Assignment 3 - Sparse matrices

Contents

This assignment makes up 30% of the overall marks for the course. The deadline for submitting this assignment is **5pm on Thursday 1 December 2022**.

Coursework is to be submitted using the link on Moodle. You should submit a single pdf file containing your code, the output when you run your code, and your answers to any text questions included in the assessment. The easiest ways to create this file are:

- Write your code and answers in a Jupyter notebook, then select File -> Download as -> PDF via LaTeX (.pdf).
- Write your code and answers on Google Colab, then select File -> Print, and print it as a pdf.

Tasks you are required to carry out and questions you are required to answer are shown in bold below.

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 to 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 an library's implementation of a CSR matrix. The method `__add__` will overload `+` and so allow you 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 test 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 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**.

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 A by a $2n$ by $2n$ matrix with the following structure:

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

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

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

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

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

where T is a n by 2 matrix (a tall matrix); and where 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).

By Timo Betcke & Matthew Scroggs