# Lab 9 : Gaussian Processes (GPs)

Jeroen Olieslagers
Richard-John Lin

Center for Data Science

15/01/2022

## Computing the weighted mean and variance

To calculate the mean and variance of the latents, have to use the **weighted** mean and variance of the particles.

$$\mathbb{E}[x] = \int x p(x) dx \approx \sum_i X^{(i)} W^{(i)} \tag{1}$$

$$\mathbb{V}[x] = \int (x - \mathbb{E}[x])^2 p(x) dx \approx \sum_i (X^{(i)} - \mathbb{E}[x])^2 W^{(i)} \tag{2}$$

## Updating the last sample

Due to index inside the loop being $n - 1$, have to remember to update the final estimate for the latent mean and variance.

```
### track for last sample:
w = self.compute_w(data[-1, :], z_samp)
self.est_z_mean[-1] = np.sum(w * z_samp)
self.est_z_var[-1] = np.sum(w * (z_samp - self.est_z_mean[-1]) ** 2)
```

## Use different instantiation of noise for each particle propagation

Each particle ($Z_t^{(i)}$) is propagated independently of each other.

$$Z_{t+1}^{(i)} = AZ_t^{(i)} + W_t^{(i)} \tag{3}$$

where each $W_t^{(i)} \sim \mathcal{N}(0, Q)$. Each of these $W_t^{(i)}$ is sampled **i.i.d**. The following is wrong:

$$Z_{t+1}^{(i)} = AZ_t^{(i)} + W_t \tag{4}$$

where one instance of $W_t$ is used to propagate all particles.

## Linear Regression

Given a set of points $(x, y)$, we want to find the points from a modelled signal i.e.

$$y|x \sim \mathcal{N}(f, \sigma^2)$$
$$f(x) = w^\top \psi(x).$$

- Under these assumptions, maximum likelihood is equivalent to linear regression.
- With an additional prior on the weights, the maximum a posteriori solution is equivalent to regularized linear regression.
- The full Bayesian inference would average over all likely explanations under the posterior distribution.

## Bayesian Linear Regression

Assuming :

$$w \sim \mathcal{N}(0, S)$$
$$y|x, w \sim \mathcal{N}(w^\top \psi(x), \sigma^2),$$

we can write :

$$\begin{aligned}\log p(w|\mathcal{D}) &= \log p(w) + \log p(\mathcal{D}|w) + K \\ &= -\frac{1}{2} w^\top Z^{-1} w - \frac{1}{2\sigma^2} ||\psi w - y||^2 + K.\end{aligned}$$

## Bayesian Linear Regression

Developing and completing the square, we can show that this is a multivariate Gaussian distribution :

$$w|\mathcal{D} \sim \mathcal{N}(\mu, \Sigma),$$

where,

$$\mu = \frac{1}{\sigma^2}\Sigma\psi(x)^\top y$$
$$\Sigma^{-1} = \frac{1}{\sigma^2}\psi(x)^\top\psi(x) + S^{-1}.$$

## Bayesian Linear Regression

We want the predictive distribution :

$$p(y_*|x_*, \mathcal{D}) = \int p(t_*|x_*, w)p(w|\mathcal{D})\,\mathrm{d}w.$$

All quantities are Gaussian, we can compute this integral :

$$\mu_{\text{pred}} = \mu^\top \psi(x_*)$$
$$\sigma^2_{\text{pred}} = \psi(x_*)^\top \Sigma \psi(x_*) + \sigma^2.$$

Note that this quantity could be expressed in terms of :

$$\mu_f = \mu_w^\top \psi(x)$$
$$\Sigma_f = \psi(x^\top \Sigma_w \psi(x).$$

We don't need to know $w$ anymore !
What if we now let those quantities to be arbitrary ?

## Gaussian Process

### Definition

A Gaussian Process (GP) is a collection of random variables, any finite number of which a have a joint Gaussian distribution.
We write $f(x) \sim \mathcal{GP}(m, k)$ :

$$[f(x_1), \ldots, f(x_N)] \sim \mathcal{N}(\mu, K)$$
$$\mu_i = m(x_i)$$
$$K_{ij} = k(x_i, x_j),$$

where $x_1, \ldots, x_N$ are inputs.
$m$ is the mean function, $k$ is the covariance kernel.

## Inference

We are interested in making predictions $f_*$ at points $x_*$. Hence we are interested in the distribution $p(f_*|x, y, x_*)$.
Under the following assumptions :

$$y(x) \sim \mathcal{N}\left(f(x), \sigma^2\right),$$
$$f(x) \sim \mathcal{GP}(0, k_\theta),$$

we can write :

$$\begin{bmatrix} y \\ f_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} k_\theta(x,x) + \sigma^2 I & k_\theta(x, x_*) \\ k_\theta(x_*, x) & h_\theta(x_*, x_*) \end{bmatrix}\right).$$

## Inference

We are interested in making predictions $f_*$ at points $x_*$. Hence we are interested in the distribution $p(f_*|x, y, x_*)$.
Using gaussian identities,

$$f_*|x_*, x, y, \theta \sim \mathcal{N}(\bar{f}_*, \text{cov}(f_*))$$

$$\bar{f}_* = k_\theta(x_*, x)\left[k_\theta(x, x) + \sigma^2 I\right]^{-1} y,$$

$$\text{cov}(f_*) = k_\theta(x_*, x_*) - k_\theta(x_*, x)\left[k_\theta(x, x) + \sigma^2 I\right]^{-1} k_\theta(x, x_*).$$

## Choosing hyperparameters

- The problem from the previous expression is to determine the hyperparameters $\theta$.
- If we wanted to be fully Bayesian, we could specify a prior over those hyperparameters and marginalize them.
- In practice, if the prior is flat, we can use a maximum marginal likelihood approach i.e. maximize :

$$p(y|x,\theta) = \int p(y|f,x,\theta)p(f|x,\theta)\,\mathrm{d}f.$$

For GP, it is reasonable to assume that $f|X \sim \mathcal{N}(0,K)$. We then have $y \sim \mathcal{N}(0, K + \sigma_n^2 I)$.

## Kernel trick

- In the Bayesian linear regression example, we used the kernel trick.
- The aim to express a target in function of a dot product between feature vectors i.e. $< x, x' >= \psi(x)^\top \psi(x')$.
- A kernel implements an inner product between feature vectors, typically implicitly, and often much more efficiently than the explicit dot product.
- Example :

$$\phi(x) = (1, \sqrt{2}x_1, \ldots, \sqrt{2}x_d, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \ldots, \sqrt{2}x_{d-1}x_d, x_1^2, \ldots,$$

but :

$$k(x, x') =< \phi(x), \phi(x') >= (1+ < x, x' >)^2,$$

which is linear !

## Kernel trick

- We often think directly in kernel space, rather than in feature space.
- The kernel allows to use very high dimensional feature spaces, but we do have a computational cost.
  **GP regression** :

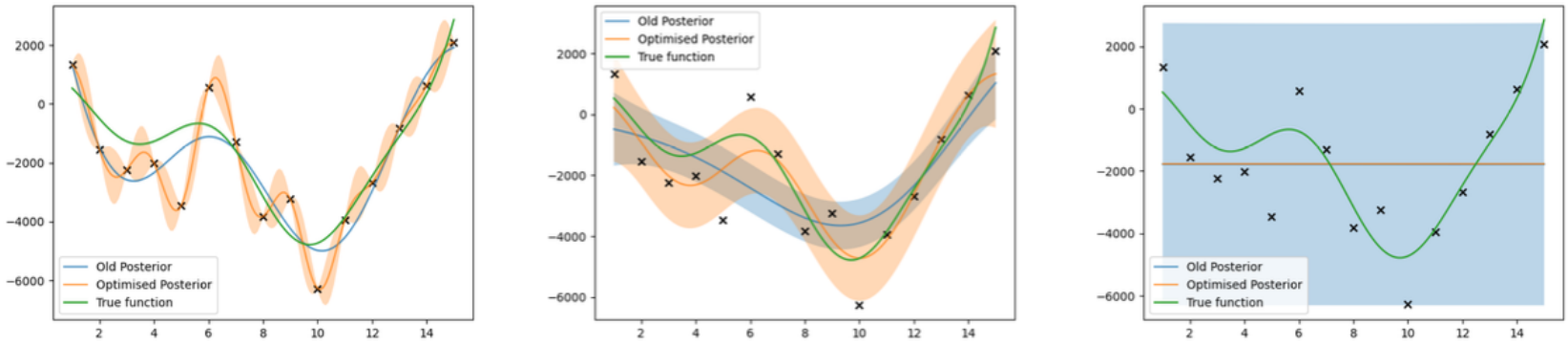
$$f_*|x_*, x, y, \theta \sim \mathcal{N}(\bar{f}_*, \text{cov}(f_*))$$

$$\bar{f}_* = k_\theta(x_*, x)\left[k_\theta(x, x) + \sigma^2 I\right]^{-1} y,$$

$$\text{cov}(f_*) = k_\theta(x_*, x_*) - k_\theta(x_*, x)\left[k_\theta(x, x) + \sigma^2 I\right]^{-1} k_\theta(x, x_*).$$

  We need to invert an $N \times N$ matrix !
- The $O(N^3)$ cost is typical of kernel methods.

# Sensitivity to initial conditions



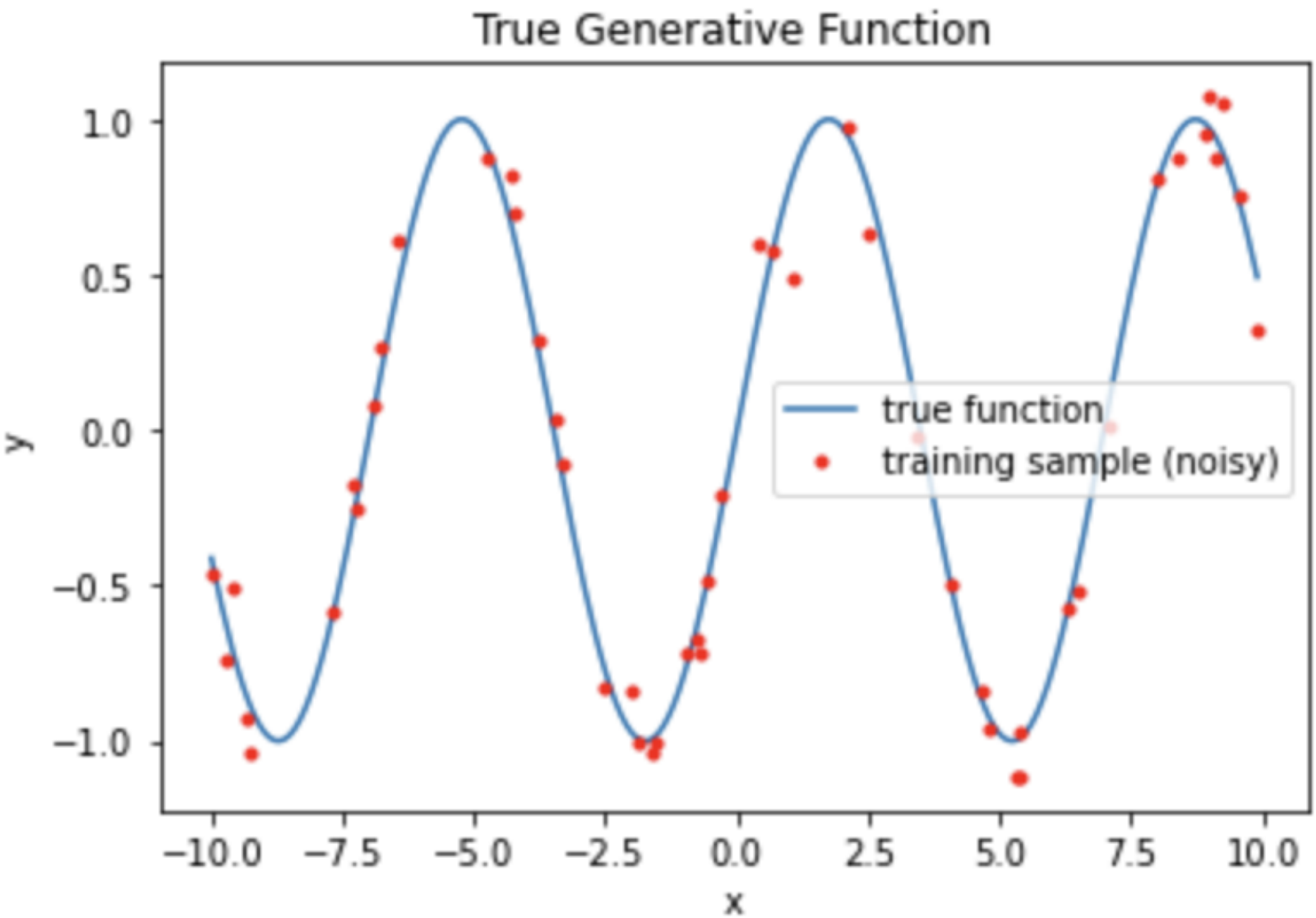

(a) Low noise, loss=1.27 (b) Intermediate noise, loss=1.21 (c) High noise, loss=3.86

Figure 1: Initial conditions

## References

- CSC 411 Lecture 20: Gaussian Processes
- Pattern Recognition and Machine Learning, Bishop

# Part I: Data generation



# Part II: Kernels

## Part II GP with sklearn

Sklearn has a very handy API for Gaussian Process regression.

http://scikit-learn.org/stable/modules/gaussian_process.html

### Kernel functions

Kernels to parametrize covariance structure

Constant Kernel: covariance is defined by a constant value

RBF (squared exponential) Kernel:

$$K(x_m, x_n) = exp\left(-\frac{||x_m - x_n||^2}{2 * l^2}\right)$$

White Kernel: accords for noise-component

$$K(x_m, x_n) = \text{noise} \quad \text{if } x_m = x_n \text{ else } 0$$

# Part II: Kernels

**fitting the GP model**

The fit() method automatically selects the hyper-parameters of given kernels.

```
gp = gaussian_process.GaussianProcessRegressor(kernel=kernel)
gp.fit(X_train.reshape(-1,1), y_train.reshape(-1,1))
```

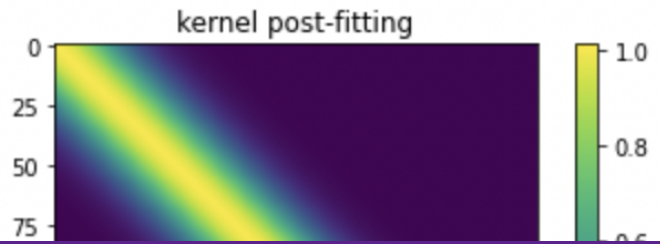
```
/opt/miniconda3/envs/main/lib/python3.9/site-packages/sklearn/gaussian_process/kernels.py:
e optimal value found for dimension 0 of parameter k1__k1__constant_value is close to the
5. Decreasing the bound and calling fit again may find a better value.
  warnings.warn(
```

```
GaussianProcessRegressor(kernel=1**2 + RBF(length_scale=2) + WhiteKernel(noise_level=1))
```

```
# print the kernel with fitted parameters
print(gp.kernel_)
plt.figure()
plt.imshow(gp.kernel_(np.array([all_x]).T))
plt.colorbar()
plt.title('kernel post-fitting')
```

```
0.00316**2 + RBF(length_scale=2.02) + WhiteKernel(noise_level=0.011)
```

```
Text(0.5, 1.0, 'kernel post-fitting')
```



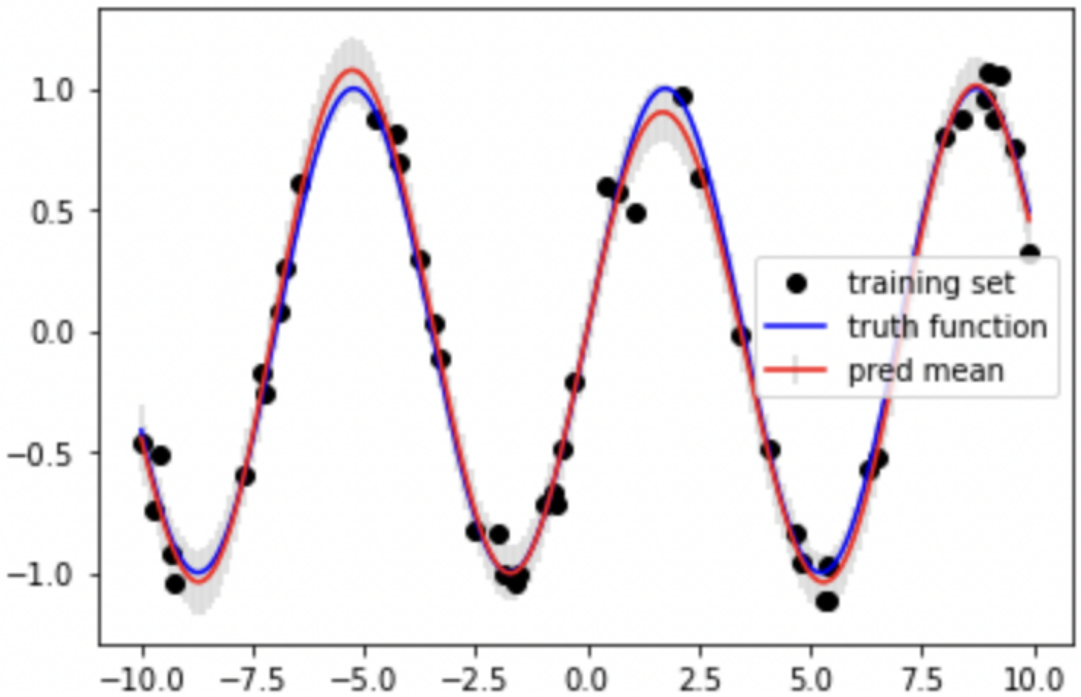
# Part II: inference with sklearn

**prediction of new values**

The predict method returns both mean and std.

```
mus, sigmas = gp.predict(all_x.reshape(-1,1), return_std=True)
```

```
plot_gp(all_x, mus[:, 0], sigmas, X_train, y_train, true_y=true_y)
```



# Part III: Cholensky predictions

$$p(\mathbf{t}^{pred}|\mathbf{t}^{train}) = N\left(\mu_{\mathbf{t}^{pred}|\mathbf{t}^{train}}, V_{\mathbf{t}^{pred}|\mathbf{t}^{train}}\right)$$

$$\mu_{\mathbf{t}^{pred}|\mathbf{t}^{train}} = K^T(C^{train})^{-1}\mathbf{t}^{train}$$
$$V_{\mathbf{t}^{pred}|\mathbf{t}^{train}} = C^{pred} - K^T(C^{train})^{-1}K$$

*Note that here, we assume zero mean*

### inference using the Cholesky Decomposition

- faster and more stable way to compute $\mu_{\mathbf{t}^{pred}|\mathbf{t}^{train}}$ and $V_{\mathbf{t}^{pred}|\mathbf{t}^{train}}$ given that $(C^{train})^{-1}$ is not guaranteed to be non-singular
- The Cholesky decomposition converts a (Hermitian, positive-definite) matrix $A$ into the product of a lower triangular matrix $L$ and its conjugate transpose $L^*$
- We use the Cholesky decomposition to get $C^{train} = LL^T$

*Because our covariance matrix $(C^{train})^{-1}$ is positive-definite and a real matrix that mirrors itself along the diagonal, it is a Hermitian matrix*

*L will be a real-value matrix so its conjugate is itself*

- From this we get:

$$\mu_{\mathbf{t}^{pred}|\mathbf{t}^{train}} = K^T(C^{train})^{-1}\mathbf{t}^{train} = K^T(LL^T)^{-1}\mathbf{t}^{train} = K^T(L^T)^{-1}L^{-1}\mathbf{t}^{train} = (L^{-1}K)^T(L^{-1}\mathbf{t}^{train})$$

$$V_{\mathbf{t}^{pred}|\mathbf{t}^{train}} = C^{pred} - K^T(C^{train})^{-1}K = C^{pred} - (L^{-1}K)^T(L^{-1}K)$$

*where **L=cholesky(C)***

$L^{-1}K$ *and* $L^{-1}\mathbf{t}^{train}$ *can be obtained by* ***solving the linear system*** $Lx = K$ *and* $Lx = \mathbf{t}^{train}$ ***using np.linalg.solve***

# Part III: inference

## prediction giving varying number of training data points

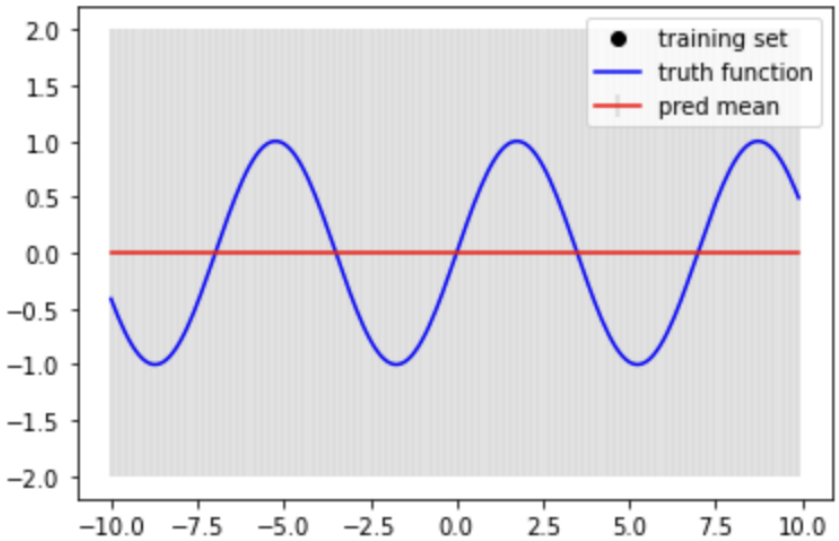
Prior distribution

$$y \sim N(\mu_0, \sigma_0^2)$$

Since we assume a zero mean function, we have $\mu_0 = E[y] = 0$.

```
mu_0 = np.zeros(len(all_x))
sigma_0 = np.sqrt(exponential_cov(0, 0, kernel_parameters))
plot_gp(all_x, mu_0, sigma_0, [], [], true_y)
print("rmse = {0}".format(np.sqrt(mean_squared_error(mu_0, true_y))))
```

```
rmse = 0.7216677922512522
```



# Part IV: sampling

## Part IV: Sampling

For this part, we implement the sample_cholesky function.

### sampling from multivariate Gaussian

use property of multivariate Gaussian where if $z \sim N(0, I)$ then $x = \mu + Lz$ gives $x \sim N(\mu, LL^T)$ where $L = cholesky(LL^T)$

```
def sample_cholesky(mu, cov, n_points, n_samples):
    """
    Function that performs sampling from multi-variate Gaussian using Cholesky Decomposition
    @param mu: a numpy array of size n
    @param cov: a numpy matrix of size n*n
    @param n_points: how many points per sample
    @param n_samples: how many samples
    @return a numpy matrix with dimension (n_points, n_samples)
    """
    ##########################################
    ### TODO: please implement this function ###
    ##########################################
    return np.zeros([n_points, n_samples])
```