Workshop 1: calibrating a deterministic model

Danny Scarponi, Andy Iskauskas

Contents

Chapter 1

Introduction to the model

This tutorial offers an interactive introduction to the main functionality of the HoMER package, using a synthetic example of an epidemiological model. In this workshop, you will be invited to carry out a series of tasks (see "Task" boxes) which will enhance your understanding of the package and its tools. Thanks to these activities, you will learn to calibrate deterministic models using history matching and model emulation, and to use your judgement to customise the process. Although self-contained, this tutorial should be considered as a natural continuation of Tutorial 1 (hyperlink), which gives a more general overview of the history matching with emulation process for deterministic models, and shows how to perform it using HoMER. Following Workshop 1, you may also want to read Tutorial 2 and Workshop 2, where we demonstrate how to calibrate stochastic models.

The model that we chose for demonstration purposes is a deterministic SEIRS model, described by the following differential equations:

$$\frac{dS}{dt} = bN - \frac{\beta(t)IS}{N} + \omega R - \mu S \tag{1.1}$$

$$\frac{dE}{dt} = \frac{\beta(t)IS}{N} - \sigma E - \mu E \tag{1.2}$$

$$\frac{dI}{dt} = \sigma E - \gamma I - (\mu + \alpha)I \tag{1.3}$$

$$\frac{dR}{dt} = \gamma I - \omega R - \mu R \tag{1.4}$$

where N is the total population, varying over time, and the parameters are as follows:

- b is the birth rate,
- μ is the rate of death from other causes,
- $\beta(t)$ is the infection rate between each infectious and susceptible individual,
- σ is the rate of becoming infectious after infection,

- α is the rate of death from the disease,
- γ is the recovery rate and
- ω is the rate at which immunity is lost following recovery.

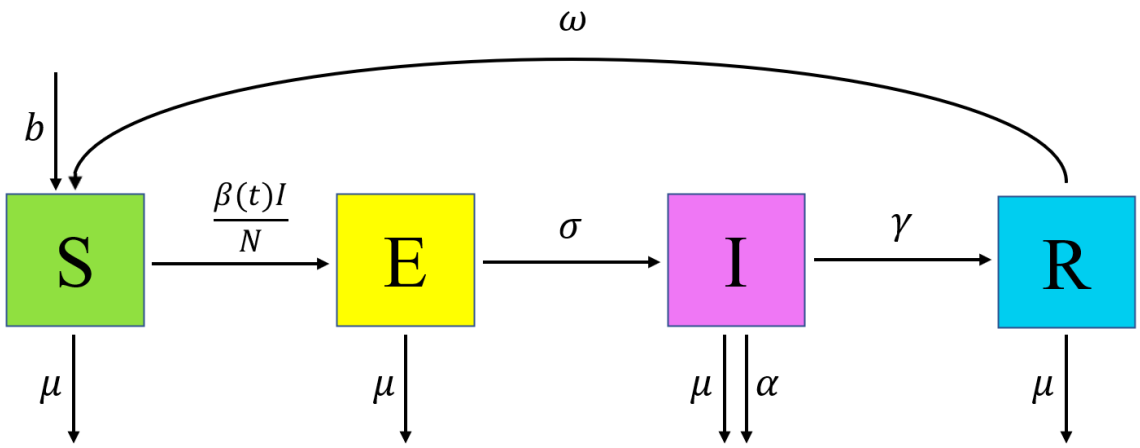


Figure 1.1: SEIRS Diagram

The rate of infection $\beta(t)$ is set to be a simple linear function interpolating between points, where the points in question are $\beta(0) = \beta_1$, $\beta(100) = \beta(180) = \beta_2$, $\beta(270) = \beta_3$ and where $\beta_2 < \beta_1 < \beta_3$. This choice was made to represent an infection rate that initially drops due to external (social) measures and then raises when a more infectious variant appears. Here t is taken to measure days. Below we show a graph of the infection rate over time when $\beta_1 = 0.3, \beta_2 = 0.1$ and $\beta_3 = 0.4$:

In order to obtain the solution of the differential equations for a given set of parameters, we will use a helper function, `ode_results`. The function assumes an initial population of 900 susceptible individuals, 100 exposed individuals, and no infectious or recovered individuals. Below we use `ode_results` with an example set of parameters and plot the model output over time.

```
example_params <- c(
  b = 1/(60*365),
  mu = 1/(76*365),
  beta1 = 0.2, beta2 = 0.1, beta3 = 0.3,
  sigma = 0.13,
  alpha = 0.01,
  gamma = 0.08,
  omega = 0.003
)
solution <- ode_results(example_params)
plot(solution)
```

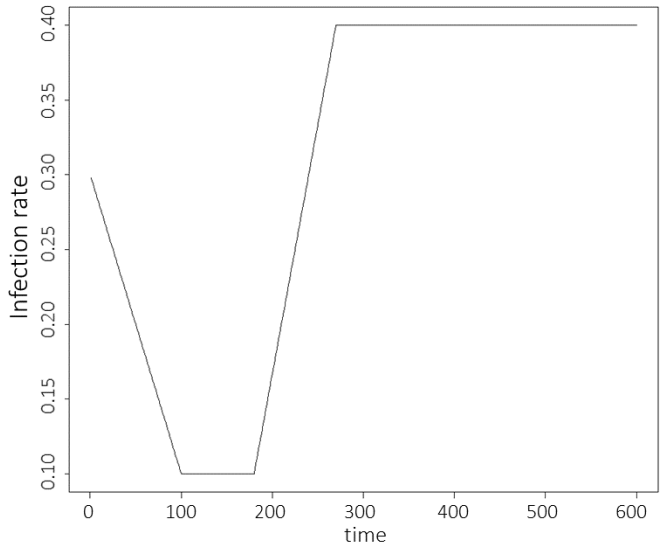
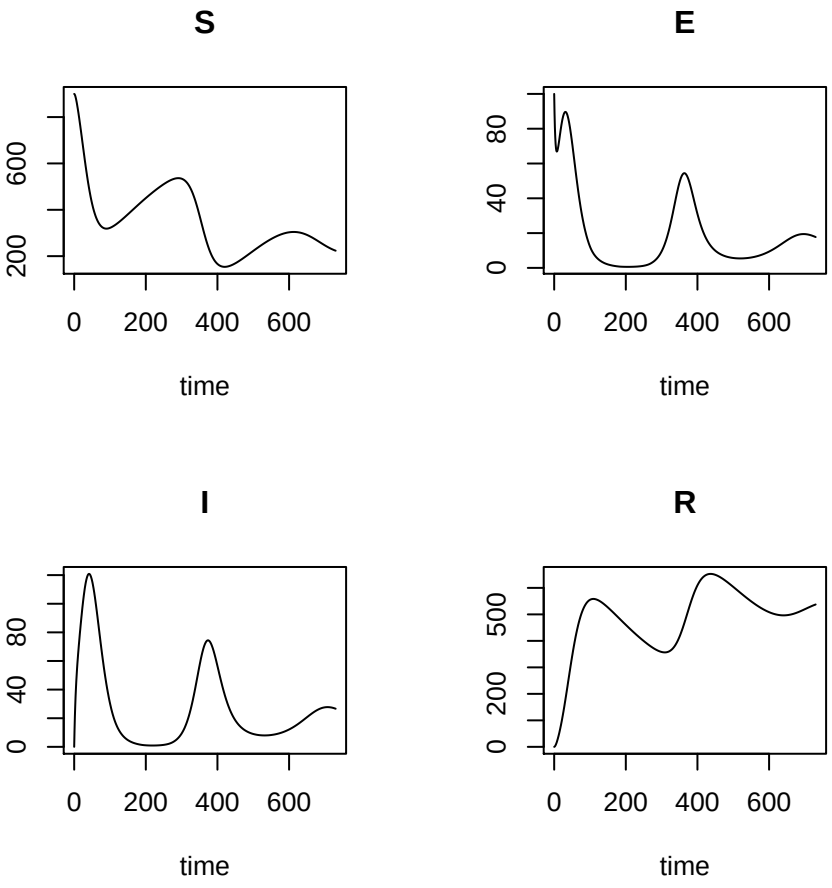


Figure 1.2: Infection rate graph



Task 1

Familiarise yourself with the model. Investigate how the plots change as you change the values of the parameters.

R tip

Show: Solution on P??

Chapter 2

Defining 'wave0' data

Before we tackle the emulation, we need to define some objects. First of all, let us set the parameter ranges:

```
ranges = list(
  b = c(1e-5, 1e-4), # birth rate
  mu = c(1e-5, 1e-4), # rate of death from other causes
  beta1 = c(0.2, 0.3), # infection rate at time t=0
  beta2 = c(0.1, 0.2), # infection rates at time t=100
  beta3 = c(0.3, 0.5), # infection rates at time t=270
  sigma = c(0.07, 0.21), # rate of becoming infectious after infection
  alpha = c(0.01, 0.025), # rate of death from the disease
  gamma = c(0.05, 0.08), # recovery rate
  omega = c(0.002, 0.004) # rate at which immunity is lost following recovery
)
```

We then turn to the targets we will match: the number of infectious individuals I and the number of recovered individuals R at times $t = 25, 40, 100, 200, 200, 350$. For each of these outputs, we define a pair (val, sigma), where 'val' represents the mean value of the output and 'sigma' represents its standard deviation:

```
targets <- list(
  I25 = list(val = 115.88, sigma = 5.79),
  I40 = list(val = 137.84, sigma = 6.89),
  I100 = list(val = 26.34, sigma = 1.317),
  I200 = list(val = 0.68, sigma = 0.034),
  I300 = list(val = 29.55, sigma = 1.48),
  I350 = list(val = 68.89, sigma = 3.44),
  R25 = list(val = 125.12, sigma = 6.26),
  R40 = list(val = 256.80, sigma = 12.84),
  R100 = list(val = 538.99, sigma = 26.95),
```

```
  R200 = list(val = 444.23, sigma = 22.21),
  R300 = list(val = 371.08, sigma = 15.85),
  R350 = list(val = 549.42, sigma = 27.47)
)
```

The 'sigmas' in our `targets` list represent the uncertainty we have about the observations. Note that in general we can also choose to include model uncertainty in the 'sigmas', to reflect how accurate we think our model is.

Show: More on how targets were set on P??

Finally we need a set of `wave0` data to start. When using your own model, you can create the dataframe 'wave0' following the steps below:

- build a named dataframe `initial_points` containing a space filling set of points, which can be generated using a Latin Hypercube Design or another sampling method of your choice. A rule of thumb is to select at least $10p$ parameter sets, where p is the number of parameters in the model. The columns of `initial_points` should be named exactly as in the list `ranges`;
- run your model on the parameter sets in `initial_points` and define a dataframe `initial_results` in the following way: the nth row of `initial_results` should contain the model outputs corresponding to the chosen targets for the parameter set in the nth row of `initial_points`. The columns of `initial_results` should be named exactly as in the list `targets`;
- bind `initial_points` and `initial_results` by their columns to form the dataframe `wave0`.

For this workshop, we generate parameter sets using a Latin Hypercube design (see fig. 2.1 for a Latin hypercube example in two dimensions).

Through the function `randomLHS` in the package `lhs` we create a hypercube design with 180 parameter sets for our nine inputs, 20 times the dimensionality of the parameter space. To access the external function `randomLHS`, we use the syntax `package::function()`:

```
initial_LHS <- lhs::randomLHS(180, 9)
```

Note that in `initial_LHS` each parameter is distributed on $[0, 1]$. This is not exactly what we need, since each parameter has a different range. We therefore re-scale each component in `initial_LHS` multiplying it by the difference between the upper and lower bounds of the range of the corresponding parameter and then we add the lower bound for that parameter. In this way we obtain `initial_points`, which contains parameter values in the correct ranges.

```
initial_points <- setNames(data.frame(t(apply(initial_LHS, 1,
                  function(x) x*unlist(lapply(ranges, function(x) x[2]-x[1])) +
                  unlist(lapply(ranges, function(x) x[1])))), names(ranges))
```

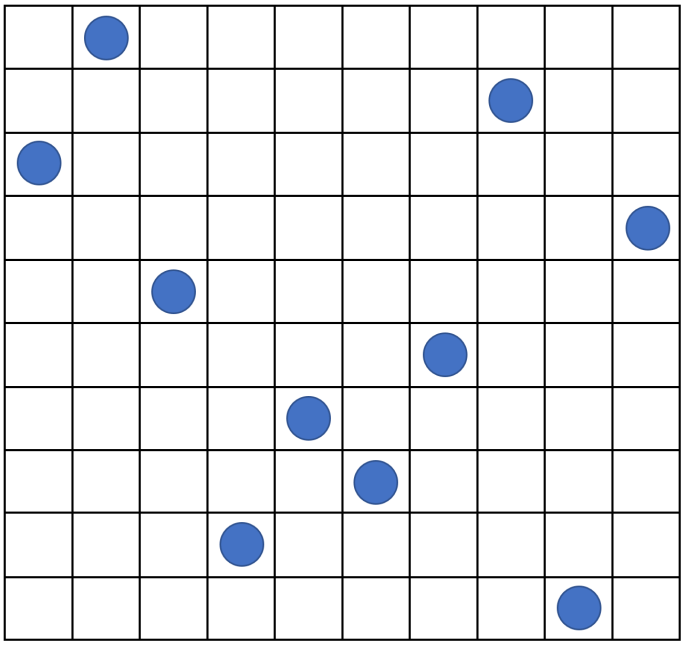


Figure 2.1: An example of Latin hypercube with 10 points in two dimensions: there is only one sample point in each row and each column.

We then run the model for the parameter sets in `initial_points` through the `get_results` function. This is a helper function that acts as `ode_results`, but has the additional feature of allowing us to decide which outputs and times should be returned: in our case we need the values of I and R at $t = 25, 40, 100, 200, 300, 350$.

```
initial_results <- setNames(data.frame(t(apply(initial_points, 1, get_results,
                    c(25, 40, 100, 200, 300, 350), c('I', 'R')))), names(targets))
```

Finally, we bind the parameter sets `initial_points` to the model runs `initial_results` and save everything in the data.frame `wave0`:

```
wave0 <- cbind(initial_points, initial_results)
```

Note that when using your own model, you can create the 'wave0' dataframe however you wish. The info box below will help you build `wave0` when working on your own model.

Chapter 3

Emulators

Let us start by splitting `wave0` in two parts: the training set, on which we will train the emulators, and a validation set, which will be used to do diagnostics of the emulators.

```
t_sample <- sample(1:nrow(wave0), 90)
training <- wave0[t_sample,]
validation <- wave0[-t_sample,]
```

3.1 A brief recap on emulators

Before building emulators, let us quickly remind ourselves what an emulator is and how it is structured. Note that a more detailed discussion about the structure of an emulator can be found in Tutorial 2 (Section 3, and Appendix A and B).

An emulator is a way of representing our beliefs about the behaviour of a function whose output is unknown. In this workshop what is unknown is the behaviour of our SEIRS model and we will emulate each of the model outputs separately. Given a training dataset, i.e. a set of model runs, we can train an emulator and use it to get expectation and variance for a model output at any parameter set, without the need to run the model at the chosen set. We think of the expectation as the prediction provided by the emulator at the chosen parameter set, and we think of the variance as the uncertainty associated to that prediction.

The general structure of a univariate emulator is as follows:

$$f(x) = g(x)^T \xi + u(x),$$

where $g(x)^T \xi$ is a regression term and $u(x)$ is a Gaussian process with mean zero. The role of the regression term is to mimic the global behaviour of the model output, while $u(x)$ represents localised deviations of the output from this global behaviour near to x.

The regression term is specified by:

- a vector of functions of the parameters $g(x)$ which determine the shape and complexity of the regression hypersurface we fit to the training data. For example, if x is one dimensional, i.e. we have just one parameter, setting $g(x) = (1, x)$ corresponds to fitting a straight line to the training data. Similarly, setting $g(x) = (1, x, x^2)$ corresponds to fitting a parabola to the training data. Fig 3.1 shows a one-dimensional example with a quadratic global behaviour: the model output to emulate is in black, while the best fitting parabola is in red.

- a vector of regression coefficients ξ. In the one dimensional case for example, if we set $g(x) = (1, x)$, then $\xi = (\xi_0, \xi_1)$, where ξ_0 is the y-intercept and ξ_1 is the gradient of the straight line fitted to the training data.

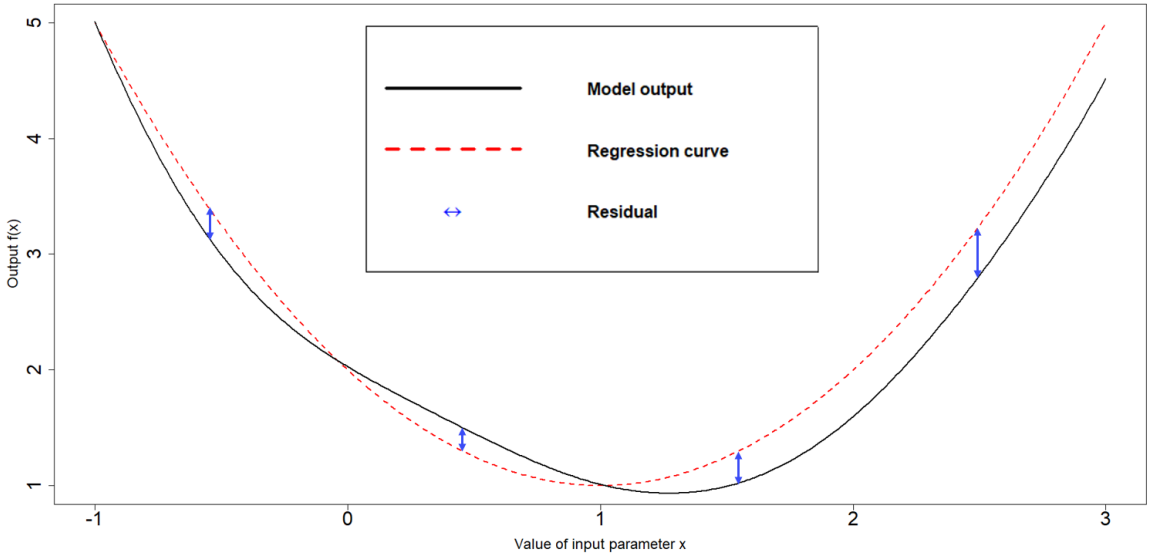


Figure 3.1: Regression term and residuals in one dimensional example

In general, and especially when dealing with complex models, we cannot expect the regression hypersurface to perfectly explain the behaviour of the output. For this reason it is necessary to account for the local deviations of the output from the regression hypersurface. These local deviations, also referred to as residuals, are shown in blue in Fig 3.1. When the parameter space is one-dimensional, they indicate how far the regression term is from the model output at each point. Since residuals are unknown, we treat them as random variables: for each parameter x, we then have a random variable $u(x)$ representing the residual at x. In the HoMER package we assume this collection of random variables $u(x)$ to be a Gaussian process, with mean zero. Informally this means the following:

- for each parameter set x, we consider the residual $u(x)$ as a normally distributed random variable with mean zero. Note that the mean is assumed to be zero, since, even if we expect to see local deviations, we do not expect the output to be systematically above (or below) the regression hypersurface;

- given any pair of parameter sets (x, x'), the pair $(u(x), u(x'))$ is a multivariate normal variable, with mean $(0, 0)$.

To fully describe the Gaussian process $u(x)$ we need to define the covariance structure, i.e. we need to say how correlated the residuals at x and x' are, for any pair (x, x'). A common covariance structure used in Gaussian processes is given by

$$\mathrm{Cov}(u(x), u(x')) = \sigma^2 c(x, x')$$

where c is the square-exponential correlation function

$$c(x, x') := \exp\left(\frac{-\sum_i (x_i - x'_i)^2}{\theta^2}\right)$$

where x_i is the ith-component of the parameter set x. This covariance structure is the default option in the HoMER package, even though other structures are available.

Let us look at the various terms in this covariance structure:

- σ^2 is the **emulator variance**, i.e the variance of $u(x)$, for all parameter sets x. The value of σ reflects how far from the regression hypersurface the model output can be. The larger the value of σ, the farthest the model output can be from the regression hypersurface. In particular, larger values of σ correspond to more uncertain emulators. For example, Fig. 3.1 was generated with a σ of 0.3. A higher σ of 1, would create wider residuals, as in the plot below:

- θ is the **correlation length** of the Gaussian process. For a given pair (x, x'), the larger θ is, the larger is the covariance between $u(x)$ and $u(x')$. This means that the size of θ determines how close two parameter sets must be in order for the corresponding residuals to be non-negligibly correlated. Informally, we can think of θ in the following way: if the distance of two parameters sets is no more than θ, then their residuals will be well correlated. In particular, a larger θ results in a smoother (less wiggly) emulator. In the one dimensional example in Fig. 3.1, a θ of 1 was used. A value of θ equal to 0.4 would result in less smooth residuals:

Choosing values for σ and θ corresponds to making a judgment about how far we expect the output to be from the regression hypersurface (σ) and about its smoothness (θ). While the HoMER package, and in particular the function `emulator_from_data`, selects values of σ and θ for us based on the provided training data, we will see in this workshop how we can intervene to customise the choice of these hyperparameters and the benefits that this operation brings.

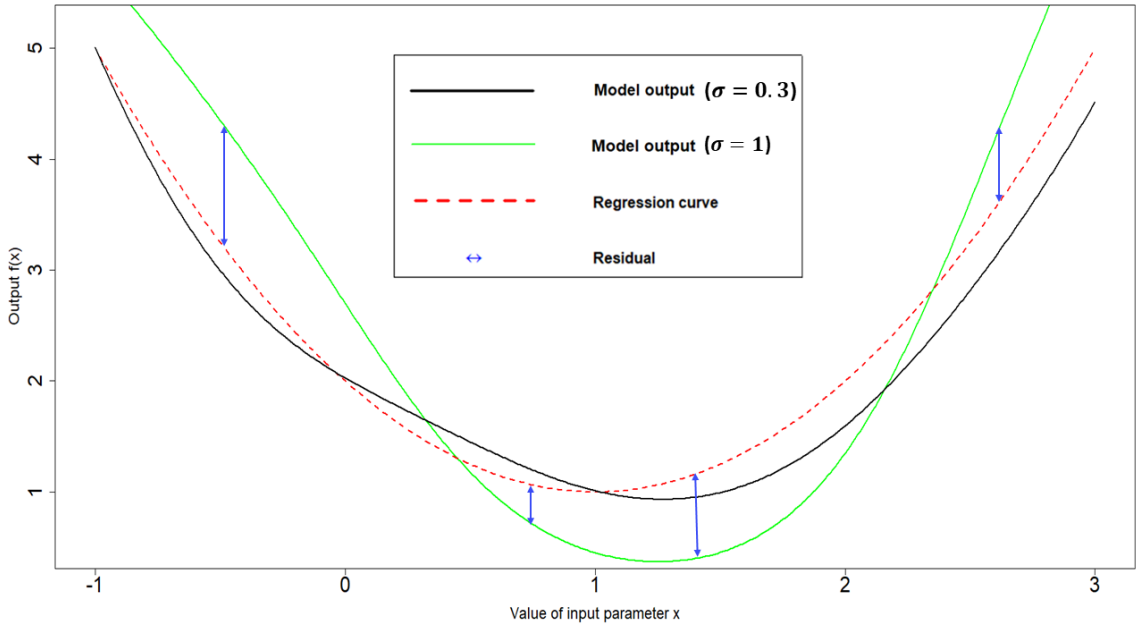


Figure 3.2: Regression term and residuals in one dimensional example, with higher σ

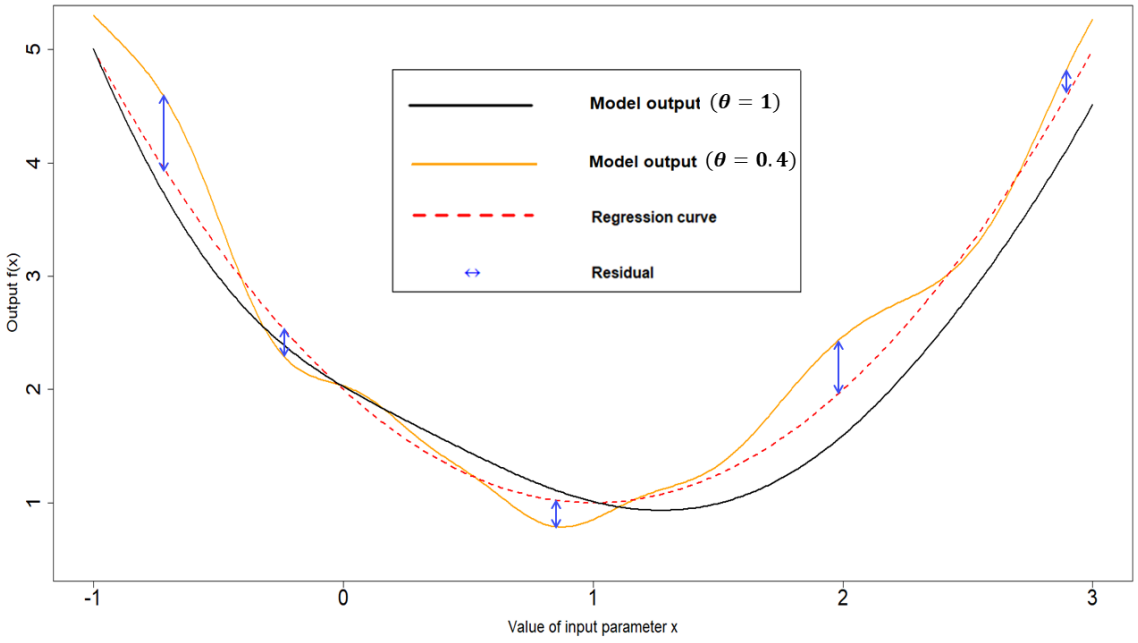


Figure 3.3: Regression term and residuals in one dimensional example, with lower θ

3.2 Training emulators

We are now ready to train the emulators using the `emulator_from_data` function, which just needs the training set, the names of the targets we want to emulate and the ranges of the parameters:

```
ems_wave1 <- emulator_from_data(training, names(targets), ranges)
```

In `ems_wave1` we have information about all emulators. Let us take a look at the emulator of the number of recovered individuals at time $t = 200$:

```
ems_wave1$R200
```

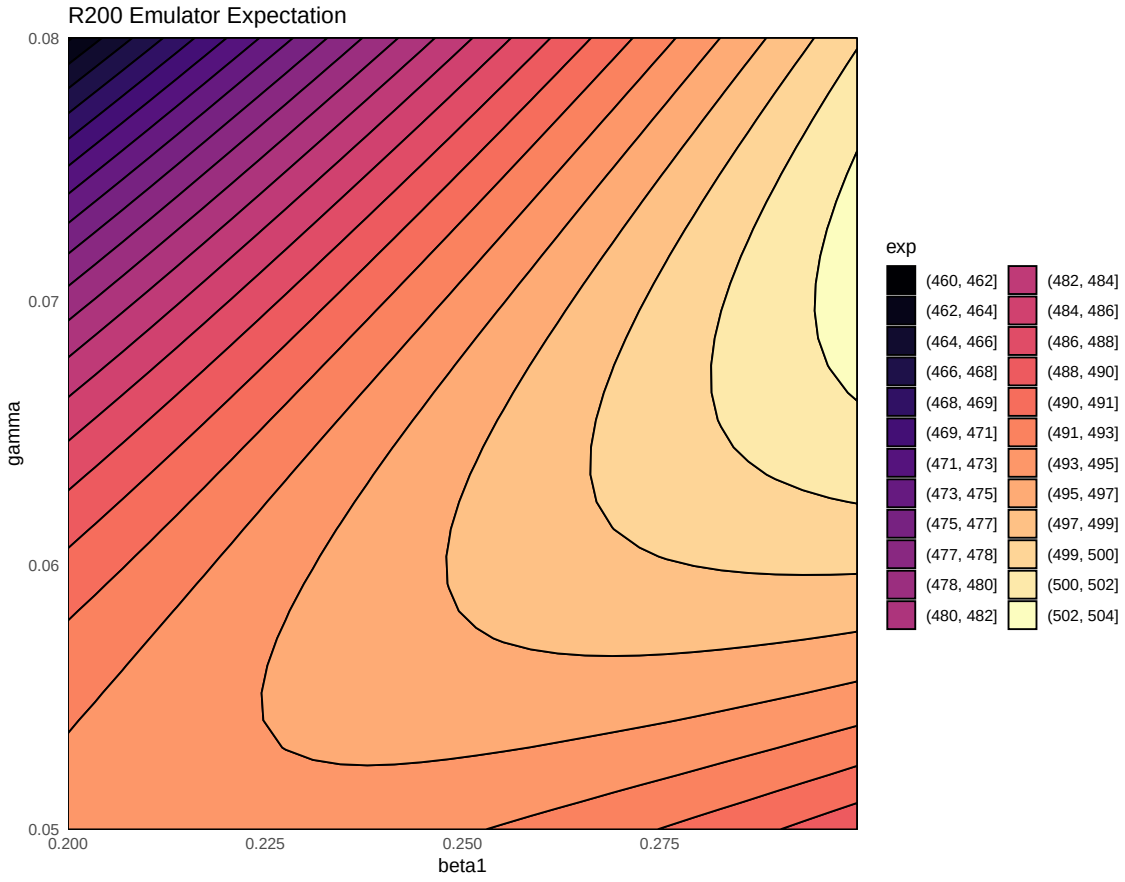
```
## Parameters and ranges:  b: c(0, 1e-04): mu: c(0, 1e-04): beta1: c(0.2, 0.3): beta2: c(0.1, 0.2): bet
## Specifications:
##   Basis functions:  (Intercept); b; mu; beta1; beta2; sigma; alpha; gamma; omega; I(beta1^2); I(beta
##   Active variables b; mu; beta1; beta2; sigma; alpha; gamma; omega
##   Regression Surface Expectation:  496.2728; 1.9876; -4.6397; 8.2317; 25.0217; -15.1378; -57.9129; -
##   Regression surface Variance (eigenvalues):  0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
## Correlation Structure:
## Bayes-adjusted emulator - prior specifications listed.
##   Variance (Representative):  2.040089
##   Expectation:  0
##   Correlation type: exp_sq
##   Hyperparameters:  theta: 0.5009
##   Nugget term: 0
## Mixed covariance:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The print statement provides an overview of the emulator specifications, which refer to the global part, and correlation structure, which refers to the local part:

- Active variables: these are the variables that have the most explanatory power for the chosen output. In our case all variables but β_3 are active.
- Basis Functions: these are the functions composing the vector $g(x)$. Note that, since by default `emulator_from_data` uses quadratic regression for the global part of the emulator, the list of basis functions contains not only the active variables but also products of them.
- First and second order specifications for ξ and $u(x)$. Note that by default `emulator_from_data` assumes that the regression surface is known and its coefficients are fixed. This explains why Regression Surface Variance and Mixed Covariance (which shows the covariance of ξ and $u(x)$) are both zero. The term Variance refers to σ^2 in $u(x)$.

We can also plot the emulators to see how they represent the output space: the `emulator_plot` function does this for emulator expectation (default option), variance, standard deviation, and implausibility. The emulator expectation plots show the structure of the regression surface, which is at most quadratic in its parameters, through a 2D slice of the input space.

```
emulator_plot(ems_wave1$R200, params = c('beta1', 'gamma'))
```



Here for each pair $(\bar{\beta}_1, \bar{\gamma})$ the plot shows the expected value produced by the emulator `ems_wave1$R200` at the parameter set having $\beta_1 = \bar{\beta}_1$, $\gamma = \bar{\gamma}$ and all other parameters equal to their mid-range value (the ranges of parameters are those that were passed to `emulator_from_data` to train `ems_wave1`). Note that we chose to display β_1 and γ, but any other pair can be selected. For consistency, we will use β_1 and γ throughout this workshop.

Task 2

The argument `fixed_vals` allows us to change where to slice the parameters that are not shown in the plots. Try changing this argument: what happens if we set all hidden parameters equal to the values in `chosen_parameters`? What do you expect the emulator expectation to be in this case?

R tip

Show: Solution on P??

Task 3

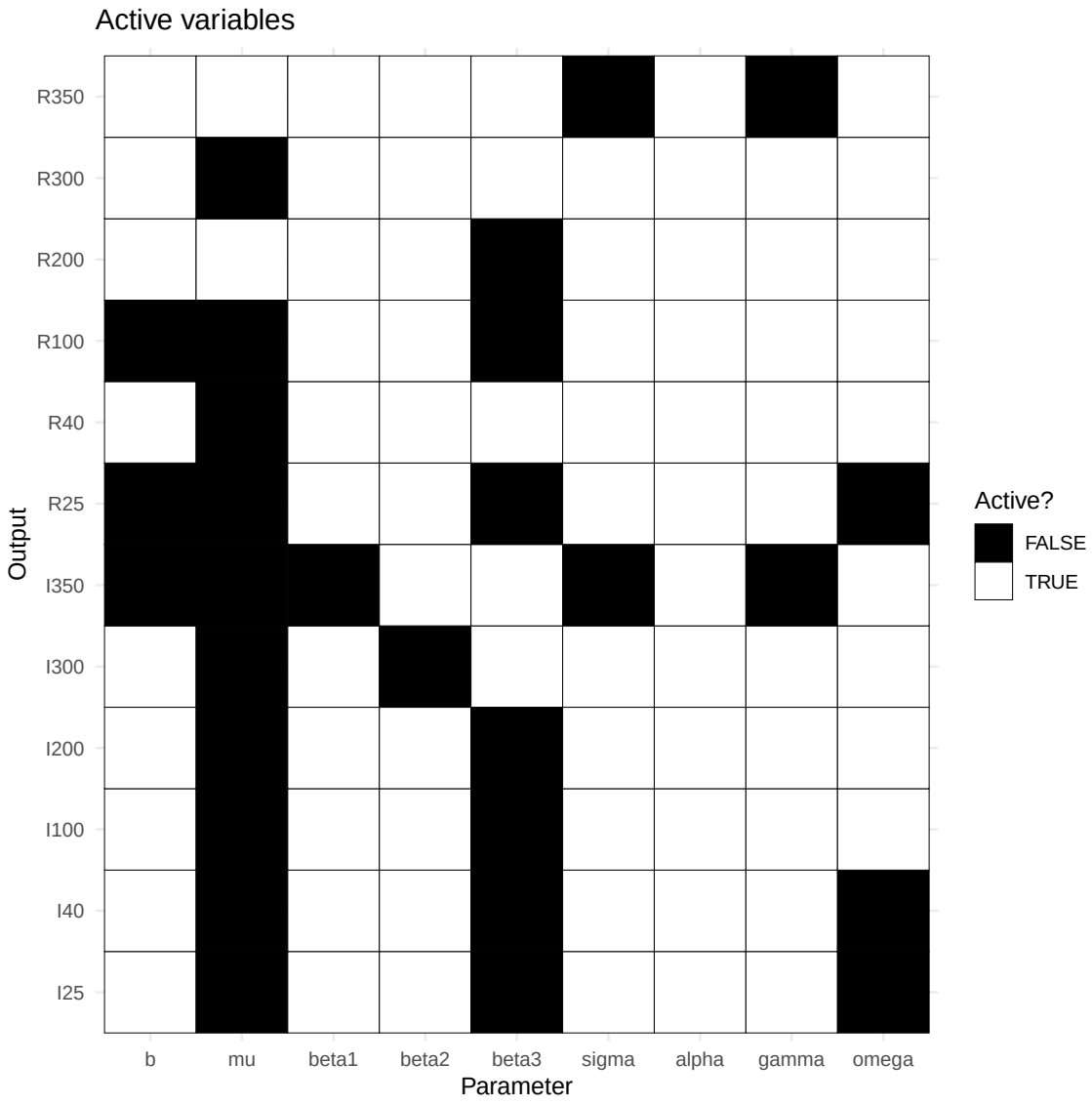
Is β_3 active for all emulators? Why?

R tip

Show: Solution on P??

Looking at what variables are active for different emulators is often an instructive exercise. The code below produces a plot that shows all dependencies at once.

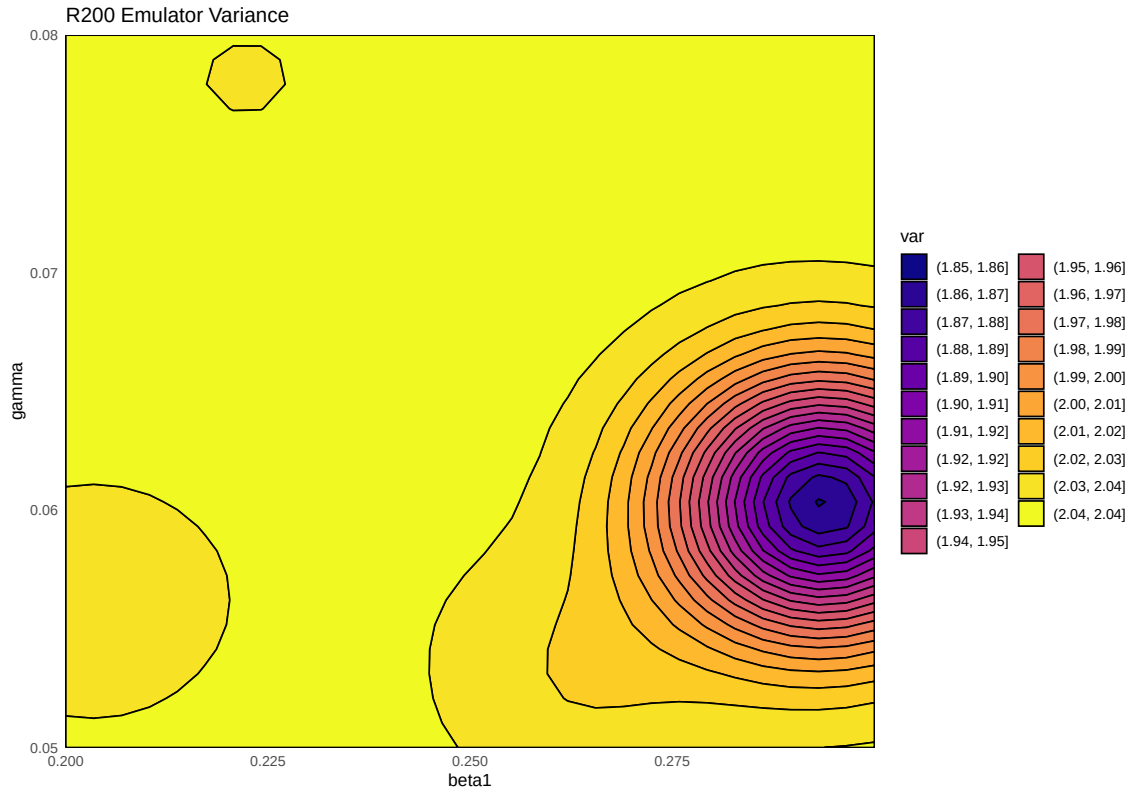
```
plot_actives(ems_wave1)
```



From this table, we can immediately see that μ is inactive for most outputs, while β_1, β_2, σ, α, γ are active for most outputs. We also notice again that β_3 tends to be active for outputs at later times and inactive for outputs at earlier times.

As mentioned above, `emulator_plot` can also plot the variance of a given emulator:

```
emulator_plot(ems_wave1$R200, plot_type = 'var', params = c('beta1', 'gamma'))
```



This plot shows the presence of a training point (purple-blue area on the right) close to the chosen slice of the input space. As discussed above, by default `emulator_plot` fixes all non-shown parameters to their mid-range, but different slices can be explored, through the argument `fixed_vals`. The purple-blue area indicates that the variance is low when we are close to the training point, which is in accordance with our expectation.

Now that we have taken a look at the emulator expectation and the emulator variance, we might want to compare the relative contributions of the global and the residual terms to the overall emulator expectation. This can be done simply by examining the adjusted R^2 of the regression hypersurface:

```
summary(ems_wave1$R200$model)$adj.r.squared
```

```
## [1] 0.9988402
```

We see that we have a very high value of the adjusted R^2. This means that the regression term explains most of the behaviour of the output $R200$. In particular, the residuals contribute little to the emulator predictions. This is not surprising, considering that we are working with a relatively simple SEIRS model. When dealing with more complex models, the regression term may be able to explain the model output less well. In such cases the residuals play a more important role.

Chapter 4

Implausibility

In this section we focus on implausibility and its role in the history matching process. Once emulators are built, we want to use them to systematically explore the input space. For any chosen parameter set, the emulator provides us with an approximation of the corresponding model output. This value is what we need to assess the implausibility of the parameter set in question.

For a given model output and a given target, the implausibility measures the difference between the emulator output and the target, taking into account all sources of uncertainty. For a parameter set x, the general form for the implausibility $I(x)$ is

$$I(x) = \frac{|f(x) - z|}{\sqrt{V_0 + V_c(x) + V_s + V_m}},$$

where $f(x)$ is the emulator output, z the target, and the terms in the denominator refer to various forms of uncertainty. In particular

- V_0 is the variance associated with the observation uncertainty (i.e. uncertainty in estimates from observed data);
- $V_c(x)$ refers to the uncertainty one introduces when using the emulator output instead of the model output itself. Note that this term depends on x, since the emulator is more/less certain about its predictions based on how close/far x is from training parameter sets;
- V_s is the ensemble variability and represents the stochastic nature of the model (this term is not present in this tutorial, since the model is deterministic);
- V_m is the model discrepancy, accounting for possible mismatches between the model and reality.

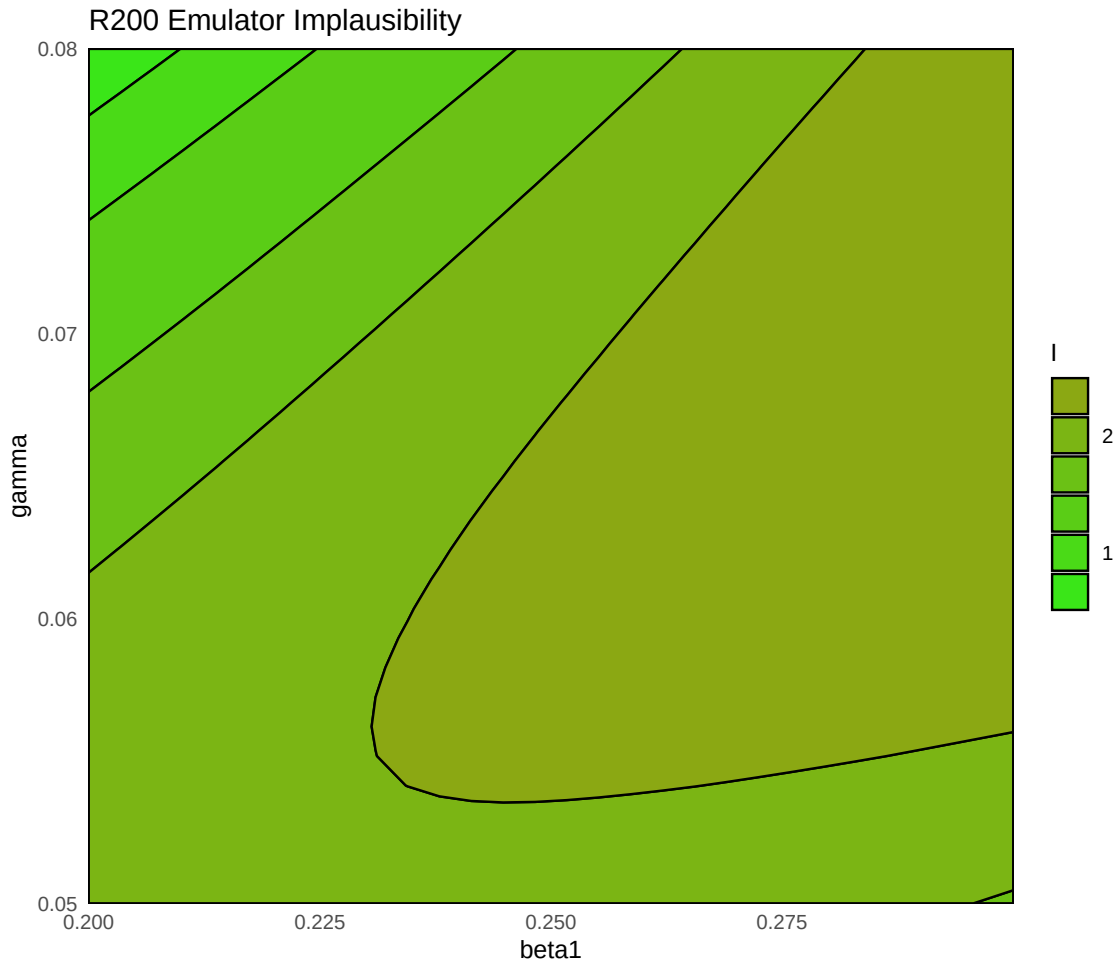
Since in this case study we want to emulate our model, without reference to a real-life analogue, the model represents the reality perfectly. For this reason we have $V_m = 0$. Similarly we have $V_s = 0$, since our model is deterministic. The observation uncertainty V_0 is represented by the 'sigma' values in the `targets` list, while V_c is the emulator variance, which we discussed in the previous section.

A very large value of $I(x)$ means that we can be confident that the parameter set x does not provide a good match to the observed data, even factoring in the additional uncertainty that comes with the use of emulators. When $I(x)$ is small, it could mean that the emulator output is very close to the model output or it could mean that the uncertainty in the denominator of $I(x)$ is large. In the former case, the emulator retains the parameter set, since it is likely to give a good fit to the observation for that output. In the latter case, the emulator does not have enough information to rule the parameter set out and therefore keeps it to explore it further in the next wave.

An important aspect to consider is the choice of cut-off for the implausibility measure. A rule of thumb follows Pukelsheim's 3σ rule, a very general result which states that for any continuous unimodal distribution 95% of the probability lies within 3 sigma of the mean, regardless of asymmetry (or skewness etc). Following this rule, we set the implausibility threshold to be 3: this means that a parameter x is classified as non-implausible only if its implausibility is below 3.

For a given emulator, we can plot the implausibility through the function `emulator_plot` by setting `plot_type='imp'`. Note that we also set `cb=TRUE` to ensure that the produced plots are colour blind friendly:

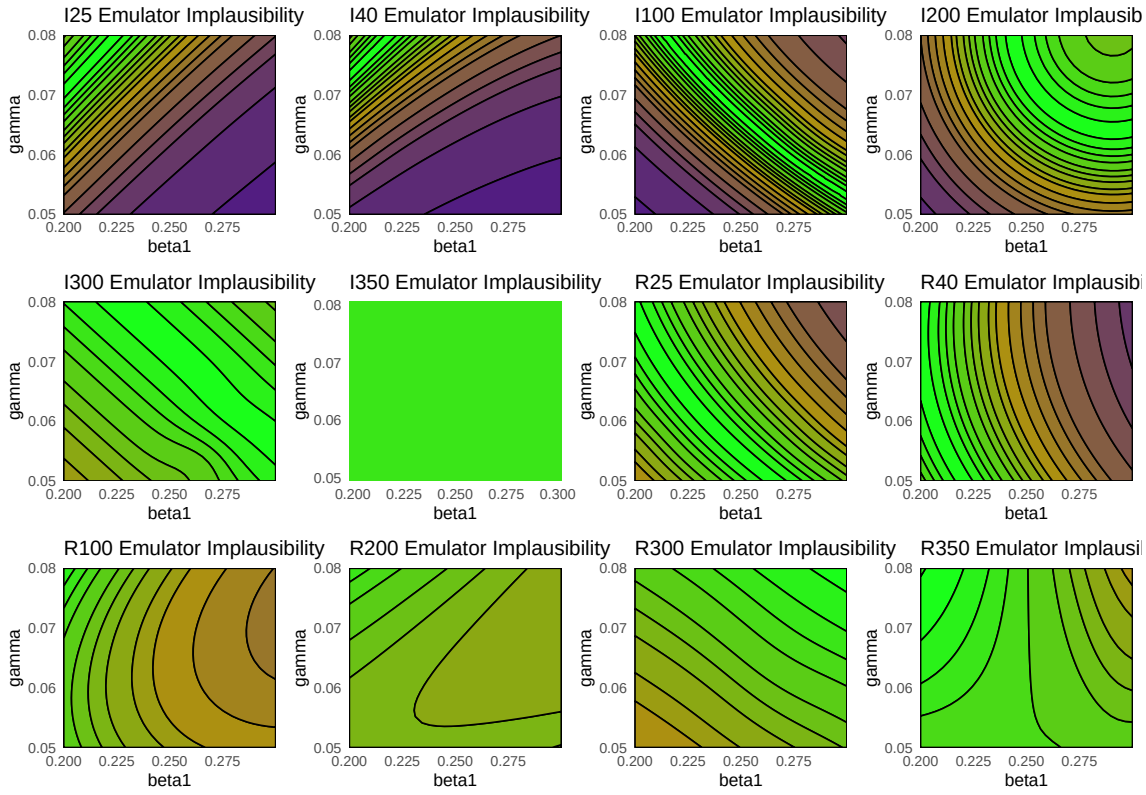
```
emulator_plot(ems_wave1$R200, plot_type = 'imp',
              targets = targets, params = c('beta1', 'gamma'), cb=TRUE)
```



This is a 2D slice through the input space: for a chosen pair $(\bar{\beta}_1, \bar{\gamma})$, the plot shows the implausibility of the parameter set having $\beta_1 = \bar{\beta}_1$, $\gamma = \bar{\gamma}$ and all other parameters set to their mid-range value. Parameter sets with implausibility more than 3 are highly unlikely to give a good fit and will be discarded when forming the parameters sets for the next wave.

Given multiple emulators, we can visualise the implausibility of several emulators at once:

```
emulator_plot(ems_wave1, plot_type = 'imp',
              targets = targets, params = c('beta1', 'gamma'), cb=TRUE)
```



This plot is useful to get an overall idea of which emulators have higher/lower implausibility, but how do we measure overall implausibility? We want a single measure for the implausibility at a given parameter set, but for each emulator we obtain an individual value for I. The simplest way to combine them is to consider maximum implausibility at each parameter set:

$$I_M(x) = \max_{i=1,\ldots,N} I_i(x),$$

where $I_i(x)$ is the implausibility at x coming from the ith emulator. Note that Pukelsheim's rule applies for each emulator separately, but when we combine several emulators' implausibilities together a threshold of 3 might be overly restrictive. For this reason, for large collections of emulators, it may be useful to replace the maximum implausibility with the second- or third-maximum implausibility. This also provides robustness to the failure of one or two of the emulators.

Task 4

Explore the functionalities of `emulator_plot` and produce a variety of implausibility plots. Here are a few suggestions: set `plot_type` to 'imp' to get implausibility plots or to 'nimp' to display the maximum implausibility plot; use the argument `nth` to obtain the second- or third- maximum implausibility plot; select a subset of all targets to pass to `emulator_plot`; change the value of the argument `fixed_vals` to decide where to slice the parameters that are not shown in the plots.

Show: Solution on P??

Chapter 5

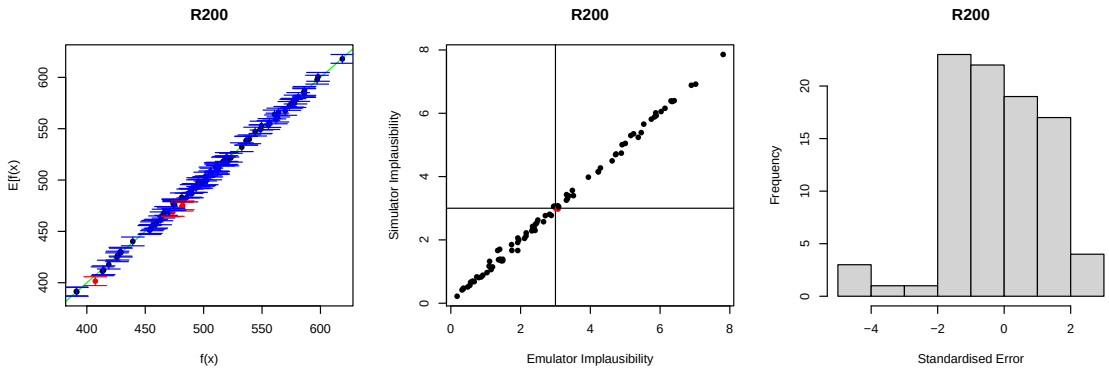
Emulator diagnostics

For a given set of emulators, we want to assess how accurately they reflect the model outputs over the input space. For a given validation set, we can ask the following questions:

- Within uncertainties, does the emulator output accurately represent the equivalent model output?
- Does the emulator adequately classify parameter sets as implausible or non-implausible?
- What are the standardised errors of the emulator outputs in light of the model outputs?

The function `validation_diagnostics` provides us with three diagnostics, addressing the three questions above.

```
vd <- validation_diagnostics(ems_wave1$R200, validation = validation, targets = targets)
```



The first plot shows the emulator outputs plotted against the model outputs. In particular, the emulator expectation is plotted against the model output for each validation point, providing the dots in the graph. The emulator uncertainty at each validation point is shown in the form of a vertical interval that goes from 3σ below to 3σ above the emulator expectation, where σ is the emulator variance at the considered point. The uncertainty interval can be expressed by the formula: $E[f(x)] \pm 3\sqrt{Var(f(x)}$. An 'ideal' emulator would exactly reproduce the model results: this behaviour is represented by the green line $f(x) = E[f(x)]$ (this is a diagonal line, visible here only in the bottom left and top right corners). Any parameter set whose emulated prediction lies more than 3σ away from the model output is highlighted in red. Note that we do not need to have no red points for the test to be passed: since we are plotting 3σ bounds, statistically speaking it is ok to have up to 5% of validation points in red (see Pukelsheim's 3σ rule). Apart from the number of points failing the diagnostic, it is also worth looking at whether the points that fail the diagnostic do so systematically. For example: are they all overestimates/underestimates of the model output?

The second column compares the emulator implausibility to the equivalent model implausibility (i.e. the implausibility calculated replacing the emulator output with the model output). There are three cases to consider:

- The emulator and model both classify a set as implausible or non-implausible (bottom-left and top-right quadrants). This is fine. Both are giving the same classification for the parameter set.
- The emulator classifies a set as non-implausible, while the model rules it out (top-left quadrant): this is also fine. The emulator should not be expected to shrink the parameter space as much as the model does, at least not on a single wave. Parameter sets classified in this way will survive this wave, but may be removed on subsequent waves as the emulators grow more accurate on a reduced parameter space.
- The emulator rules out a set, but the model does not (bottom-right quadrant): these are the problem sets, suggesting that the emulator is ruling out parts of the parameter space that it should not be ruling out.

As for the first test, we should be alarmed only if we spot a systematic problem, with 5% or more of the points in the bottom-right quadrant. Note, however, that it is always up to the user to decide how serious a misclassification is. For instance, a possible check is to identify points that are incorrectly ruled out by one emulator, and see if they would be considered non-implausible by all other emulators. If they are, then we should think about changing the misclassifying emulator.

Finally, the third column gives the standardised errors of the emulator outputs in light of the model output: for each validation point, the difference between the emulator output and the model output is calculated, and then divided by the standard deviation σ of the emulator at the point. The general rule is that we want our standardised errors to be somewhat normally distributed around 0, with 95% of the probability mass between -2 and 2. When looking at the standard errors plot, we should ask ourselves at least the following questions:

- Is more than 5% of the probability mass outside the interval $[-2, 2]$? If the answer is yes, this means that, even factoring in all the uncertainties in the emulator and in the observed data, the emulator output is too often far from the model output.

- Is 95% of the probability mass concentrated in a considerably smaller interval than $[-2, 2]$ (say, for example, $[-0.5, 0.5]$)? For this to happen, the emulator uncertainty must be quite large. In such case the emulator, being extremely cautious, will cut out a small part of the parameter space and we will end up needing many more waves of history matching than are necessary, or, even worse, we just won't be able to reduce the non-implausible parameter space.

- Is the histogram skewing significantly in one direction or the other? If this is the case, the emulator tends to either overestimate or underestimate the model output.

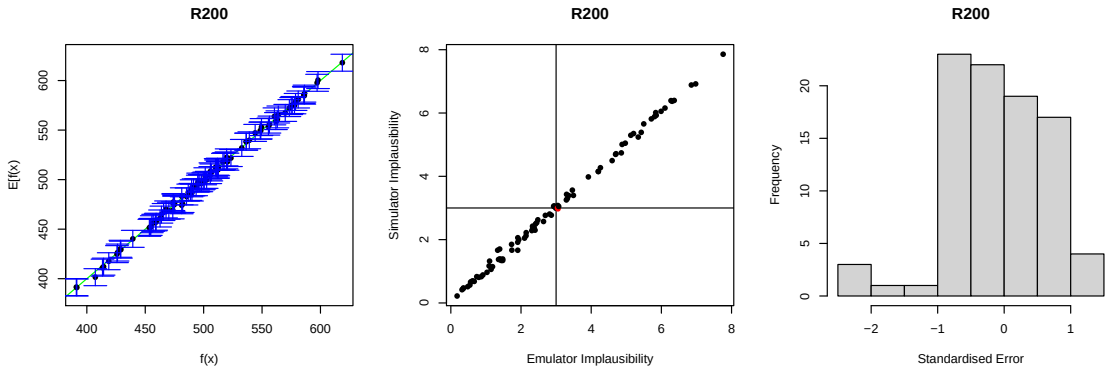
The diagnostic plot on the left shows very few points in red: this is a good result! The plot in the middle has only one red point, which is promising too. Finally, in the third diagnostic if we consider all standardised errors below -2 and above 2, we clearly get more than 5% of all errors: this is not ideal, since we would like the standardised errors to have 95% of the probability mass within ± 2.

A way of improving the performance of an emulator is by changing the variance σ^2 in the Gaussian process u:

$$\sigma^2 \left[(1 - \delta) c(x, x') + \delta I_{\{x = x'\}} \right].$$

The lower the value of σ, the more 'certain' the emulator will be. This means that when an emulator is a little too overconfident (as in our case above), we can try increasing σ. Below we train a new emulator setting σ to be twice as much as its default value, through the method `mult_sigma`:

```
sigmadoubled_emulator <- ems_wave1$R200$mult_sigma(2)
vd <- validation_diagnostics(sigmadoubled_emulator,
                             validation = validation, targets = targets)
```



A higher value of σ has therefore allowed us to build a more conservative emulator that performs better than before.

Task 5

Explore different values of σ. What happens for very small/large values of σ?

R tip

Show: Solution on P??

Chapter 6

Proposing new points

The function `generate_new_runs` is designed to generate new sets of parameters; its default behaviour is as follows.

STEP 1. If prior parameter sets are provided, step 1 is skipped, otherwise a set is generated using a Latin Hypercube Design, rejecting implausible parameter sets. Figure **??** shows an example of LH sampling (not for our model), with implausible parameter sets in red and non-implausible ones in blue:

STEP 2. Pairs of parameter sets are selected at random and more sets are sampled from lines connecting them, with particular importance given to those that are close to the non-implausible boundary. Figure 6.2 shows line sampling on top of the previously shown LH sampling: non-implausible parameter sets provided by LH sampling are in grey, parameter sets proposed by line sampling are in red if implausbile and in blue if non-implausible.

STEP 3. Using these as seeding points, more parameter sets are generated using importance sampling to attempt to fully cover the non-implausible region. Figure 6.3 has non-implausible parameter sets provided by LH and line sampling in grey and non-implausibile parameter sets found by importance sampling in blue.

The combination of the three steps above brings to the following final design of non-implausible parameter sets:

Let us generate 180 new sets of parameters, using the emulators for the time up to $t = 200$. Note that the generation of new points might require a few minutes.

```
new_points_restricted <- generate_new_runs(restricted_ems, 180, targets, verbose=TRUE)
```

```
## [1] "Performing Latin Hypercube sampling..."
## [1] "Proposing at implausibility I=3.5"
## [1] "47 points generated from LHS at I=3.5"
## [1] "Performing line sampling..."
## [1] "Line sampling generated 9 more points."
```

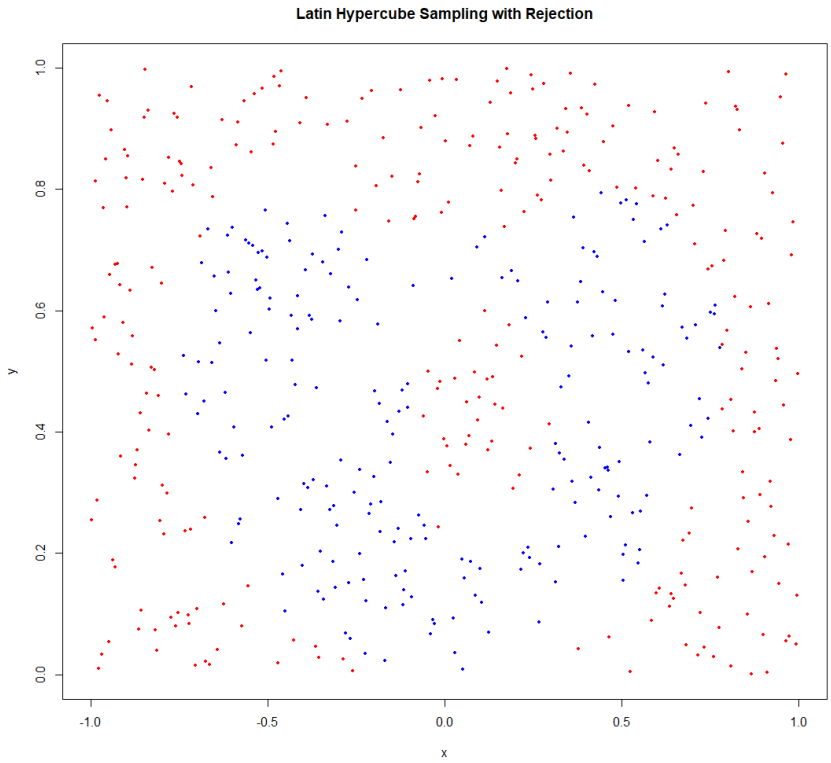


Figure 6.1: LH sampling

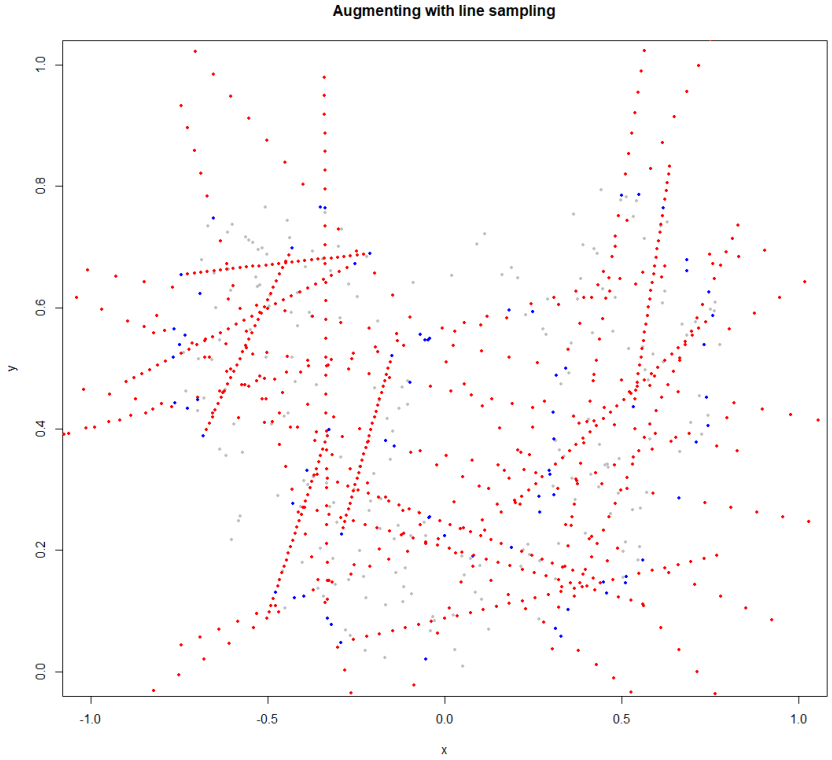


Figure 6.2: Line sampling

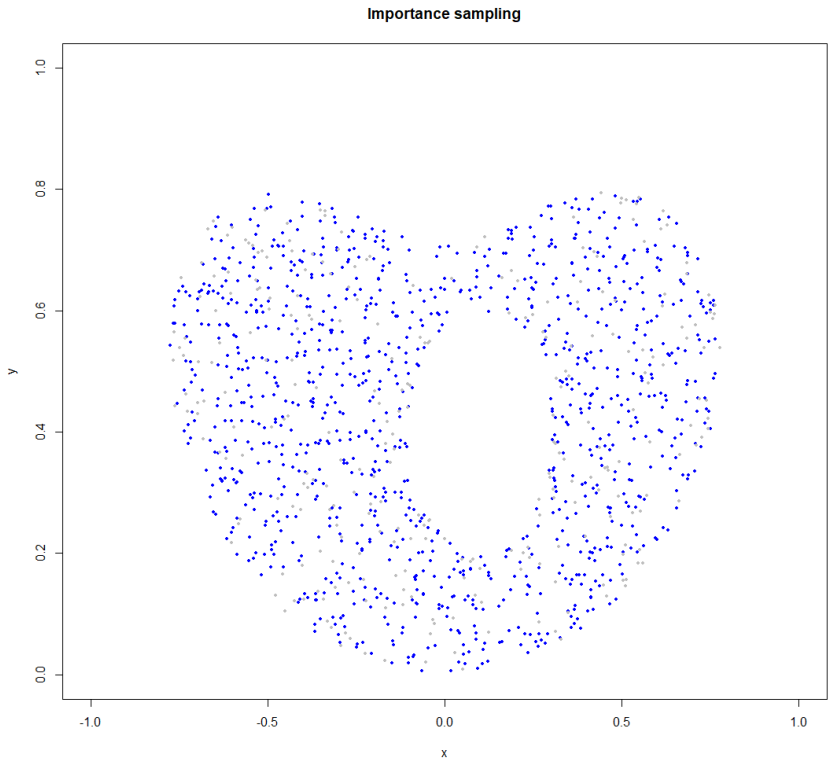


Figure 6.3: Importance sampling

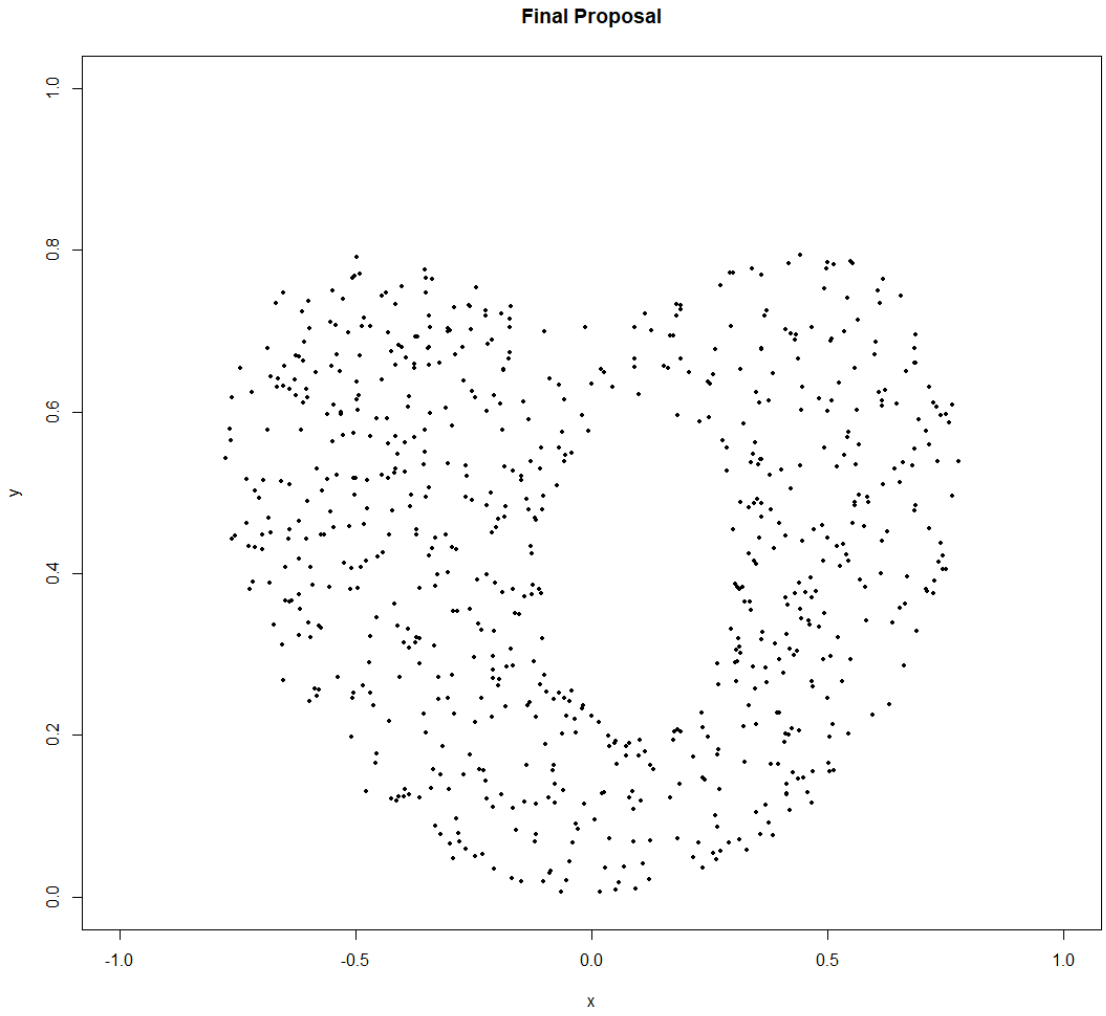
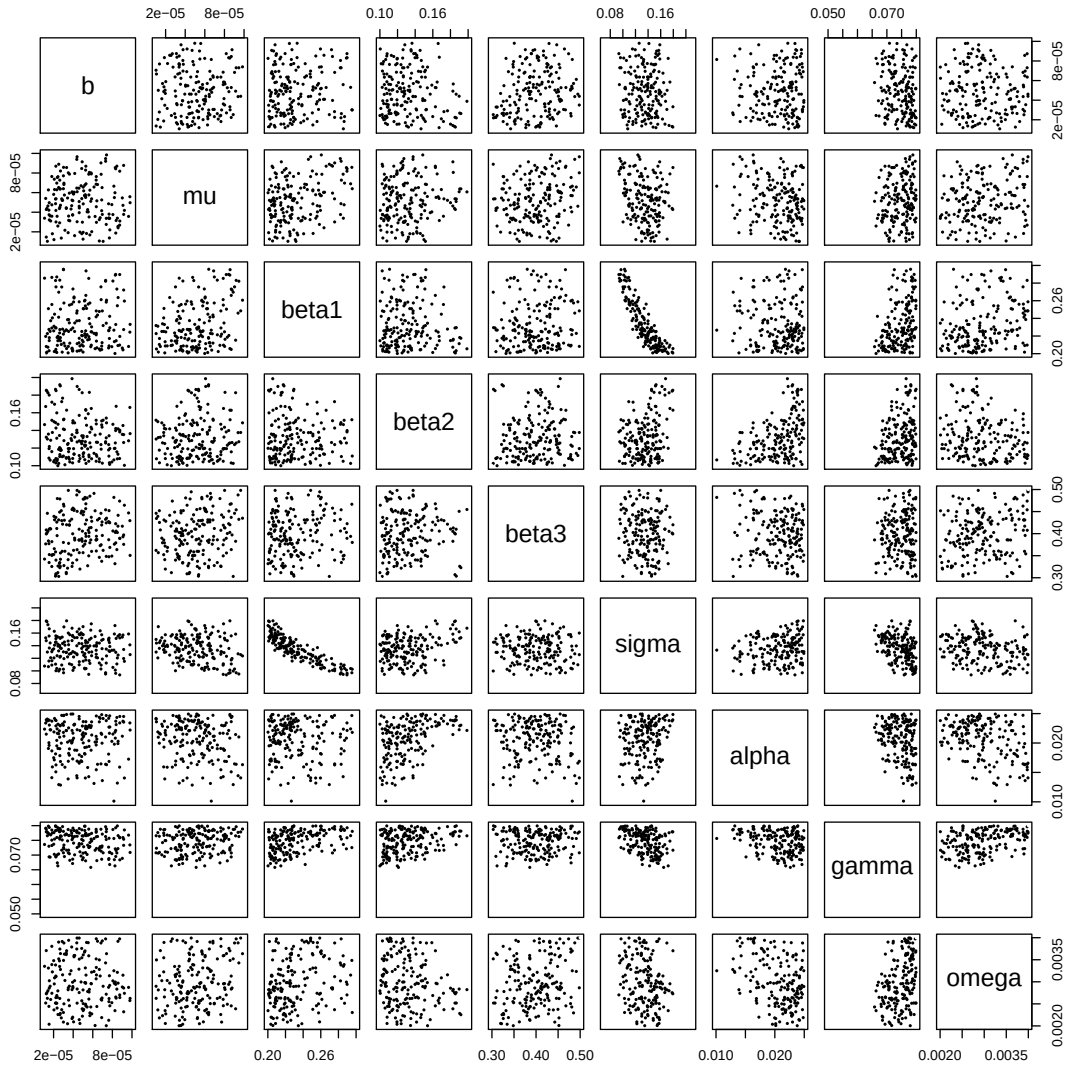


Figure 6.4: Overall design of new non-implausible parameter sets

```
## [1] "Performing importance sampling..."
## [1] "Importance sampling generated 207 more points."
## [1] "Selecting final points using maximin criterion..."
## [1] "128 points generated from LHS at I=3"
## [1] "Performing line sampling..."
## [1] "Line sampling generated 13 more points."
## [1] "Performing importance sampling..."
## [1] "Importance sampling generated 47 more points."
## [1] "Selecting final points using maximin criterion..."
## [1] "Resampling: Performing line sampling..."
## [1] "Line sampling generated 18 more points."
## [1] "Resampling: Performing importance sampling..."
## [1] "Importance sampling generated 78 more points."
## [1] "Selecting final points using maximin criterion..."
```

We now plot `new_points_restricted` through `plot_wrap`. Note that we pass `ranges` too to `plot_wrap` to ensure that the plot shows the entire range for each parameter: this allows us to see how the new set of parameters compares with respect to the original input space.

```
plot_wrap(new_points_restricted, ranges)
```



By looking at the plot we can learn a lot about the non-implausible space. For example, it seems clear that low values of γ cannot produce a match (cf. fourth column). We can also deduce relationships between parameters: β_1 and σ are an example of negatively-correlated parameters. If β_1 is large then σ needs to be small, and vice versa.

Task 6

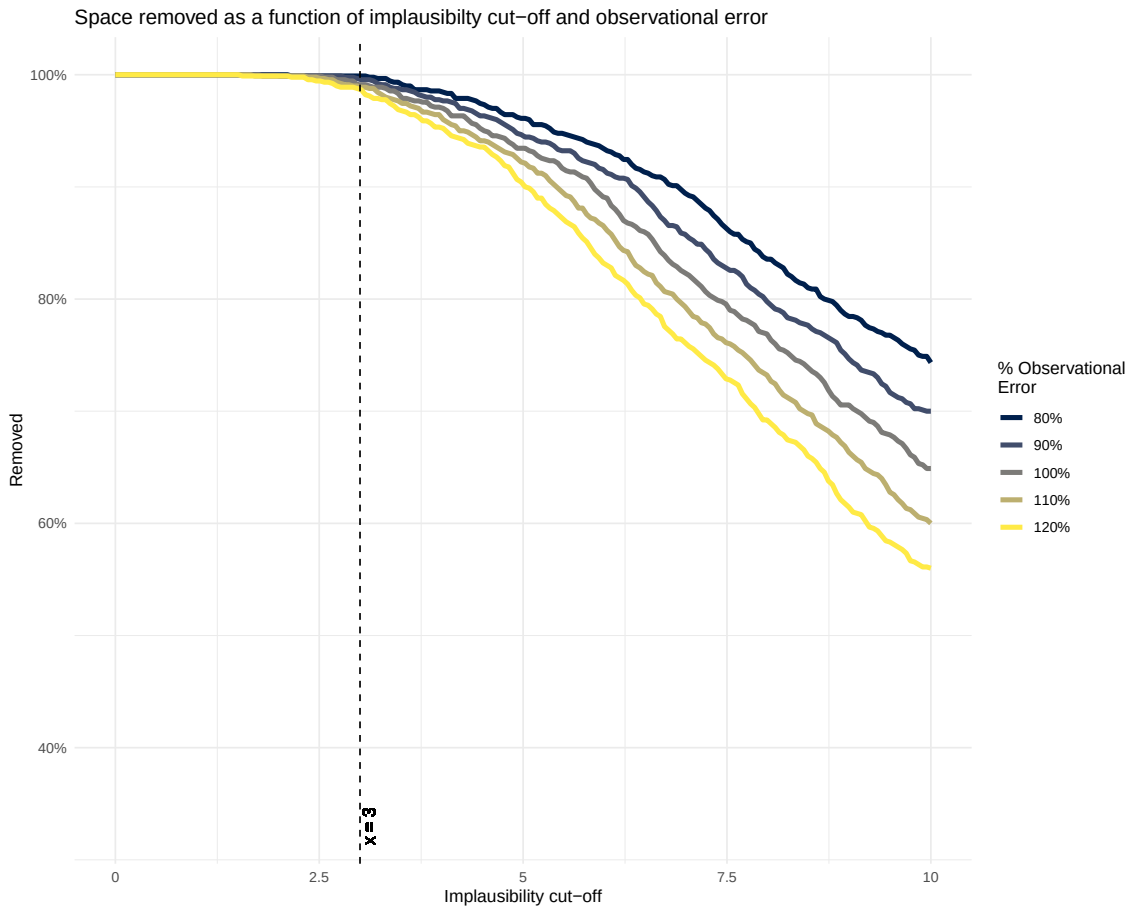
What do you think would happen if we tried generating new parameter sets using all emulators, instead of those relating to times up to $t = 200$ only? Try changing the code and seeing what happens.

Show: Solution on P??

In order to quantify how much of the input space is removed by a given set of emulators, we can use the function `space_removed`, which takes a list of emulators and a list of targets we want to match to. The output is a plot that shows the percentage of space that is removed by the emulators as a function of the implausibility cut-off. Note that here we also set the argument `ppd`, which determines the number of points per input dimension to sample at. In this workshop we have 9 parameters, so a `ppd` of 3 means that `space_removed` will test the emulators on 3^9 sets of parameters.

```
space_removed(ems_wave1, targets, ppd=3) + geom_vline(xintercept = 3, lty = 2) +
  geom_text(aes(x=3, label="x = 3",y=0.33), colour="black",
  angle=90, vjust = 1.2, text=element_text(size=11))
```

```
## Warning: Ignoring unknown parameters: text
```



By default the plot shows the percentage of space that is deemed implausible both when the observational errors are exactly the ones in `targets` and when the observational errors are 80% (resp. 90%, 110% and 120%) of the values in `targets`. Here we see that with an implausibility cut-off of 3, the percentage of space removed is around 99%. If we use a cut-off of 5% instead, then we would discard around 93% (resp. 90%) of the space with the observational errors provided by `targets` (resp. with observational errors equal to 120% of the values in `targets`). As expected, larger observational errors correspond to lower percentages of space removed.

Chapter 7

Customise the first wave

Task 7

Now that we have learnt how to customise the various steps of the process, try to improve the performance of the first wave of emulation and history matching. First, have a look at the emulator diagnostics and see if you can improve the performance of the emulators. Then generate new points using your improved emulators, and compare them to those shown in the task at the end of last section.

Show: Solution on P??

Chapter 8

Second wave

To perform a second wave of history matching and emulation we follow the same procedure as in the previous sections, with two caveats. We start by forming a dataframe `wave1` using parameters sets in `new_points`, as we did with `wave0`, i.e. we evaluate the function `get_results` on `new_points` and then bind the obtained outputs to `new_points`. Half of `wave1` should be used as the training set for the new emulators, and the other half as the validation set to evaluate the new emulators' performance. Note that when dealing with computationally expensive models, using the same number of points for the training and validation sets may not feasible. If p is the number of parameters, a good rule of thumb is to build a training set with at least $10p$ points, and a validation set with at least p points.

Now note that parameter sets in `new_points` tend to lie in a small region inside the original input space, since `new_points` contains only non-implausible points, according to the first wave emulators. The first caveat is then that it is preferable to train the new emulators only on the non-implausible region found in wave one. To do this we define new ranges for the parameters:

```
min_val <- list()
max_val <- list()
new_ranges <- list()
for (i in 1:length(ranges)) {
    par <- names(ranges)[[i]]
    min_val[[par]] <- max(min(new_points[,par])-0.05*diff(range(new_points[,par])),
                          ranges[[par]][1])
    max_val[[par]] <- min(max(new_points[,par])+0.05*diff(range(new_points[,par])),
                          ranges[[par]][2])
    new_ranges[[par]] <- c(min_val[[par]], max_val[[par]])
}
```

The list `new_ranges` contains lower and upper bounds for each parameter. The upper bound for a given parameter is determined in the following way:

- Among all points in `new_points`, the maximum value of the parameter is identified.

- 5% of the original range of the parameter is added to the maximum value found in the previous point. This step enlarges slightly the new range, to make sure that we are including all non-implausible points.
- The minimum between the value found above and the upper bound in the original `ranges` list is selected: this ensure that we do not end up with a new upper bound which is larger than the original one.

A similar calculation was adopted to determine the new lower bounds of parameters.

Since wave two emulators will be trained only on the non-implausible space from wave one, their implausibility cannot be assessed everywhere in the original input space. For this reason, the second caveat is that when generating new parameter sets at the end of wave two, we must consider implausibility across both wave one and wave two emulators, i.e. we need to pass emulators from both waves to the function `generate_new_points`. To do this we simply make the first argument of `generate_new_points` equal to the vector `c(ems_wave2, ems_wave1)`, where `ems_wave2` are the wave two emulators. Note that `generate_new_points` picks the ranges of parameters from the first emulator in the list. For this reason, it is important to put the second wave emulators first (in `c(ems_wave2, ems_wave1)`), which have smaller ranges.

In the task below, you can have a go at wave 2 of the emulation and history matching process yourself.

Task 8

Using `new_points` and `new_ranges`, train new emulators. Customise them and generate new parameter sets.

Show: Solution on P??

As we did for the wave 1 emulators, let us check the values of the adjusted R^2 for the new emulators:

```
R_squared_new <- list()
for (i in 1:length(ems_wave2)) {
  R_squared_new[[i]] <- summary(ems_wave2[[i]]$model)$adj.r.squared
}
names(R_squared_new) <- names(ems_wave2)
unlist(R_squared_new)
```

```
##       I25       I40      I100      I200      I300      I350       R25       R40
## 0.9998347 0.9993282 0.9988811 0.9940101 0.9950220 0.9310200 0.9999403 0.9997465
##      R100      R200      R300      R350
## 0.9992135 0.9996098 0.9844951 0.9942361
```

All R^2 values are very high, meaning that the regression term is contributing far more than the residuals. As all of the emulators we have seen so far have had high R^2 values, we have not discussed the customisation of θ. We now want to briefly comment on what happens when instead the R^2 are

lower and the residuals play a more substantial role. In such cases, the extent to which residuals at different parameter sets are correlated is a key ingredient in the training of emulators, since it determines how informative the model outputs at training parameter sets are. For example, if residuals are highly correlated even at parameter sets that are far away from each other, then knowing the model output at a given parameter set gives us information about a wide region around it. This results in rather confident emulators, which cut a lot of space out. If instead residuals are correlated only for parameter sets that are close to each other, then knowing the model output at a given parameter set gives us information about a small region around it. This creates more uncertain emulators, which can rule out a lot of input parameter space. It is then clear that when we don't have very high R^2 values, we can use θ to increase or decrease the amount of space cut out by emulators.

In practice, the default value of θ is chosen very carefully by the HoMER package, and most users calibrating deterministic models will not have to vary the value of θ for most of their emulators. If, however, you find that the non-implausible space is shrinking very slowly, particularly in later waves (see section 9 for details of how to check this), then the value of θ may be too conservative. If this occurs, then you can increase the θ of the emulators to increase the rate at which space is reduced. You should only do this if you are confident that your outputs are smooth enough to justify the choice of θ however, or you risk the emulators incorrectly excluding space when model fits could be found. We discuss the choice of θ further in workshop on calibrating stochastic models.

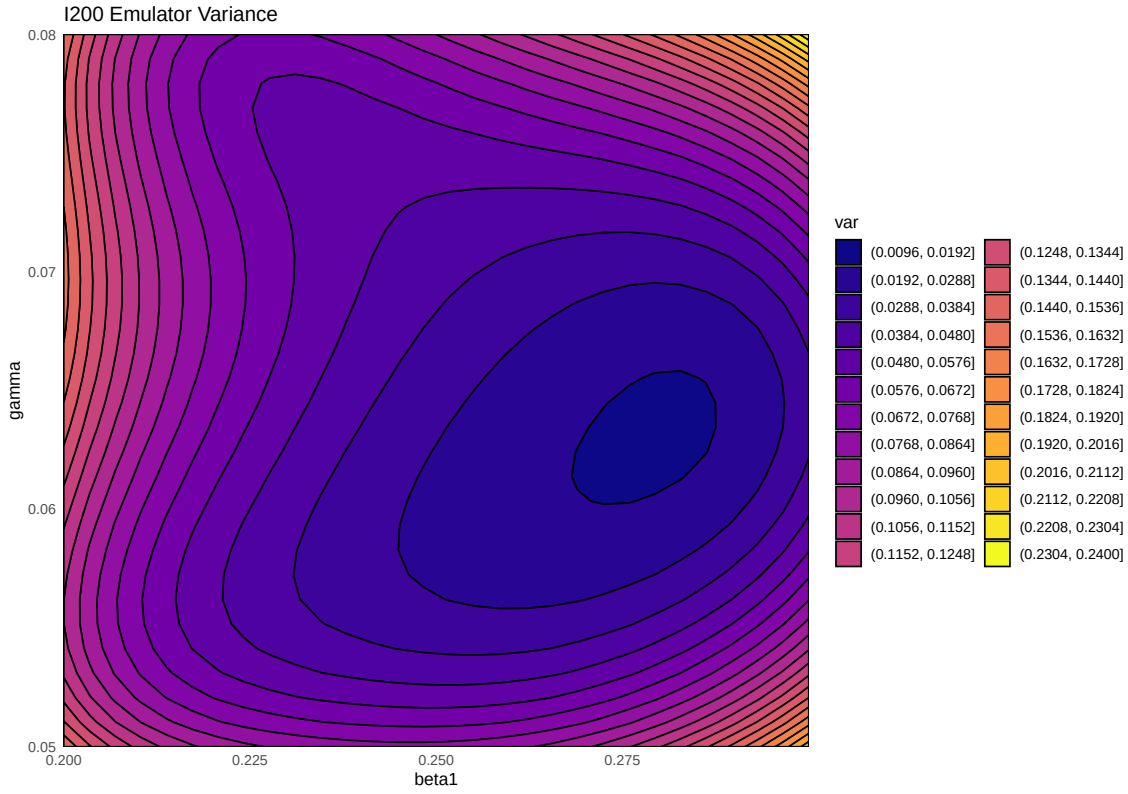
To see all this in practice, we train new wave one emulators, assuming a linear regression term by setting `quadratic=FALSE`:

```
ems_wave1_linear <- emulator_from_data(training, names(targets),
                                       ranges, quadratic=FALSE)
R_squared_linear <- list()
for (i in 1:length(ems_wave1_linear)) {
  R_squared_linear[[i]] <- summary(ems_wave1_linear[[i]]$model)$adj.r.squared
}
names(R_squared_linear) <- names(ems_wave1_linear)
unlist(R_squared_linear)
```

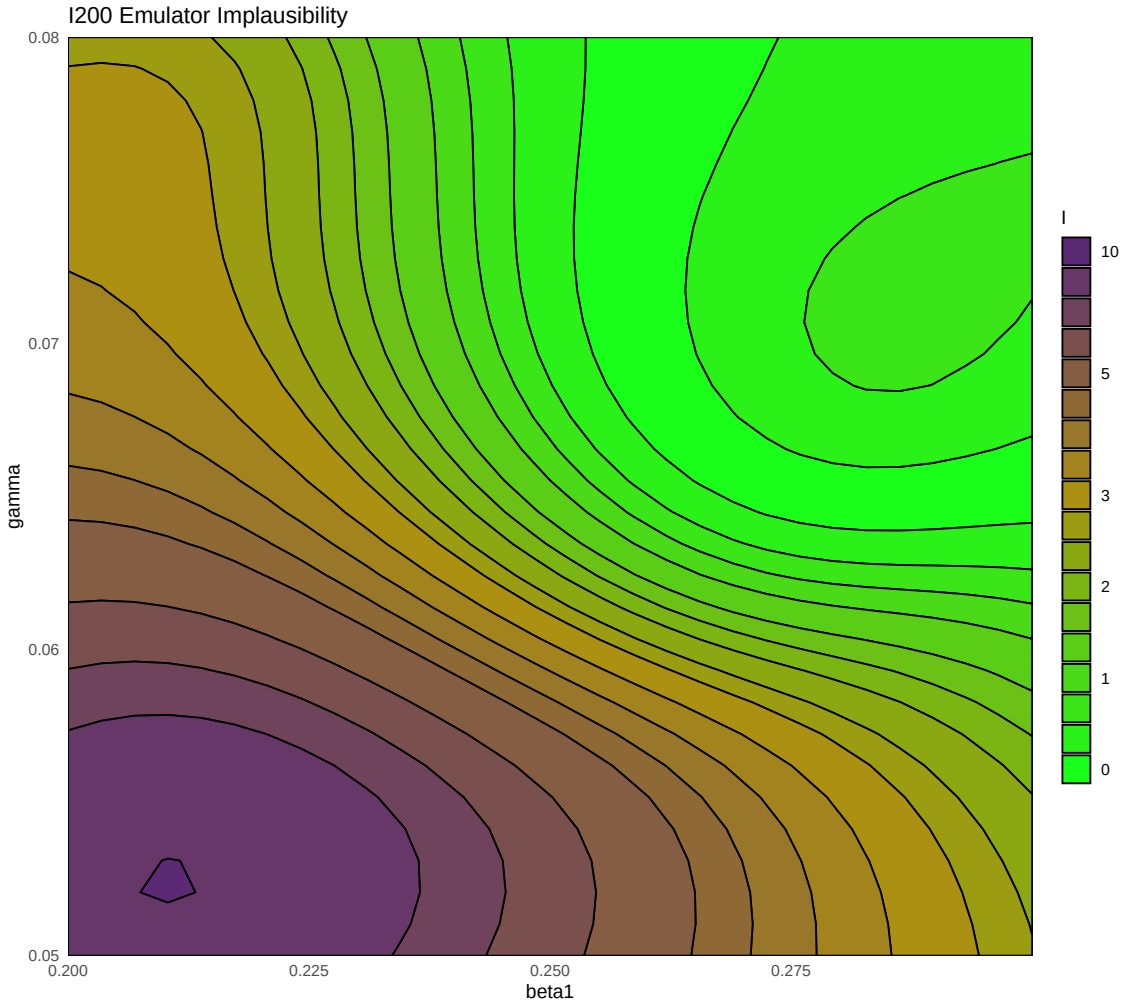
```
##       I25       I40      I100      I200      I300      I350       R25       R40
## 0.9697263 0.9491857 0.9007032 0.7157199 0.8191644 0.1823831 0.9827103 0.9809724
##      R100      R200      R300      R350
## 0.9310675 0.9645519 0.8112647 0.8390023
```

By forcing the regression hypersurface to be linear, we obtain emulators where the global term is not sufficient to explain the model output on its own. As a rough guide, R^2 values of below 0.9 indicate that the residuals are playing an important role. Let us see what happens if we plot the variance and the implausibility for the linear $I200$ emulator before and after increasing its θ by a factor of 3:

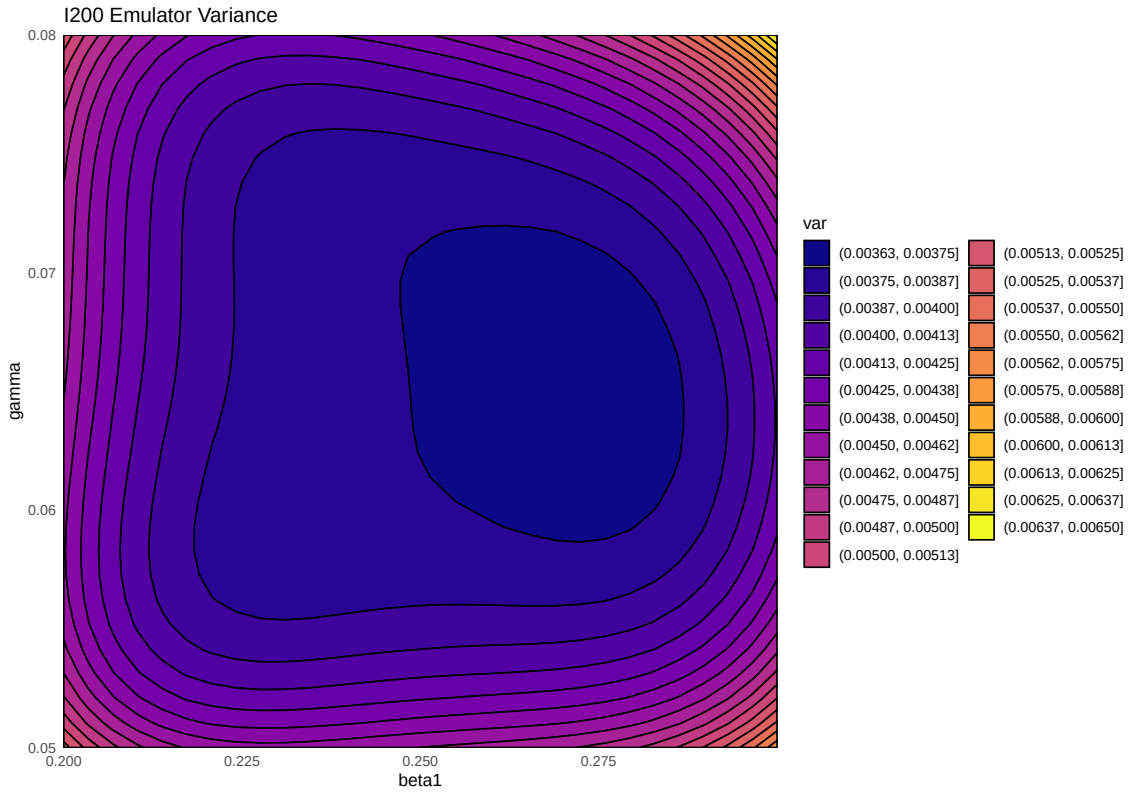
```
emulator_plot(ems_wave1_linear$I200, plot_type = 'var',
              params = c('beta1', 'gamma'))
```



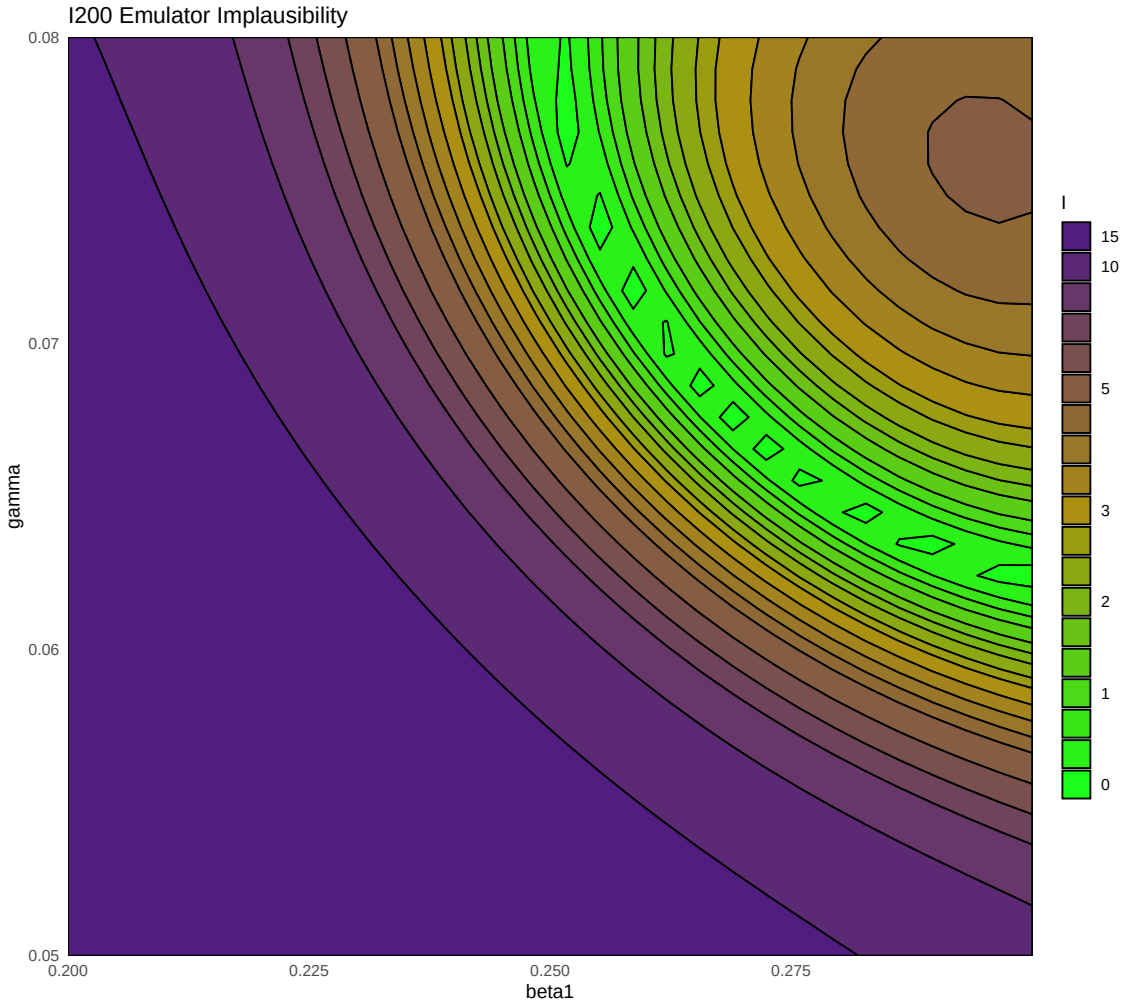

```
emulator_plot(ems_wave1_linear$I200, plot_type = 'imp', targets = targets,
              params = c('beta1', 'gamma'), cb=TRUE)
```




```
ems_wave1_linear$I200 <- ems_wave1_linear$I20$set_hyperparams(
              list(theta=ems_wave1_linear$I200$corr$hyper_p$theta *3))
emulator_plot(ems_wave1_linear$I200, plot_type = 'var',
              params = c('beta1', 'gamma'))
```




```
emulator_plot(ems_wave1_linear$I200, plot_type = 'imp', targets = targets,
              params = c('beta1', 'gamma'), cb=TRUE)
```



First of all, the blue-purple area in the variance plot becomes larger after θ increased: this shows that a higher θ results in the model output at training points influencing a wider region around itself. Second, we see that a higher θ causes the implausibility measure to have higher values: as a consequence, more space will be ruled out as implausible.

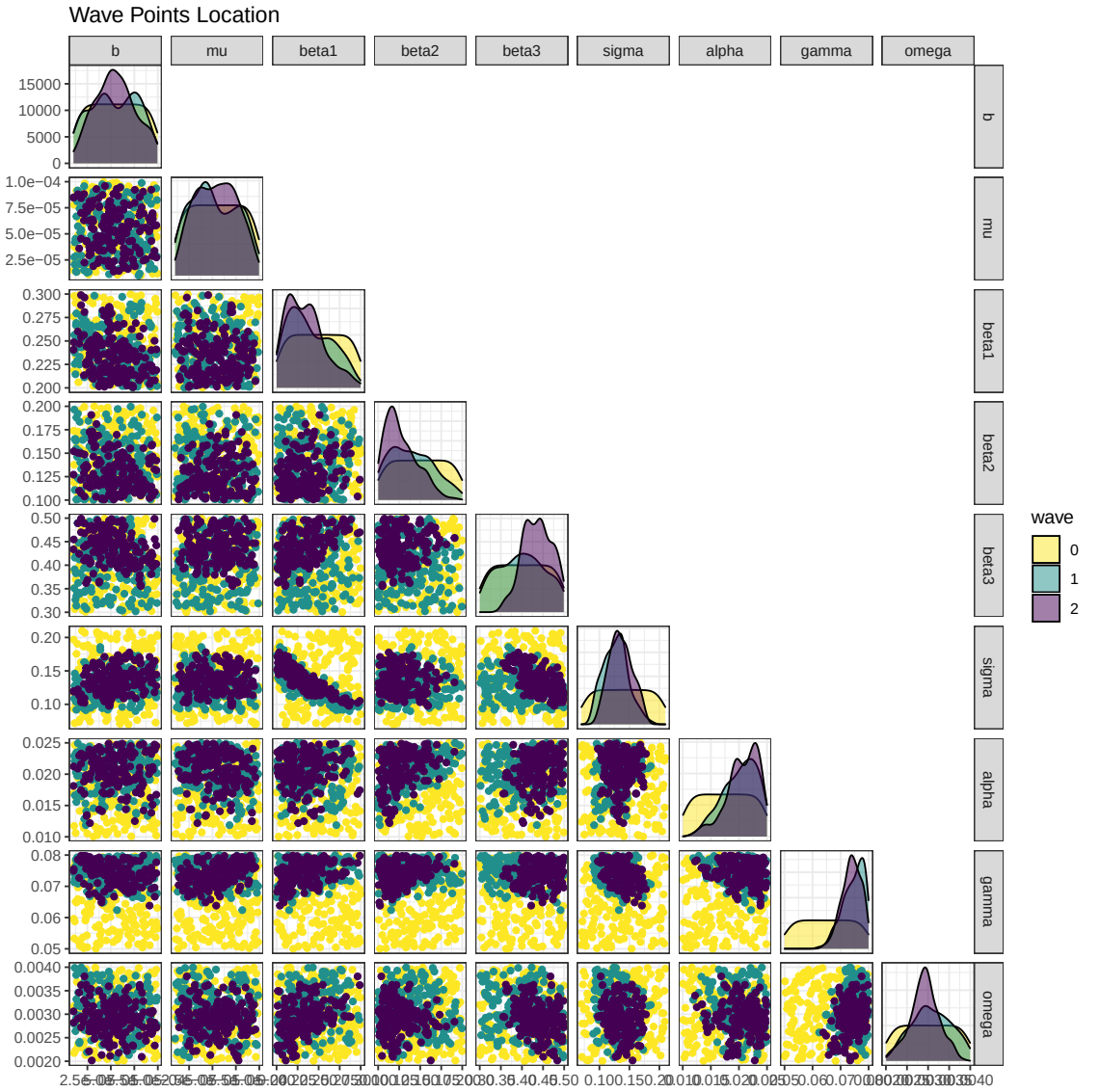
Chapter 9

Multi-wave visualisations

In this last section we present two visualisations that can be used to compare the non-implausible space identified at different waves of the process.

The first visualisation, obtained through the function `wave_points`, shows the distribution of the non-implausible space for the waves of interest. For example, let us plot the distribution of parameter sets at the beginning, at the end of wave one and at the end of wave two:

```
wave_points(list(initial_points, new_points, new_new_points), input_names = names(ranges))
```



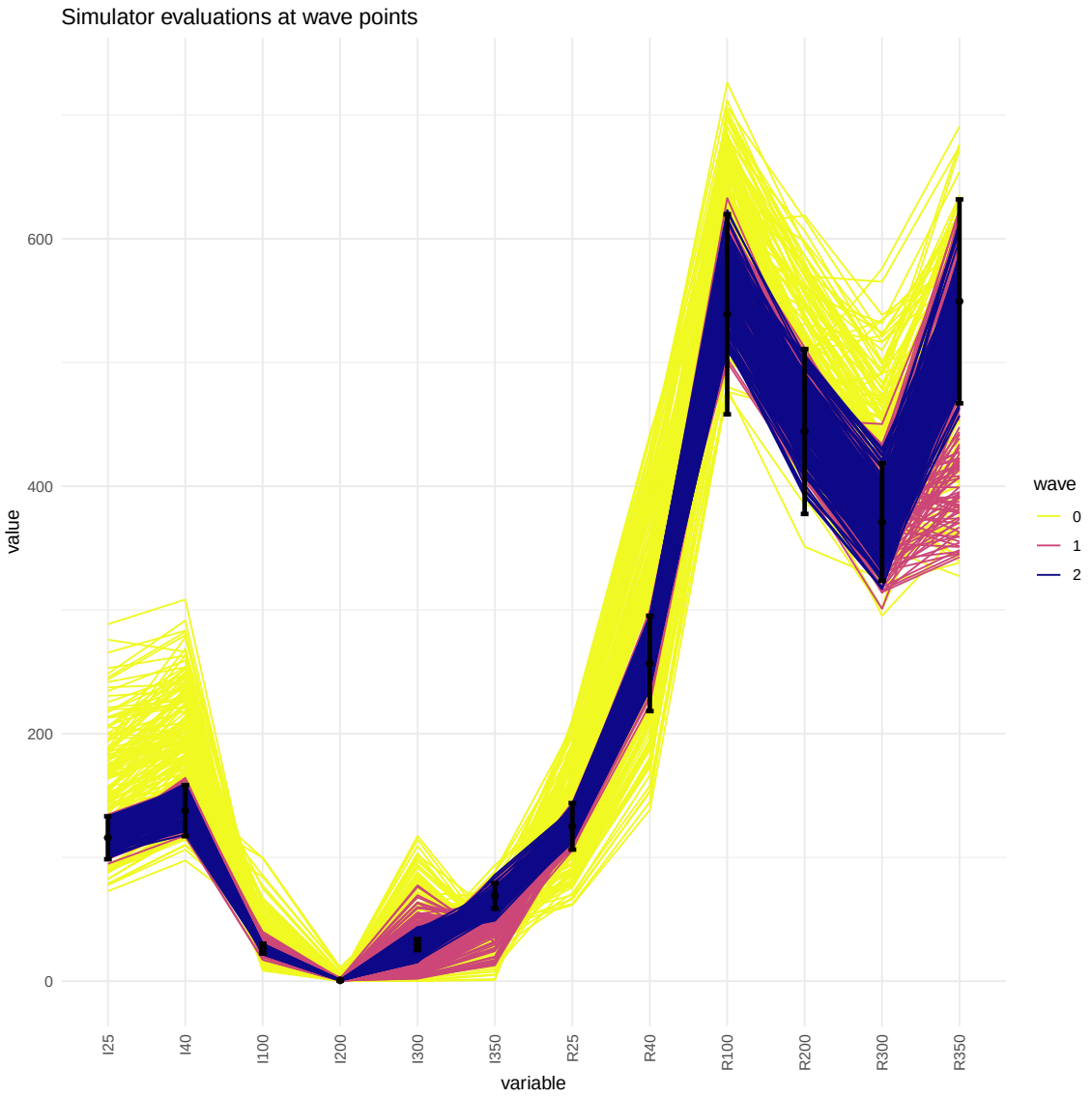
Here `initial_points` are in yellow, `new_points` are in green and `new_new_points` are in purple. The plots in the main diagonal show the distribution of each parameter singularly: we can easily see that the distributions tend to become more and more narrow wave after wave. In the off-diagonal boxes we have plots for all possible pairs of parameters. Again, the non-implausible region identified at the end of each wave clearly becomes smaller and smaller.

The second visualisation allows us to assess how much better parameter sets at later waves perform compared to the original `initial_points`. Let us first create the dataframe `wave2`:

```
new_new_initial_results <- setNames(data.frame(t(apply(new_new_points, 1,
                                    get_results, c(25, 40, 100, 200, 300, 350),
                                    c('I', 'R')))), names(targets))
wave2 <- cbind(new_new_points, new_new_initial_results)
```

We now produce the plots using the function `simulator_plot`:

```
all_points <- list(wave0, wave1, wave2)
simulator_plot(all_points, targets)
```



We can see that, compared to the space-filling random parameter sets used to train the first emulators, the new parameter sets are in much closer agreement with our targets. While the `initial_points` did not match any of the targets, we can see that at the end of wave two, we have several targets matched (I25, I40, R25, R40, R100, R200). Subsequent waves, trained on the new parameter sets, will be more confident in the new non-implausible region: this will allow them to refine the region and increase the number of targets met.

Task 9

In the plot above, some targets are easier to read than others: this is due to the targets having quite different values and ranges. To help with this, `simulator_plot` has the argument `normalize`, which can be set to `TRUE` to rescale the target bounds in the plot. Similarly, the argument `logscale` can be used to plot log-scaled target bounds. Explore these options and get visualisations that are easier to interpret.

Show: Solution on P??

\end{comment}