CODE COMPENDIUM: MULTIVARIATE AND FUNCTIONAL OUTPUT EMULATION

August 17, 2023

Maike Holthuijzen and Dave M. Higdon
Virginia Tech
maikeh@vt.edu, dhigdon@vt.edu

Contents

1 A SIMPLE EXAMPLE

First, we will set up the simple example from the chapter.

```
source("Cov_functions.R")
source("Estimation_1D.R")
source("Estimation_Basis.R")
# make the 2-d x support
s = 11; nc = 10

t1 = seq(0,1,length = s)
x1 = seq(0,1,length = nc)
x = expand.grid(t1, x1)

# a deterministic function to be emulated
f = (x[,2]+1)*cos(pi*x[,1]) + .03*(exp(x[, 2]))
```

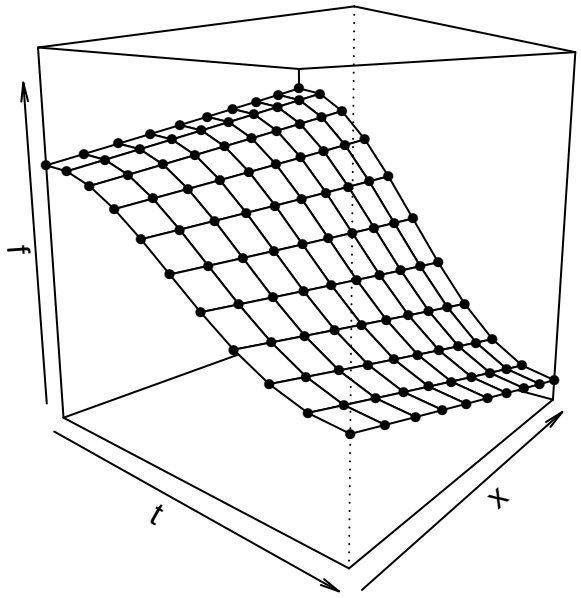


Figure 1: A simple functional model to be emulated

We are assuming that our function f can be modeled as a 0-mean GP where the covariance matrix is the Kronecker product of space and time covariance matrices. Recall that we can model $\text{vec}(Y) = f$ using a

Kronecker representation for the correlation

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

1.1 Estimating parameters

We assume that the prior for f is

$$\sim N(\mathbf{0}, \sigma^2(R_x \otimes R_t) + \nu I),$$

and we wish to obtain the posterior distribution, $f|D_n$, where D_n is the data. Using functions in `Estimation_1D.R`, we can estimate parameters ϕ_x, ϕ_t, σ^2, and ν.

Recall the likelihood function derived from the matrix normal distribution (with nugget) Stegle et al. [2011]:

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

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, Kronecker products need not be computed for evaluation of the likelihood. Note that $(S_x \otimes S_t + \nu I)^{-1} = \mathbf{v^{-1}}$ is a diagonal matrix, so its inverse as well as the log determinant can be easily computed: $\log|\sigma^2 R_x(\boldsymbol{\phi}) \otimes R_t(\boldsymbol{\phi}) + \nu I| = \mathbf{1}^\top \log(\mathbf{v})$. The other quantity of interest, the sum of squares, can also be efficiently calculated. All calculations are given in Table 0.1 in the chapter.

We can obtain estimates for covariance parameters for R_x and R_t via maximum likelihood. ML estimates for marginal variance σ^2 and lengthscale ϕ are not consistent; thus, the accuracy of estimates will not increase with the size of the data.. First, we compute the distance matrices for inputs x and t:

```
Tdist = get_distmat(t1, t1)
Xdist = get_distmat(x1, x1)
```

Now we can estimate parameters using the functions in `Estimation_1D.R`. We use the R package `dfoptim` and the function `nmkb`, which implements the Nelder-Mead algorithm for derivative-free optimization. The function `Estimate_params()` takes as arguments `distC` and `distR`, the first and second distance matrices in the Kronecker product (order matters) and `z`, computer model output to be emulated. The parameters are output in the following order:
scale estimate for the first covariance matrix,
scale estimate for the second covariance matrix,
nugget,
marginal variance.
Sensible upper and lower bounds for the optimization procedure are also specified below:

```
# this 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))
# scale for Rx
print(params$par[1])

## [1] 12.15175

# scale for Rt
print(params$par[2])

## [1] 1.126895

# nugget nu
print(params$par[3])

## [1] 1e-06
```

```
#  marginal variance: sigma2
print(params$par[4])
```

```
## [1] 100
```

The parameter estimates are: $\phi_x = 12.1517495$, $\phi_r = 1.1268946$, $\nu = 10^{-6}$, $\sigma^2 = 99.9999963$.

The following psuedocode shows how the log determinant and sum of squares are efficiently computed and are used in functions for parameter estimation in `Estimation_1D.R` (all calculations can be found in table 0.1 in the chapter).

```
nu = params$par[3]
svdx = svd(Rx)
svdt = svd(Rt)

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

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

Fmat = matrix(f, nrow = nc, ncol = s)

# these calculations are in table 0.1
Yvec = as.vector(t(Ur) %*% Fmat %*% Ux)

v = sigma2 * kronecker(Sx, St) + nu
lambda = 1/v

ssqKronecker = (Yvec)^2 %*% lambda

# log determinant calculation
LogDet = sum(log(lambda))
```

1.2 Computing the conditional mean and covariance

In this section, we use the parameter estimates from the previous section 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.

First, we build R_x and R_t using estimates for ϕ_x and ϕ_t. Note that we assume Gaussian covariance functions throughout.

```
scaleX = params$par[1]
scalet = params$par[2]

Rt = exp( -(Tdist / scalet)^2 )
Rx = exp( -(Xdist / scaleX)^2 )
```

Next, we set up a prediction grid for new x locations (x_{new}). We also construct an augmented x input space, where $x^* = [x, x_{new}]$ and construct the cross covariance matrix $R_{x^*,x}$. Next, we perform an eigendomposition of R_x, R_t and $R_{x^*,x}$ which is necessary for efficient computation of the conditional mean (see book chapter). Note that for this example, we only vary x, not t.

```
s = 11

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

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

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

# build cross covariance matrix, R_x*,x
dist_xstar_x = get_distmat(xstar_grid, x1)

R_xstar_x = exp( -(dist_xstar_x / scaleX)^2 )
```

For the calculation of the conditional mean, we perform an eigendecomposition of R_x and R_t

```
svdx = svd(Rx)
svdt = svd(Rt)

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

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

Now, we can efficiently obtain the conditional mean using Kronecker product properties (see Table 0.1 in the book chapter).

```
nu = params$par[3]
sig2 = params$par[4]

# calculate kronecker product of singular values Sx and St
lambda = diag(1 / (sig2*kronecker(Sx, St) + nu))

# reshape the response to be s * nc
f_reshape = matrix(f, nrow = s, ncol = nc)

res1 = lambda %*% as.vector(Ut_t %*% f_reshape %*% Ux)

resmat = matrix(res1, nrow = s, ncol = nc)

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

Now we can plot the conditional mean. In the code below, we only plot at test locations (x_{new}).

```
# grid for plotting
# train locations in black; test locations in red

plot_grid_train = expand.grid(t1, x1)
```

```
plot_grid = expand.grid(t1, xnew)

# this is 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)
```

Now we can inspect the predicted mean on the grid of training points.

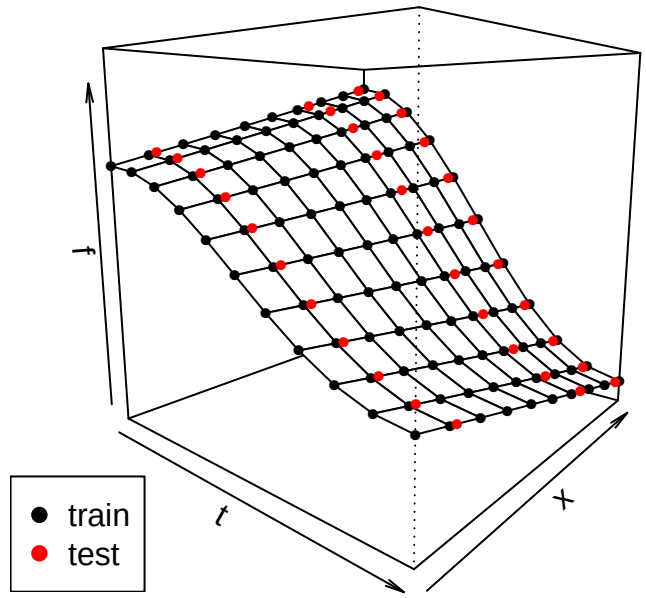


Figure 2: Conditional mean predictions (red) on the grid of training points (black)

1.2.1 Generating conditional realizations

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 et al., 2002] to generate realizations from the conditional predictive distribution.

Before we delve into details, it is instructive to recall the matrix-normal distribution. The matrix-normal distribution (with error) Stegle et al. [2011] is related to the multivariate normal distribution as follows:

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

if and only if

$$\text{vec}(\mathbf{X}) \sim \mathcal{MVN}_{np}(\text{vec}(\mathbf{M}), \mathbf{V} \otimes \mathbf{U}). \tag{1}$$

The probability distribution function (PDF) of the matrix-normal distribution is given by:

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

Suppose $\mathbf{Y}$ has a matrix normal distribution with mean matrix $Z = \mathbf{0}_{n\times p}$ and covariance matrices $\mathbf{U}$ and $\mathbf{V}$. Then for a mean matrix $\mathbf{M}$ and linear transformations $\mathbf{L}$ and $\mathbf{R}$, $\mathbf{Y}$ can be expressed as:

$$\mathbf{Y} = \mathbf{M} + \mathbf{LZR}, \tag{2}$$

where $\mathbf{Y}$ has a matrix-normal distribution with parameters $\mathbf{M}, \mathbf{LL^T}, \mathbf{R^TR}$. In (2), Cholesky decomposition is used to construct the matrices $\mathbf{L}$ and $\mathbf{R}$, the Cholesky factors of $\mathbf{U}$ and $\mathbf{V}$, respectively. If an eigendecomposition is used, then $\mathbf{LL^T} = \mathbf{U_u D_u^{1/2} D_u^{1/2} U_u^T}$ is the eigendecomposition of $\mathbf{U}$ and $\mathbf{R^TR} = \mathbf{U_v D_v^{1/2} D_v^{1/2} U_v^T}$ is the eigendecomposition of $\mathbf{V}$. This, in turn implies that (2) can be used to generate MVN variates with $\mathbf{L} = \mathbf{U_u D_u^{1/2}}$ and $\mathbf{R} = \mathbf{D_v^{1/2} U_v^T}$

To efficiently sample from a MVN distribution with mean vector m and covariance matrix $V \otimes U$, only a few steps are needed. We begin by expressing m as a matrix $\mathbf{M}$ of size $n \times p$. The matrix of $N(0,1)$ random variates $\mathbf{Z}$ must also be of size $n \times p$. We use either Cholesky or an eigendecomposition to obtain $\mathbf{L}$ and $\mathbf{R}$ in (2). We prefer the eigendecomposition, as it is more stable.

To carry out the generation of realizations, we begin by performing eigendecompositions of R_t, R_x and R_{x^*,x^*}:

```
svdR_xstar_xstar = svd(R_xstar_xstar)
Rstst_U = svdR_xstar_xstar$u
Rststd = svdR_xstar_xstar$d

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

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

The function below carries out the calculation $(U_t \tilde{Z} U_x^T)$ from table 0.1.

```
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))
}
```

Now we can carry out steps to generate realizations (steps are listed in Table 0.1 in the chapter). We generate a matrix of N(0,1) random variates and collect them into an appropriately sized matrix:

```
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)
```

Here we set up an incidence matrix to make it easier to extract **u**.

```
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))
```

The function below will enable us to efficiently calculate the psuedo conditional mean ($\mathbf{w}^*$ from Table 0.1).

```
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)

  # 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)

  res1 = lambda %*% as.vector(Ut_t %*% zreshape %*% Ux)

  resmat = matrix(res1, nrow = s, ncol = nc)

  cond_mean_samp = sig2 * as.vector(Rt %*% Ut %*% resmat %*% Ux_t %*% t(R_xstar_x) )
  cond_mean_mat[,i] = cond_mean_samp
  }
  return(cond_mean_mat)
}
```

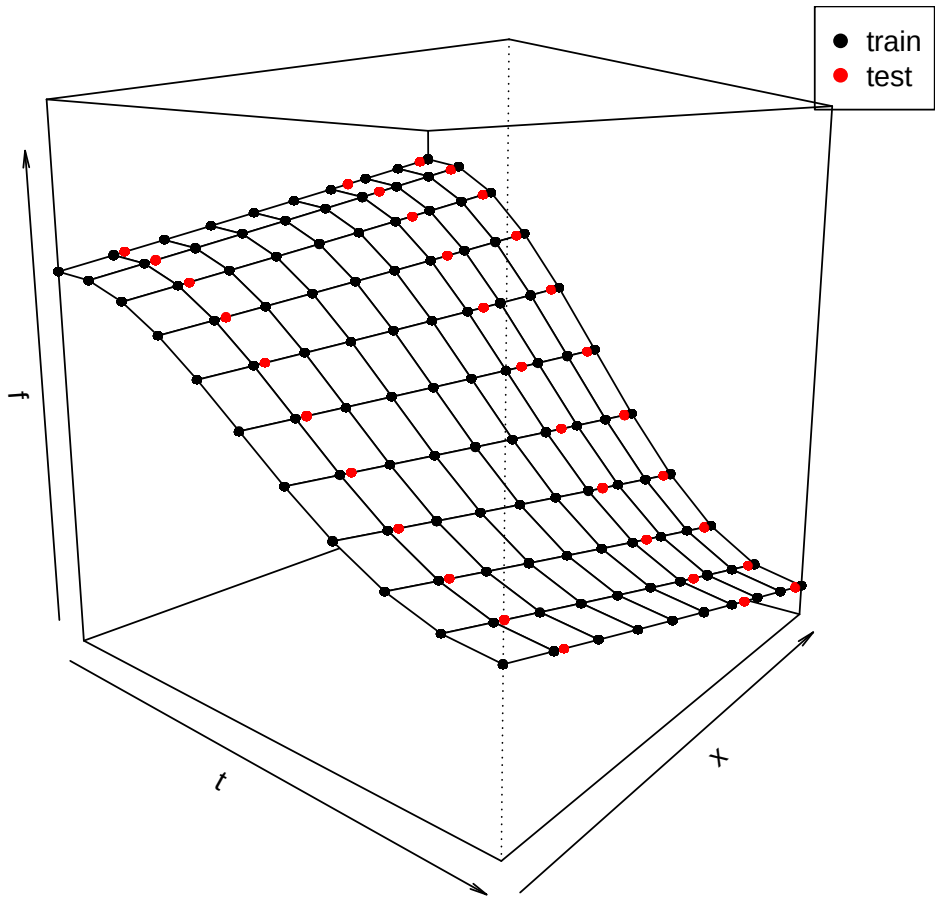
Here are all the steps (the function `rmultnorm` is located in `Cov_functions.R`):

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

# 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))
```

```
# get y_tilde
y_tilde = kmat %*% umat + epsilon
# get w_star
w_star = get_cond_mean(svdt, svdx, R_xstar_x, y_tilde, nu, sig2)
ustar = umat - w_star
# we calculated cond_mean previously
conditional_samps = as.vector(cond_mean) + ustar
```

We can plot results as before and inspect the realizations.

```
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){
  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)

  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)
```



2 PARAMETER ESTIMATION: OUTPUT BASIS STRATEGIES? IDK WHAT TO CALL THIS

In this section we show how covariance parameters can be estimated for the case when basis strategies is used for handling multivariate computer model output. Here, we choose to use bases based on singular value decomposition (see section 0.3.2.1 in the book chapter).

A common choice of basis functions are estimated empirically from the existing ensemble of model outputs held in the $s \times n_c$ matrix Y_c using empirical orthogonal functions (EOFs) Ramsay and Silverman [2005]; Hannachi et al. [2007]. We use the simple example shown in Figure 2. The goal of the optimization is to minimize the negative of the (log) likelihood shown in (7) in the book chapter. We highly recommend reviewing the optimization functions in `Estimation_Basis.R`.

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

t1 = seq(0,1,length = s)
x1 = seq(0,1,length = nc)
x = expand.grid(t1, x1)

# function to emulate
f = (x[, 2]+1)*cos(pi*x[, 1]) + .03*(exp(x[, 2]))
```

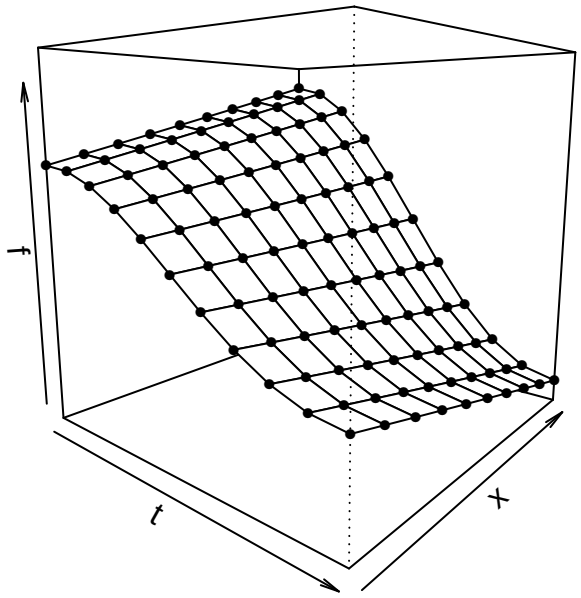


Figure 3: A simple functional model to be emulated

The optimization function `ML_pcEMU()` only requires three parameters: a vector of computer model output `fn`, computer model inputs `x`, and `q`, the number of bases used in representing computer model output. The parameter v is not estimated here and is fixed at a value of 1e-5.

```
# carry out optimization
params = ML_pcEMU(fn = f, x = x1, q = 2)
print(params)
```

```
## [[1]]
##       phi   sigma^2
```

```
## 6.7395563 0.7710988
##
## [[2]]
##       phi   sigma^2
## 7.3470944 0.8953043
```

2.1 Conditional mean and realizations

Here is a function that returns the predictive mean and covariance matrix of $\mathbf{w}_j^*$ for new input locations `xnew`.

```
# 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
get_wstar_distr_preds = function(f, xnew, nc, q, x1,t1, params){

  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

  # do svd
  fsvd = svd(fmat0)

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

  s = nrow(K)
  wlist = list()

  preds = rep(0, length(t1)*nc_new)
  for (i in 1:q){
    phi = params[[i]][1]
    sig2 = params[[i]][2]

    nu = 1e-6

    kj = K[, i]

    Rcov = sig2*exp(-(phi*xdist_aug)^2)
    nug = nu / sum(kj^2)

    # get the parts of V
    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)

    sig11_inv = solve(sigma11)

    what_j = fsvd$v[ ,i] * sqrt(nc)

    wmean = sigma21 %*% sig11_inv %*% what_j

    wcov = sigma22 - sigma21 %*%  sig11_inv %*% sigma12
    sublist = list(wmean = wmean, wcov = wcov)
    wlist[[i]] = sublist

  }

  return(list(wlist = wlist, K = K, meanf = meanf, fmat0 = fmat0))

}
```

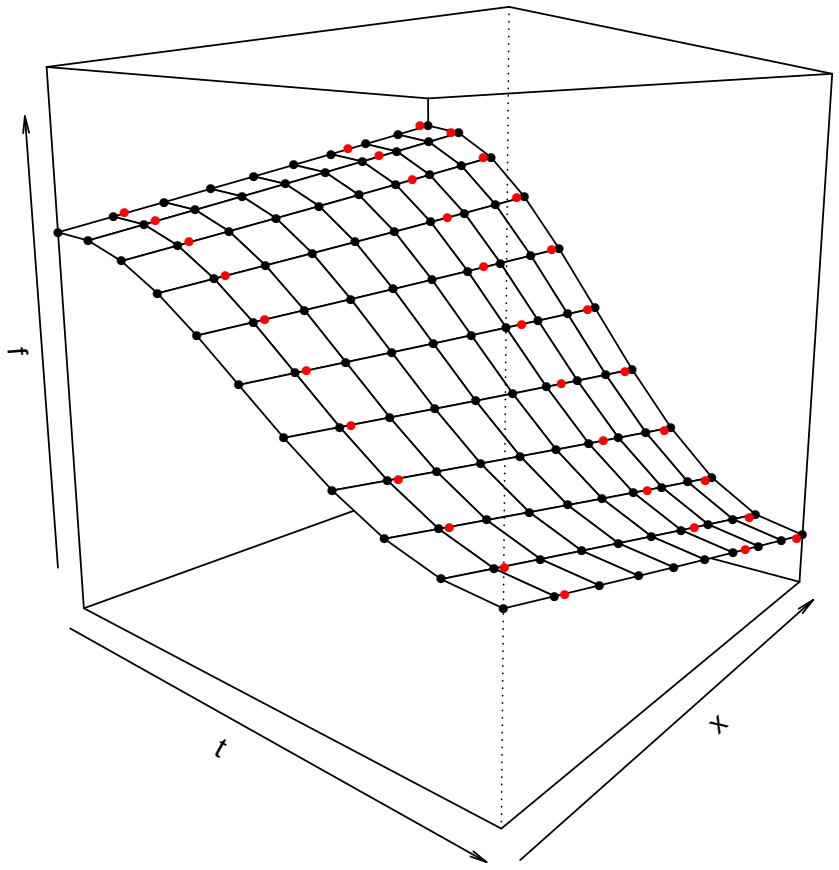
Now, we can use our previous results to generate predictions using the function `get_wstar_distr_preds()`. Note that the function also returns the mean and covariance for w^*, which can be used to generate realizations.

```
w_stuff = get_wstar_distr_preds(f=f,xnew = xnew, nc=nc,q=2, x1=x1, t1=t1, params=params)
# get predictive mean of w_1 and w_2
w1 = as.vector( w_stuff$wlist[[1]]$wmean)
w2 = as.vector( w_stuff$wlist[[2]]$wmean)

# grab the K matrix
K = w_stuff$K

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

# 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 just the predictions (not realizations) to see if they look ok.

```
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

points(trans3d(x[, 1], x[, 2],
               f,
               pmat = res),
```

```
       col = 'black',
       pch = 16,
       cex = 0.7)

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



REFERENCES

Oliver Stegle, Christoph Lippert, Joris M Mooij, ND Larence, and Karsten Borgwardt. Efficient inference in matrix-variate gaussian models with iid observation noise. 2011.

Douglas Nychka, Christopher Wikle, and J Andrew Royle. Multiresolution models for nonstationary spatial covariance functions. *Statistical Modelling*, 2(4):315–331, 2002.

JO Ramsay and BW Silverman. Principal components analysis for functional data. *Functional data analysis*, pages 147–172, 2005.

Abdel Hannachi, Ian T Jolliffe, and David B Stephenson. Empirical orthogonal functions and related techniques in atmospheric science: A review. *International Journal of Climatology: A Journal of the Royal Meteorological Society*, 27(9):1119–1152, 2007.