The Dual-MINDy/gBPKF `Package`

Matthew Singh

January 22, 2025

Introduction

This MATLAB package provides the tools to implement the gBPKF algorithm as introduced in the papers:

Efficient identification for modeling high-dimensional brain dynamics (2022).

Precision data-driven modeling of cortical dynamics reveals person-specific mechanisms underpinning brain electrophysiology (2025).

The documentation is a work-in-progress and more details will be added. This also doubles as an extended toolbox for training vanilla-RNNs with standard-BPTT (no Kalman Filter) being a very limited subset of its functionality. Thus, there are many features and options well-outside the range used in the above papers (e.g. many gradient-optimizers besides NADAM).

This code seeks to solve problems of the form:

$$x_{t+1} = f_\theta(x_t) + \varepsilon_t \tag{1}$$

$$y_t = H_t x_t + \eta_t \tag{2}$$

In terms of parameters θ. Here, x_t is the unknown, true state of the system, evolving according to f and process noise ε with covariance Q_t. The observations y_t are a linear transformation of x_t defined by the measurement matrix H_t with the addition of measurement noise η_t with covariance R_t. You will note that

in our formulation, all components (except) θ are potentially time-varying. In the current implementation, the measurement transformation H_t and the noise covariances Q_t, R_t are allowed to differ between user-defined segments of data (e.g. different recordings) but are not currently coded to vary continuously.

The objective is to recover the model parameters θ (and potentially Q_t) using only the observations y_t. The heart of the gBPKF algorithm involves further estimating the true system-states x_t for small chunks of data. However, the current implementation does not return a final estimate of x_t.

In this implementation, we start with the generic RNN form for the process:

$$f(x_t) = W\psi(x) + Dx_t + c \tag{3}$$

$$\psi(x) := \tanh(s \circ x + v) \tag{4}$$

Hence, the unknown parameters consist of the connection matrix W, the gain s, the shift v, the discrete time-constant D, the baseline "drive" vector c.

This form allows a vast variety of biological models by placing additional constraints on the parameters.

For instance, the EI models used in the original two papers can be formed by structuring x_t to have both excitatory and inhibitory activition. For 100 regions we might choose the first 100 elements of x_t to be excitatory activity and the next 100 to be inhibitory activity, so that x_t is 200-dimensional. We add meaning to neuron-types through constraints on W. In this case, W has the block form (with each block 100×100):

$$W := \begin{bmatrix} W^{EE} & \beta^{IE} \\ W^{EI} & \beta^{II} \end{bmatrix} \tag{5}$$

Here, excitatory connection matrices ($W^{EE,EI}$) are constrained to be non-negative. By contrast, inhibitory connections $\beta^{II,IE}$ are constrained to be non-positive and diagonal, reflecting the local nature of interneurons. The corresponding measurement matrix is

$$H = \begin{bmatrix} -L & 0_{m\times n} \end{bmatrix} \tag{6}$$

With L denoting the leadfield-matrix for channel-level data or (negative) identity for source-localized data (i.e. $-L = I_n$). The zeros matrix on the right, reflects that inhibitory cells do not generally contribute to the MEG/EEG signal.

Throughout we make use of the term "population" to denote one element of the state-vector x_t. In the original papers there are 100 brain regions with 200 total populations: one excitatory and one inhibitory per region.

Note that EI is read as the connection from excitatory to inhibitory. Within matrices $W_{a,b}$ is read as the connection to a from b. Model-specification generally should contain additional constraints as we discuss later.

We also strongly recommend constraining $W^{EE,EI}$ with a mask specifying which connections are forced-zero as demonstrated in the next section. Unfortunately, brain folding generates colinearities (especially around the midline) that can't be resolved without further constraints.

A full simulation example with documentation is provided in the script: BPKF_Example_Sim

1 General Remarks

Users will note that the package contains a very large number of parameter options. Many of these were never changed and are unlikely to affect results (i.e. should just be close to zero or one), but we offer full control.

The gBPKF algorithm is implemented in the main function BPKF_Full. This function takes 6 arguments (a 7th argument for inputs is under-construction).

The initial Xguess argument (for an initial guess of x_t is currently deprecated, as the recommended methods do not need an initial guess. For continuity's sake, however, we are still requiring this to be passed.

Xguess: The initial guess of x_t (can be random, this is **deprecated** for now)
MeasSet: The measurements y_t
ParStr: General settings
KalSpec: Kalman-Filter settings
ModelSpec: Model-Specification
GradSpec: Gradient-optimizer settings

2 Data Formatting

2.1 Xguess (deprecated) and MeasSet

The first two arguments are cell-arrays of timeseries. Each element of the cell-array is one continuous recording specified as an $n \times t$ matrix for n variables and t timepoints per recording. The arguments Xguess and MeasSet should contain the same number of matrices (recordings). Recordings can be different lengths and contain different channel configurations/counts (see later, not recommended for beginners). However, for each recording, the number of timepoints t should be the same between the measurements and the initial state-guess. Because we are fitting a single model, the number of rows in each Xguess matrix should be the same and equal to the dimension of x_t. Small random values are fine for Xguess which is currently deprecated (ignore it). The number of columns for each matrix in MeasSet should match the number of channels for that recording.

3 ParStr

The ParStr structure specifies some high-level model options including the loss function, batch size, number of batches, and settings for recording error, intermediary estimates, etc.

3.1 Configuring Data Sampling

At each iteration (batch), one-or-more data-chunks (minibatches) are selected as training-data. Gradients are accumulated during each minibatch and the model is updated at each batch iteration. These functions are controlled with:

ParStr.BatchSz = number of minibatches per batch ParStr.NBatch = number of batch iterations

When using EKF estimation (recommended), several proximal time-segments are selected per minibatched and share a common covariance estimate. The number of time-segments is specified by ParStr.nStack. By default, these time-segments are spaced 5 time-points apart, but this value can be changed by setting ParStr.ShiftSpace equal the desired value. The use of overlapping time-

segments allows efficient memory access while retaining some sensitivity to non-stationary features of the covariance (see SI in the 2025 paper).

4 KalSpec

KalSpec specifies the Kalman-Filter implementation. In brief, the forward-pass of the gBPKF algorithm contains 3 phases: 1) the initial estimate of x_0, P_0; 2) refining x_t estimates using the Kalman Filter; and 3) free-running multi-step predictions (without filtering) once a good x_t is found.

4.1 Specifying the Initial Estimate

The initial distribution estimates, based on mean-square-error are fully determined by y_0 and the covariances of x and y. Replacing $cov(y)$ with its theoretical value $(H_t cov(x) H_t^T + R_t)$ further yields an equation in terms of y_0 (known) and $ov(x)$ (unknown). The code includes two ways to estimate $cov(x)$: through simulating solutions (recommended) or by directly learning $cov(x)$ (not recommended).

The recommended way (simulation)is engaged by setting KalSpec.BFcov='s'. The simulation size is dictated by the fields: KalSpec.nSim: number of simulations to run (in parallel) KalSpec.SimLength: simulation duration

In general, long simulations are required to estimate a high-dimensional covariance matrix. We speed things up in two ways: 1) For efficiency, we reuse the end of previous simulations to randomly re-seed initial conditions of new simulations. This enables fairly short simulations per-batch as we don't have to wait for the system to approach the steady-state distribution. The number of previous values saved for reseeding is set by KalSpec.nSaveStart. 2) The covariance estimate Σ_0 is autoregressively updated between batches with AR-coefficient set by KalSpec.decP=$u \in (0, 1)$. For batch j:

$$\Sigma_0^{j+1} = (1-u) cov(x_{sim}) + u * \Sigma_0^j \tag{7}$$

This estimate is useful, by the end of model-training but can be very ill-behaved when the model is random (early training). Therefore, we use a convex combination of this estimate and a prior, baseline estimate of $cov(x)$:

$$cov(x) \approx (1-q)\Sigma_0 + q * P_{fix} \tag{8}$$

Here, the weighting constant q is set by KalSpec.decFix=$q \in (0,1)$ and the baseline covariance is set by KalSpec.Pfix=P_{fix}. For MEG data, we used:

KalSpec.BFcov='s';
KalSpec.Pfix=eye(n)/4;
KalSpec.decFix=0.1;
KalSpec.decP=0.95;
KalSpec.SimLength=50;
KalSpec.nSim=50;
KalSpec.nSaveStart=250;

Changing these settings (if desired) ultimately comes down to balancing runtime, numerical-stability, and accuracy-per-iteration.

Alternatively (not recommended) one can treat $cov(x)$ as an unknown parameter with the functional form: $cov(x) = P_{fix} + P_{rt}P_{rt}^T$. Here P_{fix} is fixed, positive-definite and P_{rt} is learned along with the other parameters. This option is specified by setting KalSpec.BF='y'; KalSpec.Pfix=P_{fix}. Finally, the initial value of P_{rt} is specified (via Cholesky decomposition) by setting KalSpec.Pbase=$P_{rt}P_{rt}^T$.

4.2 Specifying the loss-function

All loss functions take the form:

$$H(z^T M z) \tag{9}$$

in which H is a positive scalar function, M is a positive semi-definite matrix, and z is the model-prediction error in terms of predicting y. By default, the loss function is quadratic so H(x)=x, but H can be set to Huber loss by specifying ParStr.CostFun='Huber';

If using Huber-loss, the fields ParStr.decHuber and ParStr.HuberScale should be specified. In this case, the Huber-loss parameter alpha updates according to

$$\alpha_{t+1} = d\alpha_t + (1-d)c * med(L) \tag{10}$$

In which ParStr.decHuber=d, ParStr.HuberScale=c and L denotes the median quadratic loss ($z^T M z$)-across samples for the current minibatch. Values of α are specfic to recording, and prediction-step. This is meant to auto-adjust α across setups. In the 2025 paper, we used: ParStr.decHuber=0.99; ParStr.HuberScale=2;

The cost-matrix M is specified through ParStr.ErrMat. M can be specified as a cell-array of matrices (one-per recording) or a single matrix if all recordings have the same number of channels. You can pass specific values (a matrix or cell of matrices). For convenience, this field also accepts the string 'L2' which makes $M := I_n$, or the string 'Mahal' which makes $M_t := (H_t Q_t H_t^T + R_t)^{-1}$. This quantity is the Mahalonobis Distance when x_{t-1} is known with certainty.