Workshop 2: calibrating a stochastic model

Danny Scarponi, Andy Iskauskas

Contents

Chapter 1

Objectives

In this workshop, you will learn how to perform history matching with emulation on stochastic models, using the hmer package. Much of the calibration process is the same for stochastic and deterministic models, and we therefore recommend that before starting this workshop you complete Workshop 1, which shows how to perform history matching with emulation on deterministic models. In this workshop, we cover the overall process in less detail, with new information that is specific to stochastic models highlighted using boxes:

> Boxes like these contain material that is specific to the calibration of stochastic models, and is therefore not covered in the deterministic workshop.

There are two reasons why epidemiologists may choose to use a stochastic model structure. The first is when modelling small numbers (e.g. a small population size, or small numbers of cases), where we are interested in the results of individual runs (e.g. to answer the question of what is the probability of an outbreak taking off following the introduction of an infection?). In this first case, we may wish to find input parameter sets where any individual model run is consistent with the calibration targets.

The second is when some characteristic that we wish to include in a model requires the use of an individual-based model, which are necessarily stochastic (e.g. if we wish to accurately capture patterns of repeated contact within and outside households). In this second case, we are not interested in the variation between individual runs at the same input parameter set, only in the mean result when averaged over large numbers of model runs. We therefore want to find input parameter sets where the mean outputs are consistent with the calibration targets.

In this workshop we focus on the second case, and develop an emulator structure that helps us match to the mean of the model. This allows us to rule out large portions of the input parameter space, and it can be very useful even in the case in which one is interested in single runs of the model. An implementation of history matching with emulation dealing with single runs of stochastic models will appear in future versions of our package.

Note that when running the workshop code on your device, you should not expect the hmer visualisation tools to produce the same exact output as the one you can find in the following sections. This is mainly because the maximinLHS function, that you will use to define the initial parameter sets on which emulators are trained, does return different Latin Hypercube designs at each call.

Before starting the tutorial, you will need to run the code contained in the box below: it will load all relevant dependencies and a few helper functions which will be introduced and used later.

Show: Code to load relevant libraries and helper functions on P??

Chapter 2

An overview of history matching with emulation and hmer

A video presentation of this section can be found here.

In this section we briefly recap the history matching with emulation workflow and we explain how various steps of the process can be performed using hmer functions. For a more detailed introduction to history matching with emulation in the case of stochastic models you can check Ian Vernon's presentation.

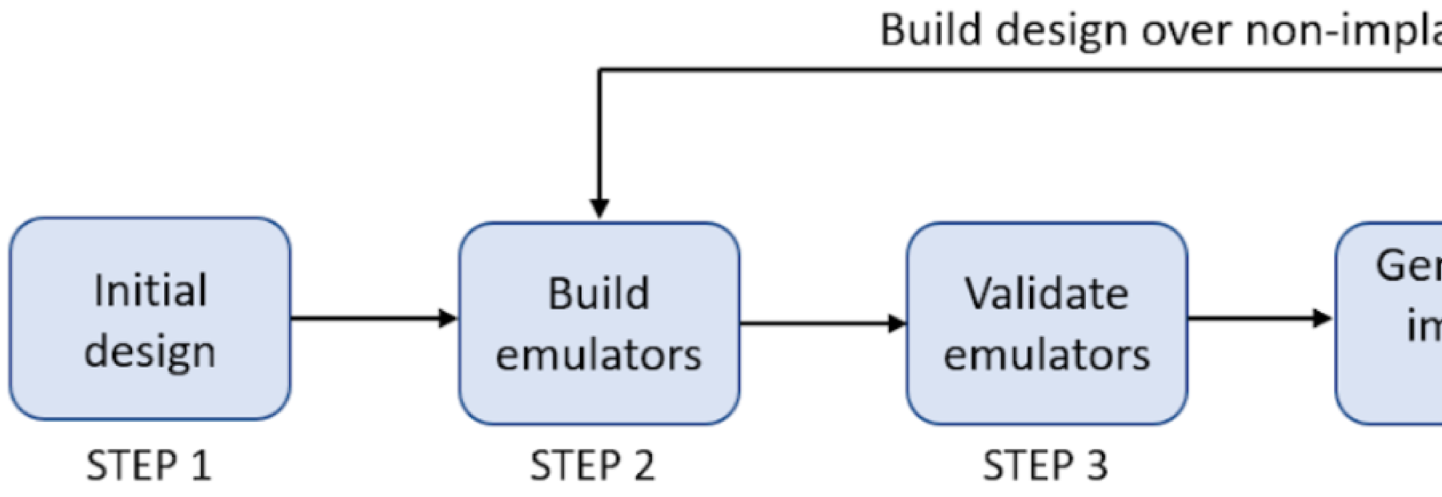
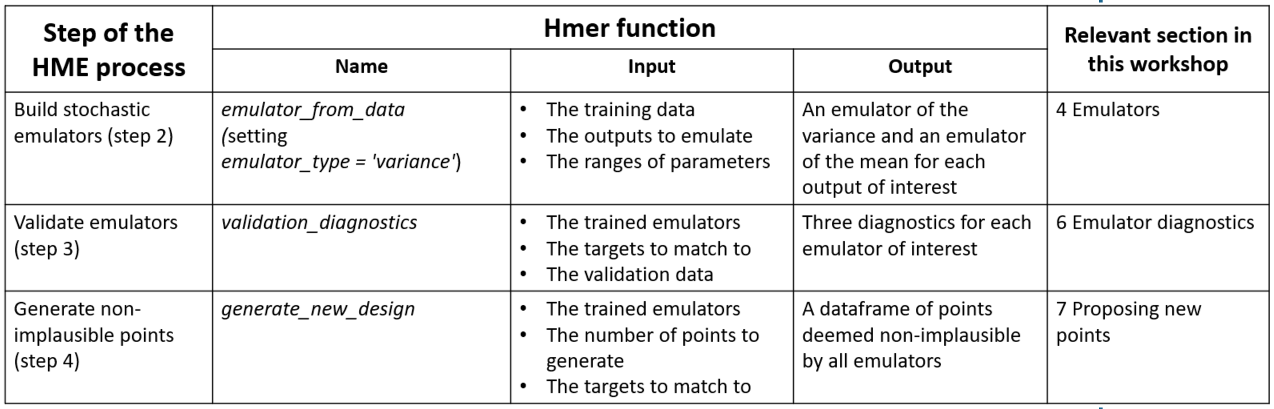


Figure 2.1: History matching with emulation workflow

As shown in the above image, history matching with emulation proceeds in the following way:

1. The model is run on the initial design points, a manageable number of parameter sets that are spread out uniformly across the input space.
2. Emulators are built using the training data (model inputs and outputs) provided by the model runs.
3. Emulators are tested, to check whether they are good approximations of the model outputs.
4. Emulators are evaluated at a large number of parameter sets. The implausibility of each of these is then assessed, and the model run using parameter sets classified as non-implausible.
5. The process is stopped or a new wave of the process is performed, going back to step 2.

The table below associates each step of the process with the corresponding hmer function. Note that step 1 and 5 are not in the table, since they are specific to each model and calibration task. The last column of the table indicates what section of this workshop deals with each step of the process.



Step of the HME process	**Hmer function**			**Relevant section in this workshop**
	Name	**Input**	**Output**	
Build stochastic emulators (step 2)	*emulator_from_data* *(*setting *emulator_type = 'variance'*)	• The training data • The outputs to emulate • The ranges of parameters	An emulator of the variance and an emulator of the mean for each output of interest	4 Emulators
Validate emulators (step 3)	*validation_diagnostics*	• The trained emulators • The targets to match to • The validation data	Three diagnostics for each emulator of interest	6 Emulator diagnostics
Generate non-implausible points (step 4)	*generate_new_design*	• The trained emulators • The number of points to generate • The targets to match to	A dataframe of points deemed non-implausible by all emulators	7 Proposing new points

Chapter 3

Introduction to the model

A video presentation of this section can be found here.

In this section we introduce the model that we will work with throughout our workshop.

To facilitate the comparison between the deterministic and the stochastic setting, we will work with the SEIRS model which we used in Workshop 1, but this time we introduce stochasticity. The deterministic SEIRS model used in Workshop 1 was described by the following differential equations:

$$\frac{dS}{dt} = bN - \frac{\beta(t)IS}{N} + \omega R - \mu S \tag{3.1}$$
$$\frac{dE}{dt} = \frac{\beta(t)IS}{N} - \epsilon E - \mu E \tag{3.2}$$
$$\frac{dI}{dt} = \epsilon E - \gamma I - (\mu + \alpha)I \tag{3.3}$$
$$\frac{dR}{dt} = \gamma I - \omega R - \mu R \tag{3.4}$$

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,
- ϵ 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.

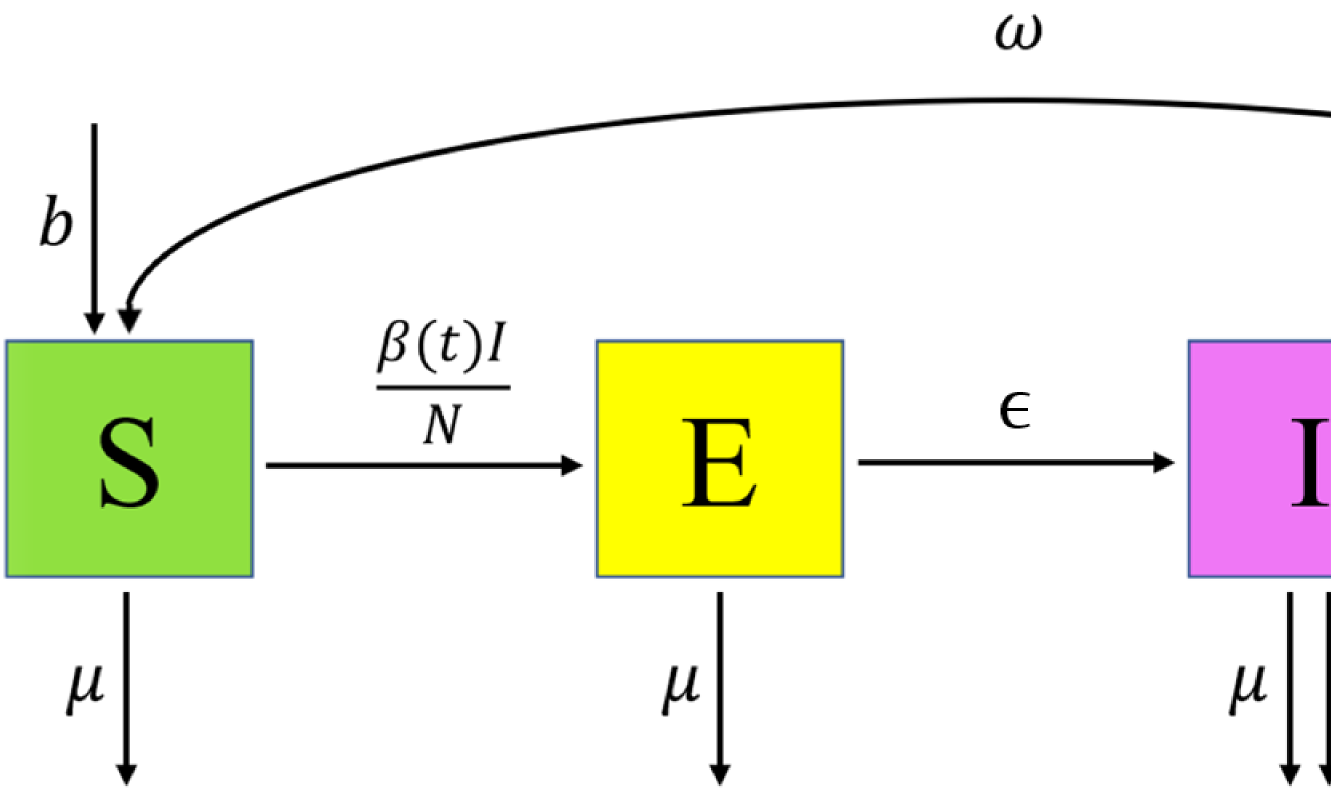


Figure 3.1: SEIRS Diagram

The rate of infection between each infectious and susceptible person $\beta(t)$ 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 $\beta_3 = 0.4$:

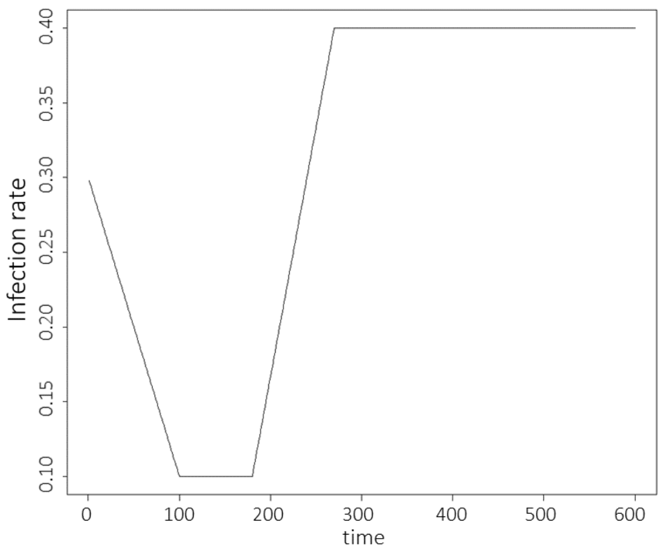


Figure 3.2: Graph of the rate of infection between each infectious and susceptible person

In order to obtain the solutions of the stochastic SEIR model for a given set of parameters, we will use a helper function, `get_results` (which is defined in the R-script). This function assumes an initial population of 900 susceptible individuals, 100 exposed individuals, and no infectious or recovered individuals, and uses the Gillespie algorithm to generate trajectories of the model. The minimum specifications for this function are: the parameter set(s) to run the model on, a set of outputs (e.g. `c("S","R")`), and a set of times (e.g. `c(10,100,150)`) that we are interested in. The default behaviour of `get_results` is to run the model 100 times on the parameter set(s) provided, but more or fewer repetitions can be obtained, using the argument `nreps`. The function `get_results` returns a dataframe containing a row for each repetition at each parameter set provided, with the first nine columns showing the values of the parameters and the subsequent columns showing the requested outputs at the requested times. If `raw` is set to `TRUE`, all outputs and all times are instead returned: this is useful if we want to plot "continuous" trajectories.

As in Workshop 1, the outputs obtained with the parameter set

```r
chosen_params <- c(
  b = 1/(76*365),
  mu = 1/(76*365),
  beta1 = 0.214, beta2 = 0.107, beta3 = 0.428,
  epsilon = 1/7,
  alpha = 1/50,
  gamma = 1/14,
  omega = 1/365
)
```

will be used to define the target bounds for our calibration task.

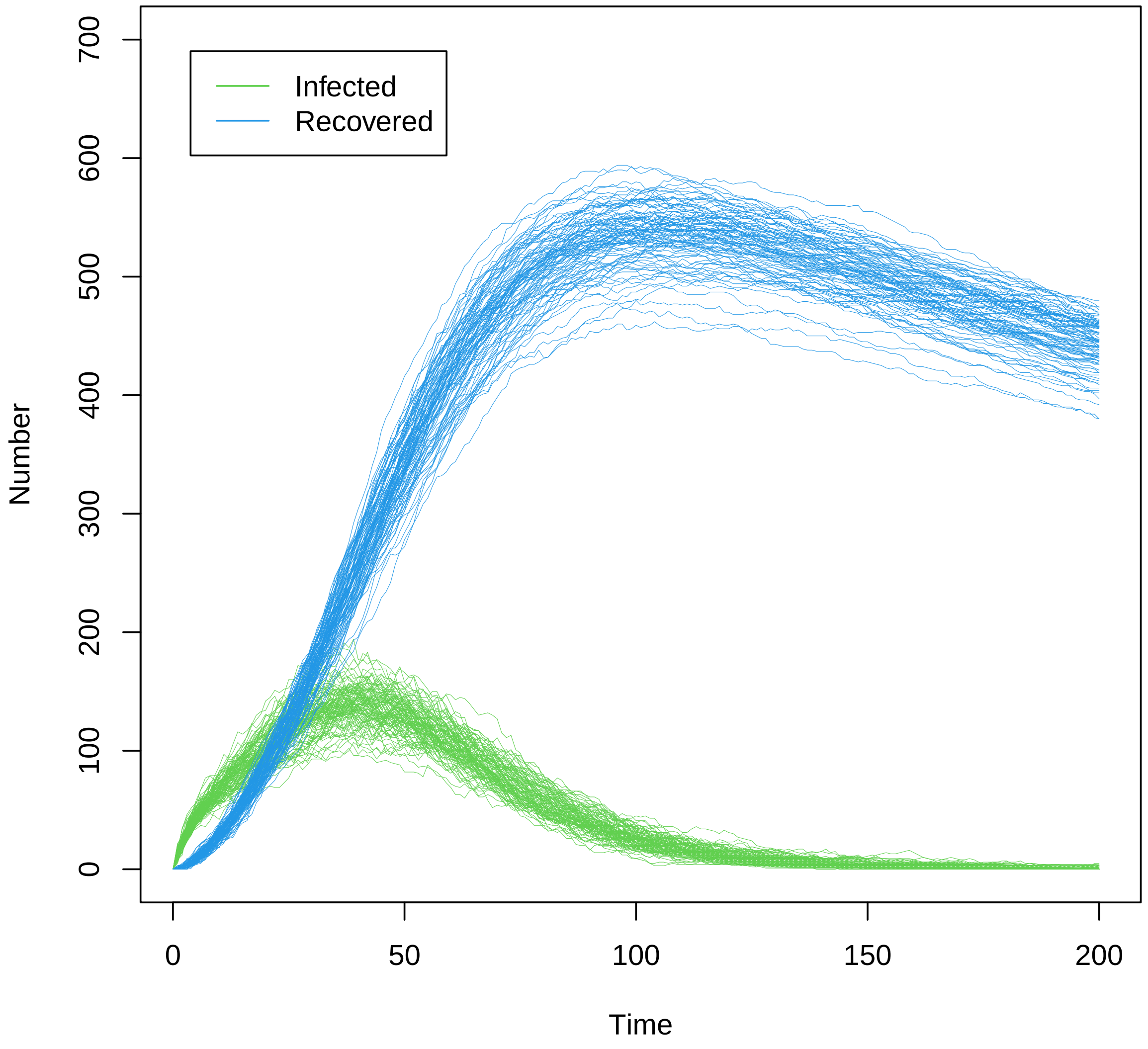
Using `get_results` on `chosen_params`

```r
solution <- get_results(chosen_params, outs = c("I", "R"),
                        times = c(25, 40, 100, 200), raw = TRUE)
```

we get a 3-dimensional array `solution` such that `solution[t,j,i]` contains the number of individuals in the j-th compartment at time t for the i-th run of the model at `chosen_params`. In particular, t can take values $1, 2, ..., 201$, j can take values $1, 2, 3, 4, 5$ corresponding to S, E, I, R, D (where D stands for the cumulative number of deaths occurred), and i can be $1, 2, 3, ..., 100$.

Plotting the results for “I” and “R”, we have

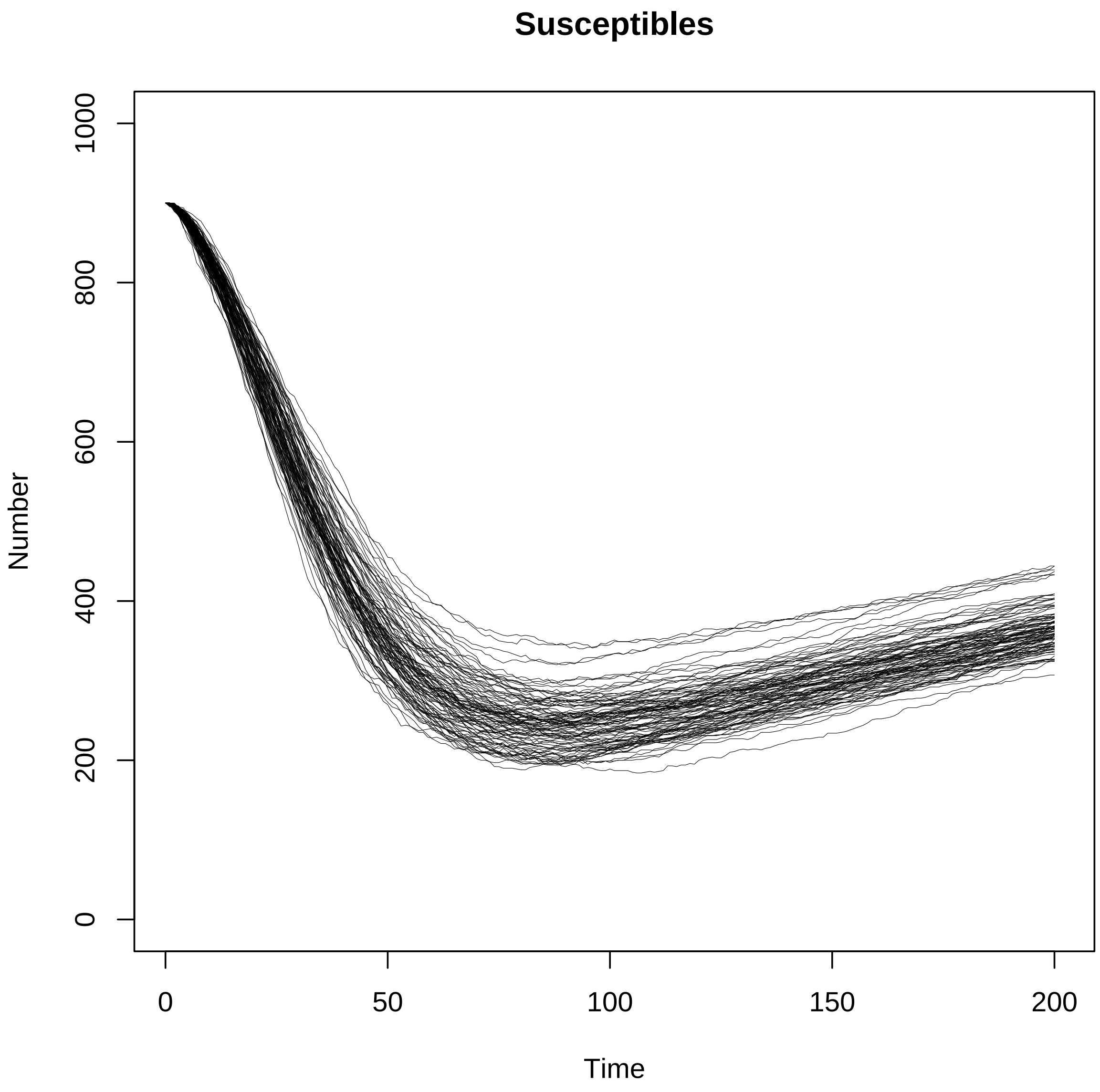
```r
plot(0:200, ylim=c(0,700), ty="n", xlab = "Time", ylab = "Number")
for(j in 3:4) for(i in 1:100) lines(0:200, solution[,j,i], col=(3:4)[j-2], lwd=0.3)
legend('topleft', legend = c('Infected', "Recovered"), lty = 1,
       col = c(3,4), inset = c(0.05, 0.05))
```



The plot above clearly shows the stochasticity of our model. Furthermore we can also appreciate how the spread of the runs varies across the time: for example, the variance in the number of infected individuals is largest around $t = 40$ and quite small at later times.

Plotting the results for "S" also shows stochasticity:

```r
plot(0:200, ylim=c(0,1000), ty="n", xlab = "Time", ylab = "Number", main = "Susceptibles")
for(i in 1:100) lines(0:200, solution[,1,i], col='black', lwd=0.3,
                      xlab = "Time", ylab = "Number", main = "Susceptibles")
```



Task 1

If you would like, familiarise yourself with the model. Investigate how the plots change as you change the values of the parameters.

Show: R tip on P**??**

Show: Solution on P**??**

Chapter 4

'wave0' - parameter ranges, targets and design points

A video presentation of this section can be found here.

In this section we set up the emulation task, defining the input parameter ranges, the calibration targets and all the data necessary to build the first wave of emulators.

Note that, for the sake of clarity, in this workshop we will adopt the word 'data' only when referring to the set of runs of the model that are used to train emulators. Empirical observations, that would inform our choice of targets if we were modelling a real-world scenario, will instead be referred to as 'observations'.

First of all, let us set the parameter ranges:

```r
ranges = list(
  b = c(1e-5, 1e-4), # birth rate
  mu = c(1e-5, 1e-4), # rate of death from other causes
  beta1 = c(0.05, 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
  epsilon = c(0.01, 0.21), # rate of becoming infectious after infection
  alpha = c(0.01, 0.025), # rate of death from the disease
  gamma = c(0.01, 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 $t = 25, 40, 100, 200$. The targets can be specified by a pair (val, sigma), where 'val' represents the measured value of the output and 'sigma' represents its standard deviation, or by a pair (min, max), where min represents the lower bound and max the upper bound for the target. Since in Workshop 1 we used the former formulation, here we will show the latter:

```
targets <- list(
  I25 = c(98.51, 133.25),
  I40 = c(117.17, 158.51),
  I100 = c(22.39, 30.29),
  I200 = c(0.578, 0.782),
  R25 = c(106.34, 143.9),
  R40 = c(218.28, 295.32),
  R100 = c(458.14, 619.84),
  R200 = c(377.6, 510.86)
)
```

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.

For this workshop, we generate parameter sets using a Latin Hypercube design. A rule of thumb is to select at least $10p$ parameter sets to train emulators, where p is the number of parameters (9 in this workshop). Using the function `maximinLHS` in the package `lhs`, we create a hypercube design with 100 parameter sets for the training set and one with 50 parameter sets for the validation set. Note that we could have chosen 100 parameter sets for the validation set: we chose 50 simply to highlight that it is not necessary to have as many validation points as training points. Especially when working with expensive models, this can be useful and can help us manage the computational burden of the procedure. After creating the two hypercube designs, we bind them together to create `initial_LHS`:

```
initial_LHS_training <- maximinLHS(100, 9)
initial_LHS_validation <- maximinLHS(50, 9)
initial_LHS <- rbind(initial_LHS_training, initial_LHS_validation)
```

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 * purrr::map_dbl(ranges, diff) +
                                        purrr::map_dbl(ranges, ~.[[1]])))), names(ranges))
```

We then run the model for the parameter sets in `initial_points` through the `get_results` function, specifying that we are interested in the outputs for I and R at times $t = 25, 40, 100, 200$. Note that we use the `progressr` package to create a progress bar that updates every time a new parameter set is run: when you run the code on your own, this will indicate at what stage of the process you are.

```
initial_results <- list()
with_progress({
  p <- progressor(nrow(initial_points))
for (i in 1:nrow(initial_points)) {
  model_out <- get_results(unlist(initial_points[i,]), nreps = 25, outs = c("I", "R"),
                           times = c(25, 40, 100, 200))
  initial_results[[i]] <- model_out
  p(message = sprintf("Run %g", i))
}
})
```

Note that `initial_results` is a list of length 3750, since it has a row for each of the 25 repetitions of the 150 parameter sets in `initial_points`. Finally, we bind all elements in `initial_results` to obtain a data frame `wave0` and we split it in two parts: the first 2500 elements (corresponding to the first 100 parameter sets) for the training set, `all_training`, and the last 1250 for the validation set (corresponding to the last 50 parameter sets), `all_valid`.

```
wave0 <- data.frame(do.call('rbind', initial_results))
all_training <- wave0[1:2500,]
all_valid <- wave0[2501:3750,]
output_names <- c("I25", "I40", "I100", "I200", "R25", "R40", "R100", "R200")
```

Chapter 5

Emulators

A video presentation of this section can be found here.

In this section we will train stochastic emulators, which take into account the fact that each time the model is run using the same parameter set, it will generate different output.

5.1 Brief recap on the structure of an emulator

We very briefly recap the general structure of a univariate emulator (see Workshop 1 for more details):

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

where $g(x)^T \xi$ is a regression term and $u(x)$ is a weakly stationary process with mean zero.

The regression term, which mimics the global behaviour of the model output, 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, and a vector of regression coefficients ξ.

The weakly stationary process $u(x)$ (similar to a Gaussian process), accounts for the local deviations (or residuals) of the output from the regression hypersurface.

In this workshop, $u(x)$ is assumed to be a Gaussian process, with covariance structure given by

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

where c is the square-exponential correlation function

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

where x_i is the ith-component of the parameter set x. The term σ^2 is the **emulator variance**, i.e. the variance of $u(x)$, and reflects how far we expect the output to be from the regression hypersurface, while θ is the **correlation length** of the process, and determines how close two parameter sets must be in order for the corresponding residuals to be non-negligibly correlated.

In the deterministic workshop, we said that in order to train an emulator, the `emulator_from_data` function first estimated the value for σ, using the provided training data. It is important to note that **this estimated value of σ was constant**, i.e., it was the same for all parameter sets. When working with stochastic models, different areas of the parameter space often give rise to quite different levels of stochasticity in the outputs. Taking this phenomenon into account when dealing with stochastic models will allow us to train more accurate and efficient emulators.

5.2 Stochastic emulators

To train stochastic emulators we use the function `emulator_from_data` specifying `emulator_type = 'variance'`, which requires the training data, the names of the outputs to emulate, and the ranges of the parameters:

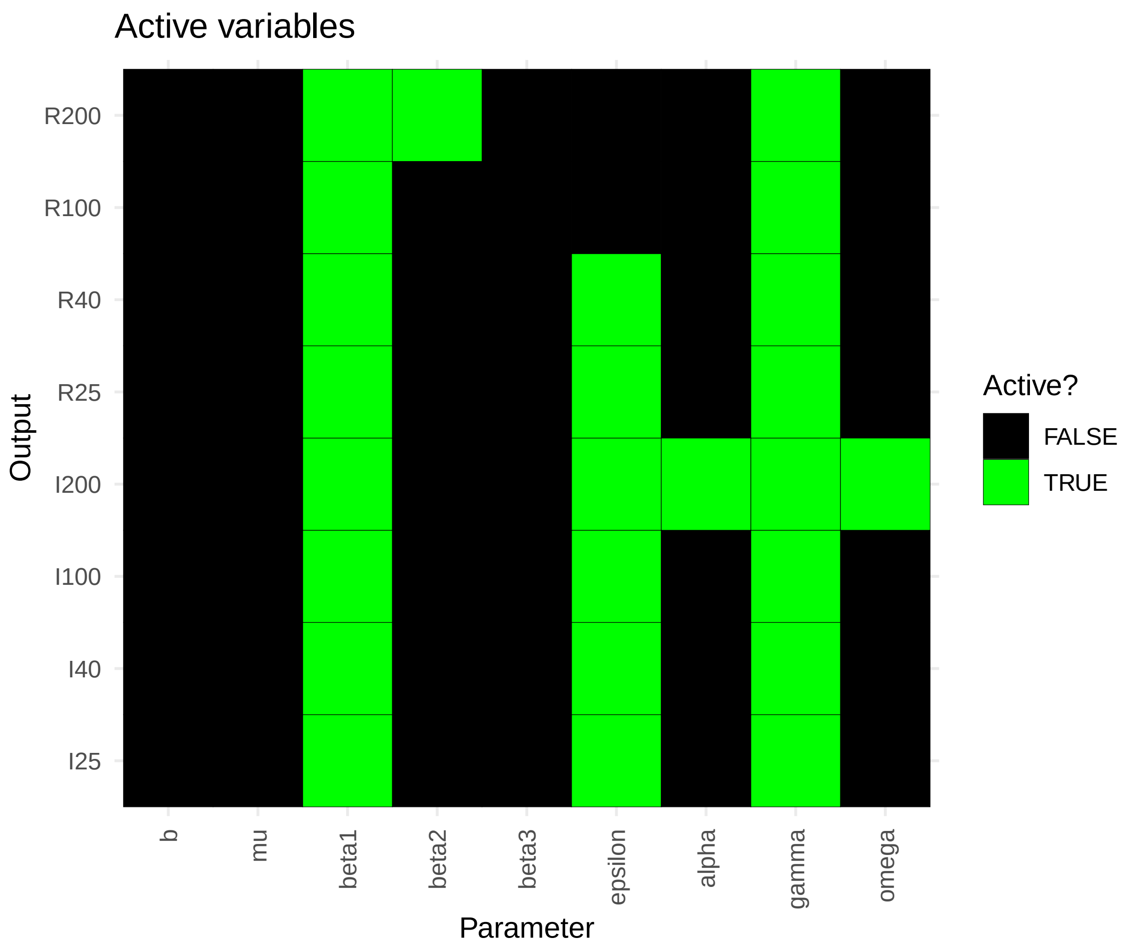
```
stoch_emulators <- emulator_from_data(all_training, output_names, ranges,
                                      emulator_type = 'variance')
```

The function `emulator_from_data` with `emulator_type = 'variance'` returns two sets of emulators: one for the variance and one for the expectation of each model output. Behind the scenes, `emulator_from_data` does the following:

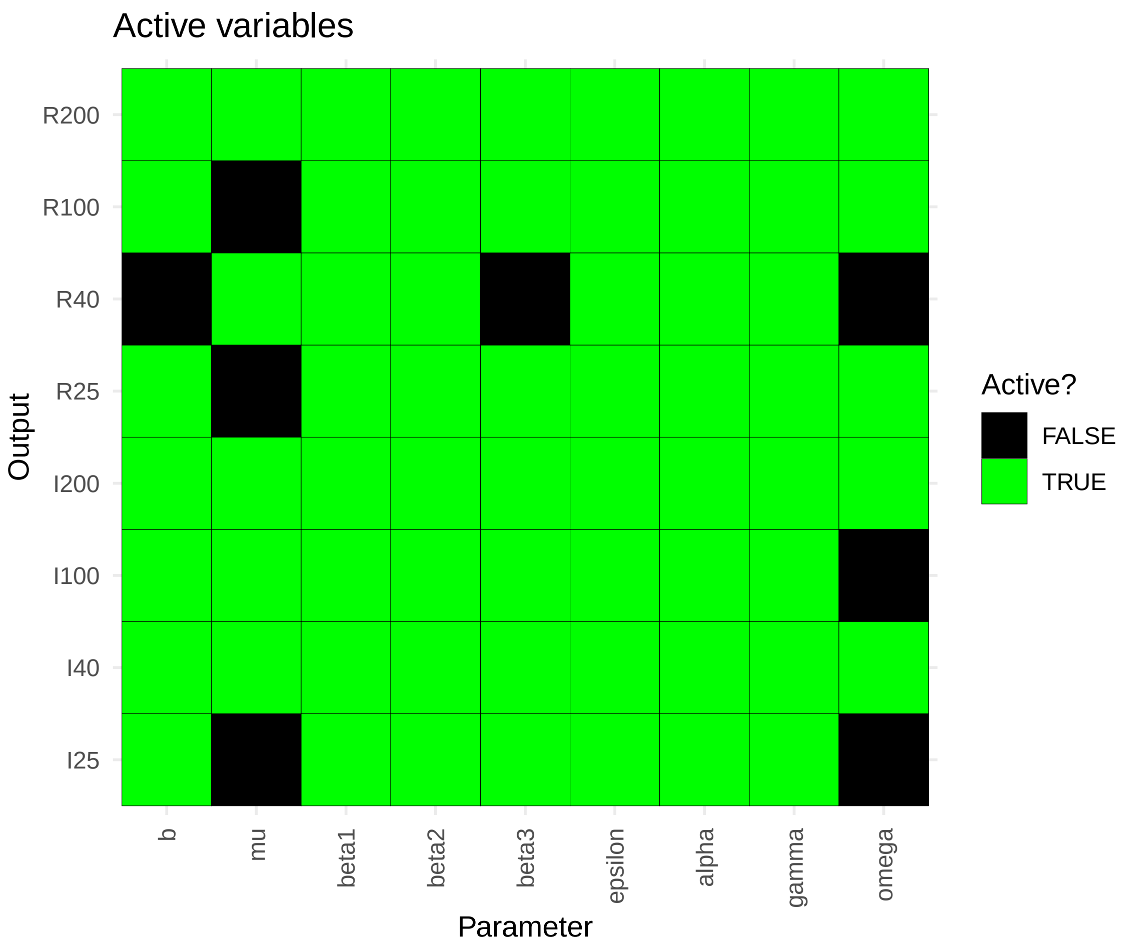
- First, it estimates the variance of each model output of interest at each provided parameter set: this is possible since `all_training` contains several model runs at each parameter set.
- Using the obtained variance data, it then trains an emulator for the variance of each of the outputs of interest. These emulators will be referred to as **variance emulators** and can be access typing `stoch_emulators$variance`.
- Finally, emulators for the mean of each output are built. These emulators will be referred to as mean emulators and can be accessed typing `stoch_emulators$expectation`. The main difference in the training of emulators between the stochastic and the deterministic case is that here, for each output, we use the variance emulator to inform our choice for a component of σ, that which is due to the stochasticity of the model: for each parameter set x, the variance emulator gives us an estimate of the variance of the model output at x. Note that this step is specific to the treatment of stochastic models, where it is important to account for the fact that different areas of the parameter space often give rise to quite different levels of stochasticity in the outputs.

Let's take a look at the two sets of emulators stored in `stoch_emulators`, starting with plots of active variables:

```r
plot_actives(stoch_emulators$variance)
```



```r
plot_actives(stoch_emulators$expectation)
```

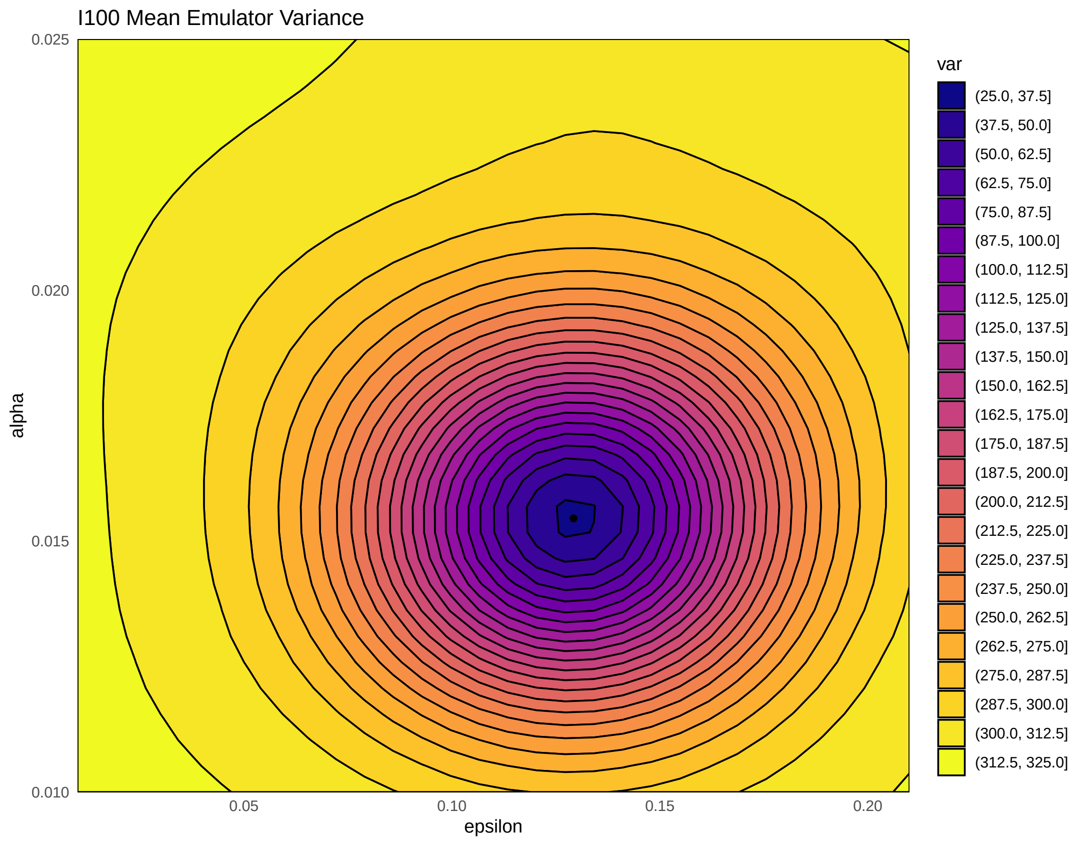


These plots show what variables are active, i.e. have the most explanatory power, for each of the mean and variance emulators. We see that b and μ are inactive for almost all outputs in the variance emulators, even though they are active for some outputs in the mean emulators. Parameters β_1, ϵ, α and γ are active for most or all outputs in the mean emulators.

As in the deterministic case, the `emulator_plot` can be used to visualise how emulators represent the output space. Since we have two sets of emulators, we have several options in terms of plots. The default behaviour of `emulator_plot` is to plot the expectation of the passed emulator: for example, to plot the expectation of the variance emulator for $I40$, we just need to pass `stoch_emulators$variance$I40` to `emulator_plot`.

It is important to stress the following point: unlike in the deterministic case, here mean emulators do not have zero variance at 'known' points, i.e., at points in the training set. Let us verify this, by plotting the variance of the mean emulator for $I100$ in the (ϵ, α)-plane, with all unshown parameters to be as in the first row of `all_training`. Note that since the plot produced below depends on the location of the first run, when running the code with your own `all_training` you will get a different result:

```
emulator_plot(stoch_emulators$expectation$I100, params = c('epsilon', 'alpha'),
     fixed_vals = all_training[1, names(ranges)[-c(6,7)]], plot_type = 'var') +
   geom_point(data = all_training[1,], aes(x = epsilon, y = alpha))
```



We see that the black point in the dark blue area, corresponding to the first point in `all_training`, does not have variance zero. This is because the mean emulator is trying to predict the 'true' mean at the point, but it has only been given a sample mean, which it cannot assume is equal to the 'true' mean. The emulator internally compensates for this incomplete information, resulting in the variance not being zero, even at points in the training set. This compensation also means that we can work with different numbers of replicates at different parameter sets. In general, for each training/validation parameter set, we should use as many repetitions as is feasible given the model complexity. More repetitions will provide a better estimate of the variance, which in turn allows to train more accurate mean emulators. If a model is relatively fast, then it is worth doing 50-100 repetitions per parameter set; if it is slower, then we can work with fewer repetitions (even just 5-10 per parameter set). This applies to different regions of the parameter space within a single model, too: if one part of the parameter space is much slower to run (e.g. parameter sets with higher beta values, which tend to produce more infections, slowing down agent-based models), then we could run fewer repetitions in those parts of parameter space.

Chapter 6

Implausibility

A video presentation of this section can be found here.

We will first briefly review the definition of an implausibility measure.

For a given stochastic model output and a given target, the implausibility measures the difference between the output and the target, taking into account all sources of uncertainty. For a parameter set x, a particular choice of implausibility, useful for either matching to the mean of the stochastic model, or its spread of possible values, is:

$$\text{Imp}(x) = \frac{|\text{E}[m(x)] - z|}{\sqrt{V_0 + V_c(x) + V_s(x) + V_m}},$$

where $\text{E}[m(x)]$ is the expected value of the mean of the stochastic model, z the target, and the terms in the denominator refer to various forms of uncertainty. In particular

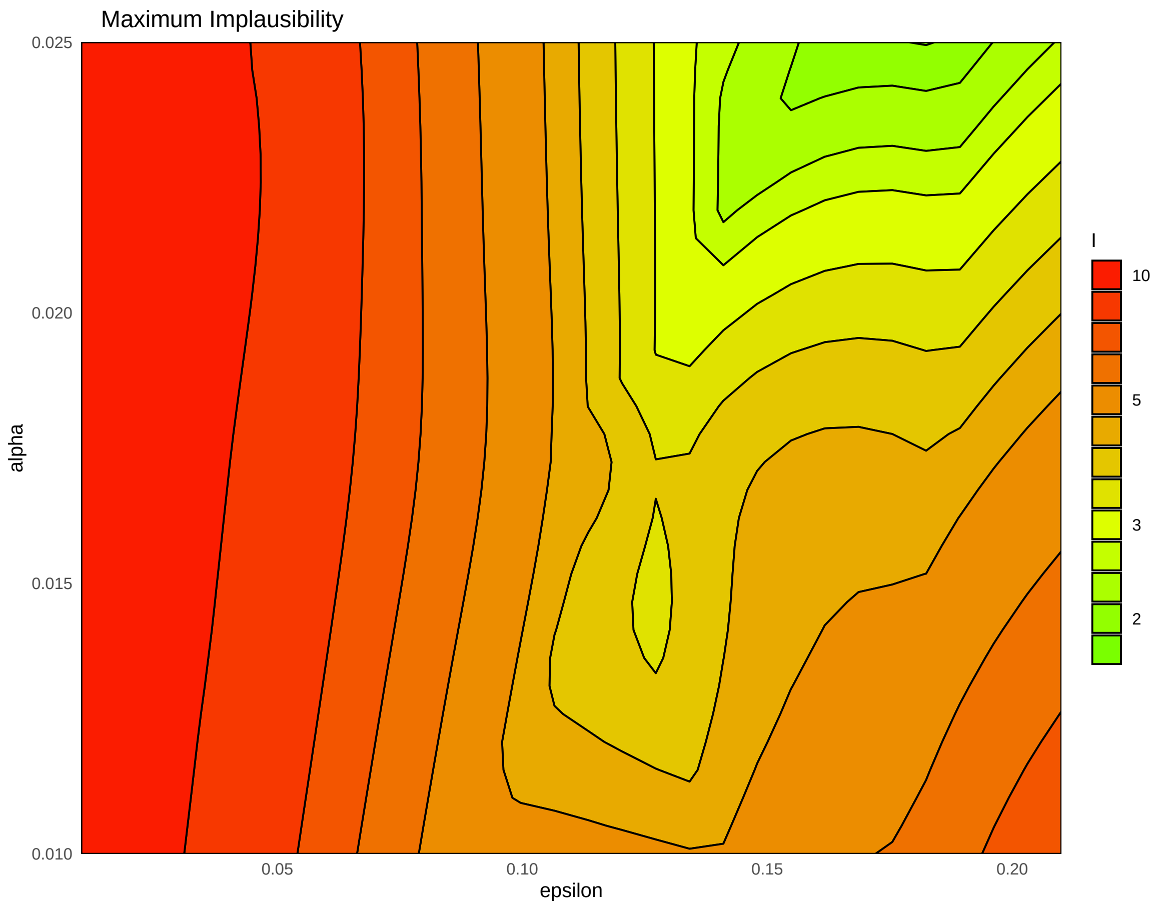
- V_0 is the variance associated with the observation uncertainty;
- $V_c(x)$ refers to the uncertainty one introduces when using the emulator output instead of the model output itself;
- $V_s(x)$ is the ensemble variability and represents the stochastic nature of the model. This term is used to create an appropriate representation of the uncertainty on either the mean or the set of possible realisations of the stochastic model.
- 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 $V_m = 0$. The term V_s, which was zero when we worked on the deterministic SEIRS model (Workshop 1), is now non-zero, in order to account for the stochasticity of the model.

A very large value of $\text{Imp}(x)$ 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.

To plot the maximum implausibility (across all emulators), we set `plot_type` to `nimp` in the call to `emulator_plot`:

```r
emulator_plot(stoch_emulators, plot_type = 'nimp',
              targets = targets, params = c('epsilon', 'alpha'))
```



Task 2

Use the argument `fixed_vals` to set the parameters that are not shown in the plot to be as in `chosen_params`. Verify that the implausibility at `chosen_params` is below 3.

Show: Solution on P??

Chapter 7

Emulator diagnostics

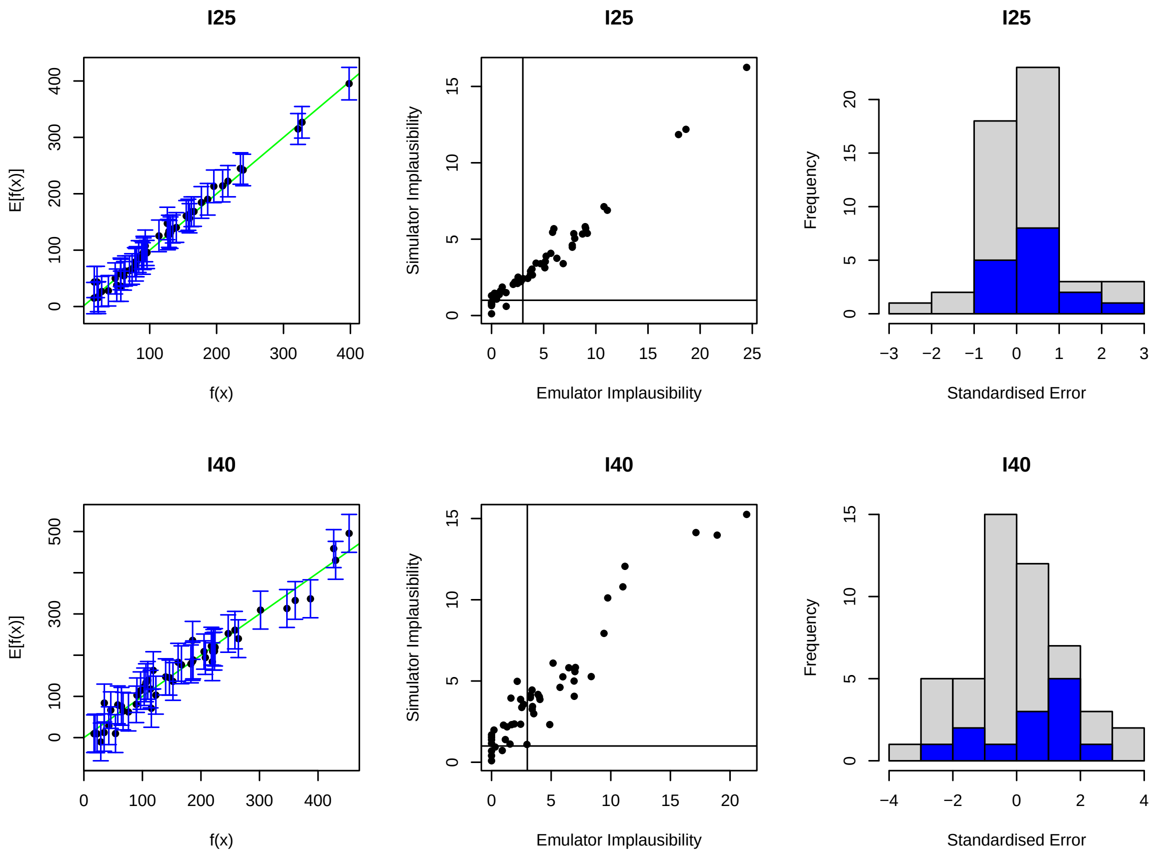
A video presentation of this section can be found here.

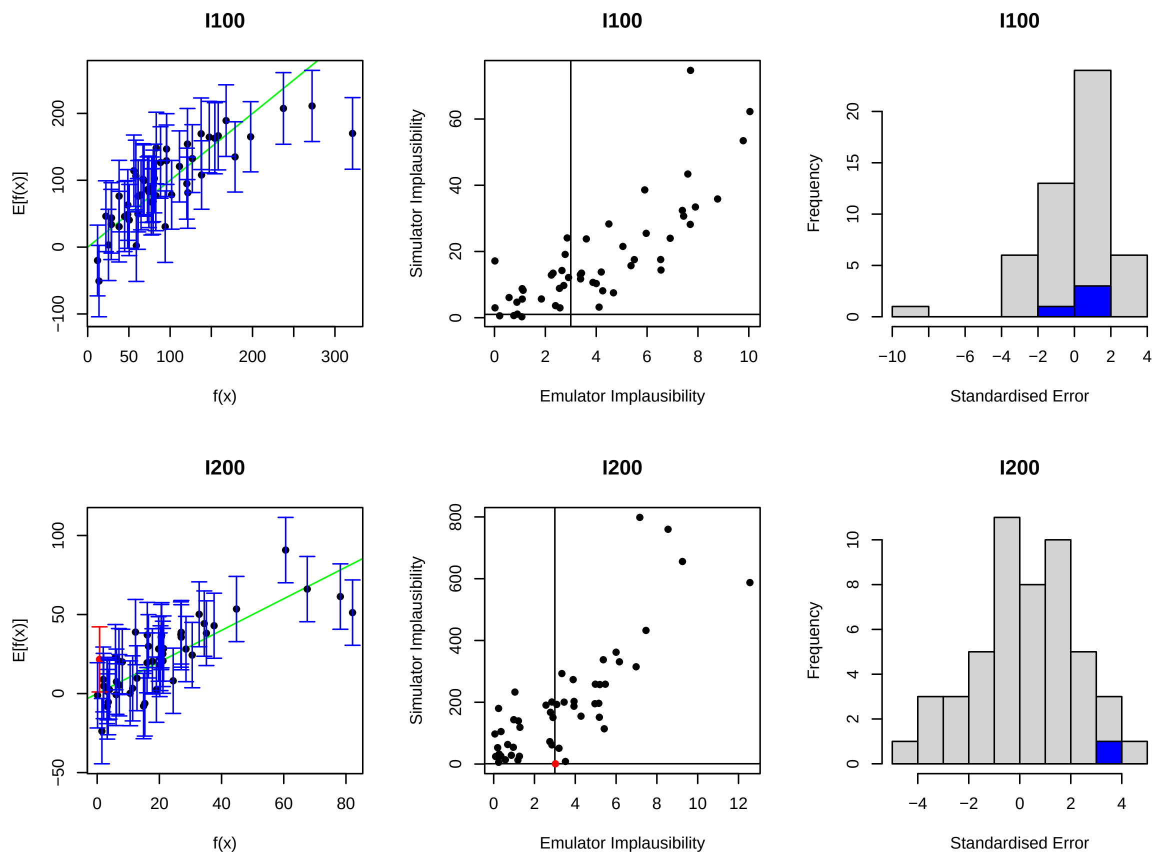
The function `validation_diagnostics` can be used as in the deterministic case, to get three diagnostics for each emulated output.

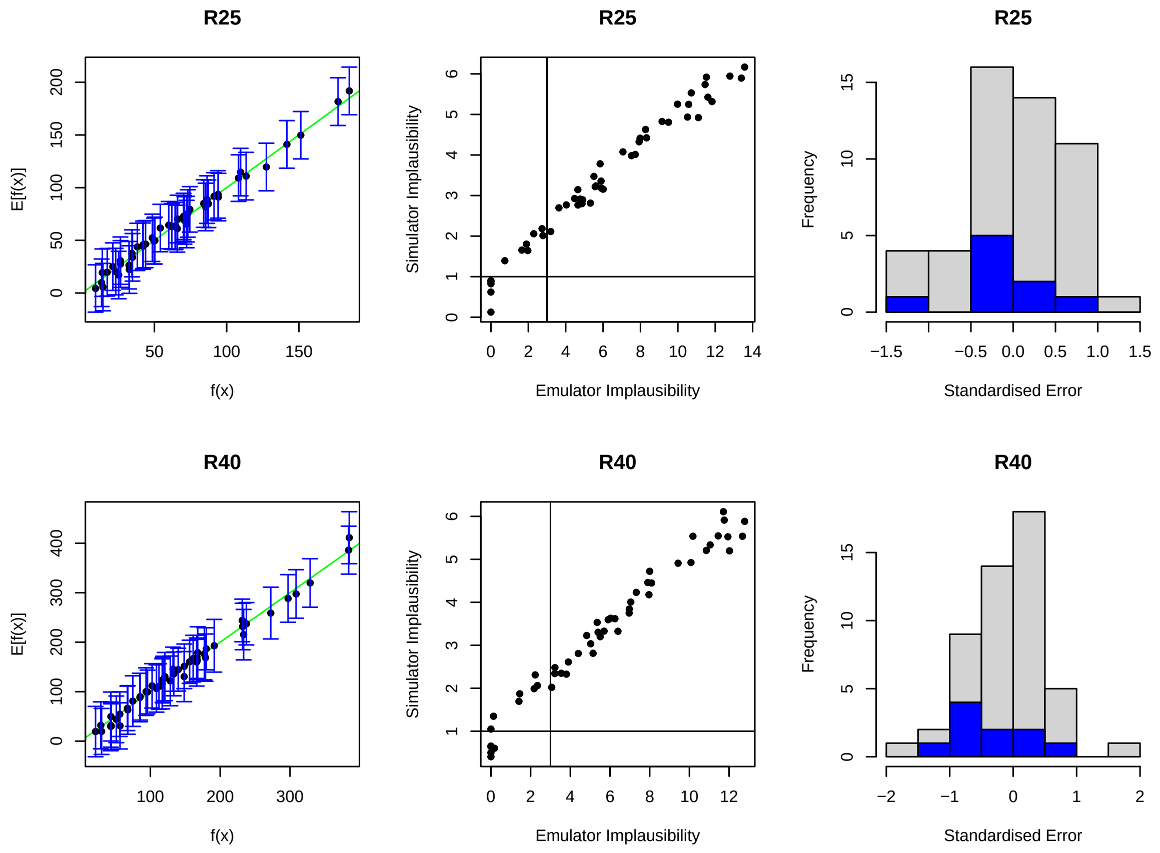
Show: Remind me of what each diagnostics means on P??

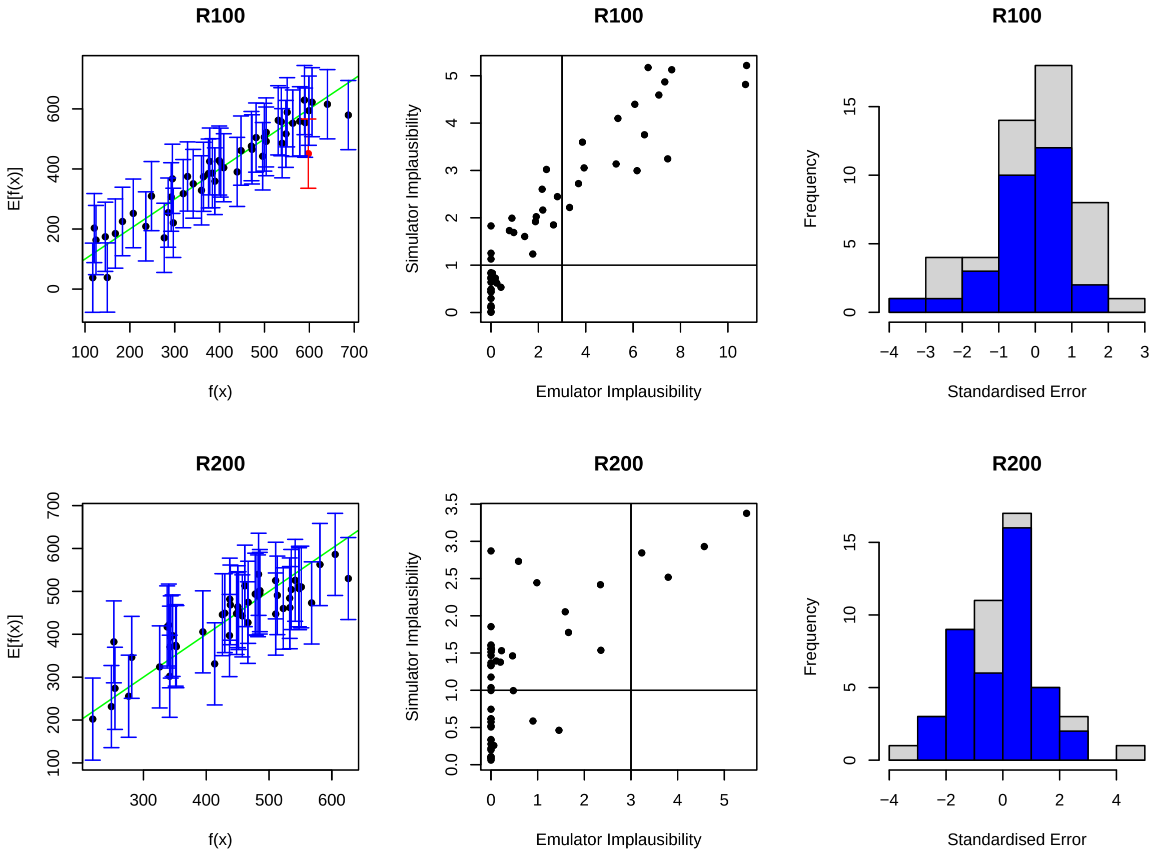
We call `validation_diagnostics` passing it the mean emulators, the list of targets, the validation set:

```
vd <- validation_diagnostics(stoch_emulators$expectation, targets, all_valid, plt=TRUE, row=2)
```


I25
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error
I40


I100
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error
I200


R25
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error
R40



Note that by default `validation_diagnostics` groups the outputs in sets of three. Since here we have 8 outputs, we set the argument `row` to 2: this is not strictly necessary but improves the format of the grid of plots obtained. As in deterministic case, we can enlarge the σ values of the mean emulators to obtain more conservative emulators, if needed. Based on the diagnostics above, all emulators perform quite well, so we won't modify any σ values here. In order to access a specific mean emulator, you can type `ems$expectation$variable_name`. For example, to double the σ of the mean emulator for $I25$, you would type `stoch_emulators$expectation$I25 <- stoch_emulators$expectation$I25$mult_sigma(2)`.

Note that the output `vd` of the `validation_diagnostics` function is a dataframe containing all parameter sets in the validation dataset that fail at least one of the three diagnostics. This dataframe can be used to automate the validation step of the history matching process. For example, one can consider the diagnostic check to be successful if `vd` contains at most 5% of all points in the validation dataset. In this way, the user is not required to manually inspect each validation diagnostic for each emulator at each wave of the process.

It is possible to automate the validation process. If we have a list of trained emulators `ems`, we can start with an iterative process to ensure that emulators do not produce misclassifications:

1. check if there are misclassifications for each emulator (middle column diagnostic in plot above)

with the function `classification_diag`;

2. in case of misclassifications, increase the `sigma` (say by 10%) and go to step 1.

The code below implements this approach:

```
for (j in 1:length(ems)) {
      misclass <- nrow(classification_diag(ems$expectation[[j]], targets, validation, plt = FALS
      while(misclass > 0) {
        ems$expectation[[j]] <- ems$expectation[[j]]$mult_sigma(1.1)
        misclass <- nrow(classification_diag(ems$expectation[[j]], targets, validation, plt = FA
      }
}
```

The step above helps us ensure that our emulators are not overconfident and do not rule out parameter sets that give a match with the empirical data.

Once misclassifications have been eliminated, the second step is to ensure that the emulators' predictions agree, within tolerance given by their uncertainty, with the simulator output. We can check how many validation points fail the first of the diagnostics (left column in plot above) with the function `comparison_diag`, and discard an emulator if it produces too many failures. The code below implements this, removing emulators for which more than 10% of validation points do not pass the first diagnostic:

```
bad.ems <- c()
for (j in 1:length(ems)) {
          bad.model <- nrow(comparison_diag(ems$expectation[[j]], targets, validation, plt = FAL
          if (bad.model > floor(nrow(validation)/10)) {
            bad.ems <- c(bad.ems, j)
   }
}
ems <- subset_emulators(ems, names(targets)[!seq_along(ems) %in% bad.ems])
```

Note that the code provided above gives just an example of how one can automate the validation process and can be adjusted to produce a more or less conservative approach. Furthermore, it does not check for more subtle aspects, e.g. whether an emulator systematically underestimates/overestimates the corresponding model output. For this reason, even though automation can be a useful tool (e.g. if we are running several calibration processes and/or have a long list of targets), we should not think of it as a full replacement for a careful, in-depth inspection from an expert.

Chapter 8

Proposing new points

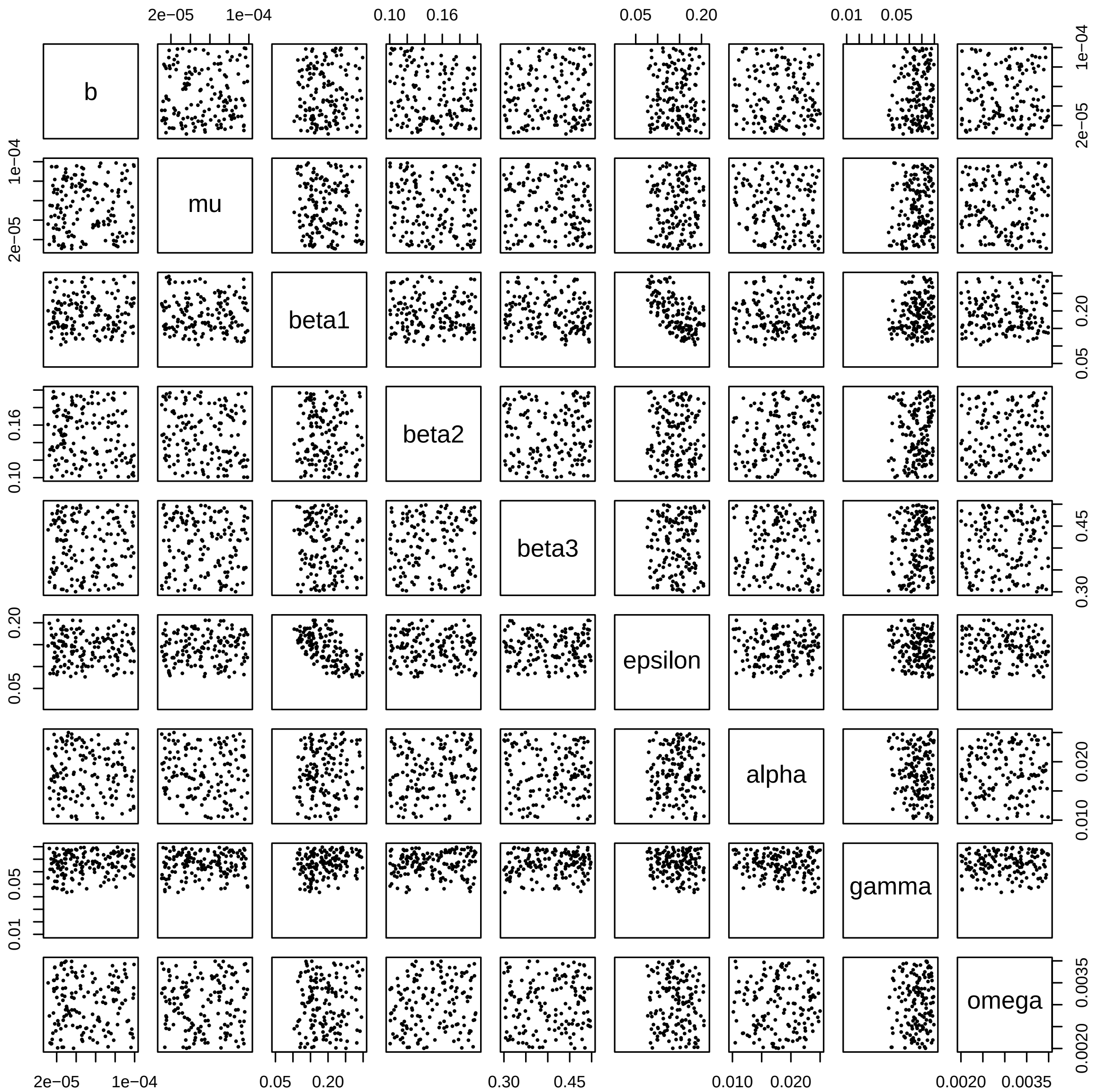
A video presentation of this section can be found here.

To generate a set of non-implausible points, based on the trained emulators, we use the function `generate_new_design`, exactly as in the deterministic case:

```r
new_points <- generate_new_design(stoch_emulators, 150, targets)
```

Here we generated 150 points, since we are going to use 100 of them to train new emulators and 50 of them to validate new emulators (as done in the first wave). We can visualise the non-implausible space at the end of this first wave using `plot_wrap`:

```r
plot_wrap(new_points, ranges)
```



Here we see which parameters are more or less constrained at the end of the first wave. For example, it seems clear that low values of γ cannot produce a match (cf. penultimate column). We can also deduce relationships between parameters: β_1 and ϵ are an example of negatively-correlated parameters. If β_1 is large then ϵ needs to be small, and vice versa. Other parameters, such as ω or μ, are instead still spread out across their initial range.

As shown in Tutorial 2, it is also possible to perform a full wave of emulation and history matching using the function `full_wave`, which needs the following information:

- A dataset that will be split by the function into training data and test data;

- A list of ranges for the parameters;
- The targets: for each of the model outputs to emulate, we need a pair (val, sigma) or (min, max) that will be used to evaluate implausibility.

Chapter 9

Second wave

A video presentation of this section can be found here.

To perform a second wave of history matching and emulation we follow the same procedure as in the previous sections, with two caveats. First, since the parameter sets in `new_points` tend to lie in a small region inside the original input space, we will train the new emulators only on the non-implausible region found in wave one. This can be done simply setting the argument `check.ranges` to `TRUE` in the function `emulator_from_data`.

It is a good strategy to increase the number of repetitions used at each wave: this, paired with the fact later waves emulators are trained on a reduced space, allows to create more and more precise emulators. Using this principle, we run the model 50 times on each parameter set in `new_points` (for the first wave we used 25 repetitions). We then bind all results to create a dataframe `wave1` and we split this into two subsets, `new_all_training` to train the new emulators and `new_all_valid` to validate them. We select the first 5000 rows (corresponding to the first 100 parameter sets in `new_points`) for the training data and the last 2500 (corresponding to the last 50 parameter sets in `new_points`) for the validation data.

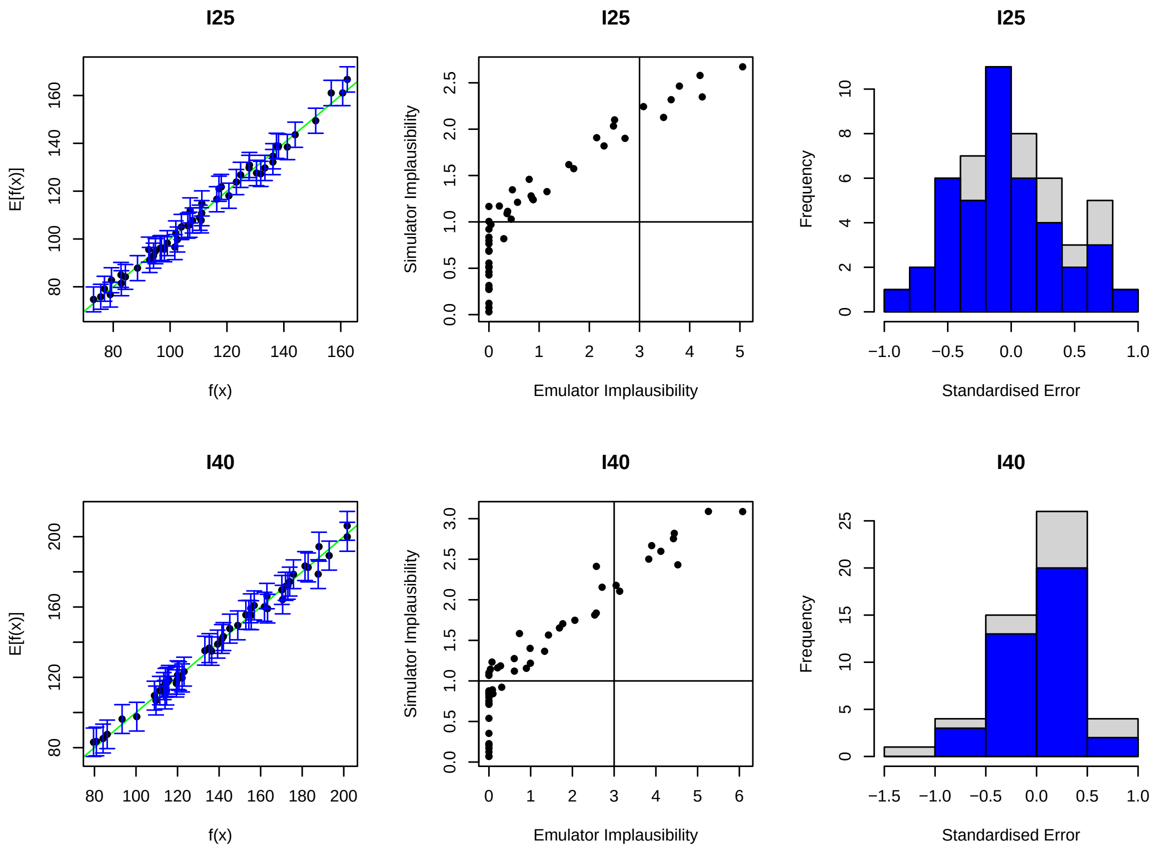
```r
new_results <- list()
with_progress({
  p <- progressor(nrow(initial_points))
  for (i in 1:nrow(new_points)) {
  model_out <- get_results(unlist(new_points[i,]), nreps = 50, outs = c("I", "R"),
                           times = c(25, 40, 100, 200))
  new_results[[i]] <- model_out
  p(message = sprintf("Run %g", i))
  }
})
wave1 <- data.frame(do.call('rbind', new_results))
new_all_training <- wave1[1:5000,]
new_all_valid <- wave1[5001:7500,]
```

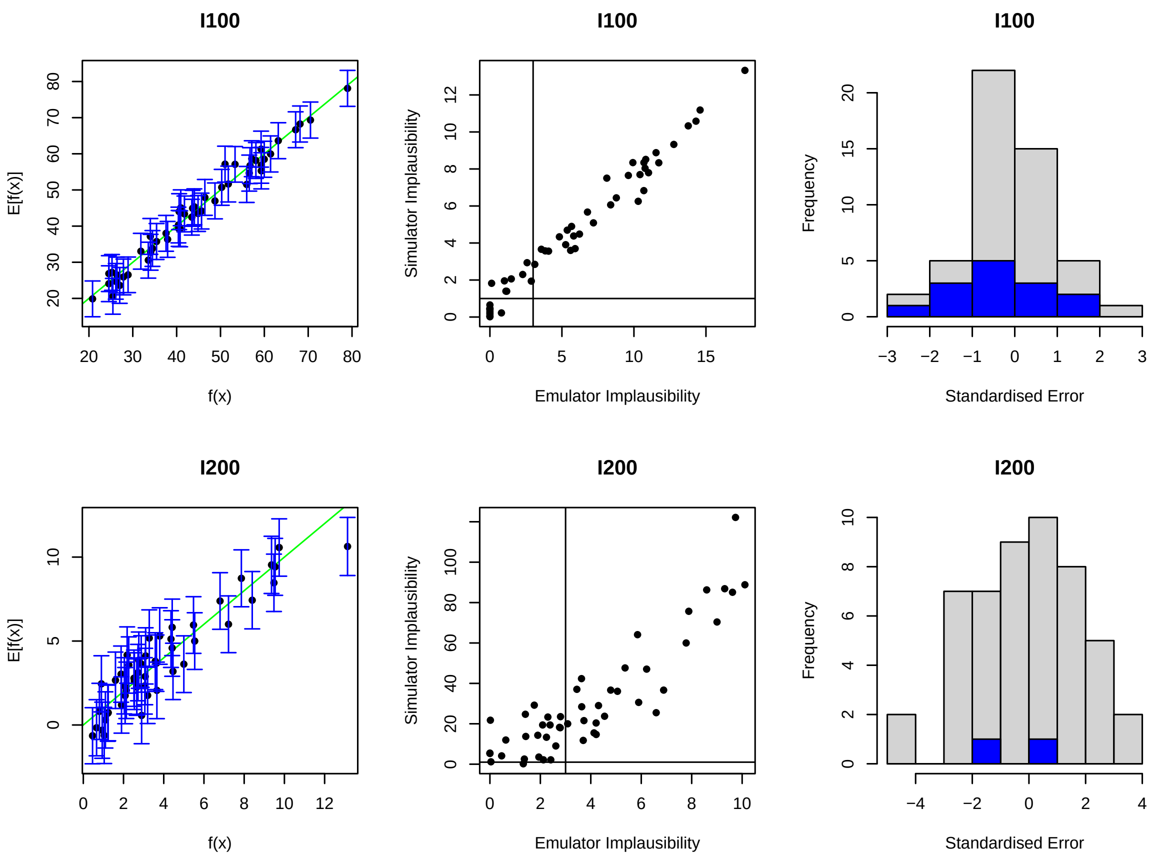
To train new emulators we use `emulator_from_data`, passing the new training data, the outputs names and the initial ranges of the parameters. We also set `check.ranges` to `TRUE` to ensure that the new emulators are trained only on the non-implausible region found in wave one, and `emulator_type = 'variance'` to indicate that we want to train stochastic emulators.

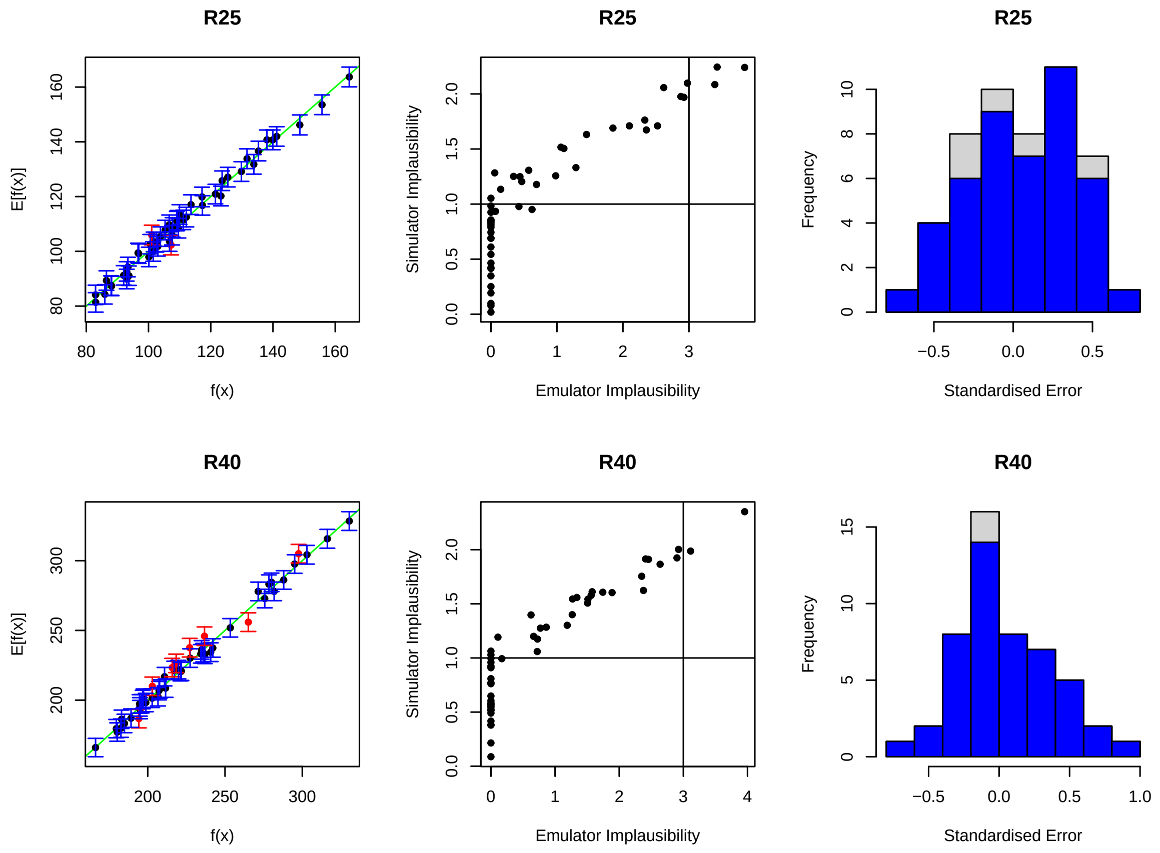
```
new_stoch_emulators <- emulator_from_data(new_all_training, output_names, ranges,
                                          check.ranges=TRUE, emulator_type = 'variance')
```

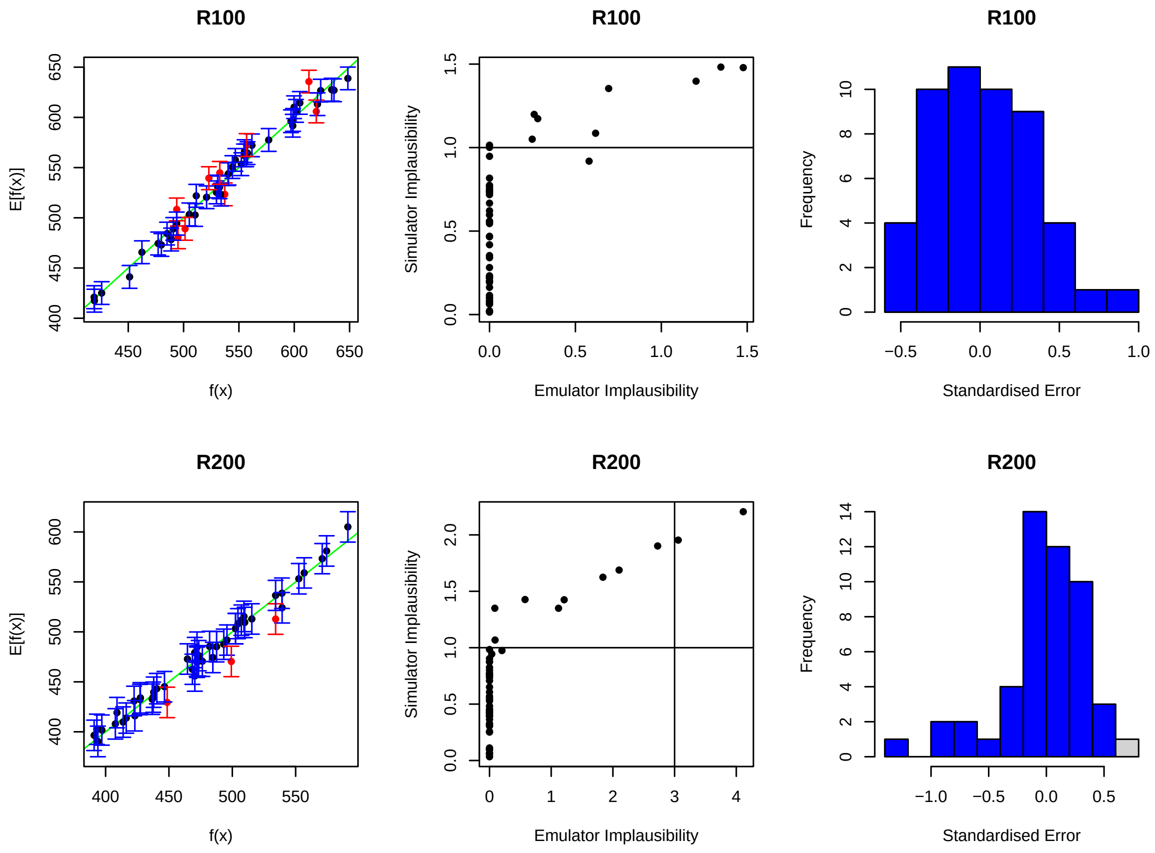
As usual, before using the obtained emulators, we need to check their diagnostics. We use the function `validation_diagnostics` which takes the new emulators, the targets and the new validation data:

```
vd <- validation_diagnostics(new_stoch_emulators, targets, new_all_valid, plt=TRUE, row=2)
```




I100
I100
I100
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error
I200
I200
I200
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error


R25
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error
R40
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error



Since these diagnostics look good, we can generate new non-implausible points. Here is the second caveat: we now need to pass both `new_stoch_emulators` and `stoch_emulators` to the `generate_new_design` function, since a point needs to be non-implausible for all emulators trained so far, and not just for emulators trained in the current wave:

```
new_new_points <- generate_new_design(c(new_stoch_emulators, stoch_emulators), 150, targets)
```

Below we run the model on the points provided by the second wave of history matching:

```
new_new_results <- list()
with_progress({
  p <- progressor(nrow(initial_points))
for (i in 1:nrow(new_new_points)) {
  model_out <- get_results(unlist(new_new_points[i,]), nreps = 100, outs = c("I", "R"),
                           times = c(25, 40, 100, 200))
  new_new_results[[i]] <- model_out
  p(message = sprintf("Run %g", i))
}
})
wave2 <- data.frame(do.call('rbind', new_new_results))
```

```
new_new_all_training <- wave2[1:10000,]
```

Chapter 10

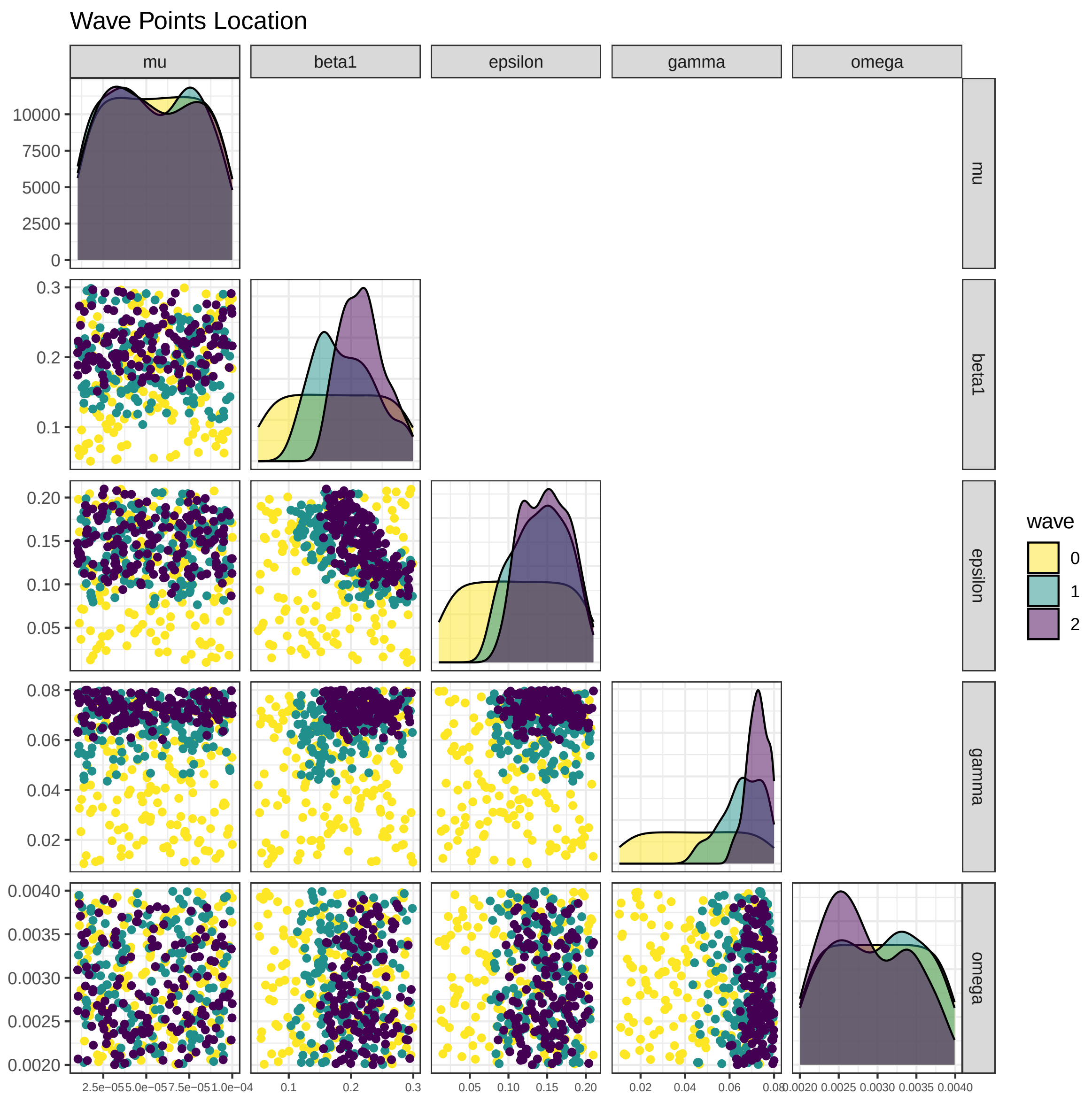
Visualisation of the non-implausible space by wave

A video presentation of this section can be found here.

In this last section we present three 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 five of the 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)[c(2,3,6,8,9)]) +
            ggplot2::theme(axis.text.x = ggplot2::element_text(size = 6))
```



Here we can easily see that the distributions of the second and third wave points are more narrow than those of the first wave.

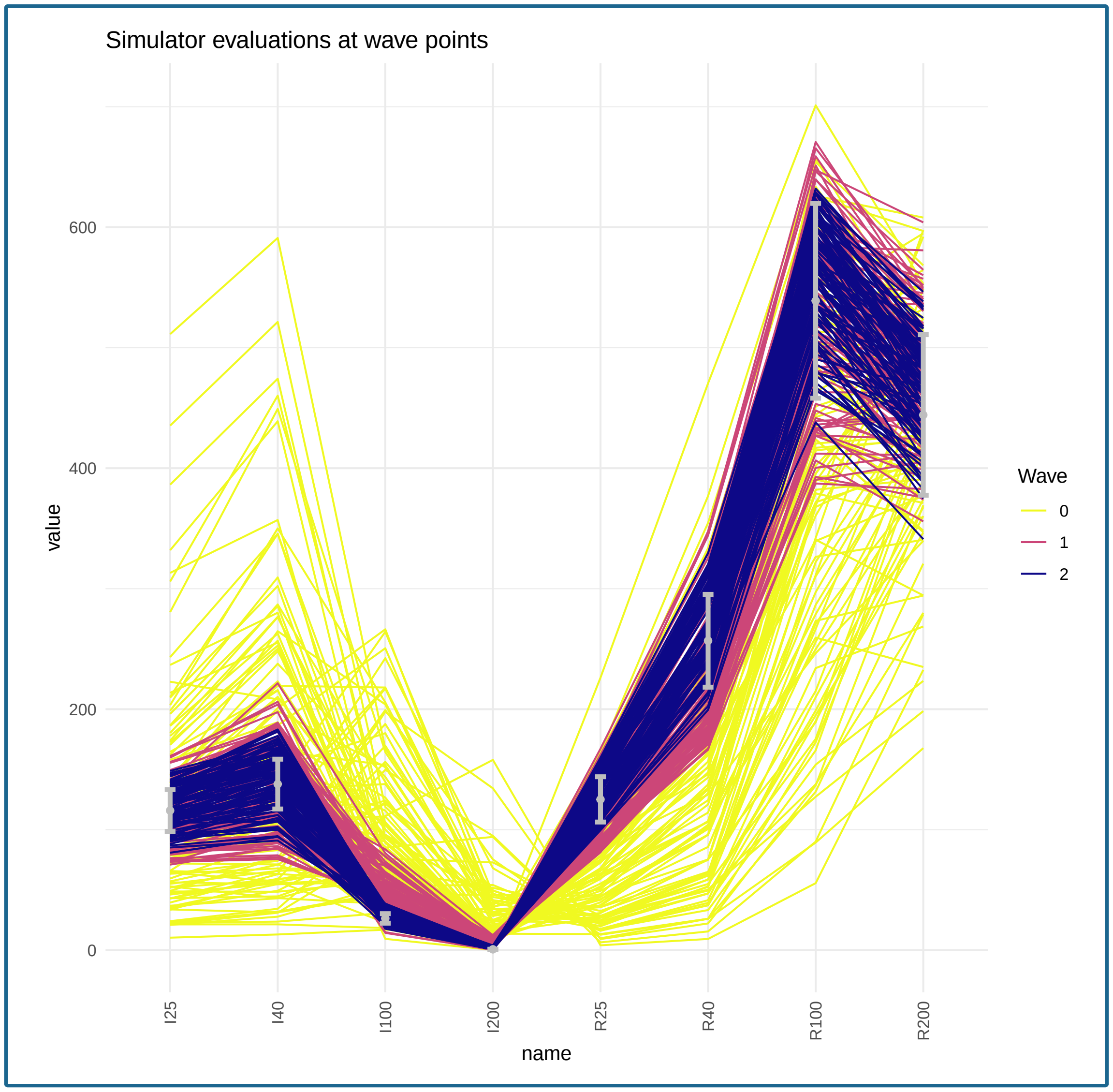
The second visualisation allows us to assess how much better parameter sets at later waves perform compared to the original `initial_points`.

In this workshop we assume that for a given parameter set x, we are interested in matching the mean of the stochastic model to the observed data, rather than in the output of each realisation at x, which will be dealt with in future versions. For this reason, we will use the helper function `aggregate_points`, which calculates the mean of each output across different realisations.

```
all_training_aggregated <- aggregate_points(all_training, names(ranges))
new_all_training_aggregated <- aggregate_points(new_all_training, names(ranges))
new_new_all_training_aggregated <- aggregate_points(new_new_all_training, names(ranges))
all_aggregated <- list(all_training_aggregated, new_all_training_aggregated,
                       new_new_all_training_aggregated)
```

We are now ready to compare the performance of parameter sets at the end of each wave, passing the list `all_aggregated` and the list `targets` to the function `simulator_plot`:

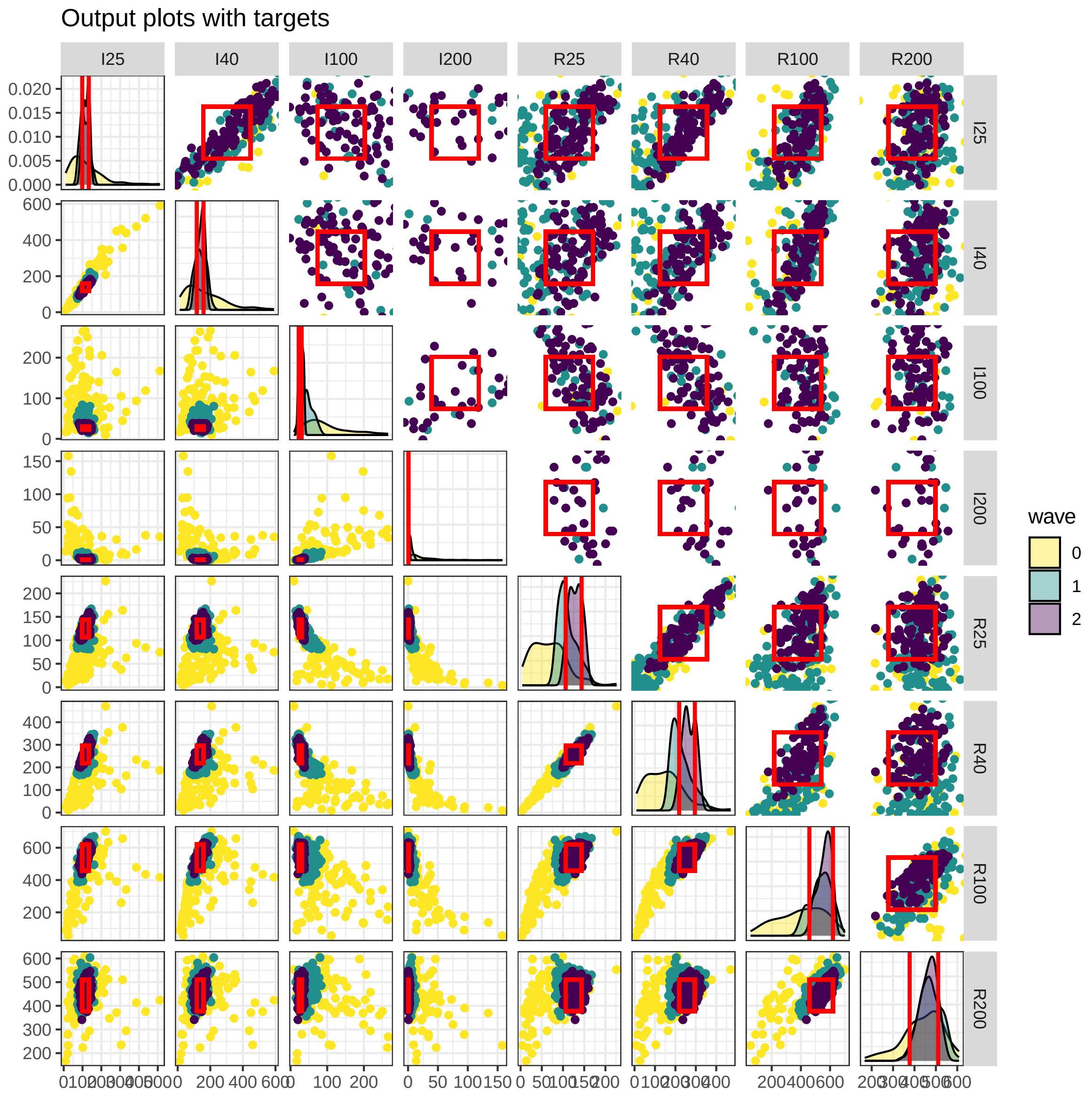
```
simulator_plot(all_aggregated, targets, barcol = "grey")
```



Note that in this call we set `barcol="grey"` in order to have the target intervals in white, instead of black. This plot clearly shows that parameter sets at the end of the first and second wave perform better than the initial set of points.

In the third visualisation, output values for non-implausible parameter sets at each wave are shown for each combination of two outputs:

```
wave_values(all_aggregated, targets, l_wid=1)
```



The argument `l_wid` is optional and helps customise the width of the red lines that create the target boxes. The main diagonal shows the distribution of each output at the end of each wave, with the vertical red lines indicating the lower and upper bounds of the target. Above and below the main diagonal are plots for each pair of targets, with rectangles indicating the target area where full fitting points should lie (the ranges are normalised in the figures above the diagonals). These graphs can provide additional information on output distributions, such as correlations between them. For example, here we see positive correlations between $I25$ and $R40$.

In this workshop we have shown how to perform the first three waves of the history matching process on a stochastic model. Of course, more waves are required, in order to complete the calibration task.

Show: Remind me of possible stopping criteria on P??

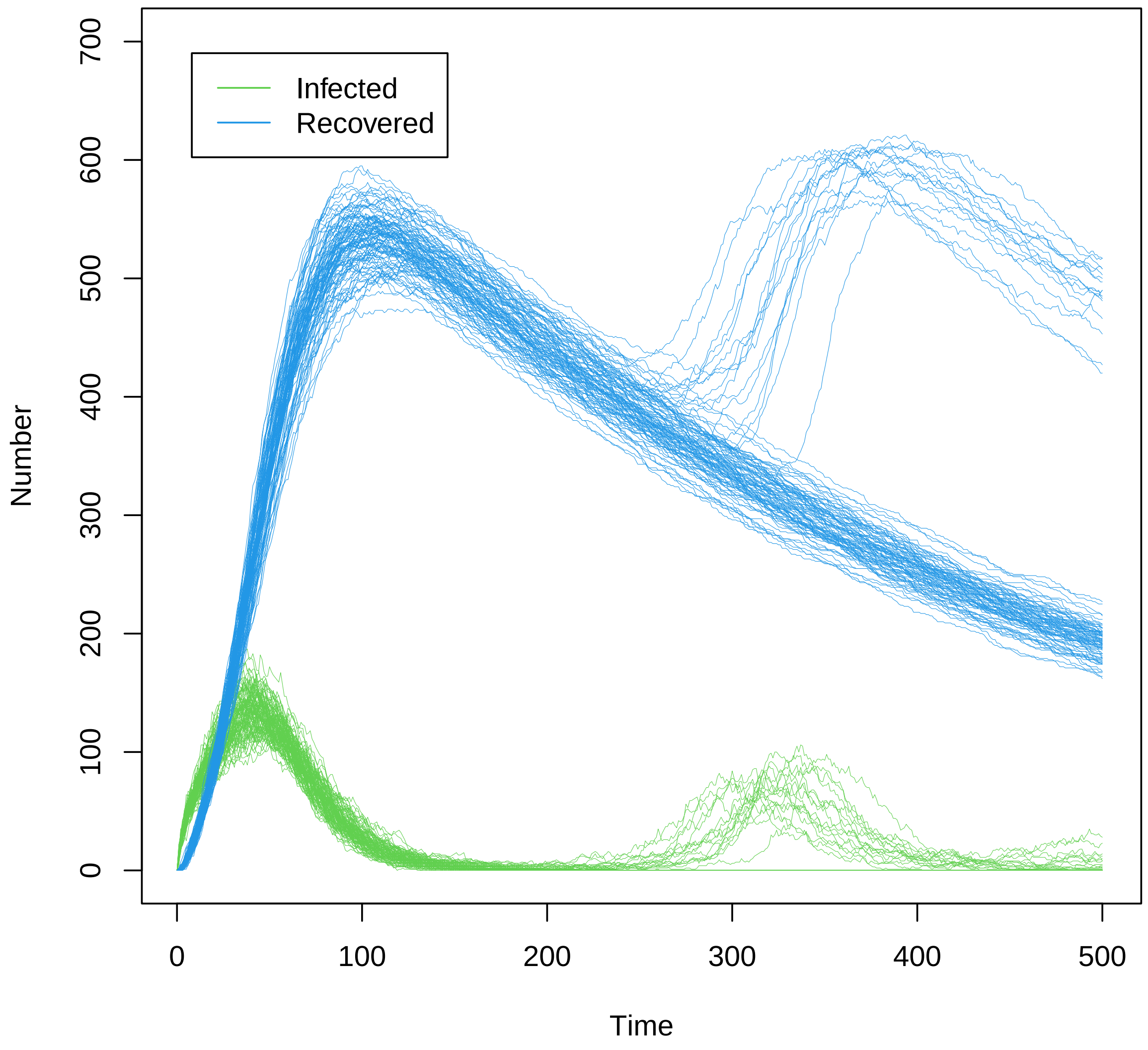
Chapter 11

Dealing with bimodality

A video presentation of this section can be found here.

In the previous sections of this workshop we dealt with stochasticity, but not with bimodality, i.e. when one can perform multiple repetitions at a given parameter set and find two distinct classes of behaviour. This is because our SEIRS model did not show any bimodal behaviour in the outputs for the times considered, i.e. for t up to 200. Let us suppose now that we are interested also in later times. Running the model on `chosen_params` using `get_results` up to time $t = 500$ and plotting the results for “I” and “R” we obtain:

```
solution <- get_results(chosen_params, outs = c("I", "R"),
                        times = c(25, 40, 100, 200, 300, 400, 500), raw = TRUE)
plot(0:500, ylim=c(0,700), ty="n", xlab = "Time", ylab = "Number")
for(j in 3:4) for(i in 1:100) lines(0:500, solution[,j,i], col=(3:4)[j-2], lwd=0.3)
legend('topleft', legend = c('Infected', "Recovered"), lty = 1,
       col = c(3,4), inset = c(0.05, 0.05))
```



The most common example of bimodality, 'take off vs die out', now enters the picture: from time $t = 250$ on, there are a large number of trajectories in green that have zero infected individuals (note the horizontal green line) and a few trajectories where transmission takes off again, producing a second wave of the epidemic (around $t = 300$).

Task 3

Investigate how different parameter sets give rise to different levels of bimodality.

Show: R tip on P??

Show: Solution on P??

In the following subsections we will describe how to deal with bimodality when performing history matching with emulation. Since most parts of the process are very similar to what we have seen already for stochastic models, we will only discuss the areas where the process changes.

> Similar to the philosophy of the previous sections, we will match the observed data to the mean within each of the two modes. This can be used to then exclude a large portion of the input space without resorting to modelling the specific distributional form of the model output. In later workshops we will also learn to match to the set of possible realisations of a stochastic model, by emulating its covariance structure.

11.1 ‘bimodal_wave0’ - design points

While the parameters’ ranges are the same as before, we need to redefine the targets’ list, adding values for times later than $t = 200$:

```
bimodal_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),
  I250 = list(val = 9.67, sigma = 4.76),
  I400 = list(val = 15.67, sigma = 5.36),
  I500 = list(val = 14.45, sigma = 5.32),
  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),
  R250 = list(val = 361.08, sigma = 25.85),
  R400 = list(val = 569.39, sigma = 26.52),
  R500 = list(val = 508.64, sigma = 28.34)
)
```

Note that these are the same exact targets chosen for the deterministic workshop, Workshop 1.

Since we are now interested in times after $t = 200$ too, we need to rerun the parameter sets in `initial_points`, obtaining `bimodal_initial_results`. We then bind all elements in `bimodal_initial_results` to obtain a dataframe `bimodal_wave0`, which we split into a training and a validation set:

```
bimodal_initial_results <- list()
with_progress({
  p <- progressor(nrow(initial_points))
for (i in 1:nrow(initial_points)) {
  model_out <- get_results(unlist(initial_points[i,]), nreps = 50, outs = c("I", "R"),
                           times = c(25, 40, 100, 200, 250, 400, 500))
  bimodal_initial_results[[i]] <- model_out
  p(message = sprintf("Run %g", i))
}
})
bimodal_wave0 <- data.frame(do.call('rbind', bimodal_initial_results))
bimodal_all_training <- bimodal_wave0[1:5000,]
bimodal_all_valid <- bimodal_wave0[5001:7500,]
bimodal_output_names <- c("I25", "I40", "I100", "I200", "I250", "I400", "I500",
                          "R25", "R40", "R100", "R200", "R250", "R400", "R500")
```

11.2 Bimodal Emulators

To train bimodal emulators we use the function `emulator_from_data` specifying `emulator_type = 'multistate'`, which requires the training data, the names of the outputs to emulate, and the ranges of the parameters:

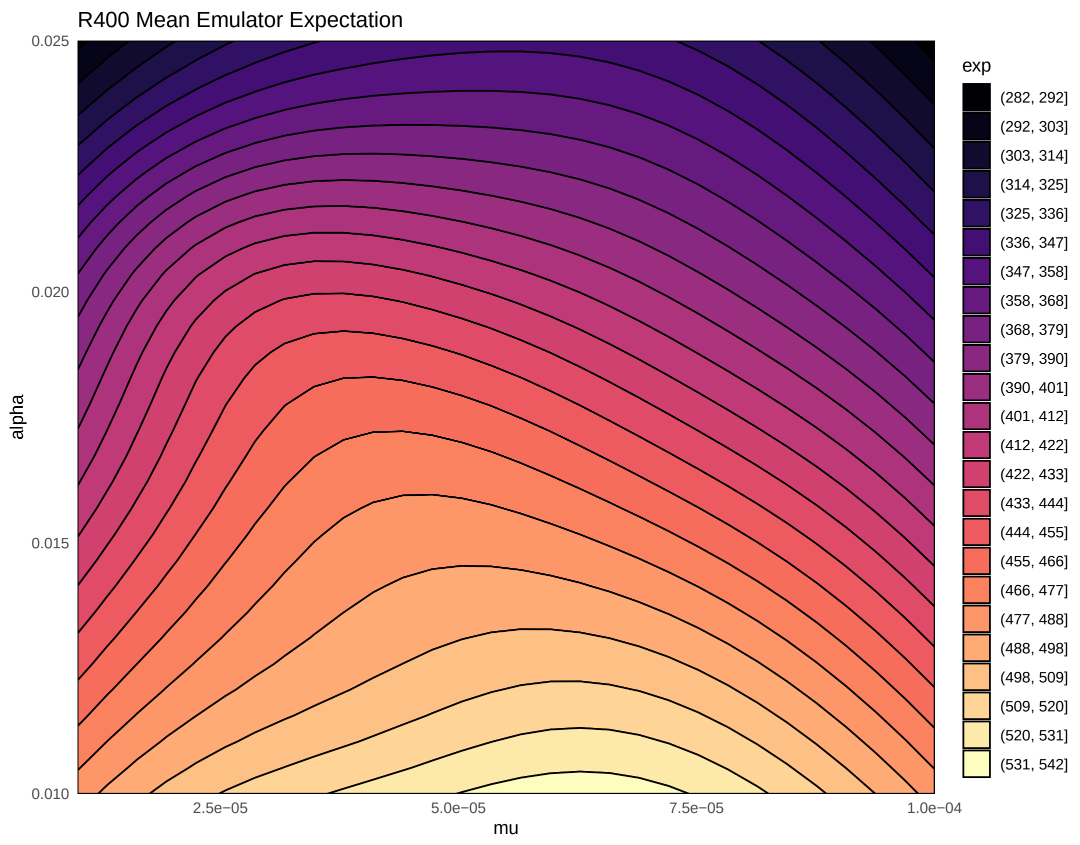
```
bimodal_emulators <- emulator_from_data(bimodal_all_training,
                                        bimodal_output_names, ranges,
                                        emulator_type = 'multistate')
```

Behind the scenes, this function does the following:

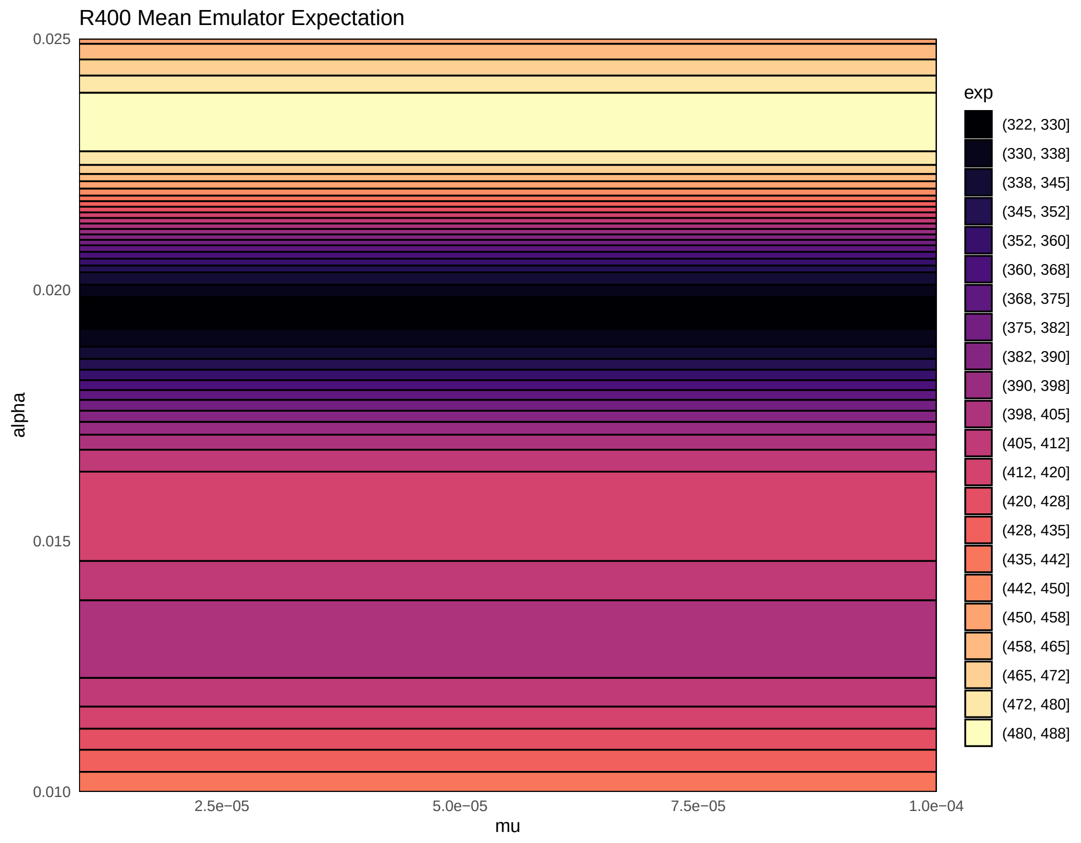
- First it looks at the provided training data and identifies which of the outputs are bimodal (see box below if interested in the method used to identify bimodal outputs).
- For the outputs where bimodality is found, the repetitions at each parameter set are clustered into two subsets, based on the mode they belong to.
- For outputs without bimodality, stochastic emulators are trained as before. For outputs with bimodality, variance and mean emulators are trained separately for each of the two modes. To access the mean emulators for the first mode, we type `bimodal_emulators$mode1$expectation`, and to access the variance emulators for the second mode we type `bimodal_emulators$mode2$variance`.
- Finally, an emulator for the proportion of points in each mode is also trained (this is a single emulator, as in the deterministic case). This emulator can be accessed by typing `bimodal_emulators$prop` and will be referred to as proportion emulator.

Let us now plot the expectation of the mean emulator of $R400$ for each mode fixing all non-shown parameters to their values in `chosen_params`:

```
emulator_plot(bimodal_emulators$mode1$expectation$R400, params = c('mu', 'alpha'),
              fixed_vals = chosen_params[!names(chosen_params) %in% c('mu', 'alpha')])
```



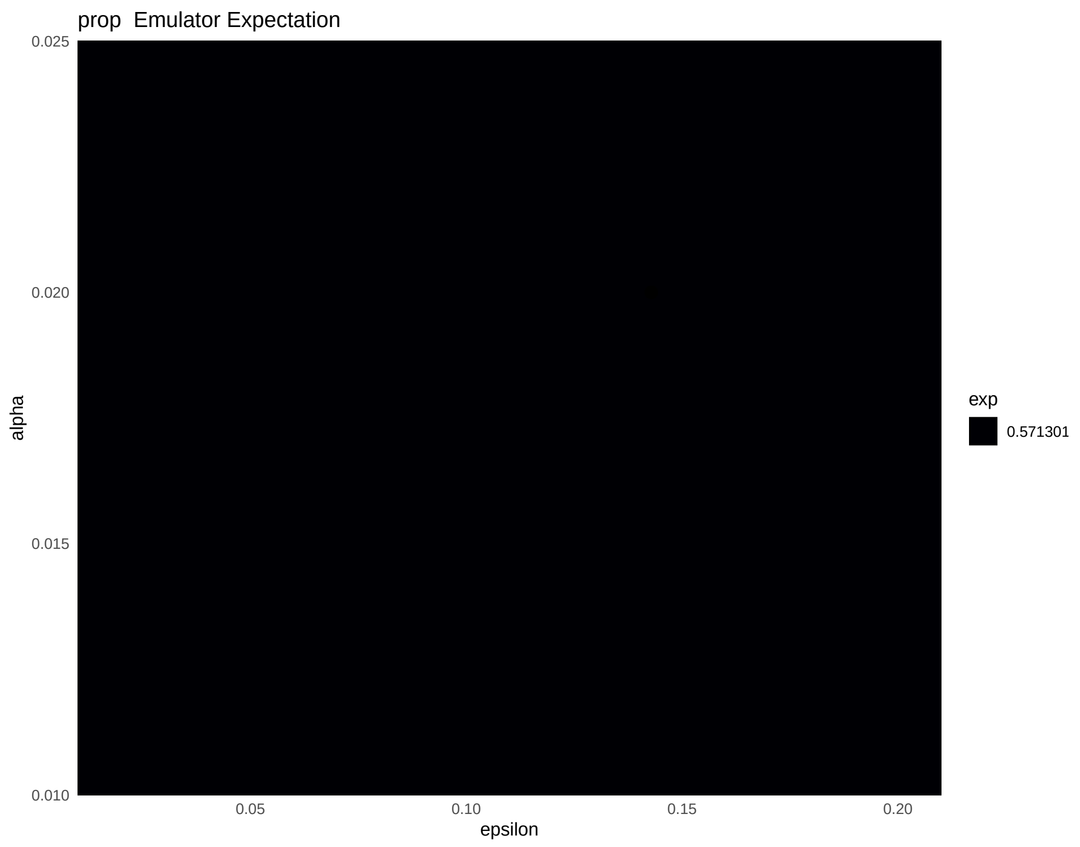
```
emulator_plot(bimodal_emulators$mode2$expectation$R400, params = c('mu', 'alpha'),
                fixed_vals = chosen_params[!names(chosen_params) %in% c('mu', 'alpha')])
```



We notice that mode 2 tends to contain higher values than mode 1. This indicates that mode 1 is the one where the infection dies out.

It is also instructive to plot the expectation of the proportion emulator. We use the argument `fixed_vals` to set the parameters that are not shown in the plot to be as in `chosen_params`:

```
emulator_plot(bimodal_emulators$prop, params = c('alpha', 'epsilon'),
              fixed_vals = chosen_params[!names(chosen_params) %in% c('alpha' ,'epsilon')]) +
              geom_point(aes(x=1/7, y=1/50), size=3)
```



The plot shows that the proportion of points in mode 1, where the infection dies out, is around 0.58. This is in accordance with the plot produced at the beginning of this section, where the infection died out on the majority of trajectories.

Show: How are bimodal outputs identified? on P**??**

Task 4

Confirm that mode 1 is where the infection dies out by comparing the plots of the expectation of the mean emulator for $I400$ for both modes.

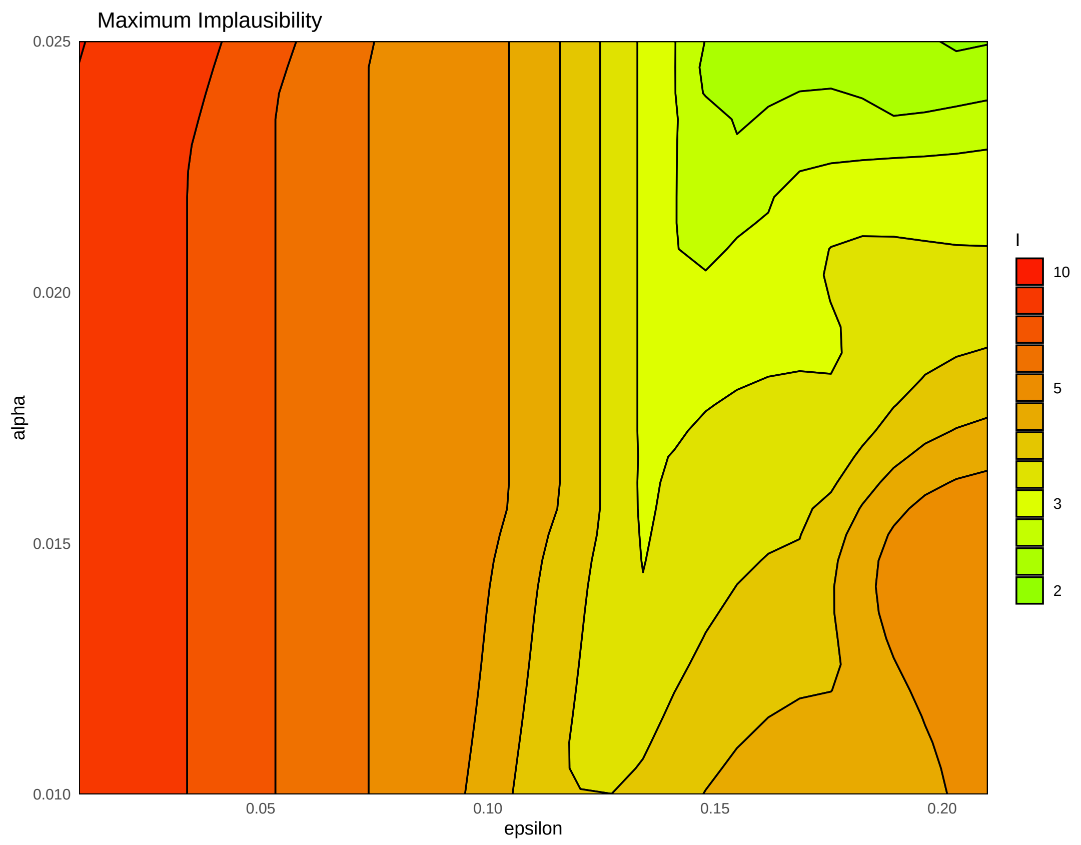
Show: Solution on P**??**

11.3 Implausibility

Since we are now in a bimodal setting, we want to regard a point as non-implausible for a given target if it is valid with respect to either mode. This is the default behaviour of the package, when dealing with bimodal emulators.

For a given output and a given point, the implausibility for mode 1 and the implausibility for mode 2 are calculated, and the minimum of them is taken. The maximum (or second-maximum, third-maximum etc) of this collection of minimised implausibilities is then selected, depending on the user choice. For example, to plot the maximum of these minimised implausibilities, we set `plot_type` to `nimp`:

```
emulator_plot(bimodal_emulators, plot_type = 'nimp', targets = bimodal_targets,
              params = c('alpha', 'epsilon'))
```



Task 5

Set the argument `plot_type` to `imp` to produce implausibility plots for each output, for mode 1 and mode 2. Which implausibility plots are the same for each mode, and which are different? Why?

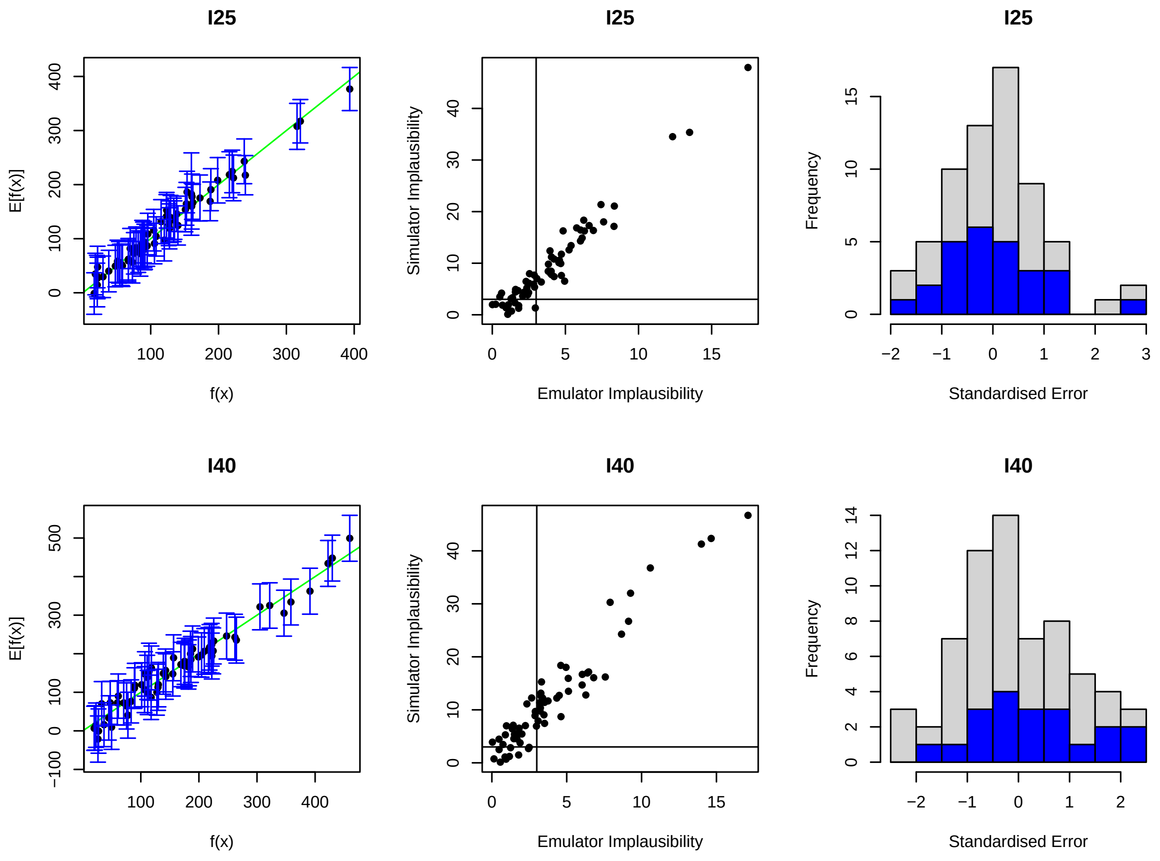
Show: Solution on P??

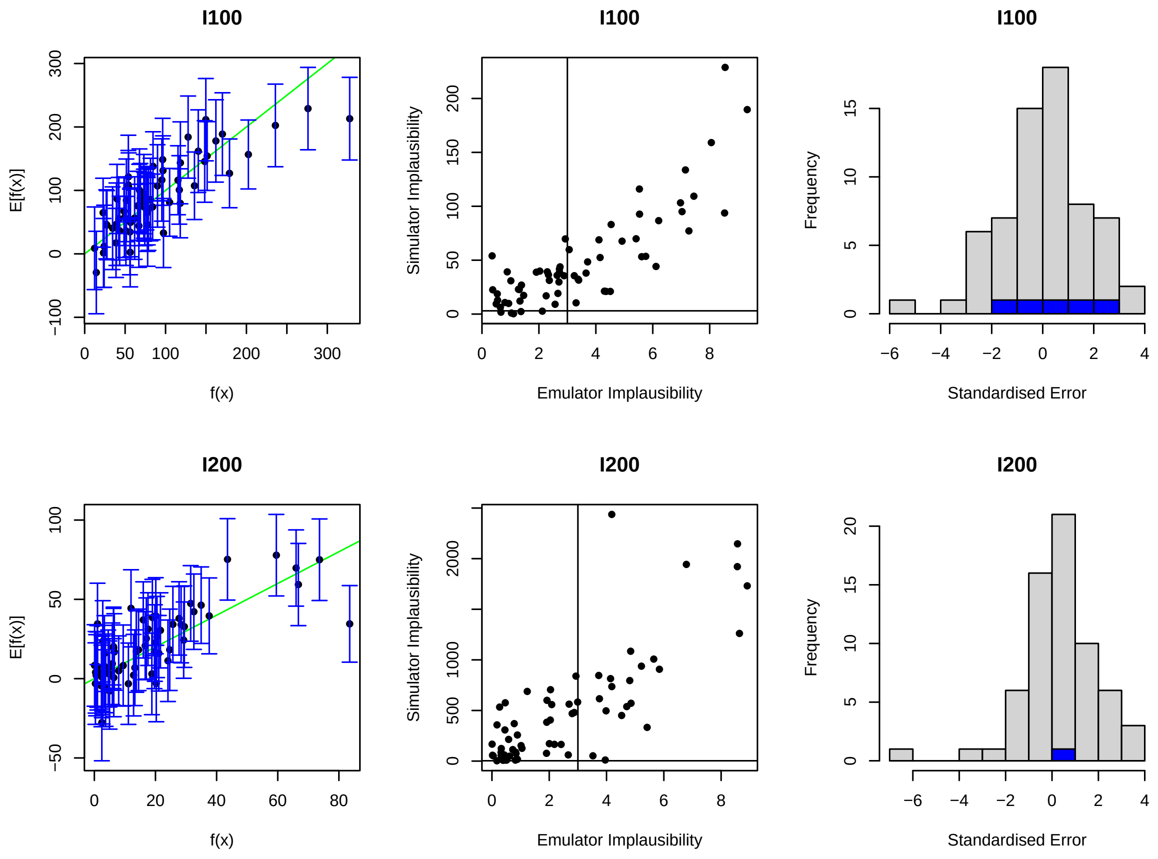
11.4 Emulator diagnostics

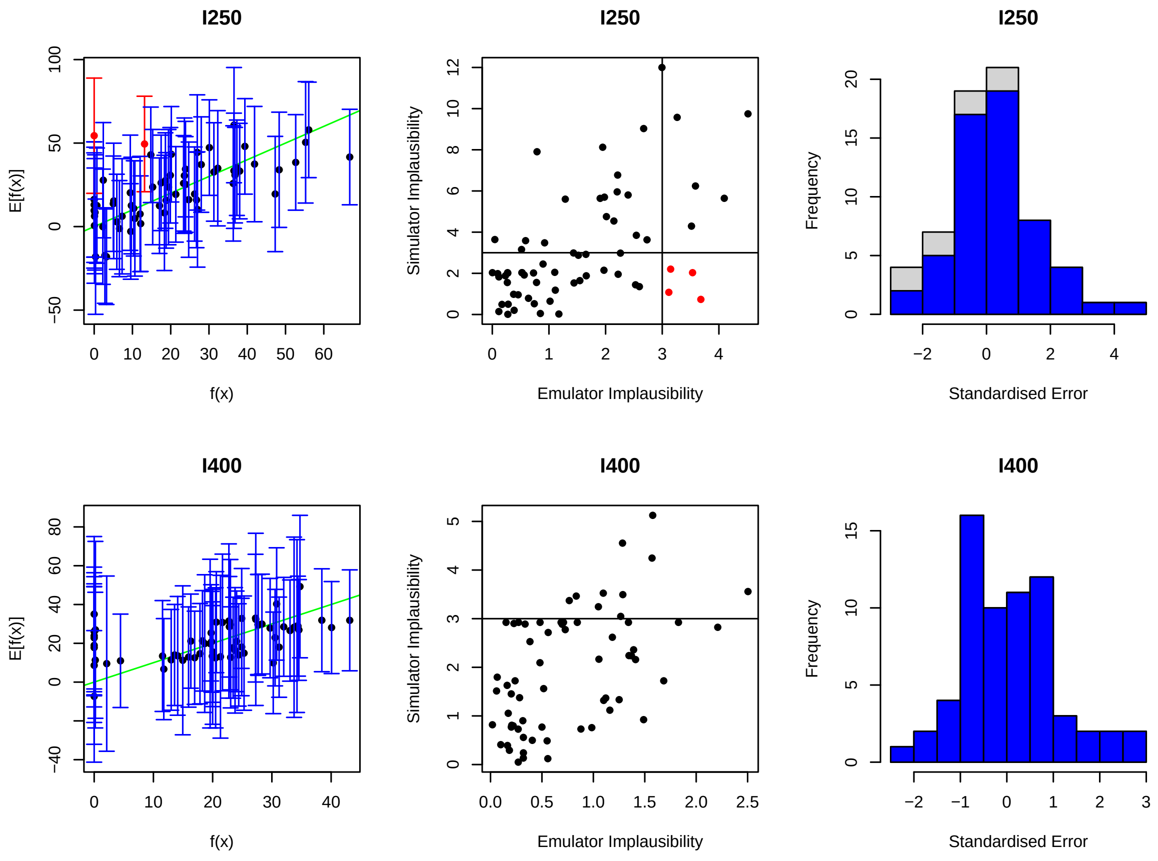
As before, the function `validation_diagnostics` can be used to get three diagnostics for each emulated output. For example, to produce diagnostics for the first four emulators we type:

```
vd <- validation_diagnostics(bimodal_emulators, bimodal_targets, bimodal_all_valid,
                             plt=TRUE, row=2)
```

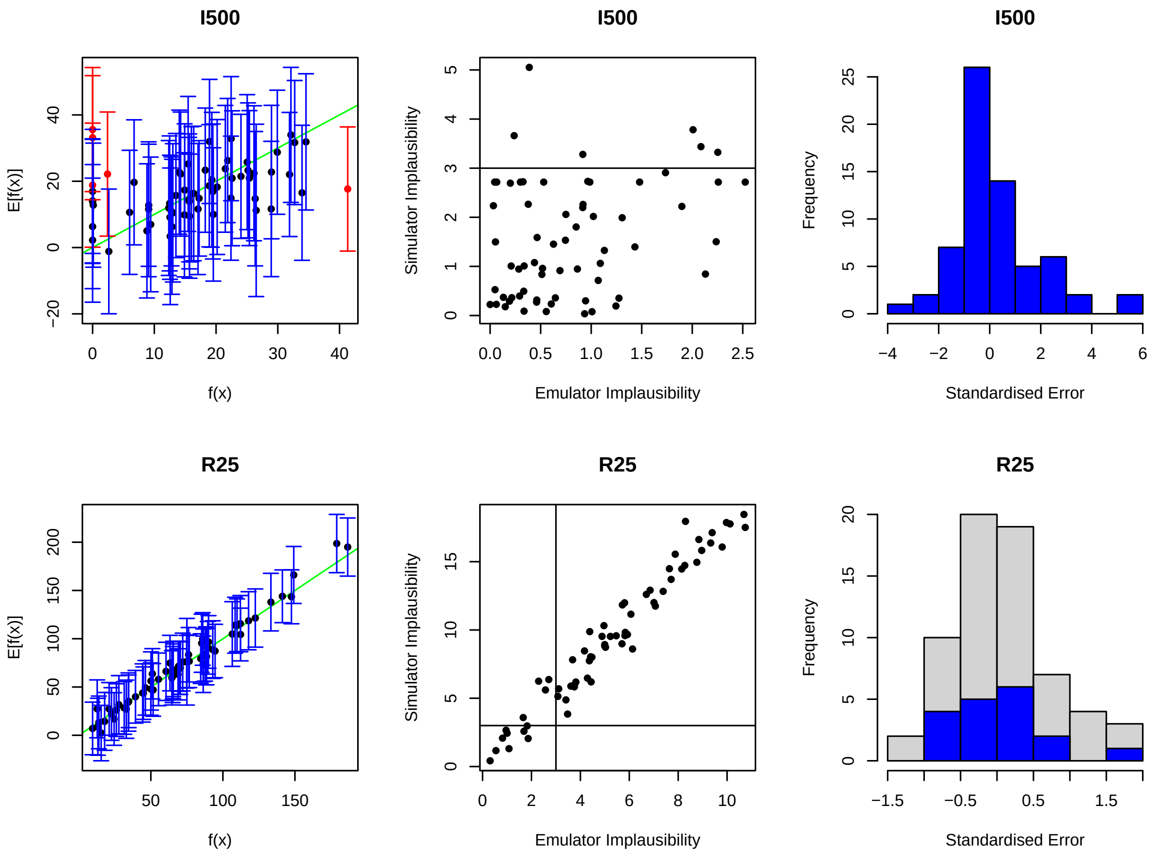
```
## Warning in fanny(suppressWarnings(daisy(v_outputs)), k = 2): the memberships
## are all very close to 1/k. Maybe decrease 'memb.exp' ?
```

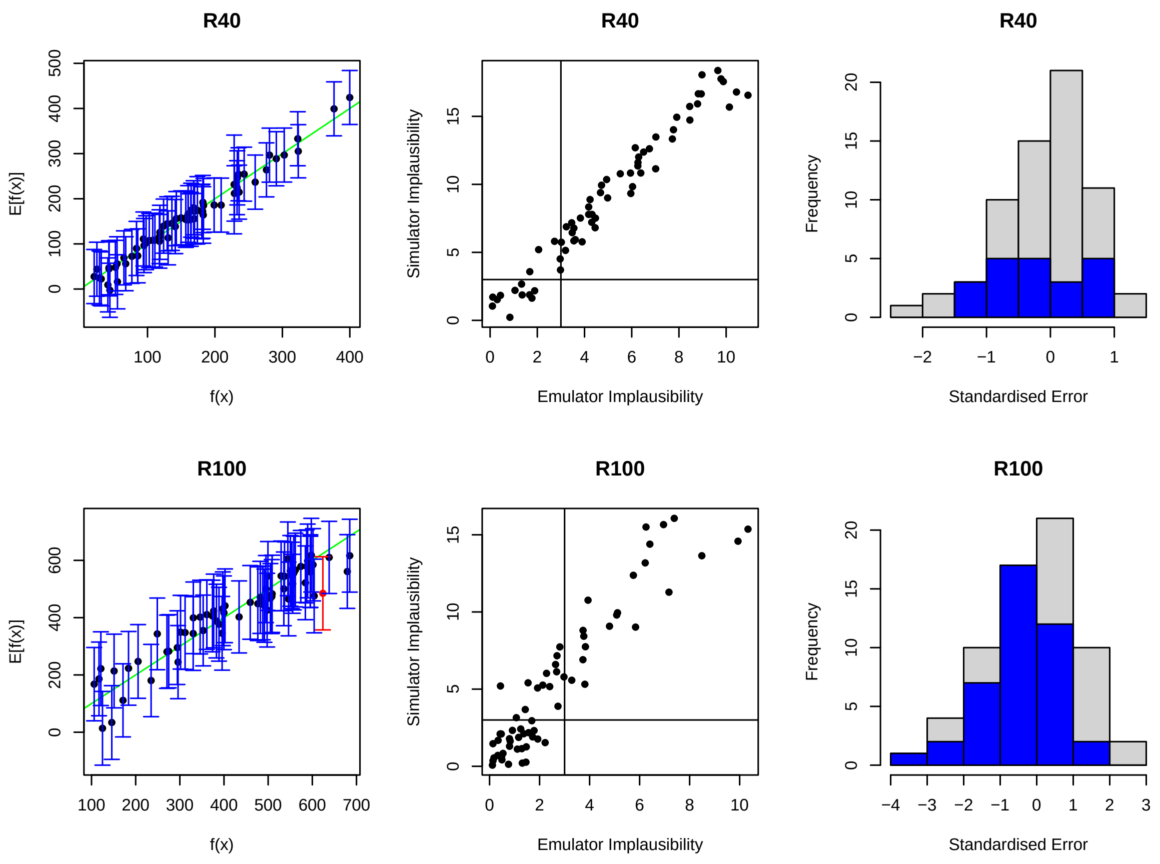

I25
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error
I40
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error

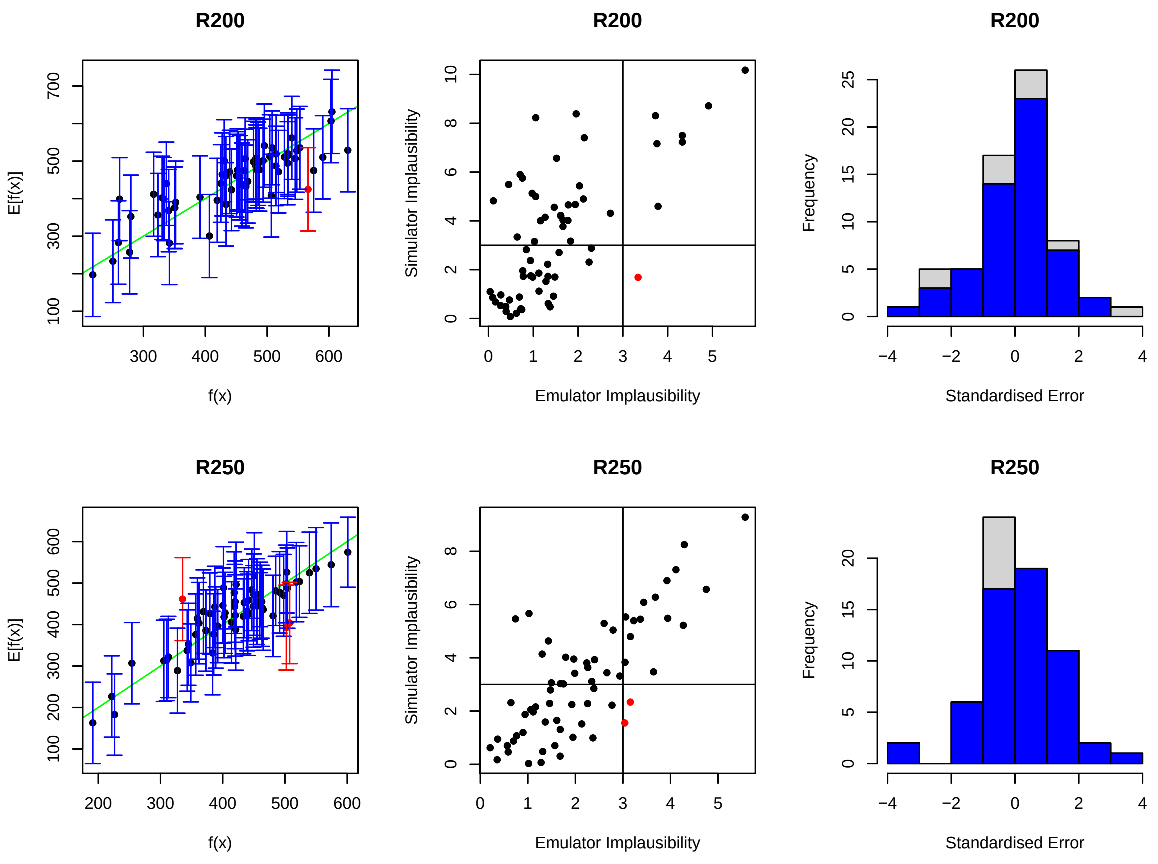

I100
E[f(x)]
f(x)
I100
Simulator Implausibility
Emulator Implausibility
I100
Frequency
Standardised Error
I200
E[f(x)]
f(x)
I200
Simulator Implausibility
Emulator Implausibility
I200
Frequency
Standardised Error

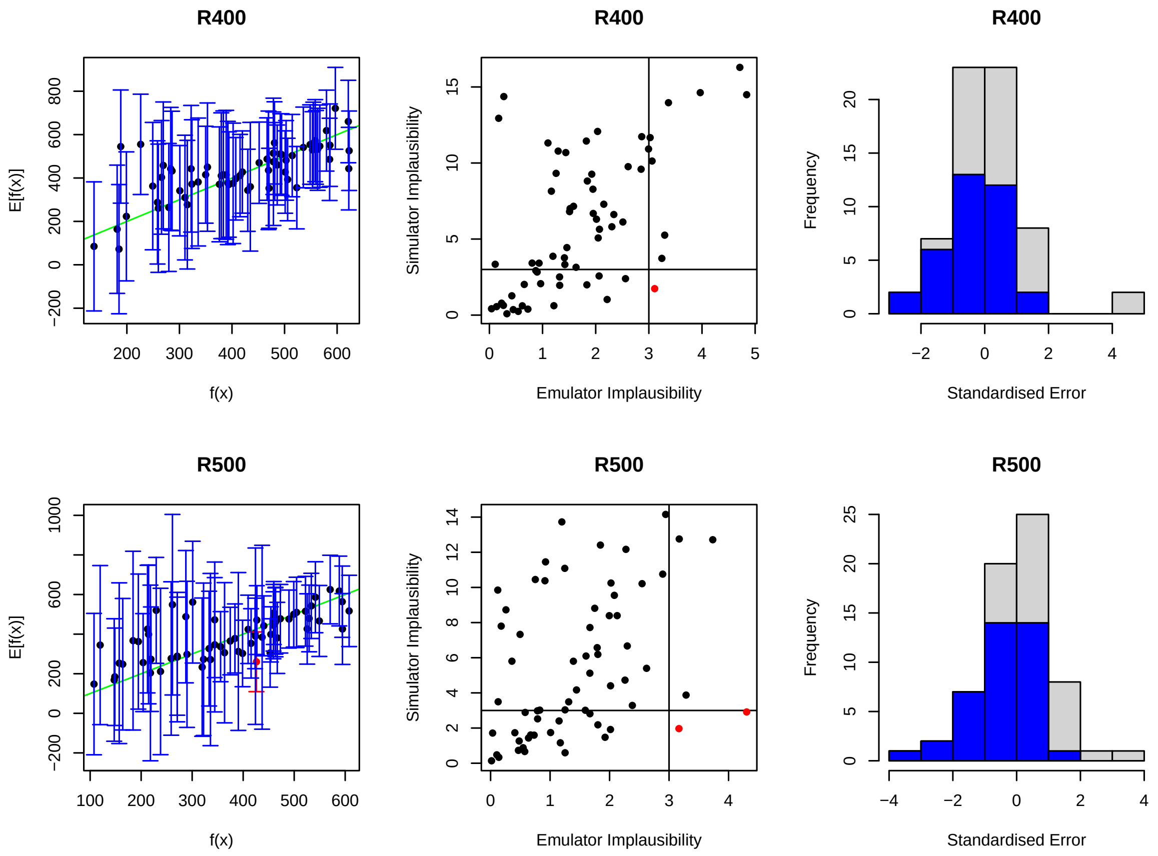


I250
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error
I400


I500
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error
R25
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error


R40
E[f(x)]
f(x)
R40
Simulator Implausibility
Emulator Implausibility
R40
Frequency
Standardised Error
R100
E[f(x)]
f(x)
R100
Simulator Implausibility
Emulator Implausibility
R100
Frequency
Standardised Error


R200
E[f(x)]
f(x)
Simulator Implausibility
Emulator Implausibility
Frequency
Standardised Error
R250



You may have noticed that in the first column we have more points plotted than we have validation points. This is because we do diagnostics for each mode for each output, i.e. we compare the predictions of the mean emulators for mode 1 and mode 2 with the model output values, which are also clustered in two subsets, due to bimodality.

As in the deterministic case, one can enlarge the σ values to obtain more conservative emulators, if needed. For example, to double the σ value for the mean emulators for $I500$ in both modes we would type:

```
bimodal_emulators$mode1$expectation$I500 <- bimodal_emulators$mode1$expectation$I500$mult_sigma(
bimodal_emulators$mode2$expectation$I500 <- bimodal_emulators$mode2$expectation$I500$mult_sigma(
```

11.5 Proposing new points

Generating a set of non-implausible points, based on the trained emulators, is done in exactly the same way as it was in Section 8, using the function `generate_new_design`. In this case we need to use the function `subset_emulators` to remove the emulators for the outputs at $t = 450$ (the seventh and fourteenth emulators), for which we did not set targets:

```
new_points <- generate_new_design(bimodal_emulators, 150, bimodal_targets, nth=1)
```

Similarly, performing the second wave is done in the same as it was in Section 9, replacing `emulator_type = 'variance'` with `emulator_type = 'multistate'` when calling `emulator_from_data`.

Appendix A

Answers

Placeholder