Inference on the Champagne Model using a Gaussian Process

TODO

- Change outputs

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
from scipy.stats import norm

import tensorflow as tf
import tensorflow_probability as tfp
from tensorflow_probability.python.distributions import normal

tfb = tfp.bijectors
tfd = tfp.distributions
tfk = tfp.math.psd_kernels
tfp_acq = tfp.experimental.bayesopt.acquisition
```

Model itself

```
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
```

```python
    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)
    i = 0

    while (i < num_events) or (t < 30):
        i += 1
        if S_0 == N:
            while t < 31:
                t += 1
                new_stages = {
                    "t": t,
                    "S_0": N,
                    "S_L": 0,
                    "I_0": 0,
                    "I_L": 0,
                    "inc_counter": inc_counter,
                }
                list_of_outcomes.append(new_stages)
            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
        )
```

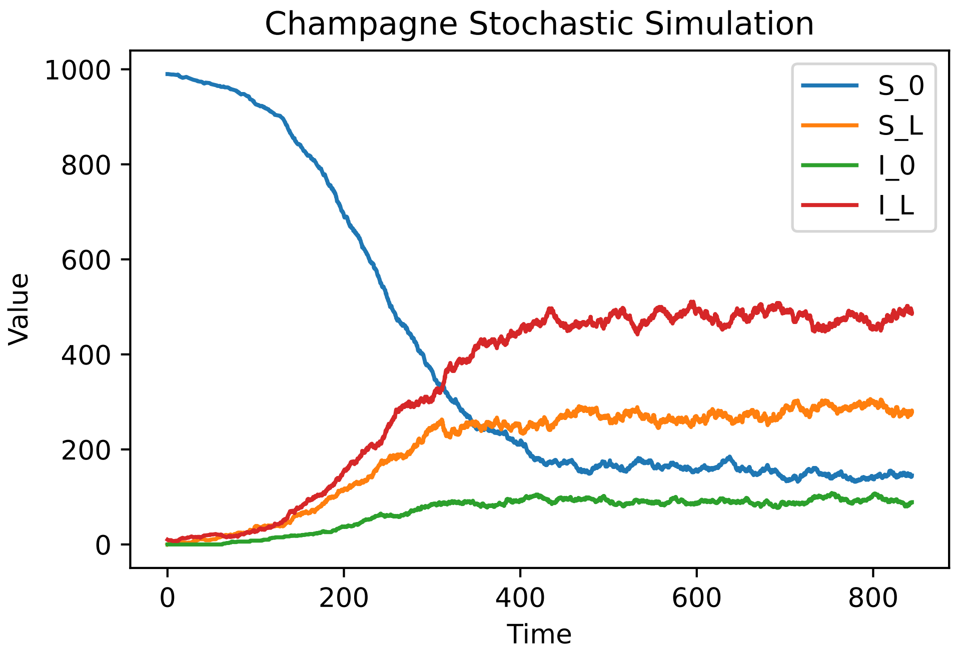
```
        delta_t = np.random.exponential(1 / total_rate)
        new_stages_prob = [
            S_0_to_I_L / total_rate,
            S_0_to_S_L / total_rate,
            I_0_to_S_0 / total_rate,
            I_0_to_I_L / total_rate,
            I_L_to_I_0 / total_rate,
            I_L_to_S_L / total_rate,
            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.sum(np.abs((x - observed_sum_stats) / observed_sum_stats))
    # return np.linalg.norm((x - observed_sum_stats) / observed_sum_stats)
    return np.log(np.linalg.norm((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 = 100

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)
```

```python
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  0.947868  0.001360  0.024440  0.053912  0.016944
1  0.332324  0.098249  0.001562  0.009264  0.030982  0.005292
2  0.836050  0.528836  0.007612  0.038457  0.015414  0.006343
3  0.566773  0.363482  0.007795  0.007177  0.002909  0.011431
4  0.880603  0.278997  0.003764  0.020626  0.023896  0.010783
      alpha      beta   gamma_L    lambda         f         r
0  0.370004  0.951175  0.003733  0.125161  0.022409  0.009974
1  0.370004  0.951175  0.003733  0.125161  0.022409  0.009974
2  0.370004  0.951175  0.003733  0.125161  0.022409  0.009974
3  0.959612  0.815478  0.012922  0.000021  0.000649  0.008105
4  0.959612  0.815478  0.012922  0.000021  0.000649  0.008105
```

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    1.743364
1    2.264622
2    2.033671
3    0.539962
4    0.549306
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())
```

```
2024-05-02 18:27:56.124188: I external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:9
2024-05-02 18:27:56.162873: W tensorflow/core/common_runtime/gpu/gpu_device.cc:2251] Cannot d
Skipping registering GPU devices...
```

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=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=0.5,
        #     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 = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=1.0,
            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 = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=1.0,
            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 = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=1.0,
            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 = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=1.0,
            dtype=np.float64,
            name="r_tp",
        )
        # self.bias_mean = tfp.util.TransformedVariable(
        #     bijector=constrain_positive,
        #     initial_value=1.0,
        #     dtype=np.float64,
        #     name="bias_mean",
        # )
        self.bias_mean = tf.Variable(-2.0, dtype=np.float64, name="bias_mean")

    def __call__(self, x):
        return (
            self.bias_mean
            # + 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
```

```
        )


quad_mean_fn().__call__(x=np.array([[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]]))  # should return 1
```

```
<tf.Tensor: shape=(1,), dtype=float64, numpy=array([-2.])>
```

Custom Linear Mean Function

```
class lin_mean_fn(tf.Module):
    def __init__(self):
        super(lin_mean_fn, self).__init__()
        # self.amp_alpha_lin = tfp.util.TransformedVariable(
        #     bijector=constrain_positive,
        #     initial_value=1.0,
        #     dtype=np.float64,
        #     name="amp_alpha_lin",
        # )
        # self.amp_beta_lin = tfp.util.TransformedVariable(
        #     bijector=constrain_positive,
        #     initial_value=0.5,
        #     dtype=np.float64,
        #     name="amp_beta_lin",
        # )
        self.amp_gamma_L_lin = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=1.0,
            dtype=np.float64,
            name="amp_gamma_L_lin",
        )
        self.amp_lambda_lin = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=1.0,
            dtype=np.float64,
            name="amp_lambda_lin",
        )
        self.amp_f_lin = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=1.0,
            dtype=np.float64,
```

```
            name="amp_f_lin",
        )
        self.amp_r_lin = tfp.util.TransformedVariable(
            bijector=constrain_positive,
            initial_value=1.0,
            dtype=np.float64,
            name="amp_r_lin",
        )
        # self.bias_lin = tfp.util.TransformedVariable(
        #     bijector=constrain_positive,
        #     initial_value=1.0,
        #     dtype=np.float64,
        #     name="bias_lin",
        # )
        self.bias_lin = tf.Variable(0.0, dtype=np.float64, name="bias_mean")

    def __call__(self, x):
        return (
            self.bias_lin
            # + self.amp_alpha_lin * (x[..., 0])
            # + self.amp_beta_lin * (x[..., 1])
            + self.amp_gamma_L_lin * (x[..., 2])
            + self.amp_lambda_lin * (x[..., 3])
            + self.amp_f_lin * (x[..., 4])
            + self.amp_r_lin * (x[..., 5])
        )
```

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=1.,
    dtype=np.float64,
    name="observation_noise_variance_champ",
)
```

```
length_scales_champ = tfp.util.TransformedVariable(
    bijector=constrain_positive,
    initial_value=[1., 1., 1., 1., 1., 1.],
    dtype=np.float64,
    name="length_scales_champ",
)
```

```
kernel_champ = tfk.FeatureScaled(
    tfk.MaternFiveHalves(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_s
```

Train the Hyperparameters

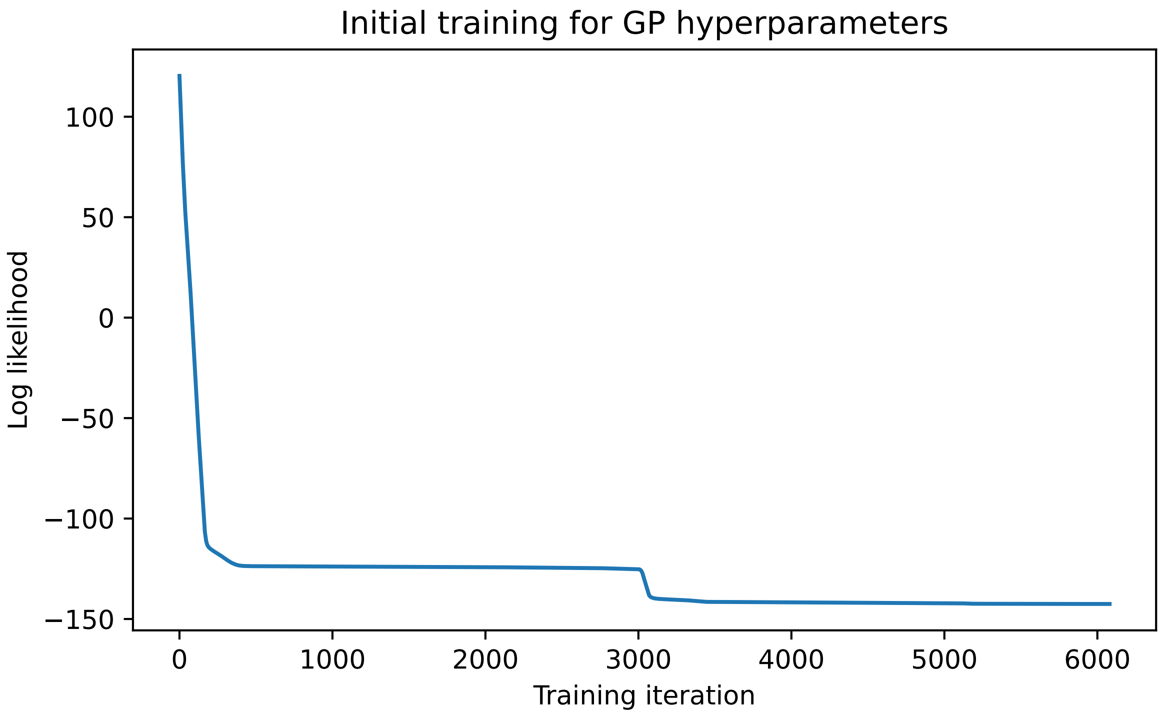
```python
# predictive log stuff
@tf.function(autograph=False, jit_compile=False)
def optimize():
    with tf.GradientTape() as tape:
        K = (
            champ_GP.kernel.matrix(index_vals, index_vals)
            + tf.eye(index_vals.shape[0], dtype=np.float64)
            * observation_noise_variance_champ
        )
        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 6081.
```

```python
print("Trained parameters:")
for var in champ_GP.trainable_variables:
    if '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 0.967

length_scales_champ:0 is [4.2000e-01 6.6905e+01 2.1000e-02 5.0000e-02 4.7600e+00 1.1000e-02]

observation_noise_variance_champ:0 is 0.307

amp_f_mean:0 is 144.824

amp_gamma_L_mean:0 is 842.901

amp_lambda_mean:0 is 111.254

amp_r_mean:0 is 11.458

bias_mean:0 is -6.616

f_tp:0 is 0.049

gamma_L_tp:0 is 0.029

lambda_tp:0 is 0.08

r_tp:0 is 0.802
```

```python
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_log_discrep.pdf")
plt.show()
```



Creating slices across one variable dimension

```
plot_samp_no = 21
plot_gp_no = 200
gp_samp_no = 50
```

```
slice_samples_dict = {
    "alpha_slice_samples": np.repeat(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,
    ), 3, axis = 0),
    "alpha_gp_samples": np.concatenate(
```

```
            (
                np.linspace(0, 1, plot_gp_no, dtype=np.float64).reshape(-1, 1),  # alpha
                np.repeat(pv_champ_beta, plot_gp_no).reshape(-1, 1),  # beta
                np.repeat(pv_champ_gamma_L, plot_gp_no).reshape(-1, 1),  # gamma_L
                np.repeat(pv_champ_lambda, plot_gp_no).reshape(-1, 1),  # lambda
                np.repeat(pv_champ_f, plot_gp_no).reshape(-1, 1),  # f
                np.repeat(pv_champ_r, plot_gp_no).reshape(-1, 1),  # r
            ),
            axis=1,
        ),
        "beta_slice_samples": np.repeat(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,
        ), 3, axis = 0),
        "beta_gp_samples": np.concatenate(
            (
                np.repeat(pv_champ_alpha, plot_gp_no).reshape(-1, 1),  # alpha
                np.linspace(0, 1, plot_gp_no, dtype=np.float64).reshape(-1, 1),  # beta
                np.repeat(pv_champ_gamma_L, plot_gp_no).reshape(-1, 1),  # gamma_L
                np.repeat(pv_champ_lambda, plot_gp_no).reshape(-1, 1),  # lambda
                np.repeat(pv_champ_f, plot_gp_no).reshape(-1, 1),  # f
                np.repeat(pv_champ_r, plot_gp_no).reshape(-1, 1),  # r
            ),
            axis=1,
        ),
        "gamma_L_slice_samples": np.repeat(np.concatenate(
            (
                np.repeat(pv_champ_alpha, plot_samp_no).reshape(-1, 1),  # alpha
                np.repeat(pv_champ_beta, plot_samp_no).reshape(-1, 1),  # beta
                -10*pv_champ_gamma_L
                * np.log(
                    np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
                ).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,
    ), 3, axis = 0),
    "gamma_L_gp_samples": np.concatenate(
        (
            np.repeat(pv_champ_alpha, plot_gp_no).reshape(-1, 1),  # alpha
            np.repeat(pv_champ_beta, plot_gp_no).reshape(-1, 1),  # beta
            np.linspace(
                -10*pv_champ_gamma_L
                * np.log(
                    np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
                ).reshape(-1, 1)[0],
                -10*pv_champ_gamma_L
                * np.log(
                    np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
                ).reshape(-1, 1)[-1], plot_gp_no, dtype=np.float64
            ),  # gamma_L
            np.repeat(pv_champ_lambda, plot_gp_no).reshape(-1, 1),  # lambda
            np.repeat(pv_champ_f, plot_gp_no).reshape(-1, 1),  # f
            np.repeat(pv_champ_r, plot_gp_no).reshape(-1, 1),  # r
        ),
        axis=1,
    ),
    "lambda_slice_samples": np.repeat(np.concatenate(
        (
            np.repeat(pv_champ_alpha, plot_samp_no).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
            -pv_champ_lambda
            * np.log(
                np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
            ).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,
    ), 3, axis = 0),
```

```
    "lambda_gp_samples": np.concatenate(
        (
            np.repeat(pv_champ_alpha, plot_gp_no).reshape(-1, 1),  # alpha
            np.repeat(pv_champ_beta, plot_gp_no).reshape(-1, 1),  # beta
            np.repeat(pv_champ_gamma_L, plot_gp_no).reshape(-1, 1),  # gamma_L
            np.linspace(
                -pv_champ_lambda
                * np.log(
                    np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
                ).reshape(-1, 1)[0],
                -pv_champ_lambda
                * np.log(
                    np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
                ).reshape(-1, 1)[-1], plot_gp_no, dtype=np.float64
            ),  # lambda
            np.repeat(pv_champ_f, plot_gp_no).reshape(-1, 1),  # f
            np.repeat(pv_champ_r, plot_gp_no).reshape(-1, 1),  # r
        ),
        axis=1,
    ),
    "f_slice_samples": np.repeat(np.concatenate(
        (
            np.repeat(pv_champ_alpha, plot_samp_no).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
            -10*pv_champ_f
            * np.log(
                np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
            ).reshape(
                -1, 1
            ),  # f
            np.repeat(pv_champ_r, plot_samp_no).reshape(-1, 1),  # r
        ),
        axis=1,
    ), 3, axis = 0),
    "f_gp_samples": np.concatenate(
        (
            np.repeat(pv_champ_alpha, plot_gp_no).reshape(-1, 1),  # alpha
            np.repeat(pv_champ_beta, plot_gp_no).reshape(-1, 1),  # beta
            np.repeat(pv_champ_gamma_L, plot_gp_no).reshape(-1, 1),  # gamma_L
            np.repeat(pv_champ_lambda, plot_gp_no).reshape(-1, 1),  # lambda
```

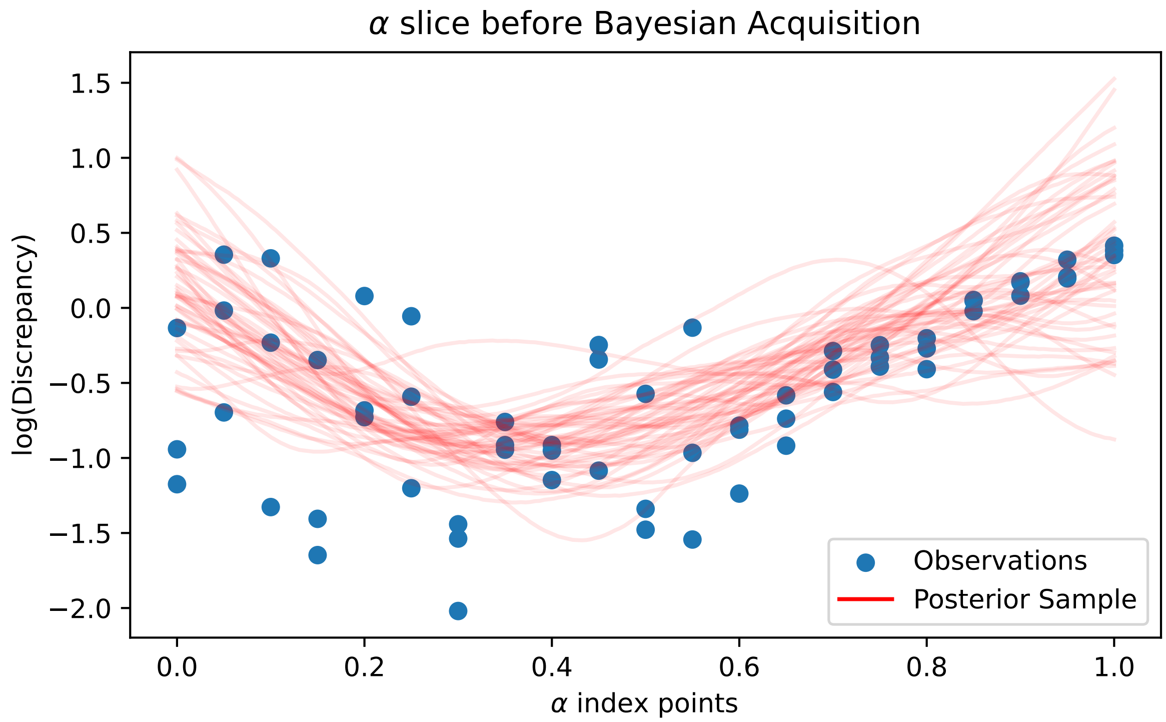
```
            np.linspace(
                -10*pv_champ_f
                * np.log(
                    np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
                ).reshape(-1, 1)[0],
                -10*pv_champ_f
                * np.log(
                    np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
                ).reshape(-1, 1)[-1], plot_gp_no, dtype=np.float64
            ),  # f
            np.repeat(pv_champ_r, plot_gp_no).reshape(-1, 1),  # r
        ),
        axis=1,
    ),
    "r_slice_samples": np.repeat(np.concatenate(
        (
            np.repeat(pv_champ_alpha, plot_samp_no).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
            -2*pv_champ_r
            * np.log(
                np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
            ).reshape(
                -1, 1
            ),  # r
        ),
        axis=1,
    ), 3, axis = 0),
    "r_gp_samples": np.concatenate(
        (
            np.repeat(pv_champ_alpha, plot_gp_no).reshape(-1, 1),  # alpha
            np.repeat(pv_champ_beta, plot_gp_no).reshape(-1, 1),  # beta
            np.repeat(pv_champ_gamma_L, plot_gp_no).reshape(-1, 1),  # gamma_L
            np.repeat(pv_champ_lambda, plot_gp_no).reshape(-1, 1),  # lambda
            np.repeat(pv_champ_f, plot_gp_no).reshape(-1, 1),  # f
            np.linspace(
                -2*pv_champ_r
                * np.log(
                    np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
                ).reshape(-1, 1)[0],
```

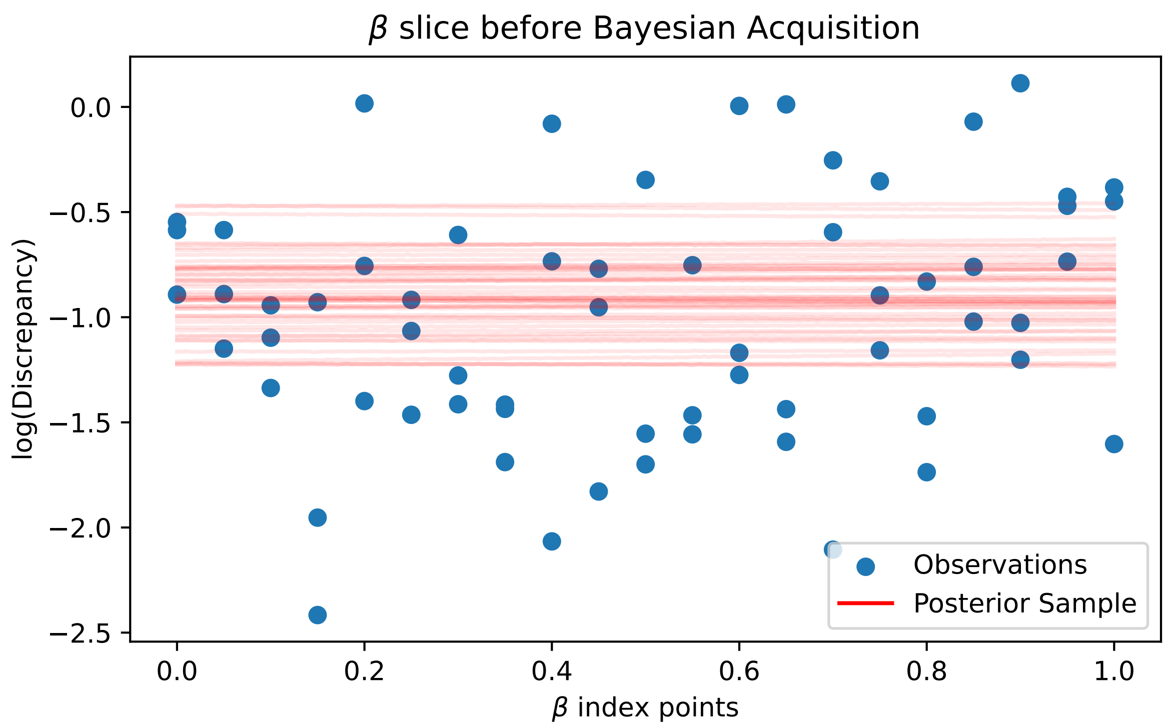
```
                -2*pv_champ_r
                * np.log(
                    np.linspace(0, 1, plot_samp_no + 2, dtype=np.float64)[1:-1]
                ).reshape(-1, 1)[-1], plot_gp_no, dtype=np.float64
            ),  # r
        ),
        axis=1,
    ),
}
```

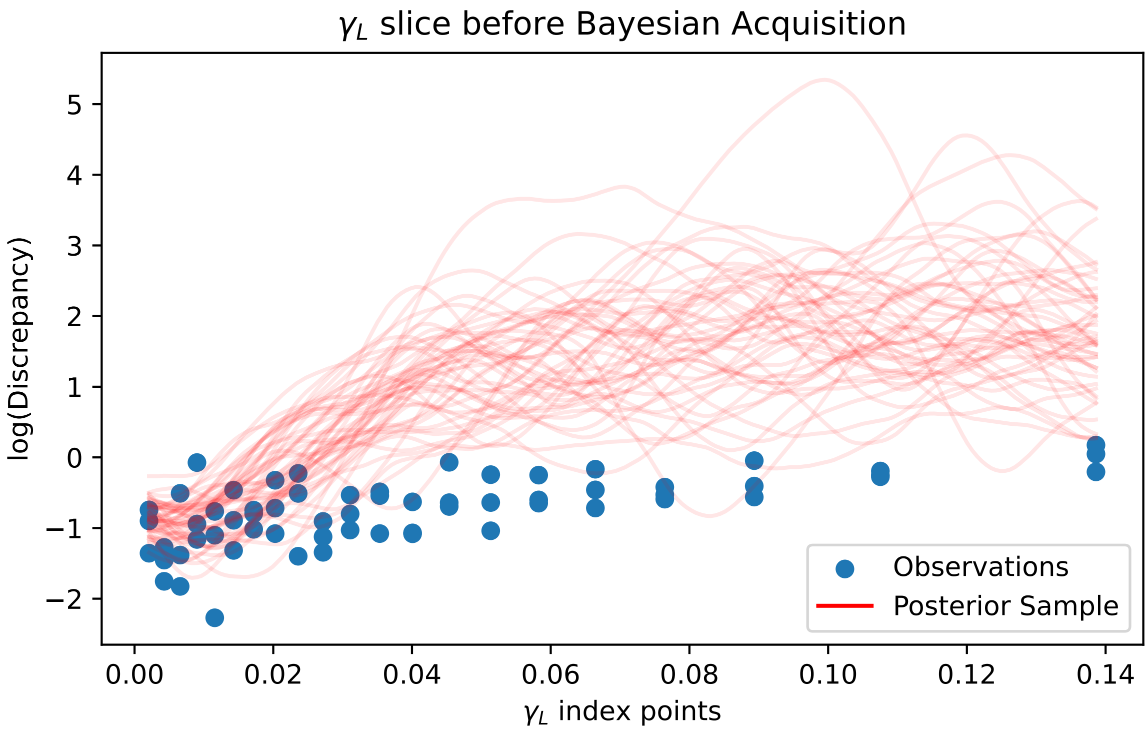
Plotting the GPs across different slices

```
GP_seed = tfp.random.sanitize_seed(4362)
vars = ["alpha", "beta", "gamma_L", "lambda", "f", "r"]
slice_indices_dfs_dict = {}
slice_index_vals_dict = {}
slice_discrepencies_dict = {}

for var in vars:
    val_df = pd.DataFrame(
        slice_samples_dict[var + "_slice_samples"], columns=variables_names
    )
    slice_indices_dfs_dict[var + "_slice_indices_df"] = val_df
    slice_index_vals_dict[var + "_slice_index_vals"] = val_df.values
    discreps = val_df.apply(
        lambda x: discrepency_fn(
            x["alpha"], x["beta"], x["gamma_L"], x["lambda"], x["f"], x["r"]
        ),
        axis=1,
    )
    slice_discrepencies_dict[var + "_slice_discrepencies"] = discreps


    gp_samples_df = pd.DataFrame(
        slice_samples_dict[var + "_gp_samples"], columns=variables_names
    )
    slice_indices_dfs_dict[var + "_gp_indices_df"] = gp_samples_df
    slice_index_vals_dict[var + "_gp_index_vals"] = gp_samples_df.values

    champ_GP_reg = tfd.GaussianProcessRegressionModel(
```

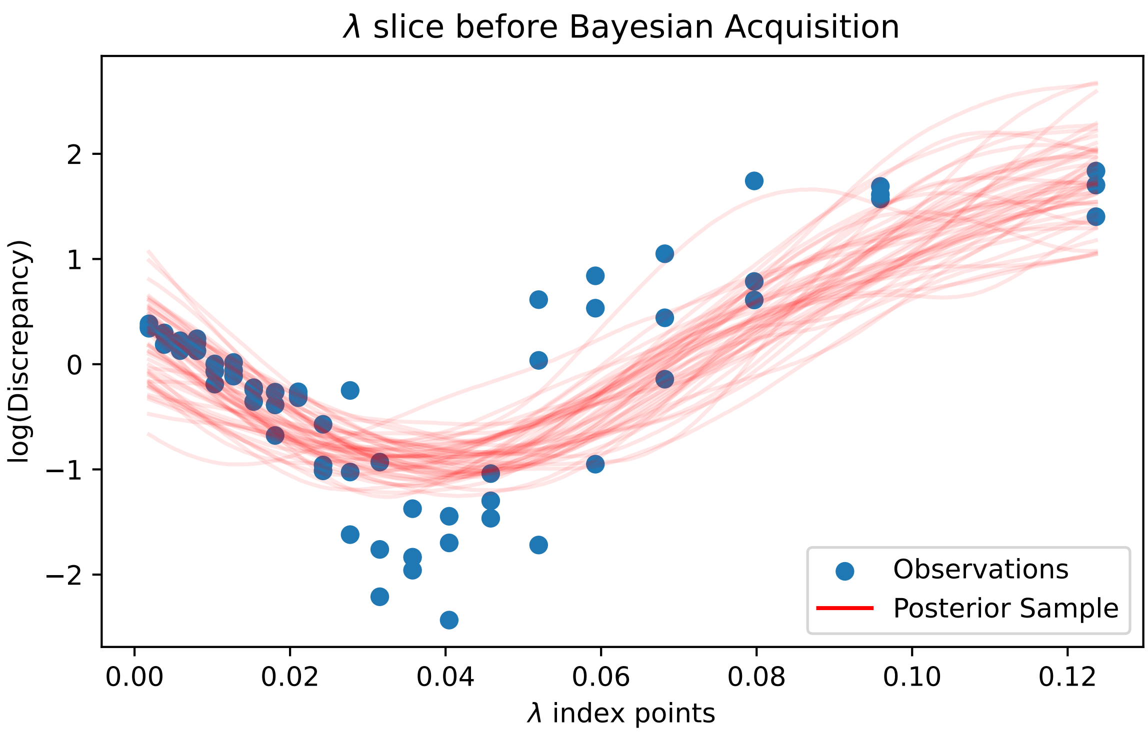
```
        kernel=kernel_champ,
        index_points=gp_samples_df.values,
        observation_index_points=index_vals,
        observations=obs_vals,
        observation_noise_variance=observation_noise_variance_champ,
        predictive_noise_variance=0.0,
        mean_fn=quad_mean_fn(),
    )
    GP_samples = champ_GP_reg.sample(gp_samp_no, seed=GP_seed)

    plt.figure(figsize=(7, 4))
    plt.scatter(
        val_df[var].values,
        discreps,
        label="Observations",
    )
    for i in range(gp_samp_no):
        plt.plot(
            gp_samples_df[var].values,
            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)
    if var in ["f", "r"]:
        plt.xlabel("$" + var + "$ index points")
        plt.title("$" + var + "$ slice before Bayesian Acquisition")
    else:
        plt.xlabel("$\\" + var + "$ index points")
        plt.title("$\\" + var + "$ slice before Bayesian Acquisition")
    # if var not in ["alpha", "beta"]:
    #     plt.xscale("log", base=np.e)
    plt.ylabel("log(Discrepancy)")
    plt.savefig("champagne_GP_images/initial_" + var + "_slice_log_discrep.pdf")
    plt.show()
```


α slice before Bayesian Acquisition
log(Discrepancy)
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.0
0.2
0.4
0.6
0.8
1.0
α index points
Observations
Posterior Sample

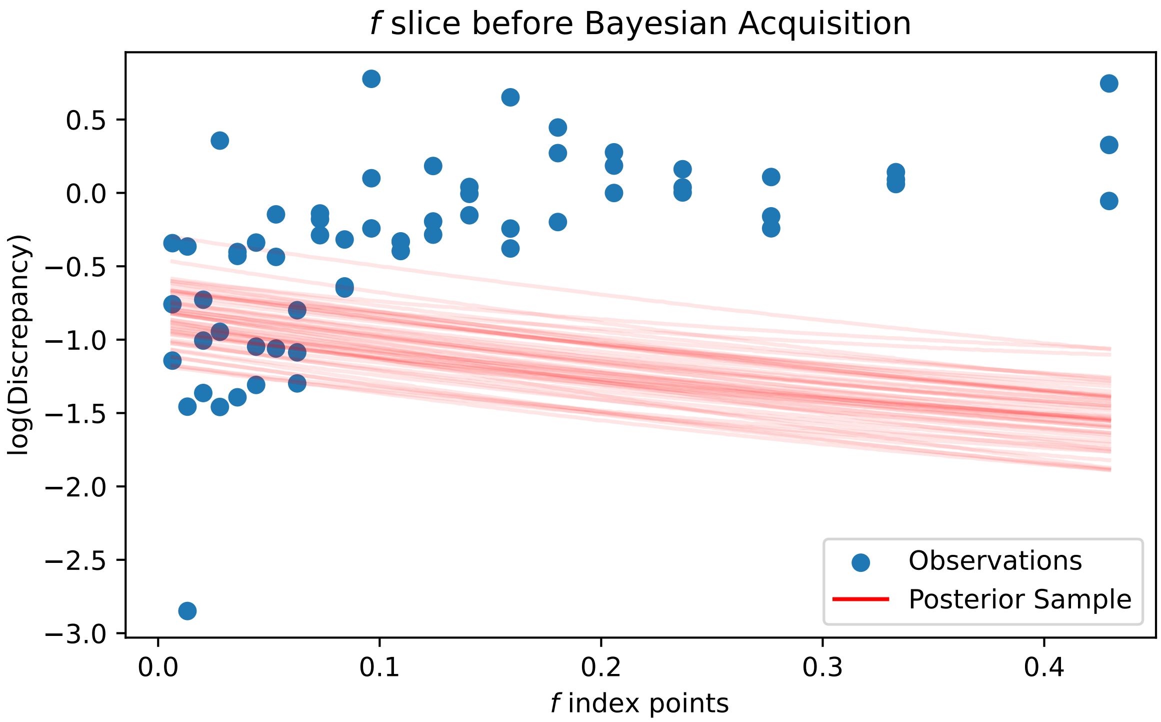

β slice before Bayesian Acquisition
log(Discrepancy)
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0.0
0.2
0.4
0.6
0.8
1.0
β index points
Observations
Posterior Sample

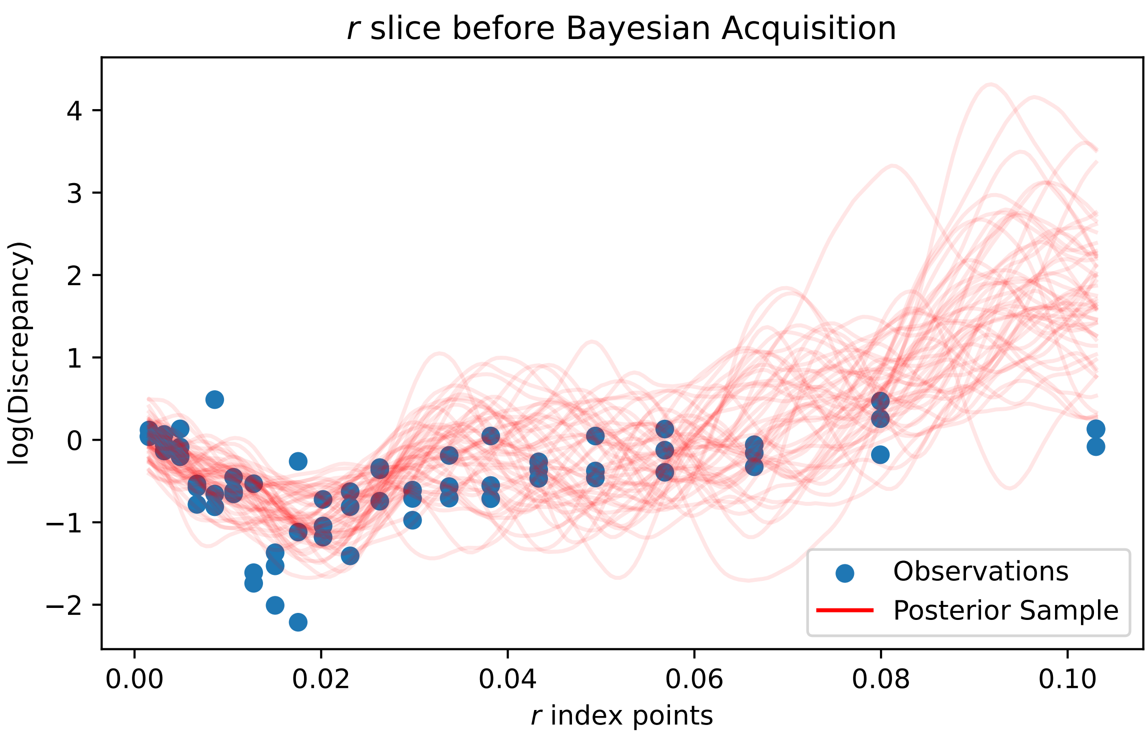


γ_L slice before Bayesian Acquisition
log(Discrepancy)
γ_L index points
Observations
Posterior Sample



λ slice before Bayesian Acquisition
log(Discrepancy)
λ index points
Observations
Posterior Sample


f slice before Bayesian Acquisition
log(Discrepancy)
f index points
Observations
Posterior Sample


r slice before Bayesian Acquisition
log(Discrepancy)
r index points
Observations
Posterior Sample

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 0.935
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. 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. 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. 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.]
[[ 2.37467979e+00 -8.80988637e-01 -8.80988637e-01 ...  1.24986038e-03
   1.24986038e-03  1.24986038e-03]
 [-8.80988637e-01  2.37467979e+00 -8.80988637e-01 ...  1.24986038e-03
   1.24986038e-03  1.24986038e-03]
 [-8.80988637e-01 -8.80988637e-01  2.37467979e+00 ...  1.24986038e-03
```

```
  1.24986038e-03  1.24986038e-03]
 ...
 [ 1.24986038e-03  1.24986038e-03  1.24986038e-03 ...  2.74935129e+00
  -5.06317135e-01 -5.06317135e-01]
 [ 1.24986038e-03  1.24986038e-03  1.24986038e-03 ... -5.06317135e-01
   2.74935129e+00 -5.06317135e-01]
 [ 1.24986038e-03  1.24986038e-03  1.24986038e-03 ... -5.06317135e-01
  -5.06317135e-01  2.74935129e+00]]
Variance function is [0.935]
Variance function is 0.935
```

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=tfb.Sigmoid(),
    dtype=np.float64,
    name="next_gamma_L",
)

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

```python
next_f = tfp.util.TransformedVariable(
    initial_value=0.1,
    bijector=tfb.Sigmoid(),
    dtype=np.float64,
    name="next_f",
)

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

next_vars = (
    (next_alpha.trainable_variables[0],
    next_beta.trainable_variables[0],
    next_gamma_L.trainable_variables[0],
    next_lambda.trainable_variables[0],
    next_f.trainable_variables[0],
    next_r.trainable_variables[0],)
)

next_vars
```

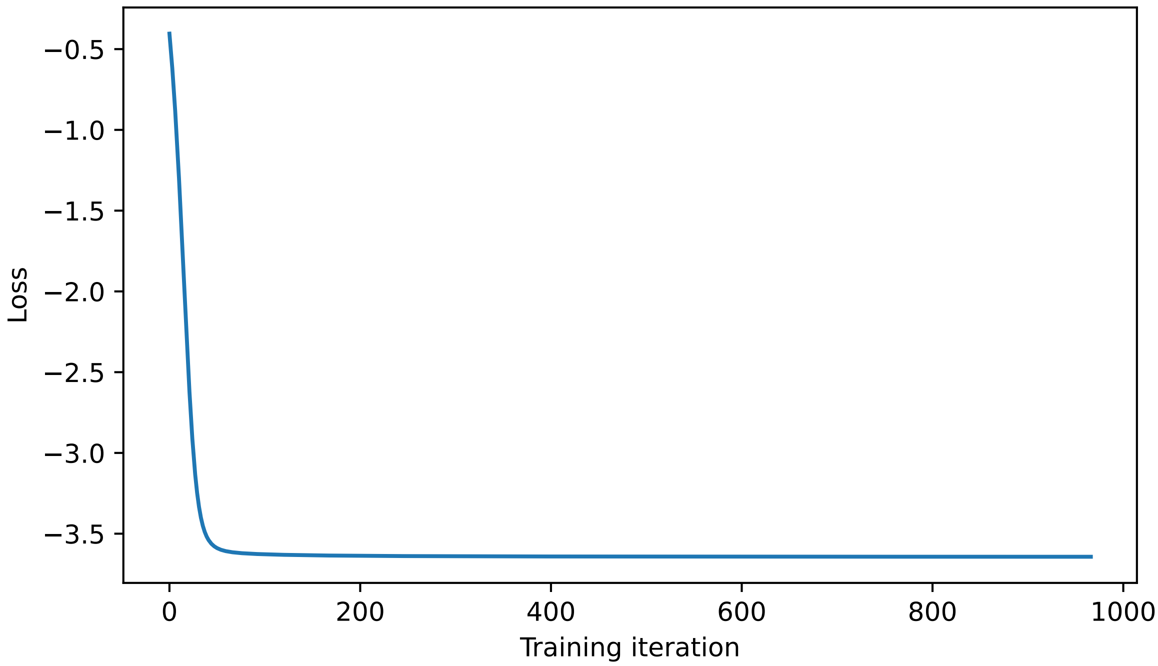
```
(<tf.Variable 'next_alpha:0' shape=() dtype=float64, numpy=0.0>,
 <tf.Variable 'next_beta:0' shape=() dtype=float64, numpy=0.0>,
 <tf.Variable 'next_gamma_L:0' shape=() dtype=float64, numpy=-2.197224577336219>,
 <tf.Variable 'next_lambda:0' shape=() dtype=float64, numpy=-2.197224577336219>,
 <tf.Variable 'next_f:0' shape=() dtype=float64, numpy=-2.197224577336219>,
 <tf.Variable 'next_r:0' shape=() dtype=float64, numpy=-2.197224577336219>)
```

```python
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(
            tf.stack(
                [next_alpha, next_beta, next_gamma_L, next_lambda, next_f, next_r]
```

```
            ),
            [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

    previous_loss = loss

print("Trained parameters:")
for var in next_vars:
    print("{} is {}".format(var.name, (tfb.Sigmoid().forward(var).numpy().round(3))))
# 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 967.
Trained parameters:
next_alpha:0 is 0.372
next_beta:0 is 0.5
next_gamma_L:0 is 0.988
next_lambda:0 is 0.988
next_f:0 is 0.988
next_r:0 is 0.988
```

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




```
def update_GP():
    @tf.function(autograph=False, jit_compile=False)
    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_UCB():
    optimizer_fast = tf.optimizers.Adam(learning_rate=1.0)

    @tf.function(autograph=False, jit_compile=False)
    def opt_var():
        with tf.GradientTape() as tape:
            next_guess = tf.reshape(
```

```
                tf.stack(
                    [next_alpha, next_beta, next_gamma_L, next_lambda, next_f, next_r]
                ),
                [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

    next_guess = tf.reshape(
        tf.stack([next_alpha, next_beta, next_gamma_L, next_lambda, next_f, next_r]),
        [1, 6],
    )
    print(
        "The final UCB loss was {}".format(loss.numpy().round(3))
        + " with predicted mean of {}".format(
            champ_GP_reg.mean_fn(next_guess).numpy().round(3)
        )
    )
    for var in optimizer_fast.variables:
        var.assign(tf.zeros_like(var))
```

```python
def update_var_EI():
    optimizer_fast = tf.optimizers.Adam(learning_rate=1.0)

    @tf.function(autograph=False, jit_compile=False)
    def opt_var():
        with tf.GradientTape() as tape:
            next_guess = tf.reshape(
                tf.stack(
                    [next_alpha, next_beta, next_gamma_L, next_lambda, next_f, next_r]
                ),
                [1, 6],
            )
            mean_t = champ_GP_reg.mean_fn(next_guess)
            std_t = champ_GP_reg.stddev(index_points=next_guess)
            delt = min_obs - mean_t
            loss = -tf.squeeze(
                delt * tfd.Normal(0, std_t).cdf(delt)
                + std_t * champ_GP_reg.prob(delt, index_points=next_guess)
            )
        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-9  # Set your desired tolerance level
    previous_loss = np.float64("inf")

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

        # Check if change in loss is less than tolerance
        if (i > 200) and (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))


# EI = tfp_acq.GaussianProcessExpectedImprovement(champ_GP_reg, obs_vals)


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

```
# optimizer_fast = tf.optimizers.Adam(learning_rate=1.)
# update_var_EI()
# plt.figure(figsize=(7, 4))
# plt.plot(lls_)
# plt.xlabel("Training iteration")
# plt.ylabel("Loss")
# plt.show()
```

```
exploration_rate = 0.00001
d = 6
update_freq = 20  # how many iterations before updating GP hyperparams
eta_t = tf.Variable(0, dtype=np.float64, name="eta_t")
min_obs = tf.Variable(100, dtype=np.float64, name="min_obs", shape=())
min_index = index_vals[
    champ_GP_reg.mean_fn(index_vals) == min(champ_GP_reg.mean_fn(index_vals))
][
    0,
]

for t in range(201):
    min_index = index_vals[
        champ_GP_reg.mean_fn(index_vals) == min(champ_GP_reg.mean_fn(index_vals))
    ][
        0,
    ]
    optimizer_slow = tf.optimizers.Adam()
    eta_t.assign(new_eta_t(t, d, exploration_rate))
    min_obs.assign(min(champ_GP_reg.mean_fn(index_vals)))
    print("Iteration " + str(t))
    # print(eta_t)
```

```
######################################################################
var_num = 0

for var in next_vars:
    if ("alpha" in var.name) or ("beta" in var.name):
        var.assign(tfb.Sigmoid().inverse(np.float64(np.random.uniform())))
    else:
        var.assign(tfb.Sigmoid().inverse(np.float64(np.random.uniform())))
    var_num += 1

update_var_UCB()
# update_var_EI()
# print(next_vars)

new_params = np.array(
    [
        next_alpha.numpy(),
        next_beta.numpy(),
        next_gamma_L.numpy(),
        next_lambda.numpy(),
        next_f.numpy(),
        next_r.numpy(),
    ]
).reshape(1, -1)
print("The next parameters to simulate from are {}".format(new_params.round(3)))

for repeats in range(2):
    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,
        new_params,
        axis=0,
    )
    obs_vals = np.append(obs_vals, new_discrepency)
```

```
####################################################################
# var_num = 0

# for var in next_vars:
#     if ('alpha' in var.name) or ('beta' in var.name):
#         var.assign(tfb.Sigmoid().inverse(min_index[var_num]))
#     else:
#         var.assign(constrain_positive.inverse(min_index[var_num]))
#     var_num += 1

# # for var in next_vars:
# #     if ('alpha' in var.name) or ('beta' in var.name):
# #         var.assign(tfb.Sigmoid().inverse(np.float64(np.random.uniform())))
# #     else:
# #         var.assign(constrain_positive.inverse(np.float64(np.random.uniform())))
# #     var_num += 1

# update_var_UCB()
# # update_var_EI()
# # print(next_vars)

# new_params = np.array(
#     [
#         next_alpha.numpy(),
#         next_beta.numpy(),
#         next_gamma_L.numpy(),
#         next_lambda.numpy(),
#         next_f.numpy(),
#         next_r.numpy(),
#     ]
# ).reshape(1, -1)
# print(new_params)

# for repeats in range(2):
#     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,
#         new_params,
#         axis=0,
#     )
#     obs_vals = np.append(obs_vals, new_discrepency)
######################################################################
if (t+1) % 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,
    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(),
)

if (t > 0) & (t % 50 == 0):
    print("Trained parameters:")
    for train_var in champ_GP.trainable_variables:
        print(
            "{} is {}\n".format(
                train_var.name,
                tfb.Sigmoid().forward(train_var).numpy().round(3),
            )
        )
    # if "length" in train_var.name:
    #     print(
    #         "{} is {}\n".format(
    #             train_var.name,
    #             tfb.Sigmoid().forward(train_var).numpy().round(3),
    #         )
    #     )
    # else:
```

```
#     if "tp" in train_var.name:  # or "bias" in var.name:
#         print(
#             "{} is {}\n".format(train_var.name, train_var.numpy().round(3))
#         )
#     else:
#         print(
#             "{} is {}\n".format(
#                 train_var.name,
#                 constrain_positive.forward(train_var).numpy().round(3),
#             )
#         )
for var in vars:
    champ_GP_reg = tfd.GaussianProcessRegressionModel(
        kernel=kernel_champ,
        index_points=slice_indices_dfs_dict[var + "_gp_indices_df"].values,
        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=(7, 4))
    plt.scatter(
        slice_indices_dfs_dict[var + "_slice_indices_df"][var].values,
        slice_discrepencies_dict[var + "_slice_discrepencies"],
        label="Observations",
    )
    for i in range(gp_samp_no):
        plt.plot(
            slice_indices_dfs_dict[var + "_gp_indices_df"][var].values,
            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)
    if var in ["f", "r"]:
        plt.xlabel("$" + var + "$ index points")
```

```python
                plt.title(
                    "$" + var + "$ slice after " + str(t) + " Bayesian acquisitions"
                )
            else:
                plt.xlabel("$\\" + var + "$ index points")
                plt.title(
                    "$\\" + var + "$ slice after " + str(t) + " Bayesian acquisitions"
                )
            plt.ylabel("log(Discrepancy)")
            plt.savefig(
                "champagne_GP_images/"
                + var
                + "_slice_"
                + str(t)
                + "_bolfi_updates_log_discrep.pdf"
            )
            plt.show()
```

```
Iteration 0
Acquisition function convergence reached at iteration 17.
The final UCB loss was -5.446 with predicted mean of [-2.]
The next parameters to simulate from are [[0.252 0.58  0.999 0.999 1.    0.999]]
Iteration 1
Acquisition function convergence reached at iteration 90.
The final UCB loss was -5.883 with predicted mean of [-1.995]
The next parameters to simulate from are [[1.    0.981 0.999 0.988 1.    0.939]]
Iteration 2
Acquisition function convergence reached at iteration 114.
The final UCB loss was -6.11 with predicted mean of [-1.986]
The next parameters to simulate from are [[0.    0.005 0.991 0.989 1.    0.888]]
Iteration 3
Acquisition function convergence reached at iteration 95.
The final UCB loss was -6.27 with predicted mean of [-1.986]
The next parameters to simulate from are [[0.996 0.044 0.894 0.999 1.    1.   ]]
Iteration 4
Acquisition function convergence reached at iteration 97.
The final UCB loss was -6.374 with predicted mean of [-1.971]
The next parameters to simulate from are [[0.997 0.985 0.999 0.984 1.    0.837]]
Iteration 5
Acquisition function convergence reached at iteration 93.
The final UCB loss was -6.485 with predicted mean of [-1.986]
The next parameters to simulate from are [[0.    0.989 0.91  1.    1.    0.947]]
```

```
Iteration 6
Acquisition function convergence reached at iteration 104.
The final UCB loss was -6.532 with predicted mean of [-1.953]
The next parameters to simulate from are [[1.    0.    1.    0.818 1.    0.971]]
Iteration 7
Acquisition function convergence reached at iteration 72.
The final UCB loss was -6.499 with predicted mean of [-1.854]
The next parameters to simulate from are [[0.    0.998 1.    1.    1.    0.959]]
Iteration 8
Acquisition function convergence reached at iteration 96.
The final UCB loss was -6.664 with predicted mean of [-1.959]
The next parameters to simulate from are [[0.    0.016 0.81  0.989 1.    1.   ]]
Iteration 9
Acquisition function convergence reached at iteration 97.
The final UCB loss was -6.707 with predicted mean of [-1.95]
The next parameters to simulate from are [[0.983 1.    0.998 0.998 1.    0.782]]
Iteration 10
Acquisition function convergence reached at iteration 139.
The final UCB loss was -6.731 with predicted mean of [-1.927]
The next parameters to simulate from are [[0.009 0.003 0.999 0.995 1.    0.736]]
Iteration 11
Acquisition function convergence reached at iteration 112.
The final UCB loss was -6.794 with predicted mean of [-1.948]
The next parameters to simulate from are [[0.    0.011 0.92  0.825 1.    0.997]]
Iteration 12
Acquisition function convergence reached at iteration 37.
The final UCB loss was -6.775 with predicted mean of [-1.89]
The next parameters to simulate from are [[0.001 0.996 1.    0.728 1.    1.   ]]
Iteration 13
Acquisition function convergence reached at iteration 122.
The final UCB loss was -6.629 with predicted mean of [-1.712]
The next parameters to simulate from are [[0.    0.    0.927 1.    1.    0.999]]
Iteration 14
Acquisition function convergence reached at iteration 78.
The final UCB loss was -6.635 with predicted mean of [-1.684]
The next parameters to simulate from are [[1.    1.    0.97  0.861 1.    1.   ]]
Iteration 15
Acquisition function convergence reached at iteration 80.
The final UCB loss was -6.955 with predicted mean of [-1.972]
The next parameters to simulate from are [[0.999 0.001 0.885 0.975 1.    0.895]]
Iteration 16
Acquisition function convergence reached at iteration 131.
The final UCB loss was -6.968 with predicted mean of [-1.957]
```

```
The next parameters to simulate from are [[1.    0.985 0.817 0.998 1.    0.943]]
Iteration 17
Acquisition function convergence reached at iteration 94.
The final UCB loss was -7.0 with predicted mean of [-1.962]
The next parameters to simulate from are [[0.001 0.993 0.902 0.975 1.    0.845]]
Iteration 18
Acquisition function convergence reached at iteration 294.
The final UCB loss was -7.006 with predicted mean of [-1.943]
The next parameters to simulate from are [[0.001 0.999 0.802 0.987 1.    0.89 ]]
Iteration 19
Acquisition function convergence reached at iteration 305.
The final UCB loss was -7.027 with predicted mean of [-1.941]
The next parameters to simulate from are [[0.992 0.993 0.894 0.988 1.    0.794]]
Iteration 20
Acquisition function convergence reached at iteration 123.
The final UCB loss was -18.065 with predicted mean of [-1.861]
The next parameters to simulate from are [[1.    0.59  0.799 0.984 0.999 0.706]]
Iteration 21
Acquisition function convergence reached at iteration 79.
The final UCB loss was -18.035 with predicted mean of [-1.763]
The next parameters to simulate from are [[1.    0.737 0.999 0.584 1.    0.89 ]]
Iteration 22
Acquisition function convergence reached at iteration 103.
The final UCB loss was -18.171 with predicted mean of [-1.835]
The next parameters to simulate from are [[0.998 0.235 0.62  0.973 0.999 1.   ]]
Iteration 23
Acquisition function convergence reached at iteration 155.
The final UCB loss was -18.073 with predicted mean of [-1.676]
The next parameters to simulate from are [[0.004 0.685 0.453 0.981 0.998 0.999]]
Iteration 24
Acquisition function convergence reached at iteration 163.
The final UCB loss was -18.316 with predicted mean of [-1.86]
The next parameters to simulate from are [[0.    0.5   1.    0.982 0.994 0.637]]
Iteration 25
Acquisition function convergence reached at iteration 183.
The final UCB loss was -18.294 with predicted mean of [-1.782]
The next parameters to simulate from are [[1.    0.987 0.992 0.995 1.    0.544]]
Iteration 26
Acquisition function convergence reached at iteration 64.
The final UCB loss was -17.568 with predicted mean of [-1.003]
The next parameters to simulate from are [[0.992 0.899 0.001 1.    0.997 0.999]]
Iteration 27
Acquisition function convergence reached at iteration 142.
```

```
The final UCB loss was -18.466 with predicted mean of [-1.849]
The next parameters to simulate from are [[0.    0.632 0.698 0.986 0.998 0.8  ]]
Iteration 28
Acquisition function convergence reached at iteration 123.
The final UCB loss was -18.413 with predicted mean of [-1.748]
The next parameters to simulate from are [[1.    0.333 0.999 0.605 0.998 0.792]]
Iteration 29
Acquisition function convergence reached at iteration 134.
The final UCB loss was -18.406 with predicted mean of [-1.693]
The next parameters to simulate from are [[0.991 0.558 0.989 0.973 0.998 0.457]]
Iteration 30
Acquisition function convergence reached at iteration 224.
The final UCB loss was -18.354 with predicted mean of [-1.594]
The next parameters to simulate from are [[0.004 0.305 0.999 0.983 0.996 0.374]]
Iteration 31
Acquisition function convergence reached at iteration 101.
The final UCB loss was -18.615 with predicted mean of [-1.81]
The next parameters to simulate from are [[0.    0.918 0.61  0.986 1.    0.908]]
Iteration 32
Acquisition function convergence reached at iteration 136.
The final UCB loss was -18.64 with predicted mean of [-1.793]
The next parameters to simulate from are [[1.    0.61  0.798 1.    1.    0.613]]
Iteration 33
Acquisition function convergence reached at iteration 109.
The final UCB loss was -18.105 with predicted mean of [-1.224]
The next parameters to simulate from are [[1.    0.71  1.    1.    0.278 0.917]]
Iteration 34
Acquisition function convergence reached at iteration 276.
The final UCB loss was -18.406 with predicted mean of [-1.476]
The next parameters to simulate from are [[1.    0.13  0.982 1.    1.    0.289]]
Iteration 35
Acquisition function convergence reached at iteration 115.
The final UCB loss was -18.714 with predicted mean of [-1.746]
The next parameters to simulate from are [[0.    0.909 0.62  0.986 0.997 0.706]]
Iteration 36
Acquisition function convergence reached at iteration 111.
The final UCB loss was -18.361 with predicted mean of [-1.355]
The next parameters to simulate from are [[0.998 0.445 1.    0.993 0.997 0.208]]
Iteration 37
Acquisition function convergence reached at iteration 142.
The final UCB loss was -18.765 with predicted mean of [-1.722]
The next parameters to simulate from are [[0.    0.478 1.    0.637 0.999 0.693]]
Iteration 38
```

```
Acquisition function convergence reached at iteration 135.
The final UCB loss was -18.532 with predicted mean of [-1.46]
The next parameters to simulate from are [[1.    0.044 0.8   1.    0.551 0.76 ]]
Iteration 39
Acquisition function convergence reached at iteration 256.
The final UCB loss was -18.819 with predicted mean of [-1.705]
The next parameters to simulate from are [[0.998 0.34  0.517 0.992 0.999 0.826]]
Hyperparameter convergence reached at iteration 9663.
Iteration 40
Acquisition function convergence reached at iteration 68.
The final UCB loss was -17.751 with predicted mean of [-1.001]
The next parameters to simulate from are [[0.001 0.853 0.998 1.    1.    0.   ]]
Iteration 41
Acquisition function convergence reached at iteration 88.
The final UCB loss was -18.515 with predicted mean of [-1.732]
The next parameters to simulate from are [[0.    0.809 0.713 0.633 0.999 0.999]]
Iteration 42
Acquisition function convergence reached at iteration 188.
The final UCB loss was -18.255 with predicted mean of [-1.441]
The next parameters to simulate from are [[0.    0.037 0.277 0.972 0.999 1.   ]]
Iteration 43
Acquisition function convergence reached at iteration 117.
The final UCB loss was -18.568 with predicted mean of [-1.723]
The next parameters to simulate from are [[0.    0.742 0.807 0.986 1.    0.53 ]]
Iteration 44
Acquisition function convergence reached at iteration 299.
The final UCB loss was -17.826 with predicted mean of [-0.951]
The next parameters to simulate from are [[0.001 0.45  0.803 0.998 0.996 0.   ]]
Iteration 45
Acquisition function convergence reached at iteration 109.
The final UCB loss was -18.128 with predicted mean of [-1.224]
The next parameters to simulate from are [[1.    0.521 0.999 0.997 1.    0.129]]
Iteration 46
Acquisition function convergence reached at iteration 110.
The final UCB loss was -18.594 with predicted mean of [-1.662]
The next parameters to simulate from are [[0.001 0.226 0.999 0.662 1.    0.584]]
Iteration 47
Acquisition function convergence reached at iteration 311.
The final UCB loss was -18.539 with predicted mean of [-1.579]
The next parameters to simulate from are [[0.    0.446 0.876 0.434 0.999 0.999]]
Iteration 48
Acquisition function convergence reached at iteration 98.
The final UCB loss was -18.731 with predicted mean of [-1.743]
```

```
The next parameters to simulate from are [[1.    0.012 0.778 0.646 1.    0.841]]
Iteration 49
Acquisition function convergence reached at iteration 97.
The final UCB loss was -18.398 with predicted mean of [-1.387]
The next parameters to simulate from are [[0.999 0.795 1.    0.331 0.997 0.997]]
Iteration 50
Acquisition function convergence reached at iteration 403.
The final UCB loss was -18.671 with predicted mean of [-1.631]
The next parameters to simulate from are [[0.995 0.93  0.433 0.999 0.999 0.905]]
Trained parameters:
amplitude_champ:0 is 0.75

length_scales_champ:0 is [0.464 1.    0.042 0.093 0.236 0.019]

observation_noise_variance_champ:0 is 0.255

amp_f_mean:0 is 0.901

amp_gamma_L_mean:0 is 0.013

amp_lambda_mean:0 is 0.891

amp_r_mean:0 is 0.034

bias_mean:0 is 0.413

f_tp:0 is 0.297

gamma_L_tp:0 is 0.879

lambda_tp:0 is 0.39

r_tp:0 is 0.593

Iteration 51
Acquisition function convergence reached at iteration 1978.
The final UCB loss was -17.846 with predicted mean of [-0.787]
The next parameters to simulate from are [[0.    0.692 1.    1.    0.076 1.   ]]
Iteration 52
Acquisition function convergence reached at iteration 932.
The final UCB loss was -18.114 with predicted mean of [-1.04]
The next parameters to simulate from are [[1.    0.199 1.    0.526 0.361 0.999]]
Iteration 53
```

```
Acquisition function convergence reached at iteration 132.
The final UCB loss was -18.682 with predicted mean of [-1.567]
The next parameters to simulate from are [[0.999 0.578 0.537 0.618 0.996 0.998]]
Iteration 54
Acquisition function convergence reached at iteration 116.
The final UCB loss was -17.979 with predicted mean of [-0.84]
The next parameters to simulate from are [[0.025 0.084 0.62  1.    0.993 0.001]]
Iteration 55
Acquisition function convergence reached at iteration 128.
The final UCB loss was -18.865 with predicted mean of [-1.702]
The next parameters to simulate from are [[0.001 0.251 0.813 0.592 1.    0.919]]
Iteration 56
Acquisition function convergence reached at iteration 115.
The final UCB loss was -18.87 with predicted mean of [-1.684]
The next parameters to simulate from are [[0.    0.51  0.629 0.969 1.    0.614]]
Iteration 57
Acquisition function convergence reached at iteration 42.
The final UCB loss was -18.174 with predicted mean of [-0.965]
The next parameters to simulate from are [[1.    0.694 0.894 0.001 1.    0.998]]
Iteration 58
Acquisition function convergence reached at iteration 186.
The final UCB loss was -18.848 with predicted mean of [-1.617]
The next parameters to simulate from are [[0.002 0.886 0.472 0.993 1.    0.756]]
Iteration 59
Acquisition function convergence reached at iteration 142.
The final UCB loss was -18.902 with predicted mean of [-1.649]
The next parameters to simulate from are [[0.003 0.995 0.615 0.669 0.99  0.851]]
Iteration 60
Acquisition function convergence reached at iteration 135.
The final UCB loss was -19.62 with predicted mean of [-1.49]
The next parameters to simulate from are [[0.    0.077 1.    0.444 1.    0.94 ]]
Iteration 61
Acquisition function convergence reached at iteration 118.
The final UCB loss was -19.682 with predicted mean of [-1.524]
The next parameters to simulate from are [[0.    0.016 0.361 0.997 1.    0.838]]
Iteration 62
Acquisition function convergence reached at iteration 127.
The final UCB loss was -19.224 with predicted mean of [-1.044]
The next parameters to simulate from are [[0.999 0.201 0.993 0.764 1.    0.074]]
Iteration 63
Acquisition function convergence reached at iteration 161.
The final UCB loss was -19.83 with predicted mean of [-1.627]
The next parameters to simulate from are [[0.    0.328 0.803 1.    1.    0.44 ]]
```

```
Iteration 64
Acquisition function convergence reached at iteration 208.
The final UCB loss was -19.768 with predicted mean of [-1.548]
The next parameters to simulate from are [[1.    0.197 0.905 0.466 0.999 0.839]]
Iteration 65
Acquisition function convergence reached at iteration 122.
The final UCB loss was -19.766 with predicted mean of [-1.521]
The next parameters to simulate from are [[0.999 0.837 0.804 0.988 1.    0.354]]
Iteration 66
Acquisition function convergence reached at iteration 118.
The final UCB loss was -19.821 with predicted mean of [-1.558]
The next parameters to simulate from are [[0.    0.911 0.514 0.657 0.992 0.928]]
Iteration 67
Acquisition function convergence reached at iteration 95.
The final UCB loss was -19.805 with predicted mean of [-1.523]
The next parameters to simulate from are [[1.    0.113 0.678 0.712 0.949 0.934]]
Iteration 68
Acquisition function convergence reached at iteration 148.
The final UCB loss was -19.882 with predicted mean of [-1.578]
The next parameters to simulate from are [[0.    0.958 0.929 0.677 0.999 0.5  ]]
Iteration 69
Acquisition function convergence reached at iteration 149.
The final UCB loss was -19.792 with predicted mean of [-1.469]
The next parameters to simulate from are [[1.    0.786 1.    0.632 0.998 0.418]]
Iteration 70
Acquisition function convergence reached at iteration 1847.
The final UCB loss was -19.704 with predicted mean of [-1.37]
The next parameters to simulate from are [[1.    0.319 0.999 0.702 0.516 0.843]]
Iteration 71
Acquisition function convergence reached at iteration 436.
The final UCB loss was -19.482 with predicted mean of [-1.133]
The next parameters to simulate from are [[1.    0.234 1.    1.    0.254 0.8  ]]
Iteration 72
Acquisition function convergence reached at iteration 138.
The final UCB loss was -19.803 with predicted mean of [-1.424]
The next parameters to simulate from are [[0.002 0.053 0.323 0.782 1.    0.922]]
Iteration 73
Acquisition function convergence reached at iteration 136.
The final UCB loss was -19.695 with predicted mean of [-1.297]
The next parameters to simulate from are [[0.032 0.226 0.2   1.    0.997 0.879]]
Iteration 74
Acquisition function convergence reached at iteration 566.
The final UCB loss was -19.393 with predicted mean of [-0.977]
```

```
The next parameters to simulate from are [[1.    0.77  0.    0.99  1.    0.882]]
Iteration 75
Acquisition function convergence reached at iteration 267.
The final UCB loss was -19.847 with predicted mean of [-1.413]
The next parameters to simulate from are [[0.    0.678 0.8   0.986 0.999 0.277]]
Iteration 76
Acquisition function convergence reached at iteration 191.
The final UCB loss was -19.626 with predicted mean of [-1.175]
The next parameters to simulate from are [[0.    0.772 0.138 0.999 1.    0.956]]
Iteration 77
Acquisition function convergence reached at iteration 1987.
The final UCB loss was -19.338 with predicted mean of [-0.921]
The next parameters to simulate from are [[1.   0.39 1.   0.06 1.   1.  ]]
Iteration 78
Acquisition function convergence reached at iteration 401.
The final UCB loss was -19.929 with predicted mean of [-1.448]
The next parameters to simulate from are [[0.    0.667 0.701 0.998 0.598 0.958]]
Iteration 79
Acquisition function convergence reached at iteration 128.
The final UCB loss was -19.79 with predicted mean of [-1.287]
The next parameters to simulate from are [[0.    0.645 0.82  0.998 1.    0.192]]
Iteration 80
Acquisition function convergence reached at iteration 135.
The final UCB loss was -20.567 with predicted mean of [-1.367]
The next parameters to simulate from are [[0.    0.07  0.387 0.587 0.998 1.   ]]
Iteration 81
Acquisition function convergence reached at iteration 187.
The final UCB loss was -20.554 with predicted mean of [-1.341]
The next parameters to simulate from are [[0.    0.053 1.    0.284 0.999 0.857]]
Iteration 82
Acquisition function convergence reached at iteration 83.
The final UCB loss was -20.54 with predicted mean of [-1.307]
The next parameters to simulate from are [[0.002 0.356 0.616 0.332 1.    0.999]]
Iteration 83
Acquisition function convergence reached at iteration 367.
The final UCB loss was -20.135 with predicted mean of [-0.884]
The next parameters to simulate from are [[1.    0.552 1.    1.    0.    0.699]]
Iteration 84
Acquisition function convergence reached at iteration 199.
The final UCB loss was -19.784 with predicted mean of [-0.589]
The next parameters to simulate from are [[0.    0.489 0.824 0.682 1.    0.999]]
Iteration 85
Acquisition function convergence reached at iteration 123.
```

```
The final UCB loss was -20.889 with predicted mean of [-1.606]
The next parameters to simulate from are [[1.    0.506 0.625 0.986 1.    0.525]]
Iteration 86
Acquisition function convergence reached at iteration 118.
The final UCB loss was -20.435 with predicted mean of [-1.135]
The next parameters to simulate from are [[0.    0.923 0.205 0.603 1.    1.   ]]
Iteration 87
Acquisition function convergence reached at iteration 126.
The final UCB loss was -20.224 with predicted mean of [-0.912]
The next parameters to simulate from are [[1.    0.818 0.056 0.696 0.997 1.   ]]
Iteration 88
Acquisition function convergence reached at iteration 72.
The final UCB loss was -21.055 with predicted mean of [-1.724]
The next parameters to simulate from are [[0.    0.267 0.837 0.664 1.    0.749]]
Iteration 89
Acquisition function convergence reached at iteration 112.
The final UCB loss was -20.861 with predicted mean of [-1.515]
The next parameters to simulate from are [[0.    0.047 0.616 0.991 0.998 0.446]]
Iteration 90
Acquisition function convergence reached at iteration 155.
The final UCB loss was -20.767 with predicted mean of [-1.407]
The next parameters to simulate from are [[1.    0.4   0.684 0.39  0.995 0.894]]
Iteration 91
Acquisition function convergence reached at iteration 108.
The final UCB loss was -20.943 with predicted mean of [-1.566]
The next parameters to simulate from are [[0.999 0.406 0.468 0.987 0.999 0.659]]
Iteration 92
Acquisition function convergence reached at iteration 163.
The final UCB loss was -21.075 with predicted mean of [-1.684]
The next parameters to simulate from are [[0.001 0.492 0.821 0.695 0.997 0.656]]
Iteration 93
Acquisition function convergence reached at iteration 102.
The final UCB loss was -20.858 with predicted mean of [-1.455]
The next parameters to simulate from are [[1.    0.349 0.691 1.    0.612 0.858]]
Iteration 94
Acquisition function convergence reached at iteration 157.
The final UCB loss was -20.85 with predicted mean of [-1.43]
The next parameters to simulate from are [[1.    0.851 0.332 1.    1.    0.71 ]]
Iteration 95
Acquisition function convergence reached at iteration 65.
The final UCB loss was -20.124 with predicted mean of [-0.69]
The next parameters to simulate from are [[1.    0.893 0.461 0.993 0.992 0.005]]
Iteration 96
```

```
Acquisition function convergence reached at iteration 1508.
The final UCB loss was -20.097 with predicted mean of [-0.695]
The next parameters to simulate from are [[1.    0.749 1.    1.    1.    0.056]]
Iteration 97
Acquisition function convergence reached at iteration 715.
The final UCB loss was -20.616 with predicted mean of [-1.162]
The next parameters to simulate from are [[1.    0.17  0.547 1.    0.375 0.999]]
Iteration 98
Acquisition function convergence reached at iteration 116.
The final UCB loss was -20.815 with predicted mean of [-1.339]
The next parameters to simulate from are [[0.    0.46  0.261 0.996 0.997 0.772]]
Iteration 99
Acquisition function convergence reached at iteration 153.
The final UCB loss was -20.994 with predicted mean of [-1.504]
The next parameters to simulate from are [[0.    0.09  0.473 0.999 1.    0.568]]
Hyperparameter convergence reached at iteration 8608.
Iteration 100
Acquisition function convergence reached at iteration 174.
The final UCB loss was -21.39 with predicted mean of [-1.423]
The next parameters to simulate from are [[0.005 0.69  1.    0.362 1.    0.749]]
Trained parameters:
amplitude_champ:0 is 0.77

length_scales_champ:0 is [0.475 1.    0.042 0.094 0.208 0.019]

observation_noise_variance_champ:0 is 0.243

amp_f_mean:0 is 0.872

amp_gamma_L_mean:0 is 0.057

amp_lambda_mean:0 is 0.082

amp_r_mean:0 is 0.094

bias_mean:0 is 0.291

f_tp:0 is 0.33

gamma_L_tp:0 is 0.891

lambda_tp:0 is 0.398
```

```
r_tp:0 is 0.608

Iteration 101
Acquisition function convergence reached at iteration 123.
The final UCB loss was -21.123 with predicted mean of [-1.14]
The next parameters to simulate from are [[1.    0.846 0.795 1.    1.    0.112]]
Iteration 102
Acquisition function convergence reached at iteration 142.
The final UCB loss was -21.362 with predicted mean of [-1.365]
The next parameters to simulate from are [[0.001 0.963 1.    0.645 0.997 0.326]]
Iteration 103
Acquisition function convergence reached at iteration 534.
The final UCB loss was -21.062 with predicted mean of [-1.058]
The next parameters to simulate from are [[1.    0.525 0.818 1.    0.226 1.   ]]
Iteration 104
Acquisition function convergence reached at iteration 117.
The final UCB loss was -21.639 with predicted mean of [-1.616]
The next parameters to simulate from are [[1.    0.875 0.647 0.648 0.999 0.755]]
Iteration 105
Acquisition function convergence reached at iteration 133.
The final UCB loss was -21.603 with predicted mean of [-1.566]
The next parameters to simulate from are [[1.    0.118 0.758 0.643 1.    0.574]]
Iteration 106
Acquisition function convergence reached at iteration 224.
The final UCB loss was -21.25 with predicted mean of [-1.206]
The next parameters to simulate from are [[0.    0.736 0.805 1.    0.374 0.825]]
Iteration 107
Acquisition function convergence reached at iteration 694.
The final UCB loss was -20.818 with predicted mean of [-0.76]
The next parameters to simulate from are [[1.    0.644 0.16  1.    0.475 1.   ]]
Iteration 108
Acquisition function convergence reached at iteration 78.
The final UCB loss was -20.872 with predicted mean of [-0.797]
The next parameters to simulate from are [[1.    0.971 0.999 0.612 0.997 0.002]]
Iteration 109
Acquisition function convergence reached at iteration 536.
The final UCB loss was -20.088 with predicted mean of [0.]
The next parameters to simulate from are [[1.    0.604 0.    0.997 0.994 0.   ]]
Iteration 110
Acquisition function convergence reached at iteration 103.
The final UCB loss was -20.595 with predicted mean of [-0.495]
The next parameters to simulate from are [[0.    0.399 0.309 0.998 0.998 0.003]]
Iteration 111
```

```
Acquisition function convergence reached at iteration 692.
The final UCB loss was -21.366 with predicted mean of [-1.253]
The next parameters to simulate from are [[0.    0.603 0.93  0.674 0.997 0.246]]
Iteration 112
Acquisition function convergence reached at iteration 302.
The final UCB loss was -21.237 with predicted mean of [-1.112]
The next parameters to simulate from are [[0.999 0.524 0.922 0.664 0.999 0.164]]
Iteration 113
Acquisition function convergence reached at iteration 180.
The final UCB loss was -21.401 with predicted mean of [-1.265]
The next parameters to simulate from are [[1.    0.41  0.765 0.279 1.    0.953]]
Iteration 114
Acquisition function convergence reached at iteration 161.
The final UCB loss was -21.695 with predicted mean of [-1.545]
The next parameters to simulate from are [[1.    0.199 0.646 0.625 0.999 0.669]]
Iteration 115
Acquisition function convergence reached at iteration 165.
The final UCB loss was -21.567 with predicted mean of [-1.406]
The next parameters to simulate from are [[1.    0.694 0.999 0.407 0.996 0.637]]
Iteration 116
Acquisition function convergence reached at iteration 130.
The final UCB loss was -21.636 with predicted mean of [-1.463]
The next parameters to simulate from are [[0.    0.617 0.496 0.608 0.999 0.789]]
Iteration 117
Acquisition function convergence reached at iteration 154.
The final UCB loss was -21.53 with predicted mean of [-1.345]
The next parameters to simulate from are [[0.    0.599 0.422 0.594 0.999 0.86 ]]
Iteration 118
Acquisition function convergence reached at iteration 149.
The final UCB loss was -21.668 with predicted mean of [-1.471]
The next parameters to simulate from are [[0.    0.504 0.719 0.649 0.994 0.491]]
Iteration 119
Acquisition function convergence reached at iteration 108.
The final UCB loss was -21.15 with predicted mean of [-0.943]
The next parameters to simulate from are [[1.    0.194 0.998 0.423 1.    0.21 ]]
Hyperparameter convergence reached at iteration 9311.
Iteration 120
Acquisition function convergence reached at iteration 196.
The final UCB loss was -21.742 with predicted mean of [-1.273]
The next parameters to simulate from are [[1.    0.318 0.914 0.689 0.462 0.926]]
Iteration 121
Acquisition function convergence reached at iteration 188.
The final UCB loss was -21.905 with predicted mean of [-1.421]
```

```
The next parameters to simulate from are [[1.    0.085 0.63  0.973 0.989 0.362]]
Iteration 122
Acquisition function convergence reached at iteration 166.
The final UCB loss was -21.377 with predicted mean of [-0.885]
The next parameters to simulate from are [[0.    0.898 0.373 0.999 0.317 1.   ]]
Iteration 123
Acquisition function convergence reached at iteration 308.
The final UCB loss was -21.112 with predicted mean of [-0.616]
The next parameters to simulate from are [[0.999 0.932 0.    0.428 0.989 0.999]]
Iteration 124
Acquisition function convergence reached at iteration 105.
The final UCB loss was -21.613 with predicted mean of [-1.103]
The next parameters to simulate from are [[1.    0.643 0.625 0.691 0.393 0.999]]
Iteration 125
Acquisition function convergence reached at iteration 156.
The final UCB loss was -21.645 with predicted mean of [-1.116]
The next parameters to simulate from are [[0.992 0.245 0.116 0.999 0.986 0.808]]
Iteration 126
Acquisition function convergence reached at iteration 111.
The final UCB loss was -21.562 with predicted mean of [-1.035]
The next parameters to simulate from are [[0.    0.064 0.764 0.154 0.999 1.   ]]
Iteration 127
Acquisition function convergence reached at iteration 215.
The final UCB loss was -21.407 with predicted mean of [-0.859]
The next parameters to simulate from are [[1.    0.644 1.    0.48  0.996 0.121]]
Iteration 128
Acquisition function convergence reached at iteration 151.
The final UCB loss was -21.622 with predicted mean of [-1.065]
The next parameters to simulate from are [[1.    0.996 0.772 0.638 0.355 0.955]]
Iteration 129
Acquisition function convergence reached at iteration 130.
The final UCB loss was -21.825 with predicted mean of [-1.255]
The next parameters to simulate from are [[1.    0.129 0.836 0.262 0.999 0.885]]
Iteration 130
Acquisition function convergence reached at iteration 97.
The final UCB loss was -21.52 with predicted mean of [-0.936]
The next parameters to simulate from are [[0.999 0.309 0.004 0.993 0.993 0.744]]
Iteration 131
Acquisition function convergence reached at iteration 111.
The final UCB loss was -21.668 with predicted mean of [-1.08]
The next parameters to simulate from are [[1.    0.006 0.821 1.    0.254 0.913]]
Iteration 132
Acquisition function convergence reached at iteration 1183.
```

```
The final UCB loss was -21.48 with predicted mean of [-0.883]
The next parameters to simulate from are [[0.    0.136 0.874 0.657 0.211 1.   ]]
Iteration 133
Acquisition function convergence reached at iteration 409.
The final UCB loss was -21.82 with predicted mean of [-1.216]
The next parameters to simulate from are [[1.    0.83  0.511 1.    0.547 0.938]]
Iteration 134
Acquisition function convergence reached at iteration 248.
The final UCB loss was -22.062 with predicted mean of [-1.439]
The next parameters to simulate from are [[0.004 0.517 0.731 0.423 0.995 0.794]]
Iteration 135
Acquisition function convergence reached at iteration 148.
The final UCB loss was -21.992 with predicted mean of [-1.357]
The next parameters to simulate from are [[1.    0.508 0.328 0.955 1.    0.62 ]]
Iteration 136
Acquisition function convergence reached at iteration 186.
The final UCB loss was -21.847 with predicted mean of [-1.202]
The next parameters to simulate from are [[0.999 0.517 0.198 1.    0.999 0.669]]
Iteration 137
Acquisition function convergence reached at iteration 149.
The final UCB loss was -21.962 with predicted mean of [-1.306]
The next parameters to simulate from are [[0.999 0.034 0.62  0.997 1.    0.283]]
Iteration 138
Acquisition function convergence reached at iteration 298.
The final UCB loss was -22.059 with predicted mean of [-1.394]
The next parameters to simulate from are [[0.997 0.4   0.456 1.    0.99  0.484]]
Iteration 139
Acquisition function convergence reached at iteration 206.
The final UCB loss was -20.673 with predicted mean of [0.003]
The next parameters to simulate from are [[0.    0.605 0.    0.    0.987 1.   ]]
Iteration 140
Acquisition function convergence reached at iteration 365.
The final UCB loss was -22.423 with predicted mean of [-1.198]
The next parameters to simulate from are [[0.    0.883 0.634 0.964 0.997 0.207]]
Iteration 141
Acquisition function convergence reached at iteration 97.
The final UCB loss was -21.749 with predicted mean of [-0.556]
The next parameters to simulate from are [[0.    0.298 0.035 0.998 0.445 1.   ]]
Iteration 142
Acquisition function convergence reached at iteration 209.
The final UCB loss was -22.598 with predicted mean of [-1.356]
The next parameters to simulate from are [[1.    0.98  0.867 0.43  0.999 0.596]]
Iteration 143
```

```
Acquisition function convergence reached at iteration 132.
The final UCB loss was -22.136 with predicted mean of [-0.881]
The next parameters to simulate from are [[0.    0.34  0.001 0.997 0.994 0.666]]
Iteration 144
Acquisition function convergence reached at iteration 133.
The final UCB loss was -22.536 with predicted mean of [-1.272]
The next parameters to simulate from are [[0.    0.508 1.    0.361 1.    0.531]]
Iteration 145
Acquisition function convergence reached at iteration 120.
The final UCB loss was -22.329 with predicted mean of [-1.055]
The next parameters to simulate from are [[0.997 0.234 0.617 0.99  0.999 0.127]]
Iteration 146
Acquisition function convergence reached at iteration 153.
The final UCB loss was -22.204 with predicted mean of [-0.931]
The next parameters to simulate from are [[0.    0.76  0.979 0.999 0.479 0.25 ]]
Iteration 147
Acquisition function convergence reached at iteration 321.
The final UCB loss was -22.35 with predicted mean of [-1.059]
The next parameters to simulate from are [[1.    0.1   0.997 0.112 0.995 0.922]]
Iteration 148
Acquisition function convergence reached at iteration 107.
The final UCB loss was -22.095 with predicted mean of [-0.799]
The next parameters to simulate from are [[0.997 0.339 0.902 0.997 0.511 0.155]]
Iteration 149
Acquisition function convergence reached at iteration 107.
The final UCB loss was -21.477 with predicted mean of [-0.238]
The next parameters to simulate from are [[0.    0.675 1.    0.696 0.001 0.999]]
Iteration 150
Acquisition function convergence reached at iteration 132.
The final UCB loss was -22.172 with predicted mean of [-0.86]
The next parameters to simulate from are [[0.    0.845 0.862 0.992 0.996 0.064]]
Trained parameters:
amplitude_champ:0 is 0.776

length_scales_champ:0 is [0.475 1.    0.041 0.095 0.203 0.019]

observation_noise_variance_champ:0 is 0.237

amp_f_mean:0 is 0.865

amp_gamma_L_mean:0 is 0.036

amp_lambda_mean:0 is 0.052
```

```
amp_r_mean:0 is 0.013

bias_mean:0 is 0.504

f_tp:0 is 0.356

gamma_L_tp:0 is 0.72

lambda_tp:0 is 0.367

r_tp:0 is 0.935

Iteration 151
Acquisition function convergence reached at iteration 126.
The final UCB loss was -22.645 with predicted mean of [-1.314]
The next parameters to simulate from are [[0.    0.957 0.467 0.996 0.999 0.395]]
Iteration 152
Acquisition function convergence reached at iteration 233.
The final UCB loss was -22.782 with predicted mean of [-1.443]
The next parameters to simulate from are [[0.    0.86  0.492 0.648 0.999 0.694]]
Iteration 153
Acquisition function convergence reached at iteration 85.
The final UCB loss was -21.907 with predicted mean of [-0.566]
The next parameters to simulate from are [[0.998 0.86  0.997 0.217 0.991 0.163]]
Iteration 154
Acquisition function convergence reached at iteration 161.
The final UCB loss was -22.6 with predicted mean of [-1.244]
The next parameters to simulate from are [[0.    0.527 0.596 0.999 0.478 0.771]]
Iteration 155
Acquisition function convergence reached at iteration 531.
The final UCB loss was -22.407 with predicted mean of [-1.042]
The next parameters to simulate from are [[1.    0.798 0.464 0.268 0.997 1.   ]]
Iteration 156
Acquisition function convergence reached at iteration 72.
The final UCB loss was -22.153 with predicted mean of [-0.777]
The next parameters to simulate from are [[1.    0.011 0.813 0.627 1.    0.005]]
Iteration 157
Acquisition function convergence reached at iteration 237.
The final UCB loss was -22.017 with predicted mean of [-0.659]
The next parameters to simulate from are [[0.    0.38  0.892 0.999 0.176 0.952]]
Iteration 158
Acquisition function convergence reached at iteration 110.
```

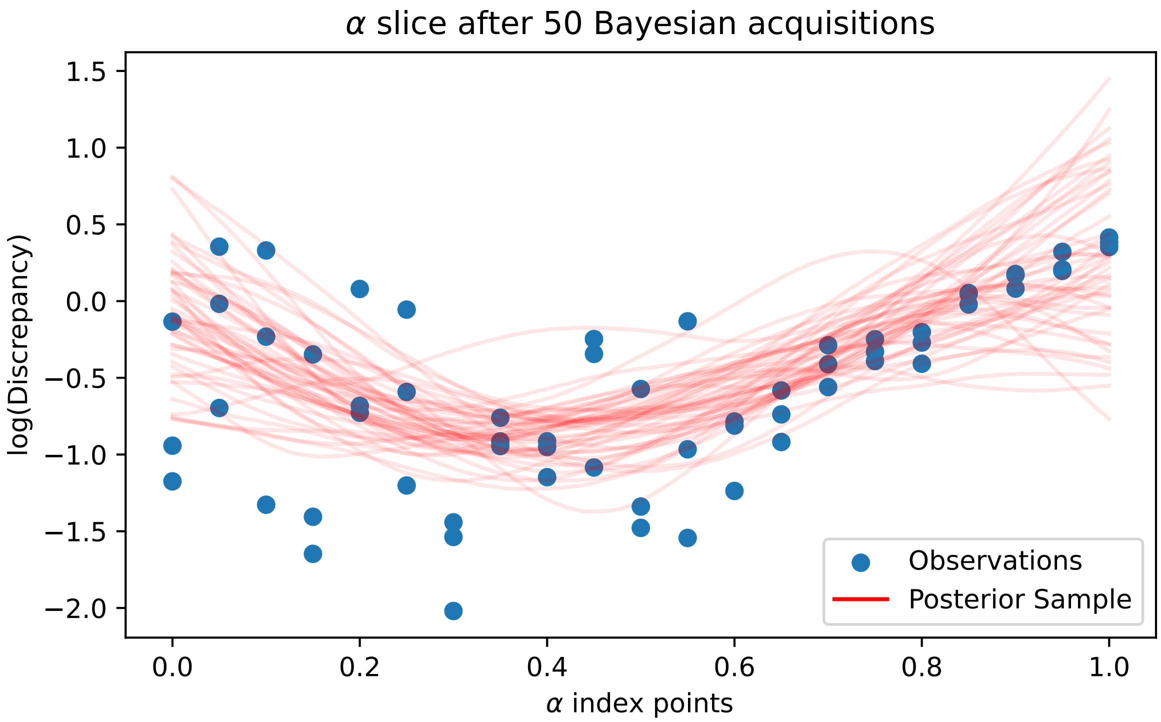
```
The final UCB loss was -22.56 with predicted mean of [-1.169]
The next parameters to simulate from are [[1.    0.892 0.999 0.365 1.    0.46 ]]
Iteration 159
Acquisition function convergence reached at iteration 202.
The final UCB loss was -22.827 with predicted mean of [-1.424]
The next parameters to simulate from are [[0.001 0.214 0.59  0.615 0.996 0.571]]
Iteration 160
Acquisition function convergence reached at iteration 125.
The final UCB loss was -22.595 with predicted mean of [-0.792]
The next parameters to simulate from are [[1.    0.709 0.985 0.717 0.    0.753]]
Iteration 161
Acquisition function convergence reached at iteration 129.
The final UCB loss was -23.008 with predicted mean of [-1.197]
The next parameters to simulate from are [[0.    0.974 0.475 0.952 0.996 0.31 ]]
Iteration 162
Acquisition function convergence reached at iteration 192.
The final UCB loss was -22.703 with predicted mean of [-0.885]
The next parameters to simulate from are [[0.998 0.155 0.    0.793 1.    0.819]]
Iteration 163
Acquisition function convergence reached at iteration 106.
The final UCB loss was -22.886 with predicted mean of [-1.064]
The next parameters to simulate from are [[1.    0.146 0.886 0.299 0.601 0.937]]
Iteration 164
Acquisition function convergence reached at iteration 156.
The final UCB loss was -23.108 with predicted mean of [-1.27]
The next parameters to simulate from are [[1.    0.556 0.314 0.998 0.99  0.534]]
Iteration 165
Acquisition function convergence reached at iteration 174.
The final UCB loss was -23.026 with predicted mean of [-1.18]
The next parameters to simulate from are [[1.    0.301 0.315 0.994 0.984 0.446]]
Iteration 166
Acquisition function convergence reached at iteration 541.
The final UCB loss was -23.104 with predicted mean of [-1.256]
The next parameters to simulate from are [[0.    0.509 0.513 0.999 0.504 0.866]]
Iteration 167
Acquisition function convergence reached at iteration 76.
The final UCB loss was -22.591 with predicted mean of [-0.73]
The next parameters to simulate from are [[0.998 0.394 0.615 0.    1.    0.999]]
Iteration 168
Acquisition function convergence reached at iteration 178.
The final UCB loss was -23.254 with predicted mean of [-1.384]
The next parameters to simulate from are [[0.001 0.315 0.739 0.421 0.999 0.704]]
Iteration 169
```

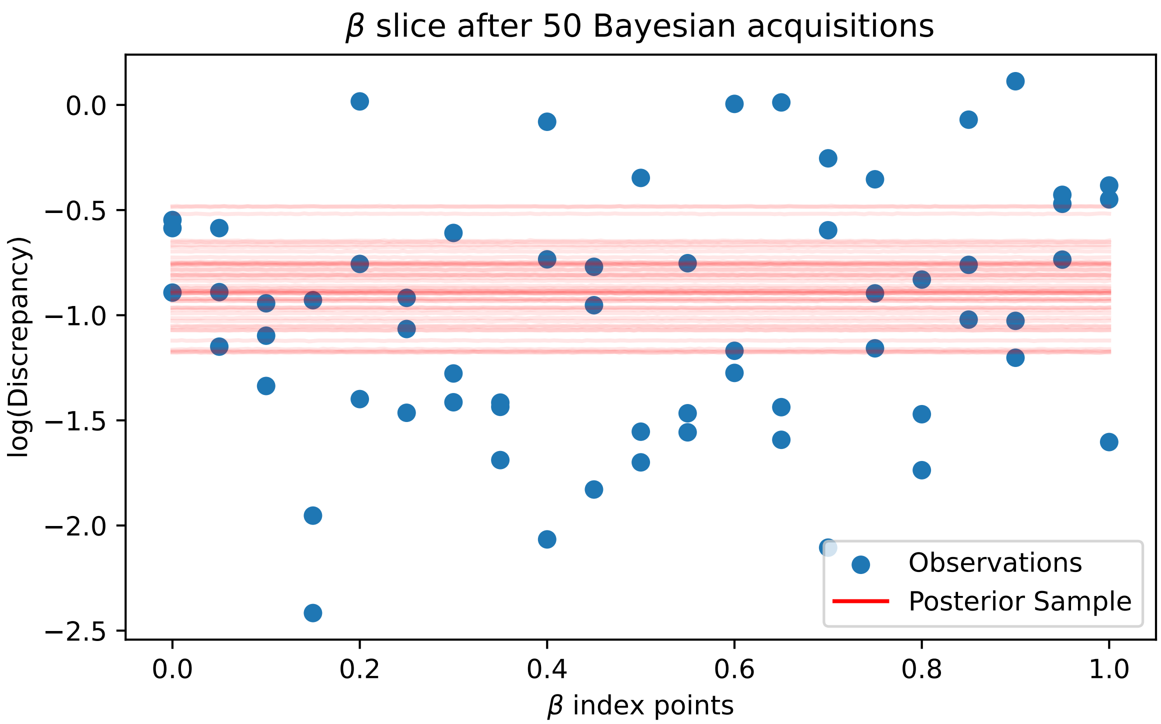
```
Acquisition function convergence reached at iteration 152.
The final UCB loss was -22.719 with predicted mean of [-0.838]
The next parameters to simulate from are [[1.    0.068 0.827 1.    0.    0.677]]
Iteration 170
Acquisition function convergence reached at iteration 120.
The final UCB loss was -23.176 with predicted mean of [-1.287]
The next parameters to simulate from are [[0.999 0.69  0.349 0.639 1.    0.75 ]]
Iteration 171
Acquisition function convergence reached at iteration 384.
The final UCB loss was -22.673 with predicted mean of [-0.779]
The next parameters to simulate from are [[1.    0.008 0.934 0.999 0.    0.86 ]]
Iteration 172
Acquisition function convergence reached at iteration 227.
The final UCB loss was -22.347 with predicted mean of [-0.447]
The next parameters to simulate from are [[0.    0.482 0.996 0.312 0.994 0.   ]]
Iteration 173
Acquisition function convergence reached at iteration 233.
The final UCB loss was -22.986 with predicted mean of [-1.072]
The next parameters to simulate from are [[1.    0.056 0.464 0.985 1.    0.227]]
Iteration 174
Acquisition function convergence reached at iteration 100.
The final UCB loss was -22.869 with predicted mean of [-0.947]
The next parameters to simulate from are [[0.    0.385 0.466 0.982 0.998 0.153]]
Iteration 175
Acquisition function convergence reached at iteration 95.
The final UCB loss was -22.378 with predicted mean of [-0.455]
The next parameters to simulate from are [[1.    0.11  0.716 0.998 0.482 0.   ]]
Iteration 176
Acquisition function convergence reached at iteration 256.
The final UCB loss was -23.138 with predicted mean of [-1.203]
The next parameters to simulate from are [[0.001 0.193 0.873 0.265 0.997 0.787]]
Iteration 177
Acquisition function convergence reached at iteration 109.
The final UCB loss was -23.152 with predicted mean of [-1.207]
The next parameters to simulate from are [[1.    0.96  0.581 0.316 0.996 0.831]]
Iteration 178
Acquisition function convergence reached at iteration 453.
The final UCB loss was -18.412 with predicted mean of [1.072]
The next parameters to simulate from are [[1.    0.187 0.    0.    1.    1.   ]]
Iteration 179
Acquisition function convergence reached at iteration 121.
The final UCB loss was -23.015 with predicted mean of [-1.06]
The next parameters to simulate from are [[1.    0.208 0.728 0.185 0.999 0.845]]
```

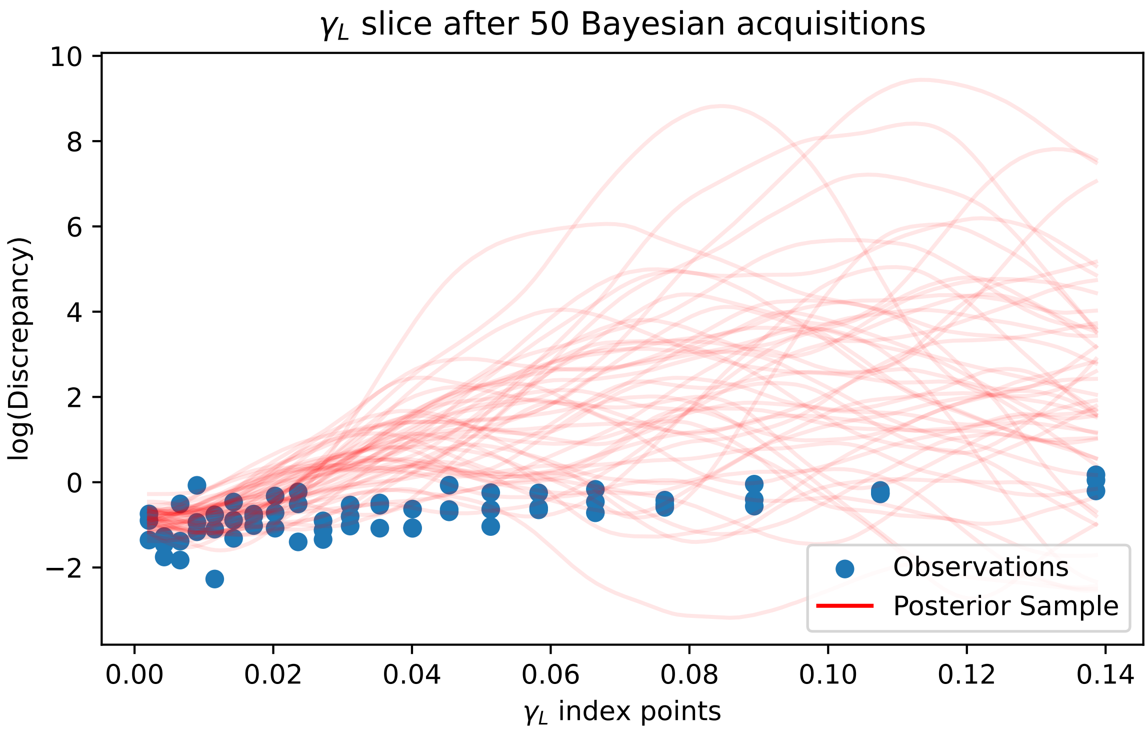
```
Iteration 180
Acquisition function convergence reached at iteration 206.
The final UCB loss was -22.935 with predicted mean of [-0.668]
The next parameters to simulate from are [[0.    0.307 0.657 0.615 1.    0.   ]]
Iteration 181
Acquisition function convergence reached at iteration 884.
The final UCB loss was -23.124 with predicted mean of [-0.876]
The next parameters to simulate from are [[0.    0.799 1.    0.215 0.463 1.   ]]
Iteration 182
Acquisition function convergence reached at iteration 161.
The final UCB loss was -23.418 with predicted mean of [-1.135]
The next parameters to simulate from are [[0.001 0.081 0.202 0.99  0.998 0.587]]
Iteration 183
Acquisition function convergence reached at iteration 115.
The final UCB loss was -23.534 with predicted mean of [-1.245]
The next parameters to simulate from are [[1.    0.601 0.9   0.999 0.576 0.408]]
Iteration 184
Acquisition function convergence reached at iteration 133.
The final UCB loss was -23.422 with predicted mean of [-1.127]
The next parameters to simulate from are [[0.    0.037 0.999 0.209 0.997 0.683]]
Iteration 185
Acquisition function convergence reached at iteration 198.
The final UCB loss was -23.711 with predicted mean of [-1.404]
The next parameters to simulate from are [[0.    0.358 0.823 0.629 0.993 0.399]]
Iteration 186
Acquisition function convergence reached at iteration 135.
The final UCB loss was -23.647 with predicted mean of [-1.337]
The next parameters to simulate from are [[0.    0.565 1.    1.    0.464 0.583]]
Iteration 187
Acquisition function convergence reached at iteration 116.
The final UCB loss was -23.214 with predicted mean of [-0.892]
The next parameters to simulate from are [[1.    0.449 0.309 0.282 1.    1.   ]]
Iteration 188
Acquisition function convergence reached at iteration 151.
The final UCB loss was -23.247 with predicted mean of [-0.919]
The next parameters to simulate from are [[1.    0.895 0.728 0.999 0.532 0.24 ]]
Iteration 189
Acquisition function convergence reached at iteration 299.
The final UCB loss was -23.203 with predicted mean of [-0.867]
The next parameters to simulate from are [[1.    0.632 0.    0.735 0.999 0.928]]
Iteration 190
Acquisition function convergence reached at iteration 1107.
The final UCB loss was -23.011 with predicted mean of [-0.718]
```

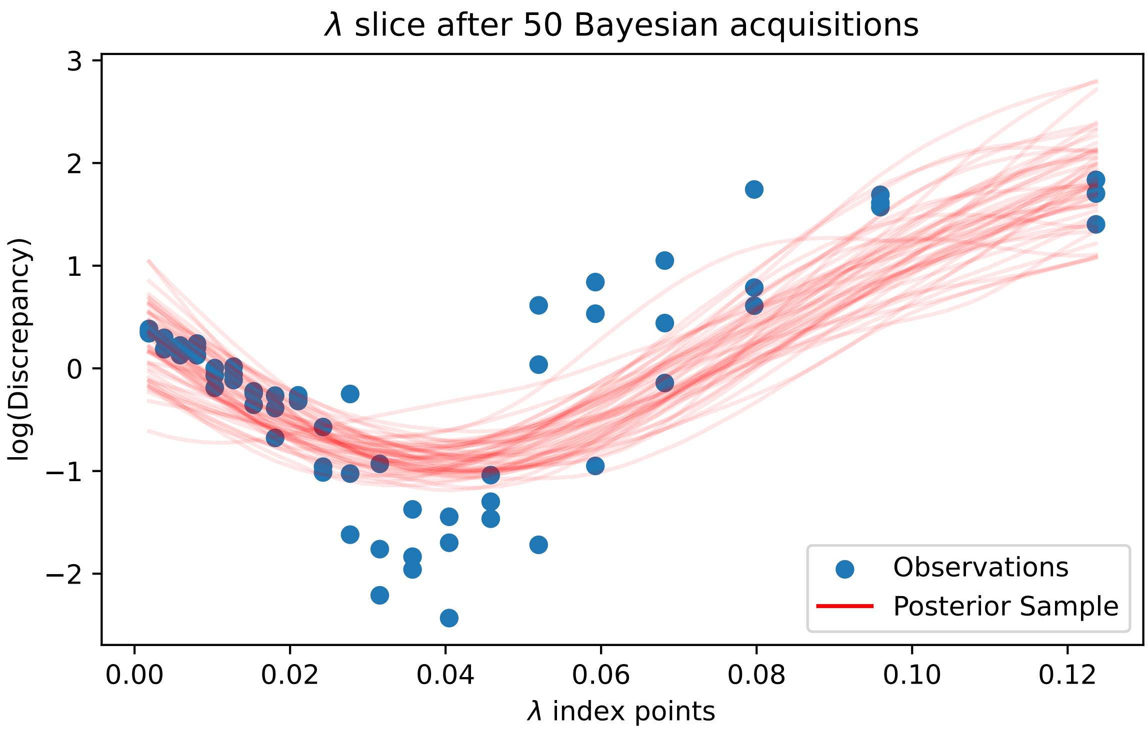
```
The next parameters to simulate from are [[1.    0.096 0.    1.    1.    0.94 ]]
Iteration 191
Acquisition function convergence reached at iteration 102.
The final UCB loss was -23.422 with predicted mean of [-1.069]
The next parameters to simulate from are [[0.002 0.647 0.309 0.98  0.996 0.362]]
Iteration 192
Acquisition function convergence reached at iteration 95.
The final UCB loss was -23.579 with predicted mean of [-1.219]
The next parameters to simulate from are [[1.    0.024 0.268 0.646 1.    0.825]]
Iteration 193
Acquisition function convergence reached at iteration 177.
The final UCB loss was -23.337 with predicted mean of [-0.976]
The next parameters to simulate from are [[1.    0.419 0.742 0.393 0.46  1.   ]]
Iteration 194
Acquisition function convergence reached at iteration 158.
The final UCB loss was -23.595 with predicted mean of [-1.226]
The next parameters to simulate from are [[1.    0.067 0.726 0.669 0.461 0.881]]
Iteration 195
Acquisition function convergence reached at iteration 561.
The final UCB loss was -20.46 with predicted mean of [0.089]
The next parameters to simulate from are [[1.    0.774 0.    0.999 1.    0.97 ]]
Iteration 196
Acquisition function convergence reached at iteration 113.
The final UCB loss was -23.842 with predicted mean of [-1.46]
The next parameters to simulate from are [[0.    0.859 0.902 1.    0.707 0.684]]
Iteration 197
Acquisition function convergence reached at iteration 183.
The final UCB loss was -23.532 with predicted mean of [-1.135]
The next parameters to simulate from are [[0.999 0.425 0.4   0.391 0.984 0.927]]
Iteration 198
Acquisition function convergence reached at iteration 312.
The final UCB loss was -23.379 with predicted mean of [-0.978]
The next parameters to simulate from are [[1.    0.341 1.    0.111 0.984 0.799]]
Iteration 199
Acquisition function convergence reached at iteration 162.
The final UCB loss was -23.724 with predicted mean of [-1.313]
The next parameters to simulate from are [[1.    0.32  0.649 0.635 0.998 0.41 ]]
Iteration 200
Acquisition function convergence reached at iteration 106.
The final UCB loss was -23.566 with predicted mean of [-1.086]
The next parameters to simulate from are [[0.    0.602 1.    0.462 0.434 0.801]]
Trained parameters:
amplitude_champ:0 is 0.782
```

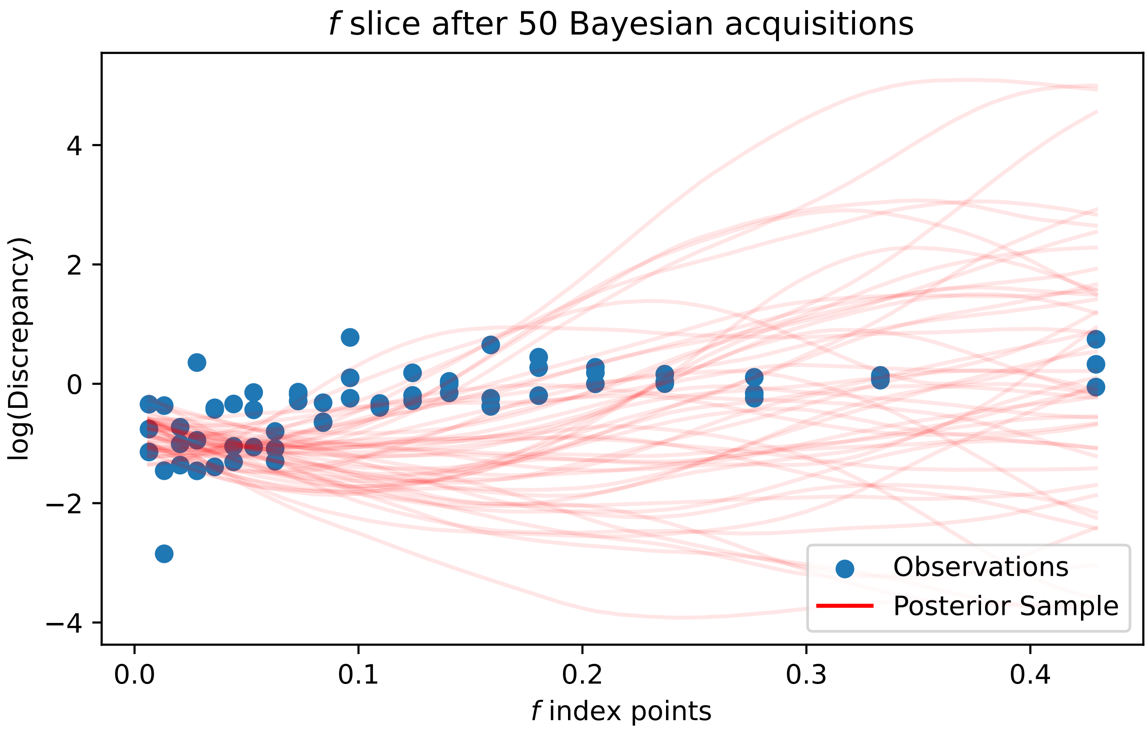
```
length_scales_champ:0 is [0.473 1.    0.04  0.095 0.185 0.019]

observation_noise_variance_champ:0 is 0.229

amp_f_mean:0 is 0.632

amp_gamma_L_mean:0 is 0.055

amp_lambda_mean:0 is 0.615

amp_r_mean:0 is 0.032

bias_mean:0 is 0.407

f_tp:0 is 0.401

gamma_L_tp:0 is 0.596

lambda_tp:0 is 0.104

r_tp:0 is 0.912
```

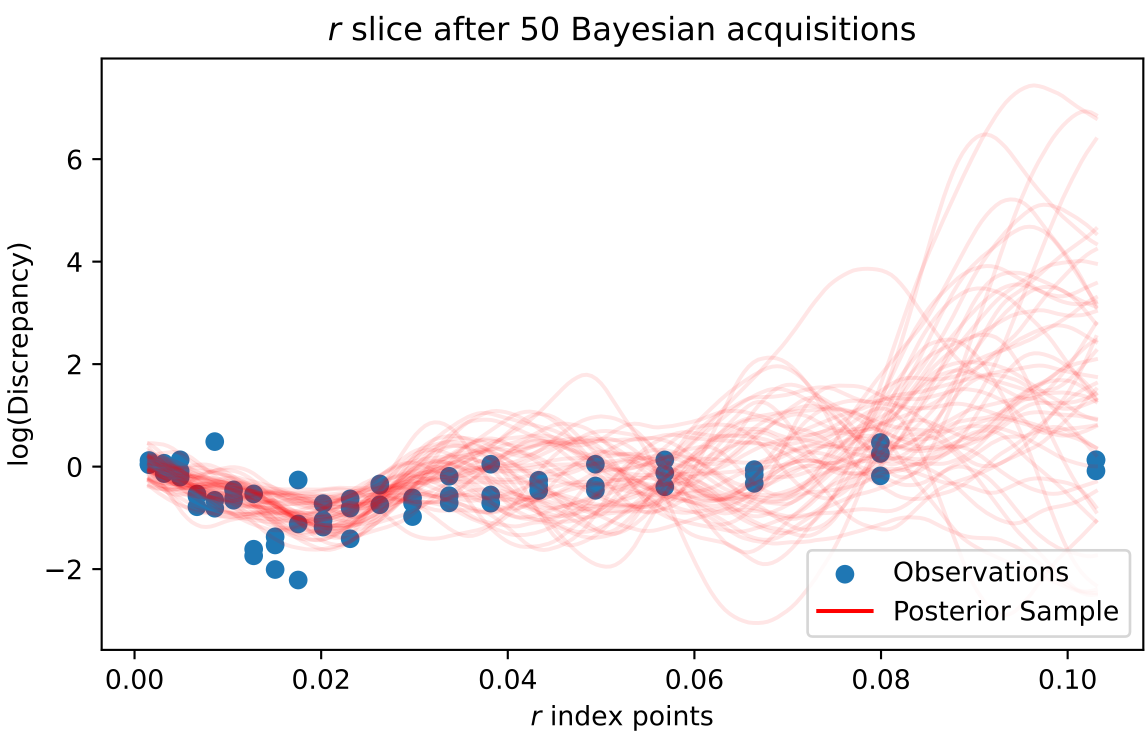

α slice after 50 Bayesian acquisitions
log(Discrepancy)
α index points
Observations
Posterior Sample
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.0
0.2
0.4
0.6
0.8
1.0

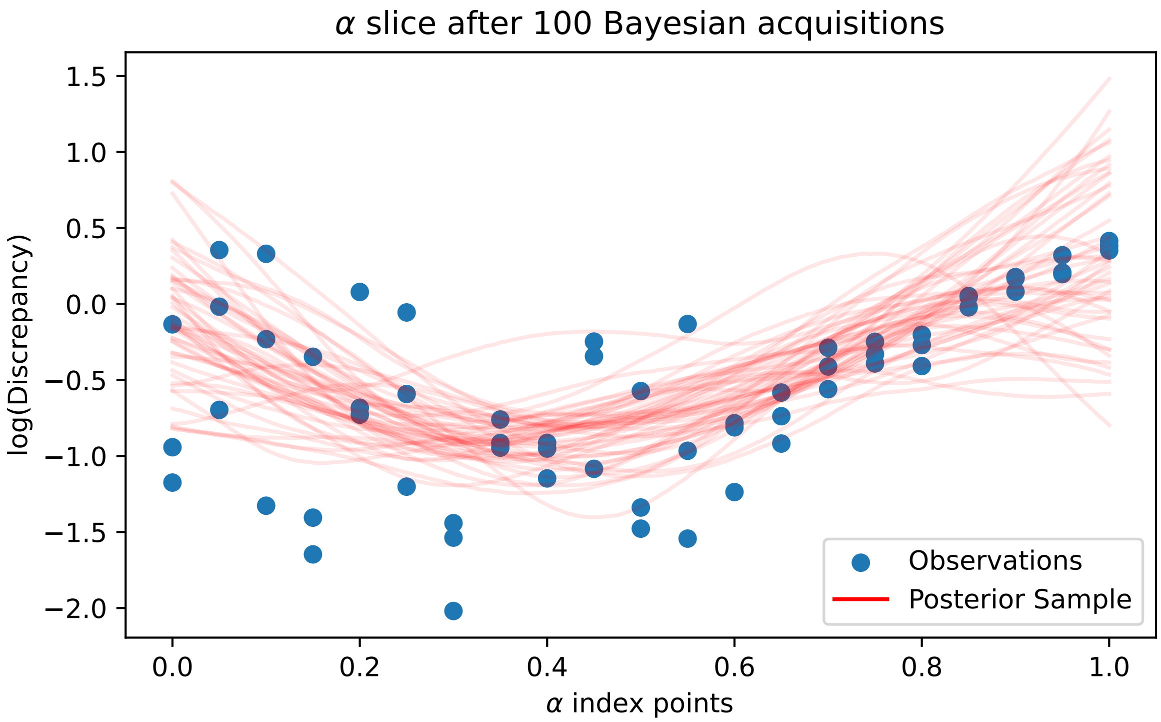

β slice after 50 Bayesian acquisitions
log(Discrepancy)
β index points
Observations
Posterior Sample
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0.0
0.2
0.4
0.6
0.8
1.0

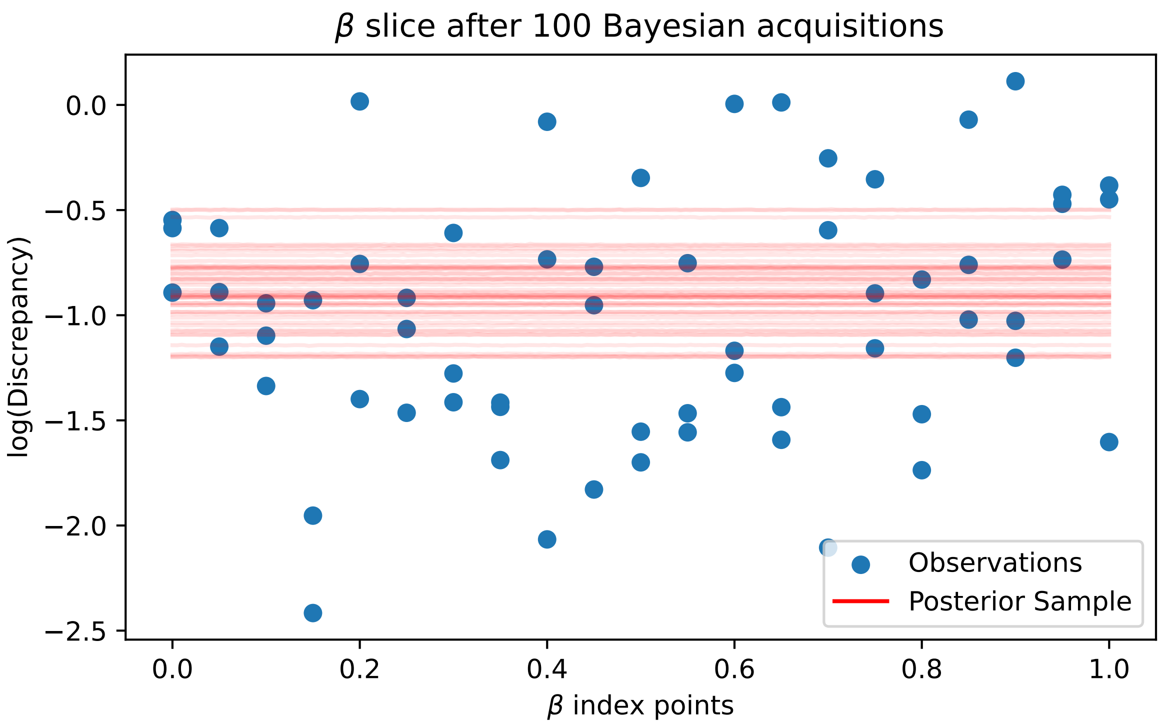

γ_L slice after 50 Bayesian acquisitions
log(Discrepancy)
γ_L index points
Observations
Posterior Sample
10
8
6
4
2
0
−2
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14

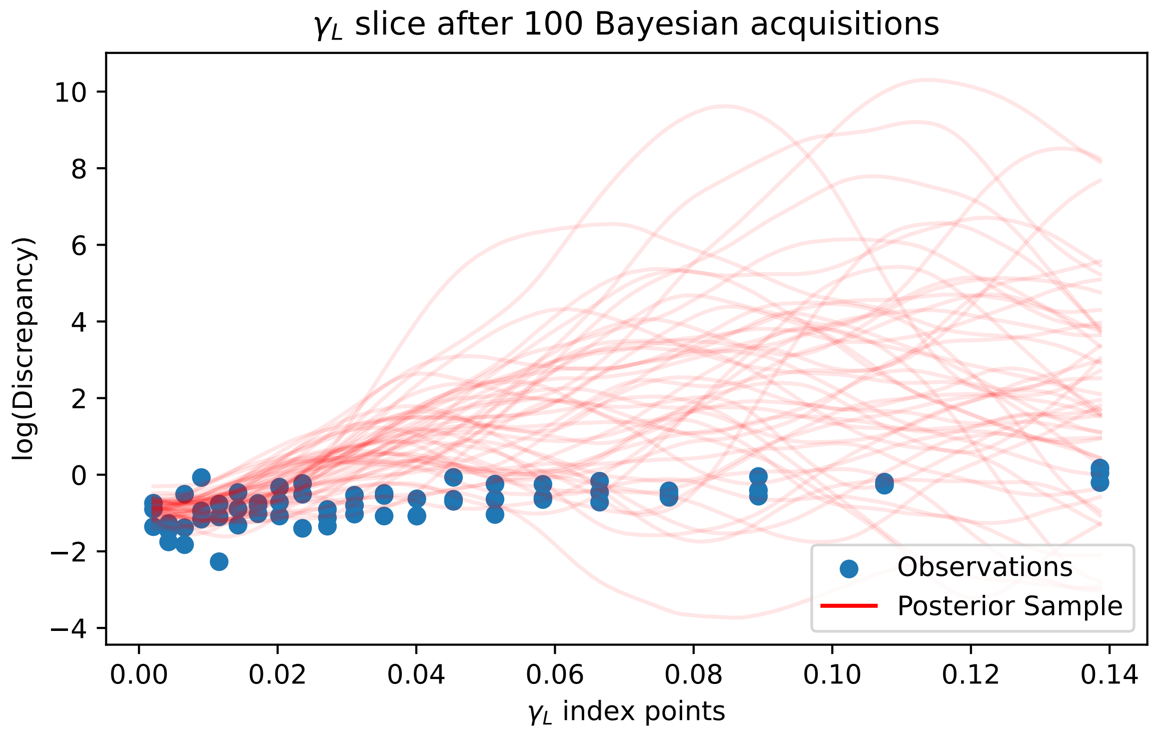

λ slice after 50 Bayesian acquisitions
log(Discrepancy)
λ index points
Observations
Posterior Sample
3
2
1
0
−1
−2
0.00
0.02
0.04
0.06
0.08
0.10
0.12

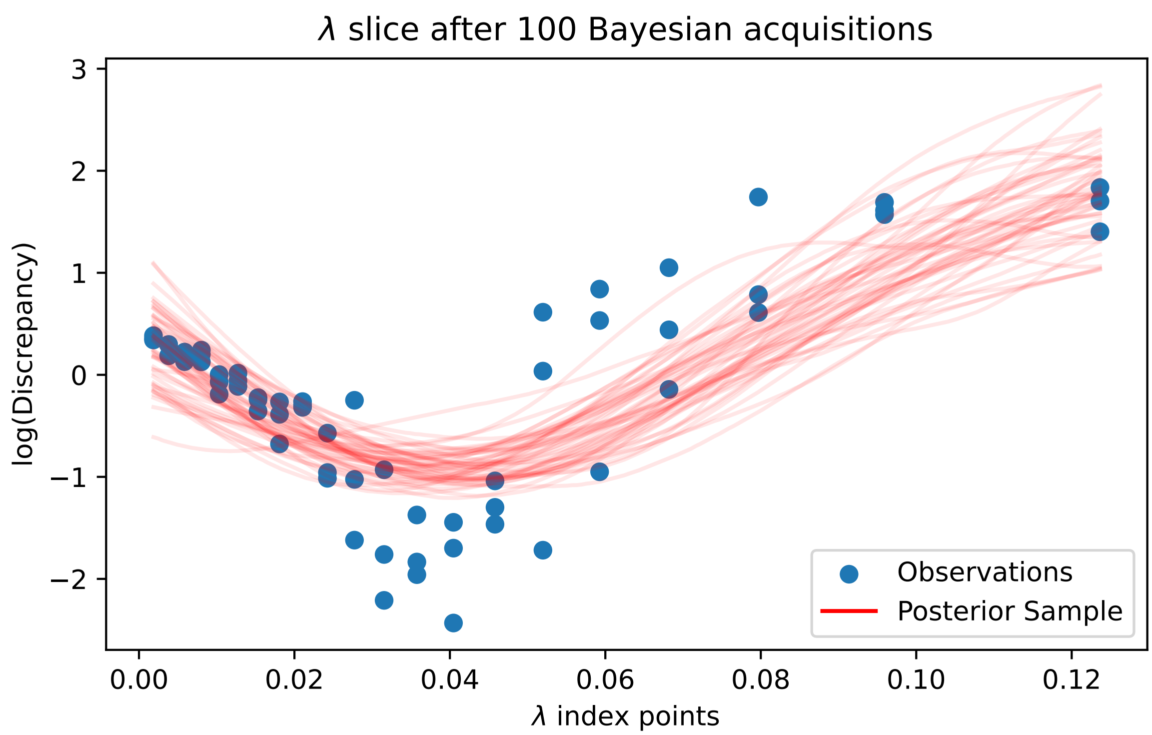

f slice after 50 Bayesian acquisitions
log(Discrepancy)
4
2
0
−2
−4
0.0
0.1
0.2
0.3
0.4
f index points
Observations
Posterior Sample

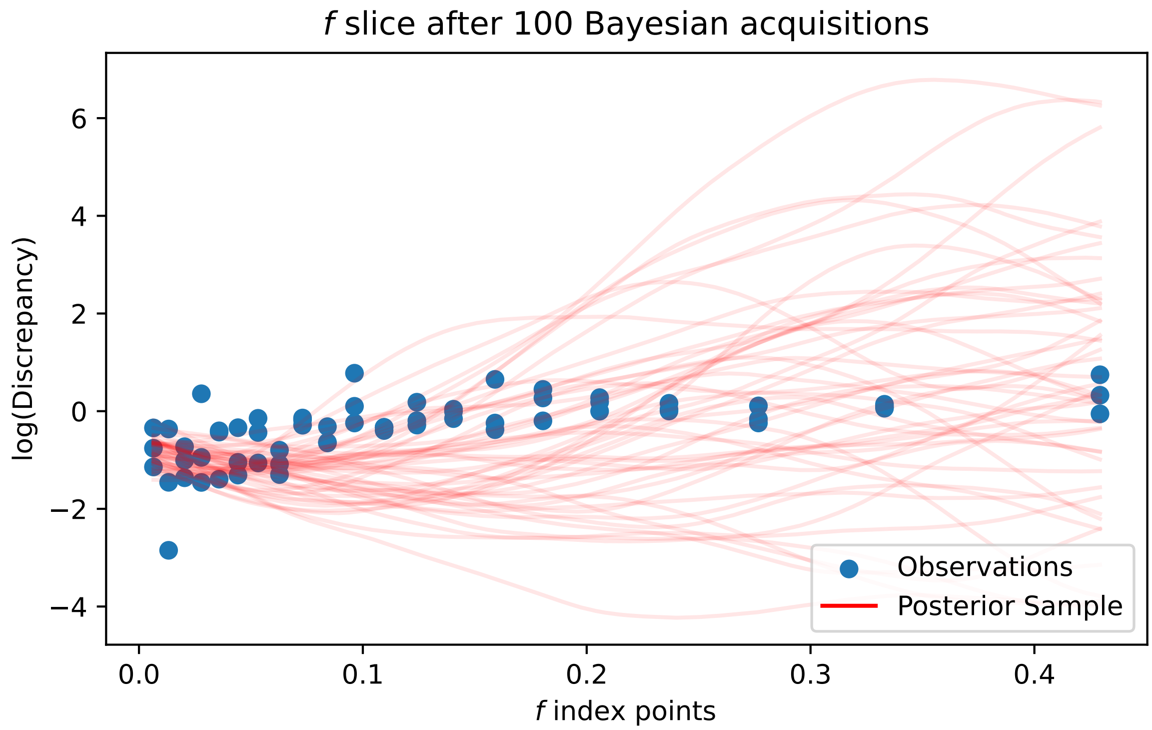

r slice after 50 Bayesian acquisitions
log(Discrepancy)
6
4
2
0
−2
0.00
0.02
0.04
0.06
0.08
0.10
r index points
Observations
Posterior Sample

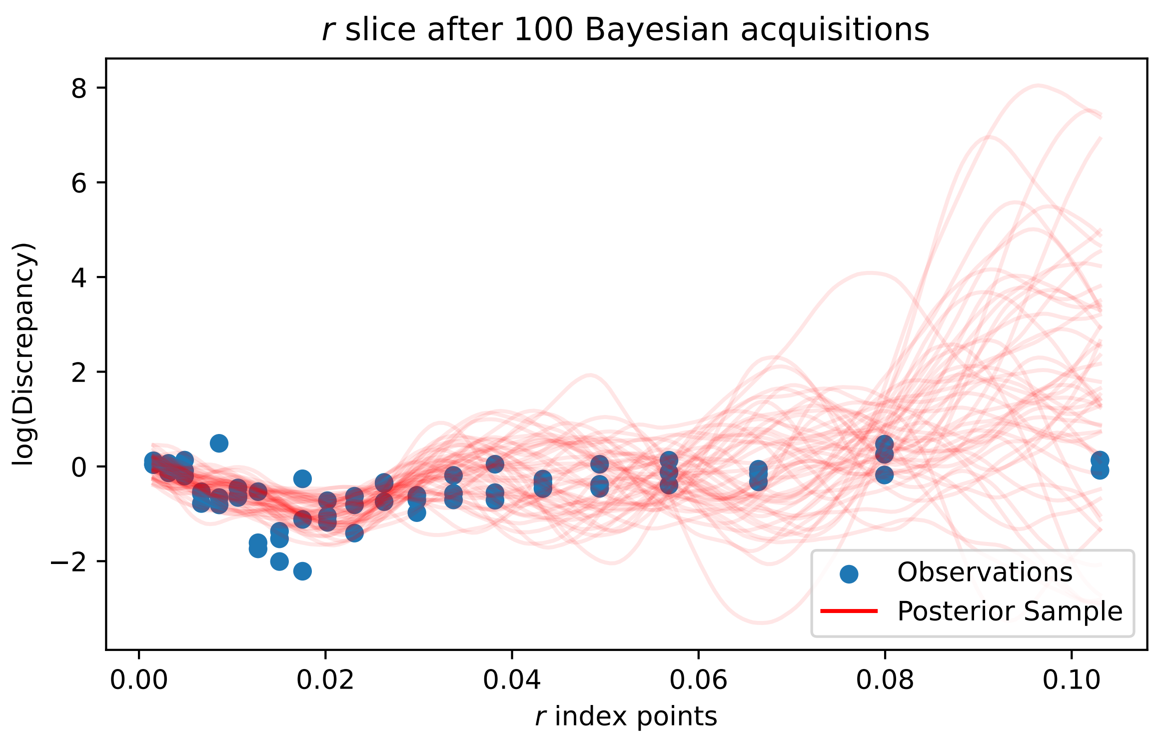

α slice after 100 Bayesian acquisitions
log(Discrepancy)
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.0
0.2
0.4
0.6
0.8
1.0
α index points
Observations
Posterior Sample

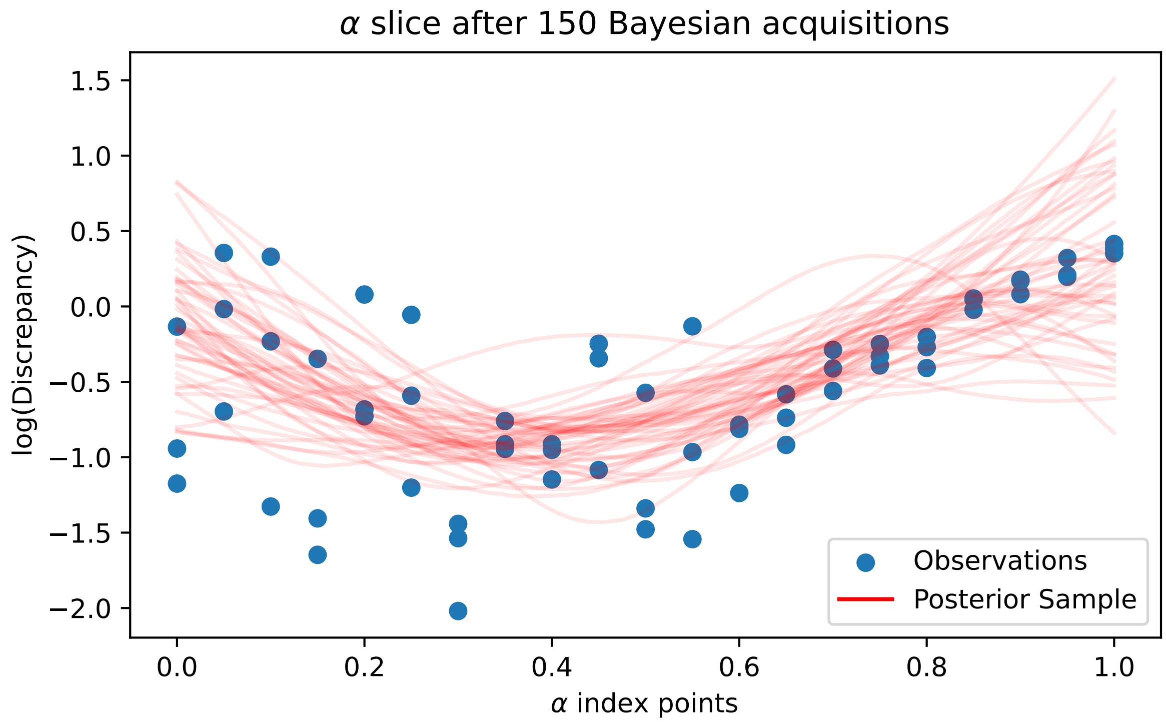

β slice after 100 Bayesian acquisitions
log(Discrepancy)
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0.0
0.2
0.4
0.6
0.8
1.0
β index points
Observations
Posterior Sample

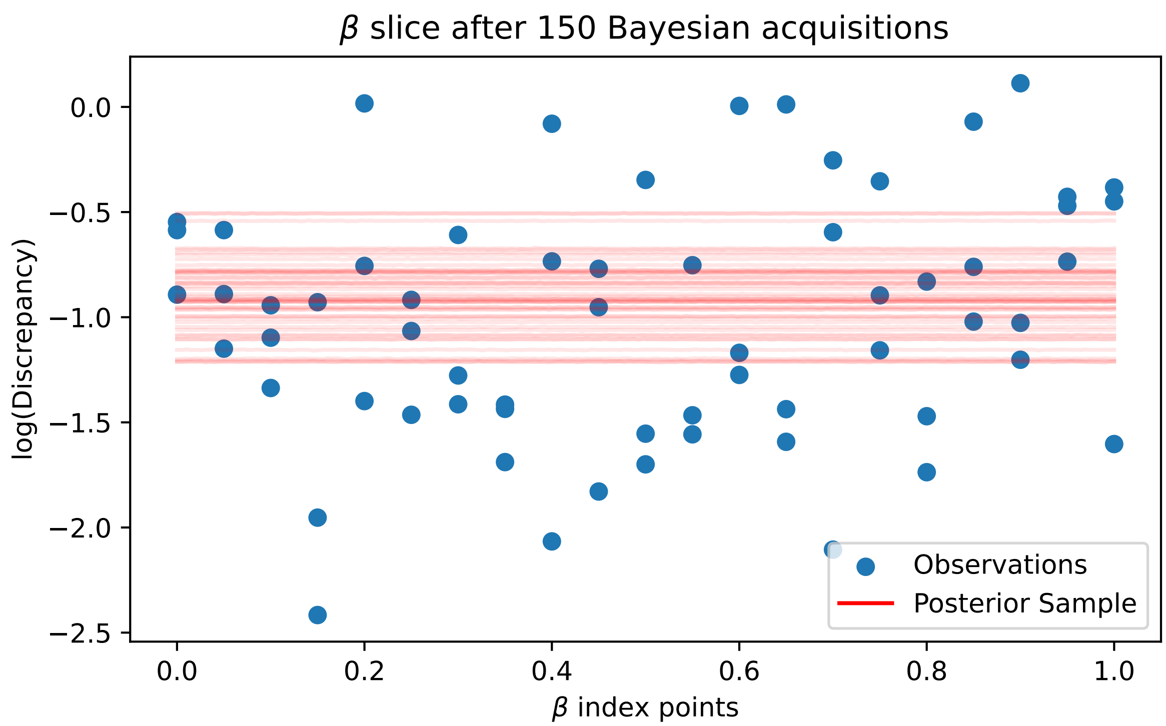

γ_L slice after 100 Bayesian acquisitions
log(Discrepancy)
γ_L index points
Observations
Posterior Sample

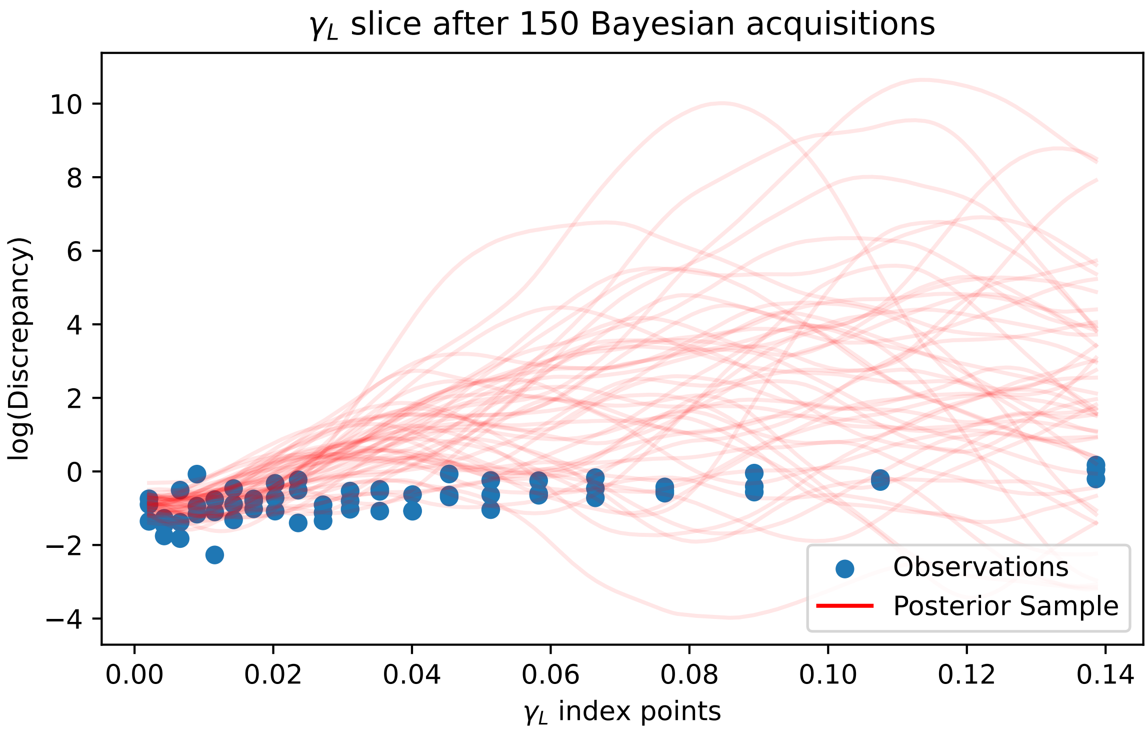

λ slice after 100 Bayesian acquisitions
log(Discrepancy)
λ index points
Observations
Posterior Sample


f slice after 100 Bayesian acquisitions
log(Discrepancy)
f index points
Observations
Posterior Sample

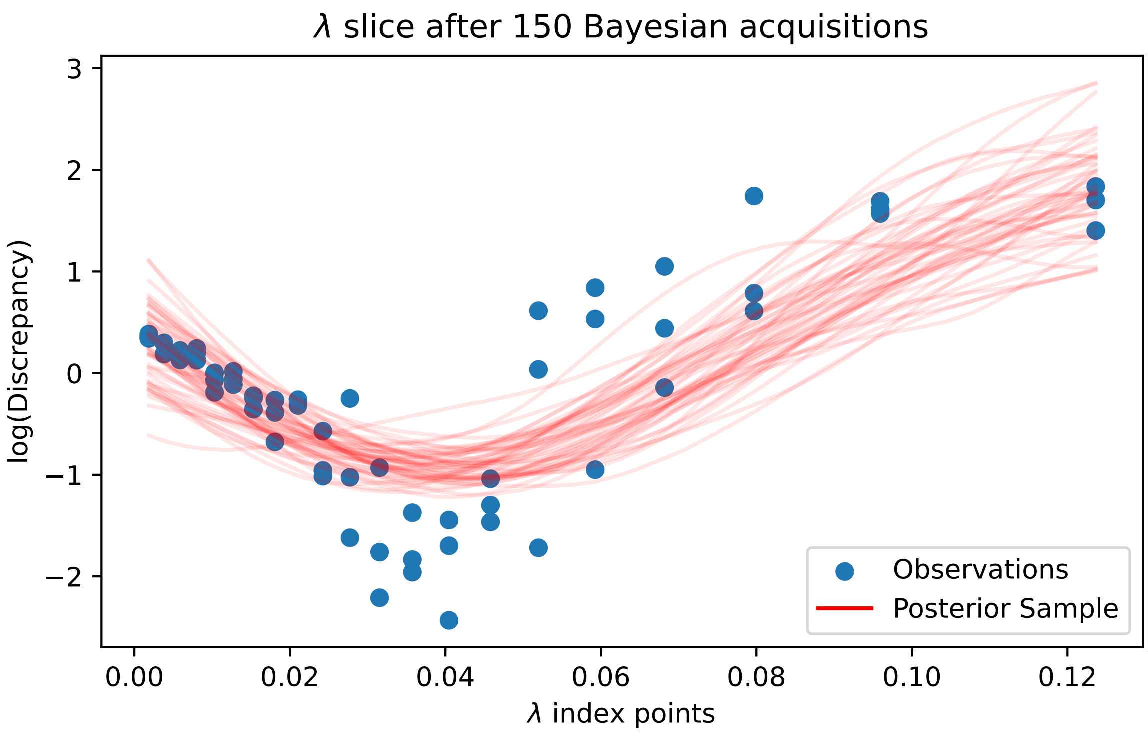

r slice after 100 Bayesian acquisitions
log(Discrepancy)
r index points
Observations
Posterior Sample


α slice after 150 Bayesian acquisitions
log(Discrepancy)
α index points
Observations
Posterior Sample

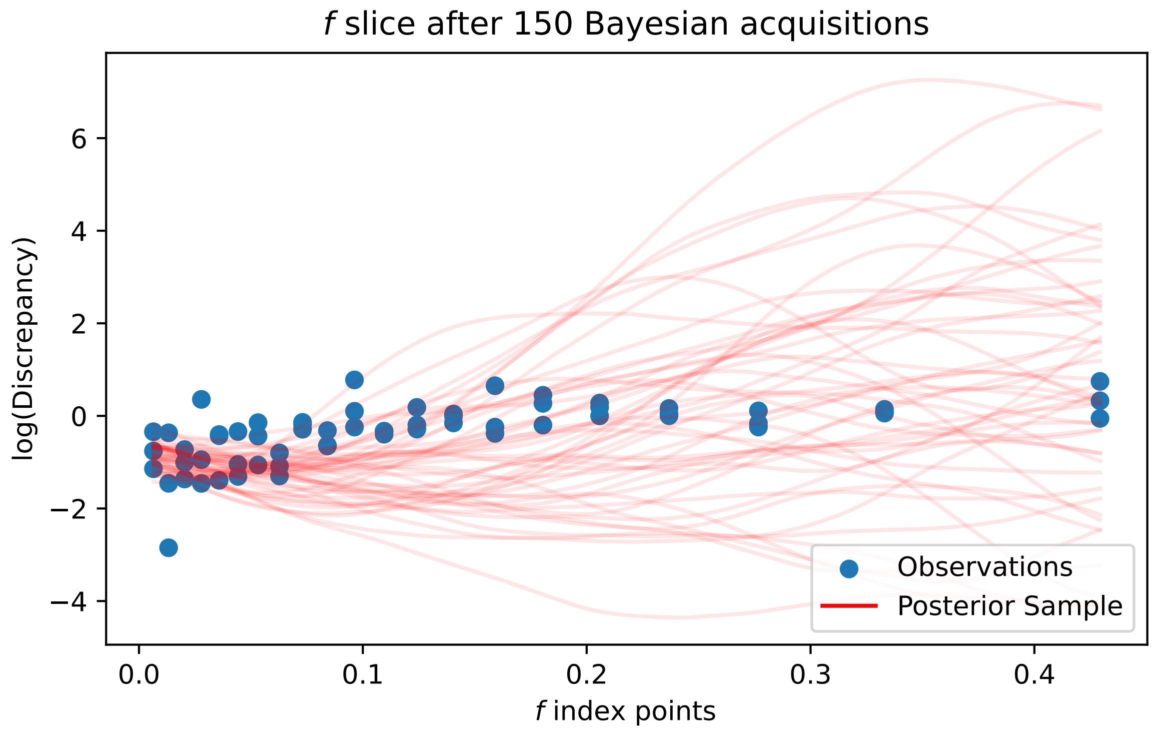

β slice after 150 Bayesian acquisitions
log(Discrepancy)
β index points
Observations
Posterior Sample



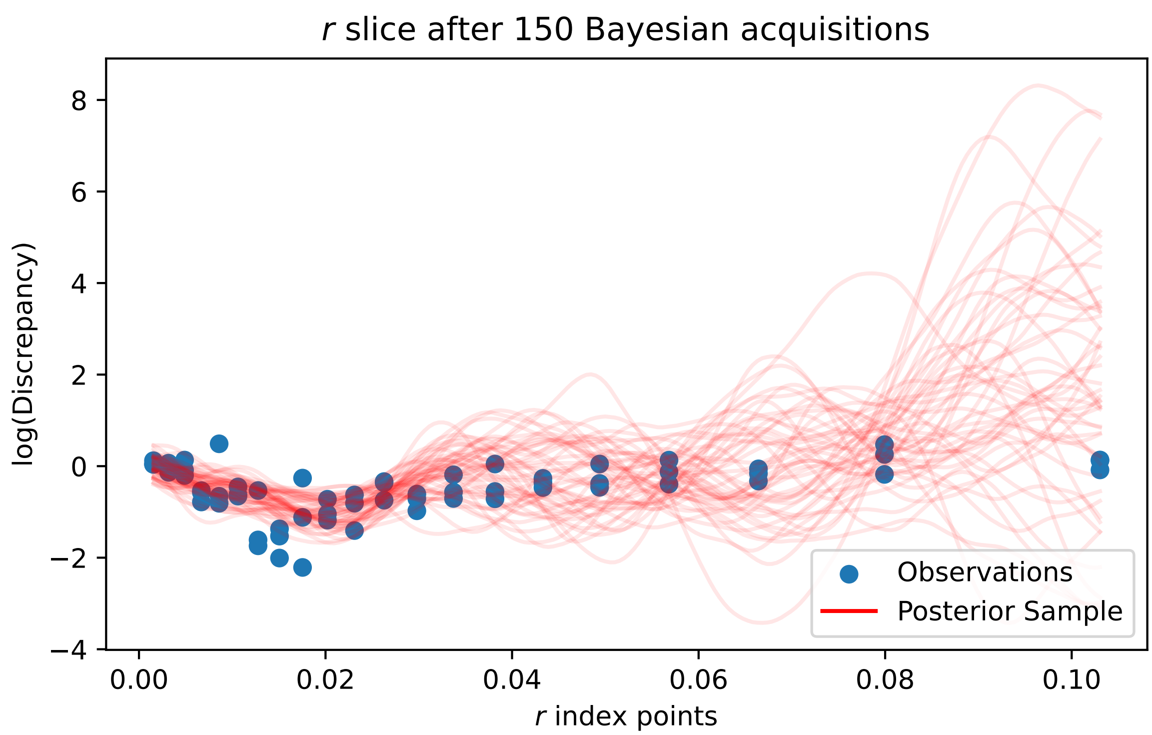
γ_L slice after 150 Bayesian acquisitions
log(Discrepancy)
γ_L index points
Observations
Posterior Sample



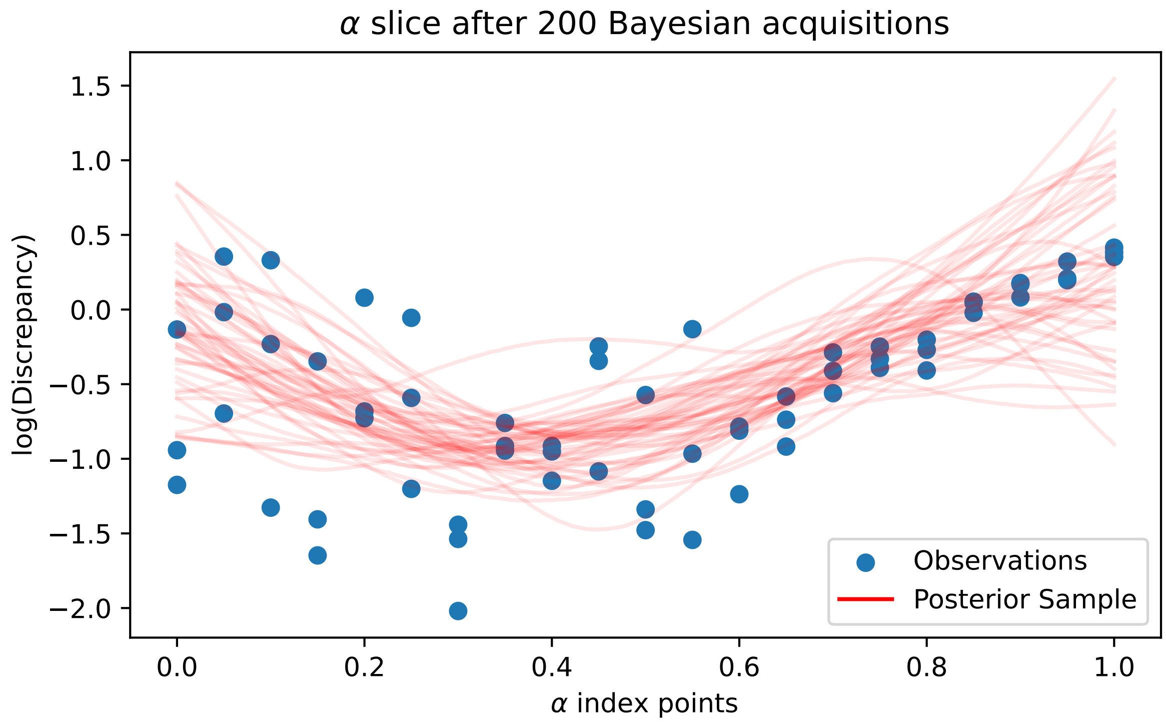
λ slice after 150 Bayesian acquisitions
log(Discrepancy)
λ index points
Observations
Posterior Sample

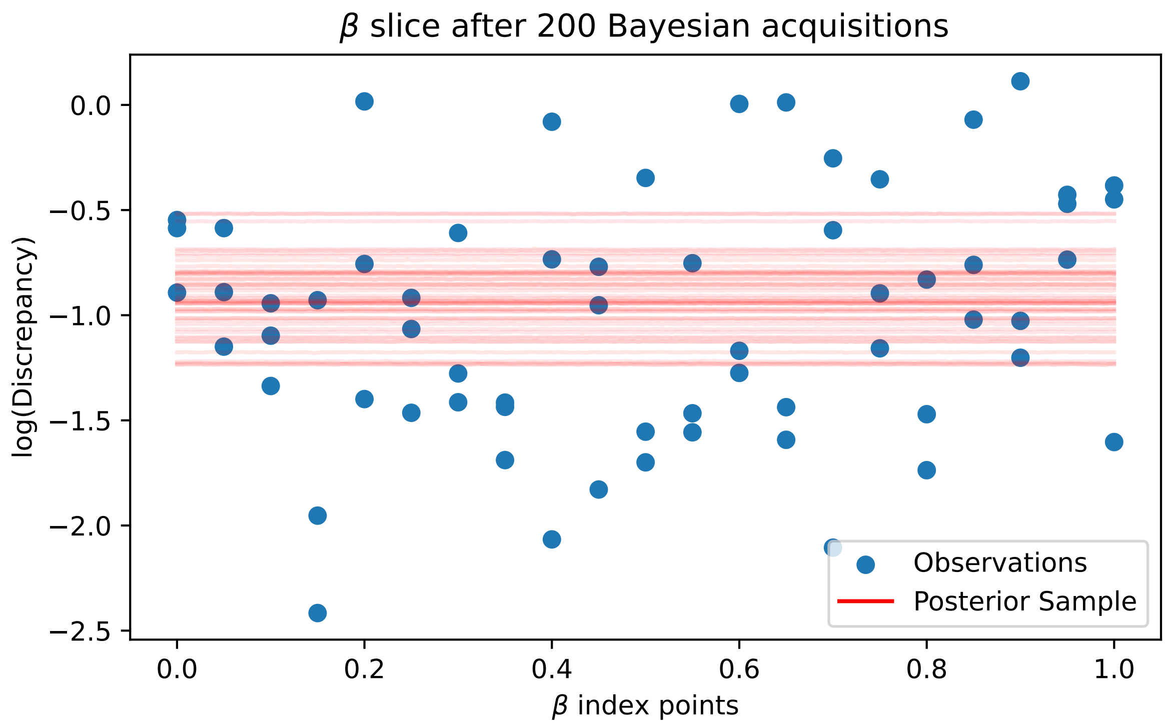


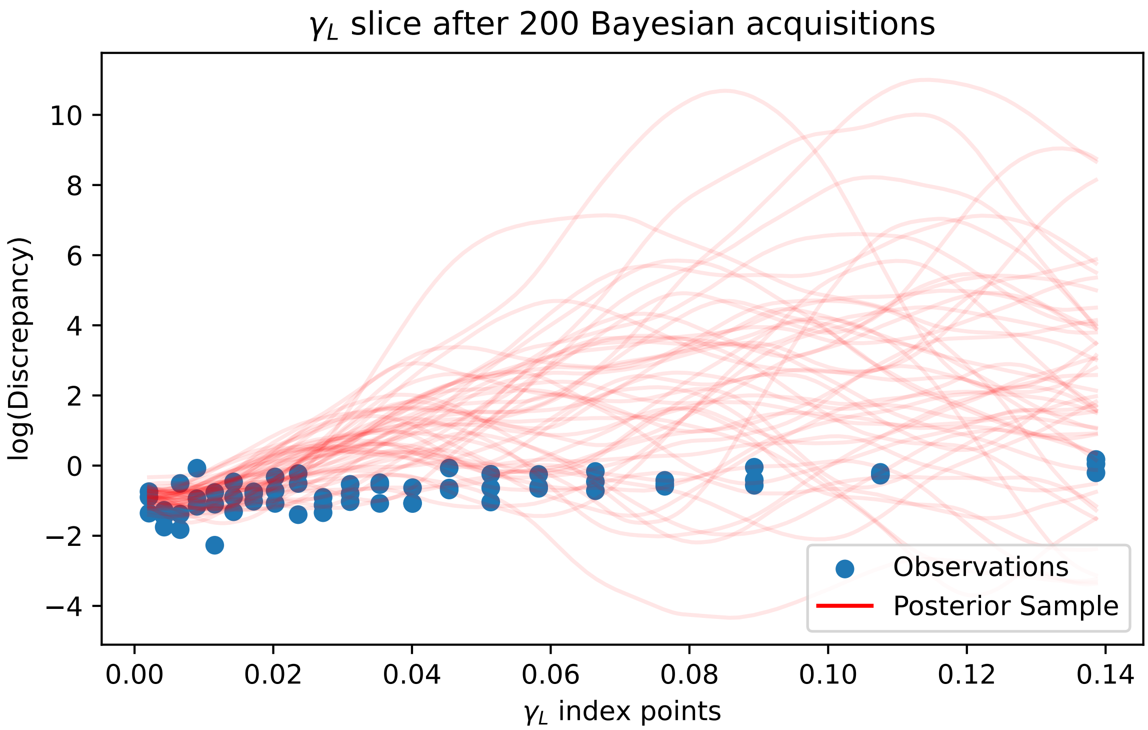
f slice after 150 Bayesian acquisitions
log(Discrepancy)
f index points
Observations
Posterior Sample

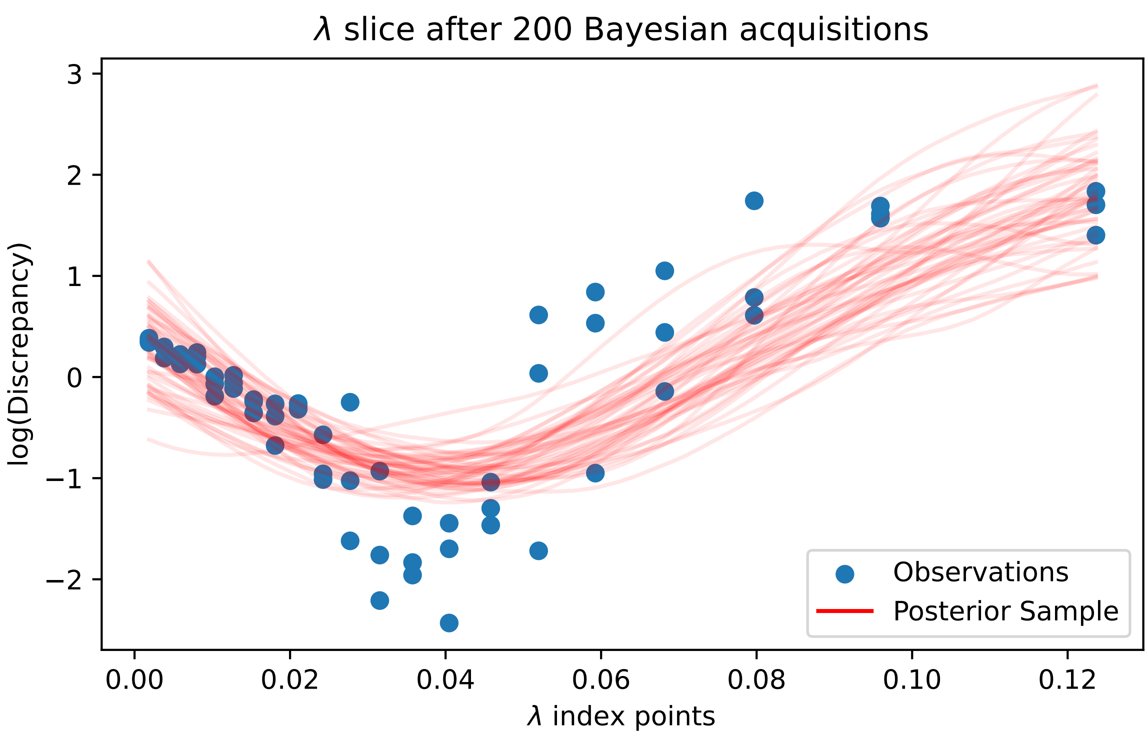


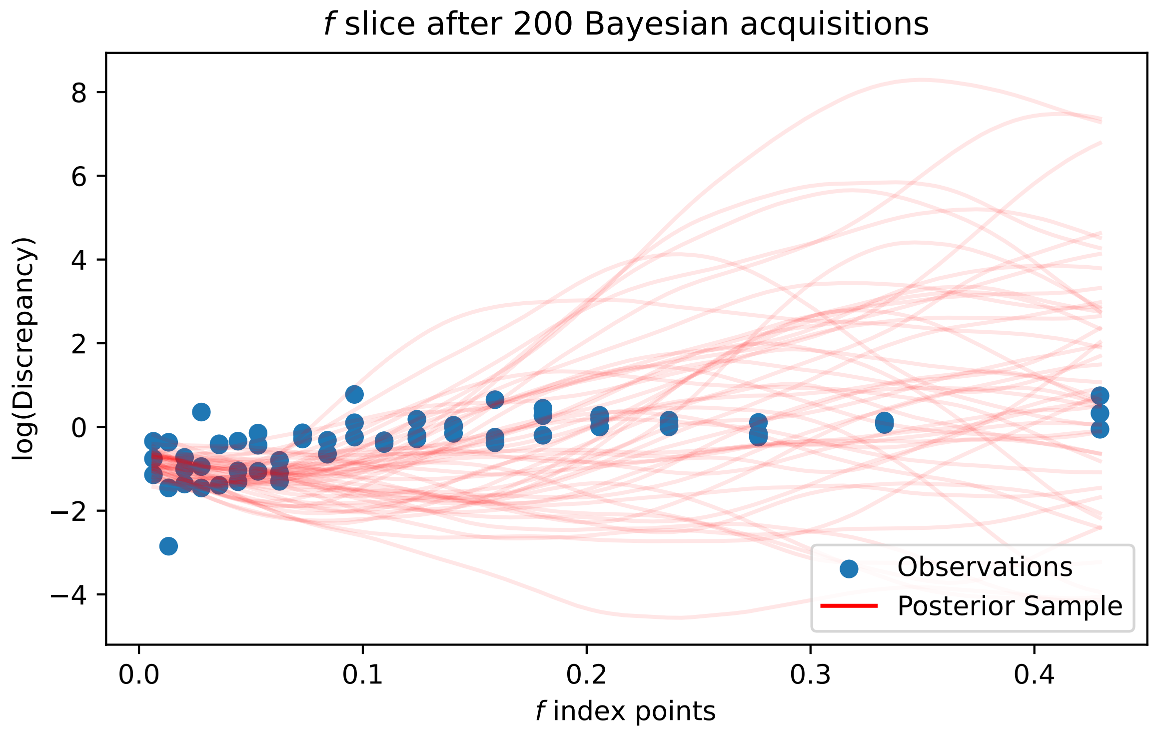
r slice after 150 Bayesian acquisitions
log(Discrepancy)
r index points
Observations
Posterior Sample

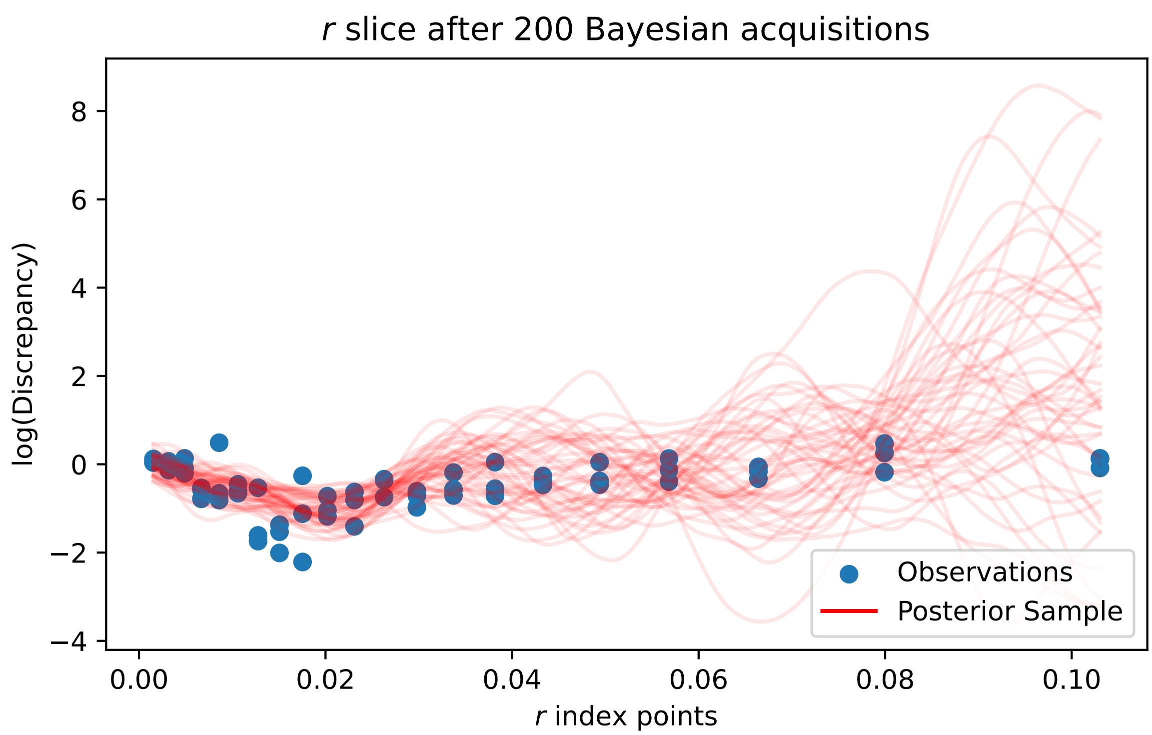

α slice after 200 Bayesian acquisitions
log(Discrepancy)
α index points
Observations
Posterior Sample


β slice after 200 Bayesian acquisitions
log(Discrepancy)
β index points
Observations
Posterior Sample


γ_L slice after 200 Bayesian acquisitions
log(Discrepancy)
γ_L index points
Observations
Posterior Sample


λ slice after 200 Bayesian acquisitions
log(Discrepancy)
λ index points
Observations
Posterior Sample


f slice after 200 Bayesian acquisitions
log(Discrepancy)
8
6
4
2
0
−2
−4
0.0
0.1
0.2
0.3
0.4
f index points
Observations
Posterior Sample


r slice after 200 Bayesian acquisitions
log(Discrepancy)
8
6
4
2
0
−2
−4
0.00
0.02
0.04
0.06
0.08
0.10
r index points
Observations
Posterior Sample

```
# print(index_vals[-600,].round(3))
# print(index_vals[-400,].round(3))
print(index_vals[-200,].round(3))
print(index_vals[-80,].round(3))
print(index_vals[-40,].round(3))
print(index_vals[-20,].round(3))
print(index_vals[-8,].round(3))
print(index_vals[-4,].round(3))
print(index_vals[-2,].round(3))
print(index_vals[-1,].round(3))
```

```
[1.    0.846 0.795 1.    1.    0.112]
[0.    0.974 0.475 0.952 0.996 0.31 ]
[0.    0.799 1.    0.215 0.463 1.   ]
[0.002 0.647 0.309 0.98  0.996 0.362]
[0.999 0.425 0.4   0.391 0.984 0.927]
[1.    0.32  0.649 0.635 0.998 0.41 ]
[0.    0.602 1.    0.462 0.434 0.801]
[0.    0.602 1.    0.462 0.434 0.801]
```