Exporting to Latex

The purpose of this lecture is to build a function that can take data stored in a numeric array and output a fully formatted Latex table.

1 Formatting Data Into a String

The function `sprintf` is the main tool we will use to take numeric or text data and convert it into a Latex table. The function takes two inputs: a character array and the data. The character array is used to specify the way you want to format the data. The data is usually a numeric array.

The character array is referred to as `formatSpec`. It uses the syntax `'% options conversion_character'`, but we can also include ordinary text. The part of the `formatSpec` that begins with a percent sign and ends with a conversion character, while allowing for options in the middle, is referred to as a `formatting operator`. We will discuss what are the conversion characters and the options in a formatting operator.

The conversion character determines how to take the data and convert it to text. For example:

```
% print a float number
sprintf('%f', pi)
% print a float number with 2 numbers after the decimal point
sprintf('%.2f', pi)
% print a float number with 0 numbers after the decimal point
sprintf('%.0f', pi)
% print a float number with 4 numbers after the decimal point
sprintf('%.4f', pi)
% print a float number with the exponential notation
sprintf('%e', pi)
% print a float number with the exponential notation and only 2
% numbers after the decimal point
sprintf('%.2e', pi)
```

The `formatSpec` allows us to format numbers, but we can also position the formatted number in a phrase:

```
sprintf('Pi with 4 decimal cases: %.4f', pi)
sprintf('Pi with 16 decimal cases: %.16f', pi)
```

We can also insert text into text:

```
% ask user to input a letter
```

```
done = false
while ~done
    letter = input('What is your favorite letter? ', 's');
    if ischar(letter) && length(letter) == 1
        done = true
    else
        disp('Please respond with a single character');
    end
end
% print a single character
sprintf('Your favorite letter is %c', letter)

% ask user to input a name
done = false
while ~done
    name = input('What is your name? ', 's');
    if ischar(name) && length(name) > 0
        done = true
    else
        disp('Please respond with your name');
    end
end
% print a string
sprintf('Hello, %s!', name)
```

We can use `sprintf` to generate text and fill it with data. We can pass more than one number to the function. If there is only one formatting operator, then the `formatSpec` is repeated for each data passed to the function.

```
data = [1.2, 2.1, 3.4];
% print multiple formatted numbers
sprintf('%.2f,', data)
sprintf('%.2f,', pi, exp(1), log(10))
```

We can specify multiple formatting operators to deal with data arrays:

```
sprintf('%.2f,%.2f,%.2f', 1.2, 3.1, 4.2)
```

We can specify the order that the data should appear in the formatted text with the syntax `position$`:

```
% display first the last number, than the first number, and
   finally
% the middle number
sprintf('%3$.2f   %1$.2f   %2$.2f', 1.2, 3.1, 4.2)
```

It is possible to specify the width (number of characters) of the formatted data:

```
% specify the width of the formatted data
sprintf('|%10s|%10s|', "Header A", "Header B")
sprintf('%8.2f,%8.2f', [pi, 2*pi])
```

Finally, we can add flags to the formatting operator:

```
% add the flag '-' to left-justify
sprintf('|%-10s|%10s|', "Header A", "Header B")
% add the flag '+' to display the sign character
sprintf('%-+10.4f|%+10.4f', pi, -pi)
% add the flag '0' to pad the width with zeros
sprintf('%010.4f', pi)
```

We covered the options we will use to create Latex tables from data, but more details are available on the `sprintf` reference page.

The final piece we need to understand are special characters. For example, in the `formatSpec` the percent sign is a special character that determines when the `formatting operator` begins. There are other special characters that we can use in the `formatSpec`:

```
% \n represents a new line
sprintf('%-14s|%14s\n%-14s|%14s', 'First Name', 'Last Name',
   ...
        'Guilherme', 'Salome')
% \t represents a horizontal tab
sprintf('This tab \t add a lot of space')
% \\ represents a backslash
sprintf('This is a backslash \\')
% %% represents a single percent sign
sprintf(['A double percent sign %%%% is not interpreted as
   the start ' ...
         'of a formatting operator, but rather as a single %%
            '])
% '' represents a single quotation mark
sprintf('We use '''' to represent a single quotation mark ''.
    ')
```

We can now use this function to start building a function that takes data and outputs a Latex table that we can add to a `.tex` file.

2 Basic Table in Latex

A basic table in Latex can be constructed with the following code:

```
\begin{table}[options]
  \centering
  \begin{tabular}{alignment}
    Header A & Header B & Header C\\
    \toprule
    1.34 & 2.21 & -3.78\\
    (0.2) & (0.3) & (0.4)\\
    \midrule
    A & B & C\\
    \bottomrule
  \end{tabular}
```

```
  \caption{Important stuff.}
  \label{table_label}
\end{table}
```

I use the package `float` (see here for more details) to position figures in Latex files. This implies that the `options` in the code above becomes `[H]`. The `alignment` specifies how to align the columns of the tables. In the table above, `alignment` could be substituted for `ccc`. If you are not familiar with how to construct tables in Latex, please refer to this tutorial. Compiling the code above yields Table 1.

Header A	Header B	Header C
1.34	2.21	-3.78
(0.2)	(0.3)	(0.4)
A	B	C

Table 1: Important stuff.

3 Generating a Basic Table in Latex With Code

Let's create Table 1 with code. We will use the function `sprintf` to generate the final table, but we will also use the functions `join` and `strcat` to manipulate string arrays.

```
% create the beginning of the table
begin_table = ["\\begin{table}[H]", ...
               "  \\centering", ...,
               "  \\begin{tabular}{ccc}"];
begin_table = join(begin_table, '\n');

% create the end of the table
end_table = ["  \\end{tabular}", ...
             "  \\caption{Important stuff.}", ...
             "  \\label{tbl:label}", ...
             "\\end{table}"];
end_table = join(end_table, '\n');

% create the middle part of the table
headers = join(["Header A", "Header B", "Header C\\\\\n"], '&
   ');
headers = strcat("    ", headers, "    \\toprule");

data = [1.34 2.21 -3.78;
        0.2  0.3   0.4];
middle1 = strcat("    ", sprintf('%.2f & %.2f & %.2f', data
   (1, :)), ...
                 "\\\\\n");
middle2 = strcat("    ", sprintf('(%.2f) & (%.2f) & (%.2f)',
   data(2, :)), ...
                 "\\\\");
```

```
middle = strcat(middle1, middle2);

bottom = ["     \\midrule", ...
          "     A & B & C\\\\", ...
          "     \\bottomrule"];
bottom = join(bottom, '\n');

middle_table = join([headers middle bottom], '\n');

% join everything and apply sprintf to obtain the table
latex_table = sprintf(join([begin_table middle_table
   end_table], '\n'));
disp(latex_table)
```

The code above generates a table that we can directly copy and paste into a `.tex` file. However, it is specialized to generate Table 1. Next, we will create a function that allow us to pass arbitrary headers and data and obtain a Latex table from that.

4 A Function for Generating Latex Tables from Data

To create a function that generates a Latex table from data, we need to be able to substitute the headers, the data and the bottom with arbitrary arrays.

Let's start with the beginning of the table:

```
% matrix_to_latex.m
function latex_table = matrix_to_latex(headers, data, bottom,
    caption, label)
    space = "  ";
    [rows, cols] = size(data);
    %% Beginning of the table
    alignment = join(repmat("c", cols, 1), '');
    begin_table = ["\\begin{table}[H]", ...
                   strcat(space, "\\centering"), ...,
                   strcat(space, "\\begin{tabular}{",
                      alignment, "}")];
    begin_table = join(begin_table, '\n');

    %% Join all to create the Latex table
    latex_table = sprintf(begin_table);
end
```

Now, we will extend the function to add the end of the table:

```
% matrix_to_latex.m
function latex_table = matrix_to_latex(headers, data, bottom,
    caption, label)
    space = "  ";
    [rows, cols] = size(data);
    %% Beginning of the table
```

```
    alignment = join(repmat("c", cols, 1), '');
    begin_table = ["\\begin{table}[H]", ...
                   strcat(space, "\\centering"), ...,
                   strcat(space, "\\begin{tabular}{",
                      alignment, "}")];
    begin_table = join(begin_table, '\n');
    %% End of the table
    end_table = [strcat(space, "\\end{tabular}"), ...
                 strcat(space, "\\caption{", caption, "}"),
                    ...
                 strcat(space, "\\label{", label, "}"), ...
                 "\\end{table}"];
    end_table = join(end_table, '\n');
    %% Join all to create the Latex table
    latex_table = sprintf(join([begin_table, end_table], '\n'
       ));
end
```

Next, we add the headers to the table:

```
% matrix_to_latex.m
function latex_table = matrix_to_latex(headers, data, bottom,
    caption, label)
    space = "  ";
    [rows, cols] = size(data);
    %% Beginning of the table
    % ...
    %% End of the table
    % ...
    %% Headers
    headers_row = [strcat(space, space, join(headers, '&'),
       "\\\\"), ...
                   strcat(space, space, "\\toprule")];
    %% Middle of the table
    middle_table = join(headers_row, '\n');
    %% Join all to create the Latex table
    latex_table = sprintf(join([begin_table, middle_table,
       end_table], '\n'));
end
```

Next, add the data to the table:

```
% matrix_to_latex.m
function latex_table = matrix_to_latex(headers, data, bottom,
    caption, label)
    space = "  ";
    [rows, cols] = size(data);
    %% Beginning of the table
    % ...
    %% End of the table
```

```
    % ...
    %% Headers
    % ...
    %% Data
    data_rows = strings(rows + 1, 1);
    for i = 1:rows
        new_row = strip(sprintf('%.4f &', data(i, :)), 'right
           ', '&');
        data_rows(i) = strcat(space, space, new_row, "\\\\");
    end
    data_rows(rows + 1) = strcat(space, space, "\\midrule");
    data_rows = join(data_rows, '\n');
    %% Middle of the table
    middle_table = join([headers_row data_rows], '\n');
    %% Join all to create the Latex table
    latex_table = sprintf(join([begin_table, middle_table,
       end_table], '\n'));
end
```

Notice the use of `strip` to remove the trailing ampersand left by `sprintf`.

Complete the function by adding the bottom row:

```
% matrix_to_latex.m
function latex_table = matrix_to_latex(headers, data, bottom,
    caption, label)
    space = "  ";
    [rows, cols] = size(data);
    %% Beginning of the table
    % ...
    %% End of the table
    % ...
    %% Headers
    % ...
    %% Data
    % ...
    %% Bottom
    bottom_row = [strcat(space, space, join(bottom, '&'),
       "\\\\"), ...
                  strcat(space, space, "\\bottomrule")];
    %% Middle of the table
    middle_table = join([headers_row data_rows bottom_row], '
       \n');
    %% Join all to create the Latex table
    latex_table = sprintf(join([begin_table, middle_table,
       end_table], '\n'));
end
```

The complete function with light documentation is:

```
% matrix_to_latex.m
```

```
function latex_table = matrix_to_latex(headers, data, bottom,
    caption, label)
% matrix_to_latex generates a Latex table from data stored in
    a
% numeric matrix
    space = "  ";
    [rows, cols] = size(data);
    %% Beginning of the table
    alignment = join(repmat("c", cols, 1), '');
    begin_table = ["\\begin{table}[H]", ...
                   strcat(space, "\\centering"), ...,
                   strcat(space, "\\begin{tabular}{",
                      alignment, "}")];
    begin_table = join(begin_table, '\n');
    %% End of the table
    end_table = [strcat(space, "\\end{tabular}"), ...
                 strcat(space, "\\caption{", caption, "}"),
                    ...
                 strcat(space, "\\label{", label, "}"), ...
                 "\\end{table}"];
    end_table = join(end_table, '\n');
    %% Headers
    headers_row = [strcat(space, space, join(headers, '&'),
       "\\\\"), ...
                   strcat(space, space, "\\toprule")];
    %% Data
    data_rows = strings(rows + 1, 1);
    for i = 1:rows
        new_row = strip(sprintf('%.4f &', data(i, :)), 'right
           ', '&');
        data_rows(i) = strcat(space, space, new_row, "\\\\");
    end
    data_rows(rows + 1) = strcat(space, space, "\\midrule");
    data_rows = join(data_rows, '\n');
    %% Bottom
    bottom_row = [strcat(space, space, join(bottom, '&'),
       "\\\\"), ...
                  strcat(space, space, "\\bottomrule")];
    %% Middle of the table
    middle_table = join([headers_row data_rows bottom_row], '
       \n');
    %% Join all to create the Latex table
    latex_table = sprintf(join([begin_table, middle_table,
       end_table], '\n'));
end
```

Let's generate some fake data to test the function:

```
ltable = matrix_to_latex(["A", "B", "C", "D", "E"], rand(2,
```

```
    5), ...
2                                   ["a", "b", "c", "d", "e"], 'Test
                                       Table Function', ...
3                                   'tbl:890_matlab_test_table_function'
                                       );
```

We can compile the output to obtain Table 2.

A	B	C	D	E
0.3510	0.4018	0.2399	0.1839	0.4173
0.5132	0.0760	0.1233	0.2400	0.0497
a	b	c	d	e

Table 2: Test Table Function

5 Extending the Function to Allow for Row Headers

The `matrix_to_latex` function generates tables with a column header. However, we often also need to add a column with headers for the rows. Let's modify the code so that it can take both column headers and row headers.

We need to add an extra column, so we need to modify the `alignment` variable to have an extra column. We also need to add an empty string to the headers for columns. Then, we need to change the `data_rows` so that the first value in each row is one of the header rows. Last, we need to add an empty string to the bottom row.

```
1  % matrix_to_latex.m
2  function latex_table = matrix_to_latex(col_headers,
       row_headers, data, bottom, caption, label)
3  % matrix_to_latex generates a Latex table from data stored in
       a
4  % numeric matrix
5      space = "  ";
6      [rows, cols] = size(data);
7      col_for_row_headers = 1;
8      %% Beginning of the table
9      total_cols = cols + col_for_row_headers;
10     alignment = join(repmat("c", total_cols, 1), '');
11     begin_table = ["\\begin{table}[H]", ...
12                    strcat(space, "\\centering"), ...,
13                    strcat(space, "\\begin{tabular}{",
                          alignment, "}")];
14     begin_table = join(begin_table, '\n');
15     %% End of the table
16     end_table = [strcat(space, "\\end{tabular}"), ...
17                  strcat(space, "\\caption{", caption, "}"),
                        ...
18                  strcat(space, "\\label{", label, "}"), ...
19                  "\\end{table}"];
```

```
    end_table = join(end_table, '\n');
    %% Headers
    col_headers = horzcat("", col_headers); % col_headers is
       a row vector
    headers_row = [strcat(space, space, join(col_headers, '&'
       ), "\\\\"), ...
                   strcat(space, space, "\\toprule")];
    %% Data
    data_rows = strings(rows + 1, 1);
    for i = 1:rows
        new_row = strip(sprintf('%.4f &', data(i, :)), 'right
           ', '&');
        header_col = strcat(row_headers(i), ' &');
        data_rows(i) = strcat(space, space, header_col,
           new_row, "\\\\");
    end
    data_rows(rows + 1) = strcat(space, space, "\\midrule");
    data_rows = join(data_rows, '\n');
    %% Bottom
    bottom = horzcat("", bottom);
    bottom_row = [strcat(space, space, join(bottom, '&'),
       "\\\\"), ...
                  strcat(space, space, "\\bottomrule")];
    %% Middle of the table
    middle_table = join([headers_row data_rows bottom_row], '
       \n');
    %% Join all to create the Latex table
    latex_table = sprintf(join([begin_table, middle_table,
       end_table], '\n'));
end
```

Let's create some data to test the function:

```
betas = [0.3213; 0.5456; -0.53];
stderr = [0.02; 0.1; 0.05];
data = vertcat(betas', stderr');
col_headers = ["$\\hat{\\alpha}$", "$\\hat{\\beta}_1$", "$\\
   hat{\\beta}_2$"];
row_headers = ["Estimates", "Std. Error"];
bottom = ["", "", ""];
caption = "Regression Results"
label = "tbl:890_matlab_regression_results_example";
disp(matrix_to_latex(col_headers, row_headers, data, bottom,
   caption, ...
                        label))
```

Notice the use of two backslashes in the `col_headers`.

We can compile the output to obtain Table 3.

	$\hat{\alpha}$	$\hat{\beta}_1$	$\hat{\beta}_2$
Estimates	0.3213	0.5456	-0.5300
Std. Error	0.0200	0.1000	0.0500

Table 3: Regression Results

The function we just created is good enough for the majority of cases. We will extend it in some ways over the assignment problems. The function is useful to quickly export our results to a format that can be presented to others. However, if you need a table that is very specialized, it is often faster to export a crude version of the table first, say using `matrix_to_latex`, and then modify it to your liking directly in Latex.

6 Using the Format Specification to Import Data

The format specification used in the function `sprintf` is also used in other programming languages for formatting text. We can also use it with the function `textscan` to load data from text files in any format.

Before we use `textscan` we need to understand how to open and traverse files in Matlab. We can open a file with the function `fopen`, which returns an integer representing the file ID. This integer number is what Matlab uses to identify files it has opened.

```
% open the AAPL.csv file
fID = fopen('AAPL.csv');
% the ID should be an integer >= 3, if the ID is -1 there was a
% problem and the file could not be opened
if fID == -1
    error('File could not be opened.');
end
```

We can read lines of the file by calling `fgetl` multiple times:

```
% read the first line of the file
disp(fgetl(fID))
% notice that fgetl removes the newline character \n from the line

% read the next line
disp(fgetl(fID))
% each time we call fgetl we move on to the next line
disp(fgetl(fID))
disp(fgetl(fID))
```

We can keep calling `fgetl` to get all lines of the file. When there are no more lines to get, `fgetl` returns a `-1`. We can use this behavior to know when to step reading from the file. Let's rewind to the beginning of the file using `frewind` and count the number of lines in the file:

```
% rewind back to the first line
frewind(fID);
% count the number of lines
total = 0;
done = false;
while ~done
    file_line = fgetl(fID);
    if file_line == -1
        done = true;
    else
        total = total + 1;
    end
end
disp(sprintf('Number of lines in file: %.0f', total));
```

We can use `textscan` to parse each line of the file:

```
% rewind back to the first line
frewind(fID);
% get a line
file_line = fgetl(fID);
% display a line to analyze contents
disp(file_line);
% the line contains: integer, a comma, another integer, a
   comma,
% and a float with 2 decimal cases

% use textscan to parse the line
values = textscan(file_line, '%d,%d,%.2f');
```

Notice that `textscan` outputs a cell array, since a cell array can different data types. We can continue reading each line, parse it with `textscan` and concatenate the parsed values.

```
% get a new line
file_line = fgetl(fID);
new_values = textscan(file_line, '%d,%d,%.2f');
% join the results
values = vertcat(values, new_values);

% repeat for the rest of the file
done = false;
while ~done
    file_line = fgetl(fID);
    if file_line == -1
        done = true;
    else
        new_values = textscan(file_line, '%d,%d,%.2f');
        values = vertcat(values, new_values);
```

```
    end
end
```

The issue with while-loop above is that it recreates a cell array on each iteration, copying all the data to a new array that is one row larger. To be efficient, we need to pre-allocate the cell array:

```
frewind(fID);
% pre-allocate cell array for results
results = cell(total, 3);
% loop over lines
done = false;
for i=1:total
    file_line = fgetl(fID);
    results(i, :) = textscan(file_line, '%d,%d,%.2f');
end

% join results
dates = vertcat(results{:, 1});
times = vertcat(results{:, 2});
prices = vertcat(results{:, 3});

% we can also put the results in a table for easier data
   management
data = table(dates, times, prices, 'VariableNames', ["date",
   "time", "price"]);
```

Now that we have extracted the data from the file, we can close it using `fclose`:

```
fclose(fID);
% returns 0 if successful, and -1 if some error occurs
```

The approach we used above can handle data in various formats, provided we adapt the format specification. For example, if the stock data was saved in a `.txt` file where columns were separated by tabs, then we could use the special symbol `\t` in the format specification to parse the lines: `'%d\t%d\%.2f'`. If the data also had strings, then we could parse them with the `'%s'` format specification. In addition, if there were fields you wished to ignore when parsing a line, then you could use the special operator `*` right after the percent sign:

```
fID = fopen('AAPL.csv');
file_line = fgetl(fID);
% ignore the date and time stamps
disp(textscan(file_line, '%*d,%*d,%.2f'));
```

To recap: we open a file with `fopen`, go through each line with `fgetl`, parse the line with `textscan` and an appropriate format specification, save the data to matrices or a table, and close the file with `fclose`.

The `textscan` function works differently if instead of passing it a character array, we pass it a file ID. When `textscan` receives a file ID, it will go through all lines of the associated file and parse each of them using the format specification.

```
% open file, parse lines and close it
fID = fopen('AAPL.csv');
results = textscan(fID, '%d,%d,%.2f');
fclose(fID);
% store results in a table
disp(results);
data = table(results{:}, 'VariableNames', ["date", "time", "
   price"]);
```

7 Assignment

Problem 1 *Extend* `matrix_to_latex` *so that even lines in the* `data` *matrix are formatted with* `'(%.4f) &'` *instead of* `'%.4f &'`*. This is useful when reporting standard error estimates below the coefficient estimates.*

Problem 2 *What happens with* `matrix_to_latex` *if the input* `col_headers` *is not a string array, but a numeric array? Does* `matrix_to_latex` *output a helpful error message?*

Problem 3 *Is it possible to make* `matrix_to_latex` *work with numeric headers? Modify it so that it can deal with* `col_headers` *being a string array, a numeric array or a cell array.*

Problem 4 *Add input validation to the* `matrix_to_latex` *function.*

Problem 5 *Add documentation to the* `matrix_to_latex` *function. Your documentation is good enough when someone that has never seen your code can read the documentation of the function and use it without issues.*

Problem 6 *Allow the* `bottom` *input to be optional. How should the table look like if there is no bottom row?*

Problem 7 *Allow the* `caption` *and* `label` *inputs to be optional. How should the table look like if there is no caption and no label?*

Problem 8 *Extend the* `matrix_to_latex` *function using the function* `clipboard`*.*

Problem 9 *Run the linear regression suggested in Equation (8) of Pace and Barry (1997). Report the estimation results using the function* `matrix_to_latex`*.*

Problem 10 *(Summary Statistics) Read the documentation of the functions* `mean`*,* `median`*,* `quantile`*,* `corrcoef`*,* `autocorr`*,* `var` *and* `std`*. Create a function that takes as input:*

- *A numeric array where each column represents different variables and each row represents observations of the variables;*
- *A string array with the names of the variables*

The function should output a Latex table with the summary statistics of all the variables that were passed to the function.

References

Pace, R Kelley and Ronald Barry (1997). "Sparse spatial autoregressions". In: *Statistics & Probability Letters* 33.3, pp. 291–297. URL: https://doi.org/10.1016/S0167-7152(96)00140-X.