

B-part 20210108

January 21, 2021

1 B.1a

```
[1]: import csv
import sys
import matplotlib.pyplot as plt

def convert_to_float(s):
    """Convert a string to a float, or None if it is an empty string"""
    if s != '':
        return float(s)
    else:
        return None

def read_csv(filename):
    """
    Read the csv file containing child mortality data and returns it as a
    dictionary, as well as the years found in the file
    """
    with open(filename, 'r') as f:
        raw_data = list(csv.reader(f, delimiter=',', quotechar='"'))

    years = [int(v) for v in raw_data[0][4:]]
    data = raw_data[1:]

    child_mortality = {
        row[2]: [convert_to_float(v) for v in row[4:]]
        for row in data
    }

    return (years, child_mortality)

def get_child_mortality(country, year, child_mortality, years):
    """
    Return child mortality for a specific country in a specific year, both
    given as parameters.
    """
    return child_mortality[country][years.index(year)]
```

```

years, child_mortality = read_csv("child_mortality.csv")

years_plot = [1920, 1945, 1970, 1995, 2020]

fig, ax = plt.subplots(len(years_plot), 1, figsize=(6, 10))

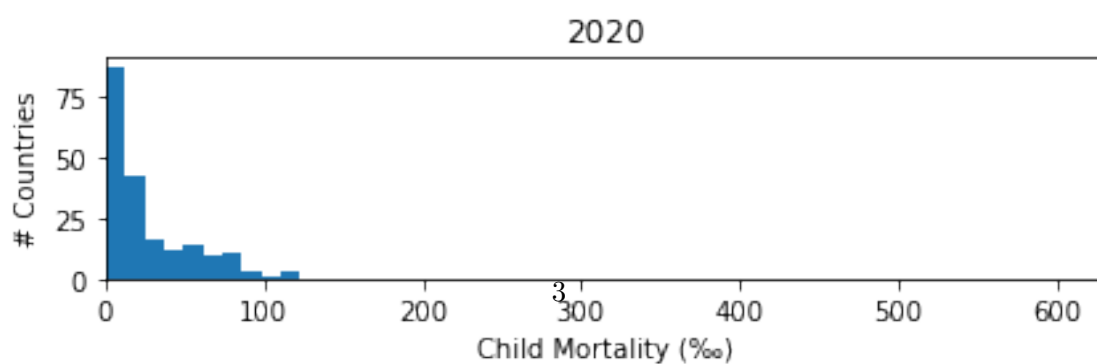
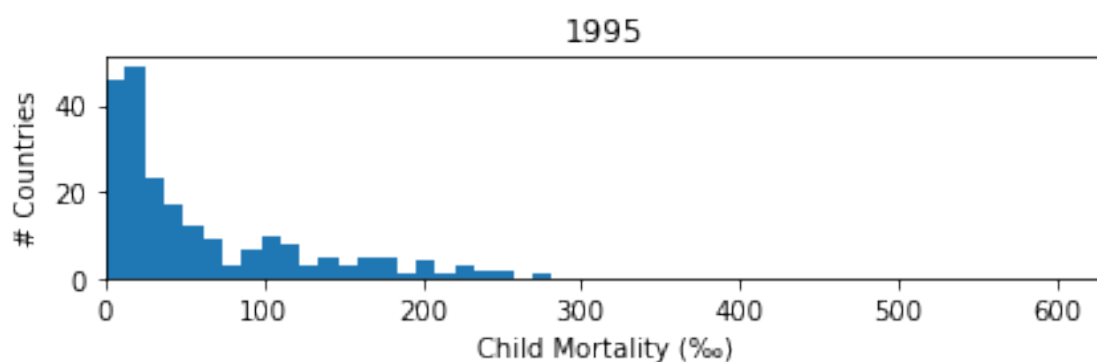
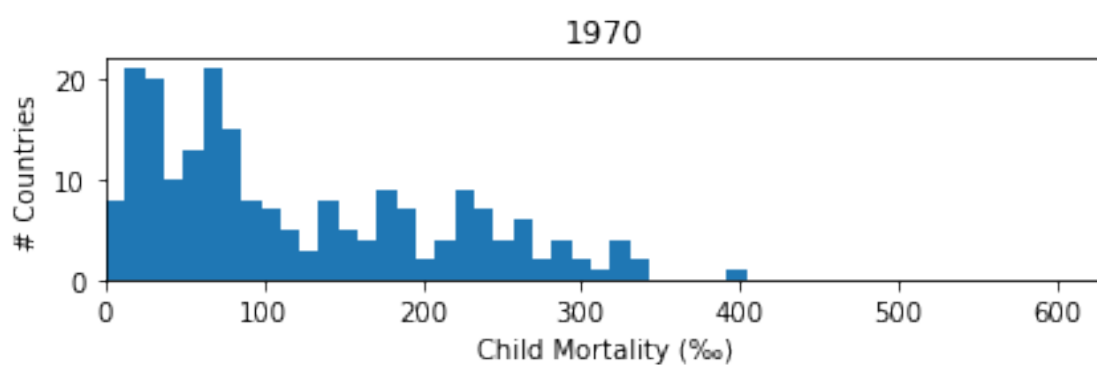
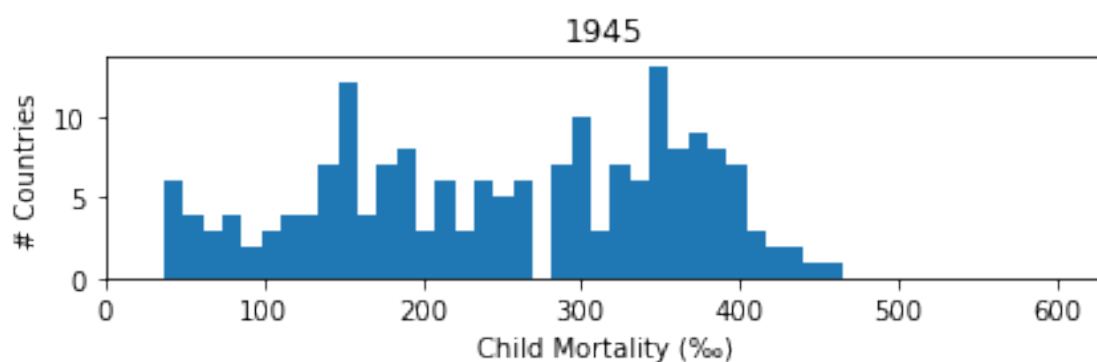
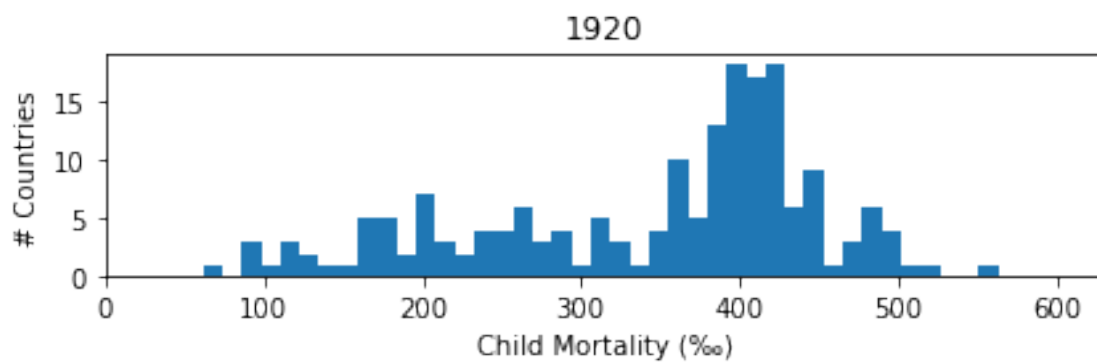
for i, year in enumerate(years_plot):
    ax[i].hist(
        [
            get_child_mortality(country, year, child_mortality, years)
            for country in child_mortality
            if get_child_mortality(country, year, child_mortality, years) !
            ↪= None
        ],
        bins=49, range=(0, 600),
    )
    ax[i].set_title(year)
    ax[i].set_xlim(0)

    ax[i].set_xlabel('Child Mortality (%)')
    ax[i].set_ylabel('# Countries')

fig.tight_layout() # Make sure labels don't overlap with the figures

plt.show()

```



2 B.1b

```
[2]: import csv
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

def convert_to_float(s):
    """Convert a string to a float, or None if it is the empty string"""
    if s != '':
        return float(s)
    else:
        return None

def read_csv(filename, start_col, key_col):
    """
    Read a csv file, where the data is identified by keys in column
    `key_col` and starts at column `start_col`. All values in the data
    will be parsed as floats.

    Parameters
    -----
        filename: str
        start_col: int
        key_col: int

    Return
    -----
        {str: [float]}
    """
    with open(filename, 'r') as f:
        raw_data = list(csv.reader(f, delimiter=',', quotechar='"'))

    years = [int(v) for v in raw_data[0][start_col:]]
    data = raw_data[1:]

    return (years, {
        row[key_col]: [convert_to_float(v) for v in row[start_col:]]
        for row in data
    })
```

```

def read_cont(filename):
    """
        Read a csv file containing continent information for all countries.
        All strings are converted to lower case.

        Parameters
        -----
            filename: str

        Return
        -----
            {str: str}
    """
    with open(filename, 'r') as f:
        raw_data = list(csv.reader(f, delimiter=',', quotechar='"'))

    return {row[4].lower(): row[1].lower() for row in raw_data}

# Child Mortality
years, chmo = read_csv("child_mortality.csv", 4, 2)

# TFR
_, tfr = read_csv("total_fertility_rate.csv", 4, 2)

# Pop
_, pop = read_csv("population.csv", 3, 0)

# Continents
continents = read_cont("country-continent.csv")

# Set nice colors for each continent depending on the continent code
colors = {
    'as': '#ff5872',
    'eu': '#ffe700',
    'an': '#ffffff',
    'af': '#00d5e9',
    'oc': '#ff5872',
    'na': '#7feb00',
    'sa': '#7feb00',
}

def get_data(data, country, year):
    """
        Return the value for a specific country and year contained in the
        data dictionary provided as parameter.
    """

```

```

Parameters
-----
    data: {str: [float]}
    country: str
    year: int

Return
-----
    float
"""
return data[country][years.index(year)]

def common_keys(data_list):
    """
    Returns a list of common keys contained in the list of dictionaries.

    Parameters
    -----
        data_list: [{str: T}]

    Return
    -----
        [str]
    """
    return [country
            for country in data_list[0]
            if all([country in data for data in data_list[1:]])]

countries = sorted(
    common_keys([chmo, tfr, pop, continents]),
    key=lambda c: pop[c], reverse=True)

years_plot = [1990, 2020, 2050]

fig, ax = plt.subplots(1, len(years_plot), sharey=True, figsize=(15, 5))

for i, year in enumerate(years_plot):
    x = np.array([get_data(chmo, country, year) for country in countries])
    y = np.array([get_data(tfr, country, year) for country in countries])
    s = np.array([get_data(pop, country, year) for country in countries])
    c = [colors[continents[country]] for country in countries]

```

```

s = 10**3.25*s/s.max()

ax[i].scatter(x, y, s=s, c=c, edgecolors='k')
ax[i].set_xscale('log')
ax[i].set_xlim(0.75, 500)
ax[i].set_title(year)
ax[i].set_xlabel('Child Mortality (%)')

legend_elements = [
    Line2D([0], [0], marker='o', color='w',
           label=continent,
           markerfacecolor=color, markeredgecolor='k', markersize=10)
    for (continent, color) in [
        ('Africa', '#00d5e9'),
        ('Asia, Oceania', '#ff5872'),
        ('Europe', '#ffe700'),
        ('Americas', '#7feb00'),
    ]
]

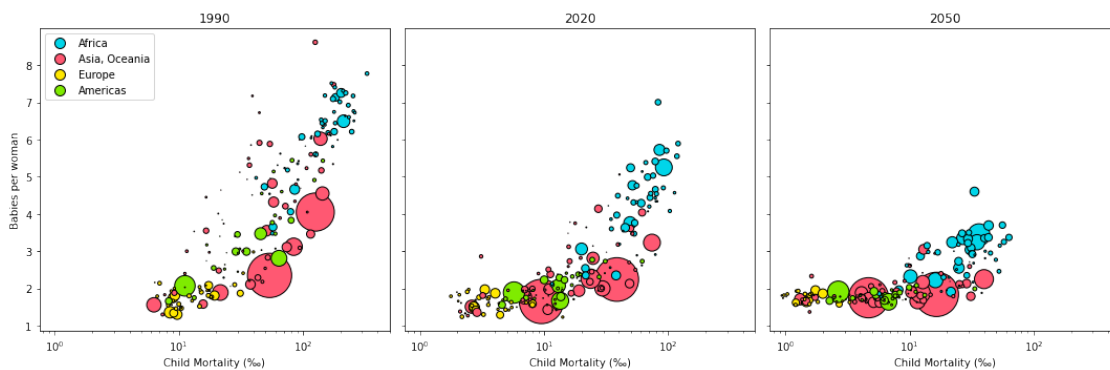
ax[0].legend(handles=legend_elements, loc=0)
ax[1].set_xlabel('Child Mortality (%)')
ax[0].set_ylabel('Babies per woman')

fig.tight_layout()

plt.savefig("chmotfr.jpg")

plt.show()

```



3 B.2a

```
[3]: class Grid:
    """
    Grid of cells

    Attributes
    -----
        grid: [[bool]]
        x_max: int
        y_max: int
    """
    def __init__(self, grid):
        """
        Build a grid from a list of lists of booleans.

        Parameters
        -----
            grid: [[bool]]
        """
        self.grid = [row.copy() for row in grid]
        self.x_max = len(grid)
        self.y_max = len(grid[0])

    def flip_cell(self, coord):
        """
        Invert the state located at `coord`.

        Parameters
        -----
            coord: (int, int)
        """
        x, y = coord
        self.grid[x][y] = not self.grid[x][y]

    def n_neighbors(self, coord):
        """
        Compute the number of neighbors surrounding the cell located
        at `coord`.

        Parameters
        -----
            coord: (int, int)

        Return
        -----
            int
        """
```



```

"""

x, y = coord
count = 0

for dx in [-1, 0, 1]:
    for dy in [-1, 0, 1]:
        if 0 <= x + dx < self.x_max and 0 <= y + dy < self.y_max:
            # Check we don't count the center cell
            if self.grid[x + dx][y + dy] and (dx != 0 or dy != 0):
                count += 1

return count

#Alt. with list-comprehension:
#return sum([
#     self.grid[x + dx][y + dy]
#     for dx in [-1, 0, 1] for dy in [-1, 0, 1]
#     if ((0 <= x + dx < self.x_max and 0 <= y + dy < self.y_max)
#         and (dx != 0 or dy != 0))])

def step(self):
    """
        Update the grid by one step.
    """

    new_grid = [row.copy() for row in self.grid]
    for x in range(self.x_max):
        for y in range(self.y_max):
            # Save the #neighbors in a variable to avoid computing it twice
            n_neighbors = self.n_neighbors((x, y))
            if self.grid[x][y]:
                new_grid[x][y] = (n_neighbors in [2, 3])
            else:
                new_grid[x][y] = (n_neighbors == 3)

    self.grid = new_grid

#Alt. with list-comprehension:
#self.grid = [
#     [(self.grid[x][y] and self.n_neighbors((x, y)) in [2, 3])
#     or (not self.grid[x][y] and self.n_neighbors((x, y)) == 3)
#     for y in range(self.y_max)]
#     for x in range(self.x_max)]

def update(self, n, snap_freq):
    """
        Update the grid by `n` steps and prints the grid every `snap_freq`
        steps.
    """

```

```

        Parameters
        -----
        n: int
        snap_freq: int
    """
    for i in range(n):
        self.step()
        if i % snap_freq == 0:
            self.print_snapshot()

    def print_snapshot(self):
        """
        Print the current state of the grid, using '.' for dead cells and
        '#' for living cells.
        """
        for row in self.grid:
            print(''.join(['#' if cell else '.' for cell in row]))
        print()

grid = Grid([[False for y in range(4)] for x in range(3)])

for coord in [(0, 2), (1, 2), (1, 0), (2, 2)]:
    grid.flip_cell(coord)

grid.print_snapshot()
grid.update(4, 1)

```

```

..#.
#.#.
..#.

.#..
..##
.#..

..#.
.##.
..#.

.##.
.###
.##.

.#.#
#..#

```

.#.#

4 B.2b

One solution would be to use a list containing two lists of integers:

- the list at index 0 would contain the number of neighbors required for a dead cell to become alive
- the list at index 1 would contain the number of neighbors required for a living cell to stay alive.

The `__init__` function would then take an extra argument, that we could call `rules` and store this list in an attribute with the same name. In `step`, after computing the number of neighbors, we can check if this value is in the first or the second list, depending of the current state of the cell.

In this way, we can represent any set of rules. For instance the default rules presented in the problem description would be `[[3], [2, 3]]`. The rules presented in this question would be `[[3, 4, 5], [2, 5, 7]]`.

5 B.2c

1. In the code provided before, we would need to change the following things:
 - When computing the size of the grid in `__init__`, we would need to search for the coordinates with minimum and maximum index, i.e. we would need two extra attributes `x_min` and `y_min`
 - When flipping a cell, we would need to first check that the cell exists in the set. If it does, we should remove it, if it does not exist, we should add it.
 - When computing the number of neighbors, instead of using the values from the grid, we can use the values returned by `(x + dx, y + dy)` in `grid` for each potential neighbors and sum them up as we did before.
 - In `step`, instead of looping through every coordinates in the grid, we need to first generate a list of all the cells and their potential neighbors (i.e. not only do we need to iterate over the coordinates in the set, but also over the coordinates where cells can potentially become alive), and then filter this new list according the the neighboring rules. From this filtered list we can generate a new set, the new updated state of the grid.
 - Finally, in `print_snapshot`, we would need to go from `y_min` to `y_max` and from `x_min` to `x_max` and print `#` for all coordinates that are in the set, else `..`
2. The code presented in this document was written with this question in mind already, so it might not be a very fair example. However, some things to think about when writing code that we want to be general are:
 - As much as possible, use small, reusable functions and methods and use them as much as possible, instead of accessing the attributes directly. In this way, if you decide to change the design of your class and change some attributes, it should be enough to redefine only a handful of methods.

- Use variables whenever you need to store a value that you will need to use several times. If you need to change this value later, you only need to change it once.

[]: