#@mfunction("d, si, h")

import os

import numpy as np

import struct

class Abf():

def \_\_init\_\_(self, fn=None):

self.fn = fn

def abfload(self):

# \*\* function [d,si,h]=abfload(fn,varargin)

# loads and returns data in ABF (Axon Binary File) format.

# Data may have been acquired in the following modes:

# (1) event-driven variable-length (currently only abf versions < 2.0)

# (2) event-driven fixed-length or waveform-fixed length

# (3) gap-free

# Information about scaling, the time base and the number of channels and

# episodes is extracted from the header of the abf file.

#

# OPERATION

# If the second input variable is the char array 'info' as in

# [d,si,h]=abfload('d:\data01.abf','info')

# abfload will not load any data but return detailed information (header

# parameters) on the file in output variable h. d and si will be empty.

# In all other cases abfload will load data. Optional input parameters

# listed below (= all except the file name) must be specified as

# parameter/value pairs, e.g. as in

# d=abfload('d:\data01.abf','start',100,'stop','e'),

#

# >>> INPUT VARIABLES >>>

# NAME TYPE, DEFAULT DESCRIPTION

# fn char array abf data file name

# start scalar, 0 only gap-free-data: start of cutout to be

# read (unit: s)

# stop scalar or char, only gap-free-data: of cutout to be

# 'e' read (unit: sec). May be set to 'e' (

# of file).

# sweeps 1d-array or char, only episodic data: sweep numbers to be

# 'a' read. By default, all sweeps will be read

# ('a').

# channels cell array names of channels to be read, like

# or char, 'a' {'IN 0','IN 8'} (make sure spelling is

# 100% correct, including blanks). If set

# to 'a', all channels will be read.

# chunk scalar, 0.05 only gap-free-data: the elementary chunk

# size (megabytes) to be used for the

# 'discontinuous' mode of reading data

# (fewer channels to be read than exist)

# machineF char array, the 'machineformat' input parameter of the

# 'ieee-le' matlab fopen function. 'ieee-le' is the

# correct option for windows, deping on

# the platform the data were recorded/shall

# be read by abfload 'ieee-be' is the

# alternative.

# << OUTPUT VARIABLES <<<

# NAME TYPE DESCRIPTION

# d the data read, the format deping on the record-

# ing mode

# 1. GAP-FREE:

# 2d array 2d array of size

# <data pts> by <number of chans>

# Examples of access:

# d(:,2) data from channel 2 at full length

# d(1:100,:) first 100 data points from all channels

# 2. EPISODIC FIXED-LENGTH/WAVEFORM FIXED-LENGTH:

# 3d array 3d array of size

# <data pts per sweep> by <number of chans> by <number

# of sweeps>.

# Examples of access:

# d(:,2,:) a matrix containing all episodes

# (at full length) of the second

# channel in its columns

# d(1:200,:,[1 11]) contains first 200 data points of

# episodes 1 and 11 of all channels

# 3. EPISODIC VARIABLE-LENGTH:

# cell array cell array whose elements correspond to single sweeps.

# Each element is a (regular) array of size

# <data pts per sweep> by <number of chans>

# Examples of access:

# d{1} a 2d-array which contains episode 1

# (all of it, all channels)

# d{2}(1:100,2) a 1d-array containing the first 100

# data points of channel 2 in episode 1

# si scalar the sampling interval in us

# h struct information on file (selected header parameters)

#

#

# CONTRIBUTORS

# Original version by Harald Hentschke (harald.hentschke@uni-tuebingen.de)

# Exted to abf version 2.0 by Forrest Collman (fcollman@Princeton.edu)

# pvpmod.m by Ulrich Egert (egert@bccn.uni-freiburg.de)

# Date of this version: May 20, 2009

start = 0.0

stop = 'e'

# episodic

sweeps = 'a'

# general

channels = 'a'

# the size of data chunks (see above) in Mb. 0.05 Mb is an empirical value

# which works well for abf with 6(1,1)6 channels and recording durations of

# 5-30 min

chunk = 0.05

machineF = '<' #'ieee-le'

verbose = 1

# if first and only optional input argument is string 'info' the user's

# request is to obtain information on the file (header parameters), so set

# flag accordingly

doLoadData = True

# some constants

BLOCKSIZE = 512

# output variables

d = []

si = []

h = []

# check existence of file

if not os.path.exists(self.fn):

print 'could not find file '

return

# -------------------------------------------------------------------------

# PART 2a: determine abf version

# -------------------------------------------------------------------------

fid = open(self.fn, 'rb')#, machineF)

# on the occasion, determine absolute file size

fileSz = os.stat(self.fn)[7]

# \*\*\* read value of parameter 'fFileSignature' [i.e. abf version) from header \*\*\*

sz = 4

fFileSignature = char2str(struct.unpack('c'\*sz,fid.read(sz)))#, ''B'=>char'))

# rewind

fid.seek(0)

# transpose

#fFileSignature = fFileSignature.cT

# -------------------------------------------------------------------------

# PART 2b: define file information ('header') parameters of interest

# -------------------------------------------------------------------------

# The list of header parameters created below (variable 'headPar') is

# derived from the abf version 1.8 header section. It is by no means

# exhaustive [i.e. there are many more parameters in abf files) but

# sufficient for proper upload, scaling and arrangement of data acquired

# under many conditons. Further below, these parameters will be made fields

# of struct h. h, which is also an output variable, is then used in PART 3,

# which does the actual job of uploading, scaling and rearranging the data.

# That part of the code relies on h having a certain set of fields

# irrespective of ABF version.

# Unfortunately, in the transition to ABF version 2.0 many of the header

# parameters were moved to different places within the abf file and/or

# given other names or completely restructured. In order for the code to

# work with pre- and post-2.0 data files, all parameters missing in the

# header must be gotten into h. This is accomplished in lines ~262 and

# following:

# if h.fFileVersionNumber>=2

# ...

# Furthermore,

# - h as an output from an ABF version < 2.0 file will not contain new

# parameters introduced into the header like 'nCRCEnable'

# - h will in any case contain a few 'home-made' fields that have

# proven to be useful. Some of them dep on the recording mode. Among

# the more or less self-explanatory ones are

# -- si sampling interval

# -- recChNames the names of all channels, e.g. 'IN 8',...

# -- dataPtsPerChan sample points per channel

# -- dataPts sample points in file

# -- recTime recording start and stop time in seconds from

# midnight (millisecond resolution)

# -- sweepLengthInPts sample points per sweep (one channel)

# -- sweepStartInPts the start times of sweeps in sample points

# (from beginning of recording)

# call local function for header proper

headPar = self.define\_header(fFileSignature)

TagInfo = self.constTagInfo()

if 'ABF' in fFileSignature: # \*\* note the blank

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# abf version < 2.0

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# do nothing, header already defined above

pass

elif 'ABF2' in fFileSignature:

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# abf version >= 2.0

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Sections = self.Sections()

ProtocolInfo = self.constProtocolInfo()

ADCInfo = self.constADCInfo()

else:

print 'unknown or incompatible file signature'

# convert headPar to struct

#s = cell2struct(headPar, ('name'), 'offs'), 'numType'), 'value')), 2)

# -------------------------------------------------------------------------

# PART 2c: read parameters of interest

# -------------------------------------------------------------------------

# convert names in structure to variables and read value from header

h = {}

for k,v in headPar.items():

fid.seek(v[0])

# fid.close()

# print 'something went wrong locating '+ g).name]))

#h[k] = struct.unpack(k[1], fid.read(sz)) #read value(s) into dictionary

if hasattr(v[2],'size'):

c = v[2].size

else:

c=1

h[k] = np.fromfile(file=fid, dtype=v[1], count = v[2][0]\*v[2][1] ).reshape(v[2])

h[k] = h[k][0] # hack, how to get directly not a nested list?

# transpose

#h.fFileSignature = h.fFileSignature.cT

# several header parameters need a fix or version-specific refinement:

if 'ABF2' in fFileSignature:

# h["fFileVersionNumber"] needs to be converted from an array of integers to

# a 'f'

h["fFileVersionNumber"] = h["fFileVersionNumber"][3] + h["fFileVersionNumber"][2] \* .1 + h["fFileVersionNumber"][1] \* .001 + h["fFileVersionNumber"][0] \* .0001

# convert ms to s

h["lFileStartTime"] = h["uFileStartTimeMS"] \* .001

else:

# h["fFileVersionNumber"] is a float32 the value of which is sometimes a

# little less than what it should be (e.g. 1.6499999 instead of 1.65)

h["fFileVersionNumber"] = .001 \* round(h["fFileVersionNumber"] \* 1000)

# in abf < 2.0 two parameters are needed to obtain the file start time

# with millisecond precision - let's integrate both into parameter

# lFileStartTime (unit: s) so that nFileStartMillisecs will not be needed

h["lFileStartTime"] = h["lFileStartTime"] + h["nFileStartMillisecs"] \* .001

# \*\*\* read file information that has gone elsewhere in ABF version >= 2.0

# and assign values \*\*\*

if h["fFileVersionNumber"] >= 2:

# --- read in the Sections

offset = 76

for k,v in Sections.items():

exec(k+'='+ ReadSectionInfo(fid,offset)) in locals()

offset = offset + 4 + 4 + 8

# --- read in the Strings

fid.seek(StringsSection["uBlockIndex"] \* BLOCKSIZE)

BigString = struct.unpack('c',fid.read(StringsSection["uBytes"]))#, 'char'))

# this is a hack

goodstart = lower(BigString).find('clampex')

#this exts the hack to deal with axoscope files

if goodstart > -1:

goodstart = lower(BigString).find('axoscope')

BigString = BigString[goodstart[0]:]

stringends = (BigString == 0).nonzero()

stringends = [0].append(stringends)

Strings = []

for i in range(len(stringends) - 1):

Strings[i] = str(BigString[stringends[i]+1:stringends[i+1]-1])

h["recChNames"] = []

h["recChUnits"] = []

# --- read in the ADCSection

for i in range(ADCSection["llNumEntries"]):

ADCsec[i] = ReadSection(fid, ADCSection["uBlockIndex"] \* BLOCKSIZE + ADCSection["uBytes"] \* (i - 1), ADCInfo)

ii = ADCsec[i].nADCNum + 1

h["nADCSamplingSeq"][i] = ADCsec[i].nADCNum

h["recChNames"] = strvcat(h["recChNames"], Strings(ADCsec[i].lADCChannelNameIndex))

h["recChUnits"] = strvcat(h["recChUnits"], Strings(ADCsec[i].lADCUnitsIndex))

h["nTelegraphEnable"][ii] = ADCsec[i].nTelegraphEnable

h["fTelegraphAdditGain"][ii] = ADCsec[i].fTelegraphAdditGain

h["fInstrumentScaleFactor"][ii] = ADCsec[i].fInstrumentScaleFactor

h["fSignalGain"][ii] = ADCsec[i].fSignalGain

h["fADCProgrammableGain"][ii] = ADCsec[i].fADCProgrammableGain

h["fInstrumentOffset"][ii] = ADCsec[i].fInstrumentOffset

h["fSignalOffset"][ii] = ADCsec[i].fSignalOffset

# --- read in the protocol section

ProtocolSec = ReadSection(fid, ProtocolSection["uBlockIndex"] \* BLOCKSIZE, ProtocolInfo)

# ---

h["nOperationMode"] = ProtocolSec["nOperationMode"]

h["fSynchTimeUnit"] = ProtocolSec["fSynchTimeUnit"]

h["nADCNumChannels"] = ADCSection["llNumEntries"]

h["lActualAcqLength"] = DataSection["llNumEntries"]

h["lDataSectionPtr"] = DataSection["uBlockIndex"]

h["nNumPointsIgnored"] = 0

# in ABF version < 2.0 h["fADCSampleInterval"] is the sampling interval

# defined as

# 1/(sampling freq\*number\_of\_channels)

# so divide ProtocolSec.fADCSequenceInterval by the number of

# channels

h["fADCSampleInterval"] = ProtocolSec["fADCSequenceInterval"] / h["nADCNumChannels"]

h["fADCRange"] = ProtocolSec["fADCRange"]

h["lADCResolution"] = ProtocolSec["lADCResolution"]

h["lSynchArrayPtr"] = SynchArraySection["uBlockIndex"]

h["lSynchArraySize"] = SynchArraySection["llNumEntries"]

else:

TagSection = {}

TagSection["llNumEntries"] = h["lNumTagEntries"]

TagSection["uBlockIndex"] = h["lTagSectionPtr"]

TagSection["uBytes"] = 64

Tagsec = []

for i in range(TagSection["llNumEntries"]):

Tagsec[i] = ReadSection(fid, TagSection["uBlockIndex"] \* BLOCKSIZE + TagSection["uBytes"] \* (i - 1), TagInfo)

Tagsec[i]["sComment"] = Tagsec[i]["sComment"]

h["Tags"] = Tagsec

# -------------------------------------------------------------------------

# PART 2d: groom parameters & perform some plausibility checks

# -------------------------------------------------------------------------

if h["lActualAcqLength"] < h["nADCNumChannels"]:

fid.close()

print 'less data points than sampled channels in file'

# the numerical value of all recorded channels (numbers 0..15)

recChIdx = h["nADCSamplingSeq"][0:h["nADCNumChannels"]]

# the corresponding indices into loaded data d

recChInd = range(len(recChIdx))

if h["fFileVersionNumber"] < 2:

# the channel names, e.g. 'IN 8' (for ABF version 2.0 these have been

# extracted above at this point)

qq = h["sADCChannelName"].reshape((16, 10))

del h["sADCChannelName"]

qq = qq[recChIdx,:]

h["recChNames"] = []

for rc in range(qq.shape[0]):

h["recChNames"].append(struct.unpack(str(qq.shape[1])+'s',qq[rc,:])[0])

# same with signal units

qq = h["sADCUnits"].reshape((16,8))

del h["sADCUnits"]

qq = qq[recChIdx,:]

h["recChUnits"] = []

for rc in range(qq.shape[0]):

h["recChUnits"].append(struct.unpack(str(qq.shape[1])+'s',qq[rc,:])[0])

# convert to cell arrays

# h["recChNames"] = deblank(cellstr(h["recChNames"]))

# h["recChUnits"] = deblank(cellstr(h["recChUnits"]))

# check whether requested channels exist

chInd = []

eflag = 0

if isinstance(channels, basestring):

if channels == 'a':

chInd = recChInd

else:

fid.close()

print 'input parameter \'channels\' must either be a cell array holding channel names or the single character \'a\' (=all channels)'

else:

for i in range(len(channels)):

tmpChInd = h["recChNames"].find(channels[i])

if len(tmpChInd) > 0:

chInd = [chInd, tmpChInd]

else:

# set error flag to 1

eflag = 1

if eflag:

fid.close()

print'\*\*\*\* available channels:'

print h["recChNames"]

print' '

print'\*\*\*\* requested channels:'

print channels

print 'at least one of the requested channels does not exist in data file (see above)'

# display available channels if in info mode

if not doLoadData:

print '\*\*\*\* available channels:'

print h["recChNames"]

# gain of telegraphed instruments, if any

if h["fFileVersionNumber"] >= 1.65:

addGain = h["nTelegraphEnable"] \* h["fTelegraphAdditGain"]

addGain = np.where(addGain==0, 1, addGain)#addGain(addGain == 0] = 1

else:

addGain = np.ones(h["fTelegraphAdditGain"].shape)

# determine offset at which data start

if h["nDataFormat"] == 0:

dataSz = 2 # bytes/point

precision = 'i2' #int16

elif h["nDataFormat"] == 1:

dataSz = 4 # bytes/point

precision = 'f'

else:

fid.close()

print 'invalid number format'

headOffset = h["lDataSectionPtr"] \* BLOCKSIZE + h["nNumPointsIgnored"] \* dataSz

# h["fADCSampleInterval"] is the TOTAL sampling interval

h["si"] = h["fADCSampleInterval"] \* h["nADCNumChannels"]

# assign same value to si, which is an output variable

si = h["si"]

if isinstance(sweeps,basestring) and sweeps == 'a':

nSweeps = h["lActualEpisodes"]

sweeps = np.arange(h["lActualEpisodes"])

else:

nSweeps = len(sweeps)

# -------------------------------------------------------------------------

# PART 3: read data (note: from here on code is generic and abf version

# should not matter)

# -------------------------------------------------------------------------

if h["nOperationMode"] == 1:

print 'data were acquired in event-driven variable-length mode'

if h["fFileVersionNumber"] >= 2.0:

print 'abfload currently does not work with data acquired in event-driven variable-length mode and ABF version 2.0'

else:

if (h["lSynchArrayPtr"] <= 0 or h["lSynchArraySize"] <= 0):

fid.close()

print 'internal variables \'lSynchArraynnn\' are zero or negative'

if h["fSynchTimeUnit"] == 0: # time information in synch array section is in terms of ticks

h["synchArrTimeBase"] = 1

else: # time information in synch array section is in terms of usec

h["synchArrTimeBase"] = h["fSynchTimeUnit"]

# the byte offset at which the SynchArraySection starts

h["lSynchArrayPtrByte"] = BLOCKSIZE \* h["lSynchArrayPtr"]

# before reading Synch Arr parameters check if file is big enough to hold them

# 4 bytes/long, 2 values per episode (start and length)

if h["lSynchArrayPtrByte"] + 2 \* 4 \* h["lSynchArraySize"] < fileSz:

fid.close()

print 'file seems not to contain complete Synch Array Section'

synchArr = struct.unpack('i', fid.read(h["lSynchArraySize"] \* 2))

if h["lSynchArraySize"] \* 2 != len(synchArr):

fid.close()

print 'something went wrong reading synch array section'

# make synchArr a h["lSynchArraySize"] x 2 matrix

synchArr = synchArr.cT.reshape(2, h["lSynchArraySize"])[2:1]

# the length of episodes in sample points

segLengthInPts = synchArr[:, 2] / h["synchArrTimeBase"]

# the starting ticks of episodes in sample points WITHIN THE DATA FILE

segStartInPts = np.array([0, segLengthInPts[:-2].cT] \* dataSz).cumsum() + headOffset

# start time (synchArr(:,1)) has to be divided by h["nADCNumChannels"] to get true value

# go to data portion

# \*\* load data if requested

if doLoadData:

for i in range(nSweeps):

# if selected sweeps are to be read, seek correct position

if nSweeps != h["lActualEpisodes"]:

fid.seek(segStartInPts[sweeps[i]])

tmpd = np.fromfile(file=fid, dtype=precision, count = segLengthInPts[sweeps[i]])

#tmpd = struct.unpack(precision, fid.read(segLengthInPts[sweeps[i]]))

# if n != segLengthInPts(sweeps(i]):

# print 'something went wrong reading episode '+str(sweeps(i])+ ': '+ segLengthInPts(sweeps(i])+' points should have been read, ' +str(n)+ ' points actually read'

h["dataPtsPerChan"] = tmpd.size / h["nADCNumChannels"]

if n % h["nADCNumChannels"] > 0:

fid.close()

print 'number of data points in episode not OK'

# separate channels..

tmpd = tmpd.reshape((h["nADCNumChannels"], h["dataPtsPerChan"]))

# retain only requested channels

tmpd = tmpd[chInd,:].cT

# if data format is integer, scale appropriately, if it's 'f', tmpd is fine

if not h["nDataFormat"]:

for j in range(len(chInd)):

ch = recChIdx[chInd[j]] + 1

tmpd[:,j] = tmpd[:, j] / (h["fInstrumentScaleFactor"][ch] \

\* h["fSignalGain"][ch] \* h["fADCProgrammableGain"][ch] \

\* addGain[ch]) \* h["fADCRange"] / \

h["lADCResolution"] + h["fInstrumentOffset"][ch] - h["fSignalOffset"][ch]

# now place in cell array, an element consisting of one sweep with channels in columns

d[i] = tmpd

elif h["nOperationMode"] in (2, 5):

if h["nOperationMode"] == 2:

print'data were acquired in event-driven fixed-length mode'

else:

print'data were acquired in waveform fixed-length mode (clampex only)'

# extract timing information on sweeps

if (h["lSynchArrayPtr"] <= 0 or h["lSynchArraySize"] <= 0):

fid.close()

print 'internal variables \'lSynchArraynnn\' are zero or negative'

if h["fSynchTimeUnit"] == 0: # time information in synch array section is in terms of ticks

h["synchArrTimeBase"] = 1

else: # time information in synch array section is in terms of usec

h["synchArrTimeBase"] = h["fSynchTimeUnit"]

# the byte offset at which the SynchArraySection starts

h["lSynchArrayPtrByte"] = BLOCKSIZE \* h["lSynchArrayPtr"]

# before reading Synch Arr parameters check if file is big enough to hold them

# 4 bytes/long, 2 values per episode (start and length)

if h["lSynchArrayPtrByte"] + 2 \* 4 \* h["lSynchArraySize"] > fileSz:

fid.close()

print 'file seems not to contain complete Synch Array Section'

fid.seek(h["lSynchArrayPtrByte"])

# fid.close()

# print 'something went wrong positioning file pointer to Synch Array Section'

synchArr = np.fromfile(file=fid, dtype='i', count = h["lSynchArraySize"] \* 2)

#struct.unpack('i',fid.read(h["lSynchArraySize"] \* 2))

#if n != h["lSynchArraySize"] \* 2:

# fid.close()

# print 'something went wrong reading synch array section'

# make synchArr a h["lSynchArraySize"] x 2 matrix

# synchArr = synchArr.reshape((2,h["lSynchArraySize"]))[2, 1] # rearrange dimension, why???

if np.unique(synchArr.transpose()[1]).size > 1:

fid.close()

print 'sweeps of unequal length in file recorded in fixed-length mode'

# the length of sweeps in sample points (\*\*note: parameter lLength of

# the ABF synch section is expressed in samples (ticks) whereas

# parameter lStart is given in synchArrTimeBase units)

h["sweepLengthInPts"] = np.unique(synchArr[1]) / h["nADCNumChannels"]

# the starting ticks of episodes in sample points (t0=1=beginning of

# recording)

h["sweepStartInPts"] = np.unique(synchArr[0]) \* (h["synchArrTimeBase"] / h["fADCSampleInterval"] / h["nADCNumChannels"])

# recording start and stop times in seconds from midnight

h["recTime"] = h["lFileStartTime"]

h["recTime"] = h["recTime"] + np.hstack(([0], 1e-6 \* (h["sweepStartInPts"] + h["sweepLengthInPts"]) \* h["fADCSampleInterval"] \* h["nADCNumChannels"]))

# determine first point and number of points to be read

startPt = 0

h["dataPts"] = h["lActualAcqLength"]

h["dataPtsPerChan"] = h["dataPts"] / h["nADCNumChannels"]

if h["dataPts"] % h["nADCNumChannels"] > 0 or h["dataPtsPerChan"] % h["lActualEpisodes"] > 0: #remainders

fid.close()

print 'number of data points not OK'

# temporary helper var

dataPtsPerSweep = h["sweepLengthInPts"] \* h["nADCNumChannels"]

fid.seek(startPt \* dataSz + headOffset) #!= 0:

# fid.close()

# print 'something went wrong positioning file pointer (too few data points ?)'

d = np.zeros((h["sweepLengthInPts"], len(chInd), nSweeps),dtype = precision)

# the starting ticks of episodes in sample points WITHIN THE DATA FILE

selectedSegStartInPts = (sweeps \* dataPtsPerSweep) \* dataSz + headOffset

# \*\* load data if requested

if doLoadData:

for i in range(nSweeps):

fid.seek(selectedSegStartInPts[i])

tmpd = np.fromfile(file=fid, dtype=precision, count = dataPtsPerSweep)

h["dataPtsPerChan"] = tmpd.size / h["nADCNumChannels"][0] #one of the arguments is not list, address explicitly

if tmpd.size % h["nADCNumChannels"] > 0:

fid.close()

print 'number of data points in episode not OK'

# separate channels..

tmpd = tmpd.reshape((h["dataPtsPerChan"],h["nADCNumChannels"]))

# retain only requested channels

tmpd = tmpd[:,chInd]

# tmpd = tmpd.cT

# if data format is integer, scale appropriately, if it's 'f', d is fine

if not h["nDataFormat"]:

for j in range(len(chInd)):

ch = recChIdx[chInd[j]] # + 1

tmpd[:,j] = tmpd[:,j] / (h["fInstrumentScaleFactor"][ch] \* h["fSignalGain"][ch] \* h["fADCProgrammableGain"][ch] \* addGain[ch]) \* h["fADCRange"][0] / h["lADCResolution"][0] + h["fInstrumentOffset"][ch] - h["fSignalOffset"][ch]

# now fill 3d array

d[:,:, i] = tmpd

elif h["nOperationMode"] == 3:

print'data were acquired in gap-free mode'

# from start, stop, headOffset and h["fADCSampleInterval"] calculate first point to be read

# and - unless stop is given as 'e' - number of points

startPt = np.floor(1e6 \* start \* (1 / h["fADCSampleInterval"]))

# this corrects undesired shifts in the reading frame due to rounding errors in the previous calculation

startPt = np.floor(startPt / h["nADCNumChannels"]) \* h["nADCNumChannels"]

# if stop is a char array, it can only be 'e' at this point (other values would have

# been caught above)

if isinstance(stop,basestring):

h["dataPtsPerChan"] = h["lActualAcqLength"] / h["nADCNumChannels"] - np.floor(1e6 \* start / h["si"])

h["dataPts"] = h["dataPtsPerChan"] \* h["nADCNumChannels"]

else:

h["dataPtsPerChan"] = np.floor(1e6 \* (stop - start) \* (1 / h["si"]))

h["dataPts"] = h["dataPtsPerChan"] \* h["nADCNumChannels"]

if h["dataPts"] <= 0:

fid.close()

print 'start is larger than or equal to stop'

if h["dataPts"] % h["nADCNumChannels"] > 0:

fid.close()

print 'number of data points not OK'

tmp = 1e-6 \* h["lActualAcqLength"] \* h["fADCSampleInterval"]

# if verbose:

# print 'total length of recording: '), num2str(tmp, '%5.1f')), ' s ~ '), num2str(tmp / 60, '%3.0f')), ' min')

# 8 bytes per data point expressed in Mb

# print 'memory requirement for complete upload in matlab: '), num2str(round(8 \* h["lActualAcqLength"] / 2 \*\* 20)), ' MB')

# recording start and stop times in seconds from midnight

h["recTime"] = h["lFileStartTime"]

h["recTime"] = np.hstack([h["recTime"], h["recTime"] + tmp])

#if fid.seek(startPt \* dataSz + headOffset) != 0:

# fid.close()

# print 'something went wrong positioning file pointer (too few data points ?)'

if doLoadData:

# \*\*\* decide on the most efficient way to read data:

# [i] all (of one or several) channels requested: read, done

# [ii] one (of several) channels requested: use the 'skip' feature of

# fread

# [iii] more than one but not all (of several) channels requested:

# 'discontinuous' mode of reading data. Read a reasonable chunk of data

# (all channels), separate channels, discard non-requested ones [if

# any), place data in preallocated array, repeat until done. This is

# faster than reading the data in one big lump, separating channels and

# discarding the ones not requested

if len(chInd) == 1 and h["nADCNumChannels"] > 1:

# --- situation [ii]

# jump to proper reading frame position in file

if fid.seek((chInd - 1) \* dataSz) != 0:

fid.close()

print 'something went wrong positioning file pointer (too few data points ?)'

skip = dataSz \* (h["nADCNumChannels"] - 1)

prec = precision + 'x'\*skip #pad bytes

nbytes=struct.calcsize(prec)

# read, skipping h["nADCNumChannels"]-1 data points after each read

d = struct.unpack(prec, fid.read(h["dataPtsPerChan"]\*nbytes))

# if n != h["dataPtsPerChan"]:

# fid.close()

# print 'something went wrong reading file ('+ str(h["dataPtsPerChan"])+ ' points should have been read, '+str(n)+ ' points actually read'

elif len(chInd) / h["nADCNumChannels"] < 1:

# --- situation [iii]

# prepare chunkwise upload:

# preallocate d

d = np.empty((h["dataPtsPerChan"], len(chInd)))

# the number of data points corresponding to the maximal chunk size,

# rounded off such that from each channel the same number of points is

# read (do not forget that each data point will by default be made a

# double of 8 bytes, no matter what the original data format is)

chunkPtsPerChan = floor(chunk \* 2 \*\* 20 / 8 / h["nADCNumChannels"])

chunkPts = chunkPtsPerChan \* h["nADCNumChannels"]

# the number of those chunks..

nChunk = floor(h["dataPts"] / chunkPts)

# ..and the remainder

restPts = h["dataPts"] - nChunk \* chunkPts

restPtsPerChan = restPts / h["nADCNumChannels"]

# chunkwise row indices into d

dix = range(1,chunkPtsPerChan,h["dataPtsPerChan"])

dix[:, 2] = dix[:, 1] + chunkPtsPerChan - 1

dix[-1, 2] = h["dataPtsPerChan"]

# if verbose and nChunk:

# printmcat(['reading file in '), int2str(nChunk), ' chunks of ~'), num2str(chunk), ' Mb')]))

# do it

for ci in mslice[1:size(dix, 1) - 1]:

tmpd = struct.unpack(precision, fid.read(chunkPts))

# if n != chunkPts:

# fid.close()

# print mcat(['something went wrong reading chunk #'), int2str(ci], ' ('), int2str(chunkPts), ' points should have been read, '), int2str(n), ' points actually read')]))

# separate channels..

tmpd = tmpd.reshape((h["nADCNumChannels"], chunkPtsPerChan))

d[dix[ci, 1]:dix[ci, 2], :] = tmpd[chInd, :].cT

# collect the rest, if any

if restPts:

tmpd = struct.unpack(precision, fid.read(restPts))

# if n != restPts:

# fid.close()

# print mcat(['something went wrong reading last chunk ('), int2str(restPts), ' points should have been read, '), int2str(n), ' points actually read')]))

# separate channels..

tmpd = tmpd.reshape((h["nADCNumChannels"], restPtsPerChan))

d[dix[-1, 1]:dix[-1, 2], :] = tmpd[chInd, :].cT

else:

# --- situation [i]

#d = struct.unpack(precision, fid.read(h["dataPts"])

d = np.fromfile(file=fid, dtype=precision, count = h["dataPts"]).reshape((h["nADCNumChannels"], h["dataPtsPerChan"]))

# if n != h["dataPts"]:

# fid.close()

# print mcat(['something went wrong reading file ('), int2str(h["dataPts"]), ' points should have been read, '), int2str(n), ' points actually read')]))

# separate channels..

# d = reshape(d,

#d = d'

# if data format is integer, scale appropriately, if it's 'f', d is fine

if not h["nDataFormat"]:

for j in range(len(chInd)):

ch = recChIdx[chInd[j]] + 1

scaling\_factor = 1.0/ ((h["fInstrumentScaleFactor"][ch] \* h["fSignalGain"][ch] \* h["fADCProgrammableGain"][ch] \* addGain[ch]) \* h["fADCRange"] / h["lADCResolution"] + h["fInstrumentOffset"][ch] - h["fSignalOffset"][ch])

print "scaling data with scaling factor of %f" % scaling\_factor

print d

tmp = np.empty(d.shape, dtype=np.float64)

tmp[:][j] = d[:][j] / (h["fInstrumentScaleFactor"][ch] \* h["fSignalGain"][ch] \* h["fADCProgrammableGain"][ch] \* addGain[ch]) \* h["fADCRange"] / h["lADCResolution"] + h["fInstrumentOffset"][ch] - h["fSignalOffset"][ch]

d = tmp

print d

else:

print'recording mode is \'high-speed oscilloscope\' which is not implemented -- returning empty matrix'

d = []

h["si"] = []

fid.close()

return d, h

def constSections(self):

Sections =['ProtocolSection',

'ADCSection',

'DACSection',

'EpochSection',

'ADCPerDACSection',

'EpochPerDACSection',

'UserListSection',

'StatsRegionSection',

'MathSection',

'StringsSection',

'DataSection',

'TagSection',

'ScopeSection',

'DeltaSection',

'VoiceTagSection',

'SynchArraySection',

'AnnotationSection',

'StatsSection']

return Sections

def define\_header(self,fileSig=None):

if 'ABF' in fileSig: # \*\* note the blank

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# abf version < 2.0

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#

# temporary initializing var

tmp = repmat(-1, 1, 16)

# define vital header parameters and initialize them with -1: set up a cell

# array (and convert it to a struct below, which is more convenient)

# column order is

# name, position in header in bytes, type, value)

headpar = {

'fFileSignature':(0,'c',(4,1)),

'fFileVersionNumber':(4,'f',(1,1)),

'nOperationMode':(8,'i2',(1,1)),

'lActualAcqLength':(10,'i',(1,1)),

'nNumPointsIgnored':(14,'i2',(1,1)),

'lActualEpisodes':(16,'i',(1,1)),

'lFileStartTime':(24,'i',(1,1)),

'lDataSectionPtr':(40,'i',(1,1)),

'lTagSectionPtr':(44,'i',(1,1)),

'lNumTagEntries':(48,'i',(1,1)),

'lSynchArrayPtr':(92,'i',(1,1)),

'lSynchArraySize':(96,'i',(1,1)),

'nDataFormat':(100,'i2',(1,1)),

'nADCNumChannels':(120,'i2',(1,1)),

'fADCSampleInterval':(122,'f',(1,1)),

'fSynchTimeUnit':(130,'f',(1,1)),

'lNumSamplesPerEpisode':(138,'i',(1,1)),

'lPreTriggerSamples':(142,'i',(1,1)),

'lEpisodesPerRun':(146,'i',(1,1)),

'fADCRange':(244,'f',(1,1)),

'lADCResolution':(252,'i',(1,1)),

'nFileStartMillisecs':(366,'i2',(1,1)),

'nADCPtoLChannelMap':(378,'i2',(1,16)),

'nADCSamplingSeq':(410,'i2',(1,16)),

'sADCChannelName':(442,'u1',(1, 160)),

'sADCUnits':(602,'u1',(1, 128)),

'fADCProgrammableGain':(730,'f',(1,16)),

'fInstrumentScaleFactor':(922,'f',(1,16)),

'fInstrumentOffset':(986,'f',(1,16)),

'fSignalGain':(1050,'f',(1,16)),

'fSignalOffset':(1114,'f',(1,16)),

'nTelegraphEnable':(4512,'i2',(1,16)),

'fTelegraphAdditGain':(4576,'f',(1,16)) }

elif 'ABF2' in fileSig:

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# abf version >= 2.0

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

headpar = {

'fFileSignature':(1,'S',(1,4)),

'fFileVersionNumber':(4,'i',(1,4)), #bit8=>int ???

'uFileInfoSize':(8,'I',-(1,1)),

'lActualEpisodes':(12,'I',(1,1)),

'uFileStartDate':(16,'I',(1,1)),

'uFileStartTimeMS':(20,'I',(1,1)),

'uStopwatchTime':(24,'I',(1,1)),

'nFileType':(28,'s',(1,1)),

'nDataFormat':(30,'s',(1,1)),

'nSimultaneousScan':(32,'s',(1,1)),

'nCRCEnable':(34,'s',(1,1)),

'uFileCRC':(36,'I',(1,1)),

'FileGUID':(40,'I',(1,1)),

'uCreatorVersion':(56,'I',(1,1)),

'uCreatorNameIndex':(60,'I',(1,1)),

'uModifierVersion':(64,'I',(1,1)),

'uModifierNameIndex':(68,'I',(1,1)),

'uProtocolPathIndex':(72,'I',(1,1))}

return headpar

def constProtocolInfo(self):

ProtocolInfo={

'nOperationMode':('s',1),

'fADCSequenceInterval':('f',1),

'bEnableFileCompression':('?',1),

'sUnused1':('c',3),

'uFileCompressionRatio':('I',1),

'fSynchTimeUnit':('f',1),

'fSecondsPerRun':('f',1),

'lNumSamplesPerEpisode':('i',1),

'lPreTriggerSamples':('i',1),

'lEpisodesPerRun':('i',1),

'lRunsPerTrial':('i',1),

'lNumberOfTrials':('i',1),

'nAveragingMode':('s',1),

'nUndoRunCount':('s',1),

'nFirstEpisodeInRun':('s',1),

'fTriggerThreshold':('f',1),

'nTriggerSource':('s',1),

'nTriggerAction':('s',1),

'nTriggerPolarity':('s',1),

'fScopeOutputInterval':('f',1),

'fEpisodeStartToStart':('f',1),

'fRunStartToStart':('f',1),

'lAverageCount':('i',1),

'fTrialStartToStart':('f',1),

'nAutoTriggerStrategy':('s',1),

'fFirstRunDelayS':('f',1),

'nChannelStatsStrategy':('s',1),

'lSamplesPerTrace':('i',1),

'lStartDisplayNum':('i',1),

'lFinishDisplayNum':('i',1),

'nShowPNRawData':('s',1),

'fStatisticsPeriod':('f',1),

'lStatisticsMeasurements':('i',1),

'nStatisticsSaveStrategy':('s',1),

'fADCRange':('f',1),

'fDACRange':('f',1),

'lADCResolution':('i',1),

'lDACResolution':('i',1),

'nExperimentType':('s',1),

'nManualInfoStrategy':('s',1),

'nCommentsEnable':('s',1),

'lFileCommentIndex':('i',1),

'nAutoAnalyseEnable':('s',1),

'nSignalType':('s',1),

'nDigitalEnable':('s',1),

'nActiveDACChannel':('s',1),

'nDigitalHolding':('s',1),

'nDigitalInterEpisode':('s',1),

'nDigitalDACChannel':('s',1),

'nDigitalTrainActiveLogic':('s',1),

'nStatsEnable':('s',1),

'nStatisticsClearStrategy':('s',1),

'nLevelHysteresis':('s',1),

'lTimeHysteresis':('i',1),

'nAllowExternalTags':('s',1),

'nAverageAlgorithm':('s',1),

'fAverageWeighting':('f',1),

'nUndoPromptStrategy':('s',1),

'nTrialTriggerSource':('s',1),

'nStatisticsDisplayStrategy':('s',1),

'nExternalTagType':('s',1),

'nScopeTriggerOut':('s',1),

'nLTPType':('s',1),

'nAlternateDACOutputState':('s',1),

'nAlternateDigitalOutputState':('s',1),

'fCellID':('f',3),

'nDigitizerADCs':('s',1),

'nDigitizerDACs':('s',1),

'nDigitizerTotalDigitalOuts':('s',1),

'nDigitizerSynchDigitalOuts':('s',1),

'nDigitizerType':('s',1),

}

return ProtocolInfo

def constADCInfo(self):

ADCInfo={

'nADCNum':('s',1),

'nTelegraphEnable':('s',1),

'nTelegraphInstrument':('s',1),

'fTelegraphAdditGain':('f',1),

'fTelegraphFilter':('f',1),

'fTelegraphMembraneCap':('f',1),

'nTelegraphMode':('s',1),

'fTelegraphAccessResistance':('f',1),

'nADCPtoLChannelMap':('s',1),

'nADCSamplingSeq':('s',1),

'fADCProgrammableGain':('f',1),

'fADCDisplayAmplification':('f',1),

'fADCDisplayOffset':('f',1),

'fInstrumentScaleFactor':('f',1),

'fInstrumentOffset':('f',1),

'fSignalGain':('f',1),

'fSignalOffset':('f',1),

'fSignalLowpassFilter':('f',1),

'fSignalHighpassFilter':('f',1),

'nLowpassFilterType':('c',1),

'nHighpassFilterType':('c',1),

'fPostProcessLowpassFilter':('f',1),

'nPostProcessLowpassFilterType':('c',1),

'bEnabledDuringPN':('?',1),

'nStatsChannelPolarity':('s',1),

'lADCChannelNameIndex':('i',1),

'lADCUnitsIndex':('i',1),

}

return ADCInfo

def constTagInfo(self):

TagInfo={

'lTagTime':('i',1),

'sComment':('c',56),

'nTagType':('s',1),

'nVoiceTagNumber\_or\_AnnotationIndex':('s',1)

}

return TagInfo

def ReadSection(fid,offset,Format):

#s=cell2struct(Format,{'name','numType','number'},2),

fid.seek(offset)

Section ={}

for k,v in Format.items(): # i=1:length(s)

Section[k] =struct.unpack(v[1],fid.read(len(Format)))

return Section

def ReadSectionInfo(fid,offset):

fid.seek(offset)

SectionInfo = {}

SectionInfo["uBlockIndex"]=struct.unpack('I',fid.read(4))

fid.seek(offset+4)

SectionInfo["uBytes"]=struct.unpack("I",fid.read(4))#,''I''),

fid.seek(offset+8)

SectionInfo["llNumEntries"]=struct.unpack("f",fid.read(8))

return SectionInfo

def repmat(a,m,n):

from scipy import r\_, c\_

a = eval('r\_['+m\*'a,'+']')

return eval('c\_['+n\*'a,'+']')

def char2str(list, format = 'c'):

import array

return array.array(format, list).tostring()

#Find: \ +'(\w+)'\)\ \*\n\ +(\d+)\ \*\n\ +'(\w+)'\)\ \*\n\ +(-\*\d+),

# Replace: '$1':\($2,$3,$4\),\n

if \_\_name\_\_ == '\_\_main\_\_':

f = '/Users/hd/Documents/DataBase/invivocortex/2009\_08\_11\_0001.abf'

abfreader = Abf(f)

d = abfreader.abfload()

from MultiLinePlot import MultiLinePlot

try:

pc = MultiLinePlot(np.squeeze(d), 1000)

except BaseException, e:

print e