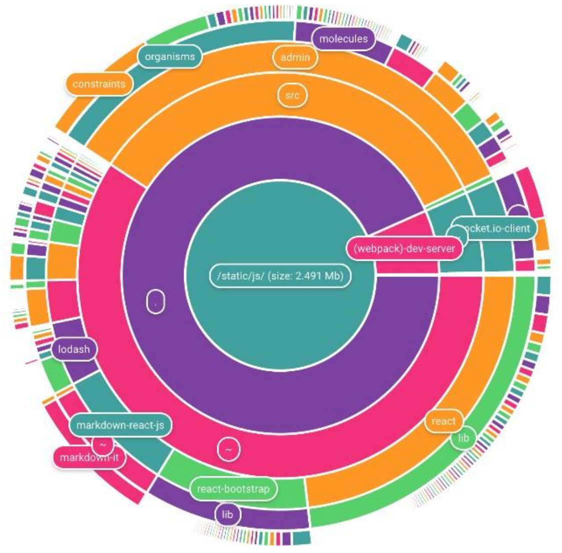
Research Track II
Data visualization

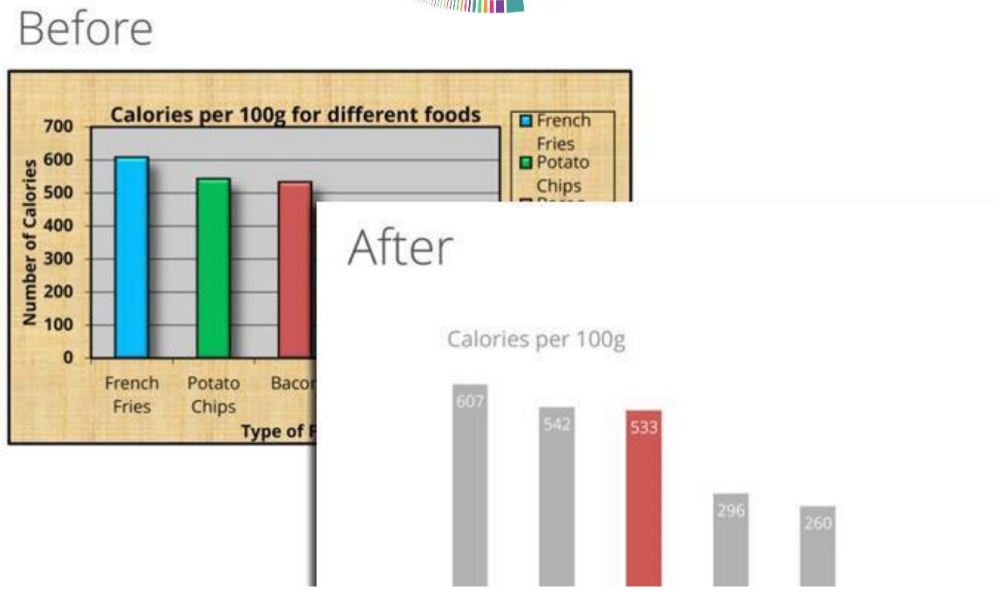
Carmine Tommaso Recchiuto

Why build visuals?

- Exploratory Data analysis
- Communicate data clearly
- Share unbiased representation of data
- Use them to support recommendations or choices
- Fast debug

Sometimes less is more: concise, straight- forward information is the most effective way to convey a message.



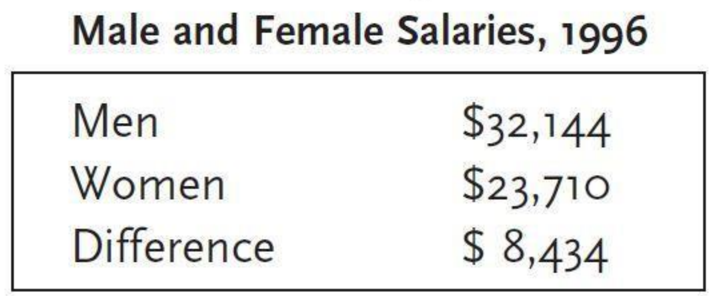


Booth, W. C., Colomb, G. G., & Williams, J. M. (2003). *The craft of research*. University of Chicago press.

Communicating Evidence Visually

Some reports of quantitative data are just as clear verbally as visually:

In 1996, on average,
men earned $32,144 a year,
women $23,710,
a difference of $8,434.



Male and Female Salaries, 1996

Men	$32,144
Women	$23,710
Difference	$ 8,434

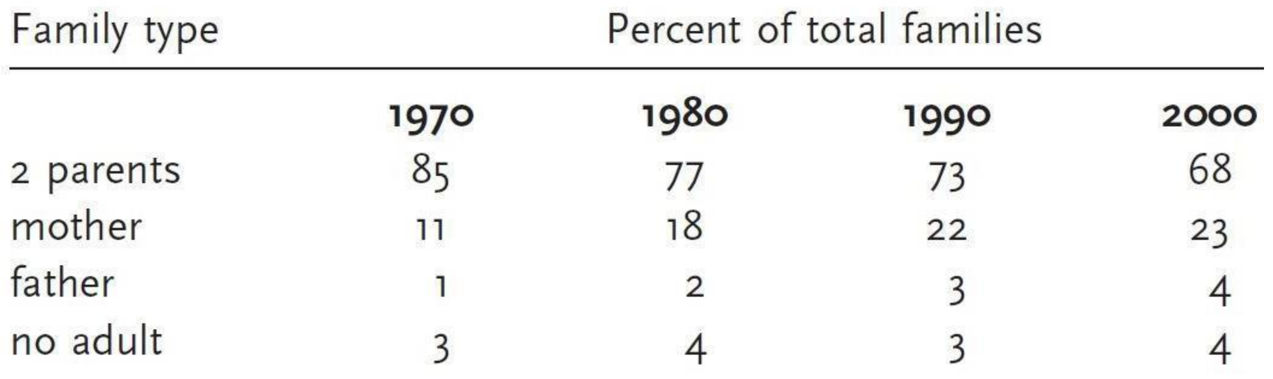
But when the numbers are more complex, readers need a more systematic presentation, first simply to absorb them, then to analyze and understand them.

In 1970 almost nine out of ten families had two parents—85 percent. But in 1980 that number declined to 77 percent, then to 73 percent in 1990, and to 68 percent in 2000. The number of one-parent families rose, particularly families headed by just a mother. In 1970 just 11 percent of families were headed by a single mother. In 1980 that number rose to 18 percent, in 1990 to 22 percent, and to 23 percent in 2000. Single fathers headedjust 1 percent of the families in 1970, 2 percent in 1980, 3 percent in 1990, and 4 percent in 2000. Families with no adult in the home have remained stable at 3–4 percent from 1970–2000.

Communicating Evidence Visually

Those numbers would be far more accessible in a table:

In 1970 almost nine out of ten families had two parents—85 percent. But in 1980 that number declined to 77 percent, then to 73 percent in 1990, and to 68 percent in 2000. The number of one-parent families rose, particularly families headed by just a mother. In 1970 just 11 percent of families were headed by a single mother. In 1980 that number rose to 18 percent, in 1990 to 22 percent, and to 23 percent in 2000. Single fathers headedjust 1 percent of the families in 1970, 2 percent in 1980, 3 percent in 1990, and 4 percent in 2000. Families with no adult in the home have remained stable at 3–4 percent from 1970–2000.

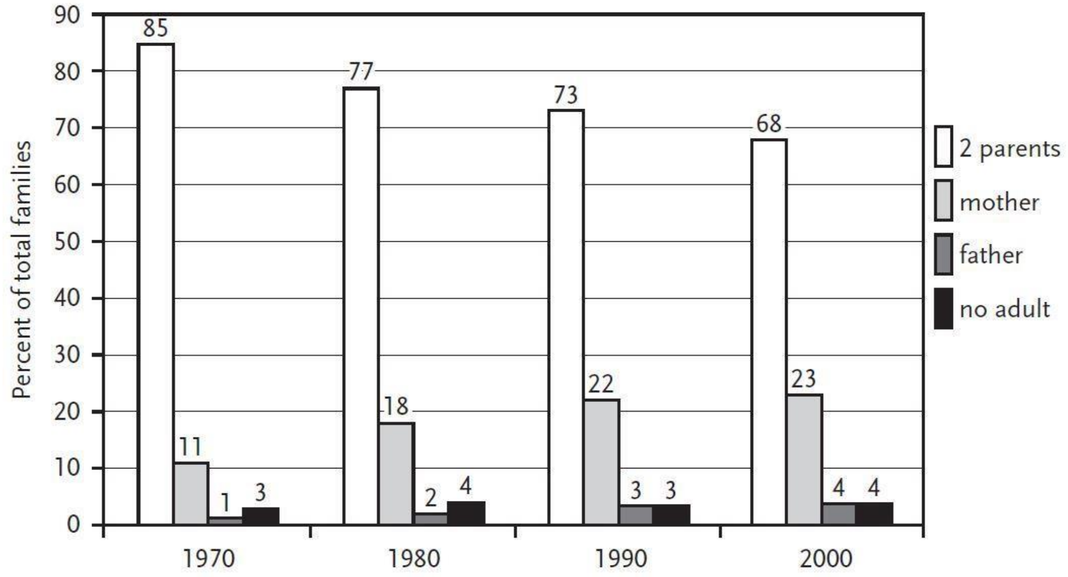


Family type	Percent of total families			
	1970	1980	1990	2000
2 parents	85	77	73	68
mother	11	18	22	23
father	1	2	3	4
no adult	3	4	3	4

Communicating Evidence Visually

Or as a bar chart:

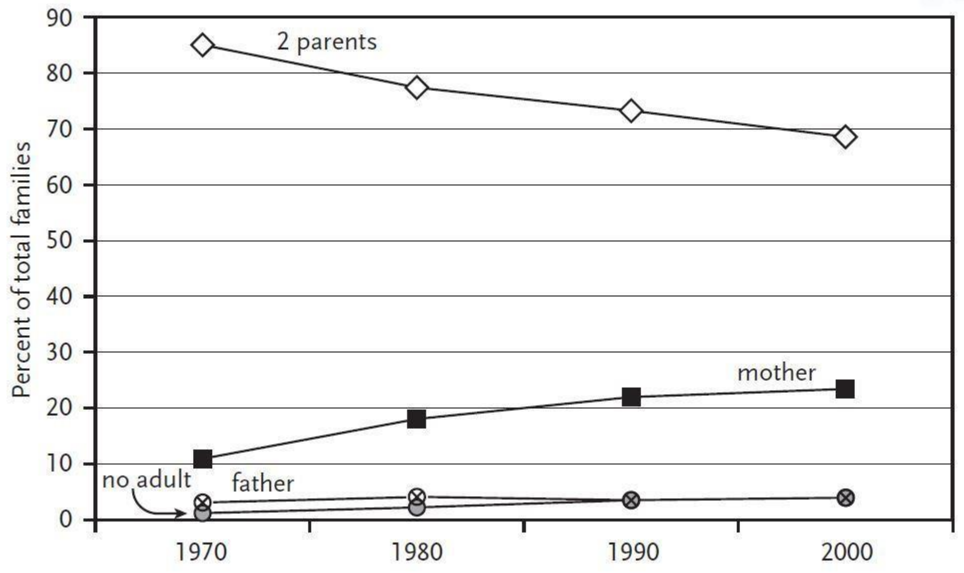
In 1970 almost nine out of ten families had two parents—85 percent. But in 1980 that number declined to 77 percent, then to 73 percent in 1990, and to 68 percent in 2000. The number of one-parent families rose, particularly families headed by just a mother. In 1970 just 11 percent of families were headed by a single mother. In 1980 that number rose to 18 percent, in 1990 to 22 percent, and to 23 percent in 2000. Single fathers headedjust 1 percent of the families in 1970, 2 percent in 1980, 3 percent in 1990, and 4 percent in 2000. Families with no adult in the home have remained stable at 3–4 percent from 1970–2000.



Communicating Evidence Visually

Or as a line graph:

In 1970 almost nine out of ten families had two parents—85 percent. But in 1980 that number declined to 77 percent, then to 73 percent in 1990, and to 68 percent in 2000. The number of one-parent families rose, particularly families headed by just a mother. In 1970 just 11 percent of families were headed by a single mother. In 1980 that number rose to 18 percent, in 1990 to 22 percent, and to 23 percent in 2000. Single fathers headedjust 1 percent of the families in 1970, 2 percent in 1980, 3 percent in 1990, and 4 percent in 2000. Families with no adult in the home have remained stable at 3–4 percent from 1970–2000.



Communicating Evidence Visually

The choice about how to represent data will have different rhetorical effects:

- The table of numbers feels precise and objective. It does not impose on us any predigested outcome. It lets us compare the numbers systematically and come to our own conclusion.

- The bar chart gives us less exact information (though that is compensated for by adding the numbers above the bars). But it visually communicates its point quickly. In particular, it helps us make individual comparisons.

- The line graph also gives us less exact information but offers an even more striking image of a story. It helps us see trends easily.

You must report evidence in a way that is clear, appropriate, and fair, but every choice unavoidably ***"spins" your evidence, giving it a particular rhetorical effect.***

Communicating Evidence Visually

Visual or Verbal?

Here are two general guidelines:

- Choose a table if your readers are likely to want very precise numbers and you don't want to impose on your data a visual image implying the point you want them to support.
- Choose a figure if your readers are less interested in precise details than in a general point, and you want to reinforce your point with a strong image

There are of course some exceptions: for example, if you had a vast amount of quantitative data, which refers to different aspects, no single figure could represent such complexity; you would have to use at least one table

Constructing Tables

If you choose to present your data as a table, observe the following principles

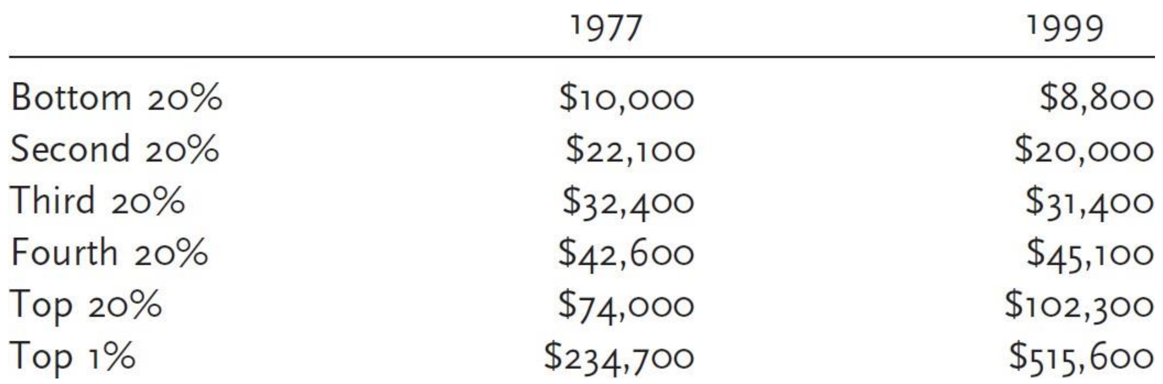
1. Introduce your data with a sentence that explicitly tells the reader what to see in them. Then give the table, graph, or chart a title that explicitly names its purpose.

2. Organize your table, bar chart, or line graph in a way that anticipates how your readers will use it, and highlight those data most relevant to the claim you want the data to support.

For example, on first reading, it's hard to see how these next data relate to the claim they seem intended to support because we have to do a lot of calculating:

Though the United States has had unprecedented economic growth in the last twenty-five years that has benefited some, most Americans have lost ground.

Income

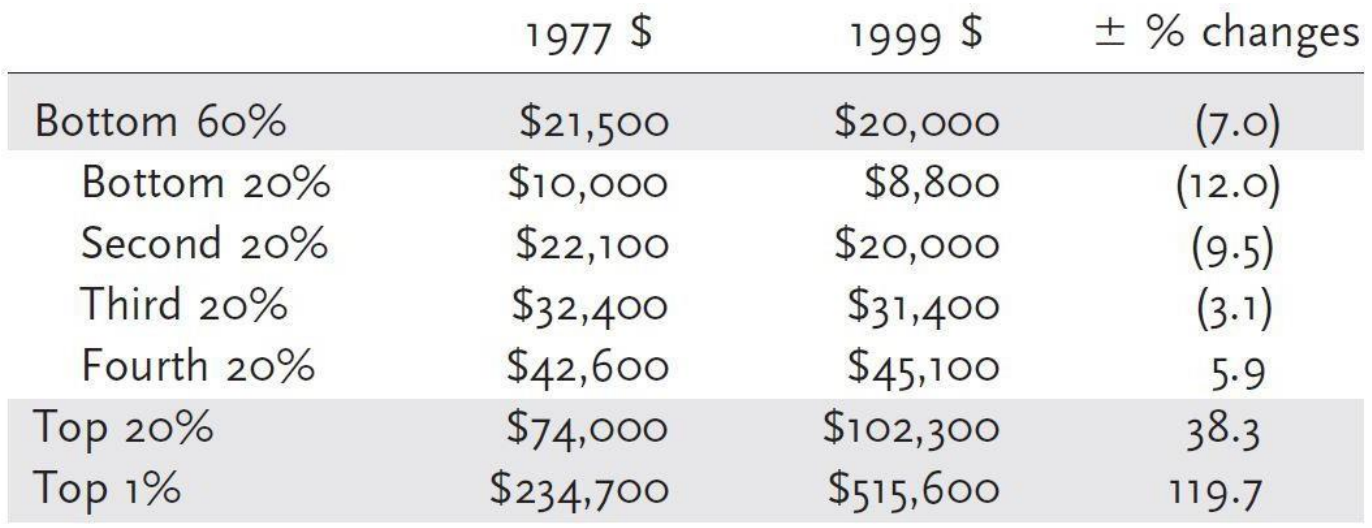


	1977	1999
Bottom 20%	$10,000	$8,800
Second 20%	$22,100	$20,000
Third 20%	$32,400	$31,400
Fourth 20%	$42,600	$45,100
Top 20%	$74,000	$102,300
Top 1%	$234,700	$515,600

Constructing Tables

We would understand the relevance of those data more quickly and clearly with four changes: (1) a prior sentence interpreting them, (2) an informative title specifying the topic, (3) the key comparisons calculated, and (4) their results highlighted. (In the table, negative numbers are represented in parentheses.)

Though the United States has had unprecedented economic growth in the last twenty-five years that has benefited some, most Americans have lost ground. **Between 1977 and 1999, the top 20 percent of wage earners increased their income by more than 38 percent, and the top 1 percent more than doubled theirs, but the bottom 60 percent of the population earned less in 1999 than they did in 1977.**

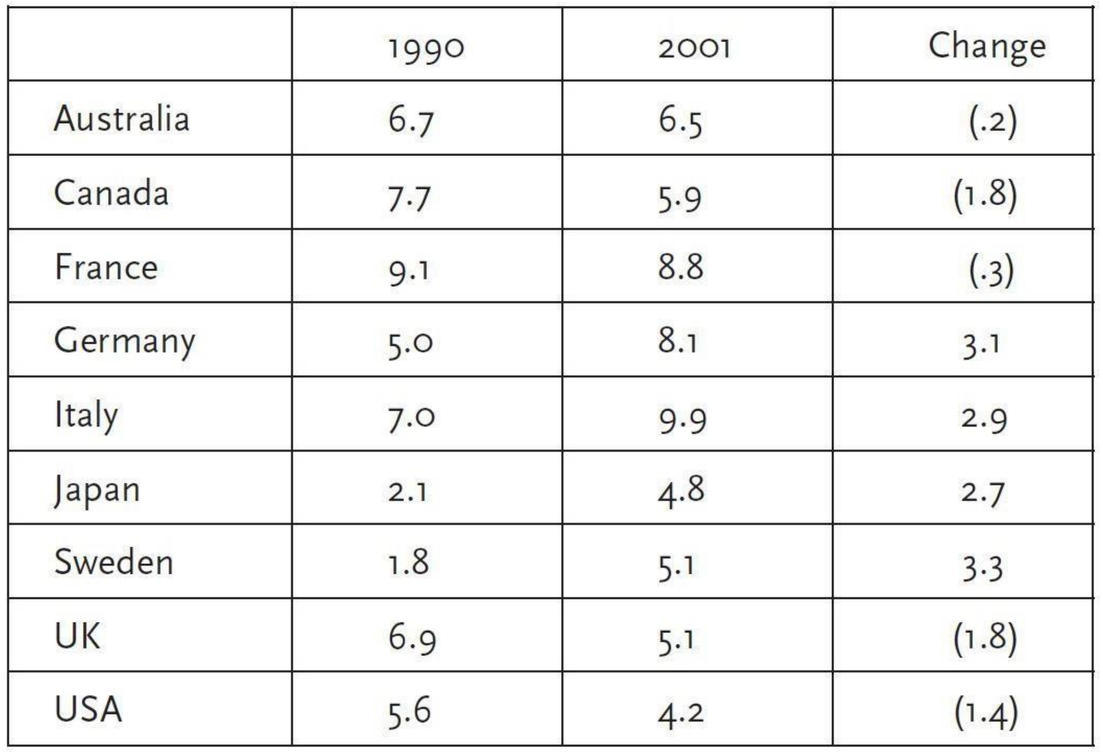


	1977 $	1999 $	± % changes
Bottom 60%	$21,500	$20,000	(7.0)
Bottom 20%	$10,000	$8,800	(12.0)
Second 20%	$22,100	$20,000	(9.5)
Third 20%	$32,400	$31,400	(3.1)
Fourth 20%	$42,600	$45,100	5.9
Top 20%	$74,000	$102,300	38.3
Top 1%	$234,700	$515,600	119.7

Constructing Tables

For example, suppose you want table A to show that *English-speaking nations* have reduced unemployment most in recent years. How easy is it to find the relevant data? How could you change this table?

Unemployment Rates of Major Industrial Nations

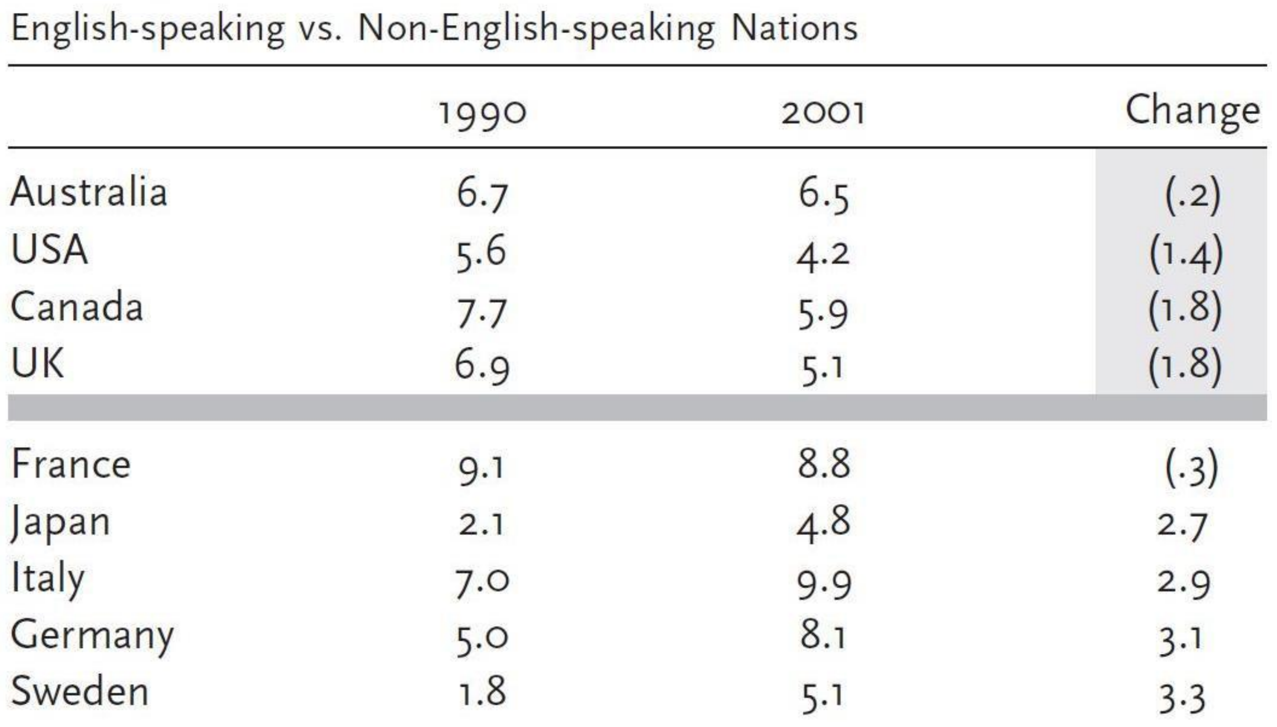


	1990	2001	Change
Australia	6.7	6.5	(.2)
Canada	7.7	5.9	(1.8)
France	9.1	8.8	(.3)
Germany	5.0	8.1	3.1
Italy	7.0	9.9	2.9
Japan	2.1	4.8	2.7
Sweden	1.8	5.1	3.3
UK	6.9	5.1	(1.8)
USA	5.6	4.2	(1.4)

Constructing Tables

Table B has a title that makes its point more clearly, but it also groups nations by language, orders each group by degree of change, and highlights the four relevant data points.

Changes in Unemployment Rates



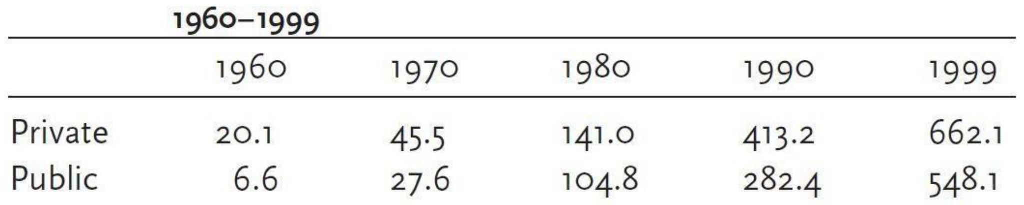
English-speaking vs. Non-English-speaking Nations

	1990	2001	Change
Australia	6.7	6.5	(.2)
USA	5.6	4.2	(1.4)
Canada	7.7	5.9	(1.8)
UK	6.9	5.1	(1.8)
France	9.1	8.8	(.3)
Japan	2.1	4.8	2.7
Italy	7.0	9.9	2.9
Germany	5.0	8.1	3.1
Sweden	1.8	5.1	3.3

Constructing Figures

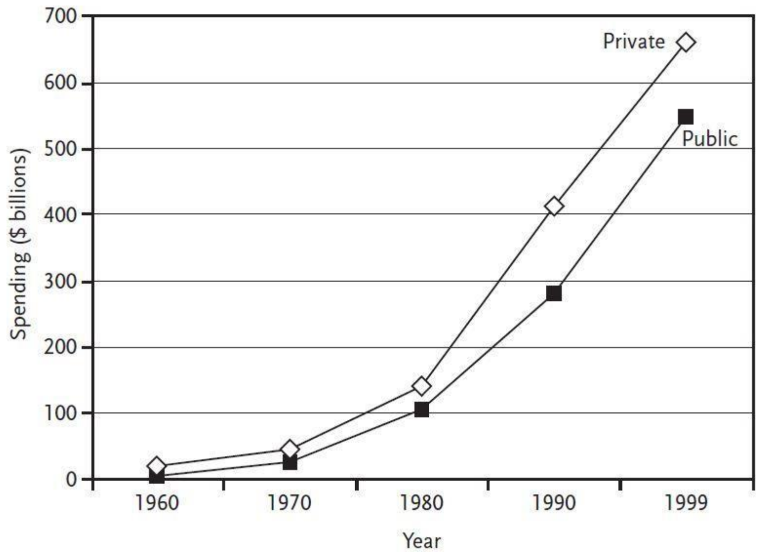
Choose a figure over a table if precise numbers are less important than readers' getting an image of the story in your data. Which of these representations tells its story more powerfully?

Rise in public and private spending on Health (in Billions)



1960–1999

	1960	1970	1980	1990	1999
Private	20.1	45.5	141.0	413.2	662.1
Public	6.6	27.6	104.8	282.4	548.1



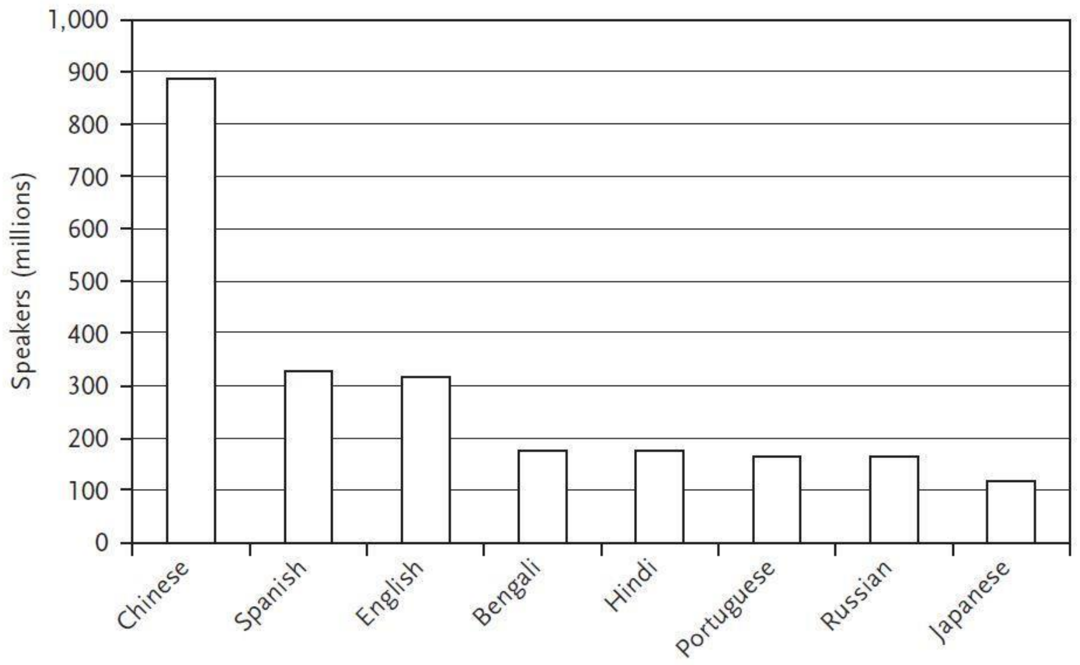
Rise in public and private spending on Health (in Billions)

But, how to choose the correct type of visual?

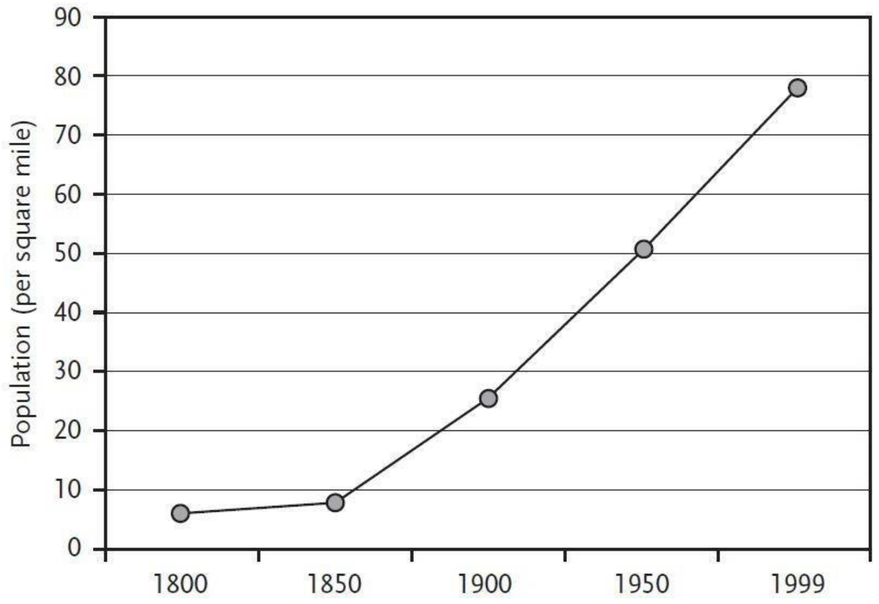
Constructing Figures

There is a general principle in choosing between a graph and a chart:

- Choose a vertical bar chart to represent static situations, where entities (the dependent variables) are measured at a moment in time:



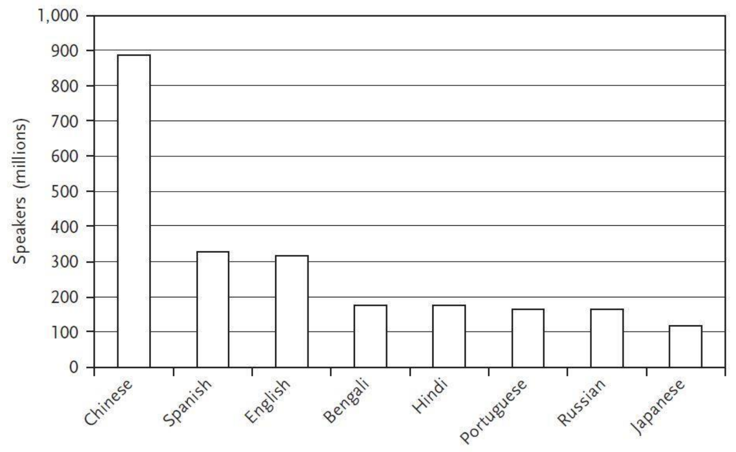
Languages spoken by more than 100 Million Speakers

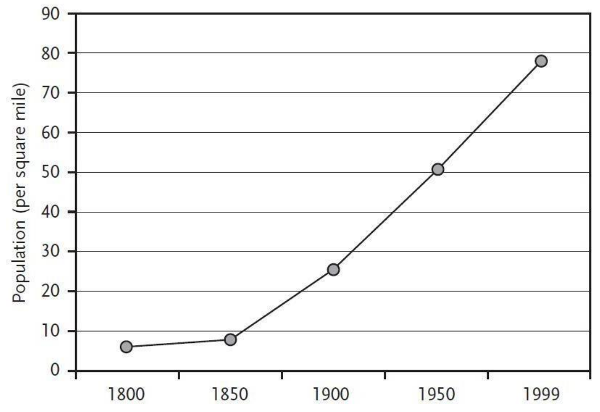


Increases in Population Density in the US, 1800 - 1999

General Principles

- As you do with tables, introduce all figures with a sentence that tells the reader the point of the data and create a title that reinforces the point.
- Use "tick" marks on the vertical Y-axis to help readers see the point of measure; if the tick marks are finely graded, boldface every fifth one.
- Use grid lines to help readers estimate numbers (run the grid lines *behind* the bars).





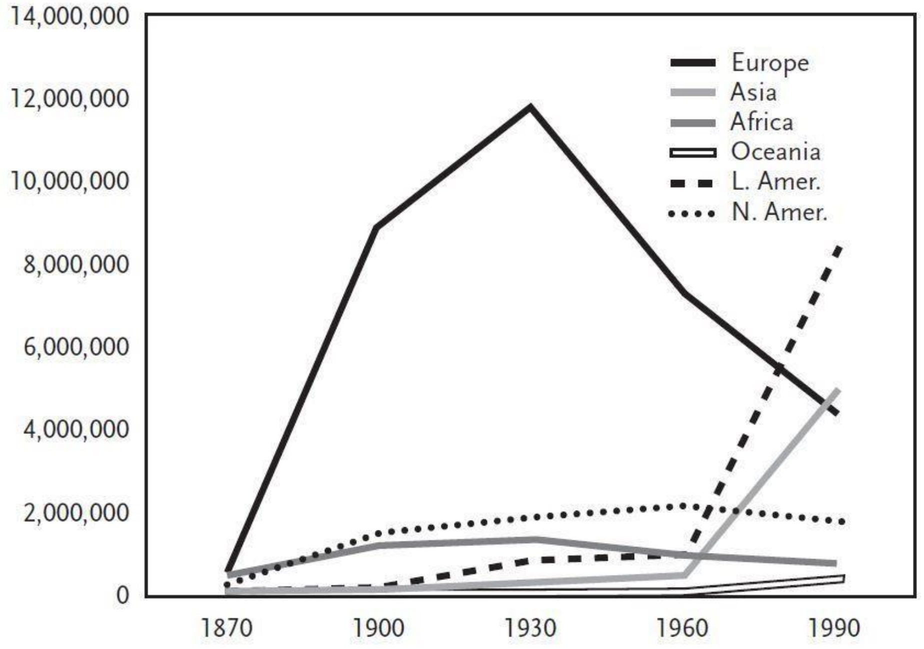
Line Graphs

Keep in mind these three principles:

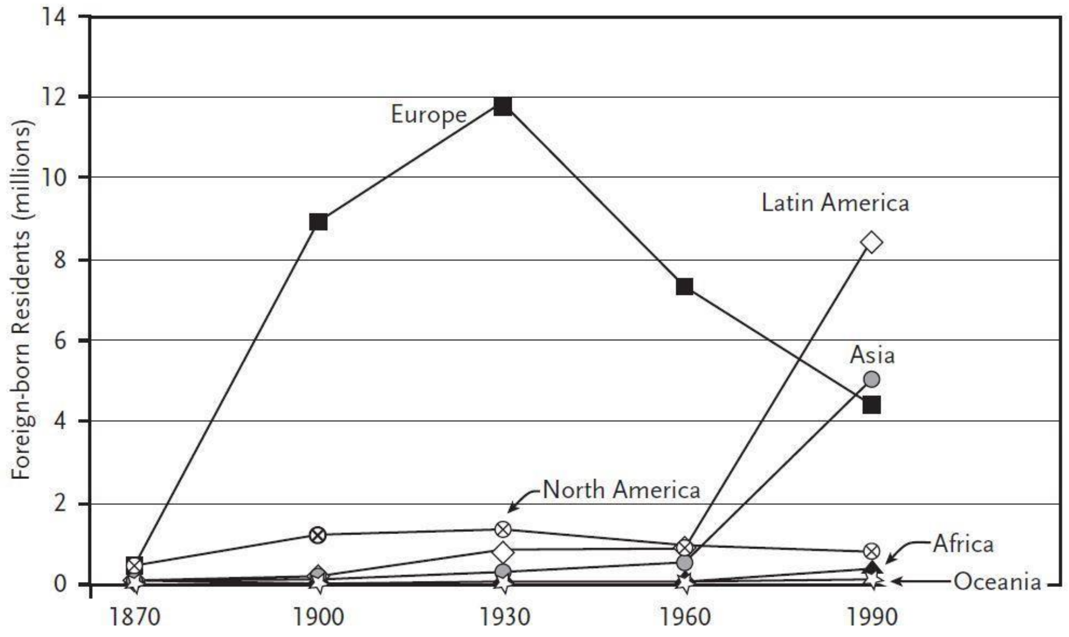
- Keep the image as uncluttered as possible. If you are plotting more than four dependent variables, you risk confusing your readers.
- If you cannot divide a complex graph into two graphs, then clearly distinguish the lines for each element; if you can, label the lines rather than using a legend (even though the added labels further complicate the image).
- Help readers see clearly the data points on the line. Put a dot at each relevant data point.

Line Graphs

Which is easier to understand?
What does that graph do to help you understand its data?



Foreign-born Residents in the United States

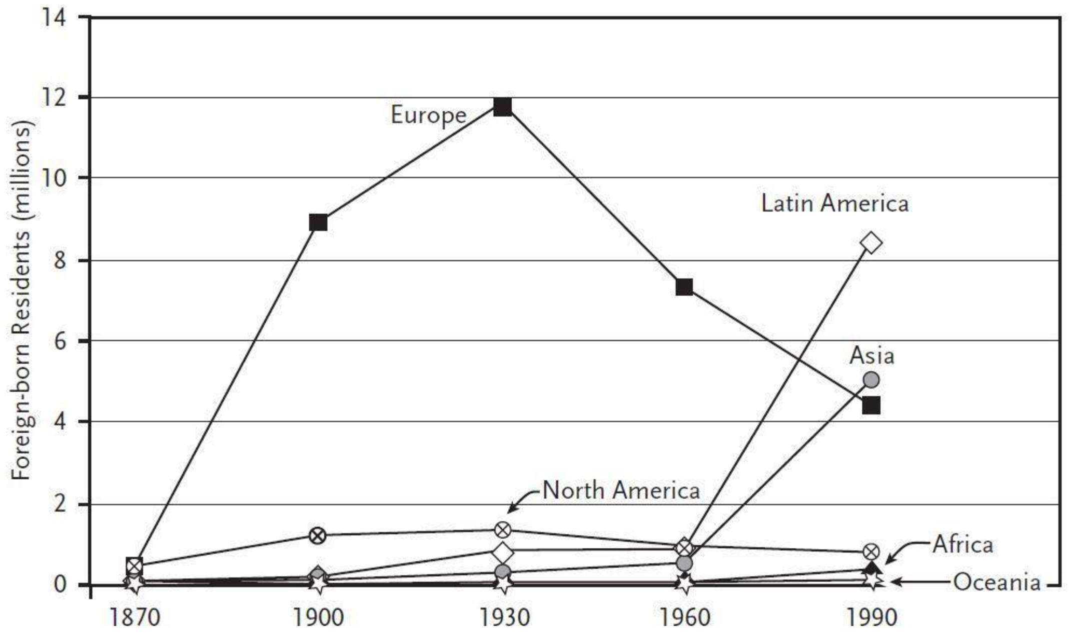


Foreign-born Residents in the United States

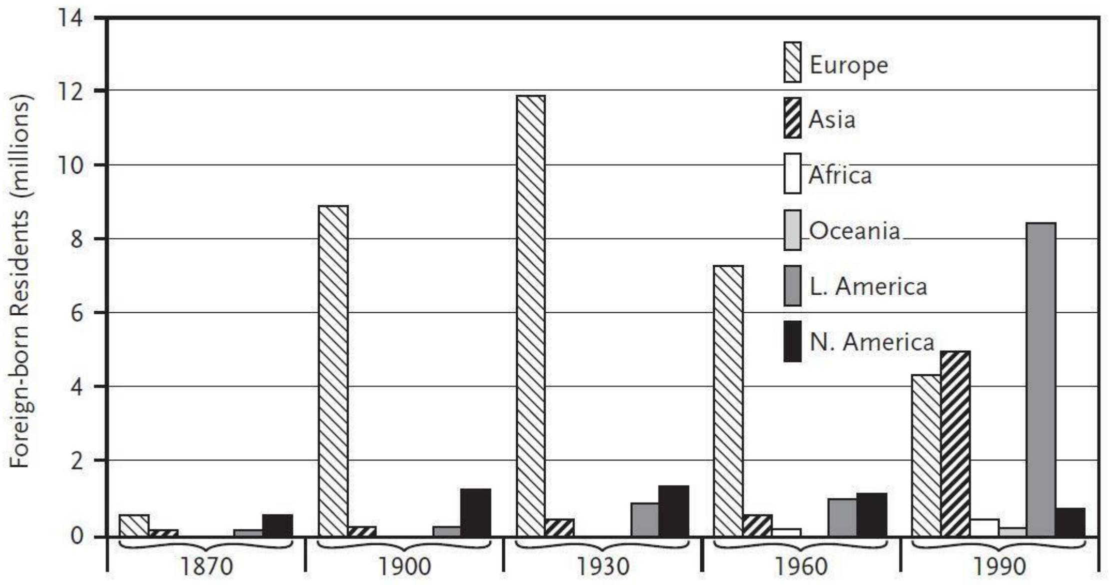
Bar Charts

A bar chart may also be used to compare discrete dependent variables at a single moment in time. It is possible to represent time for multiple entities on a bar chart, but the image becomes very complicated, making it difficult for a reader to see relevant differences.

In effect, you get a series of little bar charts imposed on a single big one.



Foreign-born Residents in the United States



Foreign-born Residents in the United States

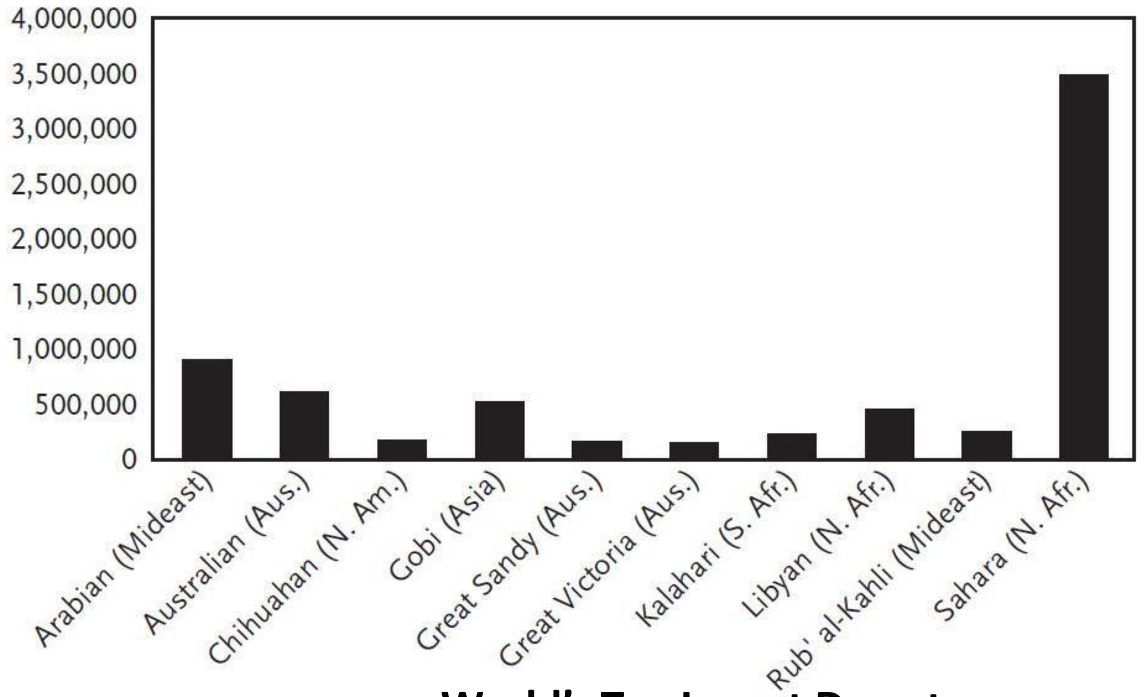
Bar Charts

If your X-axis does not represent time, you are generally free to order the bars along the X-axis as you wish, but there are some principles for doing that:

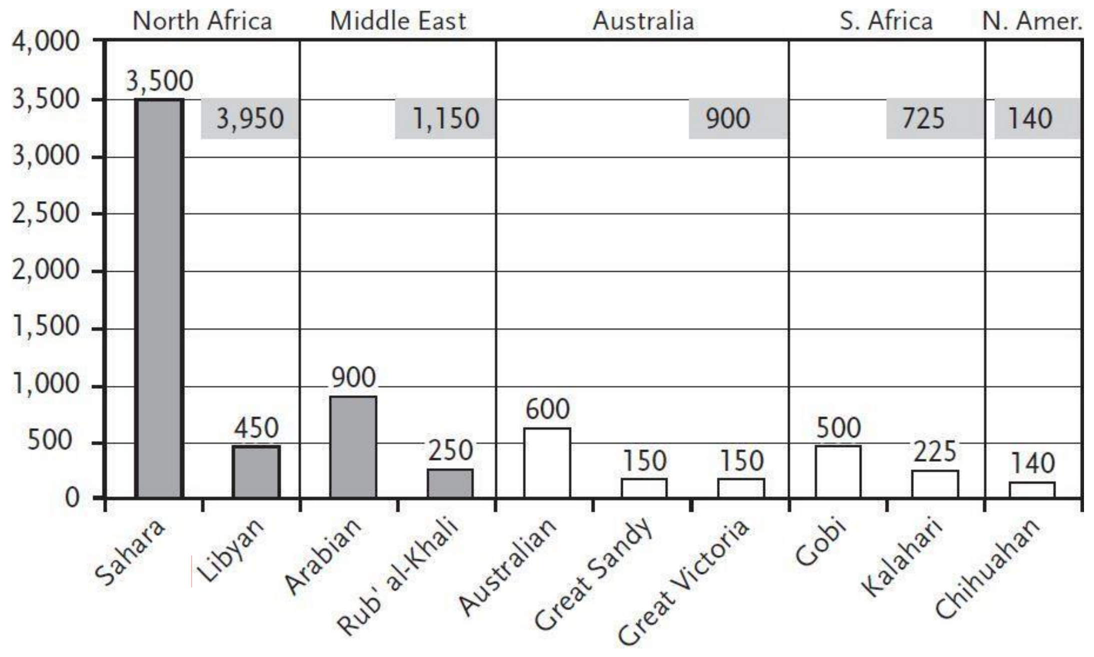
- Group the bars into related sets whenever possible.
- Arrange the bars so that they give an image of order.
- Highlight a bar if it is a relevant point of comparison for the others.
- Keep visual contrasts simple: black, white, and one or two shades of gray. If possible, avoid cross-hatching, stripes, and so on (impossible if you try to chart too many cases).
- If necessary, include numbers above the bars to give readers more precision.

Bar Charts

Figure A is organized alphabetically, but that does not help readers find the data that support the claim. We see no numbers associated with the bars. No grid lines help us connect numbers to the Y-axis. And the Y-axis has no tick marks. In contrast, figure 15.11 is organized into a coherent picture to support that claim



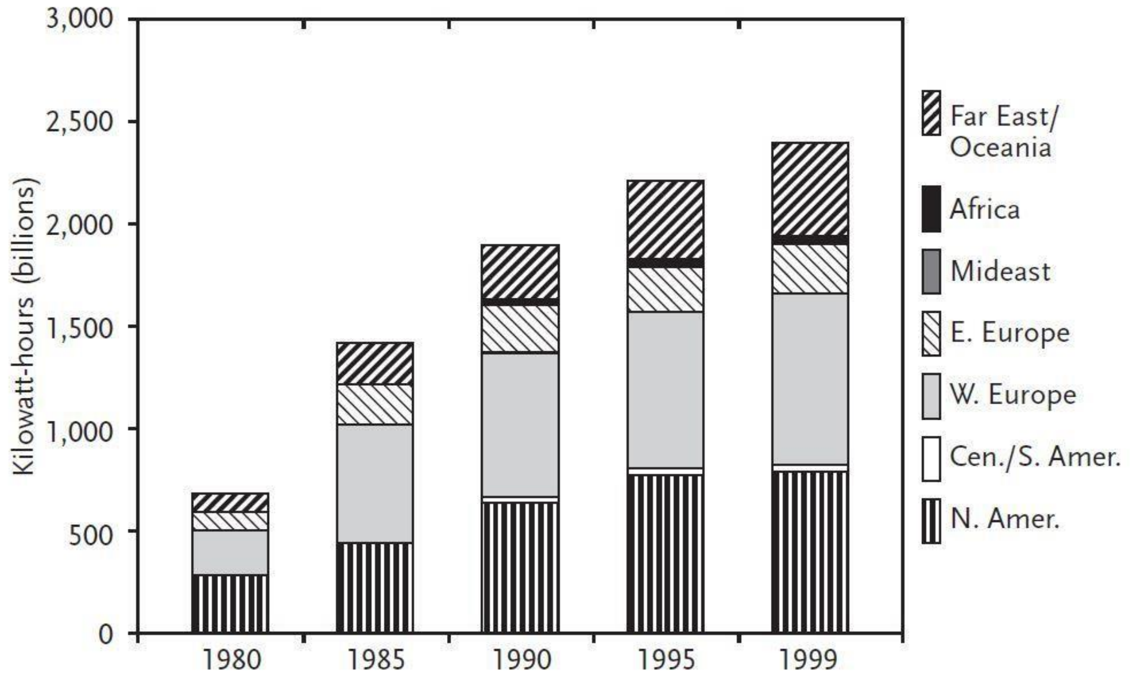
World's Ten Largest Deserts



World's Ten Largest Deserts

Stacked Bar Charts

Stacked bars are a variation on side-by-side bars. Stacked bars divide the bar into its relative proportions of 100 percent of some other variable. They can be difficult to process because they force readers to make comparisons and proportions by eye alone.

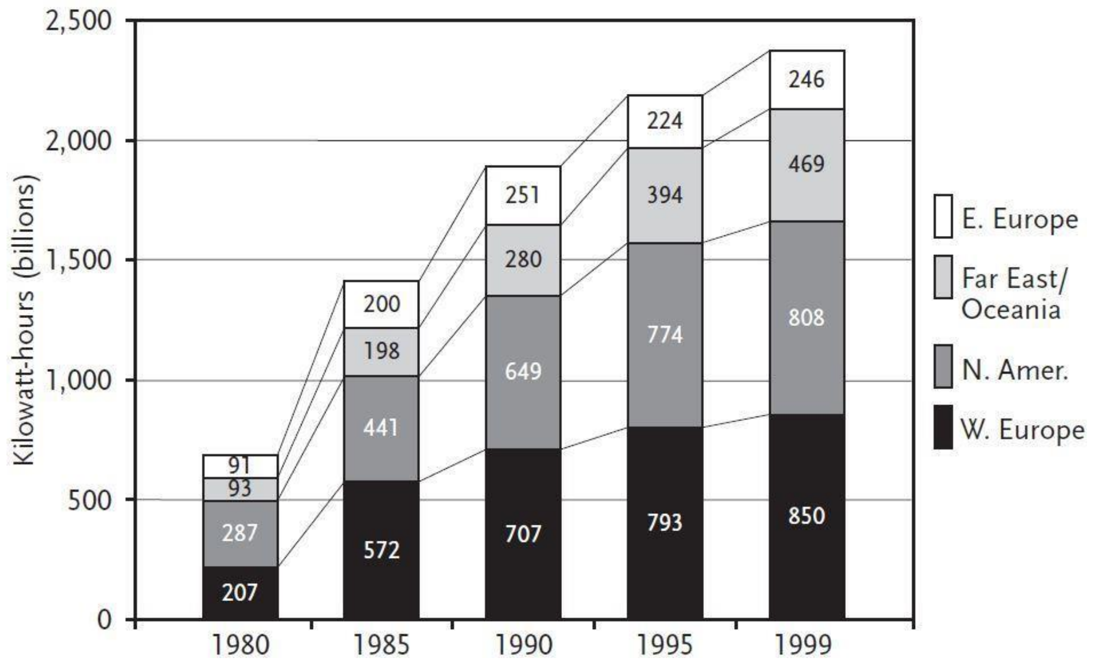


World Generation of Nuclear Energy, 1980 - 1999

Stacked Bar Charts

If you want to use stacked bars, help readers by following these principles:

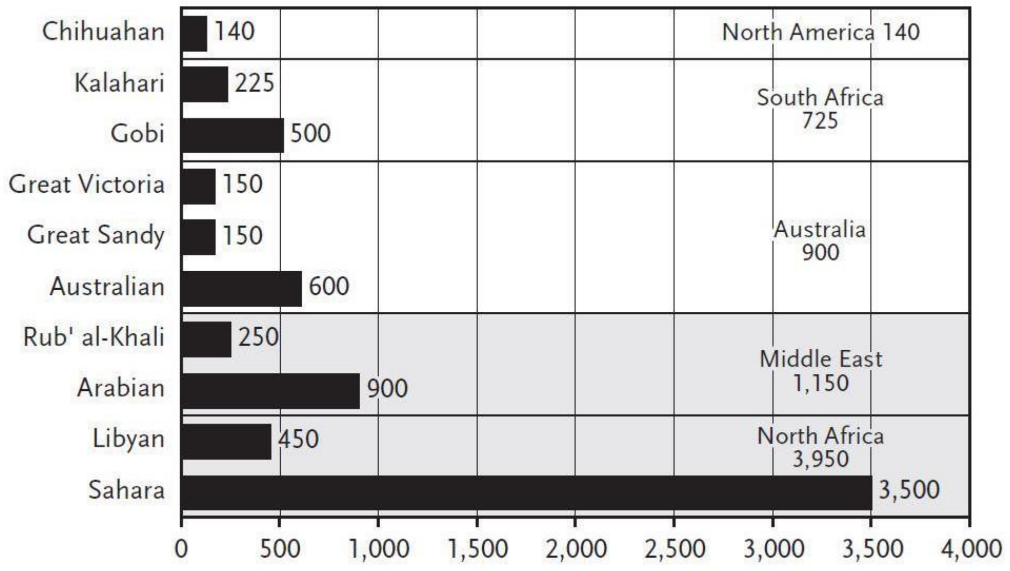
- Arrange the segments in some principled order, from bottom to top. If feasible, put the largest elements at the bottom, smaller ones on top, and use the darkest color at the bottom, lightest at the top.
- Use numbers and connecting lines to clarify proportions.
- Don’t bother to include cases whose numbers are too small w.r.t. larger ones



Largest Generators of Nuclear Energy, 1980 - 1999

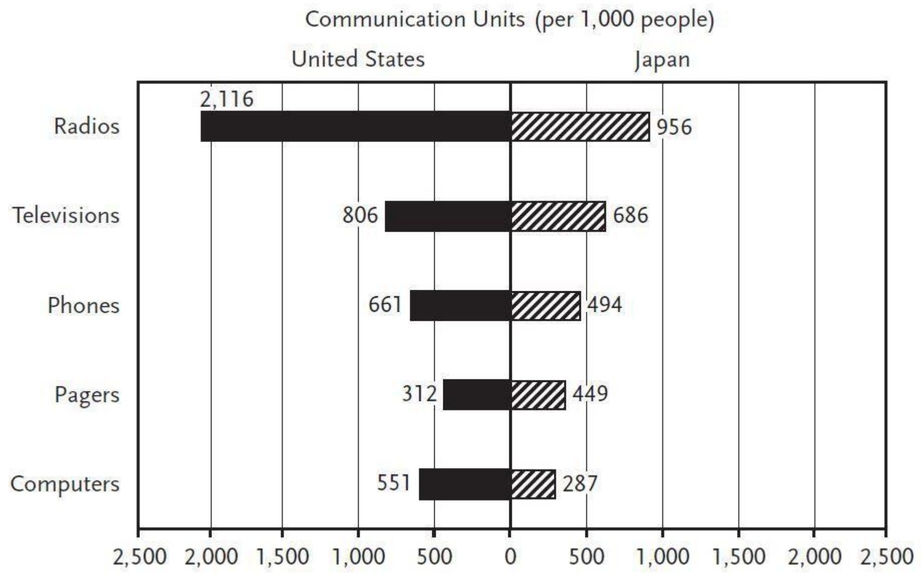
Horizontal Bar Charts

The only advantage of a horizontal bar chart over a vertical one is typographical: it lets you get the whole name of an item next to a bar:



World's Ten Larget Deserts

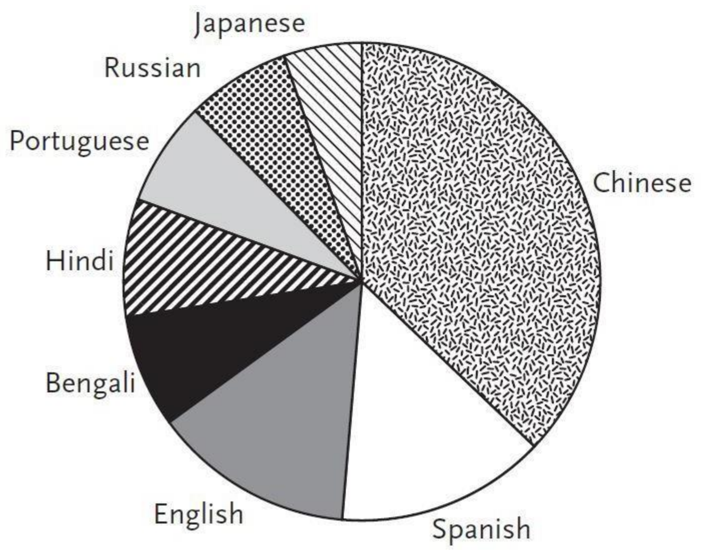
A variation on a horizontal bar chart is a centrally divided horizontal bar chart. It puts two dependent variables on either side of a center line and then displays a number of independent variables. The same data can be represented in a side-by-side vertical bar chart, but it is more uncommon.



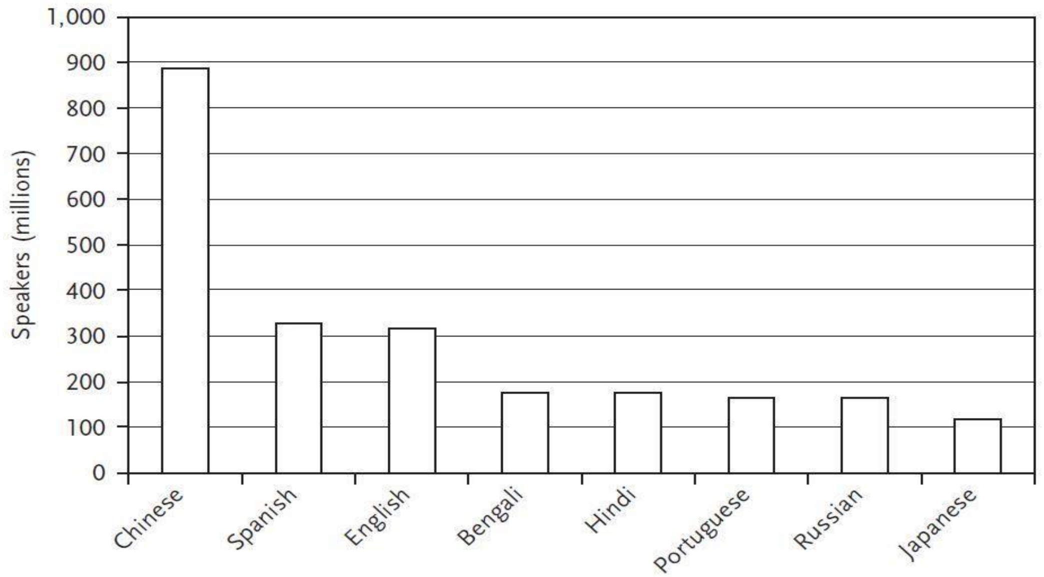
Electronic Communications, Year 2000

Pie Charts

Pie charts are favorites of newspapers and annual business reports but are often considered a bit amateurish for academic research. At best, they allow readers to see crude proportions among a few elements that constitute 100 percent of a whole. They are hard to read when they have more than four or five segments, particularly when the segments are thin and readers have to look at a key to match the patterns in the segments with categories. When readers try to judge the relative size of segments, they are likely to be wrong.



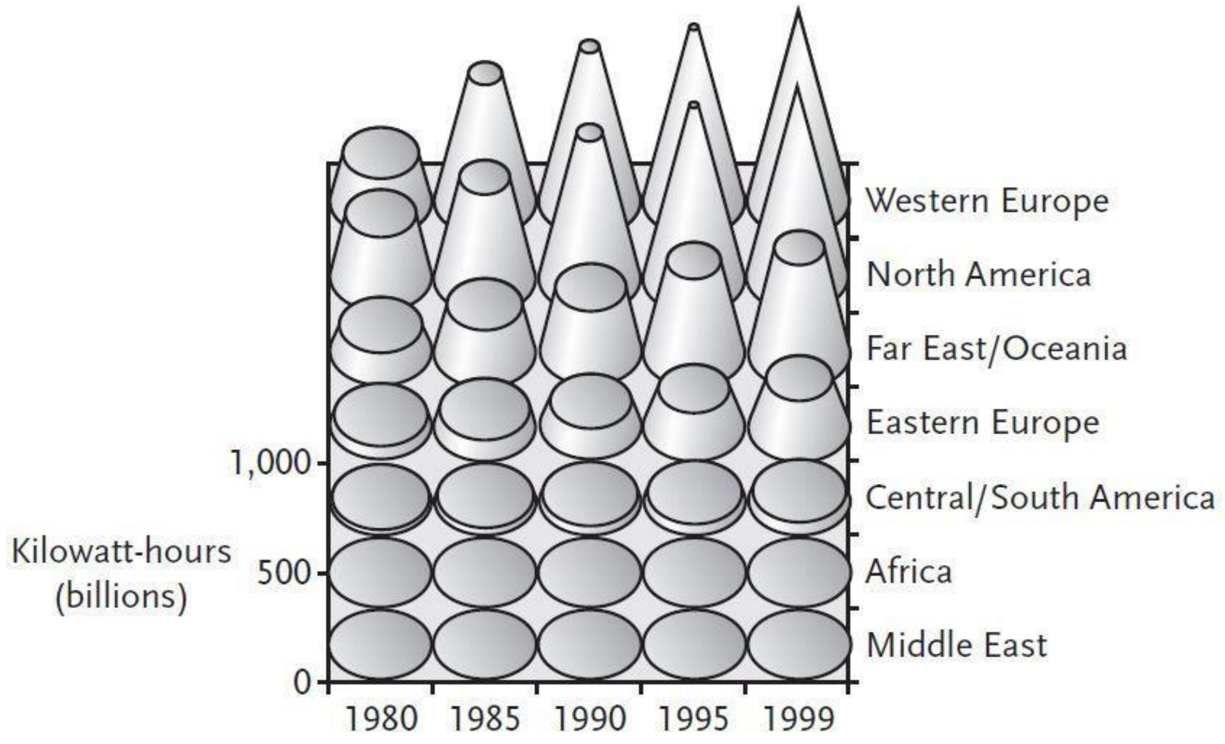
Languages with More than 100 Million Speakers



Languages spoken by more than 100 Million Speakers

Three-Dimensional Graphics

Only rarely dataare complex enough to require a three-dimensional representation. Most often, the third dimension is purely decorative, which is to say distracting from its point.



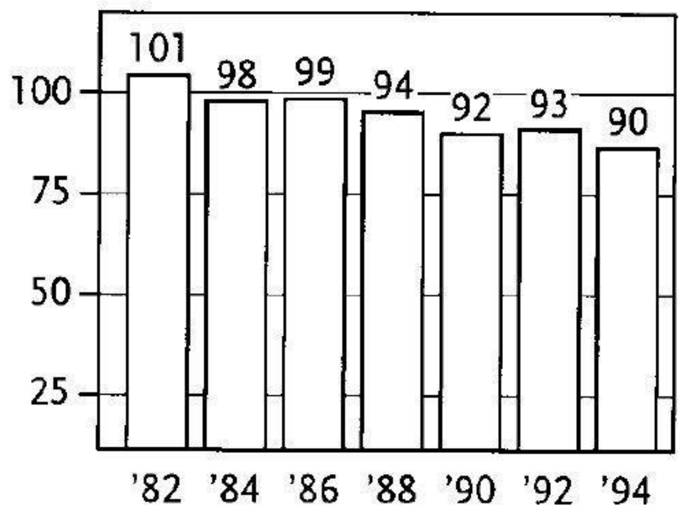
World Generation of Nuclear Energy, 1980 - 1999

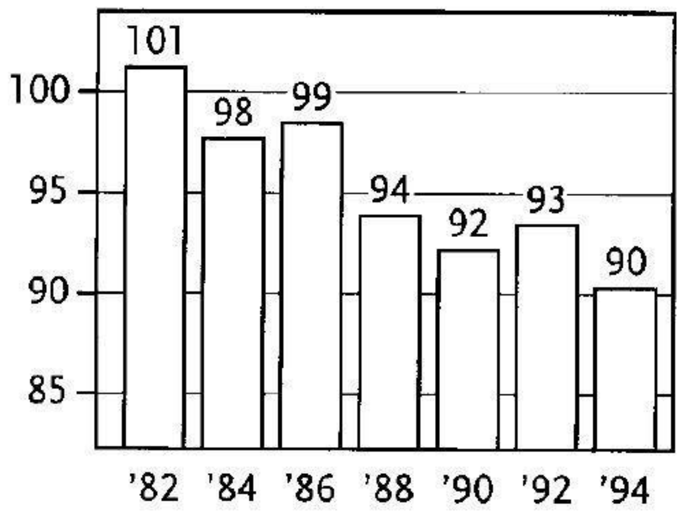
Visual Communication and Ethics

Whenever you present data visually, you have to balance your rhetorical goals and your responsibility not just to the facts but to the fairness of their appearance.

Tables, charts, and graphs always seem objective, and so they can fool inexperienced readers. But they will make experienced readers suspicious if you seem to distort the image to serve your story.

For example, compare the two charts here. The data in the two are identical, but look at the slope of the bars:



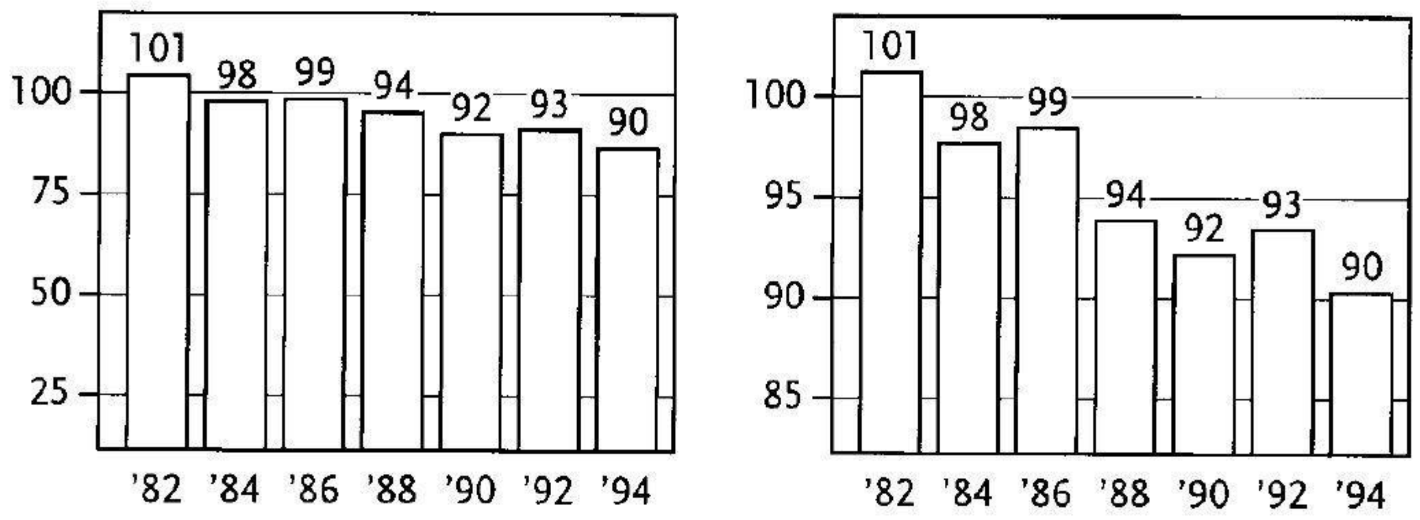


Capitol City Pollution Index, 1982- 1994

Visual Communication and Ethics

The chart on the right suggests more improvement, a story that might mislead some readers and that might even be considered dishonest by others.

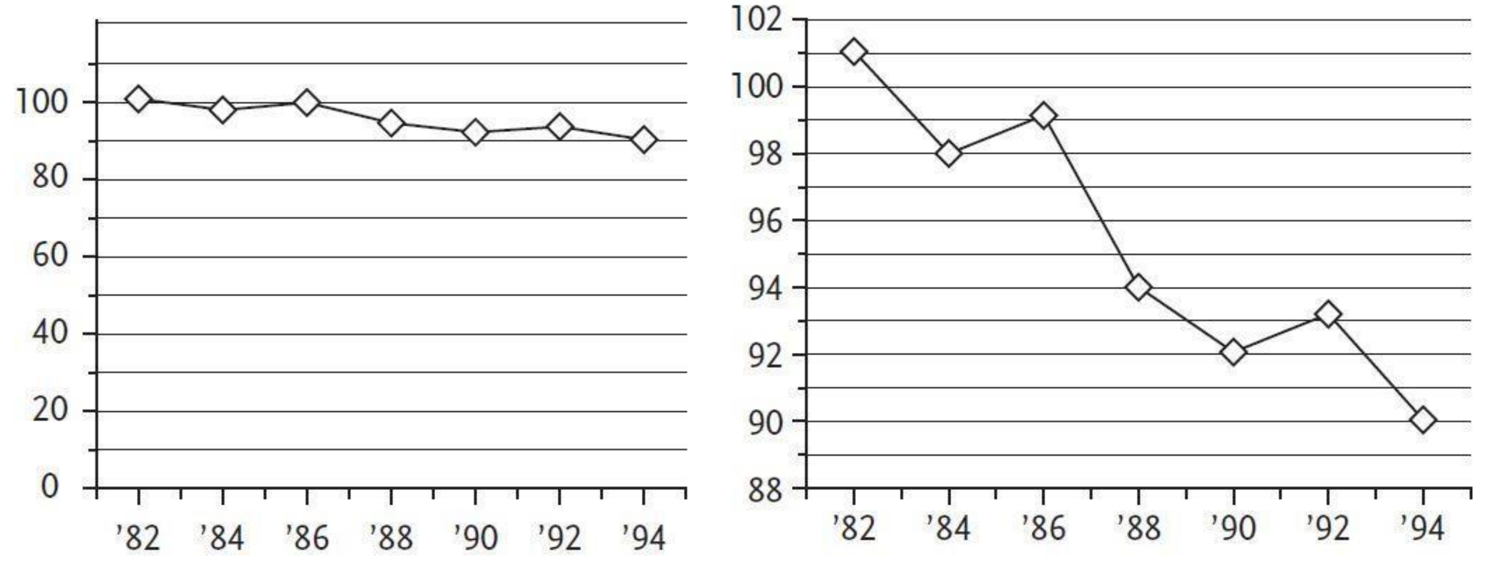
The distortion in the figure on the right is mitigated by the fact that the bars are clearly labeled with precise values.



Capitol City Pollution Index, 1982- 1994

Visual Communication and Ethics

But a writer who truncates the vertical axis of a graph to make a slope seem sharper may cross the line of honesty

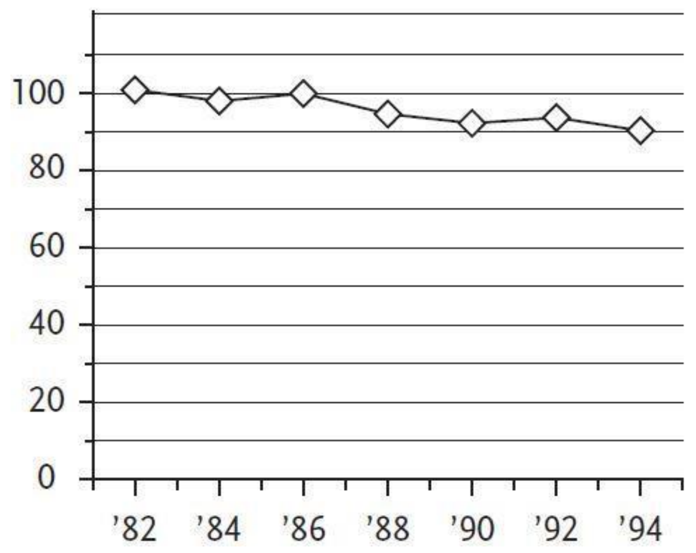


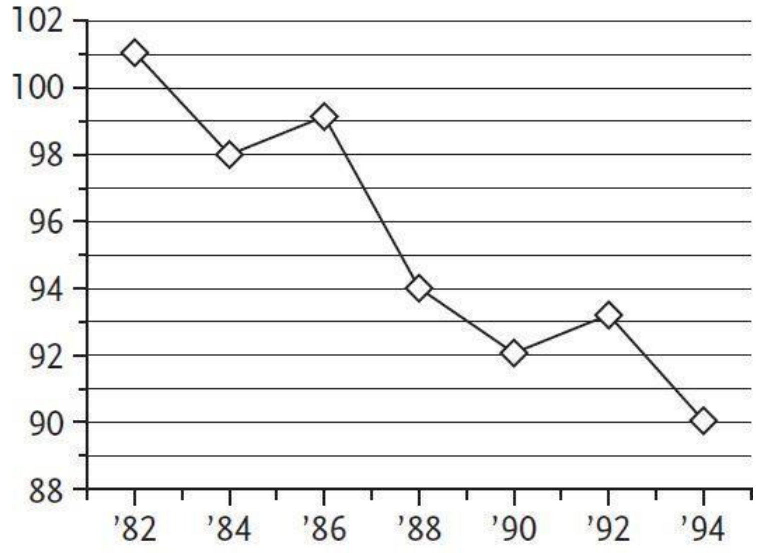
Capitol City Pollution Index, 1982- 1994

To the viewer the slope of a graph is always the predominant image!

Visual Communication and Ethics

On the other hand, it is not always easy to distinguish what is “objective” from what is “ethical.” Suppose you are an environmental scientist and you know that any expert would consider these seemingly small decreases to be highly significant. If you were certain that your statistically unsophisticated readers would dismiss the visually slight differences on the left as meaningless, then that larger visual difference on the right would more accurately communicate the real scientific significance. In that case, it’s harder to decide which graph is more honest.





Capitol City Pollution Index, 1982- 1994

Aid To Thinking

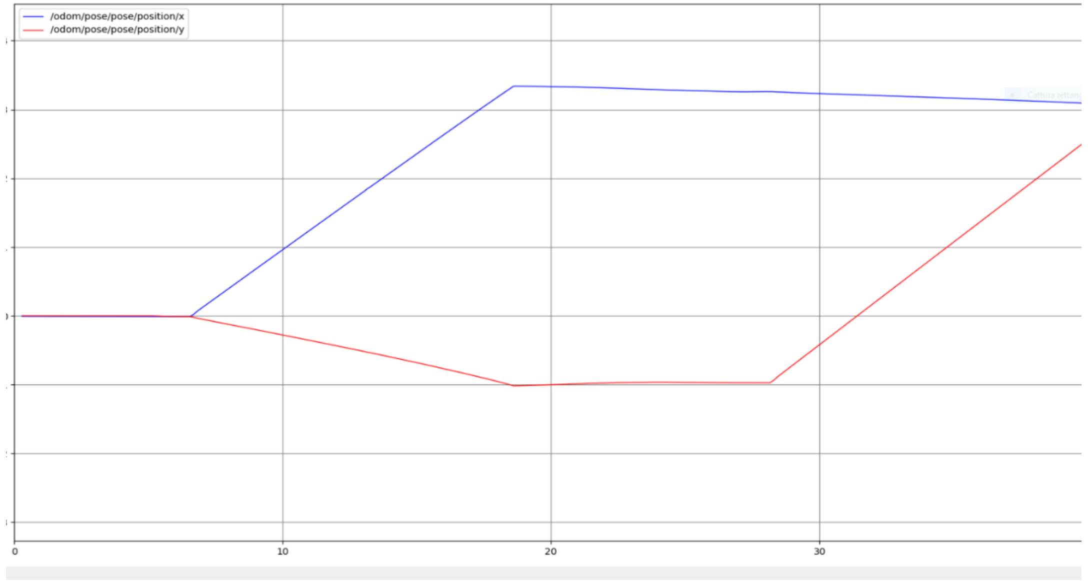
Data can help you to see things in new ways, to see trends, discover new relationships, recognize the significance of a particular set of data.

So don’t just think of graphics as an appealing way to show readers lots of data. Try out many combinations of data and kinds of representations: you can never tell what insight you will have after looking at those pictures. We display data not just to make them clear, but to help us see them in new and striking ways

Implementation - ROS

Let’s now check some more details about implementation. ROS has already some embedded tools for data visualization:

- *rviz*
- some *rqt* plugins
- *rqt_plot*, which allows for visualizing in run-time data published on ROS topics. There are two ways to give the topic names to rqt_plot: from commandline or directly using the GUI. In both ways, topics that are set in previous run is resumed (as far as the program was shut down without error).



Rqt_plot has different plotting backend options, among which we are using matplotlib.
Limitations: no all datatypes may be supported, and there is no way to plot a 2D or a 3D graph.

Python e Matplotlib

Matplotlib is one of the most widely used, if not the most popular data visualization library in Python. It was created by John Hunter, who was a neurobiologist and was part of a research team that was working on analyzing Electrocorticography Signals.

Matplotlib's architecture is composed of three main layers:

- ✓ the back-end layer
- ✓ the artist layer
- ✓ the scripting layer,

The latter is the appropriate layer for everyday purposes and is considered a lighter scripting interface to simplify common tasks and for a quick and easy generation of graphics and plots

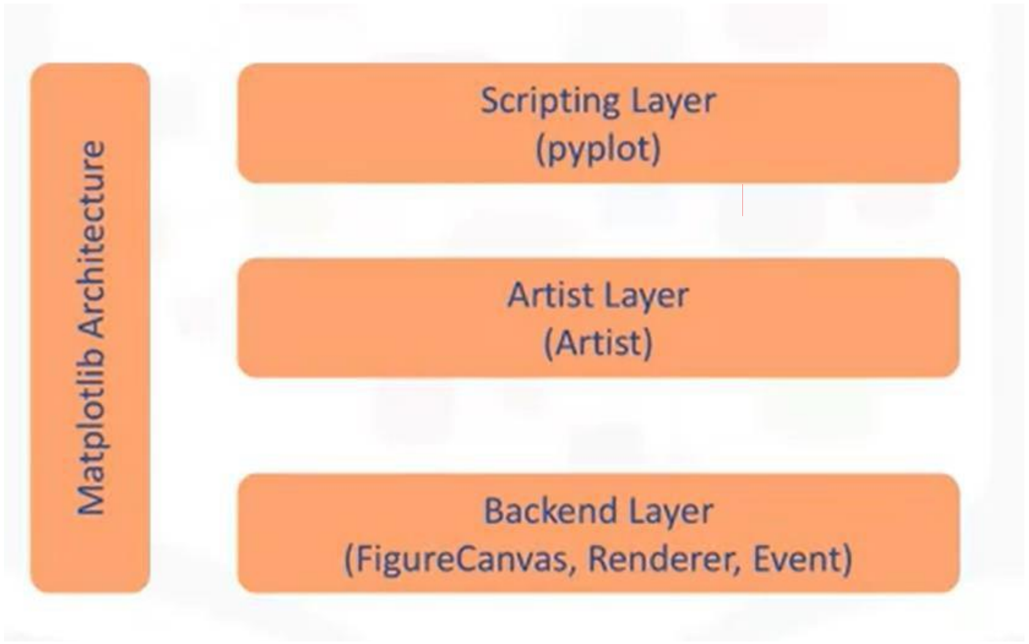
Python e Matplotlib

The **back-end layer** has three built-in abstract interface classes:

- FigureCanvas, which defines and encompasses the area on which the figure is drawn -> **matplotlib.backend_bases.FigureCanvas**

- Renderer, which knows how to draw on the figure canvas. -> **matplotlib.backend_bases.Renderer**

- Event, which handles user inputs such as keyboard strokes and mouse clicks. -> **matplotlib.backend_bases.Event**



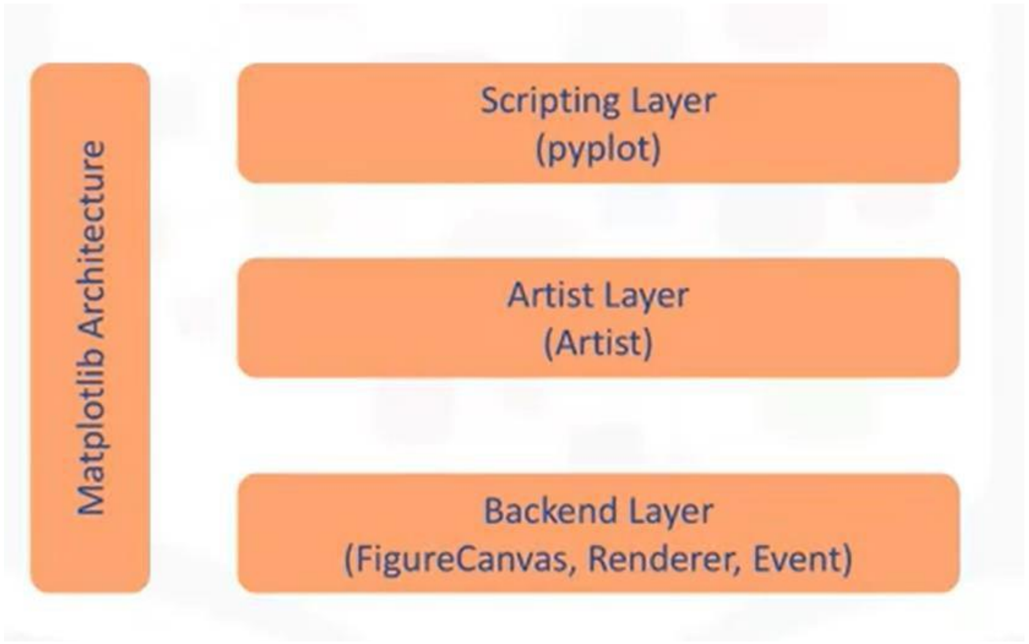
Python e Matplotlib

The **artist layer** is composed of one main object, which is the artist. The artist is the object that knows how to take the Renderer and use it to put ink on the canvas.

Everything you see on a Matplotlib figure is an artist instance: title, lines, tick labels and images, all correspond to **individual Artist instances**.

There are two types of Artist objects:
- primitive types, such as a line, a rectangle, a circle, or text
- composite type, such as the figure or the axes.

Each composite artist may contain other composite artists as well as primitive artists. So a figure artist for example would contain an axis artist as well as a rectangle or text artists.



Artist Layer

Let's see how we can use the Artist Layer to generate a graph, for example a histogram of some data.

```
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure
fig = Figure()
Canvas = FigureCanvas(fig)
```

First we import the **figure canvas** from the backend. Note that agg stands for anti grain geometry which is a high-performance library that produces attractive images.

```
import numpy as np
x = np.random.randn(10000)
ax = fig.add_subplot(111)
```

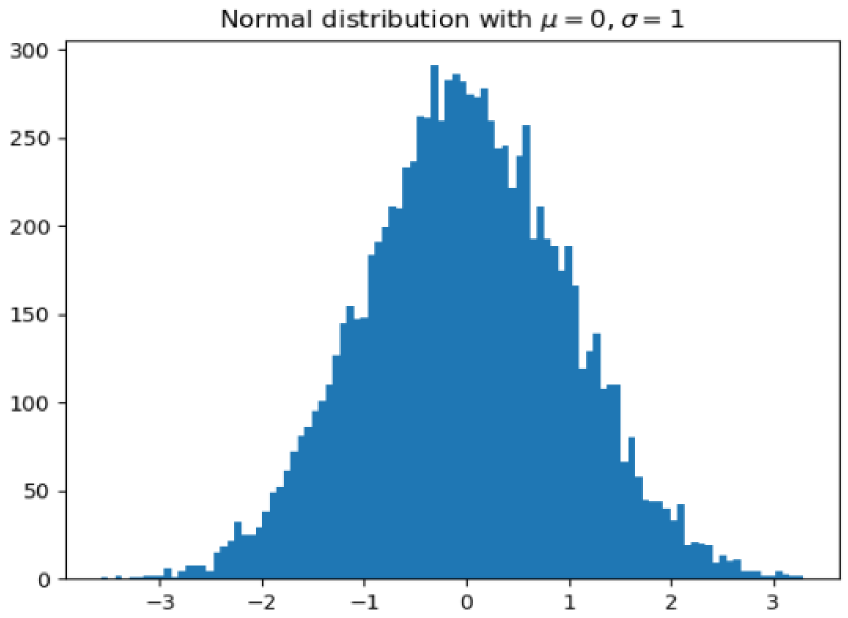
Then we import the Numpy library to generate the random numbers. Next we create an **axes artist***. The axes artist is added automatically to the figure container. Note here that (111) is from the MATLAB convention so it creates a grid with one row and one column and uses the first cell in that grid for the location of the new axes.

* Axes is the plotting area into which most of the objects go

Artist Layer

```
ax.hist(x, 100)
ax.set_title('Normal distribution with $\mu=0, \sigma=1$')
fig.savefig('matplotlib_histogram.png')
```

Finally we call the axes method hist, to generate the histogram. Hist creates a sequence of rectangle artists for each histogram bar and adds them to the axes container. Here 100 means create 100 bins. Finally, we decorate the figure with a title and we save it.



Pyplot

It was developed for scientists who are not professional programmers: the artist layer is syntactically heavy as it is meant for developers and not for individuals whose goal is to perform quick exploratory analysis of some data.

The scripting layer is comprised mainly of **pyplot** (plotlib.pyplot interface), which automates the process of defining a canvas and defining a figure artist instance and connecting them.

So let's see how the same code that we used earlier using the artist layer to generate a histogram of 10,000 random numbers would now look like.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.randn(10000)
plt.hist(x,100)
plt.title('Normal distribution with $\mu=0, \sigma=1$ ')
plt.savefig('matplotlib_histogram.png')
plt.show()
```

all the methods associated with creating the histogram and other artist objects (hist, title, show, …) are also part of the pyplot interface.

PyPlot

The Axes is one of the most important Artist objects: it contains most of the figure elements (Axis, Tick, Line2D, Text, Polygon) and sets the coordinate system.

From the example before we can see that we do not need to explicitly have an instance of a Figure, or an Axis to actually plot a figure using Matplotlib. Indeed, for each Axes graphic method (hist, set_title,…) there is a corresponding function in the matplotlib.pyplot module that performs that plot on the current axes, creating that axes (and its parent figure) if they do not exist yet.

Thus, the codes:

```
fig = Figure()
Canvas = FigureCanvas(fig)
ax = fig.add_subplot()
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
```

and

```
fig, ax = plt.subplots()              # Create a figure containing a single axes.
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])   # Plot some data on the axes.
```

```
plt.plot([1, 2, 3, 4], [1, 4, 2, 3])   # Matplotlib plot.
```

have the same effect

OO interface and pyplot interface

As noted above, there are essentially two ways to use Matplotlib using the pyplot library:

- Explicitly create figures and axes, and call methods on them (the "object-oriented (OO) style").
- Rely on pyplot to automatically create and manage the figures and axes, and use pyplot functions for plotting.

```
x = np.linspace(0, 2, 100)

# Note that even in the OO-style, we use `.pyplot.figure` to create
the figure.
fig, ax = plt.subplots() # Create a figure and an axes.
ax.plot(x, x, label='linear') # Plot some data on the axes.
ax.plot(x, x**2, label='quadratic') # Plot more data on the axes...
ax.plot(x, x**3, label='cubic') # ... and some more.
ax.set_xlabel('x label') # Add an x-label to the axes.
ax.set_ylabel('y label') # Add a y-label to the axes.
ax.set_title("Simple Plot") # Add a title to the axes.
ax.legend() # Add a legend.
```

```
x = np.linspace(0, 2, 100)

# Note that even in the OO-style, we use `.pyplot.figure` to create
the figure.
plt.plot(x, x, label='linear') # Plot some data on the (implicit) axes.
plt.plot(x, x**2, label='quadratic') # etc.
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()

plt.show()
```

Plotting Window toolbar

It shows the original plotting area.

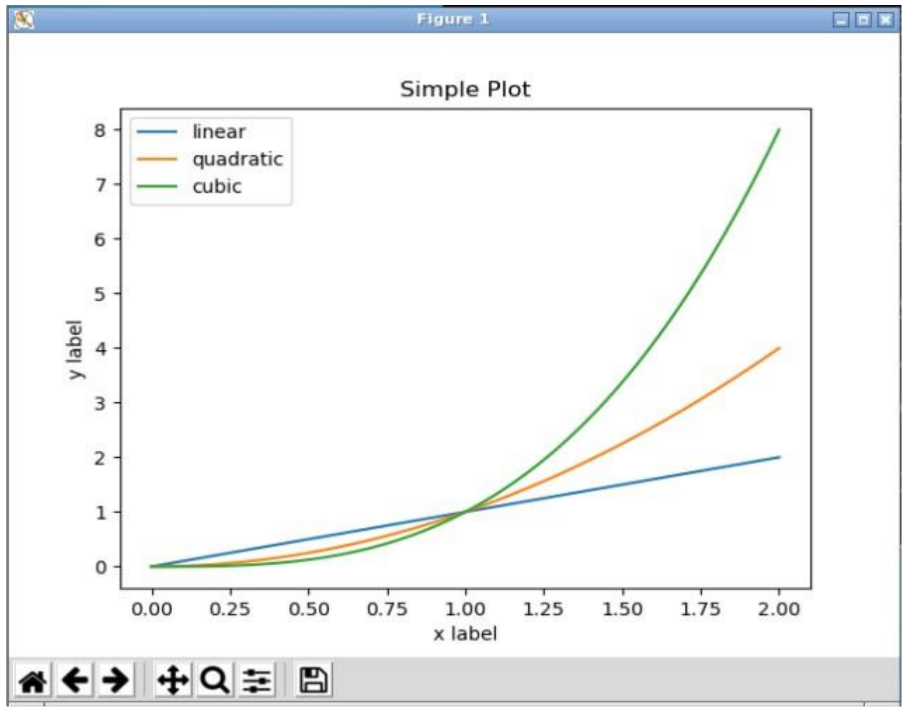
Undo/Redo visualization

Navigation inside the plot

Zoom of a portion of the plotting area

Subplot personalization

Saving / Exporting the figure



Some pyplot basic commands

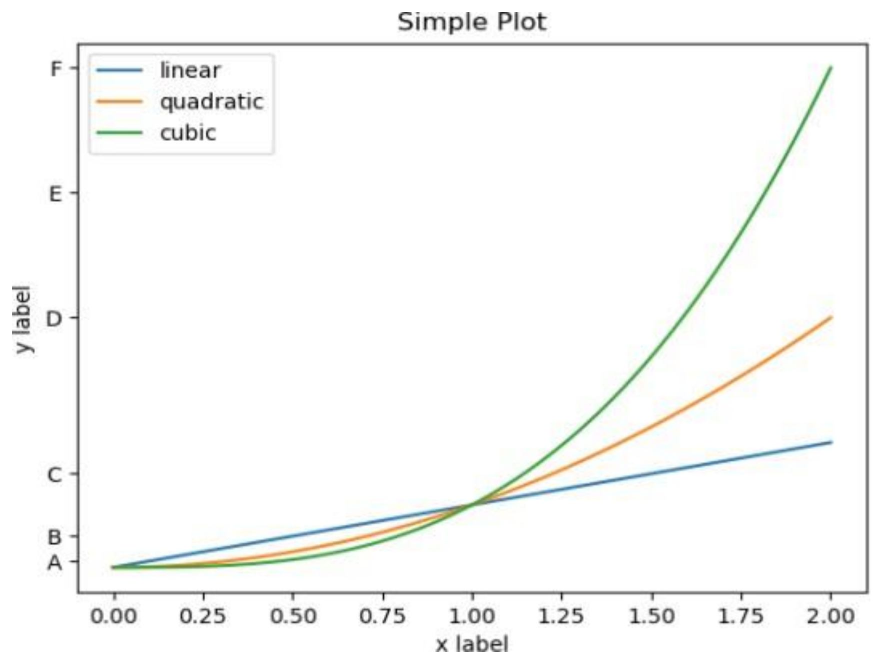
The pyplot.axis ([X_{min} , X_{max} , Y_{min} , Y_{max}]) gives the possibility of modifying the graph axes extremes. As default, the axes dimension will be the same of the range of the data.

The function pyplot.xticks() / yticks() allows for modifying the behaviour of pyplot for the tick marks of the axis (x/y)

es. plt.xticks([0.25*k for k in range (-4,5)]

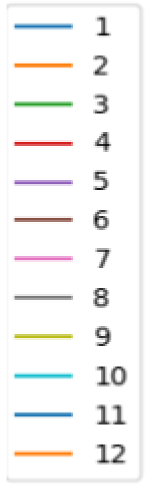
The first argument of pyplot.xticks() (or yticks()) may also be a list with numbers that determine the distance of the ticks: it is also possible to pass a list of labels, to be associated to the ticks of the related axis.

plt.yticks ([0.1, 0.5, 1.5, 4.0, 6.0, 8.0], ['A', 'B', 'C', 'D', 'E', 'F'])



Some pyplot basic commands

With pyplot we can also set the title, the labels for the x and y axes, the legend, and you can also have multiple graphs in the same plot (see the example before)



After 10 graph, the sequence of color repeats itself

However, besides the graph coordinates, with pyplot.plot() there is the possibility of passing additional arguments for modifying the behaviour of the graph, concerning the colour, or the line style:

plt.plot(x, 3+x**2, label='11', color = 'red', linestyle = '--')*

Possilbe arguments for linestyle are:

-	solid line style (default)	--	dashed line style
-.	dash-dot line style	:	dotted line style

* To plot it in a different figure, you can use the object-oriented style, or use plt.figure() with an index

Some pyplot basic commands

It's also possible to add some markers to the graph (for example some circle)

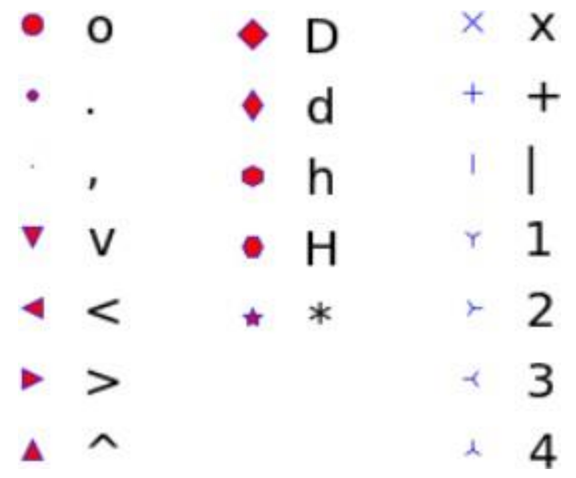
```
pyplot.plot(x, y, marker = 'o', color = 'red' )
```

A less verbose syntax may also be adopted:

```
pyplot.plot(x, y, 'r--o')
```

Inside the pyplot command, the *label* property is used to insert the description of the graph, which will be visualized in the legend (plt.legend())

pyplot also foresees some additional properties for handling the markers



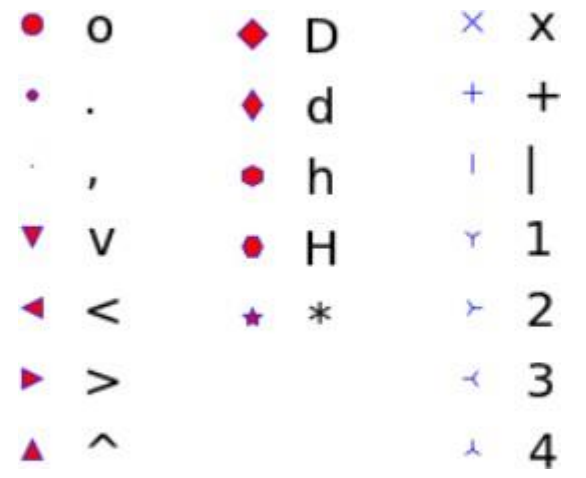
Some pyplot basic commands

E.g. *markersize = float*, gives the possibility to set the dimensions of the marker

markeredgewidth = float sets the dimension of the marker's edge

markerfacecolor = 'color string' sets the color of the internal face of the marker

markeredgecolor = 'color string' sets the color of the edge of the marker



By combining all these features, it's possible to obtain different kind of graphs (e.g. a scatter plot)

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-1.0,1.0,10000)
z = np.random.rand(10000)
y = x**2+z
plt.plot(x,y, linestyle='', marker=',', markerfacecolor='blue')
plt.show()
```

Some pyplot basic commands

A legend is made up of the following elements:

- legend entry

A legend is made up of one or more legend entries.
An entry is made up of exactly one key and one label.

- legend key

The colored/patterned marker to the left of each legend label.

- legend label

The text which describes the handle represented by the key.

- legend handle

The original object which is used to generate an appropriate entry in the legend.

Calling legend() with no arguments automatically fetches the legend handles and their associated labels. This functionality is equivalent to:

handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)

The get_legend_handles_labels() function returns a list of handles/artists which exist on the Axes which can be used to generate entries for the resulting legend - it is worth noting however that not all artists can be added to a legend

Some pyplot basic commands

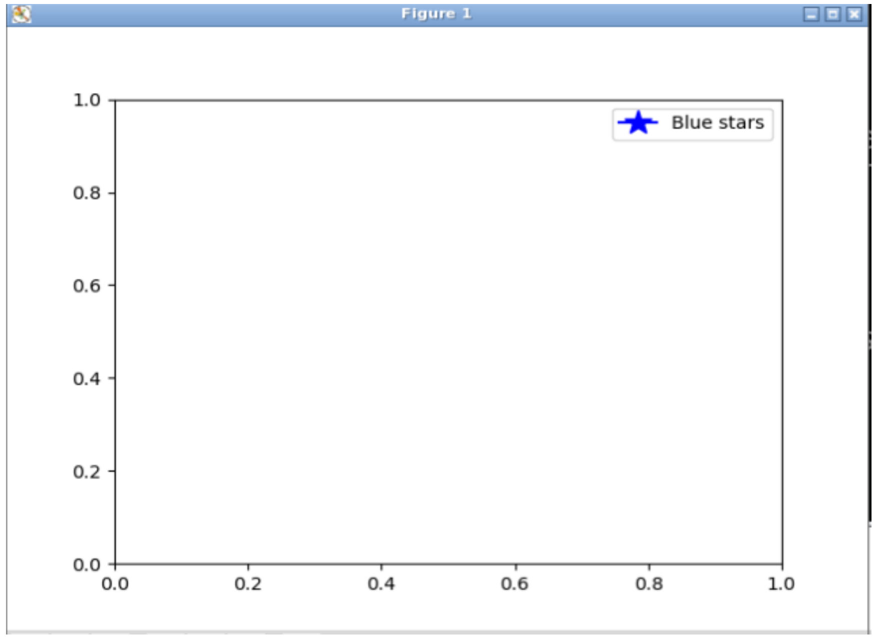
```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3], label='Line 2')
plt.plot([3, 2, 1], label='Line 1')
plt.legend()
plt.show()
```

For full control of what is being added to the legend, it is common to pass the appropriate handles directly to legend():

```
line_up, = plt.plot([1, 2, 3], label='Line 2')
line_down, = plt.plot([3, 2, 1], label='Line 1')
plt.legend(handles=[line_up, line_down])
```

Legend handles don't have to exist on the Figure or Axes in order to be used:

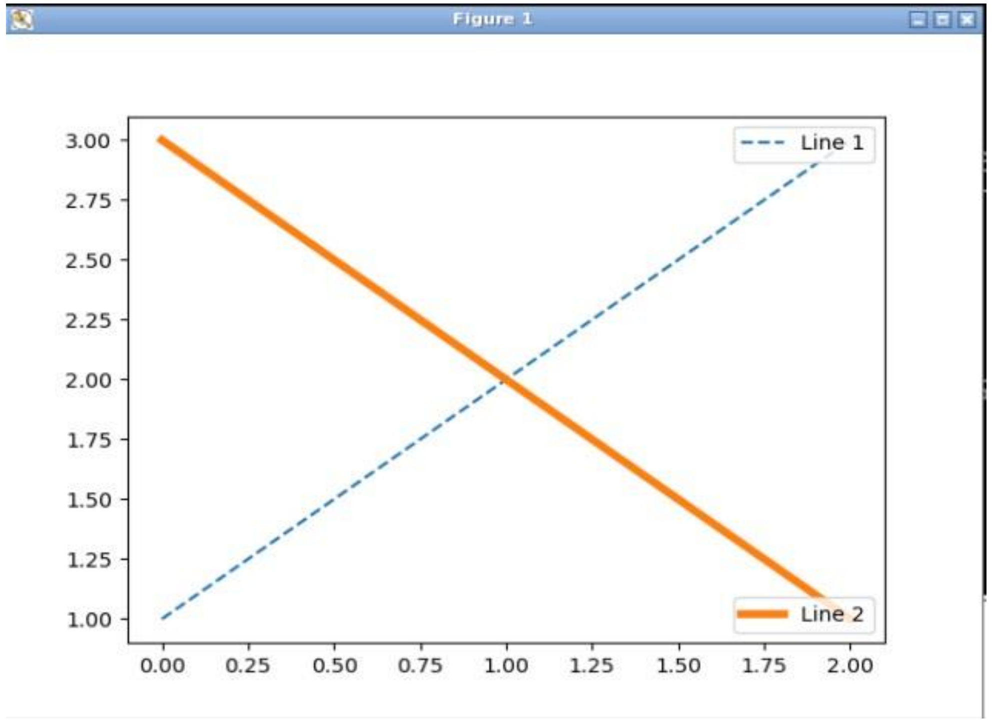
```
Import matplotlib.pyplot as plt
import matplotlib.lines as mlines
blue_line = mlines.Line2D([], [], color='blue', marker='*',markersize=15, label='Blue stars')
plt.legend(handles=[blue_line])
plt.show()
```



Some pyplot basic commands

Sometimes it is more clear to split legend entries across multiple legends. Whilst the instinctive approach to doing this might be to call the legend() function multiple times, you will find that only one legend ever exists on the Axes. This has been done so that it is possible to call legend() repeatedly to update the legend to the latest handles on the Axes. To keep old legend instances, we must add them manually to the Axes:

```
import matplotlib.pyplot as plt

line1, = plt.plot([1, 2, 3], label="Line 1", linestyle='--')
line2, = plt.plot([3, 2, 1], label="Line 2", linewidth=4)

# Create a legend for the first line.
first_legend = plt.legend(handles=[line1], loc='upper right')

# Add the legend manually to the current Axes.
plt.gca().add_artist(first_legend)

# Create another legend for the second line.
plt.legend(handles=[line2], loc='lower right')

plt.show()
```

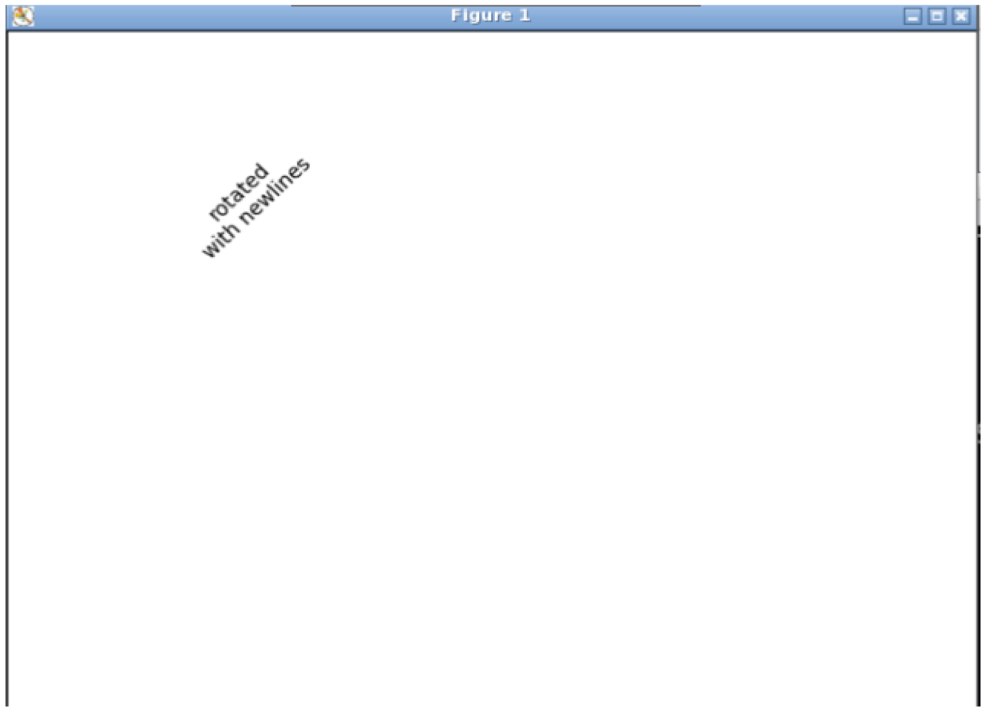


Some pyplot basic commands

Matplotlib has extensive text support, including support for mathematical expressions, truetype support for raster and vector outputs, newline separated text with arbitrary rotations, and unicode support.

Besides the common commands for text (titles, axes, ticks, …) it's possible to insert text at any place, with any rotation:

```
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])

ax.text(0.25, 0.75, 'rotated\nwith newlines',
        horizontalalignment='center',
        verticalalignment='center',
        rotation=45,
        transform=ax.transAxes)

plt.show()
```

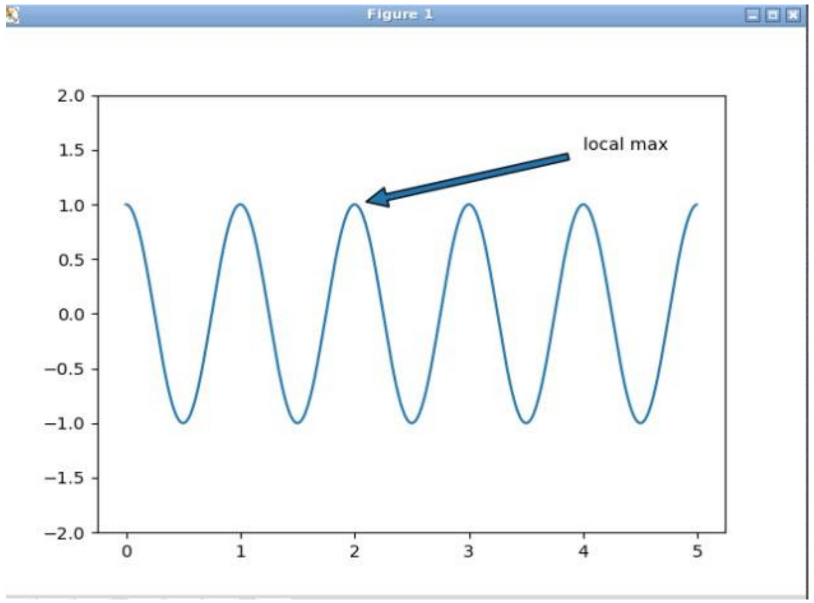


Some pyplot basic commands

A common use for text is to annotate some feature of the plot, and the annotate method provides helper functionality to make annotations easy. In an annotation, there are two points to consider: the location being annotated represented by the argument xy and the location of the text xytext. Both of these arguments are (x, y) tuples.

Moreover, the arrowprops property may be used to draw an arrow between the position *xy* and the *xytext*.

```
import matplotlib.pyplot as plt
import numpy as np

ax = plt.subplot()
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s)

plt.annotate('local max', xy=(2, 1), xytext=(4, 1.5),
arrowprops=dict(shrink=0.05))

plt.ylim(-2,2)
plt.show()
```

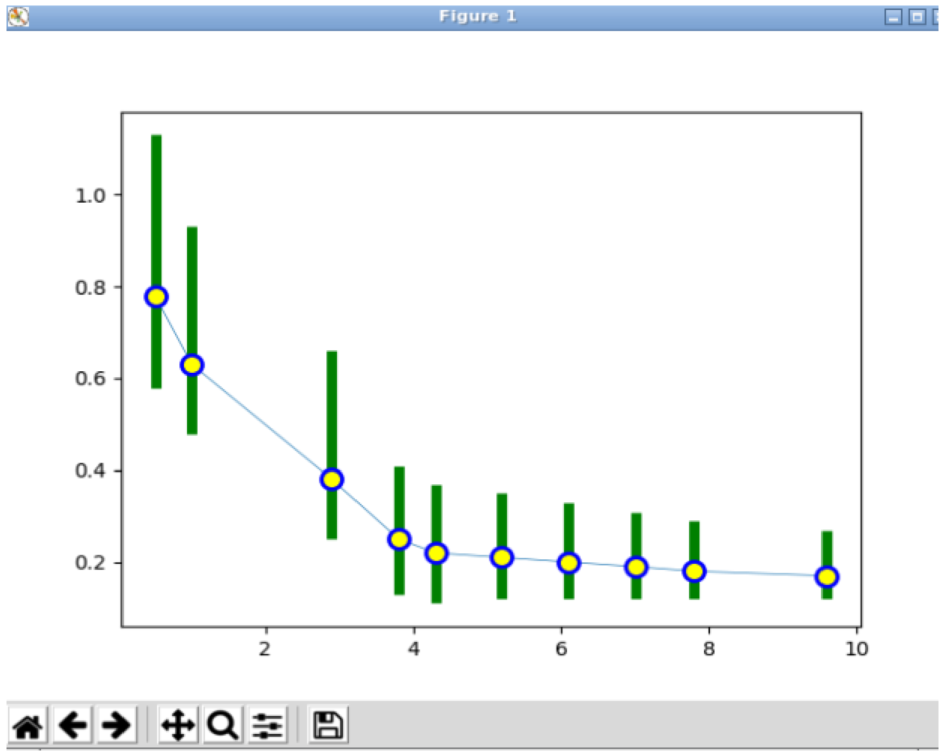


Some pyplot basic commands

Pyplot can also be used to plot different kind of graphs:

- *error bars*, where the yerr (o xerr bar) should be of the same size of x and y:

```
x = [0.5,1.0,2.9,3.8,4.3,5.2,6.1,7.0,7.8,9.6]
y = [0.78,0.63,0.38,0.25,0.22,0.21,0.20,0.19,0.18,0.17]
ym = [0.2,0.15,0.13,0.12,0.11,0.09,0.08,0.07,0.06,0.05]
yM = [0.35,0.3,0.28,0.16,0.15,0.14,0.13,0.12,0.11,0.1]
plt.errorbar(x,y, yerr=(ym,yM), linewidth=0.5,
marker='o',markeredgecolor='blue',
markeredgewidth=2,markerfacecolor='yellow',
ecolor='green',markersize=10,elinewidth=5.0)
plt.show()
```



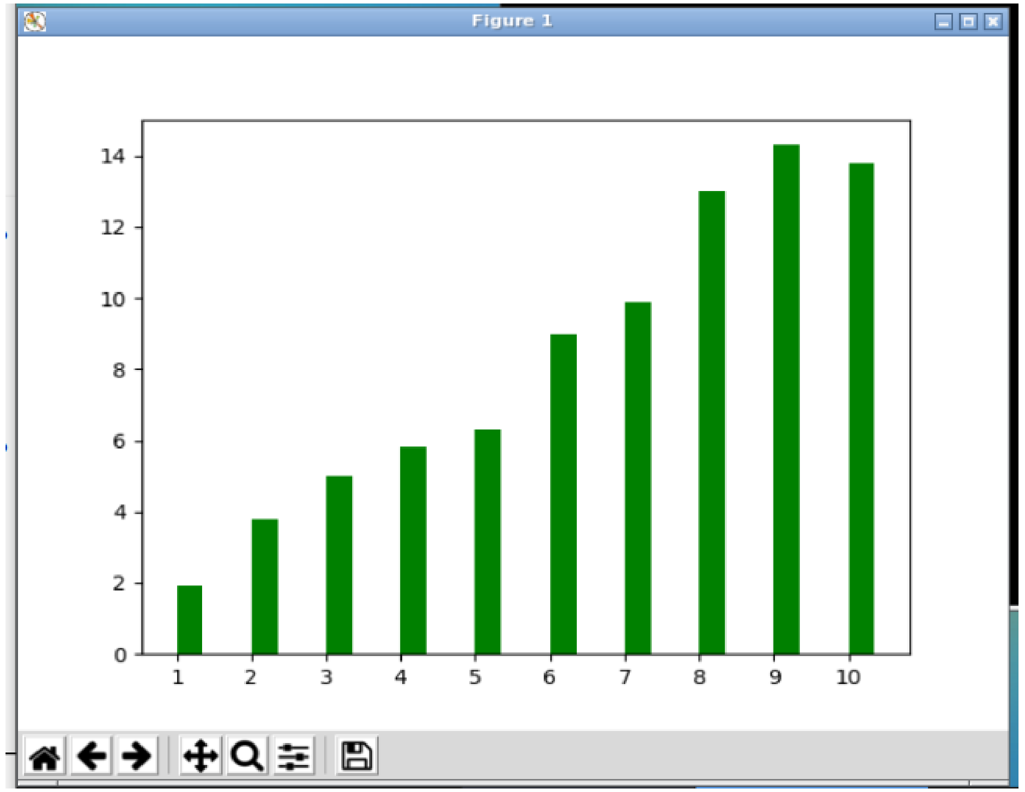
Some pyplot basic commands

- *bar plots*, where differently from pyplot.plot, in pyplot.bar the first argument (the x array) is mandatory.

By default, the x coordinate corresponds to the center of the bar; however the *align* property gives the possibility to modify this behaviour.

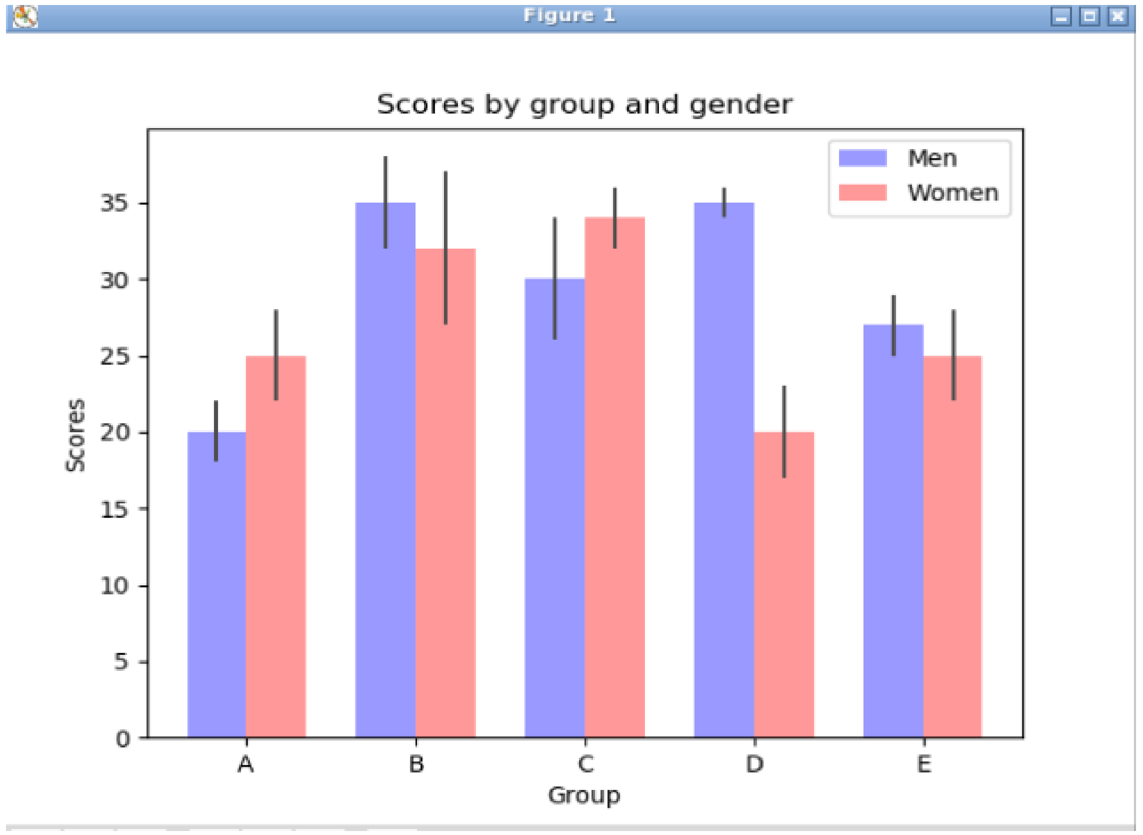
The argument of the property *width* may also be an array, to obtain bars of different dimensions

```
y = [1.9,3.8,5.0,5.8,6.3,9.0,9.9,13.0,14.3,13.8]
x = np.linspace(1,10,10)
plt.xticks(np.linspace(0,12,13))
plt.bar(x,y, align='edge', width=0.35, color='green')
```



Some pyplot basic commands

```
import numpy as np
import matplotlib.pyplot as plt
avr_men = (20,35,30,35,27); std_men = (2,3,4,1,2)
avr_women = (25,32,34,20,25); std_women = (3,5,2,3,3)
index = np.arange(5);
eConf = {'ecolor': '0.3'}
plt.bar(index, avr_men, align='edge', width=0.35, alpha=0.40,
color='b', yerr=std_men, error_kw=eConf, label='Men')
plt.bar(index+0.35, avr_women, align='edge', width=0.35,
alpha=0.40,
color='r', yerr=std_women, error_kw=eConf, label='Women')
plt.xlabel('Group')
plt.ylabel('Scores')
plt.title('Scores by group and gender')
plt.xticks(index + 0.35, ('A', 'B', 'C', 'D', 'E'))
plt.legend()
plt.show()
```



Some pyplot basic commands

We have already seen how we can build a histogram, with the *hist* method. Some additional info:

- the property *bins* in *plt.hist* sets the number of intervals of the graph: it may also be an array with the N+1 extremes of the intervals   (*plt.hist(data, bins=30)*)

- multiple sets of data may be shown in the same graph:

```
import numpy as np
import matplotlib.pyplot as plt
data = np.random.normal(loc=3, scale=0.5, size=10000)
plt.hist(data[:5000],bins=15,rwidth=0.5,  align='mid')
plt.hist(data[5000:],bins=15,rwidth=0.5,  align='right')
plt.show()
```

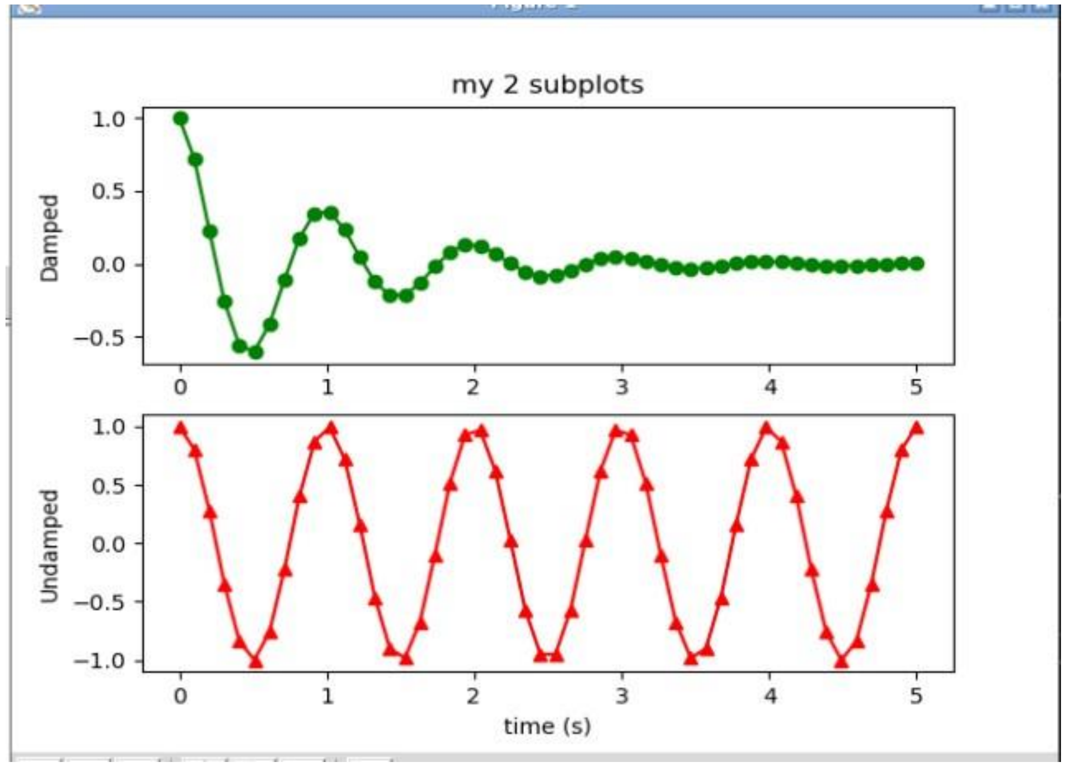
- The two series of data can be also stacked, by using a single call to *pyplot.hist:*

```
plt.hist([data[:5000],data[5000:]],bins=15,rwidth=0.5,  stacked='true')
```

Some pyplot basic commands

The *subplot* function allows for building multiple subplots in the same figure (very Matlab style). An example:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0.0, 5.0)
y1 = np.cos(2*np.pi*x)*np.exp(-x)
y2 = np.cos(2*np.pi*x)
plt.subplot(2, 1, 1)
plt.plot(x, y1, 'go-')
plt.title('my 2 subplots')
plt.ylabel('Damped')
plt.subplot(2, 1, 2)
plt.plot(x, y2, 'r^-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')
plt.show()
```

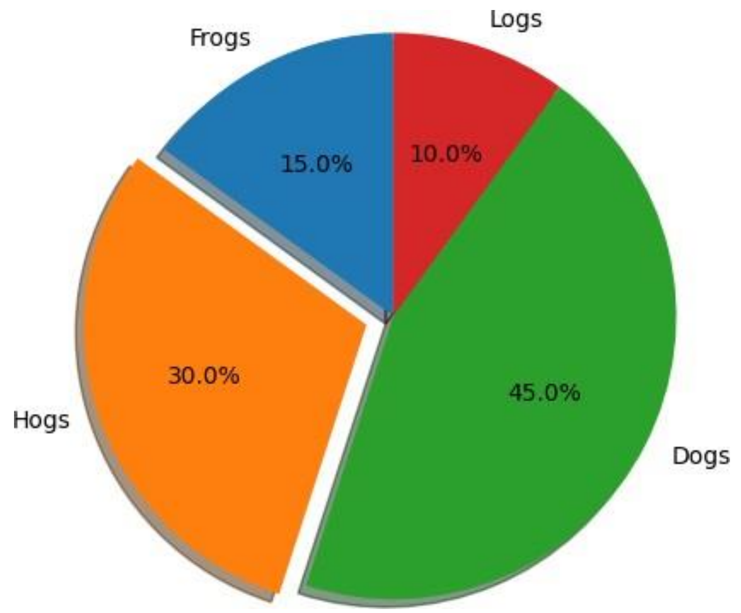


Some pyplot basic commands

Finally, pyplot may be used for generating different kind of graphs:

- *e.g. pie charts:*

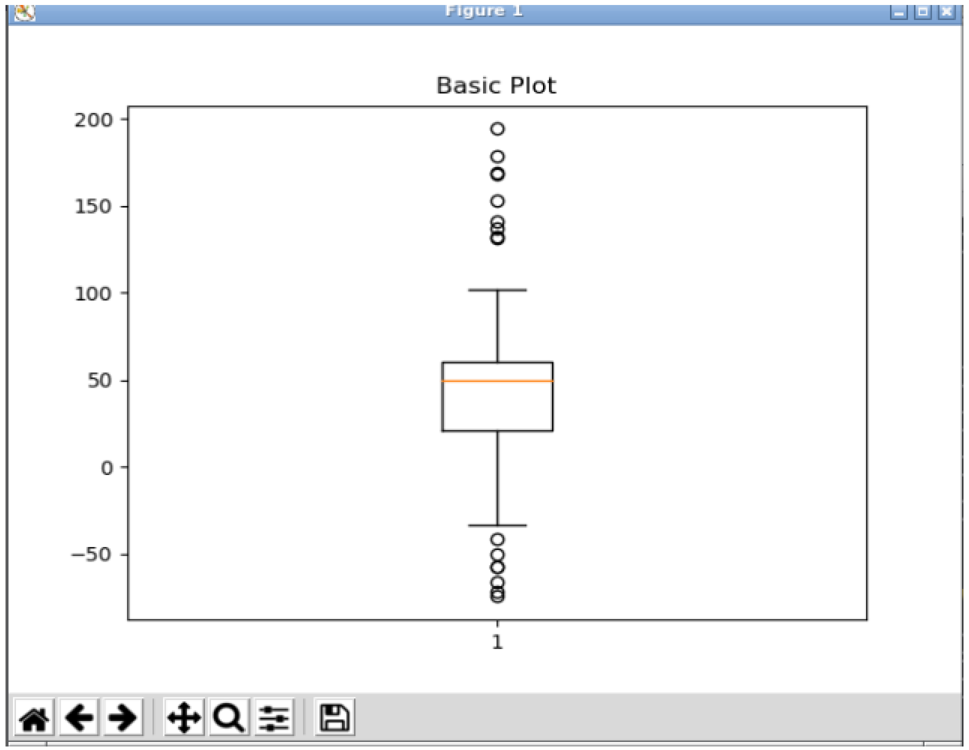
```
import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-clockwise:
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.

plt.show()
```



Some pyplot basic commands

- *boxplots:*

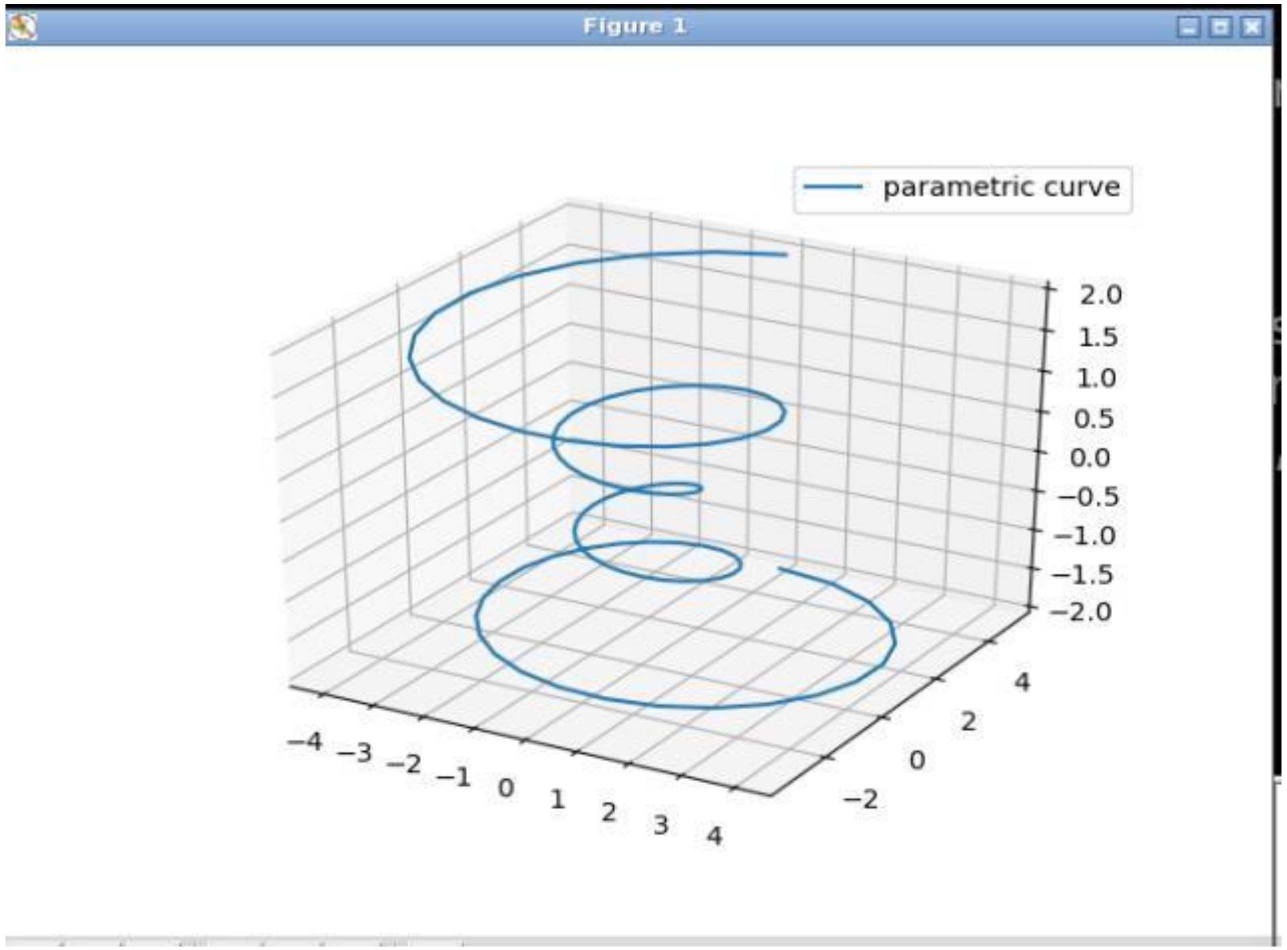
```
import numpy as np
import matplotlib.pyplot as plt

# fake up some data
spread = np.random.rand(50) * 100
center = np.ones(25) * 50
flier_high = np.random.rand(10) * 100 + 100
flier_low = np.random.rand(10) * -100
data = np.concatenate((spread, center, flier_high, flier_low))

plt.title('Basic Plot')
plt.boxplot(data)
plt.show()
```



Some pyplot basic commands

- *3D plots*

```
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
mpl.rcParams['legend.fontsize'] = 10
fig = plt.figure()
ax = fig.gca(projection='3d')
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z, label='parametric curve')
ax.legend()
plt.show()
```



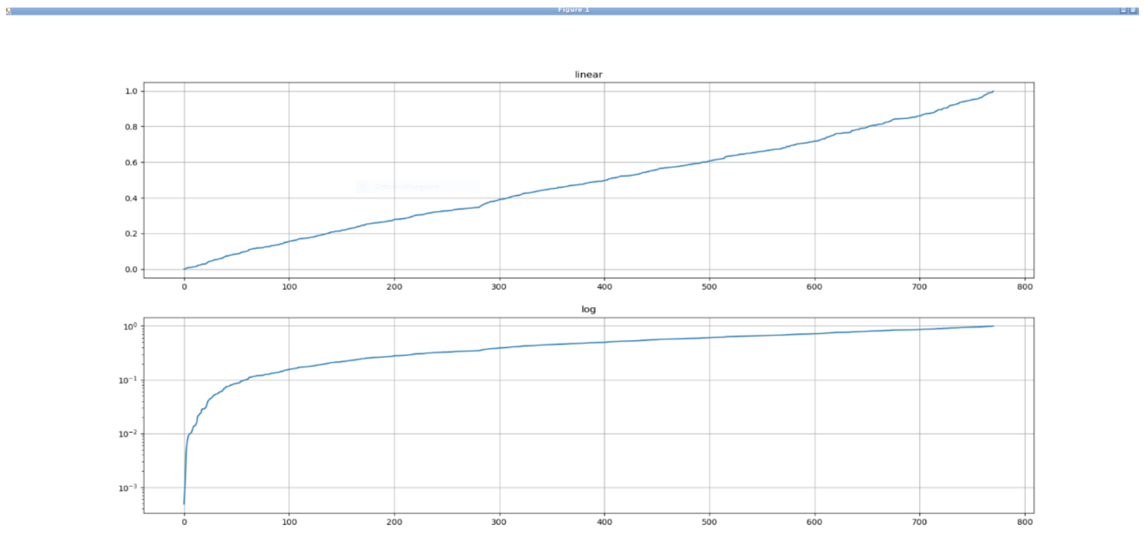
Some pyplot basic commands

matplotlib.pyplot supports not only linear axis scales, but also logarithmic and logit scales. This is commonly used if data spans many orders of magnitude. Changing the scale of an axis is easy:

plt.xscale('log')

```
import matplotlib.pyplot as plt
import numpy as np

y =
np.random.normal(loc=0.5,
scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

plt.figure()

plt.subplot(211)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)
```

```
#log
plt.subplot(212)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)

plt.show()
```



Some pyplot basic commands

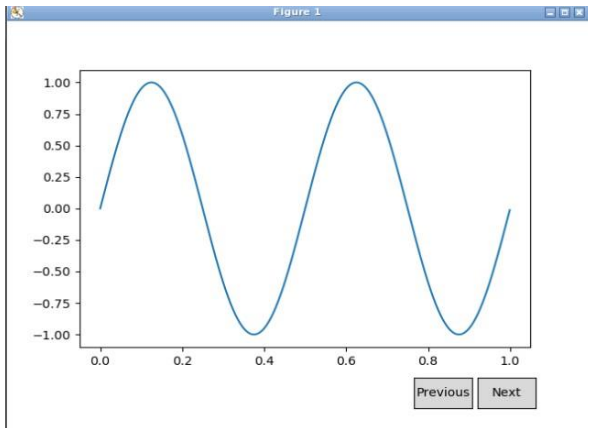
Matplotlib has basic GUI widgets that are independent of the graphical user interface you are using, allowing you to write cross GUI figures and widgets.

For example, it is possible to add Buttons:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import
Button

freqs = np.arange(2, 20, 3)

fig, ax = plt.subplots()
plt.subplots_adjust(bottom=0.2)

t = np.arange(0.0, 1.0, 0.001)
s = np.sin(2*np.pi*freqs[0]*t)
l, = plt.plot(t, s)
```

```
class Index:
   ind = 0

   def next(self, event):
                 self.ind += 1
                 i = self.ind % len(freqs)
                 ydata = np.sin(2*np.pi*freqs[i]*t)
                 l.set_ydata(ydata)
                 plt.draw()

   def prev(self, event):
                 self.ind -= 1
                 i = self.ind % len(freqs)
                 ydata = np.sin(2*np.pi*freqs[i]*t)
                 l.set_ydata(ydata)
                 plt.draw()
```

```
callback = Index()
axprev = plt.axes([0.7, 0.05, 0.1, 0.075])
axnext = plt.axes([0.81, 0.05, 0.1, 0.075])
bnext = Button(axnext, 'Next')
bnext.on_clicked(callback.next)
bprev = Button(axprev, 'Previous')
bprev.on_clicked(callback.prev)

plt.show()
```

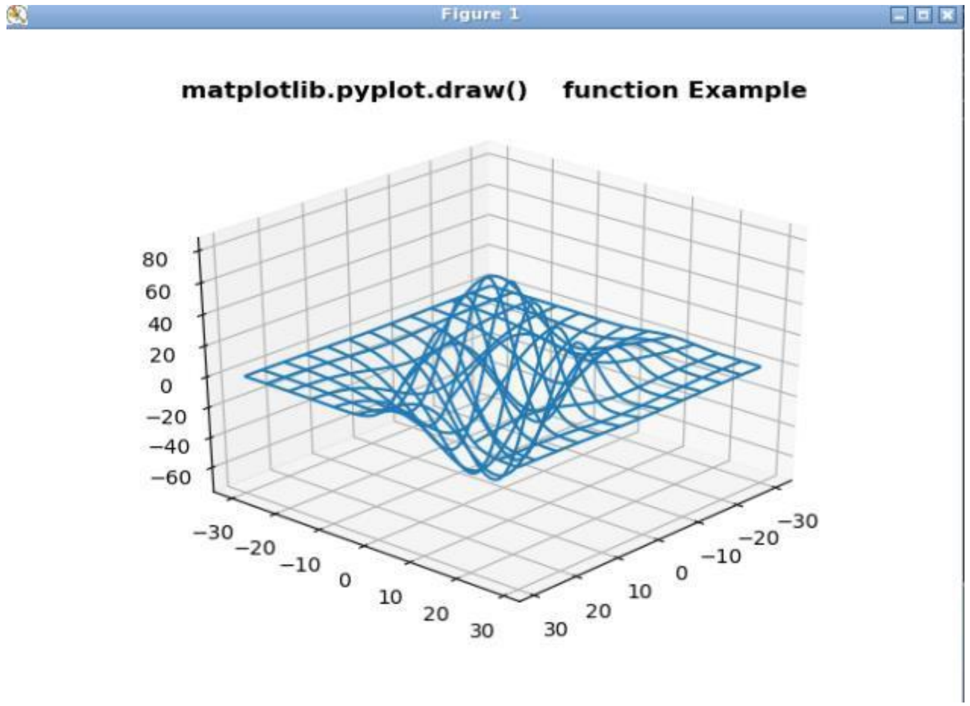


Some pyplot basic commands

The *draw* function allows for redrawing the current figure. This is used to update a figure that has been altered, but not automatically re-drawn. This allows you to work in interactive mode and, should you have changed your data or formatting, allow the graph itself to change.

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111, projection ='3d')

X, Y, Z = axes3d.get_test_data(0.1)
ax.plot_wireframe(X, Y, Z, rstride = 5, cstride = 5)

for angle in range(0, 360):
   ax.view_init(30, angle)
   plt.draw()
   plt.pause(.001)
   ax.set_title('matplotlib.pyplot.draw()\ function
   Example', fontweight ="bold")
```

Set the elevation and the azimuth of the axes in degrees



Some pyplot basic commands

As GUI widgets, we can also add a slider:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.widgets import Slider, Button

# The parametrized function to be plotted
def f(t, amplitude, frequency):
        return amplitude * np.sin(2 * np.pi * frequency * t)

t = np.linspace(0, 1, 1000)

# Define initial parameters
init_amplitude = 5
init_frequency = 3

fig, ax = plt.subplots()
line, = plt.plot(t, f(t, init_amplitude, init_frequency), lw=2)
ax.set_xlabel('Time [s]')

axcolor = 'lightgoldenrodyellow'
ax.margins(x=0)

# adjust the main plot to make room for the sliders
plt.subplots_adjust(left=0.25, bottom=0.25)

# Make a horizontal slider to control the frequency.
axfreq = plt.axes([0.25, 0.1, 0.65, 0.03], facecolor=axcolor)
freq_slider = Slider(ax=axfreq, label='Frequency [Hz]', valmin=0.1,
valmax=30, valinit=init_frequency)

axamp = plt.axes([0.1, 0.25, 0.0225, 0.63], facecolor=axcolor)
amp_slider = Slider(ax=axamp, label="Amplitude", valmin=0,
valmax=10, valinit=init_amplitude, orientation="vertical")

# The function to be called anytime a slider's value changes
def update(val):
        line.set_ydata(f(t, amp_slider.val, freq_slider.val))
        fig.canvas.draw_idle()

# register the update function with each slider
freq_slider.on_changed(update)
amp_slider.on_changed(update)

# Create a `matplotlib.widgets.Button` to reset the sliders to initial
values.
resetax = plt.axes([0.8, 0.025, 0.1, 0.04])
button = Button(resetax, 'Reset', color=axcolor, hovercolor='0.975')

def reset(event):
        freq_slider.reset()
        amp_slider.reset()

button.on_clicked(reset)

plt.show()
```

Interactive Mode

Matplotlib may be in "**interactive mode"** when plotting to the screen occurs, and when a script or shell session continues after a plot is drawn on the screen.

The default Boolean value is set by the matplotlibrc file, and may be customized like any other configuration parameter. It may also be set via ***matplotlib.interactive(),*** and its value may be queried via ***matplotlib.is_interactive().***

From an ordinary python prompt:

```
>>> import matplotlib.pyplot as plt
>>> plt.ion()
>>> plt.plot([1.6, 2.7])
```

This will pop up a plot window, but your terminal prompt will remain active, so that you can type additional commands such as:

Interactive Mode

```
plt.title("interactive test")
plt.xlabel("index")
```

On most interactive backends, the figure window will also be updated if you change it via the object-oriented interface. E.g. get a reference to the Axes instance, and call a method of that instance:

```
ax = plt.gca()
ax.plot([3.1, 2.2])
```

In the non-interactive scenario, [(plt.ioff()], the terminal command line is unresponsive after pyplot.show(); pyplot.show() blocks the input of additional commands until you manually kill the plot window.

Non-interactive mode delays all drawing until show() is called; this is more efficient than redrawing the plot each time a line in the script adds a new feature.

Styles

The style package adds support for easy-to-switch plotting "styles" with the same parameters as a matplotlib rc file (which is read at startup to configure Matplotlib).

There are a number of pre-defined styles provided by Matplotlib. For example, there's a pre-defined style called "ggplot" : to use this style, just add

```
import numpy as np
import matplotlib.pyplot as plt

plt.style.use('ggplot')
data = np.random.randn(5000)
plt.hist(data,100)
plt.show()
```

Integration with ROS and Animation

The easiest way to make a live animation in matplotlib is to use one of the Animation classes.

Animation in matplotlib makes use of 'blitting', a standard technique in computer graphics. The general idea is to take an existing bit map and then add one more artist on top. Thus, by managing a saved 'clean' bitmap, we can only re-draw the few artists that are changing at each frame and possibly save significant amounts of time. We can use blitting by passing **blit=True**

An example usage of FuncAnimation is the following:

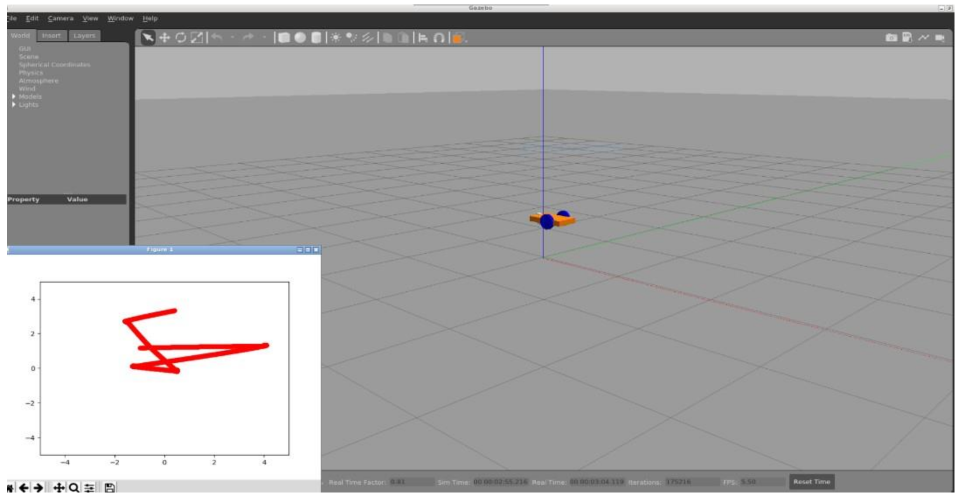
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import
FuncAnimation

fig, ax = plt.subplots()
xdata, ydata = [], []
ln, = plt.plot([], [], 'ro')

def init():
            ax.set_xlim(0, 2*np.pi)
            ax.set_ylim(-1, 1)
            return ln,

def update(frame):
    xdata.append(frame)
    ydata.append(np.sin(frame))
    ln.set_data(xdata, ydata)
    return ln,

ani = FuncAnimation(fig, update, frames=np.linspace(0, 2*np.pi, 128), init_func=init, blit=True)
plt.show()
```

Integration with ROS and Animation

Obviously this can be used for visualizing data published in a ROS topic:

```
import matplotlib.pyplot as plt
import rospy
import tf
from nav_msgs.msg import Odometry
from tf.transformations import quaternion_matrix
import numpy as np
from matplotlib.animation import FuncAnimation
```

```
rospy.init_node('odom_visualizer_node')
vis = Visualiser()
sub = rospy.Subscriber('/odom', Odometry, vis.odom_callback)

ani = FuncAnimation(vis.fig, vis.update_plot, init_func=vis.plot_init)
plt.show(block=True)
```



Assignment

- Continue the exercise with the Jupyter Notebook, by using the FuncAnimation for plotting the robot position

- Add also a plot for the number of reached/not-reached targets

- In the end, the notebook should have:
  - an interface to assign (or cancel) goals to the robot
  - a plot with the robot's position and targets' positions in the environment
  - a plot for the number of reached/not-reached targets