Applying Gaussian process models to hippocampal grid cell data

These notes are structured as an iPython/Jupyter notebook tutorial written in python 3, and depend only on the numpy and scipy libraries. [download as: .ipynb , .html , .pdf]

Gaussian Processes (GPs) generalize the idea of multivariate Gaussian distributions to distributions over functions. In neuroscience, they can be used to estimate how the firing rate of a neuron varies as a function of other variables (e.g. to track retinal waves). Lately, we've been using Gaussian processes to describe the firing rate map of hippocampal grid cells .

These notes briefly review Bayesian inference and Gaussian processes. We then explore applications of Gaussian Process methods to analyzing grid cell data, eventually constructing a GP model of the log-rate that accounts for the Poisson noise in spike count data. Along the way, we also discuss fast approximations for these methods, like kernel density estimation , or approximating GP inference using convolutions.

Introduction

First, we briefly review Bayesian inference for multivariate Gaussian variables and Gaussian processes. Then, we construct some synthetic spike-count observations, similar to what one might see in hippocampal grid cells. We then review how to estimate the underlying firing rate map using kernel density estimation, and discuss some regularization choices when data are limited.

Recall: Bayesian inference in multivariate Gaussian distributions

Loosely, Gaussian processes can be viewed as "really big" multivariate Gaussian distributions, with infinitely many variables. It's helpful to review Bayesian inference for multivariate Gaussian variables before continuing.

Consider estimating some jointly Gaussian variables z from observations y. Bayes' rule states that the posterior distribution $\Pr(z|y)$ is proportional to our prior, $\Pr(z)$, times the likelihood of observing y given z, $\Pr(y|z)$:

$$\Pr(z|y) \propto \Pr(y|z)\Pr(z).$$

Consider a case where both $\Pr(y|z)$ and $\Pr(z)$ are Gaussian:

$$\begin{aligned}\Pr(z) &= \mathcal{N}(\mu_0, \Sigma_0)\\ \Pr(z|y) &= \mathcal{N}(y, \Sigma_\epsilon)\end{aligned}$$

We can estimate $\Pr(z|y)$ using Bayes' rule, by multiplying these two probability distributions (and normalizing the result to integrate to one). This product of two multivariate Gaussian distributions is also a multivariate Gaussian distribution, $\hat{z} \sim \mathcal{N}(\mu, \Sigma)$. Its mean and covariance can be calculated as:

$$\begin{aligned}\Sigma &= \left[\Sigma_0^{-1} + \Sigma_\epsilon^{-1}\right]^{-1}\\ \mu &= \Sigma\left[\Sigma_0^{-1}\mu_0 + \Sigma_\epsilon^{-1}y\right]\end{aligned} \tag{1}$$

In other textbooks or tutorials, you might also see this written as

$$\begin{aligned}\Sigma &= \Sigma_0 - \Sigma_0[\Sigma_0 + \Sigma_\epsilon]^{-1}\Sigma_0 \\ \mu &= \mu_0 + \Sigma_0[\Sigma_\epsilon + \Sigma_0]^{-1}(y - \mu_0).\end{aligned} \tag{2}$$

Both forms are equivalent, and are related to each other by applying the Sherman–Morrison–Woodbury matrix inversion lemma .

Recall: Gaussian process regression

Gaussian processes are commonly used to estimate a smooth underlying trend from noisy observations. Reter Roelants' notes on Gaussian processes is a clear and detailed introduction.

Consider a GP regression problem for learning $y = f(\mathbf{x})$, where $\mathbf{x} = \{x_1, x_2\}$ are coordinates in 2D. Here, our prior over functions is specified not by a mean and covariance, but by a mean function $m(\mathbf{x})$ and a two-point correlation function $\kappa(\mathbf{x}, \mathbf{x}')$, called a kernel. These are simply functions that accept a set of points, and return a mean vector and covariance matrix evaluated at those points.

For the regression problem, we'd like learn a model of $y = f(\mathbf{x})$ given some initial data $(\mathbf{x}_a, y_a)$. GP regression build a posterior distribution over possible functions $f(\mathbf{x})$, given our prior (mean and kernel), and these observations. For any finite collection of inputs $\mathbf{x}_b$, we can evaluate the GP posterior $y_b = f(\mathbf{x}_b)$ at a specified (also finite) set of desired output points to get a posterior mean and covariance of y_b.

$$\begin{aligned}y_b &\sim \mathcal{N}(\mu, \Sigma) \\ \mu &= \mu_b + \Sigma_{ab}^\top[\Sigma_{aa} + \Sigma_\epsilon]^{-1}(y_1 - \mu_1) \\ \Sigma &= \Sigma_{bb} - \Sigma_{ab}^\top[\Sigma_{aa} + \Sigma_\epsilon]^{-1}\Sigma_{ab},\end{aligned} \tag{3}$$

where the means and covariances are computed according to the prior mean and kernel, $\mu_i{=}m(\mathbf{x}_i)$ and $\sigma_{ij}{=}\kappa(\mathbf{x}_i, \mathbf{x}_j)$, respectively. The observation noise model Σ_ϵ is typically assumed to be i.i.d. Gaussian noise with variance ξ^2, i.e. $\Sigma_\epsilon = \xi^2 I$, although we'll explore some other options here.

To connect GP regression to the Bayesian update for multivariate normal variables, consider sampling both the data and the posterior over the same a fixed set of points $\mathrm{X}_0 = \{\mathbf{x}_1, .., \mathbf{x}_L\}$, i.e. $\mathrm{X}_a = \mathrm{X}_b = \mathrm{X}_0$. In this case, $\mu_a = \mu_b = \mu_0$ and $\Sigma_{aa} = \Sigma_{ab} = \Sigma_{bb} = \Sigma_0$, and the GP regression simplifies to:

$$\begin{aligned}\Sigma &= \Sigma_0 - \Sigma_0[\Sigma_0 + \Sigma_\epsilon]^{-1}\Sigma_0 \\ \mu &= \mu_0 + \Sigma_0[\Sigma_0 + \Sigma_\epsilon]^{-1}(y_1 - \mu_0),\end{aligned} \tag{4}$$

This is identical to the posterior distribution for a multivariate Gaussian model we discussed earlier. Indeed, if your data consist of Gaussian observations over a set of points, and you evaluate the posterior at these same locations, there is no difference between Gaussian Process regression and ordinary Bayesian inference for multivariate Gaussian variables.

Exploring Gaussian Process methods in grid cell data

First, let's set up our Python environment in the notebook.

```
%matplotlib inline
# First, set up envirinment
from pylab import *
mpl.rcParams['figure.figsize'] = (8,2.5)
mpl.rcParams['figure.dpi'] = 200
mpl.rcParams['image.origin'] = 'lower'
mpl.rcParams['image.cmap']='magma'
np.seterr(divide='ignore', invalid='ignore');
```

Simulating some data

Now, let's generate some fake grid cell data. We'll simulate a $L \times L$ spatial grid, and define a periodic grid-like firing intensity. Real data are always a bit messy, so we'll make the arena irregularly shaped, and model some background rate fluctuations, and non-uniform sampling of the grid (maybe the rat visits some locations more than others).

```
L = 100  # Grid size
P = 13   # Grid spacing
α = 0.5  # Grid "sharpness"
μ = 0.2  # Mean firing rate (spikes per sample)

# 2D grid coordinates as complex numbers
c = arange(L)-L//2
coords = 1j*c[:,None]+c[None,:]

def ideal_hex_grid(L,P):
    # Build a hexagonal grid by summing three cosine waves
    θs = exp(1j*array([0,pi/3,2*pi/3]))
    return sum([cos((θ*coords).real*2*pi/P) for θ in θs],0)

# Generate intensity map: Exponentiate and scale mean rate
λ0 = exp(ideal_hex_grid(L,P)*α)
λ0 = λ0*μ/mean(λ0)
```

We also add a bit of zero padding around the data. This allows us to apply convolution kernels using circular convolution, without mixing up data from opposite ides. This will be useful later, since circular convolution can be computed very quickly using the Fast Fourier Transform (FFT) . More generally, we might also want to mask out parts of the space if e.g. the rat was exploring an arena with something other than a square shape.

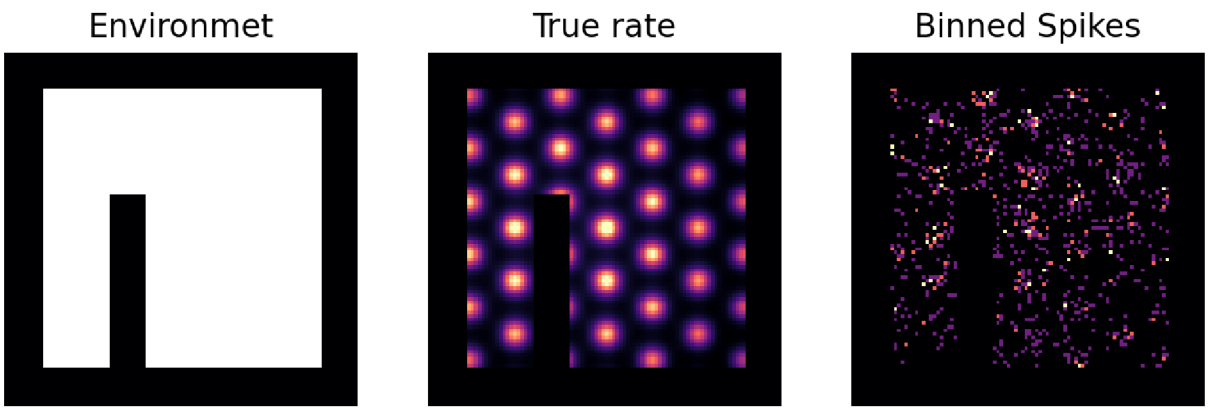
```
# Zero pad edges
mask = np.maximum(abs(real(coords)),abs(imag(coords)))<.4*L

# Simulate oddly shaped arena
mask[:-L*4//10,L*3//10:L*4//10] = False
λ0 = λ0*mask

# For realism, add some background rate changes
λ0 = λ0*(1-abs(coords/L+0.1))
```

We summarize the data in terms of two $L \times L$ arrays: N, which counts the number of times the rat visits each location, and K, which counts the total number of spikes observed in each location. Spikes are sampled as a conditionally-Poisson process with rate λ equal to the intensity at each location.

```
# Simulated a random number of visits to each location
# as well as Poisson spike counts at each location
N = poisson(2*(1-abs(coords/L-0.2j)),size=(L,L))*mask
K = poisson(λ0*N)
```

Let's plot things.

```
def pscale(x,q1=0.5,q2=99.5,domask=True):
    # Plot helper: Scale data by percentiles
    u  = x[mask] if domask else x
    p1 = percentile(u,q1)
    p2 = percentile(u,q2)
    x  = clip((x-p1)/(p2-p1),0,1)
    return x*mask if domask else x

def showim(x,t='',domask=True):
    # Plot helper: Show image with title, no axes
    if len(x.shape)==1: x=x.reshape(L,L)
    imshow(pscale(x,domask=domask));
    axis('off');
    title(t);

subplot(131); showim(mask,'Environmet')
subplot(132); showim(λ0,'True rate')
subplot(133); showim(K,'Binned Spikes');
```



Estimating rate in each bin

The simplest way to estimate the rate at each location is to simply divide the number of observed spikes K by the number of visits N to each location. This is a very noisy estimate (below, left), and is undefined when N is zero.

$$\hat{\lambda} = \frac{K}{N}$$

It's tempting to add a little ad-hoc regularization to handle the $N = 0$ case gracefully, for example

$$\hat{\lambda} = \frac{K}{N + \frac{1}{2}}$$

Tricks like this might seem arbitrary (and perhaps wrong), but can be more formally motivated via Bayesian statistics. We assume that each time the rat visits a location with intensity λ, we observe y spikes, which are Poisson distributed:

$$\Pr(y|\lambda) = \frac{\lambda^y e^{-\lambda}}{\Gamma(y+1)} \tag{5}$$

This gives us a likelihood for estimating λ given y. The gamma distribution is the conjugate prior for Poisson rates, with shape parameter α and rate parameter β:

$$\Pr(\lambda|\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)}\lambda^{\alpha-1}e^{-\beta\lambda}. \tag{6}$$

We can write the likelihood of observing a count observation y given rate λ as:

$$\Pr(\lambda|y) \propto \lambda^y e^{-\lambda}.$$

To combine N observations $y_t \in \{y_1,..,y_T\}$, take the product of the likelihoods for each observation. This reduces to a simplified expression in terms of the total number of spikes $K = \sum_t y_t$:

$$\Pr(\lambda|y_1,..,y_T) \propto \textstyle\prod_t \lambda^{y_t}e^{-\lambda} = \lambda^K e^{-N\lambda}. \tag{7}$$

To assign a rate estimate to locations with missing data, we can define a Bayesian prior for λ. This regularizes bins that have limited data, reducing variance at the expense of increased bias. For regularization strength $\rho > 0$, we set $\Pr(\lambda) \sim \operatorname{Gamma}(\alpha_0, \beta_0)$, with $\beta_0 = \rho$ and $\alpha_0 = \rho(\mu - 1) + 1$, and were μ is the overall average firing rate of the neuron, regardless of location. This leads to a posterior distribution of:

$$\begin{aligned}\Pr(\lambda|y_1,..,y_T) &\propto \lambda^K e^{-N\lambda} \cdot [\lambda^{\alpha_0-1}e^{-\beta_0\lambda}] \\ &= \lambda^{K+\rho(\mu-1)+1}e^{-(N+\rho)\lambda}.\end{aligned} \tag{8}$$

This gives a gamma-distributed posterior with $\alpha = K + \rho(\mu - 1) + 1$ and $\beta = N + \rho$. The posterior mean, α/β, is a regularized estimator $\hat\lambda_\mu$ of the rate:

$$\hat\lambda_\mu = \frac{K + \rho(\mu-1)+1}{N+\rho}. \tag{9}$$

This is biased toward higher rates due the +1 in the numerator. Using the mode $(\alpha - 1)/\beta$ is another option, which lacks this bias:

$$\hat\lambda_{\text{mode}} = \frac{K+\rho\mu}{N+\rho}. \tag{10}$$

One can interpolate between these mean-based and mode-based regularizers with another parameter $\gamma \in [0, 1]$, where $\gamma = 0$ corresponds to the mode-based estimator, and $\gamma = 1$ to the mean-based estimator:

$$\hat\lambda = \frac{K+\rho(\mu-\gamma)+\gamma}{N+\rho}. \tag{11}$$

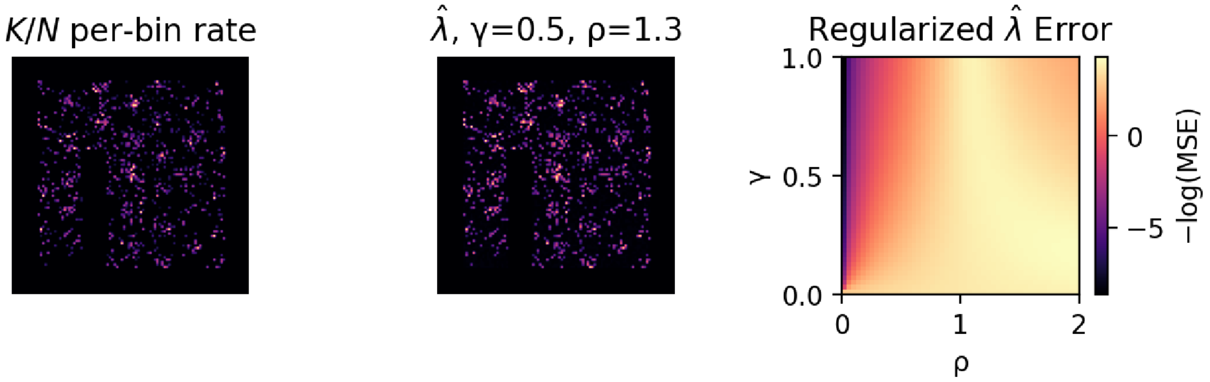
We use $\gamma = 0.5$ and $\rho = 1.3$ as the default here.

```
def regλ(N,K,ρ=1.3,γ=0.5):
    # Regularized rate estimate
    return (K+ρ*(sum(K)/sum(N)-γ)+γ)/(N+ρ)
```

Even with regularization, estimating the rate directly in each bin is far too noisy to be useful (below, right). Why go through all this trouble to define a principled way to regularize counts for single bins then? These regularized rate estimators provide a principled way to define how a rate estimator should behave when data are limited, and can be incorporated into better estimators that pool data from adjacent bins. Next, we explore a simple way to pool data from adjacent bins using kernel density smoothing.

```
from scipy.stats import pearsonr
def printstats(a,b,message):
    # Print RMSE and correlation between two rate maps
    a,b = a[mask],b[mask]
    NMSE = mean((a-b)**2)/sqrt(mean(a**2)*mean(b**2))
    print(message+':')
    print('• Normalized MSE: %0.1f%%'%(100*NMSE))
    print('• Pearson correlation: %0.2f'%pearsonr(a,b)[0])

# Rate per bin using naive and regularized estimators
λhat1 = nan_to_num(K/N)
λhat2 = regλ(N,K)
printstats(λ0,λhat1,'K/N Estimator')
printstats(λ0,λhat2,'Regularized Estimator')

# Effect of regularization on error
ρs = linspace(1e-2,2,51)
γs = linspace(0,1,51)
MAE = array([[nanmean(abs(λ0-regλ(N,K,ρ,γ))**2)\
              for ρ in ρs] for γ in γs])

subplot(131); showim(λhat1,'$K/N$ per-bin rate')
subplot(132); showim(λhat2,'$\hat\lambda$, γ=0.5, ρ=1.3')
subplot(133); imshow(-log(MAE),extent=(0,2,0,1),aspect=2)
xticks([0,1,2]);  xlabel('ρ');
yticks([0,.5,1]); ylabel('γ');
title('Regularized $\hat\lambda$ Error')
colorbar(label='$-\log(\operatorname{MSE})$')
subplots_adjust(wspace=.3,bottom=.25,top=.75);
```

```
K/N Estimator:
• Normalized MSE: 155.3%
• Pearson correlation: 0.27
Regularized Estimator:
• Normalized MSE: 102.6%
• Pearson correlation: 0.29
```



Estimating rate by smoothing

Estimating rate via Kernel Density Estimation (KDE)

The simplest way to estimate rate is to just average together the spike counts in a given region. We'll use a Gaussian blur here. The 2D Gaussian blur is a separable filter , so we can compute it using two 1D Gaussian blurs in each direction. This can also be done quickly using the Fast Fourier Transform (FFT) . This amounts to Kernel Density Estimation (KDE) .

```
def blurkernel(L,σ,normalize=False):
    # Gaussian kernel
    k = exp(-(arange(-L//2,L//2)/σ)**2)
    if normalize:
        k /= sum(k)
    return fftshift(k)

def conv(x,K):
    # Compute circular 2D convolution using FFT
    # Kernel K should already be fourier-transformed
    return real(ifft2(fft2(x.reshape(K.shape))*K))

def blur(x,σ,**kwargs):
    # 2D Gaussian blur via fft
    kern = fft(blurkernel(x.shape[0],σ,**kwargs))
    return conv(x,outer(kern,kern))
```

KDE is usually used to estimate a probability density from a collection of samples. In our case, we must also account for the nonuniform sampling of space. The rat visits some locations more than others. The solution is to smooth the spike counts K and location visits N separately, and then estimate the rate.

If we want to use the regularized rate estimator defined earlier, we should normalize our smoothing kernel $\kappa(\mathbf{x},\mathbf{x}')$ to unit height. This accounts for the fact that smoothing pools multiple observations, and so increases the certainty of our rate estimate relative to the prior.

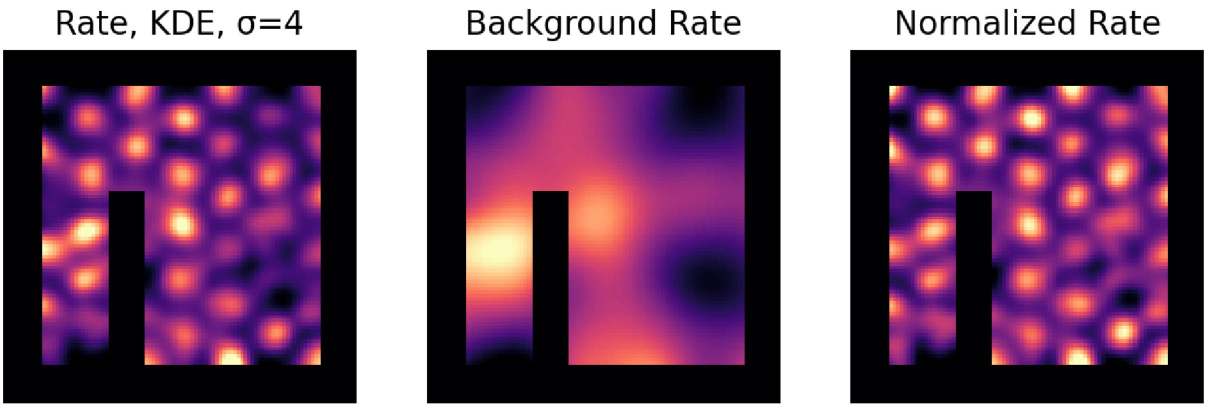
$$\begin{aligned} \hat\lambda_{\text{kde}} &= \frac{\kappa\otimes K + \rho(\mu-\gamma)+\gamma}{\kappa\otimes N+\rho} \\ \kappa(\mathbf{x},\mathbf{x}') &= \exp\left[-\tfrac{(\mathbf{x}-\mathbf{x}')^2}{2\sigma^2}\right] \end{aligned} \tag{12}$$

```
def kdeλ(N,K,σ,**kwargs):
    # Estimate rate using Gaussian KDE
    return regλ(blur(N,σ),blur(K,σ),**kwargs)
```

The result (below, left) is not too bad! For analyzing the underlying grid, we might also want to remove large-scale variations in rate across the arena. We can estimate a background rate also via Gaussian smoothing, and divide out this rate to get a normalized estimate of how rate changes with location (below, right).

```
bluro = 4  # Kernel radius
bgσ   = 12 # Background kernel radius

# KDE estimate of rate and background rate
λhat  = kdeλ(N,K,bluro)
λbg   = kdeλ(N,K,bgσ)
λbar  = λhat/λbg

printstats(λ0,λhat,'KDE Error')
subplot(131); showim(λhat,'Rate, KDE, σ=%d'%bluro);
subplot(132); showim(λbg ,'Background Rate');
subplot(133); showim(λbar,'Normalized Rate');
```

```
KDE Error:
• Normalized MSE: 23.2%
• Pearson correlation: 0.71
```



Inspecting the data

Kernel density smoothing yields a good estimate of the rate map, but we first need to know how much to blur the spike count data. Here, we set the blur radius σ manually, but we can also pick σ in a principled way by examining the autocorrelation of the data. We can calculate the 2D autocorrelation efficiently using the FFT . To focus on fluctuations around the mean rate, we should first subtract any constant component.

```
def zeromean(x):
    # Mean-center data, accounting for masked-out regions
    x = x.reshape(mask.shape)
    return (x-mean(x[mask]))*mask

def fft_acorr(x):
    # Zero-lag normalized to match signal variance
    x   = zeromean(x)
    # Window attenuates boundary artefacts
    win = hanning(L)
    win = outer(win,win)
    # Calculate autocorrelation using FFT
    psd = (abs(fft2(x*win))/L)**2
    acr = fftshift(real(ifft2(psd)))
    # Adjust peak for effects of mask, window
    return acr*var(x[mask])/acr[L//2,L//2]
```

We can collapse this 2D autocorrelation down to 1D by averaging this 2D autocorrelation as a function of radial distance. This radial autocorrelation has a large peak at zero lag, but also several smaller peaks due to the periodicity of the firing function.

```
def radial_average(y):
    # Get radial autocorrelation by averaging 2D autocorrelogram
    i = int32(abs(coords)) # Radial distance
    a = array([mean(y[i==j]) for j in range(L//2+1)])
    return concatenate([a[::-1],a[1:-1]])

def radial_acorr(y):
    # Autocorrelation as a function of distance
    return radial_average(fft_acorr(y))
```

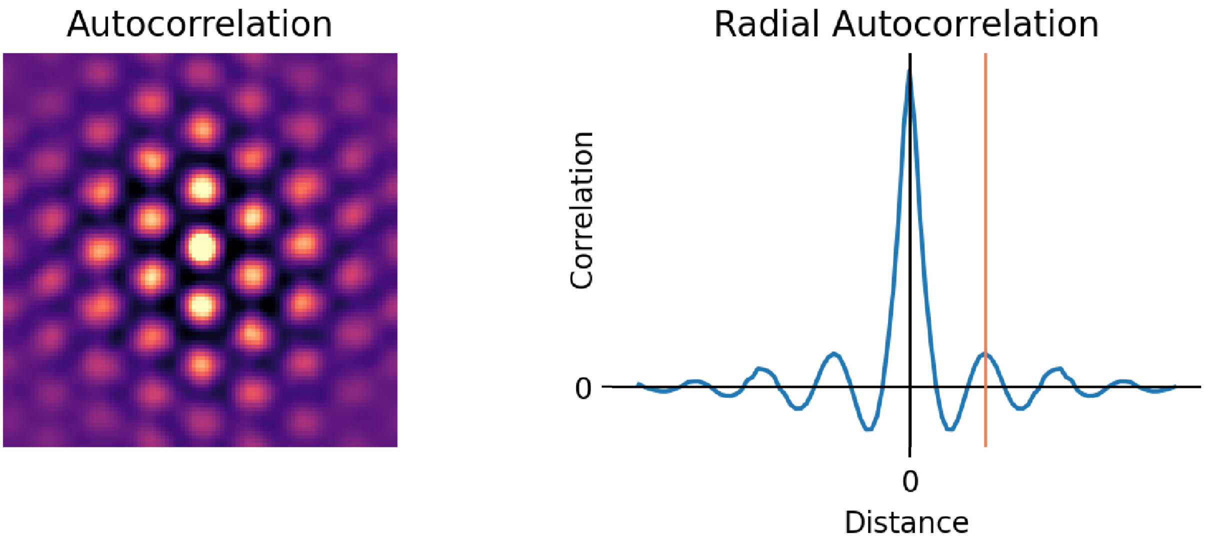
We can estimate the grid spacing based on the location of the first non-zero-lag peak. Here, we use sinc interpolated, computed using the FFT, to find the location of the peak that corresponds to the grid spacing.

```
from scipy.signal import find_peaks
```

```
def acorr_peak(r,F=4):
    # sinc upsample at ×F resolution to get distance to first peak
    z  = zeros((len(r)*(F-1))//2)
    pd = concatenate([z,fftshift(fft(r)),z])
    r2 = real(ifft(fftshift(pd)))
    return min(find_peaks(r2[len(r2)//2:])[0])/F-1
```

For grid cells, the 2D autocorrelation should show a hexagon, which reflects the three sinusoidal components that make up the periodic grid tiling (below, left). Sometimes, cells don't have a perfectly hexagonal grid, or might have slightly different spacing in different directions. However, in this case we focus on cells that have an approximately uniform grid.

```
λhat   = kdeλ(N,K,1.5)          # Small blur for initial esitmate
acorr2 = fft_acorr(λhat)        # Get 2D autocorrelation
acorrR = radial_average(acorr2) # Get radial autocorrelation
P = acorr_peak(acorrR)          # Get distance to first peak in bins

subplot(121); showim(acorr2,'Autocorrelation',domask=False);
subplot(122); plot(acorrR)
[gca().spines[s].set_visible(0) for s in\
 ['top','right','bottom','left']]
xticks([L//2],'0'); xlabel('Distance');
yticks([0]); ylabel(' '*9+'Correlation',labelpad=-9)
title('Radial Autocorrelation');
axhline(0,color='k',lw=1);
axvline(L//2,color='k',lw=1)
axvline(L//2+1+P,color=[.9,.5,.35],lw=1);
```

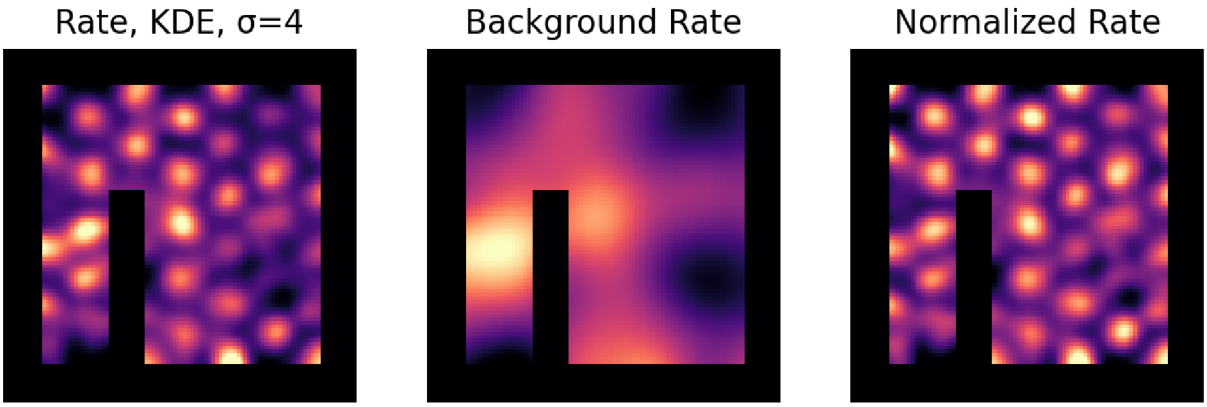


Once we have grid spacing P, we can define other relevant scales. We want to smooth as much as possible, but not so much that we erase the underlying grid. A Gaussian with $\sigma = P/\pi$ is a good heuristic for the largest acceptable smoothing radius. For subtracting the background, we the smallest radius that still filters out the grid. We use $\sigma_{bg} = 3P/\pi$ here.

```
blurσ = P/pi
bgσ   = blurσ*3
λhat  = kdeλ(N,K,blurσ)
λbg   = kdeλ(N,K,bgσ)
λbar  = λhat/λbg

printstats(λ0,λhat,'KDE')
```

```
subplot(131); showim(λhat,'Rate, KDE, σ=%d'%blurσ);
subplot(132); showim(λbg ,'Background Rate');
subplot(133); showim(λbar,'Normalized Rate');
```

```
KDE:
• Normalized MSE: 24.2%
• Pearson correlation: 0.70
```



Smoothing with Gaussian Process regression

KDE smoothing is ok, but we can do better. Gaussian Process (GP) regression provides a flexible way to handle missing data, and also lets us encode more assumptions about the spatial correlations in the underlying rate map.

Let's start by implementing smoothing using GP regression. Recall the formula for the GP posterior mean :

$$\mu=\left[\Sigma_0^{-1}+\Sigma_\epsilon^{-1}\right]^{-1}\left[\Sigma_\epsilon^{-1}y+\Sigma_0^{-1}\mu_0\right] \tag{13}$$

If we set the prior means to zero, the posterior mean simplifies to:

$$\hat\lambda_{\text{gp}}=\left[\Sigma_0^{-1}+\Sigma_\epsilon^{-1}\right]^{-1}\Sigma_\epsilon^{-1}y \tag{14}$$

To set up our GP regression problem, we need to define the prior covariance Σ_0 and measurement precisions Σ_ϵ^{-1}. For now, we assume uniform measurement noise of σ_ϵ^2 per time-point. Rather than treating each time-point separately, we work with the binned spike counts K, and the total number of times the rat visits each location N. Binning observations to a grid pools N observations together into a single estimate. Points with more visits therefore have less error. If the error for a single measurement is σ_ϵ^2, then the error for N measurements is σ_ϵ^2/N.

```
# Prepare error model for GP
ε0 = mean(K)/mean(N) # variance per measurement
τe = N.ravel()/ε0    # precision per bin
```

Bins without data ($N=0$) might seem to pose an issue at first, since their measurement values and error is undefined. This is not an issue in practice if we work with *precision* rather than variance. Precision τ of a Gaussian variable is defined as the reciprocal of its variance, $\tau=1/\sigma^2$ (or, for multivariate Gaussians: the inverse of the covariance matrix). For N measurements, then, we have $\tau=N/\sigma_\epsilon^2$, which is well-defined when $N=0$. We therefore define the precision matrix of the observations as

$$\begin{aligned}\Sigma_\epsilon^{-1} &= \operatorname{diag}[\tau_\epsilon] \\ \tau_\epsilon &= \frac{\operatorname{vec}[N]}{\sigma_\epsilon^2}\end{aligned} \tag{15}$$

where $\operatorname{diag}[]$ denotes constructing a diagonal matrix from a vector, and $\operatorname{vec}[]$ denotes unravelling the $L \times L$ array into a length L^2 vector.

We construct the prior covariance matrix Σ_0 by evaluating the kernel $\kappa(\mathbf{x}, \mathbf{x}')$ for all pairs of bins. Here, we configure the kernel heuristically: we pick an arbitrary smoothing radius, and scale the kernel height to match the estimated variance of the rate map.

```
# Build 2D kernel for the prior
# Scale kernel height to match data variance (heuristic)
k1   = blurkernel(L,blurσ*2)
y    = nan_to_num(K/N)
kern = outer(k1,k1)*var(y[mask])

from scipy.linalg import circulant

def kernel_to_covariance(kern):
    # Covariance is a doubly block-circulant matrix
    # Use np.circulant to build blocks, then copy
    # with shift to make 2D block-circulant matrix
    assert(argmax(kern.ravel())==0)
    L = kern.shape[0]
    b = array([circulant(r) for r in kern])
    b = b.reshape(L**2,L).T
    s = array([roll(b,i*L,1) for i in range(L)])
    return s.reshape(L**2,L**2)
```

Here, we explore a grid size of $L = 100$. For larger problems, the prior covariance might not fit in memory. An alternative solution is possible by solving the linear system via gradient descent and leveraging the FFT. The conjugate gradient option for `scipy.optimize.minimize` performed well in my tests.

The numerical stability of our GP regression will be poor if our prior covariance has small eigenvalues. The eigenvalues of our covariance correspond to the coefficients of the Fourier transform of our kernel, so we can "repair" our kernel by setting too-small eivenvalues with a small positive value.

```
def repair_small_eigenvalues(kern,mineig=1e-6):
    # Kernel must be positive; fix small eigenvalues
    assert(argmax(kern.ravel())==0)
    kfft = fft2(kern)
    keig = abs(kfft)
    υmin = mineig*np.max(keig)
    zero = keig<υmin
    kfft[zero] = υmin
    kern = real(ifft2(maximum(υmin,kfft)))
    return kern
```

To solve our GP regression problem, we use the following form for the posterior mean. This form avoids inverting the prior covariance, so its a bit more numerically stable.

$$\mu = [\Sigma_0 \operatorname{diag}[\tau_\epsilon] + I]^{-1} \Sigma_0^{-1} \operatorname{diag}[\tau_\epsilon] y \tag{16}$$

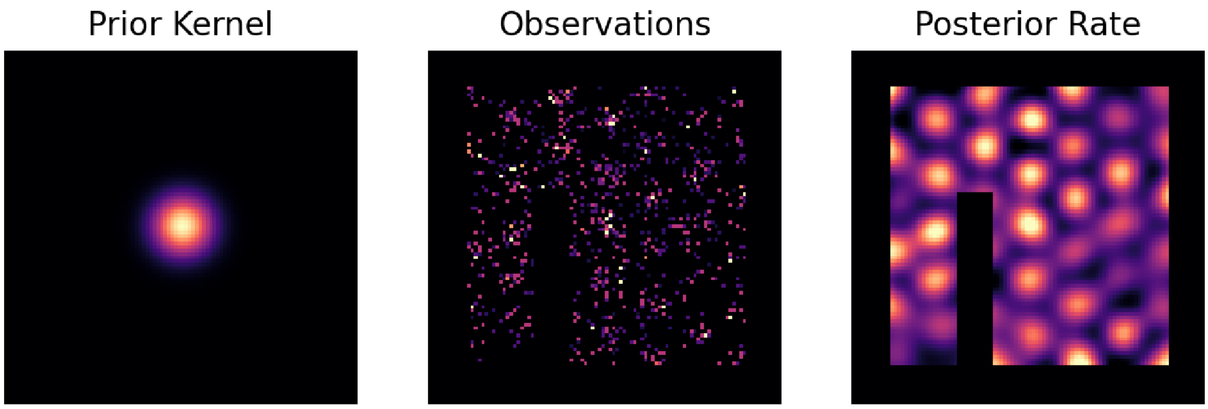
We apply a few optimizations for speed. The product $\operatorname{diag}[\tau_\epsilon]y$ can be evaluated as $\operatorname{diag}[\tau_\epsilon \circ y]$, where $\circ$ is element-wise multiplication. The product $\Sigma_0 \operatorname{diag}[\tau_\epsilon]$ can be evaluated with row-wise multiplication

$\Sigma_0 \circ (\tau_\epsilon \mathbf{1}^\top)$.

The product $\Sigma_0^{-1}\mathbf{u}$ can be evaluated via convolution using the FFT. Since Σ_0^{-1} is the inverse of the prior, we construct an "inverse convolution" by taking the reciprocal of the Fourier coefficients of our prior kernel. Once we've applied this convolution, we also "trim" away any masked regions, to reduce the size of the linear system that we need to solve.

```
def trim(x):
    # Remove masked regions (faster)
    i = mask.ravel()
    return x[i,:][:,i] if len(x.shape)==2 else x[i]

def untrim(x):
    # Put back masked regions in trimmed vector
    return (eye(L*L)[mask.ravel()].T@x).reshape(L,L)

def solveGP(kern,y,τe):
    kern = repair_small_eigenvalues(kern)
    Σ0 = trim(kernel_to_covariance(kern))
    Lm = sum(mask)
    τy = τe.reshape(L,L)*zeromean(y)
    Στy= trim(conv(τy,fft2(kern)).ravel())
    Στe= Σ0*trim(τe)
    return untrim(solve(Στe+eye(Lm),Στy)) + mean(y[mask])

# Solve for posterior mean
λGP1 = solveGP(kern,y,τe)
printstats(λ0,λGP1,'GP')

def showkn(k,t):
    # Convolution kernels always shifted when plotting
    imshow(fftshift(k)); axis('off'); title(t);
subplot(131); showkn(kern,'Prior Kernel');
subplot(132); showim(y,'Observations');
subplot(133); showim(λGP1,'Posterior Rate');
```

```
GP:
• Normalized MSE: 14.5%
• Pearson correlation: 0.80
```



Sometimes GP regression reduces to convolution

It seems like GP regression yields similar results to kernel density estimation. Can we relate these two operations? Recall the solution for the GP posterior:

$$\mu = \left[\Sigma_0^{-1} + \Sigma_\epsilon^{-1}\right]^{-1}\Sigma_\epsilon^{-1} y.$$

The prior Σ_0 is a positive semi-definite matrix, so it can be written in terms of the eigenvalue decomposition

$$\Sigma_0 = \mathrm{F}\,\mathrm{diag}[\tilde{k}]\mathrm{F}^{-1},$$

where $\tilde{k}$ is a vector of eigenvalues and F is a unitary basis. If Σ_ϵ^{-1} can also be diagonalized by F as $\Sigma_\epsilon^{-1} = F\,\mathrm{diag}[\tilde{\tau}]F^\top$, then the solution for the GP posterior simplifies to

$$\mu = \mathrm{F}\,\mathrm{diag}\left[\frac{\tilde{k}\tilde{\tau}}{\tilde{k}\tilde{\tau}+1}\right]\mathrm{F}^{-1} y$$

Generally, Σ_ϵ^{-1} won't share an eigenspace with Σ_0. But, in the special case that all measurements have noise σ_ϵ^2, the precision matrix $\Sigma_\epsilon^{-1} = I/\sigma_\epsilon^2$ is proportional to the identity. The GP posterior then reduces to:

$$\mu = \mathrm{F}\,\mathrm{diag}\left[\frac{\tilde{k}}{\tilde{k}+\sigma_\epsilon^2}\right]\mathrm{F}^{-1} y$$

When the Gaussian process is evaluated on a regularly-spaced grid as we have done here, the eigenspace F is Fourier space, and is the operator F that performs the (unitary) Fourier transform. The above matrix operations can therefore be computed as a convolution using the FFT, vastly accelerating performance. This implies that the GP regression on a regular grid, with homogeneous measurement noise, is equivalent to KDE smoothing with the kernel $g(\mathbf{x}, \mathbf{x}')$, and can be computed with the following convolution ($\otimes$):

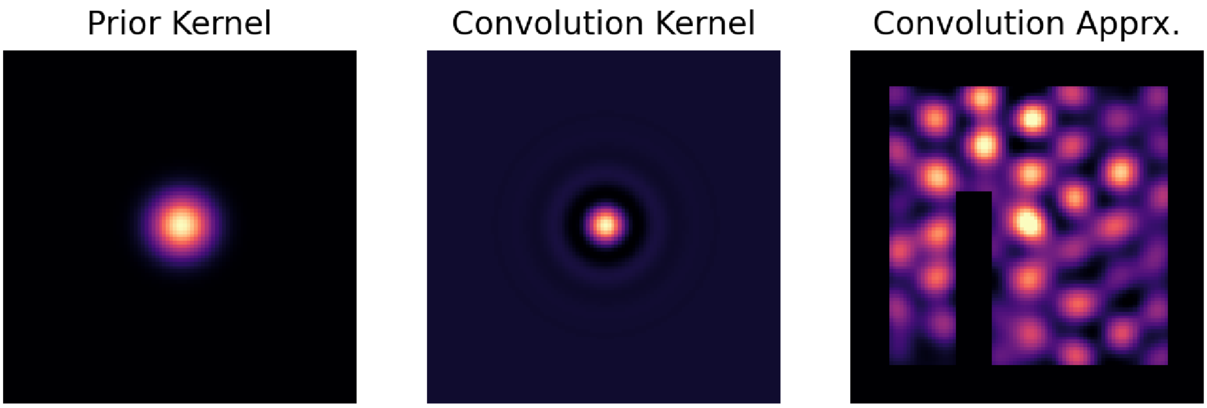
$$\begin{aligned} g(\mathbf{x}, \mathbf{x}') &= \mathrm{F}^{-1}\,\mathrm{diag}\left[\frac{\tilde{k}}{\tilde{k}+\sigma_\epsilon^2}\right] \\ \mu &\approx g \otimes y. \end{aligned} \tag{17}$$

When the measurement error is large $\sigma_\epsilon^2 \gg \tilde{k}$, $g(\mathbf{x}, \mathbf{x}')$ reduces to a convolution kernel approximately proportional to our prior covariance kernel k. When the measurement error is small $\sigma_\epsilon^2 \ll \tilde{k}$, convolution with $g(\mathbf{x}, \mathbf{x}')$ approximates the identity transformation, i.e. $g(\mathbf{x}, \mathbf{x}')$ approximates a delta distribution.

In practice this connection between convolution and GP estimation is not especially useful: cases where a GP is evaluated on a regular grid with homogeneous measurement noise are rare. But, this highlights that sometimes simply filtering the data in space with a convolution kernel gives you something almost as good as a GP regression, at a fraction of the computational cost. In many cases this is good enough.

```
# If measurement error is homogenous
# then GP inference is convolution
kft1 = fft2(kern)
μτ   = mean((N/ε0)[mask])
kftc = (kft1*μτ)/(kft1*μτ+1)
μy   = mean(y[mask])
λcnv = conv(y-μy,kftc)+μy

printstats(λcnv,λGP1,'Error between GP regresion and convolution')
printstats(λcnv,λ0  ,'Error of convolution approximation')
subplot(131); showkn(kern,'Prior Kernel');
subplot(132); showkn(real(ifft2(kftc)),'Convolution Kernel');
subplot(133); showim(λcnv,'Convolution Apprx.');
```

```
Error between GP regresion and convolution:
• Normalized MSE: 16.3%
• Pearson correlation: 0.90
Error of convolution approximation:
• Normalized MSE: 32.2%
• Pearson correlation: 0.73
```



Better priors

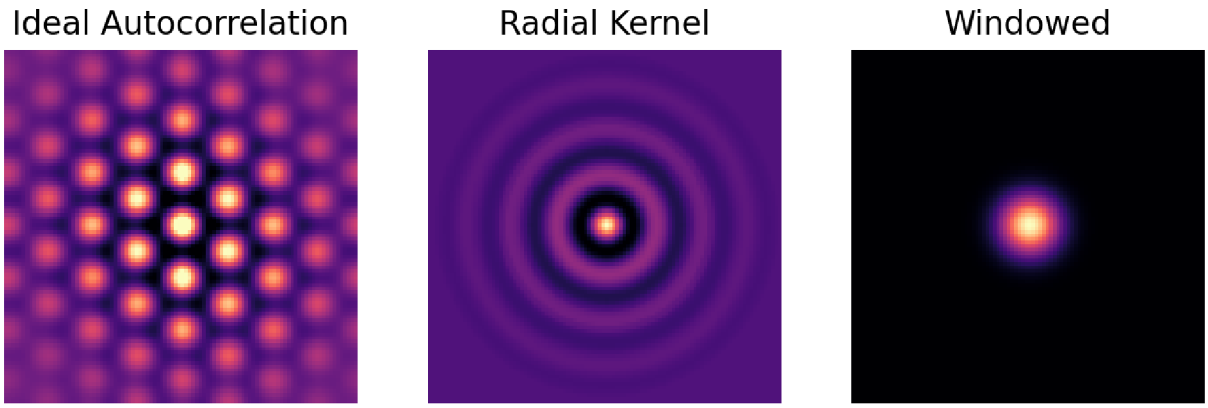
So far, we've only used GP regression with a Gaussian prior. When analyzing data from grid cells, the real power of GP regression lies in being able to encode the knowledge that the grid should be periodic into the GP prior kernel. In this section, we'll construct a periodic prior.

To construct a periodic prior, we estimate the idealize (theoretical) autocorrelation from a perfect grid. To avoid assuming any particular orientation, we make the kernel radially symmetric.

```
from scipy.interpolate import interp1d
def radial_kernel(rk):
    # Make radially symmetric 2D kernel from 1D radial kernel
    r    = abs(coords)
    kern = interp1d(arange(L//2),rk[L//2:],
                    fill_value=0,bounds_error=0)(r)
    return fftshift(kern)

# Make symmetric kernel from autocorrelation of ideal grid
grid  = ideal_hex_grid(L,P)
acgrd = fft_acorr(grid)
axR   = radial_average(acgrd)
kernR = radial_kernel(axR)
```

To avoid inferring long-range interactions where none exist, we taper the kernel to look only at the local neighborhood.

```
# Restrict kernel to local neighborhood and normalize
negative = where(axR[L//2:]<0)[0]
nextpeak = np.min(negative[negative>P])
window   = abs(coords)<nextpeak
kern0    = kernR*fftshift(window)
kern0    = kern0/np.max(kern0)

subplot(131); showim(acgrd,'Ideal Autocorrelation',domask=False);
subplot(132); showkn(kernR,'Radial Kernel');
subplot(133); showkn(kern ,'Windowed');
```



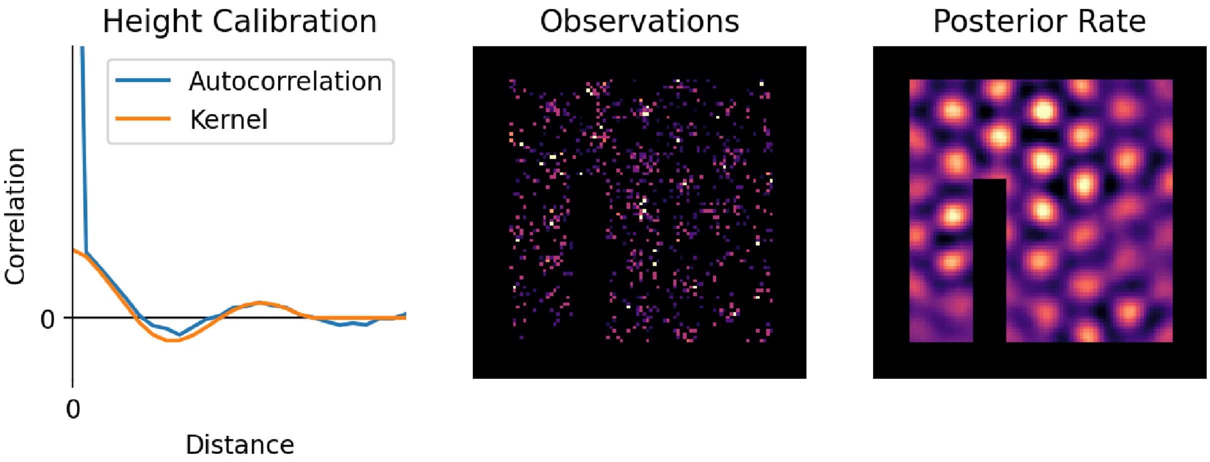
Finally, we adapt the kernel to the observed statistics of the spike count data by scaling the zero-lag peak in the kernel to match a estimate of the variance in the rate. The zero-lag autocorrelation of the data reflects the sum of the true variance in the underlying rates, plus the average measurement noise. To remove the contribution from the measurement noise, we estimating the zero-lag variance by fitting a quadratic polynomial to the correlation at nearby, nonzero lags.

```
def zerolag(ac,r=3):
    # Estimate true zero-lag variance via quadratic interpolation.
    z = array(ac[L//2-r:L//2+r+1])
    v = arange(r*2+1)
    return polyfit(v[v!=r],z[v!=r],2)@[r**2,r,1]

# Estimate zero-lag variance and scale kernel
acorrR1 = radial_acorr(regλ(N,K))
acorrR2 = copy(acorrR1)
v0      = zerolag(acorrR1)
kern    = kern0*v0
acorrR2[L//2] = v0
```

This prior encodes the assumption that the observed spike counts have a periodic underlying structure, and leads to better recovery of the grid fields:

```
# Do GP inference
y    = nan_to_num(K/N)
ε0   = mean((K/N)[N>0])
τe   = N.ravel()/ε0
λGP2 = solveGP(kern,y,τe)
printstats(λ0,λGP2,'GP with periodic kernel')

subplot(131)
axhline(0,color='k',lw=.8)
plot(acorrR1[L//2:],label='Autocorrelation')
plot(kern[0,:L//2],label='Kernel')
legend()
xticks([0]); xlabel('Distance');    xlim(0,L//4)
yticks([0]); ylabel('Correlation'); ylim(ylim()[0],v0*4)
[gca().spines[s].set_visible(0) for s in ['top','right','bottom']];
gca().set_aspect(diff(xlim())/diff(ylim()))
title('Height Calibration')
subplot(132); showim(y,'Observations');
subplot(133); showim(λGP2,'Posterior Rate');
```

```
GP with periodic kernel:
• Normalized MSE: 21.9%
• Pearson correlation: 0.86
```



Heuristic approximation of Poisson noise

Neuronal spiking is typically treated as conditionally Poisson, which means its variance should be proportional to the firing rate. Let's explore a heuristic way to incorporate a Poisson noise assumption into our GP regressions. Earlier, we discussed how the gamma distribution could serve a conjugate prior for Poisson count data. We can also use a Gamma distribution to model the measurement uncertainty from a collection of Poisson observations, and incorporate this model of uncertainty into our GP regression.

The variance of a $\operatorname{Gamma}(\alpha,\beta)$ distribution is $\sigma^2=\alpha/\beta^2$. As discussed earlier, our regularized rate estimator given K spikes in N visits to a given location yields a Gamma distribution with with $\alpha=K+\rho(\mu-1)+1$ and $\beta=N+\rho$. The variance of this distribution, then, is

$$\sigma^2_\epsilon=\frac{\alpha}{\beta^2}=\frac{K+\rho(\mu-1)+1}{(N+\rho)^2}=\frac{\hat\lambda}{N+\rho}.$$

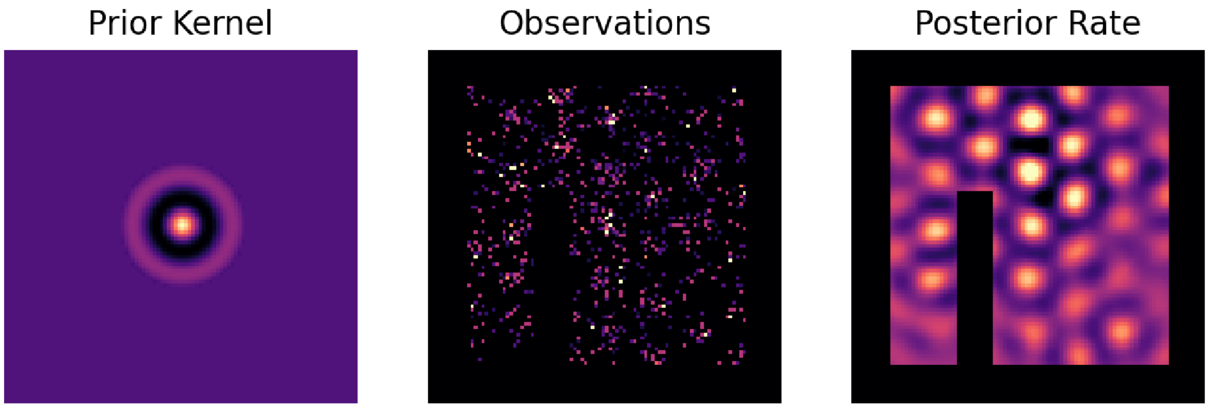
For GP regression, we want to incorporate most of our prior assumptions into the GP prior kernel. This means that we should us only a small amount of regularization ρ when calculating the observations and their uncertainty in each bin. For $\rho=0$ this reduces to

$$\sigma^2_\epsilon=\frac{\alpha}{\beta^2}=\frac{K+1}{N^2}.$$

Performance for this model of the error is mixed: it can work better than assuming constant error when data are limited, but sometimes performs worse than simply assuming uniform variance equal to the neuron's average firing rate. We discuss a more principled way to handle Poisson noise in the next section.

```
# Use estimated rate as measurement error variance
τe    = (N**2/(N*regλ(N,K,1,.5)+1)).ravel()
λGP3 = solveGP(kern,nan_to_num(K/N),τe)
printstats(λ0,λGP3,'GP')
subplot(131); showkn(kern,'Prior Kernel');
subplot(132); showim(y,'Observations');
subplot(133); showim(λGP3,'Posterior Rate');
```

```
GP:
• Normalized MSE: 60.0%
• Pearson correlation: 0.80
```



Log-Gaussian Cox Processes

So far, we've been using GP regression to estimate the underlying rate. This works, but entails some heuristic decisions about how to model measurement noise. Can we do better?

We can get an even better model of the data by fitting a log-Gaussian Cox process model to the binned count observations. This places a Gaussian process prior on the logarithm of the intensity, $\ln(\lambda)$, and assumed that spike count observations are conditionally Poisson:

$$\begin{aligned} y &\sim \operatorname{Poisson}(\lambda) \\ \lambda &= \exp(\mathbf{w}^{\top}\mathbf{x} + \beta) \\ \mathbf{w} &\sim \mathcal{N}(0, \Sigma_0) \end{aligned} \tag{18}$$

Above, $\mathbf{w}$ are the log-rates that we want to infer, and $\mathbf{x}$ is a an indicator vector which is 1 for the rat's current binned location and zero otherwise. The parameter β is a constant bias term.

Recall that the probability of observing spike count y given rate λ, for Poisson-distributed spike counts, is:

$$\Pr(y|\lambda) = \frac{\lambda^y e^{-\lambda}}{\Gamma(y+1)}$$

We work with log-probability for numerical stability. The log-probability of observing spike count y given rate λ, for Poisson-distributed spike counts, is:

$$\ln \Pr(y|\lambda) = y \ln \lambda - \lambda + \text{const.}$$

We estimate a posterior distribution on $\mathbf{w}$ by multiplying our Gaussian process prior by this Poisson likelihood, for all T time points.

The Maximum a posteriori estimate

We find $\mathbf{w}$ that maximizes the posterior probability of the observed spike counts. This is the Maximum A Posteriori (MAP) estimator. For this, we need only calculate the posterior log-probability up to a constant. By convention, we work with the negative log-posterior so that finding the MAP is a minimization problem.

The negative log-posterior $\mathcal{L} = -\ln \Pr(Y|x, \mathbf{w}, \beta)$, summed over all observations $Y = \{y_1, .., y_T\}$, $\mathbf{X} = \{\mathbf{x}_1, .., \mathbf{x}_T\}$, is:

$$\begin{aligned}\mathcal{L} &= \tfrac{1}{2}\mathbf{w}^\top\Sigma_0^{-1}\mathbf{w} - \textstyle\sum_{t=1}^{T}[y_t\ln(\lambda_t) - \lambda_t] + \text{const.}\\ \lambda_t &= \exp(\mathbf{w}^\top\mathbf{x}_t + \beta)\end{aligned}$$

We bin the data into $r \in 1..R$ spatial regions. Each site r has n_r visits in which we observe k_r spikes. Define $\tilde{\lambda}_r = k_r/n_r$ as the empirical rate in each region. We can rewrite the sum over all timepoints in the log likelihood, as a sum over all spatial regions:

$$\textstyle\sum_{t=1}^{T} y_t\ln(\lambda_t) - \lambda_t = \sum_{r=1}^{R} n_r[\tilde{\lambda}_r\ln(\lambda_r) - \lambda_r] \tag{19}$$

The negative log-posterior can then be written as:

$$\mathcal{L} = \tfrac{1}{2}\mathbf{w}^\top\Sigma_0^{-1}\mathbf{w} + \textstyle\sum_{r=1}^{R} n_r[\lambda_r - \tilde{\lambda}_r\ln(\lambda_r)] + \text{const.}$$

Written as a sum over bins like this, we write λ_r as $\lambda_r = e^{\mathbf{w}^\top\mathbf{x}_r+\beta} = e^{\mathbf{w}_r+\beta}$, and the log-posterior simplifies to:

$$\mathcal{L} = \tfrac{1}{2}\mathbf{w}^\top\Sigma_0^{-1}\mathbf{w} + \textstyle\sum_{r=1}^{R} n_r[e^{\mathbf{w}_r+\beta} - \tilde{\lambda}_r(\mathbf{w}_r + \beta)] + \text{const.} \tag{20}$$

We find the MAP by minimizing the above as a function of $(\mathbf{w}, \beta)$. This can be solved via gradient descent. However, I found that most of the optimization routines in Scipy's `minimize` function performed poorly, either crashing, failing to terminate. Scipy's conjugate gradient method performed the best, but achieved poor error tolerance. For larger problems, using the FFT to calculate the log-prior offered some performance improvements.

Overall, however, I found that implementing my own Newton-Raphson method converged the most quickly and to high accuracy. However, computing the Hessian may become prohibitive for larger problems. In this case, I found that applying Scipy's conjugate gradient solver via `scipy.optimize.minimize` , and then "finishing" the optimization with ordinary gradient descent, led to good results.

Finding the Maximum a posteriori using Newton-Raphson

Newton-Raphson solves a linear system on each iteration. Each iteration takes the same amount of time as solving a single GP regression problems. Indeed, one can view each stage of Newton-Raphson as its own GP regression problem. This is the idea behind the Iteratively Reweighted Least Squares (IRLS) approach to fitting Generalized Linear Models (GLMs). The Gaussian process model used here can be viewed as a Poisson GLM with the GP prior acting as a regularizer. Lieven Clement has a good introduction on IRLS.

If we define the concatenated parameter vector $\Theta = (\mathbf{w}, \beta)$, each iteration of Newton-Raphson updates the parameters as

$$\Theta_{i+1} = \Theta_i - \mathbf{H}^{-1}\mathbf{J}, \tag{21}$$

where $\mathbf{J} = \nabla\mathcal{L}$ and $\mathbf{H} = \nabla\nabla^\top\mathcal{L}$ are the Jacobian (gradient) and Hessian (curvature) of our negative log-posterior at the current parameter estimate Θ_i.

To apply Newton-Raphson we need to calculate the Hessian matrix and Jacobian vector. Let's make our lives a bit easier by considering the log-prior and log-likelihood separately.

The negative log-prior is $\frac{1}{2}\mathbf{w}^\top\Sigma_0^{-1}\mathbf{w}$ (up to a constant), and only depends on $\mathbf{w}$. We can express its contribution to the Hessian and Jacobian in terms of vector derivatives in $\mathbf{w}$:

$$\begin{aligned}\mathbf{J}_0^{\mathbf{w}} &= \nabla_{\mathbf{w}}[\tfrac{1}{2}\mathbf{w}^\top\Sigma_0^{-1}\mathbf{w}] \\ &= \tfrac{1}{2}[\mathbf{w}^\top\Sigma_0^{-1} + \Sigma_0^{-1}\mathbf{w}] \\ &= \Sigma_0^{-1}\mathbf{w} \\ \mathbf{H}_0^{\mathbf{w}} &= \nabla\nabla_{\mathbf{w}}^\top[\tfrac{1}{2}\mathbf{w}^\top\Sigma_0^{-1}\mathbf{w}] \\ &= \Sigma_0^{-1}\end{aligned} \tag{22}$$

For the negative log-likelihood $\ell = \sum_{r=1}^R n_r[e^{\mathbf{w}_r+\beta} - \tilde\lambda_r(\mathbf{w}_r+\beta)]$, treat the spatial weight vector $\mathbf{w}$ and bias term β separately. For the weights, only the rate λ_r contributes to the corresponding derivatives in w_r:

$$\begin{aligned}\partial_{w_r}\ell &= n_r[e^{\mathbf{w}_r+\beta} - \tilde\lambda_r] = n_r[\lambda_r - \tilde\lambda_r] \\ \partial^2_{w_r}\ell &= n_r e^{\mathbf{w}_r+\beta} = n_r\lambda_r.\end{aligned} \tag{23}$$

These can be written in vector form as

$$\begin{aligned}\mathbf{J}_\ell^{\mathbf{w}} &= N \circ (\lambda - \tilde\lambda) \\ \mathbf{H}_\ell^{\mathbf{w}} &= \operatorname{diag}[N \circ \lambda],\end{aligned} \tag{24}$$

where $\circ$ denotes element-wise multiplication and $N = \{n_1, . . , n_R\}$, $\lambda = \{\lambda_1, . . , \lambda_R\}$, and $\lambda = \{\tilde\lambda_1, . . , \tilde\lambda_R\}$, are column vectors of the number of visits per bin, the current estimated rates, and the empirical rates, respectively.

For the bias β, we get $\mathbf{J}_\ell^\beta = \sum_r n_r[\lambda_r - \tilde\lambda_r]$ for the Jacobian, and $\mathbf{H}_\ell^\beta = \sum_r n_r\lambda_r$ for the Hessian. Finally, we need the second derivatives between $\mathbf{w}$ and β to complete the Hessian, which are $\mathbf{H}_\ell^{\mathbf{w},\beta} = \nabla_{\mathbf{w}}\partial_\beta\ell = N \circ \lambda$.

Putting it all together, the Jacobian and Hessian for our minimization problem is:

$$\begin{aligned}\mathbf{J} &= \begin{bmatrix}\mathbf{J}_0^{\mathbf{w}} + \mathbf{J}_\ell^{\mathbf{w}} \\ \mathbf{J}_\ell^\beta\end{bmatrix} \\ \mathbf{H} &= \begin{bmatrix}\mathbf{H}_0^{\mathbf{w}} + \mathbf{H}_\ell^{\mathbf{w}} & \mathbf{H}_\ell^{\mathbf{w},\beta} \\ \mathbf{H}_\ell^{\mathbf{w},\beta\top} & \mathbf{H}_\ell^\beta\end{bmatrix}\end{aligned} \tag{25}$$

Initializing a prior for log-Gaussian inference

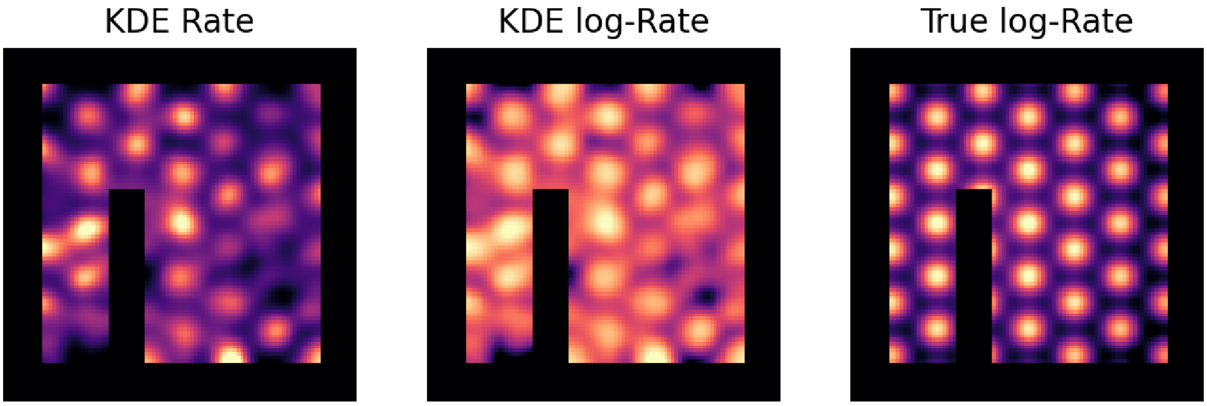
So far, we have defined a Poisson observation model and log-posterior loss function for a log-Gaussian point-process model of the grid cell. We also need to initialize a sensible prior for the weights $\mathbf{w}$, which will correspond to our estimates of $\ln\lambda$.

We do this similarly to as before, but normalize the kernel height based on the variance of a heuristics estimate of the log--rate.

```
# define a "safe log" function
slog = lambda x:log(maximum(1e-2,x))

# Get log-rate from KDE
λhat = kdeλ(N,K,blurσ)
lλh  = slog(λhat)

# Calibrate prior for normalized rate
```

```
acorr = radial_acorr(lλh)
kern  = kern0*zerolag(acorr)

printstats(log(λ0),lλh,'KDE for log-rate')
subplot(131); showim(λhat,'KDE Rate');
subplot(132); showim(lλh,'KDE log-Rate');
subplot(133); showim(log(maximum(1e-5,λ0)),'True log-Rate');
```

```
KDE for log-rate:
• Normalized MSE: 4.7%
• Pearson correlation: 0.68
```



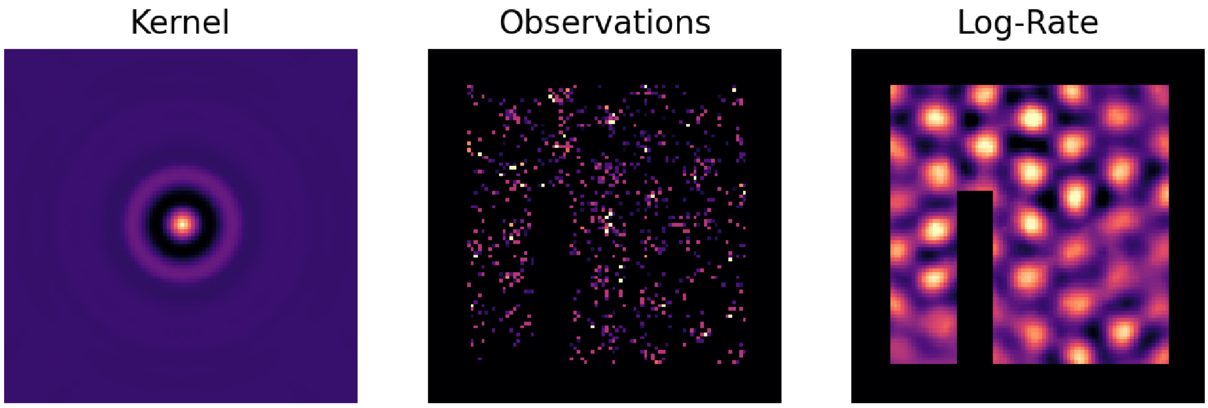
LGCP Inference

To solve the linear problem in the Newton-Raphson update, we take advantage of the fact that the Hessian is positive definite. This means that it has a Cholesky factorization, and that the linear problem can be re-written as two triangular linear problems for improved performance.

```
from scipy.linalg import cholesky as chol
from scipy.linalg import solve_triangular as stri

def csolve(H,J):
    # Solve PSD linear system via Cholesky
    C = chol(H)
    return stri(C,stri(C.T,J,lower=True))
```

```
# Precompute variables; Passed as globals to jac/hess
n    = N.ravel()
y    = nan_to_num(K/N)
kern = repair_small_eigenvalues(kern)
kift = 1.0/fft2(kern)
kinv = real(ifft2(kift))
T0   = kernel_to_covariance(kinv)

def unpack_rate(wβ):
    wβ  = wβ.ravel()
    w,β = wβ[:-1],wβ[-1]
    lnλ = w+β
    return w,β,lnλ,exp(lnλ)

def objective(wβ):
    w,β,lnλ,λ = unpack_rate(wβ)
    dwp = conv(w.reshape(L,L),kift).ravel()
    return 0.5*w@dwp+sum(n*(λ-lnλ*y.ravel()))

def jacobian(wβ):
    w,β,lnλ,λ = unpack_rate(wβ)
    dwp = conv(w.reshape(L,L),kift).ravel()
    dwl = n*(λ-y.ravel())
    return block([dwp+dwl,sum(dwl)])

def hessian(wβ):
    w,β,lnλ,λ = unpack_rate(wβ)
    nλ  = n*λ
    Hww = T0 + diag(nλ)
    Hwβ = array([nλ]).T
    Hββ = array([[sum(nλ)]])
    return block([[Hww,Hwβ],
                  [Hwβ.T,Hββ]])

def newton_raphson(lλh,jacobian,hessian,tol=1e-3):
    # Prepare initial guess
    w = lλh.ravel()
    β = mean(w)
    u = concatenate([w-β,[β]])
    for i in range(10):
        H = hessian(u)
        J = jacobian(u)
        Δ = -csolve(H,J).ravel()
        u += Δ
        convergence = max(abs(Δ))
        print('iter %2d Δ=%7.4f'%(i,convergence))
        if convergence<tol: return unpack_rate(u)
    print('Iteration did not converge')
    return unpack_rate(u)

# Fit model and unpack result
w,β,lnλ,λ = newton_raphson(lλh,jacobian,hessian)
LGCP1 = lnλ.reshape(L,L)
printstats(slog(λ0),LGCP1,'LGCP, log-rate')
subplot(131); showkn(kern,'Kernel');
subplot(132); showim(y,'Observations');
subplot(133); showim(LGCP1,'Log-Rate');
```

```
iter  0 Δ= 2.8042
iter  1 Δ= 0.1126
iter  2 Δ= 0.0075
iter  3 Δ= 0.0000
LGCP, log-rate:
• Normalized MSE: 2.5%
• Pearson correlation: 0.86
```

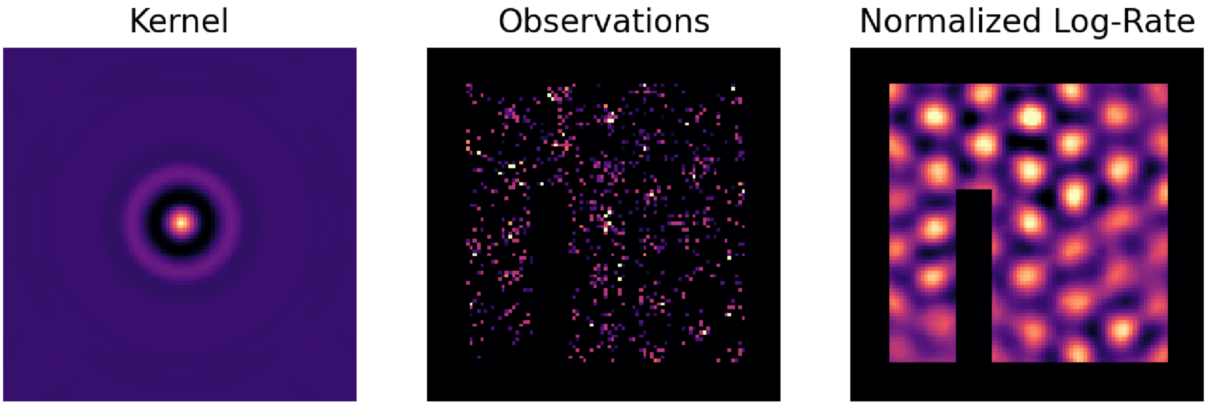


Subtracting the background

We can modify our log-Gaussian cox process model to subtract background rate by adding the estimated background log-rate as an offset during inference.

```
bgσ  = P/pi*3           # Background blur radius
λhat = kdeλ(N,K,blurσ) # Foreground rate
λbg  = kdeλ(N,K,bgσ)   # Background rate
lλh  = slog(λhat)      # Log rate
lλb  = slog(λbg)       # Log background

# Precompute variables; Passed as globals to jac/hess
kern = kern0*zerolag(radial_acorr(lλh-lλb))
kern = repair_small_eigenvalues(kern)
kift = 1.0/fft2(kern)
kinv = real(ifft2(kift))
T0   = kernel_to_covariance(kinv)

# Change rate calculation to add log-background
def unpack_rate(wβ):
    wβ  = wβ.ravel()
    w,β = wβ[:-1],wβ[-1]
    lnλ = w+β+lλb.ravel()
    return w,β,lnλ,exp(lnλ)

# Fit model and unpack result
w2,β2,lnλ2,λ2 = newton_raphson(lλh-lλb,jacobian,hessian)
LGCP2 = lnλ2.reshape(L,L)
printstats(slog(λ0),LGCP2,'LGCP, log-rate')
subplot(131); showkn(kern,'Kernel');
subplot(132); showim(y,'Observations');
subplot(133); showim(w2,'Normalized Log-Rate');
```

```
iter  0 Δ= 2.4889
iter  1 Δ= 0.0739
iter  2 Δ= 0.0026
iter  3 Δ= 0.0000
LGCP, log-rate:
• Normalized MSE: 2.7%
• Pearson correlation: 0.86
```



Convolution approximation

We can also approximate the log-Gaussian model as a convolution. This amounts to considering only the first iteration of Newton-Raphson as if it were a GP regression problem, and replacing the per-bin measurement noise with its average. Consider a single iteration of the Newton-Raphson iteration, for the weights alone. This is:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - [\Sigma_0^{-1} + \Sigma_\varepsilon^{-1}]^{-1}\left[\Sigma_0^{-1}\mathbf{w}_i + N\circ(\lambda-\tilde{\lambda})\right], \tag{26}$$

where $\Sigma_\varepsilon = \mathrm{diag}[N\circ\lambda]^{-1}$. Now, approximate $\Sigma_\varepsilon \approx \sigma_\epsilon^2 I$ where $\sigma_\epsilon^2 = \langle (N\circ\lambda)^{-1}\rangle$ and write

$$\begin{aligned}\hat{\mathbf{w}} &\approx \mathbf{w}_0 - [\Sigma_0^{-1} + \tfrac{1}{\sigma_\epsilon^2}I]^{-1}\left[\Sigma_0^{-1}\mathbf{w}_i + N\circ(\lambda-\tilde{\lambda})\right] \\ &= \mathbf{w}_0 - \mathrm{F}\frac{\sigma_\epsilon^2}{\tilde{k}+\sigma_\epsilon^2}\mathrm{F}^{-1}\mathbf{w}_0 - \mathrm{F}\frac{\sigma_\epsilon^2\tilde{k}}{\tilde{k}+\sigma_\epsilon^2}\mathrm{F}^{-1}N\circ(\lambda-\tilde{\lambda})\end{aligned} \tag{27}$$

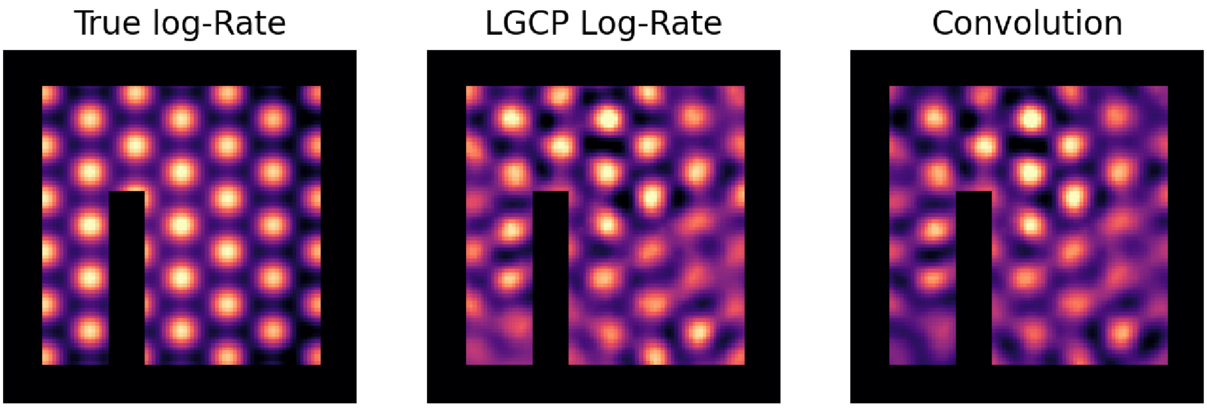
This implies an approximate solution in terms of two convolutions:

$$\begin{aligned}\hat{\mathbf{w}} &\approx \mathbf{w}_0 - g\otimes\{\mathbf{w}_0 + \kappa\otimes[N\circ(\lambda-\tilde{\lambda})]\} \\ g(\mathbf{x},\mathbf{x}') &= \mathrm{F}^{-1}\left[\frac{\sigma_\epsilon^2}{\tilde{k}+\sigma_\epsilon^2},\right]\end{aligned} \tag{28}$$

where $\kappa(\mathbf{x},\mathbf{x}')$ is the GP prior and $\tilde{k}$ is its Fourier transform. This can be calculated almost instantaneously, and differs from the MAP in this example by only a few percent.

```
def LGCP_convolutional(N,K,blurσ,bgσ,kern):
    # Evaluate via convolution
    kern = repair_small_eigenvalues(kern)
    y  = nan_to_num(K/N)
    λ  = kdeλ(N,K,blurσ)
    lb = slog(kdeλ(N,K,bgσ))
    w  = slog(λ)-lb
    β  = mean(w[N>0])
    w  = w-β
    c  = mean(1/(N*λ)[N>0])
    Σf = fft2(kern)
    Gf = c/(c+Σf)
    w -= conv(w+conv(N*(λ-y),Σf),Gf)
    lλ = w+β+lb
    return w,β,lλ,exp(lλ)

w3,β3,LGCP3,λ3 = LGCP_convolutional(N,K,blurσ,bgσ,kern)
printstats(LGCP2,LGCP3,'Error between Newton-Raphson and convolution')
printstats(slog(λ0),LGCP3,'Convolution log-rate estimator error')

subplot(131); showim(slog(λ0),'True log-Rate');
subplot(132); showim(w2,'LGCP Log-Rate')
subplot(133); showim(w3,'Convolution');
```

```
Error between Newton-Raphson and convolution:
• Normalized MSE: 0.6%
• Pearson correlation: 0.96
Convolution log-rate estimator error:
• Normalized MSE: 2.3%
• Pearson correlation: 0.85
```



True log-Rate

LGCP Log-Rate

Convolution

Interim summary

So far, we have

- Introduction:
  - Reviewed Bayesian inference for Gaussian processes
  - Simulated spiking observations from a hippocampal grid cell
  - Discussed estimators of spike rate in single regions
- Smoothing approaches:
  - Used KDE smoothing to infer the underlying grid map
  - Illustrated smoothing using GP regression
  - Showed that GP smoothing is similar to KDE
  - Developed a periodic kernel for inferring grid maps using GP regression
- Log-Gaussian Cox processes (LGCP)
  - Introduced the idea of a LGCP for for inferring log-rates
  - Derived the Jacobian and Hessian for the log-posterior of the LGCP
  - Solved for the MAP estimator of the LGCP using Newton-Raphson
  - Illustrated that the LGCP MAP estimate can be rapidly approximated via convolution
  - Shown how to subtract background rate variations in the LGCP model

We're almost done! By now, we have several very nice ways to smooth grid maps, many of which can leverage an assumption of underlying periodicity of the grid. Some are even fast!

Estimating confidence intervals around peaks

One question that might be nagging you at this point is: should we believe the inferred rate maps? When we see a peak (a "grid field"), is this real, or just a noisy fluctuation? For this, it is useful generate some sort of confidence bounds or other summary of uncertainty in the inferred grid map. First, let's fine our peaks

```
def findpeaks(q,th=-inf,r=1):
    '''
    Find points higher than threshold th, that are also higher
    than all other points in a square neighborhood with radius r.
    '''
    L  = q.shape[0]
    D  = 2*r
    Δ  = range(D+1)
    q0 = q[r:-r,r:-r,...]
    p  = q0>th
    for i,j in {(i,j) for i in Δ for j in Δ if i!=r or j!=r}:
        p &= q0>=q[i:L+i-D,j:L+j-D,...]
    p2 = zeros(q.shape,bool)
    p2[r:-r,r:-r,...] = p
    return p2

# Find peaks
q   = w2.reshape(L,L)
pxy = array(where((findpeaks(q)*mask).T))
```

Both GP regression and LGCP model provide estimates of the posterior covariance, which encodes the uncertainty in our posterior mean (or mode). For GP regression, the covariance is

$$\Sigma_{\text{post}} = \left[\Sigma_0^{-1} + \Sigma_\epsilon^{-1}\right]^{-1} \tag{29}$$

For the LGCP model, we can use the Laplace approximation to model the uncertainty in our MAP estimate $\hat{\mathbf{w}}$. This models the posterior covariance as the inverse of the Hessian, evaluated at $\hat{\mathbf{w}}$. Intuitively, directions with higher curvature in our loss function are more constrained, and so have lower posterior variance. Conversely, unconstrained directions have low curvature, and therefore large posterior variance.

$$\begin{aligned} \Sigma_{\text{post}} &= \mathbf{H}_{\mathbf{w}}^{-1} = \left[\Sigma_0^{-1} + \operatorname{diag}[N \circ \lambda]\right]^{-1} \\ \lambda &= \exp(\hat{\mathbf{w}} + \hat{\beta}) \end{aligned} \tag{30}$$

Incidentally, the curvature of the observation likelihood in the LGCP model, $\operatorname{diag}[N \circ \lambda]$, is equivalent to the measurement precision Σ_ϵ^{-1} in GP regression. This highlights that what the LGCP model is simply finding λ such that the observed measurement errors are consistent with a Poisson error model, $\sigma_\varepsilon^2 = \lambda$.

In these notes, GP regression inferred a posterior distribution on λ, and the LGCP model inferred a posterior distribution on $\ln \lambda$. For generality, we'll refer to both of these in terms of a generic GP posterior for function $f(\mathbf{x})$, and denote $\mathbf{f}$ as the vector that results from to evaluating f over our discrete $L \times L$ grid.

Using the Laplace approximation to calculate uncertainty in peak location

The posterior covariance describes a distribution over different possible rate maps. We can denote this as the posterior mean (or mode) $\mu(\mathbf{x})$, plus some fluctuations $\epsilon(\mathbf{x})$:

$$\begin{aligned} f(\mathbf{x}) &= \mu(\mathbf{x}) + \epsilon(\mathbf{x}) \\ \epsilon(\mathbf{x}) &\sim \mathcal{GP}\left[0, \Sigma_{\text{post}}(\mathbf{x}, \mathbf{x}')\right]. \end{aligned} \tag{31}$$

The fluctuations $\epsilon(\mathbf{x})$ represent the uncertainty in our smoothed rate map. We are interested in how much these fluctuations $\epsilon(\mathbf{x})$ might shift a local maxima in the rate map at location $\mathbf{x}_0$.

If $\mu(\mathbf{x})$ is our inferred posterior mean (or mode), then a perturbation $\epsilon(\mathbf{x})$ changes the rate map to $\mu() + \epsilon(\mathbf{x})$. If perturbations are small relative to the height of out peak, they will move the inferred local maximum by an amount $\Delta\mathbf{x}_0$.

One can calculate $\Delta\mathbf{x}_0$ given $\epsilon(\mathbf{x})$ by considering a second-order Taylor expansion of our rate map as a function of location $\mathbf{x}$. The slope is zero at $\mathbf{x}_0$, since we are at a local maxima. The shift $\Delta\mathbf{x}_0$ is therefore influences by the second-order term:

$$\Delta \mathbf{x}_0 = -\mathbf{H}_\mu^{\mathbf{x}\,-1}\mathbf{J}_\epsilon^{\mathbf{x}}. \tag{32}$$

Above, $\mathbf{J}_\epsilon^{\mathbf{x}} = \nabla_{\mathbf{x}}\epsilon(\mathbf{x}_0)$ is the slope of our perturbation $\epsilon(\mathbf{x})$ at $\mathbf{x}_0$. The larger this is, the more our the peak moves. The size of the shift is also controlled by the curvature of our rate map at the peak, $\mathbf{H}_\mu^{\mathbf{x}} = \nabla_{\mathbf{x}}\nabla_{\mathbf{x}}^\top\mu(\mathbf{x}_0)$. More curved directions shift less, i.e. sharper peaks are more difficult to move.

We are interested in summarizing the overall uncertainty in the location of a peak. This is captured by by the covariance $\Sigma_{\Delta\mathbf{x}_0} = \langle\Delta\mathbf{x}_0(\Delta\mathbf{x}_0)^\top\rangle$:

$$\begin{aligned}
\Sigma_{\Delta\mathbf{x}_0} &= \langle\Delta\mathbf{x}_0(\Delta\mathbf{x}_0)^\top\rangle \\
&= \mathbf{H}_\mu^{\mathbf{x}\,-1}\left\langle\mathbf{J}_\epsilon^{\mathbf{x}}\mathbf{J}_\epsilon^{\mathbf{x}\,\top}\right\rangle\mathbf{H}_\mu^{\mathbf{x}\,-1} \\
&= \mathbf{H}_\mu^{\mathbf{x}\,-1}\left\langle\nabla_{\mathbf{x}}\epsilon(\mathbf{x}_0)\epsilon(\mathbf{x}_0)^\top\nabla_{\mathbf{x}}^\top\right\rangle\mathbf{H}_\mu^{\mathbf{x}\,-1} \\
&= \mathbf{H}_\mu^{\mathbf{x}\,-1}\left[\nabla_{\mathbf{x}}\Sigma_{\text{post}}(\mathbf{x}_0,\mathbf{x}_0)\nabla_{\mathbf{x}}^\top\right]\mathbf{H}_\mu^{\mathbf{x}\,-1}
\end{aligned} \tag{33}$$

In our analyses, we evaluate the posterior mean and covariance on an $L \times L$ grid. Peaks in the posterior mean or mode be identified as local maxima in this grid.

The term $\nabla_{\mathbf{x}}\Sigma(\mathbf{x}_0,\mathbf{x}_0)\nabla_{\mathbf{x}}^\top$ reflects the distribution of the gradient of our rate map at $\mathbf{x}_0$. We need to calculate Σ_{post} to get this. This requires inverting the Hessian, which is nontrivial. However, we don't need to calculate Σ_{post} per se, but rather the product $\Sigma_{\text{post}}\nabla_{\mathbf{x}}^\top$. This can be calculated by getting the Cholesky factor of the Hessian, and then solving a triangular linear system.

```
# Get posterior hessian and Cholesky factor
H = hessian(concatenate([w2,[β2]]))[:-1,:-1]
D = chol(H)
```

For larger problems, it may not be practical to store the full posterior covariance in memory. However, the above vector-matrix products can be solved by re-phrasing them as linear systems, and solving these systems with gradient descent. The inverse of the posterior variance is the sum of the kernel, which can be applied to a vector as a convolution via FFT, and a diagonal matrix of observation errors, which can also be represented compactly.

The term $\nabla_{\mathbf{x}}\Sigma(\mathbf{x}_0,\mathbf{x}_0)\nabla_{\mathbf{x}}^\top$ can then be calculated numerically, by constructing the linear operators for the discrete derivative in each direction of our coordinates $\mathbf{x}$. These derivative operators are simply vectors $\mathbf{d}_{\mathbf{x}_0}^{x_i}$ such that

$$\begin{aligned}
\partial_{x_i} f(\mathbf{x})\Big|_{\mathbf{x}_0} &\approx \mathbf{d}_{\mathbf{x}_0}^{x_i\,\top}\mathbf{f} \\
\nabla_{\mathbf{x}} f(\mathbf{x})\Big|_{\mathbf{x}_0} &\approx \begin{bmatrix}\mathbf{d}_{\mathbf{x}_0}^{x_1} & \mathbf{d}_{\mathbf{x}_0}^{x_2}\end{bmatrix}^\top = \mathbf{D}_{\mathbf{x}_0}^\top\mathbf{f}.
\end{aligned} \tag{34}$$

```
def dx_op(L):
    # 2D difference operator in the 1st coordinate
    dx = zeros((L,L))
    dx[0, 1]=-.5
    dx[0,-1]= .5
    return dx
```

The curvature at our peak can be calculated as $\mathbf{H}_\mu^{\mathbf{x}} = \nabla_{\mathbf{x}}\nabla_{\mathbf{x}}^\top f(\mathbf{x}_0) \approx \mathbf{D}_{\mathbf{x}_0}\mathbf{D}_{\mathbf{x}_0}^\top\mathbf{f}$. We can calculate this quickly for all points at once by convolving our discrete difference operators with the data using FFT.

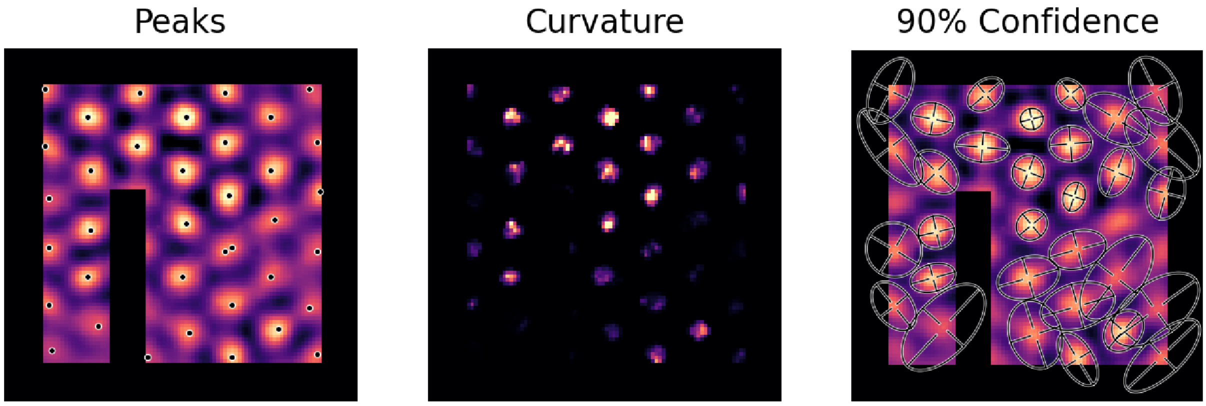
```
def hessian_2D(q):
    # Get Hessian at all points
    dx  = dx_op(q.shape[0])
    f1  = fft2(dx)
    f2  = fft2(dx.T)
    d11 = conv(q,f1*f1)
    d12 = conv(q,f2*f1)
    d22 = conv(q,f2*f2)
    return array([[d11,d12],[d12,d22]]).transpose(2,3,0,1)

# Get Hessian at all points
dx = dx_op(L)
Hx = hessian_2D(q)
Dx = det(Hx)
```

One we have calculated $\Sigma_{\Delta \mathbf{x}_0}$ for a given peak, we can estimate a ellipsoid confidence region for how much that peak location might shift, given our posterior uncertainty.

```
from scipy.stats import chi2
def covariance_crosshairs(S,p=0.8):
    # Generate a collection of (x,y) lines denoting the confidence
    # bound for p fraction of data from 2D covariance matrix S
    sigma  = chi2.isf(1-p,df=2)
    e,v    = eigh(S)
    lines  = list(exp(1j*linspace(0,2*pi,181)))
    lines += [nan]+list(   linspace(-1,-.2,5))
    lines += [nan]+list(1j*linspace(-1,-.2,5))
    lines += [nan]+list(   linspace(.2,.95,5))
    lines += [nan]+list(1j*linspace(.2,.95,5))
    lines = array(lines)
    lines = array([lines.real,lines.imag])*sigma*(e**0.5)[:,None]
    return solve(v,lines)
```

Overall, this enables reasonably fast approximate confidence intervals on the peak locations.

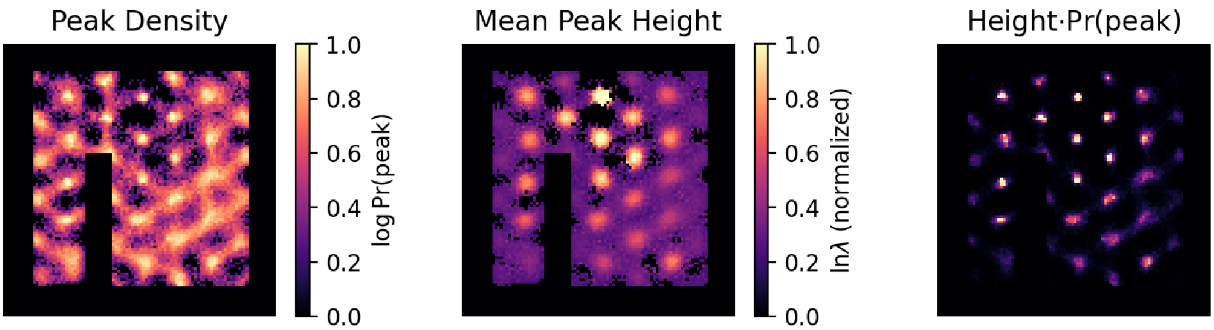
```
def plot_peakbounds(pxy,Hx,D,P):
    # D should be the cholesky factor of the Hessian of the log-posterior
    lx,ly = [],[]
    for x2,x1 in pxy.T:
        # Jacobian at x0
        Δx1 = roll(dx  ,(x1,x2),(0,1))
        Δx2 = roll(dx.T,(x1,x2),(0,1))
        J   = array([Δx1,Δx2]).reshape(2,L**2)
        # Peak location confidence
        vv  = stri(D.T,J.T,lower=True).T
        Ux0 = solve(Hx[x1,x2],vv)
        Σx0 = Ux0@Ux0.T
        # Plot if peak is acceptably localized
        if max(eigh(Σx0)[0])<P:
            cx,cy = covariance_crosshairs(Σx0,p=0.9)
            lx += [nan] + list(cx+x2)
            ly += [nan] + list(cy+x1)
    plot(lx,ly,color='w',lw=1.0)
    plot(lx,ly,color='k',lw=0.5)
    axis('off'); title('90% Confidence');
    xlim(0,L); ylim(0,L)

subplot(131); showim(q,'Peaks');
scatter(*pxy,s=5,facecolor='k',edgecolor='w',lw=0.4)
subplot(132); showim(maximum(0,Dx*maximum(0,q)),'Curvature')
subplot(133); showim(q); plot_peakbounds(pxy,Hx,D,P);
```



Use sampling to assess the probability of a peak

Another approach to characterizing uncertainty is to simply sample from the posterior distribution, and test how many of these samples have a certain property, for example, having a peak at location $\mathbf{x}_0$. This is impractical for larger problems, but can work surprisingly well on modern hardware, even for a grid size of $L = 100$, which has a covariance matrix with 10^8 entries.

```
softmask = blur(mask,5,normalize=True)

def peak_density(w,D,Niter=1000):
    # w: posterior mean or mode vector
    # D: cholesky factor of log-posterior Hessian
    q = stri(D,randn(L**2,Niter))
    q = (q+w.ravel()[:,None]).reshape(L,L,Niter)
    q = (q-mean(mean(q,2)[mask]))*softmask[:,:,None]
    peaks = findpeaks(q,th=std(q))
    dnsty = mean(peaks,axis=2)
    μhght = nan_to_num(sum(q*peaks,2)/sum(peaks,2))
    return dnsty,μhght

dnsty,μhght = peak_density(w3,D)
dnsty = maximum(1/5000,dnsty)

subplot(131); showim(log10(dnsty),'Peak Density')
colorbar(label='$\log\,\Pr(\operatorname{peak})$')
subplot(132); showim(μhght,'Mean Peak Height')
colorbar(label='$\ln\lambda$ (normalized)')
subplot(133); showim(μhght*dnsty,'Height$\cdot\Pr$(peak)')
tight_layout()
subplots_adjust(left=0,right=1,bottom=0.2,wspace=0.15);
```



In sum, we have illustrated the following workflow for analyzing firing rate maps for hippocampal grid cells:

1. Bin spikes into a histogram of total number spikes and visits to each region
2. Use autocorrelation to estimate the grid scale
3. Use an idealized grid to set a prior for the log rate
4. Infer log-rate using kernel density estimation (KDE)
5. Use this KDE estimate as initializer for LGCP regression
6. Heuristically fit a log-Gaussian Cox process model using convolution
7. Identify grid field centers local maxima in the inferred rate map
8. Use the Laplace approximation to fit confidence regions for grid field centers
9. Sample from the GP posterior to estimate the probability of a grid field center in each region

Putting it all together

In this document, we discussed several ways to infer the underlying tuning of hippocampal grid cells. The methods outlines here will also work more generally, for any neuron tuned to a 2D feature space.

Histogram-based estimators are easy, and will work if low spatial resolution is acceptable. However, they aren't very statistically efficient, so they need a lot of data to return a meaningful result. It's also hard to define notions of confidence for histograms.

Kernel density estimators (KDEs) pool data from nearby regions. They are more efficient than Histograms, especially if one optimizes the kernel bandwidth to match the underlying variations in neuronal tuning. For moderate amounts of data, KDE estimators are fast, and return a reasonable estimate of the rate map.

The next step above KDE estimators is Gaussian process (GP) regression. GP regression is more statistically efficient, and also estimate posterior covariance. This enables one to estimate confidence bounds for the inferred rate maps. GP regression requires solving a large linear system. This may not always be possible for large problems, or on limited computer hardware. However, fast convolutional approximations to GP regression are possible, and can perform better than kernel density estimators with similar computational complexity.

Log-Gaussian Cox process (LGCP) regression is a generalization of GP regression. It infers the log-firing rate under a Poisson noise assumption, and can return good estimates from limited data. LGCP regression must be fit via gradient descent or Newton-Raphson. Each iteration of Newton-Raphson has the same computational cost as solving a single GP regression problem. However, convolutional approximations also exist for LGCP regression, providing a fast and statistically-efficient way to estimate rate maps

Overall, we illustrated several approaches to Gaussian-process regression to hippocampal grid cell data. We covered kernel density estimation, Gaussian process regression, and log-Gaussian Cox process regression. Throughout, we discussed practical issues necessary to achieve good performance, like using the FFT when possible, choosing numerically stable forms of the equations, and fast approximations based on convolution. Ultimately, we derived an FFT-based approximation to log-Gaussian Cox process regression. This provides an approach to analyzing grid cell data that is both statistically and computationally efficient.