Comparing multiple simulators using Bayesian emulators

OXLADE, RACHEL,HEATHER

Comparing multiple simulators using Bayesian emulators

Rachel Heather Oxlade

A Thesis presented for the degree of
Doctor of Philosophy

Department of Mathematical Sciences
University of Durham
England

May 2012

Comparing multiple simulators using Bayesian emulators

Rachel Heather Oxlade

Submitted for the degree of Doctor of Philosophy
May 2012

Abstract

Bayesian emulation has proved to be a useful tool for working with complicated, high dimensional simulators, approximating the simulator's behaviour in a probabilistic way, enabling operations such as prediction or calibration, and therefore providing an efficient approximation to the simulator's representation of the system.

Complex systems, however, are often modelled by several different simulators, each with different strengths and weaknesses. Combining them to better understand the system, or comparing their behaviour as functions, is very difficult. This is largely because their input spaces cannot be directly linked.

In this thesis, we present two methods for using emulation to jointly model two simulators, allowing them to be compared. We also introduce two simulators of the ocean carbon cycle, OG99NPZD and HadOCC. The ocean carbon cycle is of interest largely because it concerns the biological processes by which some carbon is stored in the deep ocean. These simulators have different input spaces and model the system differently, and standard emulation proves to be unable to compare them.

The first method for two simulators, *hierarchical emulation*, works with pairs of simulators for which one is an extension of the other, and therefore whose input spaces are mostly similar. This uses the relationship between the simulators to emulate the more complex as a sum of the simpler simulator and some newly created functions. Validation studies using hierarchical emulators to model two versions of

HadOCC show that the hierarchical emulator outperforms the standard methods in modelling both the extended simulator and the difference between the two.

The second, *intermediate variable emulation*, makes no constraint on the relationship between the simulators, instead making connections using sub-processes represented in both. This allows the representations of a system by two simulators to be directly compared; the contributions of the different sub-processes can be contrasted, and the sub-processes themselves can be used to gain better understanding of the relationship between the two input spaces. Intermediate variable emulators are used to compare OG99NPZD and HadOCC.

Finally, to enable an efficient and robust implementation of these methods, as well as of the standard emulation method, an object-oriented framework for emulation is presented.

Declaration

The work in this thesis is based on research carried out at the Probability and Statistics Group, the Department of Mathematical Sciences, the University of Durham, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

The last 42 months have flown by, and it is only thinking of all the people who've helped and supported me that fills it out in my mind.

Being supervised by Michael and Peter has been a tremendous privilege. They have helped me grow enormously in thinking and pursuing ideas, and have been constantly encouraging, patient and positive. I'm particularly glad I was tricked into learning how to program. I'm also very thankful to John Hemmings for his very prompt and helpful emails, and for his extraordinary depth of knowledge of the simulators.

There have been very few dull or lonely moments in the department, and almost always fun and entertaining people around; Amani, Ben, Ben, Caroline, Danny, Fliss, Joey, John, Jonathan, Ian, Nathan and Ric to name a few.

My parents have encouraged me throughout my PhD, and have been supportive of whatever I've chosen to do, and I'm very grateful to them for never putting me under pressure to do something 'proper'. I've been very glad of my brother David, for lots of coffee times on despondent afternoons.

Gemma and Sarah have been wonderful friends and housemates, and a great source of hilarity, comfort and advice.

Finally, Rick has loved, amused and looked after me through all manner of fretting, ranting and confusion, and has constantly sought to remind me that, while a thesis is an interesting and enjoyable endeavour, it is only the word of God that will endure forever, and for that I am most thankful.

Contents

Chapter 1

Introduction

1.1 Computer Simulators

Whether we consider it or not, computer simulators now play a significant part in everyday life. One area where this is especially the case is weather prediction - the forecasts we use daily or hear of in the press are the results of hours of intense computation involving sophisticated models. The output, whether it covers the next few hours or the next several hundred years, will often be used to make crucial decisions or communicated to the public for our use. Ultimately, the simulator should be able to be used to gain insight into the real system, for prediction (forecasting the value of the system under particular conditions) or calibration (deducing which sets of inputs lead to output which most closely matches observations of the real system).

What is sometimes less well understood is the amount of error in these predictions, whether owing to a lack of understanding of the process, to insufficient computing power or to some other factor. While the forecasts are often interpreted as precise and true, a closer look reveals many sources of uncertainty. There are several reasons why a simulator might *not* give an accurate likeness of the real world, as listed for example by Kennedy and O'Hagan (2001).

A simulator relies on knowing the values of various aspects of the system, which are entered as input parameters. These may each have a clear physical meaning, for example the viscosity of the ocean, or the speed at which dead phytoplankton sink to the sea floor. They may be more abstract, and constructed to fit the working of the simulator, for example a ‘nutrient uptake half-saturation constant’ (Hemmings, 2000). Either way, obtaining a correct value for these inputs is usually either extremely difficult or impossible. It is likely that the quantity varies across the spatial or temporal region of interest so that there is no ubiquitously ‘true’ value. It could well be that no experiment to obtain an estimate of the value can even be conceived of, much less performed. This means that there are many parameters whose true values are unknown, and none about which we are certain.

This is made worse by the fact that no simulator will ever truly and fully describe the system it models. Processes or parameters will be absent, and the understanding behind the modelling choices may even be entirely wrong. This means that even if we were to obtain the ‘true’ parameter values and run the simulator with them, the output we obtained would not match the system.

A further source of error is that with deterministic computer simulators (the focus of this work), once values have been specified for each parameter, the output will always be the same. With the real world this is not the case - even when all the specified conditions remain the same there will be residual variation. This may be because of features of the system that have been omitted, and so could in theory be reduced by refining the simulator (in which case this fits into the previous category). On the other hand there may be elements that are stochastic and inherently unpredictable.

Finally, the dimension of the space of input parameters and the speed at which the simulator evaluates usually mean that exploring a simulator’s behaviour over its input space is impossible.

Addressing some of these issues will be the focus of the first part of this thesis. In Chapter 2, the ocean carbon cycle will be briefly explained, and two simulators which model it, OG99NPZD and HadOCC, will be introduced.

In Chapter 3, we introduce Bayesian emulation and explain how an emulator is built. This involves several decisions, concerning such issues as the design of experiments, choosing a suitable regression surface and specifying a correlation structure, and we will focus on these in some detail. To illustrate the emulation process, we will build emulators for OG99NPZD and HadOCC. After this, attention will be turned to emulation for multiple simulators. In Chapter 3, a method is introduced for the construction of large Latin hypercube designs, with particularly good properties for validation studies. This method is employed in the example in Chapter 5.

1.2 Studying multiple simulators

While emulation can help to understand the uncertainties around a particular simulator, by enabling the user to better handle the high-dimensional input space, its performance is limited by the simulator itself. If the simulator represents the system badly, so will the emulator, and it does not offer any way to expose missing or poorly modelled processes, a key source of error mentioned in the previous section.

One approach that has been proposed to address this weakness is to study multiple simulators. Any system that it would be beneficial to understand is likely to be modelled by several different simulators. Many countries have their own weather and climate models, banks independently predict the financial market's behaviour, oil companies each create models predicting yield, and so on. Depending on the level of interaction between these interested parties, the simulators may differ significantly. Some will include certain processes where others omit them. There will be different ways to parameterise elements of the system's behaviour, and varying levels of complexity.

It seems intuitively plausible that using these simulators together should lead to a better understanding of the system in question. Rather than regard one as the best, and all others as redundant, many scientists take the view that each has strengths and weaknesses, and can give insight as modellers seek to improve their representations of the system.

In Chapter 4, a brief summary is given of existing work involving multiple simulators, most of which deal with a single calibrated output from each simulator. In contrast, we seek to treat each simulator as a function of its input space, using emulators. There follows an exploration into the difficulties involved in comparing simulators when their input spaces are not the same, and a breakdown of some ways in which simulators can differ.

Chapters 5 and 6 then introduce new methods for emulating multiple simulators, each applicable to a particular sort of difference. Hierarchical emulation, the focus of Chapter 5, is able to emulate a pair of simulators for which one is a particular sort of extension of the other, by incorporating the relationship between the two into the emulator. An example is given using two versions of HadOCC, and in a validation study the hierarchical emulators outperform their standard equivalents.

Intermediate variable emulation, introduced in Chapter 6, is applicable to a much broader class of pairs of simulators. The simulators must share some similar process, but there needn't be any clear links between their input spaces. This method is used to compare the effects of each input space on the processes modelled in each simulator, and to explore the ways these processes contribute to each representation of the system. Intermediate variable emulation is then used to compare OG99NPZD and HadOCC, and to show up differences and similarities between the two that would be very difficult to infer by other methods.

Having seen the process of emulation in Chapter 3, and two extensions to this in Chapters 5 and 6, it will be clear that any emulator will involve highly structured data, and a large number of calculations. In Chapter 7 we therefore propose an object-oriented framework for emulation, using S4 objects in R (R Development Core Team, 2011). This will be extended to incorporate hierarchical emulation and intermediate variable emulation.

Overall, it is hoped that these methods demonstrate the value of comparing multiple simulators viewed as functions, and of emulation as a suitable technique for doing this.

Chapter 2

Modelling the Ocean Carbon Cycle

Much of the existing work on computer simulators focusses on climate, a crucial part of which is the transfer of carbon between the air and the sea. In order to show the methods later introduced in action, we first introduce a physical system, the ocean carbon cycle, and briefly describe two simulators of it.

It is estimated that roughly 2 gigatonnes of the world's anthropogenic carbon is held in the oceans (Palmer and Totterdell, 2001). While much of this is held by the water itself, some is captured by ocean-dwelling life forms, and eventually sinks to the ocean floor. This transfer of carbon to the deep ocean through biological tissue is known as the *biological pump.* The capture of carbon occurs through phytoplankton photosynthesising in the surface layer of the ocean, where sufficient light and nutrients are available. Rather than release the carbon in the same place, however, the phytoplankton often sinks to the deep ocean before the carbon is released, and it is this process which leads to a reduction in partial pressure between the ocean surface and the atmosphere, leading to a carbon flux (Oschlies and Garçon, 1999).

Because the ocean is such a vast and complex system, this process is affected by many other physical, chemical and biological processes. In each part of the ocean surface there will be particular currents and flows, temperatures, salinity and viscosity. There will also be various predators, particular types and concentrations of nutrients, and prevalent types of behaviour within the plankton community.

In an attempt to capture some of these details, both models introduced below couple a biological model to a physical model. In this study, the biological models will be our focus, leaving the forcing functions and models governing the physical processes alone, and so before introducing the two specific simulators used in this thesis, we will discuss the seminal compartmental ocean ecosystem model on which they are based.

2.1 Compartmental ocean ecosystem modelling

The goal of Fasham et al. (1990) (whose simulator we will refer to as FDM90) was to learn about how ocean ecosystems affect atmospheric carbon dioxide, in particular focussing on the cycles in plankton and nutrient populations, whose roles had been shown to be important by both microbiological results and satellite data. The link between carbon dioxide and the phytoplankton population had been explored through simulation before, [see Fasham et al. (1990) for references], but this approach was new, in that they chose to model the system from the bottom up, allowing the microbial processes and the resulting transfers of nitrogen to govern population dynamics and biogeochemical cycles.

Because nitrogen is widely accepted as the limiting nutrient for the production of phytoplankton, a compartmental ecosystem model was used, with each compartment representing a possible form nitrogen could take in the system. These were phytoplankton (P), zooplankton (Z), bacteria (B), nitrate nitrogen (N_n), ammonium nitrogen (N_r), labile dissolved inorganic nitrogen (N_d) and detritus (D), each measured in mMol Nitrogen m^{-3}. Their choice of compartments is explained and detailed in Fasham et al. (1990).

The compartmental model works on the premise that within the mixed layer, the speed of the flows and mixing are fast enough relative to the biological processes that it may be considered biologically homogeneous. The *mixed layer* is the topmost layer of the ocean, where there is enough light to support photosynthesis based life such as phytoplankton, and so it is here that the biological activity of interest takes place. In FDM90, the depth of the mixed layer is given by $\mathrm{h}(t)$, a forcing function.

This simply gives a depth for each time point, which is then fed into the simulator. Solar radiation is also given by a forcing function, whose values come from climate data.

The compartmental populations in FDM90 are governed by differential equations describing the flows of nitrogen between compartments, for example

$$\frac{dZ}{dt} = \underbrace{\beta_1 \, \mathrm{G}_1(P)}_{\text{Grazing on P}} + \underbrace{\beta_2 \, \mathrm{G}_2(B)}_{\text{Grazing on B}} + \underbrace{\beta_3 \, \mathrm{G}_3(D)}_{\text{Grazing on D}} - \underbrace{\mu_2 Z}_{\text{Excretion}} - \underbrace{\mu_5 Z}_{\text{Natural death}} - \underbrace{\mathrm{h}(t)\frac{Z}{M}}_{\text{Effect of MLD}} .$$

This shows how nitrogen can join the zooplankton compartment from another, or leave the zooplankton compartment to contribute to another. Here, β_1, β_2, β_3, μ_2 and μ_5 are biological input parameters to be specified by the user. The final term represents the change in concentration of zooplankton with the change of mixed layer depth. Unlike phytoplankton, which is a plant, zooplankton is deemed able to move around in order to remain where it can survive, and so when the mixed layer depth changes, the zooplankton concentration adjusts accordingly.

The simulators introduced in this chapter, and used throughout the rest of this thesis, both contain compartmental ecosystem models similar to FDM90. However, unlike FDM90, both are coupled to numerical circulation models, describing the currents and flows in the ocean.

2.2 Oschlies-Garçon (OG99NPZD)

The physical model in Oschlies and Garçon (1999), referred to as OG99NPZD, covers the North Atlantic from 15°S to 65°N, and down to a depth of around 5.5 kilometres. The vertical grid-scale is smallest nearest the surface, where the nitrogen and light based biological action takes place. The top ten layers are all around 11-15m thick, having been altered to be more precise than the original physical model. Forcing functions are used to describe wind stresses, friction stresses, salinity and sea surface temperature, and the mixed layer depth, vertical diffusivity, viscosity and advection currents are all then simulated.

OG99NPZD uses this modelled physical system behaviour to calculate the change in each biological tracer concentration C_i, in terms of nitrogen, by equations of the form

$$\frac{\partial C_i}{\partial t} = -\underbrace{\nabla \cdot (\mathbf{u}C_i)}_{\text{advection}} - \underbrace{A_\rho \nabla^4 C_i}_{\text{horizontal diffusion}} + \underbrace{\frac{\partial}{\partial z}\left(K_\rho \frac{\partial C_i}{\partial z}\right)}_{\text{vertical mixing}} + \underbrace{\text{sms}\,(C_i)}_{\text{`source minus sink'}} .$$

It is the fourth term that summarises the biological behaviour analogous to that in FDM90.

Oschlies and Garçon reduce FDM90's seven compartments to four: nutrient (N), phytoplankton (P), zooplankton (Z) and detritus (D). The dynamics of each population are governed by the source minus sink equations

$$\begin{aligned}
\text{sms}(P) &= \underbrace{\bar{J}\,(z,t,N)\,P}_{\text{growth}} - \underbrace{G\,(P)\,Z}_{\text{grazing}} - \underbrace{\left(\mu_P P + \mu_{PP} P^2\right)}_{\text{death}} \\
\text{sms}(Z) &= \underbrace{\gamma_1 G(P) Z}_{\text{grazing}} - \underbrace{\gamma_2 Z}_{\text{excretion}} - \underbrace{\mu_{ZZ} Z^2}_{\text{mortality}} \\
\text{sms}(D) &= \underbrace{(1-\gamma_1)\,G(P)Z}_{\text{unassimilated food}} + \underbrace{\mu_{PP}P^2}_{\text{dead P}} + \underbrace{\mu_{ZZ}Z^2}_{\text{dead Z}} - \underbrace{\mu_D D}_{\text{remineralisation}} - \underbrace{w_s \frac{\partial D}{\partial z}}_{\text{sinking}} \\
\text{sms}(N) &= \underbrace{\mu_D D}_{\text{remineralisation}} + \underbrace{\gamma_2 Z}_{\text{excretion}} + \underbrace{\mu_P P}_{\text{dead P}} - \underbrace{\bar{J}\,(z,t,N)\,P}_{\text{P growth}} .
\end{aligned}$$

Here, $\bar{J}\,(z,t,N)$ is average daily phytoplankton growth as a function of depth (z), time (t) and nutrient concentration (N), and the function $G(P)$ uses several biological inputs to determine the rate at which phytoplankton is predated by zooplankton.

Figure 2.1 shows more clearly the transfers of nitrogen between compartments in OG99NPZD, and Table 2.1 lists and explains the input parameters. The notation for these matches that in Oschlies and Garçon (1999). The maximum and minimum columns show appropriate regions in for each input variable. These do not reflect where OG99NPZD can be run, for in many cases it can perform far outside these bounds, but rather regions in which the true system value, or 'best' value at which to run OG99NPZD, almost certainly lies. These judgements were made by John Hemmings, of the National Oceanography Centre.

Name	Parameter	Max.	Min.
γ_1	Zooplankton food assimilation efficiency	0.3	0.95
a	Maximum photosynthetic rate at temp = 0°C (day^{-1})	0.3	3
α	Initial slope of P-I curve (d^{-1}/ (Wm^{-2}))	0.05985	0.06615
b	Max. photosynthesis - base factor for temperature variation	fixed at 1.066	
c	Max. photosynthesis - variation of temperature factor exponent ((°C)$^{-1}$)	0	1.25
γ_2	Excretion rate (days^{-1})	0	0.1
k_w	Light attenuation due to water (m^{-1})	fixed at 0.04	
k_c	Light attenuation due to phytoplankton (m^{-1} (mmolm^{-3})$^{-1}$)	fixed at 0.03	
ϵ	Prey capture rate $\left(\left(\text{mmolm}^{-3}\right)^{-2}\text{d}^{-1}\right)$	0.5	5
g	Maximum grazing rate (day^{-1})	0.25	2.5
C_{pp}	Ratio of phytoplankton to pigment (molNg^{-1})	0.4	0.6
PAR	Ratio of PAR to total downwelling solar irradiance at sea surface (P)	0.387	0.473
μ_P	Specific phytoplankton mortality rate (day^{-1})	0	0.1
μ_{PP}	Density dependent phytoplankton mortality (d^{-1}(mmolNm^{-3})$^{-1}$)	0	0.1
μ_{ZZ}	Density dependent zooplankton mortality (d^{-1}(mmolNm^{-3})$^{-1}$)	0	0.6
μ_D	Remineralisation rate (day^{-1})	0.025	0.075
K_1	Half-saturation conc. for DIN uptake (mmolm^{-3})	0.05	1
w_s	Sinking velocity (m d^{-1})	2	30

Table 2.1: *OG99 input parameters.*

The notation here matches that in Oschlies and Garçon (1999).

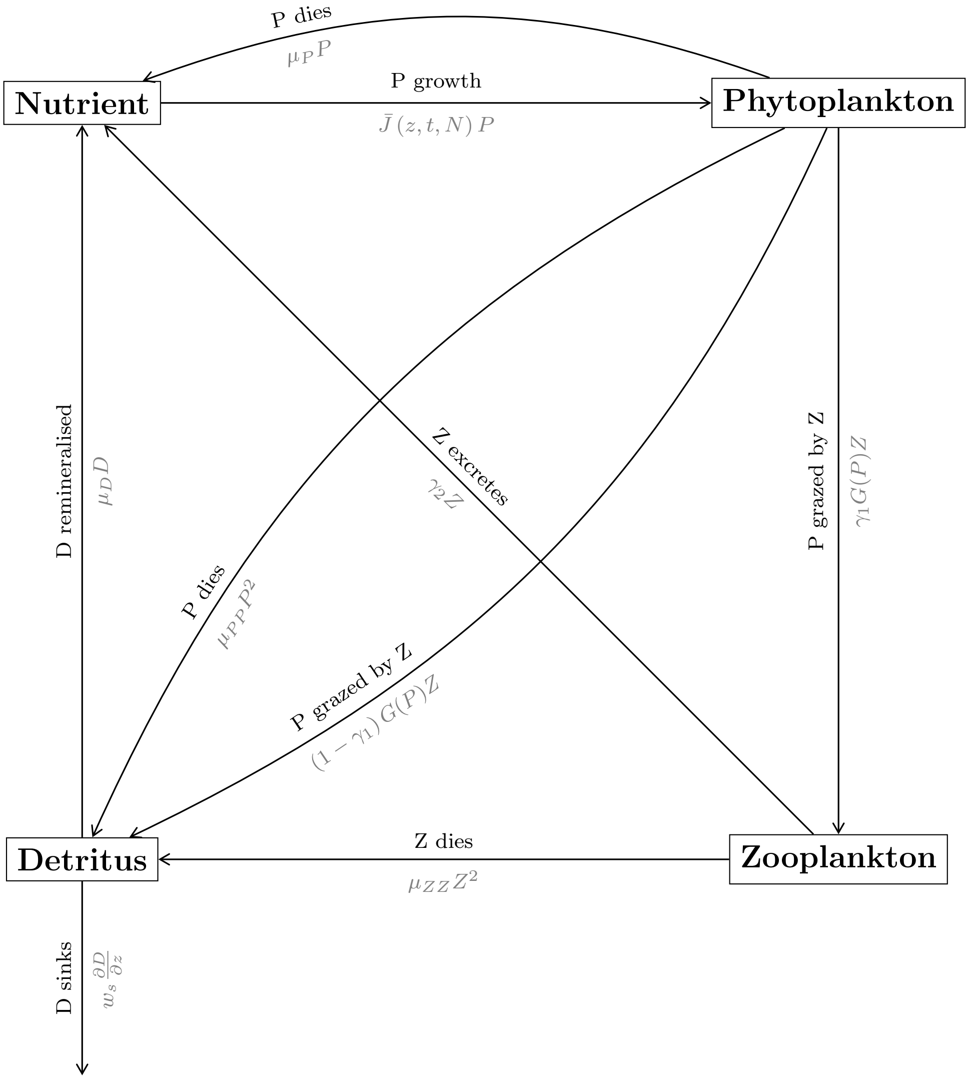


Figure 2.1: *Transfers of nitrogen modelled by the Oschlies-Garçon simulator*

2.3 Hadley Centre Ocean Carbon Cycle model (HadOCC)

Conceptually, HadOCC (Palmer and Totterdell, 2001; Hemmings et al., 2008) splits into three components; a physical model of the ocean, a model of inorganic carbonate chemistry and a biological model. The processes modelled by the first two of these are relatively well understood, but the biology, and in particular the process by which carbon is exported to the deep ocean, is more difficult.

The physical model is similar to that used in OG99NPZD, but detailed approximations to the coastlines and the topography of the ocean floor are also included. Tracers are subject to diffusion and advection, and are mixed using the same mixed layer scheme as that in OG99NPZD. Forcing functions are used for heat, wind-stress, salinity, sea surface temperature, fluxes of freshwater, precipitation and evaporation. The values for these may come from observed data or from ocean or climate simulators.

HadOCC adds two compartments to those used by OG99NPZD; dissolved inorganic carbon (DIC) and total alkalinity. Their dynamics are derived from the concentrations of the other four biological tracers, and so we can treat the biological component of HadOCC as having the same four compartments as OG99NPZD. Although the concentrations of nutrient, phytoplankton, zooplankton and detritus are governed by nitrogen, they are also considered in terms of carbon content, determined by fixed carbon:nitrogen ratios. Sometimes, in order to maintain these ratios, HadOCC is forced to ‘throw away’ some nitrogen or carbon.

The overall changes in concentrations of biological tracers in terms of nitrogen are determined by equations of the form

$$\frac{dT_i}{dt} = \text{advection} + \text{diffusion} + \text{mixing} + \text{sinking} + \text{biology}.$$

Here, the focus is on the biological components,

$$\begin{aligned}
\left.\frac{\partial P}{\partial t}\right|_{\text{biol}} &= \underbrace{RP}_{\text{P grows}} - \underbrace{G_{\text{p}}}_{\text{P eaten}} - \underbrace{mP^2}_{\text{P dies}} - \underbrace{\eta P}_{\text{P respires}} \\
\left.\frac{\partial Z}{\partial t}\right|_{\text{biol}} &= \underbrace{G_z}_{\text{Z grazes}} - \underbrace{\left(\mu_1 Z + \mu_2 Z^2\right)}_{\text{Z dies}} \\
\left.\frac{\partial D}{\partial t}\right|_{\text{biol}} &= \underbrace{m_D P^2}_{\text{P dies}} + \underbrace{\frac{1}{3}\left(\mu_1 Z + \mu_2 Z^2\right)}_{\text{Z dies}} + \underbrace{E_D}_{\text{Z excretes}} - \underbrace{\lambda D}_{\text{remineralisation}} - \underbrace{G_d}_{\text{D eaten}} \\
\left.\frac{\partial N}{\partial t}\right|_{\text{biol}} &= - \underbrace{RP}_{\text{P grows}} + \underbrace{\left(m - m_D\right) P^2}_{\text{P dies}} + \underbrace{\eta P}_{\text{P respires}} \\
&\quad + \underbrace{\frac{2}{3}\left(\mu_1 Z + \mu_2 Z^2\right)}_{\text{Z dies}} + \underbrace{\left(G_p + G_d - G_z - E_D\right)}_{\text{N released during grazing}} + \underbrace{\lambda D}_{\text{D remineralised}} .
\end{aligned}$$

In these equations, R is a function of nitrogen and several of the biological parameters governing phytoplankton growth from photosynthesis, and G_{p} and G_{d} determine

Name	Parameter	Values	Default
`co2sys`	CO2 system option	0: off, 1: on	1
`rcchlopt`	C:Chl option	0: fixed, 1: dynamic	0
`chltracer`	chlorophyll tracer option	0: off, 1: on	0
`nh4tracer`	NH4 tracer option	0: off, 1: on	0
`vsupply`	vertical nutrient supply within biology step	0: off, 1: on	0
`dsinkopt`	implementation of detrital sinking	0: external, 1: internal	1

Table 2.2: *HadOCC switch inputs*

the grazing of phytoplankton and detritus by zooplankton. The processes modelled by HadOCC are shown in Figure 2.2, and the input parameters are described in Tables 2.2 and 2.3. HadOCC includes several 'switch' variables, listed in Table 2.2, which turn model features on or off, or choose between different parameterisations of processes. For example the `photopt` variable determines which photosynthesis submodel is used to calculate R, and the switch `rcchlopt` determines whether the carbon:chlorophyll ratio is fixed or varying.

The notation used in Figure 2.2 and in the biological equations comes from Palmer and Totterdell (2001) where possible, and failing that, Hemmings et al. (2008). However MarMOT, our implementation of HadOCC, introduced in Section 2.4, fixes the input space to that in Table 2.3, and so from here on the names in Table 2.3, which match those in MarMOT, will be used. Most of the quantities in Figure 2.2 are derived from combinations of the input parameters in Table 2.3, but where they match a particular input this is indicated in the table. Again, the maximum and minimum values reflect the judgements of John Hemmings, our ocean ecosystem model expert.

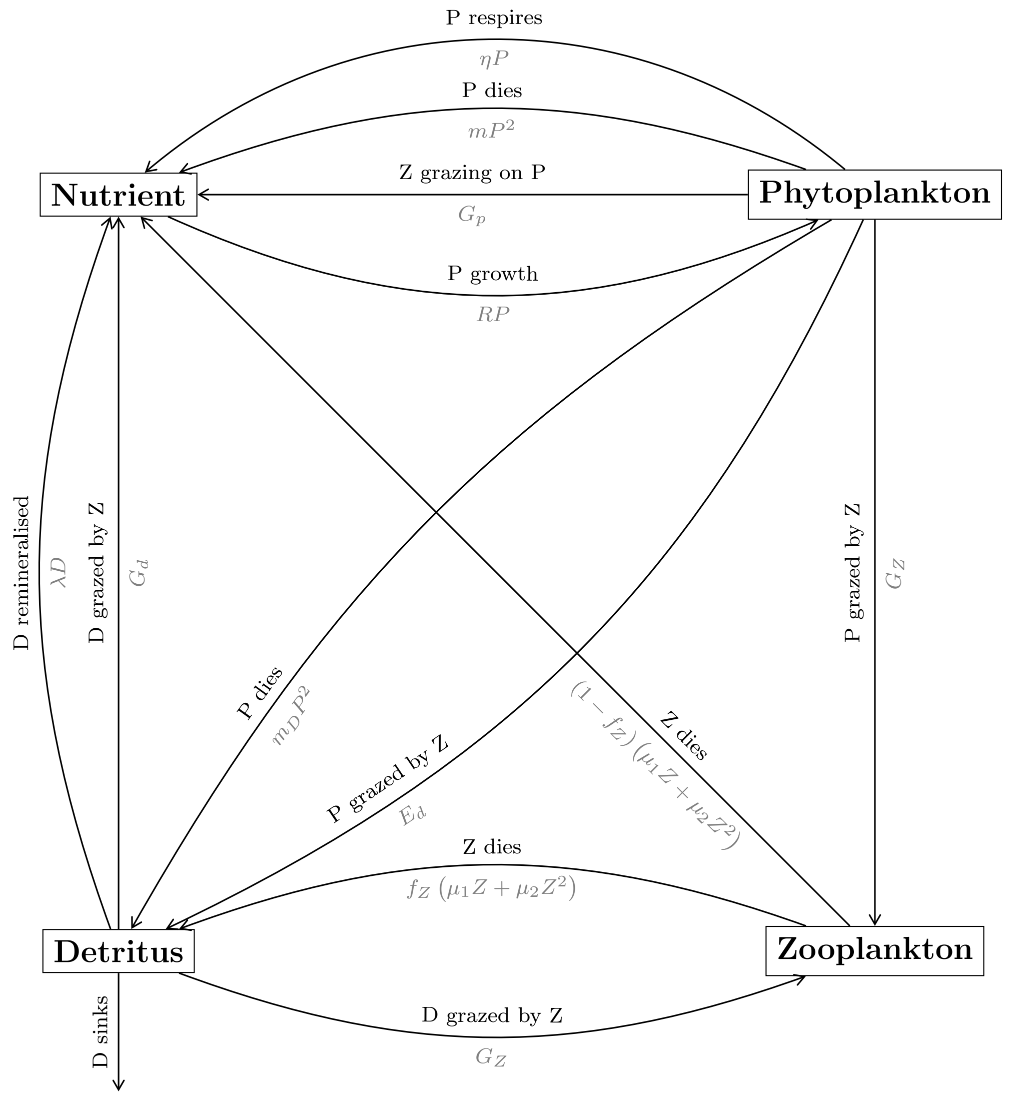


Figure 2.2: *Transfers of nitrogen modelled by HadOCC*

Table 2.3: *HadOCC input parameters. The notation here comes from the MarMOT code (see Section 2.4), and does not match exactly to Palmer and Totterdell (2001), Hemmings et al. (2008) or the HadOCC code. The quantities in Figure 2.2 in whose calculation the input is used is noted in the 'used in' column, with an equals sign if the parameter is the value of a quantity. When there is no entry in the 'Used in' column, the parameter is involved in most or all of the processes. This table spans the following two pages.*

Name	Parameter	Max.	Min.	Used in
`rcchl`	C:Chl ratio (if fixed) (mgC/mgChl)	20	200	

Name	Parameter	Max.	Min.	Used in
`rcchlmin`	Minimum C:Chl ratio (mgC/mgChl)	20	200	
`rcchlmax`	Maximum C:Chl ratio (mgC/mgChl)	20	200	
`rcnphy`	C:N ratio for phytoplankton	5.3	7.95	
`rcnzoo`	C:N ratio for zooplankton	4.5	6.75	
`rcndet`	C:N ratio for detritus	6	9	
`rparsol`	Ratio of PAR to total downwelling solar irradiance at sea surface	0.387	0.473	
`rchlpig`	Ratio of chlorophyll to total pigment	0.64	0.96	
`photmax`	Maximum photosynthetic rate (d^{-1})	1	3	R
`alphachl`	Initial slope of photosynthesis v irradiance curve (mg C (mg Chl)$^{-1}$ (E m^{-2})$^{-1}$)	2.78	8.33	R
`kdin`	Half-saturation conc. for nutrient uptake (mmol N m^{-3})	0.05	1	
`presp`	Phytoplankton specific respiration (d^{-1})	0	0.1	$= \eta$
`pmortdd`	Conc. dependent phytoplankton specific mortality (d^{-1}(mmol N m^{-3})$^{-1}$)	0	0.1	m, m_D
`pminmort`	Threshold for phytoplankton mortality (mmol N m^{-3})	0	0.02	m, m_D
`fpmortdin`	Fraction of phytoplankton mortality to DIN	0	0.3	m, m_D
`gmax`	Maximum grazing rate (d^{-1})	0.25	2.5	G_P, G_Z, G_D
`holling`	Holling function exponent for grazing model (integer)	Fixed at 2		G_P, G_Z, G_D
`epsfood`	Prey capture rate (d^{-1}(mmol N m^{-3})$^{-n}$)	0.5	5	G_P, G_Z, G_D
`fmingraz`	Food threshold for grazing function (mmol N m^{-3})	0	0.2	G_P, G_Z, G_D
`fingest`	Fraction of grazed material ingested	0.5	1	G_P, G_Z, G_D
`betap`	Zooplankton assimilation efficiency for phytoplankton	0.3	0.95	G_P, G_Z
`betad`	Zooplankton assimilation efficiency for detritus	0.3	0.95	G_Z, G_D
`fmessyd`	Fraction of messy feeding to detritus	0	1	G_P, G_Z, G_D
`zmort`	Base zooplankton specific mortality (d^{-1})	0	0.1	$= \mu_1$
`zmortdd`	Conc. dependent zooplankton specific mortality (d^{-1}(mmol N m^{-3})$^{-1}$)	0	0.6	$= \mu_2$
`fzmortdin`	Fraction of zooplankton mortality to DIN	0.2	1	$= f_Z$

Name	Parameter	Max.	Min.	Used in
`nitrifeuph`	Nitrification rate of ammonium in euphotic zone (d^{-1})	0	0.05	
`nitrifaph`	Nitrification rate of ammonium below euphotic zone (d^{-1})	0	0.05	
`dsink`	Detritus sinking velocity (m day^{-1})	2	30	D sinks
`rco3pprod`	Carbonate precipitated per unit primary production	0.0065	0.0195	

2.4 MarMOT

The Marine Model Optimization Testbed, or 'MarMOT', described in detail in Hemmings and Challenor (2011), is a piece of software developed to facilitate the analysis of plankton models. Of particular interest are the uncertainties and model error that often make inferences about the real system so unreliable. MarMOT incorporates HadOCC and OG99NPZD, and enables the user to run both models at multiple input points, making the same choices about vertical gridding, time scale, forcing functions, and other aspects of the simulators not relating to the biological parameters.

On a practical level, MarMOT makes the way both simulators are run very similar, so that input parameter sets, options tables and time, depth and output choices are made using files of the same format for both simulators.

The choices of vertical gridding and time scale in particular mean that output variables from both simulators are comparable in terms of time and space. The user can also choose the output variables returned by the simulators, and there are many variables available to both models.

Naïvely, MarMOT may appear to unify HadOCC and OG99NPZD to such a level that parts of their input spaces can be identified with one another. In the MarMOT input files, many of the biological parameters have names matching one another. For example the OG99NPZD input μ_{ZZ} is labelled `zmortdd` in the MarMOT input files, implying that it has the same meaning as the HadOCC input of the same name. While the meanings are clearly linked, as both are density dependent zooplankton mortality rates, the different parameterisations of zooplankton mortality along with other processes in HadOCC and OG99NPZD mean that we cannot simply treat the two input variables as corresponding. It is partly for this reason that the chosen simulator input notation is from different sources, to emphasise that at least initially we must treat these as two entirely independent input spaces. The idea of simulator difference is an interesting one, and we will return to it in Section 4.2.

Both simulators are able to give either profile output, where the variable's value at each depth level is given for each time step, or scalar output, which has only one value per time step. Some profile variables can be depth-integrated, through MarMOT, in order to give a sort of across-depth average value. The prefix "`iz.`" indicates that an output is the depth-integrated version of a profile output variable.

2.5 Choosing output variables

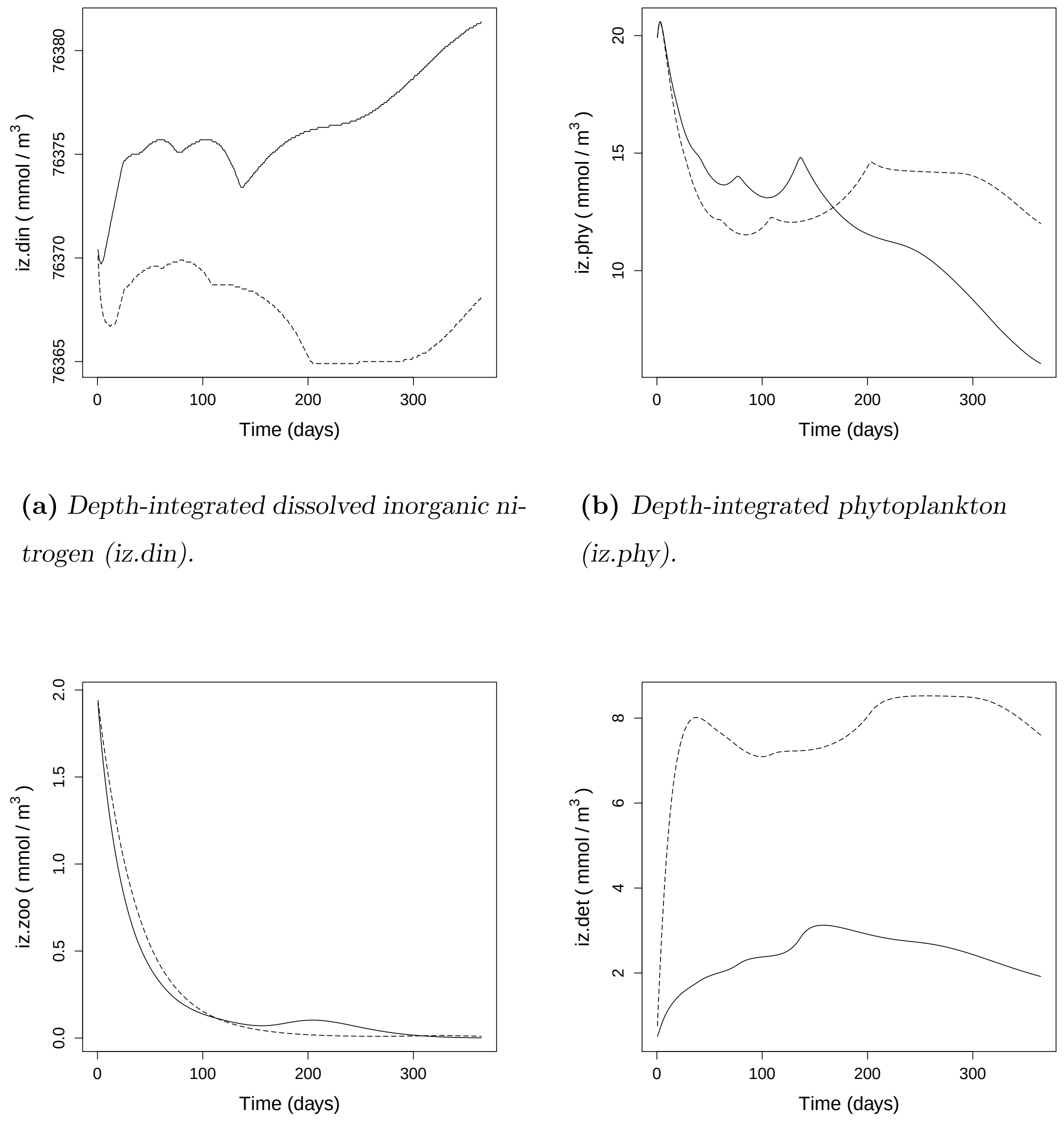
The output variables fit into various sorts of categories, some of which can only be produced by one of the simulators. For example, many of the outputs can be produced in terms of carbon by HadOCC, using its C:N ratios, but not by OG99NPZD. As our goal is the comparison of simulators, these outputs are of no use to us.

Both simulators can return the concentrations (in terms of nitrogen) of the four tracers (nutrient, phytoplankton, zooplankton and detritus) at each depth level and time-step, and can calculate depth-integrated values, summarising the concentration over the depths to produce a scalar output. Figure 2.3 shows an example of HadOCC and OG99NPZD output for each of the four tracers.

There are also linked quantities, for example particulate organic nitrogen (`pon`), whose value is the sum of phytoplankton, zooplankton and detritus, and chlorophyll (`chl`), which is derived from the phytoplankton concentration.

We will often be concerned only with scalar output, and will standardise the time scale to have one output per day for a year. This means that for a particular simulator and output variable, each input point corresponds to a time series of 365 points.

There are different motivations for focussing on a particular output variable for



(a) *Depth-integrated dissolved inorganic nitrogen (iz.din).*

(b) *Depth-integrated phytoplankton (iz.phy).*

(c) *Depth-integrated zooplankton (iz.zoo).*

(d) *Depth-integrated detritus (iz.det).*

Figure 2.3: *Time series outputs for the depth-integrated versions of the four main tracers. The input points were the two 'default' points, one for each simulator. The solid line corresponds to the HadOCC run and the dashed line to the OG99NPZD run.*

emulation. If one wishes to predict the value of the system at new inputs, it would make sense to choose an output representing an interesting feature of the system. Many of the outputs from HadOCC and OG99NPZD are specifically linked to the biological pump mentioned earlier, for example biologically driven vertical carbon transport (`ctranbio`), and so were we to use the simulators to mimic the effect of the ocean carbon cycle, this might be an appropriate choice.

Sometimes emulators are used to calibrate the models, and find regions of the input space that are 'plausible' in light of observed system data. In this case, the output variable must be one for which we have access to observed system data, so that the emulator can be used to find regions of input space that are likely to lead to similar values. The Bermuda Atlantic Time Series experiment (BATS) (Steinberg et al., 2001) collects data from the geographical region in which we are notionally running HadOCC and OG99NPZD. One of their measurements is analogous to particulate organic nitrogen (`pon`). This can be output by both simulators, and so would potentially be a good choice of emulator output variable were one intending to find plausible input regions.

The goal of this thesis is to create methods to emulate and compare multiple simulators, and so the choice of output should facilitate that. In Chapter 5 we emulate depth-integrated chlorophyll (`iz.chl`), because the two simulators being compared differ in their parameterisation of the carbon:chlorophyll ratio, and so the effect on `iz.chl` is more pronounced than on many of the other outputs. In Chapter 6, we will emulate depth-integrated particulate organic nitrogen (`iz.pon`).

2.6 Summary

We have introduced two compartmental ocean ecosystem models, OG99NPZD and HadOCC. While these have different input spaces, and model the system in different ways, using MarMOT we are able to produce the same output variables on the same spatial and temporal scales. Our chosen output variables for the remaining chapters are depth-integrated chlorophyll concentration (`iz.chl`) and depth-integrated particulate organic nitrogen (`iz.pon`), two scalar variables producing time series of daily values for a year for each input point.

Later we will become interested in the processes at work in both simulators, and so will return to some of the details explained in this chapter. Our next step, however, is to build an emulator of a single simulator, and this is the focus of the next chapter.

Chapter 3

Emulation

Many, though by no means all, of the issues raised surrounding complex simulators arise from their being very slow to run, and having high-dimensional input domains. Emulation confronts these by building a statistical model of the simulator. At first glance, building a model of the simulator may seem somewhat counter-productive. However, it will hopefully become clear that it can be hugely beneficial.

An emulator is a statistical representation of our beliefs about a simulator which, rather than giving a single precise output for a given set of input points $\mathbf{x}$, as the simulator does, gives a probability distribution for the simulator's output $s(\mathbf{x})$. Not only do we obtain the mean of the distribution, which is the emulator's approximation to $s(\mathbf{x})$, but in the variance of the distribution we also have a measure of how certain we are of the approximation.

If the emulator is constructed in such a way that it is much faster to compute than the original simulator then we can obtain many more approximate values of the simulator than we could realistically obtain true values, and if the approximation is a good one then we can confidently use these values for our analysis. See Craig et al. (1997, 2001); Kennedy and O'Hagan (2001); Kennedy et al. (2006); O'Hagan (2006); Sacks et al. (1989); Santner et al. (2003) for more on this, and for a helpful introduction to Bayesian emulation.

In this chapter we introduce emulation and explain the process of building an emulator of a simulator.

3.1 A brief overview

To build an emulator, we begin by representing the simulator output $s\left(\mathbf{x}\right)$ (which at this point we will assume to be single-valued) for a collection $\mathbf{x}$ of n input points by a function $f(\mathbf{x})$,

$$f(\mathbf{x}) \quad = \quad \sum_{i=1}^{q} \xi_{\mathrm{i}}\left(\mathbf{x}\right)\beta_i + \epsilon\left(\mathbf{x}\right). \tag{3.1}$$

Here, $\boldsymbol{\beta}$ is a vector of coefficients, about which we are uncertain, the $\xi_{\mathrm{i}}\left(\mathbf{x}\right)$ are known functions of $\mathbf{x}$ and $\epsilon\left(\mathbf{x}\right)$ is a correlated error term. To simplify notation we can write the regression term

$$\sum_{i=1}^{q} \xi_{\mathrm{i}}\left(\mathbf{x}\right)\beta_i$$

using a design matrix $\mathbf{X}$, such that for a set of input points $\mathbf{x} = x_1, \ldots, x_n$

$$\mathbf{X} = \begin{pmatrix} \xi_1\left(x_1\right) & \ldots & \xi_{\mathrm{k}}\left(x_1\right) \\ \vdots & & \vdots \\ \xi_1\left(x_n\right) & \ldots & \xi_{\mathrm{k}}\left(x_n\right) \end{pmatrix}.$$

Thus Equation 3.1 becomes

$$f(\mathbf{x}) = \mathbf{X}\boldsymbol{\beta} + \epsilon\left(\mathbf{x}\right).$$

Our emulator should meet two requirements (O'Hagan, 2006):

1. At any of the points $x_1, \ldots, x_n$, where we already know exactly the output of the simulator, we should have

$$f(x) = X\boldsymbol{\beta} + \epsilon(x) = s\left(x\right),$$

and the emulator's distribution should have zero variance, because we know with certainty that the simulator will always produce this same value[1]. We refer to these input and output points, for which we know the simulator's behaviour, as *training data*, because we use them to 'train' the emulator.

[1] Recently work has been done to emulate stochastic simulators, for example Henderson et al. (2009), for which this is not the case, but in this thesis simulators will always be assumed to be deterministic, and in the applications we have in mind that is very often the case.

2. For any input point x not in $\{x_1, \ldots, x_n\}$, the emulator's approximation should be plausible in view of our beliefs and the data we have, and the probability distribution should reflect our uncertainty about what the simulator may do at this point.

Verifying the first of these is simple. Checking the second requires more care, and some validation techniques for univariate emulators are described in Section 3.5.

The correlated error term is modelled by a Gaussian process,

$$\epsilon\left(\mathbf{x}\right) \sim N\left(\mathbf{0}, \sigma_\epsilon^2\, \Sigma(\mathbf{x})\right),$$

where σ_ϵ^2 is an unknown variance parameter and $\Sigma(\mathbf{x})$ the matrix of correlations between the error at each pair of input points. That $\epsilon\left(x\right)$ has a Gaussian Process distribution means that for any collection of points $x_1, \ldots, x_n$ in the domain of x, the joint distribution of $\epsilon\left(x_1\right), \ldots, \epsilon\left(x_n\right)$ is multivariate normal.

Combining this structure with Equation 3.1 gives

$$f(\mathbf{x}) = \mathbf{X}\boldsymbol{\beta} + \epsilon\left(\mathbf{x}\right) \mid \boldsymbol{\beta},\, \sigma_\epsilon^2 \sim N\left(\mathbf{X}\boldsymbol{\beta},\; \sigma_\epsilon^2\, \Sigma(\mathbf{x})\right).$$

We would like the correlation between simulator output at two points to depend on how far apart the points are, and so we introduce a correlation function $\rho\left(\cdot, \cdot\right)$ such that for two input points $x_i,\; x_j$

$$\Sigma(\mathbf{x})_{ij} = \operatorname{cor}\left(\epsilon(x_i), \epsilon(x_j)\right) = \rho\left(x_i, x_j\right).$$

Usually the correlation function will involve some parameters, which give us control over the function's behaviour. Because of the deterministic nature of the simulator, we require that

$$\Sigma\left(\mathbf{x}\right)_{ii} = \rho\left(x_i, x_i\right) = 1,$$

enforcing the property that once we know the simulator's output $s\left(x_i\right)$ at a particular point x_i, the emulator can predict it with certainty. As pairs of points become further away from one another, the correlation between their errors tends to zero. We will return to the correlated error term in more detail in Section 3.3.3

Our goal is to be able to emulate the simulator's behaviour at new inputs $\tilde{\mathbf{x}} = \{\tilde{x}_1, \ldots, \tilde{x}_m\}$ using the training data. This is equivalent to finding the posterior distribution

$$s\left(\tilde{\mathbf{x}}\right) \mid s\left(\mathbf{x}\right).$$

To do this, we need a prior distribution for $\{\sigma_\epsilon^2, \beta\}$. Options for this abound, and it is difficult to know which distribution is appropriate. The choice is often made for convenience, and in the case where we have a reasonably large amount of well-designed data, it is not crucial to the reliability of the emulator.

A popular choice is the conjugate normal inverse-gamma prior,

$$p(\boldsymbol{\beta}, \sigma_\epsilon^2) \propto (\sigma_\epsilon^2)^{-\frac{d+2}{2}} \exp\left[-\frac{a}{2\sigma_\epsilon^2}\right] (\sigma_\epsilon^2)^{-1} \exp\left[-\frac{1}{2\sigma_\epsilon^2}\left(\beta - \boldsymbol{\beta}_0\right)' B_0^{-1}\left(\beta - \boldsymbol{\beta}_0\right)\right]$$

which splits up to give

$$\begin{aligned} \tau_\epsilon = (\sigma_\epsilon^2)^{-1} &\sim \Gamma\left(\frac{d}{2}, \frac{a}{2}\right) \\ \beta \mid \sigma_\epsilon^2 &\sim N_2\left(\boldsymbol{\beta}_0, \sigma_\epsilon^2 B_0\right), \end{aligned}$$

where d, a, $\boldsymbol{\beta}_0$ and B_0 are to be specified, ideally through expert elicitation. The derivation of the posterior distribution

$$p\left(\boldsymbol{\beta}, \sigma_\epsilon^2 \mid s\left(\mathbf{x}\right)\right)$$

for the univariate case with the normal inverse-gamma prior can be found on page 330 of O'Hagan and Forster (2004).

One might also use its weak form

$$p\left(\boldsymbol{\beta}, \sigma_\epsilon^2\right) \propto \frac{1}{\sigma_\epsilon^2},$$

which puts infinite prior variance on $s\left(x\right)$.

In situations where simulator data is much more scarce, the choice of prior becomes a crucial part of the process, and much care is taken to ensure it is a wise one (Rougier, 2009).

Having chosen a prior distribution we can derive the posterior distribution of the simulator's output $s\left(\tilde{\mathbf{x}}\right)$ for any new set $\tilde{\mathbf{x}}$ of input points. For either of the prior choices above, we find that

$$s\left(\tilde{\mathbf{x}}\right) \mid s\left(\mathbf{x}\right)$$

has a location-scale multivariate t-distribution. There is a relatively straightforward separable extension of this is to emulate multivariate output, and Section 3.2 derives the distribution in this case.

3.2 The emulator's distribution

If a simulator has more than one output variable of interest, we can emulate them jointly from the input variables. Extending the univariate framework described above introduces various choices regarding the flexibility of the relationships between the different outputs within the emulator. Wherever multivariate emulators are built in this thesis, they are built around the following framework, which is explained in more detail by Conti and OHagan (2010).

Our training data contains n input points $\mathbf{x} = \{x_1, \ldots, x_n\}$ and the $n \times k$ matrix $\mathbf{s}(\mathbf{x})$ of associated output, whose i^{th} row is the vector $\mathbf{s}(x_i) = (s_1(x_i), \ldots, s_k(x_i))$.

The marginal emulators all take the form

$$f_i(\cdot) = \mathbf{X}\beta_i + \epsilon_i(\cdot)$$

with q common regression functions $\xi_1(\cdot), \ldots, \xi_{\mathrm{q}}(\cdot)$ (including a constant term) used to form the design matrix $\mathbf{X}$ (an $n \times q$ matrix), and the same covariance function $\rho(\cdot, \cdot)$ for $\epsilon(\cdot)$, but with different coefficient vectors β_i, which together form the $q \times k$ matrix $\mathbf{B}$. We also introduce a covariance matrix Γ between the outputs, so that for a single new input x_i the multivariate emulator has the distribution

$$\mathbf{f}(x_i) \mid \mathbf{B}, \Gamma, \Theta \sim N_k\Big(X\mathbf{B},\ \rho(x_i, x_i)\,\Gamma\Big).$$

Using the non-informative prior

$$\pi(\mathbf{B}, \Gamma \mid \Theta) \propto |\Gamma|^{-\frac{k+1}{2}},$$

and working through, we find the emulator's posterior distribution for the simulator's output at a new input point $\tilde{x}$ to be

$$\mathbf{s}(\tilde{x}) \mid \Theta, \mathbf{s}(\mathbf{x}) \sim t_{n-q}\Big(\mathbf{m}(\tilde{x}), \mathbf{c}(\tilde{x}, \tilde{x})\,\hat{\Gamma}\Big).$$

This can be generalised for a collection $\tilde{\mathbf{x}} = \{\tilde{x}_1, \ldots, \tilde{x}_q\}$ of m new input points to give the matrix t-distribution

$$\mathbf{s}(\tilde{\mathbf{x}}) \mid \Theta, \mathbf{s}(\mathbf{x}) \sim T_{m,k}\left(n-q, \mathbf{m}(\tilde{\mathbf{x}}), \mathbf{c}(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}), \hat{\Gamma}\right),$$

where

$$\begin{aligned}\mathbf{m}(\tilde{\mathbf{x}}) &= \tilde{\mathbf{X}}\hat{\mathbf{B}} + \left(\mathbf{s}(\mathbf{x}) - \mathbf{X}\hat{\mathbf{B}}\right)' A^{-1}\rho(\cdot, \tilde{\mathbf{x}}) \\ \mathbf{c}(\tilde{x}_i, \tilde{x}_j) &= \rho(\tilde{x}_i, \tilde{x}_j) - \rho(\tilde{x}_i, \cdot) A^{-1} \rho(\cdot, \tilde{x}_j) \\ &\quad + \left[\tilde{\mathbf{X}}_i - \mathbf{X}A^{-1}\rho(\cdot, \tilde{x}_i)\right]' \left(\mathbf{X}A^{-1}\mathbf{X}\right)^{-1} \left[\tilde{\mathbf{X}}_j - \mathbf{X}A^{-1}\rho(\cdot, \tilde{x}_j)\right]\end{aligned}$$

and

A is the spatial correlation matrix for the training data, $A_{ij} = \rho(x_i, x_j)$

$\hat{\mathbf{B}} = \left(\mathbf{X}' A^{-1}\mathbf{X}\right)^{-1} \left(\mathbf{X}' A^{-1}\mathbf{s}(\mathbf{x})\right)$ is the generalised least squares estimate of $\mathbf{B}$

$\hat{\Gamma} = \dfrac{1}{n-q}\left(\mathbf{s}(\mathbf{x}) - \mathbf{X}\hat{\mathbf{B}}\right)' A^{-1}\left(\mathbf{s}(\mathbf{x}) - \mathbf{X}\hat{\mathbf{B}}\right)$ is the GLS estimator of Γ

$\rho(\cdot, \tilde{x}_i) = \left(\rho(x_1, \tilde{x}_i), \ldots, \rho(x_n, \tilde{x}_i)\right).$

There are various ways to deal with the correlation lengths Θ, some of which will be discussed in Section 3.3.3. As we will see in detail in Chapter 7, this process can be coded using an object-oriented structure.

Having outlined the basic structure of an emulator, we will now turn to examine each stage of its construction in more detail. In Chapter 6, we will build multivariate emulators, but for now we will continue with emulators of a single output variable.

3.3 Building an emulator

3.3.1 Design of experiments

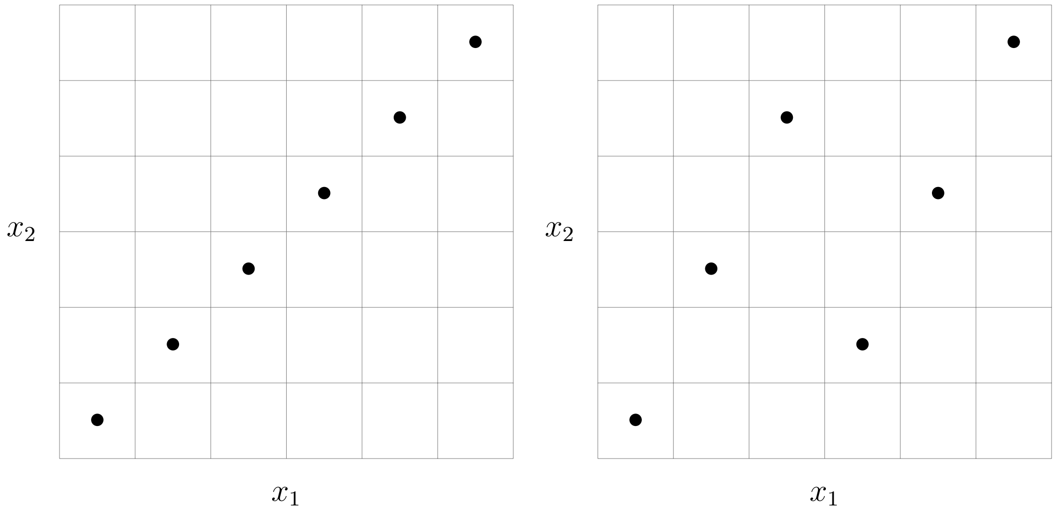
Before an emulator can be constructed, one needs training data. In many circumstances, the statistician building the emulator has little or no input into how the training data is constructed, but happily for this thesis we were able to run the simulator ourselves, and therefore had complete control over the input values included in each training data set.

As seen in Chapter 2, HadOCC has around thirty inputs, and initially we chose to vary all of these. To place a point in every corner of this input space would therefore require around $2^{30} \approx 1.1 \times 10^9$ points. Clearly this makes covering the input space with any satisfactory density extremely difficult. Many of our emulators were built with around 1,000 points, and so strategic choice of these points was imperative.

The subject of experimental design is well established, and there are many standard techniques across various industries and research areas, such as engineering, agriculture and medicine. However, in order to maximise the information contained in the training data, experiments must be tailored to fit the nature of deterministic simulators. Whereas, for example, in a clinical trial the same drug dose and patient criteria will not yield the same results every time, deterministic simulators behave otherwise. Therefore although experimental design techniques tailored towards clinical trials replicate input points, this would simply waste simulator runs. It is also likely that many of the simulator's inputs have little effect on some output variables. Were we to run the simulator at two points which had the same value in all but the i^{th} dimension, the output would inform us only about the effect of the i^{th} input. If it turns out that changes in this input alone have little influence, the simulator has effectively been evaluated at the same point twice. For this reason, designs such as factorial or fractional factorial designs, which contain no replicates of points but do replicate values of each input variable, are often seen as inappropriate.

Prompted by these issues while designing experiments for a fluid flow simulator, McKay et al. (1979) proposed Latin Hypercube sampling. To construct a n-point Latin hypercube design (LHD) over k dimensions, one first divides the range of each dimension into n intervals of equal probability (usually under a uniform distribution, so that they are equally spaced), and samples a single value from each interval. Design points are then formed by matching at random a value from each dimension. If the LHD is projected onto any subset of the k input dimensions (this is effectively what happens when some input variables have no effect), the remaining dimensions are still well covered, with no duplicated points.

Since 1979, much work has been done on creating LHDs that are optimal in some sense, focussing mostly on making them as space-filling and as orthogonal



(a) *Perfectly correlated LHD*

(b) *A more space-filling and orthogonal LHD*

Figure 3.1: *Two 6-point, 2-dimensional LHDs, constructed using different methods.*

as possible. Even with a valid LHD, care must be taken to ensure that the design allows as much information as possible to be drawn from the simulator. For example, the LHD in Figure 3.1a, with perfectly correlated points, is completely unable to distinguish the effect of x_1 from the effect of x_2. While in practice randomness and large size would prevent a hypercube with such poor properties from occurring, this demonstrates the need to ensure good orthogonality properties when building an LHD. Figure 3.1b shows an LHD with better space-filling and orthogonality properties.

Orthogonality is a popular criterion, which Iman and Conover (1982) and Owen (1994) showed to be optimal for numerical integration. Both propose algorithms for constructing near orthogonal LHDs. More recently, similar algorithms have been proposed by Bingham et al. (2009) and C. Devon Lin and Tang (2010).

Another challenge for the design of computer experiments is the vastness of the input space; in some way, we would like to span as much of the simulator's domain as possible. Morris and Mitchell (1995) proposed the *maximin* criterion with the aim of minimising posterior variance at any set of points whose output value had yet to be observed. They defined a *distance list*, comprising the distances between all pairs of points in the design, sorted from smallest to largest, and a corresponding *index list* containing the number of pairs separated by each distance. The designs termed 'Maximin' were those which maximised the smallest distance, and minimised the number of pairs separated by that distance. They then extended this criterion by working through the other elements of the lists to reach a unique LHD.

Where maximin LHDs are used here they have been constructed by an iterative local search type algorithm proposed by Grosso et al. (2009). This algorithm proceeds by making small perturbations to a Latin hypercube, then keeping any improvements (in terms of minimum pairwise distance), and rejecting changes that do not improve the design. The algorithm begins by generating an LHD with no constraints, and then cycling through the following steps:

1. Set $n_{iter} = 0$.

2. Find all *critical points* in the LHD. These are those that are separated from another point by the smallest distance (often this will just be the two points closest together)

3. Choose one of the critical points at random. Call this point p_c.

4. Choose one of the non-critical points at random. Call this point p_{nc}

5. Choose a dimension at random. Call this k_{iter}.

6. Swap the k_{iter}^{th} column of all points between p_c and p_{nc} in cyclic order, so that between p_c and p_{nc} the i^{th} value in the k_{crit}^{th} dimension becomes the $(i+1)^{st}$, and the first of p_c and p_{nc} in the design takes the value of the other.

7. If the minimum distance of the resulting LHD is greater than or equal to that of the previous design, keep this new LHD, reset n_{iter} to zero, and return to step 2.

8. If the new minimum distance is smaller, reject the new LHD, set $n_{iter} = n_{iter}+1$ and:

   - If $n_{iter} < 100$ (or some chosen value) return to step 2 (with the old LHD).

- If $n_{iter} = 100$ (or some chosen value) stop and keep the current LHD.

It is appealing to strike a balance between being space filling and orthogonal, since these two properties do not necessarily go hand in hand. Various algorithms are proposed doing just this, for example one in Joseph and Hung (2008). However, having built an LHD using a maximin algorithm, it is easy to check its orthogonality properties, and to reject it and start again if they are poor. A common way to proceed is to create a large number of designs and keep the best, a strategy that can be employed for any criterion.

We will return to the subject of experimental design in Chapters 5 and 6 when our needs will become more specific. As well as this, a new method for constructing very large LHDs which are built from many smaller LHDs is discussed in Section 3.5.1. This strategy will be used in Chapter 5 for a validation study.

3.3.2 Regression surface

Within the emulation framework there are many different sorts of emulators, and arguably one of the most pronounced differences is the place of the regression surface.

At one extreme, some choose to have only a single constant term, a surface of order zero, so that all the interpolation and prediction work falls to the correlated error term. This approach is shown in the examples in Oakley and O'Hagan (2004) and Sacks et al. (1989). This can lead to catastrophic errors in prediction if the correlated error term is poorly specified or the experiment poorly designed, since if a new point is too far away from the training data for the correlation to have a pronounced effect, the prediction begins to default to the constant mean (Kaufman et al., 2011). A conventional and simplistic choice is to include all terms linearly (Oakley and O'Hagan, 2004; Conti et al., 2009), enabling the regression function to capture some of the global variation.

As we mentioned briefly in Section 3.3.1, in most computer simulators, only a small selection of the input variables have a strong effect on the outputs, and so the notion of *active variables* was introduced. Here, the regression surface involves a subset $\mathbf{x}_A$ of the input variables $\mathbf{x}$. The functions $\xi_{\mathrm{i}}\left(\mathbf{x}_A\right)$ then tend to involve more complicated terms than the monomials described above, chosen such that they have considerable linear effects on $s\left(\mathbf{x}\right)$ (Craig et al., 1996). In this scenario, the emulator (for univariate output) becomes

$$s\left(\mathbf{x}\right)=\mathbf{X}_A\boldsymbol{\beta}_A+\epsilon\left(\mathbf{x}_A\right)+\mathrm{u}\left(\mathbf{x}\right),$$

where $\mathbf{X}_A$, $\boldsymbol{\beta}_A$ are the design matrix and coefficients for the new regression surface. The final term $\mathrm{u}\left(\mathbf{x}\right)$ is a *nugget*, an uncorrelated residual which 'soaks up' the error caused by leaving out some of the input variables from the regression surface and correlated error. This is necessary because in considering only some of the input dimensions the emulator no longer has the property that a particular simulator input point should always yield the same output. The emulator no longer functions as a perfect interpolator of the full training data, as it did when all input variables were included. If the emulator has been well designed, the variance of this nugget should be small, and the simplification of making the nugget uncorrelated will not matter (Craig et al., 1997).

To choose the $\xi_{\mathrm{i}}\left(\mathbf{x}\right)$ appropriately, one must, as far as possible, combine detailed knowledge of the simulator with a careful analysis of the training data (Craig et al., 1997). If simulator data is plentiful, then techniques such as stepwise model selection can be used (Cumming and Goldstein, 2010). Rougier (2009) observes that carefully chosen regression functions, incorporating expert knowledge of the simulator, are helpful when attempting to extrapolate beyond the convex hull of the training inputs. A more detailed regression surface also reduces the effect of the error introduced through assumptions made for the correlated error, covered in Section 3.3.3.

Clearly there is no 'correct' or 'best' choice of regression surface, and various criteria could be established by which to select a regression surface. Some favour parsimony (O'Hagan, 2006), while others prefer a surface with many terms (Rougier, 2009). Our approach will vary, but will hopefully be justified in each case.

3.3.3 Correlated error

The correlation between the errors at two input points x and x' is governed by the correlation function

$$\mathrm{cor}\left(\epsilon\left(x\right), \epsilon\left(x'\right)\right) = \rho\left(x, x'\right),$$

and the modelling of the error by

$$\epsilon\left(\mathbf{x}\right) \sim N\left(0, \sigma_{\epsilon}^{2}\, \Sigma\left(\mathbf{x}\right)\right).$$

This notion of a correlation function $\rho\left(\cdot,\cdot\right)$ and a single variance value σ_{ϵ}^{2} assumes that the correlation between the error at two points is determined solely by their position relative to one another, regardless of their absolute position in the input space. This is known as *stationarity.* How true this is rests largely on how well the regression functions have been selected.

In general, the functions commonly used for $\rho\left(\cdot,\cdot\right)$ assume that the simulator's output is smooth, which is usually, though not always, the case (O'Hagan, 2006). Indeed, choice of the correlation function $\rho\left(\cdot,\cdot\right)$ depends mainly on how smooth we wish the emulator to be. The Matérn class of correlations can be relatively 'rough', and offer a high degree of flexibility in terms of differentiability and local behaviour (Stein, 1999; Handcock and Wallis, 1994; Rougier, 2009), and are therefore popular with some. The limiting case of the Matérn correlation function, in terms of smoothness, is the Gaussian correlation function

$$\rho\left(x, x'\right) = \exp\left[-\left(x - x'\right)^{T} \Theta\left(x - x'\right)\right],$$

which we will use here, mainly for its convenient properties.

At the most general, we can assign a non-zero value for Θ_{ij} for all $i,\ j$, however usually this is simplified so that

$$\Theta_{ij} = 0 \text{ for } i \neq j,$$

making the function *separable.* Again, one would hope that any significant interactions between inputs are captured by the regression surface, and so this need not be a problem.

Assuming separability, we need only specify the diagonal elements θ_i of Θ, and therefore may write

$$\rho\left(x, x'\right) = \exp\left[-\sum_{i=1}^{p} \theta_i \left(x_i - x_i'\right)^2\right].$$

A further simplification is to make the function *isotropic*, by making all the correlation lengths equal, $\theta_i = \theta\ \forall i$. This is only viable if the input data have been rescaled so that the range of each input is the same. This extends the stationarity property by asserting that the correlation between errors at two points depends only on the magnitude of the distance between them, and not the direction.

Having decided upon a correlation function, we must attempt to deal with the correlation hyperparameters θ_i. Although technically in a Bayesian framework they are unknown parameters, they are often treated as fixed, and so the problem becomes one of estimation. Finding an appropriate value is crucial to accurate prediction; a small value will decrease the predictive variance, and the emulator may be overly confident, whereas a large one will cause the emulator to be too uncertain. The values can be validated by checking the performance of the emulator as a predictor, using simulator data that wasn't used to build the emulator (O'Hagan, 2006). A common approach, and the one used predominantly in this thesis, is to use maximum marginal likelihood estimation, where $\boldsymbol{\beta}$ and σ_ϵ^2 are integrated out in order to maximise the log-likelihood over θ. Diagnostics can be used to flag up problems arising from poorly chosen functions or misplaced assumptions (Bastos and O'Hagan, 2009), and these will be explored in Section 3.5.

3.4 Limitations of this approach

Emulation as described above is not automatically a good choice for any particular simulator. The model relies on assumptions which are not always appropriate.

Firstly, the model described in this chapter is only suitable if the simulator's output is continuous everywhere in the input space, or if the regression surface can be chosen such that it perfectly captures any discontinuities. If this is not the case, the training data may combine with these wrong assumptions to badly damage the emulator's predictions. It is wise to ask a simulator expert if they expect the output to be continuous, and whether given the output at a point $\mathbf{x}$, they would expect to be informed about the output at a very nearby point $\mathbf{x}'$ (Oakley, 2002).

The posterior distribution $s\left(\tilde{\mathbf{x}}\right) \mid s\left(\mathbf{x}\right)$ for a Gaussian process emulator is determined by the choice of prior $p\left(\beta, \sigma_\epsilon^2\right)$. As we have mentioned, this choice is usually biased towards convenience rather than an accurate representation of beliefs. It is rare that one sees a choice other than the conjugate Normal Inverse-Gamma or its weak form shown in Equation 3.2. Although neither of these will ever be correct, they have the advantage of being conjugate, and therefore leading to relatively simple computations. Expecting a simulator expert to specify 'the best' prior distribution for β and σ_ϵ^2, without limiting him to such a family, would be quite unreasonable.

In using the weak and non-informative prior,

$$p\left(\beta, \sigma_\epsilon^2\right) \propto \frac{1}{\sigma_\epsilon^2}, \tag{3.2}$$

the model asserts that we have no information at all about the coefficients β, and this is unlikely to be true. The Normal Inverse-Gamma prior allows the user to specify some information, even though it may have a high variance. This is usually done using a combination of two methods. Firstly, one can pose questions about the behaviour of the simulator at various inputs to an expert, and find parameters that fit these, a process known as *elicitation*, explained in more detail by Oakley (2002). Secondly, one can use simulator data itself to estimate appropriate parameter values. Craig et al. (1997) combine these approaches in their case study. When simulator training data is scarce, the specification of the prior distribution is crucial to making good predictions. The elicitation approach is a demanding one however, and in the absence of a dedicated expert and the presence of many simulator runs, a non-informative prior is a pragmatic choice.

The choice of correlation function further constrains the model, and is another source of contention. The Gaussian correlation function is often criticised for being too smooth. For example Rougier (2009) prefers the Matérn class, even though this leads to less tractable results. Constraining the correlated error to be separable and even isotropic is another potentially inappropriate simplification. It may be that in order to capture the behaviour of the simulator, off-diagonal terms must be included in the correlation matrix Σ, or there may at least be different levels of smoothness in different directions.

Although we have emphasised the emulator's efficiency compared to the simulator's, it is limited in one way that the simulator is not. To build an emulator requires the inverse or the factorisation of the correlation matrix $\Sigma(\mathbf{x})$. For training data containing n points, this is an $n \times n$ positive definite matrix. Rather than a straightforward inverse function, such as R's 'solve', the Cholesky decomposition can therefore be used, which improves stability and increases the number of points that can be handled. Even so, this operation limits the amount of training data an emulator can handle, and can still lead to numerical instability.

Kaufman et al. (2011) propose the use of a correlation function with finite support, such that for points sufficiently far apart the correlation function is zero. This makes the correlation matrix $\Sigma(\mathbf{x})$ sparse, and drastically increases the capacity of the emulator, through the use of sparse matrix techniques.

In general, building an emulator requires the arrangement and monitoring of a large number of quantities, and of the modelling choices made at each step. By the time one comes to using an emulator for prediction of a simulator's behaviour at some new points, the original training data, regression and correlation length specifications and so on could easily have become confused, or have been lost. Although these issues can be avoided by careful organisation, they are still real. With this in mind, we present an object-oriented framework for emulation, in Chapter 7, which enforces a tight structure on the entire emulation process. This framework also brings benefits in computational savings and in ease of adaptation. Indeed, once the core framework has been introduced, which fits around the techniques in this chapter, it will be extended to include the methods presented in the later chapters.

Any of the issues raised in this section would be rich areas for study. In this thesis however, the focus is on developing new and fairly general frameworks for emulating multiple simulators, rather than on building the best possible emulators for a particular setting. The methods developed in Chapters 5 and 6 can be used with any of the modelling choices described in this chapter, and so the choices made to illustrate them will often be fairly simple and pragmatic ones.

Having said the above, it is important to check the emulator's performance, in order to find any poorly specified values or poor modelling choices. We now therefore turn to emulator diagnostics.

3.5 Verification and validation

Having built an emulator, it is wise to check that it appears to be doing its job well. Emulation methods introduce various approximations and simplifications which, for some simulators, may not be at all appropriate. Even if the modelling choices were good, poor values may have been used for the correlation lengths, or for other numbers we have 'plugged in', such as parameters for prior distributions. A helpful summary of diagnostics for Gaussian process emulators is given by Bastos and O'Hagan (2009), which includes some measures not appearing in this section.

Using training data alone

It may be that the simulator is costly to run or that data is scarce, and so we would like to use all the data available to us to build our emulator. Rougier (2009) presents two diagnostics that use only the training data. The first is the "leave-one-out" method, where an emulator is built using all but one of the data points, and then used to predict the remaining data point. This is repeated for each point in the training data. This gives some indication of the emulator's performance across the input space, and gives a measure of the uncertainty we can expect in our predictions.

The second is the "one-step-ahead" technique, which builds up the emulator by using the prior specifications alone to predict the first data point, then an emulator built from the prior specifications and the first data point to predict the second, and so on. This shows us how fast the emulator learns from each piece of data. In this project we had access to plenty of simulator data, and so will not use these techniques, but instead ones which make use of some new simulator data.

Using new data

In terms of notation, we have an emulator built from data $(\mathbf{x}, s(\mathbf{x}))$ and a set of m validation runs of the simulator, $(\tilde{\mathbf{x}}, s(\tilde{\mathbf{x}}))$. For shorthand we will write $\tilde{\mathbf{s}}$ to indicate the new output $s(\tilde{\mathbf{x}})$, $\hat{s}(\tilde{\mathbf{x}})$ for the emulator's expected simulator output $\mathrm{E}(\tilde{\mathbf{s}} \mid \mathbf{s})$, and $\tilde{\mathrm{V}}(\tilde{\mathbf{x}})$ (sometimes shortened to $\tilde{\mathrm{V}}$) to denote the emulator's variance matrix $\mathrm{var}(\hat{s}(\tilde{\mathbf{x}}) \mid s(\mathbf{x}))$.

First of all, the individual errors

$$\hat{s}(\tilde{x}_i) - \tilde{s}_i$$

can give useful insight into the emulator's performance, particularly by studying their behaviour in relation to input and output variables.

Linked to this, a measure for comparing the predictions with the true values is the *root mean squared error*,

$$\mathrm{RMSE}(\tilde{\mathbf{s}}) = \sqrt{\frac{1}{m}\sum_{1}^{m}\left(\hat{s}(\tilde{x}_i) - \tilde{s}_i\right)^2}.$$

This should be as small as possible, and is a measure of the accuracy of the emulator's prediction without taking into account its variance.

To include the effects of variance, one can find the individual *standardised prediction errors*

$$\mathrm{SPE}(\hat{s}(\tilde{x}_i)) = \frac{\hat{s}(\tilde{x}_i) - \tilde{s}_i}{\sqrt{\tilde{\mathrm{V}}_{ii}}}.$$

If the correlation lengths are small enough relative to the ranges of the inputs, these should approximately follow a standard normal distribution. If there are some abnormally large values[2], they should be investigated, perhaps by running the simulator in the vicinity of the inputs in question, to see where this conflict between emulator and simulator arises.

A systematic trend in these errors away from $\mathrm{N}(0,1)$ can indicate a more systematic shortfall in the emulator. For example, if the SPE are mostly unexpectedly small (or large), this indicates that there may be a problem with the correlated error

[2] Bastos and O'Hagan (2009). suggest 2 as a cut off, whereas Craig et al. (1997) suggests 3

term. If there is a tendency for the SPE to be small (or large) near to training data points, then we may have plugged in poor values of the correlation lengths, leading to over (or under) estimates of variance. Otherwise there may be a problem with our estimation of the variance σ_{ϵ}^2.

If the errors are predominantly of the same sign, this indicates a poorly chosen regression function. The numerator of the SPE should centre around zero, but if the mean function of the emulator is not a good fit to the data, this will not be the case.

Systematic trends are often easier to spot using plots. If the errors show an obvious trend against emulator predictions $\hat{s}(\tilde{\mathbf{x}})$, rather than being randomly scattered around zero, there may be a problem with the mean function or with the estimation of the regression coefficients β_i. If there is evidence of heteroscedasticity, the stationarity assumption (induced by making the correlation dependent only on the distance between two points, see Section 3.3.3) may be ill founded.

Plotting the error against individual input variables can be enlightening, as trends can flag up effects of the inputs that are not adequately represented by the regression surface. These plots can also help pin down sources of problems with the correlated error term, if there are trends in the variance of the SPE for some inputs.

An emulator of the SPE values is a powerful validation tool. If a high proportion of variation can be explained, this suggests that there are systematic trends in the simulator that are not being captured by the emulator. In particular, there may be problems with the regression surface. Emulating the SPE using a more complex regression surface than that of the emulator of the output can particularly help to expose problems, or to show whether the regression surface in the output's emulator is appropriate. This tool will be used in the example in Section 6.6.

In order to include the covariance between emulator predictions, which the measures mentioned so far do not incorporate, Bastos and O'Hagan (2009) suggest looking at the *Mahalanobis distance*,

$$\mathrm{MD}(\tilde{\mathbf{s}}) = (\tilde{\mathbf{s}} - \hat{s}(\tilde{\mathbf{x}}))' \, \tilde{\mathrm{V}}^{-1} (\tilde{\mathbf{s}} - \hat{s}(\tilde{\mathbf{x}})) \,.$$

Under the assumptions made in emulation, this should have a scaled F distribution with m and $n-q$ degrees of freedom,

$$\frac{n-q}{m\left(n-q-2\right)}\,\mathrm{MD}\left(\tilde{\mathbf{s}}\right) \mid \mathbf{s},\Theta \sim F_{m,n-q},$$

where m is the number of points in $\tilde{\mathbf{x}}$, n is the number of points in the training data and q is the number of terms in the regression surface of the emulator.

Under these assumptions,

$$\begin{aligned}\mathrm{E}\left[\mathrm{MD}\left(\tilde{\mathbf{s}}\right)\right] &= m\\ \mathrm{var}\left[\mathrm{MD}\left(\tilde{\mathbf{s}}\right)\right] &= \frac{2m\left(m+n-q-2\right)}{n-q-4}.\end{aligned}$$

Clearly for each set of new inputs $\tilde{\mathbf{x}}$, there will only be one Mahalanobis distance, and so the distribution cannot be checked, but these summaries can be used to see how well the emulator fits the emulation model.

In the examples in Sections 3.6 and 5.5 and Chapter 6 we will use these diagnostic tools on real emulators, and examine plots that show features of the emulators in more detail.

3.5.1 A method for generating large LHDs

For validation studies, particularly in computer experiments, a large number of input designs can be very useful. Ideally, these should each have good space-filling and orthogonality properties, and the design as a whole, when they are combined, should also possess these features. However, as designs increase in size, particularly if they are being made to fit some optimality criterion, both generation and storage can become difficult. whole may have desirable properties, when divided into c designs of size m, these properties are not necessarily retained.

In this section, a novel method is introduced for constructing large Latin hypercubes in such a way that neither generation nor storage need become an issue. The method is compared to several alternatives, and is used to perform a validation study with one million input points in Chapter 5. The method turns out to be similar in its goal to that of Qian (2012).

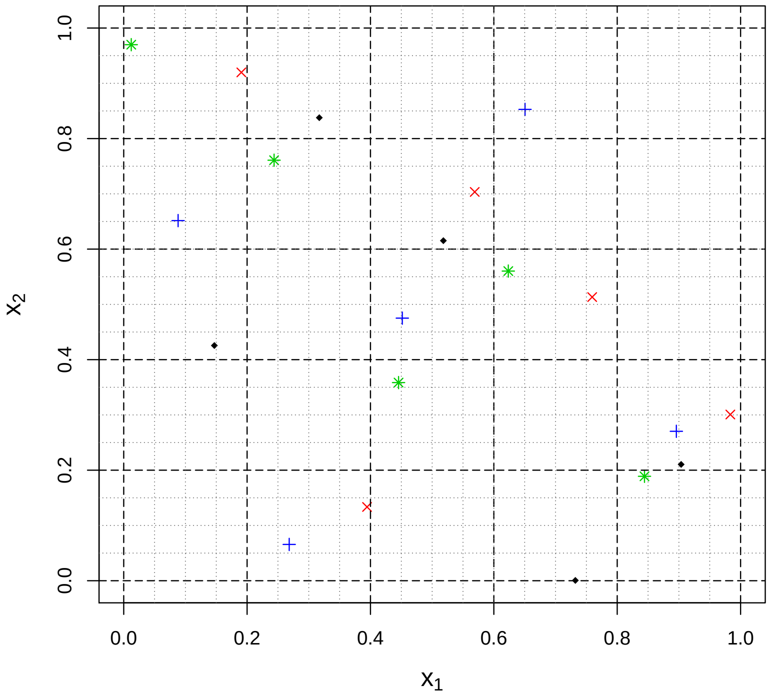


Figure 3.2: *A 20-point LHD with $K = 2$ produced using $c = 4$ and $m = 5$, arranged using a 4-point index LHD with 2 dimensions, whose columns are the vectors* $(3, 4, 2, 1)$, $(1, 3, 2, 4)$.

Staggered Latin Hypercube Designs

We will assume that the designs for the validation study are each of size m and dimension k, and that there are c of them in total. The product $m \times c$, the total size of the combined design, will be denoted by N.

The *staggered Latin hypercube* method uses multiple smaller LHDs to build a larger design which is itself an LHD. To build each column of the design, we split the interval $[0, 1]$ into m sub-intervals, each of which is then divided into c pieces. Altogether this divides $[0, 1]$ into N parts. Then for $i = 1, \ldots, c$ an m-point LHD is built whose co-ordinates can only be in a particular part of each of the m sub-intervals.

To avoid having a regimented structure in this design, rather than assign every point for sub-LHD i into the i^{th} piece of the sub-intervals in every dimension, a further Latin hypercube, the *index LHD*, is built, containing c points in k dimensions.

Then, the sub-interval in each dimension to which the points in the i^{th} sub-LHD are assigned is determined by the i^{th} row of the $c \times k$ index LHD. A staggered LHD is shown in Figure 3.2.

While this method ensures that both the N-point and m-point designs are Latin hypercubes, at no point does information about the entire design have to be kept together. This is a great advantage, since the memory required for a large design can severely limit the design sizes available. The sub-LHDs need not be stored or generated together, since the index LHD (which is usually relatively small) can be stored and used to generate each part of the design separately. Ensuring that these designs are LHDs allows us to claim the advantageous properties mentioned by Stein (1987), and the staggered method enables this to be done at relatively low cost.

We found that an effective strategy was to use distributed computing to generate the design, and a table in a database to store it, using the R package 'RMySQL' (James and DebRoy, 2010). Parts of the design can then easily be accessed and used in R (R Development Core Team, 2011).

Comparison Study

Table 3.5.1 compares summaries for designs built using the staggered LHD method with summaries for some intuitive alternative methods for building large designs. In this study, $m = 100,\ c = 100$ and therefore $N = 10^4$. The final two options both use the staggered LHD method, one by building unconstrained LHDs and one using the maximin algorithm described in Section 3.3.1. We were interested to see whether there was any positive or detrimental effect to the overall design by imposing the maximin criterion on each sub-design, and also to see how the properties of the sub-designs compared.

The methods, as numbered in Table 3.5.1 are

1. Generate c unconstrained LHDs each of size m using (imposing no constraint),

2. Generate c maximin LHDs each of size m using the algorithm from Grosso et al. (2009) (explained in Section 3.3.1),

Method	Maximum Correlation		Minimum pairwise distance	
	Over N points	Over sets of m points	Over N points	Over sets of m points
1	0.0251 (0.0398)	0.374 (0.465)	0.156 (0.102)	0.258 (0.157)
2	0.0234 (0.0354)	0.357 (0.472)	0.153 (0.0824)	0.565 (0.523)
3	0.0247 (0.0405)	0.374 (0.445)	0.153 (0.0957)	0.254 (0.169)
5	0.0246 (0.0365)	0.376 (0.462)	0.151 (0.0889)	0.255 (0.161)
6	0.0233 (0.0338)	0.361 (0.410)	0.153 (0.0998)	0.561 (0.507)

Table 3.1: *Summaries of designs built by the methods listed above, where each design contains $c = 100$ chunks of size $m = 100$, and has dimension $k = 10$. The figures shown are the means over 100 repetitions of the experiment, with the worst figure given in brackets.*

3. Generate one N-point unconstrained LHD and, using random sampling, split it into c chunks of size m,

4. Generate one N-point maximin LHD and, using random sampling, split it into c chunks of size m,

5. Generate a staggered design with c chunks of size m, where each chunk is an unconstrained LHD,

6. Generate a staggered design with c chunks of size m, where each chunk is maximin.

Of these, the overall designs created by the first two methods are not LHDs, and nor are the sub-designs created by the third and fourth. The fourth method, which involves the generation of a maximin LHD of size N, is unrealistic for large designs, and so is not shown in the results table.

It seems that the staggered LHD method, shown in the bottom two rows of Table 3.5.1, produces overall designs with similar orthogonality and space-filling properties to those created by the other methods mentioned. Whether the sub-LHDs were created using the maximin algorithm seems to make little difference to the overall N-point design in terms of correlation or minimum pairwise distance.

This method could be developed by larger studies, and by investigations into the difference made by the relative sizes of m and c. It could also prove useful for the generation of training data, as well as validation data, for methods such as the sparse correlation emulators proposed by Kaufman et al. (2011), where a much larger set of training data can be used to build an emulator.

The size of N in this study is limited by the difficulty in finding the minimum distance over many points, rather than by the methods themselves (except the third and fourth, which involve storing the whole design at once). The staggered LHD method will be used in the example in Chapter 5 for a validation study of size $N = 10^6$, with $m = c = 1000$.

3.6 Example: OG99NPZD and HadOCC

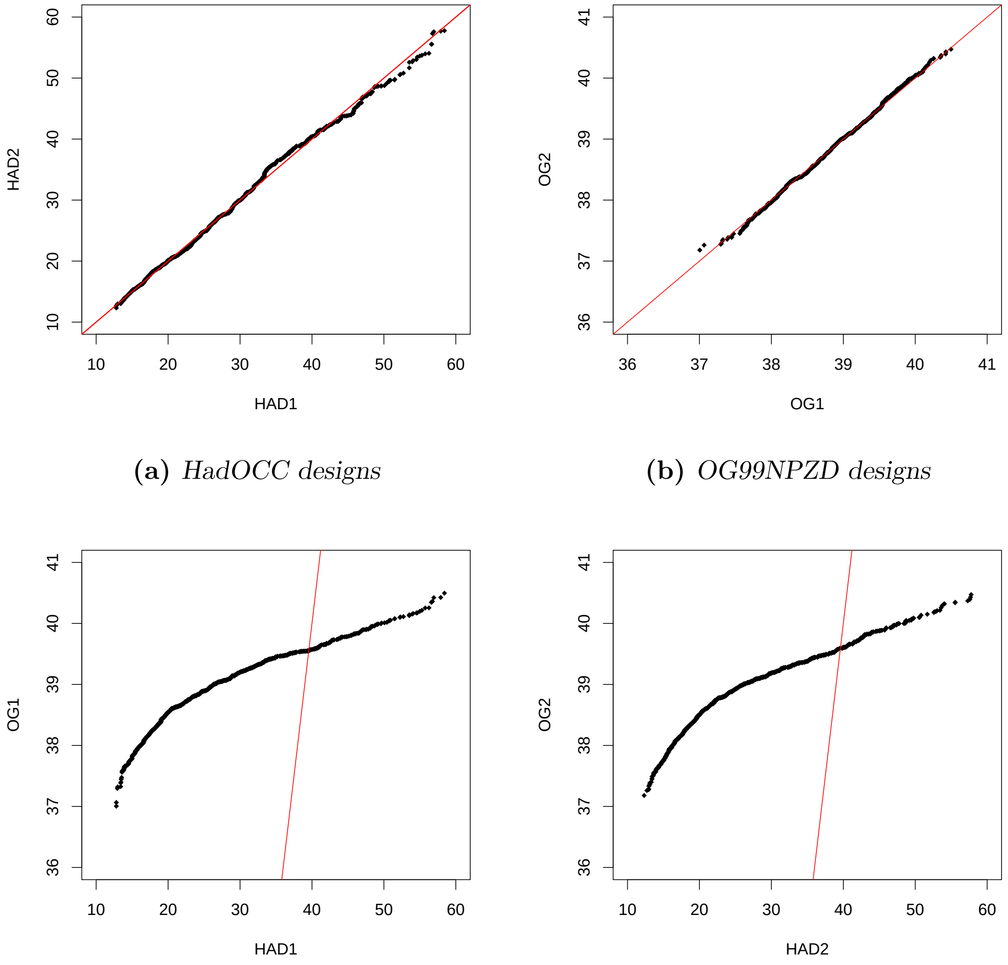
Having outlined methods for emulation we can now use them to emulate the two simulators introduced in Chapter 2, OG99NPZD and HadOCC.

The output variable here is the annual mean of `iz.chl`. This can be produced by both simulators, with the same units and over the same spatial and temporal region, and hence in such a way that the outputs should be expected to correspond in meaning.

For input designs, four 1,000 point maximin Latin hypercubes were created, two for each input space, which were named OG1, OG2, HAD1 and HAD2. It was checked that each was roughly orthogonal.

The simulator input variables are listed in Tables 2.1 and 2.3, and the ranges listed there were used to build these designs. As HadOCC has 27 varying inputs (with `rcchlopt` set to zero) and OG99NPZD only 15, clearly there will be a much higher density of points in the OG99NPZD input design.

As well as for validating the emulators, a motivation for building two designs for each simulator was that they should provide some indication of the dependence of choices and estimates on the design. For example, if the two designs for HadOCC lead to a different subset of active variables being chosen, or to very different estimate regression coefficients or correlation lengths, this should raise concerns.



(a) *HadOCC designs*

(b) *OG99NPZD designs*

(c) *HAD1 and OG1*

(d) *HAD2 and OG2*

Figure 3.3: *Comparing OG99NPZD and HadOCC output data (annual mean* `iz.chl`*), after sorting each design's output into numerical order. The straight line in each plot has gradient one and intercept zero.*

3.6.1 Emulating OG99NPZD and HadOCC

Before beginning to build an emulator, it is sensible to look at the simulator output itself. Figure 3.3 compares the output values for the four designs, both within and across simulators, and shows HadOCC and OG99NPZD to behave very differently.

The Box-Cox procedure suggested that for HadOCC mean `iz.chl` should be log-transformed, but that OG99NPZD should use the untransformed mean. The datasets OG1 and HAD1 were used as training data, and OG2 and HAD2 for validation.

Initially, the approach taken to emulate OG99NPZD and HadOCC was a simple and fairly standard one. All input variables were included as active, and a single correlation length was estimated using maximum likelihood for each emulator. The regression surface was linear and included all inputs.

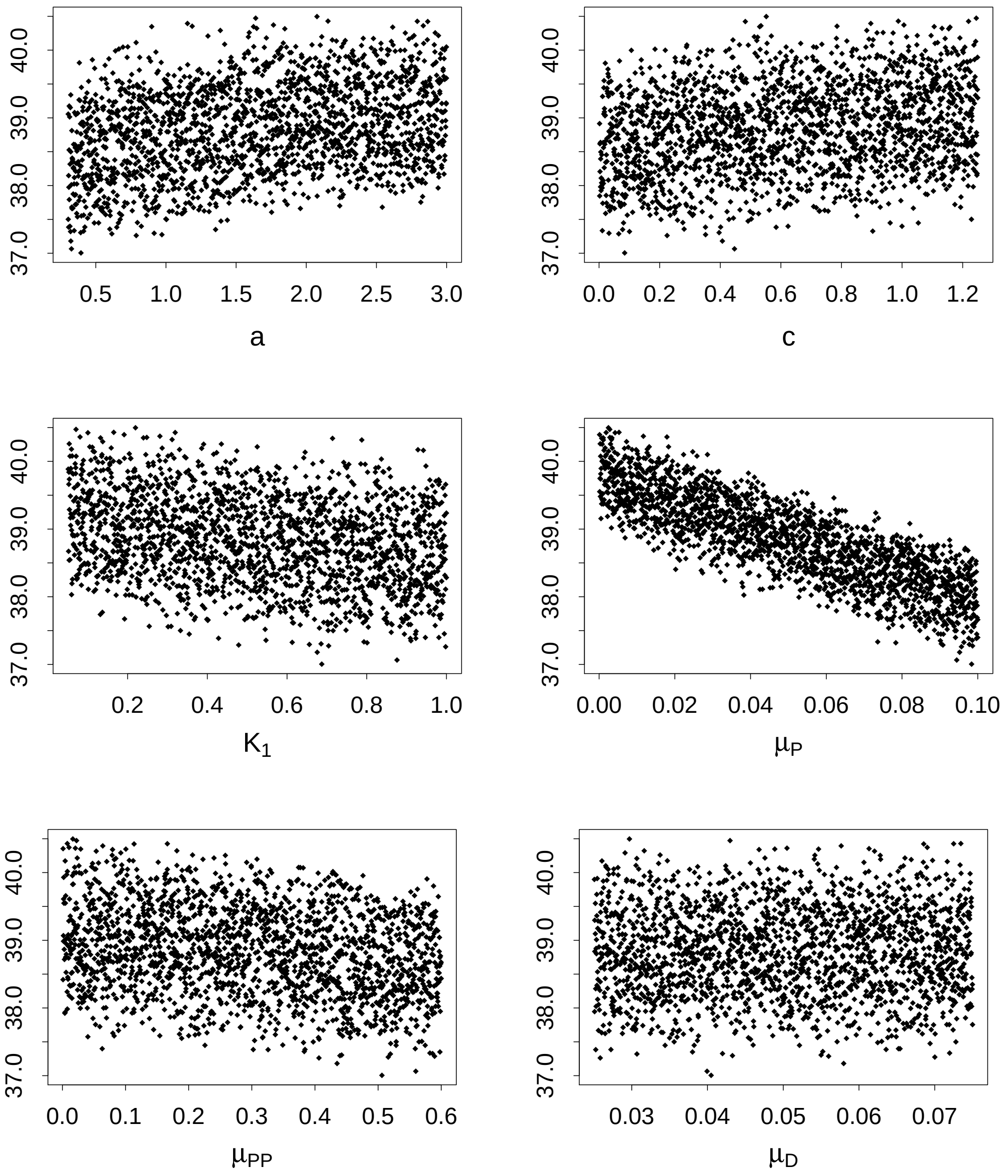


Figure 3.4: *Some main effects plots for OG99, for OG1 and OG2 data combined. These show the output plotted against some input variables.*

Having chosen regression functions, namely the first order polynomial with all input variables included, the training data was used to find posterior summaries of the coefficients, showing crudely the effect of each input variable on the simulator output. Although all inputs were included in the emulators, for both simulators only a small subset showed any significant effect. For OG99NPZD, the effect of μ_P is clearly the most pronounced, followed by a, K_1 and μ_{PP}, whereas μ_D appears to have little marginal effect. For HadOCC, the most active variable was `rcchl`, and to a lesser extent `rcnphy`.

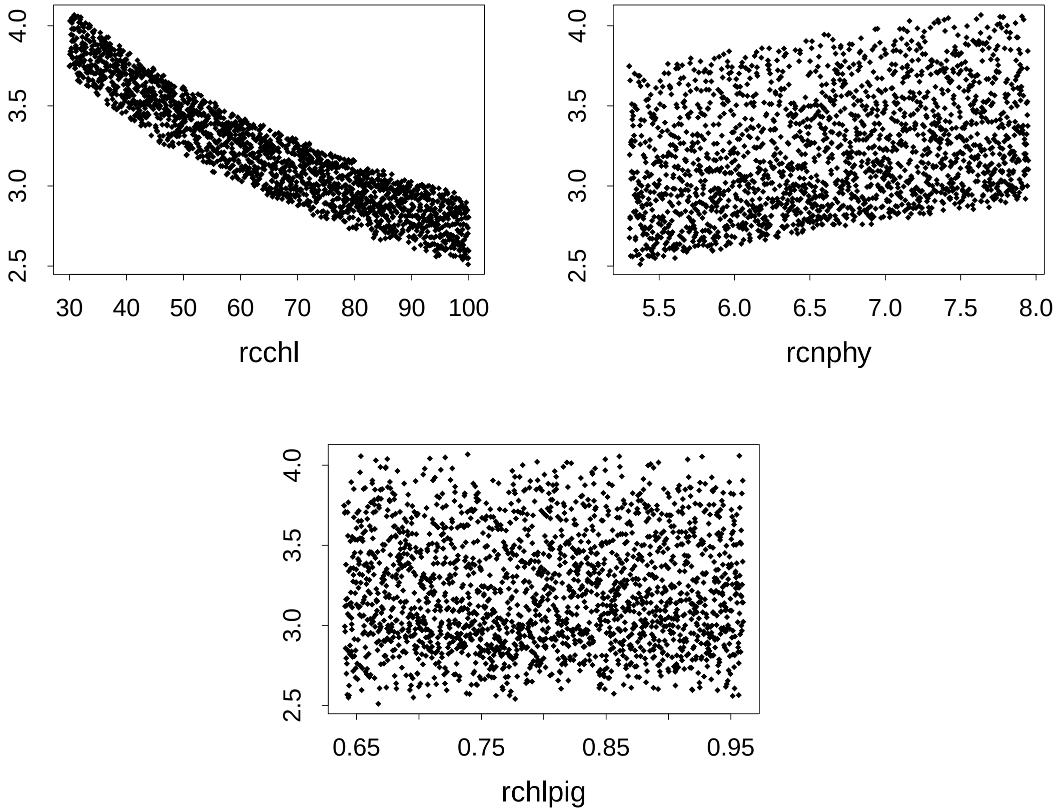


Figure 3.5: *Some main effects plots for HadOCC, for HAD1 and HAD2 data combined. These show the output, annual mean* $\log$ (`iz.chl`)*, plotted against three input variables.*

These can be seen from the main effect plots in Figures 3.4 and 3.5, as well as from the posterior expectations for the coefficients β in Table 3.2. The 'relative difference' column shows the difference of the two coefficients for each input, divided by the largest coefficient for the pair of emulators (the constant terms for OG1 and

HAD2).

	OG1	OG2	Relative difference
Const.	39.61	39.58	-0.00060
α	0.06	0.04	-0.00060
a	0.66	0.73	0.00186
γ_1	0.03	-0.02	-0.00138
c	0.32	0.32	-0.00007
w_s	-0.05	-0.04	0.00037
ϵ	0.02	-0.05	-0.00189
g	-0.04	0.01	0.00108
K_1	-0.40	-0.39	0.00027
μ_P	-1.92	-1.93	-0.00031
μ_D	-0.04	-0.01	0.00069
PAR	0.15	0.17	0.00048
C_{PP}	0.10	0.03	-0.00196
γ_2	0.00	0.08	0.00181
μ_{ZZ}	0.00	-0.01	-0.00017
μ_{PP}	-0.64	-0.60	0.00080

Table 3.2: *Posterior coefficients for both OG99NPZD emulators.*

	HAD1	HAD2	Relative difference
Const.	3.5804	3.6477	0.0185
`rcchl`	-1.3206	-1.3365	-0.0044
`rcnphy`	0.4066	0.4143	0.0021
`rcnzoo`	-0.0072	0.0101	0.0047
`rcndet`	0.0006	-0.0057	-0.0017
`rparsol`	0.0018	0.0097	0.0022
`rchlpig`	-0.0000	-0.0045	-0.0012
`photmax`	0.0162	0.0023	-0.0038
`alphachl`	0.0134	0.0219	0.0023
`kdin`	-0.0075	-0.0354	-0.0076
`presp`	-0.0481	-0.0621	-0.0038
`pmortdd`	-0.0097	-0.0022	0.0021
`pminmort`	0.0089	0.0098	0.0003
`fpmortdin`	0.0044	0.0014	-0.0008
`gmax`	0.0009	0.0058	0.0013
`epsfood`	-0.0154	-0.0015	0.0038
`fmingraz`	-0.0116	0.0043	0.0043
`fingest`	-0.0005	0.0043	0.0013
`betap`	-0.0017	-0.0144	-0.0035
`betad`	0.0031	-0.0150	-0.0050
`fmessyd`	0.0046	-0.0058	-0.0028
`zmort`	0.0048	-0.0164	-0.0058
`zmortdd`	0.0110	0.0122	0.0003
`fzmortdin`	0.0108	0.0147	0.0011
`nitrifeuph`	-0.0177	0.0018	0.0053
`nitrifaph`	0.0151	-0.0035	-0.0051
`dsink`	-0.0064	-0.0079	-0.0004
`rco3pprod`	0.0008	-0.0129	-0.0038

Table 3.3: *Posterior coefficients for both HadOCC emulators.*

A more thorough sensitivity analysis could be done, enabling us to assess the influence of each input, using the techniques of Saltelli et al. (2000). In particular the 'correlation ratio' described on p28-29, where $\text{var}(Y)$ (the variance of the output) is compared to $\text{var}(Y \mid x_i = x_i')$ (the variance of the output when input x_i is fixed at x_i') to discern the effect of input x_i, might be helpful in this setting.

When two separate emulators are built for each simulator, one using each training design, the correspondence between posterior coefficients is high, shown by the 'relative difference' columns in Tables 3.2 and 3.3. This implies that the amount of data in each set of training data is sufficient to capture the major trends. The estimated correlation lengths are also similar for both sets of training data. The OG99NPZD emulators have correlation lengths of 0.347 for OG1 and 0.314 for OG2, whereas the HadOCC emulators have lengths of 0.0150 and 0.0154 (except for the `rcchl` correlation length, which was later increased to 1.5).

Again, these are reassuringly similar for each pair of designs.

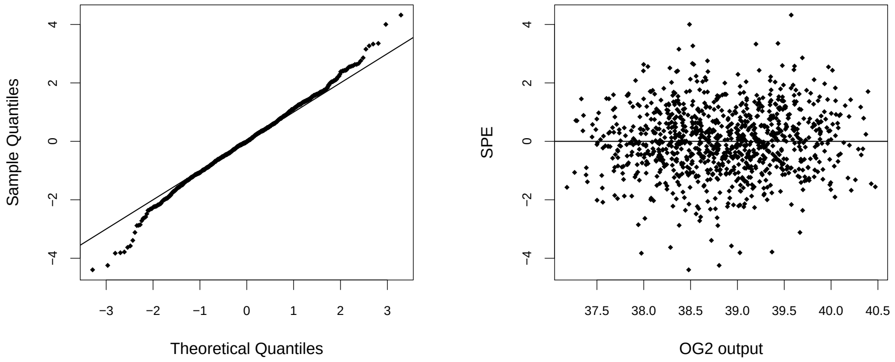


Figure 3.6: *Standard prediction error for the OG99NPZD emulator with a linear surface.*

3.6.2 Validating the emulators

Using some of the validation techniques discussed in Section 3.5, the performance of each emulator can be checked. Figure 3.6 shows some behaviour of the standardised prediction errors (SPE) for the OG99NPZD emulator[3].

[3] Many of the plots in this thesis contain a straight line. This is always either the line with gradient 1 and intercept 0, if two quantities are being compared (such as in Figure 3.3), or a

The values appear to be roughly $\mathrm{N}(0,1)$ and show no trend with output, which is ideal. Similar plots showing SPE against input variables or predicted output show no obvious trend, suggesting that a linear surface and an isotropic Gaussian correlation structure is a good choice for OG99NPZD.

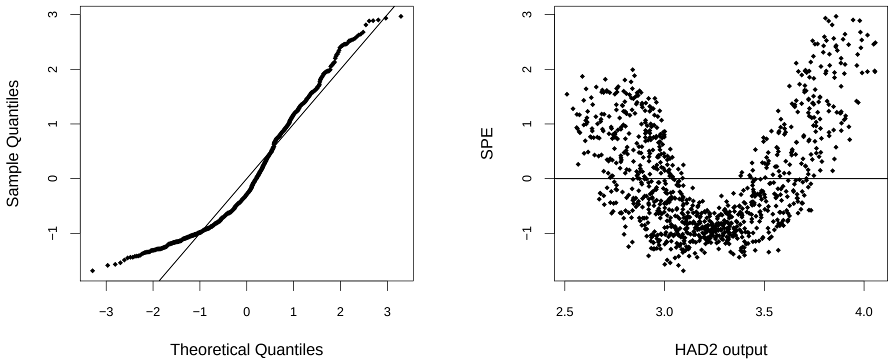


Figure 3.7: *Standard prediction error for the HadOCC emulator with linear surface and isotropic correlation lengths.*

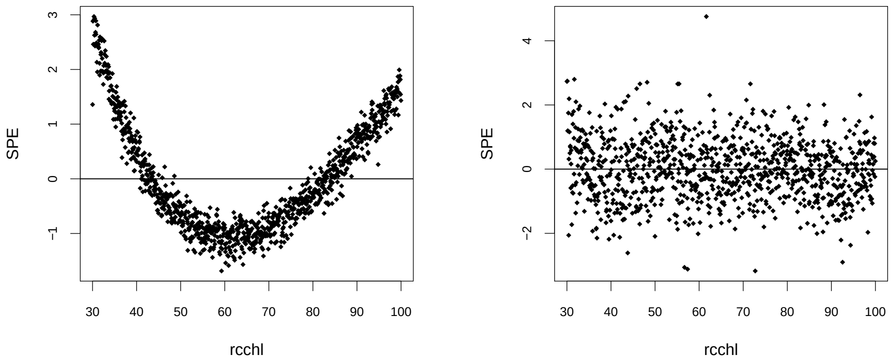


Figure 3.8: *Standard prediction error for HadOCC emulators with isotropic correlation (left) and a larger correlation length for* `rcchl` *(right).*

Figure 3.7 shows the same plots for the HadOCC emulator with a linear surface and isotropic correlated error. Unlike the OG99NPZD emulator this shows very

horizontal line at 0, usually to show whether some sort of error depends on another quantity (such as in Figure 3.8).

undesirable behaviour. The pattern in the SPE values with output suggests that the emulator is not capturing the behaviour of the simulator. Studying the behaviour of the SPE values in response to changes in input, it becomes clear that the non-linear effects of `rcchl` on output are not being captured. Increasing the correlation length for `rcchl`, θ_{rcchl}, to 1.5 removes this undesirable behaviour. Figure 3.8 shows SPE against `rcchl` for both emulators, and Figure 3.9 shows the general behaviour of the SPE with $\theta_{\text{rcchl}} = 1.5$.

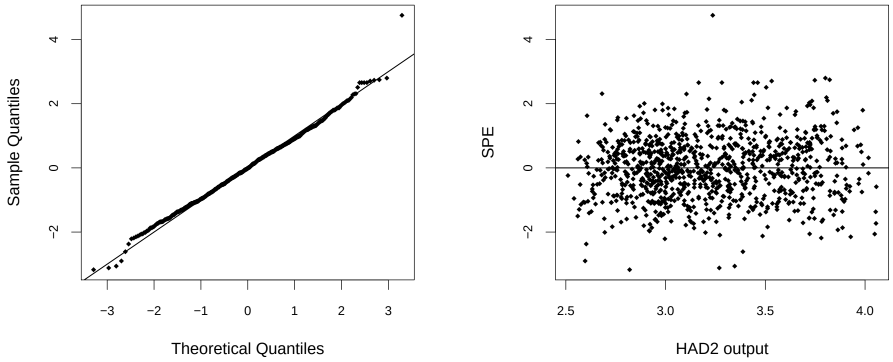


Figure 3.9: *Standard prediction error for the HadOCC emulator with a linear surface, but with a larger correlation length for* `rcchl`*.*

Figure 3.10 shows predictions and errors when these emulators are used to predict the simulators' behaviours at new input points (those in OG2 and HAD2).

3.6.3 Combining the emulators

Having built emulators of both OG99NPZD of HadOCC, we can use them to predict both simulators' values of mean annual `iz.chl` at new input points. However, it isn't obvious how they can be combined to provide any insight into the relationship between the two simulators.

It is clear from the plots in Figure 3.3 that the distribution of output is different for the two simulators. One might deduce from the fact that a non-isotropic correlation function is required to capture HadOCC's behaviour that it is somehow more complicated. However, far fewer inputs are strongly active in HadOCC than in OG99NPZD. Ideally, one might like to know if the same aspects of the system

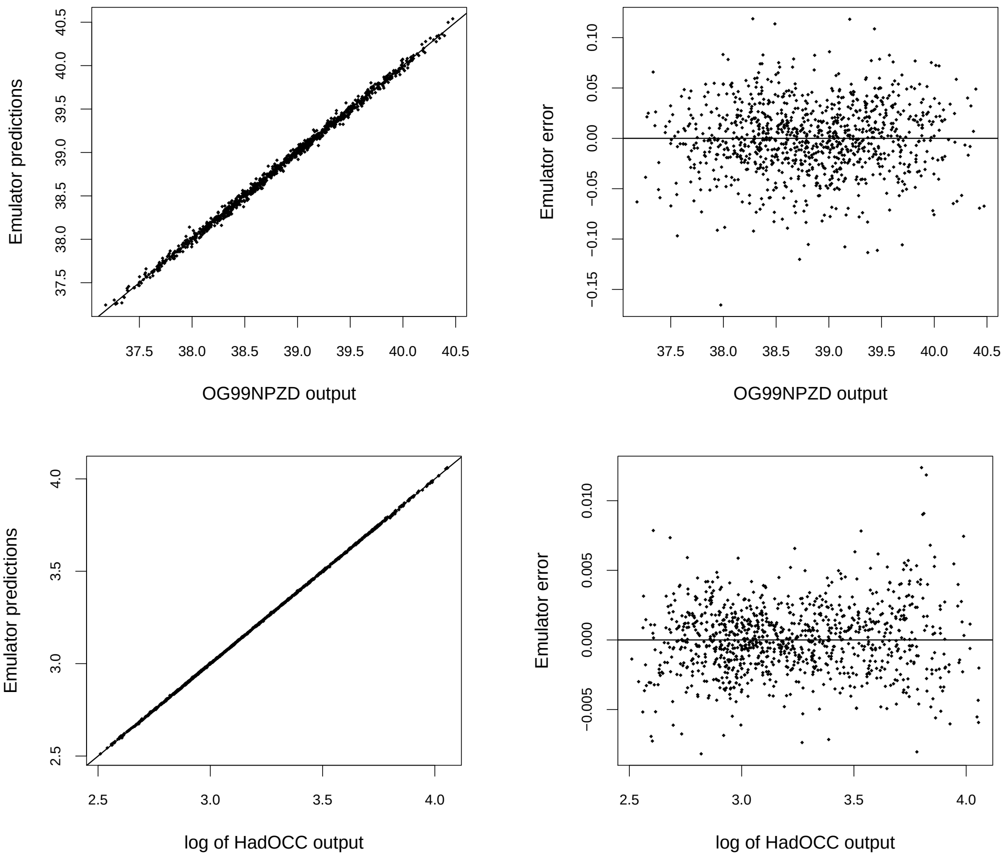


Figure 3.10: *Plots of predictions and errors for the OG99NPZD emulator (top) and non-isotropic HadOCC emulator (bottom). The emulators built from OG1 and HAD1 were used to predict the outputs for OG2 and HAD2.*

are crucial in the two simulators. As they are modelling the same system and quantities, and Tables 2.1 and 2.3 show that they have some inputs with very similar meanings, it seems intuitive that the same sorts of quantities should be important. However, the main effects plots in Figures 3.4 and 3.5 show this not to be the case. The two most important inputs to HadOCC, `rcchl` and `rcnphy`, don't have equivalents in OG99NPZD. In OG99NPZD the most active inputs were a and c (both similar to HadOCC's input texttttphotmax), K_1 (similar to HadOCC's `kdin`) and μ_P and μ_{PP}, which are used to model phytoplankton mortality in a different way from the linked HadOCC inputs `pmortdd, fpmortdin` and `pminmort`. None of these semi-corresponding HadOCC input variables appear to have any significant effect on HadOCC output, from either their main effects plots or their coefficients' posterior means and variances. Does this mean that the two simulators are behaving entirely differently? Certainly, it appears that finding a 'similar' point in HadOCC and OG99NPZD input space is not a simple task. These are some of the issues we attempt to address throughout the remainder of this thesis, and in particular our aim is to develop methods for emulating multiple simulators. In Chapters 5 and 6 we present methods for doing this in two particular circumstances.

3.7 Summary

This chapter has introduced emulation, a method for creating a statistical approximation to a simulator. Some of the choices one faces when building an emulator have been explored, such as methods for selecting the terms in the regression surface, and possible structures for the correlated error.

We have also considered some of the weaknesses of emulation, and how these might manifest themselves. Some diagnostic tools have been presented that help pinpoint poor modelling choices in particular emulators.

These techniques have been used to emulate both OG99NPZD and HadOCC, and it has been observed that, while these emulators are each useful for predicting the behaviour of their respective simulator, they cannot easily be combined to help understand the similarities and differences in how the two simulators model the ocean carbon cycle.

Having laid this foundation, we now consider situations involving more than one simulator. However, before beginning to pursue methods for emulating multiple simulators, we will think in more detail about issues surrounding the topic.

Chapter 4

Multiple simulators

So far, we have studied the emulation of computer simulators with a view to building an emulator of one simulator. However, an emulator does not in itself tell us anything about the system, but only the simulator for which it was built. If the simulator represents the system poorly then so will the emulator.

It is often the case, when a system is of interest, that it will be modelled by more than one simulator. Sometimes these will be very similar in their structure, and will give very similar predictions for the system. Often however, they will represent the system in significantly different ways, and will not always behave similarly. We have seen this phenomenon at work through OG99NPZD and HadOCC, two models of the ocean carbon cycle, which were introduced in Chapter 2, and then used to build emulators in Section 3.6.

Comparing the behaviours of the different simulators may help scientists to better understand the system. It may reveal which aspects of the simulators are crucial and which are barely contributing, or could be handled in a much simpler way. Where one simulator is much more computationally expensive than another, it may help discern whether the extra effort is worth spending. Combining the predictions of several simulators may also give more accurate forecasts of the system.

Because these complex simulators are often high-dimensional and slow to run, and therefore somewhat unwieldy, emulation is an attractive method for dealing with them. Looking at multiple simulators introduces new issues for emulation, especially concerning the input spaces of the simulators, and it is vital to consider these before thinking about methods for emulation. As we saw in the example in Section 3.6, it is not immediately obvious how an emulator can be used to understand the relationships between two or more simulators.

This chapter contains a summary of existing work concerning multiple simulators, and details ways in which two simulators of a system can be different, laying a foundation for Chapters 5 and 6 where specific emulation methods will be introduced.

4.1 Multiple simulators in the literature

Joint emulation of two complex physical simulators has already been studied under particular special circumstances. A strategy is presented by Cumming and Goldstein (2010) for dealing with two simulators, one of which, $s^c\left(\cdot\right)$ is a fast approximation to the other, $s\left(\cdot\right)$. The models share the same input and output variables, but in the faster simulator in their example, the vertical gridding is ten times coarser. This makes $s^c\left(\cdot\right)$ run much faster, and it therefore gives many more approximate simulator outputs than the full simulator could give exact ones in the same time. Cumming et. al. assume that there will be strong links between the behaviour of the two simulators, and that they can therefore use evaluations of $s^c\left(\cdot\right)$ to build an emulator from which, given a specified belief structure, they can learn about $s\left(\cdot\right)$, aided by a sparse collection of runs from $s\left(\cdot\right)$. Because both simulators share the same inputs, their emulators' regression functions and correlation structures can be made to match, and so the two emulators can be linked probabilistically.

Although this technique does involve the emulation of multiple models, its goal is to accurately emulate the slower model $s\left(\cdot\right)$, rather than to study the relationship between the two. It is also limited, from our point of view, in forcing both models to share the same input variables. With different simulators of a particular system this is rarely the case.

Hung et al. (2009) focus on simulators with q branching factors $z_1, \ldots, z_q$, each associated with m_u nested factors $\mathbf{v}^{z_u} = \left(v_1^{z_u}, \ldots, v_{m_u}^{z_u}\right)$. Branching factors are usually qualitative and switch-like, and introduce different 'nested' parameters depending on the level at which they are set. Therefore at different levels of the branching factor, the simulator could be thought of as several different, albeit very similar, simulators. This arises often in engineering contexts, where the use of a different tool or technique will introduce different parameters, while a large group of 'common' parameters $\mathbf{x} = (x_1, \ldots, x_t)$ remains the same. Design of experiments becomes more complicated in this setting, and Hung et. al. present several branching Latin hypercube design criteria, and a maximin design strategy. Because comparison of the nested factors for different branching factors is meaningless, the distances between points in terms of each collection $\mathbf{v}^{z_u}$ of nested factors are treated separately from those across the common parameters $\mathbf{x}$.

Having designed their experiments, Hung et. al. go on to present methods for emulation (under the synonym 'kriging') with branching and nested factors. They model the simulator's output using a constant mean term (a regression surface of order one) and a Gaussian process residual $Z(\mathbf{w})$, where $\mathbf{w} = (\mathbf{x}, \mathbf{z}, \mathbf{v})$ is a simulator input point. All the work of interpolation and prediction is done therefore by the stochastic Gaussian process.

In order to enable the same emulator to handle the different nested factors introduced by the branching factors, Hung et. al. split the exponent of their correlation function into two parts,

$$\begin{aligned} \operatorname{cor}\left[Z\left(\mathbf{w}_1\right), Z\left(\mathbf{w}_2\right)\right] = \exp\Bigg\{ & -\sum_{i=1}^{t} \alpha_i \left(x_{1i} - x_{2i}\right)^2 \\ & -\sum_{u=1}^{q} \left[\sum_{i=1}^{m_u} \sum_{j=1}^{k_u} \gamma_{uij} \left(v_{1i}^{z_{1u}} - v_{2i}^{z_{2u}}\right)^2 I_{[z_{1u}=z_{2u}=z_{u,j}]} + \theta_u I_{[z_{1u} \neq z_{2u}]}\right]\Bigg\}, \end{aligned}$$

where q is the number of branching factors z_u, m_u the number of nested variables associated with the branching factor z_u, and k_u the number of levels branching factor z_u can take. The first term models correlation for the common inputs, and works in the same way as the separable Gaussian process described in Section 3.3.3, with correlation lengths α_i. The second handles the branching and nested factors, and contains two terms with indicator functions. If the branching factors of two points are at the same level, then a correlation function over the nested factors is applied, working similarly to the correlation function for the common variables.

This introduces correlation lengths γ_{uij}, for each nested variable within each level of each branching factor. On the other hand, if the branching factors for two points are set to different levels, a relatively large constant term θ_u is added to reduce the correlation at the two points.

While this enables Hung et. al. to emulate these different cases of a simulator, in terms of comparing simulators it has a few drawbacks. The separation of common and nested variables in the correlation function means that changes in overall simulator behaviour involving more than just the nested inputs at different levels of the branching factors cannot express themselves; the common inputs have the same correlation lengths α_i regardless of the settings of the branching factors. In order to compare the behaviour of the simulator at different levels of the branching factors therefore, one must study the correlation function's parameters θ_u and γ_{uij}. However, as discussed in Section 3.3.3, accurate estimation of these numbers is far from trivial, and so comparisons may be quite tenuous. A very similar scenario involving alternative versions of the same simulator is described in Section 4.2.1, and a method for emulation in this case is presented in Chapter 5. Again, this method is limited by the demand that the simulators have almost the same input space. In Chapter 6 an emulation strategy for multiple simulators is presented that does not make this demand.

In recent years, the existence of many different simulators modelling the earth's climate has lead people to consider how to combine the information they hold. It is mostly agreed that a multi-model ensemble will perform better than any of the single simulators it contains (Buser et al., 2009). Each simulator will have its own strengths and weaknesses, and may closely fit some aspects of the system but not others, and so in any situation it is best to consider as many [serious] simulators as possible.

Raftery et al. (2005) use Bayesian Model Averaging to combine the forecasts from an ensemble of simulator predictions $s_1, \ldots, s_k$ in order to predict the true value y using the law of total probability,

$$p\left(y \mid s_1, \ldots, s_k\right) = \sum_{i=1}^{k} w_i p_i\left(y \mid s_i\right).$$

The w_i are the probabilities that each simulator is the best in the ensemble, and therefore sum to one. The function $p_i\left(y \mid s_i\right)$ should be interpreted as the probability of y conditional on s_i, given that s_i is the best forecast in the ensemble. These distributions are assumed to be Normal, with common variance, and this model is used to estimate the w_i by maximum likelihood. The result is a PDF for the true system value y, conditional on the simulator ensemble, which is a weighted sum of normal distributions.

In a similar vein, Smith et al. (2009) use weighted averages of forecasts from an ensemble, but their approach differs in that rather than attach to each simulator a probability w_i of being the best in the ensemble, they weight each simulator by its precision τ_i. Labelling the current observed system value (they use the example of mean temperature) as X_0, and the simulators' outputs for the current and future system states as $(X_i,\ Y_i)$ respectively, for $i = 1, \ldots, m$, they assume

$$\begin{aligned} X_0 &\sim N\left(\mu,\ \lambda_0^{-1}\right)\ (\lambda_0 \text{ known}) \\ X_i &\sim N\left(\mu,\ \lambda_i^{-1}\right) \\ Y_i \mid X_i &\sim N\left(\nu + \beta\left(X_i - \mu\right), \left(\theta\lambda_j\right)^{-1}\right), \end{aligned}$$

attaching uniform prior distributions to μ, ν and β, and Γ priors to θ, $\lambda_1, \ldots, \lambda_m$ and to the λ_i hyperparameters. Using monte carlo they generate random samples from the distributions of these parameters, and eventually the quantities of interest, the future predictions Y_i.

Leith and Chandler (2010) acknowledge that in order to predict a more detailed feature of a system, for example a time series of temperatures, one must include more detail of the simulators' underlying structures. They begin by assuming that all climate simulators will model more or less the same processes, and noting that rather than aiming to give time-series predictions exactly matching the true climate, they intend to match the system statistically. The outputs are therefore likely all to follow a similar pattern, which can be summarised by the same parameters. They then use the distribution of these parameters, arising from the notional population of all climate simulators, to summarise uncertainty in the simulators' outputs.

Rather than treat the simulators as functions over their input space, as in emulation, Raftery et. al and Smith et. al. use only a single prediction from each simulator, along with some sort of judgement of each simulator's validity or reliability. Wilks (2006) gives a summary of methods for predicting climate in this way from a multi-model ensemble. Although they give a more detailed treatment of the outputs, Leith et. al. ignore the input variables, and assume that each simulator models the same processes in a similar way. In many applications this is simply not the case.

While these ensemble methods may produce a more accurate prediction than using just one simulator, they cannot easily compare the behaviours of the simulators across their domains, or contrast the treatments of a particular system feature in each simulator. It is hoped that emulation can help with these tasks, and enabling this is the focus of Chapters 5 and 6. However, before devising methods for emulating multiple simulators, it is important to consider some of the ways in which two or more simulators can differ. Because we intend to emulate the two simulators, we focus on differences in the input spaces, and give examples from HadOCC and OG99NPZD, the two simulators introduced in Chapter 2.

4.2 Breaking down simulator differences

Given two simulators, $s_1(x_1)$ and $s_2(x_2)$, of a particular system, where $x_1,\ x_2$ are their respective collections of input variables, it is likely that we will be able to link them in some way by their input space and their treatment of various processes. They may have almost entirely the same components and input spaces, as do those studied by Cumming and Goldstein (2010) or Hung et al. (2009). Equally, they may model the system, or aspects of it, in very different ways, making such links more complicated.

Approaching the problem in this way requires understanding of the simulators, gained through expert advice and access to the code and surrounding documentation. In this project we were fortunate enough to have these, and so are able to give examples where possible of how differences within and between HadOCC and OG99NPZD fit into this framework.

4.2.1 Simple extensions

A simulator will often have the option of including more parameters, usually either by modelling a particular process in a more complicated way, or by including an entirely new process. If the initial simulator is $s_0(x)$, the extended version may be written $s_1(x, v, w)$, where $x,\ v,\ w$ are each collections of input variables. The v are those inputs whose values determine the relationship between $s_0(\cdot)$ and $s_1(\cdot)$. This distinction will be made clearer in Chapter 5.

This situation naturally splits into two cases:

1. There exists a value v^*, such that $s_1(x, v^*, w) = s_0(x)$ for all valid $x,\ w$,

2. There is no such v^*; $s_1(x, v, w)$ cannot be constrained through its extra inputs to be the same as $s_0(x)$.

It may be useful to study the behaviour of the new parameters $v,\ w$. They could be active only in the new module, or they may be used throughout the code. Either way, the effects of the extension and the new parameters $v,\ w$ will propagate through the simulator's state. In extreme cases, we may want to consider whether this changes the meaning of the other inputs x, and whether equating the same value of x in both versions of the simulator is valid.

The value v^* exists

This scenario is approached, albeit with a different application in mind, in Goldstein and Rougier (2009) (see especially Sections 4.3 and 6.2). Goldstein and Rougier are interested in making improvements to a simulator, and eventually in the concept of the 'Reified' simulator. They form a relationship between the two simulators $s_0(x)$ and $s_1(x, v, w)$ which we use to develop a method to jointly emulate the two simulators in Chapter 5.

In HadOCC, one can choose whether to make the Carbon:Chlorophyll (C:Chl) ratio constant (by setting `rcchlopt = 0`), or varying (by setting `rcchlopt = 1`). In the case where C:Chl varies, two new parameters `rcchlmin` and `rcchlmax` are introduced, setting the minimum and maximum of the ratio. Thus

$$v = \{\text{rcchlmin, rcchlmax}\}.$$

In either case, there is a parameter `rcchl`, setting the initial value of C:Chl (this remains the same if C:Chl is constant), which belongs to x. If C:Chl is varying and we set `rcchlmin = rcchl = rcchlmax`, then HadOCC behaves exactly as if we set C:Chl constant at the same rate. So, with some reparameterisation, we can have a v^* such that $s_0(x) = s_1(x, v^*, w)$. This forms the basis of the example in Chapter 5.

There is no v^*

When asked about C:Chl and the related parameters, John Hemmings, our expert from the National Oceanography Centre, Southampton, suggested that we could keep `rcchlmax` fixed at a value much higher than `rcchl` or `rcchlmin` can attain, so that the extra parameter space contains only the input `rcchlmin`. It is then impossible to achieve

$$\texttt{rcchlmin} = \texttt{rcchl} = \texttt{rcchlmax},$$

and therefore to make the two versions of HadOCC equal one another.

More than one extension

It could be that a simulator $s_0(x)$ can be extended in different ways, to $s_1(x, v_1)$ and $s_2(x, v_2)$. Within this there are two possibilities:

1. It may be that the two extensions could be added simultaneously, in which case $s_{12}(x, v_1, v_2)$ is also possible,

2. It may be that they are mutually exclusive, for example two different ways of adding in the same process, in which case adding both together is impossible, and there is no $s_{12}(\cdot)$.

In terms of emulation, this distinction is probably unimportant, since s_{12}, if it exists, is just another type of the same problem. Where it makes more of a difference is in comparison, since in the second case (if the two extensions are modelling the same process) we may want to compare them directly with one another in a different way from two extensions modelling different processes. It is also likely that v_1 and v_2 will contain similar parameters in the second case, but not in the first.

In this scenario where a simulator $s_0(x)$ is extended somehow, a possible strategy for emulation is to emulate the simple, unextended simulator $s_0(x)$, and then somehow extend the emulator. This is the approach taken in Chapter 5.

4.2.2 Different parameterisations of a process

This situation is similar to that above, but crucially the 'simpler' model $s_0(x)$ cannot be run on its own. There may be several ways for this to occur, a common one being where a particular process must be included in the simulator, but where the user can choose how it is modelled. Here we have $s_1(x, v_1)$ and $s_2(x, v_2)$, but not $s_{12}(x, v_1, v_2)$ or $s_0(x)$.

When the simpler model $s_0(x)$ existed and could be evaluated, a possible strategy was to emulate $s_0(x)$ and then extend the emulator to emulate $s_1(x, v_1)$ or $s_2(x, v_2)$. It is possible that, if the parameters v_1 and v_2 can be fixed in such a way that the two models become the same, we may be able to conceive of some sort of 'common' emulator, analogous to the emulator of $s_0(x)$ in Section 4.2.1.

This, as well as the previous section on extensions, involves branching and nested variables as described by Hung et al. (2009). Their approach is to treat the common inputs the same regardless of which extension applies, by estimating the mean and correlation lengths and then adapting the correlation function depending on the extension. The methods presented in Chapter 6 will offer an alternative means of joint emulation here, in a way that facilitates comparison and understanding of different simulators.

Photosynthesis in HadOCC

In HadOCC we must choose a submodel for photosynthesis, two of which are OG99PHOT (Oschlies and Garçon, 1999) and A93D (Anderson, 1993). In the MarMOT interface (introduced in Section 2.4), the groups of parameters which we are here labelling v_1, v_2 are given the same names, meanings and units for either submodel, however since they enter HadOCC through completely different equations, it seems more appropriate to treat them as different sets of input parameters. In Hung et al. (2009) terminology, the choice of photosynthesis submodel is the branching variable and v_1 and v_2 the corresponding sets of nested variables.

In OG99PHOT, the photosynthesis-irradiance curve is determined by

$$J = \frac{P_{\max}\alpha_d PAR_d}{\sqrt{P_{\max}^2 + (\alpha_d PAR_d)^2}},$$

whereas in A93D it is

$$J = P_{\max}\left[1 - \exp\left(-\frac{\alpha_d PAR_d}{P_{\max}}\right)\right].$$

Here

$$\begin{array}{rcl} P_{\max} & = & \texttt{photmax} \\ \alpha_d & = & \texttt{alphachl} \\ PAR_d & \text{involves} & k_{\mathrm{dPAR}}, \text{ which uses } \texttt{rparsol}. \end{array}$$

The HadOCC code and the source code for the two photosynthesis submodels show that the parameters do not split easily into photosynthesis or non-photosynthesis inputs. Many of the input variables appear in both the photosynthesis submodels and other processes in the code. This makes the idea of treating the two versions of HadOCC as essentially the same simulator, but with a different submodel affecting only a subset of the parameters, somewhat spurious.

4.2.3 Different simulators with similar processes

In many cases, two simulators of a system will have entirely different input spaces, and processes will be parameterised in different ways. However, it is not hard to imagine that we may well be able to match up processes between the simulators. In ocean ecosystem models, processes such as mortality, grazing and growth will be incorporated into most models, even if they are handled differently. For example, HadOCC and OG99NPZD both deal with the grazing of phytoplankton by zooplankton, but with different parameters and to different levels of detail.

Depending on how neatly the simulators split into comparable sub-processes, we may be able to make comparisons between simulators at process level. This is similar to Section 4.2.2, in that we are still concerned with how processes match up, except that here we have no concept of a core model $s_0(x)$, or a core group of inputs x. Processes in different simulators can appear to be similar to varying extents.

Zooplankton and Phytoplankton mortality

Zooplankton mortality is an example of a process which may appear strongly linked for OG99NPZD and HadOCC. In HadOCC, zooplankton mortality causes a loss of nitrogen from the zooplankton compartment and gains to detritus and nutrient (determined by the proportion `fzmortdin` contributing to detritus). There is a constant mortality rate `zmort` and a concentration dependent rate `zmortdd` (because there should be more predators where there is more prey) such that loss from zooplankton owing to mortality is

$$\texttt{zmort}Z + \texttt{zmortdd}Z^2,$$

where Z is the concentration of the tracer zooplankton. Of this,

$$\texttt{fzmortdin}\left(\texttt{zmort}Z + \texttt{zmortdd}Z^2\right)$$

contributes to detritus and

$$(1 - \texttt{fzmortdin})\left(\texttt{zmort}Z + \texttt{zmortdd}Z^2\right)$$

to nutrient.

In OG99NPZD, zooplankton mortality is entirely concentration dependent, modelled by

$$\mu_{ZZ}Z^2,$$

and all resulting nitrogen contributes to detritus. This means that if the HadOCC parameters `fzmortdin` and `zmort` are set to be 1 and 0 respectively, the simulators handle the process in the same way.

However, situations are rarely this simple. OG99NPZD contains a parameter γ_2, modelling excretion, so that $\gamma_2 Z$ is lost from zooplankton and gained by nutrient, whereas HadOCC doesn't explicitly model excretion. If we treat γ_2 (in OG99NPZD) as functionally equivalent to `zmort` (in HadOCC), which Hemmings (2009) suggests, then there is no way to make the two models the same.

This highlights the issue that although input parameters may have the same name, units and descriptions in two different simulators, they shouldn't necessarily be identified, even if the system meaning is clear. For example asserting here that μ_{ZZ} (from OG99NPZD) and `zmortdd` (from HadOCC) should be the same (they have the same name, description and units in MarMOT) makes little sense, because of the overall modelling of the process and their interaction with the other parameters.

It is unclear whether we can capitalise on being able to link a process 'exactly' across two different simulators, when the behaviour of the rest of the simulators means that the outputs will never match up. It is also unclear how we are to think of the link between parameters which are given the same meaning and units in two different simulators, especially when there are different groups of related parameters.

Another processes with complex input links is phytoplankton mortality. Phytoplankton mortality in HadOCC is controlled by the parameters `pmortdd` and `pminmort`. Loss of nitrogen from phytoplankton owing to mortality is `pmortdd`P^2 (where P is the concentration of phytoplankton) unless $P <$ `pminmort`, in which case there is no loss.

In OG99NPZD, nitrogen loss from phytoplankton through mortality is $\mu_P P + \mu_{PP} P^2$. Of this, $\mu_P P$ goes to nutrient, and $\mu_{PP} P^2$ to detritus. In HadOCC, a constant proportion `fpmortdin` of the nitrogen from dead phytoplankton goes to detritus, and the remainder to nutrient. So regardless of the values of the inputs, HadOCC and OG99NPZD model the same processes but in different ways.

Intermediate variable emulation, the method presented in Chapter 6, uses emulation and understanding of the simulators to enable comparisons in this setting.

4.2.4 Different processes

There will be situations where the processes in two simulators of a particular system can't be linked. It could be that there is no clear way to split the system up, or that it is very poorly understood. The simulators could model the system in terms of entirely different properties. However, even in pairs of simulators for which there are many matching processes, it is likely there will still be some events that are represented in one simulator but not in the other. For example, in OG99NPZD the zooplankton feeds exclusively on phytoplankton, whereas in HadOCC it also eats detritus. Not only does this create a series of nitrogen transfers involving detritus in HadOCC that are not mirrored in OG99NPZD, it may also affect comparisons between HadOCC and OG99NPZD involving zooplankton feeding on phytoplankton. It is quite feasible that to produce plausible output, the zooplankton consumes more phytoplankton in OG99NPZD than in HadOCC, in order to compensate for not grazing on detritus.

It may be that if we are able to devise a method for emulating different simulators where the processes match up, as in Section 4.2.3, we can extend it to be able to deal with slight differences. However, jointly emulating two simulators that do not match up even at process level could be very difficult. At the most cautious level, one could derive a joint emulator $f(x_1, x_2)$, where $x_1,\ x_2$ are the distinct sets of inputs for the two simulators. This could then emulate both simulators' outputs. However, in order to make this different from two separate emulators, one of each simulator, one would have to specify beliefs linking the behaviours of the two simulators, and this would require serious thought.

Figure 4.1 shows the processes modelled by HadOCC and OG99NPZD, and highlights any processes included by only one of the simulators.

4.3 Summary

In this chapter we have reviewed existing work concerning multiple simulators, and summarised some possible relationships between two simulators. We noted that at present, methods for emulation demand that the simulators be identical or very similar in terms of input space, and that this is a very restrictive constraint.

In the following chapter we introduce *hierarchical emulation*, a method for emulating simulators with simple extensions, as in Section 4.2.1. *Intermediate variable emulation*, which enables emulation of two simulators whose sub-processes can be at least partly matched, as in Section 4.2.3, and is therefore less restrictive, follows in Chapter 6.

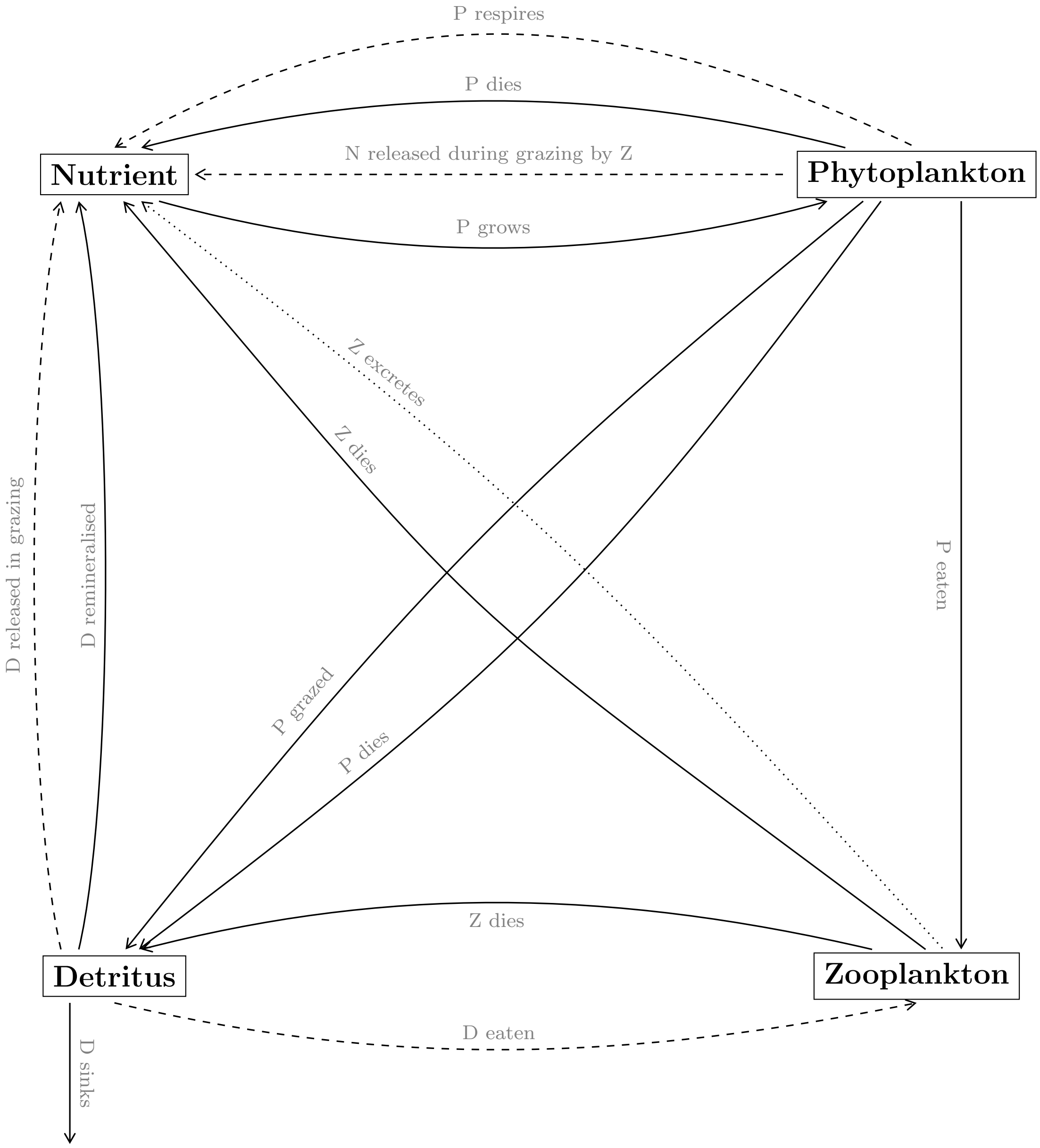


Figure 4.1: *Comparing processes represented by HadOCC and OG99NPZD. A dashed line represents a process that is only modelled by HadOCC, and a dotted line a process that is modelled only by OG99NPZD. A solid line shows a process that is included by both simulators.*

Chapter 5

Hierarchical emulation

The first type of simulator difference, described in Section 4.2.1, was to have two simulators for which one, $s_1(x, v, w)$ is an extension of the other, $s_0(x)$. This is a common occurrence, as there is often some choice over exactly which processes to include in a simulator, and how detailed to make some parameterisations. In this chapter we introduce *hierarchical emulation*, a method for emulation in this situation, and give an example using two versions of HadOCC.

Our concern is the scenario in which the two versions of the simulator can be made exactly the same by fixing the set of parameters v to a particular value v^*, so that

$$s_1(x, v^*, w) \equiv s_0(x) \tag{5.1}$$

for all valid values of x and w. The set of input variables w contains those input variables belonging only to the more complicated simulator $s_1(\cdot)$, but whose values do not affect the matching up of $s_1(\cdot)$ with $s_0(x)$ at $v = v^*$. We will refer to the set v as *hierarchical variables* and w as *extra variables.*

5.1 An emulation structure

The structure of a hierarchical emulator enables us to emulate $s_1(x, v, w)$ in a way that incorporates the information in Equation 5.1.

Firstly, assume that *transformation functions* $g_i(\cdot)$ exist such that $g_i(v_i^*) = 0$ for each hierarchical variable v_i. Superficially this is not always the case, for example in the HadOCC example involving the carbon:chlorophyll ratio in Section 5.5 where making the two simulators equal involves a relationship between three of the inputs, however, by re-parameterisation it can be achieved.

The simulators can then be linked to one another in terms of some new functions $\psi(\cdot)$. With one hierarchical variable v, the more complex simulator $s_1(x, v, w)$ can be re-written

$$s_1(x, v, w) = s_0(x) + g(v)\,\psi(x, v, w)\,.$$

At $v = v^*$, the two simulators are equal, and the relationship is preserved. This relationship is also used by Goldstein and Rougier (2009), where the more complex simulator, $s'(\cdot)$ in their notation, is a step toward making the simulator a better representation of the real system.

When there are k hierarchical variables $v_1, \ldots, v_k$, this is complicated slightly. Simply having an $s_0(x)$ term and a single hierarchical term

$$\left(\prod_{i=1}^{k} g_i(v_i)\right)\psi(x, v_1, \ldots, v_k)$$

will not work, because this will cause the two simulators to be equal when $v_i = v_i^*$ for any i, and this is not the case. Were we to use a function $g(v_1, \ldots, v_k)$, which was zero only at $(v_1^*, \ldots, v_k^*)$, having a single hierarchical term could work, but this would make choice of the function $g(\cdot)$ quite complicated. To limit equality to $v_i = v_i^*$ for all i therefore, we need to have at least

$$s_1(x, v, w) = s_0(x) + \sum_{i=1}^{k} g_i(v_i)\,\psi_i(x, v_i, w)\,.$$

However this still fails to express the relationship between the two simulators correctly. For example, in the case with two hierarchical variables, $v = (v_1, v_2)$, and no extra variables w, this would give

$$s_1(x, v_1, v_2) = s_0(x) + g_1(v_1)\,\psi_1(x, v_1) + g_2(v_2)\,\psi_2(x, v_2)$$

and therefore

$$\begin{aligned}
s_1(x, v_1, v_2^*) &= s_0(x) + g_1(v_1)\,\psi_1(x, v_1)\\
s_1(x, v_1^*, v_2) &= s_0(x) + g_2(v_2)\,\psi_2(x, v_2)\,.
\end{aligned}$$

But this forces

$$s_1\left(x, v_1, v_2\right) = s_1\left(x, v_1, v_2^*\right) + s_1\left(x, v_1^*, v_2\right) - s_0\left(x\right),$$

which is not necessarily the case. Often interactions between variables have an important effect on simulator output, and so to include these, the hierarchical emulator structure for k hierarchical variables must be

$$s_1\left(x, v, w\right) = s_0\left(x\right) + \sum_{i=1}^{k} g_i\left(v_i\right)\psi_i\left(x, v_i, w\right) + \sum_{\substack{i=1\\j>i}}^{k} g_{ij}\left(v_i, v_j\right)\psi_{ij}\left(x, v_i, v_j, w\right) + \\ \ldots + g_{1\ldots k}\left(v_1, \ldots, v_k\right)\psi_{1\ldots k}\left(x, v, w\right), \tag{5.2}$$

so that every possible interaction is allowed for. In some ways this is similar to functional ANOVA decomposition, where the main effects and interactions are modelled by separate functions (see, for example Kaufman and Sain (2010)). Now, each $g_\cdot\left(\cdot\right)$ function must satisfy

$$g_{[i]}\left(v_{[i]}\right) \neq 0 \iff v_j \neq v_j^* \; \forall j \in [i],$$

where $v_{[i]}$ is the set of hierarchical variables associated with the function $\psi_{[i]}\left(\cdot\right)$. The subscript $[i]$ here denotes an index that may involve several numbers, for example the $\psi_{ij}\left(\cdot\right)$ in Equation 5.2. A simple strategy, which we will follow from here, is to have

$$g_{[i]}\left(v_{[i]}\right) = \prod_{j\in[i]} g_j\left(v_j\right).$$

Note that the function $g_{[i]}\left(\cdot\right)$ uniquely determines the function $\psi_{[i]}\left(\cdot\right)$. We will consider the implications of this in Section 5.2.3. Although this model looks somewhat complicated, conversations with others in the field suggest that it is rare for there to be more than a couple of hierarchical variables attached to any additional process.

Having decomposed the simulator $s_1\left(x, v, w\right)$ using the relationship in Equation 5.1, a hierarchical emulator can be built using a collection of standard emulators constructed as in Chapter 3

5.2 Building a hierarchical emulator

The hierarchical emulation strategy is to emulate $s_1(x, v, w)$ by emulating $s_0(x)$ and each of the functions $\psi_{[i]}\left(x, v_{[i]}, w\right)$ separately, so that

$$f_0(x) = X_0\beta_0 + \epsilon_0(x)$$

is an emulator of $s_0(x)$, and

$$h_{[i]}\left(x, v_{[i]}, w\right) = H_{[i]}\beta_{[i]} + \epsilon_{[i]}\left(x, v_{[i]}, w\right) \tag{5.3}$$

is an emulator of $\psi_{[i]}\left(x, v_{[i]}, w\right)$.

The emulation framework in Chapter 3 enables us to build the emulators above, so that for m new input points $(\tilde{\mathbf{x}}, \tilde{\mathbf{v}}, \tilde{\mathbf{w}})$ the posterior distributions

$$\begin{gathered} s_0(\tilde{\mathbf{x}}) \mid s_0(\mathbf{x}) \\ \psi_{[i]}\left(\tilde{\mathbf{x}}, \tilde{\mathbf{v}}_{[i]}, \tilde{\mathbf{w}}\right) \mid \psi_{[i]}\left(\mathbf{x}, \mathbf{v}_{[i]}, \mathbf{w}\right) \end{gathered} \tag{5.4}$$

can readily be found, assuming data $\boldsymbol{\psi}_{[i]}$ from each function $\psi_{[i]}(\cdot)$ is available (the design criteria necessary for this will be addressed in Section 5.2.2). However, it is not obvious how these emulators can be combined to find

$$s_1(\tilde{\mathbf{x}}, \tilde{\mathbf{v}}, \tilde{\mathbf{w}}) \mid s_1(\mathbf{x}, \mathbf{v}, \mathbf{w}),$$

which is the goal of hierarchical emulation.

In Equation 5.4, the data has been split up such that the output $\mathbf{s}_0$ alone is used to train the emulator for $s_0(\cdot)$, the vector $\boldsymbol{\psi}_1$ to train the emulator of $\psi_1(\cdot)$ and so on, and this has not yet been justified. Nor have we given a probabilistic framework by which we can connect these separate emulators to form an emulator of $s_1(\cdot)$, or explained how the data from $\psi_{[i]}\left(\tilde{\mathbf{x}}, \tilde{\mathbf{v}}_{[i]}, \tilde{\mathbf{w}}\right)$ is found. These issues are resolved in the following sections.

5.2.1 Prior structure and separability

To construct emulators for the separate functions $s_0(x)$ and the $\psi_{[i]}\left(x, v_{[i]}, w\right)$, one needs prior distributions

$$\pi_{[i]}\left(\beta_{[i]}, \sigma^2_{[i]}\right).$$

Some options for these are discussed in Section 3.1. However, to build a hierarchical emulator for $s_1(x, v, w)$, these emulators must be probabilistically linked, and so a joint prior,

$$\pi\left(\beta_0, \sigma_0^2, \ldots, \beta_k, \sigma_k^2\right),$$

is required.

Ideally, this prior distribution should allow the hierarchical emulator for $s_1(\cdot)$ to be constructed from the emulators in Equation 5.4, rather than forcing all data from $s_0(\cdot)$ and $\psi_i(\cdot)$ to be informative for each emulator.

O'Hagan (1998) investigates covariance structures for a random Gaussian process function $f(x, y)$ defined for $x \in \mathcal{X},\ y \in \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ may be finite or infinite domains. Of particular interest is the covariance structure necessary to ensure that, given observations $f(x, y')$, further observations $f(x', y')$, for $x' \neq x$, will provide no more information about $f(x, y)$, for $y \neq y'$. That is, (x, y) is *separated* from (x', y') by (x, y').

In the hierarchical emulation setting this is equivalent to saying that having observed $s_0(x) = s_1(x, v^*, w)$, no further observation $s_1(x, v, w)$ will be informative for $s_0(\tilde{x}) = s_1(\tilde{x}, v^*, w)$. This means that so long as data $s_0(x)$ is present, $s_1(x, v, w)$ will not inform the emulator for $s_0(\cdot)$ at new points.

O'Hagan asserts that the separation described above is true if the property $\mathcal{M}(\mathcal{X}; \mathcal{Y})$, that

$$c\left[(x, y), (x', y')\right] = \frac{c\left[(x, y), (x', y)\right] c\left[(x', y), (x', y')\right]}{c\left[(x', y), (x', y)\right]}$$

holds for all $x, x' \in \mathcal{X},\ y, y' \in \mathcal{Y}$, where $c\left[(x, y), (x', y')\right] = \operatorname{cov}\left[f(x, y), f(x', y')\right]$. The points (x, y) and (x', y') are *separated* by $(x',\ y)$[1]. O'Hagan proves that this property $\mathcal{M}(\mathcal{X}; \mathcal{Y})$ holds if and only if there exists a function $a(x, y)$ on $\mathcal{X} \times \mathcal{Y}$ such that the covariance structure of the random variables

$$\gamma(x, y) = \frac{f(x, y)}{a(x, y)}$$

[1] They are also separated by (x, y'), but the formulation above makes it easier to apply to the problem at hand.

has a Kronecker product form, i.e.

$$\text{cov}\left[\gamma(x,y),\gamma(x',y')\right]=c_x\left\{x,x'\right\}c_y\left\{y,y'\right\}.$$

Applying this to our problem, to justify using only the data from a particular function $s_0(\cdot)$ or $\psi_i(\cdot)$ for the corresponding emulator term, we require that $s_0(\tilde{x})$ be separated from $s_1(x,v,w)$ by $s_0(x)$, and that $\psi_{[i]}\left(\tilde{x},\tilde{v}_{[i]},\tilde{w}\right)$ be separated from $s_1(x,v,w)$ by $\psi_{[i]}\left(x,v_{[i]},w\right)$. We must therefore ensure that

$$\text{cov}\left[f_0(\tilde{x}),f_1(x,v,w)\right]=\frac{\text{cov}\left[f_0(\tilde{x}),f_0(x)\right]\text{cov}\left[f_0(x),f_1(x,v,w)\right]}{\text{var}\left[f_0(x)\right]},\tag{5.5}$$

where $f_0(\cdot)$ and $f_1(\cdot)$ are the emulators of $s_0(\cdot)$ and $s_1(\cdot)$.

This follows easily by thinking of $s_0(x)$ as $s_1(x,v^*,w)$, and therefore of $y=(v^*,w)$ and $y'=(v,w)$ in the notation of O'Hagan.

By a similar argument, we can ensure that $\psi_{[i]}\left(\tilde{x},\tilde{v}_{[i]},w\right)$ is separated from further evaluations $s_1(x,v,w)$ so long as $\psi_{[i]}\left(x,v_{[i]},w\right)$ is known by enforcing

$$\text{cov}\left[h_{[i]}\left(\tilde{x},\tilde{v}_{[i]},w\right),f_1(x,v,w)\right]=\frac{\text{cov}\left[h_{[i]}\left(\tilde{x},\tilde{v}_{[i]},w\right),h_{[i]}\left(x,v_{[i]},w\right)\right]\text{cov}\left[h_{[i]}\left(x,v_{[i]},w\right),f_1(x,v,w)\right]}{\text{var}\left[h_{[i]}\left(x,v_{[i]},w\right)\right]}.$$

Note that at the moment we do not have direct access to evaluations of $\psi_{[i]}\left(x,v_{[i]},w\right)$. We will turn to the design issues this raises in Section 5.2.2.

If we have only one hierarchical variable, v, then

$$\begin{aligned}\text{cov}\left[f_0(\tilde{x}),f_1(x,v,w)\right]&=\text{cov}\left[f_0(\tilde{x}),f_0(x)+g(v)\,h(x,v,w)\right]\\&=\text{cov}\left[f_0(\tilde{x}),f_0(x)\right]+g(v)\,\text{cov}\left[f_0(\tilde{x}),h(x,v,w)\right]\end{aligned}\tag{5.6}$$

and

$$\text{cov}\left[f_0(x),f_1(x,v,w)\right]=\text{var}\left[f_0(x)\right]+g(v)\,\text{cov}\left[f_0(x),h(x,v,w)\right].\tag{5.7}$$

Therefore achieving separability is equivalent to stipulating that

$$\text{cov}\left[f_0(\tilde{x}),h(x,v,w)\right]=\frac{\text{cov}\left[f_0(\tilde{x}),f_0(x)\right]\text{cov}\left[f_0(x),h(x,v,w)\right]}{\text{var}\left[f_0(x)\right]},\tag{5.8}$$

as seen by substituting Equations 5.6 and 5.7 into Equation 5.5. This is certainly true if

$$\beta_i,\sigma_i^2\perp\beta_j,\sigma_j^2,\ \text{for all } i\neq j,\tag{5.9}$$

because then

$$\operatorname{cov}\left[f_0\left(x\right), h(x,v,w)\right] = \operatorname{cov}\left[f_0\left(\tilde{x}\right), h(x,v,w)\right] = 0.$$

In the general case, with k hierarchical variables,

$$\operatorname{cov}\left[f(\tilde{x}), f_1\left(x,v,w\right)\right] = \operatorname{cov}\left[f_0\left(\tilde{x}\right), f_0\left(x\right)\right] + \sum_{i=1}^{2^k-1} g_{[i]}\left(v_{[i]}\right)\operatorname{cov}\left[f_0\left(\tilde{x}\right), h_{[i]}\left(x, v_{[i]}, w\right)\right]$$

and

$$\operatorname{cov}\left[f(x), f_1\left(x,v,w\right)\right] = \operatorname{var}\left[f_0\left(x\right)\right] + \sum_{i=1}^{2^k-1} g_{[i]}\left(v_{[i]}\right)\operatorname{cov}\left[f_0\left(x\right), h_{[i]}\left(x, v_{[i]}, w\right)\right],$$

and so Equation 5.5 becomes

$$\sum_{i=1}^{2^k-1} g_{[i]}\left(v_{[i]}\right)\operatorname{cov}\left[f_0\left(\tilde{x}\right), h_i\left(x, v_{[i]}, w\right)\right] = \frac{\operatorname{cov}\left[f_0\left(\tilde{x}\right), f_0\left(x\right)\right]\left(\sum_{i=1}^{2^k-1} g_{[i]}\left(v_{[i]}\right)\operatorname{cov}\left[f_0\left(x\right), h_i\left(x, v_{[i]}, w\right)\right]\right)}{\operatorname{var}\left[f_0\left(x\right)\right]}.$$

Again, if the prior specifications in Equation 5.9 are used, separability holds.

So, we shall stipulate that $\beta_i,\ \sigma_i^2 \perp \beta_j,\ \sigma_j^2$, and therefore claim this separability property. This ensures that if we have observed $s_0\left(\mathbf{x}\right)$ we do not need to incorporate $s_1\left(\mathbf{x},\mathbf{v},\mathbf{w}\right)$ in order to predict $s_0\left(\tilde{\mathbf{x}}\right)$. Likewise, if $\psi_{[i]}\left(\mathbf{x},\mathbf{v},\mathbf{w}\right)$ is known then further simulator values $s_1\left(\mathbf{x},\mathbf{v},\mathbf{w}\right)$ are not informative for $\psi_{[i]}\left(\tilde{\mathbf{x}},\tilde{\mathbf{v}},\tilde{\mathbf{w}}\right)$.

A consequence of this prior structure, where $\left(\beta_i,\sigma_i^2\right) \perp \left(\beta_j,\sigma_j^2\right)$, is that the variance of $s_1\left(\mathbf{x},\mathbf{v},\mathbf{w}\right)$ necessarily increases as the v move away from v^*. Studying the behaviour of the emulators as this happens will show whether this presents a problem in practice. Other prior structures may also lead to this same separability property, and finding some would be a useful avenue for further research.

It has already been observed that data from the $\psi_{[\mathrm{i}]}\left(\cdot\right)$ are not readily available. This problem and the separability criterion impose restrictions on the design of training data, and these are discussed in the following section.

5.2.2 Training data design

The design structure required for hierarchical emulation as it has been set up here is tied to two features of the model. One relates to the separability issues discussed above, the other to ensuring we are able to procure training data for each function we aim to emulate.

In order to emulate a function $s(x)$, it is necessary to have training data. Because $s_0(x)$ is a simulator, it can be evaluated, and training data gathered so that it can be emulated. But this is not the case with the functions $\psi_{[i]}\left(x, v_{[i]}, w\right)$. These are defined only in relation to one another and to the simulators $s_0(x)$ and $s_1(x, v, w)$, and cannot be 'run'. To gather training data from them therefore, we must be intentional in our designs for $s_0(x)$ and $s_1(x, v, w)$.

All the data used to build these emulators must be collected through the simulators $s_0(x)$ and $s_1(x, v, w)$, including the data from the functions $\psi_{[i]}\left(x, v_{[i]}, w\right)$. The following explanation of how that is done is given for the case where there are two hierarchical variables $v = (v_1, v_2)$, and so the data structure in terms of functions $g_{[i]}(\cdot)$ and $\psi_{[i]}(\cdot)$ is

$$\begin{aligned} s_1(x, v, w) = {} & s_0(x) + g_1(v_1)\,\psi_1(x, v_1, w) + g_2(v_2)\,\psi_2(x, v_2, w) \\ & + g_{12}(v_1, v_2)\,\psi_{12}(x, v, w)\,. \end{aligned} \tag{5.10}$$

It is not difficult to see how it would extend to any number of hierarchical variables.

We begin with the assumption that we have run $s_1(\cdot)$ for a set of n points $\{\mathbf{x}, \mathbf{v}, \mathbf{w}\}$, giving the n-vector of output $\mathbf{s}_1$. We can easily find the corresponding vector $\mathbf{s}_0$ of outputs from $s_0(\cdot)$ by computing either $s_0(\mathbf{x})$ or $s_1(\mathbf{x}, \mathbf{v}^*, \mathbf{w})$ for any valid $\mathbf{w}$. However these data will not enable us to calculate the corresponding vectors $\boldsymbol{\psi}_{[i]}$ of data from each of the $\psi_{[i]}(\cdot)$. For this, we must isolate each $\psi_{[i]}(\cdot)$ in turn, by setting some of the hierarchical variables to v^*. For example, in the $k = 2$ setting, when $v_1 = v_1^*$ (and therefore $g_1(v_1) = 0$) and $v_2 \neq v_2^*$ (and so $g_2(v) \neq 0$), we have

$$s_1(x, v, w) = s_0(x) + g_2(v_2)\,\psi_2(x, v_2, w)\,,$$

and so

$$\psi_2(x, v_2, w) = \frac{s_1(x, v, w) - s_0(x)}{g_2(v_2)}. \tag{5.11}$$

This enables us to find the vector $\boldsymbol{\psi}_2$, and similarly, setting $v_2 = v_2^*,\ v_1 \neq v_1^*$ enables us to find $\boldsymbol{\psi}_1$. Finally, computing the vector $\boldsymbol{\psi}_{12}$ requires all the data collected so far. In terms of vectors of data,

$$\mathbf{s}_1 = \mathbf{s}_0 + \mathrm{diag}(g_1(\mathbf{v}_1))\boldsymbol{\psi}_1 + \mathrm{diag}(g_2(\mathbf{v}_2))\boldsymbol{\psi}_2 + \mathrm{diag}(g_{12}(\mathbf{v}_1, \mathbf{v}_2))\boldsymbol{\psi}_{12}$$

and therefore

$$\boldsymbol{\psi}_{12} = [\mathbf{s}_1 - \mathbf{s}_0 - \mathrm{diag}(g_1(\mathbf{v}_1))\boldsymbol{\psi}_1 - \mathrm{diag}(g_2(\mathbf{v}_2))\boldsymbol{\psi}_2] \left[\mathrm{diag}(g_1(\mathbf{v}_1))\,\mathrm{diag}(g_2(\mathbf{v}_2))\right]^{-1}.$$

The matrix $\mathrm{diag}(g_i(\mathbf{v}_i))$ is the diagonal matrix with the vector $g_i(\mathbf{v}_i)$ as its diagonal, and so the matrix $\left[\mathrm{diag}(g_1(\mathbf{v}_1))\,\mathrm{diag}(g_2(\mathbf{v}_2))\right]^{-1}$ is simply the diagonal matrix with terms analogous to the denominator in Equation 5.11.

With k hierarchical variables, in order to find the vectors of data necessary to emulate each term the simulator $s_1(x, v, w)$ must be run 2^k times for each input point at which $v_i \neq v_i^* \; \forall i$. To use such a point (x, v, w) in a hierarchical emulator we must know $s_1(x, v, w)$, $s_0(x)$ and $s_1\left(x, v_{[i]}, v^*_{[-i]}, w\right)$ for all possible subsets $[i]$ of the indices $\{1, \ldots, k\}$. The subscript $[-i]$ here denotes the set of indices left out by the set $[i]$. That is, to include a point with any hierarchical inputs not at their v^* values, the same point must be included with every possible subset of the hierarchical inputs set to v^*.

Having completed this process we have 2^k corresponding vectors of data, $\mathbf{s}_0$, $\boldsymbol{\psi}_1$, $\boldsymbol{\psi}_2, \ldots,$ $\boldsymbol{\psi}_{12}$ and so on, and so can build emulators for each function using the techniques in Chapter 3.

Recall from Section 5.2.1 that in order to justify not treating all evaluations of $s_1(x, v, w)$ as informative for all functions $s_0(x)$ and $\psi_{[i]}\left(x, v_{[i]}, w\right)$, new evaluations of each function must be separated from evaluations of all other functions by values from the same function. This means that, for example, if we know the value of $\psi_2(x, v_2, w)$, and wish to predict the value of $\psi_2(\tilde{x}, \tilde{v}_2, \tilde{w})$, no additional data $s_1(x, v, w)$ are informative. Therefore, for any observation $s_1(x, v, w)$ in the training data, we must also have observations $s_0(x)$ and $\psi_{[i]}\left(x, v_{[i]}, w\right)$ for all $[i]$.

The design structure described in the above argument ensures this. If we choose to include the point (x, v, w), we must also include $\left(x, v_{[i]}, v^*_{[-i]}, w\right)$, for all possible subsets of indices $[i]$, as only then have we access to the functions $\psi_{[i]}\left(x, v_{[i]}, w\right)$.

Conversely, data from $s_0(x)$ can be included on its own, without the corresponding evaluations of $s_1(x)$, and indeed any point can be included without including any corresponding points ‘higher up’ in the hierarchy, that is, with fewer of the hierarchical inputs set to their v^* values. In a situation where $s_0\left(x\right)$ is very cheap to run, and $s_1\left(x, v, w\right)$ is expensive, this could turn out to be advantageous.

5.2.3 Choosing appropriate transformation functions

The $g_{[i]}\left(v_{[i]}\right)$ functions used to transform the hierarchical variables determine the values and distributions of the $\boldsymbol{\psi}_{[i]}$ data. A poor choice could lead to a poor emulator, and especially to poor variance estimates for predictions.

The obvious approach when transforming the dependent variable for a regression type model is to use Box-Cox. The Box-Cox transformation of a dependent variable Y is

$$Y^{(\lambda)} = \begin{cases} \frac{Y^{\lambda}-1}{\lambda} & \lambda \neq 0 \\ \log(Y) & \lambda = 0 \end{cases}$$

where the optimal value of λ can be found using maximum likelihood. This value is optimal in that the transformed Y is a linear function of the independent variables, and the errors are approximately normal with constant variance.

However, in finding the suitable transformation functions $g_i\left(\cdot\right)$, we are not looking to transform the dependent variable but the hierarchical variables, from which the dependent variables $\psi_i\left(\cdot\right)$ will then be found. For example, in a system with one hierarchical input v, we have

$$\psi\left(x, v, w\right) = \frac{s_1(x, v, w) - s_0(x)}{g(v)} = \mathbf{H}_1\beta_1 + \epsilon_1.$$

We wish to find an appropriate transformation of v, rather than of the dependent variable $\psi\left(x, v, w\right)$, and therefore cannot use Box-Cox to find the $g_i\left(\cdot\right)$.

Before thinking of specific transformations, it is worth considering what properties the functions $g_{[i]}\left(\cdot\right)$ must have. Clearly in order to maintain the relationship between $s_0\left(\cdot\right)$ and $s_1\left(\cdot\right)$ we must have $g_{[i]}\left(v^*_{[i]}\right) = \mathbf{0}$ and $g_{[i]}\left(v_{[i]}\right) \neq \mathbf{0}$ for $v_{[i]} \neq v^*_{[i]}$, for all $[i]$. In order to be able to emulate the functions $\psi_{[i]}\left(\cdot\right)$ they must be continuous and finite, and so the $g_{[i]}\left(\cdot\right)$ must be continuous and non-zero for all $v_{[i]} \neq v^*_{[i]}$.

In keeping with the sorts of transformations allowed by Box-Cox we will consider functions of the type

$$g(v) = (v - v^*)^\lambda \tag{5.12}$$

and

$$g(v) = [\log(v - v^* + 1)]^\lambda, \tag{5.13}$$

for $\lambda > 0$, where these are functions of a single hierarchical input v. If, for some i, v_i^* is not on the boundary of the input space, or if $v_i \leq v_i^*$, these functions will need to be adapted.

As mentioned earlier, in order to simplify matters, when a term involves more than one hierarchical input we will use the product of the transformation functions,

$$g_{[i]}\left(v_{[i]}\right) = \prod_{j \in [i]} g_j(v_j),$$

so for example

$$g_{12}(v_1, v_2) = g_1(v_1)\, g_2(v_2).$$

In classical linear regression, the optimal situation is for the residuals to appear to be independent and normally distributed with mean zero. With this in mind, the transformations could be chosen such that when the regression surface used in the emulator is fitted to the resulting $\boldsymbol{\psi}$ vector, the residuals are as close to normal as possible. However, the correlated error term means that these errors aren't independent, and so another possibility is to find the transformation such that the residuals have minimal skew. We will return to this matter in the examples in Section 5.5.

5.2.4 The resulting emulator

Having set up the emulator, and designed training data (TD) ensuring separability between terms, we have the result that for a set of new input points $(\tilde{x}, \tilde{v}, \tilde{w})$

$$\begin{aligned} \mathrm{E}\left[s_1\left(\tilde{x}, \tilde{v}, \tilde{w}\right) \mid \mathrm{TD}\right] &= \mathrm{E}\left[s_0\left(\tilde{x}\right) \mid s_0\left(x\right)\right] \\ &\quad + \sum_{[i]} \mathrm{diag}\left(g_{[i]}\left(\tilde{v}_{[i]}\right)\right) \mathrm{E}\left[\psi_{[i]}\left(\tilde{x}, \tilde{v}_{[i]}, \tilde{w}\right) \mid \psi_{[i]}\left(x, v_{[i]}, w\right)\right] \end{aligned} \tag{5.14}$$

$$\begin{aligned} \mathrm{var}\left[s_1\left(\tilde{x}, \tilde{v}, \tilde{w}\right) \mid \mathrm{TD}\right] &= \mathrm{var}\left[s_0\left(\tilde{x}\right) \mid s_0\left(x\right)\right] \\ &\quad + \sum_{[i]} \mathrm{diag}\left(g_{[i]}\left(\tilde{v}_{[i]}\right)^2\right) \mathrm{var}\left[\psi_{[i]}\left(\tilde{x}, \tilde{v}_{[i]}, \tilde{w}\right) \mid \psi_{[i]}\left(x, v_{[i]}, w\right)\right] . \end{aligned} \tag{5.15}$$

The emulators for the separate terms can be used to find these values, and also enable us to sample from $s_1\left(\tilde{x}, \tilde{v}, \tilde{w}\right)|\mathrm{TD}$.

A serendipitous feature of the hierarchical emulator is that it includes within it an emulator of the difference between the simulators, since

$$s_1\left(x, v, w\right) - s_0\left(x\right) = \sum_{[i]} g_{[i]}\left(\tilde{v}_{[i]}\right) \psi_{[i]}\left(\tilde{x}, \tilde{v}_{[i]}, \tilde{w}\right) .$$

Having already built emulators of the functions $\psi_{[i]}\left(\cdot\right)$, we can easily find the mean and variance of the simulator difference for new input points, or sample from the posterior distribution. This is an attractive result, especially if the effect of the extension of $s_0\left(x\right)$ to $s_1\left(x, v, w\right)$, and the change in this effect across the input space, are of interest.

Building a hierarchical emulator requires many calculations and a lot of sorting and storing of data. To do this effectively requires a careful framework, and an object-oriented structure for hierarchical emulation in R (R Development Core Team, 2011) is presented in Chapter 7. This structure requires the user to provide training data, transformation functions and details of the hierarchical, extra and common inputs. It then checks that the data fit the design criteria for hierarchical emulation, and sorts the data according to the values of the hierarchical inputs, so that they are ready to be used to build an emulator.

In order to build an emulator, the user must specify the method for building the regression surface and the structure of the correlated error. These will then be applied to construct the separate terms of the hierarchical emulator. Finally, new input points can be given, and the hierarchical emulator will use each of the terms to predict the outputs of the simpler and extended simulators, and the difference between them. Because the structure is object-oriented, the information for each stage (data organisation, emulator building and prediction) is stored as separate objects. The structure of the objects and methods ensures that the process is rigidly organised, and no necessary information lost.

5.3 Comparing the hierarchical emulator with the 'standard'

In order to see whether hierarchical emulation is worth pursuing, we must compare it to the status quo. Questions we must therefore ask are:

1. What tasks are we asking the hierarchical emulator to perform?
2. What are the 'standard' emulators against which we will compare it?

These issues are explored below, before comparing different emulation strategies using two versions of HadOCC.

5.3.1 Tasks for comparison

Predicting $s_0(x)$ output

When the hierarchical emulator is used to emulate $s_0(x)$, all terms apart from the first are 'switched off', and we are left with an emulator of $s_0(x)$. Therefore the standard and hierarchical emulators of $s_0(x)$ should perform identically.

Predicting $s_1(x, v, w)$ output

Unlike any standard method, the hierarchical emulator for $s_1(x, v, w)$ is built from several terms, each of which is a separate emulator. Apart from the $s_0(x)$ term, each of these is multiplied by vectors $g(v)$, and there is potential for this to disrupt things. Because of the separability property established by the prior specification, the variance of the hierarchical emulator's prediction is the sum of the variances of the individual terms, and this has potential to become large as the hierarchical variables increase.

Predicting $s_1(x, v, w)$ output is useful in its own right, and in a situation where $s_0(x)$ is much cheaper to run than $s_1(x, v, w)$, building a hierarchical emulator by using many runs from $s_0(x)$ and fewer from $s_1(x, v, w)$ could be an attractive option.

Predicting some measure of the difference between s_0 and s_1

In comparing the two simulators, being able to reliably predict the difference between them in some way will be a tremendous help. It may also enable us to discern the circumstances in which the two simulators are very different, and when they behave similarly. The prediction variable will be the difference

$$s_1(x, v, w) - s_0(x)$$

or, if the logs of the outputs are used, the ratio

$$\frac{s_1(x, v, w)}{s_0(x)},$$

so long as both functions are positive. Which of these is predicted by the hierarchical emulator will depend on whether the simulator output or its logarithm is chosen.

Predicting s_1 output 'near' s_0

A key concern is how the emulators' predictions and variances depend on the values of the hierarchical inputs v. This may reveal features of the emulation models that are not appropriate. In particular, it may be interesting to compare predictions of the more complex simulator's behaviour when it is very near to the simpler simulator, that is when the v are very close to v^*.

5.3.2 Standard emulators

By 'standard' emulators, here we refer to those built using the methods in Chapter 3, where the emulator is the sum of one regression surface and one correlated error term. Therefore in these terms, a hierarchical emulator is a linear combination of standard emulators. In either setting there are choices of prior distribution, regression surface and correlation function, and while these are important they are not the focus of this chapter. In building any sort of emulator they should be made to best cater to the simulator at hand. In this chapter, the default choice will be to use the weak prior

$$p\left(\beta_i, \sigma_i^2\right) \propto \frac{1}{\sigma_i^2},$$

include all input variables as active, include either all first order or all first and second order terms in the regression surface, and model the error by a stationary, isotropic Gaussian process, whose correlation length is the maximum likelihood estimator. We will check that these choices are appropriate before continuing.

In comparing the hierarchical emulator with the standard approach, we focus here particularly on the choice of independent variable and the use of training data, and will try to make the other emulation choices comparable where possible. Having determined the tasks set for the emulators, the set of 'standard' emulators against which we are to compare the hierarchical emulators should include the choices that intuitively best suit those tasks.

Emulators of s_1

Firstly, we can build a standard emulator of $s_1\left(x, v, w\right)$. This emulator can be used to predict both $s_1\left(\tilde{x}, \tilde{v}, \tilde{w}\right)$ and $s_0\left(\tilde{x}\right)$. If the inputs are all processed together, as a data frame containing $\left(\tilde{x}, \tilde{v}, \tilde{w}\right)$ and $\left(\tilde{x}, v^*, w\right)$ then the covariance matrix will also enable us to calculate $\text{var}\left[s_1\left(\tilde{x}, \tilde{v}, \tilde{w}\right) - s_0\left(\tilde{x}\right)\right]$. This emulator can therefore be used to achieve each of the chosen tasks.

An emulator of $s_1\left(\cdot\right)$ can be built using either only the data where $v \neq v^*$, or all the simulator data available, including data from lower down in the hierarchy. The examples Section 5.5 will include both.

Separate emulators of s_0 and s_1

Instead of using an emulator of $s_1\left(\cdot\right)$ only, one could build separate emulators of $s_1\left(x, v, w\right)$ and $s_0\left(x\right)$, then use these to predict $s_1\left(\tilde{x}, \tilde{v}, \tilde{w}\right)$ and $s_0\left(\tilde{x}\right)$. These can be combined to find the expected difference. Bounds on the variance of the difference for each input point can be calculated using the Cauchy-Schwarz inequality, but the exact value $\text{var}\left[s_1\left(\tilde{x}, \tilde{v}, \tilde{w}\right) - s_0\left(\tilde{x}\right)\right]$ cannot.

Emulating the difference

So long as the training data is set up in the correct way, the difference can be calculated exactly, then emulated using standard methods. Intuitively, this should produce the best prediction of the difference. It doesn't, however, provide a way to see the difference relative to the values of each output, and so can only really be useful when combined with an emulator of $s_0\left(\cdot\right)$ or $s_1\left(\cdot\right)$.

A truly comparable standard emulator

In comparing the performance of the hierarchical emulator with that of the standard method, it seems appropriate, as far as possible, to include the same information in both emulators. In terms of input data, this can be achieved by using the same training data. However, the very structure of the hierarchical emulator includes the information that

$$s_1\left(x, v^*, w\right) = s_0\left(x\right) \quad \text{for all } x, w, \tag{5.16}$$

because of the $g\left(\cdot\right)$ functions which switch off terms as necessary. A fair question to ask then is, can this same information be included in a standard emulator?

A crucial aspect of this information is that when $v = v^*$, the value of w doesn't affect the value of $s_1\left(\cdot\right)$. This information could be incorporated into a regression function, simply by making sure that all terms involving w also involved v in such a way that this was achieved. However, for Equation 5.16 to hold in the correlated error term, the correlation structure would have to be drastically changed. One method would be to have correlation lengths for w that are a function of v, such that when $v = v^*$ the extra inputs w add no variance.

This seems a sufficiently serious deviation from standard emulation for us to be able to compare the hierarchical emulator with those in Section 5.3.2, and to think of the capacity to include the information in Equation (5.16) as a benefit, rather than an unfair advantage, of hierarchical emulation.

5.4 Method summary

Before seeing an example, we quickly summarise the process of building an hierarchical emulator, in a heuristic algorithm.

1. Work out the common, hierarchical and extra input variables, and for each hierarchical variable v_i (for $i = 1, \dots, k$) find v_i^*. This may require the input space to be reparameterised, as in the example in Section 5.5.

2. Design the training data, adhering to the criteria in Section 5.2.2, and run the simulator at these points.

3. Separate the training data into 2^k 'chunks', one for each possible combination of hierarchical variables at their v^* value.

4. For each hierarchical input v_i, use the training data to choose a suitable transformation function $g_i\left(v_i\right)$. This will depend on the choice of regression surface to be used in each each emulator. See Section 5.2.3.

5. Use the $g_i\left(v_i\right)$ functions and the training data to compute the $\boldsymbol{\psi}_{[i]}$ vectors.

6. Build emulators for $s_0\left(x\right)$ and each of the $\left(2^k - 1\right)$ functions $\psi_{[i]}\left(x, v_{[i]}, w\right)$.

7. Use these emulators, combined with the $g_{[i]}\left(\mathbf{v}_{[i]}\right)$ vectors, to emulate $s_1\left(x, v, w\right)$ and $s_1\left(x, v, w\right) - s_0\left(x\right)$, using the results in Section 5.2.4.

8. Validate the emulators using techniques from Section 3.5. In the example in Section 5.5 we use the error, RMSE, SPE and MD. If these expose modelling flaws in the emulators, the previous steps should be revisited, focussing particularly on the structures of the regression surfaces and correlated errors.

5.5 Example - C:Chl ratio in HadOCC

In Section 4.2 various examples were given of simulator differences in HadOCC and OG99NPZD. One that is relevant to this method is the ability to change the carbon:chlorophyll (C:Chl) ratio in HadOCC, explained in Section 4.2.1. If the switch variable `rcchlopt` is set to 0, then C:Chl is constant and determined by the input parameter `rcchl`. If `rcchlopt` $= 1$, C:Chl varies, and as well as `rcchl` (which now functions as the initial value of C:Chl) we must give `rcchlmin` and `rcchlmax`, the minimum and maximum values for C:Chl. When

$$\texttt{rcchlmin = rcchl = rcchlmax},$$

the `rcchlopt` $= 1$ version of HadOCC performs identically to the `rcchlopt` $= 0$ version with the same value of `rcchl`. Therefore this pair of versions of HadOCC can be used to build a hierarchical emulator. The variable `iz.chl` is a good choice of output because it is strongly affected by the C:Chl ratio.

5.5.1 Different parameterisations

Although the two versions of HadOCC described above are identical for certain input points, we cannot immediately discern the hierarchical variables v and extra variables w. For this, the inputs must be re-parameterised. Two valid parameterisations of `rcchlmin, rcchl` and `rcchlmax` are introduced here, each of which could be used to build a hierarchical emulator.

Cuboid parameterisation

One difference between the possible parameterisations is the shape of the new input space. This parameterisation takes the three inputs related to C:Chl; `rcchlmin, rcchl` and `rcchlmax`, and produces three more, R, m_1 and m_2, whose ranges form a cuboid. These are defined by

$$\begin{aligned} R &= \texttt{rcchlmax} - \texttt{rcchlmin} \in [0, M_+ - M_-], \\ m_1 &= \frac{\texttt{rcchlmin} - M_-}{M_+ - M_- - R} \in [0,1], \\ m_2 &= \frac{\texttt{rcchlmax} - \texttt{rcchl}}{R} \in [0,1], \end{aligned}$$

where M_- and M_+ are the minimum and maximum respectively for `rcchlmin, rcchl` and `rcchlmax`, given that each has the same range.

R denotes the difference between `rcchlmin` and `rcchlmax`, and is the only hierarchical variable. When $R = 0$, the two versions of HadOCC are the same. The input m_1 is a common input variable, since this is how the value of `rcchl` is found in order to run $s_0(x)$ in the cuboid parameterisation.

When $R = 0$, `rcchlmin = rcchl`, and

$$m_1 = \frac{\texttt{rcchl} - M_-}{M_+ - M_-},$$

and so m_1 replaces `rcchl` in the common inputs. Finally, m_2 is an extra variable, and exists only when $R \neq 0$.

The fact that this parameterisation induces a cuboid shape in the inputs means that generating input Latin hypercubes is simple. In fact, because of the constraint in the original input space for $s_1(x, \texttt{rcchlmin, rcchlmax})$, that

$$\texttt{rcchlmin} \leq \texttt{rcchl} \leq \texttt{rcchlmax},$$

generating designs is more straightforward in the cuboid input space than before. That said, this parameterisation may create scaling issues. When R is small, the effect of m_1 is strong, since the small range can be anywhere within $[M_-, M_+]$, but the effect of m_2 is small, because there is only a small range in which `rcchl` can sit. When R is large, this is reversed. Whether or not this is a problem is not immediately obvious. Another advantage of this parameterisation is that having only one hierarchical variable, R, means that the training data design criteria do not force us to have too great a number of points.

Non-cuboid parameterisation

A second valid re-parameterisation is to keep `rcchl` as it is, and to introduce

$$d_{min} = \texttt{rcchl-rcchlmin} \in [0, \texttt{rcchl} - M_-]$$
$$d_{max} = \texttt{rcchlmax-rcchl} \in [0, M_+ - \texttt{rcchl}].$$

These are now both hierarchical inputs, as both must be zero in order to achieve

$$\texttt{rcchlmin = rcchl = rcchlmax}.$$

This parameterisation does not create the possible scaling problems that the cuboid parameterisation does, but the shape of the input space is more complicated, which makes generating a training data input design less straightforward. Because there are now two hierarchical inputs, the hierarchical emulator includes four terms, and so in order to find data for $s_0(\cdot)$, $\psi_1(\cdot)$ $\psi_2(\cdot)$ and $\psi_{12}(\cdot)$, each point (x, d_{min}, d_{max}) for which $d_{min}, d_{max} \neq \mathbf{0}$ demands that we also evaluate $s_1(\cdot)$ at $(x, d_{min}, 0)$, $(x, 0, d_{max})$ and $(x, 0, 0)$. This parameterisation would therefore make hierarchical emulation more costly and impractical were s_1 expensive to run.

For this example, we will use only the cuboid parameterisation, but an investigation into the effect of the choice of parameterisation would no doubt raise interesting questions. It is likely that in many cases there will be a choice of emulator input space, and choosing the most suitable one is an important part of the problem.

5.5.2 Cuboid design - example

Here, we show some hierarchical emulators of the two versions of HadOCC built using the cuboid parameterisation, and compare them to some standard emulators.

Training data

The training data for this example was formed using a 1,000 point Latin hypercube over the entire re-parameterised input space of $s_1(x, R, m_2)$. HadOCC was run at each point, and the annual mean `iz.chl` calculated. To satisfy the design requirements, the annual mean of `iz.chl` was also found for $s_0(x)$ for each point. This gave an input design of 2,000 points, which we will refer to as 'lhd1'. Of this, the sub-design containing points at which $R = 0$ will be 'lhd1_0' and the subset of points at which $R \neq 0$ will be 'lhd1_1'.

Note that the corresponding points in lhd1_0 and lhd1_1 share common m_1 values, rather than common `rcchl` values. This means we must be careful to use the reparameterised input variables when building standard emulators of the difference.

Emulation choices

This study involves the building of many emulators, and in order for them to be as comparable as possible, we decided initially to fix the type of emulator. Therefore in this section, the regression terms are all first order, with all input variables included, and the correlated error terms are isotropic, with the correlation length found using maximum likelihood.

On looking at the annual mean `iz.chl` output values, it became apparent that the logarithm of the annual mean would be the most appropriate choice of output variable. Figure 5.1 shows the s_1 and s_0 output compared both on the original and log scale. Each point on the plot compares the two outputs with all shared inputs the same in the cuboid parameterisation. Where C:Chl is varying, the values of R and m_2 are different for each point, as the data is lhd1, a randomly generated LHD. It appears that on the original output scale, the relationship between the two simulators is highly multiplicative. The Box-Cox procedure strongly suggested $\log(\texttt{iz.chl})$ as the best choice of output for standard emulators of $s_0(\cdot)$ and $s_1(\cdot)$.

Finding the best $g(R)$

In Section 5.2.3, some candidates for the $g(\cdot)$ functions were proposed. Here, only one function $g(R)$ is necessary, and we will consider the possibilities

$$g(R) = R^{\lambda}$$

and

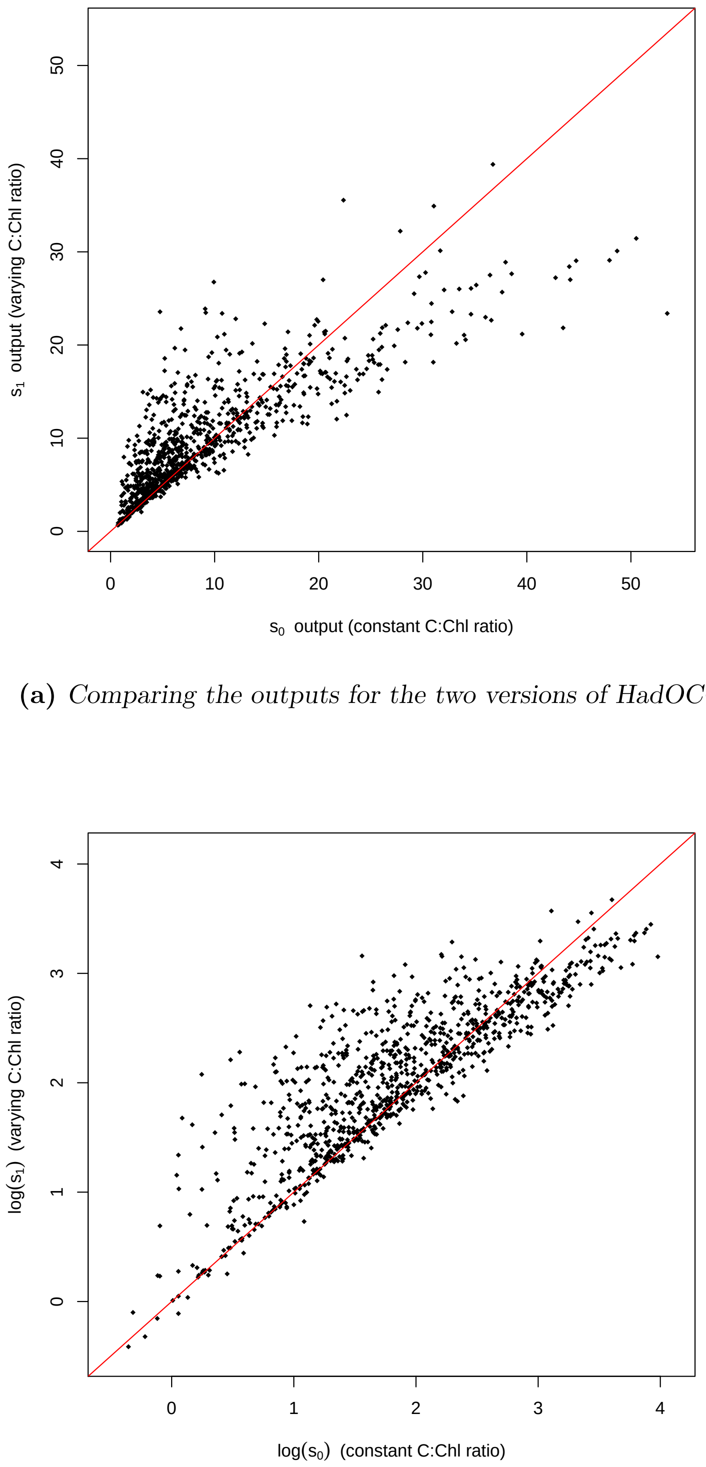
$$g(R) = [\log(R+1)]^{\lambda}$$

for some $\lambda > 0$. Evaluating these functions on our training data lhd1 allows us to find the corresponding $\boldsymbol{\psi}$ vectors, by

$$\boldsymbol{\psi} = \frac{\mathbf{s}_1 - \mathbf{s}_0}{g(R)}.$$

First order regression surfaces can then be fit to these vectors, and the vector of residuals (referred to hereafter as the 'h residuals') analysed for skewness and closeness to normality[2]. Table 5.1 shows summaries for some different functions $g(R)$.

[2] By "closest to normality", we mean that the standardised regression residuals of the $\mathbf{s}$ vector resulting from this transformation gave the highest p-value in the Kolmogorov-Smirnov test, using $N(0,1)$ as the null distribution.



(a) *Comparing the outputs for the two versions of HadOCC*

(b) *The same plot as above, but on logarithm scale*

Figure 5.1: *Annual mean* `iz.chl` *(top) and* $\log(\texttt{iz.chl})$ *(bottom) from HadOCC with varying C:Chl compared with fixed, using the dataset lhd1. The line* $y = x$ *is added.*

The two best-looking transformations are

$$g(R) = R^{0.5278}$$

which minimises skewness, and

$$g(R) = \left[\log(R+1)\right]^{1.881}$$

which gives standardised residuals that are most plausible under $N(0,1)$. We will pursue both, in order to gain insight into which might be a better criterion.

$g(R)$	Skewness of h residuals	Kurtosis of h residuals	p-value from Kolmogorov-Smirnov test
R	-3.38	29.0	3.05×10^{-8}
$\log(R+1)$	0.162	4.77	0.00152
Optimised with respect to standardised h residuals from lhd1			
$R^{0.5169}$	0.0114	4.68	*0.0111*
$R^{0.5278}$	$\mathit{-4.89 \times 10^{-5}}$	4.65	0.00932
$\left[\log(R+1)\right]^{1.881}$	-0.0998	4.86	*0.0427*
$\left[\log(R+1)\right]^{1.582}$	$\mathit{-1.34 \times 10^{-4}}$	4.85	0.00932

Table 5.1: *Summaries of the standardised h-residual vectors from linear regressions of the $\psi(x, R, m_2)$ vectors created by some $g(R)$. The skewness and kurtosis should be 0 and 3 respectively if the h residuals are normally distributed. In the bottom two chunks, the quantities in italics are the optimised quantities. The Kolmogorov-Smirnov test is used here as a goodness-of-fit test with $N(0,1)$ as the null distribution. A higher probability indicates a better fit to the distribution.*

Validation data

Fortunately, HadOCC is relatively quick to run, and so we were able to produce a relatively large dataset with which to validate our emulators. The design 'lhd6' was formed using a one million point Latin hypercube built using the staggered LHD method introduced in Section 3.5.1, with $c = 1000$ and $m = 1000$. This means that it breaks down into 1,000 sub-LHDs, each containing 1,000 points. Each of these points was matched by a corresponding point with $R = 0$. HadOCC was then run at all two million points, to produce both s_1 and s_0 data. For each emulator therefore, we can produce 1,000 sets of prediction summaries for each version of HadOCC, one for each sub-LHD.

Standard emulator performance

Diagnostics are summarised in all following tables by their minima, maxima, mean and standard deviation. Plots will also be given where they show interesting results. Because the Mahalanobis distance combines so much information, it will not be used to choose between methods, although the values will be shown for some emulators.

This section summarises the standard emulators' performance at the first three tasks described in Section 5.3.1: predicting $s_0(x)$, predicting $s_1(x, v, w)$ and predicting the difference. For quick reference, the 'best standard emulator' choices are summarised in Table 5.2.

Emulator		Min.	Max.	Mean	SD
Cuboid	RMSE $(\log(s_0))$	0.114	0.137	0.125	0.00437
inputs, lhd1_0,	Mean SPE $(\log(s_0))$	-0.103	0.0502	-0.0261	0.0268
(Table 5.3)	Variance SPE $((s_0))$	0.940	1.43	1.18	0.0804
Cuboid	RMSE $(\log(s_1))$	0.126	0.157	0.140	0.00507
inputs, lhd1_1,	Mean SPE $(\log(s_1))$	-0.120	0.0705	-0.0268	0.0301
(Table 5.4)	Variance SPE $((s_1))$	0.940	1.44	1.15	0.0817
Difference	RMSE $\left(\log\left(\frac{s_1}{s_0}\right)\right)$	0.0941	0.115	0.103	0.00357
data,	Mean SPE $\left(\log\left(\frac{s_1}{s_0}\right)\right)$	-0.0979	0.114	-0.00878	0.0304
(Table 5.7)	Variance SPE $\left(\log\left(\frac{s_1}{s_0}\right)\right)$	0.948	1.39	1.12	0.0706

Table 5.2: *Summaries for the best standard emulators of* $\log(s_0)$, $\log(s_1)$ *and* $\log(s_1 / s_0)$ *used over lhd6.*

Predicting $s_0(x)$ output

Table 5.3 shows diagnostics for a standard emulator of $s_0(x)$ built using $s_0(x)$ data only. This uses the cuboid parameterisation and only the data lhd1_0 from $s_0(x)$, and is therefore exactly the emulator used for the first term of the hierarchical emulator. Diagnostics for s_0 are therefore omitted from the hierarchical emulator tables. Output from $s_0(x)$ can also be predicted using emulators of $s_1(\cdot)$, and so diagnostic summaries for $s_0(x)$ are also shown in Tables 5.4, 5.5 and 5.6.

In Section 3.6, the emulator of HadOCC with isotropic correlation lengths had very poor properties, shown most clearly in the plot of SPE against `rcchl` in Figure 3.8. In this Section therefore it was important to check the behaviour of the SPE values, particularly in relation to `rcchl` and related variables. It turns out that when the cuboid parameterisation is used, the SPE from a similar emulator shows good behaviour against all inputs and against the predicted values. Figure 5.2a shows this when the $s_1(\cdot)$ data (and not the $s_0(\cdot)$ data) are used to emulate $s_1(\cdot)$.

The emulator in Table 5.3, built using lhd1_0 ($s_0(x)$ data only) and the cuboid parameterisation outperforms each of the other standard emulators at predicting $s_0(x)$. The RMSEs are smallest, mean error closest to zero and the mean and standard deviation of the SPE consistently close to 0 and 1. The alternatives perform much worse, with those built from lhd1_1 (shown in Table 5.4) severely overestimating $\log(\texttt{iz.chl})$, and SPE summaries in Table 5.6 showing that the full lhd1 dataset gives very poor variances for $s_0(x)$.

	Minimum	Maximum	Mean	SD
RMSE $(\log(s_0))$	0.1126	0.1366	0.1247	0.004366
Mean error $\log(s_0)$	-0.006476	0.01185	0.002920	0.003120
Mean SPE $(\log(s_0))$	-0.1033	0.05020	-0.02609	0.02683
Variance SPE $(\log(s_0))$	0.9402	1.4300	1.1778	0.08041

Table 5.3: *Diagnostic summaries for the thousand sets of predictions found using the standard emulator of* $\log(s_0)$ *over lhd6, using the cuboid parameterisation, and lhd1_0. This is exactly the emulator of* s_0 *used for the first term of the hierarchical emulator*

Predicting $s_1(\cdot)$ output

Tables 5.4, 5.5, 5.6 show performance summaries of emulators of $\log(s_1(x, R, m_2))$, built using various combinations of training data.

The best method for predicting $s_1(x)$ appears to be the cuboid parameterisation built with the training data lhd1_1 (Table 5.4).

	Minimum	Maximum	Mean	SD
RMSE $(\log(s_0))$	0.1742	0.1991	0.1864	0.004218
Mean error $\log(s_0)$	-0.08592	-0.06535	-0.07648	0.003446
Mean SPE $(\log(s_0))$	-0.6250	-0.4664	-0.5512	0.02468
Variance SPE $(\log(s_0))$	1.243	1.685	1.486	0.07501
RMSE $(\log(s_1))$	0.1258	0.1573	0.1397	0.00507
Mean error $\log(s_1)$	-0.0164	0.00740	0.00445	0.00395
Mean SPE $(\log(s_1))$	-0.1203	0.0705	-0.02679	0.03011
Variance SPE $(\log(s_1))$	0.9401	1.437	1.150	0.08170
RMSE $(\log(s_1 / s_0))$	0.1363	0.1623	0.1482	0.00384
Mean error $\log(s_1 / s_0)$	0.0649	0.0804	0.0721	0.00246
Mean SPE $(\log(s_1 / s_0))$	1.545	1.876	1.700	0.05626
Variance SPE $(\log(s_1 / s_0))$	5.640	12.90	8.464	1.145

Table 5.4: *Diagnostic summaries for the thousand sets of predictions using the standard emulator of* $\log(s_1)$ *over lhd6, using the cuboid parameterisation and the data lhd1_1.*

	Minimum	Maximum	Mean	SD
RMSE $(\log(s_0))$	0.1869	0.2086	0.1985	0.003341
Mean error $\log(s_0)$	-0.1457	-0.1229	-0.1354	0.03389
Mean SPE $(\log(s_0))$	-1.0217	-0.8776	-0.9532	0.02302
Variance SPE $(\log(s_0))$	0.8301	1.1892	1.0039	0.05165
RMSE $(\log(s_1))$	0.1324	0.1635	0.1467	0.005049
Mean error $\log(s_1)$	-0.01962	0.005789	-0.007970	0.003860
Mean SPE $(\log(s_1))$	-0.1396	0.04308	-0.0586	0.02887
Variance SPE $(\log(s_1))$	1.008	1.496	1.217	0.08090

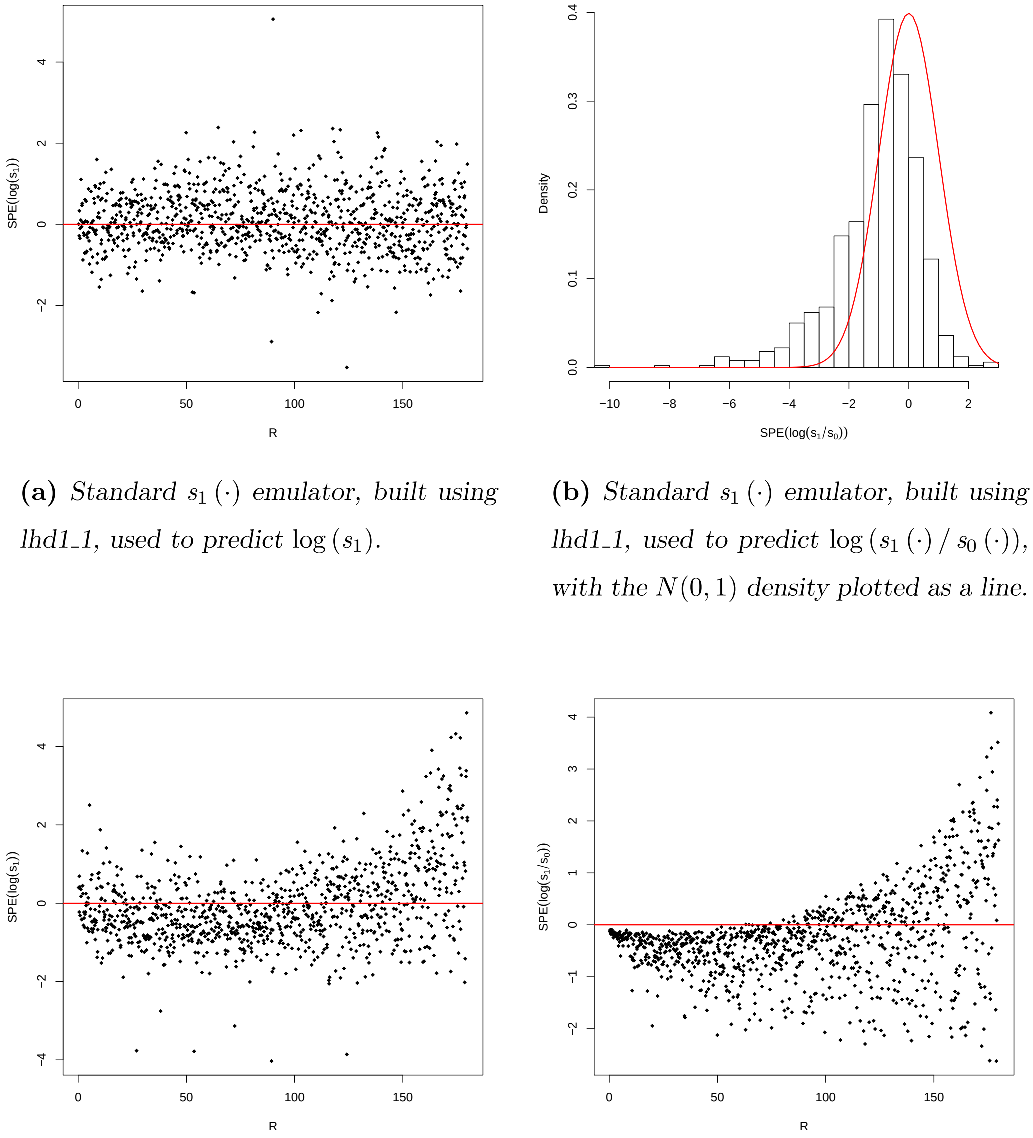
Table 5.5: *Diagnostic summaries for the thousand sets of predictions over lhd6 using the standard emulator of* $\log(s_1)$*, on the original HadOCC input space and using the data lhd1_1.*

	Minimum	Maximum	Mean	SD
RMSE $(\log(s_0))$	0.1652	0.1996	0.1801	0.005196
Mean error $\log(s_0)$	-0.02144	0.01007	-0.006785	0.004972
Mean SPE $(\log(s_0))$	-0.08462	0.02336	-0.03239	0.01825
Variance SPE $(\log(s_0))$	0.3699	0.5229	0.4458	0.02400
RMSE $(\log(s_1))$	0.1777	0.2132	0.1942	0.005093
Mean error $\log(s_1)$	-0.03457	0.001028	-0.01420	0.005443
Mean SPE $(\log(s_1))$	-0.1355	0.004913	-0.05873	0.02071
Variance SPE $(\log(s_1))$	0.4836	0.6611	0.5695	0.02894
RMSE $(\log(s_1 / s_0))$	0.0972	0.1254	0.1126	0.004334
Mean error $\log(s_1 / s_0)$	-0.01672	0.003560	-0.007415	0.003398
Mean SPE $(\log(s_1 / s_0))$	-0.1579	0.1580	-0.0060	0.04652
Variance SPE $(\log(s_1 / s_0))$	1.724	3.846	2.389	0.2763

Table 5.6: *Summaries of the diagnostics for the thousand sets of predictions found using the standard emulator of* $\log(s_1)$ *over lhd6, using the standard parameterisation, and lhd1, that is all* s_0 *and* s_1 *training data (2,000 points).*

The plots in Figure 5.2 illustrate the standard emulators' performance as the hierarchical input R changes. The top two plots show the SPE for the standard emulator of $s_1(\cdot)$ built using lhd1_1, the emulator summarised in Table 5.4. Both the SPE for $\log(s_1)$ and the SPE for $\log(s_1 / s_0)$ show no trend with R, but the distribution of the SPE values for the ratio is heavily biased. The bottom two plots show the same summaries for the emulator built from lhd1. This time, both show a considerable trend in SPE against R, particularly the ratio prediction. Several of the other standard emulators, particularly the emulator built from the difference data, show a similar pattern, with SPE values becoming more variable with increased R.

Figure 5.3c shows the Mahalanobis distances, transformed to fit the F-distribution as described in Section 3.5, compared with the true F-distribution, and the correspondence is poor.



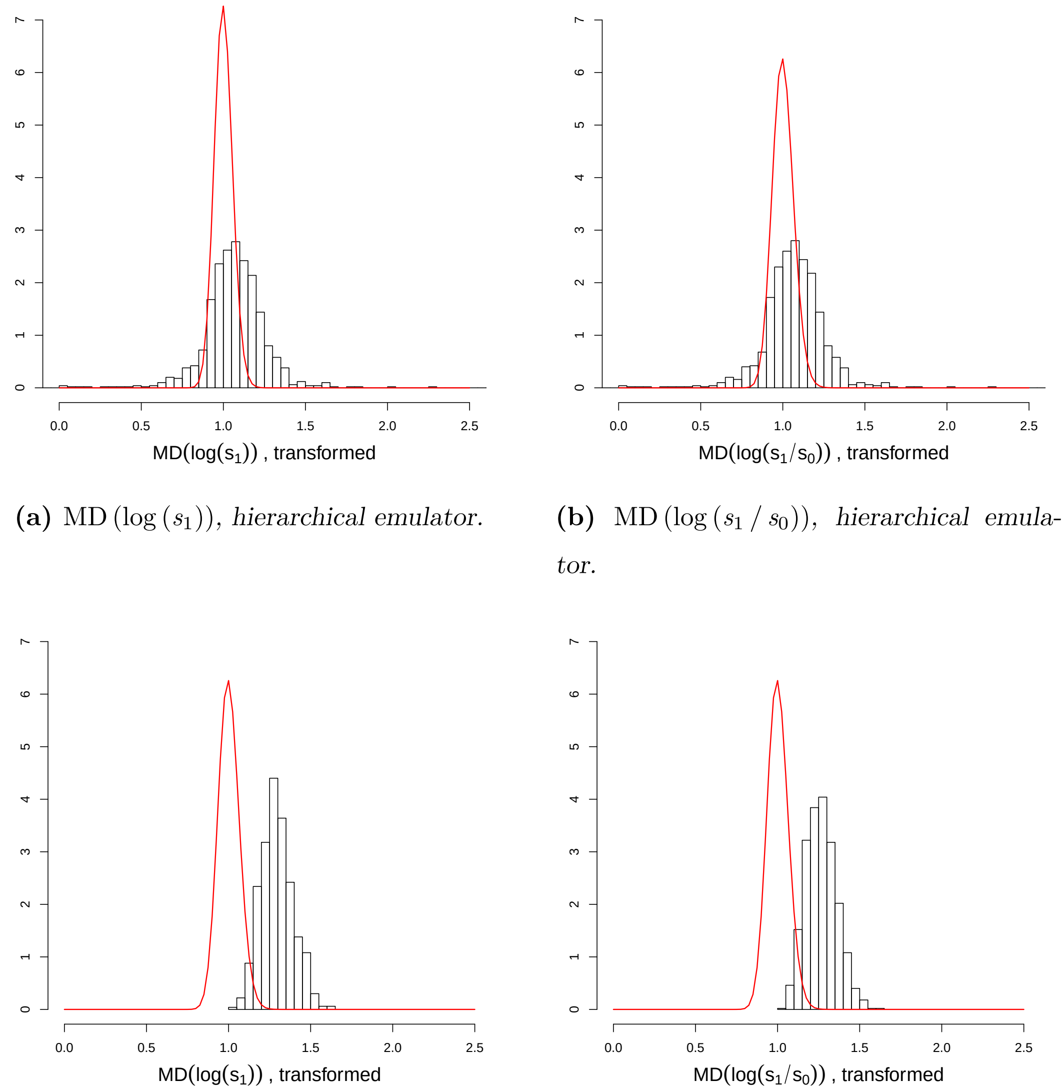
(a) *Standard $s_1(\cdot)$ emulator, built using lhd1_1, used to predict $\log(s_1)$.*

(b) *Standard $s_1(\cdot)$ emulator, built using lhd1_1, used to predict $\log(s_1(\cdot)\,/\,s_0(\cdot))$, with the $N(0,1)$ density plotted as a line.*

(c) *Standard $s_1(\cdot)$ emulator, built using lhd1, used to predict $\log(s_1)$.*

(d) *Standard $s_1(\cdot)$ emulator, built using lhd1, used to predict $\log(s_1(\cdot)\,/\,s_0(\cdot))$.*

Figure 5.2: *Comparing standardised prediction errors (SPE) for predictions of $\log(s_1)$ (left-hand plots) and $\log[s_1(\cdot)\,/\,s_0(\cdot)]$ (right hand plots) for standard emulators built with different training data. The prediction data set used for these plots is a pair of 1,000 point sub-LHDs from the lhd1 data.*



(a) MD $(\log(s_1))$, *hierarchical emulator.*

(b) MD $(\log(s_1 \,/\, s_0))$, *hierarchical emulator.*

(c) MD $(\log(s_1))$, *standard emulator of* $\log(s_1)$ *built from lhd1_1.*

(d) MD $(\log(s_1 \,/\, s_0))$, *standard emulator of* $\log(s_1(\cdot) \,/\, s_0(\cdot))$ *built from ratio data.*

Figure 5.3: *Comparing the transformed Mahalanobis distances for the predictions of* $\log(s_1)$ *(left-hand panels) and* $\log(s_1(\cdot) \,/\, s_0(\cdot))$ *(right-hand panels), for the hierarchical emulator with* $g(R) = R^{0.5278}$ *(top panels) and the best standard emulator (bottom panels). The* F*-distribution densities are added, which should match the distribution of the transformed MD values.*

Predicting the difference

Because the emulator output is log (`iz.chl`), the output for the difference emulator becomes

$$\log(s_1(x, R, m_2)) - \log(s_0(x)) = \log\left(\frac{s_1(x, R, m_2)}{s_0(x)}\right),$$

and so in fact it is the ratio of simulator outputs being considered.

Table 5.7 shows performance summaries of an emulator of the difference between simulators. The difference can also be predicted using an emulator of $s_1(\cdot)$ built with the cuboid parameterisation, and so summaries are also shown in Tables 5.4 and 5.6. The emulator in Table 5.5 can't be used to find ratios in the same way, because the paired points have common m_1 values, rather than common `rcchl` values.

The difference is best predicted by the direct emulator of the difference between logs calculated from the data lhd1, as shown in Table 5.7. Again, although the predictions from the lhd1 emulator in Table 5.6 are quite accurate, the SPE and MD summaries show that the emulator's variance is not as it should be according to the model.

Figure and 5.3d shows the Mahalanobis distances, transformed to fit the F-distribution as described in Section 3.5, compared with the true F-distribution, and the correspondence is poor.

	Minimum	Maximum	Mean	SD
RMSE $(\log(s_1 / s_0))$	0.09409	0.1145	0.1030	0.00357
Mean error $\log(s_1 / s_0)$	-0.01059	0.009917	-0.001917	0.00295
Mean SPE $(\log(s_1 / s_0))$	-0.0979	0.1135	-0.00878	0.0304
Variance SPE $(\log(s_1 / s_0))$	0.9477	1.390	1.115	0.0706

Table 5.7: *Diagnostic summaries for the thousand emulators built from the standard emulator of* $\log(s_1 / s_0)$ *over lhd6, using the cuboid parameterisation.*

Hierarchical Emulators

The same diagnostics can be found for the hierarchical emulators as for the standard ones, and these are shown in Tables 5.8 (for $g(R) = R^{0.5278}$) and 5.9 (for

$g(R) = [\log(R+1)]^{1.881}$).

	Minimum	Maximum	Mean	SD
RMSE $(\log(s_1))$	0.1110	0.1411	0.1249	0.00483
Mean error $\log(s_1)$	-0.0153	0.00488	-0.00544	0.00351
Mean SPE $(\log(s_1))$	-0.1054	0.04253	-0.03368	0.02529
Variance SPE $(\log(s_1))$	0.6628	1.1083	0.8266	0.06385
RMSE $(\log(s_1 / s_0))$	0.07543	0.09944	0.08623	0.00391
Mean error $\log(s_1 / s_0)$	-0.0101	0.00641	-0.00252	0.00239
Mean SPE $(\log(s_1 / s_0))$	-0.1212	0.05287	-0.04070	0.02752
Variance SPE $(\log(s_1 / s_0))$	0.9144	1.2911	1.0863	0.06558

Table 5.8: *Summaries of the diagnostics for the thousand emulators build from the hierarchical emulator with $g(R) = R^{0.5278}$, used over lhd6.*

	Minimum	Maximum	Mean	SD
RMSE $(\log(s_1))$	0.1137	0.1430	0.1269	0.004896
Mean error $\log(s_1)$	-0.01579	0.004716	-0.005260	0.003615
Mean SPE $(\log(s_1))$	-0.1102	0.04184	-0.03355	0.02592
Variance SPE $(\log(s_1))$	0.6773	1.1235	0.8475	0.06432
RMSE $(\log(s_1 / s_0))$	0.07969	0.1020	0.08904	0.003881
Mean error $\log(s_1 / s_0)$	-0.009962	0.007323	-0.002326	0.002503
Mean SPE $(\log(s_1 / s_0))$	-0.09556	1.205	0.002468	0.07526
Variance SPE $(\log(s_1 / s_0))$	0.9851	1587.8	6.6829	59.73

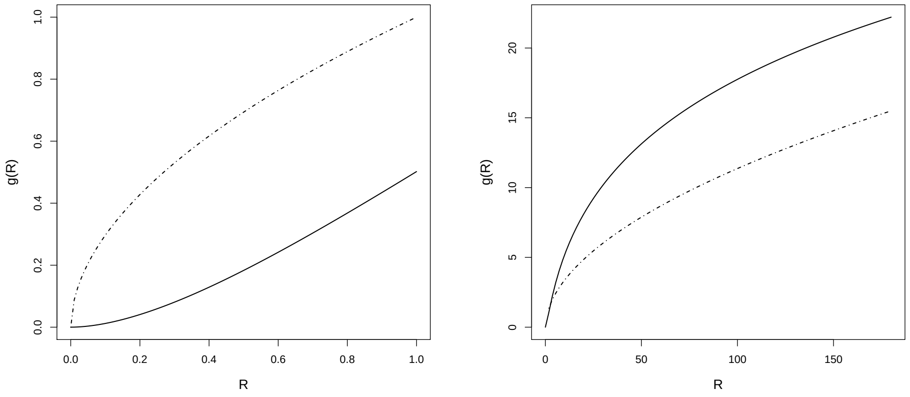
Table 5.9: *Summaries of the diagnostics for the thousand emulators build from the hierarchical emulator with $g(R) = [\log(R+1)]^{1.881}$, used over lhd6. Eight of the sub-LHDs didn't run properly, and some ratio predictions gave very unlikely values.*

The logarithmic transformation function created some problems for values of R close to zero, with the result that eight of the 1,000 LHDs in the lhd1 design produced singular matrices and failed to run. Problems can also be seen through the distribution of the SPE values, where the maximum value of 1588 is the result of a very small R.

Despite this, the predictions are more accurate using a hierarchical emulator with either $g\left(R\right)$ than with the best standard alternatives shown in Table 5.2, shown by the smaller RMSE values for both $\log\left(s_1\right)$ and $\log\left(s_1 \,/\, s_0\right)$. Recall that the summaries for the hierarchical emulators' predictions of $\log\left(s_0\right)$ will be identical to that of the best standard emulator, shown in Tables 5.2 and 5.3.

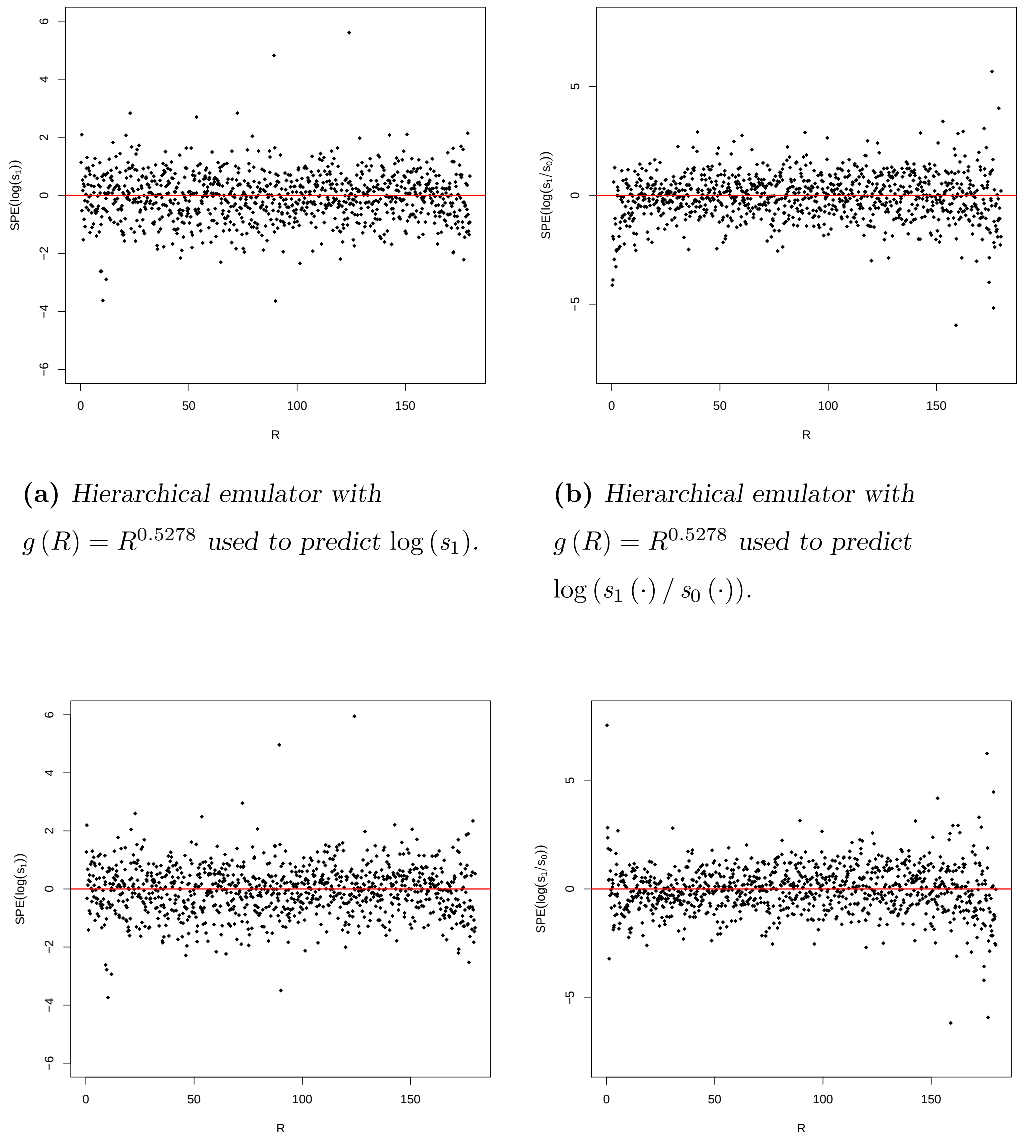
Diagnostics incorporating the variance structure show $g\left(R\right) = R^{0.5278}$ to produce better results than $g\left(R\right) = \left[\log\left(R+1\right)\right]^{1.881}$. The SPE values for the log-ratio are very promising, with mean and variance consistently close to 0 and 1 respectively, although there is a slight negative bias. Similarly, the SPE for $\log\left(s_1\right)$ shows a slight negative bias, but is close to having mean zero and variance one. Figure 5.5 shows the behaviour of the SPEs for predictions of $\log\left(s_1\right)$ and $\log\left[s_1 \,/\, s_0\right]$ for the two hierarchical emulators.

The hierarchical emulator with $g\left(R\right) = \left[\log\left(R+1\right)\right]^{1.881}$ has particularly poor SPE values for the ratio, but the individual results reveal that errors particularly unlikely under $N(0,1)$ usually come from points with very small R values. Figure 5.4 shows the two candidate $g\left(R\right)$ functions over the interval $[0,\ 1]$, and over the full range of R. While $R^{0.5278}$ enlarges values close to zero, the function $\left[\log\left(R+1\right)\right]^{1.881}$ shrinks them, which turns out to be a most undesirable property.



(a) *Over* $(0,1)$ **(b)** *Over* $(0,180)$

Figure 5.4: *Comparing the two* $g\left(R\right)$ *functions under consideration,* $\left[\log\left(R+1\right)\right]^{1.881}$ *(solid line) and* $R^{0.5278}$ *(dot-dashed line), over* $[0,1]$ *(left) and over the range of* R *(right).*



(a) *Hierarchical emulator with* $g\left(R\right) = R^{0.5278}$ *used to predict* $\log\left(s_1\right)$.

(b) *Hierarchical emulator with* $g\left(R\right) = R^{0.5278}$ *used to predict* $\log\left(s_1\left(\cdot\right) / s_0\left(\cdot\right)\right)$.

(c) *Hierarchical emulator with* $g\left(R\right) = \big(\log\left(R+1\right)\big)^{1.881}$ *used to predict* $\log\left(s_1\right)$.

(d) *Hierarchical emulator with* $g\left(R\right) = \big(\log\left(R+1\right)\big)^{1.881}$ *used to predict* $\log\left(s_1\left(\cdot\right) / s_0\left(\cdot\right)\right)$.

Figure 5.5: *Comparing standardised prediction error (SPE) values for predictions of* $\log\left(s_1\right)$ *(left-hand plots) and* $\log\left[s_1\left(\cdot\right) / s_0\left(\cdot\right)\right]$ *(right hand plots) for hierarchical emulators built using two* $g\left(R\right)$ *functions and training data lhd1. The prediction data set used for these plots is a pair of 1,000 point sub-LHDs from the lhd6 data.*

Figure 5.3 shows the distribution of the Mahalanobis distances from the validation study, for the hierarchical emulator with $g\left(R\right) = R^{0.5278}$, and for the most successful standard emulators. Although none of these plots shows a good fit to the true distribution, shown by a solid line in each plot, the hierarchical emulator's values are closer.

5.5.3 Working with reduced $s_1(\cdot)$ data

An advantage of hierarchical emulation that was mentioned earlier is the ability to use training data containing many runs of $s_0(x)$ and comparatively few of $s_1(x, v, w)$. This is particularly helpful when $s_1(x, v, w)$ is much more costly to run than $s_0(x)$.

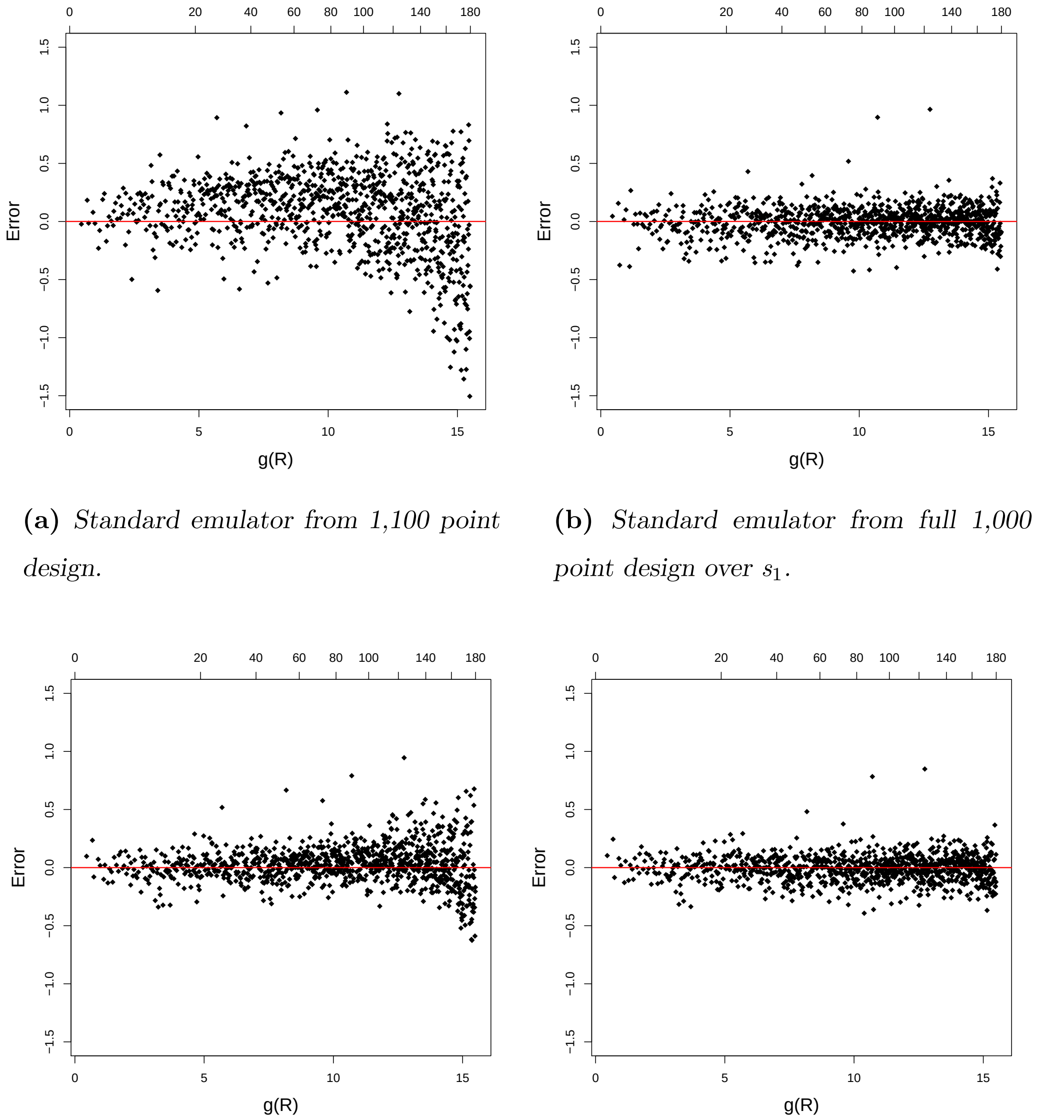
So far, our example has concentrated on emulators built from a training data design containing equal numbers of points from the two simulators. In this section, we build an emulator using 1,000 input points for the simpler version of HadOCC, and only 100 for the more complex version. These hundred runs from s_1 are all matched in their x and m_1 values by a point in the s_0 data, and therefore the design satisfies the criteria in Section 5.2.2. They were taken from a 2,000 point design, so that a hierarchical emulator could also be built with 1,000 points each in the s_0 and s_1 input spaces.

The emulator was constructed using the same choices as the previous ones in this example, namely a first order regression surface involving all terms and a single estimated correlation length for each level of the hierarchy. The transformation function $g(R) = R^{0.5278}$ was used. Three other emulators were also built, to compare with this reduced s_1 emulator. A standard emulator was built using the 100 s_1 and 1,000 s_0 points (i.e. the same reduced s_1 data as the hierarchical emulator), and another using just the 1,000 s_1 points[3]. A second hierarchical emulator was built from the full 2,000 point design.

It was suspected that when a comparatively small number of s_1 data were used, the emulator would perform considerably better 'closer to s_0', i.e. for smaller values of R. This feature has not manifested itself particularly in the emulators with equal numbers of points for both simulators. The errors (emulator prediction minus simu-

[3] From the previous examples of standard emulators, this appears to be a better strategy than using all 2,000 points. Compare Tables 5.5 and 5.6

lator output) for each emulator are plotted against R in Figures 5.6 (for emulators of $\log(s_1)$) and 5.7 (for emulators of $\log(s_1) - \log(s_0)$). The standard emulators summarised in Figures 5.6a and 5.7a use the same data as the hierarchical emulators in Figures 5.6c and 5.7c.



(a) *Standard emulator from 1,100 point design.*

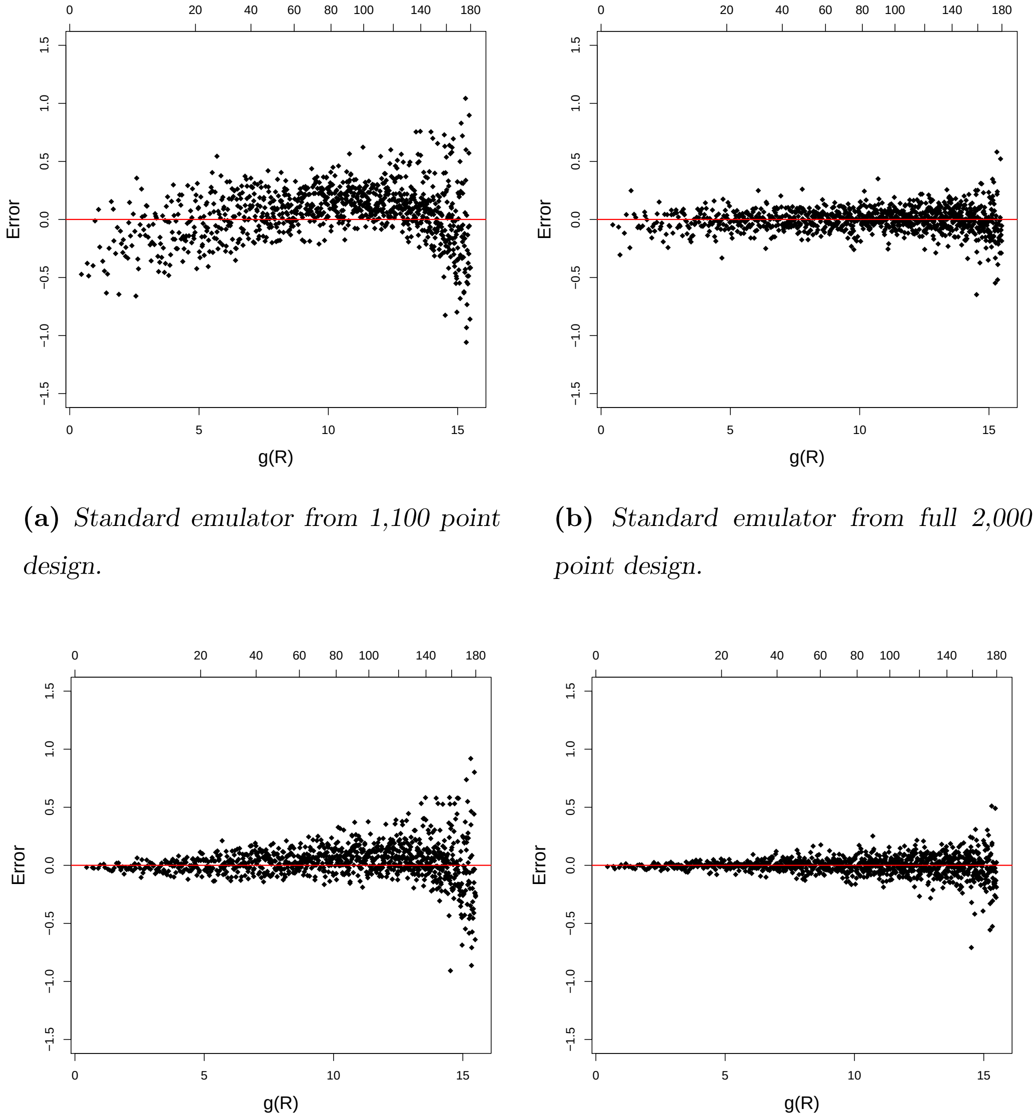
(b) *Standard emulator from full 1,000 point design over s_1.*

(c) *Hierarchical emulator from 1,100 point design.*

(d) *Hierarchical emulator from full 2,000 point design.*

Figure 5.6: *Errors for four emulators of* $\log(s_1)$, *plotted against* $g(R)$ *(bottom axis) and* R *(top axis).*

This clearly shows a great reduction in the error when a hierarchical emulator is used, particularly when R is small. Improvement is greatest in the hierarchical emulators of the difference, where the hierarchical structure enforces the relationship between the simulators, so that the difference when $R = 0$ is always zero. The error gradually increases with R, (see Figure 5.7c) until its accuracy appears roughly the same as the standard alternative.



(a) *Standard emulator from 1,100 point design.*

(b) *Standard emulator from full 2,000 point design.*

(c) *Hierarchical emulator from 1,100 point design.*

(d) *Hierarchical emulator from full 2,000 point design.*

Figure 5.7: *Errors for four emulators of* $\log(s_1) - \log(s_0)$*, plotted against* $g(R)$ *(bottom axis) and* R *(top axis).*

Figure 5.8 shows how the RMSE changes as the range of R in the data changes. Because the number of points used to find the RMSE changes along the x axis, we should be careful when comparing these values. In particular, the values with small maximum R contain far fewer points, and are therefore much more changeable.

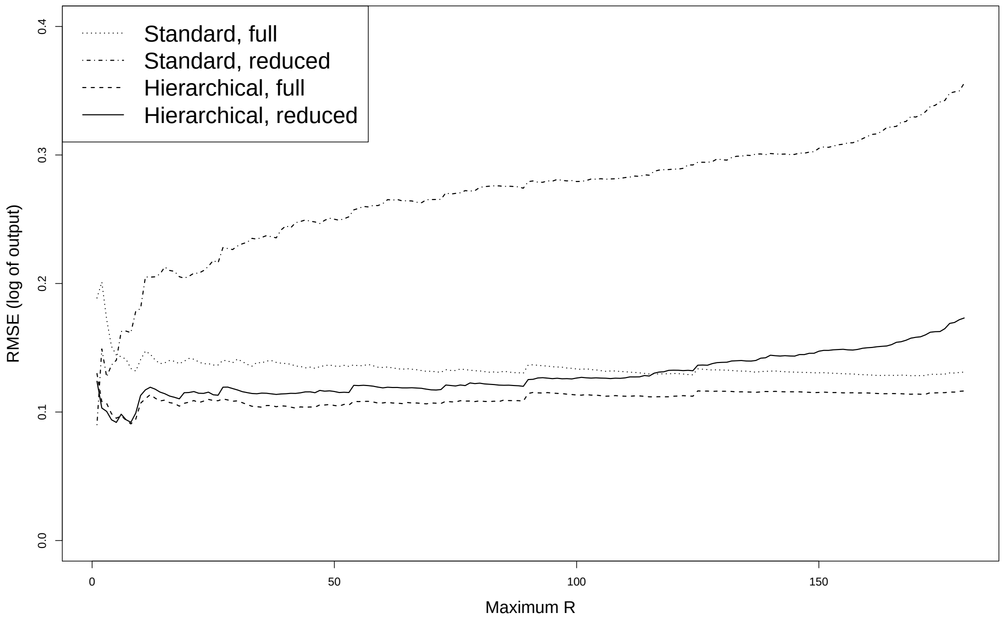
Figure 5.8a compares four emulators being used to predict $\log\left(s_1\right)$. The hierarchical emulator built with the reduced s_1 data (the solid line) shows a clear increase in RMSE as R increases, performing very well when R is small. For R less than roughly 100, this emulator outperforms the standard emulator built with all 1,000 s_1 input points. The hierarchical emulator built with all 2,000 data points also shows increasing RMSE as R increases, but not as noticeably.

Figure 5.8b shows emulators of the difference, $\log\left(s_1\right) - \log\left(s_0\right)$. The standard emulators are emulators of the difference calculated from the training data, which showed to be the best standard method in Section 5.5. The reduced s_1 standard emulator is therefore built from the difference of log-output at each of the 100 points included with $R \neq 0$. The hierarchical emulator of the difference in this case, where there is only one hierarchical variable, is really built from the points for which a difference (or $\boldsymbol{\psi}$ data) is available, and therefore the hierarchical emulator built from reduced s_1 data is also built only from these 100 points.

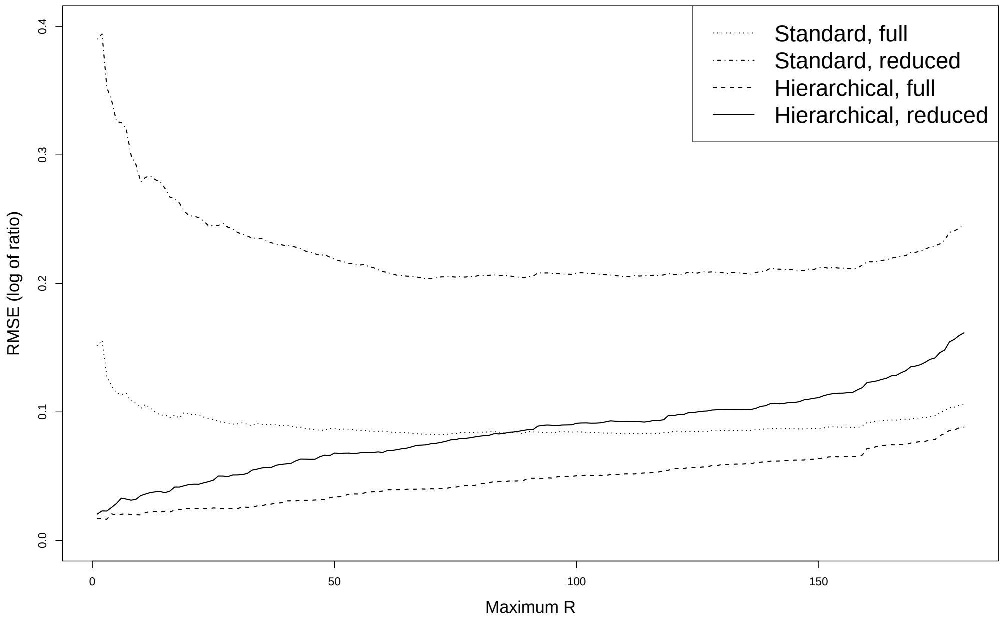
The advantage in the hierarchical emulation structure is much more marked with reduced s_1 data. The only extra information contained in the hierarchical emulator, compared to the standard emulator with reduced s_1 data (dot-dashed), is the inclusion of the relationship

$$s_1\left(x, R, m_2\right) = s_0\left(x\right) + g\left(R\right)\psi\left(x, R, m_2\right),$$

and it appears from Figure 5.8 that this is a valuable addition to the emulator. This supports the idea that when the more complex simulator is costly to run compared to the simpler one, hierarchical emulation is a very effective strategy.



(a) *Emulators of* $\log(s_1)$



(b) *Emulators of* $\log(s_1) - \log(s_0) = \log(s_1(\cdot) \,/\, s_0(\cdot))$.

Figure 5.8: *RMSE for four emulators, changing as the subset of data used changes. The prediction data was restricted to R less than 'Maximum R' for values from 1 to 180 (the maximum R takes in our experiment). The number of input points considered ranges from 5 (when* $R \leq 1$*) to 1,000 (when* $R \leq 180$*).*

5.6 Summary

In this chapter we have introduced hierarchical emulation, a method for emulation for two simulators where one is an extension of the other. These must have the property that when a small subset of the inputs, the *hierarchical inputs*, are set to particular values, the two simulators behave identically. Hierarchical emulation makes use of this relationship to emulate the more complicated simulator using a combination of other emulators, one of which is an emulator of the simpler simulator.

We have established a prior structure for the emulator that ensures separability between terms. The implications of this method on the design of experiments have been explored, and some criteria established for the structure of the training data. A hierarchical emulator requires some *transformation functions* $g(\cdot)$ for the hierarchical variables, and desirable properties for these have been explored.

In order to assess the performance of hierarchical emulation, a validation study was conducted using two versions of HadOCC. For this, a 1 million point staggered LHD was created (see Section 3.5.1) so that hierarchical and standard emulators could be compared with respect to various tasks. Overall, this showed hierarchical emulation to outperform the standard method, both in its predictive accuracy and its coherence with the emulation model. A further experiment to assess the performance of a hierarchical emulator built with a reduced amount of data from the more complicated simulator showed very promising results compared to the standard emulator, and reinforced our beliefs that including the relationship between the two simulators is beneficial. In Chapter 7, we present an object-oriented method for programming hierarchical emulation.

Although this makes some progress towards emulating multiple simulators, the set of situations in which it applies is still fairly restrictive. In Chapter 6, we introduce *intermediate variable* emulation, whose scope is much wider.

Chapter 6

Intermediate Variable Emulation

The methods introduced so far jointly emulate two simulators only when their input spaces are almost or entirely the same. However, this is not often the case, and so for a method to be useful for comparing simulators, it should not make this demand.

In Section 4.2.3, we noted that even when they do not share the same input space, two simulators of a particular system will often represent many of the same processes. The descriptions of HadOCC and OG99NPZD in Chapter 2 and in the examples of simulator differences in Section 4.2 reveal this to be true in their case.

In this chapter we introduce *intermediate variable emulation*, which takes advantage of this feature in order to emulate multiple simulators with different input spaces in such a way that they can be compared. The idea is related to a technique developed by Strong et al. (2012), where intermediate variables created in a similar way are used to help understand structural uncertainty in computer models.

6.1 Synopsis

The basic premise of intermediate variable emulation is that a simulator can be looked at with regard to three stages, each represented by sets of variables:

1. Input variables

2. Intermediate variables

3. Output variables.

The input variables are the numbers input by the user to run the simulator, such as those in Tables 2.1 and 2.3. The output variables are those that are returned by the simulator, which an emulator aims to predict. These will usually be some summary of the raw simulator output, for example the mean as in the example in Section 3.6, or a multivariate summary of a time series.

In standard emulation, as described in Chapter 3, an emulator takes in the input variables and produces a probability distribution for the values of the output variables. However, when dealing with several simulators of the same system, each with a different input space, this method makes comparison very difficult, as we discovered when emulating OG99NPZD and HadOCC in Section 3.6. Seldom can direct links between particular input variables be made, and the emulators cannot easily be used to make inferences about the different modelling choices made in each simulator.

Intermediate variables are a new construct, representing the states of some set of sub-processes in a simulator[1]. Simulator code often contains sub-modules which, as they run, produce variables other than the simulator output. These are usually invested with some system meaning, for example the death, growth or concentration of a particular species, or a transfer or flux that takes place. Two simulators of the same system may well contain some corresponding sub-modules.

Suppose two computer simulators of an ecosystem each have a process representing the death of a particular species. Although their input variables are different, making it impossible to identify points in each input space with one another, both are able to produce a time series of this species' mortality. These have a meaning that can be treated as the same, and so features of these time series can be compared. Producing a very similar time series from each model may enable us to identify links between portions of the input spaces, or to see how different the parameterisations of the process are in practice. The input variables can be used to emulate intermediate variables to better enable this task.

[1] For a particular simulator there may be more than one possible set of intermediate variables; they are not unique.

Further than this, we can list all these intermediate variables for each simulator in such a way that, at least in theory, if we know the values for all the intermediate variables we no longer need to know the values of the input variables to calculate the simulator output. Having done this, in principle we can use the values of the intermediate variables to build emulators of the simulators' outputs, since the relationship is deterministic.

Simulators of a particular system will usually have outputs that are linked in meaning. Often they track the population of a particular species, or the concentration of a chemical. In the case of HadOCC and OG99NPZD, MarMOT, which was introduced in Section 2.4, enables outputs to be produced that are assigned exactly the same system meaning.

In the intermediate variable emulation framework, the emulators built from the intermediate variables will have input (the intermediate variables) and output (the output variables) with the same physical meanings for each simulator. This means that the treatment of the sub-processes by the different simulators can be compared. As well as making inferences about the relationships between the input spaces and the intermediate variables, we can compare the relationships between the same intermediate and output variables for the different simulators. By emulating the intermediate variables from each simulator's input space, we can compare the effect of the input variables on a range of variables that covers each simulator's behaviour, since the intermediate variables separate the input from the output.

Emulating the intermediate variables from the inputs of each simulator is mainly informative for the relationships between the two input spaces. Emulating the output from the intermediate variables helps compare the behaviour of each simulator in terms of how the processes they model contribute to the overall representation of the system. Figure 6.1 shows the relationship between the variables.

In the rest of this chapter, we will investigate how these emulators, and the intermediate variables themselves, can be used to explore simulator differences. Each step in the process will be explained and summarised, and then illustrated using OG99NPZD and HadOCC, the simulators introduced in Chapter 2.

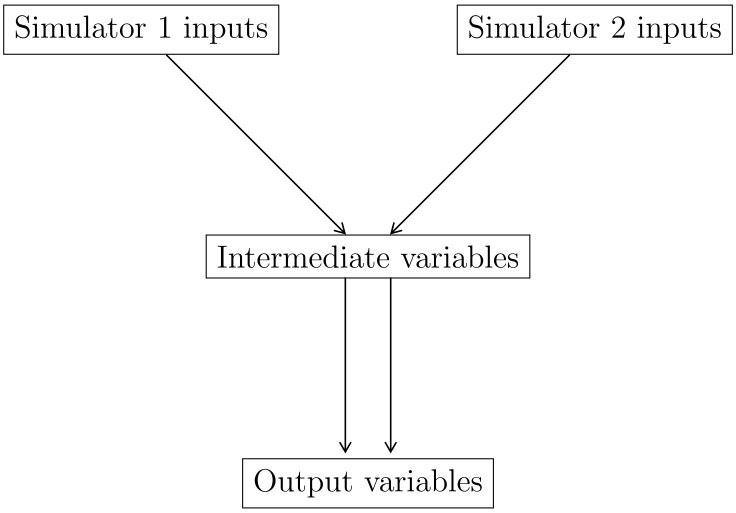


Figure 6.1: *A framework for intermediate variable emulation of two simulators. The quantities in the boxes represent variables, not exact values. Although the intermediate and output values for the different simulators will have different numerical values, in terms of system meaning they are the same.*

Intermediate variable emulation for a single simulator

As well enabling the comparison of different simulators, intermediate variable emulation is also a useful tool for understanding a single simulator. The intermediate variables allow one to probe into how the simulator is modelling the system, and learn more about how the inputs contribute to various aspects of the model. In Section 6.5 the intermediate variables will be used to refine the input region in a way that cannot necessarily be done using output variables. This is done using either observations of the intermediate variables, if available, or by ruling out inputs leading to unrealistic behaviour in some intermediate variables. This could prove especially useful for refining the input space in inputs that are not very active in the output, but are used in some intermediate variables.

Although the focus in this chapter is on the use of intermediate variable emulation for comparing two simulators, many parts of the method and analysis could be implemented for one simulator. Implementing the theory in the examples throughout this chapter will lead to a greater understanding of each simulator on its own, as well as to the relationship between them.

6.2 Adding intermediate variables

Whereas standard emulation requires no understanding of the simulator beyond its input and output variables, intermediate variable emulation demands some appreciation of the simulator's workings. This will involve careful study of the source code and any available literature, and liaising with experts of the system and the simulators where possible. The quantities chosen as intermediate variables will not necessarily be included as output to either simulator, and so the code may need to be adapted to incorporate them.

Simulators of complex systems usually produce very structured high dimensional output, often over both space and time. We will often restrict our interest to time series, but the method can be extended to spatial data. It will also be assumed throughout that the dimension of the output variables has already been reduced, or the variables summarised somehow if necessary. The technique used for doing this does not affect the methods described in this chapter.

For many pairs of simulators there will be more than one possible set of intermediate variables, and which works best may depend on the goal of emulation. It may be that one set of intermediate variables has a much simpler relationship to the input or output variables than the other, and might therefore add very little information. A good strategy is to look for common ground between the two simulators, for example processes given the same meanings, or concentrations of the same tracers. There are likely to be quantities calculated at each time step in the simulator, and these are often a good place to look.

For intermediate variable emulation methods to work, the intermediate variables must separate (or very nearly separate) the input variables from the output variables. That is, given the values of all the intermediate variables, no information about the input variables should add much information about the output variables. This requires a thorough understanding of the processes taking place in the simulator and of the output, gained through some combination of source code, documentation or expert advice. It is difficult to verify that this has been achieved, though if it hasn't, the emulators from intermediate to output variables will be severely wanting.

The key steps therefore in selecting intermediate variables are

- Work with simulator experts, firstly to ask their advice about possible quantities to be used as intermediate variables, and in the later stages to gain their feedback on the chosen quantities.
- Study papers and other documentation of the simulators, to gain further insight. In particular, if there are some equations governing the simulators, work out the primitive quantities in terms of which they are written.
- Work through the source code, to see how the quantities suggested for intermediate variables manifest themselves. Study the calculation of the output variable, and all the stages where input variables are used in its calculation.

These may well not be done in this order, and it is quite possible that the resources mentioned will not all be available.

6.2.1 Example: HadOCC and OG99NPZD

Here, we investigate one possible description of HadOCC and OG99NPZD in terms of intermediate variables, which will be used in the example sections throughout the rest of this chapter.

Both HadOCC and OG99NPZD are compartmental simulators, modelling flows of nitrogen between the same compartments. They are described more fully in Sections 2.2 and 2.3. In both simulators the biological models are governed by equations representing these nitrogen flows, such as the following ‘source minus sink’ equations used in OG99NPZD:

$$\begin{aligned}
\mathrm{sms}(P) &= \underbrace{\bar{J}\,(z,t,N)\,P}_{\text{growth}} - \underbrace{G\,(P)\,Z}_{\text{grazing}} - \underbrace{\left(\mu_P P + \mu_{PP} P^2\right)}_{\text{death}} \\
\mathrm{sms}(Z) &= \underbrace{\gamma_1 G(P) Z}_{\text{grazing}} - \underbrace{\gamma_2 Z}_{\text{excretion}} - \underbrace{\mu_Z Z^2}_{\text{mortality}} \\
\mathrm{sms}(D) &= \underbrace{(1-\gamma_1)\,G(P)Z}_{\text{unassimilated food}} + \underbrace{\mu_{PP}P^2}_{\text{dead P}} + \underbrace{\mu_Z Z^2}_{\text{dead Z}} - \underbrace{\mu_D D}_{\text{remineralisation}} - \underbrace{w_s \frac{\partial D}{\partial z}}_{\text{sinking}} \\
\mathrm{sms}(N) &= \underbrace{\mu_D D}_{\text{remineralisation}} + \underbrace{\gamma_2 Z}_{\text{excretion}} + \underbrace{\mu_P P}_{\text{dead P}} - \underbrace{\bar{J}\,(z,t,N)\,P}_{\text{P growth}}.
\end{aligned}$$

The representation in HadOCC is similar. At each time step, these equations are used to update the concentrations of each compartment. Quantities that can be monitored at each time step and have the same meaning in both simulators therefore include the concentrations of each tracer and the amounts of nitrogen transferred between each pair of compartments, and so these fit the requirements of intermediate variables.

Many of the outputs are fairly simply related to the tracers' concentrations; for instance, `pon` is the sum of phytoplankton, zooplankton and detritus, and `chl` is a multiple of phytoplankton. We therefore chose nitrogen transfers as our intermediate variables, in the hope that they would have a more interesting relationship to the simulators' outputs. Nitrogen transfer was also the quantity suggested by John Hemmings, an expert in HadOCC and OG99NPZD.

Transfer	Processes in HadOCC	Processes in OG99NPZD	Variable name
N to P	P growth	P growth	`iz.np`
P to N	P grazed, P respires, P mortality	P mortality	`iz.pn`
P to Z	P grazed	P grazed	`iz.pz`
P to D	P mortality, P grazed	P mortality, P grazed	`iz.pd`
Z to N	Z mortality, Z grazing by-products	Z excretion	`iz.zn`
Z to D	Z mortality	Z mortality	`iz.zd`
D to N	D remineralised, D grazed	D remineralised	`iz.dn`
D to Z	D grazed		`iz.dz`
D lost	D sinks	D sinks	`iz.ds`

Table 6.1: *Intermediate variables in terms of nitrogen transfer for HadOCC and OG99NPZD, and the system processes that contribute in each simulator.*

The nitrogen transfers at each time-step can be calculated by noting the subtractions and additions of nitrogen made to each compartment. The amount of nitrogen leaving nutrient and being added to phytoplankton in OG99NPZD, for instance, is

$$\bar{J}\,(z, t, N)\,P.$$

This transfer is associated with photosynthesis. The nitrogen transfers made at each time step are listed in full for HadOCC and OG99NPZD in Table 6.1, along with the physical processes each transfer represents in each simulator, and shown diagramatically in Figure 6.2. This formulation leads to nine intermediate variables in HadOCC and eight in OG99NPZD.

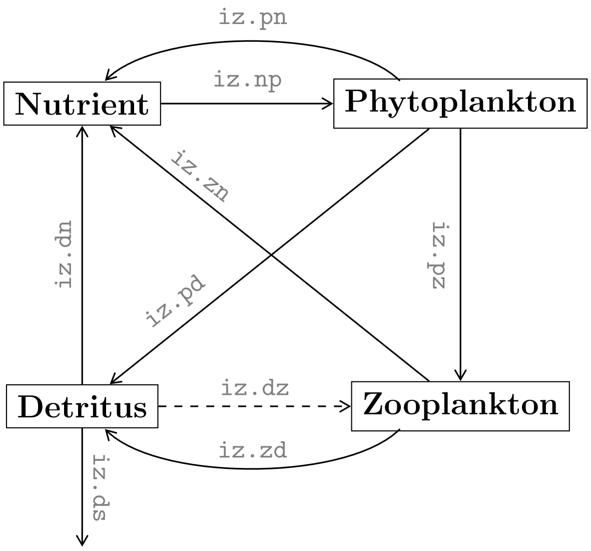


Figure 6.2: *Intermediate variables in HadOCC and OG99NPZD, matching those in Table 6.1. The arrows represent transfers of nitrogen, and therefore intermediate variables. The dashed line shows a transfer present only in HadOCC.*

Data

In order to be able to put intermediate variable methods into practice, four data sets were constructed. ‘OG100’ and ‘OG1000’ are both data from OG99NPZD, created using 100 and 1,000 point LHDs over OG99NPZD’s input space, respectively. It was checked that these were roughly orthogonal and had good space-filling properties. Similarly, ‘HAD100’ and ‘HAD1000’ are two datasets from HadOCC, containing 100 and 1,000 input points respectively, chosen by building LHDs over HadOCC’s input space. Each input point produces time series for each of the intermediate variables listed in Table 6.1 and a time series for any simulator outputs. The input ranges used were those given in Tables 2.1 and 2.3.

To illustrate this method, we choose depth-integrated particulate organic nitrogen (`iz.pon`) as the overall output for both simulators.

In both simulators, each intermediate variable is evaluated at each time and depth level of the model. This means that in its raw form, the data for each simulator contains 57 time series for each intermediate variable and input point, one for each depth-level. MarMOT enables depth-integration of any quantity, providing a spatial average. Although this loses some information, John Hemmings suggested that working with the depth-integrated nitrogen transfers would be likely to work well, thus reducing the data back to one time series per intermediate variable per input point for each simulator.

In keeping with the MarMOT naming convention, we will name the intermediate variables `iz.tran`, where `tran` represents the nitrogen transfer taking place. For example `iz.pn` represents the transfer of nitrogen from phytoplankton to nutrient. These names are shown in Table 6.1 and on Figure 6.2. The exception to this rule is `iz.ds`, which represents detrital sinking. From here on, the general term ‘intermediate variables’ applied to OG99NPZD or HadOCC refers to these depth integrated variables. In order to specify the simulator and time point, we will write

$$\texttt{iz.tran}^{\text{simulator}}_{\text{time}}$$

so that, for example, $\texttt{iz.np}^{\text{H}}_{24}$ is the depth-integrated nutrient to phytoplankton transfer for HadOCC at time 24.

6.3 Dimension reduction

A crucial part of the intermediate variable emulation process is choosing summaries of the intermediate variables to be used in the emulators built in Sections 6.5 and 6.6. Each input point produces intermediate variables with multiple values, often in a time series over several spatial regions. In order to be able to emulate using these variables, they must be summarised.

These summaries must contain enough information about the intermediate variables for the intermediate variable emulation model to capture the behaviour of the simulator adequately. They must also avoid being too highly correlated with one another, and must reduce the dimension of the intermediate variable space enough to make emulation possible. Ideally, they should be quantities that are simple enough to make any analysis fairly clear.

A simple method for summarising a time series, and the one we will employ, is to select a subset of values from each time series to represent the data. To select the most appropriate subset, we will use *principal variable* methods. There are various algorithms for selecting principal variables, and here we will use one proposed by Cumming and Wooff (2007).

The first principal variable is that for which the sum of the squares of correlations with each other variable is the largest, found using the correlation matrix R between the full collection of variables. This variable explains the highest proportion of the variance of the data.

The second principal variable is found in a similar way, using the unscaled partial correlation matrix of the remaining variables given the first principal variable. This process continues until some chosen threshold is reached. The full algorithm is presented in Cumming and Wooff (2007). At each step a variable is chosen that captures features of the variation not yet captured by the existing principal variables. For our purpose, there is a delicate balance between representing the data in enough detail and explaining enough variation on the one hand, and including too many variables, therefore inducing cripplingly high collinearity among the principal variables, and hindering analysis at the later stages on the other.

Popular ways of summarising time series often approximate the series by an analytic function. Examples include orthogonal polynomials (Narula, 1979), wavelets (Bayarri et al., 2007) and smoothing splines (Silverman, 1985; Wang, 1998). Ramsay and Silverman (1997) and Ramsay and Dalzell (1991) cover various methods for the analysis of functional data. Some of these techniques may also prove to be effective strategies for intermediate variable emulation, but principal variables have the advantage of being relatively easy to find, and make the analysis and comparison of time series a simple task.

Many of the curve-fitting techniques mentioned above are suited to different shapes of curves to various degrees, and may produce poor results if applied blindly. In using them for intermediate variable emulation it would be necessary to look at the general shape of each intermediate variable, and to consider exactly which curve-fitting method must be applied, and how. This is not necessary with principal variables. There are possible sorts of data, for example a constant time series with random noise added, for which principal variables will perform very poorly, but in most situations they are robust.

In order to find the principal variables for each intermediate variable, a stopping criterion must be chosen for the algorithm outlined by Cumming and Wooff (2007). We choose to stipulate a minimum proportion of total variation in the data that must be explained by the principal variables. This can be found, as shown in Cumming and Wooff, by

$$1 - \frac{\operatorname{tr}\left(\tilde{S}_{PV}\right)}{\operatorname{tr}(R)}, \tag{6.1}$$

where $\operatorname{tr}(\cdot)$ denotes the trace, R is the full correlation matrix of the data and $\tilde{S}_{PV}$ is the unscaled partial correlation matrix of the non-principal variables given the principal variables.

This stopping criterion enables us to ensure that the principal variables are capturing information. If the number of principal variables for each intermediate were specified instead, there would be no assurance of this. If too many were included, unnecessary degrees of collinearity would be introduced. If too few, then too little information would be retained. Stipulating the proportion of variance explained also ensures that in a situation where principal variables will do very badly, such as the extreme case of vectors of random noise, the algorithm will force many more principal variables to be selected that might be desirable, thus raising alarm.

By construction, a linear combination of principal variables can be used to almost perfectly reconstruct the full dataset. The $k_{\texttt{int}}$ principal variables used to represent a collection of n full time-series of intermediate variable `int` from simulator `sim` form an $(n \times k_{\texttt{int}})$ matrix $P^{\texttt{sim}}_{\texttt{int}}$. For each intermediate variable `int` and simulator

$\texttt{sim}$, a $(k_{\texttt{int}} \times t)$ (where t is the length of a full time series) transform matrix $T^{\texttt{sim}}_{\texttt{int}}$ can be found such that

$$P^{\texttt{sim}}_{\texttt{int}} T^{\texttt{sim}}_{\texttt{int}}$$

approximately re-forms the entire dataset. One method for finding the elements of $P^{\texttt{sim}}_{\texttt{int}}$ is to use the least squares coefficients, found by using the principal variables to predict each time point in the full dataset in turn.

Principal variables with two simulators

To find the principal variables using data from more than one simulator, there are several possible ways to proceed. For each intermediate variable shared by the simulators, the data can be pooled and the principal variables chosen which best represent the combined dataset. This makes for a simpler analysis, and is the option taken where possible in this chapter. Alternatively, the principal variables could be chosen that best represent each intermediate variable for each separate set of simulator data. A third option is to choose principal variables for each simulator, and then use the union of all these to represent each simulator's data, however this is likely to result in high degrees of collinearity. If the time series behave very differently for the different simulators, pooling the data to find principal variables may result in some features being missed. Methods using pooled data assume a roughly equal quantity of data for each simulator; if this is not the case, the method should be adapted.

Studying the behaviour of the intermediate variable data can provide insight into which of the above options is best. If the designs over each simulator's input space are well chosen, for example Latin hypercubes with good orthogonality and space-filling properties, they can be treated as representative samples from the input spaces. Therefore the resulting intermediate variables can be treated as representative samples from the intermediate variable space. With appropriate caution, they can be used to infer the behaviour of the populations of intermediate variables, and to compare the ranges and shapes of the different simulators. A simple way to do this is to plot a sample of the full time series for each intermediate variable and simulator.

The variances of the intermediate variables can also be compared as time progresses, to show which parts of each time series are more variable for the different simulators. For each simulator, the variance of each intermediate variable can be calculated at each time-step. These can then be used to find time-series of the ratio of the variances of the two simulators' values for each intermediate variable. Some examples of plots of these time series are shown in Figure 6.4. This helps to expose areas where one simulator may favour the placement of a principal variable, and the other may not.

Care must be taken in interpreting these variance ratios. If a variable has consistently higher variance in one simulator than another, it may be that it is inherently more varying in some sense in that simulator. It may be, however, that the input design is much more restricted in the other simulator, leading to much more constrained values. Because the input spaces are different, judging whether the two input designs are similarly restrictive is impossible.

After this stage, the principal variables will be used to represent both simulators' data, and so it is important to understand their relationship to the full datasets. If the transform matrices $T_{\mathtt{int}}^{\mathtt{sim}}$ are different for the two simulators, and therefore the relationship between the principal variables and the full set of intermediate variables is different for each simulator, then we must bear this in mind when treating the principal variables as comparable.

An effective strategy is to use the transform matrix for one simulator to reconstruct the full time series from the other, by

$$P_{\mathtt{int}}^{\mathtt{sim1}} T_{\mathtt{int}}^{\mathtt{sim2}}.$$

The errors from this reconstruction can then be compared to the errors found when using the principal variable matrix from the same simulator. If there is a large difference, this implies that caution should be taken when treating the principal variables as representing both simulators' data in the same way.

Finally, depending on the overall goal of the emulation study, a different choice of time points or summaries may be preferable. In this chapter, the main focus is on learning as much as possible about the comparisons to be made between two simulators, more or less in the absence of any system data. If observed data of some kind is available, for example of the output at a particular time point, then a better choice may be to choose time points of the intermediate variables that give the best prediction of this quantity. Rather than applying a technique such as principal variables, a better strategy might be step-wise model selection, or studying the code to work out any time dependencies between the quantities.

Method summary

The key steps for dimension reduction using principal variables are:

1. Plot samples of time series for each intermediate variable for each simulator, and variance ratio time series plots for each intermediate variable. If these reveal very different behaviours for the simulators, investigate further, and consider an alternative strategy to pooling both simulators' data to find principal variables.

2. Combine the different simulators' data for each intermediate variable and find the principal variables. Ensure that the stopping criterion, usually the proportion of variance explained, is achieved not just in the combined data, but also in the data for each simulator on its own.

3. Find a transform matrix for each intermediate variable and simulator that approximately reconstructs the full time series from the principal variables. For each intermediate variable, use the transform matrix from one simulator's data to reconstruct the data from the other. If this results in relatively large errors, note that the relationship between the principal variables and full time series differs between the simulators.

This could be considered part of the analysis of the intermediate variable data, and indeed it is crucial that the time series are summarised well, and that the relationship to the full dataset is understood, for any further inferences to be credible. In Section 6.4, methods for analysing the principal variable data are introduced.

6.3.1 Example: Dimension reduction

Step 1: Plotting time series data

The plots in Figure 6.3 (which spans three pages) show the hundred time series for each intermediate variable from OG100 and HAD100. The vertical lines show the principal variables, which will be found later. Similar plots for the larger datasets HAD1000 and OG1000 show no different behaviour, but are more difficult to interpret because of the density of the lines.

The two simulators appear to produce similar trends and distributions of values except for `log.iz.zd`, and a few extreme values of `iz.ds` in OG99NPZD and of `iz.pd` and `iz.dn` in HadOCC. The pattern of `iz.np` time-series appear mostly similar, except that the first half of the OG99NPZD series are much less smooth than the HadOCC series. Otherwise, the data are similar in terms of smoothness and general trend.

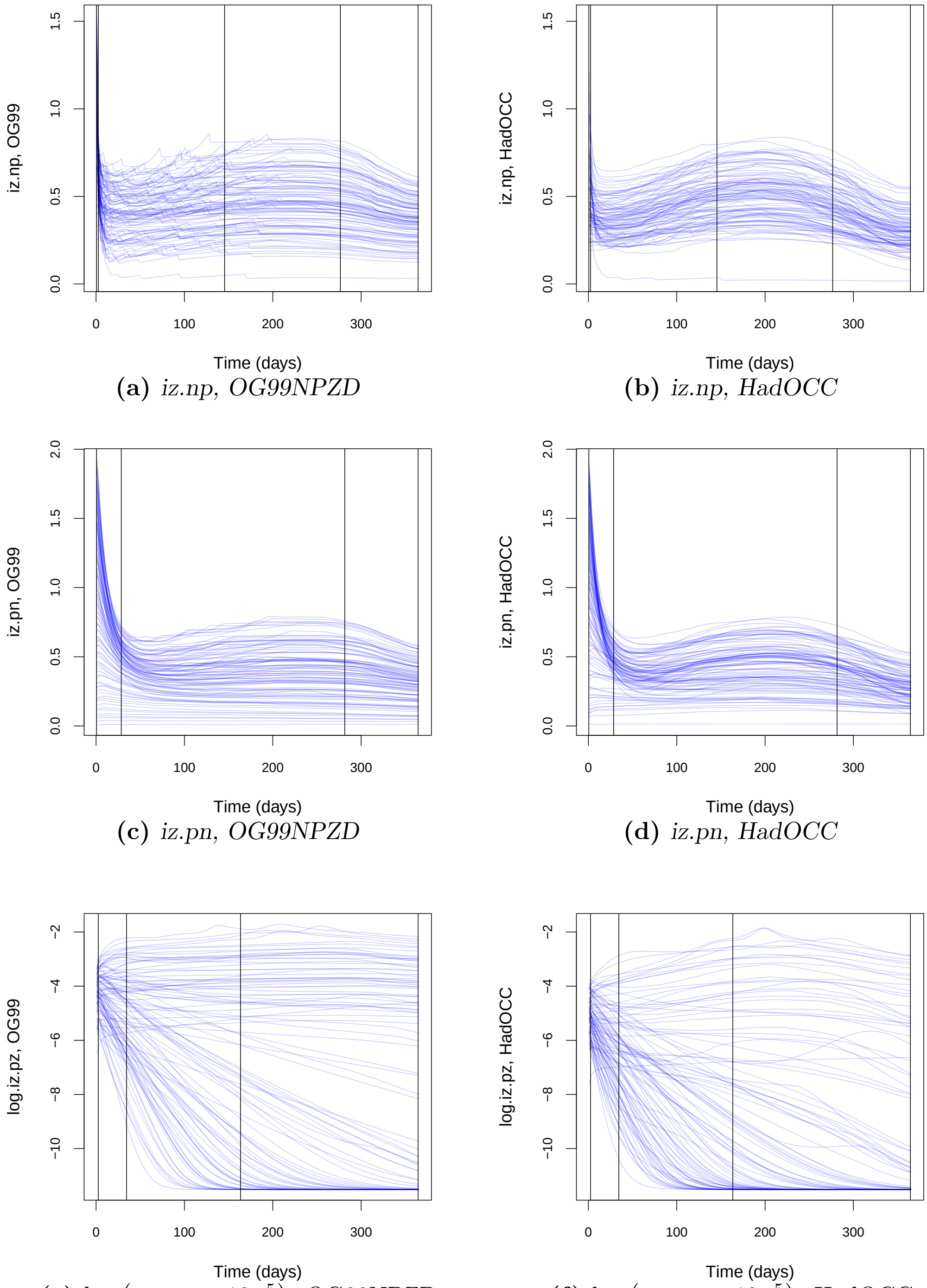
Further comparison can be made by plotting the ratio of the variances for the two simulators at each time point, as in Figure 6.4. For each intermediate variable,

$$\log_{10}\left[\frac{\mathrm{var}\left(\texttt{iz.tran}_t^{\mathrm{OG99NPZD}}\right)}{\mathrm{var}\left(\texttt{iz.tran}_t^{\mathrm{HadOCC}}\right)}\right]$$

is plotted against time. The horizontal red lines are at zero, and so anything above this line shows that at that time, the OG99NPZD data is more variable than the HadOCC data, and anything below the line shows the converse. OG99NPZD shows much higher variation for `log.iz.zd` and `iz.ds`, and also slightly higher at early and late times for `iz.np` and `iz.pn`.

There is no obvious pattern of higher variability in one simulator's data than in the other. The ratios mostly vary even within a single intermediate variable. This implies that rather than being due to the input ranges for the different simulators, at least some of the variability is inherent in their modelling of the system.

Keeping in mind these slight differences between HadOCC and OG99NPZD, we decided to combine the OG1000 and HAD1000 data to find principal variables, and to continue with the methods in Section 6.3.



(a) *iz.np, OG99NPZD*

(b) *iz.np, HadOCC*

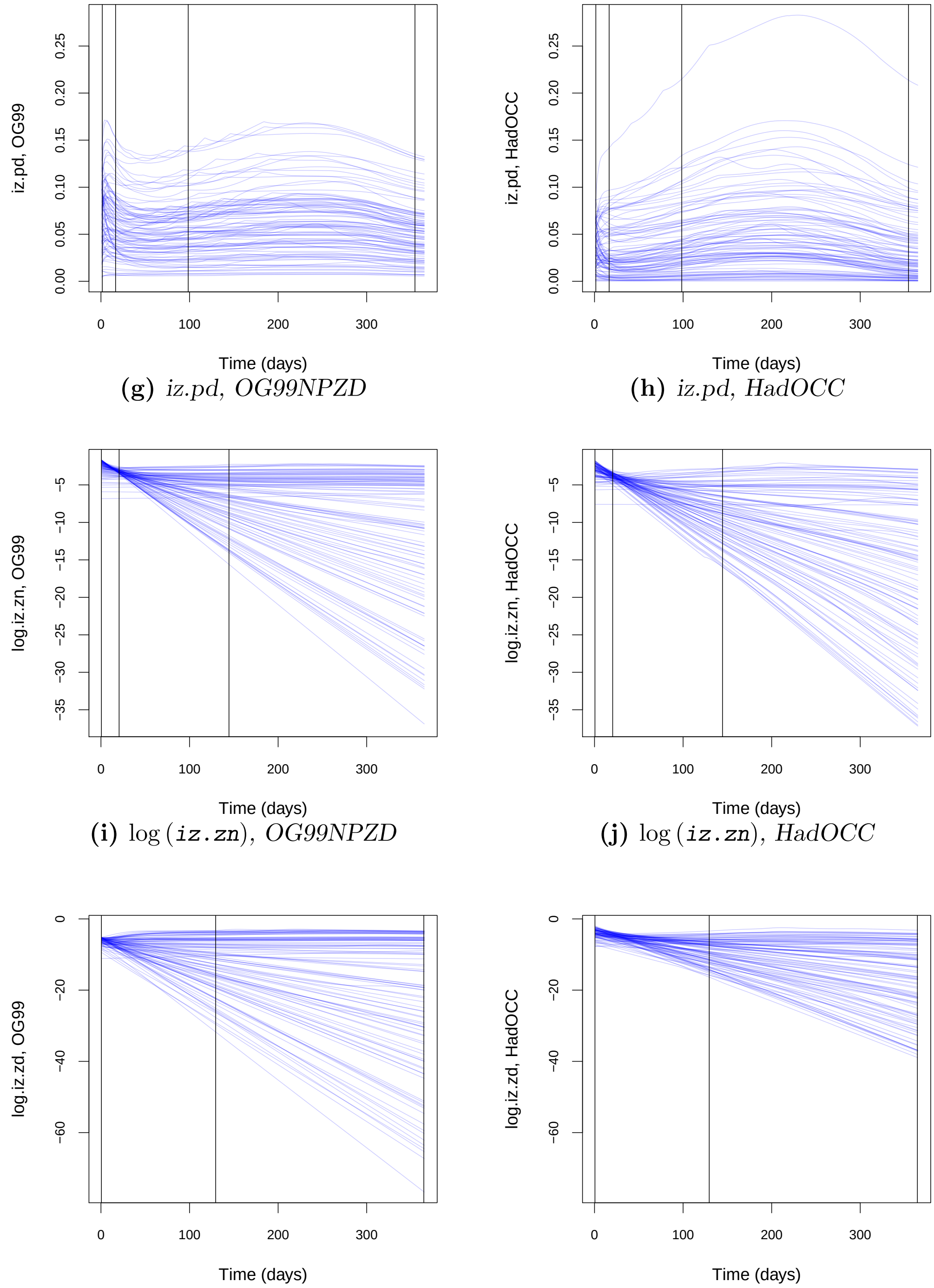
(c) *iz.pn, OG99NPZD*

(d) *iz.pn, HadOCC*

(e) $\log\left(\mathtt{iz.pz} + 10^{-5}\right)$, *OG99NPZD*

(f) $\log\left(\mathtt{iz.pz} + 10^{-5}\right)$, *HadOCC*

Figure 6.3: *Time series plots of intermediate variable output for OG100 (left) and HAD100 (right). Each plot covers a year. The zooplankton-related intermediates are transformed in both simulators, for reasons that will be explained later.*



(g) *iz.pd, OG99NPZD*

(h) *iz.pd, HadOCC*

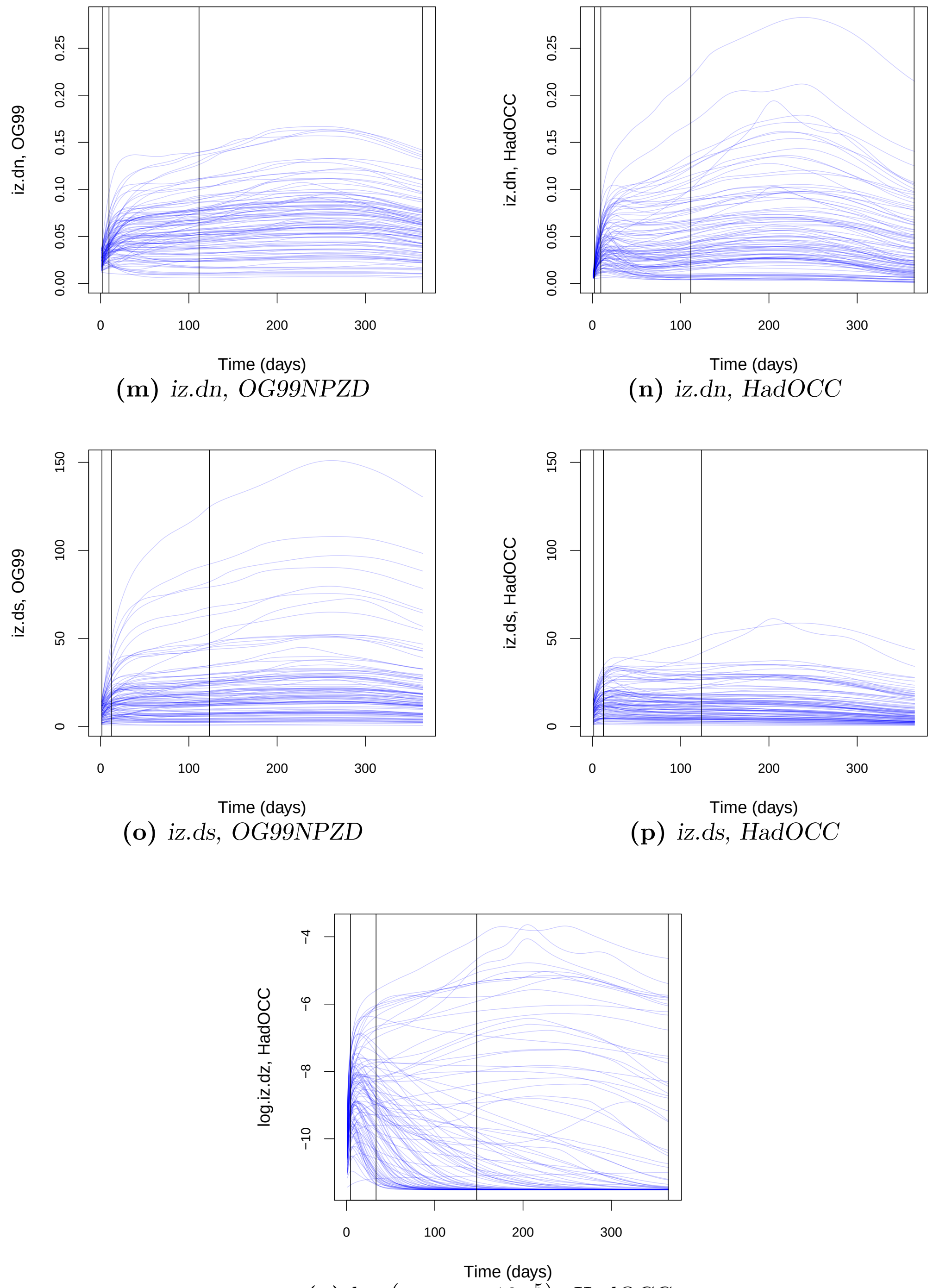
(i) log (**iz.zn**), *OG99NPZD*

(j) log (**iz.zn**), *HadOCC*

(k) log (**iz.zd**), *OG99NPZD*

(l) log (**iz.zd**), *HadOCC*

Figure 6.3: *Time series plots of intermediate variable output for OG100 (left) and HAD100 (right). Each plot covers a year. The zooplankton-related intermediates are transformed in both simulators, for reasons that will be explained later.*



(m) *iz.dn, OG99NPZD*

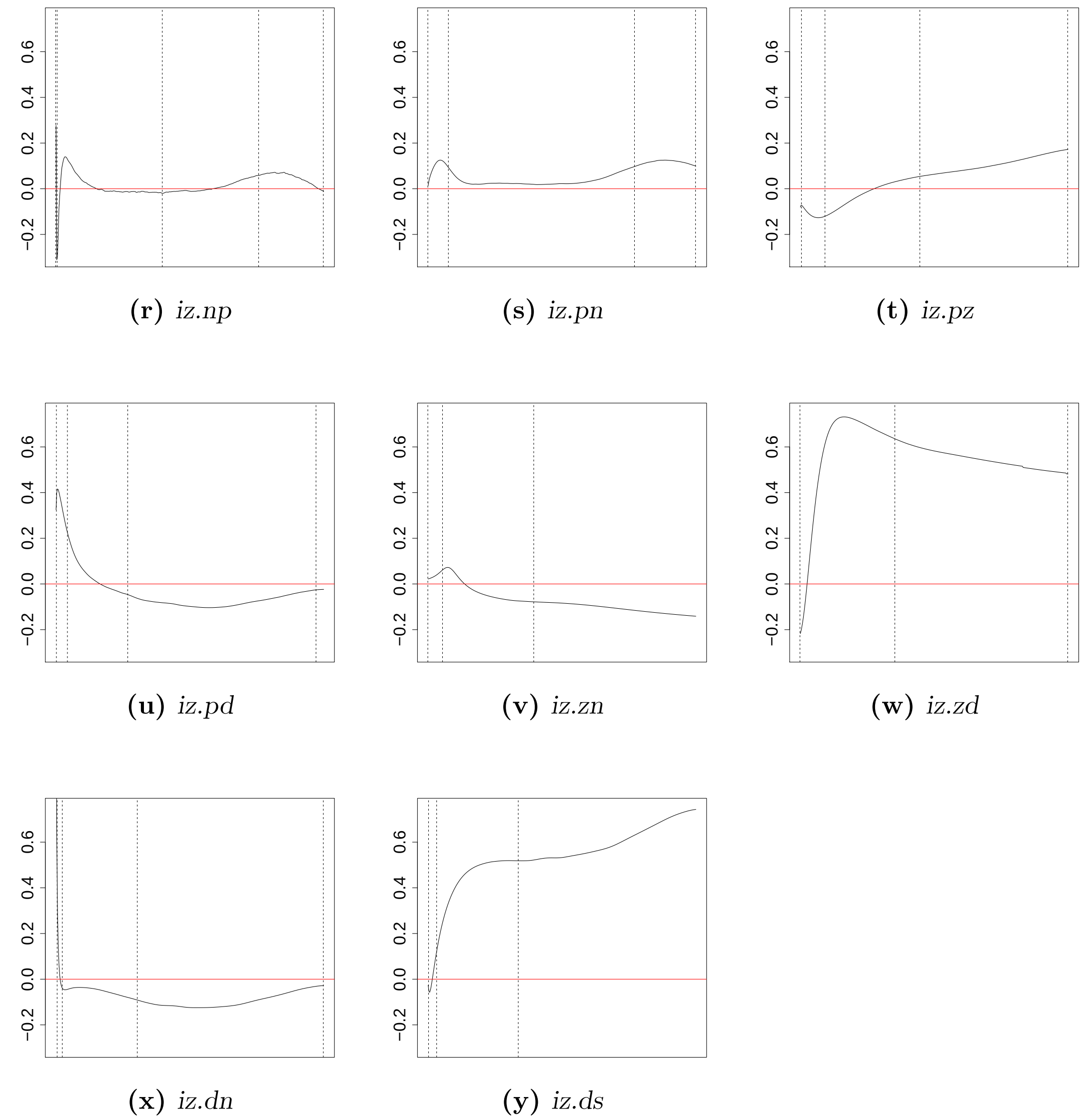
(n) *iz.dn, HadOCC*

(o) *iz.ds, OG99NPZD*

(p) *iz.ds, HadOCC*

(q) $\log\left(\texttt{iz.dz} + 10^{-5}\right)$, *HadOCC*

Figure 6.3: *Time series plots of intermediate variable output for OG100 (left) and HAD100 (right). Each plot covers a year. The zooplankton-related intermediates are transformed in both simulators, for reasons that will be explained later.*



(r) *iz.np*

(s) *iz.pn*

(t) *iz.pz*

(u) *iz.pd*

(v) *iz.zn*

(w) *iz.zd*

(x) *iz.dn*

(y) *iz.ds*

Figure 6.4: *Log variance-ratios for OG1000 and HAD1000. These are calculated by finding the variance of each intermediate variable at each time point for both simulators, then dividing the OG99NPZD variance by the HadOCC variance, and taking the base 10 logarithm of the ratio. In each plot the horizontal line is at 0, where both variances are equal. Values above 0 indicate that the OG99NPZD data is more variable, values below 0 that HadOCC is more variable. Principal variables are shown by vertical lines.*

Step 2: Finding the principal variables

Having plotted various features of the full time series, we can now reduce the dimensions by finding principal variables.

Intermediate variable	OG1000 & HAD1000 combined		
	Time points	OG1000 variance explained	HAD1000 variance explained
`iz.np`	1, 3, 146, 277, 365	0.994	0.993
`iz.pn`	1, 29, 282, 365	0.998	0.993
`log.iz.pz`	3, 35, 164, 365	0.998	0.995
`iz.pd`	2, 17, 99, 355	0.997	0.996
`log.iz.zn`	21, 145, *(1)*	0.994 *(0.998)*	0.988 *(0.993)*
`log.iz.zd`	1, 130 *(365)*	0.992 *(0.999)*	0.984 *(0.995)*
`iz.dn`	3, 10, 112, 365	0.996	0.995
`iz.ds`	2, 13, 124	0.997	0.992
`log.iz.dz`	5, 34, 148, 365		0.994

Table 6.2: *Principal variables chosen when 99% of variation is to be explained. The 'time points' column shows the time points for each intermediate variable that have been selected as principal variables. The dataset contains all the HAD1000 and OG1000 data, and has therefore 2,000 time series for each intermediate variable. The two 'variance explained' columns show how much of the variance of the individual data sets is explained when these principal variables are used, calculated using Equation 6.1. The bracketed numbers in italics show updated figures when more time points are added, so that at least 99% of variation in each dataset is explained by the principal variables found from the combined dataset.*

It is clear from the plots involving zooplankton in Figure 6.3 (sub-figures 6.3(e), (f), (i), (j), (k), (l) and (q)) that after a certain time, a high proportion of runs have intermediate variables with values of zero, or almost zero[2]. For this reason, we will deal with the logarithms of each of the intermediate variables involving zooplankton, which was recommended by the Box-Cox model selection procedure. Details of how this was applied are given in Appendix C.1, along with plots summarising the results. We will return to this feature of the data when applying the techniques in Section 6.5. These variables will be written `log.iz.tran`.

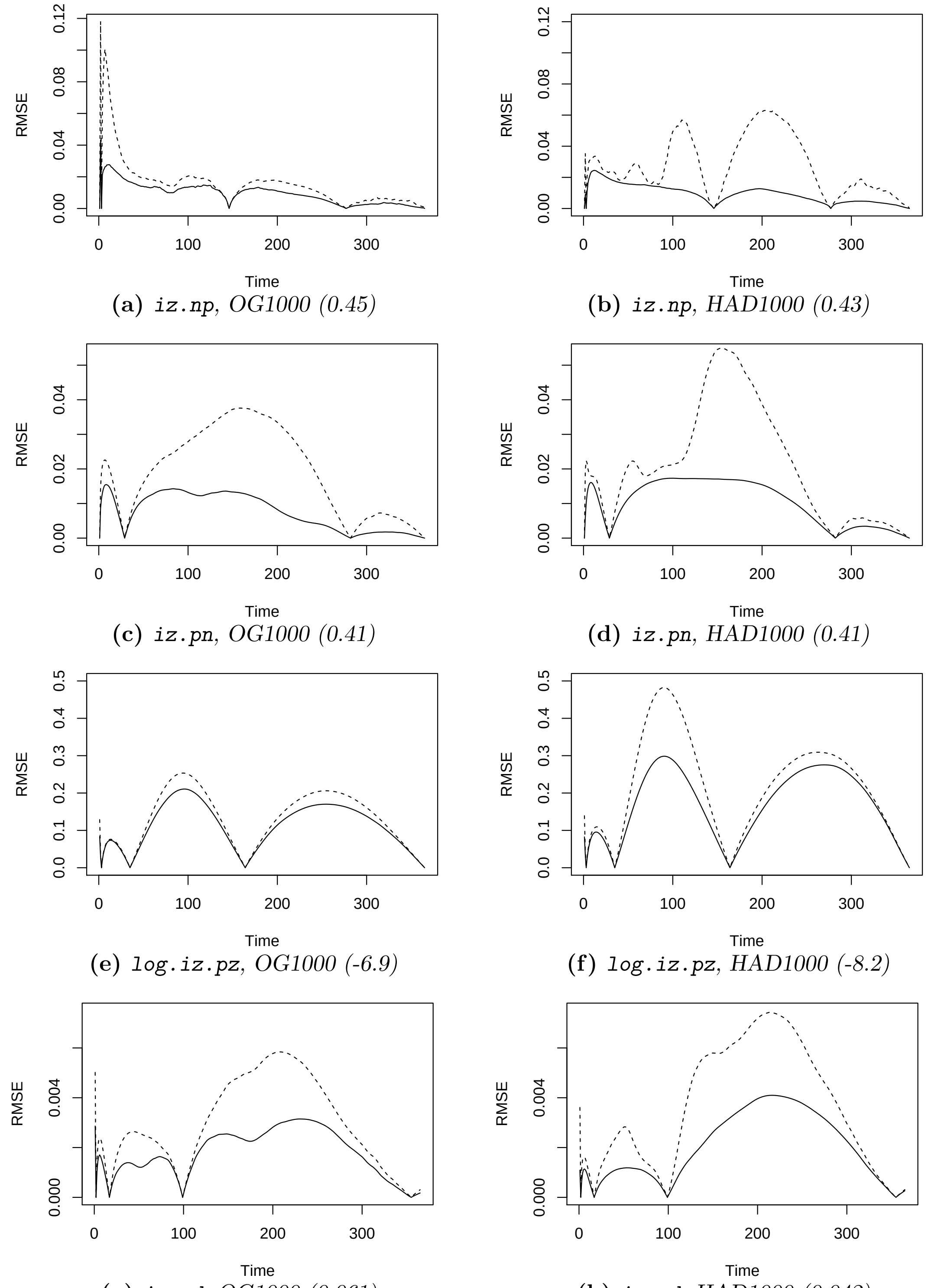
To find the principal variables for each intermediate variable, we must specify the proportion of variance they should explain (or a different stopping criterion) and the dataset from which they are to be found. Table 6.2 shows the time points selected for each intermediate variable such that 99% of the variance is explained for the combined OG1000 and HAD1000 data. These time points are also marked by vertical lines on the plots in Figures 6.3 and 6.4. For `log.iz.zn` and `log.iz.zd` a larger number of principal variables is required in HadOCC than in OG99NPZD to explain the same proportion of the variance. To address this, some more time points were added, so that at least 99% of the variability in both HAD1000 and OG1000 was accounted for by the principal variables. These extra time points are shown in italics and in brackets in Table 6.2

The emulators created in the stages described in Sections 6.5 and 6.6 can be used to give an indication of the amount of information retained through these stages. This can therefore show up possible problems in the selection of the intermediate variables and the dimension reduction stages. This is explained and illustrated with an example in Appendix C.2.

Step 3: Reconstructing the full datasets

The final stage in dimension reduction is to verify that the principal variables for each simulator relate to the full data in a similar way. We find $T^{\mathtt{OG}}_{\mathtt{int}}$ and $T^{\mathtt{HAD}}_{\mathtt{int}}$, the $k_{\mathtt{int}} \times 365$ transform matrices, for each intermediate variable. Using these, we can

[2] The values for `iz.pz` and `iz.dz` do reach zero, and so were transformed to $\mathtt{iz.tran} + 10^{-5}$ to enable us to use the Box-Cox procedure. The lines at roughly -11.5 therefore represent $\log\left(10^{-5}\right)$. The choice of value was fairly arbitrary, but 10^{-5} leads to a very conclusive Box-Cox transformation



(a) `iz.np`, *OG1000 (0.45)*

(b) `iz.np`, *HAD1000 (0.43)*

(c) `iz.pn`, *OG1000 (0.41)*

(d) `iz.pn`, *HAD1000 (0.41)*

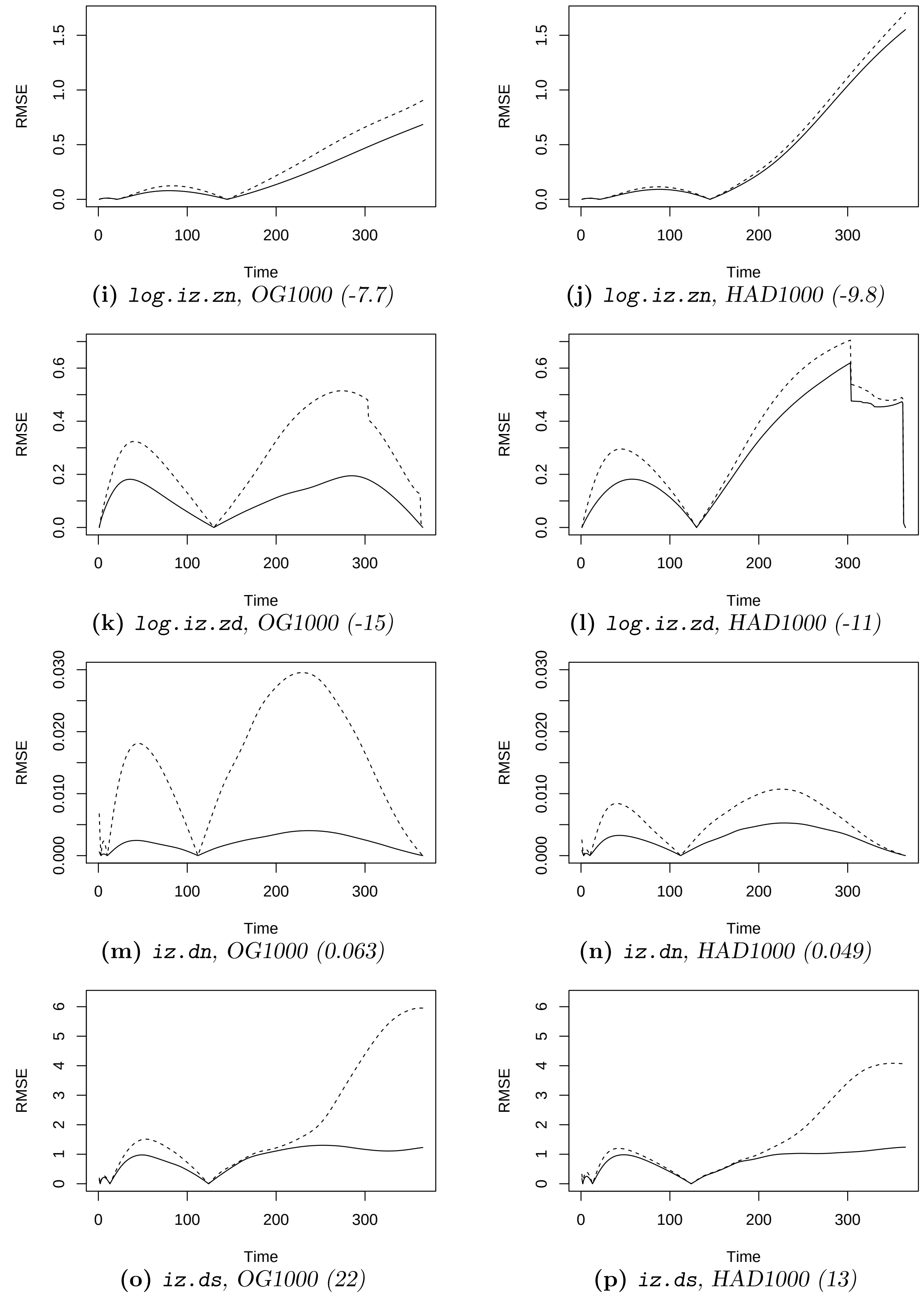
(e) `log.iz.pz`, *OG1000 (-6.9)*

(f) `log.iz.pz`, *HAD1000 (-8.2)*

(g) `iz.pd`, *OG1000 (0.061)*

(h) `iz.pd`, *HAD1000 (0.042)*

Figure 6.5: *Time series of RMSE from reconstructing the full intermediate variable data, using the correct transform matrix (solid line) and the matrix for the other dataset (dashed line). The mean of each intermediate variable is given in brackets.*



(i) `log.iz.zn`, *OG1000 (-7.7)*

(j) `log.iz.zn`, *HAD1000 (-9.8)*

(k) `log.iz.zd`, *OG1000 (-15)*

(l) `log.iz.zd`, *HAD1000 (-11)*

(m) `iz.dn`, *OG1000 (0.063)*

(n) `iz.dn`, *HAD1000 (0.049)*

(o) `iz.ds`, *OG1000 (22)*

(p) `iz.ds`, *HAD1000 (13)*

Figure 6.5: *Time series of RMSE from reconstructing the full intermediate variable data, using the correct transform matrix (solid line) and the matrix for the other dataset (dashed line). The mean of each intermediate variable is given in brackets.*

approximate the full time series from OG1000 by $P^{\mathtt{OG}}_{\mathtt{int}}T^{\mathtt{OG}}_{\mathtt{int}}$, or those from HAD1000 by $P^{\mathtt{HAD}}_{\mathtt{int}}T^{\mathtt{HAD}}_{\mathtt{int}}$.

In order to check that the data are reconstructed in a similar way, we compare the reconstructions of OG1000 given by $P^{\mathtt{OG}}_{\mathtt{int}}T^{\mathtt{OG}}_{\mathtt{int}}$ with those from $P^{\mathtt{OG}}_{\mathtt{int}}T^{\mathtt{HAD}}_{\mathtt{int}}$, and the reconstructions of HAD1000 given by $P^{\mathtt{HAD}}_{\mathtt{int}}T^{\mathtt{HAD}}_{\mathtt{int}}$ with those from $P^{\mathtt{HAD}}_{\mathtt{int}}T^{\mathtt{OG}}_{\mathtt{int}}$. Figure 6.5 shows the RMSE at each time point between the original data (OG1000 or HAD1000) and the data reconstructed from the principal variables. A solid line shows that the transform matrix used to reconstruct the data was found from the same data, and a dashed line shows that the transform matrix from the other simulator's data was used. The mean of each variables is given in the sub-caption in brackets.

For most of the intermediate variables, the reconstruction using the other simulator's transform matrix gives errors that are within a factor of two of that using the correct transform matrix. A notable exception is `iz.dn`, where the reconstruction of the OG1000 data using $T^{\mathtt{HAD}}_{\mathtt{dn}}$ is particularly poor.

6.4 Analysing intermediate variable data

Before being used to build any emulators, the reduced dimension intermediate variable data can be analysed to help compare the two simulators. Understanding the relationships between the intermediate variables is also crucial in the emulation stages described in Sections 6.5 and 6.6.

An indication of the correlation structure of the intermediate variable data can be found by drawing heat maps (graphical representations of matrices, where each square is coloured according to the corresponding value in the matrix) of the covariance matrices of the principal variables for the two simulators. This shows which intermediate variables are more correlated, both within their own principal variables and with other intermediate variables. However it does not take account of the behaviour of combinations of intermediate variables.

A more detailed study of the different behaviours can be made using principal component analysis (PCA). Comparing, for each simulator, the principal components across the reduced intermediate variable space will show which directions account for the most variation, and which the least.

To find the principal components for a set of intermediate variables, written here as $x_1, \ldots, x_p$, for which we have n observations, we first create an $n \times q$ matrix $\mathbf{X}$ containing the data, in a rescaled form explained below. This matrix is then used to find the $q \times q$ matrix $\mathbf{X}^T\mathbf{X}$, whose eigen decomposition can be found, such that

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\alpha_i} = \lambda_i\boldsymbol{\alpha_i}$$

for up to q eigenvectors $\boldsymbol{\alpha}_i$ (all normalised to have length 1) and eigenvalues λ_i. Assuming these are in decreasing order of λ_i, the first principal component $\boldsymbol{\alpha}_1$ gives the linear combination $\mathbf{X}\boldsymbol{\alpha}_1$ with the highest variance, and $\boldsymbol{\alpha}_q$ the linear combination with the least. If $\lambda_q = 0$ then $\mathbf{X}\boldsymbol{\alpha}_q$ represents a linear combination of intermediate variables that is constant throughout the data. Jolliffe (2002) gives a detailed introduction to PCA, and several extensions for specific scenarios.

PCA is sensitive to rescaling of the data. In this setting, if one intermediate variable has, in general, much larger values than another, and has a higher variance even though it is not in relative terms more variable, it will contribute heavily to the first few principal components. The data from both simulators is therefore combined, and rescaled so that each intermediate variable has mean zero and variance one. The rescaled combined data can then be split to give a rescaled dataset for each simulator. Each simulator's dataset is then re-centred to have mean zero, so that simulator bias does not overwhelm the analysis.

The principal components can expose combinations of variables that drive the variation. These may be different for different simulators. The eigenvectors corresponding to small eigenvalues can also help to expose linear combinations of intermediate variables that are highly correlated. If the largest few eigenvalues are much larger than the rest, this shows a subspace that is responsible for most of the variation in the data, and this could be explored.

The eigenvectors from the PCA can be used to compare the behaviour of the intermediate variables, by finding

$$\mathrm{var}\left[\mathbf{X}_{\texttt{sim1}}\boldsymbol{\alpha}_{\boldsymbol{i}}^{\texttt{sim1}}\right] \text{ and } \mathrm{var}\left[\mathbf{X}_{\texttt{sim2}}\boldsymbol{\alpha}_{\boldsymbol{i}}^{\texttt{sim1}}\right]$$

for each eigenvector $\boldsymbol{\alpha}_{\boldsymbol{i}}^{\texttt{sim1}}$ associated with the data from simulator sim1, where the (re-scaled and re-centred) data from simulator sim is written $\mathbf{X}_{\texttt{sim}}$. Significant differences in these variances indicate differences in the structure of the intermediate data.

Eigen decomposition can also be used to compare the variance matrices of the data for two simulators, in a similar fashion to that of Goldstein and Wooff (2007, chap. 9). If $\boldsymbol{\Sigma}_{\texttt{sim1}}$, $\boldsymbol{\Sigma}_{\texttt{sim2}}$ are the $q \times q$ variance matrices of the commonly rescaled intermediate variable data $\mathbf{X}_{\texttt{sim1}}$, $\mathbf{X}_{\texttt{sim2}}$ for two simulators, then the eigenvalues of $\boldsymbol{\Sigma}_{\texttt{sim1}}^{\mathbf{-1}}\boldsymbol{\Sigma}_{\texttt{sim2}}$ show up the degree of contrast between the datasets. Using the eigenvalues λ_i and eigenvectors $\boldsymbol{\alpha}_{\boldsymbol{i}}$ we have

$$\begin{array}{rrcl} & \boldsymbol{\Sigma}_{\texttt{sim1}}^{\mathbf{-1}}\boldsymbol{\Sigma}_{\texttt{sim2}}\boldsymbol{\alpha}_{\boldsymbol{i}} & = & \lambda_i\boldsymbol{\alpha}_{\boldsymbol{i}} \\ \Rightarrow & \boldsymbol{\Sigma}_{\texttt{sim2}}\boldsymbol{\alpha}_{\boldsymbol{i}} & = & \lambda_i\boldsymbol{\Sigma}_{\texttt{sim1}}\boldsymbol{\alpha}_{\boldsymbol{i}} \\ \Rightarrow & \boldsymbol{\alpha}_{\boldsymbol{i}}{}^T\boldsymbol{\Sigma}_{\texttt{sim2}}\boldsymbol{\alpha}_{\boldsymbol{i}} & = & \lambda_i\boldsymbol{\alpha}_{\boldsymbol{i}}{}^T\boldsymbol{\Sigma}_{\texttt{sim1}}\boldsymbol{\alpha}_{\boldsymbol{i}} \\ \Rightarrow & \operatorname{var}\left(\mathbf{X}_{\texttt{sim2}}\boldsymbol{\alpha}_i\right) & = & \lambda_i \operatorname{var}\left(\mathbf{X}_{\texttt{sim1}}\boldsymbol{\alpha}_i\right). \end{array} \tag{6.2}$$

If the columns of $\mathbf{X}_{\texttt{sim1}}$ and $\mathbf{X}_{\texttt{sim2}}$ represent the same principal variables therefore, this allows us to compare the variability of the linear combinations of intermediate variables given by the eigenvectors $\boldsymbol{\alpha}_i$. A large value $\lambda_i \gg 1$ shows that the linear combination is much more variable in the simulator 2 data than in the simulator 1 data, and a small value $\lambda_i \ll 1$ shows the converse.

These methods are purely data analytic, rather than employing emulation, but nevertheless can be very useful in comparing the behaviour of different simulators. They provide a useful framework for presenting an expert with information about the similarities and differences between two simulators. He may use these, and his knowledge of the system, to make judgements about each simulator's treatment of the system. Better understanding the structure of the intermediate variable data from each simulator, and the differences between them, will also help us when we come to build emulators involving the intermediate variables. This technique can also be used to compare the output variable data.

Method summary

This stage of the analysis can be summarised by the following steps:

4. Draw heat maps of the correlation matrices of the principal variables for each simulator, for a simple indication of the relationships between intermediate variables.

5. Use PCA on the principal variables for each simulator. Look at the eigenvalues, and compare the variances of linear combinations for the different simulators' data, using the eigenvectors from both datasets.

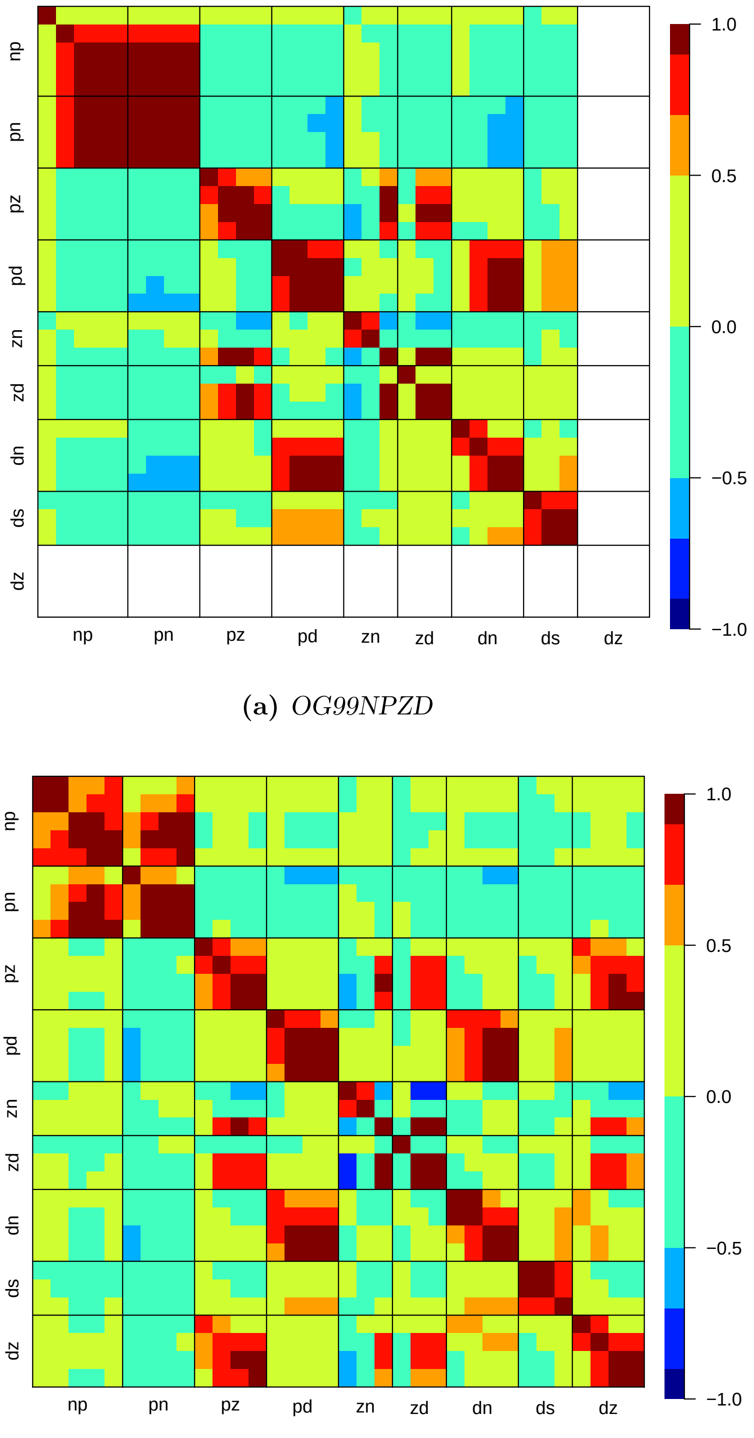
6. Using the Eigen decomposition of $\mathbf{\Sigma}_{\mathtt{sim1}}^{-1}\mathbf{\Sigma}_{\mathtt{sim2}}$, compare the variances of particular linear combinations. Plot the eigenvalues.

Having analysed the intermediate variable data itself, we can use emulation to analyse the relationships between those and the other sets of variables, using the framework shown in Figure 6.1. The findings of this section will be most useful in Section 6.6, where an understanding of intermediate variable space is crucial to our understanding of its relationship to the output.

6.4.1 Example: Analysing intermediate variable data

Having established in Section 6.3.1 that the principal variables found using the combined HAD1000 and OG1000 datasets represent both well, we will use the principal variables in Table 6.2 in this section, including the extra time points shown in italics, so that at least 99% of the variation in each dataset is captured. This produces two new datasets, OGPV99 and HADPV99, containing only the principal variable time points of the intermediate variables. This means that OGPV99 has 30 columns and HADPV99 has 34, and both have 1000 rows. HADPV99 has four extra columns to represent the variable `log.iz.dz`. This intermediate variable will mostly come into play when we investigate the relationship between intermediate variables and output in Section 6.6, but the analysis in this section will show the extent to which it contributes to the variability of the HadOCC data.

Step 4: Correlation matrices



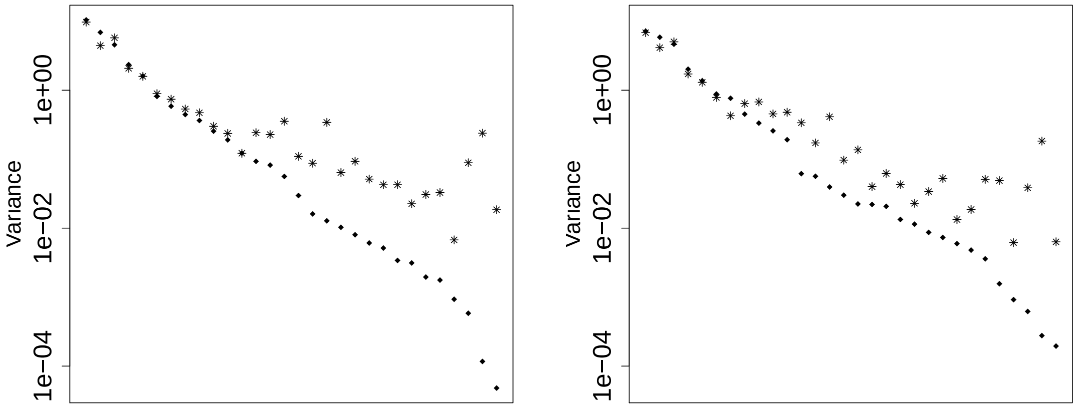
(a) *OG99NPZD*

(b) *HadOCC*

Figure 6.6: *Heat maps of correlation matrices for principal variables from OGPV99 and HADPV99. Each square represents a particular principal variable, and these are ordered by time (from left to right and from top to bottom).*

Heat maps of the correlation matrices for OG99NPZD and HadOCC's intermediate variables are shown in the plots in Figure 6.6. For both simulators these show very high levels of correlation between `iz.np` and `iz.pn`, and between `iz.pd` and later `iz.dn`. The later time points of the zooplankton related intermediates are also highly correlated, particularly in OG99NPZD. This is probably at least in part because of the tendency of these variables towards zero.

Step 5: Principal Component Analysis



(a) *Variances of principal components of* $\mathbf{X}_{OG}$ *over* $\mathbf{X}_{OG}$ *(squares) and* $\mathbf{X}_{HAD}$ *(stars).*

(b) *Variances of principal components of* $\mathbf{X}_{HAD}$ *over* $\mathbf{X}_{HAD}$ *(squares) and* $\mathbf{X}_{OG}$ *(stars).*

Figure 6.7: *Variances of principal components from* $\mathbf{X}_{OG}$ *(left) and* $\mathbf{X}_{HAD}$ *(right) applied to both datasets. The squares show the variances when the eigenvector is used on the data for which it is a principal component.*

To find the principal components of OGPV99 and HADPV99, the data had to be rescaled. The two datasets were therefore joined to form a 2,000 point dataset, and each principal variable of each intermediate variable was rescaled to have mean zero and variance one. This combined dataset was then split and the datasets for each simulator re-centred to have mean zero, so that the rescaled datasets, $\mathbf{X}_{\mathtt{OG}}$ and $\mathbf{X}_{\mathtt{HAD}}$, could be used separately. Each dataset had thirty columns[3], and therefore produced thirty eigenvectors, written $\boldsymbol{\alpha}_{\boldsymbol{i}}^{\mathtt{OG}}$ and $\boldsymbol{\alpha}_{\boldsymbol{i}}^{\mathtt{HAD}}$ for $i = 1, \ldots, 30$.

The variances of the linear combinations using both sets of principal components on both datasets are shown in Figure 6.7.

These plots show that, especially for the first few principal components, the variances are close for both simulators, implying that the most varying subspaces of OGPV99 and HADPV99 are oriented fairly similarly.

Step 6: Eigen decomposition of $\boldsymbol{\Sigma_1}^{-1}\boldsymbol{\Sigma_2}$

The variation in different directions within both datasets can be compared by studying the eigenstructure of

$$\boldsymbol{\Sigma}_{\mathrm{OG}}{}^{-1}\boldsymbol{\Sigma}_{\mathrm{HAD}},$$

where $\boldsymbol{\Sigma}_{\mathrm{OG}}$ and $\boldsymbol{\Sigma}_{\mathrm{HAD}}$ are the covariance matrices of the commonly rescaled data $\mathbf{X}_{\mathtt{OG}}$ and $\mathbf{X}_{\mathtt{HAD}}$. For this to work, the `log.iz.dz` data must be omitted from $\mathbf{X}_{\mathtt{HAD}}$.

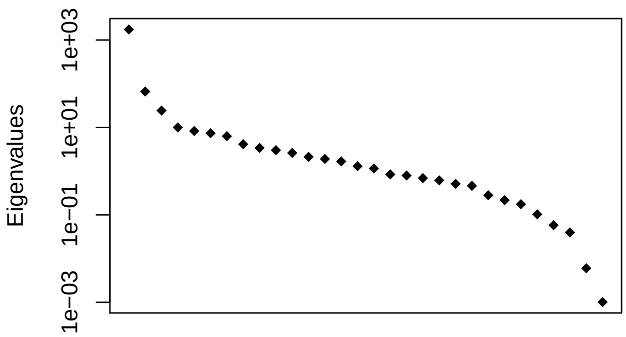


Figure 6.8: *Eigenvalues of* $\boldsymbol{\Sigma}_{OG}{}^{-1}\boldsymbol{\Sigma}_{HAD}$ *(log-scale)*

The eigenvectors with large corresponding eigenvalues represent linear combinations of variables that are much more variable in $\mathbf{X}_{\mathtt{HAD}}$ than in $\mathbf{X}_{\mathtt{OG}}$, and those with small eigenvalues represent the opposite. As shown in Equation 6.2, the eigenvalues are the factor by which the variance of that linear combination is larger in $\mathbf{X}_{\mathtt{HAD}}$ than

[3] In this example, the extra HadOCC variable `log.iz.dz` was not included for the PCA. When it is, it makes a large contribution to the first PC.

in $\mathbf{X}_{\texttt{OG}}$. The eigenvalues are shown in Figure 6.8, and the linear combinations with the two largest and two smallest eigenvalues are given in Table 6.3.

The main contributions in the largest eigenvalue's combination come from the first time point of `log.iz.zd`, which fits with Figures 6.3l and 6.4w, and in the opposite direction the first time point of `log.iz.zn`. The second largest is mainly affected by early and late time points of `iz.np` and `iz.pn`. These represent a trade-off between the two transfers as time progresses.

The very smallest eigenvalue is mainly linked to the earliest time points of `log.iz.zn` and `log.iz.zd`. The second smallest involves slightly more variables, but the main effect seems to come from `iz.dn` at times 10 and 112, in opposite directions. Figures 6.3m and 6.4x support the notion that this is much more variable in OGPV99 than in HADPV99.

Summary

This section shows that the simulators produce intermediate variables with mostly similar behaviour, shown by the patterns in Figures 6.3 and 6.6, and by the variances of the principal components' linear combinations applied to both datasets. In certain ways, however, the behaviour of the two simulators' intermediate variables seems to differ, as shown by the variance matrix comparison.

To an expert in the system, these may reflect underlying features of both simulators, and may already help with comparing them. In particular, the relationships between the different intermediate variables revealed by comparing the variance matrices may expose features of the simulators that do not reflect the system, or areas in which one simulator could be constrained. We will mainly use this analysis to help in the task of emulation, particularly when the intermediate variables are used as inputs.

6.5 Emulating intermediate variables

For each simulator, the input variables can be used to emulate the intermediate variables using methods from Chapter 3. Because intermediate variables are multi-

Intermediate	Time	Eigenvalue			
variable		1,742	66.4	6.01×10^{-3}	1.01×10^{-3}
`iz.np`	1	·	-0.44	·	·
	3	·	·	·	·
	146	·	·	·	·
	277	·	0.27	·	·
	365	·	·	·	·
`iz.pn`	1	·	0.61	-0.17	·
	29	·	·	·	·
	282	·	·	·	·
	365	·	-0.55	·	·
`log.iz.pz`	3	·	·	·	·
	35	·	·	·	·
	164	·	·	·	·
	365	-0.2	·	·	·
`iz.pd`	2	·	·	·	·
	17	·	·	0.21	·
	99	·	·	-0.34	·
	355	·	·	·	·
`log.iz.zn`	1	-0.6	·	·	0.45
	21	·	·	·	·
	145	0.23	·	·	·
`log.iz.zd`	1	0.68	·	0.14	-0.83
	130	-0.18	·	·	·
	365	·	·	·	·
`iz.dn`	3	·	·	·	·
	10	·	·	-0.7	·
	112	·	·	0.49	·
	365	·	·	·	·
`iz.ds`	2	·	·	·	·
	13	·	·	·	·
	124	·	·	·	·

Table 6.3: *Comparing the linear combinations with contrasting variances for* $\mathbf{X}_{OG}$ *and* $\mathbf{X}_{HAD}$*. For clarity, any value not contributing to the first 95% of the sum squares for its eigenvector has been replaced by a dot.*

variate, the model described in Section 3.2 is the one we will use.

We are able to build emulators from both input spaces to the same output space (the intermediate variables), in the belief that in the intermediate variables almost all of the behaviour of each simulator has been captured. Emulating the intermediate variables from the input variables can therefore further our understanding of each input space, and of links that can be drawn between them.

At this stage it is likely that a large number of quantities are being emulated, and this introduces some new choices concerning the structure of the emulator. A conventional approach in multivariate emulation, taken by Conti and OHagan (2010) for example, and assumed in Section 3.2, is to use the same regression functions for each output, and to estimate one correlation length for the whole set of data.

The intermediate variables, the outputs of the emulators at this stage, are organised into two tiers; the intermediate variables and the set of principal variables (or other reduced dimension summary) for each. This could therefore be viewed as one multivariate emulation problem, in which the same regression surface is used for every intermediate variable at each of its time points, as a set of multivariate emulation problems, one for each intermediate variable, or as a set of univariate emulation problems, one for each principal variable of each intermediate.

In this example, we have taken the second approach: each intermediate variable may have a different regression surface and correlation length, but within each intermediate variable, the principal variables are jointly emulated using the same surface and correlation lengths. It is hoped that this will help in identifying the effects of inputs on the different intermediate variables.

For each intermediate variable, the regression functions were chosen using stepwise model selection in R (R Development Core Team, 2011) for each time point, and including the union of all terms. To avoid the inclusion of too many terms, three dummy variables, containing randomly generated uniform values, were added onto the input data. Any interaction terms added after one of these was included was removed, and any linear term less active than one of these, and not involved in a remaining interaction, was also removed. In practice, this made very little difference in the examples shown here. Because the input design is approximately orthogonal, and each input is rescaled to cover the interval $[-1,\ 1]$ the coefficients should be simple to interpret.

Separating the behaviour of the simulator into particular intermediate processes allows us to inspect the effects of each input variable on particular parts of the simulator in closer detail. It may sometimes be that observed data is available for some intermediate variables, in which case *history matching* can be used. This method, explained in detail by Cumming and Goldstein (2010) or Vernon et al. (2010), combines historical system observations z with simulator output $s(x)$ to refine the input space $\mathcal{X}$, by ruling regions of input space out as 'implausible'.

It is more likely that some parts of the input spaces will lead to values of intermediate variables that we believe to be impossible. Unrealistic values of some input variables may not necessarily lead to unrealistic output, but they may lead to unrealistic values for some intermediate variables.

In this case, emulators can be used to rule out regions of input space leading to values of intermediate variable `int` outside a certain interval. The value of `int` at input point x is written $s_{\texttt{int}}(x)$, and the emulator's prediction $f_{\texttt{int}}(x)$.

Using the emulator, we can produce $(1-p_\alpha)$ credible intervals for the simulator's value of `int`, $s_{\texttt{int}}(x)$, at any input point x, by

$$\mathrm{E}\left(f_{\texttt{int}}(x)\right) \pm t_{(p_\alpha/2,\, df)}\, \mathrm{sd}\left(f(x)\right), \tag{6.3}$$

using the t_{df}-distribution resulting from the emulation model. If this interval is completely outside of $(U_{\texttt{int}}, L_{\texttt{int}})$, the input point should be rejected. For the conclusions to be meaningful, the exact values of $U_{\texttt{int}}$, $L_{\texttt{int}}$ and p_α should be chosen by experts, bearing in mind their beliefs about the simulator's discrepancy.

This technique can be used to create a new, refined input design of approximately size n for simulator `sim` in the following way:

1. Using a large dataset from `sim`, find the approximate fraction k_{sim} of inputs which lead to implausible behaviour

2. Generate a $(1-k_{sim})^{-1} \times n$ point input design for *sim*

3. Using an emulator, calculate $1-p_\alpha$ credible intervals for each intermediate variable over the input design

4. For each input point, if the interval is entirely outside $(L_{\texttt{int}}, U_{\texttt{int}})$, disregard the point. Otherwise, keep it.

The simulator can now be run over this refined input design to produce training data, and an emulator can be built in the same way as before. Because the input space is no longer a carefully chosen Latin hypercube, it will not necessarily have good orthogonality properties. This can be checked by studying the eigenstructures of the correlation matrices of the newly refined input designs. If the ratio of the largest to smallest eigenvalues is relatively close to one, the design is still fairly orthogonal. If the ratio is large, there is collinearity in the data, and this should be investigated. A heat-map of the correlation matrix is a good way to see quickly which inputs are highly correlated.

A fundamental difficulty in comparing different simulators, as mentioned in Chapter 4, is that their input spaces often do not correspond. Input variables from one simulator cannot automatically be linked to inputs in another, even though they may be assigned the same meaning in different simulators.

There are two ways in which inputs from different simulators can be compared. Firstly, in terms of the meaning they are assigned in each simulator. Inputs from different simulators can be given exactly the same meaning, or a completely different meaning. Somewhere between these extremes, inputs may have meanings that represent different aspects of the same process. Often, a process is represented by more parameters in one simulator than in the other. In the simpler version, the inputs may all be matched in meaning by inputs in the more complicated simulator, but the extra parameters contributing to the process may adjust the meaning slightly. This information can be learned from the simulators' code and documentation or from expert advice.

The other direction of comparison is the inputs' effect on the simulator's behaviour, in this case their effects on the intermediate variables. This is learned from studying the emulators, in particular the coefficients associated with the inputs in question, and those of any strong interaction terms present. For each simulator there is an emulator for each intermediate variable, with coefficients for each principal variable.

Because the intermediate variable values and input spaces are different, comparing coefficients numerically is difficult, and so we introduce the idea of a *relative coefficient*: every posterior coefficient value is divided by the largest coefficient relating to the same output, so that all range between plus and minus 1. For a particular intermediate variable, we may then compare the coefficients for each input for the different simulators, and more easily tell which behave similarly in terms of their proportional effect. As with the example in Chapter 3, a more thorough understanding of the influences of the input variables could be gained by using some of the techniques in the vast sensitivity analysis literature, for example Saltelli et al. (2000), but here we opt for simpler measures.

If two input variables, given the same meaning in each simulator, have a very similar effect on the intermediate variables, this supports the notion of linking them. If input variables that sound similar in fact affect intermediate variables rather differently, it may not be sensible to think of them as representing the same quantity. It may be that inputs that are active in one simulator for a particular intermediate variable have no apparently analogous variables in the other, in which case an expert may be able to make judgements about which simulator best captures that aspect of the system.

Having studied the behaviour of a particular intermediate variable, an expert might decide that in fact it does not behave as he should like. Having an emulator linking it to the input space may show whether this is a problem with the model itself, or with the numbers chosen for the inputs. This insight may help in adapting and improving the simulators to better capture the system, or in removing superfluous aspects of the simulator.

Method summary

The steps that should be taken in this analysis are:

7. Study the input spaces of each simulator. Identify the links in meaning between inputs, and groups of inputs that represent aspects of the same processes. Also identify inputs from one simulator that have no obvious equivalent or linked input in the other.

8. Build emulators of each intermediate variable for each simulator, using the methods in Chapter 3.

9. If there is any clearly unrealistic behaviour in the intermediate variables, use the emulators to refine the input space(s). Build emulators of the intermediate variables using the newly refined input space.

10. Using the coefficient estimates, calculate the relative coefficients for each emulator.

11. Compare the behaviour of the inputs for the different simulators, asking:
    - Do the inputs identified as similar, or linked, behave similarly?
    - Are the inputs unique to one simulator important? If so, how?
    - Where a process has more inputs in one simulator than another, do all these inputs seem to be active?
    - Are there any pairs of inputs that are given unrelated meanings but behave similarly?

6.5.1 Example: Emulating intermediate variables

In this example, the output variable `iz.pon` (depth-integrated particulate organic nitrogen) was also summarised using principal variables, using the same techniques as those used in Section 6.3 for the intermediate variables. The time points chosen were 1, 13, 105 and 365.

Step 7: Studying the input spaces

The first stage in this process is to understand the input spaces. Having studied the inputs of OG99NPZD and HadOCC in Chapter 2 and in the examples in Chapter 4, we are in a good position to summarise the relationships between the input parameters of OG99NPZD and HadOCC. Tables 2.1 (page 9) and 2.3 (page 13) list the input parameters for OG99NPZD and HadOCC. These show that whereas some are attributed the same meanings and units in each simulator, some input parameters in one simulator have no obvious analogue in the other.

OG99NPZD input	HadOCC input	Meaning
γ_1	`betap`	Zooplankton assimilation efficiency for phytoplankton
α	`alphachl`	Initial slope of photosynthesis v irradiance curve (mg C (mg Chl)$^{-1}$ (E m^{-2})$^{-1}$)
ϵ	`epsfood`	Prey capture rate (d^{-1}(mmol N m^{-3})$^{-n}$)
g	`gmax`	Maximum grazing rate (d^{-1})
C_{pp}	`rchlpig`	Ratio of chlorophyll to total pigment
PAR	`rparsol`	Ratio of PAR to total downwelling solar irradiance at sea surface
μ_{PP}	`pmortdd`	Conc. dependent phytoplankton specific mortality (d^{-1}(mmol N m^{-3})$^{-1}$)
μ_{ZZ}	`zmortdd`	Conc. dependent zooplankton specific mortality (d^{-1}(mmol N m^{-3})$^{-1}$)
K_1	`kdin`	Half-saturation conc. for nutrient uptake (mmol N m^{-3})
w_s	`dsink`	Detrital sinking velocity (m day^{-1})

Table 6.4: *Input parameters given the same meaning in OG99NPZD and HadOCC.*

OG99NPZD inputs	HadOCC inputs	Process
a, c	`photmax`	Photosynthetic rate
μ_P, μ_{PP}	`pmortdd, pminmort, fpmortdin`	Phytoplankton mortality
μ_{ZZ}	`zmort, zmortdd, fzmortdin`	Zooplankton mortality
γ_1, ϵ, g, γ_2	`betap, betad, epsfood, gmax, fmingraz, fingest, fmessyd`	Zooplankton grazing

Table 6.5: *Groups of input parameters representing aspects of the same process.*

Tables 6.4, 6.5, 6.6 and 6.7 show which of the input parameters for OG99NPZD and HadOCC are directly linked in meaning, which belong to groups of linked parameters, and which inputs to one simulator appear to have no linked inputs in the other. Some inputs appear in more than one table. This is because while two input parameters may be given exactly the same meaning in both simulators, for example the concentration dependent phytoplankton mortality rates μ_{PP} (in OG99NPZD) and `pmortdd` (in HadOCC), the extent to which we expect them to behave similarly may be mitigated by other parameters relating to the same process.

OG99NPZD input	Description
γ_2	Excretion rate (days^{-1})
μ_P	Specific phytoplankton mortality rate (day^{-1})
μ_D	Remineralisation rate (day^{-1})
a	Maximum photosynthetic rate at temp = 0 ° C (day^{-1})
c	Max. photosynthesis - variation of temperature factor exponent ((° C)$^{-1}$)

Table 6.6: *Input variables in OG99NPZD that have no obvious equivalent in HadOCC.*

HadOCC input	Description
`rcchl`	C:Chl ratio (if fixed) (mgC/mgChl)
`rcnphy`	C:N ratio for phytoplankton
`rcnzoo`	C:N ratio for zooplankton
`rcndet`	C:N ratio for detritus
`presp`	Phytoplankton specific respiration (d^{-1})
`pminmort`	Threshold for phytoplankton mortality (mmol N m^{-3})
`fpmortdin`	Fraction of phytoplankton mortality to DIN
`fmingraz`	Food threshold for grazing function (mmol N m^{-3})
`fingest`	Fraction of grazed material ingested
`betad`	Zooplankton assimilation efficiency for detritus
`fmessyd`	Fraction of messy feeding to detritus
`zmort`	Base zooplankton specific mortality (d^{-1})
`fzmortdin`	Fraction of zooplankton mortality to DIN
`rco3pprod`	Carbonate precipitated per unit primary production
`nitrifeuph`	Nitrification rate of ammonium in euphotic zone (d^{-1})
`nitrifaph`	Nitrification rate of ammonium below euphotic zone (d^{-1})

Table 6.7: *Input variables in HadOCC that have no obvious equivalent in OG99NPZD.*

The groups of parameters linked to phytoplankton mortality, which include μ_{PP} and `pmortdd`, are shown in Table 6.5. This shows that μ_P is a constant phytoplankton mortality rate in OG99NPZD, and that in HadOCC there is a threshold `pminmort`, below which no phytoplankton dies, and a fraction `fpmortdin`, determining the path of the nitrogen from dead phytoplankton. These parameters, which reflect aspects of the phytoplankton mortality process unique to one simulator, are also shown in Tables 6.6 and 6.7.

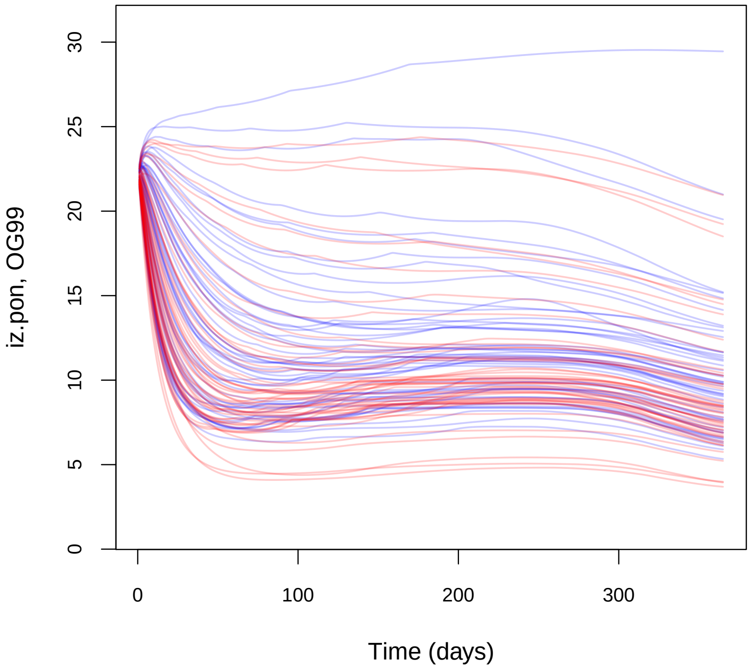
Steps 8 & 9: Emulating, and dealing with any unrealistic behaviour

The first emulators to be built used the input and intermediate variables in OGPV99 and HADPV99 as training data, each of which has 1,000 input points.

In an attempt to be able to use the emulators to understand the most active variables and pairwise interactions in each simulator's representation of the intermediate variables, a second order surface was chosen for each intermediate variable using stepwise model selection with the Bayesian information criterion (BIC), and one correlation length was estimated per intermediate variable. This may present problems if the different principal variables within a particular intermediate variable behave very differently, so we will monitor the performance of these emulators.

One of the purposes of this step is to use unrealistic values of the intermediate variables to rule out portions of input space. From the plots in Figure 6.3, it is clear that any nitrogen transfer involving zooplankton can very quickly go to zero and remain there (see Figure 6.3(e), (f), (i), (j), (k), (l) and (q), where each has a bold line at a low value after a certain time, representing a large number of runs). This feature has already manifested itself numerically in strongly supporting a log transform for these intermediate variables. This implies that significant proportions of each input space lead to zooplankton becoming extinct. John Hemmings, our expert, told us that because this is not the case in the real world, any input points leading to zooplankton transfers tending to zero should be ruled out if possible.

Without studying these intermediate variables, this behaviour would not necessarily have been observed. While these simulator runs have highly unrealistic zooplankton related intermediates, their output values are indistinguishable from those with more realistic zooplankton behaviour. Figure 6.9 shows `iz.pon` time series from OG100 and HAD100, with those for which zooplankton becomes extinct (in that at least one zooplankton related transfer is below 10^{-8} by the final time step) shown in red, and those for which it doesn't in blue.



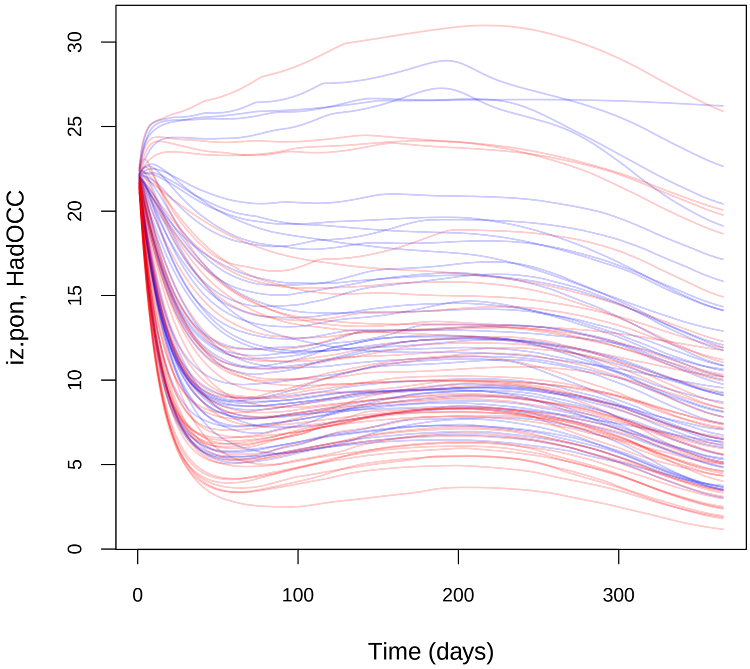
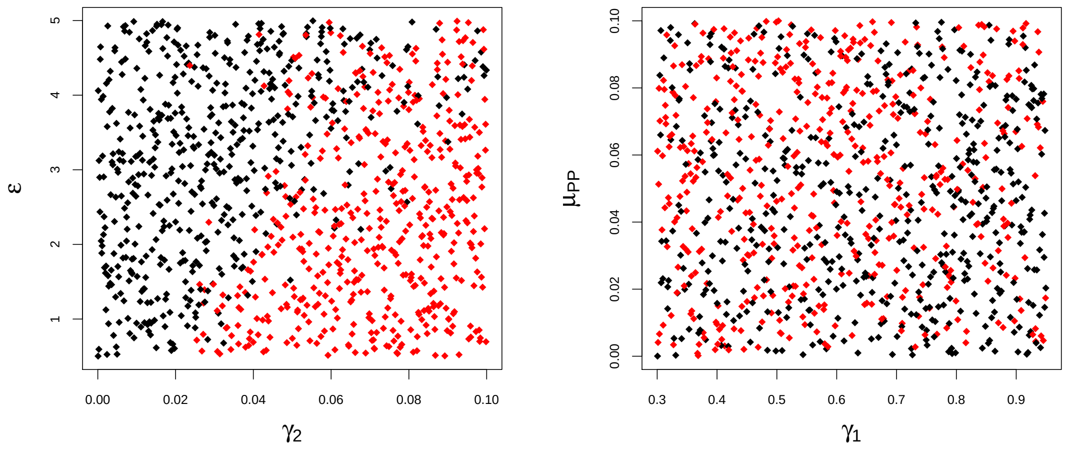


Figure 6.9: *Time series of* `iz.pon` *for OG99NPZD (top) and HadOCC (bottom) for OG100 and HAD100, coloured according to whether zooplankton appears to become extinct (red) or not (blue).*

Figure 6.11b shows a very slight trend for zooplankton extinction to arise away from high values of `presp` and low values of `epsfood`. Therefore, even were we to have access to accurate observations of `iz.pon`, they would not enable us to rule out (simply) the portion of input space leading to zooplankton extinction. It is worth pointing out that numerically, both HadOCC and OG99NPZD are unable to produce zero for some of the intermediates relating to zooplankton, and so in fact we are concerned with sufficiently small outputs, rather than zero outputs. How small is too small is for an expert to decide.

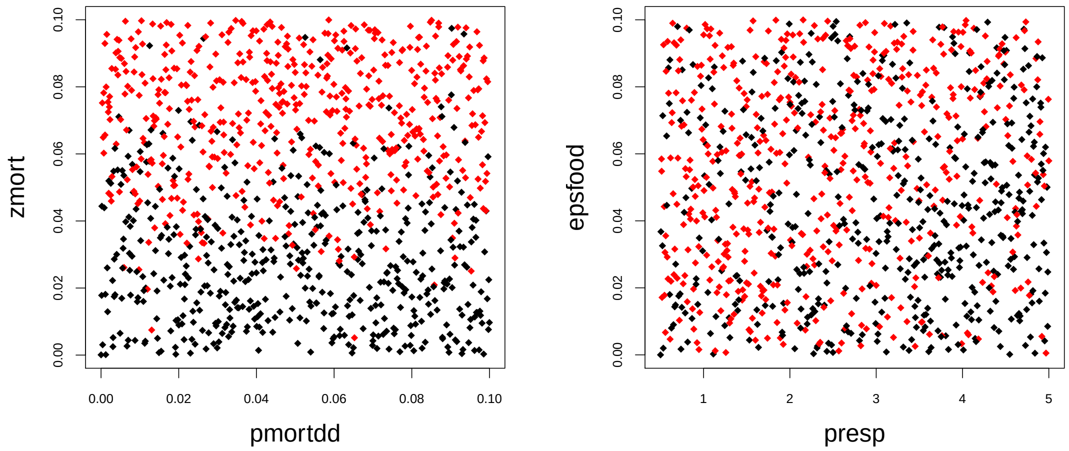
The plots in Figures 6.10 and 6.11 were drawn using OGPV99 and HADPV99, and show points leading to at least one zooplankton related intermediate below 10^{-8} in red, and others in black. These indicate that large portions of space, at least in some dimensions, should be ruled out by this procedure. In OG99NPZD, it seems the main area of zooplankton extinction contains points with high γ_2 and low ϵ, as shown in Figure 6.10a. The pairs plots involving other variables mostly show very little pattern, as in Figure 6.10b, although even here there appears to be a higher density of 'good' points for high γ_1 and low μ_{PP}.



(a)

(b)

Figure 6.10: *Pairs of inputs from OGPV99 plotted against one another, coloured according to the latest time points of the zooplankton related intermediates. If any is below* 10^{-8}*, the point is red, otherwise it is black.*



(a)

(b)

Figure 6.11: *Pairs of inputs from HADPV99 plotted against one another, coloured according to the latest time points of the zooplankton related intermediates. If any is below* 10^{-8}*, the point is red, otherwise it is black.*

The most influential input variable in HadOCC is `zmort`, as shown in Figure 6.11a, with large values more likely to lead to zooplankton extinction. There are interaction effects with `epsfood` and `pmortdd`, in opposite directions, but these are weak.

Using the strategy outlined earlier in this section, the emulators enable us to rule out portions of both input spaces leading to undesirable values of zooplankton-related intermediate variables. Although in reality the values of p_α (the confidence level of the interval) and $L_{\texttt{int}}$ (the lower limit of the acceptable interval for intermediate variable `int`) should be chosen carefully by an expert, for the purposes of this example we chose $p_\alpha = 0.05$, $L_{\texttt{int}} = 10^{-10}$ (for OG99NPZD) and $L_{\texttt{int}} = 10^{-6}$ (for HadOCC) for each of the zooplankton-related intermediates. The different values of $L_{\texttt{int}}$ ruled out roughly the desired fractions of the input spaces. The need for different values for HadOCC and OG99NPZD is most likely an artefact of the higher dimension of HadOCC's input space, which leads to lower precision in an emulator built with the same amount of training data. Using the same value of $L_{\texttt{int}}$ would therefore lead much less of HadOCC's input space to be ruled out than

OG99NPZD's, because the credible intervals are wider. Because the focus is on values that are too low, $U_{\texttt{int}}$ is effectively infinite.

In order to follow the method, a time point must be chosen at which each intermediate variable is studied. Because the zooplankton-related variables are almost entirely decreasing, we will use the latest principal variable for each one.

In order to create new input spaces, estimates for the proportions k_{OG} and k_{HAD} of the current input regions leading to unrealistic behaviour must be found. OGPV99 and HADPV99 were used, and k_{sim} set to be the proportion of input space leading to at least one zooplankton related intermediate with a value of less than 10^{-8} at its latest time step. This gave $k_{OG} = 0.478$ and $k_{HAD} = 0.512$.

To generate new input designs containing roughly 1,000 points, we therefore included $1,916$ points in the initial design for OG99NPZD, and $2,049$ for HadOCC. The emulators built from the unrefined datasets OGPV99 and HADPV99 were run over these input designs, and used to create 95% credible intervals for each intermediate variable at each input point, using equation 6.3.

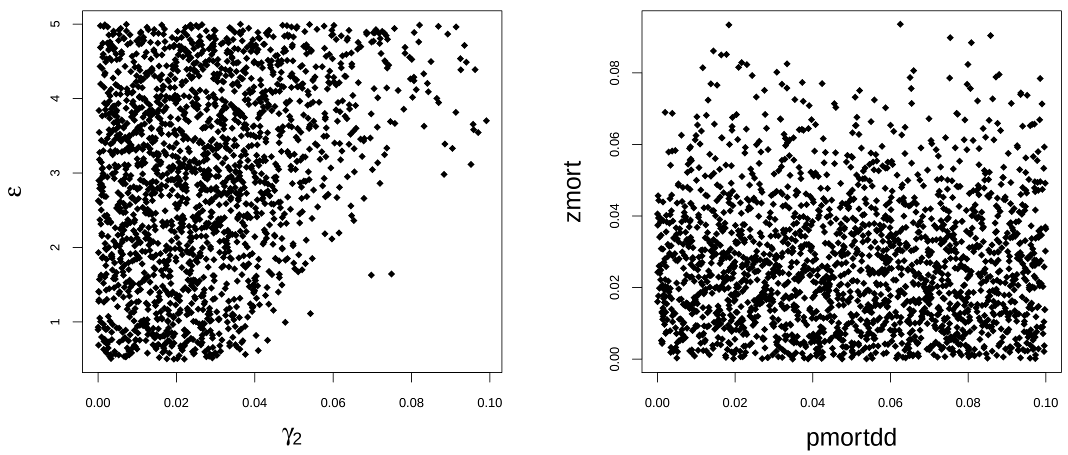
In this example, the emulators for zooplankton related transfers all emulate the logarithm of the output, and so intervals are built on the log-scale, and the bounds then transformed back to the original scale.

This has the advantage that the lower bounds are all above zero, reflecting the simulators' inability to produce negative values. Using the emulators built from OGPV99 and HADPV99, any point in the input space can be categorised as either being very likely to result in zooplankton extinction, or not being likely enough to disregard.

This procedure was followed three times for each of HadOCC and OG99NPZD, twice with $n = 1000$ (to build training and prediction data) and once with $n = 100$ (to build prediction data). This resulted in six new datasets: OG948 (with 948 points), OG947 (with 947 points), HAD1007 (with 1,007 points), HAD1005 (with 1,005 points), OG98 (with 98 points) and HAD119 (with 119 points).

Figure 6.12 shows the distribution of input points in the new datasets between pairs of inputs that were shown to be instrumental in zooplankton extinction. A lower density of points for high `zmort` in HAD1005 and HAD1007 and for high γ_2

and low ϵ in OG948 and OG947 is clear, as expected.



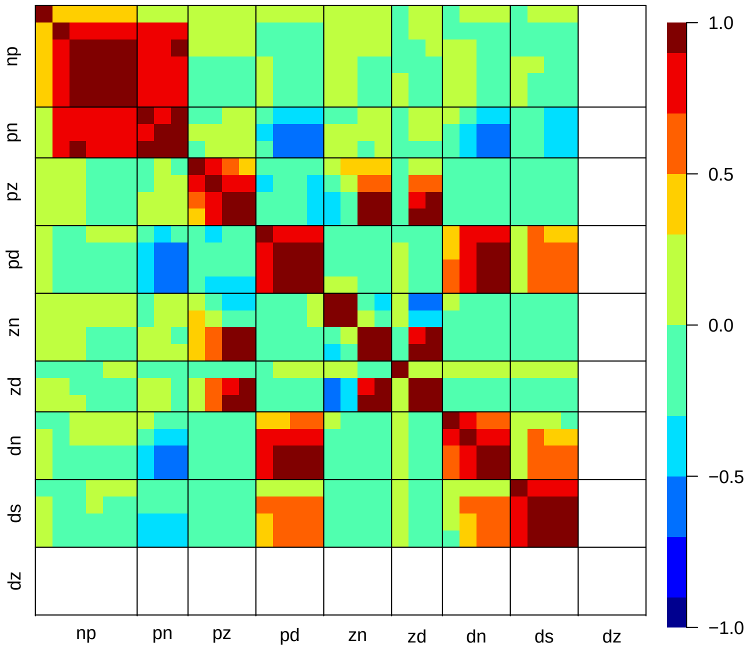
(a) *Combined OG948 and OG947 input points*

(b) *Combined HAD1005 and HAD1007 input points*

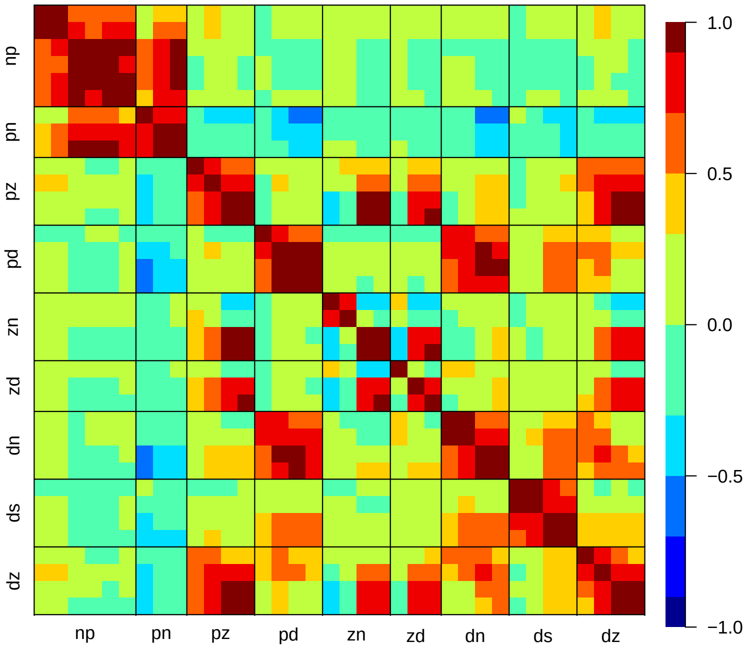
Figure 6.12: *Pairwise distribution of points in the refined input designs.*

Variable	Principal variable time points	
	OG1000 and HAD1000	OG948 and HAD1005
`iz.np`	1, 3, 146, 277, 365	277, 1, 3, 365, 12, 146
`iz.pn`	1, 29, 282, 365	282, 1, 30
`log.iz.pz`	3, 35, 164, 365	133, 3, 31, 359
`iz.pd`	2, 17, 99, 355	109, 1, 15, 355
`log.iz.zn`	21, 145, 1	172, 28, 343, 1
`log.iz.zd`	1, 130 365	141, 1, 365
`iz.dn`	3, 10, 112, 365	115, 1, 7, 253
`iz.ds`	2, 13, 124	127, 1, 6, 319
`log.iz.dz`	5, 34, 148, 365	148, 5, 34, 365
`log.iz.pon`	99, 1, 13, 365	105, 1, 13, 365

Table 6.8: *Principal variables for the old datasets and for the new, refined dataset. These were chosen so that at least 99% of the variation is explained in both datasets, and are given in the order in which they were selected. There are therefore now 32 principal variables for OG99NPZD, and 36 for HadOCC.*

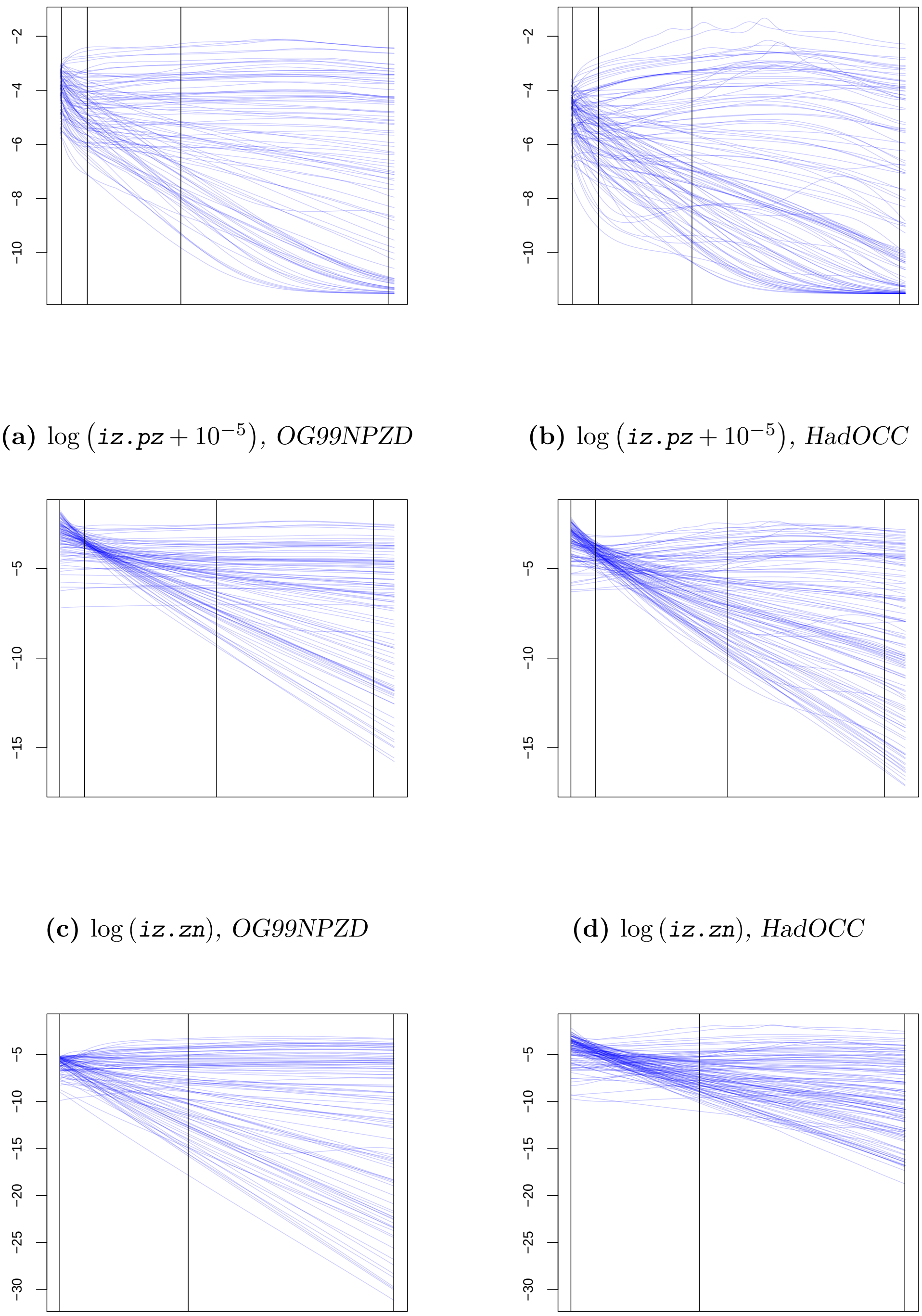


(a) *OG99NPZD*



(b) *HadOCC*

Figure 6.13: *Heat maps of correlation matrices for principal variables from OG947 and HAD1007. Each square represents a particular principal variable, and these are ordered by time (from left to right and from top to bottom).*



(a) $\log\left(\texttt{iz.pz} + 10^{-5}\right)$, *OG99NPZD*

(b) $\log\left(\texttt{iz.pz} + 10^{-5}\right)$, *HadOCC*

(c) $\log\left(\texttt{iz.zn}\right)$, *OG99NPZD*

(d) $\log\left(\texttt{iz.zn}\right)$, *HadOCC*

(e) $\log\left(\texttt{iz.zd}\right)$, *OG99NPZD*

(f) $\log\left(\texttt{iz.zd}\right)$, *HadOCC*

Figure 6.14: *Time series plots of zooplankton related intermediate variable output for OG98 (left) and HAD119 (right). Each plot covers a year.*

Figure 6.14 shows time series of zooplankton related intermediates from OG98 and HAD119, comparable to those in Figure 6.3. Although there are still signs of zooplankton extinction, a higher density of non-zero zooplankton transfers is evident. Box-Cox model selection, applied as before, still supports a log transformation for each zooplankton related intermediate.

New principal variables, shown in Table 6.8, were then found for each intermediate variable, using the combined data from OG948 and HAD1005. These were then used to build new emulators, which will be used from now on. As before, a second order surface was fitted for each intermediate variable, using stepwise model selection with the Bayesian information criterion (BIC), and one correlation length was estimated per intermediate variable. Heatmap plots of the correlation matrices for the new datasets OG947 and HAD1007 are shown in Figure 6.13.

This stage illustrates the value of intermediate variables for analysing individual simulators, since as we have already mentioned, these implausible regions of input space could not have been found using the output variables alone. The input regions could be further refined by re-iterating this method, but this is not something we will pursue here.

This history matching exercise already exposes some difference in the simulators, in that in both cases there is a single input parameter almost entirely responsible for the extinction of zooplankton. These two parameters, γ_2 in OG99NPZD and `zmort` in HadOCC, are given different meanings: γ_2 is the rate of zooplankton excretion, and `zmort` a constant zooplankton mortality rate. The influences of these two parameters are shown in Figures 2.1 (page 10) and 2.2 (page 13), where it is clear that unless `fzmortdin` $= 0$ in HadOCC, they do not relate to exactly the same nitrogen transfers.

Steps 10 & 11: Finding and interpreting the relative coefficients

The relative coefficients from the emulators built over OG948 and HAD1005 were then calculated, by dividing each posterior coefficient by the largest coefficient relating to the same output. An abridged version of these is shown in Appendix B,

Table 6.9: R^2 *for regression surfaces in each simulator's input to intermediate emulators.*

Variable	Time	OG99NPZD	HadOCC
iz.np	1	0.996	0.972
	3	0.980	0.980
	12	0.995	0.982
	146	0.990	0.968
	277	0.992	0.973
	365	0.995	0.983
iz.pn	1	1.00	1.00
	30	0.997	0.993
	282	0.998	0.992
log.iz.pz	3	0.999	0.999
	31	0.993	0.994
	133	0.969	0.977
	359	0.944	0.938
iz.pd	1	1.00	0.998
	15	0.992	0.988
	109	0.986	0.968
	355	0.986	0.950
log.iz.zn	1	0.972	0.992
	28	0.960	0.988
	172	0.939	0.973
	343	0.947	0.952
log.iz.zd	1	0.992	0.984
	141	0.975	0.960
	365	0.956	0.946
iz.dn	1	1.00	1.00
	7	0.999	0.999
	115	0.987	0.975
	253	0.986	0.967
iz.ds	1	1.00	1.00
	6	0.999	0.998
	127	0.974	0.966
	319	0.968	0.948
log.iz.dz	5		0.996
	34		0.988
	148		0.963
	365		0.907

	OG99NPZD	HadOCC
`iz.np`	Pos: μ_P, a (esp. early), c (early), $a \times K_1$, $a \times \mu_P$, $c \times K_1$ (early) Neg: K_1, a^2,γ_2^2 (late), $a \times c$ (early), $K_1 \times \mu_P$ (early)	Pos: `presp`, `alphachl`, `photmax`, `photmax` × `presp`, `kdin`^{2} (esp. early), `photmax` × `kdin`, `presp` × `zmort` `alphachl` × `presp` Neg: `kdin`, `kdin` × `presp`, `presp`^{2}, `rcchl` × `presp`, `rcchl`, `zmort`^{2} (late)
`iz.pn`	Pos: μ_P Neg: μ_P^2	Pos: `presp`, `photmax` × `presp`, `alphachl`, `photmax` Neg: `presp`^{2}, `kdin`
`log.iz.pz`	Pos: ϵ, γ_1, $\epsilon \times \gamma_2$ (late) Neg: γ_2, ϵ^2, γ_2^2	Pos: `epsfood`, `betap`, `fingest`, `epsfood` × `zmort` Neg: `zmort`, `presp`, `kdin`, `epsfood`^{2} (early)
`iz.pd`	Pos: μ_{PP}, μ_P^2, $\gamma_1 \times \gamma_2$, $a \times K_1$, $\epsilon \times \gamma_2$ Neg: μ_P, γ_2^2, γ_2, a^2	Pos: `pmortdd`, `presp`^{2} (late) Neg: `presp`, `presp` × `pmortdd`, `kdin`, `kdin` × `pmortdd`
`log.iz.zn`	Pos: γ_2 (early), ϵ, $\epsilon \times \gamma_2$, γ_1, $\gamma_1 \times \gamma_2$ Neg: γ_2 (late), γ_2^2	Pos: `epsfood`, `zmort` (early), `fzmortdin` (early) Neg: `zmort` (late), `zmort`^{2}
`log.iz.zd`	Pos: μ_{ZZ} (early), ϵ, $\epsilon \times \gamma_2$, γ_1, $\gamma_1 \times \gamma_2$ Neg: γ_2, μ_{ZZ}^2 (early), γ_2^2	Pos: `epsfood`, `zmort` (early) Neg: `zmort` (late), `zmort`^{2}
`iz.dn`	Pos: μ_D (early), μ_{PP}, μ_P^2 (late), $a \times K_1$, ϵ Neg: μ_P, γ_2, a^2, $\gamma_1 \times \gamma_2$, $\gamma_2 \times \mu_{ZZ}$	Pos: `pmortdd`, `presp`^{2} (late), `zmort` (early), `pmortdd` × `zmort` Neg: `presp`, `presp` × `pmortdd` (late), `fzmortdin` (early), `zmort` × `fzmortdin` (early), `kdin`
`iz.ds`	Pos: w_s, μ_{PP}, μ_P^2 Neg: $w_s \times \mu_P$, μ_P, $w_s \times \mu_D$, μ_D	Pos: `dsink`, `pmortdd`, `pmortdd` × `dsink`, `presp`^{2}, `pmortdd` × `zmort` Neg: `presp`, `presp` × `dsink`, `presp` × `pmortdd`
`log.iz.dz`		Pos: `epsfood`, `pmortdd` (early), `betad` (early), `rcnphy` (early), `fingest` Neg: `zmort`, `presp`, `kdin`, `epsfood`^{2} (early)

Table 6.10: *Summaries of the key input parameters (those with relative coefficients of greater magnitude than* 0.3*) from each simulator for each intermediate variable, in decreasing order of activity.*

along with some performance summaries from predictions over OG947 and HAD1007.

Any term with no relative coefficients above 0.15 are omitted, and values below 0.15 are replaced by a dot, for ease of reference.

Clearly this, and the analysis in Section 6.6.3, are only meaningful if the emulators have been checked with diagnostic tools such as those described in Section 3.5. For the emulators built in this Chapter, the behaviour of the error and SPE values were plotted against each input (or intermediate variable in Section 6.6.3) and no clear trends were found. The R^2 values for the regression surfaces in each input to intermediate emulator are given in Table 6.9. In general the OG99NPZD surfaces appear to capture a slightly higher proportion of the variance than those in the HadOCC emulator.

The main inputs affecting each intermediate variable are given in Table 6.10. For each simulator and each intermediate variable, these give the terms in the regression surface in roughly descending order of importance, and show which have a positive and which a negative coefficient. These were found using the relative coefficient tables in Appendix B. It is clear that only a subset of each set of inputs is active in each emulator. The terms mentioned in Table 6.10 are those with a relative coefficient whose magnitude is more than 0.3 for at least one time point.

At this point, knowing about any collinearity in the new input designs is crucial, since if the designs are far from orthogonal our interpretations of these coefficients will be meaningless. A first tool is to find the eigenvalues of the correlation matrix of the design. The ratio of the highest and lowest eigenvalues of the correlation matrix of OG948 is 2.66, and the ratio for HAD1005 is 2.69. Neither of these is high enough to raise any alarm, and so we continue with the method.

Combining this information with the details of the inputs' relationships to one another, the questions from step 11 on page 140 can be addressed. A summary for the first, concerning pairs of inputs given the same meaning in the two simulators, is given in Table 6.11. Most of the input parameters that appear to represent the same quantities do behave similarly, although many of them are mostly inactive. Tables 6.12 and 6.13 summarise the effects of the input parameters unique to each simulator, addressing the second question. These show some much more important parameters, particularly μ_P and γ_2 in OG99NPZD and `presp` and `zmort` in HadOCC.

OG99NPZD input	HadOCC input	Summary
γ_1	`betap`	Little effect in either, except in `log.iz.pz`. Slight effect in other zooplankton transfers in OG99NPZD.
α	`alphachl`	`alphachl` active in `iz.np` and `iz.pn`, whereas α has very little effect.
ϵ	`epsfood`	Mostly similar, ϵ more active. Different interactions in `log.iz.zd`.
g	`gmax`	Very little effect.
C_{pp}	`rchlpig`	Very little effect.
PAR	`rparsol`	Very little effect.
μ_{PP}	`pmortdd`	Similar for `iz.pd` and `iz.ds`, and for `iz.dn`.
μ_{ZZ}	`zmortdd`	Mostly inactive, except for μ_{ZZ} in `log.iz.zd`.
K_1	`kdin`	Similarly active in `iz.np`. HadOCC also active for `iz.pn, iz.pd,` `iz.dn` and `log.iz.pz`
w_s	`dsink`	Very similar for `iz.ds`, inactive elsewhere.

Table 6.11: *Summaries of effects of input parameters given the same meaning in OG99NPZD and HadOCC.*

OG99NPZD input	Summary
γ_2	Active in `log.iz.pz, log.iz.zn, log.iz.zd`, mildly in `iz.np,` `iz.pd`
μ_P	Strongly active in `iz.np, iz.pn, iz.pd, iz.dn, iz.ds`
μ_D	Active in `iz.dn` and mildly in `iz.ds`
a	Active in `iz.np`
c	Mildly active in `iz.np`

Table 6.12: *Input variables in OG99NPZD that have no equivalent in HadOCC.*

The third question, about groups of parameters representing the same process, deals with a combination of these results. A summary of these groups of input parameters is given in Table 6.5. Of the photosynthetic rate parameters, a is much more active than c in OG99NPZD, and both behave fairly differently from `photmax` in HadOCC. The extra phytoplankton mortality inputs, `fpmortdin` and `pminmort` seem largely inactive in HadOCC, whereas μ_P in OG99NPZD is clearly very important. The linked phytoplankton mortality related inputs, μ_{PP} and `pmortdd`, have a very similar effect, being influential in `iz.pd, iz.dn` and `iz.ds` in similar ways in both simulators. Thus it seems that the differences in modelling of phytoplankton mortality potentially may make a real difference to the values of some intermediate variables, rather than cancelling out owing to different uses of the linked parameters.

HadOCC input	Description
`rcchl`	Mildly active in `iz.np`
`rcnphy`	Mildly active in `log.iz.dz`
`rcnzoo`	Inactive
`rcndet`	Inactive
`photmax`	Active in `iz.pn` and `iz.np`
`presp`	Strongly active in all except `log.iz.zn` and `log.iz.zd`
`pminmort`	Inactive
`fpmortdin`	Inactive
`fmingraz`	Inactive
`fingest`	Mildly active in `log.iz.pz` and `log.iz.dz`
`betad`	Mildly active in `log.iz.dz`
`fmessyd`	Inactive
`zmort`	Strongly active in `log.iz.pz, log.iz.zn, log.iz.zd, log.iz.dz`, mildly active in most others
`fzmortdin`	Mildly active in `log.iz.zn` and `iz.dn`
`rco3pprod`	Inactive
`nitrifaph`	Inactive
`nitrifeuph`	Inactive

Table 6.13: *Input variables in HadOCC that have no equivalent in OG99NPZD.*

Of HadOCC's extra zooplankton mortality parameters `zmort` and `fzmortdin`, `fzmortdin` is only mildly active while `zmort` is active in each of the zooplankton related intermediates, and some more besides. The linked inputs μ_{ZZ} and `zmortdd` have very little effect, apart from μ_{ZZ} in `log.iz.zd`. HadOCC's extra grazing parameters `fmingraz, fingest, betad` and `fmessyd` are mostly inactive, except slightly in `log.iz.pz` and HadOCC's extra intermediate variable `log.iz.dz`. This is perhaps not surprising, since these intermediates arise from the grazing of phytoplankton and detritus by zooplankton.

Finally, there may be pairs of inputs that are given different meanings in the two simulators but behave very similarly. These are likely to be linked to similar nitrogen transfers. Two candidate pairs are μ_P and `presp` (both of which cause transfers from phytoplankton to nutrient), and γ_2 and `zmort` (causing nitrogen to leave zooplankton).

The effects of μ_P and `presp` are very similar throughout, particularly in `iz.np, iz.pn, iz.pd` and `iz.dn` where both are very active. In the emulators from inputs to output (`log.iz.pon`), shown in Table B.1, these are the most active variables at each time point. In OG99NPZD μ_P is a mortality rate for phytoplankton, whereas in HadOCC `presp` is the phytoplankton respiration rate. This perhaps indicates that a better understanding of these areas of the system would much improve modelling.

The similarity between the effects of γ_2 in OG99NPZD and `zmort` in HadOCC is not quite so striking, but still apparent; `log.iz.pz, log.iz.zn, log.iz.zd, iz.np` and `iz.dn` show mostly the same effects, but with interaction effects that are not always so easy to identify between simulators. This is perhaps because γ_2 concerns only a transfer from zooplankton to nutrient (through excretion), whereas, depending on the value of `fzmortdin`, `zmort` can also affect transfers from zooplankton to detritus and nutrient.

Summary

For each simulator, this stage has increased our understanding of how each input variable contributes to the representation of the ocean carbon cycle. The conclusions drawn are independent of the choice of output variable, and so can be used to count certain input parameters as less active in general than others. Even with only one simulator, this stage enables reduction of the input space, using unrealistic intermediate variable values, that would not necessarily be achieved using the output, as seen in the example.

In Section 6.6, the intermediate variables are used to emulate the output. This will be used to compare how the different processes interact to form the output for each simulator. Once it is known which intermediate variables are more active, the results of this section will show which inputs are most active, and where effort in understanding the relationship between the two simulators and between their input spaces should be focused.

6.6 Emulating output from intermediate variables

Knowing the effects of the intermediate variables on the output can increase our understanding of each simulator and of the differences between their representations of the system.

Because the intermediate variables (before dimension reduction) should capture all the information in the simulator at each time point, it should be possible to use the dimension-reduced intermediate variables to build emulators of the output for each simulator. This is a good test of the choice and implementation of the intermediate variables and their dimension-reduction. If these have been done well, it should be possible to make accurate predictions from intermediate to output variables for other datasets. If it isn't, this is a sign that at some point in the process things have gone awry.

As in the previous section, these methods rely on the emulators being well constructed, and so they must be carefully validated before being used to compare simulators. Appendix C.2 shows some methods for validation of the intermediate to output emulators, studying the behaviour of the SPE. Appendix C.3 shows how this stage of intermediate variable emulation can be combined with the previous stage to give an indication of the retention of information by the dimension-reduced intermediate variables. This is done by combining the input to intermediate and intermediate to output emulators to produce an input to output emulator, from which we can sample from the posterior distribution of the output at new input points. How similar the samples are to samples from a standard input to output emulator indicates how much information has been lost in the process. These methods are illustrated using OG99NPZD and HadOCC.

In the intermediate variables, the emulators for each simulator now have input variables with the same meaning, whose effects can be directly compared. The emulators may reveal some of the differences and similarities between the simulators in terms of how they use the processes they model. Where there are processes represented in only one simulator, and therefore an extra group of intermediate variables in its emulator that cannot be linked to any variables in the others, the emulator can be used to show whether this process performs a key role in the simulator. However, whereas the behaviour of the input to intermediate emulators was relatively simple to interpret, this is not so for the intermediate to output emulators.

6.6.1 Intermediate variables as inputs

The main thrust of the difficulties encountered in working with intermediate variables is that we are not in control of the values they take. To produce training data one would ideally like to be able to choose points in intermediate variable space, and then run the simulator over these in the way that one can usually run it from the input variables. However, this is not a simple idea. Often, the simulators each calculate all variables (intermediate and output) at the first time step, then use these values and the inputs to calculate the same variables at the second, and so on. The value of a particular variable at time step i is derived from the values of (potentially) all variables at time step $i-1$, and so the time series are all heavily intertwined.

Although in theory it is conceivable that one could rearrange a simulator so that given a full set of (non-dimension-reduced) intermediate variables it was able to produce the output, this would likely be very difficult. Furthermore, although a particular point in intermediate variable space should be deterministically associated with an output value, it is also associated with an input point, from which it was (at least notionally) created. This means that there are potentially many invalid intermediate points, which a simulator is not capable of producing from any input point. Along with a version of the simulator that had intermediate variables as input, one would also need a way of rejecting invalid parts of intermediate variable space.

Because the intermediate variables represent processes that are linked to one another, and because the principal variables we choose to represent each intermediate variable will often come from very structured data over time and space, we can expect high correlation between the intermediate variables. The PCA in Section 6.4.1 may also show, through the least principal components, that there are some linear combinations of intermediate variables whose variance is very small, and therefore some pairs of linear combinations that are highly correlated.

This lack of orthogonality means that the coefficients in the regression surface are not independent, and therefore cannot be compared as before. It may lead to more serious problems, such as the 'bouncing beta' phenomenon, where estimates and variances for coefficients become very high. Some possible strategies for dealing with this problem are summarised by Kiers and Smilde (2007).

Rather than focus on the comparison of coefficients therefore, as in the input to intermediate stage (where the input space was designed to be roughly orthogonal), methods in this stage use simple performance summaries and plots relating to emulator or simulator output in order to observe the effects of the intermediate variables. We also take advantage of the common input space (the intermediate variables) the two emulators share by running each emulator over data from the other simulator.

6.6.2 Methods and analysis

This analysis splits into two phases:

- Studying the effects of the intermediate variables within each simulator
- Investigating the differences between the two simulators in terms of their use of intermediate variables to produce output

An idea of the behaviour of each simulator can be gained by studying the correlations between the intermediate variables and the output. This can be done for each intermediate variable in turn to investigate the main effects, or using products of intermediate variables to show the effects of interactions on the output variables. The first method is simple to display, for example by plotting the correlations between each principal variable and the output variable, grouped by intermediate variable. The second can be displayed by creating a matrix whose $(i, j) - th$ entry is the correlation between the element-wise product of the i^{th} and j^{th} intermediate variables and the output, and plotting this matrix using a heatmap. Examples of both sorts of plot are given later in this section and in Appendix C.2.

This stage gives useful insight into each simulator's representation of the system, and so may even be used in a single simulator context to learn more about how the processes within a simulator are used to reach the output. Already these plots may reveal different features in the two simulators, particularly if an intermediate variable is very active in one but not in the other. However, the emulators can be used for simulator comparison in a much more informative way.

Because the emulators from intermediate to output variables now have the same input space for each simulator, an emulator of one simulator can be used over data from the other. If an emulator predicts another simulator's behaviour fairly well, that indicates that the two simulators' treatments of the intermediate processes are not so different. If performance is poor, this indicates a contrast.

When analysing the differences between these emulators, we must keep in mind the properties of each intermediate variable space, learned through analysing the intermediate variable data in Section 6.4. In particular, our lack of control over the intermediate variable space makes us unable to define the ranges of the intermediate variables in the training or prediction data, and the two simulators may produce values with very different ranges. Inferences made using the emulator of one simulator to predict the behaviour of another will be highly unreliable if the emulator is operating outside the range of its training data unless the emulator's variance, which will be high at these points, is taken into account.

For this reason, the standardised prediction error (SPE) is more a reliable quantity for analysis than those that do not take account of the emulator variance. Suppose there is a cluster of high error (emulator minus simulator, unstandardised) values in a particular region. If the SPE values are also high, this indicates that even when standardised by the emulator variance (which in this case is clearly relatively small) this trend exists. If there is no prevalence of high SPE values in this region, the emulator variance there is high, and therefore the difference highlighted by the error values should not be given much weight. This will especially be the case when a particular region of intermediate variable space is more sparsely populated by one simulator than by the other.

The behaviour of the SPEs from predictions using the emulator of one simulator over data from the other can be monitored across intermediate variable space. Extending the validation method of emulating SPE values mentioned in Section 3.5, these SPE values can be emulated with the intermediate variables as inputs. If the emulator captures a large proportion of the variation in the SPEs, using the intermediate variables as inputs, this indicates that there is systematic behaviour in the SPE values. This in turn indicates that the emulator is not capturing some of the behaviour of the data, and therefore gives reason to believe that there are systematic differences between the two simulators.

For comparison, the SPE from predictions using an emulator of the same simulator should also be emulated, as well as some random vectors. The proportion of variation captured by the regression surface when each of these is emulated forms a good basis for comparison, particularly when the number of points is small relative to the dimension of the intermediate variable space, which could lead to over-fitting. If a relatively high proportion of variance in the errors for the emulator of another simulator is explained by the regression surface, this indicates that there are systematic differences between the two simulators. These may not be particularly easy to discern or describe, but methods shown in the rest of this section should reveal the most important differences.

Plots similar to those used to understand the effects of intermediate variables on the output can show the effects of the intermediate variables on the SPE. Instead of plotting correlations between intermediate variables (or products of intermediate variables) and output, we plot their correlation with the SPE. Examples of these plots are given in Figures 6.19, 6.20, 6.21 and 6.22. Such plots are also used in Appendix C.2 to validate the emulators on data from the same simulator as their training data. This is imperative if conclusions are to be drawn from the emulators' performance over data from other simulators.

If the SPEs show a pronounced trend against a particular intermediate variable, this indicates that there is an effect in the prediction data which is not captured by the emulator of the other simulator. This intermediate variable may be inactive in the other simulator, or it may contribute differently. The plots showing correlation with simulator output will help discern which is the case, and will also enable one to differentiate between intermediates which have a similar and strong effect on both simulators, and those which are much less active.

Finally, understanding gained from this stage can be combined with that from the input to intermediate stage to show further which inputs are linked, and which are driving differences in the simulators.

Method summary

To summarise, when analysing the relationship between intermediate and output variables, the following steps are useful

12. Study the behaviour of each simulator in terms of its intermediate variables, using correlations between intermediate variables (or products of pairs of intermediate variables) and output.

13. Use the emulator of one simulator to predict the behaviour of the other. Compare the RMSE to that using an emulator of the same simulator. If there is little difference, the simulators appear to use their intermediate variables in similar ways.

14. If the RMSE from the emulator of the other simulator is much larger, study the SPE values to unearth the roots of the difference. If a high proportion of variance can be explained by a regression surface, this indicates some systematic trends. Study the correlations between intermediate variables and SPEs to try to reveal these trends.

These steps should show where the main differences lie between each simulator's handling of the intermediate processes. Having studied the effect of each input space on the intermediate variables in Section 6.5, our findings can be combined to show which of the input to intermediate relationships are the most crucial to understand.

For intermediate variables that have the same effect on both simulators, differences could still arise from the way they are created from the input variables. If an intermediate variable is very active therefore, working to understand the different ways the simulators calculate this intermediate variable could be very useful.

Input variables that are active only in intermediate variables that are largely inactive do not necessarily need to be as well understood. However, the intermediates for which they are active may be more important for different output variables.

This stage of intermediate variable emulation not only provides an insightful method for understanding a single simulator, but enables a direct comparison to be made using emulators, so that the differences between two simulators as functions can be seen and studied.

6.6.3 Example: Intermediate to output

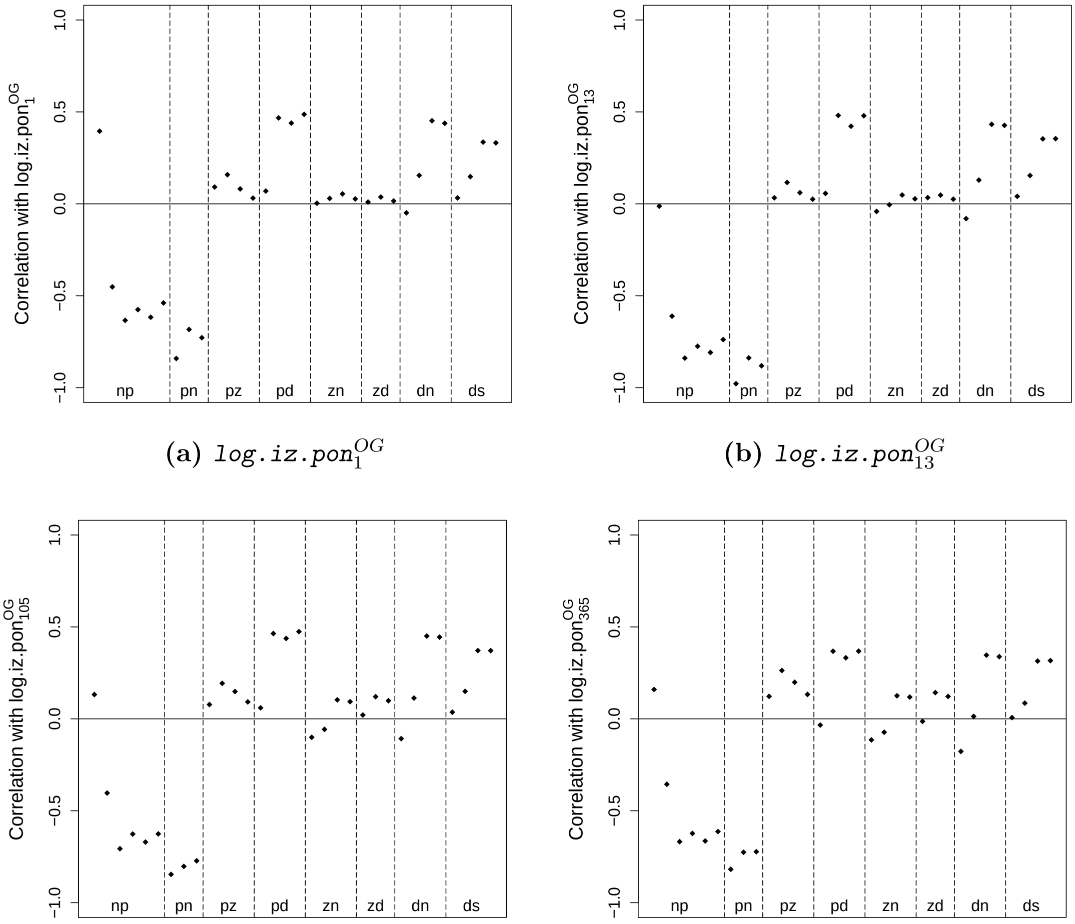
Step 12: Studying the behaviour of each simulator

An impression of the effects of intermediate variables on the OG99NPZD and HadOCC output is given by plotting the correlation between intermediate and output variables. Figures 6.15 and 6.17 show the correlations between each intermediate variable and each principal variable of `log.iz.pon` for OG947 and HAD1007.

Figures 6.16 and 6.18 show heatmaps of correlations between pairs of intermediate variables and output variables. The intermediate variables are all positive on their original scales, but the logarithms of the zooplankton related intermediates, which are the quantities used to produce these plots, are negative. This affects the signs of correlations between products of intermediate variables when one is zooplankton related.

OG99NPZD

In OG99NPZD it appears from Figure 6.15 that `iz.pn` has a very negative effect on `log.iz.pon` at each time, as well as all but the first time point of `iz.np`.

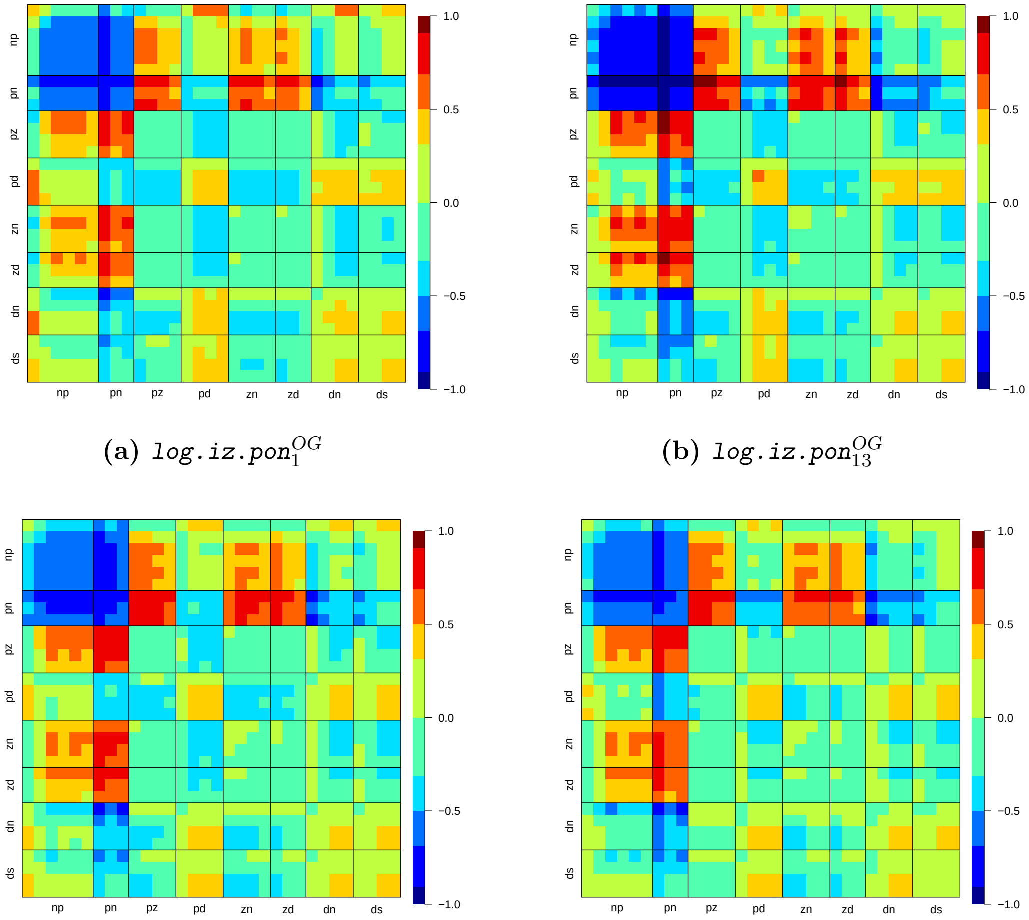


(a) $\mathtt{log.iz.pon}_1^{OG}$ **(b)** $\mathtt{log.iz.pon}_{13}^{OG}$

(c) $\mathtt{log.iz.pon}_{105}^{OG}$ **(d)** $\mathtt{log.iz.pon}_{365}^{OG}$

Figure 6.15: *Correlations between each intermediate variable and each output variable from OG947. Each set of principal variables is in time order.*

As shown in Figure 6.13, the correlations between these intermediate variables is very high. Discerning therefore which are actually affecting `log.iz.pon` is difficult. Both `iz.dn` and `iz.ds` (at their later two times) and `iz.pd` have a mildly positive effect, though these are more pronounced at the earlier time points of `log.iz.pon`.



(a) $\mathtt{log.iz.pon}_{1}^{OG}$ **(b)** $\mathtt{log.iz.pon}_{13}^{OG}$

(c) $\mathtt{log.iz.pon}_{105}^{OG}$ **(d)** $\mathtt{log.iz.pon}_{365}^{OG}$

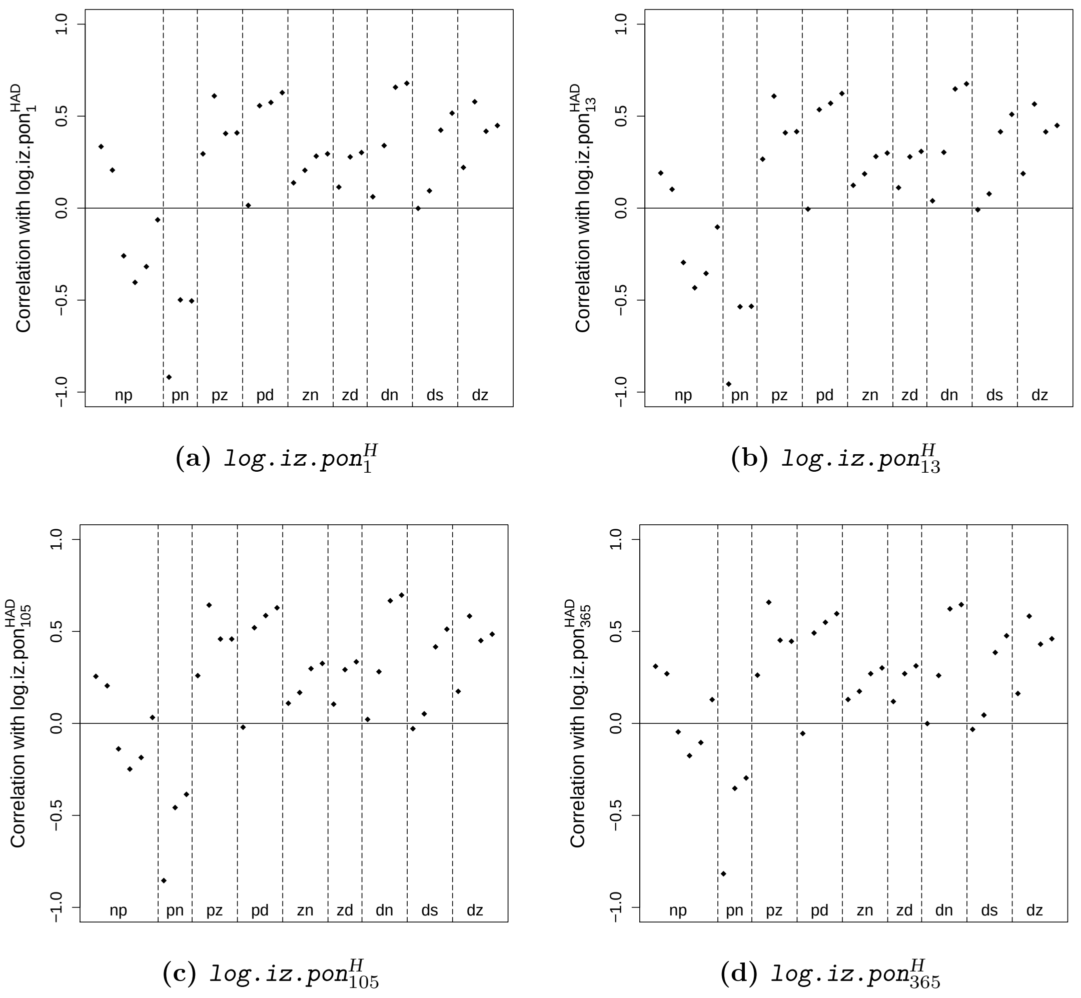
Figure 6.16: *Correlations between products of pairs of intermediate variables and each output variable from OG947. Each set of principal variables is in time order.*

Correlations between iz.dn and iz.pd are very high, leading to more difficulties in interpretation. The strongest second order effects in OG99NPZD (shown in Figure 6.16) are of iz.np^2 and iz.pn^2, particularly iz.pn$_1^2$. There appear to be some strong interactions between iz.pn and other intermediates, especially the zooplankton related ones.

HadOCC

Figure 6.17 shows that in HadOCC also iz.pn has a negative effect, particularly iz.pn$_H^1$. In contrast, iz.np seems to have very little effect on HadOCC output.

As in OG99NPZD, `iz.pd` and the latter two time points of `iz.dn` and `iz.ds` have a positive effect. Whereas in OG99NPZD the zooplankton related intermediate variables on their own show almost no correlation with output, in HadOCC they all show a slightly positive correlation. This is particularly true of `log.iz.pz` and `log.iz.dz`, whose correlations are very similar at all time points. The correlations between `log.iz.pz` and `log.iz.dz` in HAD1007 are very high, and so discerning the individual effect of either on HadOCC output will be difficult.



(a) $\texttt{log.iz.pon}_1^H$

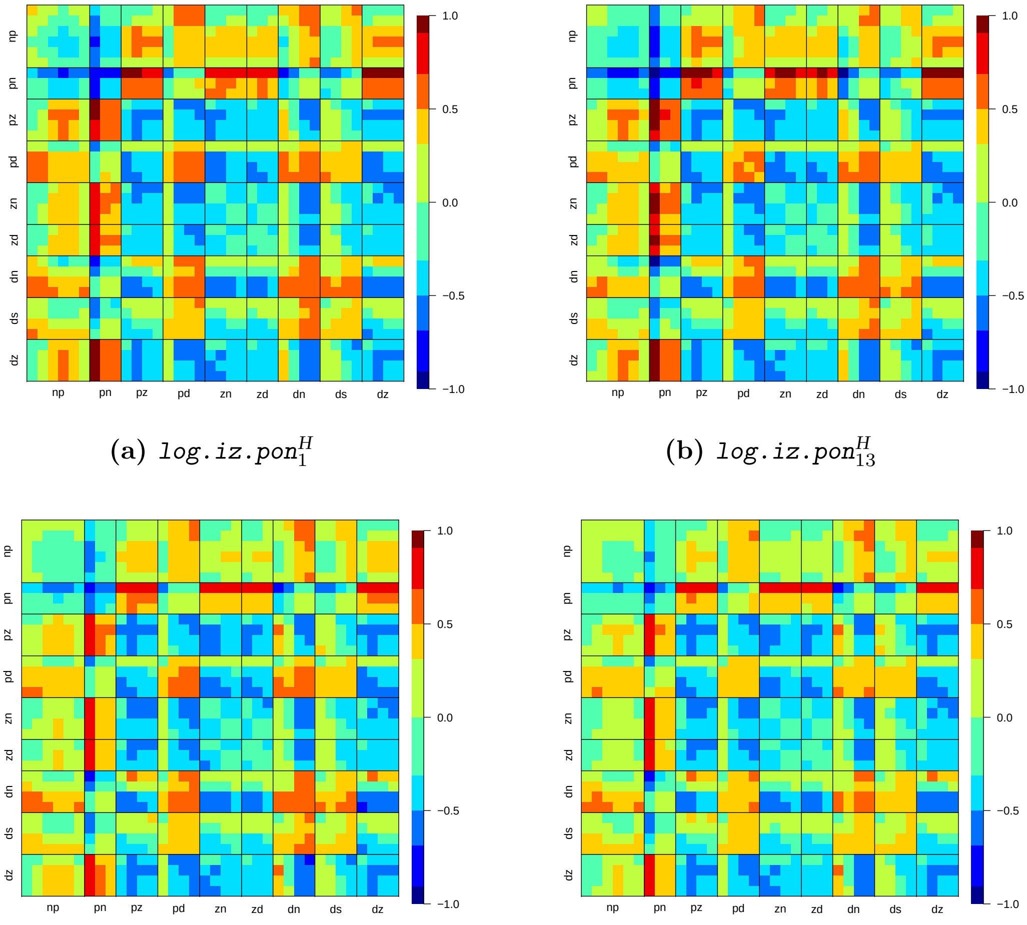
(b) $\texttt{log.iz.pon}_{13}^H$

(c) $\texttt{log.iz.pon}_{105}^H$

(d) $\texttt{log.iz.pon}_{365}^H$

Figure 6.17: *Correlations between each intermediate variable and each output variable from HAD1007. Each set of principal variables is in time order.*

The second order effects in HadOCC (Figure 6.18) appear much less simple than those in OG99NPZD. The effect of $(\texttt{iz.pn}_H^1)^2$ is strongly negative, and interactions between $\texttt{iz.pn}^1_H$ and `log.iz.pz` and `log.iz.dz` appear particularly strong. The correlations between output and $\texttt{iz.pd}^2$ and $\texttt{iz.dn}^2$ are quite high, though less so with $\texttt{log.iz.pon}^{365}_H$.



(a) $\texttt{log.iz.pon}^H_1$

(b) $\texttt{log.iz.pon}^H_{13}$

(c) $\texttt{log.iz.pon}^H_{105}$

(d) $\texttt{log.iz.pon}^H_{365}$

Figure 6.18: *Correlations between products of pairs of intermediate variables and each output variable from HAD1007. Each set of principal variables is in time order.*

Conclusions

To summarise the apparent behaviour of the individual simulators:

- Both simulators show a strongly negative effect of `iz.pn` and a mildly positive effect of `iz.pd` and later `iz.dn` on `log.iz.pon`.

- Both models show evidence of interaction effects between some time points of iz.pn and the zooplankton related intermediates.

- OG99NPZD shows a negative correlation between iz.np and log.iz.pon (though this could be an artefact of high correlation between iz.np and iz.pn in the data).
- HadOCC iz.pon appears to be influenced by log.iz.pz and log.iz.dz (though again high correlations in the data make the effects hard to separate).

Step 13: One emulator over the data of another

Before either emulator could be used on data from the other simulator, the prediction datasets had to be altered slightly. Because there are no transfers from D to Z in OG99NPZD, this was added as a variable so that the HadOCC emulator could be evaluated on OG99NPZD data, but was fixed at $\log(10^{-5})$ (the value representing no nitrogen transfer, since 10^{-5} is added to iz.dz in HadOCC before taking logs).

	OG947 data			HAD1007 data		
Time	Mean (& SD) log.iz.pon (OG947)	OG emulator	HAD emulator	Mean (& SD) log.iz.pon (HAD1007)	HAD emulator	OG emulator
1	3.10 (0.014)	0.0005	0.0076	3.09 (0.014)	0.0001	0.0035
13	2.86 (0.204)	0.0025	0.156	2.83 (0.251)	0.0019	0.0189
105	2.41 (0.332)	0.026	0.580	2.41 (0.505)	0.0198	0.165
365	2.22 (0.321)	0.0341	2.18	2.08 (0.664)	0.0510	0.214

Table 6.14: *RMSE for each emulator used to predict log.iz.pon from OG947 and HAD1007, as well as the mean and standard deviation of log.iz.pon at each time point.*

Table 6.14 shows the RMSE values for each vector of predictions using the emulators built from OG948 and HAD1005 over OG947 and HAD1007. At most times, the RMSE from the emulator constructed using the other simulator's data is over ten times the RMSE from the prediction of the emulator constructed using data from the same simulator. The errors are particularly large when the HadOCC emulator is used over OG99NPZD data at the latest time point. This is likely to be because the ranges of the intermediate variables are mostly wider in OG99NPZD than in HadOCC, and so the HadOCC emulator is extrapolating far beyond its training data. Because the RMSE, unlike the SPE, is not standardised by predictive variance, this is not accounted for.

Step 14: Studying the SPE values

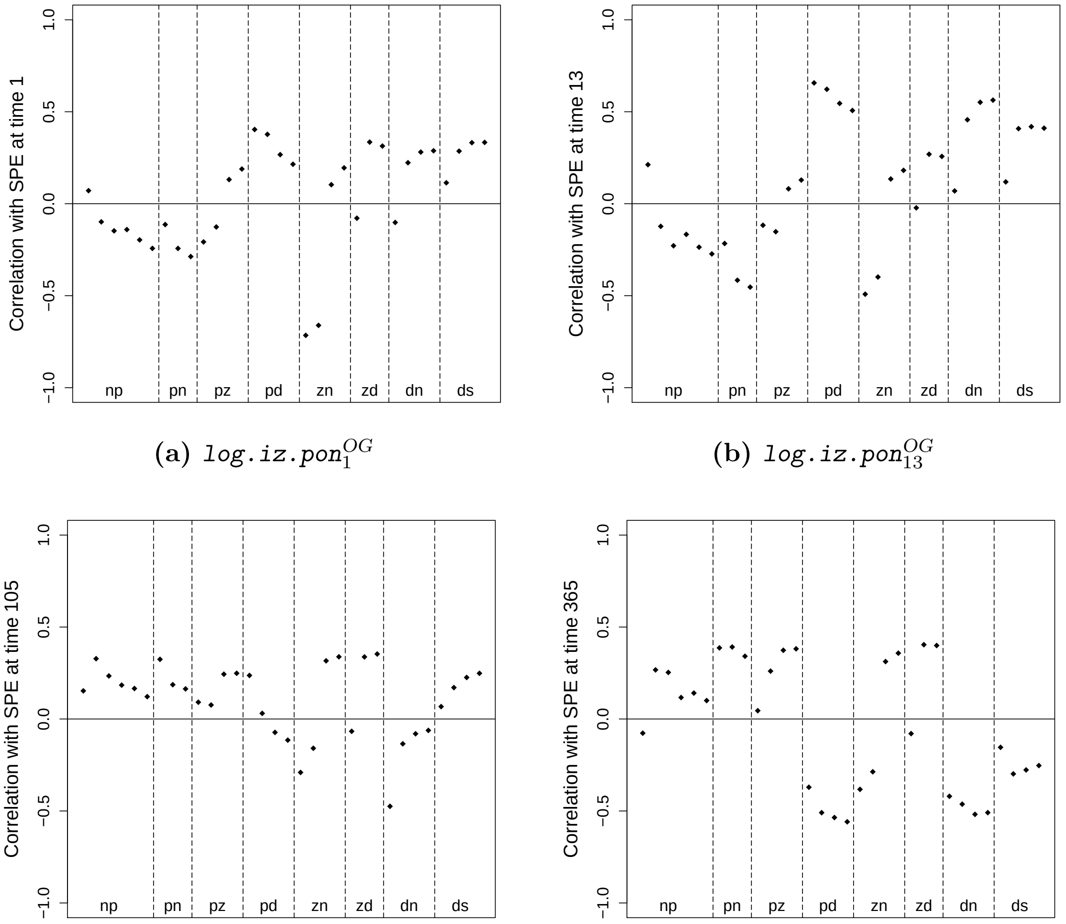
To investigate systematic trends in the SPE values when an emulator of the other simulator is used, emulators were built of each SPE vector, with intermediate variables as input. As well as this, emulators were built for the SPE vectors for predictions from an emulator of the same simulator as the data, and eight random vectors were generated, four each for the OG99NPZD and HadOCC data. These were drawn from a normal distribution with the same mean and variance as the SPE (in most cases these vectors appeared to be roughly normal). Each emulator was built with a second order regression surface, chosen using R's step function, and an isotropic correlation function with its correlation length estimated by maximum likelihood.

		Adjusted R^2			
Time		1	13	105	365
OG947	SPE (HadOCC)	0.839	0.984	0.958	0.980
	SPE (OG99NPZD)	0.077	0.105	0.246	0.205
	Random	0.033	0.024	0.0087	0.021
HAD1007	SPE (OG99NPZD)	0.973	0.953	0.966	0.919
	SPE (HadOCC)	0.259	0.152	0.178	0.228
	Random	0.011	0.0081	0.081	0.012

Table 6.15: *Variation captured by regression surfaces for SPE vectors when an emulator of the other simulator is used to predict* `log.iz.pon`*, compared with that for the emulator of the same simulator, and for similarly distributed random vectors.*

Table 6.15 shows the variation captured by the chosen regression surface for each of these vectors. In each case, the notion is supported that the SPE from using an emulator over data from the other simulator (the top row in each block of the table) contains systematic trends over the intermediate variables. Because of the strange shape of the intermediate variable spaces, and the collinearity in the data, using the emulator outside its training data range or interpreting the coefficients of the regression surface might lead to misleading results. Instead, we note that clear systematic trends are present and continue with the methods outlined.

The regression surfaces for the SPE from the emulator of the same simulator as the data (the second line in each part of Table 6.15) capture very little variation. This is encouraging, since a systematic trend would imply that the surface was not capturing the behaviour of the simulator.



(a) $\texttt{log.iz.pon}_{1}^{OG}$

(b) $\texttt{log.iz.pon}_{13}^{OG}$

(c) $\texttt{log.iz.pon}_{105}^{OG}$

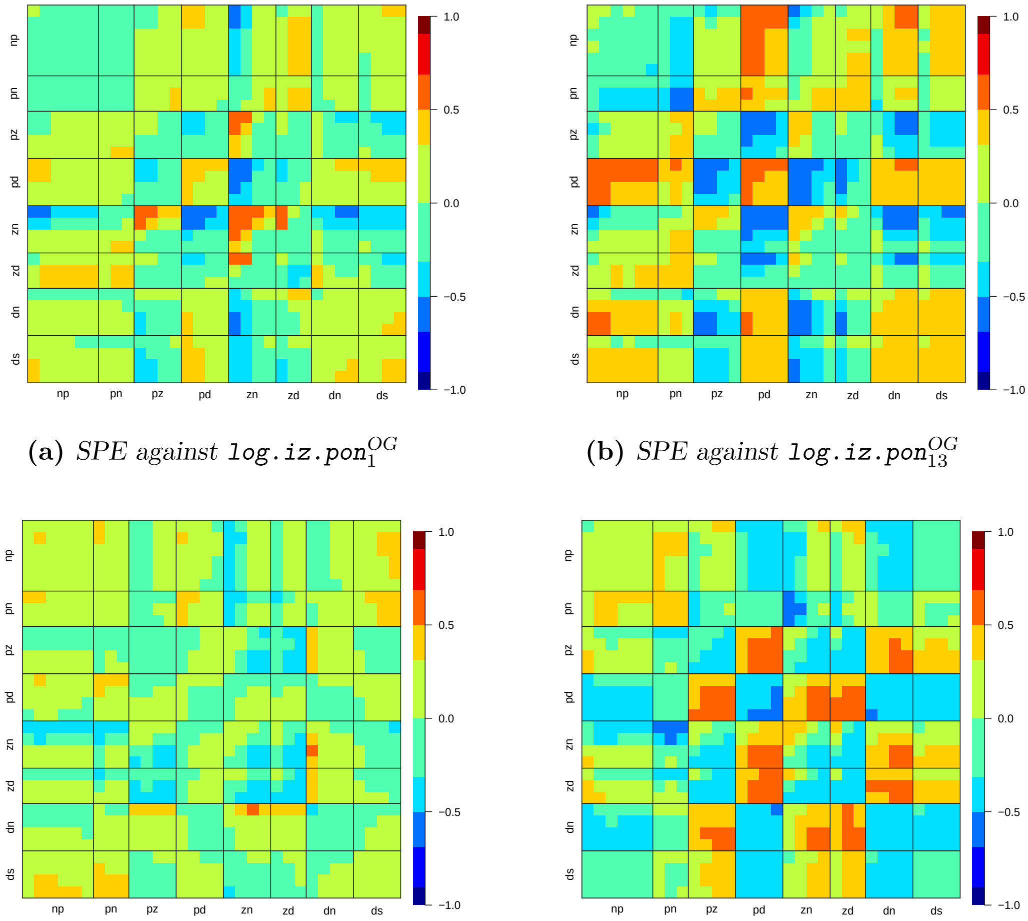
(d) $\texttt{log.iz.pon}_{365}^{OG}$

Figure 6.19: *Correlations between each intermediate variable in OG947 and the SPE using an emulator built from HAD1005.*

HadOCC emulator over OG947

Figures 6.19 and 6.20 show correlations between intermediate variables (or pairs of intermediate variables) in OG947 and the SPE when the emulator built from HadOCC data is used over OG947.

In both simulators, `iz.np` and `iz.pn` appear to have very little effect on the SPE values. That the effects of `iz.pn` should be mostly similar is perhaps surprising, since the transfer represents different processes in the two simulators.



(a) *SPE against* `log.iz.pon`$^{OG}_{1}$

(b) *SPE against* `log.iz.pon`$^{OG}_{13}$

(c) *SPE against* `log.iz.pon`$^{OG}_{105}$

(d) *SPE against* `log.iz.pon`$^{OG}_{365}$

Figure 6.20: *Correlations between products of pairs of intermediate variables in OG947 and the SPE using an emulator built from HAD1005.*

The most pronounced effect on the SPEs over OG947 from the HadOCC emulator come from the first two time points of `log.iz.zn` (at the earliest two time points of `log.iz.pon`) and `iz.pd`. The effect of `log.iz.zn` on SPE is difficult to trace, as this variable appears to have no strong effect in either simulator. Both simulators show positive links between output and `iz.pd`, however the effect `iz.pd` has on the SPE suggests that this effect is not the same. The latter two time points of `iz.dn`, which are highly correlated with `iz.pd`, show the same link, further preventing us from distinguishing between the two. Interactions between some of the zooplankton related intermediates and `iz.pd` appear to contribute strongly to the SPE of the HadOCC emulator over OG947, particularly at times 13 and 365. There is more evidence of these interactions being systematically related to output in HadOCC (from Figure 6.18) than OG99NPZD. In the example in Section 6.3.1 we saw that the principal variables for `iz.dn` represent the full HadOCC and OG99NPZD datasets somewhat differently. The apparent difference between their use of `iz.dn` could at least in part be attributable to this.

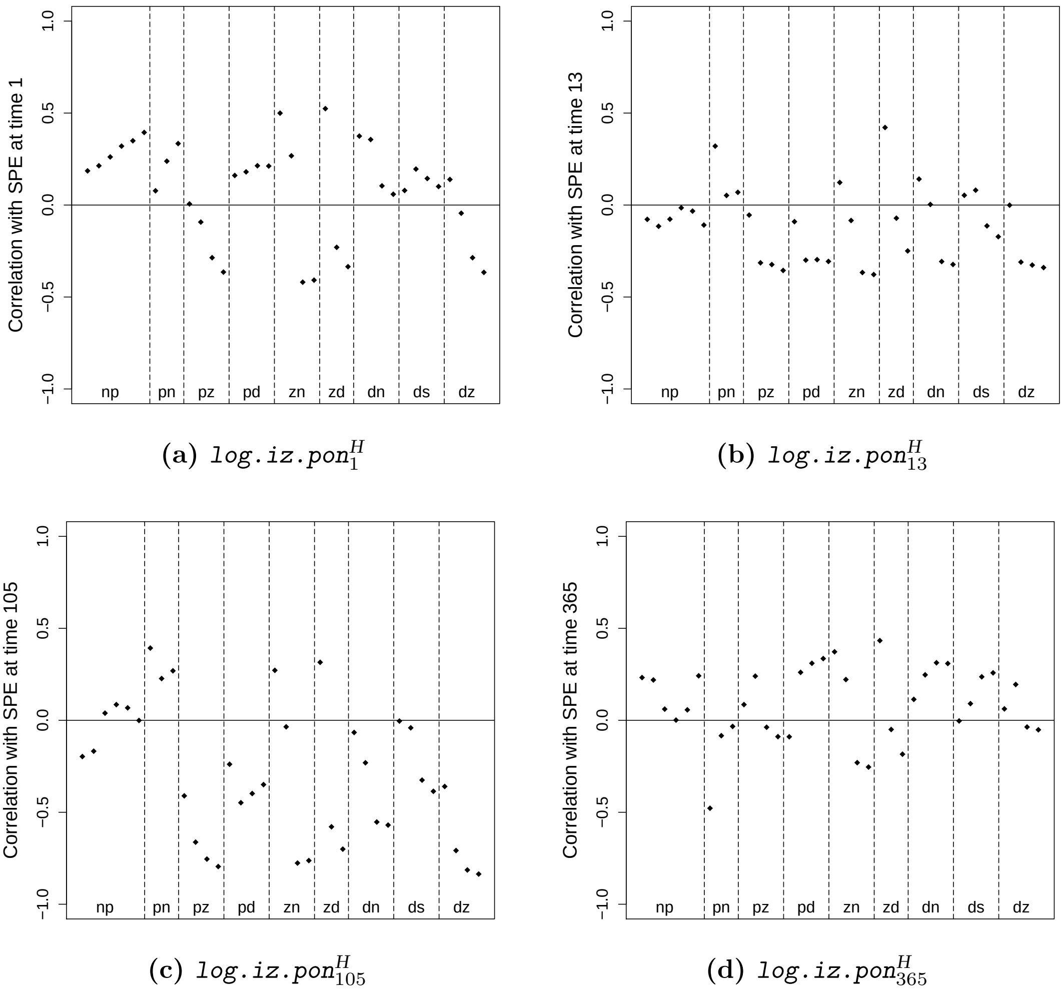
OG99NPZD emulator over HAD1005

Figures 6.21 and 6.22 show correlations between intermediate variables (or pairs of intermediate variables) in HAD1005 and the SPE when the emulator built from OG99NPZD data is used.

The zooplankton related intermediates show strong links to the SPE when the OG99NPZD emulator is used over HAD1007, particularly for $\texttt{log.iz.pon}_H^{105}$. This fits with the strong links between these variables and output in HadOCC that appear to be absent in OG99NPZD. The effects of `log.iz.pz` and `log.iz.dz` remain very similar. That there is no especially strong link between `log.iz.dz` and the SPE from the OG99NPZD emulator suggests that its effects on `log.iz.pon` in HadOCC are not very important. In order to find out more, one could use the input to intermediate variable emulators to produce intermediate variable designs where `log.iz.pz` and `log.iz.dz` were less highly correlated.

Although `iz.pn` is the most influential intermediate variable in both OG99NPZD and HadOCC, focussing on their use of `iz.pd` and `iz.dn`, and investigating whether it is appropriate for the zooplankton related intermediates to be more influential (as in HadOCC) or less (as in OG99NPZD) would best help understand the differences between them at this level.

As Table 6.10 shows, very few input variables in either simulator are strongly active in `iz.pn`. The apparent link between μ_P (in OG99NPZD) and `presp` (in HadOCC), which affect `iz.pn`, `iz.pd` and most other intermediates very similarly, remains important. Some of the inputs that are active in HadOCC's calculation of `iz.pn` (for example `photmax`, `alphachl` and `kdin`) have apparent links with OG99NPZD inputs that do not appear active in `iz.pn`. Studying the different uses of these input variables may help to further understand differences between OG99NPZD and HadOCC.



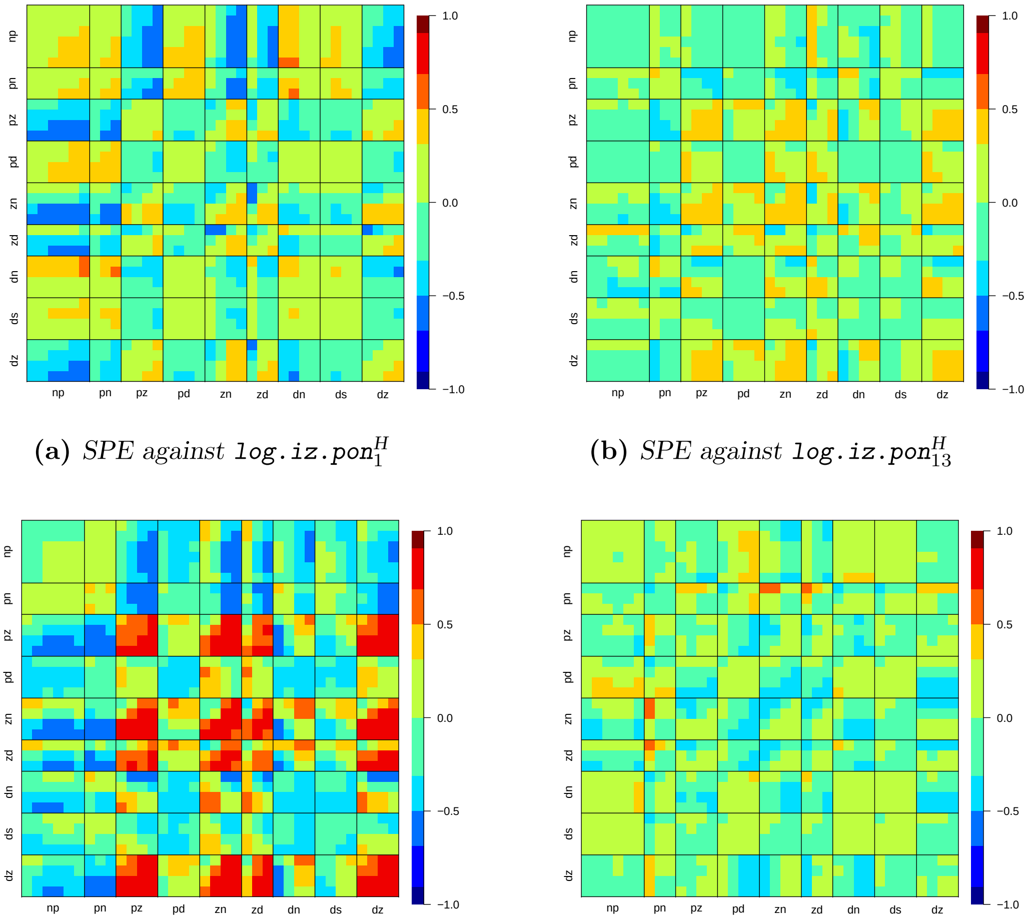
(a) *log.iz.pon*$^H_1$

(b) *log.iz.pon*$^H_{13}$

(c) *log.iz.pon*$^H_{105}$

(d) *log.iz.pon*$^H_{365}$

Figure 6.21: *Correlations between each intermediate variable in HAD1007 and the SPE using an emulator built from OG948.*



(a) *SPE against* $\texttt{log.iz.pon}_1^H$

(b) *SPE against* $\texttt{log.iz.pon}_{13}^H$

(c) *SPE against* $\texttt{log.iz.pon}_{105}^H$

(d) *SPE against* $\texttt{log.iz.pon}_{365}^H$

Figure 6.22: *Correlations between products of pairs of intermediate variables in HAD1007 and the SPE using an emulator built from OG948.*

Conclusions

To summarise what has been found about the differences between HadOCC and OG99NPZD:

- The RMSE and SPE values found when using an emulator of one simulator over data from the other suggest strong systematic differences between the two simulators.
- The correlations between `iz.np` and `iz.pn` in OG947 or HAD1007 and the SPE from the errors using an emulator of the other simulator are very low,

implying that the two simulators use these intermediate variables in a similar way.

- High correlations between `iz.pd` and the latter two points of `iz.dn` with SPE for both datasets suggest that these intermediate variables are used differently in the two simulators.
- When the OG99NPZD emulator is used over HAD1007, the zooplankton related intermediates are quite strongly correlated with the SPE. This consolidates the conclusion from step 13 that these intermediates are more influential in HadOCC.

Summary

This stage has improved our understanding of how the different intermediate processes contribute to OG99NPZD and HadOCC, and where differences lie. Studying the correlations between intermediate variables and output gave an indication of the most important intermediate variables and of their effects. Using the emulator of one simulator over the intermediate variable data of the other, made possible by the formation of the same intermediate variables in each simulator, enables a direct analysis of how similarly the two simulators use their processes. Both of these steps enabled us to identify possible avenues for further investigation of differences between OG99NPZD and HadOCC.

6.7 Further directions

This chapter has covered the process of using intermediate variable emulation to compare two simulators and better understand each of them. Various aspects of the methods shown could be developed, refined or augmented to bring improvement. Some possible areas for further work are discussed here.

Dimension reduction

Principal variables have been used in this chapter, largely because they are relatively robust to different trends and patterns in the data, and simple to interpret in the latter stages of intermediate variable emulation. In some cases it might be that a functional approximation, for example a smoothing spline, might much better capture the features of the intermediate variable data, and allow for more information to be retained. This requires careful study of how the method will be chosen using the data, how the parameters will be interpreted, and of situations in which using an alternative dimension reduction method would be particularly beneficial.

Choice of dimension reduction technique may also depend on the overall goal of emulation. If a particular output variable is of interest, it may be better to choose a subset of the full intermediate variables that best predicts this value, rather than choosing the subset that best represents the intermediate variables. This may involve purely data analytic techniques, but it could make use of understanding of the simulators. For example, if only the previous time point is used at each stage in the process, the intermediate variables from the time point before the one in question might be a good choice.

Experimental design in intermediate variable space

It has been noted that one of the main difficulties in emulating the output from the intermediate variables is our lack of control over the intermediate variables, and therefore our inability to specify a design over that space. While this is inevitably true (unless the simulator itself can be re-written to take intermediate variables as inputs), history matching techniques could be used, along with the emulators from input to intermediate variables, to create an approximate design.

In order to be able to specify the nature of the intermediate variable space, the full set of intermediate variables would have to be jointly emulated from the input variables, which is not the case in this chapter. There is also no guarantee that all regions of the intermediate variable space can be filled; the values of intermediate variables may be inherently linked in such a way that prevents, for example, a high value of one and a low value of another. This may in itself be interesting, particularly if the limitations are different for the different simulators.

Where two intermediates are generally highly correlated, it may be possible to use the input to intermediate variable emulators to produce data where this is not the case. This could then provide more information about the effects of these variables, in situations like that of `iz.pz` and `iz.dz` in HadOCC, whose effects are difficult to tell apart in the example in Section 6.6.3.

Model selection for intermediate to output variables

In the examples in this chapter, the regression surfaces were built using the stepwise selection procedure in R (R Development Core Team, 2011), searching by adding and deleting terms, and allowing squared and second-order interaction terms. However this has certain shortfalls, particularly in being unable to choose between highly correlated variables, and therefore potentially leading to spurious conclusions if the effects of some variables are attributed solely to one. Existing work on model selection in regression problems with highly collinear input variables could be used here.

One potential solution would be to emulate the output using the principal components of the intermediate variables, rather than the intermediate variables themselves (as in principal component regression). Although this might seem more difficult to interpret, the diagnostics and plots used in the example in Section 6.6.3 are all also possible with a principal component emulator.

When one simulator is ‘better’

In this chapter the simulators are being compared without one being judged to be more reliable or accurate. However, often it may be the case that one generally performs better against observed data, or has had much more time and effort invested in it, than another. In this case, intermediate variable emulation could perhaps be viewed more as a tool for using the ‘better’ simulator to inform the other. In particular, the ranges and behaviour of the intermediate variables could be viewed as standards, and used to deduce sensible input ranges for the other. This leads into the much broader area of using expert knowledge in order to interpret the findings of intermediate variable emulation.

As a general emulation strategy

It has been mentioned throughout that intermediate variable emulation can be of use in a single simulator context. This has mostly been related to an increased understanding of the simulator through use of intermediate variables. However, there might be situations in which intermediate variable emulation is a better strategy than standard emulation. Appendix C.3 explores the idea of combining the input to intermediate and intermediate to output variable emulators to form an emulator from input to output variables. Figures C.8 and C.9 show samples from the combined intermediate variable emulators.

Because intermediate variable emulation splits the simulator into different stages, and offers more flexibility in how the individual emulators are constructed, it may be better equipped to deal with simulators that contain complicated relationships for particular processes.

6.8 Summary

This chapter has presented *intermediate variable emulation*, a method enabling emulation of multiple simulators of the same system in a way that improves understanding of each, and facilitates comparison. Methods have been illustrated throughout using OG99NPZD and HadOCC. Some pairs of highly active input parameters, given different meanings in HadOCC and OG99NPZD, were shown to affect almost all intermediate variables similarly, therefore suggesting links between the two input spaces. Other inputs that are unique to one simulator were shown to be largely inactive, lessening the motivation to link the input spaces in full.

Emulators from the intermediate to output variables showed that there are systematic differences between the two simulators. The transfer of nitrogen from phytoplankton to nutrient, `iz.pn`, is the most active in both HadOCC and OG99NPZD, and appears to be treated very similarly. Other transfers, particularly `iz.dn` and `iz.pd`, appear to contribute quite differently to the two simulators.

Unlike hierarchical emulation, intermediate variable emulation does not require that the simulators' input spaces be almost the same, but instead makes use of similar process represented in each simulator, using them to create a set of 'intermediate' variables. By analysing the distributions and trends of the intermediate variables, differences in the general behaviour of the simulators can be understood.

For each simulator, the input variables can be used to emulate the intermediate variables, enabling a detailed study of the relationships between the input spaces. Using expert knowledge of the system, unrealistic values of the intermediate variables can be used to refine the input spaces through history matching and similar techniques.

Emulators of the output variables from the intermediate variables can also be created for each simulator. Although the intermediate variable spaces are likely to present difficulties for emulation because of their irregular shapes and collinearity, having emulators with the same input and output variables for all simulators enables direct comparison. Not only can the effects of the intermediate variables on the output be observed for each simulator, the emulator of one simulator can be used to predict the behaviour of another. Studying the behaviour of the errors for these predictions reveals the key systematic differences between the simulators' representations of the system.

Chapter 7

An object-oriented structure for emulation

Up to this point, the focus has been on methods for emulation, rather than on their implementation. Because of the quantity of data and the number of operations involved in building emulators, the only feasible approach is to program. For this thesis, all emulation was done in R (R Development Core Team, 2011), and in an object-oriented way using the S4 class structure. In this chapter we explore the benefits of object-oriented programming and apply them specifically to emulation. First of all, we motivate object-oriented programming, and then introduce the S4 classes in R. A framework for emulation is then presented, and extended to incorporate the new methods from Chapters 5 and 6.

7.1 Why use objects?

In this thesis, methods for emulation have been presented that use large amounts of simulator data, perform many calculations, and result in large collections of results. Many collections of the same sort of data or results may be stored, and may need to be accessed by different people or after long breaks, and so the potential for mistakes and inefficiency is high. For instance, time consuming calculations such as finding the inverse or Cholesky decomposition of the covariance matrix of the correlated errors (the matrix $\Sigma(\mathbf{x})$ in the notation of Chapter 3) or estimating the correlation lengths (see Section 3.3.3) may be repeated often as new techniques are tried, or even as the same operations are carried out at different times or by different people. The details of the correlation or regression surface associated with a set of predictions may be lost or confused.

It is also likely that as understanding of the problem develops and new techniques are devised, existing code will need to be adapted to deal with new sorts of simulator data, or to perform new tasks. For example, code that performs standard emulation as in Chapter 3 may need to be extended to be able to use a new correlation function or to perform hierarchical emulation (Chapter 5) or intermediate variable emulation (Chapter 6). Ideally, this would not require a new set of functions written entirely from scratch, but could be built on an existing foundation. Object-oriented programming (OOP) addresses each of these issues.

Rather than focus on functions, OOP revolves around tightly structured objects and their interactions. In OOP, information that belongs together is *encapsulated* as one object. All objects belong to a particular *class*, and classes have strict definitions; knowing the class of a particular object means knowing exactly what each part of the object is, and how the different components relate to one another. The structure of the data is maintained without any part being lost, a feature that is not guaranteed when components are stored separately. In emulation, there may be delays between designing an experiment, running a simulator, building emulators and making predictions, and several similar processes may be ongoing at once. An object-oriented structure ensures that no information is confused or lost.

Classes can be related to one another through *inheritance*. Alfons et al. (2010) describe this as one of the main advantages of object-oriented programming. Inheritance allows *sub-classes* to inherit their structure and behaviour from their *super-class*, each sub-class extending the super-class in some way. Thus several classes may be created representing fundamentally the same sort of information, but each in a slightly different way, or with extra features.

Outside of OOP, in order to deal with different forms of the same sort of data, functions must contain many checks to discern the meanings and features of their arguments each time they are evaluated, in order to know how to behave. Another advantage of OOP is *multiple dispatch*. This streamlines the way functions are made and used. A *generic function* is created representing the goal or task at hand, and *methods* are written for this function, dispatching on various combinations of classes of arguments, or *signatures*. When the function is called, the classes of the arguments are checked against the signature of each method, and the correct method is dispatched. Many methods can be written for each function, so that the same task can be performed using the same function with any manifestation of the same sort of information, so long as the relevant methods have been written.

This helps enormously with maintaining and adapting code. Suppose one has a framework for emulation for a particular sort of simulator data, held in objects of a particular class, and that throughout the code the simulator data is handled using functions with methods defined for that class. In the event that another class is created, containing a slightly different form of simulator data, one can simply write new methods for each function to be able to handle the new class, and any code using these functions will still work. In a non object-oriented setting, it can be much more difficult to adapt code to deal with such a fundamental change. Methods also provide flexibility in allowing objects of the same class to be created from various different combinations of arguments.

OOP is also often preferred at an ideological level. Rather than think in terms of long sequences of instructions and procedures starting with primitive items, it is posited by many that people generally think in terms of meaningful objects, and interesting operations one might want to perform on them. Leisch (2004), who holds this view, uses the example of probability distributions. He argues that it is intuitive to store pdf's and cdf's as objects, and to define operations on them, for example the mean, variance, random sampling or some sort of plot, rather than to write separate functions for each operation and distribution.

As the S4 emulation framework is presented in Section 7.3 and extended in Sections 7.4 and 7.5, the benefits of OOP for emulation will become clearer. Before outlining this framework, we will introduce the S4 class structure in R (R Development Core Team, 2011; Chambers, 1998).

7.2 S4 objects

In R, data is organised using *classes*. The most prevalent class system is S3, or 'old-style', which includes classes such as "lm". These are not rigid in their definition; different instances of a particular S3 class may include different parts. Although this isn't necessarily a problem if programming is done diligently and structure is enforced by the writer, it can make dealing with complicated data structures difficult and messy.

The S4 or 'new-style' class system is far more rigid in its approach. When an S4 class is defined, the user must specify its *representation*; exactly what components, known as *slots*, make up an instance of this class. In the class representation each slot is given a name and assigned a class. R will simply not allow an object of a certain class to be created if its components don't fit this specification. While this can be frustrating, it ensures that all instances of a particular class have exactly the same structure, and that the programmer is fully aware of this structure. Further validation criteria can be added if necessary, so that the values of the slots must fit certain constraints.

For any class, a *sub-class* can be defined, extending it. A sub-class must contain at least the same slots as its *super-class*, the class it extends, and they must have the same classes. The sub-class can contain further slots, enabling more information to be stored. Through inheritance, any function that can take an instance of the super-class will also accept an object of the sub-class, although new methods can also be written that work only for the sub-class. This will be explored later.

Having defined classes, one can write *methods* enabling functions to dispatch on different combinations of objects. Methods enable the same function call to behave differently depending on the arguments it is given. Multiple dispatch is a common occurrence in R, even in S3; the function `plot` will behave differently given two vectors from when it is given a time-series or a single vector. Similarly, methods can be defined in S4 for different combinations of inputs. There is actually more control here, since under S3 the class of the first argument alone determines the method, whereas in S4 the *signature* (the list of classes of the arguments) can be of any length.

Todorov and Filzmoser (2009) present an object-oriented structure for multivariate analysis using S4 classes in R, providing helpful insight. They also explain how OOP enables a user to much more easily adapt their code in order to introduce a new function, or to enable existing functions to deal with a new sort of object. A presentation by Leisch (2004) uses the example of classes representing images to demonstrate inheritance and the writing of functions, with many illustrative examples of code. Different classes can be created to store different types of image, for example bitmap, JPEG or SVG, and there may be different subclasses depending on whether the image is black and white or colour. Functions using the images (for example to plot the image, or to convert it to pdf format) can be made to behave appropriately whatever type of object they are given, by writing methods for the different classes.

For more general information about object-oriented programming in R using S4, see Chapters 7 and 8 of Chambers (1998) or Chapter 5 of Venables and Ripley (2000).

We are now in a position to develop a framework for emulation using S4 objects. First of all, this will be for standard emulation as described in Chapter 3.

7.3 Emulation using S4

Broadly speaking, there are three stages to emulation. Firstly, one needs simulator data with which to train the emulator. Secondly, the regression surface and correlation function must be specified, and used with the training data to build an emulator. Finally, the emulator can be given new input data, and asked to predict the simulator's output there.

Because each stage involves structuring data, and large amounts of information belonging together, it seems appropriate to tackle the process in an object-oriented way. It is possible that there might be large time intervals between each of these stages, or that one might want to revisit parts of the emulation process to alter some specifications, and so containing all the information relevant to that stage in one highly structured object is an attractive concept. The core emulation structure

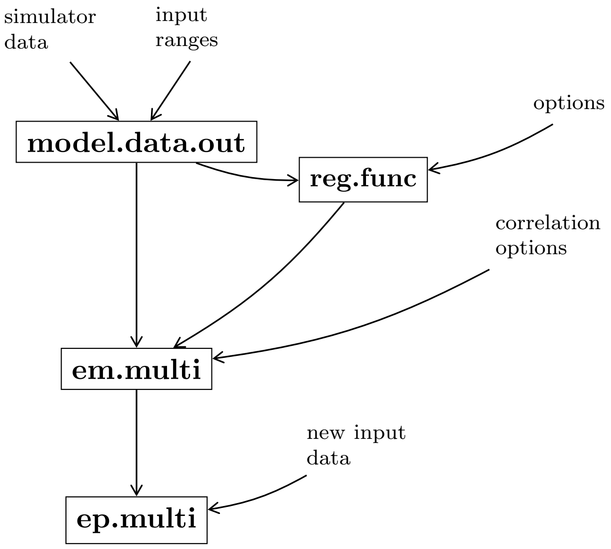


Figure 7.1: *An S4 object structure for emulation. The names in bold denote classes, and the other text describes the inputs used to make objects of those classes. The arrows represent the information needed to create an object of each class.*

used here is shown schematically in Figure 7.1, and code for the core structure is given in Appendix D.

The classes defined in this chapter can mostly be categorised into three groups, mirroring the three stages mentioned above: simulator data, emulator, and prediction. In this section, the framework developed to build a 'standard' emulator (as in Chapter 3) is explained. It will then be extended to cater for the extra needs of hierarchical emulation (Chapter 5) and intermediate variable emulation (Chapter 6). Throughout, a "class" is written in speech-marks, and a `function` in typewriter font.

Simulator input data is stored in instances of "model.data", where the values and possible ranges (usually provided by an expert) of each input are slots. The sub-class "model.data.out" also has slots containing a data frame of simulator output and a corresponding vector of output names. Instances of these classes are created using the function `model.data`, it being common practice in the S4 framework to name a creator function after the class (or super-class) of the object it creates. These, and the other classes described in this section, are detailed in Tables 7.1 and 7.2.

The method dispatched for the creator function `model.data` depends on the classes of the arguments it is given. If it is given two data frames, one containing input points and one of corresponding input ranges, an instance of "model.data" is created. If a vector of output names is also given, matching columns of output data in the data frame containing input points, a "model.data.out" object is made. These two classes are related by inheritance, and so any method that can dispatch on a "model.data" object will also dispatch on a "model.data.out" object[1]. The converse is not true.

Various methods exist for accessing parts of these objects, for example for creating a data frame of input and output together from a "model.data.out" object. Another function, `rescale`, uses the ranges and the input data to rescale the inputs to a particular interval, (the interval $[-1, 1]$ by default) and returns the data frame of rescaled values.

At the third stage (prediction), an emulator is used to create a probability distribution, and so an object of class "em.multi", created in the second stage, contains everything necessary to evaluate this distribution at new input points. The training data is included, via a slot of class "model.data.out", and any information needed for the regression surface and correlated error. These details are stored in two more classes of object, "reg.func" and "corr.mats".

These classes were introduced to allow flexibility in how the emulator is constructed, and to facilitate changes to the emulator once it has been built. The regression function is stored in the "reg.func" object, which can be created in several ways. At the most basic, one may specify a list of functions to be used. Options exist to use all linear terms or build a full second order surface. It is also possible to specify a desired number of active variables, in which case the method calls a function written to find the 'most active' of the inputs. There are currently several other options, including step-wise model selection with various constraints, but whichever approach is taken, the resulting "reg.func" object contains exactly the same slots

[1] When a function is called, and the arguments match signatures for more than one method, a hierarchy is in place. This means that the method using "model.data.out" will take precedence over one using "model.data". The class "ANY", which can admit an argument of any class, takes the least priority.

with exactly the same meanings, and can be used to create an "em.multi" object in the same way regardless of how it was made.

Similarly, the "corr.mats" object, which represents the correlated error function, can be formed by specifying one correlation length, or a length for each input. In the emulation process outlined in Figure 7.1, the "corr.mats" object is concealed within the "em.multi" object. When correlation lengths are specified numerically, the "corr.mats" object can be created on its own, from a "model.data" object and these lengths. However, often optimisation is used to find the correlation lengths, and in this case the regression functions must already be specified. Therefore the "corr.mats" object is usually created within the `em.multi` creator function, rather than on its own, using options (either numerical correlation lengths or optimisation criteria) passed to `em.multi`.

The "corr.mats" object contains either the Cholesky decomposition or the inverse of the correlation matrix, one of which must be available at later stages in the emulation process. Indeed, at the "em.multi" stage, any calculation necessary for prediction that does not require the new input points can be performed once and stored, rather than being repeatedly performed at the prediction stage.

Predictions take the form of an "ep.multi" object. This is created using an "em.multi" object and a data frame of new input points. The "ep.multi" object contains the location, scale and degrees of freedom of the output's posterior distribution, the t-distribution summarised in Section 3.1.

It might seem immediately more intuitive to have a prediction function, rather than a prediction class, which takes an "em.multi" object and new inputs and returns summaries of the t-distribution. Indeed, the creator function `ep.multi` can be used in this way. However, a prediction class brings with it extra advantages. An "ep.multi" object has a slot containing the "em.multi" object used, and so the information used for these predictions is stored with them, avoiding confusion or error. Furthermore, encapsulating the prediction information in a single object makes interaction with the information much simpler. One might want to sample from the distribution, draw particular plots or give certain summary statistics. Methods could also be developed for the prediction classes in the hierarchical and intermediate variable emulation frameworks. The “ep.multi” class is also used throughout the hierarchical and intermediate variable emulation frameworks, both of which involve relating various standard emulators to one another in a rigid structure.

7.4 Hierarchical emulation in S4

Having established a core object-oriented structure for emulation in R we can extend it to build hierarchical emulators, which were described in Chapter 5. The core structure revolves around three sorts of class; data, model and prediction. For the standard emulation framework outlined in Section 7.3, these classes are “model.data”, “em.multi” and “ep.multi”. In hierarchical emulation, the same three stages are present, but the structure of the emulator requires some new functionality, and therefore new classes.

The initial problem with hierarchical emulation is to organise the training data into its separate blocks, each informative for only one of the functions. The data class in the hierarchical setting is “hier.data”. Like “model.data” this is made using a data frame of inputs (and optionally outputs) and a data frame of input ranges. To enable the hierarchical features, it must also be given the names of the hierarchical variables, along with their v^* values and $g\left(\cdot\right)$ functions, and the names of any extra variables w. The data is checked to make sure that the design criteria discussed in Section 5.2.2 are satisfied, and organised into its separate blocks, according to the function, $\psi\left(\cdot\right)$ or $s_0\left(\cdot\right)$, for which it is informative. The $g\left(\cdot\right)$ functions, hierarchical variables and output data are then used to find the $\psi\left(\cdot\right)$ data. These blocks are then used to create a list containing a “model.data” object[2] for each term in the emulator. As well as the simulator output variable, each of these also has an output ‘h’, which is the data from the $\psi_{[i]}\left(\cdot\right)$ function (or s_0 for the first block), calculated using the data and the structure information. The emulators built for each $\psi_{[i]}\left(\cdot\right)$ function will use h as their output.

The class “hier.model” corresponds to “em.multi”, and requires the same choices about the regression and correlation functions, along with an object of class

[2] This can also be an object of any subclass of “model.data”, for example “model.data.out”.

Class	Slots	Slot's class	Description
model.data	input	data.frame	Input points
	oldrange	data.frame	Ranges of inputs
model.data.out (extends model.data)	input	data.frame	Input points
	oldrange	data.frame	Ranges of inputs
	outdf	data.frame	Output data
	outname	vector	Names of output variables
em.multi	data.obj	model.data.out	Training data object
	names.out	character	Name(s) of output variables
	reg.obj	reg.func	Regression surface object
	cm.obj	corr.mats	Correlated error object
	HcmH	matrix	The matrix $\left(\mathbf{X}A^{-1}\mathbf{X}\right)$ in Section 3.2
	chol.HcmH	matrix or try-error	The cholesky decomposition of HcmH (or an error if this fails)
	beta.gls	matrix	The GLS estimate of the regression coefficients
	sigma.gls	matrix	The GLS estimate of output covariance matrix ($\hat{\Gamma}$ in Section 3.2)
ep.multi	mod	em.multi	The emulator object
	xnew	data.frame	New input points
	loc	matrix	Location of predicted output's t distribution
	scale	array	Scale of predicted output's t distribution
	deg.f	numeric	Degrees of freedom of predicted output's t distribution

Table 7.1: *Classes used in the core emulation structure, with details about each of their slots.*

Class	Slots	Slot's class	Description
reg.func	functions	list	List of regression functions
	active	vector*	Names of active variables
	summary	summary.lm*	Summary of regression surface
	priormean	vector*	Mean of regression coefficients
	priorvar	vector*	Variance of regression coefficients
	options	list*	The options used to build the regression surface
	data	model.data*	The model data object used to build the regression surface
corr.mats	data1	model.data	Data for correlation matrix (n points)
	data2	model.data*	Optional second data set (m points)
	corrlen	data.frame	Correlation lengths for each input dimension
	corrmat	matrix	Correlation matrix ($n \times m$, or $n \times n$ if 'data1' is null)
	cholcm	matrix*	Cholesky factorisation of 'corrmat' if it can be found
	cminv	matrix*	Inverse of the correlation matrix 'corrmat', if 'cholcm' cannot be found
	nugget	numeric	Variance for a nugget term (zero by default)

Table 7.2: *Classes representing the regression surface and the correlated error, with details about each of their slots. An asterisk in the 'class' column indicates that this slot can also have class "null".*

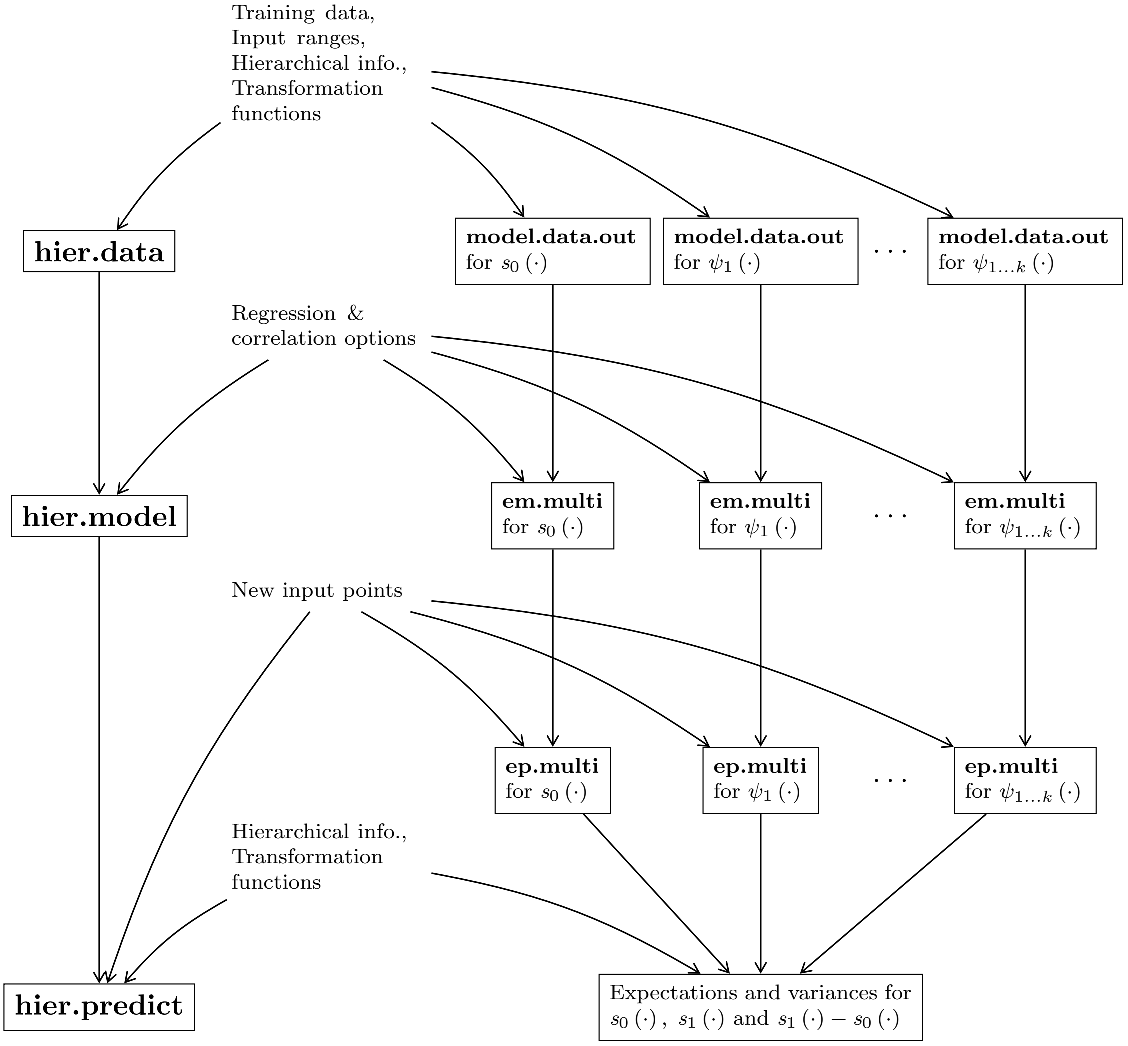

Training data,
Input ranges,
Hierarchical info.,
Transformation functions
hier.data
model.data.out for $s_0(\cdot)$
model.data.out for $\psi_1(\cdot)$
...
model.data.out for $\psi_{1\ldots k}(\cdot)$
Regression & correlation options
hier.model
em.multi for $s_0(\cdot)$
em.multi for $\psi_1(\cdot)$
...
em.multi for $\psi_{1\ldots k}(\cdot)$
New input points
ep.multi for $s_0(\cdot)$
ep.multi for $\psi_1(\cdot)$
...
ep.multi for $\psi_{1\ldots k}(\cdot)$
Hierarchical info.,
Transformation functions
hier.predict
Expectations and variances for $s_0(\cdot)$, $s_1(\cdot)$ and $s_1(\cdot) - s_0(\cdot)$

Figure 7.2: *An S4 object structure for hierarchical emulation, in terms of the hierarchical emulation classes (left), and standard emulation classes (right). The names in bold denote objects of classes from this chapter, and the other text describes the inputs used to make those objects. The arrows represent the information needed to create each object.*

"hier.data". The object then contains the "hier.data" object and a list of "em.multi" objects, one for each of the "model.data" objects in the "hier.data" object's list. Having established the standard emulation framework therefore, this stage of hierarchical emulation is simple.

Finally, objects of class "hier.predict" are created using a "hier.model" object and a data frame of new inputs. The "em.multi" objects from the list in the "hier.model" object are used to combine these predictions to predict the values of $s_0(\cdot)$ and the $\psi_{[i]}(\cdot)$ at each of the new points, by creating a list of "ep.multi" objects. The $g(\cdot)$ functions and hierarchical inputs stored in the "hier.data" object are then used to produce vectors "loc.s1", "loc.s0" and "loc.diff" and matrices "var.s1", "var.s0" and "var.diff", which give the expected values and variances of s_1, s_0 and $s_1 - s_0$ respectively.

Figure 7.2 shows the flow of information through the hierarchical emulation framework, in terms of both the specialised hierarchical emulation classes and the standard emulation classes explained in Section 7.3.

At present, the hierarchical emulation structure works for univariate output only, as it is described in Chapter 5. Extending it to multiple output variables is possible, but introduces new questions about various aspects of the process, for example whether to use the same transformation function for all outputs for each hierarchical input, or to find the optimal function for each output.

7.5 Intermediate variable emulation in S4

As with hierarchical emulation, one of the main challenges in adapting the emulation framework to intermediate variable emulation is in the structure of the data; once this is incorporated into the code emulation is fairly straightforward. The classes created for intermediate variable emulation are shown in Table 7.4, and the flow of information between the classes is shown in terms of the intermediate variable classes and the standard emulation classes in Figure 7.3.

In order to retain flexibility in the choice of dimension reduction technique, the structure presented here deals with the dimension reduced intermediate variables.

Class	Slots	Slot's class	Description
hier.data	data.list	list	List of model.data.out objects containing training data and output for each level of the hierarchy
	hier.inputs	character	Names of hierarchical variables
	extra.var	vector*	Names (if any) of extra variables
	cond.vec	vector	v^* values for each hierarchical variable
	trans.fun	list	List of transformation functions, one for each hierarchical variable
hier.model	data.obj	hier.data	The training data object
	model.list	list	List of em.multi objects, one for each term in the emulator
hier.model	md.new	model.data	New input points ($\tilde{\mathbf{x}},\ \tilde{\mathbf{v}},\ \tilde{\mathbf{w}}$)
	model.obj	hier.model	Emulator object
	predict.list	list	List of ep.multi objects, one for each term in the emulator
	loc.s0	vector	Location for $s_0\left(\tilde{\mathbf{x}}\right)$
	loc.s1	vector	Location for $s_0\left(\tilde{\mathbf{x}}, \tilde{\mathbf{v}}, \tilde{\mathbf{w}}\right)$
	loc.diff	vector	Location for $s_0\left(\tilde{\mathbf{x}}\right) - s_0\left(\tilde{\mathbf{x}}, \tilde{\mathbf{v}}, \tilde{\mathbf{w}}\right)$
	var.s0	matrix	Scale for $s_0\left(\tilde{\mathbf{x}}\right)$
	var.s1	matrix	Scale for $s_0\left(\tilde{\mathbf{x}}, \tilde{\mathbf{v}}, \tilde{\mathbf{w}}\right)$
	var.diff	matrix	Scale for $s_0\left(\tilde{\mathbf{x}}\right) - s_0\left(\tilde{\mathbf{x}}, \tilde{\mathbf{v}}, \tilde{\mathbf{w}}\right)$

Table 7.3: *Classes used in the hierarchical emulation structure, with details about each of their slots. An asterisk in the 'class' column indicates that this slot can also have class "null". At present, this only works for univariate output.*

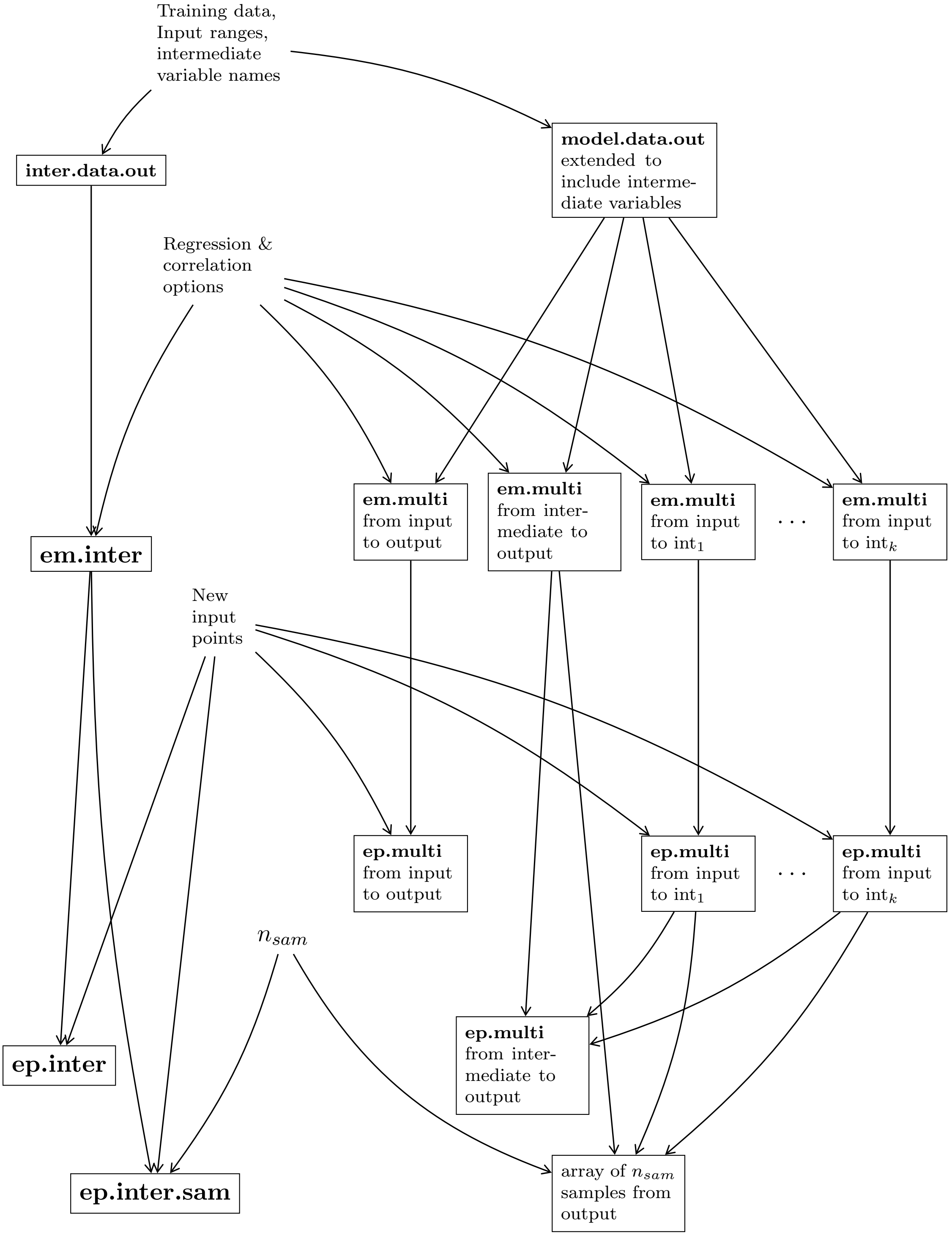

Training data,
Input ranges,
intermediate
variable names
inter.data.out
model.data.out extended to include intermediate variables
Regression & correlation options
em.inter
em.multi from input to output
em.multi from intermediate to output
em.multi from input to int_1
· · ·
em.multi from input to int_k
New input points
ep.multi from input to output
ep.multi from input to int_1
· · ·
ep.multi from input to int_k
n_{sam}
ep.inter
ep.multi from intermediate to output
ep.inter.sam
array of n_{sam} samples from output

Figure 7.3: *An S4 object structure for intermediate variable emulation, in terms of the intermediate variable emulation classes (left), and standard emulation classes (right). The names in bold denote objects of classes from this chapter, and the other text describes the inputs used to make those objects. The arrows represent the information needed to create each object, and* $\text{int}_1, \ldots, \text{int}_k$ *are intermediate variables.*

Functions could be written to apply various techniques to raw data in order to produce dimension reduced data in order to begin this process. The framework outlined here will work for principal variables, summaries of a smoothing spline, coefficients of orthogonal polynomials or any other summary, so long as functions are written which reduce the dimension of the initial data, and then rebuild output if necessary from the predictions.

Simulator data is stored in objects of class “inter.data.out”, which extends “model.data.out”. As well as storing input and output data, instances of “inter.data.out” store the intermediate variable data in a slot named ‘interdf’, the intermediate variable names in ‘internames’ and their ranges in ‘interranges’. The names of the intermediate variables in the ‘interdf’ data frame must each begin with one of the ‘internames’, as this will be used to organise the variables later in the process. Because this class extends the data classes in the core structure, functions such as `rescale` or `out.name`, with methods for “model.data” and “model.data.out”, will work for them through inheritance.

Setting the ranges of intermediate variables is not as simple as in standard emulation. Because the values of intermediate variables cannot be chosen in the same way as inputs, their ranges cannot be set, and so the range slot here contains approximate ranges formed using the data, rather than strict ones. Because they are only used to rescale data, it is not crucial that all intermediate variable values are within these ranges; unless the values can extend far outside the reach of the training data, the rescaled data will be approximately within the correct interval.

An intermediate variable emulator, which has class “em.inter”, is formed by combining an “inter.data.out” training data object with choices about the regression surfaces and the correlated error. This object is built from many “em.multi” objects. The slots ‘em.in.out’ and ‘em.inter.out’ hold emulators of the output variables from the inputs and the intermediate variables respectively. These are formed by passing on the regression and correlated error choices with the relevant data to create separate “em.multi” objects. Because of the number of separate “em.multi” objects to be built at this stage, programming and usage are much simpler if criteria for the regression surface and correlated error are given, rather than specific functions and correlation lengths.

The slot ‘in.inter.list’ contains an “em.multi” object for each element of the ‘internames’ slot of the “inter.data.out” object. In the example in Chapter 6, this feature was used to group the intermediate variables by nitrogen transfer. By including ‘iz.dn’ in ‘internames’, for example, $\texttt{iz.dn}_{3}^{\mathrm{OG}}$, $\texttt{iz.dn}_{11}^{\mathrm{OG}}$, $\texttt{iz.dn}_{110}^{\mathrm{OG}}$ and $\texttt{iz.dn}_{243}^{\mathrm{OG}}$, whose names in the ‘input’ table all began with ‘iz.dn’, were jointly emulated in one “em.multi” object. Had their full names each been included in ‘internames’, they would have been emulated with four separate “em.multi” objects. Alternatively, having only one element in ‘internames’, with which the names of all columns in ‘interdf’ began, would cause all intermediate variables to be jointly emulated.

Predictions of the simulator’s behaviour at new input points are held in objects of classes “ep.inter” and “ep.inter.sam”. The first, “ep.inter”, simply uses the new inputs and the “em.multi” objects in an “em.inter” object to create corresponding “ep.multi” objects. Because new intermediate points are not given with the inputs, the ‘ep.inter.out’ slot is formed using the predictions from the “ep.multi” objects in the ‘in.inter.list’ slot. The variance attributes of ‘ep.inter.out’ are therefore conditional on these predictions being correct. The various “em.multi” and “ep.multi” objects can each be accessed and analysed in order to apply the techniques described in Chapter 6.

In order to give access to the distribution of the predictions formed by using inputs to predict intermediate variables which are then used to predict the output, the “ep.inter.sam” class was created. This requires new input points, an “em.inter” object and a number n_{sam}. It then generates a sample of size n_{sam} from the emulator’s output distribution, by generating n_{sam} points in intermediate space for each input (using the ‘in.inter.list’ slot), and then sampling once from the ‘ep.inter.out’ slot for each of these points. The resulting output values are stored in the “array” slot ‘loc.inter.out’.

Class	Slots	Slot's class	Description
inter.data.out (extends model.data.out)	input	data.frame	Data frame of input values
	oldrange	data.frame	Ranges of input variables
	interdf	data.frame	Data frame of intermediate variable values
	internames	vector	Names of intermediate variables.
	interrange	data.frame	Ranges of intermediate variables.
	outdf	data.frame	Output data
	outname	vector	Names of output variables
em.inter	em.in.out	em.multi	An emulator from input to output
	in.inter.list	list	A list of em.multi objects of emulators from input to intermediate variables, one for each element of internames.
ep.inter	em.inter.out	em.multi	An emulator from intermediate to output
	ep.in.out	ep.multi	A prediction object for new inputs, using em.in.out
	in.inter.list	list	A list of prediction objects for new inputs, one for each of the em.multi objects in the in.inter.list slot of the em.inter object
	ep.inter.out	ep.multi	A prediction object for new inputs, using em.inter.out, and the predicted values of in.inter.list as intermediate variables.
ep.inter.sam	ep.in.out	ep.multi	As in “ep.inter”
	in.inter.list	list	As in “ep.inter”
	loc.inter.out	array	An array of sampled predicted outputs, created by sampling intermediate variables from the in.inter.list predictions, then using these with the em.inter.out to sample from the output distribution.

Table 7.4: *Classes used in the intermediate variable emulation structure, with details about each of their slots. An asterisk in the 'class' column indicates that this slot can also have class “null”.*

7.6 Efficiency and versatility

One of the main advantages of the object-oriented emulation framework is the efficiency arising from the use of the same classes throughout. For example, the “em.multi” and “ep.multi” classes feature in both the hierarchical and intermediate variable emulation frameworks. In both settings, once data has been appropriately organised, the procedure is mostly reduced to building a collection of standard emulators. This cuts down the amount of code to be written for creating and interacting with the objects. By collecting similar sorts of information into identically structured objects, the code ensures that, for example, an “em.multi” object will always work in the same way, and be able to be used by the same functions, whether it is on its own or part of a “hier.model” or “em.inter” object. This is particularly useful in intermediate variable emulation when using the intermediate to output variable emulator of one simulator on data from another, as in Section 6.6.3.

Encapsulating everything relating to one aspect of the problem in an object also makes adding new features to the code very simple. The use of an object does not depend on how it was formed, but purely on its class; although the objects mentioned throughout this chapter can sometimes be created in various different ways, objects of the same class will always have the same features.

For instance, functionality could be added for the Matérn correlation function by altering the creator function `corr.mats`, and the resulting “corr.mats” objects will work wherever `corr.mats` is used throughout the framework.

The ability to write multiple methods for any function also leads to versatility in how objects are created. There may be several possible combinations of data and options that enable the creation of a particular type of object. A simple example is the creator function `reg.func`, which has up to three arguments, and creates an object representing the regression surface. At the simplest, `reg.func` accepts a list of regression functions, so the signature is

```
"list", "missing", "missing".
```

Using the class “missing” enables a function to accept just one argument, even though the generic function definition lists three. This then creates a “reg.func” object containing these functions in its ‘functions’ slot (see Table 7.2). All other slots are “null”, because no data was given with which to provide summaries of the surface.

If instead the arguments matched the signature

```
"list", "model.data.out", "missing",
```

one of two things may happen. If the list contains functions, as before, then these will be used as regression functions, and the data will now be used to calculate values for the other slots. Otherwise, the list should be a list of options specifying the nature of the surface to be built. One can choose to include all first order terms, or to use step-wise selection to find a second order surface. A desired number of active variables can be given, in which case some different techniques can be employed to find the most active. Any of these choices produces an object with exactly the same features as if the list of functions had been given.

The final argument allows one to select the output variable(s), so that the signature is

```
"list", "model.data.out", "character".
```

If several output variables are included in the “model.data.out” object, this allows the user to emulate a subset of them.

Methods are also invaluable when some functionality is to be added in such a way that objects of some new class cannot be treated in the same way as their analogies from the original framework. For example, suppose a principal component emulator were to be built, where the input variables to the emulator are the principal components of the inputs to the simulator. To achieve this, a new super-class of “model.data.out”, “model.data.pc” could be created. In most respects, this data will be used for emulation in almost exactly the same way as ordinary input and output data. However, whereas in the standard emulator the inputs of training data are often rescaled to be in a particular interval (in our case usually $[-1, 1]$), when the inputs are principal components it is common practice to standardise them to have mean zero and unit variance. Therefore, whereas the method of `rescale` used for a “model.data.out” object uses the elements of the ‘oldrange’ slot as minima and maxima, the method for “model.data.pc” should interpret them as a mean and standard deviation with which to standardise the data. Because of inheritance, functions that should treat both classes in the same way do not need to have a new method written.

Because the information is kept together with a tightly controlled structure, this framework is also beneficial when working over a long time frame, or when making changes after periods of inactivity. If an “em.multi” or “ep.multi” object is saved, all the information necessary to continue using the emulator is retained and kept together.

It is also possible that one might like to alter an emulation object by making slightly different choices, for example to try a different set of regression functions, or new correlation lengths.

A key function in the emulation framework is `change.obj`. This can be given an object of any class from Tables 7.1, 7.3 or 7.4, and a list named ‘changes’, whose elements must be named after arguments used somewhere in the emulation structure, and will make a new instance of the same class but with the changed arguments taking effect. For example, if some predictions had been made (to form an “ep.multi” object ‘pred.old’), and diagnostics suggested using smaller correlation lengths ‘new.corrlen’ the call

```
pred.new <- change.obj(pred.old, changes = list(corrlen=new.corrlen))
```

would be sufficient to create an “ep.multi” object ‘pred.new’ where the “corr.mats” and “em.multi” objects had been re-built with the new correlation lengths.

7.7 Summary

This chapter has given a brief introduction to object-oriented programming (OOP), and motivated the development of an object-oriented framework for emulation. In order to pursue this, we then introduced the S4 class structure in R (R Development Core Team, 2011).

A framework was built for standard emulation, as described in Chapter 3. The classes in this framework mostly fit the three key stages of data, emulation and prediction. We also explained how the object-oriented nature of the framework helps structure the more involved elements of the process, such as the regression function or the correlated error.

The standard emulation framework was then extended twice; once to incorporate hierarchical emulation (introduced in Chapter 5), and once for intermediate variable emulation (from Chapter 6). For both of these, a key part of the problem is the structuring of the training data. Once this has been achieved, the standard emulation classes can be used while the structure is maintained through specially created classes mirroring the emulation and prediction stages mentioned earlier.

Having established these frameworks, the benefits of OOP were investigated further, focussing particularly on the flexibility and adaptability that these frameworks allow.

Chapter 8

Conclusion

The initial goal of this thesis was to develop the technique of Bayesian emulation to be able to handle two simulators. This aim was motivated by the observation that while no simulator will fully or accurately capture all aspects of the system it models, different simulators have different strengths. Somehow being able to compare their behaviour and representation is therefore a useful skill.

To illustrate our methods two simulators of the ocean carbon cycle, HadOCC and OG99NPZD, were introduced. These both model the biological processes in the ocean responsible for the 'biological sink', the transportation of carbon to the deep ocean by organic matter. Their input spaces are not obviously linked in any way, and as we saw in the example in Section 3.6, this means that standard emulation methods are unable to help compare them. It was hoped that methods developed in this thesis might enable us to draw some conclusions about the differences and similarities in their representations of the ocean carbon cycle.

Part of the attraction of using emulation was that the simulators are treated as functions, and can therefore be compared across their input spaces. As we saw in Chapter 4, this is not the case with very many of the current methods involving multiple simulators. This led to a study of the possible ways in which two simulators of the same system could be different, and in particular how their input spaces might be related. These ranged from two versions of a simulator that could be made to be the same, to two simulators modelling the system in terms of very different processes. It seemed sensible to capitalise on any links that could be made between the two, and so the methods developed in this thesis applied to different situations from among those listed in Section 4.2.

Hierarchical emulation was introduced in Chapter 5, and focusses on the situation in which one simulator ($s_1(x, v, w)$) extends the other ($s_0(x)$), in such a way that

$$s_0(x) = s_1(x, v^*, w)$$

for all valid x and w. This situation arises often, when a new process can be added to a simulator, or existing components can be made more detailed. The ability in HadOCC to make the carbon:chlorophyll (C:Chl) ratio either constant or varying fits into this category.

Hierarchical emulation preserves the relationship between the two simulators and enables emulation of either, and of the difference between them, by writing the more complicated function as a sum of several terms, one of which is the simpler function. A large validation study showed this method to outperform standard emulators in the case of HadOCC, and to therefore be a useful tool for studying the relationship between the two versions. Hierarchical emulation proved to be particularly effective compared to standard methods when given only a small amount of data from s_1 compared to s_0. This makes it an especially attractive option when the extended simulator is much more costly to run than the simpler version.

The prior specification is an important aspect of hierarchical emulation. In our formulation each of the terms in the hierarchical emulator is independent of the others, for reasons given in Section 5.2.1. Investigation into other prior distributions that would maintain the properties we desire, but for which the terms of the emulator are not independent, may further improve results.

In the grand scheme of simulator difference, hierarchical emulation can be used on very few pairs of simulators. In particular, it could not enable us to compare OG99NPZD and HadOCC, whose input spaces are entirely different.

Intermediate variable emulation, introduced in Chapter 6, applies to a much broader class of pairs of simulators. The only requirement it makes is that both simulators should contain sub-processes that have the same meaning. In two simulators of the same system this should not be uncommon. Examples of how this can be achieved in HadOCC and OG99NPZD are given in Section 6.2.1. These intermediate variables enable the simulators to be compared in two stages.

Firstly, relationships between the input variables and the intermediate processes are studied using emulators, enabling links to be made between the input spaces of the two simulators. In the case of OG99NPZD and HadOCC, this showed that most of the input parameters that appear to be equivalent in the two simulators either behave very similarly or have little effect in either. The most active input variables in each simulator turned out to be some that had no equivalent (in terms of their descriptions in the code and documentation) in the other. However, studying the relevant emulators provided evidence that these inputs might be linked. Why these important quantities appear to have different meanings could perhaps be an area for furthering understanding of the system.

The second stage is to study the relationships between the intermediate and output variables in each simulator, to compare the ways the sub-processes are used to form the output. Because both simulators have the same intermediate variable space, the intermediate to output variable emulator built using data from one simulator can be used over data from the other. This enabled us to see directly how differently the two simulators behave at this stage. This showed that the output was affected most strongly by the same intermediate variable in both OG99NPZD and HadOCC. Using each emulator over data from the other simulator showed that the effects of this variable were very similar. Some of the intermediate variables appeared to have somewhat different effects on the output in each simulator. These findings could be used to further investigate how each simulator uses the sub-processes it models.

It appeared from our use of intermediate variable emulation that OG99NPZD and HadOCC are fairly similar. The intermediate variable data from each have very similar patterns, and one can make some links between the input spaces. There are differences in how each handles its intermediate processes, but the most influential variable was very similar in both. Possible avenues for further investigation of their differences are given in Chapter 6. Many of these would benefit enormously from the advice of an expert.

Many possible developments could be made to intermediate variable emulation, some of which are mentioned in Section 6.7. Perhaps one of the most compelling is the possibility of experimental design in intermediate variable space. For reasons that are explained in Chapter 6, this is not a simple matter, but would greatly facilitate comparison of the simulators, particularly where intermediate variables are usually very highly correlated. Developing the method to incorporate observed system data or judgements about which simulator is 'better' also has great potential to increase the method's usefulness.

Finally, an object-oriented framework was presented in Chapter 7. This covered each of the emulation methods studied in this thesis, and demonstrates some of the advantages of using object-oriented programming.

Bibliography

Alfons, A., M. Templ, and P. Filzmoser (2010). An object-oriented framework for statistical simulation: The r package simframe. *Journal of statistical software 37*, 1–35.

Anderson, T. R. (1993). A Spectrally averaged model of light penetration and photosynthesis. *Limnology and Oceanography 38*(7), 1403–1419.

Bastos, L. S. and A. O'Hagan (2009). Diagnostics for Gaussian Process Emulators. *Technometrics 54*(4), 425–438.

Bayarri, M., J. Berger, J. Cafeo, G. Garcia-Donato, F. Lui, J. Palomo, R. Parthasarathy, R. Paulo, J. Sacks, and D. Walsh (2007). Computer model validation with functional output. *The Annals of Statistics 35*(5), 1874–1906.

Bingham, D., R. R. Sitter, and B. Tang (2009). Orthogonal and nearly orthogonal designs for computer experiments. *Biometrika 96*(1), 51–65.

Buser, C., H. Knsch, D. Lthi, M. Wild, and C. Schr (2009). Bayesian multi-model projection of climate: bias assumptions and interannual variability. *Climate Dynamics 33*(6), 849–868.

C. Devon Lin, Derek Bingham, R. R. S. and B. Tang (2010). A new and flexible method for constructing designs for computer experiments. *The Annals of Statistics 38*(3), 1460–1477.

Chambers, J. (1998). *Programming with data: a guide to the S language.* Springer.

Conti, S., J. P. Gosling, J. Oakley, and A. O'Hagan (2009). Gaussian process emulation of dynamic computer codes. *Biometrika 96*(3), 663–676.

Conti, S. and A. OHagan (2010). Bayesian emulation of complex multi-output and dynamic computer models. *Journal of Statistical Planning and Inference 140*(3), 640 – 651.

Craig, P. S., M. Goldstein, J. C. Rougier, and A. H. Seheult (2001, June). Bayesian Forecasting for complex systems using computer simulators. *Journal of the American Statistical Association 96*(454), 717–729.

Craig, P. S., M. Goldstein, A. H. Seheult, and J. A. Smith (1996). Bayes linear strategies for matching hydrocarbon reservoir history. *Bayesian Statistics 5*, 69–95.

Craig, P. S., M. Goldstein, A. H. Seheult, and J. A. Smith (1997). Pressure matching for hydrocarbon reservoirs: a case study in the use of Bayes linear strategies for large computer experiments. *Case Studies in Bayesian Statistics 3*, 37–93.

Cumming, J. and D. Wooff (2007). Dimension reduction via principal variables. *Computational Statistics and Data Analysis 52*(1), 550 – 565.

Cumming, J. A. and M. Goldstein (2010). Bayes linear uncertainty analysis for oil reservoirs based on multiscale computer experiments. In A. O'Hagan and M. West (Eds.), *The Oxford handbook of applied Bayesian analysis*, pp. 241–270. Oxford University Press.

Fasham, M., H. Ducklow, and S. McKelvie (1990). A nitrogen-based model of plankton dynamics in the oceanic mixed layer. *Journal of Marine Research 48*(3), 591–639.

Goldstein, M. and J. Rougier (2009, March). Reified Bayesian Modelling and Inference for Physical Systems. *Journal of Statistical Planning and Inference 139*(3), 1221–1239.

Goldstein, M. and D. Wooff (2007). *Bayes Linear Statistics, Theory and Methods.* Wiley Series in Probability and Statistics. Wiley.

Grosso, A., A. Jamali, and M. Locatelli (2009). Finding maximin latin hypercube designs by iterated local search heuristics. *European Journal of Operational Research 197*, 541547.

Handcock, M. S. and J. R. Wallis (1994). An approach to statistical spatial-temporal modeling of meteorological fields. *Journal of the American Statistical Association 89*(426), 368–378.

Hemmings, J. (2000). Validation of oceanic ecosystem models using satellite ocean colour and data assimilation techniques: a preliminary experiment. Technical report 58, Southampton Oceanography Centre.

Hemmings, J. C. (2009). A marine model optimization test-bed for ecosystem model evaluation: MarMOT version 1.0 description and user guide. Technical report, National Oceanography Centre, Southampton.

Hemmings, J. C., R. M. Barciela, and M. J. Bell (2008). Ocean color data assimilation with material conservation for improving model estimates of air-sea CO_2 flux.

Hemmings, J. C. and P. G. Challenor (2011). Addressing the impact of environmental uncertainty in plankton model calibration with a dedicated software system: the marine model optimization testbed (marmot).

Henderson, D. A., R. J. Boys, K. J. Krishnan, C. Lawless, and D. J. Wilkinson (2009). Bayesian emulation and calibration of a stochastic computer model of mitochondrial dna deletions in substantia nigra neurons. *Journal of the American Statistical Association 104*(485), 76–87.

Hung, Y., V. R. Joseph, and S. N. Melkote (2009). Design and Analysis of Computer Experiments With Branching and Nested Factors. *Technometrics 51*(4), 354–365.

Iman, R. L. and W. Conover (1982). A distribution-free approach to inducing rank correlation among input variables. *Communications in Statistics - Simulation 11*(3), 311–334.

James, D. A. and S. DebRoy (2010). *RMySQL: R interface to the MySQL database.* R package version 0.8-0.

Jolliffe, T. (2002). *Principal Component Analysis.* Springer.

Joseph, V. R. and Y. Hung (2008). Orthogonal-maximin latin hypercube designs. *Statistica Sinica 18*, 171–186.

Kaufman, C., D. Bingham, S. Habib, K. Heitmann, and J. Frieman (2011). Efficient Emulators of Computer Experiments Using Compactly Supported Correlation Functions, With an Application to Cosmology. *ArXiv e-prints*.

Kaufman, C. and S. Sain (2010). Bayesian Functional ANOVA Modeling Using Gaussian Process Prior Distributions. *Bayesian Analysis 5*(1), 123–150.

Kennedy, M. C., C. W. Anderson, S. Conti, and A. O'Hagan (2006). Case Studies in Gaussian process modelling of computer codes. *Reliability Engineering and System Safety 91*, 1301–1309.

Kennedy, M. C. and A. O'Hagan (2001). Bayesian Calibration of computer models. *Journal Of The Royal Statistical Society Series B 63*(3), 425–464.

Kiers, H. A. and A. K. Smilde (2007). A comparison of various methods for multivariate regression with highly collinear variables. *Statistical Methods & Applications 16*(2), 193–228.

Leisch, F. (2004, May 20–22). S4 classes and methods. Keynote lecture at "useR! 2004", Vienna, Austria.

Leith, N. A. and R. E. Chandler (2010). A framework for interpreting climate model outputs. *Journal of the Royal Statistical Society 59*(2), 279296.

McKay, M., R. Beckman, and W. Conover (1979). A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics 21*(2), 239–245.

Morris, M. D. and T. J. Mitchell (1995). Explanatory designs for computational experiments. *Journal of Statistical Planning and Inference 43*(3), 381–402.

Narula, S. C. (1979). Orthogonal polynomial regression. *International Statistics Review 47*(1), 31–36.

Oakley, J. E. (2002). Eliciting gaussian process priors for complex computer codes. *The Statistician 51*(1), 81–97.

Oakley, J. E. and A. O'Hagan (2004). Probabilistic sensitivity analysis of complex models: A Bayesian approach. *Journal Of The Royal Statistical Society Series B 66*(3), 751–769.

O'Hagan, A. (1998). A markov Property for Covariance Structures. url=`http://www.tonyohagan.co.uk/academic/ps/kron.ps`.

O'Hagan, A. (2006). Bayesian analysis of computer code outputs: A tutorial. *Reliability Engineering and System Safety 91*(10-11), 1290–1300.

O'Hagan, A. and J. Forster (2004). *Bayesian Inference* (second ed.), Volume 2B of *Kendall's Advanced Theory of Statistics*. Arnold.

Oschlies, A. and V. Garçon (1999). An eddy-permitting coupled physical-biological model of the North Atlantic. 1. Sensitivity to advection numerics and mixed layer physics. *Global Biogeochemical cycles 13*.

Owen, A. B. (1994). Controlling correlations in latin hypercube samples. *Journal of the American Statistical Association 89*(428), 1517–1522.

Palmer, J. and I. Totterdell (2001). Production and export in a global ocean ecosystem model. *Deep-Sea Research 48*(5), 11691198.

Qian, P. (2012). Sliced latin hypercube designs. *to appear in Journal of the American Statistical Association*.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. ISBN 3-900051-07-0.

Raftery, A. E., T. Gneiting, F. Balabdaoui, and M. Polakowski (2005). Using Bayesian Model Averaging to Calibrate Forecast Ensembles. *Monthly Weather Review 133*, 1155–1174.

Ramsay, J. and C. Dalzell (1991). Some tools for functional data analysis. *Journal Of The Royal Statistical Society Series B 53*(3), 539–572.

Ramsay, J. and B. Silverman (1997). *Functional Data Analysis.* Springer.

Rougier, J. (2009). Expert Knowledge and Multivariate Emulation: The Thermosphere-Ionosphere Electrodynamics General Circulation Model (TIE-GCM). *Technometrics 51*(4), 414–424.

Sacks, J., W. J. Welch, T. J. Mitchell, and H. P. Wynn (1989). Design and analysis of computer experiments. *Statistical Science 4*(4), 409–423.

Saltelli, A., K. Chan, and E. Scott (2000). *Sensitivity Analysis.* Wiley.

Santner, T. J., B. J. Williams, and W. Notz (2003). *The design and analysis of computer experiments.* New York: Springer.

Silverman, B. (1985). Some aspects of the spline smoothing approach to non-parametric regression curve fitting. *Journal Of The Royal Statistical Society Series B 47*(1), 1–52.

Smith, R. L., C. Tebaldi, D. Nychka, and L. O. Mearns (2009). Bayesian Modeling of Uncertainty in Ensembles of Climate Models. *Journal of the American Statistical Association 104*(485), 97–116.

Stein, M. (1987). Large sample properties of simulations using latin hypercube sampling. *Technometrics 29*(2), 143–151.

Stein, M. (1999). *Interpolation of spatial data: some theory for kriging.* Springer series in statistics. Springer.

Steinberg, D. K., C. A. Carlson, N. R. Bates, R. J. Johnson, A. F. Michaels, and A. H. Knap (2001). Overview of the US JGOFS bermuda atlantic time-series study (BATS): a decade-scale look at ocean biology and biogeochemistry. *Deep Sea Research Part II: Topical Studies in Oceanography 48*(8-9), 1405 – 1447.

Strong, M., J. E. Oakley, and J. Chilcott (2012). Managing structural uncertainty in health economic decision models: a discrepancy approach. *Journal Of The Royal Statistical Society Series C 61*(1), 25–45.

Todorov, V. and P. Filzmoser (2009). An object-oriented framework for robust multivariate analysis. *Journal of Statistical Software 32*(3), 1–47.

Venables, W. and B. Ripley (2000). *S programming.* Statistics and computing. Springer.

Vernon, I., M. Goldstein, and R. G. Bower (2010, December). Galaxy formation : a bayesian uncertainty analysis. *Bayesian analysis. 05*(04), 619–670.

Wang, Y. (1998). Smoothing spline models with correlated random errors. *Journal of the American Statistical Society 93*(441), 341–348.

Wilks, D. S. (2006). Comparison of ensemble-MOS methods in the Lorenz '96 setting. *Meteorological Applications 13*(3), 243–256.

Appendix A

Notation

This table lists the more important notation used in this thesis, a brief description of the meaning, and the section in which it was first introduced. For simulator input notation, which is not listed here, see Tables 2.1, 2.2 and 2.3, and for intermediate variables for the example in Chapter 6 see Table 6.1.

Table A.1: *Notation used, with brief description and where first introduced.*

Notation	Description	Introduced
FDM90	The seminal compartmental PZN simulator, see Fasham et al. (1990)	Sec. 2.1, p. 6
P	Phytoplankton concentration in terms of nitrogen, in a compartmental model such as FDM90	Sec. 2.1, p. 6
Z	Zooplankton concentration in terms of nitrogen	Sec. 2.1, p. 6
D	Detritus concentration in terms of nitrogen	Sec. 2.1, p. 6
OG99NPZD	The compartmental ecosystem model from Oschlies and Garçon (1999)	Sec. 2.2, p.7
N	Nutrient concentration in terms of nitrogen	Sec. 2.2, p. 8
HadOCC	The Hadley Centre Ocean Carbon Cycle model (Palmer and Totterdell, 2001; Hemmings et al., 2008), another compartmental ocean ecosystem simulator	Sec. 2.3, p. 10
MarMOT	The Marine Model Optimization Testbed, the software through which we run the simulators	Sec. 2.4, p. 15
`iz.`	A prefix to an output variable to denote that the values have been depth-integrated. A sort of average over all depth levels.	Sec 2.4, p. 16

Notation	Description	Introduced
$\mathbf{x}$	A set of n input points to a simulator.	Sec 3, p. 20
$s\left(\cdot\right)$	A simulator	Chapter 3, p. 20
$f(\cdot)$	An emulator	Sec 3.1, p. 21
$\xi_{\mathrm{i}}\left(\cdot\right)$	Regression functions for the emulator, usually for $i = 1, \ldots, q$, where $\xi_1\left(\cdot\right) = 1$.	Sec 3.1, p. 21
$\boldsymbol{\beta}$	A vector of unknown coefficients for the regression terms of the emulator	Sec 3.1, p. 21
β_i	The coefficient of $\xi_{\mathrm{i}}\left(\mathbf{x}\right)$	Sec 3.1, p. 21
$\epsilon\left(\cdot\right)$	The correlated error function for the emulator	Sec 3.1, p. 21
$\mathbf{X}$	The $n \times q$ design matrix resulting from points $\mathbf{x}$ and functions $\xi_{\mathrm{i}}\left(\cdot\right)$	Sec 3.1, p. 21
$\Sigma\left(\cdot\right)$	The correlation matrix for the correlated error $\epsilon\left(\cdot\right)$	Sec 3.1, p. 22
σ_ϵ^2	The unknown variance of $\epsilon\left(\cdot\right)$, assumed to be the same for each input point	Sec 3.1, p. 22
$\rho\left(\cdot, \cdot\right)$	The correlation function used to build $\Sigma\left(\cdot\right)$	Sec 3.1, p. 22
$\tilde{\mathbf{x}}$	A set of m new input points, in contrast to $\mathbf{x}$, for which we usually know $s\left(\mathbf{x}\right)$	Sec 3.1, p. 23
LHD	A Latin hypercube design	Sec 3.3.1, p. 26
Θ	The matrix of correlation lengths used by $\rho\left(\cdot, \cdot\right)$	Sec 3.3.3, p. 31
θ_i	The correlation length associated with input x_i, when Θ is diagonal	Sec 3.3.3, p. 32
$\tilde{\mathrm{V}}$	Shorthand for $\mathrm{var}\left(\hat{s}\left(\tilde{\mathbf{x}}\right) \mid s\left(\mathbf{x}\right)\right)$	Sec 3.5, p. 36
$RMSE\left(\cdot\right)$	The root mean squared error between a vector of emulator predictions and the true simulator outputs	Sec 3.5, p. 36
$SPE\left(\cdot\right)$	The standardised prediction errors	Sec 3.5, p. 36
$MD\left(\cdot\right)$	The Mahalanobis distance	Sec 3.5, p. 37
`rcchlopt`	A switch variable in HadOCC, determining whether the C:Chl ratio is constant (`rcchlopt` $= 0$) or varying (`rcchlopt` $= 1$).	Section 4.2.1, p. 58
$s_1\left(x, x, w\right)$	The more complex of two simulators in a hierarchical setting	Chapter 5, p. 66
$s_0\left(x\right)$	The simpler of two simulators in a hierarchical setting	Chapter 5, p. 66
v	The hierarchical variables	Chapter 5, p. 66
w	The extra variables	Chapter 5, p. 66

Notation	Description	Introduced
v^*	The value of v such that $s_1(x, v^*, w) = s_0(x)$ for all valid x, w	Chapter 5, p. 66
$g_{[i]}(\cdot)$	The transformation function for hierarchical variable v_i	Section 5.1, p. 66
$\psi_{[i]}(\cdot)$	One of the functions making up the hierarchical emulator	Section 5.1, p. 67
$h_{[i]}\left(x, v_{[i]}, w\right)$	An emulator of $\psi_{[i]}(\cdot)$	Section 5.2, p. 69
$H_{[i]}$	The design matrix from $h_{[i]}\left(x, v_{[i]}, w\right)$	Section 5.2, p. 69
R	The hierarchical variable in the `rcchlopt` example in the hierarchical emulation chapter.	Section 5.5.1, p. 83
m_2	The hierarchical variable in the `rcchlopt` example in the hierarchical emulation chapter.	Section 5.5.1, p. 83
‘lhd1’	The 2,000 point training data in the `rcchlopt` example in the hierarchical emulation chapter.	Section 5.5.2, p. 85
‘lhd1_0’	The portion of lhd1 in which $R = 0$.	Section 5.5.2, p. 85
‘lhd1_1’	The portion of lhd1 in which $R \neq 0$.	Section 5.5.2, p. 85
OG100, OG1000, HAD100, HAD1000	Initial designs for intermediate variable emulation example	Section 6.2.1, p. 112
$\texttt{iz.tran}_{\text{time}}^{\text{simulator}}$	Intermediate variable `iz.tran` from simulator `sim` at time t	Section 6.2.1, p. 113
$k_{\texttt{int}}$	The number of principal variables used to represent intermediate variable `int`	Section 6.3, p. 115
$P_{\texttt{int}}^{\texttt{sim}}$	The $n \times k_{\texttt{int}}$ matrix of principal variables from full $n \times t$ datasets of `int`	Section 6.3, p. 115
$T_{\texttt{int}}^{\texttt{sim}}$	The $(k_{\texttt{int}} \times t)$ transform matrix such that $P_{\texttt{int}}^{\texttt{sim}} T_{\texttt{int}}^{\texttt{sim}}$ approximately reconstructs the full data	Section 6.3, p. 116
$\mathbf{X}_{\texttt{sim}}$	A matrix of intermediate variable data from simulator `sim`	Section 6.4, p. 130
$\mathbf{\Sigma}_{\texttt{sim}}$	The variance matrix of $\mathbf{X}_{\texttt{sim}}$	Section 6.4, p. 130
OGPV99, HADPV99	The datasets formed using the principal variables of OG1000 and HAD1000	Section 6.4.1, p. 131

Notation	Description	Introduced
$(U_{\texttt{int}}, L_{\texttt{int}})$	The upper and lower bounds within which intermediate variable `int` is considered acceptable	Section 6.5, p. 138
$s_{\texttt{int}}(x)$	The simulator's value of `int` at input points x	Section 6.5, p. 138
$f_{\texttt{int}}(x)$	The emulator's prediction of $s_{\texttt{int}}(x)$	Section 6.5, p. 138
OG948, OG947, OG98, HAD1007, HAD1005, HAD119	Datasets for OG99NPZD and HadOCC, refined to avoid zooplankton extinction	Section 6.5.1, p. 148
OOP	Object-oriented programming	Section 7.1, p. 183

Appendix B

Input to intermediate relative coefficients

The following tables (Tables B.2 to B.11) show the standardised generalised least squares estimates for the coefficients for emulators of each intermediate variable at each of its time points, from emulators built with second order regression surfaces found using R's stepwise model selection. The emulators had OG948 or HAD1005 as training data. To check their performance, the root mean squared error (RMSE) is given from using each emulator to predict the output for OG947 or HAD1005. The mean output is also given for comparison. The values have been standardised by dividing each column by the largest coefficient (in magnitude) for that output, so that each column's values can range between plus and minus one. Table B.1 shows the two emulators for `log.iz.pon`, the overall output. Any coefficient whose absolute value is below 0.15 has been replaced by a dot, and any row with no relative coefficient larger than ±0.15 has been omitted.

Time	1	13	105	365
a	0.38	·	·	·
c	0.27	·	·	·
K_1	-0.33	·	·	·
μ_P	-1	-1	-1	-1
μ_P^2	·	·	0.55	0.38
a^2	-0.21	·	-0.22	-0.24
γ_2^2	·	·	0.2	0.24
$a \times \mu_P$	·	·	0.19	0.2
$a \times c$	·	·	·	-0.16
Mean	3.1	2.9	2.4	2.2
RMSE	0.0010	0.012	0.086	0.11

Time	1	13	105	365
`photmax`	0.15	·	0.25	0.29
`alphachl`	·	·	0.21	0.22
`kdin`	-0.31	-0.22	-0.33	-0.4
`presp`	-1	-1	-1	-1
`presp`^{2}	·	·	0.31	·
`photmax`^{2}	·	·	-0.16	-0.19
`kdin`^{2}	0.23	·	·	·
`kdin`×`presp`	·	·	-0.38	-0.45
`photmax`×`presp`	·	·	0.29	0.35
`photmax`×`kdin`	·	·	0.23	0.25
`rcchl`×`presp`	·	·	-0.16	-0.23
Mean	3.1	2.8	2.4	2.1
RMSE	0.0010	0.011	0.088	0.14

Table B.1: *Relative coefficients and emulator summaries for* `log.iz.pon` *for OG99NPZD (left) and HadOCC (right)*

Time	1	3	12	146	277	365
a	1	0.33	·	0.15	·	0.15
c	0.71	0.16	·	·	·	·
K_1	-0.91	-0.2	·	·	·	·
μ_P	·	1	1	1	1	1
μ_D	·	·	·	·	·	0.15
PAR	·	0.19	·	·	·	·
C_{PP}	0.17	0.35	·	·	·	·
γ_2^2	·	·	·	-0.33	-0.37	-0.46
μ_P^2	·	·	-0.3	·	·	-0.27
a^2	-0.58	-0.53	-0.15	-0.17	·	-0.17
K_1^2	0.27	-0.17	·	·	·	·
c^2	·	-0.17	·	·	·	·
$\mu_P \times \gamma_2$	·	·	·	0.2	0.18	0.19
$a \times K_1$	0.19	0.58	·	·	·	0.15
$a \times \mu_P$	·	0.4	·	0.16	·	·
$\epsilon \times \gamma_2$	·	·	·	0.17	0.21	0.26
$\gamma_1 \times \gamma_2$	·	·	·	·	·	0.17
$a \times c$	·	-0.51	·	·	·	·
$c \times \mu_P$	·	0.27	·	·	·	·
$K_1 \times \mu_P$	·	-0.36	·	·	·	·
$c \times K_1$	0.2	0.29	·	·	·	·
$\gamma_2 \times \mu_{ZZ}$	·	·	·	·	·	-0.15
Mean	1.0	0.59	0.42	0.48	0.47	0.362
RMSE	0.042	0.037	0.021	0.031	0.031	0.026

Table B.2: *OG99NPZD emulator of* `iz.np`.

Time	1	3	12	146	277	365
rcchl	-0.26	-0.25	-0.28	·	·	-0.34
rcnphy	·	·	-0.17	-0.17	-0.2	-0.18
photmax	0.49	0.5	0.53	0.35	0.44	0.58
alphachl	0.37	0.42	0.55	0.49	0.52	0.6
kdin	-1	-1	-0.92	-0.57	-0.76	-1
presp	·	0.17	0.82	1	1	0.68
pmortdd	·	·	·	0.2	0.18	·
presp^2	·	·	-0.65	-0.53	-0.64	-0.94
rcchl^2	·	·	-0.24	-0.25	-0.26	-0.19
photmax^2	·	·	-0.23	-0.18	-0.2	-0.25
kdin^2	0.73	0.45	·	·	·	0.18
zmort^2	·	·	·	-0.16	-0.35	-0.43
kdin×presp	·	-0.2	-1	-0.65	-0.8	-1
photmax×presp	·	·	0.53	0.42	0.49	0.61
rcchl×presp	·	·	-0.43	-0.33	-0.35	-0.58
alphachl×presp	·	·	0.31	0.21	0.24	0.35
photmax×kdin	-0.2	·	0.29	0.29	0.27	0.3
presp×zmort	·	·	·	0.38	0.38	0.4
presp×pmortdd	·	·	·	-0.18	-0.19	-0.26
presp×epsfood	·	·	·	-0.19	-0.18	-0.18
rcchl×alphachl	·	·	0.17	·	0.16	0.2
rcchl×kdin	0.36	0.15	·	·	·	·
rchlpig×alphachl	·	·	·	·	·	·
alphachl×kdin	-0.35	-0.23	·	·	·	-0.18
fingest×zmort	·	·	·	·	0.16	0.19
zmort×zmortdd	·	·	·	·	·	-0.19
epsfood×zmort	·	·	·	·	0.15	0.17
betap×zmort	·	·	·	·	·	0.16
photmax×alphachl	0.17	·	·	·	·	·
rcchl×photmax	-0.16	·	·	·	·	·
Mean	0.51	0.47	0.39	0.49	0.44	0.32
RMSE	0.045	0.027	0.024	0.036	0.031	0.029

Table B.3: *HadOCC emulator of* iz.np.

Time	1	30	282
μ_P	1	1	1
μ_P^2	·	-0.67	-0.19
γ_2^2	·	·	-0.22
$\mu_P \times \gamma_2$	·	·	0.15
$\epsilon \times \gamma_2$	·	·	0.15
Mean	0.92	0.38	0.30
RMSE	0.0016	0.024	0.026

Time	1	30	282
`rcnphy`	·	·	-0.19
`photmax`	·	0.21	0.31
`alphachl`	·	0.21	0.39
`kdin`	·	-0.36	-0.54
`presp`	1	0.73	1
`presp`^{2}	·	-1	-0.75
`rcchl`^{2}	·	·	-0.18
`photmax`^{2}	·	·	-0.15
`zmort`^{2}	·	·	-0.33
`photmax`×`presp`	·	0.23	0.38
`alphachl`×`presp`	·	0.22	0.3
`rcchl`×`presp`	·	-0.18	-0.23
`photmax`×`kdin`	·	·	0.21
`presp`×`zmort`	·	·	0.19
`epsfood`×`zmort`	·	·	0.25
`fingest`×`zmort`	·	·	0.16
Mean	0.91	0.43	0.39
RMSE	0.0024	0.030	0.034

Table B.4: *Relative coefficients and emulator summaries for* `iz.pn` *for OG99NPZD (left) and HadOCC (right).*

Time	3	31	133	359
a	·	0.18	·	·
γ_1	0.58	0.62	0.41	0.33
ϵ	1	1	0.64	0.58
μ_P	-0.16	-0.27	·	·
γ_2	·	-0.84	-1	-1
μ_{PP}	·	-0.16	·	·
ϵ^2	-0.48	-0.39	-0.2	·
γ_2^2	·	·	-0.34	-0.36
a^2	·	-0.17	·	·
$\epsilon \times \gamma_2$	·	0.2	0.41	0.41
$\gamma_1 \times \gamma_2$	·	·	0.27	0.24
$\mu_D \times \gamma_2$	·	·	·	0.17
Mean	-4.1	-4.8	-5.8	-7.3
RMSE	0.060	0.15	0.39	0.93

Time	3	31	133	359
`rcnphy`	0.22	0.19	·	·
`photmax`	·	0.38	·	·
`kdin`	-0.21	-0.62	-0.23	-0.19
`presp`	-0.23	-0.8	-0.26	-0.28
`epsfood`	1	0.89	0.45	0.42
`fmingraz`	-0.23	-0.19	·	·
`fingest`	0.35	0.33	0.19	0.17
`betap`	0.59	0.56	0.3	0.27
`zmort`	·	-1	-1	-1
`epsfood`^{2}	-0.47	-0.34	·	·
`photmax`^{2}	·	-0.2	·	·
`zmort`^{2}	·	·	-0.26	-0.25
`kdin`×`presp`	·	-0.56	-0.17	·
`betap`×`zmort`	·	·	0.22	0.23
`photmax`×`presp`	·	0.33	·	·
`epsfood`×`zmort`	·	·	0.28	0.33
`photmax`×`kdin`	·	0.25	·	·
Mean	-4.9	-5.3	-5.9	-7.3
RMSE	0.061	0.14	0.39	1.1

Table B.5: *Relative coefficients and emulator summaries for* `log.iz.pz` *for OG99NPZD (left) and HadOCC (right).*

Time	1	15	109	355
a	·	0.2	0.2	·
γ_1	-0.16	·	·	0.16
c	·	·	0.17	·
ϵ	·	0.18	0.36	0.36
K_1	·	-0.17	-0.2	-0.16
μ_P	·	-0.86	-1	-1
μ_D	·	·	0.3	0.28
C_{PP}	·	·	0.23	0.23
γ_2	·	·	-0.52	-0.59
μ_{PP}	1	1	0.95	0.73
μ_P^2	·	0.43	0.83	0.58
a^2	·	-0.41	-0.53	-0.37
μ_D^2	·	·	-0.21	-0.17
γ_2^2	·	·	-0.73	-1
$\mu_D \times \mu_{PP}$	·	·	0.15	·
$\mu_P \times \mu_{PP}$	·	-0.21	·	·
$a \times K_1$	·	0.35	0.43	0.29
$a \times \mu_P$	·	0.17	0.29	0.18
$\gamma_1 \times \gamma_2$	·	·	0.7	0.79
$\gamma_1 \times \epsilon$	·	-0.17	-0.3	-0.31
$\mu_P \times C_{PP}$	·	·	-0.2	·
$a \times c$	·	-0.22	-0.28	-0.17
$c \times K_1$	·	0.2	0.28	0.21
$K_1 \times \mu_P$	·	·	-0.25	·
$\epsilon \times \gamma_2$	·	·	0.45	0.57
$c \times \mu_P$	·	·	0.2	·
$\mu_P \times \mu_D$	·	·	-0.17	-0.18
$a \times \mu_D$	·	·	0.16	·
$K_1 \times \mu_D$	·	·	-0.18	·
$\gamma_2 \times \mu_{ZZ}$	·	·	-0.19	-0.41
$C_{PP} \times \gamma_2$	·	·	0.15	0.21
$\mu_P \times \gamma_2$	·	·	·	-0.28
$\mu_D \times \gamma_2$	·	·	0.24	0.27
Mean	0.28	0.18	0.15	0.13
RMSE	0.0023	0.0072	0.012	0.013

Time	1	15	109	355
`rcndet`	-0.17	-0.18	-0.16	·
`photmax`	·	0.18	0.15	·
`alphachl`	·	·	0.2	0.15
`kdin`	·	-0.38	-0.31	-0.33
`presp`	·	-0.76	-0.86	-1
`pmortdd`	1	1	1	0.82
`fpmortdin`	-0.18	-0.18	-0.17	·
`presp`^{2}	·	0.25	0.47	0.47
`pmortdd`^{2}	·	·	·	-0.15
`zmort`^{2}	·	·	-0.2	-0.3
`presp`×`pmortdd`	·	-0.69	-0.65	-0.6
`alphachl`×`pmortdd`	·	0.16	0.22	0.18
`kdin`×`pmortdd`	·	-0.36	-0.26	-0.23
`alphachl`×`presp`	·	·	-0.19	·
`kdin`×`presp`	·	·	-0.18	·
`pmortdd`×`zmort`	·	·	0.3	0.37
`rcchl`×`presp`	·	·	-0.16	-0.16
`pmortdd`×`fpmortdin`	-0.18	-0.16	·	·
`photmax`×`pmortdd`	·	0.18	·	·
`rcndet`×`pmortdd`	-0.17	-0.15	·	·
`presp`×`zmort`	·	·	-0.24	-0.51
`zmort`×`zmortdd`	·	·	·	-0.15
Mean	0.035	0.034	0.045	0.034
RMSE	0.0012	0.0031	0.0071	0.0078

Table B.6: *Relative coefficients and emulator summaries for* `iz.pd` *for OG99NPZD (left) and HadOCC (right).*

Time	1	28	172	343
γ_1	·	·	0.34	0.35
ϵ	·	0.15	0.53	0.54
γ_2	0.54	·	-1	-1
γ_2^2	-1	-1	-0.67	-0.58
$\gamma_1 \times \gamma_2$	·	·	0.32	0.35
$\epsilon \times \gamma_2$	·	·	0.41	0.5
Mean	-3.3	-3.8	-5.8	-7.5
RMSE	0.41	0.41	0.65	1.16

Time	1	28	172	343
`kdin`	·	·	·	-0.16
`presp`	·	·	-0.18	-0.24
`epsfood`	·	·	0.33	0.37
`fingest`	·	·	0.15	0.16
`betap`	·	·	0.24	0.26
`zmort`	0.34	-0.29	-1	-1
`fzmortdin`	0.27	0.31	·	·
`zmort`^{2}	-1	-1	-0.34	-0.31
`betap`×`zmort`	·	·	0.22	0.28
`epsfood`×`zmort`	·	·	0.25	0.35
`fingest`×`zmort`	·	·	·	0.16
`presp`×`zmort`	·	·	·	-0.21
Mean	-3.6	-4.2	-5.9	-7.4
RMSE	0.23	0.22	0.46	1.08

Table B.7: *Relative coefficients and emulator summaries for* `log.iz.zn` *for OG99NPZD (left) and HadOCC (right)*

Time	1	141	365
γ_1	·	0.32	0.31
ϵ	·	0.46	0.5
γ_2	·	-1	-1
μ_{ZZ}	1	·	·
μ_{ZZ}^2	-0.83	·	·
γ_2^2	·	-0.25	-0.4
$\epsilon \times \gamma_2$	·	0.35	0.48
$\gamma_1 \times \gamma_2$	·	0.24	0.32
Mean	-6.2	-9.0	-13.1
RMSE	0.38	0.81	2.19

Time	1	141	365
`kdin`	·	-0.15	-0.17
`presp`	·	-0.17	-0.24
`epsfood`	·	0.32	0.37
`fingest`	·	0.17	0.17
`betap`	·	0.24	0.26
`zmort`	0.33	-1	-1
`fzmortdin`	-0.75	-0.25	·
`fzmortdin`^{2}	-0.61	-0.21	·
`zmort`^{2}	-1	-0.35	-0.31
`betap`×`zmort`	·	0.21	0.28
`fingest`×`zmort`	·	·	0.17
`epsfood`×`zmort`	·	0.21	0.35
`presp`×`zmort`	·	·	-0.2
Mean	-4.6	-6.6	-8.5
RMSE	0.39	0.49	1.20

Table B.8: *Relative coefficients and emulator summaries for* `log.iz.zd` *for OG99NPZD (left) and HadOCC (right)*

Time	1	7	115	253
a	·	·	0.27	0.18
c	·	·	0.24	0.25
ϵ	·	0.15	0.33	0.38
K_1	·	·	-0.21	·
μ_P	·	-0.29	-1	-1
μ_D	1	1	0.32	0.27
C_{PP}	·	·	0.22	0.26
γ_2	·	·	-0.45	-0.52
μ_{PP}	·	0.95	0.96	0.9
μ_P^2	·	·	0.84	0.75
a^2	·	-0.34	-0.54	-0.39
μ_D^2	·	-0.24	-0.21	-0.2
$\mu_D \times \mu_{PP}$	·	0.33	0.15	·
$\gamma_2 \times \mu_{PP}$	·	·	0.19	0.21
$a \times K_1$	·	0.27	0.46	0.34
$a \times \mu_P$	·	·	0.3	0.25
$a \times c$	·	-0.17	-0.29	-0.26
$c \times K_1$	·	0.16	0.27	0.27
$K_1 \times \mu_P$	·	·	-0.23	-0.2
$\gamma_1 \times \gamma_2$	·	·	0.32	0.36
$\mu_P \times \mu_D$	·	-0.4	-0.23	·
$a \times \mu_D$	·	0.17	0.2	0.16
$\mu_P \times C_{PP}$	·	·	-0.18	-0.2
$c \times \mu_P$	·	·	0.21	0.22
$K_1 \times \mu_D$	·	-0.17	-0.21	-0.18
$\gamma_1 \times \epsilon$	·	·	-0.17	-0.17
$\gamma_2 \times \mu_{ZZ}$	·	·	-0.34	-0.47
$\epsilon \times \mu_{ZZ}$	·	·	·	0.19
Mean	0.031	0.093	0.15	0.18
RMSE	0.0001	0.0019	0.011	0.015

Time	1	7	115	253
rcnphy	0.23	0.21	·	·
rcndet	-0.31	-0.31	-0.17	-0.16
photmax	·	·	0.21	0.18
alphachl	·	·	0.2	0.26
kdin	·	-0.23	-0.33	-0.27
presp	·	-0.34	-0.85	-0.93
pmortdd	1	1	1	1
fpmortdin	-0.18	-0.18	-0.15	·
zmort	0.51	0.39	-0.21	-0.26
fzmortdin	-0.55	-0.5	·	·
presp^2	·	·	0.6	0.72
rcchl^2	·	·	·	-0.18
kdin^2	·	0.17	·	·
presp×pmortdd	·	-0.27	-0.62	-0.49
alphachl×presp	·	·	-0.24	-0.3
alphachl×pmortdd	·	·	0.2	0.17
kdin×pmortdd	·	-0.19	-0.24	-0.19
rcchl×presp	·	·	-0.17	-0.27
pmortdd×zmort	·	·	0.36	0.48
kdin×presp	·	·	-0.16	·
betap×zmort	·	·	0.22	0.3
kdin×epsfood	·	·	-0.16	-0.16
pmortdd×epsfood	·	·	-0.15	-0.22
presp×epsfood	·	·	-0.2	-0.18
epsfood×zmort	·	·	-0.26	-0.21
pmortdd×fpmortdin	-0.17	-0.18	·	·
rcndet×pmortdd	-0.16	-0.16	·	·
rchlpig×presp	·	·	·	-0.16
kdin×zmort	·	·	0.15	0.21
fmessyd×zmort	·	·	0.17	0.19
zmort×fzmortdin	-0.52	-0.43	·	0.18
Mean	0.0073	0.028	0.056	0.067
RMSE	0.0001	0.0012	0.0093	0.014

Table B.9: *Relative coefficients and emulator summaries for* iz.dn *for OG99NPZD (left) and HadOCC (right).*

Time	1	6	127	319
c	·	·	0.16	0.17
w_s	1	1	1	1
μ_P	·	·	-0.5	-0.53
μ_D	·	·	-0.43	-0.48
C_{PP}	·	·	0.16	0.17
μ_{PP}	·	0.39	0.43	0.4
μ_P^2	·	·	0.43	0.38
μ_D^2	·	·	0.25	0.29
a^2	·	·	-0.2	·
$w_s \times \mu_{PP}$	·	0.29	0.2	0.17
$w_s \times \mu_P$	·	-0.3	-0.66	-0.6
$w_s \times \mu_D$	·	-0.19	-0.55	-0.58
$\mu_P \times \mu_D$	·	·	0.29	0.29
$a \times w_s$	·	·	0.16	·
$a \times K_1$	·	·	0.21	0.16
$w_s \times K_1$	·	·	-0.17	·
$\gamma_2 \times \mu_{ZZ}$	·	·	-0.2	-0.29
Mean	9.8	27	52	53
RMSE	0.023	0.67	5.4	6.2

Time	1	6	127	319
`rcndet`	·	·	·	-0.16
`photmax`	·	·	·	0.17
`alphachl`	·	·	0.16	0.2
`kdin`	·	·	-0.19	-0.24
`presp`	·	·	-0.54	-0.81
`pmortdd`	·	0.22	0.6	0.74
`zmort`	·	0.17	-0.23	-0.22
`fzmortdin`	·	-0.2	-0.2	-0.15
`dsink`	1	1	1	1
`presp`^{2}	·	·	0.36	0.58
`zmort`^{2}	·	·	-0.2	·
`presp`×`dsink`	·	·	-0.48	-0.73
`pmortdd`×`dsink`	·	0.2	0.46	0.52
`presp`×`pmortdd`	·	·	-0.36	-0.41
`fzmortdin`×`dsink`	·	·	-0.18	-0.21
`alphachl`×`dsink`	·	·	·	0.15
`kdin`×`dsink`	·	·	-0.18	-0.25
`rcchl`×`presp`	·	·	·	-0.23
`alphachl`×`presp`	·	·	·	-0.2
`pmortdd`×`zmort`	·	·	0.22	0.38
`rcndet`×`dsink`	·	·	·	-0.18
`pmortdd`×`epsfood`	·	·	·	-0.2
`betap`×`zmort`	·	·	0.16	0.21
`kdin`×`epsfood`	·	·	·	-0.15
`fmessyd`×`zmort`	·	·	·	0.15
`zmort`×`fzmortdin`	·	-0.18	·	·
`zmort`×`dsink`	·	0.16	·	·
`kdin`×`zmort`	·	·	·	0.16
Mean	8.0	11.2	15	12
RMSE	0.041	0.39	2.4	2.6

Table B.10: *Relative coefficients and emulator summaries for* `iz.ds` *for OG99NPZD (left) and HadOCC (right)*

Time	3	33	137	314
`rcnphy`	0.34	0.27	·	·
`rcnzoo`	0.19	0.17	·	·
`rcndet`	-0.16	·	·	·
`photmax`	·	0.35	·	·
`kdin`	-0.2	-0.62	-0.21	-0.19
`presp`	-0.18	-0.75	-0.25	-0.29
`pmortdd`	0.58	0.46	·	·
`epsfood`	1	0.84	0.43	0.4
`fmingraz`	-0.22	-0.17	·	·
`fingest`	0.33	0.33	0.19	0.18
`betap`	·	0.17	0.25	0.27
`betad`	0.51	0.33	·	·
`zmort`	·	-1	-1	-1
`fzmortdin`	-0.24	·	·	·
`presp`^{2}	·	·	·	0.16
`epsfood`^{2}	-0.35	-0.26	·	·
`zmort`^{2}	·	·	-0.18	-0.16
`photmax`^{2}	·	-0.18	·	·
`pmortdd`^{2}	-0.22	-0.25	·	·
`kdin`×`presp`	·	-0.5	-0.17	·
`betap`×`zmort`	·	·	0.24	0.27
`presp`×`epsfood`	·	-0.2	·	·
`photmax`×`presp`	·	0.25	·	·
`kdin`×`epsfood`	·	-0.18	·	·
`photmax`×`kdin`	·	0.2	·	·
`epsfood`×`zmort`	·	·	0.16	0.19
`kdin`^{2}	0.15	·	·	·
Mean	-9.2	-8.3	-8.6	-9.0
RMSE	0.073	0.18	0.48	0.88

Table B.11: *HadOCC emulator of* `log.iz.dz`.

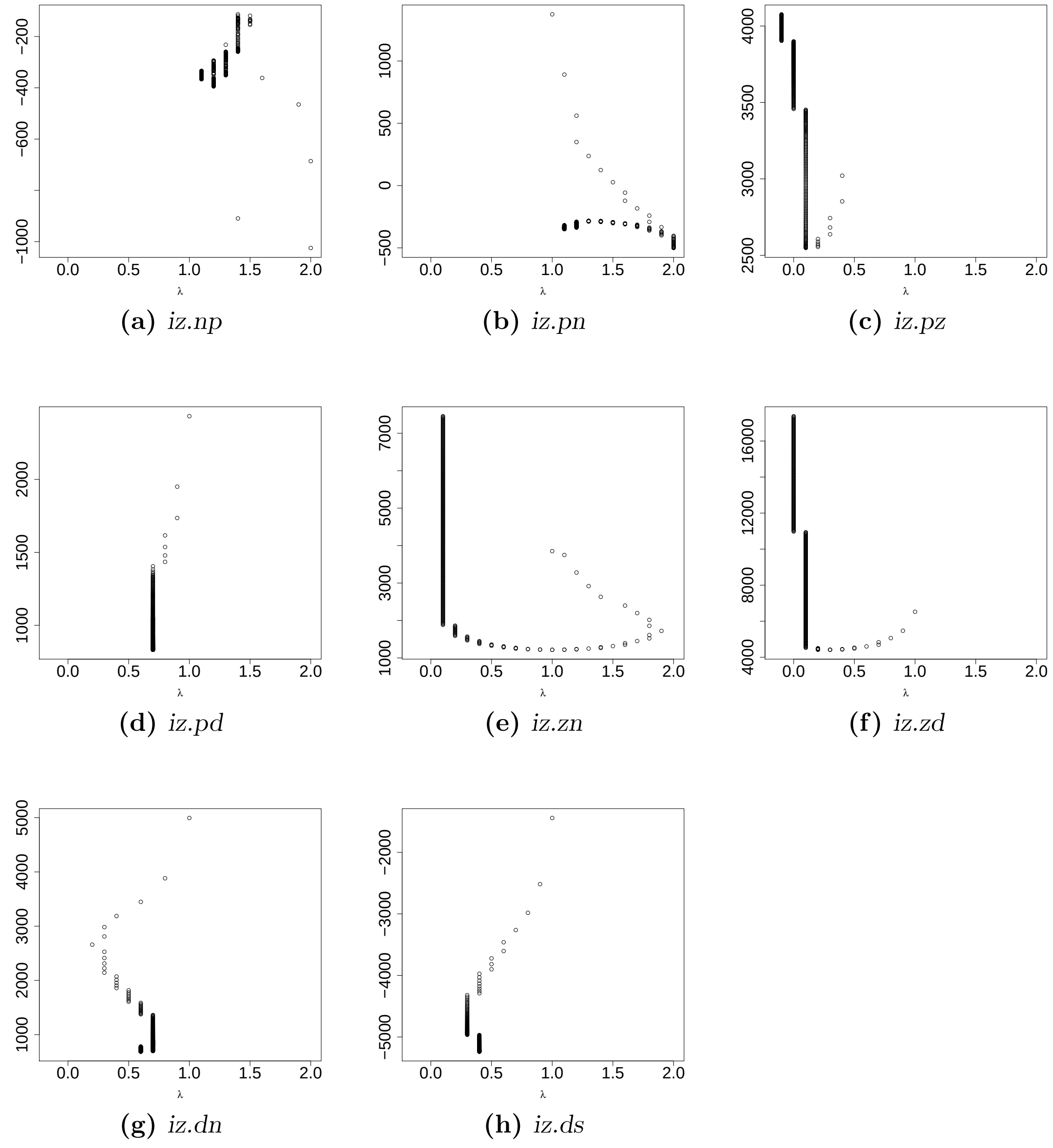
Appendix C

Further plots for intermediate variable emulators

C.1 Box-Cox for intermediate variables

Applying the Box-Cox model selection procedure to the intermediate variables supported using the logarithm of all zooplankton related variables. When the Box-Cox procedure was applied, each intermediate variable had 365 time points for each input point, in the datasets OG1000 and Had1000. A linear model was constructed (with a first order surface including all input variables), and the optimal transformation of the intermediate variable found, for each time point of each intermediate variable and for each simulator.

The plots in Figures C.1 and C.2 each contain 365 points, and show the optimal value of λ and the associated log-likelihood at each time point. It is clear from the relatively high likelihood values associated with values of λ around zero for the zooplankton related intermediates that the logarithm is a suitable choice of transformation. These plots could also be used to support a transformation of `iz.pd, iz.dn` and `iz.ds`, but this was not pursued because of the relatively low likelihood values, and to avoid complicating the procedure. Diagnostics later in the intermediate variable emulation process show that these quantities can be emulated well with a second order surface.



(a) *iz.np*

(b) *iz.pn*

(c) *iz.pz*

(d) *iz.pd*

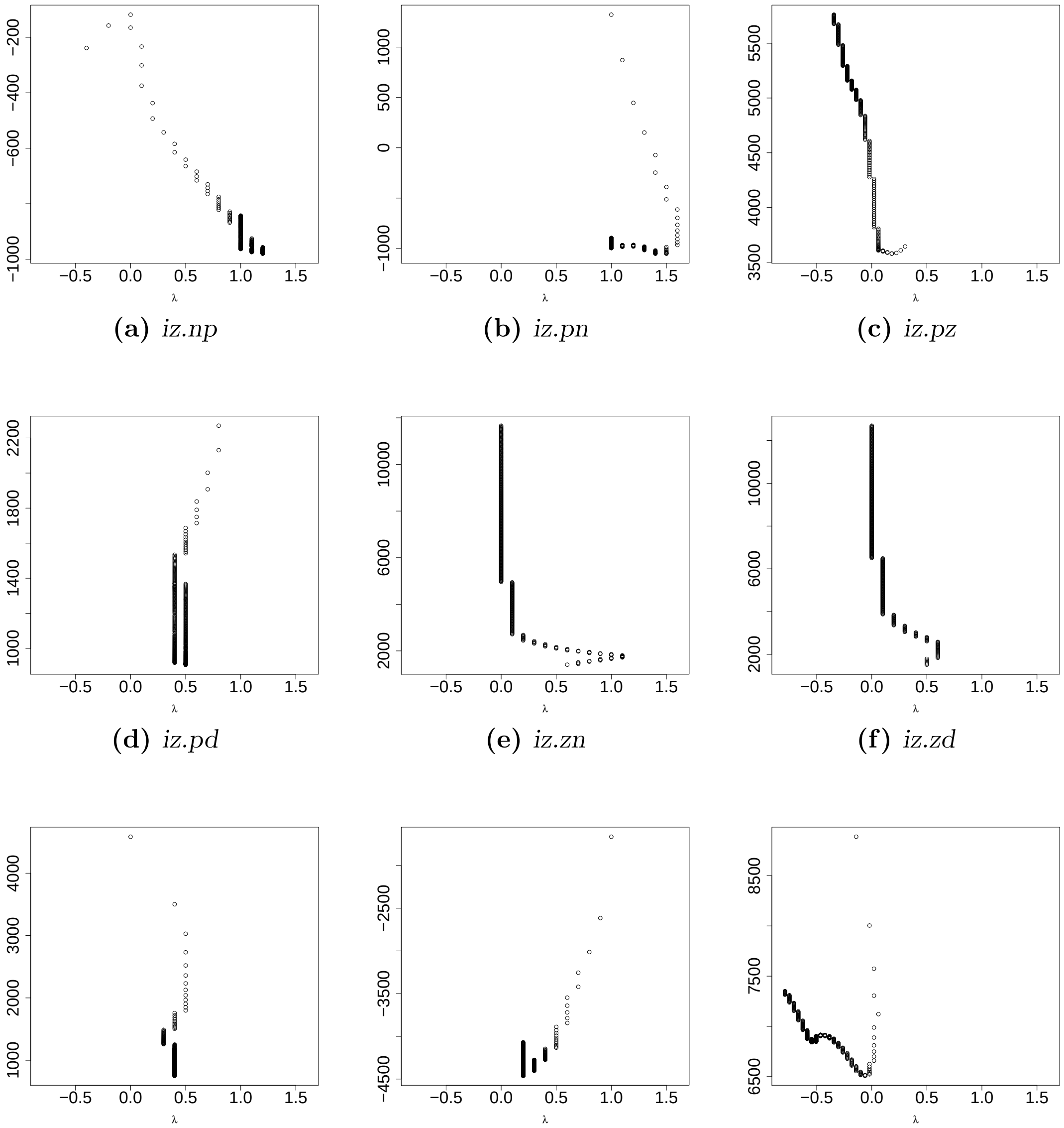
(e) *iz.zn*

(f) *iz.zd*

(g) *iz.dn*

(h) *iz.ds*

Figure C.1: *Log-likelihood values for the optimal λ given by the Box-Cox transformation procedure at each time point, for each intermediate variable in OG1000.*



(a) *iz.np*

(b) *iz.pn*

(c) *iz.pz*

(d) *iz.pd*

(e) *iz.zn*

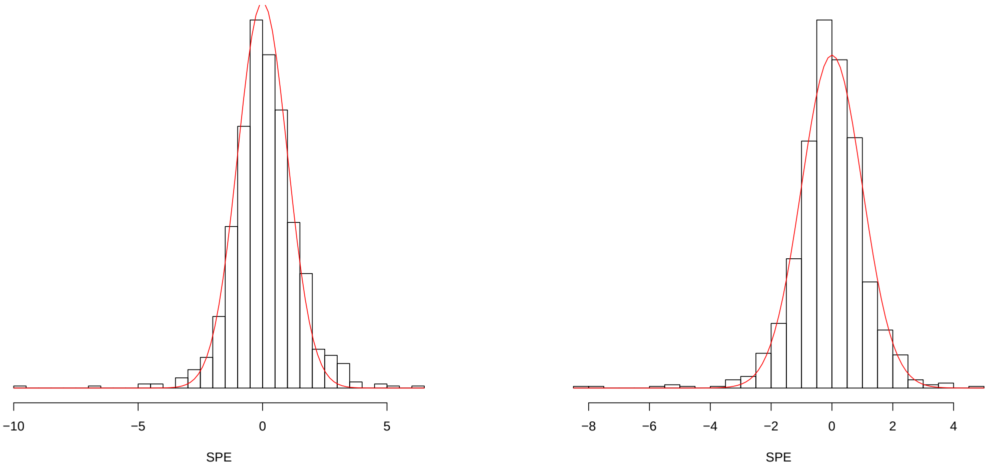
(f) *iz.zd*

(g) *iz.dn*

(h) *iz.ds*

(i) *iz.dz*

Figure C.2: *Log-likelihood values for the optimal λ given by the Box-Cox transformation procedure at each time point, for each intermediate variable in HAD1000.*



(a) *SPE for OG947 predicted by the OG99NPZD emulator.*
Mean = 0.096, SD = 1.30

(b) *SPE for HAD1007 predicted by the HadOCC emulator.*
Mean = -0.069, SD = 1.13

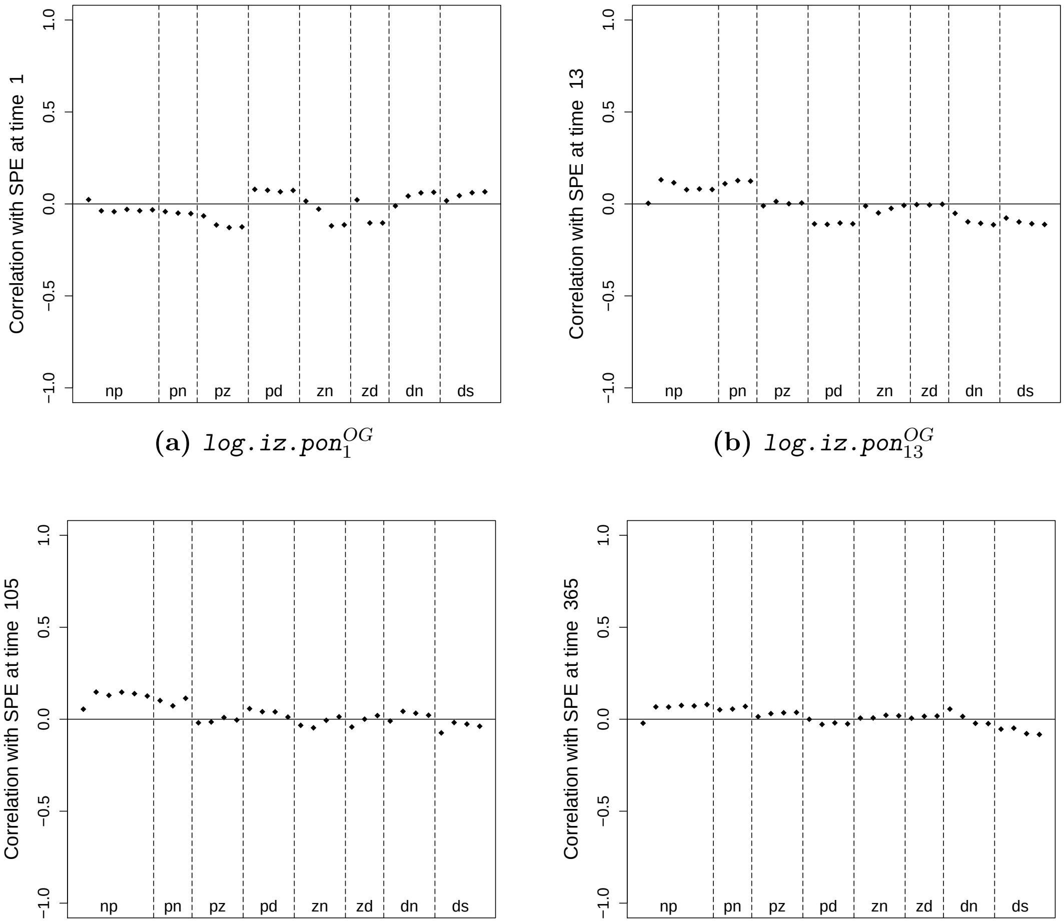
Figure C.3: *SPE at time 105 for each emulator over its validation data, with* $n(0,1)$ *shown by a solid line.*

C.2 Validating intermediate to output emulators

In order to be able to trust the conclusions drawn from studying the emulator of one simulator over the data of another, we must have confidence in the emulators themselves. A particularly important aspect of this is the behaviour of the SPE when the emulator is used over data from the same simulator. If these show systematic trends with intermediate variables, this would undermine any inferences made in Section 6.6.3. In this section we validate the intermediate to output variable emulators built from OG948 and HAD1005, using OG947 and HAD1007 respectively.

The mean, standard deviation and distribution of each SPE is shown in Figure C.3. By attempting to emulate these SPEs in section 6.6.3, we have already seen that there is little systematic behaviour in them, shown by the summaries in Table 6.15. This could be further investigated by studying the properties of the regions leading to the highest and lowest SPE values.

Figures C.4 and C.6 show the correlations between each intermediate variable and the SPE. Figures C.5 and C.7 show correlations between SPE and second order combinations of intermediate variables. None of these plots shows any marked effect



(a) $\texttt{log.iz.pon}_{1}^{OG}$ **(b)** $\texttt{log.iz.pon}_{13}^{OG}$

(c) $\texttt{log.iz.pon}_{105}^{OG}$ **(d)** $\texttt{log.iz.pon}_{365}^{OG}$

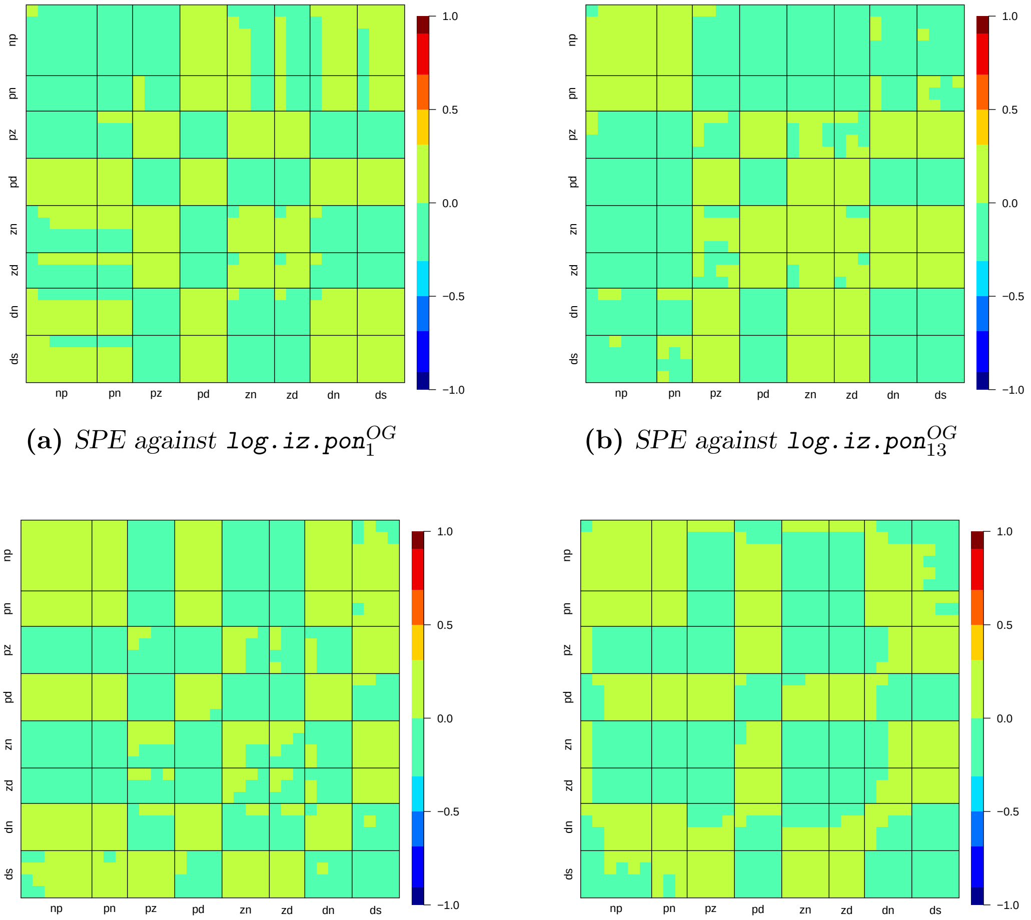
Figure C.4: *Correlations between each intermediate variable in OG947 and the SPE using an emulator built from OG948.*

of any intermediate variables on the SPE, allowing us to put our confidence in the emulators.

C.3 Combining the emulators

The emulators from both stages of intermediate variable emulation can be combined to create an emulator from input to output variables. Samples of size n can be generated from this emulator's distribution in the following way:

1. For each input point, generate a sample of size n from the intermediate variable space, using the emulator of input to intermediate variables.



(a) *SPE against* `log.iz.pon`$_1^{OG}$

(b) *SPE against* `log.iz.pon`$_{13}^{OG}$

(c) *SPE against* `log.iz.pon`$_{105}^{OG}$

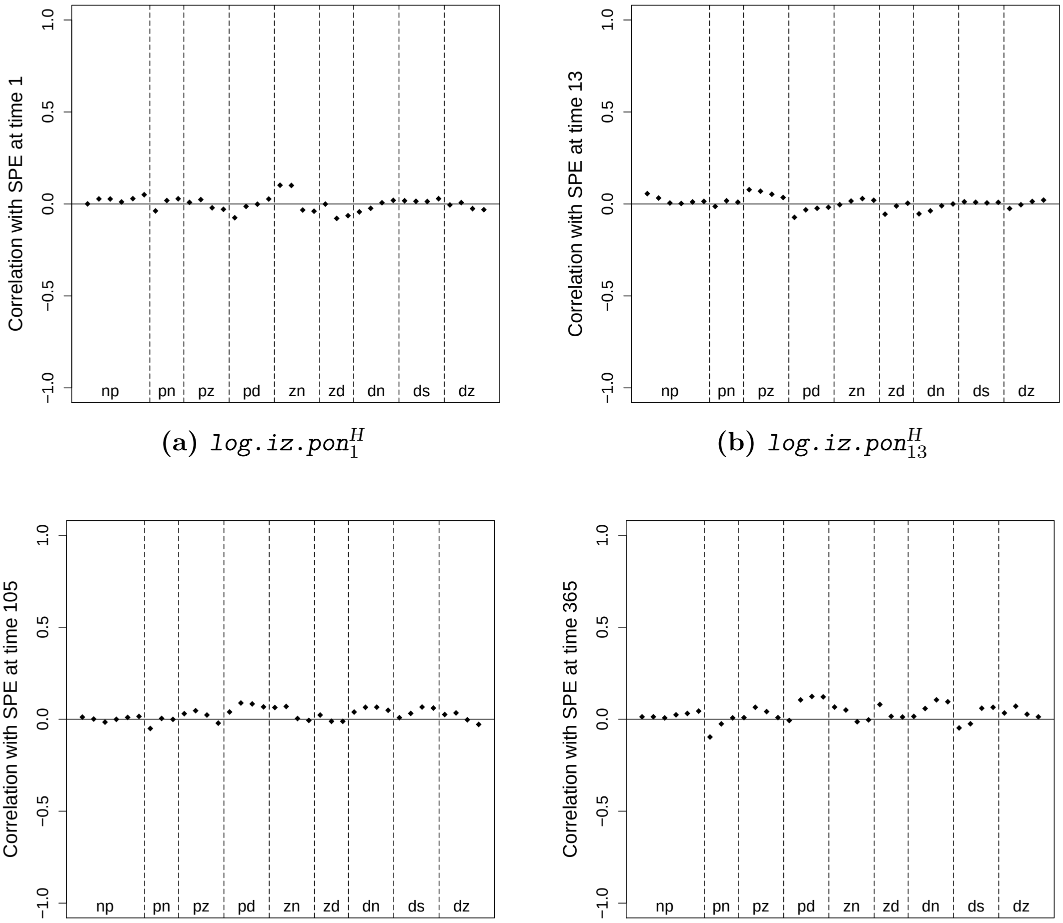
(d) *SPE against* `log.iz.pon`$_{365}^{OG}$

Figure C.5: *Correlations between products of pairs of intermediate variables in OG947 and the SPE using an emulator built from OG948.*

2. For each of these points, generate a random point from the output space, using the intermediate to output variable emulator.

Each of these samples can then be compared with a sample of size n from the input to output emulator's distribution at the same input point.

This validates the entire process, but in particular the amount of information kept in the emulator. If a crucial aspect of one of the simulators has been omitted while selecting the intermediate variables, these emulators will perform poorly. If the dimension reduction does not adequately represent the intermediate variables, the samples will also be poor.



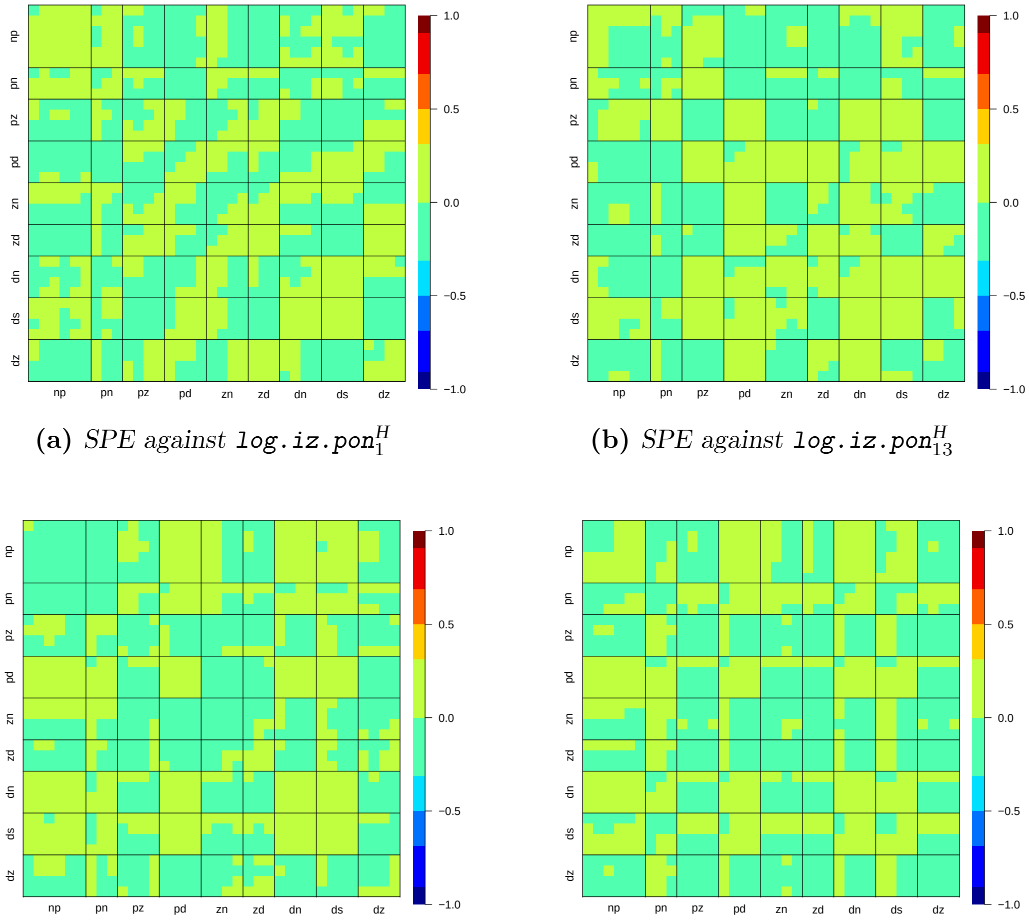
(a) $\mathtt{log.iz.pon}_1^H$ **(b)** $\mathtt{log.iz.pon}_{13}^H$

(c) $\mathtt{log.iz.pon}_{105}^H$ **(d)** $\mathtt{log.iz.pon}_{365}^H$

Figure C.6: *Correlations between each intermediate variable in HAD1007 and the SPE using an emulator built from HAD1005.*

Problems could also arise from the independence between the input to intermediate variable emulators. Here, for example, the `iz.pn` variables have been emulated jointly, but independently of all other intermediate variables. It may be that this loss of structure leads to a much poorer representation of the simulator. If performance of the combined intermediate variable emulators is poor compared to the standard emulator, each part of the process should be studied carefully until the causes are found.

Various summaries of the emulators' performances could be used, but an effective plotting strategy is to compare the samples to the true output value using boxplots, as in Figures C.8 and C.9. These show time series from OG98 and HAD119, two



(a) *SPE against* $\texttt{log.iz.pon}_{1}^{H}$ **(b)** *SPE against* $\texttt{log.iz.pon}_{13}^{H}$

(c) *SPE against* $\texttt{log.iz.pon}_{105}^{H}$ **(d)** *SPE against* $\texttt{log.iz.pon}_{365}^{H}$

Figure C.7: *Correlations between products of pairs of intermediate variables in HAD1007 and the SPE using an emulator built from HAD1005.*

datasets created in the example in Section 6.5.1, with boxplots of samples of size 100 from the standard input to output emulator (in red, on the right of each pair) and from the combined intermediate variable emulator (in blue and on the left).

In most plots, the distributions are very similar, indicating that the intermediate variable emulation has been done well. There are few input points (for example the top-right plot in Figure C.8, showing run 6 of OG98) for which the intermediate variable emulator performs relatively poorly. There are also some (for example the final point in the first page of Figure C.8, corresponding to run 56) for which the intermediate variable emulator is much more accurate than the standard.

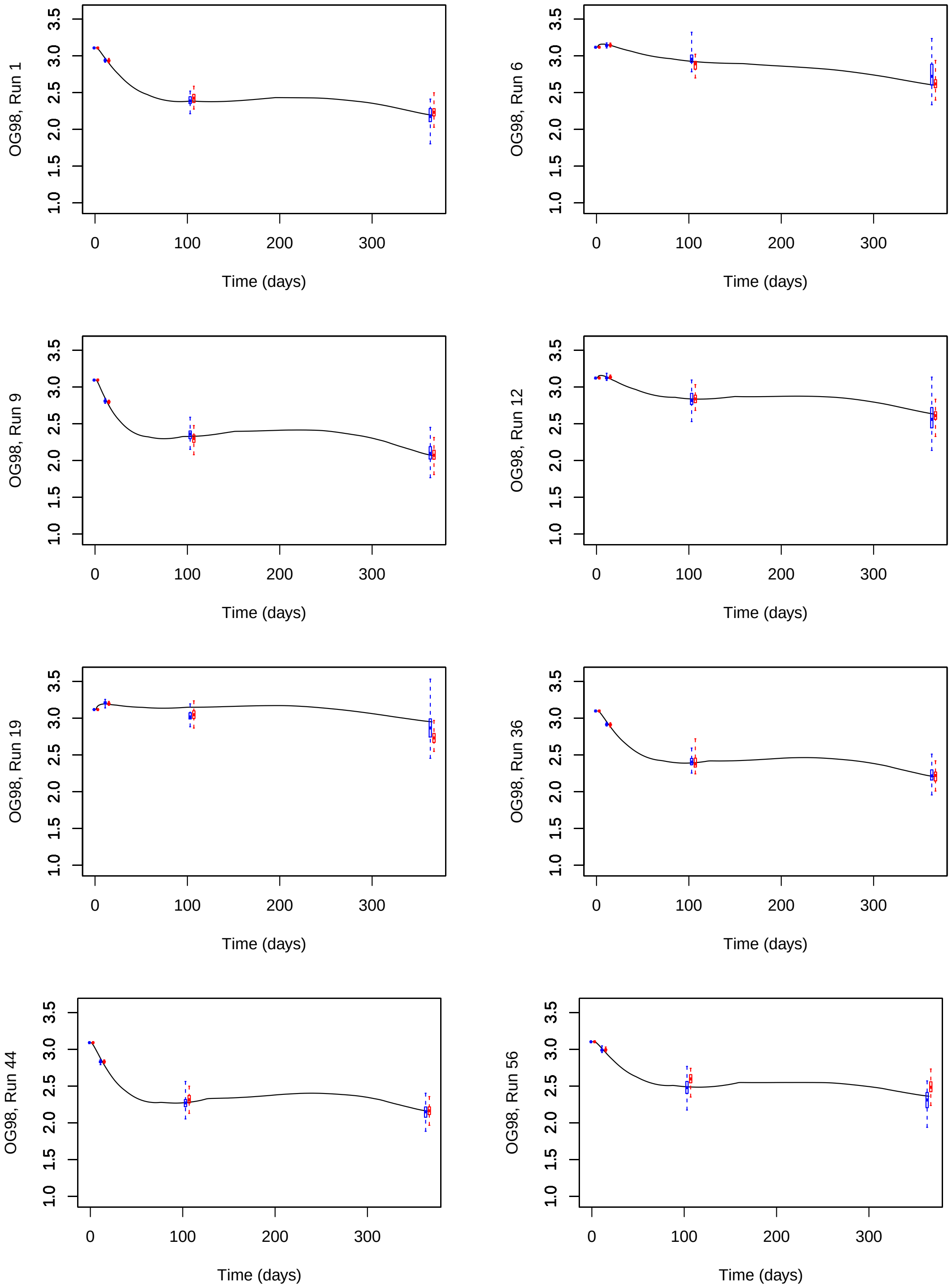


Figure C.8: *Intermediate variable and standard emulators for some points from OG98. The line shows the OG98 time series of log(iz.pon), the blue box-plots (left in each pair) summarise 100 draws from an intermediate variable emulator at each principal variable time point, the red box-plots (right) summarise 100 draws from a standard emulator.*

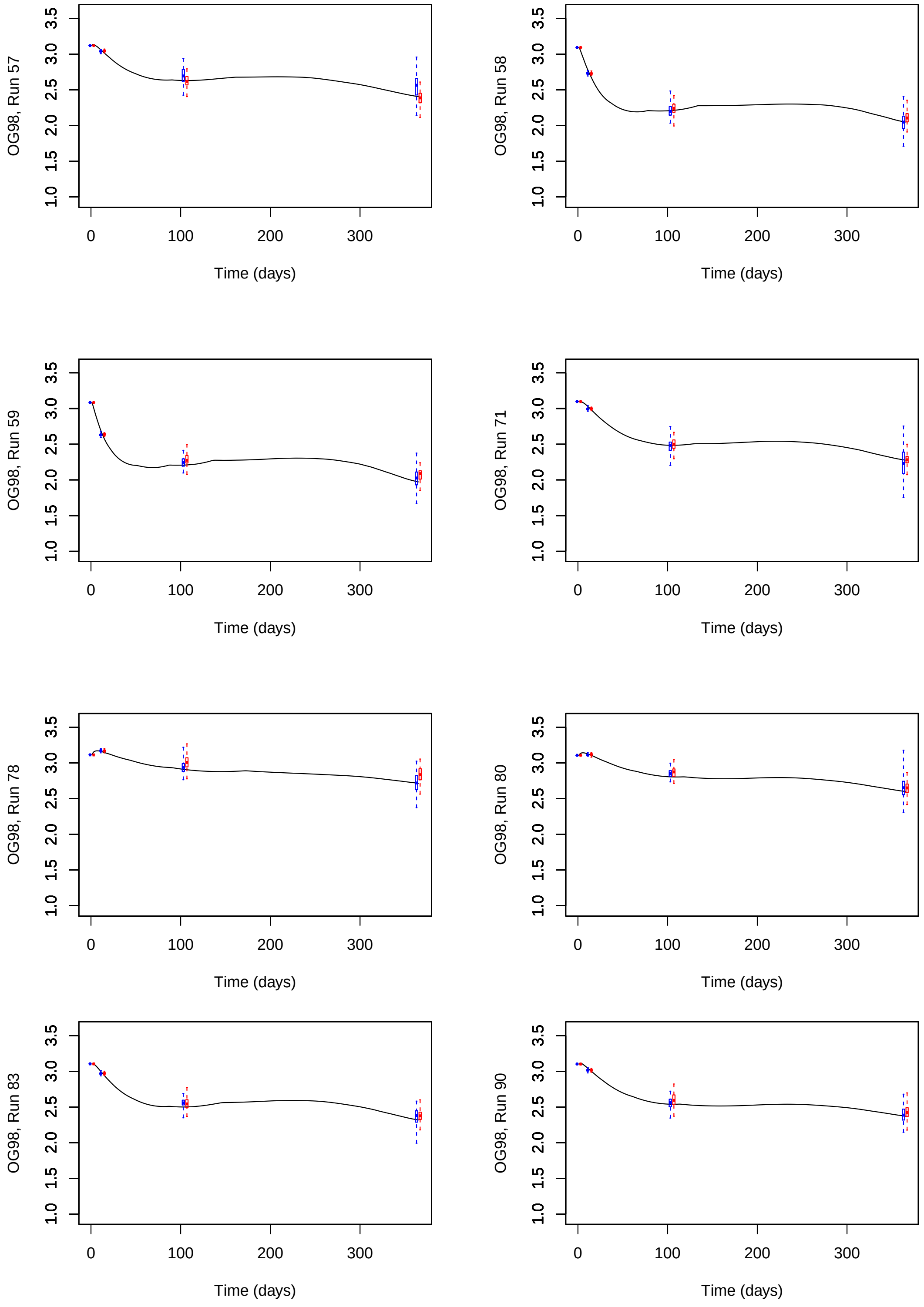


Figure C.8: *Intermediate variable emulators compared to standard for OG99NPZD, continued.*

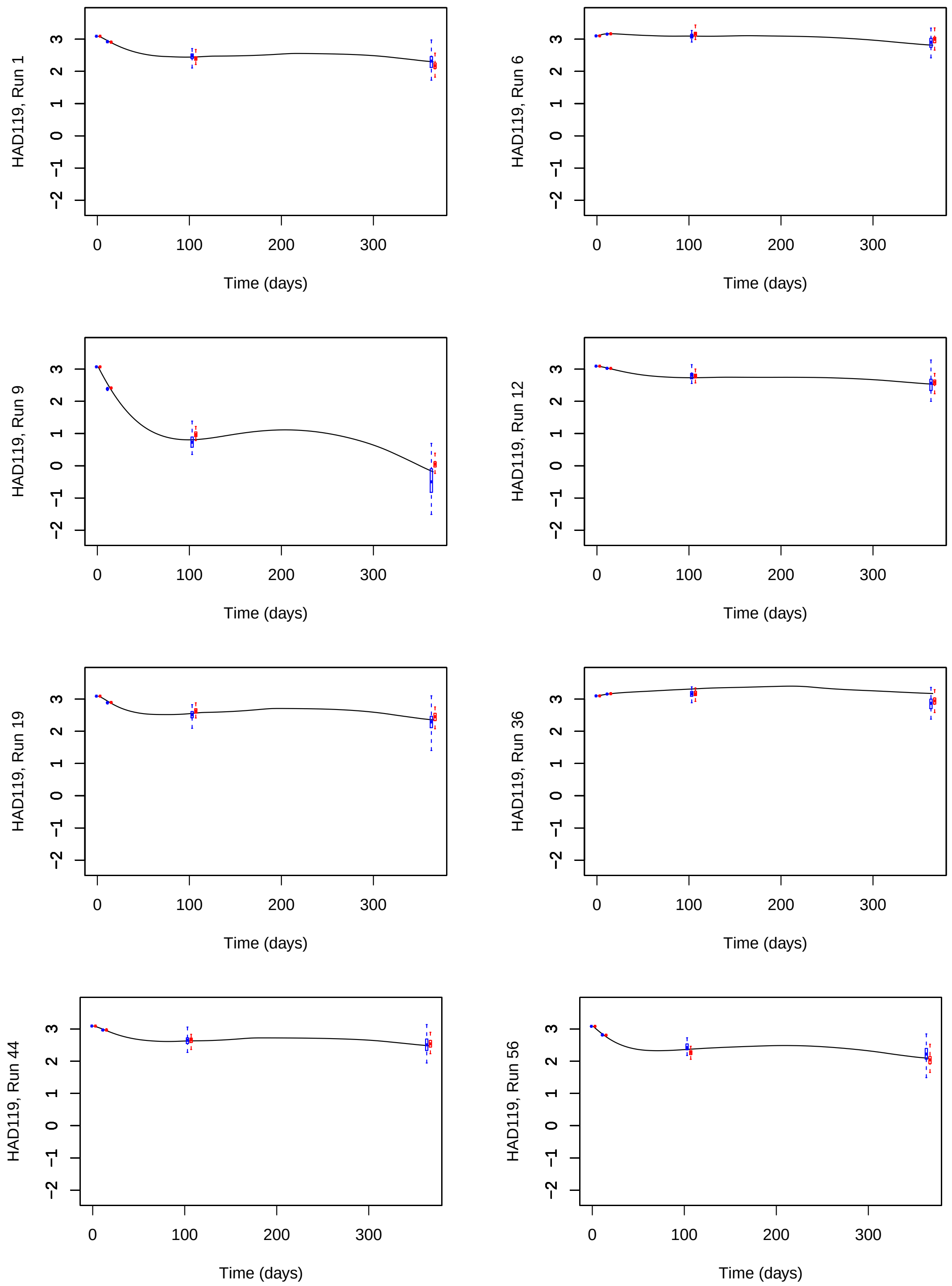


Figure C.9: *Intermediate variable and standard emulators for some points from HAD119. The line shows the HAD119 time series of log(iz.pon), the blue box-plots (left in each pair) summarise 100 draws from an intermediate variable emulator at each principal variable time point, the red box-plots (right) summarise 100 draws from a standard emulator.*

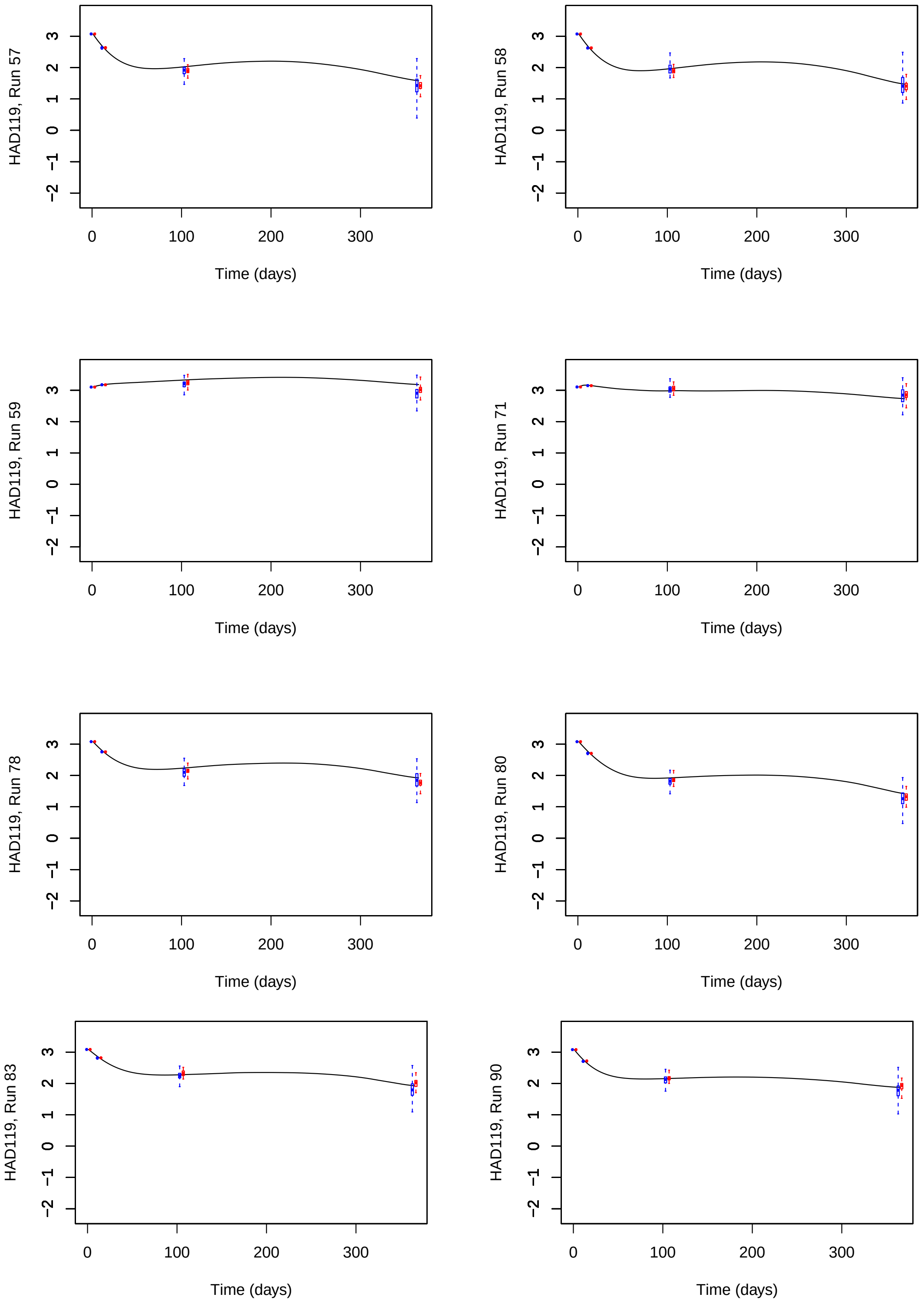


Figure C.9: *Intermediate variable emulators compared to standard for HadOCC, continued.*

Appendix D

S4 emulation code

D.1 Core emulator

This appendix works through the core emulator code described in Chapter 7. Class definitions match those in Table 7.1, which gives descriptions of each of the slots.

First of all, in order to be able to have slots that can belong to different classes, some class unions must be defined.

```
setClassUnion("list_or_null", members=c("list","NULL"))
setClassUnion("vec_or_null", members=c("vector","NULL"))
setClassUnion("mat_or_null", members=c("matrix","NULL"))
setClassUnion("df_or_null", members=c("data.frame","NULL"))

setClassUnion("list_or_vec", members=c("vector","list"))
setClassUnion("df_or_vec", members=c("vector","data.frame"))
setClassUnion("missing_or_log", members=c("missing", "logical"))
setClassUnion("missing_or_vec", members=c("missing", "vector"))
setClass("summary.lm")
setClass("summary.lm_or_null")
setClassUnion("summary.lm_or_null",members="NULL")
setIs("summary.lm", "summary.lm_or_null")

setClassUnion("md_or_null", members=c("model.data","NULL"))
setClassUnion("missing_or_md", members=c("missing", "model.data"))

# Forming an S4 class analogous to the S3 class "try-error"
```

```
setClass("try_S4")
setOldClass("try-error",prototype=tr, S4Class="try_S4")
setClassUnion("matrix_or_error", members=c("matrix", "try_S4"))
```

The data classes “model.data” and “model.data.out” can then be defined. The representation gives the name of each slot, and the class to which it belongs (the same information given in Table 7.1). The line `contains = "model.data"` shows that `model.data.out` is a subclass of `model.data`.

```
setClass(
  "model.data",
  representation(
    input = "data.frame",
    oldrange = "data.frame"
  )

)
setClass(
  "model.data.out",
  representation(
    input = "data.frame",
    oldrange = "data.frame",
    outdf = "data.frame",
    outname = "vector"
  ),
  contains = "model.data"
)
```

In order to build objects of these classes, creator functions should be written. Firstly, a generic function `model.data` is made. This specifies the names of the arguments any methods for this function takes.

```
setGeneric("model.data",
  function(tc.data, old.range, name.out, method, crit)
    standardGeneric("model.data")
)
```

Methods can now be defined, corresponding to situations where each argument is of a particular class. The vector of classes is the *signature* of the method, and matches the list of arguments in the generic function. The classes “ANY” (which matches any class) and “missing” (where the argument need not be entered) are useful for flexibility in defining methods. The final two arguments are not used until the core structure is extended to handle more complicated sorts of simulator data.

Sub-classes and super-classes are also at work here. For example, an object of class “character” is also of class “vector”, as is an object of class “numeric”, and so a method will match any of these to a signature requiring a vector. The function `new` creates an object of the class given as the first argument, using the following arguments to fill its slots (so long as they fit the class definition).

```
setMethod("model.data",
  c("data.frame", "data.frame", "vector", "missing", "missing"),
  function(tc.data, old.range, name.out){
    if(length(name.out)>1)
      outvec <- data[,match(name.out, names(data))]
    else if(length(name.out)==1)
      outvec <- data.frame(data[,match(name.out, names(data))])
    names(outvec) <- name.out

    names.or <- names(old.range)
    names.data <- names(data)
    match.names <- match(names.or, names.data, nomatch=F)

    input.df <- data[,match.names]
    new("model.data.out",
      input = input.df, oldrange = old.range, outdf = outvec, outname = name.out
    )
  }
)
setMethod("model.data",
  c("data.frame", "data.frame", "missing", "missing", "missing"),
  function(tc.data, old.range){
    names.or <- names(old.range)
    names.data <- names(data)
    match.names <- match(names.or, names.data, nomatch=F)

    input.df <- data[,match.names]
    new("model.data",
      input = input.df, oldrange = old.range
    )
  }
)
```

The emulator class `em.multi` can now be defined in a similar way:

```
setClass(
  "em.multi",
  representation(
    data.obj = "model.data.out",
    names.out = "character",
    reg.obj = "reg.func",
    cm.obj = "corr.mats",
    HcmH = "matrix",
    chol.HcmH = "matrix_or_error",
    beta.gls = "matrix",
    sigma.gls = "matrix"
  ),
)
```

and a generic function `em.multi` made as a creator function:

```
setGeneric("em.multi",
  function(data, reg, corrlen, outnames=NULL)
  standardGeneric("em.multi")
).
```

The most high level method, which requires only a `model.data.out` object, a `reg.func` object and correlation length choices, is defined first. The first two lines of the function implement the option to emulate only some of the outputs from the `model.data.out` object. The function `out.name` is an accessor function, accessing the `outname` slot of a `model.data.out` object.

```
setMethod(
  "em.multi",
  c("model.data.out", "reg.func", "df_or_vec", "missing_or_vec"),
  function(data, reg, corrlen, outnames){
    if(missing(outnames))
      outnames <- out.name(data)
    H <- des.mat(data, reg)
    Y <- out.vec(data, name = outnames)
    if(length(outnames)==1){
Y <- data.frame(Y)
names(Y) <- outnames
    }
    n.data <- nrow(Y)
    q <- ncol(H)
    if(is.numeric(corrlen)){
      corr.mat.obj <- corr.mats(data, corrlen)
    } else if (tolower(corrlen) == "estimate"){
message(sprintf(
  "Estimating correlation lengths using %s as output",
  outnames[1]
))
dist.array <- da_listp(rescale(data, new.range=c(-1,1), out.col=F), p=2)
corrlen.val <- est.corrlen.uni(
  da=dist.array,
  H=H,
  y=out.vec(data, name = outnames[1])
)
corr.mat.obj <- corr.mats(data, corrlen.val)
    }
    if(is.null(corr.mat.obj@cholcm)){
      HcmH <- t(H)%*%corr.mat.obj@cminv%*%H
      HcmY <- t(H)%*%corr.mat.obj@cminv%*%Y
    } else {
      alp.cmH <- backsolve(corr.mat.obj@cholcm, H, transpose = T)
      HcmH <- t(alp.cmH)%*%alp.cmH
      alp.cmY <- backsolve(corr.mat.obj@cholcm, Y, transpose = T)
      HcmY <- t(alp.cmH) %*% alp.cmY
    }

    chol.HcmH <- try(chol(HcmH), silent=T)
    if(class(chol.HcmH)=="try-error"){
      HcmH.inv <- ginv(HcmH)
    } else {
      alp.s2h <- backsolve(chol.HcmH, HcmY, transpose=T)
      alp.bh <- backsolve(chol.HcmH, diag(ncol(chol.HcmH)), transpose=T)
      HcmH.inv <- t(alp.bh)%*%alp.bh
    }
    beta.gls <- HcmH.inv%*%HcmY

    out.err <- Y - H%*%beta.gls
    if(is.null(corr.mat.obj@cholcm)){
      sig.gls <- (1/(n.data - q))*t(out.err)%*%corr.mat.obj@cminv%*%out.err
    } else {
      alp.sig <- backsolve(corr.mat.obj@cholcm, out.err, transpose = T)
      sig.gls <- (1/(n.data - q))*t(alp.sig)%*%alp.sig
    }
    if(!is.matrix(sig.gls))
      sig.gls <- matrix(sig.gls, nrow=1, ncol=1)

    new("em.multi",
      data.obj = data, names.out = outnames,
      reg.obj = reg, cm.obj = corr.mat.obj,
      HcmH = HcmH, chol.HcmH = chol.HcmH,
      beta.gls = beta.gls , sigma.gls = sig.gls
    )

  }
)
```

More methods can now be defined that use different information to arrive at a collection of objects that can be used to build an `em.multi` object. The method below accepts lists for the second argument. The list may contain either functions to be used for the regression surface, or criteria for building the surface. The function `reg.func` creates the `reg.func` object, and dispatches the correct method depending on which of these it is given.

```
setMethod(
  "em.multi",
  c("model.data.out", "list", "df_or_vec", "missing_or_vec"),
  function(data, reg, corrlen, outnames){
    if(missing(outnames))
      outnames <- out.name(data)
    if(is.function(reg[[1]]))
      reg.obj <- reg.func(func.list)
    else
      reg.obj <- reg.func(data, reg, output.name = outnames)
    em.multi.fun(data, reg.obj, corrlen, outnames)
  }
)
```

Finally, the class `ep.multi`, holding prediction information, is created.

```
setClass(
  "ep.multi",
  representation(
    mod = "em.multi",
    xnew = "data.frame",
    loc = "matrix",
    scale = "array",
    deg.f = "numeric"
  )
)
```

This is generated in the usual way, using a creator function. Some comments show roughly what parts of the function are doing.

```
setGeneric("ep.multi",
  function(xnew, mod.obj, names.out)
  standardGeneric("ep.multi")
)

setMethod(
  "ep.multi",
  c("data.frame", "em.multi", "missing_or_vec"),
  function(xnew, mod.obj, names.out){
    ep.multi.fun(xnew, mod.obj, names.out)
  }
)

ep.multi.fun <- function(
  xnew,
  mod.obj,
  names.out
){

# The following lines arrange the data, forming a model.data object with the
# new data, and building both design matrices.

  old.range <- range.df(mod.obj@data.obj)
  new.data <- model.data(xnew, old.range)
  xnew.r <- rescale(new.data, new.range = c(-1,1))

  names.out <- out.name(mod.obj)
  n.out <- length(names.out)

  n.new <- nrow(xnew)
  H.new <- des.mat(new.data, mod.obj@reg.obj)
  H.old <- des.mat(mod.obj@data.obj, mod.obj@reg.obj)
  n.old <- nrow(H.old)
  q <- ncol(H.new)

# The following lines use the design matrices and some information from the
# em.multi object to find the location matrix of the predictions'
# distributions

  loc.new1 <- t(mod.obj@beta.gls)%*%t(H.new)

  out.old <- out.vec(mod.obj@data.obj, name = mod.obj@names.out)
  err.old <- out.old - H.old%*%mod.obj@beta.gls
  corr.new.old <- corr.mats(mod.obj@data.obj, mod.obj@cm.obj@corrlen, new.data)

  if(is.null(mod.obj@cm.obj@cholcm)){
    loc.new2 <- t(err.old)%*%mod.obj@cm.obj@cminv%*%cmat(corr.new.old)
  } else {
    alp.errcm <- backsolve(mod.obj@cm.obj@cholcm, err.old, transpose=T)
    alp.cmnm <- backsolve(mod.obj@cm.obj@cholcm, cmat(corr.new.old), transpose=T)
    loc.new2 <- t(alp.errcm) %*% alp.cmnm
  }

  loc.new <- loc.new1 + loc.new2

# The following lines compute the scale array, such that
# scale.array[i,i, , ] is the scale matrix for the ith output across all points
# scale.array[ , ,i,i] is the scale matrix for outputs at the ith point

  cm.new <- corr.mats(new.data, mod.obj@cm.obj@corrlen, inv=F)

  if(is.null(mod.obj@cm.obj@cholcm)){
    c.star1 <- cmat(cm.new)
- t(cmat(corr.new.old))%*%mod.obj@cm.obj@cminv%*%cmat(corr.new.old)
    cs2 <- t(H.new) - t(H.old)%*%mod.obj@cm.obj@cminv%*%cmat(corr.new.old)
  } else {
    alp.cs1 <- backsolve(mod.obj@cm.obj@cholcm, cmat(corr.new.old), transpose=T)
    alp.H.old <- backsolve(mod.obj@cm.obj@cholcm, H.old, transpose=T)
    c.star1 <- cmat(cm.new) - t(alp.cs1)%*%alp.cs1
    cs2 <- t(H.new) - t(alp.H.old)%*%alp.cs1
  }

  if(class(mod.obj@chol.HcmH)=="matrix"){
    alp.cs2 <- backsolve(mod.obj@chol.HcmH, cs2, transpose=T)
    csm2 <- t(alp.cs2)%*%alp.cs2
  } else {
    HcmH.inv <- solve(mod.obj@HcmH)
    csm2 <- t(cs2)%*%HcmH.inv%*%cs2
  }

  c.star.mat <- c.star1 + csm2

  scale.array <- array(0, c(n.out, n.out, n.new, n.new))
  for (i in 1:n.out){
    for (j in 1:n.out){
      scale.array[i,j,,] <- mod.obj@sigma.gls[i,j]*c.star.mat
    }
  }

  new("ep.multi",
    mod = mod.obj, xnew = xnew,
    loc = loc.new, scale = scale.array, deg.f = n.old-q
  )
}
```