PharmaSUG 2021 - Paper AP-101

Re-pagination of ODS RTF Outputs to Automate Page Breaks and Minimize Splits Across Pages

Jeremy Gratt, Modular Informatics LLC
Aditya Tella, Seagen Inc.;

ABSTRACT

One of the more tedious tasks of clinical study report table creation involves planning out page breaks for RTF outputs. To improve the review of table outputs, there are groups of rows we would like to keep together to avoid splitting across pages. For presentation of individual parameters our goal is to group statistics or categories within a parameter on the same page; for by-visit displays we group rows by visit or by visit/parameter; and for multi-level categorical displays such as AEs and conmeds we try to minimize the splits within hierarchical tiers. Avoiding these inconvenient splits requires manual work on the part of programmers to generate organized outputs that aid communication. An automated approach to page breaks would relieve programmers of this manual task and improve the quality of outputs.

We present background, implementation experience, and programming details for a SAS pagination macro that employs an algorithm to automate page breaks while minimizing in-group splits across pages. The macro reads a standard ODS RTF file as input and generates a re-paginated RTF along with a QC data set. The SAS macro components include an RTF parser to read the input RTF file, an algorithm to calculate expected word-wrapping, a page break algorithm to output table rows, and a verification step to compare and confirm the output RTF content matches the QC data set.

INTRODUCTION

In the examples below, Page 1 and 2 of an adverse event table are presented to demonstrate content splitting across pages. The summarization uses a 4-level hierarchy using system organ class, high level group term, high level term and preferred term. In this example, the “White blood cell disorders” grouping does not fit on the page and must be split across two pages. When this happens, the hierarchy headings for not just “White blood cell disorders (cont’d)” but also the system organ class “Blood and lymphatic system distorders (cont’d)” should be re-drawn at the top of the 2nd page.

System Organ Class High Level Group Term High Level Term Preferred Term	Trt A (N=##) n (%)
Blood and lymphatic system disorders	2 (66.7)
Anaemias nonhaemolytic and marrow depression	2 (66.7)
Anaemias NEC	2 (66.7)
Anaemia	2 (66.7)
White blood cell disorders	2 (66.7)
Leukopenias NEC	2 (66.7)
Lymphopenia	2 (66.7)
Leukopenia	2 (66.7)
Dictionary: MedDRA v20.0	

Figure 1: Page 1 of content splitting example

System Organ Class High Level Group Term High Level Term Preferred Term	Trt A (N=##) n (%)
Blood and lymphatic system disorders (cont'd)	
White blood cell disorders (cont'd)	
Neutropenias	1 (33.3)
Neutropenia	1 (33.3)

Figure 2: Page 2 of content splitting example

To improve comprehension and readability of reports, it is our goal to minimize these kinds of splits across pages as much as possible. Keeping groups of information together aids review of results. Before the re-pagination algorithm was implemented, programmers would need to keep track of available lines and write their own code to insert these splits manually into the input reporting data set using the PGBREAK variable.

To improve table production and to automate these types of layout decisions, we developed an algorithm that would perform a "best fit" calculation for deciding where to put the page breaks in a manner that minimizes splits across pages.

There is another reason to implement a new re-pagination algorithm. There are times when SAS, both with ODS RTF and ODS TAGSETS.RTF, will produce output that is mistakenly spread across 2 pages. Creation of the new pagination algorithm was intended to fix this issue. In the example below demonstrating this type of error, the footnote intended for the bottom of Page 1 is printed at the top of Page 2 because SAS ODS assigned more content for Page 1 than was actually available:

White blood cell disorders	2 (66.7)
Leukopenias NEC	2 (66.7)
Lymphopenia	2 (66.7)
Leukopenia	2 (66.7)
Neutropenias	1 (33.3)
Neutropenia	1 (33.3)

Figure 3: Bottom of Page 1 - Pagination Error

Dictionary: MedDRA v20.0
Source: C:\path\to\program\t-ae.sas
Output: t-ae.rtf

Figure 4: Top of Page 2 - Pagination Error

THE PAGINATION ALGORITHM

For such an algorithm, we wanted to avoid these types of issues:

• **Cases where a new grouping starts near the bottom of the page:** If only one or two lines of a grouping fit on a page before continuing to the next page, then it is better to just force a page break to the next page to keep the grouping together.

• **Cases where a grouping with many rows starts near the top of the page:** On the other hand, if a page only has a few rows filled out, then if a page break is forced, it will look bad to force a page split.

• **Very long groupings:** If a group contains more rows than could even fit on a new page (for example a grouping with 50 rows), then splitting the information across pages cannot be avoided. In this case, the algorithm should just fill each page with whatever lines will fit, and split where required by dimensions of the page.

The pagination algorithm for splitting information across pages was defined as follows:

1. Measure the height of the next grouping of output rows (taking into account the available width inside of each cell, and any necessary word-wrap within cells).

2. If the group of rows would fit within the remaining height on the page, then output the rows on this page.

3. If the group of rows will not fit on the current page then do the following:

    a. If the group of rows by itself is so large that it would spread across more than one page, then output the rows and allow the grouping to split across pages.

    b. Otherwise, if at least 50% of the page has already been populated with rows of content, then force a page break and start the grouping of rows on the next page.

    c. If less than 50% of the page has been filled with rows of content, then go ahead and split the row grouping: place all rows that will fit on the current page, and then continue to place rows on the next page.

The reason for 3b and 3c in the algorithm was to avoid having the algorithm create very sparse pages with only a few lines of content on each page. In this case, we require at least half of the page to already contain rows of content before we consider to force a page break.

FEATURES OF THE SAS IMPLEMENTATION

We describe the SAS programming techniques used for the RTF re-pagination implementation:

1. Creating the temporary RTF using PROC REPORT ODS RTF
2. Reading the temporary RTF file into a QC data set and “re-drawing” the RTF
3. Parsing the structure of the RTF
4. Measuring text width using Adobe Font Metrics
5. Gathering measurements from RTF control words
6. Calculating word-wrap and vertical height of cells, rows, and row groups
7. Calculating available vertical height
8. Assigning page breaks
9. Comparing the QC data set to the final RTF.

1. CREATING THE TEMPORARY RTF USING PROC REPORT ODS RTF

The pagination algorithm was built into the existing reporting process represented in high-level terms by the diagram below.

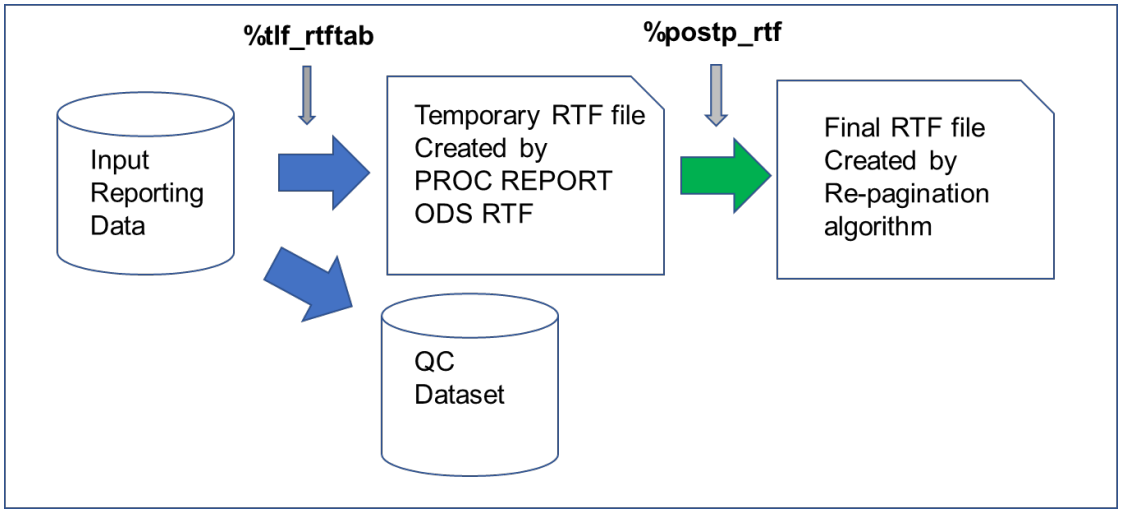

Figure 5: Pagination Macro Inputs and Outputs

The input reporting data created by the programmers is processed by the standard reporting macro %tlf_rtftab (blue arrows). This creates both a temporary RTF file, as well as a QC data set representing the contents of the RTF. Each observation of the QC data set corresponds to a row in the body of the RTF and is used for verification/validation activities, specifically (a) independent or dual programming of all values displayed in the table followed by a proc compare as part of routine programmer QC and also (b) confirmation that steps (2) through (8) in our repagination algorithm did not unintentionally drop information, as per step (9) described further below. The new pagination algorithm %postp_rtf (green arrow) is called inside of %tlf_rtftab and processes the temporary RTF to produce the final RTF with page breaks applied.

Below shows an example of the input reporting data that is prepared by the programmers and processed by the standard reporting macro:

ROW_LEVEL1	ROW_LEVEL2	ROW_LEVEL3	COL_1	COL_2	DISPSTAT	SKIPLINE	PGBREAK
Agents Acting On The Renin-Angiotensin System			2 (67)	20 (43)	1	0	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors And Calcium Channel Blockers		0	1 (2)	2	0	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors And Calcium Channel Blockers	Coroval B	0	1 (2)	3	1	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors And Diuretics		0	1 (2)	2	0	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors And Diuretics	Zestoretic	0	1 (2)	3	1	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors, Plain		2 (67)	10 (21)	2	0	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors, Plain	Lisinopril	1 (33)	7 (15)	3	0	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors, Plain	Enalapril	1 (33)	1 (2)	3	0	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors, Plain	Benazepril	0	1 (2)	3	0	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors, Plain	Benazepril Hydrochloride	0	1 (2)	3	0	0
Agents Acting On The Renin-Angiotensin System	Ace Inhibitors, Plain	Ramipril	0	1 (2)	3	1	0
Agents Acting On The Renin-Angiotensin System	Angiotensin Ii Antagonists And Diuretics		0	3 (6)	2	0	0
Agents Acting On The Renin-Angiotensin System	Angiotensin Ii Antagonists And Diuretics	Co-diovan	0	2 (4)	3	0	0
Agents Acting On The Renin-Angiotensin System	Angiotensin Ii Antagonists And Diuretics	Hyzaar	0	1 (2)	3	1	0
Agents Acting On The Renin-Angiotensin System	Angiotensin Ii Antagonists, Plain		0	6 (13)	2	0	0
Agents Acting On The Renin-Angiotensin System	Angiotensin Ii Antagonists, Plain	Irbesartan	0	2 (4)	3	0	0
Agents Acting On The Renin-Angiotensin System	Angiotensin Ii Antagonists, Plain	Losartan	0	2 (4)	3	0	0
Agents Acting On The Renin-Angiotensin System	Angiotensin Ii Antagonists, Plain	Losartan Potassium	0	2 (4)	3	0	0
Agents Acting On The Renin-Angiotensin System	Angiotensin Ii Antagonists, Plain	Olmesartan Medoxomil	0	1 (2)	3	0	0
Agents Acting On The Renin-Angiotensin System	Angiotensin Ii Antagonists, Plain	Valsartan	0	1 (2)	3	1	0

Figure 6: Sample Input Reporting Data

A description of the input variables is presented in this table:

Variable	Description
Row_level1 – Row_level3	These are the variables displayed in the left most column of the table, in hierarchical fashion (indented levels).
Col_1 – Col_2	These are the treatment or other columns displayed after the first column
Dispstat	Identifies the indented hierarchical reporting level for column 1. Level 1 is furthest to the left, level 2 has one indent, level 3 has 2 indents and so on.
Skipline	When skipline = 1 then add a blank row after the row. This identifies groupings of row content.
Pgbreak	When pgbreak = 1 then force a page break after the row. When the re-pagination algorithm is used this variable is not needed as page breaks are handled by the algorithm.

Table 1: Input Reporting Data Structure

The Row_level1, 2, and 3 variables are used to build the first column of the report. Internally in the %tlf_rtftab macro this is derived as the Catlabel variable. For each observation, the Catlabel variable is assigned the value of the non-missing Row_level# variable with highest value for #. E.g., in the example above, when Row_level3 is non-missing the value of Catlabel is assigned to Row_level3 including two indents. Indents are applied by adding the \li RTF control word.

The input data structure also includes the special variables **Dispstat**, **Skipline** and **Pgbreak**. The **Dispstat** variable identifies the column 1 hierarchical row levels (e.g., tabbed indents), the **Skipline** variable identifies where to add blank rows and thus defines groupings in the report, and the **Pgbreak** variable is for user-override requested page breaks. To pass this information to the pagination algorithm, we embed this information as invisible text in the first column of the RTF output. This is done using the RTF {\v} control word to create invisible text. Here is an example showing how the information is embedded into the column 1 Catlabel variable:

```
catlabel = "`R'{\v dispstat_"||strip(put(dispstat,8.0))
           ||" skipline_"||strip(put(skipline,8.))
           ||" pgbreak_"||strip(put(pgbreak,8.0))||"}'"
           ||strip(catlabel);
```

The temporary RTF file will look like below when invisible text is shown:

{tc "The Report Procedure" \f C \l 1}{tc "Table 1" \f C \l 2}		
Therapeutic Subgroup (ATC 2nd Level) Chemical Subgroup (ATC 4th Level) Preferred WHO Name	Trt A (N=xx) n (%)	Trt B (N=yy) n (%)
dispstat_1 skipline_0 pgbreak_0Agents Acting On The Renin-Angiotensin System	2 (67)	20 (43)
dispstat_2 skipline_0 pgbreak_0Ace Inhibitors And Calcium Channel Blockers	0	1 (2)
dispstat_3 skipline_1 pgbreak_0Coroval B	0	1 (2)
dispstat_2 skipline_0 pgbreak_0Ace Inhibitors And Diuretics	0	1 (2)
dispstat_3 skipline_1 pgbreak_0Zestoretic	0	1 (2)
dispstat_2 skipline_0 pgbreak_0Ace Inhibitors, Plain	2 (67)	10 (21)

Figure 7: Temporary RTF with invisible text

2. READING THE RTF FILE INTO A DATA SET AND "RE-DRAWING" THE RTF

To implement the re-pagination algorithm, we take a "measure everything" approach to the RTF. We read in the temporary RTF and measure the width and height of columns, rows, titles, footnotes, headers, cells, and the width and height of each character and word of text. With these precise measurements, we can "re-draw" the RTF in a manner that minimizes the amount of splitting across pages.

We start the re-pagination process by reading in the original RTF. The RTF is read one character at a time into the RTF_STREAM data set:

```
data rtf_stream;
   infile "&in_rtf" recfm=f lrecl=1;
   input e pib1.;
   format e 8. c $ascii1.;
   c=byte(e);
run;
```

The RTF_STREAM data set contains one record per character in the input RTF file – this includes both the visible text along with all the RTF control words. We then parse and analyze this data set to discover the starting and stopping character position of relevant portions inside the RTF. Through this analysis, we determine the position within the file of each row of the RTF, along with positions of other items such as the document header, titles, headers and footers. We end up with a data set PAGINATED_ROWS, which contains one observation per row in the RTF.

When it is time to "re-draw" the RTF, this is performed by using the PAGINATED_ROWS data set and a data _null_ step. For example, when we want to write the following row to the RTF:

Blood and lymphatic system disorders	2 (66.7)

Figure 8: A single row of output in the RTF

The PAGINATED ROWS data set stores the byte position of the row in the original RTF using the row_stream_start and row_stream_stop variables:

Newpagenum	Rown	Row_stream_start	Row_stream_stop	First_page
1	1	5981	6205	1

Table 2: Corresponding observation in the PAGINATED_ROWS data set

In the original RTF, at character positions 5981-6205 we will see the code for the RTF row. \trowd signifies the start of a row, and {\row} signifies the end of the row.

```
\trowd\trkeep\trqc
\cltxlrtb\clvertalt\clcbpat8\clpadt30\clpadft3\clpadr30\clpadfr3\cellx7063
\cltxlrtb\clvertalt\clcbpat8\clpadt30\clpadft3\clpadr30\clpadfr3\cellx9012
\pard\plain\intbl\sb30\sa30\ql\f1\fs20\cf1{Blood and lymphatic system disorders\cell}
\pard\plain\intbl\sb30\sa30\qc\f1\fs20\cf1{2 (66.7)\cell}
{\row}
```

The data _null_ step that writes these looks as follows:

```
data _null_;
   set paginated_rows end=end;
   by newpagenum rown;

   file "&out_rtf" recfm=f lrecl=1;
   format out_byte pib1.;
   . . .
   do i = row_stream_start to row_stream_stop;
```

```
      set rtf_stream point=i;
      out_byte=e;
      put @1 out_byte pib1.;
   end;
   . . .
```

In the code above, for each observation in our PAGINATED_ROWS data set, we read in the RTF_STREAM data set using the SET statement with the POINT= option. The DO loop uses the PUT statement to write to the output RTF every character from the row_stream_start to row_stream_stop positions.

Within the data _null_ step we repeat this approach to draw all other required elements of the RTF, including titles, headers, footnotes, and page breaks.

3. PARSING THE STRUCTURE OF THE RTF

After the RTF_STREAM file is created, a new data set WORDS is created that contains one observation per word in the RTF. A word may be either an RTF control word (like \fs20 for font size 10), or actual text to be displayed in the output.

For parsing the RTF, we take an approach described by the WHATWG HTML5 standards. For each part of an HTML document, different insertion modes are defined to control the processing. Examples of HTML insertion modes include: “initial” insertion mode, “in body” insertion mode, “in table” insertion mode, “in row” insertion mode, and “in cell” insertion mode. As we work through the WORDS data set word by word, we define similar insertion modes to read through the RTF content.

The processing uses a data step, using SET with POINT option to step through the document. The insertion mode starts with in_rtfheader=1, meaning we start in the RTF header. As we process each word, the insertion mode changes as different parts of the RTF are encountered. Similar to the HTML processing, the insertion mode changes from in_rtfheader to before_titles, before_header, in_body, in_footer, in_row, in_cell, and in_text.

Since the parsing data step contains many sections, we employ “named destinations” using link and return commands to better organize the code. The following code shows the central processing section of the data step. Each link statement is called to process a different section of the RTF:

```
   j = 1;
   do while( j le &n_words );
      set words point = j;
      link load_word;
      if in_rtf_header = 1 then do;
                . . .
      end;
      if before_titles = 1 then link before_titles;
      if before_header = 1 then link before_header;
      if in_header = 1 then link in_header;
      else
      if in_body = 1 then link in_body;
      else
      if in_footnote = 1 then link in_footnote;
      if in_footer = 1 then link in_footer;
      j = j + 1;
   end;
```

We toggle between the different insertion modes by setting the value of the insertion mode variables. For example, if we are in the before_header=1 insertion mode, then in the example below, when the code in the before_header named destination is executed, the RTF control word \trowd marks the end of the titles section of the RTF and the beginning of the column header rows, and we change the processing to the “in_header” insertion mode.

```
before_header:
  . . .
    if controlx = "\trowd" then do;
       before_header = 0;
       in_header = 1;
       . . .
    end;
  . . .
return;
```

The parsing data step produces 5 different data sets that are used to perform the pagination calculation and later re-draw the RTF.

RTF Parser Output Data Set	Structure	Description
RTF_FONTS	One obs per font definition in the RTF	Identifies fonts used in the RTF
PAGE_DIM	One obs	Page dimensions and margins
ROW_STREAM_ENDPOINTS	One obs per row in the RTF	Contains position of the start/end of each row and other row-based information
DATA_ROWS	One obs per data row in the RTF	Similar to ROW_STREAM_ENDPOINTS but keeps only the data rows from the body of the RTF table.
CELL_TEXT_ENDPOINTS	One obs per cell in the RTF	Contains position of the start/end of the text for each cell
CELL_DEF_ENDPOINTS	One obs per cell in the RTF	Contains the position of the cell definition, including information such as cell padding, margins, width

Table 3: RTF Parser Output Datasets

4. MEASURING THE WIDTH OF TEXT USING ADOBE FONT METRICS

For the “measure everything’ approach to handling the RTF file, it is necessary to measure both the lineheight and the width of every character of output. If text will not fit horizontally on a single line within a cell then the text must wrap to the next line. To measure this text we use Adobe Font Metrics files. AFM files may be found online, or may be extracted directly from system .ttf True Type font files using font design software like Fontlab 7.

AFM files are available for each font used in the RTF. There is a separate file for each font type: regular, bold, italic and bolditalic. These AFM files contain precise Individual Character Metrics including character widths. For example the AFM file for Times New Roman regular looks as follows:

```
25  StartCharMetrics 315
26  C 32 ; WX 250 ; N space ; B 0 0 0 0 ;
27  C 33 ; WX 333 ; N exclam ; B 130 -9 238 676 ;
28  C 34 ; WX 408 ; N quotedbl ; B 77 431 331 676 ;
29  C 35 ; WX 500 ; N numbersign ; B 5 0 496 662 ;
30  C 36 ; WX 500 ; N dollar ; B 44 -87 457 727 ;
31  C 37 ; WX 833 ; N percent ; B 61 -13 772 676 ;
```

Figure 9: Sample AFM text file contents

The number after the letter C is is ascii encoding for the character. The number after the WX contains a relative width for the character. The space character is ASCII 32, and in the above example the character width for the space character is WX = 250. To calculate width we multiply this value by the (scale factor of the font) / 1000. Thus a space character for 10 point Times New Roman would have the width = 250 * (10 / 1000) = 2.5 points (= 0.034722" with 72 points/inch).

To process the RTF character data efficiently, we define separate SAS formats for each Adobe font metric file. For example:

Format Name	Description
wx1fmt	Times New Roman Regular ASCII to Width
wx1bfmt	Times New Roman Bold ASCII to Width
wx1ifmt	Times New Roman Italic ASCII to Width
wx1bifmt	Times New Roman BoldItalic ASCII to Width
wx2fmt	Arial Regular ASCII to Width
wx2bfmt	Arial Bold ASCII to Width
. . .	. . .

Table 4: Formats that map ASCII to Font Width

To calculate the width of a word, we loop through each character and then use the PUTN function to add character width to the width of the word contained in the wordgroup_width variable;

```
** Example: ;
** this_fidx = "1" for Times New Roman;
** this_fsize = 10 for 10 pt font;
** e = numeric ascii value of the current character in the word;
wordgroup_width  = wordgroup_width
 + this_fsize * ( input(putn(e,"wx"||strip(this_fidx)||"fmt."),8.) / 1000
) ;
```

The PUTN statement allows us to choose the SAS format to convert ascii to width, based on the font type defined in the this_fidx variable. In the example above the wx1fmt. format is selected for Times New Roman.

5. GATHERING MEASUREMENTS FROM RTF CONTROL WORDS

In addition to measuring the width of text, for our calculations we also need to extract measurements directly from the RTF control words. If we go to our same example of the row for "Blood and lymphatic system disorders":

Blood and lymphatic system disorders	2 (66.7)

Table 5: A single row of output in the RTF

The rtf code contains RTF control words with relevant information:

```
\trowd\trkeep\trqc
\cltxlrtb\clvertalt\clcbpat8\clpadt30\clpadft3\clpadr30\clpadfr3\cellx7063
\cltxlrtb\clvertalt\clcbpat8\clpadt30\clpadft3\clpadr30\clpadfr3\cellx9012
\pard\plain\intbl\sb30\sa30\ql\f1\fs20\cf1{Blood and lymphatic system disorders\cell}
\pard\plain\intbl\sb30\sa30\qc\f1\fs20\cf1{2 (66.7)\cell}
{\row}
```

Figure 10: RTF code containing control words

The meaning of the highlighted control words are described in the following table:

RTF Control Word	Description
\cellx7063	The position of the right side of the first column in twips (1/20th of a point). In this example 7063 twips = 7063 / (20*72) = 4.904" width for 1st column.
\clpadt30	Cell Padding for the left side of the cell in twips = 0.0208"
\sb30 and \sa30	The space before and space after the cell on left and right hand side in twips = 0.0208"
\fs20	Font Size in half-points. \fs20 = 10-point font

Table 6: RTF Control Words and Description

All of these measurements are read from the RTF and incorporated into the measurement calculations. Spacing and padding may be small but over multiple rows if they are not taken into account it can affect the overall measurements and force the pagination algorithm to fail.

6. CALCULATING WORD-WRAP AND VERTICAL HEIGHT OF CELLS, ROWS, AND ROW GROUPS

For the page layout calculations, we must take into account the potential for "word-wrap". These are cases where text must spread across multiple lines. In the example cell below, the word "Disorders" will not fit on the first line and must be "wrapped" to the second line in the cell:

Blood and Lymphatic System Disorders

Table 7: RTF Cell with Word-Wrap

We process the CELL_TEXT_ENDPOINTS data set described in (3) above to calculate the height of each cell. Each observation in this data set corresponds to a cell in the RTF table, and the data set contains the numeric variables Cell_word_start and Cell_word_stop that point to the start/stop observation in the WORDS data set (one obs per word).

```
data cell_heights;
   set cell_text_endpoints end=end;
   . . .
   do j = cell_word_start to cell_word_stop;
         set words point=j;
          . . .
   end;
```

Within the j-loop above, the width of each word is measured and compared to the space remaining on the current line. If the word fits within the available cell width, then the current width is increased by the word width. If the word will not fit within the available cell width, then a new line is started and the width of the current line is set to the width of the word.

Once we have the height of each row calculated, then the vertical height required for each row of the RTF is determined to be the maximum required height across the cells in the row. Continuing in this fashion, we can then sum the required heights across multiple rows to determine the height of each row grouping used for the pagination algorithm.

7. CALCULATING AVAILABLE VERTICAL HEIGHT

To determine the vertical height available for data rows within the RTF, we analyze the first page of the RTF to determine the height of the titles, column headers and footnotes, determining the vertical height of each section using the method described in (6). The available height for display of data rows is reresented graphically in the figure below using the red arrow.

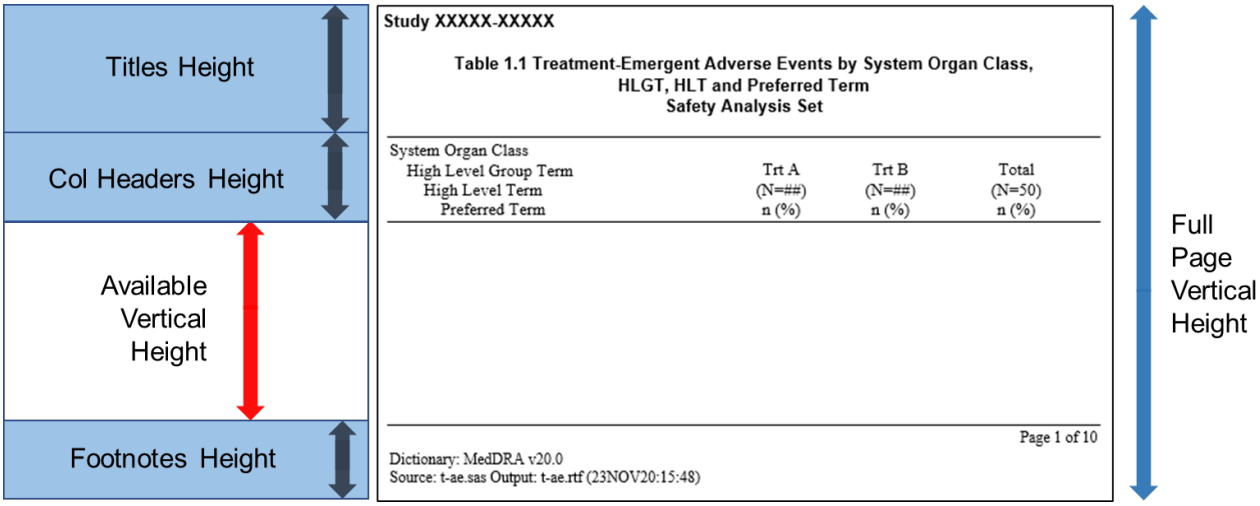

Figure 11: Available Vertical Height

The formula for Available Vertical Height =
(Full Page Vertical Height) – (Titles Height) – (Column Headers Height) – (Footnotes Height).

For our implementation, we only need to analyze page 1 because we assume that the titles, headers and footnotes will be the same on every page. Note that our implementation would require changes to be compatible with techniques that split RTF output columns across pages, for example when using the Proc Report DEFINE statement with ID and PAGE options.

8. ASSIGNING PAGE BREAKS

After calculating the height of each row group from (6) and the available height from (7), the next step is to assign new page breaks and new page numbers using the pagination algorithm. For the coding of page breaks, we use the DATA_ROWS data set described in (3) that contains one observation per row within the body of the report. In the code example below, we show the calculation to implement step 3b in the algorithm.

```
data paginated_rows;
   set data_rows;
   by rowgroup rown;
   ...;
   if first.rowgroup then do;
   ...;
   if rowgroup_lineheight le &available_height
      and remaining_height < .50*&available_height
```

```
            then do;
               newpagenum = newpagenum + 1;
               remaining_height = &available_height;
            end;

      end;
      ...;
   run;
```

Data Step & Macro Variables	Description
Rown	Unique data row number
Rowgroup	A grouping variable for each group of rows intended to keep together and separated by skipline=1.
Rowgroup_lineheight	The height of the group of rows in twips.
Remaining_height	The height in twips of the space available for
Newpagenum	The new page number created by the pagination algorithm.
&available_height	Height available on a page

Figure 12: Page Break Data Step Variables

An example of a step in the pagination algorithm is shown in the figure below.

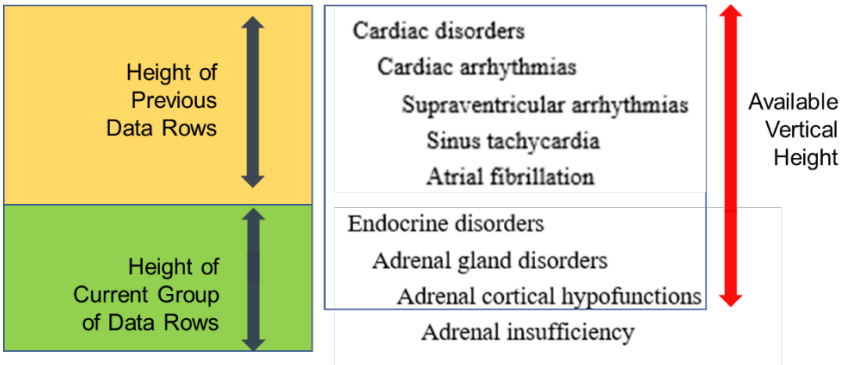

Figure 13: Pagination Example 1 (Green rows start after page break)

In the example above, the available vertical height (&available_height) is represented again by the red arrow. For this step, the "Cardiac disorders" grouping has been previously placed on the current page. The height of the "Endocrine disorders" grouping clearly exceeds remaining height (Available Vertical Height – Height of Previous Data Rows). Since the page has already been half-filled by the (yellow) "Cardiac disorders" rows, then according to the algorithm described in step 3b, this will force a page break and output the complete (green) "Endocrine disorders" on the next page.

In the next example, since the height of the (yellow) previous data rows for "Thyroid gland disorders" has not yet filled half of the available vertical height on the page, then according to the algorithm described in step 3c, we will split the current "Eye disorders" grouping after "Eye allergy". In this case the algorithm allows for splitting the grouping across pages in order to avoid sparse pages containing only a few lines of output.

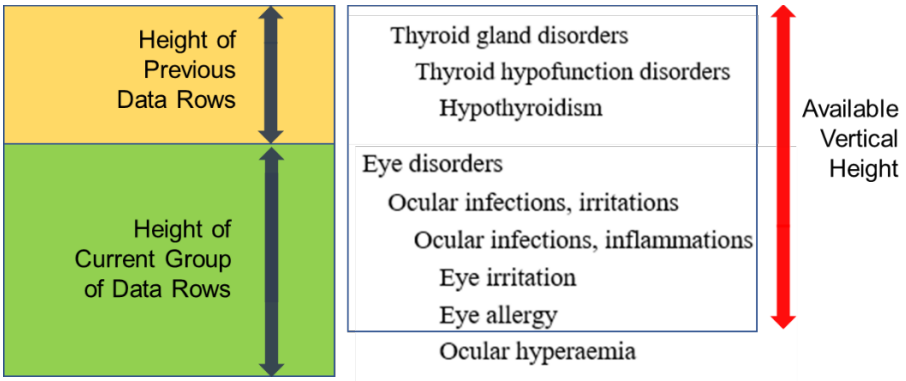

Figure 14: Pagination Example 2 (Green rows start after page break)

9. COMPARING THE RTF TO THE QC DATA SET

After the final RTF is created, the macro reads this RTF back in using the same steps described above, but this time it extracts all of the text content into a data set. This data set is compared to the input QC data set created by the RTFTAB macro. This helps to confirm that no content was lost through the RTF re-pagination process. For double-programming QC activities, the programmers must be assured that the QC data set is an accurate representation of the content found in the RTF. This additional step provides this verification.

MANUALLY CONFIRMING RTF MEASUREMENT ACCURACY

The re-pagination algorithm is dependent on the accuracy of the measurements collected from the RTF file. As part of the development process we manually compared the measured results in our macro vs. hand-measured results using the measurement tool in Adobe Acrobat. Although this crosscheck may introduce manual error using the mesurement tool, we see our vertical measurement error ranging from .001” - .07”. As a reference, a single line of 10pt times new roman uses roughly 0.20” of vertical space. In all of our measurements we have been able to keep our measurement error well below this amount.

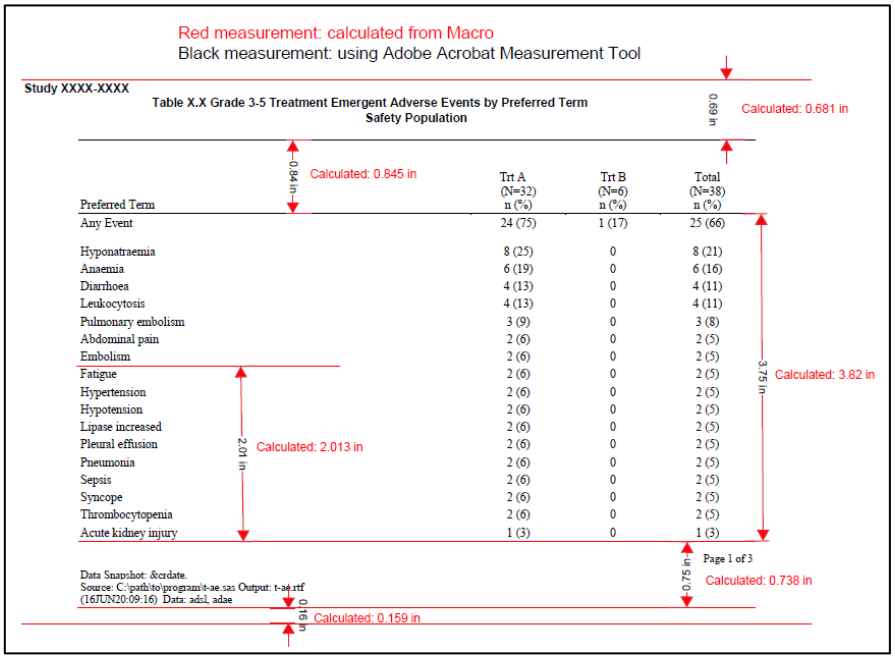

Figure 15: Accuracy of RTF Measurements

CONCLUSION

A new RTF re-pagination algorithm was developed that processes RTF output from PROC REPORT to create output with automated page breaks to minimize splits across pages. Now that the macro has been released and in use for several months in production, the algorithm has successfully limited the requirement for programmers to manually paginate their outputs and has led to new production efficiencies.

REFERENCES

Adobe Systems, Inc: “Adobe Font Metrics File Format Specification v4.1”, Accessed April 5th, 2021. https://adobe-type-tools.github.io/font-tech-notes/pdfs/5004.AFM_Spec.pdf

WHATWG: “WHATWG HTML Standard - 13.2.6.4: The rules for parsing tokens in HTML content”, Accessed April 5th, 2021. https://html.spec.whatwg.org/#parsing-main-inhtml

ACKNOWLEDGMENTS

Special thanks to Ellen Lin for suggestions on fine-tuning the pagination algorithm, and Michiel Hagendoorn for valuable feedback on the paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Jeremy Gratt
Modular Informatics, LLC
jeremy.gratt@m-informatics.com

Aditya Tella
Seagen Inc.
atella@seagen.com