Implementation of the Gamma test in MATLAB using a fast near neighbour search algorithm in C++

By Sunil Singh

A dissertation submitted in partial fulfilment of the requirement
for the degree of MSc

Department of Computer Science, Cardiff University

Supervisor Dr. Dafydd Evans

2005/6

DECLARATION

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed …… (candidate) Date ……

STATEMENT 1

This thesis is being submitted in partial fulfilment of the requirements for the degree of MSc Computing

Signed …… (candidate) Date ……

STATEMENT 2

This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references.

Signed …… (candidate) Date ……

STATEMENT 3

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed …… (candidate) Date ……

STATEMENT 4 - BAR ON ACCESS APPROVED

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loans after expiry of a bar on access approved by the Graduate Development Committee.

Signed …… (candidate) Date ……

Abstract

This project undertook the implementation of the Gamma test [24] in MATLAB^{TM} using a fast near neighbour search algorithm implemented in C++ that ran in O(*MlogM*) time, where *M* is the number of data points. The Gamma test is a statistical technique for estimating the extent to which a given set of data points can be modelled by an unknown smooth non-linear function. The implementation was successfully achieved on two platforms (Windows and Linux) with the use of a kd-tree data structure [11]. The Euclidean, Max and Manhattan Minkowski metrics were implemented for measuring the distance between data points. A performance investigation was carried out to assess run time efficiency.

Acknowledgements

I would like to thank my supervisor Dr. Dafydd Evans for all his supportive, informative and practical ideas during this project; and his guidance during some very difficult periods. His fine dynamics of supervision allowed me to explore issues relevant to the project without going too far down inappropriate avenues of thought.

I would also like to thank Prof. Antonia Jones for her timely advice and unending enthusiasm and ideas which steered me safely through many mine fields.

Thanks also go to Dr. Samuel Kemp of University Glamorgan for his help and encouragement with certain aspects of this project.

Last but not least, I am extremely grateful for the advice and support of some good friends.

Table of Contents

List of Figures

Chapter 1: Introduction

1.0 Overview

The Gamma test is an algorithm used for non-linear modelling and analysis. MATLAB^{TM} is a powerful numerical analysis and visualisation package. In order to make the Gamma test more accessible to the academic community, a MATLAB implementation was seen to be necessary.

In its less efficient form the Gamma test runs in O(M^2) time, where M is the number of data points. This is unfeasibly slow for most applications. However by using a fast near neighbour search algorithm [11], it is possible to run it in O($MlogM$) time. To achieve this efficiency the near neighbour search algorithm needed to be created as a standalone module written in a low level language such as C++ or Fortran. This module could then be called from the MATLAB command line. Further manipulation and visualisation of the data could then be carried out using built in MATLAB commands, thus allowing users to carry out their own experiments using the Gamma test.

It was also necessary to attain a quality finished software package, with various tools extending the core Gamma test algorithm (such as the M-test [1]), so that this package could be distributed to end users of various scientific disciplines.

1.1 Project Objectives

The main purpose of this project was to implement the Gamma test [17] in MATLAB using a fast near neighbour search algorithm. It was also required for this implementation to run on *both* Windows and Linux platforms.

It was necessary for the near neighbour search algorithm to calculate distances using

various Minkowski [26] metrics including the Euclidean, Max and Manhattan metrics. These alternative metrics were provided for experimental purposes.

It was also needed to *validate* the software to a high degree, by comparing results with known data sets. Also it was necessary to ensure the software package was reliable and maintainable; thus extensive error checking and modularity in design was needed. Performance tests were important to appreciate the efficiency of the near neighbour search with respect to metric, dimension, number of data points and approximation settings [3].

A further feature to this implementation was to provide an M-test [17]. The M-test (calculating Gamma statistics for increasing *M*) provides a way to evaluate the accuracy of the Gamma statistic. We do this by visual inspection of the extent to which the computed Gamma statistics have converged to a stable value. It was also required to implement heuristic confidence intervals recently developed by Jones and Kemp [1].

Other general objectives were:

- To read and understand the Gamma test and its wider applications
- To understand the main factors affecting performance of a near-neighbour search algorithm
- To understand C++ and MATLAB in depth

1.2 Scope of this project

This project's main focus was to implement the Gamma test with a fast near-neighbour search algorithm written in C++ that would interface with MATLAB. In order to do this it was also necessary to *validate* the work to a precise level in order to ensure correct implementation. This project took into account non-functional requirements such as maintainability, usability and reliability. The finished software package was expected to be used by scientists and engineers all over the world, thus needed to meet these non-functional requirements.

In terms of usability, it was aimed at fairly proficient MATLAB users; a graphical user interface was not provided. This project took reliability into account by checking for as many error conditions as possible, and catching these errors. The software might be further developed, thus maintainability was also taken into account. Some performance tests were also carried out to get a feel for the execution time behaviour of the algorithm under various conditions.

1.3 Layout of this dissertation

The layout of this dissertation reflects the general approach taken. The main objectives have been set out in the introduction (Chapter 1), and are fully defined in Chapter 3.

Chapter 2: Background

This chapter covers some broader issues related to the Gamma test, and how this tool fits into the computer modelling process in general. It describes the Gamma test itself as well as some typical applications. We also describe the MATLAB tool known as MEX, which allows *user defined modules* written in C++ to be called from the MATLAB command line. For this project, the 'user defined module' will be a near neighbour search algorithm, thus some theory related to these algorithms is described.

Chapter 3: Specification and development methodology

In this chapter the requirements specifications are set out in detail and a development methodology discussed.

Chapter 4: Preliminary Investigation

A near neighbour search algorithm library written by Mount et al. [3] was obtained, and its functionality was investigated before it could be used properly. This code also needed to be interfaced with MATLAB, thus it was necessary to understand the inner workings of MEX. From this *initial investigation* a design could be developed.

Chapter 5: Design

Diagrams of the core system design are described, from high level to lower level details.

Other components of design are also explained, such as the metric selection script, error checking and usability.

Chapter 6: Implementation

Creating the near neighbour libraries using a C++ Makefile[19] (for both platforms Windows and Linux) is described, as is MEX compilation. Implementation of the components of the system are described, including the interface between MATLAB and the kd-tree, the core Gamma test algorithm, metrics, error checking and the M-test. An overview of the entire package is given so that modularity of design can be appreciated and understood. Lastly, examples of figures produced by the package are shown.

Chapter 7: Validation

This chapter describes the uncovering of implementation problems, which were eventually resolved. It discusses validation of the near neighbour search; the system as a whole with a calibration data set; and finally error checking.

Chapter 8: Performance testing

This section looks at the performance of the implemented kd-tree near neighbour search algorithm when various factors were altered: approximation setting, dimension, number of data points and metrics.

Chapter 9: Evaluation

This chapter critically analyses the work done. It also evaluates the project in terms of difficulties faced, and how they were overcome.

Chapter 10: Future work

Future improvements to the software system are outlined as well as speculative ideas and applications.

Chapter 11: Conclusions

This chapter briefly sums up the software package created as compared with the original objectives.

Chapter 2: Background

2.1 Modelling in general

Science can be thought of as the search for relationships between variables. We search for these relationships by building models and comparing the results of our models to real world phenomena. If our model can generalise and predict the behaviour of the real world, a hypothesis can be tested to an extent depending on how closely the model reflects the real world. If a model is too abstracted from reality, it might only predict the behaviour of the real world for limited scenarios. In other words, the model is unable to generalise to previously unseen scenarios. A model, by being an abstraction of a real system can sometimes help us understand real life phenomena and its causative factors.

We can broadly take two approaches to modelling:

1. From theory (parametric)
2. Data-derived (non-parametric)

There is some overlap between these two approaches depending on what we understand about the system we are trying to model, and how much data is available.

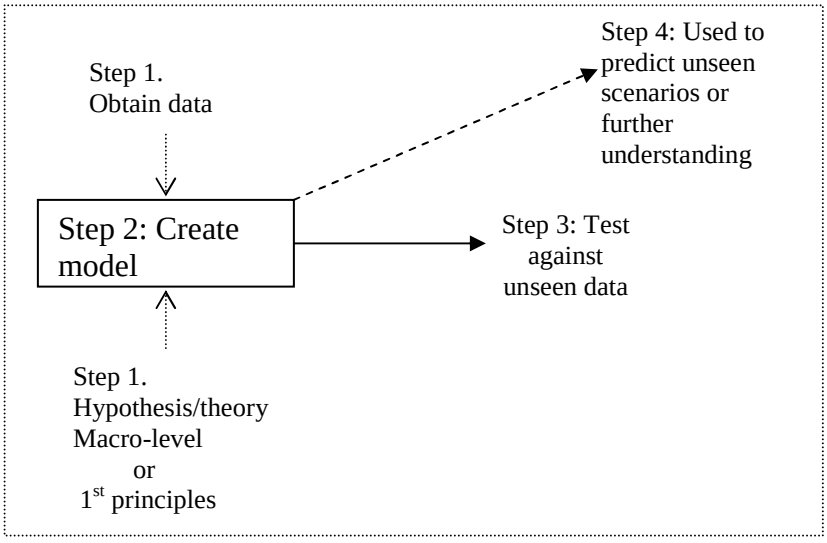

Figure 2.1: Illustrates two basic approaches to model building: data-driven or theory based. Diagram also illustrates the basic idea of testing the model against the real world phenomena and its subsequent use in predicting or generalising to unseen scenarios.

First we will describe the 'theory' approach. This method of modelling requires that we know something about the underlying components of the system, and how they interact. From this we can derive parametric, logical or probabilistic equations to describe the components and their relations to each other. The more we know about the underlying theory of the system, the more accurately we can model its finer details. However, we cannot begin at the atomic level if we are to have any realistic expectations of modelling things at a macro level. We should only include things we think are important to the model's behaviour. This is especially true for computer simulation, where we have finite processing power. For example, we would not model the weather by describing every atom in each rain drop in each cloud for the entire earth, as this would take longer to compute than the age of the universe. The reductionist [31] approach is infeasible for describing such large complex systems, and yet using abstractions or approximations can also be problematic as complex systems are often chaotic thus sensitive to initial conditions [32]. In the case of predicting the weather, the classic example is that of a butterfly flapping its wings in Tokyo could cause a hurricane in New York a week later. In computer models, the degree of abstraction we assume depends on the computational resources available, and how closely we want to understand the principles of the system. By taking a (relative) macro level approach, we can for example assume a parametric form between inputs and output of the system, then it is just a simple matter of performing a regression of these parameters to unseen data. However, we could be prone to over-simplifying the chosen parametric form, especially if we do not understand the underlying workings of the system. Thus we may decide, if sufficient data is available, to create our model *directly* from this data, without assuming beforehand what the relation is. This is known as data-derived modelling.

Data-derived modelling is particularly useful when we do not understand the inner workings of the system we wish to model, thus we treat it like a so called 'black box'. In some cases we may understand the physical laws that govern certain entities within the black box, but there are so many of them that their combined effect is impossible to model. By observing the real world as opposed to modelling it from first principles, we are more

likely to preserve the hidden interactions within the system that we were unable to model by intuitive understanding. We can construct the model directly from the data, but that does not necessarily help us understand the system. We may also have limited data of the observed system, and it may be subject to a high degree of noise.

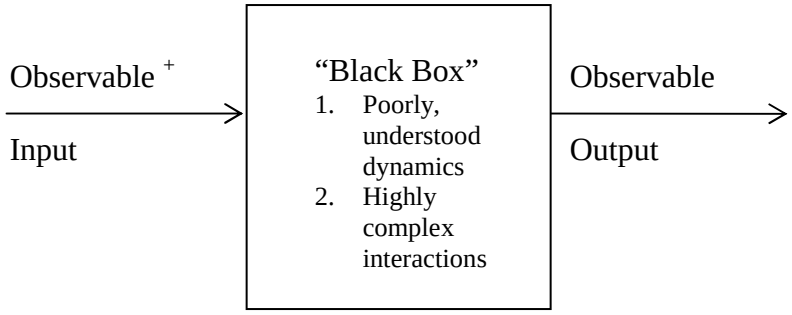

Figure 2.2: Emphasizing the 'black box' approach to modelling, where the model is constructed only from observation of inputs and outputs. In other words the model is data-derived. + In some cases, inputs can be set and thus considered noiseless, but in other systems the input can only be observed, and so will contain noise [2].

Whenever we observe phenomena in the natural world, we use some sort of measuring device. By definition we are discretely sampling a continuous reality (except at the quantum level). As a result, errors will be introduced by this sampling process, and also by imperfections in the measuring device. This noise will affect our ability to construct an accurate model. The Gamma test estimates the variance of the noise modulo the best possible smooth model (with bounded derivatives) present in observations of the real system, and therefore quantifies the extent to which the system can be described by such a smooth model. A neural network, for example, could then be used to create the actual model. We will describe the Gamma test in more detail in chapter 2.

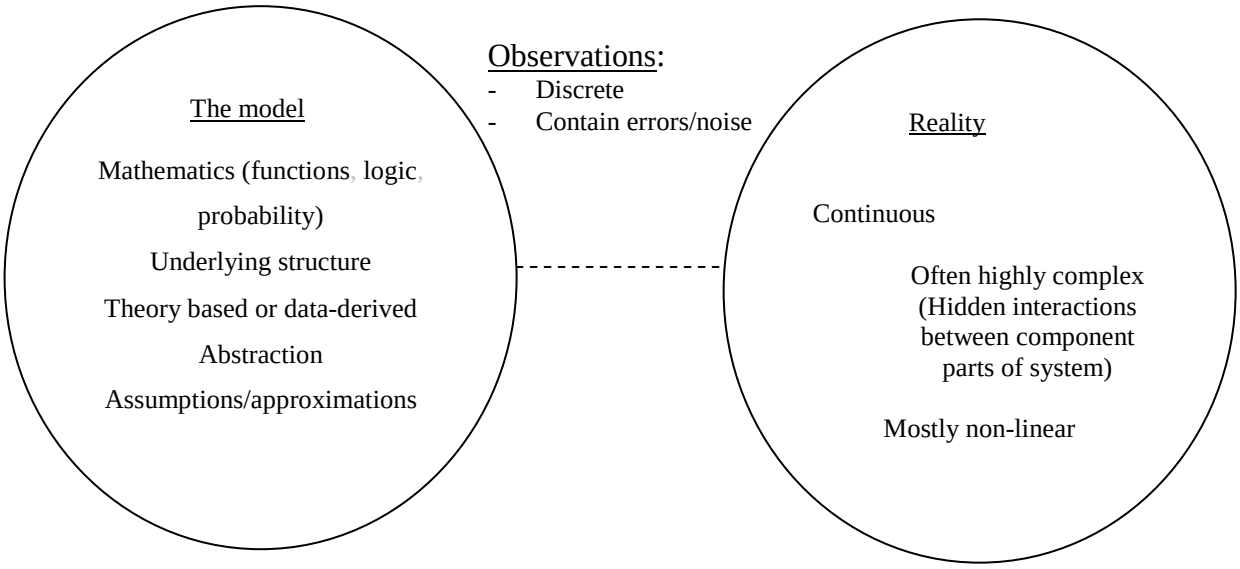

Figure 2.3: Illustrates that the model and reality exist independently of each other, but are related via observations. We can call these observations data. Data will always contain errors. However, these errors will vary about some norm, so conceivably observations will occasionally be 'correct'.

Let us consider the following statement:

Process + error → data

By the term 'process' we refer to the real system. When we observe the process to obtain data, errors are introduced into the data. The aim of data-driven modelling is to find the underlying process, by removing the errors from the data, thus creating an accurate model. The Gamma test can help do this by estimating the variance of that part of the output which cannot be accounted for by a smooth model, which corresponds to noise in the data set.

Models are representations of reality, and it is important to determine whether things that are observed in reality hold true in the model. Thus when a model has been constructed, from either first principles or observed data of the real world system, it needs to be validated using unseen data from the system. The ability of the model to generalize or

predict depends on how far away the unseen data is compared with previous data. This is equivalent to interpolating or extrapolating the model. This is important to consider as data-derived models are based on a system working under specific conditions, thus the model may not extrapolate well to predict outcomes of the system under different conditions. However a model derived from first principles may be able to compensate for altering conditions, assuming these conditions are parameterized within the model. A good model would also be able to quantify the extent to which its predictions are reliable. The Gamma test now offers heuristic confidence intervals [1].

A simple modelling example

We may attempt to model the relationship between two variables, say x and y, given data about them. From Figure 2.4 it would be wrong to assume the parametric form of this relation is linear. In fact the 'true' relation is quadratic and the deviation from it is due to noise (see Figure 2.5). We assume that we have enough data to ensure that this relation was not an outcome of chance. By using parametric methods we are often in danger of over-simplifying the relationship, especially as the true function could be one of an infinite number of smooth functions. We may also over-complicate the relationship, whereby we increase the order of the polynomial until we fit all the data points as shown in Figure 2.5.

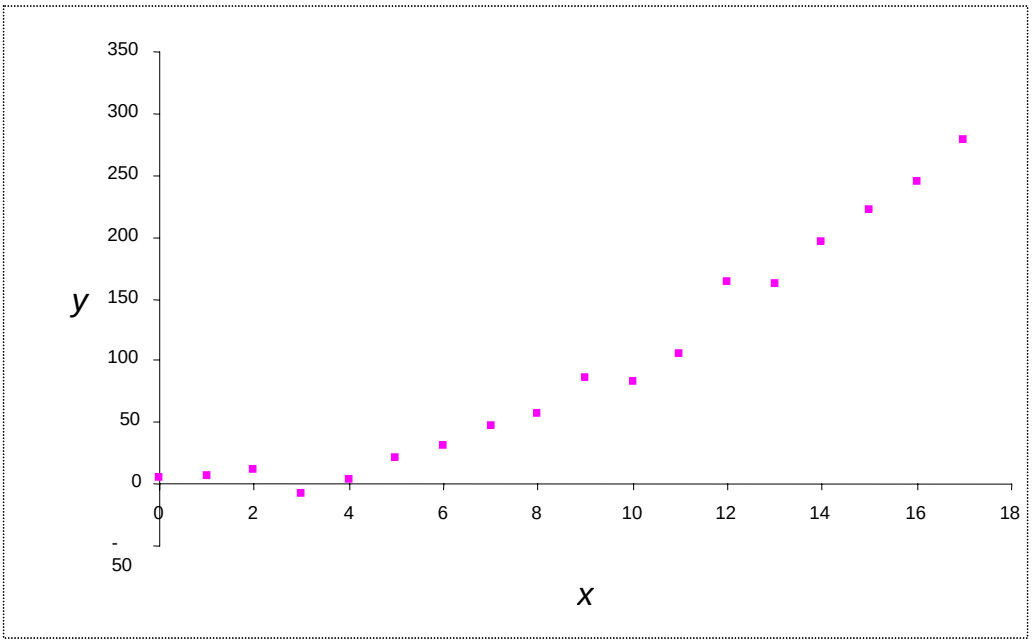

Figure 2.4: Some hypothetical raw data

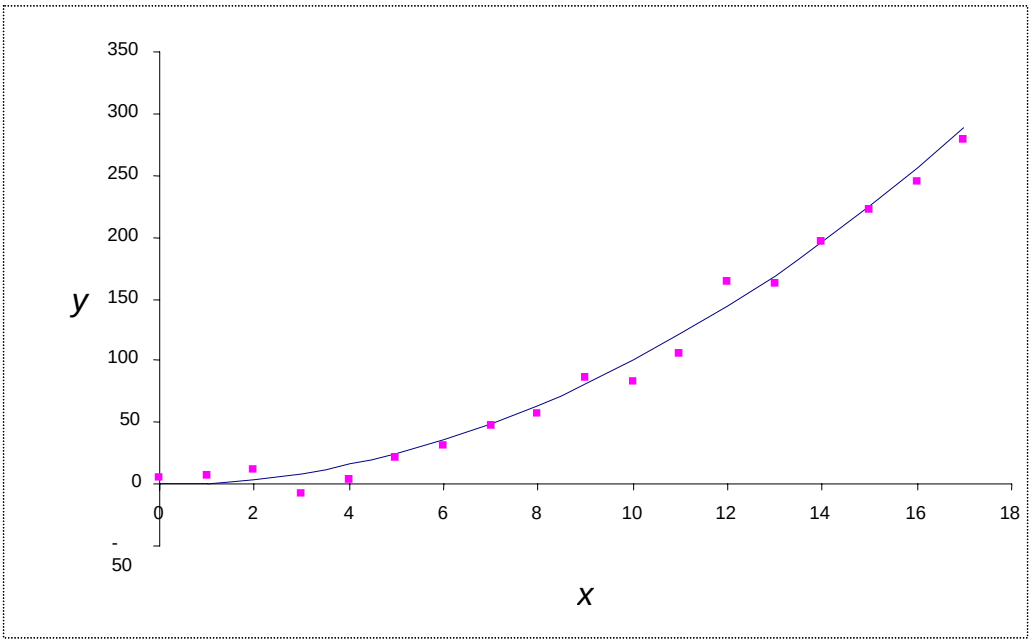

Figure 2.5: Ideal fit to data.

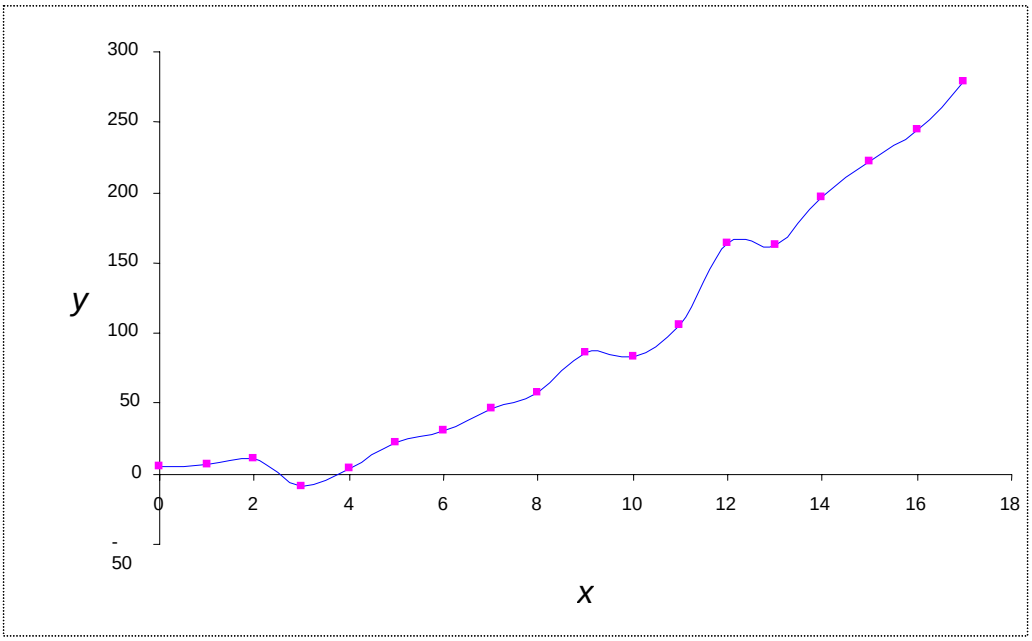

Figure 2.6: Over fitting data.

The question is: at what point does our model start to incorporate errors, and at what point is the true relationship found. This is the problem neural networks face: at what point should we stop training stop to avoid modelling the noise? The Gamma test is able to solve this by estimating the variance of the noise modulo the asymptotically best smooth function (as the number of data points increases to infinity) within the data, thus making model construction much more accurate.

Discovering the underlying relationship requires that we have sufficient data, which is not too noisy. Even if it appears to be random, we can deduce something about the system and its causative factors; for example, that not enough explanatory variables have been considered.

2.2 The Gamma test

2.2.1 Description

The Gamma test [26] is a non-linear analysis tool which allows quantification of the extent to which a smooth relationship exists within a numerical input/output data set. It is akin to a (least squares) linear regression, but can be applied to data that has *any* underlying smooth non-linear function. It is able to estimate the lowest attainable mean squared error by *any* smooth model.

Suppose we have a set of M input output observations of the form:

$$\{ (\boldsymbol{x}_i, y_i): 1 \leq i \leq M \}$$

where the inputs $\boldsymbol{x}_i \in \mathrm{R}^m$ are vectors confined to some closed bounded set $C \subset \mathrm{R}^m$, and the corresponding outputs $y \in \mathrm{R}$ are scalars.

The relationship between an input $\boldsymbol{x}$ and it's corresponding output y can be expressed as

$$y = f(\boldsymbol{x}) + r \qquad \text{EQ. 2.0}$$

where

f is some smooth unknown function representing the system, $f: C \rightarrow R,\ C \subset R^m$

r is a random variable having expectation zero representing noise.

The Gamma test allows the variance of the noise variable r (we will call this $Var(r)$) to be estimated, despite the fact that f is unknown. We do not assume anything about the parametric equations governing the system, only that the underlying function is smooth with bounded derivatives. However there are some conditions relating to the bounds of f, the distribution of noise r, and the set of $\boldsymbol{x}_i$ which are stated below.

The assumptions needed to be aware of (summarised from a proof of the Gamma test [24]):

1. The function f has bounded first and second-order partial derivatives over a convex closed bounded input space C, where $C \subset R^m$.
2. The random variable r has an expectation of zero, with respect to its probability density function
3. It is supposed the input points $\boldsymbol{x}_i$ are selected according to some sampling distribution having density function ϕ defined over a non-empty compact subset $C \subset R^m$. It is required that

(i) C is of bounded diameter $0 < c_1 < \infty$

(ii) $\phi(x) > 0$ for all $\boldsymbol{x} \in C$;

(iii) C contains no isolated points (however this is not critical for a continuous density function ϕ)

A crucial aspect to the Gamma test is its computational time; in view of the fact that we want to apply it to large data sets. In its less efficient form it will run in O(M^2) time. However, when implemented using an efficient near neighbour search algorithm such as a kd-tree [11], it will run with time complexity O($MlogM$).

The Gamma test can estimate *Var(r)* directly from the data, even though the underlying function f is unknown. This estimate is calculated by computing Equations EQ. 2.1 and EQ. 2.2.

$$\delta_M(k) = \frac{1}{M}\sum_{i=1}^{M} | x_{N[i,k]} - x_i |^2 \qquad \text{where} \quad 1 \le k \le p \qquad \text{EQ. 2.1}$$

|.| denotes Euclidean distance

N[i,k] denotes the index of the *k*th nearest neighbour to x_i.

p is the number of near neighbours, typically *p* = 10.

Thus $\delta_M(k)$ is the mean square distance to the *k*th nearest neighbour.

$$\gamma_M(k) = \frac{1}{2M}\sum_{i=1}^{M} | y_{N[i,k]} - y_i |^2 \qquad \text{EQ. 2.2}$$

$y_{N[i,k]}$ is the corresponding output of $x_{N[i,k]}$

|.| denotes Euclidean distance

By plotting the linear regression line of $\gamma_M(k)$ versus $\delta_M(k)$ for $1 \le k \le p$. The intercept of this regression estimates *Var(r)* and is known as the Gamma statistic denoted by Γ. It has been shown that $\Gamma \rightarrow Var(r)$ as $M \rightarrow \infty$, where the convergence is in probability [23,24]. The gradient of this regression can also be returned which gives an indication of the models complexity.

It can be useful to plot all near neighbour distances (squared) of EQ. 2.3 and EQ. 2.4: δ against its corresponding γ value for all pairs $1 \le i \le M$. This is known as the *scatter plot* and can give us more visual insight of the nature of the data set.

$$\delta = | x_{N[i,k]} - x_i |^2 \qquad \text{EQ. 2.3}$$

$$\gamma = \frac{1}{2} | y_{N[i,k]} - y_i |^2 \qquad \text{EQ. 2.4}$$

The Gamma test exploits the continuity of the assumed underlying smooth function *f*.

Despite not knowing what this function is, from the definition of continuity: "as an input $\boldsymbol{x}$ tends towards its neighbour $\boldsymbol{x}_0$, $|f(\boldsymbol{x}_0)-f(\boldsymbol{x})| \to 0$". However, if noise is present $f(\boldsymbol{x}_0) \neq y_0$, then $|y_0 - y|$ will be greater than zero (as corresponding $\boldsymbol{x}$ tends to $\boldsymbol{x}_0$) from which some information about the noise can be deduced.[1]

2.2.2 Pseudo code for the Gamma test

x(i) (input) is a real vector of dimension *m* and
y(i) (output) is a real scalar
where $1 \leq i \leq M$
(*M* = number of data points)

for i = 1 to *M* do
    for k = 1 to *p*
        Compute *N(i, k)* index where $\boldsymbol{x}_{N[i,k]}$ is the k th nearest neighbour to $\boldsymbol{x}(i)$.
        (this can be done in O(logM) time)
    endfor k
endfor *i*

for k = 1 to *p* do
    compute $\delta_M(k)$ as in EQ. 2.1
    compute $\gamma_M(k)$ as in EQ. 2.2
endfor k
perform least-squares linear regression on coordinates $\{\delta_M(k), \gamma_M(k)\}$
for $1 \leq k \leq$ p obtaining (say) $\gamma = A\delta + \Gamma$
return (Γ, A)

142

[1] For a comprehensive description of the Gamma test, see [2, 17, 24].

The first loop of finds the near neighbour indices runs in O(*MlogM*) using an efficient near neighbour data structure (*p* is small, typically 10), and the second loop of $\delta_M(k)$, $\gamma_M(k)$ calculations runs in O(*M*) time (as seen from EQ 2.1 and EQ2.2 for all *M*). The reader should refer to section 2.4 for more information about near neighbour search run times.

2.2.3 The M-test

The M-test is a way to gauge whether the Gamma statistic estimates *Var(r)* reliably. It is performed by computing the Gamma statistic for a given subset of the available data, where by at each successive computation of the Gamma statistic we increase *M* by some small step, until we have either used all the data or the statistic has converged sufficiently towards a fixed value. If the Gamma statistic converges, then we can say that all predictive input variables are present, and assuming that the underlying system is smooth, then the asymptotic value must be due only to *Var(r)* (refer to EQ. 2.0).

Heuristic confidence intervals can also be added to the M-test in order to further quantify the accuracy of the Gamma statistic for a given number of data points. These confidence intervals can be calculated using a heuristic method as described in [1], because data is usually limited.

2.2.4 Applications of the Gamma test

There are a huge number of scientific fields to which the Gamma test can be applied. The following list outlines some of these applications:

1. Used as a metric to stop adapting a model to fit the data. For example when training a neural network, it would be trained to the point where the mean squared

errors made by the neural network equal the variance of the noise given by the Gamma statistic. Thus separate test data would not be needed, and so *all* the data can be used to create the model

2. Used to find the best embedding dimension and lags times for time series analysis E.g. climate modelling, predicting the Earth's future temperature based on past temperatures, carbon dioxide levels, solar radiation at regular or non-regular intervals [33, 36]. With an efficient implementation of the Gamma test, relevant input factors (providing the dimension is not too high) could conceivably be found using a combination of Delta correlation [34] and a random or exhaustive embedding search approach.

3. If the Gamma statistic is small then the data can be modelled by a smooth function. Examining the M-test asymptote we can also tell whether we have sufficient data to form a non-linear model. The Gamma statistic will also indicate how good that model is likely to be.

When building a non-linear model we wish to know answers to questions such as:

- Which inputs are usefully predictive for determining the output?
- How many data points does one need to make a prediction?
- Is the output of the system determined by a smooth function of the inputs?
- Given an input vector, how accurately can we predict the output?

All the above can often be readily answered by the Gamma test if we are given sufficient data. Thus we can determine the 'quality' of the data.

The Gamma test will indicate if an input variable is usefully predictive, but not necessarily if that input is causative, in terms of its physical dynamics. Figure 2.7 illustrates this.

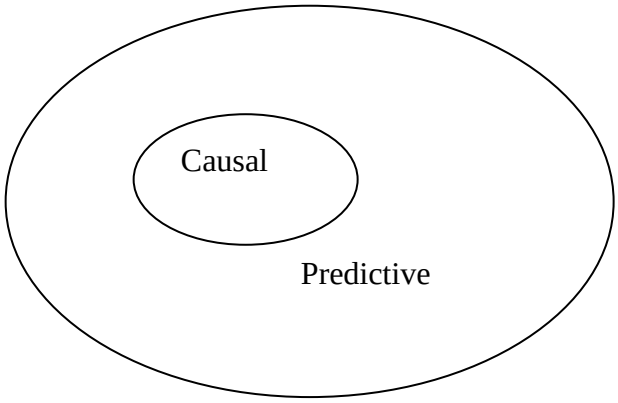

Figure 2.7: Illustrates that causal factors are a subset of predictive factors.

2.3 MATLAB and MEX

2.3.1 MEX description

MEX is a MATLAB tool that allows user defined functions written in C++ or Fortran to be called from MATLAB. It stands for MATLAB executable. It essentially uses the local compiler, such as Borland's C++ compiler, to compile a user defined subroutine, whilst simultaneously adding a wrapper that allows MATLAB to recognise the subroutine, and pass data to and from it. It consists of a library of functions and special data types that are used to act as an interface between MATLAB and the low level C++/Fortran subroutine. See Figure 2.8 for an overview.

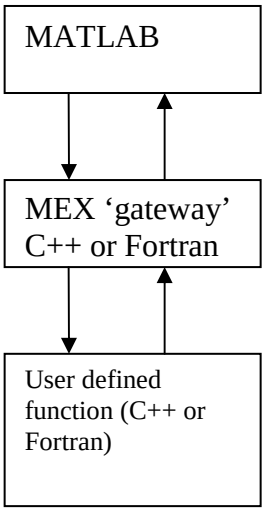

Figure 2.8: Overview of how MEX interfaces between MATLAB and a user defined function written in C++ or Fortran.

In order for MATLAB to gain access, manipulate and return values from a user defined C++ program, MEX functions are used. In particular the following MEX functions, data types and declarations are needed in the user defined C++ program:

1. Declare header file `mex.h`
2. The `mexFunction` (gateway to the user defined function)
3. `mxArray` (a data structure that stores incoming or outgoing data)
4. Functions that extract and return data to MATLAB

The first thing to do when constructing a MEX gateway in C++ is to declare the `mex.h` header file, in order to make use of the MEX library functions. Next `mexFunction()` is opened. `mexFunction()` allows MATLAB access to the subroutine via an array of pointers which point to `mxArrays`. `mxArrays` are special MEX data types which store everything related to the input and output data of the `mexFunction`. However it is not necessary that `mexFunction` take or return any arguments. `mxArrays` can be thought of as objects: they contain attributes and have a state depending on what data is read or written to them.

Syntax of `mexFunction`:

```
mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray *prhs[])
```

`nlhs`: Number of expected `mxArrays` (equivalent to number of outputs of the MATLAB function)

`plhs`: Array of pointers to expected outputs

`nrhs`: Number of inputs to the MATLAB function

`prhs`: Array of pointers to input data. (The input data is read-only and should not be altered by the rest of the mexFunction.)

MEX commands can then be implemented on the pointers to `mxArrays`, in order to gain access to or write back results to `mxArrays`. When the results are written back to the output `mxArrays`, these are automatically picked up by MATLAB. By using pointers, memory is shared between MATLAB and the low level subroutine, as implied by Figure 2.9.

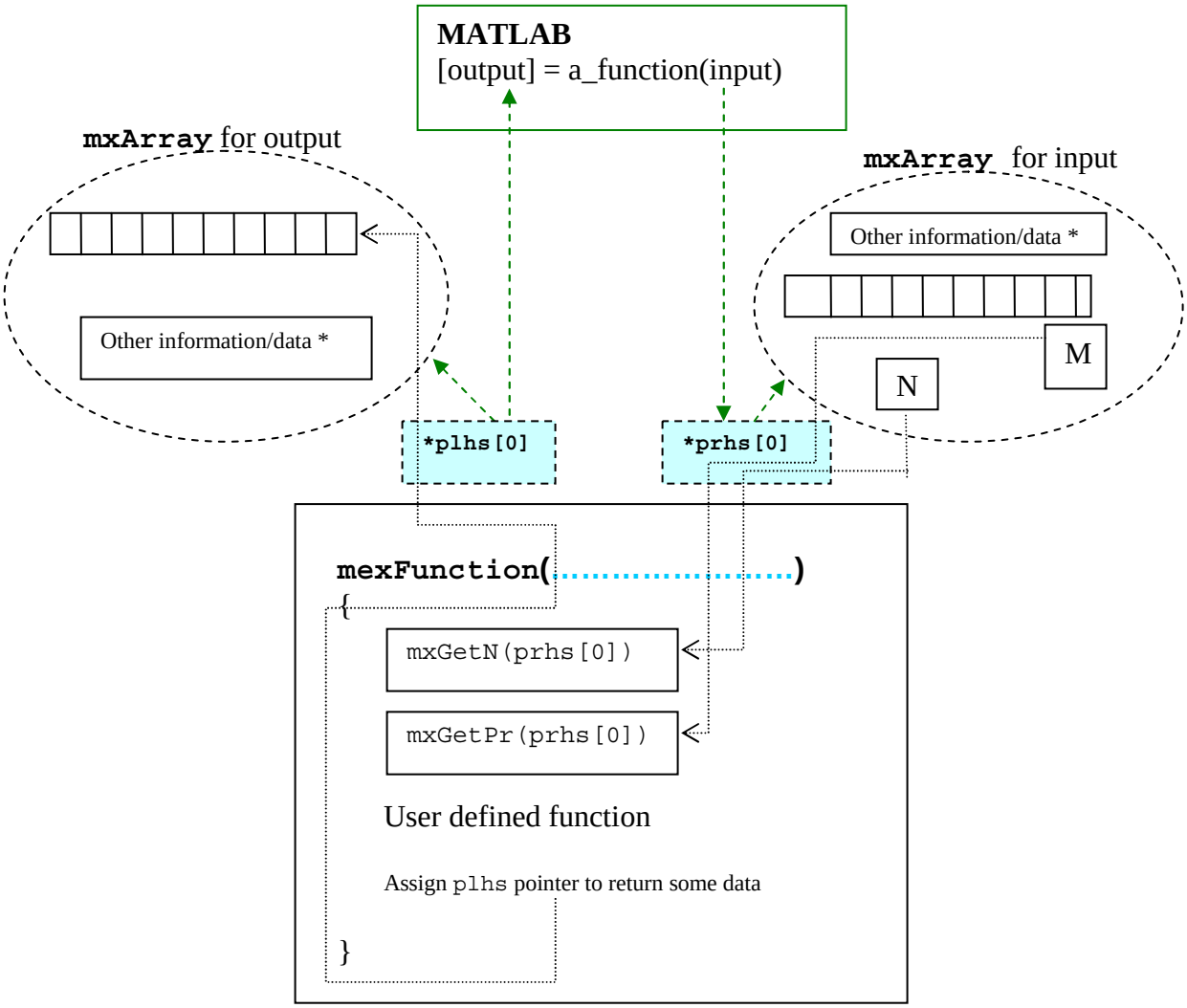

Figure 2.9: Shows the connection between MATLAB and low level C++ subroutine, via the use of pointers *prhs and *plhs, which point to mxArrays. Note also the use of MEX commands to extract or return data to the mxArrays. Note that this is always done via the pointers to mxArrays. Not all data from the mxArrays is shown to be transferred in this figure so as not to overcrowd the diagram.

MEX function API's prefixed with mx allow you to create, access, manipulate, and destroy mxArrays. Where as functions prefixed with mex perform operations back in the MATLAB environment

It is perhaps a little confusing to show scalars being passed through mxArrays objects. However, this is the notation that MATLAB uses. In fact everything from strings and

arrays to multidimensional matrices are passed through `mxArrays`. `mxArrays` also have other information associated with them, for example if they hold a matrix, then the number of rows and columns are specified. The command `mxGetN` and `mxGetM` obtains the number of columns and rows respectively, as illustrated in Figure 2.7. Using these parameters, it is then possible to unravel the `mxArray` into the format required.

Compiler set up for the MEX tool is described in the preliminary investigation Chapter 4, section 4.2.1.

2.3.1 Why MEX is needed

MEX is needed for various reasons. MATLAB scripts are not compiled; they execute line by line, i.e. they are interpreted, and are thus slow compared to pre-compiled routines that could do the task. MATLAB is particularly slow when doing large recursive operations. By writing this recursive operation in low level language, efficiency can be improved. By using a low level language such as C++, we also have more flexibility in creating specialized data structures, or can make use of a wide range of C++ libraries. Once a user defined function is interfaced with MATLAB using MEX, it can be incorporated into a whole range toolboxes that MATLAB offers [27], and powerful experiments can be carried out swiftly.

2.4 Near neighbour search algorithms

This section explains some basics about near neighbour search algorithms. As described earlier, the computationally intensive part of the Gamma test is finding the set of nearest neighbours for each point in the data set.

The near neighbour search problem can be defined as follows; given a set of M data points

where each data point consists of *m* attributes, we wish to find the closest set of points to a query point. Each attribute can be measured on some quantifiable scale, and there may be many metrics to determine an attributes size. For our purposes we will assume this metric space is *m*-dimensional real space, R^m .

When designing a near neighbour search algorithm we must be aware of efficiency with respect to the number of data points *M* and the dimension of each data point, *m*. There are various data structures which have been proposed for solving the near neighbour problem. [6,7,8,9,10]. Typically data is arranged in a tree-like structure, thus searching the structure scales as a logarithmic function of the total number of nodes within the tree. One such class of near neighbour search algorithms we will discuss is the kd-tree.

If a naive brute force approach is taken, finding near neighbours for all points in the data set would run in $O(M^2)$. This is impractical for most real life applications where data sets could have in excess of 100,000 points. The kd-tree as described by Bentley [11] use $O(M)$ space, and $O(logM)$ query time. However, these methods suffer a loss of performance as dimension '*m*' increases [28]. Indeed "the constant factors hidden in the asymptotic running time grow at least as fast as 2^m (depending on the metric)" [6]. Sproull [12] also observed from experiments that the running time of kd-trees does increase quite rapidly with dimension. Thus conceivably for high dimensions, a brute force search will perform similarly to the kd-tree. However, it has been shown by Arya and Mount [13] and Arya et al [14] that if near neighbour searches are calculated *approximately* it is possible to gain considerable improvements in running time. This approximation approach may be particularly useful when dealing with high-dimensional datasets, but would be highly dependent on the application: the degree of accuracy needed versus available computing power. Scaling with dimension is something to be aware of and was investigated in this project.

Another aspect to consider for near neighbour search algorithms is the metric used to measure the distance between points. Minkowski metrics [26] are typical distance measures; these include the well known Euclidean distance, Manhattan distance and Max

distance.

Computational time will vary depending on the metric chosen. This is due to varying efficiencies of low level hardware that carries out the basic calculations such as addition and multiplication. Likewise, the software implementation of these metric calculations is also an issue to take into consideration. The metric chosen does not affect big-O scaling as M tends to infinity, but is never-the-less a significant factor. Even if this factor is small, say 3, this would equate to an algorithm running for say three days compared with something that could run in one day. This factor may be even more significant for high dimensional data, due to exponential run time dependence. Ultimately absolute run times depend on the end users' application, but we should always aim to make the most efficient algorithm because users will always try to push the limits of their application.

In the next section we will describe one such near neighbour data structure known as the kd-tree.

2.4.1 The kd-tree

The kd-tree data structure represents the subdivision of k-dimensional space into k dimensional rectangular regions. By effectively compressing spatial information implicitly within its structure, data points which are spatially close to each other, will also be close on the kd-tree. Thus neighbouring data points can be found quickly when searching the tree.

Each node of the kd-tree represents a region of the space. All the data points are bounded by a box associated with the root node. Moving from root to leaf, the region associated with a node becomes smaller as it is recursively sub-divided by hyper planes at each branch point of the tree. As long as the data points that a node spatially encompasses is greater than a small number called the bucket size, the space will continue to be sub-divided. When the bucket size equals the number of data points in the current node, this

node is declared a leaf node. Thus the data points themselves are usually stored in the leaf nodes[2].

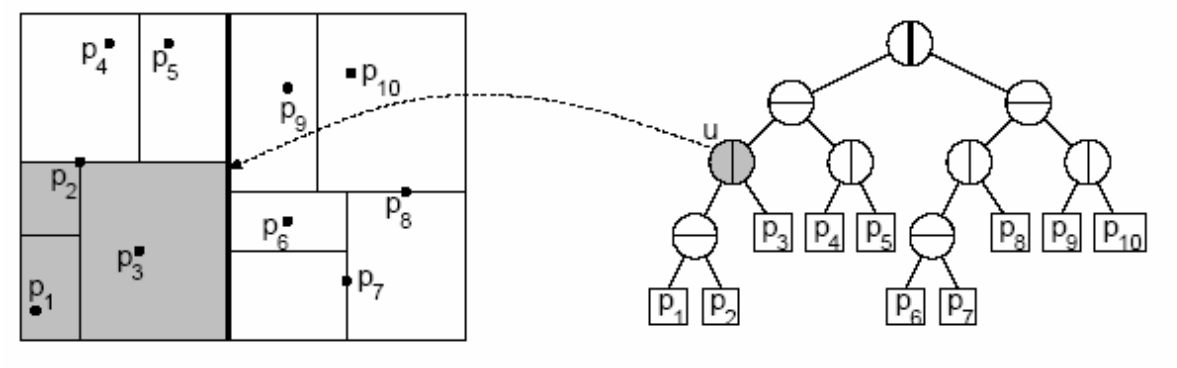

Figure 2.10: A kd-tree of bucket-size one and the corresponding spatial decomposition. Figure courtesy of [15].

There are many aspects to kd-tree design such as the splitting rule and the near neighbour search method. We will discuss the former briefly.

The splitting rule refers to how the space is partitioned when constructing of the kd-tree. It is very important to obtain a balanced tree structure thus subsequent searching will be on average faster. Usually the cutting planes are orthogonal to the axis. Ideally, partitions should split both the space and the set of points evenly. This ensures a *log(M)* height tree, for a set of *M* data points. However this may become very difficult if the data points are highly clustered causing partitions to become highly elongated. To overcome this, numerous other procedures can be carried out to re-balance the tree; one such method is to use a box decomposition tree [14].

142

[2] There are some variants to this, whereby data points can be stored in non-leaf nodes. This would require each splitting plane to pass through a data point itself. The influence of where the data is stored is beyond the scope of this project, but is thought not to be hugely significant in terms of computational complexity. Readers should refer to [16] for further information.

Chapter 3: Specification and development methodology

3.1 Requirements specification

From talking with the author's supervisor and others in the Evolutionary Computing Research Group, functional and non-functional requirements for this project could be defined. Once the specification was defined, some general approaches to design could be identified (see design chapter 5). A suitable development methodology could also be chosen to carry out the project. Not all requirements were defined at the beginning of the project due to some unknowns about how the software package should be set up for users. It was initially decided to carry out the known core requirements, and incrementally develop others.

3.1.1 Non-functional requirements (NFR)

The software system developed should be (in no particular order):

NFR1: Reliable: the system should not crash or leak memory.
NFR2: Maintainable: this is necessary to allow the system to be updated in the future
NFR3: User friendly: aimed at users with basic MATLAB knowledge.
NFR4: Able to conform to legal requirements

3.1.2 Functional requirements (FR)

This project and the software system developed should meet the following specific requirements (roughly in order of priority, note that optional requirements imply optional in terms of if a *user* wants to invoke that option):

FR1: To return the Gamma statistic to an accuracy of 15 digits
FR2: To optionally return the slope of the regression line to an accuracy of 15 digits
FR3: Computational complexity should scale as O(*MlogM*) where *M* is the number of data points
FR4: The software package should be portable and be able to run on both Windows and Linux platforms
FR5: To optionally set the number of near neighbours used for regression. Default set to 10.
FR6: To optionally set an error bound which allows approximate near neighbour calculations. Default set to zero. This error bound will reduce run time at the cost of approximations in near neighbour distance calculations.
FR7: To optionally set the metric with which distances are calculated. These include the Max, Manhattan and Euclidean metrics. Default should be set to Euclidean.
FR8: The input data is an '*M* by *m*' matrix, where *M* is the number of data points and *m* is the dimension of the data.
FR9: The output is an '*M* by 1' matrix. In other words the corresponding output for each input is a scalar.
FR10: A demo script, Readme file and user manual should be included to help end users understand how the package works.
FR11: To optionally plot the linear regression line of EQ 2.1 and EQ 2.2
FR12: To optionally plot the full scatter diagram of EQ. 2.3 and EQ 2.4 with regression line
FR13: To run tests to view trends in computational performance of the algorithm.

3.2 Implications of requirements to design and implementation

FR3 necessitates the need for a fast near neighbour search algorithm, given that this is the computational bottle neck within the whole system, it is implicit that this near neighbour search algorithm be implemented in a low level language such as C++. Building complex data structures using MATLAB scripts would be prohibitively expensive in terms of computation overheads.

In order to carry out these functional requirements, the near neighbour algorithm needed to be *validated* against a brute force near neighbour search, and the overall system needed to be validated against an artificial dataset. This dataset has a known Gamma statistic value derived from previous implementations of the Gamma test. Finally extensive testing needed to be carried out to ensure all error conditions were caught, thus making the system reliable. See chapter 7 for validation.

Specification FR13 regarding performance tests is described more clearly in Chapter 8. It required a fairly controlled environment in which to carry out the tests, so that computational time was not affected by other factors such as the operating system background processes.

3.3 Development methodology

Given that the core specifications (FR1, FR2, FR3, FR4) were fairly well understood from the start of the project, an incremental methodology was chosen. Other functional requirements were initially undefined but became apparent as the software developed. Specifications were prioritized, and depending on time constraints, it was aimed to implement as many of the requirements as possible in order of priority. The order of priority is to some extent shown in functional requirements list of section 3.1.2. FR1, FR2, FR3, FR4 were the most essential. Non-functional requirements (NFR1, NFR2, NFR3)

were always kept in mind so as to build a high quality software package. It was known that other features would become apparent over time thus a modular approach to design was taken in order to implement features as and when they arose.

For an in depth description of various development methodologies the reader should refer to one of many books on software engineering, for example Sommerville [25]. For this project a waterfall model could not be used because new specifications gradually became apparent as the project progressed. To some extent an evolutionary approach was taken in order to allow some flexibility in development; and *full* system tests could be carried out at the end of development.

To meet specification FR4 (portability across platforms, Windows and Linux), C++ code for the fast near neighbour search was developed on both platforms, and *validated*. After this, MATLAB script development is considered portable (providing the same version of MATLAB is used on each platform), thus would only require development on one platform. Final system tests were conducted on both platforms.

A preliminary investigation was also required before a design could be fully created; this preliminary investigation is described in Chapter 4.

Chapter 4: Preliminary investigation

This preliminary investigation showed a viable way in which to interface MATLAB with a near neighbour search subroutine, from which a design could be made in Chapter 5, and subsequently implemented in full as described in Chapter 6.

The specification defined in Chapter 3 can be illustrated as a diagram as shown in Figure 4.1. However from this diagram is it not clear how the data should be loaded into the near neighbour search subroutine: either directly or via MATLAB. It is also unclear how the other metrics will be implemented. There is also an issue regarding the use of MEX and the format in which data is passed to a user defined function. If MEX was not used, it would be possible to transfer data by reading and writing to files, however MEX offers a more efficient way in which to share data between MATLAB and a user defined C++ subroutine, thus was investigated further in this chapter.

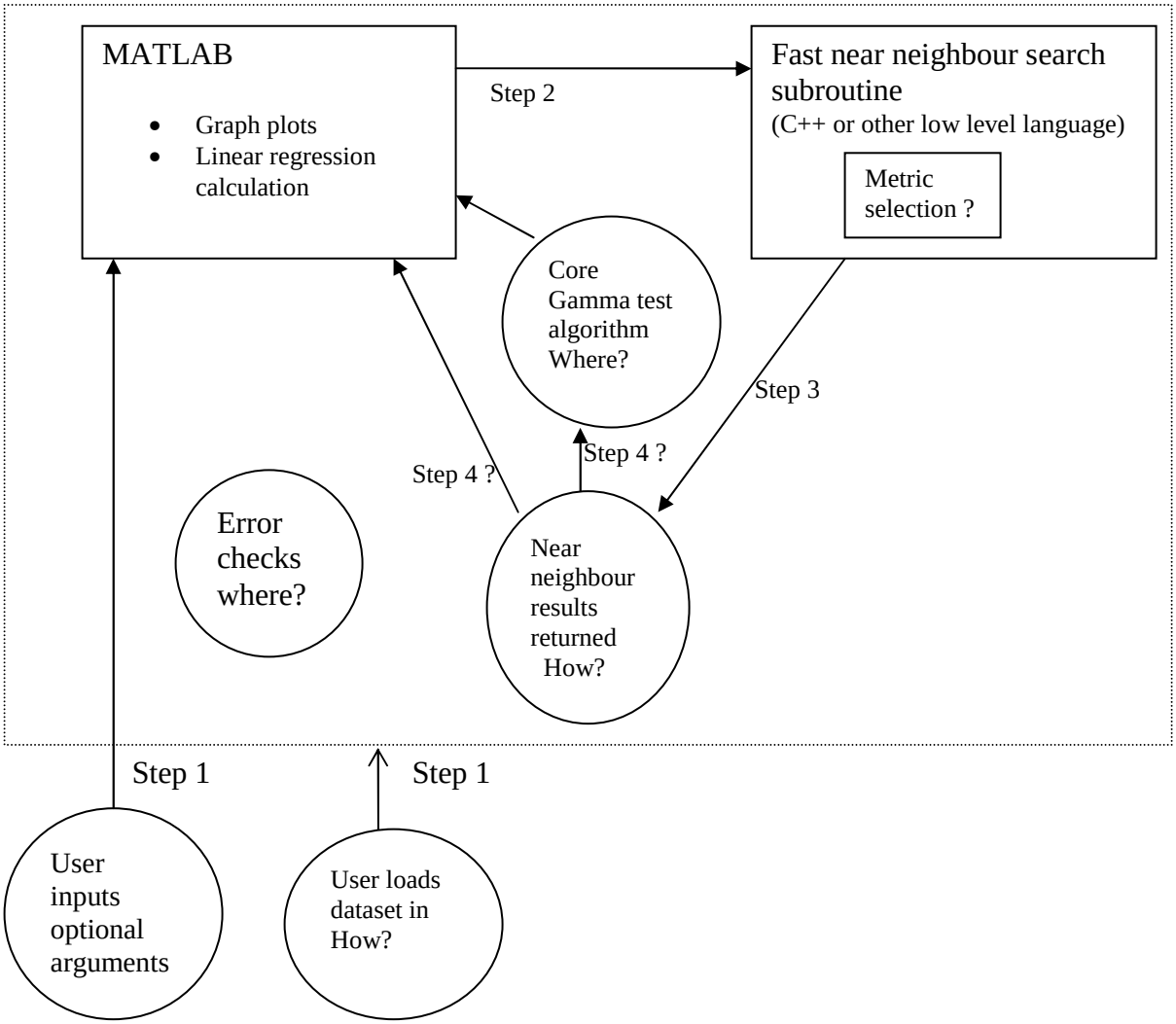

Figure 4.1: Illustrates the specification (FR1, FR2, FR3, FR7 –see section 3.1.2) but does not show *how* it will be carried out; thus necessitating the need for a preliminary investigation into MEX and near neighbour search algorithms.

4.1 An approximate near neighbour (ANN) C++ library

An internet search was undertaken to try to find existing near neighbour search algorithm implementations in C++. Initially some were found but did not offer sufficient flexibility, and building one from scratch would be too time consuming given the duration of this project. Fortunately, a C++ library of functions able to build and search near neighbour

data structures was later acquired. It was originally written by Mount et al. and can be found online at [3], it is called the ANN (approximate near neighbour) library. A slightly older version (but of same functionality) of this library had been used by Dr. Samuel Kemp from University of Glamorgan for an R-library [22] implementation of the Gamma test. The version used in this project was ANN version 1.1 (release 05/03/05). This library comes with options for building various data structures depending on the distribution of the data set. It also has different methods for searching and supports different Minkowski metrics, all of which will be discussed in more detail. This library fulfils all functional requirements set out in Chapter 3. The purpose of this chapter was to investigate how data is input and returned when building and searching a kd-tree of the ANN library. From this, we could judge if MATLAB's MEX tool was able to interface with the kd-tree, and a subsequent design could be made.

The library also offers approximate near neighbours to be calculated by the setting of an error bound e. For $e > 0$, the ith nearest neighbour detected may exceed the true distance to the ith *real* nearest neighbour (from the query point) by a factor of $(1+e)$. This approximation has been shown to significantly improve running times [14].

4.1.1 Creating the ANN library using makefiles

The ANN library, like any C library, is a collection of functions. Before the ANN functions could be used, the first step was to *create* this library from the original source code. This was achieved by using a Makefile [19] written by the author's supervisor, and adapted slightly for this ANN library (see Appendix A.5). Makefiles provide a convenient way to maintain software systems, especially when dealing with large projects consisting of many source files. Separate Makefiles were created for both platforms: Linux and Windows. C++ compilers used were:

- g++ : a GNU project C++ compiler [20] (on Linux)
- Borland C++ compiler 5.5 available free from [21] (on Windows)

.

Associated with the ANN library are various header files. In particular, `ANN.h` contains declarations for special data types, which can be altered to create *separate* libraries that support different metrics. In order for a program to use the ANN library, the header file `ANN.h` needs to be declared at the top of the source file, and the library must be linked to the program; this requires various compiler flags. In the next section we describe the compilation of a sample program provided with the ANN package.

4.1.2 A sample program

Once the library had been created and copied to its own separate folder location, it was ready to be used. A sample program was provided with the ANN package and readers should refer to [3] for the source code. Readers should refer to Figure C.1 (Appendix C) to see a visualisation of the decomposed 2 dimensional space generated by this ANN library.

Steps taken to compile and run `ann_sample.cpp` program (on Linux):

1. On the Linux terminal the following was typed:

```
g++ ann_sample.cpp -o ann_sample -I/home/scmss3/include
-L/home/scmss3/lib –lann
```

Note that the path of the header files and library are stated, as is the name of the library.

2. To run the program, the following was typed:

```
ann_sample -df data.pts -qf query.pts
```

This program reads in some multidimensional data and query points from the files `data.pts` and `query.pts` using a C++ read file function. The format of the data in these files is specific to this program. (Refer to the ANN manual for more information

[15]). The near neighbour indexes and distances are then returned to the command line.

The above steps were also carried out successfully on the Windows platform. This confirmed that the ANN library was a feasible approach for use in this project, thus justifying further investigation as described in the Chapter 5 on design.

4.2 MEX

The main purpose of this section was to understand the exact format in which MATLAB transfers data to and from a user defined function in C++. Therefore, a simple C++ program was written which transposed a 2 dimensional matrix. This program is called `mytranspose.c` and can be found in Appendix A.1

We first describe configuring MATLAB to use the MEX tool.

4.2.1 Configuring MATLAB for use with MEX

In order to use the MEX tool, MATLAB must be set up to use the compiler of your choice for generating the external MEX subroutines. '`mex – setup`' was typed at the MATLAB command line, C++ compilers were chosen: GNU g++ on Linux, and Borland's C++ compiler version 5.5 on Windows. Configuration settings were generated automatically in files called `mexopts`.bat and `mexopts.sh` for Windows and Linux platforms respectively. An alteration was also made to `mexopts.sh` in order for the correct compiler (g++) to be used. For completeness these configuration files have been put in Appendix A.6.

Note that since MATLAB 7.1 (R14SP3), file extension for 32-bit Windows MEX-files was changed from "`.dll`" to "`.mexw32`". For more information see MATLAB 7.1 Release Notes: "New File Extension for MEX-Files on Windows".

4.2.2 The format of data transfer

In this section we aimed to understand how MEX transfers data from MATLAB to a simple user defined C++ function, in this case a matrix transpose function. Also by transposing the matrix we could be absolutely sure about how a two dimensional matrix from MATLAB was passed to a one dimensional MEX data type. This was crucial to understand so that data passed via the MEX interface would not be jumbled in the wrong order. The C++ code was named `mytranspose.c`, listed in Appendix A.1. It was MEX compiled (from within MATLAB) using the command:

```
mex mytranspose.c
```

This created `mytranspose.mexw32` (for Windows) and `mytranspose.mexglx` for Linux. From running this program and transposing an arbitrary size matrix it was deduced that the MEX data type, known as an `mxArray`, stores data linearly as shown in figure 4.2. Other MEX functions are used to obtain its dimensions so that the data can be extracted as if it were a 2 dimensional array.

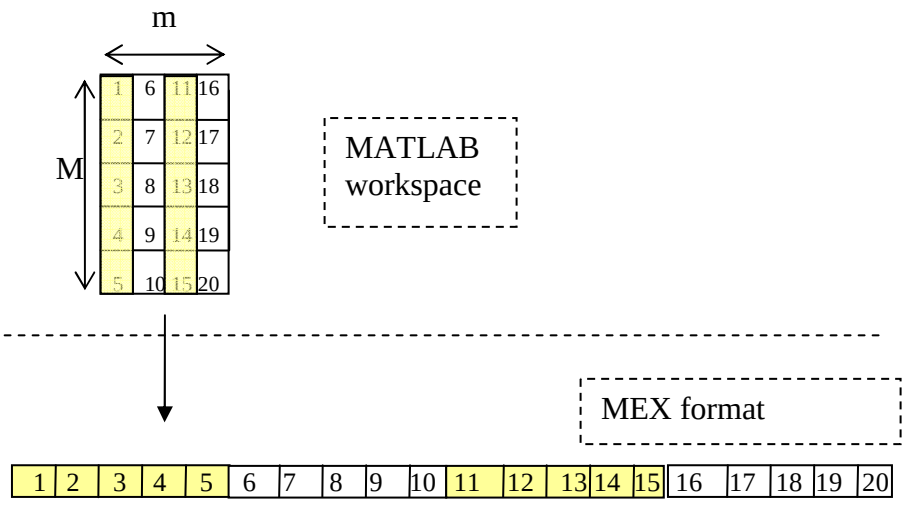

Figure 4.2: Illustrates the arrangement of data for a 2 dimensional matrix in MATLAB which corresponds to a 1 dimensional array of MEX data type.

4.3 Conclusion

From this preliminary investigation, it was concluded that a near neighbour search algorithm could be successfully compiled (for both Linux and Windows platforms) using the ANN C++ library; and that MEX was a viable way to link MATLAB with this near neighbour search algorithm. Thus a design was created using both of these tools, as described in the next chapter.

Chapter 5. Design

5.1 Approach to meet non-functional requirements

In order to meet the non-functional requirements laid out in Chapter 3, we identified some general approaches to design. To make the system reliable (NFR1 specification), error checks were decided to be performed in MATLAB, thus ensuring everything is in order before even calling the C++ MEX subroutine. To make the system maintainable, a modular approach was taken, where by functional parts of the system were separated into different executables and scripts. This will also allow users to experiment with different components of the system should they wish to do so. For example the fast near neighbour search subroutine alone has a wide range of applications [18]. Use of the ANN library was acknowledged in the source code to adhere to copyright laws set out by these authors.

5.2 High level design

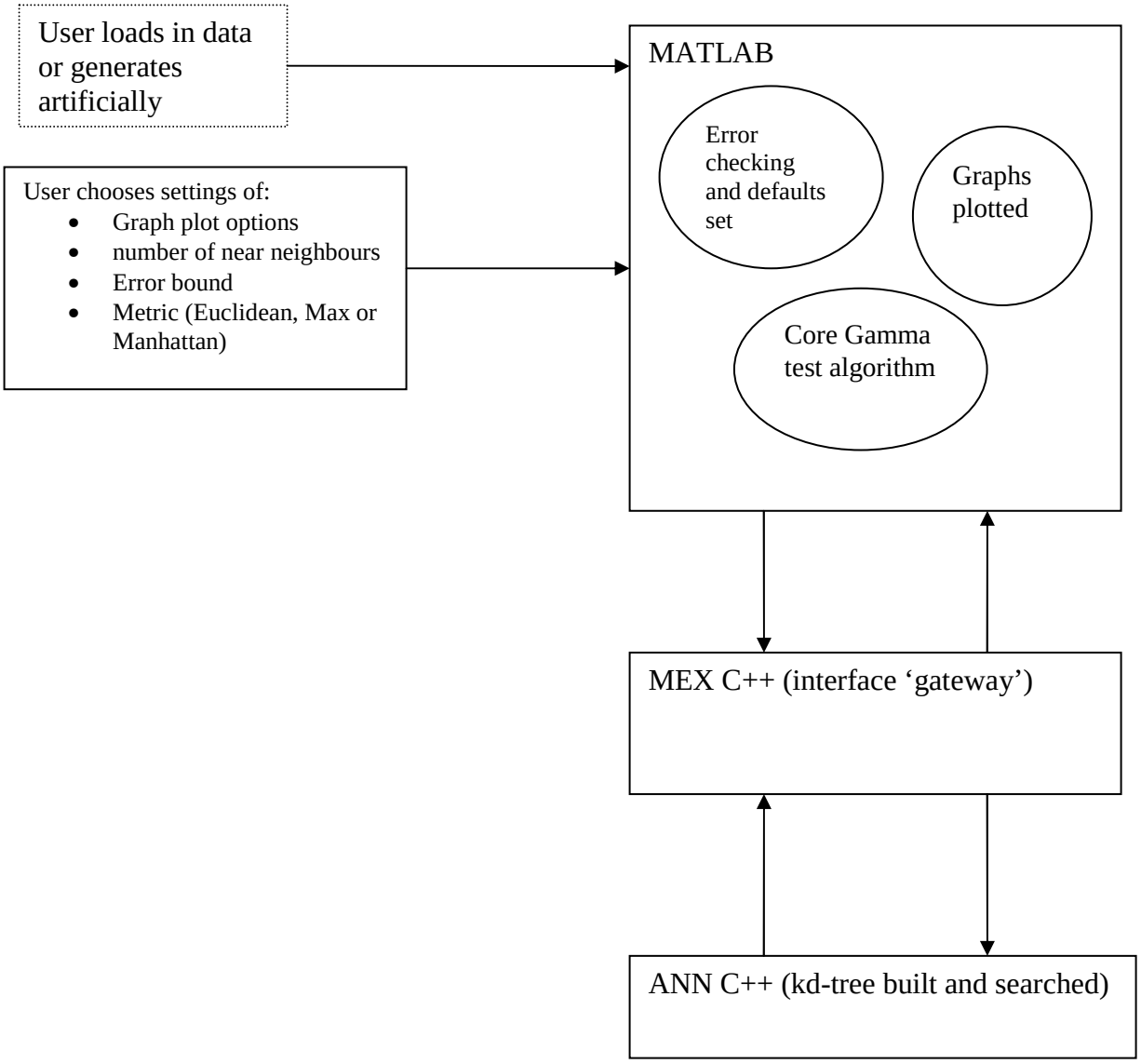

Figure 5.1: High level design that includes main functional components. Note that metrics are not explicitly represented here and the reader should refer to Figure 5.2 to understand how different metrics were implemented. The M-test is also not shown here.

It was decided to use the kd-tree data structure of the ANN library. The alternative would be a box decomposition tree (bd-tree) [14]. Both run in O(*MlogM*) time, however bd-trees

are able to deal with clustered datasets better, by the use of a shrinking rule which effectively focuses decomposition of space around the clusters. It does however require more computational overheads, so there is a trade off. Given the wide range of applications (and hence varying data distributions) that the Gamma test may be applied to, we can never be sure which data structure would be fastest. Thus it was decided to use the kd-tree. In most situations it is thought that the distribution of data will be fairly even and not highly clustered, though this will not always be the case. Ultimately either data structure would meet the O(*MlogM*) requirement specified in FR3.

The standard kd-tree search method was chosen which is an adaptation of the search algorithm described by Bentley, Friedman and Finkel [11], but for approximate nearest neighbours. Another search method was available in the ANN library, called *priority search*, whereby cells are visited with increasing distance from the query point, consequently should converge quicker to the true nearest neighbour. This requires higher computational overheads; but has been shown to be slightly faster (than standard searching) when the error bound is high [15].

The *core* Gamma test algorithm was designed in MATLAB because this is not the most computationally intensive part overall, and runs in O(*M*) time, where *M* is the number of data points; compared with O(*MlogM*) for building and searching the C++ kd-tree, thus ensuring that FR3 was met. Note here the distinction between the 'Gamma test', and the '*core* Gamma test'; the latter refers to the heart of the Gamma test algorithm which runs in O(*M*) time when given *pre-computed* near neighbour lists. The former 'Gamma test' refers to the system as a whole, which implicitly includes computation of near neighbours, thus runs as O(*MlogM*) as a whole.

Graph plots are readily done in MATLAB, which has a range of built in tools to plot all kinds of graphs. The graphs plotted will be 'the regression plot' as described in the Background chapter section 2.2, and also an option to include all near neighbour distances (not just averaged), which we will call 'the scatter plot' (EQ. 2.3 and EQ.2.4) . The linear regression line can be easily calculated using MATLAB's built in functions. This will

fulfil specifications: FR1, FR2, FR11, FR12.

It is the users responsibility to load their data sets into the MATLAB workspace. The format in which this is done will be described in the low level design, section 5.3

Various settings are required as stated in FR5, FR6, FR7, FR11, FR12. These settings will be optionally chosen by the user as input arguments to the MATLAB function. Defaults will be set as defined in the requirements specification.

The example file included with the ANN source code (discussed in section 4.1.2) could have been modified to our purposes, whereby near neighbour index results could have been written to a file, and subsequently read by MATLAB. However the MEX utility exists, which allows data to be passed *directly* from MATLAB to a user defined function written in C++, and results passed back to MATLAB. This allows memory to be shared, by the use of C++ pointers, this avoiding some inefficiency in file writing and reading. This was described in the Background Chapter section 2.3.1.

Data could have been loaded in at the C++ level, but MATLAB provides a wide selection of easy to use data loading functions. It also means that the data is readily available in the MATLAB workspace should the user wish to visually display or manipulate the data using standard MATLAB commands.

In part to meet NFR3 (user friendly) specification, a usage description will be designed into the code so that when the function is called with no arguments; a friendly usage message is printed on the command line. A demo script, readme file and basic user manual will also be provided to get users started with the package. (see Appendix A.2 and Appendix B)

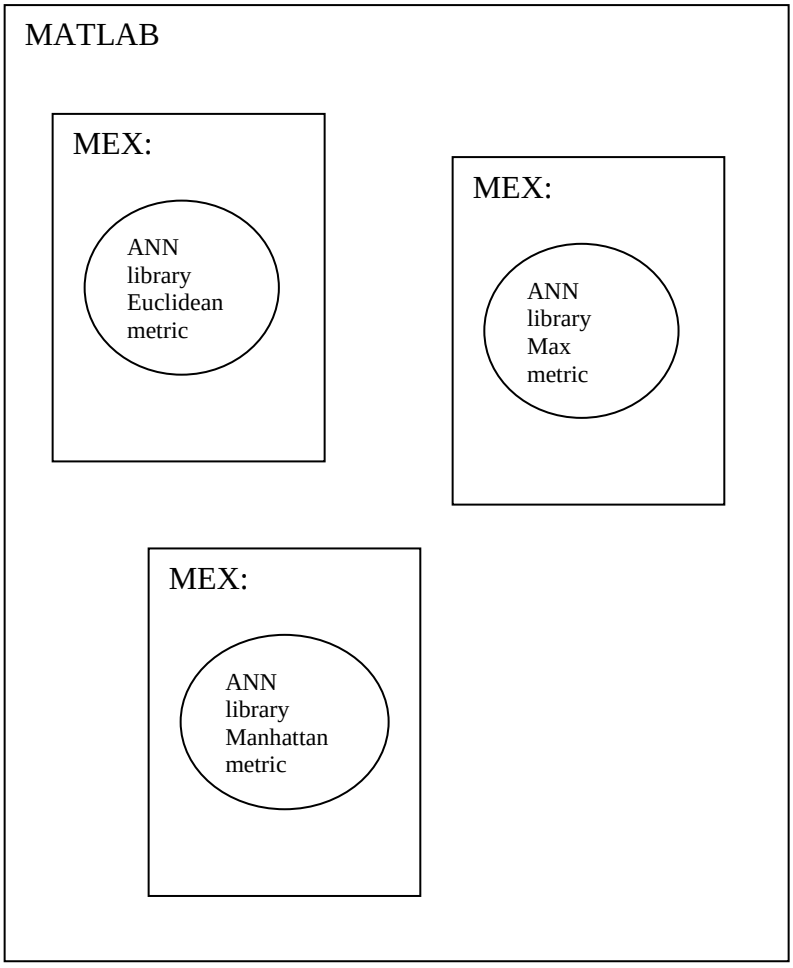

Figure 5.2: illustrates the three MEX functions required, each compiled using a separate library (Each ANN library created is specific to a metric). A script written in MATLAB selects the appropriate MEX function depending on which metric the user selects. See Figure 5.4.

3.3 Low level design

In this section we describe in more detail the flow of data between MATLAB, MEX and the kd-tree. Figure 5.3 summarises this.

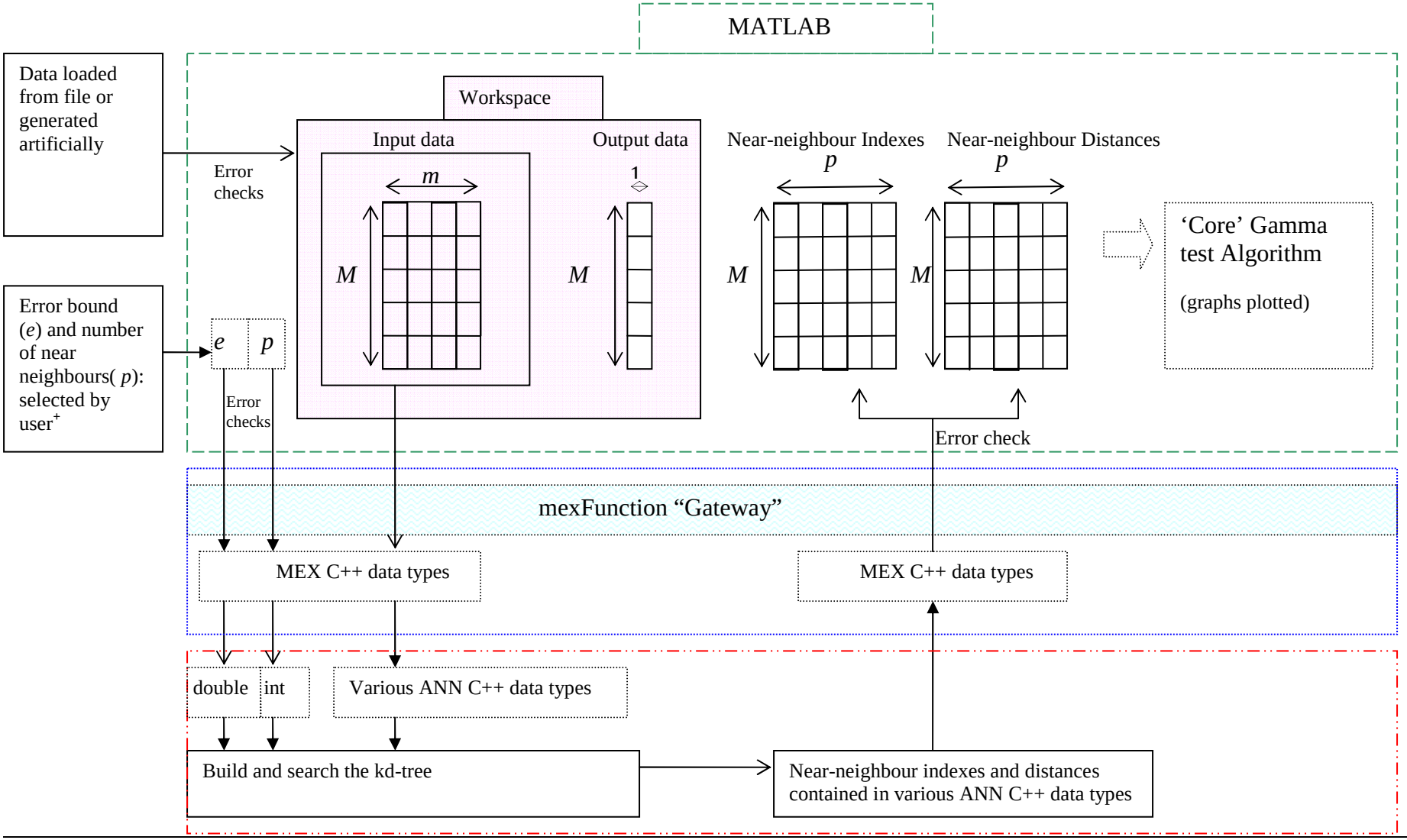

Figure 5.3: Low level design: shows main flow of data, and how it changes form from MATLAB to ANN via the MEX data types. Once near neighbour indices and distances are returned to MATLAB, the core Gamma test algorithm executes. Error checks are carried out in various places on the MATLAB script level.

Notes referring to figure 5.3

[+] Selection to use other Minkowski metrics is not shown on this diagram. The argument for the Minkowski metric is dealt with by a metric selection script, see section 5.4. This script simply calls the appropriate MEX function depending on which metric is chosen. If no metric is specified by the user, Euclidean is set as the default. If the metric is not recognised, error checking will catch this.

[+] Also note that graph selection options are not shown on this diagram. The graph selection argument is simply a string that is passed to the core Gamma test script.

Components of design

- The data set loaded into the MATLAB workspace will have the following format:
  - Input data will be an '*M* by *m*' matrix, where *M* is the number of data points (rows) and '*m*' is the dimension of the data (columns).
  - Output data will be an '*M* by one' matrix. In other words the output is one dimensional (one column).

- Similarly near neighbour distances and indexes will be returned to MATLAB in the form of '*M* by *p*' matrices, where *p* is the number of near neighbours specified. However, these matrices will not be available in the MATLAB workspace unless the fast near neighbour module is used separately.

- Error checks are all done on the MATLAB level. This keeps design modular. There are three main areas where error checking is done:
  - on the input output data pairs;
  - on the input arguments: graph selection, near neighbours, error bound and metric
  - on the returned near neighbour matrices

If an error condition is encountered, the program will return a useful error message and exit safely. This will be described in more detail in section 6.3.

5.4 Metric selection

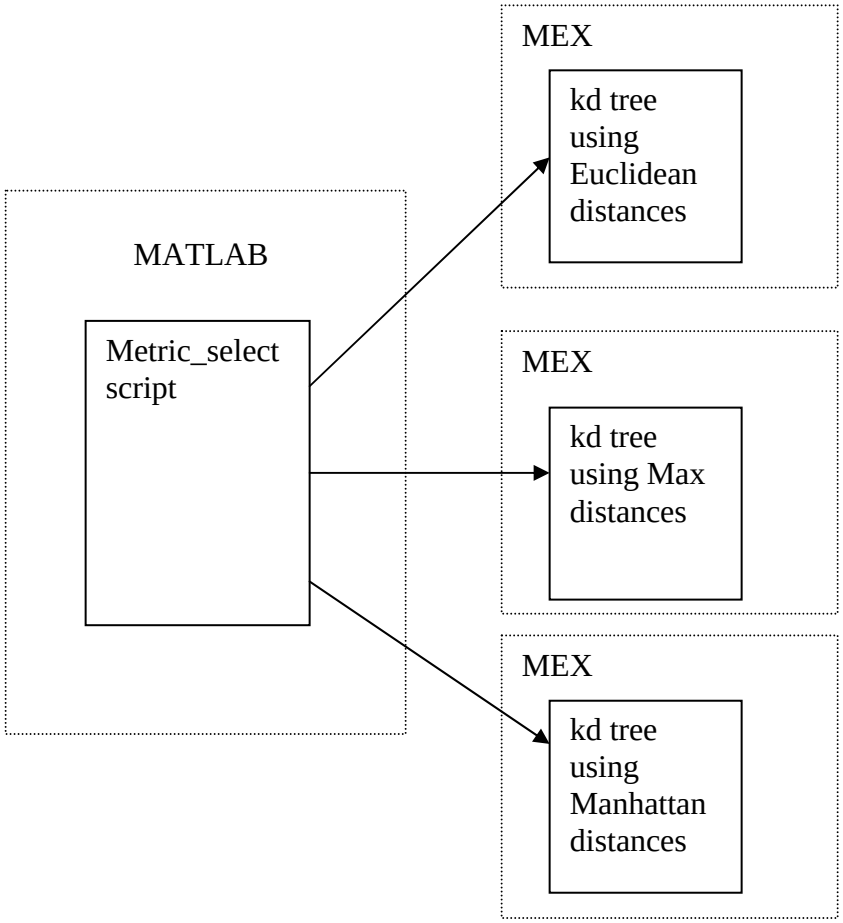

Figure 5.4: illustrates the MATLAB script used to control metric selection. Note that each metric is a separate MEX file.

Chapter 6. Implementation of design

Implementation can be broken down into the following sections:
(Modularity of the system should become apparent through this)

- Interfacing between MATLAB, MEX and ANN data types.
- Core Gamma test (and figure plots).
- Metrics:
    - modifications to `ANN.h` header file to create different ANN libraries
    - the metric selection script
- Error checking, default settings and usage messages.
- The M-test with confidence intervals.

6.1 Interface between MATLAB, MEX and ANN

Figure 6.1 illustrates the main flow and shape of data for calculating the near neighbour matrices in C++ using the ANN library functions. Other options of graphs plots and metric selection are not shown on this diagram: these arguments are dealt with by MATLAB scripts `fastnn.m` and `gammatest.m`

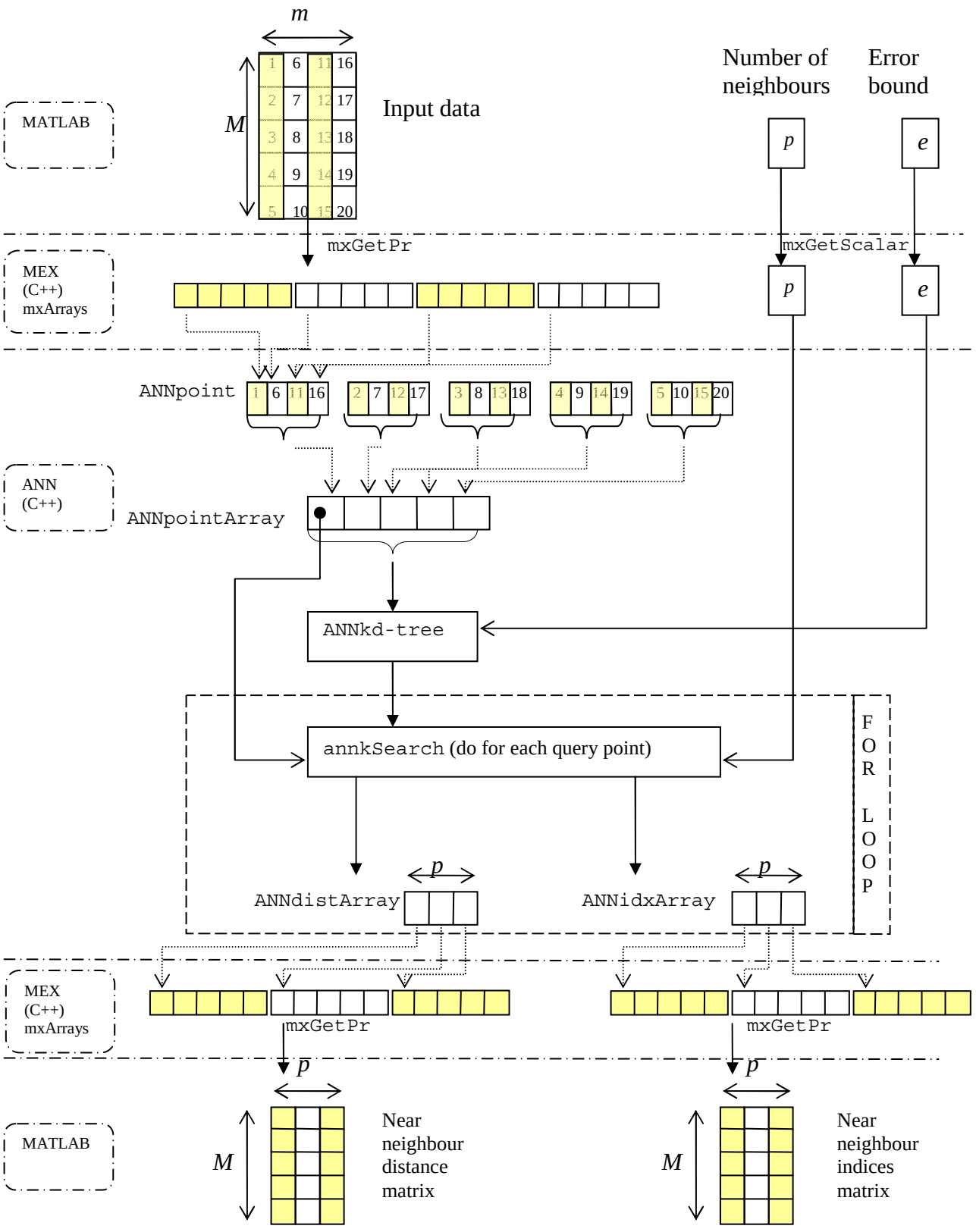

Figure 6.1: shows how data is passed between MATLAB and kd-tree via `mxArrays`, to obtain the near neighbour matrices.

In figure 6.1, shading and numbering of the arrays and matrices emphasises the actual format of the `mxArray`: column by column going across the array. This ordering was explained in section 4.2.2.

Once the index and distance matrices have been returned to MATLAB, they can be sent to the core Gamma test algorithm.

Main functional parts of C++ code which reflect Figure 6.1

This section explains some core functionality in using ANN library functions and passing data to and from MATLAB. For complete code list (including comments), see appendix A.2. The reader should also refer back to section 2.3 and Chapter 4 for additional information about this code.

Step 1: header files

It was necessary to include the following header files in order to make use of ANN and MEX library data types and functions.

```
#include "ANN.h"
#include "mex.h"
```

Step 2: the gateway

The 'gateway' MEX function was initialised (As explained in section 2.3.1)

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

Step 3: declaring variables

Pointers to MEX data types, known as `mxArrays` were declared. Standard C++ data variables (`int` and `double`) were declared to the hold data of the `mxArrays`. ANN data types were also declared.

Step 4: extracting data from MATLAB using MEX functions

Parameters describing the form of the MATLAB matrix were then extracted into the standard C++ variables using MEX operations. Similarly the error bound and number of neighbours were extracted. The dimensions of the matrix `M_rows` and `N_cols` could then be used to transfer data to the ANN data types `ANNpoint` and `ANNpointArray`, as shown in the following loop. A pointer to the input matrix was set: called `input0` (which pointed to the `mxArray` input)

```
// create ANN pt_array object: read input0
//(mxArray format--> ANN format)

for(int i=0;i<M_rows;i++) {
     ANNpoint pt = new ANNcoord[N_cols];
     for(int j=0;j<N_cols;j++) {
            pt[j] = input0[i+(M_rows*j)];  //put data
                                           //into ANNpoint
     }
     pt_array[i] = pt;
}
```

Step 5: build the kd-tree

The following code built the kd-tree using the `ANNpointArray`, which was created from step 4. The other arguments necessary were the number of data points *M* and the dimension of the data *m*. (note that MATLAB uses *N* for represent columns of a matrix, hence the author's choice of variable name: `N_cols`)

```
// build the ANN  kd-tree using the ANN variables

the_tree = new ANNkd_tree(
          pt_array, // data points (an array of ANNpoints)
          M_rows,     // number of data points (M)
          N_cols);  // dimension of data (m)
```

Step 6: set pointers to output

Two MEX matrix data types were created, and were pointed to by the pointers: `*prhs[0]` and `*prhs[1]`. This allowed the near neighbour matrices to be returned to MATLAB.

```
// output the near neighbour arrays.. to return to MATLAB
//(workspace)..
     plhs[0] = mxCreateDoubleMatrix(M_rows, num_nbrs, mxREAL);
     plhs[1] = mxCreateDoubleMatrix(M_rows, num_nbrs, mxREAL);
     output0 = (double*)mxGetPr(plhs[0]);
     output1 = (double*)mxGetPr(plhs[1]);
```

Step 7: Search kd-tree and return results to MATLAB

The kd-tree was searched by querying it with each point in the data set. It sent the near neigbour *lists* back to MATLAB via the pointers set in step 6. Note also the **+1 (in bold),** because MATLAB indexes arrays with one, while C++ does so starting with zero. Note that all zeroth near neigbours are ignored, see chapter 7 on validation for full explanation of why the code was implemented as shown.

```
for(int i=0;i<M_rows;i++) {

      // search kd-tree
      the_tree->annkSearch(
      pt_array[i],            // query point
      num_nbrs,               // number of NNs (+1 here if NN of itself
is
                              // NOT ignored -see ANN.h)
      nn_index_list,          // near neighbour index list (returned)
      nn_distance_list,       // near neighbour distance list (returned)
      error_bound);           // error bound


      // insert indices/distances back
     //into MATLAB mxArray format
          for(int k=0; k<num_nbrs; k++) {
            output0[i +M_rows*k] = nn_index_list[k]+1;      // +1
                                  //because matlab counts from 1 (not zero)
            output1[i +M_rows*k] = nn_distance_list[k];
                          //  use [k+1] if ANN.h flag is set to true.
                          // ..i,e. if NN of itself is not ignored,
                          //then [k+1] needed here.
         }
      }
```

Step 8: Memory management

Finally the used data structures are deleted to avoid memory leaks

```
delete pt_array;
delete nn_index_list;
delete nn_distance_list;
delete the_tree;
```

6.2 The *core* Gamma test

By core, we refer to the Gamma test algorithm when given pre-computed near neighbour indices and distances, which were named `nniMatrix` and `nndMatrix` respectively. This code was implemented in MATLAB, as reasoned in the design chapter. The code has been implemented as stated in equations: EQ 2.1 and EQ 2.2 of chapter 2. Other arguments to the `gammatest` function are the input we call '*y*' and the graph selection flag we call '`graph`'. The outputs to this function are the Gamma statistic and the slope of the regression line.

```
function [GT_stat slope] = gammatest(y, nniMatrix, nndMatrix, graph)

% usage message and default setting (not shown here)

[M p] = size(nniMatrix);

gamma = zeros(M,p);
delta = zeros(M,p);
for i=1:M
    for k=1:p
        gamma(i,k) = 0.5*(y(i) - y(nniMatrix(i,k)))^2;
    end
end
Gamma = mean(gamma);
Delta = mean(nndMatrix);

% linear regression
co = polyfit(Delta,Gamma,1);

% graphs plotted (not shown here)
```

Note that the near neighbour distances are returned from the ANN code, thus only corresponding gamma values (EQ.2.2) are calculated in the loop above. When other

metrics are calculated in the ANN code, such as Max and Manhatten, the corresponding gamma values are still Euclidean. These other metrics had been added for experimental purposes.

A linear regression plot was also implemented in this function; see Appendix A.2 for source code. Two options are also supplied should the user wish to plot a scatter of all gamma and delta values (EQ. 2.3 and EQ. 2.4), or simply their averages for each near neighbour distance (EQ. 2.1 and EQ. 2.2)

6.3 Metrics

To implement different metrics in the near neighbour calculations it was necessary to create different ANN libraries for each metric, then to MEX compile the C++ code of section 6.1. For each metric the C++ code file name was changed; the following names were chosen:

```
euclidfastnn.cpp
maxfastnn.cpp
manfastnn.cpp
```

which became:

```
euclidfastnn.mexw32*
maxfastnn.mexw32*
manfastnn.mexw32*
```

* or `.mexglx` (on the Linux platform)

Files were MEX compiled with the following command on the MATLAB command prompt:

```
mex -I"C:\include" -L"C:\lib" -lann euclidfastnn.cpp
```

and similarly for Linux, except with different path references for the library and header files. For each MEX compilation, as stated earlier a library specific to that metric was needed. In order to do this, one of the ANN header files (`ANN.h`) was modified, and the *make* utility was re-executed, see Appendix A.5 for the Makefile used. See Appendix A.2 for code segment of ANN.h that was modified.

To incorporate the metric selection into the whole software package, a script was written called `fastnn.m`, which took a string argument that allowed the appropriate MEX module to be called. See Appendix A.2 for this script. Note the Max and Manhattan distances are squared when returned back to the core Gamma test script, `gammatest.m`. This was something that was overlooked in specification and design. However given that the use of these other metrics is experimental, squaring them in the same way as Euclidean may not be appropriate.

6.4 Error checking, default settings and usage messages

To keep the whole package modular, most error checks were done in the (main) script: `gammastat.m`. The reader should refer to the code in Appendix A.2. The following error checks were implemented:

a. Check if input output data set is numeric
b. Check that at least two arguments are entered.
c. Check that the input data has the same number of columns as the output data
d. Check that number of near neighbours is a positive scalar integer.
e. Check the error bound is zero or a positive numerical scalar
f. Checks the number of near neighbours specified is less than the number of data points
g. Checks that metric argument was spelt correctly

The following defaults were set in `gammastat.m`:

a. Graph was set to plot regression.
b. Number of near neighbours was set to 10
c. Error bound was set to 0.0
d. Metric was set to Euclidean

The following error checks and defaults were set in `mtest.m`:
(note that most errors were caught by `gammastat.m`)

a. Checks that at least two arguments were entered
b. If no arguments are given, the script loads a test file supplied called `Sin500.txt`[3]
c. Number of near neighbours was set to 10
d. The step size for the M-test was set to 10
e. The confidence level was set to 90%

The scripts `gammastat.m`, `fastnn.m` and `gammatest.m` all return friendly usage messages describing their input arguments; and default settings. To get this usage message the user simply enters the name of the script without any arguments.

6.5 The M-test with heuristic confidence intervals

The M-test was implemented by iteratively calling `gammastat.m` with an increasing number of data points. The step size of increase was set to a default of 10. Each increase in the number of data points contained the subset of all previously used data points that

142

[3] Sin500.txt consists of two columns of 500 real numbers. The first column represents input data; the second column represents the output data. The columns are separated by a space. Later versions of MATLAB will also accept a comma as separator.

were used to calculate the Gamma statistic. Also note that the regression plot was automatically suppressed so as not to slow down the M-test script.

Heuristic confidence intervals [1]

These were implemented with the help of Samuel Kemp of University of Glamorgan. The confidence intervals were calculated using a heuristic method because the distribution of Γ is at present theoretically inaccessible. We are unlikely to have enough data to make an empirical approach feasible. Even if we had sufficient data, it would take too long to estimate each distribution of Γ empirically.

In an experiment conducted using artificially generated data, EQ 6.1 was observed to hold approximately, thus from Central Limit theorem we can assume that the distribution of Γ is normal (except at the tails) as *M* tends to infinity.

$$\Gamma_{SD} \; \alpha \frac{1}{\sqrt{M}} \qquad \text{EQ. 6.1}$$

Γ_{SD} = standard deviation of Γ

Consequently for an M-test we can estimate the standard deviation for all *M*, without computing the actual distribution. From this we can then calculate the standard error, *se*.

Using *se* and Γ, confidence intervals were calculated using the Students t test [35]. MATLAB has a built in function for computing this, thus was implemented as seen in Appendix A.2: `mtest.m`. The confidence intervals were returned and plotted to a graph, see Figure 6.6 for example plot.

6.5 Overview of finished package

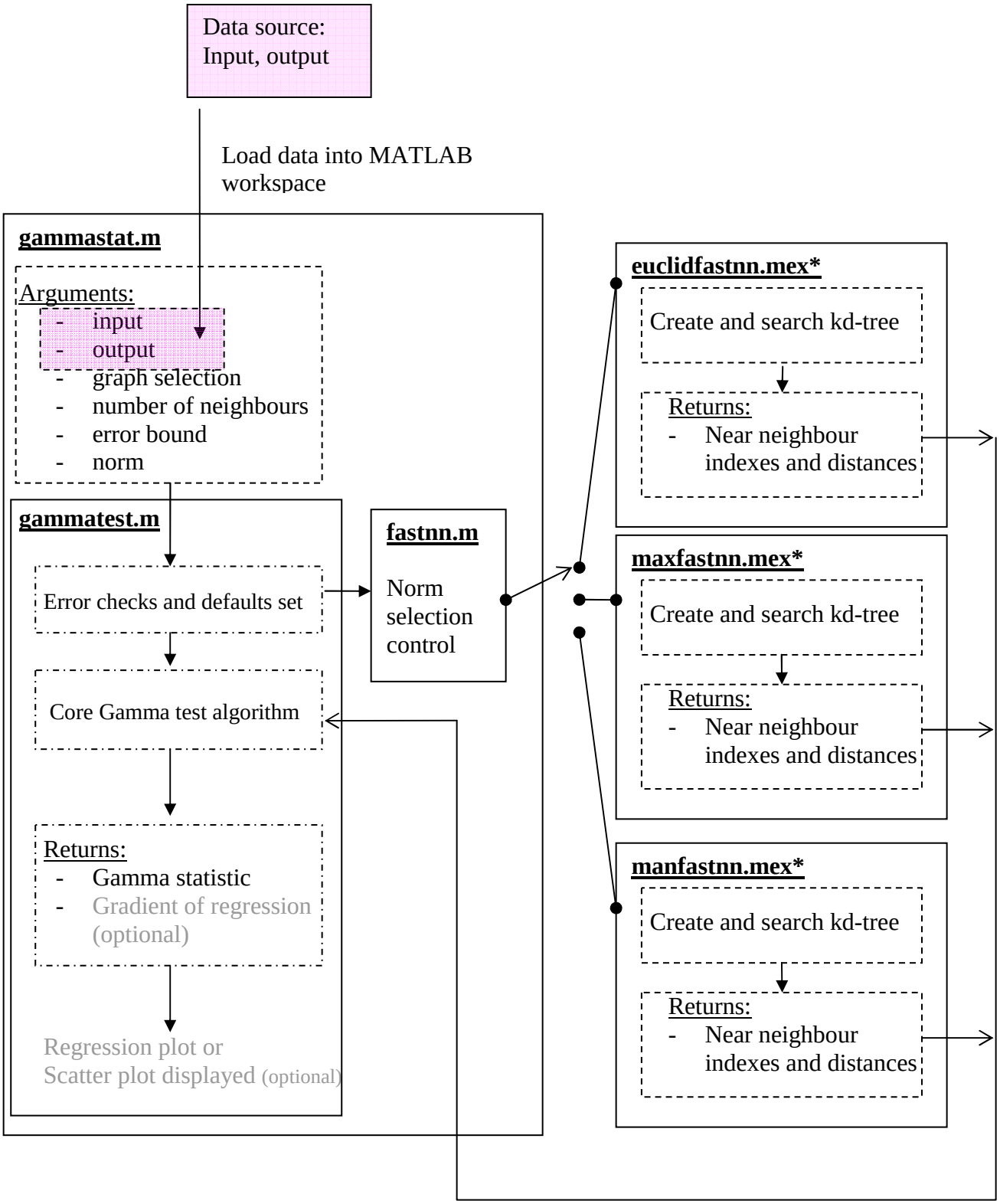

Figure 6.2: Overview of main components of code. * File extension is `.mexw32` on windows (since MATLAB version 7.1 (R14SP3)), and `.mexglx` on Linux. M-test script not shown in this diagram.

Figure 6.2 shows that there are three MATLAB scripts and three MEX modules. This emphasizes the modularity of design and implementation, thus making the package as a whole more maintainable and understandable should a user wish to use each MATLAB script module separately, or upgrade parts of the code. The M-test effectively uses the entire package, by calling `gammastat.m` for increasing *M*.

In order to load data in, the MATLAB `load` command is used. Data in the file can be of various formats, however for the purposes of this project, data was loaded in from numerical text files. The user manual in Appendix B describes a recommended format for data files. Alternatively, a script was written to generate data artificially for experimental purposes (the code of which can be found in Appendix A.2, filename: `noisy_sin_func.m`).

6.7 Output figures: regression, scatter and M-test plots

To obtain and plot 500 data points of an underlying sin function with noise variance 0.075 (normally distributed noise) the code below was run and Figure 6.3 was obtained.

```
[x y]=noisy_sin_func(500,0.075);
figure(1);
plot(x,y,'+')
xlabel('x');
ylabel('y');
```

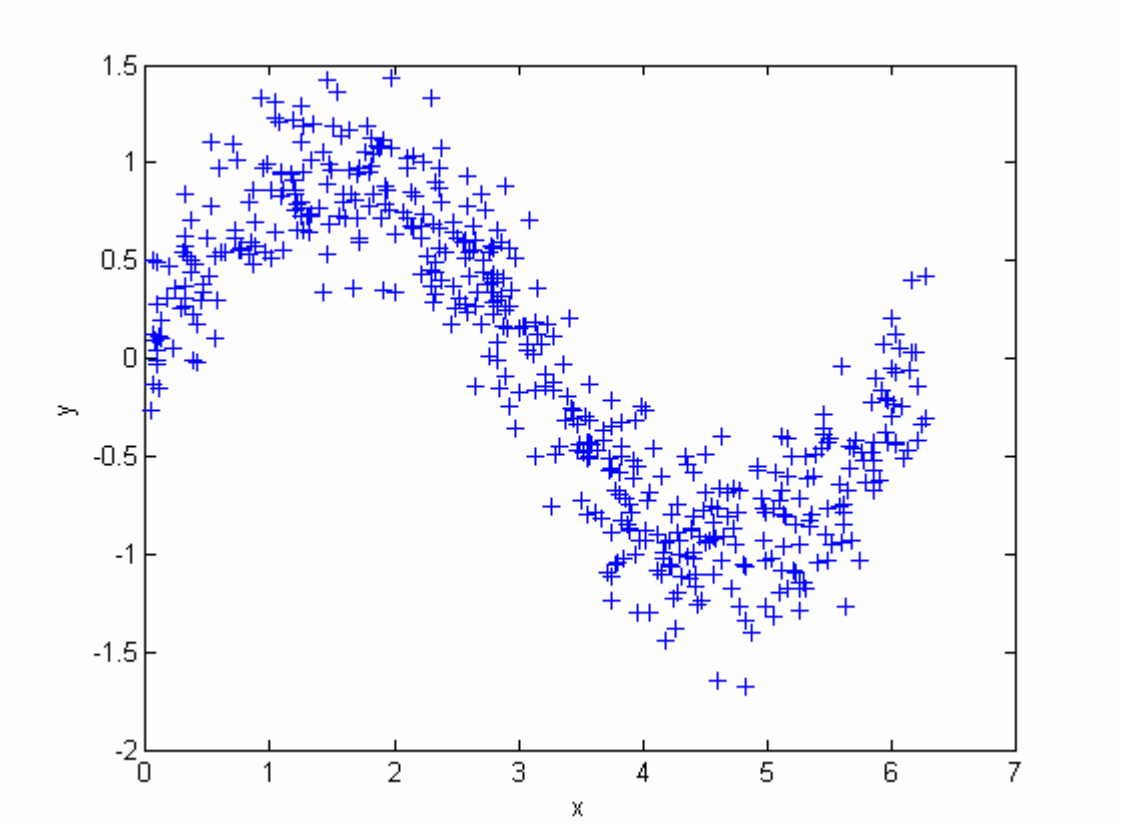

Figure 6.3: Noisy sin data generated using `noisy_sin_func.m` script. See Appendix A.2 for source code.

The Gamma test is run by typing for example, `gammastat(x,y,'regression')`, or `gammastat(x,y,'scatter')`. Defaults would then be 10 near neighbours, zero error bound and Euclidean distances.

In this case a reasonable estimate for the variance of the noise of 0.07330162468721 was returned, the slope was 0.50725252200484. Figure 6.4 and 6.5 show the regression plots. Figure 6.5 includes all the gamma and delta values (EQ. 2.3 and EQ. 2.4) known as 'the scatter plot', where as Figure 6.4 only shows gamma and delta values (for each near neighbour) as calculated in EQ. 2.1 and EQ. 2.2 with linear regression line. Code for these graph plots can be found in appendix A.2: `gammatest.m`.

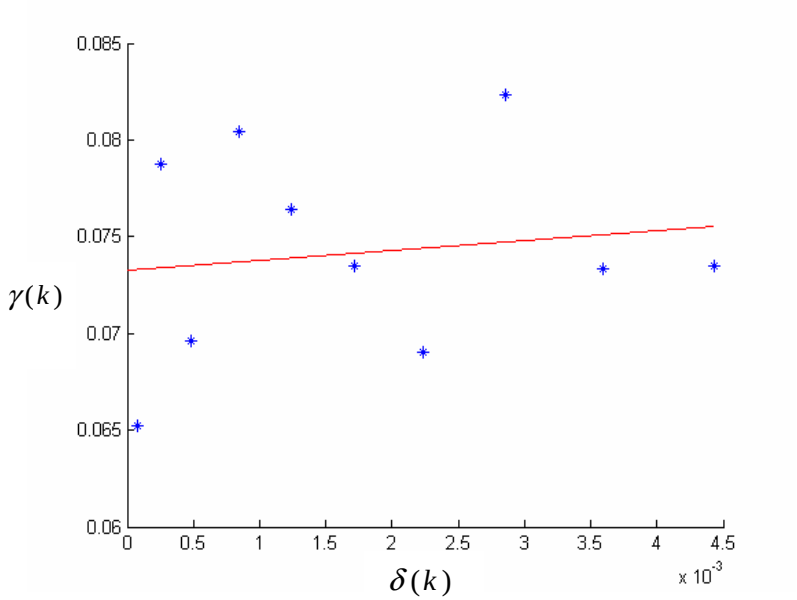

Figure 6.4: shows regression plot for averaged gamma and delta values (as stated in equations EQ 2.1 and EQ 2.2 (see Chapter 2, section 2.2) and 10 near neighbours.

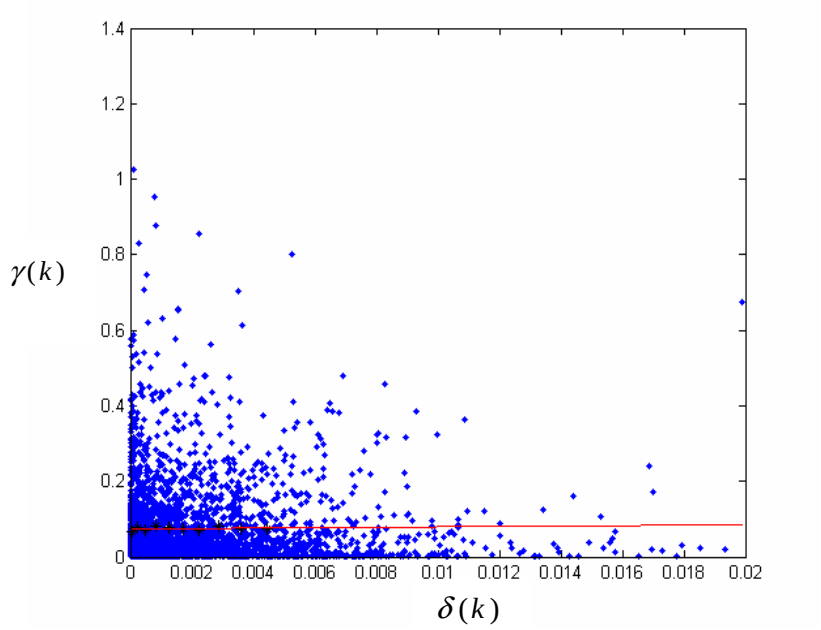

Figure 6.5: This is the same as figure 6.4 but also includes all delta and gamma values as calculated in EQ.2.3 and EQ.2.4 (see Chapter 2, section 2.2) for10 near neighbours.

An M-test (see Appendix A.2 for source code) was also run using the data generated from the `noisy_sin_func.m` script which produced Figure 6.6.

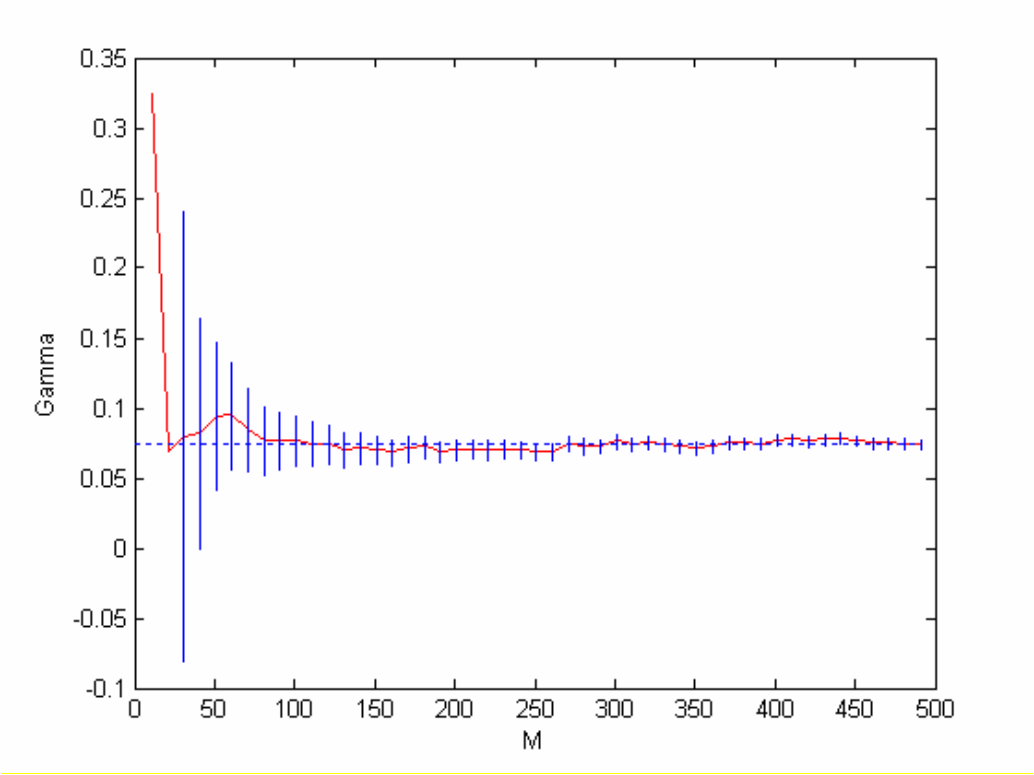

Figure 6.6: M-test using data generated from noisy_sin_func.m. 500 data points and a variance of 0.075. As can be clearly seen, the Gamma statistic (red line) converges to the true variance of the noise (dashed line). The 90% confidence interval (vertical bars) also gets smaller. M = number of data points.

Chapter 7: Validation

7.1 Validating near neighbour detection

In this section near neighbour detection of the ANN library was validated by comparing results with a brute force near neighbour search written by the author (see appendix A.3 for code: `mybrute.m`, `check.m`, `visualise.m`). These scripts initially arose concerns as it was noticed that the ANN functions were *not* detecting near neighbours correctly.

The near neighbour detection anomaly was first noticed by visual inspection (similar to Figure 7.1, generated from `visualise.m`) but its nature was unclear. By using a brute force near neighbour search, further quantification of the errors could be deduced. However compounding this problem were different results observed on different platforms, which may have suggested something to do with precision handling on each platform when numbers are close to zero. Thus, to test this hypothesis, it was decided to shift some query points *outside* the original data set. This solved the problem but only on one platform. Consequently it was decided to delve into the ANN source files to see if there was anything related to precision handling. After a very long search through many source files, the author came across a flag for self matching which was changed.

The flag was found in `ANN.h` and was set as follows:

```
const ANNbool  ANN_ALLOW_SELF_MATCH = ANNfalse;
```

Code in `euclidfastnn.cpp`, `manfastnn.cpp` and `maxfastnn.cpp` was changed also to:

```
output0[i +M_rows*k] = nn_index_list[k]+1;
output1[i +M_rows*k] = nn_distance_list[k];
```

as opposed to:

```
output0[i +M_rows*k] = nn_index_list[k+1]+1;
output1[i +M_rows*k] = nn_distance_list[k+1];
```

This code change allowed the first (zeroth) near neighbour of itself to be ignored, thus the first near neighbour detected by the ANN search *is* indeed the first near neighbour to that query point. Thus on recompilation of the C++ ANN library (using Makefiles); MEX (C++) compilation of the `euclidean.cpp` source code; and subsequent running of validation scripts `visualise.m`, `mybrute.m` and `check.m` on *both* Linux and Windows platforms, the problem of incorrect near neighbour detection was eventually resolved. The *extra* anomaly between platforms was in fact due to different versions of the ANN library used by accident; an update of the ANN library around the time of development had occurred.

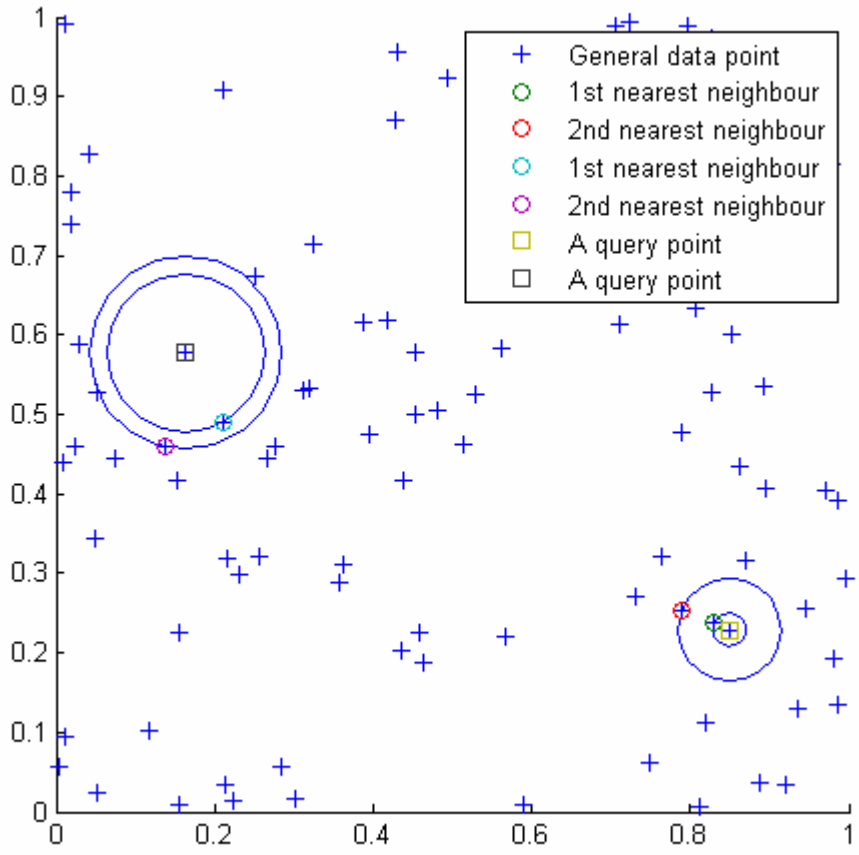

Figure 7.1: Validation by visual inspection of correct near neighbours found using ANN library functions (Euclidean distance metric and zero error bound) for randomly generated *2 dimensional* data.

From Figure 7.1 it can be seen for two query points: 1st and 2nd near-neighbour were successfully found. See Appendix A.3 for validation code: `mybrute.m` and `check.m`. Code for this graph plot can be found in appendix A.3: script name: `visualise.m`.

7.2 Validation against a known data set

In this section the Gamma test was run on a data set that had a known Gamma statistic derived from previous implementations of the Gamma test. This validation was carried out on both platforms: Linux and Windows. The calibration data set (called sin500.asc) was obtained from [22]. It's description has been put in appendix A.3 for completeness. The MATLAB implementation returned correctly to 17 digits the value of the Gamma statistic and the slope correctly to 13 digits as follows:

Gamma statistic = 0.0733545595048562
Slope = 0.711221348889964

Calibration values were:
Gamma statistic = 0.0733545595048562
Slope = 0.711221348889957

Since solving the problems of near neighbour detection (see section 7.1) this degree of accuracy was well within validation requirements.

7.3 Validating error checks

Extensive tests were also done to ensure that all possible error conditions for input arguments were caught to avoid MATLAB crashing. Error checks employed as described in the Implementation chapter (section 6.4) showed to be very robust. Also friendly error messages were returned to the user providing helpful information regarding the specific error encountered.

8. Performance testing

In this section we looked at how execution time varied with:

- Dimension
- Error bound
- Metric
- The number of data points

For error bounds, we also looked at the number of incorrectly found near neighbours.

The purpose of this section was to get a feel for performance *trends* of the ANN library. All experiments call the ANN MEX modules directly from MATLAB. Performance of the Gamma test as a whole is not looked at because the 'core' Gamma test is known to scale as O(*M*), which is insignificant compared with O(*MlogM*) of the fast near neighbour search. However if time was permitted, it would have been preferable to test performance of the system as a whole.

It is not advisable for the reader to look at absolute running time results, as these performance experiments were carried out on a fairly standard desktop PC (Pentium 4, 2.6Ghz, 480MB RAM), with numerous other background processes running related to the operation system (Windows XP). Some effort was made to minimized the effect of other processes running thus to some extent results in this section are atleast comparable to *one another*, but absolute execution times should be taken as rough indications. Scripts for all these tests can be found in Appendix A.4.

8.1 Error bound and dimension

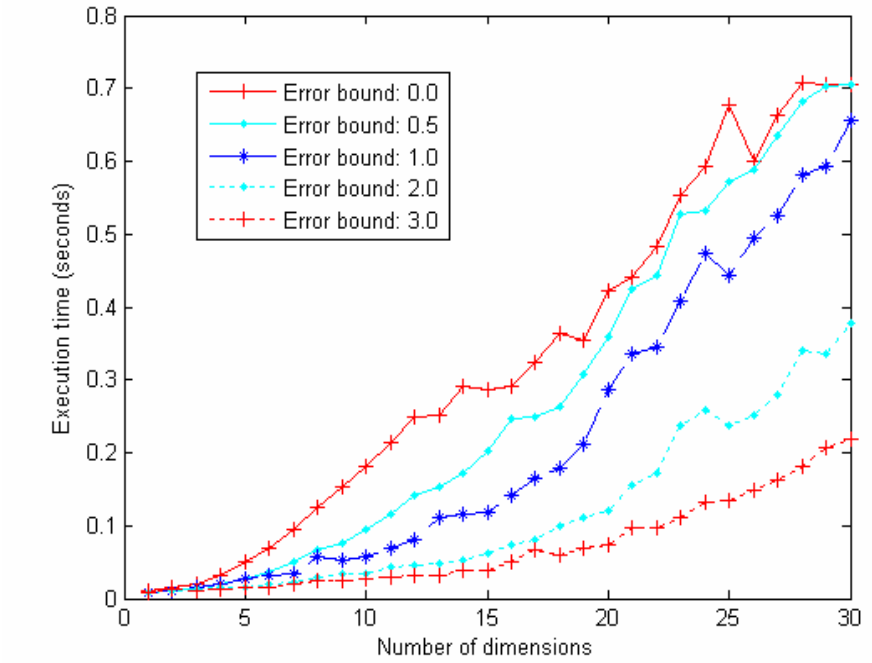

Figure 8.1: Execution time versus number of dimensions, for different error bounds. Other settings: 10 near neighbours, 1000 randomly generated data points.

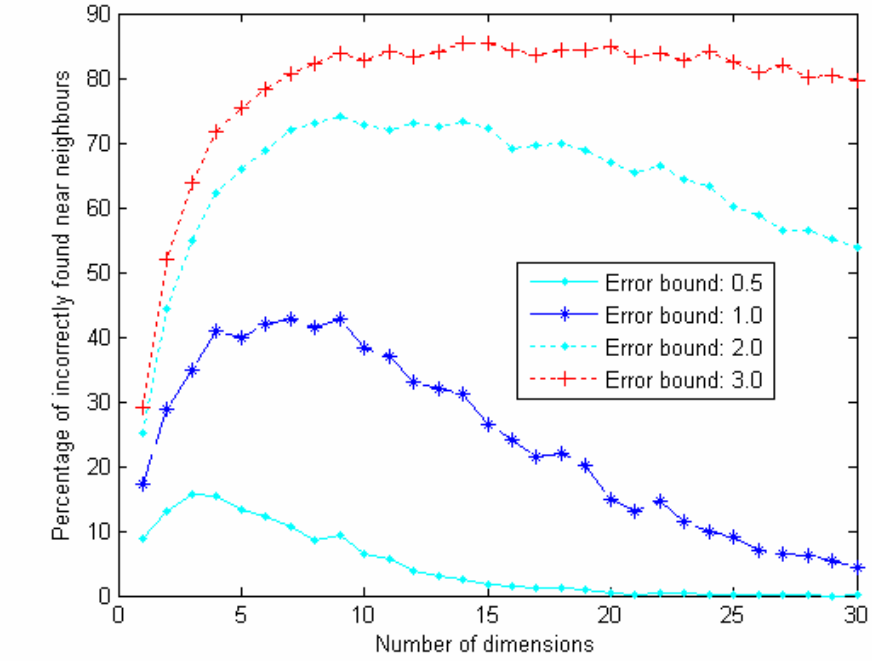

Figure 8.2: Percentage of incorrectly found near neighbours versus number of dimensions, for different error bounds. Other settings: 10 near neighbours, 1000 randomly generated data points.

From Figure 8.1 it can be seen that execution time does increase quickly with dimension. However it does not appear to be increasing exponentially as was previously described in Chapter 2. There is perhaps some exponential dependence to start with, particularly for zero error bound, but after 12 dimensions appears to increase linearly. The scales of these graphs will distort what we can visually deduce, thus higher dimensions (> 30) would need to be tested to view the trend properly.

From figure 8.2 we can deduce that the number of errors is increasing, as expected, but also starts to decrease, especially at lower error bounds. The reason for this is unknown by the author. The relationship is clearly logarithmic at least initially. However these percentage errors cannot be taken too seriously because they do not account for how *far* the incorrect detection of near neighbours is. For example if the *true* indexes for a set of 5 nearest neighbours to a query point was 22,44,58,10,15; and if the error bound increased so that the nearest neighbour list became **44,22**,58,10,15; the algorithm used here (see Appendix A.4) would detect that two near neighbours were found incorrectly (i.e. 40% incorrect). However the list **58**,44,**22**,10,15 would also be detected as 40% incorrect, despite the fact that the true 1st nearest neighbour index (of **22**) is now *two* places out, thus is quantifiably more incorrect. However for the purposes of this project, this level of quantification was not pursued, but readers should be aware of this limitation. In fact, the error bound *e*, as described in section 4.1 *may* cause any near neighbour distance detected to exceed its true distance by up to a factor of (1+*e*).

Next we look at execution time versus error bound for various dimensions, and also look at the percentage of incorrectly found near neighbours. As can be seen from Figure 8.3 execution time does decrease quite rapidly initially, at the cost of increased errors as shown in Figure 8.4. To observe this relation more closely the scatter plot of Figure 8.5 vaguely indicates that as the error bound increases, there comes a point where execution time does not really improve at the cost of increased errors. In other words the percentage errors are saturating at some point, but this (as stated earlier) is probably mainly due to the limitations of the error method used to detect the percentage errors. The code for this can be found in Appendix A.4.

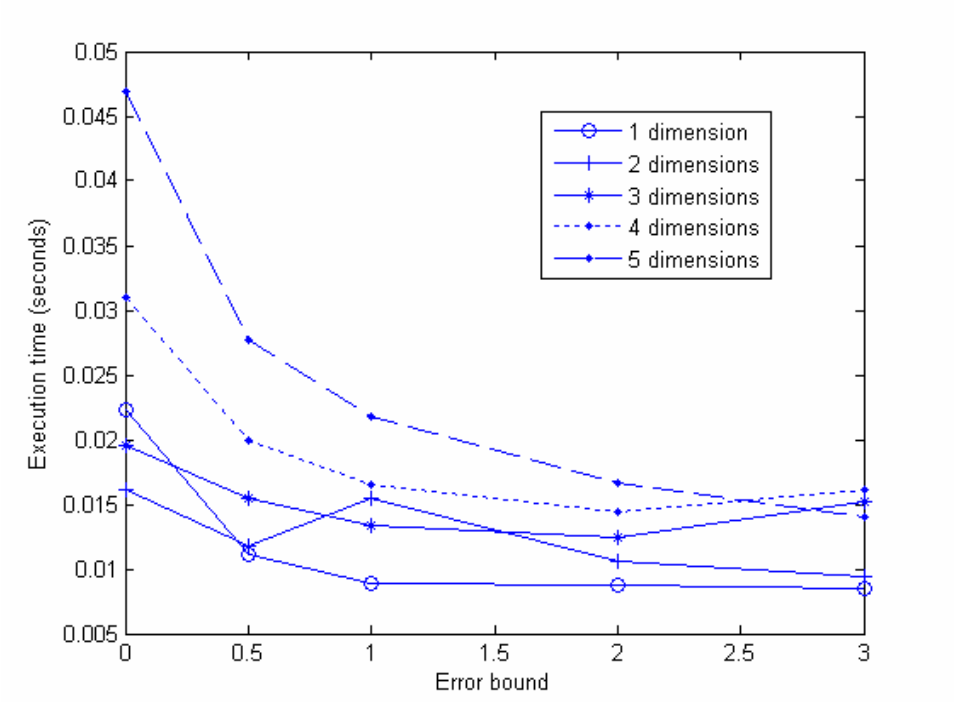

Figure 8.3: Execution time versus error bound, for various dimensions. Other settings:10 near neighbours and 1000 data points.

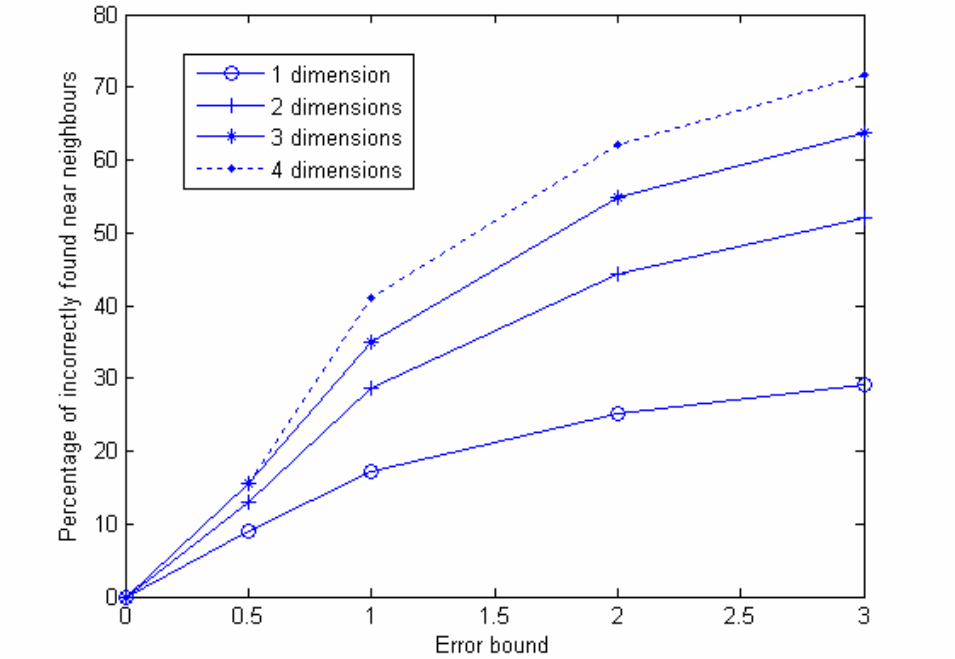

Figure 8.4: Percentage of incorrectly found near neighbours versus error bound for various dimensions. Other settings:10 near neighbours and 1000 data points.

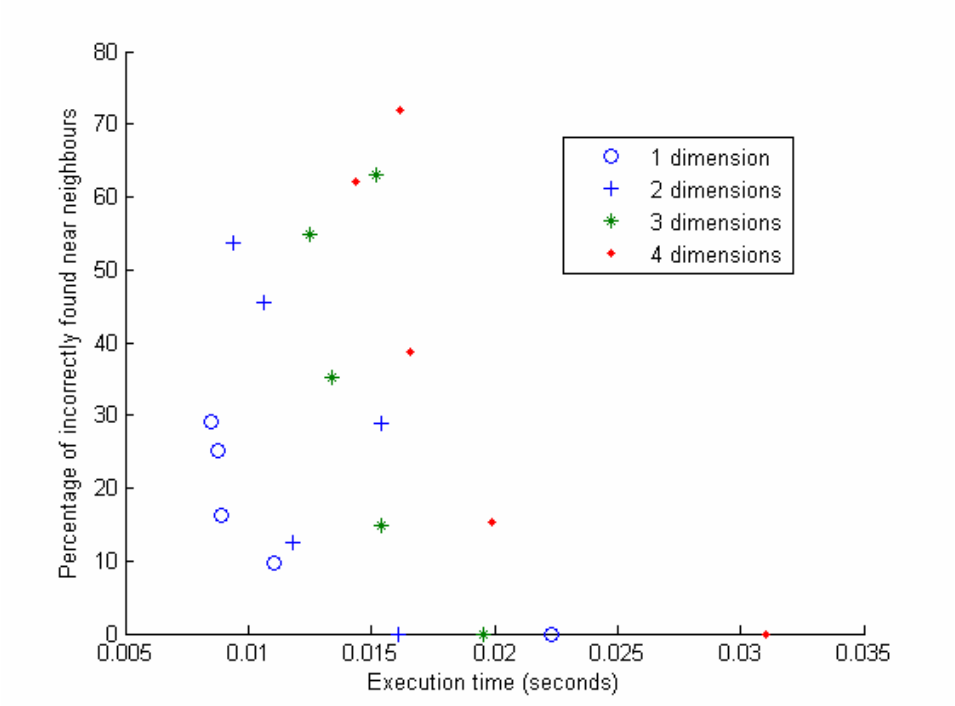

Figure 8.5: Correlation of percentage errors versus execution time.

8.2 Metrics

In this section we look at the execution time performance for different metrics. Note that for consistency, as stated in EQ. 2.1 and EQ. 2.2 (chapter 2), the Gamma test requires squared Euclidean distances, however the ANN library returns non-squared distances for the other metrics of Max and Manhattan. For consistency these were squared using a standard MATLAB command. The implication of this increases execution times for Manhattan and Max metrics, but is quite negligible as shown in figure 8.9 (for 1 dimensional data); for higher dimensions the effect of squaring is probably more significant, but due to limited time in this project this was not investigated.

Figure 8.6 shows that the Manhattan metric is approximately *twice* as slow as the other metrics; this is particularly noticeable as the dimensions increase. This again may be due to the effects of the squaring operation done in MATLAB, but further experiments would be needed to verify this. The Max metric is similar in performance to Euclidean. Perhaps surprisingly Euclidean is the fastest. This could be due to the fact that results are already pre-squared at the faster C++ level, or that low level hardware operations are optimized for

squaring operations. Further analysis is out of the scope of this project but is worth being aware of.

Figures 8.7 and 8.8 show how performance varies with the number of data points. For 1 dimensional data, perhaps un-surprisingly all metrics perform similarly. However for 5 dimensional data the Manhattan distance is (yet again) slower in execution by approximately a factor of two (see figure 8.8)

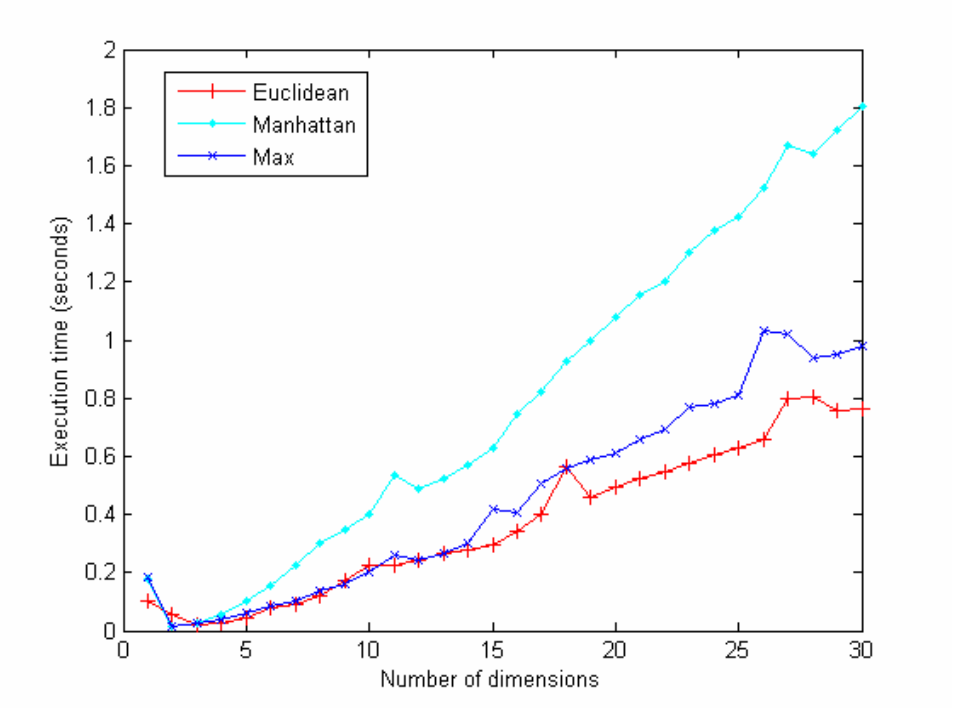

Figure 8.6: Execution time versus the number of dimensions, for different metrics. Other settings:10 near neighbours and 1000 data points. (Note: Manhattan and Max distances were squared in MATLAB, Euclidean metric was returned *pre*-squared the from C++ ANN MEX module.)

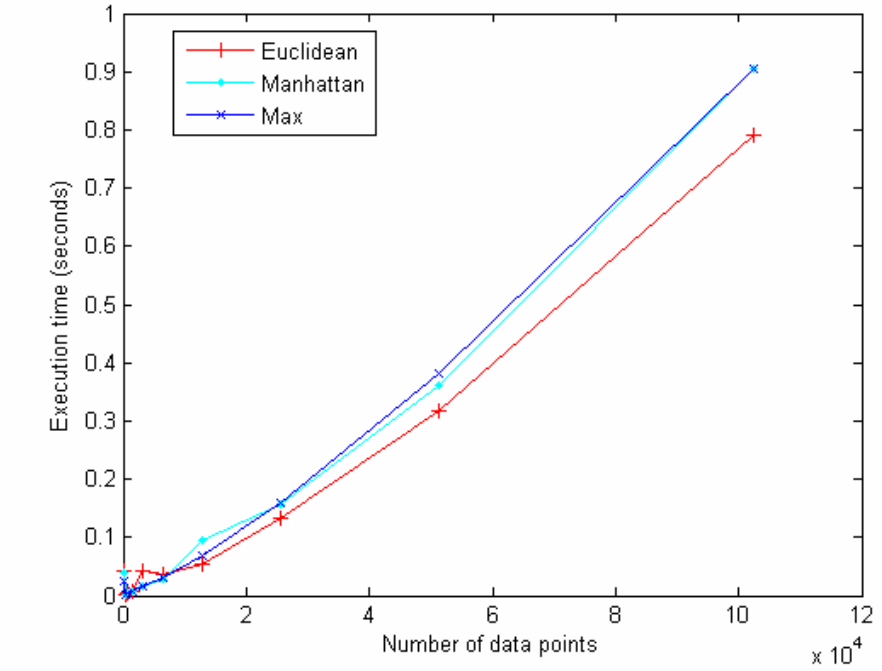

Figure 8.7: Execution time versus the number of data points for different metrics. Other settings:10 near neighbours; 1 dimension. (Note: Manhattan and Max distances were squared in MATLAB, Euclidean metric was returned *pre*-squared from the C++ ANN MEX module.)

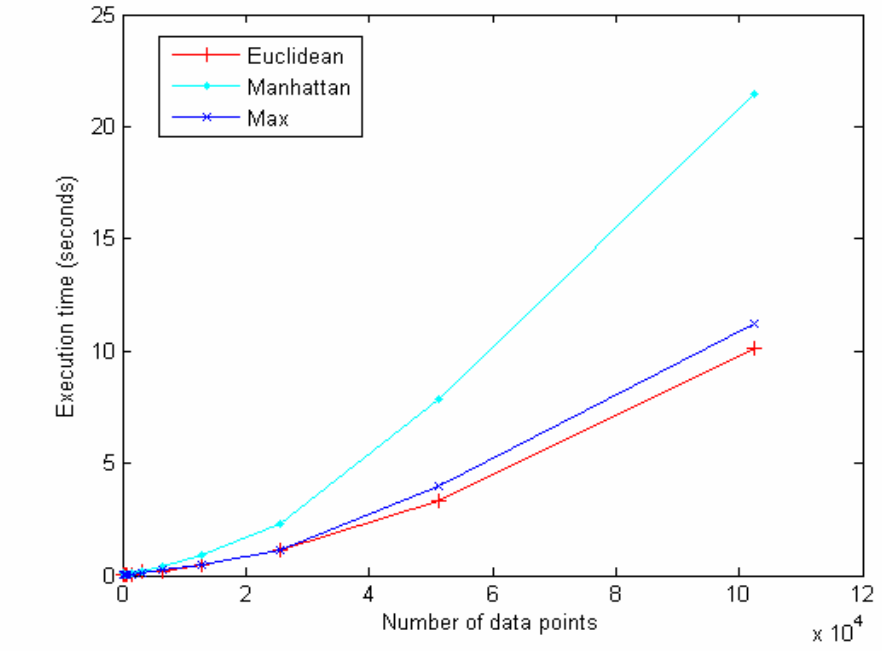

Figure 8.8: Execution time versus the number of data points for different metrics. Other settings:10 near neighbours, 5 dimensions. (Note: Manhattan and Max distances were squared in MATLAB, Euclidean metric was returned *pre*-squared from the C++ ANN MEX module.)

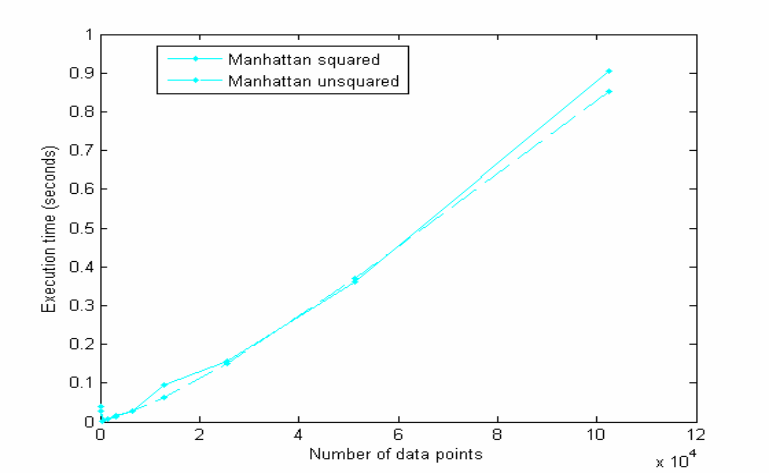

Figure 8.9: shows the effect of the squaring operation in MATLAB is negligible to overall performance. Unsquared result was <0.1 seconds different to squared metric for >100,000 data points. Other settings: 10 near neighbours, 1 dimension.

8.3 Conclusions

From our performance tests we can draw a few basic conclusions.

- The error bound significantly improves running time at the cost of errors, but the quantification of these errors was limited as described earlier.
- The Manhattan metric appears to run approximately twice as slow as the other metrics Euclidean and Max. This is more apparent as the number of data points or dimensions increase. This result is unlikely to be caused by the squaring operation carried out in MATLAB for Max and Manhattan metrics, but further tests will need to be done to confirm this. For example by investigating the inner workings of the ANN library and understanding exactly how the metrics are calculated.
- Execution time does not appear to grow exponentially with dimension as was previously thought (see section 2.4). However more tests would need to be done to view this trend fully.
- Overall, due to limited time in this project performance results were quite rough and ready. More care could have been taken in monitoring, for example, memory bottle necks within the system, or other background processes related to the operating system. However these results should still give users a rough indication of the possible scale of their application; and the main factors that affect performance.

Chapter 9: Evaluation

In order to implement the software system (as a coherent package) to a high degree of accuracy and reliability, validation took up a substantial amount of development time, mainly due to the near neighbour detection error that it uncovered. Trying to deduce the cause of these errors was initially very difficult given the large number of source files used in the ANN library. However, the validation scripts (see Appendix A.3) could be automated in such a way so as so understand the nature of the errors and interpret potential areas of where the problem could lie. From validation tests, the way in which the ANN library dealt with zeroth near neighbours was made clearer. However due to the author's unfamiliarity with the ANN library and C++, this error was very time consuming to track down. Compounding this problem was the authors attempt to develop on two platforms simultaneously: Linux and Windows, to meet specification: FR4. There were also some inconsistent results on each platform which was due a mix up of ANN library versions from the ANN website [3] (the ANN library version updated around the time of development). On retrospect, development should have been focussed on one platform only, but the inconsistencies between platforms initially raised questions about precision handling on different platforms which turned out to be untrue. Until these inconsistencies were *understood*, it was thought that development should continue on both platforms in order to make an overall valid system.

The specifications as laid out in section 3.1.1 and 3.1.2 were not *all* defined at the beginning of the project, however the core functionality of FR1, FR2, FR3 and FR4 were defined. Most other specifications were added during development, thus the incremental approach to development had to allow some adaptability. In this sense an evolutionary methodology was followed for the rest of the requirements. Features such as the M-test were added after the core functionality was implemented. Given the modularity of the code, it was necessary for any changes made to be consistent through each module. Optional arguments to functions had to be carefully ordered and set so as not to clash with other modules (see Appendix B for user manual). However this modularity will allow extra features to be added in the future, without the need for extensive testing of all parts of

the system: it is also easier to distinguish the relative independence of each module. For example if more variants of the kd-tree were to be implemented in C++, it is easier to have a script module dedicated to call individual variants, along with error checks associated with them, rather than trying to bundle the whole system into one function. It also means that the kd-tree modules can be used separately if a user should so wish.

Much time was spent to ensure error checks implemented in this package covered most scenarios in which errors may occur. However in practice it is very difficult to make a system *completely* bug free, never the less by using a modular approach to design and implementation most errors conditions should have been caught.

The importance of having a clear specification makes design and implementation more focussed. Specifications in this project were prioritized in the same way an incremental methodology would follow, in order to deliver and test core parts of the system as early as possible. However the sooner all the specifications are known, the more prepared one can be to design the system. If (as in this project) many specifications were defined later, the author had to be prepared to foresee those modifications by following a modular approach to design.

As stated earlier (Chapter 4) a preliminary investigation was needed to find the best approach to tackle this project. After this preliminary investigation was completed it was also easier to quantify a time scale in which to complete the project, and incrementally complete components of the system.

The initial specification overlooked the handling of zeroth near neighbours (mainly due to other priorities). However, after closer inspection (during validation, see section 7.1) of the ANN source code and by talking with others in the Evolutionary Computing Research Group, it was decided in this case to ignore *all* zeroth near neighbours.

Equidistant near neighbours were dealt with automatically by the ANN kd-tree. It appears that equi-distant near neighbours were ordered arbitrarily. The effect of dealing with

equidistant near neighbours by grouping them together in the Gamma test calculations was out of the scope of this project, due to time limitations.

Using C++ and taking care with memory management was a new concept for the author to understand. In Java for example this is done automatically. It was important to be aware of memory leaks as this could drastically reduce performance especially if the Gamma test was to be run for long periods of time, perhaps days. This software has not been tested for very large data sets, so minor memory leaks may exist, but are unlikely. MATLAB does some automatic memory management in some of its releases, but generally the designer of the MEX module should take responsibility for memory management.

It was initially thought to include an experimental part to this project. However, due to time constraints this was not possible.

It also became apparent during implementation of the question of whether or not to square the other metrics Max and Manhattan in the same way Euclidean metrics are squared as in EQ. 2.1. It seemed more intuitive to square these distances, so this was done (see Appendix A.2 for code: `fastnn.m`). The inclusion of these other metrics was largely experimental. Performance tests were generally carried out *including* this squaring operation as part of the overall near neighbour calculation. However, it seemed to have a negligible effect on performance, but this was not investigated in enough detail, nor was it a particular priority for this project.

There was significant overlap between the preliminary investigation, design and implementation chapters, but this report should still convey the logical approach taken to carry out the project objectives.

Chapter 10: Future work

There is much scope for further work ranging from improving the usability and features of this MATLAB implementation, to applying the Gamma test to numerous applications.

A graphical user interface would broaden the access of the Gamma test to scientists and engineers unfamiliar with the MATLAB command line interface. Given that the Gamma test has a wide breadth of multidisciplinary uses; it would make sense to improve the user interface to allow additional accessibility to less computer literate fields. MATLAB does in fact provide tools to make a graphical user interface. Further implementations on platforms such as Mac would also increase this accessibility.

After near neighbour indexes and distances are returned from the MEX module, the rest of the Gamma test is done within MATLAB; this part of the package runs in O(M) time. However, MATLAB scripts, in particular loops, are known to run quite slowly given that the code is not compiled. Also if there are many outputs in the data set, this would require significant computing to be carried out at the MATLAB script level. Thus a future improvement to this package would be to implement the rest of this Gamma test code into C++.

Speculatively, there may be a way to further optimise the near neighbour search depending on the distribution of data. Perhaps by taking a randomly sampled sub set of the data, to gauge its distribution, we could then pick an optimal near neighbour data structure and search algorithm *for that* distribution. For example the bd-tree offered in the ANN library is optimised for clustered data.

Experiments could be carried out to investigate the effects of altering various parameters or options of the Gamma test such as

- determining the best number of near-neighbours to choose for the linear regression, and possibly how this relates to the rate of M-test convergence.
- determining the affects of different metrics, and its impact on computing the Gamma statistic for different kinds of data.

- determining the affect of the approximation setting (used in the ANN library) on the Gamma statistic for a given data set. Also its impact on computational time could be further investigated.

Many of these experiments mentioned could be carried quite rapidly in conjunction with powerful and flexible MATLAB tools.

In fact a wide range of experiments could be carried out with Gamma test in combination with the extensive variety of MATLAB toolboxes. The de-noising of data using the universal threshold and wavelets is one such possible research area [29, 30].

For larger applications a distributed system may be more appropriate, whereby the fast near neighbour algorithm is distributed, and results are sent back to MATLAB.

Further tests could be carried out on data that contains zeroth near neighbours or equidistance near neighbours. Performance tests could also be carried out on fixed data sets, in order to compare performance times for different implementations of the Gamma test.

Applications to time series analysis such as in climate science are numerous. The Gamma test could be used to pick time varying attributes of the earth, such as carbon levels, dust levels, methane levels, ocean temperatures, acidity or even carbon deposits in the sea bed. Data could be obtained from various sources such as coral samples, ice core samples [4] and deep sea ocean cores. The Gamma test in its most efficient form could then be run to pick out values of these attributes at various time intervals in the *past*, in order to predict say carbon dioxide or temperature levels at a future time. If the correct variables are found (at the correct time lags) that could reliably predict the Earths temperature, this model could be iterated into the *unknown* future. Non-parametric models have an advantage in that they may be able to capture dynamics of a system that would be very difficult (perhaps impossible) to capture from attempting to understand the underlying principles the system. Although iterating a short term model to obtain a longer term prediction has inherent pitfalls.

Chapter 11: Conclusions

Implementation of the Gamma test in MATLAB using a fast near neighbour search algorithm in C++ was successful. The system also worked on *both* platforms: Linux and Windows. It was also validated to a fairly high degree against various test scripts and a calibration data set. The M-test was implemented with the added feature of confidence intervals as described in [1]. All functional requirements specified in Chapter 3 were met. Non- functional requirements were met to an acceptable level, and the system reliably caught many errors conditions. The system was also modular in design, thus easier to understand and maintain in the light of probable future modifications.

The system has not been run for long periods of time to check memory is allocated and de-allocated in the most efficient way. This is something for future work, but it is thought to be reliable in terms of memory management.

All zeroth near neighbours are ignored in this implementation. It appears that equidistant near neighbours are arbitrarily ordered automatically by the ANN library's kd-tree. To understand the exact way in which equidistant near neighbours are handled would require examining the ANN library's source code.

Other metrics of Max and Manhattan distances were implemented for experimental purposes and users can choose a metric most appropriate to their application. These other metrics are squared when used to derive the Gamma statistic as in EQ 2.1 (see section 2.2.1).

Performance tests for varying approximation settings, metrics, data points and dimension showed some interesting trends. It highlighted the use of different metrics having different computational times, which varied by up to a factor of two. However asymptotic behaviour is thought not to be affected. These tests also seemed to suggest that run times varied *polynomially* with dimension, rather than exponentially as was previously thought (see section 2.4). However, further tests would need to be done.

There is much scope for future work and improvements as outlined in the chapter 10, ranging from more efficient implementations; the affects of different metrics and applications such as climate modelling.

Glossary

MEX : is a MATLAB tool for compiling an external *user defined* function (written in C++ or Fortran) into callable MATLAB executable function. It also a library of functions which form the interface between a user defined subroutine and MATLAB.

MATLAB: A powerful numerical analysis and visualization package. It contains numerous toolboxes for applications ranging from engineering, statistics to biology. Each toolbox is essentially a MEX module and supporting MATLAB scripts.

ANN: Approximate near neighbour. Refers in the context of this project to the C++ library written by Mount et al.[3]

***Core* Gamma test algorithm**: in this project refers to the part of the Gamma test that does not compute near neigbours but *uses* the results of a near neighbour search algorithm.
The Gamma test: refers to the entire algorithm including calculation of near neighbours. The Gamma test is an algorithm that can calculate the variance of the noise (or error) within an input output dataset. For a full explanation the reader should refer to [17].

O(…) , E.g. O(M) : A measure of computational complexity.
Big – O notation is used to describe asymptotic behaviour of computational overheads (usually memory and/or CPU usage) for an algorithm, as the number of data points, M tends to infinity.
O(M) is described as linear run time
O(M^2) is described a 'M squared' run time
O(C^M) is described as exponential run time, where C is some constant.

API : Application programming interface. Usually refers to tools/applications (in the form of functions) available to the programmer for a given application.

M-test: is a way to gauge if the Gamma statistic estimates the variance of the noise reliably. See [1,17] for further information.

Appendix A: Code

A.1: Preliminary investigation MEX C++ code.

mytranpose.c

```
// This program takes an MxN matrix of doubles and returns
// the inverted matrix.. it also prints out the mxArray/mxGetPr
// to prove the order in which matlab represents command line
// input within the MexFunction.

// Author: Sunil Singh
// Date: 8/7/06

#include "mex.h"


#if NAN_EQUALS_ZERO
#define IsNonZero(d) ((d) != 0.0 || mxIsNaN(d))
#else
#define IsNonZero(d) ((d) != 0.0)
#endif

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

   int elements; // needed for loop
   int Minput, Ninput; // needed to create output datastruc and in loop
   double *input, *output;  // matlab representation (1D arrays) for input
and
                              //output
   int j,k;     // for loops
  int count =0; // to index the output array


  int n;    // disgnostic for loop


/* Check for proper number of input and output arguments. */
  if (nrhs != 1) {
    mexErrMsgTxt("One input argument required.");
  }
  if (nlhs > 1) {
    mexErrMsgTxt("Too many output arguments.");
  }

  /* Check data type of input argument. */
  if (!(mxIsDouble(prhs[0]))) {
    mexErrMsgTxt("Input array must be of type double.");
  }

  // assign pointer to the input..
  input = (double *)mxGetPr(prhs[0]);     // assumed input is only real
```

```
                                // .. and not imaginary.. mxGetPi
                                // input used like an array! (even though
                                // .. its a C pointer.

   Minput  = mxGetM(prhs[0]);
   Ninput  = mxGetN(prhs[0]);

   // create output datastruc...
   plhs[0] = mxCreateDoubleMatrix(Ninput, Minput , mxREAL);
   // note: N and M inverted in transpose
   output = mxGetPr(plhs[0]);   // .. and pointer to output..



  /* Get the number of elements in the input argument. */
  elements = mxGetNumberOfElements(prhs[0]);

  // some diagnostics: proving how matlab arranges its array ---------------
  for (n=0; n<elements; n++)
  {
  mexPrintf("input[%d] is:  %f \n",n, *(input+n));
  }
  mexPrintf("number of elements: %d", elements);

  // ------------------------------------------------------------------------

  for ( j=0; j<Minput; j++){
      for (k=0; k<Ninput; k++){
      output[count] = input[k*Minput +j];
      count++;
      }
  }

}
```

A.2: Code related to finished software package

Code and associated files appear in this order:

1. `euclidfastnn.cpp` - note this code is exactly the same as manfastnn.cpp and maxfastnn.cpp.
2. `gammastat.m` - Gamma test which automatically calls fast near-neighbour algorithm (fastnn.m) and gammatest.m.
3. `fastnn.m` - Call to Approximate near neighbour algorithm
4. `gammatest.m` - Core Gamma test (gets near neighbour indexes from fastnn.m)
5. `demo.m` - To get users started with using this package!
6. `mtest.m` - M test with confidence intervals
7. `noisy_sin_func.m` - artificial generation of noisy sin data with normal distribution
8. Sin500.txt - description of calibration data (in two columns, space delimited)
9. Readme.txt - To get users started with using this package!
10. `ANN.h` - partial code segment from ANN library relevant to metrics

`euclidfastnn.cpp`

```
// ------------------------------------------
// Filename:      euclidfastnn.cpp
// Created:       8 August 2006
// Author:        Sunil Singh
// version:       0.2.1
// ------------------------------------------
//
// history:
// v 0.0: initial - all working, flag and near neighbour detection sorted!!
// v 0.1: changed name to euclidfastnn.cpp
// v 0.2: tidied and error checks commented out
// v 0.2.1 : minor additional comments
// ------------------------------------------
// Status: this code has been sucessfully compiled and MEX'd on
// both Linux and Windows platforms.

// This code makes use of the Approximate Near neighbour (ANN) C++ library
// written by David M. Mount and Sunil Arya
// please visit http://www.cs.umd.edu/~mount/ANN/.

#include "ANN.h"
#include "mex.h"


/* If you are using a compiler that equates NaN to zero, you must
 * compile this example using the flag -DNAN_EQUALS_ZERO. For
 * example:
 *
 *     mex -DNAN_EQUALS_ZERO findnz.c
 *
 * This will correctly define the IsNonZero macro for your
   compiler. */
```

```
#if NAN_EQUALS_ZERO
#define IsNonZero(d) ((d) != 0.0 || mxIsNan(d))
#else
#define IsNonZero(d) ((d) != 0.0)
#endif

// use standard namespace to avoid typing 'std::'
using namespace std;



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

      // declare variables
      unsigned int      num_elts;
      unsigned int      M_rows;
      unsigned int      N_cols;
      unsigned int      num_nbrs;
      double            error_bound;
      double            *input0, *input1, *output0, *output1;

    // Error checks ------------------------------------------------------
    // These error checks are redundant: errors caught in gammastat.m
    // uncomment these error checks only if this mexfunction is used alone.
    // -------------------------------------------------------------------
    // check for correct number of arguments
    //if (nrhs != 3) {
      //    mexErrMsgTxt("Error in mex file FASTNN:\nFirst input: data_points
(matrix of type double)\nSecond input: number_of_near_neighbours (int)\nThird
input: error_bound (double)");
    //}
      // else if (nlhs > 2) {
      //    mexErrMsgTxt("First output: nn_index_array (matrix of type
int)\nSecond output: nn_distance_array (matrix of type double)");
    //}
    //
      // check that first input is (matrix) of type double
      //if(!mxIsDouble(prhs[0])) {
      //    mexErrMsgTxt("data_points must be a matrix of type double");
      //}
    //
      // check that second input is a scalar -- ss- not sure what this does
(smth to check its scalar)
      //if (!mxIsDouble(prhs[1]) || mxIsComplex(prhs[1]) ||
mxGetN(prhs[1])*mxGetM(prhs[1]) != 1) {
      //    mexErrMsgTxt("number_of_near_neighbours must be a real scalar.");
      //}
    //
      // check that third input is a scalar
      //if (!mxIsDouble(prhs[2]) || mxIsComplex(prhs[2]) ||
mxGetN(prhs[2])*mxGetM(prhs[2]) != 1) {
      //    mexErrMsgTxt("error_bound must be a real scalar.");
      //}
      //mexPrintf("made it passed error checks!\n");

      // set parameters ----------------------------------------------------
-
      num_elts = mxGetNumberOfElements(prhs[0]);
```

```
      M_rows = (int)mxGetM(prhs[0]);
      N_cols = (int)mxGetN(prhs[0]);
      num_nbrs = (int)(mxGetScalar(prhs[1]));
      error_bound = (double)(mxGetScalar(prhs[2]));
      //mexPrintf("parameters read into mex func\n");
      //-----------------------------------------------------------------
--

    // declare ANN variables/data types
      ANNpointArray     pt_array;               // data points
      ANNidxArray       nn_index_list;               // nn index list
      ANNdistArray      nn_distance_list;       // nn distance list
      ANNkd_tree        *the_tree;                  // kd-tree

      // memory allocation for ANN objects
      pt_array          = annAllocPts(M_rows,N_cols);
      nn_index_list           = new ANNidx[num_nbrs+1];
      nn_distance_list  = new ANNdist[num_nbrs+1];

      //mexPrintf("ANN objects created\n");

      input0 = (double*)mxGetPr(prhs[0]);       // set pointer to input


      // create ANN pt_array object: read input0 (mxArray format--> ANN
format)
      for(int i=0;i<M_rows;i++) {
            ANNpoint pt = new ANNcoord[N_cols];

            for(int j=0;j<N_cols;j++) {
                  pt[j] = input0[i+(M_rows*j)];       //put data into
ANNpoint
            }

      pt_array[i] = pt;
      }



      // build the ANN  kd-tree using the ANN variables
      the_tree = new ANNkd_tree(
            pt_array,   // data points (an array of ANNpoints)
            M_rows,     // number of data points (M)
            N_cols);    // dimension of data (m)

      //mexPrintf("kd_tree built");


      // ---------------------------------------
      // output the near neighbour arrays.. to return to MATLAB workspace..
      plhs[0] = mxCreateDoubleMatrix(M_rows, num_nbrs, mxREAL);
      plhs[1] = mxCreateDoubleMatrix(M_rows, num_nbrs, mxREAL);
      output0 = (double*)mxGetPr(plhs[0]);
      output1 = (double*)mxGetPr(plhs[1]);
      // ---------------------------------------

      // loop over data points
      for(int i=0;i<M_rows;i++) {
```

```
            // search kd-tree
            the_tree->annkSearch(
            pt_array[i],            // query point
            num_nbrs,           // number of NNs (+1 here if NN of itself is
NOT
                                    // ignored -see ANN.h)
                              // ***************************
            nn_index_list,          // near neighbour index list (returned)
            nn_distance_list, // near neighbour distance list (returned)
            error_bound);           // error bound


            // insert indices/distances back into MATLAB mxArray format
            for(int k=0; k<num_nbrs; k++) {
                                    // ************************
                  output0[i +M_rows*k] = nn_index_list[k]+1;      // +1
because
                                          //matlab counts from 1 (not zero)
                  output1[i +M_rows*k] = nn_distance_list[k];
                              // use [k+1] if ANN.h flag is set to true.
                                // ..i,e. if NN of itself is not ignored,
then
                              //[k+1] needed here.
            }
      }


      // some memory management!
      delete pt_array;
      delete nn_index_list;
      delete nn_distance_list;
      delete the_tree;

    // finito
      return;
}
```

gammastat.m

```
% The Gamma test (core) calls fastnn.m and gammatest.m
%================================================
% Author: Sunil Singh
% Date: 27/9/06 (final package for web)
% version: 1.4
% For updates on Gamma test related software and papers see:
% http://users.cs.cf.ac.uk/Antonia.J.Jones/GammaArchive/IndexPage.htm
%================================================
% This script calls various other scripts and matlab executables: including:
% fastnn.m, gammatest.m, euclidfastnn.mex, manfastnn.mex, maxfastnn.mex
% (or .dll for older MATLAB versions)
%================================================
% This code makes use of the Approximate Near neighbour (ANN) C++ library
% written by David M. Mount and Sunil Arya
% please visit http://www.cs.umd.edu/~mount/ANN/.


% History
% version 0.0 : Core Gamma test algorithm
% version 0.1 : with figures and regression line
%               also returns gradient (optional)
% version 1.0 : calls fastnn.m automatically: passes arguments (num_nbr, eps)
%               .. directly from here.. more error checks
%               friendly usage message when no arguments are supplied
% version 1.1 : using nn distance matrix from fastnn.m
% version 1.2 : argument to choose norm, default set to 'euclid' (as a
%               string). Error checks included here to remove checks done in
MEX
%               Better modular design: error checks all in one place.
% version 1.3 : gammatest algorithm put into separate function that is
%               called from here. Note gammatest.m does not have error
%               checks
% version 1.4 : argument for graph plot, so this can be used with mtest.m
%             : scatter plot added to gammatest.m (as option)


function [GT_stat slope] = gammastat(x,y,graph,num_nn,err_bound,norm)


% inputs:   x                 input of dataset
%           y                 output of dataset
%           graph             can be either: 'regression', 'scatter' or
'none'
%           num_nn            Number of near-neighbours
%           err_bound         error bound for kd-tree (of ANN)
%           norm              metric: 'euclidnorm', 'mannorm' or 'maxnorm'
% outputs:  GT_stat            Gamma statistic
%           slope             slope of regression


% ------------------------------------------------------------------------
% more user friendly error messages: to check correct no. of inputs/ give
usage
% USAGE MESSAGE
if nargin < 1
      fprintf('\n\nUsage:\n\n');
```

```
      fprintf('[GT_stat
slope]=gammastat(input,output,''regression''*+,num_neighbours*,err_bound*,''e
uclidnorm''**)\n\n');
      fprintf('* = optional, but arguments must be in this order\n\n');
    fprintf('*+ = optional other plot: ''scatter'' or ''none'', default is
''regression'' \n\n');
    fprintf('** = optional other norms:\n''mannorm'' for Manhatten norm\n');
    fprintf('or ''maxnorm''for Max norm\n');
    fprintf('note these must be in quotes\n');
    fprintf('default is set to ''euclidnorm''for Euclidean norm\n\n');
      return;
end

% check minimum of 2 arguments ------------------------
if nargin < 1 | nargin <2
            fprintf('Error: at least two arguments must be supplied\n');
            GT_stat = [];
      return;
end



% ------------------------------------------------------------

% set default parameters
if nargin < 3
    graph = 'regression';
end
if nargin < 4
      num_nn = 10;
end
if nargin < 5
    err_bound = 0.0;
end
if nargin < 6
    norm = 'euclidnorm';
end

% ------------------------------------------------------------
if ~isnumeric(x)
    fprintf('Error: First argument must be matrix of type double\n');
    GT_stat = [];
    slope = [];
    return;
end

[M2 n] = size(y);

if ~isnumeric(y) | n>1
    fprintf('Error: Second argument must be column vector of type double\n');
    GT_stat = [];
    slope = [];
    return;
end



% params/ error checks -----------------------------------------------
[M m]  = size(x);
```

```
[M2 n] = size(y);
%[M3,p] = size(nnMatrix);
if M ~= M2
    fprintf('Error: input and output  must have the same number of rows\n');
    GT_stat = [];
    slope = [];
    return;
end


% ------------------------------------------------------------
% after if defaults are set, check num_nn is whole number integer
num_nn_int = cast(num_nn, 'int32');
if ~(num_nn_int == num_nn) | ~isnumeric(num_nn) | ~isscalar(num_nn) |
num_nn<=0
    fprintf('Error: Number of near neighbours must be a positive integer');
    GT_stat = [];
    slope = [];
    return;
end

%-----------------------------------------------
if ~isnumeric(err_bound) | err_bound<0 | ~isscalar(err_bound)
    fprintf('Error: Error bound must be a numeric value greater than or equal
to zero');
    GT_stat = [];
    slope = [];
    return;
end

% -----------------------------------------------
if M<=num_nn
      fprintf('Error: Number of data points supplied must be greater\nthan
the number near nearbours specified: if near neighbours \nis not specified,
default is set to 10\n');
      GT_stat = [];
    slope = [];
      return;
end



% call fastnn ----------------------------------------------------

[nniMatrix nndMatrix]=fastnn(x,num_nn,err_bound,norm);
% exit if call to fastnn fails for some reason
if isequal(nniMatrix,[])
      GT_stat = [];
    slope = [];
    return;
end

% Call Core Gamma test algorithm ---------------------------------

[GT_stat slope] = gammatest(y, nniMatrix, nndMatrix, graph);
```

fastnn.m

```
% script to call fast nearest neighbour algorithm
%=================================================
% Author: Sunil Singh
% Date: 27/9/06 (final package for web)
% version: 1.3
% For updates on Gamma test related software and papers see:
% http://users.cs.cf.ac.uk/Antonia.J.Jones/GammaArchive/IndexPage.htm
%=================================================
% This code makes use of the Approximate Near neighbour (ANN) C++ library
% written by David M. Mount and Sunil Arya
% please visit http://www.cs.umd.edu/~mount/ANN/.

% history:
% v.0.0: basic (this script was created to replace  getNN.m)
% v.0.1: error checks to stop MATLAB crashing if num_nn > num_data_points
% v.1.0: chooses norm.
% v.1.1: same as v 1.0, just tidied up. commented out error checks and
defaults.
% all error checks and defaults done from calling script (i.e. gammastat.m)
% v.1.2: more tidying: variable names changed to coicide with gammastat.m
% v. 1.3: norm selection

function [nniMatrix,nndMatrix] = fastnn(x,num_nn,err_bound, norm)
% inputs    : x         - input data matrix(Mxm)(one point on each row)
%           : num_nn    - number of near neighbours (p) to compute(default =
10)
%           : err_bound  - error bound for approx searching (default = 0)
%           : norm         - either Euclidean, Max or Manhatten
% outputs   : nniMatrix     - (Mxp) Matrix of near neighbour indices
%           : nndMatrix     - (Mxp) Matrix near neighbour distances


% friendly usage message ----------------------------------

if nargin < 1
      fprintf('\n\nUsage:\n\n');
      fprintf('[nni nnd] =
      fastnn(input,num_neighbours*,err_bound*,''euclidnorm''**)\n\n');
      fprintf('* = optional, but arguments must be in this order\n\n');
      fprintf('** = optional other norms:\n''mannorm'' for Manhatten
norm\n');
      fprintf('or ''maxnorm''for Max norm\n');
      fprintf('note these must be in quotes\n');
      fprintf('default is set to ''euclidnorm''for Euclidean norm\n\n');
      return;
end


% set default parameters and error checks ----------------------------------
---

if nargin < 2
      num_nn = 10;
end
if nargin < 3
    err_bound = 0.0;
```

```
end
if nargin < 4
    norm = 'euclidnorm';
end


% error checking here! To avoid MATLAB crashing, no. of data points must be
greater
% .. than the number of neighbours. Uncomment this if you used this script
% as a standalone function
%[M n] = size(data_pts);
%if M<=num_nn
%     fprintf('Error: Number of data points supplied must be greater\n');
%   fprintf('than the number near nearbours specified: if near neighbours');
%   fprintf('\nis not specified, default is set to 10\n');
%     nniMatrix =[];
%     nndMatrix=[];
%     return;
%end
% ---------------------- END error checks --------------------------------


% call the mex functions ----------------------------

if strcmp(norm,'euclidnorm')
    [nniMatrix,nndMatrix] = euclidfastnn(x,num_nn,err_bound);
    %fprintf('\nEuclidean norm (squared) calculated\n');
elseif strcmp(norm,'mannorm')
     [nniMatrix,nndMatrix] = manfastnn(x,num_nn,err_bound);
        nndMatrix = nndMatrix.*nndMatrix;         % needed because Gamma test
should take 'squared' metric
        %fprintf('\nManhatten norm (squared) calculated\n');
elseif strcmp(norm,'maxnorm')
    [nniMatrix,nndMatrix] = manfastnn(x,num_nn,err_bound);
        nndMatrix = nndMatrix.*nndMatrix;         % needed because Gamma test
should take 'squared' metric
        %fprintf('\nMax norm (squared) calculated\n');

% ----------- catch error ---------------------------

else fprintf('Error: norm input argument has been mispelt \n');
    fprintf('use either: ''euclidnorm'', ''maxnorm'' or ''mannorm'' \n');
    fprintf('or leave blank for default of euclidnorm\n');
    nniMatrix =[];
      nndMatrix=[];
    return;
end
```

gammatest.m

```
% Gamma test algorithm (core) for use with gammastat.m
% plots figures of linear regression or scatter.
%=================================================
% Author: Sunil Singh
% Date: 27/9/06 (final package for web)
% version: 0.0
% For updates on Gamma test related software and papers see:
% http://users.cs.cf.ac.uk/Antonia.J.Jones/GammaArchive/IndexPage.htm
%=================================================
% note: no error checks are done here; all error checks are done
% by calling script gammastat.m, thus users using this script alone must
% take care to do their own error checks!
% This code makes use of the Approximate Near neighbour (ANN) C++ library
% written by David M. Mount and Sunil Arya
% please visit http://www.cs.umd.edu/~mount/ANN/.



function [GT_stat slope]= gammatest(y, nniMatrix, nndMatrix, graph)

% inputs:   y                 output of dataset
%           nniMatrix         NN indexes
%           nndMatrix         NN distances
%           graph             a flag for regression plot or scatter plot
% outputs:  GT_stat            Gamma statistic
%           slope             slope of regression

% friendly usage message ---------------------------------
if nargin < 1
      fprintf('\n\nUsage:\n\n');
      fprintf('[GT_stat slope] =
gammatest(output,NNIndexMatrix,NNdistanceMatrix, ''regression''*)\n\n');

    fprintf('* = optional other plot: ''scatter'' or ''none'', default is
''regression'' \n\n');
    fprintf('* note this must be in quotes\n\n');
      return;
end

% default settings ----------------------------------------

if nargin < 4
    graph = 'regression';
end

% ---------------------------- core Gamma test algorithm
[M p] = size(nniMatrix);

gamma = zeros(M,p);
delta = zeros(M,p);
for i=1:M
    for k=1:p
        gamma(i,k) = 0.5*(y(i) - y(nniMatrix(i,k)))^2;
        %delta(i,k) = euclideanNorm(x(i,:) - x(nniMatrix(i,k),:))^2;
    end
end
```

```
Gamma = mean(gamma);
%Delta = mean(delta);
Delta = mean(nndMatrix);

% call MATLAB's linear regression -----------------------
co = polyfit(Delta,Gamma,1);


% return linear regression params-------------------------
rline = polyfit(Delta,Gamma,1);
GT_stat = rline(2);                   % returned to screen if u want
slope = rline(1);                   % slope: return to screen if u want

% graph plots -------------------------------------------

if strcmp(graph,'scatter')
    figure(2)
    clf
    plot(nndMatrix,gamma,'b.');
    hold
    plot(Delta,Gamma,'k*');
    maxd = max(max(nndMatrix));  % need max delta value to plot regression
    plot([0,maxd],[GT_stat,slope*maxd+GT_stat],'r');
    %plot([0,Delta(p)],[GT_stat,slope*Delta(p)+GT_stat],'r');
    xlabel('\Delta(k)');
    ylabel('\Gamma(k)');
    hold off
elseif strcmp(graph,'regression')
    figure(2)                 % figure 2
    clf
    hold on
    plot(Delta,Gamma,'*');
    plot([0,Delta(p)],[GT_stat,slope*Delta(p)+GT_stat],'r');
    xlabel('\Delta(k)');
    ylabel('\Gamma(k)');
end
```

demo.m

```
% demo.m
% Used to demonstrate use of the Gamma test, with dataset Sin500.txt
% or with artificially generated data (with normal distribution
% of noise). Comment 'in or out' lines as you wish!

%=================================================
% Author: Sunil Singh
% Date: 27/9/06 (final package for web)
% version:  0.0
% For updates on Gamma test related software and papers see:
% http://users.cs.cf.ac.uk/Antonia.J.Jones/GammaArchive/IndexPage.htm
%=================================================

format long     % MATLAB precision

% 1. Load data in ----------------------------------------------
load Sin500.txt;
x= Sin500(:,1);
y = Sin500(:,2);

% or can generate data artificially: ------------------------

%M=5000;         % datapoints
%m=1;            % dimensions
%VAR = 0.075;    % variance
%[x y]=noisy_sin_func(M,m,VAR);

% 2. plot -----------------------------------------------------
figure(1);
plot(x,y,'+')
xlabel('x');
ylabel('y');


% 3. call gammastat.m -----------------------------------------
% uncomment or comment in as you wish (only call once though!):
[GT_value slope] = gammastat(x,y)
% different norms: comment in or out as u like:
%[GT_value slope] = gammastat(x,y,10,0,'euclidnorm')
%[GT_value slope] = gammastat(x,y,10,0,'mannorm')
%[GT_value slope] = gammastat(x,y,10,0,'maxnorm')
% OR as separate modules --------------------------------
%[nni nnd] = fastnn(x);
%[gt_val] = gammatest(y,nni,nnd);
```

mtest.m

```
%=================================================
% Author: Sunil Singh and Samuel Kemp
% Date: 27/9/06
% For updates on Gamma test related software and papers see:
% http://users.cs.cf.ac.uk/Antonia.J.Jones/GammaArchive/IndexPage.htm
%=================================================
%M test with Heuristic Confidence Intervals
%=================================================
%Heuristic confidence intervals for the Gamma test.
%Antonia J. Jones and Samuel E. Kemp.
%The 2006 International Conference on Artificial Intelligence (ICAI'06):
%June 26-29, 2006, Las Vegas, USA.
%=================================================
%%%%%%%%%%%%%%%%COMMENTS %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%NEEDS function gammastat which uses the ANN compiled C package
%=================================================
% Notes about this script:
% If no arguments are supplied, defaults are set as follows:
% input and output are loaded from sin500.txt
% number of near neighbours, num_NN = 10
% start = num_NN+1
% step = 10 (step for M)
% confidence, cl = 0.9 (i.e. 90%)

%status: axis settings are MATLABS defaults



function [] = mtest (x,y,p,start,step, cl)

% Usage description ------------------------------------------

if nargin < 1
    fprintf('\n\nUsage:\n\n');
      fprintf('mtest (input*,output*,num_NN*,start*,step*,
confidence*)\n\n');
      fprintf('* = optional, but arguments must be in this order\n\n');
    fprintf('defaults set:\n\t input and output are read from Sin500.txt\n');
    fprintf('\t num_NN set to 10\n');
    fprintf('\t start set to num_NN+1, i.e 11 \n');
    fprintf('\t step set to 10\n');
    fprintf('\t confidence set to 0.9, i.e. 90%% \n\n\n');

end


%%%%%%%%%%%Set defaults etc.%%%%%%%%%%%%%%%%%%%%%

if nargin == 1
    fprintf('data pair must be provided');
    return;
end

if  nargin <2
    load Sin500.txt;
    x= Sin500(:,1);
```

```
    y = Sin500(:,2);
end

if nargin<3
    p = 10;
end

if nargin<4
    start =p+1;
end
if nargin <5
    step =10;
end
if nargin<6
    cl =0.9;        % 90% confidence level
end

%========================== DATA COMMENTS ==============================
%%%%Notes on data file format%%%%%
% 1. data consists of real numbers only.
% 2. Each row of real numbers is separated by spaces. Later versions of
% MatLab can tolerate commas as separators.
% 3. The last entry in a row is the output corresponding to the vector input
% formed by the other entries in the row.
% The first column represnts the first input variable etc.
% The last column represents the output variable.
% 4. The number of rows will typically lie between 100 and 10,000
% More than 50,000 rows will produce long run times.
%=======================================================================

%====================LOAD DATA=========================================
load('Sin500.txt') %Supplied test file


%%%%%%Do Mtest ----------------------------------------------------------
graph = 'none';  % to suppress Gamma regression plot

Mtotal = size(x);  % The length of the data.

index = 1;                % to index mlist
for M=start:step:Mtotal   %-(start)
    mlistg(index) = gammastat(x(1:M,:),y(1:M,:),graph,p);
    mlistm(index) = M;
    index = index+1;
end

[n samples] = size(mlistg);

%------------------Calculate heuristic confidence intervals--------

index=1;
for L=3:1:samples
    testmean = sum( sqrt(mlistm(1:L))*mlistg(1:L)')/(sqrt(mlistm(L)) * L) ;
    temp =  ( (sqrt(mlistm(1:L)).*mlistg(1:L))  - (sqrt(mlistm(1:L)) *
testmean) );
    sd = sqrt(temp*temp' / (mlistm(L)*(L-1)));
    se = sd/sqrt(L);
    alpha = 1- cl;
    LCI(index) = mlistg(L) - se * tinv(1-(alpha/2), (L-1));
```

```
    HCI(index) = mlistg(L) + se * tinv(1-(alpha/2), (L-1));
    index = index+1;
end

%----------------Plot Mtest -------------------------

figure(3)
plot(mlistm(1:samples),mlistg(1:samples),'r') % plot starts at 3 for
consistency with confidence intervals
hold

%---------------Plot heuristic confidence intervals---------
for h=1:1:samples-2
    plot([mlistm(h+2), mlistm(h+2)],[LCI(h), HCI(h)]); % NB mlists are a bit
larger than list of confidences
end

plot([0,mlistm(samples)],[0.075,0.075],':')
xlabel('M');
ylabel('Gamma');

% may want to leave this out. axis setting specific to this data and not
% users arbitrary data
%axis([ 0 mlistm(samples) 0.06 0.12])
hold off
```

noisy_sin_func.m

```
% creates a 'noisy' sin function between 0 and 2pi
% (uses normally distributed noise)
% Author: Sunil Singh
% Date: 27/9/06 (final package for web)
% version 1.1

% history:
% version 1.0: initial
% version 1.1: removed m dimension argument.

% Inputs:   M datapoints
%           VAR (variance)
% Output:   x (input)
%           y (output)


function [x y]=noisy_sin_func(M,VAR)

m =1;   % preset to 1 dimensional 'x' input only.
x=2*pi*rand(M,m);
fx = sin(x);
% VAR = 0.075;
r = sqrt(VAR)*randn(M,m);
y = fx + r;
```

Sin500.txt (calibration data)

Quoted from http://users.cs.cf.ac.uk/Antonia.J.Jones/GammaArchive/IndexPage.htm:

"A sine curve with artificially added uniformly distributed noise having a variance of 0.075. This is the classic example used in many papers and theses. Gamma = 0.0733545595048562, Gradient = 0.711221348889957, Standard Error = 0.00376506542836251, V-Ratio = 0.127617177846676, Near Neighbours = 10, Start Vector = 1, Unique Points = 500, Evaluated Output = 1, Zeroth Nearest Neighbours = 0, Lower 95% Confidence = - ,Upper 95% Confidence = -, Mask = 1. Where there are *Zeroth nearest neighbours* (repeated input point, which may or may not have identical output values) in the test set, the pointwise variance is computed and returned as Lower 95% Confidence and Upper 95% Confidence. In this case there are no Zeroth nearest neighbours in the data set."

ReadMe file for release of this package

% version: 0.0
% Status: this is a very basic readme file
%
% For updates on Gamma test related software and papers see:
% http://users.cs.cf.ac.uk/Antonia.J.Jones/GammaArchive/IndexPage.htm

==========

MATLAB script files contained in this package are:
1. gammastat.m // Core Gamma test which automatically calls fast near-neighbour algorithm
2. fastnn.m // Call to Approximate near neighbour algorithm
3. gammatest.m // Core Gamma test
4. demo.m // To get users started with using this package!
5. mtest.m // M test with confidence intervals
6. noisy_sin_func.m // artificial generation of noisy sin data with normal distribution
7. Sin500.txt // some sample data (in two columns, space delimited)

MATLAB executables are:
1. euclidfastnn.dll
2. maxfastnn.dll
3. manfastnn.dll
4. euclidfastnn.mexw32*
5. maxfastnn.mexw32*
6. manfastnn.mexw32*

* note: for latest versions of MATLAB. .mexglx for Linux package.

Basic users need only be concerned with using gammastat.m
By typing gammastat at the command line, a basic usage description
will be given. (read demo.m to get you started if need be)

For more advanced users, who wish to understand the near-neighbour
calculations, you can use:
1. fastnn.m (to return near-neighbour distances and indexes)
2. gammatest.m (to run the Gamma test)

(type fastnn or gammatest on command line to get basic usage description)

Further quantification of the Gamma statistic can be analysed using
the M-test (with confidence intervals)
See http://users.cs.cf.ac.uk/Antonia.J.Jones/GammaArchive/IndexPage.htm
for further details

ANN.h (only sub-section of code shown here to illustrate parts modified to calculate different Minkowski metrics)

```
//----------------------------------------------------------------
--------
//   Use the following for the Euclidean norm
//----------------------------------------------------------------
--------
#define ANN_POW(v)              ((v)*(v))
#define ANN_ROOT(x)             sqrt(x)
#define ANN_SUM(x,y)            ((x) + (y))
#define ANN_DIFF(x,y)           ((y) - (x))

//----------------------------------------------------------------
--------
//   Use the following for the L_1 (Manhattan) norm
//----------------------------------------------------------------
--------
// #define ANN_POW(v)           fabs(v)
// #define ANN_ROOT(x)          (x)
// #define ANN_SUM(x,y)         ((x) + (y))
// #define ANN_DIFF(x,y) ((y) - (x))

//----------------------------------------------------------------
--------
//   Use the following for a general L_p norm
//----------------------------------------------------------------
--------
// #define ANN_POW(v)           pow(fabs(v),p)
// #define ANN_ROOT(x)          pow(fabs(x),1/p)
// #define ANN_SUM(x,y)         ((x) + (y))
// #define ANN_DIFF(x,y) ((y) - (x))

//----------------------------------------------------------------
--------
//   Use the following for the L_infinity (Max) norm
//----------------------------------------------------------------
--------
// #define ANN_POW(v)           fabs(v)
// #define ANN_ROOT(x)          (x)
// #define ANN_SUM(x,y)         ((x) > (y) ? (x) : (y))
// #define ANN_DIFF(x,y) (y)

//----------------------------------------------------------------
--------
```

A.3: Validation scripts

mybrute.m

```
% version: 0.0
% this script will find all NN for each point of the dataset..
% used to validate ANN algorithm
% Author: Sunil
% Date: 12/8/06
%-------------------------------------------------------
%x=data_pairs;                  % needed when used with experiment scripts
                          %(noisy_sin.m, getNN.m, estimateNoise.m, automate.m )
% x=rand(10,2)                  % asuumes x is already in workspace (from
                                %myfastnn.c/visualise.m)
% -------------------------------------------------------

[M N]=size(x);

total = 0;
d_col=1;                % index for distance results (needed to avoid zeros)

for m= 1:M

      query= x(m,:);            % gives first row (i.e. first point is query)
      %query(1) = query(1)+0.05;          % shifted query
      %query(2) = query(2)+0.04;
      for mm= 1:M
            if not(mm==m)           % need if statement to avoid subtracting
                                    %query that is same as datapoint
                                    % remove if statement if shifted query
used
                  for n= 1:N
                        xydist(n) = ( query(n) - x(mm,n) )*( query(n) - x(mm,n) );
                        total = xydist(n)+total;
                  end

            distance(m,d_col) = sqrt(total);
            d_col = d_col+1;
            end %if
      total=0;
      end

d_col=1;
end


% sort the distances to find NN's
distance_sort = sort(distance,2);         % sorts


%some plots...
%scatter(x(:,1),x(:,2),'+')
%hold
%scatter(x(1,1),x(1,2))
%scatter(x(2,1),x(2,2))
%hold off
```

check.m

```
% version: 0.0

% this script is used to validate ANN algorithm as ..
% .. compared with mybrute.m algorithm.
% so far assumes we are calculating 2NN's (as sent to myfastnn.c via
visualise.m)

% assumes current variables:
% nnd - output from myfastnn.c/visualise.m  (or getNN.m)
% distance_sort - output from mybrute.m

nndsqrt=sqrt(nnd);    % to get real distances for comparison with
mybrute.m script
distance_sort_10NN=distance_sort(:,1:10);   % to get 10 nearest
neighbours
%distance_sort_3NN=distance_sort(:,1:3);  % to get 3 NN etc..
%distance_sort_2NN_first2= distance_sort_2NN(1:2,:); % just NN
distances for first 2 query points!

% just comparing first two points (to coincide with visual plot which
only circles first 2 points)
%nndsqrt_first2 = nndsqrt(1:2,:);
%sub= nndsqrt_first2 - distance_sort_2NN_first2;

sub = nndsqrt - distance_sort_10NN;

[M N]=size(sub);

flag=0;
for n=1:M*N
      if (sub(n)>0.01 || sub(n) <-0.01)
      flag=flag+1;
      end %if
end

flag
```

visualise.m

```
% version : 0.1
% History of modifications:
% v0.1: to interface with automate.m script. i,e just takes data from these
% scripts, as oppose to generating its own! and plots it as figure(2). note
% that fastnn.mex has already been called in getNN.m thus assumes that nnd
% and nni variables exist in workspace already.


% to plot the random points as a scatter, i.e. to visualise where the NN
should be.........
% stick to 2 dimensions as this we are only doing 2D plots!!!!

% ----------------------------------------------------TEMP CHANGE
% with automate.m (getNN.m, noisy_sin.m,estimateNoise.m)

x=rand(100,2);          % TEMP CHANGE
%x=data_pairs           % TEMP CHANGE: needed for use with getNN.m,
noisy_sin.m and estimateNoise.m, automate.m
figure(2)               % TEMP CHANGE: so create extra plot to visualise NNs

% ----------------------------------------------------TEMP CHANGE

a=x(:,1);               % to split dimensions of x, so they can be scatter
plotted-
b=x(:,2);               %.. and used later to draw circles!
scatter(a,b,'+')
hold;

tic         % start matlab timer (to measure execution time) ----------
%%---------------------------------------------------TEMP COMMENT OUT
[nni nnd]=fastnn(x,2,0);      % stick to 2 NNs for now.. (but can change % %
                              - shouldn't affect rest of code
execution_time =toc

%-----------------------------------------------------
% plotting  circles around datapoints to visually see where the near
neoghbours should be..
% NB nnd are squared, so need to be sqaure rooted!!!
cpoints=0:pi/10:2*pi;         % generate some points to plot a rough circle..
xc= sqrt(nnd(1,1))*sin(cpoints) + a(1);   % ( a(1),b(1)   ) is position of
first query point. +0.1 coz of shift in x direction of query point
yc=sqrt(nnd(1,1))*cos(cpoints) +b(1);
%hold                         % hold the scatter figure
plot(xc,yc)                   % plot circle at 1st NN radius
xc= sqrt(nnd(1,2))*sin(cpoints) + a(1);
yc=sqrt(nnd(1,2))*cos(cpoints) +b(1);
plot(xc,yc)                   % plot circle at 2nd NN radius
%--------

xc= sqrt(nnd(2,1))*sin(cpoints) + a(2);
yc=sqrt(nnd(2,1))*cos(cpoints) +b(2);
plot(xc,yc)
xc= sqrt(nnd(2,2))*sin(cpoints) + a(2);
yc=sqrt(nnd(2,2))*cos(cpoints) +b(2);
plot(xc,yc)
% ------------------------------
```

```
%plot the NN points (in different colour/ as little circle points) - to see
if algorithm has picked good NNs
scatter( a(nni(1,1)),b(nni(1,1)) )
scatter( a(nni(1,2)),b(nni(1,2)) )
scatter( a(nni(2,1)),b(nni(2,1)) )
scatter( a(nni(2,2)),b(nni(2,2)) )

%axis([-0.1 1.1 -0.1 1.1])                      % to square up the axis so
                                    %circles do not look like elipses

% ----------------------------------------

% highlight query points
scatter(a(1),b(1),'s')        % highlighting query point (NB add shift to
a(1) and a(2) if your shifting the query point!)
scatter(a(2),b(2), 's')       % highlighting query point

% legend ---------------------------------------
legend
legend('','','','','','','','','','','')
[legend_h,object_h,plot_h,text_strings] = legend;     % this returns all the
                                                      %handle
plot_h(2:5)=[];                                    % to remove handles
legend(plot_h(1:7), 'General data point','1st nearest neighbour','2nd nearest
neighbour','1st nearest neighbour','2nd nearest neighbour','A query point','A
query point');

hold off
```

A.3: Performance test scripts

error_bound.m

```
% script to run performance experiment

% generate the data: with increasing dimensions
% for each iteration also increase error bound

%-------------------------------------------------------
for m=1:30

    x=randn(m,1000)';

tic
[nniEuc nndEuc] = euclidfastnn(x,10,0);
ex_time(m,1)=toc;

tic
[nniEuc nndEuc] = euclidfastnn(x,10,0.5);
ex_time(m,2)=toc;

tic
[nniEuc nndEuc] = euclidfastnn(x,10,1);
ex_time(m,3)=toc;

tic
[nniEuc nndEuc] = euclidfastnn(x,10,2);
ex_time(m,4)=toc;

tic
[nniEuc nndEuc] = euclidfastnn(x,10,3);
ex_time(m,5)=toc;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

figure(1)
plot(ex_time(:,1),'r-+');     % zero err bound
hold
plot(ex_time(:,2),'c.-');     % increasing err bound
plot(ex_time(:,3),'b--*');     % increasing err bound
plot(ex_time(:,4),'c.:');     % increasing err bound
plot(ex_time(:,5),'r:+');     % increasing err bound

 legend
legend('','','','','')
[legend_h,object_h,plot_h,text_strings] = legend;legend(plot_h(1:5),
'Error bound: 0.0','Error bound: 0.5','Error bound: 1.0', 'Error
bound: 2.0','Error bound: 3.0');
xlabel('Number of dimensions');
```

```
ylabel('Execution time (seconds)');

hold off


figure(2)
err_b=[0;0.5;1;2;3]
plot(err_b,ex_timet(:,1),'-bo',err_b,ex_timet(:,2),'-
b+',err_b,ex_timet(:,3),'-
b*',err_b,ex_timet(:,4),'b.:',err_b,ex_timet(:,5),'b.--')
hold
legend('1 dimension','2 dimensions', '3 dimensions','4 dimensions','5
dimensions');
xlabel('Error bound');
ylabel('Execution time (seconds)');
hold off
```

Metrics_and_dimension.m

```
% script to run performance experiment

% generate the data, dimension, for different metrics:


for m=1:30

    x=randn(m,1000)';

tic
[nniEuc nndEuc] = euclidfastnn(x,10,0);
ex_time2(m,1)=toc;

tic
[nniMan nndMan] = manfastnn(x,10,0);
nndManSq = nndMan.*nndMan;
ex_time2(m,2)=toc;

tic
[nniMax nndMax] = maxfastnn(x,10,0);
nndMaxSq = nndMax.*nndMax;
ex_time2(m,3)=toc;

end


%figure(1)
%plot(ex_time1(:,1),'r+');     % euclid
%hold
%plot(ex_time1(:,2),'c');     % man
%plot(ex_time1(:,3),'b');     % max
%hold off

figure(2)
plot(ex_time2(:,1),'r-+');     % euclid
hold
plot(ex_time2(:,2),'c.-');     % man
plot(ex_time2(:,3),'b-x');     % max
hold off


legend
legend('','','','')   % create empty legends
[legend_h,object_h,plot_h,text_strings] = legend;
legend(plot_h(1:3), 'Euclidean','Manhattan','Max');
xlabel('Number of dimensions');
ylabel('Execution time (seconds)');
```

metric_and_data.m

```
M=100;
m= 1;           % 1 dimensional
looped=0;
for loop=1:10

    clear x
    [x]= noisy_sin_func(M,m);

    tic
    [nniEuc nndEuc] = euclidfastnn(x,10,0);
    timeeuc(loop)= toc;
    clear nniEuc nndEuc

    tic
    [nniMan nndMan] = manfastnn(x,10,0);
    nndManSq = nndMan.*nndMan;
    timeman(loop) = toc;
    clear nniMan nndMan nndManSq

    tic
    [nniMax nndMax] = maxfastnn(x,10,0);
    nndMaxSq = nndMax.*nndMax;
    timemax(loop) = toc;
    clear nniMax nndMax nndMaxSq

% ------------ err bound times - not really used much yet
    tic
    [nni nnd] = euclidfastnn(x,10,loop);
    timeeuc_err(loop)= toc;
    clear nni nnd
    %--------------
    M=M*2;
    looped = looped +1          % return loop status to screen.
    Mkeep(loop)=M
end

save timemax timemax       % save filename variablename
save timeman timeman
save timeeuc timeeuc
save Mkeep Mkeep
save timeeuc_err timeeuc_err     % not really used yet

% call script to plot ----------------------
plot_for_write_up
```

metric_and_data_5D.m

```
M=100;
m= 5;           % 5 dimensional
looped=0;
for loop=1:10

    clear x
    [x]= noisy_sin_func(M,m);


    tic
    [nniEuc nndEuc] = euclidfastnn(x,10,0);
    timeeuc(loop)= toc;
    clear nniEuc nndEuc

    tic
    [nniMan nndMan] = manfastnn(x,10,0);
    nndManSq = nndMan.*nndMan;
    timeman(loop) = toc;
    clear nniMan nndMan nndManSq


    tic
    [nniMax nndMax] = maxfastnn(x,10,0);
    nndMaxSq = nndMax.*nndMax;
    timemax(loop) = toc;
    clear nniMax nndMax nndMaxSq

% ------------ err bound times - not really used much yet
    tic
    [nni nnd] = euclidfastnn(x,10,loop);
    timeeuc_err(loop)= toc;
    clear nni nnd
    %--------------
    M=M*2;
    looped = looped +1          % return loop status to screen.
    Mkeep(loop)=M
end

save timemax timemax       % save filename variablename
save timeman timeman
save timeeuc timeeuc
save Mkeep Mkeep
save timeeuc_err timeeuc_err     % not really used yet - need to do
proper experiment for err_bound

% call script to plot
plot_for_write_up
```

plot_for_write_up.m

```
load timeeuc
load timeman
load timemax

load MKeep

%Mkeep(15:16)=[];

figure(3)
plot(Mkeep(1:10),timeeuc(1:10),'r-+')
hold
plot(Mkeep(1:10),timeman(1:10),'c.-')
plot(Mkeep(1:10),timemax(1:10),'b-x')


hold off

legend
legend('','','','')   % create  empty legends
[legend_h,object_h,plot_h,text_strings] = legend;legend(plot_h(1:3),
'Euclidean','Manhattan','Max');
xlabel('Number of data points');
ylabel('Execution time (seconds)');
```

A.5: C++ Makefiles (for windows and Linux)

To automate the process of generating a file (or files), the *make* utility is used. This utility looks for a file called ‘makefile’ or ‘Makefile’. This makefile specifies all the other tools and files (source files and object files) needed to generate the final output file. The final output file (application) is usually an executable or a library. Generating an application typically entails compiling each source code file into its corresponding object code file with a compiler; then using a linker to join these object code files into executables or libraries. For the case of this project, the ANN library was created.

One main advantage of using makefiles is for time saving. If a modification is made to any source file of the application, only that source file will be re-compiled. This is especially useful in large scale projects where compiling some source files may take a long time. Dependencies between source files can be defined, thus changes to one source file may not affect other parts of the overall application.

Further information can be found in many books written about C and C++, for example see [19].

Below are the makefiles written by the author’s supervisor, and modified slightly for the source files of this particular ANN library (under Linux and windows).

`MakeFile` (windows)

```
#==============================================================================
#  makefile for ann library: windows version..
#
#==============================================================================
#
#  NOTE: only the Borland-supplied "make" command should be used.
#
#==============================================================================
#
# $(makro_name) allows access to real name/variable
# this file only creates a .lib

BASEDIR = "C:"
INCDIR = $(BASEDIR)\include
LIBDIR = $(BASEDIR)\lib

CPP         = bcc32
LINK        = ilink32
AR                = tlib

CCOPTS            = -w- -jb -j1 -Hc
CCLINKOPTS  = -lGn -tWM-
CPPFLAGS    = $(CCOPTS) $(CCLINKOPTS)

SOURCES = ANN.cpp brute.cpp kd_tree.cpp kd_util.cpp kd_split.cpp \
      kd_dump.cpp kd_search.cpp kd_pr_search.cpp kd_fix_rad_search.cpp \
      bd_tree.cpp bd_search.cpp bd_pr_search.cpp bd_fix_rad_search.cpp \
      perf.cpp

HEADERS = kd_tree.h kd_split.h kd_util.h kd_search.h \
      kd_pr_search.h kd_fix_rad_search.h perf.h pr_queue.h pr_queue_k.h

OBJECTS = $(SOURCES:.cpp=.obj)

LIBNAME = ann

all: $(LIBNAME)

$(LIBNAME): $(OBJECTS)
   $(AR) /u $(LIBNAME).lib $(OBJECTS)
   copy $(LIBNAME).lib $(LIBDIR)


.cpp.obj:
    $(CPP) -I$(INCDIR) $(CPPFLAGS) -c {$? }

clean:
   -@if exist *.obj del *.obj                >nul
   -@if exist *.exe del *.exe                >nul
   -@if exist *.lib del *.lib                >nul
   -@if exist *.tds del *.tds                >nul
```

`Makefile` (Linux)

```
# makefile for ANN library: Linux version
CC = g++
AR = ar
CCFLAGS = -c -Wno-deprecated
ARFLAGS = -rc
IDIRS =
LDIRS =
LOGFILE = ./logfile
INC_INSTALL = $(CPATH)
LIB_INSTALL = $(LD_LIBRARY_PATH)

SRC =       ANN.cc \
            bd_pr_search.cc \
            bd_search.cc \
            bd_tree.cc \
            kd_pr_search.cc \
            kd_search.cc \
            kd_split.cc \
            kd_tree.cc \
            kd_util.cc \
            perf.cc \
            kd_fix_rad_search.cc \
            kd_dump.cc \
            brute.cc \
            bd_fix_rad_search.cc

OBJ = $(SRC:.cc=.o)
TARGET = libann.a

all : $(TARGET)
      echo "The ann library has been updated."

$(TARGET) : $(OBJ)
      $(AR) $(ARFLAGS) $(TARGET) $(OBJ)
      ranlib $(TARGET)

%.o : %.cc %.h
      date >> logfile
      $(CC) $(CCFLAGS) $(IDIRS) $< -o $@ 2>> $(LOGFILE)

install : $(TARGET)
      cp $(TARGET) $(LIB_INSTALL)
      ranlib $(LIB_INSTALL)/$(TARGET)
      cp *.h $(INC_INSTALL)
      echo "The ann library has been installed."

clean :
      rm $(TARGET)
      rm -f $(OBJ)
```

A.6: Compiler settings for MEX

Linux and Windows mexopts.bat and mexopts.sh files. These have been included in the appendix for completeness, should anyone wish to know the exact compiler settings.

Mexopts.bat (Windows configuration settings generated automatically)

```
@echo off
rem BCC55FREEOPTS.BAT
rem
rem    Compile and link options used for building MEX-files
rem    with the Borland C compiler
rem
rem    Created automatically from bcc55opts.bat
rem
rem
************************************************************************
rem General parameters
rem
************************************************************************
set MATLAB=%MATLAB%
set BORLAND=C:\Borland\bcc55
set PATH=%BORLAND%\BIN;%MATLAB_BIN%;%PATH%
set INCLUDE=%BORLAND%\INCLUDE
set LIB=%BORLAND%\LIB
set PERL="%MATLAB%\sys\perl\win32\bin\perl.exe"

rem
************************************************************************
rem Compiler parameters
rem
************************************************************************
set COMPILER=bcc32
set COMPFLAGS=-c -3 -P- -w- -pc -a8 -I"%INCLUDE%" -DMATLAB_MEX_FILE
set OPTIMFLAGS=-O2 -DNDEBUG
set DEBUGFLAGS=-v
set NAME_OBJECT=-o

rem
************************************************************************
rem Library creation command
rem
************************************************************************
set PRELINK_CMDS1=copy
"%MATLAB%\extern\lib\win32\borland\%ENTRYPOINT%.def"
"%OUTDIR%%MEX_NAME%.def"

rem
************************************************************************
rem Linker parameters
rem
************************************************************************
```

```
set LIBLOC=%MATLAB%\extern\lib\win32\borland
set LINKER=%PERL% %MATLAB_BIN%\link_borland_mex.pl
set LINKFLAGS=-aa -c -Tpd -x -Gn -L\"%BORLAND%\"\lib\32bit -
L\"%BORLAND%\"\lib -L\"%LIBLOC%\" libmx.lib libmex.lib libmat.lib
c0d32.obj import32.lib cw32mt.lib "%OUTDIR%%MEX_NAME%.def"
set LINKOPTIMFLAGS=
set LINKDEBUGFLAGS=-v
set LINK_FILE=
set LINK_LIB=
set NAME_OUTPUT="%OUTDIR%%MEX_NAME%"%MEX_EXT%
set RSP_FILE_INDICATOR=@

rem
****************************************************************
rem Resource compiler parameters
rem
****************************************************************
set RC_COMPILER=brcc32 -w32 -D_NO_VCL -fomexversion.res
set RC_LINKER=

set POSTLINK_CMDS=del "%OUTDIR%%MEX_NAME%.def"
```

mexopts.sh (Compiler settings for Linux: minor change made to this file is in **bold**)

```
#
# mexopts.sh     Shell script for configuring MEX-file creation script,
#                mex.  These options were tested with the specified
compiler.
#
# usage:         Do not call this file directly; it is sourced by the
#                mex shell script.  Modify only if you don't like the
#                defaults after running mex.  No spaces are allowed
#                around the '=' in the variable assignment.
#
# Note: For the version of system compiler supported with this
release,
#       refer to Technical Note 1601 at:
#       http://www.mathworks.com/support/tech-notes/1600/1601.html
#
#
# SELECTION_TAGs occur in template option files and are used by MATLAB
# tools, such as mex and mbuild, to determine the purpose of the
contents
# of an option file. These tags are only interpreted when preceded by
'#'
# and followed by ':'.
#
#SELECTION_TAG_MEX_OPT: Template Options file for building MEX-files
via the system ANSI compiler
#
# Copyright 1984-2004 The MathWorks, Inc.
# $Revision: 1.78.4.9.28.1 $  $Date: 2006/02/02 01:45:38 $
#--------------------------------------------------------------------
-------
#
    TMW_ROOT="$MATLAB"
    MFLAGS=''
    if [ "$ENTRYPOINT" = "mexLibrary" ]; then
        MLIBS="-L$TMW_ROOT/bin/$Arch -lmx -lmex -lmat -lmwservices -
lut"
    else
        MLIBS="-L$TMW_ROOT/bin/$Arch -lmx -lmex -lmat"
    fi
    case "$Arch" in
        Undetermined)
#--------------------------------------------------------------------
-------
# Change this line if you need to specify the location of the MATLAB
# root directory.  The script needs to know where to find utility
# routines so that it can determine the architecture; therefore, this
# assignment needs to be done while the architecture is still
# undetermined.
#--------------------------------------------------------------------
-------
```

```
            MATLAB="$MATLAB"
            ;;
        glnx86)
#----------------------------------------------------------------------
-------
            RPATH="-Wl,-rpath-link,$TMW_ROOT/bin/$Arch"
            CC='gcc'
            CFLAGS='-fPIC -ansi -D_GNU_SOURCE -pthread -fexceptions -
m32'
            CLIBS="$RPATH $MLIBS -lm -lstdc++"
            COPTIMFLAGS='-O -DNDEBUG'
            CDEBUGFLAGS='-g'
#
            CXX='g++'
            CXXFLAGS='-fPIC -ansi -D_GNU_SOURCE -pthread '
            CXXLIBS="$RPATH $MLIBS -lm"
            CXXOPTIMFLAGS='-O -DNDEBUG'
            CXXDEBUGFLAGS='-g'
#
#           NOTE: g77 is not thread safe
            FC='g77'
            FFLAGS='-fPIC -fexceptions'
            FLIBS="$RPATH $MLIBS -lm -lstdc++"
            FOPTIMFLAGS='-O'
            FDEBUGFLAGS='-g'
#
            LD="$COMPILER"
            LDEXTENSION='.mexglx'
            LDFLAGS="-pthread -shared -m32 -Wl,--version-
script,$TMW_ROOT/extern/lib/$Arch/$MAPFILE"
            LDOPTIMFLAGS='-O'
            LDDEBUGFLAGS='-g'
#
            POSTLINK_CMDS=':'
#----------------------------------------------------------------------
-------
            ;;
        glnxa64)
#----------------------------------------------------------------------
-------
            RPATH="-Wl,-rpath-link,$TMW_ROOT/bin/$Arch"
            CC='gcc'
            CFLAGS='-fPIC -fno-omit-frame-pointer -ansi -D_GNU_SOURCE
-pthread -fexceptions'
            CLIBS="$RPATH $MLIBS -lm -lstdc++"
            COPTIMFLAGS='-O -DNDEBUG'
            CDEBUGFLAGS='-g'
#
            CXX='g++'
            CXXFLAGS='-fPIC -fno-omit-frame-pointer -ansi -
D_GNU_SOURCE -pthread '
            CXXLIBS="$RPATH $MLIBS -lm"
            CXXOPTIMFLAGS='-O -DNDEBUG'
```

```
            CXXDEBUGFLAGS='-g'
#
#           NOTE: g77 is not thread safe
            FC='g77'
            FFLAGS='-fPIC -fno-omit-frame-pointer -fexceptions'
            FLIBS="$RPATH $MLIBS -lm -lstdc++"
            FOPTIMFLAGS='-O'
            FDEBUGFLAGS='-g'
#
            LD="$COMPILER"
            LDEXTENSION='.mexa64'
            LDFLAGS="-pthread -shared -Wl,--version-
script,$TMW_ROOT/extern/lib/$Arch/$MAPFILE"
            LDOPTIMFLAGS='-O'
            LDDEBUGFLAGS='-g'
#
            POSTLINK_CMDS=':'
#-------------------------------------------------------------------
-------
            ;;
        sol2)
#-------------------------------------------------------------------
-------
            CC='cc'
            CFLAGS='-KPIC -dalign -xlibmieee -D__EXTENSIONS__ -
D_POSIX_C_SOURCE=199506L -mt'
            CFLAGS="$CFLAGS -D_XOPEN_SOURCE=600"
            CLIBS="$MLIBS -lm -lc"
            COPTIMFLAGS='-xO3 -xlibmil -DNDEBUG'
            CDEBUGFLAGS='-xs -g'
#
            CXX='CC -compat=5'
            CCV=`CC -V 2>&1`
            version=`expr "$CCV" : '.*\([0-9][0-9]*\)\.'`
            if [ "$version" = "4" ]; then
                    echo "SC5.0 or later C++ compiler is required"
            fi
            CXXFLAGS='-KPIC -dalign -xlibmieee -D__EXTENSIONS__ -
D_POSIX_C_SOURCE=199506L -mt'
            CXXLIBS="$MLIBS -lm -lCstd -lCrun"
            CXXOPTIMFLAGS='-xO3 -xlibmil -DNDEBUG'
            CXXDEBUGFLAGS='-xs -g'
#
            FC='f90'
            FFLAGS='-KPIC -dalign -mt'
            FLIBS="$MLIBS -lfui -lfsu -lsunmath -lm -lc"
            FOPTIMFLAGS='-O'
            FDEBUGFLAGS='-xs -g'
#
            LD="$COMPILER"
            LDEXTENSION='.mexsol'
            LDFLAGS="-G -mt -M$TMW_ROOT/extern/lib/$Arch/$MAPFILE"
            LDOPTIMFLAGS='-O'
```

```
            LDDEBUGFLAGS='-xs -g'
#
            POSTLINK_CMDS=':'
#-----------------------------------------------------------------------
-------
            ;;
        mac)
#-----------------------------------------------------------------------
-------
            CC='gcc-3.3'
            CFLAGS='-fno-common -no-cpp-precomp -fexceptions'
            CLIBS="$MLIBS -lstdc++"
            COPTIMFLAGS='-O3 -DNDEBUG'
            CDEBUGFLAGS='-g'
#
            CXX=g++-3.3
            CXXFLAGS='-fno-common -no-cpp-precomp -fexceptions'
            CXXLIBS="$MLIBS -lstdc++"
            CXXOPTIMFLAGS='-O3 -DNDEBUG'
            CXXDEBUGFLAGS='-g'
#
            FC='f77'
            FFLAGS='-f -N15 -N11 -s -Q51 -W'
            ABSOFTLIBDIR=`which $FC | sed -n -e '1s|bin/'$FC'|lib|p'`
            FLIBS="-L$ABSOFTLIBDIR -lfio -lf77math"
            FOPTIMFLAGS='-O -cpu:g4'
            FDEBUGFLAGS='-g'
#
            LD="$CC"
            LDEXTENSION='.mexmac'
            LDFLAGS="-bundle -Wl,-flat_namespace -undefined suppress -
Wl,-exported_symbols_list,$TMW_ROOT/extern/lib/$Arch/$MAPFILE"
            LDOPTIMFLAGS='-O'
            LDDEBUGFLAGS='-g'
#
            POSTLINK_CMDS=':'
#-----------------------------------------------------------------------
-------
            ;;
    esac
#######################################################################
#######
#
# Architecture independent lines:
#
#     Set and uncomment any lines which will apply to all
architectures.
#
#-----------------------------------------------------------------------
-------
# added here two lines below
     CC="$CXX"
     LD="$CXX"
```

```
#            CC="$CC"
#            CFLAGS="$CFLAGS"
#            COPTIMFLAGS="$COPTIMFLAGS"
#            CDEBUGFLAGS="$CDEBUGFLAGS"
#            CLIBS="$CLIBS"
#
#            FC="$FC"
#            FFLAGS="$FFLAGS"
#            FOPTIMFLAGS="$FOPTIMFLAGS"
#            FDEBUGFLAGS="$FDEBUGFLAGS"
#            FLIBS="$FLIBS"
#
#            LD="$LD"
#            LDFLAGS="$LDFLAGS"
#            LDOPTIMFLAGS="$LDOPTIMFLAGS"
#            LDDEBUGFLAGS="$LDDEBUGFLAGS"
#-----------------------------------------------------------------------
-------
##########################################################################
#######
```

Appendix B: User manual

1. Arrangement of input output data set

- Input data must be arranged as a *M* x *m* matrix in the MATLAB workspace. Where *M* is the number of rows (datapoints) and '*m*' is the number of columns (dimension of the data)
- Output data must have the same number of rows as the input data, and consist of only one column. If the user has a vector output, then separate runs of the Gamma test need to be run on each output individually.

2. Recommendation of data format (with numerical text files):

For 1 dimensional data:

In 'columns' where the first column is input, and second 'column' is the corresponding output.

Columns should be delimited by a space or tab, but later versions of MATLAB can tolerate commas, when using the 'load' command. E.g:

```
load Sin500.txt;          % data set available from ref [22]
x= Sin500(:,1);
y = Sin500(:,2);)
```

will place the first and second columns of data into the variables `x` and `y` respectively. The file `Sin500.txt` is delimited using commas, however, MATLAB can easily cope with other delimiters: e.g. tab. A range of loading options can be found in the MATLAB documentation should a user have a non-standard format. However it is recommended that the user has only numeric data within the file, delimited by either a comma or tab. Other delimiters should be fine, but users should refer to their MATLAB documentation.

For 2 dimensional data:

Similar to above, but easiest option is to divide input and output data into separate files. If you keep all data in one file, it is very easy to divide up the input output data into an *M* x *m* matrix for input and *M* x 1 matrix for the outputs using standard MATLAB commands.

As long as the data file format is consistent in some way, it is very easy to divide up, transpose and generally manipulate data in MATLAB to get it into the format that the `gammastat`.m script needs.

3. Tips

3.1 By typing

gammastat

at the MATLAB command line, a usage message will be displayed which details all inputs and outputs to the function, as well defaults set if an argument is optional. Similar usage messages are available for `fastnn, gammatest` and `mtest` scripts.

3.2 By typing

demo

at the command line. It is recommended (for beginners) to *read* the `demo.m` file, and alter lines within it to experiment with this package initially

4. Usages of functions within this package are as follows:

[GT_stat slope] = gammastat(x,y,graph*,num_nn*,err_bound*,norm*)

* = optional but arguments must be in this order

inputs:	x	input of dataset
	y	output of dataset
	graph	can be either: 'regression', 'scatter' or 'none', default = ‘regression’
	num_nn	Number of near-neighbours, default = 10
	err_bound	error bound for kd-tree (of ANN), default = 0.0
	norm	metric: 'euclidnorm', 'mannorm' or 'maxnorm', default = ‘euclidnorm’
outputs:	GT_stat	Gamma statistic
	slope	slope of regression

[nniMatrix,nndMatrix] = fastnn(x,num_nn*,err_bound*, norm*)

* = optional but arguments must be in this order

inputs	x	input data matrix (Mxm) (one point on each row)
	num_nn	number of near neighbours (p) to compute (default = 10)
	err_bound	error bound for approx searching (default = 0)
	norm	either Euclidean, Max or Manhatten, default = ‘euclidnorm’
outputs	nniMatrix	(Mxp) Matrix of near neighbour indices
	nndMatrix	(Mxp) Matrix near neighbour distances

[GT_stat slope]= gammatest(y, nniMatrix, nndMatrix, graph*)

* = optional but arguments must be in this order

inputs:	y	output of dataset
	nniMatrix	Near neighbour indexes
	nndMatrix	Near neighbour distances
	graph	can be either: 'regression', 'scatter' or 'none', default = ‘regression’
outputs:	GT_stat	Gamma statistic
	slope	slope of regression

mtest (input*,output*,num_NN*,start*,step*, confidence*)

* = optional, but arguments must be in this order

Inputs:	defaults set:
	input and output are read from Sin500.txt
	num_NN set to 10
	start set to num_NN+1, i.e 11
	step set to 10
	confidence set to 0.9, i.e. 90%
Output:	M-test plot with confidence intervals set accordingly
	Returning other numerical parameters has not presently been set.

-+

[x y]=noisy_sin_func(M,VAR)

Inputs:	M	Number of datapoints
	VAR	Variance of the noise
Outputs:	x	M data points randomly sampled between 0 and 2pi
	y	Corresponding noisy sin data

Appendix C: ANN

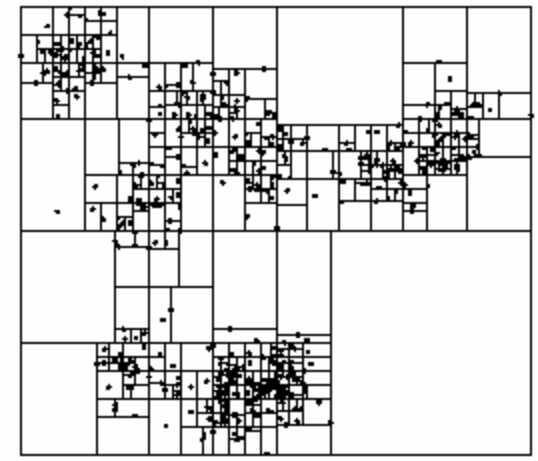

Figure C1: Sample output for a kd-tree produced by the ANN library [3].

ANN source files:

The following source files were downloaded from ANN website and used for this project (version 1.1 release date 05/03/05)

```
ANN.cpp, brute.cpp, kd_search.cpp, kd_util.cpp,
bd_fix_rad_search.cpp, kd_dump.cpp, kd_search.h, kd_util.h,
bd_pr_search.cpp, kd_fix_rad_search.cpp, kd_split.cpp,
Makefile, bd_search.cpp, kd_fix_rad_search.h, kd_split.h,
perf.cpp, bd_tree.cpp, kd_pr_search.cpp, kd_tree.cpp,
pr_queue.h, bd_tree.h, kd_pr_search.h, kd_tree.h,
pr_queue_k.h, ANN.h, ANNperf.h, ANNx.h.
```

References

[1] Antonia J. Jones and S. E. Kemp. *Heuristic confidence intervals for the Gamma test*. The 2006 International Conference on Artificial Intelligence (ICAI'06): June 26-29, 2006, Las Vegas, USA

[2] Antonia J. Jones, D Evans and S. E. Kemp. *A Note on the Gamma test Analysis of Noisy Input/Output data and Noisy Time Series*. Submitted paper

[3] Approximate near neighbour library, ANN Version 1.1 (Release date: 05/03/05). http://www.cs.umd.edu/~mount/ANN/

[4] Latest results from the EPICA Dome C ice core Press Release, CLIMATE AND ENVIRONMENTAL PHYSICS INSTITUTE, UNIVERSITY OF BERN, SWITZERLAND. See: http://www.climate.unibe.ch/press251105.pdf

[5] MATLAB Release Notes: External Interface/API Upgrade Issues (See MATLAB help files with MATLAB version 7.1.0.246 (R14) Service Pack 3)

[6] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbour searching. *J. ACM*, 45:891{923, 1998.

[7] J. L. Bentley. K-d trees for semi dynamic point sets. In *Proc. 6th Ann. ACM Symposium.*
Comput. Geom., pages 187{197, 1990.

[8] K. L. Clarkson. Nearest neighbour queries in metric spaces. In *Proc. 29th Annu. ACM Symposium. Theory Comput.*, pages 609{617, 1997.

[9] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimension. In *Proc. 29th Annu. ACM Sympos. Theory Comput.*, pages 599{608, 1997.

[10] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[11]Friedman, J. H., Bentley, J. L., and Finkel, R. A. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software 3*, 3, 209{226

[12] Sproull, R. L. 1991. Refinements to nearest-neighbor searching. *Algorithmica 6*, 579{589.

[13] S. Arya and D. M. Mount. Approximate nearest neighbor queries in _xed dimens In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271{280, 1993.

[14] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *J. ACM*, 45:891{923, 1998.

[15] http://www.cs.umd.edu/~mount/ANN/.
Approximate near neigbour library, ANN Version 1.1 (Release date: 05/03/05) manual.

[16] H. Samet. *Applications of spatial data structures*. Addison-Wesley, Reading MA, 1990

[17] *New Tools in Non-linear Modelling and Prediction*. Antonia J. Jones. Computational Management Science, 1(2):109-149, 2004.

[18] Nearest-Neighbor Methods in Learning and Vision: Theory and Practice MIT Press, March 2006. G. Shakhnarovich, T. Darrell, P. Indyk, eds.

[19] C: The Complete Reference, McGraw-Hill Professional, by Herbert Schildt. Publication Date: 26 Apr 2000

[20] http://gcc.gnu.org/

GCC, the GNU Compiler Collection

[21] Borlands C++ compiler 5.5 (‘command line tools’)
http://www.borland.com/

[22] http://users.cs.cf.ac.uk/Antonia.J.Jones/GammaArchive/IndexPage.htm

[23] Asymptotic moments of near neighbour distance distributions. D. Evans, Antonia J. Jones, W. M. Schmidt. Proc. Roy. Soc. Lond. Series A, 458(2028):2839-2849, 2002.

[24] *A proof of the Gamma test*. D. Evans and Antonia J. Jones. Proc. Roy. Soc. Series A, **458**(2027), 2759-2799, 2002.

[25] "Software Engineering 6 th Edition", by Ian Sommerville, Addison-Wesley 2001.

[26] Applied Multivariate Data Analysis. by G. (Graham) Dunn, Brian S. Everitt – 2001 Published by Oxford University Press US.

[26] Adalbjörn Stefánsson, N. Koncar and Antonia J. Jones. *A note on the Gamma test*, Neural Computing & Applications **5**(3):131-133, 1997.

[27] For information on MATLAB visit: http://www.mathworks.com/

[28] P. Indyk and R. Motwani. Approximate nearest neighbor: towards removing the curse of dimensionality. *Proceedings of the Symposium on Theory of Computing*, 1998.

[29] A noise estimation method for corrupted correlated data. *Statistical Methods and Applications.* Physica Verlag, An Imprint of Springer-Verlag Pages343-356 Online DateTuesday, December 06, 2005

[30] WAVELETS FOR KIDS. A Tutorial Introduction. By Brani Vidakovic and Peter Mueller Duke University. http://www2.isye.gatech.edu/~brani/wp/kidsA.pdf.

[31] Advanced Mathematical Approach to Biology. By Takeyuki Hida. Publisher: World Scientific. Publication Date: 1 Jun 1998

[32] Chaos: An Introduction to Dynamical Systems. Edited by Kathleen Alligood, Tim Sauer, J Yorke. Publisher: Springer. Publication Date:1 Jan 1996

[33] PhD thesis: *Gamma test analysis tools for non-linear time series***.** Samuel E. Kemp. Department of Computing & Mathematical Sciences, Faculty of Advanced Technology, University of Glamorgan, Wales UK, 2006

[34] PhD thesis: *winGamma*™: *a non-linear data analysis and modelling tool with applications to flood prediction.* P. Durrant. Department of Computer Science, Cardiff University, 2001.

[35] Statistics: An Introduction Using R. By Michael J. Crawley. Publisher: John Wiley and Sons. 6 May 2005

[36] Takens, F. (1981). Detecting strange attractors in turbulence. In Dynamical Systems and Turbulence, Volume 898 of Lecture notes in Mathematics, pp. 366{381. Springer-Verlag.