

Software Specification of PSET —  
A Page Segmentation Evaluation Toolkit  
Version 1.01

Song Mao and Tapas Kanungo

Language and Media Processing Lab  
Center for Automation Research  
University of Maryland  
College Park, MD 20742  
kanungo@cfar.umd.edu

September 29, 2000



# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
<b>2</b>	<b>The Page Segmentation Problem</b>	<b>13</b>
<b>3</b>	<b>Performance Evaluation Methodology</b>	<b>15</b>
<b>4</b>	<b>Algorithm Description</b>	<b>17</b>
4.1	Page Segmentation Algorithm . . . . .	17
4.1.1	The X-Y Cut Page Segmentation Algorithm . . . . .	17
4.1.2	The Docstrum Page Segmentation Algorithm . . . . .	18
4.1.3	The Voronoi-Diagram-Based Page Segmentation Algorithm . . . . .	20
4.2	The Textline-based Error Metric and Benchmark (Error Counter) Algorithm . . . . .	20
4.2.1	Page Segmentation Definition . . . . .	21
4.2.2	Error Measurements and Metric Definitions . . . . .	21
4.2.3	Benchmark (Error Counter) Algorithm . . . . .	24
4.3	The Training Algorithm . . . . .	24
<b>5</b>	<b>Architecture, File Formats, and Evaluation Methodology</b>	<b>29</b>
5.1	Architecture and File Formats . . . . .	29
5.2	Implementing the Evaluation Methodology . . . . .	31
5.3	Algorithm Calling Mode in the Segmentation Algorithm Module	38
<b>6</b>	<b>Tutorial</b>	<b>43</b>
6.1	Basic Integration Elements . . . . .	43
6.2	Development Environments . . . . .	44
6.3	Examples . . . . .	44

6.3.1	Use a Page Segmentation Algorithm . . . . .	44
6.3.2	Use the Textline-Based Benchmarking Algorithm . . . . .	44
<b>7</b>	<b>Command Line Specification</b>	<b>49</b>
7.1	The X-Y Cut Page Segmentation Algorithm Command . . . . .	49
7.2	The Docstrum Page Segmentation Algorithm Command . . . . .	50
7.3	The Voronoi-Based Segmentation Algorithm Command . . . . .	50
7.4	The Algorithm Training Command . . . . .	51
7.5	The Algorithm Testing Command . . . . .	52
<b>8</b>	<b>Data Structures</b>	<b>53</b>
<b>9</b>	<b>Function Specifications</b>	<b>75</b>

# List of Figures

4.1	The X-Y Cut segmentation algorithm. . . . .	18
4.2	The Docstrum segmentation algorithm. . . . .	19
4.3	The Voronoi-based segmentation algorithm. . . . .	20
4.4	Samples of a benchmark algorithm parameter file (bpr) (a) and a weight file (wgt) (b). . . . .	22
4.5	The Textline-Based benchmarking algorithm. . . . .	24
4.6	Experiment Example . . . . .	26
4.7	The Simplex optimization algorithm. . . . .	27
4.8	Simplex operations . . . . .	28
5.1	Overall PSET architecture. The left half of the architecture represents the training phase; the right half represents the test- ing phase. Note that in the testing phase, the optimal page segmentation parameter found in the training phase is used. The training and testing phases use the same performance metric related input files (benchmark algorithm parameter file (bpr) and weight file (wgt)) and the same segmentation algo- rithm shell file (sh). . . . .	30
5.2	Input file formats. The training protocol file format is shown in (a), the test protocol file format is shown in (b), and the al- gorithm parameter file format is shown in (c). The description of the attributes in (a) and (b) is given in (d). . . . .	32
5.3	The training report file format. The format is shown in (a) and the description of each column entry in (a) is shown in (b). . . . .	33
5.4	The test report file format. The format is shown in (a) and the description of each column entry in (a) is shown in (b). . . . .	34

- 5.5 Parameter reading stage of the training phase (a) and the testing phase (b). At this level, various parameter files are read into their corresponding data structures which are fed into the Train and Test modules. . . . . 35
- 5.6 The Train module. In this module, the objective function is optimized over a given training dataset. Two files are generated by this module, a train report file (trr) and an optimal segmentation algorithm parameter file (spr). . . . . 36
- 5.7 Software architectures of the objective function module and the test module. Module A represents the page segmentation algorithm module, module B represents the page segmentation error counter and scoring module, and module C represents the objective function module. The test module in (b) has sub-modules similar to those in (a). It also has a module for computing a final testing performance score (average textline accuracy). . . . . 37
- 5.8 Sample protocol files. From both the train protocol file (a) and the test protocol file (b), we can see that the list files of the training dataset and test dataset are *train.lst* and *test.lst* respectively, the optimization algorithm used is the *Simplex* algorithm, the benchmarking algorithm used is the *Textline-based* algorithm, the page segmentation algorithm trained is the *Docstrum* algorithm, and the page segmentation algorithm tested is the *X-Y cut* algorithm. We can also find the locations of the groundtruth files, image files and training and test result files. Moreover, the suffixes for various files are given for file name manipulation in the PSET API. . . . . 38
- 5.9 Samples of an optimization algorithm parameter file (opr) and a segmentation algorithm parameter file (spr). A sample file for the Simplex optimization algorithm is shown in (a) and a sample file for the X-Y cut segmentation algorithm is shown in (b). Their detailed parameter descriptions can be found in [10]. . . . . 39

5.10	Samples of a training report file format (a) and a test report file format (b). The comment lines provide experimental environment information about the training and test experiments. They are automatically generated by calling various GNU C functions. They are crucial for replicating experimental results. In the data area, both intermediate information and final results are recorded. This information can be used to analyze the convergence properties of the training process and to study the statistical significance of the test experiment results. A detailed description of each column entry can be found in Figure 5.3(b) and Figure 5.4(b). . . . .	40
5.11	A sample shell file. . . . .	40
5.12	Page segmentation algorithm calling modes: function call and shell call. The left half represents the function calling mode and the right half represents the shell calling mode. The shell calling mode can be used only when the algorithm executable is available; otherwise the function calling mode can be used. Note that the executable is called by the function <i>sh_c</i> . . . . .	41
6.1	<b>sample1.c</b> : A sample code to use the X-Y cut page segmentation algorithm by calling the algorithm function. . . . .	45
6.2	The UNIX command line usage of the X-Y cut page segmentation algorithm. A001BIN.TIF is the input image and A001.dafs is the output segmentation result file in DAFS format. . . . .	45
6.3	<b>sample2.c</b> : A sample code to use the textline-based benchmarking algorithm by calling the algorithm function. . . . .	46
6.4	The makefile for the code in Figure 6.3 and Figure 6.1. . . . .	47



# List of Tables

- 4.1 Summary of error measurements and the corresponding symbols defined in this section. . . . . 23
- 5.1 Summary of the file formats in the PSET package. . . . . 31



# Chapter 1

## Introduction

It is important to quantitatively monitor progress in any scientific field. The information retrieval community and the speech recognition community, for example, have yearly competitions in which researchers evaluate their latest algorithms on clearly defined tasks, datasets, and metrics. To make such evaluations possible, researchers have access to standardized datasets, metrics, and freely available software for scoring the results produced by algorithms [18, 1].

In the Document Image Analysis area, regular evaluations of OCR accuracy have been conducted by UNLV [2]. Page segmentation algorithms, which are crucial components of OCR systems, were at one time evaluated by UNLV based on the final OCR results, but not on the geometric results of the segmentation. Recently [12], we empirically compared various commercial and research page segmentation algorithms, using the University of Washington dataset. We used a well-defined (geometric) line-based metric and a sound statistical methodology to score the segmentation results. Furthermore, unlike the UNLV evaluations, we trained the segmentation algorithms prior to evaluating them.

In this document we describe in detail the software specification of a package called PSET, which we used in [12] to evaluate page segmentation algorithms. This package was developed by us at the University of Maryland and will be made available to researchers at no cost. Publication of the package will allow researchers to implement our five-step evaluation methodology and evaluate their own algorithms.

Software architecture can be described using methods such as Petri Nets and Data Flow Diagrams [7]. We describe the architecture of PSET, the

I/O file formats, etc. using Object-Process Diagrams (OPDs) [5], which are similar in spirit to Petri Nets.

The package, called the Page Segmentation Evaluation Toolkit (PSET), is modular, written using the C language, and runs on the SUN/UNIX platform. The software has been structured so that it can be used at the UNIX command line level or compiled into other software packages by calling API functions. The description in this document will aid users in using, updating, and modifying the PSET package. It will also help users to add new algorithm modules to the package and to interface it with other software tools and packages. The PSET package includes three research page segmentation algorithms; <sup>1</sup> a textline-based benchmarking algorithm; and a Simplex-based optimization algorithm for estimating algorithm parameters from training datasets.

This document is organized as follows. In Chapter 2, we discuss the page segmentation problem. In Chapter 3, we present our five-step page segmentation performance evaluation methodology. In Chapter 4, we give a detailed description of all algorithms in the PSET package. In Chapter 5, we describe the architecture and file formats of our PSET package in detail and show how to implement each step of our five-step performance evaluation methodology. In Chapter 6, we show several examples of how to call the PSET API functions. In Chapter 7, we describe command lines of the PSET package. In Chapter 8, we describe each data structure in the PSET package. Finally in Chapter 9, we describe each function in the PSET package.

---

<sup>1</sup>We implemented the X-Y cut algorithm [13] and the Docstrum algorithm [15]. Kise [9] provided us the C implementation of his Voronoi-based algorithm.

## Chapter 2

# The Page Segmentation Problem

There are two types of page segmentation, physical and logical. Physical page segmentation is a process of dividing a document page into homogeneous zones. Each of these zones can contain one type of object. These objects can be of type text, table, figure, halftone image, etc. Logical page segmentation is a process of assigning logical relations to physical zones. For example, reading order labels order the physical zones in the order in which they should be read. Similarly, assigning section and sub-section labels to physical zones creates a hierarchical document structure. In this document, we focus on physical page segmentation and refer to it as simply page segmentation hereafter.

Page segmentation is a crucial preprocessing step for an OCR system. In many cases, OCR engine recognition accuracy depends heavily on page segmentation accuracy. For instance, if a page segmentation algorithm merges two text zones horizontally, the OCR engine will recognize text across text zones and hence generate unreadable text. Page segmentation algorithms can be categorized into three types: top-down, bottom-up, and hybrid approaches. Top-down approaches iteratively divide a document page into smaller zones according to some criterion. The X-Y cut algorithm developed by Nagy *et al.* [13] is a typical top-down algorithm. Bottom-up approaches start from document image pixels, and iteratively group them into bigger regions. The Docstrum algorithm of O’Gorman [15] and the Voronoi-based algorithm of Kise *et al.* [9] are representative bottom-up approaches. Hybrid approaches are usually a mixture of top-down and bottom-up approaches.

The algorithm of Pavlidis and Zhou [16] is an example of the hybrid approach that employs a split-and-merge strategy.

# Chapter 3

## Performance Evaluation Methodology

In order to objectively evaluate page segmentation algorithms, a performance evaluation methodology should take into consideration the performance metric, the dataset, the training and testing methods, and the methodology of analyzing experimental results. In this chapter, we introduce a five-step methodology that we proposed earlier [12, 10, 11]. The PSET package is an implementation of this methodology.

Let  $\mathcal{D}$  be a given dataset containing (document image, groundtruth) pairs  $(I, G)$ , and let  $\mathcal{T}$  and  $\mathcal{S}$  be a training dataset and a test dataset respectively. The five-step methodology is described as follows:

1. Randomly divide the dataset  $\mathcal{D}$  into two mutually exclusive datasets: a training dataset  $\mathcal{T}$  and a test dataset  $\mathcal{S}$ . Thus,  $\mathcal{D} = \mathcal{T} \cup \mathcal{S}$  and  $\mathcal{T} \cap \mathcal{S} = \phi$ , where  $\phi$  is the empty set.
2. Define a computable performance metric  $\rho(I, G, R)$ . Here  $I$  is a document image,  $G$  is the groundtruth of  $I$ , and  $R$  is the OCR segmentation result on  $I$ . In our case,  $\rho(I, G, R)$  is defined as textline accuracy, as described in the Appendix.
3. Given a segmentation algorithm  $A$  with a parameter vector  $\mathbf{p}^A$ , automatically search for the optimal parameter value  $\hat{\mathbf{p}}^A$  for which an objective function  $f(\mathbf{p}^A; \mathcal{T}, \rho, A)$  assumes the optimal value on the training dataset  $\mathcal{T}$ . In our case, this objective function is defined as the average

textline error rate on a given training dataset:

$$f(\mathbf{p}^A; \mathcal{T}, A, \rho) = \frac{1}{\#\mathcal{T}} \left[ \sum_{(I,G) \in \mathcal{T}} 1 - \rho(G, Seg_A(I, \mathbf{p}^A)) \right].$$

4. Evaluate the segmentation algorithm  $A$  with the optimal parameter  $\hat{\mathbf{p}}^A$  on the test dataset  $\mathcal{S}$  by

$$\Phi \left( \{ \rho(G, Seg_A(I, \hat{\mathbf{p}}^A)) \mid (I, G) \in \mathcal{S} \} \right)$$

where  $\Phi$  is a function of the performance metric  $\rho$  on each (document image, groundtruth) pair  $(I, G)$  in the test dataset  $\mathcal{S}$ , and  $Seg_A(\cdot, \cdot)$  is the segmentation function corresponding to  $A$ . The function  $\Phi$  is defined by the user. In our case,

$$\Phi \left( \{ \rho(G, Seg_A(I, \hat{\mathbf{p}}^A)) \mid (I, G) \in \mathcal{S} \} \right) = 1 - f(\hat{\mathbf{p}}^A; \mathcal{S}, \rho, A),$$

which is the average of the textline accuracy  $\rho(G, Seg_A(I, \hat{\mathbf{p}}^A))$  achieved on each (document image, groundtruth) pair  $(I, G)$  in the test dataset  $\mathcal{S}$ .

5. Perform a statistical analysis to evaluate the statistical significance of the evaluation results, and analyze the errors to identify/hypothesize why the algorithms perform at their respective levels.

# Chapter 4

## Algorithm Description

In this chapter, we first describe three research page segmentation algorithms, then we describe the textline-based benchmarking algorithm, and finally we describe the Simplex optimization algorithm.

### 4.1 Page Segmentation Algorithm

Page segmentation algorithms can be categorized into three classes: top-down approaches, bottom-up approaches and hybrid approaches. We implemented the X-Y cut algorithm (a top-down algorithm) and the Docstrum algorithm (a bottom-up algorithm). Kise provided us a C implementation of his Voronoi-based algorithm (a bottom-up algorithm). Two commercial products, Caere's segmentation algorithm [4] and ScanSoft's segmentation algorithm [17], were selected for evaluation. They are representative state-of-art commercial products. Both are black-box algorithms with no free parameters. In the following subsections, we describe the three research algorithms.

#### 4.1.1 The X-Y Cut Page Segmentation Algorithm

The X-Y cut segmentation algorithm [13] is a tree-based, top-down algorithm. The root node of the tree represents the entire document page image  $I$ , an interior node represents a rectangle on the page, and all the leaf nodes together represent the final segmentation. While this algorithm is easy to implement, it can only work on deskewed document pages with Manhattan

layout and rectangular zones. The algorithm works as shown in Figure 4.1.

1. Create the horizontal and vertical prefix sum tables  $H_X$  and  $H_Y$  as follows:
 
$$H_X[i][j] = \#\{p \in D(I) | X(p) = j, Y(p) \leq i, I(p) = 1\},$$

$$H_Y[i][j] = \#\{p \in D(I) | X(p) \leq j, Y(p) = i, I(p) = 1\},$$
 where  $D(I) \subseteq \mathcal{Z}^2$  is the domain of the image  $I$  and  $I(p)$  is the binary value of the image at pixel  $p$ , and  $X(p)$  and  $Y(p)$  are the X and Y coordinates of the pixel  $p$  respectively.
2. Initialize a tree with the entire document image as the root node. For each node do the following:
  - (a) Compute X and Y black pixel projection profile histograms of the current node as follows:
 
$$HIS_X[i] \leftarrow H_X[Y_2(Z)][i] - H_X[Y_1(Z)][i],$$

$$HIS_Y[j] \leftarrow H_Y[j][X_2(Z)] - H_Y[j][X_1(Z)],$$
 where  $Z$  is the zone corresponding to the current node, and  $(X_1(Z), Y_1(Z))$  and  $(X_2(Z), Y_2(Z))$  are upper-left and lower-right points of the zone.
  - (b) Shrink each current zone bounding box until it “tightly” encloses the the zone body. Noise removal thresholds  $T_X^n$  and  $T_Y^n$  are then used to classify and remove background noise pixels. Since noise pixels in the background are assumed to be distributed uniformly, the noise removal thresholds  $T_X^n$  and  $T_Y^n$  for a particular node are scaled linearly based on the current zone’s width and height.
  - (c) Repeat step 2a.
  - (d) Obtain the widest zero valleys  $V_X$  and  $V_Y$  in the X and Y projection profile histograms  $HIS_X$  and  $HIS_Y$ .
  - (e) If  $V_X > T_X$  or  $V_Y > T_Y$ , where  $T_X$  and  $T_Y$  are two width thresholds, split at the mid-point of the wider of  $V_X$  and  $V_Y$  and generate two child nodes. Otherwise, make the current node a leaf node.

Figure 4.1: The X-Y Cut segmentation algorithm.

### 4.1.2 The Docstrum Page Segmentation Algorithm

Docstrum [15] is a bottom-up page segmentation algorithm that can work on document page images with non-Manhattan layout and arbitrary skew angles. This algorithm tends to fragment non-text regions (figures, tables and halftone images) and text zones with irregular font sizes and spacings.

Moreover, it does not perform well when document images contain sparse characters.

The basic steps of the Docstrum segmentation algorithm are shown in Figure 4.2. In our implementation, we did not estimate orientation since all

1. Obtain connected components ( $C_i$ s) using a space-efficient two-pass algorithm [8].
2. Remove small and large noise or non-text connected components using low and high thresholds  $l$  and  $h$ .
3. Separate the  $C_i$ s into two groups, one with dominant characters and the other with characters in titles and section headings. A parameter  $f_d$  controls the clustering.
4. Find the  $K$  nearest neighbors,  $\text{NN}_K(i)$ , of each  $C_i$ .
5. Compute the distance and angle of each  $C_i$  and its  $K$  nearest neighbors:  $(\rho_j^i, \theta_j^i)$ , such that  $j \in \text{NN}_K(i)$ .
6. Compute a within-line nearest-neighbor distance histogram from the following set  $W_\rho : W_\rho = \{\rho_j^i | j \in \text{NN}_K(i), \text{ and } -\theta_h \leq \theta_j^i \leq \theta_h\}$ , where  $\theta_h$  is the horizontal angle tolerance threshold. Estimate the within-line inter-character spacing  $cs$  as the location of the peak in the histogram.
7. Compute a between-line nearest-neighbor distance histogram from the set  $B_\rho : B_\rho = \{\rho_j^i | j \in \text{NN}_K(i), \text{ and } 90^\circ - \theta_v \leq \theta_j^i \leq 90^\circ + \theta_v\}$ , where  $\theta_v$  is the vertical angle tolerance threshold. Estimate the inter-line spacing  $ls$  as the location of the peak in the histogram.
8. Perform transitive closure on within-line nearest neighbor pairings to obtain textlines  $L_i$ s using within-line nearest neighbor distance threshold  $T_{cs} = f_t \cdot cs$ .
9. Perform transitive closure on the  $L_i$ s to obtain structural blocks or zones  $Z_i$ s using parallel distance threshold  $T_{pa} = f_{pa} \cdot cs$  and perpendicular distance threshold  $T_{pe} = f_{pe} \cdot ls$ . The parallel and perpendicular distances are computed as “end-end” distance, not “centroid-centroid” distance.

Figure 4.2: The Docstrum segmentation algorithm.

pages in the dataset were deskewed. Furthermore, we used a resolution of 1 pixel/bin for constructing the within-line and between-line histograms, and did not perform any smoothing of these histograms.

### 4.1.3 The Voronoi-Diagram-Based Page Segmentation Algorithm

Kise’s segmentation algorithm [9] is also a bottom-up algorithm based on the Voronoi diagram. This method can work on document page images that have non-Manhattan layout, arbitrary skew angles, or non-linear textlines. A set of connected line segments are used to bound text zones. Since we evaluate all algorithms on document page images with Manhattan layouts, this algorithm has been modified to generate rectangular zones. This algorithm has limitations similar to those of the Docstrum algorithm. The algorithm steps are shown in Figure 4.3.

1. Label connected components. Sample points on their borders. The parameter  $sr$  controls the number of sample points used.
2. Remove noise connected components using maximum noise zone size threshold  $nm$ , maximum width threshold  $C_w$ , maximum height threshold  $C_h$ , and maximum aspect ratio threshold  $C_r$  for all connected components.
3. The Voronoi diagram for each connected component is generated using the sample points on its border.
4. Delete superfluous Voronoi edges to generate text zone boundaries according to a spacing and area-ratio criteria [9].
5. Remove noise zones using minimum area threshold  $A_z$  for all zones, and using minimum area threshold  $A_l$ , and maximum aspect ratio threshold  $B_r$  for the zones that are vertical and elongated.

Figure 4.3: The Voronoi-based segmentation algorithm.

## 4.2 The Textline-based Error Metric and Benchmark (Error Counter) Algorithm

In the following sections, we define page segmentation, a set of textline-based error measurements and a performance metric that we used in our previous evaluation of page segmentation algorithms [12, 11]. These definitions are based on set theory and mathematical morphology [8]. We then define a general metric that users can customize for their individual tasks.

### 4.2.1 Page Segmentation Definition

Let  $I$  be a document image, and let  $G$  be the groundtruth of  $I$ . Let  $Z(G) = \{Z_q^G, q = 1, 2, \dots, \#Z(G)\}$  be a set of groundtruth zones of document image  $I$  where  $\#$  denotes the cardinality of a set. Let  $L(Z_q^G) = \{l_{qj}^G, j = 1, 2, \dots, \#L(Z_q^G)\}$  be the set of groundtruth textlines in groundtruth zone  $Z_q^G$ . Let the set of all groundtruth textlines in document image  $I$  be  $\mathcal{L} = \bigcup_{q=1}^{\#Z(G)} L(Z_q^G)$ . Let  $A$  be a given segmentation algorithm, and  $Seg_A(\cdot, \cdot)$  be the segmentation function corresponding to algorithm  $A$ . Let  $R$  be the segmentation result of algorithm  $A$  such that  $R = Seg_A(I, \mathbf{p}^A)$  where  $Z(R) = \{Z_k^R | k = 1, 2, \dots, \#Z(R)\}$ .

Let  $D(\cdot) \subseteq \mathcal{Z}^2$  be the domain of its argument. The groundtruth zones and textlines have the following properties: 1)  $D(Z_q^G) \cap D(Z_{q'}^G) = \phi$  for  $Z_q^G, Z_{q'}^G \in Z(G)$  and  $q \neq q'$ , and 2)  $D(l_i^G) \cap D(l_{i'}^G) = \phi$  for  $l_i^G, l_{i'}^G \in \mathcal{L}$  and  $i \neq i'$ .

### 4.2.2 Error Measurements and Metric Definitions

In this section, we define four error measurements and a metric. Let  $T_X, T_Y \in \mathcal{Z}^+ \cup \{0\}$  be two length thresholds (in pixels) that determine if the overlap is significant or not. Each of these thresholds is defined in terms of an absolute threshold and a relative threshold. The absolute threshold is in pixels and the relative threshold is a percentage.  $T_X$  and  $T_Y$  are defined as follows:

$$T_X = \min\{HPIX, (100 - HTOL) \cdot h/100\} \quad (4.1)$$

$$T_Y = \min\{VPIX, (100 - VTOL) \cdot v/100\} \quad (4.2)$$

where  $HPIX$  and  $VPIX$  are the the two thresholds in pixels,  $HTOL$  and  $VTOL$  are the two thresholds in percentages, and  $h, v$  are the minimum width and height (in pixels) of two regions that are tested for significant overlap. Users must specify the  $HTOL, VTOL, HPIX$  and  $VPIX$  parameter values in the benchmark algorithm parameter file (bpr). Figure 4.4(b) shows a sample benchmark algorithm parameter file.

Let  $E(T_X, T_Y) = \{e \in \mathcal{Z}^2 | -T_X \leq X(e) \leq T_X, -T_Y \leq Y(e) \leq T_Y\}$  be a region of a rectangle centered at  $(0, 0)$  with a width of  $2T_X + 1$  pixels, and a height of  $2T_Y + 1$  pixels where  $X(\cdot)$  and  $Y(\cdot)$  denote the  $X$  and  $Y$  coordinates of the argument, respectively. We now define two morphological operations: dilation and erosion [8]. Let  $A, B \subseteq \mathcal{Z}^2$ . Morphological *dilation* of  $A$  by  $B$  is denoted by  $A \oplus B$  and is defined as  $A \oplus B =$

# The Textline-Based Benchmark	
# Algorithm Parameters	
HTOL	= 90
VTOL	= 80
HPIX	= 11
VPIX	= 8

# weight file	
wSpl	= 0
wMrg	= 0
wMis	= 0
wFA	= 0
wSplLine	= 1
wMrgLine	= 1
wMisLine	= 1
wFAZone	= 0

(a)

(b)

Figure 4.4: Samples of a benchmark algorithm parameter file (bpr) (a) and a weight file (wgt) (b).

$\{c \in Z^2 | c = a + b \text{ for some } a \in A, b \in B\}$ . Morphological *erosion* of  $A$  by  $B$  is denoted by  $A \ominus B$  and is defined as  $A \ominus B = \{c \in Z^2 | c + b \in A \text{ for every } b \in B\}$ .

We now define three types of textline based error measurements:

- 1) Groundtruth textlines that are missed:

$$C_L = \{l^G \in \mathcal{L} | D(l^G) \ominus E(T_X, T_Y) \subseteq (\cup_{Z^R \in Z(R)} D(Z^R))^c\}, \quad (4.3)$$

- 2) Groundtruth textlines whose bounding boxes are split:

$$S_L = \left\{ l^G \in \mathcal{L} | \begin{aligned} & (D(l^G) \ominus E(T_X, T_Y)) \cap D(Z^R) \neq \phi, \\ & (D(l^G) \ominus E(T_X, T_Y)) \cap (D(Z^R))^c \neq \phi, \\ & \text{for some } Z^R \in Z(R) \end{aligned} \right\}, \quad (4.4)$$

- 3) Groundtruth textlines that are horizontally merged:

$$M_L = \left\{ l_{qj}^G \in \mathcal{L} | \exists l_{q'j'}^G \in \mathcal{L}, Z^R \in Z(R), q \neq q', \right. \\ \left. \begin{aligned} & Z_q^G, Z_{q'}^G \in Z(G) \text{ such that} \\ & (D(l_{qj}^G) \ominus E(T_X, T_Y)) \cap D(Z^R) \neq \phi, \\ & (D(l_{q'j'}^G) \ominus E(T_X, T_Y)) \cap D(Z^R) \neq \phi, \\ & ((D(l_{qj}^G) \ominus E(0, T_Y)) \oplus E(\infty, 0)) \cap D(Z_q^G) \neq \phi, \\ & ((D(l_{q'j'}^G) \ominus E(0, T_Y)) \oplus E(\infty, 0)) \cap D(Z_{q'}^G) \neq \phi \end{aligned} \right\}. \quad (4.5)$$

- 4) Noise zones that are falsely detected (false alarm):

$$F_L = \{Z^R \in Z(R) | D(Z^R) \subseteq (\cup_{l^G \in \mathcal{L}} (D(l^G) \ominus E(T_x, T_Y)))^c\} \quad (4.6)$$

#### 4.2. THE TEXTLINE-BASED ERROR METRIC AND BENCHMARK (ERROR COUNTER) ALGO.

Let the number of groundtruth error textlines be  $\#\{C_L \cup S_L \cup M_L\}$  (mis-detected, split, or horizontally merged), and let the total number of groundtruth textlines be  $\#\mathcal{L}$ . We define the performance metric  $\rho(I, G, R)$  as textline accuracy:

$$\rho(I, G, R) = \frac{\#\mathcal{L} - \#\{C_L \cup S_L \cup M_L\}}{\#\mathcal{L}}.$$

In the PSET package, we also define some other error measurements. Table 4.1 shows the error measurements, the metric defined in the PSET package, and the corresponding symbols used in the above discussion.

Table 4.1: Summary of error measurements and the corresponding symbols defined in this section.

Error Measure Defined in the PSET package	Equivalent Term in this Section	Description
$nSpl$	none	The number of split errors.
$nMrg$	none	The number of horizontal merge errors.
$nFA$	$\#F_L$	The number of false alarm errors.
$nSplL$	$\#S_L$	The number of split textlines.
$nMrgL$	$\#M_L$	The number of horizontally merged textlines.
$nMisL$	$\#C_L$	The number of mis-detected textlines.
$nErrL$	$\#\{C_L \cup S_L \cup M_L\}$	The number of error textlines (textlines that are either split, horizontally merged or mis-detected).
$nGtl$	$\#\mathcal{L}$	The number of groundtruth textlines.

In general, the performance metric can be any function of the error measures shown in Table 4.1. In the PSET package, a performance metric can be defined as a weighted sum of these error measures in function *BenchScoring*. Let  $wSpl$  be the weight of the error measurement  $nSpl$ . The weights of other error measurements are defined similarly. A general performance metric is defined as follows:

$$\begin{aligned}
 N &= wSpl * nSpl + wMrg * nMrg + wFA * nFA + wSplL * nSplL \\
 &\quad + wMrgL * nMrgL + wMisL * nMisL, \\
 D &= wSpl + wMrg + wFA + wSplL + wMrgL + wMisL, \\
 \rho^*(I, G, R) &= \frac{N}{D}.
 \end{aligned} \tag{4.7}$$

### 4.2.3 Benchmark (Error Counter) Algorithm

Based on this performance metric, the textline-based benchmarking algorithm works as shown in Figure 4.5. Figure 4.6 gives a set of possible errors

- 1) For each groundtruth textline  $l^G \in \mathcal{L}$ , if there exists a segmentation zone  $Z^R \in Z(R)$  such that Equation 4.4 is satisfied, save the index of  $Z^R$  and SPLIT error type and set split flag to ON in the textline data structure of  $l^G$ , also increment number of splits error counter by 1.
- 2) For each possible pair of groundtruth textline  $l^{qj}$  and  $l^{q'j'} \in \mathcal{L}$ ,  $q \neq q'$ , if there exists a segmentation zone  $Z^R \in Z(R)$  such that Equation 4.5 is satisfied, save the index of  $Z^R$  and MERGE error type and set merge flag to ON in the textline data structures of  $l^{qj}$  and  $l^{q'j'}$ , also increment number of merges error counter by 1.
- 3) For each groundtruth textline  $l^G \in \mathcal{L}$ , if in its data structure, no segmentation zone  $Z^R \in Z(R)$  index has been saved, i.e. Equation 4.3 is satisfied, set miss flag to ON in the textline data structure of  $l^G$ , also increment number of mis-detection error counter by 1.
- 4) For each groundtruth textline  $l^G \in \mathcal{L}$ , if at least one error flag is ON (merge, split and miss), increment error textline counter by 1. If split flag is ON, increment split textline counter by 1, if merge flag is ON, increment merge textline counter by 1, if miss flag is ON, increment miss textline counter by 1.
- 5) Save all error counter values into an error measure data structure.

Figure 4.5: The Textline-Based benchmarking algorithm.

as well as an experimental example.

## 4.3 The Training Algorithm

Direct search methods are typically used to solve the optimization problem described in Section 4.1. We choose the simplex search method proposed by Nelder and Mead [14] to minimize our objective function.

We give the notation used to describe the simplex method: Let  $\mathbf{q}_0$  and  $\lambda_i, i = 1, \dots, n$  be a starting point and a set of scales, let  $\mathbf{e}_i, i = 1, \dots, n$  be  $n$  orthogonal unit vectors in  $n$ -dimensional parameter space, let  $\mathbf{p}_0, \dots, \mathbf{p}_n$  be  $(n + 1)$  ordered points in  $n$ -dimensional parameter space such that their corresponding function values satisfy  $f_0 \leq f_1 \leq \dots \leq f_n$ , let  $\bar{\mathbf{p}} = \sum_{i=0}^{n-1} \mathbf{p}_i / n$  be the centroid of the  $n$  best (smallest) points, let  $[\mathbf{p}_i \mathbf{p}_j]$  be the  $n$ -dimensional Euclidean distance from  $\mathbf{p}_i$  to  $\mathbf{p}_j$ , let  $\alpha, \beta, \gamma$  and  $\sigma$  be the *reflection, contraction, expansion and shrinkage coefficient*, respectively, and let  $T$  be the

threshold for the stopping criterion. We use the standard choice for the coefficients:  $\alpha = 1$ ,  $\beta = 0.5$ ,  $\gamma = 2$ ,  $\sigma = 0.5$ . We set  $T$  to  $10^{-6}$ . Figure 4.8 shows the various simplex operations.

For a segmentation algorithm with  $n$  parameters, the Nelder-Mead algorithm works as shown in Figure 4.7.

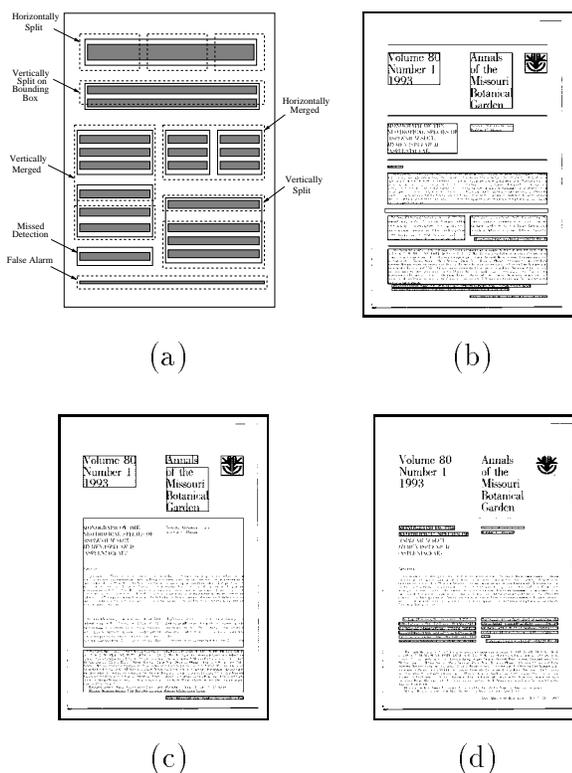


Figure 4.6: (a) This figure shows a set of possible textline errors. Solid-line rectangles denote groundtruth zones, dashed-line rectangles denote OCR segmentation zones, dark bars within groundtruth zones denote groundtruth textlines, and dark bars outside solid lines are noise blocks. (b) A document page image from the University of Washington III dataset with the groundtruth zones overlaid. (c) OCR segmentation result on the image in (b). (d) Segmentation error textlines. Notice that there are two horizontally merged zones just below the caption and two horizontally merged zones in the middle of the text body. In OCR output, horizontally split zones cause reading order errors whereas vertically split zones do not cause such errors.

- 1 Given  $\mathbf{q}_0$  and the  $\lambda_i$ , form the initial simplex as  
 $\mathbf{q}_i = \mathbf{q}_0 + \lambda_i \mathbf{e}_i, i = 1, \dots, n.$
- 2 Relabel the  $n + 1$  vertices as  $\mathbf{p}_0, \dots, \mathbf{p}_n$  with  
 $f(\mathbf{p}_0) \leq f(\mathbf{p}_1) \cdots \leq f(\mathbf{p}_n),$
- 3 Get a reflection point  $\mathbf{p}_r$  of  $\mathbf{p}_n$  by  $\mathbf{p}_r = (1 + \alpha)\bar{\mathbf{p}} - \alpha\mathbf{p}_n$   
 where  $\alpha = [\mathbf{p}_r\bar{\mathbf{p}}]/[\mathbf{p}_n\bar{\mathbf{p}}].$
- 4.1 If  $f(\mathbf{p}_r) \leq f(\mathbf{p}_0)$ , replace  $\mathbf{p}_n$  by  $\mathbf{p}_r$  and  $f(\mathbf{p}_n)$  by  
 $f(\mathbf{p}_r)$ , get an expansion point  $\mathbf{p}_e$  of  $\mathbf{p}_n$  by  
 $\mathbf{p}_e = (1 - \gamma)\bar{\mathbf{p}} + \gamma\mathbf{p}_n$  where  $\gamma = [\mathbf{p}_e\bar{\mathbf{p}}]/[\mathbf{p}_n\bar{\mathbf{p}}] > 1.$   
 If  $f(\mathbf{p}_e) < f(\mathbf{p}_n)$ , replace  $\mathbf{p}_n$  by  $\mathbf{p}_e$  and  $f(\mathbf{p}_n)$  by  $f(\mathbf{p}_e).$   
 Go to step 5.
- 4.2 Else if  $f(\mathbf{p}_r) \geq f(\mathbf{p}_{n-1})$ , if  $f(\mathbf{p}_r) < f(\mathbf{p}_n)$  replace  $\mathbf{p}_n$  by  $\mathbf{p}_r$  and  $f(\mathbf{p}_n)$  by  $f(\mathbf{p}_r)$ ,  
 get a contraction point  $\mathbf{p}_c$   
 of  $\mathbf{p}_n$  by  $\mathbf{p}_c = (1 - \beta)\bar{\mathbf{p}} + \beta\mathbf{p}_n, \beta = [\mathbf{p}_c\bar{\mathbf{p}}]/[\mathbf{p}_n\bar{\mathbf{p}}] < 1.$   
 If  $f(\mathbf{p}_c) \geq f(\mathbf{p}_n)$ , shrink the simplex around the best  
 vertex  $\mathbf{p}_0$  by  $\mathbf{p}_i = (\mathbf{p}_i + \mathbf{p}_0)\sigma, i \neq 0$ , else replace  $\mathbf{p}_n$   
 by  $\mathbf{p}_c$  and  $f(\mathbf{p}_n)$  by  $f(\mathbf{p}_c)$ , go to step 5.
- 4.3 Else, replace  $\mathbf{p}_n$  by  $\mathbf{p}_r$  and  $f(\mathbf{p}_n)$  by  $f(\mathbf{p}_r)$ .
- 5 If  $\sqrt{\sum_{i=0}^n (f(\mathbf{p}_i) - f(\bar{\mathbf{p}}))^2/n} < T$ , stop else go to step 2.

Figure 4.7: The Simplex optimization algorithm.

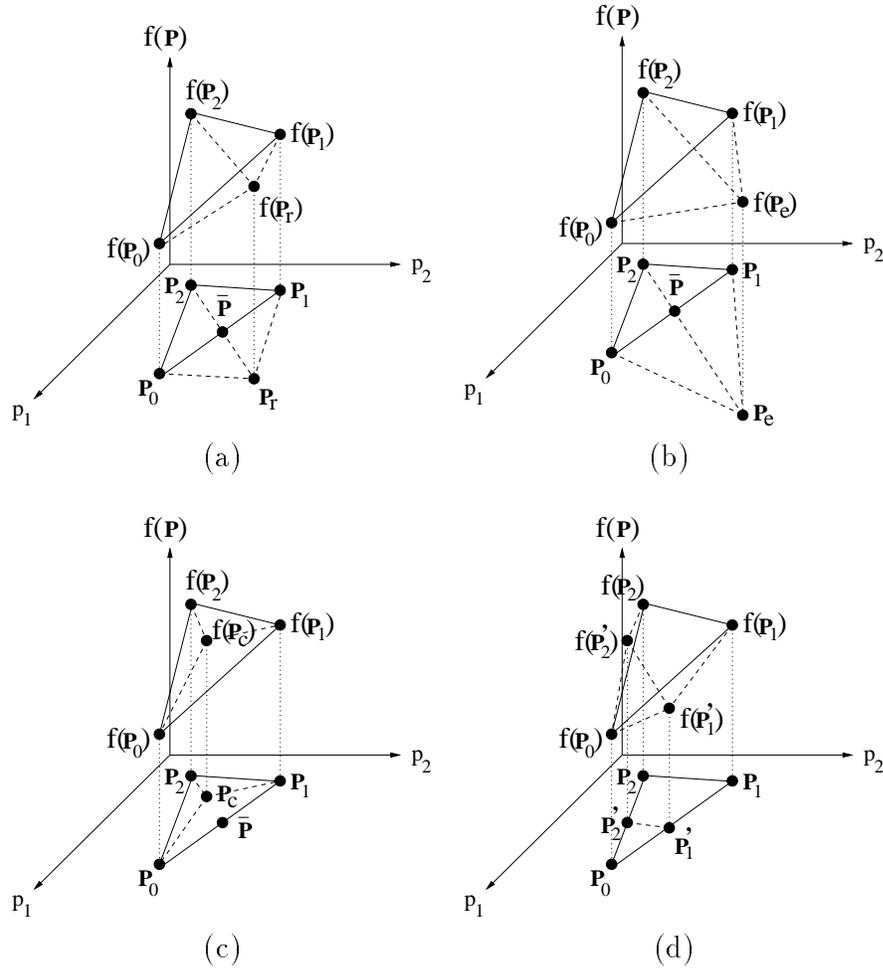


Figure 4.8: This figure shows four simplex operations in a two-dimensional parameter space. The solid lines denote the simplex before any operation and the dashed lines denote the simplex after the operation.  $\mathbf{p}_2$  and  $\mathbf{p}_0$  are the vertices for which the objective function  $f(\cdot)$  assumes the biggest and smallest values respectively, and  $\bar{\mathbf{p}} = \sum_{i=0}^2 \mathbf{p}_i / 3$  is the centroid of the two best vertices. The operations are (a) a reflection  $\mathbf{p}_r$  of  $\mathbf{p}_2$  with respect to the centroid point  $\bar{\mathbf{p}}$ , (b) an expansion  $\mathbf{p}_e$  of  $\mathbf{p}_2$  with respect to the centroid point  $\bar{\mathbf{p}}$ , (c) a contraction  $\mathbf{p}_c$  of  $\mathbf{p}_2$  with respect to the centroid point  $\bar{\mathbf{p}}$ , and (d) a shrinkage of all  $\mathbf{p}_i, i \neq 0$  toward  $\mathbf{p}_0$ . A local minimum can be obtained after an appropriate sequence of such operations.

# Chapter 5

## Architecture, File Formats, and Evaluation Methodology

In this section, we first describe the software architecture of the PSET package and the formats of the files used to communicate with the package. Next we show how this software package can be used to implement the five steps of the page segmentation evaluation methodology described in Chapter 3. Generic file format descriptions as well as specific examples are provided, for clearer understanding. This description of the architecture and file formats will allow users to i) understand the working of the PSET package, ii) replicate our results, iii) modify the parameter files for datasets, metrics, etc. and conduct their own evaluation experiments, iv) understand, maintain and improve the software, and v) evaluate new algorithms and compare the results with existing algorithms. The PSET package has been used to evaluate five page segmentation algorithms [12, 11].

### 5.1 Architecture and File Formats

The PSET package can be used to i) automatically train a given page segmentation algorithm, i.e., automatically select optimal algorithm parameters on a given training dataset, and ii) evaluate the page segmentation algorithm with the optimal parameters found in i) on a given test dataset. Figure 5.1 shows the overall architecture of the PSET package and illustrates these two functionalities.

The overall architecture shows all the input files that are needed to con-

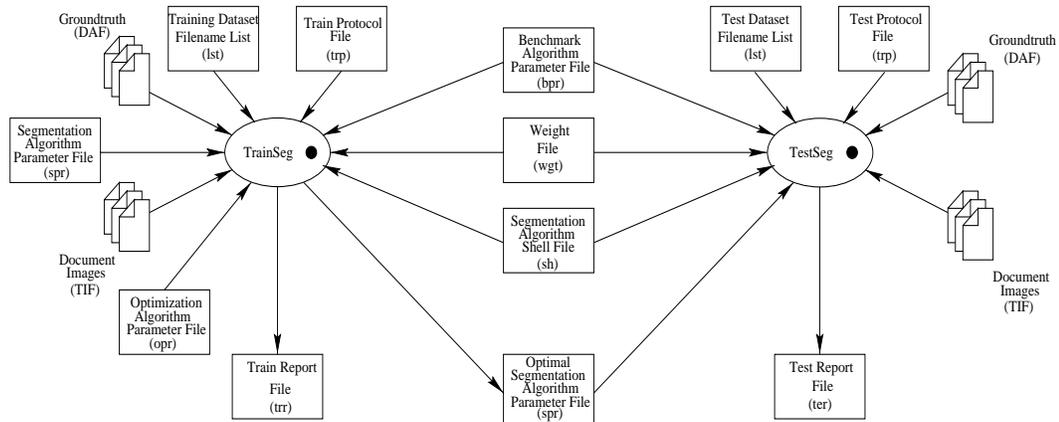


Figure 5.1: Overall PSET architecture. The left half of the architecture represents the training phase; the right half represents the testing phase. Note that in the testing phase, the optimal page segmentation parameter found in the training phase is used. The training and testing phases use the same performance metric related input files (benchmark algorithm parameter file (bpr) and weight file (wgt)) and the same segmentation algorithm shell file (sh).

duct the training and testing experiments for a given page segmentation algorithm, and all the output files generated by the training and testing procedures. Table 5.1 lists all the files used, their purposes, and their file name extensions.

Input files include various initial algorithm parameter files (an optimization algorithm parameter file (opr), a page segmentation algorithm parameter file (spr), and a benchmark algorithm parameter file (bpr)), dataset files (lst), a shell file (sh), and experimental protocol files (training protocol file (trp) and test protocol file (tep)). Users need to provide these files to the PSET package to conduct training or testing experiments. The output files of the training phase include a training report file (trr) and an optimal segmentation algorithm parameter file (spr). The training report file (trr) records intermediate as well as final training results of the training experiment. The optimal segmentation algorithm parameter file (spr) records the optimal segmentation algorithm parameter values found in the training phase. The output of the testing phase is a testing report file (ter), which records a set of error measures, timing and performance scores for each image in the test dataset,

Table 5.1: Summary of the file formats in the PSET package.

File Type	Extension	Description
Dataset List File	lst	It saves the root name of each image in a dataset.
Train Protocol File	trp	It saves the protocol parameters of the training experiment.
Test Protocol File	tep	It saves the protocol parameters of the testing experiment.
Segmentation Algorithm Parameter File	spr	It saves the parameters of a page segmentation algorithm that are to be trained.
Benchmarking Algorithm Parameter File	bpr	It saves all parameters of a benchmarking algorithm.
Optimization Algorithm Parameter File	opr	It saves all parameters of an optimization algorithm.
Groundtruth File	DAF	It saves document images and their groundtruth information.
Segmentation Result File	dafs	It saves document images and their segmentation results.
Train Report File	trr	It saves the training result of a segmentation algorithm.
Test Report File	ter	It saves the test result of a segmentation algorithm.
Weight File	wgt	It saves a set of weights for a set of error measures.
Segmentation Algorithm Shell File	sh	It saves a shell command for running segmentation algorithm executable. It is a Bourne shell program.

and a final average performance score over all images in the test dataset. Figure 5.2 shows various input file formats. Figure 5.3 shows the training report file format and Figure 5.4 shows the test report file format.

The parameter values in the parameter files are first read into the corresponding data structures inside the TrainSeg and the TestSeg modules as shown in Figure 5.5. The Train module shown in Figure 5.5(a) is shown at a finer level of detail in Figure 5.6, where the interaction of the optimization algorithm and the objective function computation module is illustrated. A detailed view of the *Objective Function Genscore* showing the interaction between the segmentation algorithm module and the performance metric computation module is shown in Figure 5.7(a). Finally, a blown-up view of the Test module shown in Figure 5.5(b) is shown in Figure 5.7 (b).

## 5.2 Implementing the Evaluation Methodology

In this section we show how a user can implement each step of the five-step evaluation methodology described in Chapter 3. Each variable in the methodology is mapped to a specific parameter file and each step is mapped to a specific group of modules in the package.

1. The training dataset  $\mathcal{T}$  is specified in the image root name list file (lst). The file name and location of the list file and the location of the







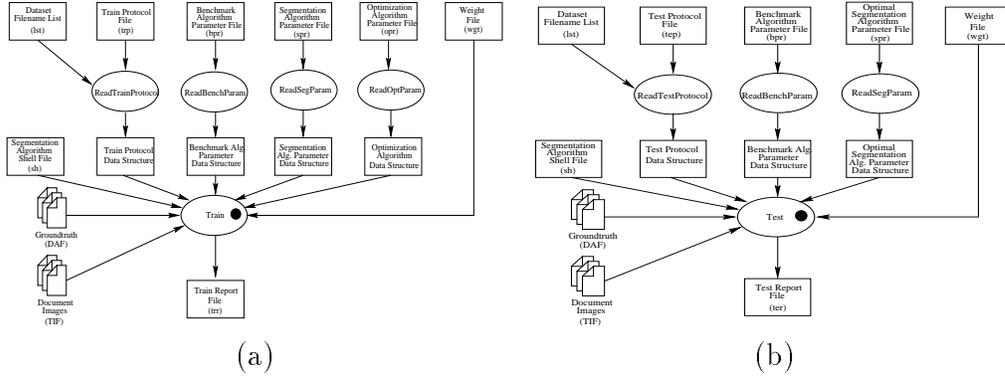


Figure 5.5: Parameter reading stage of the training phase (a) and the testing phase (b). At this level, various parameter files are read into their corresponding data structures which are fed into the Train and Test modules.

2. The performance metric  $\rho(I, G, R)$  is computed in module B, shown in Figures 5.7(a) and (b).  $(I, G)$  is an (image, groundtruth) pair, which is represented by two single pages in the architecture, and  $R$  is the segmentation result file represented by Segmentation Result (dafs). The error counter algorithm for generating a set of error measures is implemented in the *Bench* module. In the *BenchScoring* module, a weighted error measure  $1 - \rho(I, G, R)$  is computed. The formal definitions of error measures and performance metrics are given in the Appendix. To compute a performance metric, two input files, a benchmark Algorithm Parameter File (bpr) and a weight file (wgt), are required. Examples of these two files are shown in Figure 4.4. Users can substitute their own performance metrics and error counters in place of these two modules. However, this also requires that the users write a new *ReadBenchParam* module and define a new benchmark algorithm parameter data structure as shown in Figure 5.5.
3. The objective function  $f(\mathbf{p}^A; \mathcal{T}, A, \rho)$  is represented by the module C in Figure 5.7(a), where page segmentation algorithm  $A$  is represented by module A, the training dataset  $\mathcal{T}$  is specified in the train protocol parameter data structure, the computation of performance metric  $\rho$  is conducted in module B, and objective function parameter vector  $\mathbf{p}^A$  is represented by the segmentation algorithm parameter data

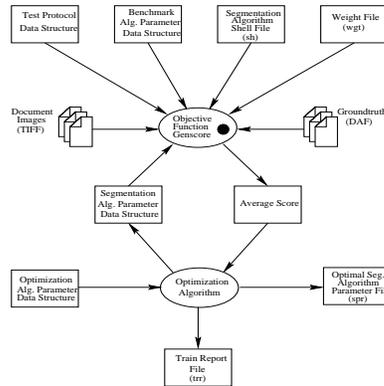


Figure 5.6: The Train module. In this module, the objective function is optimized over a given training dataset. Two files are generated by this module, a train report file (trr) and an optimal segmentation algorithm parameter file (spr).

structure in the architecture. The optimization procedure is shown in Figure 5.6 in a simplified representation. In addition, a benchmark algorithm parameter file (bpr), weight file (wgt), shell file (sh), list file (lst), training protocol file (trp), optimization algorithm parameter file (opr) and segmentation algorithm parameter file (spr) are required to conduct objective function optimization. Samples of opr and spr are shown in Figure 5.9. The generic file format of these sample files is shown in Figure 5.2.

The optimal objective function parameter vector  $\hat{\mathbf{p}}^A$  is stored in the file optimal segmentation algorithm parameter file (spr) shown in Figure 5.6. Users can substitute their own objective function in place of the architecture shown in Figure 5.7(a) and their own optimization algorithm module in the place of the *Optimization Algorithm* module shown in Figure 5.6. Again, they need to write new parameter reading functions and define corresponding data structures. This step generates two files, a training report file (trr) and an optimal segmentation algorithm parameter file (spr). Figure 5.10(a) shows a sample training report file.

4. After the optimal objective function parameter vector  $\hat{\mathbf{p}}^A$  has been found, the page segmentation algorithm is evaluated on a given test

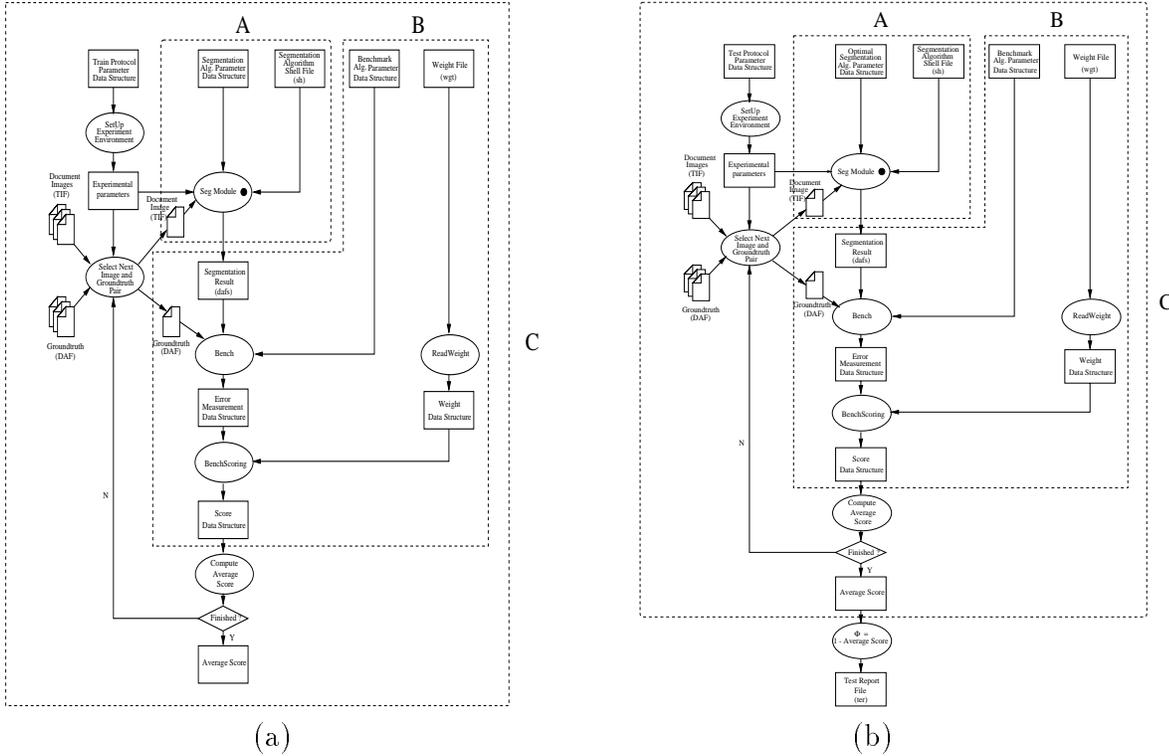


Figure 5.7: Software architectures of the objective function module and the test module. Module A represents the page segmentation algorithm module, module B represents the page segmentation error counter and scoring module, and module C represents the objective function module. The test module in (b) has sub-modules similar to those in (a). It also has a module for computing a final testing performance score (average textline accuracy).

dataset  $\mathcal{S}$ . Figure 5.7(b) shows the architecture of the test procedure. The test dataset  $\mathcal{S}$  is specified in the test protocol parameter data structure. Performance metric  $\rho$  is computed in module B. Note that module C here has the same architecture as module C in Figure 5.7(a). The computation of the final performance value  $\Phi$  is represented in module  $\Phi$ . Users can define their own  $\Phi$  function by changing the *Bench*, *BenchScoring*, *Compute Average Score*, and  $\Phi$  modules in Figure 5.7(b). This step generates a test report file (ter) which records a performance score for each image in the test dataset as well as a final average performance score over all images in the test dataset. Figure 5.10(b) shows a sample

<pre># Training experiment protocol # By: Song Mao # Feb. 21, 2000 # LAMP, UMCP  DATASET           = train.lst GROUNDTRUTH_DIR  = /fs/mirak2/LAMP/UWIII/ENGLISH/LINEWORD/DAFS/ IMG_DIR           = /fs/mirak2/LAMP/UWIII/ENGLISH/LINEWORD/IMAGEBIN/ GT_SUFFIX        = .DAF SG_SUFFIX        = .dafs IMG_SUFFIX       = BIN.TIF TRAIN_RESULT_DIR = ./ OPT_ALG          = simplex BEN_ALG          = textline_based SEG_ALG          = docstrum</pre>	<pre># Test experiment protocol # By: Song Mao # Feb. 21, 2000 # LAMP, UMCP  DATASET           = test.lst GROUNDTRUTH_DIR  = /fs/mirak2/LAMP/UWIII/ENGLISH/LINEWORD/DAFS/ IMG_DIR           = /fs/mirak2/LAMP/UWIII/ENGLISH/LINEWORD/IMAGEBIN/ GT_SUFFIX        = .DAF SG_SUFFIX        = .dafs IMG_SUFFIX       = BIN.TIF TEST_RESULT_DIR  = ./ BEN_ALG          = textline_based SEG_ALG          = xy cut</pre>
(a)	(b)

Figure 5.8: Sample protocol files. From both the train protocol file (a) and the test protocol file (b), we can see that the list files of the training dataset and test dataset are *train.lst* and *test.lst* respectively, the optimization algorithm used is the *Simplex* algorithm, the benchmarking algorithm used is the *Textline-based* algorithm, the page segmentation algorithm trained is the *Docstrum* algorithm, and the page segmentation algorithm tested is the *X-Y cut* algorithm. We can also find the locations of the groundtruth files, image files and training and test result files. Moreover, the suffixes for various files are given for file name manipulation in the PSET API.

test report file.

5. The statistical analysis of the test experimental results can be conducted using a standard statistics software package such as S-PLUS [3] or SPSS [6].

### 5.3 Algorithm Calling Mode in the Segmentation Algorithm Module

An important feature of the PSET package is that there are two page segmentation algorithm calling modes: function call and shell call. If the source code of a segmentation algorithm is available as a function, the user can link the function into the training and testing modules. In many cases, however, source code of a segmentation algorithm is not available, but executable code is. In such cases the shell calling mode can be used to run the segmentation algorithm from within the training or testing module. Furthermore, if a segmentation algorithm source code is not well debugged, e.g., if it leaks memory after each function call, the leaked memory can accumulate after

### 5.3. ALGORITHM CALLING MODE IN THE SEGMENTATION ALGORITHM MODULE39

```
# The Simplex Optimization
# Algorithm Parameters
NDIM      =      4
CRIFLG    =      nelder-mead
NMAX      =      500
FTOL      =      0.000001
ALPHA     =      1.0
BETA      =      0.5
GAMMA     =      2.0
SIGMA     =      0.5
P         =      100,80,100,50
SCALE     =      20,20,20,20
```

(a)

```
# The X-Y Cut Page Segmentation
# Algorithm Parameters
ALG_MODE  = func_call
TNX       = 100
TNY       = 80
TCX       = 100
TCY       = 50
```

(b)

Figure 5.9: Samples of an optimization algorithm parameter file (opr) and a segmentation algorithm parameter file (spr). A sample file for the Simplex optimization algorithm is shown in (a) and a sample file for the X-Y cut segmentation algorithm is shown in (b). Their detailed parameter descriptions can be found in [10].

many function calls and can finally cause algorithm crash at some point. The shell call mode is a good solution to this problem since in this case the executable code is used, and after each call all leaked memory is freed. The disadvantage of the shell call mode is that it can be slower than the function call mode. Figure 5.12 shows the architecture of the software implementation of these two calling modes. A shell file is required in the page segmentation algorithm shell call mode. A sample shell file is shown in Figure 5.11.



### 5.3. ALGORITHM CALLING MODE IN THE SEGMENTATION ALGORITHM MODULE41

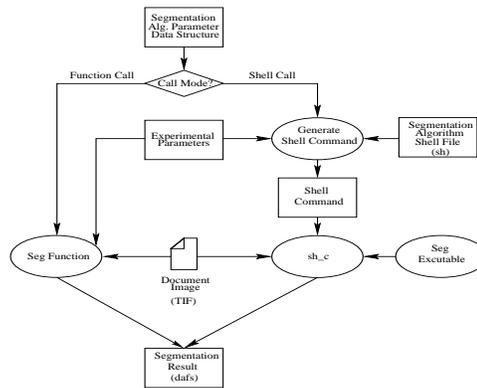


Figure 5.12: Page segmentation algorithm calling modes: function call and shell call. The left half represents the function calling mode and the right half represents the shell calling mode. The shell calling mode can be used only when the algorithm executable is available; otherwise the function calling mode can be used. Note that the executable is called by the function *sh\_c*.



# Chapter 6

## Tutorial

The PSET package provide C API functions for users to write their own applications. In this chapter, several examples are given for aiding users to understand how to perform some simple tasks by correctly calling C API functions and compiling their application code. These examples are designed to cover the usage of the basic algorithm modules.

There are two algorithm calling mode, function call and shell call, i.e. we can call an algorithm by its function or its executable. In this chapter, both the function call as well as shell call implementations are shown for some of examples.

### 6.1 Basic Integration Elements

The PSET package contains a training part and the a testing part that include the following components:

- 1 C header file (**pset.h**),
- 53 C function files (**Bench.c DocstrumFunction.c ...**),
- 1 makefile **Makefile** for making the PSET library.
- 14 sample parameter files for conducting training and testing experiments.
- 5 C main files. They will generate five tools , i.e., **TrainSeg**, **TestSeg**, **Docstrum**, **Voronoi**, **Xycut**.

- 5 related documents.

In addition to these component, dafs library and tiff library are also included in the PSET package. User can run the makefile under PSET directory to compile them and generate corresponding libraries **libdafs.a**, **libtiff.a** and **librutil.a**.

## 6.2 Development Environments

The PSET package is developed on Ultra 1,2 and 5 Sun workstations running the Solaris 2.6 operating system. The compiler used was GNU gcc 2.7.2.

## 6.3 Examples

In this section, a few sample code are provided. They cover the usage of the basic algorithm modules in the PSET package. The corresponding makefile is also given for showing users how to correctly compile the sample code.

### 6.3.1 Use a Page Segmentation Algorithm

In this section, we show how to use the X-Y cut page segmentation algorithm in both function call and shell call modes to segment a given TIFF image and obtain a segmentation result in DAFS format. Figure 6.1 shows the function call implementation and Figure 6.2 shows the UNIX command line usage. To use the X-Y cut algorithm function, users need to include the header file **pset.h** inside which all functions in the PSET package are described.

### 6.3.2 Use the Textline-Based Benchmarking Algorithm

In this section, we show how to use the textline-based benchmarking algorithm in the function call mode. The inputs to the algorithm is its parameter values, two DAFS files, one of which is the groundtruth DAFS file and the other is the segmentation result DAFS file. The output is a data structure what saves a set of error measures. a segmentation result in DAFS format. Figure 6.3 shows the function call implementation. It print out a set of error measure values. To use the textline-based algorithm function, users need to

```

#include <stdlib.h>
#include "pset.h"

int main(int argc, char **argv)
{
    XYCUT_PARAM *para;

    para = (XYCUT_PARAM *) malloc (sizeof(XYCUT_PARAM));
    para->Tnx = 50;
    para->Tny = 50;
    para->Tcx = 50;
    para->Tcy = 50;

    /* argv[1] pointer to a string that stores input image file name */
    /* argv[2] pointer to a string that stores output segmentation
    result file name */

    XycutFunction(argv[1], para, argv[2]);

    return(0);
}

```

Figure 6.1: **sample1.c**: A sample code to use the X-Y cut page segmentation algorithm by calling the algorithm function.

```
Xycut -a 50,50 -c 50,50 A001BIN.TIF A001.dafs
```

Figure 6.2: The UNIX command line usage of the X-Y cut page segmentation algorithm. A001BIN.TIF is the input image and A001.dafs is the output segmentation result file in DAFS format.

include the header file **pset.h** inside which all functions in the PSET package are described.

Figure 6.4 shows the makefile for both samples shown in Figure 6.1 and Figure 6.3.

```

#include <stdio.h>
#include <stdlib.h>
#include "pset.h"

int main(int argc, char **argv)
{
    BENCH_PARAM *para;
    ERR_MEASURE *err_measure;

    para = (BENCH_PARAM *) malloc (sizeof(BENCH_PARAM));
    err_measure = (ERR_MEASURE *) malloc (sizeof(ERR_MEASURE));

    para->htol = 90;
    para->vtol = 80;
    para->hpix = 11;
    para->vpix = 8;

    /* argv[1] pointer to a string that stores input image file name */
    /* argv[2] pointer to a string that stores groundtruth file name */
    /* argv[3] pointer to a string that stores segmentation result file name */

    Bench(argv[1], para, argv[2], argv[3], &err_measure);

    printf("nSpl = %d\n", err_measure->nSpl);
    printf("nMrg = %d\n", err_measure->nMrg);
    printf("nMis = %d\n", err_measure->nMis);
    printf("nFA = %d\n", err_measure->nFA);
    printf("nSplLine = %d\n", err_measure->nSplLine);
    printf("nMrgLine = %d\n", err_measure->nMrgLine);
    printf("nMisLine = %d\n", err_measure->nMisLine);
    printf("nFAZone = %d\n", err_measure->nFAZone);
    printf("nErrLine = %d\n", err_measure->nErrLine);
    printf("nGtLine = %d\n", err_measure->nGtLine);

    free(para);
    free(err_measure);
    return(0);
}

```

Figure 6.3: **sample2.c**: A sample code to use the textline-based benchmarking algorithm by calling the algorithm function.

```

CC = gcc

CFLAGS= -O3 -g

INCDIRS = -I ../include -I../dafs/src/tiff/ \
          -I ../dafs/src/dafslib/ -I ../dafs/src/rutil/
LIBDIRS= -L ../lib -L../dafs/src/tiff/obj/ \
          -L../dafs/src/dafslib/obj/ -L../dafs/src/rutil/obj/

INCS= $(INCDIRS)
LIBS= $(LIBDIRS) -lpset -ldafs -ltiff -lrutil -lm

SRCS = sample1.c sample2.c

OBJS = $(SRCS:.c=.o)

.c.o:
    $(CC) $(CFLAGS) $(INCS) -c $<

all: sample1 sample2

sample1: $(OBJS)
    $(CC) $(CFLAGS) -o $@ $(INCS) sample1.o $(LIBS)

sample2: $(OBJS)
    $(CC) $(CFLAGS) -o $@ $(INCS) sample2.o $(LIBS)

clean:
    \rm -f $(OBJS) core

depend:
    makedepend -- $(INCS) -- $(SRCS)

# DO NOT DELETE

```

Figure 6.4: The makefile for the code in Figure 6.3 and Figure 6.1.



# Chapter 7

## Command Line Specification

In this chapter, detailed description of command line in the PSET package is given. There are total five commands in the PSET package that are described in the following sections.

### 7.1 The X-Y Cut Page Segmentation Algorithm Command

The command line is:

```
Xycut -a <val,val> -c <val,val> <img> <seg_file>.
```

The description is shown as follows:

**Xycut** The X-Y cut page segmentation algorithm executable.

**-a <val,val>** Noise removal threshold. The first **val** denotes noise removal threshold in the vertical direction in an image, whereas the second **val** denotes noise removal threshold in the horizontal direction in an image, the unit of these threshold are in pixels assuming image has a resolution of 300 dpi. For the image with different resolution, these thresholds should be scaled accordingly.

**-c <val,val>** Widest zero valley width cut threshold. The first **val** denotes widest zero valley width cut threshold in the vertical direction in an image, whereas the second **val** denotes widest zero valley width cut threshold in the horizontal direction in an image, the unit of these threshold are in pixels assuming image has a resolution of 300 dpi. For

the image with different resolution, these thresholds should be scaled accordingly.

<img> Input image file name. The format of the image should be TIFF.

<seg\_file> Output segmentation result file name. The format of the output file is DAFS.

## 7.2 The Docstrum Page Segmentation Algorithm Command

The command line is:

```
Docstrum -t <val> -p <val> -u <val> -d <val>
        <img> <cc_file> <seg_file>.
```

The description is shown as follows:

-t <val> Nearest neighbor threshold factor parameter.

-p <val> Textline parallel distance threshold factor parameter.

-u <val> Textline perpendicular distance threshold factor parameter.

-d <val> Connected component size ratio factor parameter.

<img> Input image file name. The format of the image should be TIFF.

<cc\_file> Input connected component file name. It is an ascii file where the coordinates bounding boxes of connected components are stored.

<seg\_files> Output segmentation result file name. The format of the output file is DAFS.

## 7.3 The Voronoi-Based Segmentation Algorithm Command

The command line is:

```
Voronoi -s <val> -n <val> -f <val> -t <val>
        <img> <seg_file>.
```

The description is shown as follows:

- s <val> Sampling rate parameter.
- n <val> Maximum size threshold of noise connected component parameter, the unit of it is pixel.
- f <val> Margin control factor parameter for interline spacing estimate.
- t <val> Connected component area ratio factor parameter.
- <img> Input image file name. The format of the image should be TIFF.
- <seg\_files> Output segmentation result file name. The format of the output file is DAFS.

## 7.4 The Algorithm Training Command

The command line is:

```
TrainSeg -p <train_protocol_file> -o <opt_para_file>
        -b <bench_param_file> -s <seg_param_file>
        -w <weight_file> -t <TrainReport>
        -r <Optimal_Seg_Parameter_file>
```

The description is shown as follows:

**TrainSeg** The algorithm training package executable.

- p <train\_protocol\_file> Input training protocol file name.
- o <opt\_para\_file> The parameter file name of the optimization algorithm that is used in the training step.
- b <bench\_param\_file> The parameter file name of the benchmarking algorithm that is used in the training step.
- s <seg\_param\_file> The parameter file name of the page segmentation algorithm.
- w <weight\_file> The the weight file name.
- t <TrainReport> The output training report file name.
- r <Optimal\_Seg\_Parameter\_file> The output optimal segmentation algorithm parameter file.

## 7.5 The Algorithm Testing Command

The command line is:

```
TestSeg -p <test_protocol_file> -b <bench_param_file>  
        -s <seg_param_file> -w <weight_file> -t <TestReport>
```

The description is shown as follows:

**TestSeg** The algorithm testing package executable.

**-p <test\_protocol\_file>** Input testing protocol file name.

**-b <bench\_param\_file>** The parameter file name of the benchmarking algorithm that is used in the training step.

**-s <seg\_param\_file>** The parameter file name of the page segmentation algorithm.

**-w <weight\_file>** The the weight file name.

**-t <TestReport>** The output testing report file name.

# Chapter 8

## Data Structures

In this chapter, data structures that are crucial for user interface are described in detail. Each data structure is described in the following categories: data structure itself, brief explanation, data structure members, comments and related data structures.

*TRAIN\_PROTOCOL Structure*

```

typedef struct train_protocol {
    DATASET *dataset;
    char *gt_dir;
    char *img_dir;
    char *gt_suffix;
    char *sg_suffix;
    char *img_suffix;
    char *train_result_dir;
    char opt_alg;
    char ben_alg;
    char seg_alg;
} TRAIN_PROTOCOL;

```

This data structure contains all information needed for training setup, such as the dataset used, where the input should come from and where the output should go, and which optimization, benchmark and segmentation algorithms should be used. Defined in: TrainSeg.h

**Members**

**dataset** Pointer to a data structure storing the image file name and total number of image files in a training dataset.

**gt\_dir** Specify the location of groundtruth files.

**img\_dir** Specify the location of image files.

**gt\_suffix** Specify the suffix of a groundtruth file, e.g. the suffix of the groundtruth file A001.DAF is .DAF.

**sg\_suffix** Specify the suffix of a segmentation result file, e.g. the suffix of the segmentation result file A001.dafs is .dafs.

**img\_suffix** Specify the suffix of a image file, e.g. the suffix of the image file A001.BIN.TIF is .BIN.TIF.

**train\_result\_dir** Specify the location where a user want to store the training result file.

**opt\_alg** Specify which optimization algorithm is used in this training process, currently the available optimization algorithm is “simplex”.

**ben\_alg** Specify which benchmark algorithm is used in this training process, currently the available optimization algorithm is “textline-based”.

**seg\_alg** Specify which segmentation algorithm is used in this training process, currently the available optimization algorithm is “xycut”, “doctrum” and “voronoi”.

### **Comments**

- User should specify the training protocol in terms of members of this data structure which covers sufficient information for somebody else to repeat the training result.

### **See Also**

**TEST\_PROTOCOL**

*DATASET Structure*

```
typedef struct dataset {  
    int nImg;  
    char **ImgName;  
} DATASET;
```

This data structure contains image file names and the total number of image files in the dataset it represents. Defined in: TrainSeg.h

**Members**

**nImg** Variable that stores the total number of image files in the dataset.

**ImgName** Pointer to an array each of which points to a image filename string.

**Comments**

- This data structure specifies the dataset used, user can use different dataset and store the its information in this data structure, e.g. in test process, test dataset can be used and in training process, training dataset can be used.

**See Also**

*BENCH\_PARAM Structure*

```

typedef struct bench_param {
    int htol;
    int vtol;
    int hpix;
    int vpix;
    int wSpl;
    int wMrg;
    int wFA;
    int wSplLine;
    int wMrgLine;
    int wMisLine;
    int wFAZone;
} BENCH_PARAM;

```

This data structure contains parameters needed in benchmark (textline-based) algorithm. Defined in: Bench.h

**Members**

**htol** Variable that stores horizontal overlapping tolerance threshold in percent. It controls if a overlap between two rectangular regions is significant or not in horizontally direction.

**vtol** Same as htol except in vertical direction.

**hpix** Same as htol except in pixel values.

**vpix** Same as hpix except in vertical direction.

**wSpl** Weight for number of split errors.

**wMrg** Weight for number of horizontally merge errors

**wFA** Weight for number of False-Alarm errors.

**wSplLine** Weight for number of split groundtruth textlines.

**wMrgLine** Weight for number of horizontally merged groundtruth textlines.

**wMisLine** Weight for number of Mis-detected groundtruth textlines.

**wFAZone** Weight for number of False-Alarm regions.

### **Comments**

- This data structure stores textline-based benchmark algorithm parameter values. For a different benchmark algorithm, user shall define a different data structure for it.

### **See Also**

**xycut\_param**

*TEXTLINE Structure*

```

typedef struct textline {
    int SG_ID[MAX_SG];
    int SG_TYPE[MAX_SG];
    int sg_index;
    int iBox;
    int miss_det;
    int h_split;
    int h_merge;
    int v_split;
    int v_merge;
    int textline_flg;
} TEXTLINE;

```

This data structure contains all information of a segmented textline, including physical layout, line ID, relation between the line and other “touching” segmented zones, error types and error numbers. It is used in textline-based benchmark algorithm. Defined in: Bench.h

**Members**

**SG\_ID**[MAX\_SG ] Array that stores the ID of segmented zones that “touches” the current textline.

**SG\_TYPE**[MAX\_SG ] Array that stores the types of “touch” between the current textline and a zone which can be “enclose”, “split” and “apart”.

**sg\_index** Variable that stores the index of the current textline.

**box** A data structure that stores physical layout (bounding box) of the current textline.

**miss\_det** Variable that stores the total number of mis-detected errors that happen on the current textline.

**h\_split** Variable that stores the total number of horizontal split errors that happen on the current textline.

**h\_merge** Variable that stores the total number of horizontal merge errors that happen on the current textline.

**v\_split** Variable that stores the total number of vertical split errors that happen on the current textline.

**v\_merge** Variable that stores the total number of vertical merge errors that happen on the current textline.

**textline\_flg** Variable that signifies if the current textline is a real textline or not.

### Comments

- This data structure stores all information needed for a segmented textline to benchmark the segmentation algorithm.

### See Also

## *ZONE Structure*

```
typedef struct zone {
    iBox box;
    int false_alarm;
    int line_counter;
} ZONE;
```

This data structure contains all information of segmented zones including physical layout, false-alarm flag and the total number of lines “touched” by the current zone. Defined in: Bench.h

## **Members**

**box** Data structure that stores the physical layout (bounding box) information of the current zone.

**ImgName** Pointer to an array each of which points to a image filename string.

**false\_alarm** The flag that signals if the current zone is a false-alarm zone or not.

**line\_counter** Variable that stores the total number of segmented lines that the current zone “touches”.

## **Comments**

- This data structure stores all information needed for a segmented zone to benchmark the segmentation algorithm.

## **See Also**

*ERR\_MEASURE Structure*

```
typedef struct err_measure {
    int nSpl;
    int nMrg;
    int nMis;
    int nFA;
    int nSplLine;
    int nMrgLine;
    int nMisLine;
    int nFAZone;
    int nErrLine;
    int nGtLine;
} ERR_MEASURE;
```

This data structure contains all types of error measures made on groundtruth textlines. Defined in: Bench.h

**Members**

**nSpl** Variable that stores the total number of splits.

**nMrg** Variable that stores the total number of horizontal merges.

**nMis** Variable that stores the total number of mis-detects.

**nFA** Variable that stores the total number of false-alarms.

**nSplLine** Variable that stores the total number of split groundtruth textlines.

**nMrgLine** Variable that stores the total number of horizontally merged textlines.

**nMisLine** Variable that stores the total number of mis-detected textlines.

**nFAZone** Variable that stores the total number of false-alarm regions.

**nErrLine** Variable that stores the total number of incorrectly segmented textlines (here incorrectly segmented textlines are split, horizontally merged or mis-detected groundtruth textlines).

**nGtLine** Variable that stores the total number of groundtruth textlines.

### **Comments**

- These error measure types are based on the textline-based benchmark metric. When user comes up with a new metric, a new set of error measures have to be used. Moreover, by using different weight on different error measure type, user can customize the scoring function.

### **See Also**

*DOCSTRUM\_PARAM Structure*

```

typedef struct docstrum_param {
    char alg_mode;
    char *concom_dir;
    char *concom_suffix;
    int knn;
    int lsize;
    int hsize;
    int hatol;
    int vatol;
    int tcc;
    int par;
    int per;
    int scr;
    int dis;
} DOCSTRUM_PARAM;

```

This data structure contains all parameters of the Docstrum parameter based on O’Gorman’s implementation ???. Defined in: DocstrumFunction.h

**Members**

**alg\_mode** Variable that indicate if the algorithm is called by its function or by its executable, possible values are “FUNC\_CALL” and “SHELL\_CALL”.

**\*concom\_dir** Pointer to a connected component file name string. Connected component bounding box is pre-generated and stored in a connected component file.

**\*concom\_suffix** Pointer to a connected component file name suffix string, e.g. for a file named A001.concom, its suffix is “.concom”.

**knn** Variable that stores the total number of nearest neighbors needed for clustering connected components into textlines.

**lsize** Variable that stores the minimal size (height in pixels) of connected component accepted as possible characters.

**hsize** Variable that stores the maximal size (height in pixels) of connected component accepted as possible characters.

**hatol** Variable that stores the horizontal angle tolerance threshold for clustering connected components into textlines.

**vatol** Variable that stores the vertical angle tolerance threshold for clustering connected components into textlines.

**tcc** Variable that stores the nearest neighbor threshold factor. It times estimated character spacing and the result is the character spacing threshold to be used for clustering into a textline.

**par** Variable that stores the parallel distance threshold factor. It times estimated character spacing the result is the spacing threshold to be used for clustering textlines into zones.

**per** Variable that stores the perpendicular distance threshold factor. It times estimated character spacing the result is the spacing threshold to be used for clustering textlines into zones.

**src** Variable that stores the superscript and subscript character distance threshold factor. It is used to include some close superscript and subscript of a character inside a same textline even though the angle between them exceeds angle tolerance thresholds.

**dis** Variable that stores the connected component size ratio factor. This factor is used to divide connected component into two group with very different size (height in pixels).

## Comments

- The first three parameters are environment related parameters and the rest are algorithm related parameters.

## See Also

**XYCUT\_PARAM**

**VORONOI\_PARAM**

*EST\_PARA Structure*

```
typedef struct est_para {  
    double or;  
    int cs;  
    int ts;  
} EST_PARA;
```

This data structure contains the parameters to be estimated in the Docstrum segmentation algorithm using distance-angle pairs `??`. Defined in:

DocstrumFunction.h

**Members**

**or** Variable that stores skew angle (in degrees) of a document.

**cs** Variable that stores average character spacing.

**ts** Variable that stores average textline spacing.

**Comments**

- 

**See Also**

**XYCUT\_PARAM**

**VORONOL\_PARAM**

*SIMPLEX\_PARAM Structure*

```

typedef struct simplex_param {
    int ndim;
    int criflg;
    int nmax;
    float ftol;
    float alpha;
    float beta;
    float gamma;
    float sigma;
    float **p;
    float *y;
    float *t;
    float *scale;
} SIMPLEX_PARAM;

```

This data structure contains the parameters of the Simplex optimization algorithm. Defined in: Simplex.h

**Members**

**ndim** Variable that stores the dimensionality of objective function to be optimized.

**criflg** Variable that indicate which stop-criterion to use, possible values are NM (Nelder-Mead) or NR (Numerical Recipe).

**nmax** Variable that stores maximum number of function evaluations allowed.

**ftol** Variable that stores minimal objective function values different. If the difference is greater than it, stop.

**alpha** Variable that stores simplex reflection coefficient.

**beta** Variable that stores simplex contraction coefficient.

**gamma** Variable that stores simplex expansion coefficient.

**sigma** Variable that stores simplex shrinkage coefficient.

**\*\*p** Variable that stores objective function variable vector.

**\*y** Variable that stores objective function value vector.

**\*t** Variable that stores timing information vector.

**\*scale** Variable that stores initial simplex scale vector.

### Comments

- Some parameters are general to any objective function optimization problem, e.g. “ndim”, “ftol”, “nmax”, “\*\*p”, “\*y” and “t”. If another optimization algorithm is used, one probably need these parameters. The other parameters are unique to the Simplex optimization algorithm.

### See Also

### *VORONOI\_PARAM Structure*

```

typedef struct voronoi_param {
    char alg_mode;
    int sr;
    int nm;
    float fr;
    int ta;
    int sw;
    int Cw;
    int Ch;
    int Cr;
    int Az;
    int Al;
    int Br;
} VORONOI_PARAM;

```

This data structure contains the parameters of the Voronoi-based segmentation algorithm ???. Defined in: VoronoiFunction.h

### Members

- alg\_mode** Variable that indicate if the algorithm is called by its function or by its executable, possible values are “FUNC\_CALL” and “SHELL\_CALL”.
- sr** Variable that stores the sampling rate on the border of connected components.
- nm** Variable that stores the maximum size threshold of noise connected component.
- fr** Variable that stores the margin control factor for inter-line spacing estimate.
- ta** Variable that stores the area ratio threshold between connected components.
- sw** Variable that stores the size of smoothing window for inter-character and inter-line spacing estimates.

**Cw** Variable that stores the maximum width threshold of connected component.

**Ch** Variable that stores the maximum height threshold of connected component.

**Cr** Variable that stores the maximum connected component aspect ratio threshold.

**Az** Variable that stores the minimum zone area threshold.

**Al** Variable that stores the minimum area threshold for the zones that are vertical and elongated.

**Br** Variable that stores the maximum aspect ratio threshold for zones that are vertical and elongated.

### **Comments**

- The Voronoi-based segmentation algorithm is based on ??.

### **See Also**

*XYCUT\_PARAM Structure*

```

typedef struct xycut_param {
    char alg_mode;
    int Tnx;
    int Tny;
    int Tcx;
    int Tcy;
} XYCUT_PARAM;

```

This data structure contains the parameters of the X-Y cut segmentation algorithm ???. Defined in: XycutFunction.h

**Members**

**alg\_mode** Variable that indicate if the algorithm is called by its function or by its executable, possible values are “FUNC\_CALL” and “SHELL\_CALL”.

**Tnx** Variable that stores the X widest zero valley width threshold in pixels.

**Tny** Variable that stores the Y widest zero valley width threshold in pixels.

**Tcx** Vertical noise removal threshold in pixels.

**Tcy** Horizontal noise removal threshold in pixels.

**Comments**

- The X-Y cut segmentation algorithm is based on ???.

**See Also**

*TEST\_PROTOCOL Structure*

```
typedef struct test_protocol {  
    DATASET *dataset;  
    char *gt_dir;  
    char *img_dir;  
    char *gt_suffix;  
    char *sg_suffix;  
    char *img_suffix;  
    char *test_result_dir;  
    char ben_alg;  
    char seg_alg;  
} TEST_PROTOCOL;
```

This data structure contains all information needed for test setup, such as the dataset used, where the input should come from and where the output should go, and which benchmark and segmentation algorithms should be used. Defined in: TestSeg.h

**Members**

**dataset** Pointer to a data structure storing the image file name and total number of image files in a test dataset.

**gt\_dir** Specify the location of groundtruth files.

**img\_dir** Specify the location of image files.

**gt\_suffix** Specify the suffix of a groundtruth file, e.g. the suffix of the groundtruth file A001.DAF is .DAF.

**sg\_suffix** Specify the suffix of a segmentation result file, e.g. the suffix of the segmentation result file A001.dafs is .dafs.

**img\_suffix** Specify the suffix of a image file, e.g. the suffix of the image file A001.BIN.TIF is .BIN.TIF.

**test\_result\_dir** Specify the location where a user want to store the test result file.

**ben\_alg** Specify which benchmark algorithm is used in this training process, currently the available optimization algorithm is “textline-based”.

**seg\_alg** Specify which segmentation algorithm is used in this training process, currently the available optimization algorithm is “xycut”, “doctrum” and “voronoi”.

### **Comments**

- User should specify the test protocol in terms of members of this data structure which covers sufficient information for somebody else to repeat the test result.

### **See Also**

**TEST\_PROTOCOL**

*WEIGHT Structure*

```
typedef struct weight {
    int wSpl;
    int wMrg;
    int wMis;
    int wFA;
    int wSplLine;
    int wMrgLine;
    int wMisLine;
    int wFAZone;
} WEIGHT;
```

This data structure contains a weight values for each error measure member except `nErrLine` and `nGtLine` in error measure data structure. Defined in:

Bench.h

**Members**

**wSpl** Weight for error measure data structure member **nSpl**.

**wMrg** Weight for error measure data structure member **nMrg**.

**wMis** Weight for error measure data structure member **nMis**.

**wFA** Weight for error measure data structure member **nFA**.

**wSplLine** Weight for error measure data structure member **nSplLine**.

**wMrgLine** Weight for error measure data structure member **nMrgLine**.

**wMisLine** Weight for error measure data structure member **nMisLine**.

**wFAZone** Weight for error measure data structure member **nFAZone**.

**Comments**

- The weight and error measure is based textline-based metric ??.

**See Also**

## Chapter 9

# Function Specifications

In this chapter, function specification for each function in the software package except Voronoi-based segmentation algorithm functions is given. Each function is modular, implemented in C and can be called by users for their specific evaluation task. Each function is described in the following categories: function explanation, return value, parameter description, comments and related functions.

*Amotry*

```
float Amotry(TRAIN_PROTOCOL *train_protocol, void *seg_param,
void *ben_param, char *weight_file, SIMPLEX_PARAM *simplex_param,
float *psum, void (*genscore)(), int ihi, int *nFuncEval_p, float fac, FILE
*fp)
```

The Amotry function is used by Simplex function to do simplex operations, e.g. reflection, expansion, contraction and shrinkage. It returns the adjusted objective function value.

Defined in: Simplex.c

**Return Value**

float.

**Parameters**

**train\_protocol** Data structure to hold training protocol parameter values(**TrainSeg.h**).

**seg\_param** Data structure to hold segmentation algorithm parameter values.

**ben\_param** Data structure to hold benchmark algorithm parameter values.

**weight\_file** Pointer to a string that stores weight file name.

**simplex\_param** Data structure to hold Nelder-Mead algorithm parameter values.

**psum** Objective function variable vectors sum of the simplex.

**genscore** Objective function to be optimized.

**ihi** Highest objective function value array index.

**nFuncEval\_p** Pointer to a variable holding number of objective function evaluations.

**fac** Simplex factor.

**fp** File pointer of a file that saves the training results.

### **Comments**

- 

**See Also**

*Bench*

```
char *img, void Bench(BENCH_PARA *para, char *gtdafs, char *sgdafs,  
ERR_MEASURE **err_measure_p )
```

The Bench function is used to get a set of error measurements by comparing groundtruth and segmentation result.

Defined in: GenscoreXycut.h GenscoreDocstrum.h GenscoreVoronoi.h

**Return Value**

void

**Parameters**

**img** Image root name.

**para** Data structure that stores Benchmarking algorithm parameter values.

**gtdafs** File name of a groundtruth file.

**sgdafs** File name of a segmentation result file.

**err\_measure\_p** Pointer to a data structure storing error measurement values.

**Comments**

- This function is based on Mao and Kanungo's textline based performance metric.

**See Also**

*BenchScoring*

**float BenchScoring**(**BENCH\_PARAM** \*para, **ERR\_MEASURE** \*err, **WEIGHT** \*weight)

The BenchScoring function is used to get a score from a set of error measurements and user specified weights. The returned value is a score data structure which stores both metric value and timing value.

Defined in: Bench.h

**Return Value**

float.

**Parameters**

**para** Data structure to hold Benchmarking algorithm parameter values.

**err\_measure** Data structure storing error measurement values.

**weight** Data Structure storing a set of weight values.

**Comments**

- This function is can be customized by users to have their own scoring function.

**See Also**

**Bench.c**

*CalendarTime***char \*CalendarTime()**

The CalendarTime function is used to get a formatted calendar time output, e.g. 03/08/2000/ 15:23:38.

Defined in: WriteTrainReportHeader.h

**Return Value**

char \*.

**Parameters****Comments**

- 

**See Also**

WriteTrainReportHeader.c

WriteTestReportHeader.c

*DocstrumFunction*

```
int DocstrumFunction(char *tif_file, char *cc_file, DOCSTRUM_PARAM
*para, char *dafs_file)
```

The DocstrumFunction function is an implementation of the Docstrum segmentation algorithm developed by O’Gorman [15].

Defined in: GenscoreDocstrum.h

**Return Value**

**int.**

**Parameters**

**tif\_file** Input image file name.

**cc\_file** Connected components coordinate file name.

**para** Data structure to hold Docstrum algorithm parameter values.

**dafs\_file** Segmentation result file name (in dafs format).

**Comments**

- This algorithm is a bottom-up page segmentation algorithm ??.
- It only deals with text blocks.

**See Also**

**XycutFunction**

**VoronoiFunction**

*EvalDocstrum*

```
void EvalDocstrum(TEST_PROTOCOL *test_protocol, void *docstrum_param,  
void *bench_param, char *weight_file, char **test_report_p)
```

The EvalDocstrum function is used to evaluate the Docstrum segmentation algorithm on test dataset in the test phase.

Defined in: Test.h

**Return Value**

void.

**Parameters**

**test\_protocol** Data structure to hold test protocol parameter values.

**docstrum\_param** Data structure to hold the Docstrum algorithm parameter values.

**ben\_param** Data structure to hold benchmark algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**test\_report\_p** Pointer to a pointer that points to a string that stores the test report file name.

**Comments**

- User can come up with their own evaluation function of his algorithm.

**See Also**

EvalXycut

EvalVoronoi

DocstrumFunction

*EvalVoronoi*

```
void EvalVoronoi(TEST_PROTOCOL *test_protocol, void *voronoi_param,
void *bench_param, char *weight_file, char **test_report_p)
```

The EvalVoronoi function is used to evaluate the Voronoi-based segmentation algorithm on test dataset in the test phase.

Defined in: Test.h

**Return Value**

**void**.

**Parameters**

**test\_protocol** Data structure to hold test protocol parameter values.

**voronoi\_param** Data structure to hold the Docstrum algorithm parameter values.

**bench\_param** Data structure to hold benchmark algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**test\_report\_p** Pointer to a pointer that points to a string that stores the test report file name.

**Comments**

- User can come up with their own evaluation function of his algorithm.

**See Also**

**EvalDocstrum**

**EvalXycut**

**VoronoiFunction**

**Voronoi**

*EvalXycut*

```
void EvalXycut(TEST_PROTOCOL *test_protocol, void *xycut_param,  
void *bench_param, char *weight_file, char **test_report_p)
```

The EvalXycut function is used to evaluate the X-Y cut segmentation algorithm on test dataset in the test phase.

Defined in: Test.h

**Return Value**

void.

**Parameters**

**test\_protocol** Data structure to hold test protocol parameter values.

**xycut\_param** Data structure to hold the X-Y cut algorithm parameter values.

**bench\_param** Data structure to hold benchmark algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**test\_report\_p** Pointer to a pointer that points to a string that stores the test report file name.

**Comments**

- User can come up with their own evaluation function of his algorithm.

**See Also**

**EvalDocstrum**

**EvalVoronoi**

**XycutFunction**

*FreeJobTest*

```
void FreeJobTest(TEST_PROTOCOL *test_protocol, char *test_report,
char *test_protocol_file, void *ben_param, void *seg_param, char *seg_file,
char *ben_file, char *weight_file)
```

The FreeJobTest function is used to free any allocated memory that have not been freed in testing phase.

Defined in: TestSeg.h

**Return Value**

void.

**Parameters**

**test\_protocol** Data structure to hold testing protocol parameter values.

**test\_report** String array to hold testing report file name.

**test\_protocol\_file** String array to hold testing protocol parameter file name.

**ben\_param** Data structure to hold benchmark algorithm parameter values.

**seg\_param** Data structure to hold segmentation algorithm parameter values.

**seg\_file** String array to hold segmentation algorithm parameter file name.

**ben\_file** String array to hold benchmark algorithm parameter file name.

**weight\_file** Pointer to a string storing weight file name.

**Comments**

- 

**See Also**

**FreeJobTrain**

*FreeJobTrain*

```
void FreeJobTrain(TRAIN_PROTOCOL *train_protocol, char *train_report,  
char *opt_seg_param, char *train_protocol_file, void *opt_param, void *ben_param,  
void *seg_param, char *opt_file, char *seg_file, char *ben_file, char *weight_file)
```

The FreeJobTrain function is used to free any allocated memory that have not been freed in training phase.

Defined in: TrainSeg.h

**Return Value**

void.

**Parameters**

**train\_protocol** Data structure to hold training protocol parameter values.

**train\_report** Pointer to a train report file name string.

**opt\_seg\_param** Pointer to a file name string of a file storing optimal segmentation algorithm parameters.

**train\_protocol\_file** String array to hold training protocol parameter file name.

**opt\_param** Data structure to hold optimization algorithm parameter values.

**ben\_param** Data structure to hold benchmark algorithm parameter values.

**seg\_param** Data structure to hold segmentation algorithm parameter values.

**opt\_file** String array to hold optimization algorithm parameter file name.

**seg\_file** String array to hold segmentation algorithm parameter file name.

**ben\_file** String array to hold benchmark algorithm parameter file name.

**weight\_file** Pointer to a string storing weight file name.

## Comments

- 

## See Also

FreeJobTest

*GenscoreDocstrum*

```
void GenscoreDocstrum(TRAIN_PROTOCOL *train_protocol, void
*docstrum_param, void *bench_param, char *weight_file, float *val, SCORE
**score_p)
```

The GenscoreDocstrum function is used to generate a score from the Docstrum segmentation algorithm segmentation result and the corresponding groundtruth using a benchmark algorithm.

Defined in: Train.h

**Return Value**

**void.**

**Parameters**

**train\_protocol** Data structure to hold training protocol parameter values.

**docstrum\_param** Data structure to hold Docstrum algorithm parameter values.

**bench\_param** Data structure to hold benchmark algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**val** Array to hold objective function variable vector values. Here, it holds the values of segmentation algorithm parameter vector to be trained.

**score\_p** Pointer to a data structure that holds a score and timing measure.

**Comments**

- This function behaves as an objective function. User can come up with other objective functions to be optimized.

**See Also**

**GenscoreXycut**

**GenscoreVoronoi**

**GenscoreTestDriver**

*GenscoreTestDriver*

```
void GenscoreTestDriver(TRAIN_PROTOCOL *train_protocol, void
*seg_param, void *bench_param, char *weight_file, float *val, SCORE
**score_p)
```

The GenscoreTestDriver function is used to test out optimization algorithm on a possible multivariate function with known minimum/maximum and variable vector values to get this minimum/maximum value.

Defined in: Train.h

**Return Value**

**void.**

**Parameters**

**train\_protocol** Data structure to hold training protocol parameter values.

**seg\_param** Data structure to hold segmentation algorithm parameter values.

**bench\_param** Data structure to hold benchmark algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**val** Array to hold objective function variable vector values.

**score\_p** Pointer to a data structure that holds a score and timing measure.

**Comments**

- This function behaves as an objective function with known characteristics for testing. User can come up with other test objective function to validate the optimization algorithm.

**See Also**

**GenscoreDocstrum**

**GenscoreXycut**

**GenscoreVoronoi**

*GenscoreVoronoi*

```
void GenscoreVoronoi(TRAIN_PROTOCOL *train_protocol, void *voronoi_param,
void *bench_param, char *weight_file, float *val, SCORE **score_p)
```

The GenscoreVoronoi function is used to generate a score from the Voronoi-based segmentation algorithm segmentation result and the corresponding groundtruth using a benchmark algorithm.

Defined in: Train.h

**Return Value**

**void**.

**Parameters**

**train\_protocol** Data structure to hold training protocol parameter values.

**voronoi\_param** Data structure to hold Voronoi algorithm parameter values.

**bench\_param** Data structure to hold benchmark algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**val** Array to hold objective function variable vector values.

**score\_p** Pointer to a data structure that holds a score and timing measure.

**Comments**

- This function behaves as an objective function.
- User can come up with other objective function to be optimized.

**See Also**

**GenscoreDocstrum**

**GenscoreXycut**

**GenscoreTestDriver**

*GenscoreXycut*

```
void GenscoreXycut(TRAIN_PROTOCOL *train_protocol, void *xy-  
cut_param, void *bench_param, char *weight_file, float *val, SCORE **score_p)
```

The GenscoreXycut function is used to generate a score from the X-Y cut segmentation algorithm segmentation result and the corresponding groundtruth using a benchmark algorithm.

Defined in: Train.h

**Return Value**

**void**.

**Parameters**

**train\_protocol** Data structure to hold training protocol parameter values.

**voronoi\_param** Data structure to hold X-Y cut algorithm parameter values.

**bench\_param** Data structure to hold benchmark algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**val** Array to hold objective function variable vector values.

**score\_p** Pointer to a score data structure that stores a score measure and timing measure.

**Comments**

- This function behaves as an objective function.
- User can come up with any other objective function to be optimized.

**See Also**

**GenscoreDocstrum**

**GenscoreVoronoi**

**GenscoreTestDriver**

*ParseArgTest*

```
void ParseArgTest(int argc, char **argv, char **test_report_p, char  
**test_protocol_file_p, char **seg_file_p, char **ben_file_p, char **weight_file_p)
```

The ParseArgTest function is used to parse the command line arguments.

Defined in: TestSeg.h

**Return Value**

**void.**

**Parameters**

**argc** Total number of command line arguments.

**argv** Pointer to command line argument string.

**test\_report\_p** Pointer to a string that stores the testing report file name.

**test\_protocol\_file\_p** Pointer to a string that stores the testing protocol file name.

**seg\_file\_p** Pointer to the segmentation algorithm parameter file name string.

**ben\_file\_p** Pointer to the benchmark algorithm parameter file name string.

**weight\_file** Pointer to the weight file name string.

**Comments**

- 

**See Also**

**ParseArgTrain**

*ParseArgTrain*

```
void ParseArgTrain(int argc, char **argv, char **train_report_p, char
**opt_seg_param_p, char **train_protocol_file_p, char **opt_file_p, char **seg_file_p,
char **ben_file_p, char **weight_file_p)
```

The ParseArgTrain function is used to parse the command line arguments.

Defined in: TrainSeg.h

**Return Value**

void.

**Parameters**

**argc** Total number of command line arguments.

**argv** Pointer to a command line argument string.

**train\_report\_p** Pointer to a pointer that points to a training report file name string.

**opt\_seg\_param\_p** Pointer to a pointer that points to a file name string of a file that saving a set of optimal segmentation algorithm parameters.

**train\_protocol\_file\_p** Pointer to a training protocol file name string.

**opt\_file\_p** Pointer to a optimization algorithm parameter file name string.

**seg\_file\_p** Pointer to a segmentation algorithm parameter file name string.

**ben\_file\_p** Pointer to a benchmark algorithm parameter file name string.

**weight\_file** Pointer to the weight file name string.

**Comments**

- 

**See Also**

ParseArgTest

## *ReadBenchParam*

**void ReadBenchParam(char \*bench\_file, void \*\*ben\_param\_p)**

The ReadBenchParam function is used to read benchmark algorithm parameter values from a file into a data structure.

Defined in: TrainSeg.h

## **Return Value**

void.

## **Parameters**

**bench\_file** Pointer to a benchmark algorithm parameter file name string.

**ben\_param\_p** Pointer to a data structure holding benchmark algorithm parameter values.

## **Comments**

- For a new algorithm, user can write its own parameter reading function like this one.

## **See Also**

**ReadDocstrumParam**

**ReadFileNames**

**ReadSimplexParam**

**ReadTrainProtocol**

**ReadTestProtocol**

**ReadVoronoiParam**

**ReadXycutParam**

*ReadDocstrumParam*

**void ReadDocstrumParam(char \*docstrum\_file, void \*\*docstrum\_param\_p)**

The ReadDocstrumParam function is used to read Docstrum segmentation algorithm parameter values from a file into a data structure.

Defined in: TrainSeg.h

**Return Value**

void.

**Parameters**

**docstrum\_file** Pointer to a string that stores the Docstrum segmentation algorithm parameter file name.

**docstrum\_param\_p** Pointer to a data structure holding Docstrum algorithm parameter values.

**Comments**

- For a new algorithm, user can write its own parameter reading function like this one.

**See Also**

**ReadBenchParam**

**ReadFileNames**

**ReadSimplexParam**

**ReadTrainProtocol**

**ReadTestProtocol**

**ReadVoronoiParam**

**ReadXycutParam**

## *ReadFileNames*

```
void ReadFileNames(char *datafile, DATASET **dataset_p)
```

The ReadFileNames function is used to read file names from a data file into a dataset data structure.

Defined in: TrainSeg.h

## **Return Value**

void.

## **Parameters**

**datafile** Pointer to a dataset-file name string.

**dataset\_p** Pointer to a data structure holding dataset image file names.

## **Comments**

- For a new algorithm, user can write its own parameter reading function like this one.

## **See Also**

**ReadBenchParam**

**ReadDocstrumParam**

**ReadSimplexParam**

**ReadTrainProtocol**

**ReadTestProtocol**

**ReadVoronoiParam**

**ReadXycutParam**

*ReadSimplexParam*

```
void ReadSimplexParam(char *simplex_file, void **simplex_param_p)
```

The ReadSimplexParam function is used to read Simplex optimization algorithm parameter values from a file into a data structure.

Defined in: TrainSeg.h

**Return Value**

void.

**Parameters**

**simplex\_file** Pointer to a Simplex optimization algorithm parameter file name string.

**simplex\_param\_p** Pointer to a data structure holding Simplex algorithm parameter values.

**Comments**

- For a new algorithm, user can write its own parameter reading function like this one.

**See Also**

**ReadBenchParam**

**ReadDocstrumParam**

**ReadFileNames**

**ReadTrainProtocol**

**ReadTestProtocol**

**ReadVoronoiParam**

**ReadXycutParam**

## *ReadTestProtocol*

```
void ReadTestProtocol(char *test_protocol_file, TEST_PROTOCOL  
**test_protocol_p)
```

The ReadTestProtocol function is used to read testing protocol parameter values from a file into a data structure.

Defined in: TestSeg.h

## **Return Value**

**void.**

## **Parameters**

**test\_protocol\_file** Pointer to a testing protocol parameter file name string.

**test\_protocol\_p** Pointer to a data structure holding testing protocol parameter values.

## **Comments**

- For a new algorithm, user can write its own parameter reading function like this one.

## **See Also**

**ReadBenchParam**

**ReadDocstrumParam**

**ReadFileNames**

**ReadSimplexParam**

**ReadTrainProtocol**

**ReadVoronoiParam**

**ReadXycutParam**

*ReadTrainProtocol*

```
void ReadTrainProtocol(char *train_protocol_file, TRAIN_PROTOCOL  
**train_protocol_p)
```

The ReadTrainProtocol function is used to read training protocol parameter values from a file into a data structure.

Defined in: TrainSeg.h

**Return Value**

void.

**Parameters**

**train\_protocol\_file** Pointer to a training protocol parameter file name string.

**train\_protocol\_p** Pointer to a data structure holding training protocol parameter values.

**Comments**

- For a new algorithm, user can write its own parameter reading function like this one.

**See Also**

**ReadBenchParam**

**ReadDocstrumParam**

**ReadFileNames**

**ReadSimplexParam**

**ReadTestProtocol**

**ReadVoronoiParam**

**ReadXycutParam**

## *ReadVoronoiParam*

**void ReadVoronoiParam(char \*voronoi\_file, void \*\*voronoi\_param\_p)**

The ReadVoronoiParam function is used to read Voronoi-based segmentation algorithm parameter values from a file into a data structure.

Defined in: TrainSeg.h

## **Return Value**

void.

## **Parameters**

**voronoi\_file** Pointer to a Voronoi-based algorithm parameter file name string.

**voronoi\_param\_p** Pointer to a data structure holding Voronoi-based algorithm parameter values.

## **Comments**

- For a new algorithm, user can write its own parameter reading function like this one.

## **See Also**

**ReadBenchParam**

**ReadDocstrumParam**

**ReadFileNames**

**ReadSimplexParam**

**ReadTrainProtocol**

**ReadXycutParam**

*ReadXycutParam*

```
void ReadXycutParam(char *xycut_file, void **xycut_param_p)
```

The ReadXycutParam function is used to read X-Y cut segmentation algorithm parameter values from a file into a data structure.

Defined in: TrainSeg.h

**Return Value**

void.

**Parameters**

**xycut\_file** Pointer to the X-Y cut algorithm parameter file name string.

**xycut\_param\_p** Pointer to a data structure holding the X-Y cut algorithm parameter values.

**Comments**

- For a new algorithm, user can write its own parameter reading function like this one.

**See Also**

**ReadBenchParam**

**ReadDocstrumParam**

**ReadFileNames**

**ReadSimplexParam**

**ReadTrainProtocol**

**ReadVoronoiParam**

*ReadWeight*

```
void ReadWeight(char *weight_file, WEIGHT **weight_p)
```

The ReadWeight function is used to read a set of user specified weight values from a file into a data structure.

Defined in: Bench.h

**Return Value**

**void**.

**Parameters**

**weight\_file** Pointer to the weight file name string.

**weight\_p** Pointer to a data structure holding a set of weight values.

**Comments**

- 

*Simplex*

```
int Simplex(TRAIN_PROTOCOL *train_protocol, SIMPLEX_PARAM
*opt_param, void *ben_param, void *seg_param, char *weight_file, void
(*genscore)(), char **train_report_p, char **opt_seg_param_p)
```

The Simplex function is an implementation of Nelder-Mead Simplex optimization ?? algorithm.

Defined in: Train.h

**Return Value**

**int**.

**Parameters**

**train\_protocol** Pointer to a data structure holding the training protocol parameter values.

**opt\_param** Pointer to a data structure holding the Simplex optimization algorithm parameter values.

**ben\_param** Pointer to a data structure holding the benchmark algorithm parameter values.

**seg\_param** Pointer to a data structure holding the segmentation algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**(\*genscore)()** Pointer to a objective function.

**train\_report\_p** Pointer to a pointer that points to a training report file name string.

**opt\_seg\_param\_p** Pointer to a pointer that points to a file name of a file storing optimal segmentation algorithm parameters.

## Comments

- This optimization algorithm is a local optimization algorithm, it deals with the objective functions that have no explicit mathematical forms (or black-box objective function).

## See Also

### Generic

### SimulatedAnnealing

*Test*

```
void Test(TEST_PROTOCOL *test_protocol, void *ben_param, void
*seg_param, char *weight_file, char **test_report_p)
```

The Test function is used to evaluate a given page segmentation algorithm using a given benchmarking algorithm in the testing phase.

Defined in: TrainSeg.h

**Return Value**

**void.**

**Parameters**

**test\_protocol** Pointer to a data structure holding a testing protocol parameter values.

**ben\_param** Pointer to a data structure holding a benchmark algorithm parameter values.

**seg\_param** Pointer to a data structure holding a segmentation algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**test\_report\_p** Pointer to a testing report file name string.

**Comments**

- 

**See Also**

**Train**

*Train*

```
void Train(TRAIN_PROTOCOL *train_protocol, void *opt_param, void
*ben_param, void *seg_param, char *weight_file, char **train_report_p, char
**opt_seg_param_p)
```

The `Train` function is used to train or optimize a given page segmentation algorithm using a given optimization algorithm and benchmark algorithm in the training phase.

Defined in: `TrainSeg.h`

**Return Value**

`void`.

**Parameters**

**train\_protocol** Pointer to a data structure holding a training protocol parameter values.

**opt\_param** Pointer to a data structure holding a Simplex optimization algorithm parameter values.

**ben\_param** Pointer to a data structure holding a benchmark algorithm parameter values.

**seg\_param** Pointer to a data structure holding a segmentation algorithm parameter values.

**weight\_file** Pointer to a string storing weight file name.

**train\_report\_p** Pointer to a training report file name string.

**opt\_seg\_param\_p** Pointer to a pointer that points to a file name of a file storing optimal segmentation algorithm parameters.

**Comments**

-

**See Also**

**Test**

*Usage***void Usage()**

This function is required by a shell system call.

Defined in: TrainSeg.h

**Return Value**

**void.**

**Parameters****Comments**

- 

**See Also****Test**

*VoronoiFunction*

```
int VoronoiFunction(char *tif_filename, VORONOI_PARAM *para,
char *dafs_filename)
```

The VoronoiFunction function is an implementation of the Voronoi-based segmentation algorithm developed by Kise *et. al.*

Defined in: GenscoreVoronoi.h

**Return Value**

**int.**

**Parameters**

**tif\_filename** Image file name.

**para** Data structure holding the Voronoi-based segmentation algorithm parameter values.

**dafs\_file** Segmentation result file name.

**Comments**

- This algorithm is a bottom-up algorithm ??.
- It only deals with text blocks.
- This code is obtained directory from the author (Dr. Kise) which have some memory related bugs, continuous run of this function can cause segmentation fault. Hence a shell call mode is recommended.

**See Also**

**DocstrumFunction**

**XycutFunction**

*WriteTestReportHeader*

```
void WriteTestReportHeader(int argc, char **argv, TEST_PROTOCOL
*test_protocol, char **file_p)
```

The WriteTestReportHeader function is used to write testing report header information, e.g. user, purpose, date, working directory, machine name, operating system version and command line, etc.

Defined in: TestSeg.h

**Return Value**

void.

**Parameters**

**argc** Total number of command line arguments.

**argv** Pointer to a command line argument string.

**test\_protocol** Data structure holding testing protocol parameter values.

**file\_p** Testing report file name.

**Comments**

- This automatically printout experimental environment parameters which can be conveniently referred in the future.

**See Also**

**WriteTrainReportHeader**

*WriteTrainReportHeader*

```
void WriteTrainReportHeader(int argc, char **argv, TRAIN_PROTOCOL
*train_protocol, char **file_p)
```

The WriteTrainReportHeader function is used to write training report header information, e.g. user, purpose, date, working directory, machine name, operating system version and command line, etc.

Defined in: TrainSeg.h

**Return Value**

void.

**Parameters**

**argc** Total number of command line arguments.

**argv** Pointer to a command line argument string.

**train\_protocol** Data structure holding a training protocol parameter values.

**file\_p** Training report file name.

**Comments**

- This automatically printout experimental environment parameters which can be conveniently referred in the future.

**See Also**

**WriteTestReportHeader**

*XycutFunction*

```
int XycutFunction(char *tif_filename, XYCUT_PARAM *para, char
*dafs_filename)
```

The XycutFunction function is an implementation of the X-Y cut segmentation algorithm developed by Nagy.

Defined in: GenscoreXycut.h

**Return Value**

**int**.

**Parameters**

**tif\_filename** Image file name.

**para** Data structure holding the X-Y cut algorithm parameter values.

**dafs\_filename** Segmentation result file name.

**Comments**

- This algorithm is a top-down algorithm ??.

**See Also**

**DocstrumFunction**

**VoronoiFunction**

### *Voronoi-related functions*

**bit\_func**, **brect**, **cline**, **dinfo**, **edgelist**, **erase**, **geometry**, **hash**, **heap**, **img\_to\_site**, **label\_func**, **memory**, **output**, **read\_image**, **sites**, **usage**, **voronoi**

The functions are used for the implementation of the Voronoi-based segmentation algorithm developed by Kise *et.al.*

Defined in: VoronoiFunction.h

### **Return Value**

### **Parameters**

### **Comments**

- This functions are developed by a different programmer, the detailed information is not available.

### **See Also**

*nrutil functions*

`nrerror`, `vector`, `ivector`, `dvector`, `matrix`, `dmatrix`, `imatrix`, `cmatrix`, `submatrix`, `free_vector`, `free_ivector`, `free_dvector`, `free_matrix`, `free_dmatrix`, `free_imatrix`, `free_cmatrix`, `free_submatrix`, `convert_matrix`, `free_convert_matrix`

The functions are Numerical Recipe standard functions for array and matrix manipulations.

Defined in: `nrutil.h`

**Return Value****Parameters****Comments**

- 

**See Also**

*sort*

**sort, sort2**

The functions are Numerical Recipe Heapsort functions.

Defined in: DocstrumFunction.h

## **Return Value**

## **Parameters**

## **Comments**

- 

## **See Also**

*util***err, Alloc, fOpen, sh\_c**

The functions are utility functions that can be used to call a function by their executable.

Defined in: util.h

**Return Value****Parameters****Comments**

- 

**See Also**

# Bibliography

- [1] *DARPA Broadcast News Workshop*, Herndon, VA, February 1999.  
<http://www.itl.nist.gov/iaui/894.01/publications/darpa99/index.htm>.
- [2] A. D. Bagdanov. The fourth annual test of OCR accuracy. In A. D. Bagdanov, editor, *Annual Report*. Information Science Research Institute, University of Nevada, Las Vegas, NV, 1995.
- [3] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The New S Language*. Wadsworth & Brooks/Cole, Pacific Grove, CA, 1988.
- [4] Caere Co. *Caere Developer's Kit 2000*. <http://www.caere.com/>.
- [5] D. Dori, I. Phillips, and R. M. Haralick. Incorporating documentation and inspection into computer integrated manufacturing: An object-process approach. In S. Adiga, editor, *Applications of Object-Oriented Technology in Manufacturing*. Chapman & Hall, London, UK, 1994.
- [6] J. J. Foster. *Data Analysis Using SPSS for Windows — A Beginner's Guide*. SAGE Publications, London, UK, 1998.
- [7] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [8] R. M. Haralick and L. G. Shapiro. *Computer and Robot Vision*. Addison-Wesley, Reading, MA, 1992.
- [9] K. Kise, A. Sato, and M. Iwata. Segmentation of page images using the area Voronoi diagram. *Computer Vision and Image Understanding*, 70:370–382, 1998.

- [10] S. Mao and T. Kanungo. A methodology for empirical performance evaluation of page segmentation algorithms. Technical Report CAR-TR-933, University of Maryland, College Park, MD, December 1999. <http://www.cfar.umd.edu/kanungo/pubs/trsegeval.ps>.
- [11] S. Mao and T. Kanungo. Automatic training of page segmentation algorithms: An optimization approach. In *Proceedings of International Conference on Pattern Recognition*, pages 531–534, Barcelona, Spain, September 2000.
- [12] S. Mao and T. Kanungo. Empirical performance evaluation of page segmentation algorithms. In *Proceedings of SPIE Conference on Document Recognition and Retrieval*, pages 303–314, San Jose, CA, January 2000.
- [13] G. Nagy, S. Seth, and M. Viswanathan. A prototype document image analysis system for technical journals. *Computer*, 25:10–22, 1992.
- [14] J. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.
- [15] L. O’Gorman. The document spectrum for page layout analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:1162–1173, 1993.
- [16] T. Pavlidis and J. Zhou. Page segmentation and classification. *Graphical Models and Image Processing*, 54:484–496, 1992.
- [17] ScanSoft Co. *TextBridge: Application Programmer’s Interface*. <http://www.scansoft.com>.
- [18] E. M. Voorhees and D. K. Harman, editors. *The Seventh Text REtrieval Conference (TREC 7)*. National Institute of Standards and Technology, 1998. <http://trec.nist.gov/pubs.html>.

[

Index]Index

# Index

Amotry, 48  
BENCH\_PARAM Structure, 29  
BenchScoring, 51  
Bench, 50  
CalendarTime, 52  
DATASET Structure, 28  
DOCSTRUM\_PARAM Structure, 36  
DocstrumFunction, 53  
ERR\_MEASURE Structure, 34  
EST\_PARA Structure, 38  
EvalDocstrum, 54  
EvalVoronoi, 55  
EvalXycut, 56  
FreeJobTest, 57  
FreeJobTrain, 58  
GenscoreDocstrum, 59  
GenscoreTestDriver, 60  
GenscoreVoronoi, 61  
GenscoreXycut, 62  
ParseArgTest, 63  
ParseArgTrain, 64  
ReadBenchParam, 65  
ReadDocstrumParam, 66  
ReadFileNames, 67  
ReadSimplexParam, 68  
ReadTestProtocol, 69  
ReadTrainProtocol, 70  
ReadVoronoiParam, 71  
ReadXycutParam, 72  
SIMPLEX\_PARAM Structure, 39  
Simplex, 73  
TEST\_PROTOCOL Structure, 44  
TEXTLINE Structure, 31  
TRAIN\_PROTOCOL Structure, 26  
Test, 75  
Train, 76  
Usage, 77  
VORONOI\_PARAM Structure, 41  
Voronoi-related functions, 82  
VoronoiFunction, 78  
WriteTestReportHeader, 79  
WriteTrainReportHeader, 80  
XYCUT\_PARAM Structure, 43  
XycutFunction, 81  
ZONE Structure, 33  
nrutil functions, 83  
sort, 84  
util, 85