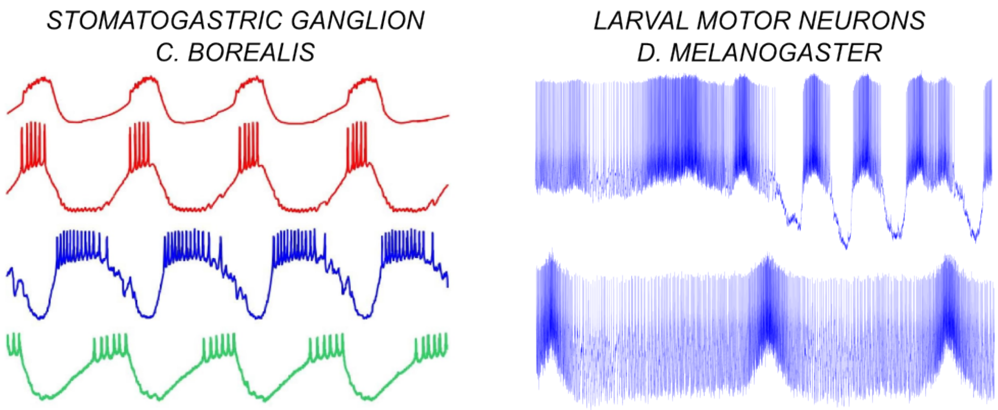
Identifying bursting and tonic activity

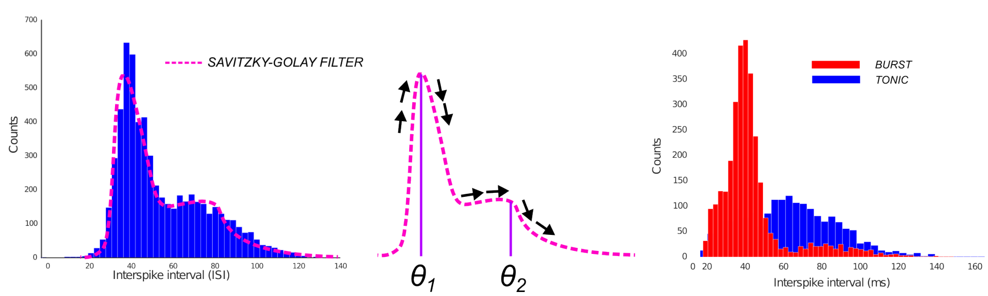
In soma neurons, bursts are easily identifiable. Other times, bursts can be very difficult to differentiate from normal activity, especially in fast spiking neurons.
Question: Is there a robust mechanism to identify bursting and tonic activity?



Approach: Bayesian statistics

- Large positive dV followed by large negative dV signals a potential spike. Conservative potential spikes are averaged to create a filter that is convolved through the trace.
- Spikes are identified by a 1-s moving window so that changes in spike properties during a trace do not affect distal spikes.
- Interspike intervals of identified spikes are binned and then applied to the following algorithm:
  1. The spike ISI histogram is smoothed using a Savitzky-Golay filter.
  2. Peaks are identified using a discrete derivative operation; the ISI value of the peak becomes proto-parameter θ that will seed the prior distribution.
  3. Prior distributions are tested with loss functions.
  4. Markov Chain Monte Carlo simulation assigns data from the priors to the posterior distributions.

We can then use the posterior ISI distributions to assign a confidence to each spike ISI belonging to one distribution or the other.



Savitzky-Golay filter

A convolution coefficient matrix is created and its pseudo-inverse (+) taken

$$\begin{bmatrix} h_0^0 & h_1^0 & \dots & h_n^0 \\ h_0^1 & h_1^1 & \dots & h_n^1 \\ \dots & \dots & \dots & \dots \\ h_0^{degree} & h_1^{degree} & \dots & h_n^{degree} \end{bmatrix}^+$$

where h_i^k is the i^{th} element of the half-window raised to the power k, which increases from 0 to *degree*. The first line of this provides a least-squares solution to the polynomial of degree 0 convolved with the trace, or $A_{0,:} * y$ where y is the trace. My version interpolates the result to make it the same length as the original.

```
def s_golay(x, winsize, degree):
  # Simple Savitzky-Golay filter, inspired by matlab
  assert winsize % 1 == 0, 'winsize␣must␣be␣int!'
  assert winsize % 2 == 1, 'winsize␣must␣be␣odd!'
  assert winsize > 0, 'winsize␣must␣be␣positive!'
  assert winsize > degree + 2, 'winsize␣must␣be␣>␣(degree+2)'

  # Condition data and establish convolution coefficients
  r_degree = range(degree+1)
  halfwin = int((winsize-1)/2)
  convmat = np.mat([ [k**i for i in r_degree]
                   for k in range(-halfwin, halfwin+1) ])
  m = np.linalg.pinv(convmat).A[0]

  # Stretch the signal at extrema so that computation can continue
  start = np.interp(np.linspace(0.5, 0.5*2*halfwin, 2*halfwin),
                   np.linspace(0.,halfwin, halfwin), x[0:halfwin])
  end = np.interp(np.linspace(0.5, 0.5*2*halfwin, 2*halfwin),
                 np.linspace(0.,halfwin, halfwin), x[-(halfwin):])
  x = np.concatenate((start, x, end))

  result = np.convolve(m[::-1], x, mode='valid')
  # In case its too long, shrink it
  pad = int((len(x)-len(result))/2)
  if pad > 0:
    return result[pad-1:-pad]
  return result
```

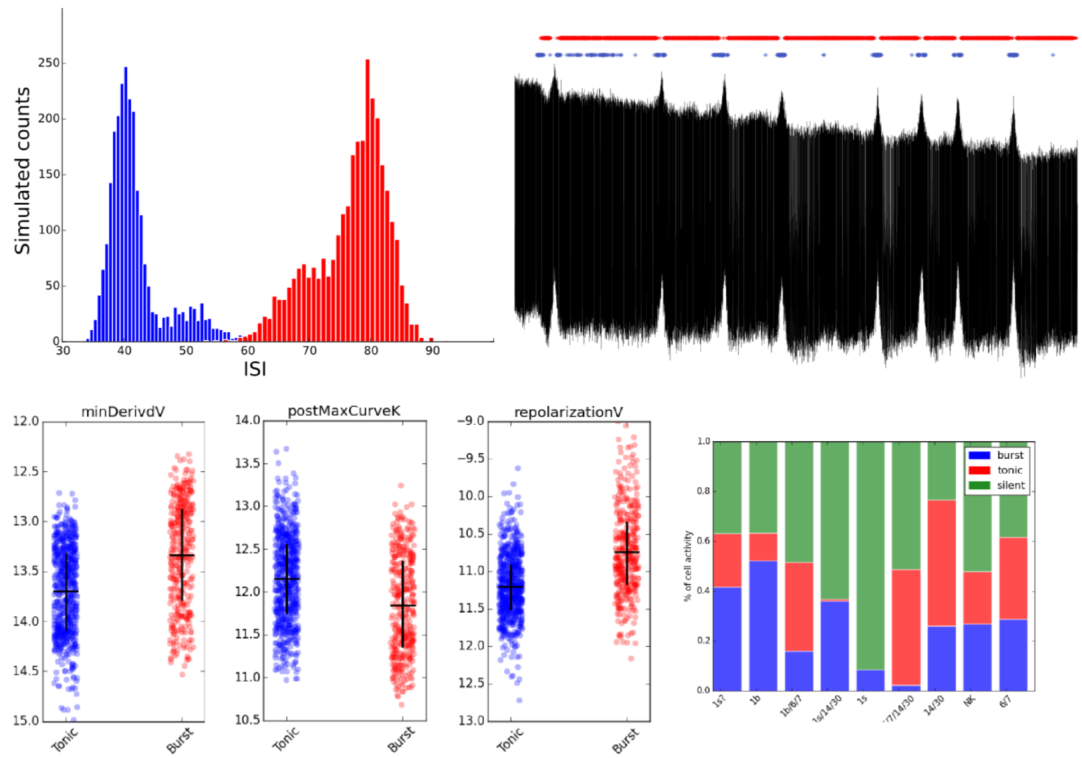
Simulation of ISIs

Prior distributions are normal,

$$N(x|\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where $\mu_i = \theta_i$, determined above with the smoothed ISI histograms, and σ is taken to be a uniformly distributed prior $\sum_i^N \frac{1}{N}$ on $(0, 100)$.

Using Markov chain Monte Carlo packages (PyMC or emcee) allows convergence to parameters for μ and σ. We achieve posterior distributions that are more distinct than expected, but this is actually useful to help provide confidence intervals for each ISI.



Code snippet: Running Markov chain Monte Carlo with PyMC

```
def runMCMC(df, cents, show=False):
  # Run the MCMC algo for as many centers as needed
  if type(cents) is not list:
    cents = [cents]
  numCents = len(cents)
  p = None

  # Tau = the precision of the normal distribution (of the above peaks)
  taus = 1. / pm.Uniform('stds', 0, 100, size=numCents)**2 # tau = 1/sigma**2
  centers = pm.Normal('centers', cents, [0.0025 for i in cents],
                    size=numCents)

  if numCents == 2: # Assignment probability
    p = pm.Uniform('p', 0, 1)
    assignment = pm.Categorical('asisgnment', [p, 1-p],
                          size=len(df.intervals))
    @pm.deterministic
    def center_i(assignment=assignment, centers=centers):
      return centers[assignment]
    @pm.deterministic
    def tau_i(assignment=assignment, taus=taus):
      return taus[assignment]
    observations = pm.Normal('obs', center_i, tau_i, value=df.intervals,
                        observed=True)
```

```
    # Create the model 2 peaks
    mcmc = pm.MCMC([p, assignment, observations, taus, centers])

  else:
    observations = pm.Normal('obs', value=df.intervals, observed=True)
    mcmc = pm.MCMC([observations, taus, centers]) # Create model, 1 peak

  # Run the model
  mcmc.sample(50000)
  center_trace = mcmc.trace("centers")[:]
  try:
    clusts = [center_trace[:,i] for i in range(numCents)]
  except:
    clusts = [center_trace]

  if show:
    for i in range(numCents):
      plt.hist(center_trace[:,i], bins=50, histtype='stepfilled',
               color=['blue', 'red'][i], alpha=0.7)
    plt.show()
  return clusts
```