Workshop 2: calibrating a stochastic model

Danny Scarponi, Andy Iskauskas

Contents

Chapter 1

Objectives

In this workshop, you will learn to perform history matching with emulation on models with stochasticity, using the hmer package. This workshop should be addressed after working on Workshop 1, which shows how to perform history matching with emulation on deterministic models.

Show: Code to load relevant libraries and helper functions on P54

Chapter 2

An overview of history matching with emulation and hmer

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.

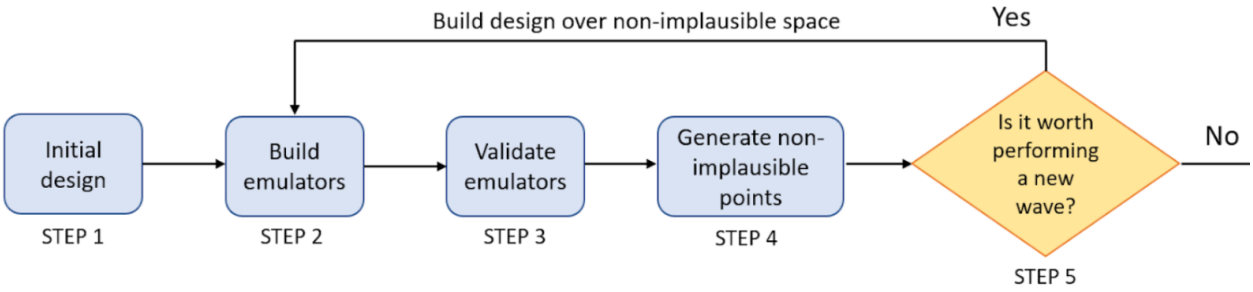


Figure 2.1: History matching with emulation workflow

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 this workshop
	Name	Input	Output	
Build stochastic emulators (step 2)	*variance_emulator_from_data*	• 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 diagnos
Generate non-implausible points (step 4)	*generate_new_runs*	• 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

Figure 2.2: The three main hmer functions

Chapter 3

Introduction to the model

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.

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 $\beta_3 = 0.4$:

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

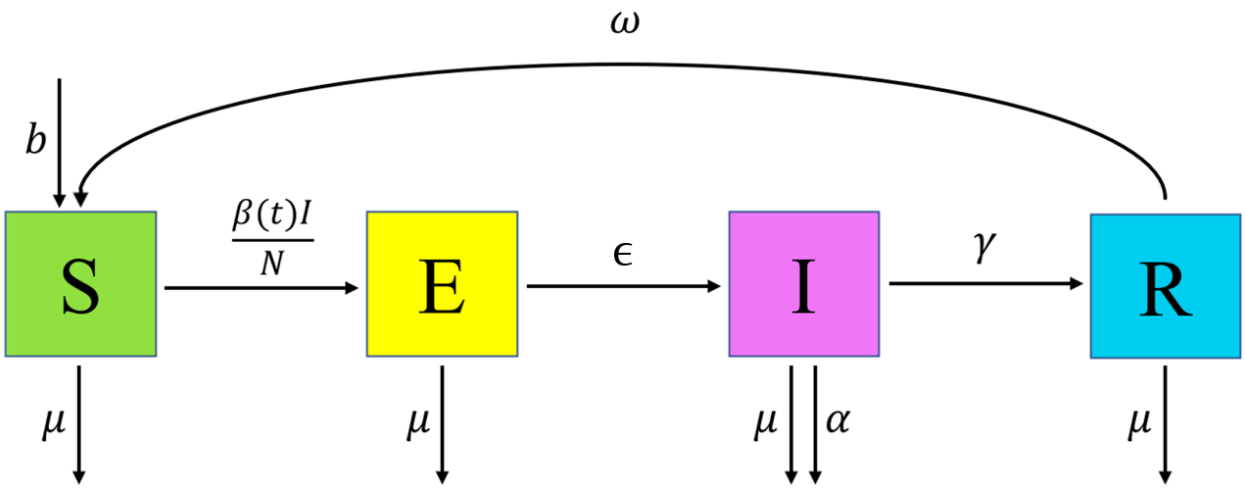


Figure 3.1: SEIRS Diagram

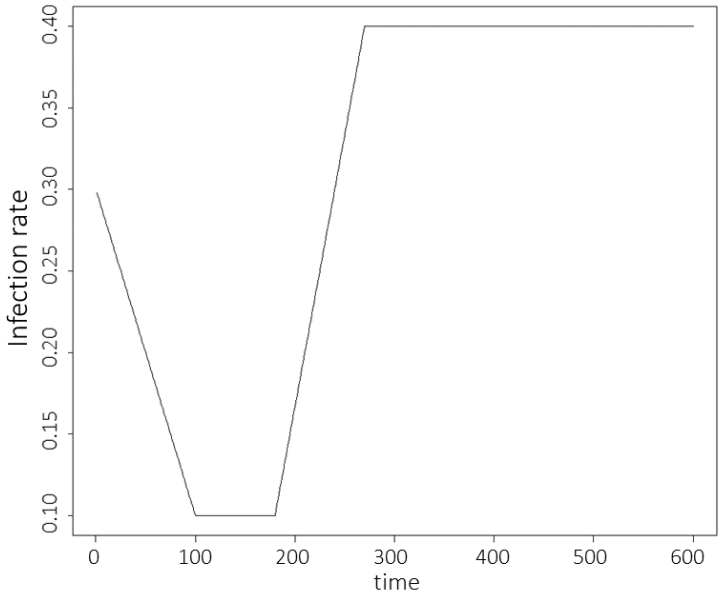


Figure 3.2: Graph of the rate of infection between each infectious and susceptible person

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

```
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.

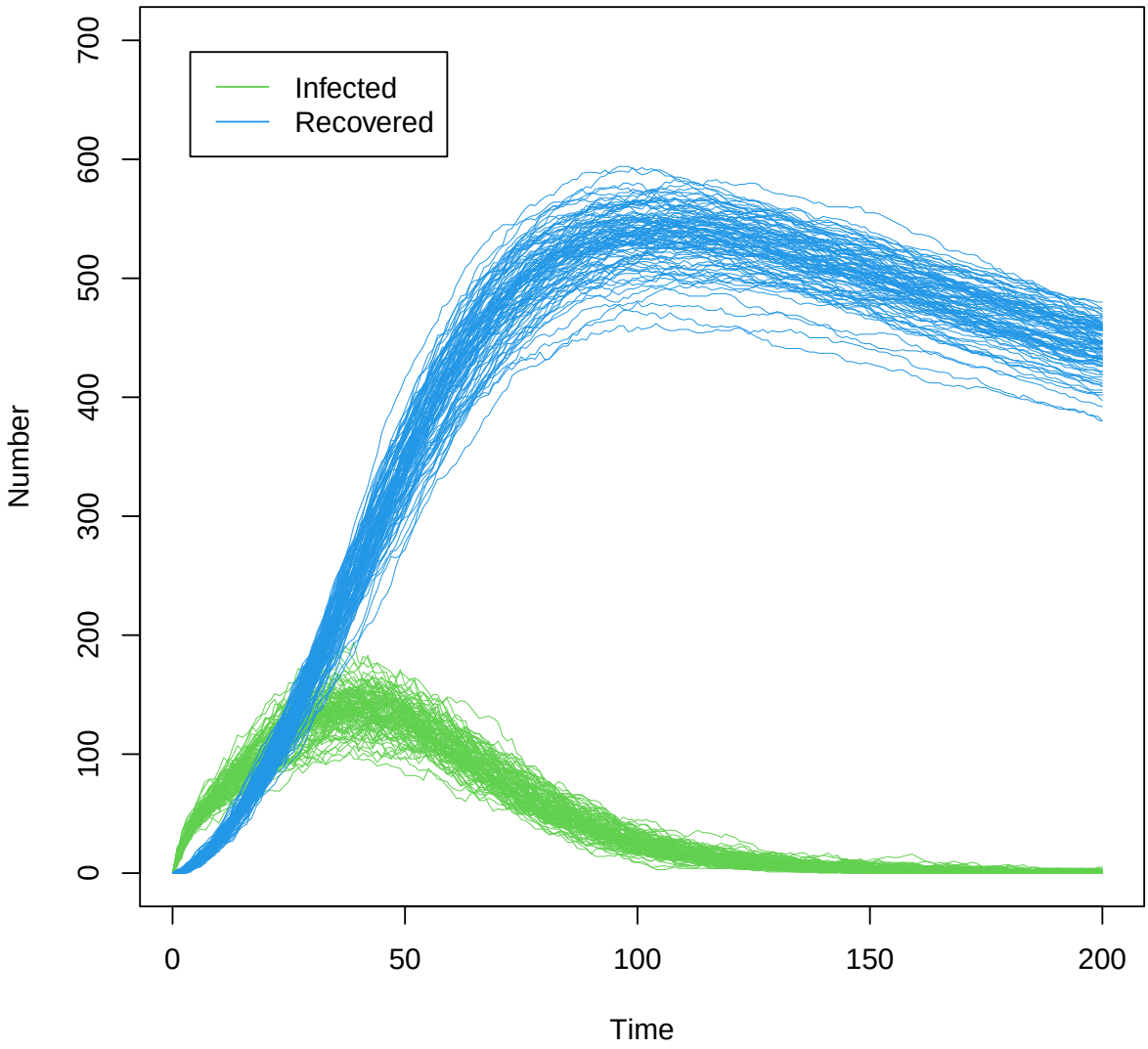
Using `get_results` on `chosen_params`

```
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

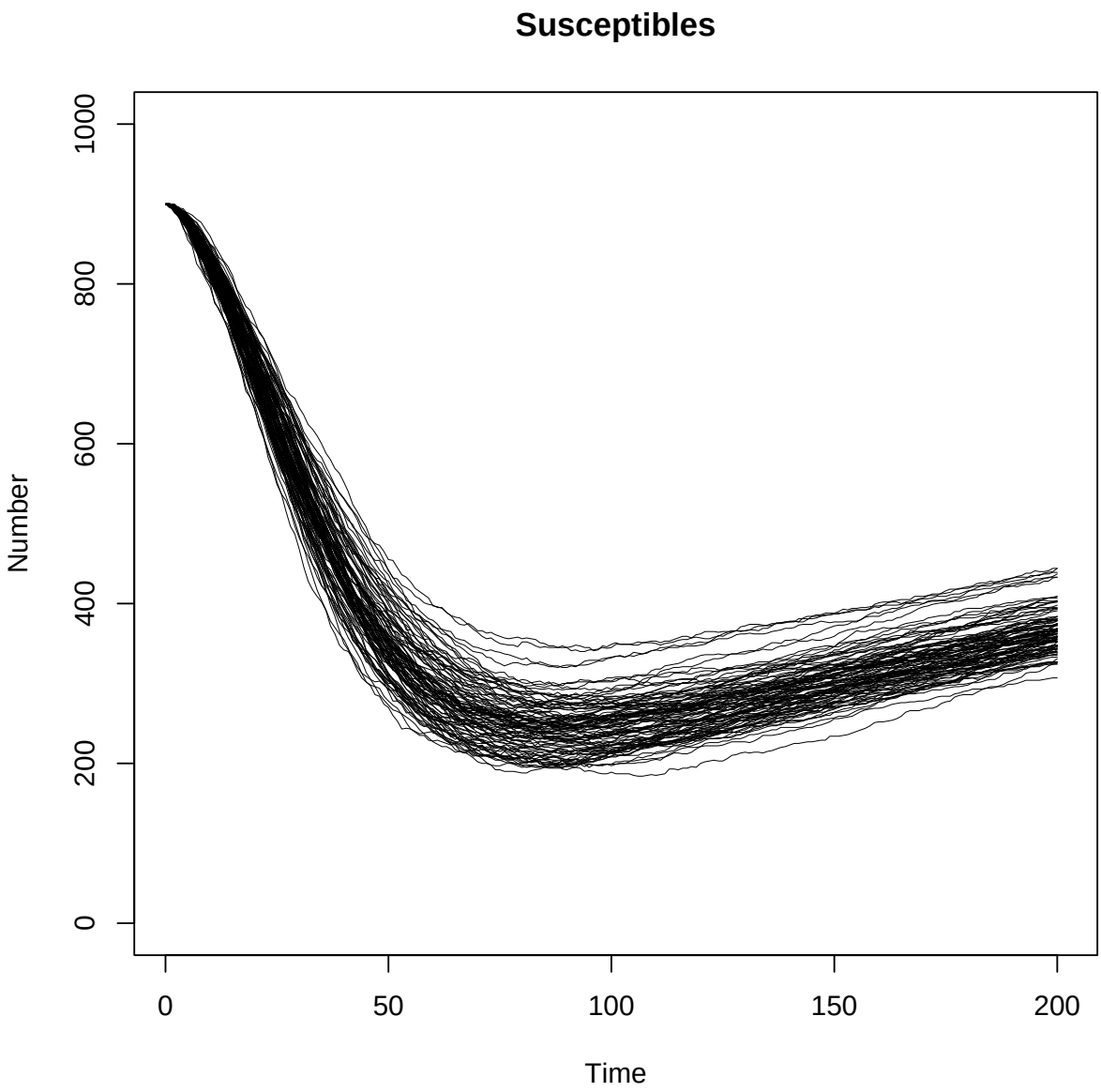
```
plot(0:200, ylim=c(0,700), ty="n", xlab = "Time", ylab = "Number")
for(j in 3:4) for(i in 1:100) lines(0:200, solution[,j,i], col=(3:4)[j-2], lwd=0.3)
legend('topleft', legend = c('Infected', "Recovered"), lty = 1,
       col = c(3,4), inset = c(0.05, 0.05))
```



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:

```
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 P55

Show: Solution on P45

Chapter 4

‘waveO’ - parameter ranges, targets and design points

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:

```
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
  epsilon = 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 $t = 25, 40, 100, 200$. 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. Also note that we can also define targets using lower and upper bounds, instead of value and standard deviation. For example, if we want to add a target `I60` with lower bound 100 and upper bound 120, then we would add `I60 = c(100,120)` in the list above.

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 150 parameter sets: we will use 100 of them for the training set and 50 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.

```
initial_LHS <- maximinLHS(150, 9)
#validation_lhs <- lhs::randomLHS(50, 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` and `validation_points`, which contain 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 $t = 25, 40, 100, 200$.

```
initial_results <- list()
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))
  initial_results[[i]] <- model_out
}
```

Note that `initial_results` is a list of length 7500, since it has a row for each of the 50 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 5000 elements (corresponding to the first 100 parameter sets) for the training set, `all_training`, and the last 2500 for the validation set (corresponding to the last 50 parameter sets), `all_valid`.

```
wave0 <- data.frame(do.call('rbind', initial_results))
all_training <- wave0[1:5000,]
all_valid <- wave0[5001:7500,]
output_names <- c("I25", "I40", "I100", "I200", "R25", "R40", "R100", "R200")
```

Chapter 5

Emulators

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

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

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. 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 `variance_emulator_from_data`, which requires the training data, the names of the outputs to emulate, and the ranges of the parameters:

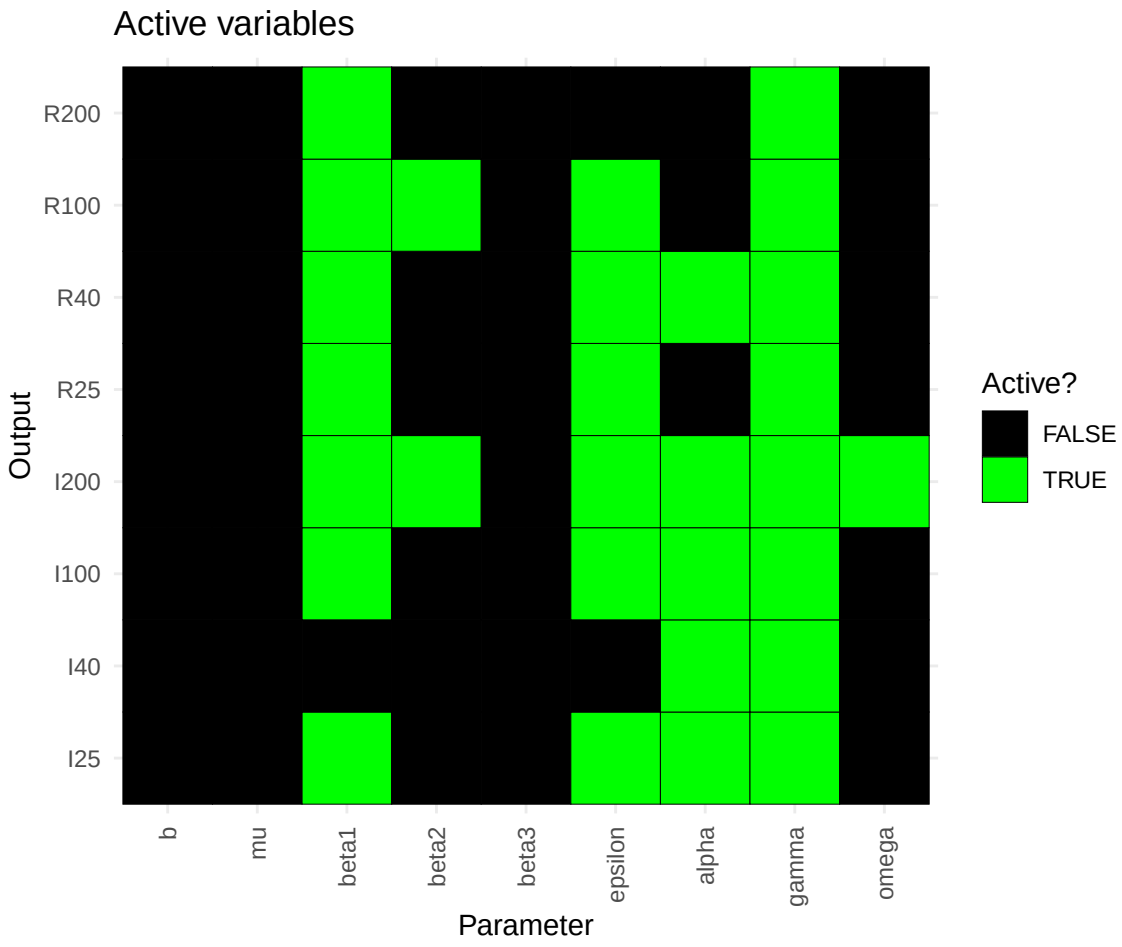
```
stoch_emulators <- variance_emulator_from_data(all_training, output_names, ranges)
```

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

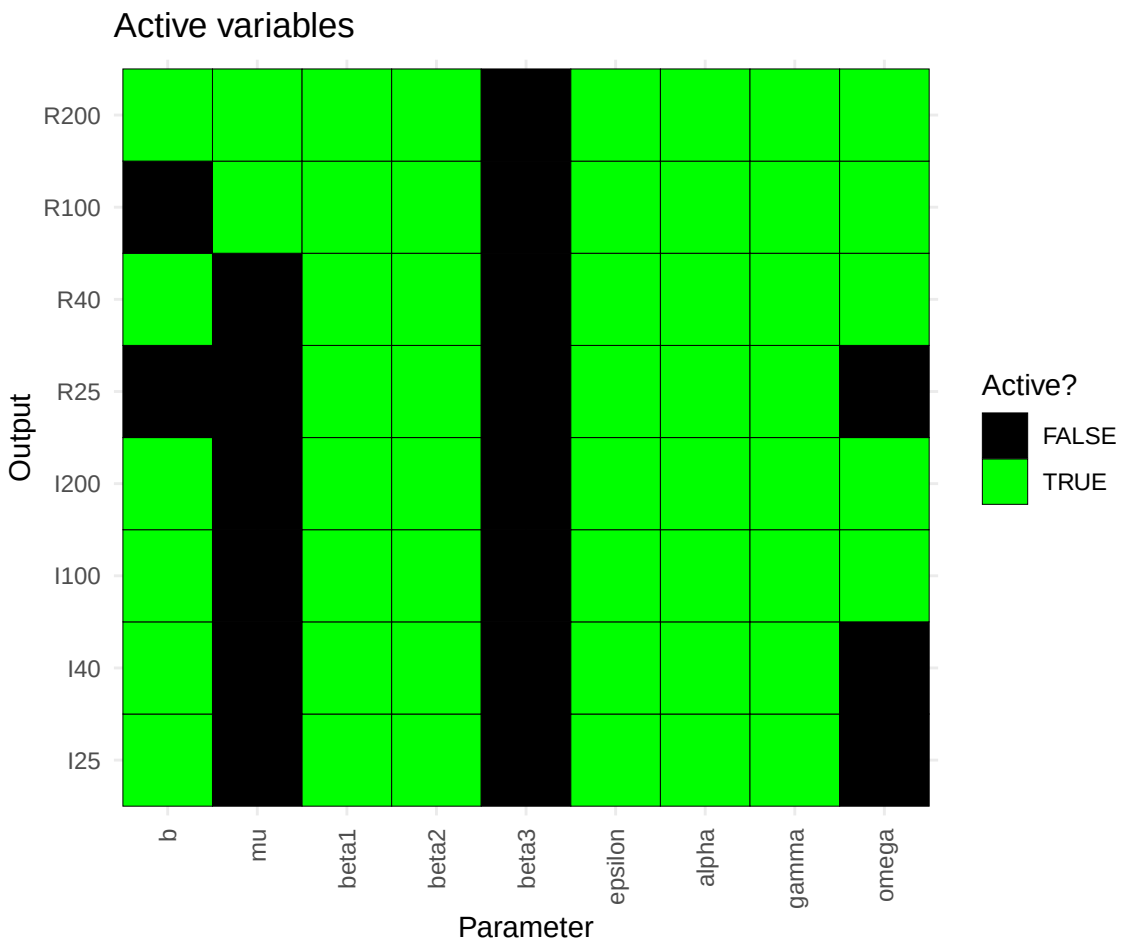
- First, it estimates the variance of each model output of interest at each provided parameter set: this is possible since `all_training` contains several model runs at each parameter set.
- Using the obtained variance data, it then trains an emulator for the variance of each of the outputs of interest. 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 of σ: for each parameter set x, the variance emulator gives us an estimate of the variance of the model output at x. This means that the prior for σ varies from parameter set to parameter set, and is therefore estimated more adequately across the input space, compared to the deterministic case, where σ was constant.

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

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



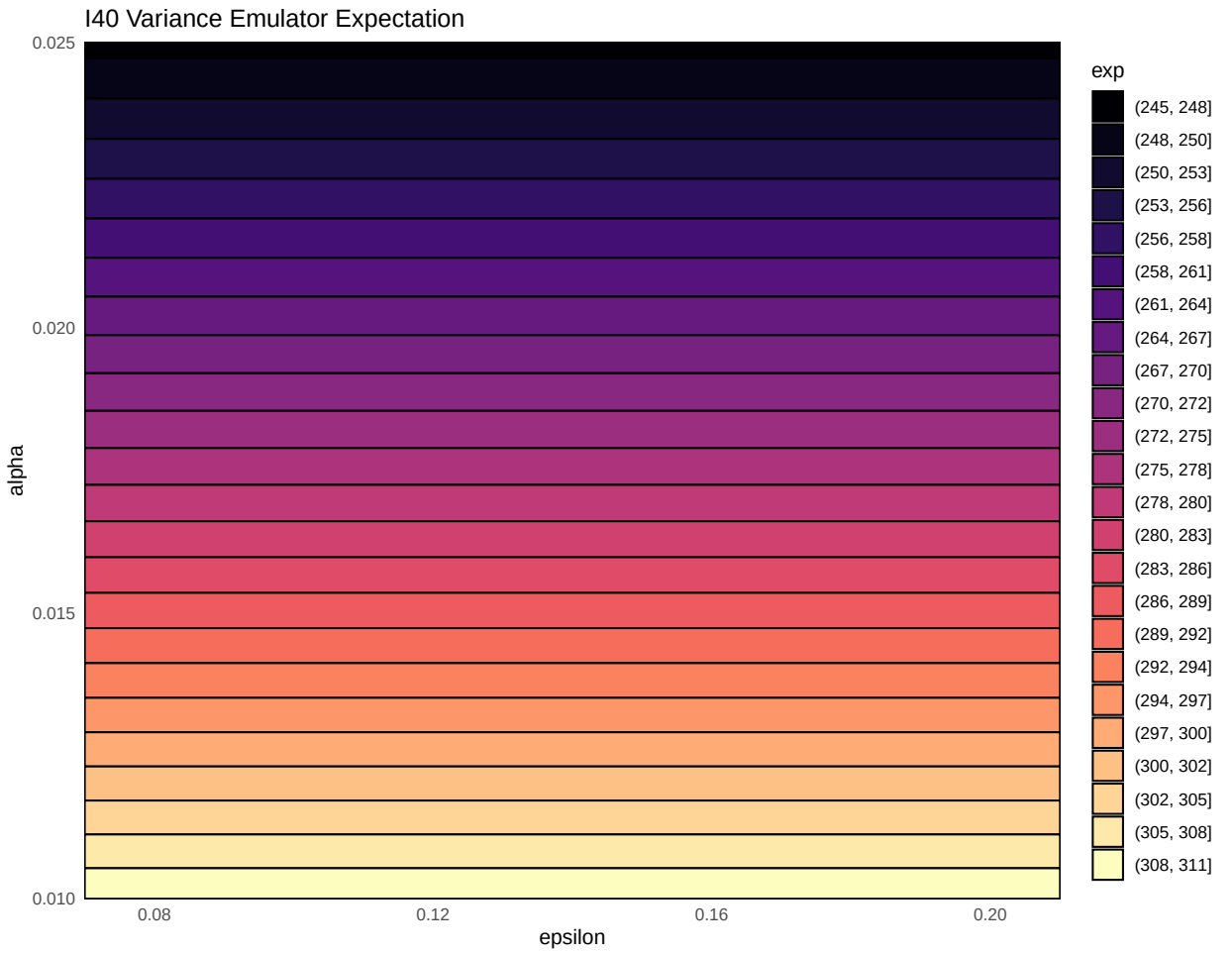

```
plot_actives(stoch_emulators$expectation)
```



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, β_2, ϵ, α are active for most or all outputs in the mean emulators.

As in 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. If we want to plot the expectation of the variance emulator for $I40$, we just need the following command:

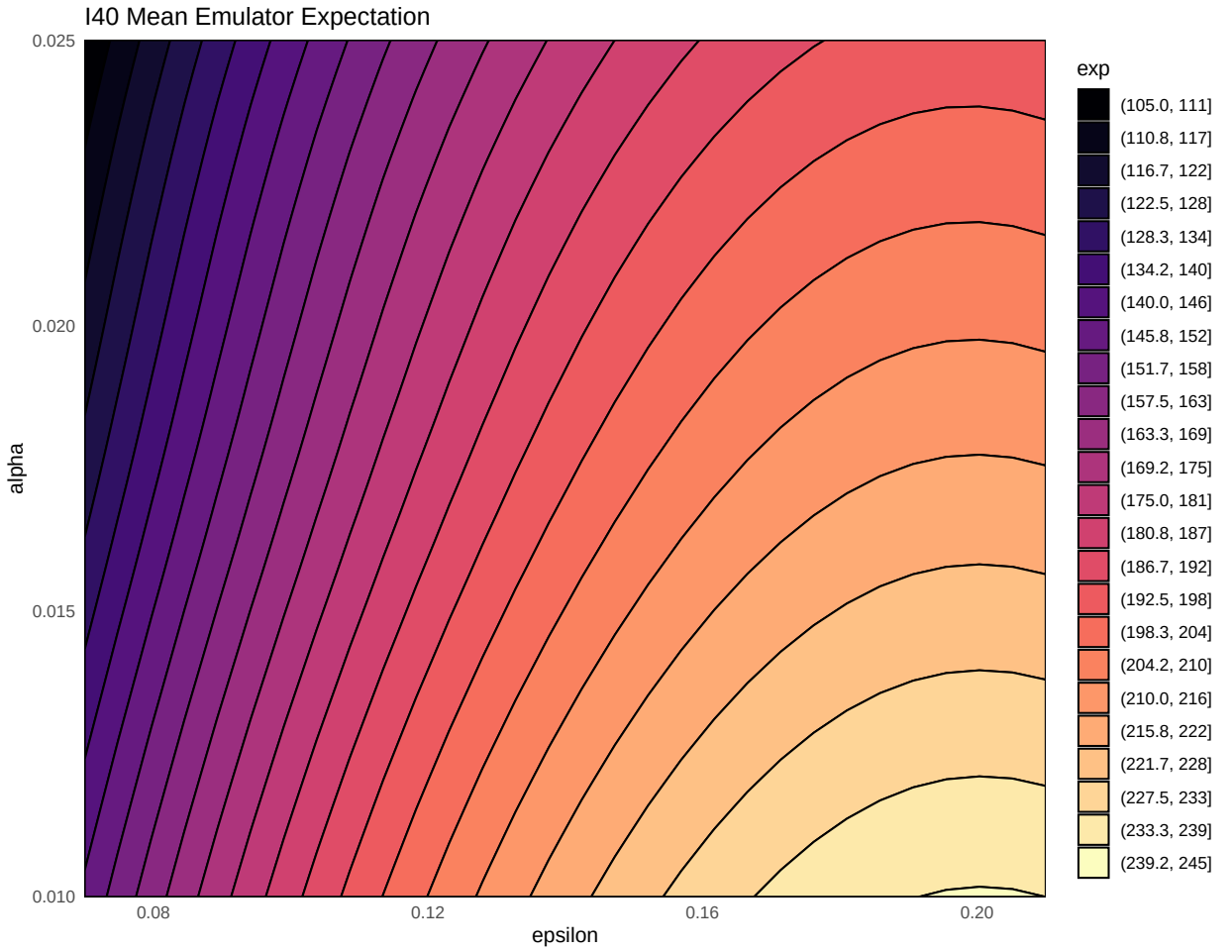
```
emulator_plot(stoch_emulators$variance$I40, params = c('epsilon', 'alpha'))
```



where we chose to show the two parameters ϵ and α.

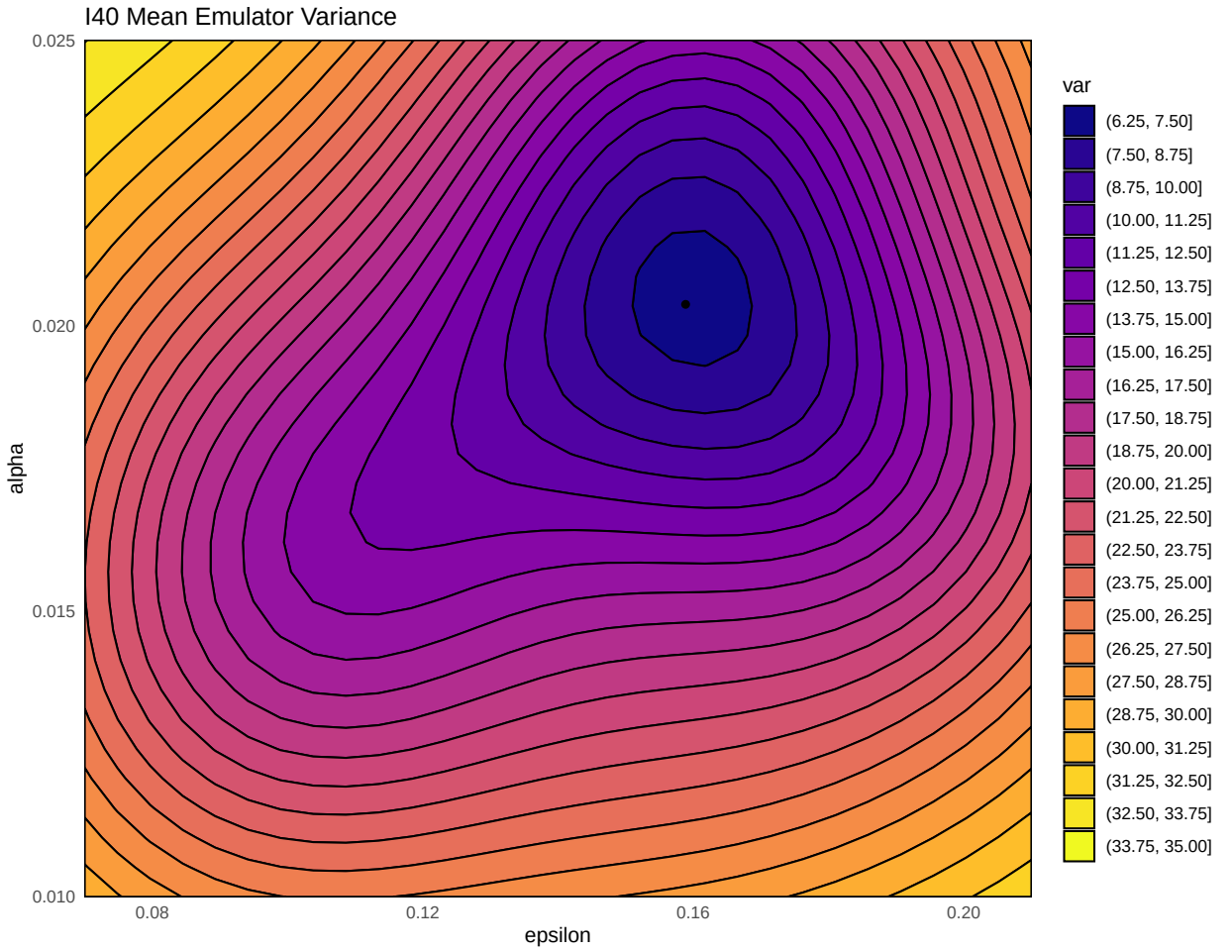
To produce a similar plot for the expectation of the mean emulator for $I40$, we type:

```
emulator_plot(stoch_emulators$expectation$I40, params = c('epsilon', 'alpha'))
```



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's verify this, by plotting the variance of the mean emulator for $I40$ in the (ϵ, α)-plane, with all unshown parameters to be as in the first row of `all_training`:

```
emulator_plot(stoch_emulators$expectation$I40, 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 (as in this workshop); if it is slower, then we can work with less 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 should run fewer repetitions in those parts of parameter space.

Chapter 6

Implausibility

We will first briefly review the definition of implausibility measure. 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 $\text{Imp}(x)$ is

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

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

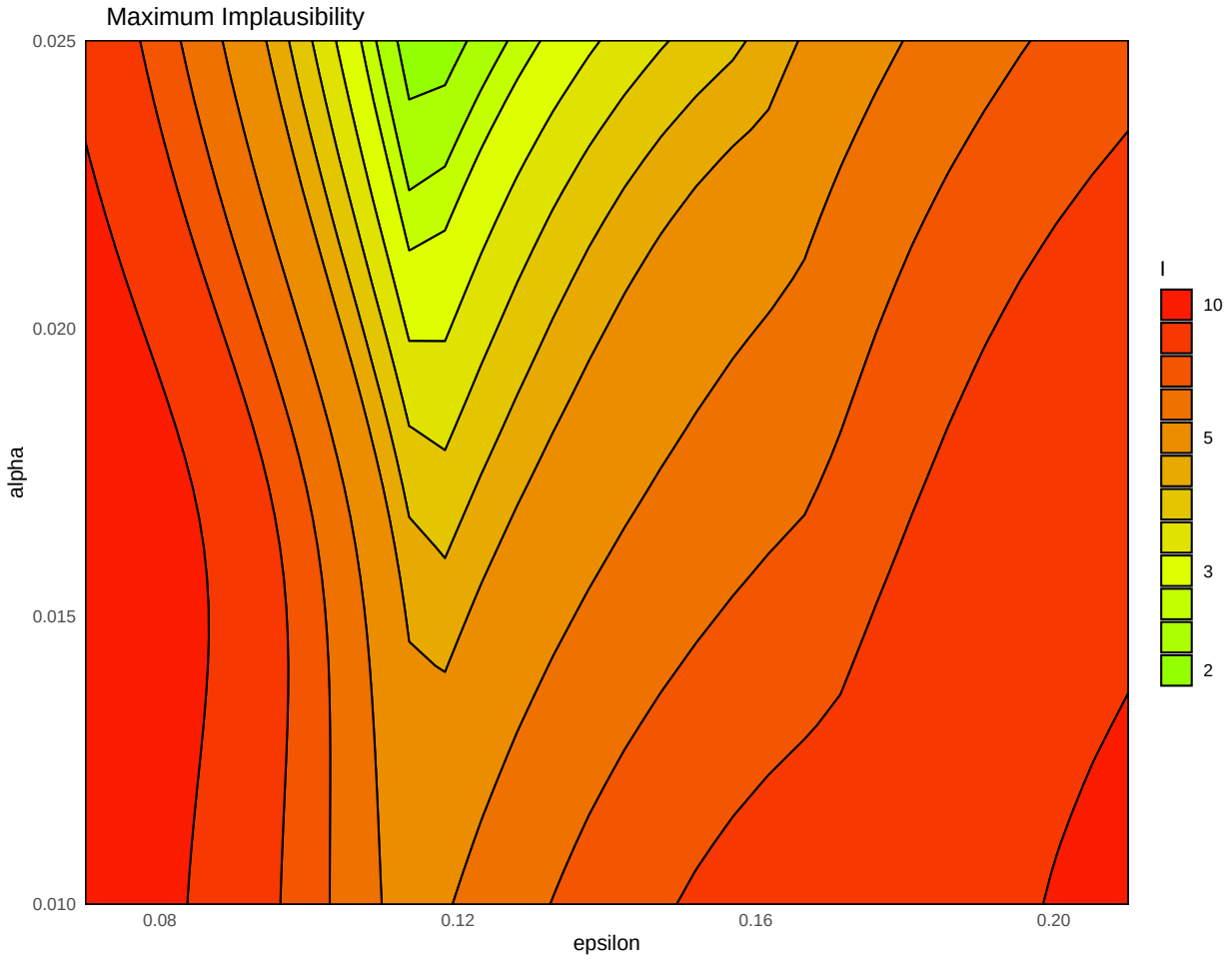
- V_0 is the variance associated with the observation uncertainty;
- $V_c(x)$ refers to the uncertainty one introduces when using the emulator output instead of the model output itself;
- V_s is the ensemble variability and represents the stochastic nature of the model;
- V_m is the model discrepancy, 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`:

```
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 P50

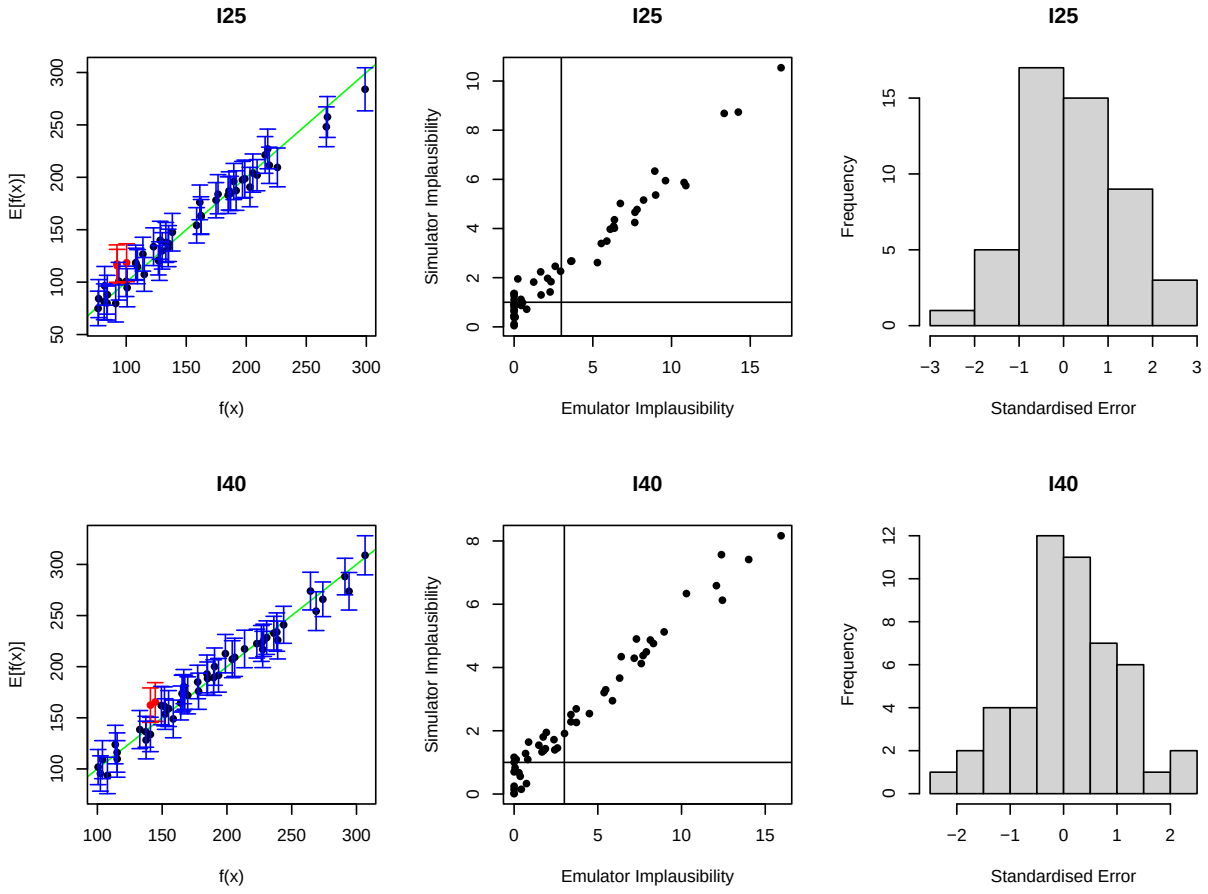
Chapter 7

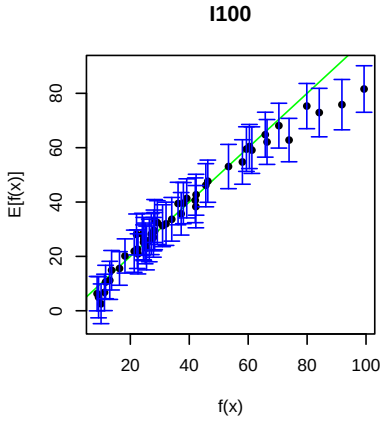
Emulator diagnostics

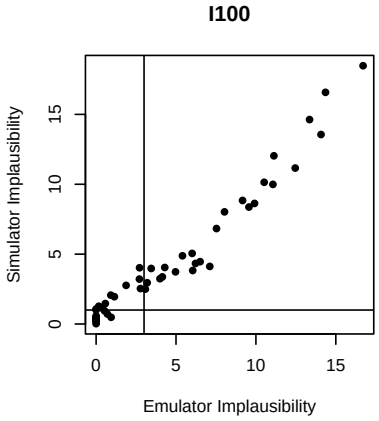
The function `validation_diagnostics` can be used as in the deterministic case, to get three diagnostics for each emulated output.

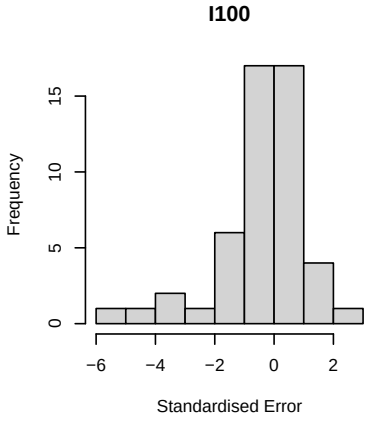
Show: Remind me of what each diagnostics means on P55

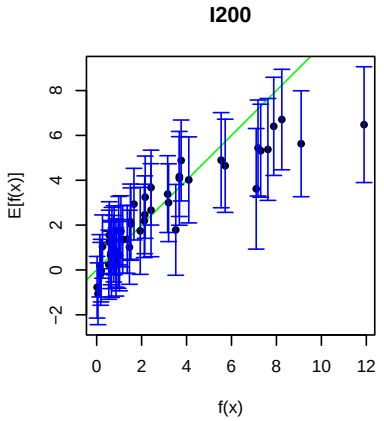
```
vd <- validation_diagnostics(stoch_emulators$expectation, targets, all_valid, plt=TRUE, row=2)
```

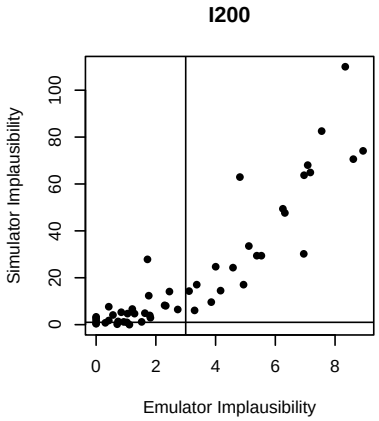


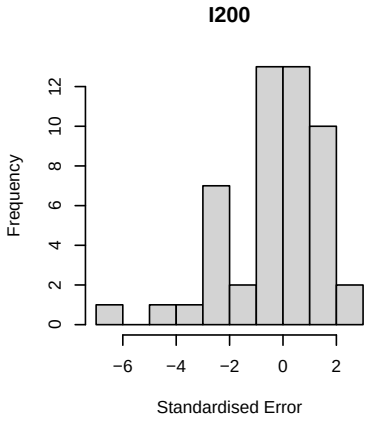

I100
E[f(x)]
f(x)

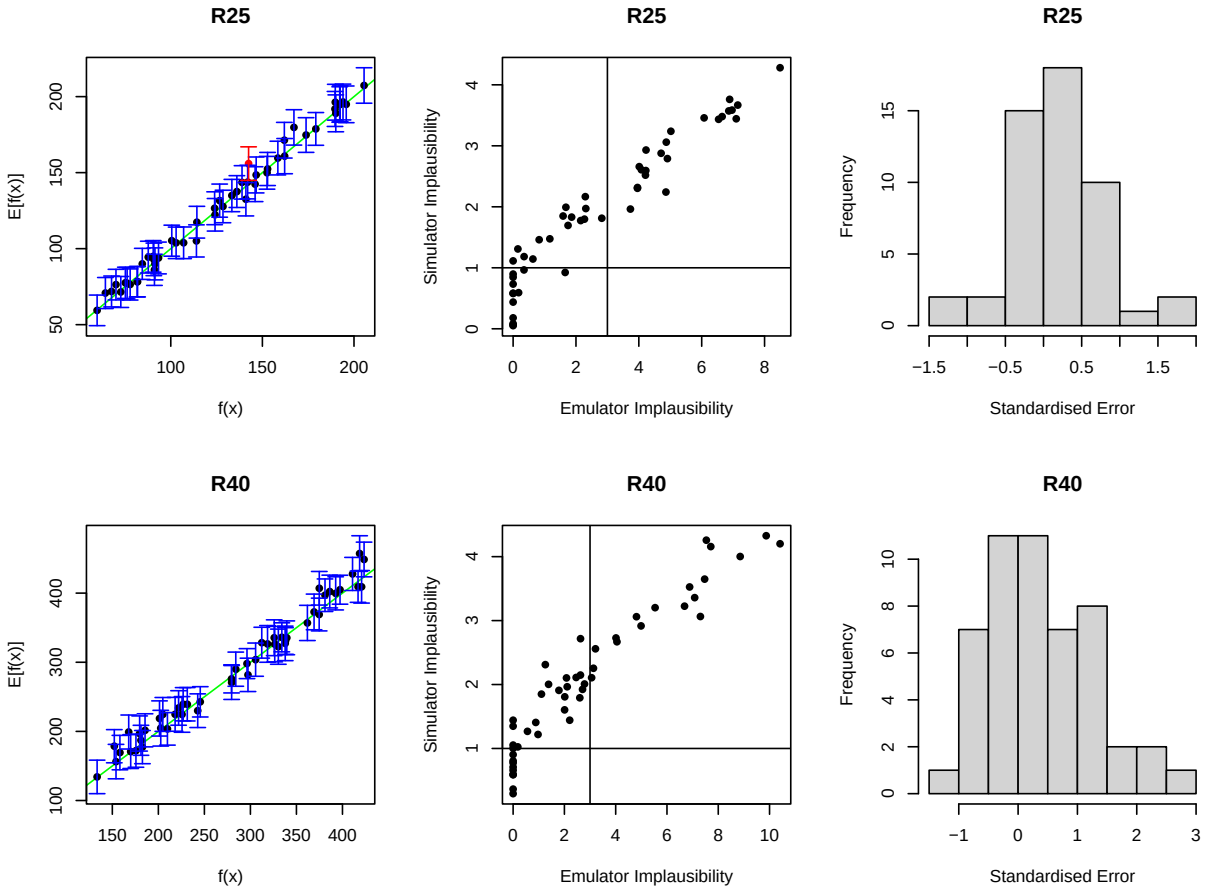

I100
Simulator Implausibility
Emulator Implausibility

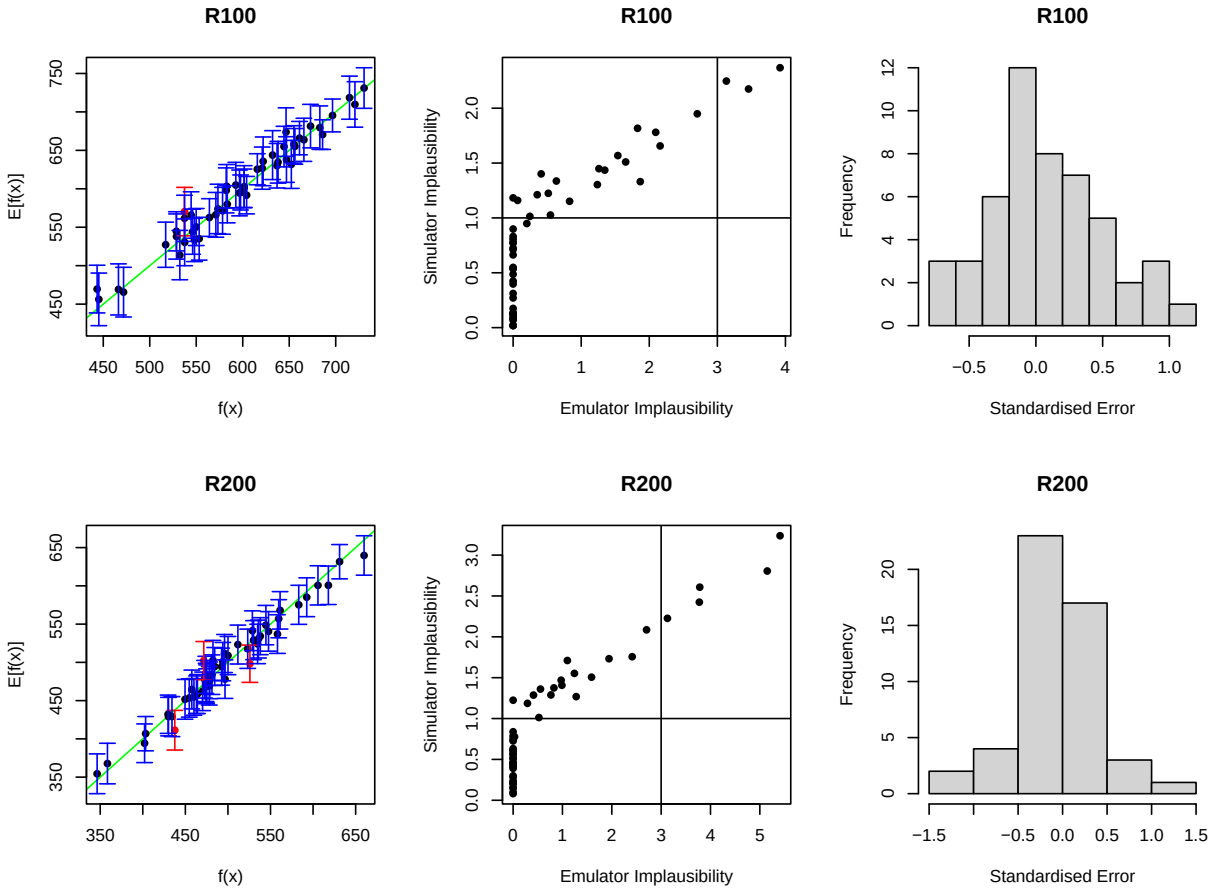

I100
Frequency
Standardised Error


I200
E[f(x)]
f(x)


I200
Simulator Implausibility
Emulator Implausibility


I200
Frequency
Standardised Error


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



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.

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.

Chapter 8

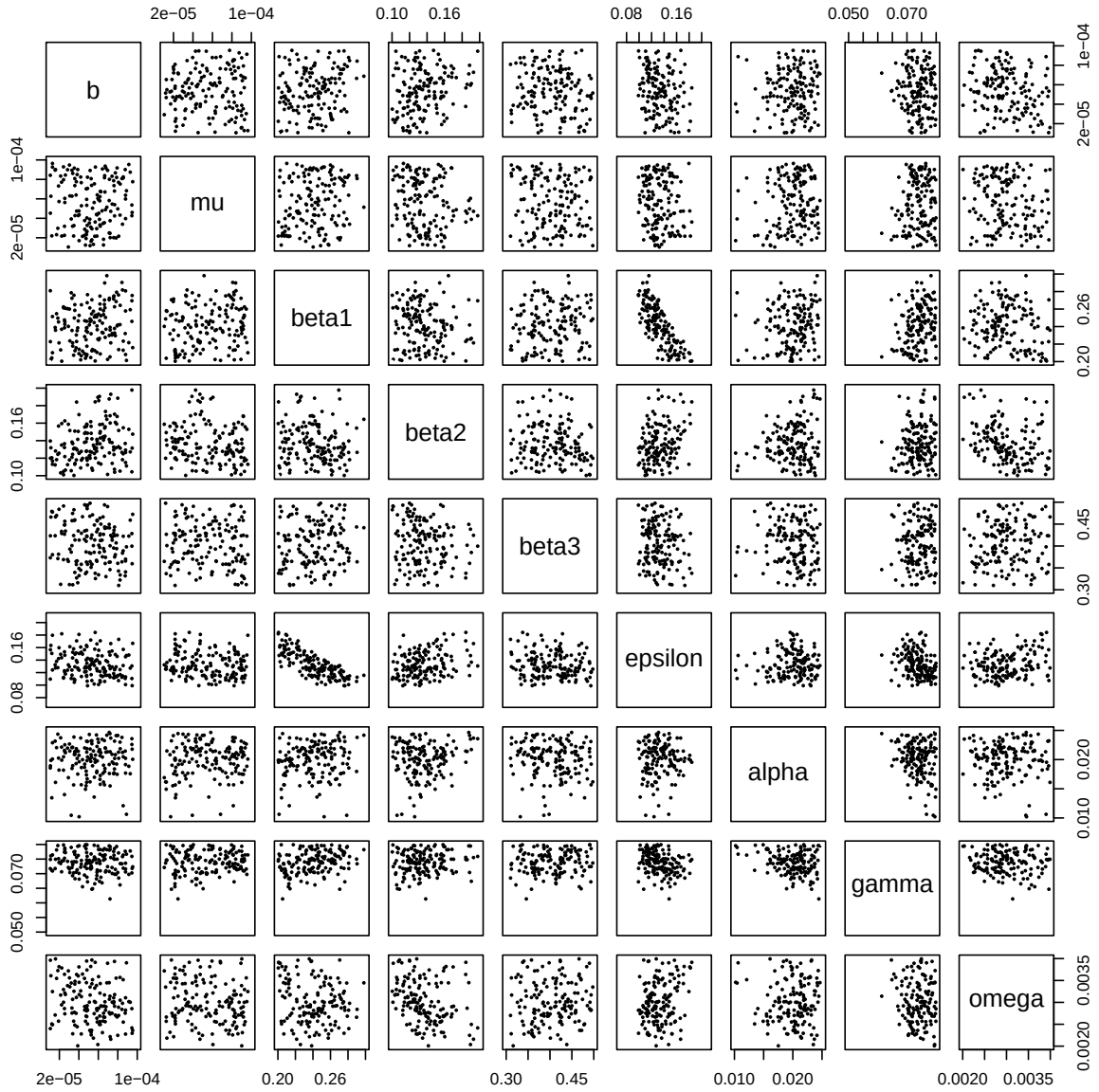
Proposing new points

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

```
new_points <- generate_new_runs(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`:

```
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

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 `variance_emulator_from_data`.

We now run the model on `new_points`, 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. As before 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.

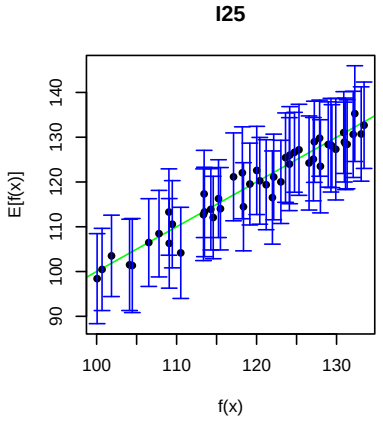
```
new_results <- list()
for (i in 1:nrow(new_points)) {
  model_out <- get_results(unlist(new_points[i,]), nreps = 50, outs = c("I", "R"),
                           times = c(25, 40, 100, 200))
  new_results[[i]] <- model_out
}
wave1 <- data.frame(do.call('rbind', new_results))
new_all_training <-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

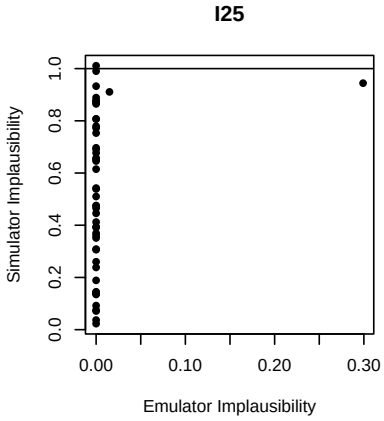
To train new emulators we use `variance_emulator_from_data`, passing the new training data, the outputs names and the initial ranges of the parameters. We also set `check.ranges` to `TRUE` to ensure that the new emulators are trained only on the non-implausible region found in wave one.

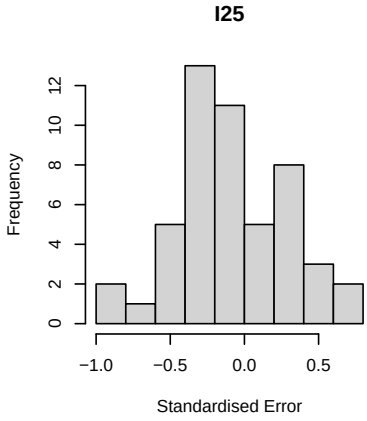
```
new_stoch_emulators <- variance_emulator_from_data(new_all_training, output_names, ranges,
                                                   check.ranges=TRUE)
```

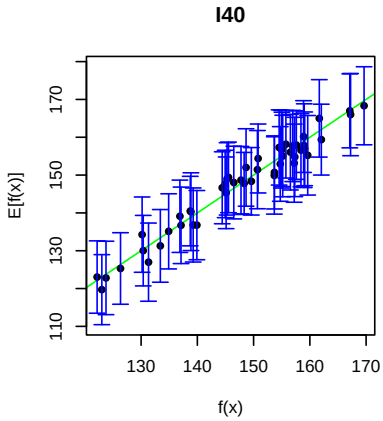
As usual, before using the obtained emulators, we need to check their diagnostics. We use the function `validation_diagnostics` which takes the new emulators, the targets and the new validation data:

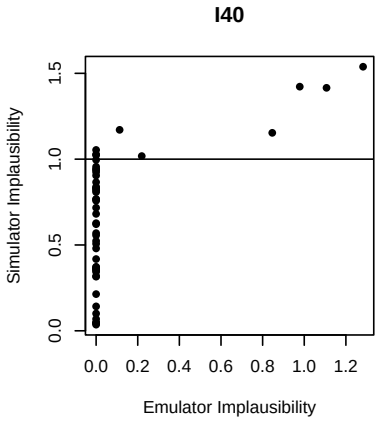
```
vd <- validation_diagnostics(new_stoch_emulators, targets, new_all_valid, plt=TRUE, row=2)
```

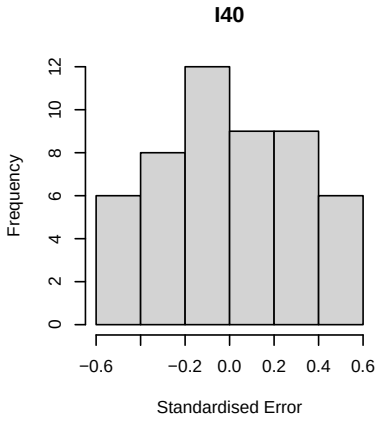

I25
E[f(x)]
f(x)

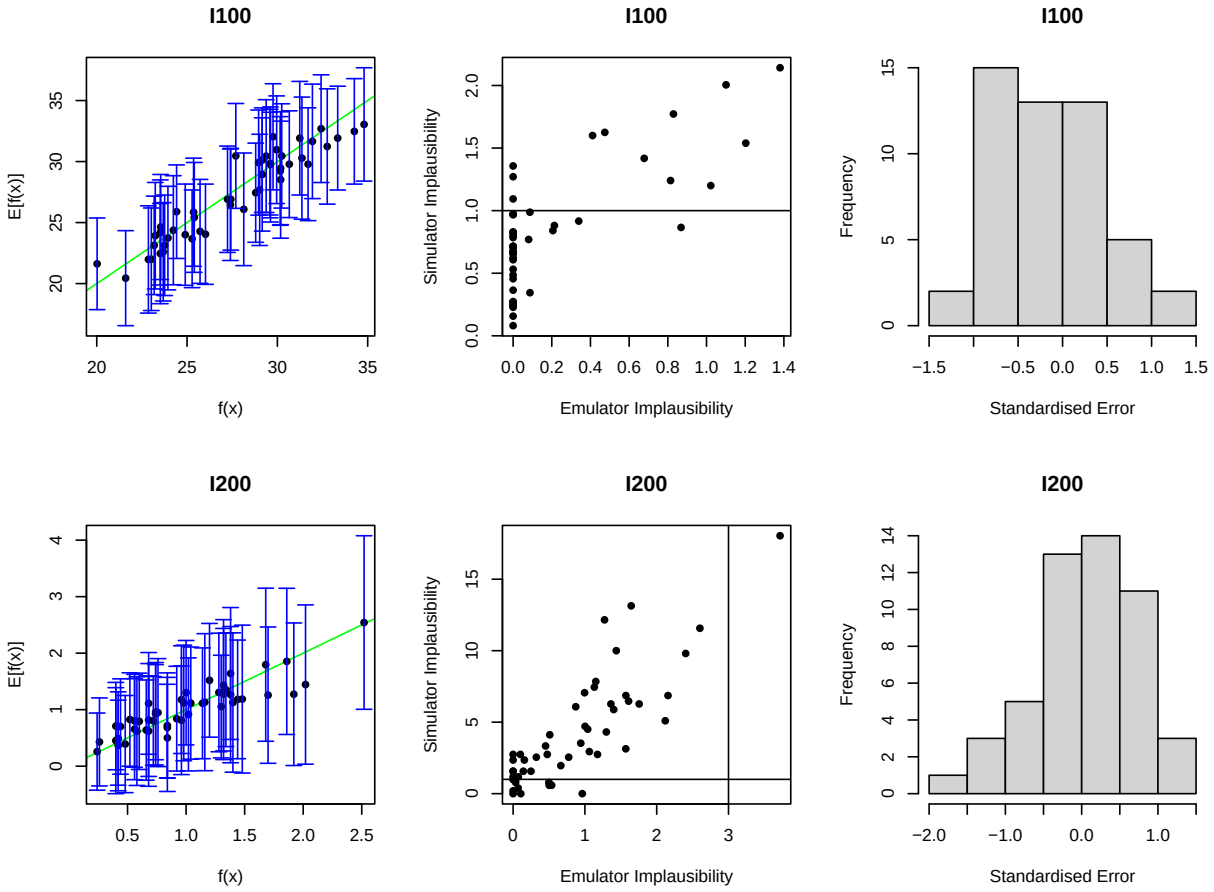

I25
Simulator Implausibility
Emulator Implausibility

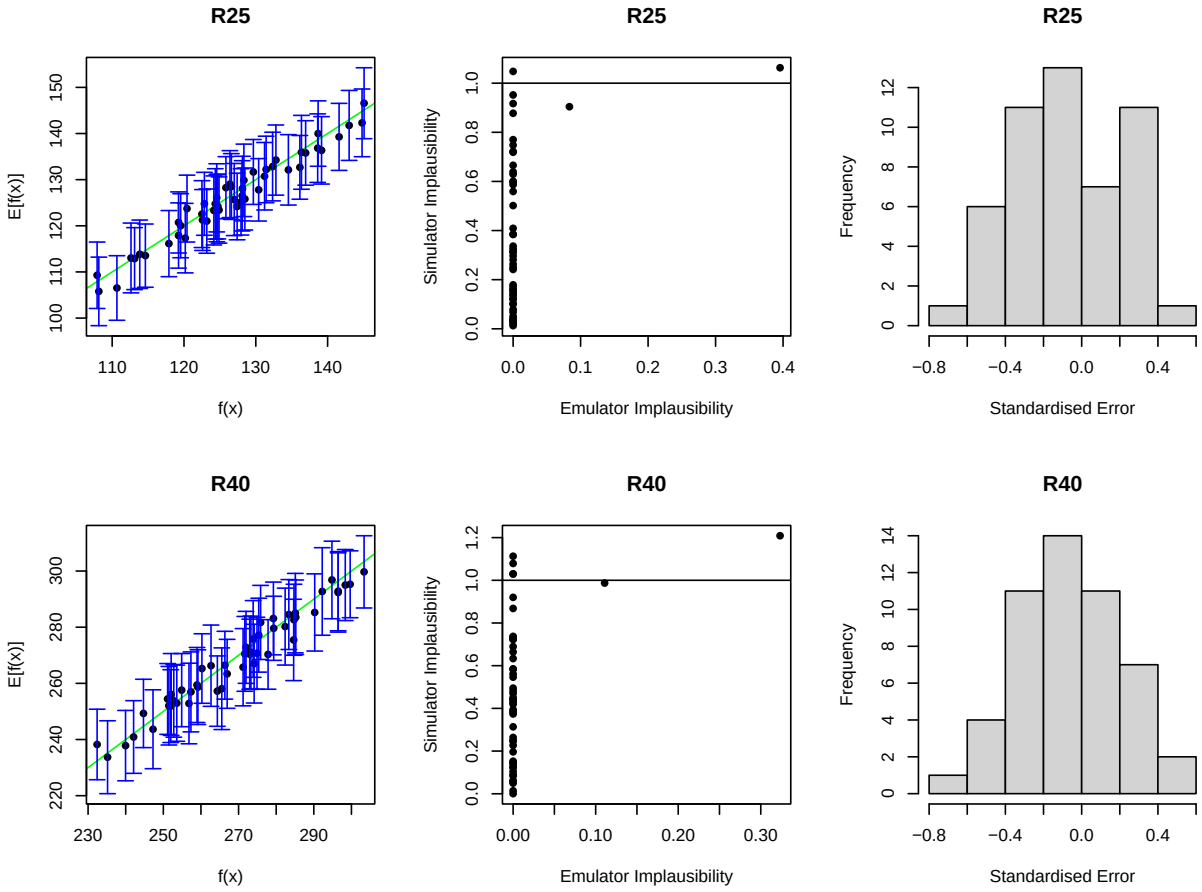

I25
Frequency
Standardised Error

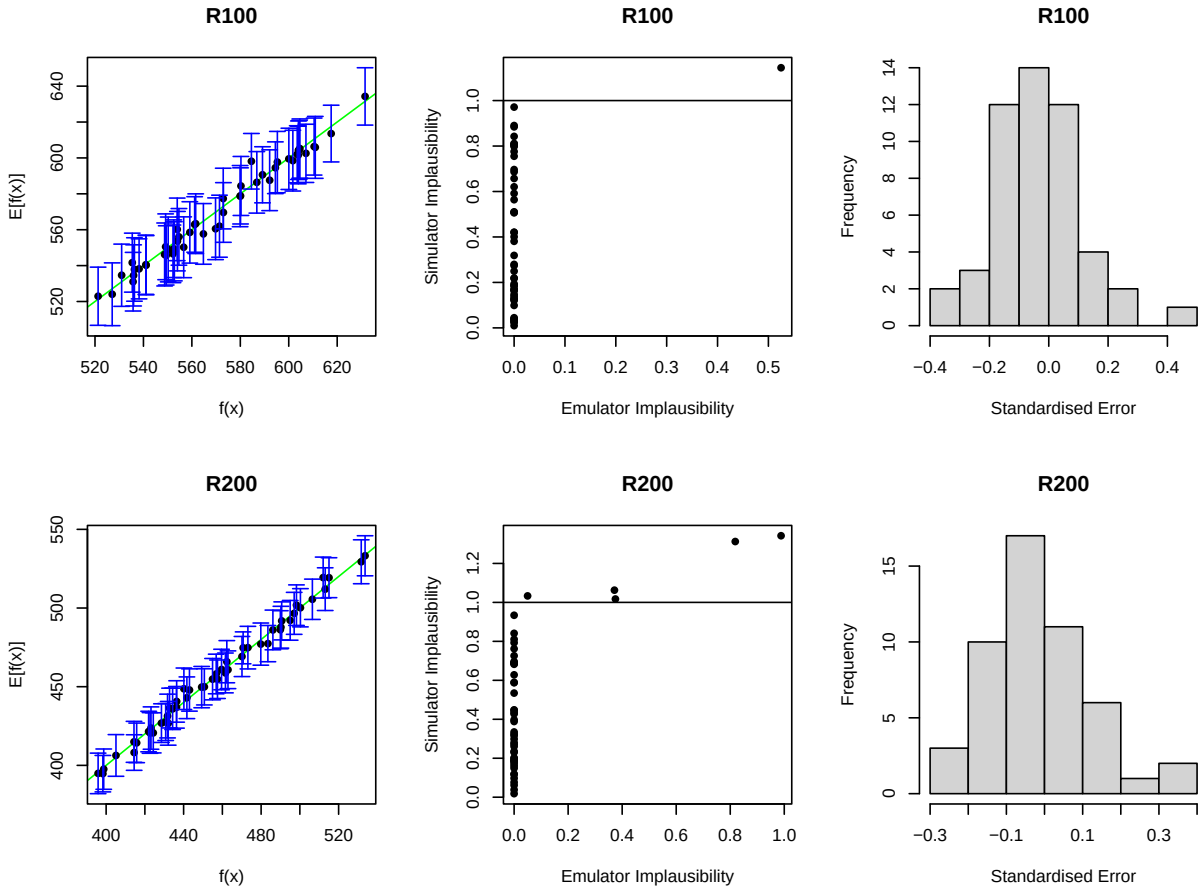

I40
E[f(x)]
f(x)


I40
Simulator Implausibility
Emulator Implausibility


I40
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


R25
E[f(x)]
f(x)
R25
Simulator Implausibility
Emulator Implausibility
R25
Frequency
Standardised Error
R40
E[f(x)]
f(x)
R40
Simulator Implausibility
Emulator Implausibility
R40
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_runs` 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_runs(c(new_stoch_emulators, stoch_emulators), 150, targets)
```

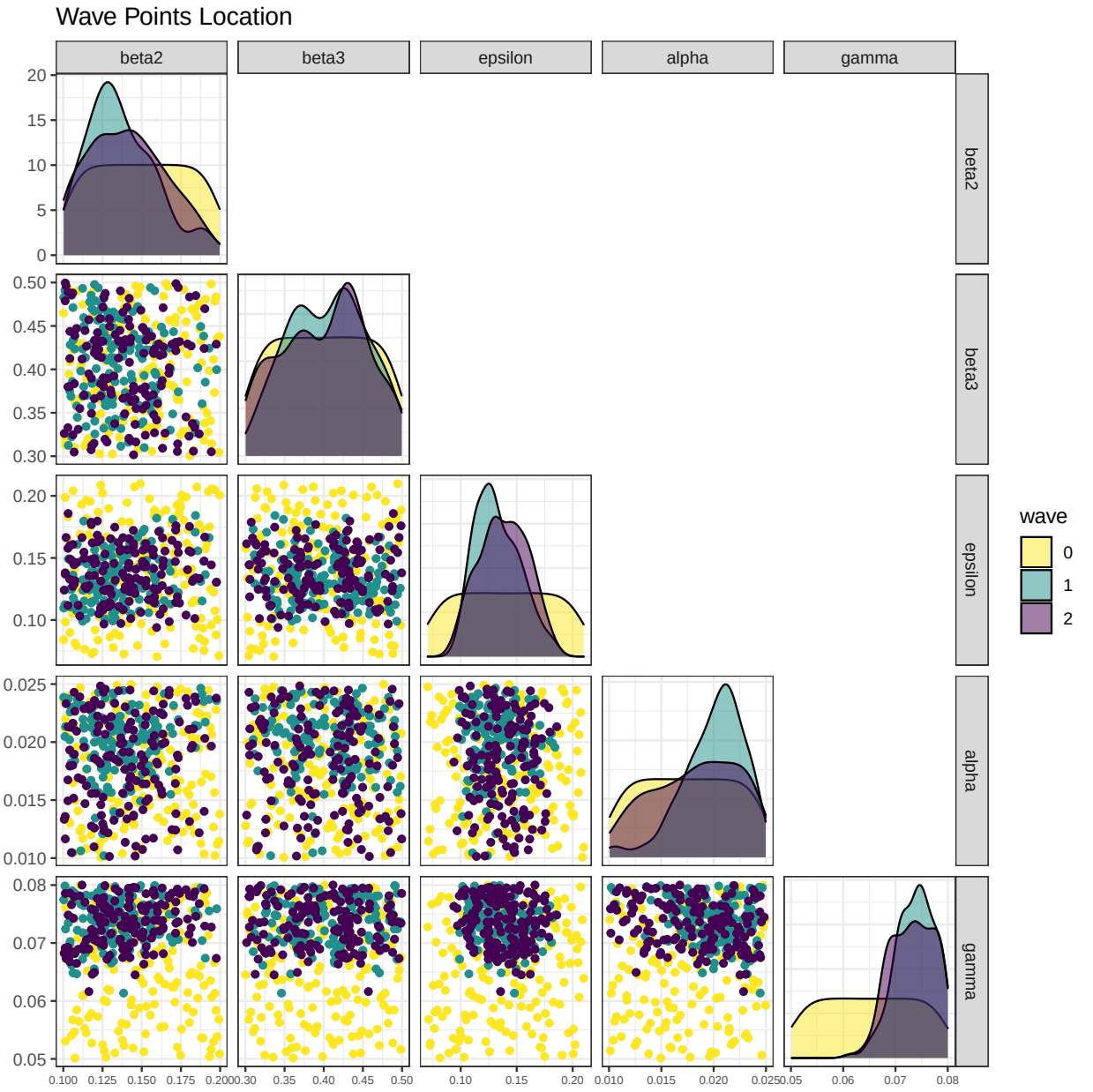
Chapter 10

Visualisation of the non-implausible space by wave

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)[4:8]) +
            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`. Let us start by evaluating the model at parameter sets in `new_new_points`:

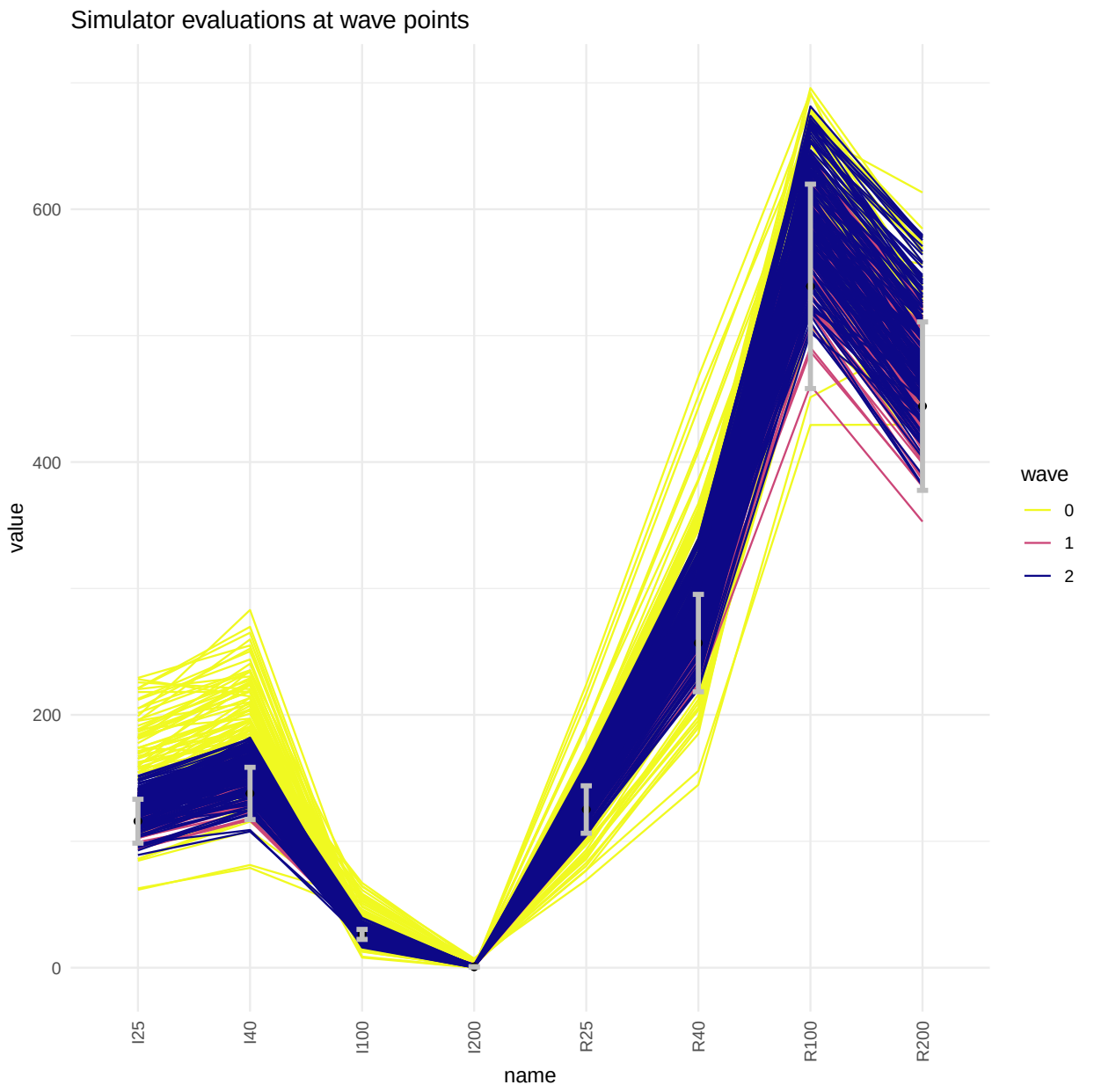
```
new_new_training_list <- list()
for (i in 1:nrow(new_new_points)) {
  model_out <- get_results(unlist(new_new_points[i,]), nreps = 50, outs = c("I", "R"),
                           times = c(25, 40, 100, 200, 300, 350, 450))
  new_new_training_list[[i]] <- model_out
}
new_new_all_training <-  data.frame(do.call('rbind',new_new_training_list))
```

In this workshop we assume that for a given parameter set x, we are interested in the mean of each model output at x across different realisations, rather than in the output of each realisation at x. 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`:

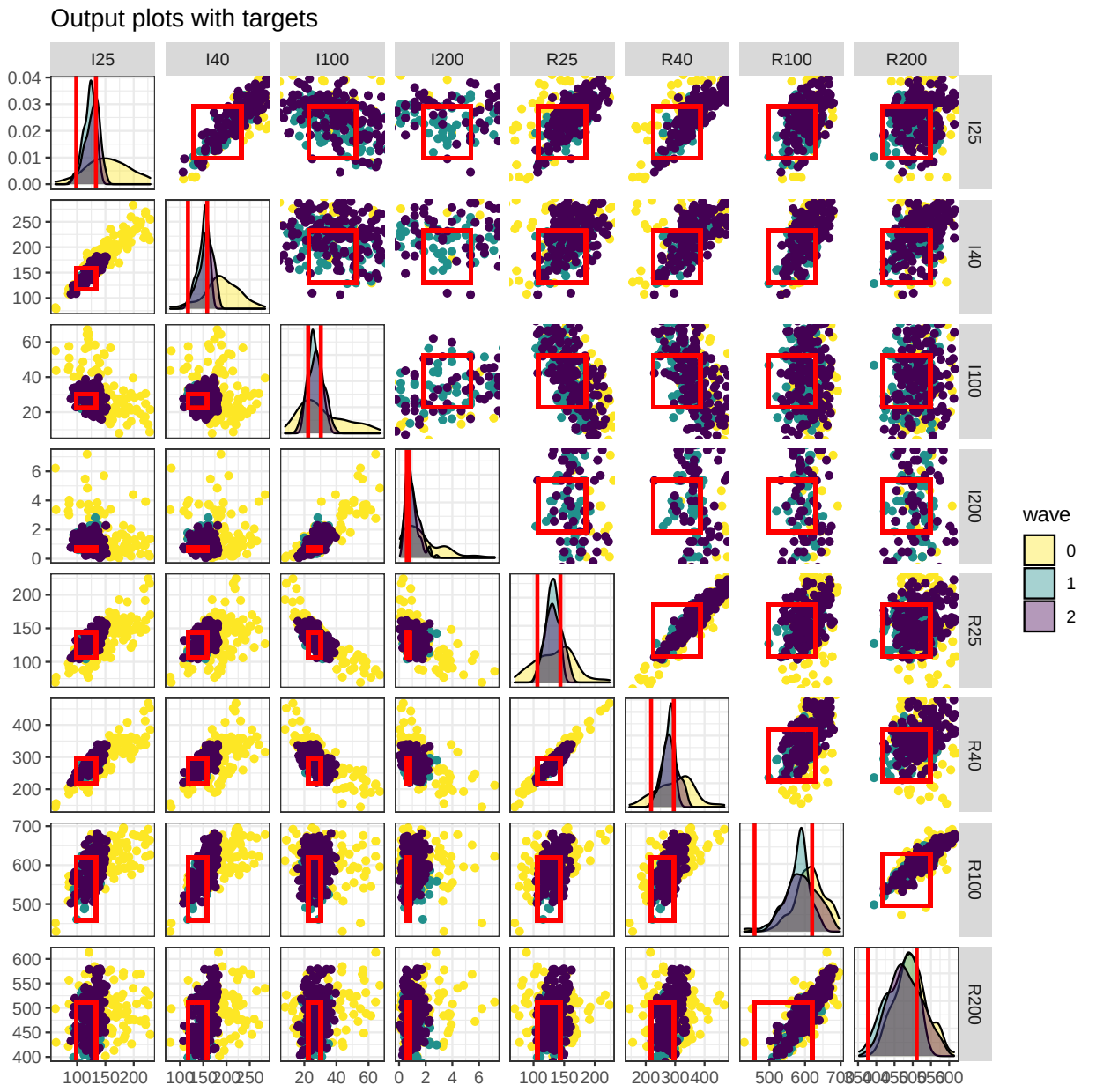
```
simulator_plot(all_aggregated, targets, barcol = "grey")
```



Note that in this call we set `barcol="white"` 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 $R25$ and $R40$.

In this workshop we have shown how to perform the first two 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 P56

We conclude by noting that in 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.

The most common example of bimodality is ‘take off vs die out’: if the initial number of infected people is low, the disease might infrequently find its way into the population, but blowing up on those rare circumstances it does). The calibration of stochastic models that can have a bimodal behaviour is addressed in Workshop 3.

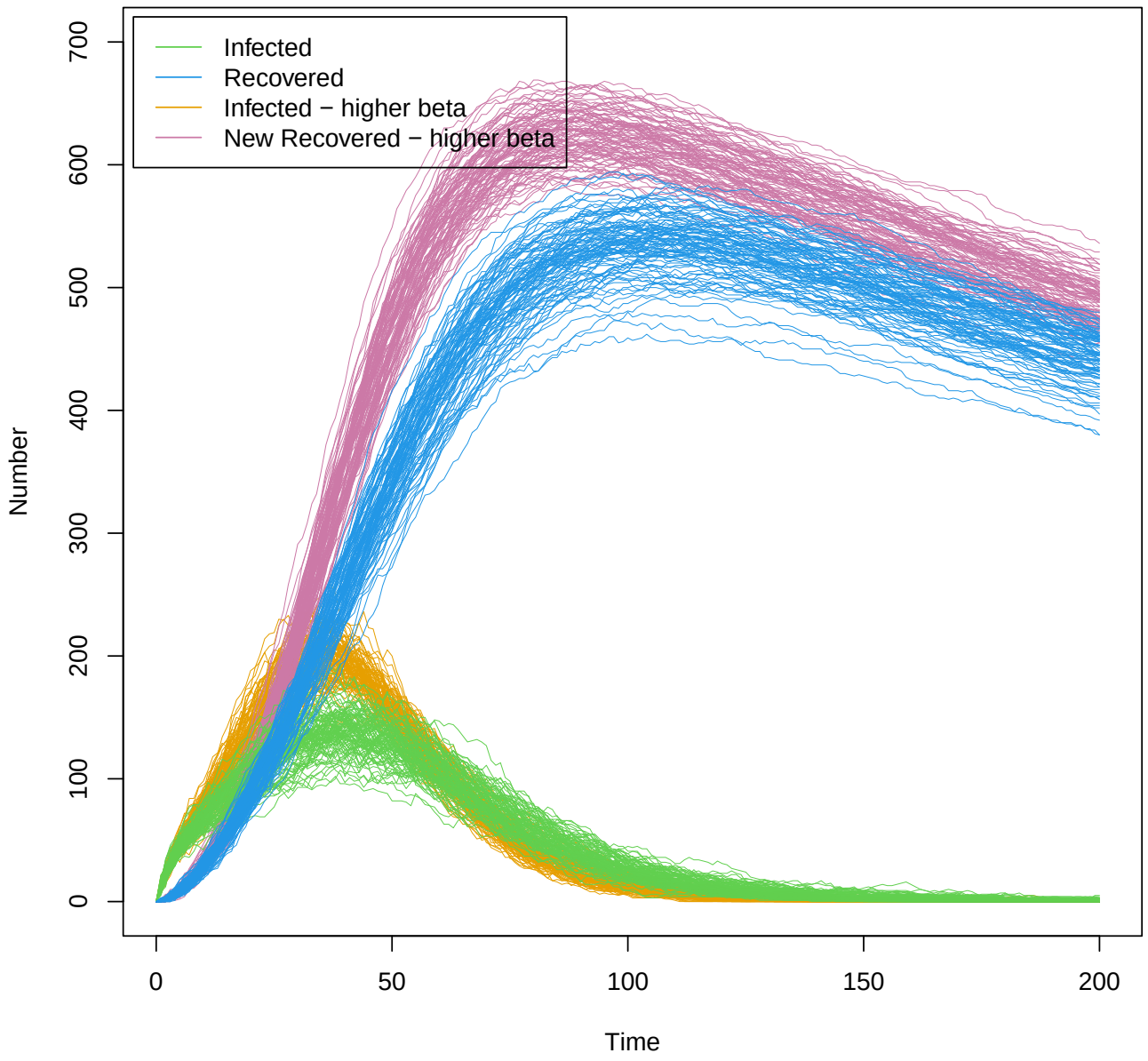
Appendix A

Answers

Solution 1

Let us see what happens when a higher rate of infection between each infectious and susceptible person is considered:

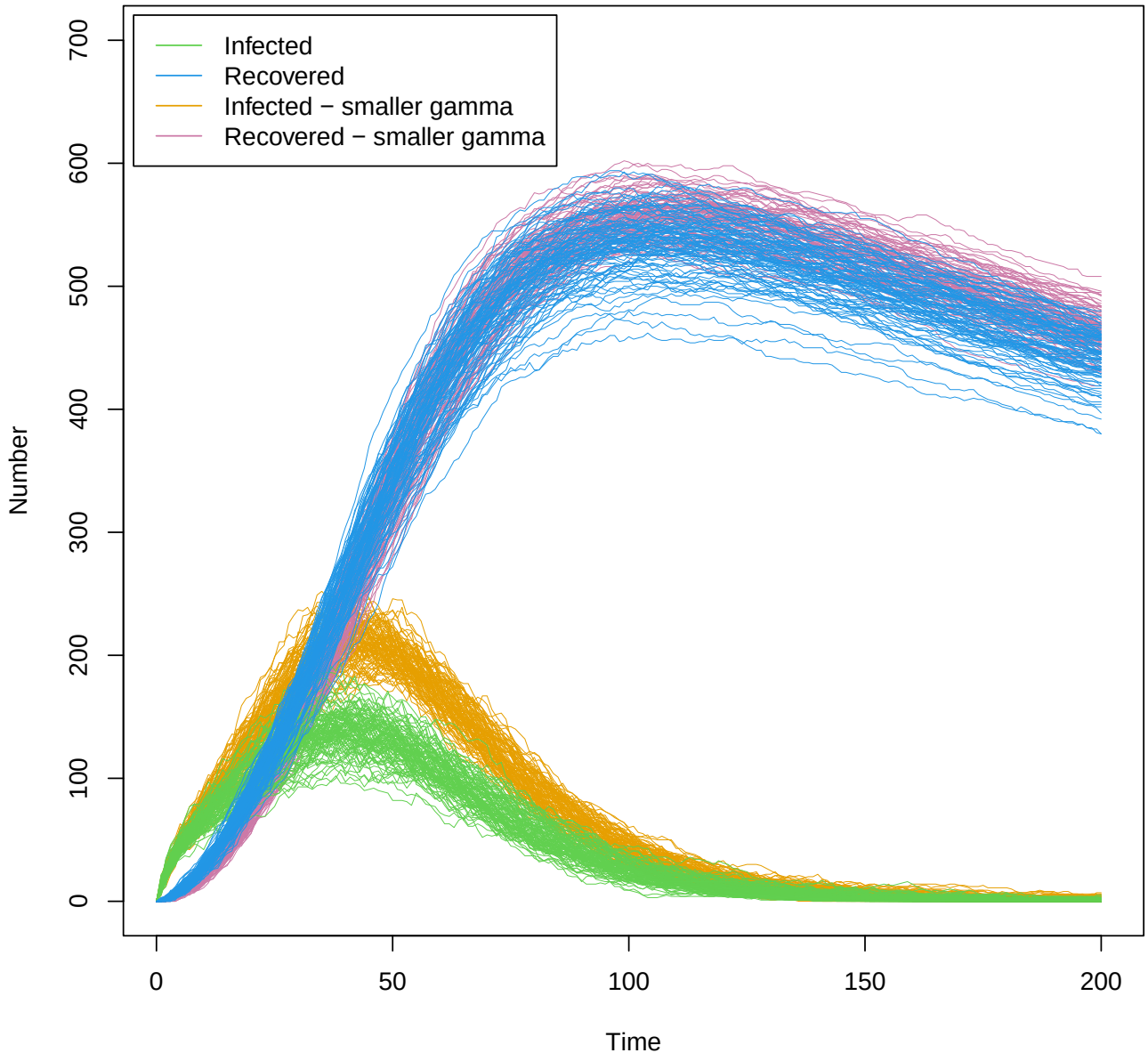
```
higher_beta_params <- c(
  b = 1/(76*365),
  mu = 1/(76*365),
  beta1 = 0.3, beta2 = 0.1, beta3 = 0.5,
  epsilon = 1/7,
  alpha = 1/50,
  gamma = 1/14,
  omega = 1/365
)
higher_beta_solution <- get_results(higher_beta_params, outs = c("I", "R"),
                                    times = c(25, 40, 100, 200), raw = TRUE)
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, higher_beta_solution[,j,i], col=(c("#E69F00","#CC79A7"))[j-2], l
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", "Infected - higher beta", "New Recovered - higher beta
       col = c(3,4,"#E69F00","#CC79A7"), inset = c(0.01, 0.01))
```



Comparing this with the plot produced before the task, we see that the peaks of recovered and infectious individuals have now increased, as expected.

What happens when a lower value of the recovery rate γ is used?

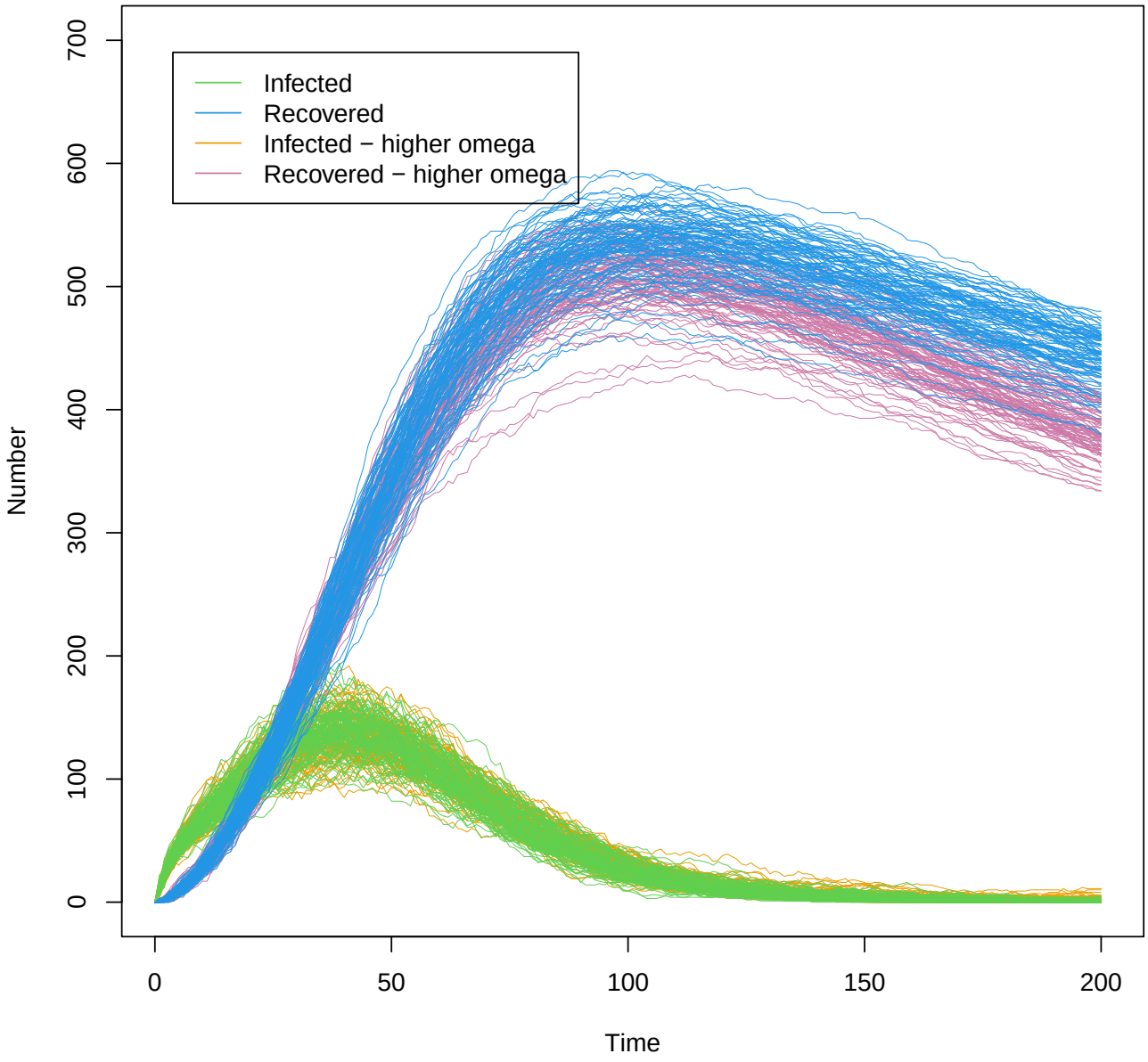
```r
smaller_gamma_params <- c(
  b = 1/(60*365),
  mu = 1/(76*365),
  beta1 = 0.214, beta2 = 0.107, beta3 = 0.428,
  epsilon = 1/7,
  alpha = 1/50,
  gamma = 1/20,
  omega = 1/365
)
smaller_gamma_solution <- get_results(smaller_gamma_params, outs = c("I", "R"),
                                      times = c(25, 40, 100, 200), raw = TRUE)
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, smaller_gamma_solution[,j,i],
                                    col=(c("#E69F00","#CC79A7"))[j-2], lwd=0.3)
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", "Infected - smaller gamma", "Recovered - smaller gamma
       col = c(3,4,"#E69F00","#CC79A7"), inset = c(0.01, 0.01))
```



As one expects, this causes the peak of infectious individuals to increase.

Finally, what happens when we increase the rate at which immunity is lost ω?

```r
higher_omega_params <- c(
  b = 1/(60*365),
  mu = 1/(76*365),
  beta1 = 0.214, beta2 = 0.107, beta3 = 0.428,
  epsilon = 1/7,
  alpha = 1/50,
  gamma = 1/14,
  omega = 0.004
)
higher_omega_solution <- get_results(higher_omega_params, outs = c("I", "R"),
                                     times = c(25, 40, 100, 200), raw = TRUE)
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, higher_omega_solution[,j,i],
                                    col=(c("#E69F00","#CC79A7"))[j-2], lwd=0.3)
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", "Infected - higher omega", "Recovered - higher omega"
       col = c(3,4,"#E69F00","#CC79A7"), inset = c(0.05, 0.05))
```



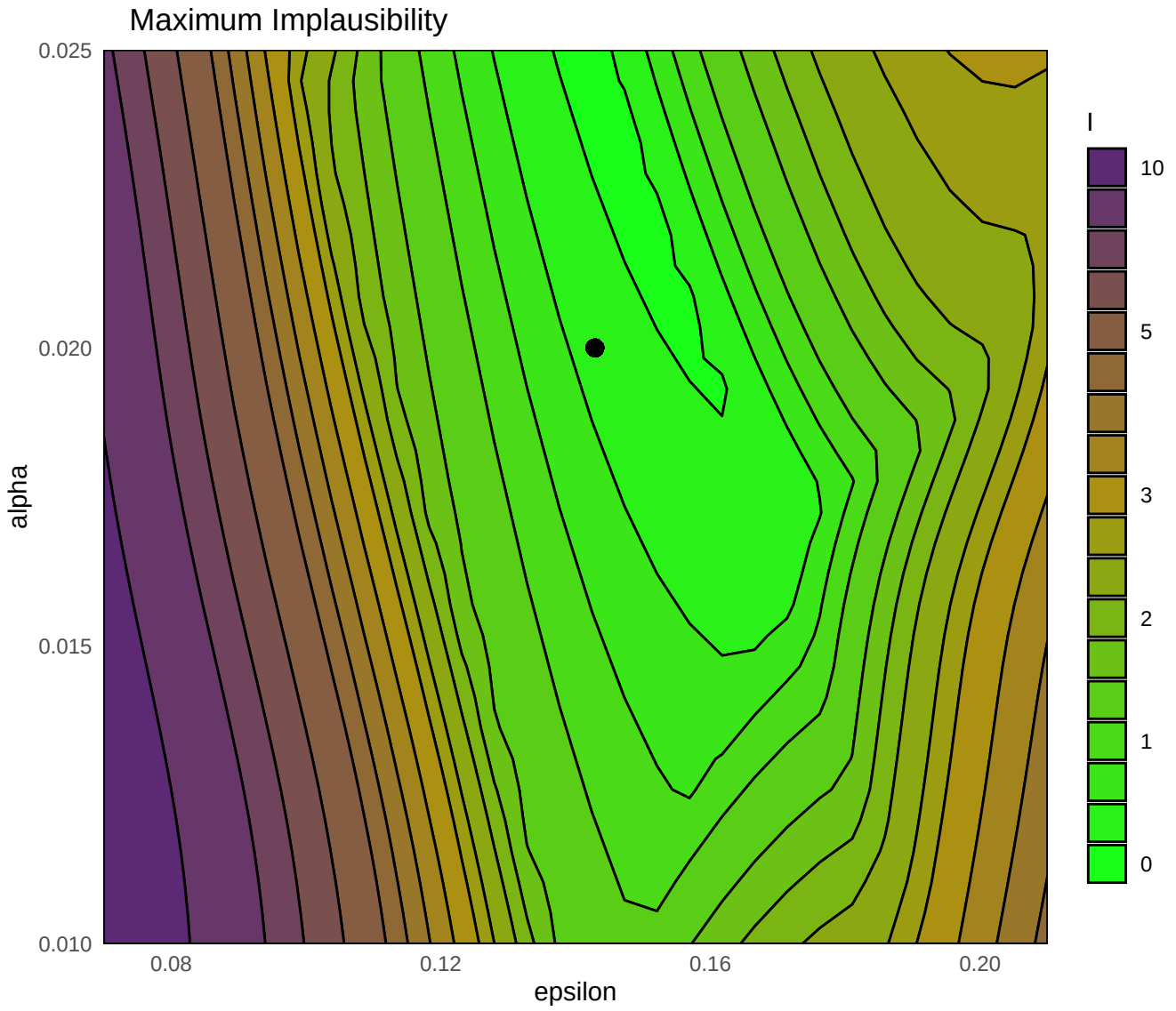
This causes the peak of recovered individuals to slightly decrease.

Return to task on P13

Solution 2

We set `fixed_vals` to `chosen_params[!names(chosen_params) %in% c('epsilon', 'alpha')]`, plot the maximum implausibility and add a point corresponding to the values of σ and $alpha$ in `chosen_params`:

```
emulator_plot(stoch_emulators, plot_type = 'nimp', targets = targets,
              params = c('epsilon', 'alpha'),
              fixed_vals = chosen_params[!names(chosen_params) %in% c('epsilon', 'alpha')],
              cb=T) +geom_point(aes(x=1/7, y=1/50), size=3)
```



The plot shows what we expected: when ϵ and α are equal to their values in `chosen_params`, the implausibility measure is below the threshold 3 (cf. black point in the box).

Return to task on P26

Appendix B

Additional information

Code to load relevant libraries and helper functions

```
library(hmer)
library(lhs)
library(deSolve)
library(ggplot2)
library(reshape2)
library(purrr)
library(tidyverse)
############################## HELPER FUNCTIONS ##############################
# Function that calculates the mean of the model outputs from several runs
# at the same parameter set
aggregate_points <- function(data, input_names, func = 'mean') {
  unique_points <- unique(data[,input_names])
  uids <- apply(unique_points, 1, rlang::hash)
  data_by_point <- purrr::map(uids, function(h) {
    data[apply(data[,input_names], 1, rlang::hash) == h,]
  })
  aggregate_func <- get(func)
  aggregate_values <- do.call('rbind.data.frame', purrr::map(data_by_point, function(x) {
    output_data <- x[,!(names(x) %in% input_names)]
    apply(output_data, 2, aggregate_func)
  }))
  return(setNames(cbind.data.frame(unique_points, aggregate_values), names(data)))
}
# Function that implements the SEIRS stochastic model using the Gillespie algorithm
gillespied=function (N, T=400, dt=1, ...)
{
  tt=0
  n=T%/%dt
  x=N$M
  S=t(N$Post-N$Pre)
  u=nrow(S)
  v=ncol(S)
  xmat=matrix(0,ncol=u,nrow=n)
  i=1
  target=0
  repeat {
    h=N$h(x, tt, ...)
    h0=sum(h)
    if (h0<1e-10)
      tt=1e99
    else
      tt=tt+rexp(1,h0)
    while (tt>=target) {
      xmat[i,]=x
      i=i+1
      target=target+dt
      if (i>n)
        return(xmat)
    }
    j=sample(v,1,prob=h)
    x=x+S[,j]
  }
}
Num <- 1000
N=list()
N$M=c(900,100,0,0,0)
N$Pre = matrix(c(0,0,0,0,0,
                 1,0,0,0,0,
                 1,0,0,0,0,
                 0,1,0,0,0,
                 0,1,0,0,0,
```

Return to P5

R tip

Copy the code below, modify the value of (some) parameters and run it.

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

Return to P13

Remind me of what each diagnostics means

The first column 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. 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 emulator 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).

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.

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.
- 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.

Return to P27

Remind me of possible stopping criteria

Since this is an iterative process, at the end of each wave we need to decide whether to perform a new wave or to stop. One possible stopping criterion consists of comparing the emulator uncertainty and the target uncertainty. If the former is larger, another wave can be performed, since new, more confident emulators can potentially help further reduce the non-implausible space. If the uncertainty of emulators is smaller than the uncertainty in the targets, improving the performance of emulators would not make a substantial difference, and additional waves would not be beneficial. We may also choose to stop the iterations when we get emulators that provide us with full fitting points at a sufficiently high rate. In such a case, rather than spending time training new emulators, we can generate new points with the function `generate_new_runs` using the current emulators until we find enough full fitting ones. Finally, we might end up with all the input space deemed implausible at the end of a wave. In this situation, we would deduce that there are no parameter sets that give an acceptable match with the data: in particular, this would raise doubts about the adequacy of the chosen model, or input and/or output ranges.

Return to P43