NEU 501a – HW3

Tom Bertalan

October 19, 2013

1 Binocular Rivalry

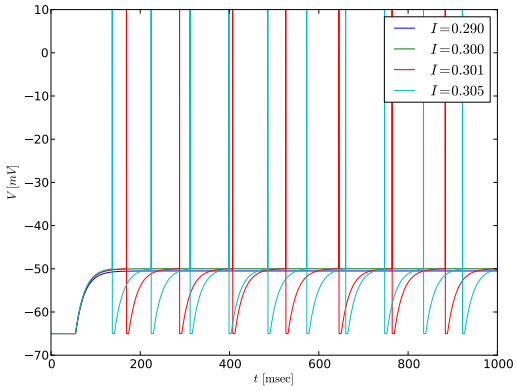
Code is in §3.

1.1 Single I/F Neuron

1.1.1 Spiking Threshold

See `part1A` in §4.1.

For plotting purposes, voltages above $V_{\text{thresh}} = -50\,[mV]$ are here replaced with sudden $+10\,[mV]$ spikes. Bifurcation to a spiking regime occurs at $I = 0.3\,[nA]$.



1.1.2 Ramped Input Current

See `part1B` in §4.1.

$$C\frac{dV}{dt} = I - g_L(V(t) - V_l)$$

Solution:

$$V(t) = c_1 e^{-g_l t/C} + I/g + V_l$$

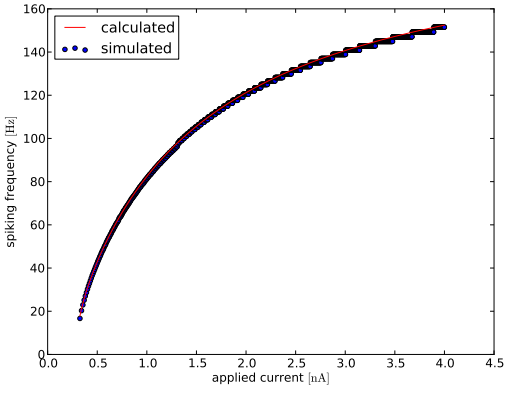
Let $V(t=0) = V_l$, so $c_1 = -I/g$.

$$V(t) = I(1 - e^{-g_l t/C})/g + V_l$$

Solve for t:

$$t(V) = -c/g \ln\left(1 - \frac{g_l}{I}(V(t) - V_l)\right)$$

So, the expected frequency is



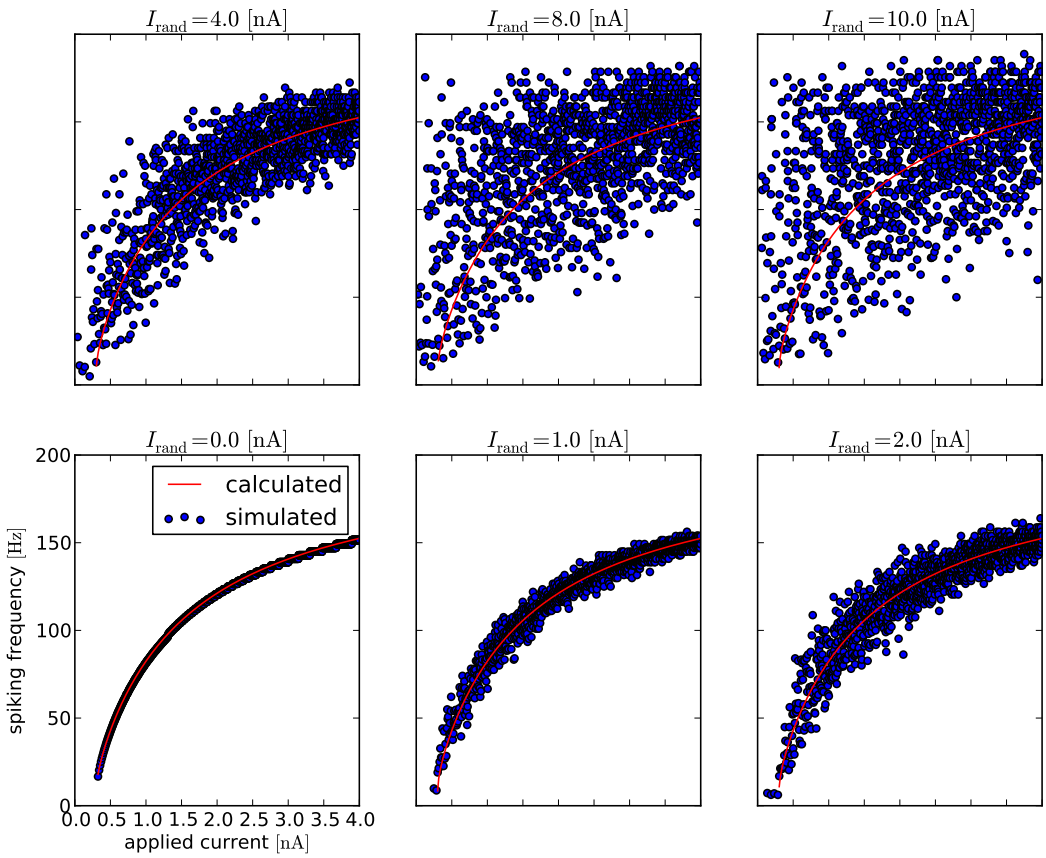
$$f = 1/[t_{\text{ref}} + t(V_{\text{thresh}})] = 1/\left[t_{\text{ref}} - C/g_l \ln\left(1 - g/I(V_{\text{thr}} - V_l)\right)\right]$$

Where t_{ref} is the refractory time of the neuron.

1.1.3 Gaussian-Random Input Current

See `part1C_ramp` in §4.1.

The average shape of the f/I curve doesn't seem to be affected, although a spread is imposed.



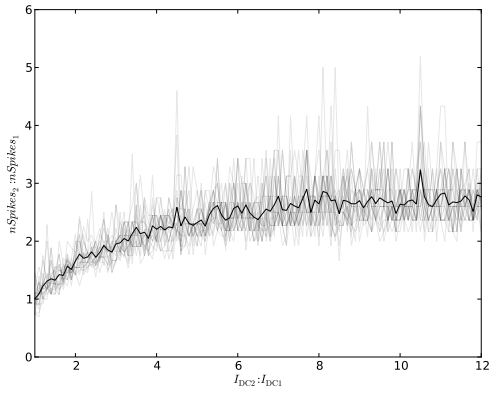
I still need to look at the effect on inter-spike times with constant I and varying I_{rand}. I'll add this non-ramping version on Saturday. But it should just be a slice of the ramping version.

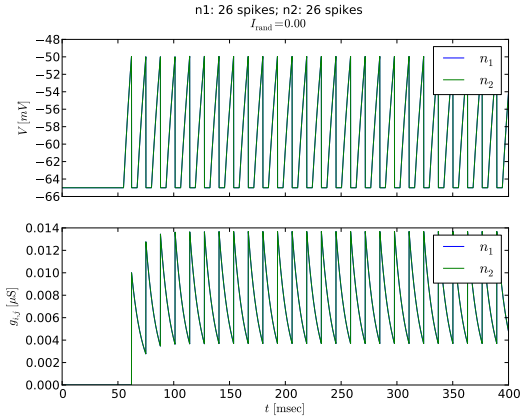
1.2 Two-I/F-Neuron Network

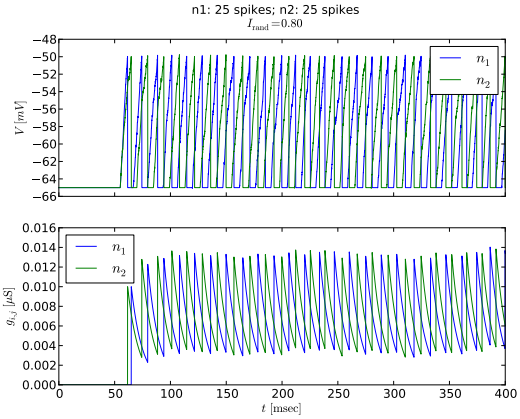
See §3.2 for the code to generate the basic two-neuron network.

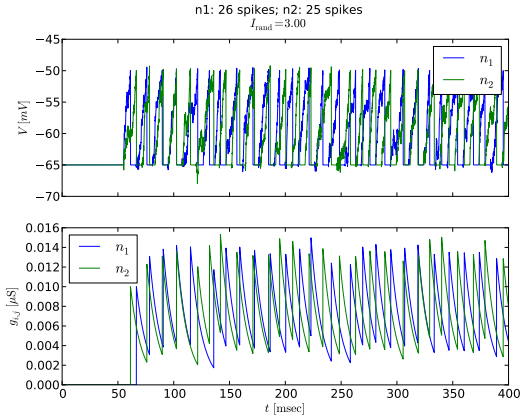
Averaged across 20 trials the ratio of neuron one's frequency to neuron two's frequency appears to be a smooth function of their ratio of input currents.

With no noise ($I_{\text{rand}} = 0$), the an unstable limit cycle is revealed in which the two neurons oscillate in synchrony. However, the smallest amount of noise ($I_{\text{rand}} = 0.8$) breaks this symmetry, and the network appears to fall into a stable limit cycle of antisymmetric firing. This degenerates as I_{rand} is further increased.





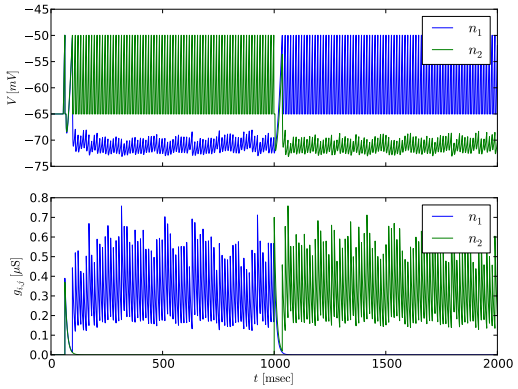




1.3 Poisson-random Synaptic Increments

With clean 1 $[nA]$ input currents to both neurons, but with synaptic increases per spike being $0.2s(m)$ $[mS/msec]$, where s is Poisson-distributed with mean m, sufficiently large values of m lead to bistability.

Here, m was set to 40. At $t = 1000\,[msec]$, The lower of the two synaptic conductances was abruptly increased to $0.7\,[\mu S]$ (analogous to a sudden synaptic increment), causing the system to switch into its other bistable state.



I should try uneven I_{DCi} (uneven saliency), and behavior as a function of m.

2 Bistability in Attractor Spaces

For future release.

3 Service Code

Code is also online at github.com/tsbertalan/501A-hw3.

3.1 integrateAndFire.py

```
import numpy as np

class ifNeuron(object):

    def __init__(self, Vleak=-65, Vreset=-65, dt=0.05, gLeak=0.02, Vthresh=-50,
                 C=0.4, t0=0, V0=-65, I=1.0, tref=5, tauSyn=10, synInc=0.2,
                 Vinhibsyn=-75, label=""):
        '''An integrate-and-fire neuron, with Euler-integration. Can be combined
        with other inhibitory neurons into a network (see ifNetwork).

        Optional Arguments
        ==================
        Vleak=-65 : scalar
            [mV] resting potential
        Vreset=-65 : scalar
            [mv] potential to which the neuron is reset after a spike.
        dt=0.05 : scalar
            [msec] timestep for Euler-integration
        gLeak=0.02 : scalar
            [uS] leak conductance
        Vthresh=-50 : scalar
            [mV] threshold potential for resetting
        C=0.4 : scalar
            [nF] membrane capacitance
        t0=0 : scalar
            [mS] start time for Euler integration
        V0=-65 : scalar
            [mV] initial membrane potential
        I=1.0 : scalar or callable
            [nA] tonic membrane current. A zero-argument, scalar-valued
            function can also be passed.
        tref=5 : scalar
            [msec] refractory time
        tauSyn=10 : scalar
            [msec] synaptic conductance decay time constant
        synInc=0.2 : scalar or callable
            [mS/msec] synaptic increment per spike received. A zero-argument,
            scalar-valued function can also be passed.
        Vinhibsyn=-75 : scalar
            [mV] Nernst potential for the inhibitory synaptic current
        label="" : string
            a label describing this neuron
        '''
```

```
        self.Vleak = Vleak
        self.Vreset = Vreset
        self.dt = dt
        self.gLeak = gLeak
        self.Vthresh = Vthresh
        self.C = C
        self.t = t0
        self.V = V0
        self.I_ = I
        self.Vhist = [V0]
        self.thist = [t0]
        self.tref = tref
        self.lastSpike = t0 - Vthresh
        self.spikeTimes = []

        self.inhibitors = []
        self.inhibWeights = np.array([])
        self.excitors = []  # unused
        self.excitWeights = np.array([])  # unused
        self.Vinhibsyn = Vinhibsyn

        self.Ghist = []  # There is no Ghist0, like there is a V0 and a t0,
        #  since the length of this vector depends on how many inhibitors are
        #  added after instantiation.

        self.tauSyn = tauSyn
        self.synInc_ = synInc

        self.newInhibWeights = self.newV = None

        self.label = label

    def synInc(self):
        '''If the supplied synInc object is a function, return its return value;
        else just return it.'''
        if callable(self.synInc_):
            return self.synInc_()
        else:
            return self.synInc_

    def addInhibitor(self, other, g0=0.0):
        '''Add another neuron to this neuron's list of inhibitors.

        Arguments
        =========
        other (neuron)
            The other neuron afferent to, and inhibiting, this neuron.

        Optional Arguments
        ==================
        g0=0.0 (float)
            The initial synaptic conductance for this pair.
        '''
        self.inhibitors.append(other)
```

```
        weights = list(self.inhibWeights)
        weights.append(g0)
        self.inhibWeights = np.array(weights)

    def Ifunc(self):
        '''If the supplied I object is a function, return its return value;
        else just return it.'''
        if callable(self.I_):
            return self.I_()
        else:
            return self.I_

    def saveNextState(self):
        '''
        We'll do Euler-integration for now.
        This does everything *except* updating the neuron's voltages and its
        vector of inhibitory connection weights. That way,
        neurons in a ifNetwork can get eachothers' last-step voltages for use in
        their own calculations.
        '''
        V = self.V
        self.t += self.dt
        self.thist.append(self.t)
        if V > self.Vthresh:  # spike
            V = self.Vreset
            self.lastSpike = self.t
            self.spikeTimes.append(self.t)
        else:
            if self.t > self.lastSpike + self.tref:  # done refracting
                V += self.dVdt() * self.dt
        # no refractory period was mentioned for the synaptic conductances.
        if len(self.inhibitors) > 0:
            self.newInhibWeights = self.inhibWeights + self.dgdt() * self.dt
        self.newV = V

    def overwriteCurrentState(self):
        '''Actually save the temporary potentials and connection weights to the
        member variables used by dVdt and dgdt. For use after all neurons in the
        network have executed saveNextState.'''
        self._setV(self.newV)
        if len(self.inhibitors) > 0:
            G = self.newInhibWeights
            self._setG(G)

    def step(self):
        '''Perform an Euler step. For use in single-neuron networks only
        (actually, just use the integrate() method).'''
        self.saveNextState()
        self.overwriteCurrentState()

    def dVdt(self):
        '''The current rate of potential change.'''
        leak = self.Ifunc() - self.gLeak * (self.V - self.Vleak)
```

```
        inhib = 0.0
        for i in range(len(self.inhibitors)):
            inhib -= self.inhibWeights[i] * (self.V - self.Vinhibsyn)
        # Show the relative contributions of leak and inhibitory components:
#         print "abs(leak):abs(inhib)", np.abs(leak) / np.abs(inhib)
        return (leak + inhib) / self.C

    def dgdt(self):
        '''The current vector of rates of synaptic conductance change.'''
        thresholds = np.array([n.Vthresh for n in self.inhibitors])
        voltages = np.array([n.V for n in self.inhibitors])
        spiking = (voltages > thresholds).astype(float)
        return -self.inhibWeights / self.tauSyn + self.synInc() * spiking

    def integrate(self, tmax):
        '''Euler-integrate a single neuron forward in time. Don't use for networks.

        Arguments
        =========
        tmax (float)
            The time (in msec) at which point integration should stop, as an absolute
            number, not a difference from the neuron's current saved time.
        '''
        while self.t < tmax:
            self.step()

    def _setV(self, V):
        '''
        Pre-allocating a numpy array for histories would be good,
        but then we'd have to keep track of our current index in that array,
        and we'd still have to extend it if we integrated past our expected
        maximum time. Still, this remains a potential source of speedup if
        any is later needed.
        '''
        self.V = V
        self.Vhist.append(V)

    def _setG(self, inhibWeights):
        """
        inhibWeights is a vector of weights. Inhibitory only for now.
        """
        self.inhibWeights = inhibWeights
        self.Ghist.append(inhibWeights)

    def _setI(self, I):
        self.I_ = I

    def getVHist(self):
        '''Returns a 1D numpy array of the potential history.'''
        return np.array(self.Vhist)

    def getTHist(self):
        '''Returns a 1D numpy array of the time history.'''
        return np.array(self.thist)
```

```
    def getGHist(self):
        '''Returns a 2D numpy array of the vector-of-synaptic-conductances history.'''
        return np.vstack(self.Ghist)


class ifNetwork(object):

    def __init__(self, neuronList):
        '''
        A simple container of several ifNeurons, to facilitate their integration.

        Arguments
        =========
        neuronList (list of ifNeuron)
            A list of neurons, which should already have called addInhibitor on
            eachother as appropriate.
            Note that the network's start time (before integration) will be taken
            from the current time of the first neuron in the supplied list.
        '''
        self.neurons = neuronList
        # Ensure that the neurons are integrated through this class rather
        #  than individually (which would be dumb):
        for n in self.neurons:
            n.step = n.integrate = None
        self.t = self.neurons[0].t  # Choice of index is arbitrary.

    def step(self):
        '''First has all neurons calculate their next state, then has them all
        save that state. This is done in two steps so they can reference
        eachothers' current state as they're calculating their next states.'''
        for n in self.neurons:
            n.saveNextState()

        for n in self.neurons:
            n.overwriteCurrentState()

    def integrate(self, tmax):
        '''This class's raison d'etre. Integrate the network of neurons forward
        in time.

        Arguments
        =========
        tmax (float)
            The time (in msec) at which point integration should stop, as an absolute
            number, not a difference from the network's current saved time.
        '''
        while self.t < tmax:
            self.step()
            self.t = self.neurons[0].t


def waitingTimes(tlist):
    '''Returns an ndarray vector 1 shorter than that given, containing
```

```
the differences between consecutive entries.'''
if len(tlist) > 0:
    T = np.array(tlist)
    return T[1:] - T[:-1]
else:
    return np.nan
```

3.2 twoNet.py

```
import numpy as np
from integrateAndFire import ifNeuron, ifNetwork

def twoNet(IDC1, IDC2, Irand, tmax=500, synInc=0.2):
    '''
    A network of two mutually-inhibiting integrate-and-fire neurons.
    Immediately performs integration upon construction. This is convenient.
    '''
    def functor_In(IDCn):
        def In():
            if Irand != 0:
                return np.random.normal(loc=IDCn, scale=Irand)
            else:
                return IDCn
        return In
    I1 = functor_In(IDC1)
    I2 = functor_In(IDC2)

    n1 = ifNeuron(I=I1, label=r"$n_1$", synInc=synInc)
    n2 = ifNeuron(I=I2, label=r"$n_2$", synInc=synInc)
    n1.addInhibitor(n2)
    n2.addInhibitor(n1)

    net = ifNetwork([n1, n2])
    net.integrate(tmax)
    return net
```

4 User Code

4.1 hw3-1.py

```
import matplotlib.pyplot as plt
import numpy as np

from integrateAndFire import ifNeuron, ifNetwork, waitingTimes
from twoNet import twoNet

def fa(**kwargs):
    '''Create a figure with a single axis. kwargs are passed to figure().'''
    f = plt.figure(**kwargs)
    a = f.add_subplot(1, 1, 1)
    return f, a


def part1A(tmax=1000):
    '''
    Clearly I=0.3 is a bifurcation point for the default parameters
    below this threshold, the steady-state I value is a balance if inward
    current, I, and outward (leak) current. Or perhaps I have my directions
    confused.
    '''
    I = 0.290
    d = ifNeuron(I=I, label=r"$I=%.3f$" % I)
    I = 0.300
    a = ifNeuron(I=I, label=r"$I=%.3f$" % I)
    I = 0.301
    b = ifNeuron(I=I, label=r"$I=%.3f$" % I)
    I = 0.305
    c = ifNeuron(I=I, label=r"$I=%.3f$" % I)
    for n in d, a, b, c:
        n.integrate(tmax)
        for i in range(len(n.Vhist)):
            if n.Vhist[i] > n.Vthresh:
                n.Vhist[i] = 10
    fig, ax = showStateHistories([d, a, b, c], showg=False)
    ax.legend(loc="best")
    save(fig, '501a-hw3-part1A', enum=False, verbose=True)


def part1B():

    n = ifNeuron(dt=0.1)
    Ihist = [0]
    thist_I = [0]

    def Ioft(t):
        return t * 4.0 / 12500.0
    def rampI():
        I = Ioft(n.t)
        Ihist.append(I)
        thist_I.append(n.t)
```

```
        return I
    n.I_ = rampI

    fig, ax = fa()

    n.integrate(tmax=12500)
    spikeDelays = []

    Vhist = n.getVHist()
    thist = n.getTHist()

    Vhist[Vhist > n.Vthresh] = 10

    spikeDelays = waitingTimes(np.array(n.spikeTimes))
    delayTimes = np.array(n.spikeTimes[1:])
    currents = Ioft(delayTimes)

    ax.scatter(currents, 1 / spikeDelays * 1000, label='simulated')
    def f(I):
        tcurve = -n.C/n.gLeak * np.log(1 - n.gLeak/I * (n.Vthresh - n.Vleak))
        return 1 / (n.tref + tcurve)
    ax.plot(currents, f(currents) * 1000, label='calculated', color="red")

    ax.legend(loc='best')
    ax.set_ylabel(r'spiking frequency $[\mathrm{Hz}]$')
    ax.set_xlabel(r'applied current $[\mathrm{nA}]$')
    save(fig, "part1B", enum=False, verbose=True)


def part1C_ramp():
    Irands = [4, 8, 10, 0, 1.0, 2.0]
    fig = plt.figure(figsize=(11,8.5))
    A = [fig.add_subplot(2, 3, i+1) for i in range(6)]
    for i, Irand in enumerate(Irands):
        ax = A[i]
        n = ifNeuron(dt=0.1)
        Ihist = [0]
        thist_I = [0]

        def Ioft(t):
            return t * 4.0 / 12500.0
        def Ifunc():
            I = Ioft(n.t)
            Ihist.append(I)
            thist_I.append(n.t)
            # np.random.normal(), without args, returns a single, standard-normal, number. So,
            return I + Irand * np.random.normal()
            # ... is equivalent to np.random.normal(loc=I, scale=Irand)
        n.I_ = Ifunc

        n.integrate(tmax=12500)
        spikeDelays = []

        Vhist = n.getVHist()
```

```
        thist = n.getTHist()

        Vhist[Vhist > n.Vthresh] = 10

        spikeDelays = np.array(n.spikeTimes[1:]) - np.array(n.spikeTimes[:-1])
        delayTimes = np.array(n.spikeTimes[1:])
        currents = Ioft(delayTimes)

        ax.scatter(currents, 1 / spikeDelays * 1000, label='simulated')

        def f(I):
            tcurve = -n.C/n.gLeak * np.log(1 - n.gLeak/I * (n.Vthresh - n.Vleak))
            return 1 / (n.tref + tcurve)
        ax.plot(currents, f(currents) * 1000, label='calculated', color="red")

        ax.set_title(r'$I_\mathrm{rand}=%.1f$ $[\mathrm{nA}]$' % Irand)

        ax.set_ylim((0, 200))
        ax.set_xlim((0, 4))

        if i is 3:
            ax.legend(loc='best')
            ax.set_ylabel(r'spiking frequency $[\mathrm{Hz}]$')
            ax.set_xlabel(r'applied current $[\mathrm{nA}]$')
        else:
            ax.set_yticklabels([])
            ax.set_xticklabels([])

        save(fig, 'part1C_ramp', enum=False)


def part2_timeCourse(Irand=6, tmax=200, IDC2=1):

    # Even base-currents
    net = twoNet(1, IDC2, Irand, tmax=tmax)
    n1, n2 = net.neurons

    V1 = n1.getVHist()
    t  = n1.getTHist()
    V2 = n2.getVHist()

    fig, axes = showStateHistories([n1, n2])

    fig.suptitle(  "n1: %d spikes" % len(n1.spikeTimes) + '; '
                 + "n2: %d spikes" % len(n2.spikeTimes) + '\n'
                 + r"$I_\mathrm{rand}=%.2f$" % Irand)
    fig.subplots_adjust(top=0.9)
    save(fig, "part2_timecourse-Irand%.2f" % Irand, enum=False, verbose=True)


def part2_ratioData(ntrials=20, IDC1=1.0, IDC2max=12.0, dI=.1, Irand=1.0):
    '''Current-ratio / spike-ratio relationship.'''
    from time import time
    start = time()
```

```
    curRats_ = []
    spiRats_ = []
    delays = []
    for trial in range(ntrials):
        print
        curTime = time() - start
        delays.append(curTime)
        if len(delays) > 1:
            remain = np.mean(waitingTimes(delays)) * (ntrials - trial)
        else:
            remain = np.nan
        print "trial", trial+1, "of", ntrials, ",",
        print "%.1f min remaining" % (remain / 60.),
        curRats = []
        spiRats = []
        for IDC2 in np.arange(IDC1, IDC2max+dI, dI):
            net = twoNet(IDC1, IDC2, Irand, tmax=200)
            n1, n2 = net.neurons
            n1spikes = len(n1.spikeTimes)
            n2spikes = len(n2.spikeTimes)
            curRats.append(IDC2 / IDC1)
            if abs(IDC2 % 1.0) < dI:
                print int(IDC2),# ":", n1spikes, n2spikes
            spiRats.append(n2spikes / float(n1spikes))
        curRats_.append(np.array(curRats))
        spiRats_.append(np.array(spiRats))

    curRats = np.vstack(curRats_)
    spiRats = np.vstack(spiRats_)
    fname = "driveRatios-tri_%d-IDC1_%.1f-IDC2max_%.1f-dI_%.1f-Irand_%.1f.npz" % (
                                            ntrials, IDC1, IDC2max, dI, Irand)
    np.savez_compressed(fname, curRats=curRats, spiRats=spiRats,
                        curRats_=curRats_, spiRats_=spiRats_,
                        ntrials=ntrials, IDC1=IDC1, IDC2max=IDC2max, dI=dI)
    print
    print 'saving', fname
    return fname

def part2_ratioShow(fname):
    '''Show the saved data from part2_ratioData.'''
    fig, ax = fa()

    data = np.load(fname)
    curRats = data['curRats']
    spiRats = data['spiRats']
    curRats_ = data['curRats_']
    spiRats_ = data['spiRats_']
    IDC1 = data['IDC1']
    IDC2max = data['IDC2max']

    for c, s in zip(curRats_, spiRats_):
        ax.plot(c, s, color=(0,0,0,.01)) # black; 1% opacity
    # average across four trials
```

```
    curRats = np.mean(curRats, axis=0)
    window_len = 32
    curRats = smooth(curRats, window_len=window_len)
    spiRats = np.mean(spiRats, axis=0)
    spiRats = smooth(spiRats, window_len=window_len)

    ax.set_xlim((IDC1, IDC2max))
    ax.plot(curRats, spiRats, color="black")
    ax.set_ylabel(r"$nSpikes_2:nSpikes_1$")
    ax.set_xlabel(r"$I_\mathrm{DC2}:I_\mathrm{DC1}$")
    print curRats.shape
    print curRats[::window_len].shape
#     for x in curRats[::window_len]:
#         ax.axvline(x=x)
    ax.set_xscale('log')
    save(fig, fname, verbose=True, enum=False)


def part3():
    '''Input current is not noisy, but syanptic conductances are Poisson-
    distributed.'''
    m = 40
    def synInc():
        return 0.2 * np.random.poisson(lam=m)
    net = twoNet(1, 1, 0, synInc=synInc, tmax=1000)
    n1, n2 = net.neurons

    # cause the lower-g neuron to have ... a higher g.
    if n1.inhibWeights[0] < n2.inhibWeights[0]:
        n1.inhibWeights[0] = 0.7
    else:
        n2.inhibWeights[0] = 0.7

    net.integrate(tmax=2000)

    fig, axes = showStateHistories([n1, n2])
    save(fig, 'part3', enum=False, verbose=True)


def showStateHistories(listOfNeurons, showg=True):
    fig = plt.figure()
    if showg:
        ax1 = fig.add_subplot(2, 1, 1)
        ax2 = fig.add_subplot(2, 1, 2)
    else:
        ax1 = fig.add_subplot(1, 1, 1)
    for n in listOfNeurons:
        T = n.getTHist()
        ax1.plot(T, n.getVHist(), label=n.label)
        if showg:
            ax2.plot(T[1:], n.getGHist()[:, 0], label=n.label)

    xlabel = r"$t$ $[\mathrm{msec}]$"
```

```
    ax1.set_ylabel(r"$V$ $[mV]$")
    ax1.legend(loc="best")
    ax1.set_xlim((T.min(), T.max()))

    if showg:
        ax1.set_xticklabels([])
        ax2.set_ylabel(r"$g_{i,j}$ $[\mathrm{\mu S}]$")
        ax2.legend(loc="best")
        ax2.set_xlim((T.min(), T.max()))
        ax2.set_xlabel(xlabel)
    else:
        ax1.set_xlabel(xlabel)

    if showg:
        return fig, [ax1, ax2]
    else:
        return fig, ax1


def save(fig, filename, enum=True, ext=".pdf", verbose=False):
    if enum:
        from time import time
        filename += "-%d" % time()
    filename += ext
    if verbose:
        print "saving", filename
    fig.savefig(filename)


def smooth(x,window_len=11,window='hanning'):
        if x.ndim != 1:
                raise ValueError, "smooth only accepts 1 dimension arrays."
        if x.size < window_len:
                raise ValueError, "Input vector needs to be bigger than window size."
        if window_len<3:
                return x
        if not window in ['flat', 'hanning', 'hamming', 'bartlett', 'blackman']:
                raise ValueError, "Window is on of 'flat', 'hanning', 'hamming', 'bartlett', 'blackman'
        s=np.r_[2*x[0]-x[window_len-1::-1],x,2*x[-1]-x[-1:-window_len:-1]]
        if window == 'flat': #moving average
                w=numpy.ones(window_len,'d')
        else:
                w=eval('np.'+window+'(window_len)')
        y=np.convolve(w/w.sum(),s,mode='same')
        return y[window_len:-window_len+1]


if __name__ == "__main__":
    part1A()
    part1B()
    part1C_ramp()
    part2_timeCourse(Irand=0.0, tmax=400)
    part2_timeCourse(Irand=0.8, tmax=400)
    part2_timeCourse(Irand=3.0, tmax=400)
```

```
#     fname = part2_ratioData(ntrials=2, IDC2max=2.0, dI=0.25)
#     fname = part2_ratioData(ntrials=44, IDC2max=24.0, dI=0.025)
#     fname = "driveRatios-tri20-IDC11.000000-IDC2max12.000000-dI0.100000.npz"
    fname = "driveRatios-tri_44-IDC1_1.0-IDC2max_24.0-dI_0.0-Irand_1.0.npz"
    part2_ratioShow(fname)
    part3()

    plt.show()
```