Nonlinear Time Series Analysis solo

Luke Fairbanks

March 2023

To keep things simple and brief, I'll give a condensed description of what I contributed to the project as well as a few things I'm aware my partner did although I can't speak to everything they did since we took different pathways around the same problem.

My contributions were primarily producing code for the 'YugeAnalysis' function which takes in time series data, conducts a average mutual information analysis to find the embedding delay τ, conducts a false nearest neighbors analysis to find the embedding dimension D_E, and run the data with those parameters through a set of metrics which have been pulled from the python library 'Nolds'. These tasks took up most of my project time, but I was involved somewhat in procuring useful literature at the beginning, producing some test datasets and rudimentary methods when we were still in the 'scattershot' phase of the project, and using latex to write up a decent number of sections in the final report.

The specific sections which I was the main contributor for were: time delay embedding, logistic map & Lorenz '63 as testbeds for chaos, average mutual information & false nearest neighbor algorithms, time delay optimization algorithms, python 'Nolds' time series analysis functions, ensemble analysis algorithms, time series examples and analysis results comparison, colpitts 'bigAnalysis' results, tisean software, disclaimer.

The figures I contributes were 1 through 61 as well as 6 later on which were the results from the 'bigAnalysis' on the two colpitts systems. I should say at least one of my figures I attributed to Daniel though, the one which contained his code to produce Lorenz '63 data.

I think we worked together well and we helped each other get much further than if we had both worked alone. The specific areas where Daniel helped me out a ton were with respect to data production for the final set of analyses, specifically Lorenz '63 and Colpitts simulated data, as well as running the physical colpitts circuit. I could have figured that one out, but he was invaluable since he was already aware of some equipment which could be used to run it, and he went ahead to produce the data sets without much help from me. I also generally took guidance from Daniel since he was the one who had the impetus to focus on nonlienar time series analysis, and he seemed more well read with respect to our source literature and the other class he was taking which was relevant. I was mostly focused on implementation and programming of the various tools for the 'YugeAnalysis'.

To show rather than tell, here is the set of all finalized functions which I programmed, note that many can be found better contextualized within the project report itself, just displaying here in a simple manner as a collection.

pieces of code for AMI & FNN (figures 1-12)

Here's code to produce time delay embedding plots, using some of the code provided above as functions (figures 13, 14)

Then the time delay optimization codes and 'derivative' codes (figures 15-24)

Finally the codes which combine everything for a full analysis of AMI, FNN, and the 'nolds' functions (figures 25-27)

Also as an aside here is a piece of code produced to conduct an intermediary analysis which was not included in the main paper (figure 28)

Also for good measure here are the relevant imports (figure 29)

For results from these algorithms, I'll point to the main paper to keep this one brief.

Thank you for your time

```python
#time delay vector maker (function)
#takes in a one dimensional vector, implements a time delay via a tau = step*multiplier = n*T where T is a multiple of sampling rate timing
#returns a time delayed vector for use in time delay embedding

def TDvec(vec, step, l, multiplier): #vec = time series data vector, step = 'shift' value, L = length of data window cut, multiplier = scaling for step shifts
    delay = step*multiplier #establish the tau = delay
    new = vec[delay:(l + delay)] #cut a 'window' out of data
    return new
```

Figure 1: returns time delayed vector

```python
#returns an array where each column of the array is a sample from the time series at a different time delay tau
def TDset(X, l, num, mult): #X = time series data vector, L = length of data window cut, num = number of columns or TD shifts, mult = TD shift scaling
    shape = (l,num)
    arr = np.empty(shape)
    for i in range(num):
        arr[:, i] = TDvec(X, i, l, mult) #call time delay vector generator
    return arr
```

Figure 2: returns set of time delayed vectors with different τ values

```python
#returns a set of histograms where each histogram corresponds to a vector of the original array
def histAll(data, binNum): #data = input array, binNum = number of different bins the data should be sorted into/'resolution' of histogram
    dataShape = np.shape(data)
    histSetShape = (binNum, dataShape[1]) #num of rows is num of hist bins, num of columns for each diff hist
    fullHist = np.empty(histSetShape)
    for i in range(dataShape[1]): #iterate over each vector of data
        hist, bins = np.histogram(data[:,i], bins = binNum) #histogram operations, numpy function
        fullHist[:,i] = hist
    return fullHist
```

Figure 3: produces histograms of a set of vectors

```python
#return JOINT probability distributions of the (undelayed) first vector in an array with ALL of the following (delayed) vectors
def jointSet(data, binNum): #data = input array, binNum = number of different bins the data should be sorted into/'resolution' of histogram
    numCombos = (np.shape(data)[1]-1) #since just comparisons with 1st vector
    arrShape = (binNum, binNum, numCombos) #note assuming SYMMETRY in binNum for 2D histograms ie symmetrical resolution
    arr = np.empty(arrShape)
    x = data[:,0]
    for i in range(numCombos):
        y = data[:,i+1]
        H, xedges, yedges = np.histogram2d(x, y, bins=binNum) #call histogram2d from numpy
        arr[:,:,i] = H
    return arr
```

Figure 4: produces 2D histograms between the first vector in a set, undelayed, and the other vectors in an array which are delayed by different amounts

```python
#return the average mutual information SET where each element is the AMI between the original data vector and a delayed vector
def AMI(singles, joints): #singles = histograms from the histALL function, joints = 2d histograms from the jointSet function
    index1 = np.shape(joints)[0]
    index2 = np.shape(joints)[1]
    dist1 = singles[:,0] #define the (undelayed) reference histogram
    index3 = np.shape(joints)[2]
    AMIset = np.empty(index3)
    for k in range(index3): #for the wider sum in the AMI function
        count=0
        dist2 = singles[:,(k+1)]
        for i in range(index1):
            for j in range(index2):
                count += joints[i,j,k]*mutual_info_score(dist1, dist2)  #where the average mutual information is added to, double sum for 2D histogram indices
        AMIset[k] = count
    return AMIset
```

Figure 5: calculates AMI for a set of 1D and 2D histograms from a time delay set

```python
#plots the results of average mutual information vs time delay for use in determining the 'optimal' tau
#data - from time series, dataWindowLength - self explanatory, numComparisons - 'how many time delays to do?'
#numBins - resolution of histograms, TDmultiplier - how large the steps do we want TD vectors to take
#NOTE takes a while to run, depending upon histogram resolution since many sort operations to conduct
def AMIanalysis(data, dataWindowLength, numComparisons, numBins, TDmultiplier):
    TDsetNow = TDset(data, dataWindowLength, numComparisons, TDmultiplier) #form set of time delay vectors
    histAllNow = histAll(TDsetNow, numBins) #determine histograms for each time delay
    jointSetNow = jointSet(TDsetNow, numBins) #determine joint probability histograms between undelayed vector and delayed vector set
    AMIsetNow = AMI(histAllNow, jointSetNow) #determines the AMIs
    plotTimeVec = np.linspace(1, (numComparisons-1), (numComparisons-1)) #plot the results
    plt.plot(plotTimeVec, AMIsetNow)
    plt.xlabel('Time delay steps, units = ' + str(TDmultiplier) + 'steps')
    plt.ylabel('Average Mutual Information')
    plt.show()
```

Figure 6: conducts an AMI analysis and plots results

```python
#function which conducts the AMI analysis but instead of plotting just returns results
#data = time series vector, dataWindowLength = cut length, TDmultiplier = TD step scaling,
#numComparisons = extent of AMI analysis tau's, numBins = resultion of histograms
def AMIanalysisResultOnly(data, dataWindowLength, numComparisons, numBins, TDmultiplier):
    TDsetNow = TDset(data, dataWindowLength, numComparisons, TDmultiplier)
    histAllNow = histAll(TDsetNow, numBins)
    jointSetNow = jointSet(TDsetNow, numBins)
    AMIsetNow = AMI(histAllNow, jointSetNow)
    return AMIsetNow
```

Figure 7: conducts an AMI analysis and returns results numerically rather than plotting

```
#returns an array of time delayed vectors with the dimensionality of the array defined by DE
#eg. if DE = 2 only original data & 1 delayed vector, if DE = 5 original vector and 4 vectors with increasingly more delay
def TD_DE_object(data, step, L, multiplier, DE): #data = time series data, step = 'shift' value, L = window length, mult = step scaling, DE = dimensionality
    shape = (L, DE)
    arr = np.empty(shape)
    for i in range(DE):
        arr[:,i] = TDvec(data, i*step, L, multiplier) #call TD vec func
    return arr
```

Figure 8: creates an array of time delay vectors with fixed tau, parametrized by the time delay embedding dimension D_E

```
#calls the nearest neighbor function of a dataset to determine the average distance and std distance between neighbors
#then returns a value, used for FNN analysis, which is a certain distance 'threshold' for neighbors to be classified with
def threshold(data, numSTDallowed): #data = time series data, numSTDallowed = 'how big do we want threshold to be?', typically choose between 3-15
    nbrs = NearestNeighbors(n_neighbors=2, algorithm='auto').fit(data) #note 'auto' means algo choosest 'optimal' method
    distances, indices = nbrs.kneighbors(data)
    avg = np.mean(distances[1])
    STD = np.std(distances[1])
    value = avg + STD*numSTDallowed #threshold value, NOTE how numSTDallowed can drastically change output of FNN analysis
    #eg very large numSTDallowed => all points have neighbors FNN constant 100%, very small numSTDallowed => 'neighborhood' tiny, FNN rapidly converges to 0
    return value
```

Figure 9: defines the threshold within which adjacent points are counted as 'neighbors' for false nearest neighbor purposes

```
#returns the % of a set of data which has a nearest neighbor within 'threshold' distance
def percentNN(data, threshold): #data = time series vector, threshold = as per 'threshold' function
    nbrs = NearestNeighbors(n_neighbors=2, algorithm='auto').fit(data)
    distances, indices = nbrs.kneighbors(data)
    percent = np.sum(distances <= threshold, axis=0)[1] / data.shape[0] #where % NN is determined from the distances
    return percent
```

Figure 10: calculates the percentage of false nearest neighbors for a given time delay array object

```
#plots the results of an analysis of false nearest neighbors vs embedding dimension
#data = time series vector, step - 'shift' for each TD vector, L = data window length, multiplier = scaling for TD shifts
#DE_max = total number of different dimensionalities to try, numSTDallowed = size of 'threshold' for nearest neighbors
#NOTE numSTDallowed can affect results significantly, see 'threshold function for detail
#tldr: if FNN very high or near 1, lower numSTDallowed. if FNN near 0 for all values, raise numSTDallowed. suggest numSTDallowed between 3 and 15
def FNNanalysis(data, step, L, multiplier, DE_max, numSTDallowed):
    FNNvsDE = np.empty(DE_max)
    thresholdData =  TD_DE_object(data, 0*step, L, multiplier, 1) #use undelayed data for threshold
    thresh = threshold(thresholdData, numSTDallowed) #define the FNN threshold
    FNNvsDE[0] = percentNN(thresholdData, thresh)  #first element of result, should be 100% or 1
    for i in range(DE_max-1):
        actualDimension = i+2 #used to avoid some issues with using 0 in certain functions
        arr = TD_DE_object(data, step, L, multiplier, actualDimension) #form a new array with dimensionality corresponding to loop iterator
        FNNvsDE[actualDimension - 1] = percentNN(arr, thresh) #FNN for the current TD system with dimensionality actualDimension
    plotDEVec = np.linspace(1, (DE_max), (DE_max))
    plt.plot(plotDEVec, FNNvsDE)
    plt.xlabel('Time delay embedding dimension')
    plt.ylabel('False Nearest Neighbor percentage')
    plt.show()
```

Figure 11: conducts a FNN analysis and plots results

```
#conduct the FNN analysis, but simply return results for further use rather than plotting
#data = time series vector, step - 'shift' for each TD vector, L = data window length, multiplier = scaling for TD shifts
#DE_max = total number of different dimensionalities to try, numSTDallowed = size of 'threshold' for nearest neighbors
#NOTE numSTDallowed can affect results significantly, see 'threshold function for detail
#tldr: if FNN very high or near 1, lower numSTDallowed. if FNN near 0 for all values, raise numSTDallowed. suggest numSTDallowed between 3 and 15
def FNNanalysisResultOnly(data, step, L, multiplier, DE_max, numSTDallowed):
    FNNvsDE = np.empty(DE_max)
    thresholdData =  TD_DE_object(data, 0*step, L, multiplier, 1) #use undelayed data for threshold
    thresh = threshold(thresholdData, numSTDallowed) #define the FNN threshold
    FNNvsDE[0] = percentNN(thresholdData, thresh)  #first element of result, should be 100% or 1
    for i in range(DE_max-1):
        actualDimension = i+2  #used to avoid some issues with using 0 in certain functions
        arr = TD_DE_object(data, step, L, multiplier, actualDimension) #form a new array with dimensionality corresponding to loop iterator
        FNNvsDE[actualDimension - 1] = percentNN(arr, thresh) #FNN for the current TD system with dimensionality actualDimension
    return FNNvsDE
```

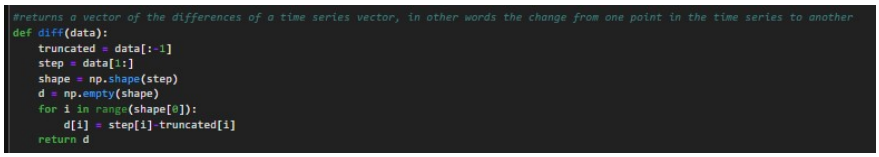
Figure 12: conducts a FNN analysis and returns results numerically

```python
#builds a 2 dimensional time delay array and plots the points as a scatter plot in time delay space
#data = time series data, TDmult = time delay step scaling, windowLength = data cut length, title = string for plot title, path = .jpg save path
def TDplot2D(data, TDmult, windowLength, title, path):
    arr = TDset(data, windowLength, 2, TDmult)
    x = arr[:,0]
    y = arr[:,1]
    fig = plt.figure()
    ax = fig.add_subplot()
    ax.scatter(x, y)
    ax.set_xlabel('raw data')
    ax.set_ylabel('Time delayed by ' + str(TDmult))
    ax.set_title(title)
    plt.savefig(path + '/TD2Dplot' + titleNow + '.jpg')
    plt.show()
```

Figure 13: 2D time delay plotting function

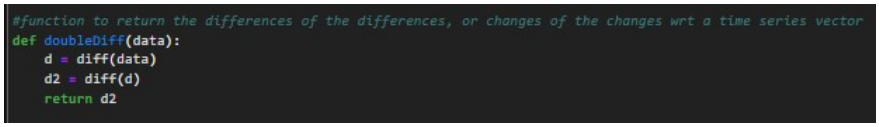
```python
#builds a 3 dimensional time delay array and plots the points as a scatter plot in time delay space
#data = time series data, TDmult = time delay step scaling, windowLength = data cut length, title = string for plot title, path = .jpg save path
def TDplot3D(data, TDmult, windowLength, title, path):
    arr =  TDset(data, windowLength, 3, TDmult)
    x = arr[:,0]
    y = arr[:,1]
    z = arr[:,2]
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(x, y, z)
    ax.set_xlabel('raw data')
    ax.set_ylabel('Time delayed by ' + str(TDmult))
    ax.set_zlabel('Time delayed by ' + str(2*TDmult))
    ax.set_title(title)
    plt.savefig(path + '/TD3Dplot' + titleNow + '.jpg')
    plt.show()
```

Figure 14: 3D time delay plotting function



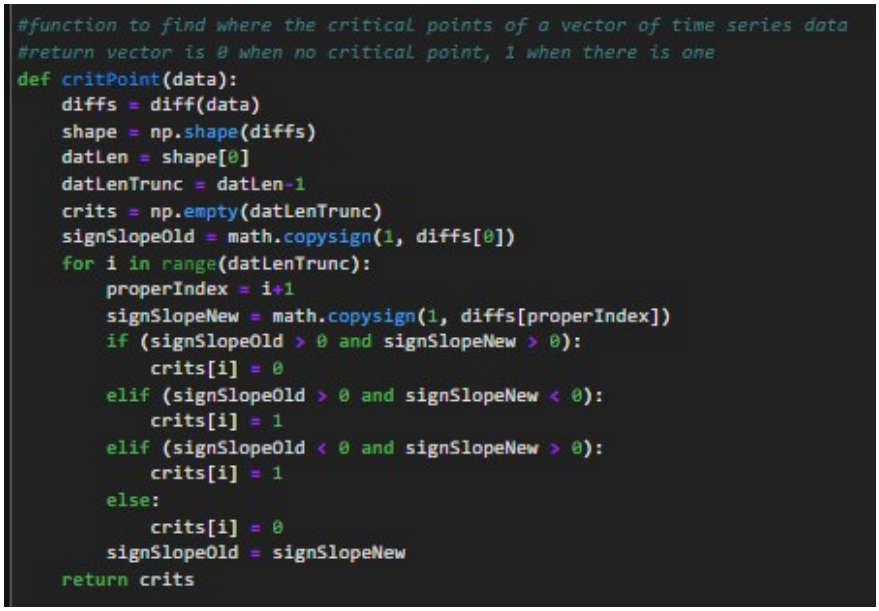
```
#returns a vector of the differences of a time series vector, in other words the change from one point in the time series to another
def diff(data):
    truncated = data[:-1]
    step = data[1:]
    shape = np.shape(step)
    d = np.empty(shape)
    for i in range(shape[0]):
        d[i] = step[i]-truncated[i]
    return d
```

Figure 15: differences between each point in a time series vector



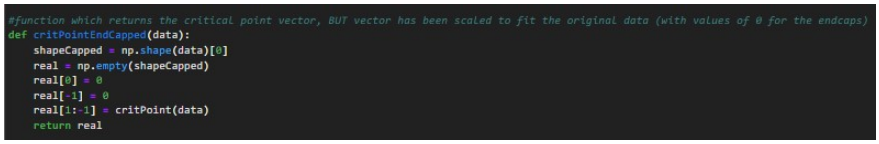
```
#function to return the differences of the differences, or changes of the changes wrt a time series vector
def doubleDiff(data):
    d = diff(data)
    d2 = diff(d)
    return d2
```

Figure 16: 2nd order differences in a time series vector



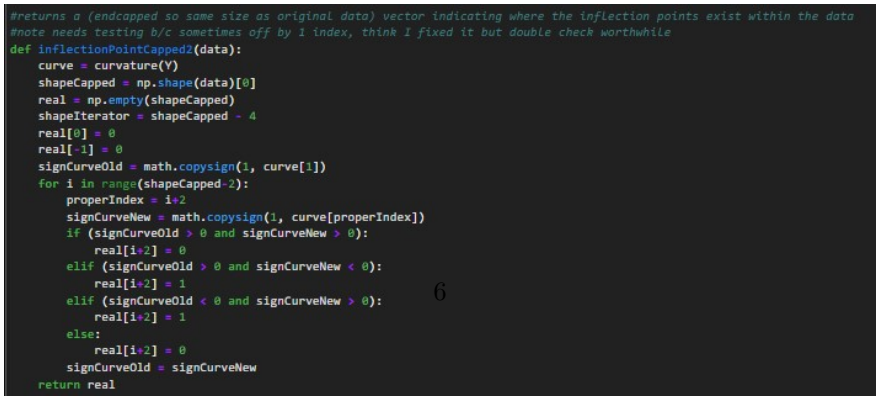
```
#function to find where the critical points of a vector of time series data
#return vector is 0 when no critical point, 1 when there is one
def critPoint(data):
    diffs = diff(data)
    shape = np.shape(diffs)
    datLen = shape[0]
    datLenTrunc = datLen-1
    crits = np.empty(datLenTrunc)
    signSlopeOld = math.copysign(1, diffs[0])
    for i in range(datLenTrunc):
        properIndex = i+1
        signSlopeNew = math.copysign(1, diffs[properIndex])
        if (signSlopeOld > 0 and signSlopeNew > 0):
            crits[i] = 0
        elif (signSlopeOld > 0 and signSlopeNew < 0):
            crits[i] = 1
        elif (signSlopeOld < 0 and signSlopeNew > 0):
            crits[i] = 1
        else:
            crits[i] = 0
        signSlopeOld = signSlopeNew
    return crits
```

Figure 17: critical point index marker for a time series vector



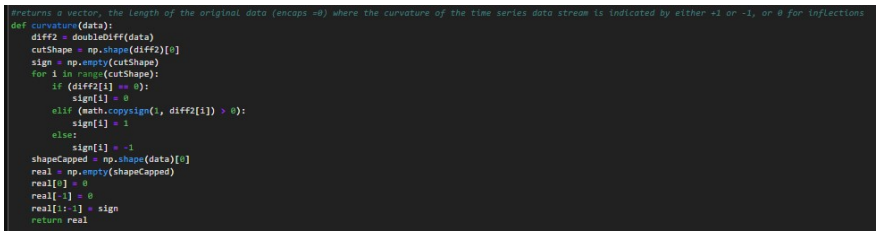
```
#function which returns the critical point vector, BUT vector has been scaled to fit the original data (with values of 0 for the endcaps)
def critPointEndCapped(data):
    shapeCapped = np.shape(data)[0]
    real = np.empty(shapeCapped)
    real[0] = 0
    real[-1] = 0
    real[1:-1] = critPoint(data)
    return real
```

Figure 18: rescaling critical point function to fit original data length



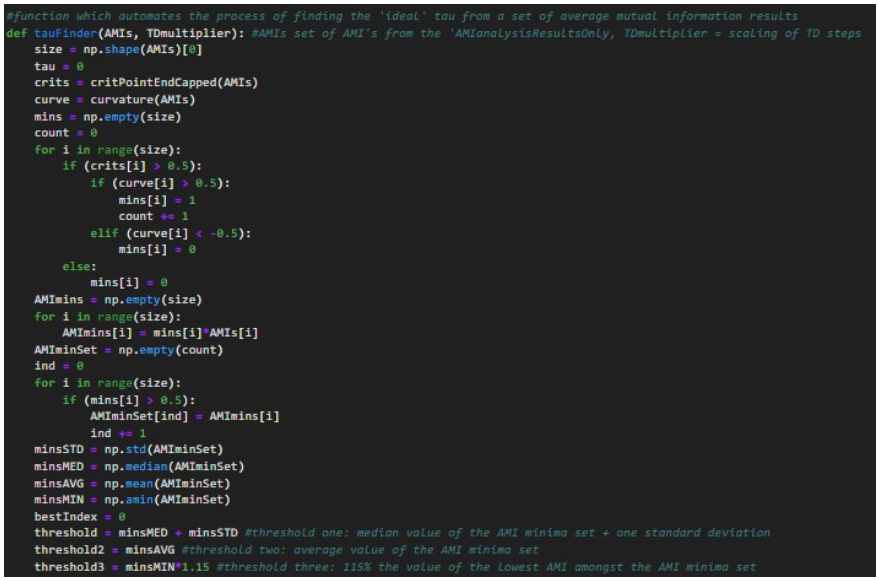
```
#returns a (endcapped so same size as original data) vector indicating where the inflection points exist within the data
#note needs testing b/c sometimes off by 1 index, think I fixed it but double check worthwhile
def inflectionPointCapped2(data):
    curve = curvature(Y)
    shapeCapped = np.shape(data)[0]
    real = np.empty(shapeCapped)
    shapeIterator = shapeCapped - 4
    real[0] = 0
    real[-1] = 0
    signCurveOld = math.copysign(1, curve[1])
    for i in range(shapeCapped-2):
        properIndex = i+2
        signCurveNew = math.copysign(1, curve[properIndex])
        if (signCurveOld > 0 and signCurveNew > 0):
            real[i+2] = 0
        elif (signCurveOld > 0 and signCurveNew < 0):
            real[i+2] = 1
        elif (signCurveOld < 0 and signCurveNew > 0):
            real[i+2] = 1
        else:
            real[i+2] = 0
        signCurveOld = signCurveNew
    return real
```

Figure 19: inflection point marker for time series data



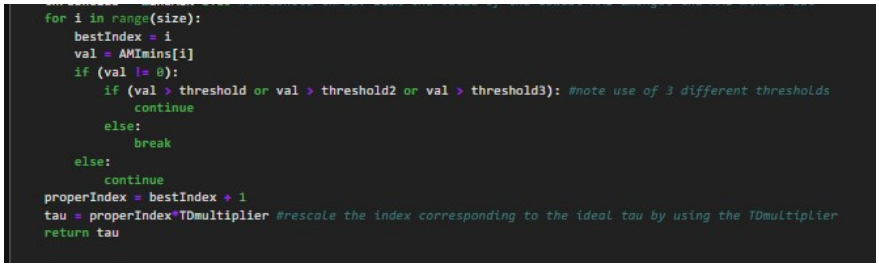
```
#returns a vector, the length of the original data (encaps =0) where the curvature of the time series data stream is indicated by either +1 or -1, or 0 for inflections
def curvature(data):
    diff2 = doubleDiff(data)
    cutShape = np.shape(diff2)[0]
    sign = np.empty(cutShape)
    for i in range(cutShape):
        if (diff2[i] == 0):
            sign[i] = 0
        elif (math.copysign(1, diff2[i]) > 0):
            sign[i] = 1
        else:
            sign[i] = -1
    shapeCapped = np.shape(data)[0]
    real = np.empty(shapeCapped)
    real[0] = 0
    real[-1] = 0
    real[1:-1] = sign
    return real
```

Figure 20: curvature indicator for time series data



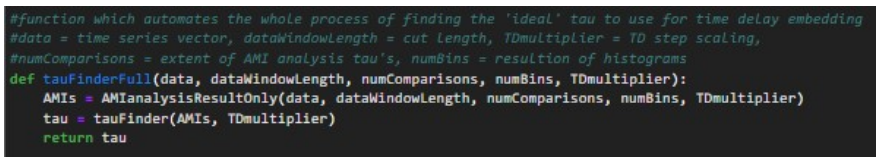
```
#function which automates the process of finding the 'ideal' tau from a set of average mutual information results
def tauFinder(AMIs, TDmultiplier): #AMIs set of AMI's from the 'AMIanalysisResultsOnly, TDmultiplier = scaling of TD steps
    size = np.shape(AMIs)[0]
    tau = 0
    crits = critPointEndCapped(AMIs)
    curve = curvature(AMIs)
    mins = np.empty(size)
    count = 0
    for i in range(size):
        if (crits[i] > 0.5):
            if (curve[i] > 0.5):
                mins[i] = 1
                count += 1
            elif (curve[i] < -0.5):
                mins[i] = 0
        else:
            mins[i] = 0
    AMImins = np.empty(size)
    for i in range(size):
        AMImins[i] = mins[i]*AMIs[i]
    AMIminSet = np.empty(count)
    ind = 0
    for i in range(size):
        if (mins[i] > 0.5):
            AMIminSet[ind] = AMImins[i]
            ind += 1
    minsSTD = np.std(AMIminSet)
    minsMED = np.median(AMIminSet)
    minsAVG = np.mean(AMIminSet)
    minsMIN = np.amin(AMIminSet)
    bestIndex = 0
    threshold = minsMED + minsSTD #threshold one: median value of the AMI minima set + one standard deviation
    threshold2 = minsAVG #threshold two: average value of the AMI minima set
    threshold3 = minsMIN*1.15 #threshold three: 115% the value of the lowest AMI amongst the AMI minima set
```

Figure 21: function to find the optimal choice of τ for TDE, part 1



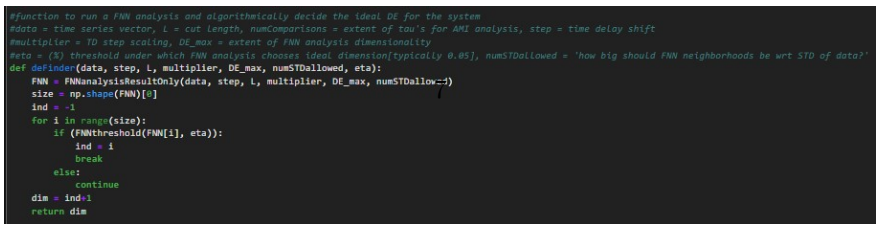
```
    for i in range(size):
        bestIndex = i
        val = AMImins[i]
        if (val != 0):
            if (val > threshold or val > threshold2 or val > threshold3): #note use of 3 different thresholds
                continue
            else:
                break
        else:
            continue
    properIndex = bestIndex + 1
    tau = properIndex*TDmultiplier #rescale the index corresponding to the ideal tau by using the TDmultiplier
    return tau
```

Figure 22: function to find the optimal choice of τ for TDE, part 2



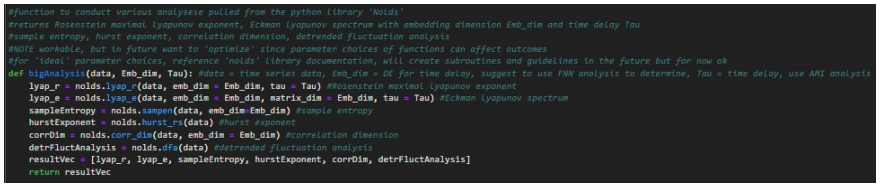
```
#function which automates the whole process of finding the 'ideal' tau to use for time delay embedding
#data = time series vector, dataWindowLength = cut length, TDmultiplier = TD step scaling,
#numComparisons = extent of AMI analysis tau's, numBins = resultion of histograms
def tauFinderFull(data, dataWindowLength, numComparisons, numBins, TDmultiplier):
    AMIs = AMIanalysisResultOnly(data, dataWindowLength, numComparisons, numBins, TDmultiplier)
    tau = tauFinder(AMIs, TDmultiplier)
    return tau
```

Figure 23: returns the optimal tau for TDE given time series data



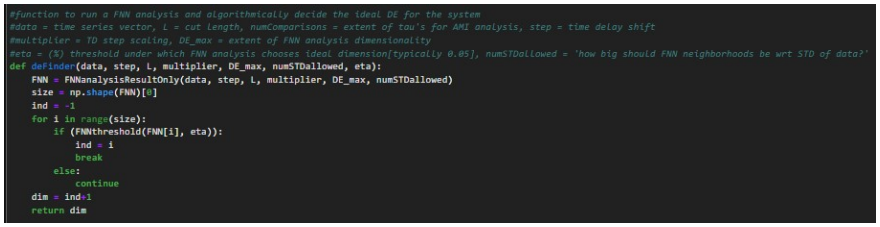
```
#function to run a FNN analysis and algorithmically decide the ideal DE for the system
#data = time series vector, L = cut length, numComparisons = extent of tau's for AMI analysis, step = time delay shift
#multiplier = TD step scaling, DE_max = extent of FNN analysis dimensionality
#eta = (%) threshold under which FNN analysis chooses ideal dimension[typically 0.05], numSTDallowed = 'how big should FNN neighborhoods be wrt STD of data?'
def deFinder(data, step, L, multiplier, DE_max, numSTDallowed, eta):
    FNN = FNNanalysisResultOnly(data, step, L, multiplier, DE_max, numSTDallowed)
    size = np.shape(FNN)[0]
    ind = -1
    for i in range(size):
        if (FNNthreshold(FNN[i], eta)):
            ind = i
            break
        else:
            continue
    dim = ind+1
    return dim
```

Figure 24: function to find the optimal choice of D_E for TDE



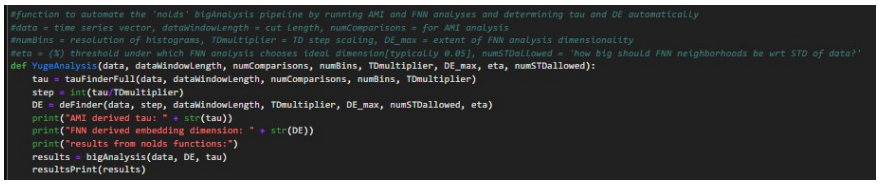
```
#function to conduct various analysese pulled from the python library 'Nolds'
#returns Rosenstein maximal lyapunov exponent, Eckman lyapunov spectrum with embedding dimension Emb_dim and time delay Tau
#sample entropy, hurst exponent, correlation dimension, detrended fluctuation analysis
#NOTE workable, but in future want to 'optimize' since parameter choices of functions can affect outcomes
#For 'ideal' parameter choices, reference 'nolds' library documentation, will create subroutines and guidelines in the future but for now ok
def bigAnalysis(data, Emb_dim, Tau): #data = time series data, Emb_dim = DE for time delay, suggest to use FNN analysis to determine, Tau = time delay, use AMI analysis
    lyap_r = nolds.lyap_r(data, emb_dim = Emb_dim, tau = Tau) #Rosenstein maximal lyapunov exponent
    lyap_e = nolds.lyap_e(data, emb_dim = Emb_dim, matrix_dim = Emb_dim, tau = Tau) #Eckman lyapunov spectrum
    sampleEntropy = nolds.sampen(data, emb_dim=Emb_dim) #sample entropy
    hurstExponent = nolds.hurst_rs(data) #hurst exponent
    corrDim = nolds.corr_dim(data, emb_dim = Emb_dim) #correlation dimension
    detrFluctAnalysis = nolds.dfa(data) #detrended fluctuation analysis
    resultVec = [lyap_r, lyap_e, sampleEntropy, hurstExponent, corrDim, detrFluctAnalysis]
    return resultVec
```

Figure 25: push data through the set of 'nolds' metric algorithms



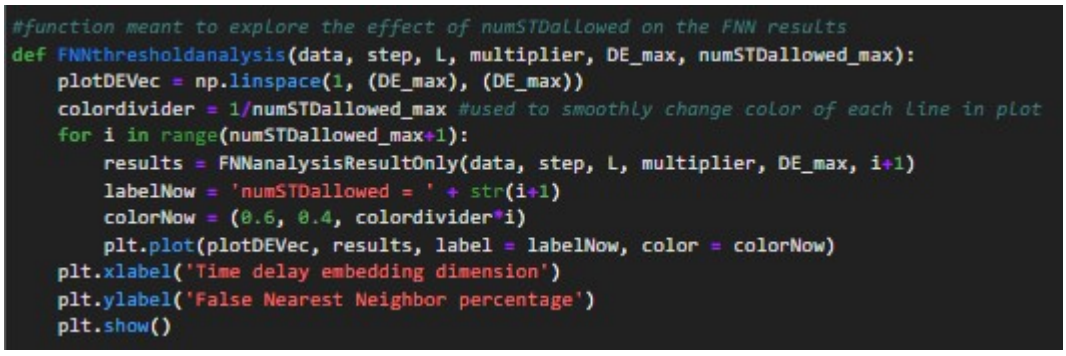
```
#function to run a FNN analysis and algorithmically decide the ideal DE for the system
#data = time series vector, L = cut length, numComparisons = extent of tau's for AMI analysis, step = time delay shift
#multiplier = TD step scaling, DE_max = extent of FNN analysis dimensionality
#eta = (%) threshold under which FNN analysis chooses ideal dimension[typically 0.05], numSTDallowed = 'how big should FNN neighborhoods be wrt STD of data?'
def deFinder(data, step, L, multiplier, DE_max, numSTDallowed, eta):
    FNN = FNNanalysisResultOnly(data, step, L, multiplier, DE_max, numSTDallowed)
    size = np.shape(FNN)[0]
    ind = -1
    for i in range(size):
        if (FNNthreshold(FNN[i], eta)):
            ind = i
            break
        else:
            continue
    dim = ind+1
    return dim
```

Figure 26: print the results from 'bigAnalysis' in a nice format



```
#function to automate the 'nolds' bigAnalysis pipeline by running AMI and FNN analyses and determining tau and DE automatically
#data = time series vector, dataWindowLength = cut length, numComparisons = for AMI analysis
#numBins = resolution of histograms, TDmultiplier = TD step scaling, DE_max = extent of FNN analysis dimensionality
#eta = (%) threshold under which FNN analysis chooses ideal dimension[typically 0.05], numSTDallowed = 'how big should FNN neighborhoods be wrt STD of data?'
def YugeAnalysis(data, dataWindowLength, numComparisons, numBins, TDmultiplier, DE_max, eta, numSTDallowed):
    tau = tauFinderFull(data, dataWindowLength, numComparisons, numBins, TDmultiplier)
    step = int(tau/TDmultiplier)
    DE = deFinder(data, step, dataWindowLength, TDmultiplier, DE_max, numSTDallowed, eta)
    print("AMI derived tau: " + str(tau))
    print("FNN derived embedding dimension: " + str(DE))
    print("results from nolds functions:")
    results = bigAnalysis(data, DE, tau)
    resultsPrint(results)
```

Figure 27: conduct the entire automated process of AMI, FNN, and subsequent 'bigAnalysis' with the automated choices of τ and D_E



```
#function meant to explore the effect of numSTDallowed on the FNN results
def FNNthresholdanalysis(data, step, L, multiplier, DE_max, numSTDallowed_max):
    plotDEVec = np.linspace(1, (DE_max), (DE_max))
    colordivider = 1/numSTDallowed_max #used to smoothly change color of each line in plot
    for i in range(numSTDallowed_max+1):
        results = FNNanalysisResultOnly(data, step, L, multiplier, DE_max, i+1)
        labelNow = 'numSTDallowed = ' + str(i+1)
        colorNow = (0.6, 0.4, colordivider*i)
        plt.plot(plotDEVec, results, label = labelNow, color = colorNow)
    plt.xlabel('Time delay embedding dimension')
    plt.ylabel('False Nearest Neighbor percentage')
    plt.show()
```

Figure 28: function used to check the effects of the 'threshold' within a false nearest neighbors algorithm

```
#imports and classes
import nolds #library for a set of different analysis functions wrt nonlinear time series
import numpy as np #manipulate vectors and arrays
import sklearn as sk #useful for certain data science tasks
import matplotlib.pyplot as plt #plotting
from scipy.integrate import odeint #ODE integration
from sklearn.metrics import mutual_info_score #mutual information funtion
from sklearn.neighbors import NearestNeighbors #nearest neighbors function
import math #used in 'optimization' algorithms
```

Figure 29: imports for 'YugeAnalysis' set of functions