Code Compendium: Multivariate and Functional Output Emulation

Suppmental Material for Adventures in Space-Time Modeling with Maike and Dave

Maike Holthuijzen (*maikeh@vt.edu*) Dave M. Higdon (*dhigdon@vt.edu*)
Sierra N. Merkes (*smerkes@vt.edu*)

June 3rd, 2025

Contents

0. A Simple Example

We begin by constructing the functional model depicted in Figure 2 of the Chapter 7. Next, we source the required R files from GitHub, which provide several utility functions used in the analysis. Finally, we define the two-dimensional input space over which the model (Eq 7.1) is evaluated.

```r
source("Cov_functions.R")
source("Estimation_1D.R")
source("Estimation_Basis.R")

nc = 10 # Number of input settings
s = 11  # Number of vector values produced by f(x)

# Creates a grid of points where x[,2] represents x and x[,1] represents t
t1 = seq(0,1, length = s)
x1 = seq(0,1, length = nc)
x = expand.grid(t1,x1)

# Input grid points into the deterministic function to be emulated
f = (x[,2]+1)*cos(pi*x[,1]) + 0.03*(exp(x[,2]))
```

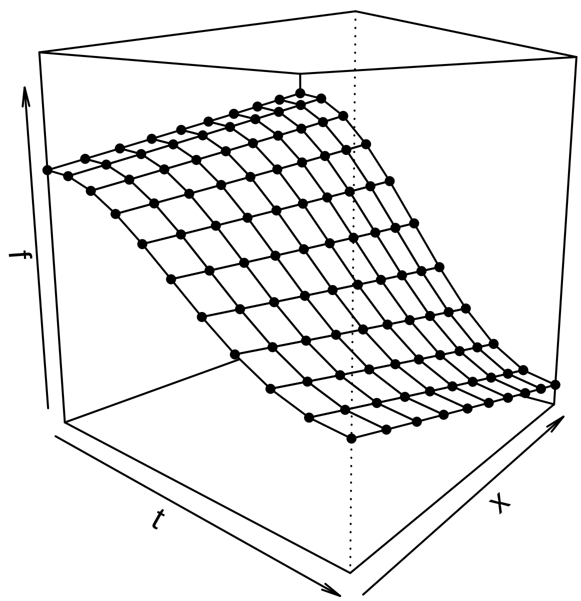


Figure 1: A simple functional model to be emulated

We assume the function f can be modeled as a zero-mean Gaussian Process (GP) with a covariance structure given by the Kronecker product of spatial and temporal covariance matrices. From Section 0.1, we can model $\text{vec(Y)} = f$ using a Kronecker representation for the correlation

$$f \sim N(0, \sigma^2(R_x \otimes R_t) + \nu I). \tag{1}$$

1. Estimating Parameters

We assume the prior for f is given by Equation 1 and aim to derive the posterior distribution, $f|D_n$ where D_n is the data. We estimate the parameters, $\phi_x, \phi_y, \sigma^2, \nu$, using the functions in `Estimation_1D.R`.

The likelihood function, derived from the matrix normal distribution with an added nugget term, is given by (Stegle et al., 2011):

$$L(\mathbf{y_c}; \nu, \sigma^2, \phi) \propto |\sigma^2 S_x \otimes S_t + \nu I|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\text{vec}\left(U_t^T Y U_x\right)^T \left(\sigma^2 s_x \otimes S_t + \nu I\right)^{-1} \text{vec}\left(U_t^T Y U_x\right)\right\},$$

given the eigendecomposition $R_x = U_x S_x U_x^T$ and $R_t = U_t S_t U_t^T$, where $UU^T = I$. Fortunately, it is not necessary to explicitly compute the Kronecker products when evaluating the likelihood. Notably, the matrix $(S_x \otimes S_t + \nu I)^{-1} = \nu^{-\mathbf{1}}$ is diagonal, allowing both its inverse and the log-determinant to be computed efficiently. In particular, the log-determinant simplifies to $\log\left|\sigma^2 R_x(\phi) \otimes R_t(\phi) + \nu I\right| = \mathbf{1}^T \log(\nu)$. The remaining quantity of interest, the sum of squares, can also be evaluated efficiently. A summary of these calculations is provided in Table 0.1 of Chapter 7, and pseudocode in Section 1.1.1 outlines how the log-determinant and sum of squares are efficiently computed and implemented in the parameter estimation functions found in `Estimation_1D.R`.

Estimates for the covariance parameters of R_x and R_t can be obtained via maximum likelihood. However, the maximum likelihood estimates (MLEs) of the marginal variance σ^2 and lengthscale ϕ are not consistent; that is, their accuracy does not improve as the sample size increases. As a first step, we compute the pairwise distance matrices for the input dimensions x and t using the `get_distmat` function from `Cov_function.R`:

```
Tdist = get_distmat(t1, t1) # 11 x 11 matrix
Xdist = get_distmat(x1, x1) # 10 x 10 matrix
```

We then estimate the covariance parameters using the functions provided in `Estimation_1D.R`. Optimization is performed using the `nmkb` function from the `dfoptim` R package which implements the Nelder-Mead algorithm for derivative-free optimization. The function `Estimate_params` takes the following arguments:

- `distC` and `distR`: the first and second distance matrices in the Kronecker product (order matters)
- `z`: the vector of computer model output to be emulated.

The `nkmb` function returns the parameters estimates in the following order:

1. scale estimate for the first covariance, R_x, matrix ϕ_x,
2. scale estimate for the second covariance, R_t, matrix ϕ_y,
3. nugget ν,
4. marginal variance,σ^2.

Sensible upper and lower bounds for the optimization procedure are also specified below:

```
# Function assumes the Kronecker product: kronecker(C, R), so order matters
params = nmkb(c(0.5, .5, .0001, .5),
              Estimate_params, distC = Xdist, distR = Tdist, z=f,
              lower = c(0.1, 0.1, 1e-6, .1), upper = c(100, 100, 100, 100))
```

The parameter estimates and associated R code to print the values are:

Parameter	ϕ_x	ϕ_y	ν	σ^2
Estimates	12.1517375	1.1269516	10^{-6}	99.999987
R Code	`params$par[1]`	`params$par[2]`	`params$par[3]`	`params$par[4]`

1.1 Efficiently caluculating the sum of squares and log determinant

In Chapter 7, Table 0.1, we present the fundamental calculations for exploiting the Kronecker structure in a multivariate emulator. The following *pseudocode* demonstrates how the log-determinant and sum of squares are computed efficiently and subsequently used in the parameter estimation functions in `Estimation_1D.R`. The process begins by performing singular value decomposition (SVD) on R_x and R_t to obtain their respective components. Specifically, we perform **eigen-decomposition** on $R_x = U_x S_x U_x^T$ and $R_y = U_y S_y U_y^T$, as shown in Table 0.1

```
svdx = svd(Rx) # Eigendecomposition of Rx
# Obtaining components from Eigendecompsition
Ux = svdx$u
Sx = svdx$d
Ux_t = t(svdx$v)

svdt = svd(Rt) # Eigendecomposition of Rt
# Obtaining components from Eigendecompsition
Ut = svdt$u
St = svdt$d
Ut_t = t(svdt$v)
```

In Table 0.1, we under the variance and inverse label we note that $(S_x \otimes S_t + \nu I)^{-1} = \text{diag}(\mathbf{v}^{-1}) = \text{diag}(\lambda)$ which is a diagonal matrix that we compute as follows:

```
nu = params$par[3]                   # obtaining the nu estimate
v = sigma2 * kronecker(Sx, St) + nu # compute v
lambda = 1/v
```

In Table 0.1, we perform multiplication of $\left(U_x^T \otimes U_t^T\right) \mathbf{y_c} = \text{vec}\left(U_t^T Y U_x\right)$ where $Y = \text{matrix}\left(\mathbf{y_c}, \text{nrow} = s, \text{ncol} = n_c\right)$. In this context, $\mathbf{y_c}$ is represented by f.

```
Ymat = matrix(f, nrow = nc, ncol = s)  # Construct the matrix Y
Yvec = as.vector(Ut_t %*% Ymat %*% Ux) # Performing Multiplication in Table 0.1
```

Lastly, we put the various elements together to evaluate the sums of squares and the log log-determinant efficiently using the Kronecker products.

```
ssqKronecker = (Yvec)^2 %*% lambda # Sums of Squares calculation
LogDet = sum(log(lambda))          # log determinant calculation in Table 0.1
```

1.2 Computing the conditional mean and covariance

In this section, we use the parameter estimates from Section 1.1 to obtain the (posterior) conditional mean and covariance. In other words, we obtain the posterior predictive distribution $f|D_n$, which is also multivariate normal distribution where D_n represents the data. Our goal is obtain computer model predictions $Y^*_{s\times n^*}$ at a new collection of n^* input settings.

First, we build the covariance matrices, R_x and R_t, using their respective scale estimates, ϕ_x and ϕ_t . Note that we assume Gaussian covariance functions throughout.

```
phi_x = params$par[1] # scale estimate for x
phi_t = params$par[2] # scale estimate for t

# Estimating the covariance matrices
Rx = exp( -(Xdist / phi_x)^2 )
Rt = exp( -(Tdist / phi_t)^2 )
```

Next, we create a prediction grid for new input locations, denoted by x_{new}. To facilitate predictions, we define an augmented input space, x^* which combines the original inputs x with the new inputs:$x^* = [x, x_{new}]$. We then construct the cross-covariance matrix $R_{x^*,x}$ and the covariance matrix of $R_{x^*} = R_{x^*,x^*}$, which are (sub) matrices defined in Section 0.1.1 in the textbook.

```
xnew = c(.135, .72, .97) # prediction grid of new locations, xnew

xstar_grid = c(x1, xnew) # make the augmented x grid, xstar

# build cross covariance matrix, R_x*,x
dist_xstar_x = get_distmat(xstar_grid, x1)
R_xstar_x = exp( -(dist_xstar_x / phi_x)^2 )

# build covariance matrix, R_x*,x*
dist_xstar_xstar = get_distmat(xstar_grid, xstar_grid)
R_xstar_xstar = exp( -(dist_xstar_xstar / phi_x)^2 )
```

For the calculation of the conditional mean, we perform an eigendecomposition of R_x and R_t, which is necessary for efficient computation of the conditional mean (see book chapter). Note that for this example, we only vary x, not t.

```
svdx = svd(Rx)  # Eigndecomposition for Rx
Ux = svdx$u
Sx = svdx$d
Ux_t = t(svdx$v)

svdt = svd(Rt)  # Eigndecomposition for Rt
Ut = svdt$u
St = svdt$d
Ut_t = t(svdt$v)
```

In Table 0.1 in the book chapter, we define the conditional mean as

$$E(\mathbf{y}^*|y_c) = \text{vec}\left(\sigma^2 \left[U_t S_t\right] V [U_x^T R_{xx^*}]\right)$$

where $V = \text{matrix}(\mathbf{v}, \text{nrow} = s, \text{ncol} = n_c)$ and $\mathbf{v} = \text{diag}(\lambda) \times \text{vec}\left(U_t^T Y U_x\right)$. We can efficiently obtain the conditional mean using Kronecker product properties and the *psuedocode* illustrated in Section 1.1.1.

To define V, we need to compute $\text{diag}(\lambda)$ and Y matrix. In Chapter 7 Table 0.1, we under the inverse label, we define $\text{diag}(\lambda) = (S_x \otimes S_t + \nu I)^{-1}$, and we define Y matrix $= \text{matrix}(y_c, \text{nrow} = s, \text{ncol} = n_c)$, under the multiplication label.

```r
nu = params$par[3]          # nu estimate
sig2 = params$par[4]      # sigma2 estimate
# calculate kronecker product of singular values Sx and St
lambda = diag(1/ (sig2 * kronecker(Sx, St) + nu))

# Construct (reshape) the response Y to be s * nc
Y_mat = matrix(f, nrow = s, ncol = nc)
```

Given $\text{diag}(\lambda)$ and Y matrix, we define $\mathbf{v} = \text{diag}(\lambda) \times \text{vec}\left(U_t^T Y U_x\right)$ and reshape the vector into a matrix defined as $V = \text{matrix}(\mathbf{v}, \text{nrow} = s, \text{ncol} = n_c)$.

```r
# Defining v and reshaping V
v = lambda %*% as.vector(Ut_t %*% Y_mat %*% Ux)
V = matrix(v, nrow = s, ncol = nc)
```

Lastly, we evaluate $E(\mathbf{y}^*|y_c) = \text{vec}\left(\sigma^2 \left[U_t S_t\right] V [U_x^T R_{xx^*}]\right)$.

```r
cond_mean = sig2 * as.vector(Rt %*% Ut %*% V %*% Ux_t %*% t(R_xstar_x) )
```

Let's plot the conditional mean, so that we can inspect the predicted mean on the grid of training points. In the code below, we only plot at test locations (x_{new}).

```r
# grid for plotting
plot_grid_train = expand.grid(t1, x1)
plot_grid = expand.grid(t1, xnew)

# Allows for differentiating train/test indices
train_idx = 1:length(x1)
train_idx_kron = 1:(length(train_idx) * length(t1))
test_idx_kron = (train_idx_kron[length(train_idx_kron)] + 1): (length(xstar_grid) * s)
```

1.2.1 Generating conditional realizations $(y^*|y_c)$

For large n, obtaining the conditional variance may not be computationally feasible. However, as stated in the book chapter, we can use a technique described in Nychka, Wikle, and Royle (2002) to generate realizations from the conditional predictive distribution.

Before diving into the details, let's first review the matrix-normal distribution. As discussed in Stegle et al. (2011), the matrix-normal distribution (with error) extends the multivariate normal distribution to matrix-valued random variables. Specifically, a random matrix $\mathbf{X}$ follows a matrix-normal distribution

$$\mathbf{X} \sim \mathcal{MN}_{n \times p}(\mathbf{M}, \mathbf{U}, \mathbf{V}),$$

if and only if its vectorized form $\text{vec}(\mathbf{X})$ follows a multivariate normal distribution:

$$\text{vec}(\mathbf{X}) \sim \mathcal{MVN}_{np}(\text{vec}(\mathbf{M}), \mathbf{V} \otimes \mathbf{U}). \tag{2}$$

where $\otimes$ denotes the Kronecker product. The probability distribution function (PDF) of the matrix-normal distribution is given by:

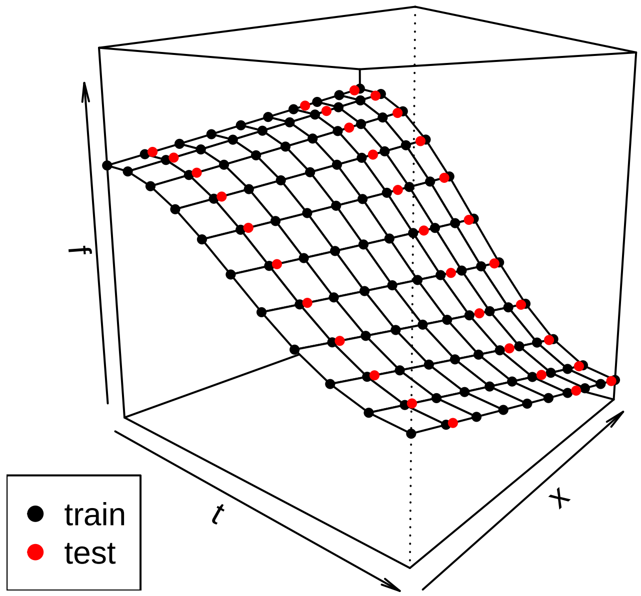


Figure 2: Conditional mean predictions (red) on the grid of training points (black)

$$P(\mathbf{X}|\mathbf{M},\mathbf{U},\mathbf{V}) = \frac{\exp(\mathrm{tr}([\mathbf{V^{-1}}(\mathbf{X}-\mathbf{M})^{\mathbf{T}}\mathbf{U^{-1}}(\mathbf{X}-\mathbf{M})])}{(2\pi)^{np/2}|\mathbf{V}|^{\mathbf{n/2}}|\mathbf{U}|^{\mathbf{p/2}}}$$
$$\mathbf{M} = n \times p \text{ mean matrix}$$
$$\mathbf{U} = n \times n \text{ row covariance matrix}$$
$$\mathbf{V} = p \times p \text{ column covariance matrix,}$$

where tr($\bullet$) represents the matrix trace of the matrix and exp($\bullet$) represents the natural exponential.

Now, suppose $\mathbf{Y}$ has a matrix-normal distribution with zero mean matrix $Z = \mathbf{0}_{n\times p}$ and covariance matrices $\mathbf{U}$ and $\mathbf{V}$. For a general mean matrix, $\mathbf{M}$, $\mathbf{Y}$ can be expressed as a linear transformation:

$$\mathbf{Y} = \mathbf{M} + \mathbf{LZR}, \tag{3}$$

that follows matrix-normal distribution with parameters $\mathbf{M}, \mathbf{LL^T}, \mathbf{R^TR}$ where $\mathbf{Z}$ is $n \times p$ matrix of independent standard normal random variables and $\mathbf{L}$ and $\mathbf{R}$ are transformation matrices such that $\mathbf{LL^T} = \mathbf{U}$ and $\mathbf{R^TR} = \mathbf{V}$. In Equation 3, $\mathbf{L}$ and $\mathbf{R}$ are be computed using the Cholesky decomposition of $\mathbf{U}$ and $\mathbf{V}$, respectively.

Alternatively, if we use an eigendecomposition, for better numerical stability, then the eigndecomposition of $\mathbf{U}$ is $\mathbf{LL^T} = \mathbf{U_u}\mathbf{D_u^{1/2}}\mathbf{D_u^{1/2}}\mathbf{U_u^T}$, and the eigndecomposition of $\mathbf{V}$ is $\mathbf{R^TR} = \mathbf{U_v}\mathbf{D_v^{1/2}}\mathbf{D_v^{1/2}}\mathbf{U_v^T}$. This, in turn implies that Equation 3 can be used to generate MVN variates with $\mathbf{L} = \mathbf{U_u}\mathbf{D_u^{1/2}}$ and $\mathbf{R} = \mathbf{D_v^{1/2}}\mathbf{U_v^T}$. This approach provides an efficient and numerically stable method to generate matrix-normal random samples with mean matrix M and covariance structure, $V \otimes U$.

To efficiently sample from a MVN distribution with mean vector m and covariance matrix $V \otimes U$, only a few steps are needed.

1. Express mean vector m into $n \times p$ matrix $\mathbf{M}$.
2. Generate an $n \times p$ matrix Z of standard $(N(0,1))$ normal random variates.

3. Compute **L** and **R** via eigendecomposition (preferred; as it is more stable) or Cholesky decomposition in Equation 3.

In order to generate the condition realizations $(y^*|y_c)$ in Chapter 7 Table 0.1 , we need (1) the eigendecompositions of R_t, R_x and R_{x^*,x^*}, (2) the function `pre_process_svd` to carry out the $\text{vec}(U_t \tilde{Z} U_x^T)$ and (3) the function `cond_mean_mat`to efficiently calculate the *psuedo conditional mean* $(\mathbf{w}^*)$. We begin by performing eigendecompositions of R_t, R_x and R_{x^*,x^*}:

```
# Eigendecomposition of Rt
svdt = svd(Rt)
Ut = svdt$u
Ut_t = t(svdt$v)
St = svdt$d

# Eigendecomposition of Rx
svdx = svd(Rx)
Sx = svdx$d
Ux = svdx$u
Ux_t = t(svdx$v)

# Eigendecomposition of R_{xstar, xstar}
svdR_xstar_xstar = svd(R_xstar_xstar)
Rstst_U = svdR_xstar_xstar$u
Rststd = svdR_xstar_xstar$d
```

Next, we define the `pre_process_svd` function enabling us to compute the realization of y by

$$\mathbf{y} = \text{vec}(U_t \tilde{Z} U_x^T)$$

where $\tilde{Z} = \text{matrix}(\tilde{\mathbf{z}}, \text{nrow} \ = s, \text{ncol} = n_c)$, $\tilde{\mathbf{z}} = \sqrt{\mathbf{s}} \times \mathbf{z}$, and $\mathbf{s} = \text{diag}\left(S_x \otimes S_t\right)$. The function used the eigendecompositions of R_t and R_{x^*,x^*}:

```
pre_process_svd = function(z, s, Ut, St, Rstst_U, Rststd){
  nc_prime = length(Rststd)
  # need to take square roots!!
  eigs = kronecker(sqrt(Rststd), sqrt(St))
  ztilde = eigs*z
  mysim = as.vector(Ut %*% matrix(ztilde, nrow = s, ncol = nc_prime) %*% t(Rstst_U))
}
```

The last step before we generate realizations is defining `get_cond_mean` function to enable us to efficiently calculate the *psuedo conditional mean* (i.e., $\mathbf{w}^*$). Recall $w^* = \sigma^2 \left(R_{x^*x} \otimes R_t\right) \left(\sigma^2 (R_x \otimes R_t) + \nu I\right)^{-1} \tilde{\mathbf{y}}$.

```
get_cond_mean = function(svdt, svdx, R_xstar_x, y_tilde, nu, sig2){

  Ut = svdt$u
  Ut_t = t(svdt$v)
  St = svdt$d

  Sx = svdx$d
  Ux = svdx$u
  Ux_t = t(svdx$v)
```

```r
  # carry out conditional mean calculation for each column of y_tilde,
  # where each col of y_tilde is a single y_tilde as in Table .01
  # If we do it this way, we can vectorize calculation steps as much as
  # possible except for this one for loop.

  lambda = diag(1 / (sig2*kronecker(Sx, St) + nu))

  cond_mean_mat = matrix(nrow = nrow(R_xstar_x)*length(St), ncol = ncol(y_tilde))

  for (i in 1:ncol(y_tilde))
  {
      y_star_samp = y_tilde[, i]

      # reshape to be s * nc
      zreshape = matrix(y_star_samp, nrow = s, ncol = nc)

      # Defining v and reshaping V
      v = lambda %*% as.vector(Ut_t %*% zreshape %*% Ux)
      V = matrix(v, nrow = s, ncol = nc)

      # Evaluate the conditional mean
      cond_mean_samp = sig2 * as.vector(Rt %*% Ut %*% V %*% Ux_t %*% t(R_xstar_x) )
      cond_mean_mat[,i] = cond_mean_samp
  }
  return(cond_mean_mat)
}
```

Given that we have set up the eigndecomposition and the our two functions, we can carry out the steps to generate realizations listed in Chapter 7 Table 0.1. We generate a matrix of N(0,1) random variates (i.e., $\mathbf{z} \sim N(\mathbf{0}, \mathbf{I})$) and collect them into an appropriately sized matrix:

```r
nreals = 100
nc_prime = length(Rststd)

# Generate a matrix of N(0,1)'s
allZs = matrix(rnorm(s*nc_prime*nreals), nrow=s*nc_prime)
```

Next, we set up an incidence matrix to make it easier to extract $\mathbf{u}$.

```r
kmat = matrix(0, nrow = length(train_idx_kron), ncol = s*nc_prime)
kmat[train_idx_kron, train_idx_kron] = diag(1, nrow= length(train_idx_kron))
```

Now, we complete the steps listed out under the conditional realization of $y^*|y_c$ tab in Chapter 7 Table 0.1. Note the `rmultnorm` function is located in `Cov_functions.R`:

1. Generate $\epsilon \sim N(\mathbf{0}, \nu \mathbf{I}_{n_c})$

```r
# Generate epsilon ~ MVN(0, nu)
epsilon = t(rmultnorm(nreals, rep(0, s*nc), diag(nu, nrow = s*nc)))
```

2. Use the `pre_process_svd()` function to get $\mathbf{u}^{\mathbf{F}} = \begin{pmatrix} \mathbf{u} \\ \mathbf{u}^* \end{pmatrix}$ and collect into a matrix

```r
# use the pre_process_svd() function to get u and collect into a matrix
umat = apply(allZs,2, function(x) pre_process_svd(z=x, s=s,
            Ut = Ut, St = St, Rstst_U = Rstst_U, Rststd = Rststd))
```

3. Set $\tilde{y} = \mathbf{u} + \epsilon$ where we get $\mathbf{u}$ by multiplying the defined incidence matrix (`kmat`) by $\mathbf{u}^{\mathbf{F}}$.

```r
# get y_tilde
y_tilde = kmat %*% umat + epsilon
```

4. Set w^* using `get_cond_mean` function.

```r
# get w_star
w_star = get_cond_mean(svdt, svdx, R_xstar_x, y_tilde, nu, sig2)
```

5. Set $\tilde{\mathbf{u}}^* = \mathbf{u}^* - \mathbf{w}^*$

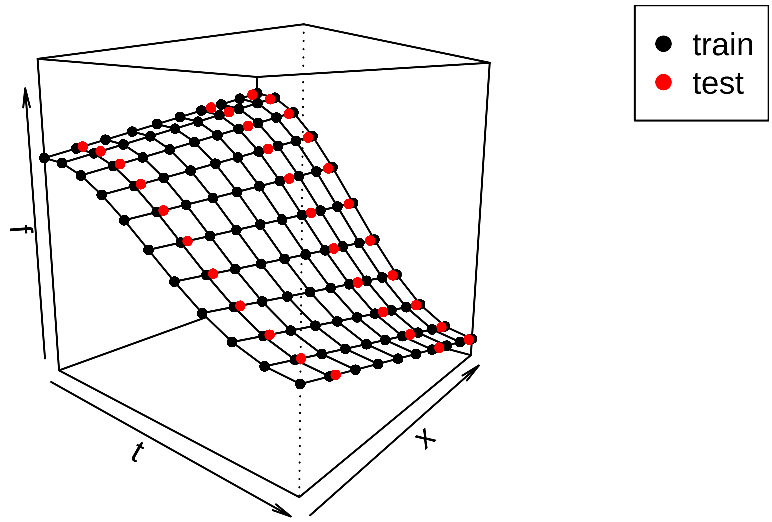
```r
ustar = umat - w_star
```

6. Calculate the realizations, $E(y^*|y_c) + \tilde{\mathbf{u}}^*$

```r
conditional_samps = as.vector(cond_mean) + ustar
```

We plot results as before and inspect the realizations.

```r
persp(t1, x1,
      matrix(cond_mean[train_idx_kron], nrow = s),
      theta = 130-90,
      phi = 10,
      xlab = 't', ylab = 'x', zlab = 'f',
      zlim = c(-2.2, 2.4)) -> res

for (i in 1:nreals){
  # Training Points
  points(trans3d(plot_grid_train[,1],
                 plot_grid_train[,2],
                 conditional_samps[train_idx_kron, i],
                 pmat = res),
         col = 'black',
         pch = 16,
         cex = 0.7)
  # Test Points
  points(trans3d(plot_grid[, 1],
                 plot_grid[, 2],
                 conditional_samps[test_idx_kron, i],
                 pmat = res),
         col = 'red',
         pch = 16,
         cex = 0.7)
}
legend("topright", legend = c("train", "test"), col = c("black", "red"), pch=19)
```


train
test
f
t
x

2. Estimating Covariance Parameters via Output Basis Strategies

In this section, we demonstrate how covariance parameters can be estimated when basis function strategies are used to represent multivariate computer model output. Specifically, we adopt basis functions derived from singular value decomposition (SVD); see Section 0.3.2.1 of the book chapter for details. A common choice for basis functions involves estimating them empirically from the ensemble of model outputs stored in the $s \times n_c$ matrix Y_c using empirical orthogonal functions (EOFs) Ramsay and Silverman (2005); Hannachi et al. (2007). We illustrate this with our simple example shown in Figure 3. The goal of the optimization is to minimize the negative log-likelihood, as defined in Equation (7) of the book chapter. We strongly recommend reviewing the optimization routines provided in `Estimation_Basis.R`.

```
s = 11          # size of output space
nc = 10         # number of computer model runs

# Creates a grid of points where x[,2] represents x and x[,1] represents t
t1 = seq(0,1, length = s)
x1 = seq(0,1, length = nc)
x = expand.grid(t1,x1)

# Input grid points into the deterministic function to be emulated
f = (x[,2]+1)*cos(pi*x[,1]) + 0.03*(exp(x[,2]))
```

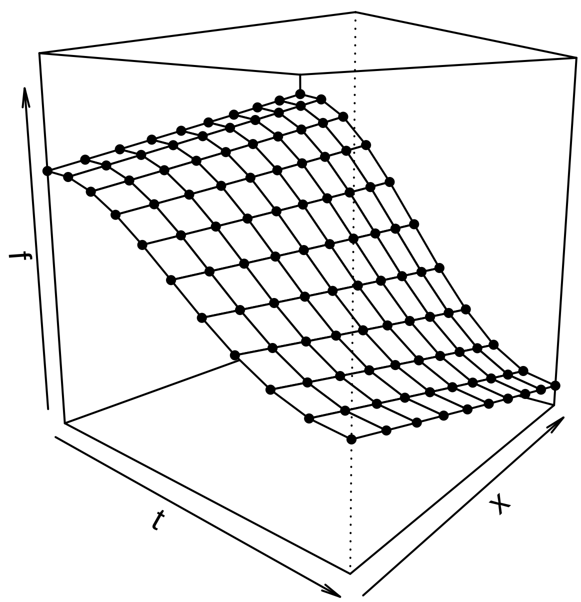


Figure 3: A simple functional model to be emulated

The optimization function `ML_pcEMU()` in `Estimation_Basis.R` only requires three parameters:

1. a vector of computer model output `fn`,
2. computer model inputs `x`,
3. `q`, the number of bases used in representing computer model output.

The function return a list where each element of the list are the ϕ_j's and σ_j^2's corresponding to q_j and the parameter ν is not estimated and fixed at a value of 1e-5.

```
params = ML_pcEMU(fn = f, x = x1, q = 2)
```

The parameter estimates and associated R code to print the values are:

Bases Number	ϕ	σ^2
j = 1	6.7395563	0.7710988
j = 2	7.3470944	0.8953043
R Code	`params[[j]][1]`	`params[[j]][2]`

2.1 Conditional mean and realizations

In the Chapter 7, Section 0.3.1 , we provide the calculation to estimate the predictive mean and covariance matrix of $\mathbf{w}_j^*$ for new input locations `xnew`. We define and show the work for the the predictive distribution to be:

$$\mathbf{w_j^*}|\hat{\mathbf{w}}_\mathbf{j} \sim N(V_{21}V_{11}^{-1}\hat{\mathbf{w}}_\mathbf{j} \quad , \quad V_{22} - V_{21}V_{11}^{-1}V_{12}),$$

where we define V_j as

$$\begin{pmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \end{pmatrix} = \left[\begin{pmatrix} \frac{\nu}{\mathbf{k}_j^T\mathbf{k}} I_{n_c} & 0 \\ 0 & 0 \end{pmatrix} + \sigma_j^2 R\left(\begin{pmatrix} X \\ X^* \end{pmatrix}; \phi_j \right) \right]; j = 1, \dots, q.$$

For more details about the notation and work, references Chapter 7 Section 0.3. The `get_wstar_distr_preds` function returns a list of the predicted mean and covariance matrix of $\mathbf{w}_j^*$ for new input locations `xnew` using the work illustrated.

```
get_wstar_distr_preds = function(f, xnew, nc, q, x1,t1, params)
{
  # f =  function output at train points
  # xnew = vector of new locations
  # nc = number of computer runs
  # q = number of bases
  # x1 = original x train locations
  # params = object you get from the basis estimation function

  nc_new = length(xnew)
  fmat = matrix(f, ncol=nc)
  xdist_aug = as.matrix(dist( c(x1, xnew)) )

  # subtract means of rows
  meanf = apply(fmat, 1, mean)
  fmat0 = fmat - meanf

  # Singular Value Decomposition on
  fsvd = svd(fmat0)

  # Construct the K matrix
  K = fsvd$u[,1:q] %*% diag(fsvd$d[1:q]) / sqrt(nc)

  # Allocate the list to hold predictive means and covariance
  wlist = list()
```

```r
  nu = 1e-6 # Set the nu estimate

  for (i in 1:q)
  {
    # Collect the Estimates of phi and sigma
    phi = params[[i]][1]
    sig2 = params[[i]][2]

    # Defining j basis vector (k_j)
    kj = K[, i]

    # Construct the R covariance matrix R(X, phi)
    Rcov = sig2*exp(-(phi*xdist_aug)^2)
    # Estimate the nugget
    nug = nu / sum(kj^2)

    # get the parts of V matrix
    sigma11 = nug * diag(1, nrow = nc, ncol = nc) +  Rcov[1:nc, 1:nc]
    sigma12 = Rcov[1:nc,(nc+1):(nc+nc_new) ]
    sigma22 = Rcov[(nc+1):(nc+nc_new), (nc+1):(nc+nc_new) ]
    sigma21 = t(sigma12)

    # Inverse of V_11
    sig11_inv = solve(sigma11)

    # Collect w_hat based on j
    w_hat_j = fsvd$v[ ,i] * sqrt(nc)

    # Calculate the predictive mean
    wmean = sigma21 %*% sig11_inv %*% w_hat_j

    # Calculate the covariance matri
    wcov = sigma22 - sigma21 %*%  sig11_inv %*% sigma12

    # Store the values
    sublist = list(wmean = wmean, wcov = wcov)
    wlist[[i]] = sublist
  }
  return(list(wlist = wlist, K = K, meanf = meanf, fmat0 = fmat0))
}
```

Using our parameter estimates of σ^2 and ϕ along with the `get_wstar_distr_preds()` function to generate predictions. Note that the function also returns the mean and covariance for w^*, which can be used to generate realizations.

```r
# Calculate the predictive mean and covariance matrix of w_j^*
w_obj = get_wstar_distr_preds(f=f,xnew = xnew, nc=nc,q=2, x1=x1, t1=t1, params=params)
```

After using the function, we can obtain various values such as the predictive means for each w_j, K matrix, the mean of our function,f.

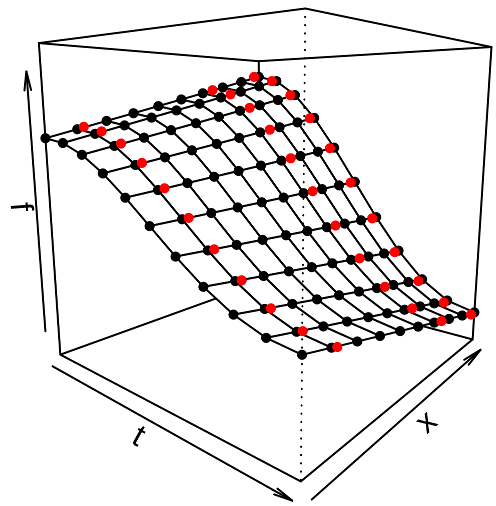
```r
# predictive mean for w_1 and w_2
w1 = as.vector(w_obj$wlist[[1]]$wmean)
w2 = as.vector(w_obj$wlist[[2]]$wmean)

# Obtain K matrix
K = w_obj$K

# mean of f (f is demeaned prior to parameter estimation)
meanf = w_obj$meanf
```

Using the obtained values from `get_wstar_distr_preds()` function, we calculate our predictions using the following code.

```r
# make w's into a q x nc matrix
w_all = rbind(w1, w2)

# get predictions, add mean back
basis_preds = K %*% w_all
basis_preds = as.vector(basis_preds + meanf)
```

Let's plot the predictions (not realizations) to see how they look.

```r
persp(t1, x1,
      matrix(f, nrow = s),
      theta = 130-90,
      phi = 10,
      xlab = 't', ylab = 'x', zlab = 'f',
      zlim = c(-2.4, 2.4)) -> res

# Train
points(trans3d(x[, 1], x[, 2],
               f,
               pmat = res),
       col = 'black',pch = 16,cex = 0.7)

# Test
points(trans3d(plot_grid[, 1],
                 plot_grid[, 2],
                 basis_preds,
                 pmat = res),
         col = 'red',pch = 16, cex = 0.7)
```



As previously mentioned, we can also generate realizations by using the returned the predictive mean and covariance for w_j^*. The following code illustrates how to preform generate the realizations.

```
# get covariance matrices for w's
w1cov = w_obj$wlist[[1]]$wcov
w2cov = w_obj$wlist[[2]]$wcov

# Generate the realizations
w1reals = rmultnorm(200, mu = w1, sigma = w1cov)
w2reals = rmultnorm(200, mu = w2, sigma = w2cov)

reals_matrix = matrix(nrow = 200, ncol = length(basis_preds))
for (i in 1:200)
{
  w_all = rbind(w1reals[i,], w2reals[i, ])
  bpreds = as.vector(K %*% w_all + meanf)
  reals_matrix[i, ] = bpreds
}
```

References

Hannachi, Abdel, Ian T Jolliffe, David B Stephenson, et al. 2007. “Empirical Orthogonal Functions and Related Techniques in Atmospheric Science: A Review.” *International Journal of Climatology* 27 (9): 1119–52.

Nychka, Douglas, Christopher Wikle, and J Andrew Royle. 2002. “Multiresolution Models for Nonstationary Spatial Covariance Functions.” *Statistical Modelling* 2 (4): 315–31.

Ramsay, James O, and Bernard W Silverman. 2005. “Principal Components Analysis for Functional Data.” *Functional Data Analysis*, 147–72.

Stegle, Oliver, Christoph Lippert, Joris M Mooij, Neil Lawrence, and Karsten Borgwardt. 2011. “Efficient Inference in Matrix-Variate Gaussian Models with\iid Observation Noise.” *Advances in Neural Information Processing Systems* 24.