Short tutorial on the hmer package

Danny Scarponi, Andy Iskauskas

Contents

Chapter 1

Objectives

This workshop offers a short introduction to the main functionality of the hmer package, using a simple, deterministic epidemiological model. hmer allows you to efficiently implement the Bayes Linear emulation and history matching process, a calibration method that has been successfully employed in a wide range of fields (e.g. epidemiology, cosmology, climate science, systems biology, geology, energy systems).

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 P80

Chapter 2

An overview of history matching with emulation and hmer

2.1 Why do we need history matching with emulation?

Computer models, otherwise known as simulators, are widely used in many fields in science and technology, to represent the fundamental dynamics of physical systems. Due to the complexity of the interactions within a system, computer models frequently contain large numbers of parameters.

Before using a model for projection or planning, it is fundamental to explore plausible values for its parameters, calibrating the model to the observed data. This poses a significant problem, considering that it may take several minutes or even hours for the evaluation of a single run of a complex model. As a consequence, a comprehensive analysis of the entire input space, requiring vast numbers of model evaluations, is often unfeasible. Emulation, combined with history matching, allows us to overcome this issue.

2.2 History Matching

History matching concerns the problem of identifying those parameter sets that may give rise to acceptable matches between the model outputs and the observed data. History matching proceeds as a series of iterations, called **waves**, where **implausible** areas of parameter space, i.e. areas that are unable to provide a match with the observed data, are identified and discarded. Each wave focuses the search for implausible space in the space that was characterized as non-implausible in all previous waves: thus the non-implausible space shrinks with each wave. To decide whether a parameter set x is implausible we introduce the **implausibility measure**, which evaluates the difference between the model results and the observed data. If such measure is too high, the parameter set is discarded in the next wave of the process.

Note that history matching as just described still relies on the evaluation of the model at a large number of parameter sets, which is often unfeasible. Here is where emulators play a crucial role.

2.3 Emulators

A long established method for handling computationally expensive models is to first construct an emulator: a **fast statistical approximation of the model** that can be used as a surrogate. One can either construct an emulator for each model output separately, or combine outputs together, using more advanced techniques. In this tutorial each model output will have its own emulator.

The model is run at a manageable number of parameter sets to provide training data for the emulator (typically at least $10p$ parameter sets, where p is the number of parameters). The emulators are then built and can be used to obtain an expected value of each model output for any parameter set x, along with an estimate of the uncertainty in the approximation.

Emulators have two useful properties. First, they are **computationally efficient** - typically several orders of magnitude faster than the computer models they approximate. Second, they allow for the uncertainty in their approximations to be taken into account. These two properties mean that emulators can be used to make inferences as a surrogate for the model itself. In particular, it is possible to evaluate the implausibility measure at any given parameter set by comparing the emulator output to observed data, rather than using model output. This greatly speeds up the process and allows for a comprehensive exploration of the input space.

The general structure of a univariate emulator is as follows:

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

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

The regression term is specified by:

- a vector of functions of the parameters $g(x)$ which determine the shape and complexity of the regression hypersurface we fit to the training data.
- a vector of regression coefficients ξ.

The term $u(x)$ captures the local deviations of the output from the regression hypersurface. For each parameter x, we have a random variable $u(x)$ representing the residual at x. All $u(x)$ are assumed to have the same variance σ^2: the larger the value of σ, the farthest the model output can be from the regression hypersurface. In particular, larger values of the **emulator variance** σ correspond to more uncertain emulators. For more details, please see our more comprehensive Tutorial 2, which explains how emulators are defined and trained using the Bayes Linear methodology.

Choosing a value for σ corresponds to making a judgment about how far we expect the output to be from the regression hypersurface. The hmer package, and in particular the function `emulator_from_data`, selects values of σ (and of other relevant hyperparameters) for us based on the provided training data.

2.4 History matching with emulation workflow

We now show the various steps of the history matching with emulation workflow and explain how each step can be performed using hmer functions.

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.

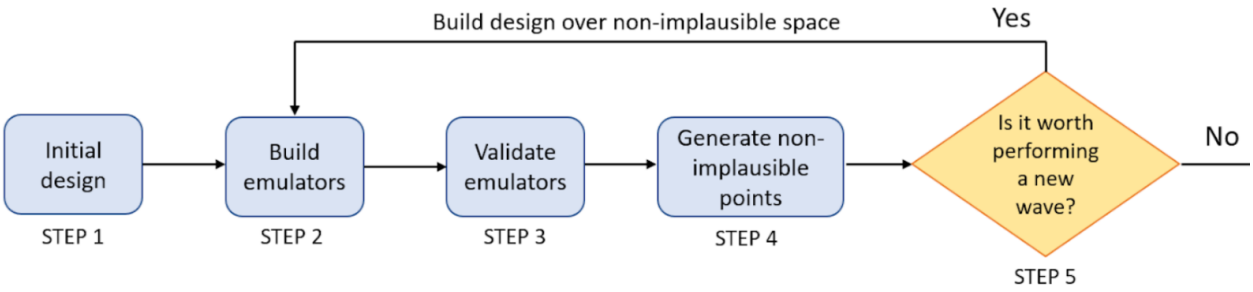


Figure 2.1: History matching with emulation workflow

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 emulators (step 2)	*emulator_from_data*	• The training data • The outputs to emulate • The ranges of parameters	An emulator of the mean for each output of interest	5 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	7 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	8 Proposing new po

Chapter 3

Introduction to the model

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

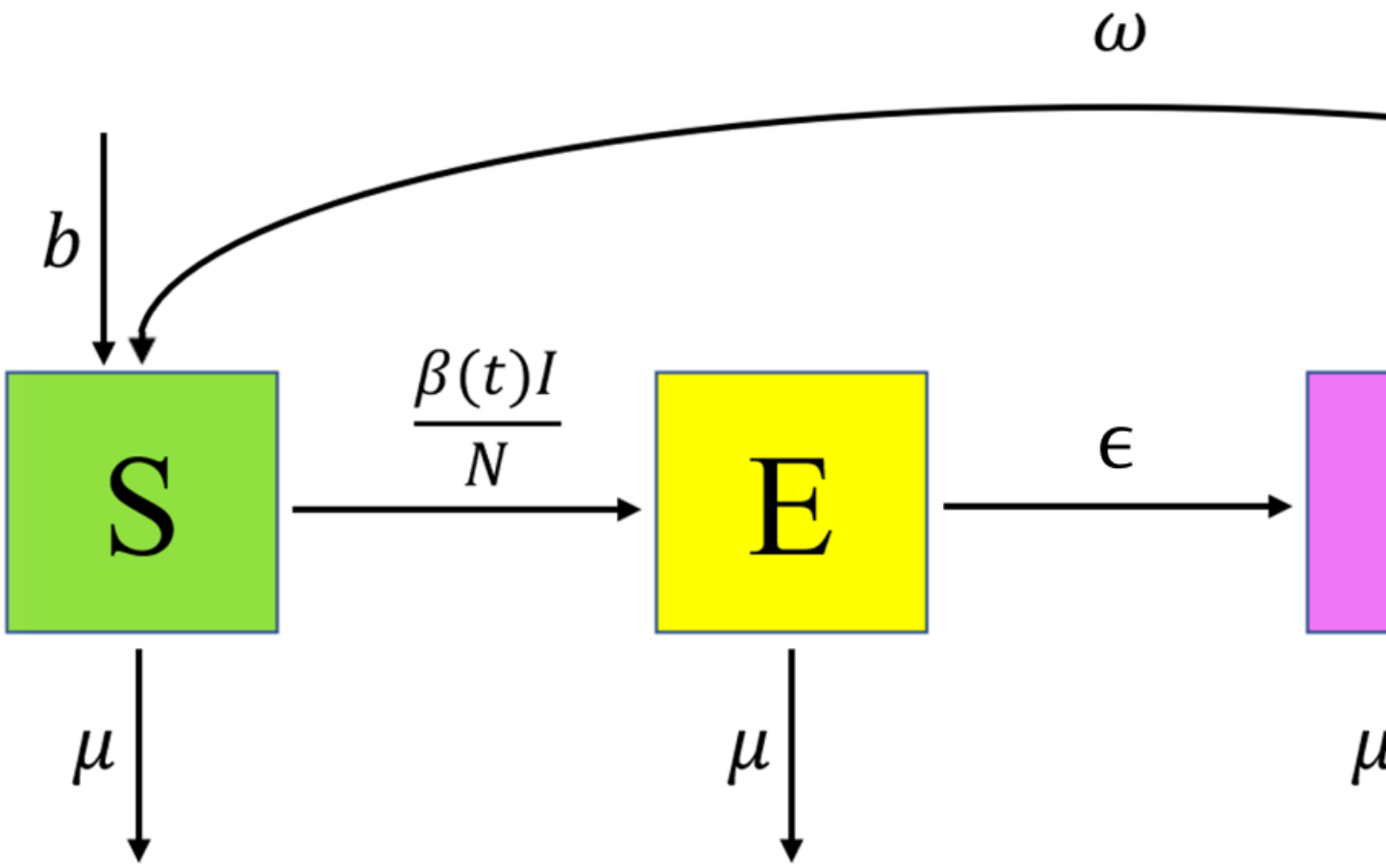


Figure 3.1: SEIRS Diagram

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

following differential equations:

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

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

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

```
example_params <- c(
  b = 1/(60*365),
  mu = 1/(76*365),
  beta1 = 0.2, beta2 = 0.1, beta3 = 0.3,
  epsilon = 0.13,
  alpha = 0.01,
  gamma = 0.08,
  omega = 0.003
)
solution <- ode_results(example_params)
par(mar = c(2, 2, 2, 2))
plot(solution)
```

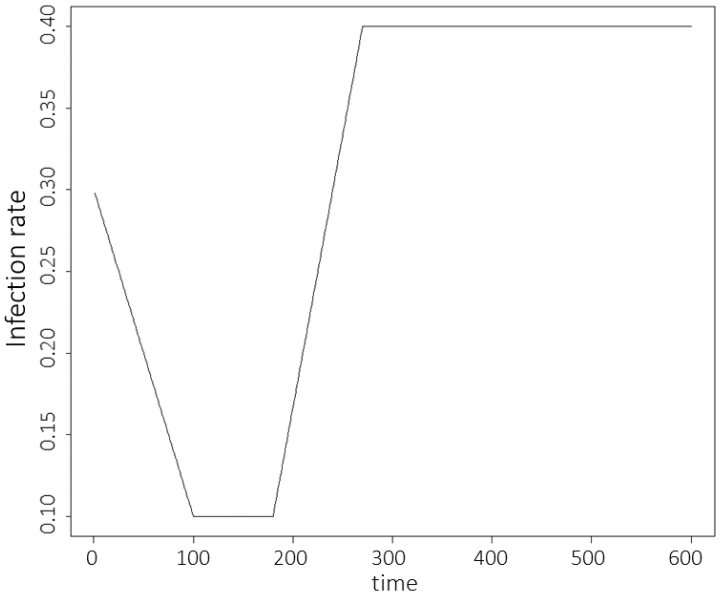
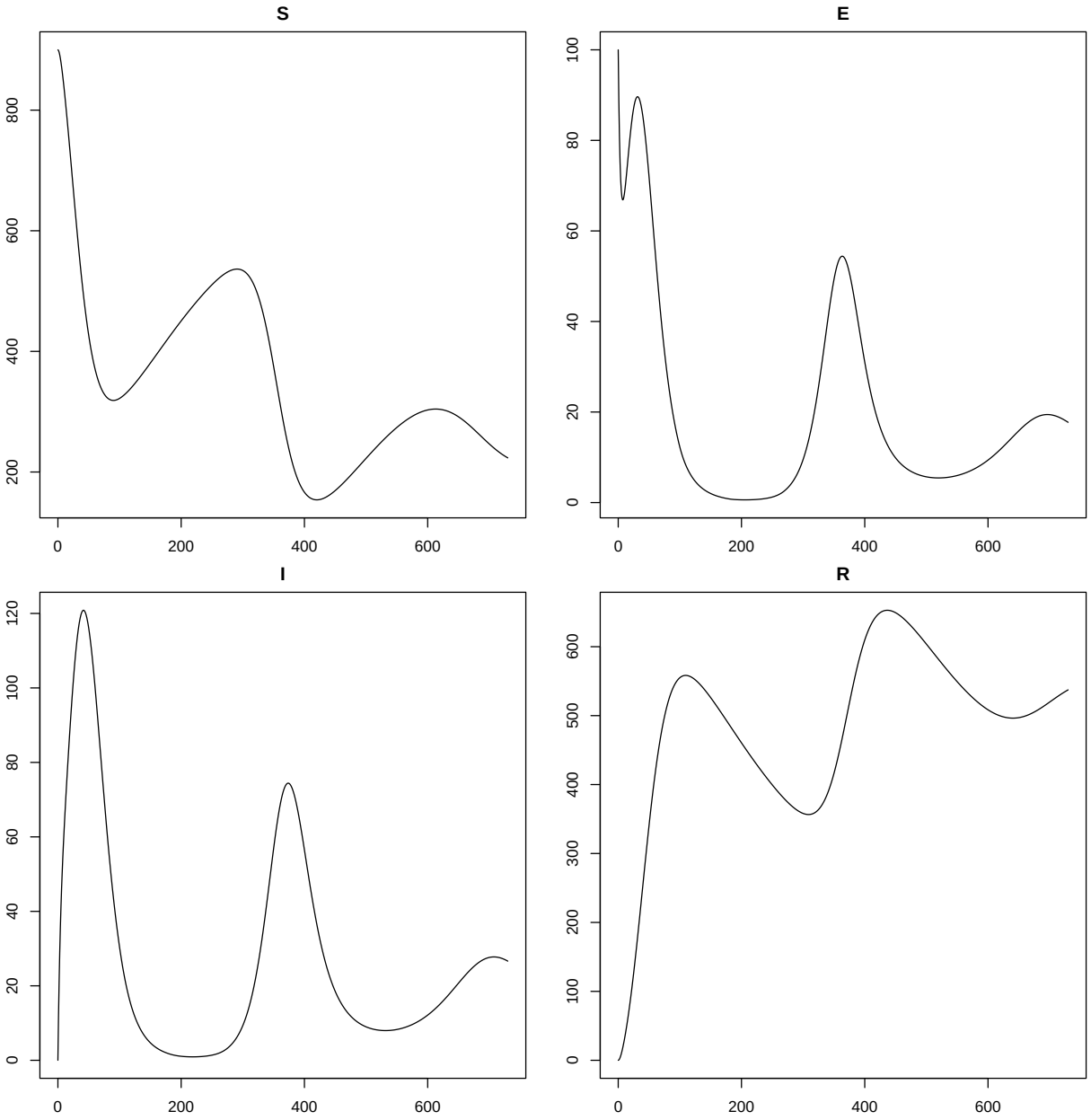


Figure 3.2: Infection rate graph



Task 1

Take some time to familiarise yourself with the model. Investigate how the plots change as you change the values of the parameters.

Show: R tip on P81

Show: Solution on P47

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.

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. Real-world observations, that inform our choice of targets, will instead be referred to as ‘observations.’ Before we tackle the emulation, we need to define some objects. First of all, let us set the parameter ranges:

```
ranges = list(
  b = c(1e-5, 1e-4), # birth rate
  mu = c(1e-5, 1e-4), # rate of death from other causes
  beta1 = c(0.2, 0.3), # infection rate at time t=0
  beta2 = c(0.1, 0.2), # infection rates at time t=100
  beta3 = c(0.3, 0.5), # infection rates at time t=270
  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 $t = 25, 40, 100, 200, 200, 350$. 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. Here we will use the former formulation (an example using the latter formulation can be found in Workshop 2):

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

```
  R100 = list(val = 538.99, sigma = 26.95),
  R200 = list(val = 444.23, sigma = 22.21),
  R300 = list(val = 371.08, sigma = 15.85),
  R350 = list(val = 549.42, sigma = 27.47)
)
```

The 'sigmas' in our `targets` list represent the uncertainty we have about the observations. Since our model is synthetic, we couldn't rely on observations to define our targets. Instead, we chose the parameter set

```
chosen_params <- list(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)
```

ran the model with it and used the relevant model outputs as the 'val' in `targets`. The 'sigma' components were chosen to be 5% of the corresponding 'val.'

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

- build a named dataframe `initial_points` containing a space filling set of points. A rule of thumb is to select at least $10p$ parameter sets for the training set, where p is the number of parameters in the model. The dataframe `initial_points` can also contain parameter sets for the validation set, even though hmer can also work without such a set, using cross-validation. The columns of `initial_points` should be named exactly as in the list `ranges`;
- run your model on the parameter sets in `initial_points` and define a dataframe `initial_results` in the following way: the nth row of `initial_results` should contain the model outputs corresponding to the chosen targets for the parameter set in the nth row of `initial_points`. The columns of `initial_results` should be named exactly as in the list `targets`;
- bind `initial_points` and `initial_results` by their columns to form the dataframe `wave0`.

For this workshop, we created a helper function, `space_filling_design`, that takes the number of parameter sets to generate, the number of parameters and the range of each parameter, and returns a space filling design with the required number of parameter sets. In this case, we request 180 parameter sets, 90 for the training set and 90 for the validation set.

```
initial_points <- space_filling_design(180, 9, ranges)
```

In `initial_points` each row corresponds to a different parameter set and each column to a different parameter.

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

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

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

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

Chapter 5

Emulators

In this section we train the first wave of emulators and explore them through various visualisations.

Let us start by splitting `wave0` in two parts: the training set (the first half), on which we will train the emulators, and a validation set (the second half), which will be used to conduct emulator diagnostics.

```
training <- wave0[1:90,]
validation <- wave0[91:180,]
```

5.1 Training emulators

We are now ready to train the emulators using the `emulator_from_data` function, which needs at least the following data: the training set, the names of the targets we want to emulate and the ranges of the parameters. By default, `emulator_from_data` assumes a square-exponential correlation function and finds suitable values for the variance σ and the correlation length θ of the process $u(x)$. In this workshop, in order to shorten the time needed to train emulators, we pass one more argument to `emulator_from_data`, setting the correlation length (a hyperparameter defining the covariance structure of the process $u(x)$) to be 0.55 for all emulators. Normally, the argument `c_lengths` will not be needed, since `emulator_from_data` will estimate them for us, based on the training data.

```
ems_wave1 <- emulator_from_data(training, names(targets), ranges,
                                c_lengths= rep(0.55,length(targets)),
                                verbose = FALSE)
```

```
## I25
## I40
## I100
## I200
## I300
## I350
## R25
## R40
## R100
## R200
## R300
## R350
## I25
```

```
## I40
## I100
## I200
## I300
## I350
## R25
## R40
## R100
## R200
## R300
## R350
```

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

```
ems_wave1$R200
```

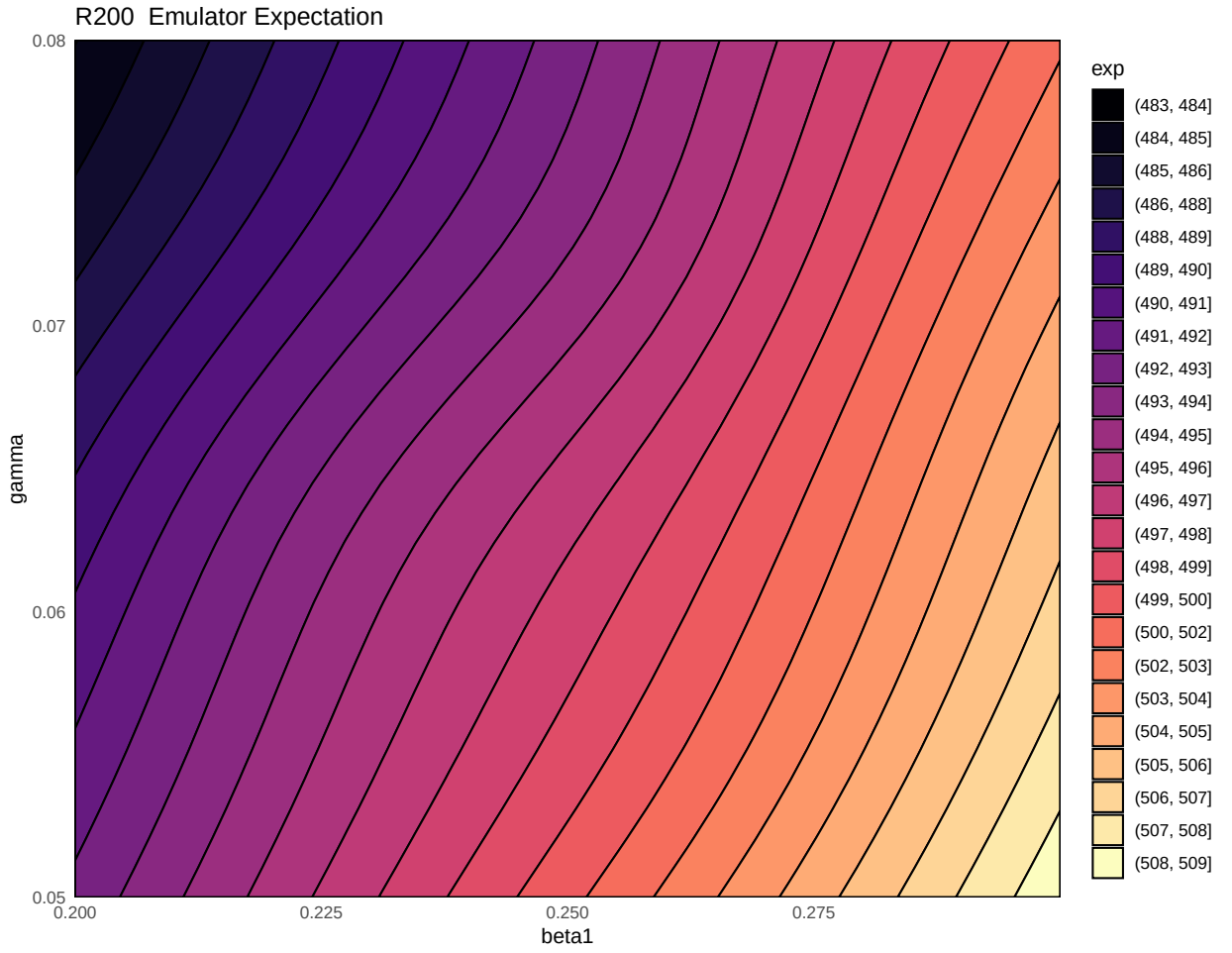
```
## Parameters and ranges:  b: c(0, 1e-04): mu: c(0, 1e-04): beta1: c(0.2, 0.3): beta2: c(0.1, 0.2): beta
## Specifications:
##   Basis functions:  (Intercept); b; mu; beta1; beta2; epsilon; alpha; gamma; omega
##   Active variables b; mu; beta1; beta2; epsilon; alpha; gamma; omega
##   Regression Surface Expectation:  496.7973; 2.1201; -3.1644; 8.4693; 23.29; -13.2785; -58.2566; -3.8
##   Regression surface Variance (eigenvalues):  0; 0; 0; 0; 0; 0; 0; 0; 0
## Correlation Structure:
## Bayes-adjusted emulator - prior specifications listed.
##   Variance (Representative):  81.04376
##   Expectation:  0
##   Correlation type: exp_sq
##   Hyperparameters:  theta: 0.55
##   Nugget term: 0.05
## Mixed covariance:  0 0 0 0 0 0 0 0 0
```

The print statement provides an overview of the emulator specifications, which refer to the global part, and correlation structure, which refers to the local part (see Section 2.3 for the distinction between the global and local part of an emulator):

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

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

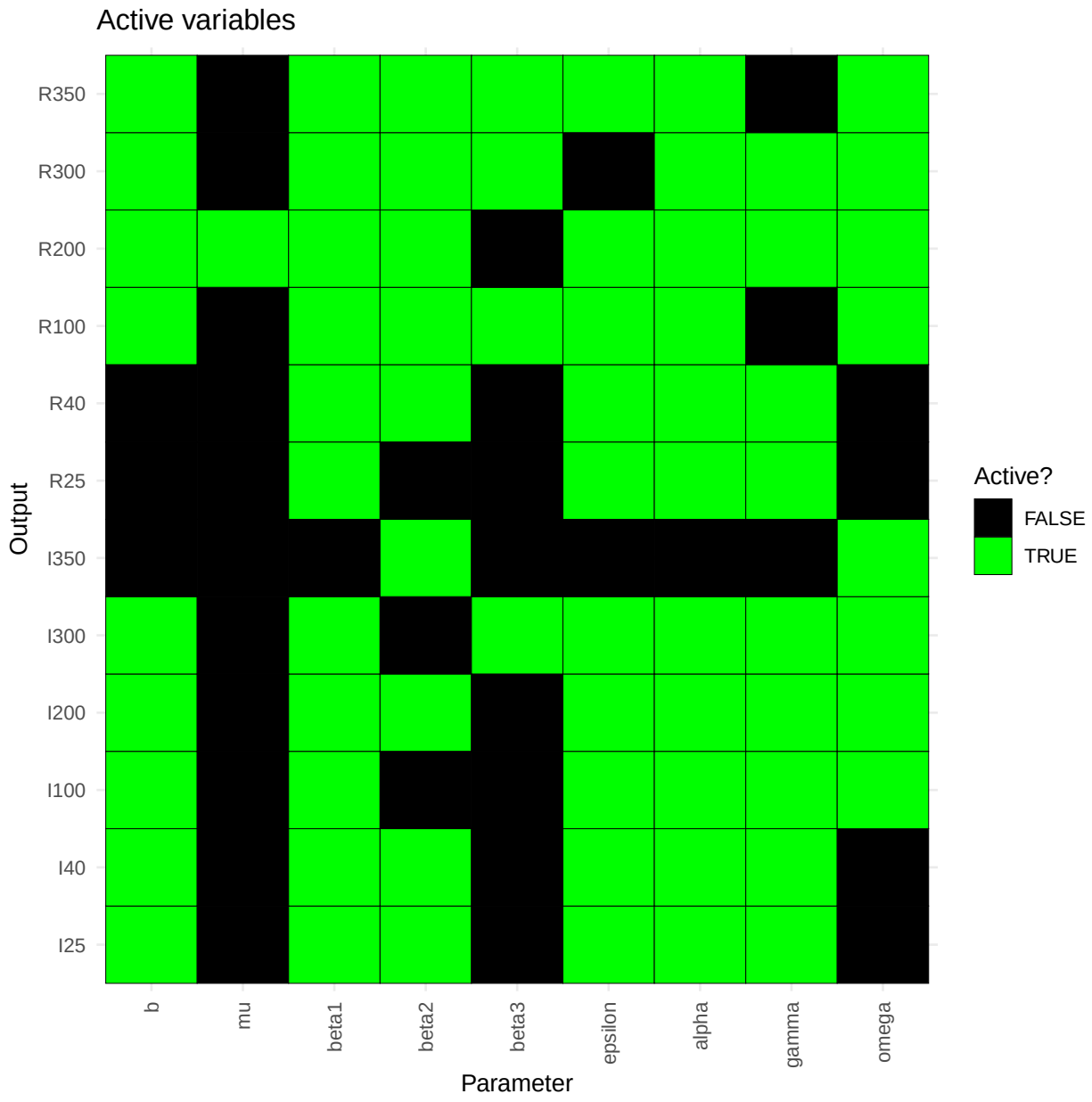
```
emulator_plot(ems_wave1$R200, params = c('beta1', 'gamma'))
```



Here for each pair $(\bar{\beta}_1, \bar{\gamma})$ the plot shows the expected value produced by the emulator `ems_wave1$R200` at the parameter set having $\beta_1 = \bar{\beta}_1$, $\gamma = \bar{\gamma}$ and all other parameters equal to their mid-range value (the ranges of parameters are those that were passed to `emulator_from_data` to train `ems_wave1`).

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

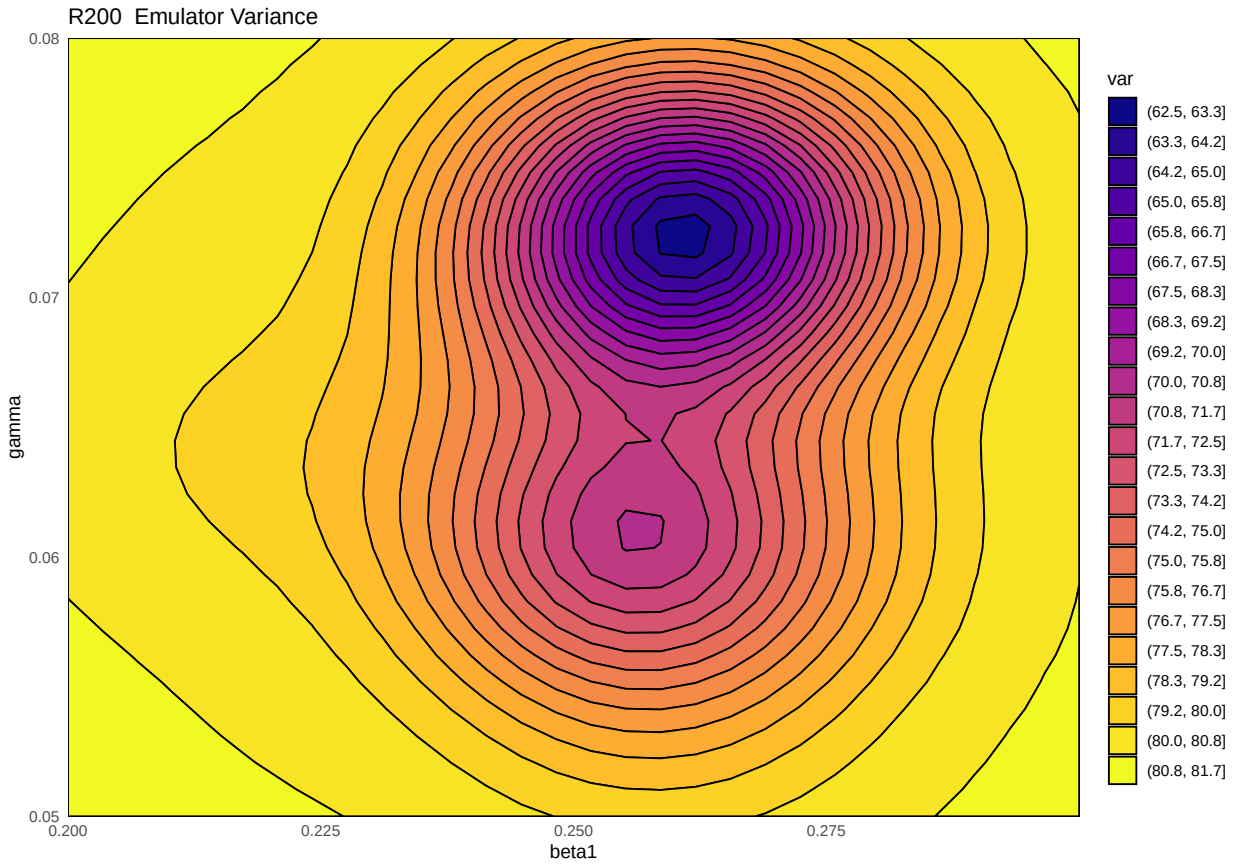
```
plot_actives(ems_wave1)
```



From this table, we can immediately see that mu is inactive for most outputs, while beta1, beta2, epsilon, alpha, gamma are active for most outputs. We also see that beta3 tends to be active for outputs at later times and inactive for outputs at earlier times, as one would expect. Note that beta3 is active for R100: even if this is not necessarily what we would have expected, this is not a problem. This is because emulators make use both of the regression hypersurface and of weakly stationary processes to account for differences between the model output and the regression surface. In particular, we should not expect the regression surface to be able to perfectly capture the model output's behaviour.

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

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



This plot shows the presence of a training point (purple-blue area on the right) close to the chosen slice of the input space. The purple-blue area indicates that the variance is low when we are close to the training point, which is in accordance with our expectation.

Chapter 6

Implausibility

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

For a given model output and a given target, the implausibility measures the difference between the emulator output and the target, taking into account all sources of uncertainty. For a parameter set x, the implausibility $\text{Imp}(x)$ takes the following form in this tutorial:

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

where $f(x)$ is the emulator output, z the target, and

- V_0 is the variance associated with the observation uncertainty (i.e. uncertainty in estimates from observed data);
- $V_c(x)$ refers to the uncertainty one introduces when using the emulator output instead of the model output itself. Note that this term depends on x, since the emulator is more/less certain about its predictions based on how close/far x is from training parameter sets.

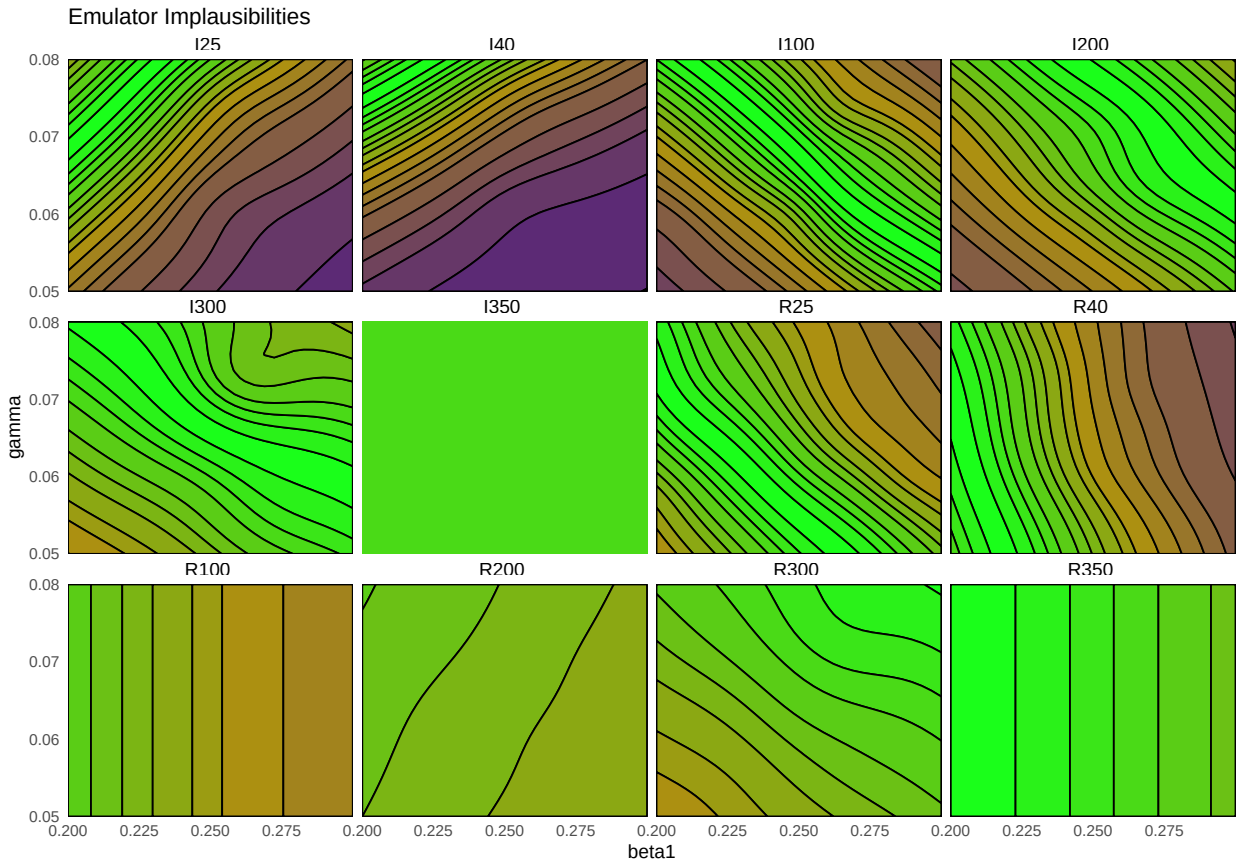
The observation uncertainty V_0 is represented by the 'sigma' values in the `targets` list, while V_c is the emulator variance, which we discussed in the previous section.

A very large value of $\text{Imp}(x)$ 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. When $\text{Imp}(x)$ is small, it could mean that the emulator output is very close to the model output or it could mean that the uncertainty in the denominator of $\text{Imp}(x)$ is large. In the former case, the emulator retains the parameter set, since it is likely to give a good fit to the observation for that output. In the latter case, the emulator does not have enough information to rule the parameter set out and therefore keeps it to explore further in the next wave.

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

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

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



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

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

$$\text{Imp}_M(x) = \max_{i=1,\dots,N} \text{Imp}_i(x),$$

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

Task 2

Explore the functionalities of `emulator_plot` and produce a variety of implausibility plots. Here are a few suggestions:

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

Show: Solution on P51

Chapter 7

Emulator diagnostics

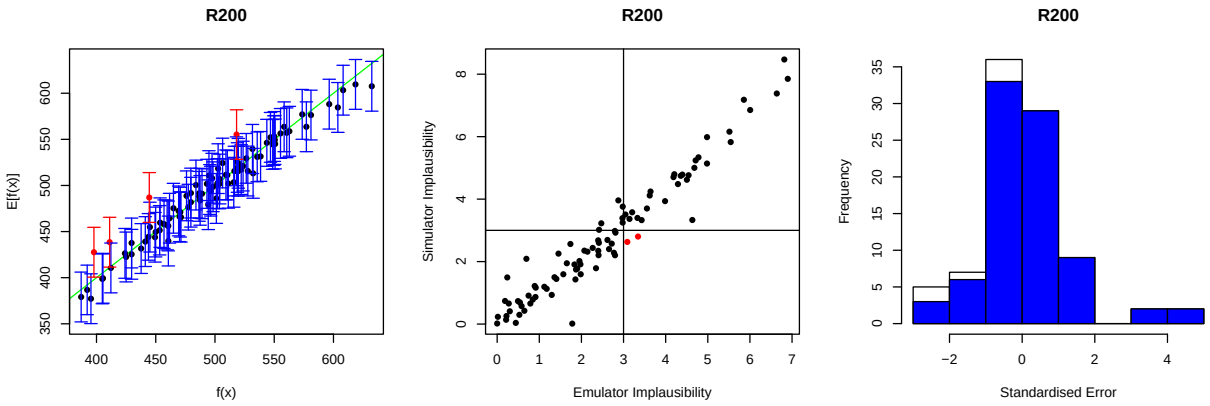
In this section we explore various diagnostic tests to evaluate the performance of the emulators and we learn how to address emulators that fail one or more of these diagnostics.

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

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

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

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



In the **first column**, the emulator expectation $E[f(x)]$ is plotted against the model output $f(x)$ 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 emulated prediction lies more than 3σ away from the model output is highlighted in red. Note that we do not need to have no red points for the test to be passed: since we are plotting 3σ bounds, statistically speaking it is ok to have up to 5% of validation points in red (see Pukelsheim's 3σ rule).

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.

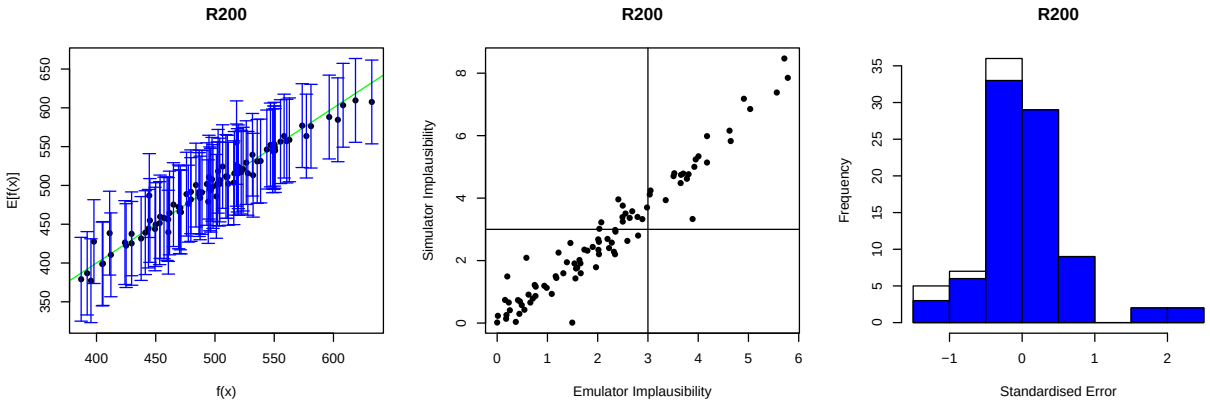
Finally, **the third column** gives the standardised errors of the emulator outputs in light of the model output: for each validation point, the difference between the emulator output and the model output is calculated, and then divided by the standard deviation σ of the emulator at the point. The general rule is that we want our standardised errors to be somewhat normally distributed around 0, with 95% of the probability mass between -3 and 3. The blue bars indicate the distribution of the standardised errors when we restrict our attention only to parameter sets that produce outputs close to the targets. 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 $[-3, 3]$? 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 $[-3, 3]$ (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.

The diagnostics above are not particularly bad, but we will try to modify our emulator to make it more conservative and avoid misclassifications in the bottom right quadrant (second column).

A way of improving the performance of an emulator is by changing its variance σ^2. The lower the value of σ, the more 'certain' the emulator will be. This means that when an emulator is a little too overconfident (as in our case above), we can try increasing σ. Below we train a new emulator setting σ to be 2 times as much as its default value, through the method `mult_sigma`:

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



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

Task 3

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

Show: R tip on P81

Show: Solution on P55

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[[j]], targets, validation, plt = FALSE))
      while(misclass > 0) {
        ems[[j]] <- ems[[j]]$mult_sigma(1.1)
        misclass <- nrow(classification_diag(ems[[j]], targets, validation, plt = FALSE))
      }
}
```

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[[j]], targets, validation, plt = FALSE))
          if (bad.model > floor(nrow(validation)/10)) {
            bad.ems <- c(bad.ems, j)
   }
}
ems <- ems[!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

In this section we generate the set of points that will be used to train the second wave of emulators. Through visualisations offered by hmer, we then explore these points and assess how much of the initial input space has been removed by the first wave of emulators.

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

STEP 1. A set of parameter sets is generated using a Latin Hypercube Design, rejecting implausible parameter sets.

STEP 2. Pairs of parameter sets are selected at random and more sets are sampled from lines connecting them, with particular importance given to those that are close to the non-implausible boundary.

STEP 3. Using these as seeding points, more parameter sets are generated using importance sampling to attempt to fully cover the non-implausible region.

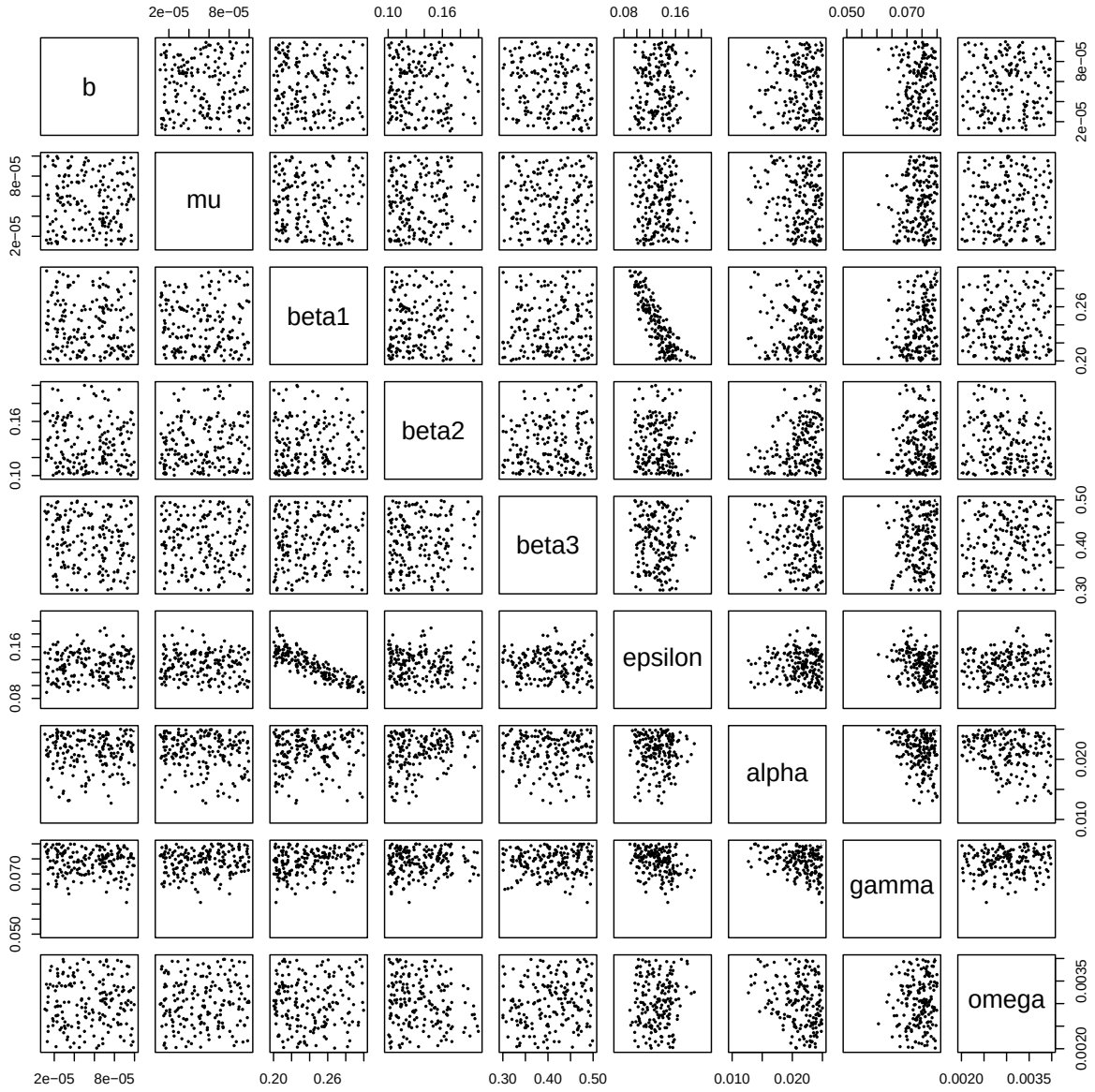
Let us generate 180 new sets of parameters, using the emulators for the time up to $t = 200$. Focusing on early time outputs can be useful when performing the first waves of history matching, since the behaviour of the epidemic at later times will depend on the behaviour at earlier times. Note that the generation of new points might require a few minutes.

```
restricted_ems <- ems_wave1[c(1,2,3,4,7,8,9,10)]
new_points_restricted <- generate_new_runs(restricted_ems, 180, targets, verbose=TRUE)
```

```
## Proposing from LHS...
## LHS has high yield - no other methods required.
## Selecting final points using maximin criterion...
```

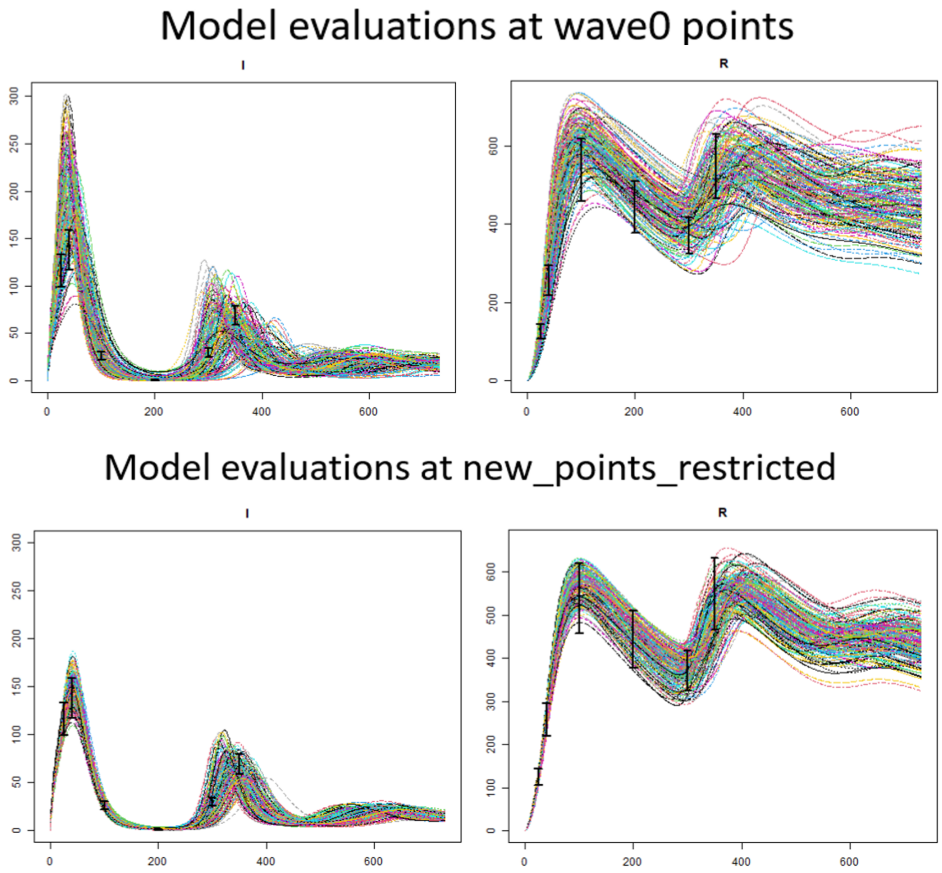
We now plot `new_points_restricted` through `plot_wrap`. Note that we pass `ranges` to `plot_wrap` as well, to ensure that the plot shows the entire range for each parameter: this allows us to see how the new set of parameters compares with respect to the original input space.

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



By looking at the plot we can learn a lot about the non-implausible space. For example, it seems clear that low values of γ cannot produce a match (cf. 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.

Now that we have generated a new set of points, we can compare the model output at points in `initial_points` with the model output at points in `new_points_restricted`



We can clearly see that the range of possible results obtained when evaluating the model at `wave0` points is larger than the range obtained when evaluating the model at points in `new_points_restricted`. This is because we have performed a wave of the history matching process, discarding part of the initial input space that is not compatible with the targets.

Task 4

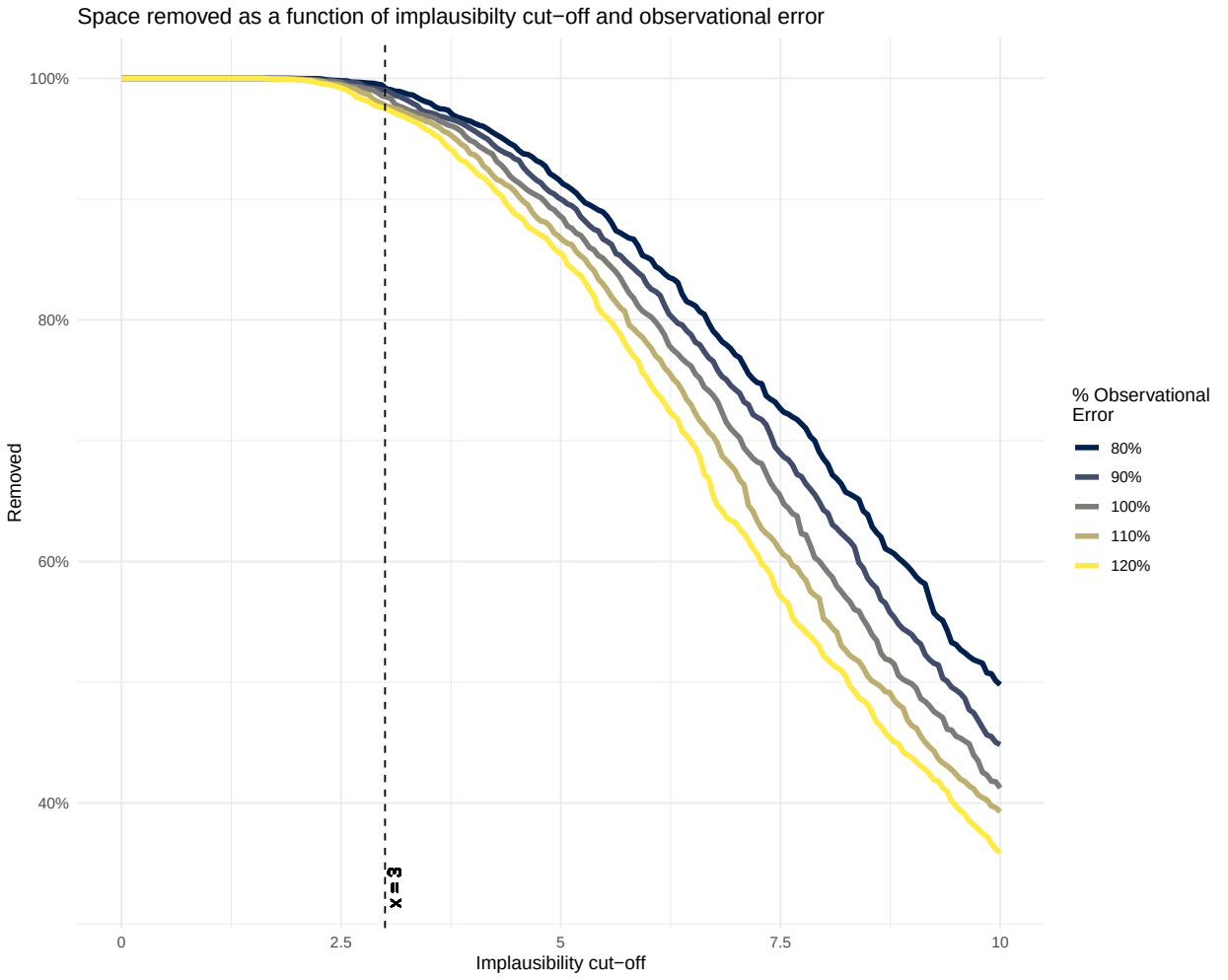
What do you think would happen if we tried generating new parameter sets using all emulators, instead of those relating to times up to $t = 200$ only? Try changing the code and seeing what happens.

Show: Solution on P57

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

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

```
## Warning in geom_text(aes(x = 3, label = "x = 3", y = 0.33), colour = "black", :
## Ignoring unknown parameters: 'text'
```



By default the plot shows the percentage of space that is deemed implausible both when the observational errors are exactly the ones in `targets` and when the observational errors are 80% (resp. 90%, 110% and 120%) of the values in `targets`. Here we see that with an implausibility cut-off of 3, the percentage of space removed is around 98%, when the observational errors are 100% of the values in `targets`. With the same implausibility cut-off of 3, the percentage of space removed goes down to 97%, when the observational errors are 120% of the values in `targets` and it goes up to 99% when the observational errors are 80% of the values in `targets`. If instead we use an implausibility cut-off of 5, we would discard around 88% (resp. 84%) of the space when the observational errors are 100% of the values in `targets` (resp. when the observational errors are 120% of the values in `targets`). As expected, larger observational errors and larger implausibility cut-offs correspond to lower percentages of space removed.

Chapter 9

Customise the first wave

This section consists of just one task: it is time to put all you have learnt so far to good use!

Task 5

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

Show: Solution on P58

Chapter 10

Second wave

In this section we move to the second wave of emulation. We will start by defining all the data necessary to train the second wave of emulators. We will then go through the same steps as in the previous section to train the emulators, test them and generate new points.

To perform a second wave of history matching and emulation we follow the same procedure as in the previous sections, with two caveats. We start by forming a dataframe `wave1` using parameters sets in `new_points`, as we did with `wave0`, i.e. we run the model at `new_points` using the function `get_results` and then bind the obtained outputs to `new_points`. Half of `wave1` should be used as the training set for the new emulators, and the other half as the validation set to evaluate the new emulators' performance.

Now note that parameter sets in `new_points` tend to lie in a small region inside the original input space, since `new_points` contains only non-implausible points, according to the first wave emulators. The first caveat is then that it is preferable to train the new emulators only on the non-implausible region found in wave one. This can be done by setting the argument `check.ranges` to `TRUE` in the function `emulator_from_data`. The second caveat is that emulators from both the first and second wave need to be passed to the `generate_new_runs` function: this is because second wave emulators tend to be rather uncertain outside of the small region of the input space where they were trained.

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

Task 6

Using `new_points` and setting `check.ranges` to `TRUE`, train new emulators. Customise them and generate new parameter sets.

Show: Solution on P66

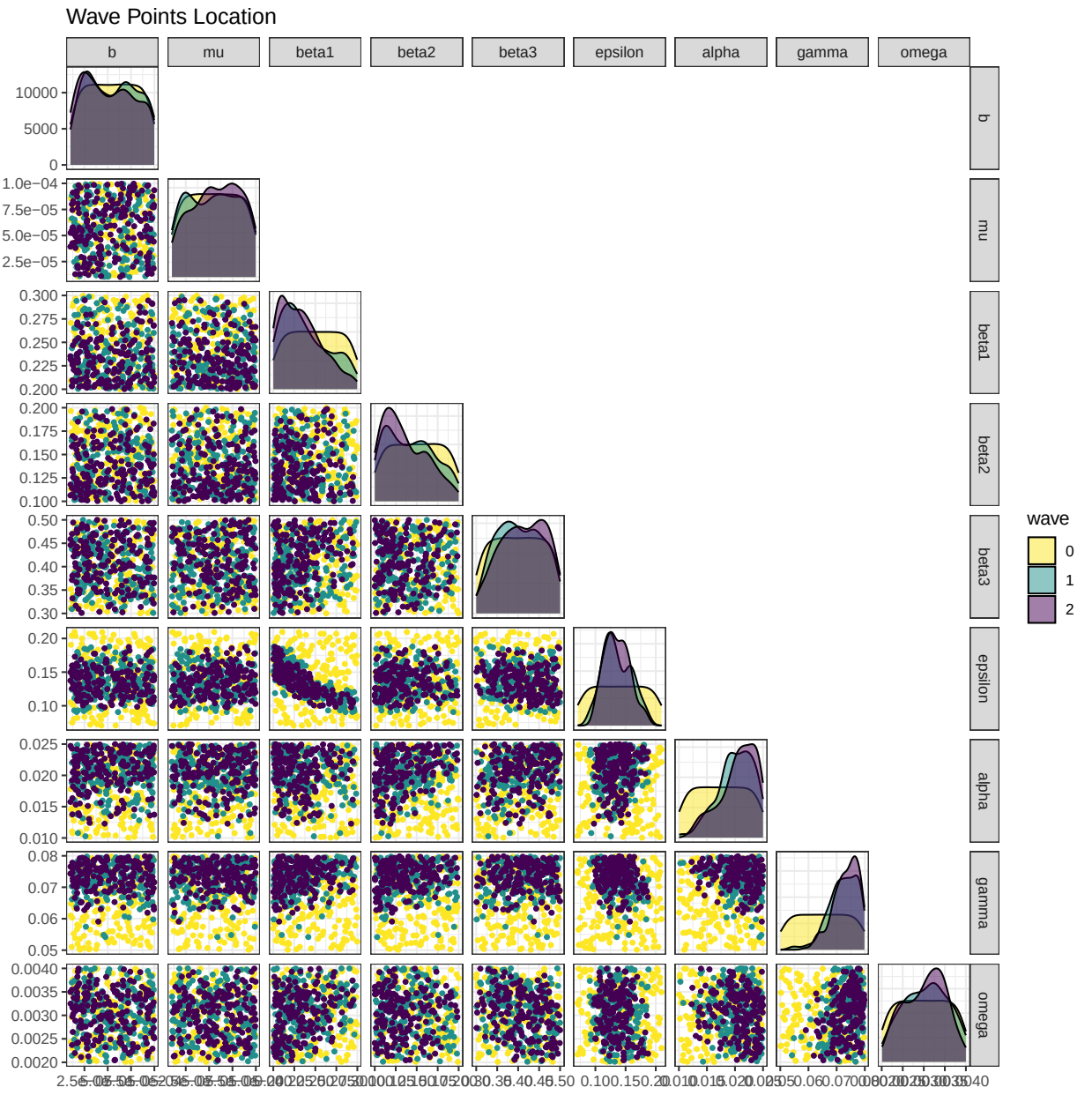
Chapter 11

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

```
wave_points(list(initial_points, new_points, new_new_points),
            input_names = names(ranges), p_size=1)
```



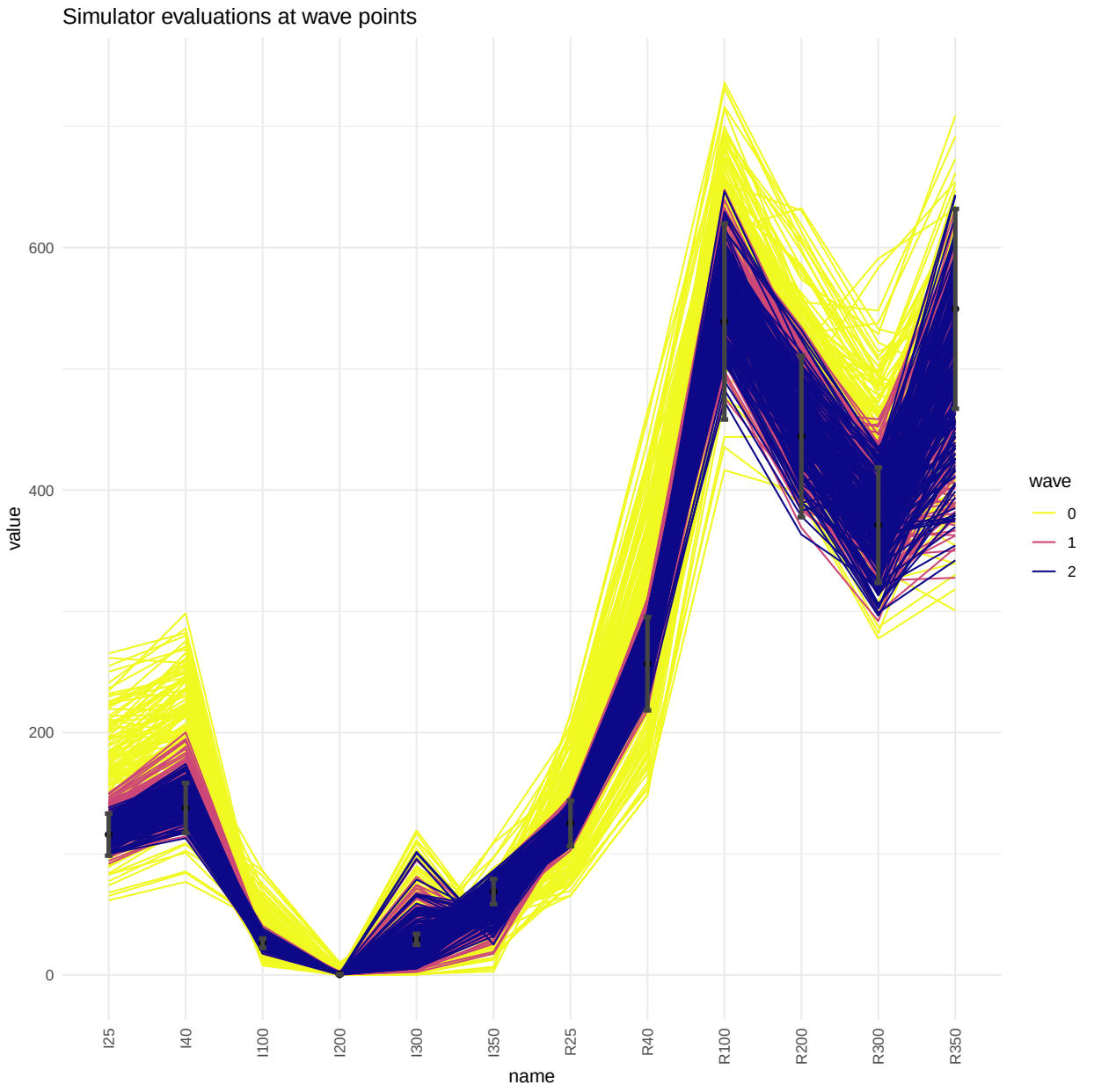
Here `initial_points` are in yellow, `new_points` are in green and `new_new_points` are in purple. We set `p_size` to 1 to make the size of the points in the plot smaller. The plots in the main diagonal show the distribution of each parameter individually: we can easily see that the distributions tend to become narrower each wave. In the off-diagonal boxes we have plots for all possible pairs of parameters. Again, the non-implausible region identified at the end of each wave clearly becomes smaller and smaller.

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

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

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

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



We can see that, compared to the space-filling random parameter sets used to train the first emulators, the new parameter sets are in much closer agreement with our targets. While there wasn't a single target matched by all parameter sets in wave zero, we have several targets matched by all parameter sets in wave 2 (I25, R25, R40, R100, R200). Subsequent waves, trained on the new parameter sets, will be more confident in the new non-implausible region: this will allow them to refine the region and increase the number of targets met.

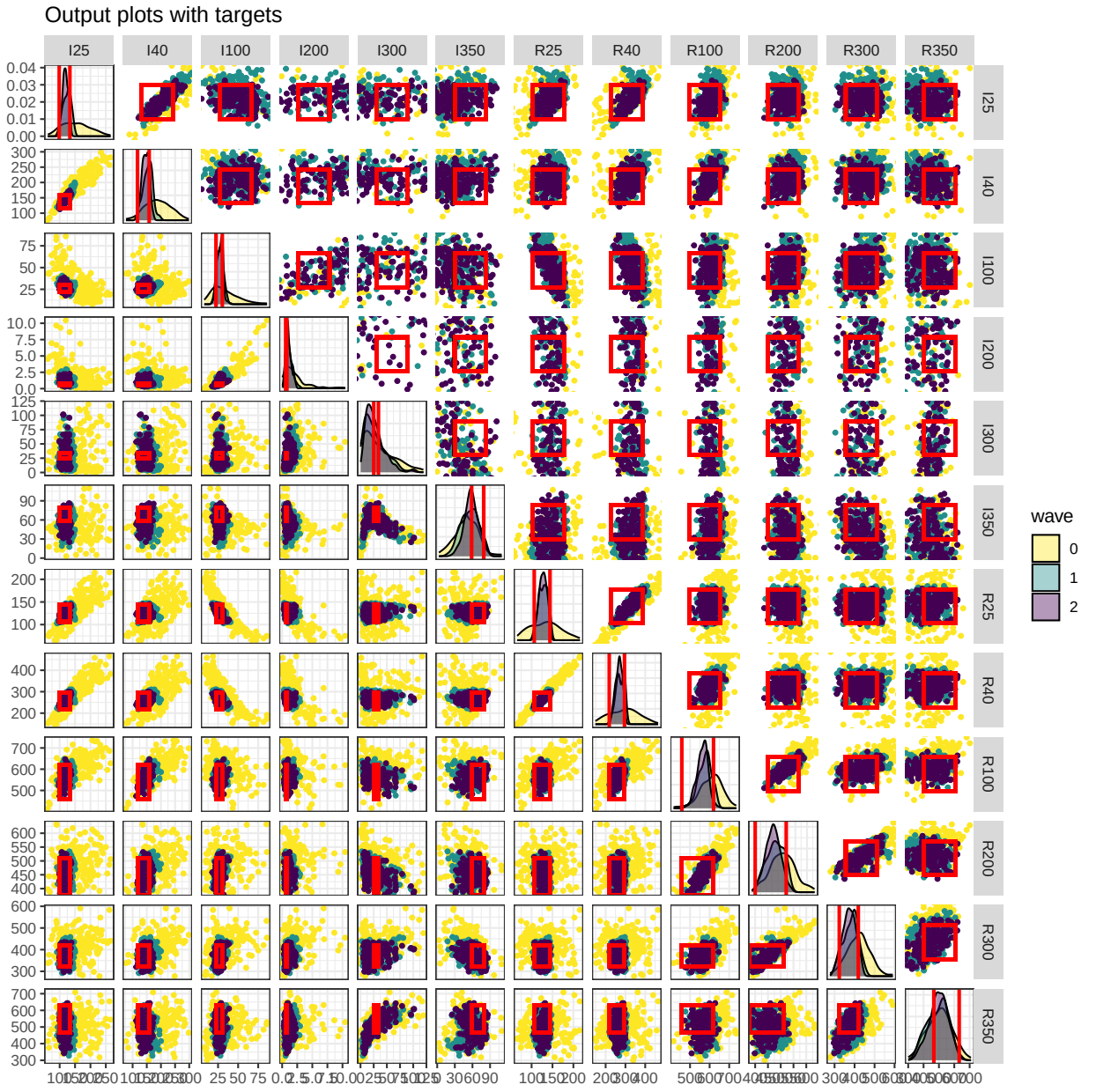
Task 7

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

Show: Solution on P75

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_points, targets, l_wid=1, p_size=1)
```



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. The argument `l_wid` is optional and helps customise the width of the red lines that create the target boxes, and `p_size` let us choose the size of the points.

In this workshop, we have shown how to perform the first two waves of the history matching process, using the hmer package. 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 it is smaller, training new 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 simply use the current emulators to generate points 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 structure, or input and/or output ranges.

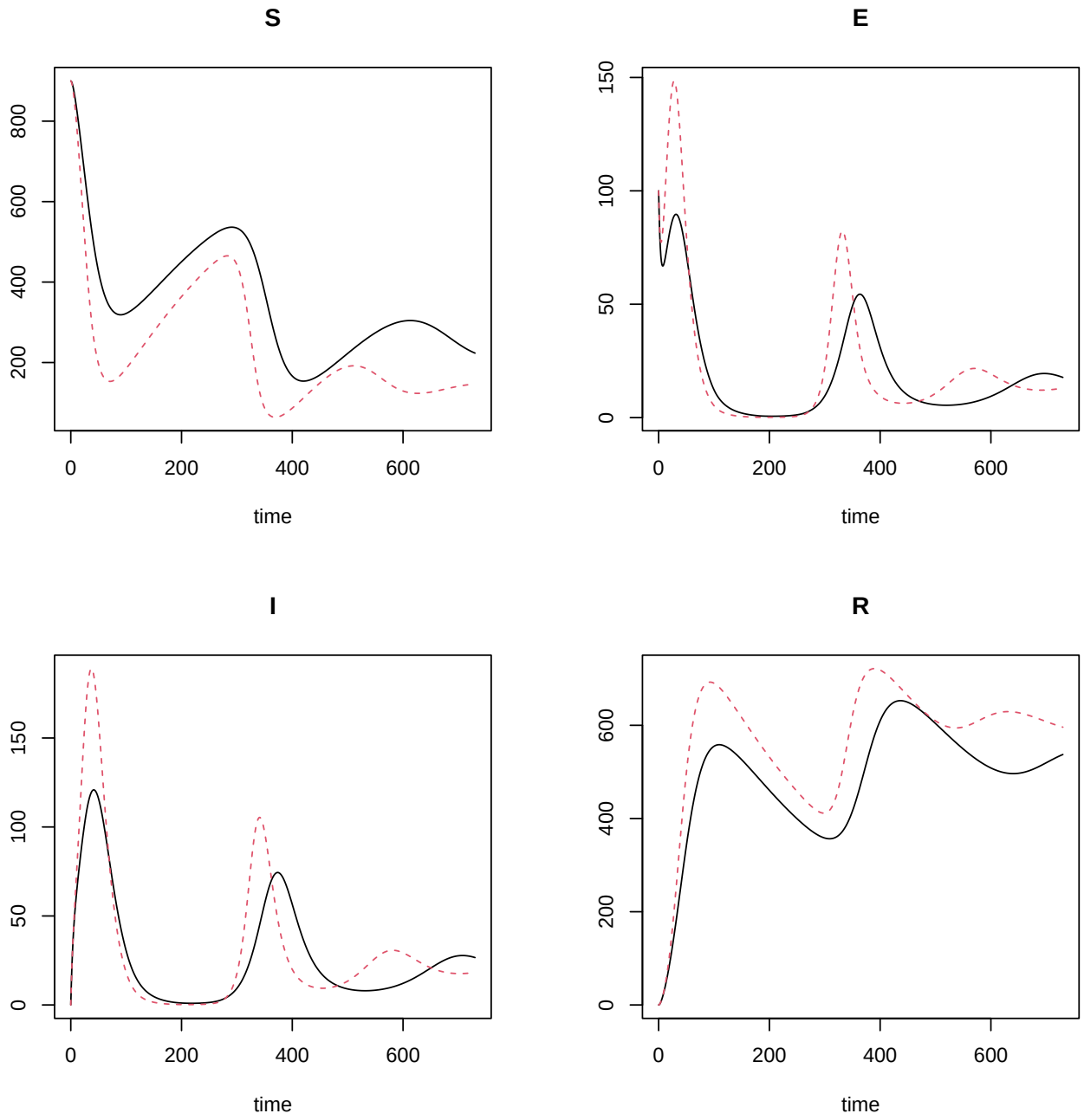
Appendix A

Answers

Solution 1

Let us see what happens when a higher force of infection is considered:

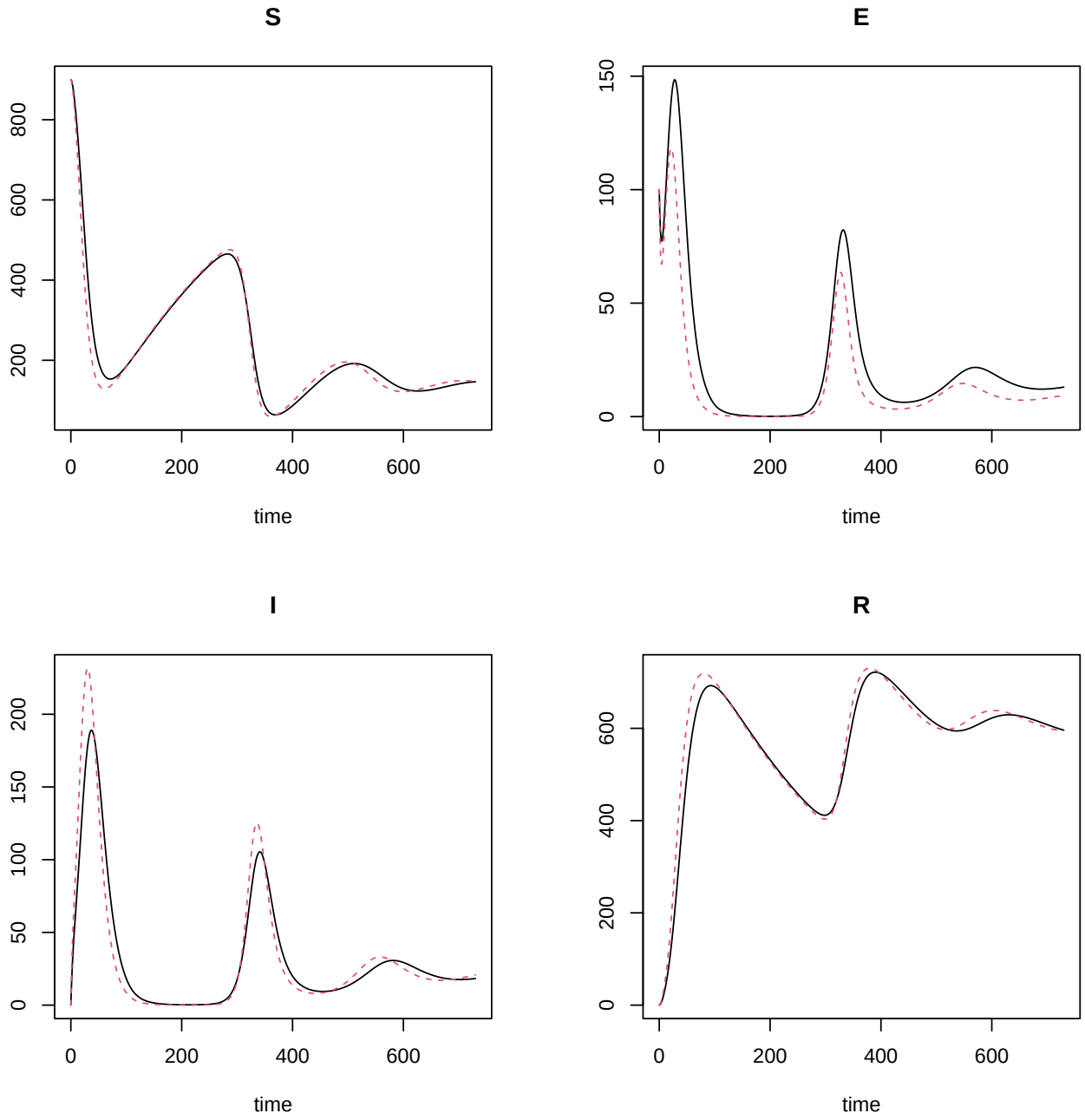
```
higher_foi_params <- c(
  b = 1/(60*365),
  mu = 1/(76*365),
  beta1 = 0.3, beta2 = 0.1, beta3 = 0.5,
  epsilon = 0.13,
  alpha = 0.01,
  gamma = 0.08,
  omega = 0.003
)
higher_foi_solution <- ode_results(higher_foi_params)
plot(solution, higher_foi_solution)
```



Here the black line shows the model output when it is run using the original parameters and the red dotted line when it is run using a higher force of infection. As expected, the number of susceptible individuals decreases, while the size of outbreaks increases.

Let us now also increase the rate of becoming infectious following infection ϵ:

```
higher_epsilon_params <- c(
  b = 1/(60*365),
  mu = 1/(76*365),
  beta1 = 0.3, beta2 = 0.1, beta3 = 0.5,
  epsilon = 0.21,
  alpha = 0.01,
  gamma = 0.08,
  omega = 0.003
)
higher_epsilon_solution <- ode_results(higher_epsilon_params)
plot(higher_foi_solution,higher_epsilon_solution)
```

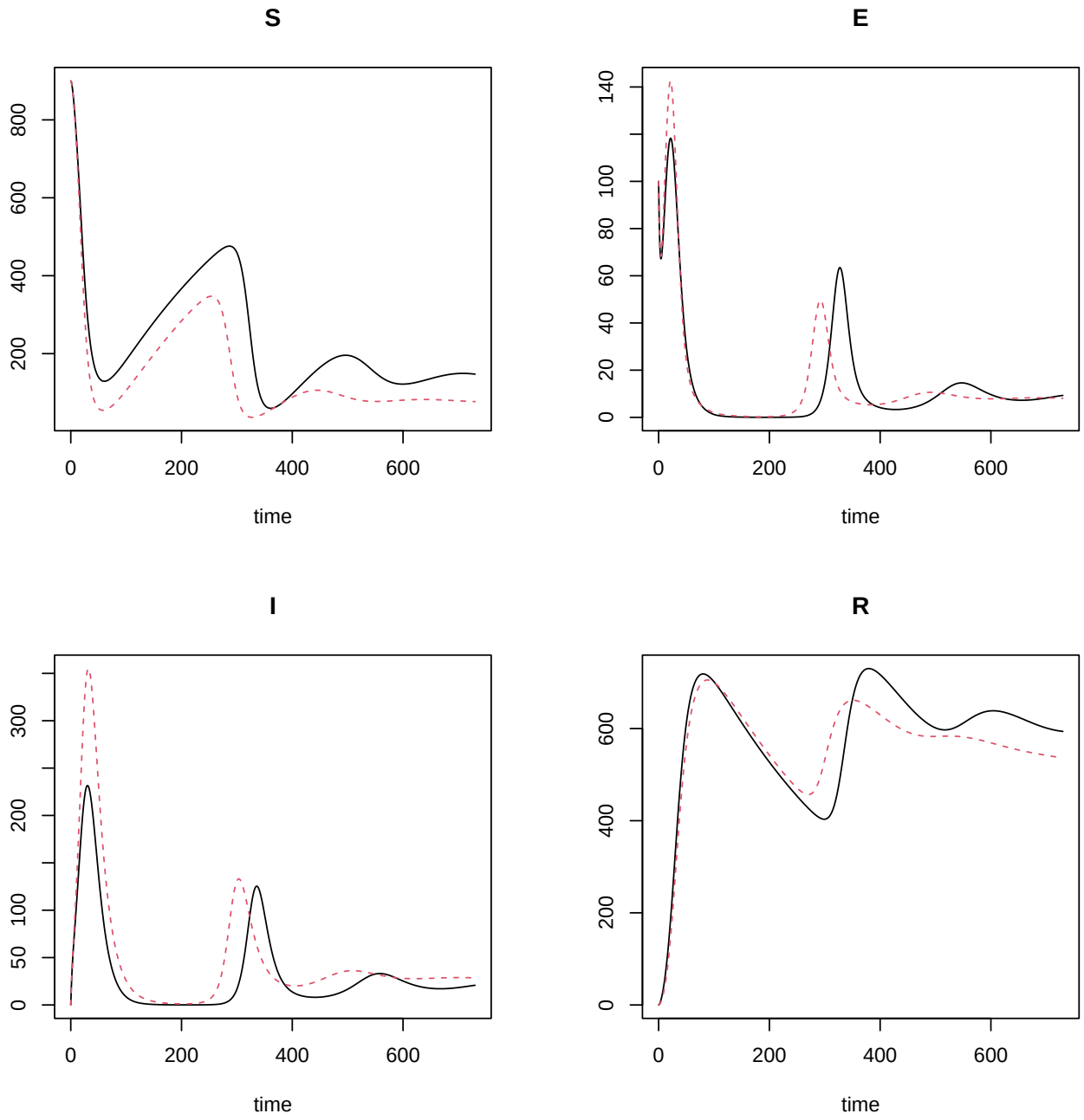


Here the black line is the model output when the model is run with the previous parameter set, and the red dotted line is the model output when we also increase epsilon. We observe a decrease in the number of exposed individuals.

Again, this is in agreement with our expectation: a higher rate of becoming infectious means that people leave the exposed compartmental to enter the infectious compartment faster than before.

Finally, what happens when a lower value of the recovery rate γ is used?

```
smaller_recovery_params <- c(
  b = 1/(60*365),
  mu = 1/(76*365),
  beta1 = 0.3, beta2 = 0.1, beta3 = 0.5,
  epsilon = 0.21,
  alpha = 0.01,
  gamma = 0.05,
  omega = 0.003
)
smaller_recovery_solution <- ode_results(smaller_recovery_params)
plot(higher_epsilon_solution, smaller_recovery_solution)
```



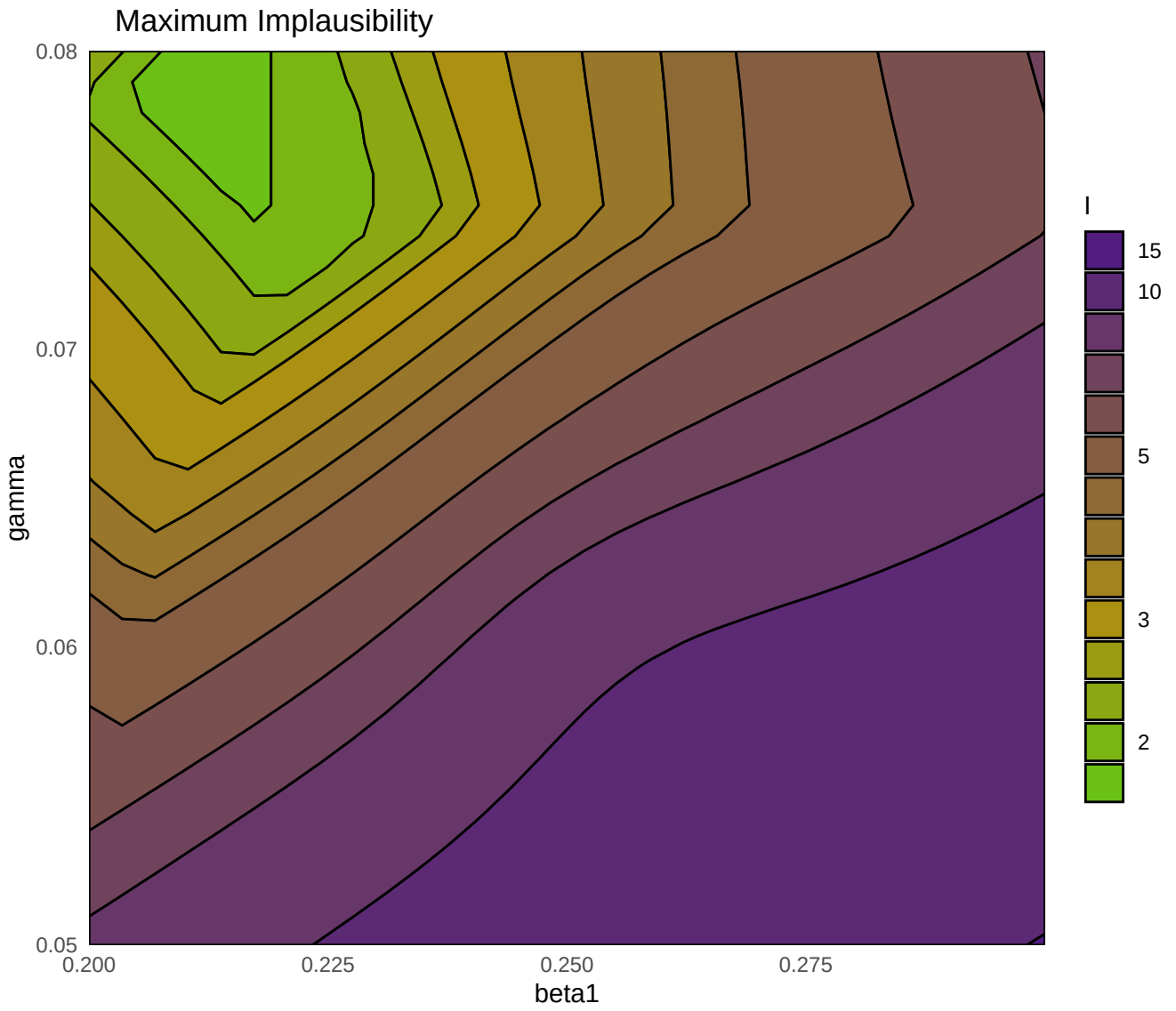
Here the black line is the model output when the model is run with the previous parameter set, and the red dotted line is the model output when we also decrease the recovery rate. Again, as one expects, this causes the number of susceptible individuals to decrease and the number of infectious individuals to increase, at least during the first peak.

Return to task on P14

Solution 2

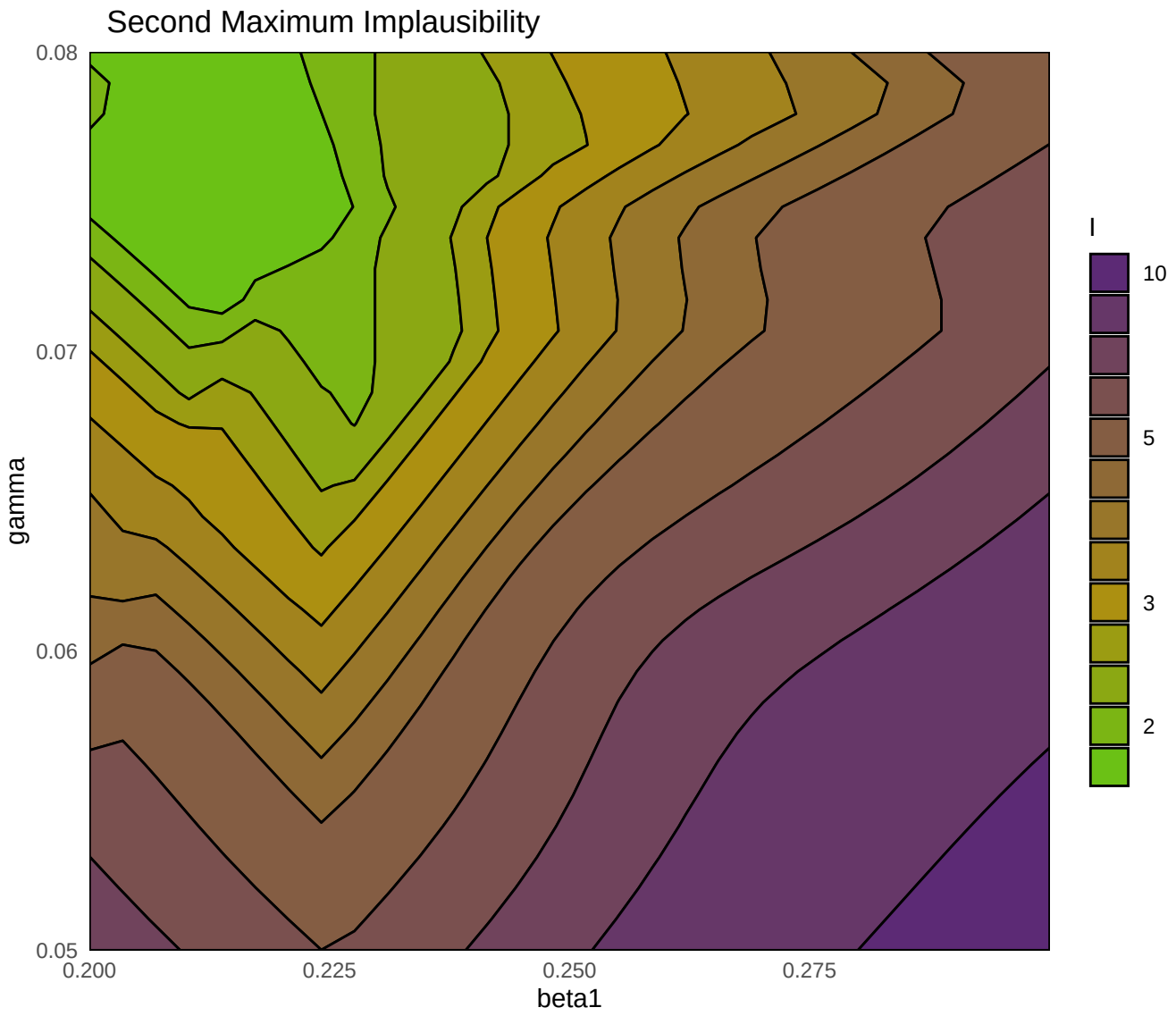
Let us start by visualising the maximum implausibility passing all emulators to `emulator_plot` and setting `plot_type='nimp'`:

```
emulator_plot(ems_wave1, plot_type = 'nimp', targets = targets,
              params = c('beta1', 'gamma'), cb=T)
```



This plot shows very high values of the implausibility for most points in the box. During the first few waves of history matching, one can consider second-maximum implausibility, rather than maximum implausibility. This means that instead of requiring the implausibility measure to be under the chosen threshold for all outputs, we allow (at most) one of them to be over it. This approach, which may result in less space cut out during the first few waves, has the advantage of being more conservative, reducing the risk that parts of the input space may be incorrectly cut. The more strict maximum implausibility measure can then be adopted in later waves, when the space to search is considerably smaller than the original input space, and the emulators less uncertain. To work with second-maximum implausibility we simply add nth=2 to the previous function call:

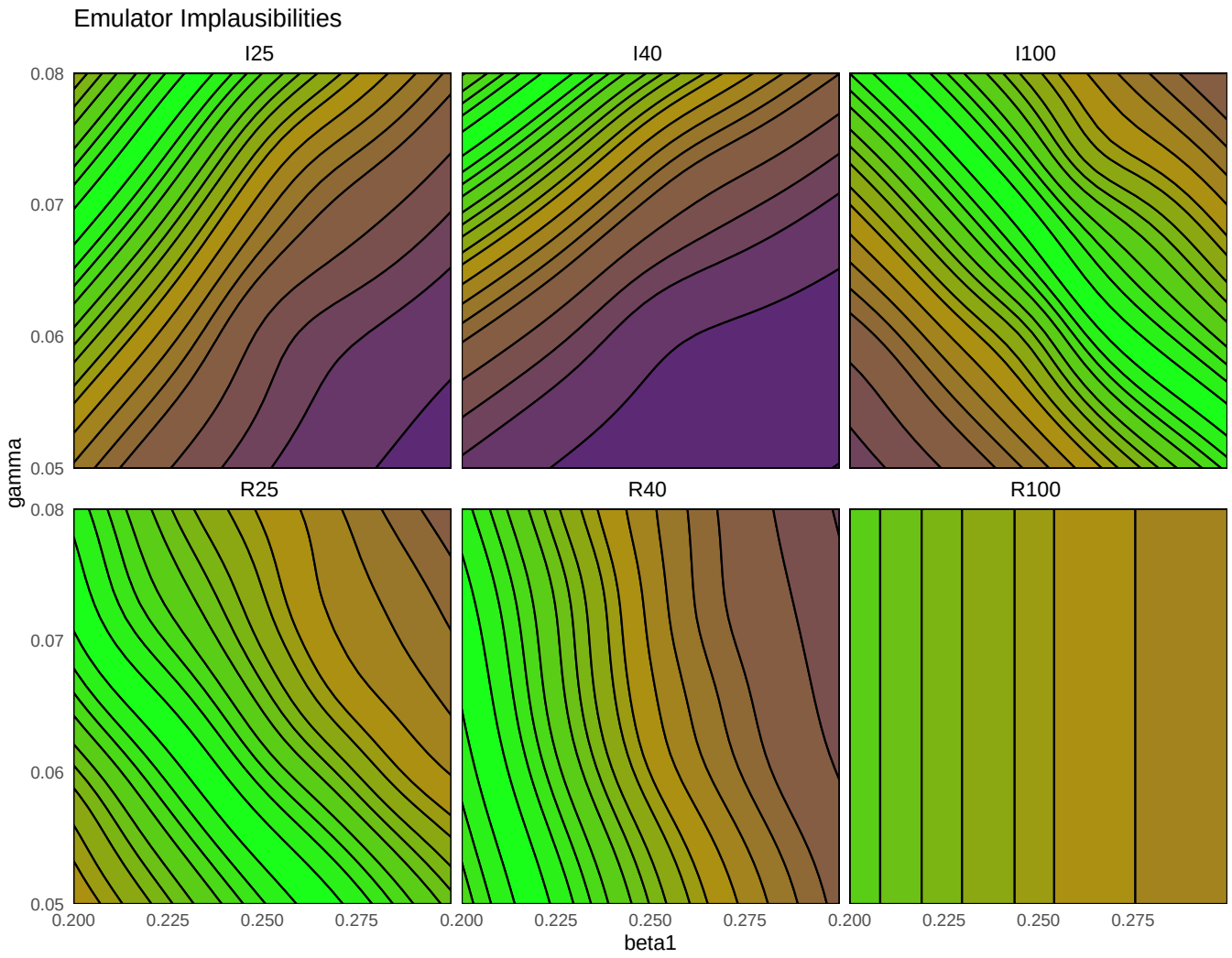
```
emulator_plot(ems_wave1, plot_type = 'nimp', targets = targets,
              params = c('beta1', 'gamma'), cb=T, nth=2)
```



One of the advantages of history matching and emulation is that we are not obliged to emulate all outputs at each wave. This flexibility can be useful, for example, when the emulator for a given output does not perform well at a certain wave: we can simply exclude that output and emulate it in later waves. Another common situation where it may be useful to select a subset of emulators is when we have early outputs and late outputs, as in this workshop. It is often the case that later outputs have greater variability compared to earlier outputs, since they have more time to diverge. As a consequence, including emulators for later outputs in the first few waves may not be desirable: it would both increase the number of calculations to make (since we would train more emulators), and would probably contribute to a lesser extent to the reduction of the parameter space.

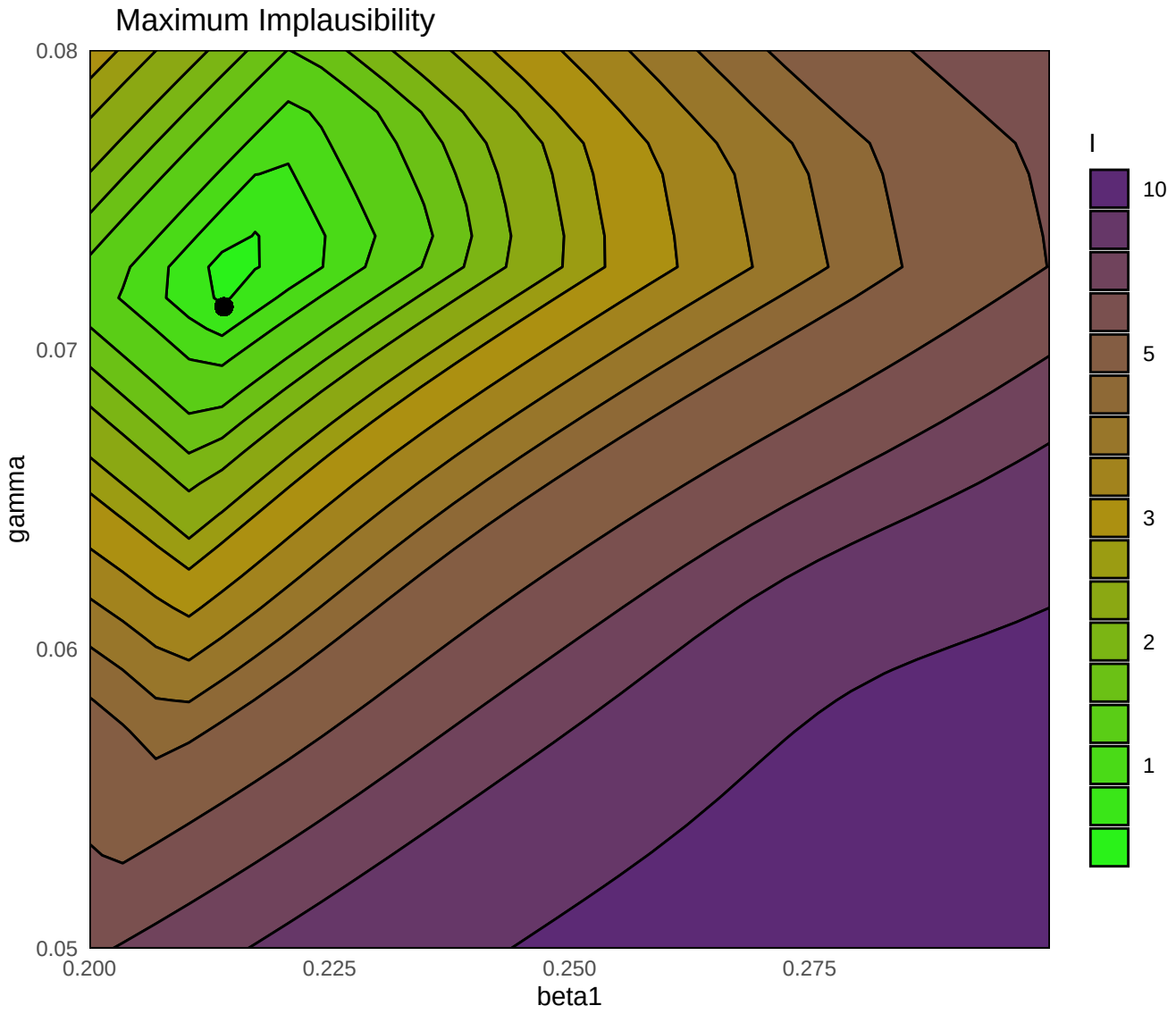
We can focus on early times outputs (e.g. up to $t = 100$), and produce implausibility plots for them:

```
restricted_ems <- ems_wave1[c(1,2,3,7,8,9)]
emulator_plot(restricted_ems, plot_type = 'imp', targets = targets,
              params = c('beta1', 'gamma'), cb=T)
```



Finally let us set the unshown parameters to be as in `chosen_params` (the parameter set used to define the targets, see 4):

```
emulator_plot(restricted_ems, plot_type = 'nimp', targets = targets[c(1,2,3,7,8,9)],
              params = c('beta1', 'gamma'),
              fixed_vals = chosen_params[!names(chosen_params) %in% c('beta1', 'gamma')],
              cb=T)+geom_point(aes(x=0.214, y=1/14), size=3)
```



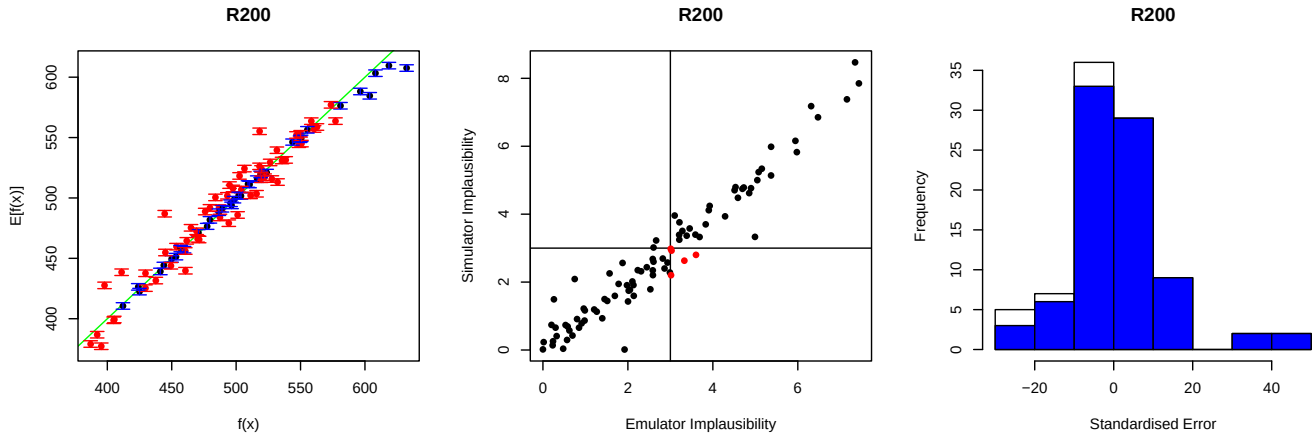
The plot shows what we expected: when β_1 and γ are equal to their values in `chosen_params`, i.e. 0.214 and $1/14$, the implausibility measure is well below the threshold 3 (cf. black point in the box).

Return to task on P27

Solution 3

Let us set σ to be ten times smaller than its default value:

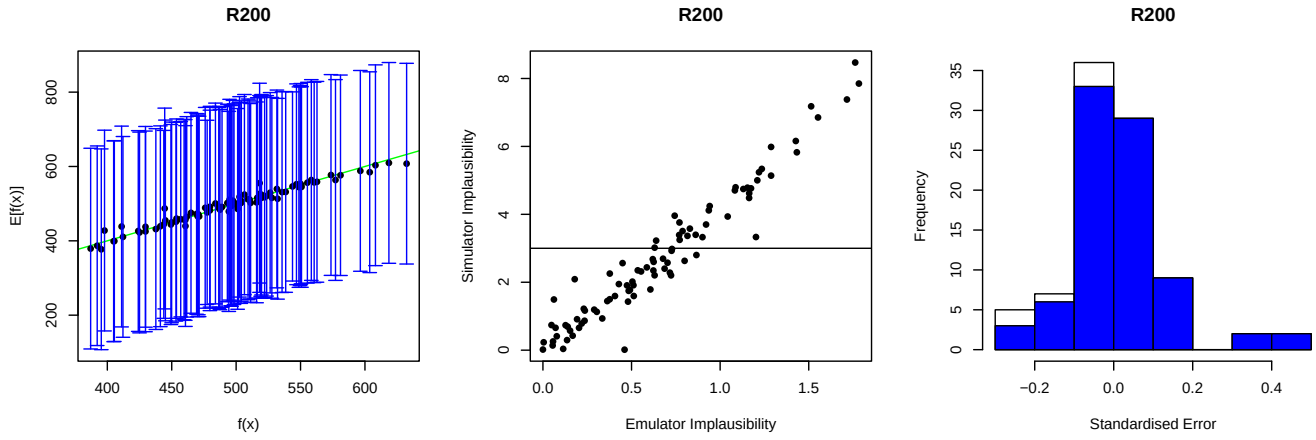
```
tinysigma_emulator <- ems_wave1$R200$mult_sigma(0.1)
vd <- validation_diagnostics(tinysigma_emulator, validation = validation,
                             targets = targets, plt=TRUE)
```



In this case we built a very overconfident emulator. This is shown by the very small uncertainty intervals in the first column: as a consequence many points are in red. Similarly, if we look at the third column we notice that the standardised errors are extremely large, well beyond the value of 3.

Let us now set σ to be ten times larger than its default value:

```
hugesigma_emulator <- ems_wave1$R200$mult_sigma(10)
vd <- validation_diagnostics(hugesigma_emulator, validation = validation, targets = targets,
                             plt=TRUE)
```



With this choice of σ, we see that our emulator is extremely cautious. If we look at the plot in the middle, we see that now a lot of points in the validation set have an implausibility less or equal to 3. This implies that this emulator will reduce the input space slowly. As explained above, having consistent very small standardised errors is not positive: it implies that, even though we trained a regression hypersurface in order to catch the global behaviour of the output, the sigma is so large that the emulator is being dominated by the correlation structure. This means at best that we will have to do many more waves of history matching than are necessary, and at worst that our emulators won't be able to reduce the non-implausible parameter space.

The above exploration highlights the importance of finding a value of σ that produces an emulator which on one hand is not overconfident and on the other is able to quickly reduce the input space.

Return to task on P31

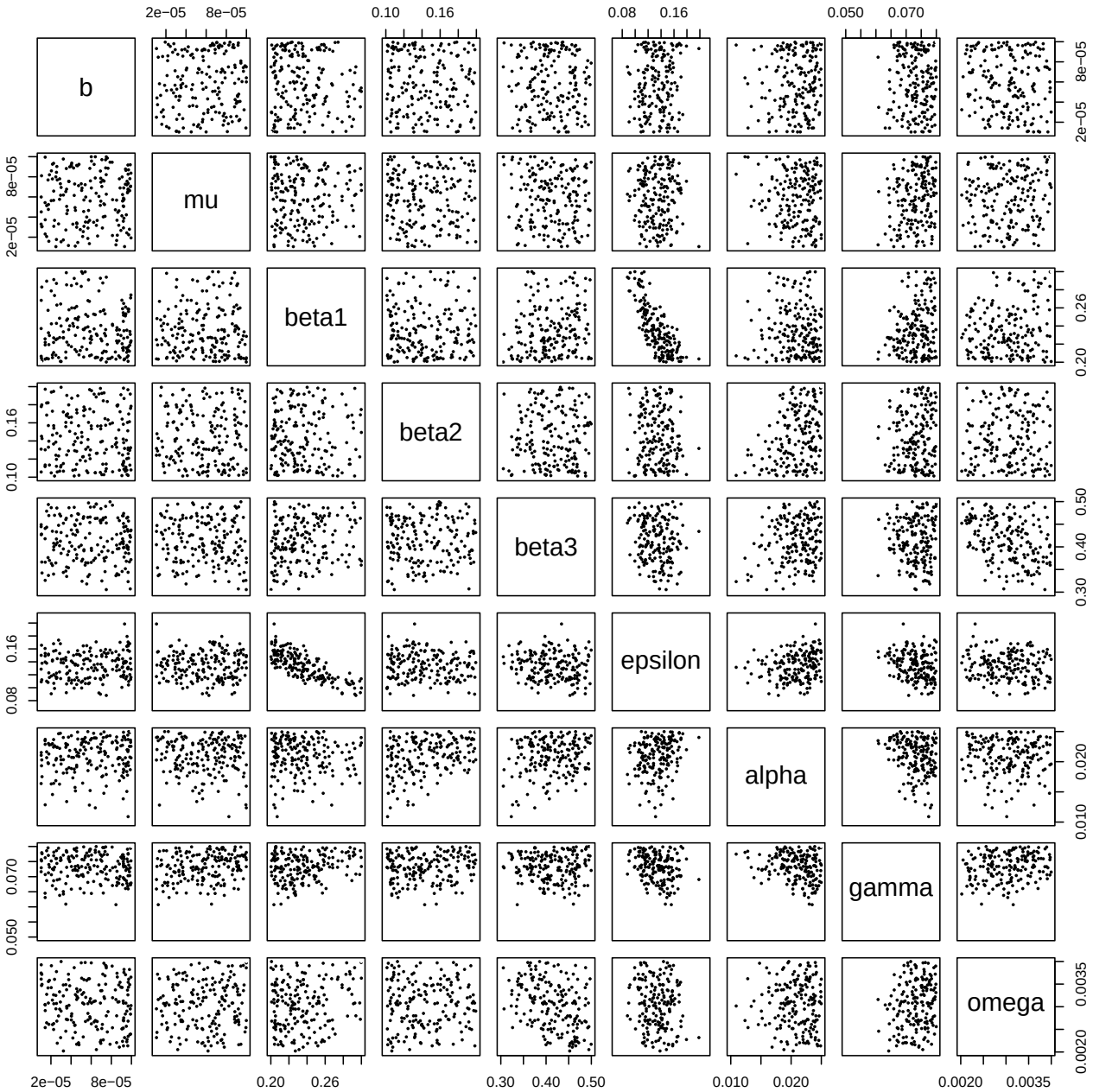
Solution 4

We expect that using all emulators (instead of just a subset of them) will make the non-implausible space smaller than before, since more conditions will have to be met. Let us check if our intuition corresponds to what the plots show:

```
new_points <- generate_new_runs(ems_wave1, 180, targets, verbose = TRUE)
```

```
## Proposing from LHS...
## LHS has high yield - no other methods required.
## Selecting final points using maximin criterion...
```

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



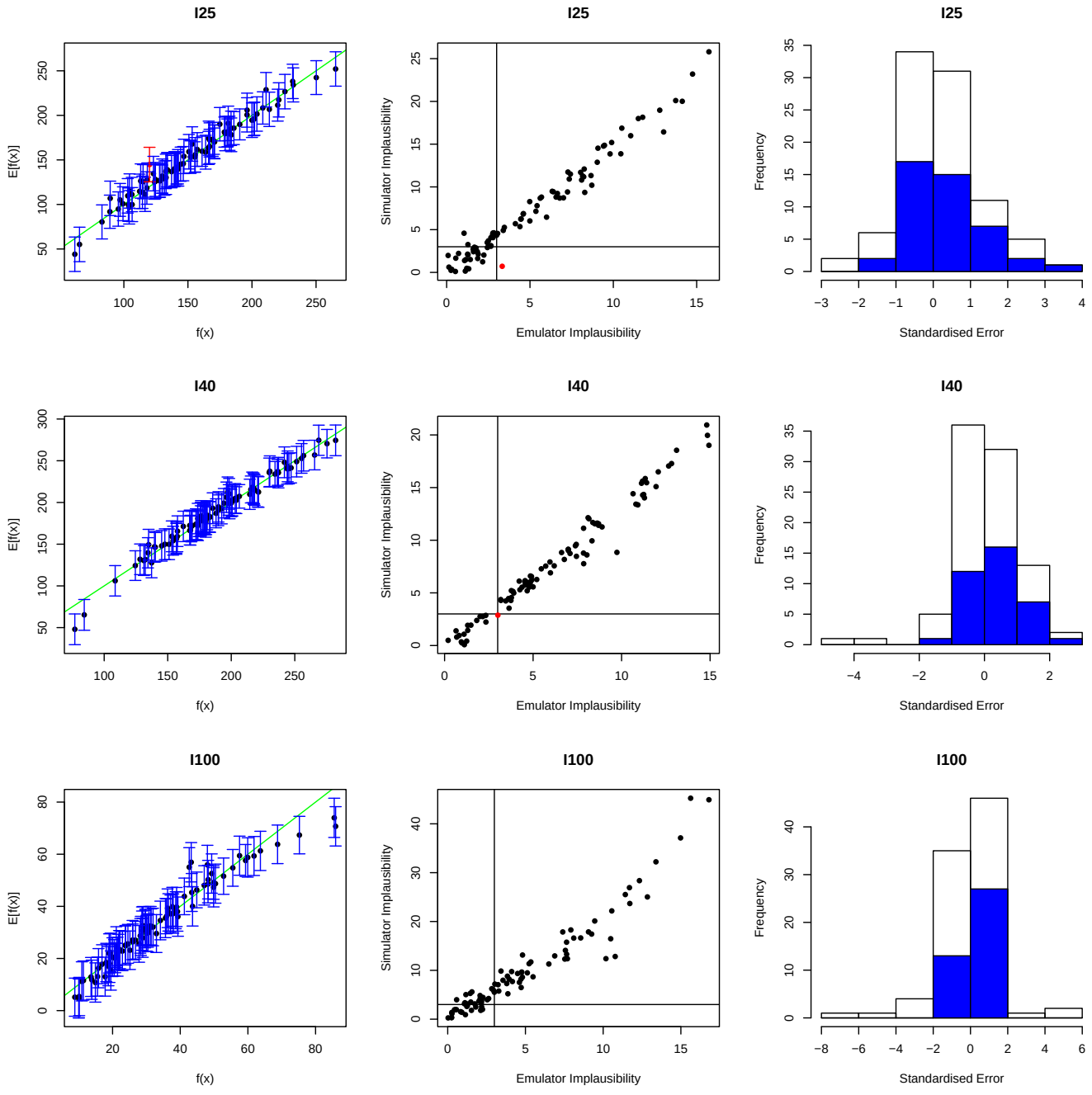
The answer is yes: points now look slightly less spread than before, i.e. a higher part of the input space has been cut-out.

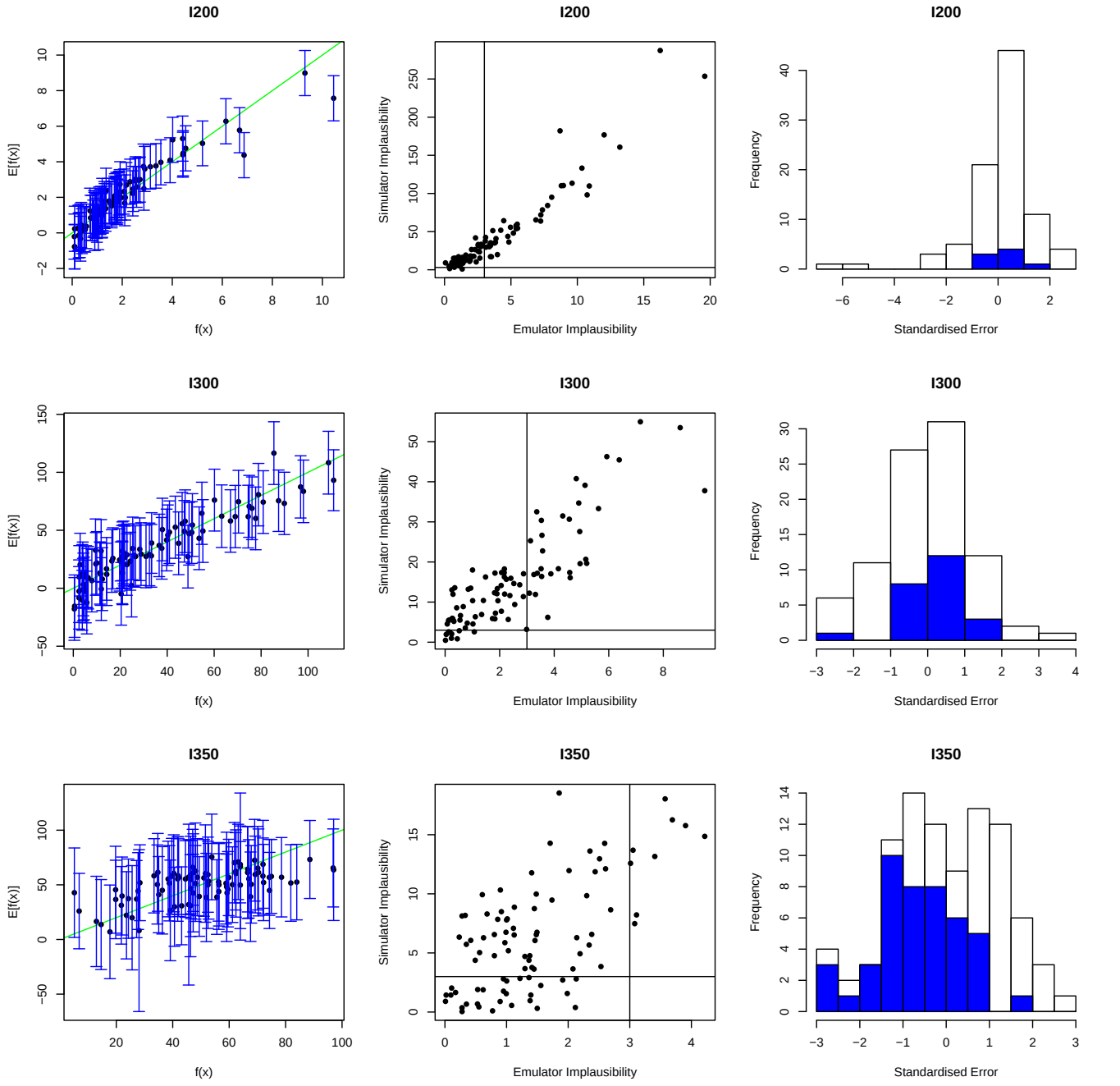
Return to task on P35

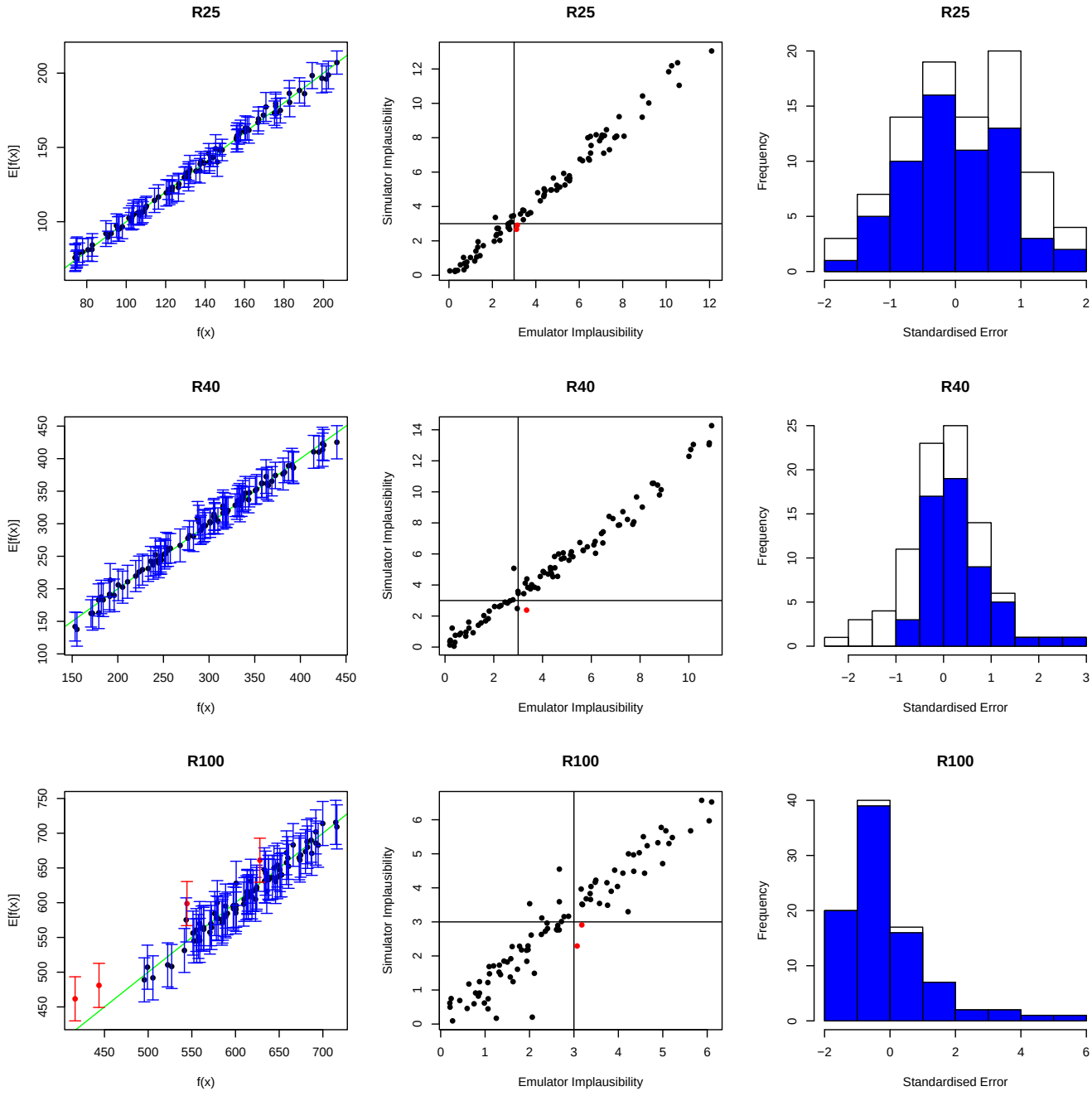
Solution 5

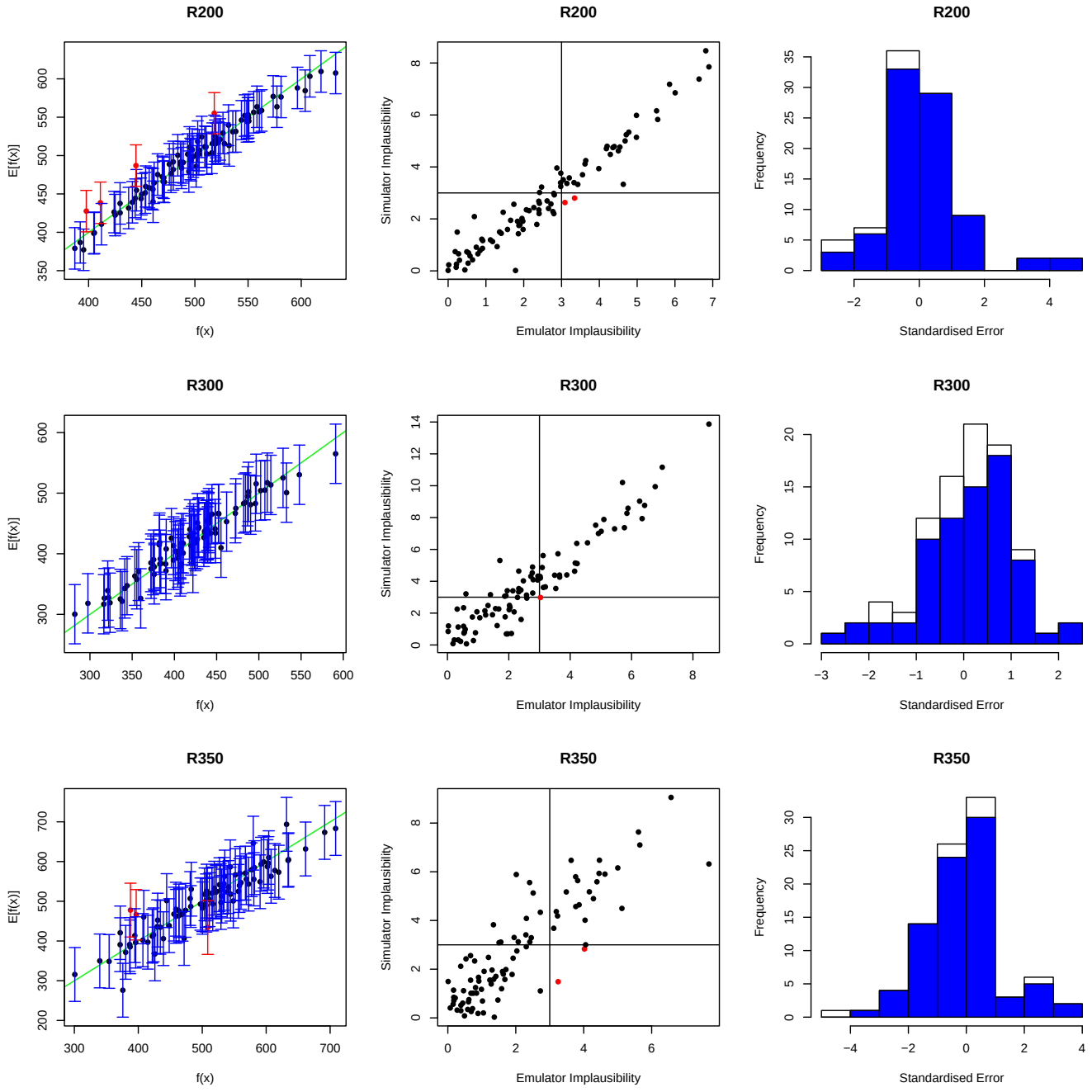
First of all let us take a look at the diagnostics of all the emulators trained in wave 1:

```
vd <- validation_diagnostics(ems_wave1, validation = validation,
                             targets = targets, plt=TRUE)
```




I200
E[f(x)]
f(x)
I200
Simulator Implausibility
Emulator Implausibility
I200
Frequency
Standardised Error
I300
E[f(x)]
f(x)
I300
Simulator Implausibility
Emulator Implausibility
I300
Frequency
Standardised Error
I350
E[f(x)]
f(x)
I350
Simulator Implausibility
Emulator Implausibility
I350
Frequency
Standardised Error

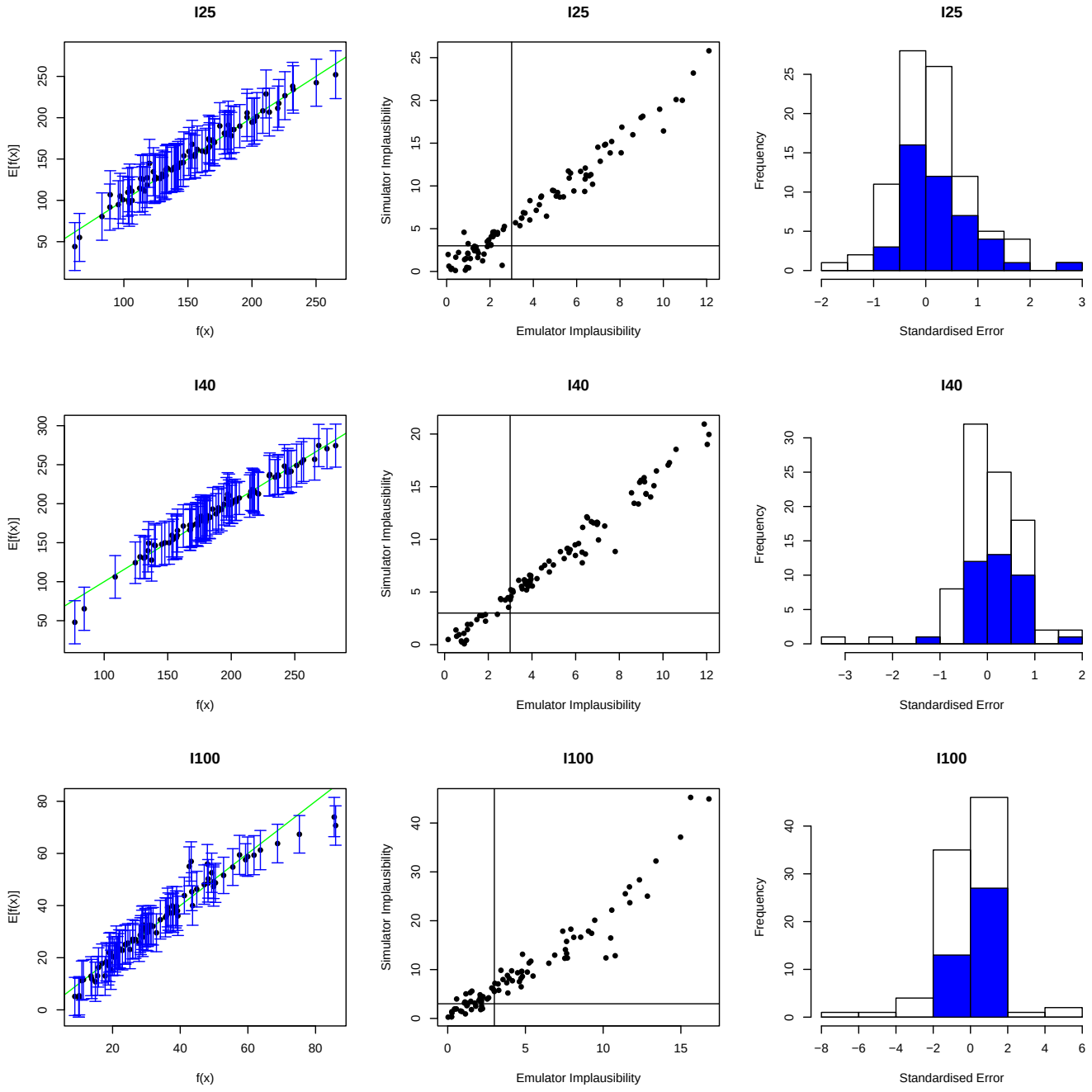

R25
E[f(x)]
f(x)
R25
Simulator Implausibility
Emulator Implausibility
R25
Frequency
Standardised Error
R40
E[f(x)]
f(x)
R40
Simulator Implausibility
Emulator Implausibility
R40
Frequency
Standardised Error
R100
E[f(x)]
f(x)
R100
Simulator Implausibility
Emulator Implausibility
R100
Frequency
Standardised Error

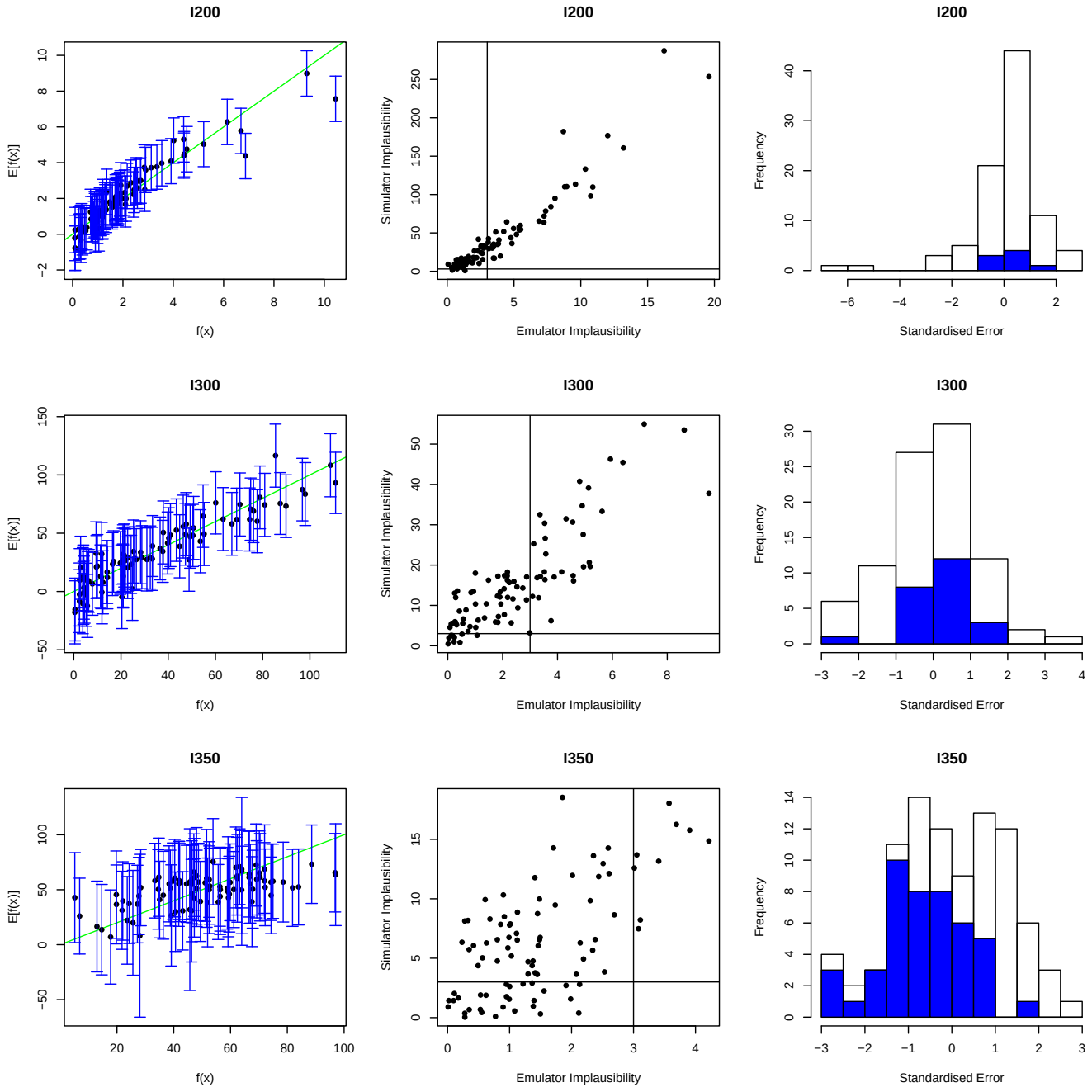


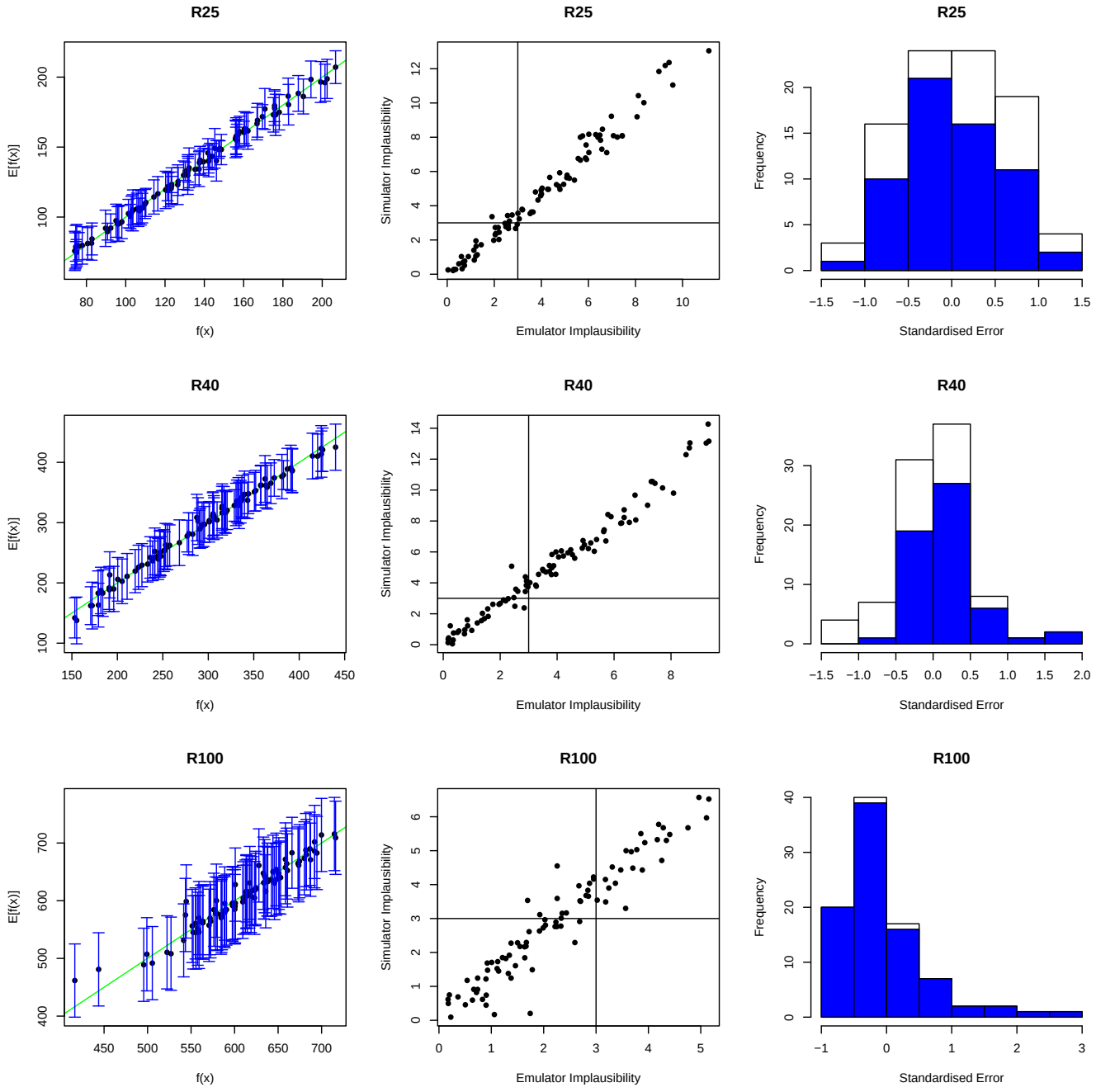
Most emulators would benefit from having slightly more conservative emulators, which we can obtain by increasing σ. After some trial and error, we chose the following values of sigma for our emulators:

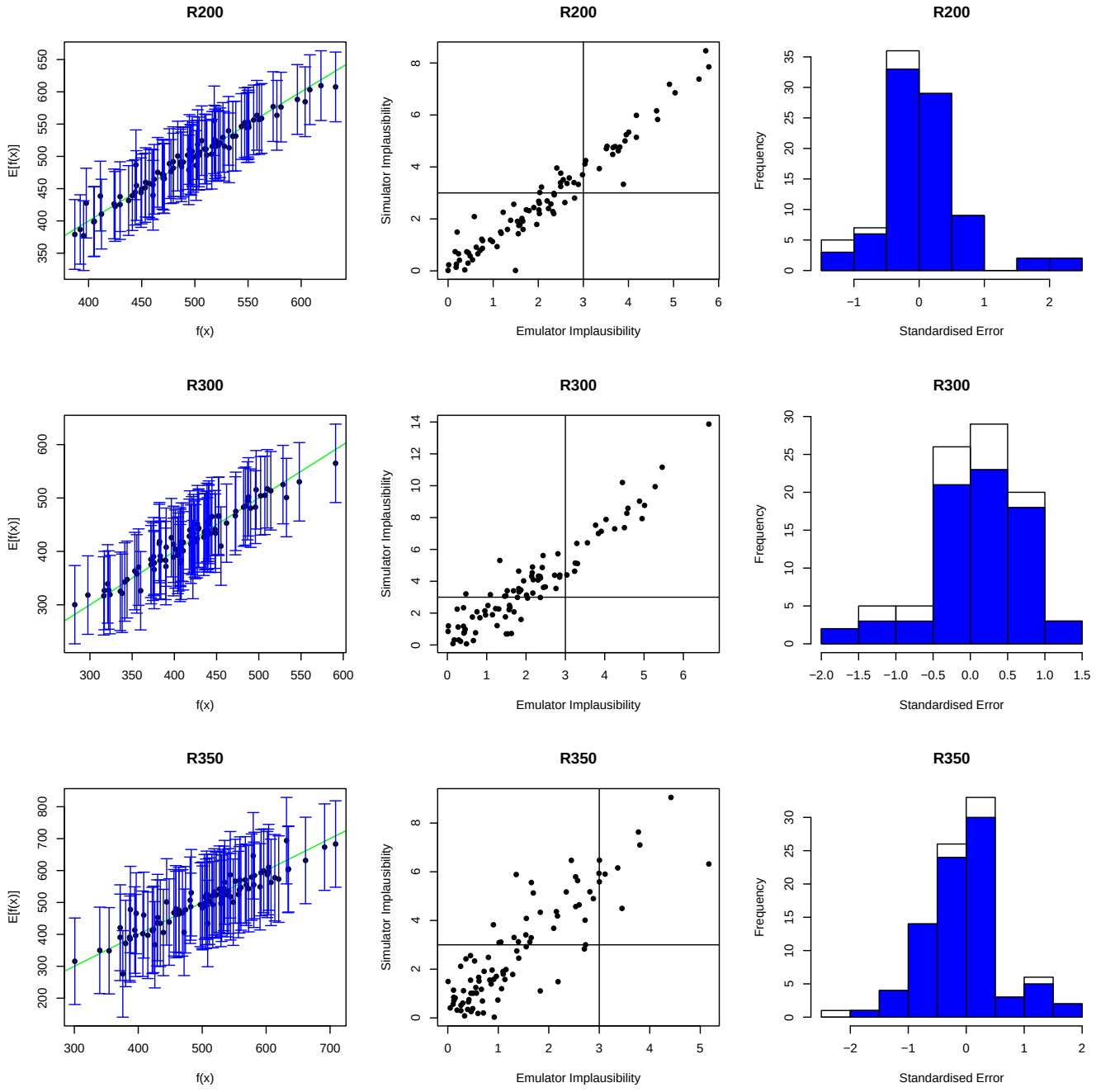
```
inflations <- c(1.5,1.5,1,1,1,1,1.5,1.5,2,2,1.5,2)
for (i in 1:length(ems_wave1)) {
  ems_wave1[[i]] <- ems_wave1[[i]]$mult_sigma(inflations[[i]])
}
```

```
vd <- validation_diagnostics(ems_wave1, validation = validation,
                             targets = targets, plt=TRUE)
```


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
I300
E[f(x)]
f(x)
I300
Simulator Implausibility
Emulator Implausibility
I300
Frequency
Standardised Error
I350
E[f(x)]
f(x)
I350
Simulator Implausibility
Emulator Implausibility
I350
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
R100
E[f(x)]
f(x)
R100
Simulator Implausibility
Emulator Implausibility
R100
Frequency
Standardised Error



The diagnostics look good now, so we can use the current emulators to generate new points:

```
new_points <- generate_new_runs(ems_wave1, 180, targets, verbose = TRUE)
```

```
## Proposing from LHS...
## LHS has high yield - no other methods required.
## Selecting final points using maximin criterion...
```

Return to task on P37

Solution 6

We start by evaluating the function `get_results` on `new_points`. In other words, we run the model using the parameter sets we generated at the end of wave 1:

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

and then binding `new_points` to the model runs `new_initial_results` to create the data.frame `wave1`, containing the input parameter sets and model outputs:

```
wave1 <- cbind(new_points, new_initial_results)
```

We split `wave1` into training and validation sets

```
new_t_sample <- sample(1:nrow(wave1), 90)
new_training <- wave1[new_t_sample,]
new_validation <- wave1[-new_t_sample,]
```

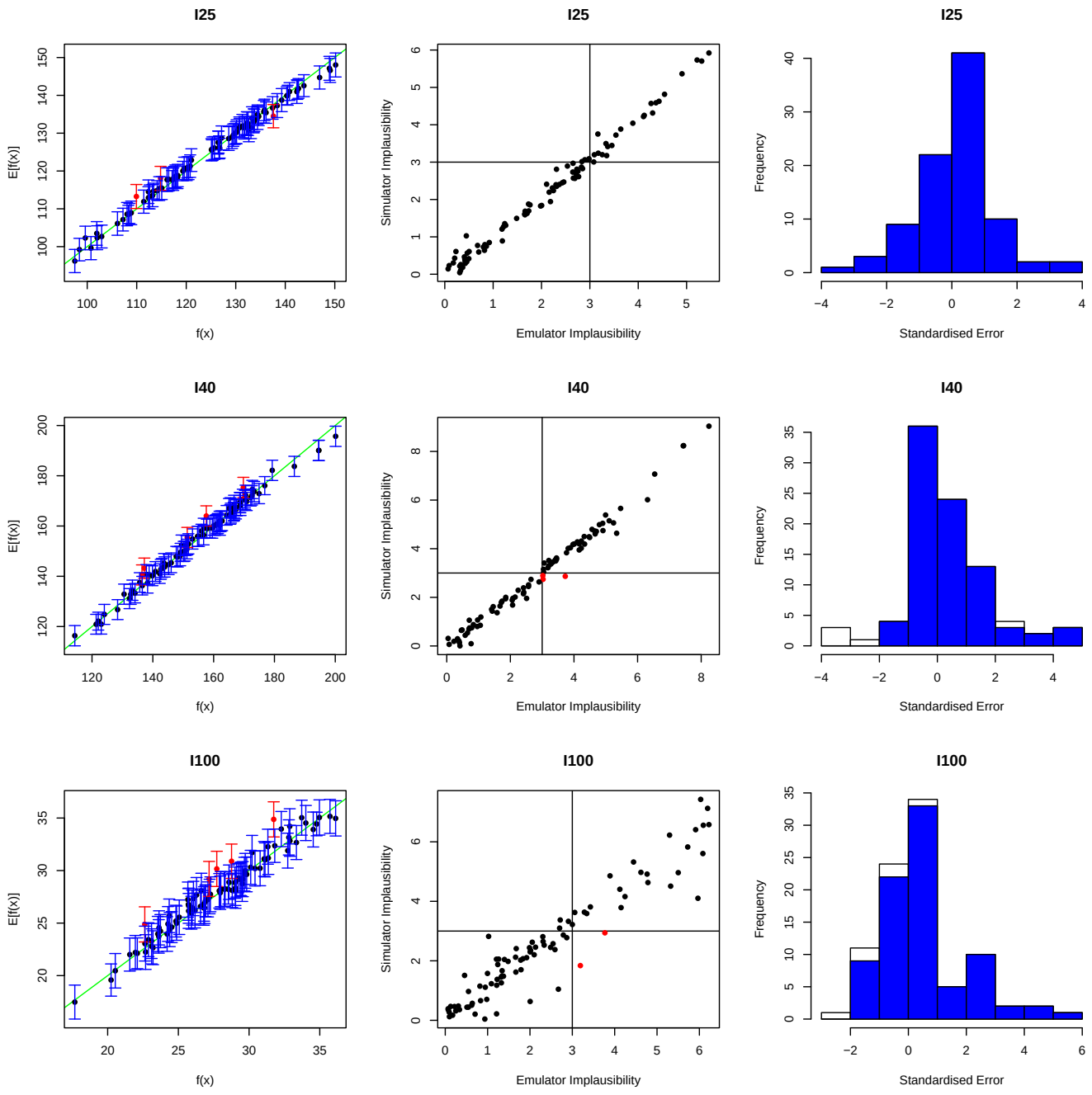
and train wave two emulators on the space defined by `new_ranges`:

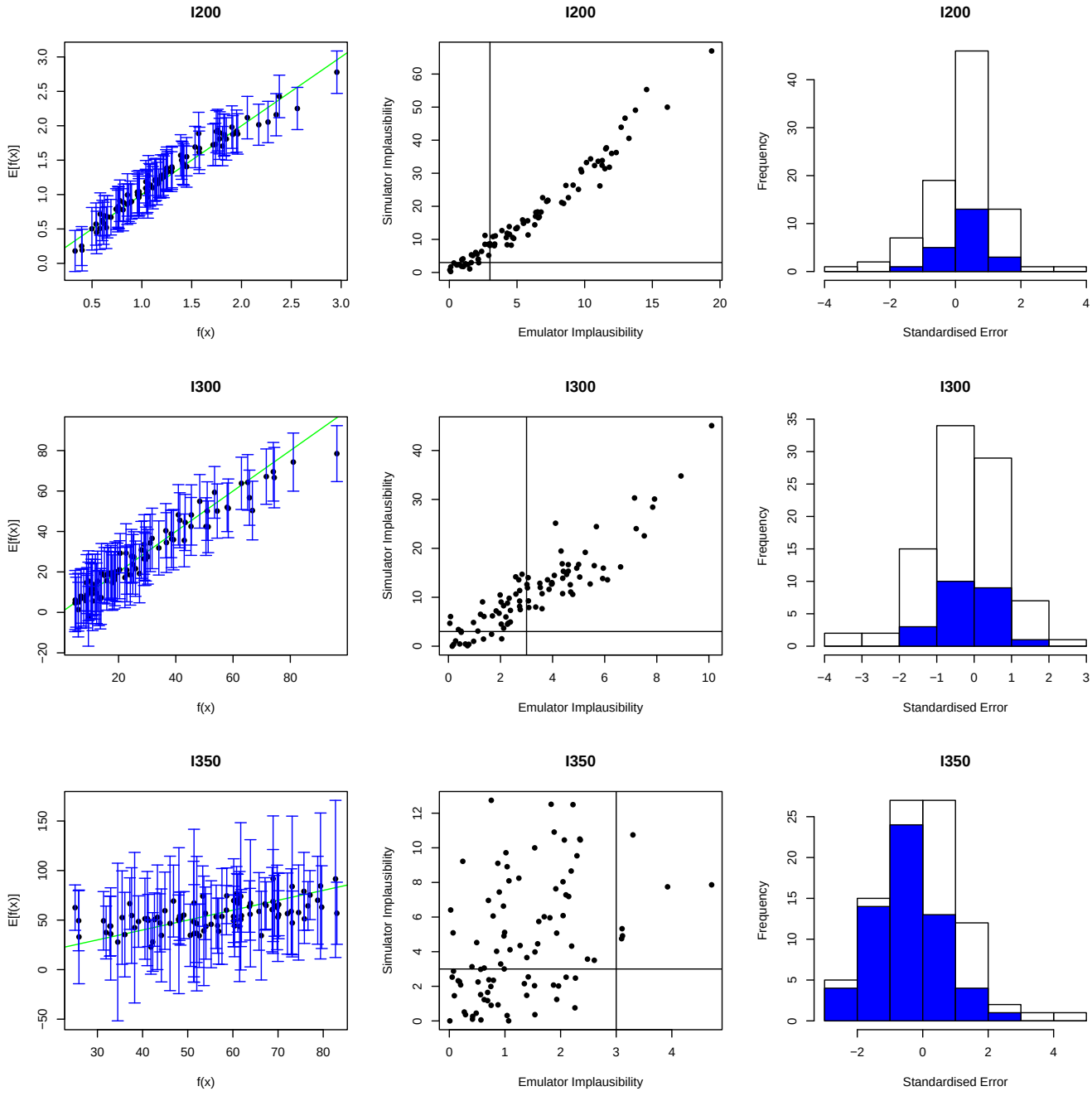
```
ems_wave2 <- emulator_from_data(new_training, names(targets), ranges,
                                check.ranges=TRUE,
                                c_lengths= rep(0.55,length(targets)))
```

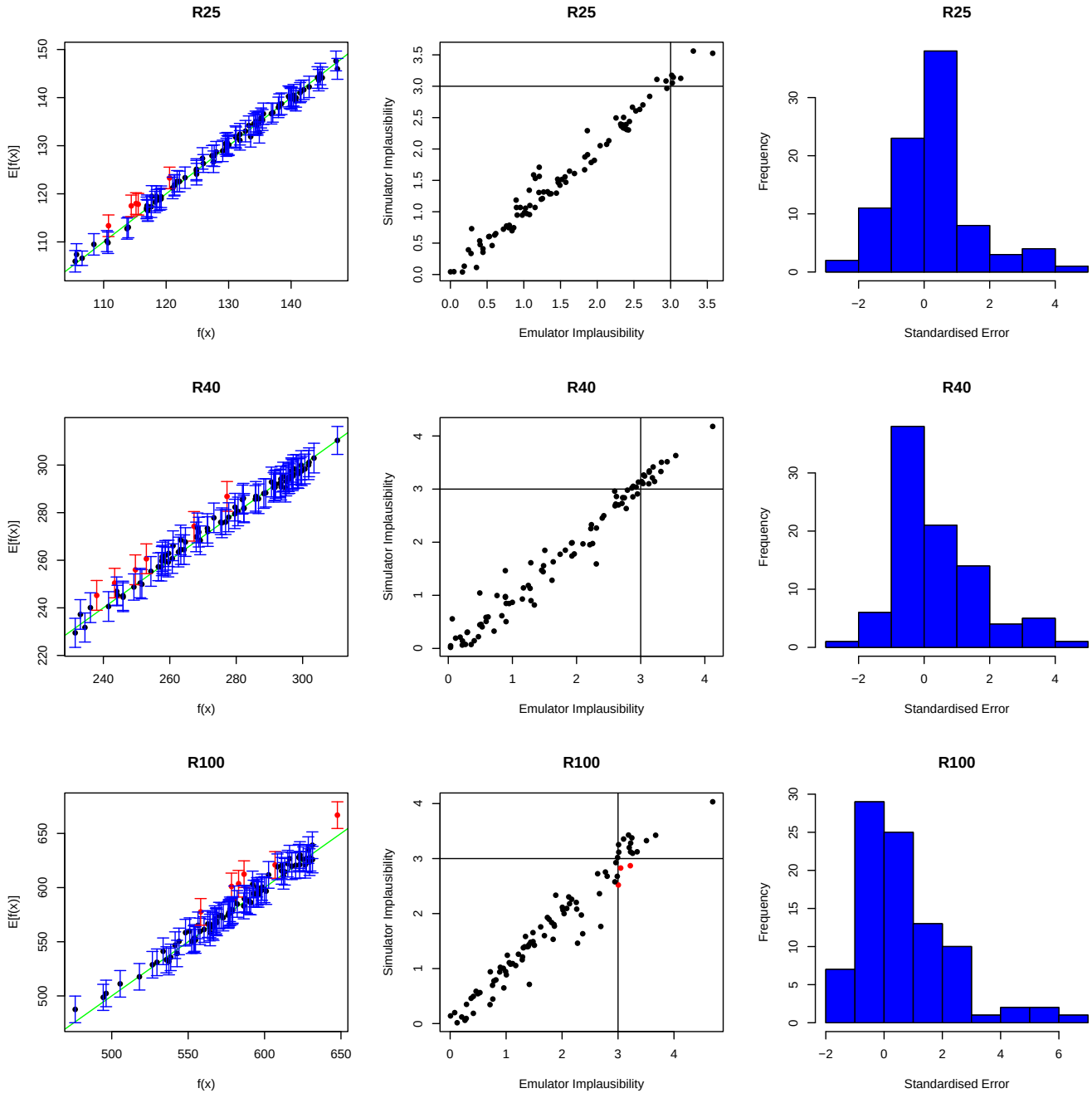
```
## I25
## I40
## I100
## I200
## I300
## I350
## R25
## R40
## R100
## R200
## R300
## R350
## I25
## I40
## I100
## I200
## I300
## I350
## R25
## R40
## R100
## R200
## R300
## R350
```

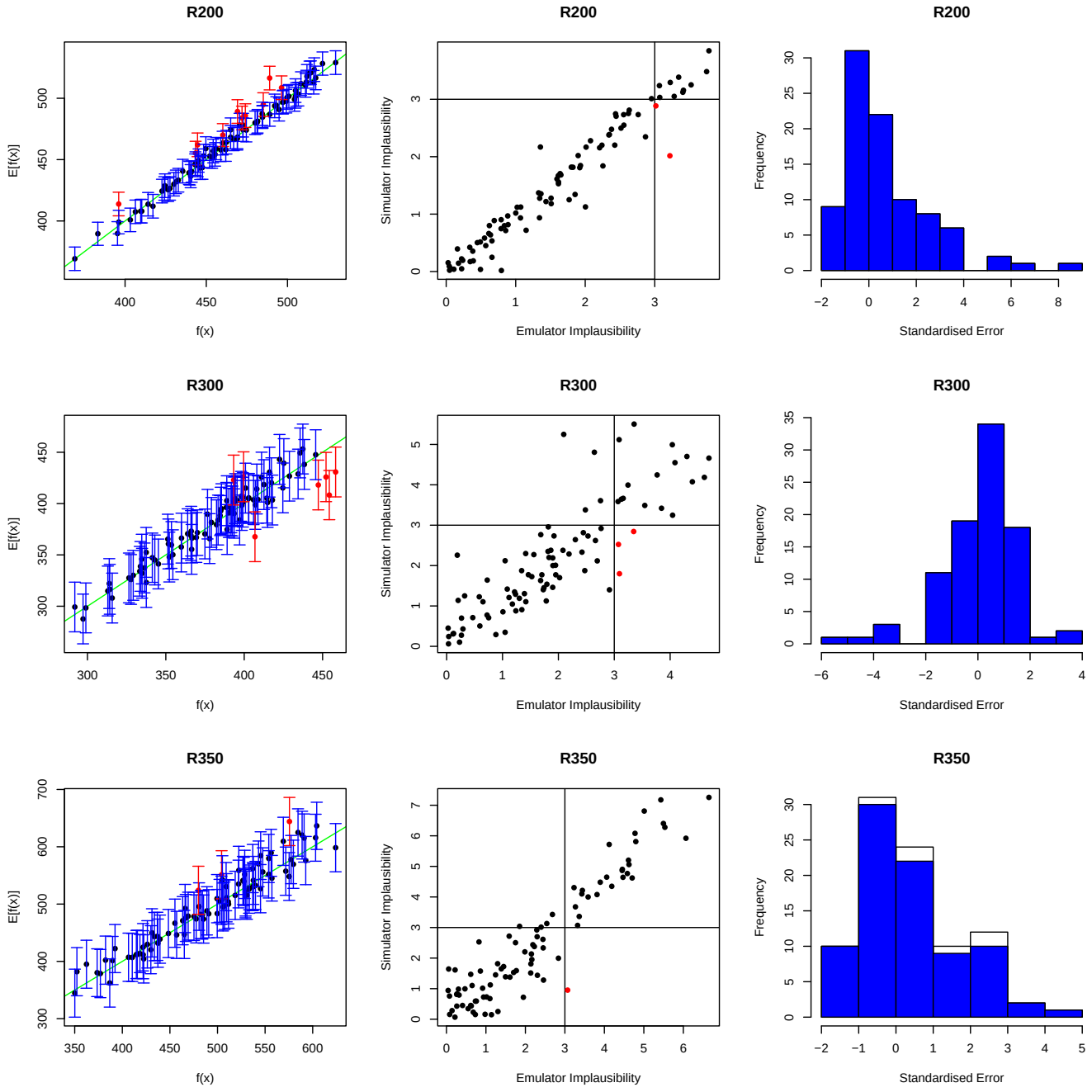
Let us check their diagnostics:

```
vd <- validation_diagnostics(ems_wave2, validation = new_validation, targets = targets,
                             plt=TRUE)
```


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

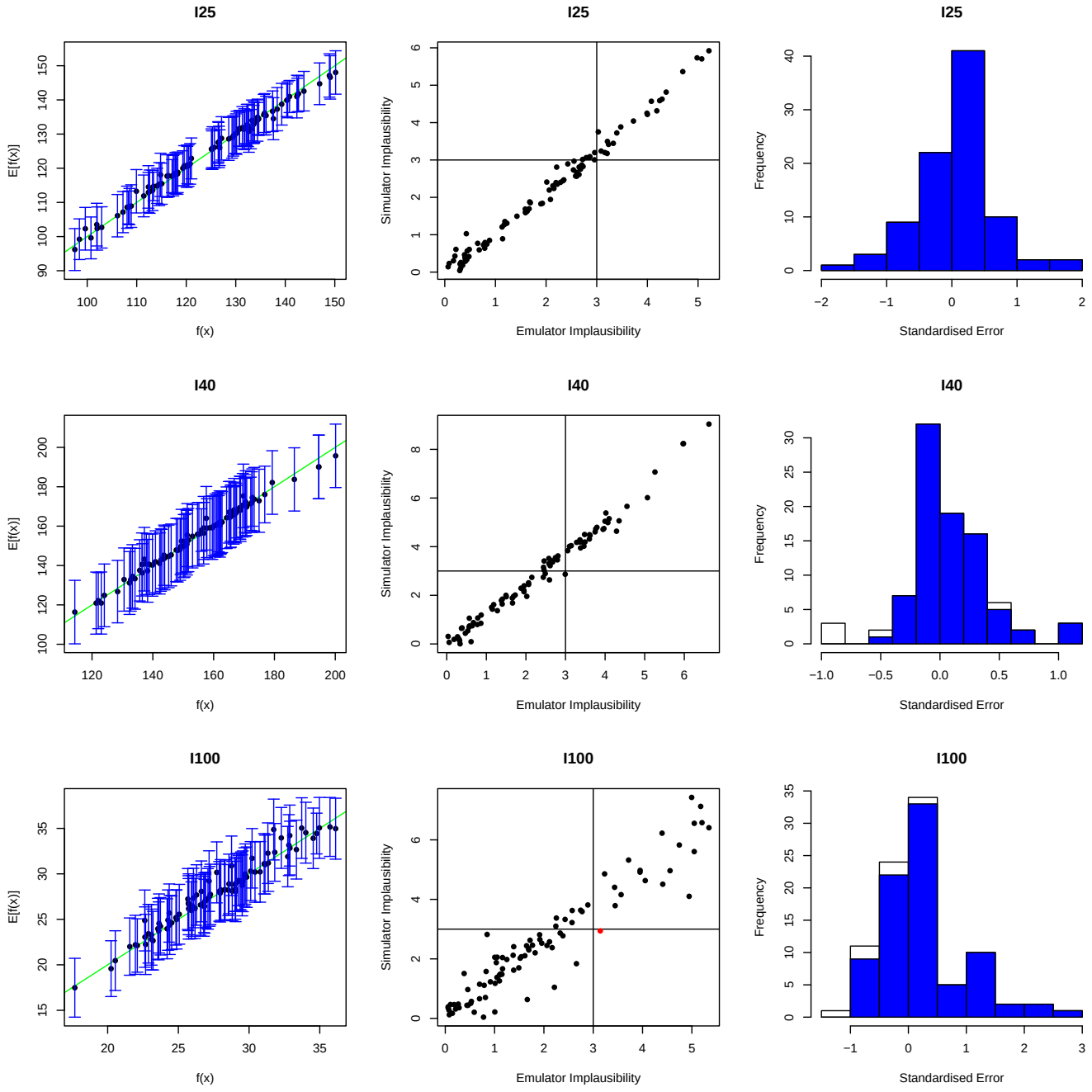

I200
E[f(x)]
f(x)
I200
Simulator Implausibility
Emulator Implausibility
I200
Frequency
Standardised Error
I300
E[f(x)]
f(x)
I300
Simulator Implausibility
Emulator Implausibility
I300
Frequency
Standardised Error
I350
E[f(x)]
f(x)
I350
Simulator Implausibility
Emulator Implausibility
I350
Frequency
Standardised Error

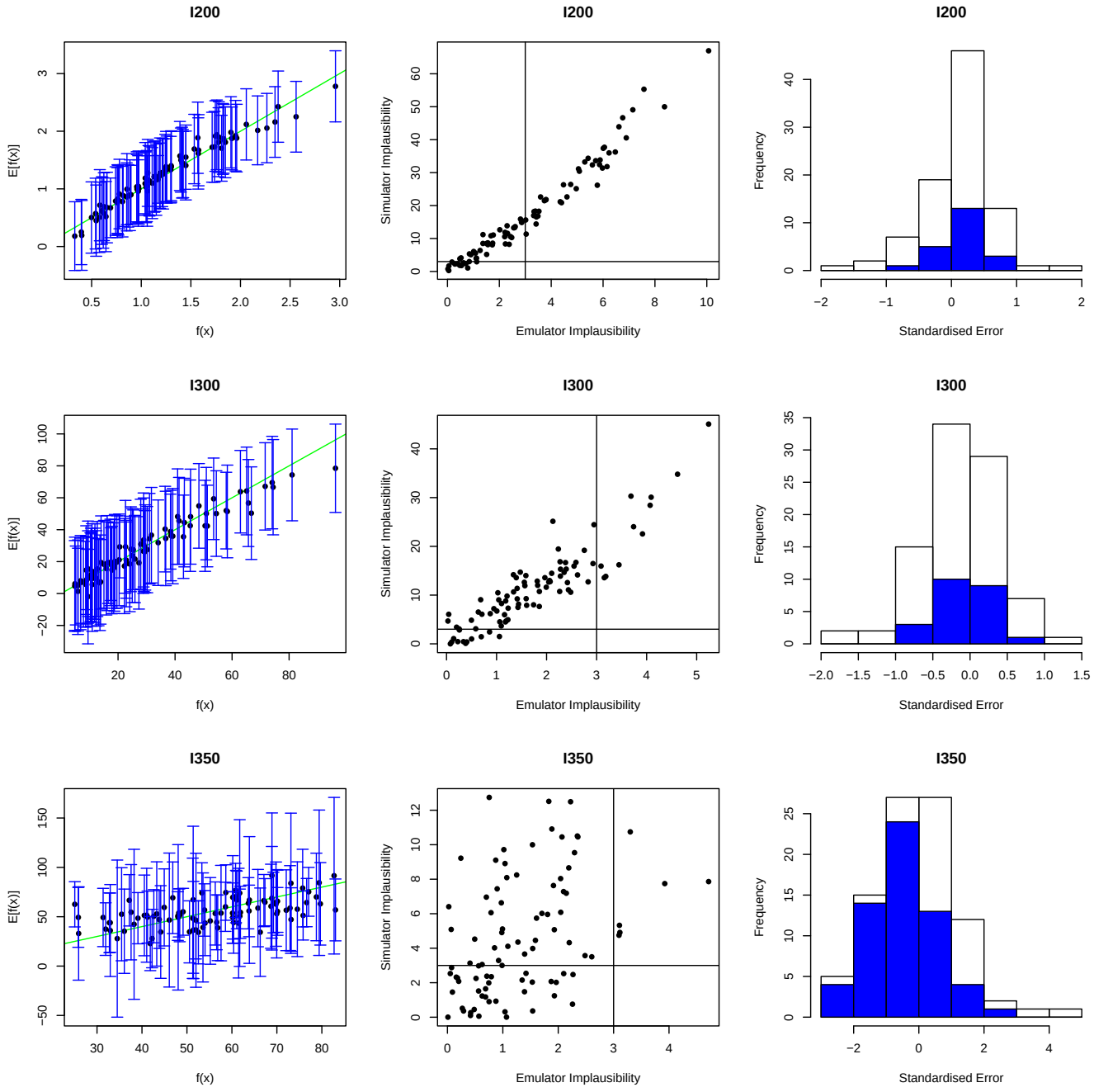

R25
E[f(x)]
f(x)
R25
Simulator Implausibility
Emulator Implausibility
R25
Frequency
Standardised Error
R40
E[f(x)]
f(x)
R40
Simulator Implausibility
Emulator Implausibility
R40
Frequency
Standardised Error
R100
E[f(x)]
f(x)
R100
Simulator Implausibility
Emulator Implausibility
R100
Frequency
Standardised Error

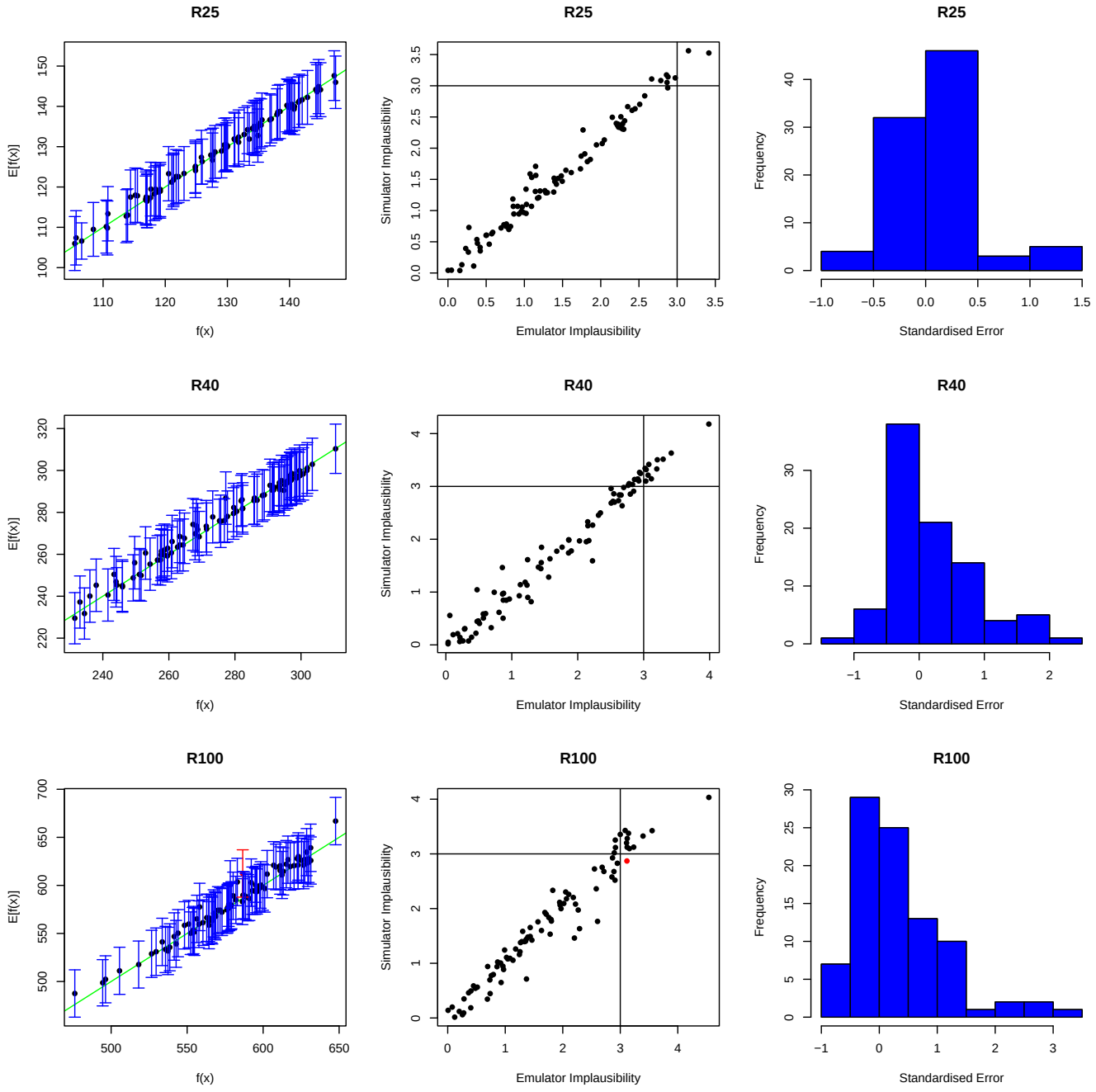


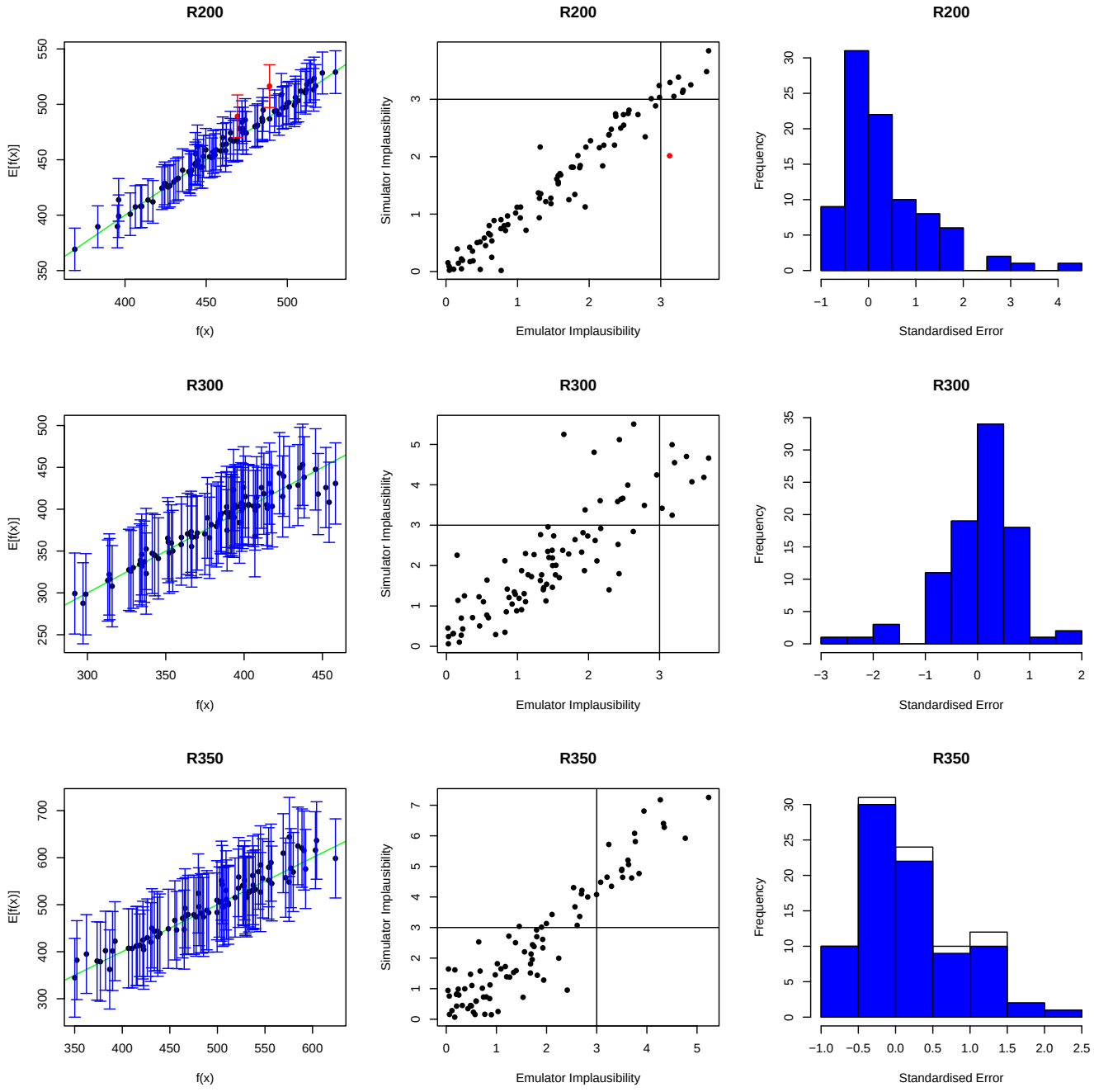
Most emulators fail at least one of the three dagnostics. Let us modify the sigmas in order to build more conservative emulators that pass the three diagnostics. After some trial and error, we chose the following values of sigma for our emulators:

```
inflations <- c(2,4,2,2,2,1,3,2,2,2,2,2)
for (i in 1:length(ems_wave2)) {
ems_wave2[[i]] <- ems_wave2[[i]]$mult_sigma(inflations[[i]])
}
vd <- validation_diagnostics(ems_wave2, validation =  new_validation,
                             targets = targets, plt=TRUE)
```


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
I300
E[f(x)]
f(x)
I300
Simulator Implausibility
Emulator Implausibility
I300
Frequency
Standardised Error
I350
E[f(x)]
f(x)
I350
Simulator Implausibility
Emulator Implausibility
I350
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
R100
E[f(x)]
f(x)
R100
Simulator Implausibility
Emulator Implausibility
R100
Frequency
Standardised Error

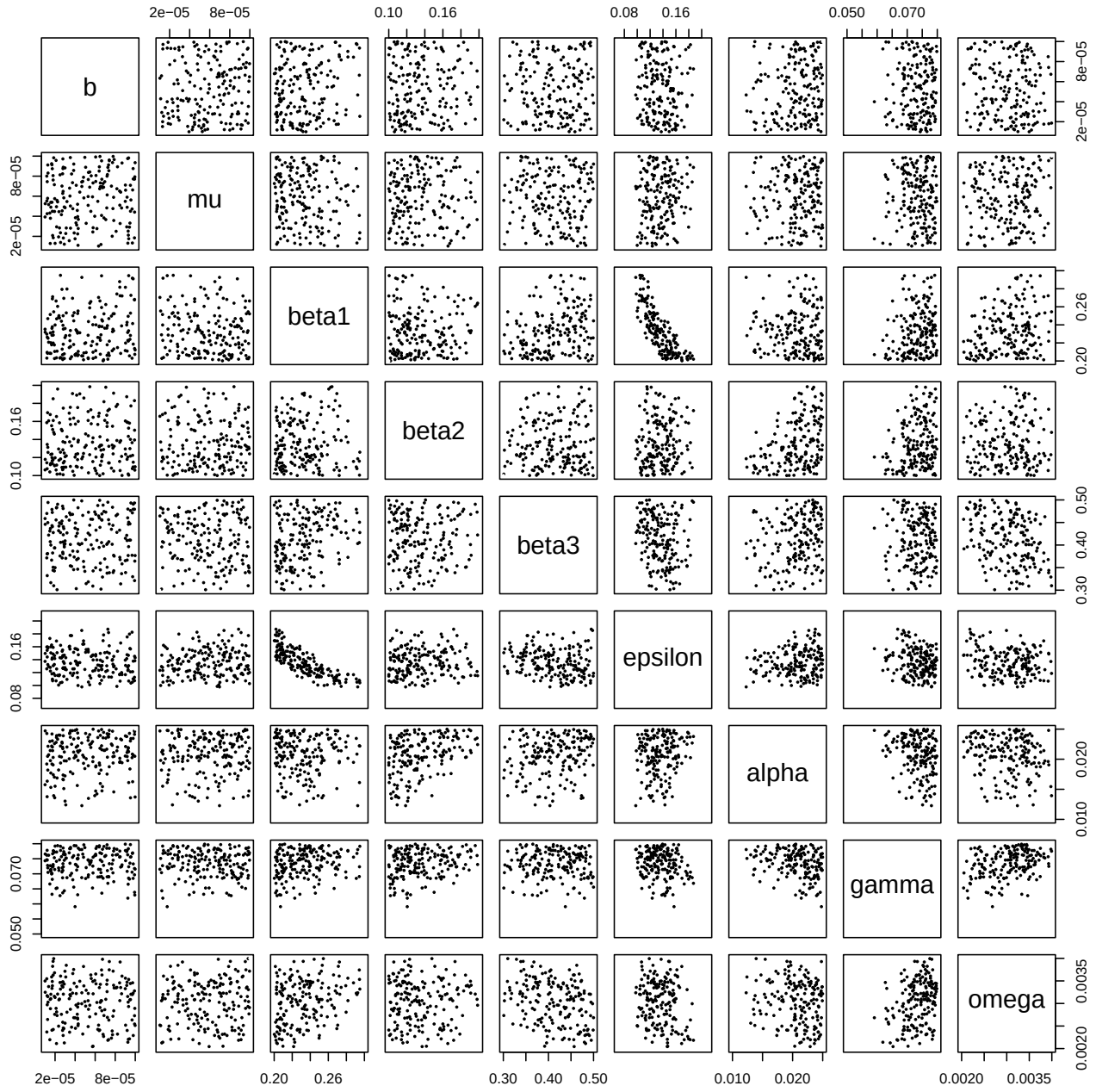


The diagnostics look ok now. Let us try to generate new parameter sets using all emulators build so far:

```
new_new_points <- generate_new_runs(c(ems_wave2, ems_wave1), 180, targets, verbose=TRUE)
```

```
## Proposing from LHS...
## LHS has high yield - no other methods required.
## Selecting final points using maximin criterion...
```

```
plot_wrap(new_new_points, ranges)
```

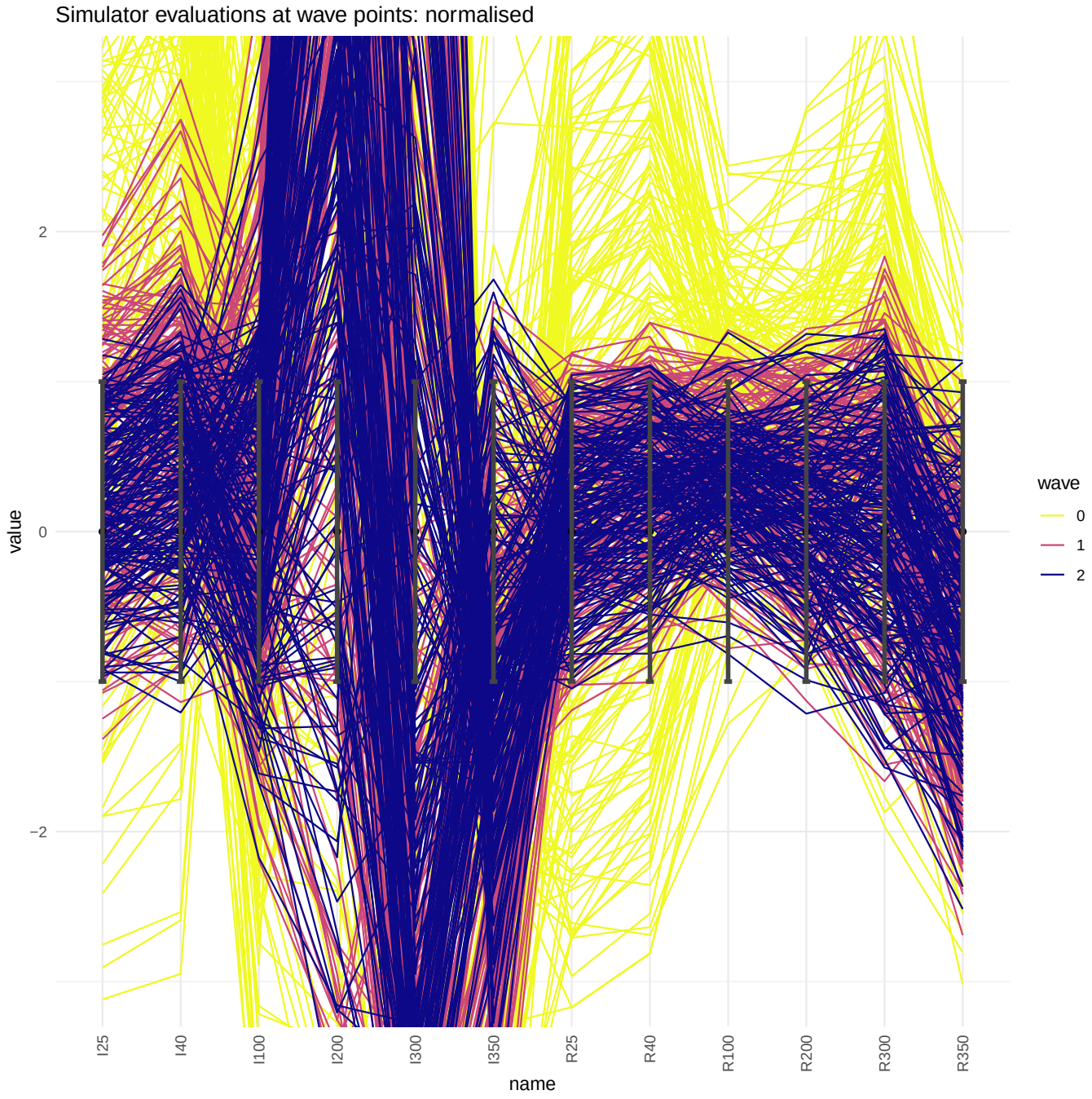


This worked well: the new non-implausible region is clearly smaller than the one we had at the end of wave one.

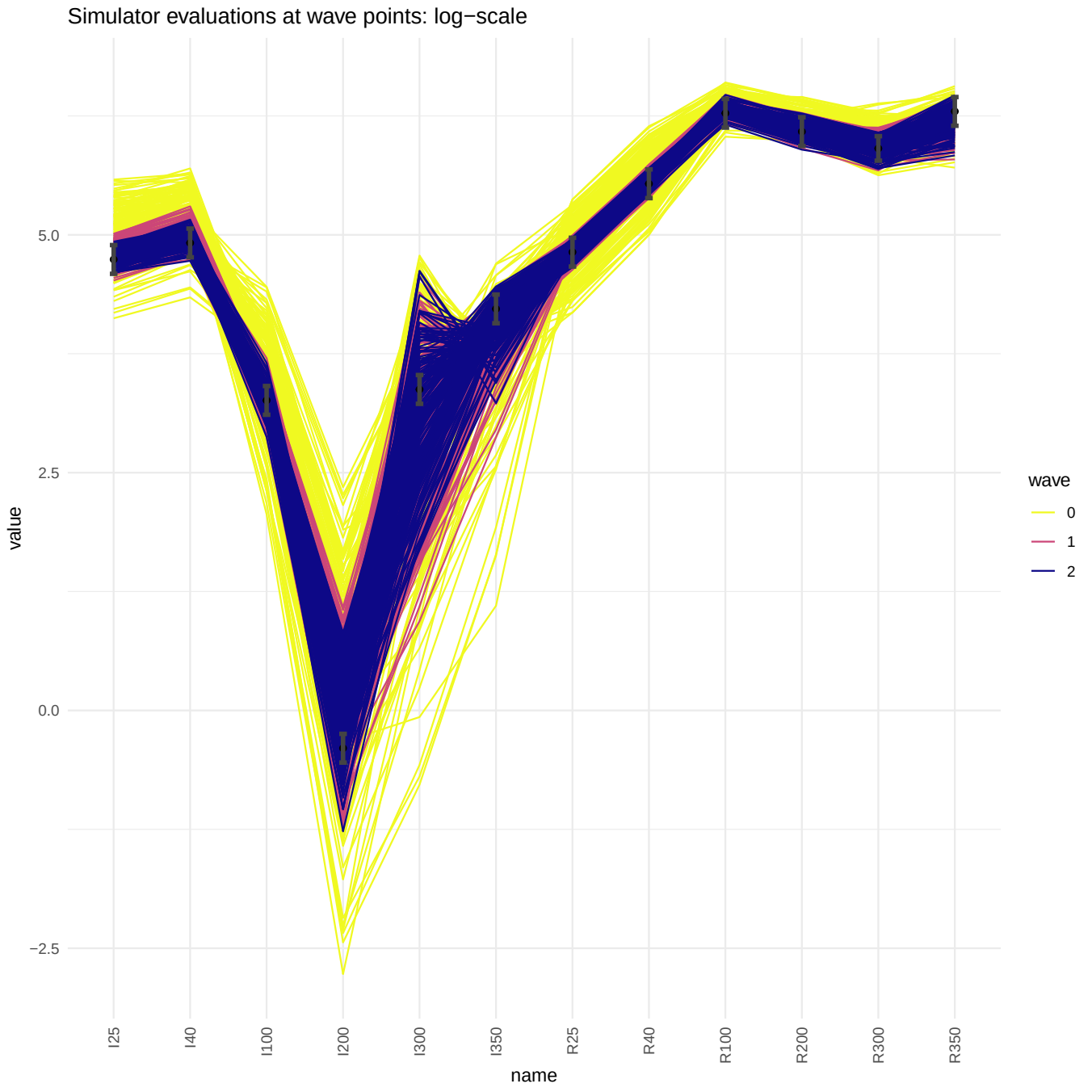
Return to task on P39

Solution 7

```
simulator_plot(all_points, targets, normalize = TRUE)
```




```
simulator_plot(all_points, targets, logscale = TRUE)
```



These normalised/logscaled plots make some targets such as I200 clearer: it is now clear that I200 is not matched yet, even at the end of wave two.

Return to task on P44

Appendix B

Additional information

Code to load relevant libraries and helper functions

```
library(hmer)
library(deSolve)
library(ggplot2)
library(reshape2)
library(purrr)
library(tidyverse)
library(lhs)
set.seed(123)
############################## HELPER FUNCTIONS ##############################
# `ode_results` provides us with the solution of the differential equations for a given
# set of parameters. This function assumes an initial population of
# 900 susceptible individuals, 100 exposed individuals, and no infectious
# or recovered individuals.
ode_results <- function(parms, end_time = 365*2) {
  forcer = matrix(c(0, parms['beta1'], 100, parms['beta2'], 180, parms['beta2'], 270, parms['beta3']),
                  ncol = 2, byrow = TRUE)
  force_func = approxfun(x = forcer[,1], y = forcer[,2], method = "linear", rule = 2)
  des = function(time, state, parms) {
    with(as.list(c(state, parms)), {
      dS <- b*(S+E+I+R)-force_func(time)*I*S/(S+E+I+R) + omega*R - mu*S
      dE <- force_func(time)*I*S/(S+E+I+R) - epsilon*E - mu*E
      dI <- epsilon*E - alpha*I - gamma*I - mu*I
      dR <- gamma*I - omega*R - mu*R
      return(list(c(dS, dE, dI, dR)))
    })
  }
  yini = c(S = 900, E = 100, I = 0, R = 0)
  times = seq(0, end_time, by = 1)
  out = deSolve::ode(yini, times, des, parms)
  return(out)
}
# `get_results` acts as `ode_results`, but has the additional feature
# of allowing us to decide which outputs and times should be returned.
# For example, to obtain the number of infected and susceptible individuals
# at t=25 and t=50, we would set `times=c(25,50)` and `outputs=c('I','S')`.
get_results <- function(params, times, outputs) {
  t_max <- max(times)
  all_res <- ode_results(params, t_max)
  actual_res <- all_res[all_res[,'time'] %in% times, c('time', outputs)]
  shaped <- reshape2::melt(actual_res[,outputs])
  return(setNames(shaped$value, paste0(shaped$Var2, actual_res[,'time'], sep = "")))
}
# `space_filling_design` generates a space filling design of parameter sets
space_filling_design <- function(num_points, num_params, ranges){
initial_LHS_training <- maximinLHS(num_points/2, num_params)
initial_LHS_validation <- maximinLHS(num_points/2, num_params)
initial_LHS <- rbind(initial_LHS_training, initial_LHS_validation)
initial_points <- setNames(data.frame(t(apply(initial_LHS, 1,
                  function(x) x*unlist(lapply(ranges, function(x) x[2]-x[1])) +
                  unlist(lapply(ranges, function(x) x[1])))), names(ranges))
return(initial_points)
}
```

Return to P5

R tip

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

```
example_params <- c(
  b = 1/(60*365),
  mu = 1/(76*365),
  beta1 = 0.2, beta2 = 0.1, beta3 = 0.3,
  epsilon = 0.13,
  alpha = 0.01,
  gamma = 0.08,
  omega = 0.003
)
solution <- ode_results(example_params)
par(mar = c(2, 2, 2, 2))
plot(solution)
```

Return to P14

R tip

If `em` is an emulator, you can change its sigma by a factor x through the following line of code: `em$mult_sigma(x)`.

Return to P31