# An S4 Object structure for emulation

## The approximation of complex functions

Rachel Oxlade

Department of Mathematical Sciences
University of Durham

Supervised by Peter Craig and Michael Goldstein

With thanks to CRISM for sponsorship
and NERC for funding

August 15, 2011

# Emulation

## A brief introduction

Computer models are examples of
**complex functions**
over **high dimensions**
that are **slow** to evaluate.

We would like to **predict** a model's behaviour
**without running the model**.

**An emulator is a statistical representation of a complex function**

> An emulator is a statistical representation of a complex function

For a collection $\tilde{\mathbf{x}}$ of input points it gives us

- a **probability distribution** for the function's value, $s\left(\tilde{\mathbf{x}}\right)$
- conditional on some known function values, $(\mathbf{x}, s(\mathbf{x}))$

We stipulate that

- at 'training points', where we know $s(x)$, the emulator gives the same value, with certainty
- at other points, the approximation should be 'plausible', and reflect our uncertainty.

We represent the function's value for input $\mathbf{x}$ as

$$s(\mathbf{x}) = \underbrace{\sum_{i=1}^{p} g_i(\mathbf{x})\beta_i}_{\text{Regression surface}} + \underbrace{\epsilon(\mathbf{x})}_{\text{Correlated error}} .$$

- The regression surface captures the general trend
- The correlated error term forces the emulator to interpolate the training data

# Emulation

## Why objects?

- **Encapsulation** - information that belongs together is collected as one object
- **Efficiency** - time-consuming computations can be performed just once
- **Tidiness** - changes and additions to code are simpler to make
- **Methods** - objects can be created from various beginnings using multiple dispatch

# Structure

## How it works (or how it *should* work. . . )

Crudely, there are three stages to emulation

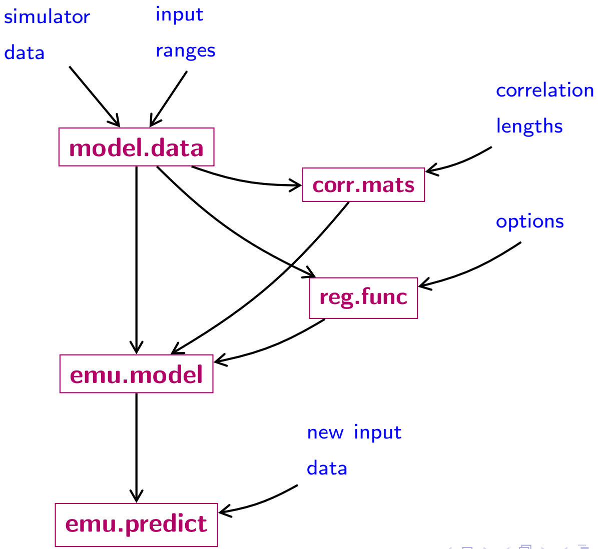
1. **Collect function data**, with which to train the emulator
2. **Make choices** about the emulator: regression functions, correlated error behaviour
3. Use the emulator to **predict** new function values.

This leads to three stages in the code:

1. **data.object** <-
   model.data(input and output data, function information)
2. **emulator.object** <-
   emu.model(**data.object**, regression and correlation choices)
3. **prediction.object** <-
   emu.predict(**emulator.object,** new input points)

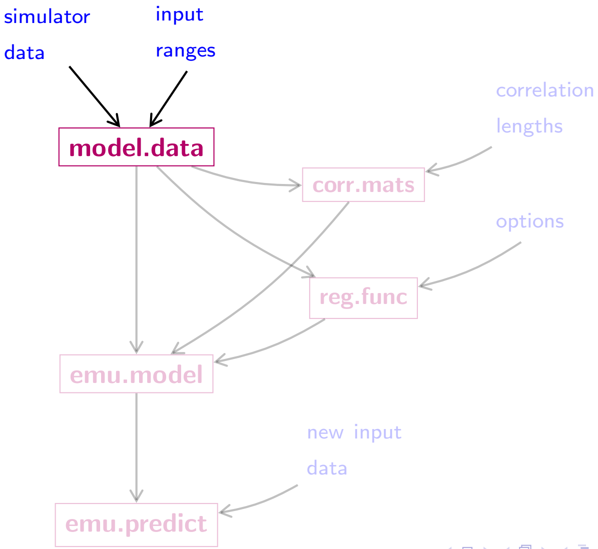
# Structure

## An overview



# Structure

## Organising model data



# Classes

## "model.data"

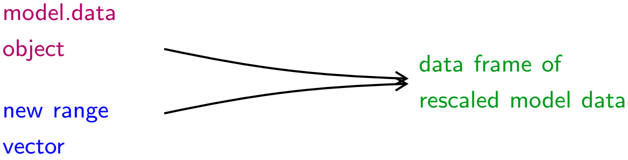
Ingredients :

- function data (possibly containing output values too)
- ranges for input variables

Slots:

- input: a data frame of input values
- oldrange: a data.frame of input ranges
- (optional) outvec and outname: output data

**"rescale"** - a method for "model.data"



# Classes

## How the data fits into the structure

Simulator output

Correlated error

$$s(\mathbf{x}) = \sum_{i=1}^{p} g_i(\mathbf{x})\beta_i + \epsilon(\mathbf{x})$$

Regression surface

# Classes

## "reg.func" - regression functions

- How many inputs should be active?
- Should the inputs be transformed?
- What order polynomial?
- How would we like to choose terms?
- Do we already know what functions

we'd like?

→

- Functions
- Active variables

$$s(\mathbf{x}) = \sum_{i=1}^{p} g_i(\mathbf{x})\beta_i + \epsilon(\mathbf{x})$$

↑

Regression surface

# Classes

## "corr.mats" - correlation matrix

Correlation is determined by **correlation lengths**

- Same values in each dimension?
- "optimise" them using the data?
- Add 'nugget' onto the diagonal?

→

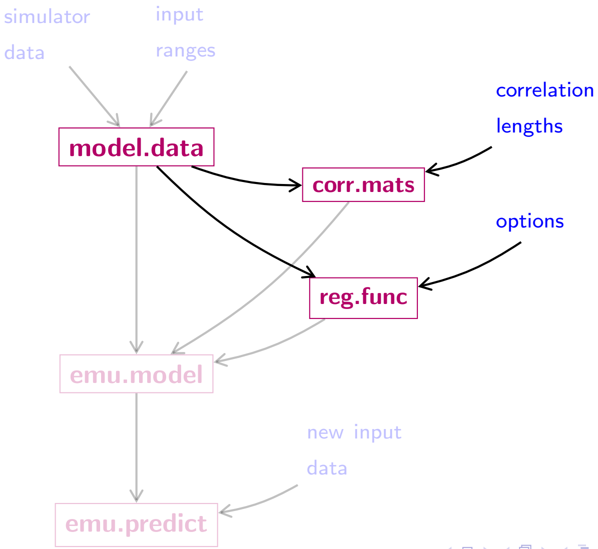
- Correlation matrix
- Cholesky factorisation of correlation matrix

$$s(\mathbf{x}) = \sum_{i=1}^{p} g_i(\mathbf{x})\beta_i + \epsilon(\mathbf{x})$$

Correlated error

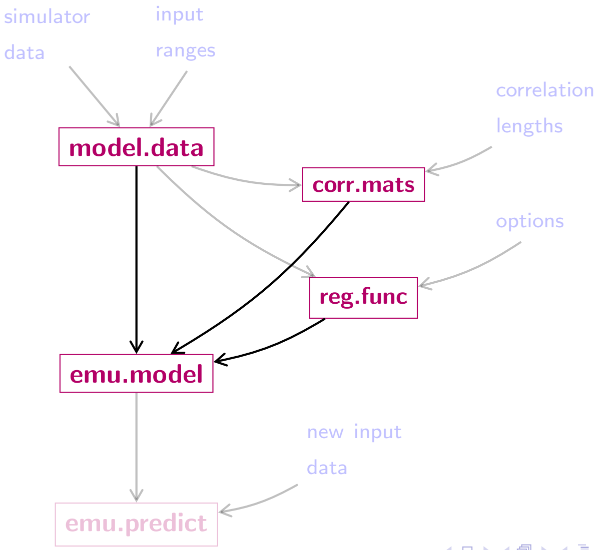
# Structure

## Creating correlation and regression objects



# Structure

## Collecting information together



# Classes

## "emu.model" - everything ready to predict

The data, regression and correlation objects together can make an "emu.model" object, containing

- data.obj ("model.data")
- func.obj ("reg.func")
- cm.obj ("corr.mats")

} Objects we've seen

- HcmH ("matrix")
- chol.HcmH ("matrix")

} Stored for computations

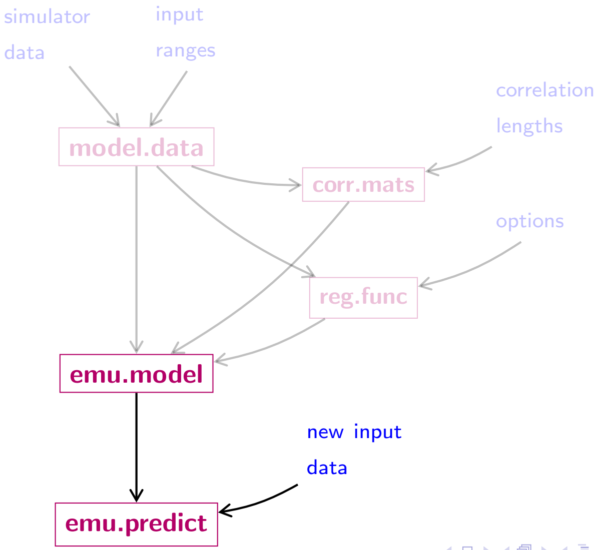
- sig.hat.sq ("numeric")
- beta.hat ("vector")

} Estimated residual variance and regression coefficients

This contains all the information we need to evaluate the probability distribution of output at new input points

# Structure

## Predicting new model output



## "emu.predict" - predicting new function values

Given model data $s(\mathbf{x})$, and new inputs $\tilde{\mathbf{x}}$,
$s\left(\tilde{\mathbf{x}}\right) \mid s\left(\mathbf{x}\right)$ has a **location-scale multivariate t-distribution**.

An "emu.predict" object contains

- **mod** - the "emu.model" used
- **xnew** - the new input points
- **loc** - vector of expected outputs (the location of the $t$-distribution)
- **scale** - the scale matrix (linked to variance)
- **deg.f** - the degrees of freedom of the $t$-distribution

One method, "**change.obj**", which requires

- **object** - an object (from this emulation structure)
- **changes** - a list of arguments to change

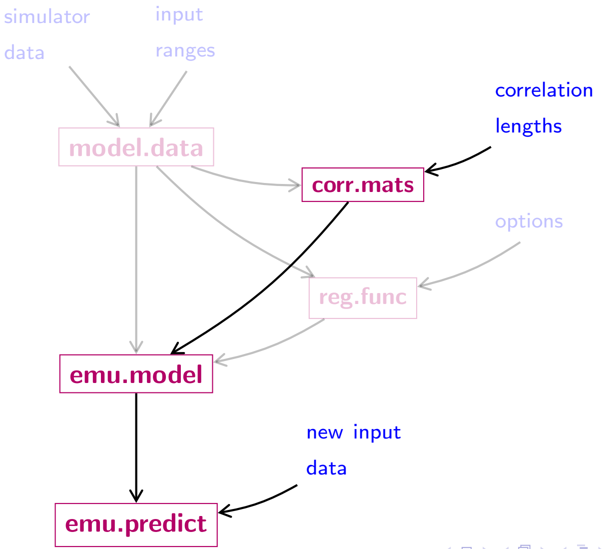
and creates a **new object** of the **same class**.

Advantages of this method:

- **quicker** - one command to remake the object
- **more transparent** - what's changed is clear
- **less error prone** - prevents use of wrong data (or deletion)

# Structure

## Using “change.obj” to change correlation length, from an “emu.predict” object



# Summary

S4 objects are an effective approach to emulation:

- **administration** - vital information for an emulator is held together (helpful for **reproducibility**)
- **efficiency** - costly calculations needn't be repeated
- **transparency** - class descriptions enforce structure
- **adaptability** - methods can be added / changed without upsetting the wider structure