Inference on the Champagne Model using a Gaussian Process

TODO

- Set seed for LHC and stuff
- Change to log discrepency with custom observation variance
- 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
```

```
2024-04-12 13:59:29.145742: I external/local_tsl/tsl/cuda/cudart_stub.cc:31] Could not find c
2024-04-12 13:59:29.267292: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] U
```

```
2024-04-12 13:59:29.267385: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Un
2024-04-12 13:59:29.269265: E external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515]
2024-04-12 13:59:29.285931: I external/local_tsl/tsl/cuda/cudart_stub.cc:31] Could not find c
2024-04-12 13:59:29.286892: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorF
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with
2024-04-12 13:59:30.443284: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Wa
```

Model itself

```python
np.random.seed(590154)

population = 1000
initial_infecteds = 10
epidemic_length = 1000
number_of_events = 15000

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,
    num_events=number_of_events,
):
    if (0 > (alpha_ or beta_)) or (1 < (alpha_ or beta_)):
```

```
        return "Alpha or Beta out of bounds"
    if 0 > (gamma_L or lambda_ or f or r):
        return "Gamma, lambda, f or r out of bounds"

    t = 0
    S_0 = N - I_L - I_0 - S_L
    inc_counter = 0

    list_of_outcomes = [
        {"t": 0, "S_0": S_0, "S_L": S_L, "I_0": I_0, "I_L": I_L, "inc_counter": 0}
    ]

    prop_new = alpha_*beta_*f/(alpha_*beta_*f + gamma_L)

    for i in range(num_events):
        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
        )

        delta_t = np.random.exponential(1 / total_rate)
        new_stages_prob = [
            S_0_to_I_L / total_rate,
            S_0_to_S_L / total_rate,
```

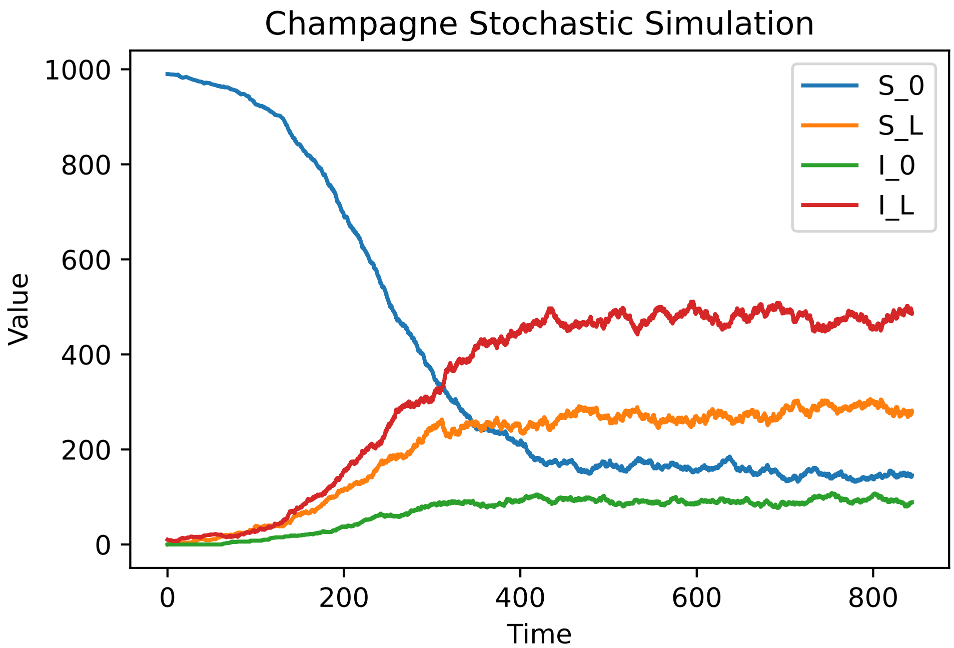
```
            I_0_to_S_0 / total_rate,
            I_0_to_I_L / total_rate,
            I_L_to_I_0 / total_rate,
            I_L_to_S_L / total_rate,
            S_L_to_S_0 / total_rate,
            S_L_to_I_L / total_rate,
        ]
        t += delta_t
        silent_incidences = np.random.poisson(
            delta_t * alpha_ * beta_ * lambda_ * (I_L + I_0) * S_0 / N
        )

        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,
                    "inc_counter": inc_counter + silent_incidences + 1,
                },
                {
                    "t": t,
                    "S_0": S_0 - 1,
                    "S_L": S_L + 1,
                    "I_0": I_0,
                    "I_L": I_L,
                    "inc_counter": inc_counter + silent_incidences + 1,
                },
                {
                    "t": t,
                    "S_0": S_0 + 1,
                    "S_L": S_L,
                    "I_0": I_0 - 1,
                    "I_L": I_L,
                    "inc_counter": inc_counter + silent_incidences,
                },
                {
                    "t": t,
                    "S_0": S_0,
                    "S_L": S_L,
```

```
                "I_0": I_0 - 1,
                "I_L": I_L + 1,
                "inc_counter": inc_counter + silent_incidences,
            },
            {
                "t": t,
                "S_0": S_0,
                "S_L": S_L,
                "I_0": I_0 + 1,
                "I_L": I_L - 1,
                "inc_counter": inc_counter + silent_incidences,
            },
            {
                "t": t,
                "S_0": S_0,
                "S_L": S_L + 1,
                "I_0": I_0,
                "I_L": I_L - 1,
                "inc_counter": inc_counter + silent_incidences,
            },
            {
                "t": t,
                "S_0": S_0 + 1,
                "S_L": S_L - 1,
                "I_0": I_0,
                "I_L": I_L,
                "inc_counter": inc_counter
                + silent_incidences
                + np.random.binomial(1, prop_new),
            },
            {
                "t": t,
                "S_0": S_0,
                "S_L": S_L - 1,
                "I_0": I_0,
                "I_L": I_L + 1,
                "inc_counter": inc_counter + silent_incidences + 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"]
        inc_counter = new_stages["inc_counter"]

    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.drop("inc_counter", axis=1).plot(x="t", legend=True)
plt.xlabel("Time")
plt.ylabel("Value")
plt.title("Champagne Stochastic Simulation")
plt.savefig("champagne_GP_images/champagne_simulation.pdf")
plt.show()
```



Function that Outputs Final Prevalence

```
def incidence(df, start, days):
    start_ind = df[df["t"].le(start)].index[-1]
    end_ind = df[df["t"].le(start + days)].index[-1]
    incidence_week = df.iloc[end_ind]["inc_counter"] - df.iloc[start_ind]["inc_counter"]
    return incidence_week


def champ_sum_stats(alpha_, beta_, gamma_L, lambda_, f, r):
    champ_df_ = champagne_stochastic(alpha_, beta_, gamma_L, lambda_, f, r)
    fin_t = champ_df_.iloc[-1]["t"]
    first_month_inc = incidence(champ_df_, 0, 30)
    fin_t = champ_df_.iloc[-1]["t"]
    fin_week_inc = incidence(champ_df_, fin_t - 7, 7)
    fin_prev = champ_df_.iloc[-1]["I_0"] + champ_df_.iloc[-1]["I_L"]

    return np.array([fin_prev, first_month_inc, fin_week_inc])


observed_sum_stats = champ_sum_stats(
```

```
    pv_champ_alpha,
    pv_champ_beta,
    pv_champ_gamma_L,
    pv_champ_lambda,
    pv_champ_f,
    pv_champ_r,
)


def discrepency_fn(alpha_, beta_, gamma_L, lambda_, f, r): # best is L1 norm
    x = champ_sum_stats(alpha_, beta_, gamma_L, lambda_, f, r)
    return np.log(np.sum(np.abs((x - observed_sum_stats) / observed_sum_stats)))
```

Testing the variances across different values of params etc.

```
# samples = 30
# cor_sums = np.zeros(samples)
# for i in range(samples):
#     cor_sums[i] = discrepency_fn(
#         pv_champ_alpha,
#         pv_champ_beta,
#         pv_champ_gamma_L,
#         pv_champ_lambda,
#         pv_champ_f,
#         pv_champ_r,
#     )

# cor_mean = np.mean(cor_sums)
# cor_s_2 = sum((cor_sums - cor_mean) ** 2) / (samples - 1)
# print(cor_mean, cor_s_2)

# doub_sums = np.zeros(samples)
# for i in range(samples):
#     doub_sums[i] = discrepency_fn(
#         2 * pv_champ_alpha,
#         2 * pv_champ_beta,
#         2 * pv_champ_gamma_L,
#         2 * pv_champ_lambda,
#         2 * pv_champ_f,
#         2 * pv_champ_r,
#     )
```

```
# doub_mean = np.mean(doub_sums)
# doub_s_2 = sum((doub_sums - doub_mean) ** 2) / (samples - 1)
# print(doub_mean, doub_s_2)

# half_sums = np.zeros(samples)
# for i in range(samples):
#     half_sums[i] = discrepency_fn(
#         pv_champ_alpha / 2,
#         pv_champ_beta / 2,
#         pv_champ_gamma_L / 2,
#         pv_champ_lambda / 2,
#         pv_champ_f / 2,
#         pv_champ_r / 2,
#     )

# half_mean = np.mean(half_sums)
# half_s_2 = sum((half_sums - half_mean) ** 2) / (samples - 1)
# print(half_mean, half_s_2)

# rogue_sums = np.zeros(samples)
# for i in range(samples):
#     rogue_sums[i] = discrepency_fn(
#         pv_champ_alpha / 2,
#         pv_champ_beta / 2,
#         pv_champ_gamma_L / 2,
#         pv_champ_lambda / 2,
#         pv_champ_f / 2,
#         pv_champ_r / 2,
#     )

# rogue_mean = np.mean(rogue_sums)
# rogue_s_2 = sum((rogue_sums - rogue_mean) ** 2) / (samples - 1)
# print(rogue_mean, rogue_s_2)

# plt.figure(figsize=(7, 4))
# plt.scatter(
#     np.array([half_mean, cor_mean, doub_mean, rogue_mean]),
#     np.array([half_s_2, cor_s_2, doub_s_2, rogue_s_2]),
# )
# plt.title("variance and mean")
# plt.xlabel("mean")
# plt.ylabel("variance")
```

```
# plt.show()
```

Gaussian Process Regression on Final Prevalence Discrepency

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

num_samples = 30

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

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, seed=my_seed)
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])

LHC_samples = np.repeat(LHC_samples, 3, axis = 0)

random_indices_df = pd.DataFrame(samples, columns=variables_names)
```

```python
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  0.081511  0.004695  0.017172  0.007355  0.021370
1  0.332324  0.374497  0.003022  0.020210  0.001350  0.002604
2  0.836050  0.570164  0.002141  0.043572  0.001212  0.008367
3  0.566773  0.347186  0.001925  0.016830  0.000064  0.003145
4  0.880603  0.316884  0.000425  0.012374  0.000358  0.003491
      alpha      beta   gamma_L    lambda         f         r
0  0.066680  0.570582  0.001707  0.002226  0.004358  0.003743
1  0.066680  0.570582  0.001707  0.002226  0.004358  0.003743
2  0.066680  0.570582  0.001707  0.002226  0.004358  0.003743
3  0.132042  0.551592  0.013131  0.036829  0.002851  0.002075
4  0.132042  0.551592  0.013131  0.036829  0.002851  0.002075
```

Generate Discrepencies

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

print(random_discrepencies.head())
```

```
0    0.542551
1    0.627749
2    0.650314
3    0.644435
4    0.667979
dtype: float64
```

Differing Methods to Iterate Function

```
# import timeit

# def function1():
#     np.vectorize(champ_sum_stats)(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_sum_stats(
#             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

```python
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=1.0,
            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=1.0,
            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=1.0,
            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=1.0,
            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=1.0,
            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=1.0,
            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",
        # )
        self.bias_mean = tf.Variable(0.0, 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=1.0,
    dtype=np.float64,
    name="amplitude_champ",
)

observation_noise_variance_champ = tfp.util.TransformedVariable(
    bijector=constrain_positive,
```

```
    initial_value=0.03,
    dtype=np.float64,
    name="observation_noise_variance_champ",
)
```

```
length_scales_champ = tfp.util.TransformedVariable(
    bijector=constrain_positive,
    initial_value=[0.1, 0.1, 0.005, 0.04, 0.01, 0.02],
    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=0.01)
```

```
(<tf.Variable 'amplitude_champ:0' shape=() dtype=float64, numpy=0.0>, <tf.Variable 'length_sc
array([-2.30258509, -2.30258509, -5.29831737, -3.21887582, -4.60517019,
       -3.91202301])>, <tf.Variable 'observation_noise_variance_champ:0' shape=() dtype=floa
```

Train the Hyperparameters

```
# predictive log stuff
@tf.function(autograph=False, jit_compile=False)
def optimize():
    with tf.GradientTape() as tape:
        K = (
            champ_GP.kernel.matrix(index_vals, index_vals)
            + tf.eye(index_vals.shape[0], dtype=np.float64)
            * observation_noise_variance_champ
        )
        means = champ_GP.mean_fn(index_vals)
        K_inv = tf.linalg.inv(K)
        K_inv_y = K_inv @ tf.reshape(obs_vals - means, shape=[obs_vals.shape[0], 1])
        K_inv_diag = tf.linalg.diag_part(K_inv)
        log_var = tf.math.log(K_inv_diag)
        log_mu = tf.reshape(K_inv_y, shape=[-1]) ** 2
        loss = -tf.math.reduce_sum(log_var - log_mu)
    grads = tape.gradient(loss, champ_GP.trainable_variables)
    Adam_optim.apply_gradients(zip(grads, champ_GP.trainable_variables))
    return loss


num_iters = 10000

lls_ = np.zeros(num_iters, np.float64)
tolerance = 1e-6  # Set your desired tolerance level
previous_loss = float("inf")

for i in range(num_iters):
    loss = optimize()
    lls_[i] = loss

    # Check if change in loss is less than tolerance
    if abs(loss - previous_loss) < tolerance:
        print(f"Hyperparameter convergence reached at iteration {i+1}.")
        lls_ = lls_[range(i + 1)]
        break

    previous_loss = loss
```

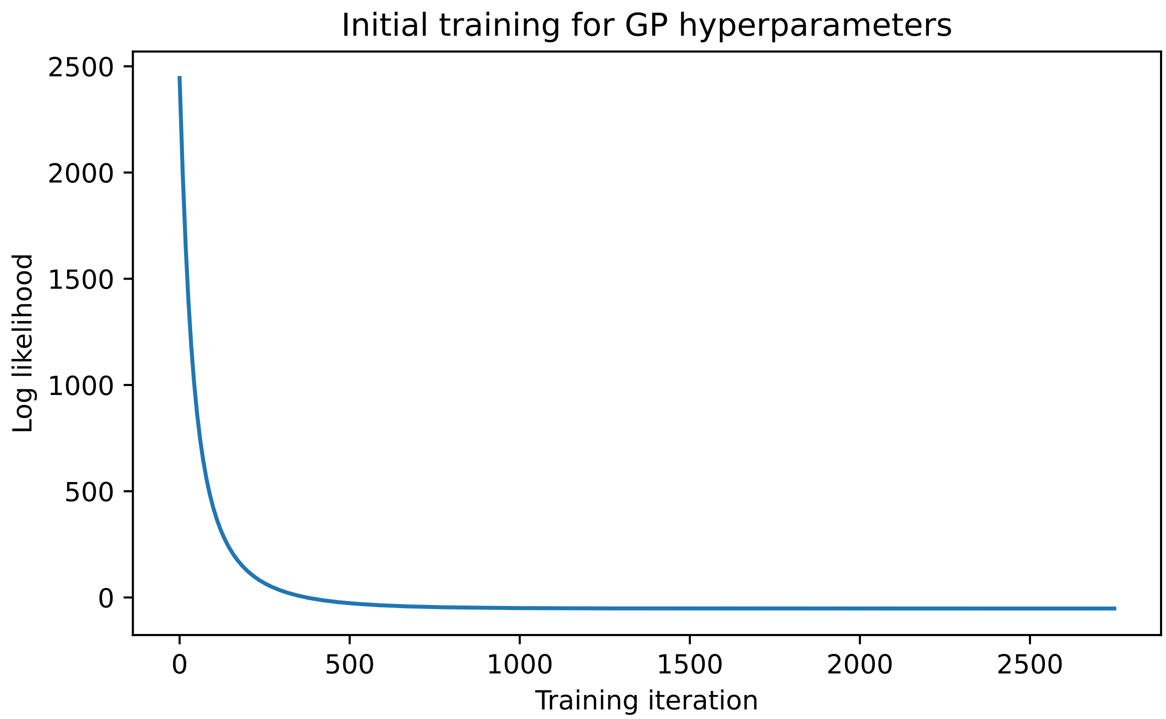
```
Hyperparameter convergence reached at iteration 2749.
```

```python
print("Trained parameters:")
for var in champ_GP.trainable_variables:
    if "tp" in var.name:  # or "bias" in var.name:
        print("{} is {}\n".format(var.name, var.numpy().round(3)))
    else:
        print(
            "{} is {}\n".format(
                var.name, constrain_positive.forward(var).numpy().round(3)
            )
        )
```

```
Trained parameters:
amplitude_champ:0 is 0.809

length_scales_champ:0 is [0.028 0.029 0.003 0.008 0.003 0.007]

observation_noise_variance_champ:0 is 0.239

alpha_tp:0 is -0.819

amp_alpha_mean:0 is 0.209

amp_beta_mean:0 is 1.256

amp_f_mean:0 is 1142.719

amp_gamma_L_mean:0 is 7.614

amp_lambda_mean:0 is 96.899

amp_r_mean:0 is 45.464

beta_tp:0 is 0.522

bias_mean:0 is 0.204

f_tp:0 is 0.016

gamma_L_tp:0 is -0.127

lambda_tp:0 is 0.041
```

```
r_tp:0 is 0.186
```

```
plt.figure(figsize=(7, 4))
plt.plot(lls_)
plt.title("Initial training for GP hyperparameters")
plt.xlabel("Training iteration")
plt.ylabel("Log likelihood")
plt.savefig("champagne_GP_images/hyperparam_loss.pdf")
plt.show()
```



Fitting the GP Regression across alpha

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

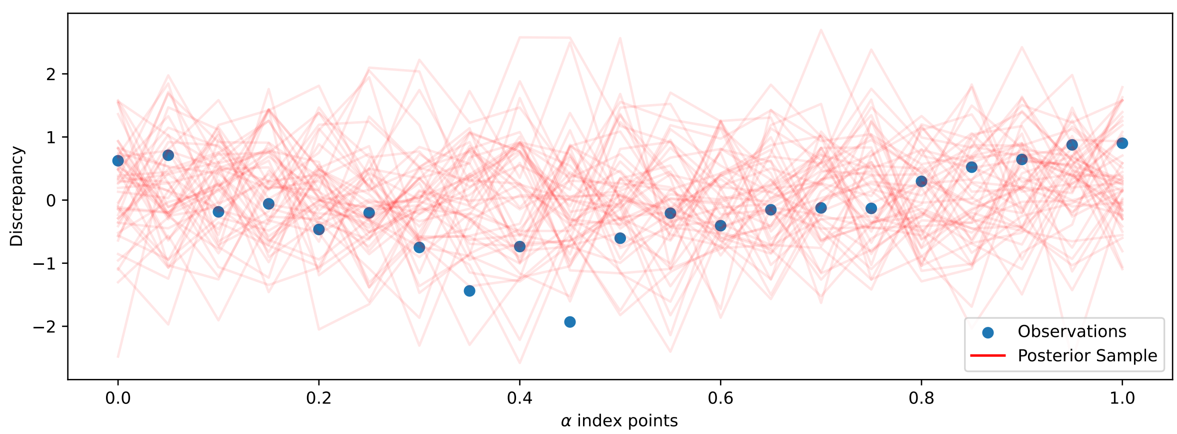
```
alpha_slice_samples = np.concatenate(
    (
        np.linspace(0, 1, plot_samp_no, dtype=np.float64).reshape(-1, 1),  # alpha
```

```
        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,
)

alpha_slice_indices_df = pd.DataFrame(alpha_slice_samples, columns=variables_names)

print(alpha_slice_indices_df.head())

alpha_slice_discrepencies = alpha_slice_indices_df.apply(
    lambda x: discrepency_fn(
        x["alpha"], x["beta"], x["gamma_L"], x["lambda"], x["f"], x["r"]
    ),
    axis=1,
)

alpha_slice_index_vals = alpha_slice_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
```

```
GP_seed = tfp.random.sanitize_seed(4362)

champ_GP_reg = tfd.GaussianProcessRegressionModel(
    kernel=kernel_champ,
    index_points=alpha_slice_index_vals,
    observation_index_points=index_vals,
    observations=obs_vals,
    observation_noise_variance=observation_noise_variance_champ,
    predictive_noise_variance=0.0,
    mean_fn=quad_mean_fn(),
)

GP_samples = champ_GP_reg.sample(gp_samp_no, seed=GP_seed)
```

```
plt.figure(figsize=(12, 4))
plt.scatter(
    alpha_slice_index_vals[:, 0], alpha_slice_discrepencies, label="Observations"
)
for i in range(gp_samp_no):
    plt.plot(
        alpha_slice_index_vals[:, 0],
        GP_samples[i, :],
        c="r",
        alpha=0.1,
        label="Posterior Sample" if i == 0 else None,
    )
leg = plt.legend(loc="lower right")
for lh in leg.legend_handles:
    lh.set_alpha(1)
plt.xlabel(r"$\alpha$ index points")
plt.ylabel("Discrepancy")
plt.savefig("champagne_GP_images/initial_alpha_slice.pdf")
plt.show()
```



Fitting the GP Regression across beta

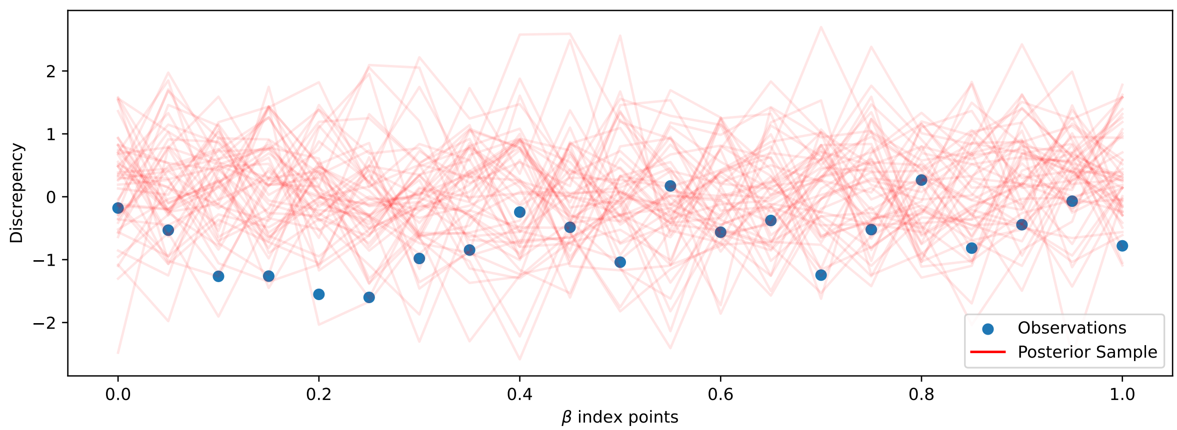
```
beta_slice_samples = np.concatenate(
    (
        np.repeat(pv_champ_alpha, plot_samp_no).reshape(-1, 1),  # alpha
```

```python
        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,
)

beta_slice_indices_df = pd.DataFrame(beta_slice_samples, columns=variables_names)

print(beta_slice_indices_df.head())

beta_slice_discrepencies = beta_slice_indices_df.apply(
    lambda x: discrepency_fn(
        x["alpha"], x["beta"], x["gamma_L"], x["lambda"], x["f"], x["r"]
    ),
    axis=1,
)

beta_slice_index_vals = beta_slice_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
```

```python
champ_GP_reg = tfd.GaussianProcessRegressionModel(
    kernel=kernel_champ,
    index_points=beta_slice_index_vals,
    observation_index_points=index_vals,
    observations=obs_vals,
    observation_noise_variance=observation_noise_variance_champ,
    predictive_noise_variance=0.0,
    mean_fn=quad_mean_fn(),
)

GP_samples = champ_GP_reg.sample(gp_samp_no, seed=GP_seed)
```

```python
plt.figure(figsize=(12, 4))
plt.scatter(beta_slice_index_vals[:, 1], beta_slice_discrepencies, label="Observations")
for i in range(gp_samp_no):
    plt.plot(
        beta_slice_index_vals[:, 1],
        GP_samples[i, :],
        c="r",
        alpha=0.1,
        label="Posterior Sample" if i == 0 else None,
    )
leg = plt.legend(loc="lower right")
for lh in leg.legend_handles:
    lh.set_alpha(1)
plt.xlabel(r"$\beta$ index points")
plt.ylabel("Discrepency")
plt.savefig("champagne_GP_images/initial_beta_slice.pdf")
plt.show()
```



Acquiring the next datapoint to test

Proof that .variance returns what we need in acquisition function

```python
new_guess = np.array([0.4, 0.4, 0.004, 0.04, 0.01, 0.17])
mean_t = champ_GP_reg.mean_fn(new_guess)
variance_t = champ_GP_reg.variance(index_points=[new_guess])
```

```
kernel_self = kernel_champ.apply(new_guess, new_guess)
kernel_others = kernel_champ.apply(new_guess, index_vals)
K = kernel_champ.matrix(
    index_vals, index_vals
) + observation_noise_variance_champ * np.identity(index_vals.shape[0])
inv_K = np.linalg.inv(K)
print("Self Kernel is {}".format(kernel_self.numpy().round(3)))
print("Others Kernel is {}".format(kernel_others.numpy().round(3)))
print(inv_K)
my_var_t = kernel_self - kernel_others.numpy() @ inv_K @ kernel_others.numpy()

print("Variance function is {}".format(variance_t.numpy().round(3)))
print("Variance function is {}".format(my_var_t.numpy().round(3)))
```

```
Self Kernel is 0.655
Others Kernel is [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[[ 2.93820932e+00 -1.24213810e+00 -1.24213810e+00 ...  1.54407072e-89
   1.54407072e-89  1.54407072e-89]
 [-1.24213810e+00  2.93820932e+00 -1.24213810e+00 ...  1.54407072e-89
   1.54407072e-89  1.54407072e-89]
 [-1.24213810e+00 -1.24213810e+00  2.93820932e+00 ...  1.54407072e-89
   1.54407072e-89  1.54407072e-89]
 ...
 [ 1.54407072e-89  1.54407072e-89  1.54407072e-89 ...  2.93820932e+00
  -1.24213810e+00 -1.24213810e+00]
 [ 1.54407072e-89  1.54407072e-89  1.54407072e-89 ... -1.24213810e+00
   2.93820932e+00 -1.24213810e+00]
 [ 1.54407072e-89  1.54407072e-89  1.54407072e-89 ... -1.24213810e+00
  -1.24213810e+00  2.93820932e+00]]
Variance function is [0.655]
Variance function is 0.655
```

Loss function

```
next_alpha = tfp.util.TransformedVariable(
    initial_value=0.5,
    bijector=tfb.Sigmoid(),
```

```
    dtype=np.float64,
    name="next_alpha",
)

next_beta = tfp.util.TransformedVariable(
    initial_value=0.5,
    bijector=tfb.Sigmoid(),
    dtype=np.float64,
    name="next_beta",
)

next_gamma_L = tfp.util.TransformedVariable(
    initial_value=0.1,
    bijector=constrain_positive,
    dtype=np.float64,
    name="next_gamma_L",
)

next_lambda = tfp.util.TransformedVariable(
    initial_value=0.1,
    bijector=constrain_positive,
    dtype=np.float64,
    name="next_lambda",
)

next_f = tfp.util.TransformedVariable(
    initial_value=0.1,
    bijector=constrain_positive,
    dtype=np.float64,
    name="next_f",
)

next_r = tfp.util.TransformedVariable(
    initial_value=0.1,
    bijector=constrain_positive,
    dtype=np.float64,
    name="next_r",
)

next_vars = [
    v.trainable_variables[0]
    for v in [next_alpha, next_beta, next_gamma_L, next_lambda, next_f, next_r]
```

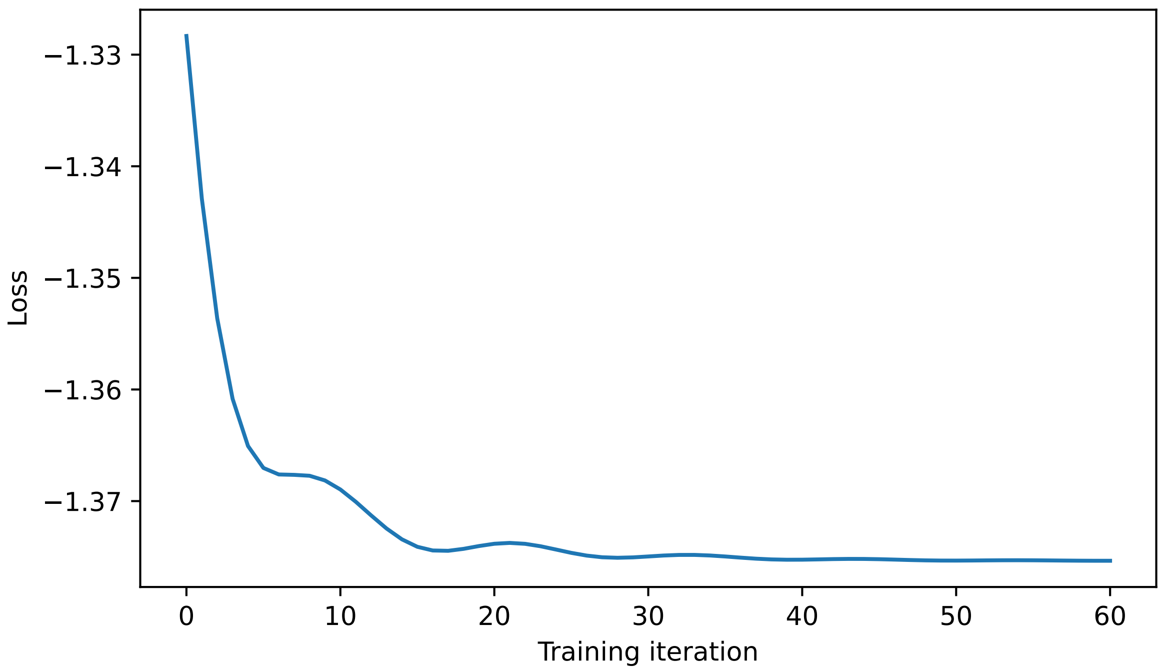
```
]
```

```
Adam_optim = tf.optimizers.Adam(learning_rate=0.1)


@tf.function(autograph=False, jit_compile=False)
def optimize():
    with tf.GradientTape() as tape:
        next_guess = tf.reshape(
            [
                tfb.Sigmoid().forward(next_vars[0]),
                tfb.Sigmoid().forward(next_vars[1]),
                constrain_positive.forward(next_vars[2]),
                constrain_positive.forward(next_vars[3]),
                constrain_positive.forward(next_vars[4]),
                constrain_positive.forward(next_vars[5]),
            ],
            [1, 6],
        )
        mean_t = champ_GP_reg.mean_fn(next_guess)
        std_t = champ_GP_reg.stddev(index_points=next_guess)
        loss = tf.squeeze(mean_t - 1.7 * std_t)
    grads = tape.gradient(loss, next_vars)
    Adam_optim.apply_gradients(zip(grads, next_vars))
    return loss


num_iters = 10000

lls_ = np.zeros(num_iters, np.float64)
tolerance = 1e-6  # Set your desired tolerance level
previous_loss = float("inf")

for i in range(num_iters):
    loss = optimize()
    lls_[i] = loss

    # Check if change in loss is less than tolerance
    if abs(loss - previous_loss) < tolerance:
        print(f"Acquisition function convergence reached at iteration {i+1}.")
        lls_ = lls_[range(i + 1)]
        break
```

```python
    previous_loss = loss

print("Trained parameters:")
for var in next_vars:
    if ("alpha" in var.name) | ("beta" in var.name):
        print(
            "{} is {}".format(var.name, (tfb.Sigmoid().forward(var).numpy().round(3)))
        )
    else:
        print(
            "{} is {}".format(
                var.name, constrain_positive.forward(var).numpy().round(3)
            )
        )
```

```
Acquisition function convergence reached at iteration 61.
Trained parameters:
next_alpha:0 is 0.402
next_beta:0 is 0.402
next_gamma_L:0 is 0.012
next_lambda:0 is 0.042
next_f:0 is 0.015
next_r:0 is 0.016
```

```python
plt.figure(figsize=(7, 4))
plt.plot(lls_)
plt.xlabel("Training iteration")
plt.ylabel("Loss")
plt.savefig("champagne_GP_images/bolfi_optim_loss.pdf")
plt.show()
```




```
def update_GP():
    @tf.function
    def opt_GP():
        with tf.GradientTape() as tape:
            K = (
                champ_GP.kernel.matrix(index_vals, index_vals)
                + tf.eye(index_vals.shape[0], dtype=np.float64)
                * observation_noise_variance_champ
            )
            means = champ_GP.mean_fn(index_vals)
            K_inv = tf.linalg.inv(K)
            K_inv_y = K_inv @ tf.reshape(obs_vals - means, shape=[obs_vals.shape[0], 1])
            K_inv_diag = tf.linalg.diag_part(K_inv)
            log_var = tf.math.log(K_inv_diag)
            log_mu = tf.reshape(K_inv_y, shape=[-1]) ** 2
            loss = -tf.math.reduce_sum(log_var - log_mu)
        grads = tape.gradient(loss, champ_GP.trainable_variables)
        optimizer_slow.apply_gradients(zip(grads, champ_GP.trainable_variables))
        return loss

    num_iters = 10000
```

```
    lls_ = np.zeros(num_iters, np.float64)
    tolerance = 1e-6  # Set your desired tolerance level
    previous_loss = float("inf")

    for i in range(num_iters):
        loss = opt_GP()
        lls_[i] = loss.numpy()

        # Check if change in loss is less than tolerance
        if abs(loss - previous_loss) < tolerance:
            print(f"Hyperparameter convergence reached at iteration {i+1}.")
            lls_ = lls_[range(i + 1)]
            break

        previous_loss = loss
    for var in optimizer_slow.variables:
        var.assign(tf.zeros_like(var))


def update_var():
    @tf.function
    def opt_var():
        with tf.GradientTape() as tape:
            next_guess = tf.reshape(
                [
                    tfb.Sigmoid().forward(next_vars[0]),
                    tfb.Sigmoid().forward(next_vars[1]),
                    tfb.Sigmoid().forward(next_vars[2]),
                    tfb.Sigmoid().forward(next_vars[3]),
                    tfb.Sigmoid().forward(next_vars[4]),
                    tfb.Sigmoid().forward(next_vars[5]),
                ],
                [1, 6],
            )
            mean_t = champ_GP_reg.mean_fn(next_guess)
            std_t = champ_GP_reg.stddev(index_points=next_guess)
            loss = tf.squeeze(mean_t - eta_t * std_t)
        grads = tape.gradient(loss, next_vars)
        optimizer_fast.apply_gradients(zip(grads, next_vars))
        return loss

    num_iters = 10000
```

```
    lls_ = np.zeros(num_iters, np.float64)
    tolerance = 1e-6  # Set your desired tolerance level
    previous_loss = float("inf")

    for i in range(num_iters):
        loss = opt_var()
        lls_[i] = loss

        # Check if change in loss is less than tolerance
        if abs(loss - previous_loss) < tolerance:
            print(f"Acquisition function convergence reached at iteration {i+1}.")
            lls_ = lls_[range(i + 1)]
            break

        previous_loss = loss
    print(loss)
    for var in optimizer_fast.variables:
        var.assign(tf.zeros_like(var))


def new_eta_t(t, d, exploration_rate):
    return np.sqrt(np.log((t + 1) ** (d / 2 + 2) * np.pi**2 / (3 * exploration_rate)))


exploration_rate = 0.1
d = 6
update_freq = 20  # how many iterations before updating GP hyperparams

for t in range(45):
    next_vars[0].assign(0)
    optimizer_fast = tf.optimizers.Adam(learning_rate=0.01)
    optimizer_slow = tf.optimizers.Adam()
    eta_t = new_eta_t(t, d, exploration_rate)
    print(t)
    new_discrepency = discrepency_fn(
        next_alpha.numpy(),
        next_beta.numpy(),
        next_gamma_L.numpy(),
        next_lambda.numpy(),
        next_f.numpy(),
        next_r.numpy(),
    )
```

```
    index_vals = np.append(
        index_vals,
        np.array(
            [
                next_alpha.numpy(),
                next_beta.numpy(),
                next_gamma_L.numpy(),
                next_lambda.numpy(),
                next_f.numpy(),
                next_r.numpy(),
            ]
        ).reshape(1, -1),
        axis=0,
    )
    obs_vals = np.append(obs_vals, new_discrepency)

    if t % update_freq == 0:
        champ_GP = tfd.GaussianProcess(
            kernel=kernel_champ,
            observation_noise_variance=observation_noise_variance_champ,
            index_points=index_vals,
            mean_fn=quad_mean_fn(),
        )
        update_GP()

    champ_GP_reg = tfd.GaussianProcessRegressionModel(
        kernel=kernel_champ,
        index_points=alpha_slice_index_vals,
        observation_index_points=index_vals,
        observations=obs_vals,
        observation_noise_variance=observation_noise_variance_champ,
        predictive_noise_variance=0.0,
        mean_fn=quad_mean_fn(),
    )
    update_var()


# print(index_vals[-200,])
print(index_vals[-20,])
print(index_vals[-2,])
print(index_vals[-1,])
```

```
0
Acquisition function convergence reached at iteration 323.
tf.Tensor(-1.9992982196470983, shape=(), dtype=float64)
1
Acquisition function convergence reached at iteration 82.
tf.Tensor(-2.594167704225814, shape=(), dtype=float64)
2
Acquisition function convergence reached at iteration 47.
tf.Tensor(-2.730453103233291, shape=(), dtype=float64)
3
Acquisition function convergence reached at iteration 42.
tf.Tensor(-2.945534010279511, shape=(), dtype=float64)
4
Acquisition function convergence reached at iteration 51.
tf.Tensor(-3.095649667447467, shape=(), dtype=float64)
5
Acquisition function convergence reached at iteration 1265.
tf.Tensor(-3.217096649839165, shape=(), dtype=float64)
6
Acquisition function convergence reached at iteration 53.
tf.Tensor(-3.3109447413308795, shape=(), dtype=float64)
7
Acquisition function convergence reached at iteration 59.
tf.Tensor(-3.3972225330239985, shape=(), dtype=float64)
8
Acquisition function convergence reached at iteration 37.
tf.Tensor(-3.4652177117928002, shape=(), dtype=float64)
9
Acquisition function convergence reached at iteration 1726.
tf.Tensor(-3.5312034052476466, shape=(), dtype=float64)
10
Acquisition function convergence reached at iteration 1345.
tf.Tensor(-3.586678091343197, shape=(), dtype=float64)
11
Acquisition function convergence reached at iteration 1629.
tf.Tensor(-3.6367781237035626, shape=(), dtype=float64)
12
Acquisition function convergence reached at iteration 1783.
tf.Tensor(-3.682265814831364, shape=(), dtype=float64)
13
Acquisition function convergence reached at iteration 1703.
tf.Tensor(-3.723850744253507, shape=(), dtype=float64)
14
```

```
Acquisition function convergence reached at iteration 1766.
tf.Tensor(-3.762116246332952, shape=(), dtype=float64)
15
Acquisition function convergence reached at iteration 1782.
tf.Tensor(-3.7975578964546566, shape=(), dtype=float64)
16
Acquisition function convergence reached at iteration 1790.
tf.Tensor(-3.8305462102193535, shape=(), dtype=float64)
17
Acquisition function convergence reached at iteration 1458.
tf.Tensor(-3.8612786156897845, shape=(), dtype=float64)
18
Acquisition function convergence reached at iteration 1722.
tf.Tensor(-3.8904581467886, shape=(), dtype=float64)
19
Acquisition function convergence reached at iteration 1885.
tf.Tensor(-3.9185416080133213, shape=(), dtype=float64)
20
Hyperparameter convergence reached at iteration 8795.
Acquisition function convergence reached at iteration 50.
tf.Tensor(-4.332877333675094, shape=(), dtype=float64)
21
Acquisition function convergence reached at iteration 53.
tf.Tensor(-4.359794521357593, shape=(), dtype=float64)
22
Acquisition function convergence reached at iteration 53.
tf.Tensor(-4.385363989795709, shape=(), dtype=float64)
23
Acquisition function convergence reached at iteration 53.
tf.Tensor(-4.409694500710833, shape=(), dtype=float64)
24
Acquisition function convergence reached at iteration 55.
tf.Tensor(-4.432888421005889, shape=(), dtype=float64)
25
Acquisition function convergence reached at iteration 55.
tf.Tensor(-4.455061748348836, shape=(), dtype=float64)
26
Acquisition function convergence reached at iteration 56.
tf.Tensor(-4.476281952743918, shape=(), dtype=float64)
27
Acquisition function convergence reached at iteration 57.
tf.Tensor(-4.496624238704728, shape=(), dtype=float64)
28
```

```
Acquisition function convergence reached at iteration 58.
tf.Tensor(-4.516153377881663, shape=(), dtype=float64)
29
Acquisition function convergence reached at iteration 59.
tf.Tensor(-4.534929843529165, shape=(), dtype=float64)
30
Acquisition function convergence reached at iteration 60.
tf.Tensor(-4.553002643915425, shape=(), dtype=float64)
31
Acquisition function convergence reached at iteration 61.
tf.Tensor(-4.57041617512142, shape=(), dtype=float64)
32
Acquisition function convergence reached at iteration 61.
tf.Tensor(-4.587220711583741, shape=(), dtype=float64)
33
Acquisition function convergence reached at iteration 60.
tf.Tensor(-4.603448400482354, shape=(), dtype=float64)
34
Acquisition function convergence reached at iteration 59.
tf.Tensor(-4.619133825107237, shape=(), dtype=float64)
35
Acquisition function convergence reached at iteration 58.
tf.Tensor(-4.634306603897773, shape=(), dtype=float64)
36
Acquisition function convergence reached at iteration 58.
tf.Tensor(-4.64899685896621, shape=(), dtype=float64)
37
Acquisition function convergence reached at iteration 56.
tf.Tensor(-4.6632371934212875, shape=(), dtype=float64)
38
Acquisition function convergence reached at iteration 55.
tf.Tensor(-4.677050843202696, shape=(), dtype=float64)
39
Acquisition function convergence reached at iteration 53.
tf.Tensor(-4.69046142209625, shape=(), dtype=float64)
40
Hyperparameter convergence reached at iteration 9344.
Acquisition function convergence reached at iteration 1517.
tf.Tensor(-5.287241552898761, shape=(), dtype=float64)
41
Acquisition function convergence reached at iteration 73.
tf.Tensor(-5.299146604237107, shape=(), dtype=float64)
42
```

```
Acquisition function convergence reached at iteration 71.
tf.Tensor(-5.313398598850882, shape=(), dtype=float64)
43
Acquisition function convergence reached at iteration 70.
tf.Tensor(-5.327113697264764, shape=(), dtype=float64)
44
Acquisition function convergence reached at iteration 71.
tf.Tensor(-5.340433821581583, shape=(), dtype=float64)
[5.00000000e-01 3.99823632e-01 4.70129397e-04 3.63982790e-02
 7.13809954e-04 1.29266685e-02]
[0.5        0.39995475 0.00464514 0.04753066 0.01588468 0.01839059]
[0.5        0.40002586 0.00463635 0.04274685 0.01702272 0.01912081]
```

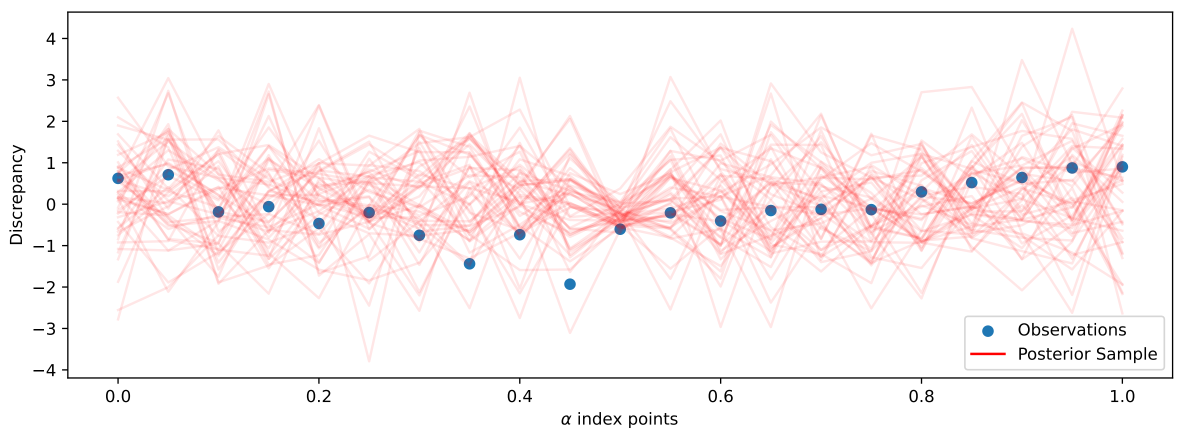
Fitting the GP Regression across alpha

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

```
GP_seed = tfp.random.sanitize_seed(4362)

champ_GP_reg = tfd.GaussianProcessRegressionModel(
    kernel=kernel_champ,
    index_points=alpha_slice_index_vals,
    observation_index_points=index_vals,
    observations=obs_vals,
    observation_noise_variance=observation_noise_variance_champ,
    predictive_noise_variance=0.0,
    mean_fn=quad_mean_fn(),
)

GP_samples = champ_GP_reg.sample(gp_samp_no, seed=GP_seed)
```

```
plt.figure(figsize=(12, 4))
plt.scatter(
    alpha_slice_index_vals[:, 0], alpha_slice_discrepencies, label="Observations"
)
for i in range(gp_samp_no):
    plt.plot(
        alpha_slice_index_vals[:, 0],
        GP_samples[i, :],
```

```
        c="r",
        alpha=0.1,
        label="Posterior Sample" if i == 0 else None,
    )
leg = plt.legend(loc="lower right")
for lh in leg.legend_handles:
    lh.set_alpha(1)
plt.xlabel(r"$\alpha$ index points")
plt.ylabel("Discrepancy")
plt.savefig("champagne_GP_images/new_alpha_slice.pdf")
plt.show()
```



Fitting the GP Regression across beta

```
champ_GP_reg = tfd.GaussianProcessRegressionModel(
    kernel=kernel_champ,
    index_points=beta_slice_index_vals,
    observation_index_points=index_vals,
    observations=obs_vals,
    observation_noise_variance=observation_noise_variance_champ,
    predictive_noise_variance=0.0,
    mean_fn=quad_mean_fn(),
)

GP_samples = champ_GP_reg.sample(gp_samp_no, seed=GP_seed)
```

```
plt.figure(figsize=(12, 4))
plt.scatter(beta_slice_index_vals[:, 1], beta_slice_discrepencies, label="Observations")
for i in range(gp_samp_no):
    plt.plot(
        beta_slice_index_vals[:, 1],
        GP_samples[i, :],
        c="r",
        alpha=0.1,
        label="Posterior Sample" if i == 0 else None,
    )
leg = plt.legend(loc="lower right")
for lh in leg.legend_handles:
    lh.set_alpha(1)
plt.xlabel(r"$\beta$ index points")
plt.ylabel("Discrepency")
plt.savefig("champagne_GP_images/new_beta_slice.pdf")
plt.show()
```