Part 1: Implementing a CSR matrix

```
from scipy.sparse.linalg import LinearOperator
from scipy import sparse
import math
import numpy as np
import matplotlib.pylab as plt
from scipy.sparse import coo_matrix, linalg
from timeit import timeit

import warnings
warnings.filterwarnings("ignore")
```

```
class CSRMatrix(LinearOperator):
    def __init__(self, *x):

        if len(x) == 1:
            coo_matrix = x[0]
            self.shape = coo_matrix.shape
            self.dtype = coo_matrix.dtype

            # indices
            data = []
            for i in range(len(coo_matrix.row)):   # sort value according to index
                data.append([coo_matrix.row[i], coo_matrix.col[i], coo_matrix.data[i]])
            data = sorted(data, key=lambda x: (x[0], x[1]))
            self.col = [x[1] for x in data]

            self.data = [x[-1] for x in data]

            # indptr
            self.row = [0] * (self.shape[0]+1)
            r = coo_matrix.row

            for i in r:
                self.row[i+1] += 1   # elem num for each row
            for i in range(len(self.row)-1):
                self.row[i+1] = self.row[i+1] + self.row[i]   # indptr

        elif len(x) == 5:
            self.shape = x[0]
            self.data = x[1]
            self.col = x[2]
            self.row = x[3]
            self.dtype = x[4]


    def __add__(self, other):
        """Add the CSR matrix other to this matrix."""
        if other.shape != self.shape:
            raise Exception('Shape should be equal')
```

```python
        t_col = self.col.copy()
        t_data = self.data.copy()
        t_row = self.row.copy()

        for i in range(1, len(other.row)):
            num = other.row[i] - other.row[i-1]

            if num != 0:   # if exists elem in this row
                row = i - 1   # row index
                for j in range(num):
                    col = other.col[other.row[row]+j]

                    # if there is a elem in specific index
                    if (t_row[i] - t_row[i-1]) != 0:
                        if col in t_col[t_row[i-1]:t_row[i]]:
                            index = t_col[t_row[i-1]:t_row[i]].index(col)   # find col

                            # update elem value in corresponding index
                            t_data[t_row[i-1]+index] += other.data[other.row[i-1]+j]
                        else:   # no elem in specific index
                            col_list = t_col[t_row[i-1]:t_row[i]]
                            col_list.append(col)
                            col_list = sorted(col_list)
                            index = col_list.index(col)   # find col index

                            # insert data
                            t_data.insert(t_row[i-1]+index, other.data[other.row[i-1]+j])
                            t_col.insert(t_row[i-1]+index, col)   # insert col

                            # update indptr
                            for k in range(i, len(t_row)):
                                t_row[k] += 1

        return CSRMatrix(self.shape, t_data, t_col, t_row, self.dtype)


    def _matvec(self, vector):
        """Compute a matrix-vector product."""
        if self.shape[1] != len(vector):
            raise Exception('Shape does not match')

        result = np.zeros((len(vector), 1)).reshape((-1,1))

        for i in range(1, len(self.row)):   # calculate by row
            if (self.row[i] - self.row[i-1]) != 0:   # exist elem in this row
                value = 0.0
                for j in range(self.row[i] - self.row[i-1]):
                    col = self.col[self.row[i-1]+j]   # col index
                    value += self.data[self.row[i-1]+j] * vector[col]
            result[i-1] = value

        return result
```

```
    def toArray(self):
        """Print matrix."""
        matrix = np.zeros(self.shape)  # matrix

        for i in range(1, len(self.row)):  # print elem by row
            for j in range(self.row[i] - self.row[i-1]):  # exists elem
                col = self.col[self.row[i-1]+j]  # col index
                matrix[i-1][col] += self.data[self.row[i-1]+j]  # use plus

        return matrix
```

- Write tests to check that the _*add*_ and matvec methods are correct.

- Firstly, we test the _add_ method.

```
# test case 1
add_a_m = coo_matrix(([0.5, 0.7], ([0, 1], [1, 0])), (2, 2))
add_a_1 = CSRMatrix(add_a_m)
add_a_1.toArray()
```

```
array([[0. , 0.5],
       [0.7, 0. ]])
```

```
add_b_m = coo_matrix(([0.5, 0.3], ([0, 1], [1, 0])), (2, 2))
add_b_1 = CSRMatrix(add_b_m)
add_b_1.toArray()
```

```
array([[0. , 0.5],
       [0.3, 0. ]])
```

```
sum_1 = add_a_1 + add_b_1
assert (sum_1.toArray() == (add_a_m+add_b_m).todense()).all()
sum_1.toArray()
```

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

```
# test case 2
add_a_m = coo_matrix(([0.5, 0.7], ([0, 1], [1, 0])), (2, 2))
add_a_2 = CSRMatrix(add_a_m)
add_a_2.toArray()
```

```
array([[0. , 0.5],
       [0.7, 0. ]])
```

```
add_b_m = coo_matrix(([0.5, 0.5], ([0, 1], [0, 1])), (2, 2))
add_b_2 = CSRMatrix(add_b_m)
add_b_2.toArray()
```

```
array([[0.5, 0. ],
```

```
       [0. , 0.5]])
```

```
sum_2 = add_a_2 + add_b_2
assert (sum_2.toArray() == (add_a_m+add_b_m).todense()).all()
sum_2.toArray()
```

```
array([[0.5, 0.5],
       [0.7, 0.5]])
```

```
# test case 3
add_a_m = sparse.coo_matrix(np.random.rand(5, 5))
add_a_3 = CSRMatrix(add_a_m)
```

```
add_b_m = sparse.coo_matrix(np.random.rand(5, 5))
add_b_3 = CSRMatrix(add_b_m)
```

```
sum_3 = add_a_3 + add_b_3
assert (sum_3.toArray() == (add_a_m+add_b_m).todense()).all()
```

```
# test case 4
add_a_m = sparse.coo_matrix(np.random.rand(10, 10))
add_a_4 = CSRMatrix(add_a_m)
```

```
add_b_m = sparse.coo_matrix(np.random.rand(10, 10))
add_b_4 = CSRMatrix(add_b_m)
```

```
sum_4 = add_a_4 + add_b_4
assert (sum_4.toArray() == (add_a_m+add_b_m).todense()).all()
```

- Next, we test the matvec method.

```
# test case 5
mul_a_m = sparse.coo_matrix(np.random.rand(3, 3))
mul_a_1 = CSRMatrix(mul_a_m)
mul_a_1.toArray()
```

```
array([[0.82317261, 0.18629444, 0.45500127],
       [0.13188246, 0.06912278, 0.55542189],
       [0.95769244, 0.71466717, 0.96253558]])
```

```
mul_b_1 = np.array([0., 0., 1.]).reshape((-1,1))
mul = mul_a_1.matvec(mul_b_1)
assert (mul == (mul_a_m@mul_b_1)).all()
mul
```

```
array([[0.45500127],
       [0.55542189],
       [0.96253558]])
```

```
# test case 6
mul_a_m = sparse.coo_matrix(np.random.rand(5, 5))
mul_a_2 = CSRMatrix(mul_a_m)
mul_a_2.toArray()
```

```
array([[0.32148504, 0.89027672, 0.30359651, 0.84187781, 0.33246901],
       [0.54644807, 0.2526251 , 0.86093601, 0.73584394, 0.72974115],
       [0.2834166 , 0.98140358, 0.9284662 , 0.34954128, 0.21058332],
       [0.23790967, 0.59060641, 0.09131423, 0.71428181, 0.93353238],
       [0.59698613, 0.33419003, 0.55077259, 0.54276834, 0.14364355]])
```

```
mul_b_2 = np.random.randn(5).reshape((-1,1))
mul_b_2
```

```
array([[ 0.3439128 ],
       [ 0.27294374],
       [ 0.01647143],
       [-0.8442373 ],
       [ 0.72967059]])
```

```
mul = mul_a_2.matvec(mul_b_2)
assert (mul == (mul_a_m@mul_b_2)).all()
mul
```

```
array([[-0.10959284],
       [ 0.18230752],
       [ 0.23919239],
       [ 0.32267436],
       [-0.04781454]])
```

```
# test case 6
mul_a_m = sparse.coo_matrix(np.random.rand(10, 10))
mul_a_3 = CSRMatrix(mul_a_m)

mul_b_3 = np.random.randn(10).reshape((-1,1))
mul = mul_a_3.matvec(mul_b_3)
assert (mul == (mul_a_m@mul_b_3)).all()
```

- Measure and compare the time taken to perform a matvec product.

```
def numpy_matvec(mat, vec):
        mat @ vec
```

```
def CSR_matvec(scr_mat, vec):
        scr_mat.matvec(vec)
```

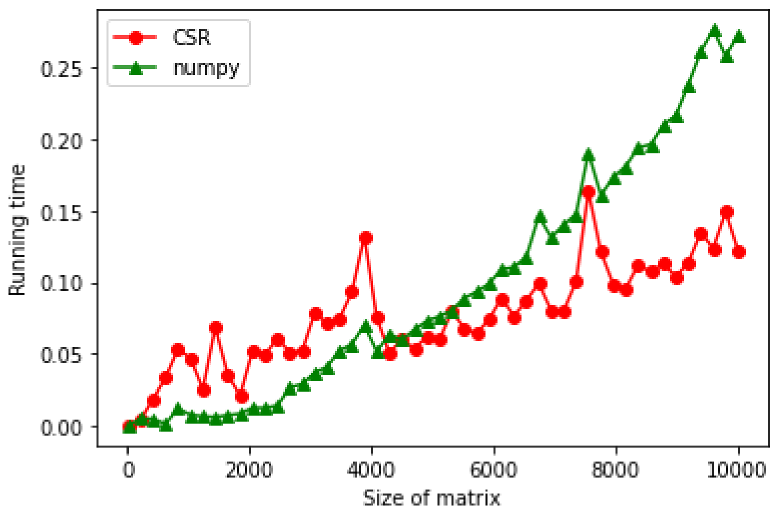
```
def generate_matrix(N):
    h = 1 / N
    k = 29 * math.pi / 2

    f = np.zeros((N + 1), dtype=np.float64)
```

```
    f[N] = 1.0

    row = [0, N]
    col = [0, N]
    data = [1, 1]

    for i in range(1, N):
        row += [i, i, i]
        col += [i, i + 1, i - 1]
        data += [2 - (h ** 2) * (k ** 2), -1, -1]

    row = np.array(row)
    col = np.array(col)
    data = np.array(data)

    return coo_matrix((data, (row, col)), (N+1, N+1))
```

```
x = np.linspace(10, 10000, 50, dtype=np.int32)
y_csr = []
y_numpy = []

for n in x:
        mat = generate_matrix(n-1)
        csr_mat = CSRMatrix(mat)

        vec = np.random.randn(n).reshape((-1,1))

        y_csr.append(timeit(lambda: CSR_matvec(csr_mat, vec), number=1))
        y_numpy.append(timeit(lambda: numpy_matvec(mat.todense(), vec), number=1))
```

```
plt.plot(x, y_csr, "ro-")
plt.plot(x, y_numpy, "g^-")

plt.xlabel("Size of matrix")
plt.ylabel("Running time")
plt.legend(["CSR", "numpy"])
```

<matplotlib.legend.Legend at 0x7fcfbfe51890>



According to the graph above, before the matrix size reach to 4000, the performance of numpy is better than CSR matrix. However, since the use of sparse storage, the number of operations in CSR is considerably small, so CSR performs much better than numpy as the matrix size continues to increase.

- Use Scipy's sparse solvers to solve matrix problems.

```
A = sparse.coo_matrix(np.random.rand(10, 10))
A = CSRMatrix(A)

b = np.random.randn(10).reshape((-1, 1))
```

```
# gmres
sol_g = linalg.gmres(A, b)
sol_g
```

```
(array([-13.45373395,   0.83564032,  -3.58518349,  26.09005547,
         -5.56023691,   7.17852306,   5.32947734, -22.44786869,
        -15.95367945,   8.90171857]), 0)
```

```
# cg
sol_cg = linalg.cg(A, b)
sol_cg
```

```
(array([ 2.40954391e+10, -6.70583133e+10, -1.05871919e+10,  4.36161537e+10,
         7.47084685e+09,  2.99183840e+10, -2.81039919e+10,  2.97345325e+10,
         7.39977728e+09, -4.09505095e+10]), 100)
```

```
assert (sol_g[0] == sol_cg[0]).all()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-205-d5542c9cc40a> in <module>
----> 1 assert (sol_g[0] == sol_cg[0]).all()

AssertionError:
```

SEARCH STACK OVERFLOW

The solutions are not the same. Refer to the return value, for gmres, the 'info' parameter of return value is 0, which means that algorithm successful exit, so gmres method have found the converged solution. However, the 'info' parameter of return value is larger than 0 in cg method, which means that convergence to tolerance not achieved, so this return array is not the converged solution.

Part 2: Implementing a custom matrix

```
class Custom_Matrix(LinearOperator):
    def __init__(self, n):
        if n <= 0:
            raise Exception('Shape should be positive')

        self.dtype = np.dtype('float64')

        self.shape = (2*n, 2*n)
        self.diagonal = np.random.randn(n)
        self.mat_T = np.random.rand(n, 2)
        self.mat_W = np.random.randn(2, n)

        self.bottom_right = self.mat_T @ self.mat_W  # bottom right matrix


    def matvec(self, vector):
        """Compute a matrix-vector product."""
        if self.shape[1] != len(vector):
            raise Exception('Shape does not match')

        result = np.zeros((len(vector), 1)).reshape((-1,1))

        n = self.shape[0] // 2

        for i in range(n):
            result[i] = self.diagonal[i] * vector[i]  # calculate top n row
            # calculate bottom n row
            result[n+i] = np.sum(self.bottom_right[i] * vector[n:].T)

        return result


    def toArray(self):
        """Print matrix."""
        n = self.shape[0] // 2
        matrix = np.zeros(self.shape)  # matrix

        for i in range(n):
            matrix[i][i] = self.diagonal[i]  # top left

            for j in range(n):
                matrix[n+i][n+j] = self.bottom_right[i][j]  # bottom right

        return matrix
```

- Compute matrix-vector products using custom matrix.

```
# test case 1
a_1 = Custom_Matrix(2)
a_1.toArray()
```

```
array([[-1.4645552 ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.22273345,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.01430358, -0.32794254],
       [ 0.        ,  0.        , -0.67508912, -0.95601114]])
```

```
b_1 = np.random.randn(4).reshape((-1,1))
b_1
```

```
array([[-1.05332248],
       [ 1.43082868],
       [ 0.44856943],
       [-0.69423593]])
```

```
mul = a_1.matvec(b_1)
assert (mul == (a_1.toArray()@b_1)).all()
mul
```

```
array([[1.54264891],
       [0.31869341],
       [0.23408564],
       [0.36087293]])
```

```
# test case 2
a_2 = Custom_Matrix(5)
b_2 = np.random.randn(10).reshape((-1,1))

mul = a_2.matvec(b_2)
assert (np.around(mul, 10) == (np.around(a_2.toArray()@b_2, 10))).all()
```

```
# test case 3
a_3 = Custom_Matrix(10)
b_3 = np.random.randn(20).reshape((-1,1))

mul = a_3.matvec(b_3)
assert (np.around(mul, 10) == (np.around(a_3.toArray()@b_3, 10))).all()
```

- Measure and compare the time taken.

```
def custom_matvec(custom_mat, vec):
    custom_mat.matvec(vec)
```

```
def numpy_matvec(np_mat, vec):
    np_mat @ vec
```

```
def coo_matvec(coo_mat, vec):
    coo_mat._mul_vector(vec)
```

```
test_matrix = Custom_Matrix(100)
test_b = np.random.randn(200).reshape((-1,1))
```

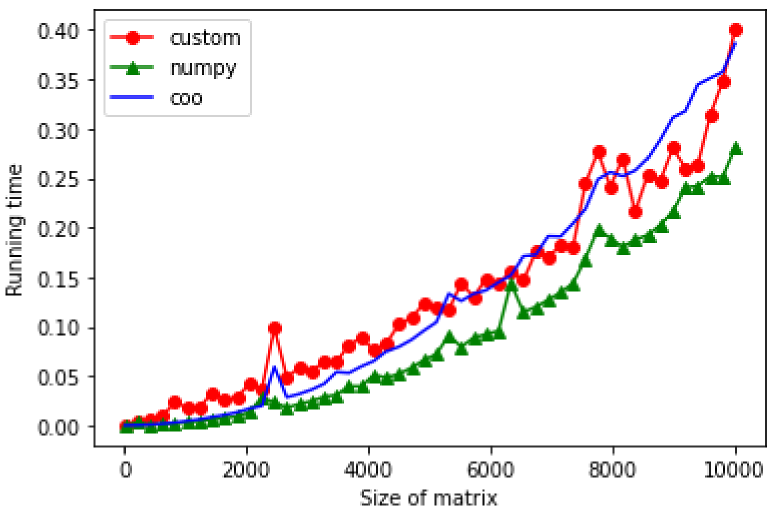
```
timeit(lambda: custom_matvec(test_matrix, test_b), number=1)
```

0.0020952530030626804

```
x = np.linspace(10, 1000, 50, dtype=np.int32)
y_custom = []
y_numpy = []
y_coo = []

for n in x:
    custom_mat = Custom_Matrix(n)
    np_mat = custom_mat.toArray()
    coo_mat = coo_matrix(np_mat)

    vec = np.random.randn(2 * n).reshape((-1,1))

    y_custom.append(timeit(lambda: custom_matvec(custom_mat, vec), number=1))
    y_numpy.append(timeit(lambda: numpy_matvec(np_mat, vec), number=1))
    y_coo.append(timeit(lambda: coo_matvec(coo_mat, vec), number=1))
```

```
plt.plot(x, y_custom, "ro-")
plt.plot(x, y_numpy, "g^-")
plt.plot(x, y_coo, "b")

plt.xlabel("Size of matrix")
plt.ylabel("Running time")
plt.legend(["custom", "numpy", "coo"])
```

<matplotlib.legend.Legend at 0x7fcfc10d6fd0>



According to the graph above, the method of coo format is the fastest.
Since compare to the method I implemented, the method of coo format is highly optimised, and the time-critical loops are usually implemented in C, C++, or Fortran, so coo format is faster.
Moreover, compare to the Numpy format, coo format uses sparse storage, which makes it take fewer operations. Therefore, the method of coo format is the fastest.