assingment3

December 4, 2023

0.1 Assignment 3 - SpMV

Student Number = *20121702*

[ ]:
```
import numpy as np
import scipy
from scipy.sparse.linalg import LinearOperator, gmres, cg, eigs
from scipy.sparse import diags
from scipy.sparse import coo_matrix as coo
import numba
import matplotlib.pyplot as plt
from scipy.linalg import block_diag
from scipy.sparse import csc_matrix, csr_matrix
from scipy.sparse import coo_matrix,csc_matrix, csr_matrix, dia_matrix,␣
↪dok_matrix, lil_matrix, diags
```

0.2 Part 1.1: Implement CSRMatrix

Implementing `__init__`, `__add__` and `matvec`

[37]:
```
class CSRMatrix(LinearOperator):

    def __init__(self, coo_matrix):
        """
        Initializes a CSR matrix from a given COO matrix.
        This method sorts the COO matrix (unless already sorted) and then␣
↪converts it to CSR format, which involves creating
        data, indices, and indptr arrays that represent the sparse matrix␣
↪efficiently.

        Args:
        coo_matrix: A sparse matrix in COO format.
        """
        # Check if the COO matrix data needs sorting (non-empty and unsorted)
        if len(coo_matrix.data) == 0 or np.all(coo_matrix.row[:-1] <=␣
↪coo_matrix.row[1:]):
            # If data is empty or already sorted, use it directly
            row = coo_matrix.row
            col = coo_matrix.col
            data = coo_matrix.data
```

```
        else:
            # If data is unsorted, sort by row and column order
            order = np.lexsort((coo_matrix.col, coo_matrix.row))
            row = coo_matrix.row[order]
            col = coo_matrix.col[order]
            data = coo_matrix.data[order]

        self.dtype = coo_matrix.dtype
        self.shape = coo_matrix.shape
        # Store the sorted data and column indices
        self.data = data
        self.indices = col
        # Initialize the index pointer array for rows
        self.indptr = np.zeros(coo_matrix.shape[0] + 1, dtype=coo_matrix.row.
↪dtype)

        # Populate the index pointer array
        for row_index in row:
            self.indptr[row_index + 1] += 1
        # Compute the cumulative sum to get the final index pointers
        self.indptr = np.cumsum(self.indptr)

    def __add__(self, other):
        """
        Adds two CSR matrices and returns the result as a new CSR matrix.

        Args:
        other: Another CSR matrix to add to this one.
        """
        assert self.shape == other.shape

        add_data, add_indices, add_indptr = [], [], []
        add_shape = self.shape

        t = 0  # Counter for total number of non-zero elements
        for i in range(self.shape[0]):
            # Append the current count to the index pointer array
            add_indptr.append(t)
            # Iterate over the non-zero elements of both matrices in row i
            self_r_start, self_r_end = self.indptr[i], self.indptr[i + 1]
            other_r_start, other_r_end = other.indptr[i], other.indptr[i + 1]
            # Append data and indices from self
            for j in range(self_r_start, self_r_end):
                add_data.append(self.data[j])
                add_indices.append(self.indices[j])
                t += 1
            # Append data and indices from other
```

```
            for m in range(other_r_start, other_r_end):
                add_data.append(other.data[m])
                add_indices.append(other.indices[m])
                t += 1

        # Finalize the index pointer array
        add_indptr.append(t)

        # Return the sum as a new CSR matrix
        return csr_matrix((add_data, add_indices, add_indptr), shape=add_shape)


    @staticmethod
    @numba.jit(nopython=True, parallel=True)
    def _wrapper_matvec(data, indices, indptr, shape, x):
        """
        Static method wrapper for matrix-vector multiplication.
        This method is optimized using Numba for JIT compilation, enabling
        faster execution, especially beneficial for large sparse matrices.

        Args:
        data, indices, indptr: CSR format arrays.
        shape: Shape of the matrix.
        x: The vector to be multiplied."""
        m = shape[0]
        y = np.zeros(m, dtype=np.float64)
        for row_index in numba.prange(m):
            col_start = indptr[row_index]
            col_end = indptr[row_index + 1]
            for col_index in range(col_start, col_end):
                y[row_index] += data[col_index] * x[indices[col_index]]
        return y

    def _matvec(self, vector):
        """
        Compute a matrix-vector product using the CSR format,
        using the wrapper function inspired by the lecture notes.

        Args:
        vector: The vector to be multiplied"""

        m = self.shape[0]
        if vector.shape != (m,):
            raise ValueError(f"Vector shape {vector.shape} invalid, must be␣
↪({m},)")
        return self._wrapper_matvec(self.data, self.indices, self.indptr, self.
↪shape, vector)
```

0.3 Part 1.2 Tests to check `add` and `matvec` methods

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

[38]:
```
def add_sparse(A,B):
    """
    Adds two matrices in sparse format.
    A created in a COO format, converted to CSR format
    B in CSR format
    Converts A to COO then to CSR format.
    The addition is performed in CSR format,
    and the result is converted back to a dense array.

    Args:
    A: First input matrix.
    B: Second input matrix, in CSR format.

    Returns:
    Sum of A and B as a dense array.
    """
    coo_A = coo_matrix(A)
    A1 = CSRMatrix(coo_A)
    B1 = csr_matrix(B)
    # The data is added in csr_format
    add = A1.__add__(B1)

    # Converting csr values to matrix
    return add.toarray()

def mv_sparse(A,v):
    """
    Performs matrix-vector multiplication using a sparse matrix.
    Utilizes _matvec method from CSRMatrix class.

    Args:
    A: Sparse matrix in CSR format.
    v: Vector for multiplication.

    Returns:
    Result of multiplication.
    """
    return A._matvec(v)

def mv_dense(A,v):
    """
    Performs matrix-vector multiplication with a dense matrix using the @␣
 ↪operator.
```

```
    Args:
    A: Dense matrix.
    v: Vector to multiply.

    Returns:
    Product of A and v.
    """

    return A @ v
```

[39]:
```
for n in range(0,200,10):
    A = scipy.sparse.random(n,n,0.2)
    B = scipy.sparse.random(n,n,0.2)
    add1 = add_sparse(A,B) # Class addition of sparse matrices
    add2 = csr_matrix(A+B).toarray() # Scipy addition of sparse matrices
    assert np.allclose(add1,add2)
print('All good!')
```

```
All good!
```

[40]:
```
for n in range(2,200,10):
    A = scipy.sparse.random(n,n,0.2)
    v = np.random.rand(n)
    A2 = CSRMatrix(A)
    A3 = A.toarray()
    mvm1 = mv_sparse(A2,v)
    mvm2 = mv_dense(A3,v)
    assert np.allclose(mvm1,mvm2)
print('All good!')
```

```
All good!
```

0.4 Part 1.3 Measure time to perfrom a `matvec` product

[41]:
```
from timeit import timeit

def function_timer(function, mat, vec, runs):
    """
    Measures the time taken by a matrix-vector multiplication operation.

    Args:
    function: Function to perform matrix-vector multiplication.
    mat: The matrix to multiply.
    vec: The vector to multiply.
    runs: Number of times to repeat the operation for averaging.
    """
```

```
    # Measure the average time taken for the specified number of runs
    return timeit(lambda: function(mat, vec), number=runs) / runs


def matvec_timer(function, runs, sizes, den):
    """
    Compares the performance of matrix-vector multiplication for different␣
 ↪matrix sizes.

    Args:
    function: Function to perform matrix-vector multiplication.
    runs: Number of times to repeat each operation for averaging.
    sizes: List of sizes for generating test matrices and vectors.
    """
    # Check if the number of runs is valid
    if runs <= 0:
        raise ValueError("Number of runs must be a positive integer")

    output = []
    for n in sizes:
        # Generate a random sparse matrix and corresponding dense matrix
        sparse_matrix = scipy.sparse.random(n, n, den)
        dense_matrix = sparse_matrix.toarray()

        # Random vector
        vector = np.random.rand(n)

        # Prepare the matrix for multiplication based on the function type
        csr_matrix = CSRMatrix(sparse_matrix) if function == mv_sparse else␣
 ↪dense_matrix

        # Measure and record the time for matrix-vector multiplication
        output.append(function_timer(function, csr_matrix, vector, runs))

    return output
```

A `sparse matrix` is characterized by having most of its elements as zero. The `density` of such a matrix is defined by the ratio of non zero elements to the total number of elements. Calculated by dividing the number of non zero elements by the total elements in the matrix.

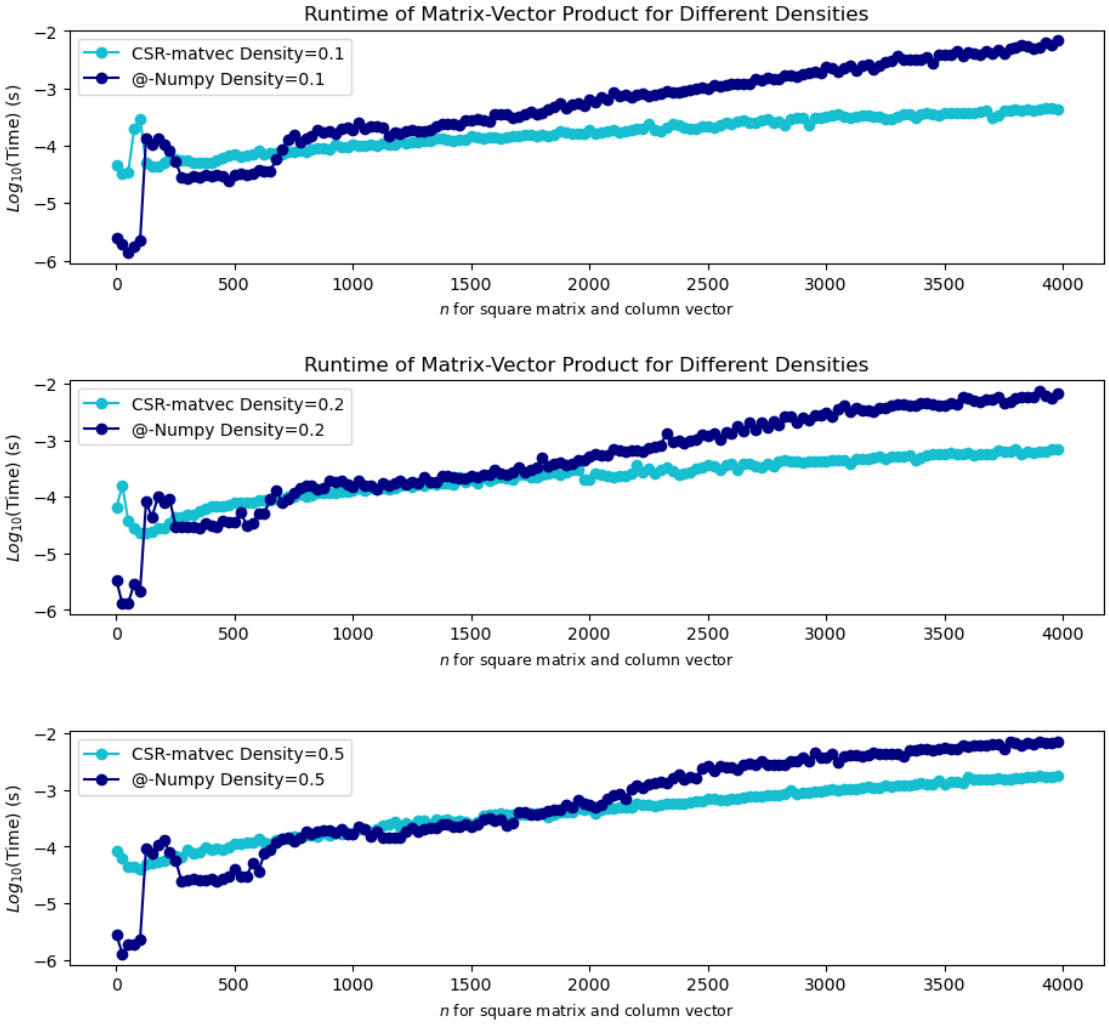
Density values chosen: 0.1, 0.2, 0.5

[57]:
```
stores = range(2, 4000, 25)
densities = [0.1, 0.2, 0.5]

# Storing results for different densities
results_sparse = {}
```

```
results_dense = {}

for density in densities:
    results_sparse[density] = matvec_timer(mv_sparse, 10, stores, density)
    results_dense[density] = matvec_timer(mv_dense, 10, stores, density)
```

[178]:
```
fig,ax = plt.subplots(3,1, figsize=(11,10))
ax[0].plot(stores, np.log10(results_sparse[0.1]), marker='o',c='#17becf',␣
 ↪label=f"CSR-matvec Density={0.1}")
ax[1].plot(stores, np.log10(results_sparse[0.2]), marker='o',c='#17becf',␣
 ↪label=f"CSR-matvec Density={0.2}")
ax[0].plot(stores, np.log10(results_dense[0.1]), marker='o', c='#000080',␣
 ↪label=f"@-Numpy Density={0.1}")
ax[1].plot(stores, np.log10(results_dense[0.2]), marker='o',c='#000080',␣
 ↪label=f"@-Numpy Density={0.2}")
ax[2].plot(stores, np.log10(results_sparse[0.5]), marker='o',c='#17becf',␣
 ↪label=f"CSR-matvec Density={0.5}")
ax[2].plot(stores, np.log10(results_dense[0.5]), marker='o',c='#000080',␣
 ↪label=f"@-Numpy Density={0.5}")
plt.subplots_adjust(hspace=0.5)  # Adjust the amount as needed

ax[0].set_title("Runtime of Matrix-Vector Product for Different Densities")
ax[1].set_title("Runtime of Matrix-Vector Product for Different Densities")
ax[0].set_xlabel("$n$ for square matrix and column vector", size=9)
ax[0].set_ylabel("$Log_{10}$(Time) (s)", size=10)
ax[1].set_xlabel("$n$ for square matrix and column vector", size=9)
ax[1].set_ylabel("$Log_{10}$(Time) (s)", size=10)
ax[2].set_xlabel("$n$ for square matrix and column vector", size=9)
ax[2].set_ylabel("$Log_{10}$(Time) (s)", size=10)
ax[0].legend()
ax[1].legend()
ax[2].legend()
```

[178]:
```
<matplotlib.legend.Legend at 0x7f03585a61d0>
```



This comparative analysis of the runtime performance for matrix-vector multiplication using a CSR format approach, as opposed to the conventional dense matrix operations using NumPy's `@` function, exhibits some interesting results.

As expected, for smaller matrix sizes denoted by nn, NumPy's operations are more efficient, while CSR performance lags. This trend is evident across all three density plots and can be attributed to the overhead associated with CSR's indirect addressing. For small matrices, this overhead renders CSR inefficient, as the setup time—including the creation of index pointers for rows and columns—may not be beneficial or efficient for small data structures. Conversely, NumPy can proceed directly with multiplication without such setup.

However, as matrix sizes increase, the advantages of the CSR method become apparent. The use of numba for multithreading significantly enhances CSR's performance, facilitating parallel processing of non-zero elements and allowing it to skip many zero multiplications. This advantage is particularly notable in larger sparse matrices, where CSR outperforms NumPy's implementation for dense matrices.

Especially when dealing with lower-density matrices, it is observed that as the matrix density increases, the advantages of the CSR methods become less pronounced when compared to dense NumPy calculations, which is to be expected.

Overall, the CSR implementation, particularly when augmented with numba's multithreading capabilities, emerges as a successful candidate to surpass NumPy in sparse matrix calculations.

0.5 Part 1.4 Solving a Matrix problem

In this section, we delve into solving linear system with a symmetric positive definite (SPD) matrix and a random vector, employing two iterative solvers: Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES). Our SPD matrix, crafted using the finite difference method, will be in CSR format.

The *CG* method, adept at handling symmetric positive-definite matrices, offers rapid convergence and low memory usage.

The *GMRES* method is a broader tool capable of addressing nonsymmetric systems by iteratively minimizing the residual.

We will do a comparative performance analysis of these two solvers, by focusing on their residuals.

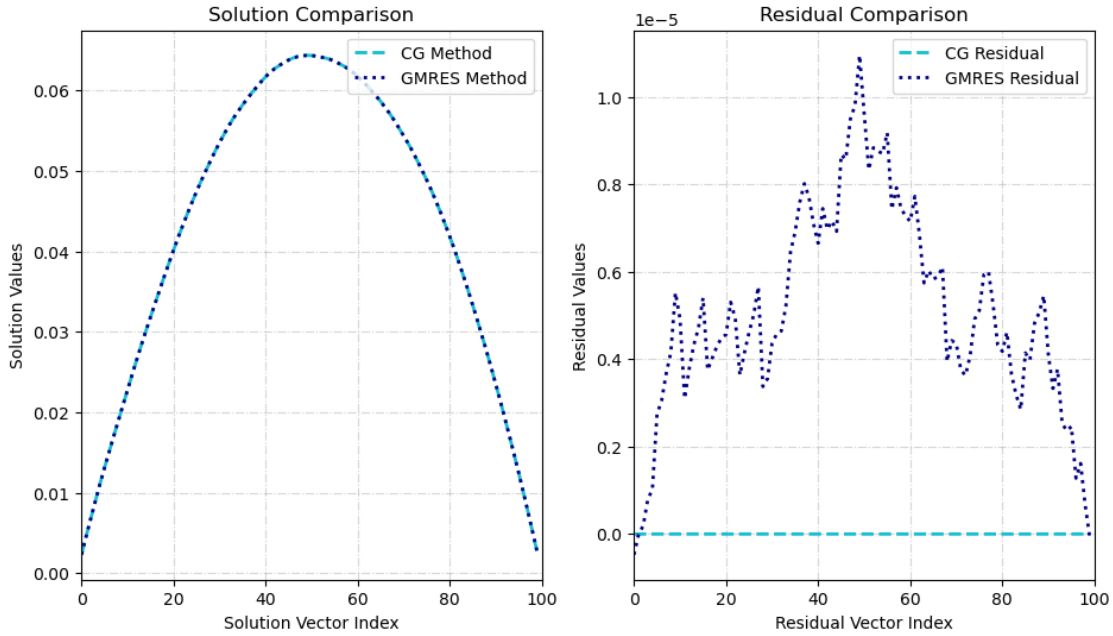
[62]:
```
def create_spd_matrix_vec(N):
    """
    Generates a symmetric positive definite (SPD) matrix and a random vector.

    The SPD matrix is created using the finite difference method for the
    second-order differential equation -y''(x). The matrix is constructed
    as a tridiagonal matrix.

    Args:
    N: Size of the matrix and vector.

    Returns:
    A: Symmetric positive definite matrix in CSR format.
    b: Randomly generated vector.
    """
    # Define the diagonal and off-diagonal values using the finite difference␣
↪scheme
    diagonal = np.full(N, 2 * N**2)
    off_diag = -N**2

    # Create a tridiagonal matrix in COO format
    A_coo = diags([off_diag, diagonal, off_diag], [-1, 0, 1], shape=(N, N),␣
↪format='coo')

    # Check if the matrix is positive definite by verifying its smallest␣
↪eigenvalue
    assert np.all(eigs(A_coo, k=1, which='SM')[0] > 0)
```

```
    # Convert the matrix to CSR format
    A = CSRMatrix(A_coo)

    # Generate a random vector
    b = np.random.rand(N)

    return A, b

# Create SPD matrix and vector
A, b = create_spd_matrix_vec(100) # size = 100

# Linear system solved using Conjugate Gradient (CG) and GMRES methods
sol_cg, info_cg = cg(A, b)
sol_gmres, info_gmres = gmres(A, b)

# Residuals to assess the accuracy of each solution
res_cg = b - A @ sol_cg      # Residual (CG solution)
res_gmres = b - A @ sol_gmres  # Residual (GMRES solution)
```

[65]:
```
fig, (ax1, ax2) = plt.subplots(1, 2)
fig.set_size_inches(11, 6)

# Sol Plot
ax1.plot(sol_cg, c='#17becf', linestyle='--', linewidth=2)
ax1.plot(sol_gmres, c='#000080', linestyle=':', linewidth=2)
ax1.set(title="Solution Comparison")
ax1.set_xlabel('Solution Vector Index', fontsize=10)
ax1.set_ylabel('Solution Values', fontsize=10)
ax1.set_xlim(0, 100)
ax1.grid(True, linestyle='-.', alpha=0.5)
ax1.legend(["CG Method", "GMRES Method"], loc='upper right')

# Residual Plot
ax2.plot(res_cg, c='#17becf', linestyle='--', linewidth=2)
ax2.plot(res_gmres, c='#000080', linestyle=':', linewidth=2)
ax2.set(title="Residual Comparison")
ax2.set_xlabel('Residual Vector Index', fontsize=10)
ax2.set_ylabel('Residual Values', fontsize=10)
ax2.set_xlim(0, 100)
ax2.grid(True, linestyle='-.', alpha=0.5)
ax2.legend(["CG Residual", "GMRES Residual"], loc='upper right')

# Print norms
print("norm of GMRES' residual: ", np.linalg.norm(res_gmres))
print("norm of CG's residual: ", np.linalg.norm(res_cg))
```

```
norm of GMRES' residual:  5.5978205679202096e-05
```

```
norm of CG's residual:  5.144041408388208e-12
```



In the "solution comparison" plot, the solutions obtained from CG and GMRES methods seem to follow a similar trend, both solvers converingn to a solution for symmetric positive definite (SPD) system. The plot showcase that both methods are effective in this scenario, expected as the matrix properties are suitable for these two algorithms.

The "residual comparison" plot, there is a noticeable difference in the residual magnitudes between the two methods. While the CG method yields a residual that is nearly negligible, the GMRES method indicates a larger residual, still wihtin a very acceptable range.ç The goal in iterative methods is to minimize the residual, bringing it as close to zero as the computational precision allows.

The discrepancy here can be attributed to the mathematical properties of each algorithm; while GMRES minimizes the Euclidean norm of the residual, the CG minimizes the weigthed $A-norm$; $||x||_A = \sqrt{x^T Ax}$, a more tailored approach to the specific structure of SPD matrices.

0.6 Part 2 Implementing a custom matrix

[91]:
```python
class CustomMatrix(LinearOperator):
    def __init__(self, N):
        """
        Construct a custom matrix composed of two non-zero block a diagonal and␣
↪a dense block.
        Args:
            N (int): The size of the matrix to be generated.
        """
```

```
        # Main matrices to compute the custom matrix
        self.D = np.diag(np.random.rand(N,N))
        self.t = np.random.rand(N,2)
        self.w = np.random.rand(2,N)

        # top left
        self.diagonal = np.diag(self.D) # Diagonal Matrix
        # bottom right (A)
        self.dense = self.t @ self.w
        # A matrix (final 2n x2n)
        A = block_diag(self.diagonal,self.dense)

        self.dtype = A.dtype  # Set dtype based on A
        self.shape = A.shape

    def _matvec(self, vector):
        """
        Matrix-vector multiplication using the custom matrix's block structure.
        Args:
            vector: The vector to be multiplied with the custom matrix.
        Returns:
            np.ndarray: The result of the matrix-vector multiplication.
        """
        assert self.shape[1] == vector.shape[0], "Dimension mismatch"

        # Compute the multiplication for the upper and lower blocks separately.
        m = self.shape[0] // 2
        upper = self.D * vector[:m]  # Use the diagonal for the upper block.
        lower = (self.t @ self.w) @ vector[m:]  # Use the dense matrix for the
↪lower block.

        # Output vector.
        return np.concatenate([upper, lower])

    def densifier(self):
        """
        Convert the structured matrix to a full dense 2D array representation.
        Returns:
            np.ndarray: The dense matrix.
        """
        # Calculate dimensions for the zero padding matrix.
        n = self.shape[0] // 2
        zero_fill = np.zeros((n, n))

        # Construct the upper and lower halves of the dense matrix.
        upper_half = np.hstack([self.diagonal, zero_fill])  # Diagonal block
↪with zeros.
```

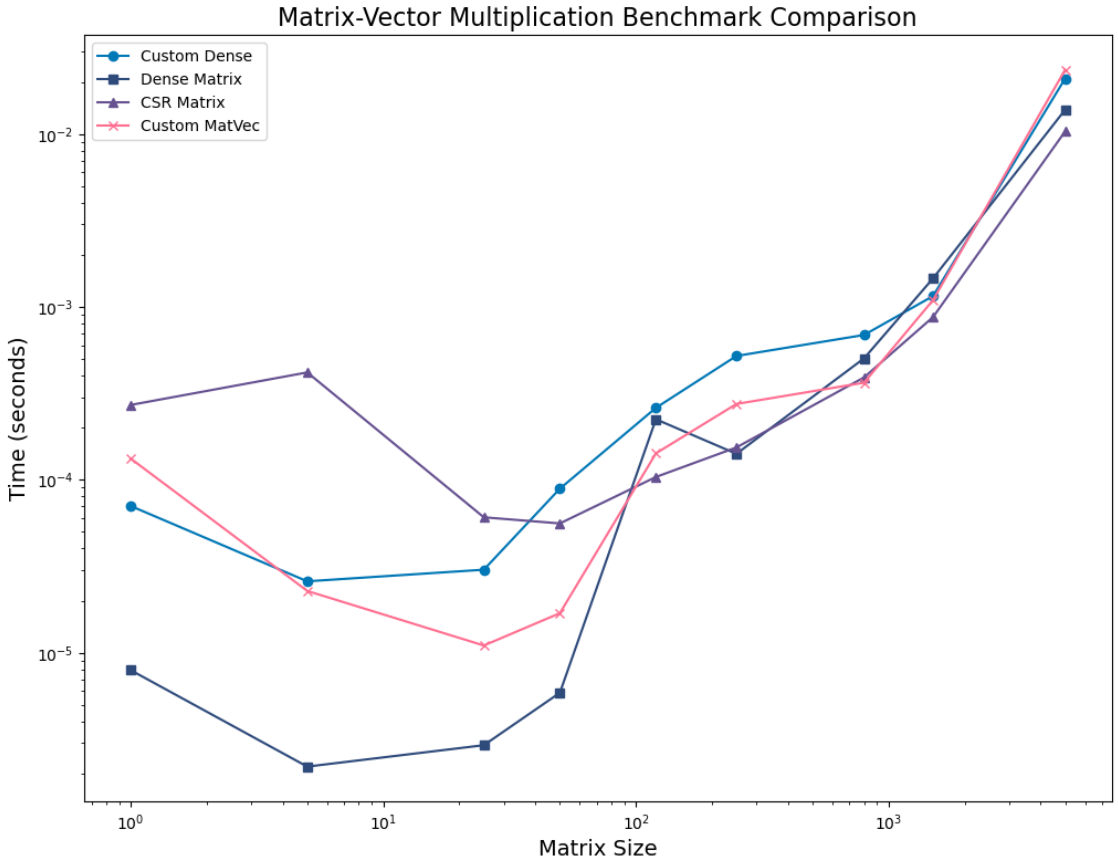
```
        lower_half = np.hstack([zero_fill, self.dense])  # Dense block with␣
↪zeros.

        # Complete dense matrix.
        return np.vstack([upper_half, lower_half])
```

[95]:
```
def generate_benchmarks(n):
    """
    Generates a custom matrix and its representations along with a vector for␣
↪benchmarks.

    Args:
        n (int): Matrix size.

    Returns:
        Custom matrix, dense matrix, CSR matrix, vector.
    """
    custom_matrix = CustomMatrix(n)  # Custom matrix.
    dense_matrix = custom_matrix.densifier()  # Dense numpy array form.
    csr_matrix = CSRMatrix(coo_matrix(dense_matrix))  # CSR format.
    vector = np.random.rand(2 * n)  # Random vector for multiplication.
    return custom_matrix, dense_matrix, csr_matrix, vector

def timer_matvec2(custom_matrix, dense_matrix, csr_matrix, vector):
    """
    Times matrix-vector multiplication for each matrix format.

    Args:
        custom_matrix, dense_matrix, csr_matrix: Matrices to benchmark.
        vector (np.ndarray): Vector for multiplication.

    Returns:
        Timings for each multiplication method.
    """
    # Measure multiplication times for each format.
    t_custom_dtw = timeit(lambda: custom_matrix @ vector, number=1)
    t_dense = timeit(lambda: dense_matrix @ vector, number=1)
    t_csr = timeit(lambda: csr_matrix @ vector, number=1)
    t_own = timeit(lambda: custom_matrix._matvec(vector), number=1)
    return t_custom_dtw, t_dense, t_csr, t_own
```

[96]:
```
matrix_sizes = [1, 5, 25, 50, 120, 250, 800, 1500, 5000]
#Storage
timing_results = np.zeros((4, len(matrix_sizes)))

for index, size in enumerate(matrix_sizes):
    # Generate benchmark matrices and vector for the current size.
```

```
    custom_linop, dense_mat, csr_mat, rand_vector = generate_benchmarks(size)
    # Record timing for matrix-vector multiplication using different methods.
    time_linop, time_dense, time_csr, time_custom = timer_matvec2(custom_linop,␣
 ↪dense_mat, csr_mat, rand_vector)
    timing_results[0, index] = time_linop
    timing_results[1, index] = time_dense
    timing_results[2, index] = time_csr
    timing_results[3, index] = time_custom
```

[173]:
```
# Define a more varied blue color palette
colors = ['#0077b6', '#2f4b7c', '#665191', '#ff6f91']
# Plotting
plt.figure(figsize=(12, 9))

# Plot for each method with distinct blue shades
plt.plot(matrix_sizes, timing_results[0], marker='o', color=colors[0],␣
 ↪label='Custom Dense')
plt.plot(matrix_sizes, timing_results[1], marker='s', color=colors[1],␣
 ↪label='Dense Matrix')
plt.plot(matrix_sizes, timing_results[2], marker='^', color=colors[2],␣
 ↪label='CSR Matrix')
plt.plot(matrix_sizes, timing_results[3], marker='x', color=colors[3],␣
 ↪label='Custom MatVec')

# Adding labels, title, and grid
plt.xlabel('Matrix Size', fontsize=14)
plt.ylabel('Time (seconds)', fontsize=14)
plt.title('Matrix-Vector Multiplication Benchmark Comparison', fontsize=16)
plt.xscale('log')
plt.yscale('log')
plt.legend()

# Show the plot
plt.show()
```



The time taken for the different data structures are similar, however it can be appreciable that for the `CustomMatrix` are slower, specially when computing by Numpy `@`operator. This might be cause the custom implementation requires constructing the matrix through matrix multiplication, which introduces a significant overhead, particularly for larger matrices. Added to that the custom function lacks of optimization techniques like Numba's Just-In-Time (JIT) compilation, slightly affecting its performance.

0.7 Part 3 Extensions

In the pursuit of optimizing sparse matrix operations, the Ellpack format (ELLPACK) stands out as a competitive alternative to the Compressed Sparse Row (CSR) format. In this extension, we aim to compare these two formats in terms of performance, specifically focusing on the matrix-vector multiplication operation.

Ellpack is designed to provide a regular data structure that is particularly advantageous for vectorized and parallel computations. Basically Ellapck format represents each row of the sparse matrix with a fixed number of elements in A, that corresponds to the number of non-zero elements in any row of the matrix. Then all rows are padded with zeros to fill up to the maximum number. A separate column J holds the inidces of all non-zero elements.

If we have a dense matrix:

```
|10   0    0   30    0    0|
| 0  40   50    0    0    0|
|60   0    0    0   70    0|
| 0  80    0   90    0  100|
|110  0    0    0    0  120|
| 0 130  140    0  150    0|
```

In Ellpack format it would be:

```
------A-------            ------J------
10  30   0   0             1   4   *   *
40  50   0   0             2   3   *   *
60  70   0   0             1   5   *   *
80  90 100   0             2   4   6   *
110 120  0   0             1   6   *   *
130 140 150  0             2   3   5   *
```

The * symbols represent arbitrary indices for the padded zeros in A

[145]:
```python
class EllpackMatrixCOO(LinearOperator):
    def __init__(self, coo):
        """
        Initialize the EllpackMatrixCOO object by converting a COO format␣
↪matrix to ELLPACK format.
        Args:
            coo (scipy.sparse.coo_matrix): The matrix in COO format to be␣
↪converted.
        """
        assert isinstance(coo, coo_matrix), "Input must be a COO matrix"

        self.shape = coo.shape  # Store the shape of the COO matrix.
        # Convert data types for compatibility with Numba.
        coo_data = coo.data.astype(np.float32)  # Non-zero values in float32.
        coo_row = coo.row.astype(np.int32)      # Row indices in int32.
        coo_col = coo.col.astype(np.int32)      # Column indices in int32.

        # Convert COO to Ellpack format.
        self.data, self.col_ind, self.rl = self.
↪convert_coo_to_ellpack(coo_data, coo_row, coo_col)
        self.rows = self.shape[0]  # Number of rows in the matrix.

    @staticmethod
    @numba.njit
    def convert_coo_to_ellpack(coo_data, coo_row, coo_col):
        """
        Convert a matrix from COO format to ELLPACK format.
        Args:
            coo_data: Array of non-zero values from the COO matrix.
            coo_row: Array of row indices for non-zero values.
```

```
            coo_col: Array of column indices for non-zero values.
        Returns:
            Tuple containing ELLPACK format data array, column indices, and row␣
↪lengths.
        """
        rows = np.max(coo_row) + 1
        max_nz_row = np.max(np.bincount(coo_row))  # Find max non-zero elements␣
↪in any row.

        # Initialize arrays for ELLPACK format.
        data = np.zeros((rows, max_nz_row), dtype=np.float32)      # Array for␣
↪non-zero values.
        col_ind = -1 * np.ones((rows, max_nz_row), dtype=np.int32) # Array for␣
↪column indices.
        rl = np.zeros(rows, dtype=np.int32)                        # Row length␣
↪array.

        # Fill the ELLPACK arrays with values and column indices.
        for i in range(len(coo_data)):
            row = coo_row[i]
            col = coo_col[i]
            value = coo_data[i]

            # Fill in the first available entry in the respective row.
            for j in range(max_nz_row):
                if col_ind[row, j] == -1:
                    data[row, j] = value
                    col_ind[row, j] = col
                    rl[row] += 1
                    break

        return data, col_ind, rl

    def _matvec(self, vec):
        """
        Override the matrix-vector multiplication for LinearOperator using␣
↪ELLPACK format.
        Args:
            vec: The vector to multiply.
        Returns:
            np.ndarray: The result of the matrix-vector multiplication.
        """
        # Ensure the input vector is a NumPy array and check dimension␣
↪compatibility.
        vec = np.asanyarray(vec, dtype=np.float32)
        if vec.shape != (self.shape[0],) and vec.shape != (self.shape[1],):
```

```
            raise ValueError('Dimension mismatch')

        # Perform matrix-vector multiplication in ELLPACK format.
        return self.matvec(vec, self.data, self.col_ind, self.rl)

    @staticmethod
    @numba.njit(parallel=True)
    def matvec(vec, data, col_ind, rl):
        """
        Perform matrix-vector multiplication in ELLPACK format.
        Args:
            vec: The vector to multiply.
            data: The non-zero values of the matrix in ELLPACK format.
            col_ind: The column indices of the non-zero values in ELLPACK␣
↪format.
            rl: The row lengths in the ELLPACK format.
        Returns:
            np.ndarray: The result of the matrix-vector multiplication.
        """
        rows = len(data)  # Number of rows in the matrix.
        result = np.zeros(rows, dtype=np.float32)  # Initialize the result␣
↪vector.

        # Perform multiplication for each row, considering only non-zero␣
↪elements.
        for row in numba.prange(rows):
            for column in range(rl[row]):
                if col_ind[row, column] != -1:
                    result[row] += data[row, column] * vec[col_ind[row, column]]
        return result
```

[148]:
```
def mv_dense(matrix, vector):
    """Perform matrix-vector multiplication using a dense matrix."""
    return np.dot(matrix, vector)

for n in range(2, 200, 10):
    # Generate a random sparse matrix in COO format and a random vector
    A_coo = scipy.sparse.random(n, n, 0.2, format='coo')
    vector = np.random.rand(n)

    # Convert the COO matrix to a dense format
    A_dense = A_coo.toarray()

    # Create an EllpackMatrixCOO instance from the COO matrix
    ellpack_matrix = EllpackMatrixCOO(A_coo)

    # Perform matrix-vector multiplication using dense matrix and Ellpack matrix
```

```
    mvm_dense = mv_dense(A_dense, vector)
    mvm_ellpack = ellpack_matrix._matvec(vector)

    # Assert that results from Ellpack are close to dense results
    assert np.allclose(mvm_ellpack, mvm_dense), "Mismatch with dense result"

print('All good!')
```

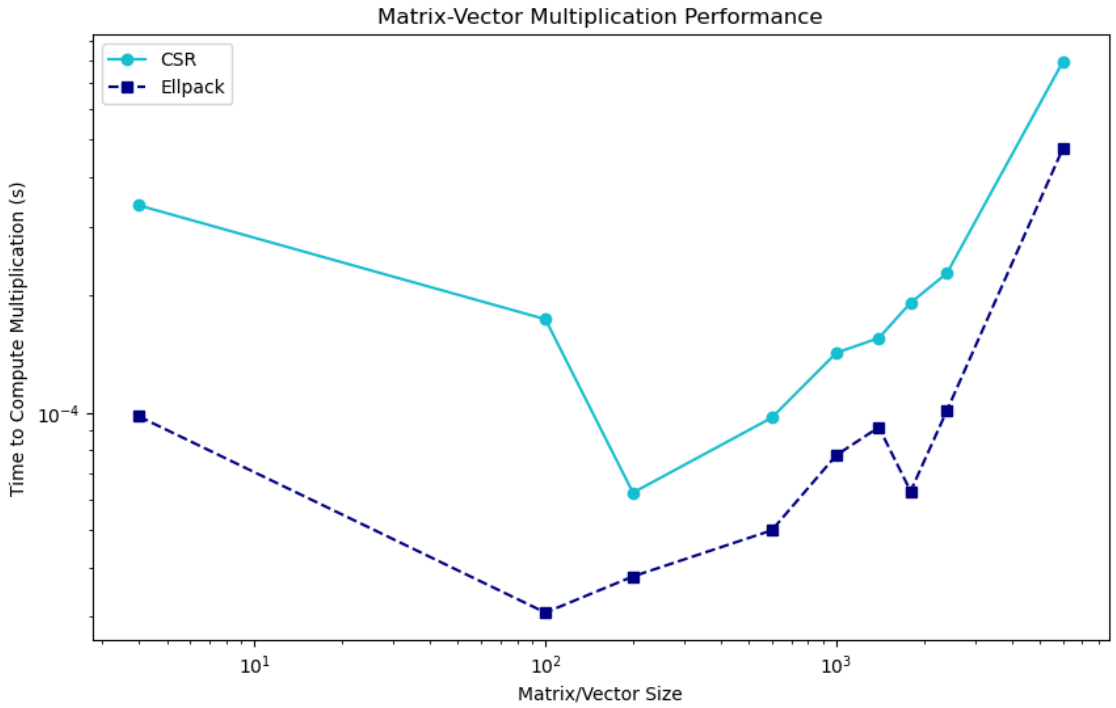
```
All good!
```

[149]:
```
# Similar functions to the previous ones used in the assignment
def generate_ext_bench(n):
    A = scipy.sparse.random(n,n,0.2)
    vector = np.random.rand(n)
    csr = CSRMatrix(A) # Using our previous class
    coo = coo_matrix(A)
    ellpack_coo = EllpackMatrixCOO(coo)
    return csr, ellpack_coo, vector

def timer_ellpack_vs_csr_matvec(csr_matrix,ellpack_matrix_coo, vector):
    t_csr = timeit(lambda: csr_matrix.matvec(vector), number=1)
    t_ellpack_coo = timeit(lambda: ellpack_matrix_coo._matvec(vector), number␣
 ↪=1)

    return  t_csr, t_ellpack_coo
```

[162]:
```
n_range = [2, 50, 100,300, 500, 700, 900, 1200, 3000 ]

time_benchmark = np.zeros((2, len(n_range)))
for i, n in enumerate(n_range):
    csr_matrix,ellpack_matrix_coo, vector = generate_ext_bench(n)
    t_csr, t_ellpack_coo =␣
 ↪timer_ellpack_vs_csr_matvec(csr_matrix,ellpack_matrix_coo, vector)
    time_benchmark[0, i] = t_csr
    time_benchmark[1, i] = t_ellpack_coo
```

[167]:
```
plt.figure(figsize=(10, 6))
plt.plot(2 * np.array(n_range), time_benchmark[0, :], '-o', color='#17becf',␣
 ↪label="CSR")
plt.plot(2 * np.array(n_range), time_benchmark[1, :], '--s', color='#000080',␣
 ↪label="Ellpack")
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Matrix/Vector Size")
plt.ylabel("Time to Compute Multiplication (s)")
plt.title("Matrix-Vector Multiplication Performance")
plt.legend()
```

```
plt.show()
```



These results indicate that for smaller matrices, the Ellpack format appears to be more efficient, which is counterintuitive since sparse matrix formats like Ellpack are typically optimized for larger matrices. This efficiency may be due to the overhead from padding in Ellpack. As matrix sizes increase, the performance gap narrows, suggesting that the padding overhead becomes less significant.

The results show that, although Ellpack has considerable potential in certain scenarios, CSR continues to provide highly efficient computation across a diverse range of situations.