Sparse Matrix data structures

We consider the following simple matrix.

```
import numpy as np

A = np.array([
    [1, 0, 0, 2, 0],
    [3, 4, 0, 5, 0],
    [6, 0, 7, 8, 9],
    [0, 0, 10, 11, 0],
    [0, 0, 0, 0, 12]
])
print(A)
```

```
[[ 1  0  0  2  0]
 [ 3  4  0  5  0]
 [ 6  0  7  8  9]
 [ 0  0 10 11  0]
 [ 0  0  0  0 12]]
```

In the following we want to give a simple overview of sparse data formats to store this matrix. In this example, although we have a number of zero entries, sparse matrix formats are not worthwile, and we use this example mainly for didactical purposes.

The COO (Coordinate) format

We start with the COO format. It is the most simple format. Let us conver the matrix into it.

```
from scipy.sparse import coo_matrix

A_coo = coo_matrix(A)
```

The coo format is a very simple format that explicitly stores the row entries. It consists of three arrays, the row indices, the column indicies and the data entries. Let us print those arrays.

```
print(A_coo.row)
print(A_coo.col)
print(A_coo.data)
```

```
[0 0 1 1 1 2 2 2 2 3 3 4]
[0 3 0 1 3 0 2 3 4 2 3 4]
[ 1  2  3  4  5  6  7  8  9 10 11 12]
```

We can easily print out the triplets of row index, column index and associated data entry.

```
list(zip(A_coo.row, A_coo.col, A_coo.data))
```

```
[(0, 0, 1),
 (0, 3, 2),
 (1, 0, 3),
 (1, 1, 4),
 (1, 3, 5),
 (2, 0, 6),
 (2, 2, 7),
 (2, 3, 8),
 (2, 4, 9),
 (3, 2, 10),
 (3, 3, 11),
 (4, 4, 12)]
```

The coo format in Scipy is most frequently used for the generation of sparse matrices. The format is very simple and we can use it to easily create sparse matrices. We only need to provide the row, column and data arrays to create the coo matrix. A major advantage is also that we can repeat indices. In the matrix creation all data entries associated with the same matrix entry is just summed up. This is a very natural operation and simplifies a number of a situations, where we need to create sparse matrices.

However, coo is not a suitable format for typical matrix operations. Also, it is not yet optimal in terms of storage requirements.

The CSR (Compressed Sparse Row) Format

If we look at the printout of the indices above in the coo format we can see that there is a lot of repetition in the row indices. We store for each nonzero entry the row index even though all row indices within the same row are identical. This motivates the idea of the CSR (Compressed Sparse Row) format. Instead of the row array we store an array of index pointers that give the starting position of the row within the column array. Let us demonstrate how this works.

We first conver the COO matrix format into the CSR format.

```
A_csr = A_coo.tocsr()
```

Let us now print out the arrays that define the CSR format. We have three arrays.

- A_csr.data - The data array containing the nonzero entries
- A_csr.indices - The column indices for the nonzero entries
- A_csr.indptr - Pointers into the column indices to store which indices belong to which row.

The first two are the same as in the COO format. The last one requires explanation. For this let us print out the three arrays.

```
print(A_csr.data)
print(A_csr.indices)
print(A_csr.indptr)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12]
[0 3 0 1 3 0 2 3 4 2 3 4]
[ 0  2  5  9 11 12]
```

Comparing the arrays shows that the first two are indeed identical to the corresponding arrays for the COO format. The third array tells us where in the `indices` array the column indices for the ith row are located, namely we have that the column indices for the ith row are located in

```
indices[indptr[i] : indptr[i + 1]]
```

Correspondingly the assocated data entries are in

```
data[indptr[i] : indptr[i + 1]]
```

The `indptr` array is always 1 element larger than the number of rows in the matrix. The last entry of the `indptr` array is the total number of nonzero elements.

There is also a variant of the CSR format that stores elements along columns and compresses the column pointers. This is called CSC (Compressed Sparse Column) Format. Both CSC and CSR are widely used in software for large sparse matrices.

CSR Matrix-vector products

The CSR format has a very simple implementation for the matrix-vector product that naturally parallelises on multithreaded CPUs. The following code shows an example implementation.

```
import numba

@numba.jit(nopython=True, parallel=True)
def csr_matvec(data, indices, indptr, shape, x):
    """Evaluates the matrix-vector product with a CSR matrix."""
    # Get the rows and columns

    m, n = shape

    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
```

Let's test this against the Scipy provided implementation of sparse matrix multiplications. As test matrix we use the matrix generated with the `discretise_poission` routine.

```
from scipy.sparse import coo_matrix

def discretise_poisson(N):
    """Generate the matrix and rhs associated with the discrete Poisson operator."""

    nelements = 5 * N**2 - 16 * N + 16

    row_ind = np.empty(nelements, dtype=np.float64)
    col_ind = np.empty(nelements, dtype=np.float64)
    data = np.empty(nelements, dtype=np.float64)

    f = np.empty(N * N, dtype=np.float64)

    count = 0
    for j in range(N):
        for i in range(N):
            if i == 0 or i == N - 1 or j == 0 or j == N - 1:
                row_ind[count] = col_ind[count] = j * N + i
                data[count] =  1
                f[j * N + i] = 0
                count += 1

            else:
                row_ind[count : count + 5] = j * N + i
                col_ind[count] = j * N + i
                col_ind[count + 1] = j * N + i + 1
                col_ind[count + 2] = j * N + i - 1
                col_ind[count + 3] = (j + 1) * N + i
                col_ind[count + 4] = (j - 1) * N + i

                data[count] = 4 * (N - 1)**2
                data[count + 1 : count + 5] = - (N - 1)**2
                f[j * N + i] = 1

                count += 5

    return coo_matrix((data, (row_ind, col_ind)), shape=(N**2, N**2)).tocsr(), f
```

```
N = 1000

A, _ = discretise_poisson(N)

# Generate a random vector
rand = np.random.RandomState(0)
x = rand.randn(N * N)

y = csr_matvec(A.data, A.indices, A.indptr, A.shape, x)

# Compare with the Scipy sparse matrix multiplication

y_exact = A @ x
rel_error = np.linalg.norm(y - y_exact, np.inf) / np.linalg.norm(y_exact, np.inf)
print(f"Error: {round(rel_error, 2)}.")
```

```
Error: 0.0.
```

This demonstrates that our implementation is correct. Not only it is correct. It also uses multithreading for parallelism. The default Scipy implementation is only single-threaded. For many sizes this does not matter. But for very large problems this can become a performance bottleneck.

Let us time our implementation against the Scipy one. We have chosen a matrix dimension of one million to have a sufficient size for the multithreading to be useful.

```
# Our implementation
%timeit y = csr_matvec(A.data, A.indices, A.indptr, A.shape, x)
```

```
5.31 ms ± 92.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
# The default Scipy implementation
%timeit y = A @ x
```

```
7.31 ms ± 102 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

We can see a small improvement against the default Scipy implementation. The improvement will be significantly more if we have many more elements per row as is the case for most three-dimensional problems or higher-order discretisation methods.

Other sparse formats.

There are a number of sparse matrix formats and Scipy is supporting several of them. More information on sparse matrix classes and operations for handling sparse matrices can be found at https://docs.scipy.org/doc/scipy/reference/sparse.html.

By Timo Betcke