Workshop 1: calibrating a deterministic model

Danny Scarponi, Andy Iskauskas

Contents

Chapter 1

Objectives

This workshop offers an interactive introduction to the main functionality of the hmer package, using a simple example of an 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 several sciences (epidemiology, cosmology, climate science, systems biology, geology, energy systems etc.). Note that, even though this workshop focuses on an epidemiological model, hmer can be used to calibrate complex models arising in any field. More detail about the methodology behind emulation and history matching, as used in this package, can be found in JSS.

In this workshop, you will be invited to carry out a series of tasks (see "Task" boxes) which will enhance your understanding of the package and its tools. Thanks to these activities, you will learn to calibrate deterministic models using history matching and model emulation, and to use your judgement to customise the process. This workshop should be considered as a natural continuation of Tutorial 1, which introduces the history matching with emulation framework with a one-dimensional example, and of Tutorial 2, which gives a general overview of the history matching with emulation process for deterministic models, and shows how to perform it using hmer. Following Workshop 1, you may also want to read Workshop 2, where we demonstrate how to calibrate stochastic models.

For further discussion and justification of the various stages of the history matching with emulation process, please see Bower, Goldstein, and Vernon (2010) and Vernon et al. (2018).

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

Before starting the tutorial, you will need to run the code contained in the box below: it will load all relevant dependencies and a few helper functions which will be introduced and used later. It is not important that you understand the code here; the functions are simply shorthand for running our "black box" model.

Show: Code to load relevant libraries and helper functions on P90

Chapter 2

An overview of history matching with emulation and hmer

A video presentation of this section can be found here.

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

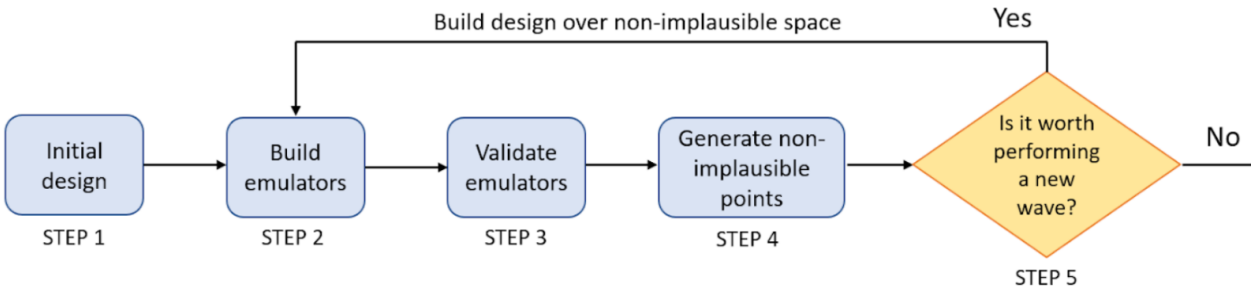


Figure 2.1: History matching with emulation workflow

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

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

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

Step of the HME process	Hmer function			Relevant section 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_design*	• The trained emulators • The number of points to generate • The targets to match to	A dataframe of points deemed non-implausible by all emulators	8 Proposing new po

Chapter 3

Introduction to the model

A video presentation of this section can be found here.

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

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 $\beta(t)$ 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.

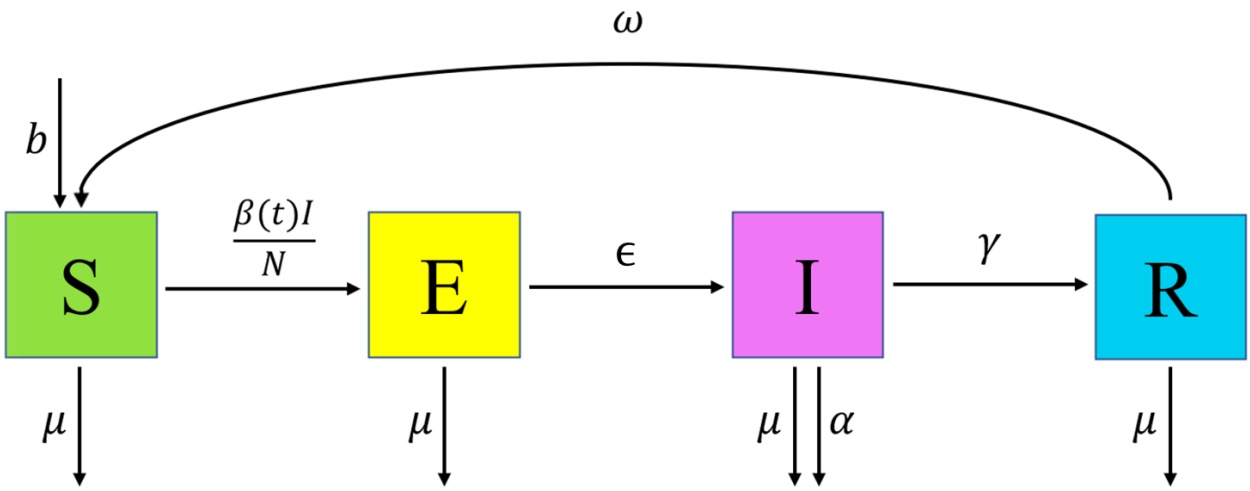


Figure 3.1: SEIRS Diagram

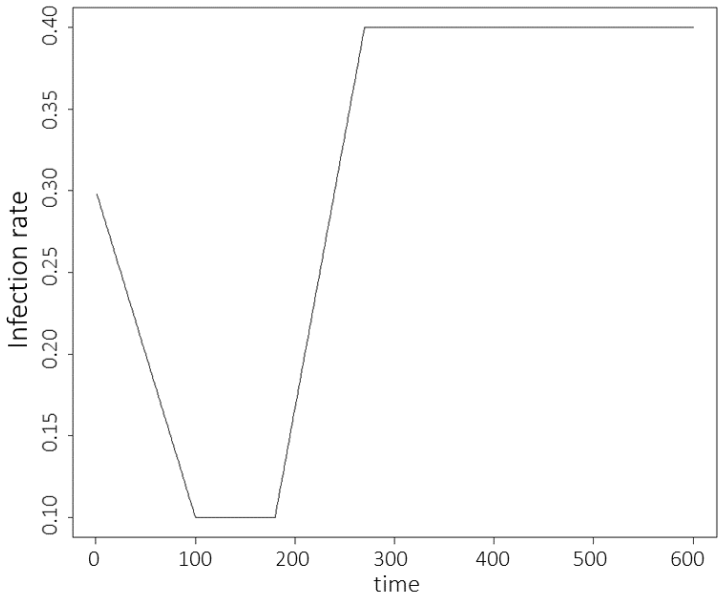
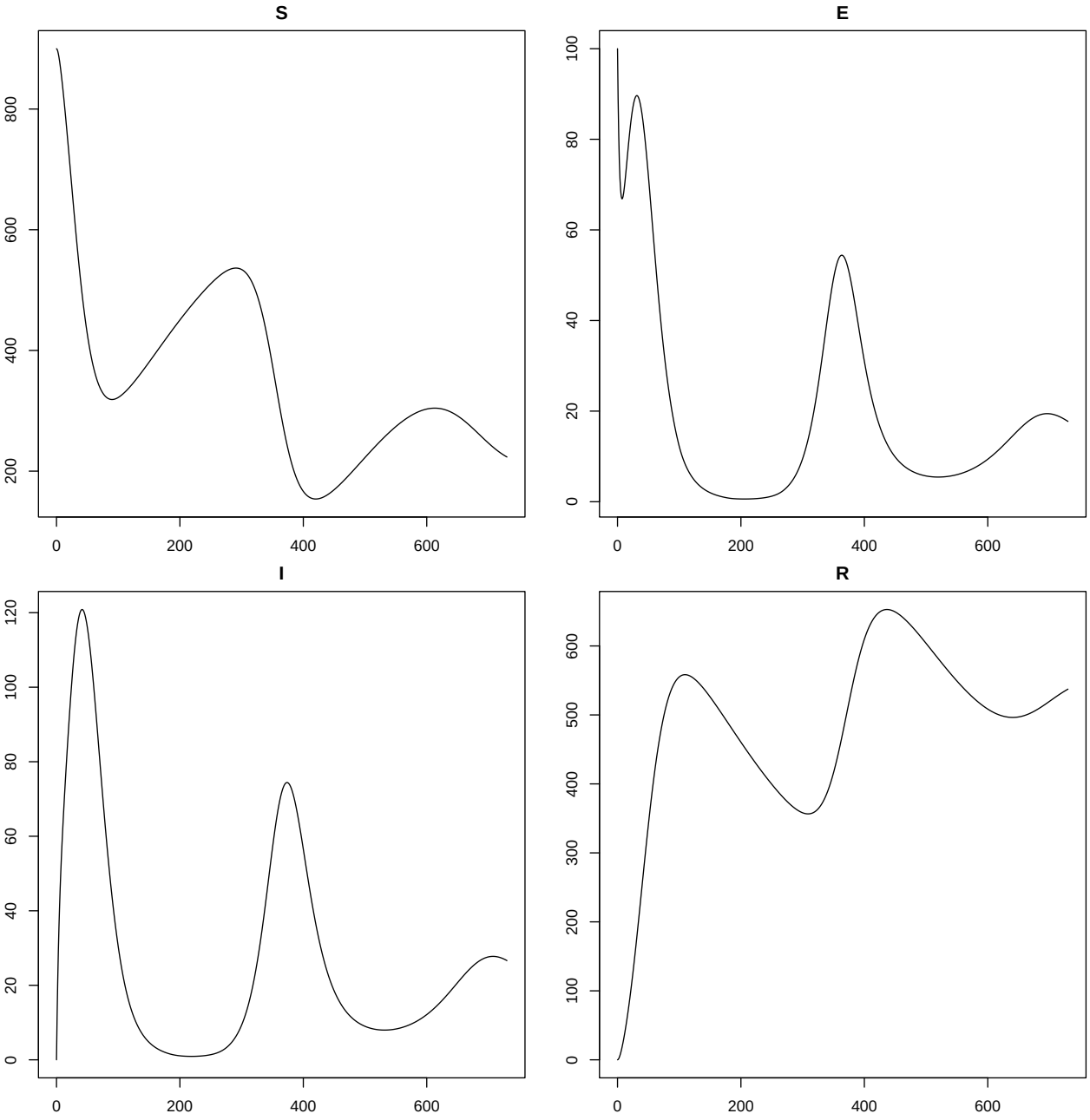


Figure 3.2: Infection rate graph

```r
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)
```



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 P90

Show: Solution on P55

Chapter 4

‘waveO’ - parameter ranges, targets and design points

A video presentation of this section can be found here.

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

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. Note that in general we can also choose to include model uncertainty in the 'sigmas', to reflect how accurate we think our model is. For further discussion regarding model uncertainty, please see Bower, Goldstein, and Vernon (2010), Vernon et al. (2018) or Andrianakis et al. (2015).

Show: More on how targets were set on P91

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

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

For this workshop, we generate parameter sets using a Latin Hypercube design (see fig. 4.1 for a Latin hyper-cube example in two dimensions).

Through the function `maximinLHS` in the package `lhs` we create a hyper-cube design with 90 (10 times the number of parameters) parameter sets for the training set and one with 90 parameter sets for the validation set. After creating the two hyper-cube designs, we bind them together to create `initial_LHS`:

```
initial_LHS_training <- maximinLHS(90, 9)
initial_LHS_validation <- maximinLHS(90, 9)
initial_LHS <- rbind(initial_LHS_training, initial_LHS_validation)
```

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

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

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`:

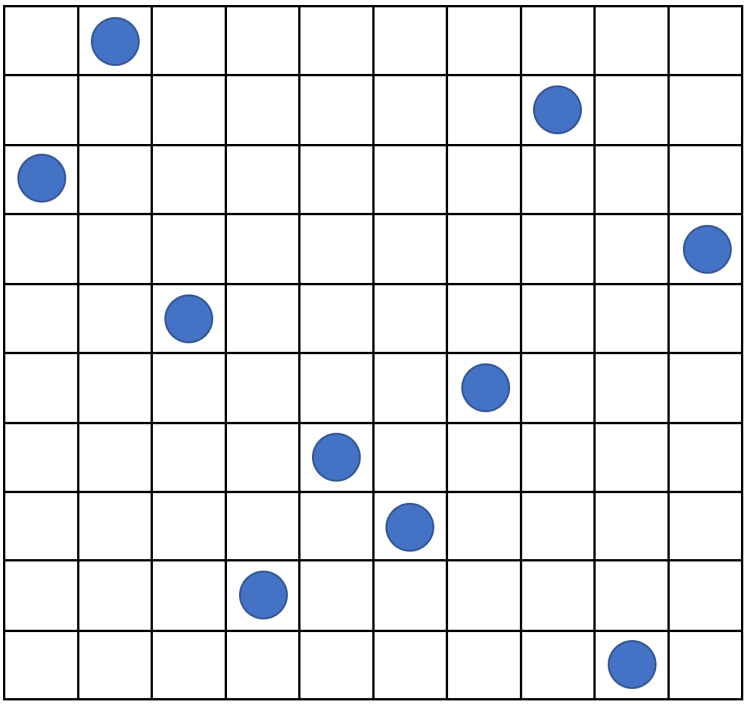


Figure 4.1: An example of Latin hypercube with 10 points in two dimensions: there is only one sample point in each row and each column.

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

Note that your model can be written in any programming languages and that when using your own model, you can create the 'wave0' dataframe however you wish.

Chapter 5

Emulators

A video presentation of this section can be found here.

This section will start with a short recap on the structure of an emulator. We will then train the first wave of emulators and explore them through various visualisations. Note that this section is a bit technical: while it is important to grasp the overall picture of how emulators work, it is not crucial that you understand each detail given in here.

For later use, we 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 do diagnostics of the emulators.

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

5.1 What is an emulator?

An emulator, for our purposes, is a statistical surrogate to a complex simulator. More detailed discussions can be found in Bower, Goldstein, and Vernon (2010) and Vernon et al. (2018), but we will motivate our discussion with a simple example. At its heart, the problem that emulation solves can be framed as follows:

- We have a model from which we can obtain outputs $f(x)$ at a finite number of points in input space x.
- We want to make meaningful statements about the whole range of x values **without** running the model lots of times (for example, because the model is slow to run), by estimating the output at points we haven't evaluated.
- We also want to know how good our estimates are at these unseen points, so that we don't make overconfident statements about the model behaviour.

In this context, an emulator is a representation of the model output that can predict the model output (the **expected** value) at unseen points, and tell us how unsure we should be about those predictions (the **uncertainty**). Let's see this with a simple example before moving on to our toy model. Suppose we have a 'model' that takes a one-dimensional input, x and returns a single output $f(x)$. For this example, we know the exact form of the output:

$$f(x) = 2x + 3x \sin\left(\frac{5\pi(x - 0.1)}{0.4}\right).$$

However, we are going to pretend that the model takes a long time to evaluate points, and that we can only obtain ten evaluations in a reasonable space of time: we only have access to values of $f(x)$ for $x = 0.05, 0.1, \ldots, 0.5$.

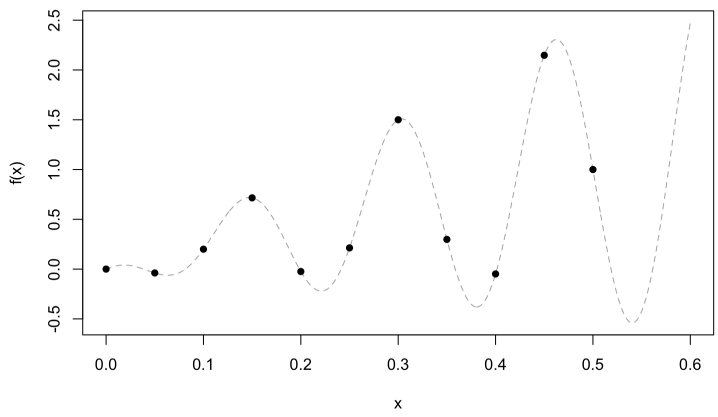


Figure 5.1: Our one-dimensional function. The grey line represents the function; the black points are the only points we are allowed to consider when building the emulator.

The general structure of a univariate emulator is as follows:

$$f(x) = h(x)^T \xi + u(x) := \text{global part} + \text{local part}.$$

The first term is the "global" or "regression" part, which characterises the response of $f(x)$ to x across the space. It splits into two pieces:

- a vector of functions $h(x)$ of the inputs x, which determine the shape and complexity of the regression. If we expect a linear response, we would have $h(x) = (1, x)$; if we had a quadratic response, we instead have $(1, x, x^2)$. Note that we always include a constant term. These functions are considered "known"; they will not change during the process once chosen.
- a corresponding vector of regression coefficients ξ, of the same length as $h(x)$. We think of these as random quantities, and merely specify what we expect them to be (and in principle what our uncertainty is about them): our specifications will therefore correspond to $\mathbb{E}[\xi]$ and $\text{Var}[\xi]$.

Of course, a regression surface is unlikely to perfectly capture the behaviour of our model. We couple it to a "local" part which characterises the small-scale variability away from the global behaviour. We represent this by a weakly stationary process, often with expectation 0 (as we don't expect that the deviations will be systematically high or low). We can define this using a statement about the covariance at two points x and x'

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

The overall variance, σ^2, represents the variability of the local deviations from the global surface. The **correlation function**, $c(x - x')$, depends only on the distance between the two points and we expect it to have the following properties:

- If the distance $x - x'$ becomes large, the covariance should become small; equivalently, $c(x - x') \to 0$ as $x - x' \to \infty$.
- If the distance between x and x' goes to 0, we should have perfect correlation: $c(0) = 1$.

There are many correlation functions one could use: here we will use the exponential-squared correlation function

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

which obeys the relations we want: the parameter θ is a **correlation length**. The larger θ is, the further apart x and x' must be before the correlation function substantially decreases. In general, a larger θ results in a "smoother" function.

For our emulator of the function $f(x)$ we need to decide on these pieces in turn.

- We choose the most simple regression surface: just a constant function, so $g(x) = [1]$. We could choose a linear term, too, but let's assume that all we know about the model is that the output is 'wiggly'.
- The corresponding vector of regression coefficients, ξ. Since we only have one regression function, we only need one coefficient. Let's just take the mean of our function values, so $\xi \approx 0.55$.
- The correlation function we take to be the exponential-squared: the function looks quite smooth.
- The 'hyper-parameters' of the correlation function are trickier to determine. The overall variance σ^2 can be estimated from the sum of squared distances away from our regression surface (equivalently, the squared residuals), and we see that $\sigma^2 \approx 0.32$. The correlation length we must decide: taking $\theta = 1/3$ implies that we don't expect the smaller points to have an impact on the larger ones, though central points have an influence that extends both sides.

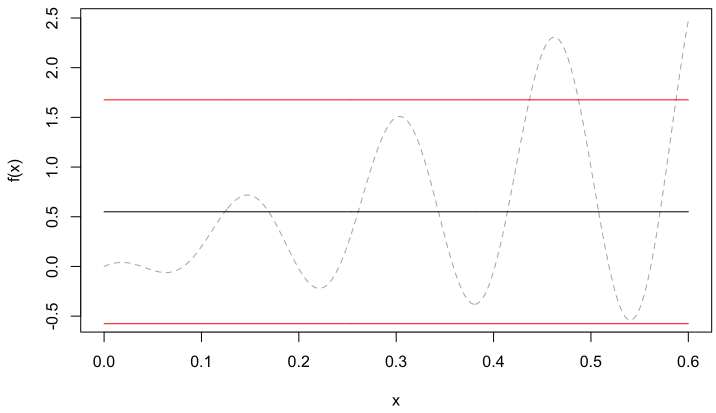


Figure 5.2: The predictions from our emulator with prior specifications as above. Red lines indicate 2-sigma bounds on the predictions (black line).

This doesn't look terribly useful…our predictions are quite useless, don't track the function, and our uncertainty is large. However, this is only our **prior** specification; the emulators consist of random quantities and we can use a modified version of Bayes theorem (see Goldstein and Wooff (2007) for more details) to update this with respect to data. We have our 10 data points from the model: let's adjust the emulator predictions with respect to this data and see what the predictions look like now.

The structure of the emulator has been updated with respect to the data, and now the local $u(x)$ term pulls our emulator predictions up or down relative to how close the point x is to a 'known' training point. The emulator prediction now follows the actual function very closely across the full range, and the uncertainty of the predictions has massively decreased. The only places where the emulator prediction and the model output don't match well are at the edges, but the emulator uncertainty is still large in these regions representing the fact that the emulator does not 'expect' to be good in those regions. Somehow, just 10 points has been sufficient to capture the complicated behaviour of this function; even with very vague prior statements.

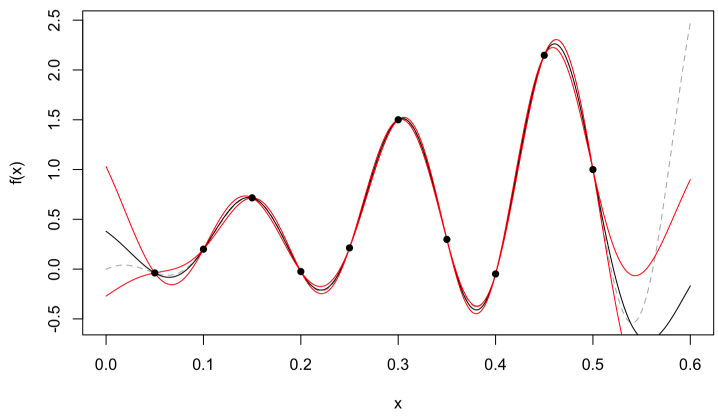


Figure 5.3: Predictions from the data-adjusted emulator above. Red lines indicate 2-sigma bounds on the predictions (black line), and the location of training points are shown.

5.1.1 Picking Priors

We could easily decide on 'sensible' prior statements for this 1d example, since we can visualise the data points easily. However, this is not the case for actual applications: how could we decide on prior parameters in more complicated examples?

The good news is that we don't have to: hmer has a function for that. The `emulator_from_data` function, which we will use in detail below, requires only the data from the model runs, the name of the output(s) we want to emulate, and the ranges of the input(s): given this, it will determine the regression surface, the coefficients, and pick sensible values for the overall uncertainty σ^2 and correlation length θ. It then performs the update with respect to data and gives us the final emulator.

Task 2

If you want to persuade yourself that no magic has been performed here, the dataset used is generated below along with code to create an emulator. Note that `emulator_from_data` does not 'see' the function itself: only the ten evaluated points.

```
func <- function(x) 2*x + 3*x*sin(5*pi*(x-0.1)/0.4)
pts <- seq(0.05, 0.5, by = 0.05)
output <- sapply(pts, func)
test_em <- emulator_from_data(input_data = data.frame(x = pts, f = output),
                              output_names = c('f'),
                              ranges = list(x = c(0, 0.6)))
```

As we go through the rest of this workshop, see if you can generate a plot similar to the one above!

5.2 Training emulators

We will now train the emulators using the `emulator_from_data` function on our actual model, 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 lengths to be 0.55 for all emulators. Normally, the argument `specified_priors` will not be needed, since the correlation lengths are determined by the `emulator_from_data` function itself.

Show: How was the value 0.55 chosen? on P91

```
ems_wave1 <- emulator_from_data(training, names(targets), ranges,
                                specified_priors = list(hyper_p = 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
```

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 $t = 40$:

```
ems_wave1$R40
```

```
## 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); mu; beta1; beta2; epsilon; alpha; gamma; omega; I(mu^2); I(beta1^2)
##   Active variables mu; beta1; beta2; epsilon; alpha; gamma; omega
##   Regression Surface Expectation:  304.9884; -0.9248; 50.5568; 8.9325; 106.9455; -24.1201; 13.4327; -
##   Regression surface Variance (eigenvalues):  0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0
## Correlation Structure:
## Bayes-adjusted emulator - prior specifications listed.
##   Variance (Representative):  2.372002
##   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 0 0 0 0 0 0 0 0 0 0 0 0
```

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

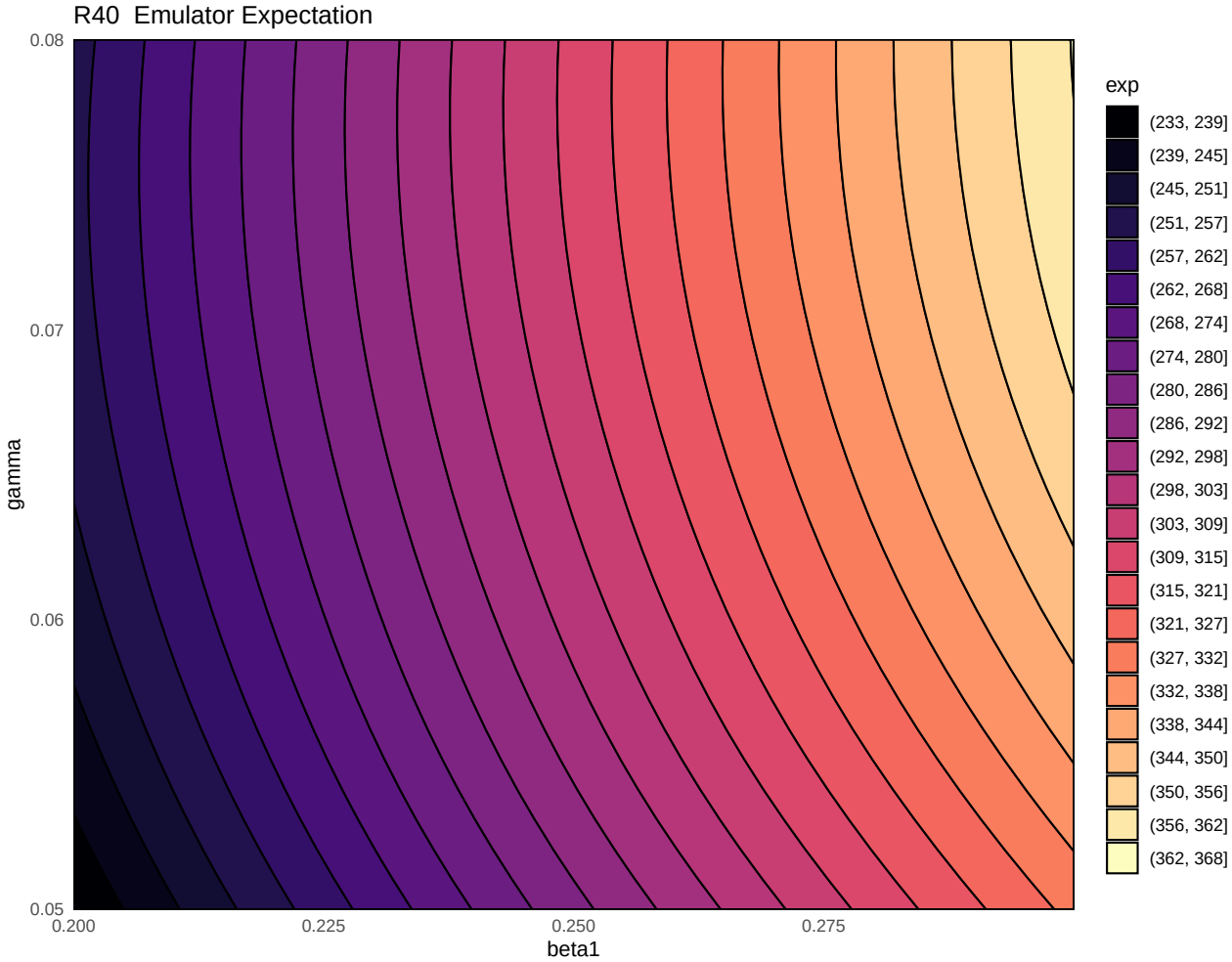
- Active variables: these are the variables that have the most explanatory power for the chosen output.

In our case all variables but b and β_3 are active.

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

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

```
emulator_plot(ems_wave1$R40, params = c('beta1', 'gamma'))
```



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

Task 3

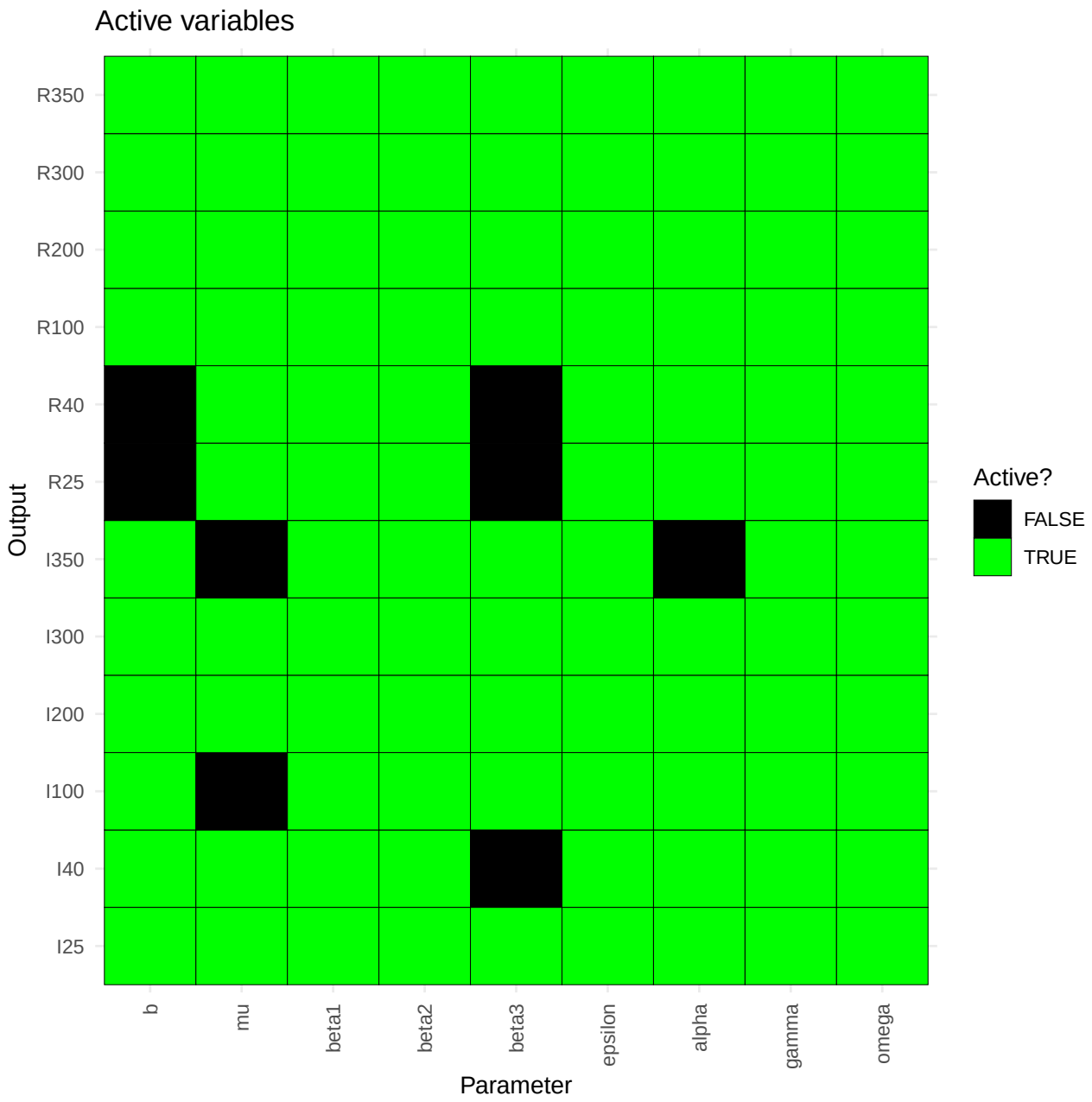
Is β_3 active for all emulators? Why?

Show: R tip on P91

Show: Solution on P59

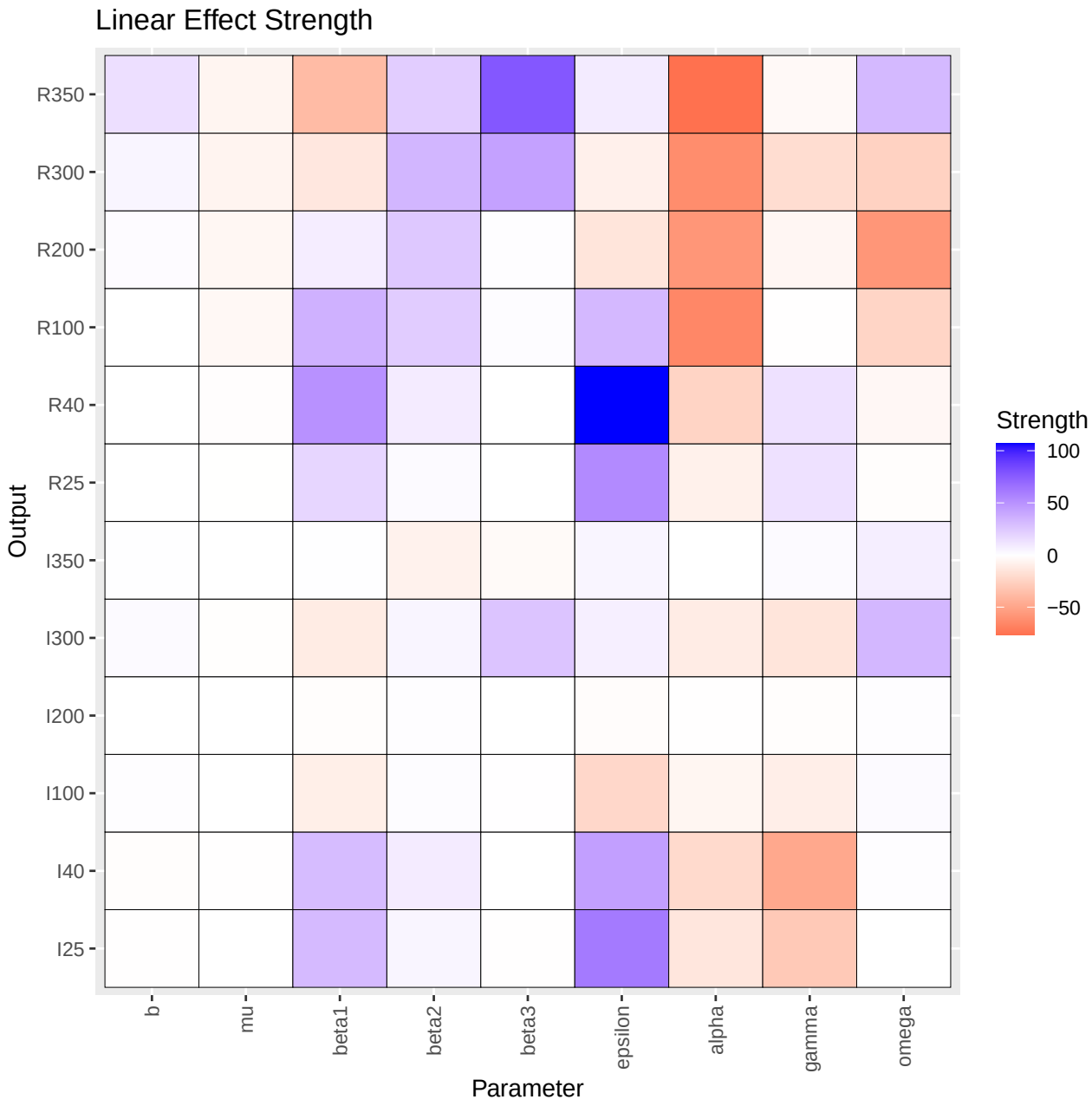
Looking at what variables are active for different emulators is often an instructive exercise. The code below produces a plot that shows all dependencies at once.

```
plot_actives(ems_wave1)
```



From this table, we can see that `mu` is comparatively inactive for numbers of infected individuals, while `b` and `beta3` do not materially affect the number of recovered people at early times. We can also consider the *strength* of the effect of each variable on each output via the `effect_strength` function. It has a number of options, but here we will just focus on linear contributions by setting `quadratic = FALSE`, and plot the results in a grid similar to the above using `grid.plot = TRUE`.

```
effect_strength(ems_wave1, plt = TRUE, grid.plot = TRUE, quadratic = FALSE)
```



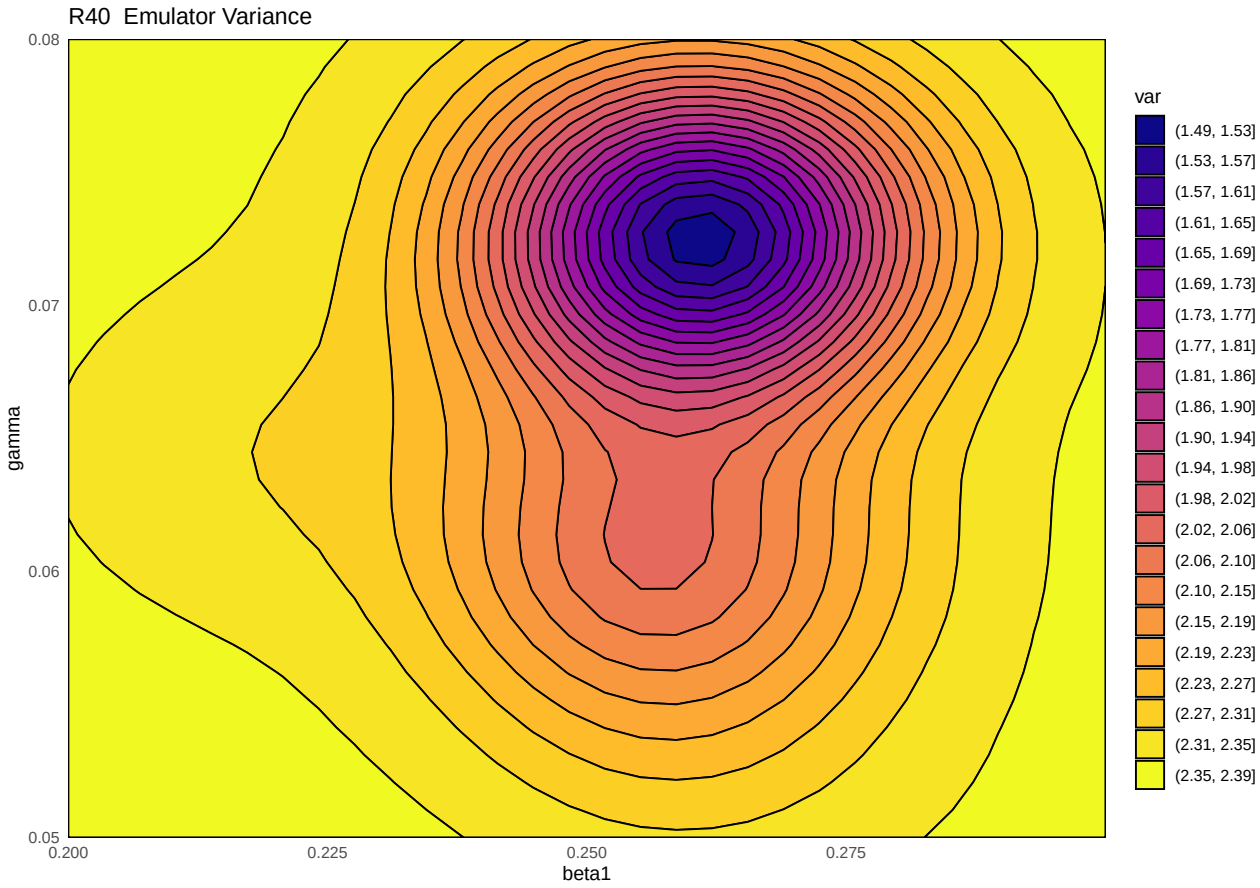

```
##              b        mu      beta1     beta2     beta3    epsilon      alpha
## I25  -0.382948  0.157043  31.677068  4.453202 -0.356442  61.649318 -13.856929
## I40  -1.276443 -0.317668  30.801216  9.085243  0.000000  44.221136 -20.753678
## I100  0.985647  0.000000  -9.244201  1.442522  0.270509 -22.126961  -5.209087
## I200  0.118222 -0.049813  -1.037867  0.951325  0.008836  -1.751816  -0.586806
## I300  2.338526 -0.800178 -10.913042  4.697695 26.979498   7.252356 -10.635246
```

```
## I350  0.484357  0.000000   0.644534 -7.509973 -2.924860   4.638873   0.000000
## R25   0.000000 -0.277618  18.603747  2.056547  0.000000  54.132898  -8.028669
## R40   0.000000 -0.924834  50.556796  8.932483  0.000000 106.945488 -24.120060
## R100  0.168179 -4.209105  35.844887 23.038942  1.504718  32.331620 -65.375844
## R200  1.844695 -4.705837   8.263751 25.214285  0.883775 -14.639224 -57.565784
## R300  4.465658 -6.207118 -13.233523 33.389707 43.168971  -8.201220 -62.444831
## R350 14.384388 -5.602156 -37.632029 22.779135 77.893046   9.283787 -76.107127
##           gamma      omega
## I25  -29.902041   0.174290
## I40  -48.392886   0.780604
## I100  -9.408427   2.250681
## I200  -1.036081   1.112744
## I300 -14.788517  32.895957
## I350   2.111609   7.649332
## R25   13.792157  -1.266613
## R40   13.432741  -4.219517
## R100  -0.352768 -23.646552
## R200  -4.884373 -57.453242
## R300 -18.889645 -25.124285
## R350  -3.382432  32.031470
```

In this plot, a blue square indicates a positive coefficient (so increasing the parameter increases the outputs) while a red square indicates the opposite; the strength of colour indicates the strength of effect. We can see the effect of the active variables in more detail here: `beta1` and `epsilon` have an appreciable positive effect on early-time infections, while `beta2` takes over this role at later times; `alpha`, on the other hand, dampens the number of infections and recoveries as it increases. These observations accord with our expectations; increase in the transmission rates and rate of infection, while if more individuals die from the disease then fewer will be recovering.

We can also look at the uncertainty of the emulators across the space. As mentioned above, `emulator_plot` can also plot the variance of a given emulator:

```
emulator_plot(ems_wave1$R40, plot_type = 'var', params = c('beta1', 'gamma'))
```



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

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

```
summary(ems_wave1$R40$model)$adj.r.squared
```

```
## [1] 0.9994323
```

We see that we have an extremely high value of the adjusted R^2. This means that the regression term explains most of the behaviour of the output $R40$. In particular, the residuals contribute little to the emulator predictions. This is not surprising, considering that we are working with a relatively simple SEIRS model. When dealing with more complex models, the regression term may be able to explain the model output less well. In such cases the residuals play a more important role.

Task 4

Do the residuals have a greater effect if we only consider (up to) linear terms in the variables?

Show: R tip on P91

Show: Solution on P60

Chapter 6

Implausibility

A video presentation of this section can be found here.

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

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

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

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

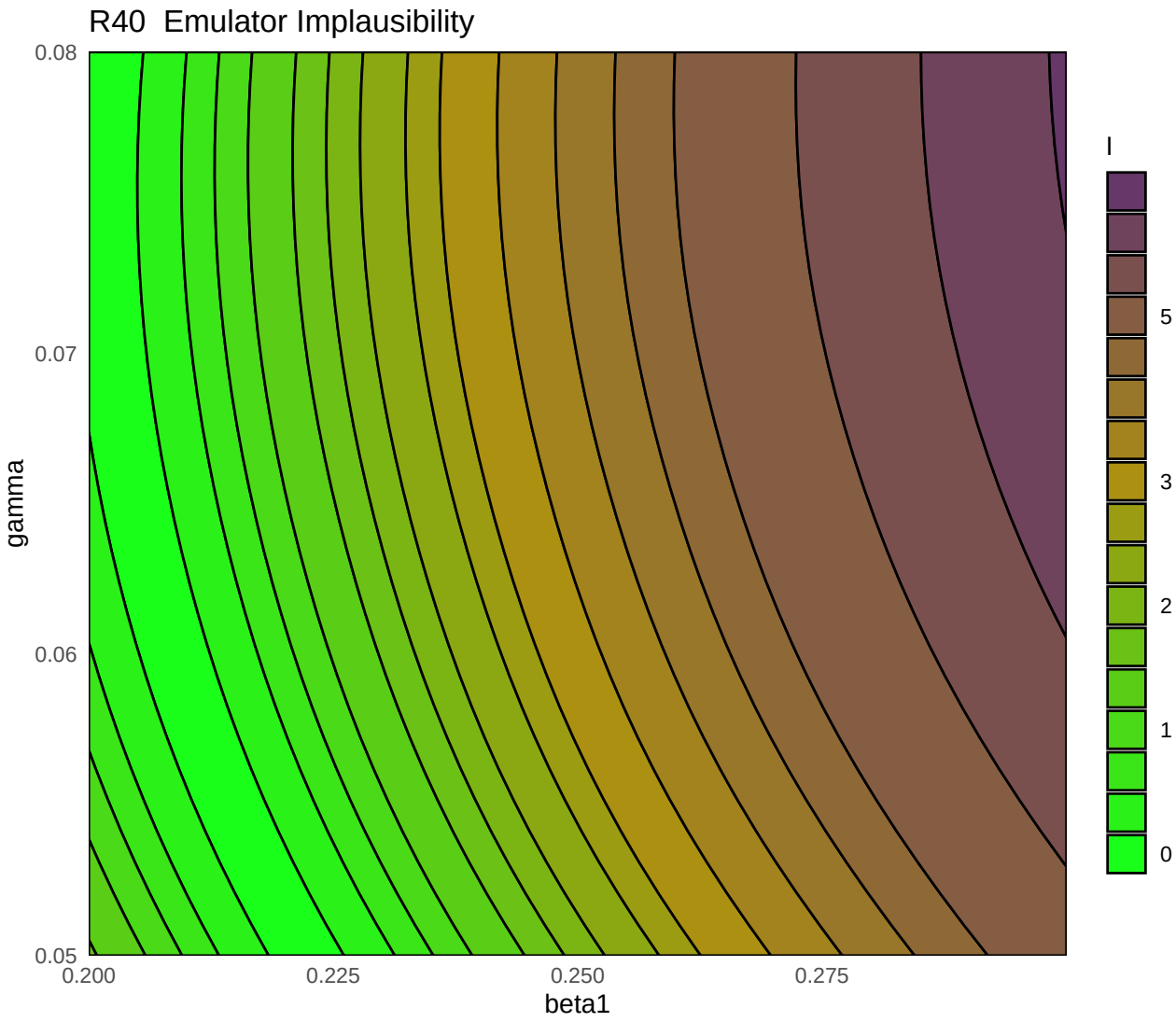
Since in this case study we want to emulate our model, without reference to a real-life analogue, the model represents the reality perfectly. For this reason we have $V_m = 0$. Similarly we have $V_s = 0$, since our model is deterministic. The observation uncertainty V_0 is represented by the 'sigma'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

A very large value of $\text{Imp}(x)$ 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 it further in the next wave.

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

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

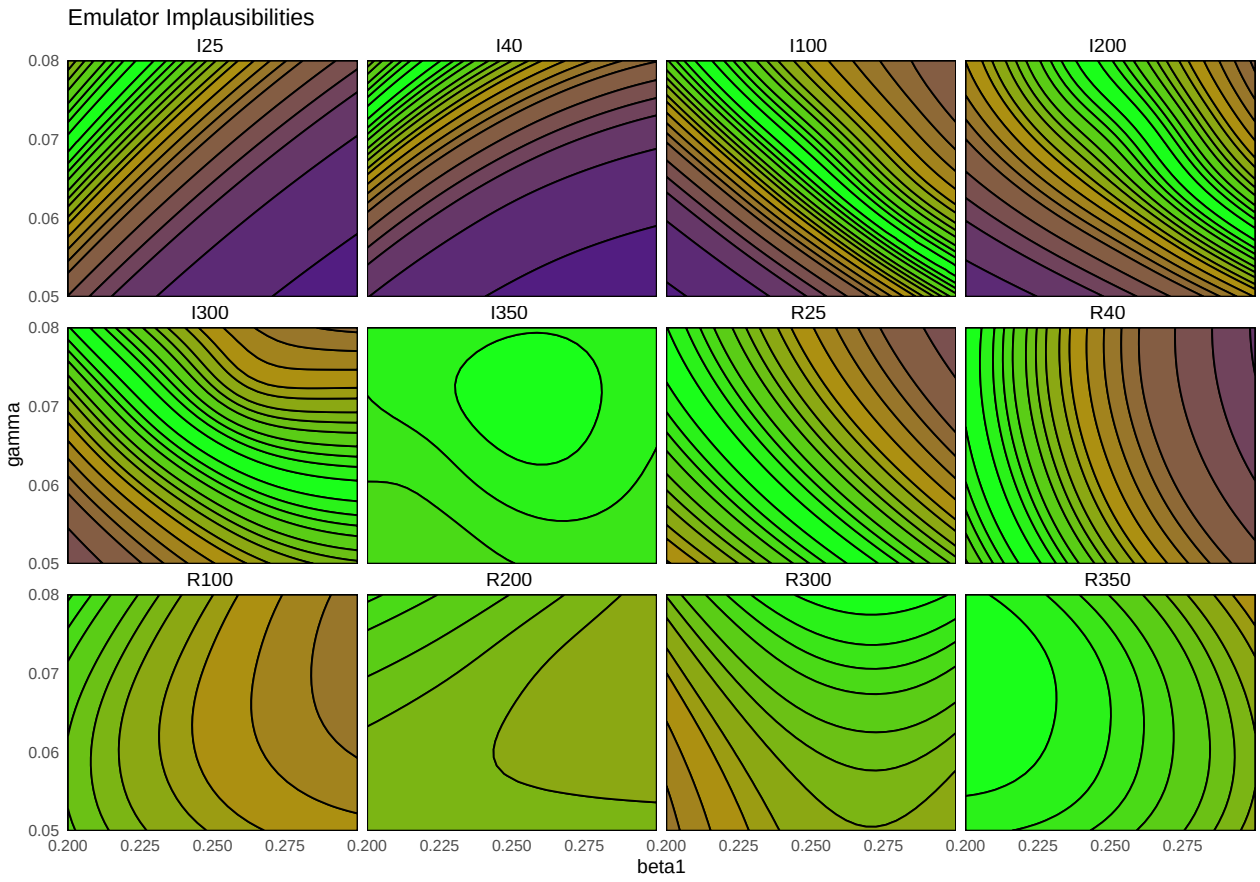
```
emulator_plot(ems_wave1$R40, plot_type = 'imp',
              targets = targets, params = c('beta1', 'gamma'), cb=TRUE)
```



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

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

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



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

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

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

Task 5

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 P60

Chapter 7

Emulator diagnostics

A video presentation of this section can be found here.

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.

R tip

The discussion below is very useful for understanding *why* emulators are not performing well, and allows us to make informed judgements about our training data, and which parts of parameter space we are failing to represent well. However, this can be a slow process when we have many outputs to emulate: `hmer` can automate this process for determinstic emulators using the `diagnostic_pass` function. It will:

- Check for structured input errors (i.e. errors which depend on the part of parameter space we are in);
- Check for structured output errors (those which depend on the scale of the output);
- Check misclassification errors as below;
- Check comparison diagnostics as below.

If any of these checks flag possible issues, it makes the appropriate adjustments to the emulator structure; if it still cannot produce a good emulator after these changes, it discards that output.

For our example, we can use this with the following command:

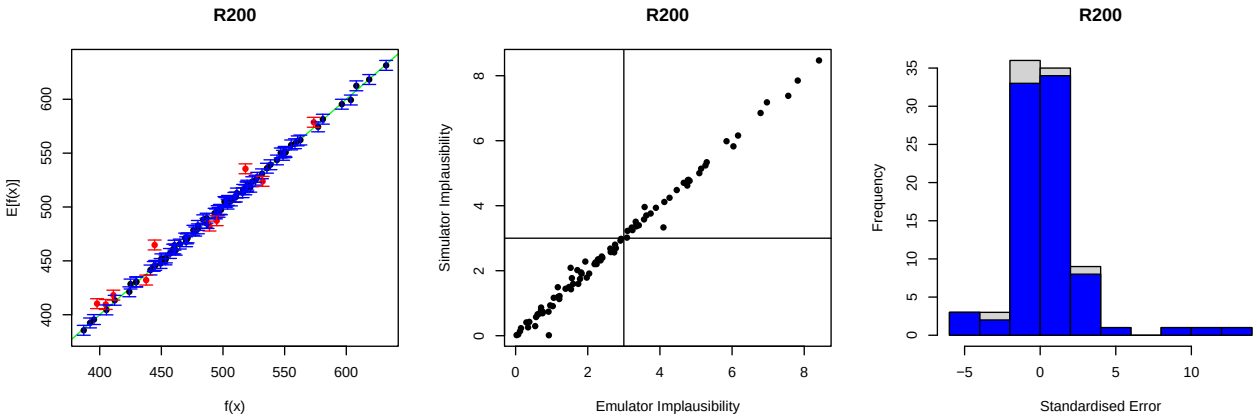
```
adjusted_ems_wave1 <- diagnostic_pass(ems_wave1, targets, validation, verbose = TRUE)
```

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

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

The function `validation_diagnostics` provides us with three diagnostics, addressing the three questions above. We will focus on the output `R200` to demonstrate some common features.

```
vd <- validation_diagnostics(ems_wave1$R200, validation = validation, targets = targets, plt=TRUE,
                             row = 1)
```



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

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

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

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

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

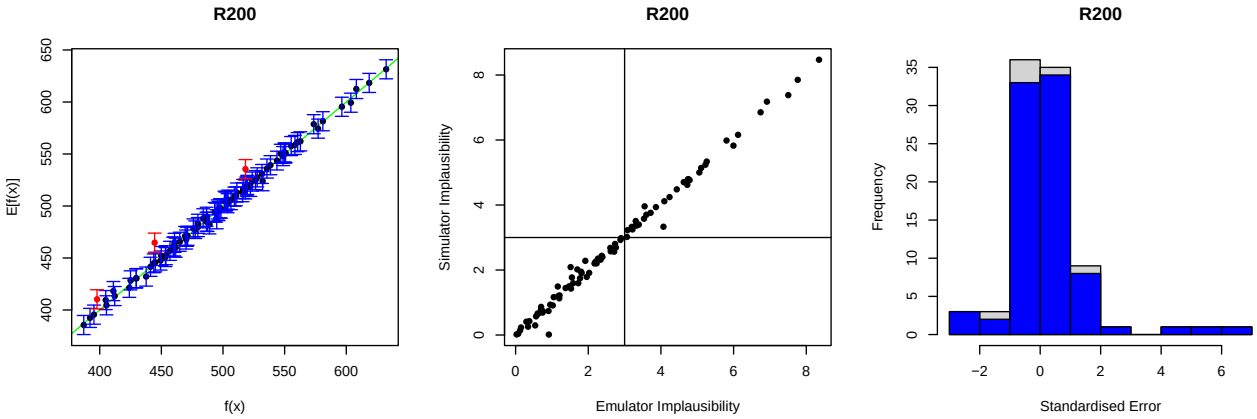
The diagnostics in the plot above are not particularly bad, but we will try to modify our emulator to make it more conservative to avoid the large number of points for which the emulator is overconfident (the red error-bars in the first column).

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

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

The lower the value of σ, the more 'certain' the emulator will be. This means that when an emulator is a little too overconfident (as in our case above), we can try increasing σ. Below we train a new emulator setting σ to be 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. While this is not particularly clear in the middle column, the error-bars in the first column have been widened, resulting in fewer validation points showing as red, and the scale on the standardised errors (third column) is changed quite substantially.

Task 6

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

Show: R tip on P91

Show: Solution on P64

Chapter 8

Proposing new points

A video presentation of this section can be found here.

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

STEP 1. If prior parameter sets are provided, step 1 is skipped, otherwise a set is generated using a Latin Hypercube Design, rejecting implausible parameter sets. Figure 8.1 shows an example of LH sampling (not for our model), with implausible parameter sets in red and non-implausible ones in blue:

STEP 2. Pairs of parameter sets are selected at random and more sets are sampled from lines connecting them, with particular importance given to those that are close to the non-implausible boundary. Figure 8.2 shows line sampling on top of the previously shown LH sampling: non-implausible parameter sets provided by LH sampling are in grey, parameter sets proposed by line sampling are in red if implausible and in blue if non-implausible.

STEP 3. Using these as seeding points, more parameter sets are generated using importance sampling to attempt to fully cover the non-implausible region. Figure 8.3 has non-implausible parameter sets provided by LH and line sampling in grey and non-implausible parameter sets found by importance sampling in blue.

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

Let us generate 180 new sets of parameters, using the emulators for the time up to $t = 200$. 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 for 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_design(restricted_ems, 180, targets, verbose=TRUE)
```

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

Note that, depending on the complexity of the calibration task, a Latin Hyper-cube may not be able to find any parameter sets with implausibility below three. If that happens, `generate_new_design` will perform step 1 (Latin Hyper-cube sampling) at a higher implausibility threshold, to find a space-filling design. Using

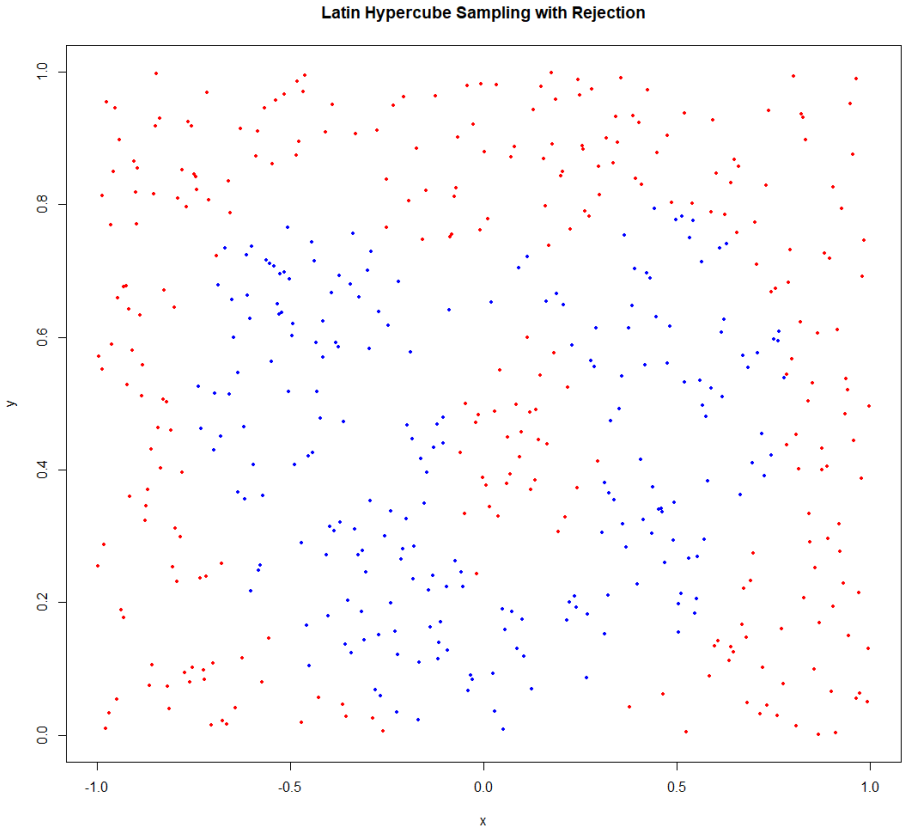


Figure 8.1: LH sampling

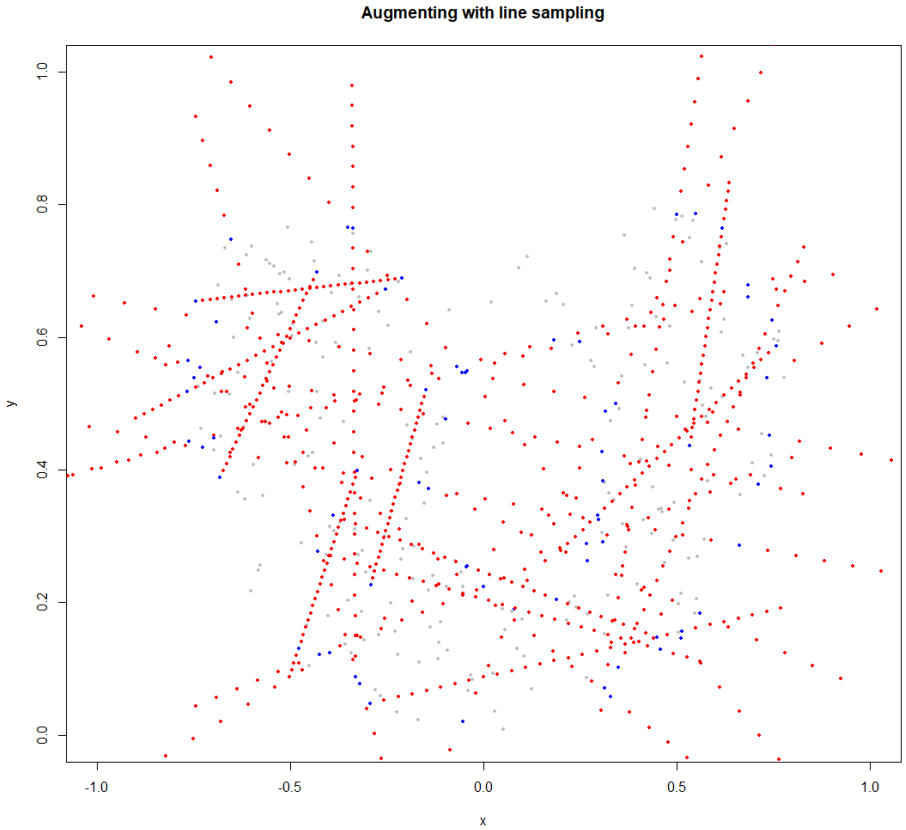


Figure 8.2: Line sampling

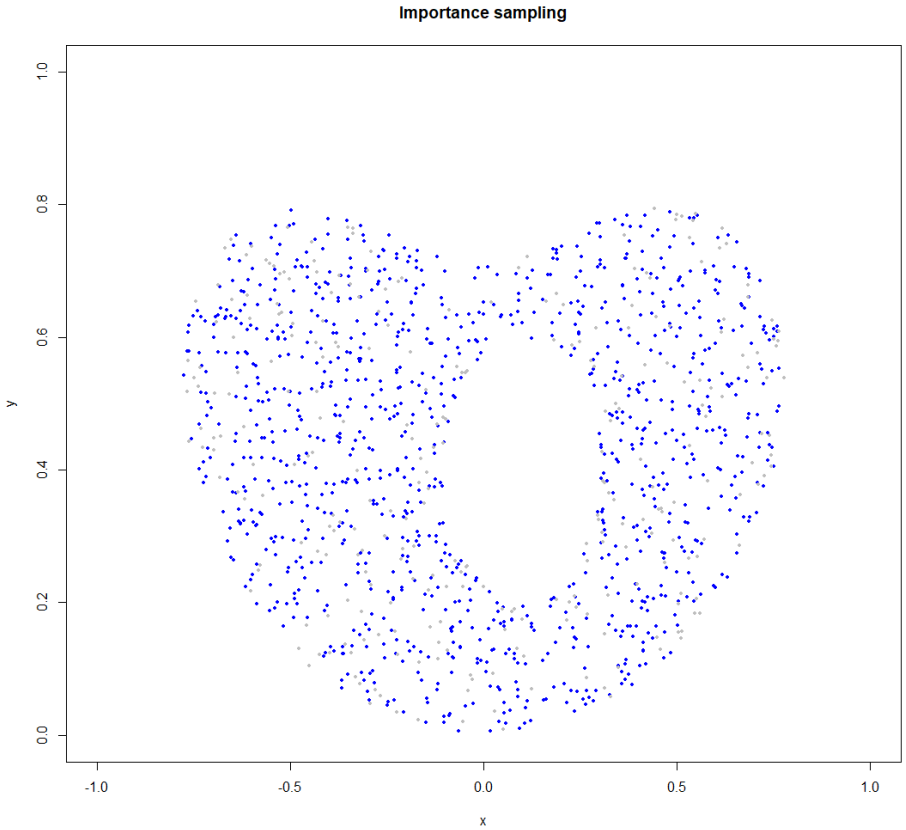


Figure 8.3: Importance sampling

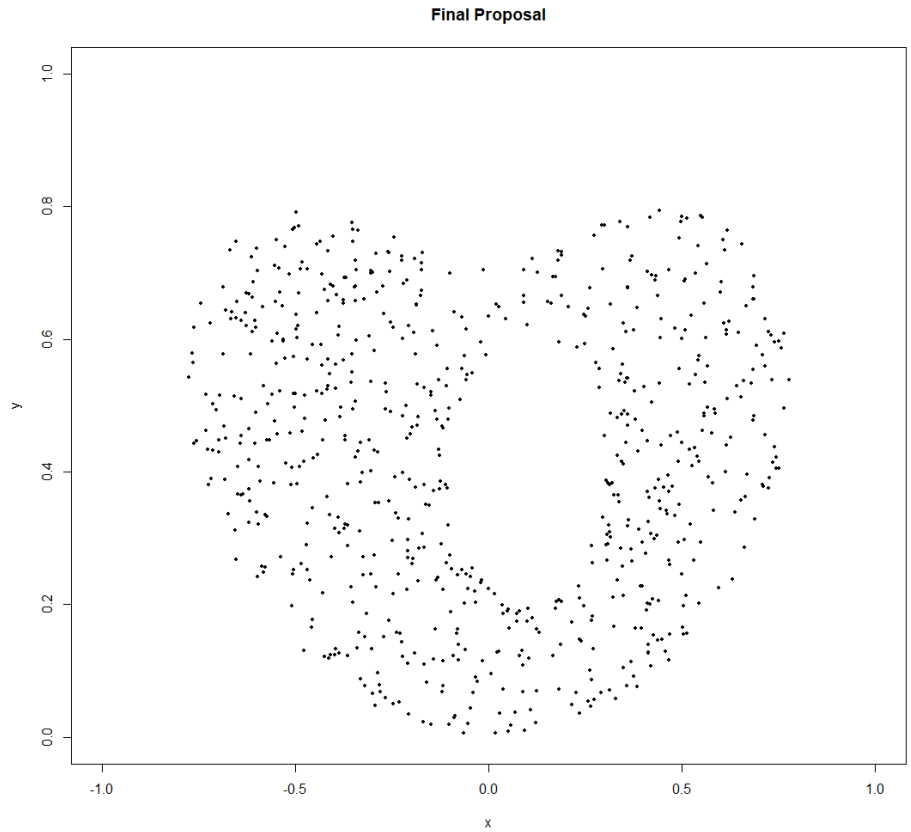
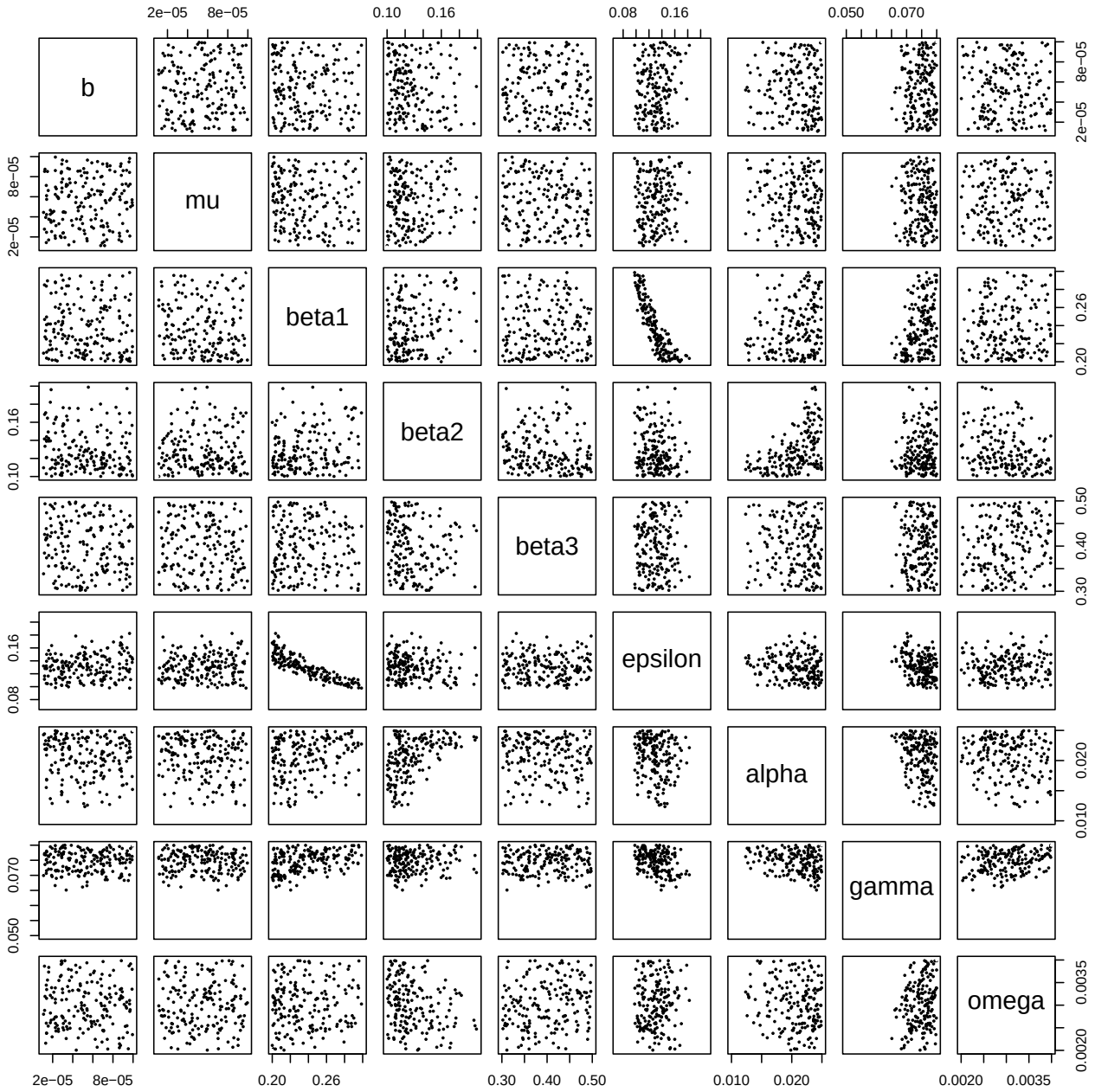


Figure 8.4: Overall design of new non-implausible parameter sets

this as a starting point, step 2 (line sampling) and step 3 (importance sampling) are then performed. From this proposal, a subset of lower-implausibility parameter sets are selected and steps 2 and 3 are repeated. This process iterates until either the desired implausibility has been reached or the process has reached a barrier to further reductions in implausibility.

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

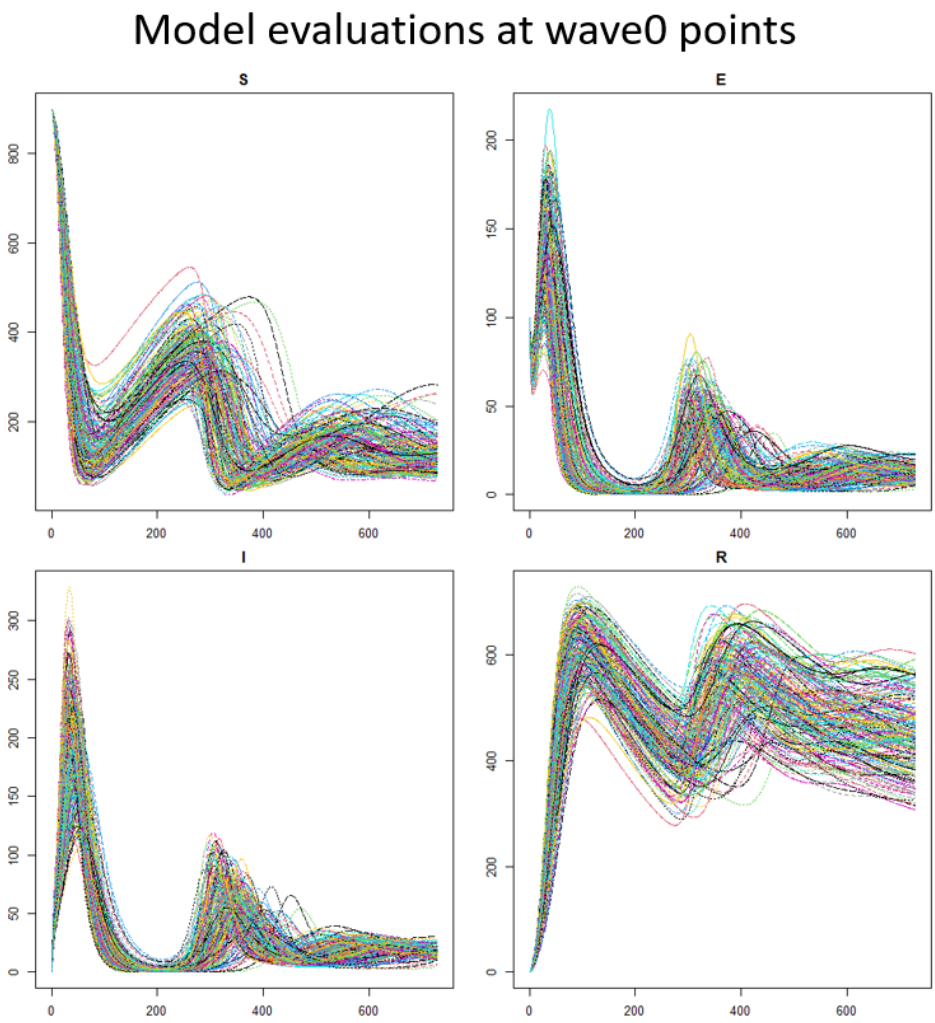
```
plot_wrap(new_points_restricted, ranges)
```



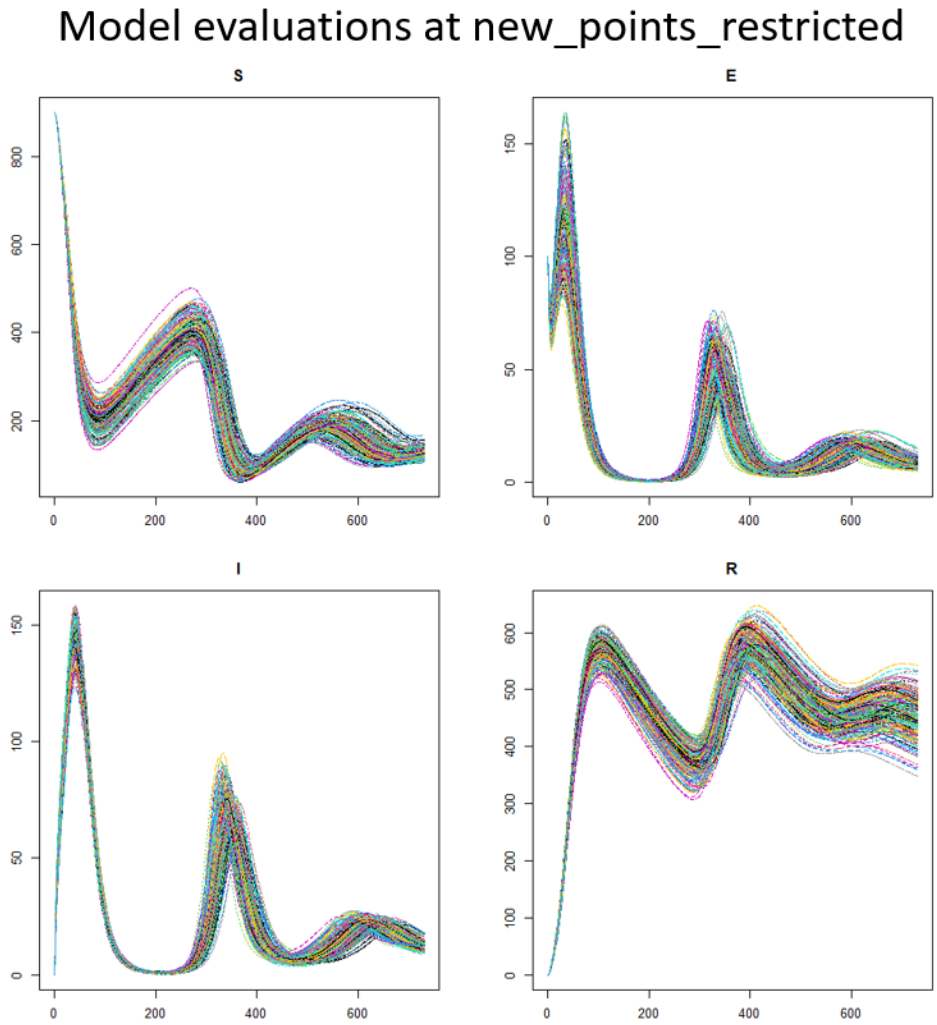
By looking at the plot we can learn a lot about the non-implausible space. For example, it seems clear that low values of γ cannot produce a match (cf. penultimate column). We can also deduce relationships between parameters: β_1 and ϵ are an example of negatively-correlated parameters. If β_1 is large 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. In the R-script, a function `plot_runs` is defined to produce plots as above. For example, to plot runs from the initial points, you can type `plot_runs(initial_points)`. Note that this functions require the dataframe to have exactly 180 points.

Task 7

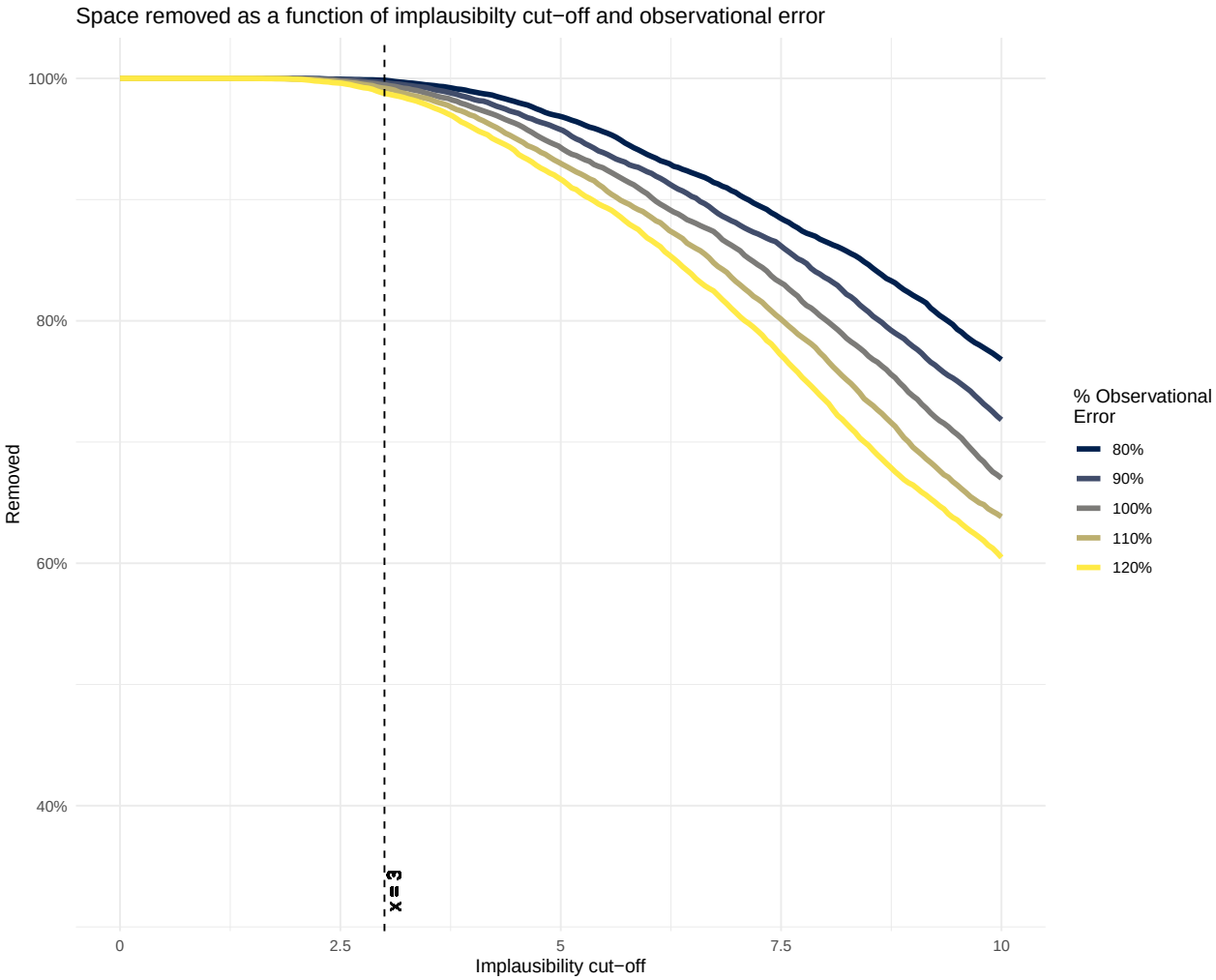
What do you think would happen if we tried generating new parameter sets using all emulators, instead of those relating to times up to $t = 200$ only? Try changing the code and seeing what happens.

Show: Solution on P65

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 8

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

Show: Solution on P67

Chapter 10

Second wave

A video presentation of this section can be found here.

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. We conclude the section with a short discussion on when and how to customise the value of the correlation lengths.

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

Now note that parameter sets in `new_points` tend to lie in a small region inside the original input space, since `new_points` contains only non-implausible points, according to the first wave emulators. The first caveat is then that it is preferable to train the new emulators only on the non-implausible region found in wave one. This can be done simply setting the argument `check.ranges` to `TRUE` in the function `emulator_from_data`.

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

Task 9

Using `new_points` and setting `check.ranges` to `TRUE`, train new emulators. Perform diagnostic checking (perhaps with `diagnostic_pass`), customise the emulators and generate new parameter sets.

Show: Solution on P76

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

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

```
##       I25       I40      I100      I200      I300      I350       R25       R40
## 0.9998404 0.9996392 0.9986963 0.9974565 0.9967800 0.9659110 0.9999257 0.9996721
##      R100      R200      R300      R350
## 0.9995921 0.9997507 0.9986806 0.9972370
```

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

In practice, if you do not pass a value for it, θ is chosen very carefully by the hmer package, and most users calibrating deterministic models will not have to vary the value of θ for most of their emulators. If, however, you find that the non-implausible space is shrinking very slowly, particularly in later waves (see section 11 for details of how to check this), then the value of θ may be too conservative. If this occurs, then you can increase the θ of the emulators to increase the rate at which space is reduced. You should only do this if you are confident that your outputs are smooth enough to justify the choice of θ however, or you risk the emulators incorrectly excluding space when model fits could be found. We discuss the choice of θ further in our workshop on calibrating stochastic models.

Chapter 11

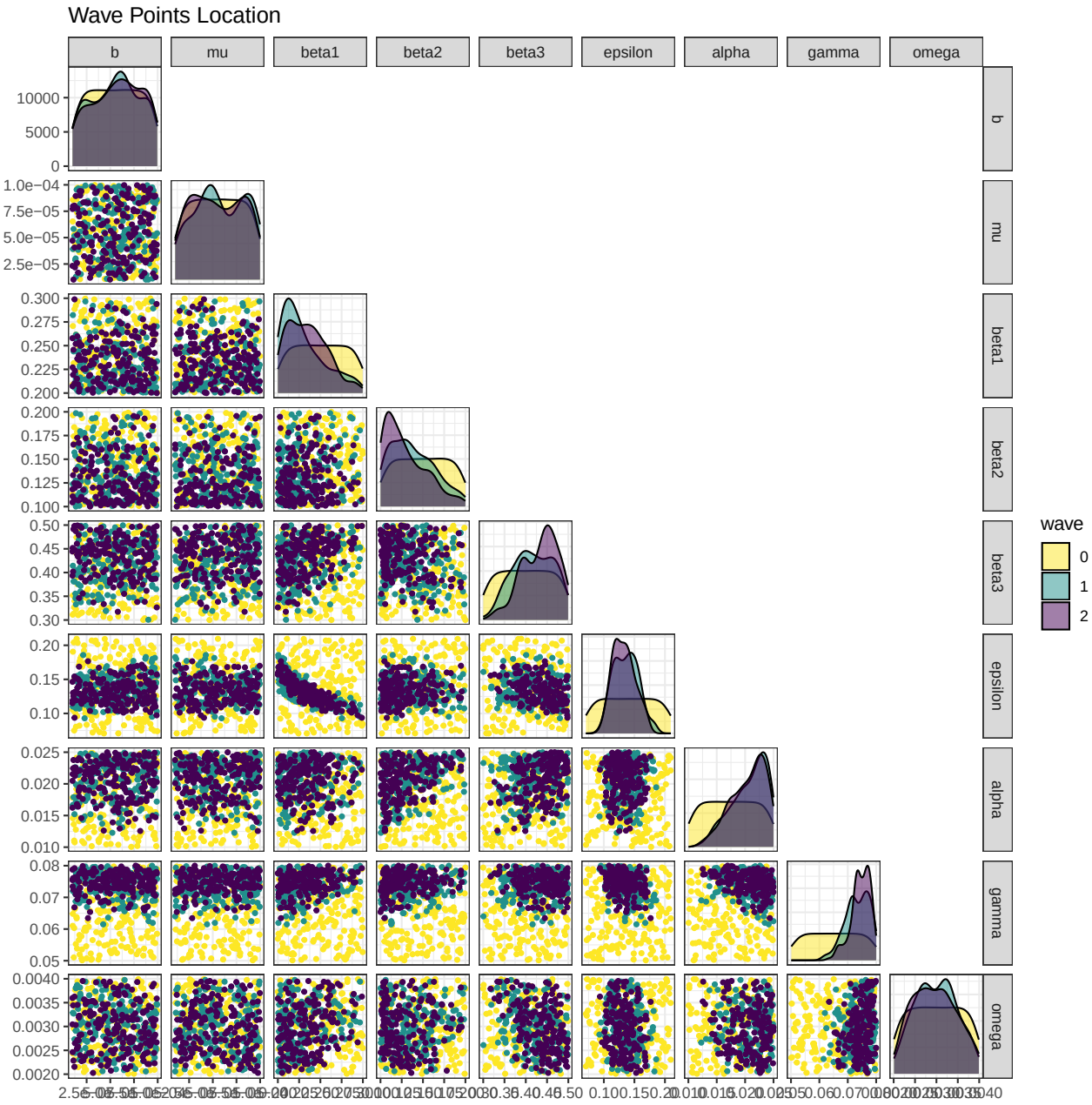
Visualisations of non-implausible space by wave

A video presentation of this section can be found here.

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

The first visualisation, obtained through the function `wave_points`, shows the distribution of the non-implausible space for the waves of interest. For example, let us plot the distribution of 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)
```



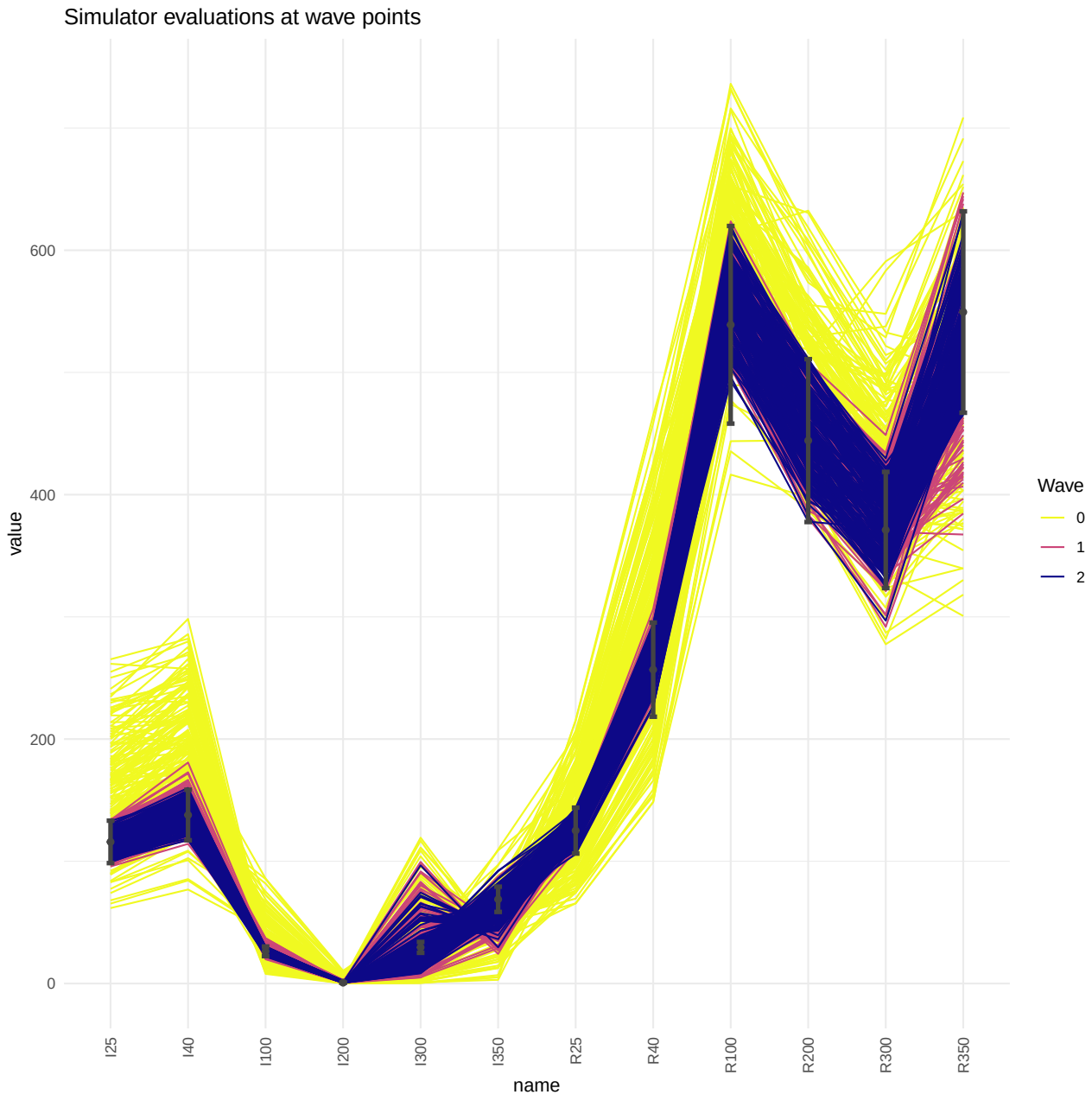
Here `initial_points` are in yellow, `new_points` are in green and `new_new_points` are in purple. We set `p_size` to 1 to make the size of the points in the plot smaller. The plots in the main diagonal show the distribution of each parameter singularly: we can easily see that the distributions tend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

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

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

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

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



We can see that, compared to the space-filling random parameter sets used to train the first emulators, the new parameter sets are in much closer agreement with our targets. While 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.

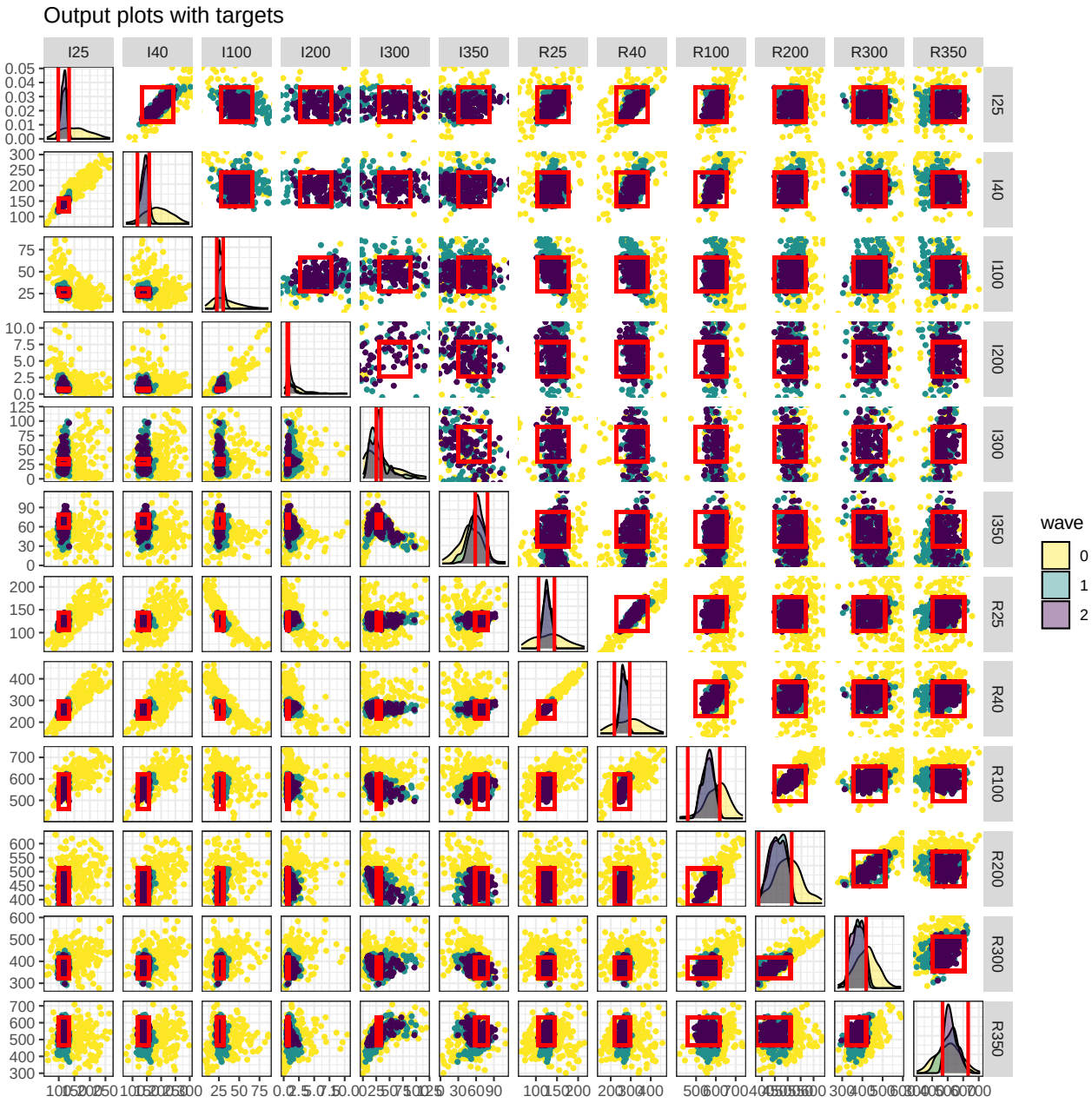
Task 10

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

Show: Solution on P85

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. Of course, more waves are required, in order to complete the calibration task. 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 simply use the current emulators and 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, or input and/or output ranges.

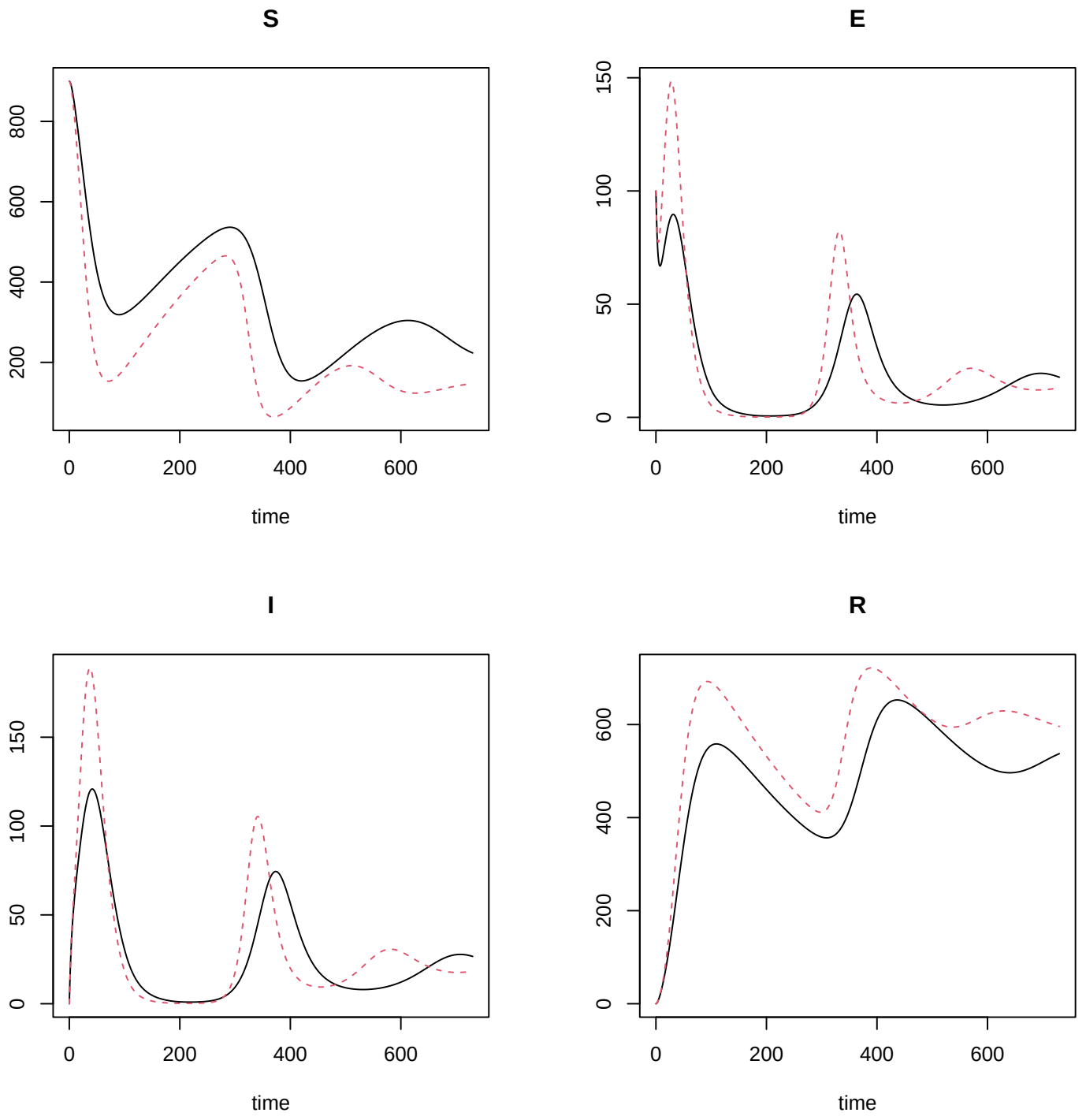
Appendix A

Answers

Solution 1

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

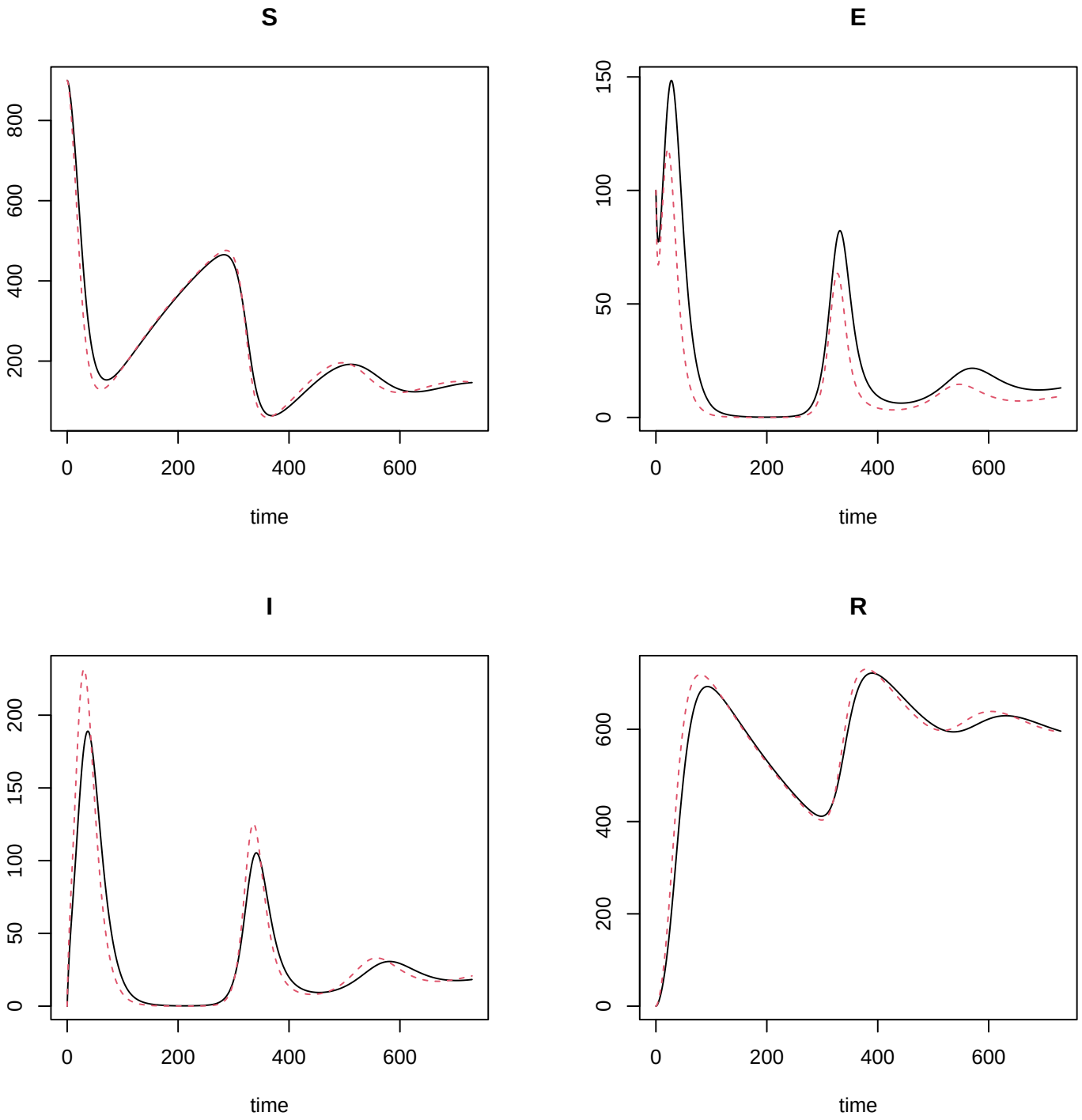
```
higher_foi_params <- c(
  b = 1/(60*365),
  mu = 1/(76*365),
  beta1 = 0.3, beta2 = 0.1, beta3 = 0.5,
  epsilon = 0.13,
  alpha = 0.01,
  gamma = 0.08,
  omega = 0.003
)
higher_foi_solution <- ode_results(higher_foi_params)
plot(solution, higher_foi_solution)
```



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)
```

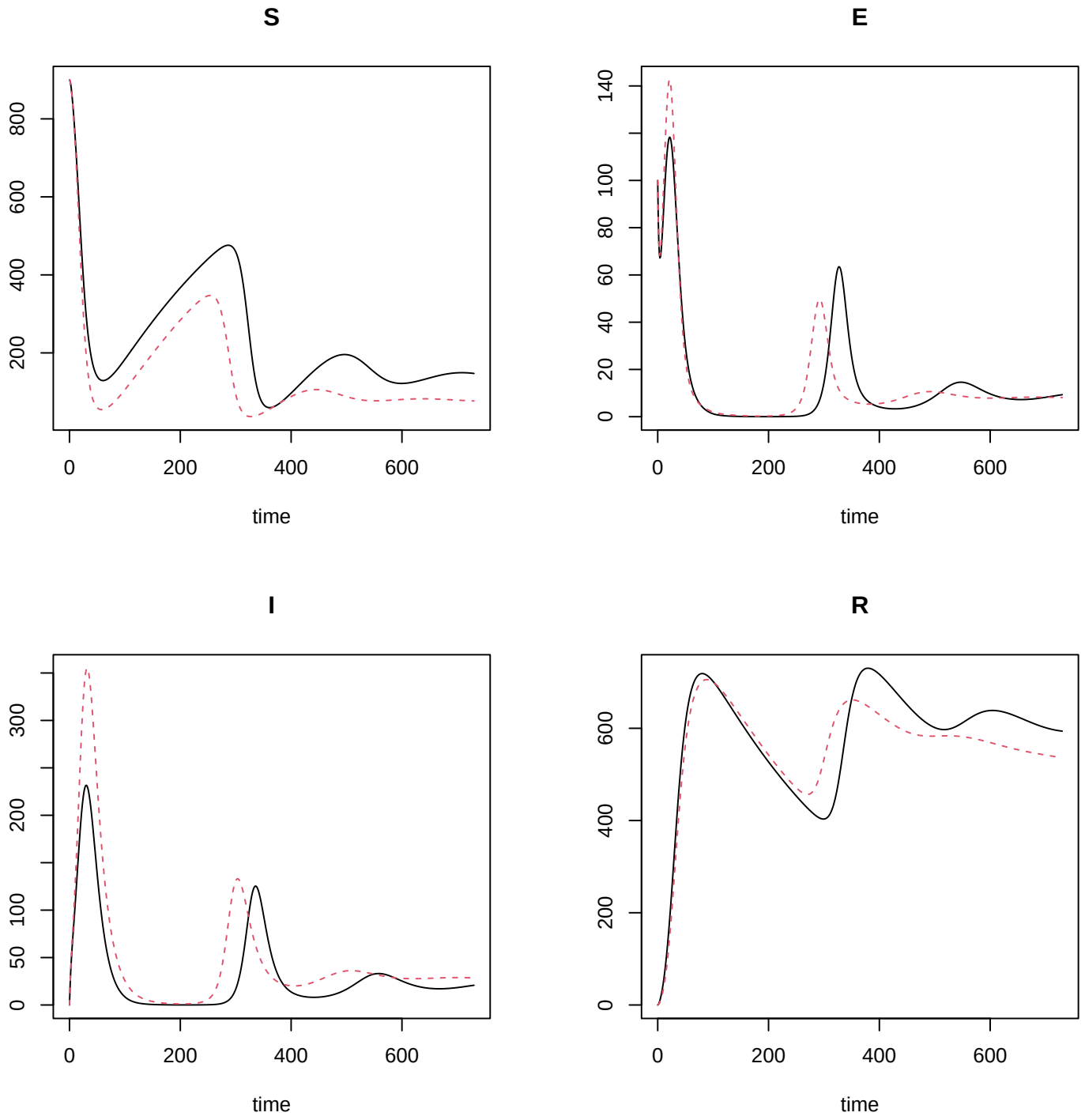


Here the black line is the model output when the model is run with the previous parameter set, and the red dotted

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 P12

Solution 3

Let us take a look at the emulators. To show what variables are active for an emulator 'em' we can simply type `em$active_vars`. For example:

```
ems_wave1$R40$active_vars
```

```
## [1] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
```

shows what variables are active for the $R40$ emulator. Since β_3 is the fifth parameter (see for example `ranges`), then we can type

```
ems_wave1$R40$active_vars[5]
```

```
## [1] FALSE
```

to just focus on β_3. To look at the role of β_3 in all emulators at once, we create a logical vector looping through `ems_wave1`:

```
beta3_role <- logical()
for (em in ems_wave1) beta3_role <- c(beta3_role, em$active_vars[5])
names(beta3_role) <- names(ems_wave1)
beta3_role
```

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

Here we see that β_3 tends to be more active at later times. This is in fact what we would expect: the later infection/recovery rates don't have an impact on early time outputs.

Return to task on P23

Solution 4

We could train emulators to each output, but let us just focus on the recovered individuals at $t = 40$ here.

```
linear_r40_em <- emulator_from_data(training, c('R40'), ranges, order = 1,
                                    specified_priors = list(hyper_p = c(0.55)))$R40
summary(linear_r40_em$model)$adj.r.squared
```

```
## [1] 0.9733467
```

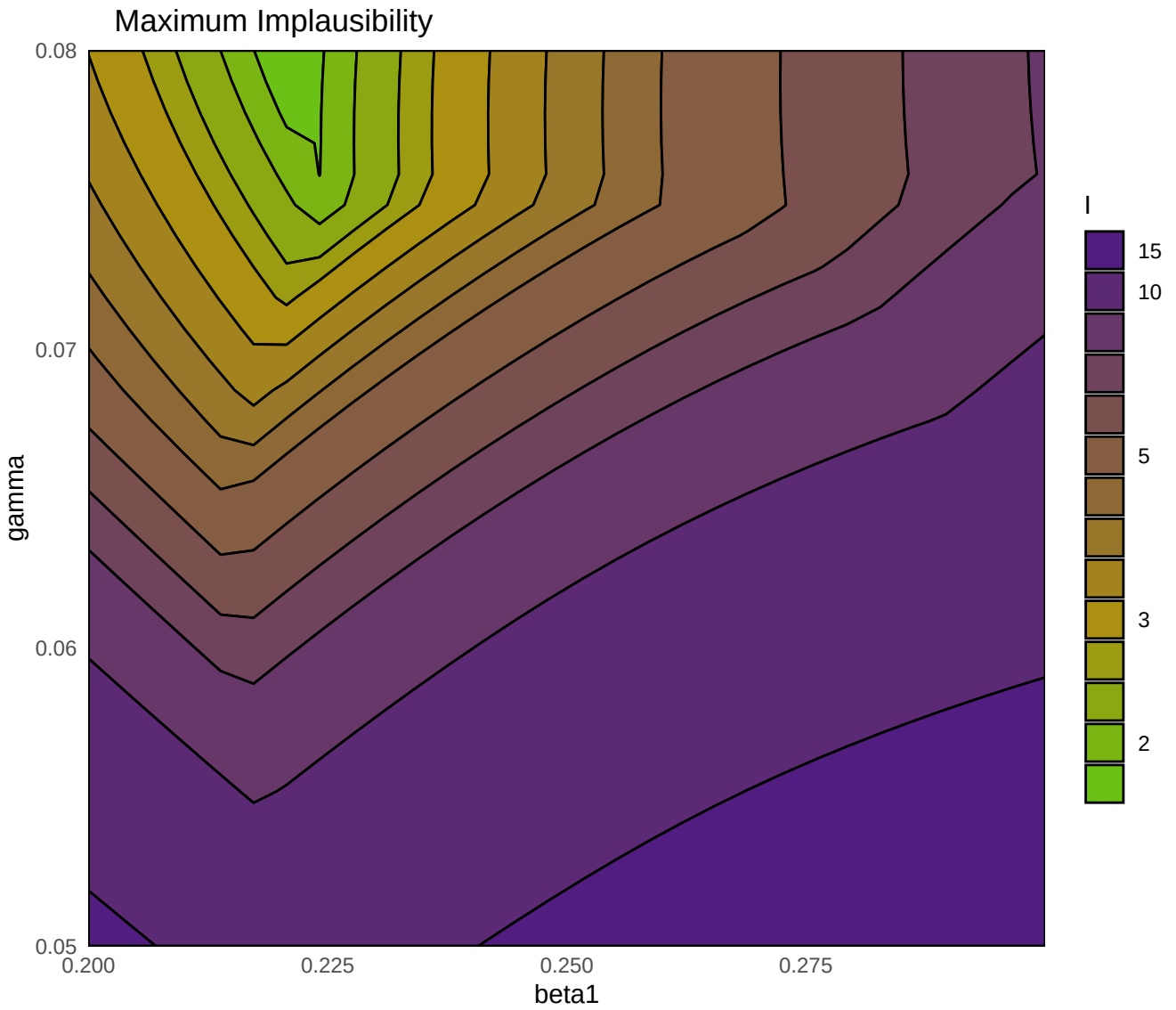
We can see that the adjusted R^2 has decreased, but is still high; in some circumstances we might therefore choose to use the linear emulators as they use fewer terms to fit to the data without a substantial reduction in accuracy.

Return to task on P26

Solution 5

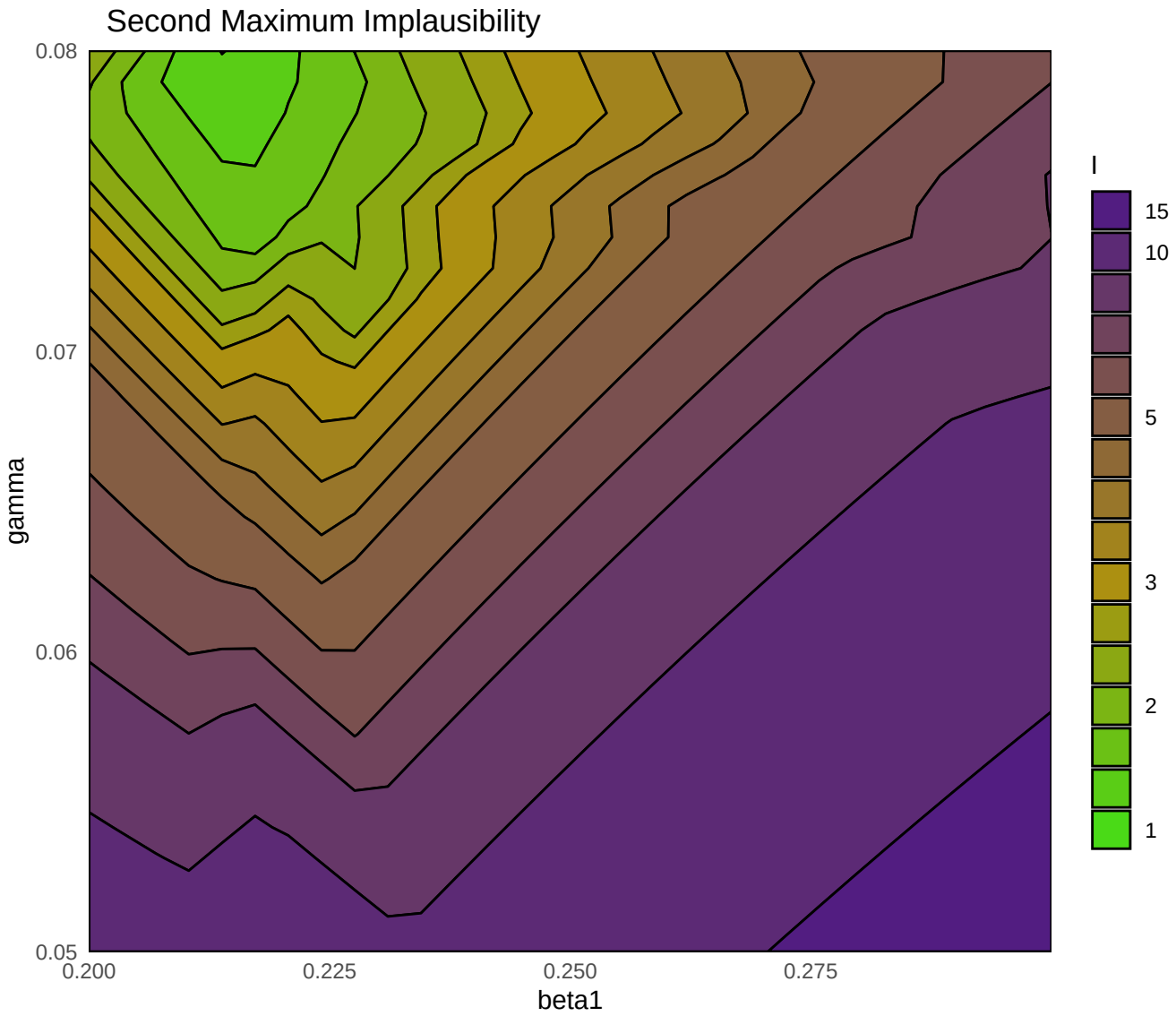
Let us start by visualising the maximum implausibility passing all emulators to `emulator_plot` and setting `plot_type='nimp'`:

```
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 will be less uncertain. To work with second-maximum implausibility we simply add nth=2 to the previous function call:

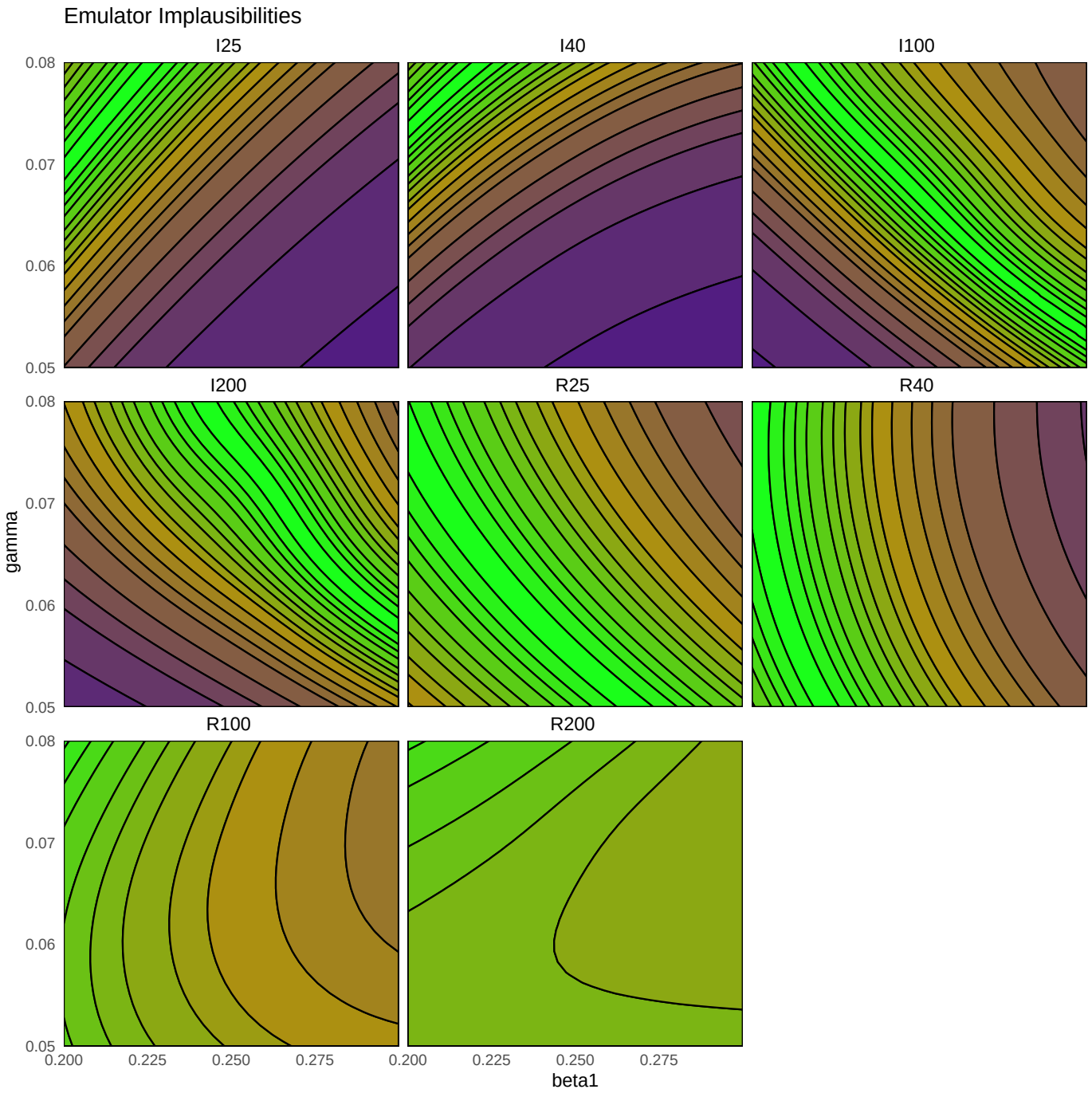
```
emulator_plot(ems_wave1, plot_type = 'nimp', targets = targets, params = c('beta1', 'gamma'), cb=T, nth=2)
```



One of the advantages of history matching and emulation is that we are not obliged to emulate all outputs at each wave. This flexibility comes in handy, 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 at a later wave. 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 particularly convenient: 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 (up to $t = 200$), and produce implausibility plots for them:

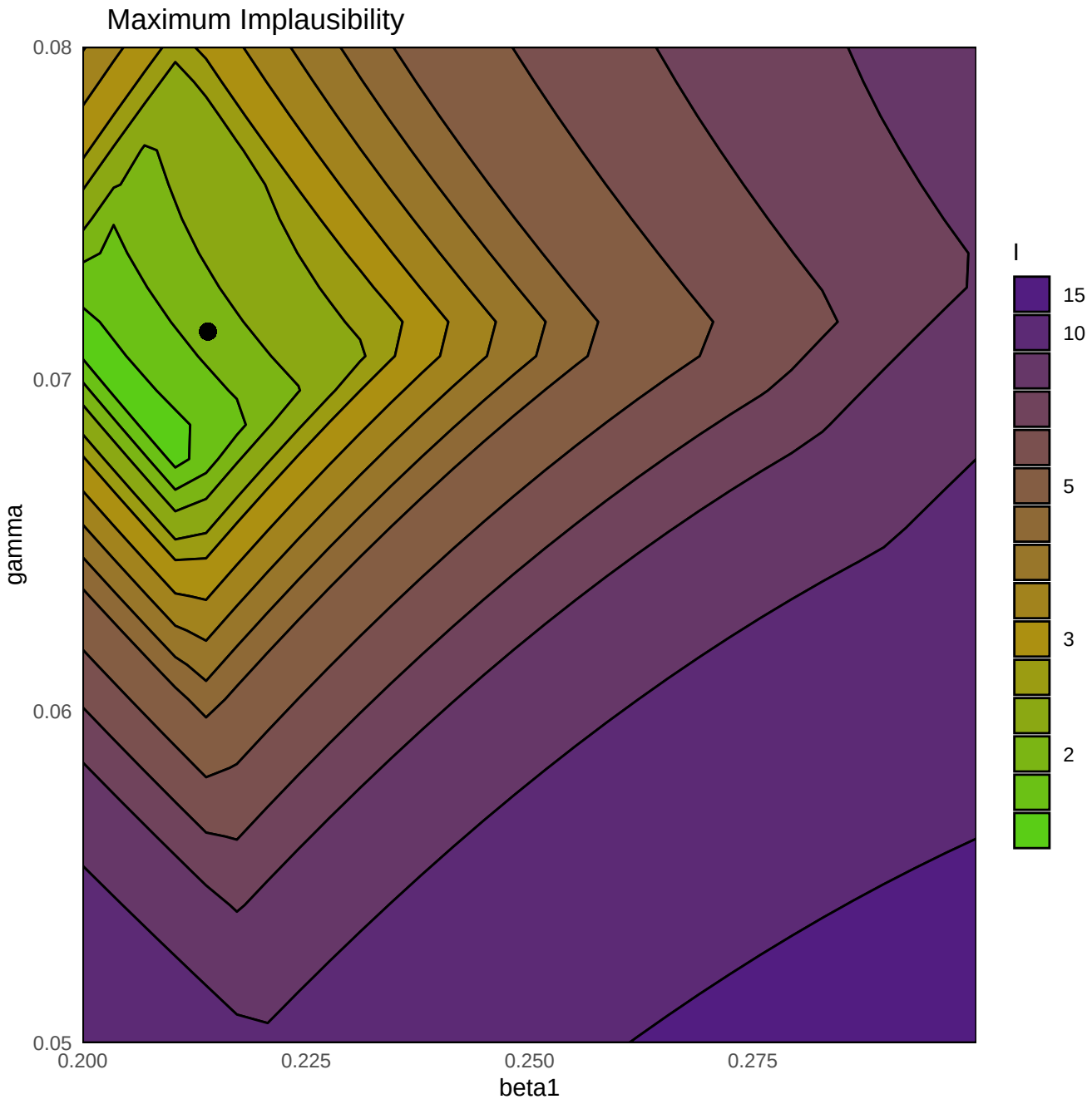
```
restricted_ems <- ems_wave1[c(1,2,3,4,7,8,9,10)]
emulator_plot(restricted_ems, plot_type = 'imp', targets = targets,
              params = c('beta1', 'gamma'), cb=T)
```



Finally let us set the unshown parameters to be as in `chosen_params`:

```
emulator_plot(restricted_ems, plot_type = 'nimp', targets = targets[c(1,2,3,4,7,8,9,10)],
              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)
```

```
## Warning in geom_point(aes(x = 0.214, y = 1/14), size = 3): All aesthetics have length 1, but the data ha
## i Please consider using `annotate()` or provide this layer with data containing
##   a single row.
```



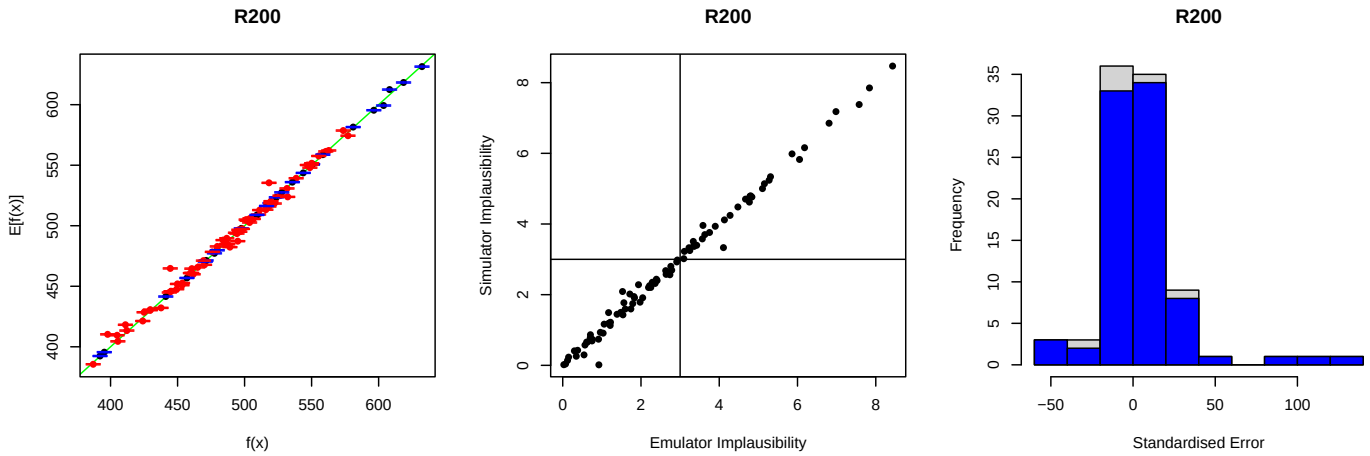
The plot shows what we expected: when β_1 and γ are equal to their values in `chosen_params`, i.e. 0.214 and $1/14$, the implausibility measure is well below the threshold 3 (cf. black point in the box). Note that when working with real models, one usually cannot check if the implausibility is low around fitting parameters, simply because these are not known. However, if one happens to have first hand fitted the model and has therefore a set of fitting parameters, then the above check can be performed.

Return to task on P29

Solution 6

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

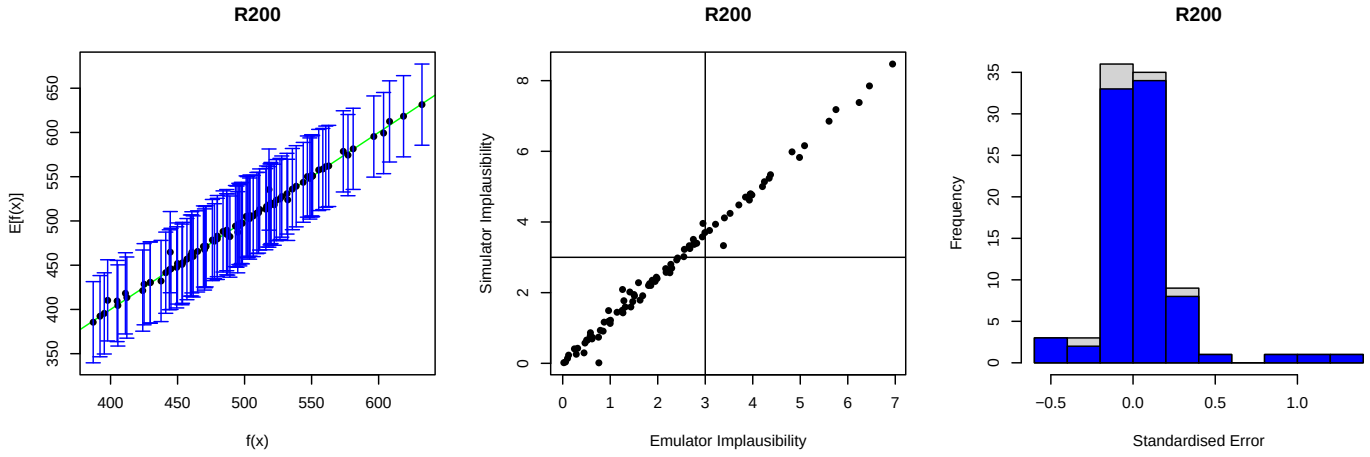
```
tinysigma_emulator <- ems_wave1$R200$mult_sigma(0.1)
vd <- validation_diagnostics(tinysigma_emulator, validation = validation, targets = targets,
                             plt=TRUE)
```



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 2.

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. Note that there is not a universal rule to navigate this tradeoff: the role of the scientist's judgement is fundamental.

Return to task on P33

Solution 7

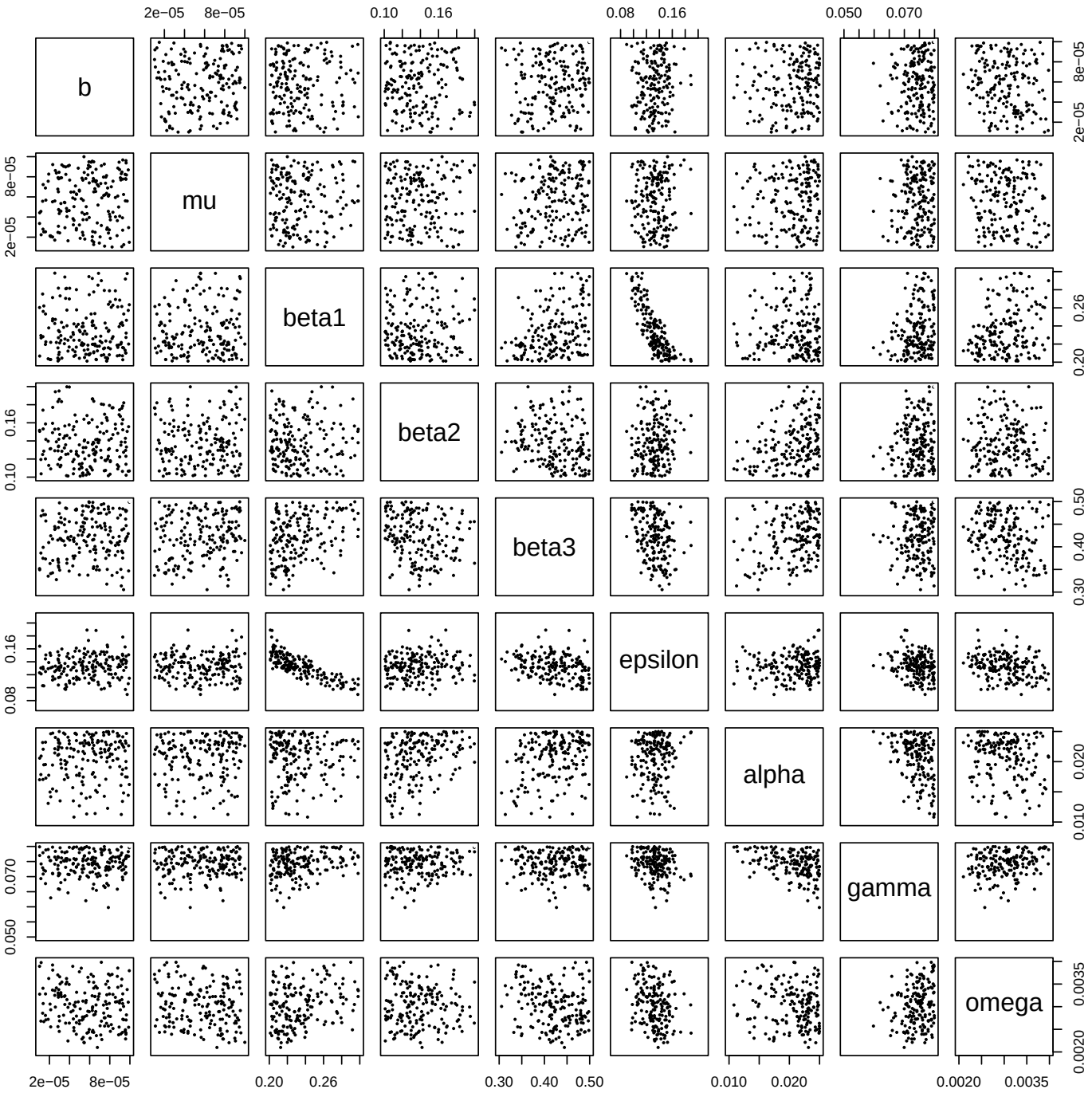
We expect that using all emulators (instead of just a subset of them) will make the non-implausible space smaller than before, since more conditions will have to be met. Let us check if our intuition corresponds to what the plots

show:

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

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

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



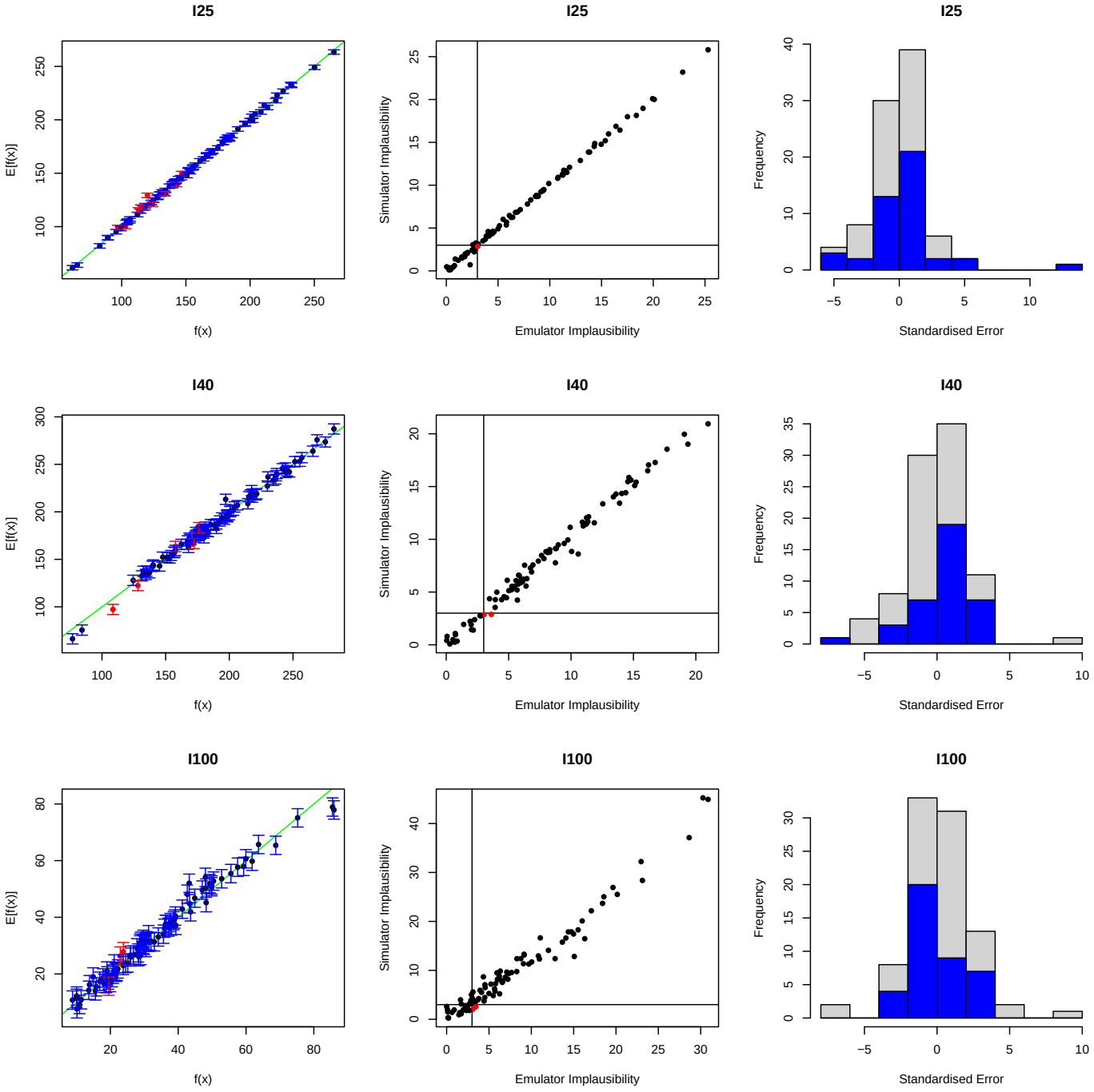
The answer is yes: points now look slightly less spread than before, i.e. a higher part of the input space has been cut-out.

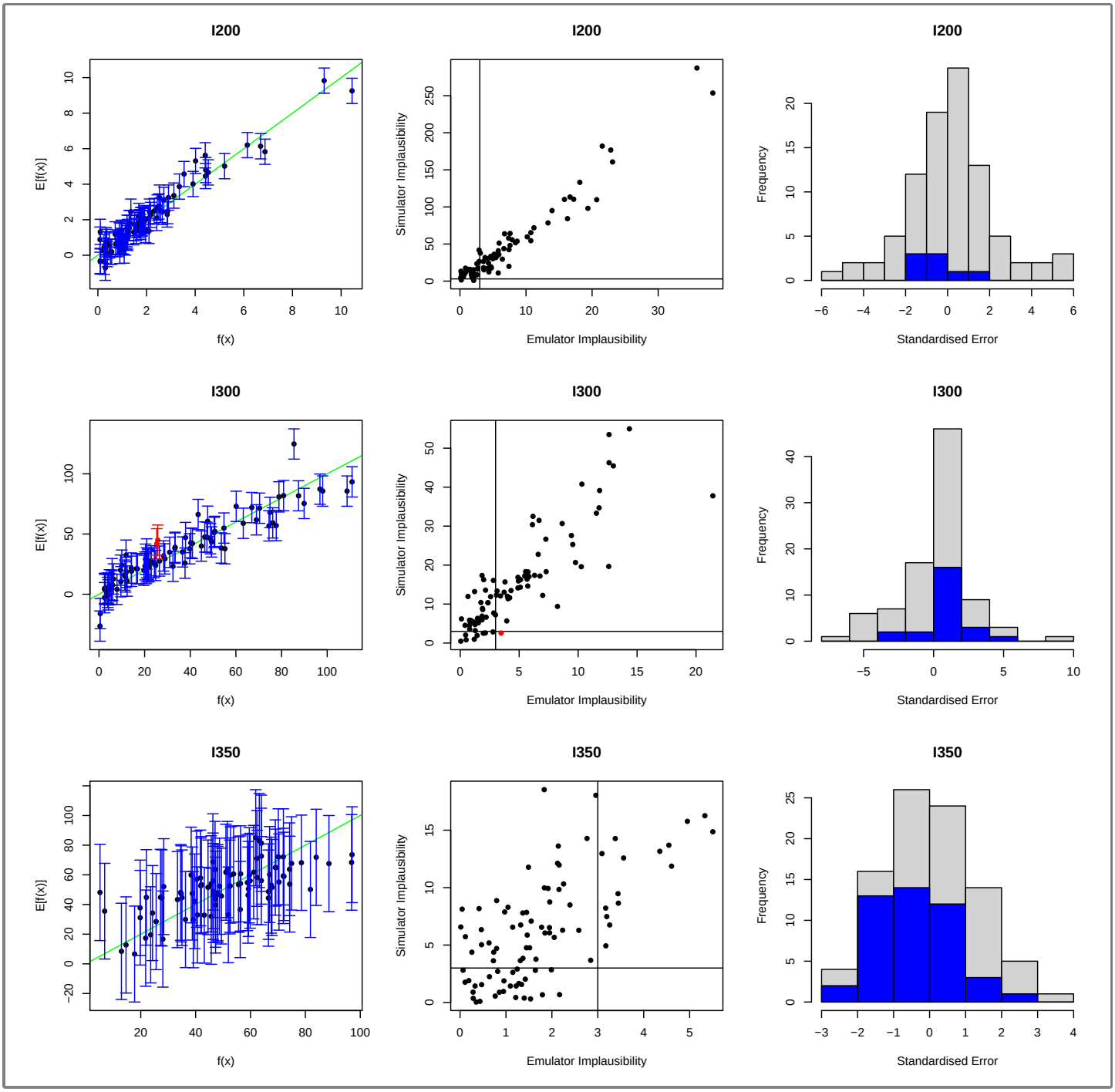
Return to task on P42

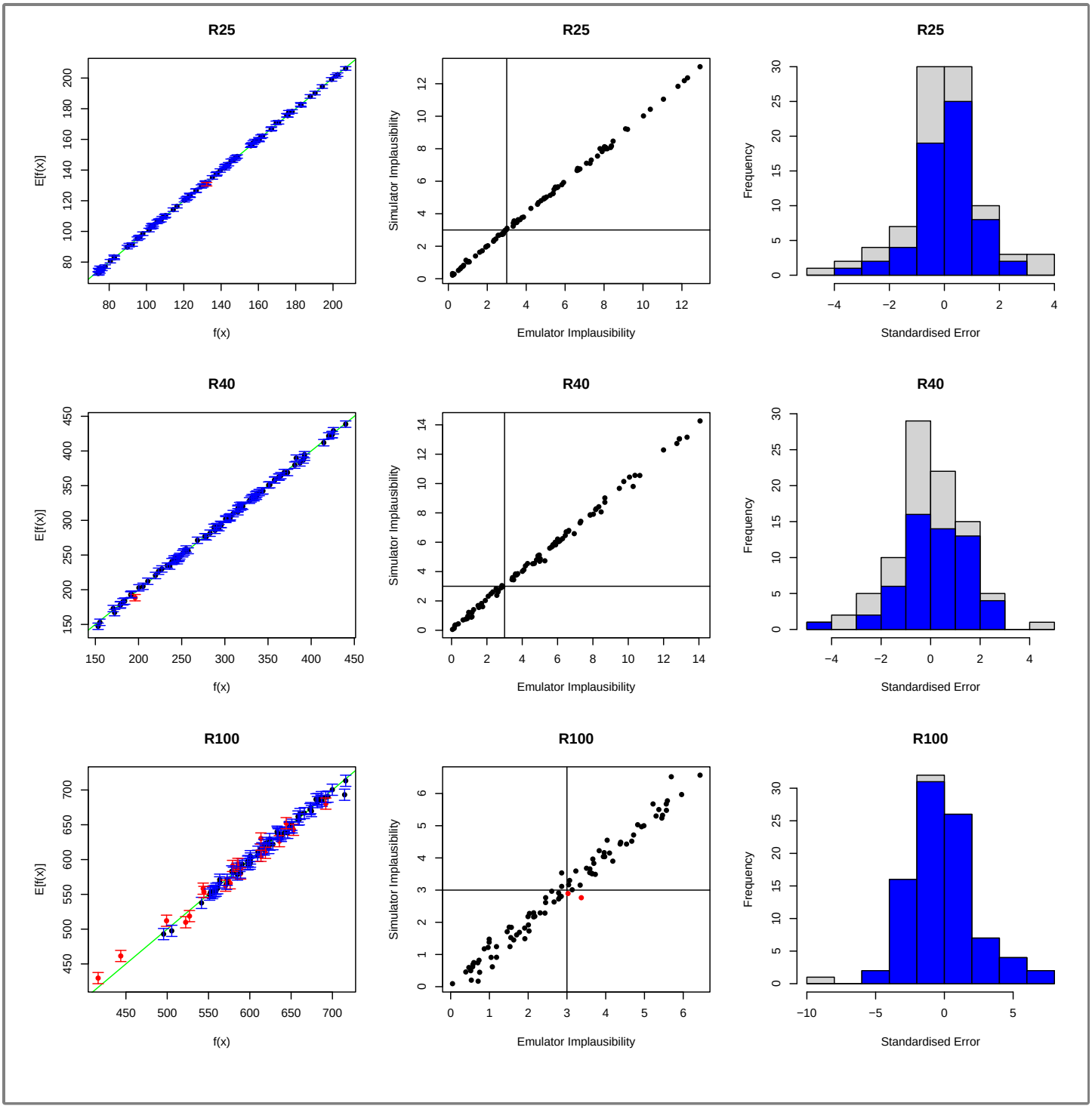
Solution 8

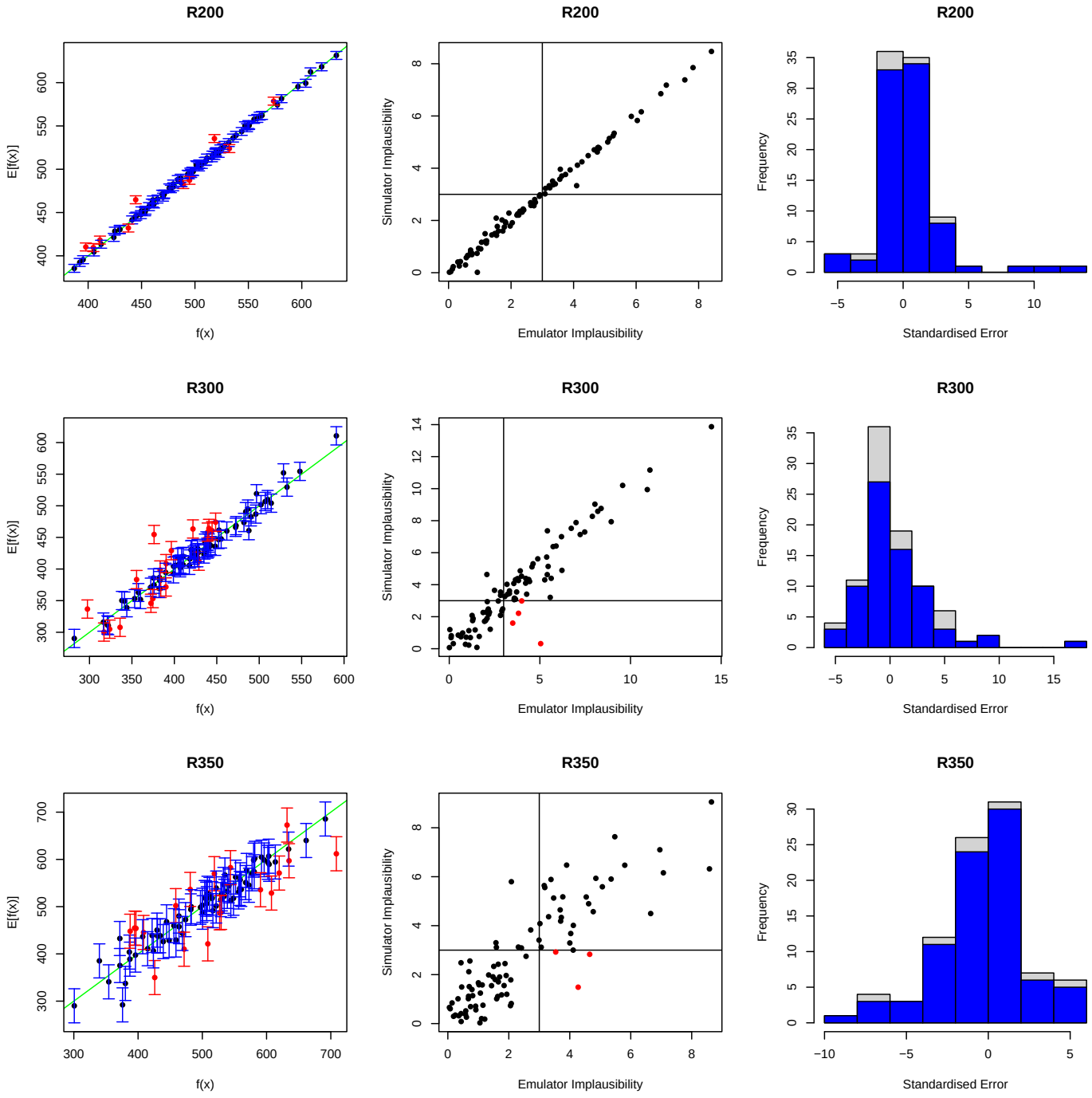
First of all let us take a look at the diagnostics of all the emulators trained in wave 1:

```
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

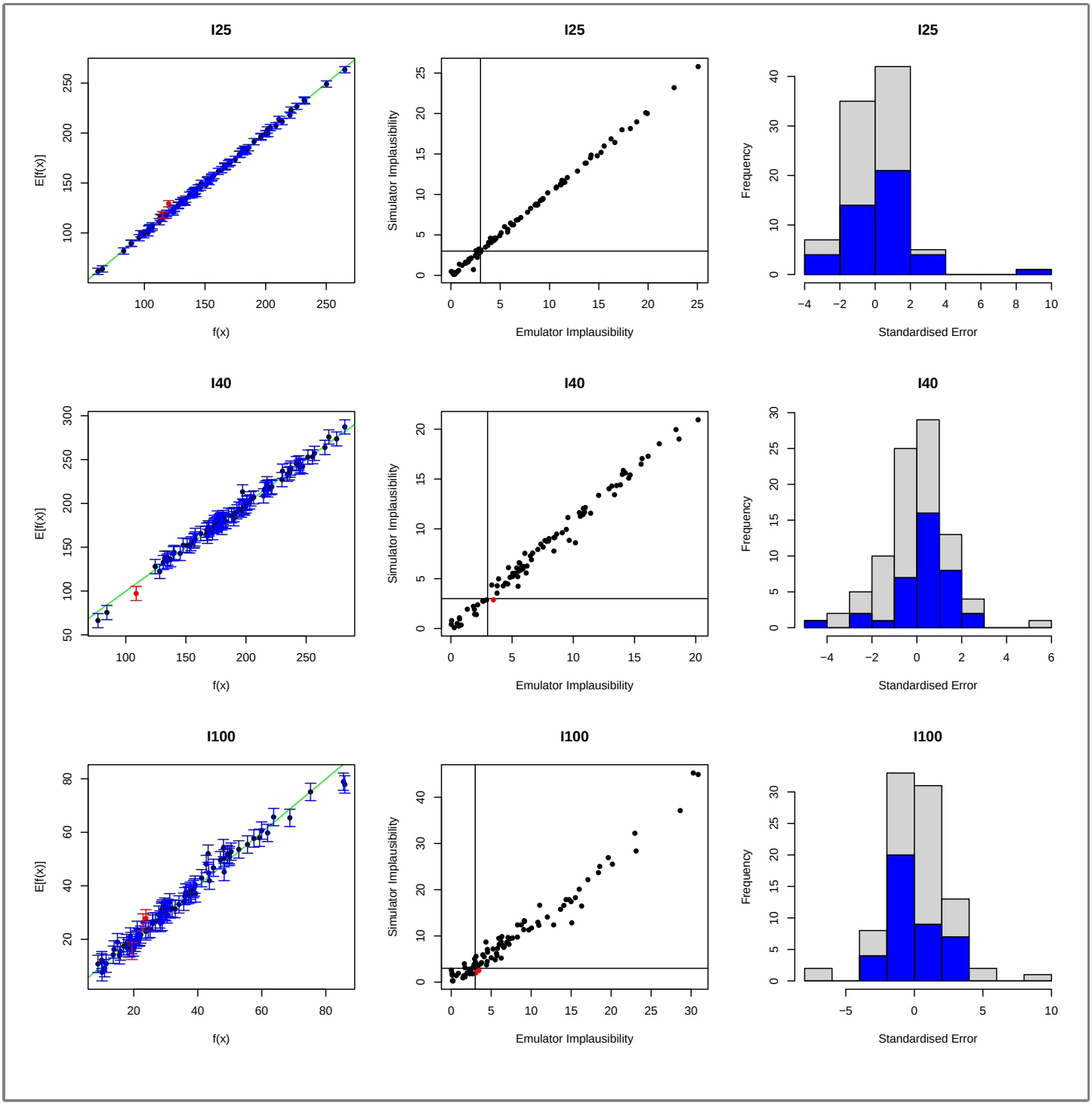

R25
E[f(x)]
f(x)
R25
Simulator Implausibility
Emulator Implausibility
R25
Frequency
Standardised Error
R40
E[f(x)]
f(x)
R40
Simulator Implausibility
Emulator Implausibility
R40
Frequency
Standardised Error
R100
E[f(x)]
f(x)
R100
Simulator Implausibility
Emulator Implausibility
R100
Frequency
Standardised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

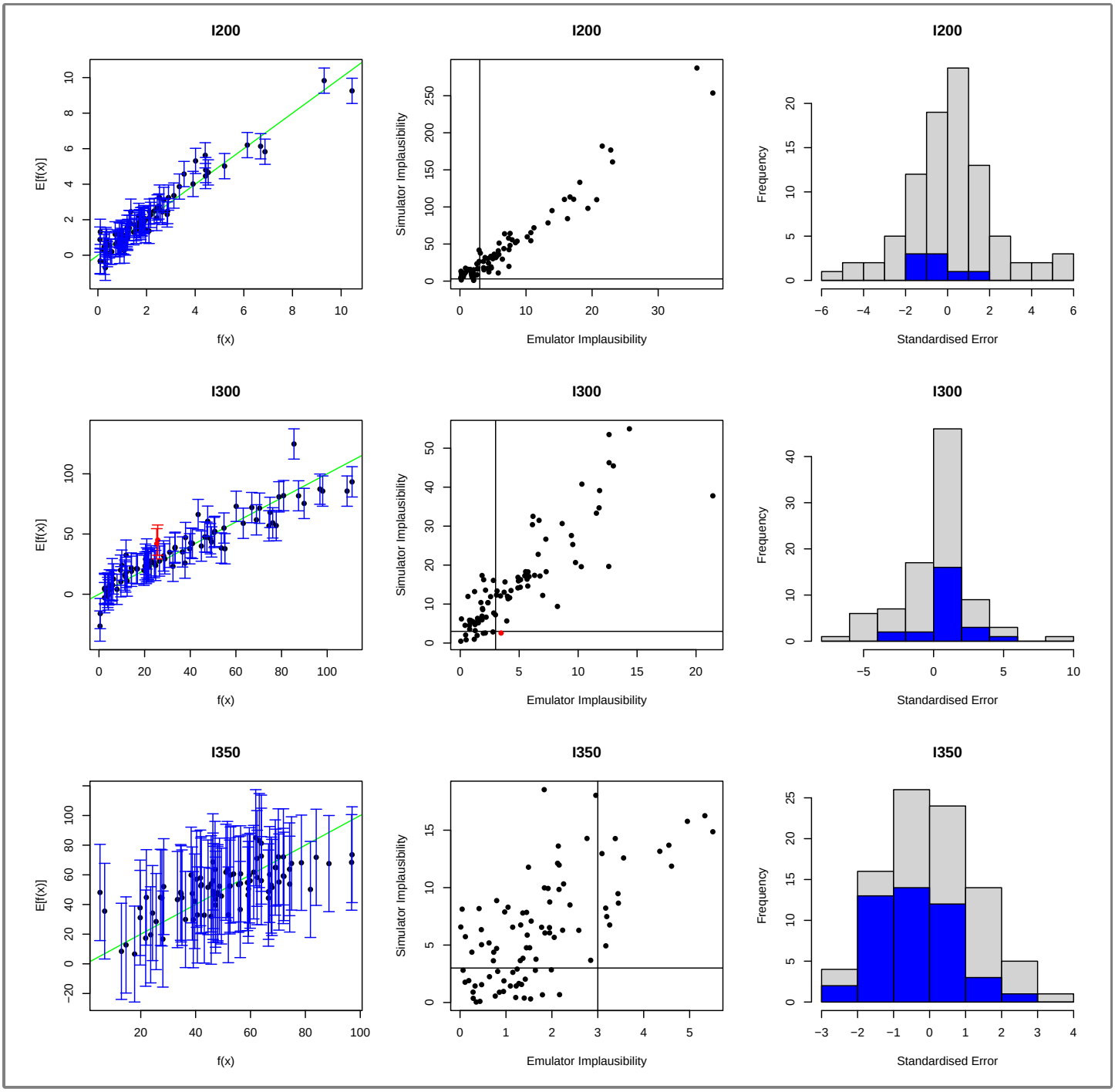


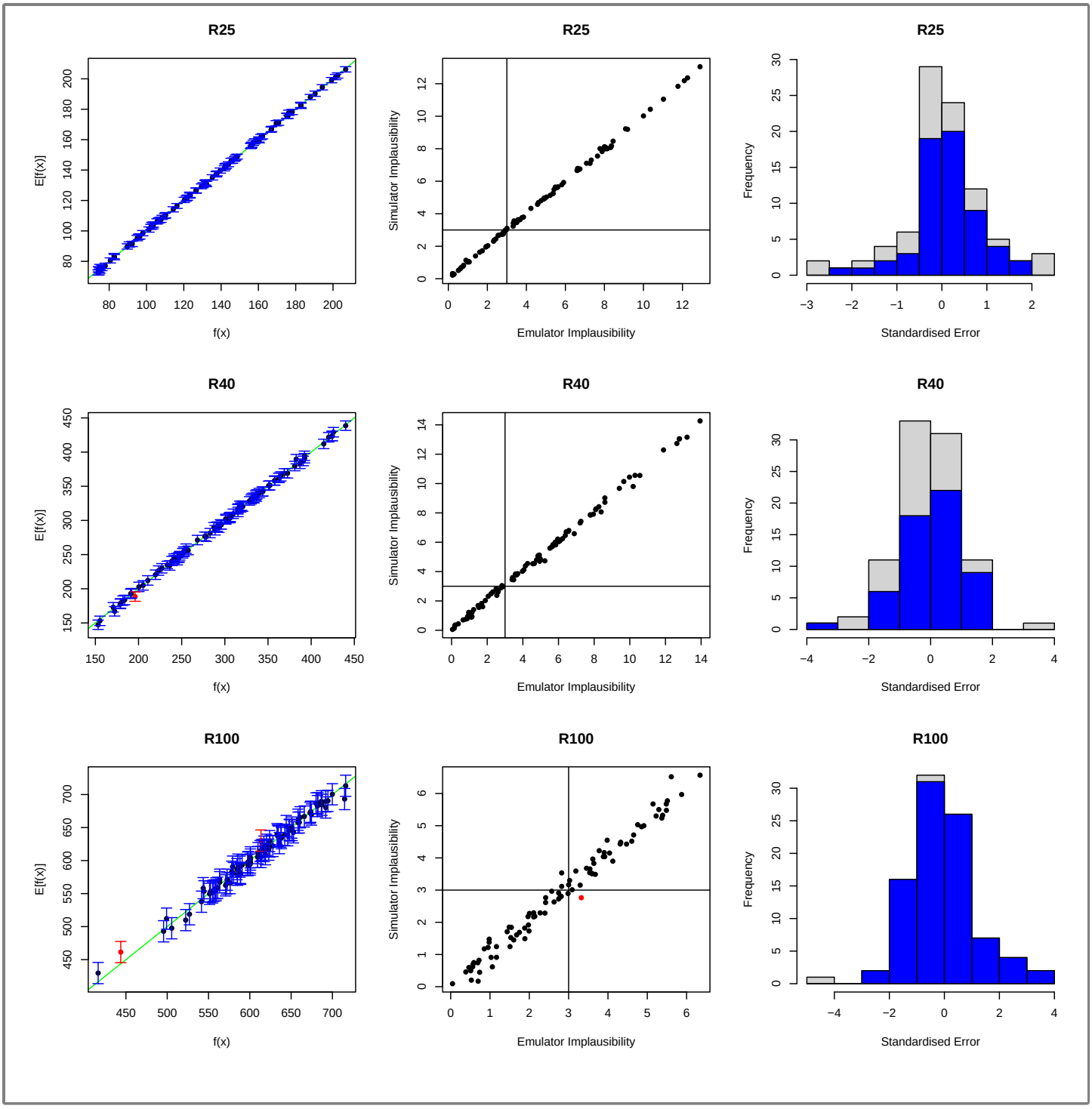
Most emulators would benefit from having slightly more conservative emulators, which we can obtain by increasing σ. After some trial and error, we chose the following values of sigma for our emulators:

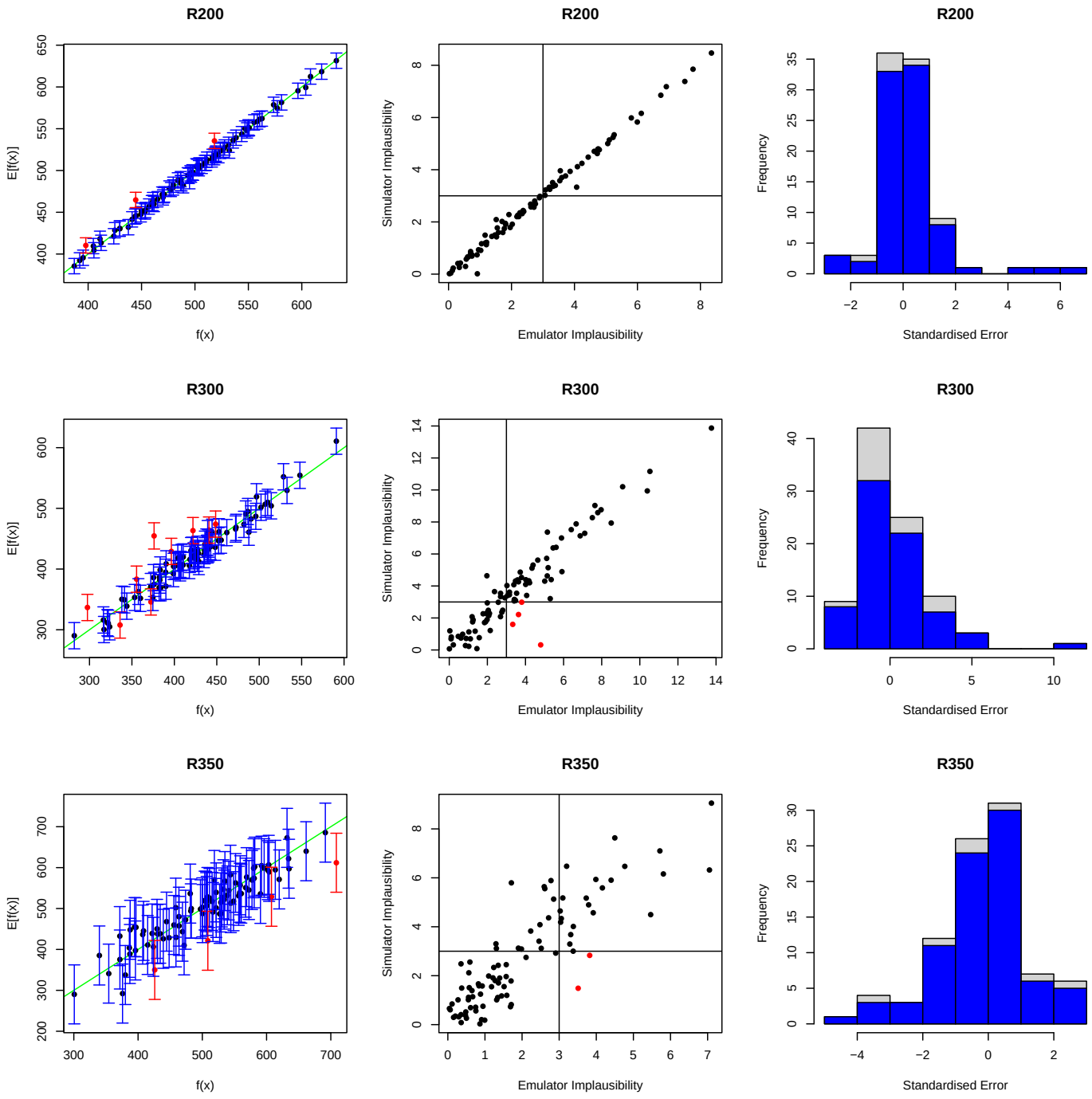
```
inflations <- c(1.5,1.5,1,1,1,1,1.5,1.5,2,2,1.5,2)
for (i in 1:length(ems_wave1)) {
  ems_wave1[[i]] <- ems_wave1[[i]]$mult_sigma(inflations[[i]])
}
```

```
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

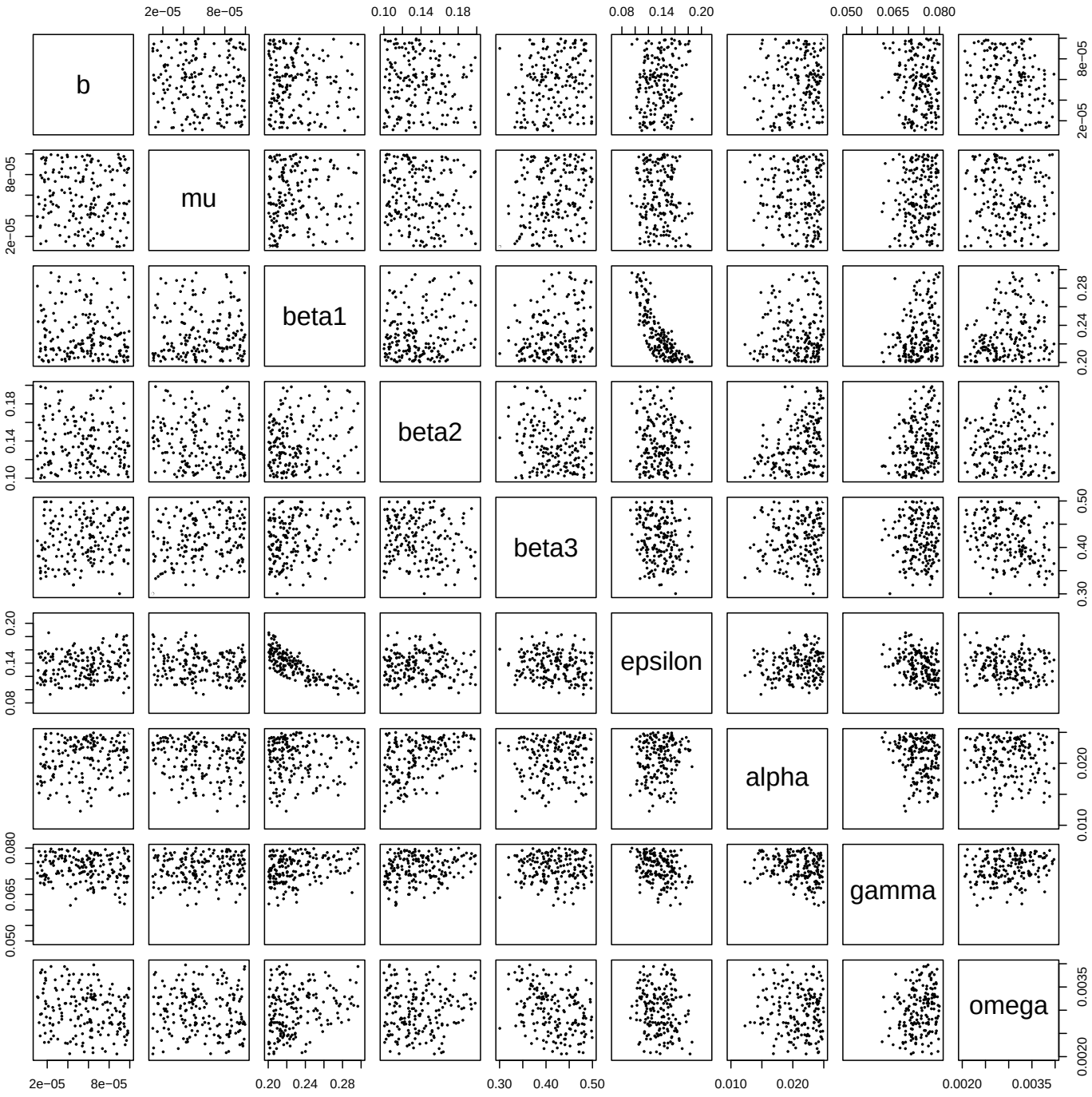


The diagnostics look good now. Note that a high increase in sigma, which is sometimes needed to pass diagnostics, may end up producing an emulator that cuts (almost) no space out. This can be checked by looking at the middle diagnostic: if (almost) no validation point has implausiblity above three, then it is advisable not to include the emulator when generating new points. This will avoid adding unnecessary calculations and will not affect the amount of space cut out at that wave. In our specific case, all emulators are contributing to shrinking the space down, so we will include all of them when generating new points:

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

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

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



If we compare the parameter sets we just generated with those generated using non-customised emulators, we note that the space has now been reduced less than before. This happened because our customisation helped us to build more conservative emulators, decreasing the risk of rejecting good parts of the input space. Building emulators carefully ensures that we end up with a set of points that are truly representative of the set of all points that fit the observations (rather than a subset of it).

Return to task on P45

Solution 9

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,]
```

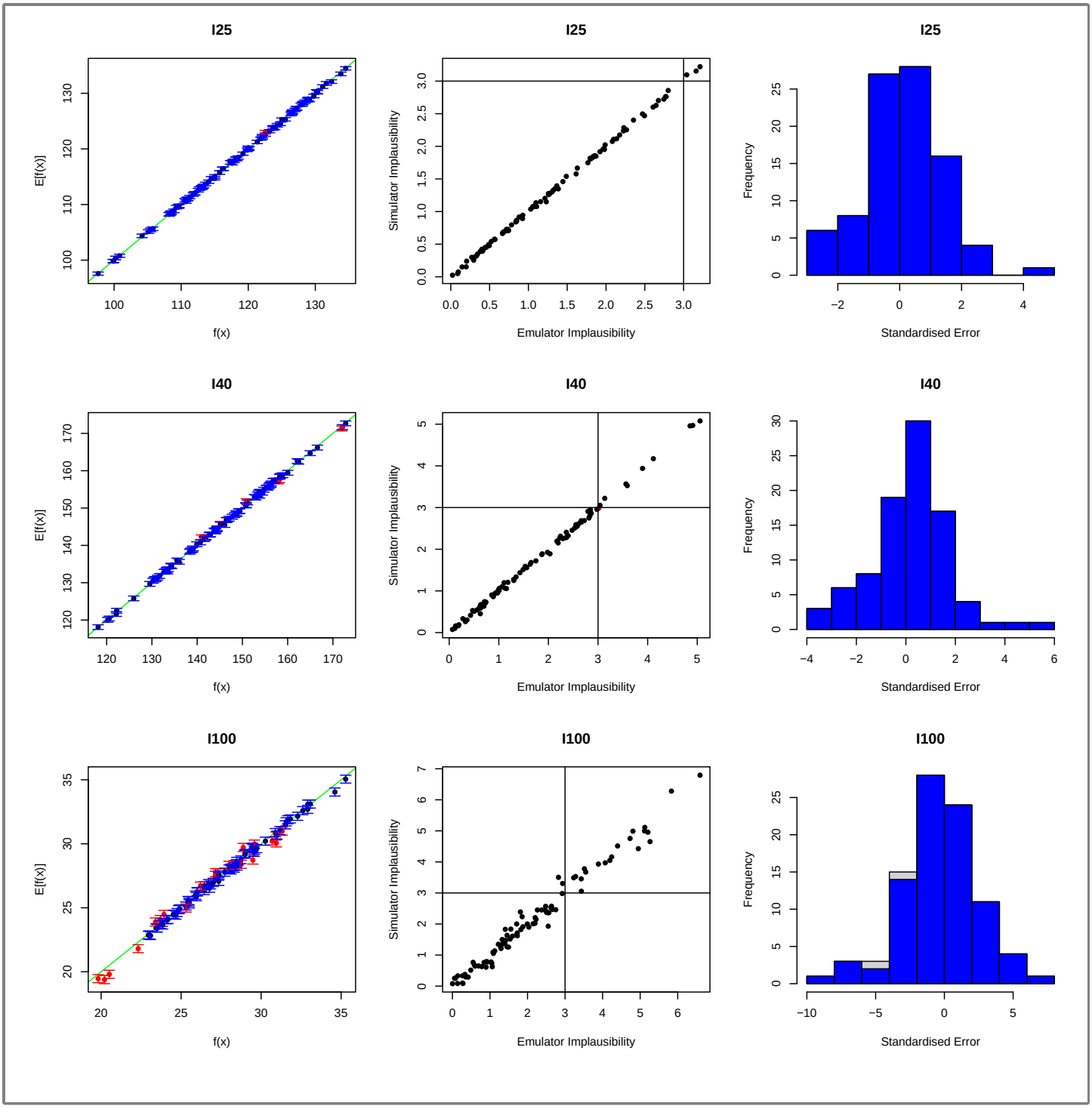
and train wave two emulators on the space defined by `new_ranges`:

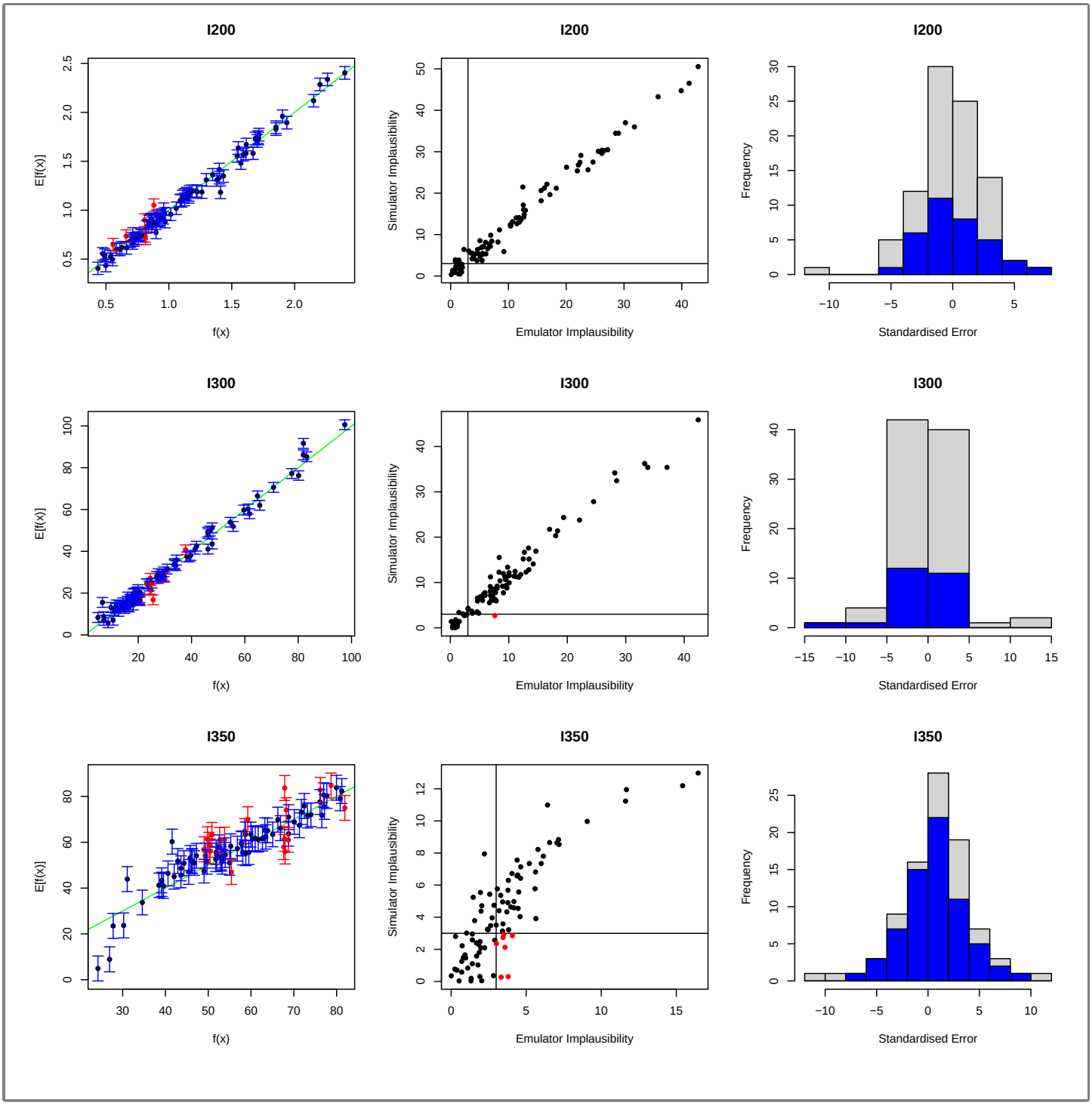
```
ems_wave2 <- emulator_from_data(new_training, names(targets), ranges,
                                check.ranges=TRUE,
                                specified_priors = list(hyper_p = rep(0.55, length(targets))))
```

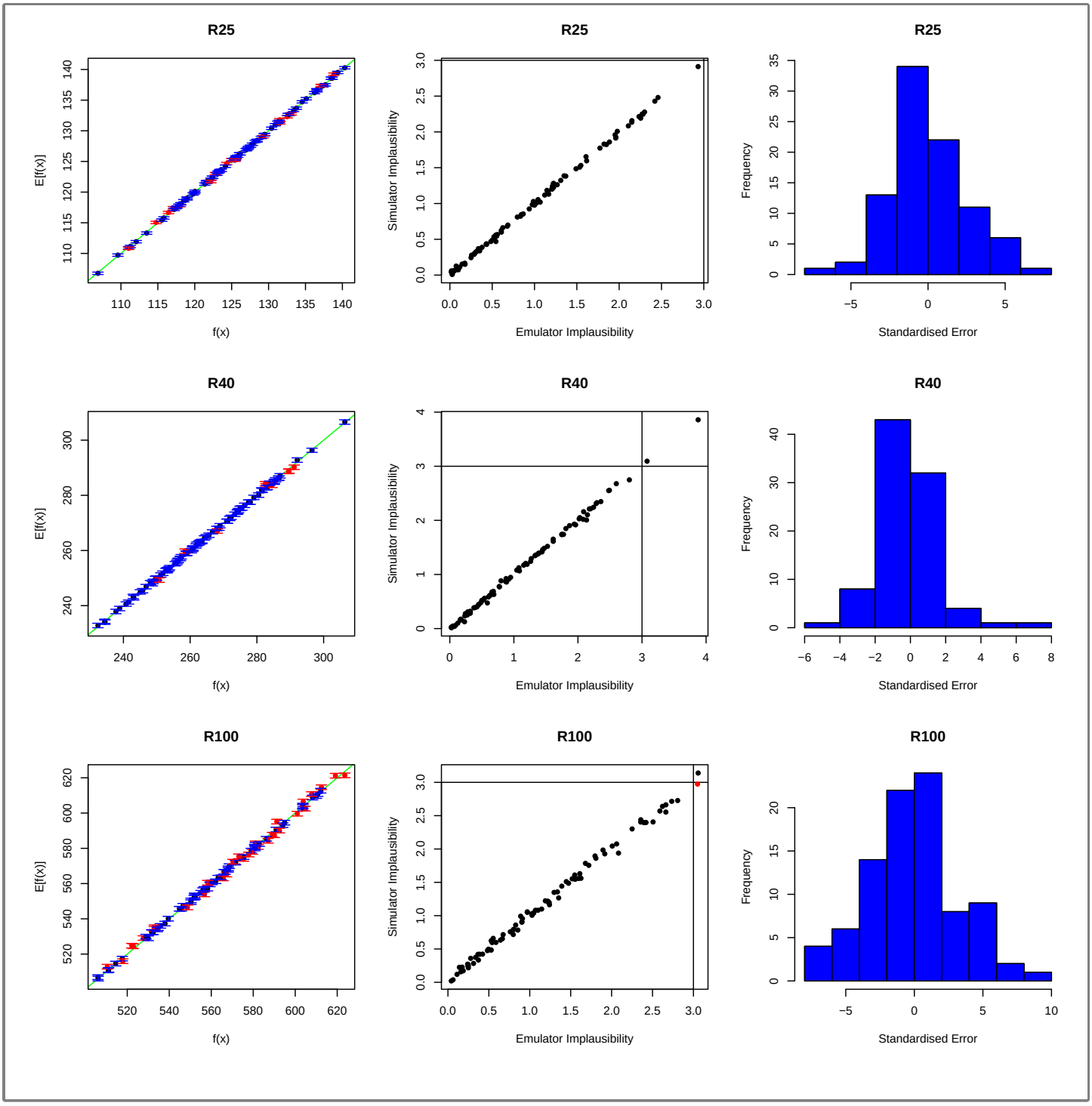
```
## I25
## I40
## I100
## I200
## I300
## I350
## R25
## R40
## R100
## R200
## R300
## R350
## I25
## I40
## I100
## I200
## I300
## I350
## R25
## R40
## R100
## R200
## R300
## R350
```

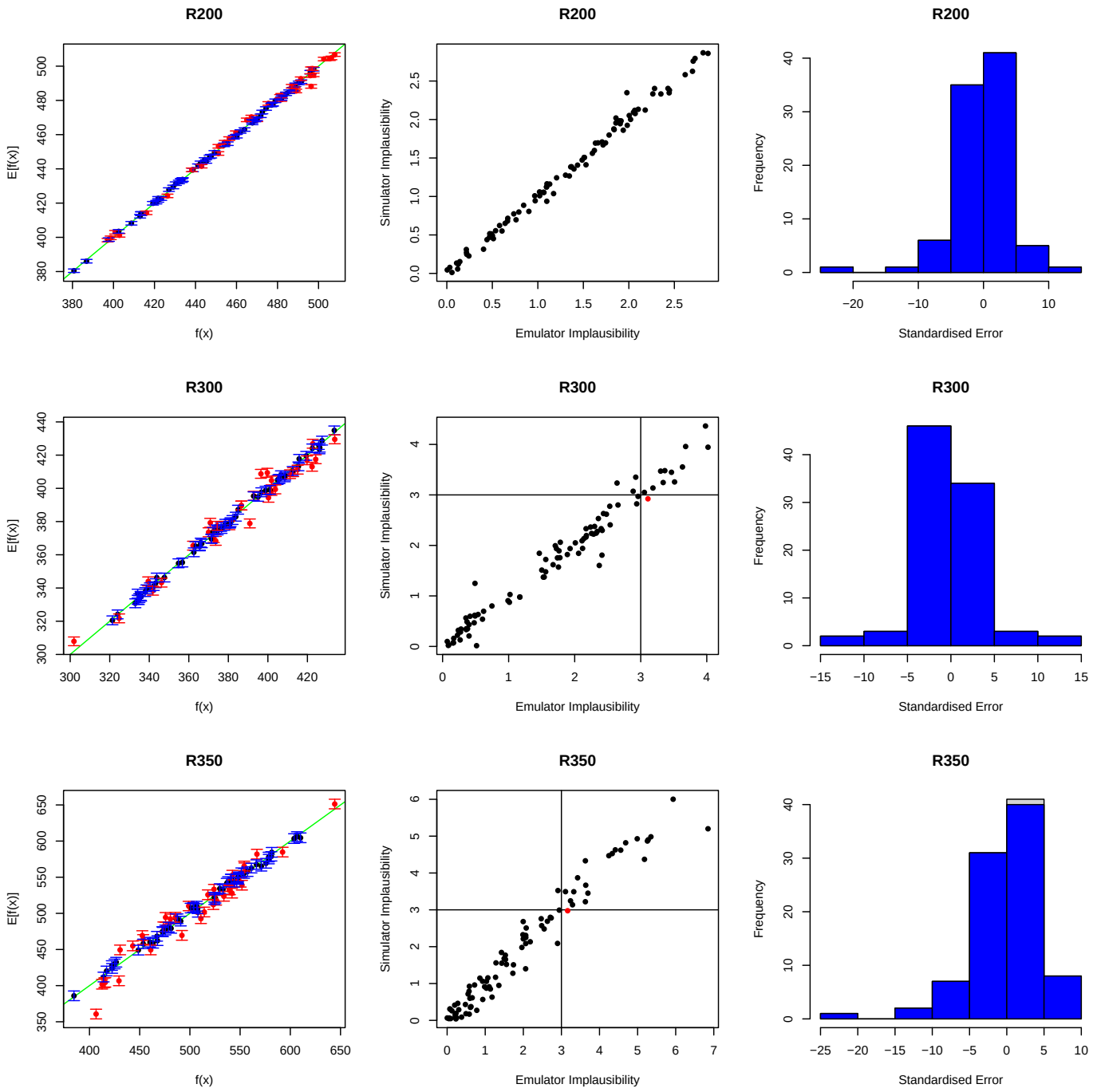
Let us check their diagnostics:

```
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



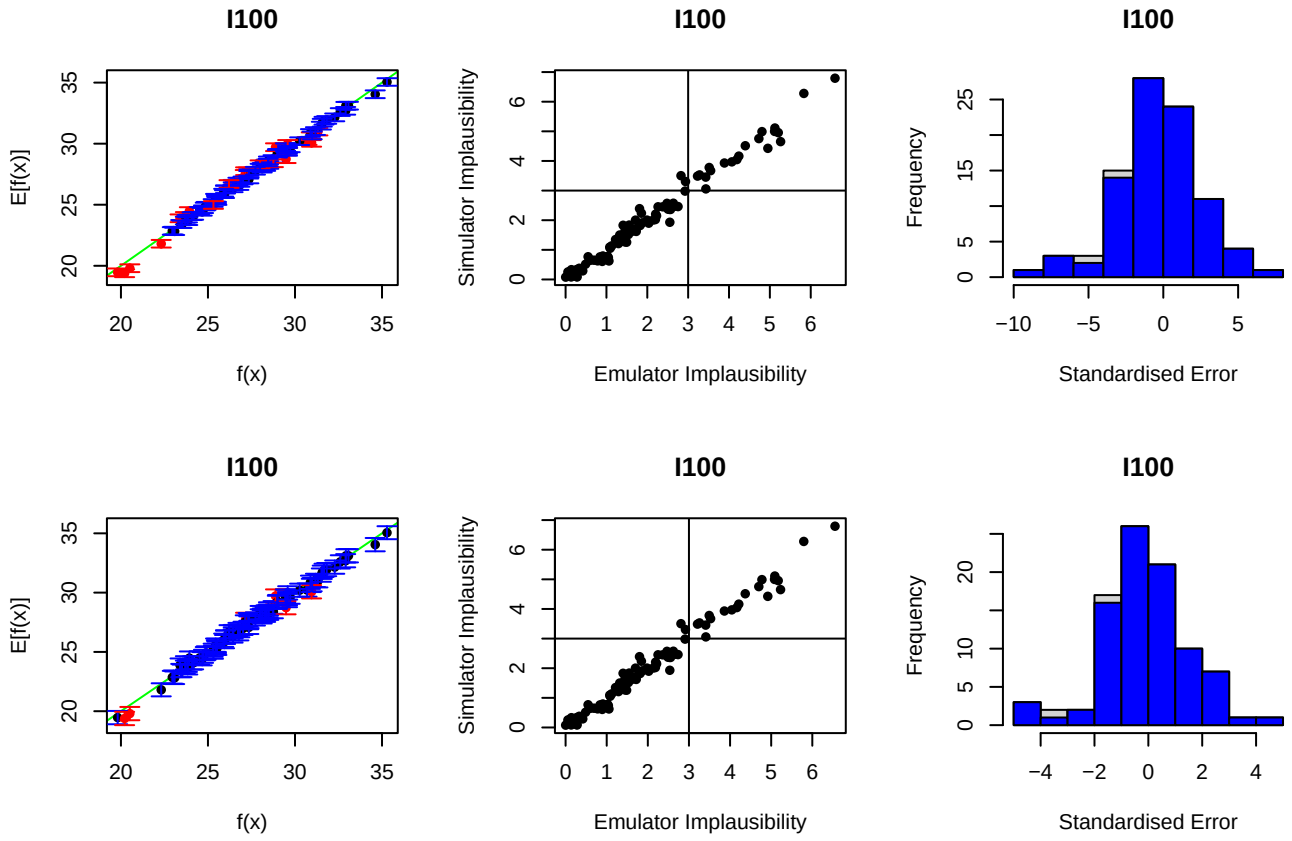
We will use the automated diagnostic pass to modify these emulators:

```
ems_wave2_new <- diagnostic_pass(ems_wave2, targets = targets, validation = new_validation, verbose = TRUE)
```

```
## Checking for structured errors in input space..
## Checking for structured errors in output space..
## Checking for problematic implausibility misclassifications..
## Classification diagnostic failures. Inflating relevant emulator uncertainties.
## Checking for issues in comparison diagnostics..
## Comparison diagnostic issues. Inflating relevant emulator uncertainties..
## Some emulators show scaling issues: I200; I300; I350; R25; R100; R200; R350
## Some emulators did not pass diagnostics:  I200; I300; I350; R25; R100; R200; R350 .
## Investigate these outputs carefully and consider adding in additional training points near problematic r
```

Note that the automated diagnostics have performed some variance inflation, as for instance we can see with the I100 output:

```
vdiag <- validation_diagnostics(list(I100 = ems_wave2$I100, I100 = ems_wave2_new$I100), targets = targets
```



It has also seen fit to remove quite a few of the emulators, since they cannot be easily modified automatically. We can look at these emulators to see if the automated process has been overly conservative, or if we even need to worry about emulating the output. *Note that it is fine to simply apply `diagnostic_pass` and use the emulators it produces: they may be more conservative than the emulators we would obtain by hand and thus rule out less of the parameter region, but they remain valid emulators for using in `generate_new_design`.

- `I200`: The validation plots look ok; there are some comparison diagnostic issues but nothing significant. We will include this as-is.

```
ems_wave2_new$I200 <- ems_wave2$I200
```

- `I300`: We might be concerned about the classification diagnostic issue, since the emulator is predicting an implausibility around 7 when the simulator implausibility is ~ 3. However, this is only an issue if that point (and its neighbourhood) would not be ruled out anyway (remember that these are univariate implausibilities). We can identify the point and look at the implausibility across all the 'good' emulators, using `classification_diag` directly to just perform the diagnostic we want.

```
which_pts <- row.names(classification_diag(ems_wave2$I300, targets = targets, validation = new_valida
nth_implausible(ems_wave2_new, new_validation[which_pts,], targets, n = 1)
```

```
##      5822
## 5.227289
```

The point is unsuitable for matching to the data anyway, since the implausibility across the other emulators is above 5, so we are unconcerned about this emulator and can include it as we did for `I200`.

```
ems_wave2_new$I300 <- ems_wave2$I300
```

- `I350`: The comparison diagnostics look messy, and there is a 'curve' to the plot that suggests that the emulator has failed to capture the core response. We could include some of the validation points in the training set and retrain; however, here we will accept that we must leave this output out of this wave.
- `R25`, `R100`, `R200`: The classification diagnostics show that, even were we to spend some time improving these

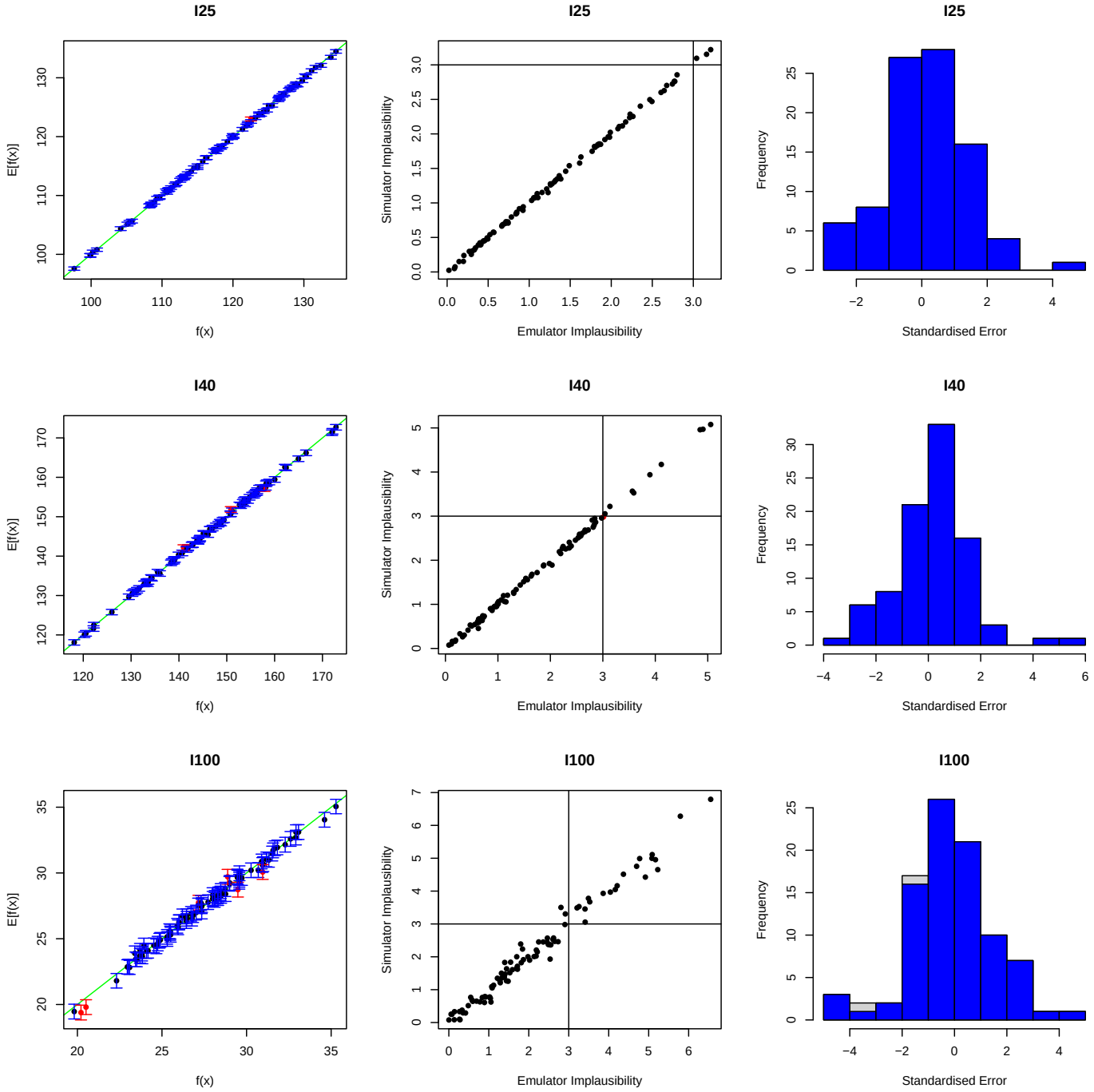
emulators, they are very unlikely to rule out any space at all. In particular, there are few or no points above simulator implausibility 3, so a more restrictive emulator is not going to be able to rule out more space without risking being overly restrictive. We can therefore leave these emulators out at this wave.

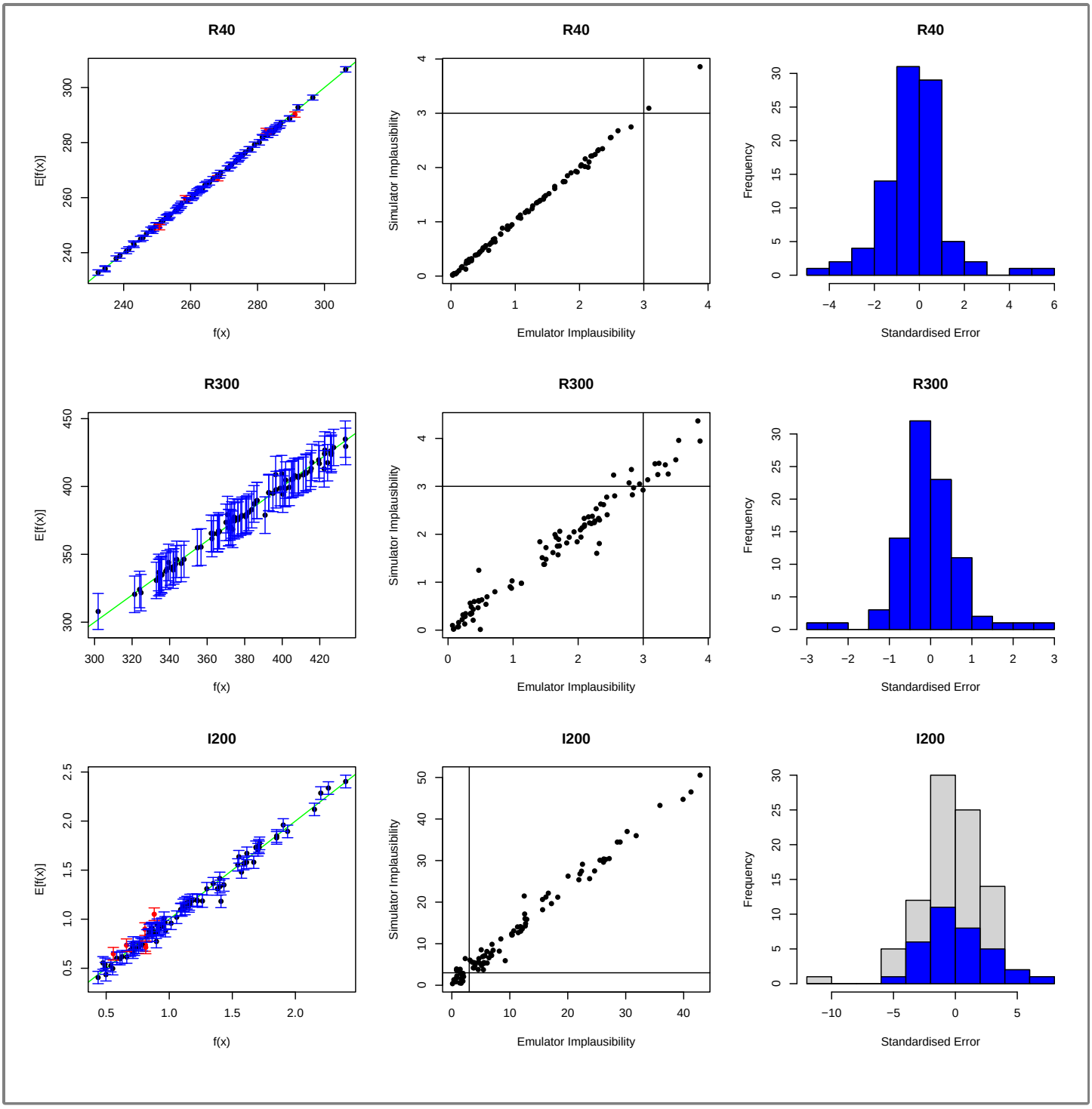
- `R350`: It looks like this would be improved by inflating the variance. We can do just that quite simply, with a bit of trial-and-error.

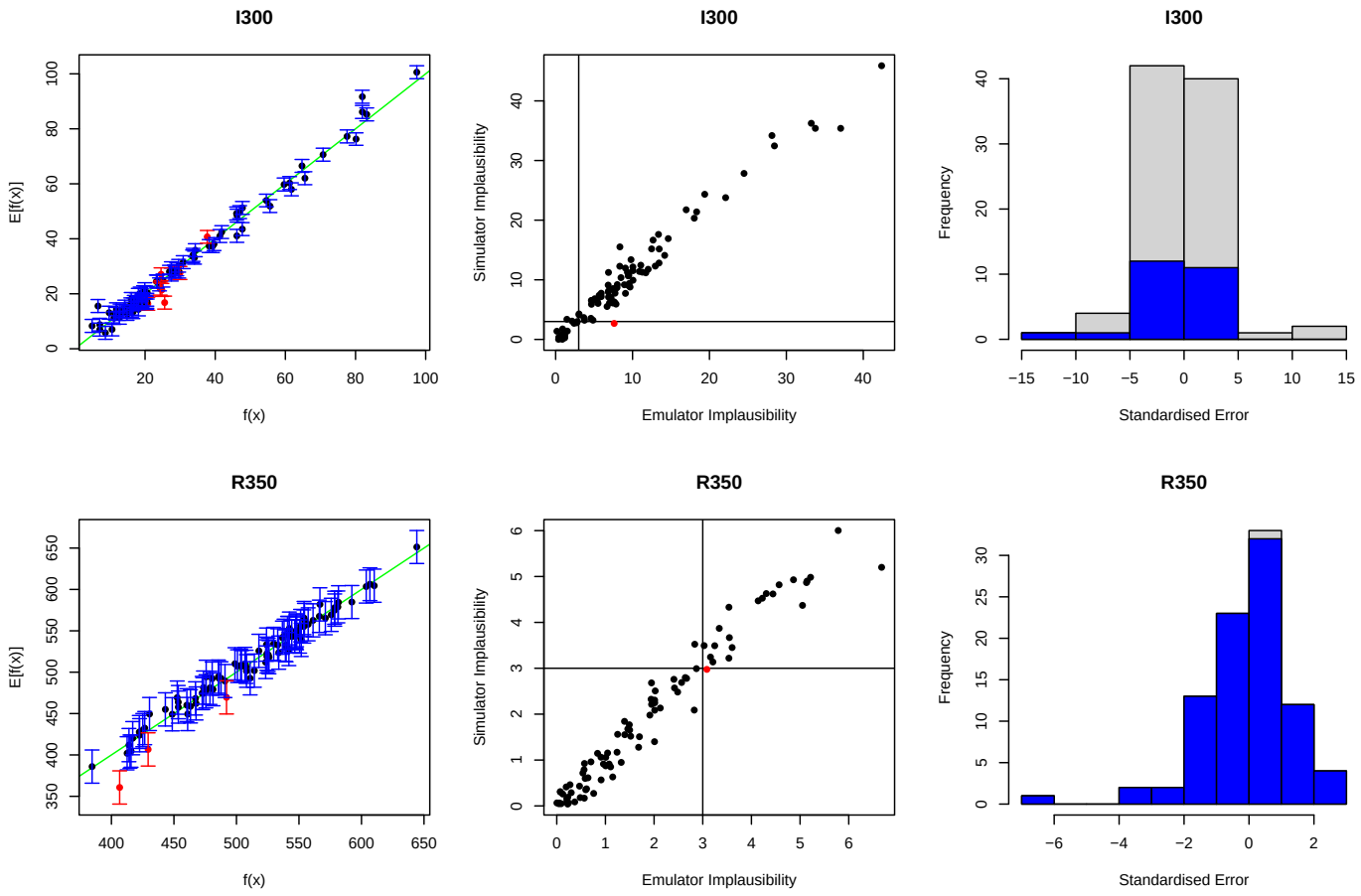
```
ems_wave2_new$R350 <- ems_wave2$R350$mult_sigma(3)
```

Having done this, we can check the diagnostic plots again to ensure the results are acceptable.

```
vd <- validation_diagnostics(ems_wave2_new, validation = new_validation, targets = targets, plt = TRUE)
```




R40
E[f(x)]
f(x)
R40
Simulator Implausibility
Emulator Implausibility
R40
Frequency
Standardised Error
R300
E[f(x)]
f(x)
R300
Simulator Implausibility
Emulator Implausibility
R300
Frequency
Standardised Error
I200
E[f(x)]
f(x)
I200
Simulator Implausibility
Emulator Implausibility
I200
Frequency
Standardised Error

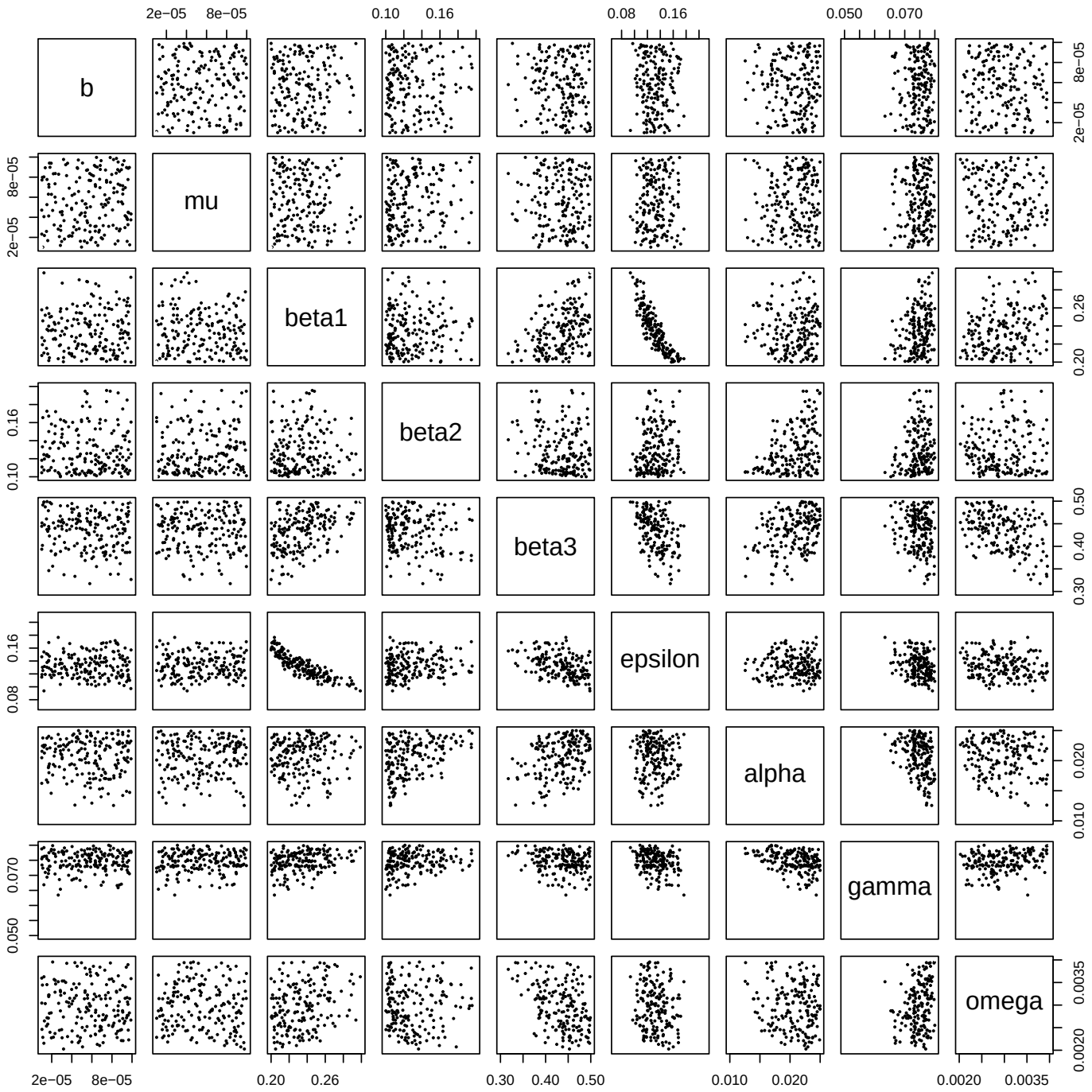


Finally, we propose points to generate the design for the next wave.

```
new_new_points <- generate_new_design(c(ems_wave2_new, ems_wave1), 180, targets, verbose=TRUE)
```

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

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

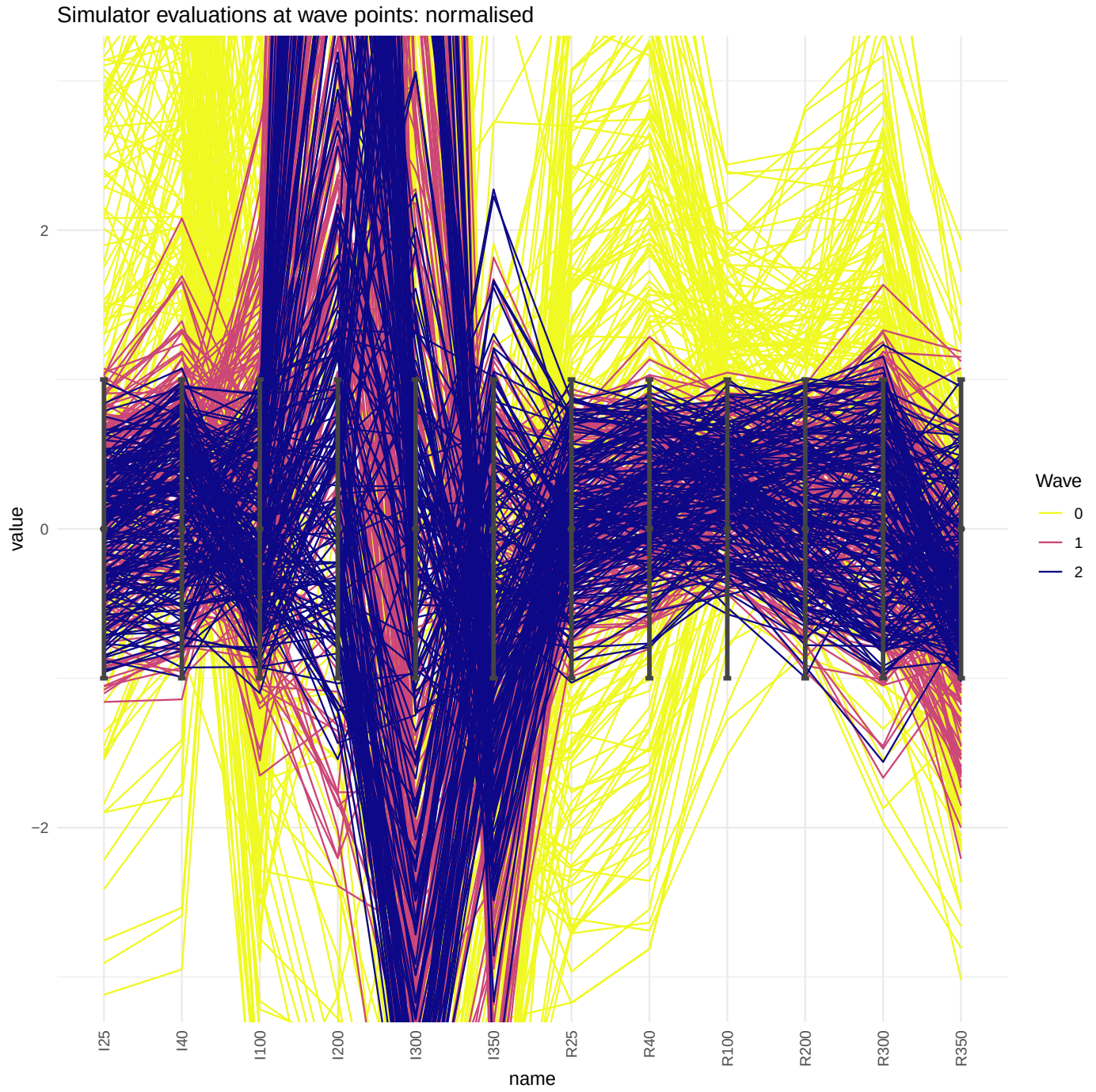


This worked well: the new non-implausible region is clearly smaller than the one we had at the end of wave one. In the next section we will show how to make visualisations to direcly compare the non-implausible space at the end of different waves of the process.

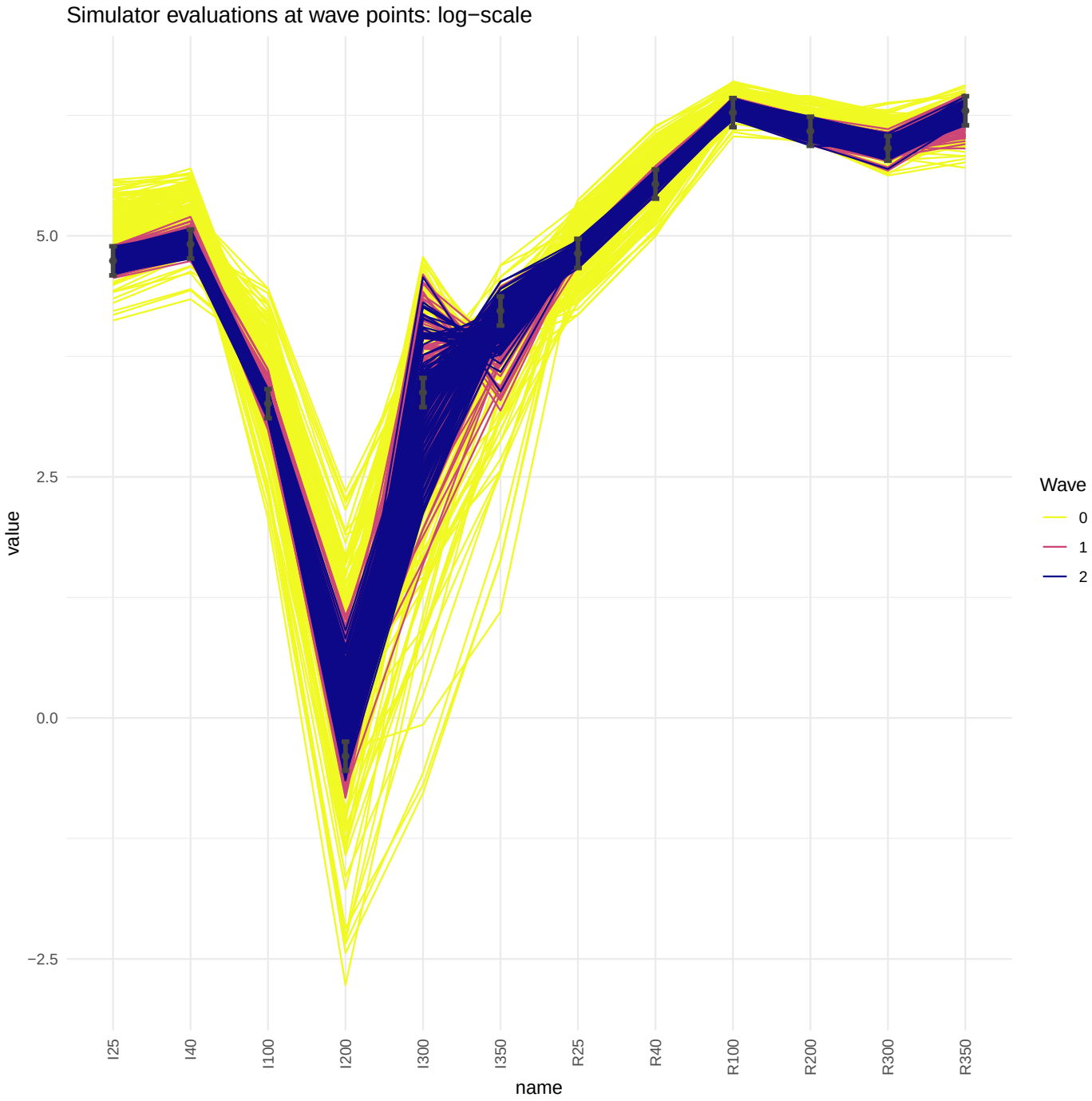
Return to task on P47

Solution 10

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




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



These normalised/logscaled plots allow us to investigate better targets such as I200: it is now clear that this is not matched yet, even at the end of wave two.

Return to task on P52

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 = "")))
}
```

Return to P5

R tip

Return to P12

More on how targets were set

Since our model is synthetic, we have no observations with which 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 = 1/36
```

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'.

Return to P14

How was the value 0.55 chosen?

The value 0.55 was chosen using the Durham heuristics, which states that the correlation length should lie in the interval $[1/(n+1), 2/(n+1)]$ where n is the degree of the fitted surface. In our case n is 2 and therefore the correlation length should be in $[1/3, 2/3]$. We chose 0.55, a value little above the midpoint of this interval, to represent our belief that the model is smooth (and therefore correlations between neighboring points should be appreciable).

Return to P21

R tip

To show what variables are active for an emulator 'em' you can access the parameter `active_vars` of the emulator, typing `em$active_vars`.

Return to P23

R tip

The `emulator_from_data` function allows us to specify the order of polynomial fitted using the `order` argument. The default is `emulator_from_data(..., order = 2)`.

Return to P26

R tip

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

Return to P33

Andrianakis, Ioannis, Ian R Vernon, Nicky McCreesh, Trevelyan J McKinley, Jeremy E Oakley, Rebecca N Nsubuga, Michael Goldstein, and Richard G White. 2015. "Bayesian History Matching of Complex Infectious Disease Models Using Emulation: A Tutorial and a Case Study on HIV in Uganda." *PLoS Computational Biology* 11 (1): e1003968.

Bower, Richard G, Michael Goldstein, and Ian Vernon. 2010. "Galaxy Formation: A Bayesian Uncertainty Analysis." *Bayesian Analysis* 5 (4): 619–69.

Goldstein, Michael, and David Wooff. 2007. *Bayes Linear Statistics: Theory and Methods*. John Wiley & Sons.

Vernon, Ian, Junli Liu, Michael Goldstein, James Rowe, Jen Topping, and Keith Lindsey. 2018. "Bayesian Uncertainty Analysis for Complex Systems Biology Models: Emulation, Global Parameter Searches and Evaluation of Gene Functions." *BMC Systems Biology* 12 (1): 1–29.