

Dylan Doxey
Math 107B Spring 2002

Optical Character Recognition

Introduction

This worksheet is a cursory study of optical character recognition (OCR). A typical application of OCR would be using a computer to translate a paper text document into an equivalent electronic text document. The process involves scanning the paper document to produce an image of the document. An OCR algorithm is then applied to that image to produce an electronic text document.

This worksheet explores the issues involved with taking an image of a single text character and identifying which character the image represents. This approach ignores the issues involved with distinguishing individual characters in the image and skips right to the business of analyzing a single isolated character. Provided with this worksheet is a set sample data images representing the upper case letters "A" through "Z" in Arial font, such as the one below.



The basic approach taken here is modeled after the "feature vector" concept outlined by Richard O. Duda of the Department of Electrical Engineering at San Jose State University. Duda's website at http://www-engr.sjsu.edu/~knapp/HCIRODPR/PR_home.htm provides some background on computerized pattern recognition.

The feature vector model proposes that a given image has certain features that make it distinguishable as representing some distinct item. In this case we are interested in identifying features in an image that make it distinguishable as a certain letter in the English alphabet. Identifying these key features is utterly trivial to a human being. But to a computer identifying such features is complex.

Procedure Definitions

```
> restart:
> with(plots):
with(linalg):
with(ListTools):
with(StringTools):
with(LinearAlgebra):
```

Warning, the name changecoords has been redefined

Warning, the protected names norm and trace have been redefined and unprotected

Warning, the assigned name Group now has a global binding

Warning, these names have been rebound: Group, Join, Reverse, Split

Warning, the names DotProduct, Map and Transpose have been rebound

Warning, the assigned name GramSchmidt now has a global binding

Global constant declarations

>

> **midintensity := 128;**

> **alphabet :=**

[A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z];

> **alpha := [A,E,H,F];**

> **barwidth := 5;**

> **barlength := 30;**

The file path must reflect the location of the sample character image files.

> **file_path := "C:\\characters\\";**

midintensity := 128

alphabet := [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]

$\alpha := [A, E, H, F]$

barwidth := 5

file_path := "C:\\characters\\"

barlength := 30

>

File Input/Output

File Input/Output

This section contains the procedures used to read portable graymap (pgm) image files and translate them into matrix objects that can be manipulated in Maple.

digitize converts image pixel intensities to their digital equivalence. 1 is returned for values above the midpoint of the intensity range of the last image read and 0 otherwise.

The 'midintensity' global variable is set in the file reading procedure `img2matrix`.

> **digitize := proc(val::string)**

```

    if string2int(val) > 128 then return 0:
    else return 1: fi:
end proc:

```

string2int accepts a string that is assumed to be one or more numeric characters, which together represent an integer. The return value is the integer value of the string passed in.

```

> string2int := proc(instring::string)
    local number, asciiList, n, m ,i:
    number := 0;
    asciiList := convert(instring,'bytes'):
    n := vectdim(asciiList):
    for i from n by -1 to 1 do
        m := n - i:
        number := number + (asciiList[i]-48)*10^m:
    end do:
    return number:
end proc:

```

invert assumes the input is either 1 or 0 and returns 0 for 1 and 1 for 0.

```

> invert := proc(value)
    if value = 1 then return 0:
    else return 1:
    end if:
end proc:

```

imgConvert accepts a 1 by n array of strings, a column count int and a row count int.

The strings in the array are assumed to represent int values. The return is a cols by rows matrix populated with the int values of the string elements of the array passed in.

```

> imgConvert := proc(cols::integer,rows::integer,pixels)
    local size, pix, f:
    size := cols * rows:
    pix := [seq(digitize(pixels[i]),i=1..size)]:
    f := (i,j) -> pix[rows*(i-1) + ((j-1) mod cols)+1]:
    return Matrix(cols,rows,f):
end proc:

```

img2matrix accepts a string representing the full name of a "pgm" type image file. The file is opened and read. The return value is a matrix representing the image. The int elements of the matrix represent the grayscale intensities of each pixel in the source image.

```

> img2matrix := proc(inputFile::string)
    local
fileHandle,comment,mNum,dimensions,cols,rows,shades,pixels:
    fileHandle := fopen(inputFile,READ,BINARY):
    mNum := readline(fileHandle):
    comment := readline(fileHandle):
    dimensions := readline(fileHandle):
    dimensions := StringTools[Split](dimensions, " "):
    cols := string2int(dimensions[2]):
    rows := string2int(dimensions[1]):
    shades := readline(fileHandle):
    pixels := readbytes(fileHandle,infinity,TEXT):
    pixels := StringTools[Split](Trim(SubstituteAll(pixels,
"\n","")), " "):
    fclose(fileHandle):
    return imgConvert(cols,rows,pixels):
end proc:

```

getLetter accepts a letter as a string and returns a matrix the entries of which represent the pixels of the source image.

```

> getLetter := proc(letter)
    local file_name:
    file_name := cat(file_path, "ari", letter, ".pgm"):
    return evalm(img2matrix(file_name)):
end proc:
>

```

Point Grouping Rules

This returns true the X coordinate of the first point is larger than the X coordinate of the second.

If the X coordinates are equal then it returns true if the first of the Y coordinates is larger.

```

> largerX := proc(a, b)
    if (a[1] - b[1]) > 0 then
        return true:
    else
        if a[1] = b[1] then
            if a[2] = b[2] then
                return false:
            else
                return largerY(a, b):
            fi:
        else
            return false:
        end if:
    end if:
end proc:

```

```

        fi:
    fi:
end proc:

```

This returns true the y coordinate of the first point is larger than the y coordinate of the second.

If the Y coordinates are equal then it returns true if the first of the X coordinates is larger.

```

> largerY := proc(a, b)
    if (a[2] - b[2]) > 0 then
        return true:
    else
        if a[2] = b[2] then
            if a[1] = b[1] then
                return false:
            else
                return largerX(a, b):
            fi:
        else
            return false:
        fi:
    fi:
end proc:

```

This returns true if the Y coordinates are within 'barwidth' distance of each other

```

> YdiffY := proc(a, b)
    if abs(a[1] - b[1]) < barwidth then return true:
    else return false: fi:
end proc:

```

This returns true if the X coordinates are within 'barwidth' distance of each other

```

> XdiffX := proc(a, b)
    if abs(a[2] - b[2]) < barwidth then return true:
    else return false: fi:
end proc:

```

This returns true if the X coordinates are equal to each other

```

> XequalX := proc(a, b)
    if a[1] = b[1] then return true:
    else return false: fi:
end proc:

```

This returns true if the X coordinates are equal to each other

```

> YequalY := proc(a, b)
    if a[2] = b[2] then return true:
    else return false: fi:
end proc:

```

This returns true if the two points are adjacent vertically, horizontally or diagonally

```

> adjPoints := proc(a, b)
    if abs(a[1] - b[1]) < 2 and abs(a[2] - b[2]) < 2 then

```

```

return true: fi:
  return false:
end proc:

```

This returns true if the first list of points is larger than the second

```

> columnCompare := proc(a, b)
  local span_a, span_b, mt_a, mt_b:
  span_a := ranges2dims(ranges(a)):
  span_a := span_a[1]*span_a[2]:
  span_b := ranges2dims(ranges(b)):
  span_b := span_b[1]*span_b[2]:
  mt_a := span_a - vectdim(a):
  mt_b := span_b - vectdim(b):
  if abs(mt_a - mt_b) < 4 and span_a > 30 and span_b > 30
then return true:
  else return false: fi:
end proc:

```

This compares two lists and returns true if the first is larger.

```

> bySize := proc(a, b)
  if nops(a) > nops(b) then return true:
  else return false: fi:
end proc:
>

```

Point Group Handling Procedures

tableLookup determines which column of a given matrix the given vector matches.

```

> tableLookup := proc(A::Matrix, x::vector)
  local i,C,s,min,index:
  if vectdim(x) <> rowdim(A)-1
  then error("Dimension mismatch") end if:
  s := A[2..rowdim(A),2..coldim(A)]:
  index := A[1,2]:
  min := norm(col(s, 1) - x):
  for i from 2 by 1 to coldim(s) do
    C := col(s, i):
    if evalb(evalf(norm(C - x)) < evalf(min))
    then
      min := norm(C - x):
      index := A[1,i+1]:
    end if:
  end do:
  return index:
end proc:

```

This determines the range of they X and Y coordinates spaned by a set of points

```

> ranges := proc(points)
  local s,d, x1,x2, y1,y2:
  d := vectdim(points):
  s := sort(points, largerX);
  x1 := round(s[d][1]):
  x2 := round(s[1][1]):
  s := sort(points, largerY):
  y1 := round(s[d][2]):
  y2 := round(s[1][2]):
  return [x1..x2,y1..y2]:
end proc:

```

ranges2dims acceptst a two item list of range type items and returns the dimensions that are spanned by those ranges.

```

> ranges2dims := proc(rngs::list)
  local v_span, h_span:
  h_span := 1+abs(($rngs[1])[vectdim([$rngs[1]])] -
($rngs[1])[1]):
  v_span := 1+abs(($rngs[2])[vectdim([$rngs[2]])] -
($rngs[2])[1]):
  return [h_span, v_span]:
end proc:

```

graphicGroups takes a list of point lists and returns a corresponding list of graphic objects which can be displayed individually.

```

> graphicGroups := proc(point_groups::list)
  local i,graphic_objects,m,go:
  graphic_objects := vector(nops(point_groups)):
  for i from 1 by 1 to vectdim(point_groups) do
    m := categoryMatrix(point_groups[i]):
    go :=
sparsematrixplot(m,'matrixview','scaling=CONSTRAINED'):
    graphic_objects[i] := go:
  od;
  return convert(graphic_objects,list):
end proc:

```

categoryMatrix accepts a list of points and returns a matrix with 1 entries corresponding to the points.

```

> categoryMatrix := proc(category_list::list)
  local rngs,dims,x_offset,y_offset,cat_mat,i,x,y:
  rngs := ranges(category_list);
  dims := ranges2dims(rngs);
  x_offset := 1-($rngs[1])[1];
  y_offset := 1-($rngs[2])[1];
  cat_mat := matrix(dims[1],dims[2],0):

```

```

    for i from 1 by 1 to vectdim(category_list) do
        x := round(category_list[i][1]) + x_offset;
        y := round(category_list[i][2]) + y_offset;
        cat_mat[x,y] := 1;
    od;
    return cat_mat;
end proc:

```

partition partitions a list of points.

```

> partition := proc(points::list, orientation::string)
    local i, e, categorized;
    if orientation <> "vertical" and orientation <>
"horizontal" then
        error("orientation must be either vertical or
horizontal"): fi;
    if orientation = "horizontal" then
        # Get rows
        categorized :=
[ListTools[Categorize](YequalY,points)]:
    else
        # Get columns
        categorized :=
[ListTools[Categorize](XequalX,points)]:
    fi;
    # Recategorize by col/row sets that are either col/row
or not
    categorized :=
[ListTools[Categorize](columnCompare,categorized)]:
    e := nops(categorized);
    for i from 1 by 1 to e do
        # Convert the lists of col/row set to lists of points
        categorized[i] := FlattenOnce(categorized[i]):
    od;
    if e > 3 then e := 3 fi;
    return sort(categorized,bySize)[1..e]:
end proc:

```

generatePartitions takes a list of characters and it will generate a list of feature partitions from the reference set of character images.

```

> generatePartitions := proc(alphabet)
    local partitions, letter, the_plot, i,j, points:
    partitions := matrix(3,nops(alphabet)):
    for i to nops(alphabet) do
        letter :=
getLetter(convert(alphabet[i],string),which):

```



```

        the_plot :=
sparsematrixplot(transpose(letter),'matrixview'):
        points :=
[op(1..nops(op(1,the_plot)),op(1,the_plot))]:
        partitions[1,i] := alphabet[i]:
        partitions[2,i] := partition(points, "vertical"):
        partitions[3,i] := partition(points, "horizontal"):
        printf("Vertical and horizontal partitions: %s.\n",
convert(alphabet[i],string)):
    end do:
    return partitions:
end proc:

```

countBars accepts a list of points and determines the number of components in the given orientation.

```

> countBars := proc(points::list,orientation::string)
    local
x,last_x,a,i,scanning,sorted_points,barcount,target,area:
    if orientation <> "vertical" and orientation <>
"horizontal" then
        error("orientation must be either vertical or
horizontal"): fi:
    if orientation = "vertical" then
        sorted_points := sort(points,largerX):
        a := 1:
    else
        sorted_points := sort(points,largerY):
        a := 2:
    fi:
    barcount := 0:
    target := sorted_points[1][a] - barwidth:
    area := 0:
    scanning := true:
    for i to nops(sorted_points) do
        last_x := x:
        x := sorted_points[i][a]:
        if scanning then
            area := area + 1:
            if x < target then
                if area >= barwidth*barlength then
                    barcount := barcount + 1:
                    scanning := false:
                fi:
                target := sorted_points[i][a] - barwidth:
                area := 0:
            fi:
        fi:
    end for:

```

```

        else
            if abs(last_x - x) > 1 then
                area := area + 1:
                scanning := true:
            fi:
        od:
    return barcount:
end proc:

```

doBarCount accepts a list of point groups which are each analyzed for horizontal and vertical components. The largest count found in any of the groups is returned.

```

> doBarCount := proc(point_groups::list,orientation::string)
    local i,c,s:
    c := vector(nops(point_groups)):
    for i to nops(point_groups) do
        c[i] := countBars(point_groups[i],orientation):
    od:
    return ListTools[Reverse](sort(convert(c,list)))[1]:
end proc:
>

```

Sample Analysis

Let's first consider that a given alphabetic character has certain significant horizontal and vertical components that are unique relative to the other characters in the alphabet. For example the letter "E" has a single vertical component and three horizontal components. Again, making this observation is trivial to the human observer. But it is not so obvious to the computer.

E

Immediately we are confronted with the issue of data representation. We must decide on what form to represent the scanned image. This worksheet has procedures defined to read the provided image files and represent them as Maple Matrix data structures. In this form the image data can be manipulated with matrix handling procedures defined in the "linalg", "Linear Algebra" and other packages as well.

```

> sample := getLetter("sampleE");
letter :=
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
[0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
%1 := [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]

%2 := [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

%3 := [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]

```

>

This example shows the letter "E" read from a "portable graymap" type image and then translated onto a corresponding matrix. Although the image is now in the form of a manageable data structure, it is still not obvious to the computer what this matrix represents.

More on the provided sample input files

The provided input files are in portable graymap format and have the ".pgm" extension. It is convenient for us in the Maple programming environment because the image data is encoded as

integer grayscale values in ascii format. The grayscale values are read as strings and converted to either 0 or 1 depending on whether or not the grayscale is above or below the assumed midpoint threshold of 128.

The portable graymap format is used in the UNIX programming environment for manipulating images in various stages of enhancement. More information is available at:

http://seawifs.gsfc.nasa.gov/~norman/seawifs_image_cookbook/faux_shuttle/pgm.html

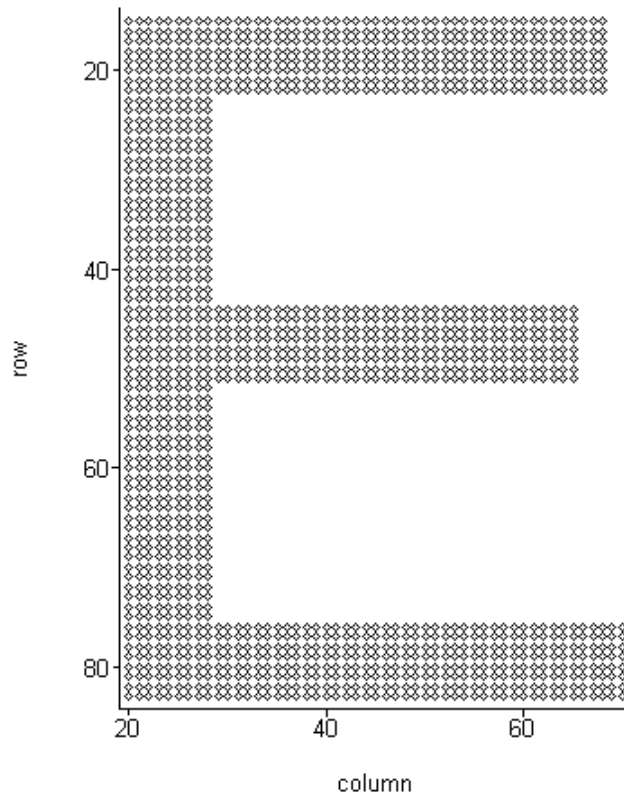
http://seawifs.gsfc.nasa.gov/~norman/seawifs_image_cookbook/faux_shuttle/

In accordance with Duda's feature vector concept, now is the time to use a feature extractor procedure to start calculating the distinguishing features of this character. The most obvious traits of the letter "E" are its horizontal and vertical components. So let's start there.

To identify the vertical component, the computer could take the magnitude of all the column vectors and pick out the ones that are more than a particular threshold. Then the computer is going to need to take measures to notice that several adjacent column vectors above the set threshold represent only a single vertical component. Otherwise the above matrix might register three vertical components.

Using this method it seemed that the computer was taking such a long time to complete the analysis. So let's consider an alternate form of representation. Using the Maple "sparsematrix" plot procedure, the matrix can be plotted as a group of points. So, let's bring up another character image.

```
> letter := getLetter("E");  
> lplot :=  
sparsematrixplot(transpose(letter), 'matrixview', 'scaling=CONSTRAINED');  
> display(lplot);
```



>

This plot is a data structure that can be dissected. The points plotted here can be pulled out as a list and handled with list procedures provided in the Maple "ListTools" package. The greatest value in this is that the points can now be sorted and categorized with the list "sort" and ListTools "Categorize" procedures respectively.

```
> lpoints := [op(1..nops(op(1,lplot)),op(1,lplot))]:
```

>

By selectively sorting and categorizing the points we can isolate key features of the character. For example, let's pull out the three major groups where the points are categorized by vertically common traits.

```
> cat_vertical := partition(lpoints, "vertical");
```

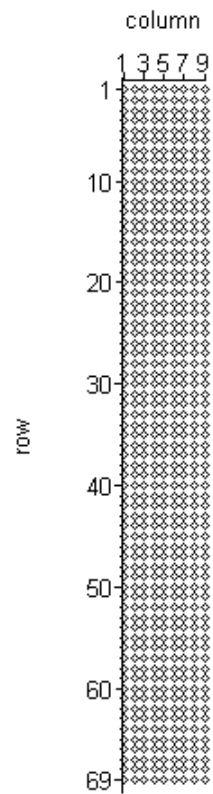
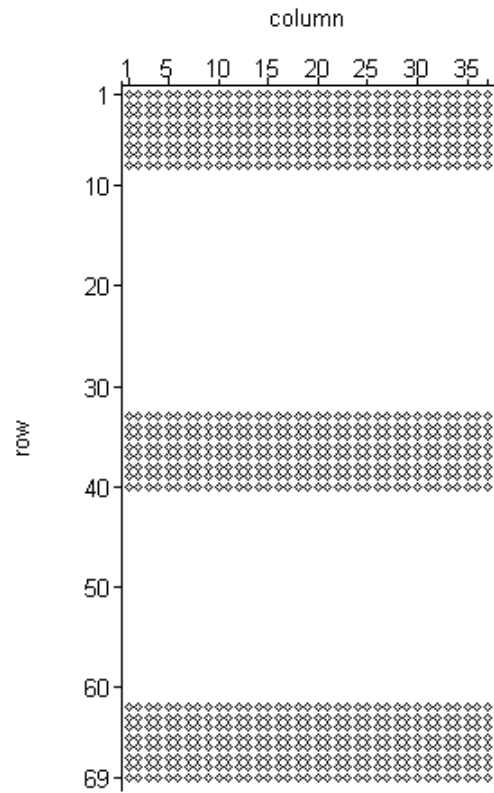
>

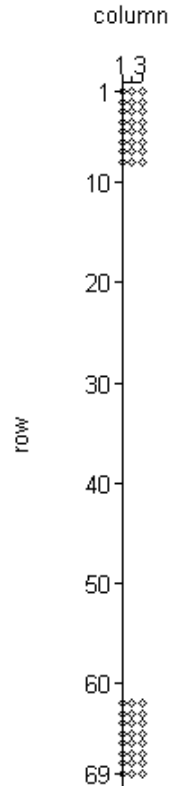
For the sake of this study, this worksheet includes a procedure for translating a list of points back into a matrix data structure that can be plotted. Doing this gives us the ability to observe what points were selected as having vertically common traits.

```
> vertical_bands := graphicGroups(cat_vertical):
```

```
> for i to nops(vertical_bands) do
```

```
    display(vertical_bands[i]);  
od;
```





>

These vertical category groupings present us with the points that compose any significant vertical components of the character. Knowing this, the computer can now use list handling procedures to analyze the points and count the number of vertical components present.

```
> v_count := doBarCount(cat_vertical, "vertical");
      v_count := 1
```

>

Now these same procedures can be applied to determine the number of horizontal components also.

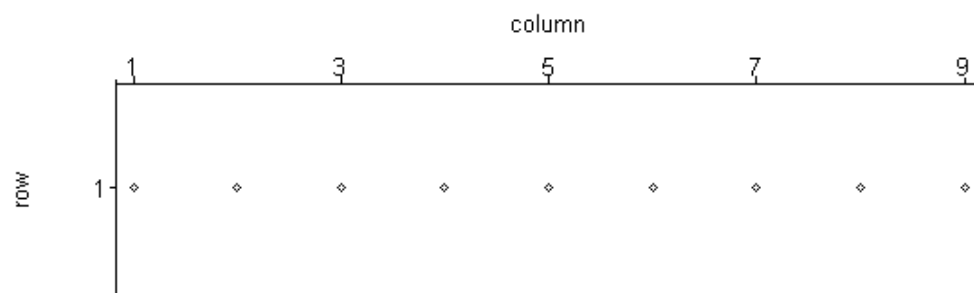
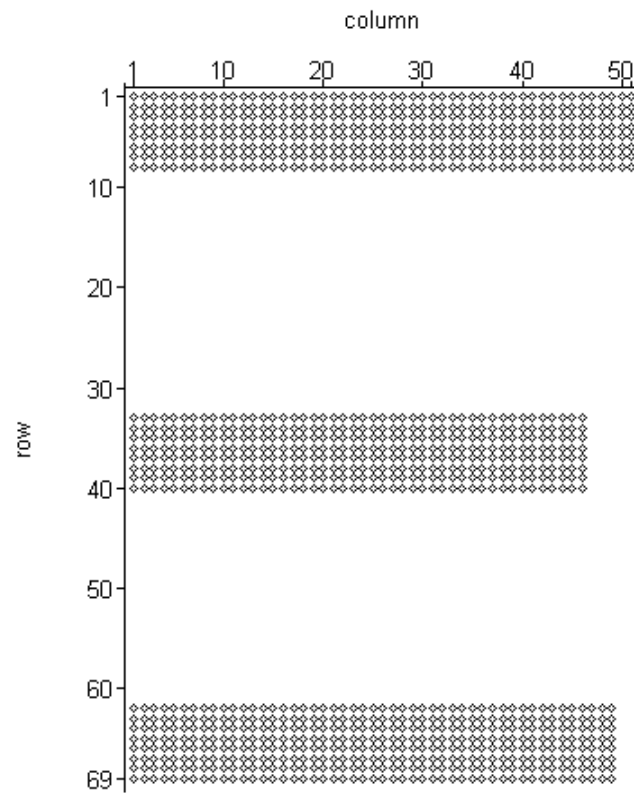
```
> cat_horizontal := partition(lpoints, "horizontal");
> h_count := doBarCount(cat_horizontal, "horizontal");
      h_count := 3
```

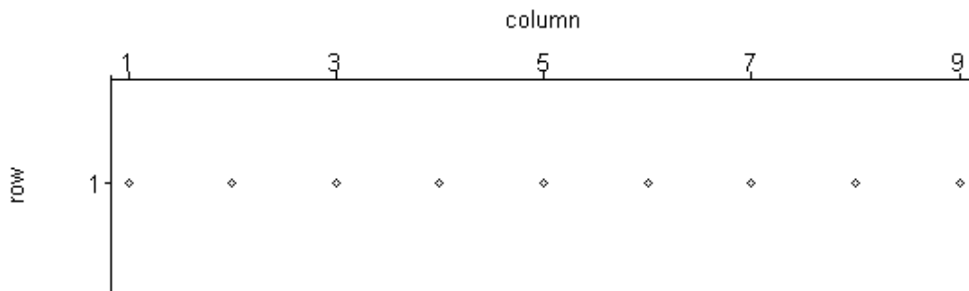
>

Again for the human observer, here is a graphic representation of the points determined to have horizontally common traits.

```
> horizontal_bands := graphicGroups(cat_horizontal):
```

```
> for i from 1 by 1 to nops(horizontal_bands) do
  display(horizontal_bands[i]);
od;
```





>

This completes the feature extraction for the given sample input. Note, as human observers it is obvious that we are examining the letter "E", but for the computer the input is still unidentified.

Feature Space

In accordance with the feature vector procedure outlined by Duda, let's create the feature vector for this character. This feature vector will be two dimensional vector which contains the number of horizontal and vertical features.

```
> features := vector([v_count, h_count]);
               features := [1, 3]
```

>

By calculating a feature vector for all of the characters in our alphabet and augmenting them to a matrix the computer can construct a feature space matrix. The dimensions will be the number of features by the number of characters. For this study our feature space will be a 2 by 26 matrix. Let's use the procedures demonstrated above and the set of supplied reference character images to build a feature space for the alphabet consisting of four letters.

First generate partitions in our alphabet called "alpha."

```

> p := generatePartitions(alpha);
Then create a feature space called "space."
> space := Matrix(3,(nops(alpha)+1)):
> space[1,1] := char:
> space[2,1] := vert:
> space[3,1] := horz:
> for j from 2 by 1 to nops(alpha)+1 do
    space[1,j] := p[1,j-1]:
    space[2,j] := doBarCount(p[2,j-1], "vertical"):
    space[3,j] := doBarCount(p[3,j-1], "horizontal"):
od:
> space;

```

$$\begin{bmatrix} \text{char} & A & E & H & F \\ \text{vert} & 0 & 1 & 2 & 1 \\ \text{horz} & 1 & 3 & 1 & 2 \end{bmatrix}$$

>

Using this feature space as a reference the computer now has the ability to make an identification of the letter we have been analyzing.

```

> tableLookup(space, features);
      E

```

>

This is a demonstration of the feature vector concept. However, the computer will begin having problems with a correct identification when we expand the feature space to include all twenty six of the upper case alphabet characters.

```

> space := matrix([
[char,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z],
[vert,0,1,1,2,1,1,1,2,1,1,1,1,2,2,2,1,2,1,0,1,2,0,0,0,1,0],
[horz,1,3,2,2,3,2,2,1,0,0,0,1,0,0,2,2,2,2,0,1,0,0,0,0,0,2]]
);
space :=
[ char, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W,
, X, Y, Z]
[ vert, 0, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 1, 2, 2, 2, 1, 2, 1, 0, 1, 2, 0, 0, 0, 1, 0
]
[ horz, 1, 3, 2, 2, 3, 2, 2, 1, 0, 0, 0, 1, 0, 0, 2, 2, 2, 2, 0, 1, 0, 0, 0, 0, 0,
2]

```

>

The discrepancy is apparent by observing that the number of horizontal and

vertical components does not give us a unique feature vector for every single character. For example the letter "A" is the only one in this feature space with a single horizontal component and no vertical components, but the "I", "J", and "K" characters are indistinguishable from each other.

>

Additional Features

Duda mentions this issue in his description of the feature space concept. The answer is to define more features to be analyzed.

The construction of a feature space with twenty six (or more if we include punctuation and lower case letters) unique columns is beyond the scope of this worksheet. However, let's consider some possibilities.

Suppose we have the computer check for curvature in certain key regions of the character image. Here is a suggested set of divisions for any given character image.



Considering letters such as "B", "P" and "R" it makes sense to check two regions on the right hand side of the image. Curvature on the left however only occurs in letters such as "C", "G", "O", "Q" and "U". The letter "S" would be a special case, which should be the only letter to register curvature on the lower right and not on the upper right. This would add another five rows to our feature space and would result in a feature space something like this:

```
> space := matrix([
[char,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z],
[vert,0,1,1,2,1,1,1,2,1,1,1,1,2,2,2,1,2,1,0,1,2,0,0,0,1,0],
[horz,1,3,2,2,3,2,2,1,0,0,0,1,0,0,2,2,2,2,0,1,0,0,0,0,0,2],
[crva,0,0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0],
[crvb,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0],
[crvc,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0],
[crvd,0,1,0,1,0,0,1,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0],
[crve,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0]
]);
```

space :=

```
[char, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W,
, X, Y, Z]
[vert, 0, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 1, 2, 2, 2, 1, 2, 1, 0, 1, 2, 0, 0, 0, 1, 0
]
[horz, 1, 3, 2, 2, 3, 2, 2, 1, 0, 0, 0, 1, 0, 0, 2, 2, 2, 2, 0, 1, 0, 0, 0, 0, 0,
2]
[crva, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0,
0]
[crvb, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0]
[crvc, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
0]
[crvd, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0,
0]
[crve, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
0]
```

>

Although this helps resolve the difference between "J" and "I", it does nothing to help us distinguish between "V", "W", and "X". The computer needs more information.

Another feature to consider could be the something similar to the "distance-versus-angle" procedure described by Rafael Gonzalez and Paul Wintz in "Digital Image Processing". This would involve overlaying the character image onto a polar coordinate system and then taking a center to perimeter distance measurement for angles 0 through 2pi. Considering those distance values in Cartesian coordinate system, you could count the maxima as the corners of the character.

Given the letter "A", the resulting "distance-versus-angle" curve in Cartesian coordinates ought to have three maxima.

Implementing this feature analysis and incorporating it into our reference feature space would give the computer the ability to distinguish between the letter "V" and the letter "W". But it would still fail to make the final distinction between the letter "T" and the letter "L" which have the same features as defined so far.

The feature vector method of pattern recognition appeals to the intuitive sense of how pattern recognition is done. However, the feature extractor procedures are burdensome to the computer in times of execution time. This means that the success of this method is entirely dependent on the implementer's ability to

carefully define features that will create as many unique columns in the feature space as possible.