

Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

Multi-Step Look-Ahead Adaptive Designs for the Estimation of Sensory Thresholds

A thesis presented in partial fulfilment of the requirements
for the degree of

Master of Applied Statistics

at Massey University, Albany, New Zealand

Mark William Wohlers

Student ID # 10148537
2013

Abstract

The estimation of sensory thresholds is an important part of the psychophysics field. The point at which a physical stimulus becomes detectable can vary from trial to trial within as well as between subjects. Often the probability of detection is modelled over a range of stimulus intensities using an assumed psychometric curve which has the threshold as a parameter. To estimate the threshold with a reasonable accuracy often requires careful placement of the stimulus levels when the total number of trials are limited. There have been a number of design schemes proposed over the years to find the optimum placement strategy to minimise a given loss function. Some of the most successful have been Bayesian adaptive designs which select the next signal intensity based on prior knowledge and the responses observed up until that point. A critical step in the adaptive designs is the choice of threshold estimator and error term, also known as the loss function, to be minimised by the design scheme. A sub-class of these look-ahead a short number of trials to calculate the expected loss function given the current posterior distribution. However sometimes it is not possible to adjust the signal after every test. Olfactory sensory threshold tests, for example, can require a large setup time. In this situation a number of sensory tests may be grouped into sessions, with any design alterations occurring between these. However this would require a look-ahead design with a number of steps equal to the number of samples in a session.

Most of the look-ahead designs have been restricted to one or two steps due to the little performance increase gained by increasing them and the computational limitations at the time they were suggested. The first point is not relevant to situations where the step size must be larger, and the second point may be less true today due to advances in computer power. This investigation demonstrates that it is possible to implement multi-step look-ahead adaptive designs in a computationally efficient manner for sessions up to sizes of twelve samples. Based on Monte-Carlo simulations, these multi-step look-ahead designs also provide encouraging results in terms of performance in minimising a number of loss functions.

Acknowledgements

It is a pleasure to have the opportunity to thank the many people who have helped me along the way to completing this thesis. The New Zealand Institute for Plant and Food Research Ltd was kind enough to allow me to continue to work full-time while studying. Many of my co-workers also deserve a special mention. In particular I would like to thank Nihal De Silva and the Biometrics team for their continued encouragement to further my study and Sara Jaeger along with the rest of the sensory team for being instrumental in inspiring my interest in the area.

My supervisor Barry McDonald has my thanks for his guidance and invaluable suggestions while writing this.

Lastly I would like to give a special thanks to my wife Evelyn for her never ending support and encouragement throughout this long journey. To her I dedicate this thesis.

Contents

Table of Figures	vi
Table of Tables	vii
1 Background	1
2 Literature Review	3
2.1 Test Protocols: Yes-No and n-AFC Experiments.....	4
2.2 Methods for Estimating the Psychometric Curve	6
2.2.1 Method of Constant Stimuli.....	6
2.2.2 Method of Limits.....	7
2.2.3 Method of Adjustment	7
2.2.4 PEST.....	7
2.2.5 Staircase Procedures.....	8
2.2.6 Maximum Likelihood Adaptive Procedures	8
2.2.7 Bayesian Adaptive Procedures.....	9
2.2.8 ASTM method	15
2.3 Odour Detection Experiments at The New Zealand Institute for Plant and Food Research.....	16
2.4 Difficulties Implementing Standard Adaptive Procedures with Olfactory and Taste Threshold Estimation	17
2.5 Proposed Adaptive Threshold Estimation for Olfactory and Taste Experiments.	18
3 Methods.....	21
3.1 Software	21
3.1.1 Numpy and Scipy.....	22
3.1.2 Matplotlib.....	22
3.1.3 PyMC	22
3.1.4 Playdoh.....	22
3.1.5 Numdifftools	22
3.1.6 Numexpr	22
3.1.7 ffnet.....	23
3.1.8 OpenOpt.....	23
3.2 ASTM method	23
3.3 Threshold Estimation using the Psychometric Function.....	25

3.3.1	Maximum Likelihood.....	25
3.3.2	Bayesian Fitting.....	28
3.3.3	Approximating the Loss Function by Simulation	41
3.4	Loss Function Minimisation	41
3.4.1	Minimisation with Continuous Stimulus Levels	42
3.4.2	Minimisation with Discrete Stimulus Levels	43
3.4.3	The Adaptive Procedure	44
3.4.4	Python code	46
4	Results.....	49
4.1	Fitting Bayesian Models through MCMC.....	49
4.1.1	Generating data for Neural Network Training.....	49
4.1.2	Training the Neural network.....	50
4.1.3	Neural Network Performance.....	51
4.1.4	Optimizing based on the Neural Network	55
4.2	D-Optimal Designs	57
4.3	Bayesian Optimal Design using Discrete Priors.....	58
4.3.1	Comparison of Adaptive Schemes	59
4.3.2	Adaptive Schemes Under Misspecified Psychometric Curve.....	63
4.3.3	Comparing Look-Ahead Step Sizes: Minent8 vs. Minent1	68
4.4	ASTM Optimal Design	71
4.5	Discrete Signal Intensities	72
5	Discussion and Suggestions for Future research	74
6	Conclusion.....	79
7	Bibliography	81
Appendix A	R Computer Code.....	86
Appendix B	Python Computer Code.....	87

Table of Figures

Figure 1	An example of a Gumbel psychometric curve	3
Figure 2	A visual summary of common sensory test protocols	5
Figure 3	Parameterisation and forms of psychometric functions	11
Figure 4	Effect of parameterisation of the logistic psychometric curve with standard uniform prior distributions	13
Figure 5	Example of estimated detection probabilities for a psychometric curve at given signal intensities.....	24
Figure 6	Autocorrelation plots for Threshold parameter estimates.....	32
Figure 7	Example of a Neural Network with 3 input, 6 hidden and one target neuron.	35
Figure 8	Example of how the inputs (x) are linked to the outputs (y) via the weights (w) and activation function (g).....	36
Figure 9	Example of a log-sigmoid activation function with $\beta = 5$	36
Figure 10	Scatter plots of the MCMC estimates based on 8 observations vs. the neural network approximations.....	51
Figure 11	Scatter plots of the MCMC estimates based on 32 observations vs. the neural network approximations.....	53
Figure 12	Posterior mean NN approximations versus MCMC means based on 8, 16, 24, and 32 samples. 54	
Figure 13	Logistic Psychometric curves used to generate responses to assess the performance of the NN based adaptive method.....	56
Figure 14	Estimated RMSE for the NN Adaptive Design	57
Figure 15	Logistic Psychometric curves under discrete standard uniform priors	59
Figure 16	RMSE based on 100 samples generated by Logistic Psychometric curves	61
Figure 17	Mean Absolute Errors based on 100 samples generated by Logistic Psychometric curves	62
Figure 18	Categorical Errors based on 100 samples generated by Logistic Psychometric curves	63
Figure 19	Weibull Psychometric curves used to generate misspecified samples	64
Figure 20	RMSE based on 100 samples generated by Weibull Psychometric curves.....	65
Figure 21	Absolute errors based on 100 samples generated by Weibull Psychometric curves	66
Figure 22	Categorical errors based on 100 samples generated by Weibull Psychometric curves	67
Figure 23	Logistic Psychometric curves used to compare the MINENT1 and MINENT8	69
Figure 24	Comparison of the convergence of the MINENT8 (red) versus MINENT1 (black)..	70
Figure 25	Comparison of RMSE for the MINENT1 and MINENT8 procedures	70
Figure 26	ASTM Expected RMSE	72

Table of Tables

Table 1	Example of BET estimates for two panellists. Ticks indicate correct detection at a given concentration, and a cross an incorrect response.	16
Table 2	Variance, Bias, and MSE estimation for teh ASTM method.....	24
Table 3	Regression Summary Statistics for Neural Network Approximation Based on samples of Length 8	52
Table 4	Summary statistics from regressing the MCMC point estimates on the NN approximations based on 32 observations per sample.	53

1 Background

The investigation underpinning this document relates to a problem encountered during the planning stage of a large consumer trial conducted at the New Zealand Institute for Plant and Food Research (PFR).

This trial required a large number of individual sensory olfaction thresholds to be calculated from a limited number of trials. The testing method was decided upon as being the 3 Alternative Forced Choice (3-AFC) with thirty two such tests per subject. A single 3-AFC test involves 3 samples being presented to the individual with one of these containing the signal. The subject then samples each of the three in a set order and records which is believed to contain the signal.

The problem was to now make adjustments to data collection and handling as to ensure high quality threshold estimates were produced. One such adjustment is the choice of how to estimate the threshold given the data. While this is an important choice as certain estimators may be more efficient than others, and it will be discussed, it is not the focus of this investigation.

Another alteration that can be made is the selection of the stimulus intensities used at the testing stage. If all the tests are administered at concentrations reasonably greater than the threshold then the subjects will always (or nearly always if the subject lapses), detect the signal resulting in little information to base a reliable threshold estimate on. Conversely if the testing is carried out at undetectable levels then the responses are all random guesses and the same is true. While these are extreme examples as most subjects should see a range of intensities from non-detectable through to always detectable, they do demonstrate that the choice of signal intensities presented can have an impact on precision of the threshold estimates.

This problem is not new and is an active area of research. There have been numerous methods proposed such as general guidelines on where to set the levels as with the American Society for the Testing of Materials (ASTM) method through to fully adaptive methods which base the next test on the previous results. While some of these

adaptive methods could in theory work there are additional problems relating to the PFR experiment setup which means they are impractical. The main difficulty is setup time. With vision or hearing tests it can be relatively quick to increase or decrease the signal level for the next test. However in the case of the PFR experiments the olfactory tests required a carefully measured volume of the compound of interest to be diluted in water. This would take too much time between tests to decide upon the next concentration and getting the sample ready especially if multiple panellists are being tested at the same time. Using sniffing sticks (Hummel, Sekinger, Wolf, Pauli, & Kobal, 1997) could allow for rapid change in signal intensity, however as this method is generally conducted one on one it would be prohibitive in terms of staffing for the large study proposed.

Therefore the investigation should focus on methods which allow for changes in concentrations to be made between blocks of samples rather than after each individual test. For example the concentrations used for the following day could be based on the previous results. It may also be simpler for the experimenter to have the same concentrations presented to all panellists and even the same across days. These additional restrictions will be looked at also.

Taking the above limitations into account the problem is that given a set or distribution of psychometric curves and a threshold estimation method what is the optimal set of concentrations to test to minimise a given error measure? The relationship between the design and the expected error is often referred to as the loss function. One could try and minimise the loss function analytically or if that proves too difficult, obtain an approximate solution through simulation.

2 Literature Review

Relating the intensity level of a stimulus to a subject's ability to detect or discriminate is an important area of Psychophysics, which is itself the more general study of connecting physical stimuli to subjective responses (Kuss, Jäkel, & Wichmann, 2005). The subject's performance in detection or discrimination tasks should improve as the stimulus level intensifies. While it is possible that a subject may suddenly jump from no detection to perfect detection after a small increase in the stimulus level, often, however, it is a more subtle change-over. The probability of detection (or discrimination) increases in a sigmoid curve shape rather than a simple step function. This monotonic curve relating a subject's performance to a physical stimulus is commonly referred to as the psychometric function $F(x)$. An example of a Gumbel psychometric curve is shown in Figure 1.

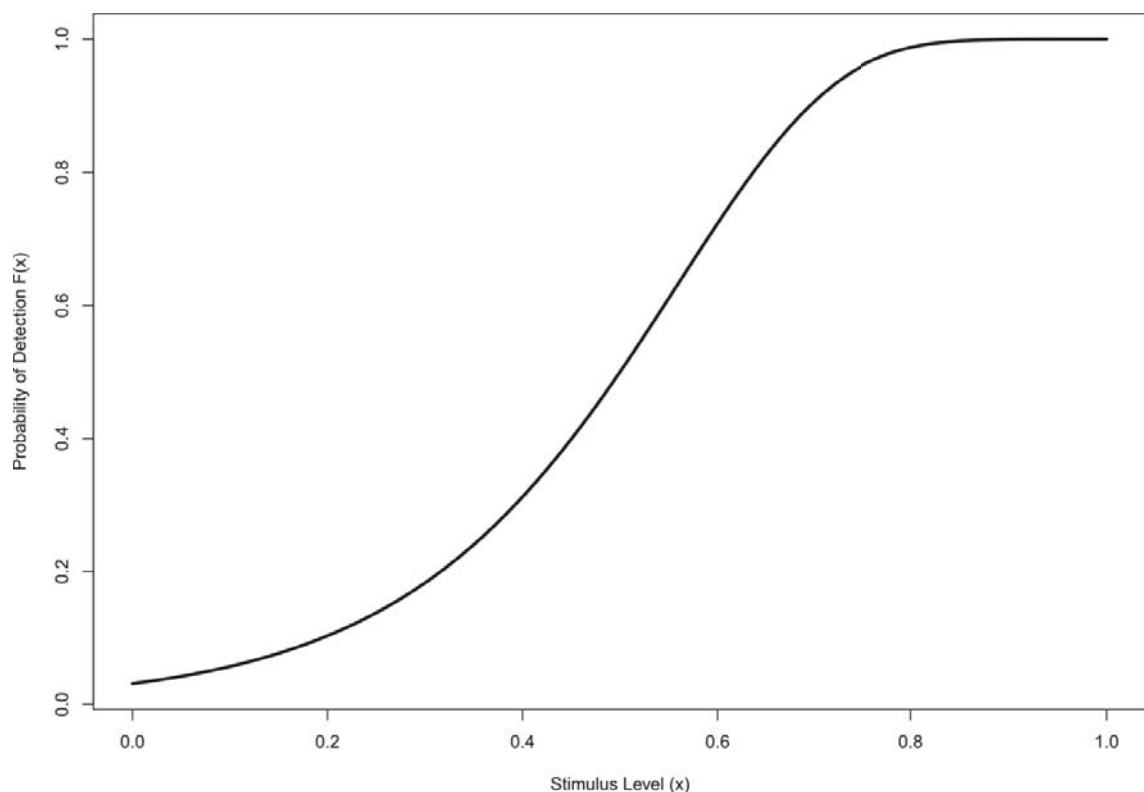


Figure 1 An example of a Gumbel psychometric curve modelling the relationship between Stimulus and Detection probability

2.1 Test Protocols: Yes-No and n-AFC Experiments

In order to approximate the underlying psychometric function or estimate a point thereon, a testing protocol for determining the response at a given stimulus intensity must first be decided upon.

Two of the most common protocols are the yes-no and n-Alternative Forced Choice (AFC) methods. The yes-no method involves the panellists being presented with either the signal or a blank sample at the given concentration, to which they respond yes or no depending on whether or not the signal was perceived. For the n-AFC procedure the subject receives n samples of which one has the signal and remaining are blanks. After sampling the n alternatives in a given order the subject indicates which one they believe contains the signal. A number of popular test procedures can be visualised in Figure 2.

















Test Protocol	Sample presentation & Instruction
	Signal=  Blank= 
Yes-no	Is this the signal or not? 
2-AFC	Which is the signal?  
3-AFC	Which is the signal?   
Duo-trio	Which one is the reference? Reference=   
Triangle	Which one is the odd one?   
Same-different	Are they the same or different?  

Figure 2 A visual summary of common sensory test protocols The figure is based on a similar one presentation at the 10th Sensometrics Conference (Lee, 2010). The original red wine glass scalable vector graphic (svg) file was downloaded from www.openclipart.org and modified for this figure.

Klein, 2001 points out that 2-AFC has been more popular than the yes-no method, which may be due to the assumption that it removes response bias. However it does have problems. The response error in the yes-no method occurs when a subject falsely believes that they can detect the signal when in fact they do not. It is also possible for the opposite to occur, that is the panellist detects the signal but believes that they have not. While the n-AFC method does remove this type of response bias it does introduce the possibility of what Klein calls interval bias. Here the subject has a preference for selecting certain positions as having the signal over others. For a 2-AFC task this may mean a higher chance of detecting the signal at a given intensity if it was sampled first in the set than if it was sampled second or vice-versa. Klein goes on to discuss methods to correct for these types of bias based on signal detection theory

(Green & Swets, 1966). Nevertheless the original experiment on which this investigation is based used a 3-AFC testing protocol for detection and therefore the methods discussed herein relate to this. It is possible to easily adjust them to suit any n-AFC experiment, and with a little more work to a yes-no task.

2.2 Methods for Estimating the Psychometric Curve

Once the testing protocol has been set, the focus now shifts to the estimation method. Often the parameter of interest is a specific stimulus level which results in a given detection probability on the psychometric curve. That is the parameter x_π , where $F(x_\pi) = \pi$, for a given probability π . Common choices for π are 50% (Kelareva, Mewing, Turpin, & Wirth, 2010) or when π minimises the so called sweat factor (Taylor, 1971). Minimizing the sweat factor should theoretically also minimise the variance of the point on the psychometric curve however this may not hold, especially when the underlying psychometric curve does not match the form of the fitted model (Garcia-Perez & Alcala-Quintana, 2007). For the purposes of this document we will define the sensory threshold as x_π , where $F(x_\pi) = 0.5$.

Various methods have been employed to estimate $F(x)$ itself or individual points thereon such as the intensity relating to 50% probability of detection. The sensory threshold T is often defined as this intensity, that is $F(T)=0.5$ (Alcalá-Quintana & García-Pérez, 2004). Three of the more traditional estimation methods are the method of Constant Stimuli, the Method of limits, and the Method of Adjustment.

2.2.1 Method of Constant Stimuli

This method requires a set of concentrations to be repeatedly tested by the subject. Each concentration is presented in a random order with equal frequency. The proportion of correct tests is then plotted against the concentrations and a parametric curve is fit allowing for interpolation between the set concentrations. This method is generally thought to produce the most accurate threshold and psychometric curve estimates, and a modified version of it has been referred to as the “gold standard” (Wise, Bien, & Wysocki, 2008). It has the advantage of completely estimating $F(x)$ although it often needs a large number of tests to make this possible. Due to this it may be unsuitable for many odour or taste threshold estimation experiments where set up time can be significant.

2.2.2 Method of Limits

The method of limits starts at an initial concentration with subsequent concentrations either always increasing or decreasing until a change over in response occurs. For ascending method of limits tasks the initial value is set at an undetectable level. The concentrations increase until the signal is detected. Similarly descending runs start at a clearly detectable level with concentrations decreasing until the signal is not detected. Averages of multiple runs ascending, descending or a mixture of the two, for a given subject can be used to improve threshold estimation. This method can introduce bias however, in the form of habituation and expectation. Habituation is the tendency for the subject to give the same response as was given previously (Amerine, Pangborn, & Roessler, 1965). Expectation on the other hand is when the subject anticipates the changeover and changes their response prematurely (Heymann & Lawless, 1999). To account for these potential biases various modifications have been used such as using an n-AFC testing method in place of a yes-no procedure.

2.2.3 Method of Adjustment

This method requires the subject to modify the signal intensity until it is barely detectable to them. This process is repeated a number of times to get an estimate with the results being aggregated. Allowing for subject to set the magnitude may or may not be feasible within the experimental setting.

In an effort to improve efficiency of threshold prediction, adaptive methods have also been used. It can be a major problem to acquire reasonably robust threshold estimates when the number of trials available is limited. An adaptive method bases the current testing level on the results of previous tests. Leek (2001) provides an overview of three such adaptive methods: Parameter Estimation by Sequential Testing (PEST), Staircase, and Maximum Likelihood.

2.2.4 PEST

The PEST procedure (Taylor & Creelman, 1967) begins at a set concentration, with a given step size and performance level to be achieved. The panellist is tested at the initial concentration multiple times until the observed proportion is deemed to provide evidence that the level is either too high or too low. This decision is made using a statistical test and it is carried out after each trial. The concentration is then adjusted

by the given step size in the appropriate direction. Using a PEST rule the step size is then adjusted and the process repeated until the step size reaches a desired level. At this point the process is terminated and the final signal magnitude is taken as the estimated concentration which results in the performance level of interest. Hall (1981) suggested using all of the collected responses to fit a psychometric curve instead of solely relying on the final test level. This also allowed for the estimation of a slope parameter and removed some of the original PEST method's sensitivity to the initial concentration and step sizes.

2.2.5 Staircase Procedures

Staircase procedures are similar to the PEST method except they remove a lot of the complexity of the decision rules to change the sample intensity. Depending on the point of interest on the psychometric curve a simple rule is used to adjust the signal level. The simplest staircase rule of one down – one up targets the 50% chance of detection. Here an initial concentration is presented, if the panellist correctly detects the signal then the next sample is of lower intensity if they do not then the level is increased. The step size and termination rules are also simplified resulting in easy to calculate signal adjustments.

2.2.6 Maximum Likelihood Adaptive Procedures

Leek (2001) defines this category as being *“characterized by stimulus placement on each trial, driven by consulting the current best estimate of the entire underlying psychometric function after every stimulus-response trial.”* One of the earliest methods to use this process was QUEST (Watson & Pelli, 1983). This method uses prior information expressed as a probability distribution of the threshold to fit the psychometric function. The function takes a predetermined form such as the cumulative density function (cdf) of the Weibull distribution, with the slope parameter fixed at a constant value prior to model fitting. After each sample the data is combined with the prior information using Bayes' theorem resulting in a threshold estimate which maximises the posterior distribution. This estimate is then taken as the concentration used in the subsequent dose-response test. Depending on the termination criteria, this process is repeated until a predetermined number of trials are reached or the confidence interval about the threshold is sufficiently narrow. The

final threshold estimate ignores the prior information and is the common maximum likelihood estimate.

2.2.7 Bayesian Adaptive Procedures

While the final estimate of the Quest method is the maximum likelihood estimate, much of the approach relies on Bayesian methods. Indeed Bayesian techniques have become increasingly popular in adaptive psychophysical methods (Garcia-Perez & Alcalá-Quintana, 2007; Kelareva et al., 2010; King-Smith, Grigsby, Vingrys, Benes, & Supowit, 1994; Kontsevich & Tyler, 1999; Kuss et al., 2005). One distinction between a Bayesian technique and other methods is the incorporation of ‘prior’ knowledge specified as a distribution, into the model. Using both the prior information and the observed data, one or more ‘posterior’ distributions are constructed which allows for inferences about the parameter of interest to be drawn. The constructions of the posterior distributions are not always trivial and often a Monte Carlo Markov Chain (MCMC) method is used to approximate these distributions. More details regarding MCMC methods are provided in the Methods section. As with the other adaptive methods the Bayesian techniques must address three main concerns: *“estimation of the psychometric parameters (threshold and slope), the termination rule, and placement of the next trial”* (Kontsevich & Tyler, 1999).

2.2.7.1 Estimation of the Psychometric Parameters

The Bayesian adaptive procedure generally links the probability of detection ψ to the sample intensity x through the relationship

$$\psi(x; \alpha, \beta, \gamma, \lambda) = \gamma + (1 - \gamma - \lambda)F(x; \alpha, \beta) \quad (1)$$

Where γ is the probability of guessing the correct response when the stimulus is undetectable to the subject and λ is the probability of lapsing and giving an incorrect response when in fact the signal is detected. In n-AFC experiments γ is often set to $1/n$. Setting $\lambda = 0$ makes the assumption that the subject always answers correctly when the signal is detected. Both α and β can be sensitive to violations of this assumption (Prins, 2012; Wichmann & Hill, 2001). Possible explanations for lapsing include blinking during a vision related detection task or incorrectly recalling the

position of the signal in an n-AFC task. This may be less of an issue with odour detection tasks where the number of samples is generally small and the subject has a reasonable amount of time to respond. $F(x; \alpha, \beta)$ is a function which models the underlying process linking signal intensity to the probability of detection, usually with two parameters α and β although there may be more depending on the family of curve being fitted. While it is convenient to assume that $F(x; \alpha, \beta)$ is invariant to whichever testing methodology is being used this so called ‘high threshold model’ has proven to be false (Klein, 2001). For example an estimated $F(x; \alpha, \beta)$ based on the 2-AFC method may differ from that produced using a 3-AFC on the same subjects. One measure that should be comparable amongst different testing protocols is the point on the full response curve relating to $d'=1$. Sometime referred to as the sensitivity index, d' is a measure of how far apart the signal and noise, as represented by standard normal distributions, are. A $d'=1$ corresponds to a signal level for an n-AFC experiment which elicits a probability of detection/discrimination equal to

$$P(y > \text{MAX}(x_1, \dots, x_{n-1})), \text{ where } y \sim N(1,1), \text{ and } x_1, \dots, x_{n-1} \sim N(0,1)$$

For a 2-AFC experiment, taking note that $y - x \sim N(1,2)$, this can be calculated as the

$$P(y > x) = P(y - x > 0) = 1 - \Phi\left(\frac{-1}{\sqrt{2}}\right) = 0.760$$

For a n-AFC where n is greater than 2 this can be more difficult to calculate but can be obtained from the literature such as $d'=1$ for a 3-AFC occurs at 0.633 (Peng, Jaeger, & Hautus, 2012). It is also reasonably straightforward to use simulation methods to find the approximate detection proportion relating to a $d'=1$. Custom R code, which can be found in the appendix, can produce an approximate proportion relating to $d'=1$ for an n-AFC design based on 100,000 simulations.

Theoretically the signal level on the 2-AFC response curve corresponding to a 76% detection rate should equate to the same signal level which relates to 63.3% rate for the 3-AFC response curve and so on. The advantage of the d' based estimates is that it is possible to make comparisons across studies. However if cross protocol comparisons are not required, then defining the threshold as the 50% detection level on the psychometric curve may be advantageous. Indeed Kuss et al., 2005 provides

alternative parameterisations of many of the common psychometric curves which include this threshold itself as a parameter.

The forms used by Kuss et al. (2005) explicitly treat both the threshold, defined as $m = F^{-1}(0.5)$, and either w , the width of the interval $F^{-1}(\alpha)$ to $F^{-1}(1-\alpha)$ for some small α , or the slope s , at threshold as parameters. That is F is parameterised in terms of either $\theta = [m, w]$ or $\theta = [m, s]$. By restricting w and s to being greater than 0 the resulting curves are also restricted to being monotonically increasing functions.

The forms themselves as presented by Kuss et al. (2005). are listed below and presented in Figure 3:

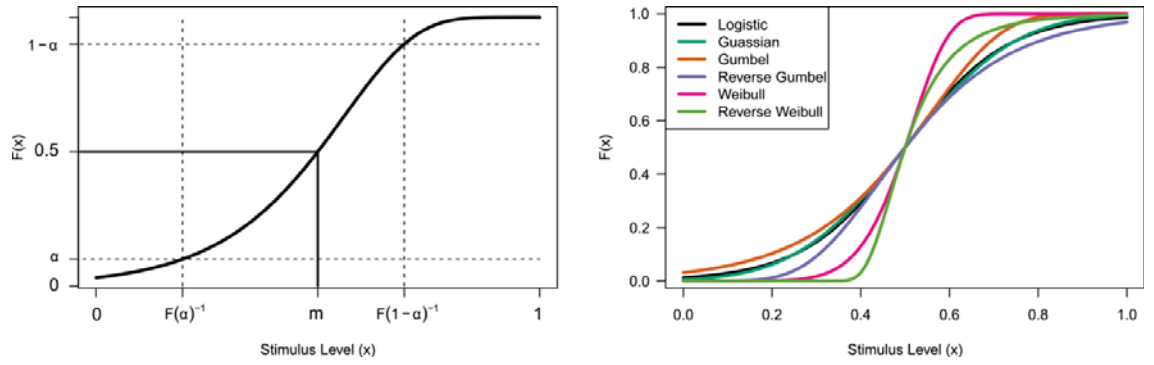


Figure 3 Parameterisation and forms of psychometric functions as presented by Kuss et al., 2005. The curves have been rescaled with all having threshold parameter $m=0.5$. For the logistic, Gaussian, and Gumbel forms the scale parameter $w=0.5$. The Weibull curves have slope at threshold parameters $s=5$.

Logistic. This is the standard logistic regression model found used in Generalized Linear Models (GLMs) re-parameterised as a function of m and w instead of the intercept and slope. This function is symmetric around the threshold, with $z(\alpha) = 2 \ln(1/\alpha - 1)$.

$$F_{logistic}(x, \theta) = \left(1 + \exp \left(-\frac{z(\alpha)}{w} (x - m) \right) \right)^{-1} \quad (2)$$

Gaussian. Similar to the standard probit model used for GLMs, but again re-parameterised to be a function of m and w .

$$F_{gauss}(x, \theta) = \Phi \left(x | m, \frac{w}{z(\alpha)} \right) \quad (3)$$

Note that here $z(\alpha) = \Phi^{-1}(1 - \alpha) - \Phi^{-1}(\alpha)$ with Φ defined as the cumulative density function (cdf) of the standard normal distribution and Φ^{-1} its inverse.

Gumbel & Reverse Gumbel. This function is a re-parameterized version of the log-log GLM, which results in an asymmetric curve. The probability of detection increases slowly over low stimulus levels but converges to 1 more rapidly as the levels increase. The asymmetry can be reversed resulting in the reverse Gumbel function.

$$F_{gumbel}(x, \theta) = 1 - \exp \left(-\exp \left(\frac{z(\alpha) - z(1 - \alpha)}{w} (x - m) + z(0.5) \right) \right) \quad (4)$$

$$F_{rgumbel}(x, \theta) = \exp \left(-\exp \left(\frac{z(1 - \alpha) - z(\alpha)}{w} (x - m) + z(0.5) \right) \right) \quad (5)$$

Where $z(\alpha) = \ln(-\ln(\alpha))$.

Weibull & Reverse Weibull. As with the Gumbel the Weibull produces an asymmetric curve. It cannot be parameterised in terms of w and instead is defined by m and s where $s = \frac{\partial F}{\partial x} \Big|_m$.

$$F_{weibull}(x, \theta) = 1 - \exp \left(-\exp \left(\frac{2sm}{\ln(2)} (\ln(x) - \ln(m)) + \ln(\ln(2)) \right) \right) \quad (6)$$

$$F_{rweibull}(x, \theta) = \exp \left(-\exp \left(-\frac{2sm}{\ln(2)} (\ln(x) - \ln(m)) + \ln(\ln(2)) \right) \right) \quad (7)$$

Interestingly this function is only defined for $x > 0$, both $F_{weibull}(x, \theta) \rightarrow 0$ and $F_{rweibull}(x, \theta) \rightarrow 0$ as $x \rightarrow 0$. This can be a desirable characteristic for a psychometric curve (Kuss et al., 2005).

These parameterisations have two advantages. Firstly as stated above, the threshold itself is a parameter and does not need to be calculated as a function of say slope and intercept estimates. Secondly, and more importantly, using the alternative parameterisation allows for the explicit specification of the prior distribution of the threshold and width. An example of this can be seen in Figure 4 where the more standard logistic regression parameterization together with a standard uniform prior on the slope strongly favours flat psychometric functions. The alternative more readily allows for a greater range of slopes, with the flatness of the prior translated into the function space. Kuss et al. (2005) provided a similar figure, with uniform priors over the range $[-1000, 1000]$ instead of $[0, 1]$ as used here. This had the effect of favouring steep rather than flat psychometric curve for the standard logistic regression parameterisation. However the point remains that using the threshold and width parameters simplify the specification of prior distributions.

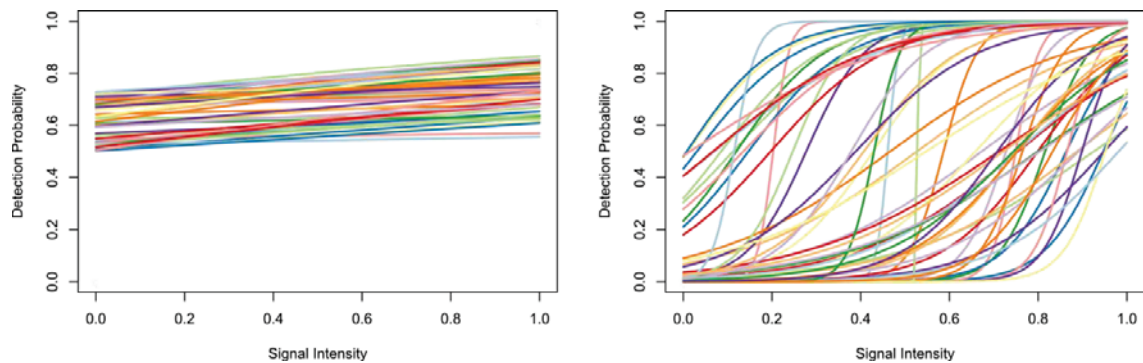


Figure 4 Effect of parameterisation of the logistic psychometric curve with standard uniform prior distributions for the parameters. The standard logistic parameterisation (left) strongly favours flat response curves, whereas the parameterisation from (2) allows for a wide range of curves.

2.2.7.2 Termination Rules

The termination rule can often be dictated by the experimental setting. One popular method is to continue until some pre-defined level of accuracy in the threshold estimate is reached. This could be a sufficiently narrow confidence interval (Watson & Pelli, 1983) or variance of the posterior distribution. This may be appropriate when a large number of queries are possible; however in the clinical setting the number of samples possible can be restricted to a low level. (Kelareva et al., 2010). In this case it can be more appropriate to stop after a set number of trials. As the original experiment, on which this investigation is based, used a small fixed number of samples, this will be assumed as the termination rule.

2.2.7.3 *Signal Intensity of the Next Sample*

There have been a number of schemes based on adaptive Bayesian methods presented in the literature to select signal intensities. Some common choices are to estimate the mean of the posterior distribution of the threshold based on the observed data up until that point and then set the next signal level equal to that mean (King-Smith et al., 1994). The median or mode have also been used in place of the posterior mean (Watson & Pelli, 1983) to place the next sample. We refer to these as is consistent with Kelareva et al., 2010, that is the MEAN, MEDIAN, and MODE query schemes respectively. How similar the results from these schemes are depends on how symmetric the posterior distribution is. A highly skewed posterior could lead to quite different designs.

Other schemes involve choosing the signal level which, based on the current estimate of the psychometric curve, minimises a characteristic of the threshold posterior distribution such as its entropy (Kontsevich & Tyler, 1999) or its variance (King-Smith et al., 1994). These are referred to as the MINENT and MINVAR procedures respectively.

Kelareva et al. provide an interesting hybrid whereby for a set number of samples n , the first $n-k$ samples follow the MINENT procedure, with the remaining k samples and final estimate are selected using one of the MEAN, MEDIAN, or MODE methods, based on a certain loss function to be minimised.

2.2.7.4 *Loss Function*

Kelareva et al. give a summary of some common loss functions used to evaluate the performance of the technique used to estimate the threshold T . The three loss functions used in that paper are referred to as ABSERR, SQERR, and CATERR and are outlined below.

ABSERR. This strategy involves minimising the mean absolute error, that is $|T - t|$, where T is the true threshold and t is the estimated threshold. This method can allow small numbers of large errors if the overall mean is still low.

SQERR. Here the aim is to minimise the mean of the square errors, i.e. $(T - t)^2$. While this is similar to the ABSERR method, it is more sensitive to large errors and thus it is appropriate when the goal is to reduce large, even if infrequent, errors.

CATERR. The categorical error is the proportion of estimated thresholds which are not equal the true threshold exactly. More formally, as defined by Kalereva, we want to minimise the mean of $L(T, t)$, where

$$L(T, t) = \begin{cases} 0, & t = T \\ 1, & \text{otherwise} \end{cases}$$

Clearly CATERR only makes sense when the threshold can only take discrete values. Therefore this loss function is confined to problems where the prior distribution of the threshold is discrete.

2.2.8 ASTM method

To estimate this threshold one could employ a non-parametric approach such as the ASTM last reversal method (ASTM International, 2011). While it is known to be biased (ASTM International, 2011) it should result in an estimate not far from stimulus level required to give 50% detection. The method works by presenting a run consisting of a number of 3-AFC tests with the stimulus level increasing each time by a constant multiplicative factor. The geometric mean of the concentration at which the last incorrect choice was observed and the next concentration is then calculated for each run to give a Best-Estimate Threshold (BET). It is also assumed that if the subject had been presented with the next step down from the lowest observed sample they would have answered incorrectly. This allows for a BET to be calculated when the subject correctly detects the signal in all samples. Similarly the next step up from the highest observed concentration is treated as being detected to account for cases where the highest concentration was not detected. An example of calculating the mean BET for two panellists is presented in Table 1. A similar table was presented in ASTM International, 2011, but was based on group rather than individual threshold estimates.

Table 1 Example of BET estimates for two panellists. Ticks indicate correct detection at a given concentration, and a cross an incorrect response.

Panellist ID	Concentrations (often in Log10 units)								Best Estimate Threshold (BET)
	1	2	3	4	5	6	7	8	
1	✓	x	x	✓	✓	✓	✓	✓	3.5
1	✓	✓	✓	✓	✓	✓	✓	✓	0.5
1	✓	✓	x	✓	✓	✓	✓	✓	3.5
1	x	x	x	✓	✓	✓	✓	✓	3.5
								Mean	2.75
2	✓	✓	✓	✓	✓	✓	✓	X	9.5
2	x	x	✓	✓	x	✓	✓	✓	5.5
2	✓	x	x	✓	✓	✓	✓	✓	3.5
2	x	✓	x	x	✓	✓	✓	✓	4.5
								Mean	5.75

2.3 Odour Detection Experiments at The New Zealand Institute for Plant and Food Research

As was previously stated the basis for this investigation is tied to experiments conducted at the New Zealand institute for Plant and Food Research (PFR). A more detailed outline of the experiments can be found in Peng et al. (2012) with a brief summary provided here. One hundred judges were divided into ten cohorts of size ten. For every cohort three odorants were tested with each odorant being presented in four repetitions of a series of eight 3-AFC trials. The series included eight concentrations which were presented in an increasing order with constant dilution factor. Filtered water was used as the solvent with the two blanks presented in each 3-AFC trial containing the solvent only. All target and blank solutions were placed in separate covered wine glasses and left for approximately one hour before being sampled. Only two sessions of differing odours, with a break in between were conducted per day in order to reduce the influence of fatigue on the panellists' detection ability. In addition the testing was conducted under green light with a 75 second delay between the three sample sets within the same series.

Threshold estimates were estimated for each series using the ASTM method, and the geometric mean of the thresholds over the four runs taken as the final threshold

estimate for the respective odour. The reason for taking the geometric means is that the concentrations with a constant dilution factor were represented on the logarithmic base 10 scale, resulting in log concentrations equally spaced. Taking arithmetic means on the log scale and then back-transforming onto the original scale is equivalent to taking the geometric mean of the un-transformed values.

The starting concentration and dilution factor for the three odours were adjusted when necessary after the completion of each cohort to ensure that the range of concentrations used for the current cohort encapsulated most of the subjects' threshold values.

2.4 Difficulties Implementing Standard Adaptive Procedures with Olfactory and Taste Threshold Estimation

In Leek's (2001) review of adaptive procedures they state that "Most of the development of these procedures has occurred in the context of vision or auditory research". While they go on to mention that Linschoten, Harvey, Eller, & Jafek, (2001) have used maximum likelihood adaptive procedures for both taste and smell studies, it still remains relatively uncommon. A major problem implementing these schemes relates to the logistics of rapidly altering the concentration levels after each response. In the experiment outlined above the samples were prepared an hour before being sampled to allow formation of the headspace. If the signal intensity needs to be changed rapidly, as with many of the methods previously mentioned, this would require a large number of samples to be prepared beforehand and/or large wait times between samples as the headspace reformed for repeated intensities.

One possible solution would be to use 'sniffin sticks' (Hummel et al., 1997), where the odour is administered by felt tip pens filled with the odorant rather than the compound diluted in water and placed in a wine glass. This would allow for the rapid altering of the odour intensities. However it requires one staff member to be present during the entire testing process to record the response and select the next odour level. When a large number of subjects are required this may become cost and time prohibitive.

For an adaptive method to be appropriate for experiments such as those used by PFR it may have to address a number of restrictions. These are:

- Any altering of the signal intensities cannot occur between samples within the same series. Due to logistical problems they must instead be altered after the end of the series or between cohorts.
- The number of tests within a series run and in total may be limited to low levels due to issues of fatigue and cost.
- It may be necessary for all subjects within a cohort to receive the same signal intensities

2.5 Proposed Adaptive Threshold Estimation for Olfactory and Taste Experiments.

Given the numerous models and points of interest on the psychometric curve it would be greatly advantageous to develop a flexible method which can easily alternate between the various psychometric and loss functions available to produce near optimal concentrations for the next series of tests. The increasingly popular Bayesian framework would be well suited to this problem. The proposed method is as follows:

1. Select a psychometric function such as Logistic, Weibull, Gaussian, etc. to model and a loss function to minimise
2. Generate a large number of psychometric functions from the prior distributions or even the posterior distributions based on previous trials.
3. Select signal intensities ξ for a series of tests.
4. Generate responses based on ξ and the chosen psychometric function.
5. Estimate the psychometric curve using MCMC methods and/or the properties of its posterior distribution.
6. Calculate the approximate loss given the estimated psychometric curve and a given loss function, e.g. MSE
7. Estimate the expected loss by averaging the losses over all simulated samples.
8. Alter the signal intensities and repeat steps 4 through 7 until the expected loss is approximately minimised.

This process does present a number of difficulties. One major issue is the expense in terms of time taken to approximate the loss function. Fitting a large number of psychometric curves based on simulated data could be very time consuming especially with MCMC methods which themselves require large numbers of iterations to approximate the posterior distributions. One possible solution would be to approximate the MCMC solution by means of a neural network. This could itself be time consuming as a large number of samples would be needed to train, validate and test the neural network. However once the network is trained it allows for rapid function approximation to be calculated.

Another option is to define priors with discrete distributions as this would allow for the posterior distributions to be calculated exactly. As long as the number of levels and/or the number of points in the design ξ are not too large then the expected loss should be estimable in a reasonable time frame.

Adjusting the concentrations to minimise the loss function is also not trivial. Even by simplifying the problem to finite sets of possible concentrations the complexity quickly increases with the number of design points needed. Methods such as MINENT only look one concentration ahead. It is reasonably straightforward to evaluate a single intensity at a time from a set and select the ‘best’ choice. However when setting the concentrations for session of k samples from n possible concentration levels we have a k -combinations with repetition. The order of the sequence is ignored, which is reasonable since the sequence is increasing in the PFR setup. More explicitly the number of possible concentration levels for the next run is

$$\binom{n+k-1}{k} = \frac{(n+k-1)!}{(k)!(n-1)!}$$

For example if $n=20$ and $k=8$, there would be 2,220,075 possible concentrations to evaluate, whereas with MINENT there would be 20. Depending on the size of n and k it may be unfeasible to evaluate every possible combination and therefore some sort of solver may be employed. Modified versions of global solvers such as the particle swarm or genetic algorithm could provide an approximate solution. It could also be possible to not restrict the concentrations to a finite set and instead treat them as

continuous. This would allow for a much wider range of solvers to be used, although the non-smooth nature of the loss functions may restrict these to derivative free methods. The issues encountered by this proposed method are discussed in more detail in the methods section.

It has been previously demonstrated that adaptive Bayesian methods have been effective in providing designs which improve the quality of sensory threshold estimates given a limited number of trials possible. The above method aims to expand these methods to situations where the design can only be altered between sessions containing a set of trials. This means that methods must be able to look-ahead multiple steps to calculate the expected loss. If this is possible the benefits of using the adaptive Bayesian designs could be applied to experiments such as those carried out at PFR, where the previous adaptive approaches were not feasible.

3 Methods

3.1 Software

The Bayesian analysis, computer simulations, neural networks and optimization algorithms were implemented in the Python programming language version 2.7.3 (www.python.org). A number of required python libraries were compiled from source under Linux OS kubuntu 10.04. Many libraries have existing precompiled windows binaries in order to install them under MS windows. Python versions 3x are based on a major re-write of some of the base code and therefore some libraries which have not been updated will not function on these versions. To complicate matter further some components required to install certain packages such as Numpy are not available for 64bit versions of python. For these versions it is recommended to download precompiled versions from Christophe Gohlke's website (<http://www.lfd.uci.edu/~gohlke/pythonlibs/>).

One other recommended method for installing various libraries is to install and use PIP. A windows binary of PIP can be found at the above website however it also requires Distribute to also be installed. The simplest way to achieve this is to download the setup file for distribute from here:

http://python-distribute.org/distribute_setup.py

First python has to be added to the environment variables path. Python 3.3 will do this automatically however any previous version will need to be added manually. Once this is done open the command shell in windows and navigate to the folder containing the setup file. Then use the following command to install Distribute:

```
python.exe C:\Path\to\distribute_setup.py
```

With Pip and Distribute installed libraries can be added to python simply by, while in the windows command shell typing:

```
pip install libraryX
```

Where libraryX is the package to be installed.

Alternatively they can be compiled from their source code using the Mingw compiler with the Msys command shell (www.mingw.org). The libraries used in the research project are listed below.

3.1.1 Numpy and Scipy

Numpy and Scipy (Jones, Oliphant, Peterson, & others, 2001; Oliphant, 2007) extend python to enable scientific programming. Numpy allows for the specification and manipulation of arrays and matrices while Scipy contains many mathematical and scientific functions including local optimization routines. Many other libraries including those listed here require Numpy and Scipy in order to function themselves. If using windows and a 64bit version of Python it is necessary to get a windows binary installer for Numpy built using Intel's MKL library as there are missing components needed to build using BLAS. These binaries can be found at Gohlke's website.

3.1.2 Matplotlib

Matplotlib (Hunter, 2007) is a 2-d plotting library and is necessary to provide diagnostic plots of the PyMC output. Many other libraries also recommend this package, most notably Numpy and Scipy, to provide publication quality figures.

3.1.3 PyMC

PyMC (Patil, Huard, & Fonnesbeck, 2010) is a Bayesian statistics library. It includes Monte Carlo Markov Chain methods to fit Bayesian models and allows for user customization to increase speed and convergence.

3.1.4 Playdoh

Playdoh provides the Particle Swarm (Vaz & Vicente, 2007) global optimizing algorithm to solve minimization problems. It also provides multicore support to speed up the optimization process. Playdoh is currently not compatible with Python versions 3x.

3.1.5 Numdifftools

Numdifftools *"Solves automatic numerical differentiation problems in one or more variables"*. This was needed to estimate the hessian matrix of the negative log-likelihood for the attempt to find D-optimal designs.

3.1.6 Numexpr

Numexpr provides a speed increase in evaluating many of Numpy's basic array functions.

3.1.7 ffnet

Ffnet (Wojciechowski, 2011) is a feed forward neural network library. It enables the fitting and evaluation of neural networks and the ability to load and save them. Much of the underlying processes are run in FORTRAN providing a great increase in speed. It also includes a version of the genetic algorithm which can be adapted to solve discrete optimization problems.

3.1.8 OpenOpt

OpenOpt (Kroshko, 2007) provides a large number of solvers relating to a variety of optimization problems. Of particular interest here, is the Global Problem (GLP) solvers which can be used to find the approximate global minimum or maximum of the function of interest.

3.2 ASTM method

Given an underlying psychometric curve which generates responses to certain levels of stimuli, the threshold estimates based on the ASTM method can be thought to follow a discrete distribution whose mean and variance is easily estimable. For each run of increasing n stimuli there are only $n+1$ possible threshold estimates. Each possible threshold is based on an unsuccessful detection at the concentration immediately prior to the threshold and successful detection for all concentrations greater than the threshold. The probabilities of each threshold estimate given the underlying function can be easily obtained, which in turn allows for the straightforward calculation of expected values and variance of the estimator at the set concentrations. An example is given below using a logistic curve with a true threshold of 5 (Figure 5 & Table 2).

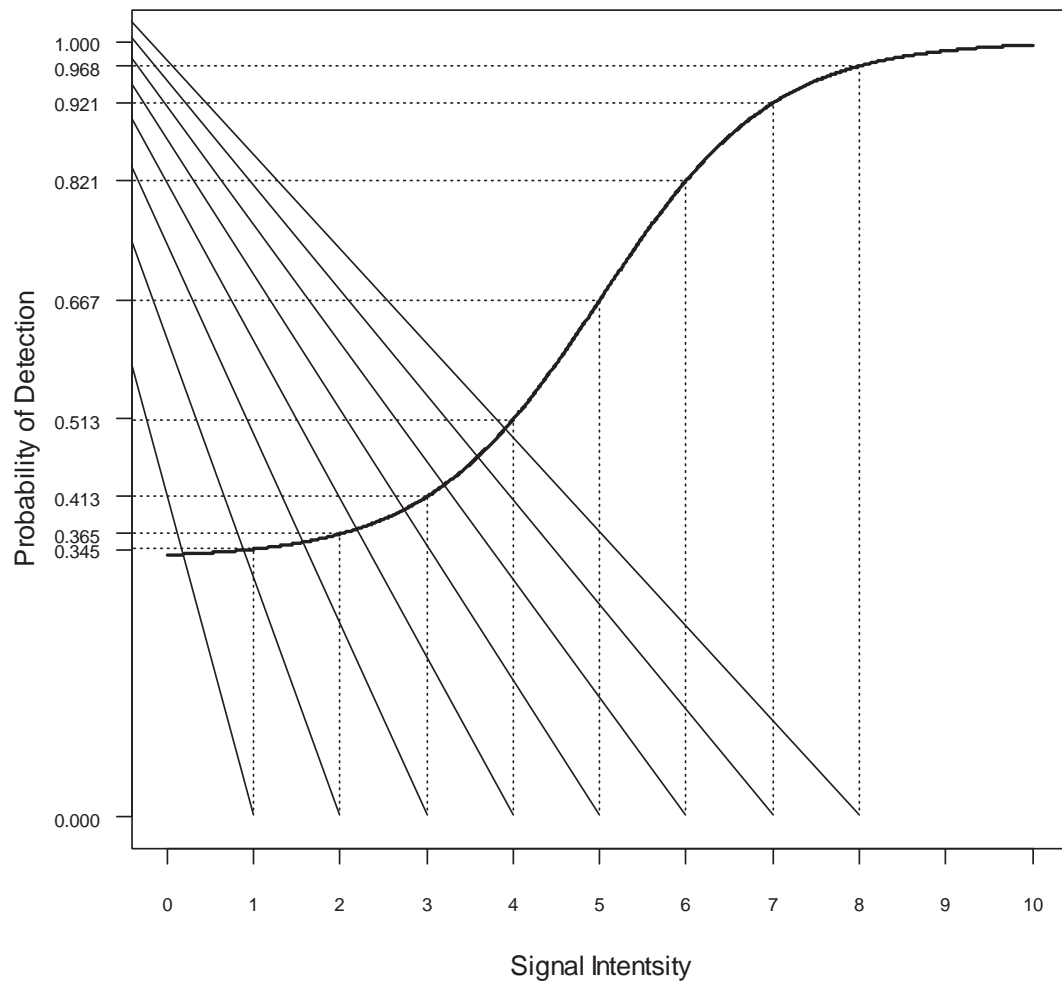


Figure 5 Example of estimated detection probabilities for a psychometric curve at given signal intensities.

Table 2 Variance, Bias, and MSE estimation for the ASTM method.

Signal Intensity x_i	Probability of detection $f(x_i)$	Probability of detection for all $x \geq x_i$	Threshold t_i	probability of threshold $g(t_i)$	$t_i \times g(t_i)$	$t_i^2 \times g(t_i)$
1	0.345	0.013025	0.5	0.013025	0.006513	0.003256
2	0.365	0.037754	1.5	0.024729	0.037093	0.05564
3	0.413	0.103436	2.5	0.065682	0.164205	0.410511
4	0.513	0.25045	3.5	0.147014	0.51455	1.800925
5	0.667	0.488207	4.5	0.237757	1.069906	4.814575
6	0.821	0.731944	5.5	0.243738	1.340556	7.37306
7	0.921	0.891528	6.5	0.159584	1.037293	6.742403
8	0.968	0.968	7.5	0.076472	0.57354	4.30155
			8.5	0.032	0.272	2.312
			Sum		5.015655	27.81392

Given the underlying psychometric function and concentrations used the expected value of the estimated threshold, as calculated above, is 5.0157. The expected Bias is therefore $5.0157 - 5 = 0.0157$. Since $Var(\hat{\theta}) = E[\hat{\theta}^2] - E[X]^2$, and $MSE(\hat{\theta}) = Var(\hat{\theta}) + Bias(\hat{\theta})^2$, the expected variance and mean square error of the estimator in this example are $27.8139 - 5.0157^2 = 2.6571$ and $2.6571 + 0.0157^2 = 2.654$ respectively. This example is over simplified as the underlying curve will be unknown in practical applications. However it demonstrates the trivial calculation of the measures of the estimator's performance. Instead of a single curve, a sample of curves could be taken from a prior distribution representing the researcher's uncertainty and the concentrations can be set based on the measures calculated above. This is a reasonably straightforward and quick process which serves as a baseline for comparison with the more time consuming Bayesian approach described later in this section.

3.3 Threshold Estimation using the Psychometric Function

3.3.1 Maximum Likelihood

Given a set of responses and the form of the psychometric curve one can estimate the threshold which maximises the likelihood. More formally for a set of concentrations c_1, c_2, \dots, c_N , the number of trials at each concentration n_1, n_2, \dots, n_N and the respective number of correct responses x_1, x_2, \dots, x_N the likelihood is

$$L(m, s) = \prod_{i=1}^N \frac{n_i!}{x_i! (n_i - x_i)!} \pi_i^{x_i} (1 - \pi_i)^{n_i - x_i}$$

Where π_i is the probability of detection at c_i and is defined by the underlying psychometric function.

As Kuss et al. point out; this is the “standard binomial mixture model for parametric functions as assumed in virtually every study on psychometric functions”.

If for example, the psychometric function takes a logistic form then with a threshold m , scale parameter w , guessing probability γ , ignoring the lapse probability and setting $z(\alpha) = 1$ for simplicity, gives:

$$\pi_i = \gamma + \frac{(1-\gamma)}{1 + \exp\left(\frac{m-x}{w}\right)} = \frac{1 + \gamma \exp\left(\frac{m-x}{w}\right)}{1 + \exp\left(\frac{m-x}{w}\right)}$$

$$\text{And } 1 - \pi_i = 1 - \gamma - \frac{(1-\gamma)}{1 + \exp\left(\frac{m-x}{w}\right)} = \frac{(1-\gamma) \exp\left(\frac{m-x}{w}\right)}{1 + \exp\left(\frac{m-x}{w}\right)}$$

$$L(m, w) = \prod_{i=1}^N \frac{n!}{x_i! (n_i - y_i)!} \left(\frac{1 + \gamma \exp\left(\frac{m-x_i}{w}\right)}{1 + \exp\left(\frac{m-x_i}{w}\right)} \right)^{y_i} \left(\frac{(1-\gamma) \exp\left(\frac{m-x_i}{w}\right)}{1 + \exp\left(\frac{m-x_i}{w}\right)} \right)^{n_i - y_i}$$

To find the threshold m and w which maximise this likelihood i.e. the MLE's of m and w , one method is to take the derivatives with respect to m and w of the log-likelihood, set it equal to zero and then solve for m and w respectively. Once this has been completed one should also confirm that it is indeed the local maximum.

$$\begin{aligned} \ln L(m, w) &= \sum_{i=1}^N \ln \frac{n!}{x_i! (n_i - y_i)!} + y_i \ln \left(1 + \gamma \exp\left(\frac{m-x_i}{w}\right) \right) \\ &\quad + (n_i - y_i) \ln \left((1-\gamma) \exp\left(\frac{m-x_i}{w}\right) \right) - n_i \ln \left(1 + \exp\left(\frac{m-x_i}{w}\right) \right) \end{aligned}$$

$$\frac{d \ln L(m, w)}{dm} = \sum_{i=1}^N \frac{\exp\left(\frac{m-x_i}{w}\right) \gamma y_i}{\left(\exp\left(\frac{m-x_i}{w}\right) \gamma + 1 \right) w} + \frac{n_i - y_i}{w} - \frac{\exp\left(\frac{m-x_i}{w}\right) n_i}{\left(\exp\left(\frac{m-x_i}{w}\right) + 1 \right) w}$$

$$\begin{aligned} \frac{d \ln L(m, w)}{dw} &= \sum_{i=1}^N - \frac{\exp\left(\frac{m-x_i}{w}\right) \gamma (m-x_i) y_i}{\left(\exp\left(\frac{m-x_i}{w}\right) \gamma + 1 \right) w^2} - \frac{(n_i - y_i)(m-x_i)}{w^2} \\ &\quad + \frac{\exp\left(\frac{m-x_i}{w}\right) n_i (m-x_i)}{\left(\exp\left(\frac{m-x_i}{w}\right) + 1 \right) w^2} \end{aligned}$$

Setting these resulting derivatives to zero and solving is not trivial using algebraic means and would be even further complicated with the addition of a lapse parameter.

For this reason the MLE's are often estimated by numerical methods such as the Newton-Raphson. This can be reasonably straightforward to perform, but it means that obtaining the expected values, variance, and MSE measures as was done for the ASTM method, requires generating every possible response pattern, calculating the probability of observing said pattern assuming the underlying model, and then estimating the MLE's for each. While this can be done in a reasonable time frame, the optimisation algorithms needed to minimise the loss functions require many such calculations leading to a computationally expensive process. One possible solution is that rather than evaluating all possible responses, instead a large sample of responses is generated using the underlying model and current concentration levels. If for example we had four repetitions of 3-AFC tests at each of eight concentration levels then there are $5^8=390,625$ possible response patterns. If instead, only say 10,000 response patterns were randomly sampled using the underlying psychometric curve, then approximate estimates of the mean, variance, etc. could be computed at a fraction of the computational cost.

3.3.1.1 D-Optimal Designs

One common frequentist approach to optimal design when classical designs are not appropriate is the D-optimal design. There are D-optimal designs available for generalised linear models such as the bivariate logistic regression (Heise & Myers, 1996). This is a computer generated design which maximises the determinant of the information matrix of the model parameters, or equivalently minimises the determinant of the inverse hessian matrix. The general approach for a logistic regression is to assume an underlying model which in this setting is a psychometric curve. Then adjust the design ξ until the determinant of the inverse hessian matrix of the negative log likelihood evaluated at the maximum likelihood estimate, is minimised. Custom python code was written to provide the D-optimal designs which was first checked against known results for the standard logistic regression. The problem with the D-optimal design outlined here is that it is optimal assuming only a single curve. Therefore for any individuals with psychometric curves different from the assumed model, the design may not be close to optimal.

3.3.2 Bayesian Fitting

One alternative to MLE estimation of model parameters θ is Bayesian inference. This has the advantage of incorporating prior knowledge in addition to the data to make inferences about θ . This relationship is defined by Bayes' rule:

$$p(\theta|data) = \frac{p(data|\theta)p(\theta)}{p(data)}$$

The relationship results in a probability distribution of θ given the observed data $p(\theta|data)$ referred to as the posterior distribution. This is constructed using the likelihood function $p(data|\theta)$ and the prior $p(\theta)$ along with the denominator $p(data)$ which is a constant. As the number of observed data points increases the posterior becomes less influenced by the prior distributions.

Returning to the previous logistic psychometric function example, given data consisting of n_i , x_i , and c_i . The posterior is defined as

$$p(m, w|data) = \frac{p(data|m, w)p(m)p(w)}{\int p(data|m, w)p(m)p(w) dm dw}$$

The denominator $\int p(data|m, w)p(m)p(w) dm dw$ is a constant which ensures that the posterior integrates to 1 as a probability distribution should. The solution to this denominator integral is often very difficult to obtain unless "conjugate" prior distributions are used as they result in a posterior with a standard p.d.f. Often however the problem is side stepped by ignoring this denominator completely and instead approximating the posterior distribution using Monte Carlo Markov Chain (MCMC) methods to sample from the posterior based on the weaker relationship:

$$p(m, w|data) \propto p(data|m, w)p(m)p(w)$$

There are various algorithms, or step methods available to sample from the posterior with popular methods being Metropolis-Hastings or Gibb's sampling. For the purposes of this investigation the PyMC python library uses the Metropolis method, which is a special case of the Metropolis-Hastings algorithm for the MCMC sampling.

3.3.2.1 Metropolis Algorithm

The Metropolis algorithm used allows for the sampling of a distribution where it is possible to calculate a value proportional to the p.d.f. Continuing the logistic psychometric example, let:

$$f(m, w) = p(data|m, w)p(m)p(w) \propto p(m, w|data)$$

Then the Metropolis algorithm in PyMC is as follows:

- 1) Set initial values m_0 and w_0 .
- 2) At the current values m_t and w_t , draw candidate values $m' \sim N(m_t, \sigma_m^2)$ and $w' \sim N(w_t, \sigma_w^2)$
- 3) If $\frac{f(m_t, w_t)}{f(m', w')} > 1$, then $m_{t+1} = m_t$ and $w_{t+1} = w_t$
- 4) Else m_{t+1} and $w_{t+1} = \begin{cases} m' \text{ and } w' \text{ with probability } \frac{f(m_t, w_t)}{f(m', w')} \\ m_t \text{ and } w_t \text{ with probability } 1 - \frac{f(m_t, w_t)}{f(m', w')} \end{cases}$
- 5) Repeat steps 2 through 4 a large number of times, tuning σ_m^2 and σ_w^2 where needed to allow for reasonable acceptance levels while at the same time trying to decrease the correlation amongst consecutive samples.

Employing the Metropolis algorithm as outlined above results in a large sample from the posterior distribution. The observations near the beginning of the chain are often discarded as they can be highly influenced by the initial starting points m_0 and w_0 . As it can take a substantial number of the subsequent samples before the model converges this “burn in” period can be large. Additionally the Metropolis method can produce samples whose sequence has high auto correlation. Using larger values of σ_m^2 and σ_w^2 can reduce this, however this will often reduce the speed of the model mixing as it can lead to a lower acceptance rate of the candidate values. A commonly employed solution is to instead “thin” the remaining samples by taking every n^{th} value of the chain, where n is adjusted until the auto correlation has been sufficiently reduced.

PyMC does allow for other step functions, including custom code. However the Metropolis algorithm provided sufficient results with a burn in of 1000, and a thinning factor of 10 to produce 900 samples for each run of model fitting. While the 900 thinned samples will have some variation due to MCMC sampling error this was

deemed sufficient due to the trade off between accuracy and the time needed to run a large number of MCMC simulations.

3.3.2.2 Bayesian Point Estimates

Once the posterior has been approximated an appropriate point estimate is reported. The choice of estimate often depends on the loss function, however the most common estimates are the median (Median), mean (Mean), or the mode, also known as the maximum a posteriori (MAP), of the posterior distribution.

The MAP can be estimated in a similar manner to the MLE and does not actually require any MCMC sampling. In this investigation the Powell's minimization method (Powell, 1964) was used in PyMC to obtain the MAP estimates of θ which maximise $p(data|\theta)p(\theta)$. It should also be noted if $p(\theta)$ is a flat prior then the MAP will be close to if not identical to the MLE.

The Mean estimate can be calculated as the mean of the MCMC sample. Similarly the Median estimate is the sample median. The choice of which statistics to report as the parameter estimates depends on the loss function. Kelereva et al. note that the MAP minimises the expected CATERR loss function, while Mean and Median minimise SQERR and ABSERR respectively.

In the simulations used in this investigation the MAP, Median, and Mean's were all calculated and stored along with the standard deviations of the posterior distributions.

3.3.2.3 MCMC Fitting

The Bayesian model fitting was conducted using PyMC 2.2 library in Python 2.7. The model parameterization used was consistent with Kuss et al. (2005). For ease of model specification both the threshold and width parameters are assumed to lie within the (0,1) interval (Treutwein & Strasburger, 1999). Any real world application would involve normalising the stimulus values (x_1, \dots, x_n) by the linear transformation $y = A + Bx$ to map X onto the (0,1) space, and back-transform the result using $x = \frac{y-A}{B}$. A and B are calculated as

$A = x_{min} - Bx_{min}$ and $B = \frac{1}{x_{max} - x_{min}}$ where x_{min} and x_{max} are the limits of the sensory space in the original units. Treutwein & Strasburger (1999) describe the same transformations as above but have an error in the formula for the calculation of B.

Standard uniform prior distributions were assigned to threshold and width parameters. While Kuss et al. (2005) recommend using a gamma or log normal prior for the width parameter the alternative approach of Trutwein and Strasburger was used. Here the width parameter, w , has a standard uniform prior also, however it is then transformed from the (0,1) interval onto (0.005,0.5) by

$$w^* = 0.005 \cdot 10^{2w}$$

And back-transformed with

$$w = \frac{1}{2} \log_{10} \left(\frac{w^*}{5} \right) + 1.5$$

The upper bound of 0.5 for w^* allows for a flat psychometric curve over the stimulus space (0,1), while the lower bound allows for a very steep curve.

For the current investigation the guess parameter was fixed at 1/3 as is consistent with the 3-AFC design, while the lapse parameter was set to 0.0001. The reason for having a non-zero lapse parameter was to avoid errors in PyMC caused by the observed data having a virtual probability of zero under a MCMC parameter sample. Instead the offset make this probability very small and so allows for rare occurrences of incorrect discrimination for sensitive individuals when the signal is very strong. The logistic psychometric curve used in the simulations is therefore:

$$P(detection|x, m, w) = \frac{1}{3} + \left(\frac{2}{3} - 10^{-4} \right) \left(1 + \exp \left(- \frac{(x - m)}{0.005 \cdot 10^{2w}} \right) \right)^{-1}$$

The model fitting involved firstly calculating the MAP and then generating 10,000 MCMC samples, of which 1000 were discarded as a burn-in and a thinning factor of 10 applied to the remaining observations (Figure 6). This was deemed sufficient, based on some simulated responses, to achieve a converged model while adjusting for

autocorrelation amongst the samples. The relatively low number of samples (900) remaining to base the posterior estimates was in the interest of speed.

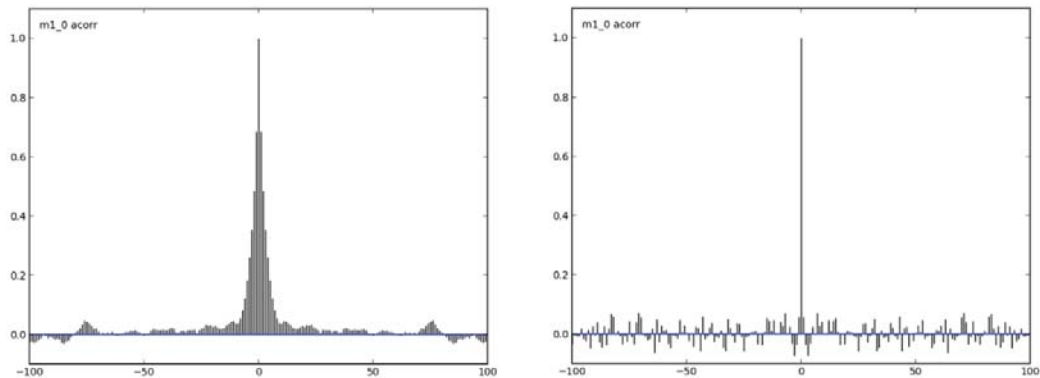


Figure 6 Autocorrelation plots for Threshold parameter estimates. Original (left) and thinned by factor of 10 (right)

3.3.2.4 Assessing MCMC Convergence and Sensitivity to Initial Values

While the choice of the number of MCMC samples, burn-in, and thinning parameters were based on reducing the auto-correlation in the parameter's trace this does not guarantee that the model has converged. The suitability of the MCMC setup was investigated through two procedures. The first of which was the Raftery–Lewis approach (Raftery & Lewis, 1995). This method as implemented in PyMC gives a recommended number of posterior samples, burn-in, and thinning to estimate a chosen percentile to within a given accuracy at a set confidence level. These estimates are based on a posterior trace from a pilot MCMC run. For each of the 256 possible response patterns to 3-AFC tests at 8 equally spaced concentrations ranging from 0 to 1, MCMC runs of length 10,000 with no burn-in or thinning were run to produce posterior samples for the threshold and scale parameters. For each of these the Raftery-Lewis diagnostic produced recommended MCMC fitting parameters for estimating the 0.025 and 0.975 quantiles of the respective posterior distributions to within 0.01 of the true quantile with 95% confidence. Over all response patterns and estimated quantiles the recommended burn-in was always less than 150 samples which indicate that the 1000 burn-in used may have been excessive. For estimating the 0.025 quantile of the threshold and scale posterior distributions the recommended run length and thinning parameters only 1 out of 256 only slightly exceeded those that

were used. Estimating the 0.975 quantile was a little more problematic with 109 out of 256 exceeding the run length and thinning used in the final estimates, sometimes by up to 3 times. However the main reason for requiring the extra samples appears to be due to produce an independence chain. If the model has converged and the run length is reasonable, posterior estimates can still be calculated from dependent samples. However looking at the 0.975 quantile to within 0.02 of the true quantile the recommended run lengths are always lower than what was eventually used, although the recommended thinning factor was greater than 10 in some cases. Based on the Raftery-Lewis diagnostic it appears that the burn-in of 1000, thinning factor of 10, and total run length of 10,000 was adequate to get reasonable estimates of the posterior threshold and scale distributions in terms of the 0.25 and 0.975 quantiles at least.

The second diagnostic used was the Gelman-Rubin method (Gelman & Rubin, 1992) which assesses the model's sensitivity to the initial values especially when multi-modality is present in the posterior. For each of the same 256 response patterns used above, ten chains with random initial values were run for each model runs of length 10,000, burn-in of 1,000 and thinning factor of 10. The Gelman-Rubin diagnostic looks at the between and within chain variances in much the same way the standard ANOVA does. In PyMC the function returns a statistic R which is based on the ratio of these two variances. Values close to 1 indicate that the chains have all converged to the same distribution. For the pilot runs the observed ratios were all within the range 0.999 to 1.061 indicating that the chains all converged despite the differing initial values.

Overall the MCMC fitting parameters appear to have been adequate to get reasonable estimates of the posterior distributions, although longer run lengths may have been beneficial. There is also the chance that for some of the response and concentration combinations these parameters would not be appropriate, however based on the pilot samples used this would seem to be unlikely.

3.3.2.5 Approximating Posterior Estimates using a Neural Network.

Simulating responses based on the prior distributions of $\theta = \{m, w\}$, then calculating the threshold estimate and its respective loss provides a means to obtain a reasonable approximation to the expected loss function. However, approximating the posterior distribution through MCMC and then evaluating a given loss function for each simulated sample can be very time consuming, let alone when this needs to be repeated multiple times in order to minimise said loss function. The calculation of the posterior Mode can be calculated reasonably quickly using optimization techniques but the Mean, Median and Entropy require the full posterior distribution. One attempt to overcome this was to simulate a large number of observations as before and calculate the posterior mean, mode, median, and discretised entropy for each sample. A Neural Network (NN), sometimes referred to as Artificial Neural Network (ANN), was then trained based on a large subset of these simulated values to predict the evaluated measures.

Neural networks have successfully been used to approximate functions (Li, 2008). A NN is a network of neurons arranged in layers. The neurons in the first layer are the predictor variables commonly referred to as 'inputs', while in the last layer they are the dependent variables or 'targets'. In practice there is usually only need for one intermediate or 'hidden' layer and very seldom are more than two layers required. There are a number of different forms of NN however for the purposes of this investigation Feed Forward NN's with one hidden layer are considered. A Feed Forward NN means that information flows from the input layer through to the hidden layer and finally to output (target) layer (Figure 7). It does not for example flow from the target layer back to the hidden layer. Often the input and target variables will be scaled to lie in the interval between 0.15 and 0.85 to aid in the optimization process.

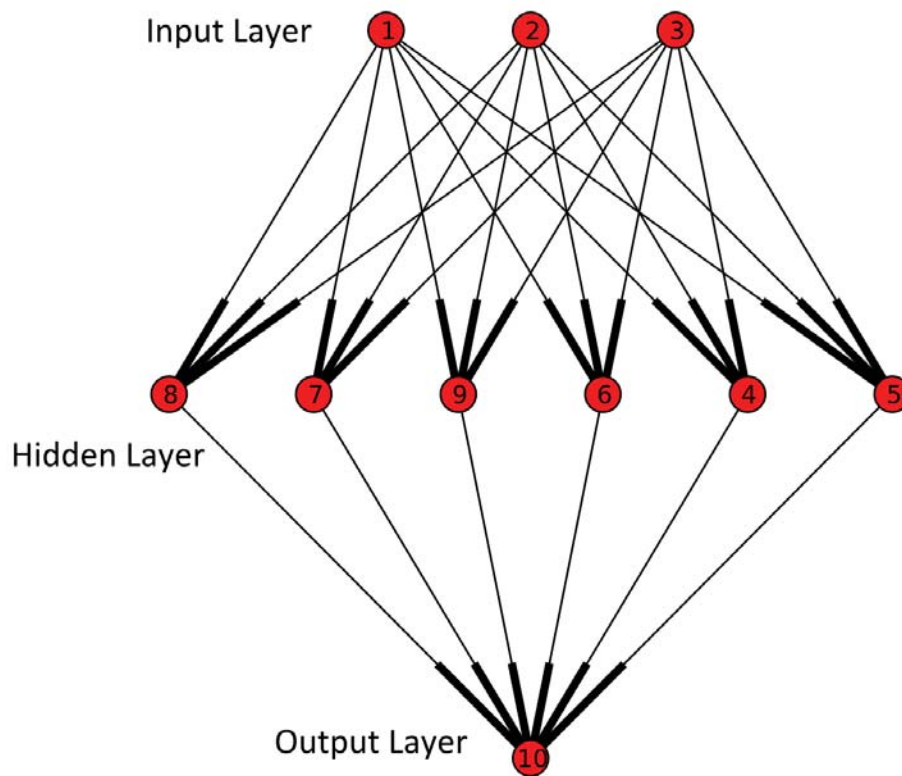


Figure 7 Example of a Neural Network with 3 input, 6 hidden and one target neuron.

The idea of NN is not new, in fact it dates back to work by Warren McCulloch and Walter Pitts in 1943 (McCulloch & Pitts, 1990). The basic process for each neuron in the hidden and target layers is to receive a number of inputs from the previous layer. Sometimes an additional “bias” input with value 1 is also included. Weights are then assigned to each of these inputs, and the sum of the weighted inputs is calculated. Finally this sum is passed through an activation function, with the resulting output passed on to the next layer or if in the target neuron, taken as the final estimate(s). In other words for a given neuron with inputs x_0 to x_n , weights w_0 to w_n , activation function g , the output y is defined as $y = g(\sum_{i=0}^n w_i x_i)$ see Figure 8

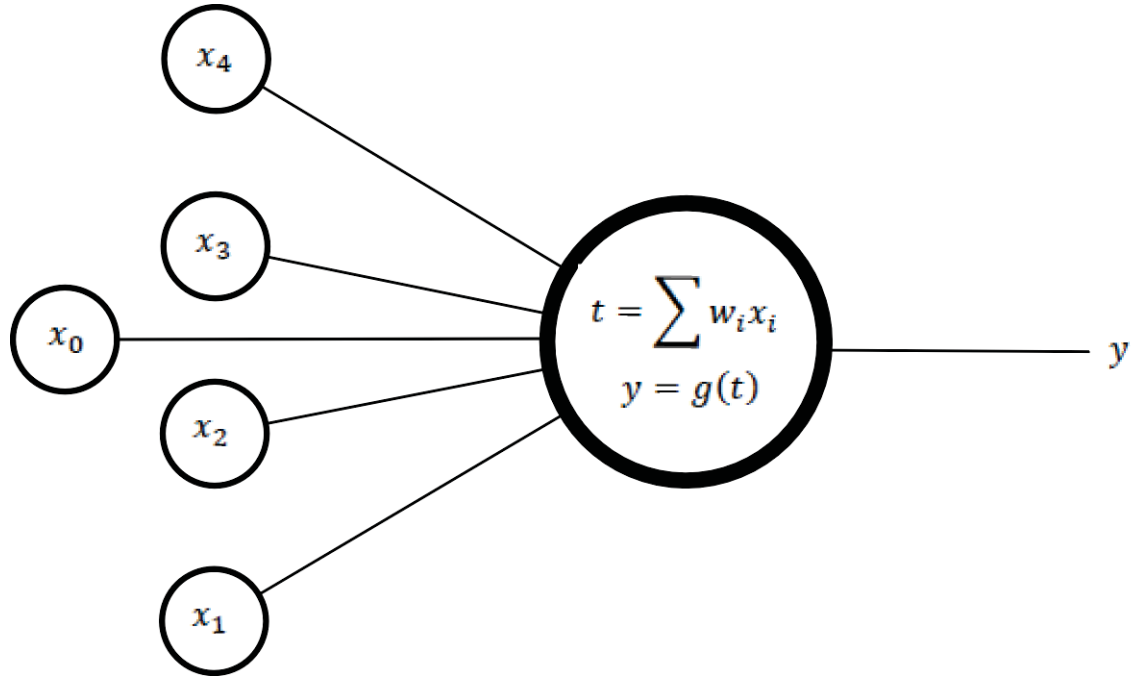


Figure 8 Example of how the inputs (x) are linked to the outputs (y) via the weights (w) and activation function (g).

The activation function g can take a number of forms, such as a step or a log-sigmoid function. For the purposes of this investigation the log-sigmoid activation function is used. That is $g(t) = \frac{1}{1+e^{\beta t}}$ where $t = \sum_{i=0}^n w_i x_i$ and β is the slope parameter.

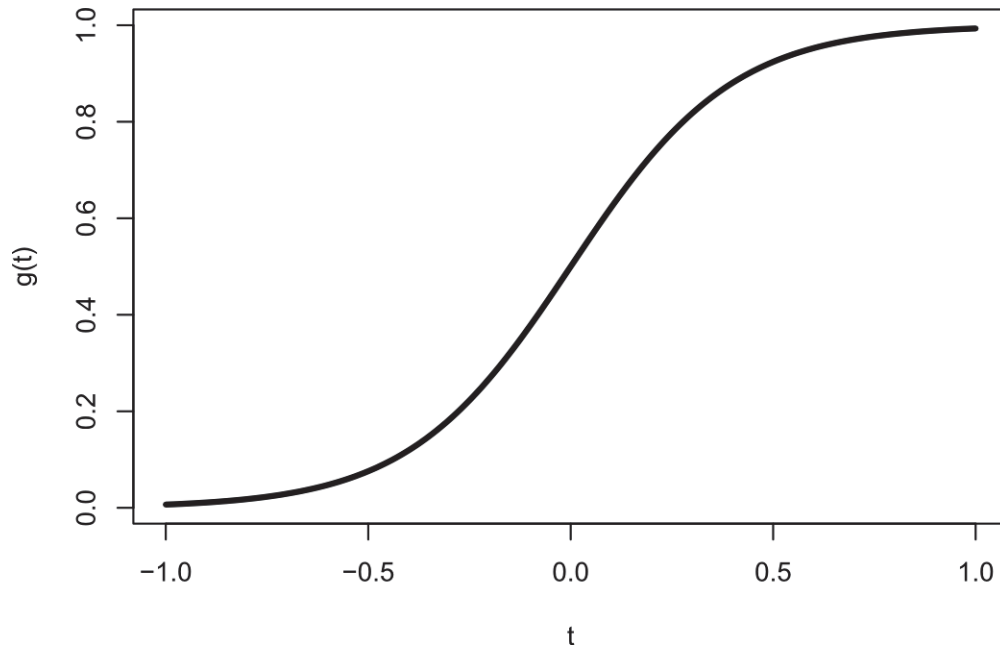


Figure 9 Example of a log-sigmoid activation function with $\beta = 5$

Once the network has been trained it is then tested on the remaining data to evaluate its ability to approximate the posterior estimates.

While generating the samples to fit the Neural Network it became apparent that by simply taking random design points it was not exploring the whole design space. For example taking a random sample of 8 from the standard uniform distribution is unlikely to result in all values less than 0.1 and so this design space is unexplored. In order to solve this problem a number of space filling designs were trialled. In particular Quasi-Monte Carlo sampling and Stratified Monte Carlo sampling (Giunta, Wojtkiewicz, & Eldred, 2003). The Quasi-Monte Carlo method appeared to be too difficult to implement although some progress was made by being able to generate the Sobol sequence needed for the process. In the end Stratified Monte Carlo sampling was preferred due to its simplicity in implementation. Stratified Monte Carlo meant dividing the n-dimensional design space into an n-dimensional grid and sampling from each cube. The result was a simulated dataset which explored the design space far better than the more standard Pseudo-Monte Carlo Sampling.

The advantage of the neural network approximation is that it greatly improves speed. The ffnet package used to train the neural network uses Fortran code for many of the calculations resulting in a speed up of over 100,000 over evaluating the same estimates using PyMC. The trade off is obviously accuracy. If the network does not approximate the MCMC estimates well it will result in poor optimal design estimates albeit produced quickly. Therefore the summary statistics of the neural network estimates versus the ‘true’ MCMC values as well as their plots were inspected to assess the suitability of the neural network approximation.

3.3.2.6 Bayesian Experimental Design with Discrete Priors

Once the Bayesian framework has been set up, one obvious approach is to select the signal intensities for the current experiment which minimise the expected loss function given the posterior predictive distribution. If we have an observed set of responses $Y = \{y_1, \dots, y_n\}$, then the posterior predictive distribution is the distribution that a new sequence of responses $\tilde{Y} = \{y_{n+1}, \dots, y_{n+m}\}$ at concentrations ξ would follow given Y . More formally

$$P(\tilde{Y}|Y, \xi) = \int P(\tilde{Y}|\theta, \xi)P(\theta|Y)d\theta$$

Where $P(\tilde{Y}|\theta, \xi)$ is the likelihood and $P(\theta|Y)$ the posterior distribution based on the observed data.

If no data has been collected the prior distribution $P(\theta)$ is used in place of the posterior. Next we define $\tilde{\theta}$ as the estimate of θ given observed data Y and unobserved data \tilde{Y} with design ξ . Note that the unobserved \tilde{Y} can be one of a finite number of possible sequences. By summing over all possible \tilde{Y} this leads to

$$P(\tilde{\theta}|\theta, \xi) = \sum P(\tilde{\theta}|\tilde{Y}_i, \xi)P(\tilde{Y}_i|\theta, \xi)$$

For a given loss function of θ and the posterior estimate $\tilde{\theta}$ the expected value given θ and ξ is

$$E[Loss(\tilde{\theta}, \theta)|\theta, \xi] = \sum Loss(\tilde{\theta}_i, \theta)P(\tilde{\theta}_i|\theta, \xi)$$

If θ is assumed to be one of a set of j finite values, that is θ has a discrete prior(s), then:

$$E[Loss(\tilde{\theta}, \theta)|\xi] = \sum_j \sum_i Loss(\tilde{\theta}_i, \theta_j)P(\tilde{\theta}_i|\theta_j, \xi)P(\theta_j)$$

$$E[Loss(\tilde{\theta}, \theta)|\xi] = \sum_j \sum_k \sum_i Loss(\tilde{\theta}_i, \theta_j)P(\tilde{\theta}_i|\tilde{Y}_k, \xi)P(\tilde{Y}_k|\theta_j, \xi)P(\theta_j)$$

In addition if observed data Y is available:

$$E[Loss(\tilde{\theta}, \theta)|Y, \xi] = \sum_j \sum_i Loss(\tilde{\theta}_i, \theta_j)P(\tilde{\theta}_i|\theta_j, \xi)P(\theta|Y)P(\theta_j)$$

$$E[Loss(\tilde{\theta}, \theta)|Y, \xi] = \sum_j \sum_k \sum_i Loss(\tilde{\theta}_i, \theta_j)P(\tilde{\theta}_i|\tilde{Y}_k, \xi)P(\tilde{Y}_k|\theta_j, \xi)P(\theta_j|Y)P(\theta_j)$$

This gives the means by which one is able to calculate the expected loss based on the discrete prior of θ , the likelihood function, and a set of design points ξ . The general procedure can be explained in the following steps:

1. Generate all possible response patterns given ξ . In general there will be 2^n such possibilities, however if there are repeated signal intensities this number can be reduced. For example if there are r repeated tests at each of m levels then the number of possible responses can be reduced from 2^{mr} to ${}^mC_{r+1}$.
2. For each value of θ possible under its discrete prior(s) calculate the loss function for each of the posteriors based on the pseudo responses.
3. Calculate the probability of detection p_{lj} at each of the l^{th} level (x_l) of ξ for every possible $\theta_j = \{m_j, w_j\}$ using the psychometric function. For the logistic psychometric curve

$$p_{lj} = 1/3 + 2/3 \left(1 + \exp \left(-\frac{x_l - m_j}{w_j} \right) \right)^{-1}$$

4. Calculate the likelihood of observing each sequence \tilde{Y}_k given θ_j as the product of the binomial probabilities for each l^{th} level of ξ :
$$P(\tilde{Y}_k | \theta_j) = \prod_l p_{lj}^{\tilde{y}_{lk}} (1 - p_{lj})^{n_l - \tilde{y}_{lk}} \binom{n_l}{\tilde{y}_{lk}}$$
5. Multiply the each loss by its likelihood and then multiply again by the respective prior, or the posterior if observed data is available, probability of θ_j .
6. Take the sum of the weighted losses as the expected loss for design ξ .

While the process is reasonably straightforward it does require a large number of calculations especially if ξ has a reasonably large number of unique levels, or if the discrete prior distribution(s) have a large number of levels. For example if it is desired to set the next 20 design levels there are $2^{20}=1,048,537$ possible responses multiplied by the number of levels of the prior(s) loss function evaluations which can become prohibitively large. Hence this investigation only looks at designs where the number of levels is 8 or less. The most complicated design used in the simulations involved four repeats of 8 levels giving ${}^8C_5 = 390,625$ possible responses.

One could instead use a continuous prior and try and estimate the expected loss function analytically, however the integrals involved are intractable. Another option would be to use the MCMC techniques previously mentioned to get an approximate solution. However as Vanlier, Tiemann, Hilbers, & van Riel (2012) point out such a nested process may not be tractable due to the often computationally expensive MCMC methods. While they avoid the problem by estimating an approximate solution via importance sampling a different approach is taken here. The first attempt to overcome this problem was to use a neural network to provide a fast approximation to the MCMC estimates.

The next attempt described here was that instead of using continuous distributions; the prior beliefs are specified by discrete approximations to reduce the computational burden. In general these priors for both the threshold and width parameters are discrete uniform of n equally spaced levels over the $[0,1]$ interval. Even if the discrete priors can be considered approximations to the continuous priors they have the advantage over the MCMC based methods of giving exact loss estimates. In particular the sample posterior entropy appeared to be rather unstable under MCMC simulations and therefore optimizing based on this value can be improved by using the exact results.

3.3.2.7 Approximate Entropy of the Posterior Distribution

As mentioned previously there are a number of adaptive design schemes which rely on the selecting stimulus levels to minimise the entropy of the posterior. When using the discrete priors the entropy can be calculated exactly but for the MCMC based posterior estimate this is more difficult. To get an approximate entropy estimate, threshold posterior distributions were divided in 10 bins of equal width. The proportion of the MCMC samples in each of the i bins, P_i , was then calculated. Finally the entropy, H , of both the discretised distributions and the posteriors based on discrete priors were calculated as

$$H = - \sum_i \log_2(P_i^{P_i})$$

This is a slightly different parameterisation from the more standard:

$$H = - \sum_i P_i \log_2 P_i$$

The reason for this is if $P_i = 0$ then in python, $0 \log_2 0 = -\infty$, while $\log_2(0^0) = \log_2(1) = 0$, which is the standard definition.

Additionally for the discrete prior method the numexpr library was used to speed up many of the matrix calculations. Numexpr does not include a \log_2 function and therefore the natural log was used instead. The impact of this change is minimal as H changes only by a constant depending on which base the logarithm uses and therefore the minimum entropy still occurs at the same design points as if \log_2 had been used.

The entropy values were stored together with the other MCMC summary statistics.

The Shannon entropy is an important measure in that minimising the expected entropy of the posterior through choosing design ξ can be thought of as maximising the expected gain in posterior information.

3.3.3 Approximating the Loss Function by Simulation

To estimate a particular loss function for a given set of concentrations one strategy would be to evaluate the threshold estimates for all possible responses and calculate the expected loss by weighting the estimates by the probability of occurrence assuming the underlying model.

However if the number of concentrations is large the number of threshold evaluations needed can be prohibitively large. In this case an alternative strategy can be employed. Here a number of responses are generated at the current stimulus levels assuming the underlying model. Threshold estimates are calculated for each of the response patterns and then the loss function calculated. This will give a reasonable approximation to the loss function with some noise due to the random sampling.

3.4 Loss Function Minimisation

Once a practical method for estimating a given loss function has been found the problem now shifts to that of finding an optimum configuration of stimulus levels so as

to minimise the loss. If the concentration levels are first restricted to take only values over the range of interest, $[0,1]$ on the rescaled, then the candidate concentrations for the next sample may take any real values within this interval. Alternatively, and perhaps more practically, the candidate concentrations may be also restricted to belonging to a finite set of stimulus levels. For example only concentrations of 0.1, 0.2, 0.3,...,0.9 are available to the researcher.

3.4.1 Minimisation with Continuous Stimulus Levels

There is a wealth of algorithms available in Python and its various packages to solve optimization problems. The OpenOpt library in particular provides a great number of optimisers for various problem types. However the interface to hook in with the objective function was problematic and therefore OpenOpt was not used in the final analysis. Instead a number of other solvers were used. Firstly Powell's minimisation algorithm (Powell, 1964) implemented in the Scipy.optimize library was used for the minimisation of the majority of the adaptive methods presented here. Many of the other optimisers available performed similarly but as noted in a discussion on the PyMC Google Groups site, Powell's method can be superior at finding the MAP of the posterior distribution.

Initially when there was an error in the code to calculate the expected loss the solver would derive a solution which contained design points far outside the $[0,1]$ interval. To force the solution to lie within the interval the solver was changed to the box constrained Broyden-Fletcher-Goldfarb-Shanno L-BFGS-B method (Zhu, Byrd, Lu, & Nocedal, 1997). However once the bug was found and fixed the box constraints were no longer necessary and Powell's method was again implemented as it has the advantage of being derivative free which could be an advantage if the loss function is not smooth. Powell may still provide a solutions outside $[0,1]$ however small discretions are allowable due to the possibility of the 'true' threshold being outside this for the logistic psychometric curve. The Weibull curve on the other hand does not allow concentrations less than zero so if this psychometric curve was assumed then the L-BFGS-B solver may be preferable.

For the training of the neural network the TNC optimiser was used as it allowed for a speed-up by allowing the function evaluations to be spread across multiple cores. The

TNC optimiser, which is also available in Scipy is a version of the Newton Conjugate-Gradient modified to allow for box constraints.

At any rate the choice of minimisation routine is rather arbitrary and therefore the python code is written to allow for the choice of non-linear solver to be changed.

Another class of solvers, called global solvers, were also implemented. One potential drawback for the non-linear solvers above is that it is possible for the algorithm to find a solution at a local rather than the global minimum. Global solvers overcome this at the price of speed. The Particle Swarm (PSwarm) and Genetic Algorithm are two examples of global solvers used in this investigation.

PSwarm involves setting a number of individual “particles” randomly within the design space and evaluating the loss function. Then each particle moves at a set “speed” with the direction based on the position of the “best” particle and the best known position that the individual particle has visited. Once the particles have finished moving the function is again evaluated and the process is repeated for a set number of iterations. The final solution is position with the minimum (or maximum) function value which was explored by any of the particles during the process.

The Genetic Algorithm mimics biological evolution by starting with a number of individuals within the design space. Each individual has a number of “genes” which are the design points from which the objective function is calculated. A number of these individuals are selected based on their respective “fitness” (loss) functions and are grouped into pairs. Each pair produces “children” which form a new generation of individuals. The children are generated by selecting one of each pair of genes from its parents. This process is repeated for a number of generations with a solution being the “fittest” individual produced over a set number of generations. The experimenter can also tune this process by adjusting the cross-over and mutation parameters.

3.4.2 Minimisation with Discrete Stimulus Levels

While the majority of this investigation deals with trying to find the optimal design points, $\xi \in \mathbb{R}$, due to restriction the investigator may only be able to select ξ from a finite set. In this situation the most straightforward option is to round the optimal

concentrations to the nearest possible signal intensity. However this rounded solution is not guaranteed to be approximately optimal.

A better approach would be to round the inputs to the nearest possible concentration before they are input into the loss function. This has the effect of making the objective function non-smooth. Inputs which are real numbers and close in magnitude will be rounded to the same value and hence produce the same loss. This non-smoothness can affect the performance of the non-linear solvers leading to the false appearance of convergence. The global solvers, however do not have this limitation, and are suitable for the task. As the problem of selecting a design from a finite set of possibilities is not the focus here, the code is only provided to demonstrate that it is possible to derive an approximate optimal set of design points from a finite set. In addition a brief number of simulation results are also provided.

3.4.3 The Adaptive Procedure

Based on the discrete prior approach the python code also contains routines for finding an optimal design based on the prior distributions as outlined above. Once data has been collected based on this design the posterior distribution is calculated and then is used as the new prior for the next set of design points. As the posteriors are calculated exactly this is allowable since $P(\theta|\tilde{Y}, Y) = P(\tilde{Y}|\theta)P(Y|\theta)P(\theta) = P(\tilde{Y}|\theta)P(\theta|Y)$. Which is equivalent to calculating the posterior distribution for θ based on data \tilde{Y} with prior $P(\theta|Y)$. This process is repeated until a set number of samples have been collected and the summary statistics based on the final posterior distribution are returned. Most of the simulations presented here involve designs of size 8 being adaptively fit for an individual curve 4 times. It is as Kelareva et al., (2010) would refer to as a look ahead 8 design, as it looks ahead 8 steps to derive the optimal design. Hence the 8-step look-ahead adaptive designs compared here will be referred to as MINENT8, MEAN8, MEDIAN8, and MODE8. The total number of tests at the end of the 4 blocks of 8 samples is 32. The reason 8 was chosen is due to the number of tests used on the original PFR study as more than 8 could possibly lead to olfactory fatigue. Some other versions of these with the same number of total tests (32), were also simulated for comparison. These were a design with 4 repeats of the same 8 design points, and another which involved looking ahead only 1 point, which are

similar to the procedures MINENT, MEAN, MEDIAN, and CATERR (Kelareva et al., 2010). The MEAN, MODE, and MEDIAN procedures may differ slightly as they explicitly set the next design point to the mean, median, and mode of the current posterior respectively. The method used here however, minimises the square, absolute, and categorical error by summing over the posterior distribution. In theory this should lead to a similar outcome to the MEAN, MEDIAN, and MODE. A look ahead 32 design was not possible with this method as it required far too many possible samples (2^{32}) to evaluate over.

These routines allow for four different loss functions: Square, Absolute, and Categorical error as well the Entropy. The square error is calculated as the squared difference between the true threshold and the posterior mean, the absolute error as the absolute difference between true threshold and the posterior median, and the categorical error as 0 if the posterior mode equals the true threshold, 0 otherwise. The reason these point estimates were used for the respective loss function is that they have been shown to minimise said losses (Kelareva et al., 2010).

The discrete priors were uniform in probability with a set number of levels. The number of these levels was varied to trade off between accuracy and speed. The simulations presented in the results were based on uniform independent priors between the interval [0,1] for both the threshold and width parameters. The number of levels for the majority of simulations was 11, which gave sequences from 0 to 1 with a step size of 0.1. This resulted in a prior distributions where there were 121 possible logistic curves all with equal probability. Using a low number of levels meant that the function evaluations were very quick and hence why it was used to run the large number of simulations. It is however possible to increase the number of levels and one simulation comparing the minimum entropy procedures with look ahead 1 and 8 uses priors with 100 equally spaced points between 0 and 1, resulting in 10,000 possible curves under the prior distribution.

When comparing the performance of the different adaptive techniques responses from simulated individuals were randomly sampled. In order to reduce the sampling variation amongst the different techniques responses were made to be consistent for

each individual across different designs. This was done for each individual by taking a random uniform(0,1) sample of length equal to the number of samples needed. This simulated response was then generated by taking the inverse of the cumulative binomial distribution given n and probability equal to the underlying psychometric function evaluated at design ξ . The random sample is used for all simulations relating to that individual but is re-generated for each subject. This means, for example, that if a simulated individual gives a correct response at concentration .5 for the first design point, then any concentration greater than 0.5 for the first design point, generated under the other schemes must also be correct. This does not hold for values less than 0.5, or across design points e.g. the second design point at a concentration of .5 may be incorrect.

3.4.4 Python code

The majority of python code used in the final analyses is available in the appendix, much of which is custom written with the exception of a function used to generate De Bruijn sequences. This function was freely available on the De Bruijn sequence entry in Wikipedia (en.wikipedia.org/wiki/De_Bruijn_sequence). The verbatim code in question is listed on the next page:

```

def de_bruijn(k, n):
    """
    De Bruijn sequence for alphabet size k
    and subsequences of length n.
    """
    a = [0] * k * n
    sequence = []
    def db(t, p):
        if t > n:
            if n % p == 0:
                for j in range(1, p + 1):
                    sequence.append(a[j])
            else:
                a[t] = a[t - p]
                db(t + 1, p)
                for j in range(a[t - p] + 1, k):
                    a[t] = j
                    db(t + 1, t)
    db(1, 1)
    return sequence

print(de_bruijn(2, 3))

```

The De Bruijn sequence was used to generate all possible responses by repeating the sequence with some reordering and reshaping it into a two dimensional array.

Much of the code is nested within functions which not only are more efficient (as it allows the same routine to be used multiple times succinctly) but it also improves the memory management. When first running the MCMC routines multiple times the memory usage continued to increase until it was full causing python to crash. This was due to the way in which python handles memory. Simply deleting an object does not remove it from memory, instead the reference count must reach zero before the system will release the memory back to the system. By creating objects such as large arrays within a function and only returning summaries of these arrays, the memory used to store them should be released once the function has terminated.

A lot of the routines used in the discrete prior method to search for the optimal design involved array manipulation as it is computationally more efficient than using loops. With some of the larger problems these arrays became too large to store in memory

and therefore were split into smaller arrays which were fed into the loss function separately and then the solutions recombined.

4 Results

4.1 Fitting Bayesian Models through MCMC

Logistic psychometric curves were fitted using PyMC with a burn in of 1000, thinning of factor of 10 and 10,000 as the total number of iterations. Prior distributions for both the threshold and scale parameters were $\text{beta}(1,1)$, which is equivalent to the standard uniform distribution. For a small sample of the simulations autocorrelation plots were inspected and deemed to be satisfactory. Under these settings a model with based designs ξ of lengths ranging between 8 and 40 points with respective binary outcomes took between 1.89 and 2.15 seconds on an Intel Xeon E5530 8 cores @ 2.4 GHz with 24 gigabyte of ram running Kubuntu 10.04, to evaluate the posterior and return various summary statistics.

While this seems a reasonable time frame, in order to carry out any sort of optimisation routine this process must be repeated a number of times. For example if there are 8 design points to be set then there are 256 possible response patterns which need to be evaluated for the expected loss to be calculated at each configuration of ξ . If there is roughly 200 loss function evaluations needed for the local solver to converge to a solution then the total time taken would be approximately 27 hours. For many practical situations this would be too long to be of any use. Therefore optimising based on a neural network approximation of the posterior estimates was investigated as a means of improving convergence speed.

4.1.1 Generating data for Neural Network Training

The first step in training the neural network is to generate the data. Firstly a 25,008 by 40 matrix of random standard uniform variables was generated. Each row corresponded to a design configuration to be used in the data simulation. As stated in the methods this did not explore the design space adequately. For example, designs with all placements less than 0.25 were virtually nonexistent as the probability of such an occurrence $= .25^{40}$. Therefore the designs were rescaled to lie within 1 of a possible 16 sections of $[0,1]$, taking up proportions of $1/4, 1/3, 1/2$, or the whole of the interval. Next 25,008 threshold and scale parameters were generated using the standard uniform prior distributions. Initially 25,000 was chosen as the number of scale and threshold parameters to generate in the interest of time while still providing

enough samples to train and test the NN's. This was increased slightly to 25,008 to allow for equal numbers to be sampled from each of the 16 sub-intervals. For each simulated design predicted probabilities of detection based on the logistic psychometric curve with the matching simulated model parameters were calculated and Bernoulli samples based on these taken. For each of these simulated samples a number of sub samples were taken and the posteriors estimated by MCMC methods as described above, with the posterior entropy, mean, mode, median, and standard deviation for both the threshold and scale parameters being stored. The sub samples were the first 8, 16, 24, 32, and all 40 observations in each sequence. The reasoning for this is to be able to estimate 8-step look-ahead loss functions given observed samples. The total time taken to generate this training data was approximately 3 days.

4.1.2 Training the Neural network

The structure of the neural networks involved having 4 times as many hidden nodes as there were input nodes, with one output node. Initial attempts treated the sampled responses and design points as separate inputs however for the neural network to recognise that each response related directly to an individual design point would require a complicated specification of the relationships amongst certain input nodes. Therefore the two parts of information were combined into a single input sequence by changing the sign of the design point to negative if the response is 0, or leaving it positive if the response is 1. If there were originally 8 design points with 8 responses the new input variable has length 8 rather than 16, and so on. The inputs were also sorted on their absolute value so that they range from the smallest concentration through to the largest. This helps to avoid inconsistent estimates where the same inputs produce different outputs based on their order.

A random sample of 20,000 of the 25,008 generated dataset was used to train the neural networks with the remaining observations used to test the model fit. The fitting process involved using python's own TNC solver with 5000 iterations and utilising all 8 cores. This was repeated for each outcome measure and for the samples of length 8, 16, 24, 32, and 40.

4.1.3 Neural Network Performance

The trained neural networks perform reasonably well when predicting the mean and median of the posterior distributions especially when the number of inputs is low. The mode, standard deviation of the posterior (not shown), and the entropy prove more difficult to approximate. Scatter plots of the MCMC estimates based on 8 observations vs. the neural network approximations for the 5008 samples in the test dataset are shown in Figure 10, with the mean having a tight fit around the $y=x$ line.

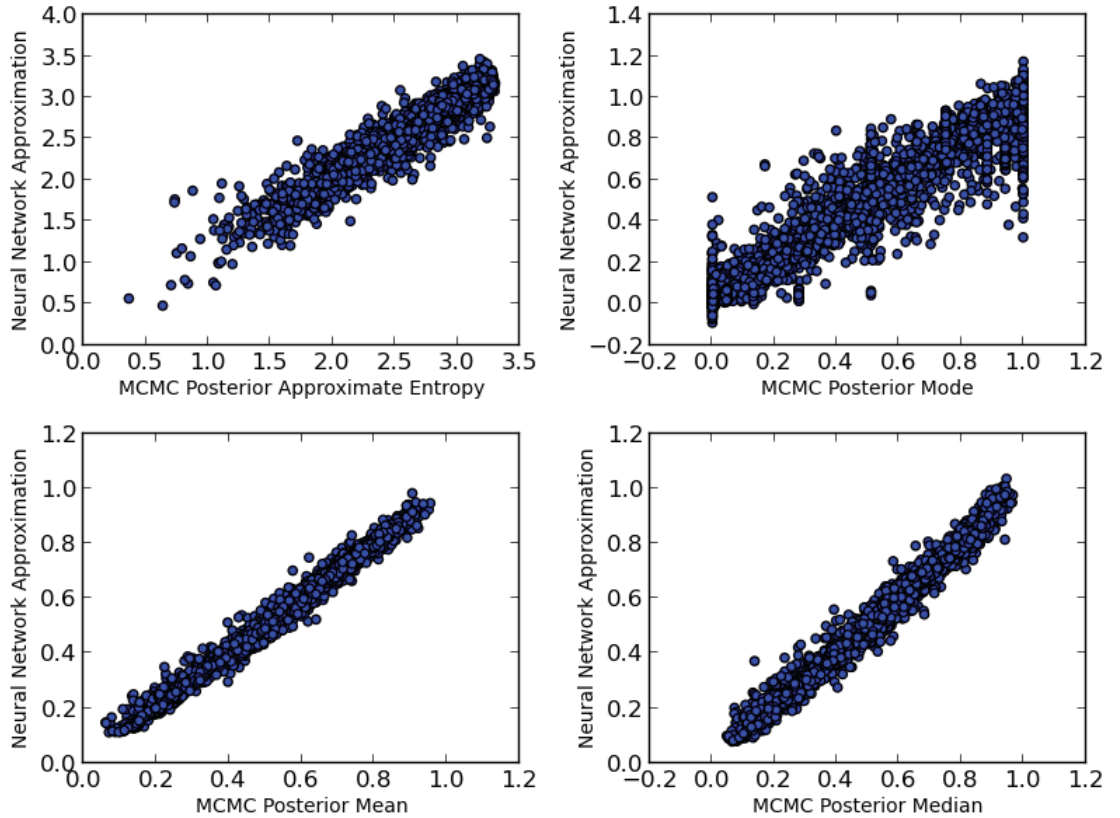


Figure 10 Scatter plots of the MCMC estimates based on 8 observations vs. the neural network approximations

The results from fitting a simple linear regression model with the NN approximation regressed onto the MCMC fits for the same test dataset are shown in Table 3. Note that an intercept of zero and a slope of one would indicate a perfect fit. The regression summary statistics (Table 3) agree with Figure 10 in that the mean shows the best neural network approximation with intercept and slope parameters close to 0 and 1 respectively. It also has the lowest standard error of the estimate, which is a measure of how much the observed data deviates from the predictions of the regression model.

Table 3 Regression Summary Statistics for Neural Network Approximation Based on samples of Length 8

	Entropy	Mode	Mean	Median
Intercept	0.1824	0.0309	0.0047	0.0065
Slope	0.9300	0.9204	0.9914	0.9870
r	0.9623	0.9576	0.9953	0.9923
Slope SE	0.0037	0.0039	0.0014	0.0017
Estimate SE	0.1166	0.0914	0.0199	0.0273
N	5008	5008	5008	5008

Based on this it appears that the NN approximates the posterior mean reasonably well, at least when the number of observations to base the estimate on is small, which in this case is 8.

The NN was also used to approximate the MCMC estimates based on 16, 24, and 32 observations. As the number of samples increased the quality of the NN approximations deteriorated, however the rate of decline differed amongst the MCMC measures. Figure 11 shows the scatter plots for the NN based on samples of length 32 from the test dataset. It is obvious the NN entropy approximation is unacceptable and appears to almost be random. The mode approximation roughly follows a 1 to 1 trend however it has a large variance around this. One explanation could be that the NN is over-fitting the training dataset but it seems unlikely as the mean and median have reasonable approximations. For these there are some points that have obviously not been predicted well, but they are few, and the remaining points are grouped around the one to one line reasonably closely.

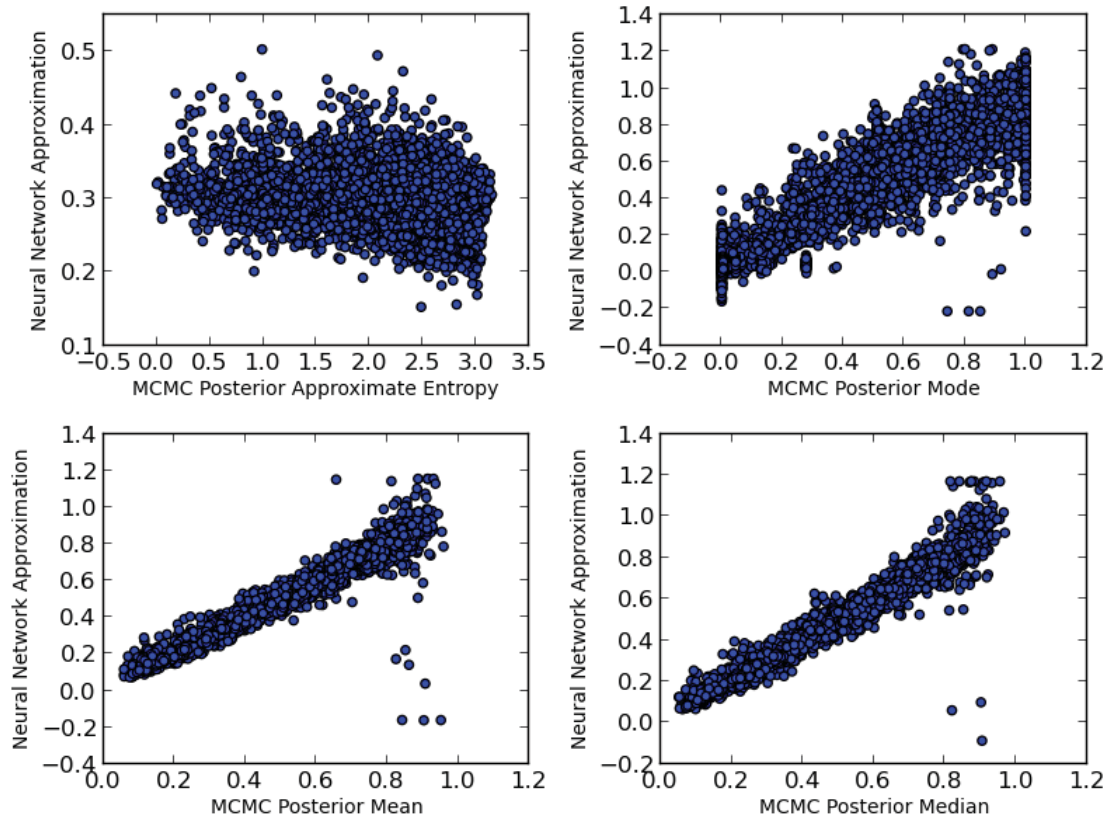


Figure 11 Scatter plots of the MCMC estimates based on 32 observations vs. the neural network approximations

The regression summary statistics in Table 4 shows the median having a slightly superior fit when compared to the mean, however the difference is negligible.

Table 4 Summary statistics from regressing the MCMC point estimates on the NN approximations based on 32 observations per sample.

	Entropy	Mode	Mean	Median
Intercept	0.3257	0.0452	0.0098	0.0062
Slope	-0.0148	0.8892	0.9777	0.9871
r	-0.2407	0.9232	0.9755	0.9824
Slope SE	0.0008	0.0052	0.0031	0.0027
Estimate SE	0.0412	0.1148	0.0507	0.0441
N	5008	5008	5008	5008

Based on these it appears that neural networks can give reasonable approximations to the means and medians of threshold posterior distributions fitted by MCMC methods. As expected, predicting from the NN provides an increase in speed when compared to

an MCMC based approach. It took 0.07 seconds to estimate the posterior mean approximations using the neural network for 25,008 samples each containing 8 samples, whereas evaluating them by MCMC techniques used here took approximately 13 hours, which is a speedup of over 650,000 times. While there are a number of possibilities for improving the time to complete the MCMC simulations it is unlikely to get fast enough to be comparable to the neural network in terms of speed.

As the mean can be reasonably approximated by the fitted NN's, the approximations could potentially be used in an attempt to find an optimal design. Testing this theory involved simulating data using parameters drawn from the prior distribution, calculating the NN approximated threshold estimates and the respective MSE of these compared to parameters from the prior. Alternatively the median could have been used in a similar way to minimise the absolute error loss function. Scatter plots demonstrating the reasonable quality of the NN approximations of the posterior means are shown in Figure 12 for samples of size 8, 16, 24, and 32.

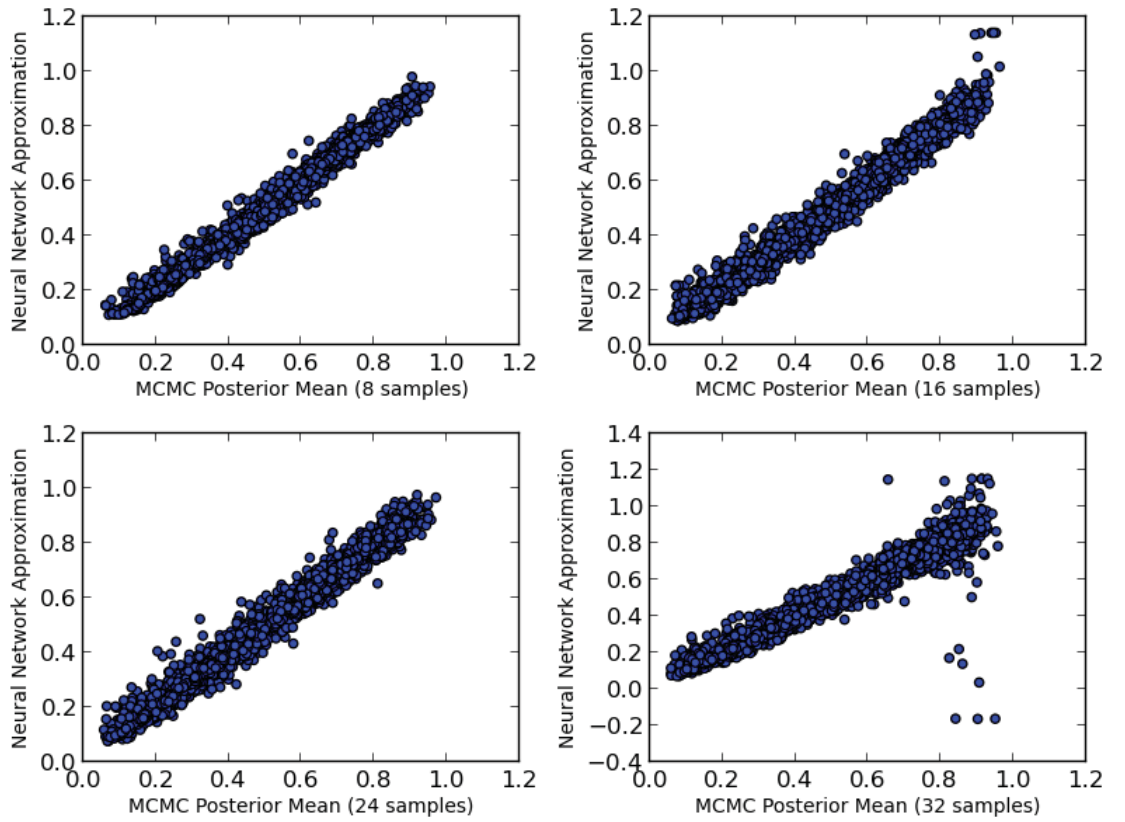


Figure 12 Posterior mean NN approximations versus MCMC means based on 8, 16, 24, and 32 samples.

4.1.4 Optimizing based on the Neural Network

In order to find an approximately optimal design the first problem is to estimate the loss function. To do this 11 threshold values equally spaced ranging from 0 to 1 and a similar sequence for the scale parameter were generated. For each of the 121 possible threshold and scale combinations 100 Bernoulli samples were drawn with probabilities calculated using the logistic psychometric function with a given design ξ . In order to make the samples consistent across different iterations, first a random sample of standard uniform values were drawn, and the Bernoulli samples taken as the quantiles of the inverse of the cumulative Bernoulli distribution at the probability relating to the respective uniform samples. This corresponds to setting the Bernoulli samples to 1 if the standard uniform sample is greater than 0.5, 0 otherwise. Calculating the samples this way removes the random sample variation between iterations. If the same design is used twice it will give the same loss. For each sample the NN approximation and its squared difference with the generating threshold was calculated. The mean of the squared differences was used as an estimate of the MSE.

Starting with the NN for samples of length 8, various optimization schemes were used to try and adjust ξ to minimise the MSE. The first attempt used Powell's minimization algorithm and converged to a solution after 208 function evaluations. Unfortunately the solution involved a design where some points were greater than 1 which is outside of the range used to train the NN. As any NN estimate based on inputs outside its support could be unreliable the solution must be discarded. The next attempt was to use a L-BFGS-B boxed bound solver which constrained the solution to lie within the range $[0,1]$. However this method often failed to converge, perhaps due to the function being non smooth or the solution lying outside the bounds.

Finally the particle swarm global solver was used. The loss function was slightly modified to return a large constant if any design point was outside $[0,1]$ so as to force the particles to explore within this interval. The search space was explored by 10 particles travelling over ten iterations. Since the PSwarm algorithm is a heuristic method it is difficult to assess convergence to a minimum. Increasing either the number of particles or the number of iterations could improve the solution however in the interest of time ten of each was deemed sufficient.

The performance of the adaptive method based on a NN approximation was assessed by simulating 100 individuals for each of 9 different logistic psychometric curves (Figure 13)

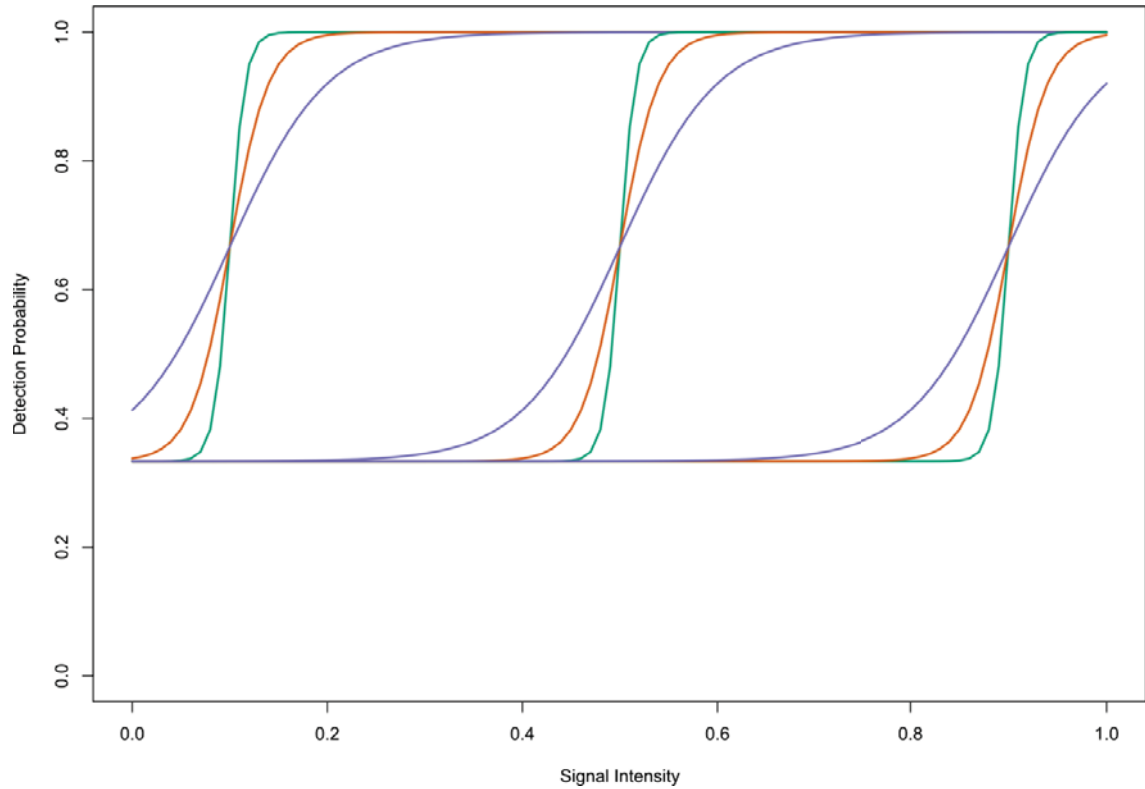


Figure 13 Logistic Psychometric curves used to generate responses to assess the performance of the NN based adaptive method.

For each individual an initial optimal design was used for the first 8 signal intensities. Simulated responses were then generated using this information together with the respective psychometric curve. These simulated responses were then used to find the minimal MSE design for the next 8 simulated responses. This was repeated until there a total of 32 design points and responses had been generated and a final threshold estimated by the NN had been calculated. The MSE was calculated for each of the nine psychometric curves as the mean of the square differences between the individuals “true” thresholds and the NN estimates. The results are presented in Figure 14 with the MSE converted to the Root Mean Square Error (RMSE). The colours span the range of values from purple for the minimum through to dark red for the maximum.

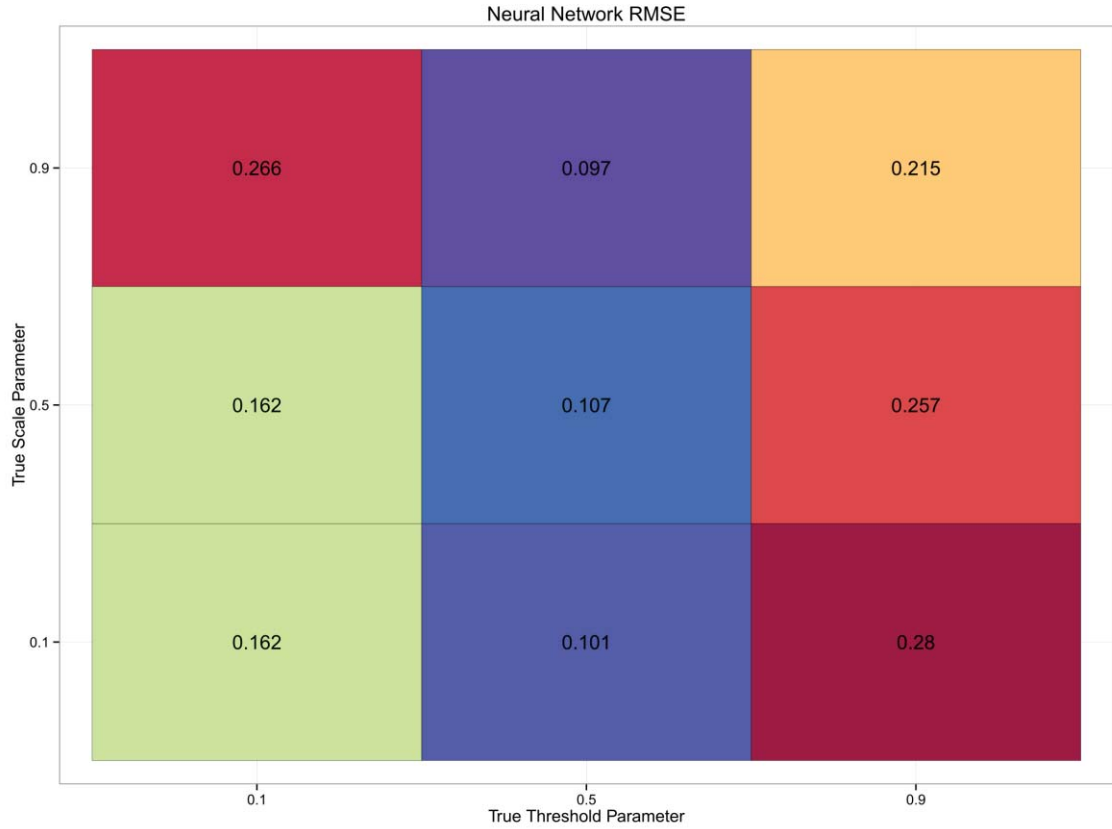


Figure 14 Estimated RMSE for the NN Adaptive Design based on 100 simulations of 32 samples for Logistic Psychometric curves with varying Threshold and Scale parameters

Over all of these minimization attempts it appears that while the NN can adequately predict the posterior mean and median, minimizing a loss based on these is very difficult. The global solvers appear to be more appropriate as they can be restricted to search in the appropriate space and are less susceptible to the non-smoothness of the objective function. The addition of the noise due to approximating the MCMC thresholds also appears to have the detrimental effect on the adaptive design with reasonably large RMSE's even for psychometric curves with small scales where theoretically it should perform well. It also appears that the adaptive design performs poorly when the true threshold is located at the extremes of the $[0,1]$ interval. With this in mind and the fact that generating enough samples to train the NN's is computationally expensive, an alternative optimization method was sought.

4.2 D-Optimal Designs

Custom python code was written to derive the d-optimal designs for a standard logistic regression model assuming a given intercept and slope parameters. The results

provided from this approach compared favourably with those found at <http://optimal-design.biostat.ucla.edu/optimal/polynomial/LogisticD.aspx> using the online calculator. Both methods produced the same 2-level designs, while the python algorithm provided a more efficient 3-level design as it did not require the design points to be equally spaced.

The next step was to alter the code to make calculations based on the logistic psychometric curve likelihood function. This proved to be problematic as often during the minimisation process the inverse of the hessian would not be positive definite leading to a non-optimal solution. A check was introduced to see if the inverse hessian matrix had any negative eigenvalues as this would indicate a non-positive definite matrix, and if so return a large value. However this still did not lead to consistent results with errors being common. Therefore this approach was abandoned in favour of the discrete prior method.

4.3 Bayesian Optimal Design using Discrete Priors

Another approach to increase the speed of deriving an estimate of the posterior distribution is to replace the continuous prior distributions with discrete approximations. In this investigation the discrete priors were restricted to discrete uniform priors equally spaced between 0 and 1. The number of levels for each prior was eleven for most comparisons which gives 121 possible psychometric curves under the prior (Figure 15). Increasing the number of prior levels was possible, and optimal designs based on priors ranging from 0 to 1 with step sizes 0.01, which gives 10,201 possible logistic curves, were successfully found in a reasonable time frame. However since the evaluation of the procedure involved many simulations, in general the 11 level priors were often used in the interest of time.

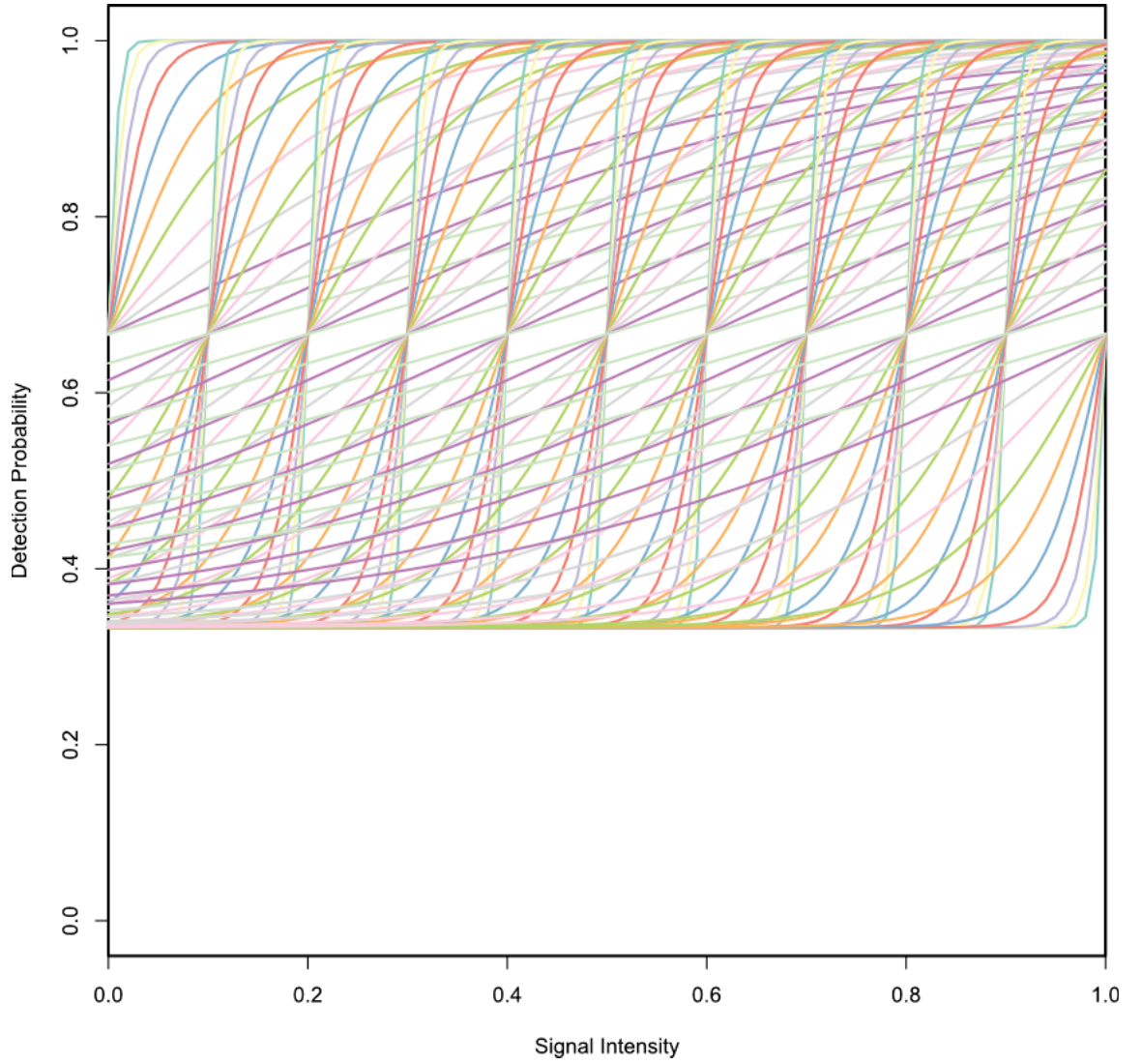


Figure 15 Logistic Psychometric curves under discrete standard uniform priors with 11 equally spaced levels for the threshold and scale parameters.

4.3.1 Comparison of Adaptive Schemes

The general method for the adaptive scheme is described in the methods section. Here we compare the adaptive methods based on four loss functions; entropy, absolute error, squared error, and categorical error. The absolute error was defined as the expected difference between the true threshold and the posterior median as that quantity has been shown to minimise the absolute error for a given posterior distribution (Kelareva et al., 2010). Similarly the squared error was the expected squared difference between the true threshold and the posterior mean, and the expected categorical error was the probability that the posterior mode did not exactly equal the true threshold.

The comparisons are based on the 121 psychometric curves show in Figure 15. Firstly optimal designs of length 8 were found for each method based solely on the prior information. Then for each design 100 Bernoulli samples were generated for each curve under the priors. These represent $100 \times 121 = 12,100$ individuals. Next the optimal designs for the next 8 samples were found using both the prior and simulated observations. This was repeated two more times with each optimal design derived assuming all previously simulated observations for that individual. A final set of Bernoulli samples are then taken for the final design points and the posterior generated to give the mean, median and mode estimates based on a total of 32 observations. With these posterior estimates of the threshold the final squared, absolute, and categorical errors were calculated. The results are tabulated in Figure 16, Figure 17, and Figure 18 with the Root Mean Square Error (RMSE) presented in place of the MSE.

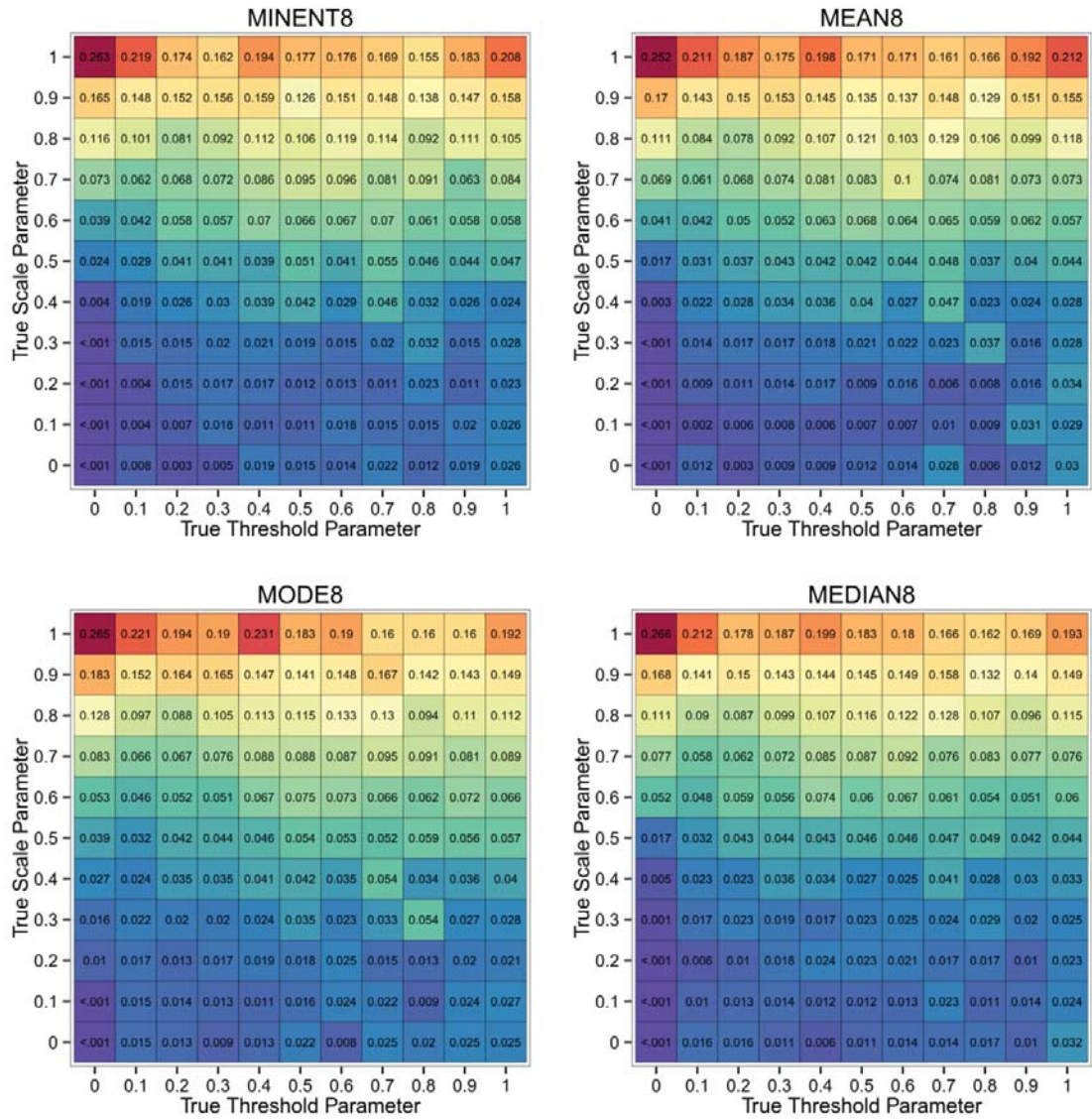


Figure 16 RMSE based on 100 samples generated by Logistic Psychometric curves of varying scales and thresholds.

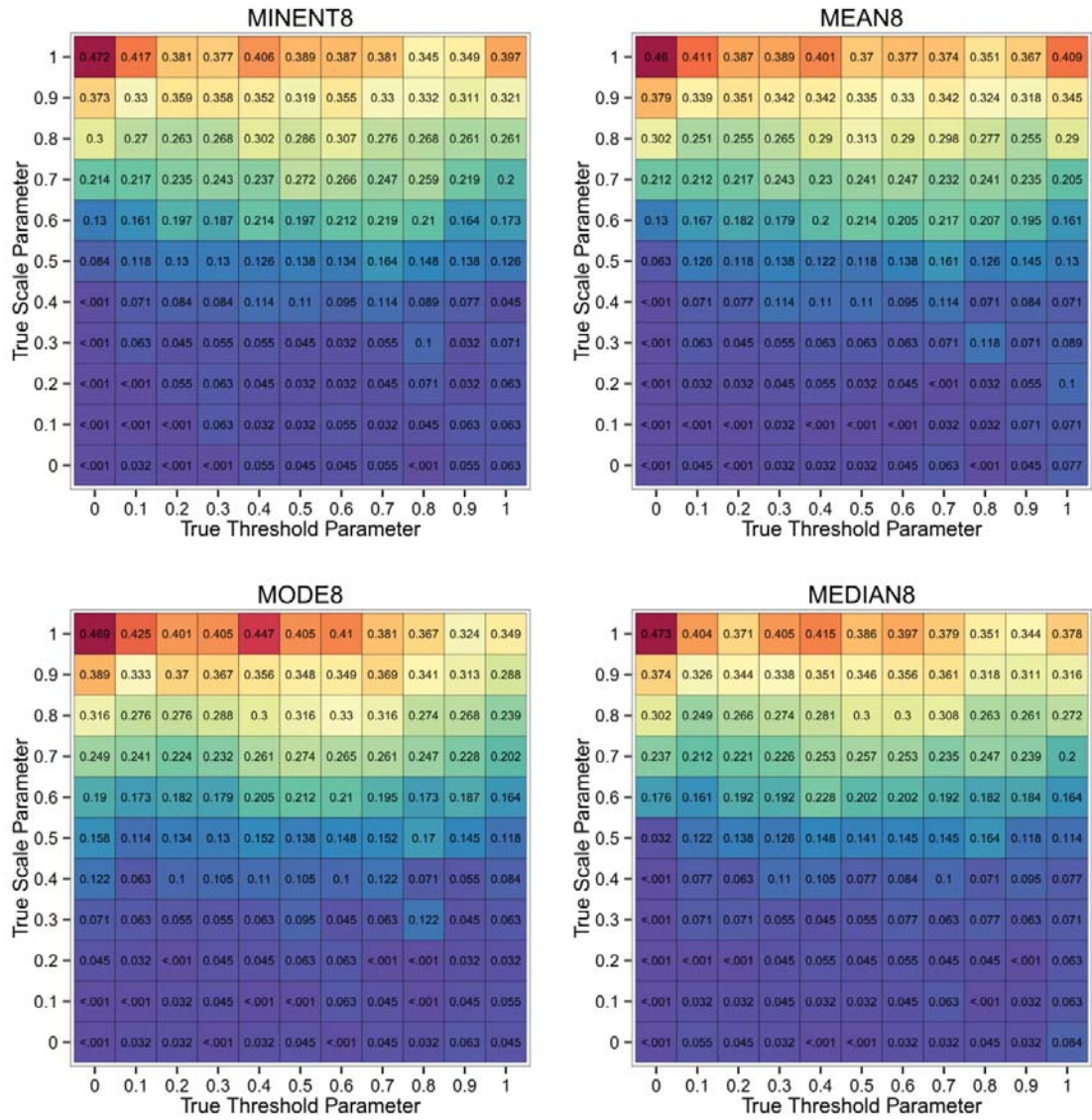


Figure 17 Mean Absolute Errors based on 100 samples generated by Logistic Psychometric curves of varying scales and thresholds.

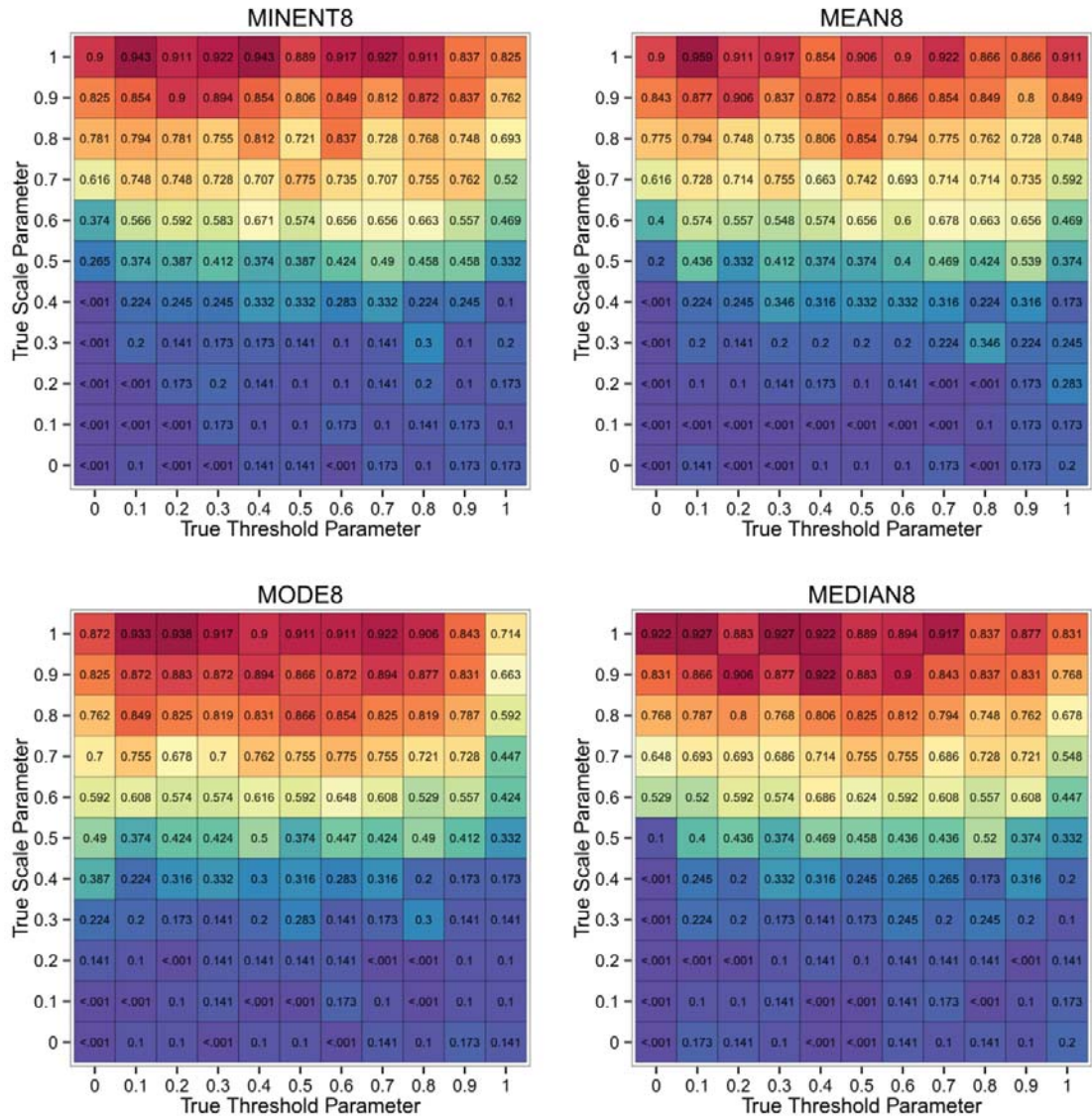


Figure 18 Categorical Errors based on 100 samples generated by Logistic Psychometric curves of varying scales and thresholds.

The figures show all four methods had similar performance in terms of the squared, absolute and categorical loss functions. Comparing the RMSE's with those of the NN based approach shows that discrete prior techniques provide a clear increase in performance with a large reduction especially for small scales or thresholds close to 0 or 1. There was some evidence that the MODE8 method produced slightly worse RMSE and Absolute errors when compared to the MEAN8, MINENT8, and MEDIAN8. This was based on paired t-tests over all 12,100 simulated individuals at the 5% level of significance. There was no evidence of a difference amongst the methods in terms of categorical error.

4.3.2 Adaptive Schemes Under Misspecified Psychometric Curve

All the adaptive schemes performed reasonably well when the samples were generated using a logistic psychometric curve with parameters encompassed by the

prior distribution. The next step was to assess their performance when the ‘true’ psychometric curve is misspecified in the model. For this samples were generating by Weibull psychometric curves which also included lapses rate of 0.05. In total there were 25 sample generating curves with each having one of five thresholds and one of five slopes at the threshold. Since the Weibull function is not defined for concentrations less than 0, any responses in this range were set to the 1/3 guessing probability. Figure 19 shows the full form of these curves.

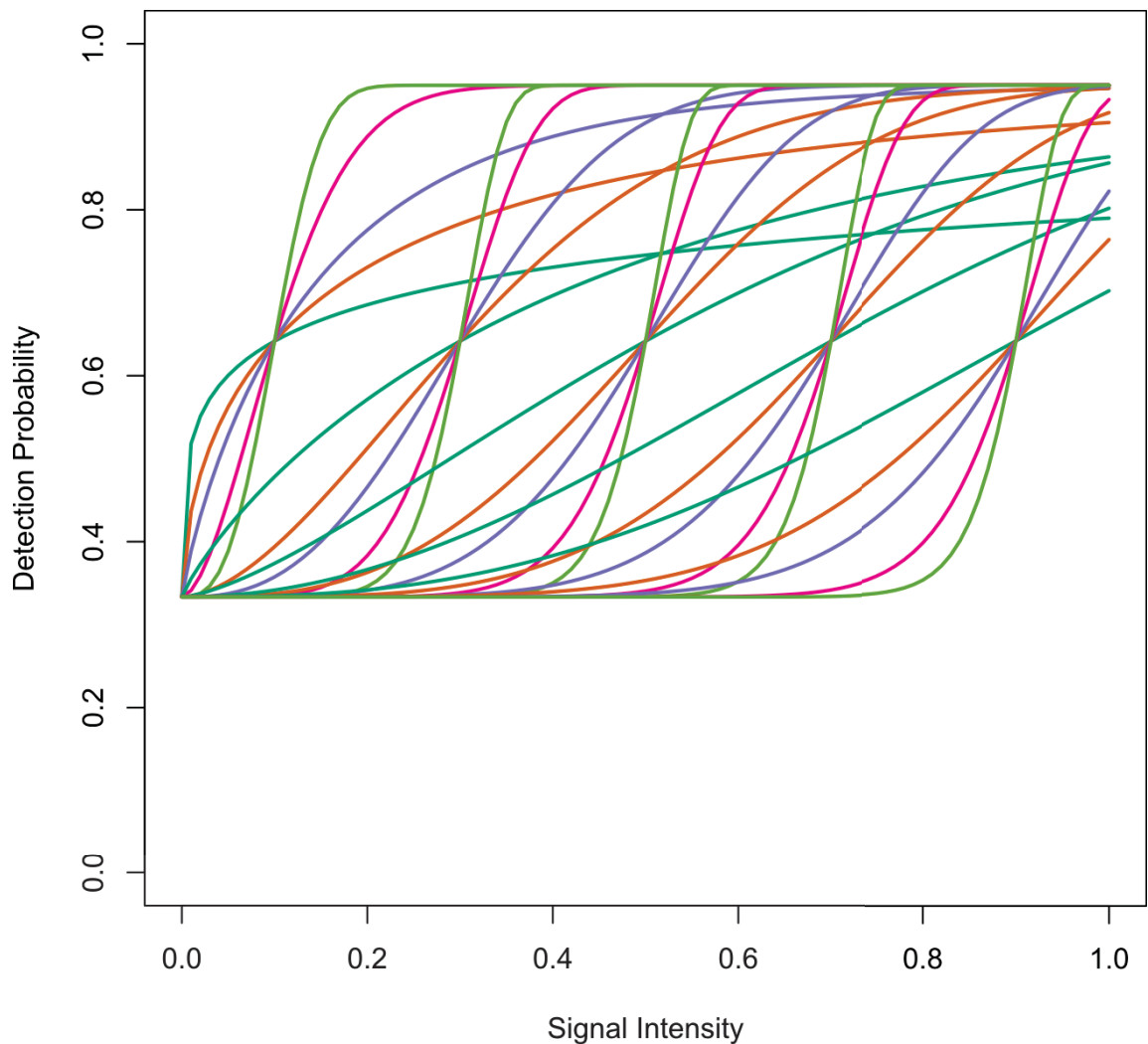


Figure 19 Weibull Psychometric curves used to generate misspecified samples with varying threshold and slope at threshold parameters.

The results of these fits were a little more mixed (Figure 20, Figure 21, and Figure 22). The RMSE was reasonably small when the underlying Weibull Psychometric curves had steep slopes but quickly deteriorated as the curves became flatter. In particular the flatter curves with low thresholds provided the worst fit for all methods, although these could be considered extreme and unlikely to be encountered in practice if the priors have been reasonably set.

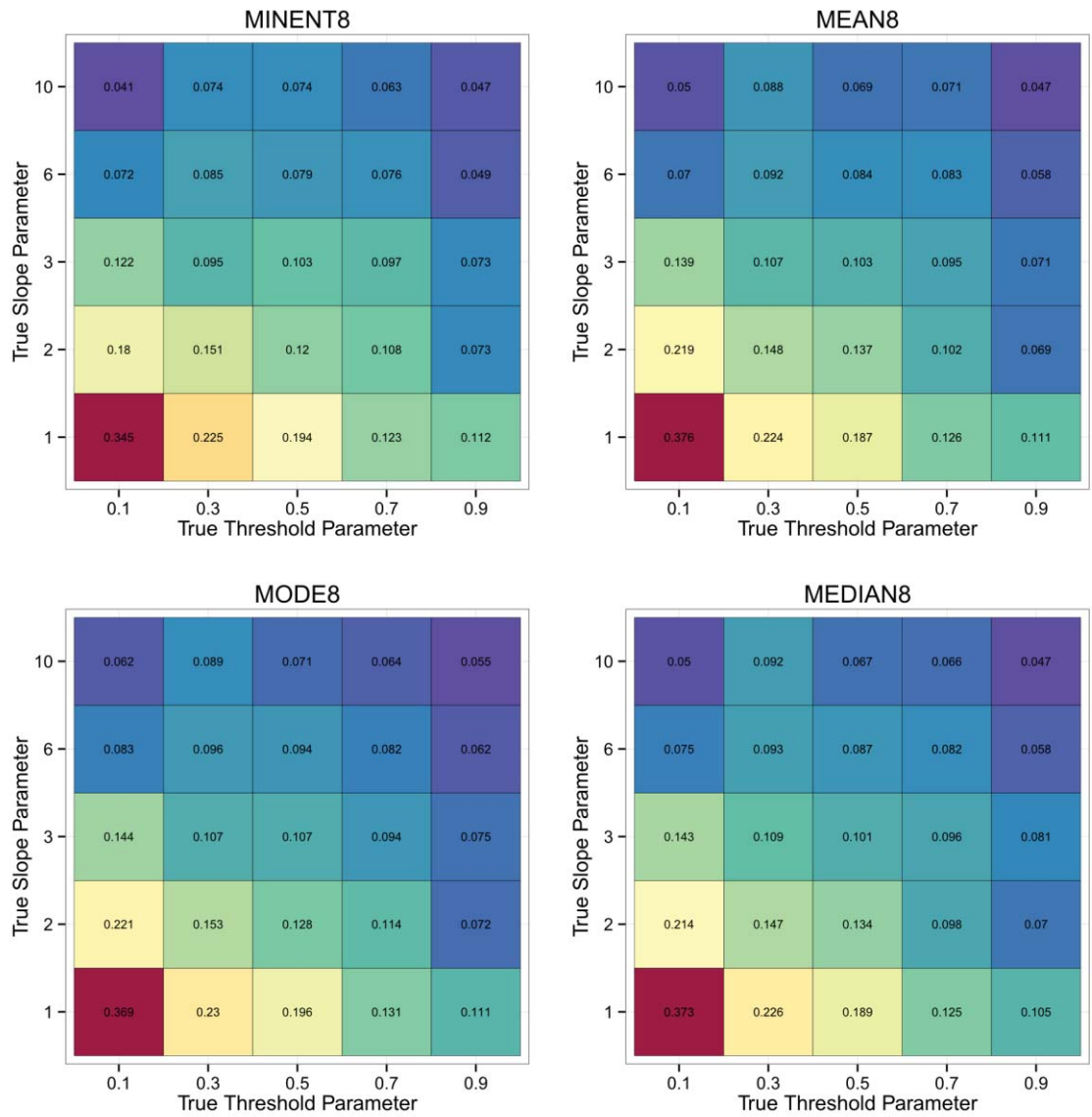


Figure 20 RMSE based on 100 samples generated by Weibull Psychometric curves of varying thresholds and slopes at the threshold.

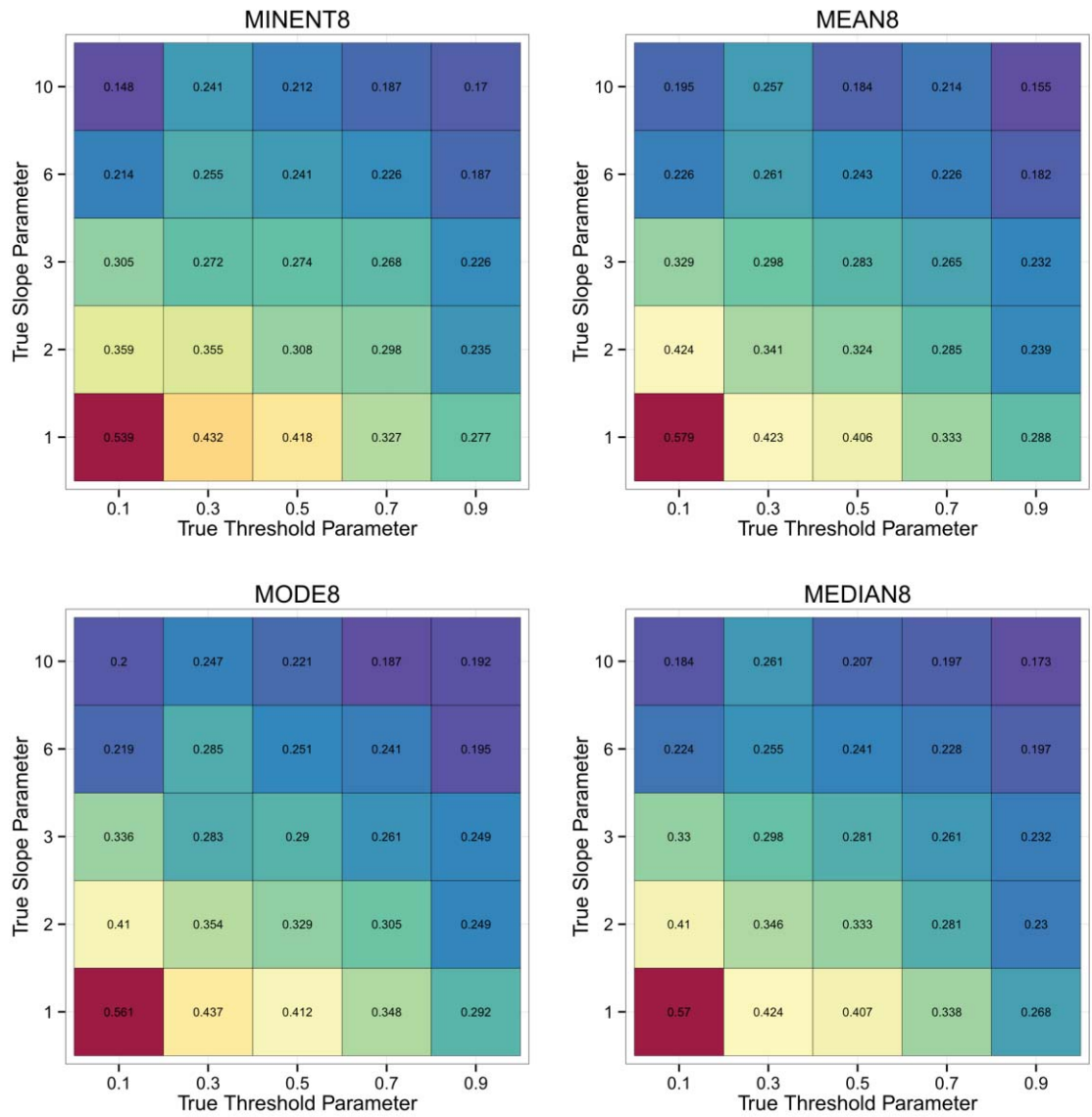


Figure 21 Absolute errors based on 100 samples generated by Weibull Psychometric curves of varying thresholds and slopes at the threshold.

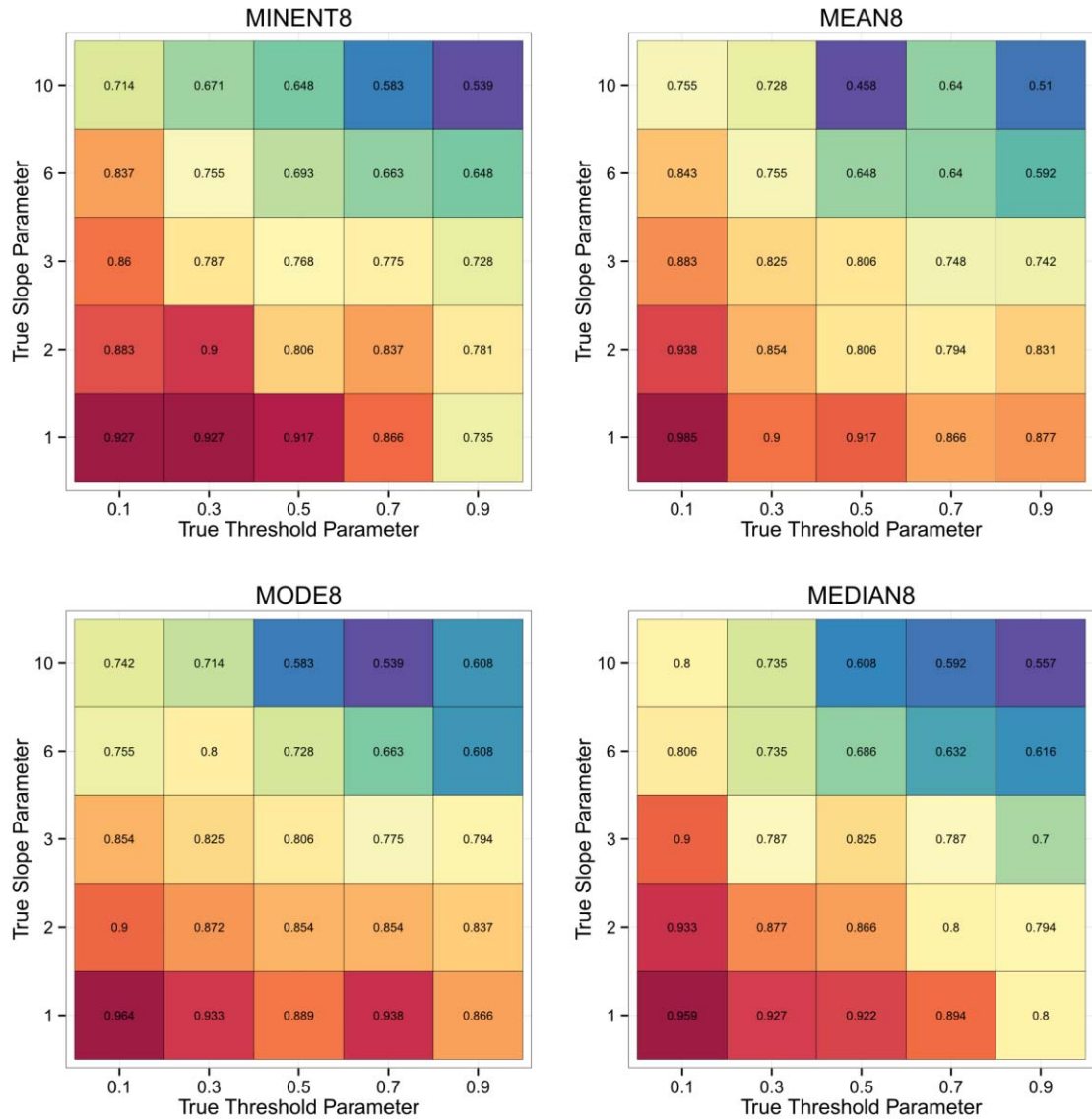


Figure 22 Categorical errors based on 100 samples generated by Weibull Psychometric curves of varying thresholds and slopes at the threshold.

The RMSE also appears to be less affected by the misspecification. Perhaps this is due to the posterior median and modes being restricted to one of 11 possible values under the discrete prior used here. The Categorical error in particular was extremely high, with the worst combination of slope and threshold resulting in a worse error than would be expected by randomly choosing a threshold.

Comparing the four methods over the misspecified models begins to show a clearer picture. Again the MODE8 performs worse in terms of RMSE and absolute error than the other methods. However the MINENT8 also has significantly lower RMSE and

absolute error than MEAN8 and MEDIAN8 schemes when compared using paired t-tests ($p < 0.05$), and lower categorical error than MODE8.

Overall the MINENT8 method appeared to perform the best and even gave reasonable results when the model was misspecified. Under the correct specification all models performed satisfactorily with 32 simulated 3-AFC samples per individual. In addition the final threshold estimate based on the posterior mean, which minimises the MSE performed, appeared to be less affected by the misspecification. Therefore the MINENT8 method, with the posterior mean as final threshold estimate was investigated further.

4.3.3 Comparing Look-Ahead Step Sizes: Minent8 vs. Minent1

While the main focus of this investigation was on multistep procedures, comparing their performance to the single-step method would give an indication of their efficiency. As the MINENT8 procedure gave the best results in the previous simulations it was compared to the MINENT1 method as described by Kontsevich & Tyler (1999). For this comparison logistic psychometric curves made up of combinations of eleven thresholds and six scale parameters (Figure 23).

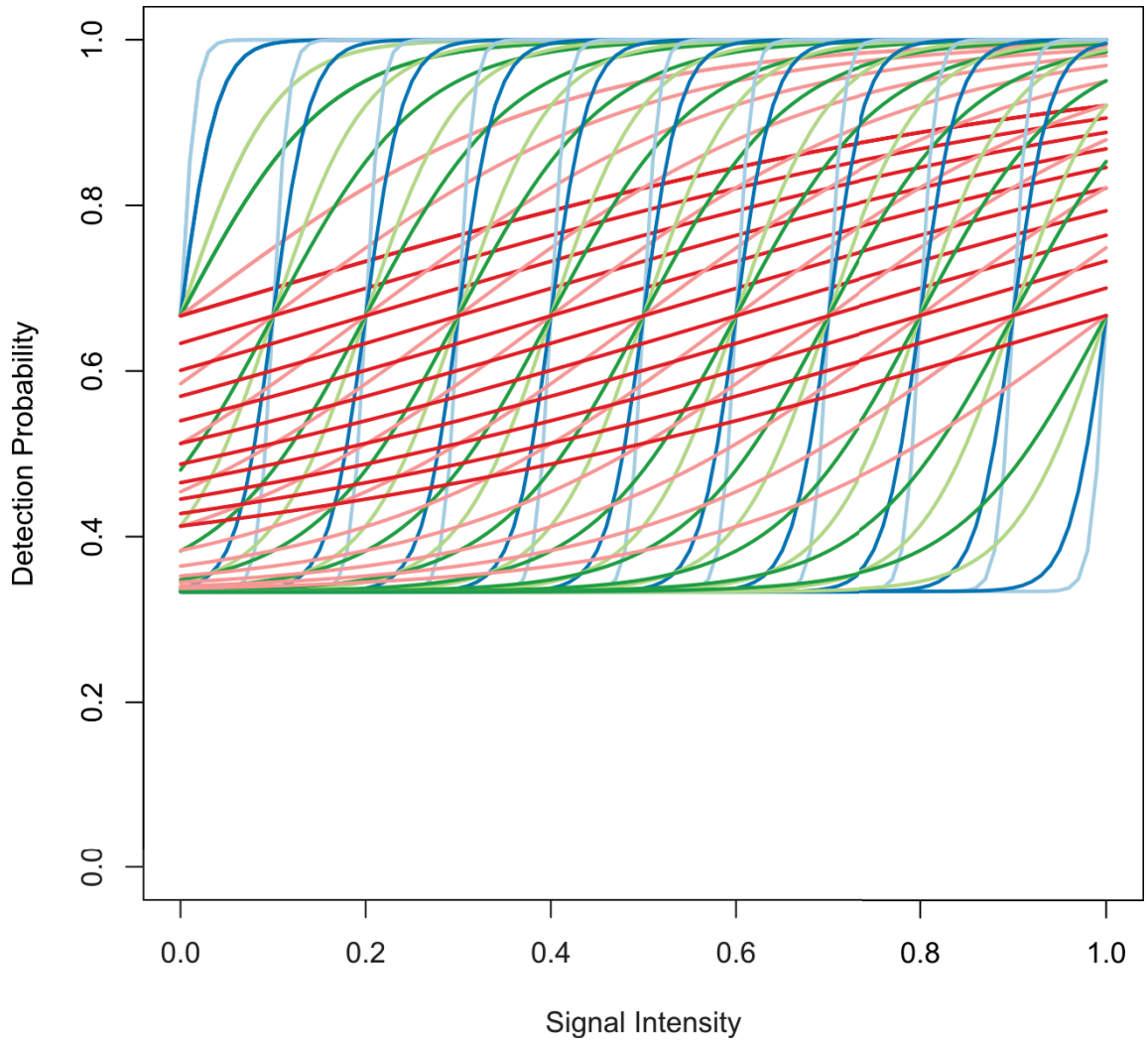


Figure 23 Logistic Psychometric curves used to compare the MINENT1 and MINENT8 performance.

All simulations involved 32 samples to be drawn in total, with the MINENT1 adapting the design after every observation resulting in 32 single sample runs, while MINENT8 after every set of 8 giving 4 runs of 8 samples. Figure 24 demonstrates the convergence process for the two methods. The position of the points on the y-axis represents the concentrations used for each sample, while the x-axis shows the sample number. Correct detections are shown as solid points, with incorrect being hollow. The posterior mean estimates of the detection threshold are plotted against the test number as solid lines for the MINENT1 and MINENT8 procedures.

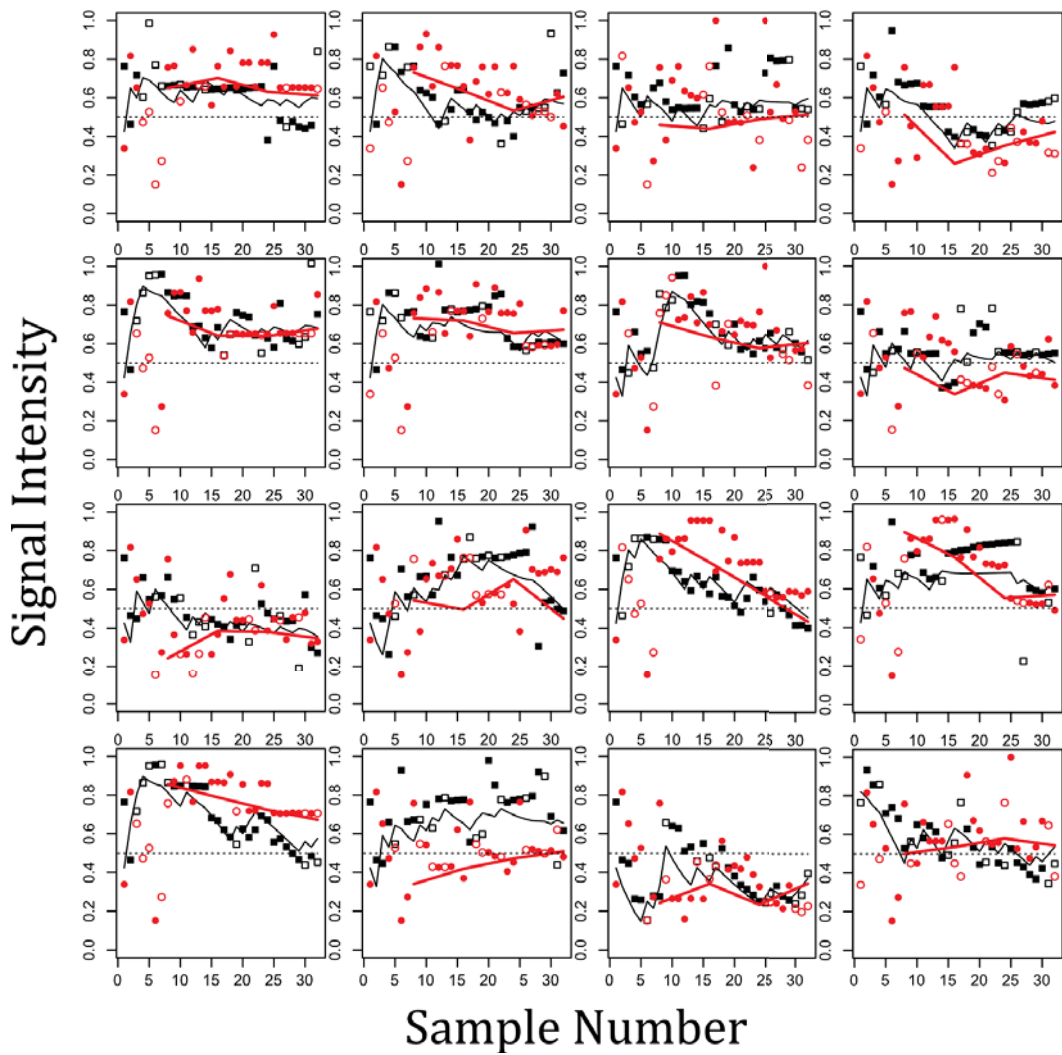
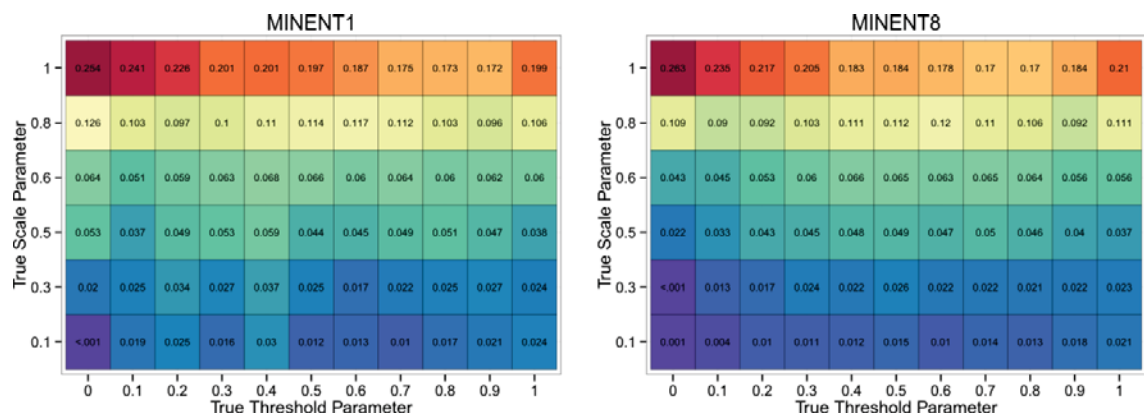


Figure 24 Comparison of the convergence of the MINENT8 (red) versus MINENT1 (black). Lines represent the posterior mean estimate of the threshold while points indicate concentration level for each test with correct answers solid point and incorrect hollow

The simulated RMSE loss measures based on 1,000 samples for each underlying psychometric curve are presented in Figures 25.



Figures 25 Comparison of RMSE for the MINENT1 and MINENT8 procedures, based on 1000 simulated individuals for various Logistic Psychometric curves

It can be seen that the MINENT8 procedure actually outperforms the MINENT1 for most curves, even if the difference is marginal. A t-test provided evidence that the MINENT8 had a lower MSE than the MINENT1 ($p=0.006$). This isn't entirely unexpected as it has been noted previously that looking ahead more than one step can perform slightly better, although the small increase in efficiency was not considered to be worth the considerable extra time cost due to computational limitations of the day (Kelareva et al., 2010; King-Smith et al., 1994).

4.4 ASTM Optimal Design

A method for generating a non-adaptive optimal design for use with the ASTM method was also investigated to provide a baseline to compare the adaptive methods with. For this the design was restricted to four runs of length 8 using the same 8 design points for each run. This is in line with the PFR experiments. The function was minimised by calculating the MSE as outlined in the methods for each of the 121 logistic psychometric functions used to evaluate the four adaptive procedures and taking its mean. The RMSE is plotted in Figure 26 and can be compared directly with Figure 16.

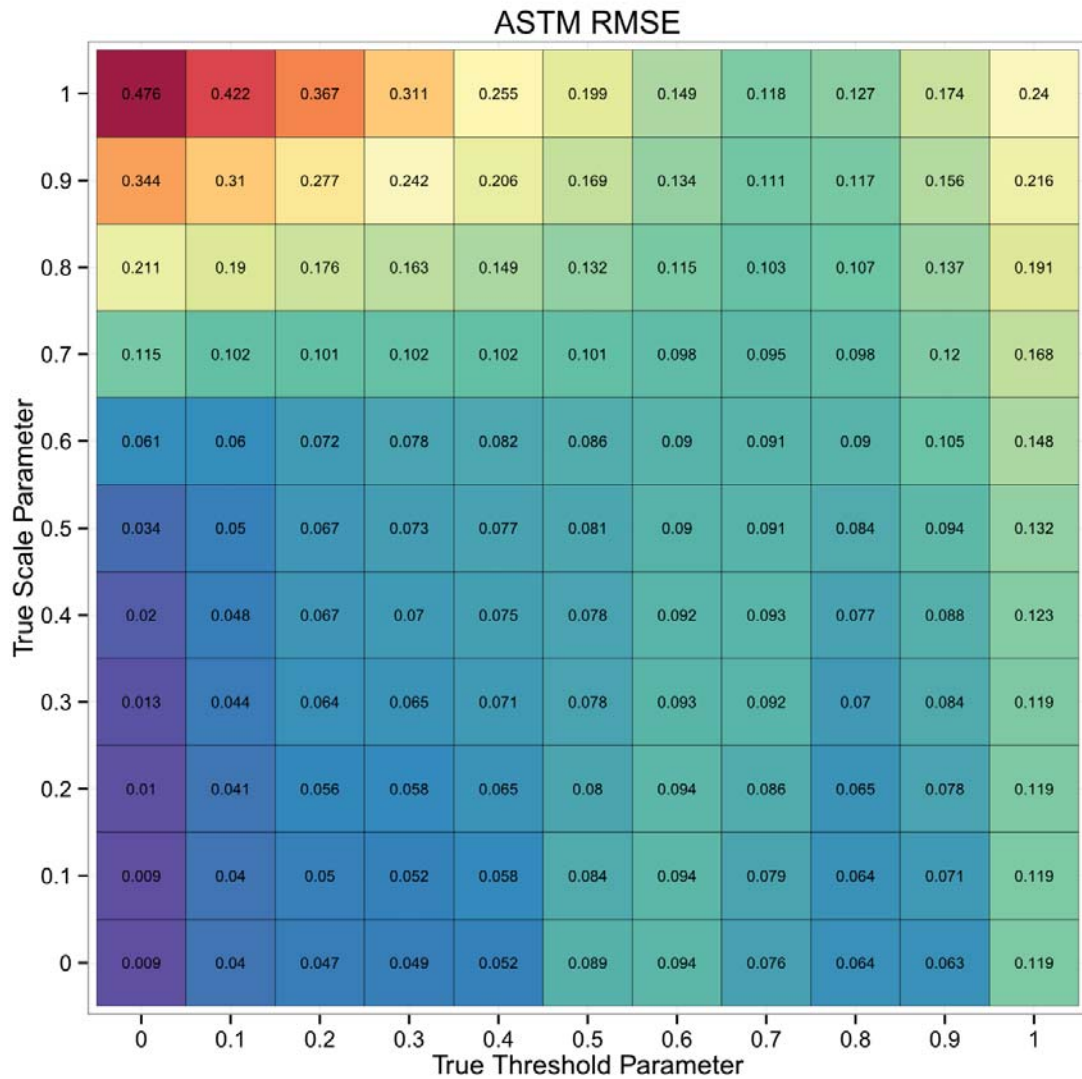


Figure 26 ASTM Expected RMSE for Logistic Psychometric curves with varying threshold and scale parameters.

The ASTM fixed design, as expected performed worse than the adaptive methods. The RMSE was approximately double that of the MINENT8 procedure and even with reasonably narrow scale parameters the ASTM fitted threshold estimates appear to be quite variable.

4.5 Discrete Signal Intensities

Under certain conditions it may only be possible to select design from a finite set of candidate points. In this case optimal designs produced by MINENT, MEAN, etc. may not be appropriate as they will highly likely contain points outside the set of candidates. A work around is implemented here. First the design points are rescaled from $[0,1]$ to the $[0,n]$ interval where m is the number of candidate design points.

These rescaled values are then converted to integers and the respective candidates values used as the design to be assessed by the minimisation algorithm as before. As this function will be non-smooth only the global solvers were considered for the optimisation routine. For the 11 point flat scale and threshold prior distributions the standard MINENT8 procedure gave an optimal design for the first run of 8 points with expected entropy of 1.67. The discrete version gave a similar if marginally worse solution. With candidate concentrations restricted to values ranging from 0 to 1 with step sizes of 0.1, the optimal design resulted in expected entropy of 1.70.

5 Discussion and Suggestions for Future research

The results presented in this study present and compare various methods for constructing designs for 3-AFC experiments with the restriction that any alterations cannot be made except between runs of samples. The runs in this case were set to four replicates with lengths of 8 samples to give a total of 32 samples per individual.

Of all the methods the Bayesian adaptive designs based on discrete priors performed the best with the lowest MSE. The MINENT8 procedure had a small increase in performance over MEAN8 and MEDIAN8, while MODE8 performed the worst of the four. The recommended final estimate under these methods is the posterior mean as it is influenced less by the discrete prior. The posterior median and mode are restricted to values which are defined exactly in the prior distribution. This was especially evident when the underlying model was a Weibull psychometric curve with a lapse rate of 0.05. Here the posterior mean from the MINENT8 procedure produced superior RMSE.

The other methods trialled all had various issues. Using continuous priors took too long to estimate the posterior distribution through MCMC methods as the process needs to be repeated many times for the solver to minimise a given loss function.

Approximating the MCMC estimates with Neural Networks only produces reasonable results for the posterior mean and medians. Unfortunately minimising the square loss of the Neural Networks did not translate to an approximately optimal design with the resulting MSE's being rather poor for the simulated results.

Deriving a D-optimal design for the standard logistic regression model was reasonably straightforward. However this proved to be too problematic for the logistic psychometric curve with the solver running into errors relating to the hessian matrix.

The ASTM based fixed design was reasonably straightforward to calculate and to estimate the MSE exactly, however its performance was worse than the adaptive procedures as expected.

Overall the MINENT8 method produced the lowest MSE's and was more robust to model misspecification. It even performed better than the MINENT1, although while the performance increase is statistically significant the difference is negligible.

The code provided in the appendices provides a framework for fitting any of the adaptive methods MINENT, MODE, MEDIAN, and MEAN for m-step look-ahead designs for a number of runs. The investigation here used mainly 8 steps and 4 runs but it is trivial to adjust the code to alter these. However choosing too many steps may require a long time to optimise. For olfactory based n-AFC designs this shouldn't be a problem as the number of steps will probably be limited to a reasonable number to avoid panellist fatigue. Using the same priors 11 level discrete priors as shown in Figure 15, took 6.5 seconds to find an optimal design for a 10-step look-ahead design ($2^{10}=1024$ possible responses), and 40.1 seconds for 12-step design (4096 possible responses), which seems reasonable.

It is also simple to adjust the code to handle AFC experiments with the number of choices other than three by editing the likelihood function. Similarly, other psychometric curves such as the Reverse Weibull, can be inserted in place of the Logistic in the likelihood calculation. In addition by altering the likelihood the framework could find an optimum design for testing protocols other than AFC. Kelareva et al., 2010, used the MINENT, MEAN, MEDIAN, and MODE adaptive methods for simulations relating to the yes-no protocol.

It is also possible to use a loss function different to those described in the methods. One alternative trialled in this investigation was the joint entropy of the threshold and scale parameters. It is not reported in the text but the code can be found in the appendix. This method would also seek to maximise the information gain on the slope as well as the threshold. The joint entropy of two variables X and Y is defined as:

$$H(X, Y) = - \sum_j \sum_i P(x_j, y_i) \log_2 P(x_j, y_i)$$

The prior distributions for the parameters can also be adjusted with one current limitation that the number of levels for the threshold and scale (or slope) parameters must be equal. This is due to how the marginal likelihoods are calculated by reshaping a vector into a square array and taking the column or row sums. However it would be possible to modify this code to allow for different dimensions. Increasing the number of levels also increases the time needed to find an optimal design. For example the 11 level priors resulting in 121 possible psychometric curves used in much of the simulations results in finding a MINENT8 solution for 8 design points in 1.36 seconds. Increasing the number of levels in each of the priors to 101 gives a sequence from 0 to 1 in step sizes of 0.01 and results in 10,201 possible curves and takes 62 seconds to reach a solution. While this is a considerable increase in time to find an optimal design it is still well within the realms of an acceptable time frame.

The priors also need not be discrete uniform as was used here either or even rescaled to the $[0,1]$ interval. Any discrete distribution could be used although some care may be needed to calculate the joint probability of the threshold and scale/slope parameters. In the examples used here they were assumed to be uniform and independent and therefore the joint probability for each threshold and scale combination is equal to one over the number of such combinations, i.e. $1/121$.

Different solvers could also be easily implemented or even combination algorithms. For example using PSwarm to get an approximate solution and then refining the search with Powell's method. The list of solvers considered here is only a small subset of those available and therefore some experimentation is encouraged. While most of the code is python 3 compatible some of the global solvers are not and therefore substitutes should be found and implemented.

Perhaps the most glaring limitation of this research is the fact that it is only based on simulations from a known model. The main goal of any future research would be to trial it on real panellists, where the noise from a real world situation would really test the method. At any rate the simulations provide a theoretical basis for future research into the area of multi-step look-ahead adaptive designs.

Other future work would involve tweaking the code to allow for easier substitution of priors, solvers, and psychometric curves. It would also be beneficial to allow for the inclusion of prior distributions for the guessing and lapse parameters, even at the risk of making the routine more complicated and therefore slower.

Improving the speed of the algorithms would also be a goal as it would allow for the algorithms to be used on designs with more points or larger number of prior levels. One possible solution is to incorporate python package such as theano or PyCUDA which allow calculations to be handled by the GPU rather than the CPU. This should provide a significant speedup for the array manipulations used presently. In particular the calculation of the posterior median as implemented here uses a rather inefficient search which cannot be sped up using numexpr.

Currently PyMC uses a mixture of native python code as well as custom fortran code which can in some cases result in speeds similar to native C code. One situation where this is true is when models are specified using array notation. This was the original intention with multiple models fit at the same time, however while it did provide a speed increase the convergence of the model was unsatisfactory and the slower scalar variable representation was used with models run individually. One implication of this is that coding in pure C may potentially give faster runs, however it would have the disadvantage of not having the additional diagnostic functions, and various python libraries available. Additionally the speedup gains would unlikely to be significant enough to solve the problem of sampling from multiple posterior distributions in a small enough time frame to be useable. Two of Python's strengths are its code readability and extensibility and both were major factors in its choice for this investigation. Future research could look at writing the underlying MCMC sampling in C++ with a python interface for usability.

Some secondary goals would be to translate the methods into the R statistical computing language(R. CoreTeam, 2013) and making a gui. This would have some advantages in terms of the initial setup. The package management system is simpler and would avoid the problems involved in installing extra libraries encountered in python. R also has excellent support for Windows, Linux, and MAC operating systems

including 64bit versions. One possible downside may be that performance in terms of speed may be reduced but by how much is unknown. A naive guess would be that R would be on par with Numpy without the speed enhancements provided by the numexpr library.

Another option to improve usability would be to add a Graphical user Interface (gui) to the python code and release it as a pre-compiled binary file. There are a number of python packages to make this process easier although it would still take some time to implement.

Nevertheless the code presented in the Appendices are in a useable form which can easily be altered to suit most needs for generating multi-step look-ahead adaptive designs for the estimation of sensory thresholds.

6 Conclusion

The results of this investigation demonstrate that it is possible to construct multi-step look-ahead adaptive designs for sensory threshold estimation in a computationally efficient manner. A wide variety of adaptive design methods have been proposed and successfully implemented for the estimation of psychometric curves. Previously, most of these adaptive methods had focused on setting the next test level at some estimate of threshold based on the data collected up until that point, or minimising the expected loss function by looking-ahead a number of steps. The look-ahead approaches generally only used one or two-step approaches due to either limitation in computer power at the time (King-Smith et al., 1994; Kontsevich & Tyler, 1999) or the price of more complex optimisation problem was not worth the effort for a relatively small improvement gained by having larger steps (Kelareva et al., 2010). However there are situations whereby the design cannot be altered at every step. The estimation of odour detection thresholds using the 3-AFC method requires significant setup time and therefore signal intensities must be known well in advance. A one or two-step design would require large waiting times for the subject while the next sample was being prepared and therefore a design which looked further would be advantageous. The design could be fixed for a given session with adjustments to the signal intensities made in time before the next.

All of the adaptive methods based on the discrete priors work reasonably well for estimating sensory thresholds when correctly assuming the logistic psychometric curve. Based on simulated a large number of individuals, each with data consisting of four sessions of eight samples the MSE appeared to be lower than comparable schemes investigated. When the generating psychometric function did not match the assumed logistic form the MINENT procedure appeared to be the most robust method. The MINENT procedure, which minimises the expected entropy to adapt the design after each session, is therefore the recommended design protocol for n-AFC experiments.

The bibliography provides python code to construct the designs described here. The framework is very flexible and can be readily altered to assume a number of psychometric curves in the underlying model as well as different prior distributions. This flexibility should enable the method to be used for most practical situations involving the estimation of olfactory thresholds using n-AFC experiments. Future research should involve verification of the method with real-world situations.

7 Bibliography

Alcalá-Quintana, R., & García-Pérez, M. A. (2004). The role of parametric assumptions in adaptive Bayesian estimation. *Psychological Methods*, 9(2), 250.

Amerine, M. A., Pangborn, R. M., & Roessler, E. B. (1965). *Principles of sensory evaluation of food*. New York: Academic Press.

ASTM International. (2011). E679 - 04: Standard Practice for Determination of Odor and Taste Thresholds By a Forced-Choice Ascending Concentration Series Method of Limits. *ASTM International, West Conshohocken, PA, USA*. Retrieved from <http://www.astm.org/Standards/E679.htm>

Garcia-Perez, M. A., & Alcala-Quintana, R. (2007). Bayesian adaptive estimation of arbitrary points on a psychometric function. *British Journal of Mathematical and Statistical Psychology*, 60, 147–174(28). doi:10.1348/000711006X104596

Gelman, A., & Rubin, D. B. (1992). Inference from Iterative Simulation Using Multiple Sequences. *Statistical Science*, 7(4), 457–472. doi:10.1214/ss/1177011136

Giunta, A. A., Wojtkiewicz, S. F., & Eldred, M. S. (2003). Overview of modern design of experiments methods for computational simulations. In *Proceedings of the 41st AIAA Aerospace Sciences Meeting and Exhibit, AIAA-2003-0649*. Retrieved from <http://aircraftdesign.nuaa.edu.cn/MDO/ref/Disciplinary%20Optimization/Data%20Sampling%20and%20Surrogate%20Models/AIAA%202003-0649.pdf>

Green, D. M., & Swets, J. A. (1966). *Signal Detection Theory and Psychophysics*. New York: Wiley.

Hall, J. L. (1981). Hybrid adaptive procedure for estimation of psychometric functions. *The Journal of the Acoustical Society of America*, 69(6), 1763–1769.

Heise, M. A., & Myers, R. H. (1996). Optimal Designs for Bivariate Logistic Regression. *Biometrics*, 52(2), 613. doi:10.2307/2532900

- Heymann, H., & Lawless, H. T. (1999). *Sensory Evaluation of Food: Principles and Practices* (first.). New York: Chapman & Hall.
- Hummel, T., Sekinger, B., Wolf, S. R., Pauli, E., & Kobal, G. (1997). “Sniffin” Sticks’: Olfactory Performance Assessed by the Combined Testing of Odor Identification, Odor Discrimination and Olfactory Threshold. *Chemical Senses*, 22(1), 39–52.
doi:10.1093/chemse/22.1.39
- Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3), 90–95.
- Jones, E., Oliphant, T., Peterson, P., & others. (2001). *SciPy: Open source scientific tools for Python*. Retrieved from <http://www.scipy.org/>
- Kelareva, E., Mewing, J., Turpin, A., & Wirth, A. (2010). Adaptive psychophysical procedures, loss functions, and entropy. *Attention, Perception, & Psychophysics*, 72(7), 2003–2012.
doi:10.3758/APP.72.7.2003
- King-Smith, P. E., Grigsby, S. S., Vingrys, A. J., Benes, S. C., & Supowit, A. (1994). Efficient and unbiased modifications of the QUEST threshold method: theory, simulations, experimental evaluation and practical implementation. *Vision Research*, 34(7), 885–912.
- Klein, S. A. (2001). Measuring, estimating, and understanding the psychometric function: a commentary. *Perception & Psychophysics*, 63(8), 1421–1455.
- Kontsevich, L. L., & Tyler, C. W. (1999). Bayesian adaptive estimation of psychometric slope and threshold. *Vision research*, 39(16), 2729–2737.
- Kroshko, D. (2007). *OpenOpt: Free scientific-engineering software for mathematical modeling and optimization*. Retrieved from <http://www.openopt.org/>
- Kuss, M., Jäkel, F., & Wichmann, F. A. (2005). Bayesian inference for psychometric functions. *Journal of Vision*, 5(5), 478–492. doi:10.1167/5.5.8

- Lee, H.-S. (2010, July 26). *Measuring food or consumers? Latest ideas and methodological issues in difference tests*. Presented at the Sensometrics, Rotterdam, The Netherlands. Retrieved from http://www.sensometric.org/Resources/Documents/2010/Meeting/Presentations/002-000-Hye-Seong%20Lee_2010.pdf
- Leek, M. R. (2001). Adaptive procedures in psychophysical research. *Perception & Psychophysics*, 63(8), 1279–1292.
- Li, F. (2008). Function Approximation by Neural Networks. In F. Sun, J. Zhang, Y. Tan, J. Cao, & W. Yu (Eds.), *Advances in Neural Networks - ISNN 2008* (pp. 384–390). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/chapter/10.1007/978-3-540-87732-5_43
- Linschoten, M. R., Harvey, L. O., Eller, P. M., & Jafek, B. W. (2001). Fast and accurate measurement of taste and smell thresholds using a maximum-likelihood adaptive staircase procedure. *Attention, Perception, & Psychophysics*, 63(8), 1330–1347.
- McCulloch, W. S., & Pitts, W. (1990). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 52(1-2), 99–115. doi:10.1007/BF02459570
- Oliphant, T. E. (2007). Python for Scientific Computing. *Computing in Science and Engineering*, 9(3), 10–20. doi:10.1109/MCSE.2007.58
- Patil, A., Huard, D., & Fonnesbeck, C. J. (2010). PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software*, 35(4), 1.
- Peng, M., Jaeger, S. R., & Hautus, M. J. (2012). Determining odour detection thresholds: Incorporating a method-independent definition into the implementation of ASTM E679. *Food Quality and Preference*, 25(2), 95–104.
- Powell, M. J. D. (1964). An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal*, 7(2), 155–162. doi:10.1093/comjnl/7.2.155

- Prins, N. (2012). The psychometric function: The lapse rate revisited. *Journal of Vision*, 12(6). doi:10.1167/12.6.25
- R. CoreTeam. (2013). *R: A Language and Environment for Statistical Computing*. Vienna, Austria. Retrieved from <http://www.R-project.org/>
- Raftery, A. E., & Lewis, S. M. (1995). Hypothesis testing and model selection. In W. R. Gilks, S. Richardson, & D. Spiegelhalter (Eds.), *Markov Chain Monte Carlo in Practice* (pp. 129–153). London: Chapman and Hall.
- Taylor, M. M. (1971). On the Efficiency of Psychophysical Measurement. *The Journal of the Acoustical Society of America*, 49(2B), 505–508. doi:10.1121/1.1912379
- Taylor, M. M., & Creelman, C. D. (1967). PEST: Efficient Estimates on Probability Functions. *The Journal of the Acoustical Society of America*, 41(4A), 782–787. doi:10.1121/1.1910407
- Treutwein, B., & Strasburger, H. (1999). Fitting the psychometric function. *Perception & Psychophysics*, 61(1), 87–106.
- Vanlier, J., Tiemann, C. A., Hilbers, P. A. J., & van Riel, N. A. W. (2012). A Bayesian approach to targeted experiment design. *Bioinformatics (Oxford, England)*, 28(8), 1136–1142. doi:10.1093/bioinformatics/bts092
- Vaz, A. I. F., & Vicente, L. N. (2007). A particle swarm pattern search method for bound constrained global optimization. *Journal of Global Optimization*, 39(2), 197–219. doi:10.1007/s10898-007-9133-5
- Watson, A. B., & Pelli, D. G. (1983). Quest: A Bayesian adaptive psychometric method. *Perception & Psychophysics*, 33(2), 113–120. doi:10.3758/BF03202828
- Wichmann, F. A., & Hill, N. J. (2001). The psychometric function: I. Fitting, sampling, and goodness of fit. *Perception & Psychophysics*, 63(8), 1293–1313.
- Wise, P. M., Bien, N., & Wysocki, C. J. (2008). Two Rapid Odor Threshold Methods Compared to a Modified Method of Constant Stimuli. *Chemosensory Perception*, 1(1), 16–23. doi:10.1007/s12078-008-9010-8

Wojciechowski, M. (2011). *ffnet: Feed-forward neural network for python*. Retrieved from <http://ffnet.sourceforge.net/>

Zhu, C., Byrd, R. H., Lu, P., & Nocedal, J. (1997). Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software*, 23(4), 550–560. doi:10.1145/279232.279236

Appendix A R Computer Code

Approximate d' R code

```
dprime_proportion<-function(n,nsim=100000,dprime=1){  
  y<-rnorm(nsim,dprime,1)  
  res=rep(0,nsim)  
  for(i in 1:(n-1)){  
    temp=y-rnorm(nsim)  
    res[temp<0]=1  
  }  
  return(1-mean(res))  
}
```

Appendix B Python Computer Code

Multi-step look-ahead adaptive designs using Discrete Priors

```
#import libraries
import numpy as np
import numexpr as ne
import time
from scipy.optimize import minimize
from scipy.stats import binom
#if have installed playdoh
import playdoh as pd

##if using python 3.x
xrange=range

###create De Bruijn sequence (code from wikipedia.org)
def de_bruijn(k, n):
    """De Bruijn Sequence for alphabet size k
    and subsequences of length n."""
    a = [0] * k * n
    sequence = []
    def db(t, p):
        if t > n:
            if n % p == 0:
                for j in range(1, p + 1): sequence.append(a[j])
            else:
                a[t] = a[t - p]
                db(t + 1, p)
                for j in range(a[t - p] + 1, k):
                    a[t] = j
                    db(t + 1, t)
    db(1,1)
    return sequence

##repeat and reshape De Bruijn sequence into an array
##with all possible sequences of alphabet x and length n
def DB_array(x,n):
    temp=np.tile(np.array(de_bruijn(x,n)),2)
    c=x**n
    temp1=np.reshape(np.repeat(0,n*x**n),(x**n,n))
    for i in xrange(c):
        temp1[i,:]=temp[i:i+n]
    return temp1

##underlying function to calculate the expected loss
## loss is expected Entropy of threshold here
def work_fun(lprobs,lprobs2,yy,n,wgts,ncombs):
    ##check if only looking 1 point ahead and use numpy
    ##if true as numexpr returns wrong shaped array
    ##causing errors later
    if np.shape(yy)[1]==1:
        ##find log-like
        g=np.sum((lprobs*(yy)+(lprobs2)*(n-yy)+ncombs),axis=2)
    else:
        ##if looking more than 1 point ahead numexpr returns
correct        ## shape array so use numexpr to calc log-like(much
faster)
```

```

        g=ne.evaluate("sum((lprobs*(yy)+(lprobs2)*(n-yy)+ncombs),axis=2)")
        ##calc posterior probs =(prior*likelihood)
        z=ne.evaluate("wgts*exp(g)")
        ##reshape to square array
        z_temp=np.reshape(z,(np.size(m0)**.5,np.size(m1)**.5,np.shape(yy)
    ) [0]))
        ##calc posterior probs for each m0(threhsold) value
        pm0=ne.evaluate("sum(z_temp,axis=1)")
        ##calc probs of rows of yy
        x_weights=ne.evaluate('sum(z,axis=0)')
        #normalise rows
        rsum=x_weights[np.newaxis,:]
        ##return entropy of posterior dist for m0 (threshold)
        return ne.evaluate('sum(-pm0*log(pm0/rsum))')

##underlying function to calculate the expected loss
## loss is expected MSE of threshold here
def work_fun_mean(lprobs,lprobs2,yy,n,wgts,ncombs):
    if np.shape(yy)[1]==1:
        g=np.sum((lprobs*(yy)+(lprobs2)*(n-yy)+ncombs),axis=2)
    else:
        g=ne.evaluate("sum((lprobs*(yy)+(lprobs2)*(n-yy)+ncombs),axis=2)")
        z=ne.evaluate("wgts*exp(g)")
        z_temp=np.reshape(z,(np.size(m0)**.5,np.size(m1)**.5,np.shape(yy)
    ) [0]))
        m0_temp=m0[:,np.newaxis]
        ##calculate posterior means for each row of yy
        mean_m0=ne.evaluate('sum(z*m0_temp,axis=0)')
        ##calculate the probability of a sample
        x_weights=ne.evaluate('sum(z,axis=0)')
        ##calculate probs of each m0 and samp combination
        m0_by_x_weights=ne.evaluate("sum(z_temp,axis=1)")
        ##calculate and return MSE
        return ne.evaluate('sum(m0_by_x_weights*(mean_m0/x_weights-
m0_unique)**2)')

##underlying function to calculate the expected loss
## loss is expected absolute error of threshold here
def work_fun_median(lprobs,lprobs2,yy,n,wgts,ncombs):
    if np.shape(yy)[1]==1:
        g=np.sum((lprobs*(yy)+(lprobs2)*(n-yy)+ncombs),axis=2)
    else:
        g=ne.evaluate("sum((lprobs*(yy)+(lprobs2)*(n-yy)+ncombs),axis=2)")
        z=ne.evaluate("wgts*exp(g)")
        z_temp=np.reshape(z,(np.size(m0)**.5,np.size(m1)**.5,np.shape(yy)
    ) [0]))
        x_weights=ne.evaluate('sum(z,axis=0)')
        m0_by_x_weights=ne.evaluate("sum(z_temp,axis=1)")
        #calculate median for each row of global yy
        #seems inefficient
        cumulative_sum=np.cumsum(m0_by_x_weights,axis=0)
        cumulative_sum=cumulative_sum/cumulative_sum[-1,:]
        np.place(cumulative_sum,cumulative_sum<.5,100)
        m0_median=m0_unique[np.argmin(cumulative_sum,axis=0),0]
        ##return mean absolute error
        return ne.evaluate('sum(m0_by_x_weights*abs(m0_median-
m0_unique))')

```



```

##underlying function to calculate the expected loss
## loss is expected categorical error of threshold here
def work_fun_mode(lprobs,lprobs2,yy,n,wgts,ncombs):
    if np.shape(yy)[1]==1:
        g=np.sum((lprobs*(yy)+(lprobs2)*(n-yy)+ncombs),axis=2)
    else:
        g=ne.evaluate("sum((lprobs*(yy)+(lprobs2)*(n-yy)+ncombs),axis=2)")
    z=ne.evaluate("wgts*exp(g)")
    z_temp=np.reshape(z,(np.size(m0)**.5,np.size(m1)**.5,np.shape(yy)[0]))
    #m0_temp=m0[:,np.newaxis]
    #mean_m0=ne.evaluate('sum(z*m0_temp,axis=0)')
    x_weights=ne.evaluate('sum(z,axis=0)')
    m0_by_x_weights=ne.evaluate("sum(z_temp,axis=1)")
    #posterior mode for each row of yy
    m0_mode=m0_unique[np.argmax(m0_by_x_weights,axis=0),0]
    #return mean categorical error
    return ne.evaluate('sum((m0_mode!=m0_unique)*m0_by_x_weights)')

##underlying function to calculate the expected loss
## loss is expected joint entropy of threshold & scale here
def work_fun_joint_ent(lprobs,lprobs2,yy,n,wgts,ncombs):
    g=ne.evaluate("sum((lprobs*(yy)+(lprobs2)*(n-yy)+ncombs),axis=2)")
    z=ne.evaluate("wgts*exp(g)")
    z_temp=np.reshape(z,(np.size(m0)**.5,np.size(m1)**.5,np.shape(yy)[0]))
    x_weights=ne.evaluate('sum(z,axis=0)')
    #normalise rows
    #rsum=ne.evaluate(('sum(pm0,axis=0)') dont need to as
x_weights=rsum
    rsum=x_weights[np.newaxis,:]
    #return mean joint entropy of m0 an m1
    return ne.evaluate("sum(-z*log(z/rsum))")

###calculate the number of combinations of choosing r from n
def nCr(n):
    temp=np.repeat(0,n)
    for i in xrange(n):
        temp[i]=np.product(np.arange(n-i,n)/np.product(np.arange(1,i+1)))
    return temp

##calculates expected entropy for all possible responses (yy) for a
given
## design (c), prior levels m0 & m1 with joint prior probs weights
## (default is equal probs).
## n= number of samples per element of c (default max of yy)
## nsplit specifics how many sections to split up yy if it
## is too large to pass to work_fun with results combined
## at end (default is no splitting)
def subarrayne(c,yy,m0,m1,weights=None,n=None,nsplit=None):
    if weights is None:
        weights=np.repeat(1.,np.size(m0))
    if n is None:
        n=np.max(yy)
    if nsplit is None:
        nsplit=1
    ncombs=np.log(nCr(n+1)[yy])
    ##create new arrays with right dims for array maniputaions

```

```

        m00=m0[:,np.newaxis]
        m11=m1[:,np.newaxis]
        ##calculate probs of detection for each level of c and m0 & m1
        probs=ne.evaluate("(0.9999/3.0+2.0/3.0*1.0/(1.0+exp((- (c-
m00)/(0.005*10**(m11*2))))))")
        ##check prior probs sum to 1
        wgts=weights[:,np.newaxis]/np.sum(weights)
        ##take logs for easier calculations (log like) in work fun
        lprobs=np.log(probs[:,np.newaxis])
        lprobs2=np.log(1.-probs[:,np.newaxis])
        ##split yy if too large
        y_split=np.array_split(yy,nsplit)
        ncombs_split=np.array_split(ncombs,nsplit)
        res=0.
        ##caculate expected entropy -looping of yy is too large
        for i in xrange(nsplit):

            res=res+work_fun(lprobs,lprobs2,y_split[i],n,wgts,ncombs_split[i]
1)

        #return expected entropy
        return res

##calculates expected MSE for all possible responses for a given
design
def subarrayne_mean(c,yy,m0,m1,weights=None,n=None,nsplit=None):
    if weights is None:
        weights=np.repeat(1.,np.size(m0))
    if n is None:
        n=np.max(yy)
    if nsplit is None:
        nsplit=1
    ncombs=np.log(nCr(n+1)[yy])
    m00=m0[:,np.newaxis]
    m11=m1[:,np.newaxis]
    probs=ne.evaluate("(0.9999/3.0+2.0/3.0*1.0/(1.0+exp((- (c-
m00)/(0.005*10**(m11*2))))))")
    wgts=weights[:,np.newaxis]/np.sum(weights)
    lprobs=np.log(probs[:,np.newaxis])
    lprobs2=np.log(1.-probs[:,np.newaxis])
    y_split=np.array_split(yy,nsplit)
    ncombs_split=np.array_split(ncombs,nsplit)
    res=0.
    for i in xrange(nsplit):

        res=res+work_fun_mean(lprobs,lprobs2,y_split[i],n,wgts,ncombs_sp
lit[i])
    #return expected MSE
    return res

##calculates expected absolute error for all possible responses for a
given design
def subarrayne_median(c,yy,m0,m1,weights=None,n=None,nsplit=None):
    if weights is None:
        weights=np.repeat(1.,np.size(m0))
    if n is None:
        n=np.max(yy)
    if nsplit is None:
        nsplit=1
    ncombs=np.log(nCr(n+1)[yy])
    m00=m0[:,np.newaxis]
    m11=m1[:,np.newaxis]

```

```

        probs=ne.evaluate("( .9999/3.0+2.0/3.0*1.0/(1.0+exp((- (c-
m00)/(0.005*10**(m11*2))))))")
        wgts=weights[:,np.newaxis]/np.sum(weights)
        lprobs=np.log(probs[:,np.newaxis])
        lprobs2=np.log(1.-probs[:,np.newaxis])
        y_split=np.array_split(yy,nsplit)
        ncombs_split=np.array_split(ncombs,nsplit)
        res=0.
        for i in xrange(nsplit):

            res=res+work_fun_median(lprobs,lprobs2,y_split[i],n,wgts,ncombs_
split[i])
            #return expected absolute error
        return res

##calculates expected categorical error for all possible responses for
a given design
def subarrayne_mode(c,yy,m0,m1,weights=None,n=None,nsplit=None):
    if weights is None:
        weights=np.repeat(1.,np.size(m0))
    if n is None:
        n=np.max(yy)
    if nsplit is None:
        nsplit=int(np.shape(yy)[0]/400)+1
    ncombs=np.log(nCr(n+1)[yy])
    m00=m0[:,np.newaxis]
    m11=m1[:,np.newaxis]
    probs=ne.evaluate("( .9999/3.0+2.0/3.0*1.0/(1.0+exp((- (c-
m00)/(0.005*10**(m11*2))))))")
    wgts=weights[:,np.newaxis]/np.sum(weights)
    lprobs=np.log(probs[:,np.newaxis])
    lprobs2=np.log(1.-probs[:,np.newaxis])
    y_split=np.array_split(yy,nsplit)
    ncombs_split=np.array_split(ncombs,nsplit)
    res=0.
    for i in xrange(nsplit):

        res=res+work_fun_mode(lprobs,lprobs2,y_split[i],n,wgts,ncombs_sp
lit[i])
        #return expected categorical error
    return res

##calculates expected joint entropy for all possible responses for a
given design
def subarrayne_joint_ent(c,yy,m0,m1,weights=None,n=None,nsplit=None):
    if weights is None:
        weights=np.repeat(1.,np.size(m0))
    if n is None:
        n=np.max(yy)
    if nsplit is None:
        nsplit=1
    ncombs=np.log(nCr(n+1)[yy])
    m00=m0[:,np.newaxis]
    m11=m1[:,np.newaxis]
    probs=ne.evaluate("( .9999/3.0+2.0/3.0*1.0/(1.0+exp((- (c-
m00)/(0.005*10**(m11*2))))))")
    wgts=weights[:,np.newaxis]/np.sum(weights)
    lprobs=np.log(probs[:,np.newaxis])
    lprobs2=np.log(1.-probs[:,np.newaxis])
    y_split=np.array_split(yy,nsplit)
    ncombs_split=np.array_split(ncombs,nsplit)

```

```

    res=0.
    for i in xrange(nsplitted):

        res=res+work_fun_joint_ent(lprobs,lprobs2,y_split[i],n,wgts,ncom
bs_split[i])
        #return expected joint entropy
        return res

##make loss functions take an array as an argument
##needed for playdoh
def ent_loss(c,nweights=None,n=None):
    return np.apply_along_axis(subarrayne, 0,
c,yy,m0,m1,nweights,n,None)*1.

def square_loss(c,nweights=None,n=None):
    return np.apply_along_axis(subarrayne_mean, 0,
c,yy,m0,m1,nweights,n,None)*1.

def abs_loss(c,nweights=None,n=None):
    return np.apply_along_axis(subarrayne_median, 0,
c,yy,m0,m1,nweights,n,None)*1.

def binary_loss(c,nweights=None,n=None):
    return np.apply_along_axis(subarrayne_mode, 0,
c,yy,m0,m1,nweights,n,None)*1.

def binary_loss_scaled(c,nweights=None,n=None):
    return np.apply_along_axis(subarrayne_mode_scaled, 0,
c,yy,m0,m1,nweights,n,None)*1.

## given responses y & design c calculate posterior probabilities
## for threshold (m0) and width (m1) parameters
## weights=prior probs - default to uniform
## n is the number of samples per design point -default 1

def calc_nweights(c,y,m0,m1,weights=None,n=None):
    if weights is None:
        weights=np.repeat(1.,np.size(m0))
    if n is None:
        n=np.max(y)
    probs=np.reshape(np.repeat(0.,(np.size(m0)*np.size(c))), (np.size
(m0),np.size(c)))
    y2=np.tile(y,(np.size(m0),1))
    cc=np.tile(c,(np.size(m0),1))
    mm0=np.transpose(np.tile(m0,(np.size(y),1)))
    mm1=np.transpose(np.tile(m1,(np.size(y),1)))
    probs=ne.evaluate(".9999/3.0+2.0/3.0*1.0/(1.0+exp((- (cc-
mm0)/(0.005*10**(mm1*2))))))")
    #if only look-ahead 1 use numpy -numexpr causes error as it does
not reduce the
    #dimension of the array's correctly otherwise numexpr for look-
ahead>1
    if np.size(c)==1:
        llike=np.sum((np.log(probs)*(y2)+np.log(1-probs)*(n-
y2)),axis=1)
        like=weights*np.exp(llike)
    else:
        llike=ne.evaluate("sum((log(probs)*(y2)+log(1-probs)*(n-
y2)),axis=1)")
        like=ne.evaluate("weights*exp(llike)")
    #check posterior probs sum to 1

```

```

        like=like/np.sum(like)
        #return posterior probs
        return like

##simulates responses from a logistic psychometric curve with
## threshold ma and width mb and calculates the optimal design
## to minimise the loss function (loss_fun) given the simulated data.
## The number of samples is based on the global variable yy (see
below)
## This is repeated a number of times (nreps) with the prior
distribution
## (nweights) updated after each rep. Optional arguments c_best is the
## optimal design based on the original prior distribution only so it
## does not need to be repeatedly found.
## After nreps completed it returns posterior Mean, Mode, and Median
def
adaptive_fun(nreps,loss_fun,ma,mb,nweights=None,n=None,c_best=None):
    if nweights is None:
        nweights=np.repeat(1./np.size(m0),np.size(m0))
    for i in xrange(nreps):
        #if c_best is not specified or isn't 1st iter then find
opt design
        if c_best==None or i>0:

            s=minimize(loss_fun,np.linspace(0.,1.,np.shape(yy)[1]),method='P
owell',tol=0.01,args=(nweights,n))
            c_best=s.x
            #calculate probs of detection at opt design, assuming
thresh=ma and scale=mb
            pp=(.9999/3.0+2.0/3.0*1.0/(1.0+np.exp((- (c_best-
ma)/(0.005*10**(mb*2))))))
            #take binomial random sample assumin pp
            samp=binom.rvs(1,pp)
            #calculate posterior probs with observed samp
            #these probs are used as prior in next iteration

            nweights=calc_nweights(c_best,samp,m0,m1,weights=nweights,n=n)
            ##calculate posterior mean, ,mode, and median & return
            nw_temp=np.sum(np.reshape(nweights,(np.size(m0)**.5,np.size(m1)*
*.5)),axis=1)
            mf_mode=m0_unique[np.argmax(nw_temp),0]
            cumulative_sum_mf=np.cumsum(nw_temp)
            cumulative_sum_mf=cumulative_sum_mf/cumulative_sum_mf[-1]
            np.place(cumulative_sum_mf,cumulative_sum_mf<.5,100)
            mf_median=m0_unique[np.argmax(cumulative_sum_mf),0]
            mf_mean=np.sum(nw_temp*m0_unique[:,0])
            return np.array([mf_mean,mf_mode,mf_median])

## similar to above except fixes intital value problem when yy is a
nx1 array
def
adaptive_fun_minent1(nreps,loss_fun,ma,mb,nweights=None,n=None,c_best=
None):
    if nweights is None:
        nweights=np.repeat(1./np.size(m0),np.size(m0))
    for i in xrange(nreps):
        if c_best==None or i>0:

            s=minimize(loss_fun,.5+np.linspace(0.,1.,np.shape(yy)[1]),method
='Powell',tol=0.01,args=(nweights,n))
            c_best=s.x

```

```

        pp=(.9999/3.0+2.0/3.0*1.0/(1.0+np.exp((-c_best-
ma)/(0.005*10**(mb*2))))))
        samp=binom.rvs(1,pp)
        #samp=binom.isf(rand_num,1,1-pp)

        nweights=calc_nweights(c_best,samp,m0,m1,weights=nweights,n=n)
        nw_temp=np.sum(np.reshape(nweights,(np.size(m0)**.5,np.size(m1)*
*.5)),axis=1)
        mf_mode=m0_unique[np.argmax(nw_temp),0]
        cumulative_sum_mf=np.cumsum(nw_temp)
        cumulative_sum_mf=cumulative_sum_mf/cumulative_sum_mf[-1]
        np.place(cumulative_sum_mf,cumulative_sum_mf<.5,100)
        mf_median=m0_unique[np.argmin(cumulative_sum_mf),0]
        mf_mean=np.sum(nw_temp*m0_unique[:,0])
        return np.array([mf_mean,mf_mode,mf_median])

## Similar to adaptive_fun except uses a random sample of standard
uniform vars
## (rand_num) to calculate the simulated responses - this allows for
direct comparison
## amongst various loss functions. Instead of random binomial vars
sampled the binomial
## inverse of the rand_num array is taken.
def adaptive_fun2(nreps,loss_fun,ma,mb,rand_num,c_best=None):
    nweights=np.repeat(1./np.size(m0),np.size(m0))
    for i in xrange(nreps):
        #checks if initial design is present and it is the first
iteration
        if c_best==None or i>0:

            s=minimize(loss_fun,np.linspace(0.,1.,np.shape(yy)[1]),method='P
owell',tol=0.01,args=(nweights,n))
            c_best=s.x
            pp=(.9999/3.0+2.0/3.0*1.0/(1.0+np.exp((-c_best-
ma)/(0.005*10**(mb*2))))))
            #Take inverse of binomial dist for rand_num percentile
            samp=binom.isf(rand_num[i],1,pp)
            #find posterior dist given samp - this is used as prior
for next iter

            nweights=calc_nweights(c_best,samp,m0,m1,weights=nweights,n=None
)
            #calculate the posterior probs for unique thresholds (m0)
            #only works if the number of levels in m0=m1
            nw_temp=np.sum(np.reshape(nweights,(np.size(m0)**.5,np.size(m1)*
*.5)),axis=1)
            #calculate posterior mode
            mf_mode=m0_unique[np.argmax(nw_temp),0]
            #calculate posterior median - seems inefficient
            cumulative_sum_mf=np.cumsum(nw_temp)
            cumulative_sum_mf=cumulative_sum_mf/cumulative_sum_mf[-1]
            np.place(cumulative_sum_mf,cumulative_sum_mf<.5,100)
            mf_median=m0_unique[np.argmin(cumulative_sum_mf),0]
            #calc postior mean
            mf_mean=np.sum(nw_temp*m0_unique[:,0])
            #return posterior mean, mode, median all in an array
            return np.array([mf_mean,mf_mode,mf_median])

## create prior threshold (m0) and width (m1) paramter values
## prior probabilities are assumed equal but can be specified
## if needed. Curently need to have an equal number of levels

```

```

## here it is 11.
m0=np.repeat(np.arange(11),11)/10.
m1=np.tile(np.arange(11),11)/10.
## create global variable yy which is an array of all possible
## sequences of length 8 and alphabet 2 (i.e. binary)
yy=DB_array(2,8)
## unique values of m0 and m1 - values from 0 to 1 with step of 0.1
m0_unique=np.unique(m0)[:,np.newaxis]
m1_unique=np.unique(m1)[:,np.newaxis]

##calculates expected entropy for all possible responses for a given
design
yy=DB_array(2,8)
## could try others e.g.
#yy=DB_array(3,8) ##note would need to use n=2 in the functions
#yy=DB_array(2,10)
m0_unique=np.unique(m0)[:,np.newaxis]
m1_unique=np.unique(m1)[:,np.newaxis]

### Example of find optimum design for yy based on prior alone
### and entropy loss function. Uses Powell's minimisation method.
### Also gives time to calculate.
### This can be used as an initial design for simulations
temp=time.clock()
s_init=minimize(ent_loss,np.linspace(0.,1.,np.shape(yy)[1]),method='Powell',tol=0.01)
time.clock()-temp
###create an array to store results from 10 simulations
res_8=np.reshape(np.repeat(0.,30),(10,3))
## define initial optimum design for first 8 samples as calculated
above
c_best_init=s_init.x
###loop for 10 simulated runs of 4 reps of 8 samples with adaptive
method
###assuming entropy loss function i.e. MINENT8 with 4 reps
###takes c_best_init as initial design so it doesn't have to re-
calculate
###it 10 times. This is a simulation assuming a true threshold of 0.3
and
### scale of 0.5. The output (posterior mean, mode, median) are stored
in
### res8. Also returns time to complete the loop

temp=time.clock()
for i in xrange(10):
    res_8[i]=adaptive_fun(4,ent_loss,.3,.5,nweights=None,n=1,c_best=
c_best_init)

time.clock()-temp

###check estimated means (1st col) modes (2nd) median (3rd)
res_8
### MSE based on 10 samples
np.mean((res_8[:,0]-.3)**.5)
### Categorical error based on 10 reps
np.mean((res_8[:,0]!=.3))
### Absolute error based on 10 reps
np.mean(np.abs(res_8[:,0]-.3))

#####find a design with observed data EXAMPLE

```

```

##first calculate new prior (actually posterior) given observed
response, concentrations
##n_obs is the number of tests at each observed concentration
new_prior=calc_nweights(concentrations_ob,response_ob,m0,m1,weights=None,n=n_obs)
##find optimal design for next set of design points (length =
number_points)
##n is the number of samples at each point probably 1
my_des=minimize(ent_loss,np.linspace(0.,1.,number_points),method='Powell',tol=0.01,args=(new_prior,n))

#####DISCRETE SOLVER#####
##define available concentrations
concs=np.arange(11)/10.
##convert [0,1] interval to integers
## ranging from 0 to number of concentrations-1
## and finally return elements of concs relating to
## these integers
def concs_discrete(c):
    ##fix up edges for converting to ints
    c[c<=.00001]=.00001
    c[c>=.99999]=.99999
    c2=np.repeat(0,np.size(c))
    for i in xrange(np.size(c)):
        c2[i]=int(c[i]*(np.size(concs)))
    return concs[c2]

##similar to subarrayne except converts design into sample from concs
def subarrayne_discrete(c,yy,m0,m1,weights=None,n=None,nsplit=None):
    ##convert design c to elements of conc
    c=concs_discrete(c)
    if weights is None:
        weights=np.repeat(1.,np.size(m0))
    if n is None:
        n=np.max(yy)
    if nsplit is None:
        nsplit=1
    ncombs=np.log(nCr(n+1)[yy])
    m00=m0[:,np.newaxis]
    m11=m1[:,np.newaxis]
    probs=ne.evaluate("(.9999/3.0+2.0/3.0*1.0/(1.0+exp((-c-
m00)/(0.005*10**(m11*2))))))")
    wgts=weights[:,np.newaxis]/np.sum(weights)
    lprobs=np.log(probs[:,np.newaxis])
    lprobs2=np.log(1.-probs[:,np.newaxis])
    y_split=np.array_split(yy,nsplit)
    ncombs_split=np.array_split(ncombs,nsplit)
    res=0.
    for i in xrange(nsplit):
        res=res+work_fun(lprobs,lprobs2,y_split[i],n,wgts,ncombs_split[i]
    ])
    ##return expected entropy
    return res

#allow array inputs for pswarm solver
def ent_loss_discrete(c,nweights=None,n=None):
    return np.apply_along_axis(subarrayne_discrete, 0,
c,yy,m0,m1,nweights,n,None)*1.

```



```

##solve with pswarm fof design of size 8
##descriptions from playdoh example
## results saved as results
if __name__ == '__main__':
    # State space dimension (D)
    dimension = 8
    # ``initrange`` is a Dx2 array with the initial intervals for
every
    # dimension
    initrange = np.tile([0., 1.], (dimension, 1))
    # Maximize the fitness function in parallel
    results = pd.minimize(ent_loss_discrete,
                           popsize=100, # size of the population
                           maxiter=10, # maximum number of iterations
                           cpu=8, # number of CPUs to use on the local
machine
                           initrange=initrange)

##show best design
results.best_pos
##los function for opt design
concs_discrete(results.best_pos)

```

D-optimal Design code for Logistic Regression

```

import numpy as np
import numdifftools as nd
import playdoh as pd
from scipy.optimize import minimize

#####calculate negative loglikelihood
##### ignores nCr as is constant
def logistic_like(p0,p=None,n=None,x=None):
    ##some default params for assumed
    ##logistic curve to find D-opt design
    if p is None:
        p=(0.5,2.5)
    if n is None:
        n=1
    if x is None:
        x=(-.94,-.2,.54)
    e=p[0]+p[1]*np.array(x)
    e1=p0[0]+p0[1]*np.array(x)
    like=np.exp(e)/(1.+np.exp(e))
    q=np.exp(e)/(1.+np.exp(e))
    y=q*1.
    z=-np.sum(y*np.log((like/(1.-like)))+1.*np.log((1.-like)))
    return(z)

##passes design (c) and intercept(a1)
##and slope (a2) of underlying logistic reg
##returns detrminant of inverse of hessian
##of likelihood at MLE
def myfun(c,a1=None,a2=None):
    if a1 is None:
        a1=0.5
    if a2 is None:
        a2=2.5
    a=(a1,a2)
    hnd=nd.Hessian(lambda a: logistic_like(a,x=c))

```

```

ll=np.linalg.det(np.linalg.inv(hnd(np.array(a))))
return(ll)

##solve D-optimal design - args(0,1), mean underlying logistic has
##intercept=0 and slope of 1 -try others. Since 2 initial values
##(-5,1) are solving for design of 2 points
s=minimize(myfun,(-.5,1.),method='L-BFGS-B',args=(0,1),bounds=((-
10.,10.),(-10.,10.)))
##try design with 3 points
s=minimize(myfun,(5,1.,3),method='L-BFGS-
B',args=np.array([[0.,1.])),bounds=((-10.,10.),(-10.,10.),(-10.,10.)))
##function for pswarm
def myfun2(p):
    z=np.apply_along_axis(myfun, 0, p)
    return(z)
##compare pswarm solver - note no args therefore assumes
##default underlying logistic intercept=0.5, slope=2.5
results =
pd.minimize(myfun2,popsize=100,maxiter=10,cpu=1,initrange=(np.array([[
-5.,5.],[ -5.,5.] ])))

#####try for logistic psychometric curve
#####Doesn't always work- needs more work -Results May Not Be
Reliable!!
##calculate negative log-likelihood at MLE
def logisticpsy_like2(p0,p,n=None,x=None):
    if p is None:
        p=(0.5,0.05)
    if n is None:
        n=100.
    if x is None:
        x=(-.94,-.2,.54)
    q=0.9999/3.+2./(3.*(1.+np.exp((p[0]-np.array(x))/p[1])))
    like=1./3.+2./(3.*(1.+np.exp((p0[0]-np.array(x))/p0[1])))
    y=q*n
    z=-np.sum(y*np.log((like/(1.-like)))+n*np.log((1.-like)))
    return(z)

##find determinant of hessian at MLE for design (c)
def myfunpsytest(a,c=None):
    if c is None:
        c=(.45,.5)
    hnd=nd.Hessian(lambda b: logisticpsy_like2(b,p=a,n=1.,x=c))
    #ll=np.linalg.det(np.linalg.inv(hnd(np.array(a))))
    ll=np.linalg.inv(hnd(np.array(a)))
    res=np.linalg.det(ll)
    ##checks if positive-definite
    ##all eigenvalues should be positive
    if np.any(np.linalg.eigvals(ll) < 0):
        res=99999
    return((res))

#try find D-Optimal design for three curves
g=np.array([[.5,.2],[.4,.1],[.3,.05]])
##doesn't work for these
#g=np.reshape(np.repeat(0.,121*2),(121,2))
#g[:,0]=np.repeat(np.arange(11)/10.,11)
#g[:,1]=np.tile(np.arange(11)/10.,11)

##function to take multiple underlying curves to

```

```

##optimise
def myfunpsy2(p):
    z=np.apply_along_axis(myfunpsytest, 1., g,p)
    return(np.mean(z))

##function for playdoh -allows array args
def myfunpsy2_pd(p):
    z=np.apply_along_axis(myfunpsy2, 0.,p)
    return(z)

##minimise using box bound L-BFGS-B solver
s=minimize(myfunpsy2, (.1,.8),method='L-BFGS-B',bounds=((0.,1.), (0.,1.)))
##minimise using PSwarm
results =
pd.minimize(myfunpsy2_pd,popsize=10,maxiter=10,cpu=4,initrange=(np.tile([0., 1.], (8, 1))))

```

Minimum MSE design for ASTM method

```

from scipy.stats import binom
from scipy.optimize import minimize
import numpy as np
##generate parameters to calc expected MSE
m0_long=np.repeat(np.arange(11)/10.,11)
m1_long=np.tile(np.arange(11)/10.,11)
##function to calculate probabilities of detection
##for a given design (d) and params m0_long & m1_long
def Logistic_psych(d):
    return (1./3.0+2.0/3.0*1.0/(1.0+np.exp((- (d-
m0_long)/(0.005*10**(m1_long*2))))))

##find probabilities for each possible ASTM threshold
def multi_probs(g):
    mp=np.zeros(np.size(g)+1)
    for i in range(0,np.size(g)+1):
        mp[i]=np.append(np.sort(np.cumprod(g[::-1])),1)[i]*(1-
np.append(0,g)[i])
    return mp

##calculate all possible ASTM thresholds
def last_reverse(g):
    g1=np.append(2.0*g[0]-g[1],g)
    k=np.size(g)
    g2=np.append(g,2.0*g[k-1]-g[k-2])
    return (g1+g2)/2.0

##estimate average MSE over all psychometric curves
## defined by each m0_long and m1_long pair
def lastreverse_mse_final(X,nreps):
    ind_probs=np.transpose(Logistic_psych((X[:,np.newaxis])))
    mps=np.apply_along_axis(multi_probs,1,ind_probs)
    t_ests=last_reverse(X)
    ##estimated variance
    est_var=np.sum(mps*(t_ests**2),axis=1)-
np.sum(mps*(t_ests),axis=1)**2
    ##estimated MSE
    est_mse=np.mean((np.sum(mps*(t_ests),axis=1)-
m0_long)**2+est_var/nreps)
    return est_mse

```

```

##estimate MSE for each individual psychometric curve
## defined by each m0_long and m1_long pair
##nreps is the number of reps for each design point
def lastreverse_mse_final_by_curve(X,nreps):
    ind_probs=np.transpose(Logistic_psych((X)[: ,np.newaxis]))
    mps=np.apply_along_axis(multi_probs,1,ind_probs)
    t_ests=last_reverse(X)
    est_var=np.sum(mps*(t_ests**2),axis=1)-
np.sum(mps*(t_ests),axis=1)**2
    est_mse=((np.sum(mps*(t_ests),axis=1)-m0_long)**2+est_var/nreps)
    return est_mse

##find solution for design of 8 points with 4 reps
s_lr=minimize(lastreverse_mse_final,np.arange(8)/7.,method='Powell',to
l=0.01,args=((4),))
s_lr.x

```

Attempted Multi-step look-ahead adaptive designs using NN approximations

```

import itertools as it
import numpy as np
concentrations=[a for a in
it.combinations((np.arange(0.1+.8/11.0,.89,.8/11.0)),6)]
import gc as gc
import pymc as mc
from scipy.stats import bernoulli
import copy
n=1
##generate 25008 samples of 40 points within
##various sub-intervals of [0,1]. These are the concentrations
##to train the Neural Networks
np.random.seed(29875)
concs_40=np.random.rand(25008,40)
centre=np.transpose([(0,.25,.5,.75,.125,.375,.625,.0,.333,.667,.1665,
.5,.0,.5,0.25,.0)])
scale=np.transpose([(0.25,0.25,0.25,0.25,0.25,0.25,0.25,.333,.333,.333
,.333,.333,.5,.5,.5,1.0)])
scale=np.repeat(scale,1563,axis=0)
centre=np.repeat(centre,1563,axis=0)
##rescale to lie within the various sub-ints
concs_40_f=concs_40*scale+centre
##random seed for reproducibility
np.random.seed(15964)
#np.random.seed(1138)
#np.random.seed(2525)
##draw thresholds and scale paramters from standard uniform
mrand=np.random.uniform(low=0.0,high=1.,size= np.shape(concs_40_f)[0])
srand=np.random.random(np.shape(concs_40_f)[0])
##calculate detection probs for given design (XX), mrand & srand
def psy_p(XX):
    return (1.0/3.0+2.0/3.0*1.0/(1.0+np.exp((- (XX-
mrand)/(0.005*10**(srand*2))))))

np.random.seed(654987)
##calc detection probs for all sim designs
prand=np.apply_along_axis(psy_p, 0, concs_40_f)
##draw binomial samples with probs=prand, n=1
samp=np.random.binomial(n,prand)
##import pyentropy and use function to find approx entropy
##by discretising the posterior
from pyentropy import quantise

```

```

def approx_ent(a,nbins):
    ll=quantise(a,nbins,uniform='bins',minmax=(0,1),centers=False)
    counts=np.trim_zeros(np.sort(np.bincount(ll[0])*1.))
    ent=-sum(counts/sum(counts)*np.log2(counts/sum(counts)))
    return(ent)

##arrays to store results
import time
result_8=np.zeros((np.size(centre),9))
result_16=np.zeros((np.size(centre),9))
result_24=np.zeros((np.size(centre),9))
result_32=np.zeros((np.size(centre),9))
result_40=np.zeros((np.size(centre),9))

temp=time.clock()

##function to initialise model for MCMC
##uniform priors for scale and threshold
##logistic psychometric function
def est_psy_model(response,concsx,nn):
    import numpy as np
    import pymc as mc
    s1 = mc.Beta('s1', 1.0, 1.0, value=np.zeros(1)+0.51)
    m1 = mc.Uniform('m1', 0.0, 1.0, value=np.zeros(1)+0.51)
    #likelihood
    #y_i=mc.Binomial('y_i',value=response,n=nn,p=(1.0/3.0+2.0/3.0*1.
0/(1.0+mc.exp((- (concsx-
m1)/(0.005*10**(s1*2)))))),observed=True,trace=False)
    @mc.deterministic(plot=False)
    def modelled_yy(c=concsx, m=m1,s=s1,trace=False):
        """modelled_yy = 1.0/3.0+2.0/3.0*(1/(1+exp((-
2.197225/wp*(XX-m1[ind])))))"""
        return (1.0/3.0+2.0/3.0*1.0/(1.0+np.exp((- (c-
m)/(0.005*10**(s*2))))))
    @mc.stochastic(observed=True,trace=False)
    def y(value=response,p0=modelled_yy,n0=nn):
        return mc.binomial_like(value,n=n0,p=p0)
    return locals()

def est_psy(response,concsx,nn):
    import pymc as mc
    import numpy as np
    import gc as gc
    ##specify model using est_psy_model
    model = mc.Model(est_psy_model(response,concsx,nn))
    ##calculate mode of posterior
    M=mc.MAP(model)
    M.fit(method='fmin_powell',tol=0.0001, verbose=1)
    ##some code for memory management
    m1_mode=copy.copy(M.m1.value[0])
    s1_mode=copy.copy(M.s1.value[0])
    del M,model
    gc.collect()
    ##sample from posterior using MCMC metropolis algorithm
    ##trace is stoed in hdf5 to keep out of memory
    Ma =
mc.MCMC(est_psy_model(response,concsx,nn),db='hdf5',dbname='tempdb2.hd
f5',dbmode='w')
    #10000 iterations, burnin 1000, and thinning factor 10
    Ma.sample(iter=10000, burn=1000, thin=10)
    ##calculate posterior mean, sd, med for thresh & scale

```

```

t3=copy.copy(np.mean(Ma.trace('m1')[:]))
t4=copy.copy(np.std(Ma.trace('m1')[:]))
t5=copy.copy(np.mean(Ma.trace('s1')[:]))
t6=copy.copy(np.std(Ma.trace('s1')[:]))
m1_mean=copy.copy(Ma.m1.stats()['mean'])
m1_med=copy.copy(Ma.m1.stats()['quantiles'][50])
m1_sd=copy.copy(Ma.m1.stats()['standard deviation'])
s1_mean=copy.copy(Ma.s1.stats()['mean'])
s1_med=copy.copy(Ma.s1.stats()['quantiles'][50])
s1_sd=copy.copy(Ma.s1.stats()['standard deviation'])
##calculate approx threshold entropy
m1_ent=approx_ent(np.reshape(Ma.m1.trace()[:], (np.size(Ma.m1.trace()[:]),)),10)
Ma.db.close()
del Ma
gc.collect()
##return results
return
[m1_ent,m1_mode,m1_mean,m1_med,m1_sd,s1_mode,s1_mean,s1_med,s1_sd]

##calculate posterior estimates for 8,16, 24,32, and 40
##observations & store them
##TAKES DAYS!
for i in xrange(np.size(concs_40_f)/40):
    temp=time.clock()
    result_8[i]=est_psy(samp[i,0:8],concs_40_f[i,0:8],1)
    result_16[i]=est_psy(samp[i,0:16],concs_40_f[i,0:16],1)
    result_24[i]=est_psy(samp[i,0:24],concs_40_f[i,0:24],1)
    result_32[i]=est_psy(samp[i,0:32],concs_40_f[i,0:32],1)
    result_40[i]=est_psy(samp[i,0:40],concs_40_f[i,0:40],1)
    ##print to see what iteration at
    if i%200==199:
        gc.collect()
    print "i",i,time.clock()-temp

##save results as dont want to re-run
import pickle
object_res = result_8
file_res = open('final_logistic_result_8b.pkl', 'w')
pickle.dump(object_res, file_res)
file_res.close()

object_res = result_16
file_res = open('final_logistic_result_16b.pkl', 'w')
pickle.dump(object_res, file_res)
file_res.close()

object_res = result_24
file_res = open('final_logistic_result_24b.pkl', 'w')
pickle.dump(object_res, file_res)
file_res.close()

object_res = result_32
file_res = open('final_logistic_result_32b.pkl', 'w')
pickle.dump(object_res, file_res)
file_res.close()

object_res = result_40

```

```

file_res = open('final_logistic_result_40b.pkl', 'w')
pickle.dump(object_res, file_res)
file_res.close()

##read in result
import pickle
result_8=pickle.load(open('final_logistic_result_8b.pkl', 'r'))
result_16=pickle.load(open('final_logistic_result_16b.pkl', 'r'))
result_24=pickle.load(open('final_logistic_result_24b.pkl', 'r'))
result_32=pickle.load(open('final_logistic_result_32b.pkl', 'r'))
result_40=pickle.load(open('final_logistic_result_40b.pkl', 'r'))
##sort concentrations (simulated designs from beginning)
concs_8_f_sorted=np.sort(concs_40_f[:,0:8],axis=1)
concs_16_f_sorted=np.sort(concs_40_f[:,0:16],axis=1)
concs_24_f_sorted=np.sort(concs_40_f[:,0:24],axis=1)
concs_32_f_sorted=np.sort(concs_40_f[:,0:32],axis=1)
concs_40_f_sorted=np.sort(concs_40_f,axis=1)
c_ind_8=np.argsort(concs_40_f[:,0:8],axis=1)
c_ind_16=np.argsort(concs_40_f[:,0:16],axis=1)
c_ind_24=np.argsort(concs_40_f[:,0:24],axis=1)
c_ind_32=np.argsort(concs_40_f[:,0:32],axis=1)
c_ind_40=np.argsort(concs_40_f,axis=1)
##sort simulated samples so they match sorted concs
samp_8=np.zeros(np.shape(samp[:,0:8]))
samp_16=np.zeros(np.shape(samp[:,0:16]))
samp_24=np.zeros(np.shape(samp[:,0:24]))
samp_32=np.zeros(np.shape(samp[:,0:32]))
samp_40=np.zeros(np.shape(samp))
for i in xrange(np.shape(samp_40)[0]):
    samp_8[i]=samp[i,0:8][c_ind_8[i]]
    samp_16[i]=samp[i,0:16][c_ind_16[i]]
    samp_24[i]=samp[i,0:24][c_ind_24[i]]
    samp_32[i]=samp[i,0:32][c_ind_32[i]]
    samp_40[i]=samp[i,c_ind_40[i]]

input_8=np.append(concs_8_f_sorted,samp_8,axis=1)
input_16=np.append(concs_16_f_sorted,samp_16,axis=1)
input_24=np.append(concs_24_f_sorted,samp_24,axis=1)
input_32=np.append(concs_32_f_sorted,samp_32,axis=1)
input_40=np.append(concs_40_f_sorted,samp_40,axis=1)
target=result_40[:,(1,3)]
##import ffnet
from ffnet import ffnet, mlgraph
#conec = mlgraph((16,56,16,6))
#conec = mlgraph((80,320,8))
##### conection construction
input_test8=concs_8_f_sorted*(2.0*(samp_8-.5))
input_test16=concs_16_f_sorted*(2.0*(samp_16-.5))
input_test24=concs_24_f_sorted*(2.0*(samp_24-.5))
input_test32=concs_32_f_sorted*(2.0*(samp_32-.5))
input_test40=concs_40_f_sorted*(2.0*(samp_40-.5))

input_test40_f=np.append(input_test40,input_test40_2,axis=0)
input_test40_f=np.append(input_test40_f,input_test40_3,axis=0)
input_test40=concs_40_f_sorted+(2.0*(samp_40-.5))
target_test40=result_40[:,1]
target_test8=result_8[:,1]
conec = mlgraph((8,64,1))
conec40 = mlgraph((40,160,1))
net_8test = ffnet(conec)

```

```

net_40test = ffnet(conec40)

net_40test_ent = ffnet(conec40)
net_40test_mode = ffnet(conec40)
net_40test_mean = ffnet(conec40)
net_40test_median = ffnet(conec40)
net_40test_sd = ffnet(conec40)

print "TRAINING NETWORK..."
net_40test_ent.train_tnc(input_test40, result_40[:,0], maxfun = 5000,
messages=1,nproc='ncpu')
net_40test_mean.train_tnc(input_test40, result_40[:,2], maxfun = 5000,
messages=1,nproc='ncpu')
net_40test_mode.train_tnc(input_test40, result_40[:,1], maxfun = 5000,
messages=1,nproc='ncpu')
net_40test_median.train_tnc(input_test40, result_40[:,3], maxfun =
5000, messages=1,nproc='ncpu')
net_40test_sd.train_tnc(input_test40, result_40[:,4], maxfun = 5000,
messages=1,nproc='ncpu')
print

np.random.seed(29875)
arr=np.arange(25008)
np.random.shuffle(arr)
conec8 = mlgraph((8,32,1))
net_8test_ent = ffnet(conec8)
net_8test_mode = ffnet(conec8)
net_8test_mean = ffnet(conec8)
net_8test_median = ffnet(conec8)
net_8test_sd = ffnet(conec8)
print "TRAINING NETWORK..."
net_8test_ent.train_tnc(input_test8[arr[0:20000],:],
result_8[arr[0:20000],0], maxfun = 5000, messages=1,nproc='ncpu')
net_8test_mean.train_tnc(input_test8[arr[0:20000],:],
result_8[arr[0:20000],2], maxfun = 5000, messages=1,nproc='ncpu')
net_8test_mode.train_tnc(input_test8[arr[0:20000],:],
result_8[arr[0:20000],1], maxfun = 5000, messages=1,nproc='ncpu')
net_8test_median.train_tnc(input_test8[arr[0:20000],:],
result_8[arr[0:20000],3], maxfun = 5000, messages=1,nproc='ncpu')
net_8test_sd.train_tnc(input_test8[arr[0:20000],:],
result_8[arr[0:20000],4], maxfun = 5000, messages=1,nproc='ncpu')
print

conec32 = mlgraph((32,128,1))
net_32test_ent = ffnet(conec32)
net_32test_mode = ffnet(conec32)
net_32test_mean = ffnet(conec32)
net_32test_median = ffnet(conec32)
net_32test_sd = ffnet(conec32)

print "TRAINING NETWORK..."
net_32test_ent.train_tnc(input_test32[arr[0:20000],:],
result_32[arr[0:20000],0], maxfun = 5000, messages=1,nproc='ncpu')
net_32test_mean.train_tnc(input_test32[arr[0:20000],:],
result_32[arr[0:20000],2], maxfun = 5000, messages=1,nproc='ncpu')
net_32test_mode.train_tnc(input_test32[arr[0:20000],:],
result_32[arr[0:20000],1], maxfun = 5000, messages=1,nproc='ncpu')
net_32test_median.train_tnc(input_test32[arr[0:20000],:],
result_32[arr[0:20000],3], maxfun = 5000, messages=1,nproc='ncpu')

```



```

net_32test_sd.train_tnc(input_test32[arr[0:20000],:],
result_32[arr[0:20000],4], maxfun = 5000, messages=1,nproc='ncpu')
print

PLT.scatter(result_8[arr[20000:],0],net_8test_ent(input_test8[arr[2000
0:],:]))
PLT.scatter(result_8[arr[20000:],1],net_8test_mean(input_test8[arr[200
00:],:]))
PLT.scatter(result_8[arr[20000:],2],net_8test_mode(input_test8[arr[200
00:],:]))
PLT.scatter(result_8[arr[20000:],3],net_8test_median(input_test8[arr[2
0000:],:]))
PLT.scatter(result_8[arr[20000:],4],net_8test_sd(input_test8[arr[20000
:],:]))

#####make function to optimise for min
##generate random values to allow for repeatable results
##samples are based on the cumulative inverse of these
rand_nums=np.reshape(np.random.random(100*np.size(m0)*32),(np.size(m0)
*100,32))
from scipy.stats import binom
##function to find probabilities based on prior threshold and scale
params
def psy_p2(XX):
    XX=XX[:,np.newaxis]
    return np.transpose(1.0/3.0+2.0/3.0*1.0/(1.0+np.exp((- (XX-
m0)/(0.005*10**(m1*2))))))

##function to find probabilities based on SPECIFIED threshold (m) and
scale (w) params
def psy_p(XX,m,w):
    return np.transpose(1.0/3.0+2.0/3.0*1.0/(1.0+np.exp((- (XX-
m)/(0.005*10**(w*2))))))

##find approximate MSE based on simulations for 8 design points
def NN8_MSE(x):
    x=np.sort(x)
    #penalty for outside range
    penalty=1.*(np.max(x)>1. or np.min(x)<0.)
    probs_temp=np.tile(psy_p2(x),(100,1))
    #y=binom.rvs(1,probs_temp)
    y=binom.isf(rand_nums[:,0:8],1,probs_temp)
    inputs=np.tile(x,(np.size(m0)*100,1))*(2.0*(y-.5))
    mse_est=np.mean((np.tile(m0,100)-
net_8test_mean(inputs)[: ,0])**2)
    return mse_est+penalty*9999.

##find approximate MSE based on simulations for 8 design points
##note also takes argument for previous 8 observed responses
def NN16_MSE(x,input_obs):
    probs_temp=np.tile(psy_p2(x),(100,1))
    penalty=1.*(np.max(x)>1. or np.min(x)<0.)
    #y=binom.rvs(1,probs_temp)
    y=binom.isf(rand_nums[:,8:16],1,probs_temp)
    inputs=np.tile(x,(np.size(m0)*100,1))*(2.0*(y-.5))
    inputs2=np.append(np.tile(input_obs,(np.size(m0)*100,1)),inputs,
axis=1)

```

```

        col_order=np.argsort(np.abs(inputs2[0,:]))
        inputs3=inputs2[:,col_order]
        mse_est=np.mean((np.tile(m0,100)-
net_16test_mean(inputs3[:,0])**2)
        return mse_est+penalty*9999.

##find approximate MSE based on simulations      for 8 design points
##note also takes argument for previous 16 observed responses
def NN24_MSE(x,input_obs):
    penalty=1.*(np.max(x)>1. or np.min(x)<0.)
    probs_temp=np.tile(psy_p2(x),(100,1))
    #y=binom.rvs(1,probs_temp)
    y=binom.isf(rand_nums[:,16:24],1,probs_temp)
    inputs=np.tile(x,(np.size(m0)*100,1))*(2.0*(y-.5))
    inputs2=np.append(np.tile(input_obs,(np.size(m0)*100,1)),inputs,
axis=1)
    col_order=np.argsort(np.abs(inputs2[0,:]))
    inputs3=inputs2[:,col_order]
    mse_est=np.mean((np.tile(m0,100)-
net_24test_mean(inputs3[:,0])**2)
    return mse_est+penalty*9999.

##find approximate MSE based on simulations      for 8 design points
##note also takes argument for previous 24 observed responses
def NN32_MSE(x,input_obs):
    penalty=1.*(np.max(x)>1. or np.min(x)<0.)
    probs_temp=np.tile(psy_p2(x),(100,1))
    #y=binom.rvs(1,probs_temp)
    y=binom.isf(rand_nums[:,24:32],1,probs_temp)
    inputs=np.tile(x,(np.size(m0)*100,1))*(2.0*(y-.5))
    inputs2=np.append(np.tile(input_obs,(np.size(m0)*100,1)),inputs,
axis=1)
    col_order=np.argsort(np.abs(inputs2[0,:]))
    inputs3=inputs2[:,col_order]
    mse_est=np.mean((np.tile(m0,100)-
net_32test_mean(inputs3[:,0])**2)
    return mse_est+penalty*9999.

##find intial design for first 8 points
from scipy.optimize import minimize
s_NN8=minimize(NN8_MSE,np.arange(8)/7.,method='Powell',tol=0.01)
## take given threshold and scale params and simulate
## an adaptive design based on NN approximations
def adaptive_sim(m0_t,m1_t):
    ##get initial design points from global var s_NN8
    obs_x=s_NN8.x
    ##random sample of detection given design and params
    obs_y=binom.rvs(1,psy_p(obs_x,m0_t,m1_t))
    ##transform to input into NN
    obs_inputs=obs_x*(2.0*(obs_y-.5))
    ##find next 8 design points assuming 8 observed
    s_NN16=minimize(NN16_MSE,np.arange(8)/7.,method='Powell',tol=0.0
1,args=((obs_inputs),))
    ##generate and append obs
    obs_x=np.append(obs_x,s_NN16.x)
    obs_y=np.append(obs_y,binom.rvs(1,psy_p(s_NN16.x,m0_t,m1_t)))
    obs_inputs=obs_x*(2.0*(obs_y-.5))
    ##find next 8 design points assuming 16 observed
    s_NN24=minimize(NN24_MSE,np.arange(8)/7.,method='Powell',tol=0.0
1,args=((obs_inputs),))
    ##generate and append obs

```

```

        obs_x=np.append(obs_x,s_NN24.x)
        obs_y=np.append(obs_y,binom.rvs(1,psy_p(s_NN24.x,m0_t,m1_t)))
        obs_inputs=obs_x*(2.0*(obs_y-.5))
        ##find next 8 design points assuming 24 observed
        s_NN32=minimize(NN32_MSE,np.arange(8)/7.,method='Powell',tol=0.0
1,args=((obs_inputs),))
        ##generate and append obs
        obs_x=np.append(obs_x,s_NN32.x)
        obs_y=np.append(obs_y,binom.rvs(1,psy_p(s_NN32.x,m0_t,m1_t)))
        obs_inputs=obs_x*(2.0*(obs_y-.5))
        return obs_inputs

##try 10 runs of adaptive assuming threshold=0.5 scale=0.4
adaptive_NN_res=np.reshape(np.repeat(0.,10*32),(10,32))
for i in xrange(10):
    adaptive_NN_res[i,:]=adaptive_sim(.5,.4)

##functions to allow for array inputs so can use PSwarm
import playdoh as pd
def myfun8(p):
    z=np.apply_along_axis(NN8_MSE, 0, p)
    return(z)

def myfun16(p,a):
    z=np.apply_along_axis(NN16_MSE, 0, p,a)
    return(z)

def myfun24(p,a):
    z=np.apply_along_axis(NN24_MSE, 0, p,a)
    return(z)

def myfun32(p,a):
    z=np.apply_along_axis(NN32_MSE, 0, p,a)
    return(z)

## function as above put uses PSwarm global solver
## rather than Powell's
def adaptive_sim2(m0_t,m1_t):
    obs_x=s_NN8.best_pos
    obs_y=binom.rvs(1,psy_p(obs_x,m0_t,m1_t))
    obs_inputs=obs_x*(2.0*(obs_y-.5))
    s_NN16=pd.minimize(myfun16,popsize=10,maxiter=10,cpu=8,initrange
=(np.tile([0., 1.], (8, 1))),args=((obs_inputs),))
    obs_x=np.append(obs_x,s_NN16.best_pos)
    obs_y=np.append(obs_y,binom.rvs(1,psy_p(s_NN16.best_pos,m0_t,m1_
t)))
    obs_inputs=obs_x*(2.0*(obs_y-.5))
    s_NN24=pd.minimize(myfun24,popsize=10,maxiter=10,cpu=8,initrange
=(np.tile([0., 1.], (8, 1))),args=((obs_inputs),))
    obs_x=np.append(obs_x,s_NN24.best_pos)
    obs_y=np.append(obs_y,binom.rvs(1,psy_p(s_NN24.best_pos,m0_t,m1_
t)))
    obs_inputs=obs_x*(2.0*(obs_y-.5))
    s_NN32=pd.minimize(myfun32,popsize=10,maxiter=10,cpu=8,initrange
=(np.tile([0., 1.], (8, 1))),args=((obs_inputs),))
    obs_x=np.append(obs_x,s_NN32.best_pos)
    obs_y=np.append(obs_y,binom.rvs(1,psy_p(s_NN32.best_pos,m0_t,m1_
t)))
    obs_inputs=obs_x*(2.0*(obs_y-.5))
    return obs_inputs

```

```

##run some simulations for different thresholds (m0_NN)
## and scale (m1_NN) parameters
adaptive_NN_res2=np.reshape(np.repeat(0.,900*32),(900,32))
m0_NN=np.tile(np.repeat(np.arange(.1,.91,.4),100),3)
m1_NN=np.repeat(np.arange(.1,.91,.4),300)
for i in xrange(900):
    adaptive_NN_res2[i,:]=adaptive_sim2(m0_NN[i],m1_NN[i])

##append final estimate of threshold after adaptive design
## with true thresh and scale parameter
results_NN_est=np.reshape(np.repeat(0.,np.size(m0_NN)*3),(np.size(m0_NN),3))
results_NN_est[:,0]=m0_NN
results_NN_est[:,1]=m1_NN
results_NN_est[:,2]=net_32test_mean(adaptive_NN_res2[:,0])
##save results as csv file
np.savetxt("results_NN_est.csv", results_NN_est, delimiter=",")

```