

Chương 3. Trực quan hóa dữ liệu

3.1	Đồ thị dạng đường	63
3.2	Đồ thị điểm rời rạc	69
3.3	Trực quan hóa lỗi	71
3.4	Đồ thị đường viền	74
3.5	Histograms và mật độ	77
3.6	Đồ thị ba chiều	80

Chương 3 có nội dung chính là tìm hiểu sâu rộng về **Matplotlib** trực quan hóa dữ liệu cho các loại biểu đồ khác nhau trong phân tích dữ liệu.

Matplotlib là thư viện trực quan hóa dữ liệu được xây dựng trên mảng NumPy và được thiết kế để hoạt động với ngăn xếp SciPy. Nó được John Hunter đưa ra vào năm 2002, cho phép vẽ sơ đồ kiểu MATLAB tương tác thông qua gnuplot từ dòng lệnh IPython.

Một trong những tính năng quan trọng nhất của Matplotlib là khả năng hoạt động tốt với đa nền tảng nhiều hệ điều hành, điều này dẫn đến một số lượng lớn người dùng với sự phổ biến trong thế giới khoa học dữ liệu sử dụng Python.

Cơ bản về Matplotlib

Trước khi đi sâu vào chi tiết tạo trực quan hóa bằng Matplotlib, có một số mẹo hữu ích nên biết về việc sử dụng Matplotlib như sau.

Nạp thư viện matplotlib

Sử dụng viết tắt tiêu chuẩn khi nạp thư viện Matplotlib:

```
import matplotlib as mpl  
import matplotlib.pyplot as plt
```

plt bao gồm nhiều hàm sẽ được sử dụng thường xuyên trong suốt chương này.

Thiết lập Style sử dụng Matplotlib

Thiết lập `plt.style` để chọn phong cách, các kiểu phù hợp cho các biểu đồ trực quan. Trong giáo trình này sẽ chọn kiểu `classic`, để đảm bảo rằng các biểu đồ được tạo ra sử dụng kiểu Matplotlib cổ điển ta thực hiện câu lệnh sau:

```
plt.style.use('classic')
```

Cách hiển thị các biểu đồ

Để phác họa và hiển thị một biểu đồ sử dụng Matplotlib có thể sử dụng một *script* - ngôn ngữ kịch bản, *IPython shell* hoặc *IPython notebook* như sau:

Trực quan dữ liệu bằng một Script

Sử dụng một tệp có tên *myplot.py* chứa các đoạn lệnh sau:

```
# ----- file: myplot.py -----
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```

Có thể chạy file trên từ giao diện dòng lệnh command-line prompt và kết quả sẽ hiển thị một biểu đồ trong cửa sổ mới bằng câu lệnh:

```
$ python myplot.py
```

Lệnh `plt.show()` thực hiện việc kết hợp nhiều thành phần của một biểu đồ và thường chỉ được sử dụng một lần ở cuối tập lệnh để hiển thị biểu đồ. Nhiều lệnh `show()` có thể dẫn đến sự phụ thuộc vào các thành phần của biểu đồ không thể đoán trước vì vậy nên hạn chế điều này.

Trực quan dữ liệu bằng IPython shell

IPython được xây dựng để hoạt động tốt với Matplotlib, để bật chế độ này, có thể sử dụng lệnh `%matplotlib` sau khi khởi động *ipython*:

```
In [1]: %matplotlib
Using matplotlib backend: TkAgg

In [2]: import matplotlib.pyplot as plt
```

Bất kỳ lệnh `plt plot` nào cũng sẽ mở cửa sổ biểu đồ ra và các lệnh khác có thể được chạy để cập nhật biểu đồ. Một số thay đổi (chẳng hạn như sửa đổi thuộc tính của các đường đã được vẽ) sẽ không được cập nhật; để bắt buộc cập nhật, sử dụng `plt.draw()`.

Trực quan dữ liệu bằng IPython notebook

IPython Notebook là một công cụ phân tích dữ liệu tương tác dựa trên trình duyệt có thể kết hợp mã nguồn, đồ họa, các phần tử HTML, v.v. vào một tài liệu thực thi duy nhất. Vẽ biểu đồ tương tác trong IPython Notebook có thể được thực hiện bằng lệnh `%matplotlib` và hoạt động theo cách tương tự với IPython shell. Trong *IPython Notebook*, cũng có tùy chọn nhúng đồ họa trực tiếp vào Notebook, với hai tùy chọn:

- `%matplotlib notebook` các biểu đồ tương tác được nhúng trong Notebook.
- `%matplotlib inline` các biểu đồ tĩnh được nhúng trong Notebook.

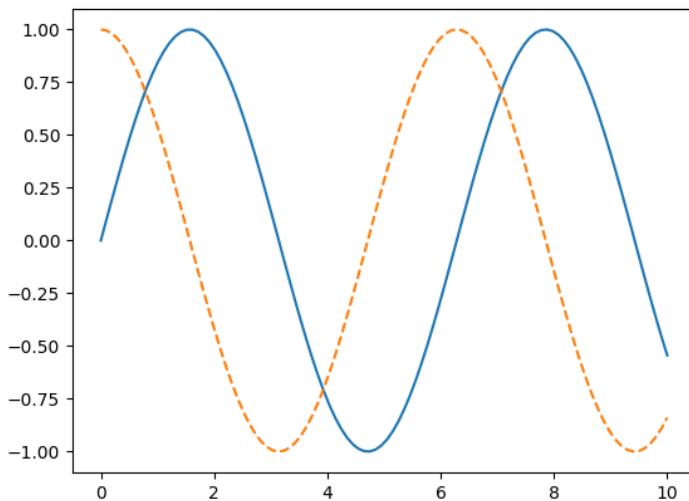
Để thống nhất trong toàn chương trình, các hướng dẫn trong chương trình lựa chọn `%matplotlib inline`:

```
In[3]: %matplotlib inline
```

Sau khi chạy lệnh này, bất kỳ ô - cell nào trong Notebook để tạo biểu đồ sẽ nhúng hình ảnh PNG trong kết quả Hình 3.1:

```
import numpy as np
x = np.linspace(0, 10, 100)

fig = plt.figure()
plt.plot(x, np.sin(x), 'r-')
plt.plot(x, np.cos(x), 'b--');
```



Hình 3.1: Ví dụ cơ bản về trực quan hóa.

Lưu ảnh vào tệp

Một tính năng của Matplotlib là khả năng lưu hình ảnh ở nhiều định dạng khác nhau, có thể lưu một hình bằng lệnh `savefig()`. Ví dụ: để lưu hình trước đó dưới dạng tệp PNG, có thể chạy lệnh sau:

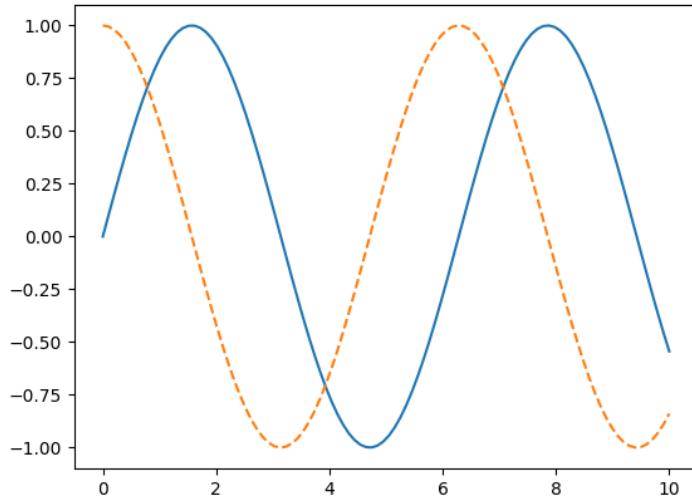
```
fig.savefig('my_figure.png')
```

Khi đó một tệp có tên `my_figure.png` được lưu trong thư mục làm việc hiện tại:

```
!ls -lh my_figure.png
-rw-r--r-- 1 mycomputer staff 37K Jul 24 11:46 my_figure.png
```

Để xác nhận rằng file ảnh vừa lưu chứa những gì, hãy sử dụng đối tượng *IPython Image* để hiển thị nội dung của tệp này (3.2):

```
from IPython.display import Image
Image('my_figure.png')
```



Hình 3.2: Kết xuất PNG của hình ảnh cơ bản.

Trong `savefig()`, định dạng tệp được suy ra từ phần mở rộng của tên tệp đã cho và có thể tìm thấy danh sách các loại tệp được hỗ trợ bằng cách sử dụng phương thức `canvas` sau của đối tượng hình:

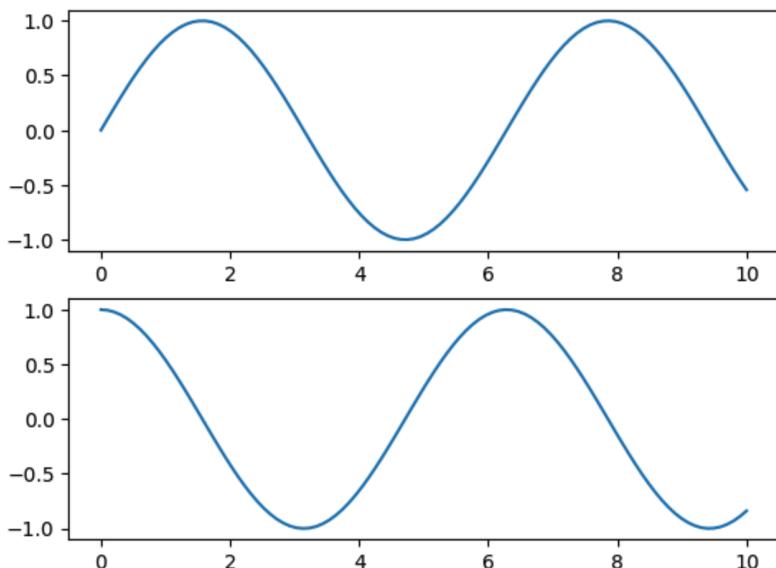
```
In[8]: fig.canvas.get_supported_filetypes()
Out[8]: {'eps': 'Encapsulated Postscript',
          'jpg': 'Joint Photographic Experts Group',
          'jpeg': 'Joint Photographic Experts Group',
```

```
'pdf': 'Portable Document Format',
'pgf': 'PGF code for LaTeX',
'png': 'Portable Network Graphics',
'ps': 'Postscript',
'raw': 'Raw RGBA bitmap',
'rgba': 'Raw RGBA bitmap',
'svg': 'Scalable Vector Graphics',
'svgz': 'Scalable Vector Graphics',
'tif': 'Tagged Image File Format',
'tiff': 'Tagged Image File Format',
'webp': 'WebP Image Format'}
```

Trực quan hóa nhiều biểu đồ trong một *Figure*

Một trong những điểm nổi bật của Matplotlib là có thể biểu diễn nhiều biểu đồ trên cùng một giao diện ảnh. Ví dụ dưới đây để vẽ hai biểu đồ trên cùng một ảnh (Hình 3.3):

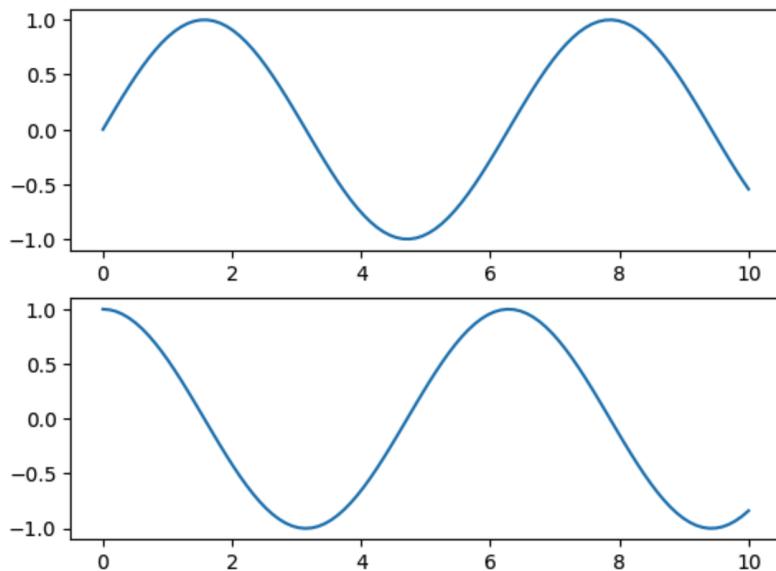
```
plt.figure() # Tao mot cua so figure de ve cac do thi
# Tao bieu do dau tien trong hai bieu do
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))
# Tao bieu do thu hai
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```



Hình 3.3: Sử dụng *subplot* để vẽ nhiều biểu đồ trong một *Figure*.

Trong trường hợp khi muốn hiển thị nhiều biểu đồ trong một hình cần sử dụng phương thức `subplots()`. Phương thức này lấy hai đối số `nrows` và `ncols` là số lượng hàng và số cột. Nó tạo ra hai đối tượng: `figure` và `axes` mà lưu trữ trong các biến `fig` và `ax` dưới đây có thể dùng để thay đổi các thuộc tính mức `figure` và `axes` tương ứng. Xem xét ví dụ kết hợp 2 biểu đồ (Hình 3.4) và 4 biểu đồ (Hình 3.51):

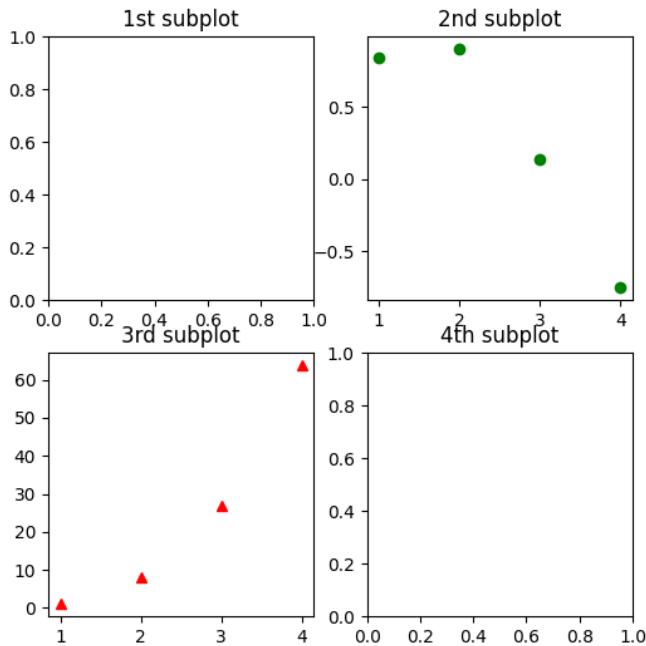
```
# Tao mot luoi cac o cho bieu do
# ax la mot mang bao gom hai phan tu Axes
fig, ax = plt.subplots(2)
# Goi phuong thuc plot() tren cac do thi tuong ung
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```



Hình 3.4: Sử dụng `subplot` để vẽ 2 biểu đồ trong một *Figure*.

```
x = np.arange(1,5)
y = x**3
# Tao mot luoi cac o cho bieu do
# ax la mot mang bao gom 4 phan tu Axes (2 hang, 2 cot)
fig, ax = plt.subplots(nrows = 2, ncols = 2, figsize = (6,6))

# Goi phuong thuc plot() tren cac do thi tuong ung
ax[0,1].plot(x, np.sin(x), 'go')
ax[1,0].plot(x, y, 'r^')
ax[0,0].set_title('1st subplot')
ax[0,1].set_title('2nd subplot')
ax[1,0].set_title('3rd subplot')
ax[1,1].set_title('4th subplot')
```



Hình 3.5: Sử dụng *subplot* để vẽ 4 biểu đồ trong một Figure.

Các phương pháp trực quan hóa dữ liệu để trình bày, mô tả và tóm tắt dữ liệu rất hữu ích trong việc quan sát dữ liệu. Phần tiếp theo của chương này sẽ tìm hiểu cách trình bày dữ liệu một cách trực quan bằng các biểu đồ sau:

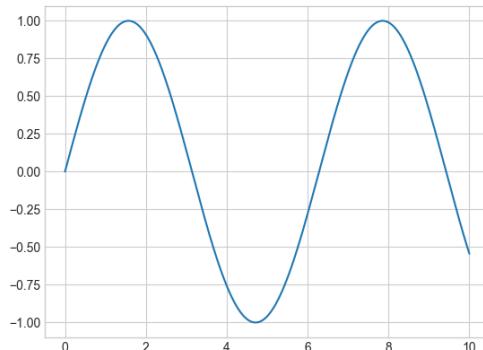
- Đồ thị dạng đường thẳng
- Đồ thị điểm rời rạc
- Trực quan hóa các biểu đồ lõi
- Đồ thị đường viền
- Histograms và mật độ
- Văn bản và chú thích
- Đồ thị ba chiều
- Đồ thị dữ liệu địa lý

3.1 Đồ thị dạng đường

Đồ thị dạng đường thẳng có lẽ là đồ thị đơn giản nhất trong tất cả các đồ thị, nó hình dung của một hàm duy nhất $y = f(x)$. Đầu tiên là việc thiết Notebook để vẽ đồ thị và nhập các hàm sẽ thực hiện bằng các câu lệnh dưới đây, sau đó tạo một hình - *figure* và một trục - *ax*:

```
In[1]: %matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
In[2]: fig = plt.figure()
ax = plt.axes()
```

Khi đã tạo một trục *ax*, có thể sử dụng hàm *ax.plot* để vẽ một số dữ liệu. Hãy bắt đầu với một hình *sin* đơn giản (Hình 3.6):

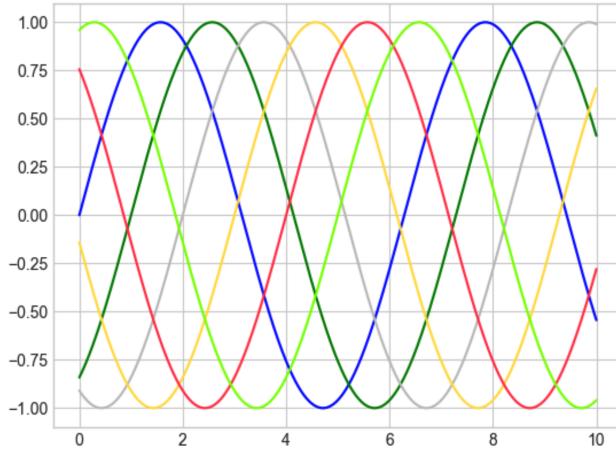


Hình 3.6: Đồ thị đường hình *sin*.

Điều chỉnh màu và kiểu dạng của đồ thị

Điều chỉnh có thể thực hiện đối với một biểu đồ là màu sắc (*colors*) và kiểu dáng của đồ thị (*styles*). Hàm *plt.plot()* chứa các đối số sử dụng để chỉ định các điều chỉnh này. Với màu sử dụng từ khóa *color* (Hình 3.7). Trong trường hợp không khai báo một màu cụ thể thì Matplotlib sẽ tự động lựa chọn một tập các màu mặc định.

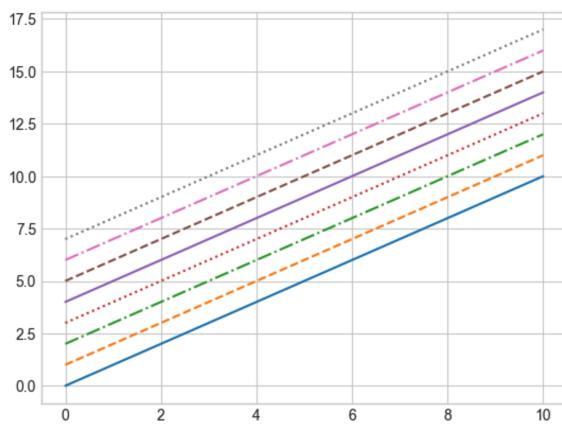
```
plt.plot(x, np.sin(x - 0), color='blue') #Mau cu theo ten
plt.plot(x, np.sin(x - 1), color='g') #Ma mau bang viet tat (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75') #Thang xam - Grayscale tu 0 den 1
plt.plot(x, np.sin(x - 3), color='#FFDD44') #Ma Hex (RRGGBB tu 00 - FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) #RGB, gia tri giua 0 va 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); #Ten mau HTML duoc ho tro
```



Hình 3.7: Thay đổi màu sắc sử dụng hàm *plot*.

Tương tự đối với *colors*, để thay đổi dạng của đường trong đồ thị cần thông qua từ khóa *linestyle* (Hình 3.8):

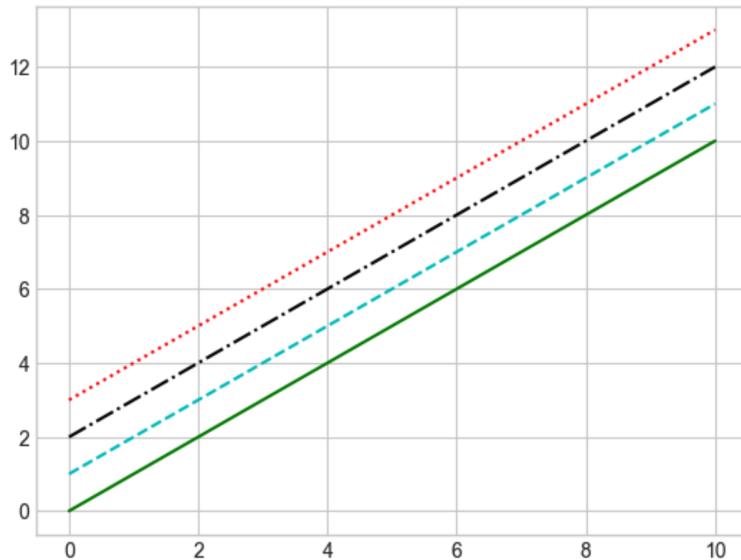
```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');
# Đang viết tắt, có thể sử dụng đoạn chương trình sau:
plt.plot(x, x + 4, linestyle='--') # solid
plt.plot(x, x + 5, linestyle='---') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':'); # dotted
```



Hình 3.8: Các kiểu dạng của đồ thị đường.

Nếu muốn cực kỳ ngắn gọn, các mã màu và kiểu đường này có thể được kết hợp thành một đối số mà không phải từ khóa duy nhất cho hàm `plt.plot()` (Hình 3.9): `linestyle` (Hình 3.8):

```
plt.plot(x, x + 0, '-g') # solid green
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot black
plt.plot(x, x + 3, ':r'); # dotted red
```



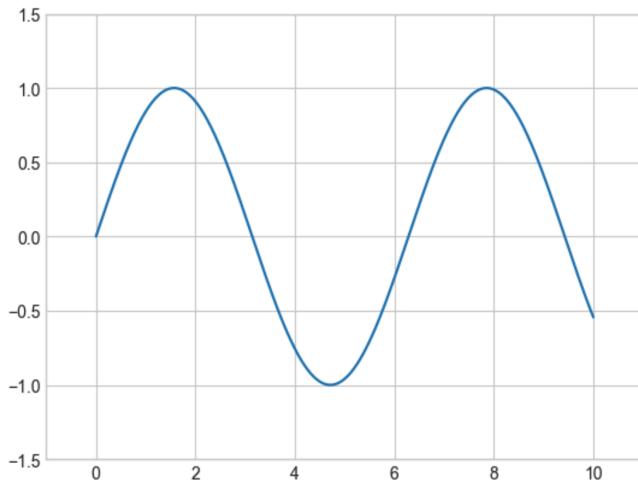
Hình 3.9: Điều chỉnh màu sắc và kiểu đồ thị bằng cách rút gọn.

Các mã màu gồm một ký tự tương ứng với các chữ viết tắt tiêu chuẩn trong hệ màu RGB (Red/Green/Blue) và CMYK (Cyan/Magenta/Yellow/black). Có nhiều từ khóa khác có thể được sử dụng để tinh chỉnh hình thức của đồ thị; để biết thêm chi tiết, có thể xem trong tài liệu của hàm `plt.plot()` bằng các công cụ trợ giúp của IPython (*help tool*).

Điều chỉnh giới hạn của trục đồ thị - Axes

Matplotlib thực hiện tốt việc chọn giới hạn biên trục toạ độ mặc định đồ thị, nhưng việc kiểm soát được các thông số này cũng rất hữu ích. Cách cơ bản nhất để điều chỉnh giới hạn trục là sử dụng các phương thức `plt.xlim()` và `plt.ylim()` (Hình 3.10):

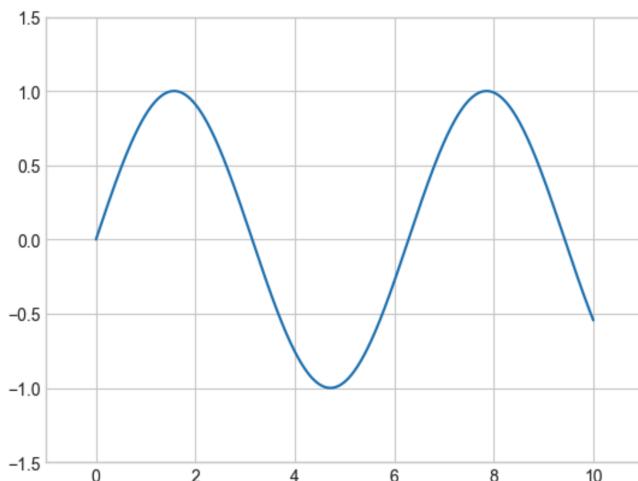
```
plt.plot(x, np.sin(x))
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```



Hình 3.10: Thay đổi biên của các trục tọa độ.

Phương thức `plt.axis()` cho phép đặt các giới hạn của trục x và trục y với một lần gọi, bằng cách truyền một danh sách `[xmin, xmax, ymin, ymax]` (Hình 3.11):

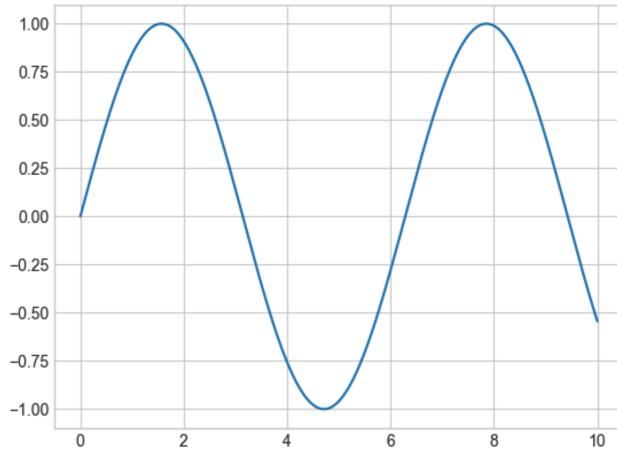
```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);
```



Hình 3.11: Thay đổi biên của các trục tọa độ bằng `plt.axis()`

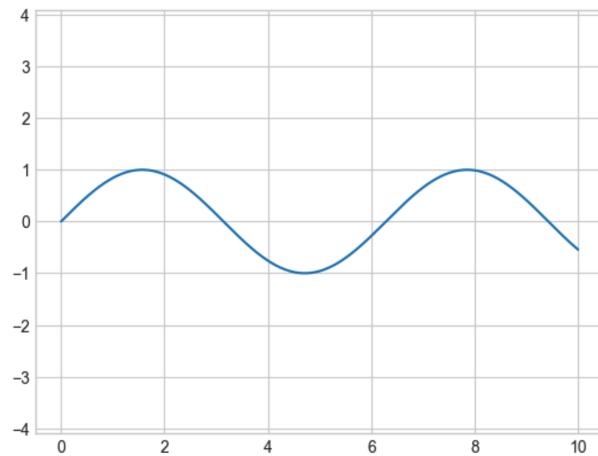
Phương thức `plt.axis()` còn cho phép thực hiện những việc như tự động loại bỏ phần giới hạn xung quanh đồ thị hiện tại mà không chứa đồ thị bằng sử dụng từ khoá "tight" (Hình 3.12) hay cho tỷ lệ các trục bằng nhau bằng "equal" (Hình 3.13):

```
plt.plot(x, np.sin(x))
plt.axis('tight');
```



Hình 3.12: Thay đổi biên của các trục tọa độ bằng *tight*.

```
plt.plot(x, np.sin(x))
plt.axis('equal');
```



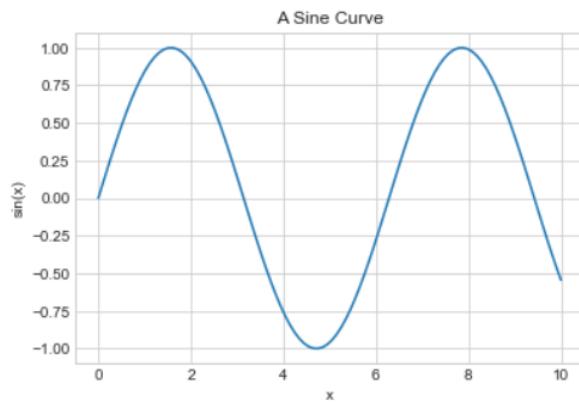
Hình 3.13: Thay đổi biên của các trục tọa độ bằng *equal*.

Để biết thêm thông tin về giới hạn trục và các tham số của phương thức `plt.axis()`, hãy tham khảo *docstring* của `plt.axis()`.

Tên đồ thị và nhãn các trục đồ thị

Nhãn các biểu đồ bao gồm tiêu đề tên đồ thị, nhãn trục và các lời chú giải. Sử dụng các phương thức dưới đây để có thể để giá trị cho tên đồ thị và nhãn trục (Hình 3.14):

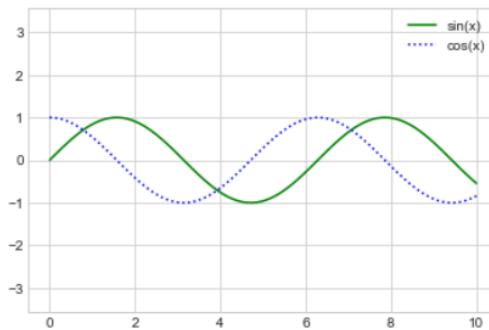
```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```



Hình 3.14: Ví dụ về đặt tên đồ thị và nhãn trục.

Khi hiển thị nhiều đường đồ thị cần thiết việc tạo chú giải nhãn cho từng loại đường, điều này được thực hiện thông qua phương thức `plt.legend()` dưới đây (Hình 3.15):

```
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.legend();
```

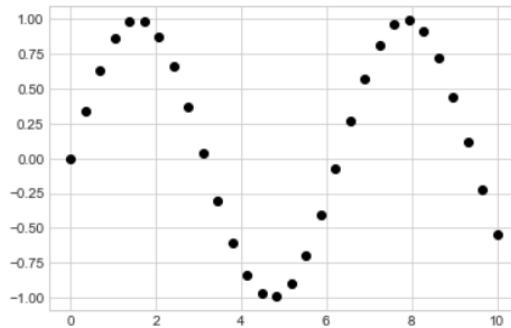


Hình 3.15: Ví dụ chú giải nhiều đường.

3.2 Đồ thị điểm rời rạc

Đồ thị điểm rời rạc các điểm thay vì nối với nhau bằng đoạn thì được biểu diễn riêng lẻ bằng dấu chấm, hình tròn hoặc hình dạng khác bằng phương thức `plt.plot/ax.plot`:

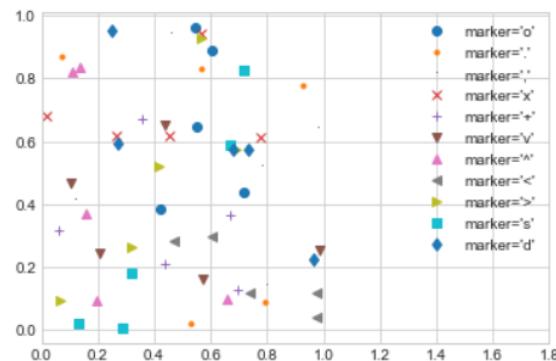
```
plt.plot(x, np.sin(x), 'o', color='black');
```



Hình 3.16: Đồ thị điểm rời rạc.

Đối số thứ ba trong `plt.plot` là đại diện cho ký hiệu sử dụng để vẽ đồ thị. Chỉ định tùy chọn như '-' và '_' cho đồ thị kiểu đường, các ký hiệu có sẵn trong tài liệu hướng dẫn sử dụng của `plt.plot`. Dưới đây là một số loại ký tự hay được sử dụng (Hình 3.17):

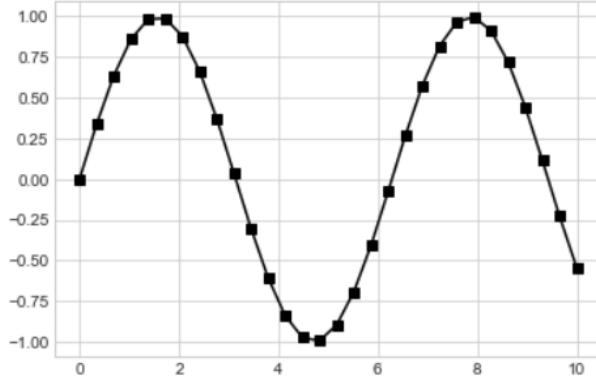
```
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker,
              label="marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```



Hình 3.17: Đồ thị điểm rời rạc.

Để có nhiều thuộc tính hơn, các mã ký tự có thể được sử dụng cùng với mã đường và mã màu để vẽ các điểm cùng với nhau trên một đường (Hình 3.18):

```
plt.plot(x, y, '-sk'); # line (-), square marker (s), black (k)
```

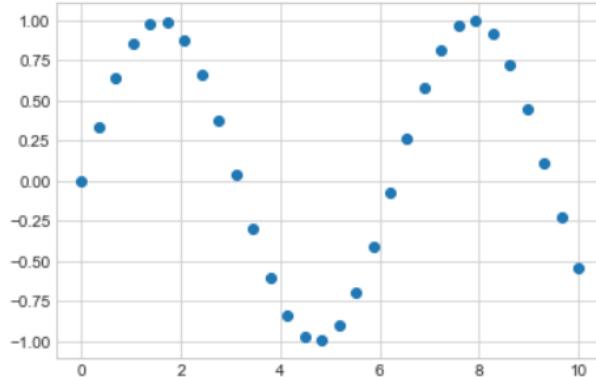


Hình 3.18: Đồ thị kết hợp điểm và đường.

Vẽ đồ thị điểm rời rạc bằng `plt.scatter`

Phương thức thứ hai, mạnh mẽ hơn để tạo đồ thị phân tán là hàm `plt.scatter`, được sử dụng rất giống với hàm `plt.plot`:

```
plt.scatter(x, y, marker='o');
```



Hình 3.19: Đồ thị điểm rời rạc sử dụng `plt.scatter`.

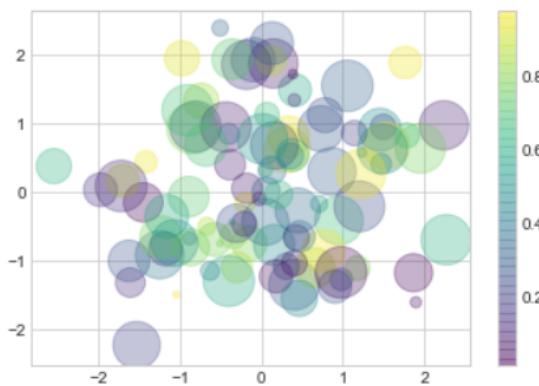
Sự khác biệt chính của `plt.scatter` so với `plt.plot` là sử dụng để tạo đồ thị với các điểm mà các thuộc tính của từng điểm là riêng lẻ và khác nhau (kích thước, màu của từng điểm, màu cạnh bao, vv..). Đồ thị dưới đây tạo ngẫu nhiên với các điểm có nhiều màu sắc và kích cỡ khác nhau. Để thấy rõ hơn các điểm trùng nhau sử dụng từ khóa `alpha` để điều chỉnh mức độ trong suốt của các điểm (Hình 3.20):

```

rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, cmap='viridis')
plt.colorbar(); # Hiển thị thang màu

```



Hình 3.20: Thay đổi kích thước, màu sắc và độ trong suốt ở các điểm phân tán.

So sánh *plot* với *scatter* về hiệu năng

Mặc dù nó không có sự khác biệt đối với lượng dữ liệu nhỏ, nhưng khi bộ dữ liệu lớn hơn vài nghìn điểm, *plt.plot* có thể hiệu quả hơn đáng kể so với *plt.scatter*. Lý do là *plt.scatter* hiển thị kích thước và/hoặc màu sắc khác nhau cho từng điểm, vì vậy phải thực hiện việc xây dựng từng điểm riêng lẻ. Trong khi *plt.plot*, các điểm về cơ bản luôn là bản sao của nhau, việc xác định hình thức các điểm chỉ được thực hiện một lần cho toàn bộ tập hợp dữ liệu. Với bộ dữ liệu lớn, *plt.plot* nên được ưu tiên hơn *plt.scatter*.

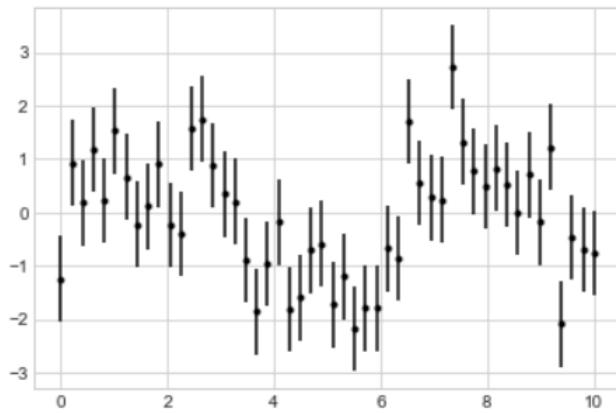
3.3 Trực quan hóa lỗi

Đối với bất kỳ phép đo khoa học nào, việc tính toán chính xác các lỗi cũng quan trọng không kém việc báo cáo chính xác con số. Khi trực quan hóa dữ liệu và các kết quả, việc hiển thị các lỗi một cách hiệu quả có thể giúp đồ thị truyền tải thông tin đầy đủ hơn nhiều.

Trực quan bằng thanh lỗi

Trục quan lỗi bằng thanh được tạo bằng một lệnh gọi hàm *Matplotlib* duy nhất (Hình 3.21):

```
x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy * np.random.randn(50)
plt.errorbar(x, y, yerr=dy, fmt='.k');
```



Hình 3.21: Trục quan bằng thanh lỗi.

Ở đây, *fmt* là một mã định dạng của các đường và điểm. Ngoài các tùy chọn cơ bản, đồ thị thanh lỗi còn có nhiều tùy chọn để tinh chỉnh đầu ra, làm cho các thanh lỗi có độ nhạt hơn các điểm (Hình 3.22):

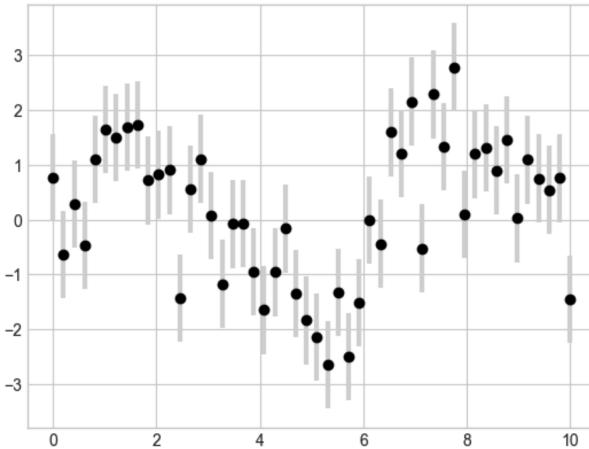
```
plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
              ecolor='lightgray', elinewidth=3, capsize=0);
```

Trục quan bằng miền lỗi liên tục

Trong một số trường hợp, mong muốn trực quan đồ thị lỗi trên miền liên tục, có thể kết hợp hàm *plt.plot* và *plt.fill_between*. Ví dụ minh họa dưới đây thực hiện hồi quy Gaussian đơn giản - *Gaussian process regression (GPR)*, sử dụng API Scikit-Learn như một phép đo lỗi trên miền liên tục như sau:

```
from sklearn.gaussian_process import GaussianProcessRegressor as GPR
from sklearn.gaussian_process.kernels import RBF

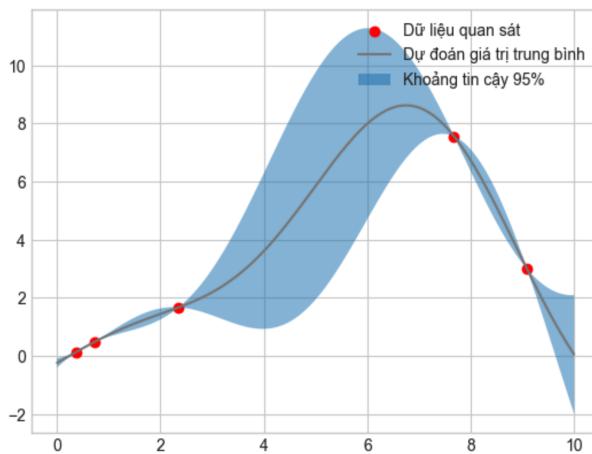
# Khoi tao gia tri cac diem du lieu
model = lambda x: x * np.sin(x)
xdata = np.linspace(0, 10, 1_000).reshape(-1, 1)
ydata = model(xdata)
```



Hình 3.22: Trực quan bằng thanh lõi.

```
# Tinh toán hồi quy Gaussian
rng = np.random.RandomState(1)
dtrain = rng.choice(np.arange(ydata.size), size=5)
X_train, y_train = xdata[dtrain], ydata[dtrain]

kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gp = GPR(kernel=kernel, n_restarts_optimizer=9)
gp.fit(X_train, y_train)
mean, std = gp.predict(X, return_std=True)
```



Hình 3.23: Trực quan miền lõi liên tục.

Ở đây X_{train} , y_{train} , $xdata$ và $mean$ để lấy mẫu sự phù hợp liên tục với dữ liệu, có

thể dùng hàm `plt.errorbar` như trên, nhưng nếu không muốn vẽ 1.000 điểm với 1.000 với thanh lỗi. Thay vào đó có thể sử dụng hàm `plt.fill_between` với màu để trực quan lỗi liên tục này (Hình 3.23):

```
plt.scatter(X_train, y_train, label="Đu lieu quan sat", color = 'red')
plt.plot(xdata, mean, label="Đu doan gia tri trung binh", color = 'gray')
plt.fill_between( xdata.ravel(), mean - 1.96 * std, mean + 1.96 * std,
                  alpha=0.5, label=r"Khoang tin cay 95%" )
plt.legend()
```

Lưu ý với hàm `fill_between` truyền một giá trị x được vẽ, sau đó là giá trị giới hạn dưới của y (*lower y-bound*), sau đó là giá trị giới hạn trên của y (*upper y-bound*) và kết quả là miền giữa các vùng này được lấp đầy. Hình 3.23 cung cấp một cái nhìn rất trực quan về thuật toán hồi quy Gaussian đang thực hiện: ở vùng gần điểm dữ liệu được quan sát, mô hình bị hạn chế mạnh và điều này phản ánh các lỗi mô hình nhỏ; ở những vùng cách xa điểm dữ liệu được đo, quan sát thì mô hình không bị hạn chế mạnh và sai số mô hình tăng lên.

3.4 Đồ thị đường viền

Đôi khi sẽ hữu ích khi hiển thị dữ liệu ba chiều ở không gian hai chiều bằng cách sử dụng các đường viền hoặc vùng mã màu chuyển tiếp khác nhau. Có ba hàm Matplotlib có thể hữu ích cho nhiệm vụ này: `plt.contour` cho các ô đường viền, `plt.contourf` cho các ô đường viền được tô màu và `plt.imshow` để hiển thị hình ảnh. Đầu tiên cần cách thiết lập *Notebook* để vẽ đồ thị và nhập các hàm sẽ sử dụng:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
import numpy as np
```

Trực quan hóa một hàm ba chiều

Ví dụ về một biểu đồ đường đồng mức trực quan hóa sử dụng hàm $z = f(x, y)$:

```
def f(x, y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

Tạo biểu đồ đường viền bằng hàm `plt.contour` với ba đối số: các giá trị x , giá trị y và giá trị z . Các giá trị x và y đại diện cho các vị trí trên biểu đồ và giá trị z sẽ được biểu thị bằng các mức đường viền. Sử dụng hàm `np.meshgrid` xây dựng các điểm dữ liệu hai chiều từ mảng một chiều:

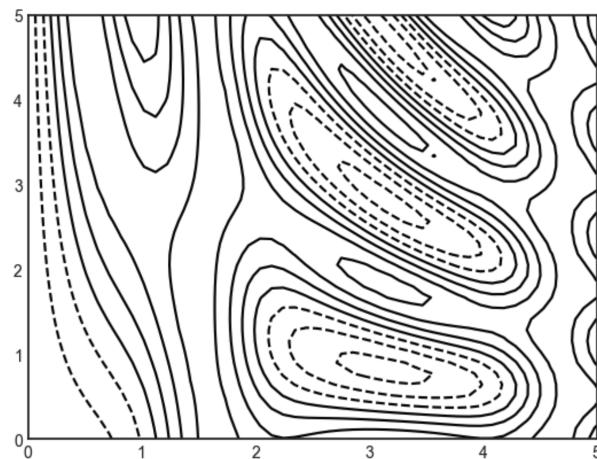
```
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)
```

$$Z=f(X, Y)$$

Một biểu đồ đường viền tiêu chuẩn chỉ có các đường viền đồng mức như Hình 3.24 dưới đây:

```
plt.contour(X, Y, Z, colors='black');
```



Hình 3.24: Trực quan hóa dữ liệu ba chiều với các đường viền.

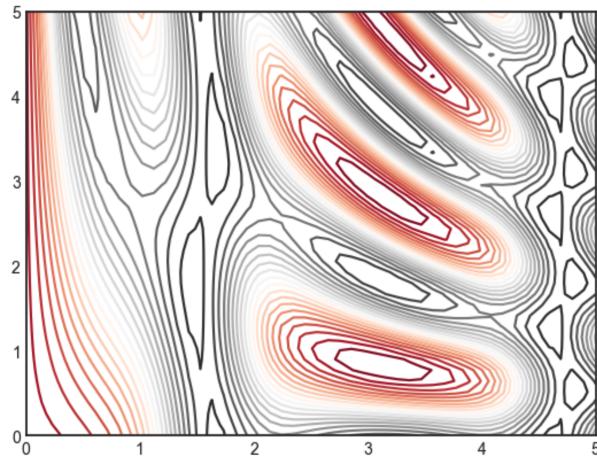
Theo mặc định khi một màu được sử dụng, các giá trị âm được biểu thị bằng các đường nét đứt và các giá trị dương được biểu thị bằng các đường liền nét. Ngoài ra, có thể đánh mã màu cho các dòng bằng cách chỉ định một bản đồ màu với đối số *cmap*. Ví dụ khi muốn vẽ nhiều đường hơn như khoảng 20 cách đều nhau trong phạm vi dữ liệu sử dụng câu lệnh sau (Hình 3.25):

```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```

Biểu đồ trông đẹp hơn so với đường viền một màu xám, nhưng khoảng cách giữa các dòng có thể làm mất tập trung. Điều này có thể thay đổi bằng cách chuyển sang biểu đồ đường viền được chuyển màu liên tục bằng cách sử dụng hàm *plt.contourf()* (chú ý *f* ở cuối), hàm này sử dụng phần lớn cú pháp giống như *plt.contour()*. Ngoài ra, thêm một lệnh *plt.colorbar()*, lệnh này sẽ tự động tạo một trực bối cảnh với thông tin màu được gắn nhãn cho biểu đồ (Hình 3.26):

```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();
```

Một vấn đề tiềm ẩn với cách trực quan hóa này là các bước màu rời rạc mà không mượt mà. Điều này có thể khắc phục bằng cách đặt số lượng đường viền với số lượng lớn, nhưng nó dẫn đến kém hiệu quả do Matplotlib phải hiển thị nhiều bước. Cách tốt hơn để xử lý việc này là sử dụng hàm *plt.imshow()*, hàm này biểu diễn lưới dữ liệu hai chiều dưới dạng một hình ảnh. Hình 3.27 là kết quả trực quan hóa bằng mã lệnh dưới



Hình 3.25: Trực quan hóa dữ liệu ba chiều với các đường viền nhiều màu và nhiều mức.

đây:

```
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
           cmap='RdGy', interpolation="bicubic")
plt.colorbar()
```

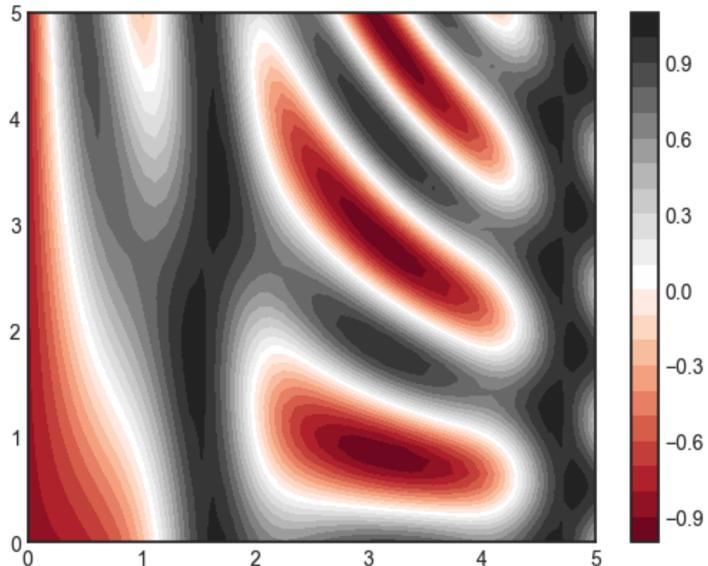
Các tham số trong *imshow()* bao gồm:

- *plt.imshow()* nhận phạm vi các giá trị $[xmin, xmax, ymin, ymax]$ của hình ảnh trên biểu đồ.
- *plt.imshow()* theo mặc định tuân theo định nghĩa mảng hình ảnh tiêu chuẩn trong đó điểm gốc nằm ở phía trên bên trái, không phải ở phía dưới bên trái như trong hầu hết các biểu đồ đường viền. Điều này phải được thay đổi khi hiển thị dữ liệu dạng lưới.
- *plt.imshow()* sẽ tự động điều chỉnh tỷ lệ khung hình trực để khớp với dữ liệu đầu vào. Ngoài ra, có thể sử dụng Phép nội suy hai khối - *Bicubic interpolation* được sử dụng khi phóng to ảnh vì thực tế các biểu đồ có xu hướng pixel mờ hơn sẽ được sử dụng yêu thích sử dụng nhiều hơn.

Đối với đối với biểu đồ đường viền cần kết hợp với các đường đồng mức như trực quan của Hình 3.28. Biểu đồ này sử dụng một ảnh nền trong suốt một phần (với độ trong suốt được thiết lập thông qua tham số *alpha*) và vẽ chồng lên các đường viền đồng mức bằng nhãn trên chính các đường viền đó (sử dụng hàm *plt.clabel()*):

```
contours = plt.contour(X, Y, Z, 3, colors='black')
plt.clabel(contours, inline=True, fontsize=8)

plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower', cmap='RdGy',
           alpha=0.5, interpolation="bicubic")
```



Hình 3.26: Trực quan hóa dữ liệu các đường viền chuyển màu liên tục.

```
plt.colorbar();
```

Sự kết hợp của ba hàm `plt.contour`, `plt.contourf` và `plt.imshow` tạo ra khả năng gần như vô hạn để hiển thị loại dữ liệu ba chiều trong một đồ thị đường viền hai chiều. Để biết thêm các tùy chọn có sẵn trong các hàm này, hãy tham khảo tài liệu hướng dẫn của các hàm trên.

3.5 Histograms và mật độ

Histograms là một biểu đồ đơn giản để hiểu tập dữ liệu, đồ thị cơ bản được trực quan hóa bằng một dòng mã lệnh, sau khi nhập một tập dữ liệu như sau:

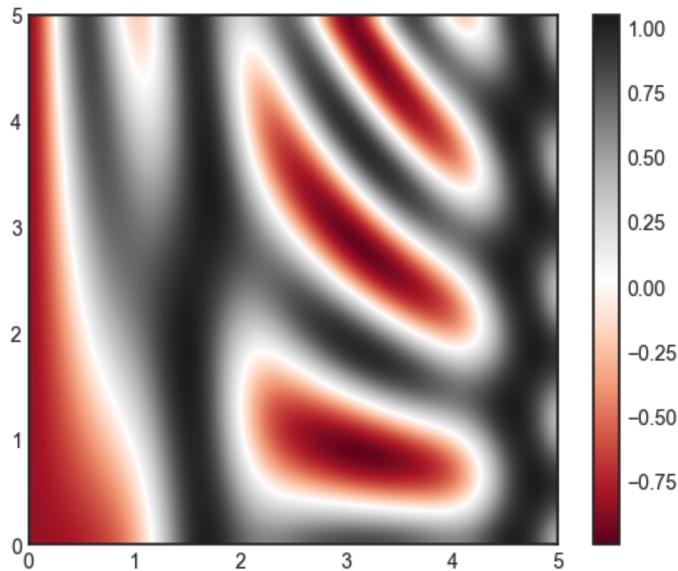
```
data = np.random.randn(1000)
plt.hist(data);
```

Hàm `hist()` có nhiều tùy chọn để điều chỉnh dưới đây là một ví dụ về đồ thị Histogram tùy chỉnh hơn (Hình 3.30):

```
plt.hist(data, bins=30, alpha=0.5, histtype='stepfilled',
          color='steelblue', edgecolor='none');
```

Có thể kết hợp `histtype='stepfills'` cùng với hệ số trong suốt `alpha` khi so sánh biểu đồ của nhiều dữ liệu phân phối khác nhau:

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
```



Hình 3.27: Trực quan hóa dữ liệu các đường viền liên tục.

```
kwargs = dict(histtype='stepfilled', alpha=0.3, bins=40)
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
```

Đồ thị histograms hai chiều và biểu đồ mật độ

Giống như tạo biểu đồ trong một chiều bằng cách chia số dòng thành các ngăn. Sử dụng dữ liệu gồm x và y được rút ra từ phân phối Gaussian đa biến như sau:

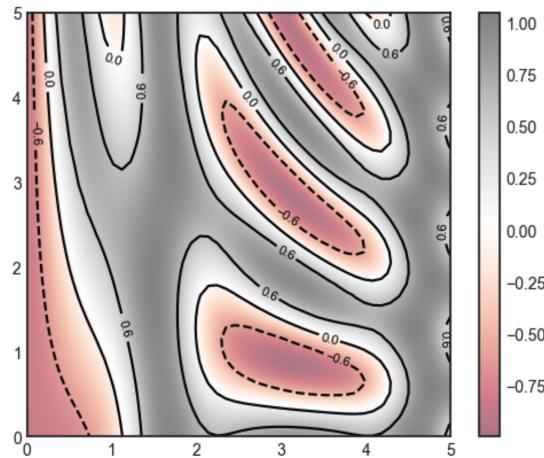
```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

Một cách đơn giản để trực quan hóa histograms hai chiều là sử dụng hàm `plt.hist2d`:

```
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```

Biểu đồ histograms hai chiều tạo ra một dãy các ô vuông trên các trục, ngoài ra có thể khác sử dụng các ô vuông này thành hình lục giác đều cho tự nhiên hơn. Với mục đích này, Matplotlib cung cấp một gói `plt.hexbin`, đại diện cho một tập dữ liệu hai chiều được đánh dấu trong một lưới các hình lục giác (Hình 3.33):

```
plt.hexbin(x, y, gridsize=30, cmap='Blues')
cb = plt.colorbar(label='count in bin')
```



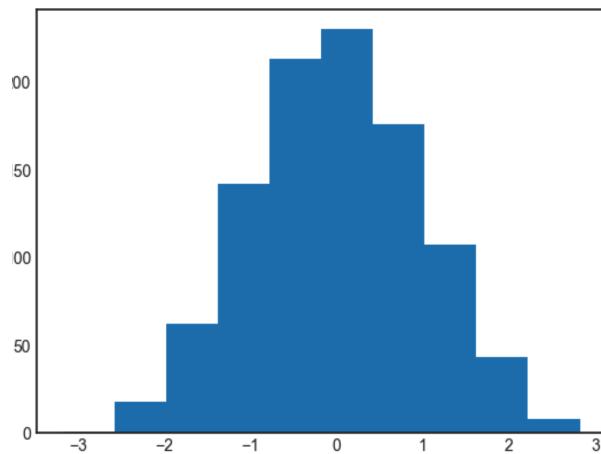
Hình 3.28: Trực quan hóa dữ liệu các đường viền liên tục và đường đồng mức.

`plt.hexbin` có một số tùy chọn khác bao gồm khả năng chỉ định trọng số cho từng điểm và thay đổi đầu ra trong mỗi ngăn thành bất kỳ tổng hợp *NumPy* nào (giá trị trung bình của trọng số, độ lệch chuẩn của trọng số, v.v.).

Ước tính mật độ nhân

Một phương pháp phổ biến khác để đánh giá mật độ theo nhiều chiều là ước tính mật độ nhân (*KDE*). KDE có thể được coi là một cách để “làm mờ” các điểm trong không gian và thêm vào kết quả để có được một hàm làm mịn dữ liệu. Có cách triển khai KDE cực kỳ nhanh chóng và đơn giản trong gói *scipy.stats*. Dưới đây là một ví dụ nhanh về việc sử dụng KDE trên dữ liệu này (Hình 3.34):

```
from scipy.stats import gaussian_kde
# Tao du lieu bang mot mang co kich thuoc [Ndim, Nsamples]
data = np.vstack([x, y])
kde = gaussian_kde(data)
# Danh gia tren mot dai gia tri chinh quy
xgrid = np.linspace(-3.5, 3.5, 40)
ygrid = np.linspace(-6, 6, 40)
Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))
# Truc quan bang imshow
plt.imshow(Z.reshape(Xgrid.shape), origin='lower', aspect='auto',
           extent=[-3.5, 3.5, -6, 6], cmap='Blues',
           interpolation="bicubic")
cb = plt.colorbar()
cb.set_label("density")
```



Hình 3.29: Một đồ thị *Histogram* đơn giản.

3.6 Đồ thị ba chiều

Matplotlib ban đầu được thiết kế với mục đích vẽ đồ thị hai chiều. Từ phiên bản 1.0, một số gói vẽ sơ đồ ba chiều đã được xây dựng trên màn hình hai chiều của *Matplotlib* để trực quan hóa dữ liệu ba chiều. Để sử dụng biểu đồ ba chiều bằng cách nhập bộ công cụ *mplot3d* (Hình 3.35):

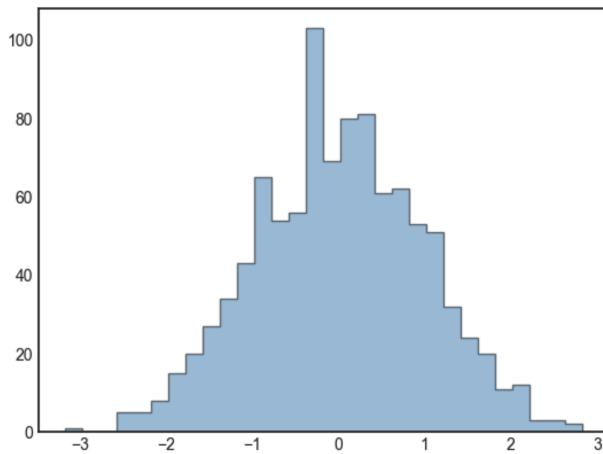
```
from mpl_toolkits import mplot3d
```

Sau khi mô-đun này được nhập, có thể tạo các trục ba chiều bằng cách chuyển từ khóa *projector='3d'* cho bất kỳ quá trình trực quan hóa tạo trục nào:

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

fig = plt.figure()
ax = plt.axes(projection='3d')
```

Chú ý 3.1 Với các trục 3D này có thể vẽ nhiều loại đồ thị ba chiều khác nhau. Đồ thị ba chiều có một chức năng mang lại lợi ích to lớn từ việc xem các số liệu đó là tương tác với đồ thị thay vì dữ liệu tĩnh trong *Notebook*; để sử dụng các số liệu tương tác, có thể sử dụng *%matplotlib notebook* thay vì *%matplotlib inline* khi chạy mã nguồn trực quan hóa các đồ thị 3D này.

Hình 3.30: Một đồ thị *Histogram* tuỳ chỉnh.

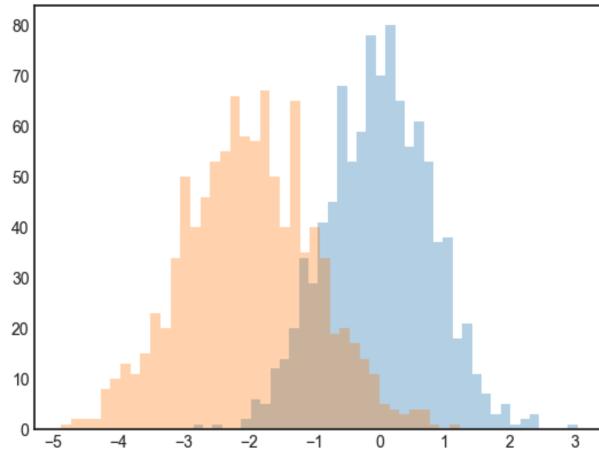
Điểm và đường ba chiều

Đồ thị ba chiều đơn giản nhất là một biểu đồ đường hoặc phân tán các điểm dữ liệu được tạo từ các bộ ba (x, y, z). Tương tự với các biểu đồ hai chiều, trực quan hóa đồ thị 3 chiều bằng cách sử dụng các hàm *ax.plot3D* và *ax.scatter3D*. Ví dụ minh họa dưới đây sẽ vẽ một đường xoắn ốc lượng giác, cùng với một số điểm được vẽ ngẫu nhiên gần đường đó (Hình 3.36):

```
%matplotlib notebook
ax = plt.axes(projection='3d')
# Dữ liệu cho đường 3 chiều
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')
# Dữ liệu cho các điểm rơi vào 3 chiều
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

Trực quan hóa đường viền ba chiều

Tương tự như các biểu đồ đường viền, *mplot3d* chứa các công cụ để tạo các biểu đồ viền ba chiều bằng cách sử dụng cùng các đầu vào. Giống như các đồ thị *ax.contour* hai chiều, *ax.contour3D* yêu cầu tất cả dữ liệu đầu vào phải ở dạng lưới thông thường hai chiều, với dữ liệu Z được đánh giá tại mỗi điểm. Ví dụ dưới đây là một đồ thị đường bao ba chiều của một hàm hình sin ba chiều (Hình 3.37):

Hình 3.31: Một đồ thị *Histogram* kết hợp.

```

def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z=f(X,Y)

ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');

```

Khi sử dụng `%matplotlib notebook` có thể xoay các góc đồ thị 3 chiều theo nhiều các góc khác nhau, trong trường hợp muốn quay một góc hoặc độ cao đồ thị mặc định sẵn có thể sử dụng phương thức `view_init` để đặt độ cao và góc phương vị của đồ thị. Trong ví dụ trên để hiển thị độ cao 60 độ so với mặt phẳng x - y và góc phương vị 35 độ nghĩa là xoay 35 độ ngược chiều kim đồng hồ về trục z (Hình 3.38):

```

ax.view_init(60, 35)
fig

```

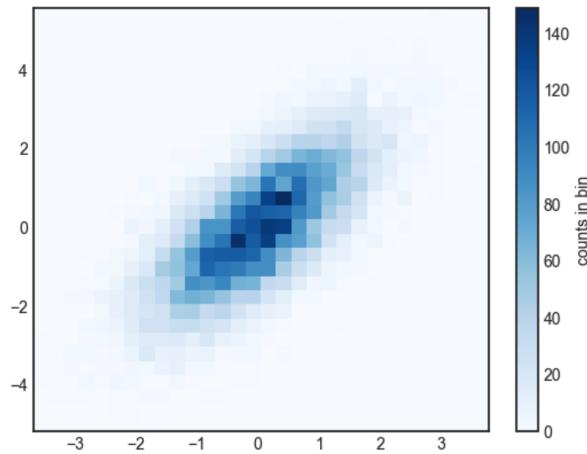
Khung lưới - wireframe và bề mặt

Hai loại biểu đồ ba chiều khác hoạt động trên dữ liệu dạng lưới là khung lưới - *wireframe* và biểu đồ bề mặt - *surface*. Dưới đây là một ví dụ sử dụng *wireframe* (Hình 3.39):

```

ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')

```

Hình 3.32: Đồ thị *histogram* hai chiều.

```
ax.set_title('wireframe');
```

Biểu đồ bề mặt - *surface* giống như biểu đồ khung lưới - *wireframe*, nhưng mỗi mặt của *wireframe* là một đa giác được lấp đầy. Việc thêm một bản đồ màu (*colormap*) vào các đa giác được hỗ trợ bằng các tham số về cấu trúc của bề mặt (Hình 3.40):

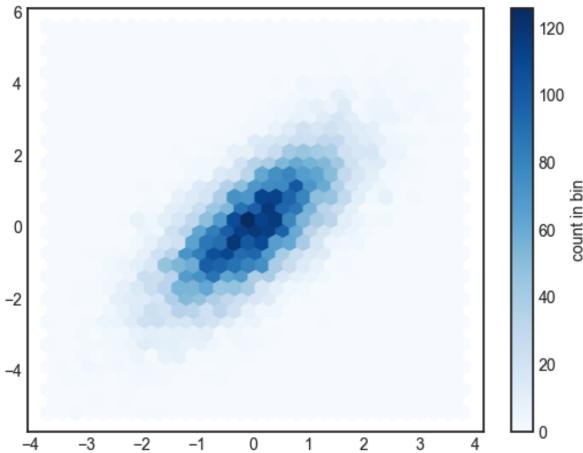
```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_title('surface');
```

Lưu ý mặc dù lưới các giá trị cho biểu đồ bề mặt cần phải là hai chiều, nhưng nó không nhất thiết phải là các giá trị tuyến tính. Dưới đây là một ví dụ về việc tạo một lưới phân cực một phần, mà khi được sử dụng với biểu đồ *surface3D* có thể tạo một lát cắt khi trực quan hóa (Hình 3.41):

```
r = np.linspace(0, 6, 20)
theta = np.linspace(-0.9 * np.pi, 0.8 * np.pi, 40)
r, theta = np.meshgrid(r, theta)

X = r * np.sin(theta)
Y = r * np.cos(theta)
Z=f(X,Y)

ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none');
```



Hình 3.33: Đồ thị *histogram* hai chiều.

Ví dụ: Trực quan hóa dải Möbius

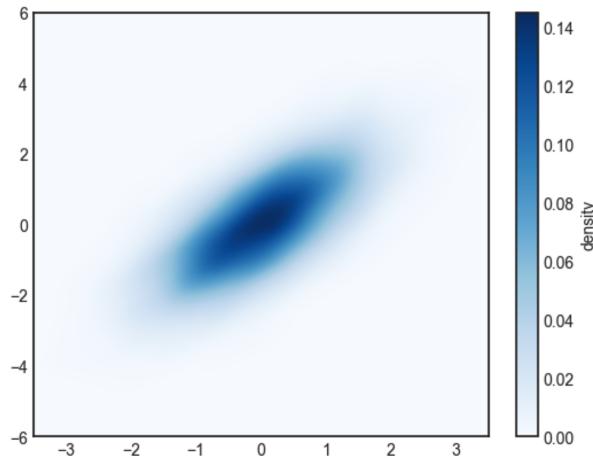
Một dải Möbius tương tự như một mảnh giấy hình chữ nhật dài được dán vào một vòng xoắn nửa (*half-twist*). Về mặt tô pô, nó khá thú vị bởi vì bề ngoài nó chỉ có một mặt! Ở đây nó sẽ trực quan hóa ba chiều bằng *Matplotlib*. Để tạo dải Möbius là tạo tham số hóa cho nó: đó là dải dữ liệu hai chiều, gọi các điểm này là θ , nằm trong khoảng từ 0 đến 2π xung quanh vòng lặp và w nằm trong khoảng từ -1 đến 1 trên chiều rộng của dải:

```
from matplotlib.tri import Triangulation
tri = Triangulation(np.ravel(w), np.ravel(theta))

ax = plt.axes(projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.triangles,
                 cmap='viridis', linewidths=0.2);

ax.set_xlim(-1, 1);
ax.set_ylim(-1, 1);
ax.set_zlim(-1, 1);
```

Một dải Möbius có hai chuyển động quay đang xảy ra: một là vị trí của vòng dây quanh tâm của nó (gọi là θ), hai là sự xoắn của dải quanh trục của nó (gọi là ϕ). Đối với dải Möbius, phải để dải xoắn một nửa trong toàn bộ vòng lặp hay nói cách khác $\Delta\phi = \Delta\theta/2$. Giá trị r xác định khoảng cách của mỗi điểm đến trung tâm và sử dụng giá trị này để tìm tọa độ x, y, z . Sử dụng *Triangulation* để xác định phép đo tam giác và chiều phép đo tam giác vào không gian ba chiều của dải Möbius (Hình 3.42):

Hình 3.34: Đồ thị *histogram* hai chiều.

3.7 Dữ liệu địa lý

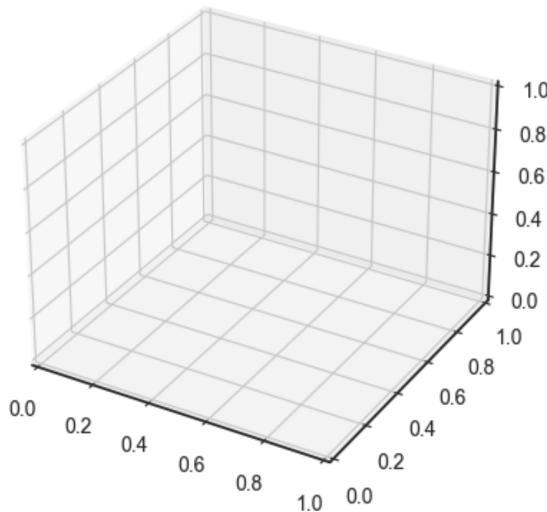
Một loại trực quan hóa phổ biến trong khoa học dữ liệu là dữ liệu địa lý. Công cụ chính của *Matplotlib* cho kiểu trực quan hóa này là bộ công cụ Bản đồ cơ sở - *Basemap*, đây là một trong một số bộ công cụ của *Matplotlib* nằm trong miền *mpl_toolkits*. Bên cạnh đó hiện nay API Google Maps được sử dụng phổ biến và là lựa chọn tốt hơn để trực quan hóa bản đồ chuyên sâu. Tuy nhiên, *Basemap* vẫn là một công cụ hữu ích để người dùng Python có trong bộ công cụ ảo. Phần này, sẽ trình bày một số ví dụ về kiểu trực quan hóa bản đồ có thể thực hiện được với bộ công cụ *Basemap*, do đó, cần cài đặt thư viện basemap trước khi tiến hành thực thi việc trực quan hóa dữ liệu địa lý.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None, lat_0=50, lon_0=-100)
m.bluemarble(scale=0.5);
```

Điều hữu ích là quả địa cầu được hiển thị không chỉ là một hình ảnh đơn thuần, nó là một hàm của *Matplotlib* có đầy đủ chức năng như các tọa độ hình cầu và cho phép dễ dàng vẽ các dữ liệu trên bản đồ. Ví dụ: có thể sử dụng một phép chiếu bản đồ địa điểm khác như phóng to Bắc Mỹ và vẽ vị trí của Seattle (Hình 3.44):

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution=None,
            width=8E6, height=8E6,
            lat_0=45, lon_0=-100,)
```



Hình 3.35: Khởi tạo trục đồ thị 3D.

```
m.e topo(scale=0.5, alpha=0.5)

# Anh xa (long, lat) sang (x, y) de truc quan hoa
x, y = m(-122.3, 47.6)
plt.plot(x, y, 'ok', markersize=5)
plt.text(x, y, 'Seattle', fontsize=12);
```

Phép chiếu bản đồ

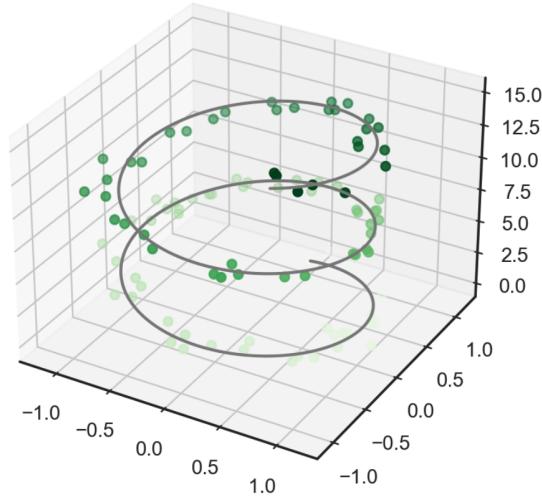
Điều đầu tiên cần quyết định khi làm việc với dữ liệu bản đồ là sử dụng phép chiếu. Tùy thuộc vào mục đích sử dụng của phép chiếu bản đồ, có một số thuộc tính nhất định của bản đồ như hướng, diện tích, khoảng cách, hình dạng hoặc các thuộc tính khác.

Gói *Basemap* thực hiện vài chức phép chiếu như vậy, tất cả được tham chiếu bằng một mã định dạng ngắn. Có thể bắt đầu bằng cách xác định một phương thức thuận tiện để vẽ bản đồ các điểm trên thế giới chỉ với các đường kinh độ và vĩ độ bằng hàm *draw_map* như sau:

```
from itertools import chain
def draw_map(m, scale=0.2):
    # Ve mot anh bong dia hinh
    m.shadedrelief(scale=scale)

    # lats va longs duoc tra ve nhu mot tu dien - dict
    lats = m.drawparallels(np.linspace(-90, 90, 13))
    lons = m.drawmeridians(np.linspace(-180, 180, 13))

    # Cac khoa chua cac mau plt.Line2D
```



Hình 3.36: Đồ thị điểm và đường ba chiều.

```

lat_lines = chain(*([tup[1][0] for tup in lats.items()]))
lon_lines = chain(*([tup[1][0] for tup in lons.items()]))
all_lines = chain(lat_lines, lon_lines)

# Thiết lập các kiểu ứng dụng cho các dòng
for line in all_lines:
    line.set(linestyle='--', alpha=0.3, color='w')

```

Phép chiếu hình trụ - *Cylindrical projections*

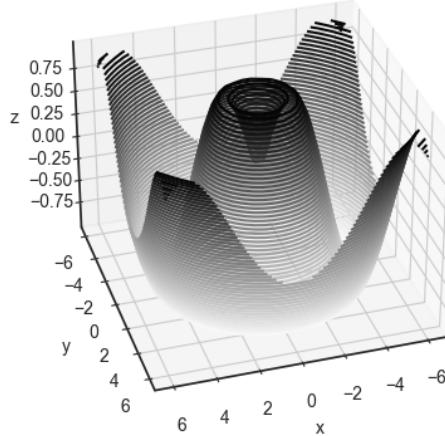
Phép chiếu bản đồ đơn giản nhất là các phép chiếu hình trụ *Cylindrical projections*, trong đó các đường có vĩ độ và kinh độ không đổi lần lượt được ánh xạ tới các đường ngang và dọc. Kiểu lập bản đồ này thể hiện các vùng xích đạo khá tốt, nhưng dẫn đến sự biến dạng cực độ ở gần các cực. Khoảng cách của các đường vĩ tuyến khác nhau giữa các hình chiếu hình trụ khác nhau, dẫn đến các đặc tính bảo tồn khác nhau và sự biến dạng khác nhau gần các cực. Trong Hình 4-104, chúng tôi chỉ ra một ví dụ về phép chiếu hình trụ cách đều, chọn một vĩ độ chia tỷ lệ bảo toàn khoảng cách dọc theo các kinh tuyến. Các phép chiếu hình trụ khác là phép chiếu Mercator (phép chiếu='merc') và phép chiếu có diện tích bằng hình trụ (phép chiếu='cea').

```

fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='cyl', resolution=None,
            llcrnrlat=-90, urcrnrlat=90,
            llcrnrlon=-180, urcrnrlon=180, )
draw_map(m)

fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='moll', resolution=None,

```



Hình 3.37: Đồ thị đường viền ba chiều.

```

lat_0=0, lon_0=0)
draw_map(m)

fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None,
            lat_0=50, lon_0=0)
draw_map(m)

fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution=None,
            lon_0=0, lat_0=50, lat_1=45, lat_2=55,
            width=1.6E7, height=1.2E7)
draw_map(m)

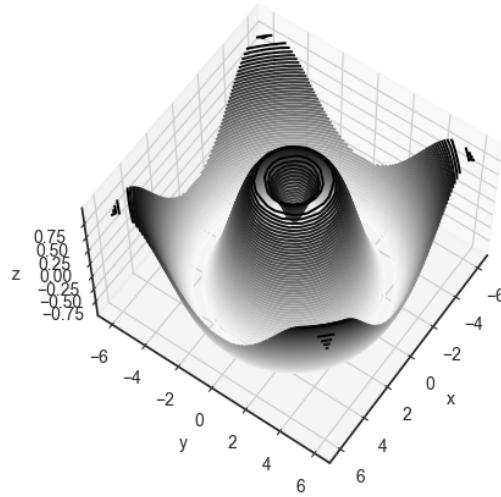
from itertools import chain
def draw_map(m, scale=0.2):
    # Vẽ một ảnh bong dia hình
    m.shadedrelief(scale=scale)

    # lats và longs được trả về như một điển - dict
    lats = m.drawparallels(np.linspace(-90, 90, 13))
    lons = m.drawmeridians(np.linspace(-180, 180, 13))

    # Các khoa chứa các màu plt.Line2D
    lat_lines = chain(*(tup[1][0] for tup in lats.items()))
    lon_lines = chain(*(tup[1][0] for tup in lons.items()))
    all_lines = chain(lat_lines, lon_lines)

    # Thiết lập các kiểu ứng dụng cho các dòng

```



Hình 3.38: Đồ thị đường viền ba chiều tùy chỉnh góc mặc định.

```

for line in all_lines:
    line.set(linestyle='--', alpha=0.3, color='w')

from itertools import chain
def draw_map(m, scale=0.2):
    # Vẽ một ảnh bóng địa hình
    m.shadedrelief(scale=scale)

    # lats và longs được trả về như một từ điển - dict
    lats = m.drawparallels(np.linspace(-90, 90, 13))
    lons = m.drawmeridians(np.linspace(-180, 180, 13))

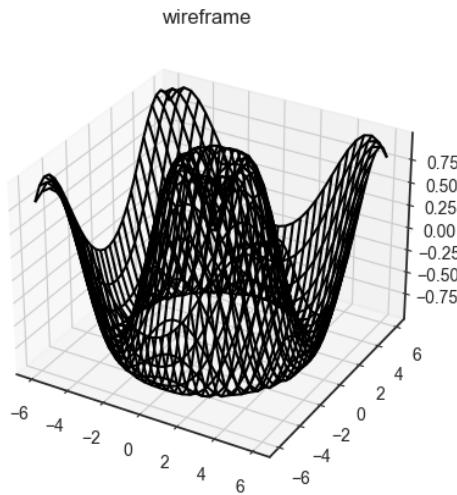
    # Các khóa chứa các mảng plt.Line2D
    lat_lines = chain(*[tup[1][0] for tup in lats.items()])
    lon_lines = chain(*[tup[1][0] for tup in lons.items()])
    all_lines = chain(lat_lines, lon_lines)

    # Thiết lập các kiểu tương ứng cho các dòng
    for line in all_lines:
        line.set(linestyle='--', alpha=0.3, color='w')

from itertools import chain
def draw_map(m, scale=0.2):
    # Vẽ một ảnh bóng địa hình
    m.shadedrelief(scale=scale)

    # lats và longs được trả về như một từ điển - dict
    lats = m.drawparallels(np.linspace(-90, 90, 13))

```



Hình 3.39: Đồ thị khung lưới *wireframe*.

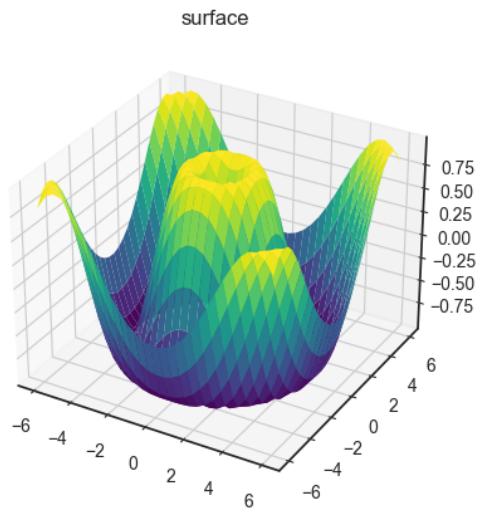
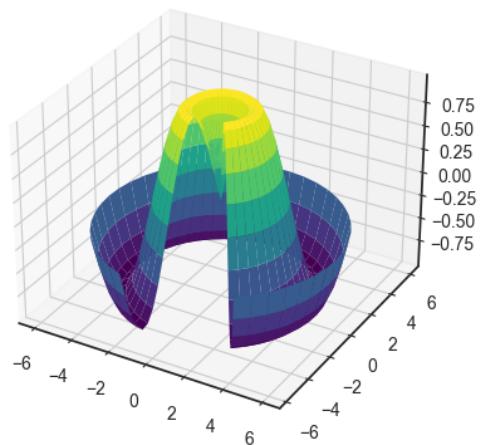
```

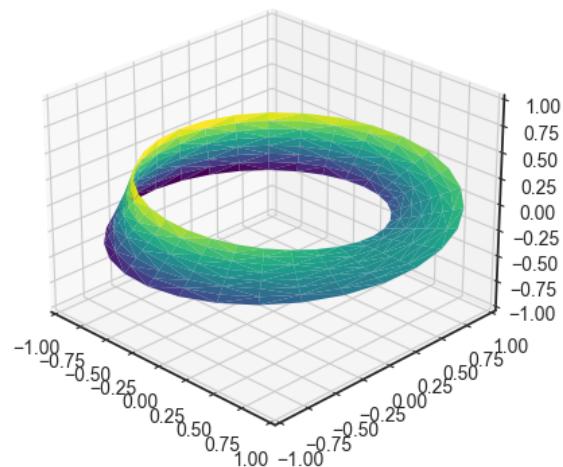
lons = m.drawmeridians(np.linspace(-180, 180, 13))

# Các khoa chưa các màu plt.Line2D
lat_lines = chain(*[tup[1][0] for tup in lats.items()])
lon_lines = chain(*[tup[1][0] for tup in lons.items()])
all_lines = chain(lat_lines, lon_lines)

# Thiết lập các kiểu tượng ứng cho các dòng
for line in all_lines:
    line.set(linestyle='--', alpha=0.3, color='w')

```

Hình 3.40: Đồ thị bề mặt *surface*.Hình 3.41: Đồ thị bề mặt cực - *polar surface*.



Hình 3.42: Trực quan hóa dải Möbius.



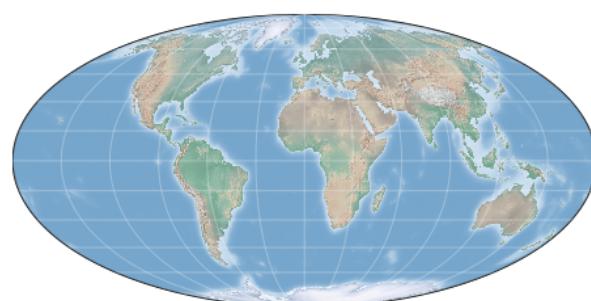
Hình 3.43: Phép chiếu "bluemarble" của Trái đất.



Hình 3.44: Vẽ dữ liệu và nhãn trên bản đồ.



Hình 3.45: Phép chiếu hình trụ.



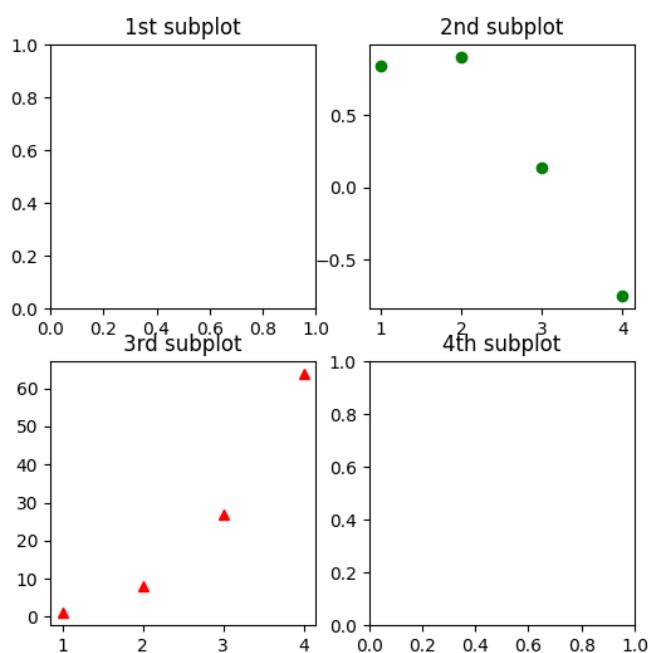
Hình 3.46: Phép chiếu Mollweide.



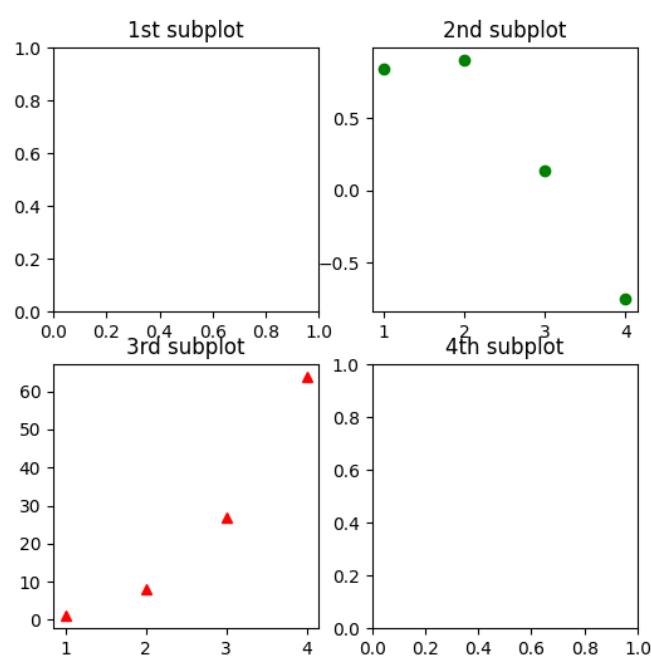
Hình 3.47: Phép chiếu trực giao.



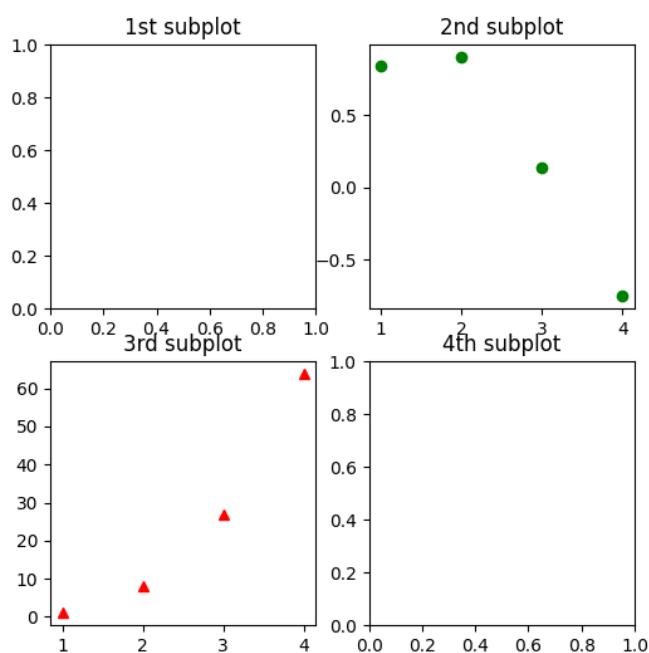
Hình 3.48: Phép chiếu Alberse.



Hình 3.49: Sử dụng *subplot* để vẽ 4 biểu đồ trong một Figure.



Hình 3.50: Sử dụng *subplot* để vẽ 4 biểu đồ trong một Figure.



Hình 3.51: Sử dụng *subplot* để vẽ 4 biểu đồ trong một Figure.

