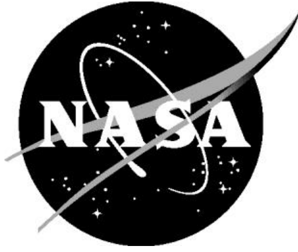
NASA/TM–2016-219174



Statistical Emulator for Expensive Classification Simulators

Jerret Ross
Wichita State University, Wichita, Kansas

Jamshid A. Samareh
Langley Research Center, Hampton, Virginia

April 2016

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM–2016-219174

Statistical Emulator for Expensive Classification Simulators

Jerret Ross
Wichita State University, Wichita, Kansas

Jamshid A. Samareh
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

April 2016

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Contents

1 Introduction **1**

2 Previous Work **2**
2.1 Gaussian Processes 3
2.2 Emulation 4
2.2.1 Bayesian Methods 5
2.2.2 The Emulator 5
2.2.3 Building an Emulator 6
2.3 Gaussian Process Emulators 6
2.3.1 Overview 6
2.3.2 Smoothness 6
2.3.3 Higher Dimensions 7
2.3.4 Design 8
2.3.5 Approximate Analysis 8
2.3.6 Bayesian Calibration 8
2.3.7 Extensions and Challenges 8
2.4 Sampling Criteria 9
2.4.1 When to Use an Emulator 10
2.4.2 Types of Inverse Problems 10
2.4.3 Calculations 11
2.4.4 Sampling Criteria 11
2.4.5 Computational Effort 13
2.5 Bayesian Optimization 13

3 Details **15**
3.1 Algorithm 15
3.2 Emulator 16
3.3 Sampling Criteria 16
3.4 Bayesian Optimization 17

4 Experiments **17**
4.1 Toy Problems 17
4.1.1 Line 18
4.1.2 Single Circle 21
4.1.3 Disconnected Sets of Circles 24
4.2 Computational Fluid Dynamics 27

5 Discussion **29**
5.1 When to Optimize 29
5.1.1 Optimization Alternative 30
5.2 Kernel Function Selection 31

A Code for Toy Experiments **34**

B Code for FUN3D Experiment **48**

C FUN3D Images **63**

List of Figures

1 Box plot of accuracy of 200 experiments of a line model 19
2 Example of the development of the Emulator(left) and Entropy(right) for point totals 3, 10, 20, 30 and 35, with optimization performed at every third point. 20
3 Box plot of accuracy of 120 experiments of a single circle model . 22
4 Example of the development of the Emulator(left) and Entropy(right) for point totals 3, 10, 20, 35 and 65 (top to bottom) with optimization performed at every tenth point. 23
5 Box plot of accuracy of 50 disjointed circle model experiments . . 25
6 Example of the development of the Emulator(left) and Entropy(right) for point totals 3, 10, 20, 55 and 125 (top to bottom) with optimization performed at every fifteenth point. 26
7 FUN3D Plot with CFL set to 500 28
8 FUN3D Accuracy Plot per Point 29
9 Example of the development of the Emulator(left) and Entropy(right) for point totals 10, 12, 14, 16 and 18 points with out optimization performed on the FUN3D simulator. 64
10 Example of the development of the Emulator(left) and Entropy(right) for point totals 20, 22, 24, 26 and 28 points with out optimization performed on the FUN3D simulator. 65
11 Example of the development of the Emulator(left) and Entropy(right) for point totals 30, 32, 34, 36 and 38 points with out optimization performed on the FUN3D simulator. 66
12 Example of the development of the Emulator(left) and Entropy(right) for point totals 40, 42, 44, 46 and 48 points with out optimization performed on the FUN3D simulator. 67
13 Example of the development of the Emulator(left) and Entropy(right) for point totals 50, 52, 54, 56 and 58 points with out optimization performed on the FUN3D simulator. 68
14 Example of the development of the Emulator(left) and Entropy(right) for point totals 60, 62, 64, 66 and 68 points with out optimization performed on the FUN3D simulator. 69
15 Example of the development of the Emulator(left) and Entropy(right) for point totals 70, 72, 74, 76 and 78 points with out optimization performed on the FUN3D simulator. 70
16 Example of the development of the Emulator(left) and Entropy(right) for point totals 80, 82, 84, 86 and 88 points with out optimization performed on the FUN3D simulator. 71
17 Example of the development of the Emulator(left) and Entropy(right) for point totals 90, 92, 94, 96 and 98 points with out optimization performed on the FUN3D simulator. 72
18 Example of the development of the Emulator(left) and Entropy(right) for point totals 100, 102, 1094, 106 and 108 points with out optimization performed on the FUN3D simulator. 73

19 Example of the development of the Emulator(left) and Entropy(right) for point totals 110, 112, 114, 116 and 118 points with out optimization performed on the FUN3D simulator. 74
20 Example of the development of the Emulator(left) and Entropy(right) for point totals 120, 122, 124, 126 and 128 points with out optimization performed on the FUN3D simulator. 75
21 Example of the development of the Emulator(left) and Entropy(right) for point totals 130, 132, 134, 136 and 138 points with out optimization performed on the FUN3D simulator. 76
22 Example of the development of the Emulator(left) and Entropy(right) for point totals 140, 142, 144, 146 and 148 points with out optimization performed on the FUN3D simulator. 77
23 Example of the development of the Emulator(left) and Entropy(right) for point totals 150, 152, 154, 156 and 158 points with out optimization performed on the FUN3D simulator. 78
24 Example of the development of the Emulator(left) and Entropy(right) for point totals 160, 162, 164, 166 and 168 points with out optimization performed on the FUN3D simulator. 79
25 Example of the development of the Emulator(left) and Entropy(right) for point totals 170, 172, 174, 176 and 178 points with out optimization performed on the FUN3D simulator. 80
26 Example of the development of the Emulator(left) and Entropy(right) for point totals 180, 182, 184, 186 and 188 points with out optimization performed on the FUN3D simulator. 81
27 Example of the development of the Emulator(left) and Entropy(right) for point totals 190, 192, 194, 196 and 198 points with out optimization performed on the FUN3D simulator. 82
28 Example of the development of the Emulator(left) and Entropy(right) for point totals 200, 202, 204, 206 and 208 points with out optimization performed on the FUN3D simulator. 83

Abstract

Expensive simulators prevent any kind of meaningful analysis to be performed on the phenomena they model. To get around this problem the concept of using a statistical emulator as a surrogate representation of the simulator was introduced in the 1980's. Presently, simulators have become more and more complex and as a result running a single example on these simulators is very expensive and can take days to weeks or even months. Many new techniques have been introduced, termed criteria, which sequentially select the next best (most informative to the emulator) point that should be run on the simulator. These criteria methods allow for the creation of an emulator with only a small number of simulator runs. We follow and extend this framework to expensive classification simulators.

1 Introduction

This paper discusses the problem of how to speed up the evaluation of expensive physical classification simulators. In this scenario a simulator is given an n-dimensional input and outputs a binary classification. The interpretation of the classification value depends on the simulator of interest. For example, the Computation Fluid Dynamics simulator we used in section 4.2 of this paper will output a 1 if the simulator converges to some desired value within a set amount of iterations and a 0 otherwise.

A complex simulator is computationally expensive, and it is infeasible to perform any kind of Monte Carlo analysis (methods requiring hundreds of thousands or millions of evaluations). Simulators that take days or hours fit into this definitions. In addition, simulators that only take a few minutes to evaluate could make Monte Carlo based analysis impractical. For example, a million simulations would take approximately 694 days for a simulator that only takes a single minute to evaluate.

There is ample research focusing on the problem of expensive simulators [1–5] and the subject is still a field of active research [6–8]. Previous research has focused on simulators with multidimensional real valued inputs [4] and single real valued outputs as well as multidimensional real valued inputs and multi dimensional real valued outputs [3]. However we have not come across any research that evaluates simulators with multidimensional real valued inputs and classification outputs. The lack of research focusing on expensive classification simulators was overcome by looking at the framework developed for expensive simulators with multi-dimensional inputs and real-valued outputs and making some modifications.

The framework for expensive simulators with multi-dimensional inputs and real-valued outputs is as follows. A statistical emulator is used as a surrogate function that represents the simulator, in other words we use the emulator as a model of the simulator. Next, in a point by point manor, a function called the Sampling Criteria [5, 9] is optimized and returns the next candidate point to be evaluated on the simulator. The point that was just evaluated is added to the

emulator, points that have actually been evaluated on the simulator are known as design points. This process is repeated until the budget for the evaluations of the simulator has been exhausted. See algorithm 2 for the pseudo-code.

In this paragraph we discuss the handling of expensive simulators as developed in previous works. First a Gaussian Process Regression model is used as the emulator due to desirable properties [4] inherent in the model that other methods, such as neural networks, do not posses. Next a Sampling Criteria needs to be chosen. This is not a simple task due to the many Sampling Criteria functions that have been developed. Each Sampling Criteria is designed to balance the trade off between exploration versus exploitation. Selection of Sampling Criteria is a problem which depends on each specific use case. Lastly the Sampling Criteria must be optimized to pick the next best point. While optimization is a difficult problem, a Sampling Criteria usually takes less time to optimize than it does to evaluate a point on the simulator. The method discussed in this paper is not well-suited for simulators that take less time to run than the time taken to optimize the Sampling Criteria function and updating the Emulator.

An overview of the modifications we made to the above mentioned framework are as follows. For the Emulator we use a Gaussian Process Classifier. This introduces computational complexity not experienced with the Gaussian Process Regression model for reasons that will be discussed in section 2. We use a Sampling Criteria named the Expected Average Entropy [5,9] which has been used as a Sampling Criteria in the case of simulators with real valued outputs. Again difficulties are introduced due to the modifications we make to the framework with respect to how the Expected Average Entropy Sampling Criteria behaves. Finally we use Bayesian optimization [10] as an optimizer to our Sampling Criteria of choice.

The paper is arranged as follows. In section 2 we discuss a brief history of statistical emulators as well as detail the topic of Sampling Criteria and the optimization required to get the next best point from the sampling criteria. In section 3 we discuss our extension to these methods as well as giving implementation details. In section 4 we present a demonstration of our methods on a few simple examples and a two-dimensional computational fluid dynamics simulator. In section 5 we discuss problems we experienced when using our method as well as some possible alternative details to overcome those problems.

2 Previous Work

In this section we discuss the fundamental pieces that we incorporated in our solution. We review these subjects individually and direct the interested readers to relevant previous research pertaining to each concept.

2.1 Gaussian Processes

The Gaussian process is fundamental to our solution implementation. We use a total of three Gaussian processes: two of them we use directly and the other we use indirectly. We use a Gaussian Process Classification model as our Emulator and a Gaussian Process Regression model to model the entropy of our emulator. We then use the entropy model to calculate the average entropy required by our Sampling Criteria method. Lastly we use Bayesian Optimization to find the maximum of our Sampling Criteria. A Gaussian Process regression model is used in Bayesian optimization to model the Sampling Criteria, and then the optimum of the Gaussian Process model is found with an acquisition function. While the optimization process is performed in a python package, GPyOpt [11], we still appreciate the fact that a Gaussian Process is fundamental to the solution that gives us the motivation to briefly discuss Gaussian Processes.

In this section we discuss the Gaussian Process. Since the Gaussian Process is a fundamental model in our work we believe that giving a brief overview of the subject is prudent.

A Gaussian Process is a generalization of the Gaussian probability distribution. While a probability distribution describes random variables which can be scalars or vectors, a process describes properties of functions. Thus a Gaussian process is a distribution over functions instead of individual points or vectors.

A Gaussian process is a Bayesian model that is composed of several parts. First an assumption is made on what form the input data will take. This assumption is referred to as the prior and takes the form of a covariance function. Several covariance functions exist and allow for data assumptions ranging from very smooth and continuous data to periodic and non-stationary data. The next Element is the likelihood probability. For a Gaussian process, a regression likelihood model takes the form of a Gaussian distribution. A third element, the marginal likelihood, takes the form of the integral over the product of the likelihood and prior. Usually this integral is intractable but in the above case it turns out that the computation can be made analytically and thus tractability. The reason for the tractability of the product of the Gaussian distribution and Gaussian process is due to the fact that a Gaussian Process is a conjugate prior of a Gaussian distribution. This means the product of a Gaussian distribution and a Gaussian Process is in fact a Gaussian Process itself. The above can be seen in Bays Rule.

$$p(y|x) = \frac{p(x|y)p(y)}{\int p(x|y)p(y)dy} \tag{1}$$

$p(x|y)$ is the likelihood, $p(y)$ is the prior, $p(y|x)$ is the posterior probability, and finally $\int p(x|y)p(y)dy$ is the marginal likelihood.

While the solution for regression using Gaussian Processes is analytical and exact, difficulty arises in the classification case. We discuss these difficulties because our problem is fundamentally a classification problem and we use Gaussian Process Classification. In the classification scenario the likelihood function takes the form of a Bernoulli distribution. This requires an integration (or

sum in the discrete case) over the product of a Gaussian Process and Bernoulli distribution. Unfortunately a Gaussian process is not a conjugate prior of the Bernoulli distribution; the product of the two creates an intractable distribution that is approximated using a Gaussian Processes. There are several iterative algorithms that exist to calculate this approximation. The most well known algorithm is the Laplace Approximation. We do not use this algorithm due to the algorithms poor accuracy performance [12]. The next most popular algorithm is the Expectation Propagation algorithm, which we choose to use and is widely considered to have good performance. The details of this algorithm are extensive and can be found in the classification chapter of [12].

We have barely touched the topic of Gaussian processes but the interested reader can find information on Gaussian Process regression, Gaussian Process Classification, Covariance functions and many other topics in [12].

2.2 Emulation

The ideas in this section where taken from [4]. While [4] is not necessarily the origin of these ideas, it presents them in an easily understandable way and should be a first stop for anybody interested in Bayesian statistical emulation.

The main idea of [4] is to use Bayesian statistics to create a statistical emulator of a simulation model. The term given to this method is Bayesian Analysis of Computer Code Outputs (BACCO) and consists of a methodology to employ emulators to address a wide range of practical questions of complex model behavior. The complex models we are interested in, and focused on in [4], come in the form of simulators. Simulators are used to model the behavior of real world systems and are utilized in almost all fields of science and technology. Simulators can be deterministic or stochastic. In our case we focus on deterministic simulators although work has been done in the domain of stochastic simulators with regard to the BACCO method. We can regard our deterministic simulator as a function $f(\cdot)$ which takes a vector $\mathbf{x}$ and produces an output $\mathbf{y} = f(\mathbf{x})$.

It is important to remember that $\mathbf{y}$ is a prediction of a real-world phenomena that is being simulated by a model of said phenomena. The consequence of this is that the prediction will probably be imperfect and there will be uncertainty about how close the prediction is to the true real-world quantities. While we did not take this into consideration one can refer to [1] for a complete taxonomy of the uncertainties involved in using simulators.

In the majority of cases the output, $\mathbf{y}$, is real valued. In our case the output of our simulator is binary. Extending the BACCO framework for a binary simulator is actually one of the original contributions of our work and will be discussed else where in the paper. We decided to express the ideas in [4] unaltered, dealing with real valued output, in the hopes that others will find the ideas useful for their own endeavors.

2.2.1 Bayesian Methods

This section discusses what Bayesian statistical methods are and how they compare to frequentist methods for our problem of interest.

Frequentist statistics is a method for interpreting the probability of an event as the number of times that event occurs approaches infinity. This limits the usefulness of frequentist method to events that are repeatable an indefinite number of times. Uncertainty of repeatable events is referred to as *aleatory* uncertainty. Since simulators are an approximation of a physical phenomena the uncertainty is due to a lack of knowledge about the particular phenomena. The type of uncertainty that is due to a lack of knowledge is call *epistemic* uncertainty. It is true that Almost all uncertainties in the analysis of simulator outputs are epistemic. For further discussion of how the distinction between aleatory and epistemic uncertainty is fundamental between frequentist and Bayesian statics [13] page 3.

Bayesian statistics is based on a much broader definition of probability. All techniques described in this paper are based in the Bayesian framework and will be discussed in detail as concepts are introduced.

2.2.2 The Emulator

While not of interest to our work many find it necessary to perform sensitivity analyses as well as uncertainty analyses on their simulators for model development as well as model use. These analysis tools require a large number of simulation runs which quickly becomes intractable. For example if a simulator takes just one second to run it would take 11.6 days of continuous CPU time to perform a comprehensive variance-based sensitivity analysis due to the need to run millions of model runs to achieve a desired accuracy. A more efficient method is needed and Bayesian methods are more efficient when used for emulation.

An emulator is a statistical approximation of the simulator. Because we have designated our simulator to be a function $f(\cdot)$ such that given $\mathbf{x}$ we can find a mapping to the output $y = f(\mathbf{x})$. It is instructive to think of an approximate function $\hat{f}(\cdot)$ of the simulator $f(\cdot)$ for use in such tasks as sensitivity or uncertainty analysis. While we do not consider using such an approximate function for these analysis tasks, it is easy to see that we could benefit from the efficiency of the approximate function due to the large expense incurred with each run or our simulator. If the approximation can be good enough then analysis, such as sensitivity and uncertainty, will be close enough to any analysis performed on the simulator itself but with much less time needed.

It should be noted that $\hat{f}(\cdot)$ is a statistical approximation and thus provides an entire probability distribution for $f(\mathbf{x})$. The mean of $f(\mathbf{x})$ can be interpreted as the approximation $\hat{f}(\cdot)$. Importantly an extra source of information comes from the use of statistical approximations. This information comes in the form of a interval around the mean which represents the uncertainty of the approximation of the simulator $f(\mathbf{x})$. In fact the emulator gives a probability

distribution across the entire function $f(\cdot)$.

2.2.3 Building an Emulator

An emulator is a statistical approximation to the simulator as well as the fact that it is used to estimate $f(\cdot)$ given a set of data in the form of training runs $y_i = f(\mathbf{x}_i)$ for $i = 1, \cdots, n$.

When building a simulator the following two criteria should be satisfied.

1. At a *design point* $\mathbf{x}_i$, the emulator should reflect the fact that we know the true value of the simulator output, so it should return $\hat{f}(\mathbf{x}_i) = y_i$ with no uncertainty.

2. At other points, the distribution for $f(\mathbf{x})$ should give a mean value $\hat{f}(\mathbf{x})$ that represents a plausible interpolation or extrapolation of the training data, and the probability distribution around this mean should be a realistic expression of the uncertainty about how the simulator might interpolate/extrapolate.

While criterion 1 is easily checked criterion 2 requires simulator runs for verification. If the simulator under investigation is very expensive then verification of criterion 2 is not feasible. Two methods mentioned that fail criterion 2 are regressions methods and neural networks. The methods that does meet both criteria are Gaussian processes. For regression, Gaussian processes are analytically very tractable while for classification approximation methods must be performed but are still very efficient.

A concept termed *code uncertainty* refers to the discrepancy created by performing statistical analysis of a simulator on the emulator. Since this concept does not pertain much to our work we omit further discussion and refer the readers interested in a comparison of BACCO and Monte Carlo methods to [4].

2.3 Gaussian Process Emulators

2.3.1 Overview

A Gaussian process is an extension of the normal distribution. While a multivariate normal distribution is a distribution for several variables, each having a marginally normal distribution, a Gaussian Process is a distribution over functions. One can think of a point $f(\mathbf{x})$ of a Gaussian Process as having a normal distribution itself. The mean of this distribution serves as the estimate of the function being approximated while the variance, or spread, of the distribution at that point serves as the uncertainty. The higher the variance at that point the larger the uncertainty there is at that point.

2.3.2 Smoothness

Gaussian Processes assume a certain level of smoothness to the function $f(\cdot)$ that is being approximated. This smoothness can range *a priori* with the choose

of covariance function as well as with the use of smoothness hyperparameters. See [12] for more details on both.

The concept of smoothness gives Gaussian processes a major advantage over Monte Carlo methods with respect to computation. If it is known that $f(x) = 1$ for $x = 3$ then, due to the smoothness assumption of Gaussian processes, it is known that all values near $f(x = 3)$ will be close to one. This is why there is less uncertainty as we evaluate near a known point. The smoothness assumption also accounts for the decrease in uncertainty as the number of design points increases simply due to the decrease of the distance of two points.

The inability of Monte Carlo methods to utilize the distance between two points accounts for the greater efficiency of using a Gaussian Process as an approximation of a function.

Degree of smoothness is also an important part of BACCO emulation. The reason is that the smoother a function is the more efficient the BACCO method is. Degree of smoothness can be thought of as how rapidly a function varies. A non-smooth function is more sensitive to small changes in the input space and many more data points are needed to emulate a non-smooth function accurately.

Selecting smoothness hyperparameters can be tricky. If a smoothness parameter is too high then the emulator will make over confident predications with respect to the uncertainty. If the smoothness parameter is too low then the emulator with predict values with overstated uncertainty.

2.3.3 Higher Dimensions

All properties described thus far still hold for higher dimensions. The certainty near an actual data point increases as we approach the data point, and the uncertainty increases rapidly as we get further from any data point. As more points are added, the overall uncertainty is reduced and the approximation adapts itself to the shape of the true function. Smoothness hyperparameters are used for each dimension. Again the values of the parameters are crucial for an accurate and robust emulator.

An important question to ask is how much does the computational cost increase with the increase of dimensionality? How does the number of training points needed to approximate the true function grow with the number of dimensions in the input? In practice models never respond strongly to all of their inputs. This means a high level of smoothness is seen in all but a few dimensions. Remembering from earlier that the smoother a function or dimension is, the fewer number of training points are needed to approximate it. For example if there are only 200 test points and 25 five dimension then the amount of space covered by those test points is very sparse. However if only 5 of those dimensions influence the output greatly, then it might be possible to approximate that function fairly well using a Gaussian Process emulator. This reflects recent work in Deep Learning, which states that the curse of dimensionality might be an illusion and the actual problem is the amount of variance in a function.

2.3.4 Design

A design is a set of input points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ at which the simulator is evaluated to get training data. The objective is to learn about the true function $f(\cdot)$ and the question is how do we select the best design to do that. In our case, we want the size of the design to be as small as possible. To learn the space with the smallest number of design points requires a fairly advanced method that we will discuss later in this paper. For applications with less stringent design size, it is possible to use a set of N Latin Hypercube samples. There are many other types of methods for determining the design set. Some designs are selected sequentially, point by point, while others are decided before a model is even created. Design selection is a current hot topic of research and has been for at least a decade. The interested reader should see [14] page 10 for a starting point.

2.3.5 Approximate Analysis

The reader interested in a comparison of the BACCO and Monte Carlo methods with regards to approximate uncertainty analysis and sensitivity analysis of a simulator should see [4]. We do not discuss them here because they do not pertain to our work.

2.3.6 Bayesian Calibration

Another interesting idea that we did not work with but has shown to be needed in other groups is Bayesian Calibration. Bayesian Calibration deals with the problem of the discrepancy between a simulators output and real-world observation that the simulator represents approximately. More specifically it is the process of finding the subset of model inputs that best reflect real-world observation. Conventional calibration usually attempts to find this subset via trial and error while holding many input variables fixed and not considering any uncertainty measure. As an alternative to the trial and error method there are Bayesian methods for calibration.

Bayesian Calibration uses a second Gaussian Process to model the discrepancy between the simulator and the real-world observations. We use a method similar to this by using a second Gaussian process. Our method differs in that the second Gaussian process is used to model the Shannon entropy of the emulator. We then calculate the the expected average entropy of the second Gaussian process in order to choose the next best point in our design. For more details on Bayesian calibration see [1, 15].

2.3.7 Extensions and Challenges

There are several open questions presented in the BACCO paper [4]. Since this paper was written in 2004 there has been advancements in many of the challenges but we will discuss them shortly here for the interested readers.

- *Computation* Prediction in Gaussian Processes requires a matrix inversion, which has a run time of $O(n^3)$. This does not scale well, and Sparse Gaussian Process methods have been created to reduce this runtime [16, 17]. Also experimentation with GPU's has taken place in an attempt to overcome the inversion bottleneck [18].

- *Smoothness* Work is being done that allows for uncertainty in smoothness parameters with increased computational complexity. Choosing the proper smoothness parameters is vital to the creation of a good emulator, and work on choosing better smoothness parameters is needed. As far as we know, very little research on the subject has been published (see 5.1.1 for further details).

- *Multiple outputs* Building separate emulators for each output is a potential solution to this problem but any correlation between the outputs is lost. Newer methods such as [3] and Gaussian Process Latent Variable models (GP-LVM) [19] might be a solution to this problem. We are aware that work has been done in this domain but we are unaware of the details.

- *Discontinuities* Many simulators do not respond smoothly to all of their inputs. One problem with using Gaussian processes as emulators is that they will potentially smooth these discontinuities that give rise to local inaccuracies. General research is being done to address this problem [20] and Chapter 4 of [12].

- *Validation* is defined in the BACCO framework as an estimate of real behavior with uncertainty around the estimate such that comparison with observational data suggests the expressed uncertainty is neither too large nor too small [7, 8].

- *Software* At the time of the writing of [4] in 2004, statistical software for non-specialist statisticians was hard to acquire. Since then packages such as R, GPy [21], GPyOpt [11], kriegInv [9] and many others have become available and most are open source.

2.4 Sampling Criteria

In this section we follow the details presented in [9], which discusses an R package name KrigInv and explains many theoretical properties that we use in our work. We have found that many concepts that we use were originally intended for other uses and also concepts can have several different names. Due to the lack of consistent naming, we have used the terminologies expressed in this paper that reflect how we use them.

Reference [9] discusses a set of methods termed sampling criteria that allow for estimating contour lines and excursions sets. These terms will be defined further down in this section.

Sampling criteria play an important role in our work. We will discuss these criteria as well as the sampling criteria examples discussed in [9]. Some of the advanced techniques that we do not use in our work will be discussed in this section with the intention that discussion of the techniques will be useful to others in the future.

2.4.1 When to Use an Emulator

Emulators are considered to be a potential replacement for simulators, if the following criteria are met:

- No closed-form expression is available for a simulator ($f(\cdot)$). This suggest no information outside of $y = f(x)$ can be obtained, such as gradients.
- The dimension of the input domain $X \subset R^d$ is moderate with d of the order of 20 or less.
- The evaluation budget is small. Evaluating the simulator $f(\cdot)$ at any point is assumed to be slow or expensive so the problem needs to be solved in at most a few hundred evaluations.
- f can be evaluated sequentially. In some cases a small fraction of the evaluation budget is dedicated to the initial design (some small number of points chosen by an algorithm such as Latin hypercube.) The remaining points are evaluated sequentially (one point at a time) at well-chosen points. The next point to evaluate is chosen by optimizing a given *Sampling criteria*.
- Simulators are noisy. Methods for handling a simulator that returns $y = f(\mathbf{x}) + \epsilon$ exist. In our work we only consider deterministic simulators.

2.4.2 Types of Inverse Problems

Sequential sampling strategies aiming at solving the following inverse problems are as follows.

- Estimating the excursion set $\Gamma^* = \{\mathbf{x} \in \mathbb{X} : f(\mathbf{x}) \geq T\}$, where T is a fixed threshold.
- Estimating the volume of excursion: $\alpha^* := \mathbb{P}_{\mathbb{X}}(\Gamma^*)$, also known as the *probability of failure estimation.*
- Estimating the contour line $C^* := \{\mathbf{x} \in \mathbb{X} : f(\mathbf{x}) = T\}$. We are interested in learning the decision boundary of our simulator. Our decision boundary takes the form of the contour line when we set $T = \frac{1}{2}$.

All the above problems are similar and fall under the term of *inversion.*

2.4.3 Calculations

The calculation of all the estimations above is dependent on the calculation of the excursion probability $p_n(\mathbf{x})$. To find the excursion probability we must acquire three items. The mean $m_n(\mathbf{x})$, the standard deviation $s_n(\mathbf{x})$, and the threshold T. With these items we can then calculate the excursion probability $\Phi(\frac{m_n(\mathbf{x})-T}{s_n(\mathbf{x})})$, where $\Phi(\cdot)$ is the c.d.f. of the standard Gaussian distribution. For a more thorough derivation of the excursion probability see page 4 of [9].

With these elements it is now possible to show how the excursion probability is used to calculate the three previous estimations.

$\hat{\Gamma} = \{\mathbf{x} \in \mathbb{X} : p_n(\mathbf{x}) \geq 1/2\}$, $\hat{\alpha} = \int_{\mathbb{X}} p_n(\mathbf{x})d\mathbf{x}$, and $\hat{C} = \{\mathbf{x} \in \mathbb{X} : p_n(\mathbf{x} = 1/2)\} = \{\mathbf{x} \in \mathbb{X} : m_n(\mathbf{x}) = T\}$ are estimators for the excursion set Γ^*, excursion volume α^*, and contour line C^*. For a detailed derivation of these estimators see [22]. In our work we are only interested in estimating the contour line. We add the other estimators for the interested reader.

With these tools it is possible to classify the users model into two classes: those classes could be concepts such as convergence/divergence or inputs that create a simulation close to real world observation/inputs that create unusable simulation.

2.4.4 Sampling Criteria

The aim of a sampling criterion is to give, at each iteration, a point or a set of points for evaluation. In our work we focus on individual points. The following algorithm details the steps we followed to implement our solution that fits into our work.

1. Evaluate f at an initial set of design points $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$,
2. Build an emulator based on $\{f(\mathbf{x}_1), \ldots, f(\mathbf{x}_n)\}$
3. While the evaluation budge is not exhausted:
   - choose the next design point $\mathbf{x}_{n+1}$ by maximizing a given sampling criterion over $\mathbb{X}^r$,
   - evaluate $f(\mathbf{x}_{n+1})$
   - update the emulator

There are many types of sampling criteria that have been developed while many others are currently under development. Sampling Criteria can be broken into two separate categories. We discuss the differences between these categories of sampling criteria for completeness and in the hopes that the information will be helpful to others interested in sampling criteria. The two categories of Sampling Criteria are *pointwise criteria* and *integral criteria*. Pointwise criteria method deals with finding a single next point $\mathbf{x}_{n+1}$ while integral criteria method is best suited for finding a batch of new points. We discuss the characteristic of a small set of Sampling Criteria in to demonstrate the differences between the two types of Sampling Criteria.

Pointwise sampling criterion Three criteria discussed are ranjan, bichon, and tmse. The main objective of these criteria is to find a point $\mathbf{x}_{n+1} \in \mathbb{X}$ such that the excursion probability is close to 1/2 and the variance ${s_n}^2(\mathbf{x}_{n+1})$ is high. We will omit a technical discussion of each criteria and focus our attention on the properties of each criteria. For a more thorough discussion see page 6 of [9].

- **tmse criterion** This criterion aims to decrease the Mean Square Error (the variance) at points where the mean m_n is close to the threshold T. A built in parameter ϵ can be modified to adjust the amount of exploration that is done or the amount of exploitation that takes place.

- **ranjan and bichon criteria** These two criteria are very similar and share a common general expression. They only differ in the default value of a hyper parameter δ. In theory the higher the δ value the more exploration the criteria will attempt. However in practice these two criteria have very similar behavior. These two methods also attempt to find the points with means closest to the threshold and a high variance. The main distinction between these two criterion and the tmse is that the expression to be maximized is an expectation of the difference between two values. The maximization of the expectation of a function is the method that we have chosen for our work.

Integral sampling criteria *Integral criteria* refer to sampling criteria involving numerical integration over the design space $\mathbb{X}$. Three integral criteria are discussed below. All three of the following criteria rely on the concept of *Stepwise Uncertainty Reduction* (SUR) [22]. The basic idea of SUR consists in defining an arbitrary measure of uncertainty given n observations A_n and attempts to find the next point $\mathbf{x}_{n+1}$ that reduces (in expectation) the uncertainty the most. It can be seen that the term uncertainty can have different definitions and thus lead to different sample criteria.

- **timse criterion** Targeted Integrated Mean Square Error Criterion. Originally designed for contour line estimation *timse* can be used for estimation the excursion set as well as its volume. Uncertainty is defined as $Uncertainty^{timse} := \int_{\mathbb{X}} tmse(\mathbf{x})\mathbb{P}_{\mathbb{X}}(d\mathbf{x})$. Where $tmse(\mathbf{x})$ is the same function from the pointwise criteria. The timse attempts to find the point in an interesting region with mean close to the threshold T and high variance.

- **SUR criterion**: Uncertainty is defined as $Uncertainty^{sur} := \int_{\mathbb{X}} p_n(\mathbf{x})(1-p_n(\mathbf{x}))\mathbb{P}_{\mathbb{X}}(d\mathbf{x})$ where $p_n(\mathbf{x})$ is the probability. For a closed form example of the expectation of the uncertainty see [9].

- **jn criterion**: Is used to estimate the excursion volume. Uncertainty is defined as $Uncertainty^{jn} := Var_n(\alpha)$ where α is the set of the random excursion set. *jn* require an integral over $\mathbb{X} \times \mathbb{X}$ which is computationally more expensive than all other criteria. *jn* tends to fill space better than

the other criteria methods and performs well when the excursion set has a complicated shape or is not connected. *jn* tends to evaluate points which are not too close to the boundary of the excursion set.

2.4.5 Computational Effort

As the dimension d increases so does the computational effort to perform the tasks discussed above. Below are some explanations as to what elements increase the computational complexity as the dimensions increase.

- The number of observations n grows as the number of dimensions grows in order to insure proper space filling. In order to calculate the mean and variance of the model, an inversion of an $n \times n$ matrix is required. This operation has a run time of $0(n^3)$ and becomes a large problem as n approaches 1000. There are sparse methods that reduce the runtime down to approximately $O(n^2)$ that might relive this problem. At the same time if the simulator in question is very expensive then 1000 evaluations could be intractable in the first place.
- Optimization of the sampling criteria gets more difficult and requires evaluation at more locations as d grows.
- If an integration criterion is used then the number of integration points needed to maintain a high accuracy increases with the value of d.

As long as the computation time is much less than the computational time for the simulator to complete then these methods are still viable.

Optimization Two major sub-problems discussed above require optimization. First, the sampling strategies require that the sampling criterion be optimized at each iteration. Second, for criteria involving numerical integration the question arises as how do we actually perform the integration.

Sampling criterion optimization Optimizing the sample criterion can be broken into discrete optimization and continuous optimization. In our work we use Bayesian optimization.

2.5 Bayesian Optimization

The majority of the information in this section are taken from [10,23]. Bayesian optimization is used when considering the problem of finding a global maxima (or minima) of some unknown objective function f. The problem can be written as $\mathbf{x}^* = \arg\max_{\mathbf{x} \in X} f(\mathbf{x})$ where X is some design space of interest. Commonly X is a compact subset of R^d but can be categorical or combinatorial as well as other types of spaces which are handled by the Bayesian optimization framework. The assumption that the function f is a *black-box* function with no simple closed

form but can be evaluated at any arbitrary query point $\mathbf{x}$ in the domain is fundamental to Bayesian Optimization. Finally $y = f(\mathbf{x})$ where $y \in R$ can be deterministic or stochastic. If the output of f is noise-corrupted such that $\mathbb{E}[y|f(x)] = f(\mathbf{x})$ The Bayesian optimization framework can still be used.

In general Bayesian optimization is used in a sequential search setting such that at iteration n, the point $\mathbf{x}_{n+1}$ is chosen and f is evaluated at that point to give observation y_{n+1}. The algorithm is given a budget of N iterations and once the budget is exhausted a final recommendation $\bar{\mathbf{x}}_N$ is chosen as the algorithms best estimate.

Bayesian optimization is very data efficient and thus it is found to be useful in situations where the evaluation of the function f is costly, non-convex and/or multi-modal. When this is the case Bayesian optimization is able to take advantage of the information provided by the history of the optimization making the search efficient.

From an abstract view Bayesian optimization has two main components. The first component is a probabilistic surrogate model of the objective function being optimized. This surrogate model is composed of a prior distribution that capturers our assumptions of how the objective function will behave as well as an observation model that describes the data generating mechanism. This statistical model can take many forms and each form depends on the type of problem being optimized. The second component is a loss function which describes how optimal a sequence of queries are. The idea is to minimize the expected loss which is typically computationally intractable. Because of this intractability heuristics have been introduced which are termed acquisition functions.The general algorithm followed in Bayesian Optimization can be found in Algorithm 1.

Acquisition functions trade off exploration and exploitation. The optima of the acquisition function is located where the uncertainty of the probabilistic surrogate model is large and/or where the prediction of the model is high. The next point of the Bayesian optimization method is chosen by finding the maximum of the acquisition function. To make the use of acquisition functions feasible they must be easier to maximize and evaluate than the black-box function f. Since acquisitions functions have analytical forms they fulfill the requirement stated above so we can use them and not have to worry about introducing too much computational overhead to the problem of interest.

Algorithm 1 Bayesian optimization

1: **for** $n = 1, 2, \ldots$ **do**
2: select new $\mathbf{x}_{n+1}$ by optimizing acquisition
3: function α
4: $\mathbf{x}_{n+1} = \arg\max_{\mathbf{x}} \alpha(\mathbf{x}; D_n)$
5: query objective function to obtain y_{n+1}
6: augment data $D_{n+1} = D_n, (\mathbf{x}_{n+1}, y_{n+1})$
7: update statistical model

The probabilistic surrogate model can be parametric or non-parametric. As an example, if the objective function of interest produces a binary output, a Beta-Bernoulli model can be used as the surrogate. Beta, an example of a parametric model, is a conjugate prior to the Bernoulli distribution. This means that the product of the Beta and the Bernoulli distribution is a Beta distribution. In the non parametric case where the objective function has a real valued output, Gaussian process are used as the surrogate model. For a more detailed discussion of the surrogate model where such topics as computation cost as well as how to handle very large data sets is discussed see [10].

The last subject of discussion is the acquisition function. The acquisition function is the policy used for selecting the sequence of query points $\mathbf{x}_{1:n}$. The useful acquisition function will be able to choose a sequence of points that returns a point, $\mathbf{x}_{n+1}$ which is closer to the optimum of the objective function than a random selection of points.

Acquisition functions can be broken into three different categories. These categories are *Improvement-based policies, Optimistic policies,* and *Information-based policies*. These details are beyond the scoop of our work since we used default values for our acquisition function of choice that came from the python package GPyOpt [11]. For a deeper discussion on the types of acquisition functions as well as other practical issues such as handling hyper-parameters, Optimizing acquisition functions and penalization see [10] as a good starting place.

3 Details

In this section we discuss the code Packages we used as well as implementation details for each building block of our solution. The hope is that the interested reader will be able to implement a solution of their own using this document as a guideline. All of our code was written using Python 2.7 as well as all the packages we used that are not in the standard Python distribution and were found on GitHub.

3.1 Algorithm

The basic algorithm we follow is found in algorithm 2. The name was given to the algorithm by our collaborators at the University of Sheffield and is the name of the original regression version of this algorithm that they created. In fact this algorithm is unaltered from their version. Differences are not seen until we actually get into implementation details such as choosing a classification emulator as well as the form the Sampling Criteria function takes.

Remember that a *Design Point* is a point that has actually been evaluated on the simulator and plays the roll of a ground truth point.

Algorithm 2 Entropic Approximate Bayesian Computation

1: Acquire n points from a space filling algorithm
2: Evaluate the n points on simulator
3: Store Design Points $D_n = (\mathbf{x}_{1:n}, y_{1:n})$
4: Create Emulator with Design Points
5: **while** Budget Not Exhausted **do**
6: Select new $\mathbf{x}_{n+1}$ by optimizing sampling criteria
7: Evaluate $\mathbf{x}_{n+1}$ on simulator to obtain y_{n+1}
8: Augment data $D_{n+1} = D_n, (\mathbf{x}_{n+1}, y_{n+1})$
9: Update Emulator

3.2 Emulator

For our Emulator we used a Gaussian Process Classification model. We used the python package GPy [21] from the university of Sheffield as we found it to be the most robust package that we could find. When using the GPy package to create a Gaussian Process Classification model, the user must specify two things: the Covariance (or Kernel) function as well as the Marginal Likelihood approximation algorithm. For the Kernel function we choose the Matern Kernel function provided by the package. We chose to use the Expectation Propagation (EP) algorithm to approximate the Marginal Likelihood. One of the main reason we choose this algorithm over the more popular Laplace Approximation algorithm is that [12] states that the Laplace Approximation performs poorly with respect to accuracy. Also the EP algorithm is the default approximation algorithm for Gaussian Process Classification in the GPy package.

3.3 Sampling Criteria

The Sampling Criteria we chose is based on the Expected Average Entropy Sampling Criteria found in [5,9]. In a regression scenario the Expected average Entropy requires an integral be taken over all of space. This integral is approximated using a Hermite-Gauss Quadrature. The calculation of this integral becomes unnecessary in the case of classification. The formula for the Expected Average Entropy for classification is:

$$p \times \texttt{aveH} + (1 - p) \times \texttt{aveH} \tag{2}$$

Where p is defined as the probability returned by the Emulator that the point of interest $(\mathbf{x}_{n+1})$ will be labeled as 1. The calculations for aveH, the average Entropy, can be found in algorithm 3.

The idea is to find the point $\mathbf{x}_{n+1}$ that reduces the Average Entropy the most on Expectation. To find this point we need to use an optimization technique to minimize our Sampling Criteria, the Expected Average Entropy. We use the python package GPyOpt [11] which performs Bayesian Optimization on our Sampling criteria function over a number of predefined iterations and returns the point that reduces the Expected Average Entropy the most.

Algorithm 3 Calculate Average Entropy

```
1: function AVEH(nextPoint, label, model)
2:     dummyModel = deepCopy(model)
3:     dummyModel.addPoint(nextPoint, label)
4:     points = spaceFilling(size(10000))
5:     probs = dummyModel.predict(points)
6:     for points in probs do
7:         Entropy.append(points × log (points))
       return Entropy.mean()
```

3.4 Bayesian Optimization

When we combine our Sampling Criteria with Bayesian Optimization we get the next point selection function from line 6 of algorithm 2. In practice we have to do very little other than selecting some hyper parameters as well as select which acquisition function we want to choose and the GPyOpt package does the rest. We select these hyper parameters and pass in our Sampling Criteria function and then the package iterates over our Sampling Criteria a predefined number of times and returns the next point to be evaluated on the simulator. The hyper parameters we choose were as follows.

- Acquisition Function: Expected Improvement
- Acquisition Parameter: .001
- Normalize: True
- Optimization restarts: 1
- Model Optimize interval: 30
- Fixed Likelihood Variance at: 10^{-6}
- Max iterations: 150

For a detailed explanation of each hyper parameter see the [11] Github page.

4 Experiments

4.1 Toy Problems

We initially test our method on three well defined simple (toy) problems. These problems are a line, a circle and two disjointed circles. For each toy experiment we test our method several times to evaluate how the variance of the accuracy behaves as the number of evaluation points increases. For the line experiment we run a total of 200 experiments with each experiment being evaluated at 35 points. We run a total of 120 experiments on the circle with a total of 65 points

for each. Lastly we run 50 experiments on the two disjointed circle scenario with 125 point evaluations for each run.

All experiments start with three points that have been selected through a space filling algorithm and are then evaluated on the toy experiment. These points are then used to create a starting emulator. All experiments use optimization on the hyper-parameters of the emulator. The implementation of optimization was different for each experiment and these differences are addressed in the following sections.

We use the Shannon Entropy measure which is a measure of uncertainty. The closer the entropy is to zero the less uncertainty in the model prediction. The closer the entropy gets to .7, the entropy max, the less certain the prediction of the model is.

4.1.1 Line

Our first and simplest experiment is a line defined by the two dimensional formula:

$$f(x_0, x_1) = \begin{cases} 0 & \text{if } 3 \times x_1 - (2 + x_0) < 0 \\ 1 & \text{if } 3 \times x_1 - (2 + x_0) \geq 0 \end{cases}$$

Where the $x_0 \in [-2, 2]$ and $x_1 \in [-2, 2]$ define the square where we limit our interrogation of our method. As the number of evaluated points increases the better our method models the line.

We ran our method on the line experiment around 200 times with each iteration evaluating the experiment up to 35 points. It can be seen in the box plot from figure 1 that as the number of evaluation points increases the accuracy of our emulator approaches 1 (100% accurate) and the variance with respect to the accuracy of all 200 experiments decreases substantially. We optimize our emulator after every third point.

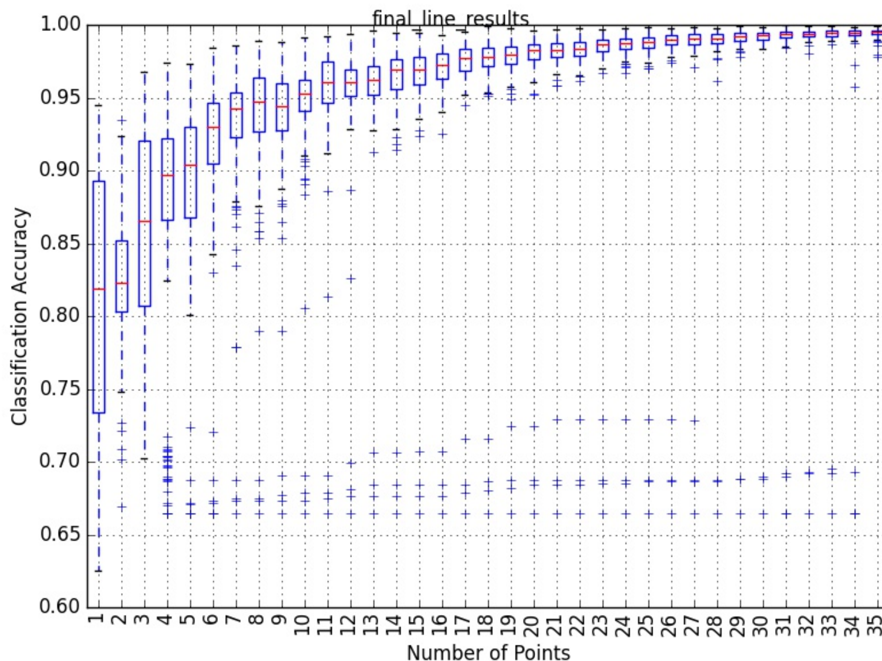


Figure 1: Box plot of accuracy of 200 experiments of a line model

Incremental results are shown as snap shots of the emulator and the entropy field (Fig. 2), where the colors range from white (zero entropy) to purple (.7 value for the entropy). It is apparent that there are many outliers that hover between 70% and 65% accuracy. The reason for this behavior is a problem we faced in many of the experiments. The behavior is from over fitting of the Gaussian Process Classifier when we optimized the model with either few examples of one label and many of the other or no examples of some labels. This brings up the question of how many point evaluations should we wait before we optimize our emulator. For simpler models we can optimize our emulator every few points. As the simulators grow in complexity we must wait longer and longer before we optimize. The problem with this realization is that the object is to use our method on a simulator of unknown complexity. This raises the question of how to handle optimization, should we omit optimizing our model or are there other methods we can implement to overcome this optimization problem?

Figure 2 shows six snap shots of our emulator (seen on the left) and the entropy of our emulator (on the right). As the number of points increases it is easy to see that our emulator does a better and better job and approximating the line. It is also shown that the overall entropy reduces and begins to focus on the decision boundary of the line as more points are evaluated.

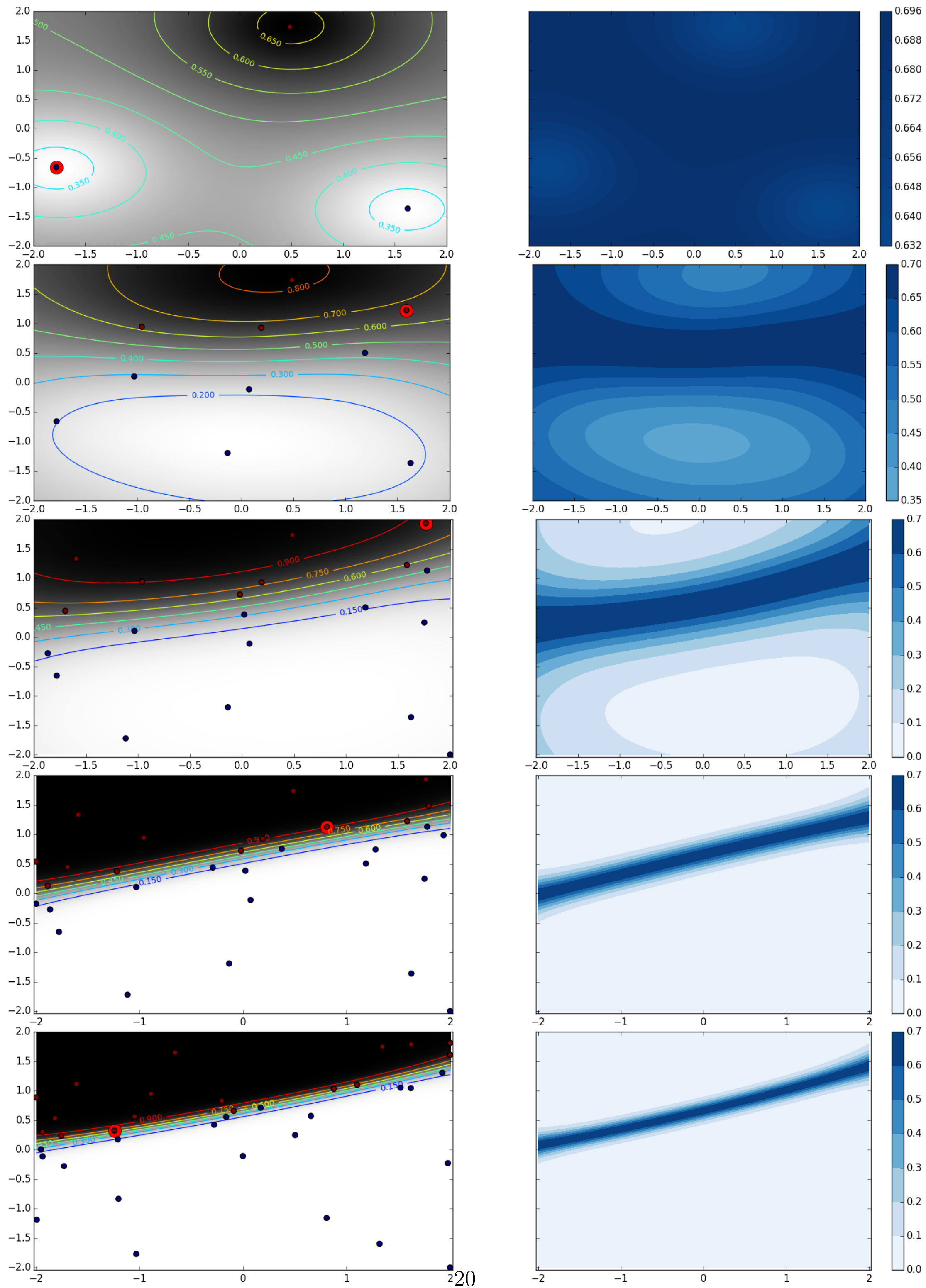


Figure 2: Example of the development of the Emulator(left) and Entropy(right) for point totals 3, 10, 20, 30 and 35, with optimization performed at every third point.

4.1.2 Single Circle

Next we experiment on a single circle defined by the two dimensional formula below and with a center at $(1, 0)$.

$$f(x_0, x_1) = \begin{cases} 1 & \text{if } \sqrt{(x_0 - 1)^2 + x_1^2} > 1 \\ 0 & \text{Otherwise} \end{cases}$$

Where the $x_0 \in [-2, 2]$ and $x_1 \in [-2, 2]$ define the square where we limit our interrogation of our method. As the number of evaluated points increases the better our method models the circle.

We ran our method on the single circle experiment around 120 times with each iteration evaluating the experiment on 65 points. It can be seen in the box plot from figure 3 that as the number of evaluation points increases the accuracy of our emulator approaches 1 and the variance with respect to accuracy of all 120 experiments decreases substantially. We optimized our emulator after each tenth point.

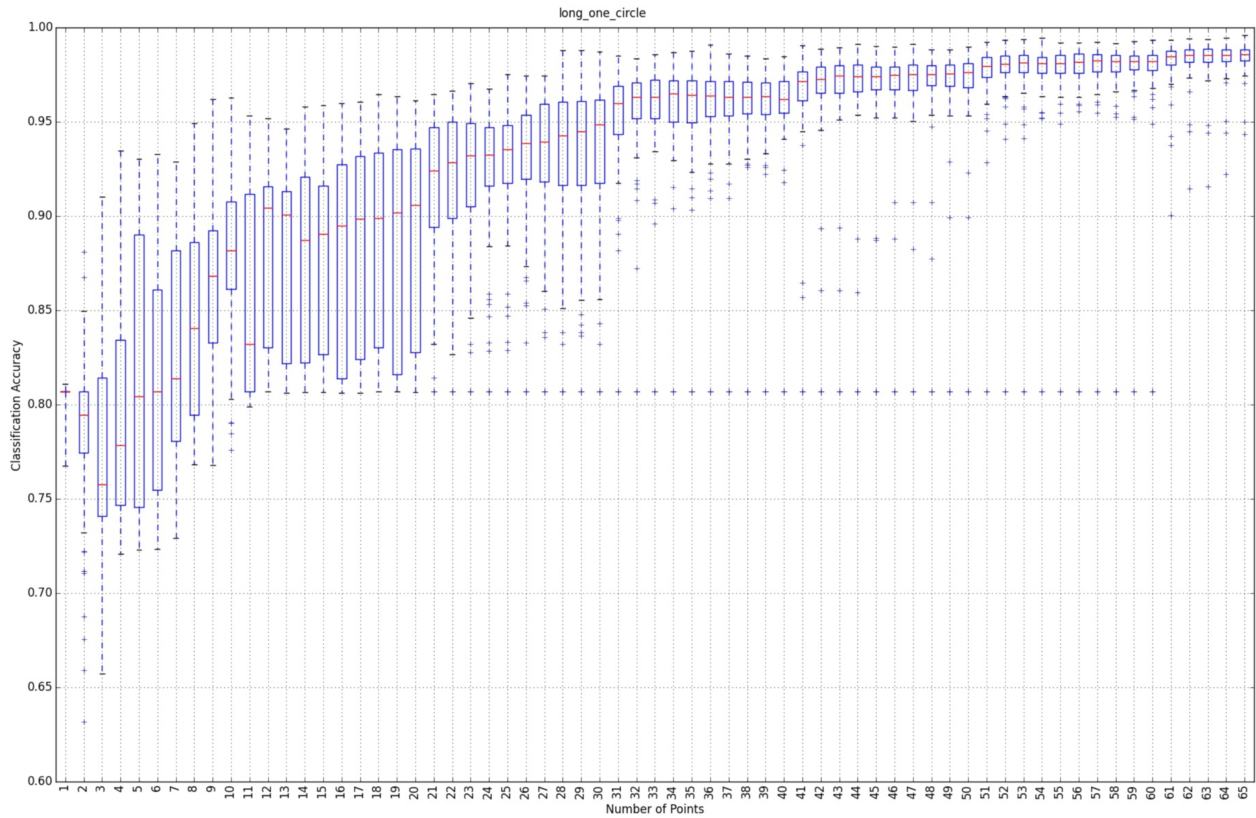


Figure 3: Box plot of accuracy of 120 experiments of a single circle model

We again see the problem of outliers with flat accuracy from the emulator over-fitting when optimization takes place when the number of points from one class far outnumbers the number of points from the other class.

To see how our emulator models the single circle refer to figure 4. We show six snap shots of our emulator (seen on the left) and the entropy of our emulator (on the right). As the number of points increases it is easy to see that our emulator does a better job of approximating the single circle. It is also shown that the overall entropy reduces and begins to focus on the decision boundary of the circle as more points are evaluated.

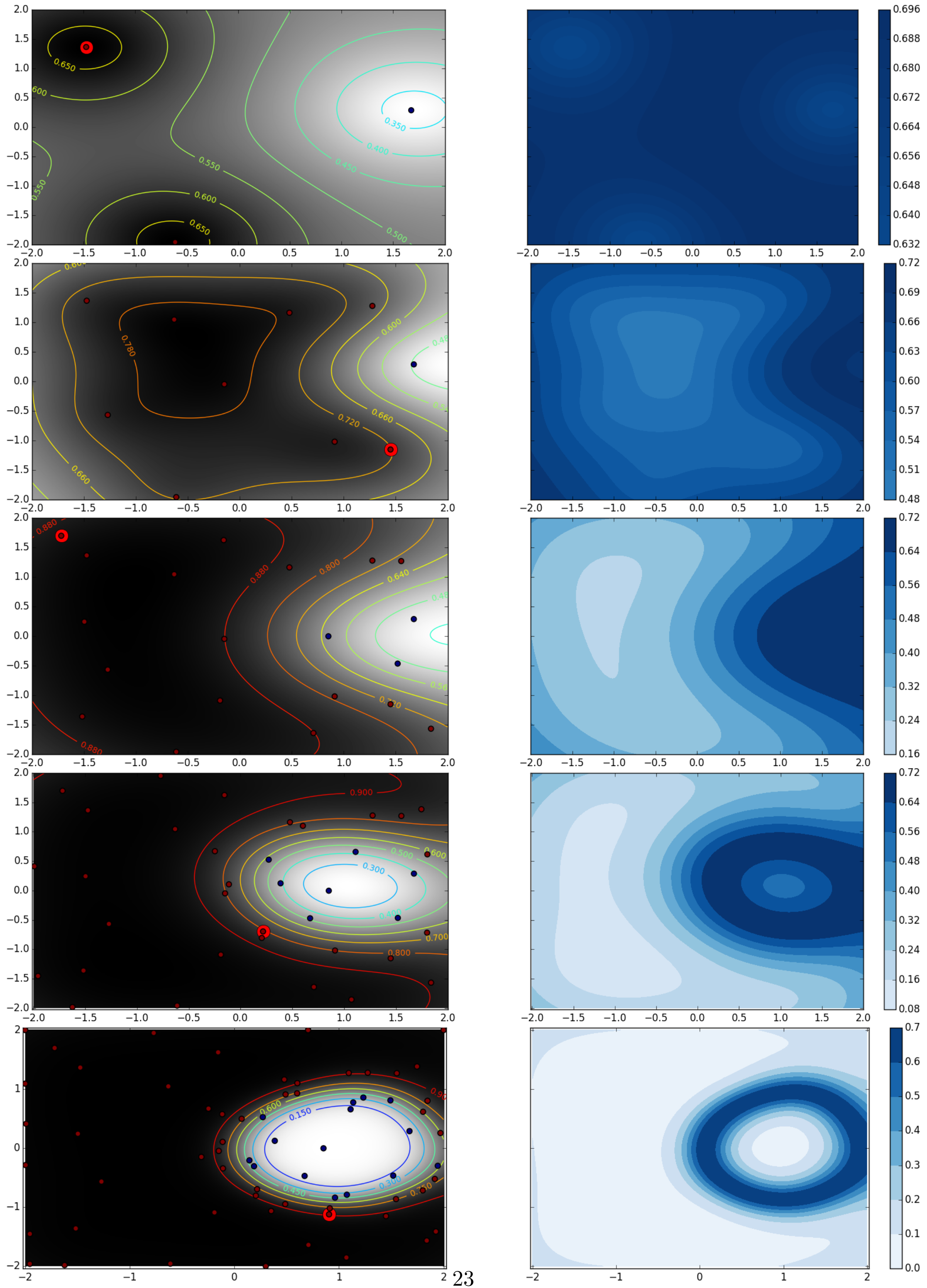


Figure 4: Example of the development of the Emulator(left) and Entropy(right) for point totals 3, 10, 20, 35 and 65 (top to bottom) with optimization performed at every tenth point.

4.1.3 Disconnected Sets of Circles

Next we experiment with two disconnected circles defined by the two dimensional formula below. The Center of the circles lies at $(-1, -1)$ and $(1, 1)$.

$$f(x_0, x_1) = \begin{cases} 1 & \text{if } \sqrt{(x_0 - 1)^2 + (x_1 - 1)^2} > 1 \\ 1 & \text{if } \sqrt{(x_0 + 1)^2 + (x_1 + 1)^2} > 1 \\ 0 & \text{Otherwise} \end{cases}$$

Where the $x_0 \in [-2, 2]$ and $x_1 \in [-2, 2]$ define the square where we limit our interrogation of our method.

We ran our method on the two circle experiment around 50 times with each iteration evaluating the experiment on 125 points. It can be seen in the box plot from figure 5 that as the number of evaluation points increases the accuracy of our emulator approaches 1 and the variance of the accuracy of all 50 experiments decreases substantially.

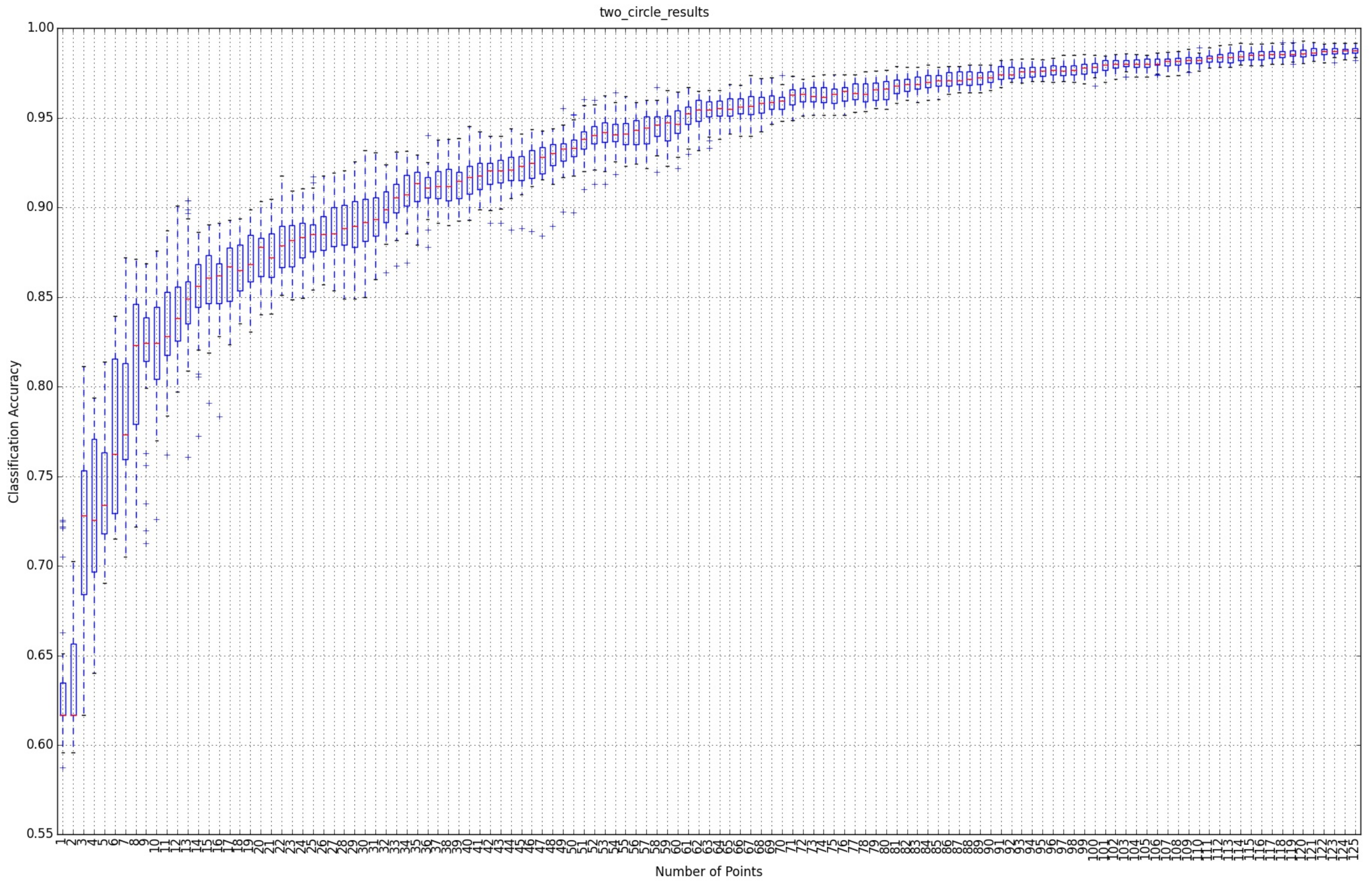


Figure 5: Box plot of accuracy of 50 disjointed circle model experiments

In this experiment we saw no outliers with flat accuracy from the emulator over-fitting. We optimized every 15 points and from observation this seems to be a good number to choose.

To see how our emulator models the two circles refer to figure 6. We show six snap shots of our emulator (seen on the left) and the entropy of our emulator (on the right). As the number of points increases it is easy to see that our emulator does a better job of approximating the two circles. It is also shown that the overall entropy reduces and begins to focus on the decision boundary of the two circle as more points are evaluated.

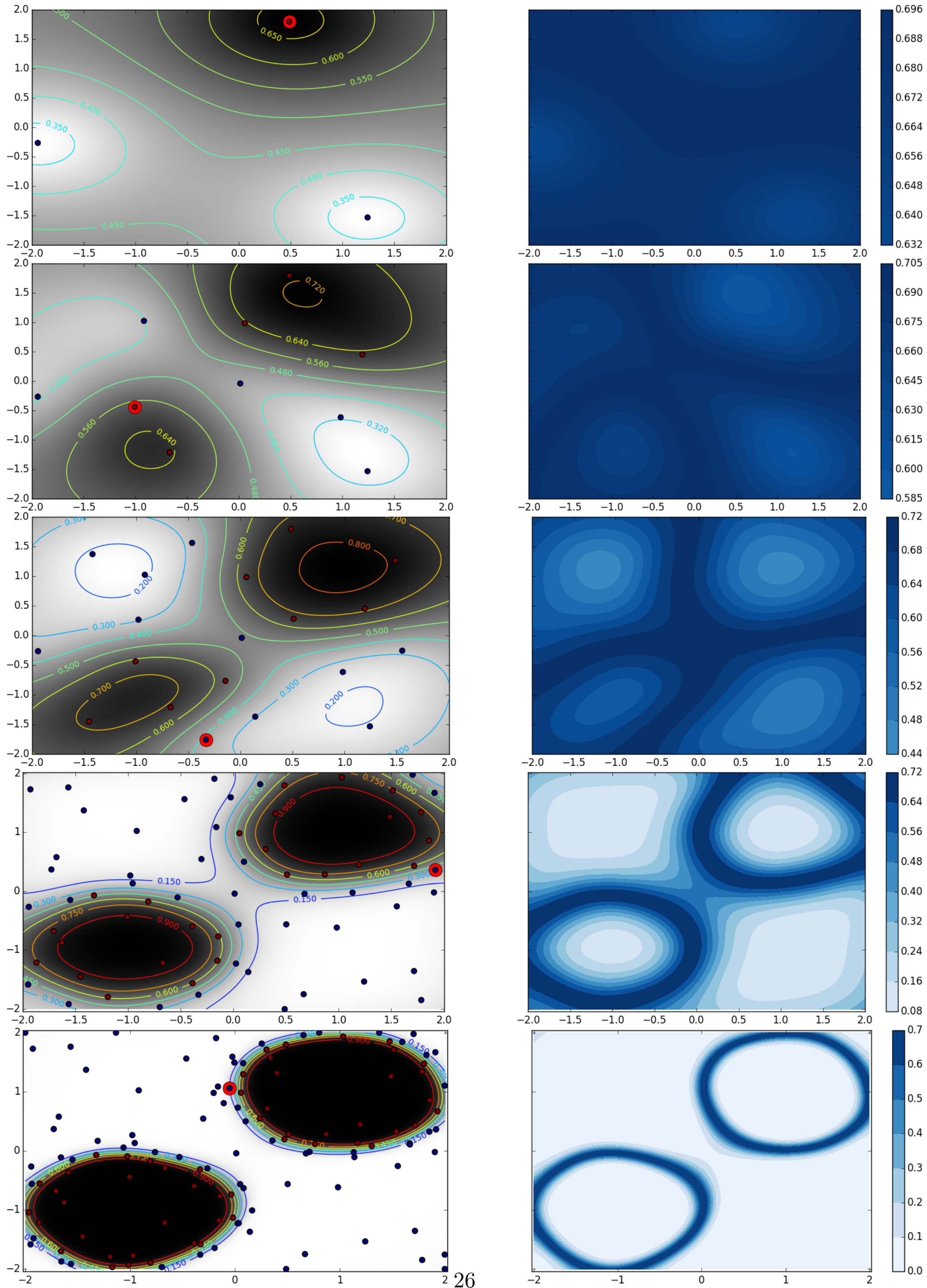


Figure 6: Example of the development of the Emulator(left) and Entropy(right) for point totals 3, 10, 20, 55 and 125 (top to bottom) with optimization performed at every fifteenth point.

4.2 Computational Fluid Dynamics

In this section we evaluate our emulator on a computational fluid dynamics simulator termed FUN3D. Our experiment on the FUN3D simulator was performed in a two dimensional setting with the dimensions of interest being the Mach number ranging from $.5 - 2.35$ and the angle of attack ranging from $0 - 30$ degrees. The FUN3D simulator results are shown in Fig. 7. The successful simulations are in blue and failed simulations are in red.

We found that optimization did not improve results thus we omitted optimization of our emulator. We were only able to run one complete experiment using the FUN3D simulator. Figure 8 shows that accuracy of our method peaks at around 83 percent and gets to that point after roughly 65 evaluations of the simulator. It took the experiment about 2000 minutes to evaluate 200 points.

We plotted our emulator and its entropy for each point evaluated by the simulator. It is much more difficult to recognize progress that is being made by our method than in the previous toy examples. Because of this we post all images with even number of points of the experiment in appendix C in the hopes that showing the images will visually aid the readers understanding of what our emulator learned.

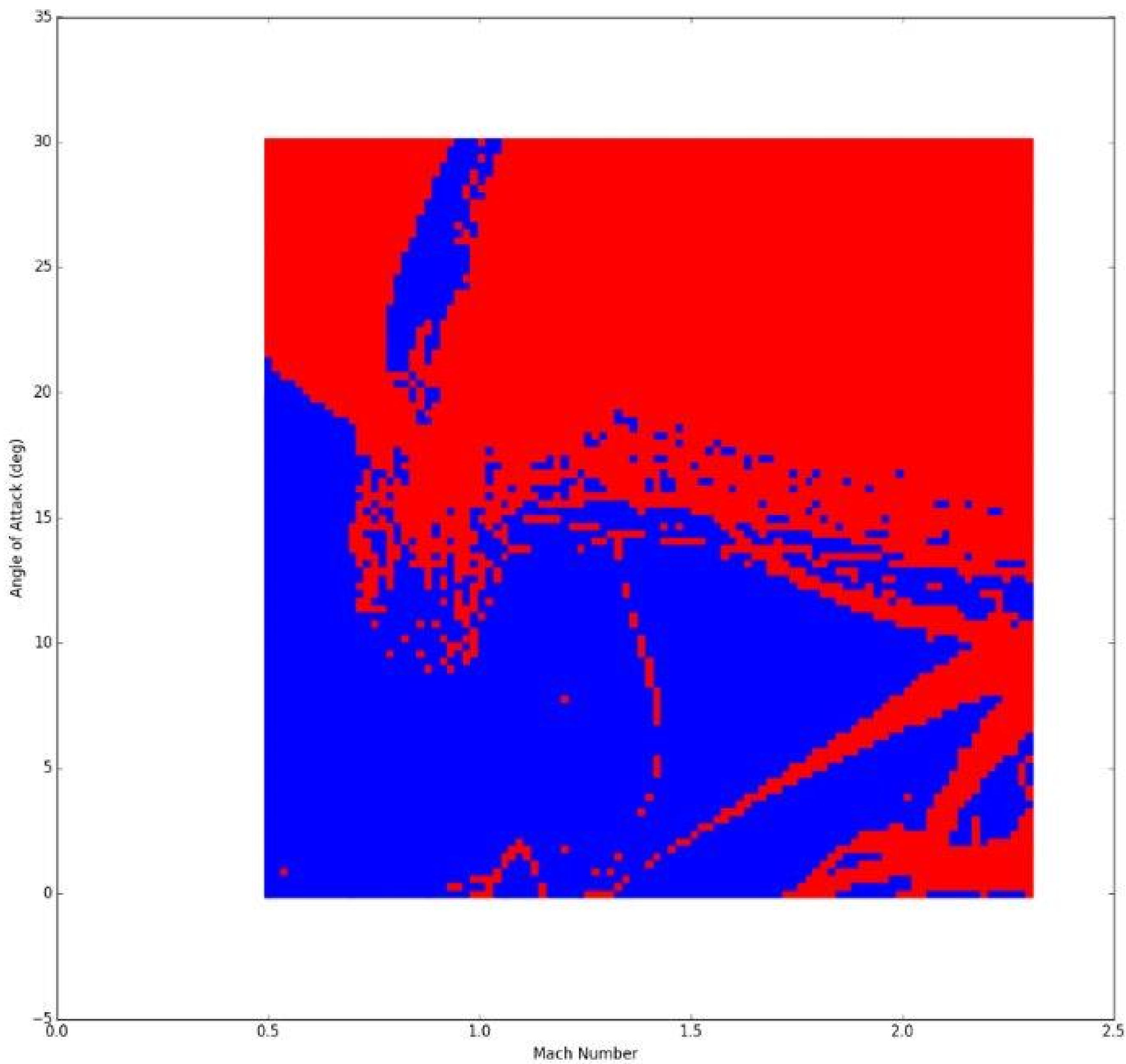


Figure 7: FUN3D Plot with CFL set to 500

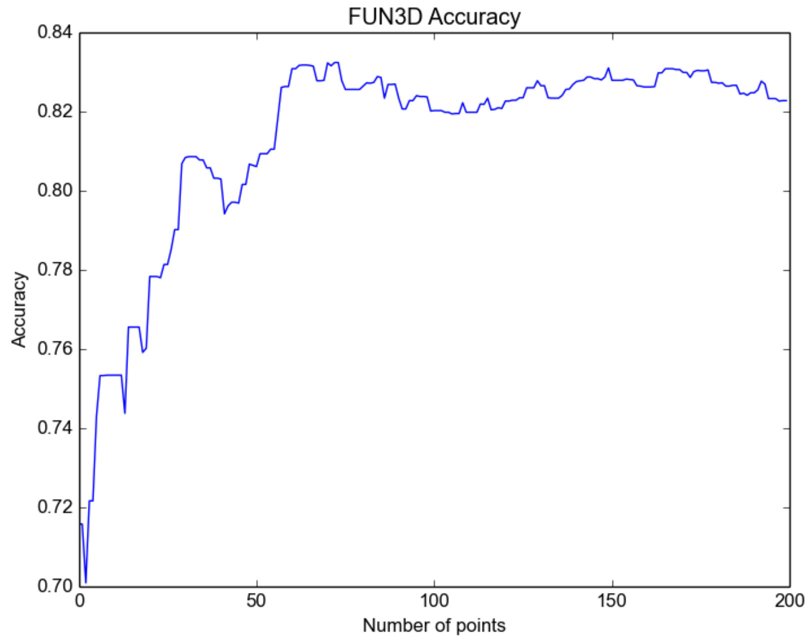


Figure 8: FUN3D Accuracy Plot per Point

5 Discussion

5.1 When to Optimize

We saw in our work that deciding when to optimize our emulator is an important question. If done right, optimization can lead to a more accurate emulator, otherwise the optimization can lead to an emulator that over-fits and performs very poorly.

Unfortunately the question of when to optimize is unique to each simulator. We chose to optimize after every three points when trying to create an emulator for the line found in 4.1.1. We increased the interval of optimization to 10 points when we tried to create an emulator for the single circle 4.1.2, and we choose to optimize every 15 points with the two disjointed circles 4.1.3. When it came to the FUN3D simulator we could not find a good interval to optimize our emulator and just ran the experiment with no optimization. A possible insight into why optimization of the FUN3D simulator showed no benefit are discussed in the next section (5.2).

5.1.1 Optimization Alternative

As an alternative to optimization, that we were not able to evaluate, we create an emulator that is composed of an ensemble of Gaussian Process Classifiers rather than a single Gaussian Process classifier. Creation of this new emulator can be seen in algorithm 4. For clarity the length scale and variance fields are hyper parameters of the Gaussian Process Classifier.

Algorithm 4 Ensemble Emulator

```
1: for length in lengthScaleList do
2:     for var in varianceList do
3:         model = New GP Classification Model
4:         model.lengthscale = length
5:         model.variance = var
6:         modelList.append(model)
   return modelList
```

The arrays defined by `lengthScaleList` as well as `varianceList` need to be well chosen. What well chosen means is again another area of future research but as a starting point we set `lengthScaleList` $= [.1, .5, 1, 1.5, 2.5, 10]$ and `varianceList` $= [.1, 1, 5]$.

With this new ensemble emulator we need to define a method to calculate the probability of a point. This method is defined in algorithm 5

Algorithm 5 Ensemble Emulator Probability

```
1: function predictProbEnsemble(emulator, nextPoint)
2:     weight = 0
3:     weights = 0
4:     probabilities = 0
5:     weightProbs = 0
6:     for model in emulator do
7:         weight += model.logLikelihood()
8:     for model in emulator do
9:         weights += model.logLikelihood()/weight
10:    for model in emulator do
11:        probabilities.append(model.predictProb(nextPoint))
12:    for probs in probabilities do
13:        for weight in weights do
14:            weightProbs += weight × probs
       return weightProbs
```

This new ensemble emulator also means that modifications to algorithm 3 must also be made. These modifications are rather obvious with algorithms 4 and 5 defined so we will omit the details to the modification of algorithm 3.

The motivation behind this method is to reduce our methods possibility of

over fitting on the data and giving incorrect predictions in regions near points that have been evaluated on the simulator. We were unable to evaluate this algorithm, but it is a good topic of future research for improvement.

5.2 Kernel Function Selection

We observed from our experiments that the choice of the Matern kernel was good for all of our toy examples. On the other hand we have an intuition that a better choice of kernel function could be used in the case of the FUN3D simulator. This intuition comes from the properties of the Matern Kernel and the behavior we see of the FUN3D simulator. To clarify, the Matern kernel assumes a stationary somewhat smooth data set. By 'somewhat smooth' we refer to the fact that some stationary kernel functions are infinity differentiable, such as the squared exponential kernel, while the Matern Kernel is only three times differentiable. This property means that the Matern kernel is good for modelling data that can be very 'wiggly' but still smooth. By 'wiggly' we mean that there are a large, non-infinite, number of points where the derivative, which is defined, changes from positive to negative or vice versa.

It can be seen in figure 7 that the FUN3D simulator is more discontinuous than the Matern Kernel can handle. This observation means that we need to investigate further if alternative kernel functions would be a better fit for creating a classification emulator that would better approximate the FUN3D simulator. Some alternative kernel properties that might be beneficial, thus worth looking into, are kernels that are non-stationary as well as kernels that are able to model highly non-smooth data. This is another good topic for future work. More detailed information on Kernel functions can be seen at chapter 4 of [12].

References

[1] M. C. Kennedy and A. O'Hagan, "Bayesian calibration of computer models," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 63, no. 3, pp. 425–464, 2001.

[2] A. O. Marc Kennedy, Clive Anderson, M. Lomas, I. Woodward, J. P. Gosling, and A. Heinemeyer, "Quantifying uncertainty in the biospheric carbon flux for england and wales," *Journal of the Royal Statistical Society. Series A (Statistics in Society)*, vol. 171, no. 1, pp. 109–135, 2008.

[3] S. Conti and A. OHagan, "Bayesian emulation of complex multi-output and dynamic computer models," *Journal of Statistical Planning and Inference*, vol. 140, no. 3, pp. 640 – 651, 2010.

[4] A. OHagan, "Bayesian analysis of computer code outputs: A tutorial," *Reliability Engineering and System Safety*, vol. 91, no. 1011, pp. 1290 – 1300, 2006. The Fourth International Conference on Sensitivity Analysis of Model Output (SAMO 2004)SAMO 2004The Fourth International Conference on Sensitivity Analysis of Model Output (SAMO 2004).

[5] C. Chevalier, J. Bect, D. Ginsbourger, E. Vazquez, V. Picheny, and Y. Richet, "Fast parallel kriging-based stepwise uncertainty reduction with application to the identification of an excursion set," *Technometrics*, vol. 56, no. 4, pp. 455–465, 2014.

[6] N. Bounceur, M. Crucifix, and R. Wilkinson, "Global sensitivity analysis of the climate–vegetation system to astronomical forcing: an emulator-based approach," *Earth System Dynamics*, vol. 6, pp. 205–224, 2015.

[7] P. B. Holden, N. R. Edwards, P. H. Garthwaite, and R. D. Wilkinson, "Emulation and interpretation of high-dimensional climate model outputs," *Journal of Applied Statistics*, pp. 1–18, 2015.

[8] P. B. Holden, N. R. Edwards, J. Hensman, and R. D. Wilkinson, "Abc for climate: dealing with expensive simulators," *arXiv preprint arXiv:1511.03475*, 2015.

[9] C. Chevalier, V. Picheny, and D. Ginsbourger, "Kriginv: An efficient and user-friendly implementation of batch-sequential inversion strategies based on kriging," *Computational Statistics and Data Analysis*, vol. 71, pp. 1021 – 1034, 2014.

[10] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of bayesian optimization," tech. rep., Universities of Harvard, Oxford, Toronto, and Google DeepMind, 2015.

[11] T. G. authors, "Gpyopt: A bayesian optimization framework in python." urlhttp://github.com/SheffieldML/GPyOpt, 2015.

[12] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[13] J. E. Oakley and A. O'Hagan, "Probabilistic sensitivity analysis of complex models: a bayesian approach," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 66, no. 3, pp. 751–769, 2004.

[14] T. J. Santner, W. B., and N. W., *The Design and Analysis of Computer Experiments*. Springer-Verlag, 2003.

[15] M. Kennedy, A. OHagan, and N. Higgins, "Bayesian analysis of computer code outputs," in *Quantitative Methods for Current Environmental Issues* (C. Anderson, V. Barnett, P. Chatwin, and A. El-Shaarawi, eds.), pp. 227–243, Springer London, 2002.

[16] E. Snelson and Z. Ghahramani, "Sparse gaussian processes using pseudo-inputs," in *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, pp. 1257–1264, MIT press, 2006.

[17] J. Hensman, N. Fusi, and N. D. Lawrence, "Gaussian processes for big data," in *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013, Bellevue, WA, USA, August 11-15, 2013*, 2013.

[18] Z. Dai, A. C. Damianou, J. Hensman, and N. D. Lawrence, "Gaussian process models with parallelization and GPU acceleration," *CoRR*, vol. abs/1410.4984, 2014.

[19] N. D. Lawrence, "Gaussian process latent variable models for visualisation of high dimensional data," in *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pp. 329–336, 2003.

[20] C. Paciorek and M. Schervish, "Nonstationary covariance functions for gaussian process regression," *Advances in neural information processing systems*, vol. 16, pp. 273–280, 2004.

[21] The GPy authors, "GPy: A gaussian process framework in python." url-http://github.com/SheffieldML/GPy, 2012–2015.

[22] J. Bect, D. Ginsbourger, L. Li, V. Picheny, and E. Vazquez, "Sequential design of computer experiments for the estimation of a probability of failure," *Statistics and Computing*, vol. 22, no. 3, pp. 773–793, 2012.

[23] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *CoRR*, vol. abs/1012.2599, 2010.

A Code for Toy Experiments

This code can run all toy experiments listed in the paper. The user only needs to modify the function $f(X)$ to change which experiment is evaluated. The code to run the FUN3D experiment is very similar to this. The only difference is, again, the form of the function $f(X)$. To run this code you must use the python Packages GPy, GPyOpt, Numpy, SciPy and PyDOE. All packages can be found on GitHub.

```
# coding: utf-8

# #Use the EntABC class to do entropic ABC

# In[5]:

import numpy as np
import time
import GPy
from multiprocessing import Pool
#from entropicABC import EntABC
#get_ipython().magic(u'matplotlib inline')
from matplotlib import pyplot as plt
import matplotlib
matplotlib.use('Agg')

matplotlib.rcParams['figure.figsize'] = (18, 5)
# np.random.seed(55) # set the seed
import numpy as np
import GPy
import sys
sys.path.append('~/SheffieldML')
import GPyOpt
import sys
sys.path.append('/xlogx/npufunc_directory/')
import npufunc as util
import copy
#from matplotlib import pyplot as plt
import json


class EntABC(object):

    def __init__(self, bounds, model,
                 data_lower=-np.inf, data_upper=.5,
                 gridtype='fixed', gridres=100,
                 GH_points=10):
        self.bounds = bounds
```

```
        self.model = model
        assert not np.all(np.isinf(
            [data_lower, data_upper])),                "you must specify
                at least one-sided ovbervation bound!"
        self.data_lower, self.data_upper = data_lower, .5

        # GH setup
        #self.gh_x, self.gh_w = np.polynomial.hermite.hermgauss(
            GH_points)#

        # grid setup
        self.gridtype, self.gridres = gridtype, gridres
        if gridtype is 'fixed':
            self.Xgrid = GPyOpt.util.general.multigrid(
                self.bounds, self.gridres)

    def select_next_point(self, max_iter=150, **options):
        """
        compute the optimal next point using Bayesian Optimization.

        maxiter is the number of BO iterations that we'll use.

        options are keyword arguments that will be passed to the
            Bayesian
        Optimizer. By default we use:

            acquisition='EI',
            acquisition_par=0.001,
            normalize=True,
            model_optimize_restarts=1,
            model_optimize_interval=30,
            verbosity=0

        See GPyOpt.BayesianOptimization for a full list.

        Returns
        --
        Xnew, the next point to sample
        m, the GPyOpt model which did the optimizing
        """
        default_options = dict(acquisition='EI',
                               acquisition_par=0.001,
                               normalize=True,
                               model_optimize_restarts=1,
                               model_optimize_interval=30,
```

```
                                        verbosity=0)
    # set the default options in the dictionary only if they're not
        set.
    for k, a in default_options.iteritems():
        if not k in options:
            options[k] = a

    m = GPyOpt.methods.BayesianOptimization(
        self.objective, self.bounds, **options)
    m.model.likelihood.variance.fix(1e-6)
    m.run_optimization(max_iter=max_iter)
    i = np.argmin(m.Y)
    Xnew = m.X[i]

    return Xnew, m

def select_next_point_grid(self, search_res=5):
    """
    Select the next point by brute-force searching on a grid.
    """

    search_grid = GPyOpt.util.general.multigrid(self.bounds,
       search_res)
    print search_grid.shape
    E_entropies = self.objective(search_grid)
   # print E_entropies
    print 'min entropies'
    print np.min(E_entropies), np.max(E_entropies), E_entropies.
       shape
    i = np.argmin(E_entropies)
    print 'grid info'
    print i, search_grid[i], search_grid[i].shape
    return search_grid[i]

def objective(self, Xtrial):
    """
    The objective function for selecting the next point at the
       point X
    """

    # handle all points one at a time. For multiple points of
       Xtrial,
    # recurse.
    if len(Xtrial.shape) == 1:
        Xtrial = Xtrial.reshape(1, -1)
```

```
        if Xtrial.shape[0] > 1:
            return np.array([self.objective(x) for x in Xtrial]).
               reshape(-1, 1)

        # work out if we need a dynamic grid
        if self.gridtype is 'dynamic':
            Xgrid = self.dynamic_grid(Xtrial)
        else:
            Xgrid = self.Xgrid

        return self.E_AveH_classification(Xtrial, Xgrid)

    def dynamic_grid(self, X):
        """
        Compute a grid which is two lengthscales around X, but still
           within self.bounds
        """
        X = X.squeeze
        assert len(X.shape) == 1, "dynamic grids are around one point
           only"
        lower = np.fmin([d[0] for d in self.bounds],
                        X - self.model.kern.lengthscale)
        upper = np.fmax([d[1] for d in self.bounds],
                        X + self.model.kern.lengthscale)
        return GPyOpt.util.general.multigrid(zip(lower, upper), self.
           gridres)

    def aveH_classification(self, Xtrial, Xgrid, label):
        Xnext = np.vstack((self.model.X, Xtrial))
        Y_label = np.vstack((self.model.Y, label))
       # model_copy = GPy.models.GPClassification(Xnext, Y_label, kernel
          =GPy.kern.Matern52(2, 1, 1.))
        model_copy = copy.deepcopy(self.model)
        worked = True
        try:
            model_copy.set_XY(Xnext, Y_label)
        except:
            print 'we had an error somewhere!!!!!!!!!!!!'
            worked = False
        p_label, _ = np.asarray(model_copy.predict(Xgrid))
        del model_copy
        return util.entropy(p_label).mean()

    def E_AveH_classification(self, Xtrial, Xgrid):
        if len(Xtrial.shape) == 1:
```

```
            Xtrial = Xtrial.reshape(1, -1)
        if Xtrial.shape[0] > 1:
            return np.array([self.E_AveH_classification(x, Xgrid) for x
                in Xtrial])
      #  pool = Pool(processes=2)
      #  global Xtrail
      # 3  global Xgrid
      #  global self.model
        px, _ = np.asarray(self.model.predict(Xtrial))

#  results = pool.map(aveH_classification, [(self, Xtrial, 0), (self,
    Xtrial, 1)])

#        pool.close()
    #  ttest, _ = np.asarray(self.model.predict(self.model.X))
      #  Xnext = np.vstack((self.model.X, Xtrial))
      #  Y_zero = np.vstack((self.model.Y, np.array([0])))
      #  Y_one = np.vstack((self.model.Y, np.array([1])))
      #  print self.model.X
      #  model_zero = copy.deepcopy(self.model)
      #  model_zero.set_XY(Xnext, Y_zero)

      #  model_one = copy.deepcopy(self.model)
      #  model_one.set_XY(Xnext, Y_one)
        #ttest , _ = np.asarray(model_zero.predict(self.model.X))
      #  print ttest
      #  p_zero, _=  np.asarray(model_zero.predict(Xgrid))
      #  p_one, _ = np.asarray(model_one.predict(Xgrid))
        # print type(model_one)
        # print 'mins and max'
        # print min(p_zero), max(p_zero), min(p_one), max(p_one)
       # AveH_zero = util.entropy(p_zero).mean()
       # AveH_one = util.entropy(p_one).mean()
      #  print type(AveH_zero), type(px)
      #  del model_one
      #  del model_zero

       # print AveH_one, AveH_zero
       # print px*AveH_one + (1-px)*AveH_zero
        return px * self.aveH_classification(Xtrial, Xgrid, np.array
           ([1])) + (1 - px) * self.aveH_classification(Xtrial, Xgrid,
           np.array([0]))

    def plot(self, iteration):
        if len(self.bounds) == 1:
```

```
            self.plot_1d
        elif len(self.bounds) == 2:
            self.plot_2d(iteration)
        else:
            raise NotImplementedError, "what should we plot in high
               dimensions?"

    def plot_1d(self):
        pass

    def plot_2d(self, iteration, show_objective=False, resolution=1000,
        show_variance=False, highlight_latest=True, show_entropy=True):

        num_plots = 1
        if show_objective:
            num_plots += 1
        if show_variance:
            num_plots += 1
        if show_entropy:
            num_plots += 1
        fig, ax = plt.subplots(1, num_plots, sharex=True,
                               sharey=True, figsize=(4 * 5, 5))
        ax = np.atleast_1d(ax)
        assert ax.size == num_plots

        Xplot = GPyOpt.util.general.multigrid(self.bounds, resolution)
        xx, yy = [x.reshape(resolution, resolution) for x in Xplot.T]

        p, _ = self.model.predict(Xplot)
        mu, var = self.model._raw_predict(Xplot)
        std = np.sqrt(np.clip(var, 0, np.inf))
        vmin, vmax = self.model.Y.min(), self.model.Y.max()
       # print 'xx yy'
       # print xx.shape, yy.shape
        CS = ax[0].contour(xx, yy, p.reshape(*xx.shape),
                           vmin=vmin, vmax=vmax, cmap=plt.cm.jet)
        plt.clabel(CS, inline=1, fontsize=10)
        if highlight_latest:
            ax[0].plot(self.model.X[-1, 0], self.model.X[-1, 1], 'ro',
               ms=16)
        ax[0].scatter(self.model.X[:, 0], self.model.X[:, 1], 40, self.
           model.Y[:, 0],
                      linewidth=1.2, vmin=vmin, vmax=vmax, cmap=plt.cm.
                         jet, zorder=10)
        # p = util.normcdf((mu)/np.sqrt(1+np.clip(var,0,np.inf)))# -
```

```
        # util.normcdf((self.data_lower-mu)/std)
        H = util.entropy(p)
        # print 'probability'

        # print np.min(p), np.max(p)
       # print 'Entropy'
       # print np.min(H), np.max(H)
        ax[0].imshow(1 - p.reshape(*xx.shape).T, cmap=plt.cm.gray,
           interpolation='bilinear',
                    origin='lower', extent=np.hstack(self.bounds),
                       aspect='auto')

        ax_count = 1
        # if show_variance:
        #     ax[ax_count].contourf(xx, yy, std.reshape(*xx.shape), cmap
           =plt.cm.Blues)
        #     ax_count += 1
        if show_entropy:
            foo = ax[ax_count].contourf(xx, yy, H.reshape(
                *xx.shape), cmap=plt.cm.Blues, vmin=0, vmax=util.
                   entropy(0.5))
            plt.colorbar(foo)
            ax_count += 1
        if show_objective:
            foo = ax[ax_count].contourf(xx, yy, self.objective(
                Xplot).reshape(resolution, resolution), cmap=plt.cm.jet
                   )
            plt.colorbar(foo)
        fig.savefig('/scratch/two_circle_image' + str(iteration) + '.
           png')
        return fig, ax


def aveH_classification(args):
    Xnext = np.vstack((args[0].X, args[1]))
    Y_label = np.vstack((args[0].Y, args[3]))

    model_copy = copy.deepcopy(args[0])
    model_copy.set_XY(Xnext, Y_label)
    p_label, _ = np.asarray(model_copy.predict(args[2]))
    del model_copy
    return util.entropy(p_label).mean()


def calculate_accuracy(model, bound, resolution):
```

```
    Xplot = GPyOpt.util.general.multigrid(bound, resolution)
    model_output, _ = model.predict(Xplot)
    # print type(model_output)
    # print len(model_output)

    model_truth = np.array(
        ([1 if i > .5 else 0 for i in model_output])).reshape(-1, 1)
    f_truth = f(Xplot)
   # print len(f_truth)
   # print f_truth.shape, model_truth.shape
   # model
   # print np.sum(model_truth == f_truth)
    # for i in model_truth: print i
    return float(np.sum(model_truth == f_truth)) / f_truth.size


def f(X):
    if len(X.shape) == 1:
        X = X.reshape(1, -1)  # print X
   # return (mTrue.predict(X)[0])
#     print np.array(np.sign([-1*j[0]+3*j[1]-2 for j in Xpoints])).
   reshape(-1,1).size
    # return np.array(([1 if -1*j[0]+3*j[1]-2 >= 0 else 0 for j in
    # X])).reshape(-1,1)
    ypoints = []
    for i in X:
        if (i[0] - 1)**2 + (i[1] - 1)**2 <= 1:
            ypoints.append(1)
        elif (i[0] + 1)**2 + (i[1] + 1)**2 <= 1:
            ypoints.append(1)
        else:
            ypoints.append(0)
            # print ypoints'''
    return np.array(ypoints).reshape(-1, 1)
 # return np.array( [ 1 if np.sqrt((j[0]-1)**2+j[1]**2)>1 else 0 for j
    in
 # X]).reshape(-1,1)


# In[6]:
def first_step(main_iteration):
    # define the problem:bounds, data limits, function and grid for
        evaluating
    # entropy.
    x = 2
```

```
bounds = [[-x, x], [-x, x]]
data_lower, data_upper = .5, .5

# draw fromt a GP for ground truth.
# xx, yy = np.mgrid[bounds[0][0]:bounds[0][1]:10j, bounds[1][0]:
    bounds[1][1]:10j]
# print xx.flatten()
# print yy.flatten()
# Xpoints = np.vstack((xx.flatten(), yy.flatten())).T
# print Xpoints
# K=GPy.kern.RBF(input_dim=2, variance=10, lengthscale=2).K(Xpoints
    )
#ypoints = np.array([1 if -1*j[0]+3*j[1]-2 >=0 else 0 for j in
    Xpoints]).reshape(-1,1)
#ypoints = np.array( [ 1 if np.sqrt((j[0]-1)**2+j[1]**2)>2 else 0
    for j in Xpoints]).reshape(-1,1)
 '''ypoints = []
  for i in Xpoints:
        if (i[0]-2)**2+(i[1]-2)**2 <= 4:
              ypoints.append(1)
        elif (i[0]+2)**2+(i[1]+2)**2 <= 4:
              ypoints.append(1)
        else:
              ypoints.append(0)
      # print ypoints'''

#       def f(X):
#           if len(X.shape)==1:
#                X = X.reshape(1,-1)          #     print X
    # return (mTrue.predict(X)[0])
    #      print np.array(np.sign([-1*j[0]+3*j[1]-2 for j in Xpoints])).
        reshape(-1,1).size
#           return np.array(([1 if -1*j[0]+3*j[1]-2 >= 0 else 0 for j in X
  ])).reshape(-1,1)
    # return np.array( [ 1 if np.sqrt((j[0]-1)**2+j[1]**2)>2 else 0 for
        j in
    # X]).reshape(-1,1)

    #xx, yy = np.mgrid[bounds[0][0]:bounds[0][1]:50j, bounds[1][0]:
        bounds[1][1]:50j]
    #Xgrid = np.vstack((xx.flatten(), yy.flatten())).T

    # In[7]:

    # first few evaluations at random points and construction of GP
        model
```

```
from pyDOE import lhs
initial_points = 3
X = lhs(2, initial_points, criterion="maximin", iterations=1000)
X = X * np.array([bounds[0][1] - bounds[0][0], bounds[1][1] -
                  bounds[1][0]]) + np.array([bounds[0][0], bounds
                     [1][0]])
# print X
Y = f(X)
#print (Y.shape == Y.shape)
# print Y
m = GPy.models.GPClassification(X, Y, kernel=GPy.kern.Matern52(2,
   1, 1.))
# m.Gaussian_noise.fix(1e-8)
# m.plot()
total_accuracy = [[]]

# In[ ]:

# In[8]:

start = time.time()
iteration = 125
accuracy_list = []
for it in range(iteration):
    start_loop = time.time()
    # optimize now and then.
    if it % 15 == 0 and it != 0:  # and it >1:
        # if it == 1:
        #     pass
        # else:
        #     pass
        m.optimize()

    # here's the EntABC instance
    mABC = EntABC(bounds, m, data_lower, data_upper)

    # choose the best point seen by the Optimization model
    Xnew, mBO = mABC.select_next_point()

    # plot
    mABC.plot(it)
    # mBO.plot_convergence()

    # actually set the intended observa
    Ynew = f(Xnew)
```

```
        # print type(Ynew), type(Xnew)
        setX = np.vstack((m.X, Xnew))
        setY = np.vstack((m.Y, Ynew))

        dummy = calculate_accuracy(m, bounds, 100)
        accuracy_list.append(dummy)
        print 'Accuracy is ' + str(dummy)
        m.set_XY(setX, setY)

        print 'Iteration Number ' + str(it + 1)
        print 'Iteration Time was ' + str(((time.time() - start_loop) /
            60.0)) + ' minutes'
        # print calculate_accuracy()
        try:

            with open("single_two_circle_results_" + str(main_iteration
                ) + ".txt", "w") as myfile:
                json.dump(accuracy_list, myfile)
        except:
            print "some where io error"

    print 'Final Iteration'
    end = time.time()

    print 'Total time was ' + str((end - start) / 60.0) + ' Minutes'
    return accuracy_list


def dummy_function():

    # In[ ]:

    # plot the truth
    fig, ax = plt.subplots(1, 3, sharex=True, sharey=True, figsize=(18,
        6))
    ff = f(Xgrid).reshape(*xx.shape)
    CS = ax[0].contour(xx, yy, ff.reshape(
        *xx.shape), np.arange(-6, 6, 2), vmin=ff.min(), vmax=ff.max(),
            cmap=plt.cm.jet)
    plt.clabel(CS, inline=1, fontsize=10)
    ptrue = (ff < data_upper) * 1.
    ax[0].imshow(1 - ptrue.reshape(*xx.shape).T, cmap=plt.cm.gray,
                 interpolation='nearest', origin='lower', extent=np.
                    hstack(bounds))
    ax[0].set_title('ground truth')
```

```
# space filler
from pyDOE import lhs
Xfill = lhs(2, samples=m.Y.size, criterion="maximin", iterations
   =1000)
Xfill = Xfill * np.array([bounds[0][1] - bounds[0][0], bounds[1]
                          [1] - bounds[1][0]]) + np.array([bounds
                             [0][0], bounds[1][0]])
Yfill = f(Xfill)
mfill = GPy.models.GPRegression(
    Xfill, Yfill, kernel=GPy.kern.Matern52(2, 1, 1.))
mfill.Gaussian_noise.fix(1e-6)
mfill.optimize_restarts(3)
mfill_mu, mfill_var = mfill.predict(Xgrid)
CS = ax[1].contour(xx, yy, mfill_mu.reshape(
    *xx.shape), np.arange(-6, 6, 2), vmin=ff.min(), vmax=ff.max(),
       cmap=plt.cm.jet)
plt.clabel(CS, inline=1, fontsize=10)

from scipy import stats

def probabilities(mu, var):
    # - stats.norm.cdf(-(mu - data_lower)/np.sqrt(var))
    return stats.norm.cdf((mu) / np.sqrt(var))
ax[1].imshow(1 - probabilities(mfill_mu, mfill_var).reshape(*xx.
   shape).T,
             cmap=plt.cm.gray, interpolation='bilinear', origin='
                lower', extent=np.hstack(bounds))
ax[1].plot(Xfill[:, 0], Xfill[:, 1], 'ro')
ax[1].set_title('space filling')

m_ent_mu, m_ent_var = m._raw_predict(Xgrid)
ax[2].imshow(1 - probabilities(m_ent_mu, m_ent_var).reshape(*xx.
   shape).T,
             cmap=plt.cm.gray, interpolation='bilinear', origin='
                lower', extent=np.hstack(bounds))
ax[2].plot(m.X[:, 0], m.X[:, 1], 'ro')
CS = ax[2].contour(xx, yy, m_ent_mu.reshape(
    *xx.shape), np.arange(-6, 6, 2), vmin=ff.min(), vmax=ff.max(),
       cmap=plt.cm.jet)
plt.clabel(CS, inline=1, fontsize=10)
ax[2].set_title('Entropic search')
ax[2].set_ylim(yy.min(), yy.max())

ptrue = ptrue.flatten()
```

```
    # get the probabilities for the space-filling design
    mu, var = mfill._raw_predict(Xgrid)
    pfill = probabilities(mu, var).flatten()
    pfill = np.where(ptrue == 1, pfill, 1 - pfill)
    print 'fill:', np.mean(np.log(pfill)),  np.mean(ptrue == (pfill >
       0.5))

    # get the probabilities for the entropy search
    p, _ = m.predict(Xgrid)
    #p = probabilities(mu, var).flatten()
    p = np.where(ptrue == 1, p, 1 - p)
    print 'ES:', np.mean(np.log(p)), np.mean(ptrue == (p > 0.5))

    # In[ ]:

    x_range = range(initial_points, iteration + initial_points)

    mfill

    # In[ ]:

    m.plot()
    plt.figure(0)
    plt.plot(x_range, accuracy_list)
    plt.ylabel('Accuracy %')
    plt.xlabel('Total Number of Evaluations')
    plt.plot()

    # In[ ]:

    m.pickle('example_model.pickle')

    # In[ ]:

    print m.Y, m.Y.size

    # In[ ]:

    plt.plot(np.random.randn(10))

    # In[ ]:


def main():
    results = []
```

```
    try:
        accuracy = first_step(1)
        results.append(accuracy)
    #     print "Epoch number "+str(i+1)+" out of 25"
    except Exception as e:
        print str(e) + 'we are going to try again'
    # with open('long_circle_accuracy.txt', 'w') as myfile:
    #     json.dump(results, myfile)

# In[ ]:


if __name__ == "__main__":
    main()
```

B Code for FUN3D Experiment

```
# coding: utf-8

# #Use the EntABC class to do entropic ABC

# In[5]:
import codecs
import subprocess
import numpy as np
import time
import GPy
from multiprocessing import Pool
#from entropicABC import EntABC
#get_ipython().magic(u'matplotlib inline')
from matplotlib import pyplot as plt
import matplotlib
matplotlib.use('Agg')

matplotlib.rcParams['figure.figsize']=(18,5)
#np.random.seed(55) # set the seed
import numpy as np
import GPy
import sys; sys.path.append('~/SheffieldML')
import GPyOpt
import sys
sys.path.append('/xlogx/npufunc_directory/')
import npufunc as util
import copy
#from matplotlib import pyplot as plt
import json
class EntABC(object):
    def __init__(self, bounds, model,
                 data_lower=-np.inf, data_upper=.5,
                 gridtype='fixed', gridres=100,
                 GH_points=10):
        self.bounds = bounds
        self.model = model
        assert not np.all(np.isinf([data_lower, data_upper])),
                        "you must specify at least one-sided
           ovbervation bound!"
        self.data_lower, self.data_upper = data_lower, .5

        #GH setup
```

```
    #self.gh_x , self.gh_w = np.polynomial.hermite.hermgauss(
        GH_points)#

    #grid setup
    self.gridtype, self.gridres = gridtype, gridres
    if gridtype is 'fixed':
        self.Xgrid = GPyOpt.util.general.multigrid(self.bounds,
            self.gridres)

def select_next_point(self, max_iter=150, **options):
    """
    compute the optimal next point using Bayesian Optimization.

    maxiter is the number of BO iterations that we'll use.

    options are keyword arguments that will be passed to the
        Bayesian
    Optimizer. By default we use:

        acquisition='EI',
        acquisition_par=0.001,
        normalize=True,
        model_optimize_restarts=1,
        model_optimize_interval=30,
        verbosity=0

    See GPyOpt.BayesianOptimization for a full list.

    Returns
    --
    Xnew, the next point to sample
    m, the GPyOpt model which did the optimizing
    """
    default_options = dict(acquisition='EI',
                           acquisition_par=0.001,
                           normalize=True,
                           model_optimize_restarts=1,
                           model_optimize_interval=30,
                           verbosity=0)
    #set the default options in the dictionary only if they're not
        set.
    for k,a in default_options.iteritems():
        if not k in options:
            options[k] = a
```

```
        m = GPyOpt.methods.BayesianOptimization(self.objective, self.
            bounds,**options)
        m.model.likelihood.variance.fix(1e-6)
        m.run_optimization(max_iter=max_iter)
        i = np.argmin(m.Y)
        Xnew = m.X[i]

        return Xnew, m

    def select_next_point_grid(self, search_res=5):
        """
        Select the next point by brute-force searching on a grid.
        """

        search_grid = GPyOpt.util.general.multigrid(self.bounds,
            search_res)
        print search_grid.shape
        E_entropies = self.objective(search_grid)
       # print E_entropies
        print 'min entropies'
        print np.min(E_entropies), np.max(E_entropies), E_entropies.
            shape
        i = np.argmin(E_entropies)
        print 'grid info'
        print i, search_grid[i], search_grid[i].shape
        return search_grid[i]


    def objective(self, Xtrial):
        """
        The objective function for selecting the next point at the
            point X
        """

        #handle all points one at a time. For multiple points of Xtrial
            , recurse.
        if len(Xtrial.shape)==1:
            Xtrial = Xtrial.reshape(1,-1)
        if Xtrial.shape[0]>1:
            return np.array([self.objective(x) for x in Xtrial]).
                reshape(-1,1)

        #work out if we need a dynamic grid
        if self.gridtype is 'dynamic':
            Xgrid = self.dynamic_grid(Xtrial)
```

```
    else:
        Xgrid = self.Xgrid

    return self.E_AveH_classification(Xtrial, Xgrid)

def dynamic_grid(self, X):
    """
    Compute a grid which is two lengthscales around X, but still
       within self.bounds
    """
    X = X.squeeze
    assert len(X.shape)==1, "dynamic grids are around one point
       only"
    lower = np.fmin([d[0] for d in self.bounds], X - self.model.
       kern.lengthscale)
    upper = np.fmax([d[1] for d in self.bounds], X + self.model.
       kern.lengthscale)
    return GPyOpt.util.general.multigrid(zip(lower, upper), self.
       gridres)
def aveH_classification(self, Xtrial, Xgrid, label):
    Xnext = np.vstack((self.model.X, Xtrial))
    Y_label = np.vstack((self.model.Y, label))
   # model_copy = GPy.models.GPClassification(Xnext, Y_label, kernel
      =GPy.kern.Matern52(2, 1, 1.))
    model_copy = copy.deepcopy(self.model)
    worked = True
    try:
         model_copy.set_XY(Xnext, Y_label)
    except:
         print 'we had an error somewhere!!!!!!!!!!!!'
         worked = False
    p_label, _=  np.asarray(model_copy.predict(Xgrid))
    del model_copy
    return util.entropy(p_label).mean()

def E_AveH_classification(self, Xtrial, Xgrid):
    if len(Xtrial.shape)==1:
        Xtrial = Xtrial.reshape(1,-1)
    if Xtrial.shape[0]>1:
        return np.array([self.E_AveH_classification(x, Xgrid) for x
            in Xtrial])
  #  pool = Pool(processes=2)
  #  global Xtrail
  #3  global Xgrid
  #  global self.model
```

```
        px, _ = np.asarray(self.model.predict(Xtrial))

#   results = pool.map(aveH_classification, [(self, Xtrial, 0), (self,
    Xtrial, 1)])

#         pool.close()
    #   ttest, _ = np.asarray(self.model.predict(self.model.X))
      #   Xnext = np.vstack((self.model.X, Xtrial))
      #   Y_zero = np.vstack((self.model.Y, np.array([0])))
      #   Y_one = np.vstack((self.model.Y, np.array([1])))
      #   print self.model.X
      #   model_zero = copy.deepcopy(self.model)
      #   model_zero.set_XY(Xnext, Y_zero)

      #   model_one = copy.deepcopy(self.model)
      #   model_one.set_XY(Xnext, Y_one)
        #ttest , _ = np.asarray(model_zero.predict(self.model.X))
      #   print ttest
      #   p_zero, _= np.asarray(model_zero.predict(Xgrid))
      #   p_one, _ = np.asarray(model_one.predict(Xgrid))
        #print type(model_one)
        #print 'mins and max'
        #print min(p_zero), max(p_zero), min(p_one), max(p_one)
       # AveH_zero = util.entropy(p_zero).mean()
       # AveH_one = util.entropy(p_one).mean()
      #   print type(AveH_zero), type(px)
      #   del model_one
      #   del model_zero

       # print AveH_one, AveH_zero
       # print px*AveH_one + (1-px)*AveH_zero
        return px*self.aveH_classification(Xtrial, Xgrid, np.array([1])
           ) +                (1-px)*self.aveH_classification(Xtrial,
           Xgrid, np.array([0]))


    def plot(self, iteration):
        if len(self.bounds)==1:
            self.plot_1d
        elif len(self.bounds)==2:
            self.plot_2d(iteration)
        else:
            raise NotImplementedError, "what should we plot in high
               dimensions?"
    def plot_1d(self):
```

```
        pass
    def plot_2d(self, iteration, show_objective=False, resolution=1000,
        show_variance=False, highlight_latest=True, show_entropy=True):

        num_plots = 1
        if show_objective: num_plots += 1
        if show_variance: num_plots += 1
        if show_entropy: num_plots += 1
        fig, ax = plt.subplots(1,num_plots, sharex=True, sharey=True,
           figsize=(4*5,5))
        ax = np.atleast_1d(ax)
        assert ax.size==num_plots

        Xplot = GPyOpt.util.general.multigrid(self.bounds, resolution)
        xx, yy = [x.reshape(resolution,resolution) for x in Xplot.T]

        p, _ = self.model.predict(Xplot)
        mu, var = self.model._raw_predict(Xplot)
        std = np.sqrt(np.clip(var,0,np.inf))
        vmin, vmax = self.model.Y.min(), self.model.Y.max()
       # print 'xx yy'
       # print xx.shape, yy.shape
        CS = ax[0].contour(xx, yy, p.reshape(*xx.shape), vmin=vmin,
           vmax=vmax, cmap=plt.cm.jet)
        plt.clabel(CS, inline=1, fontsize=10)
        if highlight_latest:ax[0].plot(self.model.X[-1,0], self.model.X
           [-1,1], 'ro', ms=16)
        ax[0].scatter(self.model.X[:,0], self.model.X[:,1], 40, self.
           model.Y[:,0],
                linewidth=1.2, vmin=vmin, vmax=vmax, cmap=plt.cm.jet,
                   zorder=10)
        #p = util.normcdf((mu)/np.sqrt(1+np.clip(var,0,np.inf)))# -
           util.normcdf((self.data_lower-mu)/std)
        H = util.entropy(p)
        #print 'probability'

        #print np.min(p), np.max(p)
       # print 'Entropy'
       # print np.min(H), np.max(H)
        ax[0].imshow(1-p.reshape(*xx.shape).T, cmap=plt.cm.gray,
           interpolation='bilinear',
                origin='lower', extent=np.hstack(self.bounds), aspect='
                   auto')

        ax_count = 1
```

```
        #if  show_variance:
        #     ax[ax_count].contourf(xx, yy, std.reshape(*xx.shape), cmap
            =plt.cm.Blues)
        #     ax_count += 1
        if show_entropy:
            foo = ax[ax_count].contourf(xx,yy, H.reshape(*xx.shape),
                cmap=plt.cm.Blues, vmin=0, vmax=util.entropy(0.5))
            plt.colorbar(foo)
            ax_count += 1
        if show_objective:
            foo = ax[ax_count].contourf(xx, yy, self.objective(Xplot).
                reshape(resolution, resolution), cmap=plt.cm.jet)
            plt.colorbar(foo)
        fig.savefig('/scratch/fun3d_image'+str(iteration)+'.png')
        return fig, ax



def aveH_classification(args):
        Xnext = np.vstack((args[0].X, args[1]))
        Y_label = np.vstack((args[0].Y, args[3]))

        model_copy = copy.deepcopy(args[0])
        model_copy.set_XY(Xnext, Y_label)
        p_label, _=  np.asarray(model_copy.predict(args[2]))
        del model_copy
        return util.entropy(p_label).mean()

def calculate_accuracy(model, bound, resolution, Xplot, f_truth):
    model_output, _ = model.predict(Xplot)
    #print  type(model_output)
    #print  len(model_output)

    model_truth = np.array(([1 if i > .5 else 0 for i in model_output])
        ).reshape(-1,1)
   #  print  len(f_truth)
   #  print  f_truth.shape,  model_truth.shape
   #  model
   #  print  np.sum(model_truth  ==  f_truth)
    #for  i  in  model_truth:  print
    print "ruth size?"
    print type(f_truth)
    print model_truth.shape, f_truth.shape
    print model_truth.size, f_truth.size, np.sum(model_truth == f_truth
        )
```

```
    return float(np.sum(model_truth == f_truth))/f_truth.size

def get_data(file_name):
     print file_name
     X = []
     epsilon = []
     data =[]
     f = codecs.open(file_name, "r", "utf-8")
     data = f.readlines()
     for line in data:
          dummy = line.split()
          X.append([float(dummy[0]), float(dummy[2])])
          epsilon.append(calculate_label(float(dummy[4])))
     f.close()
     return np.array(X), np.array(epsilon).reshape(-1,1)


def f(X):
    if len(X.shape)==1:
        X = X.reshape(1,-1)       #    print X
    classification = []
    for point in X:
         f1 = open('fun3d_template.txt', 'r')
         f2 = open('fun3d.nml', 'w')
         for line in f1:
              f2.write(line.replace('$MACH$', '%5.5f'%point[0]).replace
                 ('$CFL$', '%5.5f'%500).replace('$ANGLE$', '%5.5f'%
                 point[1]))

         f1.close()
         f2.close()


         subprocess.call("mpirun -np 7 ./nodet_mpi", shell=True)

         with open('om6inviscid_hist.dat', 'rb') as fh:
              fh.seek(-1024, 2)
              last = fh.readlines()[-1].decode()


         f = open('workfile', 'a')
         f.write('%5.5f'%point[0])
         f.write('\t')
         f.write('%5.5f'%500)
         f.write('\t')
```

```
            f.write('%5.5f'%point[1])
            f.write('\t')
            f.write(last);
            #f.write('\n');
            f.close();

            classification.append(calculate_label(last))

    return np.array(classification).reshape(-1,1)

def calculate_label(line):
        epsilon = 0
        if type(line) == type(32.2):
             epsilon = line
        else:
             epsilon = float(line.split()[1])

        if epsilon <= float(u'10E-15'):
             return 1
        else:
             return 0

def first_step(main_iteration):
     #define the problem:bounds, data limits, function and grid for
         evaluating entropy.
     bounds = [[0.5,2.3],[0,30]]
     data_lower, data_upper = .5, .5

     file_name = '/home/io/jross10/experiments/fun3d/dummyworkfile'
     Xplot, f_truth = get_data(file_name)
     print Xplot.shape, f_truth.shape
     #draw fromt a GP for ground truth.
    # xx,yy = np.mgrid[bounds[0][0]:bounds[0][1]:10j, bounds[1][0]:
        bounds[1][1]:10j]
     #print xx.flatten()
     #print yy.flatten()
    # Xpoints = np.vstack((xx.flatten(), yy.flatten())).T
     #print Xpoints
    # K=GPy.kern.RBF(input_dim=2, variance=10, lengthscale=2).K(Xpoints
        )
     #ypoints = np.array([1 if -1*j[0]+3*j[1]-2 >=0 else 0 for j in
         Xpoints]).reshape(-1,1)
     #ypoints = np.array( [ 1 if np.sqrt((j[0]-1)**2+j[1]**2)>2 else 0
         for j in Xpoints]).reshape(-1,1)
      '''ypoints = []
```

```
    for i in Xpoints:
        if (i[0]-2)**2+(i[1]-2)**2 <= 4:
            ypoints.append(1)
        elif (i[0]+2)**2+(i[1]+2)**2 <= 4:
            ypoints.append(1)
        else:
            ypoints.append(0)
      # print ypoints'''

#       def f(X):
#           if len(X.shape)==1:
#               X = X.reshape(1,-1)         #     print X
      # return (mTrue.predict(X)[0])
    #      print np.array(np.sign([-1*j[0]+3*j[1]-2 for j in Xpoints])).
        reshape(-1,1).size
#           return np.array(([1 if -1*j[0]+3*j[1]-2 >= 0 else 0 for j in X
 ])).reshape(-1,1)
          #return np.array( [ 1 if np.sqrt((j[0]-1)**2+j[1]**2)>2 else 0
              for j in X]).reshape(-1,1)

    #xx, yy = np.mgrid[bounds[0][0]:bounds[0][1]:50j, bounds[1][0]:
        bounds[1][1]:50j]
    #Xgrid = np.vstack((xx.flatten(), yy.flatten())).T



    # In[7]:

    #first few evaluations at random points and construction of GP
        model
    from pyDOE import lhs
    initial_points = 10
    X = lhs(2, initial_points, criterion="maximin", iterations=1000)
    X = X*np.array([bounds[0][1]-bounds[0][0], bounds[1][1]-bounds
        [1][0]]) + np.array([bounds[0][0], bounds[1][0]])
    #print X
    Y = f(X)
    print (Y.shape == Y.shape)
    #print Y
    m = GPy.models.GPClassification(X,Y, kernel=GPy.kern.Matern52(2,
        1, 1.))
    #m.Gaussian_noise.fix(1e-8)
   # m.plot()
    total_accuracy = [[]]
```

```
# In[ ]:




# In[8]:

start = time.time()
iteration = 200
accuracy_list = []
for it in range(iteration):
    start_loop = time.time()
    #optimize now and then.
 #    if it%25 == 0 and it != 0:# and it >1:
        #if it == 1:
        #    pass
        #else:
#           #    pass
#           m.optimize('bfgs')

    #here's the EntABC instance
    mABC = EntABC(bounds, m, data_lower, data_upper)


    #choose the best point seen by the Optimization model
    Xnew, mBO = mABC.select_next_point()

    #plot
    mABC.plot(it)
   # mBO.plot_convergence()

    #actually set the intended observa
    Ynew = f(Xnew)
    #print type(Ynew), type(Xnew)
    setX = np.vstack((m.X, Xnew))
    setY = np.vstack((m.Y, Ynew))

    dummy = calculate_accuracy(m, bounds, 100, Xplot, f_truth)
    accuracy_list.append(dummy)
    print 'Accuracy is '+str(dummy)
    m.set_XY(setX, setY)

    print 'Iteration Number '+str(it+1)
    print 'Iteration Time was '+str(((time.time()-start_loop)
       /60.0))+' minutes'
```

```
        #print calculate_accuracy()
        try:

            with open("fun3d_results_"+str(main_iteration)+".txt", "w
               ") as myfile:
                json.dump(accuracy_list, myfile)
        except:
            print "some where io error"


    print 'Final Iteration'
    end = time.time()

    print 'Total time was '+str((end-start)/60.0)+' Minutes'
    return accuracy_list
def dummy_function():

    # In[ ]:

    # plot the truth
    fig,ax = plt.subplots(1,3, sharex=True, sharey=True, figsize
       =(18,6))
    ff = f(Xgrid).reshape(*xx.shape)
    CS = ax[0].contour(xx, yy, ff.reshape(*xx.shape),np.arange(-6,6,2)
       , vmin=ff.min(), vmax=ff.max(), cmap=plt.cm.jet)
    plt.clabel(CS, inline=1, fontsize=10)
    ptrue = (ff<data_upper)*1.
    ax[0].imshow(1-ptrue.reshape(*xx.shape).T, cmap=plt.cm.gray,
       interpolation='nearest', origin='lower', extent=np.hstack(
       bounds))
    ax[0].set_title('ground truth')


    #space filler
    from pyDOE import lhs
    Xfill = lhs(2, samples=m.Y.size, criterion="maximin", iterations
       =1000)
    Xfill = Xfill*np.array([bounds[0][1]-bounds[0][0], bounds[1][1]-
       bounds[1][0]]) + np.array([bounds[0][0], bounds[1][0]])
    Yfill = f(Xfill)
    mfill = GPy.models.GPRegression(Xfill,Yfill, kernel=GPy.kern.
       Matern52(2, 1, 1.))
    mfill.Gaussian_noise.fix(1e-6)
    mfill.optimize_restarts(3)
    mfill_mu, mfill_var = mfill.predict(Xgrid)
```

```
CS = ax[1].contour(xx, yy, mfill_mu.reshape(*xx.shape),np.arange
   (-6,6,2), vmin=ff.min(), vmax=ff.max(), cmap=plt.cm.jet)
plt.clabel(CS, inline=1, fontsize=10)


from scipy import stats
def probabilities(mu, var):
    return stats.norm.cdf((mu)/np.sqrt(var)) #- stats.norm.cdf(-(
        mu - data_lower)/np.sqrt(var))
ax[1].imshow(1-probabilities(mfill_mu, mfill_var).reshape(*xx.
   shape).T, cmap=plt.cm.gray, interpolation='bilinear', origin='
   lower', extent=np.hstack(bounds))
ax[1].plot(Xfill[:,0], Xfill[:,1], 'ro')
ax[1].set_title('space filling')

m_ent_mu, m_ent_var = m._raw_predict(Xgrid)
ax[2].imshow(1-probabilities(m_ent_mu, m_ent_var).reshape(*xx.
   shape).T, cmap=plt.cm.gray, interpolation='bilinear', origin='
   lower', extent=np.hstack(bounds))
ax[2].plot(m.X[:,0], m.X[:,1], 'ro')
CS = ax[2].contour(xx, yy, m_ent_mu.reshape(*xx.shape),np.arange
   (-6,6,2), vmin=ff.min(), vmax=ff.max(), cmap=plt.cm.jet)
plt.clabel(CS, inline=1, fontsize=10)
ax[2].set_title('Entropic search')
ax[2].set_ylim(yy.min(),yy.max())

ptrue = ptrue.flatten()
#get the probabilities for the space-filling design
mu, var = mfill._raw_predict(Xgrid)
pfill = probabilities(mu, var).flatten()
pfill = np.where(ptrue==1,pfill, 1-pfill)
print 'fill:', np.mean(np.log(pfill)),  np.mean(ptrue==(pfill>0.5)
   )

#get the probabilities for the entropy search
p, _ = m.predict(Xgrid)
#p = probabilities(mu, var).flatten()
p = np.where(ptrue==1,p, 1-p)
print 'ES:', np.mean(np.log(p)), np.mean(ptrue==(p>0.5))



# In[ ]:

x_range = range(initial_points, iteration+initial_points)
```

```
    mfill


    # In[ ]:

    m.plot()
    plt.figure(0)
    plt.plot(x_range, accuracy_list)
    plt.ylabel('Accuracy %')
    plt.xlabel('Total Number of Evaluations')
    plt.plot()


    # In[ ]:

    m.pickle('example_model.pickle')


    # In[ ]:

    print m.Y, m.Y.size


    # In[ ]:

    plt.plot(np.random.randn(10))


    # In[ ]:
def main():
    results = []
    #try:
    accuracy = first_step(1)
    results.append(accuracy)
    #      print "Epoch number "+str(i+1)+" out of 25"
    #except Exception as e:
     #    print str(e)+'we are going to try again'
   # with open('long_circle_accuracy.txt', 'w') as myfile:
   #       json.dump(results, myfile)

# In[ ]:
```

```
if __name__=="__main__":
    main()
```

C FUN3D Images

This section we show all FUN3D images with even number points starting with 10 that have been evaluated on the FUN3D simulator.

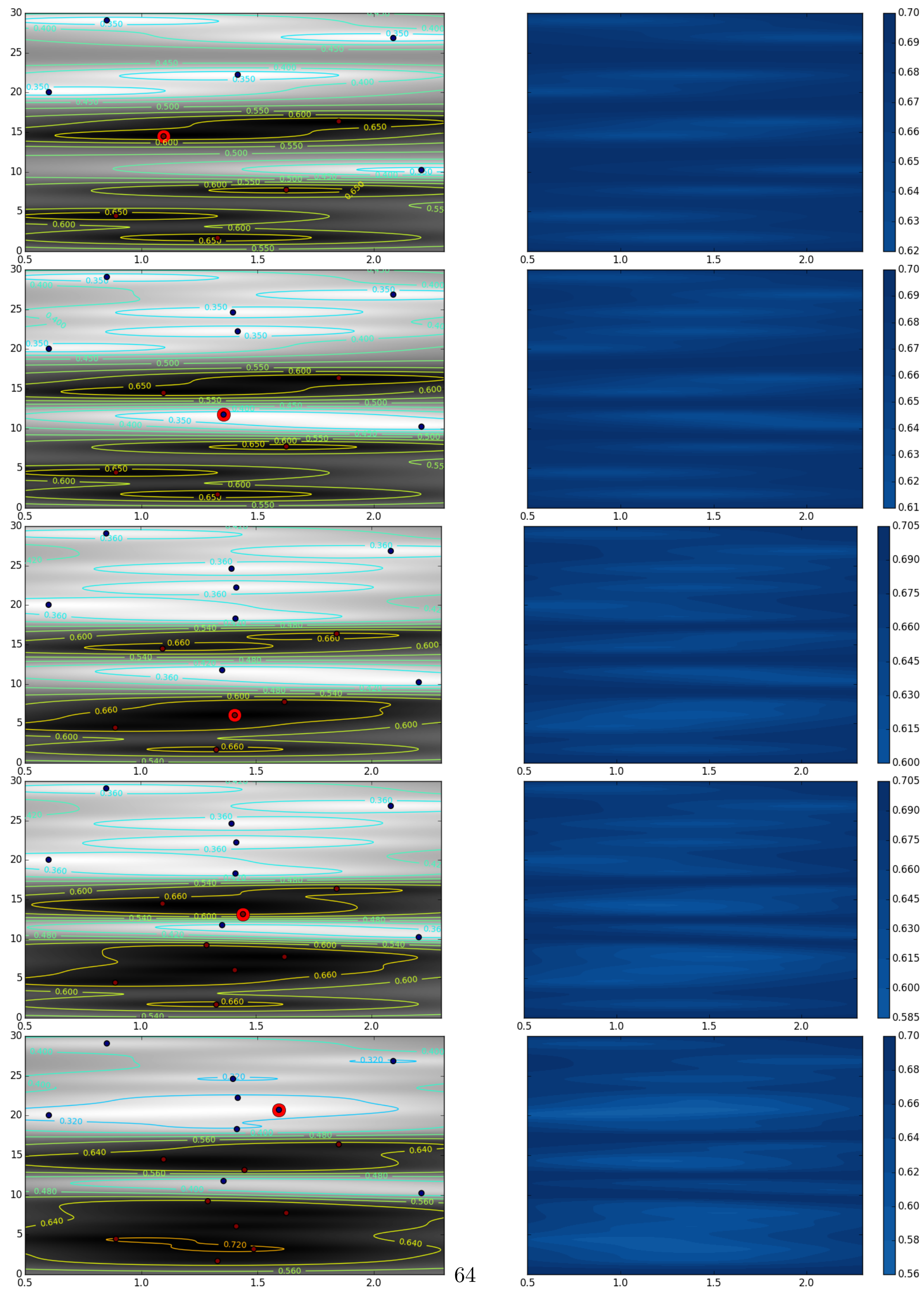


Figure 9: Example of the development of the Emulator(left) and Entropy(right) for point totals 10, 12, 14, 16 and 18 points with out optimization performed on the FUN3D simulator.

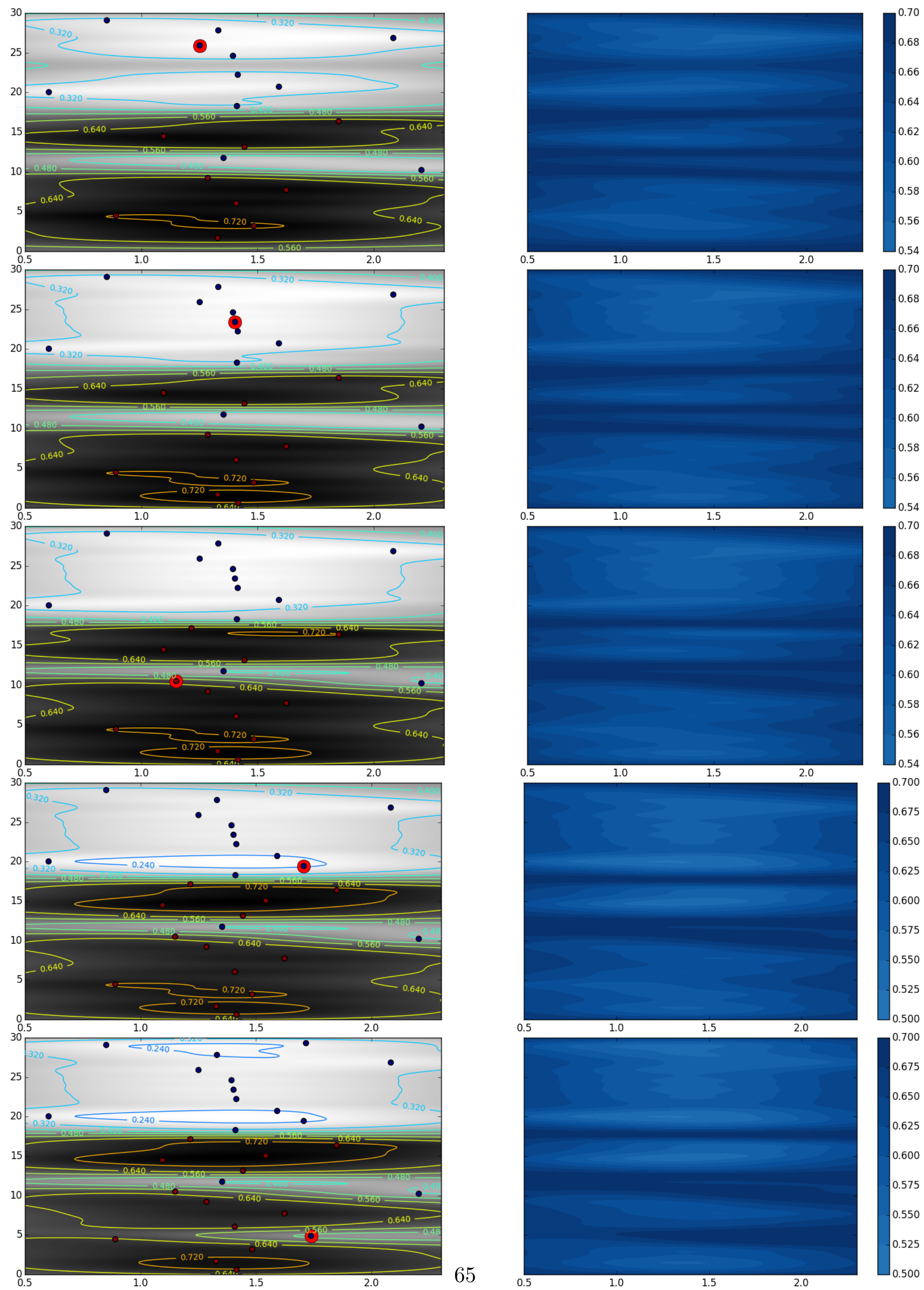


Figure 10: Example of the development of the Emulator(left) and Entropy(right) for point totals 20, 22, 24, 26 and 28 points with out optimization performed on the FUN3D simulator.

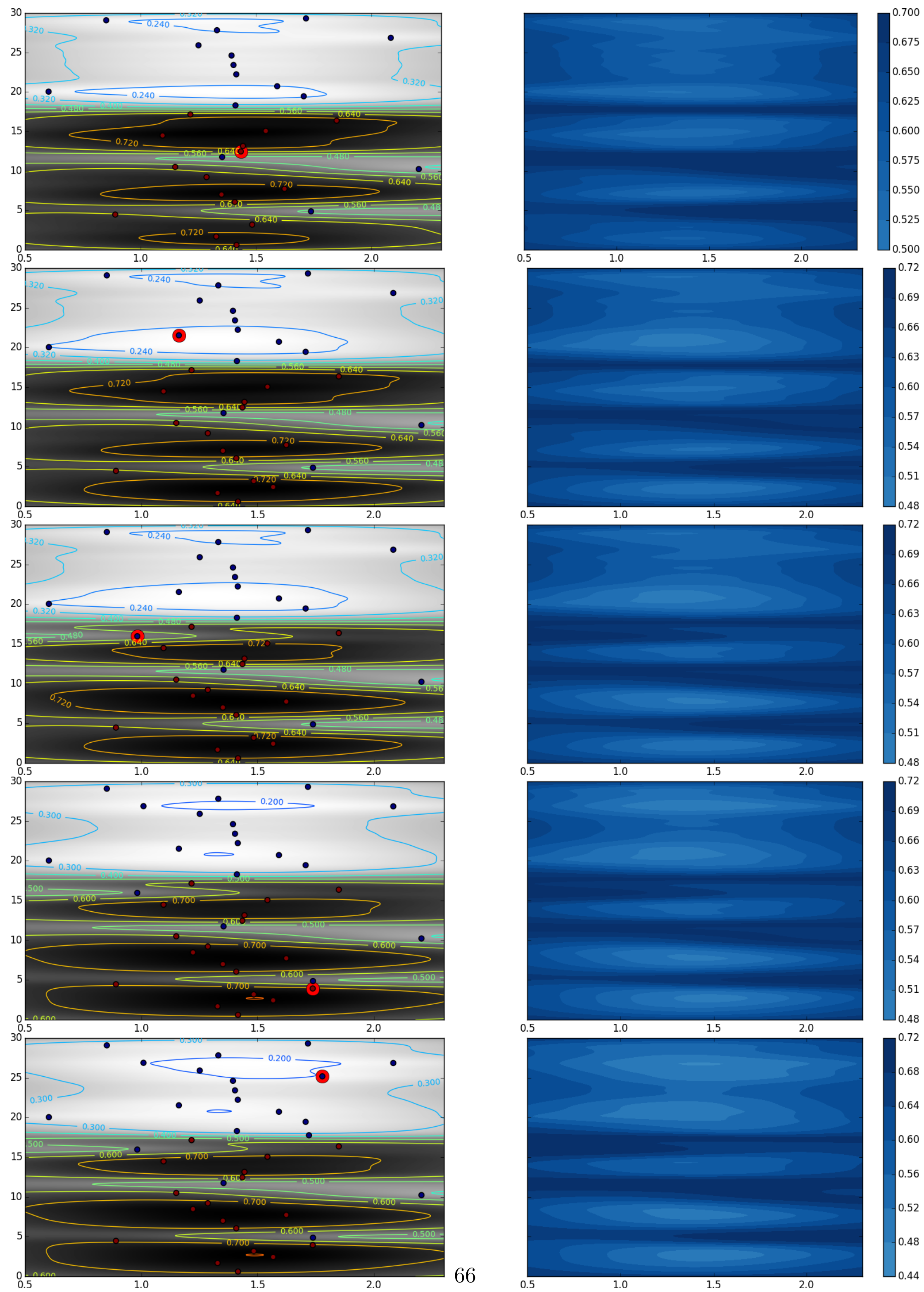


Figure 11: Example of the development of the Emulator(left) and Entropy(right) for point totals 30, 32, 34, 36 and 38 points with out optimization performed on the FUN3D simulator.

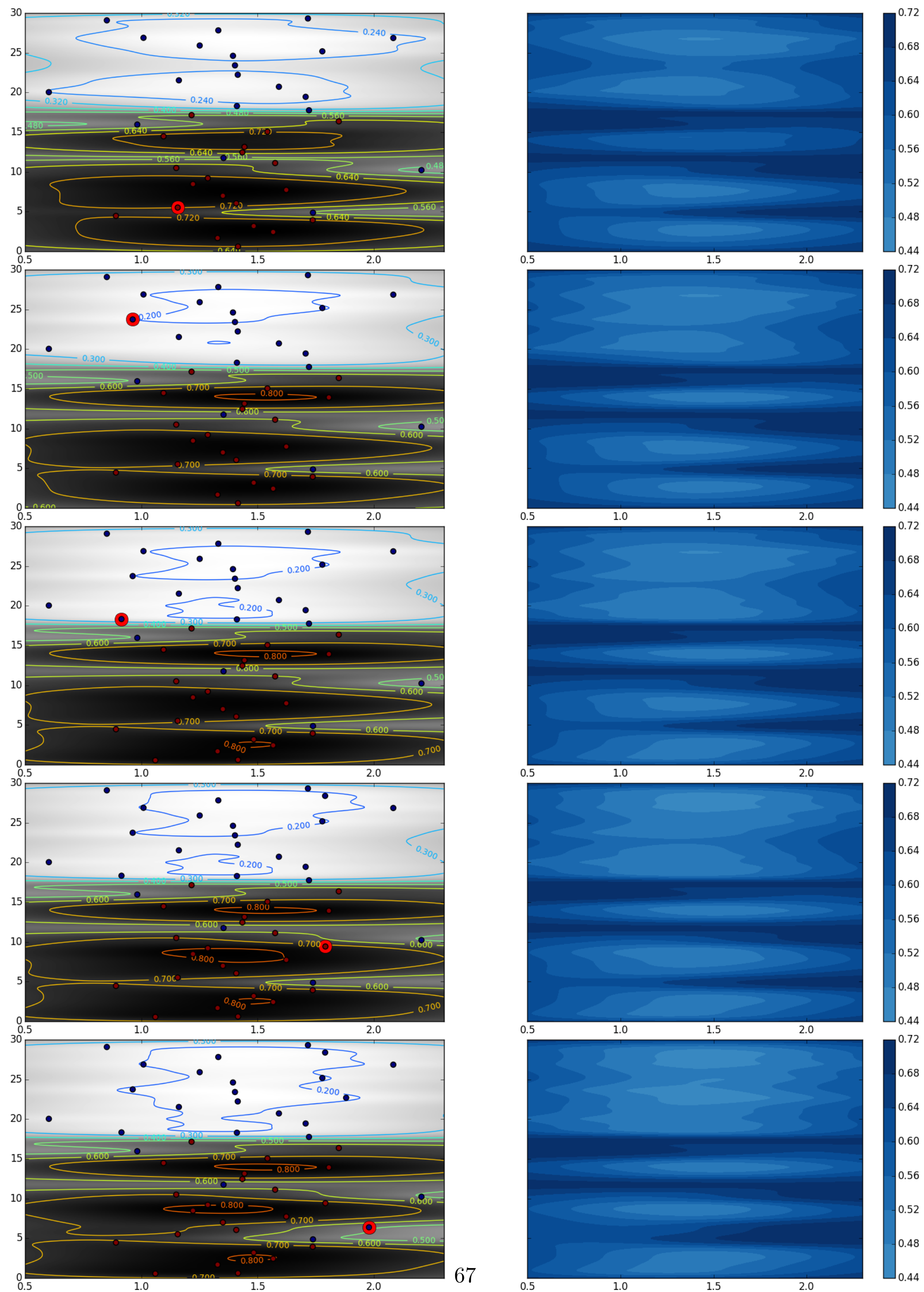


Figure 12: Example of the development of the Emulator(left) and Entropy(right) for point totals 40, 42, 44, 46 and 48 points with out optimization performed on the FUN3D simulator.

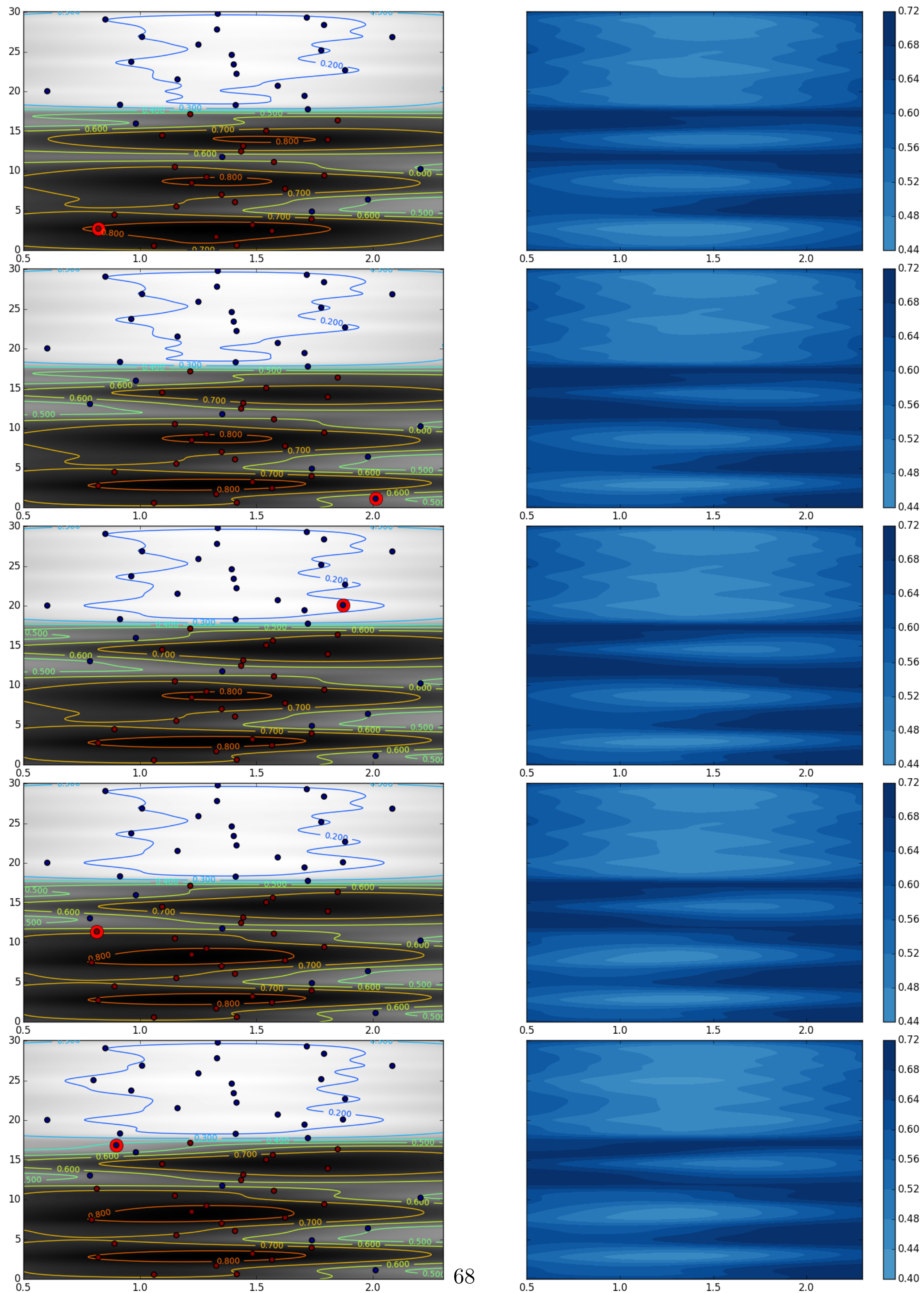


Figure 13: Example of the development of the Emulator(left) and Entropy(right) for point totals 50, 52, 54, 56 and 58 points with out optimization performed on the FUN3D simulator.

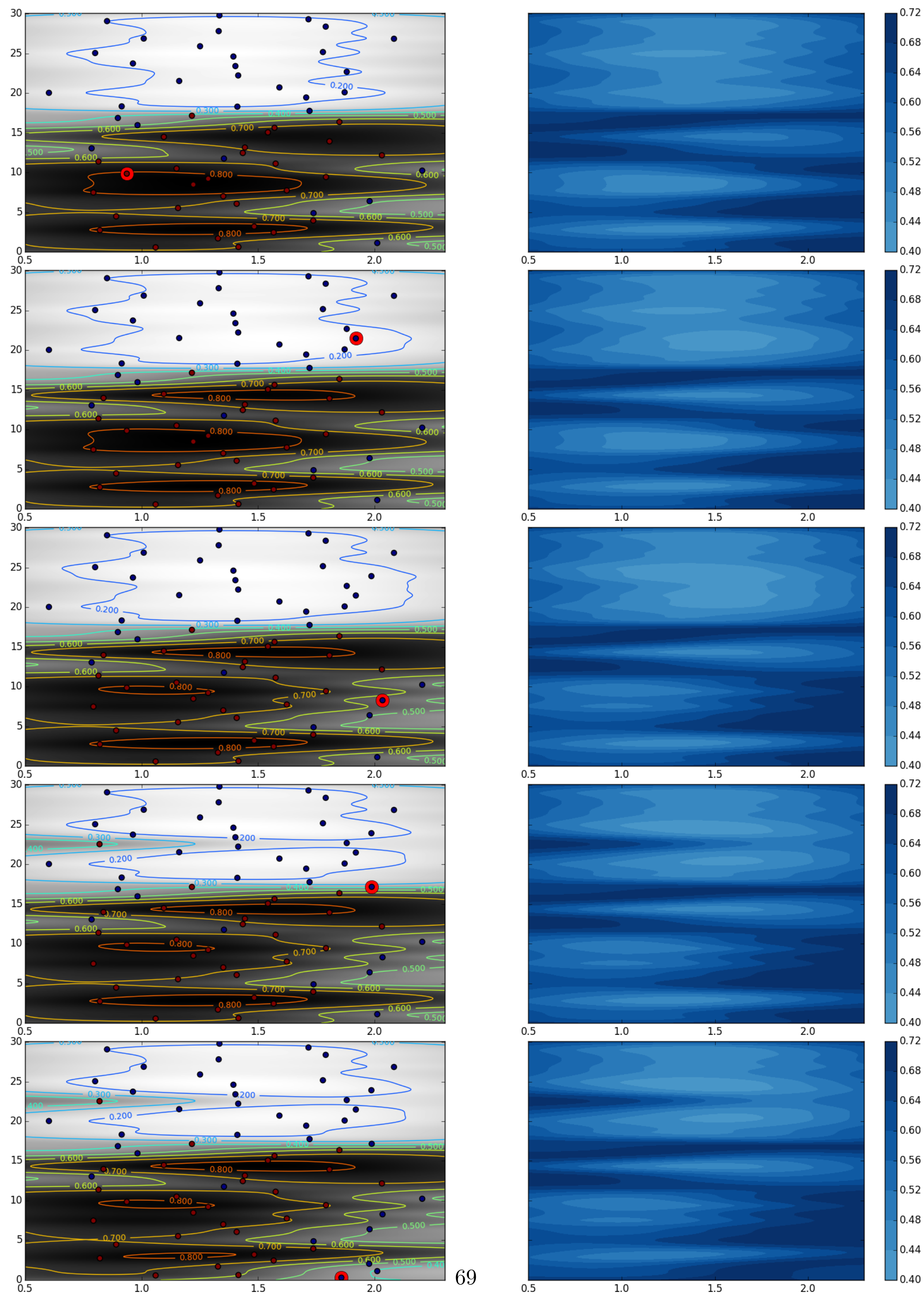


Figure 14: Example of the development of the Emulator(left) and Entropy(right) for point totals 60, 62, 64, 66 and 68 points with out optimization performed on the FUN3D simulator.

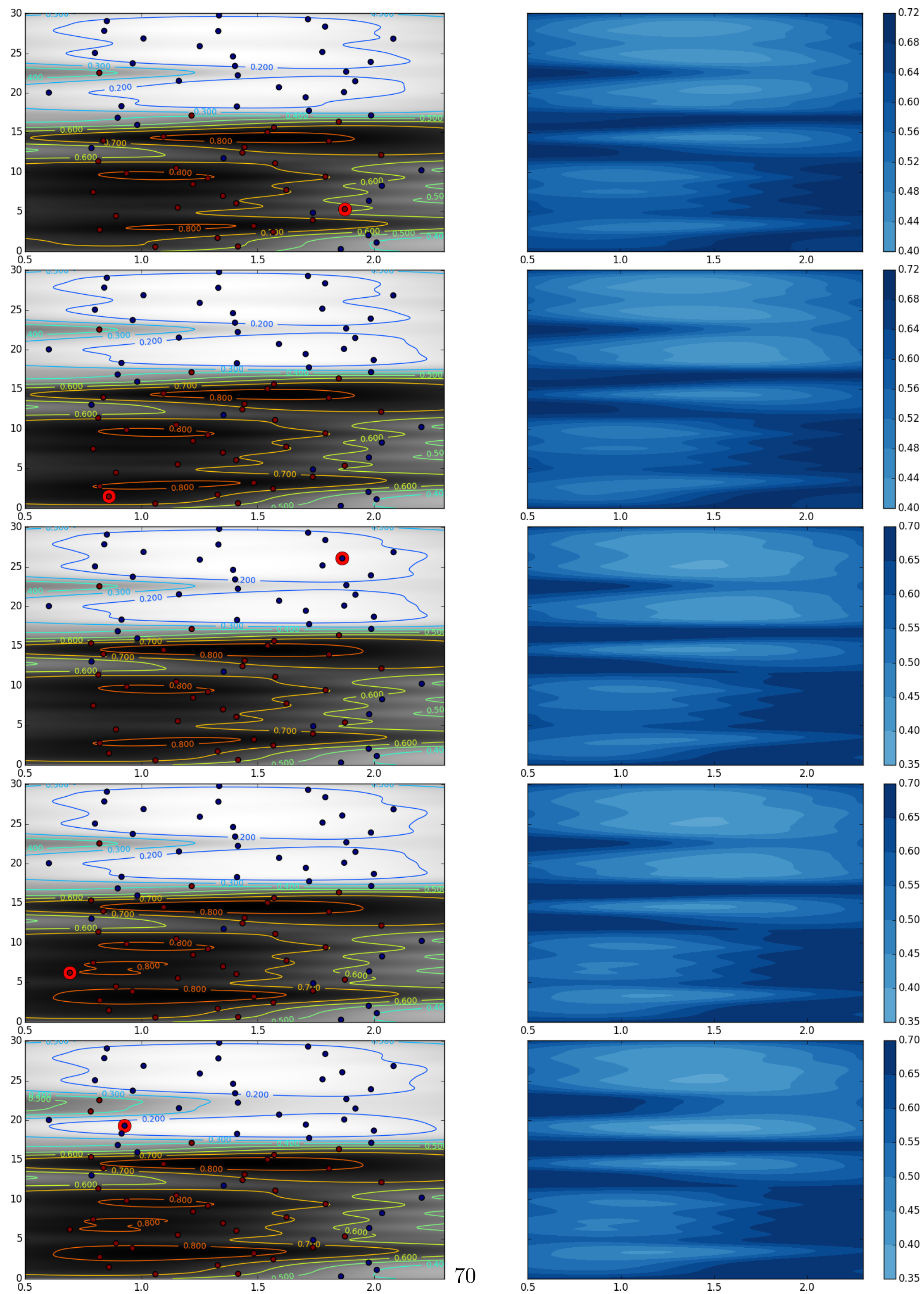


Figure 15: Example of the development of the Emulator(left) and Entropy(right) for point totals 70, 72, 74, 76 and 78 points with out optimization performed on the FUN3D simulator.

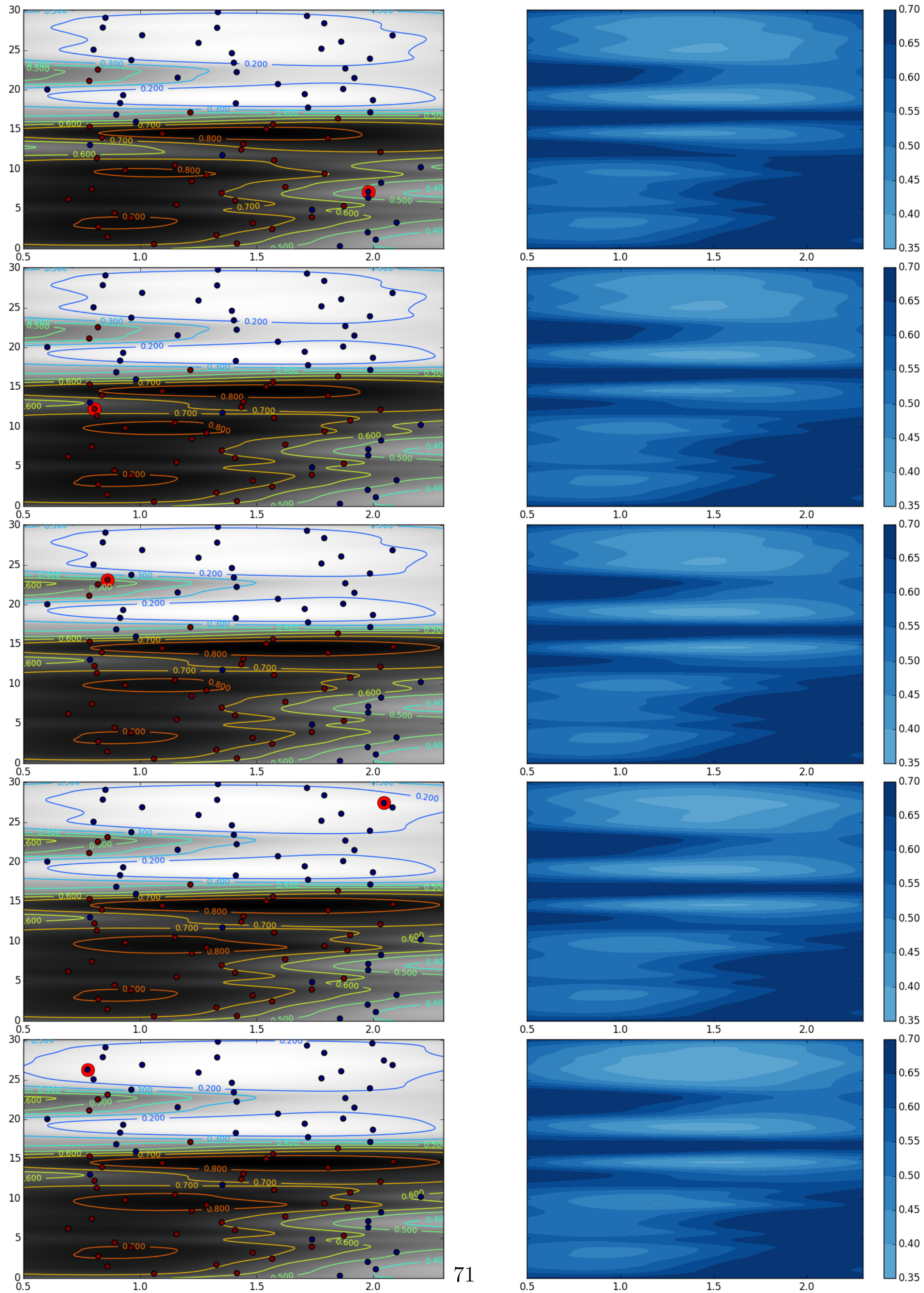


Figure 16: Example of the development of the Emulator(left) and Entropy(right) for point totals 80, 82, 84, 86 and 88 points with out optimization performed on the FUN3D simulator.

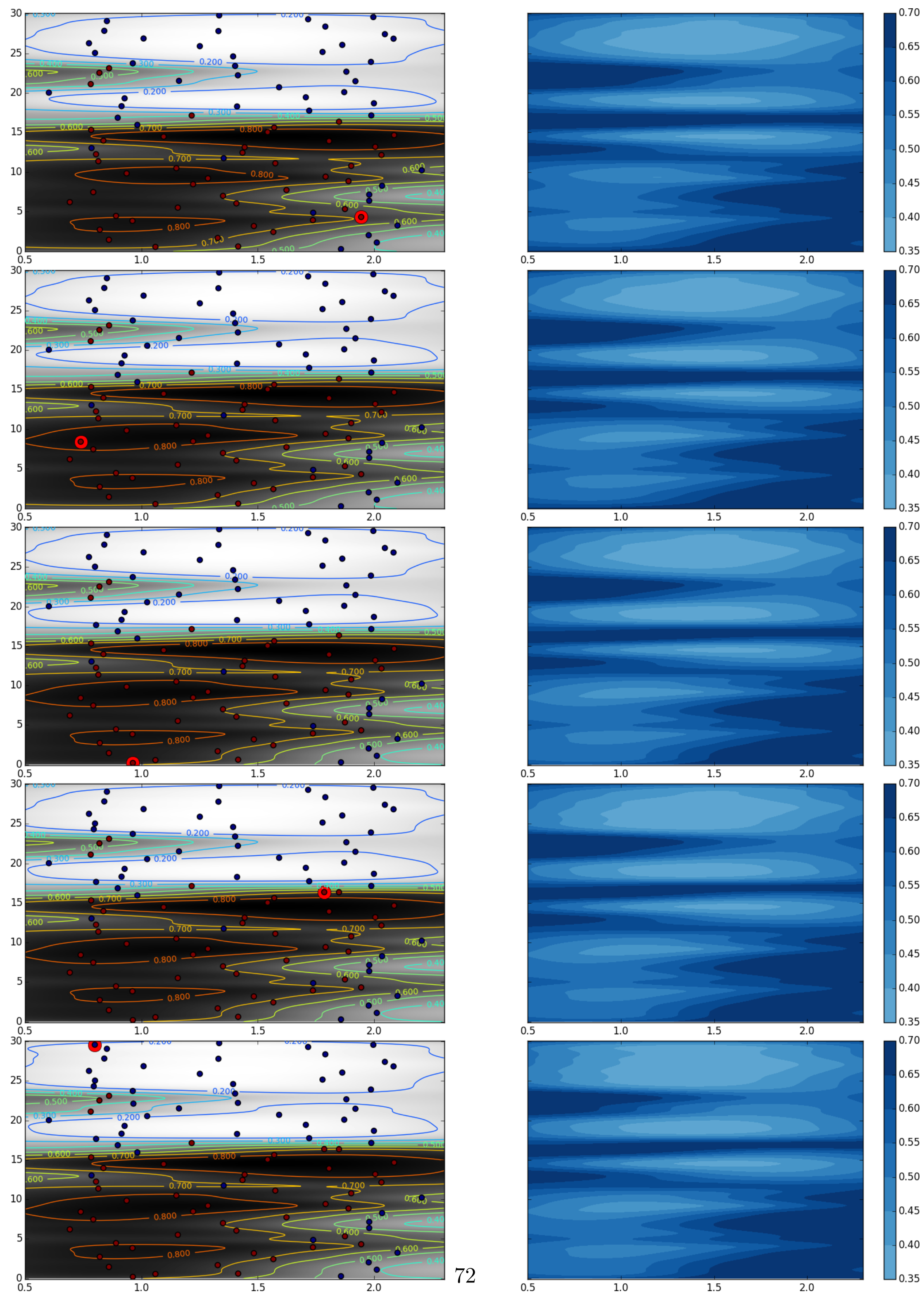


Figure 17: Example of the development of the Emulator(left) and Entropy(right) for point totals 90, 92, 94, 96 and 98 points with out optimization performed on the FUN3D simulator.

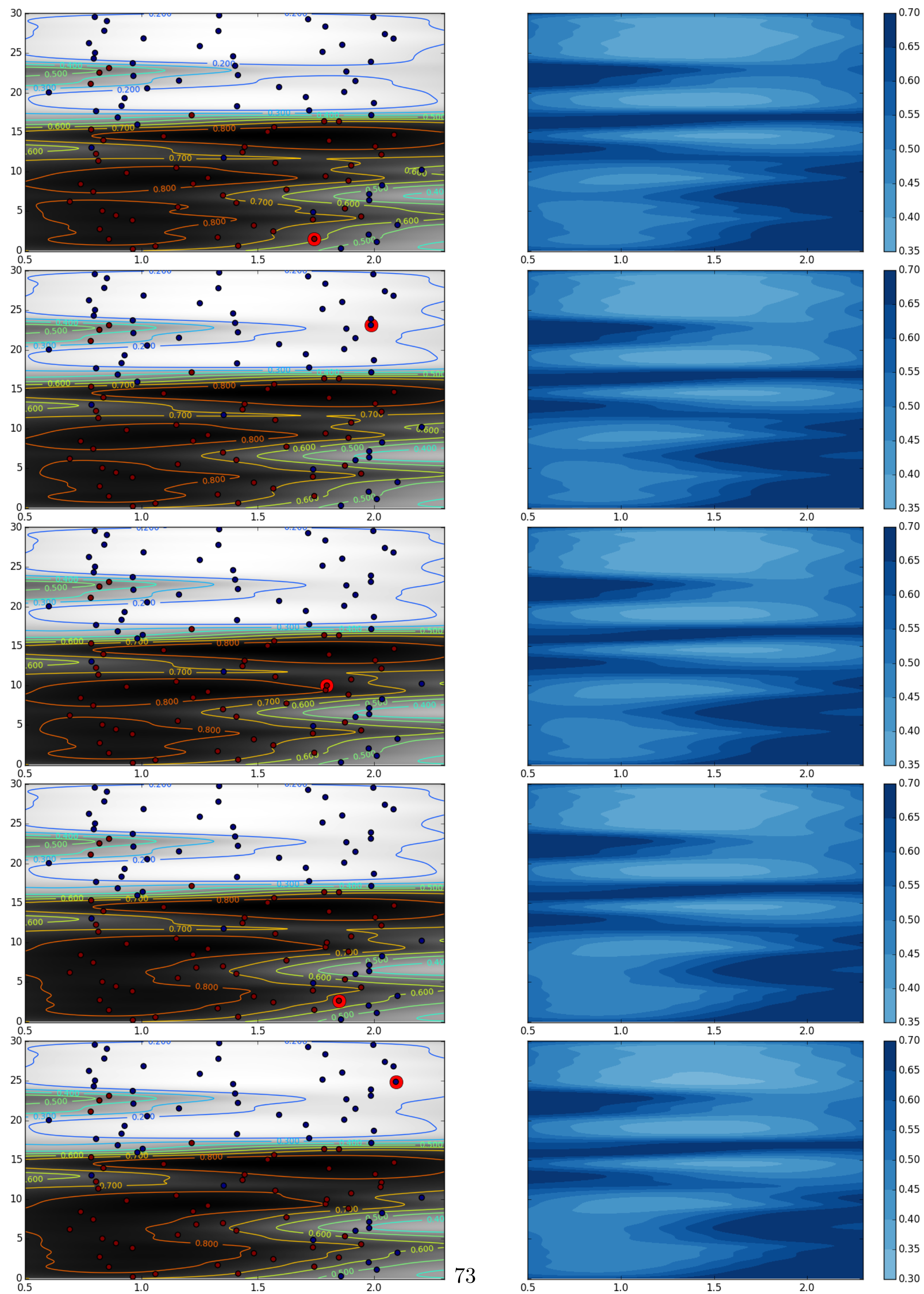


Figure 18: Example of the development of the Emulator(left) and Entropy(right) for point totals 100, 102, 1094, 106 and 108 points with out optimization performed on the FUN3D simulator.

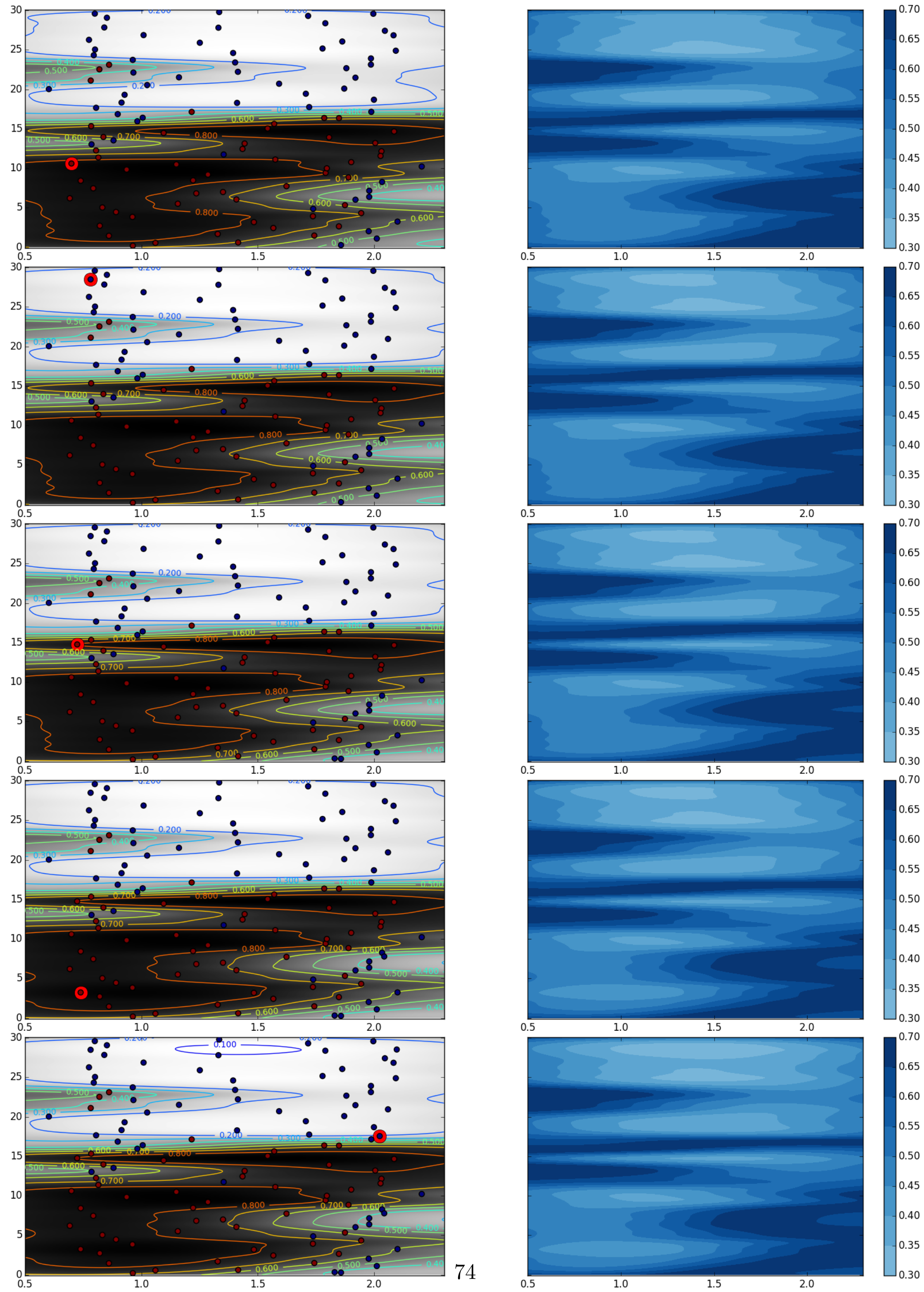


Figure 19: Example of the development of the Emulator(left) and Entropy(right) for point totals 110, 112, 114, 116 and 118 points with out optimization performed on the FUN3D simulator.

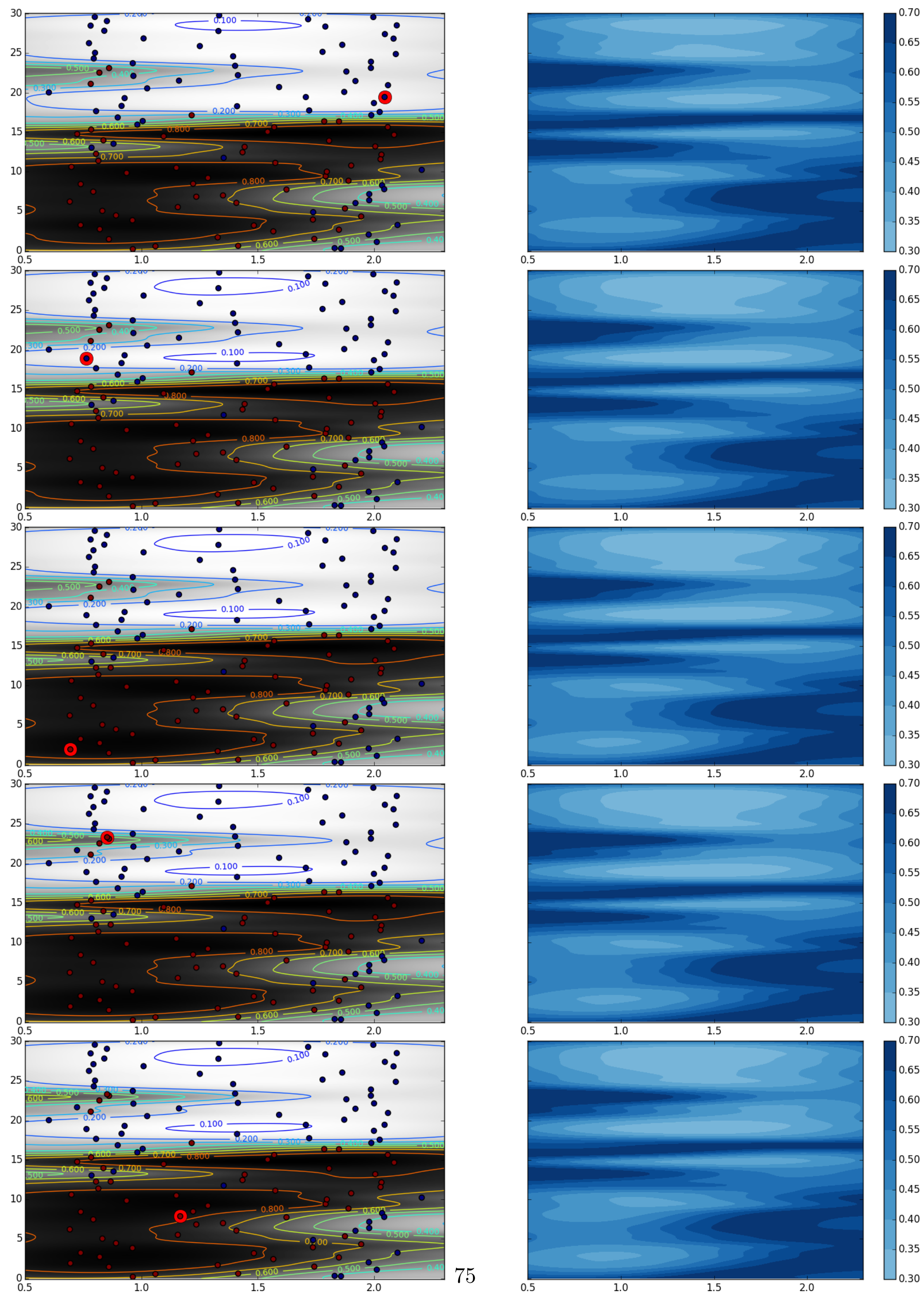


Figure 20: Example of the development of the Emulator(left) and Entropy(right) for point totals 120, 122, 124, 126 and 128 points with out optimization performed on the FUN3D simulator.

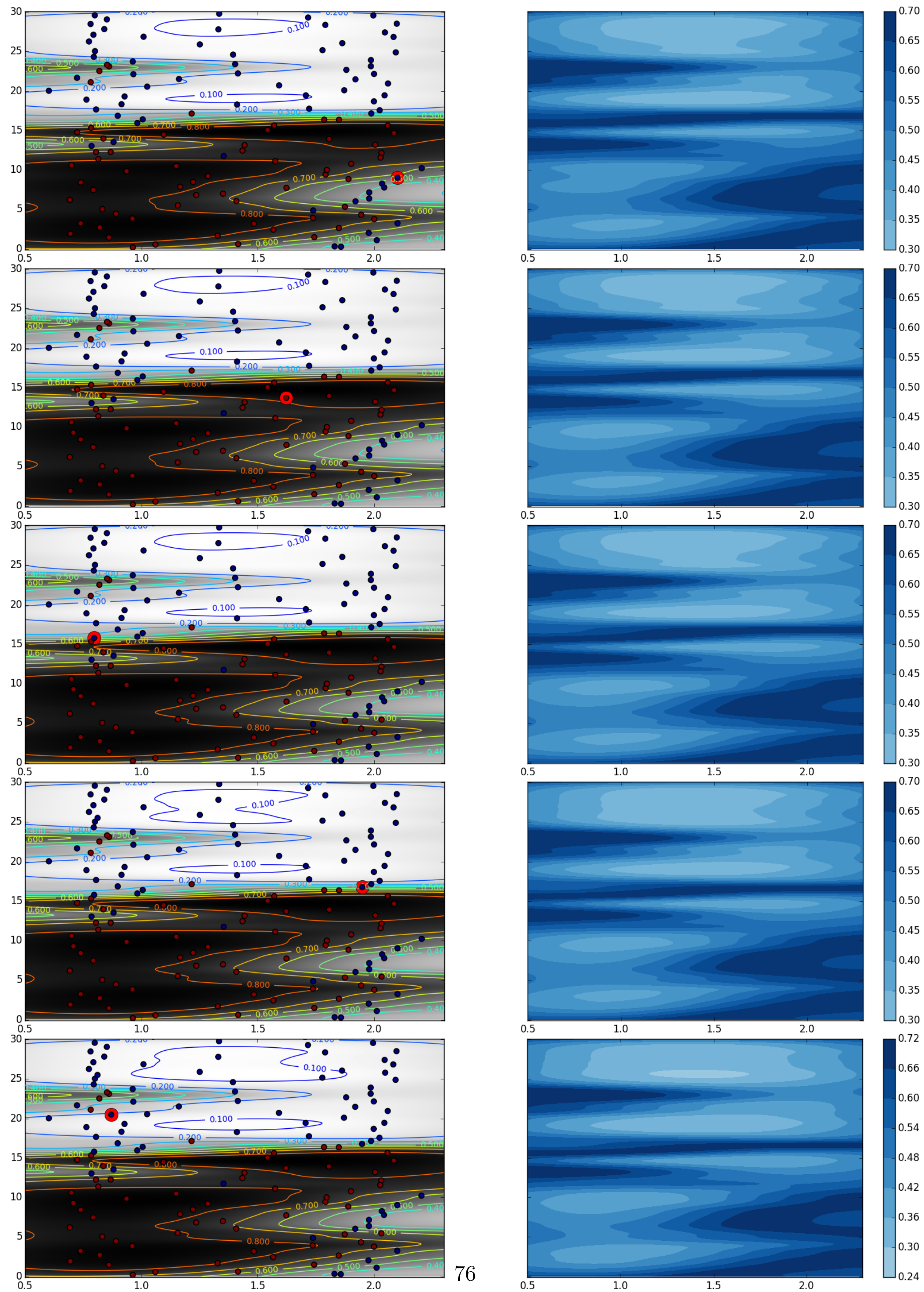


Figure 21: Example of the development of the Emulator(left) and Entropy(right) for point totals 130, 132, 134, 136 and 138 points with out optimization performed on the FUN3D simulator.

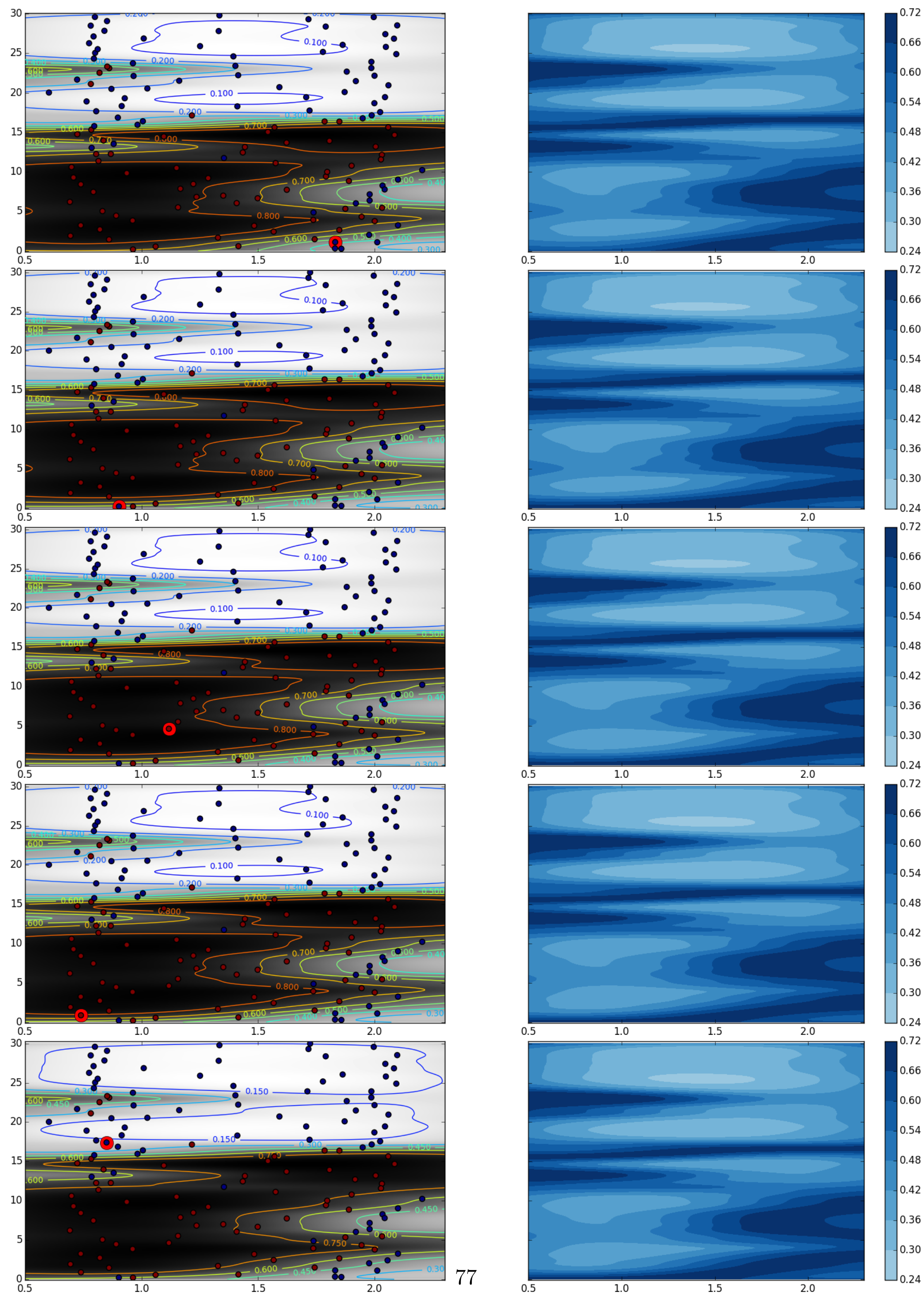


Figure 22: Example of the development of the Emulator(left) and Entropy(right) for point totals 140, 142, 144, 146 and 148 points with out optimization performed on the FUN3D simulator.

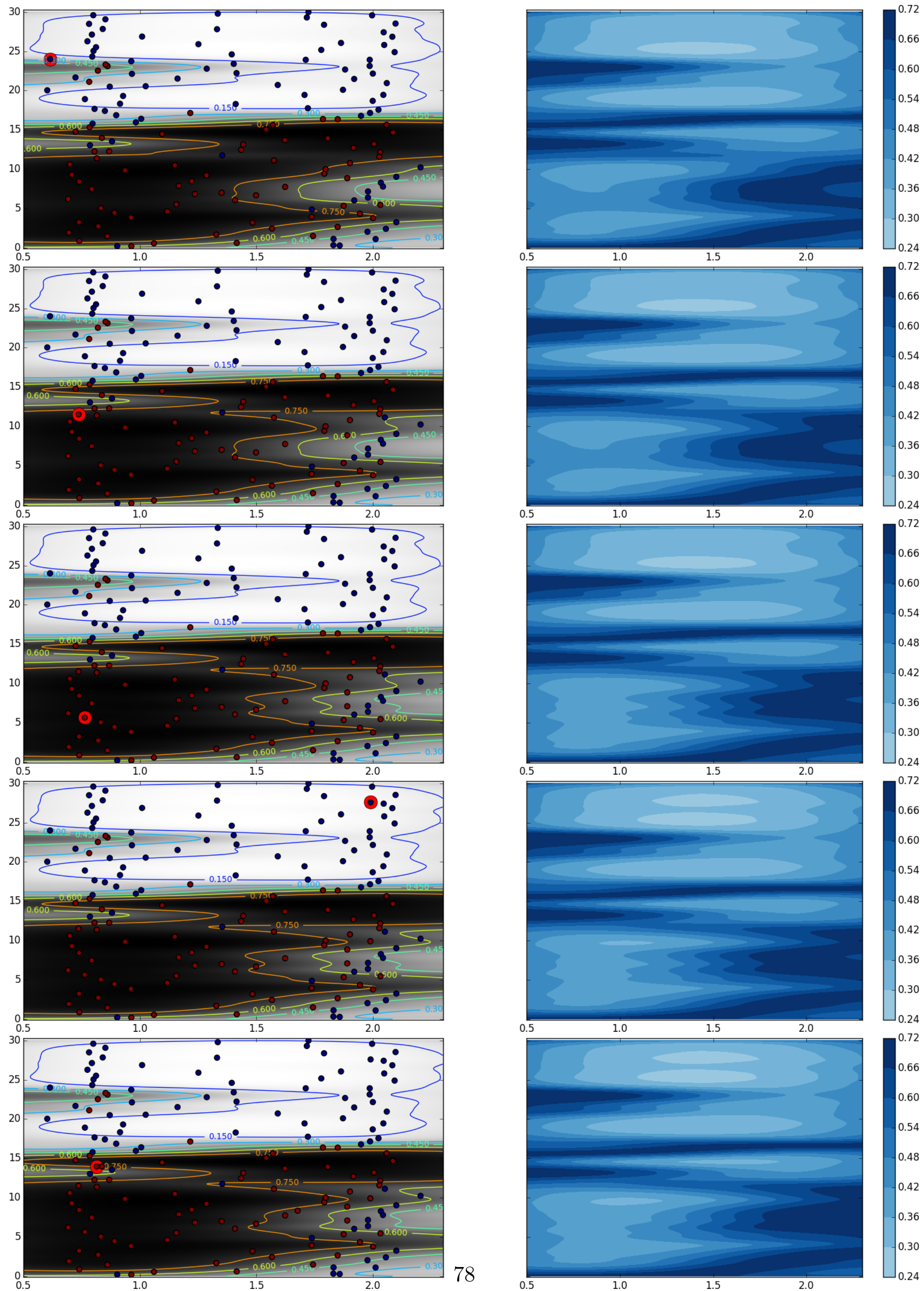


Figure 23: Example of the development of the Emulator(left) and Entropy(right) for point totals 150, 152, 154, 156 and 158 points with out optimization performed on the FUN3D simulator.

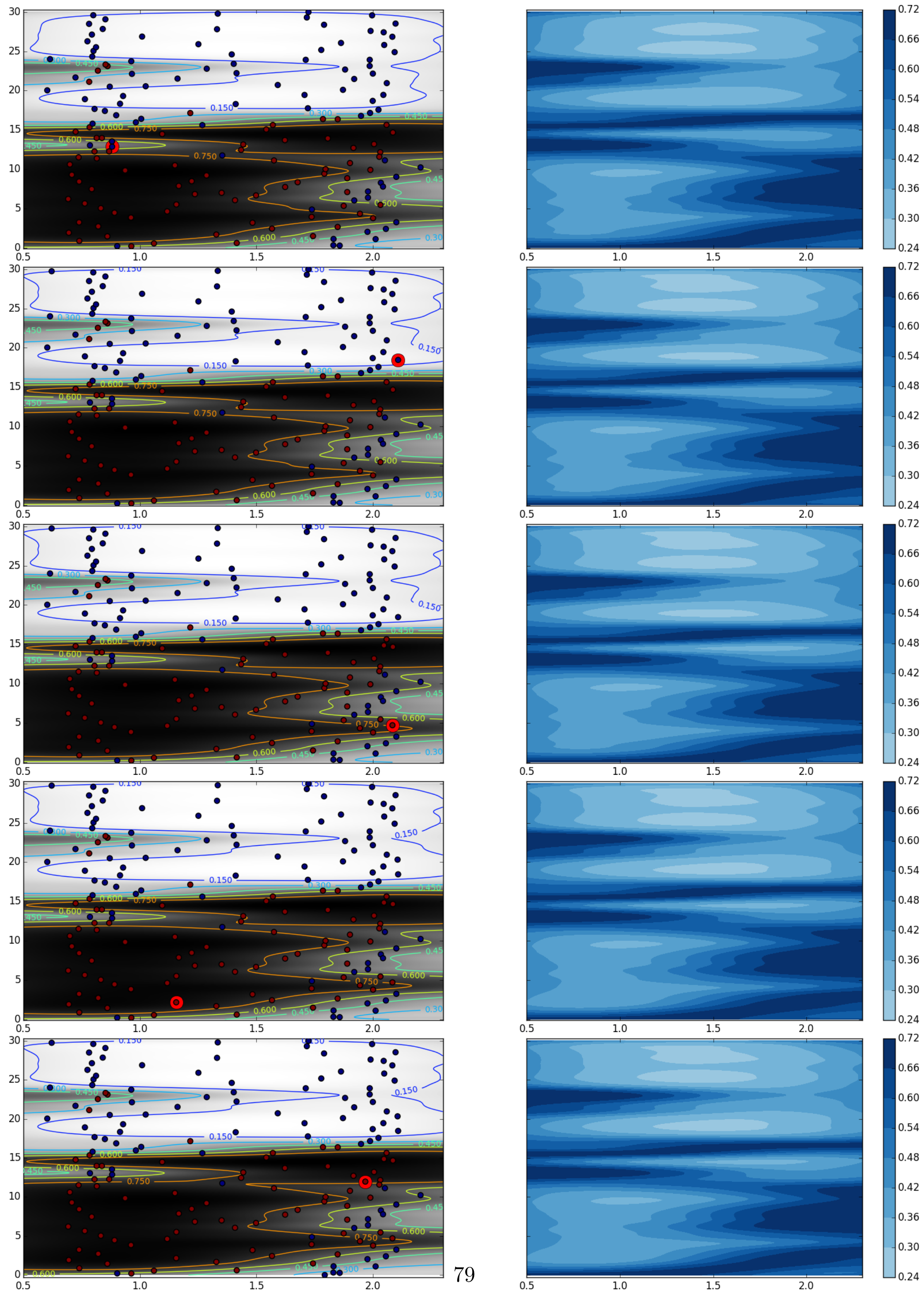


Figure 24: Example of the development of the Emulator(left) and Entropy(right) for point totals 160, 162, 164, 166 and 168 points with out optimization performed on the FUN3D simulator.

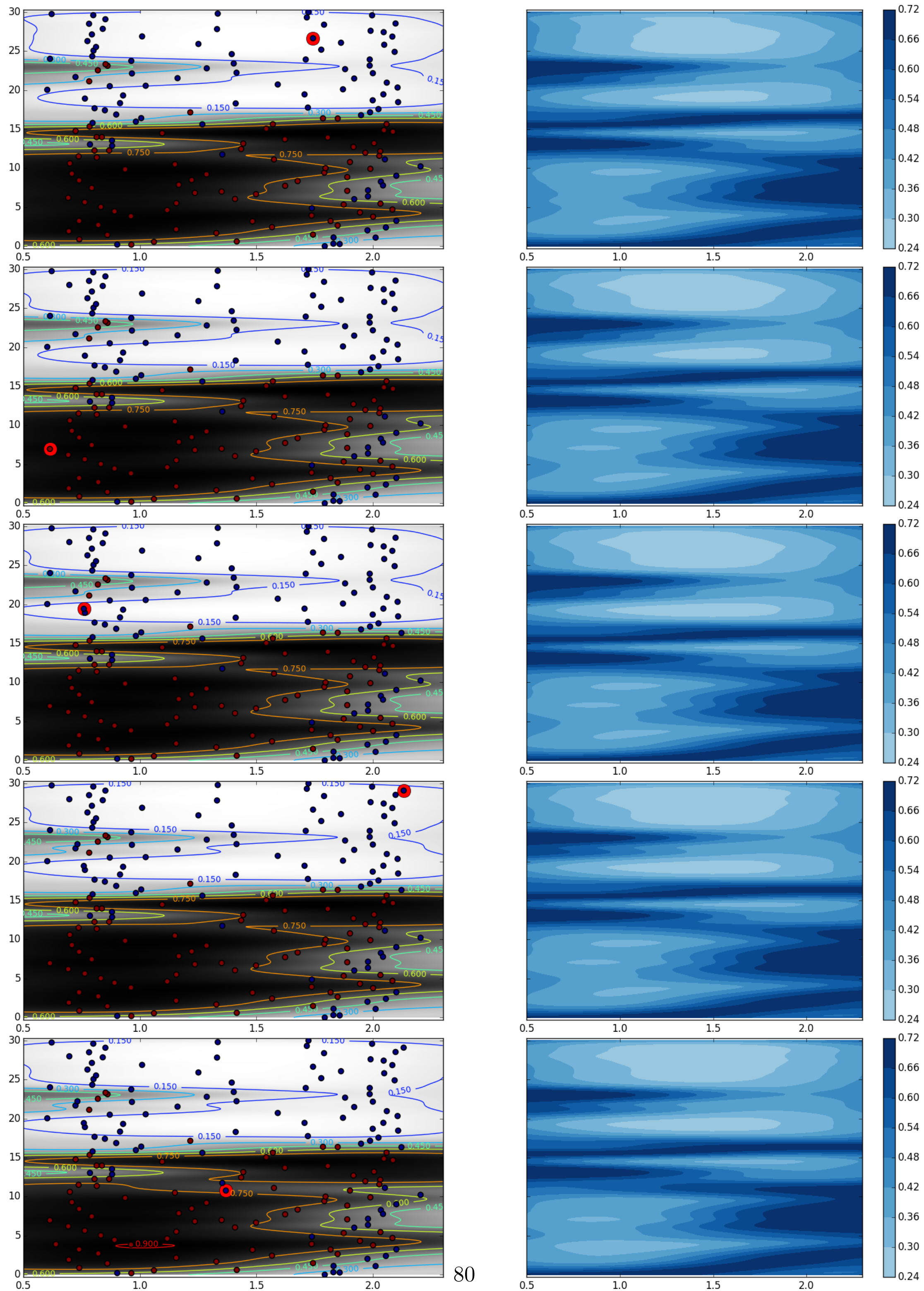


Figure 25: Example of the development of the Emulator(left) and Entropy(right) for point totals 170, 172, 174, 176 and 178 points with out optimization performed on the FUN3D simulator.

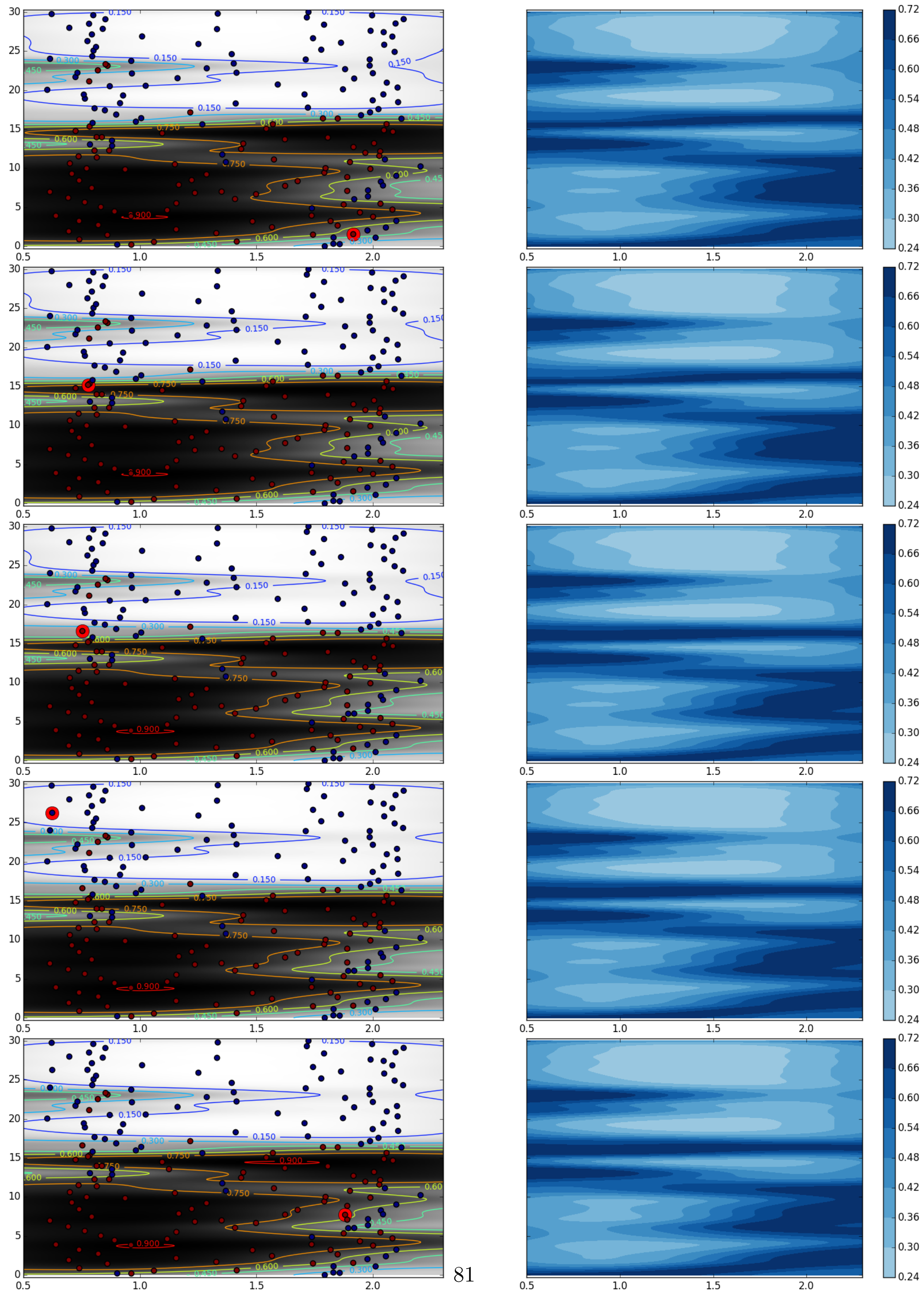


Figure 26: Example of the development of the Emulator(left) and Entropy(right) for point totals 180, 182, 184, 186 and 188 points with out optimization performed on the FUN3D simulator.

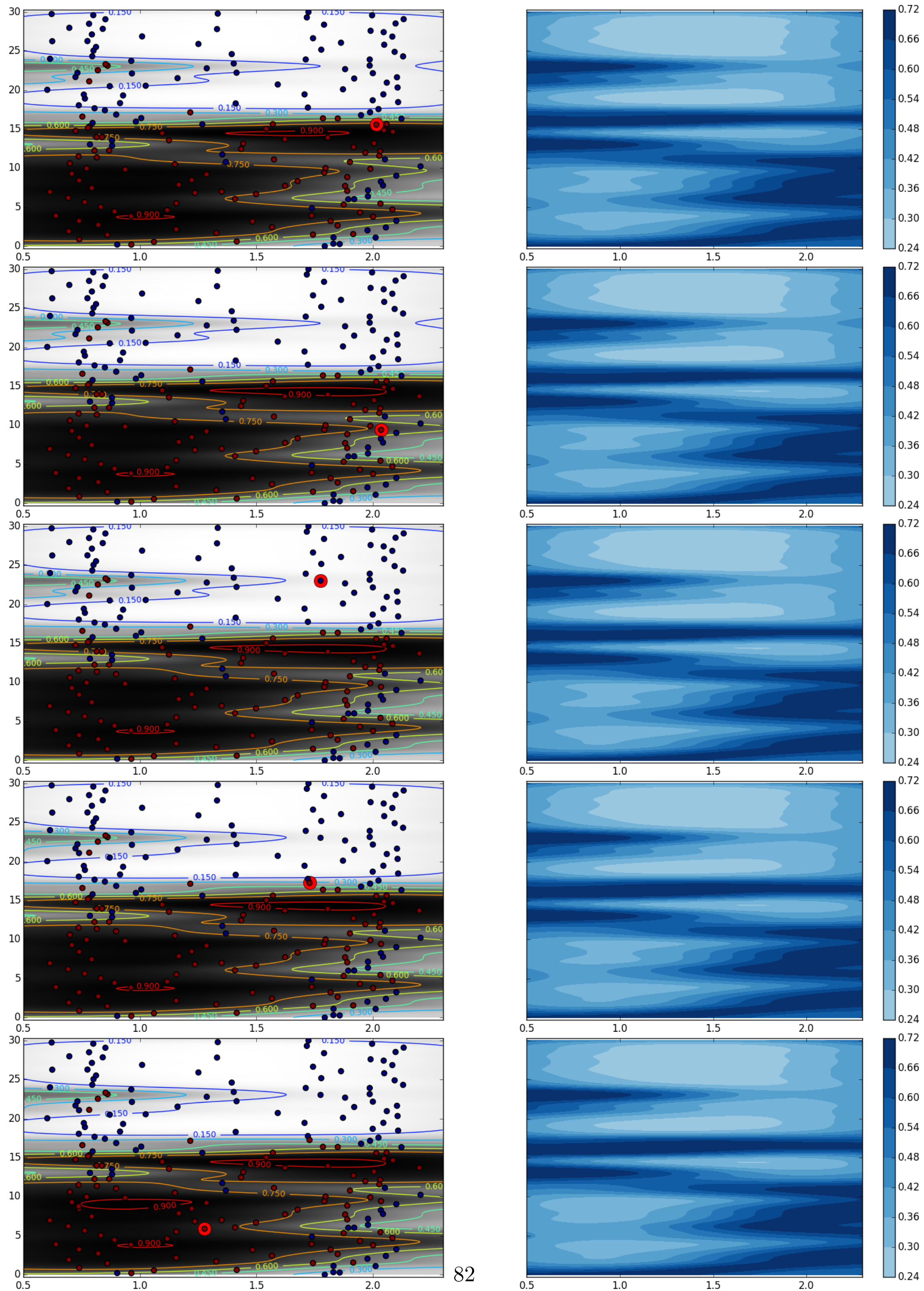


Figure 27: Example of the development of the Emulator(left) and Entropy(right) for point totals 190, 192, 194, 196 and 198 points with out optimization performed on the FUN3D simulator.

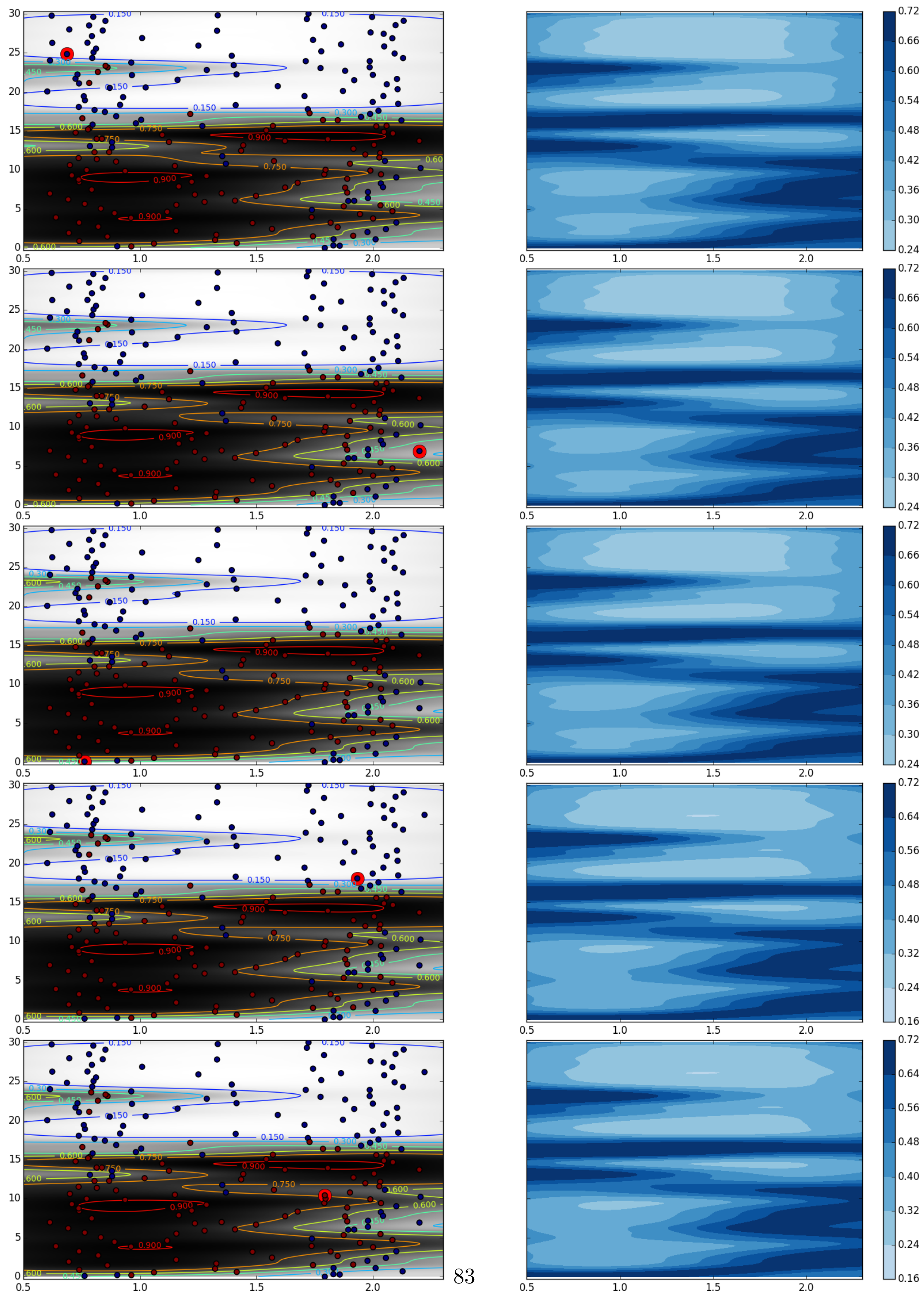


Figure 28: Example of the development of the Emulator(left) and Entropy(right) for point totals 200, 202, 204, 206 and 208 points with out optimization performed on the FUN3D simulator.

REPORT DOCUMENTATION PAGE						Form Approved OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*
01- 04 - 2016

2. REPORT TYPE
Technical Memorandum

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Statistical Emulator for Expensive Classification Simulators

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Ross, Jerret; Samareh, Jamshid A.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
388496.04.01.02

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-20675

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA-TM-2016-219174

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Expensive simulators prevent any kind of meaningful analysis to be performed on the phenomena they model. To get around this problem the concept of using a statistical emulator as a surrogate representation of the simulator was introduced in the 1980's. Presently, simulators have become more and more complex and as a result running a single example on these simulators is very expensive and can take days to weeks or even months. Many new techniques have been introduced, termed criteria, which sequentially select the next best (most informative to the emulator) point that should be run on the simulator. These criteria methods allow for the creation of an emulator with only a small number of simulator runs. We follow and extend this framework to expensive classification simulators.

15. SUBJECT TERMS
Big data; Gaussian process; Machine learning

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	92	(757) 864-9658

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18