SICP_firstTerm_positive

Unknown Author

May 6, 2014

1 Development scheme

1.1 Introduction

We will input a noisy single-molecule trace of, for instance, the activity of a molecular motor protein. These traces are very often step-like, with the size and duration of the steps containing important information regarding the enzymatic cycle of the machine. This information, unfortunately, is masked by noise that is intrinsic to the experiment and comes from many different sources (the Brownian motion of the particle of interest, for instance, etc). Our task is to unveil the molecular motor's behavior despite the underlying noise.

In the case of optical tweezers experiments conducted in passive mode, the molecular motor actively pulls itself into regions of higher or lower external forces (increase/decrease depends on the geometry of the experiment). In this case the constrained Brownian motion of the motor contributes noise that is not stationary, but changes as a function of the external force. Many other related, but seemingly different types of experiments similarly have non-stationary noise. Here we implement an algorithm to specifically address this kind of scenario. Assumptions: 1. Trace is fundamentally step-like. 2. The noise is Gaussian and independently-distributed 3. The noise is NOT stationary, but has a width that changes throughout the duration of the experiment.

1.2 Program Schematic

The program can be nicely organized by following the order of events required to do the fit.

Input data

Trace: * time * force * position

This should be populated with an input file.

Slice data into force bins

Slices: * start and end of each "slice", done by force interval (orded by index…taken from trace) * force, mean force of each slice interval * params (definition: respective ν and S_o for each slice)

The params should be populated with input file that contains entire range of possible parameters.

Fit: * start and end of each dwell (by index…taken from trace) * position of each dwell * force of each dwell (to ID which slice and therefore which $\{\nu, S_o\}$) * slice ID of each dwell (will sort on this to do SICP calculation)

This is iterated, we will converge on optimal fit by minimizing SICP (separate class, see below). Many objects of this class.

SICP: * $\hat{\sigma}_i^2$, variance of the data points attached to each i^{th} dwell * n_i, number of data points of each i^{th} dwell * $\hat{\sigma}^2$ overall variance of the data points in entire slice * number of steps per slice, d_k * SICP for each slice and * sum of SICPs to characterize entire fit

This is iterated, we will converge on the optimal fit by minimizing SICP. Many objects of this class.

NOTE: I just realized that each slice has to have one dwell start. Our initial proposed fit will have as many dwells as there are slices. If we don't do this, the SICP becomes undefined for the slices that don't yet have a dwell.

After we do this, we can just proceed as normal, adding one step at a time (checking all possibilities during each addition and selecting the optimal one).

1.3 Inputs

Two files: * trace, which has three floating point values separated by spaces: time, force, position * slice parameters, which has four space-separated floats: start force, end force, ν, S_o

1.4 Program function

1. Input the trace into the trace object (see Trace Class for format)
2. Slice up the trace into the slice object (see Slice Class for format)
3. Generate initial fit (one dwell per slice, optimize location by minimizing SICP)
4. Fit additional steps until SICP is converged
    - Add additional steps one at a time, selecting best location by minimizing SICP.
    - Keep track of the slices, and factor in slices for SICP calculation
5. Output the fit

---

2 Write Program

2.1 Import Data

Trace data is contained in external txt file, space delimited. Format is: time force position

In [1]:
```python
# import data
def data_point(time, force, position):
    '''
    Inputs: a data point will have a time, a position, and a force (floats)
    Returns a dictionary with these values
    '''
    # each data point is a dict having time, force, and position
    return {'t': time,
            'f': force,
            'p': position}

def load_trace(trace_filename):
    '''
    Opens datafile, loads in each row, and dumps time, position, and force into a trac
    --Requires data_point function
    --Trace datastructure is a list of dictionaries. Each dictionary entry is a datapo
    Returns trace datastructure
    '''
    dataFile = open(trace_filename, 'r')
    trace = []
    for row in dataFile:
        d = [float(f) for f in row.strip().split(' ')] # Have to cast each string valu
        trace.append(data_point(*d))
    dataFile.close()
    # data is now dumped in trace list (each list item is a dictionary)
    return trace
```

In [2]:
```
# print first and last couple list items to confirm it's set up correctly
trace = load_trace(trace_filename='./examp_trace_3.txt')

print "Len:", len(trace)
print "First:", trace[0]
print "Last:", trace[-1]
```

```
Len: 1000
First: {'p': 5.9311520487614615, 't': 0.0, 'f': 0.29655760243807305}
Last: {'p': 44.89004310946699, 't': 999.0, 'f': 2.2445021554733495}
```

2.2 Slice Data

Import list of nu and So for each possible force slice

In [3]:
```
# import nu, So for each slice
def slice_params(startForce, endForce, nu, So): # a slice of data will have a start/en
    '''
    Inputs SICP parameters (nu, So) for all force intervals, bounded by startForce and
    Returns dictionary of this information
    '''
    return {'sF': startForce, 'eF': endForce, 'nu': nu, 'So': So} # each possible slic

sliceFile = open('./examp_params.txt', 'r') # this is the input file that contains the
params = []; tmp = [];
for row in sliceFile:
    tmp.append(row.strip().split(' '))
[params.append(slice_params(float(tmp[i][0]),float(tmp[i][1]),float(tmp[i][2]),float(t
del tmp
sliceFile.close()
# possible parameters are now dumped in params list (each list item is a dictionary)
```

In [4]:
```
print params
```

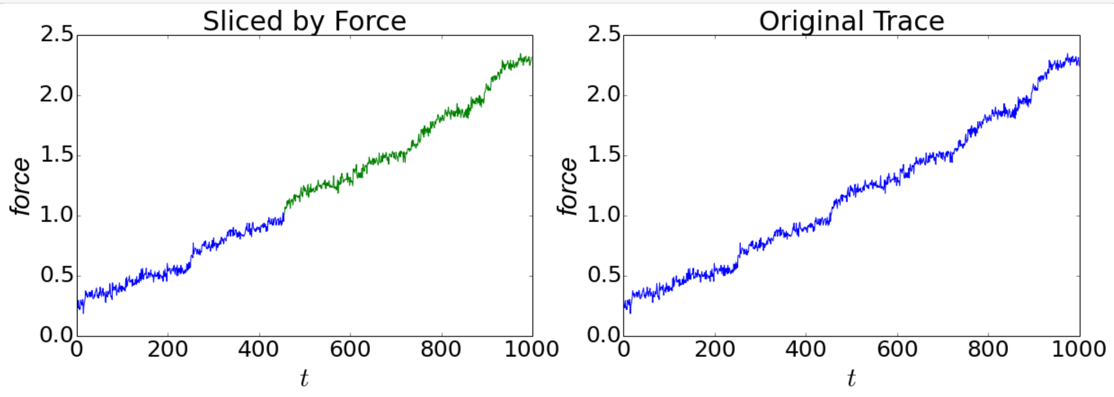
```
[{'eF': 1.0, 'So': 0.0, 'sF': 0.0, 'nu': 0.0}, {'eF': 2.0, 'So': 0.0,
'sF': 1.0, 'nu': 0.0}, {'eF': 3.0, 'So': 0.0, 'sF': 2.0, 'nu': 0.0},
{'eF': 4.0, 'So': 0.0, 'sF': 3.0, 'nu': 0.0}, {'eF': 5.0, 'So': 0.0,
'sF': 4.0, 'nu': 0.0}, {'eF': 6.0, 'So': 0.0, 'sF': 5.0, 'nu': 0.0},
{'eF': 7.0, 'So': 0.0, 'sF': 6.0, 'nu': 0.0}, {'eF': 8.0, 'So': 0.0,
'sF': 7.0, 'nu': 0.0}, {'eF': 9.0, 'So': 0.0, 'sF': 8.0, 'nu': 0.0},
{'eF': 10.0, 'So': 0.0, 'sF': 9.0, 'nu': 0.0}]
```

Slice the data according to force by recording start/end indices of each slice. Attach approprate (nu, So) to each slice

In [5]:
```
stF=sorted(trace, key=lambda x: x['f'])

def slice_indexing(startForce, startIndex, endForce, endIndex, nu, So): # slice the da
    return {'sF': startForce,
            'sI': startIndex,
            'eF': endForce,
            'eI': endIndex,
            'nu': nu,
            'So': So}

def find_nearest(array, value):
    index=argmin([abs(array[i]-value) for i in range(0,len(array))])
    return array[index]

slices = [];
for forces in range(int(round(stF[0]['f'])),int(round(stF[len(stF)-1]['f']))): # very 
    # sometimes the starting force will only be just below a certain force (i.e. 4.7 w
    # to include a 4 to 5 pN interval there would hardly be any data in that interval.
    # where it will be only just above a certain force (i.e. 10.2, in this case you wo
    # the int(round(... in the for loop above will specifically treat these possible s

    # the if/else statements correct the indices of the starting and end force interva
    # force interval, we set the indices accordingly
    if forces == int(round(stF[0]['f'])):
        sI=0;
    else:
        sI=[trace[i]['f'] for i in range(0,len(trace))].index(find_nearest([trace[i]['
    if forces+1 == int(round(stF[len(stF)-1]['f'])):
        eI = len(trace)-1
    else:
        eI=[trace[i]['f'] for i in range(0,len(trace))].index(find_nearest([trace[i]['
    sF=forces; eF=forces+1;
    nu=params[[params[i]['sF'] for i in range(0,len(params))].index(sF)]['nu'];
    So=params[[params[i]['sF'] for i in range(0,len(params))].index(sF)]['So'];
    slices.append(slice_indexing(forces, sI, forces+1, eI, nu, So))
del stF
```

Confirm that slicing is working graphically

In [6]:
```
figure(figsize=(20,6))
subplot(1,2,1)
for s in range(0,len(slices)): # iterate over each slice, s indexes slices
    force = [trace[i]['f'] for i in range(slices[s]['sI'], slices[s]['eI'])]
    time = [trace[i]['t'] for i in range(slices[s]['sI'], slices[s]['eI'])]
    plot(time, force); title('Sliced by Force', fontsize=30);
    xlabel('$t$', fontsize=30); ylabel('force', style='italic', fontsize=30); tick_par
subplot(1,2,2)
plot([trace[i]['f'] for i in range(0,len(trace))]); title('Original Trace',fontsize=30
xlabel('$t$', fontsize=30); ylabel('force', style='italic', fontsize=30); tick_params(
```



2.3 Optimal First Fit

Optimal single dwell of each slice is at mean of data

In [7]:
```
def dwell_params(startIndex, endIndex, positionLoc, forceLoc, sliceLoc): # when dwell 
    return {'sI': startIndex, 'eI': endIndex, 'p': positionLoc, 'f': forceLoc, 'slice'
def slice_initial_sicp(dwellSigSq, dwellNumPts, numSteps, whichSlice): # list, list, l
    nu = slices[whichSlice]['nu']
    So = slices[whichSlice]['So']
    sicp = (numSteps+1)*(log(2*pi)+1) + log(dwellNumPts) + (dwellNumPts+nu-(numSteps+1
    return sicp
dwellList=[]; sicpSlice=[];
for s in range(0,len(slices)): # iterate over each slice, s indexes list
    pos=[trace[i]['p'] for i in range(slices[s]['sI'], slices[s]['eI'])] # get positio
    force=[trace[i]['f'] for i in range(slices[s]['sI'], slices[s]['eI'])] # get force
    startIndex=slices[s]['sI']
    endIndex=slices[s]['eI']
    positionLoc=mean(pos)
    forceLoc=mean(force)
    sliceLoc=s
    dwellSigSq=var(pos); dwellNumPts=(endIndex-startIndex); numSteps=0;
    sicpSlice.append(slice_initial_sicp(dwellSigSq, dwellNumPts, numSteps, s))
    dwellList.append(dwell_params(startIndex, endIndex, positionLoc, forceLoc, sliceLo
del pos; del force; del startIndex; del endIndex; del positionLoc; del forceLoc; del s
# clear variable I won't use anymore
```

Recap of data structures: 1. dwellList: list of dictionaries. list index IDs dwell, dictionary contains parameters required to fully specify particular dwell 2. sicpSlice: list of sicp calcs for each slice

2.4 Add steps, stopping once overall sicp converges

Scheme: 1. Iterate through all possible locations for new trial step 2. *Replace* existing dwell surrounding trial location with two new dwells 3. Calculate sicp for all possible trial locations, select location with lowest sicp 4. If new sicp

In [8]:
```
def calc_sicp(proposedDwellList, sliceID): # list, list, list, int
    '''

    '''
    # we need to generate list of dwells in given slice
    dwellsInSlice = [];
    for dwell in range(len(proposedDwellList)):
        if proposedDwellList[dwell]['slice'] == sliceID:
            dwellsInSlice.append(proposedDwellList[dwell])
    # generate list of position lists bounded by each dwell
    # generate list of force lists bounded by each dwell
    posDwell = []; forceDwell = []; numPtsDwell = []; dwellSigSq = [];
    for dwell in range(len(dwellsInSlice)):
        posDwell.append([trace[i]['p'] for i in range(dwellsInSlice[dwell]['sI'],dwell
        forceDwell.append([trace[i]['f'] for i in range(dwellsInSlice[dwell]['sI'],dwe
        numPtsDwell.append(abs(dwellsInSlice[dwell]['eI'] - dwellsInSlice[dwell]['sI']
        dwellSigSq.append(var(posDwell[dwell]))
    # get (nu, So) for this particular slice
    # print dwellsInSlice
    n = (dwellsInSlice[-1]['eI']-dwellsInSlice[0]['sI'])
    overallSigSq = sum([a*b for a,b in zip(numPtsDwell,dwellSigSq)])/n
    nu = slices[sliceID]['nu']
    So = slices[sliceID]['So']
    d = len(dwellsInSlice)
    # calculate sicp for the slice
    sicp = 0
    # add all components to sicp except for the # dp's per dwell in slice
    sicp += d*(log(2*pi)+1) + (n+nu-d-1)*log(n*overallSigSq + So) - (n+nu-d-3)*log(n+n
    for dwell in range(len(dwellsInSlice)):
        sicp += log(numPtsDwell[dwell]) # now add dp's per dwell component
    return sicp # list of sicp's for each slice
```

In [9]:
```
def trial_step(sicpList, dwellList, l): # input the list of dwells
    '''

    '''
    for dwell in range(0,len(dwellList)): # loop through all dwells
        # print 'sIndex ' + str(dwellList[dwell]['sI']) + ' list item ' + str(l) + ' e
        if dwellList[dwell]['sI'] < l < dwellList[dwell]['eI']: # use if statement to
```

```
                # split this dwell by adding step, calculate updated sicp
                # then remove the original dwell
                leftPos=[trace[i]['p'] for i in range(dwellList[dwell]['sI'], l)] # get po
                rightPos=[trace[i]['p'] for i in range(l, dwellList[dwell]['eI'])] # get p
                leftForce=[trace[i]['f'] for i in range(dwellList[dwell]['sI'], l)] # get
                rightForce=[trace[i]['f'] for i in range(l, dwellList[dwell]['eI'])] # get
                sliceID = dwellList[dwell]['slice'] # ID slice location of dwell we're spl
                proposedDwellList = list(dwellList)
                # print 'l value: ' + str(l) + '...' + 'last dwell list position: ' + str(
                proposedDwellList.pop(dwell) # error here
                proposedDwellList.insert(dwell,{'sI': dwellList[dwell]['sI'],'eI': l,'p':
                proposedDwellList.insert(dwell+1,{'sI': l,'eI': dwellList[dwell]['eI'],'p'
                sicpList.pop(sliceID) # remove the sicp from the slice we added a step in,
                break # once you find the dwell to split and have removed the previous sic
    sicp = calc_sicp(proposedDwellList, sliceID)
    sicpList.insert(sliceID,sicp)
    return {'dwell list': proposedDwellList, 'sicp slice list': sicpList, 'sicp total'
```

In [10]:
```
def add_step(prevDwellList,sliceSicpList): # finds optimal location for next step
    '''

    '''
    existingDwellIndices = [prevDwellList[i]['sI'] for i in range(0,len(prevDwellList)
    output = []; trialStepSicp = []; trialDwellList = [];
    for l in range(0,len(trace)-1): # for every point in the trace,
        if l not in existingDwellIndices: # if it's already the location of a step, tr
            trialStepSicp = list(sliceSicpList)
            trialDwellList = list(prevDwellList)
            output.append(trial_step(trialStepSicp, trialDwellList, l)) # unfinished h
    trialLocs=sorted(output, key=lambda x: x['sicp total']) # sort trial step location
    return trialLocs[0] # return the step location that minimizes sicp
```

Add steps until SICP converges

In [11]:
```
print "SICP values: "
tmp = add_step(dwellList,sicpSlice)
print "old: " + str(sum(sicpSlice)) + " new: " + str(sum(tmp['sicp slice list']))
cnt = 1; sicp=[]
while(tmp['sicp total'] < sum(sicpSlice)):
    dwellList = tmp['dwell list']
    sicpSlice = tmp['sicp slice list']
    tmp = add_step(dwellList,sicpSlice)
    #print "old: " + str(sum(sicpSlice)) + " new: " + str(sum(tmp['sicp slice list']))
    sicp.append([cnt, sum(sicpSlice)])
    cnt+=1
```

```
SICP values:
old: 3540.09116083 new: 2745.66366096
```

Let's take the final fit and plot it on top of the trace.

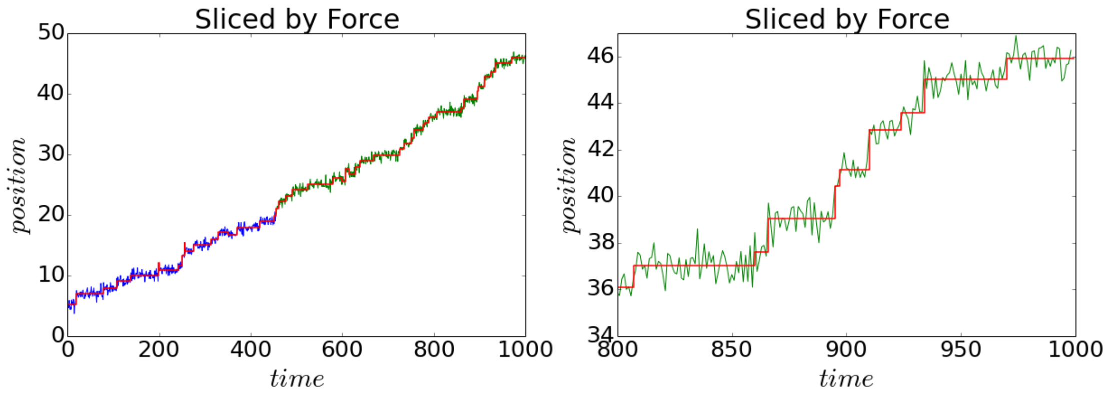
In [12]:
```
# these values are the indices of the start and end points of each dwell
# for a real trace they would have to be converted to time
a=[dwellList[i]['sI'] for i in range(len(dwellList))]
b=[dwellList[i]['eI'] for i in range(len(dwellList))]
x = [item for sublist in zip(a,b) for item in sublist]
```

In [13]:
```
# these values are the positions of each dwell
a=[dwellList[i]['p'] for i in range(len(dwellList))]
y = [item for sublist in zip(a,a) for item in sublist]
```

This is the resulting fit (in cyan) with the first term positive

In [14]:
```
figure(figsize=(20,6))
subplot(1,2,1)
for s in range(0,len(slices)): # iterate over each slice, s indexes slices
    force = [trace[i]['p'] for i in range(slices[s]['sI'], slices[s]['eI'])]
    time = [trace[i]['t'] for i in range(slices[s]['sI'], slices[s]['eI'])]
    plot(time, force); title('Sliced by Force', fontsize=30);
    xlabel('$time$', fontsize=30); ylabel('$position$', fontsize=30); tick_params(labe
plot(x,y,linewidth=1.5);
subplot(1,2,2)
for s in range(0,len(slices)): # iterate over each slice, s indexes slices
    force = [trace[i]['p'] for i in range(slices[s]['sI'], slices[s]['eI'])]
    time = [trace[i]['t'] for i in range(slices[s]['sI'], slices[s]['eI'])]
    plot(time, force); title('Sliced by Force', fontsize=30);
    xlabel('$time$', fontsize=30); ylabel('$position$', fontsize=30); tick_params(labe
plot(x,y,linewidth=1.5);
xlim(800,1000); ylim(34,47);
#xlim(0,400); ylim(0,40)
#savefig('first_term_positive.png')
(34, 47)
```

Out [14]:

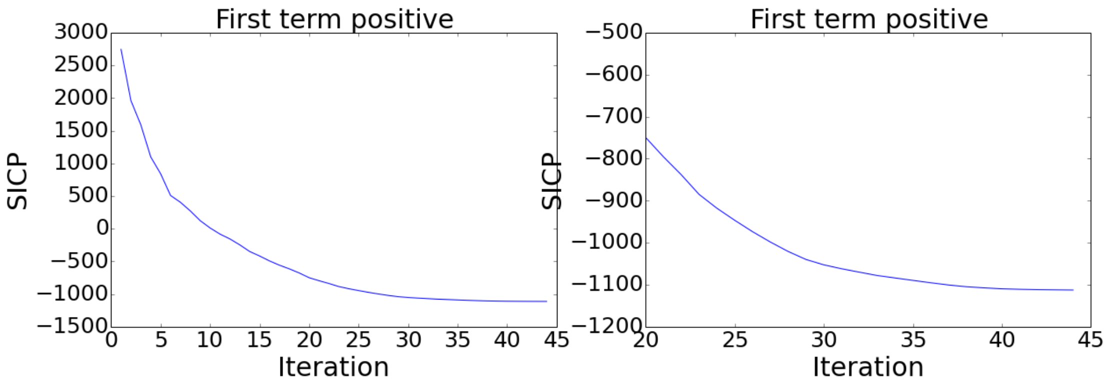


Let's look at the SICP convergence

In [16]:

```
figure(figsize=(20,6))
subplot(1,2,1)
plot([sicp[i][0] for i in range(len(sicp))],[sicp[i][1] for i in range(len(sicp))])
tick_params(labelsize=25)
title("First term positive", fontsize=30)
xlabel("Iteration", fontsize=30)
ylabel("SICP", fontsize=30)
subplot(1,2,2)
plot([sicp[i][0] for i in range(len(sicp))],[sicp[i][1] for i in range(len(sicp))])
xlim(20,45); ylim(-1200,-500)
tick_params(labelsize=25)
title("First term positive", fontsize=30)
xlabel("Iteration", fontsize=30)
ylabel("SICP", fontsize=30)
#savefig('first_term_positive_SICPvsIteration.png')
<matplotlib.text.Text at 0x7f95d227dc50>
```

Out [16]:



In [15]:

```

```