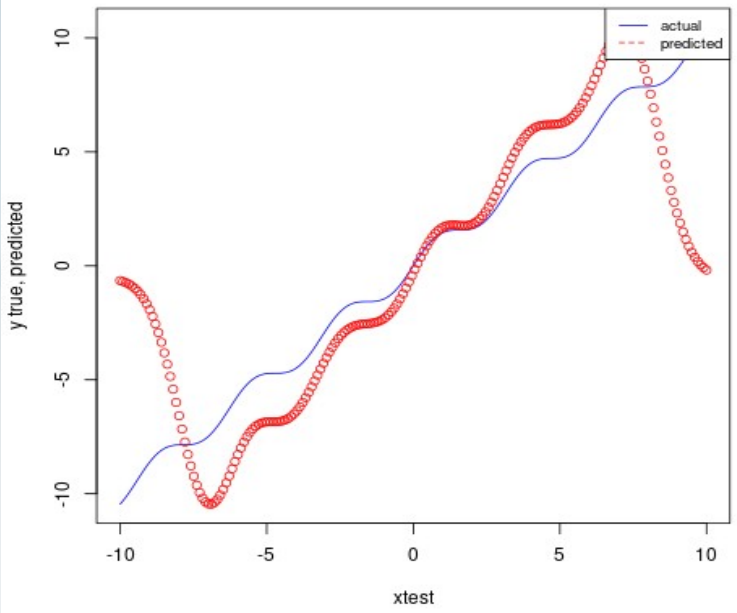
Overview of Gaussian Process Regression

- **Aim:** Model an unknown function, (e.g. *y* can be C_L and *x* can be α) over an interval.
- We have access to noisy evaluations of this function:



- On the left, the uniform distribution is used for
  - = *sin(x) * cos(x) + x* (blue curve)
- GPR mean prediction (red curve)

single point conditional

$$p(f_n|\bar{\mathbf{f}}) = \mathcal{N}(\mu_n, \lambda_n)$$

$$\mu_n = \mathbf{K}_{nM}\mathbf{K}_M^{-1}\bar{\mathbf{f}}$$

$$\lambda_n = K_{nn} - \mathbf{K}_{nM}\mathbf{K}_M^{-1}\mathbf{K}_{Mn}$$

Both mean and variance information, GP mean deviates from true value outside training interval.

Role of Covariance Function

- The regularized covariance measures similarity between data points:
- *(x: training data point,x': test data point)*

- Kernels:

Gaussian

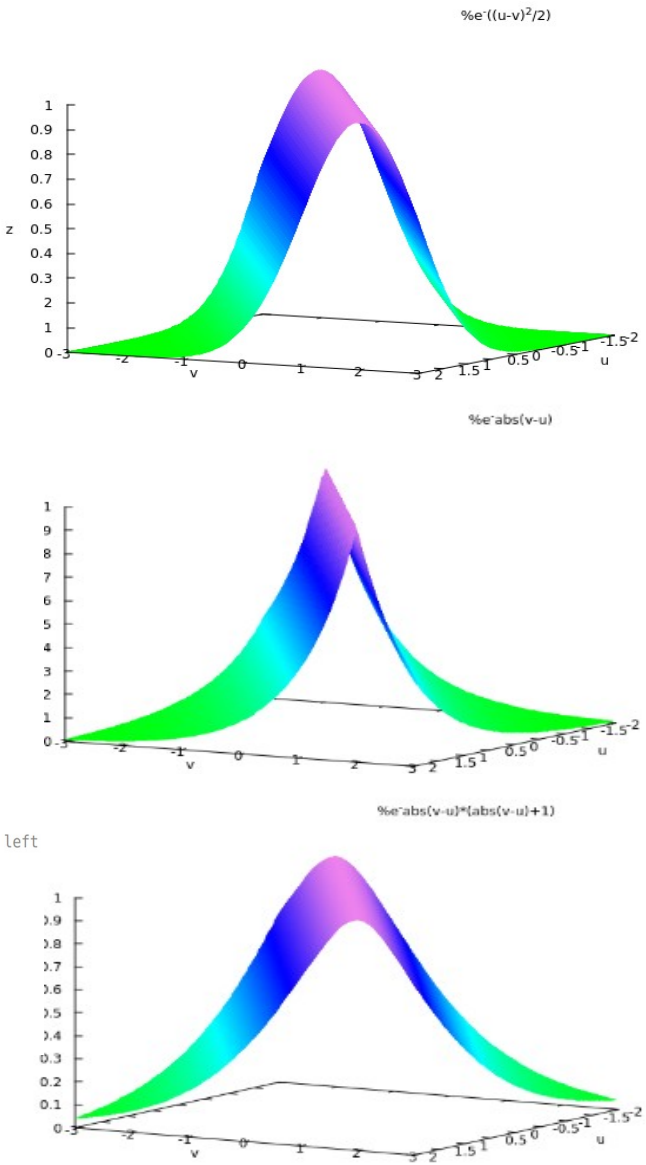
Exponential

Matern

Custom

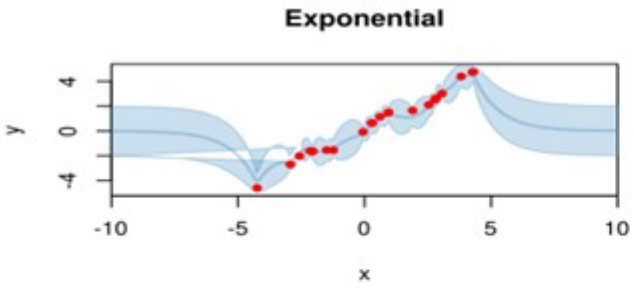
Choice depends on data distribution.

```r
gp_solve <- function(x_train, y_train, x_pred, kernel, sigma2e = 0, a, b, c) {

  # Compute the covariance matrix for training data
  K_train_train <- kernel(x_train, x_train, a, b, c)

  # Add \lambda I regularizer for numerical stability
  K_train_train <- K_train_train + max(sigma2e, 1e-6) * diag(nrow(K_train_train))

  # Compute covariance between training and test data
  K_train_pred <- kernel(x_train, x_pred, a, b, c)

  # Compute covariance between test points
  K_pred_pred <- kernel(x_pred, x_pred, a, b, c)

  # Ensure correct size of the diagonal for jitter in K_pred_pred
  if (nrow(K_pred_pred) == ncol(K_pred_pred)) {
    K_pred_pred <- K_pred_pred + jitter * diag(nrow(K_pred_pred))
  }

  # Solve for inverse using Cholesky factorization
  L <- chol(K_train_train)
  L_inv <- solve(L)

  # Calculate the inverse of K_train_train using Cholesky factorization
  K_train_inv <- t(L_inv) %*% L_inv

  # Compute the predictive mean: K_train_pred %*% (K_train_train_inv %*% y_train)
  # Compute (K_train_inv %*% y_train) first, then multiply by K_train_pred from the left
  alpha <- K_train_inv %*% y_train
  mu_pred <- t(K_train_pred) %*% alpha

  # Compute the predictive variance
  cov_pred <- K_pred_pred - t(K_train_pred) %*% K_train_inv %*% K_train_pred

  # Return the predicted mean and variance
  solution <- list(mu = mu_pred, var = cov_pred)
  return(solution)
}
```

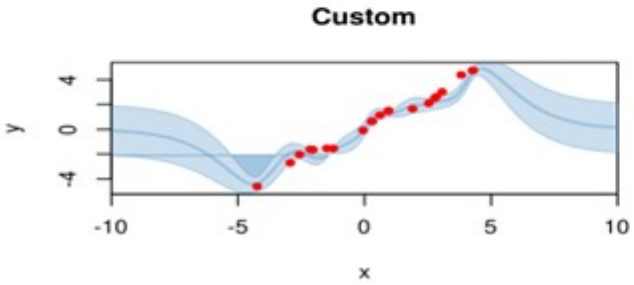


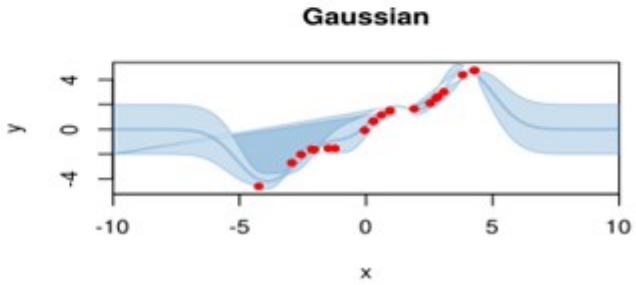
GP Kernel Candidates

Results differ with different kernels on synthetic example:




```r
# matern kernel
kfunc_matern <- function(x1, x2, a, b, c) {
  Sigma <- matrix(0, nrow=length(x1), ncol=length(x2))
  for (i in 1:nrow(Sigma)) {
    for (j in 1:ncol(Sigma)) {
      dist <- abs(x1[i] - x2[j])
      Sigma[i, j] <- a * (1 + b * dist) * exp(-c * dist)
    }
  }
  return(Sigma)
}
```





In case of non-Gaussian noise, can take sum kernel , containing samples of assumed underlying noise distribution [A].

HOW DO WE CHOOSE THE OPTIMAL KERNEL AND ITS PARAMETERS (a,b,c,d)?

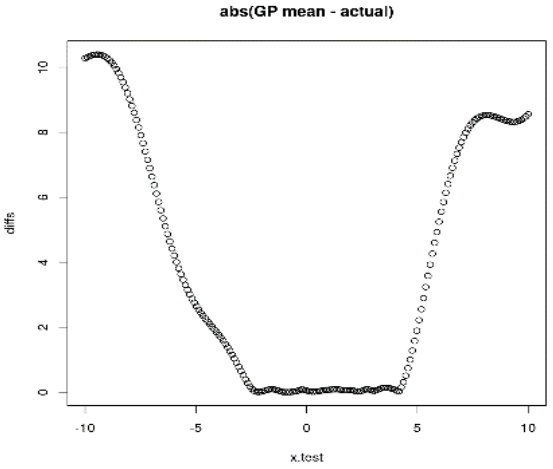
[A] Murray-Smith, Roderick, and Agathe Girard. "Gaussian Process priors with ARMA noise models." In *Irish Signals and Systems Conference, Maynooth*, pp. 147-152. 2001.

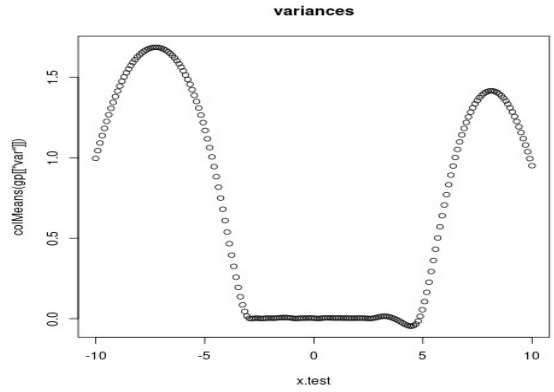
Comparison Metrics for Kernels

Metrics to compare kernel performance on a given training and test data set

- Sum of error in the range of training set values:
  - (||||)
- Sum of variance in the range of training set values:
  - ()
- Log-likelihood function (standard choice)

With and , the noisy training data.





Metrics 1 and 2 are plotted to show how they increase outside the range of training data supplied – as desired

Setting parameters

Set up function, train/test values, and noise settings.

```
# define the function to use and write split train/test values
source('gp_util.R');

ntrain = 300;
ntest = 80;
sigma2e = 1e-3

# training data
printf("setup train data and function\n");
x.train = matrix( rtruncnorm(n=ntrain, a=-10, b=10, mean=0, sd=5) ,ncol=1)
x.test = matrix( rtruncnorm(n=ntest, a=-10, b=10, mean=0, sd=5) ,ncol=1)
func_str = 'sin(v) - cos(v)*sin(v) + v';

x = x.train;
v = x;
y.train = eval(parse(text = func_str));
v = x.test;
y.test = eval(parse(text = func_str));
y.noisy = y.train + runif(length(x.train),min = -sqrt(sigma2e), max = sqrt(sigma2e)); # add noise, not necessarily Gaussian

# split train data into test/train points to test different kernels
printf("saving function and data\n");
x_ker.train = x[1:round(ntrain/3)]
x_ker.test = x[(round(ntrain/3)+1):ntrain];
fp<-file("data/function.txt",'w')
writeLines(func_str, fp)
close(fp)
write.table(x.train,'data/x.train',sep=",",col.names=FALSE,row.names=FALSE);
write.table(x.test,'data/x.test',sep=",",col.names=FALSE,row.names=FALSE);
write.table(x_ker.train,'data/x_ker.train',sep=",",col.names=FALSE,row.names=FALSE);
write.table(x_ker.test,'data/x_ker.test',sep=",",col.names=FALSE,row.names=FALSE);
write.table(y.noisy,'data/y.train_noisy',sep=",",col.names=FALSE,row.names=FALSE);
write.table(y.train,'data/y.train',sep=",",col.names=FALSE,row.names=FALSE);
write.table(y.test,'data/y.test',sep=",",col.names=FALSE,row.names=FALSE);
```

Optimizing the kernel

```
for(nt in 1:ntrials){

printf("trial %d of %d\n", nt, ntrials);
v = x.train;
y = eval(parse(text = func_str));
y.noisy = y + runif(length(x.train),0,sqrt(sigma2e))
v = x.test;
ytest_true = eval(parse(text = func_str));
svals1 = c();svals2 = c();svals3 = c();loglikevals = c();

for ( k in 1:length(kernels) ) {

printf("test kernel %d\n", k);
gp = gp_solve( x.train , y.noisy , x.test , kernels[[k]] , sigma2e, a, b, c, d )

# compute var sum
var_sums = colMeans(gp[['var']]);
sval1 = sum(var_sums);
print("sum of variances 1:");
sval1 = sum(var_sums);
print(sval1);
svals1 = c(svals1,sval1);

inds = which(x.test>-5 & x.test<5)
var_sums2 = abs(var_sums[inds]);
print("sum of variances 2:");
sval2 = sum(var_sums2)
print(sval2)
svals2 = c(svals2,sval2);

diffs = gp[['mu']] - ytest_true;
diffs = abs(diffs[inds]);
sval3 = sum(diffs);
svals3 = c(svals3,sval3);

loglikeval = get_log_likelihood( x.train , y.noisy , x.test , kernels[[1]] , sigma2e = 0, a, b, c, d );
loglikevals = c(loglikevals,loglikeval);

bestknum = which.max(loglikevals);
bestknums = c(bestknums,bestknum);

printf("per loglikelihood, best kernel is %s\n", kernel.names[which.max(loglikevals)]);
# kernel loop
}
```

Subdivide the training set into several train / test splits for each trial ; generate GP models with different kernels and record performance metrics.

The Mode (best kernel per the log-likelihood metric) is then selected out of all trials.

Optimizing the parameters of chosen kernel

We can define a function (based on some performance metric, e.g. log-likelihood):

$$\mathcal{L} = -\log p(\mathbf{y}|\boldsymbol{\theta}) = \frac{1}{2}\log\det\mathbf{C}(\boldsymbol{\theta}) + \frac{1}{2}\mathbf{y}^\top\mathbf{C}^{-1}(\boldsymbol{\theta})\mathbf{y} + \frac{N}{2}\log(2\pi)$$

where $\mathbf{C} = \mathbf{K} + \sigma^2\mathbf{I}$

Then e.g. gradient descent scheme gives:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma_n \nabla F(\mathbf{x}_n),\ n \geq 0.$$

For this, we have to estimate the gradient. We can also use a gradient free scheme (e.g. randomized coordinate descent). At each iteration:

- Pick at random parameter p to optimize (a,b,c, or d).
- Fix other parameters to previously determined values.
- For chosen parameter, loop over range of values, form GP model and evaluate **(x)**.
- Determine value of p that optimizes **(x)**, update parameter value.

Works well, when there are few parameters and the gradient cannot be estimated easily to suitable accuracy.

Optimizing the parameters

```
for ( k in 1:ncdescent_iters ) {

# pick random num in 1-4 range
coord_to_opt = round(runif(1, 1, 4));

printf(">>>>>>>>>> iter = %d -> coord_to_opt = %d\n", k, coord_to_opt);
svals1 = c();
svals2 = c();
svals3 = c();
loglikevals = c();

for ( vind in 1:length(as) ) {

if (coord_to_opt == 1){
    a = as[vind]; b = bsave; c = csave; d = dsave;
} else if (coord_to_opt == 2){
    a = asave; b = bs[vind]; c = csave; d = dsave;
} else if (coord_to_opt == 3){
    a = asave; b = bsave; c = cs[vind]; d = dsave;
} else if (coord_to_opt == 4){
    a = asave; b = bsave; c = csave; d = ds[vind];
}

printf("test kernel %d\n", k);
gp = gp_solve( x.train , y.noisy , x.test , kernels[[opt_kernel]] , sigma2e, a, b, c, d )

# compute var sum
var_sums = colMeans(gp[['var']]);
sval1 = sum(var_sums);
print("sum of variances 1:");
sval1 = sum(var_sums); print(sval1);
svals1 = c(svals1,sval1);

inds = which(x.test>-5 & x.test<5)
var_sums2 = abs(var_sums[inds]);
print("sum of variances 2:");
sval2 = sum(var_sums2); print(sval2)
svals2 = c(svals2,sval2);

diffs = gp[['mu']] - ytest_true;
diffs = abs(diffs[inds]);
sval3 = sum(diffs);
svals3 = c(svals3,sval3);

loglikeval = get_log_likelihood( x.train , y.noisy , x.test , kernels[[opt_kernel]] , sigma2e = 0, a, b, c, d );
loglikevals = c(loglikevals,loglikeval);
}

if (coord_to_opt == 1){
    asave = as[which.max(loglikevals)]; b = bsave; c = csave; d = dsave;
} else if (coord_to_opt == 2){
    a = asave; bsave = bs[which.max(loglikevals)]; c = csave; d = dsave;
} else if (coord_to_opt == 3){
    a = asave; b = bsave; csave = cs[which.max(loglikevals)]; d = dsave;
} else if (coord_to_opt == 4){
    a = asave; b = bsave; c = csave; dsave = ds[which.max(loglikevals)];
}

}
```

In each iteration, a random parameter is optimized.

R code for implementing randomized coordinate descent.

Combining multiple fidelities

Instead of running optimization loop for many iterations, what if we run a few times with different cost expenditures and combine the results?

Given two models , that both estimate the same quantity of interest, the approach [B] determines constants for the linear combination model : . This can then be extended to an arbitrary number of models.

$$\mathbf{k} = [k_1, k_2]^T \quad \text{and}$$

with:

$$\Sigma = \begin{bmatrix} \mathbb{E}[\tilde{f}_1(\mathbf{x}^*)^2] & \mathbb{E}[\tilde{f}_1(\mathbf{x}^*)\tilde{f}_2(\mathbf{x}^*)] \\ \mathbb{E}[\tilde{f}_2(\mathbf{x}^*)\tilde{f}_1(\mathbf{x}^*)] & \mathbb{E}[\tilde{f}_2(\mathbf{x}^*)^2] \end{bmatrix} = \begin{bmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{bmatrix}.$$

Both this approach [B] and co-krigging are similar: goal is to reduce the variance of the combined estimator.

```
# now apply multi fidelity thomison
printf("apply multi fidelity thomison..\n");
rho = cor(mu_vals1,mu_vals2);
mu_vals_comb = mu_vals1;
var_vals_comb = var_vals1;
for(i in 1:length(mu_vals1)){
    sig1sq = var_vals1[i];
    sig2sq = var_vals2[i];
    sig1 = sqrt(sig1sq);
    sig2 = sqrt(sig2sq);
    mu1 = mu_vals1[i];
    mu2 = mu_vals2[i];
    mu_vals_comb[i] = ((sig2sq - rho*sig1*sig2)*mu1 + (sig1sq - rho*sig1*sig2)*mu2)/(sig1sq + sig2s
    var_vals_comb[i] = (1 - rho^2)*sig1sq*sig2sq/(sig1sq + sig2sq - 2*rho*sig1*sig2);
}
```

Rho, the correlation coefficient.

[B] Thomison, William D., and Douglas L. Allaire. "A Model Reification Approach to Fusing Information from Multifidelity Information Sources." In *19th AIAA Non-Deterministic Approaches Conference*, p. 1949. 2017.

Combining multiple fidelities

Can calculate correlation coefficient in portions, in terms of the two model means.

```
n = length(mu_vals2);
rho1 = cor(mu_vals2[1:round(n/4)],mu_vals3[1:round(n/4)]);
rho2 = cor(mu_vals2[round(n/4):round(n/2)],mu_vals3[round(n/4):round(n/2)]);
rho3 = cor(mu_vals2[round(n/2):round(3*n/4)],mu_vals3[round(n/2):round(3*n/4)]);
rho4 = cor(mu_vals2[round(3*n/4):round(n)],mu_vals3[round(3*n/4):round(n)]);
mu_vals_comb = mu_vals2;
var_vals_comb = var_vals2;
for(i in 1:length(mu_vals2)){
    if(i<=n/4){
        rho = rho1;
    }
    else if(i>n/4 && i<=n/2){
        rho = rho2;
    }
    else if(i>n/2 && i<=round(3*n/4)){
        rho = rho3;
    }
    else{
        rho = rho4;
    }
    sig1sq = var_vals2[i];
    sig2sq = var_vals3[i];
    sig1 = sqrt(sig1sq);
    sig2 = sqrt(sig2sq);
    mu1 = mu_vals2[i];
    mu2 = mu_vals3[i];
```

The relationship between the two means varies over the testing interval.

To account for this, rho can be calculated locally; for extension, one can look at crossing points of the mean values of two models.

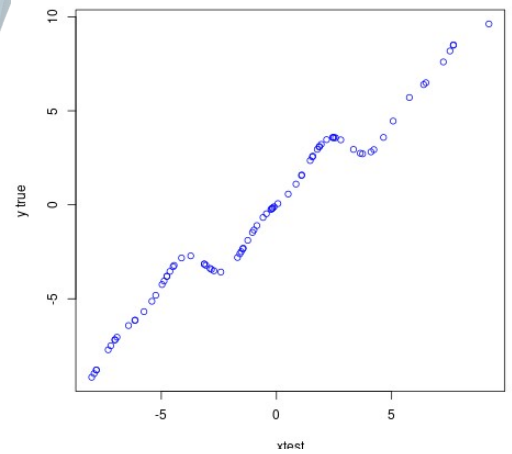
Combining multiple fidelities

Another approach based on simplified co-krigging:

(the information sources are only related to the highest fidelity information source and not to each other). GP models built for all quantities on the right.

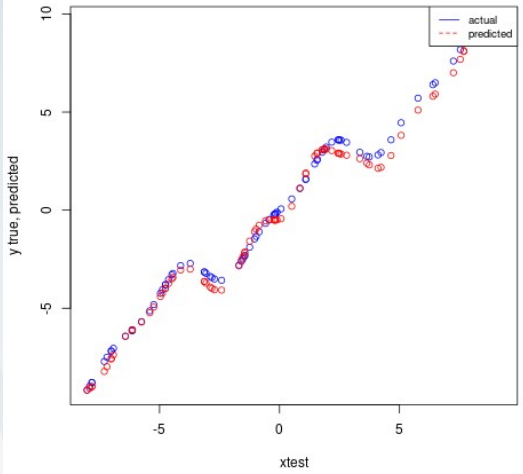
[C] Ghoreishi, Seyede Fatemeh, and Douglas L. Allaire. "Gaussian process regression for Bayesian fusion of multi-fidelity information sources." In *2018 Multidisciplinary Analysis and Optimization Conference*, p. 4176. 2018.

Example: y(x) = sin(x)*(1-cos(x)) + x



200 training values, 100 testing values
Uniform distribution noise, [-0.12,0.12]

Actual values

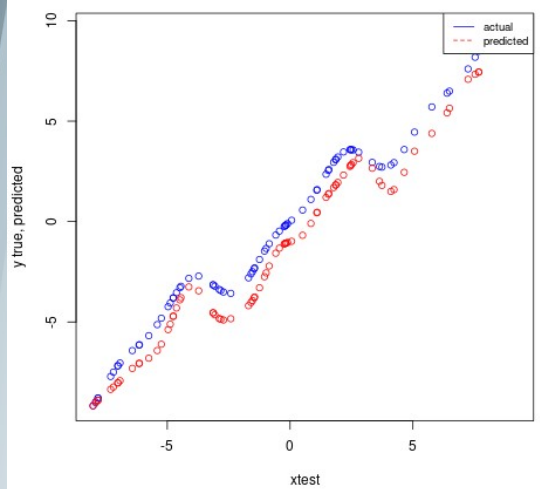


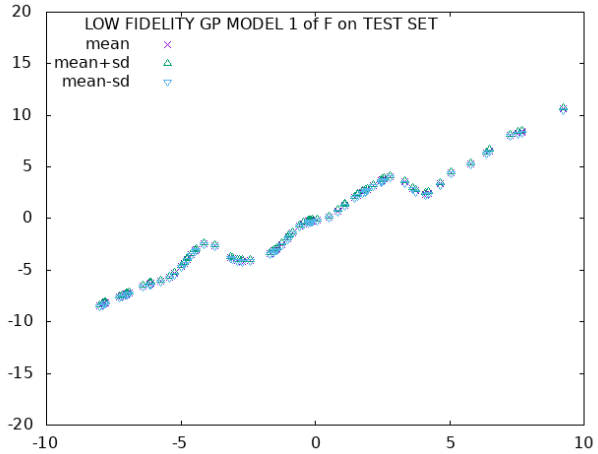
High fidelity GP result with kernel / parameter optimizer.

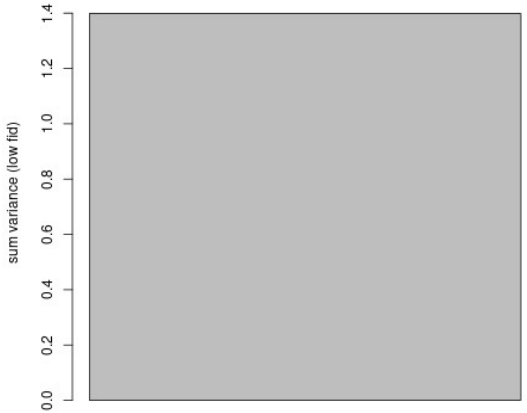
Can we merge two low fidelity approximations and obtain something similar?

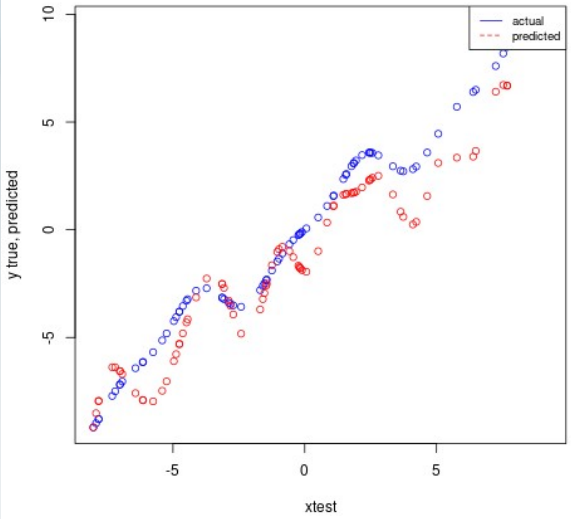
Two low fidelity models

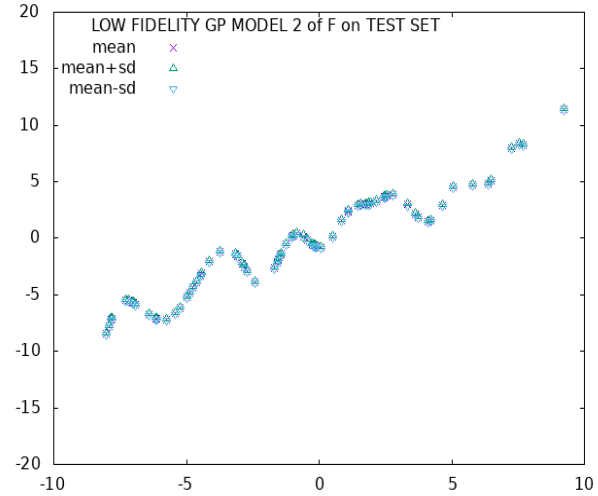
Sum variance over testing interval

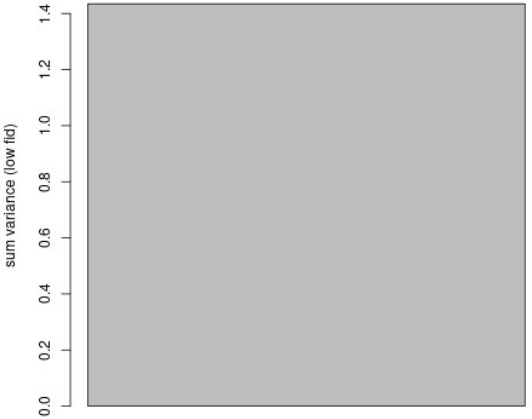




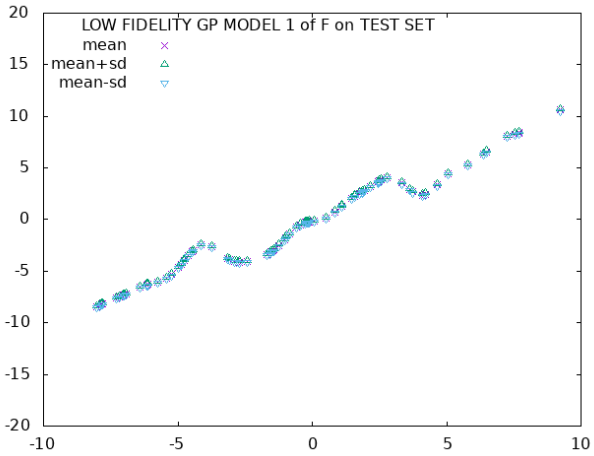


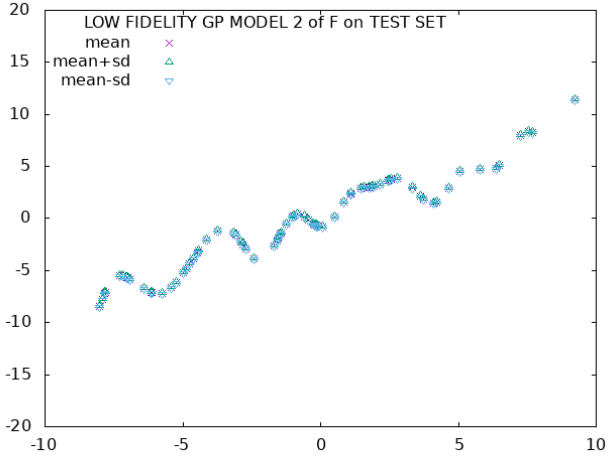


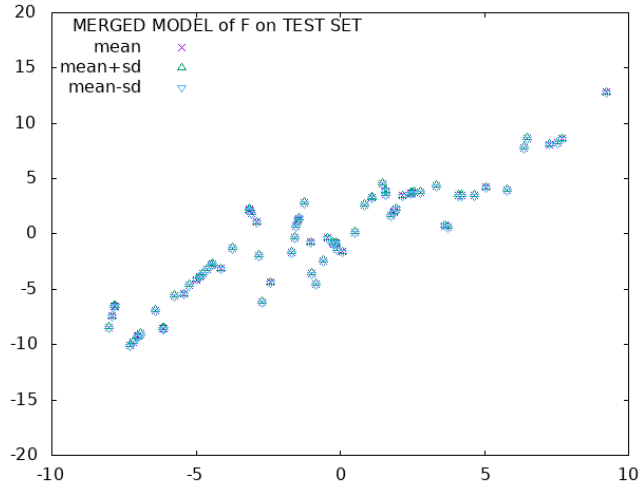


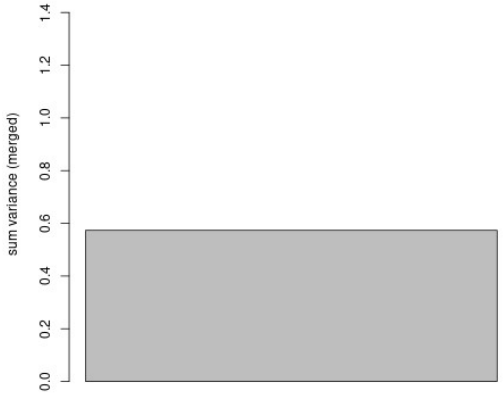


Combining multiple fidelities



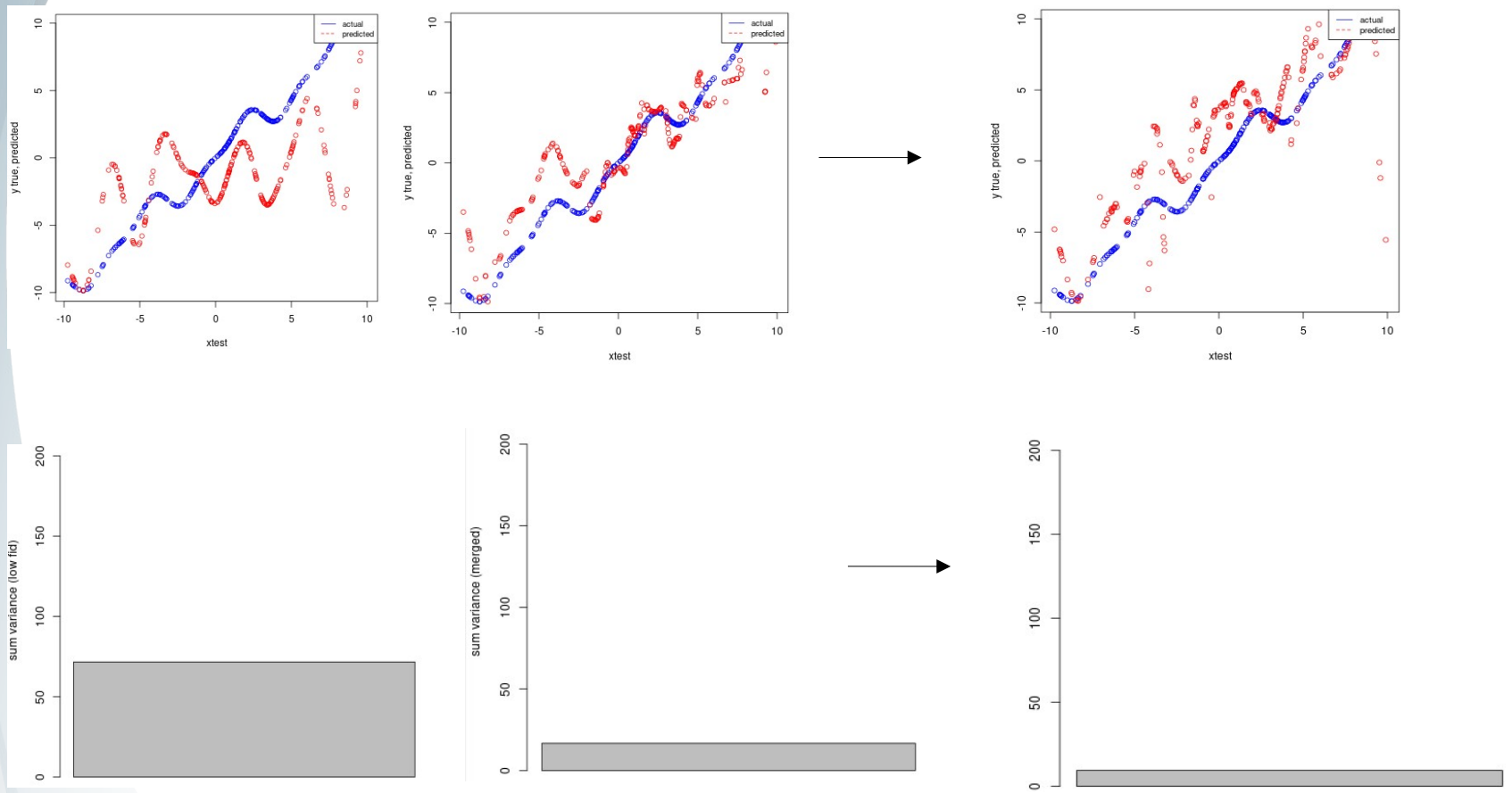






The sum variance has been reduced for the merged model.

Another merged example



Reduction in variance shown on the right is of the merged model parameters.