Derivations and implementation of Gaussian Process Factor Analysis (GPFA)

In this repo, we implement a commonly used Bayesian inference technique called GPFA [1 (https://journals.physiology.org/doi/full/10.1152/jn.90941.2008)].

The repo can be walked through in 4 notebooks:

1. readme: this notebook. It details the derivations and explains numerical implementation variants.
2. makeData. It generates simulated data for testing.
3. vanillaGPFA. It runs our implementation.
4. appendix. It compares various implementations of the matrix inversion of the form used in this package.

One way to study the brain is to directly measure the neuronal spiking activities. Spikes or also known as action potentials are binary sigals that neurons use to communicate with one another. Using a few electrodes, we can record hundreds or thousands of neurons spiking activities over seconds or minutes when the subject completes a task (for example, a decision making task or arm reach movements). If we put these data into a matrix of Y, of size number of neurons q by time T (T time bins, each 2ms long for example), we see that we would end up with a large matrix with very fine time resolution. Neuroscientists usually attempt to understand this by smoothing the data and by dimensionality reduction techniques such as PCA.

Instead of seperating data smoothing and data dimensionality reduction into 2 steps, GPFA proposes a hierarchical model where the state variable is smooth - modeled by Gaussian process - and is related to observed noisy spikes through a linear regression. Every parameter is learned with Expectation Maximization method.

1. Variables and the model

- Spiking data (**the observations**).

  $$Y \in \mathbb{R}^{q \times T}$$

  where q is the number of neurons and T is the number of time bins. $y_{:,t}$ is the t-th column of Y.
- Dimension reduced neuron states or neural trajectory (**The nuissance parameter of the model**).

  $$Z \in \mathbb{R}^{p \times T}$$

  where p is the number of dimensions of and T is the number of time bins. We use Z here instead of X as in the source paper because with Z the notation of EM will then match Bishop and other Bayesian textbooks. $z_{:,t}$ is the t-th column of Z.
- At each time t, observations and neural states are related by linear-Gaussian:

  $$y_{:,t}|z_{:,t} \sim \mathrm{N}(Cz_{:,t} + d, R)$$

  where $C \in \mathbb{R}^{q \times p}, d \in \mathbb{R}^{q \times 1}, R \in \mathbb{R}^{q \times q}$. GPFA further constrains the covariance R to be diagonal, modeling the independent noise level for each neuron. In essence, the model, like factor analysis, tries to represent the independent variance associated with each coordinate in the matrix R and capturing the covariance between neurons in the matrix C.
- For each row of neural states, $z_{i,:}$ across time are related through Gaussian process (**how how the model structure shares information between observations**). $z_{i,:} \sim (\mathbf{0}, K_i)$ where $K_i(t_1, t_2) = \sigma_{f,i}^2 \exp(-\frac{(t_1-t_2)^2}{2\tau_i^2}) + \delta_{t_1,t_2}\sigma_{n,i}$, where the Kronecker delta is 1 if $t_1 = t_2$ and 0 otherwise. We set $\sigma_{n,i}$ to be 10^{-3}. The $\sigma_{n,i}$ helps stablizes matrix K.

2. Model priors and parameter estimation

The prior for $x_{:,t}$ is $\mathrm{N}(\mathbf{0}, I)$. In other words, each neuronal state is uncorrelated with each other. This necessitates that we set $K_i(t,t) = 1$, which is $\sigma^2_{f,i} = 1 - \sigma^2_{n,i}$
**We note that across time neuronal states are correlated through the Gaussian process kernel. This is how how the model structure shares information between spike count observations over time. **

Parameters are estimated via EM algorithm. In our case, we observe $Y = y$, $Y \sim P_{Y|\Theta}$, $\Theta \sim P_\Theta$, where $\Theta = [X, Z], X = [C, d, R, \tau]$. We want MAP estimator for X marginalized over the nuisance parameter neural trajectory Z.

Much of the calculations are quite similar to those in Factor Analysis. See Bishop (https://www.microsoft.com/en-us/research/wp-content/uploads/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf) and this post (https://gregorygundersen.com/blog/2018/08/08/factor-analysis/). Overall structure of updates all match up to R and K.

Let x^t be the current estimate of parameters.

E-step: calculate $Q(x|x^{(t)}) = \mathbb{E}_{z\sim p(z|x^{(t)},y)}[\log p(x|z,y)]$

In this step, the main workload is in calculating the conditional probability of neural trajectory given current estimate of parameters and observed data. Due to the use of Gaussian process and linear-normal distribution,

$$z_{i,:} \sim \mathrm{N}(\mathbf{0}, K_i)$$
$$y_{:,t}|z_{:,t} \sim \mathrm{N}(Cz_{:,t} + d, R)$$

Concatenating all columns of Z into one long column and doing the same for Y, we see that $\bar{Z}$ and $\bar{Y}$ are jointly Gaussian. Details can be found in Equation A1-A4 in the source paper.

Using the basic result of conditioning for jointly Gaussian random variables, we have

$$\bar{Z}|\bar{Y} \sim \mathrm{N}(\bar{K}\bar{C}^T(\bar{C}\bar{K}\bar{C}^T + \bar{R})^{-1}(\bar{y} - \bar{d}), \bar{K} - \bar{K}\bar{C}^T(\bar{C}\bar{K}\bar{C}^T + \bar{R})^{-1}\bar{C}\bar{K})$$

where we implicitly condition on the current estimate of parameters.

We then use the joint distribution to calculate Q.

$$Q(x|x^{(t)}) \propto \mathbb{E}_{\bar{Z}\sim p(\bar{Z}|x^{(t)},\bar{Y})}[\log p(X, \bar{Y}, \bar{Z})]$$

$$= -\frac{1}{2}\begin{bmatrix}\mathbb{E}(\bar{Z})\\ \bar{y} - \bar{d}\end{bmatrix}^T M^{-1}\begin{bmatrix}\mathbb{E}(\bar{Z})\\ \bar{y} - \bar{d}\end{bmatrix} - \frac{1}{2}\log(\det(K)) - \frac{1}{2}\log(\det(R)) = \square$$

$$\text{where } M = \begin{bmatrix}\bar{K} & \bar{K}\bar{C}^T\\ \bar{C}\bar{K} & \bar{C}\bar{K}\bar{C}^T + \bar{R}\end{bmatrix}.$$

To simplify, we first use the following identity to expand out M inverse.

$$\begin{bmatrix}A & B\\ C & D\end{bmatrix}^{-1} = \begin{bmatrix}A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1}\\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1}\end{bmatrix}$$

$$M^{-1} = \begin{bmatrix} \bar{K}^{-1} + \bar{C}'\bar{R}^{-1}\bar{C} & -\bar{C}'\bar{R}^{-1} \\ -\bar{R}^{-1}\bar{C}' & \bar{R}^{-1} \end{bmatrix}$$

Therefore, $Q(x|x^{(t)}) = -\frac{1}{2}(\mathbb{E}(\bar{Z})^T\bar{K}^{-1}\mathbb{E}(\bar{Z}) + \mathbb{E}(\bar{Z})^T\bar{C}^T\bar{R}^{-1}\bar{C}\mathbb{E}(\bar{Z}) - 2\mathbb{E}(\bar{Z})^T\bar{C}^T\bar{R}^{-1}\bar{Y} + 2\mathbb{E}(\bar{Z})^T\bar{C}^T\bar{R}^{-1}\bar{d} + \bar{Y}^T\bar{R}^{-1}\bar{Y} - 2\bar{Y}^T\bar{R}^{-1}\bar{d} + \bar{d}^T\bar{R}^{-1}\bar{d}) - \frac{1}{2}\log(\det(K)) - \frac{1}{2}\log(\det(R))$

M-step:

Maximizing Q with respect to C and d:

Restricting to the terms in Q with C and d,
$Q(x|x^{(t)}) = \mathbb{E}(\bar{Z})^T\bar{C}^T\bar{R}^{-1}\bar{C}\mathbb{E}(\bar{Z}) - 2\mathbb{E}(\bar{Z})^T\bar{C}^T\bar{R}^{-1}\bar{Y} + 2\mathbb{E}(\bar{Z})^T\bar{C}^T\bar{R}^{-1}\bar{d} - 2\bar{Y}^T\bar{R}^{-1}\bar{d} + \bar{d}^T\bar{R}^{-1}\bar{d}$

$$\begin{bmatrix} \mathbb{E}\bar{Z}^T & 1 \end{bmatrix} \begin{bmatrix} \bar{c}^T \\ \bar{d}^T \end{bmatrix} \bar{R}^{-1} \begin{bmatrix} \bar{c} & \bar{d} \end{bmatrix} \begin{bmatrix} \mathbb{E}\bar{Z} \\ 1 \end{bmatrix} - \begin{bmatrix} \mathbb{E}\bar{Z}^T & 1 \end{bmatrix} \begin{bmatrix} \bar{c}^T \\ \bar{d}^T \end{bmatrix} \begin{bmatrix} \bar{R}^{-1}\bar{Y} & \bar{R}^{-1}\bar{Y} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Let $[\bar{C}\bar{d}] = A$, Taking the derivative of the first term with respect to A:

$$\frac{d\mathrm{tr}(\begin{bmatrix} \mathbb{E}\bar{Z}^T & 1 \end{bmatrix} A^T\bar{R}^{-1}A \begin{bmatrix} \mathbb{E}\bar{Z} \\ 1 \end{bmatrix})}{dA} = \frac{d\mathrm{tr}(A^T\bar{R}^{-1}A \begin{bmatrix} \mathbb{E}\bar{Z}\mathbb{E}\bar{Z}^T & \mathbb{E}\bar{Z} \\ \mathbb{E}\bar{Z}^T & 1 \end{bmatrix})}{dA}$$

Taking the derivative of the second term with respect to A, we get $\frac{d\mathrm{tr}(A^T\begin{bmatrix} \bar{R}^{-1}\bar{Y} & \bar{R}^{-1}\bar{Y} \end{bmatrix} \begin{bmatrix} \mathbb{E}\bar{Z}^T & 1 \\ \mathbb{E}\bar{Z}^T & 1 \end{bmatrix})}{dA}$

Using the [trace property (https://web.stanford.edu/~jduchi/projects/matrix_prop.pdf)]:
$\nabla_A \mathrm{tr}ABA^TC = CAB + C^TAB^T$

$$\frac{dQ}{dA^T} = 2 \begin{bmatrix} \mathbb{E}\bar{Z}\mathbb{E}\bar{Z}^T & \mathbb{E}\bar{Z} \\ \mathbb{E}\bar{Z}^T & 1 \end{bmatrix} A^T\bar{R}^{-1} - 2 \begin{bmatrix} \mathbb{E}\bar{Z}^T & 1 \end{bmatrix} \bar{R}^{-1}Y$$

Setting the derivative to zero yields: $A \begin{bmatrix} \mathbb{E}\bar{Z}\mathbb{E}\bar{Z}^T & \mathbb{E}\bar{Z} \\ \mathbb{E}\bar{Z}^T & 1 \end{bmatrix} = \bar{Y} \begin{bmatrix} \mathbb{E}\bar{Z}^T & 1 \end{bmatrix}$

$$A = \bar{Y} \begin{bmatrix} \mathbb{E}\bar{Z}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbb{E}\bar{Z}\mathbb{E}\bar{Z}^T & \mathbb{E}\bar{Z} \\ \mathbb{E}\bar{Z}^T & 1 \end{bmatrix}^{-1}$$

which can then be derived in nonconcatenated form as in Equation A8 in the source paper.

Maximizing Q with respect to R:

$$Q(x|x^{(t)}) = -\frac{1}{2}(\mathbb{E}(\bar{Z})^T \bar{K}^{-1}\, \mathbb{E}(\bar{Z}) + \mathbb{E}(\bar{Z})^T \bar{C}^T \bar{R}^{-1}\, \bar{C}\mathbb{E}(\bar{Z}) - 2\mathbb{E}(\bar{Z})^T \bar{C}^T \bar{R}^{-1}\, \bar{Y} + 2\mathbb{E}(\bar{Z})^T \bar{C}^T \bar{R}^{-1}\, \bar{d}$$
$$+ \bar{Y}^T \bar{R}^{-1}\, \bar{Y} - 2\bar{Y}^T \bar{R}^{-1}\, \bar{d} + \bar{d}^T \bar{R}^{-1}\, \bar{d}) - \frac{1}{2}\log(\det(K)) - \frac{1}{2}\log(\det(R)) = \square$$

Collecting the terms in Q with $\bar{R}$,

$$Q(x|x^{(t)}) = -\frac{1}{2}(\mathbb{E}(\bar{Z})^T \bar{C}^T \bar{R}^{-1}\, \bar{C}\mathbb{E}(\bar{Z}) - 2\mathbb{E}(\bar{Z})^T \bar{C}^T \bar{R}^{-1}\, \bar{Y} + 2\mathbb{E}(\bar{Z})^T \bar{C}^T \bar{R}^{-1}\, \bar{d} + \bar{Y}^T \bar{R}^{-1}\, \bar{Y}$$
$$- 2\bar{Y}^T \bar{R}^{-1}\, \bar{d} + \bar{d}^T \bar{R}^{-1}\, \bar{d}) - \frac{1}{2}\log(\det(R))$$

Adding trace() operator on each term and moving subterms, we get

$$\frac{d\square}{dR^{-1}} = -\frac{1}{2}(\bar{C}\mathbb{E}(\bar{Z})\mathbb{E}(\bar{Z})^T \bar{C}^T + (\bar{Y} - \bar{d})(\bar{Y} - \bar{d})^T - 2(\bar{Y} - \bar{d})\mathbb{E}(\bar{Z})^T \bar{C}^T) + \frac{1}{2}\bar{R}$$

where we use the fact that $\frac{\log(det(X))}{X} = (X^{-1})^T$

Because $\bar{C}\bar{Z} + \bar{d} = \mathbb{E}(\bar{Y}|\bar{Z})$, with another expectation over Z, we have:

$$\bar{C}\mathbb{E}(\bar{Z}) = \mathbb{E}(\bar{Y}) - \bar{d} = \bar{Y} - \bar{d}$$

Therefore, $\frac{d\square}{dR^{-1}} = -\frac{1}{2}((\bar{Y} - \bar{d})(\bar{Y} - \bar{d})^T - (\bar{Y} - \bar{d})\mathbb{E}(\bar{Z})^T \bar{C}^T) + \frac{1}{2}\bar{R}$

This gives the update in Equation (A9) for R.

Maximizing Q with respect to $\tau's$:

Due to the interleaving of τ inside and outside of exponential operator, there is no close form solution. But the gradient is computatable and can be used with any gradient optimization technique.

Collecting the terms in Q with $\bar{K}$, $Q(x|x^{(t)}) = -\text{tr}(\mathbb{E}(\bar{Z})^T \bar{K}^{-1}\, \mathbb{E}(\bar{Z})) - \frac{1}{2}\log(\det(K))$

By chain rule,

$$\frac{dQ}{d\tau_i} = \text{tr}([\frac{dQ}{dK_i}]' \frac{dK_i}{d\tau_i})$$

where $\frac{dQ}{dK} = \frac{1}{2}(-K^{-1} + K^{-1}\mathbb{E}(\bar{Z})\mathbb{E}(\bar{Z})^T K^{-1})$

(We use the fact of 2.5 in Matrix cookbook (http://www.ee.ic.ac.uk/hp/staff/dmb/matrix/proof002.html#dYinv_dx_p)
$\partial(\text{tr}(X^{-1}) = \text{tr}(\partial(X^{-1})) = \text{tr}(-X^{-1}(\partial X)X^{-1}) = -\text{tr}(X^{-1}(\partial X)X^{-1})$**)**

Restricting to each K_i**, we have**

$\frac{dQ}{dK_i} = \frac{1}{2}(-K_i^{-1} + K_i^{-1}\mathbb{E}(Z_{i,:})^T\mathbb{E}(Z_{i,:})K_i^{-1})$ **(Note that subindices of** Z **and direction changes accordingly.)**

$\frac{dK_i(t_1,t_2)}{d\tau_i} = \sigma_{f,i}^2 \frac{(t_1-t_2)^2}{\tau_i^3}\exp(-\frac{(t_1-t_2)^2}{2\tau_i^2})$

Calculating the expectations $\mathbb{E}(Z_{i,:}), \mathbb{E}(Z_{i,:})(Z_{i,:})^T$

In this section, due to the many C^T**'s, we use a simpler notation** C' **to denote** C^T**.**

Because these expectations show up in many M-step updates, we simplify these terms from the forms in Equation A5, reproduced above and copied again below.

$\bar{Z}|\bar{Y} \sim \text{N}(\bar{K}\bar{C}'(\bar{C}\bar{K}\bar{C}' + \bar{R})^{-1}(\bar{y} - \bar{d}), \bar{K} - \bar{K}\bar{C}'(\bar{C}\bar{K}\bar{C}' + \bar{R})^{-1}\bar{C}\bar{K})$

We reduce computational complexity of $(CKC' + R)^{-1}$ **via Woodbury matrix identity. (Eqn 157 in Matrix Cookbook (https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf)).**

$(CKC' + R)^{-1} = R^{-1} - R^{-1}C(K^{-1} + C'R^{-1}C)^{-1}CR^{-1}$

We can go from an O(#(dimension of observation)^3) operation to an O(#(dimension of state)^3) operation. More implementational details are in the Appendix.

Denote $(\bar{K}^{-1} + \bar{C}'\bar{R}^{-1}\bar{C})^{-1}$ **by** M^{-1}

$$\mathbb{E}(Z_{i,:}) = \bar{K}\bar{C}'(\bar{C}\bar{K}\bar{C}' + \bar{R})^{-1}(\bar{y} - \bar{d}) = \bar{K}(I - \bar{C}'\bar{R}^{-1}\bar{C}M^{-1})\bar{C}\bar{R}^{-1}(\bar{y} - \bar{d})$$

For the covariance term:

We note that applying the Woodbury matrix identity to $M^{-1} = (\bar{K}^{-1} + \bar{C}'\bar{R}^{-1}\bar{C})^{-1}$ **again yields**
$M^{-1} = (\bar{K}^{-1} + \bar{C}'\bar{R}^{-1}\bar{C})^{-1} = \bar{K} - \bar{K}\bar{C}'(\bar{R} + \bar{C}\bar{K}\bar{C}')\bar{C}'\bar{K}$

So the covariance is: $\bar{K} - \bar{K}\bar{C}'(\bar{C}\bar{K}\bar{C}' + \bar{R})^{-1}\bar{C}\bar{K} = M^{-1}$

$$\mathbb{E}(Z_{i,:})(Z_{i,:})^T = M^{-1} + \mathbb{E}(Z_{i,:})\mathbb{E}(Z_{i,:})^T$$

where we use the fact that for a random variable $Y, Cov(Y) = \mathbb{E}YY^T - \mathbb{E}Y\mathbb{E}Y^T$

Calculating the data likelihood

The EM algorithm optimizes the data likelihood, which, due to $\bar{Y} \sim (\bar{d}, \bar{C}\bar{K}\bar{C}^T + \bar{R})$ would have

$$\log \text{likelihood} = -\frac{1}{2}((Y-d)^T(CKC^T+R)^{-1}(Y-d)) - \frac{1}{2}\text{logdet}(CKC^T+R)$$

where the notations are simplified -- all variables should have a bar over them.

Using the same Woodbury trick to reduce the size of the matrix to be inverted, the first term

$$(CKC^T+R)^{-1} = R^{-1} - R^{-1}CM^{-1}CR^{-1}$$

and the second term

$$-\log(\det(CKC^T+R)) = \log(\det(CKC^T+R)^{-1}) = \log(\det(R^{-1} - R^{-1}CM^{-1}CR^{-1})$$
$$= -\log(\det(R)) + \log(\det(I - C^TM^{-1}CR^{-1})) + \log(\det(K)) - \log(\det(K)) = -\log(\det(R)) - \log(\det(M)) - \log(\det(K))$$

In above, we note that

$$\log(\det(K - KC^TM^{-1}CR^{-1})) = \log(\det(M))$$

using Woodbury and the identity $\det(I+A) = 1 + \det(A) + \text{Tr}(A)$**.**

3. Generating simulated dataset

In Notebook 1, we generate a similated dataset knowing the ground-truth latent states. We will compare our inferred neuronal state with the this ground truth.

4. Implementation

In Notebook 2 and the associated scripts, we implement the E and M steps from scratch. All the steps are derived above. In addition, we use numpy broadcasting as much as possible since it can lead to significant computation savings, see Appendix for details. We used some library functions for calculating determinant and inverting block per-symmetric matrices.

5. Post processing: orthonormalization of C matrix

Recall that matrix C is a mapping from latent space to neural space:

$$\mathbf{y}_{:,t}|\mathbf{x}_{:,t} \sim \mathbf{N}(\mathbf{C}\mathbf{x}_{:,t} + \mathbf{d}, \mathbf{R})$$

As noted in the manuscript, the columns of C provide an linear combination for how much each latent dimension of x should contribute a particular dimension of y. To improve the interpretability of this weighting, it is helpful to orthonormalize the columns of C, so that they can be considered an orthogonal basis for the mapping. We can then weight each dimension of x against the corresponding basis vectors of C to achieve a particular dimension of y. The orthonormalization procedure involves applying singular value decomposition to the learned $\mathbf{C}$. This yields:

$$\mathbf{C} = \mathbf{U}\mathbf{D}\mathbf{V}'$$

where $\mathbf{U} \in \mathbb{R}^{q \times p}$ and $\mathbf{V} \in \mathbb{R}^{p \times p}$ each have orthonormal columns and $\mathbf{D} \in \mathbb{R}^{p \times p}$ is diagonal. Thus, we can write:

$$\mathbf{C}\mathbf{x}_{:,t} = \mathbf{U}(\mathbf{D}\mathbf{V}'\mathbf{x}_{:,t}) = \mathbf{U}\tilde{\mathbf{x}_{:,t}}$$

where $\tilde{\mathbf{x}_{:,t}} = \mathbf{D}\mathbf{V}'\mathbf{x}_{:,t} \in \mathbb{R}^{p \times 1}$ is the orthonormalized neural state at time point t.

Note that $\tilde{\mathbf{x}_{:,t}}$ is a linear transformation of $\mathbf{x}_{:,t}$. The orthonormalized neural trajectory extracted from the observed activity Y is thus:

$$\mathbf{D}\mathbf{V}'\mathbb{E}[\mathbf{X}|\mathbf{Y}]$$

Since $\mathbf{U}$ has orthonormal columns, we can now intuitively visualize the trajectories extracted by GPFA, in much the same spirit as for PCA.

Succinctly, we use the SVD of C to linearly transform x into an orthonormalized neural state, then can use U matrix of the SVD as the orthonormal basis for mapping into the higher dimensional, observed neural activity y. This decomposition helps with post-processing and visualization.

6. Discussions and Future Studies.

In this repo, we implemented a basic version of Gaussian Process Factor Analysis on a sample dataset.

Moving forward, it would be exciting to use Gaussian Process Factor Analysis (GPFA) to analyze if the latent, low dimensional space on neural spike rasters could be used to infer behavioral or physiological state. An example of this would be to see if, by using spike count's of hippocampal CA1 neurons, latent state representation could be used to determine a rodent's location on a track.

Another interesting study would be to use GPFA on power spectral density vectors generated from cortical field potentials to assess GPFA's capabilities in identifying low-dimensional representations of oscillatory dynamics. This could be useful for implementing into brain-machine interfaces, where a low-dimensional represenation of the oscillatory dynamics may be utilized by control systems for modulating neural activity. For example, in closed-loop Deep Brain Stimulation for Parkinson's Disease, it is often useful to identify states of low gamma power and high beta power, as these states often correspond to bothersome movement symptoms. GPFA could identify latent states that correspond to bothersome symptom states from high-dimensional Power-Spectral Density vectors, and consequently, stimulation control policies for targeting movement symptoms could operate on the inferred latent states.

One could also compare how model parameters (e.g. timescale initialization) affect the latent state representation of oscillatory dynamics versus neural spike raster. It's plausible that differing kernel functions would be needed for adequately representing these separate use-cases using GPFA.

7. Appendix

In the appendix, we compare various implementations of the inversions used in the E-step.

In [ ]:

1. Generate synthetic dataset

In [1]:
```
import numpy as np
from scipy.integrate import odeint
import quantities as pq
import neo
from elephant.spike_train_generation import inhomogeneous_poisson_process
```

In [2]:
```
from elephant.gpfa import GPFA
```

The following code is taken from this tutorial (https://elephant.readthedocs.io/en/latest/tutorials/gpfa.html) as validation of our implementation.

In [3]:
```
def integrated_oscillator(dt, num_steps, x0=0, y0=1, angular_frequency=2*np.pi*1e-3):
    """
    Parameters
    ----------
    dt : float
        Integration time step in ms.
    num_steps : int
        Number of integration steps -> max_time = dt*(num_steps-1).
    x0, y0 : float
        Initial values in three dimensional space.
    angular_frequency : float
        Angular frequency in 1/ms.

    Returns
    -------
    t : (num_steps) np.ndarray
        Array of timepoints
    (2, num_steps) np.ndarray
        Integrated two-dimensional trajectory (x, y, z) of the harmonic oscillator
    """

    assert isinstance(num_steps, int), "num_steps has to be integer"
    t = dt*np.arange(num_steps)
    x = x0*np.cos(angular_frequency*t) + y0*np.sin(angular_frequency*t)
    y = -x0*np.sin(angular_frequency*t) + y0*np.cos(angular_frequency*t)
    return t, np.array((x, y))


def random_projection(data, embedding_dimension, loc=0, scale=None):
    """
    Parameters
    ----------
    data : np.ndarray
        Data to embed, shape=(M, N)
    embedding_dimension : int
        Embedding dimension, dimensionality of the space to project to.
    loc : float or array_like of floats
        Mean (“centre”) of the distribution.
    scale : float or array_like of floats
        Standard deviation (spread or “width”) of the distribution.

    Returns
    -------
    np.ndarray
       Random (normal) projection of input data, shape=(dim, N)

    See Also
    --------
    np.random.normal()

    """
    if scale is None:
        scale = 1 / np.sqrt(data.shape[0])
    projection_matrix = np.random.normal(loc, scale, (embedding_dimension, data.shape[0]))
    return np.dot(projection_matrix, data)


def generate_spiketrains(instantaneous_rates, num_trials, timestep):
    """
    Parameters
    ----------
    instantaneous_rates : np.ndarray
        Array containing time series.
    timestep :
        Sample period.
    num_steps : int
        Number of timesteps -> max_time = timestep*(num_steps-1).

    Returns
    -------
    spiketrains : list of neo.SpikeTrains
        List containing spiketrains of inhomogeneous Poisson
        processes based on given instantaneous rates.

    """

    spiketrains = []
    for _ in range(num_trials):
        spiketrains_per_trial = []
        for inst_rate in instantaneous_rates:
            anasig_inst_rate = neo.AnalogSignal(inst_rate, sampling_rate=1/timestep, units=pq.Hz)
            spiketrains_per_trial.append(inhomogeneous_poisson_process(anasig_inst_rate))
        spiketrains.append(spiketrains_per_trial)

    return spiketrains
```

In [4]:
```
# set parameters for the integration of the harmonic oscillator
timestep = 1 * pq.ms
trial_duration = 2 * pq.s
num_steps = int((trial_duration.rescale('ms')/timestep).magnitude)

# set parameters for spike train generation
max_rate = 70 * pq.Hz
np.random.seed(42)  # for visualization purposes, we want to get identical spike trains at any run

# specify data size
num_trials = 20
num_spiketrains = 50

# generate a low-dimensional trajectory
times_oscillator, oscillator_trajectory_2dim = integrated_oscillator(
    timestep.magnitude, num_steps=num_steps, x0=0, y0=1)
times_oscillator = (times_oscillator*timestep.units).rescale('s')

# random projection to high-dimensional space
oscillator_trajectory_Ndim = random_projection(
    oscillator_trajectory_2dim, embedding_dimension=num_spiketrains)

# convert to instantaneous rate for Poisson process
normed_traj = oscillator_trajectory_Ndim / oscillator_trajectory_Ndim.max()
instantaneous_rates_oscillator = np.power(max_rate.magnitude, normed_traj)

# generate spike trains
spiketrains_oscillator = generate_spiketrains(
    instantaneous_rates_oscillator, num_trials, timestep)
```

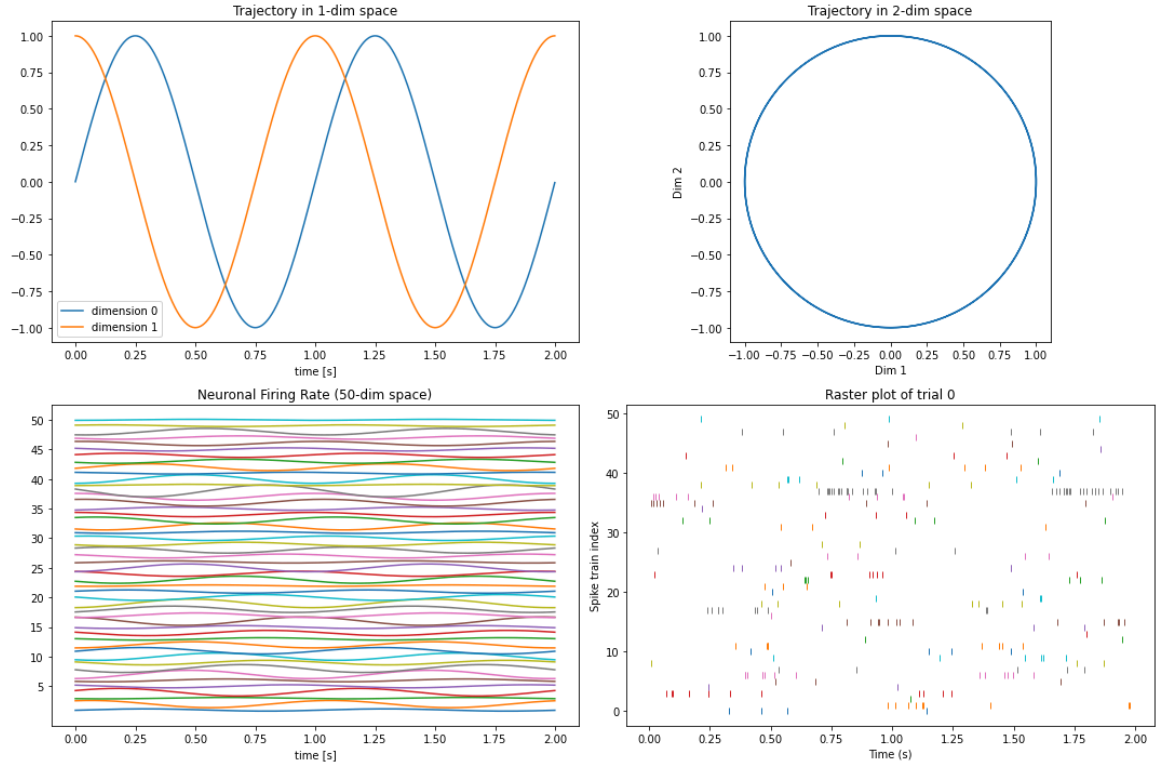
In [5]:
```
import matplotlib.pyplot as plt


f, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(15, 10))

ax1.set_title('Trajectory in 1-dim space')
ax1.set_xlabel('time [s]')
for i, y in enumerate(oscillator_trajectory_2dim):
    ax1.plot(times_oscillator, y, label=f'dimension {i}')
ax1.legend()

ax2.set_title('Trajectory in 2-dim space')
ax2.set_xlabel('Dim 1')
ax2.set_ylabel('Dim 2')
ax2.set_aspect(1)
ax2.plot(oscillator_trajectory_2dim[0], oscillator_trajectory_2dim[1])

ax3.set_title(f'Neuronal Firing Rate ({num_spiketrains}-dim space)')
ax3.set_xlabel('time [s]')
y_offset = oscillator_trajectory_Ndim.std() * 3
for i, y in enumerate(oscillator_trajectory_Ndim):
    ax3.plot(times_oscillator, y + i*y_offset)

yticks = np.arange(len(oscillator_trajectory_Ndim)) * oscillator_trajectory_Ndim.std() * 3
yticklabels = np.arange(len(oscillator_trajectory_Ndim)) + 1
ax3.set_yticks(yticks[4::5])
ax3.set_yticklabels(yticklabels[4::5])

trial_to_plot = 0
ax4.set_title(f'Raster plot of trial {trial_to_plot}')
ax4.set_xlabel('Time (s)')
ax4.set_ylabel('Spike train index')
for i, spiketrain in enumerate(spiketrains_oscillator[trial_to_plot]):
    ax4.plot(spiketrain, np.ones_like(spiketrain) * i, ls='', marker='|')

plt.tight_layout()
plt.show()
```

Here we generate neuronal spikes with a hidden state that is a circle.

We generate 20 trials of recording.

Each trial has 50 neurons spanning about 2 seconds of activities “recorded” by electrodes.

Our goal is infer the hidden circle from these 20 renditions of observation.



In [6]:
```
# bin data for GPFA
from elephant.gpfa.gpfa_util import get_seqs
```

In [7]:
```
bin_size = 20 * pq.ms
seqs = get_seqs(spiketrains_oscillator, bin_size, use_sqrt=True)
```

The final output `seqs` has the format

In [8]:
```
"""
seqs : np.recarray
        data structure, whose nth entry (corresponding to the nth experimental
        trial) has fields
        T : int
            number of timesteps in the trial
        y : (yDim, T) np.ndarray
            neural data
""";
```

In [9]:
```
np.save('simulated_data1.npy', seqs)
```

In [10]:
```
np.save('simulated_groundtruth.npy', oscillator_trajectory_2dim)
```

In [11]:
```
# load numpy array like this:
# arr = np.load('simulated_data1.npy',allow_pickle=True)
```

In [ ]:
```

```

2. Vanilla GPFA

It is vanilla in the sense that it only handles trials with the same length for now.

In [2]:
```
%reload_ext autoreload
%autoreload 2
```

In [83]:
```
import numpy as np
import quantities as pq
from sklearn.decomposition import FactorAnalysis
import matplotlib.pyplot as plt
```

In [84]:
```
from e_step import e_step
from m_step import m_step
from postprocessing import post_processing
```

In [19]:
```
# =====================
# load simulated data
# =====================

seqs = np.load('simulated_data1.npy',allow_pickle=True)
```

In [55]:
```
# ==================================
# Initialize state model parameters
# ==================================
x_dim = 2
bin_width=20.0    # in ms, this should match how we simulated the synthetic data.
tau_init=100.0
eps_init=1.0E-3
em_tol = 1.0E-3

max_iteration_num = 100

params_init = dict()
params_init['covType'] = 'rbf' # so far only rbf is implemented for this vanilla version
# GP timescale
# Assume binWidth is the time step size.
params_init['gamma'] = (bin_width / tau_init) ** 2 * np.ones(x_dim)
# GP noise variance
params_init['eps'] = eps_init * np.ones(x_dim)

# ========================================
# Initialize observation model parameters
# ========================================
print('Initializing parameters using factor analysis...')

y_all = np.hstack(seqs['y'])
fa = FactorAnalysis(n_components=x_dim, copy=True,
                    noise_variance_init=np.diag(np.cov(y_all, bias=True)))
fa.fit(y_all.T)
params_init['d'] = y_all.mean(axis=1)
params_init['C'] = fa.components_.T
params_init['R'] = np.diag(fa.noise_variance_)
params_init['x_dim'] = 2
params_init['tau'] = tau_init
```

```
Initializing parameters using factor analysis...
```

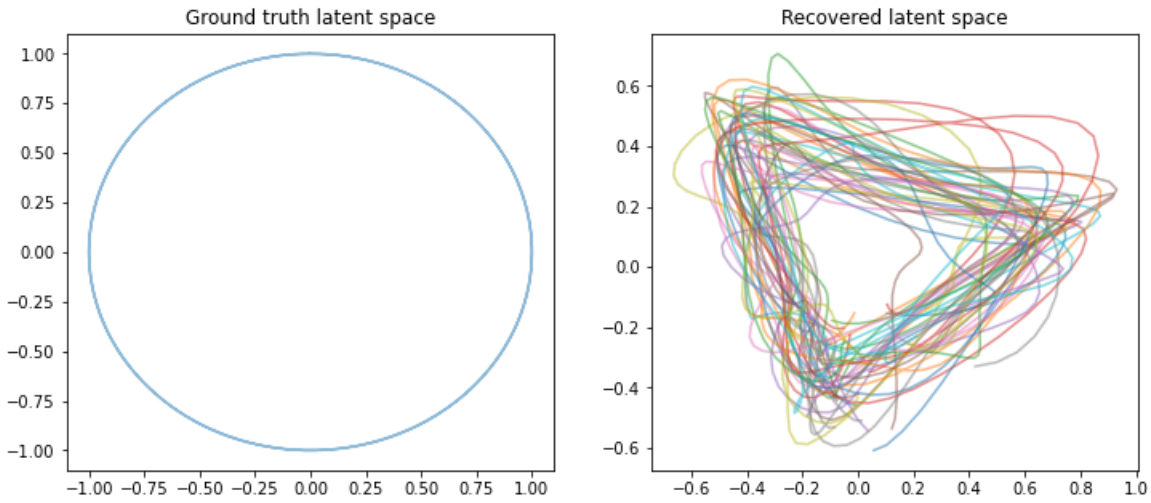
In [57]:
```
# =====================
# Fit model parameters
# =====================

params = params_init
for i in range(max_iteration_num):
    seqs_out, LL_i = e_step(seqs, params)
    params = m_step(seqs_out, params)


    # Check convergence
    if i <= 1:
        LL_base = LL_i
        LL_old = LL_i
    elif LL_i < LL_old:
        print(f"\nError: Log likelihood decreased from {LL_old:.1f} to {LL_i:.1f}")
        break
    elif (LL_i - LL_base) < (1 + em_tol) * (LL_old - LL_base):
        print(f"\nConverged after {i+1} EM iterations")
        break
    LL_old = LL_i
```

```
Converged after 42 EM iterations
```

In [86]:
```
# =====================
# Post processing
# =====================
params_est, seq_final = post_processing(params, seqs_out)
```

In [88]:
```
# results
groudtruth = np.load('simulated_groundtruth.npy',allow_pickle=True)
```

In [93]:
```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

ax1.plot(groudtruth[0,:], groudtruth[1,:], alpha = 0.5)
ax1.set_title("Ground truth latent space")

for n in range(len(seq_final)):
    reduced_data = seq_final["latent_variable_orth"][n]
    ax2.plot(reduced_data[0,:], reduced_data[1,:], alpha = 0.5)
    ax2.set_title("Recovered latent space")
```



In [ ]:

In above, different colors plot the inferred latent state from each trial.
We see that we have recovered the topological structure of the latent staes.

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 0.0048220157623291016 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))
```

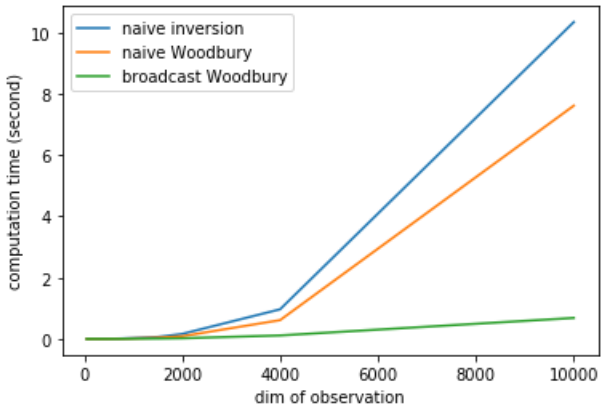
Out[10]:
```
1.6370904631912708e-11
```

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 [ ]: