# ScientificComputing

August 22, 2022

## 1 Numpy Basics

- Operations
  - type(arr) -> basic array is numpy.ndarray
  - arr.shape
  - arr.tolist()
  - arr[(arr > 2) & (arr < 11)]
  - np.where(arr > 0, arr, 0) -> if not > 0 than make 0
  - np.where(arr > 0, arr)
  - np.arange(start, stop, stride)
  - np.zeros(amount), np.zeros((amount, amount))
  - np.ones(amount)
  - np.linspace(start, stop, number of linearly-spaced numbers)
  - np.eye(amount) -> identity matrix
  - np.random.rand(rows, columns) -> float
  - np.random.randint(start, stop, amount) -> int
  - array.reshape(rows, columns)
  - array.min(axis = 0) -> min of each column
  - array.max(axis = 1) -> max of each row
  - array.argmax(), array.argmin() -> find index of max/min values
  - np.log(array)
  - np.exp(array)
  - vect1.dot(vect2) or np.dot(vect1, vect2) -> inner product
  - np.cross(vect1, vect2)
  - np.dot(Matrix1, Matrix2)
  - np.multiply(Matrix1, Matrix2) -> element wise multiplication
  - np.linalg.inv(Matrix) -> inverse
  - np.linalg.det(Matrix) -> determinant
  - np.trace(Matrix)
  - np.sum(x, axis = 0) -> sum each column; np.sum(x, axis = 1) -> sum row
  - np.poly(array) -> returns coefficients of a polynomial with the given sequence of roots
  - np.roots(array) -> returns roots of a polynomial with the given coefficients
  - np.polyint(array) -> returns an antiderivative (indefinite integral)
  - np.polyder(array) -> returns the derivative of the specified order
  - np.polyval(array, val) -> evaluates polynomial at a specific value
  - np.polyfit(array1, array2, order) -> fits a polynomial of a specified order to a set of data using a least-squares approach
- Broadcasting Rules

- The first rule of broadcasting is that if all input arrays do not have the same number of dimensions, a "1" will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.
- The second rule of broadcasting ensures that arrays with a size of 1 along a particular dimension act as if they had the size of the array with the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcast" array.

[1]:
```
# Used in basics section
# !pip3 install numpy
# !pip3 install matplotlib
# !pip3 install scipy
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist, squareform
# from scipy.misc import imread, imsave, imresize
```

[2]:
```
a = np.array([x for x in range(10)])
a + 2
```

[2]:
```
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

[3]:
```
type(a)
```

[3]:
```
numpy.ndarray
```

[4]:
```
np.array((a, a)).ravel()  # returns the array, flattened
```

[4]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[5]:
```
# Important
# Simple assignments make no copy of objects
b = a
b is a
```

[5]:
```
True
```

### 1.1 Shared Memory in Numpy (POSIX)

[6]:
```
c = a.view()
print(f"C is A: {c is a}")
print(f"C is a view of data owned by A: {c.base is a}")
print(f"C owns data at A: {c.flags.owndata}")
c = c.reshape((2, 5))  # a's shape doesn't change
c[0, 4] = 1234         # a's data changes
print(f"A:\n {a}")
print(f"C:\n {c}")
```

```
C is A: False
C is a view of data owned by A: True
C owns data at A: False
A:
 [   0    1    2    3 1234    5    6    7    8    9]
C:
 [[   0    1    2    3 1234]
 [   5    6    7    8    9]]
```

[7]:
```
# Deep Copy
d = a.copy()
print(f"D is A: {d is a}\nD shares with A: {d.base is a}")
```

```
D is A: False
D shares with A: False
```

[8]:
```
x = np.array(a, dtype=np.float64)
y = np.array(a + 2, dtype=np.float64)
print(np.add(x,y))
(np.subtract(x,y))
```

```
[2.00e+00 4.00e+00 6.00e+00 8.00e+00 2.47e+03 1.20e+01 1.40e+01 1.60e+01
 1.80e+01 2.00e+01]
```

[8]:
```
array([-2., -2., -2., -2., -2., -2., -2., -2., -2., -2.])
```

[9]:
```
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

[10]:
```
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))

# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
[[2 4 6]
 [5 7 9]]
[[ 5  6  7]
```

```
 [ 9 10 11]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 2  4  6]
 [ 8 10 12]]
```

[11]:
```
def f(x, y):
    return 10 * x + y

np.fromfunction(f, (5, 4), dtype=np.float32)
```

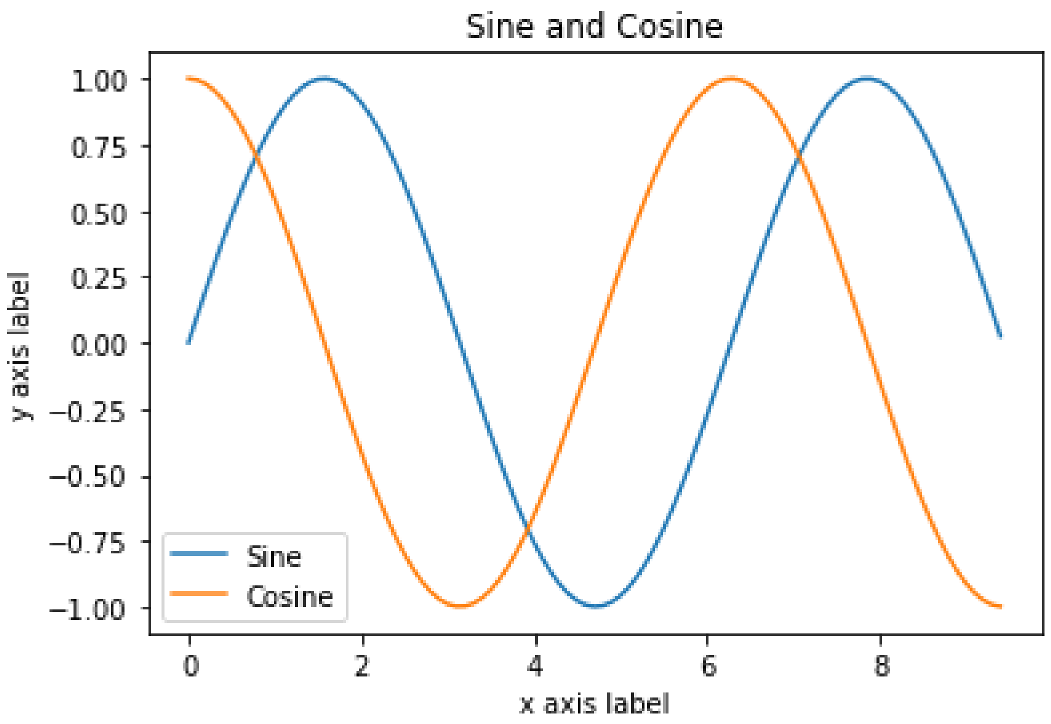
[11]:
```
array([[ 0.,  1.,  2.,  3.],
       [10., 11., 12., 13.],
       [20., 21., 22., 23.],
       [30., 31., 32., 33.],
       [40., 41., 42., 43.]], dtype=float32)
```

[12]:
```
# Distance Between Points
# Create the following array where each row is a point in 2D space:
# [[0 1]
#  [1 0]
#  [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# Compute the Euclidean distance between all rows of x.
# d[i, j] is the Euclidean distance between x[i, :] and x[j, :],
# and d is the following array:
# [[ 0.          1.41421356  2.23606798]
#  [ 1.41421356  0.          1.        ]
#  [ 2.23606798  1.          0.        ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

```
[[0 1]
 [1 0]
 [2 0]]
[[0.         1.41421356 2.23606798]
 [1.41421356 0.         1.        ]
 [2.23606798 1.         0.        ]]
```

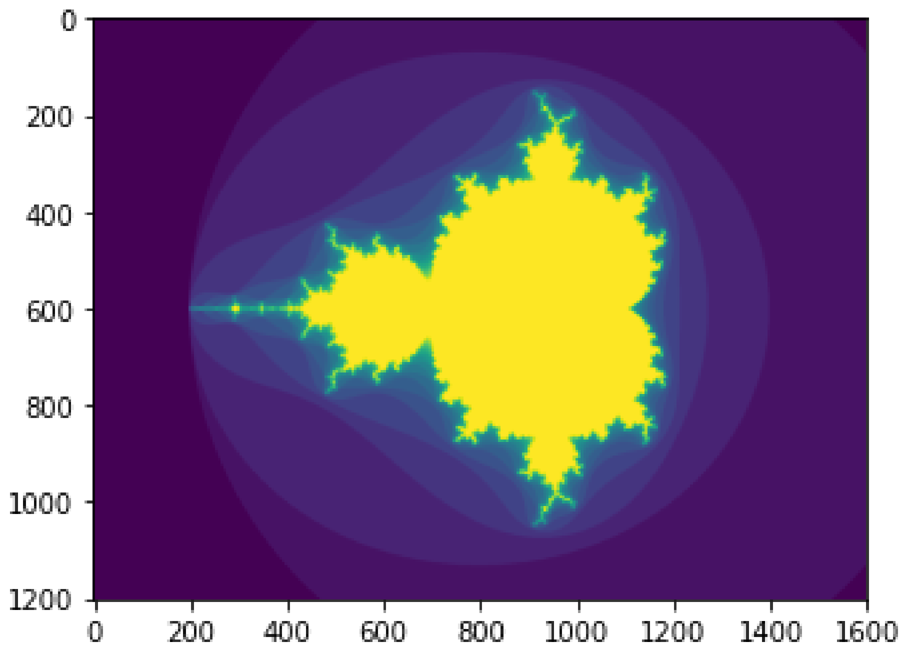
[13]:
```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
```

```
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



[14]:
```
def mandelbrot(h, w, maxit=20, r=2):

    """Returns an image of the Mandelbrot fractal of size (h,w)."""

    x = np.linspace(-2.5, 1.5, 4*h+1)

    y = np.linspace(-1.5, 1.5, 3*w+1)

    A, B = np.meshgrid(x, y)

    C = A + B*1j

    z = np.zeros_like(C)

    divtime = maxit + np.zeros(z.shape, dtype=int)
    for i in range(maxit):
        z = z**2 + C
        diverge = abs(z) > r                    # who is diverging
        div_now = diverge & (divtime == maxit)  # who is diverging now
        divtime[div_now] = i                    # note when
        z[diverge] = r                          # avoid diverging too much

    return divtime
plt.clf()
plt.imshow(mandelbrot(400, 400))
```

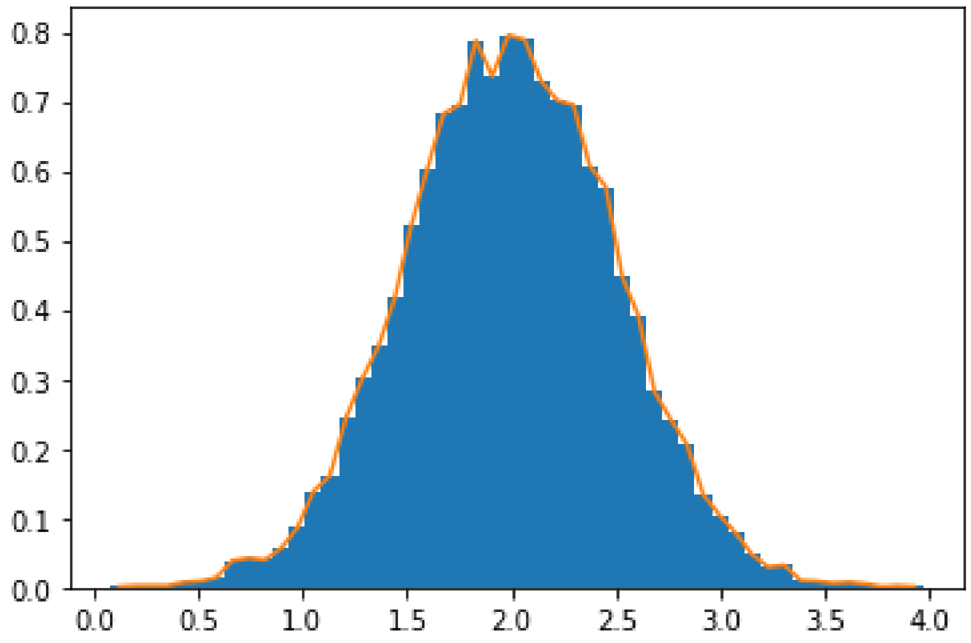
[14]:
```
<matplotlib.image.AxesImage at 0x7f1dc95e21c0>
```



[15]:
```
import numpy as np
```

```
rg = np.random.default_rng(1)

import matplotlib.pyplot as plt

# Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2

mu, sigma = 2, 0.5

v = rg.normal(mu, sigma, 10000)

# Plot a normalized histogram with 50 bins

plt.hist(v, bins=50, density=True)

# Compute the histogram with numpy and then plot it

(n, bins) = np.histogram(v, bins=50, density=True)  # NumPy version (no plot)

plt.plot(.5 * (bins[1:] + bins[:-1]), n)
```

[15]:
```
[<matplotlib.lines.Line2D at 0x7f1dc9559760>]
```



[16]:
```
# Inner Product

A = np.array([0, 1])
B = np.array([3, 4])
```

```
print(np.inner(A, B))
# Output : 4
```

```
4
```

[17]:
```
# Outter Product

A = np.array([0, 1])
B = np.array([3, 4])

print(np.outer(A, B))
```

```
[[0 0]
 [3 4]]
```

[18]:
```
# Computes Eigenvalues And Right EigenVectors
# Of A Square Matrix
vals, vecs = np.linalg.eig([[1 , 2], [2, 1]])
print(vals)
print(vecs)
```

```
[ 3. -1.]
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

[19]:
```
# hpslit -> splits into equally shaped arrays
x = np.arange(1, 25).reshape(2, 12)
np.hsplit(x, 3)
```

[19]:
```
[array([[ 1,  2,  3,  4],
        [13, 14, 15, 16]]),
 array([[ 5,  6,  7,  8],
        [17, 18, 19, 20]]),
 array([[ 9, 10, 11, 12],
        [21, 22, 23, 24]])]
```

[20]:
```
# np.save(FILENAME, arr)
np.save("threeEqualArrays.npy", np.hsplit(x, 3))
```

[21]:
```
!ls
```

```
 cumulative_error.png                 Scientific_Computing_Intro.ipynb
'Hacker Rank Basics For Numpy.pdf'   Sympy.ipynb
 HR_Numpy_Basics.ipynb               Sympy.pdf
 RedPanda.jpg                        threeEqualArrays.npy
 RedPanda.png
```

[22]:
```
print(np.load("threeEqualArrays.npy"))
```

```
[[[ 1  2  3  4]
  [13 14 15 16]]

 [[ 5  6  7  8]
  [17 18 19 20]]

 [[ 9 10 11 12]
  [21 22 23 24]]]
```

## 1.2 Custom Containers

[23]:
```
"""
To support it, we need to define the Python interfaces __add__, __lt__,
and so on to dispatch to the corresponding ufunc. We can achieve this
conveniently by inheriting from the mixin NDArrayOperatorsMixin.
"""

import numpy.lib.mixins
from numbers import Number

class DiagonalArray(numpy.lib.mixins.NDArrayOperatorsMixin):

    def __init__(self, N, value):

        self._N = N

        self._i = value

    def __repr__(self):

        return f"{self.__class__.__name__}(N={self._N}, value={self._i})"

    def __array__(self, dtype=None):

        return self._i * np.eye(self._N, dtype=dtype)

    def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):

        if method == '__call__':

            N = None

            scalars = []

            for input in inputs:
                if isinstance(input, Number):

                    scalars.append(input)

                elif isinstance(input, self.__class__):

                    scalars.append(input._i)

                    if N is not None:

                        if N != self._N:

                            raise TypeError("inconsistent sizes")

                    else:

                        N = self._N

                else:

                    return NotImplemented

            return self.__class__(N, ufunc(*scalars, **kwargs))

        else:

            return NotImplemented
```

[24]:
```
arr = DiagonalArray(5, 1)

print(arr + 3)
print(arr > 0)
```

```
DiagonalArray(N=5, value=4)
DiagonalArray(N=5, value=True)
```

[25]:
```
"""
Now let's tackle __array_function__.
We'll create dict that maps numpy functions to our custom variants.
"""
HANDLED_FUNCTIONS = {}

class DiagonalArray(numpy.lib.mixins.NDArrayOperatorsMixin):

    def __init__(self, N, value):
        self._N = N

        self._i = value

    def __repr__(self):

        return f"{self.__class__.__name__}(N={self._N}, value={self._i})"

    def __array__(self, dtype=None):

        return self._i * np.eye(self._N, dtype=dtype)

    def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):

        if method == '__call__':

            N = None

            scalars = []

            for input in inputs:

                # In this case we accept only scalar numbers or DiagonalArrays.

                if isinstance(input, Number):

                    scalars.append(input)

                elif isinstance(input, self.__class__):

                    scalars.append(input._i)

                    if N is not None:

                        if N != self._N:

                            raise TypeError("inconsistent sizes")

                    else:

                        N = self._N

                else:

                    return NotImplemented

            return self.__class__(N, ufunc(*scalars, **kwargs))
        else:

            return NotImplemented

    def __array_function__(self, func, types, args, kwargs):

        if func not in HANDLED_FUNCTIONS:

            return NotImplemented

        # Note: this allows subclasses that don't override

        # __array_function__ to handle DiagonalArray objects.

        if not all(issubclass(t, self.__class__) for t in types):

            return NotImplemented

        return HANDLED_FUNCTIONS[func](*args, **kwargs)


def implements(np_function):

   "Register an __array_function__ implementation for DiagonalArray objects."

   def decorator(func):

       HANDLED_FUNCTIONS[np_function] = func

       return func

   return decorator
```

[26]:
```
@implements(np.sum)
def sum(arr):
    "Implementation of np.sum for DiagonalArray objects"
    return arr._i * arr._N


@implements(np.mean)
def mean(arr):
    "Implementation of np.mean for DiagonalArray objects"
    return arr._i / arr._N


arr = DiagonalArray(5, 1)
```

```
print(np.sum(arr))
print(np.mean(arr))
```

```
5
0.2
```

[27]:
```
from numpy import array, argmin, sqrt, sum

observation = array([111.0, 188.0])

codes = array([[102.0, 203.0],

               [132.0, 193.0],

               [45.0, 155.0],

               [57.0, 173.0]])

diff = codes - observation    # the broadcast happens here

dist = sqrt(sum(diff**2,axis=-1))

argmin(dist)
```

[27]:
```
0
```

### 1.3 All Hacker Rank Numpy Problems

- Arrays
- Transpose
- Concatenate
- Shape
- Array Mathematics
- Eye and Identity
- Zeros and Ones
- Floor, Ceil, Rint -> Rint mean round to nearest int element-wise
- Min Max
- Sum
- Mean, Variance, Standard Deviation
- Dot and Cross Products
- Inner and Outer Products
- Polynomials
- Linear Algebra

[ ]:
```
# Arrays
def arrays(arr):
    return numpy.array(arr[::-1], dtype=float)

arr = "1 2 3 4 -8 -10".strip().split(' ')
result = arrays(arr)
print(result)
```

[ ]:
```
# Transpose
testCases = int(input().split()[0])

arr = numpy.array([input().split() for _ in range(testCases)], dtype=int)

print(arr.transpose())
print(arr.flatten())
```

[ ]:
```
# On the transpose problem, if you wanted to
# populate row indices first
rows, cols = map(int, input().split())
arr = [[] for _ in range(rows)]

for colIdx in range(cols):
    tmpVect = list(map(int, input().split()))
    idx = 0
    for rowIdx in range(rows):
        arr[rowIdx].append(tmpVect[idx])
        idx += 1

arr = numpy.array(arr, dtype=int)

print(numpy.transpose(arr))
print(arr.flatten())
```

[ ]:
```
# Concatenate
N, M, P = map(int, input().split())

A = numpy.array([list(map(int, input().split())) for _ in range(N)], dtype=int)
B = numpy.array([list(map(int, input().split())) for _ in range(M)], dtype=int)

C = numpy.concatenate((A, B), axis = 0)

print(C)
```

[ ]:
```
# Shape
numpy.array(input().split(), dtype=int).reshape(3,3)
```

[ ]:
```
# Array Mathematics
N, M = map(int, input().split())
```

```
A = numpy.array([input().split() for _ in range(N)], dtype=int)
B = numpy.array([input().split() for _ in range(N)], dtype=int)

print(numpy.add(A, B))
print(numpy.subtract(A, B))
print(numpy.multiply(A, B))
print(A // B)
print(numpy.mod(A, B))
print(A ** B)
```

[ ]:
```
# Eye and Identity
# K = 0 equals diagonal, K > 0 == upper T,
# K < 0 == lower T
numpy.set_printoptions(legacy='1.13')

print(numpy.eye(*map(int, input().split())))
```

[ ]:
```
# Zeros and Ones
args = tuple(map(int, input().split()))

print(numpy.zeros(args, dtype=int))
print(numpy.ones(args, dtype=int))
```

[ ]:
```
# Floor, Ceil, Rint -> Rint mean round to nearest int element-wise
numpy.set_printoptions(legacy='1.13')

A = numpy.array(input().split(), dtype=float)
print(numpy.floor(A))
print(numpy.ceil(A))
print(numpy.rint(A))
```

[ ]:
```
# Min Max
print(max(numpy.min(numpy.array([input().split() for _ in range(list(map(int,␣
↪input().split()))[0])], dtype=int), axis = 1)))
```

[ ]:
```
# Sum
numpy.prod(numpy.sum(numpy.array([input().split() for _ in range(list(map(int,␣
↪input().split()))[0])], dtype=int), axis=0))
```

[ ]:
```
# Mean, Variance, Standard Deviation
A = numpy.array([input().split() for _ in range(list(map(int, input().
↪split()))[0])], dtype=int)

print(numpy.round_(numpy.mean(A, axis = 1), 11))
print(numpy.round_(numpy.var(A, axis = 0), 11))
print(numpy.round_(numpy.std(A, axis = None), 11))
```

[ ]:
```
# Dot and Cross Products
N = int(input().strip())

print(numpy.dot(numpy.array([input().split() for _ in range(N)], dtype=int),␣
↪numpy.array([input().split() for _ in range(N)], dtype=int)))
```

[ ]:
```
# Inner and Outer Products
vectA = numpy.array(input().split(), dtype=int)
vectB = numpy.array(input().split(), dtype=int)

print(numpy.inner(vectA, vectB))
print(numpy.outer(vectA, vectB))
```

[ ]:
```
# Polynomials
print(np.polyval(np.array(input().split(),dtype=float),float(input())))
```

[ ]:
```
# Linear Algebra
print(numpy.round_(numpy.linalg.det(numpy.array([input().split() for _ in␣
↪range(int(input().strip()))], dtype=float)), 2))
```

## 2 Sympy (Symbolic Mathematics) Basics

[28]:
```
# Needed for code
#!pip3 install sympy
import sympy as sp
import numpy as np

from sympy import symbols, Integral, cos, sin, exp, Eq, diff, solveset, sinh,␣
↪Function, cosh, N, limit, oo, integrate, expand, factor, dsolve, plot,␣
↪lambdify
from sympy.plotting import plot3d

import matplotlib.pyplot as plt
```

[29]:
```
# Create symbols
x, y = sp.symbols("x y")
```

[30]:
```
diffyQ = (x + y) / x
```

[31]:
```
# Solve a polynomial
# x**2 + 5*x + 6 = 0
equation = Eq(x**2 + 5*x + 6, 0)
solution = solveset(equation, x)
print(type(solution))
print(list(solution))
```

```
<class 'sympy.sets.sets.FiniteSet'>
[-3, -2]
```

[32]:
```
solveset(Eq(sinh(x) ** 2 + cosh(x) ** 2, 0))
```

[32]:
$$\left\{\frac{i\left(2n\pi + \frac{\pi}{2}\right)}{2}\,\middle|\,n \in \mathbb{Z}\right\} \cup \left\{\frac{i\left(2n\pi - \frac{\pi}{2}\right)}{2}\,\middle|\,n \in \mathbb{Z}\right\}$$

[33]:
```
# Machine learning activation function
sigmoid = 1/(1 + exp(-x))
sigmoid.subs(x, 2)
```

[33]:
$$\frac{1}{e^{-2} + 1}$$

[34]:
```
N(sigmoid.subs(x, 2))
```

[34]:
$$0.880797077977882$$

## 2.1 Limits

[35]:
```
# lim_x->0 sin(x)/x
limit(sin(x)/x, x, 0)
```

[35]:
$$1$$

[36]:
```
n = symbols('n')
f = (1 + (1/n))**n
result = limit(f, n, oo)

print(result) # E
print(result.evalf())
```

```
E
2.71828182845905
```

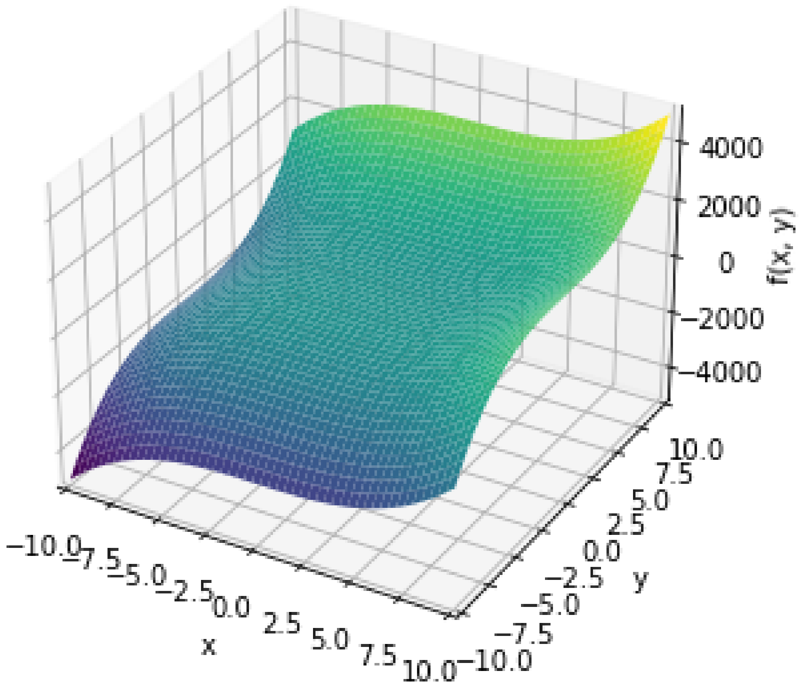
## 2.2 Derivatives

[37]:
```
diff(sin(x) * exp(x), x)
```

[37]:
$$e^{x}\sin(x) + e^{x}\cos(x)$$

[38]:
```
# Now just use Python syntax to declare function
f = 2*x**3 + 3*y**3

# Calculate the partial derivatives for x and y
dx_f = diff(f, x)
dy_f = diff(f, y)

print(dx_f) # prints 6*x**2
print(dy_f) # prints 9*y**2

# plot the function
plot3d(f)
```

```
6*x**2
9*y**2
```



[38]: `<sympy.plotting.plot.Plot at 0x7f1e06dac190>`

## 2.3 Modifying Formulas

[39]:
```
a, b = symbols('a b')
expr = a + 2 * b
print(expr + 1)
expr -= a
print(expr)
```

```
a + 2*b + 1
2*b
```

[40]:
```
expanded_expr = expand(x*x*(expr + a))
print(expanded_expr)
print(factor(expanded_expr))
```

```
a*x**2 + 2*b*x**2
x**2*(a + 2*b)
```

## 2.4 Integration

[41]:
```
z = symbols('z')
formula = Integral(cos(z)*exp(z), z)
Eq(formula, formula.doit())
```

[41]:
$$\int e^{z}\cos\left(z\right)dz = \frac{e^{z}\sin\left(z\right)}{2} + \frac{e^{z}\cos\left(z\right)}{2}$$

[42]:
```
integrate(cos(z) * exp(z), z)
```

[42]:
$$\frac{e^{z}\sin\left(z\right)}{2} + \frac{e^{z}\cos\left(z\right)}{2}$$

[43]:
```
integrate(1/z, (z, -oo,oo))
```

[43]: NaN

[44]:
```
integrate(z**2, (z, 5, 10))
```

[44]:
$$\frac{875}{3}$$

## 2.5 Series

- series(EXPONENT, x0 = stopping)
- series(EXPONENT, START, STOP)

[45]:
```
print("First")
expr1 = exp(sin(x))
print(expr1.series(x, 0, 4).removeO())
print()

print("Second")
expr2 = exp(x - 6)
print(expr2.series(x, x0=6).removeO())
```

```
First
x**2/2 + x + 1

Second
x + (x - 6)**5/120 + (x - 6)**4/24 + (x - 6)**3/6 + (x - 6)**2/2 - 5
```

### 2.6 Differential Equation

- $y^{''} - y = e^t$

```
[46]: t = symbols('t')
      m = Function('m')
      dsolve(Eq(m(t).diff(t, t) - m(t), exp(t)), m(t))
```

[46]:
$$m(t) = C_2 e^{-t} + \left(C_1 + \frac{t}{2}\right) e^t$$

### 2.7 Interface With Numpy

- lambdify -> convert SymPy expression to NumPy for speed
  - Converts SymPy names to NumPy names
  - Uses eval so make sure to use input validation

```
[47]: a = np.arange(10)
      expr = sin(x)
      f = lambdify(x, expr, "numpy")
      f(a)
```

```
[47]: array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
             -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

### 2.8 Interface With Math

- To use lambdify with numerical libraries that it does not know about, pass a dictionary of sympy_name:numerical_function pairs.
  - "sin" used as a generic placeholder

```
[48]: f = lambdify(x, expr, "math")
      f(0.1)
```

```
[48]: 0.09983341664682815
```

```
[49]: import cmath

      def convertToPolar(complexNum):
          return cmath.polar(complexNum)

      f = lambdify(x, expr, {"sin": convertToPolar})
      f(complex(60,10))
```

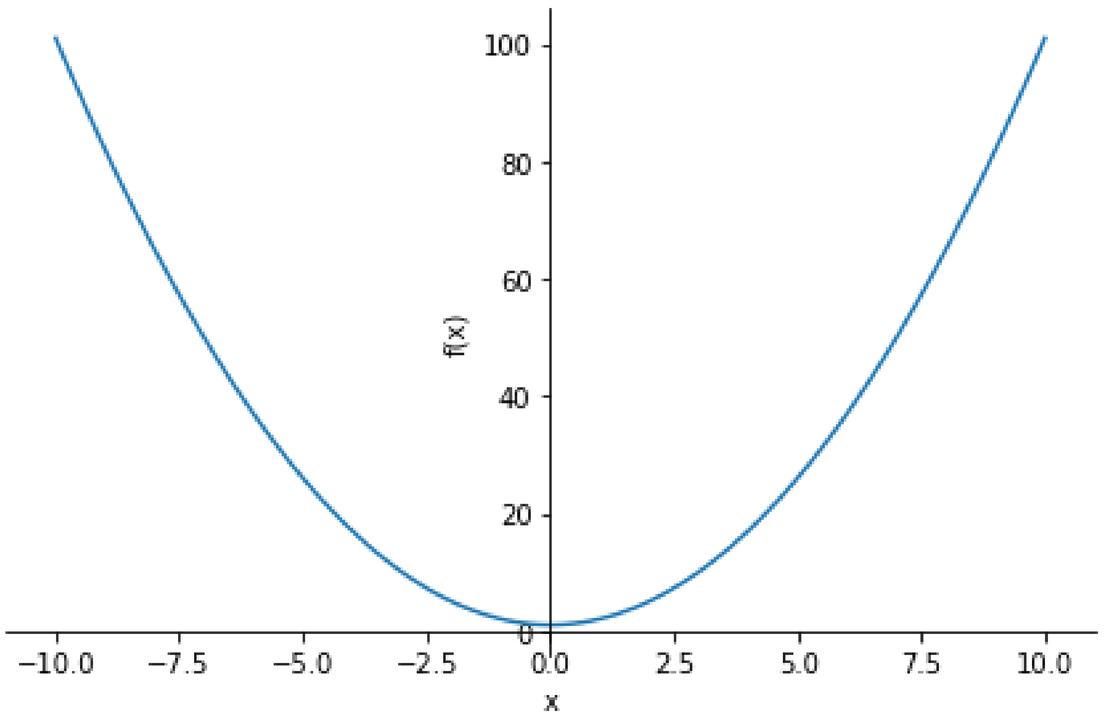
```
[49]: (60.8276253029822, 0.16514867741462683)
```

```
[50]: cmath.polar(complex(60,10))
```

```
[50]: (60.8276253029822, 0.16514867741462683)
```

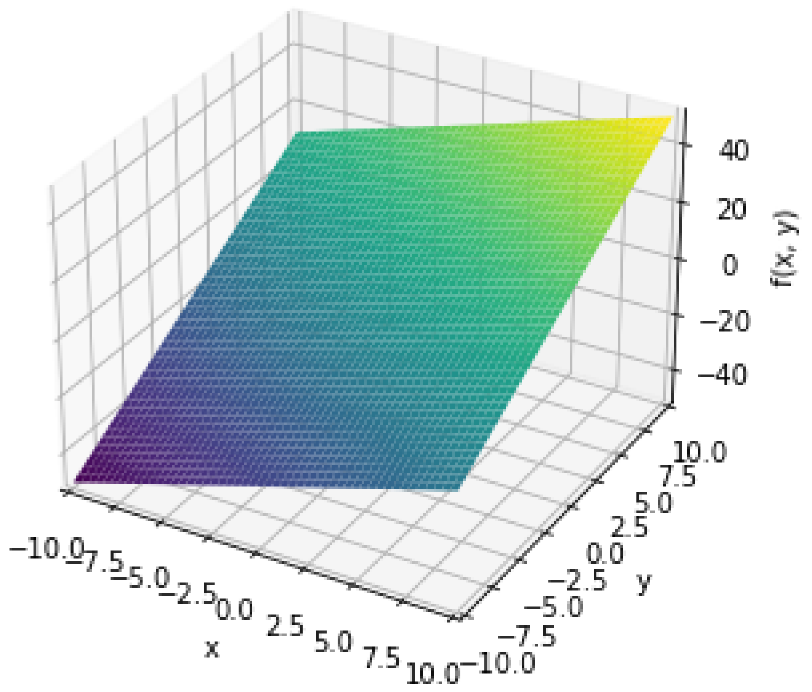
### 2.9 Plotting

[51]:
```
f = x**2 + 1
plot(f)
```



[51]:
```
<sympy.plotting.plot.Plot at 0x7f1de1ec7a30>
```

[52]:
```
f = 2 * x + 3 * y
plot3d(f)
```



```
[52]: <sympy.plotting.plot.Plot at 0x7f1de1ec7130>
```

## 2.10 Vectors And Things

```
[53]: from typing import List

class Vector:
    def __init__(self, *coordinates: List, _dtype = "float"):
        exec(f"self.coords = np.array({coordinates}, dtype={_dtype})")

    def __abs__(self):
        return np.sqrt(np.sum(self.coords**2))

RISC_V_Processor_Noise = Vector([0.42, 1.5, 0.87])
print(abs(RISC_V_Processor_Noise))
```

```
1.7841804841439108
```

## 2.11 Optimization

- Good for integer programming (multivariate functions/systems of equations)
- minimize()
  - brent is an implementation of Brent's algorithm. This method is the default.

  - golden is an implementation of the golden-section search. The documentation notes that Brent's method is usually better.
  - bounded is a bounded implementation of Brent's algorithm. It's useful to limit the search region when the minimum is in a known range.
  - Can handle multiple inputs, constraints
    * LinearConstraint: The solution is constrained by taking the inner product of the solution x values with a user-input array and comparing the result to a lower and upper bound.
    * NonlinearConstraint: The solution is constrained by applying a user-supplied function to the solution x values and comparing the return value with a lower and upper bound.
    * Bounds: The solution x values are constrained to lie between a lower and upper bound.

[54]:
```
"""
Example will work with y = 3x**4 - 2x + 1 from [0,1]
    - Scalar function is a quartic polynomial
    - Goal: minimize the function
"""

from scipy.optimize import minimize_scalar

def objective_function(x):
    return 3 * x ** 4 - 2 * x + 1

res = minimize_scalar(objective_function)
res
```

[54]:
```
     fun: 0.17451818777634331
 message: '\nOptimization terminated successfully;\nThe returned value satisfies
the termination criteria\n(using xtol = 1.48e-08 )'
    nfev: 16
     nit: 12
 success: True
       x: 0.5503212087491959
```

[55]:
```
from scipy.optimize import minimize, LinearConstraint

n_buyers = 10
n_shares = 15

np.random.seed(10)
prices = np.random.random(n_buyers)
money_available = np.random.randint(1, 4, n_buyers)
n_shares_per_buyer = money_available / prices

print(prices, money_available, n_shares_per_buyer, sep="\n")
```

```
[0.77132064 0.02075195 0.63364823 0.74880388 0.49850701 0.22479665
 0.19806286 0.76053071 0.16911084 0.08833981]
[1 1 1 3 1 3 3 2 1 1]
[ 1.29647768 48.18824404  1.57816269  4.00638948  2.00598984 13.34539487
 15.14670609  2.62974258  5.91328161 11.3199242 ]
```

[56]:
```
"""
In this code, you create an array of ones with the length n_buyers and pass it␣
 ↪as the first argument to LinearConstraint. Since LinearConstraint takes the␣
 ↪dot product of the solution vector with this argument, it’ll result in the␣
 ↪sum of the purchased shares.

This result is then constrained to lie between the other two arguments:


    The lower bound lb
    The upper bound ub

Since lb = ub = n_shares, this is an equality constraint because the sum of the␣
 ↪values must be equal to both lb and ub. If lb were different from ub, then␣
 ↪it would be an inequality constraint.

Next, create the bounds for the solution variable. The bounds limit the number␣
 ↪of shares purchased to be 0 on the lower side and n_shares_per_buyer on the␣
 ↪upper side. The format that minimize() expects for the bounds is a sequence␣
 ↪of tuples of lower and upper bounds:
"""
constraint = LinearConstraint(np.ones(n_buyers), lb=n_shares, ub=n_shares)
bounds = [(0, n) for n in n_shares_per_buyer]

"""
In this code, res is an instance of OptimizeResult, just like with␣
 ↪minimize_scalar(). As you’ll see, there are many of the same fields, even␣
 ↪though the problem is quite different. In the call to minimize(), you pass␣
 ↪five arguments:


    objective_function: The first positional argument must be the function that␣
 ↪you’re optimizing.


    x0: The next argument is an initial guess for the values of the solution.␣
 ↪In this case, you’re just providing a random array of values between 0 and␣
 ↪10, with the length of n_buyers. For some algorithms or some problems,␣
 ↪choosing an appropriate initial guess may be important. However, for this␣
 ↪example, it doesn’t seem too important.
```

```
    args: The next argument is a tuple of other arguments that are necessary to␣
↪be passed into the objective function. minimize() will always pass the␣
↪current value of the solution x into the objective function, so this␣
↪argument serves as a place to collect any other input necessary. In this␣
↪example, you need to pass prices to objective_function(), so that goes here.

    constraints: The next argument is a sequence of constraints on the problem.␣
↪You're passing the constraint you generated earlier on the number of␣
↪available shares.

    bounds: The last argument is the sequence of bounds on the solution␣
↪variables that you generated earlier.

"""

def objective_function(x, prices):
    return -x.dot(prices)

res = minimize(
    objective_function,
    x0=10 * np.random.random(n_buyers),
    args=(prices,),
    constraints=constraint,
    bounds=bounds,
)

print(res)
```

```
     fun: -8.783020157087615
     jac: array([-0.7713207 , -0.02075195, -0.63364828, -0.74880385,
-0.49850702,
       -0.22479653, -0.19806278, -0.76053071, -0.16911077, -0.08833981])
 message: 'Optimization terminated successfully'
    nfev: 187
     nit: 17
    njev: 17
  status: 0
 success: True
       x: array([1.29647768e+00, 1.07635367e-13, 1.57816269e+00, 4.00638948e+00,
       2.00598984e+00, 3.48323773e+00, 3.99680289e-15, 2.62974258e+00,
       1.94817649e-14, 2.79247930e-14])
```

[57]:
```
print("The total number of shares is:", sum(res.x))

print("Leftover money for each buyer:", money_available - res.x * prices)
```

```
The total number of shares is: 15.000000000000002
```

```
Leftover money for each buyer: [1.37667655e-14 1.00000000e+00 1.17683641e-14
2.62012634e-14
 1.36557432e-14 2.21697984e+00 3.00000000e+00 2.30926389e-14
 1.00000000e+00 1.00000000e+00]
```

## 2.12 Stochastic Gradient Descent

- Cost (loss) function -> minimized/maximized by variables (difference between actual and predicted)
  - Difference is called the residual
  - Sum of squared residuals (SSR) -> $SSR = \Sigma_i (y_i - f(x_i))^2$
  - Mean squared error (MSE) -> $MSE = \frac{SSR}{n}$
  - Both SSR and MSE use the square of the difference between the actual and predicted outputs. The lower the difference, the more accurate the prediction. A difference of zero indicates that the prediction is equal to the actual data.
  - SSR or MSE is minimized by adjusting the model parameters. For example, in linear regression, you want to find the function  ( ) =   +     +  +    , so you need to determine the weights  ,  , ...,   that minimize SSR or MSE.
  - In a classification problem, the outputs  are categorical, often either 0 or 1. For example, you might try to predict whether an email is spam or not. In the case of binary outputs, it's convenient to minimize the cross-entropy function that also depends on the actual outputs   and the corresponding predictions $p(x_i)$
    - $H = -\Sigma_i (y_i * log(p(x_i)) + (1 - y_i) * log(1 - p(x_i)))$
  - In logistic regression, which is often used to solve classification problems, the functions  ( ) and  ( ) are defined as the following
    - $p(x) = \frac{1}{1 + exp(-f(x))}$
    - $f(x) = b_0 + b_1 * x_1 + ... + b_r * x_r$
    - Again, you need to find the weights  ,  , ...,  , but this time they should minimize the cross-entropy function.
- The gradient of a function   of several independent variables  , ...,   is denoted with  (  , ...,  ) and defined as the vector function of the partial derivatives of   with respect to each independent variable:   = (  /  , ...,  /  ). The symbol   is called nabla.
- The nonzero value of the gradient of a function   at a given point defines the direction and rate of the fastest increase of  . When working with gradient descent, you're interested in the direction of the fastest decrease in the cost function. This direction is determined by the negative gradient, −  .
- Once you have a random starting point   = (  , ...,   ), you update it, or move it to a new position in the direction of the negative gradient:  →  −   , where  (pronounced "ee-tah") is a small positive value called the learning rate.
- Example cost function for ordinary least squares $C = \frac{SSR}{2n}$
  - First, you need calculus to find the gradient of the cost function   = Σ (  −   −   )² / (2 ). Since you have two decision variables,   and  , the gradient   is a vector with two components
    - $\frac{\partial C}{\partial b_0} = (\frac{1}{n})\Sigma_i (b_0 + b_1 x_i - y_i) = mean(b_0 + b_1 x_i - y_i)$
    - $\frac{\partial C}{\partial b_0} = (\frac{1}{n})\Sigma_i (b_0 + b_1 x_i - y_i) x_i = mean(b_0 + b_1 x_i - y_i) x_i$
- Online stochastic gradient descent is a variant of stochastic gradient descent in which you estimate the gradient of the cost function for each observation and update the decision variables accordingly. This can help you find the global minimum, especially if the objective function

is convex.

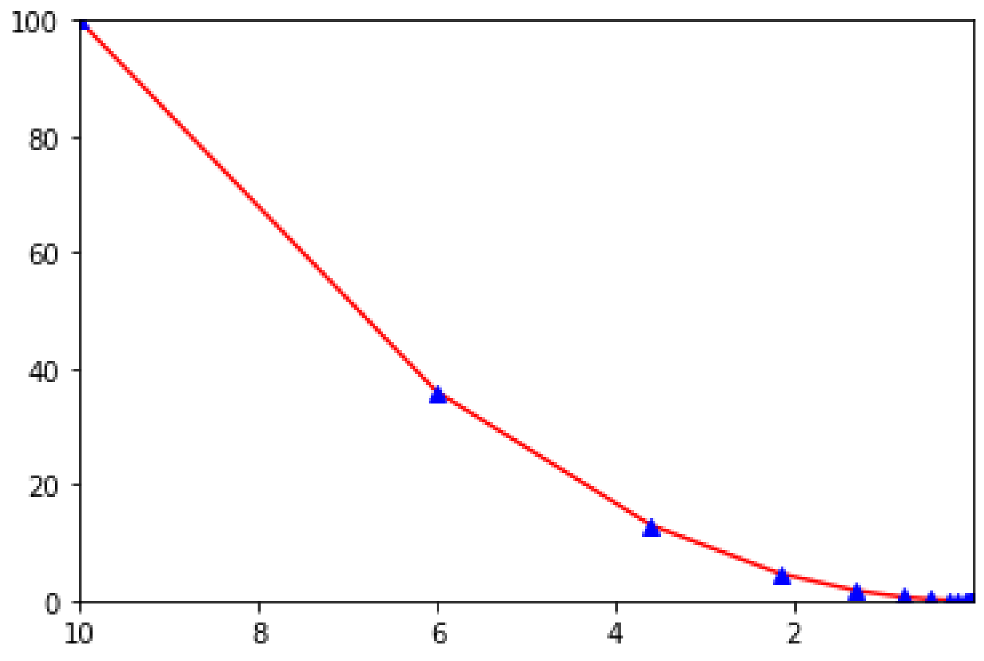
- Batch stochastic gradient descent is somewhere between ordinary gradient descent and the online method. The gradients are calculated and the decision variables are updated iteratively with subsets of all observations, called minibatches. This variant is very popular for training neural networks.

[58]:
```
"""
The precision of numpy.number subclasses is treated as a covariant
generic parameter (see NBitBase), simplifying the annotating of
processes involving precision-based casting.
"""

from typing import TypeVar, Union
from typing_extensions import NewType
import numpy.typing as npt

# Python 3.9
# T = TypeVar("T", bound=npt.NBitBase)
# F = TypeVar("F")
# Python 3.9 AI_Array = NewType('AI_Array', Union[np.floating[T], np.int[T]])
# def gradient_descent(gradient: F, start: AI_Array, learn_rate: int,␣
 ↪n_iterations: int) -> AI_Array:

def gradientDescent(gradient, startingArray, learningRate, numIterations):
    vector = startingArray
    for _ in range(numIterations):
        diff = -1 * learningRate * gradient(vector)
        vector += diff
    return vector
```

[59]:
```
"""
Now have the additional parameter tolerance,
which specifies the minimal allowed movement in each iteration
"""

from sympy import *
v = symbols("v")
equation = v**2

def gradientFunction(option, equation, variables):
    if option == "derivative":
        return diff(equation, *variables)
    else:
        return equation

def gradient(equation, variables, option):
    derivedEquation = gradientFunction(option, equation, variables)
    return lambdify(*variables, derivedEquation, "numpy"), derivedEquation

def gradientDescent2(gradientFunction, equation, variables, startingArray,␣
 ↪learningRate,
                     option="derivative", numIterations=50, tolerance=1e-06):
    array = startingArray
    original = lambdify(*variables, equation, "numpy")
    f, derivedEquation = gradient(equation, variables, option)
    plot_values = []
    for _ in range(numIterations):
        plot_values.append((array[0], original(array)[0]))
        diff = -1 * learningRate * f(array)
        if np.all(np.abs(diff) <= tolerance):
            break
        array = np.add(array, diff)
    return array, plot_values
```

[60]:
```
array = np.array([10]*10, dtype=float)
learningRate = 0.2

array, plot_values = gradientDescent2(gradientFunction, equation, [v], array,␣
 ↪learningRate)
t = [x[0] for x in plot_values]
z = [x[1] for x in plot_values]

plt.plot(t, z, 'r', t, z, 'b^')

plt.xlim([max(t), min(t)])
plt.ylim([min(z), max(z)])
```

[60]:
```
(4.887367798068914e-12, 100.0)
```



[61]:
```
"""
You can use momentum to correct the effect of the learning rate.
The idea is to remember the previous update of the vector and
apply it when calculating the next one. You don’t move the vector
exactly in the direction of the negative gradient, but you also
tend to keep the direction and magnitude from the previous move.

The parameter called the decay rate or decay factor defines how
strong the contribution of the previous update is. To include the
momentum and the decay rate, you can modify sgd() by adding the
parameter decay_rate and use it to calculate the direction and
magnitude of the vector update (diff).

Uses random start below.
"""

def ssr_gradient(x, y, b):
    res = b[0] + b[1] * x - y
    return res.mean(), (res * x).mean()  # .mean() is a method of np.ndarray

def sgd(
    gradient, x, y, n_vars=None, start=None, learn_rate=0.1,
    decay_rate=0.0, batch_size=1, n_iter=50, tolerance=1e-06,
    dtype="float64", random_state=None
):

    # Checking if the gradient is callable
    if not callable(gradient):
        raise TypeError("'gradient' must be callable")


    # Setting up the data type for NumPy arrays
    dtype_ = np.dtype(dtype)


    # Converting x and y to NumPy arrays
    x, y = np.array(x, dtype=dtype_), np.array(y, dtype=dtype_)
    n_obs = x.shape[0]

    if n_obs != y.shape[0]:
        raise ValueError("'x' and 'y' lengths do not match")

    xy = np.c_[x.reshape(n_obs, -1), y.reshape(n_obs, 1)]

    # Initializing the random number generator
    seed = None if random_state is None else int(random_state)
    rng = np.random.default_rng(seed=seed)

    # Initializing the values of the variables
    vector = (
        rng.normal(size=int(n_vars)).astype(dtype_)
        if start is None else
        np.array(start, dtype=dtype_)
    )

    # Setting up and checking the learning rate
    learn_rate = np.array(learn_rate, dtype=dtype_)

    if np.any(learn_rate <= 0):
        raise ValueError("'learn_rate' must be greater than zero")

    # Setting up and checking the decay rate
    decay_rate = np.array(decay_rate, dtype=dtype_)

    if np.any(decay_rate < 0) or np.any(decay_rate > 1):
        raise ValueError("'decay_rate' must be between zero and one")

    # Setting up and checking the size of minibatches
    batch_size = int(batch_size)

    if not 0 < batch_size <= n_obs:
        raise ValueError(
            "'batch_size' must be greater than zero and less than "
            "or equal to the number of observations"
        )

    # Setting up and checking the maximal number of iterations
    n_iter = int(n_iter)

    if n_iter <= 0:
        raise ValueError("'n_iter' must be greater than zero")

    # Setting up and checking the tolerance
    tolerance = np.array(tolerance, dtype=dtype_)

    if np.any(tolerance <= 0):
        raise ValueError("'tolerance' must be greater than zero")


    # Setting the difference to zero for the first iteration
    diff = 0

    # Performing the gradient descent loop
    for _ in range(n_iter):
        # Shuffle x and y
        rng.shuffle(xy)

        # Performing minibatch moves
        for start in range(0, n_obs, batch_size):
            stop = start + batch_size
            x_batch, y_batch = xy[start:stop, :-1], xy[start:stop, -1:]

            # Recalculating the difference
            grad = np.array(gradient(x_batch, y_batch, vector), dtype_)
            diff = decay_rate * diff - learn_rate * grad

            # Checking if the absolute difference is small enough
            if np.all(np.abs(diff) <= tolerance):
                break

            # Updating the values of the variables
            vector += diff

    return vector if vector.shape else vector.item()
```

[62]:
```
sgd(ssr_gradient, np.array([5, 15, 25, 35, 45, 55]),np.array([5, 20, 14, 32,␣
↪22, 38]), start=[0.5, 0.5], learn_rate=0.0008,batch_size=3, n_iter=100_000,␣
↪random_state=0)
```

[62]: array([5.63093736, 0.53982921])

### 2.12.1 Activation Functions

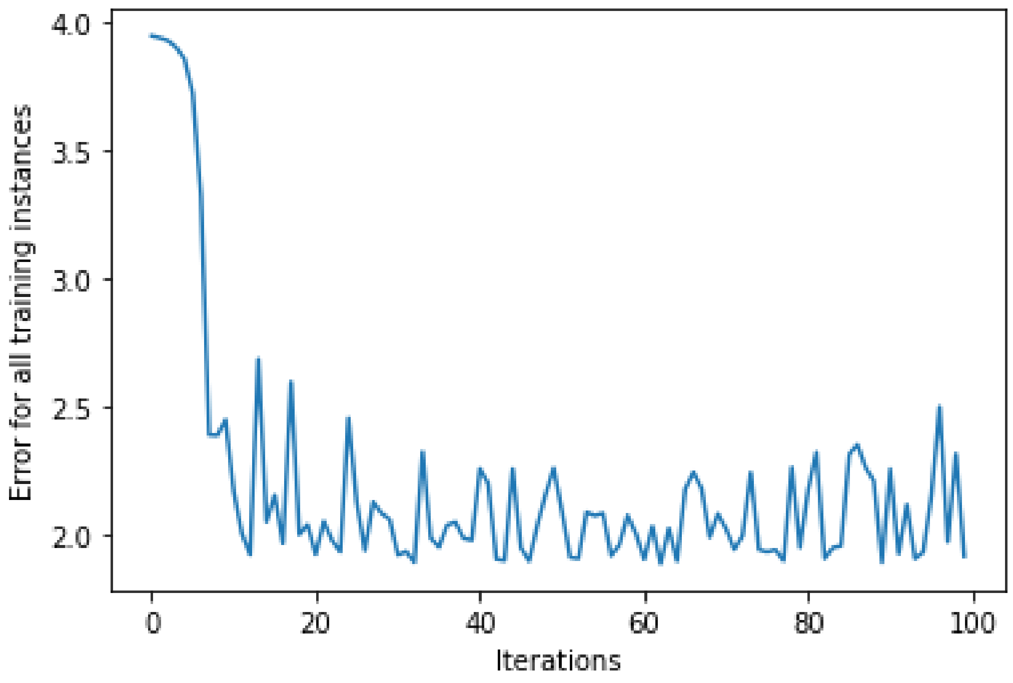
[63]:
```python
from sympy import *
x = Symbol('x')
leaky_relu = Piecewise((0.01*x,x<0),(x, x>=0))

# test leaky relu
leaky_relu.subs(x, 5)  # gives 5
leaky_relu.subs(x, -5) # gives -0.05

# integrate leaky relu
integrate(leaky_relu, x)
```

[63]:
$$\begin{cases} 0.005x^2 & \text{for } x < 0 \\ \frac{x^2}{2} & \text{otherwise} \end{cases}$$

[64]:
```python
def relu(x):
    return(np.maximum(0, x))
```

[65]:
```python
class NeuralNetwork:
    def __init__(self, learning_rate):
        self.weights = np.array([np.random.randn(), np.random.randn()])
        self.bias = np.random.randn()
        self.learning_rate = learning_rate

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def _sigmoid_deriv(self, x):
        return self._sigmoid(x) * (1 - self._sigmoid(x))

    def predict(self, input_vector):
        layer_1 = np.dot(input_vector, self.weights) + self.bias
        layer_2 = self._sigmoid(layer_1)
        prediction = layer_2
        return prediction

    def _compute_gradients(self, input_vector, target):
        layer_1 = np.dot(input_vector, self.weights) + self.bias
        layer_2 = self._sigmoid(layer_1)
        prediction = layer_2

        derror_dprediction = 2 * (prediction - target)
        dprediction_dlayer1 = self._sigmoid_deriv(layer_1)
        dlayer1_dbias = 1
        dlayer1_dweights = (0 * self.weights) + (1 * input_vector)

        derror_dbias = (
            derror_dprediction * dprediction_dlayer1 * dlayer1_dbias
        )
        derror_dweights = (
            derror_dprediction * dprediction_dlayer1 * dlayer1_dweights
        )

        return derror_dbias, derror_dweights

    def _update_parameters(self, derror_dbias, derror_dweights):
        self.bias = self.bias - (derror_dbias * self.learning_rate)
        self.weights = self.weights - (
            derror_dweights * self.learning_rate
        )

    def train(self, input_vectors, targets, iterations):

        cumulative_errors = []

        for current_iteration in range(iterations):

            # Pick a data instance at random

            random_data_index = np.random.randint(len(input_vectors))


            input_vector = input_vectors[random_data_index]

            target = targets[random_data_index]


            # Compute the gradients and update the weights

            derror_dbias, derror_dweights = self._compute_gradients(

                input_vector, target

            )


            self._update_parameters(derror_dbias, derror_dweights)


            # Measure the cumulative error for all the instances
            if current_iteration % 100 == 0:

                cumulative_error = 0

                # Loop through all the instances to measure the error

                for data_instance_index in range(len(input_vectors)):

                    data_point = input_vectors[data_instance_index]

                    target = targets[data_instance_index]


                    prediction = self.predict(data_point)

                    error = np.square(prediction - target)


                    cumulative_error = cumulative_error + error

                cumulative_errors.append(cumulative_error)


        return cumulative_errors
```

[66]:
```
input_vectors = np.array(
            [
                [3, 1.5],
                [2, 1],
                 [4, 1.5],
                 [3, 4],
                 [3.5, 0.5],
                 [2, 0.5],
                 [5.5, 1],
                 [1, 1],
            ]
)

targets = np.array([0, 1, 0, 1, 0, 1, 1, 0])
learning_rate = 0.1
neural_network = NeuralNetwork(learning_rate)
training_error = neural_network.train(input_vectors, targets, 10000)

plt.plot(training_error)
plt.xlabel("Iterations")
plt.ylabel("Error for all training instances")
plt.savefig("cumulative_error.png")
```



## 2.13 Linear Algebra Basics

### 2.13.1 Image Processing

- Single Value Decomposition (SVD)
  - $U\Sigma V^T = A$ where U and $V^T$ are square and $\Sigma$ is the same size as A
    - $\Sigma$ is a diagonal matrix and contains the singular values of A
  - According to colorimetry, it is possible to obtain a fairly reasonable grayscale version of our color image if we apply the formula
  - $Y = 0.2126R + 0.7152G + 0.0722B$

## 2.14 Read an JPEG image into a numpy array

```
img = imread('assets/cat.jpg')
print(img.dtype, img.shape)  # Prints "uint8 (400, 248, 3)"

# We can tint the image by scaling each of the color channels
# by a different scalar constant. The image has shape (400, 248, 3);
# we multiply it by the array [1, 0.95, 0.9] of shape (3,);
# numpy broadcasting means that this leaves the red channel unchanged,
# and multiplies the green and blue channels by 0.95 and 0.9
# respectively.
img_tinted = img * [1, 0.95, 0.9]

# Resize the tinted image to be 300 by 300 pixels.
img_tinted = imresize(img_tinted, (300, 300))
```

```
# Write the tinted image back to disk
imsave('assets/cat_tinted.jpg', img_tinted)
```

[67]:
```
# !pip3 install imageio
import os
import imageio

DIR = "/home/parzival/Desktop/TakingControl/Books/NumericalAndSymbolicGenerics/"
redPanda = imageio.imread(os.path.join(DIR, "RedPanda.png"))

print(redPanda.shape)
print(redPanda.dtype)
print(type(redPanda))

redPandaArr = np.array(redPanda)
print(type(redPandaArr))
```

```
/tmp/ipykernel_10765/1496481713.py:6: DeprecationWarning: Starting with ImageIO
v3 the behavior of this function will switch to that of iio.v3.imread. To keep
the current behavior (and make this warning dissapear) use `import imageio.v2 as
imageio` or call `imageio.v2.imread` directly.
  redPanda = imageio.imread(os.path.join(DIR, "RedPanda.png"))
```

```
(4000, 6000, 3)
uint8
<class 'imageio.core.util.Array'>
<class 'numpy.ndarray'>
```

[68]:
```
%matplotlib inline
plt.imshow(redPandaArr)
plt.show()
```

[69]:
```
# Get real numbers between 0-1 in each entry to represent the RGB values
redPandaArr = redPandaArr / 255

print(redPandaArr.max(), redPandaArr.min())
```

```
1.0 0.0
```

[70]:
```
# Could separate color channels
red_array = redPandaArr[:, :, 0]
green_array = redPandaArr[:, :, 1]
blue_array = redPandaArr[:, :, 2]
```

[71]:
```
#  Notice we can use the @ operator (the matrix multiplication operator for␣
 ↪NumPy arrays
from numpy import linalg

img_gray = redPandaArr @ [0.2126, 0.7152, 0.0722]

print(img_gray.shape)
plt.imshow(img_gray, cmap="gray")
plt.show()
```

```
(4000, 6000)
```

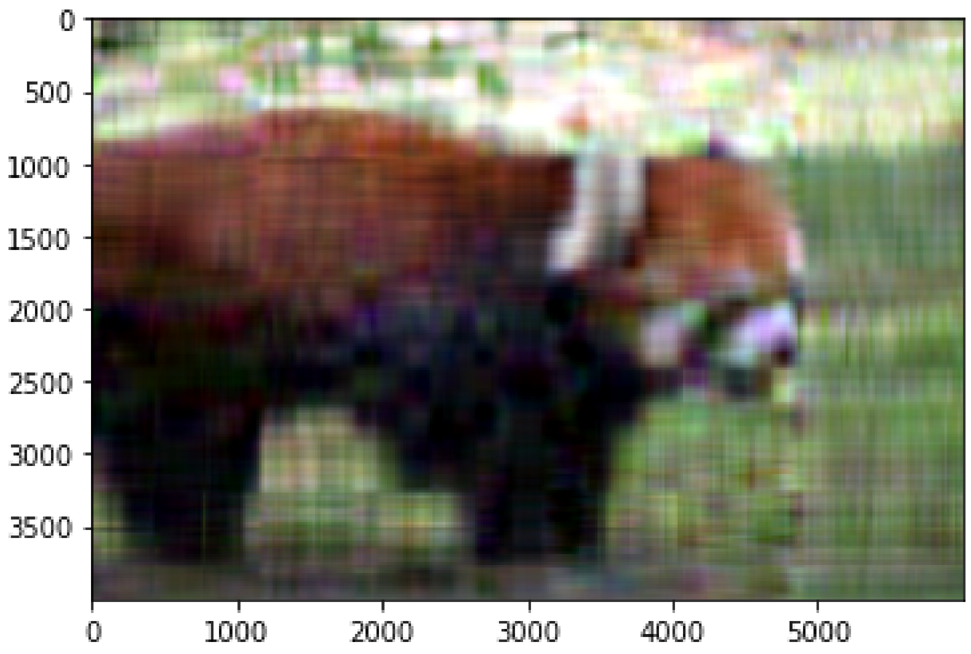
[72]:
```
u, s, vT = linalg.svd(img_gray)

print(u.shape, s.shape, vT.shape)
```

```
(4000, 4000) (4000,) (6000, 6000)
```

[73]:
```
Sigma = np.zeros((u.shape[1], vT.shape[0]))
np.fill_diagonal(Sigma, s)
```

[76]:
```
# Apply so machine learning can go faster with all colors
redPandaArr_T = np.transpose(redPandaArr, (2,0,1))
U, S, Vt = linalg.svd(redPandaArr_T)

Sigma = np.zeros((3, 4000, 6000))
for j in range(3):
    np.fill_diagonal(Sigma[j, :, :], S[j, :])


# Rebuild full SVD with no approximation
# reconstructed = U @ Sigma @ Vt
k = 10

approx_img = U @ Sigma[..., :k] @ Vt[..., :k, :]
plt.imshow(np.transpose(approx_img, (1, 2, 0)))
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



### 2.14.1 Solving Systems of Equations

- Example
  - $3x_1 + 2x_2 = 12$
  - $2x_1 - 1x_2 = 1$

[77]:
```
from scipy.linalg import solve

A = np.array(
    [
            [3, 2],
            [2, -1],
    ]
)

b = np.array([12, 1]).reshape((2, 1))
x = solve(A, b)

print(x)
```

```
[[2.]
 [3.]]
```

## 2.15 Statistics And Practical Machine Learning

- ME 539 Introduction to Scientific Machine Learning
- https://predictivesciencelab.github.io/data-analytics-se/index.html