## line_Ps 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 same-sign match or difference. Initial match is inverse deviation of variation: m = ave_|d| - |d|, rather than a minimum for directly defined match: albedo of an object doesn't correlate with its predictive value.
- 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 incremented 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 derivatives. (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 do overlap in line_patterns_olp.

In [ ]:
```
# add ColAlg folder to system path
import sys
from os.path import dirname, join, abspath

from numpy import int16, int32
sys.path.insert(0, abspath(join(dirname("CogAlg"), '..')))
import cv2
# import argparse
import pickle
from time import time
from matplotlib import pyplot as plt
from itertools import zip_longest
from frame_2D_alg.class_cluster import ClusterStructure, NoneType, comp_param
```

In [ ]:
```
class Cdert(ClusterStructure):
    i = int  # input for range_comp only
    p = int  # accumulated in rng
    d = int  # accumulated in rng
    m = int  # distinct in deriv_comp only
    mrdn = lambda: 1.0  # -> Rdn: rdn counter per P
```

In [ ]:
```
class CP(ClusterStructure):
    L = int
    I = int
    D = int
    M = int  # summed ave - abs(d), different from D
    Rdn = lambda: 1.0  # mrdn counter
    x0 = int
    dert_ = list  # contains (i, p, d, m, mrdn)
    subset = list  # 1st sublayer' rdn, rng, xsub_pmdertt_, _xsub_pddertt_, sub_Ppm_, sub_Ppd_
    # for layer-parallel access and comp, ~ frequency domain, composition: 1st: dert_, 2nd: sub_P_[ dert_], 3rd: sublayers[ sub_P_[ dert_]]:
    sublayers = list  # multiple layers of sub_P_s from d segmentation or extended comp, nested to depth = sub_[n]
    subDertt_ = list  # m,d' [L,I,D,M] per sublayer, conditionally summed in line_PPs
    derDertt_ = list  # for subDertt_s compared in line_PPs
```

In [ ]:
```
verbose = False
# 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 = 20  # 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  # obsolete: average dm / m, to project bi_m = m * 1.5
ave_splice = 50  # to merge a kernel of 3 adjacent Ps
init_y = 500  # starting row, set 0 for the whole frame, mostly not needed
halt_y = 502  # ending row, set 999999999 for arbitrary image
```

    Conventions:

- postfix 't' denotes tuple, multiple ts is a nested tuple
- postfix '_' denotes array name, vs. same-name elements
- prefix '_' denotes prior of two same-name variables
- prefix 'f' denotes flag
- 1-3 letter names are normally scalars, except for P and similar classes,
- capitalized variables are normally summed small-case variables,
- longer names are normally classes

In [ ]:
```
def line_Ps_root(pixel_):  # Ps: patterns, converts frame_of_pixels to frame_of_patterns, each pattern may be nested

    dert_ = []  # line-wide i_, p_, d_, m_, mrdn_
    _i = pixel_[0]
    # cross_comparison:
    for i in pixel_[1:]:  # pixel i is compared to prior pixel _i in a row:
        d = i - _i  # accum in rng
        p = i + _i  # accum in rng
        m = ave - abs(d)  # for consistency with deriv_comp output, else redundant
        mrdn = m + ave < abs(d)
        dert_.append( Cdert( i=i, p=p, d=d, m=m, mrdn=mrdn) )
        _i = i

    # form patterns, evaluate them for rng+ and der+ sub-recursion of cross_comp:
    Pm_ = form_P_(None, dert_, rdn=1, rng=1, fPd=False)  # rootP=None, eval intra_P_ (calls form_P_)
    Pd_ = form_P_(None, dert_, rdn=1, rng=1, fPd=True)

    return [Pm_, Pd_]  # input to 2nd level
```

In [ ]:

```python
def form_P_(rootP, dert_, rdn, rng, fPd):  # accumulation and termination, rdn and rng are pass-through to rng+ and der+
    # initialization:
    P_ = []
    x = 0
    _sign = None  # to initialize 1st P, (None != True) and (None != False) are both True

    for dert in dert_:  # segment by sign
        if fPd: sign = dert.d > 0
        else:   sign = dert.m > 0
        if sign != _sign:
            # sign change, initialize and append P
            P = CP( L=1, I=dert.p, D=dert.d, M=dert.m, Rdn=dert.mrdn+1, x0=x, dert_=[dert], sublayers=[])  # Rdn starts from 1
            P_.append(P)  # updated with accumulation below
        else:
            # accumulate params:
            P.L += 1; P.I += dert.p; P.D += dert.d; P.M += dert.m; P.Rdn += dert.mrdn
            P.dert_ += [dert]
        x += 1
        _sign = sign
    '''
    due to separate aves, P may be processed by both or neither of r fork and d fork
    add separate rsublayers and dsublayers?
    '''
    range_incr_P_(rootP, P_, rdn, rng)
    deriv_incr_P_(rootP, P_, rdn, rng)

    return P_  # used only if not rootP, else packed in rootP.sublayers
```

In [ ]:

```python
def range_incr_P_(rootP, P_, rdn, rng):

    comb_sublayers = []
    for P in P_:
        if P.M - P.Rdn * ave_M * P.L > ave_M * rdn and P.L > 2:  # M value adjusted for xP and higher-layers redundancy
            ''' P is Pm
            min skipping P.L=3, actual comp rng = 2^(n+1): 1, 2, 3 -> kernel size 4, 8, 16...
            if local ave:
            loc_ave = (ave + (P.M - adj_M) / P.L) / 2  # mean ave + P_ave, possibly negative?
            loc_ave_min = (ave_min + (P.M - adj_M) / P.L) / 2  # if P.M is min?
            rdert_ = range_comp(P.dert_, loc_ave, loc_ave_min, fid)
            '''
            rdn += 1; rng += 1
            P.subset = rdn, rng, [],[],[],[]  # 1st sublayer params, []s: xsub_pmdertt_, _xsub_pddertt_, sub_Ppm_, sub_Ppd_
            sub_Pm_, sub_Pd_ = [], []  # initialize layers, concatenate by intra_P_ in form_P_
            P.sublayers = [(sub_Pm_, sub_Pd_)]  # 1st layer
            rdert_ = []
            _i = P.dert_[0].i
            for dert in P.dert_[2::2]:  # all inputs are sparse, skip odd pixels compared in prior rng: 1 skip / 1 add to maintain 2x overlap
                # skip predictable next dert, local ave? add rdn to higher | stronger layers:
                d = dert.i - _i
                rp = dert.p + _i  # intensity accumulated in rng
                rd = dert.d + d  # difference accumulated in rng
                rm = dert.m + ave - abs(d)  # m accumulated in rng
                rmrdn = rm + ave < abs(rd)  # use Ave? for consistency with deriv_comp, else redundant
                rdert_.append(Cdert(i=dert.i, p=rp, d=rd, m=rm, mrdn=rmrdn))
                _i = dert.i
            sub_Pm_[:] = form_P_(P, rdert_, rdn, rng, fPd=False)  # cluster by rm sign
            sub_Pd_[:] = form_P_(P, rdert_, rdn, rng, fPd=True)  # cluster by rd sign

        if rootP and P.sublayers:
            new_comb_sublayers = []
            for (comb_sub_Pm_, comb_sub_Pd_), (sub_Pm_, sub_Pd_) in zip_longest(comb_sublayers, P.sublayers, fillvalue=([],[])):
                comb_sub_Pm_ += sub_Pm_  # remove brackets, they preserve index in sub_Pp root_
                comb_sub_Pd_ += sub_Pd_
                new_comb_sublayers.append((comb_sub_Pm_, comb_sub_Pd_))  # add sublayer
            comb_sublayers = new_comb_sublayers

    if rootP:
        rootP.sublayers += comb_sublayers  # no return
```

In [ ]:

```python
def deriv_incr_P_(rootP, P_, rdn, rng):

    comb_sublayers = []
    # adj_M_ = form_adjacent_M_(P_)  # compute adjacent Ms to evaluate contrastive borrow potential; but lend is not to adj only, reflected in ave?:
    # for P, adj_M in zip(P_, adj_M_); rel_adj_M = adj_M / -P.M  # allocate -Pm' adj_M in internal Pds; vs.:
    for P in P_:
        if abs(P.D) - (P.L - P.Rdn) * ave_D * P.L > ave_D * rdn and P.L > 1:  # high-D span, ave_adj_M is represented in ave_D
            rdn += 1; rng += 1
            P.subset = rdn, rng, [],[],[],[]  # 1st sublayer params, []s: xsub_pmdertt_, _xsub_pddertt_, sub_Ppm_, sub_Ppd_
            sub_Pm_, sub_Pd_ = [], []
            P.sublayers = [(sub_Pm_, sub_Pd_)]
            ddert_ = []
            _d = abs(P.dert_[0].d)
            for dert in P.dert_[1:]:  # all same-sign in Pd
                d = abs(dert.d)  # compare ds
                rd = d + _d
                dd = d - _d
                md = min(d, _d) - abs(dd / 2) - ave_min  # min_match because magnitude of derived vars corresponds to predictive value
                dmrdn = md + ave < abs(dd)  # use Ave?
                ddert_.append(Cdert(i=dert.d, p=rd, d=dd, m=md, dmrdn=dmrdn))
                _d = d
            sub_Pm_[:] = form_P_(P, ddert_, rdn, rng, fPd=False)  # cluster by mm sign
            sub_Pd_[:] = form_P_(P, ddert_, rdn, rng, fPd=True)  # cluster by md sign

        if rootP and P.sublayers:
            new_comb_sublayers = []
            for (comb_sub_Pm_, comb_sub_Pd_), (sub_Pm_, sub_Pd_) in zip_longest(comb_sublayers, P.sublayers, fillvalue=([],[])):
                comb_sub_Pm_ += sub_Pm_  # remove brackets, they preserve index in sub_Pp root_
                comb_sub_Pd_ += sub_Pd_
                new_comb_sublayers.append((comb_sub_Pm_, comb_sub_Pd_))  # add sublayer
            comb_sublayers = new_comb_sublayers

    if rootP:
        rootP.sublayers += comb_sublayers  # no return
```

In [ ]:

```
render = 0
fline_PPs = 0
frecursive = 0

start_time = time()
image = cv2.imread('.//raccoon.jpg', 0).astype(int)  # manual load pix-mapped image
assert image is not None, "No image in the path"

# Main
Y, X = image.shape  # Y: frame height, X: frame width
frame = []
for y in range(init_y, min(halt_y, Y)):  # y is index of new row pixel_, we only need one row, use init_y=0, halt_y=Y for full frame

    line = line_Ps_root( image[y,:])  # line = [Pm_, Pd_]
    if fline_PPs:
        from line_PPs import line_PPs_root
        line = line_PPs_root([line])  # line = CPp, sublayers[0] is a flat 16-tuple of P_s,
        # but it is decoded by indices as 3-layer nested tuple P_ttt: (Pm_, Pd_, each:( Lmd, Imd, Dmd, Mmd, each:( Ppm_, Ppd_)))
        if frecursive:
            from line_recursive import line_level_root
            types_ = []
            for i in range(16):  # len(line.sublayers[0]
                types_.append([i % 2, int(i % 8 / 2), int(i / 8) % 2])  # 2nd level output types: fPpd, param, fPd
            line = line_level_root(line, types_)  # line = CPp, sublayers[0] is a tuple of P_s, with implicit nesting decoded by types_

    frame.append(line)  # if fline_PPs: line is root CPp, else [Pm_, Pd_]

end_time = time() - start_time
print(end_time)
```

```
0.15802264213562012
```