Assignment 3 - Sparse Matrices

November 29, 2022

1 Assignment 3 - Sparse Matrices

link: https://tbetcke.github.io/hpc_lecture_notes/2022-assignment_3.html

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

import scipy.sparse as sps
from scipy.sparse.linalg import gmres, cg
from scipy.sparse.linalg import LinearOperator

from copy import deepcopy
import time
```

1.1 Part 1: Implementing a CSR matrix

1.1.1 1.a Implement the methods `__init__`, `__add__` and `matvec`

[2]:
```
class CSRMatrix(LinearOperator):

    def __init__(self, coo_mat):
        self.dtype = coo_mat.dtype
        self.shape = coo_mat.shape

        # Computes index ordering that sorts elements
        # with row index as primary sort, and column index as secondary sort
        sorter = np.lexsort((coo_mat.col, coo_mat.row))

        self.data = coo_mat.data[sorter]
        self.indices = coo_mat.col[sorter]
        self.indptr = self.row_indices_to_indptr(coo_mat.row[sorter], coo_mat.
 ↪shape[0])

    def row_indices_to_indptr(self, coo_rows, rows_count):
        """Computes the index pointers from an array of row indices"""
        indptr = []

        counts = np.bincount(coo_rows, minlength=rows_count)
```

```
    # Initializing by appending 1st index
    indptr.append(0)

    for count in counts:
        previous_ptr = indptr[-1]

        # Next pointer value is the previous pointer value
        # plus the number of elements in that row
        next_pointer = previous_ptr + count

        indptr.append(next_pointer)

    return np.array(indptr)



def __add__(self, other):
    """Add the CSR matrix other to this matrix."""
    col_indices1 = self.indices
    col_indices2 = other.indices
    data1 = self.data
    data2 = other.data
    index_pointers1 = self.indptr
    index_pointers2 = other.indptr

    number_of_rows = len(index_pointers1) - 1

    new_col_indices = []
    new_data = []
    new_index_pointers = []

    # Initializing index pointers
    new_index_pointers.append(0)

    for row_index in range(number_of_rows):
        # Getting index pointers, column indices, and data for this row
        indptr_row1 = index_pointers1[row_index:row_index+2]
        indptr_row2 = index_pointers2[row_index:row_index+2]

        col_indices_row1 = col_indices1[indptr_row1[0]:indptr_row1[1]]
        col_indices_row2 = col_indices2[indptr_row2[0]:indptr_row2[1]]

        data_row1 = data1[indptr_row1[0]:indptr_row1[1]]
        data_row2 = data2[indptr_row2[0]:indptr_row2[1]]

        # Computing the new column indices and data for this row
```

```
            new_col_indices_of_row, new_data_of_row = self.
↪add_1_row(col_indices_row1,
                                                                  ␣
↪col_indices_row2,
                                                                   data_row1,
                                                                   data_row2)

            # Computing the new index pointer for this row
            new_index_pointers_of_row_diff = len(new_col_indices_of_row)
            new_index_pointer_of_row = new_index_pointers[row_index] +␣
↪new_index_pointers_of_row_diff

            # Appending for the new matrix
            new_col_indices.extend(new_col_indices_of_row)
            new_data.extend(new_data_of_row)
            new_index_pointers.append(new_index_pointer_of_row)

        self.indices = new_col_indices
        self.data = new_data
        self.indptr = new_index_pointers

        # getting a deep copy to avoid having to convert to coo and then back␣
↪to csr, which
        # is what would be needed if I did "return CSRMatrix(coo_format)"
        return deepcopy(self)

    def add_1_row(self, col_indices1, col_indices2, data1, data2):
        """
        Computes the resulting indices and data for a single row of the addition
        of two matrices, 1 and 2.

        col_indices{i}: "indices" for a single row (slice of the indices␣
↪property of the original csr matrix)
        data{i}: "data" for a single row (slice of the data property of the␣
↪original csr matrix)

        returns new_col_indices, new_data
        """
        len1 = len(col_indices1)
        len2 = len(col_indices2)

        index1 = 0
        index2 = 0

        new_col_indices = []
        new_data = []
```

```
        finished_with_1 = False if index1 < len1 else True
        finished_with_2 = False if index2 < len2 else True
        finished_with_both = finished_with_1 and finished_with_2

        while not finished_with_both:
            # if finished with matrix 1, then simply add the rest of the␣
↪elements from 2
            if finished_with_1:
                col_index2 = col_indices2[index2]
                new_col_indices.append(col_index2) # add column index to␣
↪col_indices of new matrix
                new_data.append(data2[index2]) # add the corresponding data␣
↪point to data of the new matrix
                index2 += 1 # switch to next item in matrix 2

            # if finished with matrix 2, then simply add the rest of the␣
↪elements from 1
            elif finished_with_2:
                col_index1 = col_indices1[index1]
                new_col_indices.append(col_index1) # add column index to␣
↪col_indices of new matrix
                new_data.append(data1[index1]) # add the corresponding data␣
↪point to data of the new matrix
                index1 += 1 # switch to next item in matrix 2

            # if not finished with either, you must compare the column indices␣
↪to get the ordering right or add
            else:
                col_index1 = col_indices1[index1]
                col_index2 = col_indices2[index2]

                # if the column index of the 1st matrix comes before that of␣
↪the 2nd matrix,
                # then it is appended first
                if col_index1 < col_index2:
                    new_col_indices.append(col_index1) # add column index to␣
↪col_indices of new matrix
                    new_data.append(data1[index1]) # add the corresponding data␣
↪point to data of the new matrix
                    index1 += 1 # switch to next item in matrix 2

                # if the column index of the 2st matrix comes before that of␣
↪the 1nd matrix,
                # then it is appended first
                elif col_index1 > col_index2:
```

```
                    new_col_indices.append(col_index2) # add column index to␣
↪col_indices of new matrix
                    new_data.append(data2[index2]) # add the corresponding data␣
↪point to data of the new matrix
                    index2 += 1 # switch to next item in matrix 2

                # if the column indices are equal
                elif col_index1 == col_index2:
                    new_col_indices.append(col_index1)
                    new_data.append(data1[index1]+data2[index2])
                    # switch to next item for both matrices
                    index1 += 1
                    index2 += 1

            # Checking whether we have reached the end of both arrays
            if index1 >= len1:
                finished_with_1 = True
            if index2 >= len2:
                finished_with_2 = True
            finished_with_both = finished_with_1 and finished_with_2


        return new_col_indices, new_data


    def _matvec(self, vector):
        """Compute a matrix-vector product."""
        assert self.shape[1] == vector.shape[0]


        nrows = self.shape[0]
        C = np.zeros((nrows,))

        for i in range(nrows):
                indptr1 = self.indptr[i]
                indptr2 = self.indptr[i+1]

                # case where row i contains n non-zero elements
                if indptr1 == indptr2:
                    continue

                C_i = 0

                col_indices_of_row = self.indices[indptr1:indptr2]
                data_of_row = self.data[indptr1:indptr2]

                for p in range(len(data_of_row)):
```

```
                    col_index = col_indices_of_row[p]
                    C_i += data_of_row[p] * vector[col_index]

                C[i] = C_i

        return C
```

1.1.2 1.b Write tests to check that the add and matvec methods that you have written are correct

[3]:
```
# Checking __init__

test_count = 50

for i in range(test_count):
    row_length = np.random.randint(1, 200)
    col_length = np.random.randint(1, 200)

    A = sps.random(row_length, col_length, density=0.25)

    A_csr_truth = sps.csr_matrix(A)
    A_csr_mine = CSRMatrix(A)

    data_equal = np.allclose(A_csr_truth.data, A_csr_mine.data)
    indices_equal = np.array_equal(A_csr_truth.indices, A_csr_mine.indices)
    indptr_equal = np.array_equal(A_csr_truth.indptr, A_csr_mine.indptr)

    assert data_equal and indices_equal and indptr_equal
```

[4]:
```
# Checking _add__

test_count = 50

for i in range(test_count):
    row_length = np.random.randint(1, 200)
    col_length = np.random.randint(1, 200)
    dims = (row_length, col_length)

    A = sps.random(*dims, density=0.3)
    B = sps.random(*dims, density=0.3)

    A_csr_truth = sps.csr_matrix(A)
    B_csr_truth = sps.csr_matrix(B)
    C_csr_truth = A_csr_truth + B_csr_truth

    A_csr_mine = CSRMatrix(A)
    B_csr_mine = CSRMatrix(B)
```

```
    C_csr_mine = A_csr_mine + B_csr_mine

    data_equal = np.allclose(C_csr_truth.data, C_csr_mine.data)
    indices_equal = np.array_equal(C_csr_truth.indices, C_csr_mine.indices)
    indptr_equal = np.array_equal(C_csr_truth.indptr, C_csr_mine.indptr)

    assert data_equal and indices_equal and indptr_equal
```

[5]:
```
# Checking _matvec

test_count = 50

for i in range(test_count):
    row_length = np.random.randint(1, 200)
    col_length = np.random.randint(1, 200)
    dims = (row_length, col_length)

    A = sps.random(*dims, density=0.3)
    B = np.random.rand(dims[1])

    A_csr_truth = sps.csr_matrix(A)
    C_truth = A_csr_truth @ B

    A_csr_mine = CSRMatrix(A)
    C_mine = A_csr_mine @ B

    assert np.allclose(C_truth, C_mine)
```

1.1.3 1.c

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 taken 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.

[6]:
```
# Combining all three tasks of 1.c into one

sizes = [100, 200, 500, 1000, 2000, 4000, 6000, 8000, 10000]

times = np.zeros((2, len(sizes)))

for i in range(len(sizes)):
    size = sizes[i]
    B = np.random.rand(size)
```

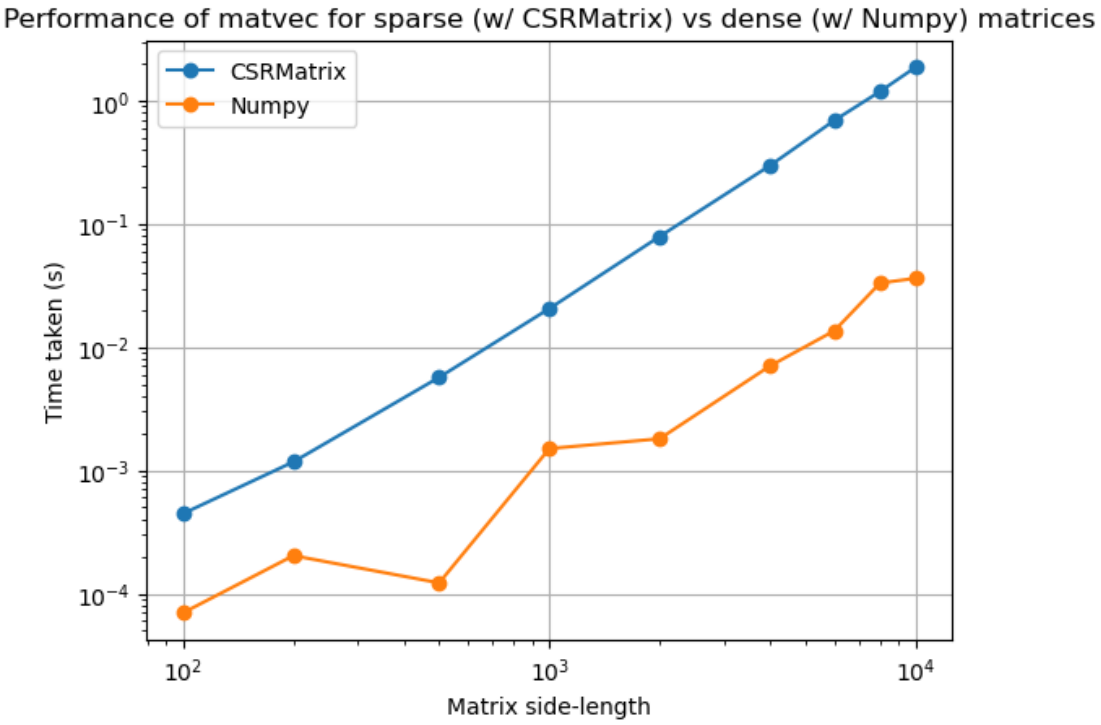
```
    A_sparse = sps.random(size, size, density=0.05)
    A_csr = CSRMatrix(A_sparse)

    start = time.time()
    A_csr @ B
    end = time.time()
    times[0, i] = end-start


    A_dense = np.random.rand(size, size)

    start = time.time()
    A_dense @ B
    end = time.time()
    times[1, i] = end-start
```

[7]:
```
plt.plot(sizes, times[0], "-o", label="CSRMatrix")
plt.plot(sizes, times[1], "-o", label="Numpy")
plt.grid()
plt.yscale("log")
plt.xscale("log")
plt.xlabel("Matrix side-length")
plt.ylabel("Time taken (s)")
plt.title("Performance of matvec for sparse (w/ CSRMatrix) vs dense (w/ Numpy)␣
 ↪matrices")
plt.legend()
plt.show()
```



Comment Numpy remains several orders of magnitude faster (~100 times faster) than CSRMatrix to compute matrix vector product, across all tested matrix sizes. This occurs despite Numpy running on dense matrices (100% density) and CSRMatrix on sparse matrices with 5% density.

1.1.4 1.e

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).

[15]:
```
size = 100

B = np.random.rand(size)

# Creating a positive definite matrix (for cg to work)
rand = np.random.RandomState(0)
Q, _ = np.linalg.qr(rand.randn(size, size))
D = np.diag(rand.rand(size))
A = sps.coo_matrix(Q.T @ D @ Q)

A_CSRMatrix = CSRMatrix(A)
```

[16]:
```
tol = 10e-5
x_gmres, info_gmres = gmres(A_CSRMatrix, B, atol=tol)
x_cg, info_cg = cg(A_CSRMatrix, B, atol=tol)
```

[17]:
```
tol = 10e-5
print(f"Are within {tol} tolerance: {np.allclose(x_gmres, x_cg, rtol=tol,␣
 ↪atol=tol)}")

tol = 10e-6
print(f"Are within {tol} tolerance: {np.allclose(x_gmres, x_cg, rtol=tol,␣
 ↪atol=tol)}")

print(f"Are equal: {np.array_equal(x_gmres, x_cg)}")
```

```
Are within 0.0001 tolerance: True
Are within 1e-05 tolerance: False
Are equal: False
```

Comment We can see that the two solutions are not exactly equal, but are equal within a certain tolerance (tol=10e-5). This is explained by the tolerance we have set for both solvers (also 10e-5), which defined the accepted closeness to the true answer. Note that the cg solver worked (and didn't give out nan) because we used a positive definite matrix.

1.2 Part 2: Implementing a custom matrix

1.2.1 2.a 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.

[18]:
```
class CustomMatrix(LinearOperator):
    def __init__(self, diag, T, W):
        assert diag.size == T.shape[0] and diag.size == W.shape[1]
        self.dtype = diag.dtype
        self.n = diag.size
        self.shape = (2*self.n, 2*self.n)

        self.diag = diag
        self.T_ = T
        self.W_ = W

        self.A_tilde = T @ W
        self.full_matrix = self.full_matrix()

    def full_matrix(self):
        """Returns the full matrix as a numpy array"""
```

```
        mat = np.zeros(self.shape)
        np.fill_diagonal(mat[:self.n, :self.n], self.diag)
        mat[self.n:, self.n:] = self.A_tilde

        return mat

    def _matvec(self, vector):
        """Computes the matrix vector product with 1D Numpy array vector"""
        assert vector.size == self.shape[1]

        upper_half = self.diag * vector[:self.n]
        lower_half = self.A_tilde @ vector[self.n:]

        return np.concatenate((upper_half, lower_half))
```

1.2.2 2.b

For a range of values of n, create matrices where the entries on the diagonal of the top-left block and in the matrices and are random numbers.

[19]:
```
n_store = [2, 5, 10, 50, 100, 200, 500, 1000, 2000, 5000, 7500, 10000]
M_store = []
B_store = []

for i in range(len(n_store)):
    n = n_store[i]

    T = np.random.rand(n, 2)
    W = np.random.rand(2, n)
    diag = np.random.rand(n)

    M = CustomMatrix(diag, T, W)
    M_store.append(M)

    B = np.random.rand(2*n)
    B_store.append(B)
```

1.2.3 2.c

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.

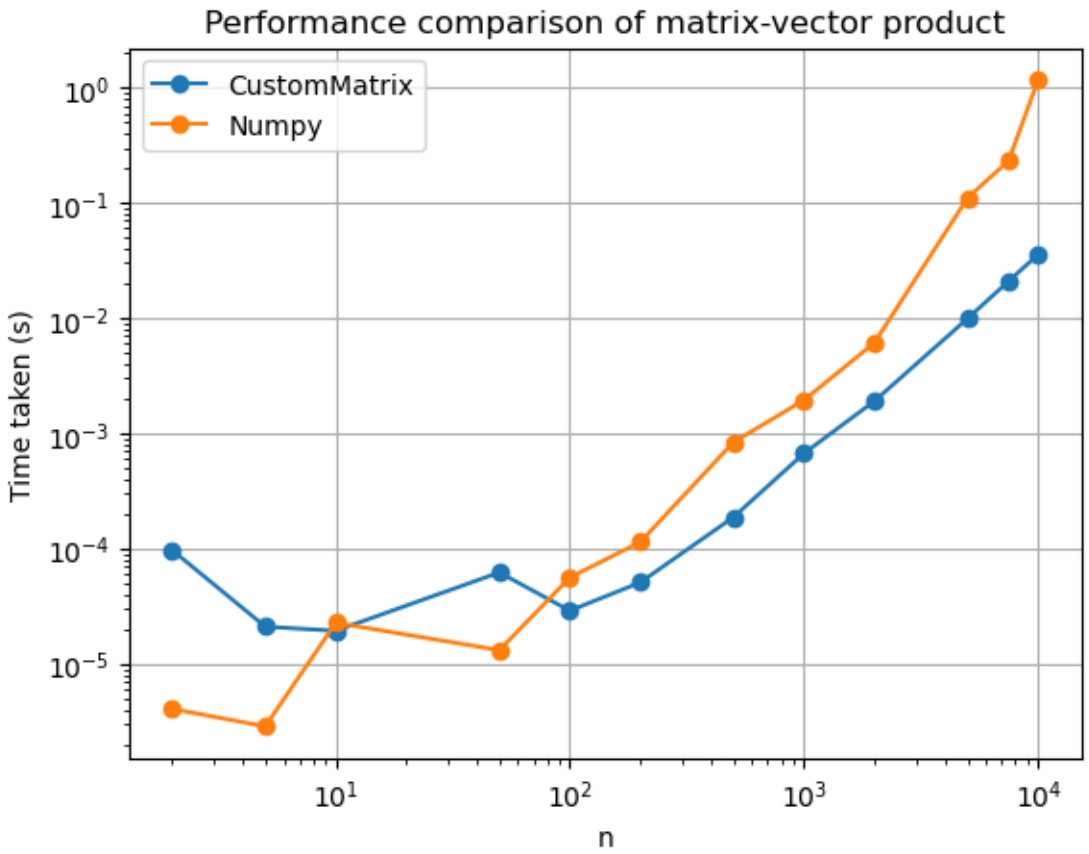
[20]:
```
custom_time_store = []
numpy_time_store = []
```

```
for i in range(len(n_store)):
    M = M_store[i]
    B = B_store[i]

    start = time.time()
    M @ B
    end = time.time()
    custom_time_store.append(end-start)

    start = time.time()
    # M.full_matrix already contains a computed Numpy array
    # of the full matrix
    M.full_matrix @ B
    end = time.time()
    numpy_time_store.append(end-start)
```

[21]:
```
plt.plot(n_store, custom_time_store, "-o", label="CustomMatrix")
plt.plot(n_store, numpy_time_store, "-o", label="Numpy")
plt.xlabel("n")
plt.ylabel("Time taken (s)")
plt.grid()
plt.yscale("log")
plt.xscale("log")
plt.legend()
plt.title("Performance comparison of matrix-vector product")
plt.show()
```



1.2.4 2.e 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).

Comment Though Numpy's matvec is faster for small matrices (up to matrix side-length $\sim 2n = 200$) because it is highly optimized, we can see CustomMatrix's matvec becomes fastest for larger matrices.

This is because Numpy doesn't take into account the matrix's structure (0s in two quadrants), and hence computes the vecmat of the full 2n by 2n matrix with the 2n-sized vector ($\sim 4n^2$ float multiplications and $4n^2$ additions). However, CustomMatrix computes n float multiplications (for the diagonal) and vecmat of a n by n matrix with an n-sized vector, both using Numpy ($\sim n^2 + n$ multiplications and n^2 additions).

This means that both operations scale as $O(n^2)$, which is reflected in the similarity of their slopes' shape, however CustomMatrix's matrix plot is shifted down by some constant factor (expected to be by about $log(4)$ from the analysis above).

[ ]: