第 3 章

开源科学集

SciPy 是一个用于数学、科学、工程领域的常用软件包，可以处理最优化、线性代数、积分、插值、拟合、特殊函数、快速傅里叶变换、信号处理、图像处理、常微分方程求解等。SciPy 包含的模块有最优化、线性代数、积分、插值、特殊函数、快速傅里叶变换、信号处理和图像处理、常微分方程求解和其他科学与工程中常用的计算。

NumPy 和 SciPy 的协同工作可以高效地解决很多问题，在天文学、生物学、气象学和气候科学，以及材料科学等多个学科领域中得到了广泛应用。

3.1 SciPy 常量模块

3.1.1 常量

SciPy 常量模块 constants 提供了许多内置的数学常数。其中，圆周率是一个数学常数，为一个圆的周长和其直径的比率，近似值约等于 3.14159，常用符号 π 来表示。

以下代码输出圆周率：

```
from scipy import constants
print(constants.pi)
3.141592653589793
```

我们可以使用 dir()函数来查看 constants 模块包含了哪些常量：

```
from scipy import constants
print(dir(constants))
['Avogadro', 'Boltzmann', 'Btu', 'Btu_IT', 'Btu_th', 'ConstantWarning', 'G',
'Julian_year', 'N_A', 'Planck', 'R', 'Rydberg', 'Stefan_Bol ...
```

3.1.2 单位类型

在 SciPy 中提供了各种类型的单位，下面对几种常用的单位进行介绍。

1. 国际单位制词头

国际单位制词头(SI prefix)表示单位的倍数和分数，目前有 20 个词头，大多数是千的整数次幂(centi 返回 0.01)。

例如，以下代码演示国际单位制词头的值：

```
from scipy import constants
print(constants.yotta)                    #1e+24
print(constants.zetta)                    #1e+21
print(constants.exa)                      #1e+18
print(constants.peta)                     #1000000000000000.0
print(constants.tera)                     #1000000000000.0
print(constants.giga)                     #1000000000.0
print(constants.mega)                     #1000000.0
print(constants.kilo)                     #1000.0
print(constants.hecto)                    #100.0
print(constants.deka)                     #10.0
print(constants.deci)                     #0.1
print(constants.centi)                    #0.01
print(constants.milli)                    #0.001
print(constants.micro)                    #1e-06
print(constants.nano)                     #1e-09
print(constants.pico)                     #1e-12
print(constants.femto)                    #1e-15
print(constants.atto)                     #1e-18
print(constants.zepto)                    #1e-21
```

2. 二进制前缀

二进制前缀用于返回字节单位(kibi 返回 1024)。

例如,以下代码演示各二进制前缀的返回值。

```
from scipy import constants
print(constants.kibi)                     #1024
print(constants.mebi)                     #1048576
print(constants.gibi)                     #1073741824
print(constants.tebi)                     #1099511627776
print(constants.pebi)                     #1125899906842624
print(constants.exbi)                     #1152921504606846976
print(constants.zebi)                     #1180591620717411303424
print(constants.yobi)                     #1208925819614629174706176
```

3. 质量单位

质量单位用于返回多少千克(gram 返回 0.001)。

例如,以下代码演示各质量单位的返回值。

```
from scipy import constants
print(constants.gram)                     #0.001
print(constants.metric_ton)               #1000.0
print(constants.grain)                    #6.479891e-05
print(constants.lb)                       #0.45359236999999997
print(constants.pound)                    #0.45359236999999997
print(constants.oz)                       #0.028349523124999998
print(constants.ounce)                    #0.028349523124999998
print(constants.stone)                    #6.3502931799999995
print(constants.long_ton)                 #1016.0469088
print(constants.short_ton)                #907.1847399999999
print(constants.troy_ounce)               #0.031103476799999998
print(constants.troy_pound)               #0.37324172159999996
print(constants.carat)                    #0.0002
```

```
print(constants.atomic_mass)                #1.66053904e-27
print(constants.m_u)                        #1.66053904e-27
print(constants.u)                          #1.66053904e-27
```

4. 角度单位

角度单位用于返回弧度(degree 返回 0.017453292519943295)。

例如,以下代码返回各角度单位的值。

```
from scipy import constants
print(constants.degree)                     #0.017453292519943295
print(constants.arcmin)                     #0.0002908882086657216
print(constants.arcminute)                  #0.0002908882086657216
print(constants.arcsec)                     #4.84813681109536e-06
print(constants.arcsecond)                  #4.84813681109536e-06
```

5. 时间单位

时间单位用于返回秒数(hour 返回 3600.0)。

例如,以下代码返回各时间单位的值。

```
from scipy import constants
print(constants.minute)                     #60.0
print(constants.hour)                       #3600.0
print(constants.day)                        #86400.0
print(constants.week)                       #604800.0
print(constants.year)                       #31536000.0
print(constants.Julian_year)                #31557600.0
```

6. 面积单位

面积单位用于返回多少平方米,平方米是面积的公制单位,其定义是:在一平面上,边长为一米的正方形之面积(hectare 返回 10000.0)。

例如,下面代码返回各面积单位的值。

```
from scipy import constants
print(constants.hectare)                    #10000.0
print(constants.acre)                       #4046.8564223999992
```

7. 体积单位

体积单位返回多少立方米,立方米为容量计量单位,1 立方米的容量相当于一个长、宽、高都等于 1 米的立方体的体积,与 1 吨水和 1 度水的容积相等,也与 1000000 立方厘米的体积相等(liter 返回 0.001)。

例如,下面代码返回各体积单位的值。

```
from scipy import constants
print(constants.liter)                      #0.001
print(constants.litre)                      #0.001
print(constants.gallon)                     #0.0037854117839999997
print(constants.gallon_US)                  #0.0037854117839999997
print(constants.gallon_imp)                 #0.00454609
print(constants.fluid_ounce)                #2.9573529562499998e-05
print(constants.fluid_ounce_US)             #2.9573529562499998e-05
print(constants.fluid_ounce_imp)            #2.84130625e-05
print(constants.barrel)                     #0.15898729492799998
```

```
print(constants.bbl)                    #0.15898729492799998
```

3.2　SciPy 优化器

SciPy 的 optimize 模块提供了常用的最优化算法函数实现，我们可以直接调用这些函数完成优化问题，比如查找函数的最小值或方程的根等。

1. 寻找方程的根

NumPy 能够找到多项式和线性方程的根，但它无法找到非线性方程的根，如下所示：

$x + \cos(x)$

因此可以使用 SciPy 的 optimize.root 函数，这个函数需要两个参数：fun 表示方程的函数；x0 是根的初始猜测。

该函数返回一个对象，其中包含有关解决方案的信息。实际解决方案在返回对象的属性 x 中。

【例 3-1】 查找 $x + \cos(x)$方程的根。

```
from scipy.optimize import root
from math import cos
def eqn(x):
  return x + cos(x)
myroot = root(eqn, 0)
print(myroot.x)
#查看更多信息
#print(myroot)
[-0.73908513]
```

2. 最小化函数

函数表示一条曲线，曲线有高点和低点。高点称为最大值，低点称为最小值。整条曲线中的最高点称为全局最大值，其余部分称为局部最大值。整条曲线的最低点称为全局最小值，其余的称为局部最小值。

在 SciPy 中，可以使用 scipy.optimize.minimize()函数来最小化函数。minimize()函数接受以下几个参数：fun 是要优化的函数；x0 表示初始猜测值；method 是要使用的方法名称，值可以是'CG' 'BFGS' 'Newton-CG' 'L-BFGS-B' 'TNC' 'COBYLA' 'SLSQP'；callback 表示每次优化迭代后调用的函数。options 是定义其他参数的字典。

```
{
    "disp": boolean - print detailed description
    "gtol": number - the tolerance of the error
}
```

【例 3-2】 使用 BFGS 求函数 x^2+x+2 最小化。

```
from scipy.optimize import minimize
def eqn(x):
  return x**2 + x + 2
mymin = minimize(eqn, 0, method='BFGS')
print(mymin)
```

运行程序，输出如下：

```
      fun: 1.75
```

```
 hess_inv: array([[0.50000001]])
      jac: array([0.])
  message: 'Optimization terminated successfully.'
     nfev: 12
      nit: 2
     njev: 4
   status: 0
  success: True
        x: array([-0.50000001])
```

3.3 SciPy 稀疏矩阵

稀疏矩阵(Sparse Matrix)指的是在数值分析中绝大多数数值为零的矩阵。反之,如果大部分元素都非零,则这个矩阵是稠密的(Dense)。在科学与工程领域中求解线性模型时经常出现大型的稀疏矩阵。

在 Python 中,scipy.sparse()提供了对稀疏矩阵的存储、计算的支持。稀疏矩阵的存储涉及各种各样的存储方式和数据结构,下面对几种结构进行介绍。

3.3.1 coo_matrix 存储方式

coo_matrix 是最简单的稀疏矩阵存储方式,采用三元组(row, col, data)(或称 ijv format)的形式来存储矩阵中非零元素的信息。在实际使用中,一般 coo_matrix 用来创建矩阵,因为 coo_matrix 无法对矩阵的元素进行增删改操作;创建成功之后可以转换为其他格式的稀疏矩阵(如 csr_matrix、csc_matrix)进行转置、矩阵乘法等操作。

coo_matrix 可以通过 4 种方式实例化,除了可以通过 coo_matrix(D)(D 代表密集矩阵),coo_matrix(S)(S 代表其他类型稀疏矩阵)或者 coo_matrix((M, N), [dtype])构建一个 shape 为 M×N 的空矩阵,默认数据类型是 d,还可以通过(row, col, data)三元组初始化:

```
import numpy as np
from scipy.sparse import coo_matrix
_row  = np.array([0, 3, 1, 0])
_col  = np.array([0, 3, 1, 2])
_data = np.array([4, 5, 7, 9])
coo = coo_matrix((_data, (_row, _col)), shape=(4, 4), dtype=np.int)
coo.todense()                    #通过 todense 方法转换为密集矩阵(numpy.matrix)
coo.toarray()                    #通过 toarray 方法转换为密集矩阵(numpy.ndarray)
array([[4, 0, 9, 0],
       [0, 7, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 5]])
```

上面通过 triplet format 的形式构建了一个 coo_matrix 对象,我们可以看到坐标点(0,0)对应值为 4,坐标点(1,1)对应值为 7 等,这就是 coo_matrix。

下面给出 coo_matrix 矩阵文件读写代码,mmread()用于读取稀疏矩阵,mmwrite()用于写入稀疏矩阵,mminfo()用于查看稀疏矩阵文件元信息(这三个函数的操作不仅仅限于 coo_matrix)。

```
from scipy.io import mmread, mmwrite, mminfo
HERE = dirname(__file__)
```

```
coo_mtx_path = join(HERE, 'data/matrix.mtx')
coo_mtx = mmread(coo_mtx_path)
print(mminfo(coo_mtx_path))
#(13885, 1, 949, 'coordinate', 'integer', 'general')
#(rows, cols, entries, format, field, symmetry)
mmwrite(join(HERE, 'data/saved_mtx.mtx'), coo_mtx)
```

至此，可以总结出 coo_matrix 存储方式的优点主要表现为：

(1) 有利于稀疏格式之间的快速转换(tobsr()、tocsr()、to_csc()、to_dia()、to_dok()、to_lil())。

(2) 允许有重复项(格式转换的时候自动相加)。

(3) 能与 CSR / CSC 格式快速转换。

coo_matrix 存储方式的缺点主要表现为：不能直接进行算术运算。

3.3.2　csr_matrix 存储方式

csr_matrix(Compressed Sparse Row Matrix)为按行压缩的稀疏矩阵存储方式，由三个一维数组 indptr、indices、data 组成。这种格式要求矩阵元按行顺序存储，每一行中的元素可以乱序存储。对于每一行就只需要用一个指针表示该行元素的起始位置即可。indptr 存储每一行数据元素的起始位置，indices 是存储每行中数据的列号，与 data 中的元素一一对应。

csr_matrix 可用于各种算术运算：它支持加法、减法、乘法、除法和矩阵幂等操作。其有 5 种实例化方法，其中前 4 种初始化方法类似 coo_matrix，即通过密集矩阵构建、通过其他类型稀疏矩阵转换、构建一定 shape 的空矩阵、通过(row, col, data)构建矩阵。其第 5 种初始化方式直接体现 csr_matrix 的存储特征：csr_matrix((data, indices, indptr), [shape＝(M, N)])，即指矩阵中第 i 行非零元素的列号为 indices[indptr[i]: indptr[i＋1]]，相应的值为 data[indptr[i]: indptr[i＋1]]。

【例 3-3】 csr_matrix 存储实例演示。

```
import numpy as np
indptr = np.array([0, 2, 3, 6])
indices = np.array([0, 2, 2, 0, 1, 2])
data = np.array([1, 2, 3, 4, 5, 6])
csr = csr_matrix((data, indices, indptr), shape=(3, 3)).toarray()
csr
array([[1, 0, 2],
       [0, 0, 3],
       [4, 5, 6]])
```

至此，可以总结出 csr_matrix 存储方式的优点主要表现为：

(1) 高效的算术运算；

(2) 高效的行切片；

(3) 快速的矩阵运算。

csr_matrix 存储方式的缺点主要表现为：

(1) 列切片操作比较慢；

(2) 稀疏结构的转换比较慢。

3.3.3　csc_matrix 存储方式

csc_matrix 和 csr_matrix 正好相反，即按列压缩的稀疏矩阵存储方式，同样由三个一维数

组 indptr、indices、data 组成，其实例化方式、属性、方法、优缺点和 csr_matrix 基本一致，这里不再赘述，它们之间唯一的区别就是按行或按列压缩进行存储。而这一区别决定了 csr_matrix 擅长行操作；csc_matrix 擅长列操作，进行运算时需要进行合理存储结构的选择。

【例 3-4】 csc_matrix 存储方式实例演示。

```
import numpy as np
from scipy import sparse
from scipy.sparse import csc_matrix
sparse.csc_matrix((3, 4), dtype=np.int8).toarray()
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
row = np.array([0, 2, 2, 0, 1, 2])
col = np.array([0, 0, 1, 2, 2, 2])
data = np.array([1, 2, 3, 4, 5, 6])
sparse.csc_matrix((data, (row, col)), shape=(3, 3)).toarray()
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]], dtype=int32)
indptr = np.array([0, 2, 3, 6])
indices = np.array([0, 2, 2, 0, 1, 2])
data = np.array([1, 2, 3, 4, 5, 6])
sparse.csc_matrix((data, indices, indptr), shape=(3, 3)).toarray()
array([[1, 0, 4],
       [0, 0, 5],
       [2, 3, 6]])
```

在本例中，第 0 列，有非 0 的数据行是 indices[indptr[0]：indptr[1]] = indices[0：2] = [0,2]，数据是 data[indptr[0]：indptr[1]] = data[0：2] = [1,2]，所以在第 0 列第 0 行是 1，第 2 行是 2。第 1 行，有非 0 的数据行是 indices[indptr[1]：indptr[2]] = indices[2：3] = [2]，数据是 data[indptr[1]：indptr[2] = data[2：3] = [3]，所以在第 1 列第 2 行是 3。第 2 行，有非 0 的数据行是 indices[indptr[2]：indptr[3]] = indices[3：6] = [0,1,2]，数据是 data[indptr[2]：indptr[3]] = data[3：6] = [4,5,6]，所以在第 2 列第 0 行是 4，第 1 行是 5，第 2 行是 6。

3.3.4 lil_matrix 存储方式

lil_matrix(List of Lists Format)，又称为“基于行的链表稀疏矩阵”。它使用两个嵌套列表存储稀疏矩阵：data 保存每行中的非零元素的值，rows 保存每行非零元素所在的列号(列号是顺序排序的)。这种格式很适合逐个添加元素，并且能快速获取行相关的数据。其初始化方式同 coo_matrix 初始化的前三种方式：通过密集矩阵构建、通过其他矩阵转换以及构建一个一定 shape 的空矩阵。

lil_matrix 可用于算术运算：支持加法、减法、乘法、除法和矩阵幂。其属性前 5 个与 coo_matrix 相同，另外还有 rows 属性，是一个嵌套 List，表示矩阵每行中非零元素的列号。lil_matrix 本身的设计是用来方便快捷地构建稀疏矩阵实例的，而算术运算、矩阵运算则转换为 CSC、CSR 格式再进行，构建大型的稀疏矩阵还是推荐使用 COO 格式。

lil_matrix 存储方式优点主要表现为：

(1) 支持灵活的切片操作，行切片操作效率高，列切片效率低。

(2) 稀疏矩阵格式之间的转换很高效。

lil_matrix 存储方式缺点主要表现为：

(1) 加法操作效率低。

(2) 列切片效率低。

(3) 矩阵乘法效率低。

【例 3-5】 lil_matrix 存储方式实例演示。

```
import numpy as np
from scipy import sparse
from scipy.sparse import lil_matrix
lil = sparse.lil_matrix((6, 5), dtype=int)                          #创建矩阵
#set individual point                                               #设置数值
lil[(0, -1)] = -1
lil[3, (0, 4)] = [-2] * 2                                           #设置两点
lil.setdiag(8, k=0)                                                 #设置主对角线
lil[:, 2] = np.arange(lil.shape[0]).reshape(-1, 1) + 1              #设置整列
lil.toarray()                                                       #转为 array
array([[ 8,  0,  1,  0, -1],
       [ 0,  8,  2,  0,  0],
       [ 0,  0,  3,  0,  0],
       [-2,  0,  4,  8, -2],
       [ 0,  0,  5,  0,  8],
       [ 0,  0,  6,  0,  0]])
#查看数据
lil.data
array([list([8, 1, -1]), list([8, 2]), list([3]), list([-2, 4, 8, -2]),
      list([5, 8]), list([6])], dtype=object)
lil.rows
array([list([0, 2, 4]), list([1, 2]), list([2]), list([0, 2, 3, 4]),
      list([2, 4]), list([2])], dtype=object)
```

3.3.5　dok_matrix 存储方式

dok_matrix(Dictionary of Keys Based Sparse Matrix)是一种类似于 coo matrix 但又基于字典的稀疏矩阵存储方式，key 由非零元素的坐标值 tuple(row, column)组成，value 则代表数据值。dok_matrix 非常适合于增量构建稀疏矩阵，并且一旦构建，就可以快速地转换为 coo_matrix。其属性和 coo_matrix 前 4 项相同；其初始化方式同 coo_matrix 初始化的前 3 种：通过密集矩阵构建、通过其他矩阵转换以及构建一个一定 shape 的空矩阵。对于 dok_matrix，可用于算术运算：它支持加法、减法、乘法、除法和矩阵幂；允许对单个元素进行快速访问(O(1))；不允许重复。

【例 3-6】 dok_matrix 存储方式实例演示。

```
from scipy.sparse import dok_matrix
a = dok_matrix((3, 10))
a[1, 2] = 2
a[2, 2] = 2
a[2, 3] = 1
a[2, 4] = 3
print(a)
print('----------------------')
```

```
print(a[2])
print('----------------------')
print(a[2].nonzero()[1])
print(a[2].nonzero())
print(a[2].values())
```

运行程序,输出如下:

```
  (1, 2)    2.0
  (2, 2)    2.0
  (2, 3)    1.0
  (2, 4)    3.0
----------------------
  (0, 2)    2.0
  (0, 3)    1.0
  (0, 4)    3.0
----------------------
[2 3 4]
(array([0, 0, 0], dtype=int32), array([2, 3, 4], dtype=int32))
dict_values([2.0, 1.0, 3.0])
```

3.3.6 dia_matrix 存储方式

dia_matrix(Sparse Matrix With DIAgonal Storage)是一种对角线的存储方式。将稀疏矩阵使用 offsets 和 data 两个矩阵来表示。offsets 表示 data 中每一行数据在原始稀疏矩阵中的对角线位置 k($k>0$,对角线往右上角移动;$k<0$,对角线往左下方移动;$k=0$,主对角线)。该格式的稀疏矩阵可用于算术运算:它们支持加法、减法、乘法、除法和矩阵幂。

dia_matrix 的 5 个属性与 coo matrix 相同,另外还有属性 offsets;dia_matrix 有 4 种初始化方式,其中前 3 种初始化方式与 coo_matrix 前 3 种初始化方式相同,即通过密集矩阵构建、通过其他矩阵转换以及构建一个一定 shape 的空矩阵。第 4 种初始化方式如下:

```
dia_matrix((data, offsets), shape=(M, N))
```

其中,data[k,:]存储着稀疏矩阵;offsets[k]对角线上的值。

【例 3-7】 dia_matrix 存储方式的实例演示。

```
from scipy.sparse import dia_matrix
import numpy as np
if __name__ == '__main__':
    data = np.array([[1,2,3,4],
                     [4,2,3,8],
                     [7,2,4,5]])
    offsets = np.array([0,-1,2])
    a = dia_matrix((data,offsets),shape=(4,4)).toarray()
    print(a)
```

运行程序,输出如下:

```
[[1 0 4 0]
 [4 2 0 5]
 [0 2 3 0]
 [0 0 3 4]]
```

3.3.7　bsr_matrix 存储方式

bsr_matrix(Block Sparse Row Matrix)这种压缩方式类似 CSR 格式,它是使用分块的思想对稀疏矩阵进行按行压缩的。所以,BSR 适用于具有 dense 子矩阵的稀疏矩阵。该种矩阵有 5 种初始化方式,如下所示:

- bsr_matrix(D, [blocksize=(R,C)]):D 是一个 M×N 的二维 dense 矩阵;blocksize 需要满足条件:M % R = 0 和 N % C = 0,如果不给定该参数,内部将会应用启发式的算法自动决定一个合适的 blocksize。
- bsr_matrix(S, [blocksize=(R,C)]):S 是指其他类型的稀疏矩阵。
- bsr_matrix((M, N), [blocksize=(R,C), dtype]):构建一个 shape 为 M×N 的空矩阵。
- bsr_matrix((data, ij), [blocksize=(R,C), shape=(M, N)]):data 和 ij 满足条件"a[ij[0,k],ij[1,k]]=data[k]"。
- bsr_matrix((data, indices, indptr), [shape=(M, N)]):data.shape 一般是 k×R×C,其中 R、C 分别代表 block 的行长和列长,代表有几个小 block 矩阵;第 i 行的块列索引存储在 indices[indptr[i]: indptr[i+1]]中,其值是 data[indptr[i]: indptr[i+1]]。

bsr_matrix 可用于算术运算:支持加法、减法、乘法、除法和矩阵幂。

【例 3-8】　bsr_matrix 存储方式演示实例。

```
from scipy.sparse import bsr_matrix
import numpy
indptr = np.array([0, 2, 3, 6])
indices = np.array([0, 2, 2, 0, 1, 2])
data = np.array([1, 2, 3, 4, 5, 6]).repeat(4).reshape(6, 2, 2)
bsr_matrix((data,indices,indptr), shape=(6, 6)).toarray()
```

运行程序,输出如下:

```
array([[1, 1, 0, 0, 2, 2],
       [1, 1, 0, 0, 2, 2],
       [0, 0, 0, 0, 3, 3],
       [0, 0, 0, 0, 3, 3],
       [4, 4, 5, 5, 6, 6],
       [4, 4, 5, 5, 6, 6]])
```

3.4　SciPy 图结构

图结构是算法学中最强大的框架之一。图是各种关系的节点和边的集合,节点是与对象对应的顶点,边是对象之间的连接。SciPy 提供了 scipy.sparse.csgraph 模块来处理图结构。

3.4.1　邻接矩阵

邻接矩阵(Adjacency Matrix)是表示顶点之间相邻关系的矩阵。邻接矩阵逻辑结构分为两部分:V 和 E 集合,其中,V 是顶点,E 是边,边有时会有权重,表示节点之间的连接强度,如图 3-1 所示。

用一个一维数组存放图中所有顶点数据,用一个二维数组存放顶点间关系(边或弧)的数

据，这个二维数组称为邻接矩阵，如图 3-2 所示。

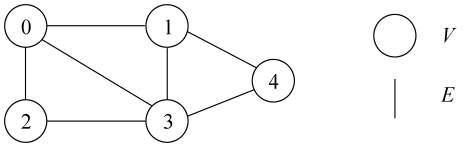

图 3-1　邻接矩阵逻辑结构图

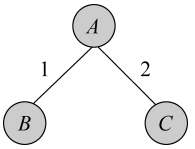

图 3-2　二维数组顶点关系图

图 3-2 中，顶点有 A、B、C，边权重有 1 和 2。A 与 B 是连接的，权重为 1。A 与 C 是连接的，权重为 2。C 与 B 是没有连接的。

这个邻接矩阵可以表示为以下二维数组：

```
   A B C
A:[0 1 2]
B:[1 0 0]
C:[2 0 0]
```

邻接矩阵又分为有向图邻接矩阵和无向图邻接矩阵。无向图是双向关系，边没有方向，如图 3-3 所示。

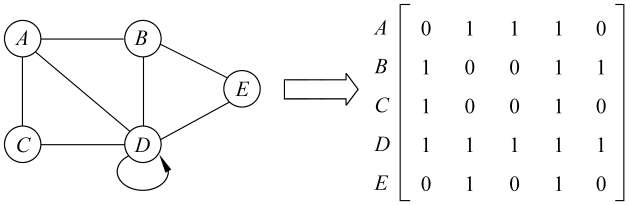

图 3-3　无向图

有向图的边带有方向，是单向关系，如图 3-4 所示。

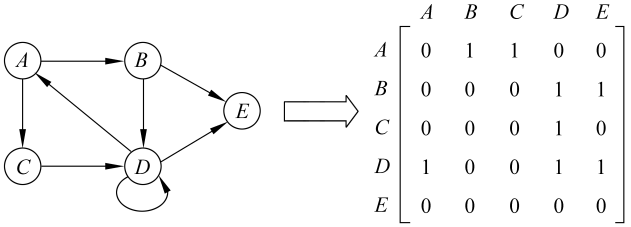

图 3-4　有向图

提示：图 3-3 及图 3-4 中的 D 节点是自环，自环是指一条边的两端为同一个节点。

3.4.2　连接组件

在 SciPy 中，提供了 connected_components()方法用于查看所有连接组件使用。

【例 3-9】　查看所有连接组件使用。

```
import numpy as np
from scipy.sparse.csgraph import connected_components
from scipy.sparse import csr_matrix
arr = np.array([
  [3, 5, 2],
  [1, 0, 0],
```

```
  [4, 1, 0]
])
newarr = csr_matrix(arr)
print(connected_components(newarr))
```

运行程序，输出如下：

```
(1, array([0, 0, 0]))
```

3.4.3　Dijkstra 最短路径

Dijkstra(迪杰斯特拉)最短路径算法，可用于求解图中某源点到其余各顶点的最短路径。

【例 3-10】 从某源点到其余各顶点的最短路径。

假设 G＝{V,{E}}是含有 n 个顶点的有向图，以该图中顶点 v 为源点，使用 Dijkstra 算法求顶点 v 到图中其余各顶点的最短路径的基本方法如下：

(1) 使用集合 S 记录已求得最短路径的终点，初始时 S＝{v}。

(2) 选择一条长度最小的最短路径，该路径的终点 w 属于 V-S，将 w 并入 S，并将该最短路径的长度记为 Dw。

(3) 对于 V-S 中任一顶点 s，将源点到顶点 s 的最短路径长度记为 Ds，并将顶点 w 到顶点 s 的弧的权值记为 Dws，如果 Dw＋Dws＜Ds，则将源点到顶点 s 的最短路径长度修改为 Dw＋Ds＝ws。

(4) 重复执行步骤(2)和(3)，并且 S＝V。

为了实现算法，使用邻接矩阵 Arcs 存储有向网，当 i＝j 时，Arcs[i][j]＝0；当 i！＝j 时，如果下标为 i 的顶点到下标为 j 的顶点有弧且弧的权值为 w，则 Arcs[i][j]＝w，否则 Arcs[i][j]＝float('inf')即无穷大。

(5) 使用 Dist 存储源点到每一个终点的最短路径长度。

(6) 使用列表 Path 存储每一条最短路径中倒数第二个顶点的下标。

(7) 使用 flag 记录每一个顶点是否已经求得最短路径，在方法中即是判断顶点是属于 V 集合，还是属于 V-S 集合。

实现的代码为：

```
#构造有向图 Graph
class Graph:
  def __init__(self,graph,labels):          #labels 为标点名称
    self.Arcs=graph
    self.VertexNum=graph.shape[0]
    self.labels=labels
def Dijkstra(self,Vertex,EndNode):          #Vertex 为源点,EndNode 为终点
  Dist=[[] for i in range(self.VertexNum)] #存储源点到每一个终点的最短路径的长度
  Path=[[] for i in range(self.VertexNum)] #存储每一条最短路径中倒数第二个顶点的下标
  flag=[[] for i in range(self.VertexNum)] #记录每一个顶点是否都求得最短路径
  index=0
  #初始化
  while index<self.VertexNum:
    Dist[index]=self.Arcs[Vertex][index]
    flag[index]=0
    if self.Arcs[Vertex][index]<float('inf'):  #正无穷
      Path[index]=Vertex
```

```
      else:
        Path[index]=-1                          #表示从顶点 Vertex 到 index 无路径
      index+=1
    flag[Vertex]=1
    Path[Vertex]=0
    Dist[Vertex]=0
    index=1
    while index<self.VertexNum:
      MinDist=float('inf')
      j=0
      while j<self.VertexNum:
        if flag[j]==0 and Dist[j]<MinDist:
          tVertex=j#tVertex 为目前从 V-S 集合中找出的距离源点 Vertex 最短路径的顶点
          MinDist=Dist[j]
        j+=1
      flag[tVertex]=1
      EndVertex=0
      MinDist=float('inf')      #表示无穷大,若两点间的距离小于 MinDist,说明两点间有路径
      #更新 Dist 列表
      while EndVertex<self.VertexNum:
        if flag[EndVertex]==0:
          if self.Arcs[tVertex][EndVertex]<MinDist and Dist[
            tVertex]+self.Arcs[tVertex][EndVertex]<Dist[EndVertex]:
            Dist[EndVertex]=Dist[tVertex]+self.Arcs[tVertex][EndVertex]
            Path[EndVertex]=tVertex
        EndVertex+=1
      index+=1
    vertex_endnode_path=[]                      #存储从源点到终点的最短路径
    return Dist[EndNode],start_end_Path(Path,Vertex,EndNode,vertex_endnode_path)
#定义 Path 递归求路径
def start_end_Path(Path,start,endnode,path):
    if start==endnode:
      path.append(start)
    else:
      path.append(endnode)
      start_end_Path(Path,start,Path[endnode],path)
    return path

if __name__=='__main__':
    #float('inf')表示无穷
    graph=np.array([[0,6,5,float('inf'),float('inf'),float('inf')],
            [float('inf'),0,2,8,float('inf'),float('inf')],
            [float('inf'),float('inf'),0,float('inf'),3,float('inf')],
            [float('inf'),float('inf'),7,0,float('inf'),9],
            [float('inf'),float('inf'),float('inf'),float('inf'),0,9],
            [float('inf'),float('inf'),float('inf'),float('inf'),0]])
    G=Graph(graph,labels=['a','b','c','d','e','f'])
    start=input('请输入源点')
    endnode=input('请输入终点')
    dist,path=Dijkstra(G,G.labels.index(start),G.labels.index(endnode))
    Path=[]
    for i in range(len(path)):
      Path.append(G.labels[path[len(path)-1-i]])
```

```
  print('从顶点{}到顶点{}的最短路径为:\n{}\n 最短路径长度为:{}'.format(start,
endnode,Path,dist))
```

运行程序,输出如下:

```
请输入源点 b
请输入终点 f
从顶点 b 到顶点 f 的最短路径为:
['b', 'c', 'e', 'f']
最短路径长度为:14
```

3.4.4　Floyd Warshall 算法

Floyd Warshall(弗洛伊德)算法又称为插点法,是一种利用动态规划的思想寻找给定的加权图中多源点之间最短路径的算法,与 Dijkstra 算法类似。该算法名称以创始人之一、1978年图灵奖获得者、斯坦福大学计算机科学系教授罗伯特·弗洛伊德命名。

Floyd Warshall 算法是解决任意两点间的最短路径的一种算法,可以正确处理有向图或负权的最短路径问题,同时也被用于计算有向图的传递闭包。Floyd Warshall 算法的时间复杂度为 $O(N^3)$,空间复杂度为 $O(N^2)$。

1. 算法原理

Floyd Warshall 算法是一个经典的动态规划算法。通俗来说,首先我们的目标是寻找从点 i 到点 j 的最短路径。从动态规划的角度看问题,我们需要为这个目标重新做一个解释。

从任意节点 i 到任意节点 j 的最短路径有两种可能,一种是直接从 i 到 j,另一种是从 i 经过若干个节点 k 到 j。所以,假设 Dis(i,j)为节点 u 到节点 v 的最短路径的距离,对于每一个节点 k,检查 Dis(i,k) + Dis(k,j) < Dis(i,j)是否成立,如果成立,证明从 i 到 k 再到 j 的路径比 i 直接到 j 的路径短,便设置 Dis(i,j) = Dis(i,k) + Dis(k,j),这样一来,当遍历完所有节点 k,Dis(i,j)中记录的便是 i 到 j 的最短路径的距离。

2. 算法描述

Floyd Warshall 算法可通过以下两点来描述。

(1) 从任意一条单边路径开始。所有两点之间的距离是边的权,如果两点之间没有边相连,则权为无穷大。

(2) 对于每一对顶点 u 和 v,看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比已知的路径更短。如果是更新它。

【例 3-11】 利用 Floyd Warshall 算法找到所有最短路径长度。

```
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib import font_manager
#使用 Floyd Warshall 算法找到所有最短路径长度
G = nx.DiGraph()
G.add_weighted_edges_from([('0', '3', 3), ('0', '1', -5),('0', '2', 2), ('1', '2', 4),
('2', '3', 1)])
#边和节点信息
edge_labels = nx.get_edge_attributes(G,'weight')
labels={'0':'0','1':'1','2':'2','3':'3'}
#生成节点位置
pos=nx.spring_layout(G)
#把节点画出来
```

```
nx.draw_networkx_nodes(G,pos,node_color='g',node_size=500,alpha=0.8)
#把边画出来
nx.draw_networkx_edges(G,pos,width=1.0,alpha=0.5,edge_color='b')
#把节点的标签画出来
nx.draw_networkx_labels(G,pos,labels,font_size=16)
#把边权重画出来
nx.draw_networkx_edge_labels(G, pos, edge_labels)
#显示 graph
plt.title('有权图',fontproperties=myfont)
plt.axis('on')
plt.xticks([])
plt.yticks([])
plt.show()
#计算最短路径长度
lenght=nx.floyd_warshall(G, weight='weight')
#计算最短路径上的前驱与路径长度
predecessor,distance1=nx.floyd_warshall_predecessor_and_distance(G, weight=
'weight')
#计算两两节点之间的最短距离,并以 NumPy 矩阵形式返回
distance2=nx.floyd_warshall_numpy(G, weight='weight')
print(list(lenght))
print(predecessor)
print(list(distance1))
print(distance2)
```

运行程序,输出如下,效果如图 3-5 所示。

```
['2', '3', '0', '1']
{'2': {'3': '2'}, '0': {'2': '1', '3': '2', '1': '0'}, '1': {'2': '1', '3': '2'}}
['2', '3', '0', '1']
[[ 0.  1. inf inf]
 [inf  0. inf inf]
 [-1.  0.  0. -5.]
 [ 4.  5. inf  0.]]
```

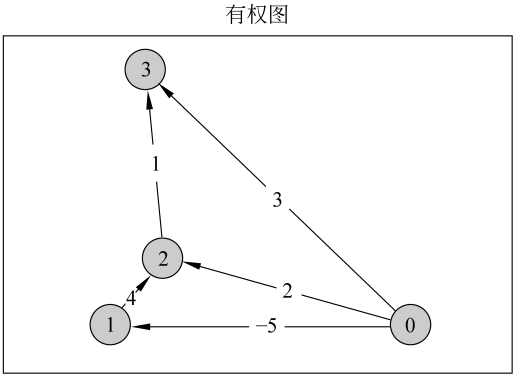

图 3-5　有权图

3.4.5　Bellman-Ford 算法

Bellman-Ford(贝尔曼-福特)算法是一种处理存在负权边的单元最短路径问题的算法。解决了 Dijkstra 无法计算的存在负权边的问题。虽然其算法效率不高,但是也有其特别的用

处。其实现方式是通过 m 次迭代求出从源点到终点不超过 m 条边构成的最短路的路径。一般情况下要求途中不存在负环。但是在边数有限制的情况下允许存在负环。因此 Bellman-Ford 算法是可以用来判断负环的。

1. 算法原理

Bellman-Ford 算法能在更普遍的情况下(存在负权边)解决单源点最短路径问题。对于给定的带权(有向或无向)图 G=(V,E),其源点为 s,加权函数 w 是边集 E 的映射。对图 G 运行 Bellman-Ford 算法的结果是一个布尔值,表明图中是否存在着一个从源点 s 可达的负权回路。如果不存在这样的回路,算法将给出从源点 s 到图 G 的任意顶点 v 的最短路径 d[v]。

Bellman-Ford 算法主要适用于:

- 单源最短路径(从源点 s 到其他所有顶点 v);
- 有向图与无向图(无向图可以看作(u,v),(v,u)同属于边集 E 的有向图);
- 边权可正可负(如有负权回路输出错误提示);
- 差分约束系统。

2. 算法流程

整体来说,Bellman-Ford 算法的流程主要有以下几步:

(1) 初始化:将除源点外的所有顶点的最短距离估计值 d[v]←+∞,d[s]←0。

(2) 迭代求解:反复对边集 E 中的每条边进行松弛操作,使得顶点集 V 中的每个顶点 v 的最短距离估计值逐步逼近其最短距离。(运行|v|−1 次)

(3) 检验负权回路:判断边集 E 中的每一条边的两个端点是否收敛。如果存在未收敛的顶点,则算法返回 False,表明问题无解;否则算法返回 True,并且从源点可达的顶点 v 的最短距离保存在 d[v]中。

【例 3-12】 利用 Bellman-Ford 算法实现最短(长)路径。

```
from collections import deque
import math
inf = math.inf
print(inf>0)
class BellmanFordSP(object):
    def __init__(self,Graph,s):
        '''
        :param Graph: 有向图的邻接矩阵
        :param s:  起点 Start
        '''
        self.Graph = Graph
        self.edgeTo = []    #用来存储路径结束的横切边(即最短路径的最后一条边的两个顶点)
        self.distTo = []                          #用来存储到每个顶点的最短路径
        self.s = s                                #起点 Start
    #打印顶点 s 到某一点的最短路径
    def PrintPath(self,end):
        path = [end]
        while self.edgeTo[end] != None:
            path.insert(0,self.edgeTo[end])  #倒排序
            end = self.edgeTo[end]
        return path
    #路径中含有正(负)权重环判定,即判断当前顶点是否存在于一个环中。
    def cycle_assert(self, vote):
        '''
```

```
    利用顶点出度、入度,当前顶点满足环的"必要条件"是至少 1 出度、1 入度。
    再查看起点能否回到起点的路径判断。两项满足则为环。
    '''
    path = [vote]
    while self.edgeTo[vote] != None:
        path.insert(0,self.edgeTo[vote])
        vote = self.edgeTo[vote]
        if path[0] == path[-1]:
            break
    print(path)
    if path[0] == path[-1]:
        return True
    else:
        return False
#主程序
def bellmanford(self):
    d = deque()                                #导入优先队列(队列性质:先入先出)
    for i in range(len(self.Graph[0])):   #初始化横切边与最短路径——"树"
        self.distTo.append(inf)
        self.edgeTo.append(None)
    self.distTo[self.s] = 0                    #将顶点 s 加入 distTo 中
    count  = 0                                 #计数标识
    d.append(self.Graph[self.s].index(min(self.Graph[self.s])))
    #将直接距离顶点 s 最近的点加入队列
    for i in self.Graph[self.s]:    #将除直接距离顶点 s 最近的点外的其他顶点加入队列
        if i != inf and count not in d:
            d.append(count)
        count += 1
    for j in d:                            #处理刚加入队列的顶点
        self.edgeTo[j] = self.s
        self.distTo[j] = self.Graph[self.s][j]
    while d:
        count = 0
        vote = d.popleft()             #将弹出该点作为顶点 s,重复操作,直到队列为空
        for i in self.Graph[vote]: #进行边的松弛技术
            if i != inf and i > 0 and self.distTo[vote] + i < self.distTo[count]:
                self.edgeTo[count] = vote
                self.distTo[count] = self.distTo[vote] + i
                self.distTo[count] = round(self.distTo[count], 2)
                if count not in d:
                    d.append(count)
            #处理满足条件且含有正(负)权重环的路径情况
            elif i != inf and i < 0 and self.distTo[vote] + i < self.distTo[count]:
                temp  = self.edgeTo[count]          #建立临时空间存储原横切边
                self.edgeTo[count]  = vote
                flage = self.cycle_assert(count)    #判读若该点构成环且该点既是起
                                                    #点又是终点,则存在环
                if flage:                           #有环,消除该环
                    self.edgeTo[count] = temp
                    self.Graph[vote][count] = inf
                else:                               #无环,与第一个 if 相同处理
                    self.distTo[count] = self.distTo[vote] + i
                    self.distTo[count] = round(self.distTo[count], 2)
```

```
                    if count not in d:
                        d.append(count)

                elif i != inf and  self.distTo[vote] + i >= self.distTo[count]:
                    self.Graph[vote][count] = inf       #删除该无用边
                count += 1
        for i in range(len(self.Graph[0])):
            path = self.PrintPath(i)
            print("%d to %d(%.2f):" %(path[0],i,self.distTo[i]),end="")
            if len(path) == 1 and path[0] == self.s:
                print("")
            else:
                for i in path[:-1]:
                        print('%d->' %(i),end = "")
                print(path[-1])
if __name__ == "__main__":
    #含有负权重值的图
    Graph = [[inf,inf,0.26,inf,0.38,inf,inf,inf],
             [inf,inf,inf,0.29,inf,inf,inf,inf],
             [inf,inf,inf,inf,inf,inf,inf,0.34],
             [inf,inf,inf,inf,inf,inf,0.52,inf],
             [inf,inf,inf,inf,inf,0.35,inf,0.37],
             [inf,0.32,inf,inf,0.35,inf,inf,0.28],
             #[0.58,inf,0.40,inf,0.93,inf,inf,inf],
             [-1.40,inf,-1.20,inf,-1.25,inf,inf,inf],
             [inf,inf,inf,0.39,inf,0.28,inf,inf],
            ]
    #路径之中含有负权重环图
    Graph1 = [[inf,inf,0.26,inf,0.38,inf,inf,inf],
              [inf,inf,inf,0.29,inf,inf,inf,inf],
              [inf,inf,inf,inf,inf,inf,inf,0.34],
              [inf,inf,inf,inf,inf,inf,0.52,inf],
              [inf,inf,inf,inf,inf,0.35,inf,0.37],
              [inf,0.32,inf,inf,-0.66,inf,inf,0.28],
              [0.58,inf,0.40,inf,0.93,inf,inf,inf],
              [inf,inf,inf,0.39,inf,0.28,inf,inf],
            ]
    Graph2 = [[inf,0,5,inf,inf,inf],
              [inf,inf,inf,30,35,inf],
              [inf,inf,inf,15,20,inf],
              [inf,inf,inf,inf,inf,20],
              [inf,inf,inf,inf,inf,10],
              [inf,inf,inf,inf,inf,inf],
             ]
    Graph3 = [[inf,0,5,inf],
              [inf,inf,inf,35],
              [inf,-7,inf,inf],
              [inf,inf,inf,inf]]
    F = BellmanFordSP(Graph,0)
    F.bellmanford()
```

运行程序，输出如下：

```
True
```

```
[0, 2, 7, 3, 6, 4]
0 to 0(0.00):
0 to 1(0.93):0->2->7->3->6->4->5->1
0 to 2(0.26):0->2
0 to 3(0.99):0->2->7->3
0 to 4(0.26):0->2->7->3->6->4
0 to 5(0.61):0->2->7->3->6->4->5
0 to 6(1.51):0->2->7->3->6
0 to 7(0.60):0->2->7
```

3.5 SciPy 空间数据

空间数据又称几何数据，它用来表示物体的位置、形态、大小分布等各方面的信息，比如坐标上的点。

SciPy 通过 scipy.spatial 模块处理空间数据，比如判断一个点是否在边界内、计算给定点周围距离最近点以及给定距离内的所有点。

3.5.1 三角测量

三角测量在三角学与几何学上，是一个借由测量目标点与固定基准线以及基准线的已知端点的角度，测量目标距离的方法。多边形的三角测量是将多边形分成多个三角形，可以用这些三角形来计算多边形的面积。

拓扑学的一个已知事实告诉我们：任何曲面都存在三角剖分。

假设曲面上有一个三角剖分，我们把所有三角形的顶点总数记为 p（公共顶点只看成一个），边数记为 a，三角形的个数记为 n，则 $e=p-a+n$ 是曲面的拓扑不变量。也就是说，不管是什么剖分，e 总是得到相同的数值。e 被称为欧拉示性数。

对一系列的点进行三角剖分点方法是 Delaunay()三角剖分。函数的格式为：

class scipy.spatial.Delaunay(points, furthest_site=False, incremental=False, qhull_options=None)。其中，points 为浮点数数组，为要进行三角剖分的点坐标。furthest_site 为布尔型，表示是否计算 furthest-site Delaunay 三角剖分，默认值为 False。incremental 为允许增量添加新点，布尔型，可选。qhull_options 为 str 类型，可选，用于传递给 Qhull 的其他选项。

【例 3-13】 通过给定的点来创建三角形。

```
import numpy as np
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt
%matplotlib inline
points = np.array([
  [2, 4],  [3, 4],  [3, 0],  [2, 2],  [4, 1]])
simplices = Delaunay(points).simplices                    #三角形中顶点的索引
plt.triplot(points[:, 0], points[:, 1], simplices)
plt.scatter(points[:, 0], points[:, 1], color='r')
plt.show()
```

运行程序，效果如图 3-6 所示。

注意：三角形顶点的 id 存储在三角剖分对象的 simplices 属性中。

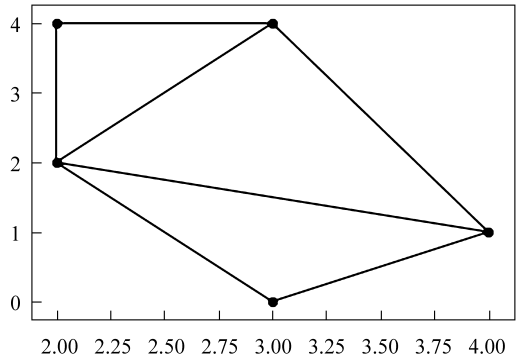

图 3-6　创建的三角形图

3.5.2　凸包

在数学上，实向量空间 $\boldsymbol{V}$ 中的一组点 X 的凸包或凸包络是包含 X 的最小凸集。通俗来说就是包围一组散点的最小凸边形。

scipy 提供了 scipy.spatial 函数计算凸包，scipy 中 convexHull 输入的参数可以是 m2 的点坐标。其返回值的属性.verticess 是所有凸轮廓点在散点(m2)中的索引值。

注意：属性.verticess 绘制出来的轮廓点是按照逆时针排序的。

Scipy 计算得到的凸包如图 3-7 所示。

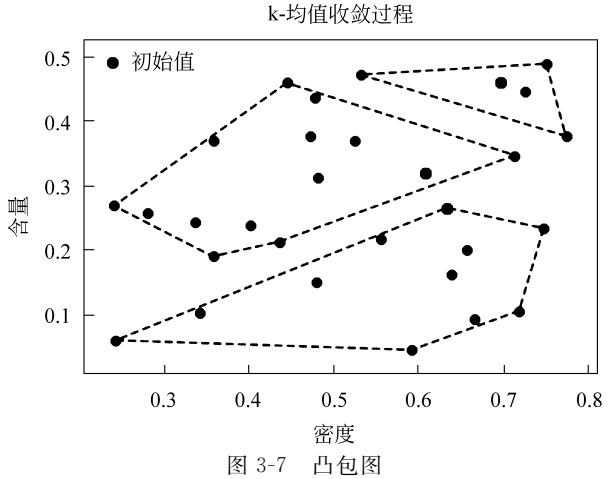

图 3-7　凸包图

【例 3-14】 通过给定的点来创建凸包。

```
import numpy as np
from scipy.spatial import ConvexHull
import matplotlib.pyplot as plt
points = np.array([[2, 4], [3, 4], [3, 0], [2, 2], [4, 1], [1, 2],
                   [5, 0], [3, 1], [1, 2],  [0, 2]
])
hull = ConvexHull(points)
hull_points = hull.simplices
plt.scatter(points[:,0], points[:,1])
for simplex in hull_points:
  plt.plot(points[simplex,0], points[simplex,1], 'r-')
```