Applied Gaussian Processes in Stan

Andre Zapico

Organization of this document

This is designed to be an applied intro, that gives background to the notation, intuition, and application of Gaussian Process models using Stan. We hope that this anyone with a background in some probability theory, a little Stan coding, and matrix algebra, will be able to write their own Stan models using Gaussian Processes after reading this case study. We will start with simple regression cases, and then move onto slightly more complex models, along the way, explaining some of the benefits of using Gaussian Process models, all while showing examples on real, publicly available datasets.

Intuition behind Gaussian Processes

Consider a draw (or sometimes called, a "realization") from a probability distribution. For a continuous distribution, say the Normal distirbution, we obtain numerical values that are centered on the mean (commonly referred to as μ) and a dispersion (or a "spread") that we define as the standard deviation (we usually denote *sigma*, but we tend to overload this greek letter with many meanings). In effect, these draws are conditioned on said mean and standard deviation, and the set of possible realizations is an *uncountably infinite* set, at least for continuous distributions. Now let's further generalize this notion of a realization, and imagine we can, instead of drawing random numbers, we can draw *random functions*. Just like conditioning on the mean and standard deviation above, we can condition this space of random functions on what we call *hyper parameters* for a set of random functions. These hyperparameters are denoted by a chosen covariance function, commonly known as a kernel, which defines the relationship between any two observations in a given dataset.

In short, throughout this document, please keep in mind that we are using MCMC methods to estimate a set of random functions that describes our data.

Why gaussian processes over other non-linear modeling techniques?

Gaussian Process Notation

Most branches of mathematics come equipped wit there own set of notation, so I find it important to take a minute to introduce the notation used in much of the Gaussian Process literature. I will try to stay as close to the notation in Gaussian Processes for Machine Learning, by Rasmussen and Williams 2006.

Gaussian Processes are focused on approximating the following target integral, which we call the marginal likelihood:

$$p(y|X) = \int p(y|f, X)p(f|X, \theta)p(\theta)df$$

where θ is a set containting hyper parameters for the kernels specified in the model, X is the data matrix (or design matrix). For example, if we use the squared exponential kernel (also commonly known as the Radial Basis Function, in Stan, we call this `gp_exp_quad_cov` for Gaussian Process Exponential Quadratic Covariance function):

$$k(x', x) = \sigma^2 exp(\frac{d(x', x)^2}{2l^2})$$

then θ in this case is $\theta := \{\sigma, l\}$. We use θ because when the kernel, or model, becomes more complex, it becomes more tedious to write out all of these parameters. You will see this notation in much of the Bayesian Statistics literature. $d(x', x)$ is usually the euclidean distance, but we can use other distance metrics as well. x' and x just refer to rows of X at different axes. This kernel generates very smooth functions. I will later generate some functions from different kernels. If you want to see a picture of some functions generated from this kernel, have a glace at section (**BLAHBLAH**).

The covariance matrix for the joint Gaussian Process is defined as follows:

$$K_{joint} = \begin{pmatrix} K_{X,X} & K_{X,X_*} \\ K_{X_*,X} & K_{X_*,X_*} \end{pmatrix}$$

The upper-left and lower-right submatrices of K_{joint} are square, and their size depends on the division of the dataset into training and test sets. Please not that we will seldom use this joint covariance matrix, K_{joint}, when writing or coding models, it's simply to denote the full process. We can use subsets of this joint process to generate predictions, as we will see later.

A gaussian process simply generates observations from a multivariate Gaussian distribution conditioned on the kernels. So, in the following GP (Gaussian Process) (where μ is the mean function and K_θ is the processes' joint kernel, or covariance matrix conditioned on the hyper parameters:

$$p(y|f) \sim N(f, \sigma) p(f|X, \theta) \sim GP(\mu, K_\theta)$$

we can generate observations for the process by using the well-known equations for the multivariate normal distribution. For example, we generate the predictive out of sample mean for the GP above by using $K_{f,f}$ and $K_{f*,f}$ from using subsets of the joint kernel above, K_{joint}. The estimates predictive mean for the latent $f*$ then becomes:

$$f_* = K_{X_*,X}(K_{X,X})f$$

and if want to generate noisy predictions for the predictive mean, similarly, we add some noise on the diagnonal of the training covariance matrix, $K_(X, X)$.

$$f_* = K_{X_*,X}(K_{X,X} + \sigma_n^2 I)f$$

where σ_n^2 is some length n vector indicating the amount of noise we would like to include in our predictions, and I is the identity matrix.

Similarly, we can find the process covariance matrix by using the equations for the multivariate Gaussian Distribution's covariance. We leverage all four disjoint subsets of the full GP's covariance matrix above. First, the non-noisy case:

$$cov(f_*) = K_{f_*,f_*} - K_{f_*,f} K_{f,f}^{-1} K_{f,f_*}$$

and then, in the noisy case, we add a diagonal matrix with some noise, as above:

$$cov(f_*) = K_{f_*,f_*} - K_{f_*,f}(K_{f,f}^{-1} + \sigma_n^2 I) K_{f,f_*}$$

Gaussian Processes with Gaussian Likelihoods

In the simplest case, we start with a regression model using GP's with Gaussian likelihoods. These have analytic solutions for a subset of realizations of the full model, but we will use Stan and NUTS (the No-U-Turn Sampler), anyway.

It is important to note that in the case of a Gaussian Likelihood, that in the target intregral above:

$$p(y|X) = \int p(y|f, X)p(f|X, \theta)p(\theta)df$$

the f, the latent function in the GP, becomes integrated out, since the following integral is proportional to the one above:

$$p(y|X) \propto \int p(y|X, \theta)p(\theta)df$$

and we need not use the latent f and can instead use y and $y*$, the training and testing y, respectively for our predictive distributions. The predictive mean and covariance for the process (for non-noisy observations) are then as follows:

$$y_* = K_{X_*,X}(K_{X,X})y$$

$$cov(y_*) = K_{X_*,X_*} - K_{X_*,X}K_{X,X}^{-1}K_{X,X_*}$$

where $y*$ denotes the estimates of the process for y, $K_{X,X}$ is the training kernel, $K_{X_*,X}$ is the cross covariance between training and test sets, etc.

Regression

We'll start with a simple non-linear regression model using Gaussian Processes. The dataset we'll use first can be obtain from the University of California Irvine (UCI) Machine Learning Repository, is entitled Boston Housing, and contains, unsuprisingly, data concerning suburbs of Boston in 1993, such as per capita crime rate, a dummy variable if a home tract the river's boundarys, etc. We'll take y to be Nitric oxide concentration N02, and Regress it on all other covariates. Here's how we format the data. And for consistency, this is how we'll do it for all of our models:

```
suppressMessages(library(rstan));                          # load rstan
set.seed(8128);                                            # for replicability

# gp regression, housing data
dt = read.csv("housing.txt", sep = "")
indeces = sample(1:nrow(dt));                              # randomize indeces
dt = dt[indeces, ];                                        # shuffle the dataset
dt = scale(dt);                                            # scale the dataset
train = floor(.75 * nrow(dt));                             # take 75% of the shuffled dataset as traini

N = train;                                                 # number of training points
D = ncol(dt) - 1;                                          # the dimension of the GP, num columns of K(
x = as.matrix(dt[1:train, -5]);
y = as.vector(dt[1:train, 5]);                             # predict N02 concentration

N_pred = nrow(dt) - train;                                 # number of out of sample to predict
```

```r
x_pred = as.matrix(dt[(train + 1):nrow(dt), -5]);          # prediction matrix, K(X*,X*)

stan_rdump(list = c("N", "D", "x", "y", "N_pred", "x_pred"),
           "housing_input.data.R");
y_test = dt[(train +1):nrow(dt), 5];                        # obtain the test set
```

We're using `stan_rdump` because `cmdstan`, the command line version of Stan, is my preferred Stan interface. It's the most stable, and it's the easist to use when you're doing dev for the Stan Math Library. So we've formatted our data properly, but what does out Stan model look like? Since this is (perhaps) our first time writing a GP model using Stan, we'll walk through most of the lines of the Stan model.

Now, let's have a look at the Stan code, and see what's happening!

```
functions {
  vector gp_pred_rng(vector[] x_pred,
                     vector y, vector[] x,
                     real magnitude, real length_scale) {
    int N1 = rows(y);
    int N2 = size(x_pred);
    vector[N2] y_pred;
    {
      matrix[N1, N1] K =   cov_exp_quad(x, magnitude, length_scale);
      matrix[N1, N1] L_K = cholesky_decompose(K);

      vector[N1] L_K_div_y = mdivide_left_tri_low(L_K, y);
      vector[N1] K_div_y = mdivide_right_tri_low(L_K_div_y', L_K)';
      matrix[N1, N2] k_x_x_pred = cov_exp_quad(x, x_pred, magnitude, length_scale);
      vector[N2] y_pred_mu = (k_x_x_pred' * K_div_y);
      matrix[N1, N2] v_pred = mdivide_left_tri_low(L_K, k_x_x_pred);
      matrix[N2, N2] cov_y_pred =   cov_exp_quad(x_pred, magnitude, length_scale) - v_pred' * v_pred
                              + diag_matrix(rep_vector(1e-6, N2));
      y_pred = multi_normal_rng(y_pred_mu, cov_y_pred);
    }
    return y_pred;
  }
}
data {
  int<lower=1> N;
  int<lower=1> D;
  vector[D] x[N];
  vector[N] y;

  int<lower=1> N_pred;
  vector[D] x_pred[N_pred];
}
transformed data {
  vector[N] mu;
  mu = rep_vector(0, N);
}
parameters {
  real<lower=0> magnitude;
  real<lower=0> length_scale;

  real<lower=0> eta;
}
```

```
transformed parameters {
  matrix[N, N] L_K;
  {
    matrix[N, N] K;
    K = gp_exp_quad_cov(x, magnitude, length_scale);
    K = add_diag(K, square(eta));
    L_K = cholesky_decompose(K);
  }
}
model {
  magnitude ~ normal(0, 2);
  length_scale ~ inv_gamma(5, 5);

  y ~ multi_normal_cholesky(mu, L_K);
}
generated quantities {
  vector[N_pred] f_pred = gp_pred_rng(x_pred, y, x, magnitude, length_scale);
  vector[N_pred] y_pred;

  for (n in 1:N_pred) y_pred[n] = normal_rng(f_pred[n], 1.0); // out of sample predictions
}
```

Since this could be the first Stan GP model we're using, I'm going to go through the model in some detail.

The function `gp_pred_rng` is simply producing the posterior predictive mean and covariance, as we've see over and over above. There are some specialized functions that make the linear algebra computations that Stan uses quicker, but please look in the Stan reference manual for more details.

The `data` block is what one might expect based on our intro. For example, our `vector[D] x[N]` is our X above, and `x_pred` is our X_* above, the test set and lower right of X_{joint}.

In `transformed data`, we're mean centering our GP, so we need a vector of 0's to represent the multivariate mean.

In `parameters`, these are the hyper parameters for our chosen covariance function, the squared exponential:

$$k(x', x) = \sigma^2 exp(\frac{d(x', x)^2}{2l^2})$$

where σ is the magnitude and l is the length scale. The ranges are restricted to be positive.

In `transformed` parameters, we're generating a kernel, and taking the cholesky decomposition (we can think of this as the "square root" of a matrix). We add a small constant to the diagonal of this matrix prior to taking the cholesky decomposition because this gives us some numerical stability (we call this jitter). We take cholesky decomposition because we want to sample from a multivariate Gaussian distribution (hence, "Gaussian" process) as follows:

$\mu + L'N(0, 1)$

where $K = L'L$ is the cholesky decomposition.

In `model`, as usual we specify the priors for our hyperparameters, and the likelihood function, which is just a guassian. We use the specialzed `multi_normal_cholesky` because this helps optimize sampling speed a bit.

In `generated quantities`, we generate normal observations from our posterior predictive distribution. In the case where we have a gaussian likelihood,

An aside on prior distributions for Gaussian Process Hyper parameters

Run the model

After we compile the model with commandstan, using `$CMDSTANDR/make /path/to/gp_regression`, we can then sample with the following command:

```
$./gp_regression sample num_warmup=400 num_samples=400 data file=housing_input.data.R
output file=gp_regression_housing_output.csv
```

Let's do some convergene diagnostics.

```
gp_csv = read_stan_csv('gp_regression_housing_output.csv')              # read in cmdstan outputs
summary(gp_csv)$summary[1:3,];                                              # quick diagnostics for pa
```

```
##                   mean    se_mean        sd       2.5%       25%       50%
## magnitude    0.9938912 0.04607148 0.3398740 0.57591445 0.7406147 0.9180545
## length_scale 4.3467515 0.12358812 1.1082253 2.67033925 3.5852750 4.1424900
## sig          0.8659040 0.04325343 0.6116959 0.05867075 0.3702530 0.7208260
##                   75%    97.5%     n_eff      Rhat
## magnitude    1.134680 1.819601  54.42165 0.9985008
## length_scale 4.872372 6.982028  80.40863 0.9973795
## sig          1.244137 2.179730 200.00000 0.9960689
```

Since this is the first model, let's take a look at summary for some preliminary diagnostics. When running chains with cmdstan, there were no divergences, which is great. Upon looking at the `summary` of our output. $\hat{R}$ is close to 1, meaning we've let out chains run long enough so get an adequate sample of the posterior distribution. We also look at the effective sample size. A "perfect" effective sample size, N_{eff} would be equal to the `num_samples` we specfied when running command stan. Since I ran about 400 samples, the effective sample size is a bit low, which could indicate some model misspecification. We also check trace plots and ACF functions to make sure we have "good" sampling from posterior parameters and the draws from out posterior are approximately uncorrelated, although I'm leaving these plots out for now.

Let's do a Bayesian multiple linear regression, and use Root Mean Squared Error (RMSE) to see how the GP regression with the squared exponential kernel performs in comparison with a linear regression model.

Here's the code for a Bayesian linear regression. This isn't necessarily recommended, but we're using $N(0,1)$ priors on regression coefficients, for simplicity. We write the Stan code so we can use the same input data as the example above, for both good workflow and replicability:

```
data {
  int<lower=1> N;
  int<lower=1> D;
  matrix[N, D] x;
  vector[N] y;

  int<lower=1> N_pred;
  matrix[N_pred, D] x_pred;
}
parameters {
  real alpha;
  vector[D] beta;
  real<lower=0> sigma;
}
model {
  alpha ~ normal(0, 1);                 // intercept prior
  beta ~ normal(0, 1);                  // regression coef priors
  sigma ~ normal(0, 1);                 // model noise

  y ~ normal(alpha + x * beta, sigma);  // likelihood function
```

```
}
generated quantities {
  vector[N_pred] y_pred;
  for (n in 1:N_pred) y_pred[n] = normal_rng(alpha + x_pred[n] * beta, sigma);
}
```

Assuming I've observed appropriate convergence diagnostics, and there are no issues, let's calculate RMSE to compare models, again, for simplicity comparing the means:

```
gp_csv = read_stan_csv('gp_regression_housing_output.csv');
lr_csv =
  read_stan_csv('linear_regression_housing_output.csv');
gp_samples = extract(gp_csv);
lr_samples = extract(lr_csv);
## calculate RMSE
sqrt(sum((colMeans(gp_samples$y_pred) - y_test) ^ 2));
```

```
## [1] 13.96043
```

```
sqrt(sum((colMeans(lr_samples$y_pred) - y_test) ^ 2));
```

```
## [1] 18.12313
```

And we can see, the RMSE of the linear model is slightly lower than the GP regression with a non-linear kernel.

a note on prior distribution selection

Regression with Automatic Relevance Determination (ARD) priors.

Earlier, I mentioned there could be an issue with model misspecficiation due to low effective sample size. The natural remedy for this is to do a simulation of the generative process, and see if your model is able to recover parameters in the generative process. This is highly recommended, but I will neglect to do that here. Instead, I'll take a different approach that I know will make my model naturally more flexible. We've so far assumed that for the kernel, each dimension has the same length scale. We're assuming, that nomatter what covariate, the relationship as we vary along any two axis (or two observations) is the same. This is not a good assumption, though, correct?

The formula for the squared exponential kernel looks like this, with i, j indexing the rows, where d is indexing each "column", or dimension of the dataset:

$$k(x', x) = \sigma^2 exp(\frac{d(x', x)^2}{2l_d^2})$$

To implement a seperate length scale for each dimension in a gaussian process regression in Stan, we can simply extend the length scale argument to have `D` length scales, i.e. `real<lower=0> lengh_scale` to `real<lower=0> length_scale[D]` and update the `pred_rng` function to take in a vector of length scales. The function signature changing from: `vector gp_pred_rng(vector[] x_pred, vector y, vector[] x, real magnitude, real length_scale)` to `vector gp_pred_rng(vector[] x_pred, vector y, vector[] x, real magnitude, real[] length_scale)`. The full stan code for an ARD regression with a squared exponential kernel is almost identical to the above GP regression:

```
functions {
  vector gp_pred_rng(vector[] x_pred,
                     vector y1, vector[] x,
                     real magnitude, real[] length_scale,
```

```
                     real sigma) {
    int N = rows(y1);
    int N_pred = size(x_pred);
    vector[N_pred] f2;
    {
      matrix[N, N] K = add_diag(gp_exp_quad_cov(x, magnitude, length_scale),
                               sigma);
      matrix[N, N] L_K = cholesky_decompose(K);
      vector[N] L_K_div_y1 = mdivide_left_tri_low(L_K, y1);
      vector[N] K_div_y1 = mdivide_right_tri_low(L_K_div_y1', L_K)';
      matrix[N, N_pred] k_x_x_pred = gp_exp_quad_cov(x, x_pred, magnitude, length_scale);
      f2 = (k_x_x_pred' * K_div_y1);
    }
    return f2;
  }
}
data {
  int<lower=1> N;
  int<lower=1> D;
  vector[D] x[N];
  vector[N] y;

  int<lower=1> N_pred;
  vector[D] x_pred[N_pred];
}
transformed data {
  vector[N] mu;
  mu = rep_vector(0, N);
}
parameters {
  real<lower=0> magnitude;
  real<lower=0> length_scale[D];
  real<lower=0> sig;

  real<lower=0> sigma;
}
transformed parameters {
  matrix[N, N] L_K;
  {
    matrix[N, N] K = gp_exp_quad_cov(x, magnitude, length_scale);
    K = add_diag(K, square(sigma));
    L_K = cholesky_decompose(K);
  }
}
model {
  magnitude ~ normal(0, 3);
  length_scale ~ inv_gamma(5, 5);
  sig ~ normal(0, 1);

  sigma ~ normal(0, 1);

  y ~ multi_normal_cholesky(mu, L_K);
}
generated quantities {
```

```
  vector[N_pred] f_pred = gp_pred_rng(x_pred, y, x, magnitude, length_scale, sigma);
  vector[N] f_pred_in = gp_pred_rng(x, y, x, magnitude, length_scale, sigma);
  vector[N_pred] y_pred;
  vector[N] y_pred_in;

  for (n in 1:N_pred) y_pred[n] = normal_rng(f_pred[n], sigma); // out of sample predictions
  for (n in 1:N) y_pred_in[n] = normal_rng(f_pred_in[n], sigma); // out of sample predictions
}
```

and after, we can again compare RMSE of the 3 models:

```
## load new regression model
gp_ard_csv = read_stan_csv("gp_regression_ard_housing_output.csv")
gp_ard_samples = extract(gp_ard_csv);

## calculate RMSE
sqrt(sum((colMeans(lr_samples$y_pred) - y_test) ^ 2));
```

```
## [1] 18.12313
```

```
sqrt(sum((colMeans(gp_samples$y_pred) - y_test) ^ 2));
```

```
## [1] 13.96043
```

```
sqrt(sum((colMeans(gp_ard_samples$y_pred) - y_test) ^ 2));
```

```
## [1] 3.147226
```

and the out of sample RMSE for the ARD regression (seperate length scale for each dimension) drops drastically.

Summing and Multiplying Kernels

We've seen one technique that can help improve out of sample prediction of models. One convenient property of Gaussian Processes is the ability to combine kernels. More specifically, we can use summation and *element wise* multiplication, known as the Hamard product to exploit structure in the data to gain interpretability and predictive performance. For example, if $k_1, k_2, ..., k_n$ are all covariance function, any combination of Hamard products or sums is still a covariance function:

$$K = k_1(x, x') \circ k_2(x, x') + k_3(x, x')$$

where $\circ$ is the Hamard Product, or element wise multiplication. As you can see, any combination of kernels is still a kernel. Let's see how it works in practice. We've only seen so far the squared exponential kernel. Another convenient kernel is the dot product kernel. From a descrption in Rasmussen and Williams, we can think of the dot product kernel as a Bayesian Regression with $N(0, 1)$ priors on regression coefficients. We define the dot product kernel as follows:

$$k(x', x) = \sigma_o^2 + x \cdot x'$$

Let's now use the same GP regression model above, except now summing the dot product kernel and the squared exponential kernel. The joint covariance function will now be:

$$K = k_1(x, x') + k_2(x, x')$$

where $k_1(x, x') = \sigma_o^2 + x \cdot x'$ and $k_2(x, x') = \sigma^2 exp(\frac{d(x', x)^2}{2l_d^2})$

The Stan code is again similar, we just update our gp_pred_rng and our model to have the desired combination of kernels:

```
functions {
  vector gp_pred_rng(vector[] x_pred,
                     vector y1, vector[] x,
                     real magnitude, real[] length_scale,
                     real sig_0,
                     real sigma) {
    int N = rows(y1);
    int N_pred = size(x_pred);
    vector[N_pred] f2;
    {
      matrix[N, N] K = add_diag(gp_dot_prod_cov(x, sig_0) +
                                gp_exp_quad_cov(x, magnitude, length_scale),
                                sigma);
      matrix[N, N] L_K = cholesky_decompose(K);
      vector[N] L_K_div_y1 = mdivide_left_tri_low(L_K, y1);
      vector[N] K_div_y1 = mdivide_right_tri_low(L_K_div_y1', L_K)';
      matrix[N, N_pred] k_x_x_pred = gp_dot_prod_cov(x, x_pred, sig_0) +
        gp_exp_quad_cov(x, x_pred, magnitude, length_scale);
      f2 = (k_x_x_pred' * K_div_y1);
    }
    return f2;
  }
}
data {
  int<lower=1> N;
  int<lower=1> D;
  vector[D] x[N];
  vector[N] y;

  int<lower=1> N_pred;
  vector[D] x_pred[N_pred];
}
transformed data {
  vector[N] mu;
  mu = rep_vector(0, N);
}
parameters {
  real<lower=0> magnitude;
  real<lower=0> length_scale[D];

  real<lower=0> sig_0;

  real<lower=0> sigma;
}
transformed parameters {
  matrix[N, N] L_K;
  {
    matrix[N, N] K = gp_dot_prod_cov(x, sig_0) + gp_exp_quad_cov(x, magnitude, length_scale);
    K = add_diag(K, square(sigma));
    L_K = cholesky_decompose(K);
  }
```

```
}
model {
  magnitude ~ normal(0, 3);
  length_scale ~ inv_gamma(5, 5);
  sig_0 ~ normal(0, 1);

  sigma ~ normal(0, 1);

  y ~ multi_normal_cholesky(mu, L_K);
}
generated quantities {
  vector[N_pred] f_pred = gp_pred_rng(x_pred, y, x, magnitude, length_scale, sig_0,sigma);
  vector[N] f_pred_in = gp_pred_rng(x, y, x, magnitude, length_scale, sig_0, sigma);
  vector[N_pred] y_pred;
  vector[N] y_pred_in;

  for (n in 1:N_pred) y_pred[n] = normal_rng(f_pred[n], sigma); // out of sample predictions
  for (n in 1:N) y_pred_in[n] = normal_rng(f_pred_in[n], sigma); // out of sample predictions
}
```

After running model, we notice a few improvements. The first, being convergence. The model fits the data better, we can see an increase effective sample for all of the GP kernel hyperparameters when we look at the summary.

If we compare the RMSE again, we see and even better increase in performance. The posterior means for the sum of the dot product at Squared Exponential ARD model are even closer to the actual y for the data:

```r
gp_2_csv = read_stan_csv("gp_regression_2_housing_output.csv");
gp_2_samples = extract(gp_2_csv);
## calculate RMSE
sqrt(sum((colMeans(lr_samples$y_pred) - y_test) ^ 2));
```

```
## [1] 18.12313
```

```r
sqrt(sum((colMeans(gp_samples$y_pred) - y_test) ^ 2));
```

```
## [1] 13.96043
```

```r
sqrt(sum((colMeans(gp_ard_samples$y_pred) - y_test) ^ 2));
```

```
## [1] 3.147226
```

```r
sqrt(sum((colMeans(gp_2_samples$y_pred) - y_test) ^ 2));
```

```
## [1] 2.940514
```

Using different kernels

As we can see combining different kernels can give us more accurate predictive distributions. Let's introduce a few more kernels. We've since seen a very common kernel, the squared exponential kernel:

$$k(x', x) = \sigma^2 exp(\frac{d(x', x)^2}{2l^2})$$

This is actually a special case of a larger class of kernels, known as the Matern Kernels, which have the following form:

$$k(x,x') = \frac{2^{1-v}}{\Gamma(v)}(\frac{\sqrt{2v}d(x,x')}{l})^v K_v(\frac{\sqrt(2vr)}{v})$$

where K_v is the modified Bessel Function and Γ is the gamma function, v and l are parameters. It is common to take v as the values $\frac{1}{2}, \frac{3}{2}, \frac{5}{2}$, and as $v \to \infty$, we recover the squared exponential kernel above. The $v = \frac{1}{2}$ is also known as the exponential kernel, and the $v = \frac{3}{2}, v = \frac{5}{2}$ are just referred to as the Matern32 and Matern52 respectively. There functional forms are as follows, in order of increasing smoothness:

$$k(x,x')_{v=\frac{1}{2}} = exp(-\frac{d(x,x')^2}{2l^2})$$

$$k(x,x')_{v=\frac{3}{2}} = (1 + \frac{\sqrt{3}d(x,x')}{l})exp(-\frac{\sqrt{3}d(x,x')}{l})$$

$$k(x,x')_{v=\frac{5}{2}} = (1 + \frac{\sqrt{5}d(x,x')}{l} + \frac{5d(x,x')^2}{3l^2})exp(-\frac{\sqrt{5}d(x,x')}{l})$$

$$k(x',x)_{v\to\infty} = exp(\frac{d(x',x)^2}{2l^2})$$

Now let's simulate a dataset and fit a model to these kernels, so we can have a picture of what kind of functions these kernels can generate independently. We'll simulate from 1D GP's from these kernels for ease of visualization. For the squared exponential kernel, simulating observations is as follows:

```
data {
  int<lower=1> N;
  real x[N];

  real<lower=0> length_scale;
  real<lower=0> magnitude;
  real<lower=0> sigma;
}

transformed data {
  matrix[N, N] cov =   add_diag(gp_exp_quad_cov(x, magnitude, length_scale), 1e-10);
  matrix[N, N] L_cov = cholesky_decompose(cov);
}
parameters {}
model {}
generated quantities {
  vector[N] f = multi_normal_cholesky_rng(rep_vector(0, N), L_cov);
  vector[N] y;
  for (n in 1:N)
    y[n] = normal_rng(f[n], sigma);
}
```

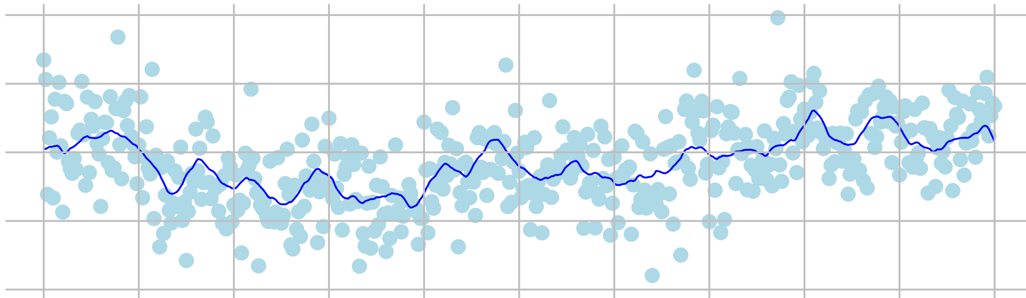
The simulated data is as follows:

```
# fix length scale and model noise to 1.0 for example
magnitude = 1.0
length_scale = 1.0
sigma = 1.0

N = 501
x = 20 * (0:(N - 1)) / (N - 1) - 10

stan_rdump(list = c("magnitude", "length_scale","sigma", "N", "x"), file = "simu.data.R");
```

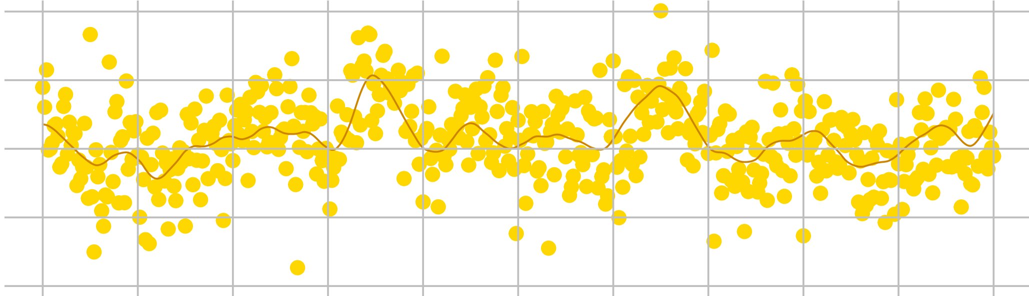
and I'll use the following `cmdstan` command to generate the data (linux): `./simulate_exp_quad sample num_samples=1 num_warmup=0 algorithm=fixed_param data file=simu.data.R output file=simu_data_exp_quad.csv`

The following is *one* realization of a random function from each of the Matern32, Matern52, and the Exponential Quadratic Covariance Functions (there are uncountably infinite random functions for each covariance function). They are visualized on a grid evenly spaced at intervals of 2, each center line being 0. Notice how the functions are smoother as v increases.

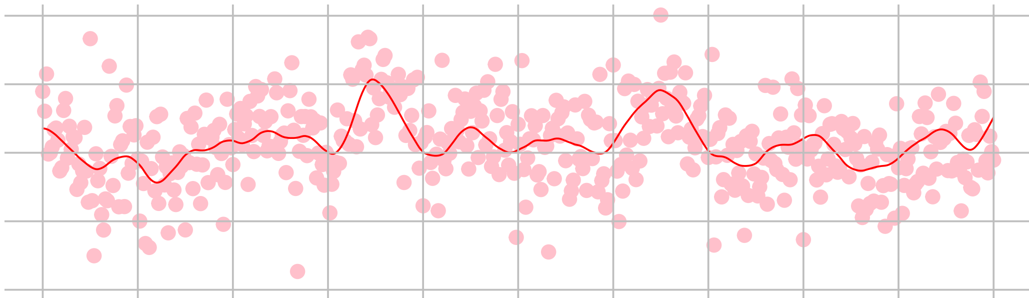
$$k(x,x')_{v=\frac{3}{2}} = (1 + \frac{\sqrt{3}d(x,x')}{l})exp(-\frac{\sqrt{3}d(x,x')}{l})$$



$$k(x,x')_{v=\frac{5}{2}} = (1 + \frac{\sqrt{5}d(x,x')}{l} + \frac{5d(x,x')^2}{3l^2})exp(-\frac{\sqrt{5}d(x,x')}{l})$$



$$k(x',x)_{v\to\infty} = exp(\frac{d(x',x)^2}{2l^2})$$



Up next:

In part two we'll explore a few more cases of applied models in Stan using Gaussian Processes: Modeling time series with Gaussian Processes, classiciation using Gaussian Processes, and a survival model using GPs.