Inference on the Champagne Model using a Gaussian Process

TODO

- Set seed for LHC and stuff
- Change from MLE to cross validation

Setting up the Champagne Model

Imports

```
import pandas as pd
import numpy as np
from typing import Any
import matplotlib.pyplot as plt

from scipy.stats import qmc

import tensorflow as tf
import tensorflow_probability as tfp
tfb = tfp.bijectors
tfd = tfp.distributions
tfk = tfp.math.psd_kernels
```

Model itself

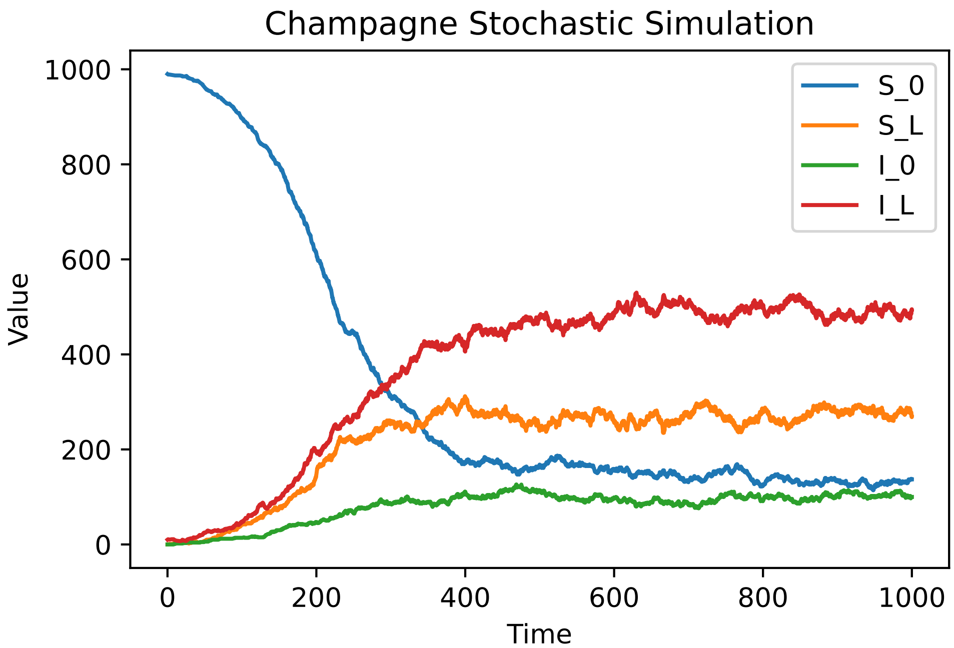
```
np.random.seed(590154)

population = 1000
initial_infecteds = 10
epidemic_length = 1000

pv_champ_alpha = 0.4  # prop of effective care
pv_champ_beta = 0.4  # prop of radical cure
pv_champ_gamma_L = 1 / 223  # liver stage clearance rate
pv_champ_delta = 0.05  # prop of imported cases
pv_champ_lambda = 0.04  # transmission rate
pv_champ_f = 1 / 72  # relapse frequency
pv_champ_r = 1 / 60  # blood stage clearance rate


def champagne_stochastic(
    alpha_,
    beta_,
    gamma_L,
    lambda_,
    f,
    r,
    N=population,
    I_L=initial_infecteds,
    I_0=0,
    S_L=0,
    delta_=0,
    end_time=epidemic_length,
):
    t = 0
    S_0 = N - I_L - I_0 - S_L
    list_of_outcomes = [{"t": 0, "S_0": S_0, "S_L": S_L, "I_0": I_0, "I_L": I_L}]

    while t < end_time:
        if S_0 == N:
            break

        S_0_to_I_L = (1 - alpha_) * lambda_ * (I_L + I_0) / N * S_0
        S_0_to_S_L = alpha_ * (1 - beta_) * lambda_ * (I_0 + I_L) / N * S_0
        I_0_to_S_0 = r * I_0 / N
        I_0_to_I_L = lambda_ * (I_L + I_0) / N * I_0
```

```
        I_L_to_I_0 = gamma_L * I_L
        I_L_to_S_L = r * I_L
        S_L_to_S_0 = (gamma_L + (f + lambda_ * (I_0 + I_L) / N) * alpha_ * beta_) * S_L
        S_L_to_I_L = (f + lambda_ * (I_0 + I_L) / N) * (1 - alpha_) * S_L

        total_rate = (
            S_0_to_I_L
            + S_0_to_S_L
            + I_0_to_S_0
            + I_0_to_I_L
            + I_L_to_I_0
            + I_L_to_S_L
            + S_L_to_S_0
            + S_L_to_I_L
        )

        t += np.random.exponential(1 / total_rate)
        new_stages_prob = [
            S_0_to_I_L / total_rate,
            S_0_to_S_L / total_rate,
            I_0_to_S_0 / total_rate,
            I_0_to_I_L / total_rate,
            I_L_to_I_0 / total_rate,
            I_L_to_S_L / total_rate,
            S_L_to_S_0 / total_rate,
            S_L_to_I_L / total_rate,
        ]
        new_stages = np.random.choice(
            [
                {"t": t, "S_0": S_0 - 1, "S_L": S_L, "I_0": I_0, "I_L": I_L + 1},
                {"t": t, "S_0": S_0 - 1, "S_L": S_L + 1, "I_0": I_0, "I_L": I_L},
                {"t": t, "S_0": S_0 + 1, "S_L": S_L, "I_0": I_0 - 1, "I_L": I_L},
                {"t": t, "S_0": S_0, "S_L": S_L, "I_0": I_0 - 1, "I_L": I_L + 1},
                {"t": t, "S_0": S_0, "S_L": S_L, "I_0": I_0 + 1, "I_L": I_L - 1},
                {"t": t, "S_0": S_0, "S_L": S_L + 1, "I_0": I_0, "I_L": I_L - 1},
                {"t": t, "S_0": S_0 + 1, "S_L": S_L - 1, "I_0": I_0, "I_L": I_L},
                {"t": t, "S_0": S_0, "S_L": S_L - 1, "I_0": I_0, "I_L": I_L + 1},
            ],
            p=new_stages_prob,
        )

        list_of_outcomes.append(new_stages)
```

```
        S_0 = new_stages["S_0"]
        I_0 = new_stages["I_0"]
        I_L = new_stages["I_L"]
        S_L = new_stages["S_L"]

    outcome_df = pd.DataFrame(list_of_outcomes)
    return outcome_df


champ_samp = champagne_stochastic(
    pv_champ_alpha,
    pv_champ_beta,
    pv_champ_gamma_L,
    pv_champ_lambda,
    pv_champ_f,
    pv_champ_r,
)  # .melt(id_vars='t')
```

Plotting outcome

```
champ_samp.plot(x = 't',legend=True)
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Champagne Stochastic Simulation')
plt.show()
```



Function that Outputs Final Prevalence

```
def champ_prevalence(alpha_, beta_, gamma_L, lambda_, f, r):
    champ_df_ = champagne_stochastic(alpha_, beta_, gamma_L, lambda_, f, r)

    return(champ_df_.iloc[-1]["I_0"] + champ_df_.iloc[-1]["I_L"])

observed_final_prevalence = champ_prevalence(pv_champ_alpha, pv_champ_beta,
pv_champ_gamma_L, pv_champ_lambda, pv_champ_f, pv_champ_r)
```

Gaussian Process Regression on Final Prevalence Discrepency

```
my_seed = np.random.default_rng(seed=1795)  # For replicability

num_samples = 2000

variables_names = ["alpha", "beta", "gamma_L", "lambda", "f", "r"]
```

```python
pv_champ_alpha = 0.4  # prop of effective care
pv_champ_beta = 0.4  # prop of radical cure
pv_champ_gamma_L = 1 / 223  # liver stage clearance rate
pv_champ_lambda = 0.04  # transmission rate
pv_champ_f = 1 / 72  # relapse frequency
pv_champ_r = 1 / 60  # blood stage clearance rate

samples = np.concatenate(
    (
        my_seed.uniform(low=0, high=1, size=(num_samples, 1)),  # alpha
        my_seed.uniform(low=0, high=1, size=(num_samples, 1)),  # beta
        my_seed.exponential(scale=pv_champ_gamma_L, size=(num_samples, 1)),  # gamma_L
        my_seed.exponential(scale=pv_champ_lambda, size=(num_samples, 1)),  # lambda
        my_seed.exponential(scale=pv_champ_f, size=(num_samples, 1)),  # f
        my_seed.exponential(scale=pv_champ_r, size=(num_samples, 1)),  # r
    ),
    axis=1,
)

LHC_sampler = qmc.LatinHypercube(d = 6)
LHC_samples = LHC_sampler.random(n = num_samples)
LHC_samples[:,2] = -pv_champ_gamma_L*np.log(LHC_samples[:,2])
LHC_samples[:,3] = -pv_champ_lambda*np.log(LHC_samples[:,3])
LHC_samples[:,4] = -pv_champ_f*np.log(LHC_samples[:,4])
LHC_samples[:,5] = -pv_champ_r*np.log(LHC_samples[:,5])

random_indices_df = pd.DataFrame(samples, columns=variables_names)
LHC_indices_df = pd.DataFrame(LHC_samples, columns=variables_names)

print(random_indices_df.head())
print(LHC_indices_df.head())
```

```
      alpha      beta   gamma_L    lambda         f         r
0  0.201552  0.702424  0.023296  0.035501  0.030907  0.002958
1  0.332324  0.657802  0.001419  0.030386  0.014104  0.021536
2  0.836050  0.962593  0.003359  0.042609  0.013526  0.022165
3  0.566773  0.763411  0.005252  0.009734  0.017709  0.002724
4  0.880603  0.689347  0.002171  0.045976  0.005510  0.023899
      alpha      beta   gamma_L    lambda         f         r
0  0.852327  0.419795  0.000310  0.007013  0.026605  0.001401
1  0.040156  0.648076  0.000251  0.011606  0.010508  0.028965
2  0.373890  0.434938  0.005779  0.029908  0.003569  0.025145
```

```
3  0.847464  0.063640  0.014582  0.193962  0.008915  0.050133
4  0.729473  0.805735  0.006799  0.048899  0.049800  0.032807
```

Generate Discrepencies

```
random_prevalences = LHC_indices_df.apply(
    lambda x: champ_prevalence(
        x["alpha"], x["beta"], x["gamma_L"], x["lambda"], x["f"], x["r"]
    ),
    axis=1,
)

random_discrepencies = np.abs(random_prevalences - observed_final_prevalence)
print(random_discrepencies.head())
```

```
0    535.0
1    409.0
2    278.0
3    401.0
4    432.0
dtype: float64
```

Differing Methods to Iterate Function

```
# import timeit

# def function1():
#     np.vectorize(champ_prevalence)(random_indices_df['alpha'],
#     random_indices_df['beta'], random_indices_df['gamma_L'],
#     random_indices_df['lambda'], random_indices_df['f'], random_indices_df['r'])
#     pass

# def function2():
#     random_indices_df.apply(
#         lambda x: champ_prevalence(
#             x['alpha'], x['beta'], x['gamma_L'], x['lambda'], x['f'], x['r']),
#             axis = 1)
#     pass
```

```
# # Time function1
# time_taken_function1 = timeit.timeit(
#     "function1()", globals=globals(), number=100)

# # Time function2
# time_taken_function2 = timeit.timeit(
#     "function2()", globals=globals(), number=100)

# print("Time taken for function1:", time_taken_function1)
# print("Time taken for function2:", time_taken_function2)
```

Time taken for function1: 187.48960775700016 Time taken for function2: 204.06618941299985

Constrain Variables to be Positive

```
constrain_positive = tfb.Shift(np.finfo(np.float64).tiny)(tfb.Exp())
```

Custom Quadratic Mean Function

```
class quad_mean_fn(tf.Module):
    def __init__(self):
        super(quad_mean_fn, self).__init__()
        self.amp_alpha_mean = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=400.0,
            dtype=np.float64,
            name="amp_alpha_mean",
        )
        self.alpha_tp = tf.Variable(pv_champ_alpha, dtype=np.float64, name="alpha_tp")
        self.amp_beta_mean = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=50.0,
            dtype=np.float64,
            name="amp_beta_mean",
        )
        self.beta_tp = tf.Variable(pv_champ_beta, dtype=np.float64, name="beta_tp")
        self.amp_gamma_L_mean = tfp.util.TransformedVariable(
```

```
            bijector=constrain_positive,
            initial_value=500.0,
            dtype=np.float64,
            name="amp_gamma_L_mean",
        )
        self.gamma_L_tp = tf.Variable(
            pv_champ_gamma_L, dtype=np.float64, name="gamma_L_tp"
        )
        self.amp_lambda_mean = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=16000.0,
            dtype=np.float64,
            name="amp_lambda_mean",
        )
        self.lambda_tp = tf.Variable(
            pv_champ_lambda, dtype=np.float64, name="lambda_tp"
        )
        self.amp_f_mean = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=15000.0,
            dtype=np.float64,
            name="amp_f_mean",
        )
        self.f_tp = tf.Variable(pv_champ_f, dtype=np.float64, name="f_tp")
        self.amp_r_mean = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=13000.0,
            dtype=np.float64,
            name="amp_r_mean",
        )
        self.r_tp = tf.Variable(pv_champ_r, dtype=np.float64, name="r_tp")
        self.bias_mean = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=50.0,
            dtype=np.float64,
            name="bias_mean",
        )

    def __call__(self, x):
        return (
            self.amp_alpha_mean * (x[..., 0] - self.alpha_tp) ** 2
            + self.amp_beta_mean * (x[..., 1] - self.beta_tp) ** 2
```

```
            + self.amp_gamma_L_mean * (x[..., 2] - self.gamma_L_tp) ** 2
            + self.amp_lambda_mean * (x[..., 3] - self.lambda_tp) ** 2
            + self.amp_f_mean * (x[..., 4] - self.f_tp) ** 2
            + self.amp_r_mean * (x[..., 5] - self.r_tp) ** 2
            + self.bias_mean
        )
```

Making the ARD Kernel

```
index_vals = LHC_indices_df.values
obs_vals = random_discrepencies.values

amplitude_champ = tfp.util.TransformedVariable(
    bijector=constrain_positive,
    initial_value=150.0,
    dtype=np.float64,
    name="amplitude_champ",
)

observation_noise_variance_champ = tfp.util.TransformedVariable(
    bijector=constrain_positive,
    initial_value=1000.0,
    dtype=np.float64,
    name="observation_noise_variance_champ",
)
```

```
length_scales_champ = tfp.util.TransformedVariable(
    bijector=constrain_positive,
    initial_value=[0.01, 0.01, 0.35, 0.02, 0.27, 0.2],
    dtype=np.float64,
    name="length_scales_champ",
)
```

```
kernel_champ = tfk.FeatureScaled(tfk.ExponentiatedQuadratic(
    amplitude=amplitude_champ), scale_diag=length_scales_champ)
```

Define the Gaussian Process with Quadratic Mean Function and ARD Kernel

```
# Define Gaussian Process with the custom kernel
champ_GP = tfd.GaussianProcess(
    kernel=kernel_champ,
    observation_noise_variance=observation_noise_variance_champ,
    index_points=index_vals,
    mean_fn=quad_mean_fn(),
)

print(champ_GP.trainable_variables)

Adam_optim = tf.optimizers.Adam(learning_rate=.01)
```

```
(<tf.Variable 'amplitude_champ:0' shape=() dtype=float64, numpy=5.0106352940962555>, <tf.Var
array([-4.60517019, -4.60517019, -1.04982212, -3.91202301, -1.30933332,
       -1.60943791])>, <tf.Variable 'observation_noise_variance_champ:0' shape=() dtype=floa
```

Train the Hyperparameters

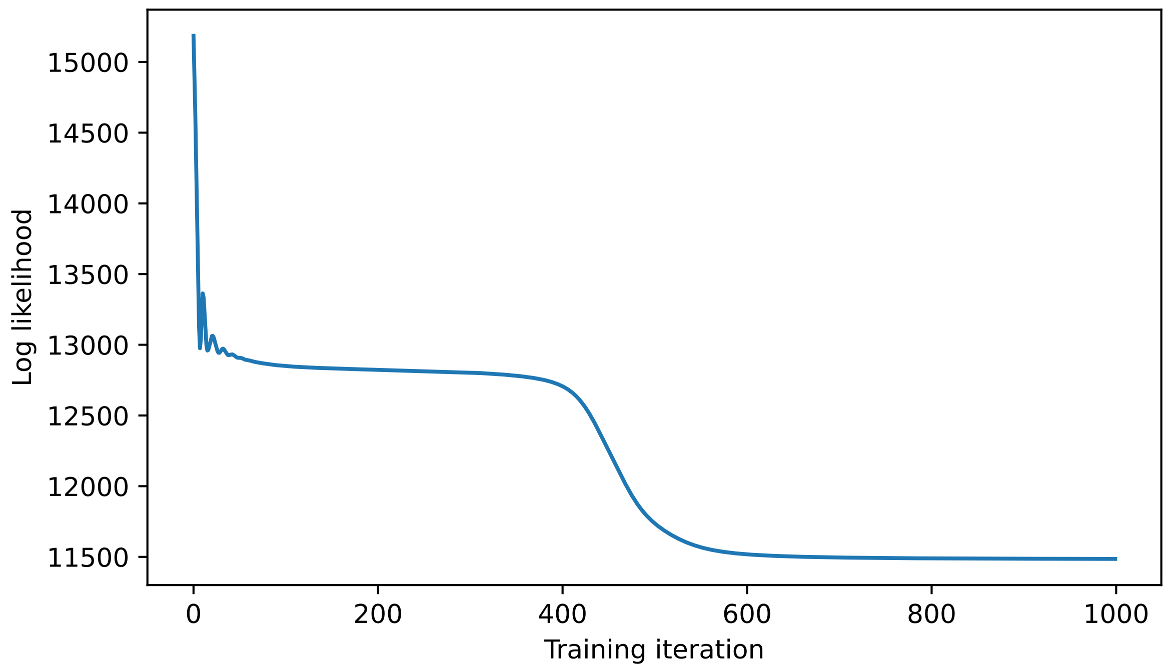
```
@tf.function()
def optimize():
    with tf.GradientTape() as tape:
        loss = -champ_GP.log_prob(obs_vals)
    grads = tape.gradient(loss, champ_GP.trainable_variables)
    Adam_optim.apply_gradients(zip(grads, champ_GP.trainable_variables))
    return loss


num_iters = 1000

lls_ = np.zeros(num_iters, np.float64)
for i in range(num_iters):
    loss = optimize()
    lls_[i] = loss
```

```
print("Trained parameters:")
for var in champ_GP.trainable_variables:
    if "tp" in var.name:
        print("{} is {}".format(var.name, var.numpy().round(3)))
    else:
```

```
        print(
            "{} is {}".format(
                var.name, constrain_positive.forward(var).numpy().round(3)
            )
        )
```

```
Trained parameters:
amplitude_champ:0 is 128.066
length_scales_champ:0 is [0.197 1.948 0.016 0.01  0.029 0.007]
observation_noise_variance_champ:0 is 2390.323
alpha_tp:0 is 0.219
amp_alpha_mean:0 is 279.981
amp_beta_mean:0 is 14.822
amp_f_mean:0 is 11565.86
amp_gamma_L_mean:0 is 30630.778
amp_lambda_mean:0 is 5914.323
amp_r_mean:0 is 23815.959
beta_tp:0 is -0.429
bias_mean:0 is 51.851
f_tp:0 is 0.089
gamma_L_tp:0 is 0.06
lambda_tp:0 is 0.18
r_tp:0 is 0.022
```

```
plt.figure(figsize=(7, 4))
plt.plot(lls_)
plt.xlabel("Training iteration")
plt.ylabel("Log likelihood")
plt.show()
```



Fitting the GP Regression across alpha

```
plot_samp_no = 21
gp_samp_no = 50
```

```
samples = np.concatenate(
    (
        np.linspace(0, 1, plot_samp_no, dtype=np.float64).reshape(-1, 1),  # alpha
        np.repeat(pv_champ_beta, plot_samp_no).reshape(-1, 1),  # beta
        np.repeat(pv_champ_gamma_L, plot_samp_no).reshape(-1, 1),  # gamma_L
        np.repeat(pv_champ_lambda, plot_samp_no).reshape(-1, 1),  # lambda
        np.repeat(pv_champ_f, plot_samp_no).reshape(-1, 1),  # f
        np.repeat(pv_champ_r, plot_samp_no).reshape(-1, 1),  # r
    ),
    axis=1,
)

plot_indices_df = pd.DataFrame(samples, columns=variables_names)

print(plot_indices_df.head())
```

```
plot_prevalences = plot_indices_df.apply(
    lambda x: champ_prevalence(
        x["alpha"], x["beta"], x["gamma_L"], x["lambda"], x["f"], x["r"]
    ),
    axis=1,
)

plot_discrepencies = np.abs(plot_prevalences - observed_final_prevalence)

plot_index_vals = plot_indices_df.values
```

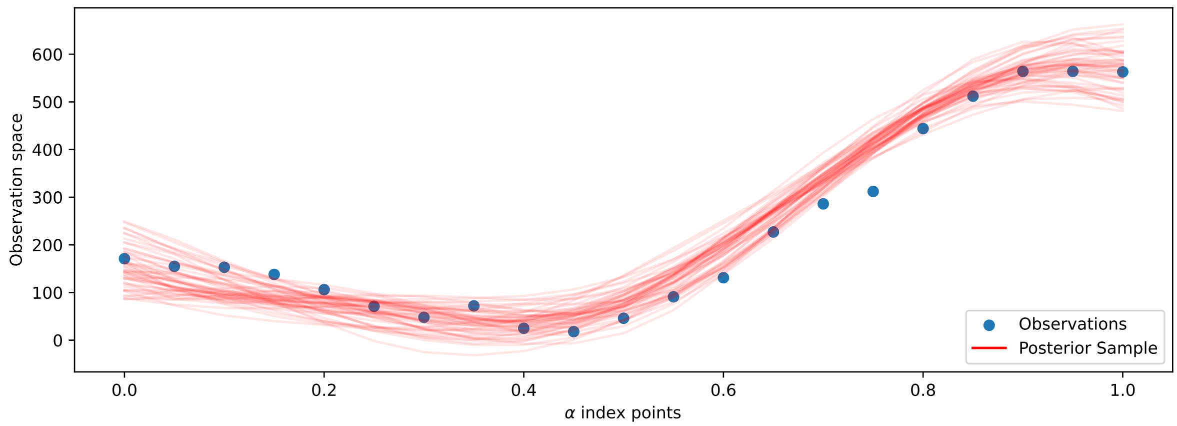
```
   alpha  beta   gamma_L  lambda         f         r
0   0.00   0.4  0.004484    0.04  0.013889  0.016667
1   0.05   0.4  0.004484    0.04  0.013889  0.016667
2   0.10   0.4  0.004484    0.04  0.013889  0.016667
3   0.15   0.4  0.004484    0.04  0.013889  0.016667
4   0.20   0.4  0.004484    0.04  0.013889  0.016667
```

```
champ_GP_reg = tfd.GaussianProcessRegressionModel(
    kernel=kernel_champ,
    index_points=plot_index_vals,
    observation_index_points=index_vals,
    observations=obs_vals,
    observation_noise_variance=observation_noise_variance_champ,
    predictive_noise_variance=0.,
    mean_fn=quad_mean_fn(),
)

GP_samples = champ_GP_reg.sample(gp_samp_no)
```

```
plt.figure(figsize=(12, 4))
plt.scatter(plot_index_vals[:, 0], plot_discrepencies,
            label='Observations')
for i in range(gp_samp_no):
  plt.plot(plot_index_vals[:, 0], GP_samples[i, :], c='r', alpha=.1,
           label='Posterior Sample' if i == 0 else None)
leg = plt.legend(loc='lower right')
for lh in leg.legendHandles:
    lh.set_alpha(1)
plt.xlabel(r"$\alpha$ index points")
```

```
plt.ylabel("Observation space")
plt.show()
```

```
/tmp/ipykernel_6894/2354376184.py:8: MatplotlibDeprecationWarning: The legendHandles attribu
  for lh in leg.legendHandles:
```



Fitting the GP Regression across beta

```
samples = np.concatenate(
    (
        np.repeat(pv_champ_alpha, plot_samp_no).reshape(-1, 1),  # alpha
        np.linspace(0, 1, plot_samp_no, dtype=np.float64).reshape(-1, 1),  # beta
        np.repeat(pv_champ_gamma_L, plot_samp_no).reshape(-1, 1),  # gamma_L
        np.repeat(pv_champ_lambda, plot_samp_no).reshape(-1, 1),  # lambda
        np.repeat(pv_champ_f, plot_samp_no).reshape(-1, 1),  # f
        np.repeat(pv_champ_r, plot_samp_no).reshape(-1, 1),  # r
    ),
    axis=1,
)

plot_indices_df = pd.DataFrame(samples, columns=variables_names)

print(plot_indices_df.head())

plot_prevalences = plot_indices_df.apply(
    lambda x: champ_prevalence(
```

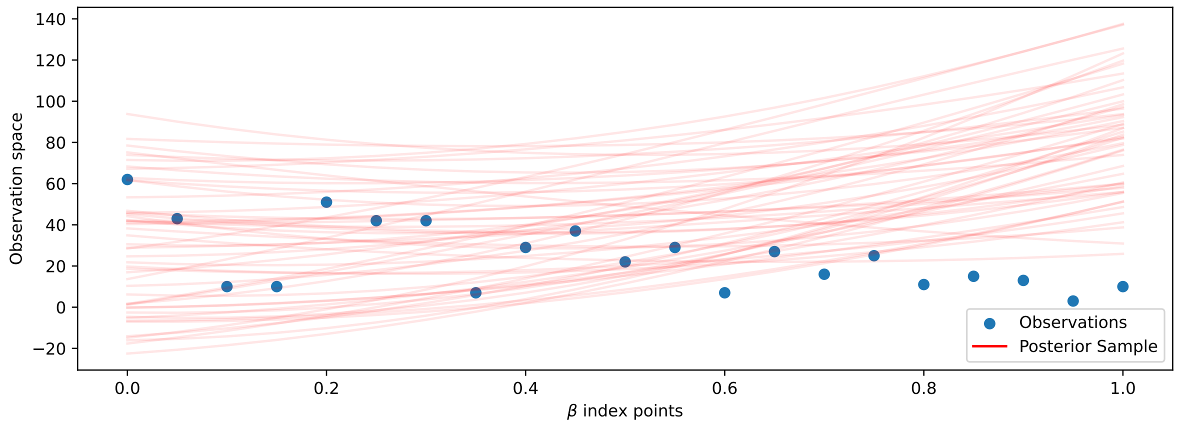
```
        x["alpha"], x["beta"], x["gamma_L"], x["lambda"], x["f"], x["r"]
    ),
    axis=1,
)

plot_discrepencies = np.abs(plot_prevalences - observed_final_prevalence)

plot_index_vals = plot_indices_df.values
```

```
   alpha  beta   gamma_L  lambda         f         r
0    0.4  0.00  0.004484    0.04  0.013889  0.016667
1    0.4  0.05  0.004484    0.04  0.013889  0.016667
2    0.4  0.10  0.004484    0.04  0.013889  0.016667
3    0.4  0.15  0.004484    0.04  0.013889  0.016667
4    0.4  0.20  0.004484    0.04  0.013889  0.016667
```

```
champ_GP_reg = tfd.GaussianProcessRegressionModel(
    kernel=kernel_champ,
    index_points=plot_index_vals,
    observation_index_points=index_vals,
    observations=obs_vals,
    observation_noise_variance=observation_noise_variance_champ,
    predictive_noise_variance=0.,
    mean_fn=quad_mean_fn(),
)

GP_samples = champ_GP_reg.sample(gp_samp_no)
```

```
plt.figure(figsize=(12, 4))
plt.scatter(plot_index_vals[:, 1], plot_discrepencies,
            label='Observations')
for i in range(gp_samp_no):
  plt.plot(plot_index_vals[:, 1], GP_samples[i, :], c='r', alpha=.1,
           label='Posterior Sample' if i == 0 else None)
leg = plt.legend(loc='lower right')
for lh in leg.legendHandles:
    lh.set_alpha(1)
plt.xlabel(r"$\beta$ index points")
plt.ylabel("Observation space")
plt.show()
```

```
/tmp/ipykernel_6894/1057594853.py:8: MatplotlibDeprecationWarning: The legendHandles attribu
  for lh in leg.legendHandles:
```



MCMC using the Gaussian Process