2017 PSS SUMMER SCHOOL
Assignment 2b

Due: 12:30pm on June 26

1 Submission instructions

We continue from where we left off in `a2a_m1_2017` (assignment 2a). The *Word Search* problem is a completely new one and the *Bonus* section references the *Estimating Macroeconomic Variables* from assignment 2a. Your submission should be contained in the same zip folder as assignment 2a, `a2_m1_2017/` but simply have more files and code in it.

2 Word Search

In this next set of problems you will be tasked with creating a series of functions that work to find a given word in a matrix of characters. Your aggregate function, which will use the smaller functions we guide you through, should take in a matrix and the word you are searching for. It should output the starting location of the word and the direction it is spelled in. The word must be spelled vertically, horizontally, or diagonally. See the schematic below for an idea of what your aggregate function should do.

$$\underbrace{\left(\begin{bmatrix} B & P & H & F & S & X & Y \\ Y & C & U & E & T & Z & F \\ E & D & S & M & L & Y & J \\ L & W & P & N & H & I & B \\ L & F & J & V & A & H & D \\ E & R & S & S & U & D & R \\ N & W & E & V & P & L & E \end{bmatrix}, \text{(YELLEN)}\right)}_{\text{inputs}} \xrightarrow{\text{function}} \underbrace{\left((2,1), (\text{down})\right)}_{\text{outputs}}$$

In light of the objective to create a function described above, keep all of your functions modular. They should have specific jobs with specific inputs and outputs. Sets of small, portable, and clear functions are better than a single complex and monolithic function. We have included in the assignment zip folder a matrix and word you can test your function with. In class we will choose some student's functions to work on a larger matrix searching for new words.

Problem 2.1. Create a function that finds the position(s) of the first character in your search word, save it as `find_firstchar` in `answers`. The inputs should be the matrix and the first character of the word. The output should be numeric coordinate pairs in (row,col) format. There will likely be more than one instance of your letter in the matrix. You will need to output all of them. For example, if you are working with the matrix and vector from above your function should return the coordinates: `(2,1)`, `(1,7)`, and `(3,6)`.

Problem 2.2. We are going to look character by character for the word, searching sequentially. Create a function that returns all possible positions that the next character in your search could be in given the location of the preceding character. This function should be called `make_poss_coords`, saved in `answers`, and should take as input your matrix and the location of the single preceding character. It should output a list off all possible coordinates for the next character. For example, for `(1,7)` your function should return all coordinates around it. These would be `(1,6), (2,6),` and `(2,7)`. Remember you *do not* want to search backwards in the word or if you are on the edge of the matrix include any positions not in the matrix.

Problem 2.3. Create a function that determines which of the possible coordinates actually contains the next character in your word *and* is in the correct direction. It should be called `eval_coords`, saved in `answers`, and should output a subsetted version of the list from `make_poss_coords`. It should take as input the matrix, the set of coordinates output by `make_poss_coords`, the position of the previous two characters (so as to determine direction), and the next character. This function should be able to handle the case when the second previous character's position is `NA`, e.g. when we have only found the first character.

Problem 2.4. Create a function that determines in which direction two adjacent matrix elements are ordered. Your function should answer questions such as, is the second element left from the first one? Is the second element diagonally up-right from the first? etc.. The function should take as input two coordinate pairs and output a character string displaying the direction they are ordered in. For diagonal directions combine the directions in y-x order, e.g. up-right or down-left. Remember to include an error catch portion of your function that reports if the two elements are not in fact adjacent to each other in the matrix. Name and save this function as `det_which_dir` in `answers`.

Problem 2.5. Create your aggregate function, `word_search`, saved in `answers`, that uses the functions from above (and more if necessary) to find all instances of a word in a matrix. Remember that during the middle of your search there may be more than one path of correct letters, you need to store these and continue to search along all of them. If one turns out to be incorrect, you should delete it. Functions in the `apply` family will be useful here, This function should take as input a matrix of characters and a search word. It should output the location(s) of the starting character(s) and the direction the word is spelled in. If a given word is not in your matrix, return a character string displaying to the user that this is the case. In the assignment's Rdata file we have included `word_test` and `mtx_test` to help you debug your code.
HINT: The most difficult part of this problem is dealing appropriately with the "found" pieces of your word as you go through your search. It is best to view this collection of possibly correct answers as a library of sorts. Each possible answer is an entry in this library and has attributes that will be needed throughout the search, e.g. all past correct coordinates or all possible coordinates of the next character. You may store this "library" anyway you find useful, we found lists where each element pertains to a possible instance of the word in the matrix to be useful. The search process has portions that are dependent on previous iterations, such as when determining if a new letter is correct by using the direction of previous letters. This means `apply`-like statements cannot always be used, nonetheless they should be used where possible. We have written psuedo-code, see Algorithm 1, that provide an example of how you might structure your code. Feel free to write yours as similar or different from ours as you please.

Algorithm 1 Word Search Algorithm

```
procedure WORD_SEARCH
    initialize matrix and word
    split word into vector of characters
    run find_first_char() on matrix and word
    create library of possible instances of the word (# times first character was found)
    store coordinates of first location(s) in library
    run make_poss_coords() on location(s) of first character
    store possible coordinates in library
    for character i in second character : last character do
        store next letter (i+1) needed in library
        run eval_coords() on each library entry's possible coordinates
        Delete library entries where no possible coordinates exist
        if no library entries exist then
            exit procedure and report the word was not found
        create new library entries for entries that found >1 possible coordinates
        run make_poss_coords() on location(s) of character i
        update library attributes (possible next coordinates, history of correct coordinates, etc.)
    if library has entries (e.g. has found words) then
        for found instance i in found instance 1 : found instance n do
            run det_which_dir() on instance i (two characters of it)
        format library for output, show starting location and direction
    else
        exit procedure and report the word was not found
    return results
```

Problem 2.6. Write an `lapply` statement that can find the location (or lack thereof) of a set of words in a single matrix. We will test this in class. Wrap the `lapply` statement in a function and save it as `lword_search` in `answers`.

3 Bonus

The following questions will push you to think hard about complex programming topics that are common in the current research environment. For those that successfully complete them, bonus points will be awarded on the assignment. Submit the answers to the bonus questions in a separate .R file appending `_bonus` at the end of the filename.

Problem 3.7. This problem is a continuation of the problem from assignment 2a. In the previous problems in that section you found both an optimal $\vec{k}$ given an arbitrary $\vec{\gamma}$ and an optimal $\vec{\gamma}$ given an arbitrary $\vec{k}$. Now you will write function(s) to do both at once. Remember that the possibilities for $\vec{k}$ are finite, as they must be integers where none can be greater than the number of years we have in our data, this puts useful constraints on the solution. Use a package function such as the one suggested above to do your constrained optimization. As a starting point we have included a pseudo-code algorithm, see Algorithm 2, that gives a hint as to a code structure you could use. This is an open ended question on purpose, we will reward hard work and creativity not simply having the correct solution.

COMMENT: I thought it best to do a nested optimization function because the inequality constraints for the $\vec{\gamma}$ search depend on having an X_M which cannot be created without already having a "guess" for $\vec{k}$. If you can figure out a way to do the optimization at once, simultaneously finding $\vec{\gamma}$ and $\vec{k}$, I would be very interested to hear about it.

Algorithm 2 Bonus Algorithm

```
 1: procedure FIND $\vec{\gamma}$'S AND $\vec{k}$'S
 2:     Define inputs $G_{\text{obs}}$ and your initial guesses for $\vec{k}_\theta$, $\vec{\gamma}_\theta$
 3:     Enter outside minimization routine to find optimal $\vec{k}$
 4:         Subset $G_{\text{obs}}$ based on initial guess for $\vec{k}$
 5:         Compute constraint vector and matrix for $\vec{k}$ search
 6:             Optimize over $\vec{k}$'s
 7:                 Enter inside minimization routine to find optimal $\vec{\gamma}$
 8:                     Subset $G_{\text{obs}}$ based on each iteration's choice for $\vec{k}$
 9:                     Compute X_mtx for current $\vec{k}$
10:                     Compute constraint vector and matrix for $\vec{\gamma}$ search
11:                     Optimize over $\vec{\gamma}$'s
12:                         Objective function: $\min_{\vec{\gamma}} \frac{\sum_{i=1}^{n}(\hat{G}-G)^2}{n}$
13:                     Return $\vec{\gamma}^*$
14:                 Objective function: $\min_{\vec{k}} \frac{\sum_{i=1}^{n}(\hat{G}-G)^2}{n}$
15:             Return $\vec{k}^*$
16:     Return $\vec{k}^*$, $\vec{\gamma}^*$
17:     Exit procedure
```