Appendix. Woodbury Matrix Identity

This notebook investigates various implementation for matrix inversion of the form $(P+UCV)^{-1}$. It is inspired by Bryon Yu et al's code for GPFA and this blog (https://gregorygundersen.com/blog/2018/11/30/woodbury/) post.

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

In [2]:
```
def generate_matrix(dim, dim_z):
    # where dim is the dimension of the observation. In our package, the number of nuerons o
bserved
    # dim_z is the dimension of the hidden state.
    # P: dim x dim
    # U: dim x dim_z
    # C: dim_z x dim_z
    # V: dim_z x dim
    # output to be used to test computational efficiency of various implementation of
    # Woodbury matrix identity
    # (P + UCV)^(-1) = P^(-1) - P^(-1)U(C^(-1) + VP^(-1)U)^(-1)VP^(-1)

    P = np.diag(np.random.normal(0,1,dim)) ** 2 #diag
    U = np.random.normal(0,1,(dim,dim_z))
    C = np.random.normal(0,1,(dim_z,dim_z))
    V = np.random.normal(0,1,(dim_z,dim))

    return P,U,C,V
```

In [3]:
```
def naive(P,U,C,V):
    return np.linalg.inv(P + U @ C @ V)
```

In [4]:
```
def woodbury(P, U, C, V):
    # Fast inversion of diagonal Psi.
    P_inv = np.diag(1./np.diag(P))
    C_inv = np.linalg.inv(C)
    # B is the k by k matrix to invert.
    B_inv = np.linalg.inv(C_inv + V @ P_inv @ U)
    return P_inv - P_inv @ U @ B_inv @ V @ P_inv
```

In [5]:
```
def woodbury_broadcast(P, U, C, V):
    tic = time.time()

    # Fast inversion of diagonal Psi.
    P_inv = 1./np.diag(P)
    C_inv = np.linalg.inv(C)
    # B is the k by k matrix to invert.
    B_inv = np.linalg.inv(C_inv + V @ (P_inv.reshape((-1,1)) * U))

    #return np.diag(P_inv) - P_inv.reshape((-1,1)) * U @ B_inv @ V @ np.diag(P_inv)
    return np.diag(P_inv) - P_inv.reshape((1,-1)) * (P_inv.reshape((-1,1)) * U @ B_inv @ V)
```

In [6]:
```
dim = 50
dim_z = 3
P,U,C,V = generate_matrix(dim, dim_z)
```

In [7]:
```
tic = time.time()

invM0 = naive(P, U, C, V)

toc = time.time()
print(f"naive inversion uses {toc - tic} seconds.")
```

```
naive inversion uses 0.004822015762329101 6 seconds.
```

In [8]:
```
tic = time.time()

invM = woodbury(P, U, C, V)

toc = time.time()
print(f"naive woodbury uses {toc - tic} seconds.")
```

```
naive woodbury uses 0.0004119873046875 seconds.
```

In [9]:
```
tic = time.time()

invM2 = woodbury_broadcast(P, U, C, V)

toc = time.time()
print(f"broadcast baked-in woodbury uses {toc - tic} seconds.")
```

```
broadcast baked-in woodbury uses 0.00041103363037109375 seconds.
```

In [10]:
```
# check correctness
np.max(np.abs(invM2 - invM))
```

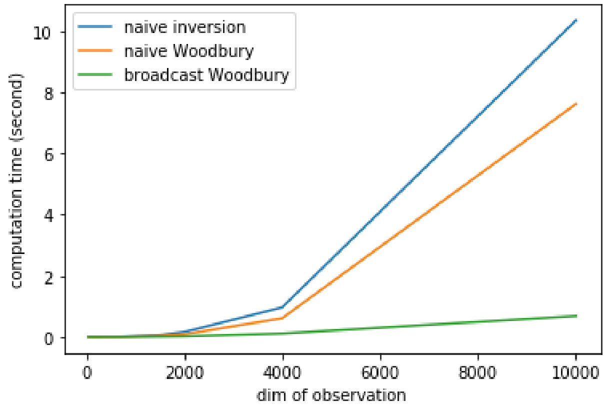
Out[10]:
```
1.6370904631912708e-11
```

scale up for many dimensions

In [18]:
```
dim_all = np.array([50,100,200,300,400,600,700,800,1500,2000,4000,10000])

t0_all = []
t1_all = []
t2_all = []
for dim in dim_all:
    # generate data
    P,U,C,V = P,U,C,V = generate_matrix(dim, 3)

    # inversion version 0
    tic = time.time()
    invM = naive(P, U, C, V)
    toc = time.time()
    t0 = toc - tic

    # inversion version 1
    tic = time.time()
    invM = woodbury(P, U, C, V)
    toc = time.time()
    t1 = toc - tic

    # inversion version 2
    tic = time.time()
    invM2 = woodbury_broadcast(P, U, C, V)
    toc = time.time()
    t2 = toc - tic

    t0_all.append(t0)
    t1_all.append(t1)
    t2_all.append(t2)

    if dim < 10000:
        assert np.max(np.abs(invM2 - invM)) < 1e-5
```

In [19]:
```
plt.plot(dim_all, t0_all, label = "naive inversion")
plt.plot(dim_all, t1_all, label = "naive Woodbury")
plt.plot(dim_all, t2_all, label = "broadcast Woodbury")
plt.legend()
plt.ylabel("computation time (second)")
plt.xlabel("dim of observation")
```

Out[19]: Text(0.5, 0, 'dim of observation')



In [ ]: