1D line_patterns_odd

line_patterns is a principal version of 1st-level 1D algorithm

Operations:

- Cross-compare consecutive pixels within each row of image, forming dert_: queue of derts, each a tuple of derivatives per pixel. dert_ is then segmented into patterns Pms and Pds: contiguous sequences of pixels forming sign match or difference. Initial match is inverse deviation of variation: m = ave_|d| - |d|, rather than minimum for directly defined match: albedo or intensity of reflected light doesn't correlate with predictive value of the objec reflects it.

- Match patterns Pms are spans of inputs forming same-sign match. Positive Pms contain high-match pixels, which are likely to match more distant pixels. Thus, positive Pms are evaluated for cross-comp of pixels over incre range.

- Difference patterns Pds are spans of inputs forming same-sign ds. d sign match is a precondition for d match, so only same-sign spans (Pds) are evaluated for cross-comp of constituent differences, which forms higher deri (d match = min: rng+ comp value: predictive value of difference is proportional to its magnitude, although inversely so)

  Both extended cross-comp forks are recursive: resulting sub-patterns are evaluated for deeper cross-comp, same as top patterns.
  These forks here are exclusive per P to avoid redundancy, but they overlap in line_patterns_olp.

  Initial bilateral cross-comp here is 1D slice of 2D 3x3 kernel, while unilateral d is a slice of 2x2 kernel.
  Odd kernels preserve resolution of pixels, while 2x2 kernels preserve resolution of derivatives, in resulting derts.
  The former should be used in rng_comp and the latter in der_comp, which may alternate with intra_P.

In [114]:
```python
# add ColAlg folder to system path
import sys
from os.path import dirname, join, abspath
sys.path.insert(0, abspath(join(dirname("CogAlg"), '..')))

import cv2
import argparse
from time import time
from utils import *
from itertools import zip_longest
from frame_2D_alg.class_cluster import ClusterStructure, NoneType, comp_param, Cdm
import csv # +++
```

In [115]:
```
class Cdert(ClusterStructure):
    p = int
    d = int
    m = int

class CP(ClusterStructure):
    sign = bool
    L = int
    I = int
    D = int
    M = int
    x0 = int
    dert_ = list
    sublayers = list
    # for line_PPs
    derP = object  # forward comp_P derivatives
    _smP = bool  # backward mP sign, for derP.sign determination, not needed thereafter
    fPd = bool  # P is Pd if true, else Pm
```

In [116]:
```
# pattern filters or hyper-parameters: eventually from higher-level feedback, initialized here as constants:

ave = 15  # |difference| between pixels that coincides with average value of Pm
ave_min = 2  # for m defined as min |d|: smaller?
ave_M = 50  # min M for initial incremental-range comparison(t_), higher cost than der_comp?
ave_D = 5  # min |D| for initial incremental-derivation comparison(d_)
ave_nP = 5  # average number of sub_Ps in P, to estimate intra-costs? ave_rdn_inc = 1 + 1 / ave_nP # 1.2
ave_rdm = .5  # average dm / m, to project bi_m = m * 1.5
init_y = 0  # starting row, the whole frame doesn't need to be processed
```

Conventions:

- postfix '_' denotes array name, vs. same-name elements
- prefix '_' denotes prior of two same-name variables
- prefix 'f' denotes binary flag
- capitalized variables are normally summed same-letter small-case variables

In [117]:
```
def cross_comp(frame_of_pixels_):  # converts frame_of_pixels to frame_of_patterns, each pattern maybe nested

    Y, X = frame_of_pixels_.shape  # Y: frame height, X: frame width
    frame_of_patterns_ = []

    # put a brake point here, the code only needs one row to process
    for y in range(init_y + 1, Y):  # y is index of new line pixel_
        # initialization:
        pixel_ = frame_of_pixels_[y, :]
        dert_ = []
        __p, _p = pixel_[0:2]  # each prefix '_' denotes prior
        _d = _p - __p  # initial comparison
        _m = ave - abs(_d)
        dert_.append( Cdert(p=__p, d=0, m=(_m + _m / 2)))  # project _m to bilateral m, first dert is for comp_P only?

        for p in pixel_[2:]:  # pixel p is compared to prior pixel _p in a row
            d = p - _p
            m = ave - abs(d)  # initial match is inverse deviation of |difference|
            dert_.append( Cdert(p=_p, d=_d, m=m + _m))  # pack dert: prior p, prior d, bilateral match
            _p, _d, _m = p, d, m
        dert_.append( Cdert(p=_p, d=_d, m=(_m + _m / 2)))  # unilateral d, forward-project last m to bilateral m
```

```
        Pm_ = form_Pm_(dert_)  # forms m-sign patterns
        if len(Pm_) > 4:
            adj_M_ = form_adjacent_M_(Pm_)  # compute adjacent Ms to evaluate contrastive borrow potential
            intra_Pm_(Pm_, adj_M_, fid=False, rdn=1, rng=3)  # evaluates for sub-recursion per Pm

        frame_of_patterns_.append(Pm_)
        # line of patterns is added to frame of patterns

    return frame_of_patterns_  # frame of patterns will be output to level 2
```

In [118]:
```
def form_Pm_(P_dert_):  # initialization, accumulation, termination

    with open("frame_of_patterns_.csv", "a") as csvFile: # +++
        write = csv.writer(csvFile, delimiter=",")        # +++
        P_ = []  # initialization:
        dert = P_dert_[0]

        _sign = dert.m > 0
        L, I, D, M, dert_, sub_H, x = 1, dert.p, dert.d, dert.m, [dert], [], 0
        # cluster P_derts by m sign
        for dert in P_dert_[1:]:
            sign = dert.m > 0
            if sign != _sign:  # sign change, terminate P
                P_.append(CP(sign=_sign, L=L, I=I, D=D, M=M, x0=x-(L-1), dert_=dert_, sublayers=sub_H, _smP=False))
                # print(L, I, D, M, x-(L-1)) # +++
                write.writerow([L, I, D, M, x-(L-1)])  # +++

                L, I, D, M, dert_, sub_H = 0, 0, 0, 0, [], []  # reset params

            L += 1; I += dert.p; D += dert.d; M += dert.m  # accumulate params, bilateral m: for eval per pixel
            dert_ += [dert]
            _sign = sign
            x += 1

        P_.append(CP(sign=_sign, L=L, x0=x-(L-1), I=I, D=D, M=M, dert_=dert_, sublayers=sub_H, _smP=False))  # incomplete P
        return P_
```

In [119]:
```
def form_Pd_(P_dert_):  # cluster by d sign, within -Pms: min neg m spans

    P_ = []  # initialization:
    dert = P_dert_[1]  # skip dert_[0]: d is None
    _sign = dert.d > 0
    L, I, D, M, dert_, sub_H, x = 1, dert.p, 0, dert.m, [dert], [], 0
    # cluster P_derts by d sign
    for dert in P_dert_[2:]:
        sign = dert.d > 0
        if sign != _sign:  # sign change, terminate P
            P_.append(CP(sign=_sign, L=L, I=I, D=D, M=M, x0=x-(L-1), dert_=dert_, sublayers=sub_H, _smP=False, fPd=True))
            L, I, D, M, dert_, sub_H = 0, 0, 0, 0, [], []  # reset accumulated params

        L += 1; I += dert.p; D += dert.d; M += dert.m  # accumulate params, m for eval per pixel is bilateral
        dert_ += [dert]
        _sign = sign

    P_.append(CP(sign=_sign, x0=x-(L-1), L=L, I=I, D=D, M=M, dert_=dert_, sublayers=sub_H, _smP=False, fPd=True))  # incomplete P
    return P_
```

In [120]:
```
def form_adjacent_M_(Pm_):  # compute array of adjacent Ms, for contrastive borrow evaluation
    '''
    Value is projected match, while variation has contrast value only: it matters to the extent that it interrupts adjacent match: adj_M.
```

```
    In noise, there is a lot of variation. but no adjacent match to cancel, so variation in noise has no predictive value.
    On the other hand, we may have a 2D outline or 1D contrast with low gradient / difference, but it terminates adjacent uniform span.
    That contrast may be salient if it can borrow sufficient predictive value from that adjacent high-match span.
    '''

    pri_M = Pm_[0].M  # comp_g value is borrowed from adjacent opposite-sign Ms
    M = Pm_[1].M
    adj_M_ = [abs(Pm_[1].M)]  # initial next_M, no / 2: projection for first P, abs for bilateral adjustment

    for Pm in Pm_[2:]:
        next_M = Pm.M
        adj_M_.append((abs(pri_M / 2) + abs(next_M / 2)))  # exclude M
        pri_M = M
        M = next_M
    adj_M_.append(abs(pri_M))  # no / 2: projection for last P

    return adj_M_
```

Recursion in intra_P extends pattern with sub_: hierarchy of sub-patterns, to be adjusted by macro-feedback:

P:

- sign, # of m | d
- dert_, # buffer of elements, input for extended cross-comp

next fork:

- fPd, # flag: select Pd vs. Pm forks in form_P_
- fid, # flag: input is derived: magnitude correlates with predictive value: m = min-ave, else m = ave-|d|
- rdn, # redundancy to higher layers, possibly lateral overlap of rng+ & der+, rdn += 1 * typ coef?
- rng, # comp range
- sublayers:
  - multiple layers of sub_P_s from d segmentation or extended comp, nested to depth = sub_[n]
  - for layer-parallel access and comp, as in frequency domain representation
  - orders of composition: 1st: dert_, 2nd: sub_P_[derts], 3rd: sublayers[sub_Ps[derts]]

In [121]:
```
def intra_Pm_(P_, adj_M_, fid, rdn, rng):  # evaluate for sub-recursion in line Pm_, pack results into sub_Pm_

    comb_layers = []  # combine into root P sublayers[1:]
    for P, adj_M in zip(P_, adj_M_):  # each sub_layer is nested to depth = sublayers[n]

        if P.sign:  # +Pm: low-variation span, eval comp at rng=2^n: 2, 4., kernel: 5, 9., rng=1 cross-comp is kernels 2 and 3
            if P.M - adj_M > ave_M * rdn and P.L > 4:  # reduced by lending to contrast: all comps form params for hLe comp?
                '''
                if localized filters:
                P_ave = (P.M - adj_M) / P.L
                loc_ave = (ave - P_ave) / 2  # ave is reduced because it's for inverse deviation, possibly negative?
                loc_ave_min = (ave_min + P_ave) / 2
                rdert_ = range_comp(P.dert_, loc_ave, loc_ave_min, fid)
                '''
                rdert_ = range_comp(P.dert_, fid)  # rng+ comp with localized ave, skip predictable next dert
                sub_Pm_ = form_Pm_(rdert_)  # cluster by m sign
                Ls = len(sub_Pm_)
                P.sublayers += [[(Ls, False, fid, rdn, rng, sub_Pm_, [], [])]]  # sub_PPm_, sub_PPd_
                # 1st layer, Dert=[], fill if Ls > min?
                if len(sub_Pm_) > 4:
                    sub_adj_M_ = form_adjacent_M_(sub_Pm_)
```

```
                P.sublayers += intra_Pm_(sub_Pm_, sub_adj_M_, fid, rdn + 1 + 1 / Ls, rng * 2 + 1)  # feedback
                # add param summation within sublayer, for comp_sublayers?
                # splice sublayers across sub_Ps:
                comb_layers = [comb_layers + sublayers for comb_layers, sublayers in
                               zip_longest(comb_layers, P.sublayers, fillvalue=[])]

        else:  # -Pm: high-variation span, min neg M is contrast value, borrowed from adjacent +Pms:
            if min(-P.M, adj_M) > ave_D * rdn and P.L > 3:  # cancelled M+ val, M = min | ~v_SAD

                rel_adj_M = adj_M / -P.M  # for allocation of -Pm' adj_M to each of its internal Pds
                sub_Pd_ = form_Pd_(P.dert_)  # cluster by input d sign match: partial d match
                Ls = len(sub_Pd_)
                P.sublayers += [[(Ls, True, 1, rdn, rng, sub_Pd_)]]  # 1st layer, Dert=[], fill if Ls > min?

                P.sublayers += intra_Pd_(sub_Pd_, rel_adj_M, rdn + 1 + 1 / Ls, rng + 1)  # der_comp eval per nPm
                # splice sublayers across sub_Ps, for return as root sublayers[1:]:
                comb_layers = [comb_layers + sublayers for comb_layers, sublayers in
                               zip_longest(comb_layers, P.sublayers, fillvalue=[])]

    return comb_layers
```

In [122]:
```
def intra_Pd_(Pd_, rel_adj_M, rdn, rng):  # evaluate for sub-recursion in line P_, packing results in sub_P_

    comb_layers = []
    for P in Pd_:  # each sub in sub_ is nested to depth = sub_[n]
        if min(abs(P.D), abs(P.D) * rel_adj_M) > ave_D * rdn and P.L > 3:  # abs(D) * rel_adj_M: allocated adj_M
            # if fid: abs(D), else: M + ave*L: complementary m is more precise than inverted diff?

            ddert_ = deriv_comp(P.dert_)  # cross-comp of uni_ds
            sub_Pm_ = form_Pm_(ddert_)  # cluster Pd derts by md, won't happen
            Ls = len(sub_Pm_)
            P.sublayers += [[(Ls, 1, 1, rdn, rng, sub_Pm_, [], [] )]]  # sub_PPm_, sub_PPd_
            # 1st layer: Ls, fPd, fid, rdn, rng, sub_P_
            if len(sub_Pm_) > 3:
                sub_adj_M_ = form_adjacent_M_(sub_Pm_)
                P.sublayers += intra_Pm_(sub_Pm_, sub_adj_M_, 1, rdn + 1 + 1 / Ls, rng + 1)
                # splice sublayers across sub_Ps:
                comb_layers = [comb_layers + sublayers for comb_layers, sublayers in
                               zip_longest(comb_layers, P.sublayers, fillvalue=[])]
    '''
    adj_M is not affected by primary range_comp per Pm?
    no comb_m = comb_M / comb_S, if fid: comb_m -= comb_|D| / comb_S: alt rep cost
    same-sign comp: parallel edges, cross-sign comp: M - (~M/2 * rL) -> contrast as 1D difference?
    '''
    return comb_layers
```

In [123]:
```
def range_comp(dert_, fid):  # skip odd derts for sparse rng+ comp: 1 skip / 1 add, to maintain 2x overlap

    rdert_ = []  # prefix '_' denotes the prior of same-name variables, initialization:
    __dert = dert_[0]  # prior-prior dert
    __i = __dert.p
    _dert = dert_[2]  # initialize _dert with sparse p_, skipping odd ps
    _i = _dert.p
    _short_rng_d = _dert.d
    _short_rng_m = _dert.m

    _d = _i - __i
    if fid:  # flag: input is d, from deriv_comp
        _m = min(__i, _i) - ave_min
    else:
        _m = ave - abs(_dert.d)  # no ave * rng: m and d value is cumulative
```

```
    _rng_m = (_m + _m / 2) + __dert.m  # back-project missing m as _m / 2: induction decays with distance
    rdert_.append(Cdert(p=__i, d=0, m=_rng_m))  # no _rng_d = _d + __short_rng_d

    for n in range(4, len(dert_), 2):  # backward comp

        dert = dert_[n]
        i = dert.p
        short_rng_d = dert.d
        short_rng_m = dert.m
        d = i - _i
        if fid:
            m = min(i, _i) - ave_min  # match = min: magnitude of derived vars correlates with stability
        else:
            m = ave - abs(d)  # inverse match: intensity doesn't correlate with stability
        rng_d = _d + _short_rng_d  # difference accumulated in rng
        rng_m = _m + m + _short_rng_m  # bilateral match accumulated in rng
        rdert_.append(Cdert(p=_i, d=rng_d, m=rng_m))
        _i, _d, _m, _short_rng_d, _short_rng_m = \
            i, d, m, short_rng_d, short_rng_m

    rdert_.append(Cdert(p=_i, d=_d + _short_rng_d, m=(_m + _m / 2) + _short_rng_m))  # forward-project _m to bilateral m
    return rdert_
```

In [124]:
```
def deriv_comp(dert_):  # cross-comp consecutive uni_ds in same-sign dert_: sign match is partial d match
    # dd and md may match across d sign, but likely in high-match area, spliced by spec in comp_P?

    ddert_ = []  # initialization:
    __i = dert_[1].d  # each prefix '_' denotes prior
    _i = dert_[2].d

    __i = abs(__i);  _i = abs(_i)
    _d = _i - __i  # initial comp
    _m = min(__i, _i) - ave_min
    ddert_.append(Cdert(p=_i, d=0, m=(_m + _m / 2)))  # no __d, back-project __m = _m * .5

    for dert in dert_[3:]:
        i = abs(dert.d)  # unilateral d, same sign in Pd
        d = i - _i  # d is dd
        m = min(i, _i) - ave_min  # md = min: magnitude of derived vars corresponds to predictive value
        ddert_.append(Cdert(p=_i, d=_d, m=_m + m))  # unilateral _d and bilateral m per _i
        _i, _d, _m = i, d, m

    ddert_.append(Cdert(p=_i, d=_d, m=(_m + _m / 2)))  # forward-project bilateral m
    return ddert
```

In [125]:
```
def cross_comp_spliced(frame_of_pixels_):  # converts frame_of_pixels to frame_of_patterns, each pattern maybe nested
    '''
    process all image rows as a single line, vertically consecutive and preserving horizontal direction
    '''
    Y, X = frame_of_pixels_.shape  # Y: frame height, X: frame width
    pixel__ = []

    for y in range(init_y + 1, Y):  # y is index of new line
        pixel__.append([ frame_of_pixels_[y, :] ])  # splice all rows into pixel__

    # initialization:
    dert_ = []
    __p, _p = pixel__[0:2]  # each prefix '_' denotes prior
    _d = _p - __p  # initial comparison
    _m = ave - abs(_d)
    dert_.append( Cdert(p=__p, d=0, m=(_m + _m / 2)))  # project _m to bilateral m, first dert is for comp_P only?
```

```python
    for p in pixel__[2:]:  # pixel p is compared to prior pixel _p in a row
        d = p - _p
        m = ave - abs(d)  # initial match is inverse deviation of |difference|
        dert_.append( Cdert(p=_p, d=_d, m=m + _m))  # pack dert: prior p, prior d, bilateral match
        _p, _d, _m = p, d, m
    dert_.append( Cdert(p=_p, d=_d, m=(_m + _m / 2)))  # unilateral d, forward-project last m to bilateral m

    Pm_ = form_Pm_(dert_)  # forms m-sign patterns
    if len(Pm_) > 4:
        adj_M_ = form_adjacent_M_(Pm_)  # compute adjacent Ms to evaluate contrastive borrow potential
        intra_Pm_(Pm_, adj_M_, fid=False, rdn=1, rng=3)  # evaluates for sub-recursion per Pm

    return Pm_  # frame of patterns, an output to line_PPs (level 2 processing)
```

In [132]:

```python
# if __name__ == "__main__":
    # Parse argument (image)
#     argument_parser = argparse.ArgumentParser()
#     argument_parser.add_argument('-i', '--image',
#                                  help='path to image file',
#                                  default='.//raccoon.jpg')
#     arguments = vars(argument_parser.parse_args())
    # Read image
#     image = cv2.imread(arguments['image'], 0).astype(int)  # load pix-mapped image

# +++
# to show image in the same window as a code
%matplotlib inline

image = cv2.imread('.//raccoon.jpg', 0).astype(int)  # manual load pix-mapped image
                                        # instead of arguments parsing for image load
plt.imshow(image, cmap='gray') # show the image below in gray
plt.show()
#+++
```

In [133]:

```python
from pprint import pprint # +++

with open("frame_of_patterns_.csv", "a") as csvFile:  # +++
    write = csv.writer(csvFile, delimiter=",")        # +++
    fieldnames = ("L=", "I=", "D=", "M=", "x0=")      # +++
    write.writerow(fieldnames)                        # +++

assert image is not None, "No image in the path"
image = image.astype(int)
```

```
start_time = time()
# Main
frame_of_patterns_ = cross_comp(image)  # returns Pm__
pprint(frame_of_patterns_) # to show the output results in the convinient way +++

fline_PPs = 0
if fline_PPs:  # debug line_PPs_draft
    from line_PPs_draft import *
    frame_PP_ = []

    for y, P_ in enumerate(frame_of_patterns_):
        PPm_, PPd_ = search(P_)
        frame_PP_.append([PPm_, PPd_])

end_time = time() - start_time
```

```
[[CP(L=2, I=196, D=22, M=-22.5, x0=0),
  CP(L=34, I=3058, D=-50, M=607, x0=2),
  CP(L=1, I=76, D=17, M=-4, x0=36),
  CP(L=15, I=1813, D=74, M=212, x0=37),
  CP(L=1, I=169, D=19, M=-10, x0=52),
  CP(L=3, I=574, D=20, M=26, x0=53),
  CP(L=4, I=574, D=-72, M=-16, x0=56),
  CP(L=22, I=2201, D=-29, M=352, x0=60),
  CP(L=5, I=295, D=10, M=-101, x0=82),
  CP(L=16, I=1385, D=-23, M=209, x0=87),
  CP(L=1, I=94, D=19, M=-12, x0=103),
  CP(L=8, I=984, D=48, M=143, x0=104),
  CP(L=1, I=122, D=-20, M=-2, x0=112),
  CP(L=4, I=375, D=-44, M=32, x0=113),
  CP(L=3, I=359, D=69, M=-45, x0=117),
  CP(L=2, I=308, D=5, M=30, x0=120),
  CP(L=2, I=257, D=-34, M=-12, x0=122),
  CP(L=4, I=393, D=-18, M=55, x0=124),
  CP(L=1, I=126, D=26, M=-7, x0=128),
```

In [134]:
```
print(end_time)
```

```
38.11854338645935
```

My questions and propositions:

1. Is it necessary to store the result of the 1D data processing (frame_of_patterns_) in such a verbose format?
2. Is it acceptable to use for this purpose, for example, a simple array, or pandas dataframe?

An example of such realization is listed below:

In [135]:
```
import pandas as pd
dataframe1 = pd.read_csv("frame_of_patterns_.csv") # this file has been previously created
                                            # in the modified form_Pm_ function
dataframe1
```

Out[135]:

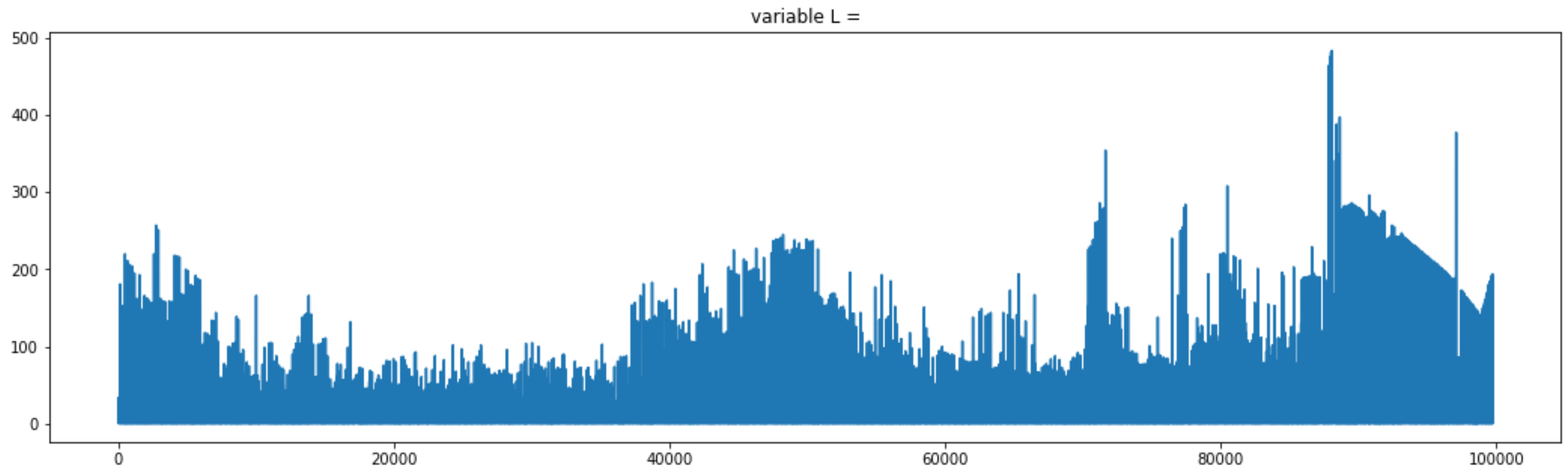
	L=	I=	D=	M=	x0=
0	2	196	22	-22.5	0
1	34	3058	-50	607.0	2
2	1	76	17	-4.0	36

	L=	I=	D=	M=	x0=
3	15	1813	74	212.0	37
4	1	169	19	-10.0	52
...	...	...	...	...	...
99728	7	1335	-9	246.0	0
99729	1	118	0	4.0	0
99730	1	137	36	-5.0	1
99731	6	662	-30	230.5	0
99732	1	81	-35	-2.0	6

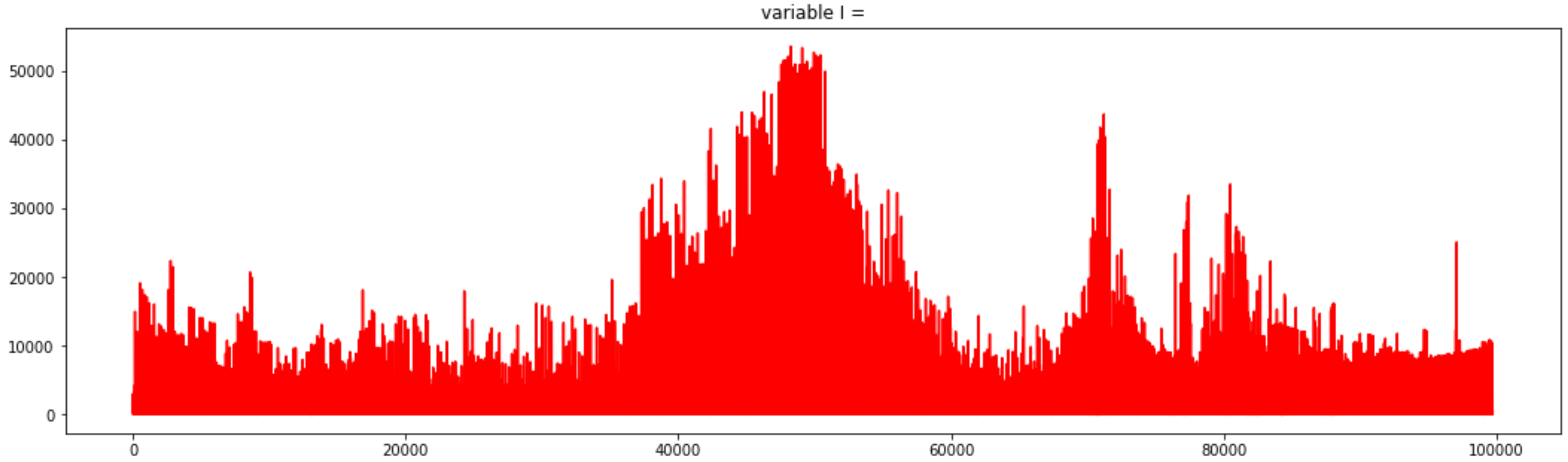
As you can see, the content of the array above corresponds to the content of the frame_of_patterns_ structure. But now it is more convenient to visualize data. In the first attempt it could look like this:

In [138]:
```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (18,5)
plt.plot(dataframe1.iloc[:,0])
plt.title("variable L =")
```

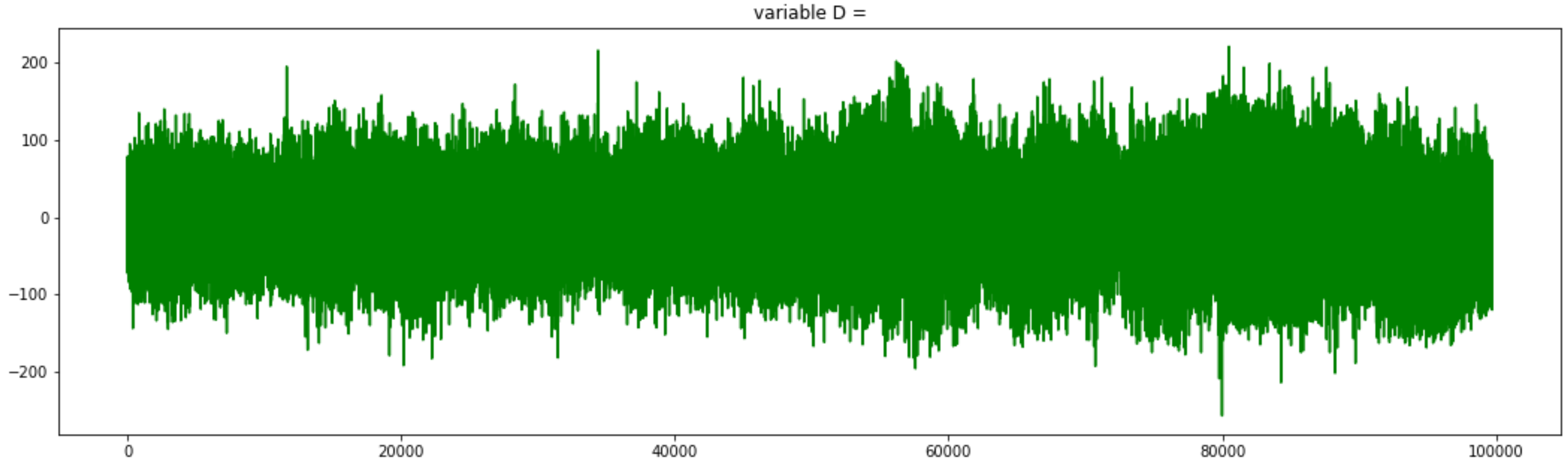
Out[138]: `Text(0.5, 1.0, 'variable L =')`



In [139]:
```
plt.plot(dataframe1.iloc[:,1], color = 'red')
plt.title("variable I =")
```
Out[139]: Text(0.5, 1.0, 'variable I =')

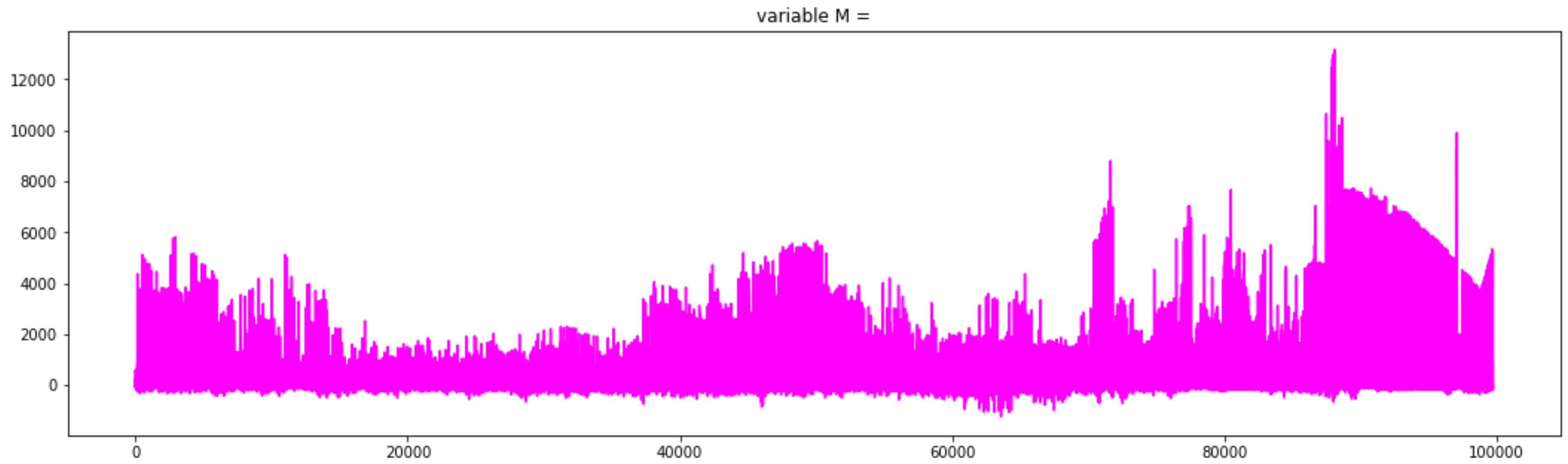


In [140]:
```
plt.plot(dataframe1.iloc[:,2], color = 'green')
plt.title("variable D =")
```
Out[140]: Text(0.5, 1.0, 'variable D =')



In [141]:
```
plt.plot(dataframe1.iloc[:,3], color = 'magenta')
plt.title("variable M =")
```

Out[141]: Text(0.5, 1.0, 'variable M =')



In [142]:
```
plt.plot(dataframe1.iloc[:,4], color = 'yellow')
plt.title("variable x0 =")
```

Out[142]: Text(0.5, 1.0, 'variable x0 =')