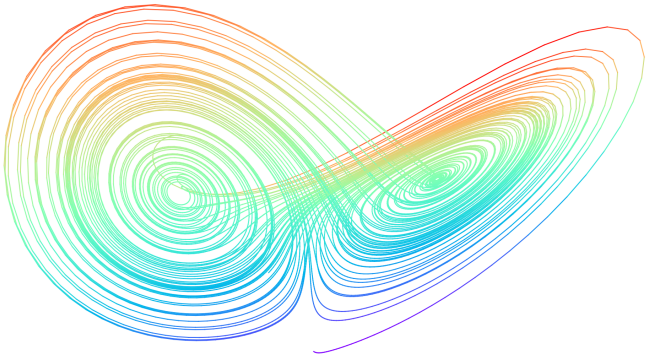
Modeling of Dynamical Systems

Lab 5 - Everything, everywhere, all at once



Alberto Mario Castillo

September 2023

Contents

List of Figures

1 Trace-Determinant

1: Trace

For any 2x2 matrix, the trace is the sum of the top-left and bottom-right elements.

The trace of a matrix is often used to determine the stability of equilibrium points. Specifically, the trace provides information about the sum of the eigenvalues of the matrix. If the trace is negative, it suggests stability, while a positive trace suggests instability.

2: Determinant

A scalar value that can be computed by multiplying the top-left element by the bottom-right element, and subtract the product of the top-right element and the bottom-left element.

For the matrix: $\mathrm{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$

The trace of A (tr A) and its determinant (det A) are:

- $a + d$
- $ad - bc$

The characteristic equation of the system is:

$$\lambda^2 - (a+d)\lambda + (ad - bc) = 0$$

Or just:

$$\lambda^2 - (\mathrm{tr}A)\lambda + \det A = 0$$

And by Bhaskara we know that:

$$\lambda_{\pm} = \frac{1}{2}(\mathrm{tr}A \pm \sqrt{(\mathrm{tr}A)^2 - 4\det A})$$

In order to explore some relationships:

1: Trace, determinant, eigenvalues

For the following system, calculate the trace, the determinant, add the eigenvalues, and multiply the eigenvalues:

$$A = \begin{pmatrix} 0 & 2 \\ -7 & 0 \end{pmatrix} \quad A = \begin{pmatrix} -7 & 3 \\ 1 & 6 \end{pmatrix} \quad A = \begin{pmatrix} -1 & -3 \\ 3 & -1 \end{pmatrix}$$

What conclusion can you draw from your results?

2 Hands-on

Find matrices satisfying:

- $(trA) > 0$ and $D > 0$
- $(trA) < 0$ and $D > 0$
- $(trA) \neq 0$ and $D = 0$
- $(trA)^2 - 4D = 0$
- $(trA)^2 - 4D < 0$

- $(trA)^2 - 4D > 0$
- For $(trA)^2 - 4D < 0$:
  - $(trA) < 0$
  - $(trA) > 0$
  - $(trA) = 0$
- For $(trA)^2 - 4D > 0$:
  - $(trA) < 0$
  - $(trA) > 0$
  - $(trA) = 0$

Calculate the eigenvalues, plot the phase space, and use your results to fill a table like the one below:

Case	Matrix	λ_1	λ_2	$\lambda_{Real} = (trA)/2$?	Plot	Classification
$(trA)^2 - 4D = 0$	...	...	...	...	...	...

Upload your single-page answer to e-aulas.

3 Dash Applications

Dash apps are web applications constructed using the Dash Python framework, which facilitates the development of interactive and data-centric web-based tools. These apps are engineered to provide an immersive and responsive interface for users to interact with and explore datasets, conduct data analysis, and visualize results.

3.1 Utils file

Before we dive into dash concepts, create a python file and name it `app`. This file contains functions to perform basic data preparation:

1. Retrieving latitude and longitude of each country:

```
    def get_country_coords(country_name, output_as='boundingbox'):
    """
    Get the coordinates of a country by name.
    : param: country_name: name of the country to retrieve coords
    : param: output_as : 'str to chose from 'boundingbox' or 'center'.
         - 'boundingbox' for [latmin, latmax, lonmin, lonmax]
         - 'center' for [latcenter, loncenter]
    """

    # Create url
    url = '{0}{1}{2}'.format('http://nominatim.openstreetmap.org/search?country=',
                             country_name, '&format=json&polygon=0')
    response = requests.get(url).json()[0]

    # parse response to list
    if output_as == 'boundingbox':
        lst = response[output_as]
        output = [float(i) for i in lst]
    if output_as == 'center':
        lst = [response.get(key) for key in ['lon', 'lat']]
        output = [float(i) for i in lst]
    return output
```

2. Formatting dates and grouping data:

```
    def raw_processor(data_raw: pd.DataFrame):
    data_raw['Training Start'] = pd.to_datetime(data_raw['Training Start'])
    data_raw['Training Start_date'] = data_raw['Training Start'].dt.date
    data_raw.sort_values(by='Training Start_date', ascending=True)

    data = (data_raw.groupby(['Country',
      'Vertical',
      'Language',
      'Status',
      'Training Start_date'])[['Demand',
        'Offers Accepted',
        'Hired',
        'Applicants',
        'Requisition ID']].agg({'Demand': 'sum',
      'Offers Accepted': 'sum',
      'Hired': 'sum',
      'Applicants': 'sum',
      'Requisition ID': 'count'}).
    reset_index().sort_values(by='Demand', ascending=False))

    countries_dict = {}
    for country in data.Country.unique():
        countries_dict[country] = get_country_coords(country, 'center')

    data.rename(columns = {'Requisition ID': 'Requisition_count'}, inplace=True)
    data['country_lat'] = data['Country'].apply(lambda country_: countries_dict[
        country_][1])
    data['country_lon'] = data['Country'].apply(lambda country_: countries_dict[
        country_][0])

    return data
```

3.2 Basic components

3.2.1 Container

In the context of web development and user interface design, a "container" refers to a fundamental layout element that is used to group and organize other elements or content on a web page. Containers help structure the layout of a webpage and provide a way to control the placement and alignment of content.

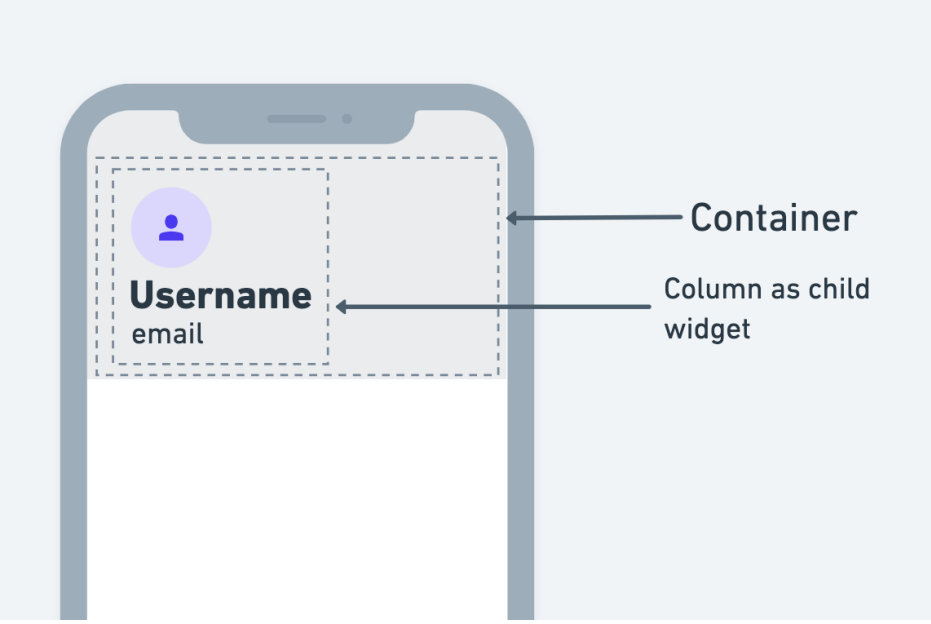


Figure 1: Container. Source: Flutter Website

3.2.2 Card

A UI component used to display content in a visually appealing and structured manner. Cards are a common design element in web applications, and they typically consist of a container with various elements like images, text, buttons, and other components that present information or functionality to the user. Cards are used to organize and present content in a structured and organized way. They often have a header, body, and footer section that can contain different types of content.

- **Header**: Cards can have a header section that typically contains a title or a heading, providing a brief description or title for the card's content.
- **Body**: The body of a card is where you place the main content or information you want to display. This can include text, images, charts, or any other components that convey the intended information.
- **Footer**: Some cards have a footer section that can include additional information, buttons, or links related to the content in the card.

An example of a card:

```
  top_country = dbc.Card(
    dbc.CardBody([
        html.I(id = 'requisition-country-card', className="bi text-success h1"),
        html.H5([html.I(className="bi bi-globe me-2"),
                 html.I(id = 'requisition-country-name-card')],
                className="text-nowrap"),
        html.P("By number of openings", className="text-muted", style={"font-size": "10
            px"}),
    ], className="border-start border-success border-5"
    ), className="text-center m-4"
)
```

3.2.3 Filters

Filters are interactive input elements that allow users to filter and refine data displayed in a Dash web application. These filters are typically used to control what data is shown in charts, graphs, tables, or other visualizations on

the web page. Filters provide a way for users to customize the data view to meet their specific needs. Examples of filters are:

1. **Date Picker Filter**: is used to select a single date or date range and filter data based on it.

```
    dcc.DatePickerSingle(
                            id='start-date-picker',
                            date=df_mds['Training_Start_date'].min(),
                            display_format='MMM Do, YY',
                            with_portal = True,
                            style={'marginRight': '10px'}
                        )
```

2. **Dropdown Filters**: provide users with a list of options to choose from. Common use cases for dropdown filters include filtering data by categories such as countries, verticals, languages, and status. Users can select one or multiple options from the dropdown lists to filter data based on their selections.

```
    dcc.Dropdown(
            id='country-dropdown',
            options=[{'label': country, 'value': country} for country in df_mds['
                Country'].unique()],
            multi=True,
            value=None,
            style={'marginBottom': '10px'},
            placeholder='Country'
        )
```

3. **Numeric Range Filters**: These filters allow users to specify a range of numeric values.

4. **Text Search Filters**: enable users to search for specific keywords or phrases within a dataset. This is useful when filtering textual data, such as documents or articles.

5. **Checkbox Filters**: are used to toggle specific categories or options on and off. They are particularly useful for binary or multi-choice filters.

3.2.4 Figures

To plot figures we use the `Graph` component from `dash core components`. It allows to create and customize various types of data visualizations, such as line charts, bar charts, scatter plots, and more. In the code we'll be working with, the figures are part of the layout. To create a graph we'll use something like this:

```
    dcc.Graph(id='country-distribution', style=CONTENT_STYLE, figure={'layout': {
      'height': 800,
      'width': 800
  }
  }, config={'displayModeBar': False}
          )
```

3.2.5 Dahsboard layout

A layout refers to the arrangement and organization of visual elements on a web page. It defines the structure and presentation of a webpage, including how different components, such as text, images, forms, navigation menus, and other user interface elements, are positioned and styled.

A Dash layout is typically constructed using Dash's own high-level components provided by the `dash_html_components` and `dash_core_components` libraries. Here's a brief overview of these components and how a Dash layout is structured:

- `dash_html_components`: This library provides a set of HTML-like elements that you can use to structure the layout of your web application. Some common elements include html.Div, html.H1, html.P, html.Table, and more. These elements are used to create the overall structure and organization of the web page.
- `dash_core_components`: This library provides more advanced interactive components, such as graphs (dcc.Graph), dropdowns (dcc.Dropdown), sliders (dcc.Slider), date pickers (dcc.DatePickerSingle), and more. These components enable interactivity and data visualization in your web application.

3.3 Updating components

In Dash, components are updated dynamically in response to user interactions or changes in data by using callback functions.

3.3.1 Callbacks

Callbacks in Dash are Python functions that are triggered by user input or by changes in the application's underlying data. These functions can update the properties of one or more components, effectively changing their content, style, or behaviour. To define a callback an update components you need to:

- **Define the callbacks**: You start by defining one or more callback functions using the `@app.callback` decorator, where `app` is your Dash application instance. Each callback function specifies the **input** components and **output** components it will work with. Input components are the ones that trigger the callback when their values change, and output components are the ones whose properties will be updated based on the callback's logic. For instance:

```
@app.callback(
[Output('inflow-bar', 'figure')],
[Input('start-date-picker', 'date'),
  Input('end-date-picker', 'date'),
  Input('country-dropdown', 'value'),
  Input('vertical-dropdown', 'value'),
  Input('language-dropdown', 'value'),
  Input('status-dropdown', 'value')]
)
```

- **Callback Function Logic**: In the callback function, you define the logic that will be executed when the input components change. You can access the values of the input components using the function arguments, and you can use these values to calculate the new properties of the output components. For example:

```
  def update_barplot(start_date, end_date, selected_countries,
      selected_verticals, selected_languages,
  selected_statuses):
  start_date = pd.to_datetime(start_date)
  end_date = pd.to_datetime(end_date)

  filtered_df = df_mds[
    (df_mds['Training_Start_date'] >= start_date) & (df_mds['
        Training_Start_date'] <= end_date)
  ]
  if selected_countries is not None and len(selected_countries) > 0:
  filtered_df = filtered_df[filtered_df['Country'].isin(selected_countries)]
  if selected_verticals is not None and len(selected_verticals) > 0:
  filtered_df = filtered_df[filtered_df['Vertical'].isin(selected_verticals)]
  if selected_languages is not None and len(selected_languages) > 0:
  filtered_df = filtered_df[filtered_df['Language'].isin(selected_languages)]
  if selected_statuses is not None and len(selected_statuses) > 0:
```

```
        filtered_df = filtered_df[filtered_df['Status'].isin(selected_statuses)]

        grouped_df = filtered_df.groupby('Training_Start_date')[['Demand', '
            Offers_Accepted',
             'Hired']].agg({'Demand': 'sum',
          'Offers_Accepted': 'sum',
          'Hired': 'sum'}).reset_index()

        barchart = make_subplots(rows=1, cols=1)
        barchart.add_trace(go.Bar(x=grouped_df['Training_Start_date'], y=grouped_df['
            Demand'], name='Demand'))
        barchart.add_trace(go.Bar(x=grouped_df['Training_Start_date'], y=grouped_df['
            Offers_Accepted'], name='Accepted Offers'))
        barchart.add_trace(go.Bar(x=grouped_df['Training_Start_date'], y=grouped_df['
            Hired'], name='Hired'))

        barchart.update_layout(title='Demand, Offers accepted, and Hires over time',
        xaxis_title='Training Start Date',
        yaxis_title='Number',
        barmode='group')

    return [barchart]
```

3.4 A working example

Create a python file and name it `app`. Using a terminal, run: `pip install -r requirements.txt`, using the `requirements.txt` file provided.
Let's begin importing the required libraries:

```
import dash_bootstrap_components as dbc
import utils as ut
import os
import pandas as pd
import plotly.graph_objs as go
from dash import dash, dcc, html, Output, Input
from plotly.subplots import make_subplots
```

An instance for the dash app, can be easily created with just:

```
app = dash.Dash(__name__, external_stylesheets=[dbc.themes.SPACELAB, dbc.icons.
    BOOTSTRAP, '/assets/custom.css'])
```

Now it is time to define styles and read our logo:

```
SIDEBAR_STYLE = {
  "position": "fixed",
  "top": 0,
  "left": 0,
  "bottom": 50,
  "width": "17rem",
  "padding": "2rem 1rem",
  "background-color": "#FFFFFF",
  "color": "#24245c"
}

SIDEBAR_FILTER_STYLE = {
```

```
    "position": "fixed",
    "top": 200,
    "left": 0,
    "bottom": 50,
    "width": "17rem",
    "padding": "2rem 1rem",
    "background-color": "#FFFFFF",
    "color": "#24245c"
  }

  CONTENT_STYLE = {
    "margin-left": "19rem",
    "margin-right": "2rem",
    "padding": "2rem 1rem",
    "color": "#24245c"
  }

  image_path = 'assets/lorenz_attractor_rainbow.png'
```

Read the data and preprocess it:

```
  df_mds = pd.read_csv('./exercise_to_bq.csv')
  df_mds = ut.raw_processor(df_mds)
  df_mds.rename(columns={'Training Start_date': 'Training_Start_date',
    'Offers Accepted': 'Offers_Accepted'}, inplace=True)
```

Let's add some cards:

```
req_count_card = dbc.Card(
dbc.CardBody([
  html.I(id = 'requisition-count-card', className="bi text-success h1"),
  html.H5([html.I(className="bi bi-person me-2"), "Number of requisitions"],
  className="text-nowrap"),
  html.P("Total", className="text-muted", style={"font-size": "10px"}),
], className="border-start border-success border-5"
), className="text-center m-4"
)

req_avg_size_card = dbc.Card(
dbc.CardBody([
  html.I(id = 'avg-requisition-size-card', className="bi text-success h1"),
  html.H5([html.I(className="bi bi-rulers me-3"), 'Requisition average size'],
      className="text-nowrap"),
  html.P("Openings per requisition", className="text-muted", style={"font-size": "10px
      "}),
], className="border-start border-success border-5"
), className="text-center m-4"
)

top_language = dbc.Card(
dbc.CardBody([
  html.I(id = 'requisition-language-card', className="bi text-success h1"),
  html.H5([html.I(className="bi bi-chat-dots-fill me-2"),
    html.I(id = 'requisition-language-name-card')],
  className="text-nowrap"),
  html.P("By number of openings", className="text-muted", style={"font-size": "10px"})
      ,
], className="border-start border-success border-5"
), className="text-center m-4"
)

top_vertical = dbc.Card(
dbc.CardBody([
  html.I(id = 'requisition-vertical-card', className="bi text-success h1"),
  html.H5([html.I(className="bi bi-building-up me-2"),
    html.I(id='requisition-vertical-name-card')],
  className="text-nowrap"),
  html.P("By number of openings", className="text-muted", style={"font-size": "10px"})
      ,
], className="border-start border-success border-5"
), className="text-center m-4"
)

top_country = dbc.Card(
dbc.CardBody([
  html.I(id = 'requisition-country-card', className="bi text-success h1"),
  html.H5([html.I(className="bi bi-globe me-2"),
    html.I(id = 'requisition-country-name-card')],
  className="text-nowrap"),
  html.P("By number of openings", className="text-muted", style={"font-size": "10px"})
      ,
], className="border-start border-success border-5"
```

```
50    ), className="text-center m-4"
51    )
```

Now, add the filters to the side bar:

```
filters_side_bar = html.Div([
  html.Div(
  children = [
    html.Img(src=image_path, alt='Logo Dynamical Systems', className='header-image',
        style=SIDEBAR_STYLE),
    html.Div([
      html.H4('Filters', style={"color": "#24245c", 'padding-top': '40px'}),
      html.Div([
        html.Label('Start Date', style={'marginRight': '5px'}),
        dcc.DatePickerSingle(
        id='start-date-picker',
        date=df_mds['Training_Start_date'].min(),
        display_format='MMM Do, YY',
        with_portal = True,
        style={'marginRight': '10px'}
        ),

        html.Label('End date', style={'marginRight': '12px'}),
        dcc.DatePickerSingle(
        id='end-date-picker',
        date= df_mds['Training_Start_date'].max(),
        display_format='MMM Do, YY',
        with_portal = True
        ),
      ], style={'display': 'block', 'alignItems': 'center', 'marginBottom': '10px'}),
      dcc.Dropdown(
      id='country-dropdown',
      options=[{'label': country, 'value': country} for country in df_mds['Country'].
          unique()],
      multi=True,
      value=None,
      style={'marginBottom': '10px'},
      placeholder='Country'
      ),
      dcc.Dropdown(
      id='vertical-dropdown',
      options=[{'label': vertical, 'value': vertical} for vertical in df_mds['Vertical
          '].unique()],
      multi=True,
      value=None,
      style={'marginBottom': '10px'},
      placeholder='Vertical'
      ),
      dcc.Dropdown(
      id='language-dropdown',
      options=[{'label': language, 'value': language} for language in df_mds['Language
          '].unique()],
      multi=True,
      value=None,
      style={'marginBottom': '10px'},
      placeholder='Language'
      ),
      dcc.Dropdown(
      id='status-dropdown',
```

```
        options=[{'label': pos_status, 'value': pos_status} for pos_status in df_mds['
            Status'].unique()],
        multi=True,
        value=None,
        style={'marginBottom': '10px'},
        placeholder='Status'
        )])
    ],
    )
], className='filters', style=SIDEBAR_FILTER_STYLE),
```

Define the layout:

```
# Create a layout for the dashboard
app.layout = html.Div([
  html.Div([
    html.H1('Sample Dashboard', style=CONTENT_STYLE)]),
  html.Div(filters_side_bar),
  html.Div([dbc.Container( # A container for the cards
    [dbc.Row([dbc.Col(top_country)]),
      dbc.Row([dbc.Col([req_count_card, top_language]),
        dbc.Col([req_avg_size_card, top_vertical]),
      ]
      )],
    fluid=True,
    )
  ], className='row', style=CONTENT_STYLE),
  dcc.Graph(id='inflow-bar', style=CONTENT_STYLE, figure={'layout': {
      'height': 800,
      'width': 800
    }
  }, config={'displayModeBar': False}
  ),
  dcc.Graph(id='country-distribution', style=CONTENT_STYLE, figure={'layout': {
      'height': 800,
      'width': 800
    }
  }, config={'displayModeBar': False}
  ),
])
```

Set the callbacks:

```
@app.callback(
[Output('requisition-count-card', 'children'),
  Output('avg-requisition-size-card', 'children'),
  Output('requisition-language-card', 'children'),
  Output('requisition-language-name-card', 'children'),
  Output('requisition-vertical-card', 'children'),
  Output('requisition-vertical-name-card', 'children'),
  Output('requisition-country-card', 'children'),
  Output('requisition-country-name-card', 'children')],
[Input('start-date-picker', 'date'),
  Input('end-date-picker', 'date'),
  Input('country-dropdown', 'value'),
  Input('vertical-dropdown', 'value'),
  Input('language-dropdown', 'value'),
  Input('status-dropdown', 'value')]
)
def update_top_cards(start_date, end_date, selected_countries, selected_verticals,
    selected_languages,
selected_statuses):

start_date = pd.to_datetime(start_date)
end_date = pd.to_datetime(end_date)

filtered_df = df_mds[
  (df_mds['Training_Start_date'] >= start_date) & (df_mds['Training_Start_date'] <=
      end_date)
]
if selected_countries is not None and len(selected_countries) > 0:
filtered_df = filtered_df[filtered_df['Country'].isin(selected_countries)]
if selected_verticals is not None and len(selected_verticals) > 0:
filtered_df = filtered_df[filtered_df['Vertical'].isin(selected_verticals)]
if selected_languages is not None and len(selected_languages) > 0:
filtered_df = filtered_df[filtered_df['Language'].isin(selected_languages)]
if selected_statuses is not None and len(selected_statuses) > 0:
filtered_df = filtered_df[filtered_df['Status'].isin(selected_statuses)]

requisition_count_sum = filtered_df['Requisition_count'].sum() if not filtered_df.
    empty else 0
demand_average = filtered_df['Demand'].mean() if not filtered_df.empty else 0

language_top = filtered_df.groupby('Language')['Demand'].sum().reset_index().
    sort_values(by='Demand', ascending=False)
language_top_name = language_top.iloc[0].to_list()[0]
language_top_name = f'{language_top_name} is the top language'
language_top_number = language_top.iloc[0].to_list()[1]

vertical_top = filtered_df.groupby('Vertical')['Demand'].sum().reset_index().
    sort_values(by='Demand', ascending=False)
vertical_top_name = vertical_top.iloc[0].to_list()[0]
vertical_top_name = f'{vertical_top_name} is the top vertical'
vertical_top_number = vertical_top.iloc[0].to_list()[1]


country_top = filtered_df.groupby('Country')['Demand'].sum().reset_index().sort_values
```

```
    (by='Demand', ascending=False)
country_top_name = country_top.iloc[0].to_list()[0]
country_top_name = f'{country_top_name} is the top country'
country_top_number = country_top.iloc[0].to_list()[1]

return [requisition_count_sum, round(demand_average,1), language_top_number,
    language_top_name,
  vertical_top_number, vertical_top_name, country_top_number,country_top_name]


@app.callback(
[Output('inflow-bar', 'figure')],
[Input('start-date-picker', 'date'),
  Input('end-date-picker', 'date'),
  Input('country-dropdown', 'value'),
  Input('vertical-dropdown', 'value'),
  Input('language-dropdown', 'value'),
  Input('status-dropdown', 'value')]
)

def update_barplot(start_date, end_date, selected_countries, selected_verticals,
    selected_languages,
selected_statuses):
start_date = pd.to_datetime(start_date)
end_date = pd.to_datetime(end_date)

filtered_df = df_mds[
  (df_mds['Training_Start_date'] >= start_date) & (df_mds['Training_Start_date'] <=
      end_date)
]
if selected_countries is not None and len(selected_countries) > 0:
filtered_df = filtered_df[filtered_df['Country'].isin(selected_countries)]
if selected_verticals is not None and len(selected_verticals) > 0:
filtered_df = filtered_df[filtered_df['Vertical'].isin(selected_verticals)]
if selected_languages is not None and len(selected_languages) > 0:
filtered_df = filtered_df[filtered_df['Language'].isin(selected_languages)]
if selected_statuses is not None and len(selected_statuses) > 0:
filtered_df = filtered_df[filtered_df['Status'].isin(selected_statuses)]

grouped_df = filtered_df.groupby('Training_Start_date')[['Demand', 'Offers_Accepted',
    'Hired']].agg({'Demand': 'sum',
  'Offers_Accepted': 'sum',
  'Hired': 'sum'}).reset_index()

barchart = make_subplots(rows=1, cols=1)
barchart.add_trace(go.Bar(x=grouped_df['Training_Start_date'], y=grouped_df['Demand'],
     name='Demand'))
barchart.add_trace(go.Bar(x=grouped_df['Training_Start_date'], y=grouped_df['
    Offers_Accepted'], name='Accepted Offers'))
barchart.add_trace(go.Bar(x=grouped_df['Training_Start_date'], y=grouped_df['Hired'],
    name='Hired'))

barchart.update_layout(title='Demand, Offers accepted, and Hires over time',
xaxis_title='Training Start Date',
yaxis_title='Number',
barmode='group')
```

```
    return [barchart]


@app.callback(
[Output('country-distribution', 'figure')],
[Input('start-date-picker', 'date'),
  Input('end-date-picker', 'date'),
  Input('vertical-dropdown', 'value'),
  Input('language-dropdown', 'value'),
  Input('status-dropdown', 'value')]
)


def update_boxplot(start_date, end_date, selected_verticals, selected_languages,
selected_statuses):
start_date = pd.to_datetime(start_date)
end_date = pd.to_datetime(end_date)

filtered_df = df_mds[
  (df_mds['Training_Start_date'] >= start_date) & (df_mds['Training_Start_date'] <=
      end_date)
]
if selected_verticals is not None and len(selected_verticals) > 0:
filtered_df = filtered_df[filtered_df['Vertical'].isin(selected_verticals)]
if selected_languages is not None and len(selected_languages) > 0:
filtered_df = filtered_df[filtered_df['Language'].isin(selected_languages)]
if selected_statuses is not None and len(selected_statuses) > 0:
filtered_df = filtered_df[filtered_df['Status'].isin(selected_statuses)]

grouped_df = filtered_df.groupby(['Training_Start_date', 'Country'])[['Demand', '
    Offers_Accepted',
     'Hired', 'Applicants']].agg({'Demand': 'sum',
  'Offers_Accepted': 'sum',
  'Hired': 'sum',
  'Applicants': 'sum'}).reset_index()

bxplot = go.Figure()
bxplot.add_trace(go.Box(x = grouped_df['Country'], y = grouped_df['Demand'], name='
    Demand'))
bxplot.add_trace(go.Box(x=grouped_df['Country'], y=grouped_df['Applicants'], name='
    Applicants'))
bxplot.add_trace(go.Box(x = grouped_df['Country'], y = grouped_df['Offers_Accepted'],
    name='Offered'))
bxplot.add_trace(go.Box(x=grouped_df['Country'], y=grouped_df['Hired'], name='Hired'))
bxplot.update_layout(title='Country distribution over time', legend=dict(y=0.5,
    traceorder='reversed'))
bxplot.update_yaxes(title="Number")

return [bxplot]
```

Finally, run your application:

```
if __name__ == '__main__':
  app.run(port=int(os.environ.get("PORT", 8989)), host='0.0.0.0', debug=False)
```

4 Everything, everywhere, all at once

4.1 User stories

User stories are concise, user-centered descriptions of a software feature or functionality that capture the needs and expectations of end users. They are a fundamental part of Agile and Scrum methodologies for software development. A user story typically follows a specific format, often known as the 'Three Cs' format:

1. **Card**: The user story is typically written on a physical or digital card to make it tangible and easily manageable.
2. **Conversation**: User stories encourage conversation and collaboration between the development team and stakeholders, fostering a deeper understanding of requirements.So if you have questions on the user stories provided below, do not hesitate and ask.
3. **Confirmation**: User stories include acceptance criteria or conditions of satisfaction that define when the story is considered complete.

In general, a user story includes the following elements:

- **Role**: The type of user or stakeholder who will benefit from the feature. It is written: 'As <role>'
- **Goal**: The objective or desired outcome the user wants to achieve. Usually goes like: 'I want to'
- **Benefit or need**: The value, need, or benefit the user will gain from the feature. Expressing: 'what is it for'

As a suggestion, when writing user stories, keep in mind the INVEST approach:

- **Independent**: User stories should be independent of each other. This means that each story should be self-contained and not dependent on the completion of other stories. Independence allows for flexibility in prioritization and execution.
- **Negotiable**: User stories should be open to negotiation and discussion. They are not rigid contracts but rather a starting point for conversations between developers and stakeholders. Teams should be open to refining and adapting stories as they learn more about the requirements.
- **Valuable**: Each user story should deliver value to the end user or customer. It should focus on solving a real problem or meeting a genuine need. The value of a story should be evident to the stakeholders.
- **Estimable**: User stories should be estimable, meaning that the team should be able to estimate the effort required to implement them. This helps with planning and prioritization. If a story is too vague or complex to estimate, it may need to be broken down or clarified.
- **Small**: User stories should be small in scope. They should represent bite-sized pieces of functionality that can be completed within a single iteration (e.g., a sprint in Scrum). Smaller stories are easier to understand, estimate, and complete.
- **Testable**: User stories should have clear acceptance criteria that define how the story will be tested and verified. This ensures that everyone understands what "done" means for the story and provides a basis for quality assurance.

Source: *Rubin, Kenneth S. Essential Scrum: A practical guide to the most popular Agile process. Addison-Wesley, 2012.*

4.2 Deliverable

Given the following user stories:

1. **As a user, I want to input a matrix so calculations can be performed**.
   Accept matrix input from the user, allowing them to specify the coefficients of a system of equations or a mathematical matrix.

2. **As a user, I want to get the eigenvalues and eigenvectors of the matrix I provided so I can have an idea of the system's behaviour**
   Calculate and display the eigenvalues and eigenvectors of the user-provided matrix, providing insights into the stability and behavior of the system.
3. **As a user, I want the application to classify the system so I can save some time**
   Automatically classify or categorize the system based on the properties of the matrix (e.g., stable, unstable, saddle point, etc.), providing a quick assessment of its behaviour.
4. **As a user, I want the option to plot the phase space of the system with the corresponding classification around the equilibrium point so I know what the behaviour is**
   Generate phase space plots for the system, including classification information (e.g., limit cycles, attractors), to visually represent the system's behaviour around equilibrium points.
5. **As a user, I need to know the canonical form of the system I provided, so I can better understand the system's classification**
   Determine and display the canonical form or representation of the user-provided system, aiding users in understanding its mathematical structure.
6. **As a user, I need to have the general solution for the system I provided.**
   Calculate and present the general solution of the system of equations represented by the provided matrix, allowing users to analyze and work with the mathematical representation of the system.

Your team should follow these steps:

- Assign specific user stories to each team member to ensure clear ownership and responsibility.
- Estimate the development effort required for each user story, providing a basis for scheduling and resource allocation.
- Communicate with the professor to inform him about the assignment of user stories.
- Create a dedicated repository for your project to manage version control and collaboration.
- Share the repository with YueCastillo, granting access for collaboration.
- Each team member should create a separate branch within the repository to work on their assigned user stories, promoting parallel development and minimizing conflicts.
- Develop and thoroughly test each functionality associated with the user stories.
- Once all functionalities have been successfully developed and tested, perform a final merge of the individual branches into the main project branch, consolidating the work.
- Prepare to present your progress in the next session. Be ready to discuss completed tasks, challenges, and any adjustments needed.
- Publish your web application, making it accessible to users and stakeholders. This step may involve deployment to a hosting platform or server.

By following these steps, you'll efficiently manage the development process, collaborate effectively, keep the professor informed, and ensure a smooth transition from individual contributions to a fully functional web application.