Discrete Pulse Transform Library Guide v0.1

Created by Gene Stoltz
on 20 April 2013
in Notepad++

*Note - if display isnt correct open pdf.

Index

-> Overview
-> How to use DPT.h
-> DPT2PGraph
-> ReconstructGraph
-> data_structure
-> connectivity
-> DPT_Graph - The IntStruct
-> Known Issues
-> The Future
-> License

Overview

This library implements the Discrete Pulse Transform in n-dimensions. It is written in C++ but can easily be changed to a C library by exchanging all the reference-pointers and the "new" commands with relative C commands (a step required for the library). It consist of two main commands, the transform and the reconstruction.

How to use DPT.h

To use the DPT.h include DPT.cpp and compile with the -O3 flag. The library is tested with Mingw32.

Use the DPT2PGraph for decomposition
Use ReconstructGraph for reconstruction

An example opencv_DPT_example.cpp is given in the repository.

DPT2PGraph

```
int DPT2PGraph( int *&connectivity,
                int *&data_structure,
                PGraphNode *&DPT_Graph)
|
|Use this function to do the transform.
```

```
    |
    |Input: connectivity    -> pointer to data_structure
    |       data_structure  -> pointer to connectivity
    |       DPT_Graph       -> pointer to DPTGraph
    |                          (where the graph must be stored)
    |
    |Output: Int - Total amount of pulses in the DPTGraph
    |
```

ReconstructGraph

```
    int *ReconstructGraph(int n_nodes,
                          PGraphNode *&DPT_Graph,
                          int *pulseRange,
                          int size_pulseRange,
                          int offset);
    |
    |Use this function to reconstruct the DPTGraph into the data_structure
    |by using pulse cardinality(size) as criteria.
    |
    |Input: n_nodes         -> the number of data points in the data_structure
    |       DPT_Graph       -> pointer to DPTGraph
    |                          (where the graph was stored)
    |       pulseRange      -> pointer to specify reconstruction(see below)
    |       size_pulseRange -> number of elements in pulseRange
    |       offset          -> the initial value for each data point.
    |
    |output: int pointer to reconstructed data_structure
    |

    pulseRange
    The size of the variable must be of multiples of 2. The first element
    is a lower bound where the second element is an upper bound, the third
    element is then again a lower bound etc.

    Example 1.
        Reconstruct all the pulses which have size from 5 to 10 and a
        size from 20 to 30.
            size_pulseRange = 4;
            pulseRange[0] = 5;
            pulseRange[1] = 10;
            pulseRange[1] = 20;
            pulseRange[1] = 30;

    Example 2
        Reconstruct all the pulses which have size of 12.
            size_pulseRange = 2;
            pulseRange[0] = 12;
            pulseRange[1] = 12;
```

---

data_structure

The "data_structure" variable is used to pass the data points itself and must consist of an element for each d-tuple thus the structure is as follow using n-dimensions where each dimension k has range [0,k_max).

```
data_structure
[0]                          = data point at (      0,       0, ..,       0)
[1]                          = data point at (      1,       0, ..,       0)
[..]
[1_max-1]                    = data point at (1_max-1,       0, ..,       0)
[1_max-1 + 1]                = data point at (      0,       1, ..,       0)
[1_max-1 + 2]                = data point at (      2,       1, ..,       0)
[..]
[1_max*2_max*..*c_max-1] = data point at (1_max-1, 2_max-1, .., n_max-1)
```

This "data_structure" relates the position(n_tuple) of the data point to an index, this index refer to the "i" in data_structure[i] and have the formula of:

```
    i = d_1 + d_2*(1_max) +
        d_3*(2_max*1_max)+..+ d_n*((n-1)_max*(n-2)_max*..*1_max)
```

Example:

Use 3d structure of size 3x2x4 thus dimension 1 range [0,3] dimension 2 range [0,2] and dimension 3 range [0,4]

```
        data_structure[0]  = pixel( 0, 0, 0)
        data_structure[1]  = pixel( 1, 0, 0)
        data_structure[2]  = pixel( 2, 0, 0)
        data_structure[3]  = pixel( 0, 1, 0)
        data_structure[4]  = pixel( 1, 1, 0)
        data_structure[5]  = pixel( 2, 1, 0)
        data_structure[6]  = pixel( 0, 0, 1)
        data_structure[7]  = pixel( 1, 0, 1)
        data_structure[8]  = pixel( 2, 0, 1)
        data_structure[9]  = pixel( 0, 1, 1)
        data_structure[10] = pixel( 1, 1, 1)
        data_structure[11] = pixel( 2, 1, 1)
        data_structure[12] = pixel( 0, 0, 2)
        data_structure[13] = pixel( 1, 0, 2)
        data_structure[14] = pixel( 2, 0, 2)
        data_structure[15] = pixel( 0, 1, 2)
        data_structure[16] = pixel( 1, 1, 2)
        data_structure[17] = pixel( 2, 1, 2)
        data_structure[18] = pixel( 0, 0, 3)
        data_structure[19] = pixel( 1, 0, 3)
        data_structure[20] = pixel( 2, 0, 3)
        data_structure[21] = pixel( 0, 1, 3)
        data_structure[22] = pixel( 1, 1, 3)
        data_structure[23] = pixel( 2, 1, 3)
```

---

connectivity

The "connectivity" variable is used to pass information regarding the "data_structure", such as the amount of dimensions(dim) n, there relative maximums and to which each data point connects to. The connection is specified by taking the change in the current position to the connected data point.

The "connectivity" structure is as follow:

```
connectivity
[0]             = d               -> amount of dimensions in data structure
[1]             = 1_max           -> dim 1, range from 0 to 1_max
[...]
[d]             = n_max           -> dim n, range from 0 to n_max
[d+1]           = k               -> amount of connections per node
[d+1+0*d+1]     = k_1_delta_dim_1 -> change in dim 1 to get to connection 1
[d+1+0*d+2]     = k_1_delta_dim_2 -> change in dim 2 to get to connection 1
[...]
[d+1+0*d+d]     = k_1_delta_dim_d -> change in dim d to get to connection 1
[d+2+1*d+1]     = k_2_delta_dim_1 -> change in dim 1 to get to connection 2
[...]
[d+2+1*d+d]     = k_2_delta_dim_d -> change in dim d to get to connection 2
[...]
[...]
[...]
[d+2+(k-1)*d+1] = k_k_delta_dim_1 -> change in dim 1 to get to connection k
[...]
[d+2+(k-1)*d+d] = k_k_delta_dim_d -> change in dim d to get to connection k
```

Example:

Using an image with 2 dimesions and 4 connectivity

```
connectivity[0] = 2;            // dimensions
connectivity[1] = img->width;   // x - dim[1] - size
connectivity[2] = img->height;  // y - dim[2] - size
connectivity[3] = 4;            // graph connections
    //right
connectivity[4] =  1;           // connection[1] change width
connectivity[5] =  0;           // connection[1] change height
    //left
connectivity[6] = -1;           // connection[1] change width
connectivity[7] =  0;           // connection[1] change height
    //bottom
connectivity[8] =  0;           //etc
connectivity[9] =  1;
    //top
connectivity[10] =  0;
connectivity[11] = -1;
```

DPT_Graph - The IntStruct

The "DPT_Graph" is output into the PGraphNode struct which can be found in "DPT.h". This struct can be converted into an array of vectors which uses the index of the array to connected the various nodes. Each node is

represented by a vector.

The following notation is equivalent:

```
vector[i] = [v1 v2 v3 v4]   =>  vector[i][0] = v1
                                vector[i][1] = v2
                                vector[i][2] = v3
                                vector[i][3] = v4
```

Each vector at index "i" has the following structure

IntStruct

```
    [i][0]      - elements in vector
    [i][1]      - size of node (cardinality of node)( number of decendants)
    [i][2]      - height of node (node value)
    [i][3]      - index of connected pulse (or parent)
    [i][4]      - n -> number of construction pulses (or children)
    [i][5]      - index of construction pulse 1 (child 1)
    [i][..]
    [i][4 + n]  - index of construction pulse n (child n)
```

Every node who's height is 0, have no children and the index of this node relates to the n-tuple(the position) of the data point as discussed in "data_structure" section. The node who's connected pulse is -1 has no parent and is thus the root of the tree.

Example 1 :

Using a1 dimensional signal with 4 data points: x = [5 8 4 4]

Applying the DPT gives four pulses:

```
        P1 = 0 3 0 0
        P2 = 1 1 0 0
        P3 = 4 4 4 4
        P4 = 0 0 0 0     => this is the zero pulse to create only one root
                            in the tree
```

These pulses can then be represented in the IntStruct

```
    IntStruct[0] = [    5   1   0   5   0]
    IntStruct[1] = [    5   1   0   4   0]
    IntStruct[2] = [    5   1   0   6   0]
    IntStruct[3] = [    5   1   0   6   0]
    IntStruct[4] = [    6   1   3   5   1   1]
    IntStruct[5] = [    7   2   1   6   2   4   0]
    IntStruct[6] = [    8   4   4   7   3   2   3   5]
    IntStruct[7] = [    6   4   0  -1   1   6]
```

Example 2 : (Demostrate the requirement for the last zero pulse)

Using a1 dimensional signal with 4 data points: x = [5 0 0 9]

Applying the DPT gives three pulses:

```
        P1 = 5 0 0 0
        P2 = 0 0 0 9
        P3 = 0 0 0 0
```

These pulses can then be represented in the IntStruct

```
    IntStruct[0] = [    5   1   0   5   0]
    IntStruct[1] = [    5   1   0   4   0]
    IntStruct[2] = [    5   1   0   6   0]
```

```
        IntStruct[3] = [    5   1   0   6   0]
        IntStruct[4] = [    6   1   5   6   1   0]
        IntStruct[5] = [    6   1   9   6   1   3]
        IntStruct[6] = [    7   4   0  -1   3   4   5]
```

Known Issues

There are some fair memory leakage which still need to be attended to.

The Future

Currently the library is doing the bare minimum. Some more advanced functions.

Port all C++ code to pure C.

There is still alot of optimization that can be done to the code
    Like not stepping through the feature table for each Ln and Un operator but storing references to the second operator to application.

Easier access to apply video.
Easier access for arbitrary node configuration.
    Unfortunatley there will always be the requirement for a zero node to adhere to the mathematics

---

License

Discrete Pulse Transform Library Guide
Discrete Pulse Transform Library
DPT Library

Copyright (C) 2013, Gene Stoltz
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of skimage nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

==========

END

==========