The assignment

Part 1: Implementing a CSR matrix

Scipy allows you to define your own objects that can be used with their sparse solvers. You can do this by creating a subclass of `scipy.sparse.LinearOperator`. In the first part of this assignment, you are going to implement your own CSR matrix format.

The following code snippet shows how you can define your own matrix-like operator.

```
from scipy.sparse.linalg import LinearOperator


class CSRMatrix(LinearOperator):
    def __init__(self, coo_matrix):
        self.shape = coo_matrix.shape
        self.dtype = coo_matrix.dtype
        # You'll need to put more code here

    def __add__(self, other):
        """Add the CSR matrix other to this matrix."""
        pass

    def _matvec(self, vector):
        """Compute a matrix-vector product."""
        pass
```

Make a copy of this code snippet and **implement the methods `__init__`, `__add__` and `matvec`.** The method `__init__` takes a COO matrix as input and will initialise the CSR matrix: it currently includes one line that will store the shape of the input matrix. You should add code here that extracts important data from a Scipy COO and computes and stores the appropriate data for a CSR matrix. You may use any functionality of Python and various libraries in your code, but you should not use a library's implementation of a CSR matrix. The method `__add__` will overload `+`, so you will be able to add two of your CSR matrices together. The `__add__` method should avoid converting any matrices to dense matrices. You could implement this in one of two ways: you could convert both matrices to COO matrices, compute the sum, then pass this into `CSRMatrix()`; or you could compute the data, indices and indptr for the sum, and use these to create a SciPy CSR matrix. The method `matvec` will define a matrix-vector product: Scipy will use this when you tell it to use a sparse solver on your operator.

Write tests to check that the `__add__` and `matvec` methods that you have written are correct. These tests should use appropriate `assert` statements.

For a collection of sparse matrices of your choice and a random vector, **measure the time taken to perform a `matvec` product**. Convert the same matrices to dense matrices and **measure the time to compute a dense matrix-vector product using Numpy**. **Create a plot showing the times of `matvec` and Numpy for a range of matrix sizes** and **briefly (1-2 sentence) comment on what your plot shows**.

For a matrix of your choice and a random vector, **use Scipy's `gmres` and `cg` sparse solvers to solve a matrix problem using your CSR matrix**. Check if the two solutions obtained are the same. **Briefly comment (1-2 sentences) on why the solutions are or are not the same (or are nearly but not exactly the same).**

Part 2: Implementing a custom matrix

Let $\mathbf{A}$ by a $2n$ by $2n$ matrix with the following structure:

- The top left n by n block of $\mathbf{A}$ is a diagonal matrix
- The top right n by n block of $\mathbf{A}$ is zero
- The bottom left n by n block of $\mathbf{A}$ is zero
- The bottom right n by n block of $\mathbf{A}$ is dense (but has a special structure defined below)

In other words, $\mathbf{A}$ looks like this, where $*$ represents a non-zero value

$$\mathbf{A} = \begin{pmatrix} * & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & * & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & * & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & * & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & * & * & \cdots & * \\ 0 & 0 & 0 & \cdots & 0 & * & * & \cdots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & * & * & \cdots & * \end{pmatrix}$$

Let $\tilde{\mathbf{A}}$ be the bottom right n by n block of $\mathbf{A}$. Suppose that $\tilde{\mathbf{A}}$ is a matrix that can be written as

$$\tilde{\mathbf{A}} = \mathbf{TW},$$

where $\mathbf{T}$ is a n by 2 matrix (a tall matrix); and where $\mathbf{W}$ is a 2 by n matrix (a wide matrix).

Implement a Scipy `LinearOperator` for matrices of this form. Your implementation must include a matrix-vector product (`matvec`) and the shape of the matrix (`self.shape`), but does not need to include an `__add__` function. In your implementation of `matvec`, you should be careful to ensure that the product does not have more computational complexity then necessary.

For a range of values of n, **create matrices where the entries on the diagonal of the top-left block and in the matrices T and W are random numbers**. For each of these matrices, **compute matrix-vector products using your implementation and measure the time taken to compute these**. Create an alternative version of each matrix, stored using a Scipy or Numpy format of your choice, and **measure the time taken to compute matrix-vector products using this format**. **Make a plot showing time taken against n**. **Comment (2-4 sentences) on what your plot shows, and why you think one of these methods is faster than the other** (or why they take the same amount of time if this is the case).

Part 1

In [19]:
```
from scipy.sparse.linalg import LinearOperator
from scipy.sparse import coo_matrix


class CSRMatrix(LinearOperator):
    def __init__(self, coo_matrix):
        self.shape = coo_matrix.shape
        self.dtype = coo_matrix.dtype
        n=self.shape[0] +1

        row, col, data = coo_matrix.row, coo_matrix.col, coo_matrix.data

        sort = np.lexsort((col, row))
        row = row[sort]
        col = col[sort]
        data = data[sort]

        indptr = np.zeros(n, dtype=int)
        np.add.at(indptr, row + 1, 1)
        np.cumsum(indptr, out=indptr)

        self.data = data
        self.indices = col
        self.indptr = indptr

    def __add__(self, other):
        n=self.shape

        first_coo = coo_matrix((self.data, (self.indptr[:-1].repeat(np.diff(self.indptr)), self.indices)), n)
        second_coo = coo_matrix((other.data, (other.indptr[:-1].repeat(np.diff(other.indptr)), other.indices)), n)

        Add_two_coo = first_coo + second_coo

        return CSRMatrix(Add_two_coo.tocoo())

    def _matvec(self, vector):
        result= np.add.reduceat(self.data * vector[self.indices], self.indptr[:-1])
        return result
```

Testing **add** and _matvec

In [21]:
```
#Write tests to check that the __add__ and matvec methods that you have written are correct. These tests should use appropriate assert statements.
from scipy.sparse import coo_matrix
import numpy as np

data1, data2 = [1, 2, 3], [4, 5, 6]
row1, row2 = [0, 1, 2], [0, 1, 2]
col1, col2 = [0, 1, 2], [0, 1, 2]

# create coo matrices
coo1 = coo_matrix((data1, (row1, col1)), shape=(3, 3))
coo2 = coo_matrix((data2, (row2, col2)), shape=(3, 3))

# initialize csr matrix
csr1 = CSRMatrix(coo1)
csr2 = CSRMatrix(coo2)

# perform add
sum_two_csr = csr1.__add__(csr2)

# test the add function
assert np.array_equal(sum_two_csr.data, [5, 7, 9]), f"Data isn't correct: {sum_two_csr.data}"
assert np.array_equal(sum_two_csr.indices, [0, 1, 2]), f"Indices aren't correct: {sum_two_csr.indices}"
assert np.array_equal(sum_two_csr.indptr, [0, 1, 2, 3]), f"Indptr isn't correct: {sum_two_csr.indptr}"
assert np.array_equal(sum_two_csr.data, (coo1+coo2).data ), f"Data isn't correct: {sum_two_csr.data}"
print(" Add method works! ")


# define a vector
vector = np.array([1, 2, 3], dtype=np.float64)

# perform multiplication
result= csr1._matvec(vector)

# test the result of the product
assert np.array_equal(result, [1, 4, 9]), f"matvec product isn't correct: {result}"
assert np.array_equal(result, coo1 @ vector), f"matvec product isn't correct: {result}"
print(" Matvec method works fine! " )
```

```
Add method works!
Matvec method works fine!
```

In [113…
```
# test the time for small matrix, csr takes longer!
print("csr")
%timeit -o  csr1 @ vector

Dense=coo1.toarray()
```

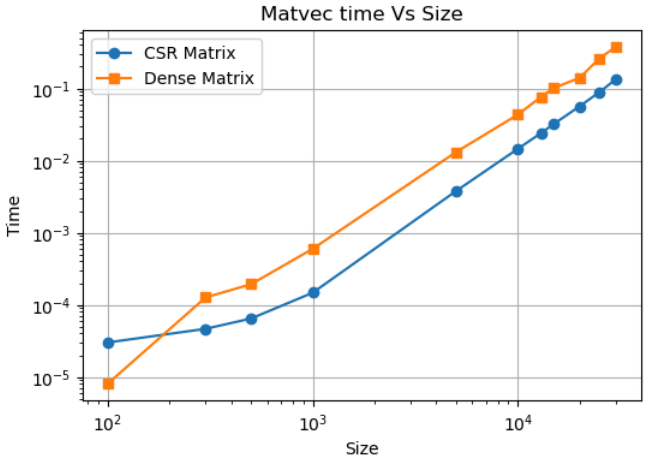
```
print("dense")
%timeit -o  Dense @ vector
```

```
csr
30.3 µs ± 908 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
dense
7.69 µs ± 69.2 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Out[113… <TimeitResult : 7.69 µs ± 69.2 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)>

In [23]:
```
import matplotlib.pyplot as plt
import numpy as np
sizes = [100, 300, 500, 1000, 5000, 10000, 13000, 15000, 20000, 25000, 30000]

csr_times = []
dense_times = []

for size in sizes:
    coo_m = coo_matrix((np.random.rand(int(0.01 * size**2)),
                        (np.random.randint(0, size, int(0.01 * size**2)),
                         np.random.randint(0, size, int(0.01 * size**2)))),
                       shape=(size, size))

    csr_m = CSRMatrix(coo_m)
    vector = np.random.rand(size)

    s_time =%timeit -o  csr_m @ vector
    csr_times.append(s_time.average)

    dense_matrix = coo_m.toarray()

    d_time =%timeit -o  dense_matrix @ vector
    dense_times.append(d_time.average)

plt.figure(figsize=(6, 4))
plt.plot(sizes, csr_times, label='CSR Matrix', marker='o')
plt.plot(sizes, dense_times, label='Dense Matrix', marker='s')
plt.title('Matvec time Vs Size')
plt.yscale("log")
plt.xscale("log")
plt.xlabel('Size')
plt.ylabel('Time')
plt.legend()
plt.grid(True)
plt.show()
```

```
30.1 µs ± 137 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
8.1 µs ± 1.01 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
46.5 µs ± 241 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
127 µs ± 7.51 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
64.7 µs ± 5.97 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
193 µs ± 26.9 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
147 µs ± 4.17 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
599 µs ± 39.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
3.78 ms ± 69 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
13.1 ms ± 1.67 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
14.5 ms ± 149 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
43.2 ms ± 5.03 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
24.1 ms ± 350 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
76.4 ms ± 1.75 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
32.2 ms ± 681 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
102 ms ± 2.01 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
56.3 ms ± 801 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
139 ms ± 5.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
88.6 ms ± 1.79 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
260 ms ± 10.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
133 ms ± 1.81 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
374 ms ± 9.68 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```



Answer:

The plot shows that the CSR matrix takes longer when the matrix sizes are small, and the reason that the dense matrix is faster for small sizes is because of its lower overhead. However, CSR performs faster than the dense matrix when the sizes are huge as it's leveraging sparsity.

In [35]:
```
import numpy as np
from scipy.sparse import coo_matrix
from scipy.sparse.linalg import gmres, cg

data = np.array([1, 2, 3], dtype=np.float64)
row = [0, 1, 2]
col = [0, 1, 2]
coo = coo_matrix((data, (row, col)), shape=(3, 3))
csr = CSRMatrix(coo)

vector = np.random.rand(csr.shape[0])

gmres_result, _ = gmres(csr, vector)

cg_result, _ = cg(csr, vector)

np.allclose(gmres_result, cg_result)
print("Both of them are the same!")
print(f"GMRES:\n{gmres_result}")
print(f"CG:\n{cg_result}")
```

```
Both of them are the same!
GMRES:
[0.45205805 0.4488017  0.18977781]
CG:
[0.45205805 0.4488017  0.18977781]
```

Answer:

The solutions are the same when the matrix is SPD as CG requires this property to ensure convergence, while GMRES is more general and can handle a wider range of systems. Initially, I've used random values with larger sizes in the CSR matrix but due to time-consuming issues I've customize my own matrix. However, that has led to different result, I got different results for GMRES and CG. This is because CG's behavior depends on the matrix's properties.

Part 2

In [26]:
```
import numpy as np
# create a matrix A with the specifed properties.
def matrix_A(n):

    # top left is diagonal.
    top_left = np.diag(np.random.rand(n))
    # top right, bottem left, are zeros
    top_right = np.zeros((n, n))
    bottom_left = np.zeros((n, n))

    # A hat = T*W where T is a tall matrix and W is wide matrix
    T = np.random.rand(n, 2)
    W = np.random.rand(2, n)

    # dense matrix
    bottom_right = np.dot(T, W)

    # combine the matrices
    top = np.hstack((top_left, top_right))
    bottom = np.hstack((bottom_left, bottom_right))
    return np.vstack((top, bottom))

# test
n = 6
A = matrix_A(n)
for row in A:
    print(" ".join(map(str, row)))
```

```
0.006844403920585718 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.08601331366312892 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.7041257935673845 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.7851752447184186 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.3329683786691481 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.7999130654816548 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.49279644860006727 0.3506417082005484 0.11626920575796494 0.8139341295138907 0.4930930673319341 0.7584155118289008
0.0 0.0 0.0 0.0 0.0 0.0 0.10447528114127039 0.11172070916069347 0.03277746959036377 0.23332990706571416 0.13286503766670665 0.24710006841438062
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.2810441524593523 0.08636902726473011 0.04160910558435011 0.2795099394123675 0.1951303292687518 0.17023232944314334
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.4262686698681754 0.23007865335185682 0.08465190204781106 0.5850118034933418 0.37103815359555997 0.4869593531326673
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5265566075230275 0.5081771856789349 0.15326319348591974 1.0867425422641328 0.6280435794433686 1.118638519315068
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5332687510815567 0.5080396239296967 0.15377854736950713 1.0898411055007444 0.6310365768685198 1.1176239504531331
```

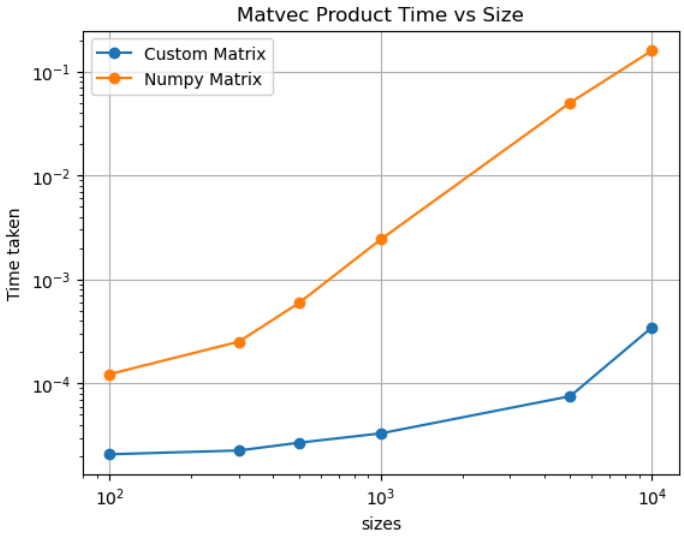
In [30]:
```
import numpy as np
from scipy.sparse.linalg import LinearOperator

class A_matrix(LinearOperator):
    def __init__(self, diag_values, T_m, W_m):

        self.n = len(diag_values)
```

```
        self.diag_values = diag_values
        self.T_m = T_m
        self.W_m = W_m
        self.shape = (2 * n, 2 * n)
        super().__init__(dtype=np.float64, shape=self.shape)

    def _matvec(self, vector):
        n = self.n
        # first split the vector into two parts
        x_top = vector[:n]
        x_bottom = vector[n:]

        # calculate the top values
        top = self.diag_values * x_top

        # calculate the bottom values
        W_x_bottom = self.W_m @ x_bottom
        bottom = self.T_m @ W_x_bottom

        return np.concatenate((top, bottom))
```

In [32]:

```
# various n values
sizes = [100, 300, 500, 1000, 5000, 10000]
custom_times = []
numpy_times = []

for n in sizes:

    diagonal = np.random.rand(n) # random diagonal
    T = np.random.rand(n, 2) # random T
    W = np.random.rand(2, n) # ransom W

    # create Custom matrix
    A_custom_matrix = A_matrix(diagonal, T, W)

    # create alt version using numpy
    D = np.diag(diagonal)
    A_alt = np.block([
        [D, np.zeros((n, n))],
        [np.zeros((n, n)), T @ W]
    ])


    vector = np.random.rand(2 * n) # random vector
    s_time =%timeit -o  A_custom_matrix._matvec(vector)
    custom_times.append(s_time.average)

    d_time =%timeit -o  A_alt @ vector
    numpy_times.append(d_time.average)


plt.plot(sizes, custom_times, label='Custom Matrix', marker='o')
plt.plot(sizes, numpy_times, label='Numpy Matrix', marker='o')
plt.yscale("log")
plt.xscale("log")
plt.xlabel('sizes')
plt.ylabel('Time taken')
plt.title('Matvec Product Time vs Size')
plt.legend()
plt.grid(True)
plt.show()
```

```
20.9 µs ± 522 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
123 µs ± 5.9 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
22.7 µs ± 3.25 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
253 µs ± 49.9 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
27 µs ± 338 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
596 µs ± 62.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
33.1 µs ± 862 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
2.43 ms ± 192 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
75.5 µs ± 1.33 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
50.1 ms ± 1.64 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
346 µs ± 45.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
158 ms ± 3.98 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```



Answer:

The plot highlights the advantage of LinearOperator by showing that the custom matrix takes less time than the numpy matrix to compute as the matrix size grows. This demonstrates the fact that LinearOperator doesn't build a complete matrix; it uses the diagonal and dense (T and W) matrices to perform the product of the vector. While NumPy's computation doesn't have this feature, instead, it will store and multiply the whole matrix, which leads to increased computational time and memory usage.

In [ ]: