Bayesian Optimization Tutorial

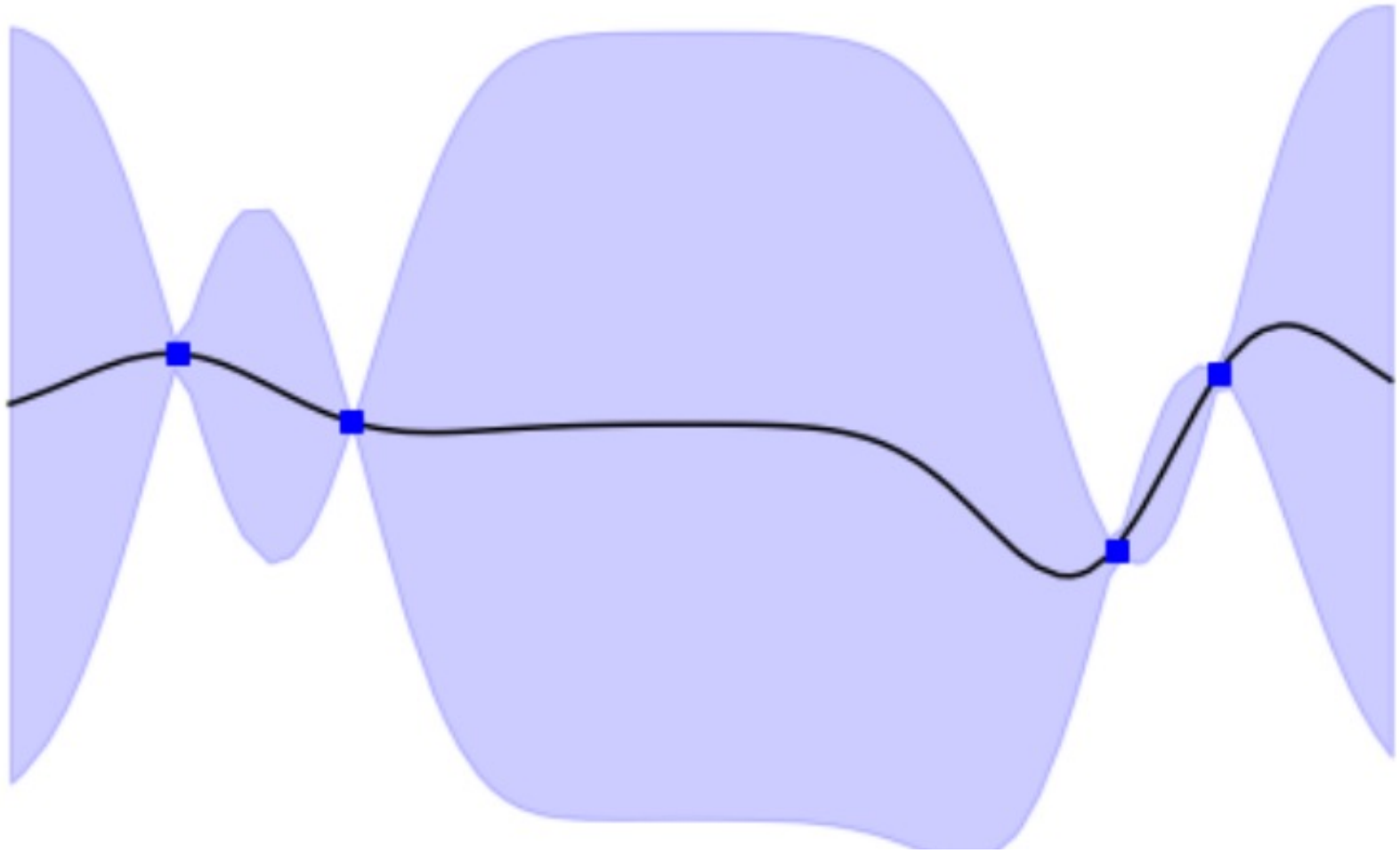
Module 1: Probabilistic Surrogate Modeling

Joel Paulson
Assistant Professor, Department of Chemical and Biomolecular Engineering, The Ohio State University

Great Lakes PSE Student Workshop, 2023

For copies of slides & code, see
https://github.com/joelpaulson/Great_Lakes_PSE_Workshop_2023

Probabilistic Models are Key for Bayesian Optimization

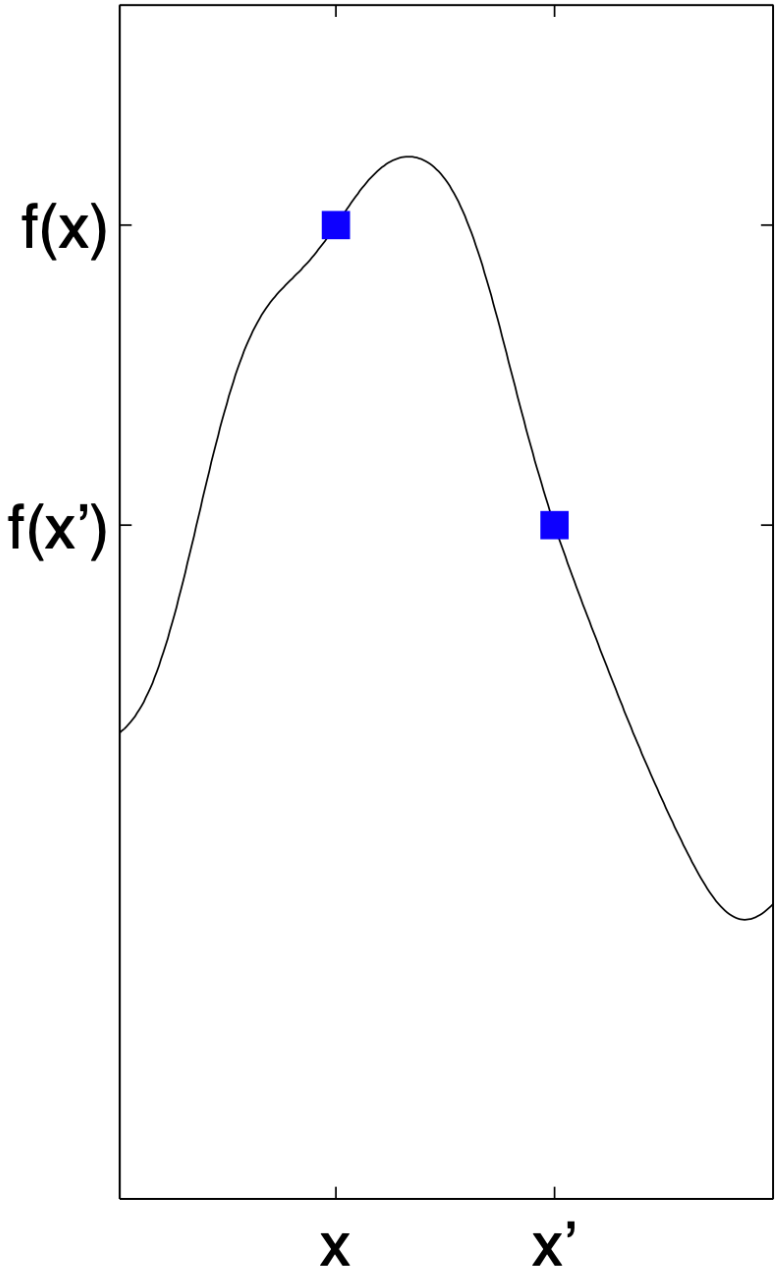


What Modeling Options are Available?

- Many options are available for training probabilistic surrogate models
  - Bayesian Linear Regression (or Neural Networks)
  - Random Forests
  - Tree-Structured Parzen Estimators
  - Gaussian Processes

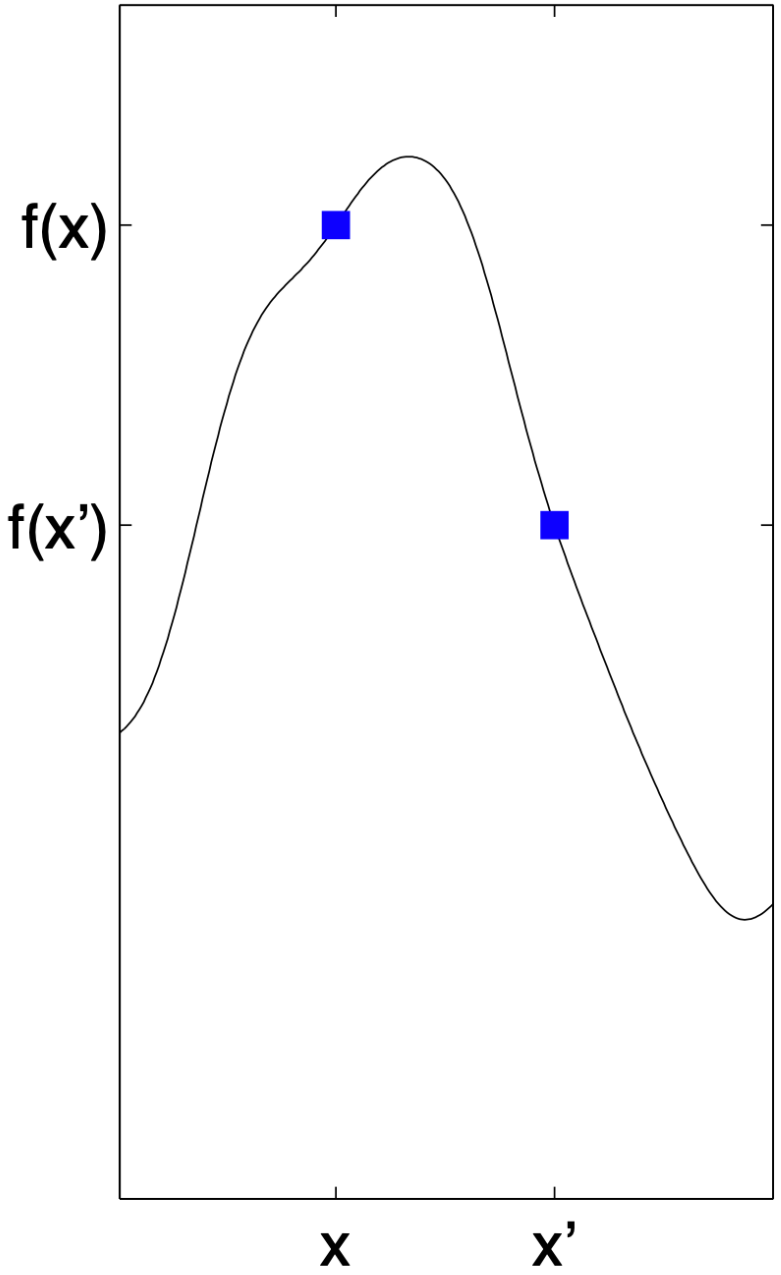
- The model must be probabilistic in order to systematically represent uncertainty (roughly think of this as providing confidence bounds)

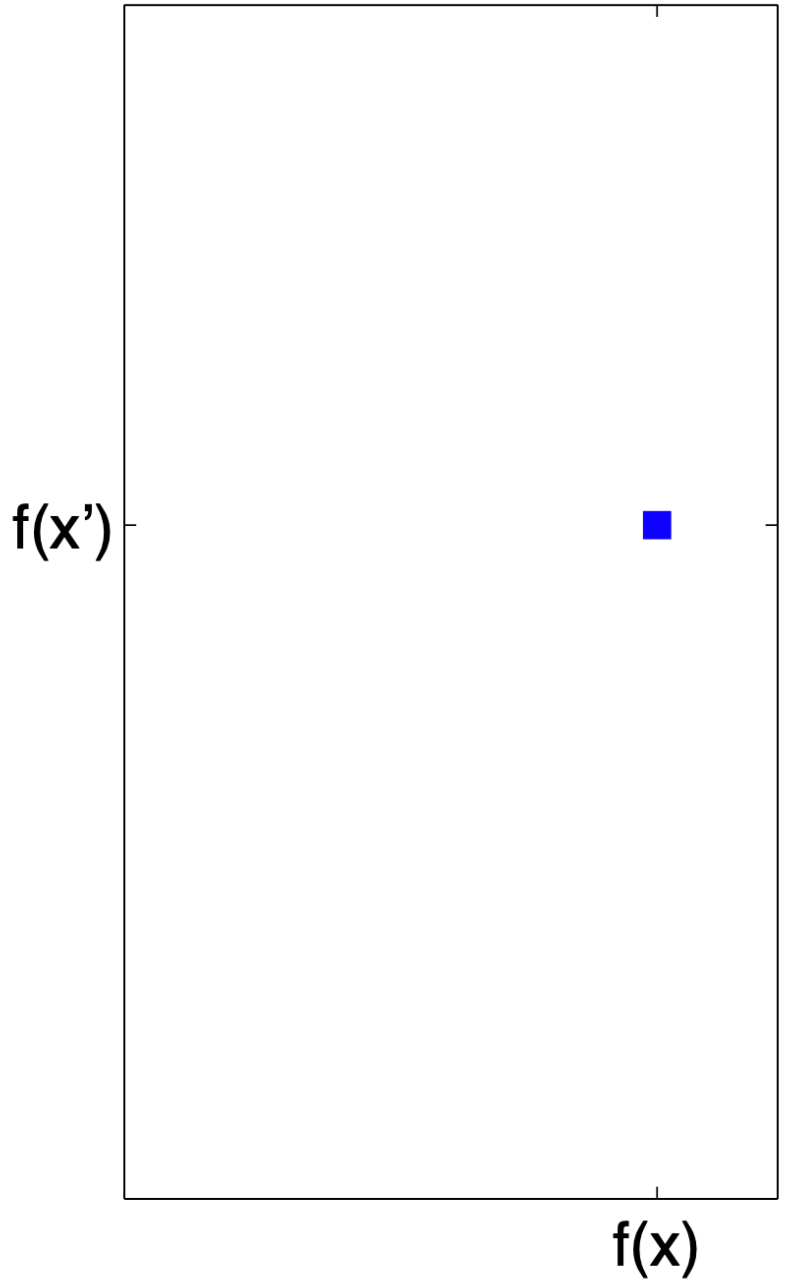
- Bayesian optimization focuses on Gaussian processes for various reasons (mostly computationally tractable + very flexible)



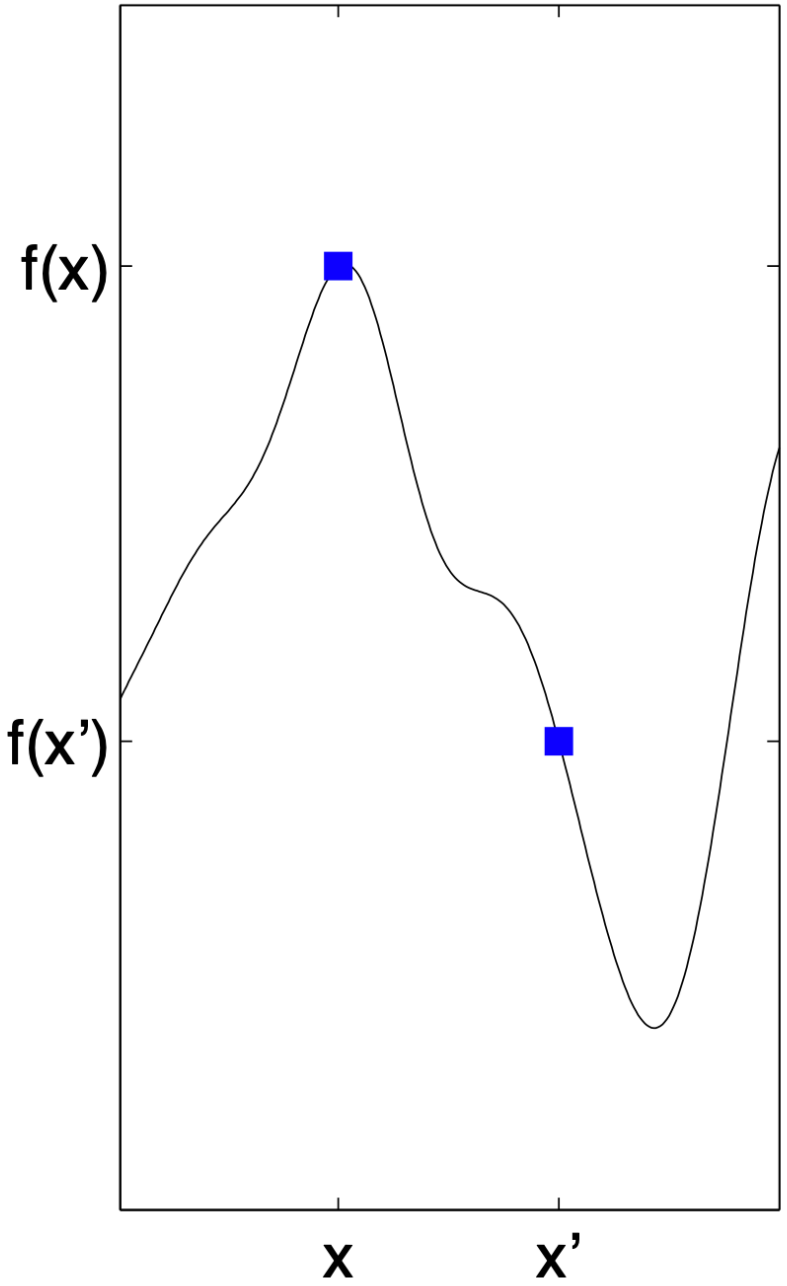
Let's Start Simply

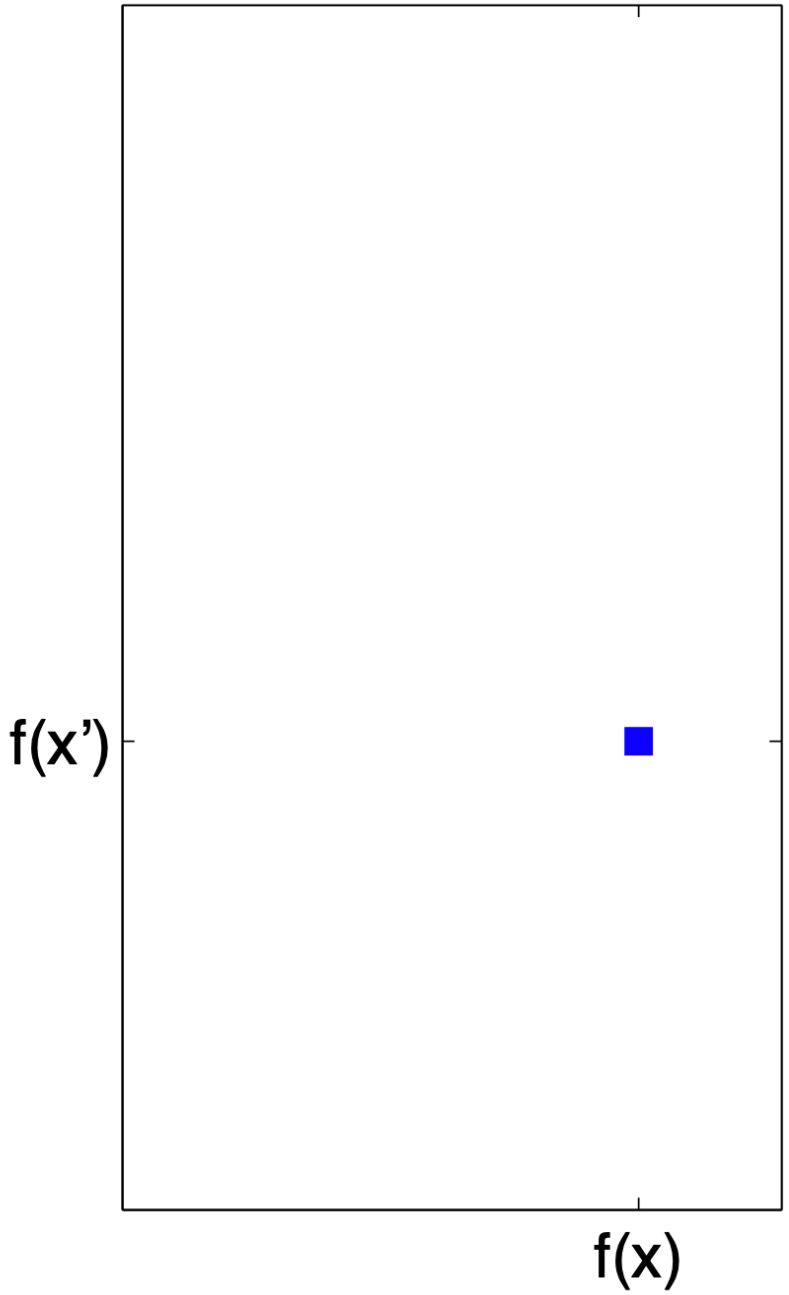
- Fix two points x and x'
- Consider the vector of outputs [f(x), f(x')]



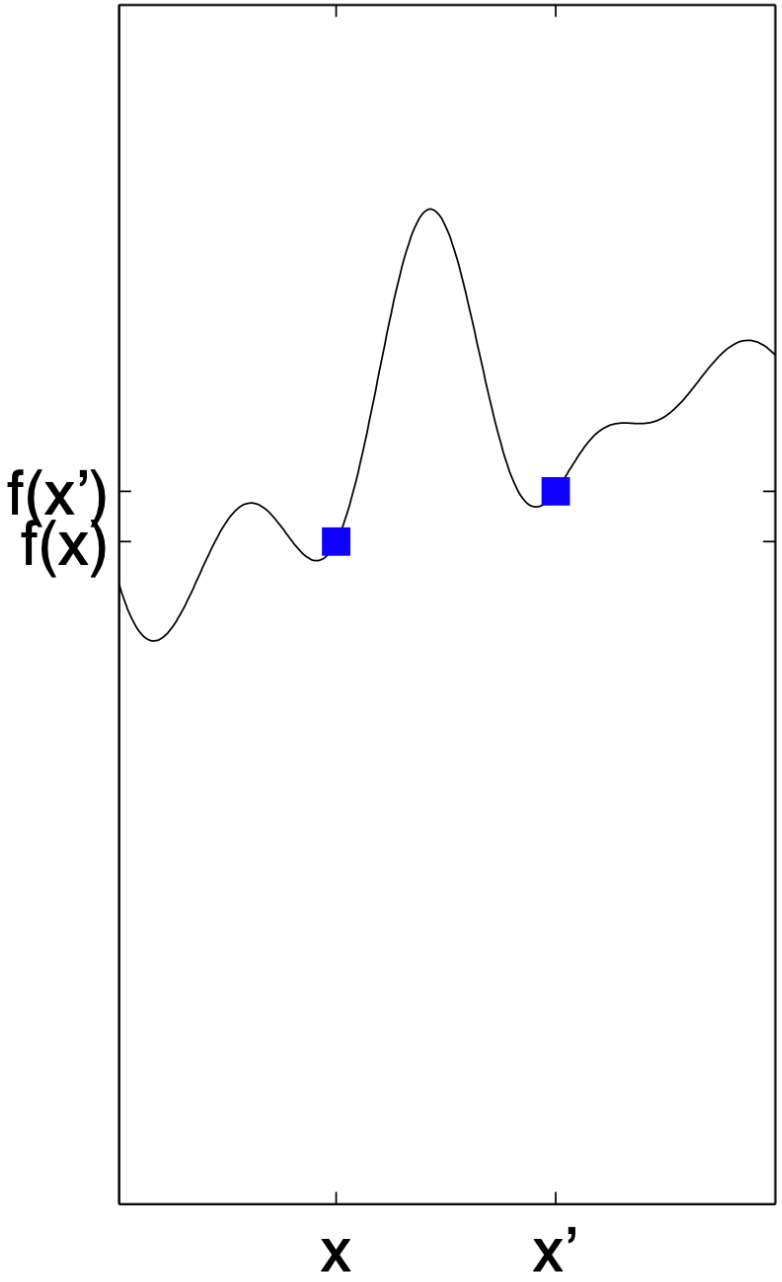


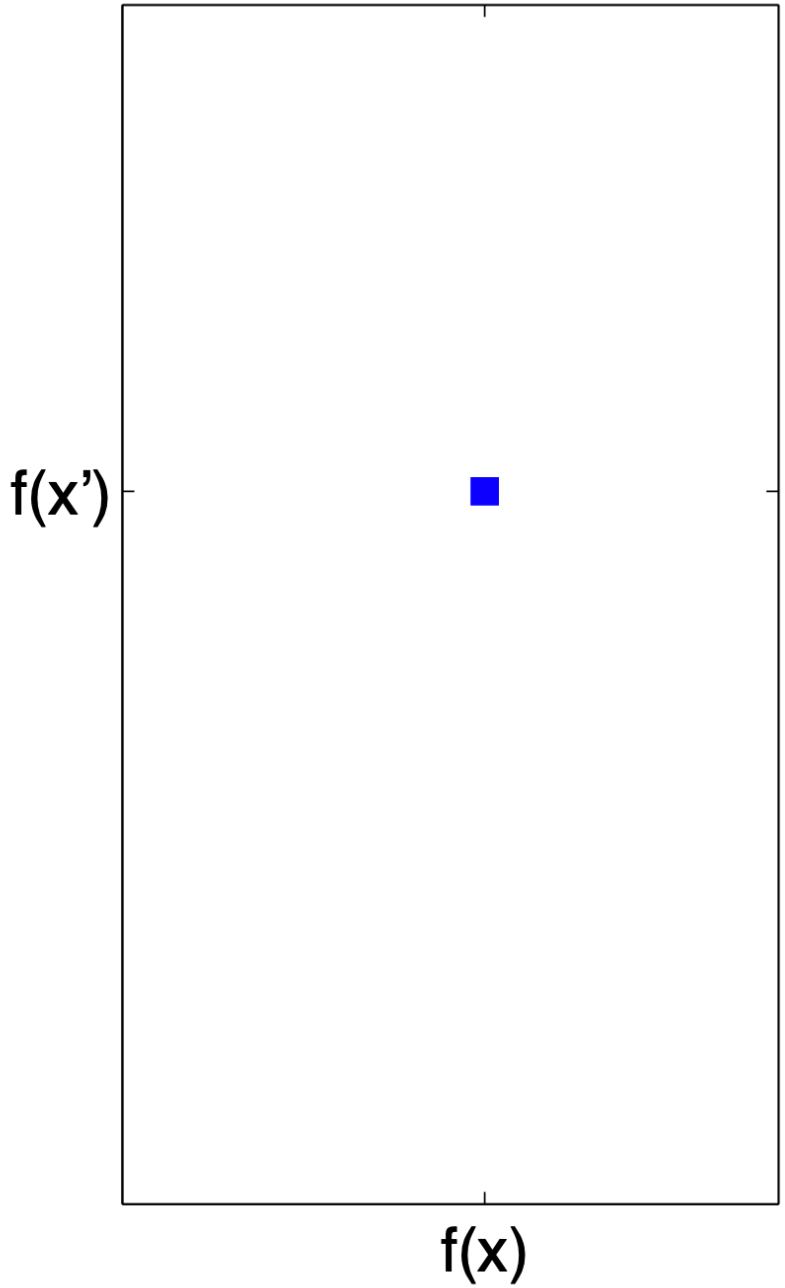
P. Frazier, INFORMS Tutorial, 2018





P. Frazier, INFORMS Tutorial, 2018





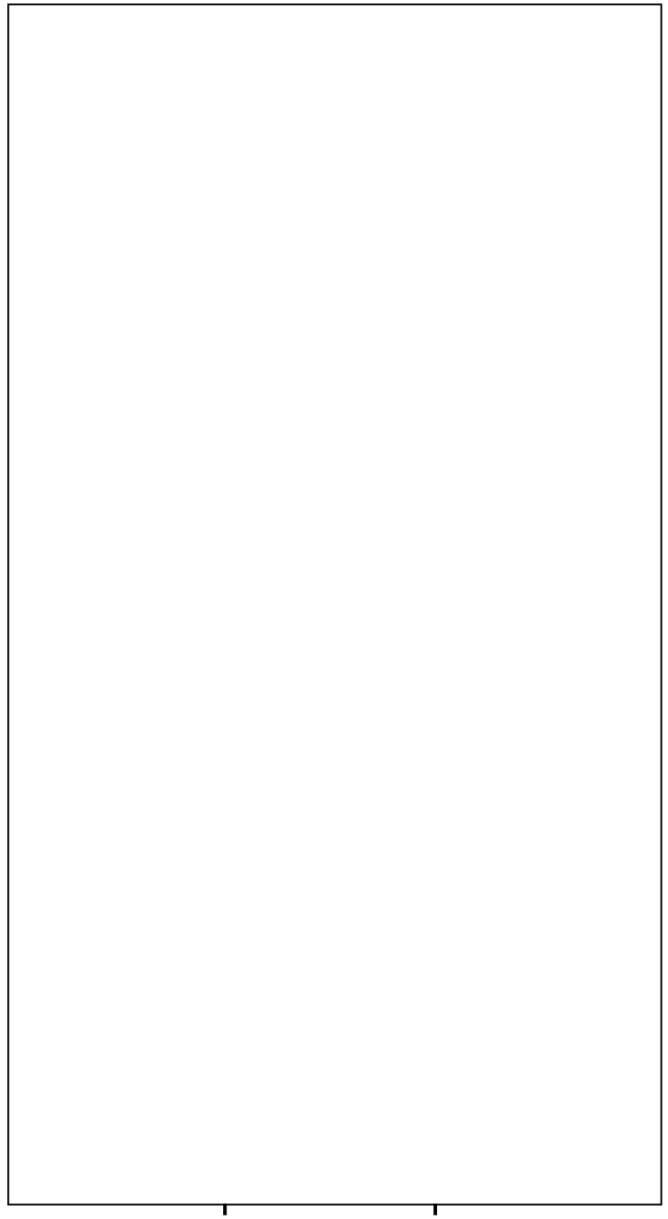
P. Frazier, INFORMS Tutorial, 2018

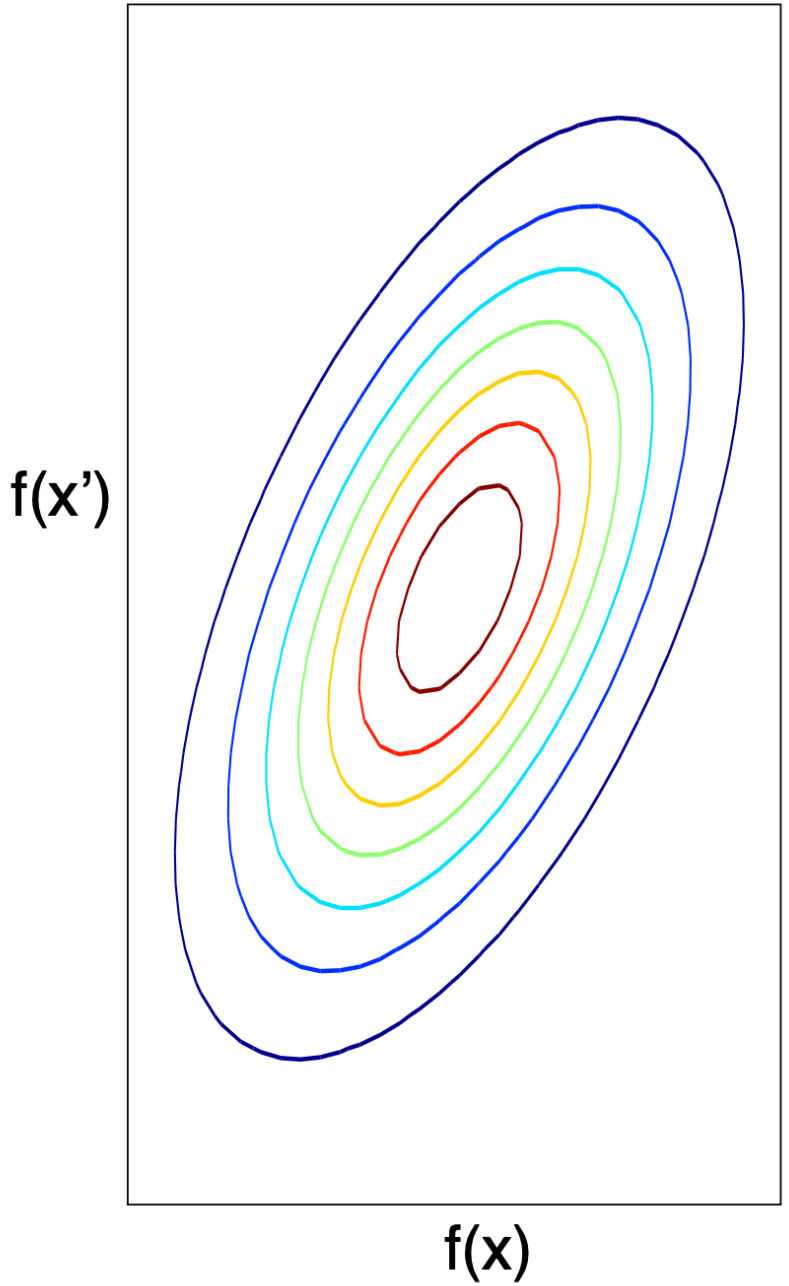
Let's Place a Multivariate Normal Prior on (f(x), f(x'))

$$\begin{bmatrix} f(x) \\ f(x') \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_0(x) \\ \mu_0(x'), \end{bmatrix}, \begin{bmatrix} k_0(x,x) & k_0(x,x') \\ k_0(x',x) & k_0(x',x') \end{bmatrix} \right)$$

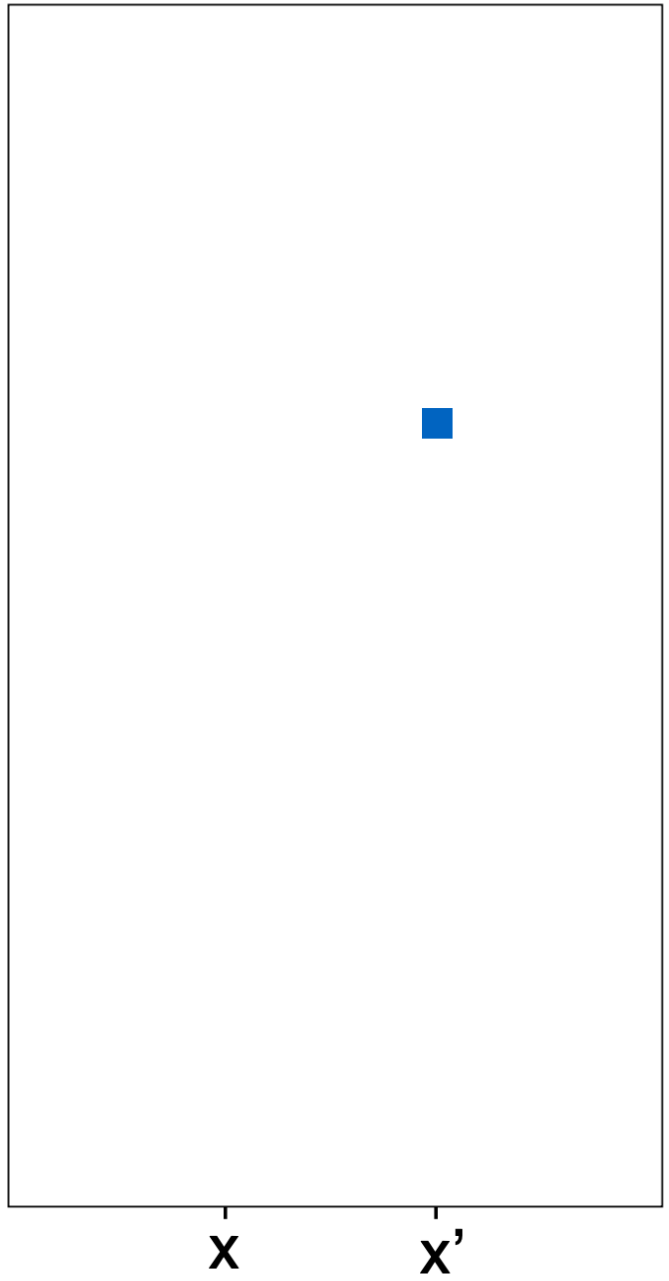
- $\mu_0 : \Omega \to \mathbb{R}$ is prior mean function
  - can depend on input features, often set to a constant (zero)

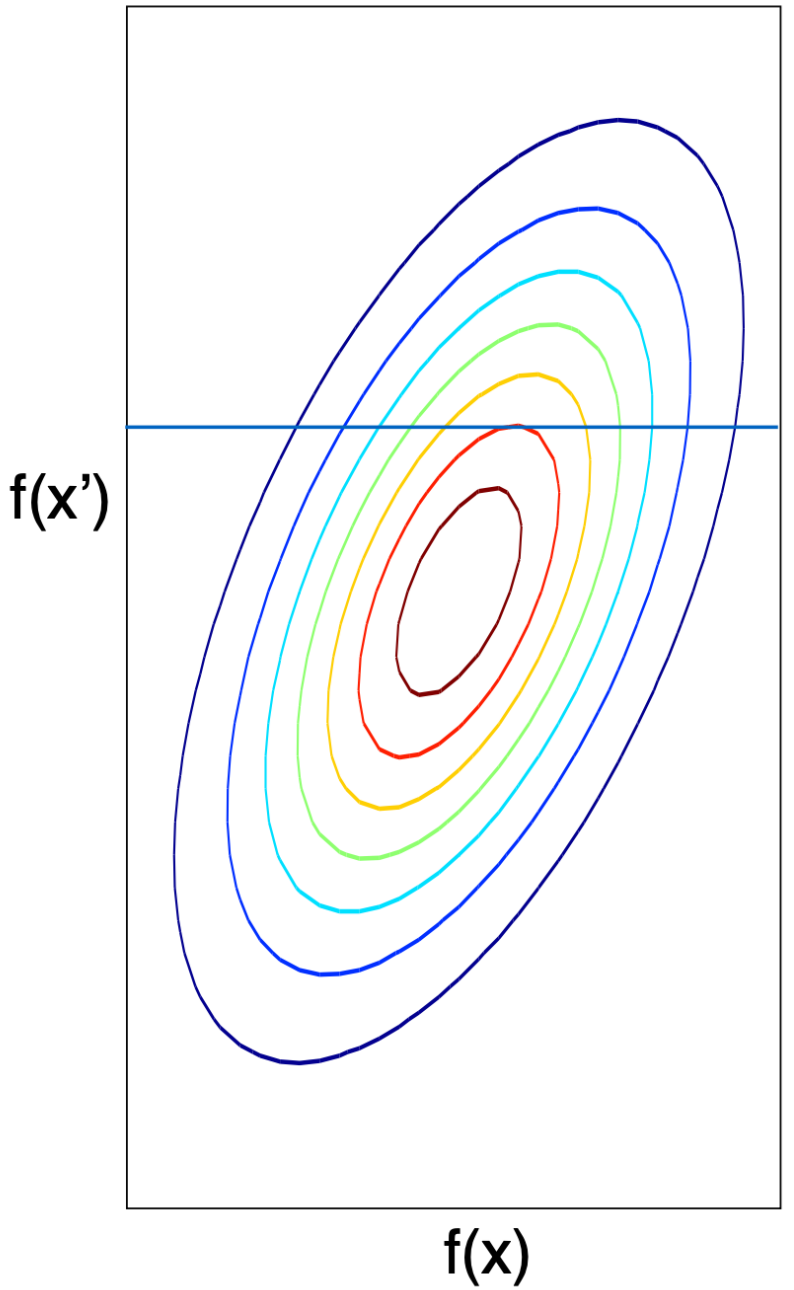
- $k_0 : \Omega \times \Omega \to \mathbb{R}$ is prior covariance function
  - must be a function of two input features x and x'
  - Often decreases with $\|x - x'\|$ → less correlation when inputs far apart



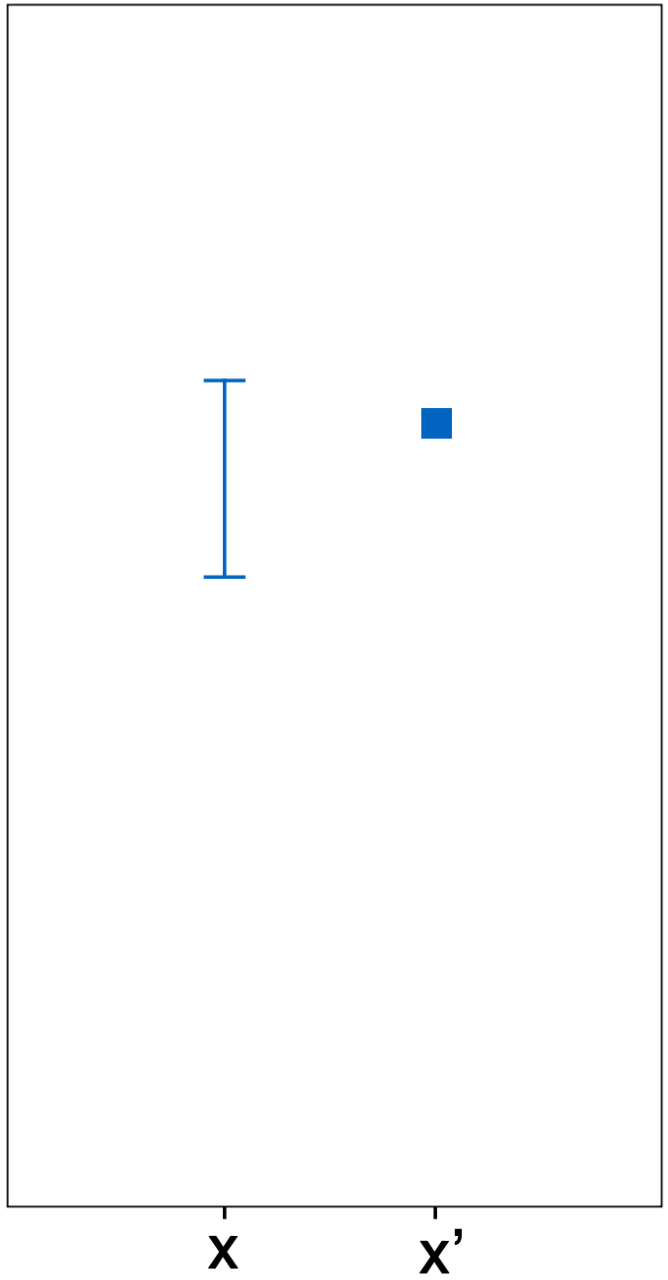


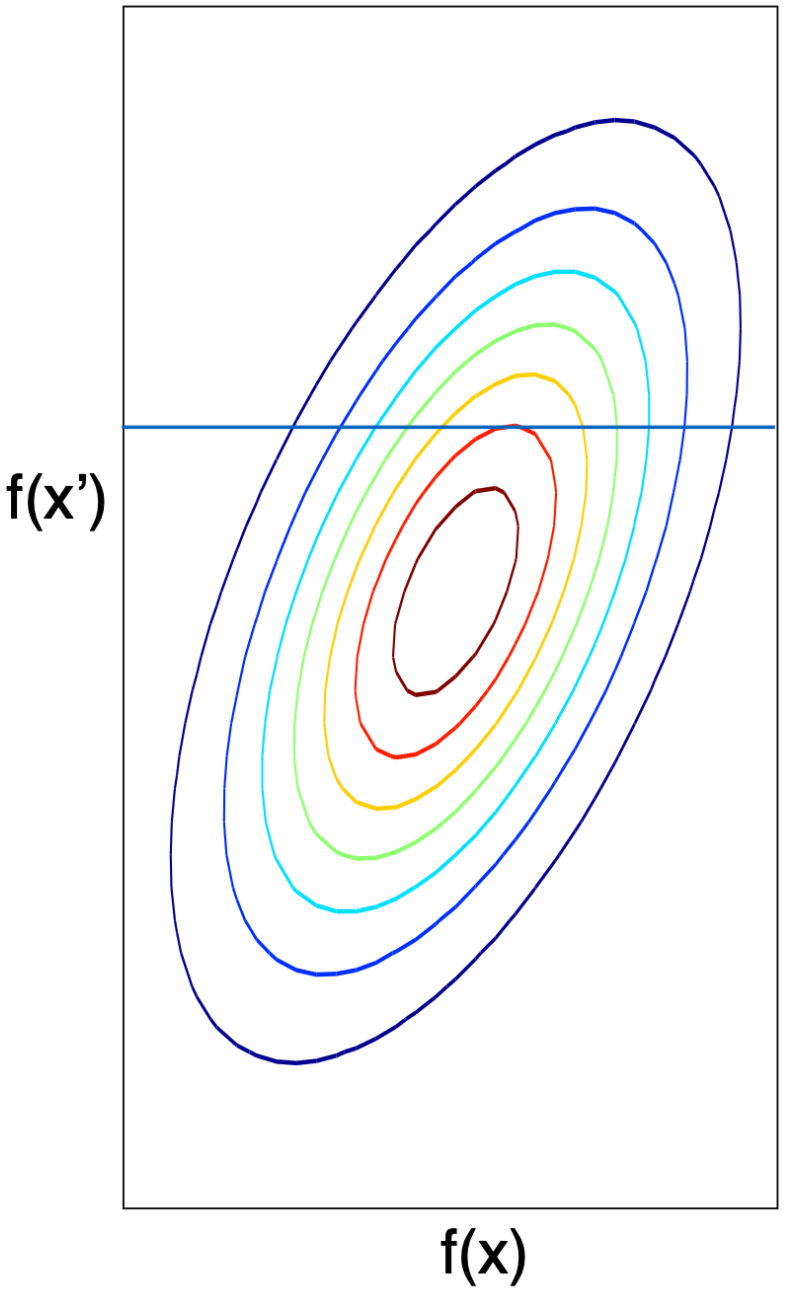
P. Frazier, INFORMS Tutorial, 2018



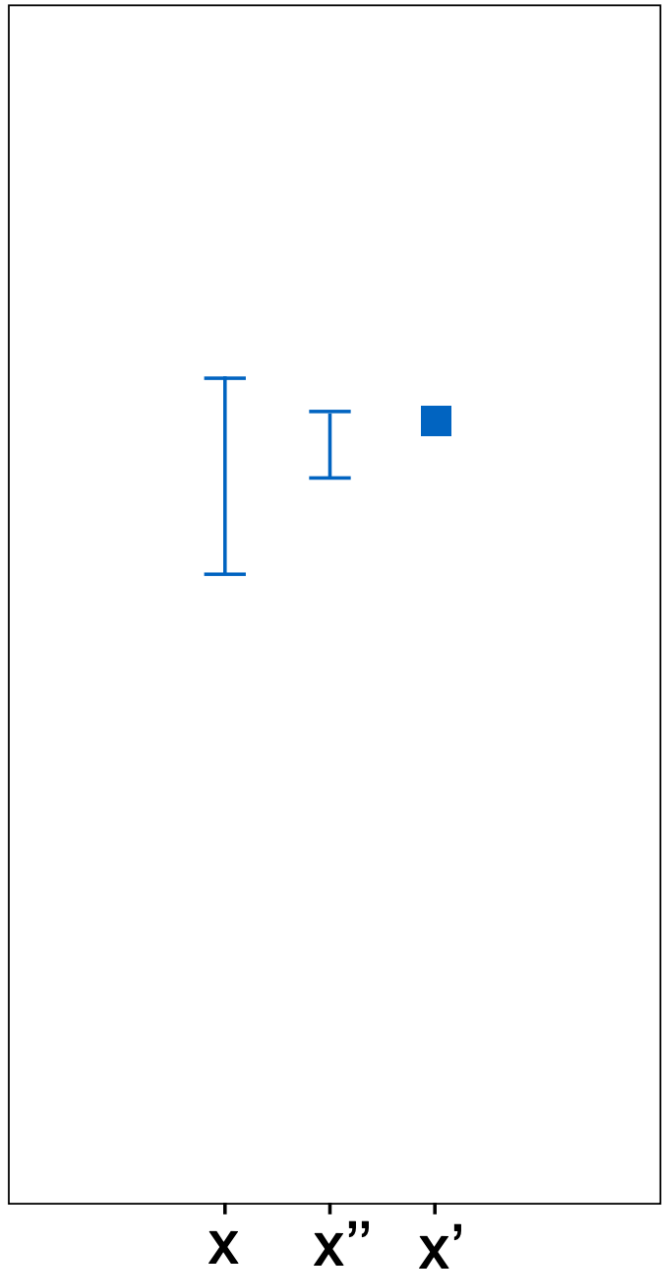


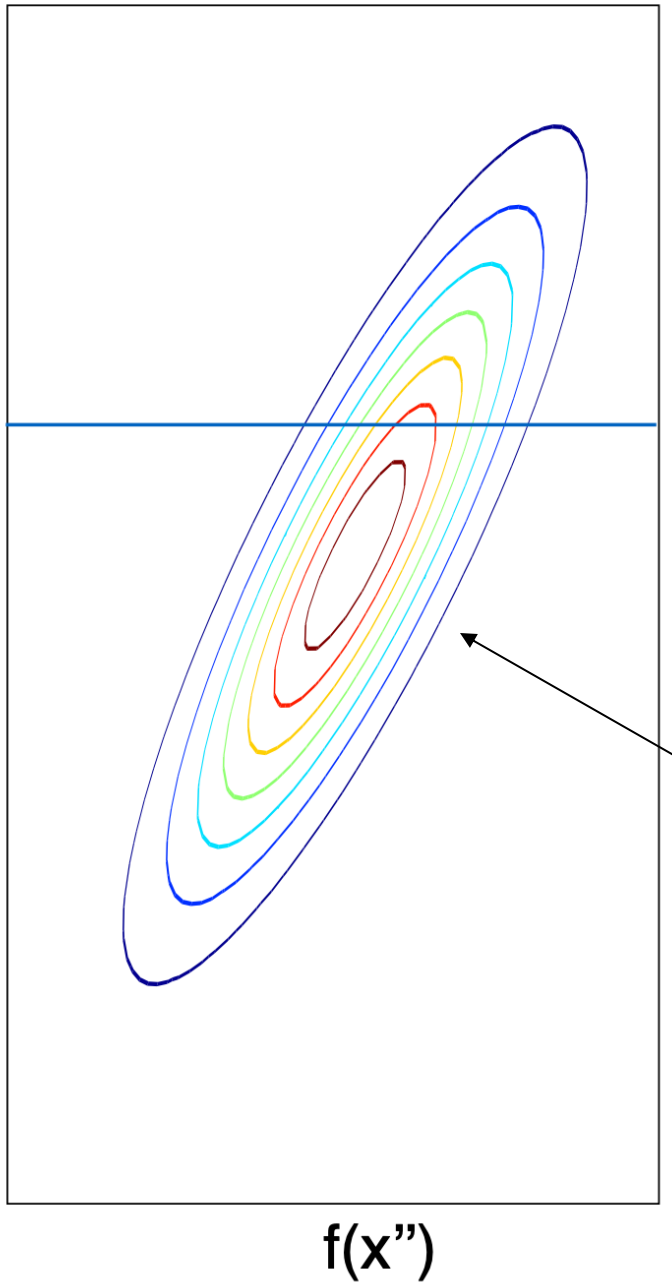
P. Frazier, INFORMS Tutorial, 2018





P. Frazier, INFORMS Tutorial, 2018





Notice how the covariance “shrinks” as x and x' get closer together

P. Frazier, INFORMS Tutorial, 2018

Gaussian Process (GP) Prior

- A prior on an unknown function f is a Gaussian process prior if, for any collection of points $x_1, \ldots, x_k$, we have

$$\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_k) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_0(x_1) \\ \vdots \\ \mu_0(x_k), \end{bmatrix}, \begin{bmatrix} k_0(x_1, x_1) & \cdots & k_0(x_1, x_k) \\ \vdots & \ddots & \vdots \\ k_0(x_k, x_1) & \cdots & k_0(x_k, x_k) \end{bmatrix} \right)$$

- We call μ_0 the "mean function"

- We call k_0 the "kernel" or "covariance function"

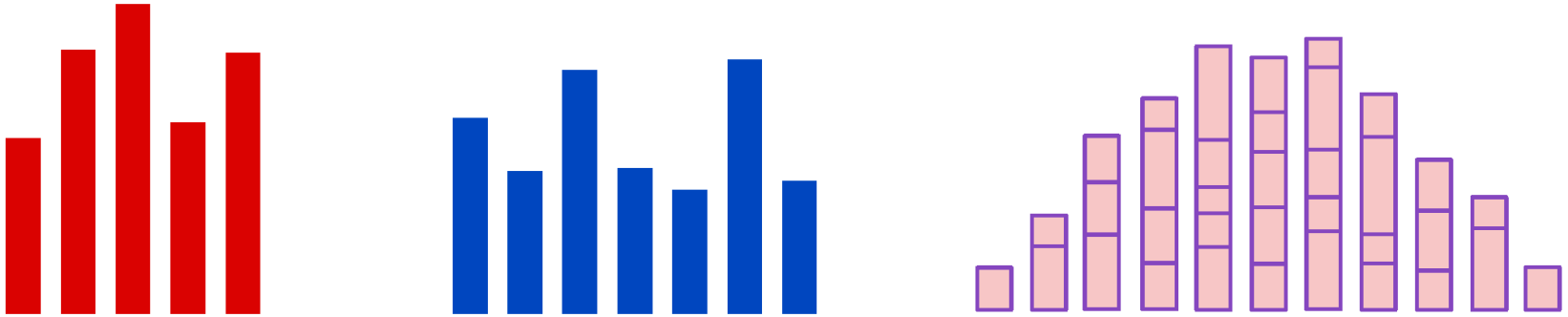
Before proceeding further with GPs, let's discuss a bit more about the important properties of Gaussian random variables

Why Gaussian Distributions? 1) Central Limit Theorem

- Let us assume that the phenomenon we want to model is made up of multiple sources of uncertainty, which are independent and possibly of different distributions, added together

$X \sim p_X(x)$ $Y \sim p_Y(y)$ $Z = X + Y$

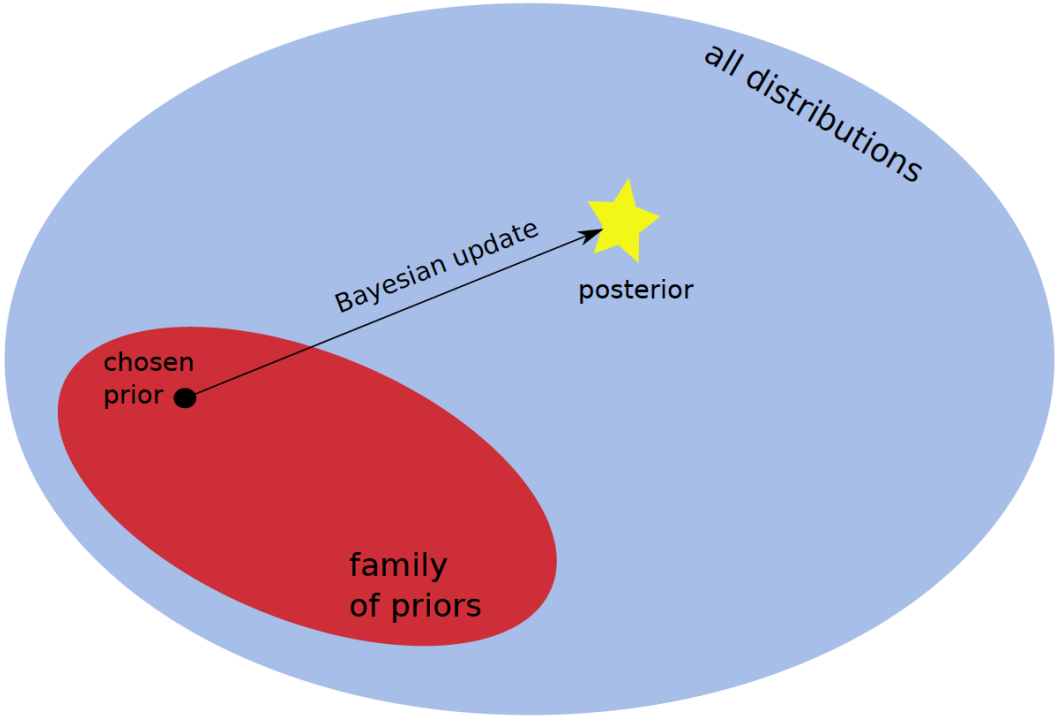
$Z \sim p_Z(z) = \sum_x p_Y(z - x)p_X(x)$



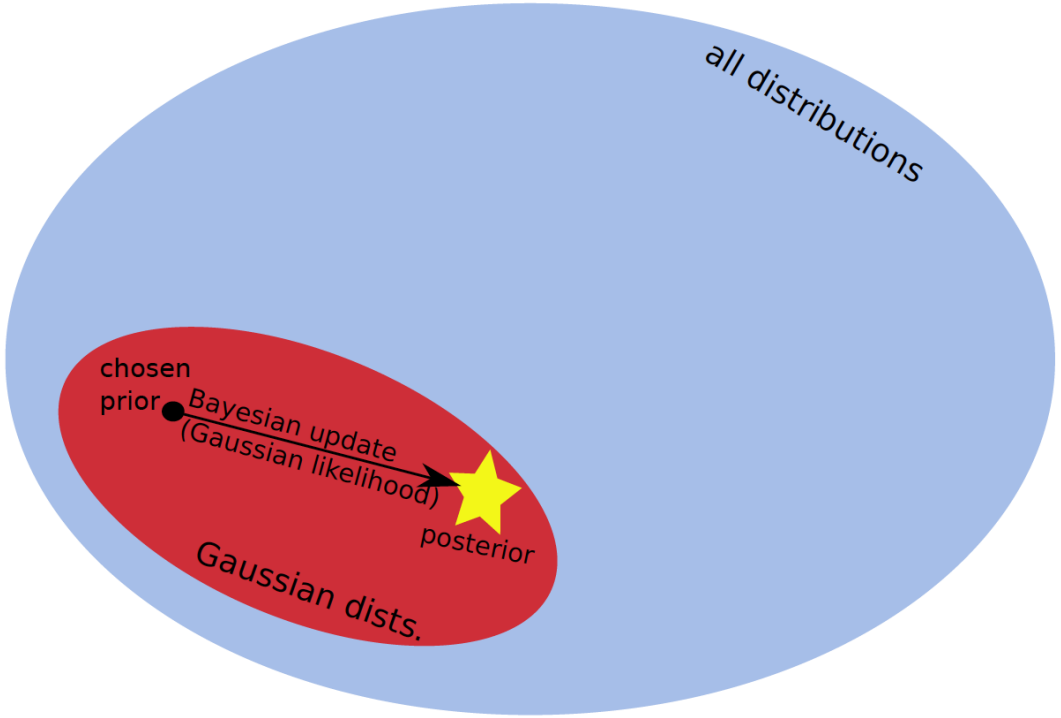
- Density of sum of two random variables is given by convolution
- Gaussian distribution is the "fixed point" (limit) of this operation → more sources of uncertainty added together leads to *more Gaussian* result

Why Gaussian Distributions? 2) Conjugacy

General Bayesian update



Conjugate Bayesian update



$$p(\theta|\text{Data}) = \frac{p(\text{Data}|\theta)p(\theta)}{p(\text{Data})}$$

The Gaussian Law (Linear Transformation)

- Gaussian distributions are closed under linear transformations:

$$X \sim \mathcal{N}(\mu_X, \Sigma_X) \quad \Rightarrow \quad Y = AX + b \sim \mathcal{N}(A\mu_X + b, A\Sigma_X A^\top)$$

- An immediate consequence is any n-dimensional Gaussian random vector can be produced from $\mathcal{N}(0, I_n)$ (independent Gaussian RVs)

- This is often known as the "whitening transformation" and can be expressed in terms of the Cholesky decomposition $\Sigma_X = LL^\top$:

$$X = \mu_X + L\mathcal{N}(0, I) \sim \mathcal{N}(\mu_X, \Sigma_X)$$

Marginal of Gaussian

- Assume that we have two random vectors that are jointly Gaussian, so that they have the following distribution

$$(\mathbf{x}, \mathbf{y}) \sim \mathcal{N}\left(\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix}, \begin{bmatrix} A & C \\ C^\top & B \end{bmatrix}\right)$$

- What is the distribution of the random variable $\mathbf{x}$?

$$\mathbf{x} \sim \mathcal{N}(\mathbf{a}, A)$$

Prove by treating marginalization as projection operation: $\mathbf{x} = \begin{bmatrix} I & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}$

Conditional of Gaussian

- It turns out any conditional of a Gaussian distribution is also Gaussian

$$(\mathbf{x}, \mathbf{y}) \sim \mathcal{N}\left(\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix}, \begin{bmatrix} A & C \\ C^\top & B \end{bmatrix}\right)$$

$$\mathbf{x}|\mathbf{y} \sim \mathcal{N}(\mathbf{a} + CB^{-1}(\mathbf{y} - \mathbf{b}), A - CB^{-1}C^\top)$$

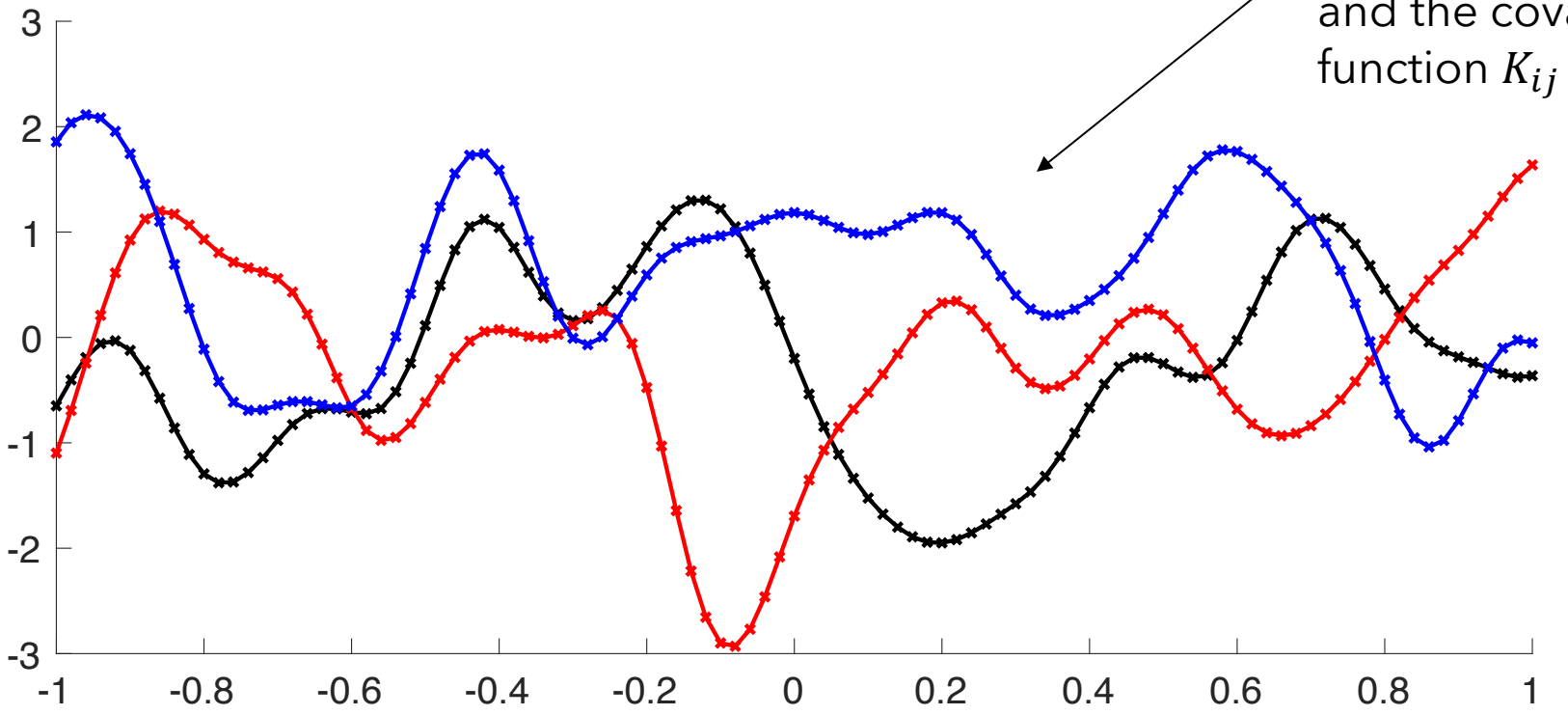
Showing this result is not as straightforward, but is a standard result that can be looked up / proved in several textbooks

Now back to GPs

Sampling From a Gaussian Process

- The complete GP is infinite-dimensional; however, we can draw finite parts of a GP sample using the marginalization property of GPs

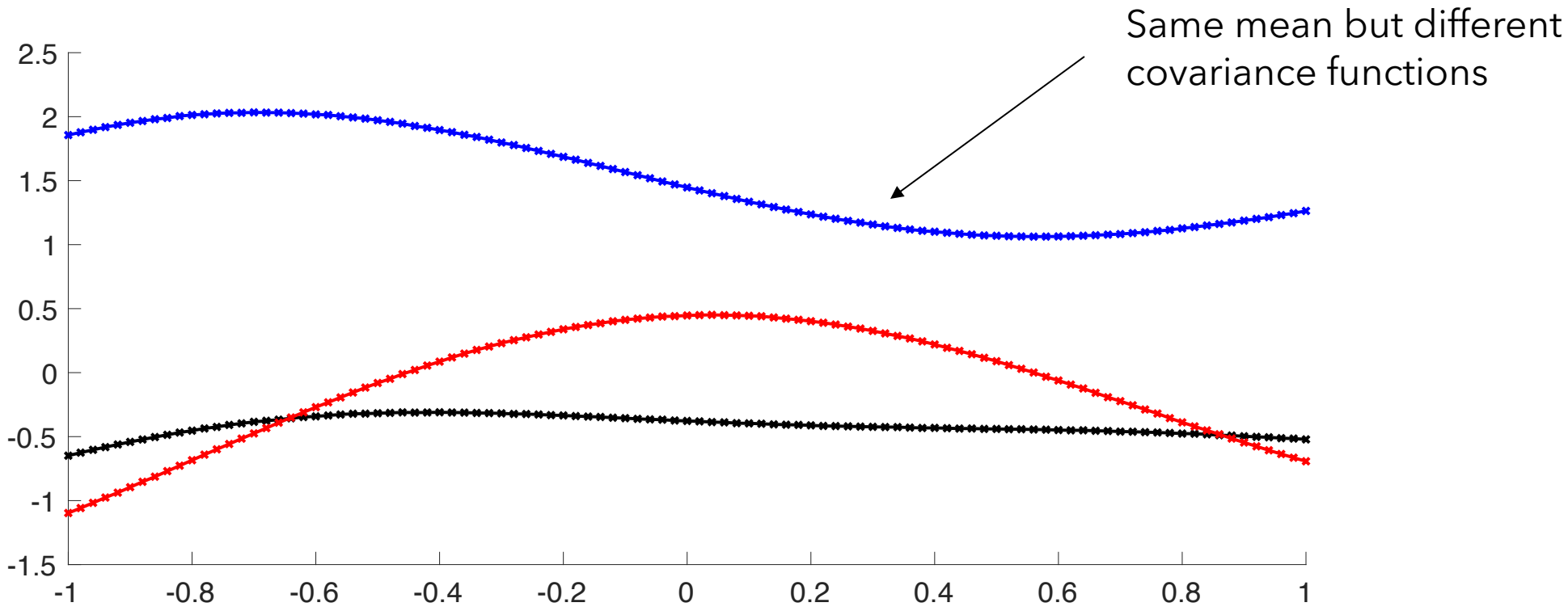
Need to select mean
(in this case zeros(100,1))
and the covariance
function $K_{ij} = k(x_i, x_j)$



- Here are 3 realizations on a 100-dimensional grid, drawn from a GP

Sampling From a Gaussian Process

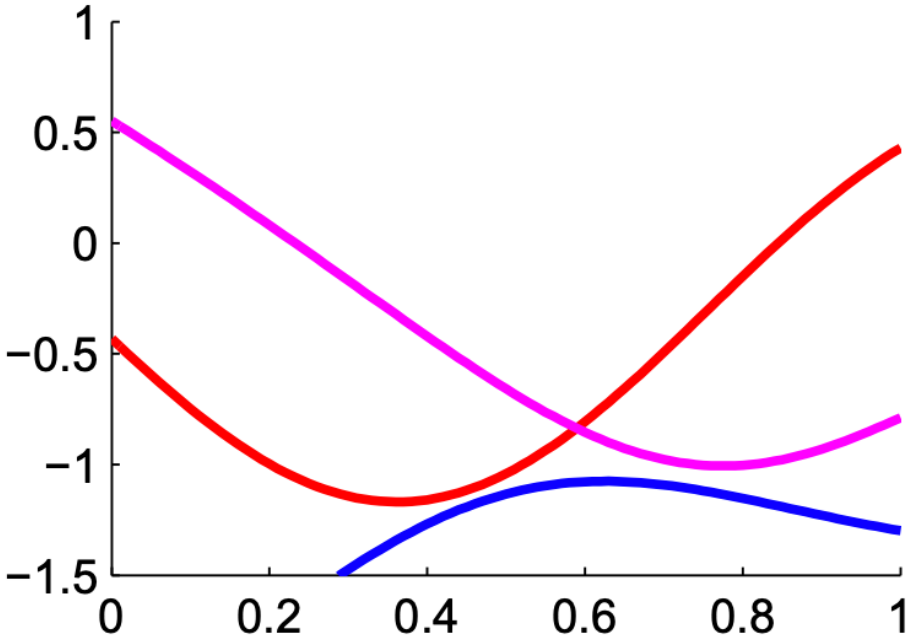
- The complete GP is infinite-dimensional; however, we can draw finite parts of a GP sample using the marginalization property of GPs



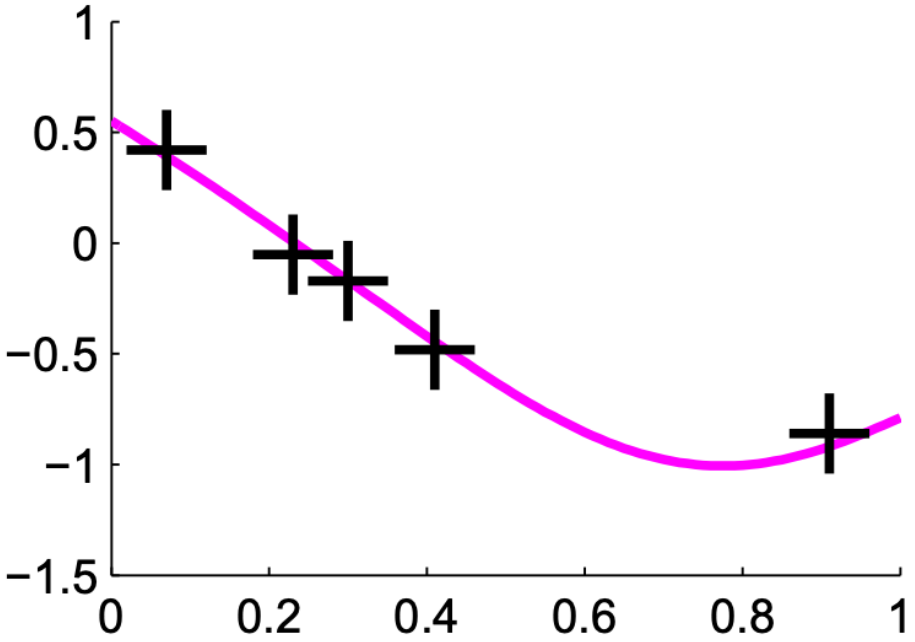
- Here are 3 realizations on a 100-dimensional grid, drawn from a GP

GP Regression Model

GP Prior



Noisy Observations



$$f \sim \mathcal{GP}(0, k(x, x'))$$

$$\mathbf{f} \sim \mathcal{N}(0, K), \quad K_{ij} = k(x_i, x_j)$$

$$y_i | f_i \sim \mathcal{N}(f_i, \sigma^2)$$

[Independent, additive Gaussian noise]

We Can Compute GP Posterior Analytically!

- The posterior on f(x′) (at a test point) given n past observations $y_1,\ldots,y_n$ is normal with mean $\mu_n(x')$ and variance $\sigma_n^2(x')$*:

$$\mu_n(x') = k_0(x', x_{1:n}) \left(k_0(x_{1:n}, x_{1:n}) + \sigma^2 I_n\right)^{-1} y_{1:n}$$

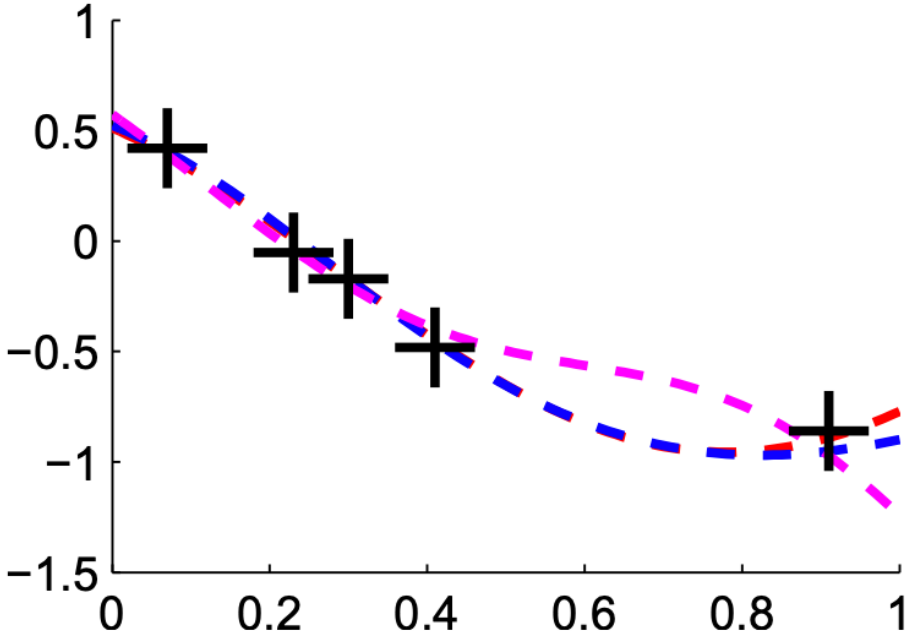
$$\sigma_n^2(x') = k_0(x', x') - k(x', x_{1:n}) \left(k(x_{1:n}, x_{1:n}) + \sigma^2 I_n\right)^{-1} k(x_{1:n}, x')$$

- Formula assumes $\mu_0(x) = 0$ but can be easily modified for general μ_0

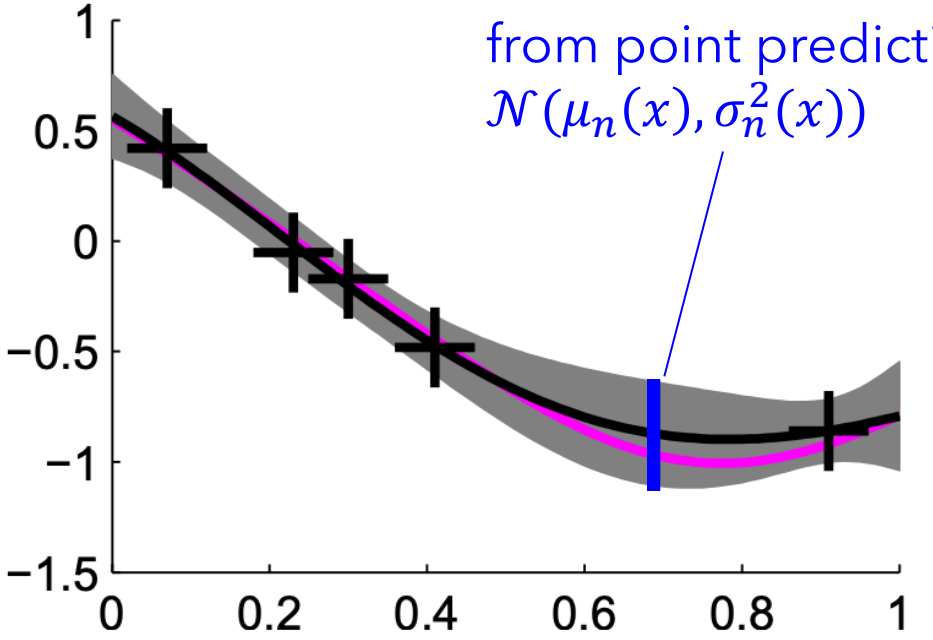
*Can be derived from Gaussian conditional formula

GP Posterior

Two (incomplete) ways of visualizing what we have learned:



Draws $\sim p(\mathbf{f}|\text{data})$

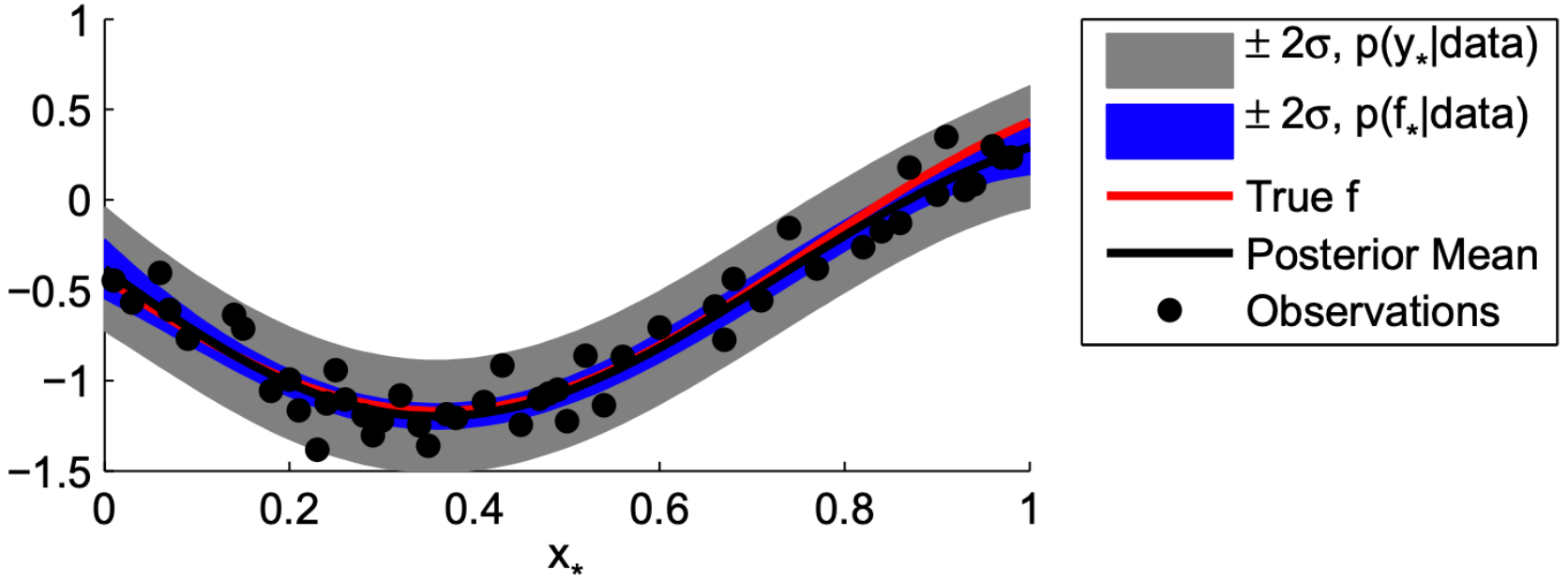


Error bars are constructed from point predictions $\mathcal{N}(\mu_n(x), \sigma_n^2(x))$

Mean and error bars

Discovery or Prediction?

- What should the error bars show?



- $f_*|y_{1:n} \sim \mathcal{N}(\mu_n(x), \sigma_n^2(x))$ says what we know about noiseless function
- $y_*|y_{1:n} \sim \mathcal{N}\left(\mu_n(x), \sigma_n^2(x)\right)$ predicts what we will see next at point x_*

Let's Review What We Have Learned So Far

- We can represent a function as a big (infinite) vector $\mathbf{f}$

- We assume this unknown vector was drawn from a big (infinite) correlated Gaussian distribution, which we call a Gaussian process
  - This is why GPs are often called "non-parametric" since they cannot be represented by a finite set of parameters

- Observing elements of function (optionally corrupted by i.i.d. Gaussian noise) creates a posterior distribution that is also a GP (conjugacy)

- From marginalization principle, we can easily ignore all positions x_i that are neither observed nor queried

Covariance Functions

- The main part that has been missing in our discussion so far is where does the covariance function (kernel) $k(x, x')$ come from?

- Also, other than making nearby points covary, what can we express with covariance functions and what do they mean?

- It turns out that the choice of kernel is extremely flexible and in fact be can selected in a way to represent most of the common model classes (including linear and neural networks)

- Let's briefly go over the general procedure and then discuss examples

General Procedure:

How Should We Choose the Mean & Kernel Functions?

1. Designate a parametric family

2. Take an initial set of samples from points sampled uniformly at random (information gathering step)

3. Choose hyperparameters using either:
   a) Maximum likelihood estimation

   b) Maximum a posteriori estimation (prior on hyperparameters)

   c) Sample hyperparmeters from their posterior and average

Simplest Kernel Derived from Bayesian Linear Regression

- We can directly construct kernels from parametric models

- Simplest example: **Bayesian linear regression**:

$$f(x_i) = w^\top x + b, \quad w \sim \mathcal{N}(0, \sigma_w^2 I), \quad b \sim \mathcal{N}(0, \sigma_b^2)$$

$$\begin{aligned}\mathrm{Cov}(f_i, f_j) &= \langle f_i f_j \rangle - \langle f_i \rangle \langle f_j \rangle \\ &= \langle (w^\top x_i + b)(w^\top x_j + b) \rangle \\ &= \sigma_w^2 x_i^\top x_j + \sigma_b^2 = k(x_i, x_j)\end{aligned}$$

- Kernel parameters are σ_w^2 and σ_b^2 (hyperparameters in Bayesian model)

- Most interesting kernels come from models with large/infinite feature space. Because weights w are integrated out, not more computationally expensive

Squared-Exponential (SE) Kernel

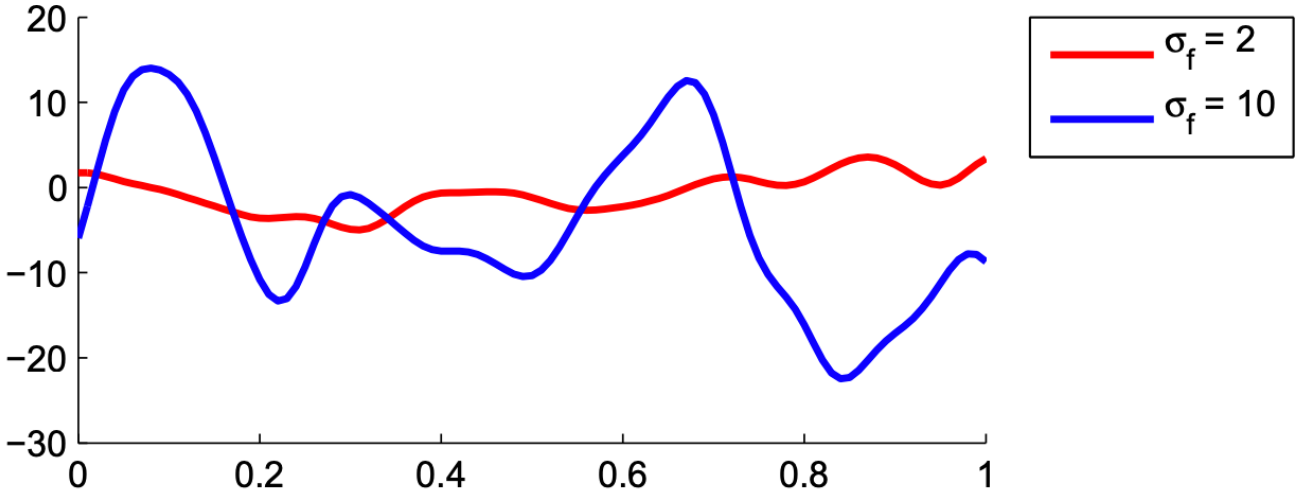
- If we do the same analysis as before with an infinite number of radial basis functions, then we will derive:

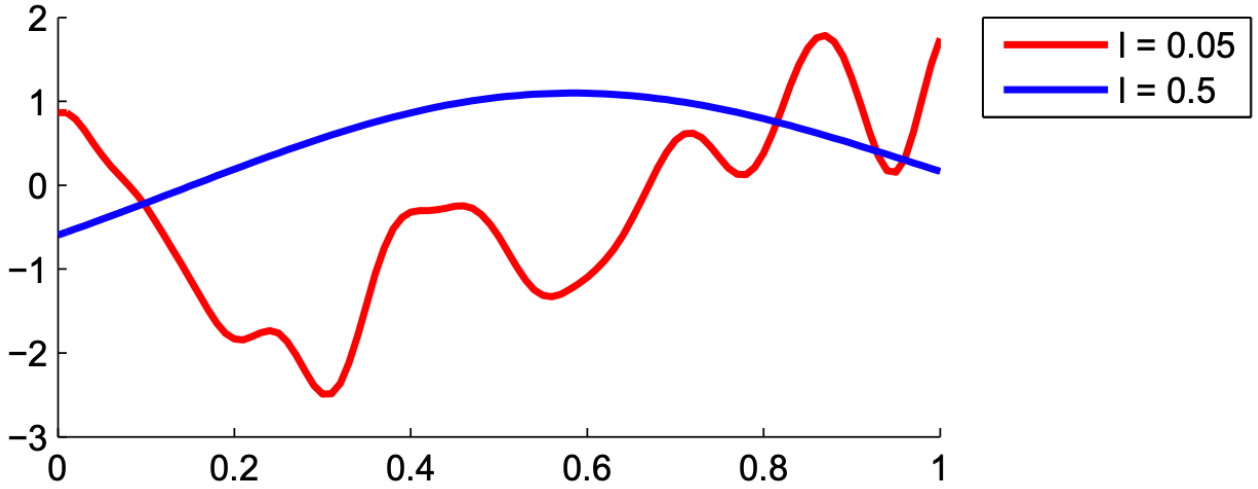
$$k(x, x') = \sigma_f^2 \exp\left[-\frac{1}{2}\sum_{d=1}^{D}\frac{(x_d - x'_d)^2}{l_d^2}\right]$$

- This is the most commonly-used kernel in machine learning
  (not always best choice since it assumes function infinitely differentiable)

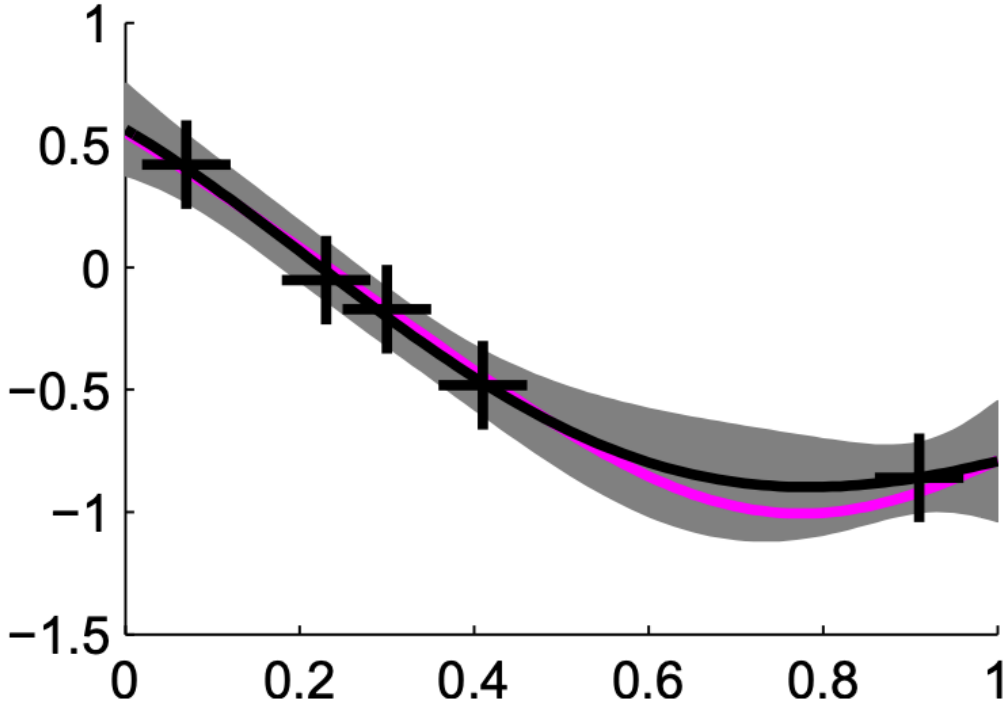
- The hyperparameters are (many kernels have these types):
  - σ_f^2 which controls the marginal function variance (scale)
  - l_d in each dimension controls the correlation length (distance between peaks)

Squared-Exponential (SE) Kernel

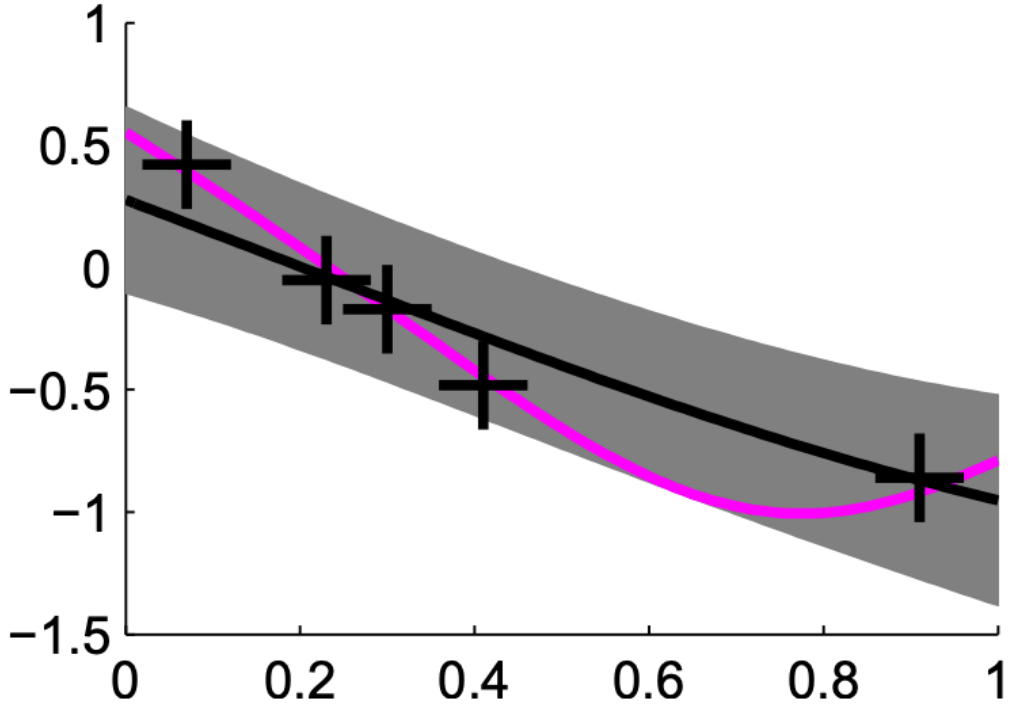




Effect of Hyperparameters: Give Different Explanation of Data



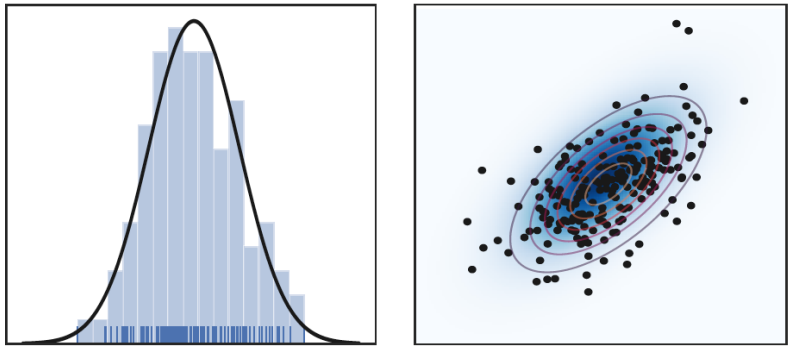
$l = 0.5,\ \sigma = 0.05$



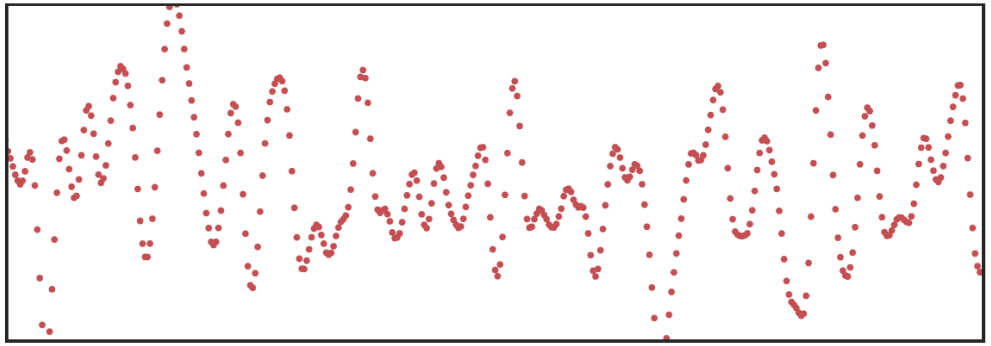
$l = 1.5,\ \sigma = 0.15$

Why Do We Have to Parametrize Kernel?

Finite Gaussians: # samples >> parameter dimension



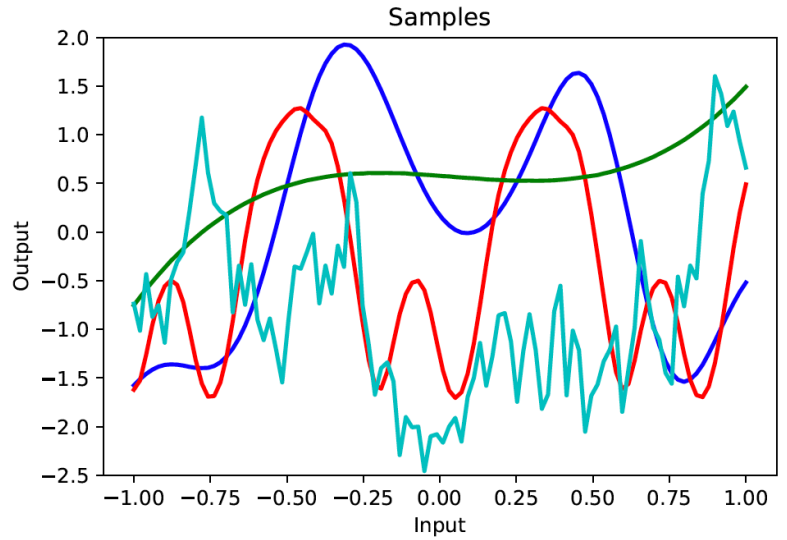
Gaussian process: # samples << 1 << param dimension

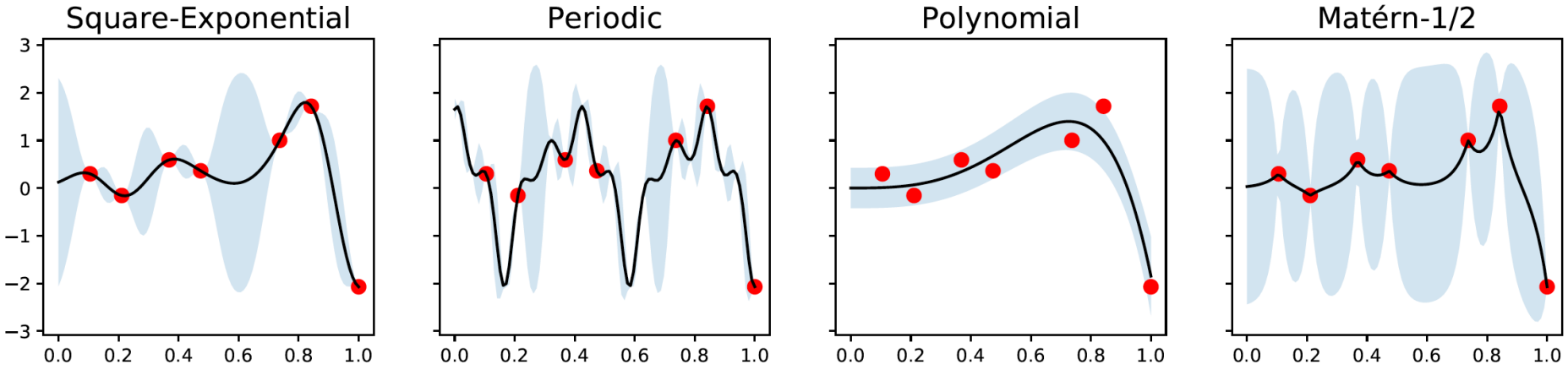


- Learning finite Gaussians in d dimensions requires many observations to fit $d + \frac{d(d+1)}{2}$ parameters (usually an **overdetermined** problem)
- With GPs, we do not even have one full observation of the model, but only parts of it. Therefore, learning a GP is inherently an **underdetermined** problem
- To learn a GP, we must *tie* the values of mean and covariance to reduce their degrees of freedom → cannot freely assign covariance to each pair (x, y) and instead choose a function $k_\theta(x, y)$ and compare different values of θ

Important: Different Kernels, Different Predictions

- Square-Exponential: $k_{\mathbf{SE}}(x, x') = \sigma^2 \exp\left(-\frac{(x-x')^2}{2l^2}\right)$
- Periodic: $k_{\mathbf{Per}}(x, x') = \sigma^2 \exp\left(-\frac{2\sin(\pi|x-x'|/p)}{l^2}\right)$
- Polynomial: $k_{\mathbf{Pol}}(x, x') = (x^\top x' + c)^d$
- Matérn-1/2: $k_{\mathbf{M}}(x, x') = \sigma^2 \exp\left(-\frac{|x-x'|}{2l^2}\right)$





Lincoln et al. The Art of the Gaussian Process

Stationary Kernels

- A stationary kernel is a function of the difference of its inputs

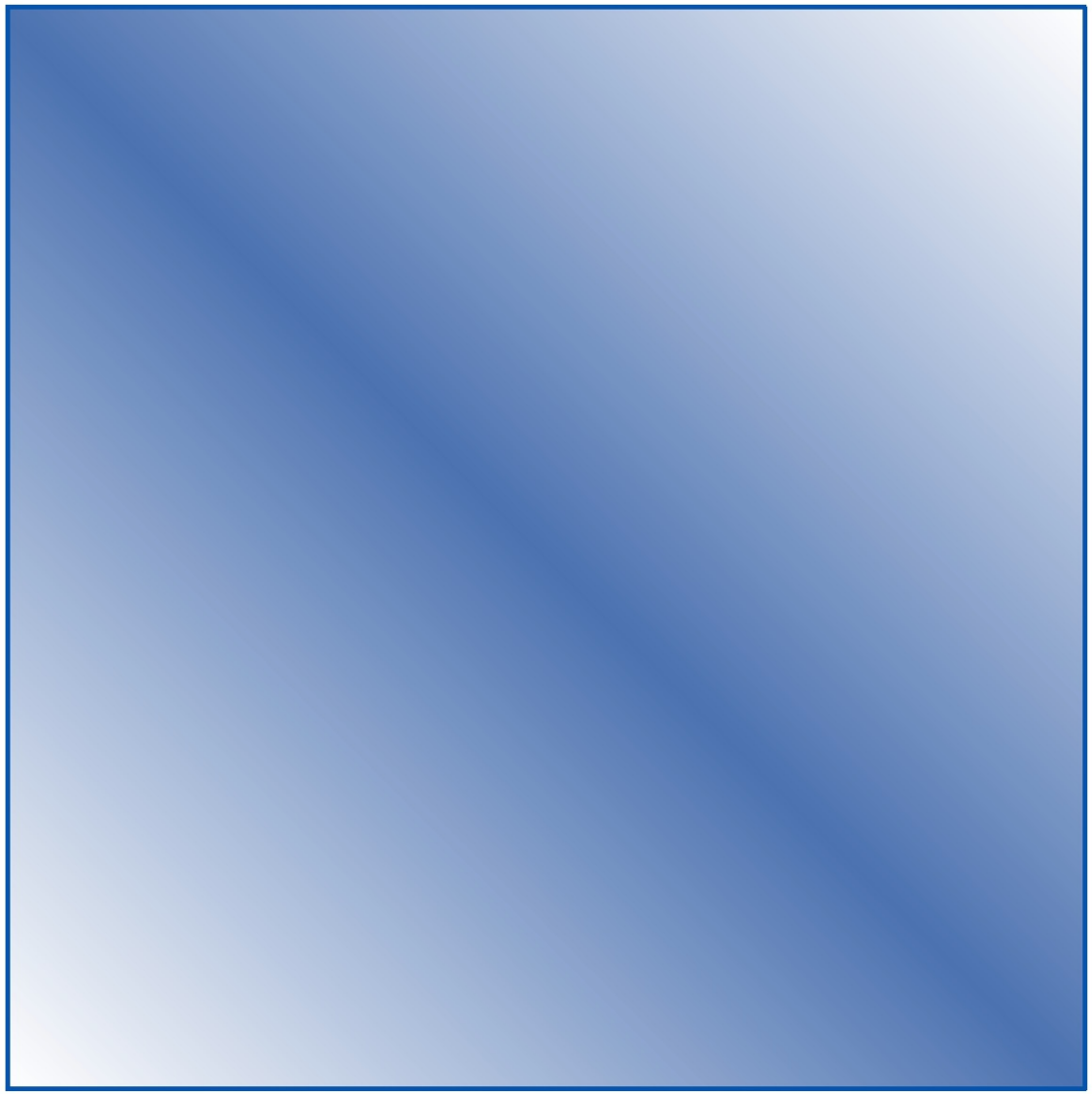
$$k_\theta(x, x') = k_\theta(x - x')$$

- Observation 1: GPs with stationary kernels are invariant under translations

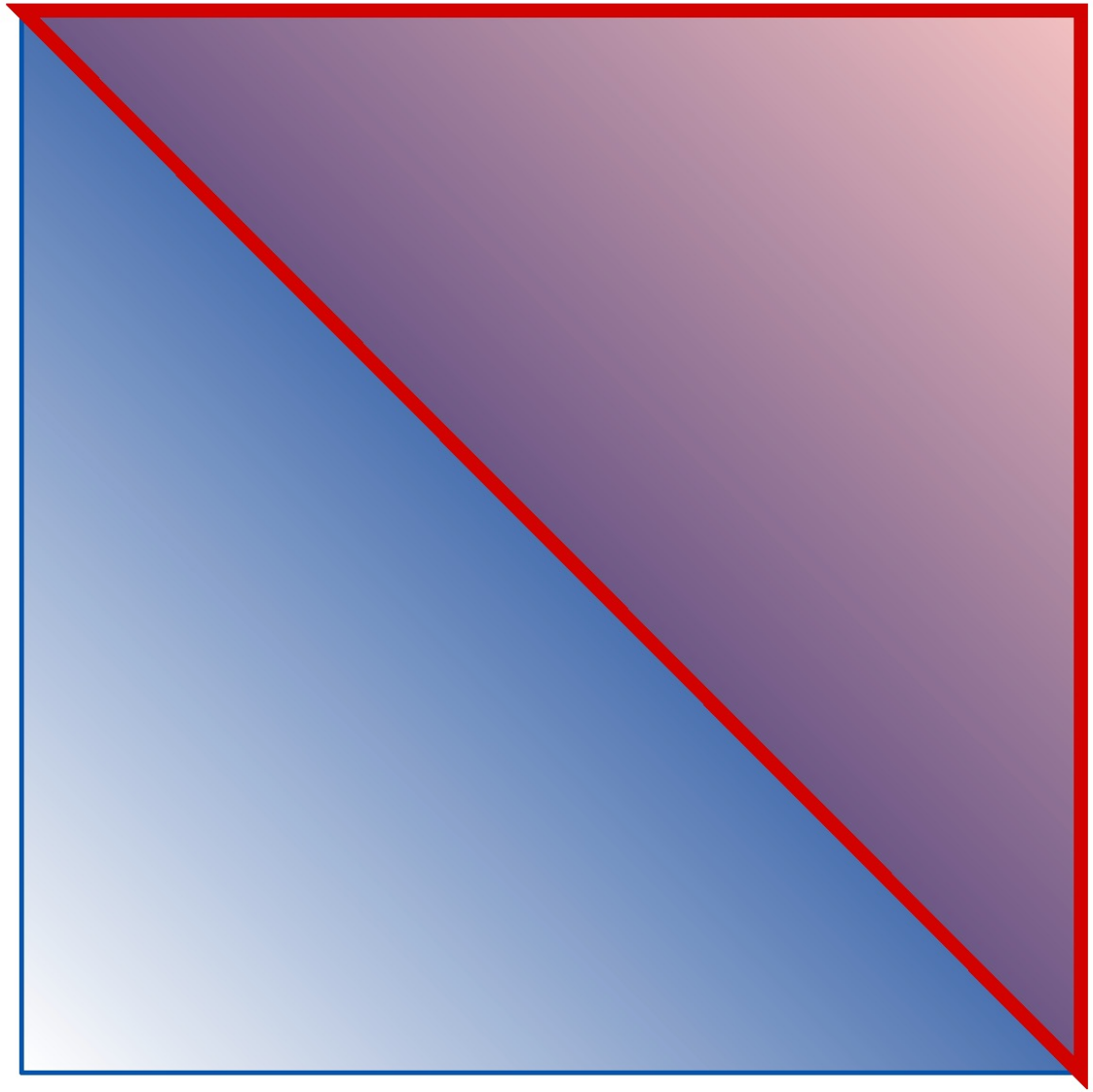
$$k_\theta(x, x') = k_\theta(x - \delta, x' - \delta), \qquad \forall \delta \in \mathbb{R}$$

- Observation 2: The assumption of stationarity dramatically reduces the design of a kernel → helpful modeling assumption (but be careful!)

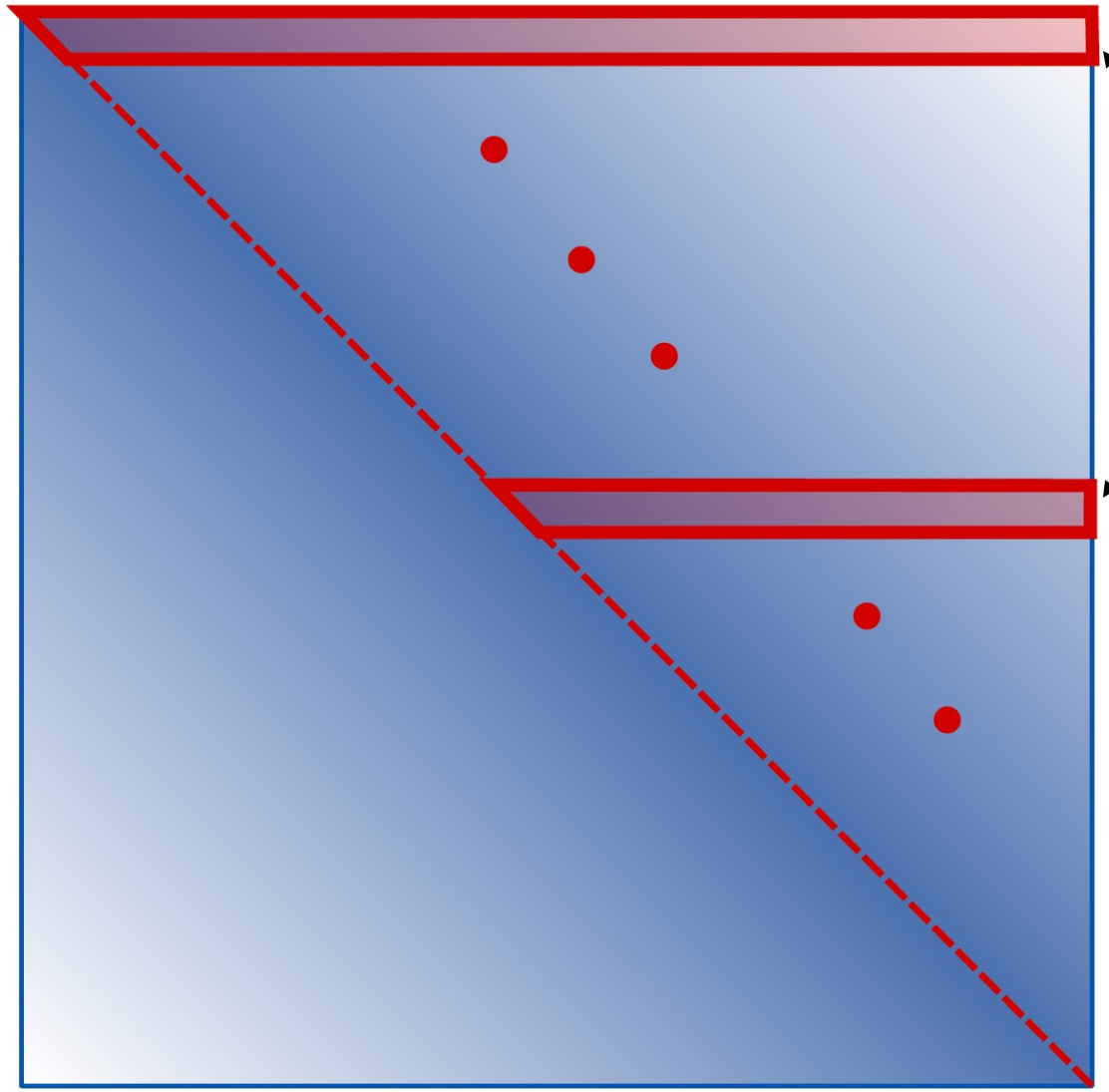
Stationary Kernels



Stationary Kernels



Stationary Kernels



Knowing one "slice" of the kernel allows me to populate the rest of the kernel, so dimensionality has been drastically reduced

Learning: Fitting GP to Data using Maximum Likelihood

- The probability of the observed data $\mathbf{y}$ is just a Gaussian

- Therefore, we can select the hyperparameters by maximizing the likelihood (same as minimizing the negative log-likelihood function)

$$-\log p(\mathbf{y} \mid \theta) = \underbrace{\frac{1}{2}(\mathbf{y}-\boldsymbol{\mu})^\top \Sigma_{\mathbf{y}}^{-1}(\mathbf{y}-\boldsymbol{\mu})}_{\text{Data fit term}} + \underbrace{\frac{1}{2}\log(|\Sigma_{\mathbf{y}}|)}_{\text{Complexity penalty}} + \frac{n_y}{2}\log(2\pi)$$

- We can compute derivatives of this function with respect to θ such that we can optimize the hyperparameters using standard algorithms

Useful Tips to Know When Learning Hyperparameters

- Set initial hyperparameters using domain knowledge wherever possible
- Otherwise:
  - Standardize input data (min-max scaling) and set lengthscales ~ 1
  - Standardize output data (mean-std scaling) and set function variance ~ 1
- Often useful: Set initial noise level high, even if you think your data has low noise to make the optimization surface easier to move in
- Random restarts / other tricks to avoid local optima are advised when minimizing negative log-likelihood using gradient-based methods

Beyond Traditional GPs

- Sparse Gaussian Processes
  - GP expressions scale cubically with the number of data points since we need to compute the inverse of the covariance matrix. One way to reduce this complexity is to optimally select a smaller set of points that summarizes the available data.
  - See M. Titsias, "Variational learning of inducing variables in sparse Gaussian processes", Artificial Intelligence and Statistics, 2009

- Heteroskedastic noise
  - GP expressions assume that noise characteristics are constant throughout the features space. An alternative is to build a GP model of the noise characteristics (creates challenges for training and prediction)
  - See notebook for an example + references

GPyTorch: Python Package for (Flexible) GP Modeling

- GPyTorch (https://gpytorch.ai/) is an efficient and modular implementation of GPs implemented on top of PyTorch (GPU acceleration)
- Several BO packages are built on top of GPyTorch including the one we will look at more later in this workshop
- There are a lot of advanced features implemented in GPyTorch, though we will only look at the relatively standard features needed for BO
  - Including sparse and heteroskedastic GP models

GPyTorch: Python Package for (Flexible) GP Modeling

Setup training data

```
# Training data is 100 points in [0,1] inclusive regularly spaced
train_x = torch.linspace(0, 1, 100)
# True function is sin(2*pi*x) with Gaussian noise
train_y = torch.sin(train_x * (2 * math.pi)) + torch.randn(train_x.size()) * math.sqrt(0.0
```

Define model class with desired mean and kernel functions (object oriented)

```
# We will use the simplest form of GP model, exact inference
class ExactGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(ExactGPModel, self).__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)

# initialize likelihood and model
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = ExactGPModel(train_x, train_y, likelihood)
```

GPyTorch: Python Package for (Flexible) GP Modeling

Call an optimizer to train the model (for some number of steps)

```
# Find optimal model hyperparameters
model.train()
likelihood.train()

# Use the adam optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)  # Includes GaussianLikelihood pa

# "Loss" for GPs - the marginal log likelihood
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

for i in range(training_iter):
    # Zero gradients from previous iteration
    optimizer.zero_grad()
    # Output from model
    output = model(train_x)
    # Calc loss and backprop gradients
    loss = -mll(output, train_y)
    loss.backward()
    print('Iter %d/%d - Loss: %.3f   lengthscale: %.3f   noise: %.3f' % (
        i + 1, training_iter, loss.item(),
        model.covar_module.base_kernel.lengthscale.item(),
        model.likelihood.noise.item()
    ))
    optimizer.step()
```

All model parameters are optimized, which includes mean, kernel, and likelihood parameters (unless they are explicitly set to not trainable)

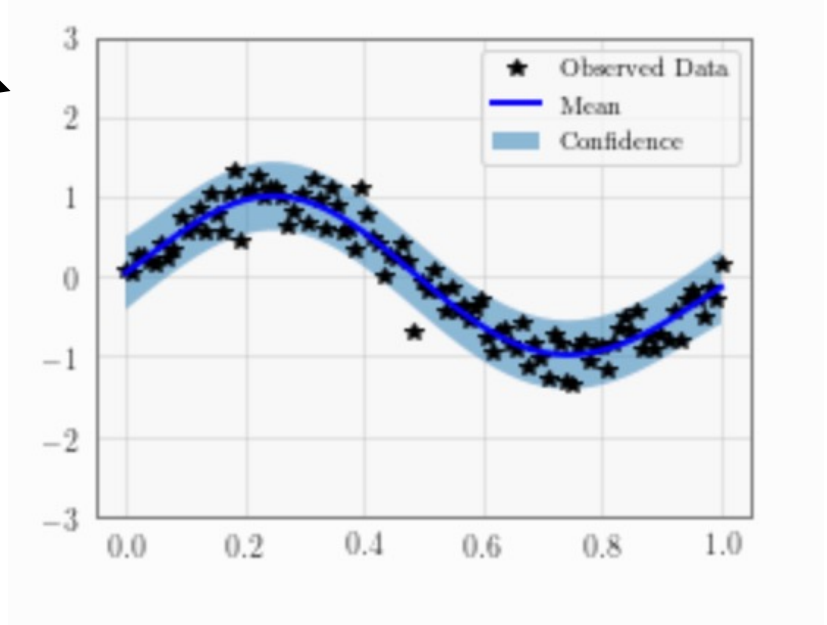
GPyTorch: Python Package for (Flexible) GP Modeling

Turn model into eval mode and evaluate model to get predictions

```
# Get into evaluation (predictive posterior) mode
model.eval()
likelihood.eval()

# Test points are regularly spaced along [0,1]
# Make predictions by feeding model through likelihood
with torch.no_grad(), gpytorch.settings.fast_pred_var():
    test_x = torch.linspace(0, 1, 51)
    observed_pred = likelihood(model(test_x))
```

Plot mean with confidence bounds calculated from predictions



CODE REVIEW

Workshop Schedule

9:00 – 9:20	Introduction: Why Go Beyond Traditional Optimization?
9:20 – 10:20	Module 1: Probabilistic Surrogate Modeling*
10:20 – 10:30	Break
10:30 – 11:20	Module 2: Quantifying the Value of Information*
11:20 – 12:20	Module 3: The BO Feedback Loop*
12:20 – 12:30	Break
12:30 – 1:00	Module 4: Beyond Bayesian Optimization

*module includes Python code review / exercises