bayesian_lasso_with_me

December 6, 2023

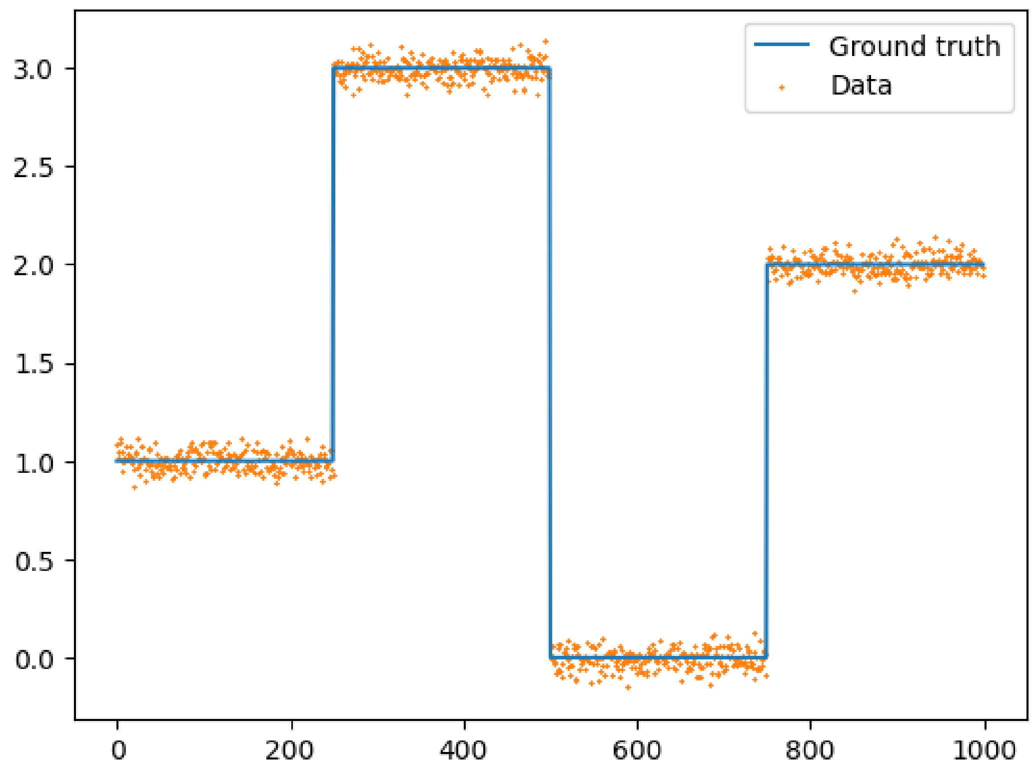
The goal of this notebook is to implement the Bayesian LASSO method for a 1D problem.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

from scipy.stats import recipinvgauss
import scipy.sparse as sps
from fastprogress import progress_bar

from IPython.display import clear_output, DisplayHandle
def update_patch(self, obj):
    clear_output(wait=True)
    self.display(obj)
DisplayHandle.update = update_patch

from runningstatistics import StatsTracker
import jlinops
```

1 Make toy problem

[2]:
```
ground_truth = jlinops.piecewise_constant_1d_test_problem()
n = len(ground_truth)
np.random.seed(0)
noise_stdev = 0.05
noise_var = noise_stdev**2
noisy_signal = ground_truth + noise_stdev*np.random.normal(size=n)
grid = np.arange(n)
```

[3]:
```
plt.plot(grid, ground_truth, label="Ground truth", color="C0")
plt.scatter(grid, noisy_signal, marker="x", label="Data", color="C1", alpha=1.
  ↪0, s=0.5)
plt.legend()
plt.show()
```



[4]:
```
# Define forward operator and regularization matrix
F = jlinops.MatrixLinearOperator(sps.eye(n))
R, _ = jlinops.first_order_derivative_1d(n, boundary="none")
R = jlinops.MatrixLinearOperator(R)

# Set regularization lambda
reg_lambda = 1e2
```

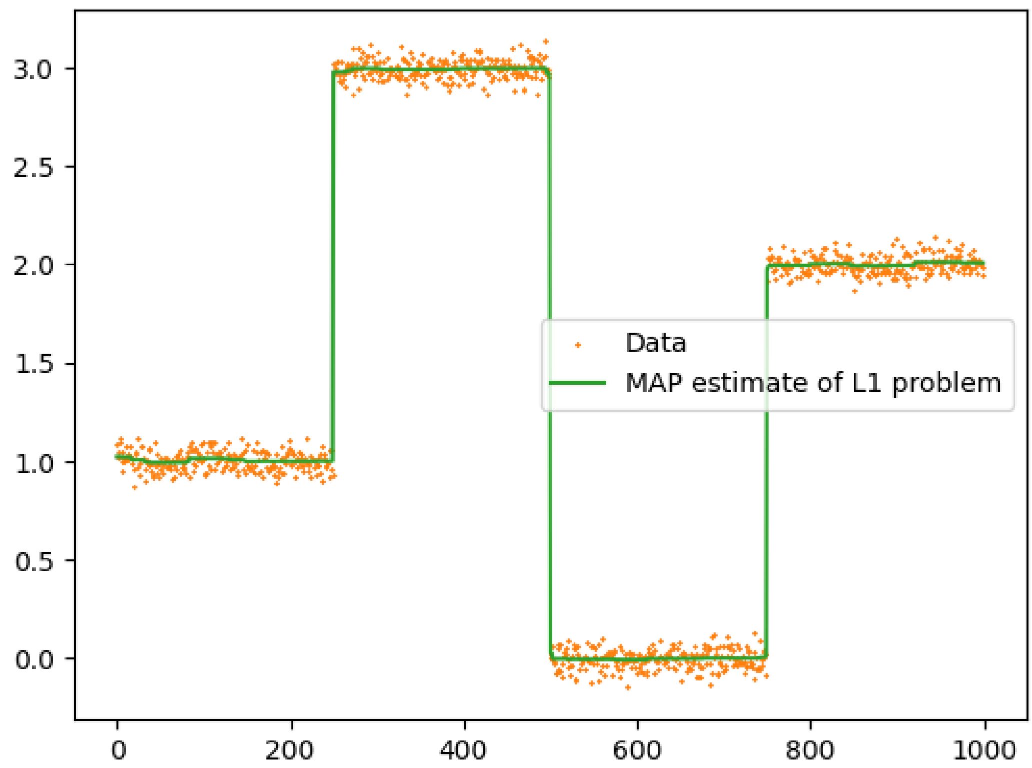
2 MAP estimate of the L1 problem

By L1 problem, I mean solving

$$\text{argmin}_x \left\{ \frac{1}{2\sigma^2} \|x - y\|_2^2 + \lambda \|Rx\|_1 \right\} = \text{argmin}_x \left\{ \frac{1}{2} \|x - y\|_2^2 + (\lambda\sigma^2) \|Rx\|_1 \right\}.$$

[5]:
```
# Solution is given by evaluating the proximal operator of the TV norm. This␣
↪code uses a FDGP method
fdgp_map_result = jlinops.prox_tv1d_norm(noisy_signal,␣
↪lam=noise_var*reg_lambda, iterations=1000)
```

[6]:
```
plt.scatter(grid, noisy_signal, marker="x", label="Data", color="C1", alpha=1.
↪0, s=0.5)
plt.plot(grid, fdgp_map_result, label="MAP estimate of L1 problem", color="C2")
plt.legend()
plt.show()
```

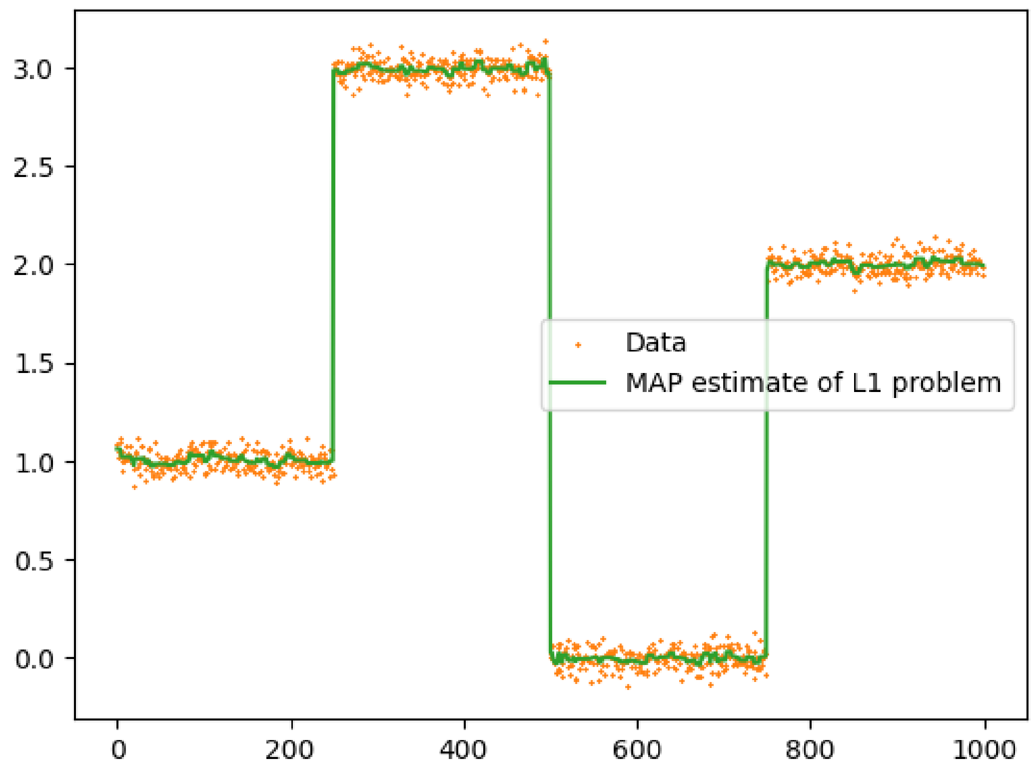


[7]:
```
reg_lambda
```

[7]: 100.0

[8]:
```
# Solution is given by evaluating the proximal operator of the TV norm. This␣
↪code uses a FDGP method
fdgp_map_result = jlinops.prox_tv1d_norm(noisy_signal, lam=noise_var*25,␣
↪iterations=100)
```

[9]:
```
plt.scatter(grid, noisy_signal, marker="x", label="Data", color="C1", alpha=1.
↪0, s=0.5)
plt.plot(grid, fdgp_map_result, label="MAP estimate of L1 problem", color="C2")
plt.legend()
plt.show()
```



3 Sample the posterior of the Gaussian model

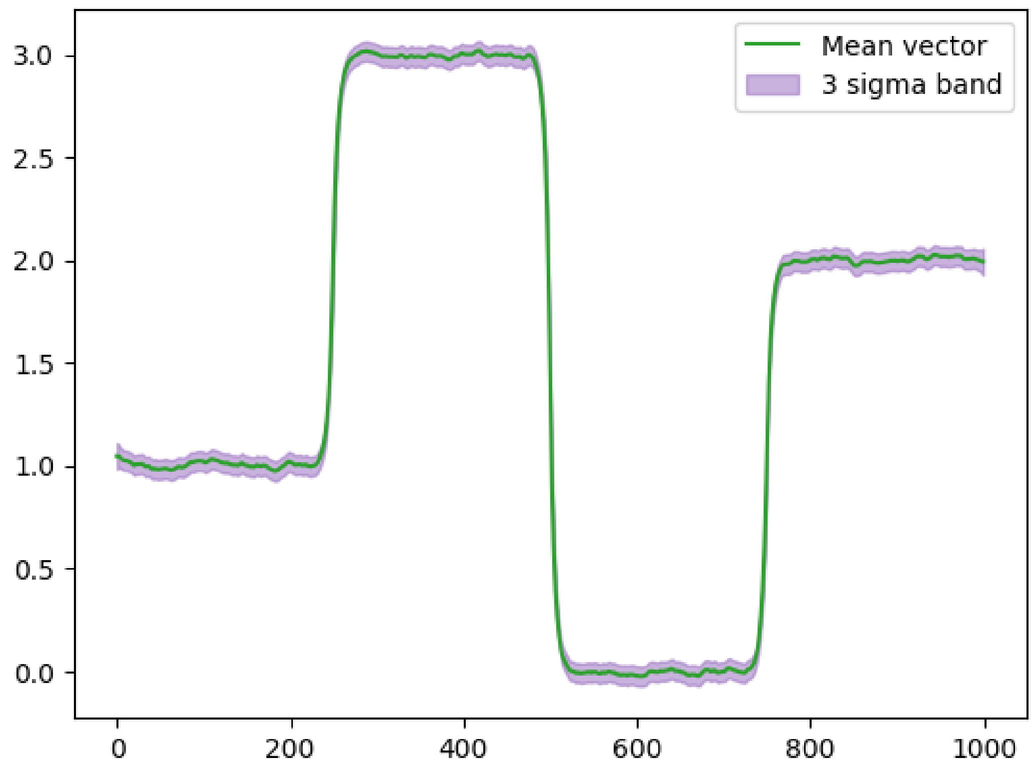
$$-\log \pi(x) = \left\{ \frac{1}{2\sigma^2} \|x - y\|_2^2 + \frac{\lambda}{2} \|Rx\|_2^2 \right\} + C$$

[10]:
```
def gauss_posterior_summary(F, R, y, noise_var=1.0, reg_lambda=1e1):
    """Computes posterior mean and stdev.
    """
    Q = sps.csc_matrix((1/noise_var)*(F.A.T @ F.A) + reg_lambda*(R.A.T @ R.A))
    Q = jlinops.MatrixLinearOperator(Q)
    Linv = jlinops.BandedCholeskyFactorInvOperator(Q)
    mean = Linv.T @ (Linv @  ((1/noise_var)*F.T @ y) )

    # Get diagonal entries of Qinv
    Qinv = Linv.T @ Linv
    var = jlinops.black_box_diagonal(Qinv)
    stdev = np.sqrt(var)
    return mean, stdev
```

```
[11]: gauss_mu, gauss_sigmas = gauss_posterior_summary(F, R, noisy_signal,␣
     ↪noise_var=noise_var, reg_lambda=1e4)
```

<IPython.core.display.HTML object>

```
[12]: #plt.scatter(grid, noisy_signal, marker="x", label="Data", color="C1", alpha=1.
     ↪0, s=0.5)
     plt.plot(grid, gauss_mu, label="Mean vector", color="C2")
     plt.fill_between(grid, gauss_mu - 3*gauss_sigmas, gauss_mu + 3*gauss_sigmas,␣
     ↪color="C4", alpha=0.5, label="3 sigma band")
     plt.legend()
     plt.show()
```



4 Bayesian LASSO

```
[13]: class BayesianLASSOGibbsSampler:
         """Implements the Bayesian LASSO hierarchical sampler for the L1 problem.
         """

         def __init__(self, F, R, y, noise_var=1.0):
```

```
        self.F = F
        self.R = R
        self.y = y
        self.noise_var = noise_var
        self.reg_lambda = reg_lambda


    def sample(self, n_samples, x0=None, n_burn=0, theta_tol=1e-2, lam0=None,␣
↪lam_update_freq=25):
        """Runs the Gibbs sampler.
        """

        # Initialize
        if x0 is None:
            x = np.zeros(self.F.shape[1])
        else:
            x = x0

        if lam0 is None:
            lam = 1.0
        else:
            lam = lam0

        # Create trackers
        x_tracker = StatsTracker(self.F.shape[1])
        theta_tracker = StatsTracker(self.R.shape[0])

        # For taking care of lambda udpates
        lam_update_fn = lambda theta_ss_est: np.sqrt( 2*self.R.shape[0]/
↪theta_ss_est )
        theta_sum_tracker = StatsTracker((1,))
        lam_hist = [lam]
        theta_sums_all = []

        # Run the sampler
        for j in progress_bar(range(n_samples+n_burn)):

            # Update theta
            theta = self.sample_theta(x, tol=theta_tol)

            # Update x
            x = self.sample_x(theta)

            # For taking care of lambda updates
            theta_sums_all.append(theta.sum())
            theta_sum_tracker.push(theta.sum())
```

```python
        if (j < n_burn) and (j%lam_update_freq == 0):

            # Get new lambda
            lam = lam_update_fn(theta_sum_tracker.mean()[0])
            lam_hist.append(lam)

            # Reset theta sum tracker
            theta_sum_tracker = StatsTracker((1,))

        # Push to tracker
        if j >= n_burn:
            x_tracker.push(x)
            theta_tracker.push(theta)


    results = {
        "x_tracker": x_tracker,
        "theta_tracker": theta_tracker,
        "lam_hist": np.asarray(lam_hist),
        "theta_sums_all": np.asarray(theta_sums_all),
    }

    return results


def sample_x(self, theta):
    """Given local variances theta, draws a sample for x.
    """

    Q = (1.0/self.noise_var)*(self.F.A.T @ self.F.A) + (1/2)*(self.R.A.T @
↪( sps.diags(1.0/theta) @ self.R.A ) )

    # # Bad way
    # Qinv = np.linalg.inv(Q.toarray())
    # mean = Qinv @ ((1.0/self.noise_var)*self.F.T @ self.y )
    # sample = np.random.multivariate_normal(mean, Qinv)

    # Good way
    Q = sps.csc_matrix(Q)
    Q = jlinops.MatrixLinearOperator(Q)
    Linv = jlinops.BandedCholeskyFactorInvOperator(Q)
    mean = Linv.T @ (Linv @ ((1.0/self.noise_var)*self.F.T @ self.y ) )
    sample = mean + ( Linv.T @ np.random.normal(size=Q.shape[0]) )

    return sample
```

```
    def sample_theta(self, x, tol=1e-2):
        """Given x, draws a sample for the thetas.
        """

        # Get Rx
        Rx = self.R @ x

        # Make output array
        sample = np.zeros(self.R.shape[0])


        # Need to check where Rx is close to zero, so we can sample from␣
↪exponential there instead
        idx_too_small = np.where(np.abs(Rx) < tol)
        idx_fine = np.where(np.abs(Rx) >= tol)

        # Break into two parts
        Rx_too_small = Rx[idx_too_small]
        Rx_fine = Rx[idx_fine]

        # For the components near zero, sample from the exponential
        theta_from_too_small = np.random.exponential(scale=1.0/self.reg_lambda,␣
↪size=len(Rx_too_small))

        # For the components not near zero, sample from the inverse Gaussian
        theta_from_fine = recipinvgauss.rvs(mu=1.0/(self.reg_lambda*np.
↪abs(Rx_fine)), scale=1.0/(self.reg_lambda**2))

        # Put all into one array
        sample[idx_too_small] = theta_from_too_small
        sample[idx_fine] = theta_from_fine

        assert np.all(sample > 0), "some thetas are no positive!"

        return sample
```

[14]:
```
lasso_sampler = BayesianLASSOGibbsSampler(F, R, noisy_signal,␣
↪noise_var=noise_var)
```

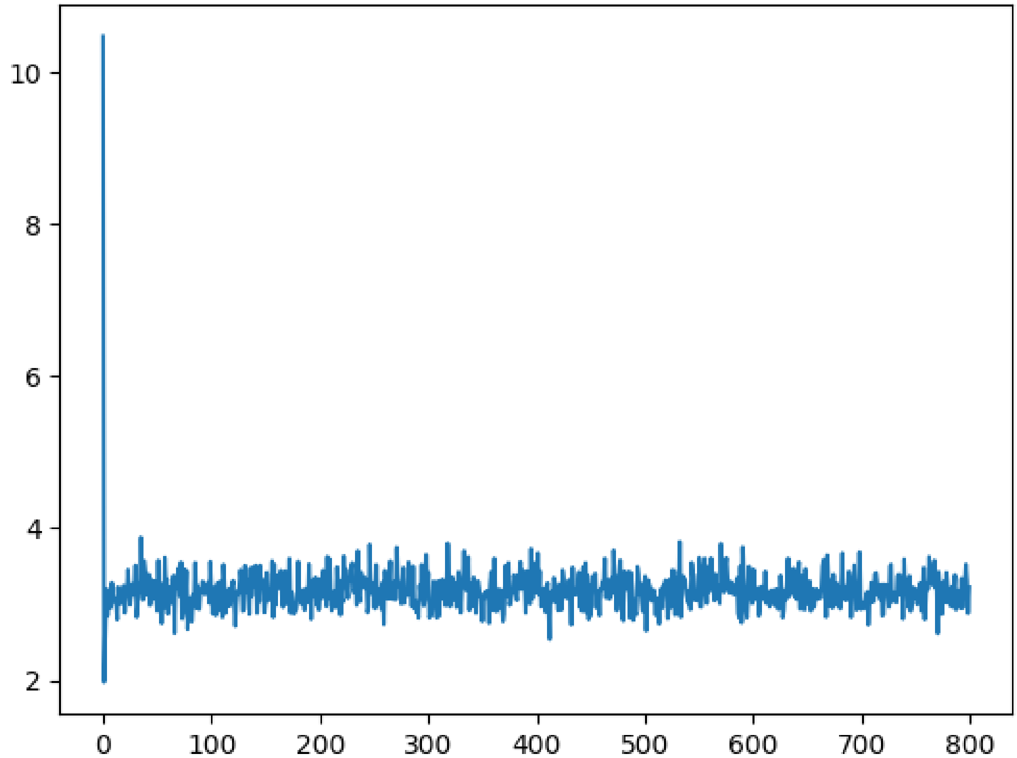
[15]:
```
sampling_result = lasso_sampler.sample(n_samples=300, n_burn=500,␣
↪lam0=100*reg_lambda, lam_update_freq=50)
```

<IPython.core.display.HTML object>

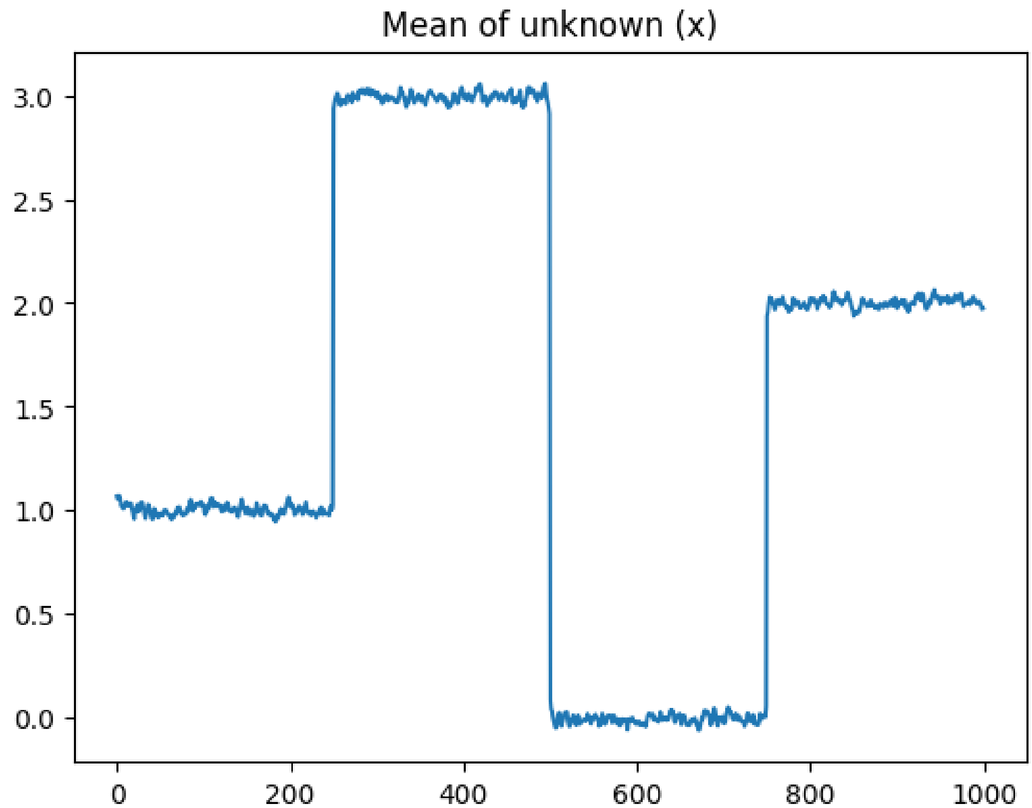
[16]:
```
sampling_result["lam_hist"]
```

[16]: array([10000.   ,   13.81413769,   25.30163257,   25.17612948,
           25.15326904,   25.16159668,   24.83322155,   25.01487996,
           25.14007003,   25.01602822,   25.17880348])

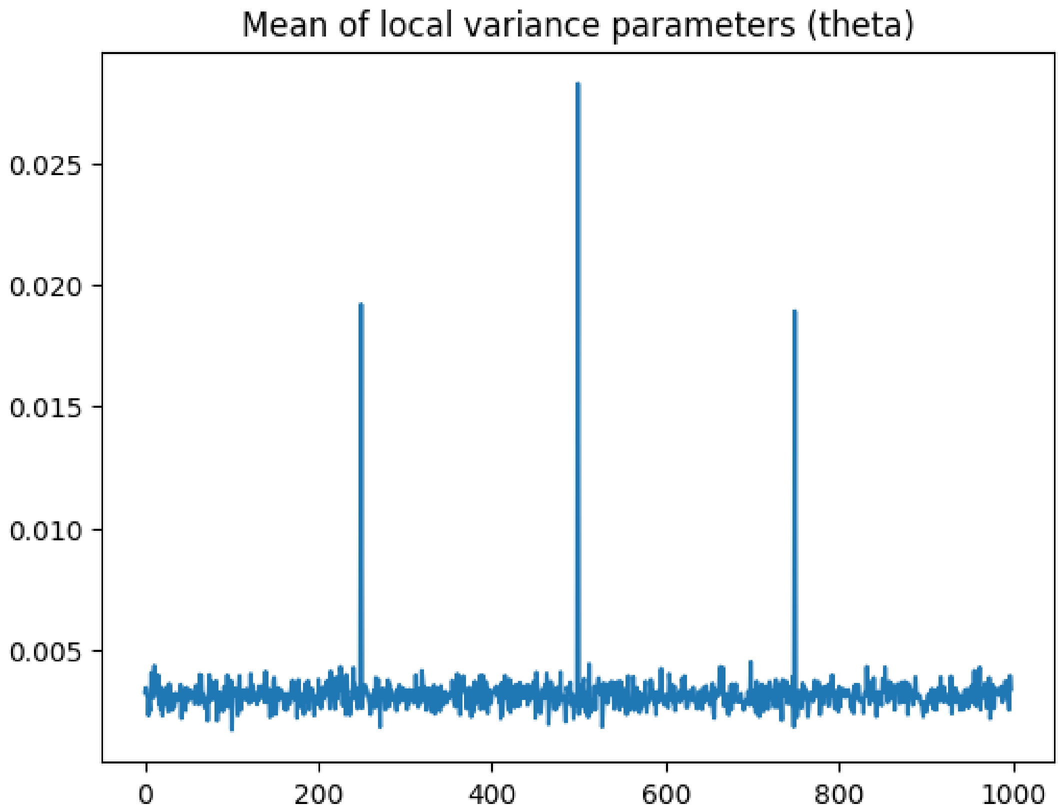
[17]:
```
plt.plot(sampling_result["theta_sums_all"])
plt.show()
```



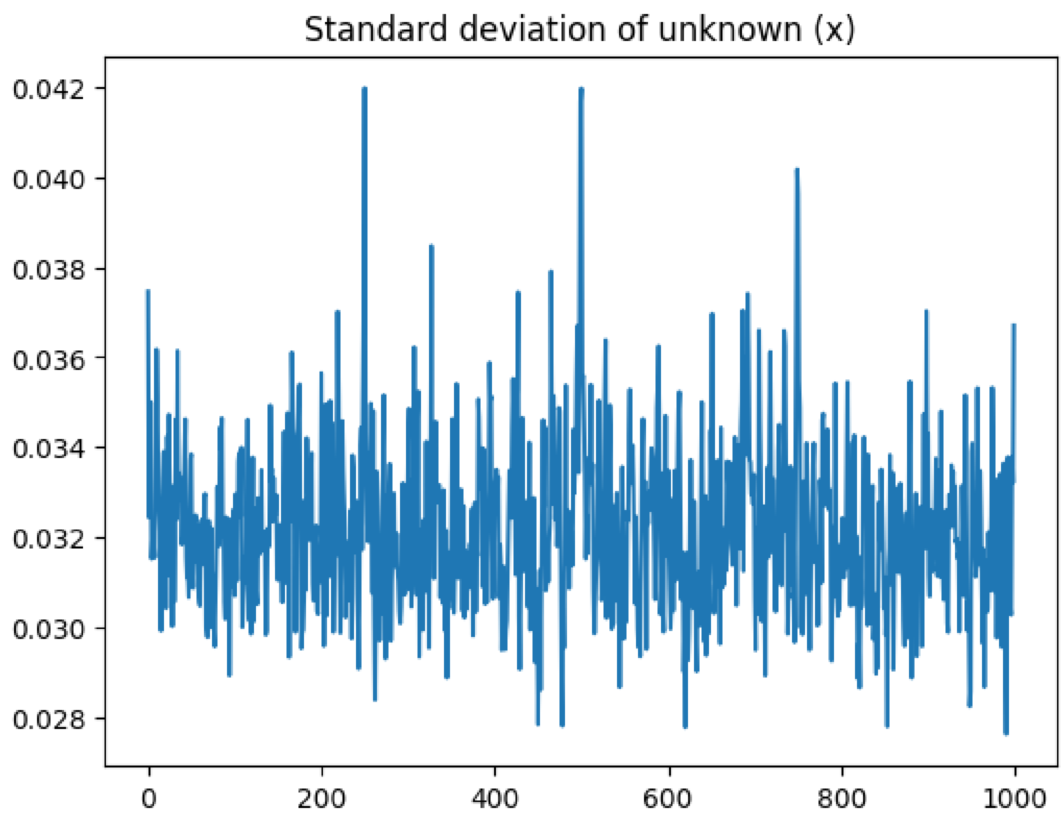
[18]:
```
plt.plot(sampling_result["x_tracker"].mean())
plt.title("Mean of unknown (x)")
plt.show()
```



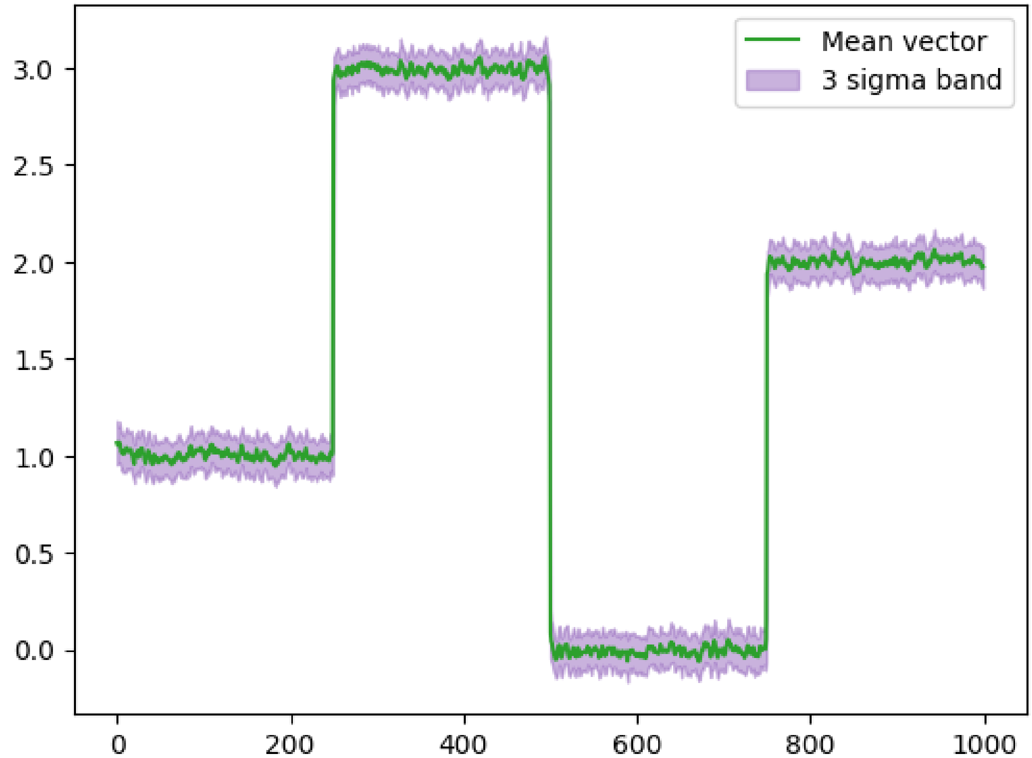
[19]:
```
plt.plot(sampling_result["theta_tracker"].mean())
plt.title("Mean of local variance parameters (theta)")
plt.show()
```



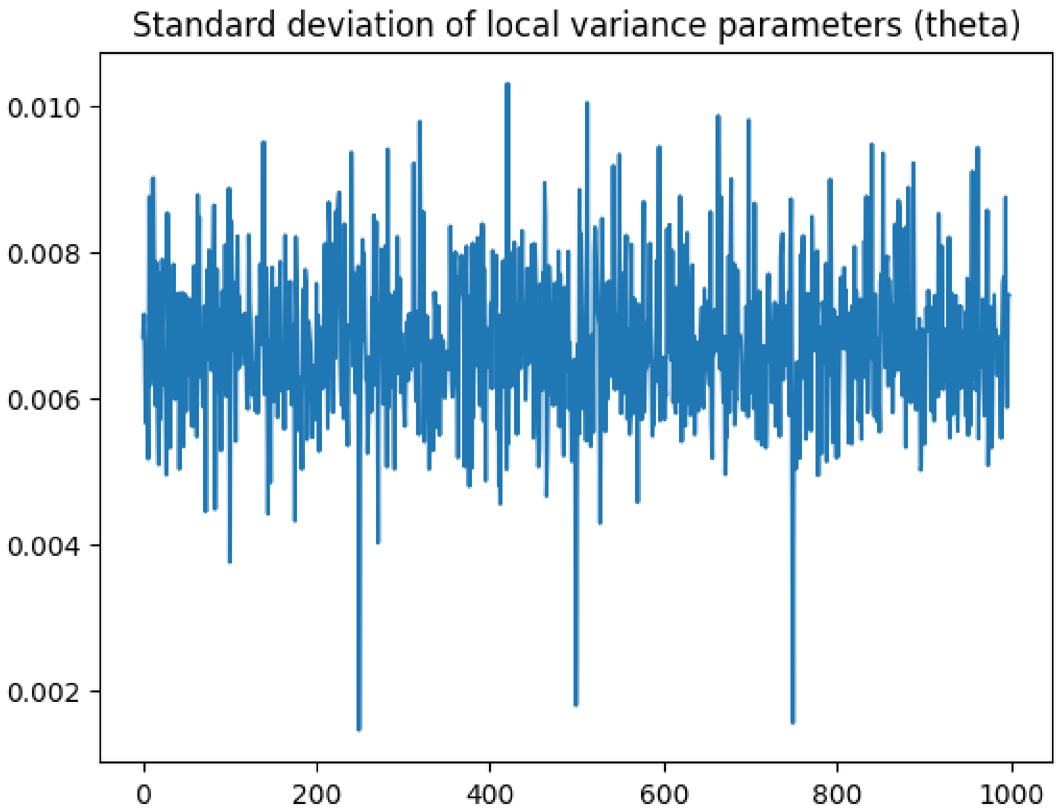
[20]:
```
plt.plot(sampling_result["x_tracker"].stdev())
plt.title("Standard deviation of unknown (x)")
plt.show()
```



[25]:
```
#plt.scatter(grid, noisy_signal, marker="x", label="Data", color="C1", alpha=1.
↪0, s=0.5)
mu, stdev = sampling_result["x_tracker"].mean(), sampling_result["x_tracker"].
↪stdev()
plt.plot(grid, mu, label="Mean vector", color="C2")
plt.fill_between(grid, mu - 3*stdev, mu + 3*stdev, color="C4", alpha=0.5,␣
↪label="3 sigma band")
plt.legend()
plt.show()
```

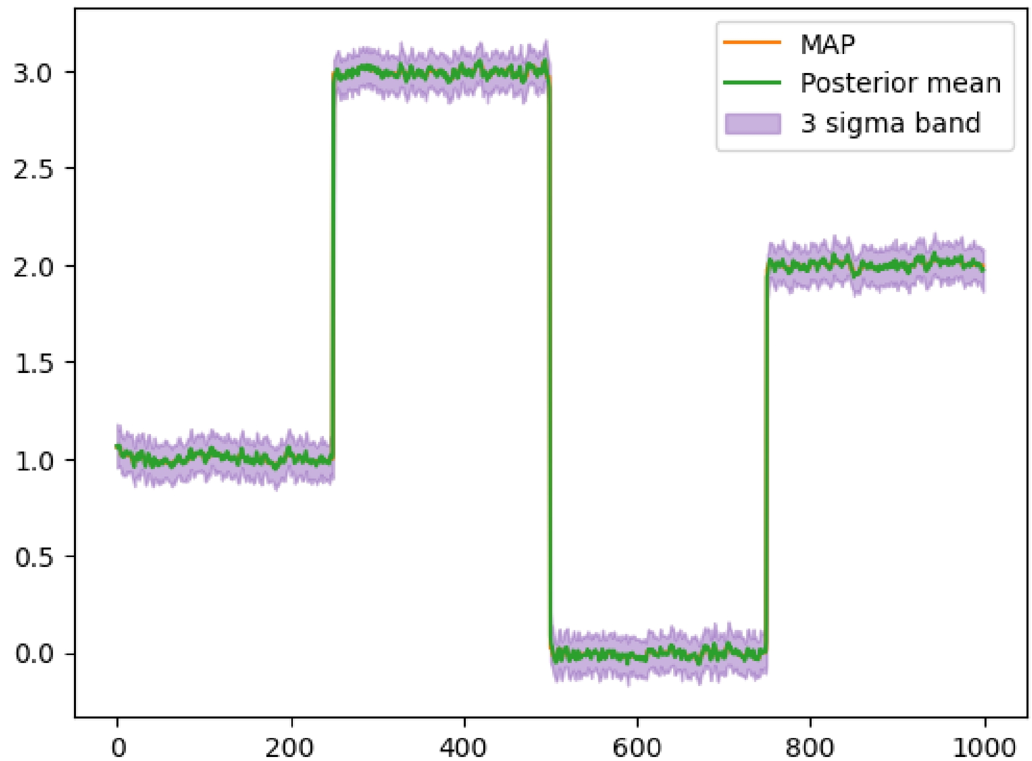


[26]:
```
plt.plot(sampling_result["theta_tracker"].stdev())
plt.title("Standard deviation of local variance parameters (theta)")
plt.show()
```



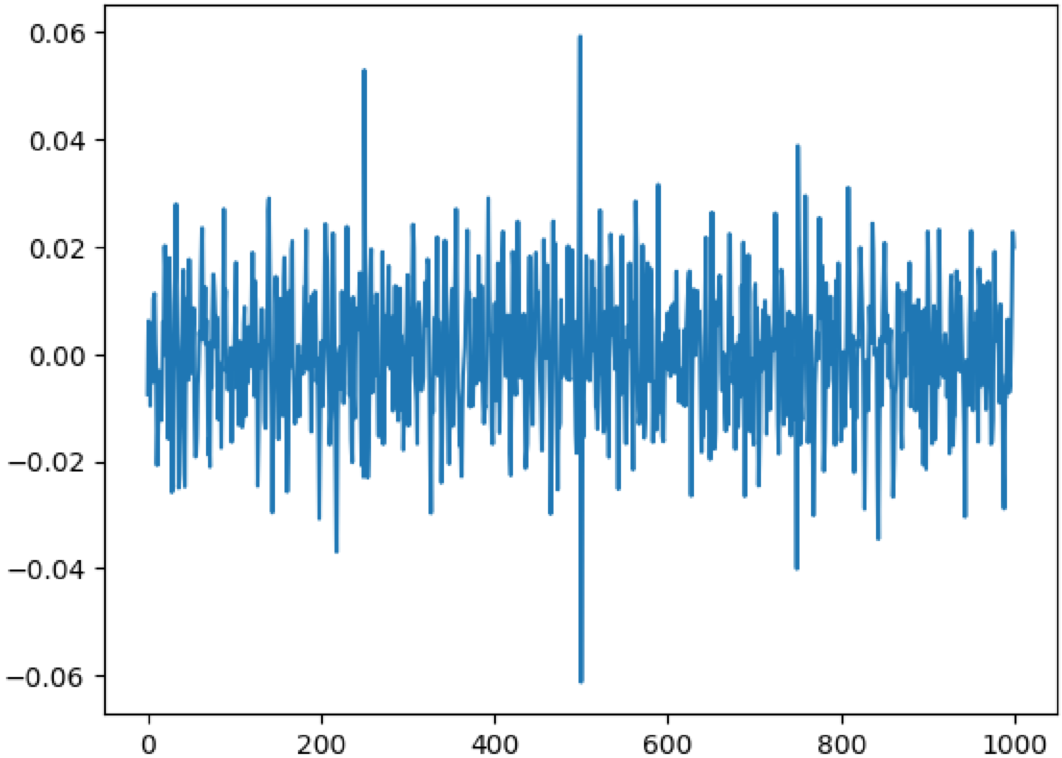
5 Comparison with other methods

[27]:
```
# Get MAP point for the lambda chosen during sampling
fdgp_map_result = jlinops.prox_tv1d_norm(noisy_signal,␣
    ↪lam=noise_var*sampling_result["lam_hist"][-1], iterations=1000)
#plt.scatter(grid, noisy_signal, marker="x", label="Data", color="C1", alpha=1.
    ↪0, s=0.5)

mu, stdev = sampling_result["x_tracker"].mean(), sampling_result["x_tracker"].
    ↪stdev()

plt.plot(grid, fdgp_map_result, label="MAP", color="C1")
plt.plot(grid, mu, label="Posterior mean", color="C2")
plt.fill_between(grid, mu - 3*stdev, mu + 3*stdev, color="C4", alpha=0.5,␣
    ↪label="3 sigma band")
plt.legend()
plt.show()
```



[28]:
```
plt.plot( fdgp_map_result - mu )
plt.plot()
```

[28]: []



0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
0
200
400
600
800
1000

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: