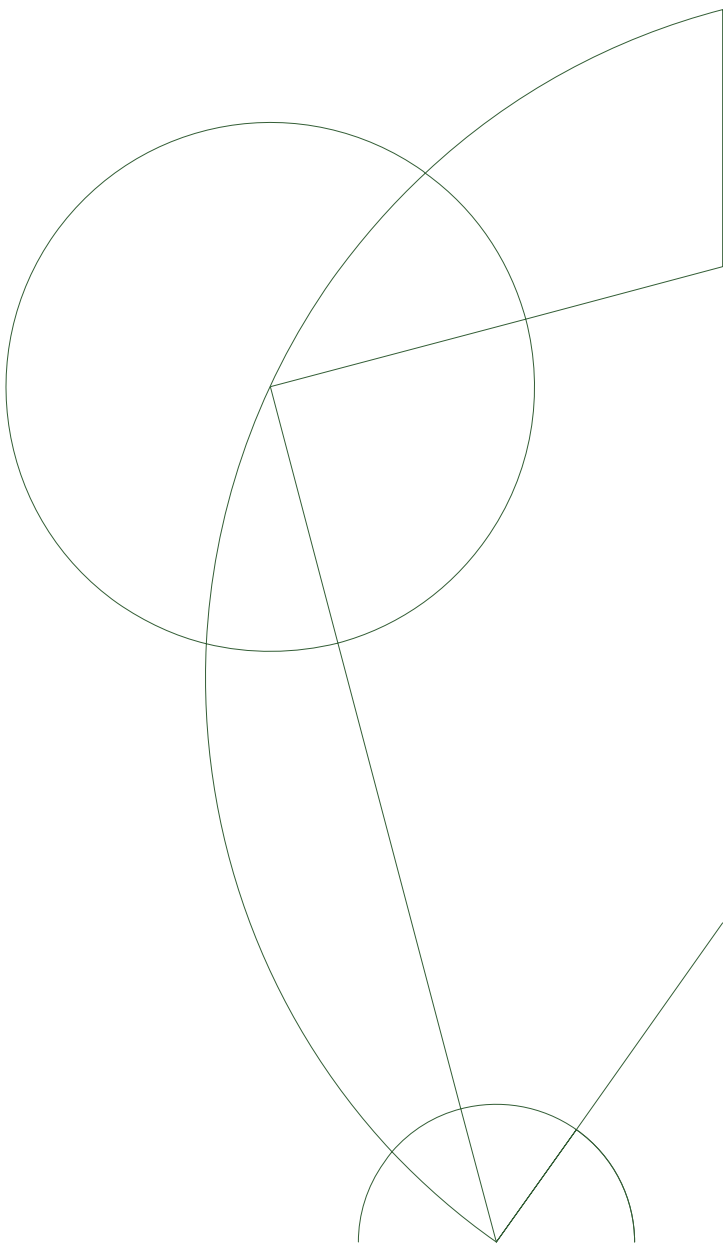
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE
NIELS BOHR INSTITUTE

Master's Thesis in Physics

Automated Parameter Tuning for the Versatile Ocean Simulator (VEROS)

Ida Lei Stoustrup



Supervisors

James Avery & Markus Jochum

ACKNOWLEDGEMENTS

First, I want to thank my supervisors, James and Markus, for their great help with this project, and James especially for taking over when Brian went away for his new job. While it is by far the most independent project I have worked on at university, requiring a lot of choices and prioritisations on my part, it was invaluable to have James' help with the overall structure, and since I am not an ocean physicist it would have been impossible to set up the ocean simulations without Markus' guidance. I'd also like to thank both the eScience group and Team Ocean for helping me figure out their respective computer clusters so I could run my experiments on them.

Secondly, I want to thank my parents for letting me come home during the two lockdowns caused by the COVID-19 pandemic, giving me a safe haven in which to work on my project and lots of support whenever I felt a bit lost in it all. I also want to thank my roommate Emma for being suitably impressed with my GUI even when it was just a window with a button that didn't do anything and for listening to my ramblings about the project, even when I got too technical, too fast and lost her five minutes in.

ABSTRACT

Simulations of the ocean, especially those containing biogeochemistry, are often complicated entities which involve a lot of parameters that govern the behaviour of the system. Many of these parameters are often uncertain or unknown, and without better tools, ocean physicists have to resort to tools like random searches or grid searches, which are often infeasible, especially when used to tune many parameters. This thesis aims to provide a user-friendly, effective tool by utilising a method called Bayesian Optimisation and developing a GUI to give users insight into the process and the ability to make adjustments, if needed. The Python code in the project has been made available as a Python package named *veropt*, available for installation through the Python Package Index (PyPI). A series of test experiments are run to verify the efficacy of the method and compare different parts of it to each other, and finally the method is tested on three increasingly complex ocean simulations.

The method performed well on the test experiments, showing significant improvements over random search and getting close to the global maximum in both single objective experiments. It performed well on the ocean experiments as well, achieving a relative error of $0.001, 0.002$ and 0.004 in the three original optimisation runs and 0.0005 in a fourth optimisation run, where the third ocean simulation was tuned again, this time with a small change in the set-up.

The optimisation problems provided by the ocean simulations turned out to be lacking in complexity, not quite serving as decisive demonstrations of the method's prowess, since they probably could be tuned satisfactorily by random search as well. Still, we demonstrated the *veropt* package's ability to easily inspect and adjust optimisation runs and then discussed some of the many ways in which the method can be made even better and more robust in the future, while maintaining its transparency and adjustability.

CONTENTS

I INTRODUCTION

1 INTRODUCTION AND OUTLINE 2

II BACKGROUND

2 OPTIMISATION 6

2.1 General Strategy . . . 6

2.2 Assumptions and Limitations . . . 8

2.3 Gaussian Process Regression . . . 8

2.3.1 Bayesian Linear Regression . . . 8

2.3.2 Using Basis Functions . . . 13

2.3.3 Gaussian Processes . . . 14

2.3.4 Kernels . . . 18

2.3.5 Optimisation of kernel parameters . . . 21

2.4 Acquisition Function . . . 25

2.4.1 Noisy Upper Confidence Bound . . . 27

2.4.2 Optimisation of the Acquisition Function . . . 27

2.5 Initial Evaluations . . . 29

2.6 Prior Information . . . 31

2.7 Difference Measure . . . 32

2.8 Multiple Objectives . . . 34

2.8.1 Normalisation in MOO . . . 38

3 OCEAN THEORY 39

3.1 The Atlantic Meridional Overturning Circulation . . . 39

3.1.1 Wind-driven Upwelling . . . 41

3.2 Eddies and Mixing . . . 42

3.2.1 Parameterising the Isopycnal Mixing . . . 43

3.2.2 Parameterising the Flattening of Isopycnals . . . 44

3.2.3 Spatially Dependant Vertical Mixing (the TKE Closure) . . . 44

4 THE CODE 46

4.1 Design Strategy . . . 46
4.1.1 Overall Structure . . . 47
4.1.2 Default Set-up . . . 48
4.2 The *veropt* package . . . 49
4.2.1 Underlying Python Packages . . . 49
4.2.2 Python Superclasses . . . 50
4.2.3 The GUI . . . 55
4.2.4 Saving the Optimiser Class . . . 58
4.2.5 The Visualisation Tools . . . 59
4.2.6 Using Priors . . . 62
4.2.7 Predefined Ocean Objectives . . . 63
4.2.8 Slurm Tools . . . 63
4.2.9 The Experiment Class . . . 64
4.2.10 Simple Example . . . 64

III TEST FUNCTIONS EXPERIMENTS

5 TEST FUNCTION EXPERIMENTS 67
5.1 BO vs Random Search . . . 67
5.1.1 Hartmann Test Function . . . 68
5.1.2 Fitting a Sine . . . 68
5.1.3 VehicleSafety Test Function . . . 69
5.2 Tuning Acquisition Function Parameters . . . 70
5.3 Future Work . . . 72

IV OCEAN SIMULATIONS

6 OCEAN SIMULATIONS 74
6.1 Optimisation Set-up . . . 74
6.2 Difference Measure . . . 74
6.3 Simulation One . . . 75
6.3.1 The ACC Set-up . . . 75
6.3.2 Parameterisations . . . 76
6.3.3 The Optimisation Problem . . . 76
6.3.4 Test Runs . . . 77
6.4 Simulation Two . . . 77
6.4.1 The Optimisation Problem . . . 78

6.4.2 Test Runs . . . 78
6.5 Simulation Three . . . 79
6.5.1 The Optimisation Problem . . . 80
6.5.2 Test Runs . . . 80
7 RESULTS & DISCUSSION 81
7.1 Simulation One . . . 81
7.2 Simulation Two . . . 82
7.3 Simulation Three . . . 86
7.3.1 With A Logarithmic Difference Measure . . . 89
7.4 Overall Evaluation . . . 91

V CONCLUSION AND FUTURE WORK

8 CONCLUSION 94
9 FUTURE WORK 96
9.1 The Python Package . . . 96
9.2 Prior Information from Ocean Physics . . . 96
9.3 Space Design, Space Warping Kernel and More . . . 97
9.4 Two Dimensional Objectives . . . 97

VI APPENDIX

10 APPENDIX 100
10.1 Examples . . . 100
10.1.1 Multiple Objectives, Vehicle Safety Test Function . . . 100
10.1.2 Ocean Objective . . . 101
10.1.3 Slurm Support . . . 105
Bibliography 106

LIST OF FIGURES

1 Waves crashing at Juno Beach. Photo by Leo Roomets on Unsplash. . 2

2 Ten functions drawn from the a priori distribution (left) and the posterior (right). The black line gives the mean of the distribution and in the plot on the right, data (sampled from a linear function with noise) with standard deviation is shown as well. 13

3 Ten functions drawn from a prior distribution using an RBF kernel (left) and a corresponding posterior conditioned on some data (right). The data is just randomly chosen numbers. The blue lines are functions drawn from the distributions, the black lines give the mean of the distributions and in the plot on the right, data is shown as red stars. 18

4 Ten functions drawn from a prior distribution using an absolute exponential kernel (left) and a corresponding posterior conditioned on some data (right). The blue lines are functions drawn from the distributions, the black lines give the mean of the distributions and in the plot on the right, data is shown as red stars. 19

5 Ten functions drawn from a prior distribution using a Matern kernel (left) and a corresponding posterior conditioned on some data (right). The blue lines are functions drawn from the distributions, the black lines give the mean of the distributions and in the plot on the right, data is shown as red stars. 20

6 The MLL and its significant two terms are plotted against the value of the lengthscale hyperparameter on the left, given an RBF kernel and a simple test set. On the right the data from the simple test set is shown and we see the mean of three posterior distributions: One with the smallest lengthscale we have used, one with the largest and one with the lengthscale that gives the highest MLL. 24

7 This plot shows the same quantities as in figure 6, except we are using a Matern kernel here. 25

8 Grid search (left) compared to random search (right) for some example objective. We see that random search gives us more point on any one parameter axis. Figure from [11]. 31
9 Example of a non-convex Pareto front. Figure taken from [46]. (The axes were inverted since we try to find a maximum, not a minimum in this thesis.) 37
10 Illustration of the hypervolume indicator in two dimensions. We see how different points contribute to the overall area covered between the Pareto front and the reference point. Figure from [21]. 37
11 The currents of the AMOC (middle), the Antarctic Circumpolar Current (bottom) and currents through the Pacific Ocean (left and right). Figure from [24]. 40
12 A vertical schematic of the volume transport and different upwelling processes of the AMOC. Figure from [24]. 41
13 Overall structure of the *veropt* package. 48
14 The veropt GUI running an optimisation problem with the Brannin Currin test function. 56
15 Prediction plot for the test problem *sine_1param*, available in the *veropt* package. Above the objective function data and corresponding model is shown and below the acquisition function is plotted. 59
16 Prediction plot for the Brannin objective in the BranninCurrin test function from the collection of test functions available in the package *BoTorch*. 60
17 Pareo front plot for the Brannin Currin test functions, showing the distribution of the objective function values for both objectives and marking the dominating (Pareto-optimal) points in black. We also see the mean and variance of the candidate points for the next round of the optimisation. 61
18 The mean (and its uncertainty) of the cumulative best objective function value for Bayesian optimisation and random search on the test function Hartmann, available through the *BoTorch* package. 68
19 Mean and its uncertainty of the cumulative best value (left) and histograms of the final best values (right) both for the test function *sine_3params*, available in the *veropt* package. 69

20 Mean and its uncertainty of the cumulative best value when taking the weighted sum of the three objectives (left) and the mean and its uncertainty of the cumulative best value of each objective (right), both of the VehicleSafety test function available in the *BoTorch* package. 71
21 Cumulative best values (of the weighted sum) for the VehicleSafety function with varying values of α (left) and ω (right). 71
22 Physical set-up for the first and second ocean simulation. Figure from [1]. 75
23 Three test runs of the first simulations. We see the zonal mean of the vertically integrated streamfunction at the southern border of the model as it changes over time for three different values of κ_j. 77
24 Contour plots of the zonally integrated meridional transport for $\kappa_j = 500$ (left) and $\kappa_j = 1500$. 79
25 The vertical minimum of the zonally integrated meridional transport at different meridional coordinates. 79
26 The vertical minimum of the zonally integrated meridional transport at 20°N. 80
27 Objective function values, model predictions and acquisition function values for the first simulation. 82
28 Objective function values at different points for the first simulation. . 83
29 Objective function values, model predictions and acquisition function values for the second simulation. This is a two-dimensional parameter space and we're seeing a slice for each parameter at the point with the best objective function value. The plot with varying κ_j is on the left and the one with varying $\min(\kappa_v)$ is on the right. 83
30 Prediction plots as in figure 29 but after refitting with different lengthscale bounds and with suggested points. 84
31 Three-dimensional figure from the optimisation of the second simulation. We see that the points from a parabola. Note that this is an old figure from an old run and that's why the labels are so small. See footnote for details. 85
32 Progress plot for the second simulation. 86
33 Prediction plot for simulation three at step 3, after the model has been refitted with wider lengthscale bounds. 86

34 Prediction plot for simulation three, after the final step. We note that the lengthscale has been fitted too small. 87
35 Prediction plot for simulation three after the final step. The model has been refitted with lengthscale bounds $[1.0, 5.0]$ but behaves erratically. 87
36 Prediction plot for simulation three after the final step. The model has been refitted with lengthscale bounds $[10.0, 12.0]$ and fits correctly but with too little variance. 88
37 Prediction plot for simulation three after the final step. The model has been refitted with a spectral mixture kernel (SMK). 88
38 Prediction plot for simulation three after the final step, here using a logarithmic distance measure. 89
39 Prediction plot for simulation three after the final step, here using a logarithmic distance measure. 90
40 Progress plots for simulation three. On the left with quadratic difference measure and on the right with the logarithm of the quadratic difference measure. 91

LIST OF TABLES

1 Table with the input variables for the class *BayesOptimiser*. 51

Part I

INTRODUCTION

1

INTRODUCTION AND OUTLINE

The ocean is an incredibly complicated system, reigned by numerous differential equations and only really approachable with simplifying assumptions.

All the same, understanding it and its large-scale currents is of incredible importance, e.g. to understand the climate of the world and how it might be changed by global warming.

To understand the behaviour of the ocean in greater detail, computer models are therefore often used to simulate its behaviour. Besides from the great complexity, this is made more difficult by two factors; The great circulations of the ocean take a very long time to come into equilibrium (up to several centuries) and they depend significantly on small-scale movements.

This means we have two conditions we'd like to avoid when trying to run a simulation; We need to run the model for a long time *and* at a high resolution.

To overcome this problem, a coarser spatial resolution is often chosen and sub-grid processes are then approximated with parameterisations to make sure the large-scale tracer developments are still correct. This then leads to a large amount of parameters with unknown

Figure 1: Waves crashing at Juno Beach. Photo by Leo Roomets on Unsplash.

or uncertain values, and when the simulations run, they often behave quite differently than the real-world ocean.

In order to make the simulations more correct, the parameters with unknown values are often *tuned* by comparing part of the model's output to some real-world data and changing the parameters' value until the output matches the data. In a more physical sense, this means that we change the behaviour or strength of the parameterisation until some chosen part of the simulation behaves the same way as its real-world counterpart. This could correspond to checking that a current runs the right way with the right strength or that some specific kind of algae has the right quantity and distribution.

The goal of this thesis is to develop a tool that can help ocean physicists tune their models without requiring too many simulation runs and then to test this tool on a couple of ocean simulations.

Now, for some specific tuning problem, we could use domain-specific knowledge to predict its overall structure and what kind of optimisation method might work well. But the real challenge in developing the kind of tool we're looking for is to create a tool that will work well not just for *one* tuning problem but for *any* tuning problem it might encounter.

This is a big challenge of course, because we don't know what kind of behaviour to anticipate and thus what kind of optimisation tool that will work well. We could choose something with randomly varying parts, like an evolutionary algorithm but the problem here is that since ocean simulations take such a long time to run, we can't expect to get more than ~ 100 evaluations. This means that any method with significant randomness will probably be too inefficient.

What do we do then? Well, one option is a method called Bayesian optimisation in which a surrogate model is fitted to the data and then optimised in place of the slow ocean simulation.

It requires entering some prior expectations by choosing the form of the surrogate model, which means that we get direct control over which structure the optimisation is expecting *and*, more importantly, that we can change this prior expectation as we want.

Of course, we don't want to make the tool too demanding of the user, so we will try to find a simple default set-up that will hopefully perform well on a wide range of problems, but we will keep the user as a safe-guard, giving them the tools to inspect the optimisation and its progress, making sure that everything is running as it should and otherwise adjust it.

To enable the user to do this, we will develop a user interface from which the model can be inspected and adjusted. This ensures that if the user sets up an optimisation run that

takes several weeks or months to complete, they will always know that the optimisation is on the right track, not wasting the computer time that it's occupying.

We will start this thesis by going through the theory for Bayesian optimisation and some of the many options we have at every part of the method. We will then go through the theory of the ocean simulations we plan to do. Then we will go through the structure, design and most important features of the Python code that has been written.

After this theoretical part, we test out the developed method, first on a couple of test functions and then on the ocean simulations themselves.

Part II

BACKGROUND

2

OPTIMISATION

> All I ask is a tall ship and a star to steer her by.
>
> *Masefield*

Optimisation can be done in many ways and for many different purposes. The focus in this thesis is on finding the best parameter values within a highly limited amount of evaluations (in the order of 100) of the function being optimised. In other words, we are assuming that the evaluations of the function of interest take a very long time which means that the time we spend on the optimisation itself can be allowed several minutes if needed and still be insignificant in comparison.

This means that we want to take advantage of the information available as much as possible and find the best possible coordinates for new points, avoiding wasting time with poorly motivated points that are unlikely to yield anything interesting.

We'll generally assume that we're looking for a maximum during this thesis. This involves no loss of generality, since one can always change the sign of the function being optimised.

While the ultimate goal of the project is to make an optimisation tool that works well for the tuning of ocean simulations specifically, we do not utilise any ocean theory while setting up our general optimisation method, so we will talk about the error between the chosen simulation output and its target value simply as the **objective function**.

2.1 GENERAL STRATEGY

In order to find an optimum within a small amount of evaluations, we need to utilise the information we have already obtained about the function being optimised (i.e. the points we have already evaluated) to the best of our ability. One way to do this is to create a **surrogate model** that tries to predict the behaviour of the objective function. This surrogate

model should optimally have both a predictive value and a measure of uncertainty at every coordinate in the function domain, and ideally it should contain all the information we have about the objective function, allowing us to predict where interesting (high-valued) points may reside.

There will have to be made some assumptions about the objective function, of course, since we could fit an infinite number of functions to any given number of points. More about this in the next section.

Once we have a surrogate function, we need to use it to find out which point(s) we would prefer to evaluate next. Of course, we'll want to look at areas where the predicted value of the objective function is high, but we might also want to look at how uncertain our model is, since areas with high uncertainty might be harbouring maxima that our model just doesn't have enough information to predict yet. The function that describes the desirability of a point as the next evaluated point (given the information from the surrogate model) is called the **acquisition function** and can take many different forms. To find the next point we simply find the maximum of this function. This means that we have to choose a method of optimisation for this as well. Since the surrogate model does not have the long evaluation time of the true objective function, we can use a different kind of optimisation for this that assumes short evaluation time and thus utilises a large amount of evaluated points.

The surrogate model will often have one or more parameters to tune it to the points evaluated so far. Before this tuning can happen, we will need some data to fit to, so we always start out an optimisation with some **initial evaluations**. This can be done randomly or one can choose a method that aims to fill out the space in a desirable way.

We can now summarise the optimisation process in the following steps:

Algorithm 1: Overall strategy

```
Evaluate initial points;
for opt_step in n_opt_steps do
    Fit hyper parameters of model to all evaluated points;
    Find optimum of acquisition function to obtain new candidate points;
    Evaluate objective function at new candidate points;
end
```

2.2 ASSUMPTIONS AND LIMITATIONS

In order to approximate the objective function with a surrogate model, we have to make some assumptions about it. This means we have to either know something about it in advance or make some general assumptions that will work for a wide range of functions. A simple assumption could be that the function is continuous and varies with some average length.

If we don't know anything prior to optimising and can't make a general assumption like the one above, optimising can become very difficult, if not impossible. A worst case scenario would be some function that had the same value everywhere except for one very narrow, high-valued peak (essentially a delta-function). With a function like that the best one can do is to perform a random search and hope to be lucky enough to stumble upon the right area by chance. Hopefully, the objective functions encountered when optimising ocean simulations will rarely be so difficult. Still, it is important to know that Bayesian optimisation hinges on the prior assumptions (whether specific to the problem or general) being somewhat reasonable. If the surrogate model fails critically, the method fails and then something as simple as a random search will perform better. Luckily, the prior information given to the Bayesian optimisation can be changed by the user during the optimisation, and so the method can prevail, even if the initial set-up was sub-optimal.

2.3 GAUSSIAN PROCESS REGRESSION

One way to construct a surrogate model is to use something called *Gaussian process regression*. We will go through the theory of that now.

We start out with simple linear regression, expand that model with basis functions and then end up using Gaussian processes to do the regression, which we will show gives us an equivalent to using an infinite amount of basis functions, without having to explicitly use the basis functions.

2.3.1 *Bayesian Linear Regression*

As described above, we want to make a surrogate model that tries to predict the behaviour of the objective function and we either have no information about the objective function

(except for a standard set of assumptions) or some case-specific initial information about what kind of function we're expecting.

The question is now how to make a model that gives us the best approximation of the objective function, given the points we have already evaluated, despite not having an equation to describe it.

One way to do this could be to just guess and e.g. assume that the function is linear. An obvious problem with this, of course, is that if the true objective function is *not* linear, our fit will fail critically. Making this assumption is thus just a stepping stone on the way to greater things and we will revoke it soon enough.

This section is inspired by Chapter 2 of [33].

Given the linearity assumption, we can start by fitting a weight for each parameter x_i in the input vector $\mathbf{x} = [x_1, x_2..x_n]$ and we'll have that the model takes the form,

$$f(x) = \mathbf{x}^\mathbf{T}\mathbf{w} \tag{1}$$

where $f(x)$ is the surrogate model and $\mathbf{w}$ is a vector of weights. To put it in another way, we are making the assumption that the objective function has an equation like the following one:

$$y = \mathbf{x}^\mathbf{T}\mathbf{w}_{true} + \epsilon \tag{2}$$

where y is the objective function, $\mathbf{w}_{true}$ is a vector of the true weights and ϵ is Gaussian noise of the form $\mathcal{N}(\mathbf{0}, \sigma_n^2)$.

The prior knowledge we have in this case would then surmise of the form of the surrogate model (which here assumes a linear objective function) and whatever guesses we make about the weights $\mathbf{w}$. One option for the latter is to assume that the weights are Gaussian distributed with zero mean and some covariance matrix Σ_w so that,

$$\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \Sigma_w) \tag{3}$$

where $\mathcal{N}(\mu, \sigma^2)$ signifies a Gaussian distribution with mean μ and variance σ^2. The $\sim$ sign here means that the variable on the left is drawn from the distribution on the right.

We can now look at what these assumptions mean practically. If we know the covariance matrix Σ_w, describing the prior information of the weights, we can draw a vector of weights from the distribution $\mathcal{N}(\mathbf{0}, \Sigma_w)$. We can then use our model $f(x) = \mathbf{x}^\mathbf{T}\mathbf{w}$ and evaluate it with the weights we have drawn, on some function domain we've been given, and we'll see a linear model with some specific slope. In one dimension, this is just a straight line.

If we did this some number of times we would then get a sense of the *distribution* of functions that we have chosen with our prior information, in one dimension simply a distribution of lines with different slopes.

If we choose our a priori information correctly it should be true that the objective function could be drawn from this distribution of functions, so in this case, the objective function should be a linear function going through the origin with some slope, the value of which is reasonably likely given the prior over the weights, $\mathcal{N}(\mathbf{0}, \Sigma_w)$. If this is not true, the surrogate model will fail to fit the data and our overall method will fail. Now, this might be alarming to the reader since we have currently chosen a very restricted model that will fail at most problems, but we will soon show that other prior distributions of functions can be chosen that will fit (very nearly) *any* function we might come across. (At least in the sense that the model will go through all data points. The quality of the fit will always depend upon how well the objective function corresponds to the prior distribution of functions.)

Another concern regarding our prior information could be whether the prior we have put on the weights is reasonable. This, however, is not as critical an issue since we can look at the *conditional* distribution of the weights, once we have evaluated our initial points. To do this we use Bayes' Theorem which says that,

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{4}$$

where A and B are events with some likeliness and $P(A \mid B)$ signifies the likeliness of A given B. In Bayesian statistical theory $P(A \mid B)$ is known as the *posterior* because it collects the information from the *prior* $P(A)$ and the *likelihood* $P(B \mid A)$. The denominator $P(B)$ is called the *marginal likelihood* and generally just serves as a normalisation factor.

In our case, the posterior will be $p(\mathbf{w} \mid \mathbf{y}(X))$, i.e. it gives us the likeliness for some vector of weights $\mathbf{w}$ given the data $\mathbf{y}$ at coordinates X. This means that the overall equation becomes,

$$p(\mathbf{w} \mid \mathbf{y}(X)) = \frac{p(\mathbf{y}(X) \mid \mathbf{w})p(\mathbf{w})}{p(\mathbf{y}(X))} \tag{5}$$

Let's go through the different parts of it for our case. The prior is the Gaussian distribution we put on the weights, so we have,

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{0}, \Sigma_w) \tag{6}$$

Of course, a part of the overall prior information we're using is also the class of functions (linear in this case) but we choose not to include it in this equation, instead letting it serve as an underlying assumption.

Our likelihood is given by the points we have already evaluated, at first given by the initial points we always evaluate at the beginning of an optimisation run. If we assume that the points are independent we can multiply the probability of each point given a specific vector of weights $\mathbf{w}$. The probability for each individual point can be found by using equation 2 and exchanging the unknown true weights $\mathbf{w}_{true}$ for the variable $\mathbf{w}$:

$$p(\mathbf{y}(X) \mid \mathbf{w}) = \prod_i p(y_i(\mathbf{x}_i) \mid \mathbf{w}) \tag{7}$$

If we then multiply all the probabilities of each point, we get the following equation for the likelihood:

$$p(\mathbf{y}(X) \mid \mathbf{w}) = \Pi_{i=1}^{n} p(y_i(\mathbf{x}_i) \mid \mathbf{w}) = \Pi_{i=1}^{n} \mathcal{N}(\mathbf{x}_i^{\mathbf{T}}\mathbf{w}, \sigma_n^2) \tag{8}$$

$$= \Pi_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_n} \exp(-\frac{(y_i - \mathbf{x}_i^{\mathbf{T}}\mathbf{w})^2}{2\sigma_n^2}) = \mathcal{N}(X^{\mathbf{T}}\mathbf{w}, \sigma_n^2 I) \tag{9}$$

As intuitively must be the case, this is simply a Gaussian with mean $X^{\mathbf{T}}\mathbf{w}$ and variance equal to the variance of the noise-parameter ϵ, so when $\mathbf{y} - X^{\mathbf{T}}\mathbf{w}$ is small (i.e. when the discrepancy between the data and the model is small) the probability of drawing the data set $\mathbf{y}$ (given the weights $\mathbf{w}$) is high.

Notice that if there is no noise, the likelihood must simply be 1.0 if the weights make the model go straight through the points and 0.0 otherwise. This means that if there is no noise, the likelihood will select (from the prior) exactly the functions that go through all the data points and the posterior will simply consist of this selection.

In this linear case, this set could maximally consist of one function, but if we didn't have the linearity assumption and had a larger set of functions in our prior (perhaps an *infinite* amount of functions), the likelihood would pick out all the ones that agreed with the data.

In this case, where we do have noise, the functions in our posterior simply consist of a distribution of functions where the most likely one are the ones that have the highest likelihood, i.e. the ones that have the least accumulated distance to the points they run through.

The marginal likelihood $p(\mathbf{y}(X))$ gives us the likelihood of the data regardless of the weights. It is called the *marginal* likelihood because it is calculated by looking at the joint distribution of the conditional distribution of the data given the weights and the prior distribution of the weights and then marginalising out the weights by integrating over all possible values of them;

$$p(\mathbf{y}(X)) = \int p(\mathbf{y}(X) \mid \mathbf{w}) p(\mathbf{w}) \mathrm{d}\mathbf{w} \tag{10}$$

To get an equation for the posterior we now need to multiply the likelihood with the prior (here choosing to omit the normalisation):

$$p(\mathbf{w} \mid \mathbf{y}(X)) \propto p(\mathbf{y}(X) \mid \mathbf{w})p(\mathbf{w}) = \mathcal{N}(\mathbf{0}, \Sigma_w) \cdot \mathcal{N}(X^{\mathbf{T}}\mathbf{w}, \sigma_n^2 I)$$
$$= \frac{1}{(2\pi\sigma_n^2)^{n/2}} \frac{1}{\Sigma_w\sqrt{2\pi}} \exp(-\frac{|y_i - X^{\mathbf{T}}\mathbf{w}|^2}{2\sigma_n^2}) \exp(\frac{1}{2}\mathbf{w}^{\mathbf{T}}\Sigma_w\mathbf{w}) \tag{11}$$

Of course, the product of two Gaussian distributions will give a new unnormalised Gaussian distribution so the only hassle is to figure out the mean and variance of this new distribution. The marginal likelihood becomes trivial because we know that a Gaussian distribution is normalised by the factor $(\sigma\sqrt{2\pi})^{-1}$where σ is the variance.

To find the mean and variance of the posterior, we simply need to rewrite the above equation as a Gaussian distribution. Doing this gives us,

$$p(\mathbf{w} \mid y(X)) = \mathcal{N}\left(\frac{1}{\sigma_n^2} A^{-1} X\mathbf{y}, A^{-1}\right) \tag{12}$$

where $A = \sigma_n^{-2} X X^{\mathbf{T}} + \Sigma_w^{-1}$.

Now, what have we achieved? We're on our way to finding a good surrogate model for our objective function and we've started out with assuming that we could use linear functions to fit it. We have then used Bayes' Theorem to go from a *prior distribution* of linear functions to a *posterior distribution* of linear functions (more specifically a posterior distribution on the weights $\mathbf{w}$ but that is easily translated to a distribution of functions by using equation 1). What remains is then to figure out how to use this distribution to get an estimate for the objective function at some specific set of coordinates $\mathbf{x}$. Since we're using Bayesian statistics and have a distribution over the weights $\mathbf{w}$, the predicted value of $\mathbf{y}$ is determined by doing an integration over this distribution, multiplying the posterior $p(\mathbf{w} \mid y(X))$ by the conditional probability of some possible value of the objective function $f(\mathbf{x})$ given some vector of weights $\mathbf{w}$: $p(f(\mathbf{x})|\mathbf{w})$. This means we end up with a *distribution* for the objective function value at each point, not just a scalar value;

$$p(f(\mathbf{x}) \mid \mathbf{y}(X)) = \int p(f(\mathbf{x})|\mathbf{w})p(\mathbf{w} \mid \mathbf{y}(X))\mathrm{d}\mathbf{w} \tag{13}$$

Writing out this expression and doing the integral gives us,

$$p(f(\mathbf{x}) \mid \mathbf{y}(X)) = \mathcal{N}(\frac{1}{\sigma_n^2}\mathbf{x}^{\mathbf{T}} A^{-1} X\mathbf{y}, \mathbf{x}^{\mathbf{T}} A^{-1}\mathbf{x}) \tag{14}$$

Notice that this is simply the posterior distribution of weights, $p(\mathbf{w} \mid \mathbf{y}(X))$, where the mean has been multiplied by $\mathbf{x}^{\mathbf{T}}$ and the variance has been multiplied by $\mathbf{x}^{\mathbf{T}}$ from the left and $\mathbf{x}$ from the right (which is intuitive since the model is $f(\mathbf{x}) = \mathbf{x}^{\mathbf{T}}\mathbf{w}$).

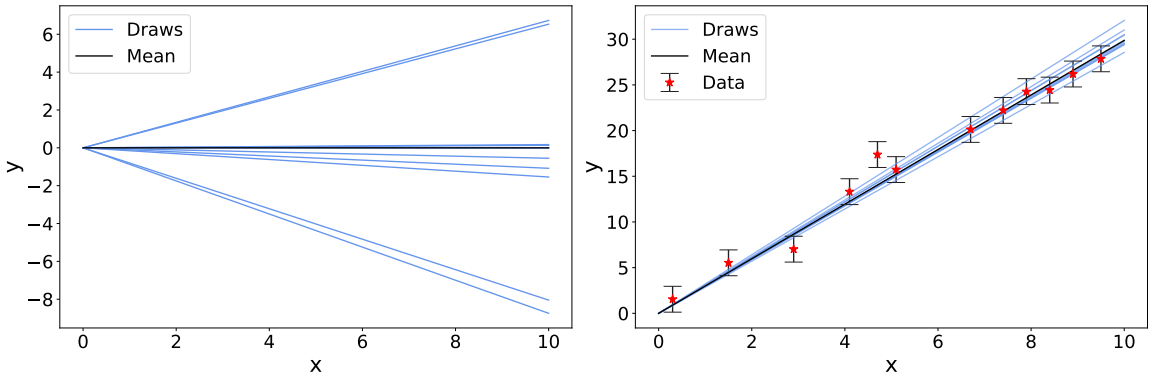


Figure 2: Ten functions drawn from the a priori distribution (left) and the posterior (right). The black line gives the mean of the distribution and in the plot on the right, data (sampled from a linear function with noise) with standard deviation is shown as well.

We have then arrived at a predictive model for our objective function $\mathbf{y}$, and since it's a distribution and not a scalar, we can look at the mean if we want a single value and at the variance for an idea of the uncertainty of the model at that point. The next step is to discard the assumption that the objective function is linear and expand the flexibility of our model.

In figure 2, ten functions have been drawn from respectively a prior distribution and a posterior distribution conditioned on a set of noisy data. Looking at the functions from the prior distribution, we can see that the distribution of weights has mean zero and a fairly large variance. In comparison we can see that the posterior weight distribution has a mean around 3 and a smaller variance because the likelihood has selected group of functions that go through the data.

2.3.2 *Using Basis Functions*

How can we expand our linear model to something that can fit any underlying objective function?

One option is to use a series of basis functions. If we choose a series that has large expressiveness so that it converges to any reasonably well-behaved, bounded function then we should be able to model any well-behaved function we might encounter reasonably well (given that the amount of basis functions we include is large enough).

To implement this we now write a new model;

$$f(\mathbf{x}) = \phi(\mathbf{x})^{\mathbf{T}}\mathbf{w} \tag{15}$$

where $\phi(\mathbf{x})$ is a function that maps the coordinates $\mathbf{x}$ to a *feature space* consisting of these new basis functions.

The attentive reader will notice that this is actually the same model as the one we used in equation 1, except we exchanged $\mathbf{x}$ with $\phi(\mathbf{x})$. This means the weights that before simply gave us the slope of our model, now tells us how much each basis function contributes to our model. It also means the dimension of the equation changed from $n_{parameters}$ to $n_{features}$where the first scalar gives the amount of parameters we are tuning and the second counts the amount of basis functions we're using.

Since the model is the same except for this substitution, we get exactly the same equations, except $\mathbf{x}$ is substituted for $\phi(\mathbf{x})$ everywhere.

Applying this we get that the predictive distribution of our surrogate model is now,

$$p(f(\mathbf{x}) \mid \mathbf{y}(X)) = \mathcal{N}(\frac{1}{\sigma_n^2}\phi(\mathbf{x})^{\mathbf{T}} A^{-1}\phi(X)\mathbf{y}, \phi(\mathbf{x})^{\mathbf{T}} A^{-1}\phi(\mathbf{x})) \tag{16}$$

where A has been similarly changed to $A = \sigma_n^{-2}\phi(\mathbf{x})\phi(\mathbf{x})^{\mathbf{T}} + \Sigma_w^{-1}$.

Introducing a new entity, $k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^{\mathbf{T}}\Sigma_w\phi(\mathbf{x}')$, we can rewrite this equation into the following form:

$$\begin{aligned} p(f(\mathbf{x}) \mid \mathbf{y}(X)) = \mathcal{N}(&k(\mathbf{x}, X)(k(X, X) + \sigma_n^2 I)^{-1}\mathbf{y}, \\ &k(\mathbf{x}, \mathbf{x}) - k(\mathbf{x}, X)(k(X, X) + \sigma_n^2 I)k(X, \mathbf{x})) \end{aligned} \tag{17}$$

In this form, we can see that the prior information (the chosen basis functions and the prior weights over them) is collected in the function $k(\mathbf{x}, \mathbf{x}')$ which we name the *kernel* or the *covariance function*.

2.3.3 *Gaussian Processes*

In the previous subsection, we expanded our linear model into a feature space with some finite amount of basis functions and thus achieved much better expressiveness and flexibility of our surrogate model. But what if we could make the model even more flexible by having an *infinite* amount of basis functions? Then we would (given that the series of basis functions converges to any reasonably well-behaved, bounded function and the objective function is indeed reasonably well-behaved and bounded) be able to fit any objective function and thus have all functions in the posterior distribution go exactly through every point in the data (or through a suitable area around the data points, if the data has noise).

The obvious problem with this, is that so far our prior information is contained in the covariance function, given by $k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^{\mathbf{T}}\Sigma_w\phi(\mathbf{x}')$ which contains two entities that are

$n_{features}$-dimensional and would thus gain infinite dimensionality, if we let the amount of basis functions go towards infinity, making the covariance function incomputable.

Unless, of course, we could write those basis functions as an infinite sum and similarly write the weights as an infinite series. This, however might be quite cumbersome to work with, and the prior information contained in these sums might be quite difficult to get a good, intuitive feeling of. But having gotten this far, we might ask: What if such a sum could be written as a simple expression with just one term? Then we'd have a simple, compact expression that for each set of points $(\mathbf{x}, \mathbf{x}')$ gave us a scalar just like the expression $\phi(\mathbf{x})^{\mathbf{T}}\Sigma_w\phi(\mathbf{x}')$ would.

To get a better understanding of what kind of information would be contained in such an expression, we turn our attention towards something entirely new for a moment, something called *stochastic processes*, specifically the kind called *Gaussian processes*.

A stochastic process is a collection of random variables where each variable follows the same statistics in some way, e.g. by being drawn from distributions of the same kind.

A Gaussian process is a stochastic process where each random variable follows a Gaussian distribution and any finite collection of these follow a multivariate Gaussian distribution. The random variables depend on the continuous coordinate $\mathbf{x}$. We write,

$$\mathcal{GP}(\mathbf{x}) = \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{18}$$

where $\mathcal{GP}(\mathbf{x})$ is a Gaussian Process, $m(\mathbf{x})$ is the mean and $k(\mathbf{x}, \mathbf{x}')$ is a covariance function. The last two quantities entirely describe the Gaussian Process. Because the variable $\mathbf{x}$ is continuous, there is an infinite number of random variables defined over some interval of it. When using a Gaussian Process in practice, however, it is evaluated over a grid and so it degenerates to a multivariate normal distribution with a finite number of random variables.

In our case such a series of random variables (and let us just keep them finite for ease of understanding) could be a series of (unknown) objective function values $\{\mathbf{y}_1, \mathbf{y}_2..\mathbf{y}_n\}$ at coordinates $\{\mathbf{x}_1, \mathbf{x}_2..\mathbf{x}_n\}$, and so a draw of the Gaussian process could give us a possible sequence of values of the objective function, given that we found a suitable mean and covariance function.

Knowing the definition of a Gaussian process we can look at the quantities $m(\mathbf{x})$ and $k(\mathbf{x}, \mathbf{x}')$. If $m(\mathbf{x})$ is described by a vector of n entries (where n is the amount of variables in our series), each entry will give the mean for the corresponding random variable, or, in our case, the mean for the objective function estimate at that coordinate. If $k(\mathbf{x}, \mathbf{x}')$ is described by a matrix of $n \times n$ entries, it will contain the covariance of each random variable and thus

describes the spread of their individual distributions along with their correlation to the other random variables.

To understand the covariance function better, we might take a look at one often used when working with Gaussian Processes; The Radial Basis Function (RBF) covariance function. It is described by the equation,

$$k(\mathbf{x}, \mathbf{x}')_{RBF} = \exp(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{l}) \tag{19}$$

where $||\mathbf{x} - \mathbf{x}'||$ is the euclidian distance between points $\mathbf{x}$ and $\mathbf{x}'$ and l is the lengthscale *hyperparameter* which we can choose the value of. More on that later.

First of all, we remark upon that this covariance function is dependent specifically on the squared difference between $\mathbf{x}$ and $\mathbf{x}'$, thus making it independent on the absolute position on the x-axis. This kind of covariance function is called a *stationary* covariance function.

Next, we notice that a squared distance of 0 between $\mathbf{x}$ and $\mathbf{x}'$ will give us a correlation of 1, whereas a large distance will have the correlation descending towards 0. This means that points close to each other will be highly correlated and thus have similar values, whereas points far from each other will be near independent. The existence of covariance functions like this one thus tells us that the draws from the Gaussian Process can be continuous.

Now that we know that the vectors drawn from the Gaussian Process can be continuous and specified with a mean and interdependence between the points, we might start to see how it could represent an estimate of our objective function. We notice though, that we, unlike before, won't be getting an analytical expression, unless we know the basis functions behind the covariance function.

But as it turns out, it is often possible to expand a given covariance function into a series of basis functions, and if that series is infinite, we call the covariance function *nondegenerate*. Specifically, *Mercer's theorem* tells us that any positive semidefinite covariance function can be expanded into a series of basis functions ([33], page 14). For a definition of a positive semidefinite kernel see [29]. Sometimes it's possible to find this expansion analytically and even if that's not possible, it may be possible to approximate the basis functions with computer algorithms [33].

The good thing about working from the perspective of Gaussian Processes is that we understand that the covariance function simply tells us the correlation between points at different locations of the model, and we don't actually need to *know* the series of basis functions in order to use a given covariance function, just that it's positive semidefinite and perhaps whether or not it's degenerate. In other words, we can set up a prior distribution of

functions simply by stating the interdependence of points, instead of defining an infinite series of basis functions. Practically, this means that instead of drawing a vector of weights for our basis functions (when we draw a function from our prior distribution) and multiplying it by our series of basis functions, we simply draw a vector from our collection of random variables and this vector then directly represents a function from our prior distribution.

In equation 17, we found the predictive distribution by explicitly using basic functions. Can we reach the same result only using Gaussian Processes?

Let us begin with the prior distribution. The covariance function of this Gaussian Process will simply be the one we have chosen to represent our prior information. We just call it $k(\mathbf{x}, \mathbf{x}')$ as we have before. The mean of the Gaussian Process can be chosen depending on our prior assumptions as well. A usual choice is to simply let it be zero.

Putting this together, our prior distribution is simply,

$$p(f(\mathbf{x})) = \mathcal{GP}(\mathbf{0}, k(\mathbf{x}, \mathbf{x}')) \approx \mathcal{N}(\mathbf{0}, k(\mathbf{x}, \mathbf{x}')) \tag{20}$$

The $\approx$ sign simply signifies that the expression on the right of it is evaluated over a grid, whereas the expression on the left is defined over a continuous variable. We use the same symbols for the mean and covariance function, even though there's a technical difference there as well; from the continuous functions to their vector/matrix representations.

How do we get to the posterior from here? Well we saw in the earlier subsection that the posterior is simply the prior times the likelihood (normalised by the marginal likelihood), and that if we have no noise, the likelihood simply *selects* the functions from the prior that go through the data. Since we already know that we will be using the method with simulations that always return the same result given some given parameter values, this is the scenario we are interested in. This means all we need to do is "reject" all the functions from the prior that do not correspond with the data.

Of course, actually doing this process would be horribly impractical, but luckily it can be done efficiently by simply conditioning the prior on the data. One might think there is a problem here: That we do not know the distribution of the data. But if our prior is correct, the data should follow that distribution. This means we have two multivariate Gaussians, both defined by the covariance function $k(\mathbf{x}, \mathbf{x}')$ and a mean of zero. One of them is given by our data and a finite Gaussian distribution, while the other is a continuous Gaussian Process, where we can choose which grid points we want to evaluate it on.

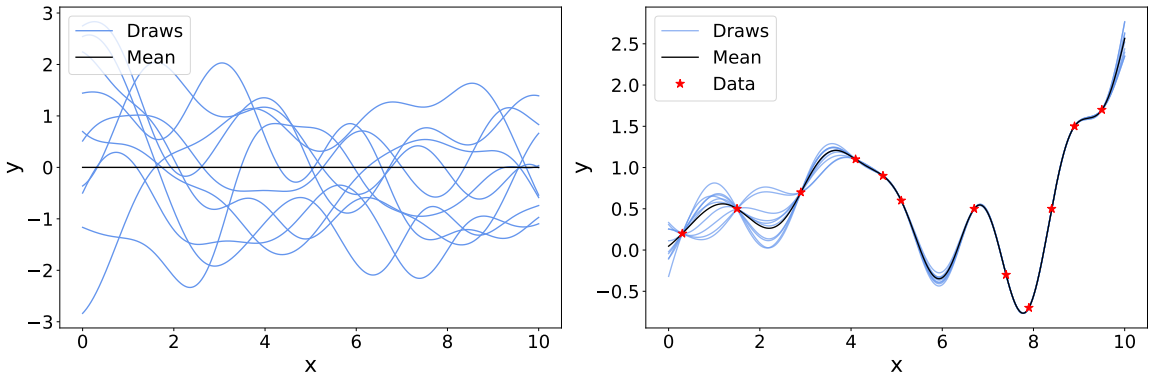


Figure 3: Ten functions drawn from a prior distribution using an RBF kernel (left) and a corresponding posterior conditioned on some data (right). The data is just randomly chosen numbers. The blue lines are functions drawn from the distributions, the black lines give the mean of the distributions and in the plot on the right, data is shown as red stars.

If we choose a grid for the Gaussian Process (thus estimating it with a Gaussian distribution) and then condition this Gaussian distribution on the Gaussian distribution of the data, we get the posterior[1],

$$p(f(\mathbf{x}) \mid \mathbf{y}(X)) = \mathcal{N}(k(\mathbf{x}, X)(k(X, X) + \sigma_n^2 I)^{-1}\mathbf{y}, \\ k(\mathbf{x}, \mathbf{x}) - k(\mathbf{x}, X)(k(X, X) + \sigma_n^2 I)k(X, \mathbf{x})) \tag{21}$$

And we see that whether we look at this problem through the lens of basis functions or Gaussian Processes we get exactly the same result! So this is completely equivalent to the basis function approach from the last subsection.

2.3.4 *Kernels*

As we saw in the previous section, the covariance function or kernel of a Gaussian Process defines its behaviour. Now the question becomes which kernels we might consider when doing our optimisation process.

In the previous section we looked at the Radial Basis Function kernel. This kernel gives a distribution of functions where every one of them is infinitely differentiable and continuous [38], and the correlation between each point is simple and easy to understand. However, the smoothness of the functions can sometimes make them rigid and prone to fluctuations between points if the length scale hyperparameter is large, because the functions have to be smooth and can't make abrupt turns to go more directly through points that are not

1 See [33] A.2 for how to condition one Gaussian distribution on another.

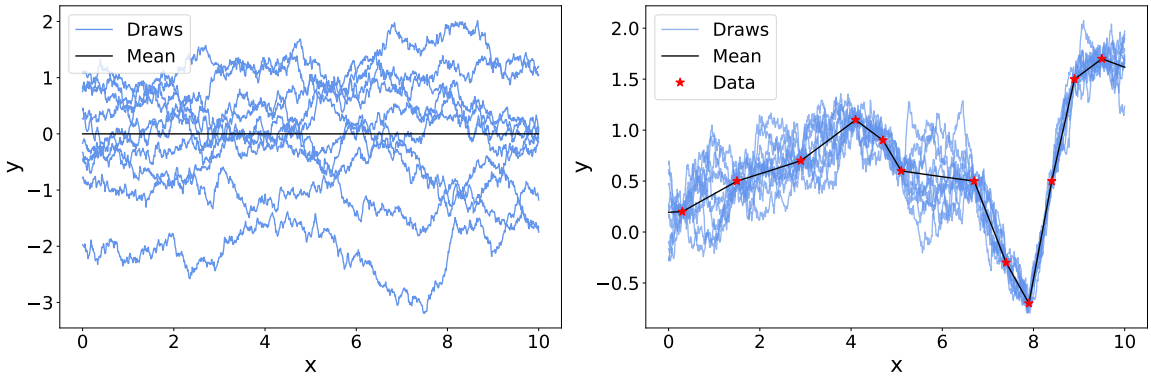


Figure 4: Ten functions drawn from a prior distribution using an absolute exponential kernel (left) and a corresponding posterior conditioned on some data (right). The blue lines are functions drawn from the distributions, the black lines give the mean of the distributions and in the plot on the right, data is shown as red stars.

directly in their current path. This might induce features in our model that are inconsistent with the data, unless, of course, the underlying objective function is known to be infinitely differentiable itself. With simulation outcomes we don't expect this to be the case.

Another option for a covariance function could then be if we removed the second power from the RBF kernel and instead did,

$$k(\mathbf{x}, \mathbf{x}')_{abs\_exp} = \exp(-\frac{||\mathbf{x} - \mathbf{x}'||}{l}) \tag{22}$$

Now this kernel turns out to make functions that are completely indifferentiable [37], look very noisy and in figure 4, we see that the posterior mean just draws straight lines through every point. We have thus gotten rid of the problem of the model "swinging out" in between points and gotten to a worse alternative: A model that here just performs linear interpolation between points and won't predict optimum for us like we want it to.

One might wonder if there's something in between these two alternatives that gives us the behaviour we've been looking for. And it turns out that there is! Its name is the *Matern* kernel and the equation is given by,

$$k(\mathbf{x}, \mathbf{x}')_{Matern} = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu} \frac{||\mathbf{x} - \mathbf{x}'||}{l} \right)^{\nu} K_{\nu} \left(\sqrt{2\nu} \frac{||\mathbf{x} - \mathbf{x}'||}{l} \right) \tag{23}$$

where $\Gamma(\cdot)$ is the gamma function and K_{ν} is the modified Bessel function of the of order ν.

This kernel turns out to be $\lceil \nu \rceil - 1$ differentiable [25] and it is plotted in figure 5 with $\nu = 2.5$, making it twice differentiable. We see that we have a continuous set of functions that go through the data nicely without "swinging out" between points like the functions generated by the RBF kernel did (compare with figure 3).

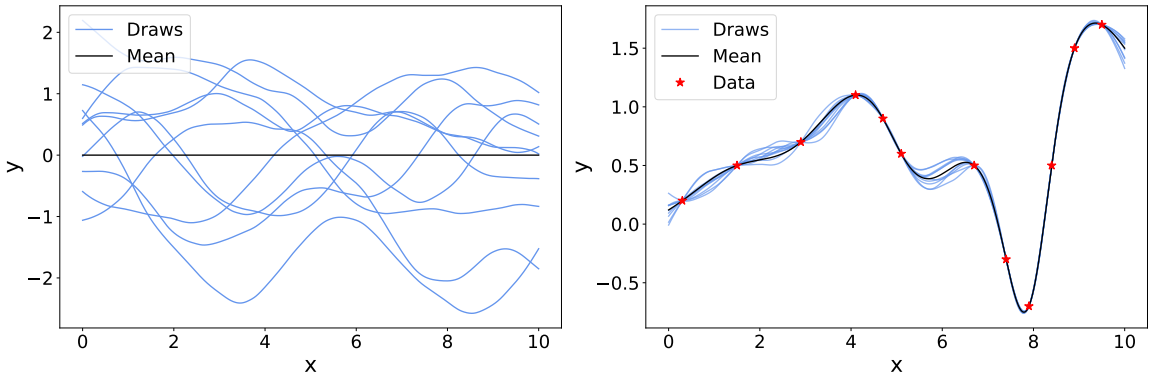


Figure 5: Ten functions drawn from a prior distribution using a Matern kernel (left) and a corresponding posterior conditioned on some data (right). The blue lines are functions drawn from the distributions, the black lines give the mean of the distributions and in the plot on the right, data is shown as red stars.

For the default set-up of our method (which won't assume any specific properties of the objective function) this might be a strong contender. It offers a simple correlation between points (at least if we utilise the information that it is something between an absolute exponential kernel and an RBF kernel), it easily fits our relatively dramatically changing little test problem without any peculiar behaviour and only has a single hyperparameter to fit.

If we have more specific information about a given problem or anticipate a very complex function structure that we don't believe the Matern kernel will be able to contain, we have many, *many* other options. There are periodic kernels, linear kernels (equivalent to the linear regression we did in chapter 2) and the rational quadratic kernel, just to mention a few (see [7] for equations for these and more). Furthermore we can *add* and *multiply* different kernels as we want, since the only requirement is that the kernel stays positive semidefinite.

For more complex kernels, we can mention the Spectral Mixture Kernel (SMK) [42] or the Deep Kernel [43]. Both of these offer great complexity and can find complicated structures in data. With the SMK we can even tune the amount of complexity by deciding how many terms in a series to include (see equation 12 in [42]). Similarly in the Deep Kernel (DK), we can design the Artificial Neural Network it incorporates as we wish, with as many layers as we want. The SMK shows good performance and beats, among others, the Matern kernel on a number of test sets that all display some kind of complex, periodic behaviour. The DK is tested on a range of large datasets (compared to the ones we expect to encounter) but is largely beaten by the SMK. It is shown to do well, however, on a step function full of discontinuities where the SMK struggles.

Clearly, there are many options (and many well-designed and intelligent options) when choosing our kernel.

Through all this though, we have to remember: We assume to work with datasets of only ~ 100 data points and when fitting for the first time we might only have ~ 20 points to work with. As such, many of these complicated kernels are likely to be overly complex unless they are chosen specifically to match the objective function (and even then over-complexity could certainly still be a problem).

Furthermore and perhaps most importantly, we should go back and remember the following: The Matern kernel (if we assume we will just use $\nu = 2.5$) only has *one* hyperparameter and it is *easily understandable*. It simply states over which distance we expect the objective function to change significantly and it can easily be fitted. As a demonstration it could be mentioned that the lengthscale in figure 5 was fitted by hand since it's a one dimensional parameter sweep and is barely worth the trouble of setting up an algorithm.

Many of the other mentioned kernels will mean that we end up with many parameters that might be internally dependant and thus introduce an additional optimisation problem into our method. Besides introducing the risk of failing this optimisation and gaining a poor model for our method, it becomes more difficult for the user to adjust the hyperparameters, should they have undesirable values.

With the Matern kernel, we can visualise the model for the user and give them the option to change the fitted lengthscale if the fit looks wrong. Furthermore we can start up the algorithm with easy-to-interpret *bounds* on the hyperparameter that prevent it from being catastrophically wrong.

Of course, if we knew we had many data points and anticipated objective functions with complex patterns, using SMK or DK would probably be preferable.

When there are multiple parameters (as there generally are), we can choose a kernel for each one or choose the same kernel for all and simply fit the hyperparameters individually to each parameter.

2.3.5 *Optimisation of kernel parameters*

All of the kernels named above have one or more parameters that we need to choose values for. Given that these parameters control our prior distribution of functions instead of being function parameters themselves, we call them *hyperparameters*.

This chapter is inspired by chapter 5 in [33].

Being in a Bayesian framework, the obvious thing to do in order to optimise their values, is to look at the probability of some potential values of them, given some data, i.e. to look at the posterior of the hyperparameters:

$$p(\theta \mid \mathbf{y}(X)) = \frac{p(\mathbf{y}(X) \mid \theta)p(\theta)}{p(\mathbf{y}(X))} \tag{24}$$

where we use θ to signify the hyperparameters, however many there might be.

Now this is a fairly complicated expression but we know that the marginal likelihood in the denominator just normalises the expression, so we can write,

$$p(\theta \mid \mathbf{y}(X)) \propto p(\mathbf{y}(X) \mid \theta)p(\theta) \tag{25}$$

Next, we might assume that the prior on the hyper parameters is either flat or just fairly broad and thus might not influence the location of our maximum. What we're left with is then just the probability $p(\mathbf{y}(X) \mid \theta)$.

We now write up Bayes' equation for the function values;

$$p(\mathbf{f}(X) \mid \mathbf{y}(X)) = \frac{p(\mathbf{y}(X) \mid \mathbf{f}(X))p(\mathbf{f}(X))}{p(\mathbf{y}(X))} \tag{26}$$

And if we consider this equation given some specific values of the hyperparameters, we get,

$$p(\mathbf{f}(X) \mid \mathbf{y}(X), \theta) = \frac{p(\mathbf{y}(X) \mid \mathbf{f}(X), \theta)p(\mathbf{f}(X), \theta)}{p(\mathbf{y}(X), \theta)} \tag{27}$$

And we get that the *marginal* likelihood in this equation is the same as the likelihood we were looking for. This is useful because seeing the likelihood in this way we can write is as,

$$p(\mathbf{y}(X) \mid \theta) = \int p(\mathbf{y}(X) \mid \mathbf{f}(X), \theta)p(\mathbf{f}(X) \mid \theta)d\mathbf{f} \tag{28}$$

and we see that we have found an equation that we can use to find the likelihood we needed.

Common practice is to take the logarithm of this expression and because of this, it is commonly known as the *marginal log likelihood* (MLL). Of course, this is a bit of a vague name as Bayesian optimisation utilises many likelihoods and any of them could be seen as *marginal* if calculated by integrating out some quantity. All the same, this is the name and it can be shown ([33], page 113) that it is equal to,

$$\log(p(\mathbf{y}(X) \mid \theta)) = -\frac{1}{2}\mathbf{y}^{\mathbf{T}}K_y^{-1}\mathbf{y} - \frac{1}{2}\log|K_y| - \frac{n}{2}\log(2\pi) \tag{29}$$

where $K_y = K_f + \sigma_n^2 I$ and K_f is $k(X, X)$, the covariance matrix for the coordinates of the data points.

This equation turns out to be very informative of the behaviour of the MLL, as each term affects the optimal choice of hyperparameters (and thus the optimal form of the surrogate model) in different ways. The first term gives a measure of how well the model follows the data, which in the case of no noise on the data is perhaps a little less intuitive than usual, since all functions in the posterior go through the data points, but the term essentially measures whether the covariance function predicts the same correlation as is given in the data vectors $\mathbf{y}$. The second term gives a punishment for complexity (or alternatively, a reward for simplicity) and the third term gives nothing really, as it's just a normalisation term and independent of θ.

The first term is one we would expect from any measure on how good our model is, but the second is very interesting. When setting up a model, we generally want to avoid unnecessary complexity that isn't represented in the data (a principle known as Occam's Razor), and with MLL we actually get this automatically!

Another favourable quality of the MLL is that it becomes more peaked when we have more data [42]. This basically shows us that the MLL converges towards a specific interpretation of the data when more data is known.

If we were to look at the RBF or Matern kernel for an example, we can look at the lengthscale hyperparameter they both have and see how it influences the MLL.

If we start with the RBF kernel (see figure 6), we can see that the data-fit term plummets after the value of ~ 1.0 (left). If we look at the right-hand side plot, where the mean of the posterior is plotted for the smallest, largest and best lengthscale value, we can see why; For the lengthscale $l = 2.0$, the mean of the posterior distribution swings wildly between data points in a way that is completely unrepresented by the data. This is, as mentioned earlier, because the RBF kernel is infinitely differentiable and with a large lengthscale, it can only get through the quickly changing points by "bending" in between. (Think of trying to force a very thick, stiff cable through a series of fastenings with varying height.)

Looking at the results from the Matern kernel instead (figure 7), we see that it doesn't have this erratic behaviour. Instead we have that the best lengthscale and the largest lengthscale almost coincide completely. We also notice that at small lengthscales the function value go towards our chosen prior mean (zero) in between points. This is simply because the functions in the prior distribution vary so rapidly that they just become noise around the mean and then "shoot up" to meet each data point. It should be noted that the RBF kernel has the same behaviour at low lengthscales. Most kernels will probably lead to this behaviour if

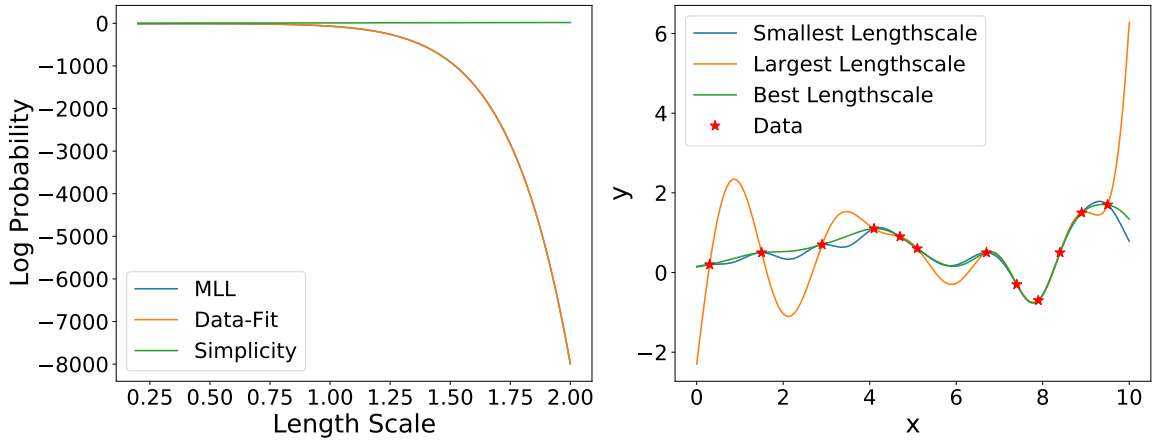


Figure 6: The MLL and its significant two terms are plotted against the value of the lengthscale hyperparameter on the left, given an RBF kernel and a simple test set. On the right the data from the simple test set is shown and we see the mean of three posterior distributions: One with the smallest lengthscale we have used, one with the largest and one with the lengthscale that gives the highest MLL.

the hyperparameters are badly fitted in a way such that the prior functions change much, much more rapidly than the data.

Furthermore, if we look at the left hand side of the plot, we can see the behaviour of the two terms of the MLL; The data-fit term still falls when we get to the higher lengthscales, but it doesn't plummet dramatically like it did for the RBF kernel. This term has a maximum around $l \approx 0.6$. The simplicity term rises as the lengthscale becomes longer. This is because the model is less complex at higher lengthscales where the variations of the model value happen over a longer distance. This means that if we look between two data points, some of the functions in our posterior distribution might predict a higher value and some of them might predict a lower value, but none of them will predict that the value goes up and down multiple times (see figure 5). To make an analogy with a well-known subject, we could think about a Taylor series. A simple Taylor series will just contain a few, low-order terms that vary simply, whereas a complex one might involve many high-order polynomials that swing up and down between the data points (if the problem is over-fitted at least). Another way of putting it is that a lower lengthscale will allow for more variation and will not be "surprised" (have a low probability) for new data points that lie outside of the current mean. For example, if we turn up the lengthscale in figure 5 to 15.0 (the one shown in the figure is 2.5), all the draws lie exactly on top of the mean, i.e. the probability of values outside of the mean falls towards 0.

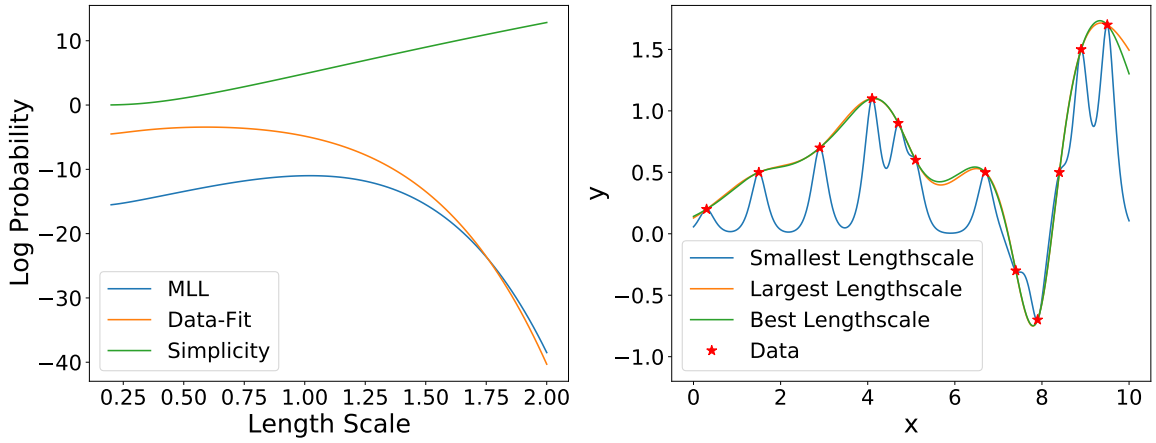


Figure 7: This plot shows the same quantities as in figure 6, except we are using a Matern kernel here.

When using a kernel with a simple hyperparameter like the lengthscale, it can be a good idea to set some bounds before letting the optimisation run. By doing this we are guaranteed not to choose a value that is dramatically low or high. At low lengthscales (before reaching the point of complete nonsense and bias towards our prior mean), we can over-estimate the uncertainty between points, because the posterior functions oscillate too much. At high lengthscales our model turns overly confident and we lose the information that is the spread in our posterior distribution of functions.

An alternative to using MLL could be to use cross validation (especially leave-one-out cross validation so we still get to utilise all of our precious, sparse data), but we won't go into this method and its pros and cons in this thesis.

As a final note, the Bayesian framework used to find an expression for the MLL could also be used to compare different models, once we have evaluated our initial points, but that has not been attempted in this thesis.

2.4 ACQUISITION FUNCTION

Once we have a good surrogate model fitting our objective function, we will need to decide how to use it for choosing candidate points. To make this decision we introduce an *acquisition function* that prioritises the two main sources of information from our model: The mean and the variance of the posterior distribution.

Naturally, we want to evaluate the objective function at coordinates where the mean of the model is large, since it then predicts that the objective function will be large there. When it comes to the variance, however, it is slightly less intuitive, because while a maximum with

low variance of course has its advantages (because we are fairly sure that it is a maximum), a place with a mean in the middle of our range but a high variance might be host to a hidden maximum. Furthermore there's an advantage to exploring areas with high variance because these are often areas where the model is lacking information and evaluating points here might further our understanding of the objective function.

A simple acquisition function then becomes,

$$a_{UCB}(\mathbf{x}) = \mu(\mathbf{x}) + \sigma^2(\mathbf{x})\beta \tag{30}$$

where μ is the mean of the model, σ^2 is the variance and β is a tunable parameter.

This acquisition function is called *Upper Confidence Bound* (UCB) because it combines the mean and variance to give a guess at an upper bound (influenced by the parameter β) of how good the objective function is expected to be at some set of coordinates. The parameter β gives a direct, linear way to set the trade-off known as *exploration vs. exploitation*, where a high β will give large exploration and thus will search places with high variance to expand our knowledge of the objective function by exploring many different regions or find maxima that weren't certainly predicted by the model yet. A low β, on the other hand, will mean that we're exploiting the regions where we already know that the objective function has a large value or where the model very certainly predicts that it does.

Another popular acquisition function is the *Expected Improvement* (EI) function that aims to calculate the expected increase in function value at any given coordinate set with the following equation [23]:

$$a_{EI}(\mathbf{x}) = \mathrm{E}[\max(f_{max} - f(\mathbf{x}), 0)] \tag{31}$$

where $E[\cdot]$ signifies the expected value and f_{max} is the maximum value of the objective function that we have found so far.

Since this acquisition function integrates over all model values at coordinate vector $\mathbf{x}$, it automatically uses the spread of the distribution as well as the mean. The advantage and disadvantage of this acquisition function then becomes the same: It doesn't have any parameters. This is great as long as it performs well, but if it doesn't we have no way to adjust its behaviour (i.e. its exploration/exploitation balance).

Furthermore, EI has the advantage that already evaluated points (without noise) will have acquisition value zero, where they with UCB will simply be a local minimum because the variance will be zero at the point but larger immediately around it. This however, leads us to another *disadvantage* with EI, which is that it will be zero in large areas and thus might be difficult to find the maximum of.

2.4.1 *Noisy Upper Confidence Bound*

During initial testing of the method, it sometimes happened that the objective function was very flat in one parameter, resulting in a nearly flat mean with a slight tilt and a constant uncertainty. In this case the optimisation could sometimes get stuck at a specific value for that parameter, because there would be a slight maximum on the right or left side of the bounds. This would keep the parameter from sampling other values and keep the optimisation from discovering any structure in the objective function along that parameter.

To mitigate this, we propose to add a little bit of Gaussian noise to the acquisition function and make that noise dependant on the size of the variance, so that more uncertain areas of the model will get more noise. The proposed acquisition function is called UCB_{Var} and has the equation,

$$a_{UCB\_Var}(\mathbf{x}) = \mu(\mathbf{x}) + \sigma^2(\mathbf{x})\beta + r\sigma^2(\mathbf{x})\beta\gamma \tag{32}$$

where r is a random number from the normal distribution $\mathcal{N}(0,1)$ and γ is a tunable parameter.

We propose a default value for γ of 0.01, so the noise only really makes a different when the model is very flat in some parameter and won't introduce too much randomness into our method.

Note that since the noise is larger when the variance is higher, this addition to UCB won't affect the optimisation when the model is confident in its predictions.

Note also that because a random perturbation is chosen every time the acquisition function is called, a powerful optimisation method like dual annealing might be able to ignore the noise and still find the point with the largest acquisition function value without taking the noise into account. Given that, it might be favourable to find a way to add noise to the acquisition function that is more constant in the parameter space, but this has not been further explored in this project.

2.4.2 *Optimisation of the Acquisition Function*

When optimising the surrogate model instead of the objective function, we have one major advantage: The surrogate model is quick to evaluate so we can evaluate it lots and lots of times.

This means we can use any standard global optimisation method we would like, as long as it performs well and finishes within some reasonable amount of time. This amount of time is

better chosen too high than too small: Getting stuck in a local maximum of the surrogate model (or worse: not finding a maximum at all) and thus choosing a bad candidate point basically means wasting an entire evaluation of the objective function and if this takes hours or even days, this is really a critical mistake. Going from e.g. 3 to 6 minutes of optimising the acquisition function, however, is completely insignificant (as long as the objective function takes much longer), so if that's the price for finding the right candidate, it's well worth it.

Of methods to use for this optimisation, we can name e.g. Dual Annealing [45] and multi-start L-BFGS-B (as implemented by the *BoTorch* Python package [2]). But as mentioned, any suitable method for global optimisation can be used.

2.4.2.1 *With Multiple Points*

If we're running multiple evaluations of the objective function at each step of the optimisation, we need to find multiple candidate points at each step as well.

This can be approached either by expanding the acquisition functions to return more points at one evaluation or by finding one point at a time and using some criterion to push the candidates away from each other.

The former is described in [44] and requires using Monte Carlo to approximate the acquisition function. It has the advantage of attempting to simply expand the criterion used to find one parameter to finding several, but we won't go further into how it works here.

The latter could be done most simply by making a hard boundary, so after finding one point, we don't allow any more inside some radius around it. The disadvantage of this is that if we were to have two peaks close to each other, we might want to have a candidate point on each one, and if we are using hard bounds, one of the peaks might get excluded.

Instead, we propose to add a punishment to the acquisition function that increases when we are close to already-chosen points. A simple choice for this punishment would be a Gaussian centred around each point we've found so far, so that the acquisition function dips down around already chosen points. A peak close to a point that is already a candidate might then still get chosen, if the punishment is not too harsh or too broad.

The scaling of the Gaussian can be chosen to be some fraction of the value of the acquisition function at that point (without any punishments) to make sure that the punishment always has the same impact, no matter the scale of the problem.

This then leaves us with two parameters: α that describes the *width* of the Gaussian and ω which describes the relative size of the dip. The equation is,

$$\text{punishment}(\mathbf{x}) = \sum_i a(\mathbf{x}) \cdot \omega \cdot \mathrm{e}^{-||x - x_{cand\_i}||^2/\alpha^2} \tag{33}$$

where $||\cdot||$ signifies the Euclidian distance, $a(\mathbf{x})$ is the value of the acquisition function, ω is a parameter, α is a parameter and x_{cand_i} is the i'th candidate point that has already been found. This punishment is subtracted from the acquisition function during optimisation.

Having the parameters α and ω is both an advantage and a disadvantage compared to using the expansion approach, because we get two simple parameters to control the spread of the points, but we then have to choose some kind of values for them. We don't expect, however, that they're overly sensitive or vary too much with each individual problem.

Another advantage over the expansion approach is the simplicity. Understanding exactly how the expanded version of e.g. UCB works isn't necessarily very easy or intuitive. But simply adding a punishment to push the points apart is simple and can be easily adjusted by the user while running an optimisation.

2.5 INITIAL EVALUATIONS

Before we can use our model to make educated guesses about where the highest objective function values might be, we need to give it some data to fit its hyperparameters. This means that we have to choose some *initial* candidate points that fill the parameter space in some way.

So how do we choose these points? One simple idea is to do a so-called *grid search* where we uniformly search the parameter space in a grid structure. This gives us the advantage of sampling the parameter space with equal spacing. The great disadvantage, however, is that the grid structure means that we use the same value of a given parameter multiple times and thus end up with fewer individual values in each dimension than if we chose a structure where none of the points are overlapping when projected to the axis of a specific parameter.

Another option then, is the *random search* where the parameter values are simply sampled from a uniform random distribution along every axis. In this method we are not guaranteed equal spacing between points but with a high amount of evaluations we do get a uniform sampling of the parameter space. On the other hand we have now dispensed with the overlapping structure of the grid search, so we now get n_{init} points on each parameter axis. The two methods are illustrated in figure 8.

In this project, we are assuming generally to have only a small number of evaluations so the fact that the random sampling becomes uniform at high amounts of points might not mean much to us. In fact, when sampling a small amount of points we might generally expect the points to "clump" together (as in figure 8) and thus give us parts of the parameter space that are well represented and parts that are not. This leads us to asking whether there might be a method that ensures good *space-filling* as it's called and simultaneously avoids the overlap issue of the grid search.

In the method called *Latin Hypercube Sampling* (LHS) we divide each axis of the parameter space into equally large parts and make sure that none of the points we sample coincide within the same divisions along each axis. This means that we are guaranteed a large degree of space-filling within each separate dimension, but we might still see some amount of clumping along linear combinations of the axes ([18], chapter 4).

Alternatively we can consider a method called minimax (or the similar one called maximin), where the points are spread in a more deterministic fashion to get the minimum maximum distance (or in maximin, the maximum minimum distance). Either metric spreads the points out through the search space and avoids "clumping" in any direction.

Either of these could possibly provide a good option, but while space-filling and uniformity are both intuitively good, sensible goals for the initial design, there is one thing we have yet to consider: What kind of sampling does the model need to fit its hyperparameters?

And here, we might immediately see a problem with space-filling designs: High-frequency variations might be missed, if *all* of the points are spread far apart, thus making e.g. the lengthscale hyperparameter of Matern or RBF models longer than it should be when it is fitted.

In [47] it is shown through six experiments with problems of increasing dimensionality (one in the first and six in the last) with 30 different lengthscales to be fitted in each experiment and 1000 realisations of space design in each, that random space design indeed performs better with a log(MSE) metric (comparing the fitted model to the data) than minimax, but at the same level as LHS (which probably makes sense as LHS still allows some degree of "clumpiness", i.e. small distances between points). To investigate which distributions of pairwise distances performed well in the random designs, they took out the 50 best performing random designs and looked at the distributions of pairwise distances in these. They then proposed a design fitted to that distribution called *beta* and finally a hybrid design that combines LHS and beta called *lhsbeta*. The latter design especially performs well and

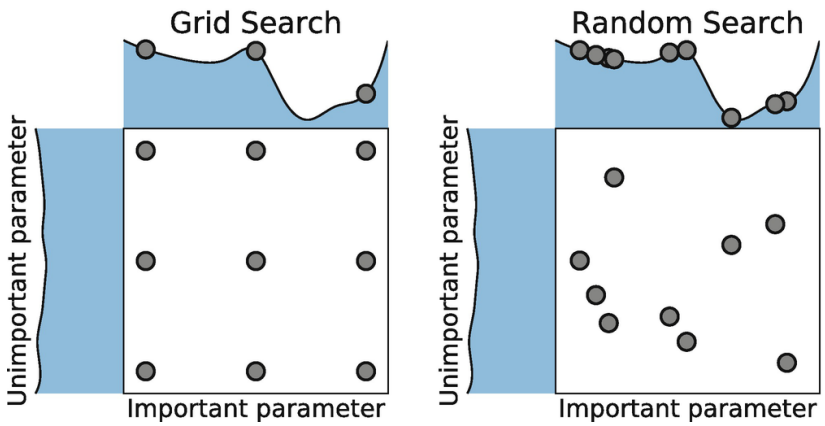


Figure 8: Grid search (left) compared to random search (right) for some example objective. We see that random search gives us more point on any one parameter axis. Figure from [11].

outperforms all the above-mentioned designs in nearly all experiments with the exception that beta sometimes does a little bit better.

Given that these results hold up across other use cases, it might be advisable to use their design for initialisation. Otherwise doing random sampling or LHS might be the best alternatives. Minimax should only be used if the lengthscale is known to be long along all parameter axes.

2.6 PRIOR INFORMATION

If we have prior information about which parameter values are more likely to yield the global maximum of the objective function, we want to use this information.

One simple solution would be to use it when finding our initial points, by sampling from this prior distribution over the parameter values, instead of e.g. sampling from a uniform random distribution as mentioned above. Then our initial points would have a greater chance of striking high objective function values and thus leading the model quickly to desirable regions of the parameter space.

A disadvantage of doing this, is that the regions with low probability in the prior would not be sampled and so our model would attribute them with high uncertainty, which means our acquisition function would likely have a high value there, causing our method to investigate by choosing candidate points there.

A solution to this was proposed in [32] where the pairwise distance in the kernel $||x - x''||^2$ is replaced by,

$$||\phi_m(x_m) - \phi_m(x'_m)||^2 \tag{34}$$

where ϕ_m is the cumulative distribution function (cdf) of the prior.

This method can be used with any kernel that uses a pairwise distance (i.e. any stationary kernel) and works by warping the space of the model so that the regions of the parameter space with high prior likelihood are expanded to fill up more of the space, while low-likelihood areas are shrunk to fill less, in such a way that the prior distribution is flat in the warped space. If the prior distribution accurately describes the likeliness of finding the global maximum, that likeliness will be flat in this warped space as well.

In the article, the method generally outperforms a BO method without prior information and a method that just samples from the prior distribution (without using BO), but it is not run against a BO method initialised with the prior.

A disadvantage of the method might be that the warping of the parameter space could transform a well-behaved objective function into something less fit-able. It might also be worthwhile to note that the acquisition function cannot immediately use the same transformation since it doesn't use pairwise distance, and will thus be operating in a different space than the model, and might then still over-sample the uninteresting regions.

An important note is also that if the prior is wrong and the global optimum is outside of the regions that have a high prior likelihood, it will be very difficult to find with this method, more so than if we just use the prior for sampling the initial points.

A possible alternative might be to simply punish the acquisition function for suggesting candidate points with low prior likeliness, perhaps by

$$a_{UCB\_prior}(\mathbf{x}) = \mu(\mathbf{x}) + \sigma^2(\mathbf{x})\beta - (1 - p_{prior}(\mathbf{x}))\tau\phi \tag{35}$$

where $\tau = \text{abs}(\mu(\mathbf{x}) + \sigma^2(\mathbf{x})\beta)$ and ϕ is some arbitrary constant with range $\phi \in [0, 1]$ that toggles how hard the punishment is. We don't have time to explore this acquisition function further in this thesis, but mention it in case it sparks inspiration for a future reader.

2.7 DIFFERENCE MEASURE

In our simulations, our objective function will generally be measuring the difference between some output of the simulation and some data that we have acquired.

Choosing the metric for this difference in a reasonable way is of great importance, since it greatly influences the shape of the objective function. Of course, the objective function will also be shaped by how the chosen output depends on the parameter values, and this relation we don't know. If we did we wouldn't need to run the optimisation at all.

The relationship between the objective function, the difference measure and the simulation can roughly be written as,

$$f_{obj} = f_{diff\_met}(\mathbf{y}(X), f_{sim}(p)) \tag{36}$$

where p signifies the parameters, f_{obj} is the objective function, f_{diff_met} is the difference measure and $f_{sim}(p)$ is the simulation output as a function of the parameters.

Given all this, we need to make a decision about the difference metric we use, and thus we need to make an assumption about $f_{sim}(p)$. The simplest thing we can do is to assume that it's linear, and if we only need to consider a small area - such as the section of the parameter space close to the global maximum of the objective function - it might be an alright assumption. If the assumptions holds in that area we will know what our global maximum looks like and thus how well our model will be able to fit it.

The simplest metric we can imagine is probably the squared difference (with a negative sign, since we're maximising the objective function). This function has the advantage that it makes a nice, smooth peak that a kernel like Matern or RBF will have an easy time fitting. The disadvantage is that the difference is squared so the optimisation will be rewarded greatly by getting to decent solutions with a few correct digits but won't be rewarded greatly by getting a large precision on the match between the chosen simulation output and the data.

The quadratic difference measure is simply,

$$f_{obj} = -(\text{output} - \text{target})^2 \tag{37}$$

This isn't necessarily a problem; It all depends on the problem at hand. If we are optimising a large amount of objectives or just one very difficult objective, this might be exactly what we want. We will find the acceptable areas of the parameter space for an objective and then be focused on getting the other objectives correct as well.

If, however, we are optimising an output where we would ideally like to get many digits correct, we will need to choose a different metric.

The least dramatic thing to do would be to simply drop the second power and use the absolute value of the difference instead. The drawback of this is that the peak of our objective

will no longer be a smooth parabola but a triangle with a non-differentiable point at the top. This might prove more challenging for some kernels. It will also still not encourage a very large precision in the output.

The absolute difference measure is given by,

$$f_{obj} = -\text{abs}(\text{output} - \text{target}) \tag{38}$$

A more drastic solution would then be to take the logarithm of the quadratic difference measure, making our objective function increase linearly when the error falls with an order of magnitude. The issue with this is that our objective might then get a very narrow peak that might be challenging for some kernels to fit, since the objective function values will essentially be diverging as the error approaches zero.

The logarithmic difference measure is given by,

$$f_{obj} = -log((\text{output} - \text{target})^2) \tag{39}$$

2.8 MULTIPLE OBJECTIVES

If we have more than one objective, we need to consider how we fit the data points from them all and how we choose our candidate points.

The simplest thing to do would be to combine the different objectives with a weighted sum. Then we'd only need to fit one model, as we did before, and we'd be able to use the same acquisition functions we did before.

Now, the immediate disadvantage of that is that we are bound to lose information by collapsing the n_{objs} different objective functions into one. So fitting one model to a weighted sum is bound to give us less information than fitting n_{objs} models to n_{objs} objective functions. Given our general situation, where we have little information and lots of time to take advantage of it, we can immediately see that we don't want to do this.

So we fit one model to each one of our objective functions and thus are left with the question of how to use our acquisition function, which is designed to consider just one input variable.

Once again, the simplest thing to do is to simply add the objective function estimates from the models in a weighted sum and use our acquisition functions as we would with one objective. To explore why this isn't necessarily a good idea we have to take a step back and ask ourselves the more fundamental questions: If we used a weighted sum, which weights would we use? If we have more than one objective, and a given vector of parameter values

gives us, not a scalar objective function value, but a vector of objective function values, which data points do we think of as good? If all the objectives were perfectly correlated and always increased in the same parameter regions, this would be easy, but that isn't necessarily the case.

The overall answer to these questions is simple: It's subjective. Or rather, it depends on the user and what their priorities are, for whatever reasons they might have.

To get a deeper answer, we need to introduce some new concepts.

First off, let's represent the vector of objective function values as,

$$\mathbf{f(x)} = [f_1(\mathbf{x}), f_2(\mathbf{x})...f_{n_{objs}}(\mathbf{x})] \tag{40}$$

Now, if this vector only has one element, it's easy to define one being superior to another.

If the vector has more elements, things become muddier, because of the above-mentioned circumstance: Choosing among these points might need a prioritisation by the user.

If, however, the two vectors we're comparing obey the condition that *every* element of the first one is larger than the respective element in the other, we can say without a doubt, that the first one is better. More specifically, we say that the first vector *dominates* the other one, or that,

$$\mathbf{v} \succ \mathbf{u} \tag{41}$$

if,

$$v_i > u_i \ \ \forall \ \ i \in \{1, ..., n_{objs}\} \tag{42}$$

If the elements are instead larger than or equal to ($\geq$ instead of $>$ in the definition above) we sat that $\mathbf{v}$ *weakly* dominates $\mathbf{u}$ and we write $\mathbf{v} \succeq \mathbf{u}$.

Given this definition we can now define a non-dominated solution, as one that is not dominated by any other solution, and then we can define the *set of solutions* that are non-dominated. We call this set the *Pareto-optimal set* and the collection of objective function values given by it is called the *Pareto front*.

We have then arrived at a possible goal for multi-objective optimisation: Finding the Pareto front.

If we know the Pareto front, we know every solution that we *might* be interested in. The optimiser can then return these solutions to the user, who can choose which one they want.

Of course, since we have a highly limited amount of evaluations, we don't actually expect to discover the full shape of the Pareto front. That is, we don't expect enough points to sample it abundantly, but we can still try to sample it as diversely as possible and try to keep

finding points that dominate the ones we already know, getting closer and closer to the true Pareto front.

The question then becomes how to sample the Pareto-front efficiently and without bias. And now we can return to the discussion of the weighted sum approach!

What we want when using the weighted sum approach, is to sample the points on the Pareto front that has the specific trade-off given by the weights between the different objectives. In practice, however, we find that different sets of weights can lead to the same points and thus the method gives us a non-uniform sampling of the Pareto front [46]. Furthermore, it only works well with convex Pareto fronts [46]. To give some quick intuition as to why that is, we can think of an example where we have a mostly convex Pareto Front, let's say in two dimensions, and we then a little "dent" (see figure 9) and the Pareto-optimal point that gives us exactly the trade-off we've requested with our weights happens to lie in this dent. Let's say that we have weights $(0.5, 0.5)$. If point A has objective values $(5.0, 3.0)$, point B has values $(3.0, 5.0)$ and C has values $(3.5, 3.5)$, then C is not dominated by either A or B (it is Pareto-optimal) and it gives us the trade-off we had requested. By using weighted-sum, however, we would arrive at point A or B because the weighted sum of these points is 8, while the weighted sum of point C is 7. Now, this might not be such a big problem. Perhaps the 0.5 value difference in one objective is not particularly important to the user and they'd rather have the sum be that much higher. But consider another scenario, where we have found a point where one objective gets a very large value, say 8.0 and the other objective at that same point is only at 0.5 and that we have another point where we have the values (4.0, 3.5). The weighted sum method prioritises the first point but (depending on the case and the user!) it might be more desirable to have two decent values rather than one high and one low, and the weighted sum method might simply not find the (4.0, 3.5) point for the user to choose.

An alternative to the weighted sum method is using the *hypervolume indicator*.

The hypervolume indicator aims to give a measure of the size of the known Pareto front. In two dimensions this would simply be the area contained by all Pareto-optimal points and some chosen reference point, as shown in figure 10. This reference point is usually chosen to be the nadir point [4], which is defined as the worst value for each objective amongst the objective values in the dominating objective vectors.

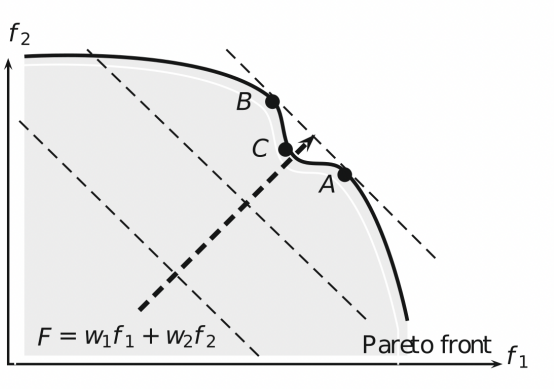
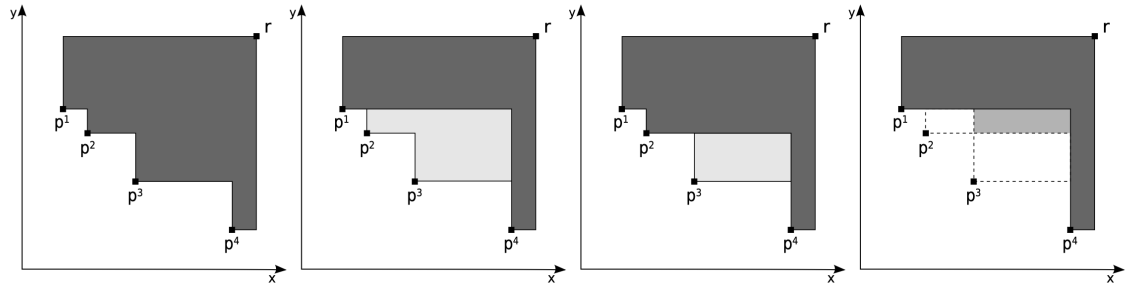


Figure 9: Example of a non-convex Pareto front. Figure taken from [46]. (The axes were inverted since we try to find a maximum, not a minimum in this thesis.)



(a) $H(\{p^1, \ldots, p^4\})$ (b) $H(\{p^2, p^3\}, \{p^1, p^4\})$ (c) $H(p^3, \{p^1, p^2, p^4\})$ (d) $H(p^2, p^3, \{p^1, p^4\})$

Figure 10: Illustration of the hypervolume indicator in two dimensions. We see how different points contribute to the overall area covered between the Pareto front and the reference point. Figure from [21].

Generally, the hypervolume indicator can be defined as [21],

$$H(S) = \Lambda(\bigcup_{\substack{p \in S \\ p \geq r}} [p, r]) \tag{43}$$

where $\Lambda()$ is the Lebesque measure and $[p, r] = \{q \in \mathbb{R}^d | \; p \geq q \textbf{ and } q \geq r\}$ is the (hyper)box contained by the points p and r. d is the amount of objectives. With this definition, we see the hypervolume, as we described above, as the union of all hyperboxes contained by the reference point r and the dominating objective vectors. Other definitions are possible, see [21].

With the hypervolume indicator we get a measure for MOO that tells us the diversity and spread of our solutions and grows monotonously as we get closer to the true Pareto front. It does not require us to state the priorities of our objectives beforehand and thus doesn't depend heavily on such a choice, like the weighted sum method would.

In [10] Emmerich, Giannakoglou and Naujoks proposed a version of the Expected Improvement acquisition function that utilised the hypervolume indicator, now known as the Expected Hypervolume Improvement (EHVI). This method essentially replaces the expected scalar improvement of a single objective in the original EI with the expected improvement in the hypervolume indicator. As such, it integrates the existing methodology we have described above with a sensible indicator of our improvement across multiple objectives and makes a good candidate for our method.

2.8.1 *Normalisation in MOO*

When we are working with multiple objectives, we suddenly see that an element that previously did not impact our algorithm much, becomes incredibly influential: The normalisation.

Whether we are using a simple weighted sum or a clever approach like the hypervolume indicator, the normalisation is essential because otherwise, different scaling of our objectives could mean that one of them completely overshadows the others. Of course, if we know beforehand that our objectives are in the same scale, we can forego the normalisation.

This isn't necessarily a given though and if we have, e.g. a difference of a factor 10^3 between objectives, the result is catastrophic in that we only really consider one objective's improvement when looking at candidate points.

One simple option is to normalise after the initial evaluation so that all objectives are transformed to have mean 0 and standard deviation 1. Alternatively, one could transform the objectives to be within the interval $[0, 1]$ or perhaps $[-1, 0]$ if one wants to converge towards 0.

The question then becomes whether or not to renormalise after that initial normalisation. If we don't, we get that an objective will be prioritised more if it is easier with our method to make improvement past the initial evaluations. This isn't necessarily undesirable behaviour. An argument in the opposite direction could be that we, with repeated renormalisation, make sure that all the objectives are always equally weighted.

3

OCEAN THEORY

> We neither fear complexity nor embrace it for its own sake, but rather face it with the faith that simplicity and understanding are within reach.
>
> *Fred Adler*

Across the Atlantic Ocean, an immense amount of water is continuously transported, running all the way from the Southern Ocean to up above Greenland. This circulation, named The Atlantic Meridional Overturning Circulation, carries incredible amounts of heat (up to around 1 PW in the North Atlantic [24]) and is of vital importance to the climate of the Earth.

In a given model of the world's oceans, it is therefore always an important part and must be reasonably faithful to the real-world version for the rest of the model to be trustworthy. For this reason, it makes a good target for tuning an ocean simulation.

3.1 THE ATLANTIC MERIDIONAL OVERTURNING CIRCULATION

The temperature and salinity of the water changes throughout all the world's oceans. They are the key players in determining the water's density [35] and thus have a large impact on the ocean's currents.

At the equator, the temperature is high and the amount of water evaporating exceeds the amount of water added to the ocean through precipitation. This means that both the temperature and the salinity are high. Note, however, that salinity and temperature affect the density in opposite ways. A simple formula for the change in density is [39],

$$\rho = \rho_0(1 - \alpha\Delta T + \beta\Delta S), \tag{44}$$

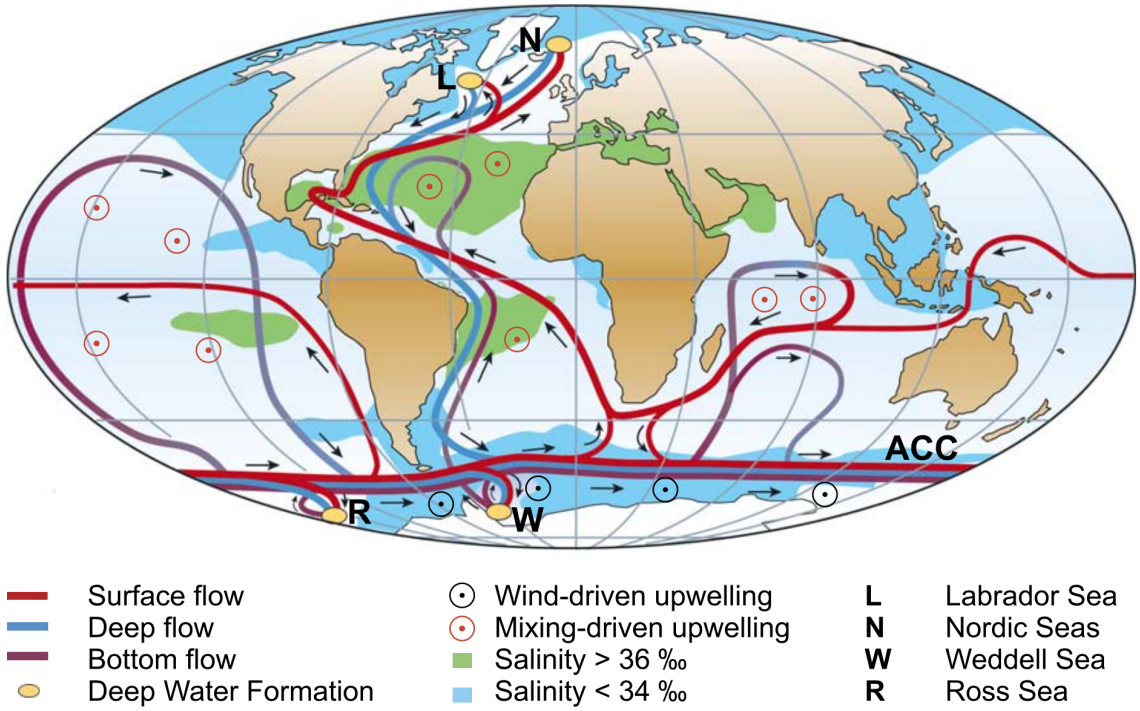


Figure 11: The currents of the AMOC (middle), the Antarctic Circumpolar Current (bottom) and currents through the Pacific Ocean (left and right). Figure from [24].

where ρ is the density, ΔT is a change in temperature, ΔS is a change in salinity, ρ_0 is some reference density, α is the thermal expansion coefficient and β is the saline expansion coefficient.

This warm, salty water from the equator then flows northward. Here, the air is cold and the water quickly loses its heat by exchange with it. Since it is saltier than the surrounding waters, its density becomes larger as it cools down, and it sinks down. Furthermore, the forming of sea ice in this region rejects the salt of the seawater, releasing it into the surroundings. This also results in water that is saltier and thus denser than the surrounding water and it sinks as well. This process is known as deep water formation.

Once the water has sunken, it moves the opposite way, running southward but now at a low depth. For the circulation to be sustainable, this water must come up again at equal volume to what sank down, but while the water sinking at high latitudes has gravity on its side, gets pulled down and loses potential energy in the process, the deep water is cold and salty and will require external energy to get back up.

The origin of this external energy and thus the driving force of the upwelling required is still a topic of discussion (see [24] for a thorough review) and one riddled with many assumptions and uncertainties, but there are two main candidates; Wind-driven upwelling and diapycnal mixing.

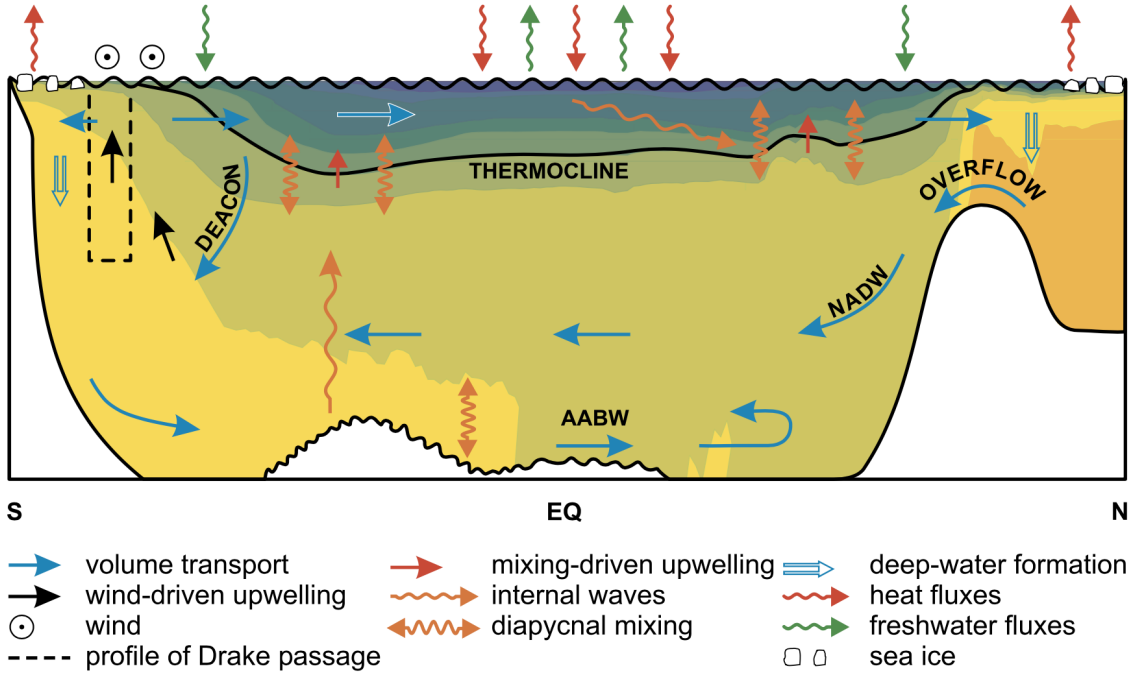


Figure 12: A vertical schematic of the volume transport and different upwelling processes of the AMOC. Figure from [24].

Diapycnal mixing has historically been thought to be the main cause of the upwelling [26], but more recent data and estimates have made this seem less likely. Instead, Kuhlbrodt [24] argues that it is likely a mix of the two.

In figure 11, the expanse of the AMOC is shown (along with the Antarctic Circumpolar Current and currents through the Pacific Ocean). We also see the locations of Deep Water Formation and the different kinds of upwelling.

3.1.1 *Wind-driven Upwelling*

In the Southern Ocean, strong westerly winds drag against the water surface and force it westward. Because of the Earth's rotation, however, the result is a northward transport of water. This is called Ekman transport [9].

The wind stress has a maximal strength around 50°S, so the northward transport is consequently also at its maximum at that latitude. This means that we get an imbalance with the amount of water being moved away from the zonal band at 50°S and the amount of water moving into it. Consequently waters at lower depths must move up to compensate for the water loss. Similarly there is a surplus of water north of this band and waters there must downwell.

This phenomenon is called Ekman upwelling and Ekman downwelling. It occurs many places in the world's oceans, including coastal regions and at the equator where the trade winds come in from opposite sides, due to the opposite sign on the coriolis force in the northern and southern hemisphere. The Ekman upwelling in the Southern Ocean, however, is especially significant for the AMOC.

The Southern Ocean is special because there, unlike anywhere else on Earth, are no barriers for a zonal flow, all the way down to a depth of 2500 meters [24]. This means that the pressure gradient becomes zero, which again means that the zonally averaged geostrophic velocity has to vanish (a geostrophic current is one where the coriolis force is balanced by the force from the pressure gradient). Thus we can only have ageostrophic meriodionial currents and the strong westerly winds mentioned above can directly force a northward current.

The Ekman upwelling and downwelling resulting from this wind forcing then creates very steep density slopes, which make for a uniquely efficient way for the deep waters to resurface [26].

In figure 12 the AMOC is shown in a vertical plot where the zonal coordinate has been integrated away. We see the different locations of the different kinds of upwelling and internal mixing and see the vertical dependency of the volume transport.

3.2 EDDIES AND MIXING

The other significant way for the deep water to upwell is through diapycnal mixing.

Most of the ocean is heavily *stratified*, which means that the water is divided into different layers with different properties, most importantly, different density. The planes of equal density are called isopycnals and the stratification makes it hard for water to mix across these.

It doesn't, however, make it impossible.

All across the ocean, there are circular currents of water called *eddies*. These range in size from centimetres up to hundreds of kilometres and are responsible for large amounts of transport and mixing in the ocean.

Given this, resolving their movement becomes very important in our ocean models, but unfortunately a significant part of them have smaller scales than that represented in our models. This means that if nothing is done, their effect on the evolution of various tracers will be missing and the upwelling of the AMOC will, for example, be too small.

3.2.1 *Parameterising the Isopycnal Mixing*

The following two subsections were inspired by [1].

To mitigate that unfortunate effect of our coarse resolution, we make a number of parameterisations to mimic the effect of the eddies.

The simplest way to do this is to just assume that the mixing that would have been done by the eddies can be approximated by standard diffusion of particles.

For a simple differential equation, describing the temporal evolution of some tracer (such as temperature, salinity, etc.), we then just get the convection-diffusion equation [5];

$$\frac{\partial \tau}{\partial t} = \nabla \cdot (\kappa \nabla \tau) - \nabla \cdot (\mathbf{u}\tau) + S \tag{45}$$

where τ is a tracer, κ is the diffusivity constant, $\mathbf{u}$ is the velocity field of the water and S contains any sources and sinks there might be for the tracer.

The first term $\nabla \cdot (\kappa \nabla \tau)$ gives us the diffusion of the tracer. The strength of the diffusion is controlled by the parameter κ. The term $-\nabla \cdot (\mathbf{u}\tau)$ gives us the advection of the tracer, as it's moved by the water currents. As mentioned, S contains the sources and sinks, so if e.g. heat is being generated by absorption of heat from the sun or salt is being added by brine rejection, these processes will be represented through this term.

Now, this simple model assumes that the diffusion is equally strong in all directions, but we already discussed how that *isn't* the case because of the stratification of the ocean.

This means that we can improve the model simply by splitting up the diffusion term, so that we can have a different diffusivity constant along isopycnals and across. Updating the equation like that, we get:

$$\frac{\partial \tau}{\partial t} = \nabla_\rho \cdot (\kappa_{iso} \nabla_\rho \tau) + \nabla_v \cdot (\kappa_v \nabla_v \tau) - \nabla \cdot (\mathbf{u}\tau) + S \tag{46}$$

where κ_{iso} is the diffusivity constant along isopycnals, ∇_ρ is the gradient along isopycnals (constant density, ρ), κ_v is the *vertical* diffusivity and ∇_v is the vertical gradient. The attentive reader will notice that while we proposed to split the diffusion into a component along isopycnals and one across, we have made one along the isopycnals and one that is simply vertical. Since most isopycnals will be approximately horizontal this should be a good approximation. The reason for it is very simply that this is how it's done in the literature used by the *veros* package that this thesis uses.

Notice that the diffusion term simplifies to $\kappa \nabla^2 \tau$ if the diffusivity κ is independent of location. See [24], chapter 3 for why that probably isn't a very good assumption, at least not in the vertical direction.

3.2.2 *Parameterising the Flattening of Isopycnals*

Another problem with not resolving sub-grid eddies, is that they work to flatten isopycnals [13]. This means that without these eddies, the isopycnals will just grow steeper and steeper over time and since a flat (horizontal) isopycnal corresponds to the lowest possible potential energy [14], this would mean that we're losing the conservation of energy or, put another way, that our system stops being adiabatic.

To mitigate this issue, Gent and McWilliams created a parameterisation in 1990 [13] that is now known as the GM parameterisation.

The GM parameterisation adds terms to our simple tracer equation (see their article [13], equation 24), and becomes simple when the constant from this parameterisation κ_{GM} is the same as κ_{iso} [20]. It can be further simplified when the isolines of the tracer are parallel to the isopycnals (see [19], pages 282-283), and then the term we're adding is simply,

$$\frac{\partial \tau}{\partial t} = -\nabla \cdot (\kappa_{GM} \mathbf{S}^2 \frac{\partial}{\partial z} \tau) + ... \tag{47}$$

where we have introduced an expression for the isopycnal steepness,

$$\mathbf{S} = -\frac{\nabla_h \rho}{\frac{\partial \rho}{\partial z}} \tag{48}$$

where ∇_h is the horizontal gradient.

Putting all this together our simple tracer equation becomes,

$$\frac{\partial \tau}{\partial t} = -\nabla \cdot (\kappa_{GM} \mathbf{S}^2 \frac{\partial}{\partial z} \tau) + \nabla_\rho \cdot (\kappa_{iso} \nabla_\rho \tau) + \nabla_v \cdot (\kappa_v \nabla_v \tau) - \nabla \cdot (\mathbf{u}\tau) + S \tag{49}$$

3.2.3 *Spatially Dependant Vertical Mixing (the TKE Closure)*

As mentioned above, the vertical diffusion likely depends significantly on spatial coordinates (see [24]), and so setting κ_v to a constant value would likely induce a relatively large error in our simulation. Because of this, we would like to have an estimate of the vertical diffusivity at a given spatial coordinate.

In 1989, Bougeault and Lacarrere made a parameterisation like that for the atmospheric case [3] and in 1990 Gaspar et al. adapted it for the oceanic case [12]. This parameterisation uses something called the *turbulence kinetic energy* and for that reason the method is widely known as TKE.

The turbulence kinetic energy is defined as [41],

$$k = \frac{1}{2}\left(\overline{(u')^2} + \overline{(v')^2} + \overline{(w')^2}\right) \tag{50}$$

where u, v and w are the three components of the velocity field of the water. The apostrophe denotes the difference between the instantaneous and average velocities, so that $u = u - \overline{u}$, and the overline denotes an average, so that $\overline{u} = \frac{1}{T}\int_0^T u(t)\,dt$.

This entity gives a measure of how much turbulent kinetic energy is available for the mixing.

The parameterisation (or closure) proposed by the article is then,

$$\kappa_v = \frac{c_k l_k \bar{k}^{1/2}}{P_{rt}} \tag{51}$$

where c_k is a constant, l_k is a mixing length scale and P_{rt} is the turbulent Prandl number (a dimensionless number defined as the ratio of momentum diffusivity to thermal diffusivity [30]). Notice that the turbulence kinetic energy is averaged and raised to $1/2$ (i.e., the overline is not a negative sign of the power).

We note that there is also a proposed closure to make κ_{GM} spatially varying (see [8]), but we won't go through that here.

4

THE CODE

> Design is not just what it looks like and feels like. Design is how it works.
>
> *Steve Jobs*

This project was written with the aim of providing Team Ocean with a tool for doing parameter tuning in as broad a range of simulations as possible. As such, making the code flexible, modular and easy to interact with has been an important goal throughout its development.

The code is written in pure python and has been made available for download on the Python Package Index as 'veropt'.

4.1 DESIGN STRATEGY

The most important goal for this tool is to demand as *little* prerequisite knowledge of Bayesian Optimisation of the user as possible, and still deliver great results.

This requires two things: To have a range of default choices that perform as well on the "average ocean simulation" (a difficult definition to be sure, but a necessary one) as possible and to make the software as easy to interact with as possible.

Next, we have a goal to make the code as modular and flexible as possible, so that *if* the user knows what they want, they can get it. This means that if the default kernel is Matern, the user can choose to exchange it for an RBF kernel or a Deep kernel. It means that if the user wants a different acquisition function they can choose one of the ones supplied or define a brand new one as they please.

Because ocean simulations can take several days to run, getting to 50-100 iterations can take quite a long time, and so it was also deemed necessary to design a graphical user

interface (GUI) that can be used to inspect the optimisation run and make sure it's still making progress and that the model is fitting the objective function correctly.

This GUI and the visualisation tools embedded in it are also a great help when intuitively inspecting the performance of different methods on test functions and spotting potential bottlenecks that can then be rectified.

The GUI also works as a view into the process behind the optimisation; Instead of a black box that does *something* and then spits out a number of candidate points, the model and acquisition functions are visible and can be adjusted during an optimisation run.

4.1.1 *Overall Structure*

The method we're using is a composite of many submethods. This gives us a tree-like structure which is reflected in the code and some of the most important parts (as implemented in the code) are shown in figure 13. At the top we just have *Optimiser* which generally signifies the overall method and in the code specifically is the top-most python class which contains all the others and a number of tools, e.g. some for visualisation or one for finding the Pareto-optimal points. At the next, lower level we have *Model*, *Objective Function* and *Acquisition Function*.

Model contains the Gaussian Progress regression we're using to create a surrogate model and the method for optimising it. Underneath the model we have a box with *Kernel* to signify the chosen covariance function but there is *not* a box with *Kernel Optimiser*, because this is currently just implemented as a method in the code and not a stand-alone python class. This could be changed in a future version of the package so that the method for training the model could more easily be changed to e.g. cross-validation.

Objective Function contains the objective function and all its necessary information like bounds, amount of objectives along with either a method to run the objective function directly or a method for saving and loading data from the objective function. The first method is typically useful for test functions while the latter is typically useful for real objective functions like ocean simulations which often need to be run on a cluster. The *Saver* and *Loader* boxes under *Objective Function* signify methods for these saving/loading methods. This package offers a method for loading and saving from *veros* simulations, but if the package is used in another context the user can simply make their own.

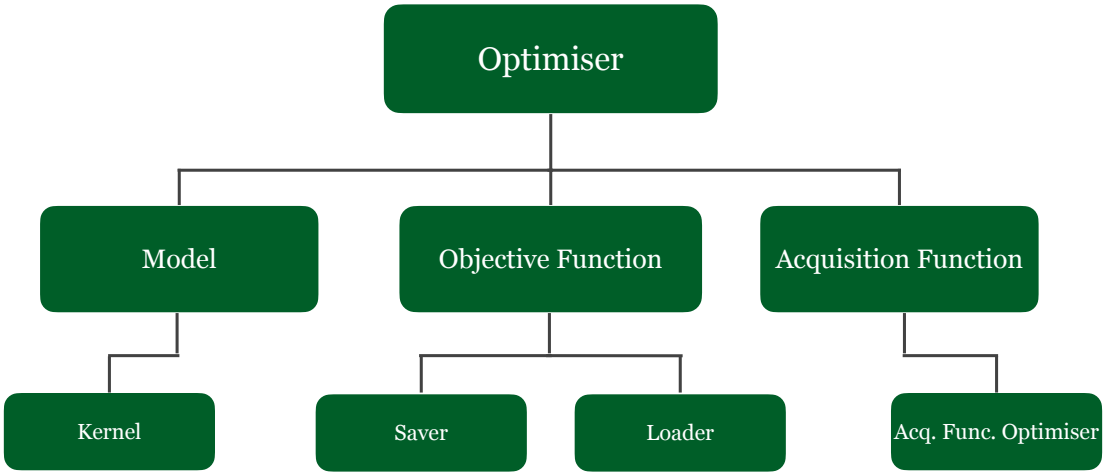


Figure 13: Overall structure of the *veropt* package.

Finally, *Acquisition Function* contains the acquisition function and any potential parameters. Under it, we have the *Acquisition Function Optimiser* which contains the method for optimising the acquisition function.

4.1.2 *Default Set-up*

For the kernel, the Matern kernel was chosen, because it is simple, yet flexible, and makes it very easy for the user to adjust the model in the GUI.

For the optimisation of the kernel we use the MLL, because it is a simple, powerful statistical measure that has a built-in punishment for overly large complexity.

For the acquisition function, the UCB_{Var} was chosen because it is a very intuitive measure that gives the user control over the exploitation vs. exploration balance and gives the model a fail-safe in cases where the parameter dependence is very low. The default values for β is 3.0 and for γ it's 0.01. In the case of multiple objectives, the EHVI acquisition function is used instead, simply because it's the only acquisition function using hyper volume that was implemented, and hyper volume methods are likely to be preferred over weighted sum methods (see the discussion in chapter 2).

For the optimisation of the acquisition function, we use the serial optimisation scheme proposed in chapter 2, because it again is very intuitive and gives the user control over the distribution of the points. The default values for both α and ω is 1.0. [1]

1 Unfortunately, there was an error in the implementation of the serial optimisation scheme. It has been fixed now but all the experiments in this thesis had already been run when it was discovered and it was too late to rerun

As should be clear from the above paragraphs, the prioritisation in these decisions was largely on intuition and simplicity. This is because of the goal to make the tool easy to understand, inspect and adjust. More complex sub-methods can easily be implemented due to the modular structure of the code, and if some of the currently chosen sub-methods turn out to be too simple, different sub-methods can be chosen in future work on this project.

4.2 THE *veropt* PACKAGE

In this section, the reader will find a brief overview of the *veropt* package, including its overall structure and its most important features.

4.2.1 *Underlying Python Packages*

This Python package builds on the *BoTorch*, *GPyTorch* and *PyTorch* packages (see [2], [16] and [31]), given here in the order of dependence (*BoTorch* relies on *GPyTorch* which relies on *PyTorch*). *BoTorch* is "A Framework for Efficient Monte-Carlo Bayesian Optimization". It supplies flexible tools needed for Bayesian Optimisation like classes for acquisition functions, kernels and so on. The kernels are built on the *GPyTorch* library which is described on their official GitHub as a "Gaussian process library" which is "designed for creating scalable, flexible, and modular Gaussian process models with ease". This means that it makes a great basis for our kernels, giving us full flexibility to define new ones if we need to. *GPyTorch* is based on *PyTorch* which gives it a great basis for GPU integration and autograd, ensuring that we can run our models fast and always spend our time as efficiently as possible. Of course, we have lots of time to find candidate points in this project, but we would still rather spend that time on ensuring that we find the best possible points (often by giving the optimisation of the acquisition function (which depends upon the model) plenty of time), instead of wasting it on an inefficient backend. Furthermore, *PyTorch* uses very similar syntax to *NumPy* which is arguably the most widely used package within scientific computing and thus a framework that a lot of people in the field will be well-acquainted with.

This package is then meant to be a further specialisation, using the tools supplied by the aforementioned packages, but requiring less of the user, giving the user tools to look into

them. This means that the candidate points beyond the first might have been spread a little too much and in a slightly biased fashion in the experiments.

the process and being tuned for the case of long evaluation time/a limited amount of data points.

Of alternatives to the underlying package we could mention *GPflowOpt* [15] which uses TensorFlow [40] as a backend instead of *PyTorch*, but it is a much smaller package than *BoTorch* and given *PyTorch's* similarity to *NumPy* one could argue that it's a preferable backend.

4.2.2 *Python Superclasses*

In this project, the aim is to have both great default options and great flexibility to change them. To this end, Python *superclasses* come in incredibly handy. In Python, one can define a class via a different class (the superclass) and thus have it inherent all methods from it.

This means that we can construct generic versions of the classes we need for the submethods shown in figure 13. If we need more specialised versions of these classes, we can then let them inherit the basic structure of the generic superclass but keep the basic structure visible for the user in the superclasses.

If the user then wants to make their own version of one of the classes (because the premade solutions do not work for them), they can let their own class inherit from the generic superclass as well and thus achieve the same structure as the one intended by the project. Furthermore, they can still use the rest of the project along with their own, specialised submethod.

This design ideal was largely achieved with the objective function and acquisition function, less so with the model. This shouldn't be a great limitation, however, as we will discuss below.

In the following four subsections, we will discuss the overall structure of the topmost class and the three at the next level.

4.2.2.1 *Optimiser*

The main class is called *BayesOptimiser* and has the other classes as some of its variables. Besides from this it has a variety of visualisation methods, methods for normalisation, for generating initial steps, saving the optimisation run and so on.

To create an instance of the class, the user must supply three things; the amount of initial points (n_init_points), the amount of points found with the method (n_bayes_points) and the objective function. Everything else has a default value and is not necessary to input.

We will now quickly go over the call signature of the class' __init__ function to get a quick overview of the class variables and what they do.

```
class BayesOptimiser:
    def __init__(self, n_init_points, n_bayes_points, obj_func:
    ↪ ObjFunction, acq_func: AcqFunction = None, model:
    ↪ BayesOptModel = None, obj_weights=None,
    ↪ using_priors=False, normalise=True,
    ↪ points_before_fitting=15, test_mode=False,
    ↪ n_evals_per_step=1, file_name=None, verbose=True,
    ↪ normaliser=None, renormalise_each_step=None):
```

The first three arguments are the mentioned necessary inputs. After this we have the class for the acquisition function and the one for the model. The rest of the parameters are listed in table 1. [2]

Parameter	**Description**
obj_weights	List with weights for the different objectives. Default is even weights.
using_priors	Flag telling whether priors are used. If enabled, the initial steps are found by sampling from the prior. (The prior information must be supplied in the objective function class).
normalise	Flag telling whether or not to normalise.
points_before_fitting	Integer telling how many points to evaluate before fitting the model. This can be lower than the amount of initial points if needed.
test_mode	Flag enabling a mode used when testing out a set-up on a test function. Shows true objective function values while using visualisation tools.
n_evals_per_step	Integer telling how many objective function evaluations to do at each optimisation step.
file_name	String giving the name used when saving the optimiser.
verbose	Flag telling whether or not to print output.
normaliser	Class giving the method to use for normalisation.
renormalise_each_step	Flag telling whether to renormalise after every optimisation step.

Table 1: Table with the input variables for the class *BayesOptimiser*.

2 Note that the code has only really been tested with the *normalise* flag set to *True* so it might require some debugging before it can safely be turned off.

After initialising the class, the method *run_opt_step()* can be used to run an optimisation step. If the objective function has a built-in method for evaluating itself this will,

Algorithm 2: run_opt_step() (1)

Suggest a candidate point.

Evaluate the objective function at the candidate point.

Add the new data from the objective function and update the model.

If the objective function instead has a *saver()* and a *loader()* method it will,

Algorithm 3: run_opt_step() (2)

Load new data with the *loader*() method and update the model.

Suggest a candidate point.

Save the candidate point with the *saver()* method.

In the latter case, the user will have to run the objective function themselves, e.g. by running a *veros* simulation on a cluster. It doesn't really matter to the method though, so it could just as well be running a different type of simulation, possibly in a different programming language or running a physical experiment in a lab and noting the result. All the method needs is a way to save its candidate points so the method running the objective function can read them and a way to load the new data from the objective function.

In the case of *veros* simulations, a saver and loader has been written already and is available on the *veropt* GitHub page.

4.2.2.2 *Objective Function*

The generic superclass for the objective function is called *ObjFunction* and it has the signature,

```
1 class ObjFunction:
2     def __init__(self, function, bounds, n_params, n_objs,
      ↪  init_vals=None, stds=None, saver=None, loader=None,
      ↪  var_names=None, obj_names=None):
```

The required input variables are the function (*None* if the objective function is run outside of the optimiser), the bounds of the tunable parameters, the number of parameters and the number of objectives.

Besides from this, there are a number of optional parameters; init_vals and stds if prior information is used, the saver and loader as described above and the var_names (should

have been names param_names) and obj_names to name the parameters and objectives for the visualisation tools and the GUI.

Inheriting from this superclass, we have the *PredefinedTestFunction*, *PredefinedFitTestFunction* and *OceanObjFunction*.

The first one of these uses a number of test functions from the *BoTorch* package and can be called in the following way: `PredefinedTestFunction("Hartmann")`, here using the *Hartmann* test function.

The second one fits functions to generated data and can be called in the following way: `PredefinedFitTestFunction("sine\_3params")`, here fitting a sine with 3 parameters.

The last one is made for use with the *veros* package and also contains a *saver()* and a *loader()* method.

4.2.2.3 *Acquisition Function*

The generic superclass for the acquisition function is called *AcqFunction* and has the following signature:

```
class AcqFunction:
    def __init__(self, function_class, bounds, n_objs,
    ↪  optimiser: AcqOptimiser = None, params=None,
    ↪  n_evals_per_step=1, acqfunc_name=None):
```

It requires the *class* of the desired acquisition function (expected to have the same structure as a *BoTorch* acquisition function class mainly in that it needs to take the GP model as an input variable when initialising), the bounds of the parameters and the amount of objectives. As optional parameters there is the optimiser for the acquisition function (more about this in a moment), *params* which is the parameter *names* (and probably should have been named par_names), the amount of objective function evaluations per optimisation step and the name of the acquisition function.

To optimise the acquisition function, we have the *AcqOptimiser* which has the signature,

```
class AcqOptimiser:
    def __init__(self, bounds, function, n_objs,
    ↪  n_evals_per_step=1):
```

And simply needs the bounds of the parameters, the function that will optimise the acquisition function, the amount of objectives and the amount of evaluations per optimisation step.

Inheriting from the *AcqFunction* superclass, we have the *PredefinedAcqOptimiser* that can be called like,

```
PredefinedAcqFunction(bounds, n_objs, n_evals_per_step,
    ↪ acqfunc_name='EHVI')
```

Where *acqfunc_name* chooses one of the predefined acquisition functions (*EI*, *UCB*, *UCB_Var*, *EHVI* and *qEHVI*).

Inheriting from the *AcqOptimiser*, we have the *PredefinedAcqOptimiser* which uses Dual Annealing as implemented in scipy [27] and multi-start L-BFGS as implemented in *BoTorch*. It can be called as,

```
PredefinedAcqOptimiser(bounds, n_objs, n_evals_per_step,
    ↪ optimiser_name='dual_annealing')
```

4.2.2.4 *Model*

As mentioned above, the class for the model (*BayesOptModel*) has not been made with a generic superclass like the ones above. This is mostly a limitation in regards to how the model is optimised, because the kernel can still be defined freely. More details below.

The *BayesOptModel* class has the signature,

```
class BayesOptModel:
    def __init__(self, n_params, n_objs, model_class_list:
        ↪ List[MaternModelBO] = None, init_train_its=1000,
        ↪ train_its=200, lr=0.1, opt_params_list=None,
        ↪ using_priors=False, constraint_dict_list=None):
```

It takes the amount of parameters and objectives as two necessary arguments. After this there is a *list* of model classes, which enables a different choice of kernel for every objective. It then has three parameters used for optimising the model's hyperparameters (init_train_its, train_its and lr), a list of dictionaries with which model parameters should be optimised, a flag indicating whether or not the model is using priors (as in the warped kernel method,

see [32]) and a list of dictionaries with constraints for the hyperparameters. The reason we have lists of dictionaries is because we can have multiple kernels. If a single dictionary is given, it is assumed to be applicable for all models. Similarly, if a single model is given, it is used for all objectives.

The hyperparameters are optimised in the class method *train_backwards()* which uses the standard *PyTorch* training loop, as described in [17]. This is very useful if integrating artificial neural networks (like when using the Deep Kernel), because the nodes in such a network can be trained in the exact same way. As mentioned, implementing a different hyperparameter optimisation method might be a bit troublesome, but could probably be achieved without too much hassle by inheriting from the *BayesOptModel* and overwriting the *train_backwards()* method with the custom method.

The class assumes that the model(s) chosen is (are) inheriting from a *GPyTorch* kernel class like *gpytorch.models.ExactGP* and the *BoTorch* class *botorch.models.gpytorch.GPyTorchModel*. A number of models are provided in the same file as *BayesOptModel* (veropt/kernels.py), such as *MaternModelBO*, *RBFModelBO* and *DKModelBO* and can be given to the *BayesOptModel* when making an instance of it.

The constraints follow the structure of the *GPyTorch* kernels and name different *modules* and then different *parameters* in those modules. If the kernel for some objective is RBF or Matern, the default constraints are,

```
self.constraint_dict_list[obj_no] = {
    "covar_module": {
        "raw_lengthscale": [0.1, 2.0]}}
```

4.2.3 *The GUI*

Since the objective functions we're expecting take a very long time to evaluate the optimisation process in its entirety will usually also be a quite lengthy affair. If the objective function takes several days to evaluate, we could be looking at weeks or even months before the necessary amount of points have been found.

Given this time scale, it seems undesirable to simply start the process and then let it run until completed, without checking on it. For this reason (and to generally make the tools of the package more easily available to the user) a GUI was developed.

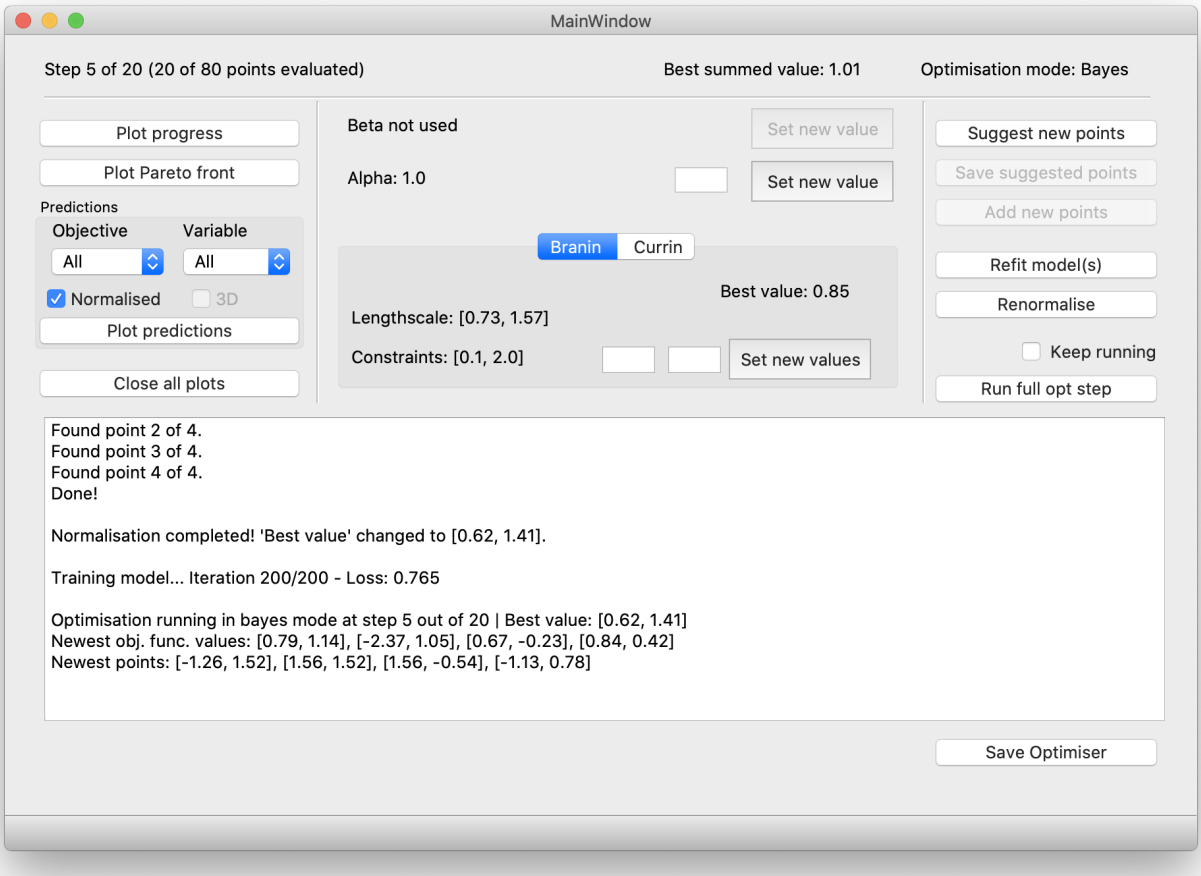


Figure 14: The veropt GUI running an optimisation problem with the Brannin Currin test function.

The GUI is shown in figure 14, optimising the Brannin Currin test function from the *BoTorch* test function library.

On the left side of the window, there is a range of buttons to easily access some of the visualisation tools of the package (more about these below).

In the middle there are a number of editable text-boxes to enter new values for the β and α parameters of the acquisition functions if those are in use (in the displayed example, β is not, so its button is greyed out) and below there are text-boxes for changing the lengthscale hyperparameter constraints, if the selected kernel uses the lengthscale hyperparameter. This means that the behaviour of both the kernel and the acquisition function can be easily altered within the GUI and afterwards inspected with the visualisation tools available at the left side. It would be desirable to automatically detect the parameters being used in both the acquisition function and the kernel, but this is non-trivial, especially for the kernel, because the amount of parameters can easily become very large and displaying them all in an orderly fashion would require some fairly time-consuming GUI development. Therefore the

parameters of the default kernel and acquisition function have simply been made available and other parameters will have to be changed in the python script being used.

At the right, a number of buttons with different functionalities are displayed. The topmost button suggests new candidate points that will afterwards be visible in the visualisation tool that plots the model's predictions. Under, there is a button for saving the suggested points and adding new data from the objective function. These will be available if the objective function has a *loader()* and *saver()* method. Next, we have an option to refit the model if the current fit doesn't seem optimal. If the lengthscale constraints are changed, the model will refit automatically. Then there is a button to re-normalise (mostly interesting with multiple objectives) and lastly there is a button to run a full optimisation step. This will have different behaviour depending on whether there is a method implemented for the objective function directly or alternatively a *saver()* and *loader()* (see above).

At the bottom of the GUI there is a text field that shows anything being printed by any of the methods working in the background. Under the text field there is a button to save the optimiser class as a pickle (.pkl) file.

The GUI only provides access to the methods of the different classes implemented in the class and so anything that can be done in the GUI can be done simply by calling the classes' methods in a script. This means that the code creating the GUI is completely separated from the code performing the optimisation and using the GUI is never a requirement, just an option to make things easier.

Multi-threading has been implemented for some of the more time-consuming methods, like finding new points and refitting the model, so the GUI doesn't just freeze while these processes are running. To make sure that the optimiser class isn't changed in an unsafe way while these methods are being used, all buttons are made inactive while they are running. The progress of the task being done will often be shown in the text box, e.g. updates on how many candidate points are found, just visible in the top of the text box in figure 14). Unfortunately, error messages from the thread that does the more time-consuming tasks, seem to get lost and this bug has not yet been fixed. If that occurs, a solution is to close the GUI and run the same method in the python script to get the error message and fix whatever problem arose.

Since the objective functions are expected to be expensive, it is likely that many of them will run externally on clusters where the GUI can't be used. Because of this, a method has been implemented for saving the optimiser class (more below), and the idea is then to start the optimisation run on the cluster, save the optimiser class after every step (this should

probably be done anyways, to make sure progress isn't lost if the run is interrupted), and then download the saved optimiser class to the local system and run it in the optimiser with a script like the following one;

```
from veropt import load_optimiser
from veropt.gui import veropt_gui

optimiser = load_optimiser("Optimiser_Name.pkl")
veropt_gui.run(optimiser)
```

This will then open the GUI with the ongoing optimisation and the method can be inspected and altered if so desired. If anything is altered, one can simply upload the optimiser class again (and overwrite the original) to implement the new changes.

4.2.4 *Saving the Optimiser Class*

As described in the previous subsection, we need a method for saving the optimiser class.

One simple option is to use the native Python method "pickle" where an object is serialised and can be saved as a file with a *.pkl* ending.

Unfortunately, there are parts of the kernels we use from the *GPyTorch* library that cannot be pickled. The solution is easy, though; It is simply a matter of using the package *dill* [6] instead. *dill* is an extended, more versatile version of *pickle* that successfully serialises the *GPyTorch* kernels.

Using this method gives us the advantage that we capture a perfect snapshot of our optimiser as it was when it was saved. The disadvantage is the more or less the exact same thing. When Python objects are de-serialised like this, they require access to the classes that created them and if these classes have changed then the de-serialisation might fail. This means that this method of saving is not very robust over different versions of the required packages and updating a package (especially *veropt* itself) could mean that old optimisation runs might not load anymore.

A future solution to this might be to always save the data and kernel separately so the most important aspects of the optimisation run are always able to de-serialise and load.

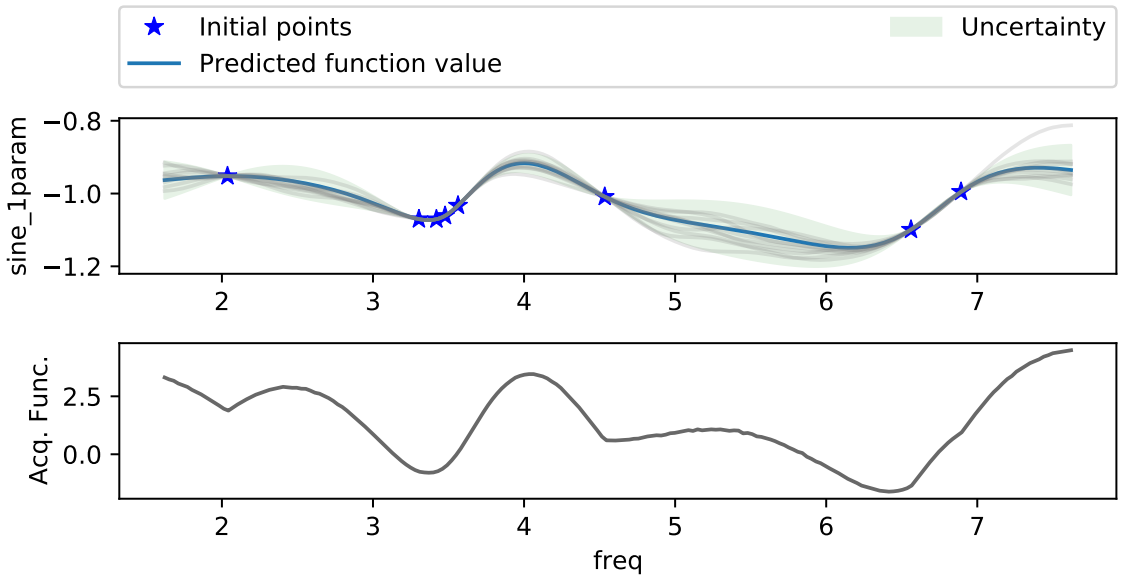


Figure 15: Prediction plot for the test problem *sine_1param*, available in the *veropt* package. Above the objective function data and corresponding model is shown and below the acquisition function is plotted.

4.2.5 *The Visualisation Tools*

For our tool to be easy to understand and easy to adjust in every situation, one thing becomes essential: Visualisation.

Having good visualisation tools allows us to lift the hood and look into the engine, understanding how things are working – or why they aren't.

4.2.5.1 *Prediction Plots*

The most important thing then becomes to give us a window into the most essential parts of the method: The model and the acquisition function. The model is a summary of all the information our method is building on, both the prior assumptions and what we've accumulated from previous objective function evaluations, and if it fails our candidate points cannot possibly be chosen intelligently. Similarly, the acquisition function determines which points we find interesting and if it malfunctions, our points will be chosen poorly.

Of course there is an immediate challenge here: Our problem might have any amount of parameters, making the model for each objective arbitrarily high-dimensioned.

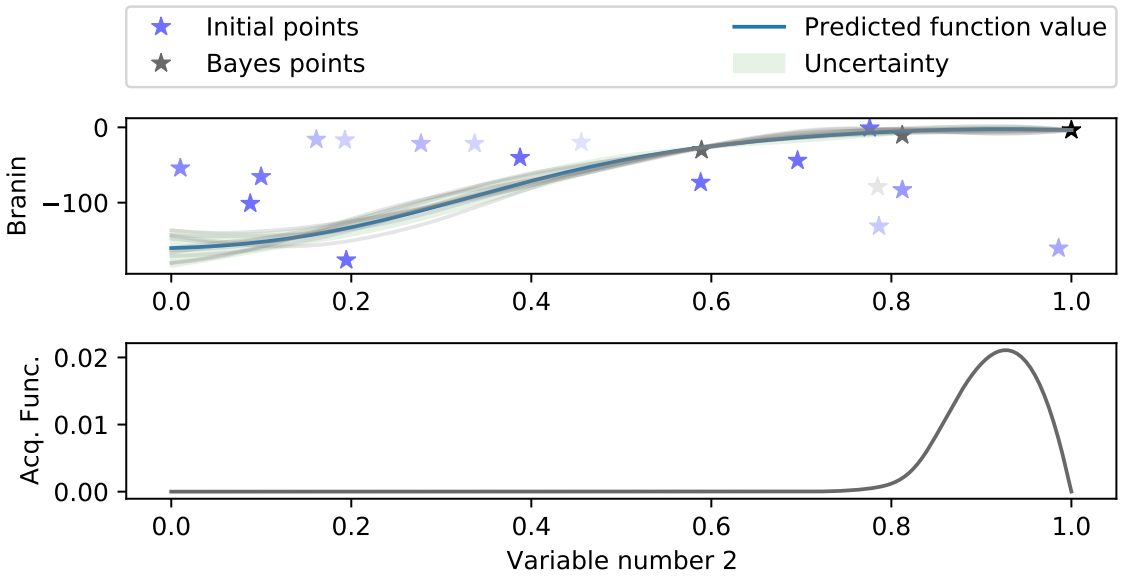


Figure 16: Prediction plot for the Brannin objective in the BranninCurrin test function from the collection of test functions available in the package *BoTorch*.

With one parameter we can make a two dimensional plot (for each objective), with two parameters we can make one in three dimensions, but if we go any higher than that, we can't display all information at once.

What we can do instead is to choose a point and let every parameter except for one be frozen. The chosen, non-frozen parameter will then sweep over its defined values and we essentially get a slice of the model at the point we chose.

For one dimension, this can be seen in figure 15 and for two it can be seen in figure 16.

In the first plot, we have only one parameter and one objective in total. Eight initial evaluations are plotted and we see how the mean of the model goes through them. Furthermore, we see the variance of the model as a green field stretching out on both sides of the mean and we see grey lines going through this field. Since we are in one dimension, the variance falls to zero at all known points. The grey lines are draws of the model and can help the user recognise whether the functions of the posterior distribution seem like reasonable candidates for the objective function (given the data available to both the user and the model at that point). If the draws seem to fluctuate overly much compared to the data, it might be recommendable to tighten the constraints of the lengthscale hyperparameter (if the chosen kernel uses that hyperparameter).

Under the model, we see the acquisition function (here UCB_Var) plotted for the same range of values. We see that it has maxima where the model has a large mean and a large

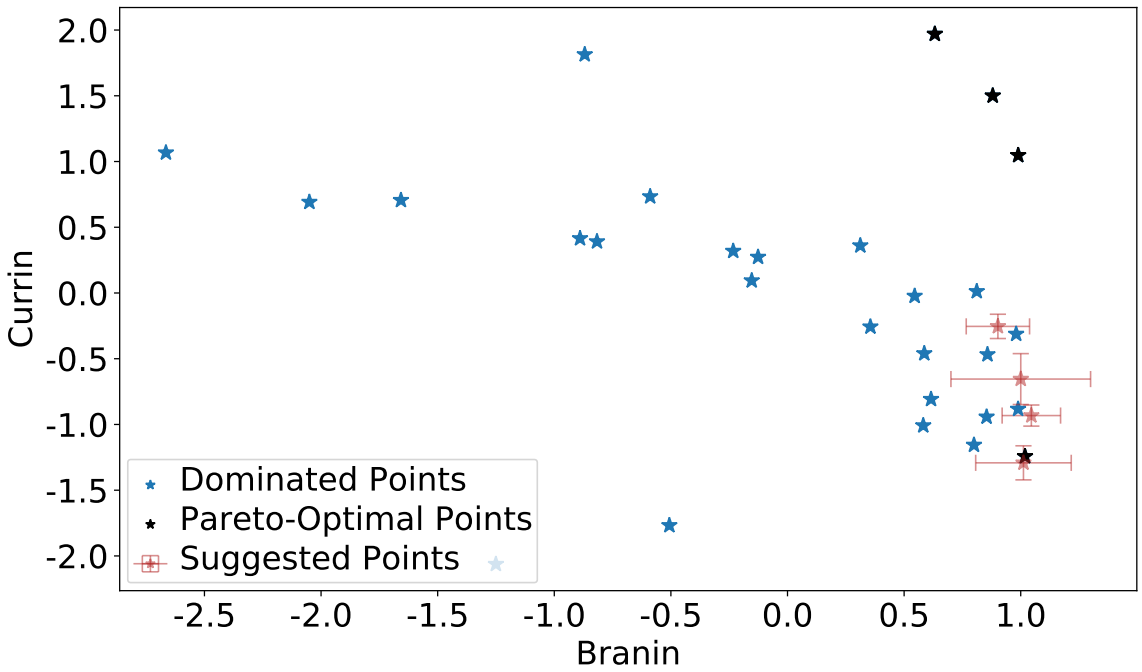


Figure 17: Pareo front plot for the Brannin Currin test functions, showing the distribution of the objective function values for both objectives and marking the dominating (Pareto-optimal) points in black. We also see the mean and variance of the candidate points for the next round of the optimisation.

variance. We also see that it has local minima at all known points, although it may not be clear at all points with the given resolution.

In the other plot (figure 16), we have two parameters and two objectives, Brannin and Currin.

We see largely the same features, although we now note that the model doesn't follow all points because some of them are not in the plane we are inspecting the model in. To make this more visible, points further away are more transparent, while points in the same plane as us are completely opaque (this is only true for the black point at the upper right corner). We also see that this acquisition function (Expected Hypervolume Improvement) is largely zero and has only one maximum (in this plane).

The selected point for the inspection is either the best known point (found by weighted sum if there is more than one objective) or the found candidate point with the highest acquisition function value. It will be the latter if a range of candidate points have been found. In that case, these candidate points will be shown in red and with uncertainty bars to show the model's uncertainty in their value.

This method was implemented in three dimensions too, but unfortunately the 3-D version still needs to be updated to support multiple objectives and isn't currently functional.

The method is available in the GUI via the *Plot predictions* button, over which the desired objectives and parameters can be chosen. The user can also choose whether or not to enable the normalisation in the plot.

4.2.5.2 *Pareto Front Plots*

If we have multiple objectives, plotting the dominating points against each objective can be very interesting and show us how our model is sampling the Pareto front. In figure 17, such a plot is shown for a run with the Brannin Currin test function. We see that three points with a nice balance between each objective have been found, but that the method seems slightly biased towards the Branin objective. The new candidate points are shown and all favour the Branin objective.

This visualisation tool is available via the GUI (by pressing the *Plot Pareto front* button), but currently it doesn't support choosing which objectives to include, so it is only active when there's two or three objectives (giving a three dimensional plot in the latter case).

4.2.6 *Using Priors*

The *veropt* package has been designed to support prior information. The prior information (currently assumed to be a mean value and standard deviation of a Gaussian prior) is entered into the objective function class. When the main optimiser class (*BayesOptimiser*) is initialised, the flag *using_priors* can then be set to *True* and the optimiser will then make an instance of the *PriorClass* with truncated normal distributions from *scipy* (see [34]), enabling us easy access to both the probability distribution function and the cumulative distribution function. The latter is needed for the warped kernel method discussed in chapter 2.

When the *using_priors* flag is activated, the optimiser will automatically sample initial points from the prior distributions. This should perhaps be made optional in a future version of the package.

The model class (*BayesOptModel*) also has a flag called *using_priors*. If this is activated it will pass the prior information to the chosen kernel, e.g. a warped kernel. When the optimiser class initialises the *PriorClass*, it automatically passes it on to the model class.

It unfortunately wasn't reached to implement the proposed acquisition function from chapter 2, a_{UCB_prior}, so it is currently not available and hasn't been tested out during the project.

4.2.7 *Predefined Ocean Objectives*

In order to run the ocean simulations discussed in chapters 6.5.2 and 7.4, an objective function class was written to be used with simulations from the *veros* package. It contains a *saver()* and *loader()* method embedded in a *Saver* and *Loader* class. The *saver()* method works by saving a *pandas* [28] DataFrame with the parameters' names and values for each candidate point along with an id for that simulation run. This id is given to the *veros* simulation when it starts and defines it filename. Furthermore, the a method from the *Loader* class can be called within the simulation and will then return the parameter values that it is supposed to use. The *loader()* method used by the optimisation can then identify the different simulation runs by looking at their filenames and will know which parameters have been used in each run.

See the appendix for code excerpts explaining how to write an objective function class with the *OceanObjFunction* as a superclass and how to use the *loader_class* in the *veros* simulation set-up file.

The *OceanObjFunction* and its submethods are currently written for one objective, but could easily be extended for multi-objective problems.

4.2.8 *Slurm Tools*

To make it easier to start an optimisation run on a cluster, a number of Python scripts (found under *veropt/slurm_support*) were written. The *slurm_controller* script manages an ongoing optimisation by running the *run_opt_step()* method of the main optimiser class and starting *veros* simulations with the parameter values from the new candidate points. The *slurm_set_up* script sets up the necessary shell scripts for submitting to the slurm queue, both for the *slurm_controller* and the *veros* simulations. If the *veros* simulations require a complicated parallel setup, the batch files can just be edited.

The *slurm_set_up* script was written for optimisation runs on the *MODI* cluster and experiment runs on the *AEGIR* cluster and might be a bit inflexible when applied other places.

Future versions of *veropt* could expand on both these scripts to make optimisation runs easy on all (or at least *most*) platforms using slurm.

4.2.9 *The Experiment Class*

To make it easier to compare different set-ups, a class was implemented called *BayesExperiment* that takes a list of *BayesOptimiser*'s and runs them all. Afterwards these can then be plotted and compared in various ways.

To make this code run faster, a multiprocessing method was implemented, both one for laptops with shared memory that uses python's native multiprocessing module and one for clusters that uses MPI. Since the problem is embarrassingly parallel this could give a speed-up that is pretty much proportional to the amount of cores that can be utilised, if the serial version doesn't already utilise parallelism. Having this class means means that if a new method is implemented and needs to be tested against an existing one, this can be done easily.

4.2.10 *Simple Example*

4.2.10.1 *Hartmann Test Function, Default Options*

To give some direct insight into how the package can be used, we here show a simple example using the Hartmann test function and the default optimisation choices. Similar examples can be found on the *veropt* GitHub page [22].

```
from veropt import BayesOptimiser
from veropt.obj_funcs.test_functions import
    ↪ PredefinedTestFunction
from veropt.gui import veropt_gui

n_init_points = 24
n_bayes_points = 64
n_evals_per_step = 4

obj_func = PredefinedTestFunction("Hartmann")

optimiser = BayesOptimiser(n_init_points, n_bayes_points,
    ↪ obj_func, n_evals_per_step=n_evals_per_step)
```

```
12
13 veropt_gui.run(optimiser)
```

Part III

TEST FUNCTIONS EXPERIMENTS

5

TEST FUNCTION EXPERIMENTS

> The most exciting phrase to hear in science, the one that heralds discoveries, is not 'Eureka!' but 'Now that's funny…'
>
> *Isaac Asimov*

Before we use our method to optimise some real ocean simulations, we want to try it out some test functions to make sure it actually works.

Now, of course there is no guarantee that these test functions share enough structural properties with the ocean simulations for the results to be transferable but this is true also from one ocean simulation to another, and an inherent challenge with trying to make a general optimisation tool.

Furthermore, the method we use has many parts and requires many choices. In chapter 4, we chose a default set-up on principles of simplicity, transparency and adjustability, but it might be interesting to actually do some experiments to test out different sub-methods against each other.

Only one such experiment was reached in the end, but we finish the chapter with a discussion of future possible experiments.

5.1 BO VS RANDOM SEARCH

First off, we would like to simply confirm that our method performs better than a random search, since that is one of the simplest methods to employ when no advanced tools are available to the user.

The BO set-up we use here will be the default from the *veropt* package, described in chapter 2. We will do 24 initial points and 76 points where the method is used.

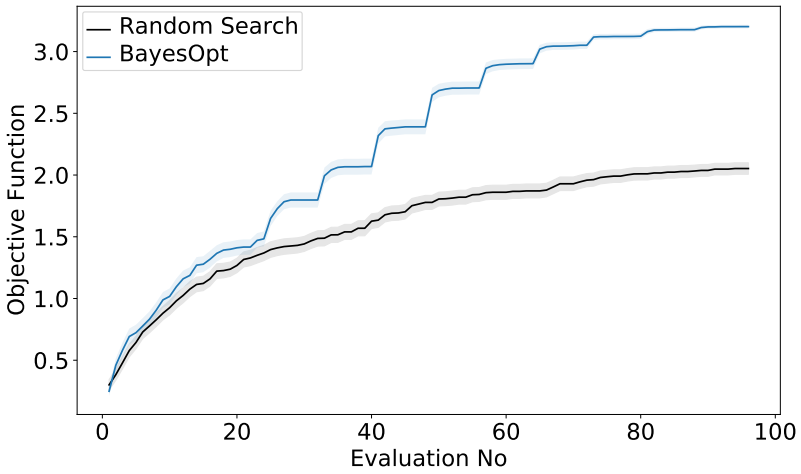


Figure 18: The mean (and its uncertainty) of the cumulative best objective function value for Bayesian optimisation and random search on the test function Hartmann, available through the *BoTorch* package.

With multiple objectives, the default in *veropt* is now to normalise at each step, but these experiments were run before that was implemented so that feature is not activated.

5.1.1 *Hartmann Test Function*

For single objective optimisation we will start off with the test function Hartmann with 6 parameters and one objective. In figure 18 we see the best value that has been found at a given point of the optimisation, both for random search and our method. We see the uncertainty of the mean of the distribution of best values for both searches (the standard deviation of that distribution divided by $\sqrt{N_{reps}}$) in a shaded area around the mean. That is, we see how the methods have done in average and how uncertain that average is.

At point 24 we see how the mean for the BO method jumps, since this is where the method is activated. We see how it improves at every batch of 8 points until it hones in on the global maximum which has the value 3.3224. This is exactly the kind of evolution we want to see, steady progress towards the global maximum.

5.1.2 *Fitting a Sine*

For the second single objective problem we try to fit three parameters of a sine. We do this by generating some noisy data with known parameters and then create the objective function by measuring the mean square error (MSE) between the data and the fit made with the

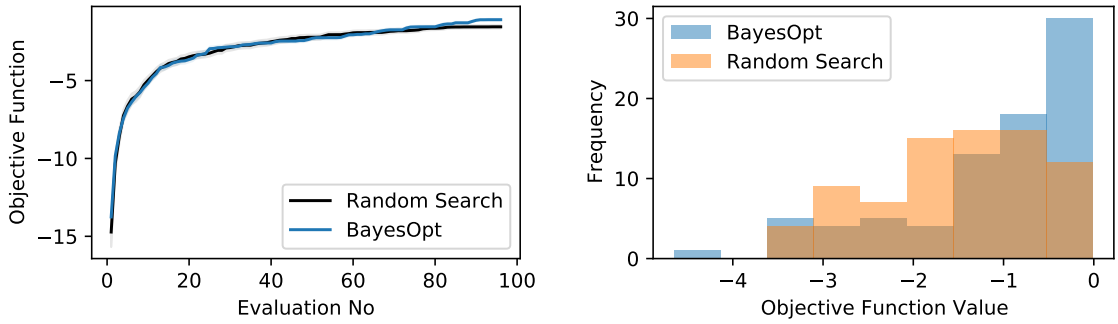


Figure 19: Mean and its uncertainty of the cumulative best value (left) and histograms of the final best values (right) both for the test function *sine_3params*, available in the *veropt* package.

proposed parameter values. In the *veropt* package, the class `FitTestFunction` can do this automatically and the class `PredefinedFitTestFunction` has a number of functions predefined, one of them *sine_3params* which is the one we use here.

This is a surprisingly difficult problem, partially because the sine obviously gives rise to some periodic behaviour in the objective function with lots of local optima, and partially because we are trying to fit the frequency, amplitude and offset and the interdependences of these parameters are quite tricky.

In figure 19, on the left side, we see the evolution of both methods, BO and random search, just like we did for the Hartmann function in figure 18. We see a smaller difference between the methods here, probably mainly because it is easy to find not-terrible parameters with scores above ~ -3. On the right side though, we see a histogram with the final best values of all 80 optimisation runs for both method. We see that BO has more than double objective function values in the interval between $[-0.5, 0.0]$. We could probably do better than this by choosing a more specialised kernel (like a periodic or a spectral mixture kernel), but the point here is to try out with the default set-up and see how it performs, and we still do see a significant improvement over random search. We might also have seen a different tendency if we had chosen an error measure (instead of MSE) with larger reward around the global maximum, i.e. larger reward for getting close to zero error.

5.1.3 *VehicleSafety Test Function*

For multi objective optimisation (MOO) we use the test function VehicleSafety (from the BoTorch [2] library of test functions), which has 3 parameters and 3 objectives.

In figure 20, we see the best point found at a given point as we have for the last two experiments. Here, however, we have multiple objectives, so the information in this figure is not quite as easily chosen. On the left side, we see the best points when that is measured by doing a weighted sum (here with equal weights) of the objectives. On the right side, we see the best value found in each objective.

In the left-hand plot we can see that the optimisation jumps to a point where the weighted sum of the objectives is large, but then doesn't find anything much better (by that metric) afterwards. In the right-hand plot we see that this jump is largely brought about by the improvement in the third objective, which jumps at point 24 and then doesn't improve much. We can see that the other two objectives also do better on average than in the random search, however, and we note that there is more gradual progress in the first objective. Now, gradual progress isn't necessarily good if it is compared to immediately finding the best possible section of the parameter space. But it might also mean getting stuck at a local optima and not improving from there.

For this experiment we only normalised once at the beginning of the optimisation runs, and that might affect how the optimisation progresses. Since the third objective quickly find a large improvement we effectively have that its values has a larger range than the other objectives. This is because the initial normalisation puts the values between roughly $[-1.5, 1.5]$ and if an objective then greatly succeeds what the intial points found, it will effectively have a range from e.g. -1.5 to 6.0 instead, thus potentially dominating the search. To counteract this there is now a parameter in the *veropt* package that makes the optimiser re-normalise at every step. It would be interesting to run an experiment to see if this re-normalisation makes the optimisation better. [1]

5.2 TUNING ACQUISITION FUNCTION PARAMETERS

5.2.0.1 *Alpha and Omega*

In the sequential optimisation of the acquisition function proposed in chapter 2, there are two parameters: α and ω. We proposed in that chapter that they should have a value of about 1.0.

To test out this idea, we ran an experiment with the VehicleSafety test function, varying α values from 0.1 to 2.0 and ω values from 0.1 to 1.0. These parameters might have some

1 Note that the optimisation runs also had to be re-normalised before entering into these plots, in order to be comparable.

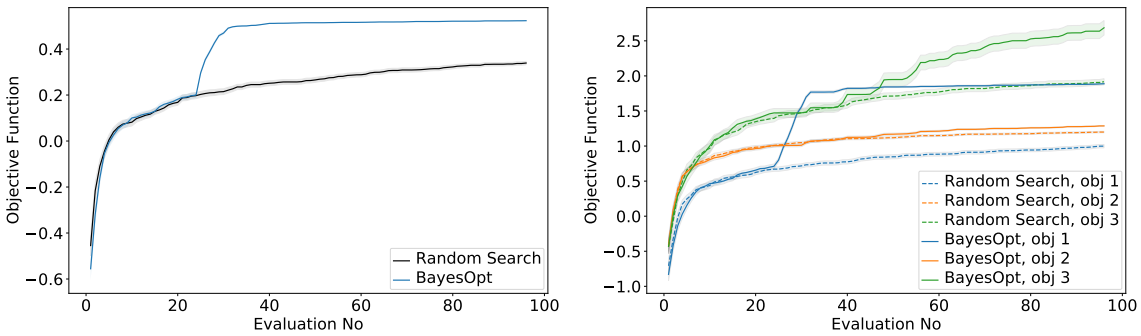


Figure 20: Mean and its uncertainty of the cumulative best value when taking the weighted sum of the three objectives (left) and the mean and its uncertainty of the cumulative best value of each objective (right), both of the VehicleSafety test function available in the *BoTorch* package.

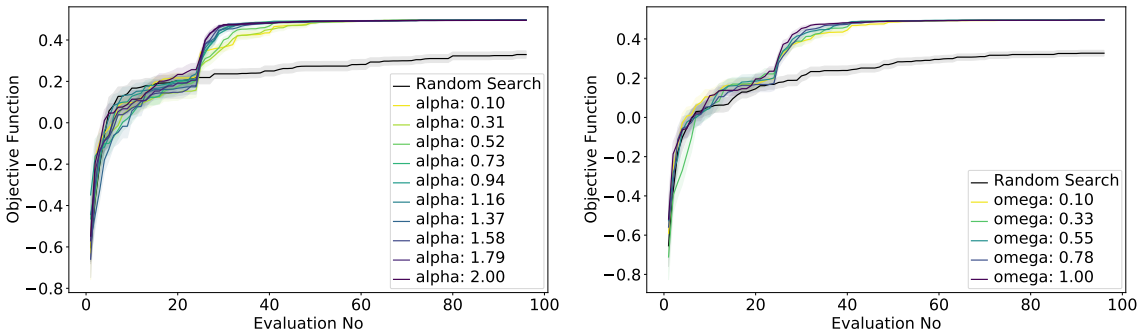


Figure 21: Cumulative best values (of the weighted sum) for the VehicleSafety function with varying values of α (left) and ω (right).

amount of interdependence but for now we just vary one at a time, keeping the other one at the default value of 1.0.

In figure 21, we see the result of this experiment. From this simple test, it would seem that higher values of α and ω generally do better, and it seems that a value of 1.0 for each of them is not completely unreasonable.

Of course, this is just the results for one test functions and with other test functions we might see other results. Particularly we would probably expect to see smaller values of α and ω perform better at smaller parameter spaces, but the default values of the method should probably be tuned more for more complex problems where it is of greater use.

Either way, it is nice to see that the intuition from chapter 2 fits with these results.

As mentioned in chapter 2, there was an error in the implementation of the serial optimisation scheme, so the results of this experiment could in principle change slightly if run again, without this error.

5.3 FUTURE WORK

There are lots of interesting experiments that could be conducted to investigate the performance of different parts of the method.

First off, it would probably be advisable to decide on a range of test functions with the desired properties. It is probably desirable to make them structurally diverse in the objective function but to make them relatively complex, since we want the method to work well for more complex problems.

One interesting thing to examine would then be the performance of different kernels on this range of test problems. One could try to compare the performance of more simple kernels like Matern to more complex kernels like SMK.

Another interesting thing could be to examine the performance of different acquisition functions, e.g. EI vs UCB_Var or EHVI vs a weighted sum acquisition function.

The shape of the objective when trying to fit functions is also of great importance, and matters greatly when we use the method to tune ocean simulations. It would be interesting to see, for example, if it is a disadvantage to have very sharp peaks in some objectives (e.g. by taking the logarithm of the error) when trying to optimise many objectives at once.

It could also be interesting to test out the influence of prior information, either by just using it for the initial evaluations or by using the warped kernel method discussed in 2.

Finally, as mentioned somewhere above, it would be interesting to see the influence of re-normalising at every step versus that of only influencing once at the beginning.

Part IV

OCEAN SIMULATIONS

6

OCEAN SIMULATIONS

> In this field we are not pushed by
> experiments but pulled by imagination.
>
> *Richard Feynman*

To test out the optimisation tool, three different *veros* set-ups were tuned. The set-ups are of increasing complexity so the tool can be tested out in a simple environment first and then be given harder and harder challenges. The source files for the experiments described in this part of the thesis are available in the *veropt* GitHub [22] page, under examples/ocean_examples. It was originally the intention to do a fourth multi-objective problem as well, but unfortunately time didn't allow it.

6.1 OPTIMISATION SET-UP

To test the method out in a straight-forward manner that would not have required expert knowledge of Bayesian Optimisation to select, we simple optimise the ocean simulations with the default *veropt* set-up. See chapter 4 for these default choices.

6.2 DIFFERENCE MEASURE

As discussed in chapter 2 the difference measure between the current output of the simulation and the target value is quite important to the behaviour of the optimisation. It determines both how easy it is for the surrogate model to fit the data and how the optimisation is rewarded for coming closer to the target value. For these ocean simulations we have simply used the quadratic difference between the target value and the simulation output. This gives us a parabolic objective function if the simulation output is linear in the chosen parameters

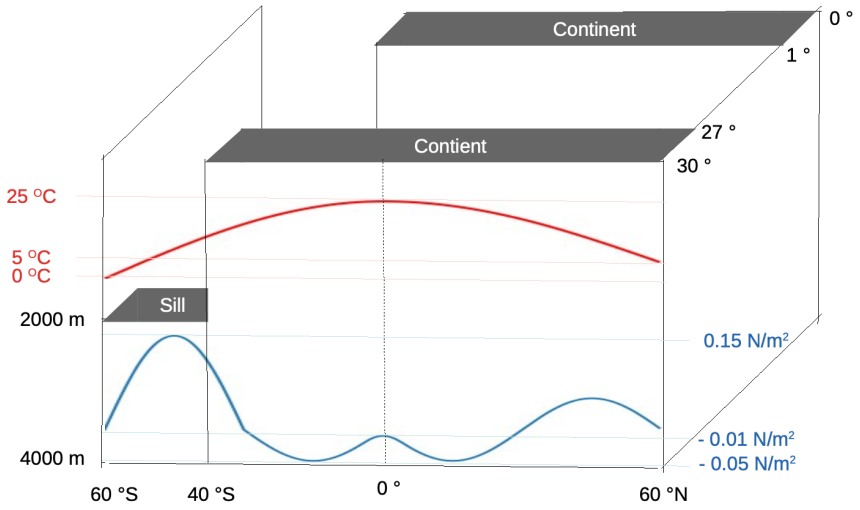


Figure 22: Physical set-up for the first and second ocean simulation. Figure from [1].

(or in areas where it is), and makes it easier for our model to fit the objective function. It also means that the optimisation is rewarded largely for matching the output in the first couple of digits but will not be greatly rewarded for higher precision.

In simulation three we will try to optimise for the logarithm of the quadratic distance as well, and see if the optimisation is capable of obtaining higher precision in that way.

6.3 SIMULATION ONE

The first simulation was constructed to be as simple as possible, giving us a way to make sure the tool performes as expected. The set-up of the model is a very simple, default one from the *veros* setup gallery [36], named *acc*. We will describe the most important details of it below.

6.3.1 *The ACC Set-up*

The boundaries of the ocean in this simulation form a rectangle, creating a simple box model that aims to mimic the Atlantic Ocean. The coordinates run from 60º South to 60º North and from 30º East to 30º West. The depth is 4000 m everywhere expect for 60ºS to 40ºS where it is 2000 m. In this bottom part there is also periodic boundary conditions between the east and west border to mimic the conditions in the Southern Ocean and create the necessary

conditions for the Antarctic Circumpolar Current. See figure 22 for a visual representation of this set-up.

The resolution of the model is `vs.nx, vs.ny, vs.nz = 30, 42, 15`, which means that there are 30 grid points in the x-direction, 42 in the y-direction and 15 in depth. This means that we have a $\sim$ 2.5 degree resolution in latitude/longitude and a $250m$ resolution in depth.

For more details on the acc set-up, see [1], chapter 3.

6.3.2 *Parameterisations*

In this simulation we use the simple diffusion along isopycnals with the parameter κ_{iso}, the Gent McWilliams parameterisation with parameter κ_{GM} and the TKE parameterisation. The EKE parameterisation, which is also implemented in *veros* has been turned off, letting us set a constant κ_{GM} instead.

6.3.3 *The Optimisation Problem*

Since we want a simple problem, we have just one parameter to change and one objective to optimise. The one parameter to change will be the joint value of κ_{GM} and κ_{iso}, as described in chapter 3. We will name this joint value κ_j for ease of notation. The output we will use for the tuning will be the zonal mean of the vertically integrated streamfunction, measured at the Southern border of the model, giving us a measure of the strength of the Antarctic Circumpolar Current (ACC). This was chosen simply because we know that there are steep isopycnals in the Southern Ocean and we know that a higher value of κ_{GM} should work to flatten those isopycnals. Thus we know very certainly that the parameter we're changing will affect our objective, and that something in our set-up is wrong if we don't see a dependence.

The target value for this streamfunction comes not from data but from an identical simulation with a known value of $\kappa_j = 1000m^2/s$. This was chosen for this first experiment because then we can try to retrieve this known value and again confirm that everything works as it should.

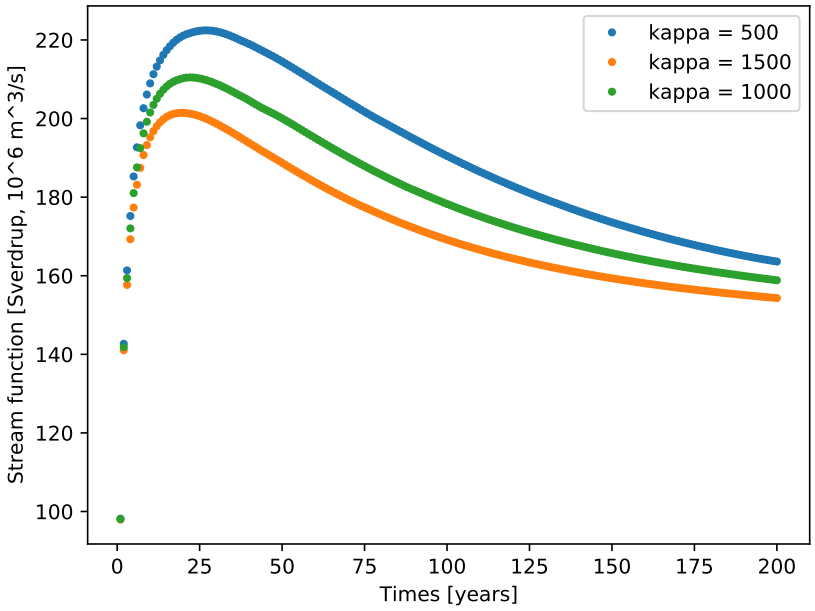


Figure 23: Three test runs of the first simulations. We see the zonal mean of the vertically integrated streamfunction at the southern border of the model as it changes over time for three different values of κ_j.

6.3.4 *Test Runs*

Before running the optimisation itself, three simulation runs were done with different values of κ_j to confirm the dependency and check the spin-up time. In figure 23 we see the measured stream function at the Southern boundary of the model from years 0 to 200 with three different values of κ_j. As we can see in the figure, the value of κ_j strongly influences the strength of the ACC.

We can also see that the model appears nearly stable around year 200, so we will make our measurements in the optimisation run there. We could choose to run it a bit longer to go into complete equilibrium, but that would impact the runtime, so we compromise and simply measure at year 200.

6.4 SIMULATION TWO

In the second simulation, we will still be using the *acc* set-up described above and the same parameterisations.

6.4.1 *The Optimisation Problem*

For this second simulation, we will change the value of one more parameter. We'd like to change the strength of the vertical mixing, but since we've activated the TKE parameterisation, we can't directly change κ_v to some scalar value. Instead we use the minimum of κ_v as a parameter, effectively changing the vertical mixing in the areas where it is weakest.

Another change from the first simulation, is that we will now actually try to use the AMOC as our objective. We will still be using a reference value from an identical set-up with $\kappa_j = 1000$ and $\min(\kappa_v) = 2e - 5$.

To measure the AMOC we will look at the vertical minimum of the zonally integrated stream function, showing us the transport in the meridional direction. The meridional location of this measurement should not matter in theory, since we expect the current to be constant in strength along its trajectory and for this trajectory to be circular (as depicted in figure 12) in a vertical/meridional view. Thus we could in principle measure the largest negative (southward) meridional transport at any meridional coordinate. In practice, however, we might find that this measurement is more stable at some locations than others. Three simulation test runs were done to examine this further.

6.4.2 *Test Runs*

To find a good location for our measurement and check the spin up time, we perform three test runs with three different values of $\kappa_j = 500, 1000, 1500$. In figure 24 we see a vertical/meridional contour plot of the meridional water transport, one for $\kappa_j = 500$ and one for $\kappa_j = 1500$, both measured at year 100. As expected we see that there is largely northward transport at the surface and southward transport in the depths. This figure confirms that the behaviour is as expected, but it is perhaps difficult to guess where the largest numerical stability will be achieved, so in figure 25, the temporal evolution of the minimum transport along the z-axis was plotted, for $\kappa_j = 1000$. Here we see that four meridional coordinates seem to settle into an equilibrium after 200 years of spin-up; -40°, -30°, 0° and 10°. Out of these we select 0° so that we choose to measure at the equator. For this output, the model seems to be adequately settled after 100 years, so that is when we will measure in the optimisation.

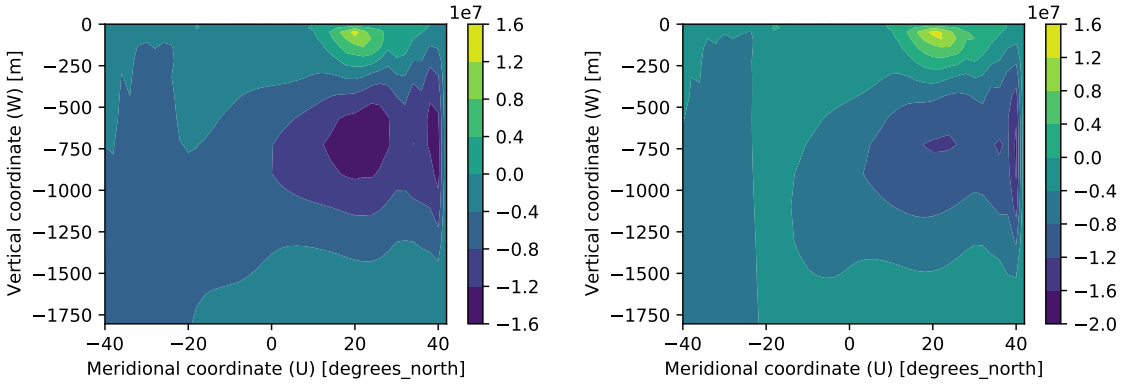


Figure 24: Contour plots of the zonally integrated meridional transport for $\kappa_j = 500$ (left) and $\kappa_j = 1500$.

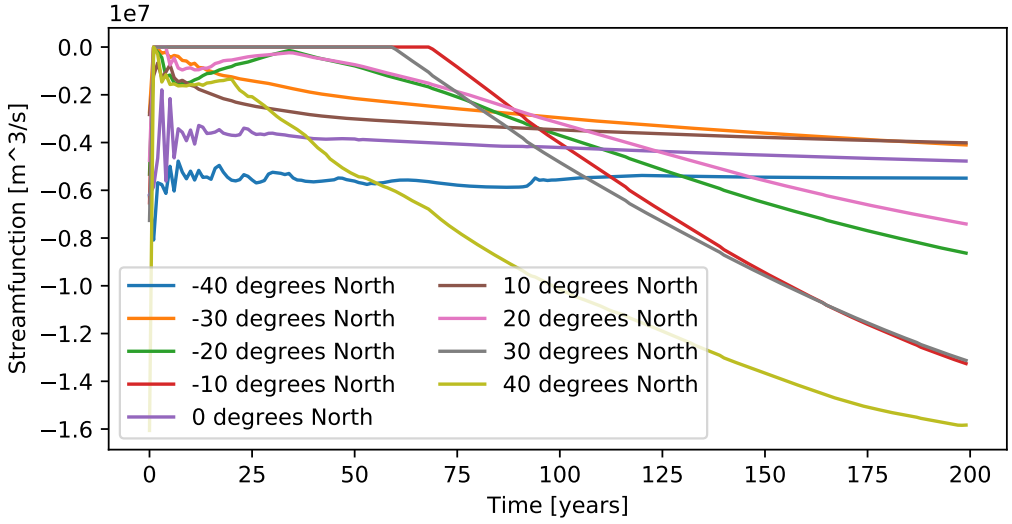


Figure 25: The vertical minimum of the zonally integrated meridional transport at different meridional coordinates.

6.5 SIMULATION THREE

In the third simulation, we finally move out of the box and work with a real model. Now we have a complete ocean system with continents and everything. The amount of grid points is `vs.nx, vs.ny, vs.nz = 90, 40, 15` so we have roughly four degrees resolution in latitude and longitude. The set-up we use here is the one called Global four-degree model in the *veros* set-up gallery [36].

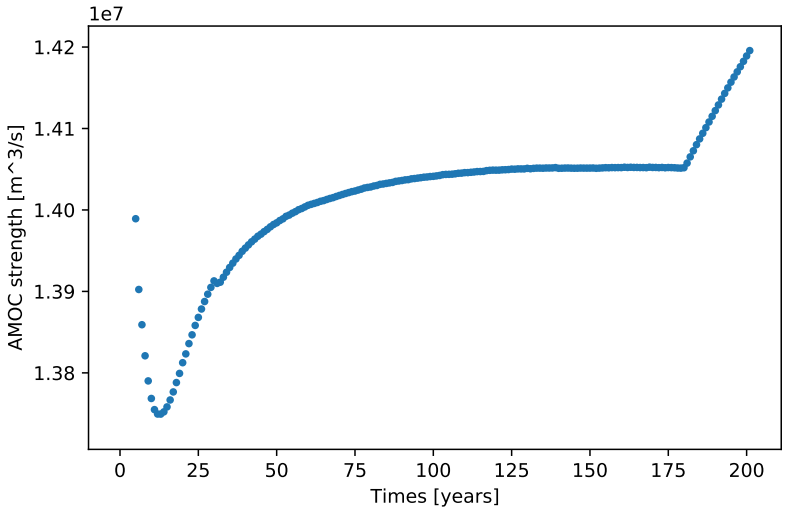


Figure 26: The vertical minimum of the zonally integrated meridional transport at 20°N.

6.5.1 *The Optimisation Problem*

For this simulation we will separate κ_{GM} and κ_{iso}, since we are in a more complicated setting and might need the parameterisations to be tuned individually. We're also still tuning $\min(\kappa_v)$, bringing us to three parameters.

For our objective, we will choose the AMOC again, measured largely in the same way. We will, however, move the meridional coordinate of the measurement to 20°N.

For this problem we will not be using an identical, reference set-up but instead tune our measurement of the AMOC to 15 Sverdrup since this is its approximate strength of it [24].

6.5.2 *Test Runs*

For this set-up we didn't reach testing an optimal meridional coordinate for the measurement, but in figure 26 the measurement at 20N can be seen. Note that there is a large swing up to 21 Sv at the first couple of years that is out of frame of the figure (to make the other tendencies visible) before the value dips and goes towards equilibrium. We see that the model is stable around 100 years, but has some kind of divergence after year 175. This might be an instability of the model or just an effect of choosing an unfortunate place to measure. Either way, the model seems stable and in equilibrium at year 100, so this is when we will measure. Further investigation into the rise in value after year 175 might have been desirable, but wasn't reached.

7

RESULTS & DISCUSSION

7.1 SIMULATION ONE

For this first simulation, we just run 8 initial points and 24 points with bayesian optimisation. We will run 8 points every round to make the process go faster.

In figure 27 the data from the optimisation is plotted, along with the mean and variance of the model. We see that the objective function is approximately parabolic in a large section of the parameter space, which must mean that the simulation output is linear in the parameter value, given our chosen difference measure. The maximum of this parabola is around $\kappa_j = 1000$ as expected.

In the left of figure 27 the model is shown as it was when the optimisation stopped. We notice that the variance is quite large between points, telling us that the functions in the posterior distribution of the model oscillate between points. This tendency doesn't seem supported in the data and likely means that the lengthscale is too small. The fitted lengthscale by the MLL optimiser was 0.35 and the default bounds for the lengthscale are $[0.1, 2.0]$, but these can easily be adjusted in the GUI. Setting the lower bound to 0.7 instead and refitting (effectively forcing the lengthscale to at least double), we get the model seen in the right of the figure. This looks more reasonable and would likely overall perform better.

Using the method `optimiser.best_coords(in_real_units=True)` we are told that the parameter value that gave us the highest objective function value was $\kappa_j = 1018.5$ which is close to the target. Of course we could easily get higher precision for such an easy problem but the overall aim with the current set-up is to get the first couple of digits right (otherwise we should choose another difference measure). This was chosen because ocean simulations are generally complex entities and the end goal of this tool is to be able to set a large amount of parameters to an acceptable value.

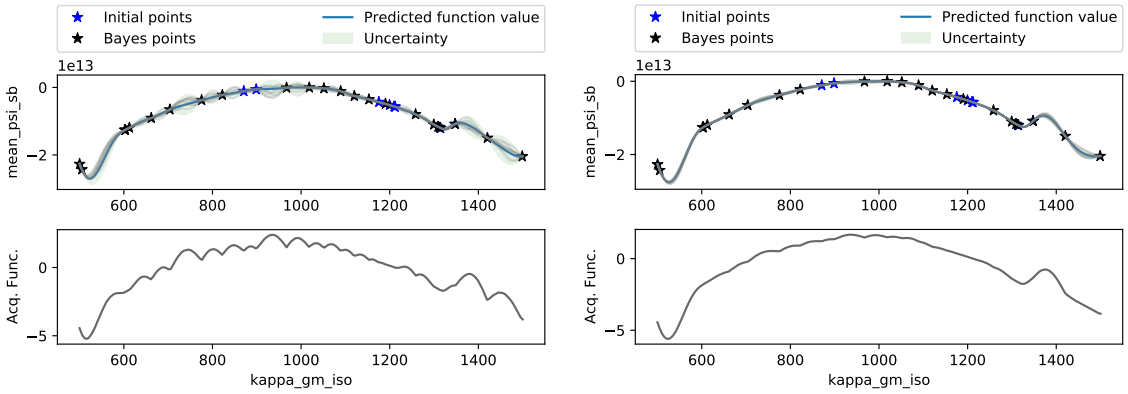


Figure 27: Objective function values, model predictions and acquisition function values for the first simulation.

The method was likely also impaired by having to evaluate eight points at each evaluation. Since this problem only has one real maximum that takes up most of the parameter space, there is not much to explore with the points beyond the first, at least not with $\alpha = 1.0$ (recall that this is the parameter that spreads points out when using the sequential optimisation of the acquisition function that was proposed in chapter 2. We might get better results then, by setting α lower, but, again, we did achieve what we wanted and got the two first digits of the parameter right.

The objective function value (found using `optimiser.best_val(in_real_units=True)`) at the best point is $-2.9e10$ which might sound dramatic given that the best possible value is 0, *but* the target value from the reference simulation is $158.85e6$ (rounded) and we have squared the difference. This means that the linear difference is $1.5e5$ and the relative error is $1.5e5/158.85e6 \approx 0.001$. So we can easily say that we've tuned the simulation to our target value. (Again, if we needed high precision in this target, we would simply have chosen a different difference measure.)

In figure 28, we see the objective function value at the different points. In this simple problem there isn't really a big difference between the two. In fact there are a few outliers from the BO, probably because it had to choose 8 points at once and some of them got pushed far away from the interesting region.

7.2 SIMULATION TWO

In the second simulation, we ran 8 initial points and 40 with bayesian optimisation, since we now have 2 parameters. We will again do 8 points per round.

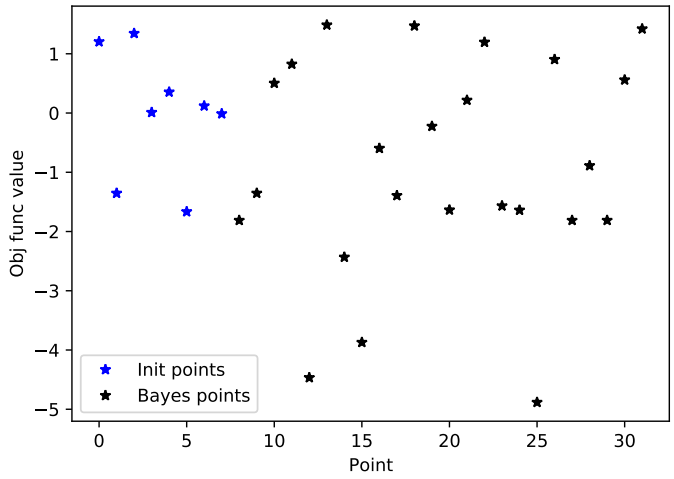


Figure 28: Objective function values at different points for the first simulation.

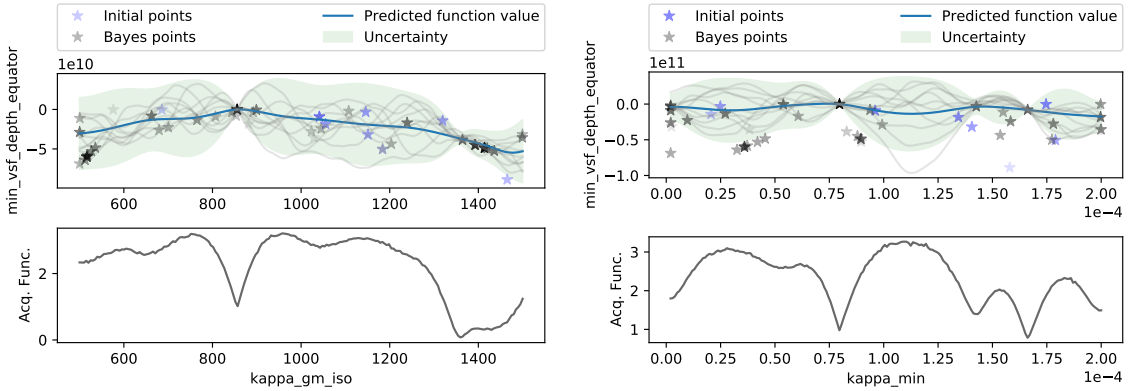


Figure 29: Objective function values, model predictions and acquisition function values for the second simulation. This is a two-dimensional parameter space and we're seeing a slice for each parameter at the point with the best objective function value. The plot with varying κ_j is on the left and the one with varying $\min(\kappa_v)$ is on the right.

In figure 29 we see the data and model from simulation two as it looked when the optimisation had just finished. We see again that the lengthscale appears to have been fitted too low with 0.38 for κ_j and 0.34 for $\min(\kappa_v)$. Just refitting the model fixed this problem for $\min(\kappa_h)$, changing the lengthscale to 1.30, but κ_j still had an overly small lengthscale. To fix this, we again increase the lower bound, this time to 1.0. Refitting then gives us a lengthscale of 1.0 for both parameters. The updated model can be seen in figure 30.

The best parameter values were $\kappa_j = 855.7$ and $\min(\kappa_h) = 7.97e-5$, means we're a bit more off than in the first simulation. As for the tuned simulation output, we had a relative error of 0.0034 from the target value (linear difference/target value).

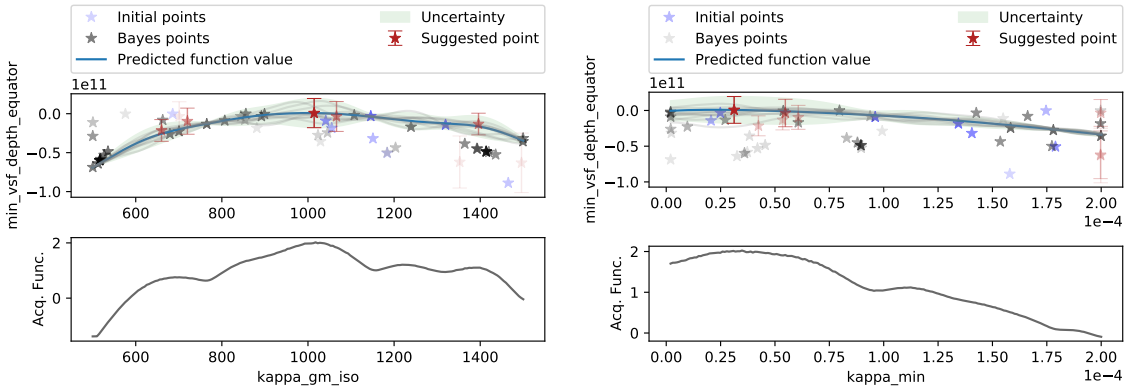


Figure 30: Prediction plots as in figure 29 but after refitting with different lengthscale bounds and with suggested points.

Now, we came very close to the correct simulation output, but let's take a look at the slightly worse parameter values. They are probably explained just by the worse model fit than in the first simulation, but there could be a few other things. First off, our range for $\min(\kappa_h)$is $[2e-6, 2e-4]$ and we are working in a linear space. This means that the largest scale $(e-4)$ takes up almost all of our space. This means that our initial steps will be largely biased towards this scale and that our model doesn't differentiate much between $2.5e-5$ and $7.97e-5$. To overcome this problem, it could be an idea to implement a parameter transformation method in *veropt* so that a parameter where the goal is to find the right scale, would be represented more reasonably, e.g. by taking the logarithm of it before it enters the optimiser's methods.

Another thing is, of course, that we're now in two dimensions. We saw in the first simulation that our data formed a nice, simple parabola with a single maximum. How does it look in this simulation? To answer this we look at a 3D plot of the data and model, shown in figure 31 [1].

All the same, we see how the data from the simulation forms a two dimensional parabola (left) and we see how we at a given projection onto the parameter axes get a one dimensional parabola. Thus, we see that there is a linear combination of the two parameters that will give us roughly the same objective function value. This means that we can't necessarily expect to retrieve the exact parameter values from our reference experiment, since there might be an

1 Note that these are unfortunately saved figures from an old optimisation run (all of the optimisation runs were rerun before the finalisation of the project, because of the problems with saving mentioned in chapter 4 and changes to the default methods of the package) and can't be easily updated because the methods to create 3D plots haven't been updated after the *veropt* package was upgraded to support multiple objectives.

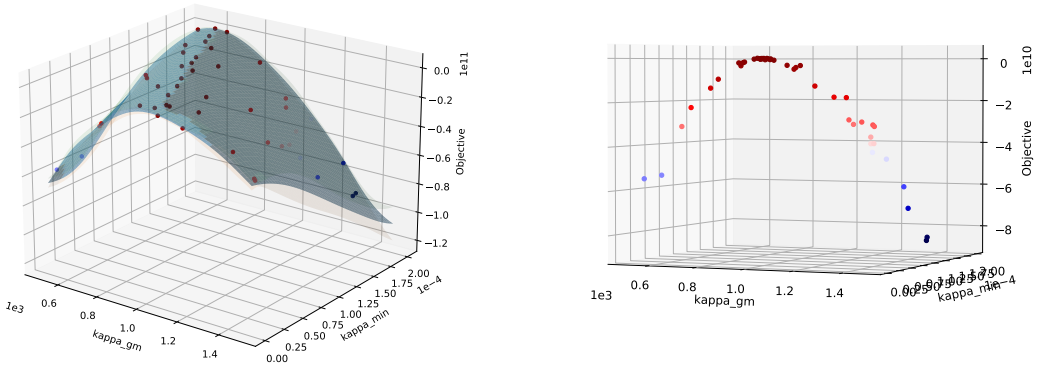


Figure 31: Three-dimensional figure from the optimisation of the second simulation. We see that the points from a parabola. Note that this is an old figure from an old run and that's why the labels are so small. See footnote for details.

infinite amount of parameters that produce the same simulation outcomes, as long as they're along that line.

Still, there might be a slight maximum along the top of the parabola, so to explore the possibilities for inspecting the method, changing parameters and running new points with the changes, we try to run one more step (we have run 6 so far) and see if we see an improvement. Since the parameter space is still pretty small with just two parameters and relatively narrow bounds, we set $\alpha = 0.5$ as well. The resulting suggested points can be seen in figure 30. The mean shows the expected values of them and the variance shows the model variance at their location.

After this extra step, we got parameter values of $\kappa_j = 1014.4$ and $\min(\kappa_v) = 3.12e - 5$, so it would seem our adjustments have worked! The relative error of the simulation output to the target value was 0.0020.

Of course, this adjustment could have been done earlier in the optimisation process so an extra step wouldn't have been necessary, and if this optimisation had been for a more complex simulation with a longer runtime, one would have lost of time between steps to check in on the optimisation and make sure everything was running smoothly.

In figure 28 we see the objective function value at the different points evaluated. Again, we don't see a large difference, possibly because the model was wrongly fitted. At the last eight points we do see that the majority of them are better than the preceding ones while

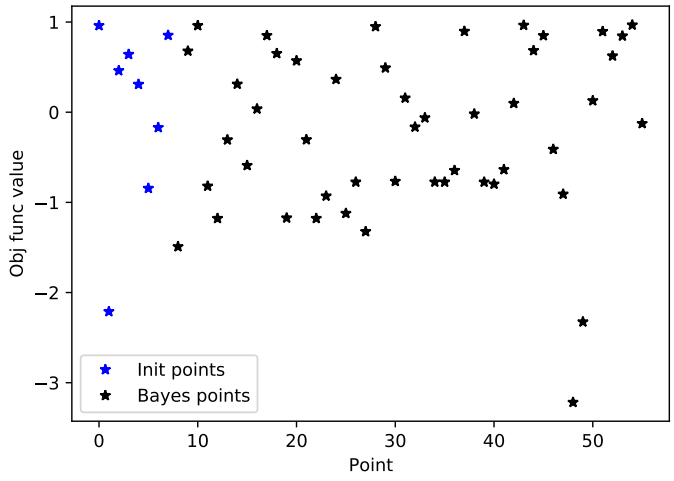


Figure 32: Progress plot for the second simulation.

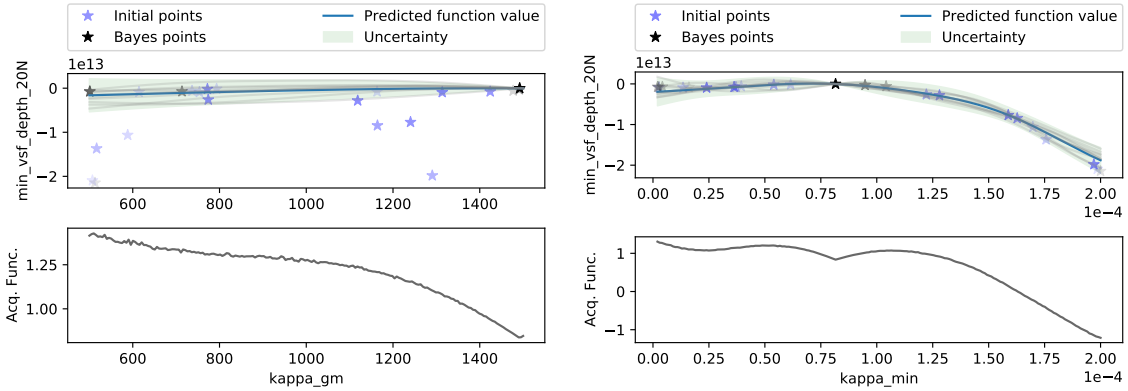


Figure 33: Prediction plot for simulation three at step 3, after the model has been refitted with wider lengthscale bounds.

two of them are outliers. If we look at figure 30 we can see that two of the suggested points were selected because of their large uncertainty. Those are likely the ones we see scoring low in the progress plot.

7.3 SIMULATION THREE

In the third simulation we ran 16 initial points and 48 points with Bayesian optimisation. Again, we ran 8 points per round.

At step 3, we again saw that the lengthscale was too small, giving overly large variance and oscillations in the posterior functions, but this time it was because the upper bound was limiting the MLL optimisation from going higher than 2.0. The upper bound was therefore changed to 5.0 and the model was fitted to lengthscales of $[5.00, 5.00, 1.22]$ for parameters κ_{iso},

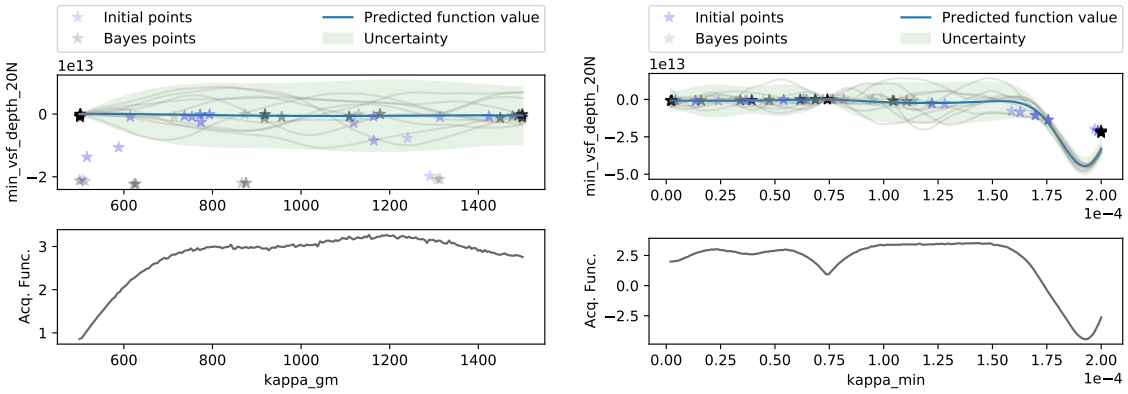


Figure 34: Prediction plot for simulation three, after the final step. We note that the lengthscale has been fitted too small.

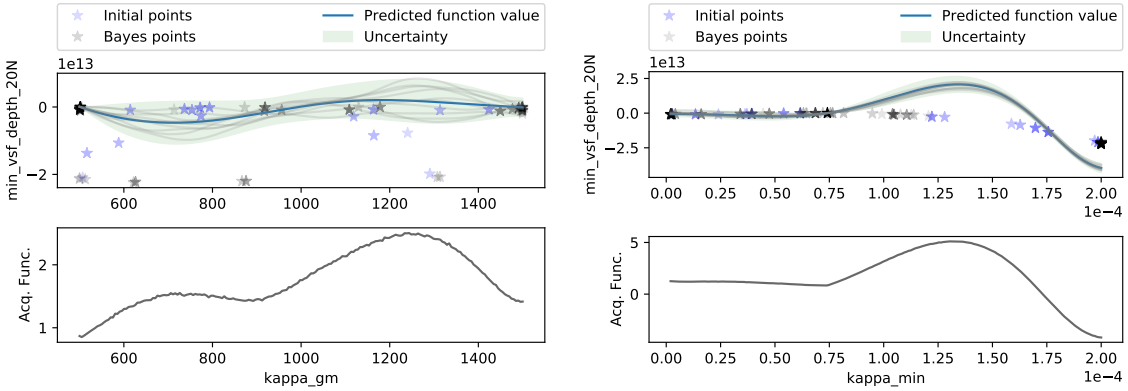


Figure 35: Prediction plot for simulation three after the final step. The model has been refitted with lengthscale bounds $[1.0, 5.0]$ but behaves erratically.

κ_{GM} and $\min(\kappa_v)$ respectively. The updated optimiser file was then uploaded to the cluster where the optimisation was running and took effect from step 4 and onwards. In figure 33 we see the model after the refit. Note that we only plotted two out of the three parameters. This is partially for practical reasons but was chosen because it seems that neither κ_{iso} or κ_{gm} has a strong dependence in the simulation outcome we're investigating, at least not at the range of well-performing points that were found.

After the eigth and final step, however, the model had some problems again. In figure 34, we see how the model has a lengthscale that is too low again. This time, however it's not as easy to fix. It turns out that if we change the lower bound to e.g. 1.0 and refit, we get a large oscillation that is not represented in the data, like we saw for the RBF kernel in chapter 2. Looking at the data as a function of $\min(\kappa_v)$, this is likely because the data points lie in an almost perfectly horizontal line until $\min(\kappa_v) = 0.75e-4$ where they dive down. As we know from chapter 2, the Matern kernel expects data that varies along roughly the same

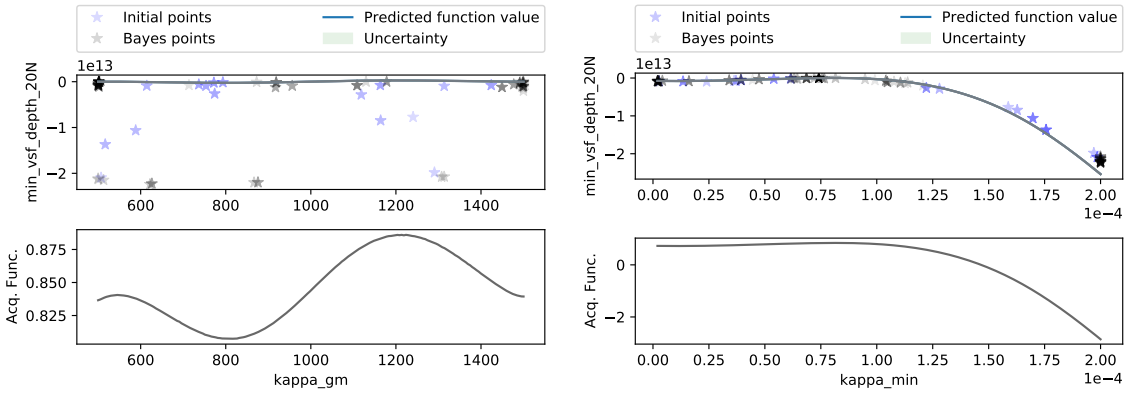


Figure 36: Prediction plot for simulation three after the final step. The model has been refitted with lengthscale bounds $[10.0, 12.0]$ and fits correctly but with too little variance.

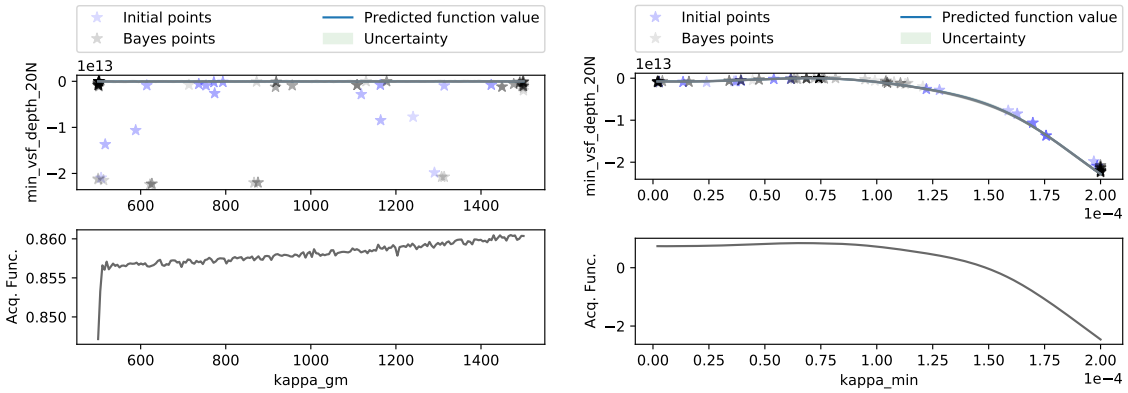


Figure 37: Prediction plot for simulation three after the final step. The model has been refitted with a spectral mixture kernel (SMK).

lengthscale so if we have a section first that has a very, very long lengthscale and then one with a much smaller one, we might not be able to fit it well. We note that this is the first time we've seen a model critically malfunction, prediction high objective function values where the data doesn't support it, and it only did it after we forced the lengthscale up.

If we turn the lower bound of the lengthscale all the way up to 10.0 (see figure 36), we interestingly see these oscillations vanishing, but we also see that the variance in the model almost vanishes, which means that the small variations in the mean in κ_{GM} and κ_{iso} dominate in the acquisition function and would lead to some areas being more favoured for candidate points than they probably should be.

In this case, it seems then that the Matern kernel isn't necessarily a good choice, and so we simply choose a different kernel. In chapter 2 we briefly discussed the SMK kernel [42], which offers more advanced data fitting capabilities. Changing the model of an ongoing optimisation can be done with

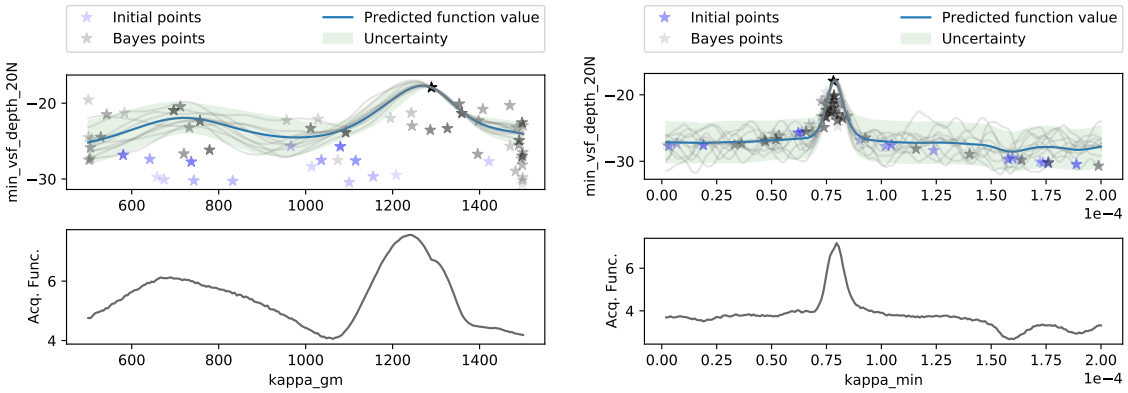


Figure 38: Prediction plot for simulation three after the final step, here using a logarithmic distance measure.

```
optimiser.set_new_model(BayesOptModel(n_params, n_objs, SMKModelBO))
```

(where optimiser is a loaded *BayesOptimiser* class).

In figure 37 we see that this indeed performs better, showing a larger variance in κ_{GM} and therefore not being as sensitive to small deviations in the mean. Here we also see why UCB_{Var} (see chapter 2) can be useful; The model is largely flat with a small uncertainty but has a small tilt. The slight random noise helps making sure the rightmost value of κ_{GM} isn't the only one that gets sampled.

In this simulation we didn't use a reference experiment, so we can't say which parameter values we should be getting. The best point had the coordinates $\kappa_{iso} = 849.2$, $\kappa_{GM} = 501.8$ and $\min(\kappa_v) = 7.4e-5$. The relative error to the target value was 0.0037.

7.3.1 *With A Logarithmic Difference Measure*

As mentioned, we've used a quadratic difference measure through all of these simulations, setting a goal for getting the right simulation output in the first couple of digits, but not going for high precision. As mentioned, this is useful when having lots of different parameters and objectives and trying to get within an acceptable region for all of them. Or it could be the right choice when the target value isn't known to higher precision anyways, or there might not be complete agreement between the real-world measured quantity and the simulation output.

To test the method in a setting where we do want high precision in one or more objectives, we now try to take the logarithm of our quadratic difference measure, rewarding the optimisation linearly with every correct digit it finds.

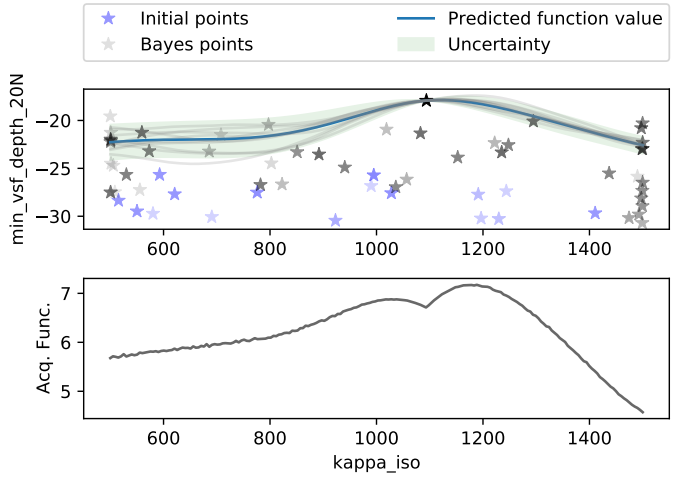


Figure 39: Prediction plot for simulation three after the final step, here using a logarithmic distance measure.

This means that our objective function will in fact be diverging, taking off towards infinity as the difference approaches zero. Optimally, our model should see this and simply fit it as a sharp peak (with a finite height).

At step 4, we saw that the lengthscale bounds were too tight, just like in the previous optimisation run with this simulation, so the bounds were expanded to $[0.1, 5.0]$.

In figures 38 and 39 we see the model and data after the final step of the optimisation. We see that with the logarithmic difference measure, the objective function landscape is much less flat along all three parameter axes, which is why we now included both κ_{GM} and κ_{iso}. The strongest dependence is still for $\min(\kappa_v)$, where we see a very sharp peak. Here we are probably approaching an exact value of $15.0Sv$ of the water transport and so we see the difference between the target and output approaching zero, causing the objective function value to shoot up, because of the logarithm involved in calculating it.

We note that the model seems to have fitted well here, even if the low lengthscale along the $\min(\kappa_v)$ only fits well right at the peak. This isn't an issue in this case, since we only have one maximum, but for other problems, this could be problematic. An idea could then be to fit a sum of two Matern kernels where one has the low lengthscale that fits the peak and another has a larger lengthscale to fit the other areas.

For this optimisation run, we got parameter values $\kappa_{iso} = 1093.7$, $\kappa_{GM} = 1289.4$ and $\min(\kappa_v) = 7.8169e-5$. We got a relative difference from the target value of 0.0005, so about an order of magnitude smaller than we did when using the quadratic difference measure.

In figure 40, on the left side, we see the progress plot for the optimisation of simulation three with the quadratic difference measure. We see that we've largely been sampling from

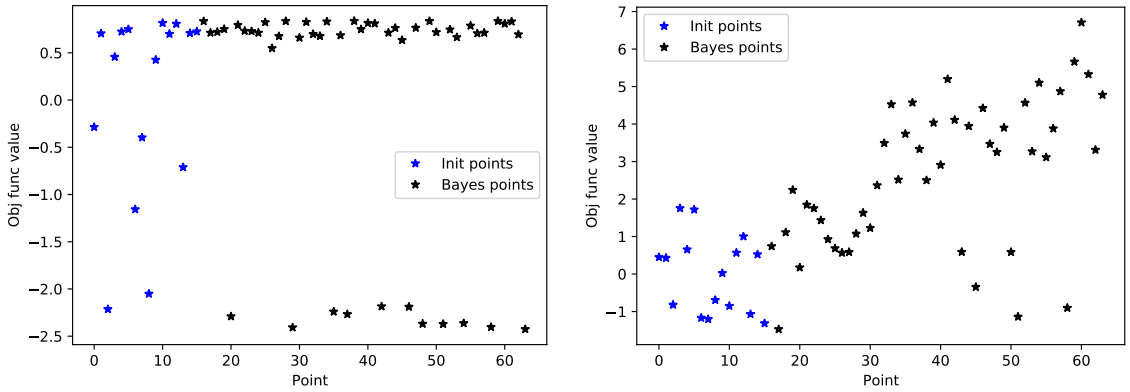


Figure 40: Progress plots for simulation three. On the left with quadratic difference measure and on the right with the logarithm of the quadratic difference measure.

the area with high objective function values, having a very different distribution than the initial, random points. We also see that there are outliers, again probably because there wasn't "space" for eight simultaneous points with the α value we chose ($\alpha = 1.0$ since we used default choices). In the same figure, on the right, we see the progress plot when the logarithmic difference measure was used. Here we see steady progress, as the optimisation digs into the maximum, getting more and more digits correct. This is the kind of progress plot we expect (or hope, at least) to see when we encounter a problem that can't easily be solved by random search.

7.4 OVERALL EVALUATION

The first thing to remark upon here must be that the optimisation problems provided by the chosen ocean simulations were too simple and too easily solved. All of these problems could have been solved by using random search, except for the very last one, when trying to get higher precision on the tuning of the measured simulation output. As mentioned, the original goal of the project was to include a fourth simulation with biogeochemistry and two objectives. This would probably have presented a more challenging problem, more in tune with what the tool was designed for.

As it is, the problems we had were perhaps a little underwhelming as to really establish the potential of the method, since we couldn't see a large advantage over random search in most cases, like we easily could in the test problems.

We have, however, demonstrated how an ongoing optimisation can be inspected with the visualisation tools and how it can be adjusted easily in the GUI.

We might also have seen that the Matern kernel could be a bit too simple in some situations and that for some optimisation problems, different kernels might be needed. We have also seen that the optimisation of the kernel hyperparameters might not be quite as robust as we would like and future development on this project might want to put some work into improving this aspect of the method, perhaps by implementing cross-validation or something similar.

We should note, however, that some of these findings might also change when presented with more challenging problems. For example, simple problems warrant fewer initial evaluations, giving our model less data for the initial fit, possibly making the lengthscale estimation harder, since the MLL is then less peaked. We also saw how doing eight points at each round was probably too much for these simple problems, but we don't expect that to be an issue at larger complexity, where the parameter space is bigger and there are probably more potential optima to investigate.

Either way, the good thing about Bayesian optimisation is, that no matter what challenges we might encounter, we have to remember that very nearly every part of the method can be exchanged for another one. As such, we don't even have to use Gaussian process regression for our surrogate model, we just need something that returns a mean and a variance. (Except, actually, we could define an acquisition function that doesn't use an uncertainty measure so we don't even need that.)

The challenge then, can be to find a model that is a good default option, but because the user can easily get involved and inspect the optimisation, it isn't strictly necessary to find one model that will *never* fail us. And as we saw here, it was pretty easy to fix whatever shortcomings we saw when inspecting.

Note that the optimisation files for these runs are available in the *veropt* project GitHub, so they can easily be loaded and inspected via the GUI or through a Python console, if the reader wants to further inspect the models or the data acquired by the optimisation. [2]

2 Note that the loading can be a little unstable because of different package versions in different Python environments and that you will need to have the ocean objective class imported or in the same folder (it can be found in the same GitHub directory).

Part V

CONCLUSION AND FUTURE WORK

8

CONCLUSION

In this thesis, a flexible, user-friendly optimisation tool utilising Bayesian optimisation was developed. The tool has been made available in a Python package called *veropt* and is geared towards cases where the objective function takes a long time to evaluate and there is no noise in the objective function evaluations, such as when tuning an ocean simulation.

While going through the theory of Bayesian Optimisation, we proposed two new extensions to the UCB acquisition function and a simple way to optimise the acquisition function when multiple candidate points are required in each round of the optimisation.

A default set-up of the method was created through the ideals of simplicity and adjustability, and this default set-up was tested, first on a couple of test problems and then on three increasingly complex ocean simulations.

In two of these test problems we saw very clear advantage of using our method over using random search and in the last one we saw a slight advantage. At this point of the project we also tested out different parameter values for the proposed optimisation of the acquisition function and saw that our intuitive guess of setting $\alpha = 1.0$ and $\beta = 1.0$ seemed sensible.

With the optimisation problems provided by the ocean simulations, we demonstrated how the tool can be used for easily inspecting and adjusting the method to ensure that the optimisation is always running as it should. We also saw that the optimisation of the model hyperparameters might need to be made more robust and that the default kernel (the Matern kernel) might be a little too simple to fit all the objective function landscapes we might encounter.

All of the ocean simulation were tuned so that there was only a small relative error (on the order of 0.001) between the measured output and the target value, but they turned out to be too simple to really show the advantage of running the method, since a similar result could likely have been achieved through random search. An exception is the last optimisation run that was performed, where the third simulation was tuned again, but now with a different

measure of the difference between the target value and the simulation output. Making this difference logarithmic, resulted in a sharper peak in the objective function, which allowed the method to converge towards higher and higher precision of the tuning.

Overall, we have created a strong foundation for an optimisation tool that offers a strong default set-up for solving a wide range of problems, large transparency into its inner workings and easy adjustment of many parts, while maintaining large modularity in the code, creating nearly endless opportunities to improve every part of it in the future.

9

FUTURE WORK

Given the many parts of the used method and the large flexibility of each of their forms, there are nearly endless ways the tool could be developed and improved.

The fact that it is also an ambition to make it user-friendly opens up even more possibilities. Here we will go through a few of them.

9.1 THE PYTHON PACKAGE

Ultimately, the *veropt* package could be developed into a complex software suite with a multitude of visualisation tools and tools for adjusting the current model or choosing a new one. For example, there could be a list of implemented models to choose from within the GUI and the fit of each one could be shown to the user. There could also be the option to add or multiply any of the kernels and supply individual constraints for the hyperparameters of all of them. Of course, the aim should still be to make the method easily accessible and easy to understand, so the more complex options should only be made available to the advanced user when they want them.

9.2 PRIOR INFORMATION FROM OCEAN PHYSICS

In this thesis, we mainly used the default set-up, but it could be interesting to look into how expert knowledge could be included in the optimisation, either by including the kind of prior information of the parameters that is always supported by the code, or by trying to guess the best model by applying knowledge from the field.

9.3 SPACE DESIGN, SPACE WARPING KERNEL AND MORE

In chapter 2, a number of theoretical methodologies were discussed that either weren't implemented or weren't tested out in either test experiments or ocean experiments. For example, we could mention the various options for space design that could be used to find the initial points, which is not currently implemented in *veropt* or the *warped kernel* method that was implemented but not tried out.

In future work on the project it would be interesting to implement and/or try out these methodologies.

9.4 TWO DIMENSIONAL OBJECTIVES

Many of the simulation outputs that we might be interested in tuning are defined in more than one spatial coordinate. Indeed, the water transports of the AMOC that we inspected in this thesis are many-dimensional too and we worked with them in one dimension for the simplicity of it and because the water transport is circular and must transport a constant amount of water along its circuit.

For some outputs, however, the spatial dependence and distribution is critically important and must be taken into account. This means that we might not be able to simply tune a minimum or mean at some specific coordinate, but will have to consider the entire matrix or tensor of the entity that we're tuning.

To efficiently do so, we would need to implement a model that could predict this distribution. Of course, we *could* simply compare the simulation's distribution to the target distribution and collapse this into a single number to feed into our current system, but in doing so the method would only see a single value for the overall mismatch, instead of seeing the distribution of it and thus getting more information to use to predict what the optimal parameter values might be.

After setting up this higher dimensional model, we can *then* take the difference to the target distribution and collapse it into a scalar objective function value to use with our already developed method.

The options for this higher dimensional model are endless and could be something like a artificial neural network built for regression, but we could also stick with a method that we've already worked with in this thesis: Gaussian process regression. To use a GP for this, we would need to add every point of the grid as the data that we fit, effectively adding the

spatial coordinates to our list of parameters. Now, this would of course add substantially to the amount of data going through the GP, but luckily *gpytorch* offers many possibilities for running GP's fast. And if it turns out to be too slow, we can always go for a different option insted.

The advantage with using a GP is that we would get an uncertainty measure as well.

Implementing this into the method would probably greatly improve its capability for tuning complex ocean simulations with complex objectives, such as biogeochemistry models.

Part VI

APPENDIX

10

APPENDIX

10.1 EXAMPLES

10.1.1 *Multiple Objectives, Vehicle Safety Test Function*

Example of optimisation run with the VehicleSafety test function with manual setting of the acquisition function and model to show how that can be done.

```
from veropt import BayesOptimiser
from veropt.obj_funcs.test_functions import *
from veropt.acq_funcs import *
from veropt.kernels import *
from veropt.gui import veropt_gui

n_init_points = 16
n_bayes_points = 64
n_evals_per_step = 4

obj_func = PredefinedTestFunction("VehicleSafety")
n_objs = obj_func.n_objs

acq_func = PredefinedAcqFunction(obj_func.bounds, n_objs,
    ↪ n_evals_per_step, acqfunc_name='EHVI', seq_dist_punish=True,
    ↪ alpha=1.0, omega=1.0)

model_list = n_objs * [MaternModelBO]
```

```
17 kernel = BayesOptModel(obj_func.n_params, obj_func.n_objs,
    ↪ model_class_list=model_list, init_train_its=1000,
    ↪ using_priors=False)
18
19 optimiser = BayesOptimiser(n_init_points, n_bayes_points,
    ↪ obj_func, acq_func, model=kernel,
    ↪ n_evals_per_step=n_evals_per_step)
20
21 veropt_gui.run(optimiser)
```

10.1.2 *Ocean Objective*

The objective function class for the third ocean simulation. We set the bounds of the parameters, the amount and names of the parameters and objectives. Then we at what year the simulation output will be measured and set a target value. We then set the method for calculating the objective function value by telling which simulation output file to look at (*filetype*) and which parameters (*param_dic* containing the *measure_year* and target value) to use for this calculation. The function *calc_y* is written as a static method of the class and works by loading the data saved by *veros* simulations as *xarray* DataFrames and then doing some user-defined calculation with them. Here we load the output *vsf_depth* from the *overturning* output file and find the vertical minimum at 20°N at year *measure_year*.

```
1 class OceanObjSimThree(OceanObjFunction):
2     def __init__(self, target_min_vsf_depth_20N,
      ↪ measure_year=100, file_path=None):
3         bounds_lower = [500, 500, 2e-6]
4         bounds_upper = [1500, 1500, 2e-4]
5         bounds = [bounds_lower, bounds_upper]
6         n_params = 3
7         n_objs = 1
8         var_names = ["kappa_iso", "kappa_gm", "kappa_min"]
9         obj_names = ["min_vsf_depth_20N"]
```

```
10
11          self.measure_year = measure_year
12          self.target_min_vsf_depth_20N =
             ↪  target_min_vsf_depth_20N
13          self.file_path = file_path
14
15          param_dic = {
16              "measure_year": measure_year,
17              "target_min_vsf_depth_20N":
                 ↪  target_min_vsf_depth_20N}
18
19          filetype = "overturning"
20
21          calc_y_method = (self.calc_y, filetype, param_dic)
22
23          super().__init__(bounds=bounds, n_params=n_params,
             ↪  n_objs=n_objs, calc_y_method=calc_y_method,
             ↪  var_names=var_names, obj_names=obj_names,
             ↪  file_path=file_path)
24
25      @staticmethod
26      def calc_y(overturning, param_dic):
27          min_vsf_depth_20N =
             ↪  float(overturning["vsf_depth"][param_dic["measure_year"]
             ↪  - 1].min("zw")[25])
28          y = - (min_vsf_depth_20N -
             ↪  param_dic["target_min_vsf_depth_20N"])**2
29          return y, min_vsf_depth_20N
```

Here we show the python script to set up an optimisation run for an ocean simulation with the default optimisation methods. We see that it resembles the set-up required for a test function, except we use the ocean objective function class instead and supply a *measure_year* and target value.

```
from veropt import BayesOptimiser
from veropt.obj_funcs.ocean_sims import OceanObjFunction

class OceanObjSimThree(OceanObjFunction): ...

n_init_points = 16
n_bayes_points = 48

n_evals_per_step = 8

measure_year = 100
target_min_vsf_depth_20N = -15 * 10**6

obj_func = OceanObjSimThree(target_min_vsf_depth_20N,
    ↪ measure_year=measure_year)

optimiser = BayesOptimiser(n_init_points, n_bayes_points,
    ↪ obj_func, n_evals_per_step=n_evals_per_step)

optimiser.save_optimiser()
```

10.1.2.1 *Modifying the veros .py file*

To use the *saver()* and *loader()* method of the superclass *OceanObjFunction* we need to make a few modifications to the *veros* set-up file.

We do two additional imports:

```
import click
from veropt.obj_funcs.ocean_sims import load_data_to_sim
```

And then we add some *click* decorators to the *run()* method and use the *load_data_to_sim()* method which loads the right parameter values for this specific run of the ocean simulation by using its id.

```
@veros.tools.cli
@click.option('--identifier', default=None, is_flag=False,
              help='Sets the identifier')
@click.option('--optimiser', default=None, is_flag=False,
              help='Path to the optimiser file')
def run(*args, **kwargs):
    global identifier
    global var_vals

    identifier, var_vals, kwargs = load_data_to_sim(kwargs)

    simulation = GlobalFourDegreeSetup(*args, **kwargs)
    simulation.setup()
    simulation.run()
```

Finally, we change the file name to include the id so the *loader()* method used inside the optimisation can identify the output files and know which parameter values they were run with, and set the relevant parameters to the desired values.

```
def set_parameter(self, vs):

    global identifier
    global var_vals

    vs.identifier = '4deg_id_' + str(identifier)
    ...
    vs.K_iso_0 = var_vals["kappa_iso"]
    ...
    vs.kappaH_min = var_vals["kappa_min"]
    ...
    vs.K_gm_0 = var_vals["kappa_gm"]
    ...
```

Note that the ellipsis ... is meant to signify hidden code and not the python object *Ellipsis*

In future versions of *veropt* it might be desirable to make this process more automatic.

10.1.3 *Slurm Support*

The following example sets up an optimisation run on the cluster *MODI* by creating a copy of the *slurm_controller.py* script and creating shell files for both that script and the *veros* simulation. The last line runs an *sbatch* command and starts the optimsation run. This works by running the *slurm_controller* which then submits further *sbatch* commands, on for each simulation run in each round.

This method can be used for other clusters of course, but the *slurm_set_up* has only been tested on *MODI* and the batch files might need editing before they are ready for other systems.

```
1 slurm_set_up.set_up(
2     optimiser.file_name, ["modi_long", "modi_short"],
      ↪ "global_four_degree.py", make_new_slurm_controller=True,
      ↪ using_singularity=True,
      ↪ image_path="~/modi_images/hpc-ocean-notebook_latest.sif",
      ↪ conda_env="python3")
3
4 slurm_set_up.start_opt_run("modi001")
```

BIBLIOGRAPHY

[1] Laurits S Andreasen. "Time scales of the Bipolar seesaw: The role of oceanic cross-hemisphere signals, Southern Ocean eddies and wind changes". en. In: (2019), p. 46. URL: https://www.nbi.ku.dk/english/theses/masters-theses/laurits-s.-andreasen/Laurits_Andreasen_MSc_thesis.pdf.

[2] *BoTorch · Bayesian Optimization in PyTorch*. en. URL: https://botorch.org/ (visited on 04/16/2021).

[3] P. Bougeault and P. Lacarrere. "Parameterization of Orography-Induced Turbulence in a Mesobeta–Scale Model". EN. In: *Monthly Weather Review* 117.8 (Aug. 1989). Publisher: American Meteorological Society Section: Monthly Weather Review, pp. 1872–1890. ISSN: 1520-0493, 0027-0644. DOI: 10.1175/1520-0493(1989)117<1872:POOITI>2.0.CO;2. URL: https://journals.ametsoc.org/view/journals/mwre/117/8/1520-0493_1989_117_1872_pooiti_2_0_co_2.xml (visited on 05/10/2021).

[4] Yongtao Cao, Byran J. Smucker, and Timothy J. Robinson. "On using the hypervolume indicator to compare Pareto fronts: Applications to multi-criteria optimal experimental design". en. In: *Journal of Statistical Planning and Inference* 160 (May 2015), pp. 60–74. ISSN: 0378-3758. DOI: 10.1016/j.jspi.2014.12.004. URL: https://www.sciencedirect.com/science/article/pii/S0378375814002006 (visited on 04/16/2021).

[5] *Convection–diffusion equation*. en. Page Version ID: 997508655. Dec. 2020. URL: https://en.wikipedia.org/w/index.php?title=Convection%E2%80%93diffusion_equation&oldid=997508655 (visited on 05/07/2021).

[6] *dill: serialize all of python*. URL: https://pypi.org/project/dill (visited on 05/16/2021).

[7] David Duvenaud. *Kernel Cookbook*. 2014. URL: https://www.cs.toronto.edu/~duvenaud/cookbook/ (visited on 04/16/2021).

[8] Carsten Eden and Richard J. Greatbatch. "Towards a mesoscale eddy closure". en. In: *Ocean Modelling* 20.3 (Jan. 2008), pp. 223–239. ISSN: 1463-5003. DOI: 10.1016/j.ocemod.2007.09.002. URL: https://www.sciencedirect.com/science/article/pii/S1463500307001163 (visited on 05/09/2021).

[9] *Ekman transport*. en. Page Version ID: 1006523953. Feb. 2021. URL: https://en.wikipedia.org/w/index.php?title=Ekman_transport&oldid=1006523953 (visited on 05/10/2021).

[10] M. T. M. Emmerich, K. C. Giannakoglou, and B. Naujoks. "Single- and multiobjective evolutionary optimization assisted by Gaussian random field metamodels". In: *IEEE Transactions on Evolutionary Computation* 10.4 (Aug. 2006). Conference Name: IEEE Transactions on Evolutionary Computation, pp. 421–439. ISSN: 1941-0026. DOI: 10.1109/TEVC.2005.859463.

[11] Matthias Feurer and Frank Hutter. "Hyperparameter Optimization". en. In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. The Springer Series on Challenges in Machine Learning. Cham: Springer International Publishing, 2019, pp. 3–33. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_1. URL: https://doi.org/10.1007/978-3-030-05318-5_1 (visited on 04/16/2021).

[12] Philippe Gaspar, Yves Grégoris, and Jean-Michel Lefevre. "A simple eddy kinetic energy model for simulations of the oceanic vertical mixing: Tests at station Papa and long-term upper ocean study site". en. In: *Journal of Geophysical Research: Oceans* 95.C9 (1990). _eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/JC095iC09p16179, pp. 16179–16193. ISSN: 2156-2202. DOI: https://doi.org/10.1029/JC095iC09p16179. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JC095iC09p16179 (visited on 05/10/2021).

[13] Peter Gent and JC McWilliams. "Isopycnal Mixing in Ocean Circulation Models". In: *Journal of Physical Oceanography* 20 (Jan. 1990), pp. 150–155. DOI: 10.1175/1520-0485(1990)020<0150:IMIOCM>2.0.CO;2.

[14] Peter Gent et al. "Parameterizing Eddy-Induced Tracer Transports in Ocean Circulation Models". In: *Journal of Physical Oceanography* 25 (Apr. 1995), pp. 463–474. DOI: 10.1175/1520-0485(1995)025<0463:PEITTI>2.0.CO;2.

[15] *GPflowOpt 0.1.1 documentation*. URL: https://gpflowopt.readthedocs.io/en/latest/intro.html (visited on 05/16/2021).

[16] *GPyTorch Regression Tutorial — GPyTorch 1.4.1 documentation*. URL: https://docs.gpytorch.ai/en/stable/examples/01_Exact_GPs/Simple_GP_Regression.html.

[17] *GPyTorch Regression Tutorial — GPyTorch 1.4.1 documentation*. URL: https://docs.gpytorch.ai/en/stable/examples/01_Exact_GPs/Simple_GP_Regression.html (visited on 05/16/2021).

[18] Robert B. Gramacy. *Surrogates: Gaussian Process Modeling, Design and Optimization for the Applied Sciences*. Chapman Hall/CRC, 2020. URL: http://bobby.gramacy.com/surrogates/.

[19] Stephen Griffies. "Fundamentals of Ocean Climate Models". In: (Jan. 2004).

[20] Stephen M. Griffies. "The Gent–McWilliams Skew Flux". EN. In: *Journal of Physical Oceanography* 28.5 (May 1998). Publisher: American Meteorological Society Section: Journal of Physical Oceanography, pp. 831–841. ISSN: 0022-3670, 1520-0485. DOI: 10.1175/1520-0485(1998)028<0831:TGMSF>2.0.CO;2. URL: https://journals.ametsoc.org/view/journals/phoc/28/5/1520-0485_1998_028_0831_tgmsf_2.0.co_2.xml (visited on 05/10/2021).

[21] Andreia P. Guerreiro, Carlos M. Fonseca, and Luís Paquete. "The Hypervolume Indicator: Problems and Algorithms". In: *arXiv:2005.00515 [cs]* (May 2020). arXiv: 2005.00515. URL: http://arxiv.org/abs/2005.00515 (visited on 04/16/2021).

[22] idax4325. *idax4325/veropt*. original-date: 2021-03-04T13:54:29Z. Mar. 2021. URL: https://github.com/idax4325/veropt (visited on 05/11/2021).

[23] Donald R. Jones, Matthias Schonlau, and William J. Welch. "Efficient Global Optimization of Expensive Black-Box Functions". en. In: *Journal of Global Optimization* 13.4 (Dec. 1998), pp. 455–492. ISSN: 1573-2916. DOI: 10.1023/A:1008306431147. URL: https://doi.org/10.1023/A:1008306431147 (visited on 04/16/2021).

[24] T. Kuhlbrodt et al. "On the driving processes of the Atlantic meridional overturning circulation". en. In: *Reviews of Geophysics* 45.2 (Apr. 2007), RG2001. ISSN: 8755-1209. DOI: 10.1029/2004RG000166. URL: http://doi.wiley.com/10.1029/2004RG000166 (visited on 05/05/2021).

[25] *Matérn covariance function*. en. Page Version ID: 995934658. Dec. 2020. URL: https://en.wikipedia.org/w/index.php?title=Mat%C3%A9rn_covariance_function&oldid=995934658 (visited on 04/16/2021).

[26] Adele K. Morrison, Thomas L. Frölicher, and Jorge L. Sarmiento. "Upwelling in the Southern Ocean". In: *Physics Today* 68.1 (Dec. 2014). Publisher: American Institute of Physics, pp. 27–32. ISSN: 0031-9228. DOI: 10.1063/PT.3.2654. URL: https://physicstoday.scitation.org/doi/full/10.1063/PT.3.2654 (visited on 05/10/2021).

[27] *Optimization and root finding (scipy.optimize) — SciPy v1.6.3 Reference Guide*. URL: https://docs.scipy.org/doc/scipy/reference/optimize.html#least-squares-and-curve-fitting (visited on 05/16/2021).

[28] *pandas - Python Data Analysis Library*. URL: https://pandas.pydata.org/ (visited on 05/16/2021).

[29] *Positive-definite kernel*. en. Page Version ID: 994195860. Dec. 2020. URL: https://en.wikipedia.org/w/index.php?title=Positive-definite_kernel&oldid=994195860 (visited on 04/16/2021).

[30] *Prandtl number*. en. Page Version ID: 1021022434. May 2021. URL: https://en.wikipedia.org/w/index.php?title=Prandtl_number&oldid=1021022434 (visited on 05/10/2021).

[31] *PyTorch*. en. URL: https://www.pytorch.org (visited on 05/16/2021).

[32] Anil Ramachandran et al. "Incorporating expert prior in Bayesian optimisation via space warping". en. In: *Knowledge-Based Systems* 195 (May 2020), p. 105663. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2020.105663. URL: https://www.sciencedirect.com/science/article/pii/S0950705120301088 (visited on 04/16/2021).

[33] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. OCLC: ocm61285753. Cambridge, Mass: MIT Press, 2006. ISBN: 978-0-262-18253-9.

[34] *scipy.stats.truncnorm — SciPy v1.6.3 Reference Guide*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.truncnorm.html (visited on 05/16/2021).

[35] *Seawater*. en. Page Version ID: 1021376652. May 2021. URL: https://en.wikipedia.org/w/index.php?title=Seawater&oldid=1021376652 (visited on 05/05/2021).

[36] *Setup gallery — Veros 0+untagged.97.gba2abd2.dirty documentation*. URL: https://veros.readthedocs.io/en/latest/reference/setup-gallery.html.

[37] *sklearn.gaussian_process.kernels.Matern — scikit-learn 0.24.1 documentation*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.Matern.html (visited on 04/16/2021).

[38] *sklearn.gaussian_process.kernels.RBF — scikit-learn 0.24.1 documentation*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.RBF.html (visited on 04/16/2021).

[39] Henry Stommel. "Thermohaline Convection with Two Stable Regimes of Flow". en. In: *Tellus* 13.2 (1961). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.2153-3490.1961.tb00079.x, pp. 224–230. ISSN: 2153-3490. DOI: https://doi.org/10.1111/j.2153-3490.1961.tb00079.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.2153-3490.1961.tb00079.x (visited on 05/05/2021).

[40] *TensorFlow*. en. URL: https://www.tensorflow.org/ (visited on 05/16/2021).

[41] *Turbulence kinetic energy*. en. Page Version ID: 1015945877. Apr. 2021. URL: https://en.wikipedia.org/w/index.php?title=Turbulence_kinetic_energy&oldid=1015945877 (visited on 05/10/2021).

[42] Andrew Gordon Wilson and Ryan Prescott Adams. "Gaussian Process Kernels for Pattern Discovery and Extrapolation". In: *arXiv:1302.4245 [cs, stat]* (Dec. 2013). arXiv: 1302.4245. URL: http://arxiv.org/abs/1302.4245 (visited on 04/16/2021).

[43] Andrew Gordon Wilson et al. "Deep Kernel Learning". In: *arXiv:1511.02222 [cs, stat]* (Nov. 2015). arXiv: 1511.02222. URL: http://arxiv.org/abs/1511.02222 (visited on 04/16/2021).

[44] James T. Wilson et al. "The reparameterization trick for acquisition functions". In: *arXiv:1712.00424 [cs, math, stat]* (Dec. 2017). arXiv: 1712.00424. URL: http://arxiv.org/abs/1712.00424 (visited on 04/16/2021).

[45] Y Xiang et al. "Generalized simulated annealing algorithm and its application to the Thomson model". en. In: *Physics Letters A* 233.3 (Aug. 1997), pp. 216–220. ISSN: 0375-9601. DOI: 10.1016/S0375-9601(97)00474-X. URL: https://www.sciencedirect.com/science/article/pii/S037596019700474X (visited on 04/16/2021).

[46] Xin-She Yang. "Chapter 14 - Multi-Objective Optimization". en. In: *Nature-Inspired Optimization Algorithms*. Ed. by Xin-She Yang. Oxford: Elsevier, Jan. 2014, pp. 197–211. ISBN: 978-0-12-416743-8. DOI: 10.1016/B978-0-12-416743-8.00014-2. URL: https:

//www.sciencedirect.com/science/article/pii/B9780124167438000142 (visited on 04/16/2021).

[47] Boya Zhang, D. Austin Cole, and Robert B. Gramacy. "Distance-distributed design for Gaussian process surrogates". In: *arXiv:1812.02794 [stat]* (June 2019). arXiv: 1812.02794. URL: http://arxiv.org/abs/1812.02794 (visited on 04/16/2021).