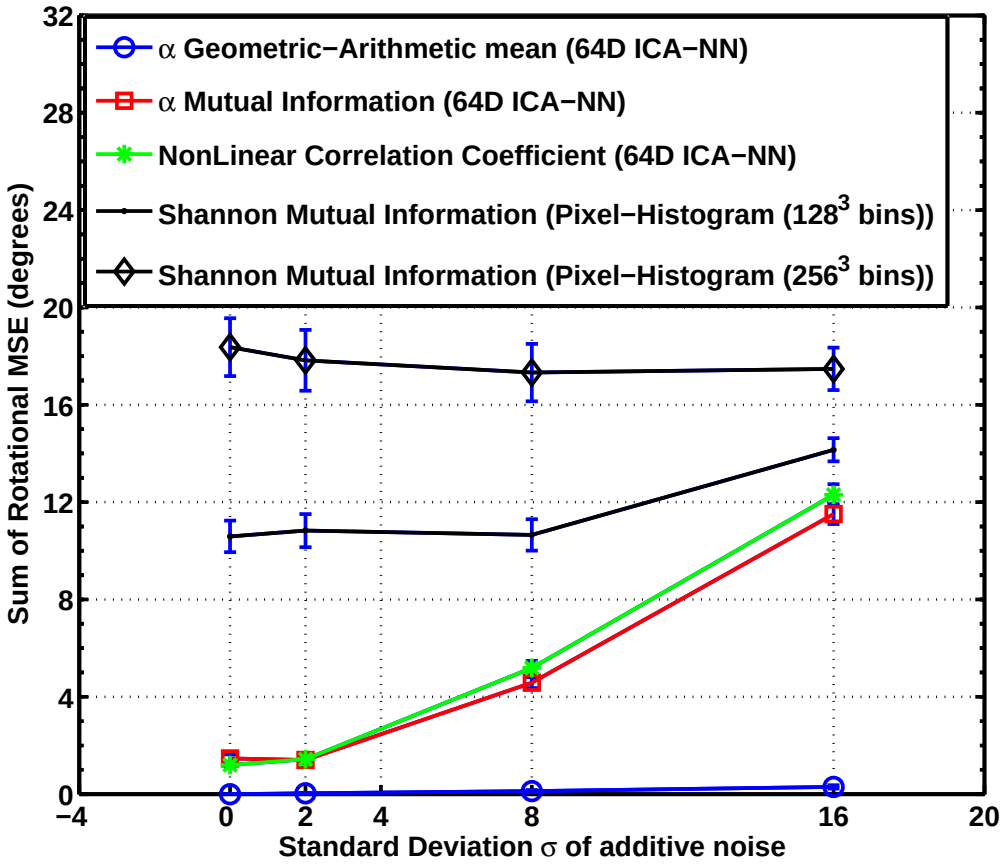


Figure 7.5.18: Quantitative performance comparison of divergence estimates while simultaneously registering three UL images of the breast. Plot shows sum of rotational mean-squared registration errors for each of the target images using 64D ICA feature vectors and α-GA, α-MI, NLCC matching functions. Shannon MI calculated using different intensity histogram bin sizes is also shown. These plots were obtained from Monte Carlo trials consisting of adding i.i.d. Gaussian distributed noise to the images prior to registration. 480 repeated experiments were conducted from within the breast UL volumetric database of 16 breast cancer patients undergoing therapy. The three slices to be registered are spatially separated by 5mm. Search was restricted to a maximum rotation angle of $\pm 16^{\circ}$. The confidence intervals represent unit standard error in the computation of the mean of the interval.

7.6 Discussion and Future Work

This section aims to provide some insight into the performance of different measures of divergence and features for the various applications studied in this thesis. This section will also identify some other extensions to this work like the multi-image registration example studied briefly here.

The first component of this study was to extend histograms to higher dimensions. There has been limited success in this direction. Besides the increased complexity and sparsity of the histogram in higher dimensions, it was also difficult to identify reliable discrete features. Tag features show only modest improvements of standard scalar single-pixel methods. Tags are designed for binary images and extension to higher dimensions is not straightforward. Discretization of feature vectors, while viable, has difficulties when creating a joint space.

The performance of ICA and wavelet basis demonstrate that there is potential benefit in extending features to higher dimensional spaces. When measured on single-pixel spaces, graph methods such as αJensen in UL registration example and αMI in satellite image registration example show performance similar to that of single-pixel Shannon MI. Performance advantage is derived from the use of higher-dimensional descriptive features and entropic graphs enable use of such features whereas histograms do not. Non-parametric kernel density estimators may be used for "plug-in" purposes, but their performance is expected to mimic that of histogram plug-in for higher dimensional spaces. A vital future direction of research in image registration would thus be the identification and use of other features in image registration. A comprehensive study examining various modalities, registration problems and factors that would influence choice of features is warranted.

Several extensions to previous work on entropy estimation for image registration have

been presented here. The new kNN estimators of the αMI and αGA have the advantage of invariance to reparameterization of the feature space. While convergence results for the kNN divergence estimators were not provided there is circumstantial theoretical evidence that they do converge. Furthermore, the numerical evaluations show that these divergence estimators outperform previous approaches to image registration. This thesis also introduced the Friedman-Rafsky (FR) multivariate run test, which is an estimator of Henze-Penrose divergence, as a new matching criterion for image registration. Of course, as compared to our kNNG divergence estimators, the FR method has the advantage of proven theoretical convergence but has the disadvantage of higher runtime complexity.

The performance of αGA and Henze-Penrose have exceeded those of other divergence measures. This thesis hypothesizes that the combination of low-dimensional complexity through the exclusive use of marginal spaces and invariance to transformations has led to superior noise performance and robustness in these measures as compared to others. Unlike the other metrics, the αJensen difference is not invariant to reparameterization, which explains its relatively poor performance for large rms noise.

Several other directions of future work can be identified. From a registration perspective, a study of the performance of different features is vital. It is also important to extend this work to 3D volumetric images that would be registered across time or across different patients. While registering several patients simultaneously to create an atlas, multi-image registration is required. This another interesting extension to this work. Approximate NN methods that were briefly studied here, should be extended after developing an understanding of the error-computational complexity trade-offs for different measures of divergence. The convergence properties of kNN estimators of αMI, αGA and NLCC remain unexamined and should be explored.

CHAPTER VIII

Conclusions

This work was motivated by a desire to initiate development of robust and accurate image registration systems for medical imaging and other image matching applications. A natural extension to current information theoretic methods such as single pixel histogram based estimates of MI was to incorporate higher dimensional features. An attempt to build histograms in higher dimensional spaces using quantized pixel neighborhoods, however, yielded only marginal improvements over pixel features.

Based on the work of B. Ma [82], this thesis then explored the use of entropic graph methods as a means to extend matching to higher dimensions without first estimating density in the high dimensional space. The key to this work was the use of entropic graphs to estimate Rényi α-entropy in higher dimensional feature spaces directly. The MST length functional displays asymptotic convergence to the Rényi α-entropy of the underlying distribution on which it is built. The MST entropy estimator was used successfully to estimate the α-Jensen divergence estimator [82]. Here the MST based estimate of α-Jensen difference was used for ultrasound image registration. Local ICA projection coefficients in 64D space provided reliable estimates of α-Jensen difference. The kNN graph whose length functional has similar convergence properties as the MST was then used for entropy estimation and image registration purposes. The kNN has a complexity that is an order of

magnitude lower than that of the MST. Subsequent registration efforts were undertaken with the kNN graph over ICA feature coefficients. Quantitative comparisons showed that both the entropic graph-based higher dimensional estimate of α-Jensen difference had robust performance and lower mis-registration errors in test cases of UL image registration as compared with the standard Shannon MI computed using pixel histograms.

The initial results of UL breast image registration provided impetus for further research in other divergence measures that could be reliably estimated in higher-dimensions through the use of entropic graphs. Subsequent research led to the development of Henze-Penrose affinity which is based on the Friedman-Rafsky higher-order test statistic. The HP affinity is estimated directly from the MST built in higher dimensional feature spaces. The α-GA mean affinity, α MI and non-linear correlation coefficient were then developed and used for image registration. These latter divergence measures were computed by approximating a local Voronoi-cell density using an NN graph.

A requirement to accomplish high dimensional image matching is a feature space adapted to the image characteristics. Higher dimensional features used for this work include those based on independent component analysis, discrete cosine transforms and multidimensional wavelet image analysis. ICA is a data-driven process of estimating a statistically independent basis and is used to extract features from ultrasound images. Divergence was estimated in wavelet coefficient spaces using entropic-graph methods for registration of geo-satellite imagery.

Methods for computing the MST and kNN graph were explored with a view toward reducing time-memory costs. A disc-based approach is used to pick candidate nodes to build the MST. Rapid MST construction was achieved for over 100,000 feature samples residing in 64 dimensional space. This acceleration allowed evaluation of entropic graphs methods over several hundreds of images with a variety of feature spaces. A kd-tree ap-

proach was used to rapidly construct the kNN graph and further accelerate the estimation process. Computational complexity is certainly less of a hurdle in registration with graph methods due to contributions made here.

This thesis extended the application of entropic graphs to new applications like Ultrasound breast image registration, multisensor satellite image registration, simultaneous multi-image registration, MRI small volume registration and matching of human face images. Ultrasound imaging is widely used to detect malignant breast lesions. However, compressibility of the breast tissue, high specular imaging noise, low resolution and small field-of-view have complicated past registration efforts. Through the use of graph-based registration methods and a data-driven features extraction process, lower registration errors in 2D test cases were seen. Numerical performance comparisons among the metrics and features were aimed at identifying the algorithms that best discriminate between rotationally aligned and misaligned images. The rotational deformations were local in nature since the thesis did not intend to focus on iterative optimization techniques. The discrimination ability for local rotational deformations provides a good comparison of the accuracy of registration for more general image deformations. Sensitivity and robustness to noise was also evaluated. Multimodal aerial images of the earth taken by geo-stationary satellites were registered using graph methods and higher-dimensional features. In registering multiple images to form a representative atlas, high dimensional estimation of density is performed. Applications such as tracking micro-calcifications in the breast and tumor in the brain were addressed with entropic graph methods which provide remarkable estimates of entropy and divergence even with a small number of features. A multimodal face matching example was performed to demonstrate the versatility of entropic graphs to perform other image matching tasks.

An exciting extension of this work is in registration of multiple images. Multiple im-

ages could be registered simultaneously to form an atlas. Multi-image registration could also be used to simultaneously register time-sampled imagery such as those acquired during periodic UL examination for cancer detection and management.

APPENDIX

APPENDIX

Computer code for divergence estimation

This appendix to the thesis contains MATLAB ™Mex program code used to construct the MST and kNN graph and calculate graph based estimates of α-Jensen difference divergence, α-MI, α-GA mean divergence and Henze-Penrose affinity. The programs were written in standard ANSI C within the mex programming guidelines provided in [2].

A.1 Program to construct MST using Kruskal algorithm

Begining of Kruskal MST program

```
/*****************************************************************************

  File: mst.c
  Rev:  a-1
  Date: 09/27/2001

  Copyright (c) 1993, 2001 by David M. Warme

*****************************************************************************
  Program to construct MST
  Modified Code Copyright (c) 2001- 2005, by Huzefa Neemuchwala
    hneemuch@umich.edu
    Please read this preamble to address memory errors before contacting me.
*****************************************************************************

  Modification Log:

  a-1:  09/27/2001  warme
    : Created from pieces of GeoSteiner version 3.1.

  a-2:  10/12/01    Huzefa Neemuchwala
  Can now read point set from file directly.
  Changed the sort to a fast qsort using C library function
  a-3 01/10/02    Huzefa Neemuchwala
  Reduced search radius allows us to locally relax "curse of dimensionality"
  Now works extremely fast!
  Fix a radius within which points are accepted to form edges.
  So for each point roughly M nearest neighbors are accepted by
  fixing a suitable radius. So from N^2 edges (for N points) we
  now have N*M edges (M<<N^2).
  Time reduced for sorting.
  Memory requirements fall to a fraction, and so huge point sets
  are now possible without writing to files.
  a-4 02/11/02    Huzefa Neemuchwala
  Linearized the entire algorithm by linearizing time/memory for edge
  selection also.
  a-5 07/25/2003  Huzefa Neemuchwala
  Mex written.
  Code has ability to give out the edge labels the full MST.
  You may need to alter the following line if you run into memory problems

  Note: However you should do that in conjunction with reducing your search radius
  The line:     E = mxMalloc (npoints1*10000*sizeof(struct edge)); in function 'principal'
  allocates memory for the edge matrix.
  Contact Huzefa Neemuchwala, hneemuch@umich.edu

*****************************************************************************/

/*Include files*/
#include <math.h>
#include "mex.h"
#include "string.h"
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

```

```
/*Some preprocessor definitions*/
#define NOT !
#define AND &&
#define OR  ||
#define EQ  ==
#define NE  !=

#define NEWA(n, type) ((type *) new ((size_t) ((n) * sizeof (type))))

/*Define an edge*/
struct edge {
  int   p1;   /* First endpoint of edge. */
  int   p2;   /* Second endpoint of edge. */
  double    len;    /* Length of edge. */
};


/*
 * This is the so-called "Disjoint Set Union-Find" data structure.
 * The operations are "makeset", "find", and "union".
 *
 * See chapter 2 of "Data Structures and Network Algorithms", by
 * Robert Endre Tarjan, SIAM, 1983 for complete details.
 */

struct dsuf {
  int * parent;
  int * rank;
  int set_size;
};


/*
 * Local Routines
 */

static void principal (double, double, double, double *, double *, double *);
static void dsuf_create (struct dsuf *, int);
static void dsuf_destroy (struct dsuf *);
static int  dsuf_find (struct dsuf *, int);
static void dsuf_makeset (struct dsuf *, int);
static void dsuf_unite (struct dsuf *, int, int);
static void fatal (char *);
static int  mst_edge_list (int, int, struct edge *, struct edge *);
static void * new (size_t);
static void sort_edge_list (struct edge *, int);

/*
 * The main routine.  Read in points, compute MST, output it.
 */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{

  double npoints, dim;

  double* prmstlength;
  double threshold, threshold2;
  /*Uncomment next line to include edges in output*/
  /*  double *edges, *edgelabels;*/
  double *V;
  double *SL;

  /* Check for proper number of arguments. */
  if (nrhs != 5) {
    mexErrMsgTxt("Five inputs required : mstmex(rrw,indexs,N,dim,threshold)");
  } else if (nlhs > 1) {
    mexErrMsgTxt("Only one output argument");
  }

  /* Assign pointers to each input */
  /* Use mex syntax */
  /* matrices for each time point */
  V = (double *)mxGetPr(prhs[0]);
  SL = (double *)mxGetPr(prhs[1]);

  /* parameters for analysis */
  npoints = *mxGetPr(prhs[2]);
  dim = *mxGetPr(prhs[3]);
  threshold = *mxGetPr(prhs[4]);
  threshold2 = threshold*threshold;

  /* Create matrix for the return argument. */
  plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);
  /*the following lines could be modified to output the MST edges also*/
  /*plhs[3] = mxCreateDoubleMatrix(npoints,2,mxREAL);
        plhs[4] = mxCreateDoubleMatrix(npoints,2,mxREAL);*/

  /* Assign pointer to the output */
  prmstlength = (double *)mxGetPr(plhs[0]);

   /*the following lines could be modified to output the MST edges also*/
   /*edges = (double *)mxGetPr(plhs[3]);
        edgelabels = (double *)mxGetPr(plhs[4]);*/

  /*the following lines could be modified to output the MST edges also*/
  principal(npoints,dim,threshold2,V,SL,prmstlength); /*,edges,edgelabels);*/
}

void
principal (
double npoints, double dim, double threshold2,
double *V, double *SL, double *prmstlength)
 /*the following lines could be modified to output the MST edges also*/
/*,
double *edges, double *edgelabels
)*/

{
int   i, j, k;
int   q,qctr,flagt,ind,nind;
int   nedges;
struct edge * E;
double *  pi;
double *  pj;
struct edge * ep;
double    dist, length, dist1, delta;
struct edge * solution;
int     ctr1=0;
int dim1, npoints1;
/*  printf("Total points %g;  Dimensions %g;  Threshold %g\n",npoints,dim, threshold2);*/
  dim1 = (int) dim;
  npoints1 = (int) npoints;
  nedges  = npoints1 * (npoints1 - 1) / 2;
```

```

/*Construct tree using full search if number of edges less than 500...tentative*/
if(npoints<3000)
{
 nedges  = (int)(npoints * (npoints - 1) / 2);

        /* Read in the points. */
        E = (struct edge *) mxMalloc( (nedges) * sizeof(struct edge) );

        /*E = NEWA (nedges, struct edge);*/
        ep = E;
        for (i = 0; i < npoints1; i++) {
                pi = &V [i * dim1];
                for (j = i+1; j < npoints1; j++) {
                        pj = &V [j * dim1];
                        /* Compute distance between points I and J. */
                        dist = 0.0;
                        for (k = 0; k < dim1; k++) {
                                delta = pi [k] - pj [k];
                                dist += (delta * delta);
                        }
                        /* Store triple (I,J,DISTANCE). */
                        ep -> p1        = i;
                        ep -> p2        = j;
                        ep -> len       = sqrt (dist);
                        ++ep;
                }
        }

        /* Allocate buffer to hold solution. */
        /*solution = NEWA (npoints - 1, struct edge);*/
  solution = (struct edge *)mxMalloc( (npoints1-1) * sizeof(struct edge) );

        k = mst_edge_list (npoints, nedges, E, solution);

        /*printf ("\nMinimum Spanning Tree (FS):");commented this one*/
        ep = solution;
        length = 0.0;
        for (i = 0; i < k; i++) {
                length += ep -> len;
                ep++;
        }
        *prmstlength = length;
        /*printf ("MST length %g\n", *prmstlength);*/
}

else {
/*****************************************************
Radius approach
Select points for edge computation using radius approach.
*****************************************************/
/*The next line is very important. It does the memory allocation for the edge matrix.
Reduce allocation if you run into memory problems. Of course you could then run into an incomplete MST!*/
  E = (struct edge *) mxMalloc((npoints1*3000)*sizeof(struct edge));
    /*E = NEWA (160000*10000, struct edge);*/
  ep = E;


  for (q=0; q< npoints1-1; q++) {
      flagt=1;
      ind = (int) SL[q];
      qctr=1;
      do {
        nind = (int) SL[q+qctr];
        pi = &V [ind * dim1];
        pj = &V [nind * dim1];
        dist1=(pi[0]-pj[0])*(pi[0]-pj[0]);
        dist = 0.0;
        for (k = 0; k < dim1; k++) {
          delta = pi [k] - pj [k];
          dist += (delta * delta);
          if (dist > threshold2)
          break;
          }
        if (dist <= threshold2){
          ep -> p1  = ind;
          ep -> p2  = nind;
          ep -> len = sqrt(dist);
          ++ep;
          ctr1++;
          }
        if (dist1 > threshold2) {flagt=0;}
        if ((q+qctr) == (npoints-1)) {flagt=0;}
        qctr++;
        }
      while (flagt==1);
      }

/************************************************/


  nedges=ctr1;
  /*printf("ctr1 is %d\n",ctr1);*/

  /* Allocate buffer to hold solution. */
  solution = (struct edge *)mxMalloc((npoints1-1)*sizeof(struct edge));
  /*solution = NEWA (npoints1 - 1, struct edge);*/
  k = mst_edge_list (npoints1, nedges, E, solution);
  ep = solution;
  length = 0.0;
  for (i = 0; i < k; i++) {
      /*edges[i] = (double)(ep -> p1);
      edges[i+npoints] = (double)(ep -> p2);

      edgelabels[i] = (double)(ep -> startdensity);
      edgelabels[i+npoints] = (double)(ep -> enddensity);*/

    length += ep -> len;
    ep++;
  }
  *prmstlength = length;

  printf("Minimum Spanning Tree LENGTH = %g\n", length);
  /*printf("FR metric = %d,%g\n",FRmetric,*prfrmetric);
  printf("Hero-Costa FR length = %g\n",FRlength);*/
  /*free(solution);
  free(E);
  free(ep);*/
}
}

/*
 * This routine computes the MST of a given list of edges.
```

```
 */

  static
  int
mst_edge_list (

int     n,    /* IN - number of vertices */
int     nedges,   /* IN - number of edges */
struct edge *   edge_list,  /* IN - list of edges */
struct edge *   edges   /* OUT - MST edge list */
)
{
int   i;
int   mst_edge_count;
int   components;
int   max_vert;
struct edge * ep;
struct edge *pp;
struct edge * ep_endp;
int   root1;
int   root2;
struct dsuf sets;
int     ctr =0;
double    perc;

    sort_edge_list (edge_list, nedges);

    /* Don't assume that the vertex numbers are well-behaved, */
  /* except that they must be non-negative.  We do a quick scan */
  /* to determine the largest vertex number and then allocate */
  /* a union-find data structure large enough to handle it.  Note */
  /* that we then use this union-find data structure in a   */
  /* completely sparse way -- we only ever access set items for */
  /* vertices that are named by an edge.       */

  max_vert = 1;   /* avoid zero-size union-find... */
  ep = edge_list;
  for (i = 0; i < nedges; i++, ep++) {
    if (ep -> p1 > max_vert) {
      max_vert = ep -> p1;
    }
    if (ep -> p2 > max_vert) {
      max_vert = ep -> p2;
    }
  }

  dsuf_create (&sets, max_vert + 1);

  /* Note that it is not a problem to "makeset" a vertex more */
  /* than once...              */
  ep = edge_list;
  for (i = 0; i < nedges; i++, ep++) {
    dsuf_makeset (&sets, ep -> p1);
    dsuf_makeset (&sets, ep -> p2);
  }

  components = n;
  mst_edge_count = 0;
  ep = edge_list;
  ep_endp = (ep + nedges);

  while (components > 1) {
    ctr++;
    if (ep >= ep_endp) {
      /* Ran out of edges before MST complete! */
      /*printf("Ran our of edges before tree complete!");commented this one
      printf("\nHowever, I have managed to get you almost all edges");*/
      perc=(mst_edge_count*100.0)/(n-1);
      printf("\nthe number of edges is %d which is %g percent of all edges \n",mst_edge_count,perc);/*commented this one*/
      /*if (mst_edge_count > (n-(0.2*n)) )
        {*/
                              pp=edge_list+ctr-1;
        /*printf("\nctr is %d and msx dist is %g\n",ctr,pp->len);*/
                              /*dsuf_destroy (&sets);*/
            return (mst_edge_count);
        /*}*/
      /*else{fatal ("mst_edge_list: Bug 1 Ran out of edges before tree complete Less than 90percent of all edges gleaned.");}*/
    }
    root1 = dsuf_find (&sets, ep -> p1);
    root2 = dsuf_find (&sets, ep -> p2);
    if (root1 NE root2) {
      dsuf_unite (&sets, root1, root2);
      *edges = *ep;
      ++edges;
      ++mst_edge_count;
      --components;
    }
    ++ep;
  }
pp=edge_list+ctr;
perc=100;

/*printf("Longest Edge was %g\t",pp->len);      commented this one
printf("Use this distance instead of threshold!");  commented this one*/
  /*dsuf_destroy (&sets);*/

  return (mst_edge_count);
}

/*
 * This routine sorts the given edge list in INCREASING order by edge length.
 */

  static
  void
sort_edge_list (

struct edge *   a,  /* IN/OUT - array of edges to be sorted. */
int     n /* IN - number of elements in array. */
)
{
int   h;
struct edge tmp;
double    key;
struct edge * p1;
struct edge * p2;
struct edge * p3;
struct edge * p4;
struct edge * endp;

  endp = &a [n];

  for (h = 1; h <= n; h = 3*h+1) {
  }
```

```
430
431       do {
432         h = h / 3;
433         p4 = &a [h];
434         p1 = p4;
435         while (p1 < endp) {
436           tmp = *p1;
437           key = tmp.len;
438           p2 = p1;
439           for (;;) {
440             p3 = (p2 - h);
441             if (p3 -> len <= key) break;
442             *p2 = *p3;
443             p2 = p3;
444             if (p2 < p4) break;
445           }
446           *p2 = tmp;
447           ++p1;
448         }
449       } while (h > 1);
450     }
451
452     /*
453      * This routine creates a collection of N disjoint sets.  They are left
454      * uninitialized so that a sparse collection can be accessed quickly.
455      */
456
457       static
458       void
459     dsuf_create (
460
461     struct dsuf * dsp,    /* IN/OUT - sets to create */
462     int   n   /* IN - number of disjoint sets */
463     )
464     {
465       if (n <= 0) {
466         fatal ("dsuf_create: Bug 1.");
467       }
468
469       dsp -> set_size  = n;
470       dsp -> parent   = (int *)mxMalloc (n*sizeof( int));
471       dsp -> rank   = (int *)mxMalloc (n*sizeof( int));
472     }
473
474
475     /*
476      * Destroy the given collection of disjoint sets.
477      */
478
479       static
480       void
481     dsuf_destroy (
482
483     struct dsuf * dsp   /* IN - sets to destroy */
484     )
485     {
486       /*free ((char *) (dsp -> rank));
487       free ((char *) (dsp -> parent));*/
488
489       dsp -> set_size = 0;
490       dsp -> parent = NULL;
491       dsp -> rank = NULL;
492     }
493
494     /*
495      * This routine makes a single disjoint set for item "i".
496      */
497
498       static
499       void
500     dsuf_makeset (
501
502     struct dsuf * dsp,    /* IN - collection of sets */
503     int   i   /* IN - item to make into a disjoint set */
504     )
505     {
506       if ((i < 0) OR (i >= dsp -> set_size)) {
507         /* Item out of bounds. */
508         fatal ("dsuf_makeset: Bug 1.");
509       }
510       dsp -> parent [i] = i;
511       dsp -> rank [i]   = 0;
512     }
513
514     /*
515      * This routine "unites" two sets that were previously disjoint.  I and J
516      * must be the "canonical" member of each disjoint set (i.e. they must
517      * each be the output of a "find" operation), and must be distinct.
518      *
519      * We perform the "union by rank" heuristic here.
520      */
521
522       static
523       void
524     dsuf_unite (
525
526     struct dsuf * dsp,    /* IN - collection of sets */
527     int   i,    /* IN - first set to unite */
528     int   j   /* IN - second set to unite */
529     )
530     {
531     int   ri;
532     int   rj;
533
534       if ((i < 0) OR (i >= dsp -> set_size)) {
535         /* Item I is out of range. */
536         fatal ("dsuf_unite: Bug 1.");
537       }
538       if ((j < 0) OR (j >= dsp -> set_size)) {
539         /* Item J is out of range. */
540         fatal ("dsuf_unite: Bug 2.");
541       }
542       if (i EQ j) {
543         /* Attempt to unite I with I. */
544         fatal ("dsuf_unite: Bug 3.");
545       }
546
547       ri = dsp -> rank [i];
548       rj = dsp -> rank [j];
549
550       if (ri EQ rj) {
551         /* Both subtrees have the same maximum depth.  We */
552         /* arbitrarily choose I to be underneath J.  The rank */
553         /* of J must then increase.       */
```

```
    dsp -> parent [i] = j;
    dsp -> rank [j]   = rj + 1;
  }
  else if (ri > rj) {
    /* Tree I is (probably) deeper.  Putting J underneath */
    /* will not increase I's rank.        */
    dsp -> parent [j] = i;
  }
  else {
    /* Tree J is (probably) deeper... */
    dsp -> parent [i] = j;
  }
}

/*
 * This routine, given a member I of one of the disjoint sets A, will
 * choose a cannonical member J of set A and return it.  Until set A gets
 * united with some other set, find (I) will always return the same J.
 *
 * This routine performs the "path compression" heuristic.
 */

  static
  int
dsuf_find (

struct dsuf * dsp,    /* IN - collection of sets */
int   i   /* IN - item to find cannonical item for */
)
{
int   j;
int   k;

  /* Yes, I know this routine is very elegent when coded  */
  /* recursively...  Here's the iterative version.  */

  j = dsp -> parent [i];
  if (i EQ j) {
    /* A cannonical element has itself as parent. */
    return (i);
  }

  /* We must search up the tree -- and compress when done... */
  for (;;) {
    k = dsp -> parent [j];
    if (j EQ k) break;
    j = k;
  }

  /* Now compress the path (make all items in chain point directly */
  /* at the root K) -- we never have to do this search again!  */
  while (i NE k) {
    j = dsp -> parent [i];
    dsp -> parent [i] = k;
    i = j;
  }

  return (k);
}

/*
 * This routine displays a fatal message and then dies!
 */

  static
  void
fatal (

char *    msg   /* IN - message to display. */
)
{
  (void) fprintf (stderr, "%s\n", msg);
  (void) fflush (stderr);
  abort ();
}

/*
 * This routine performs all dynamic memory allocation for the program.
 * We test for out of memory condition here.
 */

  static
  void *
new (

size_t    size    /* IN - size of chunk in bytes. */
)
{
void *    p;

  if (size EQ 0) {
    /* Avoid implementation-defined bahavior of malloc! */
    size = 1;
  }

  p = mxMalloc (size);
  if (p EQ NULL) {
    (void) fprintf (stderr, "Out of memory!\n");
    exit (1);
  }

  return (p);
}
```

```
678
679
680
681
682
683
```

End of Kruskal MST program

A.2 Program to construct MST using Prim algorithm

Begining of Prim MST program

```
1
2   /*************************************************************************
3     Program to construct MST using Prim's algorithm
4     File: mstprimmex.c
5     Rev:  a-1
6     Date: 06/01/2004
7     Copyright (c) 2004 by Huzefa Neemuchwala, All Rights Reserved
8     hneemuch@umich.edu
9     This code uses kd tree algorithm of Bentley.
10    Thank you Anonymous for part of kd tree code
11       See MX_mstprim.m for usage
12  *************************************************************************/
13  #include <math.h>
14  #include "mex.h"
15  #include "string.h"
16  #include <stddef.h>
17  #include <stdio.h>
18  #include <stdlib.h>
19  #include <assert.h>
20  #include "matrix.h"
21  #include "optkd.h"
22  #include "nn.h"
23
24  #define sgn(x)  ( x>0 ? 1 : (x < 0 ? -1 : 0) )
25  #define TESTTREE(PP)  ((PP) = (optkdNode *)mxMalloc(sizeof(optkdNode)))
26  #define NEWTREE(PP)  if (TESTTREE(PP)==NULL) \
27                         {printf("memory error\n");return;}
28  struct  Heap
29  {
30    double  key;
31    int  idx;
32    int  elt;
33  };
34
35
36  #define  heap_key( p )     ( _heap[p].key )
37  #define  heap_idx( p )     ( _heap[p].idx )
38  #define  heap_elt( k )     ( _heap[k].elt )
39
40  #define  in_heap( p )    ( heap_idx(p) > 0 )
41  #define  never_seen( p ) ( heap_idx(p) == 0 )
42  /*
43   * Local Routines
44   */
45
46  static void principal (double, double, double *, double *, double*, double);
47  void  allocate_heap( int );
48  void  heap_init( int );
49  void  heap_insert( int, double);
50  void  heap_decrease_key( int, double );
51  int  heap_delete_min();
52  double dist (double**, int, int, int);
53
54  /***************    kNN functions  ************************************************/
55  void Selection(double **, int, int, int, int,int*);
56  int findmaxspread(int, int, int, double **,int*);
57  optkdNode *BuildkdTree(double**, int, int, int,int*);
58  optkdNode *BuildOptTree(double **, int, int,int*);
59  void rnnEuclidean(optkdNode *, double *, double **, int, int, int*,double *, int *);
60  int *kdOptNNQuery(double **, int, double *, int, int, optkdNode *, int, int*,double*,int*);
61  void PQupheap(double *, int *, int);
62  void PQinsert(double, int, double *, int *);
63  void PQdownheap(double *, int *, int, int);
64  void PQreplace(double, double *, int *, int);
65  void optInRegion(optkdNode*,int,double **,double **,int*, double*,int*);
66  void optAddRegion(optkdNode*,int,double **,double **,int*, double*,int*);
67  int optBoundsIntersectRegion(double*,double**,int);
68  int optBoundsContainsRegion(double*,double**,int);
69  void optRangeSearch(optkdNode*,double**,int,double**,double*,int*, double*,int*);
70  int *kdOptRectQuery(optkdNode*,double**,int,double**,int *, double*,int*);
71  int Count=0;
72  /*************************************************************************************/
73  struct Heap*   _heap;
74
75  int    _max_heap_size = 0;
76  int    _heap_size = 0;
77
78  void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
79                   const mxArray *prhs[])
80  {
81
82    double npoints, dim, threshold;
83    double* edgelengths, *mstedges;
84    double *V;
85
86    /* Check for proper number of arguments. */
87    if (nrhs != 4) {
88      mexErrMsgTxt("Four inputs required : mstprimmex(rrw,N,dim,threshold)");
89    } else if (nlhs > 2) {
90      mexErrMsgTxt("Only two output arguments");
91    }
92
93    /* Assign pointers to each input */
94    /* Use mex syntax */
95    /* matrices for each time point */
96    V = (double *)mxGetPr(prhs[0]);
97
98    /* parameters for analysis */
99    npoints = *mxGetPr(prhs[1]);
100   dim = *mxGetPr(prhs[2]);
101   threshold = *mxGetPr(prhs[3]);
102   /* Create matrix for the return argument. */
103   plhs[0] = mxCreateDoubleMatrix((int) npoints,1,mxREAL);
104   /*the following lines could be modified to output the MST edges also*/
105   plhs[1] = mxCreateDoubleMatrix((int)(npoints*2),1,mxREAL);
106   /*       plhs[4] = mxCreateDoubleMatrix(npoints,2,mxREAL);*/
107
108   /* Assign pointer to the output */
109   edgelengths = (double *)mxGetPr(plhs[0]);
110   mstedges = (double *) mxGetPr(plhs[1]);
111    /*the following lines could be modified to output the MST edges also*/
112   /*edges = (double *)mxGetPr(plhs[3]);
113         edgelabels = (double *)mxGetPr(plhs[4]);*/
114
115  /*the following lines could be modified to output the MST edges also*/
116   principal(npoints,dim,V,edgelengths,mstedges,threshold); /*,edges,edgelabels);*/
117
118 }
119
120 void
```

```
principal ( double npoints, double dim, double *V, double *edgelengths, double *mstedges, double threshold)
{
  int i,n, nn1, k, j, kk, oct,jj, pp,p;
  int root = 0, kneighbors, dimension /*, posn*/;
  int *Found, *optfound, *perm, *parent;
  double **Points, /* *querpoint,*/ *nndist, **RectQuery;
  double /*boxside = 5.0,*/ d;
  optkdNode *OptkdRoot;

  n = (int) npoints;
  dimension = (int) dim;
  parent = (int*)mxMalloc( n *sizeof(int) );
  RectQuery = (double **)mxMalloc(dimension*sizeof(double *));
  for (k = 0; k < dimension; k++) {
    RectQuery[k] = (double *)mxMalloc(2*sizeof(double));
  }


  _heap = (struct Heap*)mxMalloc((n+1)*sizeof(struct Heap) );
  _max_heap_size = n;

  kneighbors = n+2;
  optfound = (int *) mxMalloc((kneighbors)*sizeof(int));
  nndist = (double *)mxMalloc((kneighbors)*sizeof(double));
  perm  = (int *) mxMalloc(n*sizeof(int));
  Points = (double **)mxMalloc((n)*sizeof(double *));
  for (k = 0; k < n; k++) {
    Points[k] = (double *)mxMalloc(dimension*sizeof(double));
  }

  for (k = 0, i = 0; k < n; k++) {
    for (j = 0; j < dimension; j++, i++) {
      Points[k][j] = V[i];
    }
  }
  OptkdRoot  = BuildOptTree(Points,  n, dimension,   perm);
  _heap_size = 0;
  for( p = 0;  p < n;  p++ ) {
    heap_idx( p ) = 0;
  }
  heap_insert( root, 0 );
  parent[root] = root;
  for( k = 0, pp = 0;  k < n;  k++ ) {
    i = heap_delete_min();
    if( i>=0 && i < n){
      edgelengths[pp] = heap_key(i);
      pp++;
      Count = 0;
      for (kk = 0; kk < dimension; kk++) {
        RectQuery[kk][0] = Points[i][kk] - threshold;
        RectQuery[kk][1] = Points[i][kk] + threshold;
      }
      Found = kdOptRectQuery (OptkdRoot, Points, dimension, RectQuery, optfound, nndist,perm);
      if (Count == 1) {
        printf("Empty NN array! MST will not be complete. Increase threshold.\n");
        parent[i] = -1;
      }
      else {
        for( oct = 1;  oct < Count;  oct++ ) {
          nn1 = Found[oct]; /*oct;*/
          if( nn1 >= 0 && nn1 < n) {
            d = 0.0;
            for (jj = 0; jj < dimension; jj++)
              d += pow(Points[nn1][jj] - Points[i][jj],2);
            d = sqrt(d);
              if( in_heap(nn1) && (d < heap_key(nn1)) ) {
                heap_decrease_key( nn1, d );
                parent[nn1] = i;
              }
              else if( never_seen(nn1) ) {
                heap_insert( nn1, d );
                parent[nn1] = i;
              }
          }
        }
      }
    }
  }
  for( i = 0, j =0;  i < n;  i++ ) {
    if (parent[i] <= n && parent[i] >= 0) {
      mstedges[2*j] = (double) parent[i];
      mstedges[2*j + 1] = (double) i;
      j++;
    }
  }

}


/*****************************************************************************/
/*****************************************************************************/




/*****************************************************************************/
/*
*/

void  allocate_heap( int n ) {

  if( _max_heap_size < n ) {
    _heap = (struct Heap*)mxRealloc( (void*)_heap, (size_t)(n+1)*sizeof(struct Heap) );

    if( ! _heap ) {
      printf( "Cannot reallocate memory in allocate_heap!\n" );
    }
    _max_heap_size = n;
  }
}


void  heap_init( int  n ) {
int  p /*register*/;

  /*allocate_heap( n );*/
  _heap_size = 0;
  for( p = 0;  p < n;  p++ ) {
    heap_idx( p ) = 0;
  }

} /* END heap_init() */

/*****************************************************************************/
```

```
245
246 void  heap_insert(
247   int   p,
248   double   key
249 )
250 {
251   int  k;       /* hole in the heap     register*/
252   int  j;       /* parent of the hole   register*/
253   int  q;       /* heap_elt(j)        register  */
254
255   heap_key( p ) = key;
256
257   if( _heap_size == 0 )
258   {
259     _heap_size = 1;
260     heap_elt( 1 ) = p;
261     heap_idx( p ) = 1;
262     return;
263   }
264
265   k = ++ _heap_size;
266   j = k/2; /*k >> 1;*/           /* k/2 */
267
268   while( (j > 0) && (heap_key(q=heap_elt(j)) > key) ) {
269
270     heap_elt( k ) = q;
271     heap_idx( q ) = k;
272     k = j;
273     j = k/2; /*k>>1;*/    /* k/2 */
274
275   }
276
277   /* store p in the position of the hole */
278   heap_elt( k ) = p;
279   heap_idx( p ) = k;
280
281 } /* END heap_insert() */
282
283
284 /*****************************************************************************/
285
286 void  heap_decrease_key
287 (
288   int   p,
289   double   new_key
290 )
291 {
292   int    k;       /* hole in the heap  register    */
293   int    j;       /* parent of the hole   register*/
294   int    q;       /* heap_elt(j)          register*/
295
296   heap_key( p ) = new_key;
297   k = heap_idx( p );
298   j = k/2; /*k >> 1;*/           /* k/2 */
299
300   if( (j > 0) && (heap_key(q=heap_elt(j)) > new_key) ) { /* change is needed */
301     do {
302
303       heap_elt( k ) = q;
304       heap_idx( q ) = k;
305       k = j;
306       j = k/2; /*k>>1;*/    /* k/2 */
307
308     } while( (j > 0) && (heap_key(q=heap_elt(j)) > new_key) );
309
310     /* store p in the position of the hole */
311     heap_elt( k ) = p;
312     heap_idx( p ) = k;
313   }
314
315 } /* END heap_decrease_key() */
316
317
318 /*****************************************************************************/
319
320 int  heap_delete_min()
321 {
322   int    min, last;
323   int  k;          /* hole in the heap     register*/
324   int  j;          /* child of the hole    register*/
325   double  l_key;     /* key of last point    register*/
326
327   if( _heap_size == 0 )            /* heap is empty */
328     return( -1 );
329
330   min  = heap_elt( 1 );
331   last = heap_elt( _heap_size -- );
332   l_key = heap_key( last );
333
334   k = 1;  j = 2;
335   while( j <= _heap_size ) {
336
337     if( heap_key(heap_elt(j)) > heap_key(heap_elt(j+1)) )
338       j++;
339
340     if( heap_key(heap_elt(j)) >= l_key)
341       break;                       /* found a position to insert 'last' */
342
343     /* else, sift hole down */
344     heap_elt(k) = heap_elt(j);    /* Note that j <= _heap_size */
345     heap_idx( heap_elt(k) ) = k;
346     k = j;
347     j = k*2; /*k << 1;*/
348   }
349
350   heap_elt( k ) = last;
351   heap_idx( last ) = k;
352
353   heap_idx( min ) = -1;   /* mark the point visited */
354   return( min );
355
356 } /* END heap_delete_min() */
357
358
359 /*****************************************************************************/
360
361
362
363 /***************************** kNN functions *********************************/
364 /*****************************************************************************/
365 /*****************************************************************************/
366 /*****************************************************************************/
367
368 /*****************************************************************************/
```

```
369
370  void Selection(a, l,N, k,discrim,perm)
371
372  /*****************************************************************/
373  /* Makes the perm partition the array Values aint the element k.          */
374  /* Adapted from Sedgewick's Algorithms in C (p. 128)                      */
375  /*****************************************************************/
376
377  double **a;
378  int l,N,k,discrim;
379  int * perm;
380  {
381    double v;
382    int t,i,j,r;
383    r=N;
384    while(r>l) {
385      v=a[perm[r]][discrim]; i=l-1; j=r;
386      for (;;) {
387        while (a[perm[++i]][discrim] < v);
388          while (a[perm[--j]][discrim] > v && j>l);
389            if (i >= j) break;
390              t=perm[i]; perm[i] = perm[j]; perm[j]=t;
391      }
392      t=perm[i]; perm[i] = perm[r]; perm[r]=t;
393      if (i>=k) r=i-1;
394      if (i<=k) l=i+1;
395    }
396  }
397
398  /*************Need*************************************************/
399
400  int findmaxspread(l,u,dimension,points,perm)
401
402  /*****************************************************************/
403
404  int l,u,dimension;
405  double **points;
406  int * perm;
407  {
408  int i,j,maxdim;
409  double max       =-99e99,
410         min       = 99e99,
411         maxspread =-99e99;
412    for (i=0; i < dimension; i++) {
413      max =-99e99;
414      min = 99e99;
415      for (j=l; j <= u; j++) {
416        if (max < points[perm[j]][i]) {
417          max = points[perm[j]][i];
418        }
419        if (min > points[perm[j]][i]) {
420        min = points[perm[j]][i];
421        }
422        if (maxspread < fabs(max-min)) {
423          maxspread = fabs(max-min);
424          maxdim = i;
425        }
426      }
427    }
428  return(maxdim);
429  }
430
431  /********Need*************************************************************/
432
433  optkdNode *BuildkdTree(points,l,u,dimension,perm)
434
435  /*****************************************************************/
436
437  int l,u;
438  double **points;
439  int * perm;
440  {
441    optkdNode *p;
442    int m;
443    /*   printf("allocating 8\n");*/
444    NEWTREE(p);
445    if (u-l+1 <= BUCKETSIZE) {
446      p->bucket = 1;
447      p->lopt = l;
448      p->hipt = u;
449      p->loson = NULL;
450      p->hison = NULL;
451    } else {
452      p->bucket =0;
453      p->discrim = findmaxspread(l,u,dimension,points,perm);
454      m=(l+u)/2;
455      Selection(points,l,u,m,p->discrim,perm);
456      p->cutval = points[perm[m]][p->discrim];
457      p->loson = BuildkdTree(points,l,m,dimension,perm);
458      p->hison = BuildkdTree(points,m+1,u,dimension,perm);
459    }
460    return(p);
461  }
462
463  /***Need*************************************************************/
464
465  optkdNode *BuildOptTree(points,numPoints,dimension,perm)
466
467  /*****************************************************************/
468  int dimension,numPoints;
469  double **points;
470  int * perm;
471  {
472
473    int j;
474    /* initialize perm array */
475    /*  printf("allocating 1\n");*/
476
477    for (j=0; j < numPoints; j++) {
478      perm[j]=j;
479    }
480    return(BuildkdTree(points,0,numPoints-1,dimension,perm));
481  }
482
483
484  /***********Need*************************************************************/
485
486  void rnnEuclidean(p,querpoint,points,dimension,numpoints,optfound,nndist,perm)
487
488  /*****************************************************************/
489
490  /* special searching algorithm to take advantage of the fact that square roots
491     do not need to be evaulated */
492
```

```
493  optkdNode *p;
494  double *querpoint;
495  double **points;
496  int dimension,numpoints;
497  int * optfound;
498  double *nndist;
499  int * perm;
500  {
501    int i,j,k;
502    double d,thisdist,val,thisx;
503    /*printf("in here 2 alroght\n");*/
504    if (p->bucket) {
505      for (i=p->lopt; i <= p->hipt; i++) {
506        thisdist=0.0;
507        for (j=0; j<dimension; j++) {
508          d=(querpoint[j]-points[perm[i]][j]);
509          thisdist=thisdist+d*d;
510        }
511        if (optfound[0] < numpoints && thisdist > 0.0) {
512          PQInsert(thisdist,perm[i],nndist,optfound);
513        } else if (thisdist > 0.0 ) {
514          PQreplace(thisdist,nndist,optfound,perm[i]);
515        }
516      }
517    } else {
518      val = querpoint[p->discrim] - p->cutval;
519      if (val < 0) {
520        rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound,nndist,perm);
521        if (nndist[1] >= val*val) {
522          rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound,nndist,perm);
523        }
524      } else {
525        rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound,nndist,perm);
526        if (nndist[1] >= val*val) {
527          rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound,nndist,perm);
528        }
529      }
530    }
531  }
532
533
534  /****************Need**************************************************************/
535
536  int *kdOptNNQuery(points,dimension, querpoint,numNN,Metric,root,MinkP,optfound,nndist,perm)
537
538  /********************************************************************************/
539
540  optkdNode *root;
541  double *querpoint, **points;
542  int dimension,numNN,MinkP;
543  int * optfound;
544  double * nndist;
545  int * perm;
546  {
547    int j;
548    /*int *optfound;*/
549    /* set up found array */
550    /*  printf("allocating 2\n");*/
551    /*optfound = (int *) mxMalloc((numNN+1)*sizeof(int));*/
552    optfound[0]=1;  /* for now */
553    /* nndist is a priority queue of the distances of the nearest neighbors found */
554    /*  printf("allocating 3\n");*/
555    /*  nndist = (double *)mxMalloc((numNN+1)*sizeof(double));*/
556    for (j=0; j < numNN+1; j++) {
557      nndist[j] = 99e99;
558    }
559    /*printf("in here alright\n");*/
560    switch(Metric) {
561      case EUCLIDEAN :  rnnEuclidean(root,querpoint,points,dimension,numNN,optfound,nndist,perm);
562      break;
563    /*case MANHATTAN : Distance=ManhattDist;
564      rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
565      break;
566    case L_INFINITY: Distance=LInfinityDist;
567      rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
568      break;
569    case L_P       : Distance=LGeneralDist;
570      rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
571      break;*/
572    }
573    /*for (j=0;j<numNN;j++)
574    printf("%g\n",nndist[j]);
575      free(nndist);*/
576    return(optfound);
577    /*free(optfound);*/
578  }
579
580
581  /*******************************************************************************/
582  /*  Code to implement the abstract data type priority queue for use in j nearest     */
583  /*  neighbor searching.  Actual implementation is done using heaps.                  */
584  /*                                                                                   */
585  /*  Adapted from Sedgewick's: Algorithms in C p. 148-160.                            */
586  /*******************************************************************************/
587
588  /*
589     The heap data structure consists of two priority queues.  One for the j-smallest
590     distances encountered, one to keep the indexes into the points array of the
591     points corresponding to the j-smallest distances.
592  */
593
594
595
596  /*********Need*************************************************************************/
597
598  void PQupheap(DistArr,FoundArr,k)
599
600  /********************************************************************************/
601
602  double *DistArr;  /* j-smallest distances encountered */
603
604  int *FoundArr,k;
605
606  {
607    double v;
608    int j;
609
610    v=DistArr[k]; DistArr[0] = 99e99;
611    j=FoundArr[k];
612
613    while(DistArr[k/2] <= v) {
614      DistArr[k] = DistArr[k/2];
615      FoundArr[k] = FoundArr[k/2];
616      k=k/2;
```

```
  }
  DistArr[k] = v;
  FoundArr[k] = j;
}

/***********Need****************************************************************/

void PQInsert(distance,index,DistArr,FoundArr)

/*******************************************************************************/

double distance,*DistArr;
int   index, *FoundArr;

{
  FoundArr[0]=FoundArr[0]+1;
  DistArr[FoundArr[0]] = distance;
  FoundArr[FoundArr[0]] = index;
  PQupheap(DistArr,FoundArr,FoundArr[0]);
}


/************Need****************************************************************/

void PQdownheap(DistArr,FoundArr,k,index)

/*******************************************************************************/

double *DistArr;  /* j-smallest distances encountered */

int *FoundArr,k,index;

{

  int j,l,N;
  double v;

  v=DistArr[k];
  N = FoundArr[0];  /* tricky patch to maintain the data structure */
  FoundArr[0]=index;

  while (k <= N/2) {
    j=k+k;
    if (j < N && DistArr[j] <DistArr[j+1]) j++;
    if (v>=DistArr[j]) break;
    DistArr[k]=DistArr[j];
    FoundArr[k]=FoundArr[j];
    k=j;
  }

  DistArr[k] = v;
  FoundArr[k]= index;
  FoundArr[0]=N;  /* restore data struct */


}

/*************Need****************************************************************/

void PQreplace(distance,DistArr,FoundArr,index)

/*******************************************************************************/

double *DistArr,distance;
int *FoundArr;

{
  DistArr[0]=distance;
  PQdownheap(DistArr,FoundArr,0,index);
}



/*************************************************************************/

void optInRegion(P,Dimension,Points,RectQuery,optfound, nndist,perm)

/*************************************************************************/
/* Determines if the treenode P falls inside the rectangular query       */
/* RectQuery.  If so, adds the array index of the point to the found     */
/* array.                                                                */
/*************************************************************************/

optkdNode *P;
int Dimension;
double **Points, **RectQuery;
double* nndist;
int* perm;
int * optfound;
{
  int index,dc,InsideRange;


  for (index=P->lopt;index<=P->hipt;index++) {
    InsideRange=1;
/*    circle(Points[perm[index]][0],Points[perm[index]][1],pradius); */

    for (dc=0; dc < Dimension; dc ++) {
      if ((Points[perm[index]][dc] < RectQuery[dc][0] || Points[perm[index]][dc] >
           RectQuery[dc][1])) {   /* P is in the region */
         InsideRange=0;
         break;
      }
    }
    if (InsideRange) {
      Count++;
      /*if ((Count % 4000) == 0) {
        mxRealloc(optfound,(4000+Count)*sizeof(int));
      }
      if (optfound == NULL) {
        printf("we have a memory problem\n");
      }*/
      optfound[Count] = perm[index];
    }
  }
}

/*************************************************************************/

void optAddRegion(P,Dimension,Points,RectQuery,optfound, nndist,perm)

/*************************************************************************/
```

```
741 /* Adds the array index of each point in the bucket the point to the found */
742 /* array.  There is no need to check if the points are in it because we    */
743 /* have proven so already.                                                 */
744 /***************************************************************************/
745
746 optkdNode *P;
747 int Dimension;
748 double **Points, **RectQuery;
749 double* nndist;
750 int* perm;
751 int * optfound;
752 {
753   int index,dc;
754
755   for (index=P->lopt;index<=P->hipt;index++) {
756     Count++;
757     /*if ((Count % 4000) == 0) {
758       mxRealloc(optfound,(4000+Count)*sizeof(int));
759     }
760     if (optfound == NULL) {
761       printf("we have a memory problem\n");
762     }*/
763     optfound[Count] = perm[index];
764   }
765 }
766
767 /***************************************************************************/
768
769 int optBoundsIntersectRegion(B,RectQuery,Dimension)
770
771 /***************************************************************************/
772 /* Returns true iff the hyper-rectangle defined by bounds array B          */
773 /* intersects the rectangular query RectQuery.                             */
774 /***************************************************************************/
775
776 double *B,**RectQuery;
777 int Dimension;
778
779 {
780   int dc;
781
782   for (dc=0; dc < Dimension; dc++) {
783     if (B[2*dc] > RectQuery[dc][1] || B[2*dc+1] < RectQuery[dc][0]) {
784       return(0);
785     }
786   }
787   return(1);
788 }
789
790 /***************************************************************************/
791
792 int optBoundsContainsRegion(B,RectQuery,Dimension)
793
794 /***************************************************************************/
795 /* Returns true iff the hyper-rectangle defined by bounds array B          */
796 /* is completely contained inside the rectangular query RectQuery.         */
797 /***************************************************************************/
798
799 double *B,**RectQuery;
800 int Dimension;
801
802 {
803   int dc;
804
805   for (dc=0; dc < Dimension; dc++) {
806     if (!(B[2*dc]   >= RectQuery[dc][0] &&
807         B[2*dc+1] <= RectQuery[dc][1])) {
808       return(0);
809     }
810   }
811   return(1);
812 }
813
814
815 /***************************************************************************/
816
817 void optRangeSearch(P,Points,Dimension,RectQuery,B,optfound, nndist,perm)
818
819 /***************************************************************************/
820
821 optkdNode *P;
822 double **RectQuery, **Points, *B;
823 int Dimension;
824 double* nndist;
825 int* perm;
826 int * optfound;
827 {
828   int dc, disc;
829   double *BHigh,*BLow;
830
831
832   if (P==NULL) {printf("somehow a null pointer got sent here\n");}
833
834   if (P->bucket) {
835     /*printf("yes, bucket exists\n");*/
836     if (optBoundsContainsRegion(B,RectQuery,Dimension)) {
837       /*printf("yes, bounds contain region\n");*/
838       optAddRegion(P,Dimension,Points,RectQuery,optfound, nndist,perm);
839     } else {
840       optInRegion(P,Dimension,Points,RectQuery,optfound, nndist,perm);
841     }
842     return;
843   }
844
845   /* Claim: P is not a bucket node */
846   disc=P->discrim; /*printf("disc: %d\n",disc);*/
847   BLow =  (double *)(mxMalloc(2*Dimension*sizeof(double)));
848   BHigh = (double *)(mxMalloc(2*Dimension*sizeof(double)));
849
850   if (BLow == NULL || BHigh == NULL) {
851    printf("we have a memory error\n");
852   }
853
854   /* copy the region B into BLow, BHigh */
855   for (dc=0; dc < 2*Dimension; dc++) {
856     BLow[dc]  = B[dc];
857     BHigh[dc] = B[dc];
858   }
859
860   /* Improve the Bounds for the subtrees */
861   BLow[2*disc+1] = P->cutval;
862   BHigh[2*disc] = P->cutval;
863 /*printf("%g %g\n",BLow[2*disc+1],BHigh[2*disc]);*/
864   if (optBoundsIntersectRegion(BLow,RectQuery,Dimension)) {
```

```
    optRangeSearch(P->loson,Points,Dimension,RectQuery,BLow,optfound, nndist,perm);
  }
  /*mxFree(BLow);*/
  if (optBoundsIntersectRegion(BHigh,RectQuery,Dimension)) {
    optRangeSearch(P->hison,Points,Dimension,RectQuery,BHigh,optfound, nndist,perm);
  }
  /*mxFree(BHigh);*/
}


/*************************************************************************/

int *kdOptRectQuery(root,Points,Dimension,RectQuery,optfound,nndist,perm)

/*************************************************************************/

optkdNode *root;
double **RectQuery, **Points;
int Dimension;
double* nndist;
int* perm;
int * optfound;
{
double *B;
int dc;

  B =  (double *)(mxMalloc(2*Dimension*sizeof(double)));

  for (dc =0; dc < Dimension; dc++) {
    B[2*dc]   = RectQuery[dc][0];
    B[2*dc+1] = RectQuery[dc][1];
  }

  Count=0;
  /*optfound = (int *)(malloc(4000*sizeof(int)));
  if (optfound == NULL) {
    printf("We have a memory problem\n");
  }*/

  optRangeSearch(root,Points,Dimension,RectQuery,B, optfound, nndist,perm);
  /*mxFree(B);*/
  optfound[0] = Count;
  return(optfound);
}

double dist (Points, index1, index2, dimension)
double ** Points;
int index1, index2;
int dimension;
{
  int j;
  double d;
  for (j = 0; j < dimension; j++)
    d += pow(Points[index1][j] - Points[index2][j],2);
}
```

End of Prim MST program

A.3 Program to estimate Henze-Penrose affinity using MST

Begining of Henze-Penrose affinity estimation program

```
/*****************************************************************************

  File: mst.c
  Rev:  a-1
  Date: 09/27/2001

  Copyright (c) 1993, 2001 by David M. Warme

*****************************************************************************
  Program to construct MST and derive Henze-Penrose affinity
  Modified Code Copyright (c) 2001- 2005, by Huzefa Neemuchwala
  hneemuch@umich.edu
    Please read this preamble to address memory errors before contacting me.
*****************************************************************************

  Modification Log:

  a-1:  09/27/2001  warme
    : Created from pieces of GeoSteiner version 3.1.

  Modified by Huzefa Neemuchwala
  10/12/01
  Can now read point set from file directly.
  Changed the sort to a fast qsort using C library function
  01/10/02
  Now works extremely fast for huge datasets.
  Fix a radius within which points are accepted to form edges.
  So for each point roughly 10 nearest neighbors are accepted by
  fixing a suitable radius. So from N^2 edges (for N points) we
  now have N*10 edges!!
  Time reduced for sorting is enormous.
  Memory requirements fall to a fraction, and so huge point sets
  are now possible.
  Basically linearized the performance of the sort program
  by reducing the number of egdes that require sorting
  02/11/02
  Linearized the entire algorithm by linearizing time/memory for edge
  selection also.
  Contact Huzefa Neemuchwala, hneemuch@umich.edu
  for details

  Note: However you should do that in conjunction with reducing your search radius
  The line:     E = mxMalloc (npoints1*10000*sizeof(struct edge)); in function 'principal'
  allocates memory for the edge matrix.

*****************************************************************************/
#include <math.h>
#include "mex.h"
#include "string.h"
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

#define NOT !
#define AND &&
#define OR  ||
#define EQ  ==
#define NE  !=

#define NEWA(n, type) ((type *) new ((size_t) ((n) * sizeof (type))))

struct edge {
  int   p1;   /* First endpoint of edge. */
  int   p2;   /* Second endpoint of edge. */
  double    len;    /* Length of edge. */
  int startdensity;
  int enddensity;
};


/*
 * This is the so-called "Disjoint Set Union-Find" data structure.
 * The operations are "makeset", "find", and "union".
 *
 * See chapter 2 of "Data Structures and Network Algorithms", by
 * Robert Endre Tarjan, SIAM, 1983 for complete details.
 */

struct dsuf {
  int * parent;
  int * rank;
  int set_size;
};


/*
 * Local Routines
 */

static void principal (double, double, double, double *, double *, double *, double *, double *, double *, double *);
static void dsuf_create (struct dsuf *, int);
static void dsuf_destroy (struct dsuf *);
static int  dsuf_find (struct dsuf *, int);
static void dsuf_makeset (struct dsuf *, int);
static void dsuf_unite (struct dsuf *, int, int);
static void fatal (char *);
static int  mst_edge_list (int, int, struct edge *, struct edge *);
static void * new (size_t);
static void sort_edge_list (struct edge *, int);

/*
 * The main routine.  Read in points, compute MST, output it.
 */

void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{

  double npoints, dim;
  double* prfrmetric;

  double* prmstlength, *prcostaherolength;
  double threshold, threshold2;

  double *edges, *edgelabels;
  double *V;
  double *SL, *densitylabel;
  double *edge_count;
```

```
  /* Check for proper number of arguments. */
  if (nrhs != 6) {
    mexErrMsgTxt("Six inputs required : FRmstmex(rrw,indexs,densitylabels,N1+N2,dim,threshold)");
  } else if (nlhs > 4) {
    mexErrMsgTxt("Only four output arguments");
  }

  /* Assign pointers to each input */
  /* Use mex syntax */
  /* matrices for each time point */
  V = (double *)mxGetPr(prhs[0]);
  SL = (double *)mxGetPr(prhs[1]);
  densitylabel = (double *)mxGetPr(prhs[2]);

  /* parameters for analysis */
  npoints = *mxGetPr(prhs[3]);
  dim = *mxGetPr(prhs[4]);
  threshold = *mxGetPr(prhs[5]);
  threshold2 = threshold*threshold;

  /* Create matrix for the return argument. */
  /* this is basically useless */
  plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);
  plhs[1] = mxCreateDoubleMatrix(1,1,mxREAL);
  plhs[2] = mxCreateDoubleMatrix(1,1,mxREAL);
  plhs[3] = mxCreateDoubleMatrix(1,1,mxREAL);

  /* Assign pointer to the output */
  prmstlength = (double *)mxGetPr(plhs[0]);
  prfrmetric = (double *)mxGetPr(plhs[1]);
  prcostaherolength = (double *)mxGetPr(plhs[2]);
  edge_count = (double *)mxGetPr(plhs[3]);

  principal(npoints,dim,threshold2,V,SL,densitylabel,prmstlength,prfrmetric,prcostaherolength,edge_count);

}

void
principal (
double npoints, double dim, double threshold2,
double *V, double *SL, double *densitylabel,
double *prmstlength, double *prfrmetric, double *prcostaherolength, double * edge_count
)

{
int   i, j, k;
int   q,qctr,flagt,ind,nind;
int   nedges;
struct edge * E;
double *  pi;
double *  pj;
struct edge * ep;
double    dist, length, dist1, delta;
struct edge * solution;
int     ctr1=0;
int         FRmetric;
double      FRlength;
int dim1, npoints1;
  printf("Total points %g;  Dimensions %g;  Threshold %g\n",npoints,dim, threshold2);
  dim1 = (int) dim;
  npoints1 = (int) npoints;
  nedges  = npoints1 * (npoints1 - 1) / 2;


/*********************************************************
Radius approach
Select points for edge computation using radius approach.
*********************************************/

    /*E = NEWA (160000*200, struct edge);*/
    E = (struct edge *) mxMalloc( 160000*400*sizeof(struct edge)); /*(npoints1*2000) * sizeof(struct edge) );*/
  ep = E;


  for (q=0; q< npoints1-1; q++) {
      flagt=1;
      ind = (int) SL[q];
      qctr=1;
      do {
        nind = (int) SL[q+qctr];
        pi = &V [ind * dim1];
        pj = &V [nind * dim1];
        dist1=(pi[0]-pj[0])*(pi[0]-pj[0]);
        dist = 0.0;
        for (k = 0; k < dim1; k++) {
          delta = pi [k] - pj [k];
          dist += (delta * delta);
          if (dist > threshold2)
          break;
          }
        if (dist <= threshold2){
          ep -> p1  = ind;
          ep -> p2  = nind;
          ep -> len = sqrt(dist);
          ep -> startdensity = (int) densitylabel[ind];
          ep -> enddensity = (int) densitylabel[nind];
          ++ep;
          ctr1++;
          }
        if (dist1 > threshold2) {flagt=0;}
        if ((q+qctr) == (npoints-1)) {flagt=0;}
        qctr++;
        }
      while (flagt==1);
      }

/*************************************************/


  nedges=ctr1;
  /*printf("ctr1 is %d\n",ctr1);*/

  /* Allocate buffer to hold solution. */
    solution = (struct edge *)mxMalloc((npoints1-1)*sizeof(struct edge));
  /*solution = NEWA (npoints1 - 1, struct edge);*/
  k = mst_edge_list (npoints1, nedges, E, solution);
  ep = solution;
  length = 0.0;
  FRmetric=0;
  FRlength=0.0;
  for (i = 0; i < k; i++) {

    length += ep -> len;
    if (ep->startdensity != ep->enddensity)
```

```
245         {
246         FRmetric++;
247         FRlength += ep->len;
248         }
249       ep++;
250     }
251     *prmstlength = length;
252     *prfrmetric = (double)FRmetric;
253     *prcostaherolength = FRlength;
254     *edge_count = k;
255   }
256
257
258   /*
259    * This routine computes the MST of a given list of edges.
260    */
261
262     static
263     int
264   mst_edge_list (
265
266   int     n,     /* IN - number of vertices */
267   int     nedges,   /* IN - number of edges */
268   struct edge *   edge_list,  /* IN - list of edges */
269   struct edge *   edges   /* OUT - MST edge list */
270   )
271   {
272   int   i;
273   int   mst_edge_count;
274   int   components;
275   int   max_vert;
276   struct edge * ep;
277   struct edge *pp;
278   struct edge * ep_endp;
279   int   root1;
280   int   root2;
281   struct dsuf sets;
282   int     ctr =0;
283   double    perc;
284
285       sort_edge_list (edge_list, nedges);
286
287     /* Don't assume that the vertex numbers are well-behaved, */
288     /* except that they must be non-negative.  We do a quick scan */
289     /* to determine the largest vertex number and then allocate */
290     /* a union-find data structure large enough to handle it.  Note */
291     /* that we then use this union-find data structure in a   */
292     /* completely sparse way -- we only ever access set items for */
293     /* vertices that are named by an edge.        */
294
295     max_vert = 1;   /* avoid zero-size union-find... */
296     ep = edge_list;
297     for (i = 0; i < nedges; i++, ep++) {
298       if (ep -> p1 > max_vert) {
299         max_vert = ep -> p1;
300       }
301       if (ep -> p2 > max_vert) {
302         max_vert = ep -> p2;
303       }
304     }
305
306     dsuf_create (&sets, max_vert + 1);
307
308     /* Note that it is not a problem to "makeset" a vertex more */
309     /* than once...             */
310     ep = edge_list;
311     for (i = 0; i < nedges; i++, ep++) {
312       dsuf_makeset (&sets, ep -> p1);
313       dsuf_makeset (&sets, ep -> p2);
314     }
315
316     components = n;
317     mst_edge_count = 0;
318     ep = edge_list;
319     ep_endp = (ep + nedges);
320
321     while (components > 1) {
322       ctr++;
323       if (ep >= ep_endp) {
324         /* Ran out of edges before MST complete! */
325         /*printf("Ran our of edges before tree complete!");
326         perc=(mst_edge_count*100.0)/(n-1);
327         printf("\nthe number of edges is %d which is %g percent of all edges \n",mst_edge_count,perc);
328         pp=edge_list+ctr-1;
329         /*  dsuf_destroy (&sets);*/
330         return (mst_edge_count);
331       }
332       root1 = dsuf_find (&sets, ep -> p1);
333       root2 = dsuf_find (&sets, ep -> p2);
334       if (root1 NE root2) {
335         dsuf_unite (&sets, root1, root2);
336         *edges = *ep;
337         ++edges;
338         ++mst_edge_count;
339         --components;
340       }
341       ++ep;
342     }
343   pp=edge_list+ctr;
344   perc=100;
345
346   /*printf("Longest Edge was %g\t",pp->len);
347   printf("Use this distance instead of threshold!");
348     /*dsuf_destroy (&sets);*/
349
350     return (mst_edge_count);
351   }
352
353   /*
354    * This routine sorts the given edge list in INCREASING order by edge length.
355    */
356
357     static
358     void
359   sort_edge_list (
360
361   struct edge *   a,  /* IN/OUT - array of edges to be sorted. */
362   int     n /* IN - number of elements in array. */
363   )
364   {
365   int   h;
366   struct edge tmp;
367   double    key;
368   struct edge * p1;
```

```
struct edge * p2;
struct edge * p3;
struct edge * p4;
struct edge * endp;

  endp = &a [n];

  for (h = 1; h <= n; h = 3*h+1) {
  }

  do {
    h = h / 3;
    p4 = &a [h];
    p1 = p4;
    while (p1 < endp) {
      tmp = *p1;
      key = tmp.len;
      p2 = p1;
      for (;;) {
        p3 = (p2 - h);
        if (p3 -> len <= key) break;
        *p2 = *p3;
        p2 = p3;
        if (p2 < p4) break;
      }
      *p2 = tmp;
      ++p1;
    }
  } while (h > 1);
}

/*
 * This routine creates a collection of N disjoint sets.  They are left
 * uninitialized so that a sparse collection can be accessed quickly.
 */

  static
  void
dsuf_create (

struct dsuf * dsp,    /* IN/OUT - sets to create */
int   n   /* IN - number of disjoint sets */
)
{
  if (n <= 0) {
    fatal ("dsuf_create: Bug 1.");
  }

  dsp -> set_size   = n;
  /*dsp -> parent   = NEWA (n, int);*/
  dsp -> parent   =(int *)mxMalloc (n*sizeof( int));
  /*dsp -> rank   = NEWA (n, int);*/
  dsp -> rank   =(int *)mxMalloc (n*sizeof( int));
}


/*
 * Destroy the given collection of disjoint sets.
 */

  static
  void
dsuf_destroy (

struct dsuf * dsp   /* IN - sets to destroy */
)
{
  /*free ((char *) (dsp -> rank));
  free ((char *) (dsp -> parent));*/

  dsp -> set_size = 0;
  dsp -> parent = NULL;
  dsp -> rank = NULL;
}

/*
 * This routine makes a single disjoint set for item "i".
 */

  static
  void
dsuf_makeset (

struct dsuf * dsp,    /* IN - collection of sets */
int   i   /* IN - item to make into a disjoint set */
)
{
  if ((i < 0) OR (i >= dsp -> set_size)) {
    /* Item out of bounds. */
    fatal ("dsuf_makeset: Bug 1.");
  }
  dsp -> parent [i] = i;
  dsp -> rank [i]   = 0;
}

/*
 * This routine "unites" two sets that were previously disjoint.  I and J
 * must be the "canonical" member of each disjoint set (i.e. they must
 * each be the output of a "find" operation), and must be distinct.
 *
 * We perform the "union by rank" heuristic here.
 */

  static
  void
dsuf_unite (

struct dsuf * dsp,    /* IN - collection of sets */
int   i,    /* IN - first set to unite */
int   j   /* IN - second set to unite */
)
{
int   ri;
int   rj;

  if ((i < 0) OR (i >= dsp -> set_size)) {
    /* Item I is out of range. */
    fatal ("dsuf_unite: Bug 1.");
  }
  if ((j < 0) OR (j >= dsp -> set_size)) {
    /* Item J is out of range. */
    fatal ("dsuf_unite: Bug 2.");
  }
  if (i EQ j) {
```

```
493       /* Attempt to unite I with I. */
494       fatal ("dsuf_unite: Bug 3.");
495     }
496
497     ri = dsp -> rank [i];
498     rj = dsp -> rank [j];
499
500     if (ri EQ rj) {
501       /* Both subtrees have the same maximum depth.  We */
502       /* arbitrarily choose I to be underneath J.  The rank */
503       /* of J must then increase.       */
504       dsp -> parent [i] = j;
505       dsp -> rank [j]   = rj + 1;
506     }
507     else if (ri > rj) {
508       /* Tree I is (probably) deeper.  Putting J underneath */
509       /* will not increase I's rank.        */
510       dsp -> parent [j] = i;
511     }
512     else {
513       /* Tree J is (probably) deeper... */
514       dsp -> parent [i] = j;
515     }
516   }
517
518   /*
519    * This routine, given a member I of one of the disjoint sets A, will
520    * choose a cannonical member J of set A and return it.  Until set A gets
521    * united with some other set, find (I) will always return the same J.
522    *
523    * This routine performs the "path compression" heuristic.
524    */
525
526     static
527     int
528   dsuf_find (
529
530   struct dsuf * dsp,    /* IN - collection of sets */
531   int   i   /* IN - item to find cannonical item for */
532   )
533   {
534   int   j;
535   int   k;
536
537     /* Yes, I know this routine is very elegent when coded  */
538     /* recursively...  Here's the iterative version.  */
539
540     j = dsp -> parent [i];
541     if (i EQ j) {
542       /* A cannonical element has itself as parent. */
543       return (i);
544     }
545
546     /* We must search up the tree -- and compress when done... */
547     for (;;) {
548       k = dsp -> parent [j];
549       if (j EQ k) break;
550       j = k;
551     }
552
553     /* Now compress the path (make all items in chain point directly */
554     /* at the root K) -- we never have to do this search again!  */
555     while (i NE k) {
556       j = dsp -> parent [i];
557       dsp -> parent [i] = k;
558       i = j;
559     }
560
561     return (k);
562   }
563
564   /*
565    * This routine displays a fatal message and then dies!
566    */
567
568     static
569     void
570   fatal (
571
572   char *    msg   /* IN - message to display. */
573   )
574   {
575     (void) fprintf (stderr, "%s\n", msg);
576     (void) fflush (stderr);
577     abort ();
578   }
579   /*
580    * This routine performs all dynamic memory allocation for the program.
581    * We test for out of memory condition here.
582    */
583
584     static
585     void *
586   new (
587
588   size_t    size    /* IN - size of chunk in bytes. */
589   )
590   {
591   void *    p;
592
593     if (size EQ 0) {
594       /* Avoid implementation-defined bahavior of malloc! */
595       size = 1;
596     }
597
598     p = malloc (size);
599     if (p EQ NULL) {
600       (void) fprintf (stderr, "Out of memory!\n");
601       exit (1);
602     }
603
604     return (p);
605   }
606
607
608
609
610
611
612
613
614
615
616
```

```
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
```

End of Henze-Penrose affinity estimation program

A.4 Program to construct kNN graph using kd tree algorithm

Begining of kNN program

```
/*****************************************************************
kNNgraphmex.c is a mex program to build a kNN graph and give out only
the cumulative graph length.
For usage see: MX_kNNlength.m
Copyright (c) by Huzefa Neemuchwala (hneemuch@umich.edu), December 2004

*****************************************************************/

/*****************************************************************************/
/* Program to perform orthogonal range searches and nearest neighbor      */
/* querys in a more sophisticated k-d tree.  In this implementation the,  */
/* nodes on any given level of the tree do not have the same              */
/* discriminating dimension as the discrimiator is chosen based on the    */
/* dimension with   the "maxspead."                                       */
/*                                                                        */
/* References:  J.H. Friedman, J.L. Bentley, R.A. Finkel  "An Algorithm   */
/* for Finding Best Matches in Logarithmic Expected Time."                */
/* ACM Transactions on Mathematical Software, Vol 3 No. 3 Sept. 1977      */
/* pp. 209-226.                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <math.h>
#include "mex.h"
#include "optkd.h"
#include "nn.h"


/* Used to create a new tree in the k-d tree */
#define TESTTREE(PP)  ((PP) = (optkdNode *)mxMalloc(sizeof(optkdNode)))
#define NEWTREE(PP)  if (TESTTREE(PP)==NULL) \
                        {printf("memory error\n");return;}
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
static void principal (double, double, double *, double, double * , /*double *,*/ double);
void Selection(double **, int, int, int, int);
int findmaxspread(int, int, int, double **);
optkdNode *BuildkdTree(double**, int, int, int);
optkdNode *BuildOptTree(double **, int, int);
void rnnEuclidean(optkdNode *, double *, double **, int, int, int*);
int *kdOptNNQuery(double ** , int, double *, int, int, optkdNode *, int);
void KillOptTree(optkdNode *);
void PQupheap(double *, int *, int);
void PQInsert(double, int, double *, int *);
void PQdownheap(double *, int *, int, int);
void PQreplace(double, double *, int *, int);
/*double EuclidDist2(double **, int, double *, int);*/

int *perm;  /* permutation array */
/*int *optfound; /*This one will be killed by KillOptTree()*/
static int Count=0;
double *nndist;

/*double (*Distance)();
extern double fabs();*/


void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{

  double numpoints, dimension, kneighbors;
  double* prLxo;
  double *rrw;
  /*double *Graph;*/
  double gamma;


  /* Check for proper number of arguments. */
  if (nrhs != 5) {
    mexErrMsgTxt("Five inputs required : kNNgraphmex (rrw, N, dimension, k_as_in_kNN, gamma)");
  } else if (nlhs > 1) {
    mexErrMsgTxt("Only one output argument");
  }

  /* Assign pointers to each input */
  /* Use mex syntax */
  /* matrices for each time point */
  rrw = (double *)mxGetPr(prhs[0]);

  /* parameters for analysis */
  numpoints = *mxGetPr(prhs[1]);
  dimension = *mxGetPr(prhs[2]);
  kneighbors= *mxGetPr(prhs[3]);
  kneighbors = kneighbors + 1;  /* Because first neighbor is the point itself */
  gamma = *mxGetPr(prhs[4]);
  if (kneighbors > numpoints)
    mexErrMsgTxt("Input Error 1: Number of NN cant be larger than total number of points ");
  /* Create matrix for the return argument. */
  /* this is basically useless */
  plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);
  /*plhs[1] = mxCreateDoubleMatrix( numpoints*kneighbors, 1, mxREAL);*/

  /* Assign pointer to the output */
  prLxo = (double *)mxGetPr(plhs[0]);
  /*Graph = (double *)mxGetPr(plhs[1]);*/
  /*printf("%g %g %g\n",numpoints, dimension,kneighbors);*/
  principal (numpoints, dimension, rrw, kneighbors, prLxo , /*Graph,*/ gamma);
  /*free(nndist);
  free(perm);*/
}


/************Need*************************************************************/

void Selection(a, l,N, k,discrim)

/*****************************************************************************/
/* Makes the perm partition the array Values along the element k.          */
/* Adapted from Sedgewick's Algorithms in C (p. 128)                       */
/*****************************************************************************/

double **a;
int l,N,k,discrim;

{
double v;
int t,i,j,r;

r=N;
```

```
121
122 while(r>l) {
123   v=a[perm[r]][discrim]; i=l-1; j=r;
124   for (;;) {
125     while (a[perm[++i]][discrim] < v);
126     while (a[perm[--j]][discrim] > v && j>l);
127     if (i >= j) break;
128     t=perm[i]; perm[i] = perm[j]; perm[j]=t;
129   }
130   t=perm[i]; perm[i] = perm[r]; perm[r]=t;
131   if (i>=k) r=i-1;
132   if (i<=k) l=i+1;
133 }
134 }
135
136 /*************Need************************************************************/
137
138 int findmaxspread(l,u,dimension,points)
139
140 /*****************************************************************************/
141
142 int l,u,dimension;
143 double **points;
144
145 {
146 int i,j,maxdim;
147 double max       =-999999999.0,
148        min       = 999999999.0,
149        maxspread =-999999999.0;
150 for (i=0; i < dimension; i++) {
151   max =-999999999.0;
152   min = 999999999.0;
153   for (j=l; j <= u; j++) {
154     if (max < points[perm[j]][i]) {
155       max = points[perm[j]][i];
156     }
157     if (min > points[perm[j]][i]) {
158       min = points[perm[j]][i];
159     }
160     if (maxspread < fabs(max-min)) {
161       maxspread = fabs(max-min);
162       maxdim = i;
163     }
164   }
165 }
166 return(maxdim);
167 }
168
169 /********Need************************************************************************/
170
171 optkdNode *BuildkdTree(points,l,u,dimension)
172
173 /*****************************************************************************/
174
175 int l,u;
176 double **points;
177
178 {
179 optkdNode *p;
180 int m;
181
182   NEWTREE(p);
183   if (u-l+1 <= BUCKETSIZE) {
184     p->bucket = 1;
185     p->lopt = l;
186     p->hipt = u;
187     p->loson = NULL;
188     p->hison = NULL;
189   } else {
190     p->bucket =0;
191     p->discrim = findmaxspread(l,u,dimension,points);
192     m=(l+u)/2;
193     Selection(points,l,u,m,p->discrim);
194     p->cutval = points[perm[m]][p->discrim];
195     p->loson = BuildkdTree(points,l,m,dimension);
196     p->hison = BuildkdTree(points,m+1,u,dimension);
197   }
198   return(p);
199 }
200
201 /***Need**********************************************************************/
202
203 optkdNode *BuildOptTree(points,numPoints,dimension)
204
205 /*****************************************************************************/
206 int dimension,numPoints;
207 double **points;
208 {
209
210 int j;
211
212   /* initialize perm array */
213   perm = (int *) mxMalloc(numPoints*sizeof(int));
214   for (j=0; j < numPoints; j++) {
215     perm[j]=j;
216   }
217   return(BuildkdTree(points,0,numPoints-1,dimension));
218 }
219
220
221 /**********Need************************************************************/
222
223 void rnnEuclidean(p,querpoint,points,dimension,numpoints,optfound)
224
225 /*****************************************************************************/
226
227 /* special searching algorithm to take advantage of the fact that square roots
228    do not need to be evaulated */
229
230 optkdNode *p;
231 double *querpoint;
232 double **points;
233 int dimension,numpoints;
234 int * optfound;
235
236 {
237   int j,j,k;
238   double d,thisdist,val,thisx;
239
240   if (p->bucket) {
241     for (i=p->lopt; i <= p->hipt; i++) {
242       thisdist=0.0;
243       for (j=0; j<dimension; j++) {
244         d=(querpoint[j]-points[perm[i]][j]);
```

```
        thisdist=thisdist+d*d;
      }

      if (optfound[0] < numpoints) {
        PQinsert(thisdist,perm[i],nndist,optfound);
      } else {
        PQreplace(thisdist,nndist,optfound,perm[i]);
      }
    }
  } else {
    val = querpoint[p->discrim] - p->cutval;
    if (val < 0) {
      rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound);
      if (nndist[1] >= val*val) {
        rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound);
      }
    } else {
      rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound);
      if (nndist[1] >= val*val) {
        rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound);
      }
    }
  }
}


/****************Need*****************************************************/

int *kdOptNNQuery(points,dimension, querpoint,numNN,Metric,root,MinkP)

/*************************************************************************/

optkdNode *root;
double *querpoint, **points;
int dimension,numNN,MinkP;

{
  int j;
  int *optfound;
  /* set up found array */
  optfound = (int *) mxMalloc((numNN+1)*sizeof(int));
  optfound[0]=1;  /* for now */

  /* nndist is a priority queue of the distances of the nearest neighbors found */
  nndist = (double *)mxMalloc((numNN+1)*sizeof(double));
  for (j=0; j < numNN+1; j++) {
    nndist[j] = 99999999999.0;
  }

  switch(Metric) {
    case EUCLIDEAN :  rnnEuclidean(root,querpoint,points,dimension,numNN,optfound);
                      break;
    /*case MANHATTAN : Distance=ManhattDist;
                      rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
                      break;
    case L_INFINITY: Distance=LInfinityDist;
                      rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
                      break;
    case L_P       : Distance=LGeneralDist;
                      rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
                      break;*/
  }
  /*for (j=0;j<numNN;j++)
  printf("%g\n",nndist[j]);
  free(nndist);*/
  return(optfound);
  /*free(optfound);*/
}


/******Need*************************************************************/

void KillOptTree(P)

/*************************************************************************/

/*  Kills a kd-tree to avoid memory holes.   */


optkdNode *P;

{
  /*if (perm != NULL) {
    free(perm);
  }  /* free permutation array */

  if (P==NULL) {
    return;
  } /* just to be sure */
  if (P->loson != NULL) {
    KillOptTree(P->loson);
  }

  if (P->hison != NULL) {
    KillOptTree(P->hison);
  }

  free(P);

}



/*****************************************************************************/
/*  Code to implement the abstract data type priority queue for use in j nearest     */
/*  neighbor searching.  Actual implementation is done using heaps.                  */
/*                                                                                   */
/*  Adapted from Sedgewick's: Algorithms in C p. 148-160.                            */
/*****************************************************************************/

/*
   The heap data structure consists of two priority queues.  One for the j-smallest
   distances encountered, one to keep the indexes into the points array of the
   points corresponding to the j-smallest distances.
*/



/*********Need*****************************************************************/

void PQupheap(DistArr,FoundArr,k)

/*****************************************************************************/
```

```
double *DistArr;  /* j-smallest distances encountered */

int *FoundArr,k;

{
double v;
int j;

v=DistArr[k]; DistArr[0] = 999999999999999.0;
j=FoundArr[k];

while(DistArr[k/2] <= v) {
  DistArr[k] = DistArr[k/2];
  FoundArr[k] = FoundArr[k/2];
  k=k/2;
}
DistArr[k] = v;
FoundArr[k] = j;
}

/***********Need****************************************************************************/

void PQInsert(distance,index,DistArr,FoundArr)

/*******************************************************************************/

double distance,*DistArr;
int   index, *FoundArr;

{
  FoundArr[0]=FoundArr[0]+1;
  DistArr[FoundArr[0]] = distance;
  FoundArr[FoundArr[0]] = index;
  PQupheap(DistArr,FoundArr,FoundArr[0]);
}


/***********Need****************************************************************************/

void PQdownheap(DistArr,FoundArr,k,index)

/*******************************************************************************/

double *DistArr;  /* j-smallest distances encountered */

int *FoundArr,k,index;

{

int j,l,N;
double v;

v=DistArr[k];

N = FoundArr[0];  /* tricky patch to maintain the data structure */
FoundArr[0]=index;

while (k <= N/2) {
  j=k+k;
  if (j < N && DistArr[j] <DistArr[j+1]) j++;
  if (v>=DistArr[j]) break;
  DistArr[k]=DistArr[j];
  FoundArr[k]=FoundArr[j];
  k=j;
}

DistArr[k] = v;
FoundArr[k]= index;
FoundArr[0]=N;  /* restore data struct */


}

/************Need****************************************************************************/

void PQreplace(distance,DistArr,FoundArr,index)

/*******************************************************************************/

double *DistArr,distance;
int *FoundArr;

{
  DistArr[0]=distance;
  PQdownheap(DistArr,FoundArr,0,index);
}


/*************************************************************************************************/

void principal (double numpoints, double dimension, double *rrw, double kneighbors, double *prLxo /*, double *Graph*/, double gamma)

/*************************************************************************************************/
{
    double **Points;
    double * querpoint;
    optkdNode *OptkdRoot;
    int *Found, MinkP;
    int Metric;
    int i, j, k;
    Metric = EUCLIDEAN;
    MinkP = 0;
  numpoints = (int) numpoints;
  dimension = (int) dimension;
  kneighbors = (int) kneighbors;
    querpoint=(double *) mxMalloc(sizeof(double)*(dimension));

    Points = (double **)mxMalloc((numpoints)*sizeof(double *));

    for (k = 0; k < numpoints; k++) {
        Points[k] = (double *)mxMalloc((dimension)*sizeof(double));
    }
    for (k = 0, i = 0; k < numpoints; k++) {
        for (j = 0; j < dimension; j++, i++) {
            Points[k][j] = rrw[i];
        }
    }
    *prLxo = 0.0;
    /*Here we are going points first, then dimension
    for (j = 0, i = 0; j < dimension; j++) {
  for (k = 0; k < numpoints; k++, i++) {
    Points[k][j] = rrw[i];
        }
```

```
    }*/

    OptkdRoot = BuildOptTree(Points, numpoints, dimension);

    for (i = 0, k = 0; i < numpoints; i++) {

      for (j = 0; j < dimension; j++)
              querpoint[j]=Points[i][j];

            Found = kdOptNNQuery (Points, dimension, querpoint, kneighbors, Metric, OptkdRoot, MinkP);

          for (j = kneighbors; j >= 1; j--, k++) /*Found[0] is a dummy. */
      {
    *prLxo += pow(sqrt( nndist[j] ),gamma);
/*    Graph[k] = Found[j] + 1;*/
      }
    }
   /*KillOptTree(OptkdRoot);
    OptkdRoot=NULL;*/
    /*free(Found);*/
    /*for (k=0;k<numpoints;k++)
    free(Points[k]);
    free(querpoint);*/
}
```

End of kNN code

A.5 Program to construct estimates of α-MI using kNN graph

Begining of kNN-based α-MI estimation program

```
/*************************************************************************
Mex program to calculate:
alpha Mutual information for multiple images (>=2) for multiple feature dimensions (>=2)
Copyright (c) Huzefa Neemuchwala, All Rights Reserved
hneemuch@umich.edu
Known Problems:
For single pixel intensity (scalar) features: Code assumes that denominator is 1.
*************************************************************************/




/*************************************************************************/
/* Program to perform orthogonal range searches and nearest neighbor      */
/* querys in a more sophisticated k-d tree.  In this implementation the,  */
/* nodes on any given level of the tree do not have the same              */
/* discriminating dimension as the discrimiator is chosen based on the    */
/* dimension with   the "maxspead."                                       */
/*                                                                        */
/* References:  J.H. Friedman, J.L. Bentley, R.A. Finkel "An Algorithm    */
/* for Finding Best Matches in Logarithmic Expected Time."                */
/* ACM Transactions on Mathematical Software, Vol 3 No. 3 Sept. 1977      */
/* pp. 209-226.                                                           */
/*************************************************************************/

#include <stdio.h>
#include <math.h>
#include "mex.h"
#include "optkd.h"
#include "nn.h"


/* Used to create a new tree in the k-d tree */
#define TESTTREE(PP)  ((PP) = (optkdNode *)mxMalloc(sizeof(optkdNode)))
#define NEWTREE(PP)  if (TESTTREE(PP)==NULL) \
                       {printf("memory error\n");return;}
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
static void principal (double, double, double, double, double *, double *, double);
void Selection(double **, int, int, int, int,int*);
int findmaxspread(int, int, int, double **,int*);
optkdNode *BuildkdTree(double**, int, int, int,int*);
optkdNode *BuildOptTree(double **, int, int,int*);
void rnnEuclidean(optkdNode *, double *, double **, int, int, int*,double *, int *);
int *kdOptNNQuery(double ** , int, double *, int, int, optkdNode *, int, int*,double*,int*);
void KillOptTree(optkdNode *);
void PQupheap(double *, int *, int);
void PQinsert(double, int, double *, int *);
void PQdownheap(double *, int *, int, int);
void PQreplace(double, double *, int *, int);
static int Count=0;

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{

  double numpoints, dimension, kneighbors, num_images;
  double* praMI;
  double *rrw;
  double gamma;


  /* Check for proper number of arguments. */
  if (nrhs != 6)
    mexErrMsgTxt("Six inputs required : kNNgraphmex (rrw, num_images, N, dimension, kneighbors, gamma)");
  else if (nlhs > 1)
    mexErrMsgTxt("Only one output argument");

  /* Assign pointers to each input */
  /* Use mex syntax */
  /* matrices for each time point */
  rrw = (double *)mxGetPr(prhs[0]);
  num_images =  *mxGetPr(prhs[1]);
  numpoints = *mxGetPr(prhs[2]);
  dimension = *mxGetPr(prhs[3]);
  kneighbors = *mxGetPr(prhs[4]);
  /*kneighbors = kneighbors + 1;*/
  gamma = *mxGetPr(prhs[5]);
  /* Create matrix for the return argument. */
  /* this is basically useless */
  plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);

  /* Assign pointer to the output */
  praMI = (double *) mxGetPr(plhs[0]);
  principal (numpoints, num_images, dimension, kneighbors, rrw, praMI, gamma);
}


/************Need*****************************************************************/

void Selection(a, l,N, k,discrim,perm)

/*************************************************************************/
/* Makes the perm partition the array Values along the element k.         */
/* Adapted from Sedgewick's Algorithms in C (p. 128)                      */
/*************************************************************************/

double **a;
int l,N,k,discrim;
int * perm;
{
  double v;
  int t,i,j,r;
  r=N;
  while(r>l) {
    v=a[perm[r]][discrim]; i=l-1; j=r;
    for (;;) {
      while (a[perm[++i]][discrim] < v);
        while (a[perm[--j]][discrim] > v && j>l);
          if (i >= j) break;
            t=perm[i]; perm[i] = perm[j]; perm[j]=t;
    }
    t=perm[i]; perm[i] = perm[r]; perm[r]=t;
    if (i>=k) r=i-1;
    if (i<=k) l=i+1;
  }
}

/************Need*****************************************************************/

int findmaxspread(l,u,dimension,points,perm)

```

```
121  /*****************************************************************************/
122
123  int l,u,dimension;
124  double **points;
125  int * perm;
126  {
127  int i,j,maxdim;
128  double max       =-99e99,
129         min       = 99e99,
130         maxspread =-99e99;
131    for (i=0; i < dimension; i++) {
132      max =-99e99;
133      min = 99e99;
134      for (j=l; j <= u; j++) {
135        if (max < points[perm[j]][i]) {
136          max = points[perm[j]][i];
137        }
138        if (min > points[perm[j]][i]) {
139        min = points[perm[j]][i];
140        }
141        if (maxspread < fabs(max-min)) {
142          maxspread = fabs(max-min);
143          maxdim = i;
144        }
145      }
146    }
147  return(maxdim);
148  }
149
150  /********Need*****************************************************************/
151
152  optkdNode *BuildkdTree(points,l,u,dimension,perm)
153
154  /*****************************************************************************/
155
156  int l,u;
157  double **points;
158  int * perm;
159  {
160    optkdNode *p;
161    int m;
162    /*   printf("allocating 8\n");*/
163    NEWTREE(p);
164    if (u-l+1 <= BUCKETSIZE) {
165      p->bucket = 1;
166      p->lopt = l;
167      p->hipt = u;
168      p->loson = NULL;
169      p->hison = NULL;
170    } else {
171      p->bucket =0;
172      p->discrim = findmaxspread(l,u,dimension,points,perm);
173      m=(l+u)/2;
174      Selection(points,l,u,m,p->discrim,perm);
175      p->cutval = points[perm[m]][p->discrim];
176      p->loson = BuildkdTree(points,l,m,dimension,perm);
177      p->hison = BuildkdTree(points,m+1,u,dimension,perm);
178    }
179    return(p);
180  }
181
182  /***Need*******************************************************************/
183
184  optkdNode *BuildOptTree(points,numPoints,dimension,perm)
185
186  /*****************************************************************************/
187  int dimension,numPoints;
188  double **points;
189  int * perm;
190  {
191
192    int j;
193    /* initialize perm array */
194    /*  printf("allocating 1\n");*/
195
196    for (j=0; j < numPoints; j++) {
197      perm[j]=j;
198    }
199    return(BuildkdTree(points,0,numPoints-1,dimension,perm));
200  }
201
202
203  /**********Need*************************************************************/
204
205  void rnnEuclidean(p,querpoint,points,dimension,numpoints,optfound,nndist,perm)
206
207  /*****************************************************************************/
208
209  /* special searching algorithm to take advantage of the fact that square roots
210     do not need to be evaulated */
211
212  optkdNode *p;
213  double *querpoint;
214  double **points;
215  int dimension,numpoints;
216  int * optfound;
217  double *nndist;
218  int * perm;
219  {
220    int i,j,k;
221    double d,thisdist,val,thisx;
222    /*printf("in here 2 alroght\n");*/
223    if (p->bucket) {
224      for (i=p->lopt; i <= p->hipt; i++) {
225        thisdist=0.0;
226        for (j=0; j<dimension; j++) {
227          d=(querpoint[j]-points[perm[i]][j]);
228          thisdist=thisdist+d*d;
229        }
230        if (optfound[0] < numpoints && thisdist > 0.0) {
231          PQInsert(thisdist,perm[i],nndist,optfound);
232        } else if (thisdist > 0.0 ) {
233          PQreplace(thisdist,nndist,optfound,perm[i]);
234        }
235      }
236    } else {
237      val = querpoint[p->discrim] - p->cutval;
238      if (val < 0) {
239        rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound,nndist,perm);
240        if (nndist[1] >= val*val) {
241          rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound,nndist,perm);
242        }
243      } else {
244        rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound,nndist,perm);
```

```
      if (nndist[1] >= val*val) {
        rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound,nndist,perm);
      }
    }
  }
}


/****************Need*****************************************************************/

int *kdOptNNQuery(points,dimension, querpoint,numNN,Metric,root,MinkP,optfound,nndist,perm)

/*************************************************************************/

optkdNode *root;
double *querpoint, **points;
int dimension,numNN,MinkP;
int * optfound;
double * nndist;
int * perm;
{
  int j;
  /*int *optfound;*/
  /* set up found array */
  /*  printf("allocating 2\n");*/
  /*optfound = (int *) mxMalloc((numNN+1)*sizeof(int));*/
  optfound[0]=1;  /* for now */
  /* nndist is a priority queue of the distances of the nearest neighbors found */
  /*  printf("allocating 3\n");*/
  /*  nndist = (double *)mxMalloc((numNN+1)*sizeof(double));*/
  for (j=0; j < numNN+1; j++) {
    nndist[j] = 99e99;
  }
  /*printf("in here alright\n");*/
  switch(Metric) {
    case EUCLIDEAN :  rnnEuclidean(root,querpoint,points,dimension,numNN,optfound,nndist,perm);
      break;
  /*case MANHATTAN : Distance=ManhattDist;
    rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
    break;
  case L_INFINITY: Distance=LInfinityDist;
    rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
    break;
  case L_P       : Distance=LGeneralDist;
    rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
    break;*/
  }
  /*for (j=0;j<numNN;j++)
  printf("%g\n",nndist[j]);
    free(nndist);*/
  return(optfound);
  /*free(optfound);*/
}


/******Need*************************************************************************/

void KillOptTree(P)

/*************************************************************************/

/*  Kills a kd-tree to avoid memory holes.   */


optkdNode *P;

{
  /*if (perm != NULL) {
    free(perm);
  }  /* free permutation array */

  if (P==NULL) {
    return;
  } /* just to be sure */
  if (P->loson != NULL) {
    KillOptTree(P->loson);
  }

  if (P->hison != NULL) {
    KillOptTree(P->hison);
  }

  free(P);

}


/*****************************************************************************/
/*  Code to implement the abstract data type priority queue for use in j nearest     */
/*  neighbor searching.  Actual implementation is done using heaps.                  */
/*                                                                                   */
/*  Adapted from Sedgewick's: Algorithms in C p. 148-160.                            */
/*****************************************************************************/

/*
   The heap data structure consists of two priority queues.  One for the j-smallest
   distances encountered, one to keep the indexes into the points array of the
   points corresponding to the j-smallest distances.
*/


/*********Need*****************************************************************************/

void PQupheap(DistArr,FoundArr,k)

/*****************************************************************************/

double *DistArr;  /* j-smallest distances encountered */

int *FoundArr,k;

{
  double v;
  int j;

  v=DistArr[k]; DistArr[0] = 99e99;
  j=FoundArr[k];

  while(DistArr[k/2] <= v) {
    DistArr[k] = DistArr[k/2];
    FoundArr[k] = FoundArr[k/2];
    k=k/2;
```

```
  }
  DistArr[k] = v;
  FoundArr[k] = j;
}

/***********Need****************************************************************/

void PQInsert(distance,index,DistArr,FoundArr)

/*******************************************************************************/

double distance,*DistArr;
int   index, *FoundArr;

{
  FoundArr[0]=FoundArr[0]+1;
  DistArr[FoundArr[0]] = distance;
  FoundArr[FoundArr[0]] = index;
  PQupheap(DistArr,FoundArr,FoundArr[0]);
}


/************Need****************************************************************/

void PQdownheap(DistArr,FoundArr,k,index)

/*******************************************************************************/

double *DistArr;  /* j-smallest distances encountered */

int *FoundArr,k,index;

{

  int j,l,N;
  double v;

  v=DistArr[k];
  N = FoundArr[0];  /* tricky patch to maintain the data structure */
  FoundArr[0]=index;

  while (k <= N/2) {
    j=k+k;
    if (j < N && DistArr[j] <DistArr[j+1]) j++;
    if (v>=DistArr[j]) break;
    DistArr[k]=DistArr[j];
    FoundArr[k]=FoundArr[j];
    k=j;
  }

  DistArr[k] = v;
  FoundArr[k]= index;
  FoundArr[0]=N;  /* restore data struct */


}

/*************Need****************************************************************/

void PQreplace(distance,DistArr,FoundArr,index)

/*******************************************************************************/

double *DistArr,distance;
int *FoundArr;

{
  DistArr[0]=distance;
  PQdownheap(DistArr,FoundArr,0,index);
}


/*****************************************************************************************************/

void principal (double numpoints, double num_images, double dimension, double kneighbors, double *rrw, double *praMI, double gamma)

/*****************************************************************************************************/
{
  double **Points, **Points1, *distx;
  double *querpoint, *querpoint1;
  optkdNode *OptkdRoot, *OptkdRoot1, *OptkdRoot2;
  int *Found, MinkP;
  int Metric;
  int i, j, k, j1, n, ii;
  double *nndist, distx1;
        int *optfound,*optfound1;
        int *perm, *perm1, *perm2;
        Metric = EUCLIDEAN;
  MinkP = 0;
  numpoints = (int) numpoints;
  dimension = (int) dimension;
  kneighbors = (int) kneighbors;
  /*printf("allocating 4\n");*/
  /*if (dimension/num_images != 1)*/
    querpoint=(double *) mxMalloc(sizeof(double)*(dimension/num_images));
  querpoint1=(double *) mxMalloc(sizeof(double)*(dimension));
  /*printf("allocating 5\n");*/
  Points = (double **)mxMalloc((numpoints)*sizeof(double *));
  /*hist2 = (double **)mxMalloc((255)*sizeof(double *));
  linear_hist2 = (double *)mxMalloc((numpoints)*sizeof(double));*/
  /*printf("allocating 6\n");*/

  for (k = 0; k < numpoints; k++) {
    Points[k] = (double *)mxMalloc((dimension)*sizeof(double));
  }
  for (k = 0, i = 0; k < numpoints; k++) {
    for (j = 0; j < dimension; j++, i++) {
      Points[k][j] = rrw[i];
      /*printf("Points: %g\n",Points[k][j]);*/
    }
  }


  /* printf("allocating 7\n");*/
  /*if (dimension/num_images != 1) {*/
    Points1 = (double **)mxMalloc((numpoints)*sizeof(double *));
    for (k = 0; k < numpoints; k++) {
      Points1[k] = (double *)mxMalloc((dimension/num_images)*sizeof(double));
    }
  /*}*/

  /*printf("%d\n",i);*/
  /*printf("%g numpoints, %g dimension",numpoints,dimension);*/
```

```
493     *praMI = 0.0;
494
495
496     optfound1 = (int *) mxMalloc((kneighbors + 2)*sizeof(int));
497     nndist = (double *)mxMalloc((kneighbors + 2)*sizeof(double));
498     perm  = (int *) mxMalloc(numpoints*sizeof(int));
499     OptkdRoot  = BuildOptTree(Points,  numpoints, dimension,   perm);
500     /*if (dimension/num_images != 1) {*/
501       perm1 = (int *) mxMalloc(numpoints*sizeof(int));
502       optfound  = (int *) mxMalloc((kneighbors + 2)*sizeof(int));
503   /*  }*/
504
505     distx = (double *) mxMalloc(numpoints * sizeof(double));
506     for (i=0; i<numpoints; i++) {
507       distx[i] = 0.0;
508         for (j = 0; j < dimension; j++)
509         querpoint1[j]=Points[i][j];
510         Found = kdOptNNQuery (Points, dimension, querpoint1, kneighbors, Metric, OptkdRoot, MinkP, optfound1, nndist,perm);
511         for (j = kneighbors; j >= 1; j--) {
512         if (Found[j]!=i) distx[i] += sqrt(nndist[j]);
513       }
514       /*printf("nndist: %g\n",nndist[1]);*/
515       distx[i] = distx[i]/kneighbors;
516       /*printf("distx i: %g %d\n",distx[i],i);*/
517     }
518
519   /*  if (dimension/num_images != 1) {*/
520       for (n = 0; n  < num_images; n++) {
521         for (k = 0; k < numpoints; k++)
522           for (i = 0, j = n*(dimension/num_images); j < (n+1)*(dimension/num_images); j++, i++)
523             Points1[k][i] = Points[k][j];
524         OptkdRoot1 = BuildOptTree(Points1, numpoints, dimension/num_images, perm1);
525         for (i=0; i<numpoints; i++) {
526           distx1 = 0.0;
527           for (ii = 0, j = n*(dimension/num_images); j < (n+1)*(dimension/num_images); j++, ii++)
528             querpoint[ii]=Points[i][j];
529           Found = kdOptNNQuery(Points1,dimension/num_images,querpoint,kneighbors,Metric,OptkdRoot1,MinkP,optfound,nndist,perm1);
530           for (j = kneighbors; j>=1; j--) {
531             if (Found[j]!=i) distx1 += sqrt(nndist[j]);
532           }
533           /*printf("nndist in loop: %g\n",nndist[1]);*/
534           distx1 = distx1/kneighbors; /*printf("distx1 in loop, i: %g %d\n",distx1,i);*/
535           /*printf("distx %g  distx1 %g  distx2 %g linear_hist2 %g\n",distx,distx1,distx2,linear_hist2[i]);*/
536           /*printf("distx b4 divn, i: %g %d\n",distx[i],i); */
537           distx[i] = distx[i]/(pow(distx1,(1/num_images))); /*printf("distx after divn, i: %g %d\n",distx[i],i); */
538           /*else { *praMI = linear_hist2[i]*pow(distx[i], (num_images)*gamma);}*/
539           /*printf("prami %g nowadded %g",*praMI,pow(distx/(pow(distx1*distx2,0.5)), 2*gamma));}*/
540         }
541       }
542     /*}*/
543
544     for (i = 0; i < numpoints; i++) {
545       *praMI += pow(distx[i], num_images*gamma);
546         /*printf("final distx: %g\n",distx[i]);*/
547       }
548     /**praMI = log(*praMI);*/
549     /*printf("aMI is %g\n",*praMI);*/
550     /*printf("distx1 = %g, distx2 = %g, distx = %g\n",distx1,distx2,distx);*/
551   }
```

End of kNN-based α-MI estimation program

A.6 Program to construct estimates of α-GA mean divergence using kNN graph

Begining of kNN-based α-GA mean divergence estimation program

```
/*************************************************************************
Mex program to calculate:
Arithmetic-Geometric mean divergence for multidensity (>2) case
Copyright (c) Huzefa Neemuchwala, All Rights Reserved
hneemuch@umich.edu

Known Problems: For bivariate (2 image) case, please use kNNlengthGAmex_eff.c
*************************************************************************/



/*************************************************************************/
/* Program to perform orthogonal range searches and nearest neighbor     */
/* querys in a more sophisticated k-d tree.  In this implementation the, */
/* nodes on any given level of the tree do not have the same             */
/* discriminating dimension as the discrimiator is chosen based on the   */
/* dimension with   the "maxspead."                                      */
/*                                                                       */
/* References:  J.H. Friedman, J.L. Bentley, R.A. Finkel  "An Algorithm  */
/* for Finding Best Matches in Logarithmic Expected Time."               */
/* ACM Transactions on Mathematical Software, Vol 3 No. 3 Sept. 1977     */
/* pp. 209-226.                                                          */
/*************************************************************************/

#include <stdio.h>
#include <math.h>
#include "mex.h"
#include "optkd.h"
#include "nn.h"


/* Used to create a new tree in the k-d tree */
#define TESTTREE(PP)  ((PP) = (optkdNode *)mxMalloc(sizeof(optkdNode)))
#define NEWTREE(PP)  if (TESTTREE(PP)==NULL) \
                        {printf("memory error\n");return;}
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
static void principal (double, double, double, double, double *, double, double *,  double);
void Selection(double **, int, int, int, int,int*);
int findmaxspread(int, int, int, double **,int*);
optkdNode *BuildkdTree(double**, int, int, int,int*);
optkdNode *BuildOptTree(double **, int, int,int*);
void rnnEuclidean(optkdNode *, double *, double **, int, int, int*,double *, int *);
int *kdOptNNQuery(double **, int, double *, int, int, optkdNode *, int, int*,double*,int*);
void KillOptTree(optkdNode *);
void PQupheap(double *, int *, int);
void PQInsert(double, int, double *, int *);
void PQdownheap(double *, int *, int, int);
void PQreplace(double, double *, int *, int);
/*double EuclidDist2(double **, int, double *, int);*/

/*int *perm;*/  /* permutation array */
/*int *optfound; /*This one will be killed by KillOptTree()*/
static int Count=0;
/*double *nndist;*/
/*double (*Distance)();
extern double fabs();*/


void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{

  double numpoints, dimension, kneighbors, num_images;
  double* prLxo;
/*  double *prkNNL;
  double *prLSC;*/
  double *rrw;
  /*double *Graph;*/
  double gamma;


  /* Check for proper number of arguments. */
  if (nrhs != 6) {
    mexErrMsgTxt("Five inputs required : kNNgraphmex (rrw, num_images, N, dimension, k_as_in_kNN, gamma)");
  } else if (nlhs > 1) {
    mexErrMsgTxt("Only one output argument");
  }

  /* Assign pointers to each input */
  /* Use mex syntax */
  /* matrices for each time point */
  rrw = (double *)mxGetPr(prhs[0]);
  num_images = *mxGetPr(prhs[1]);
  /* parameters for analysis */
  numpoints = *mxGetPr(prhs[2]);
  dimension = *mxGetPr(prhs[3]);
  kneighbors= *mxGetPr(prhs[4]);
  /*kneighbors = kneighbors + 1;*/  /* Because first neighbor is the point itself */
  gamma = *mxGetPr(prhs[5]);
  if (kneighbors > numpoints)
    mexErrMsgTxt("Input Error 1: Number of NN cant be larger than total number of points ");
  /* Create matrix for the return argument. */
  /* this is basically useless */
  plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);
  /*plhs[1] = mxCreateDoubleMatrix(1,1,mxREAL);
  plhs[2] = mxCreateDoubleMatrix(1,1,mxREAL);*/
  /*plhs[1] = mxCreateDoubleMatrix( numpoints*kneighbors, 1, mxREAL);*/

  /* Assign pointer to the output */
  /*prkNNL = (double *) mxGetPr(plhs[0]);*/
  prLxo = (double *)mxGetPr(plhs[0]);
  /*  prLSC = (double *)mxGetPr(plhs[2]); */
  /*Graph = (double *)mxGetPr(plhs[1]);*/
  /*printf("%g %g %g\n",numpoints, dimension,kneighbors);*/
  principal (numpoints, num_images, dimension, rrw, kneighbors, prLxo, gamma);
  /*free(nndist);
  free(perm);*/
}


/************Need*****************************************************************/

void Selection(a, l,N, k,discrim,perm)

/*************************************************************************/
```

```
/* Makes the perm partition the array Values along the element k.          */
/* Adapted from Sedgewick's Algorithms in C (p. 128)                       */
/*-------------------------------------------------------------------------*/

double **a;
int l,N,k,discrim;
int * perm;
{
double v;
int t,i,j,r;

r=N;

while(r>l) {
  v=a[perm[r]][discrim]; i=l-1; j=r;
  for (;;) {
    while (a[perm[++i]][discrim] < v);
    while (a[perm[--j]][discrim] > v && j>l);
    if (i >= j) break;
    t=perm[i]; perm[i] = perm[j]; perm[j]=t;
  }
  t=perm[i]; perm[i] = perm[r]; perm[r]=t;
  if (i>=k) r=i-1;
  if (i<=k) l=i+1;
}
}

/*************Need*************************************************************/

int findmaxspread(l,u,dimension,points,perm)

/*-------------------------------------------------------------------------*/

int l,u,dimension;
double **points;
int * perm;
{
int i,j,maxdim;
double max       =-99e99,
       min       = 99e99,
       maxspread =-99e99;
for (i=0; i < dimension; i++) {
  max =-99e99;
  min = 99e99;
  for (j=l; j <= u; j++) {
    if (max < points[perm[j]][i]) {
      max = points[perm[j]][i];
    }
    if (min > points[perm[j]][i]) {
      min = points[perm[j]][i];
    }
    if (maxspread < fabs(max-min)) {
      maxspread = fabs(max-min);
      maxdim = i;
    }
  }
}
return(maxdim);
}

/********Need*************************************************************************/

optkdNode *BuildkdTree(points,l,u,dimension,perm)

/*-------------------------------------------------------------------------*/

int l,u;
double **points;
int * perm;
{
optkdNode *p;
int m;
/*   printf("allocating 8\n");*/
  NEWTREE(p);
  if (u-l+1 <= BUCKETSIZE) {
    p->bucket = 1;
    p->lopt = l;
    p->hipt = u;
    p->loson = NULL;
    p->hison = NULL;
  } else {
    p->bucket =0;
    p->discrim = findmaxspread(l,u,dimension,points,perm);
    m=(l+u)/2;
    Selection(points,l,u,m,p->discrim,perm);
    p->cutval = points[perm[m]][p->discrim];
    p->loson = BuildkdTree(points,l,m,dimension,perm);
    p->hison = BuildkdTree(points,m+1,u,dimension,perm);
  }
  return(p);
}

/***Need*************************************************************************/

optkdNode *BuildOptTree(points,numPoints,dimension,perm)

/*-------------------------------------------------------------------------*/
int dimension,numPoints;
double **points;
int * perm;
{

int j;
  /* initialize perm array */
/*  printf("allocating 1\n");*/

  for (j=0; j < numPoints; j++) {
    perm[j]=j;
  }
  return(BuildkdTree(points,0,numPoints-1,dimension,perm));
}


/***********Need*************************************************************/

void rnnEuclidean(p,querpoint,points,dimension,numpoints,optfound,nndist,perm)

/*-------------------------------------------------------------------------*/

/* special searching algorithm to take advantage of the fact that square roots
   do not need to be evaulated */

optkdNode *p;
double *querpoint;
```

```
241 double **points;
242 int dimension,numpoints;
243 int * optfound;
244 double *nndist;
245 int * perm;
246 {
247   int i,j,k;
248   double d,thisdist,val,thisx;
249 /*printf("in here 2 alroght\n");*/
250   if (p->bucket) {
251     for (i=p->lopt; i <= p->hipt; i++) {
252       thisdist=0.0;
253       for (j=0; j<dimension; j++) {
254         d=(querpoint[j]-points[perm[i]][j]);
255         thisdist=thisdist+d*d;
256       }
257
258       if (optfound[0] < numpoints && thisdist > 0.0) {
259         PQInsert(thisdist,perm[i],nndist,optfound);
260       } else if (thisdist > 0.0){
261         PQreplace(thisdist,nndist,optfound,perm[i]);
262       }
263     }
264   } else {
265     val = querpoint[p->discrim] - p->cutval;
266     if (val < 0) {
267       rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound,nndist,perm);
268       if (nndist[1] >= val*val) {
269         rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound,nndist,perm);
270       }
271     } else {
272       rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound,nndist,perm);
273       if (nndist[1] >= val*val) {
274         rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound,nndist,perm);
275       }
276     }
277   }
278 }
279
280
281 /****************Need*****************************************************************/
282
283 int *kdOptNNQuery(points,dimension, querpoint,numNN,Metric,root,MinkP,optfound,nndist,perm)
284
285 /*************************************************************************************/
286
287 optkdNode *root;
288 double *querpoint, **points;
289 int dimension,numNN,MinkP;
290 int * optfound;
291 double * nndist;
292 int * perm;
293 {
294   int j;
295   /*int *optfound;*/
296   /* set up found array */
297 /*  printf("allocating 2\n");*/
298   /*optfound = (int *) mxMalloc((numNN+1)*sizeof(int));*/
299   optfound[0]=1;  /* for now */
300
301   /* nndist is a priority queue of the distances of the nearest neighbors found */
302   /*  printf("allocating 3\n");*/
303 /*  nndist = (double *)mxMalloc((numNN+1)*sizeof(double));*/
304   for (j=0; j < numNN+1; j++) {
305     nndist[j] = 99999999999.0;
306   }
307 /*printf("in here alright\n");*/
308   switch(Metric) {
309     case EUCLIDEAN :  rnnEuclidean(root,querpoint,points,dimension,numNN,optfound,nndist,perm);
310                       break;
311     /*case MANHATTAN : Distance=ManhattDist;
312                       rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
313                       break;
314     case L_INFINITY: Distance=LInfinityDist;
315                       rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
316                       break;
317     case L_P       : Distance=LGeneralDist;
318                       rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
319                       break;*/
320   }
321   /*for (j=0;j<numNN;j++)
322   printf("%g\n",nndist[j]);
323   free(nndist);*/
324   return(optfound);
325   /*free(optfound);*/
326 }
327
328
329 /******Need*************************************************************************/
330
331 void KillOptTree(P)
332
333 /*************************************************************************/
334
335 /*  Kills a kd-tree to avoid memory holes.   */
336
337
338 optkdNode *P;
339
340 {
341   /*if (perm != NULL) {
342     free(perm);
343   }  /* free permutation array */
344
345   if (P==NULL) {
346     return;
347   } /* just to be sure */
348   if (P->loson != NULL) {
349     KillOptTree(P->loson);
350   }
351
352   if (P->hison != NULL) {
353     KillOptTree(P->hison);
354   }
355
356   free(P);
357
358 }
359
360
361
362 /*************************************************************************************/
363 /*  Code to implement the abstract data type priority queue for use in j nearest     */
364 /*  neighbor searching.  Actual implementation is done using heaps.                  */
```

```
/*                                                                              */
/*  Adapted from Sedgewick's: Algorithms in C p. 148-160.                       */
/********************************************************************************/

/*
   The heap data structure consists of two priority queues.  One for the j-smallest
   distances encountered, one to keep the indexes into the points array of the
   points corresponding to the j-smallest distances.
*/



/*********Need*****************************************************************/

void PQupheap(DistArr,FoundArr,k)

/******************************************************************************/

double *DistArr;  /* j-smallest distances encountered */

int *FoundArr,k;

{
double v;
int j;

v=DistArr[k]; DistArr[0] = 99e99;
j=FoundArr[k];

while(DistArr[k/2] <= v) {
  DistArr[k] = DistArr[k/2];
  FoundArr[k] = FoundArr[k/2];
  k=k/2;
}
DistArr[k] = v;
FoundArr[k] = j;
}

/***********Need*****************************************************************/

void PQInsert(distance,index,DistArr,FoundArr)

/******************************************************************************/

double distance,*DistArr;
int   index, *FoundArr;

{
  FoundArr[0]=FoundArr[0]+1;
  DistArr[FoundArr[0]] = distance;
  FoundArr[FoundArr[0]] = index;
  PQupheap(DistArr,FoundArr,FoundArr[0]);
}



/************Need*****************************************************************/

void PQdownheap(DistArr,FoundArr,k,index)

/******************************************************************************/

double *DistArr;  /* j-smallest distances encountered */

int *FoundArr,k,index;

{

int j,l,N;
double v;

v=DistArr[k];

N = FoundArr[0];  /* tricky patch to maintain the data structure */
FoundArr[0]=index;

while (k <= N/2) {
  j=k+k;
  if (j < N && DistArr[j] <DistArr[j+1]) j++;
  if (v>=DistArr[j]) break;
  DistArr[k]=DistArr[j];
  FoundArr[k]=FoundArr[j];
  k=j;
}

DistArr[k] = v;
FoundArr[k]= index;
FoundArr[0]=N;  /* restore data struct */


}

/*************Need*****************************************************************/

void PQreplace(distance,DistArr,FoundArr,index)

/******************************************************************************/

double *DistArr,distance;
int *FoundArr;

{
  DistArr[0]=distance;
  PQdownheap(DistArr,FoundArr,0,index);
}


/*****************************************************************************************************/

void principal (double numpoints, double num_images, double dimension, double *rrw, double kneighbors, double *prLxo, double gamma)

/*****************************************************************************************************/
{
  double **Points, **Points1, **distx, * querpoint, * nndist;
  optkdNode *OptkdRoot1;
  int *Found, MinkP, *optfound, *perm;
  int i, ii, j, k, j1, n, m;
  double r1, r2, currmin;
  int midpoint, NumNN, Metric;

  Metric = EUCLIDEAN;
  MinkP = 0;
  numpoints = (int) numpoints;
  dimension = (int) dimension;
```

```
  kneighbors = (int) kneighbors;
  querpoint=(double *) mxMalloc(sizeof(double)*(dimension));
  Points = (double **)mxMalloc((numpoints)*sizeof(double *));
  distx = (double **) mxMalloc(num_images * sizeof(double *));
  for (k = 0; k < numpoints; k++) {
    Points[k] = (double *)mxMalloc((dimension)*sizeof(double));
  }

  for (k = 0, i = 0; k < numpoints; k++)
    for (j = 0; j < dimension; j++, i++)
      Points[k][j] = rrw[i];

  Points1 = (double **)mxMalloc((numpoints/num_images)*sizeof(double *));
  for (k = 0; k < numpoints/num_images; k++)
    Points1[k] = (double *)mxMalloc((dimension)*sizeof(double));

  *prLxo = 0.0;

  optfound = (int *) mxMalloc((3)*sizeof(int));
  nndist = (double *)mxMalloc((kneighbors+2)*sizeof(double));
  perm = (int *) mxMalloc(numpoints/num_images*sizeof(int));

  for (i = 0; i < num_images; i++)
    distx[i] = (double *) mxMalloc(numpoints * sizeof(double));


  for (n = 0; n  < num_images; n++) {
    for (j = n*(numpoints/num_images), k = 0; j < (n+1)*(numpoints/num_images); j++, k++)
      for (i = 0; i < dimension; i++)
        Points1[k][i] = Points[j][i];
    OptkdRoot1 = BuildOptTree(Points1, numpoints/num_images, dimension,perm);
    for (i=0; i < numpoints; i++) {
      distx[n][i] = 0.0;
      for (ii = 0; ii < dimension; ii++)
        querpoint[ii]=Points[i][ii];
      Found = kdOptNNQuery(Points1,dimension,querpoint,kneighbors,Metric,OptkdRoot1,MinkP,optfound,nndist,perm);
      for (j = kneighbors; j>=1; j--) {
        if (Found[j]!=i) distx[n][i] += sqrt(nndist[j]);
      }
    }
    distx[n][i] = pow(distx[n][i]/(kneighbors),(1/num_images)*gamma);
    /*printf("distx %g  distx1 %g  distx2 %g linear_hist2 %g\n",distx,distx1,distx2,linear_hist2[i]);*/
    /*distx[i] = (distx[i]/(pow(distx1,(1/num_images))));*/
    /*printf("prami %g nowadded %g",*praMI,pow(distx/(pow(distx1*distx2,0.5)), 2*gamma));}*/
  }


  for (i = 0; i < numpoints; i++) {
    currmin = 99e+99;
    for (n = 0; n <= num_images - 2; n++) {
      for (m = n + 1; m <= num_images - 1; m++) {
        r1 = distx[n][i]/distx[m][i];
        r2 = 1/r1;
        if (distx[n][i] != 0 && distx[m][i] != 0) {
          currmin = min(currmin, min(r1,r2));
        }
      }
    }
    *prLxo += currmin;
  }
  /*printf("GA is %g\n",*prLxo);*/
}
```

End of kNN-based α-GA divergence estimation program

A.7 Program to construct estimates of Non-Linear Correlation Coefficient using kNN graph

Begining of kNN-based NLCC estimation program

```
/*************************************************************************
Mex program to calculate:
kNN graph length
Arithmetic-Geometric mean divergence
kNN graph length, Single Counting edges
Copyright (c) Huzefa Neemuchwala, All Rights Reserved
hneemuch@umich.edu
*************************************************************************/



/*************************************************************************/
/* Program to perform orthogonal range searches and nearest neighbor     */
/* querys in a more sophisticated k-d tree.  In this implementation the, */
/* nodes on any given level of the tree do not have the same             */
/* discriminating dimension as the discrimiator is chosen based on the   */
/* dimension with   the "maxspead."                                      */
/*                                                                       */
/* References:  J.H. Friedman, J.L. Bentley, R.A. Finkel  "An Algorithm  */
/* for Finding Best Matches in Logarithmic Expected Time."               */
/* ACM Transactions on Mathematical Software, Vol 3 No. 3 Sept. 1977     */
/* pp. 209-226.                                                          */
/*************************************************************************/

#include <stdio.h>
#include <math.h>
#include "mex.h"
#include "optkd.h"
#include "nn.h"


/* Used to create a new tree in the k-d tree */
#define TESTTREE(PP)  ((PP) = (optkdNode *)mxMalloc(sizeof(optkdNode)))
#define NEWTREE(PP)  if (TESTTREE(PP)==NULL) \
                       {printf("memory error\n");return;}
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
static void principal (double, double, double, double, double *, double *, double);
void Selection(double **, int, int, int, int,int*);
int findmaxspread(int, int, int, double **,int*);
optkdNode *BuildkdTree(double**, int, int, int,int*);
optkdNode *BuildOptTree(double **, int, int,int*);
void rnnEuclidean(optkdNode *, double *, double **, int, int, int*,double *, int *);
int *kdOptNNQuery(double **, int, double *, int, int, optkdNode *, int, int*,double*,int*);
void KillOptTree(optkdNode *);
void PQupheap(double *, int *, int);
void PQInsert(double, int, double *, int *);
void PQdownheap(double *, int *, int, int);
void PQreplace(double, double *, int *, int);
static int Count=0;

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{

  double numpoints, dimension, kneighbors, num_images;
  double* praMI;
  double *rrw;
  double gamma;


  /* Check for proper number of arguments. */
  if (nrhs != 6)
    mexErrMsgTxt("Six inputs required : kNNnonlincorrmex (rrw, num_images, N, dimension, kneighbors, gamma)");
  else if (nlhs > 1)
    mexErrMsgTxt("Only one output argument");

  /* Assign pointers to each input */
  /* Use mex syntax */
  /* matrices for each time point */
  rrw = (double *)mxGetPr(prhs[0]);
  num_images =  *mxGetPr(prhs[1]);
  numpoints = *mxGetPr(prhs[2]);
  dimension = *mxGetPr(prhs[3]);
  kneighbors = *mxGetPr(prhs[4]);
  /*kneighbors = kneighbors + 1;*/
  gamma = *mxGetPr(prhs[5]);
  /* Create matrix for the return argument. */
  /* this is basically useless */
  plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);

  /* Assign pointer to the output */
  praMI = (double *) mxGetPr(plhs[0]);
  principal (numpoints, num_images, dimension, kneighbors, rrw, praMI, gamma);
}


/************Need*************************************************************/

void Selection(a, l,N, k,discrim,perm)

/*************************************************************************/
/* Makes the perm partition the array Values along the element k.        */
/* Adapted from Sedgewick's Algorithms in C (p. 128)                     */
/*************************************************************************/

double **a;
int l,N,k,discrim;
int * perm;
{
  double v;
  int t,i,j,r;
  r=N;
  while(r>l) {
    v=a[perm[r]][discrim]; i=l-1; j=r;
    for (;;) {
      while (a[perm[++i]][discrim] < v);
        while (a[perm[--j]][discrim] > v && j>l);
          if (i >= j) break;
            t=perm[i]; perm[i] = perm[j]; perm[j]=t;
    }
    t=perm[i]; perm[i] = perm[r]; perm[r]=t;
    if (i>=k) r=i-1;
    if (i<=k) l=i+1;
  }
}

/************Need*************************************************************/
```

```
117
118 int findmaxspread(l,u,dimension,points,perm)
119
120 /*************************************************************************/
121
122 int l,u,dimension;
123 double **points;
124 int * perm;
125 {
126 int i,j,maxdim;
127 double max       =-99e99,
128        min       = 99e99,
129        maxspread =-99e99;
130   for (i=0; i < dimension; i++) {
131     max =-99e99;
132     min = 99e99;
133     for (j=l; j <= u; j++) {
134       if (max < points[perm[j]][i]) {
135         max = points[perm[j]][i];
136       }
137       if (min > points[perm[j]][i]) {
138       min = points[perm[j]][i];
139       }
140       if (maxspread < fabs(max-min)) {
141         maxspread = fabs(max-min);
142         maxdim = i;
143       }
144     }
145   }
146 return(maxdim);
147 }
148
149 /********Need*********************************************************************/
150
151 optkdNode *BuildkdTree(points,l,u,dimension,perm)
152
153 /*************************************************************************/
154
155 int l,u;
156 double **points;
157 int * perm;
158 {
159   optkdNode *p;
160   int m;
161   /*   printf("allocating 8\n");*/
162   NEWTREE(p);
163   if (u-l+1 <= BUCKETSIZE) {
164     p->bucket = 1;
165     p->lopt = l;
166     p->hipt = u;
167     p->loson = NULL;
168     p->hison = NULL;
169   } else {
170     p->bucket =0;
171     p->discrim = findmaxspread(l,u,dimension,points,perm);
172     m=(l+u)/2;
173     Selection(points,l,u,m,p->discrim,perm);
174     p->cutval = points[perm[m]][p->discrim];
175     p->loson = BuildkdTree(points,l,m,dimension,perm);
176     p->hison = BuildkdTree(points,m+1,u,dimension,perm);
177   }
178   return(p);
179 }
180
181 /***Need*******************************************************************/
182
183 optkdNode *BuildOptTree(points,numPoints,dimension,perm)
184
185 /*************************************************************************/
186 int dimension,numPoints;
187 double **points;
188 int * perm;
189 {
190
191   int j;
192   /* initialize perm array */
193   /*  printf("allocating 1\n");*/
194
195   for (j=0; j < numPoints; j++) {
196     perm[j]=j;
197   }
198   return(BuildkdTree(points,0,numPoints-1,dimension,perm));
199 }
200
201
202 /**********Need*********************************************************************/
203
204 void rnnEuclidean(p,querpoint,points,dimension,numpoints,optfound,nndist,perm)
205
206 /*************************************************************************/
207
208 /* special searching algorithm to take advantage of the fact that square roots
209    do not need to be evaulated */
210
211 optkdNode *p;
212 double *querpoint;
213 double **points;
214 int dimension,numpoints;
215 int * optfound;
216 double *nndist;
217 int * perm;
218 {
219   int i,j,k;
220   double d,thisdist,val,thisx;
221   /*printf("in here 2 alroght\n");*/
222   if (p->bucket) {
223     for (i=p->lopt; i <= p->hipt; i++) {
224       thisdist=0.0;
225       for (j=0; j<dimension; j++) {
226         d=(querpoint[j]-points[perm[i]][j]);
227         thisdist=thisdist+d*d;
228       }
229       if (optfound[0] < numpoints && thisdist > 0.0) {
230         PQInsert(thisdist,perm[i],nndist,optfound);
231       } else if (thisdist > 0.0 ) {
232         PQreplace(thisdist,nndist,optfound,perm[i]);
233       }
234     }
235   } else {
236     val = querpoint[p->discrim] - p->cutval;
237     if (val < 0) {
238       rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound,nndist,perm);
239       if (nndist[1] >= val*val) {
240         rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound,nndist,perm);
```

```
      }
    } else {
      rnnEuclidean(p->hison,querpoint,points,dimension,numpoints,optfound,nndist,perm);
      if (nndist[1] >= val*val) {
        rnnEuclidean(p->loson,querpoint,points,dimension,numpoints,optfound,nndist,perm);
      }
    }
  }
}


/****************Need*************************************************************/

int *kdOptNNQuery(points,dimension, querpoint,numNN,Metric,root,MinkP,optfound,nndist,perm)

/*********************************************************************************/

optkdNode *root;
double *querpoint, **points;
int dimension,numNN,MinkP;
int * optfound;
double * nndist;
int * perm;
{
  int j;
  /*int *optfound;*/
  /* set up found array */
  /*  printf("allocating 2\n");*/
  /*optfound = (int *) mxMalloc((numNN+1)*sizeof(int));*/
  optfound[0]=1;  /* for now */
  /* nndist is a priority queue of the distances of the nearest neighbors found */
  /*  printf("allocating 3\n");*/
  /*  nndist = (double *)mxMalloc((numNN+1)*sizeof(double));*/
  for (j=0; j < numNN+1; j++) {
    nndist[j] = 99e99;
  }
  /*printf("in here alright\n");*/
  switch(Metric) {
    case EUCLIDEAN :  rnnEuclidean(root,querpoint,points,dimension,numNN,optfound,nndist,perm);
    break;
  /*case MANHATTAN : Distance=ManhattDist;
    rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
    break;
  case L_INFINITY: Distance=LInfinityDist;
    rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
    break;
  case L_P       : Distance=LGeneralDist;
    rnnGeneral(root,querpoint,points,dimension,numNN,MinkP);
    break;*/
  }
  /*for (j=0;j<numNN;j++)
  printf("%g\n",nndist[j]);
    free(nndist);*/
  return(optfound);
  /*free(optfound);*/
}


/******Need*************************************************************************/

void KillOptTree(P)

/*********************************************************************************/

/*  Kills a kd-tree to avoid memory holes.   */


optkdNode *P;

{
  /*if (perm != NULL) {
    free(perm);
  }  /* free permutation array */

  if (P==NULL) {
    return;
  } /* just to be sure */
  if (P->loson != NULL) {
    KillOptTree(P->loson);
  }

  if (P->hison != NULL) {
    KillOptTree(P->hison);
  }

  free(P);

}



/*********************************************************************************/
/*  Code to implement the abstract data type priority queue for use in j nearest     */
/*  neighbor searching.  Actual implementation is done using heaps.                  */
/*                                                                                   */
/*  Adapted from Sedgewick's: Algorithms in C p. 148-160.                            */
/*********************************************************************************/

/*
   The heap data structure consists of two priority queues.  One for the j-smallest
   distances encountered, one to keep the indexes into the points array of the
   points corresponding to the j-smallest distances.
*/



/*********Need*****************************************************************************/

void PQupheap(DistArr,FoundArr,k)

/*********************************************************************************/

double *DistArr;  /* j-smallest distances encountered */

int *FoundArr,k;

{
  double v;
  int j;

  v=DistArr[k]; DistArr[0] = 99e99;
  j=FoundArr[k];

  while(DistArr[k/2] <= v) {
```

```
    DistArr[k] = DistArr[k/2];
    FoundArr[k] = FoundArr[k/2];
    k=k/2;
  }
  DistArr[k] = v;
  FoundArr[k] = j;
}

/**********Need************************************************************************/

void PQInsert(distance,index,DistArr,FoundArr)

/*****************************************************************************/

double distance,*DistArr;
int   index, *FoundArr;

{
  FoundArr[0]=FoundArr[0]+1;
  DistArr[FoundArr[0]] = distance;
  FoundArr[FoundArr[0]] = index;
  PQupheap(DistArr,FoundArr,FoundArr[0]);
}



/***********Need************************************************************************/

void PQdownheap(DistArr,FoundArr,k,index)

/*****************************************************************************/

double *DistArr;  /* j-smallest distances encountered */

int *FoundArr,k,index;

{

  int j,l,N;
  double v;

  v=DistArr[k];
  N = FoundArr[0];  /* tricky patch to maintain the data structure */
  FoundArr[0]=index;

  while (k <= N/2) {
    j=k+k;
    if (j < N && DistArr[j] <DistArr[j+1]) j++;
    if (v>=DistArr[j]) break;
    DistArr[k]=DistArr[j];
    FoundArr[k]=FoundArr[j];
    k=j;
  }

  DistArr[k] = v;
  FoundArr[k]= index;
  FoundArr[0]=N;  /* restore data struct */


}

/************Need************************************************************************/

void PQreplace(distance,DistArr,FoundArr,index)

/*****************************************************************************/

double *DistArr,distance;
int *FoundArr;

{
  DistArr[0]=distance;
  PQdownheap(DistArr,FoundArr,0,index);
}


/*************************************************************************************************/

void principal (double numpoints, double num_images, double dimension, double kneighbors, double *rrw, double *praMI, double gamma)

/*************************************************************************************************/
{
  double **Points, **Points1, *distx;
  double *querpoint, *querpoint1;
  optkdNode *OptkdRoot, *OptkdRoot1, *OptkdRoot2;
  int *Found, MinkP;
  int Metric;
  int i, j, k, j1, n, ii;
  double * nndist, distx1, temp_distx1;
        int *optfound,*optfound1;
        int *perm, *perm1, *perm2;
        Metric = EUCLIDEAN;
  MinkP = 0;
  numpoints = (int) numpoints;
  dimension = (int) dimension;
  kneighbors = (int) kneighbors;
  /*printf("allocating 4\n");*/
  /*if (dimension/num_images != 1)*/
    querpoint=(double *) mxMalloc(sizeof(double)*(dimension/num_images));
  querpoint1=(double *) mxMalloc(sizeof(double)*(dimension));
  /*printf("allocating 5\n");*/
  Points = (double **)mxMalloc((numpoints)*sizeof(double *));
  /*hist2 = (double **)mxMalloc((255)*sizeof(double *));
  linear_hist2 = (double *)mxMalloc((numpoints)*sizeof(double));*/
  /*printf("allocating 6\n");*/

  for (k = 0; k < numpoints; k++) {
    Points[k] = (double *)mxMalloc((dimension)*sizeof(double));
  }
  for (k = 0, i = 0; k < numpoints; k++) {
    for (j = 0; j < dimension; j++, i++) {
      Points[k][j] = rrw[i];
    }
  }


  /* printf("allocating 7\n");*/
  /*if (dimension/num_images != 1) {*/
    Points1 = (double **)mxMalloc((numpoints)*sizeof(double *));
    for (k = 0; k < numpoints; k++) {
      Points1[k] = (double *)mxMalloc((dimension/num_images)*sizeof(double));
    }
  /*}*/
```

```
  /*printf("%d\n",i);*/
  /*printf("%g numpoints, %g dimension",numpoints,dimension);*/
  *praMI = 0.0;


  optfound1 = (int *) mxMalloc((kneighbors + 2)*sizeof(int));
  nndist = (double *)mxMalloc((kneighbors + 2)*sizeof(double));
  perm  = (int *) mxMalloc(numpoints*sizeof(int));
  OptkdRoot  = BuildOptTree(Points,  numpoints, dimension,   perm);
  /*if (dimension/num_images != 1) {*/
    perm1 = (int *) mxMalloc(numpoints*sizeof(int));
    optfound  = (int *) mxMalloc((kneighbors + 2)*sizeof(int));
/*  }*/

  distx = (double *) mxMalloc(numpoints * sizeof(double));
  for (i=0; i<numpoints; i++) {
    distx[i] = 0.0;
      for (j = 0; j < dimension; j++)
      querpoint1[j]=Points[i][j];
      Found = kdOptNNQuery (Points, dimension, querpoint1, kneighbors, Metric, OptkdRoot, MinkP, optfound1, nndist,perm);
      for (j = kneighbors; j >= 1; j--) {
      if (Found[j]!=i) distx[i] += sqrt(nndist[j]);
    }
    distx[i] = distx[i]/kneighbors;
  }

  /*if (dimension/num_images != 1) {*/
    for (n = 0; n  < num_images; n++) {
      for (k = 0; k < numpoints; k++)
        for (i = 0, j = n*(dimension/num_images); j < (n+1)*(dimension/num_images); j++, i++)
          Points1[k][i] = Points[k][j];
      OptkdRoot1 = BuildOptTree(Points1, numpoints, dimension/num_images, perm1);
      for (i=0; i<numpoints; i++) {
        distx1 = 0.0;
        for (ii = 0, j = n*(dimension/num_images); j < (n+1)*(dimension/num_images); j++, ii++)
          querpoint[ii]=Points[i][j];
        Found = kdOptNNQuery(Points1,dimension/num_images,querpoint,kneighbors,Metric,OptkdRoot1,MinkP,optfound,nndist,perm1);
        for (j = kneighbors; j>=1; j--) {
          if (Found[j]!=i) {
            temp_distx1 = 0.0;
            for (ii = 0; ii < dimension; ii++)
              temp_distx1 += pow((Points[i][ii] - Points[Found[j]][ii]),2);
            distx1 += sqrt(temp_distx1);
          }
        }
        distx1 = distx1/kneighbors;
        /*printf("distx %g  distx1 %g  distx2 %g linear_hist2 %g\n",distx,distx1,distx2,linear_hist2[i]);*/
        distx[i] = distx[i]/(pow(distx1,(1/num_images)));
        /*else { *praMI = linear_hist2[i]*pow(distx[i], (num_images)*gamma);}*/
        /*printf("prami %g nowadded %g",*praMI,pow(distx/(pow(distx1*distx2,0.5)), 2*gamma));}*/
      }
    }
  /*}*/

  for (i = 0; i < numpoints; i++)
    *praMI += pow(distx[i], num_images*gamma);

  /**praMI = log(*praMI);*/
  /*printf("aMI is %g\n",*praMI);*/
  /*printf("distx1 = %g, distx2 = %g, distx = %g\n",distx1,distx2,distx);*/
}
```

End of kNN-based NLCC estimation program

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] http://www.eecs.umich.edu/ hero/research.html, Website of Alfred O. Hero, Professor EECS, University of Michigan, Ann Arbor.

[2] http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html, Matlab Reference website for external program interface.

[3] A.J.Ratches, C.P Walters, R.G. Buser, and B.D. Guenther. Aided and automatic target recognition based upon sensory inputs from image forming systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):1004–1019, 1997.

[4] Y. Amit and D. Geman. Shape quantization and recognition with randomized trees. *Neural Computation*, 9:1545–1588, 1997.

[5] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[6] J. Ashley, R. Barber, M. Flickner, D. Lee, W. Niblack, and D. Petkovic. Automatic and semiautomatic methods for image annotation and retrieval in qbic. In *Proc. SPIE Storage and Retrieval for Image and Video Databases III*, pages 24–35.

[7] S. Baase. *Computer Algorithms*. Addison-Wesley, 1988.

[8] R. Baraniuk, P. Flandrin, A. J. E. M. Jensen, and O. Michel. Measuring time frequency information content using the Rényi entropies. *IEEE Trans. on Inform. Theory*, IT-47(4), April 2001.

[9] M. Basseville. Distance measures for signal processing and pattern recognition. *Signal Processing*, 18:349–369, 1989.

[10] C. F. Bazlamaçci and K. Hindi. Minimum-weight spanning tree algorithms: A survey and empirical study. *Computers and Operations Research*, 28:767–785, 2001.

[11] J. Beardwood, J. H. Halton, and J. M. Hammersley. The shortest path through many points. *Proc. Cambridge Philosophical Society*, 55:299–327, 1959.

[12] J. Beirlant, E. J. Dudewicz, L. Györfi, and E.C. van der Meulen. Nonparametric entropy estimation: an overview. *Intern. J. Math. Stat. Sci.*, 6(1):17–39, june 1997.

[13] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Software Engineering*, SE-5(4):333–340, 1979.

[14] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, Sept. 1975.

[15] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communic. Assoc. Comput. Mach.*, 18:509–517, 1975.

[16] P. Bhatti, G. LeCarpentier, M. Roubidoux, J. Fowlkes, M. Helvie, and P. Carson. Discrimination of sonographic breast lesions using frequency shift color doppler imaging, age and gray scale criteria. *Journal of Ultrasound Medicine*, 20(4):343–350, 2001.

[17] T. Butz and J. Thiran. Affine registration with feature space mututal information. In *Lecture Notes in Computer Science 2208: MICCAI 2001*, pages 549–556.

[18] Studholme C. and Cardenas V. A template-free approach to volumetric spatial normalization of brain anatomy. *Pattern Recognition Letters*, 25:1191–1202, 2004.

[19] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, Sept. 2001.

[20] R. Chellappa, C. L. Wilson, and S. Sirohey. Human and machine recognition of faces: A survey. *IEEE Proceedings*, 83(5):705–740, 1995.

[21] D. Cheriton and R. Tarjan. Finding minimum spanning trees. *SIAM Journal on Computing*, 5:724–742, 1976.

[22] J.W. Choi, T. H. Fang, W. Yoo, and M.H. Lee. Sensor data fusion using perception net for a precise assembly task. *IEEE/ASME Transactions on Mechatronics*, 8(4):513–516, 2003.

[23] Y. Choi and S. Lee. Injectivity conditions of 2D and 3D uniform cubic B-Spline functions. *Graphical Models*, 62:411–427, 2000.

[24] S. R. Cloude and E. Pottier. An entropy based classification scheme for land applications of polarimetric SAR. *IEEE Trans. on Geoscience and Remote Sensing*, 75:pp. 68–78, 1997.

[25] C. A. Cocosco, V. Kollokian, R. K. S. Kwan, and A. C. Evans. Brainweb: Online interface to a 3D MRI simulated brain database. *NeuroImage*, 5(4), 1997.

[26] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, Englewood-Cliffs NJ, 1990.

[27] N. Cristiani and J. Shaw-Taylor. *Suport Vector Machines and other kernel-based learning methods*. Cambridge U. Press, 2000.

[28] I. Csiszár. Information-type measures of divergence of probability distributions and indirect observations. *Studia Sci. Math. Hung.*, 2:299–318, 1967.

[29] J. S. de Bonet and P. Viola. Structure driven image database retrieval. In *Advances in neural information processing*, volume 10, 1997.

[30] A. Dembo and O. Zeitouni. *Large deviations techniques and applications*. Springer-Verlag, NY, 1998.

[31] M. N. Do and M. Vetterli. Texture similarity measurement using Kullback-Liebler distance on wavelet subbands. In *IEEE Int. Conf. on Image Processing*, pages 367–370, Vancouver, BC, 2000.

[32] D. Dunn, W. Higgins, and J. Wakeley. Texture segmentation using 2d gabor elementary functions. *IEEE Trans. Pattern Anal. Mach. Intelligence*, 16(2):130–149, 1994.

[33] Equinox Corporation. Human identification at distance project: http://www.equinoxsensors.com/products/hid.html/.

[34] Y. Erdi, K. Rosenzweig, A. Erdi, H. Macapinlac, Y. Hu, L. Braban, J. Humm, O. Squire, C. Chui, S. Larson, and E. Yorke. Radiotherapy treatment planning for patients with non-small cell lung cancer using pet. *Radiotherapy and Oncology*, 62(1):51–60, 2002.

[35] V. Erdogmus, J. Prncipe, and L. Vielva. Blind deconvolution with minimum rényi's entropy. In *EUSIPCO*, Toulouse, France, 2002.

[36] J. Fisher, T. Darrell, W. Freeman, and P. Viola. Learning joint statistical models for audio-visual fusion and segregation. In *Advances in Neural Information Processing Systems*, Denver, Colorado, 2000.

[37] J. Fisher and A. Willsky. Information theoretic feature extraction for atr. In *Proc. of. 34th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, 1999.

[38] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization. *Journal of ACM*, 34:596–615, 1987.

[39] B.R. Frieden and Anisa T. Bajkova. Reconstruction of complex signals using minimum rényi information. In *Proc. of Meeting of Intl. Soc. for Optical Engin. (SPIE)*, volume 2298, 1994.

[40] J. H. Friedman, F. Baskett, and L. J. Shustek. An algorithm for finding nearest neighbors. *IEEE Trans. on Computers*, C-24, 1975.

[41] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3), 1977.

[42] Jerome H. Friedman and Lawrence C. Rafsky. Multivariate generalizations of the Wald-Wolfowitz and Smirnov two-sample tests. *Annals of Statistics*, 7(4):697–717, 1979.

[43] H. N. Gabow, Z. Galil, T. H. Spencer, and R. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.

[44] D. Geman and Koloydenko A. Invariant statistics and coding of natural microimages. In *IEEE Workshop on Statist. Computat. Theories of Vision*, Fort Collins, CO, June 1999.

[45] Sébastien Gilles. Description and experimentation of image matching using mutual information. Technical report, Oxford University, 1996. `www-rocq.inria.fr/~gilles/IMMMI/mutual_info.ps.gz`.

[46] F. Glover, D. Klingman, R. Krishnan, and A Padman. An in-depth emperical investigation of non-greedy approached for the minimum spanning tree problem. *European Journal of Operations Research*, 56:353–356, 1992.

[47] P. Gordon and S. Goldenberg. Malignant breast masses detected only by ultrasound. *Cancer*, 76:626–630, 1995.

[48] S. Haney, P. Thompson, T. Cloughesy, J. Alger, and A.Toga. Tracking tumor growth rates in patients with malignant gliomas: A test of two algorithms. *Americal Journal of Neuroradiology*, 22(1):73–82, 2001.

[49] Y. He, A. Ben-Hamza, and H. Krim. An information divergence measure for ISAR image registration. *Signal Processing*, Submitted, 2001.

[50] N. Henze and M. Penrose. On the multivariate runs test. *Annals of Statistics*, 27:290–298, 1999.

[51] A. O. Hero, J. Costa, and B. Ma. Asymptotic relations between minimal graphs and alpha entropy. Technical Report 334, Comm. and Sig. Proc. Lab. (CSPL), Dept. EECS, University of Michigan, Ann Arbor, Mar, 2003. `www.eecs.umich.edu/~hero/det_est.html`.

[52] A. O. Hero, B. Ma, and O. Michel. Imaging applications of stochastic minimal graphs. In *IEEE Int. Conf. on Image Processing*, Thessaloniki, Greece, October 2001.

[53] A. O. Hero, B. Ma, O. Michel, and J. D. Gorman. Alpha-divergence for classification, indexing and retrieval. Technical Report 328, Comm. and Sig. Proc.

Lab. (CSPL), Dept. EECS, University of Michigan, Ann Arbor, July, 2001. www.eecs.umich.edu/~hero/det_est.html.

[54] A.O. Hero, J. Costa, and B. Ma. Convergence rates of minimal graphs with random vertices. *IEEE Trans. on Inform. Theory*, submitted, 2002.

[55] A.O. Hero, B. Ma, O. Michel, and J. Gorman. Applications of entropic spanning graphs. *IEEE Signal Processing Magazine*, 19(5):85–95, Sept. 2002.

[56] A.O. Hero and O. Michel. Asymptotic theory of greedy approximations to minimal k-point random graphs. *IEEE Trans. on Inform. Theory*, IT-45(6):1921–1939, Sept. 1999.

[57] D. Hill, P Batchelor, M. Holden, and D. Hawkes. Medical image registration. *Phys. Med. Biol.*, 26:R1–R45, 2001.

[58] R. Hoffman and A. K. Jain. A test of randomness based on the minimal spanning tree. *Pattern Recognition Letters*, 1:175–180, 1983.

[59] J. Huang, S. Kumar, M. Mitra, and W. Zhu. Spatial color indexing and applications. In *Proc. of IEEE Int'l Conf. Computer Vision ICCV'98*, pages 602–608.

[60] A. Hyvärinen. Fast ICA Code. www.cis.hut.fi/projects/ica/fastica/.

[61] A. Hyvärinen and E. Oja. Independent component analysis: algorithms and applications. *Neural Networks*, 13(4-5):411–430, 1999.

[62] R. Jassemi-Zargani and D. Necsulescu. Extended kalman filter-based sensor fusion for operational space control of a robot arm. *IEEE Trans. on Instrumentation and Measurement*, 51(6):1279–1282, Dec. 2002.

[63] Mark Jenkinson, Peter Bannister, Michael Brady, and Stephen Smith. Improved methods for the registration and motion correction of brain images. Technical report, Oxford University, 2002.

[64] K. Johnson, A. Cole-Rhodes, I. Zavorin, and J. Le Moigne. Multi-resolution image registration of remotely sensed imagery using mutual information. In *Proc. of SPIE OE/Aerospace Sensing, Wavelet Applications VIII*, Orlando, FL, 2001.

[65] A. Kapur, P. Carson, J. Eberhard, M. Goodsitt, K. Thomenius, M. Lokhandwalla M, D. Buckley, R. Hoctor, M. Roubidoux, M. Helvie, C. Booi, G. LeCarpentier, R. Erkamp, H-P Chan, J. Fowlkes A. Dattamajumdar, A. Hall, J. Thomas, and C. Landberg. Combination of digital mammography with semi-automated 3d breast ultrasound. *Technology in Cancer Research and Treatment*, 3:325–334, 2004.

[66] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of Association for Computing Machinery*, 42(2):321–328, 1995.

[67] R. Kedar, D. Cosgrove, I. Smith, J. Mansi, and J. Bamber. Breast carcinoma: Measurement of tumor response to primary medical therapy with color doppler flow imaging. *Radiology*, 190:825–830, 1994.

[68] T. Kieu and P. Viola. Boosting image retrieval. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2000.

[69] T. Kolb, J. Lichy, and J. Newhouse. Occult cancer in women with dense breasts: Detection with screening ultrasound: Diagnostic yield and tumor characteristics. *Radiology*, 207:191–198, 1998.

[70] C. Kreucher, K. Kastella, and A. O. Hero. Sensor management using relevance feedback learning. *IEEE Trans. on Signal Processing*, submitted, 2001.

[71] J. F. Krücker, , C.R. Meyer, G.L. LeCarpentier, J.B. Fowlkes, and P.L. Carson. 3d spatial compounding of ultrasound images using image-based nonrigid registration. *Ultrasound in Medicine and Biology*, 26(9):1475–1488, 2000.

[72] J. F. Krücker, G.L. LeCarpentier, J.B. Fowlkes, and P.L. Carson. Rapid elastic image registration for 3d ultrasound. *IEEE Transactions on Medical Imaging*, 21(11):1384–1394, 2002.

[73] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[74] S. Kullback and R.A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22:79–86, 1951.

[75] M. Lefébure and L. Cohen. Image registration, optical flow and local rigidity. *J. Mathematical Imaging and Vision*, 14(2):131–147, 2001.

[76] Michael E. Leventon and W. Eric L. Grimson. Multi-modal volume registration using joint intensity distributions. Technical report, MIT AI Laboratory, 1998. `www.ai.mit.edu/projects/vision-surgery`.

[77] M. Lewicki and B. Olshausen. Probabilistic framework for the adaptation and comparison of image codes. *J. Opt. Soc. Am.*, 16(7):1587–1601, 1999.

[78] W. Li and H. Leung. Simultaneous registration and fusion of multiple dissimilar sensors for cooperative driving. *IEEE Trans. on Intelligent Transportation Systems*, 5(2):84–98, 2004.

[79] Y. Linde, A. Buzo, and R. M. Gray. An algorithm for vector quantization design. *IEEE Trans. on Communication*, 28:84–95, 1980.

[80] C. Liu and H. Wechsler. Comparative assessment of independent component analysis. In *Proc. the 2nd International Conference on Audioand Video-based Biometric Person Authentication*, pages 22–24, Washington D. C., March 1999.

[81] Yanxi Liu, Robert T. Collins, and William E. Rothfus. Robust midsagittal plane extraction from coarse, pathological 3d images. *IEEE Trans. on Image Processing*, 9(1):132–137, 2000.

[82] B. Ma. *Parametric and non-parametric approaches for multisensor data fusion*. PhD thesis, University of Michigan, Ann Arbor, MI 48109-2122, 2001. `www.eecs.umich.edu/~hero/research.html`.

[83] W. Ma and B. Manjunath. Netra: A toolbox for navigating large image databases. In *Proc. of IEEE Int. Conf. on Image Processing*, volume 1, pages 568–571, 1997.

[84] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens. Multimodality image registration by maximization of mutual information. *IEEE Transactions on Medical Imaging*, 16(2):187–198, Apr. 1997.

[85] F. Maes, D. Vandermeulen, and P. Suetens. Medical image registration using mutual information. *Proceedings of the IEEE,* 91(10):1699–1722, 2003.

[86] J. B. Maintz and M. Viergever. A survey of medical image registration. *Medical Image Analysis*, 2(1):1–36, 1998.

[87] C. R. Meyer, J. L. Boes, B. Kim, P. H. Bland, K. R. Zasadny, P. V. Kison, K. F. Koral, K. A. Frey, and R. L. Wahl. Demonstration of accuracy and clinical versatility of mutual information for automatic multimodality image fusion using affine and thin-plate spline warped geometric deformations. *Medical Image Analysis*, 1(3):195–206, Apr. 1997.

[88] C.R. Meyer, J.L. Boes, B. Kim, P.H. Bland, G.L. LeCarpentier, J.B. Fowlkes, M.A. Roubidoux, and P.L. Carson. Semiautomatic registration of volumetric ultrasound scans. *Ultrasound in Medicine and Biology*, 25(3):339–347, 1999.

[89] O. Michel, R. Baraniuk, and P. Flandrin. Time-frequency based distance and divergence measures. In *IEEE International time-frequency and Time-Scale Analysis Symposium*, pages 64–67, Oct 1994.

[90] N. Milisavljevic and I. Bloch. Sensor fusion in anti-personnel mine detection using a two-level belief function model. *IEEE Transactions on Systems, Man and Cybernetics*, 33(2):269–283, 2003.

[91] B. M. E. Moret and D. Shapiro. How to find a minimum spanning tree in practice. *Lecture Notes in Computer Science*, 555:192–203, 1991.

[92] D. Mount and S. Arya. Approximate Nearest Neighbor Code. `http://www.cs.umd.edu/~mount/ANN`.

[93] NASA Visible Earth internet site: `http://visibleearth.nasa.gov/`.

[94] H. Neemuchwala, A. Hero, and P. Carson. Image registration using entropic graph matching criteria. In *Proc. of Asilomar Conference*, Monterey, CA, November 2002.

[95] H. Neemuchwala, A. Hero, P. Carson, and C. Meyer. Local feature matching using entropic graphs. In *Proc. of the IEEE International Symposium on Biomedical Imaging*.

[96] H. Neemuchwala, A. O. Hero, and P. Carson. Image matching using alpha-entropy measures and entropic graphs. *European Journal of Signal Processing, Special Issue on Content based image retrieval*, Accepted, 2004.

[97] H. Neemuchwala, A. O. Hero, and P. Carson. Feature coincidence trees for registration of ultrasound breast images. In *IEEE Int. Conf. on Image Processing*, Thessaloniki, Greece, Oct. 2001.

[98] S. A. Nene and S. K. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. on Pattern Anal. and Machine Intell.*, 19, 1997.

[99] J. Nesetril, E. Milkova, and H. Nesetrilova. Otakar Boruvka on minimum spanning tree problem (translation of both 1926 papers, comments, history). *DMATH: Discrete Mathematics*, 233, 2001.

[100] A.B. Nobel and R.A. Olshen. Termination and continuity of greedy growing for tree-structured vector quantizers. *IEEE Trans. on Inform. Theory*, IT-42(1):191–205, 1996.

[101] B. Olshausen. *Sparse codes and spikes*. MIT Press, 2001.

[102] H. Park and C. Meyer. Grid refinement in adaptive non-rigid registration. *Lecture Notes in Computer Science, MICCAI*, 2003.

[103] C. Penney, J. Weese, J. Little, D. Hill, and D. Hawkes. A comparison of similarity measures for used in 2-D-3-D medical image registration. *IEEE Trans. on Medical Imaging*, 17(4):586–595, 1998.

[104] R. C. Prim. Shortest connection networks and some generalizations. *Bell Syst. Tech. Journ.*, 36:1389–1401, 1957.

[105] Project Atlanta NASA Marshall Space Flight Center, Huntsville, Alabama.

[106] A. Rangarajan, I.-T. Hsiao, and G. Gindi. Integrating anatomical priors in ect reconstruction via joint mixtures and mutual information. In *IEEE Medical Imaging Conference and Symposium on Nuclear Science*, volume III, Oct. 1998.

[107] C. Redmond and J. E. Yukich. Asymptotics for Euclidean functionals with power weighted edges. *Stochastic Processes and their Applications*, 6:289–304, 1996.

[108] A. Rényi. On measures of entropy and information. In *Proc. 4th Berkeley Symp. Math. Stat. and Prob.*, volume 1, pages 547–561, 1961.

[109] T. Rohlfing, J. West, J. Beier, T. Liebig, C. Tachner, and U. Thornale. Registration of functional and anatomical mri: accuracy, assessment and applications in navigated neurosurgery. *Computer Aided Surgery*, 5(6):414–425, 2000.

[110] D. Rueckert, M. Clarkson, D. Hill, and D. Hawkes. Non-rigid registration using higher order mutual information. In *Proc. SPIE*, pages 438–447.

[111] G. Simone, A. Farina, F.C. Morabito, S.B. Serpico, and L. Bruzzone. Image fusion techniques for remote sensing applications. *Information Fusion*, 3:3–15, 2002.

[112] A. Srivastava, A. B. Lee, E. P. Simoncelli, and S. C. Zhu. On advances in statistical modeling of natural images. *Journal of Mathematical Imaging and Vision*, 18(1), Jan. 2003.

[113] A. Stavros, D. Thickman, C. Rapp, M. Dennis, S. Parker, and G. Sisney. Solid breast nodules: use of sonography to distinguish between benign and malignant lesions. *Radiology*, 196:123–134, 1995.

[114] J. M. Steele. *Probability theory and combinatorial optimization*, volume 69 of *CBMF-NSF regional conferences in applied mathematics*. Society for Industrial and Applied Mathematics (SIAM), 1997.

[115] R. Stoica, J. Zerubia, and J. M. Francos. Image retrieval and indexing: A hierarchical approach in computing the distance between textured images. In *IEEE Int. Conf. on Image Processing*, Chicago, Oct. 1998.

[116] R. Stoica, J. Zerubia, and J. M. Francos. The two-dimensional wold decomposition for segmentation and indexing in image libraries. In *Proc. IEEE Int. Conf. Acoust., Speech, and Sig. Proc.*, Seattle, May 1998.

[117] H. Stone, J. Le Moigne, and M. McGuire. The translation sensitivity of wavelet-based registration. *IEEE Trans. Pattern Anal. Mach. Intelligence*, 21(10):1074–1081, 1999.

[118] I. J. Taneja. New developments in generalized information measures. *Advances in Imaging and Electron Physics*, 91:37–135, 1995.

[119] R. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[120] A. Toga. *Brain Warping*. Academic Press, ISBN: 0126925356, 1999.

[121] D. Tomazevic, B. Likar, T. Slivnik, and F. Pernus. 3-d/2-d registration of ct and mr to x-ray images. *IEEE Transactions on Medical Imaging*, 22(11):1407–1416, 2003.

[122] N. Vasconcelos and A. Lippman. A Bayesian framework for content-based indexing and retrieval. In *IEEE Data Compression Conference*, Snowbird, Utah, 1998. `nuno.www.media.mit.edu/people/nuno/`.

[123] N. Vasconcelos and A. Lippman. Bayesian representations and learning mechanisms for content based image retrieval. In *SPIE Storage and Retrieval for Media Databases 2000*, San Jose, CA, 2000. `nuno.www.media.mit.edu/people/nuno/`.

[124] O. Vasicek. A test for normality based on sample entropy. *J. Royal Statistical Society, Ser. B*, 38:54–59, 1976.

[125] P. Viola and W. M. Wells III. Alignment by maximization of mutual information. In *Proceedings of IEEE International Conference on Computer Vision*, pages 16–23, Los Alamitos, CA, Jun. 1995.

[126] Toga W. and Thompson P. The role of image regsitration in brain mapping. *Image and Vision Computing*, 19:3–24, 2001.

[127] W. Wald and J. Wolfowitz. On a test whether two samples are from the same population. *Ann. Math. Statist.*, 11:147–162, 1940.

[128] W. J. Williams, M. L. Brown, and A. O. Hero. Uncertainty, information, and time-frequency distributions. In *Proc. of Meeting of Intl. Soc. for Optical Engin. (SPIE)*, volume 1566, pages 144–156, 1991.

[129] Y. Wu, T. Kanade, C. Li, and J. Cohn. Image registration using wavelet-based motion model. *International Journal of Computer Vision*, 38(2):129–152, 2000.

[130] Ming-Hsuan Yang, David J. Kriegman, and Narendra Ahuja. Detecting faces in images: A survey. *IEEE Trans. on Pattern Anal. and Machine Intell.*, 24(1):24–58, Jan 2002.

[131] A. Yao. An $O(|E|loglog|V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters*, 4:21–23, 1975.

[132] J. E. Yukich. *Probability theory of classical Euclidean optimization*, volume 1675 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1998.