1 The Matrix-Vector Product

Implement a function that takes as input a matrix $A \in \mathbb{R}^{m\times n}$ and a vector $x \in \mathbb{R}^n$ and returns the matrix-vector product Ax.

Implement the following ways of doing this:

1. **Dense:** Input expected as `numpy.ndarray`:
   Assume that the matrix and the vector are delivered to your function as `numpy.ndarray`.
   a) Implement the matrix-vector product "by hand" using for loops, i.e., *without* using `numpy.dot(A,x)` (or `numpy.matmul(A,x)` or `A@x`).
   b) Implement the matrix-vector product using `A.dot(x)`, `A@x`, `numpy.matmul(A,x)` or `numpy.dot(A,x)`.
2. **Sparse:** Matrix expected in CSR format:
   Assume that the matrix is delivered to your function as `scipy.sparse.csr_matrix` object. The vector x can either be expected as `numpy.ndarray` or simply as a Python `list`.
   a) Access the three CSR lists via `A.data`, `A.indptr`, `A.indices` and implement the matrix-vector product "by hand" using for loops.
   b) Implement the matrix-vector product using `A.dot(x)` or `A@x` .

Test your different routines on the matrix $A \in \mathbb{R}^{n\times n}$ given by

$$A = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

and a random input vector `x = numpy.random.rand(n)`. Play around with the dimension n (especially large $n \geq 10^5$).

For all cases:

- **Memory:** A number implemented as `float` in Python implements double precision and therefore needs `64` Bits of storage. What is the number of Gbytes needed to store an $m \times n$ array of `floats`? Print the number of Gbytes which are needed to store the matrix in all cases. For a `numpy.ndarray` you can type `A.nbytes` and for the `scipy.sparse.csr_matrix` you can type `A.data.nbytes + A.indptr.nbytes + A.indices.nbytes`.
- **Computation times:** Measure the time which is needed in each case to compute the matrix-vector product for a random input vector `x = numpy.random.rand(n)`. In the IPython shell you can simply use the *magic function* `%timeit` to measure the time for a certain operation. For example, you can type `%timeit pythonfunction(x)`. Alternatively you can use the package `timeit`.

Solution:

```
import numpy as np
import scipy.sparse as scs
import timeit

def matvec_dense(A, x, byhand=0):
```

```
    """
    computes the matrix vector product based on numpy.ndarray

    Parameters
    ----------
    A : (m,n) numpy.ndarray
       matrix
    x : (n, ) numpy.ndarray
       vector

    Returns
    -------
        A*x: matrix-vector product
    """
    if byhand:
        # read the dimensions of the input objects
        m, n = np.shape(A)
        n2 = len(x)

        # raise an error if the dimensions do not match
        if n != n2:
            raise Exception('dimension of A and x must match. The dimension for A and x were:
    {}'.format(str(np.shape(A)) +" "+ str( len(x))))

        # if dimensions match, start computing the matrix-vector product:
        else:
            # initialize the output vector
            b = np.zeros(m)
            # a loop over row indices to compute each entry of b
            for i in range(n):
                # a loop over column indices to compute the inner product
                for j in range(n):
                    b[i] += A[i,j]*x[j]
    else:
        b =A .dot(x)
    return b

# we could implement our own csr-class in python;:
#class csr_matrix:
#    def __init__(self, data, indices, indptr):
#        self.data = data
#        self.indices = indices
#        self.indptr = indptr

def matvec_sparse(A, x, byhand=0):
    """computes the matrix vector product based on numpy.ndarray

    Parameters
    ----------
    A: (m,n) matrix  stored in CSR, i.e., in terms of three lists; here:
       class with attributes data, indices, indptr
    x: (n, ) numpy.ndarray or list of length n (= number of cols) numbers
       vector

    Returns
    -------
        A*x: matrix-vector product
    """
    if byhand:
        # dimension check?
        # can we get the column dimension from sparse csr class? > depends
        b = [0] * (len(A.indptr) - 1)
```

```
        for i, pair in enumerate(zip(A.indptr[0:-1], A.indptr[1:])):
            for a_ij, j in zip(A.data[pair[0]:pair[1]], A.indices[pair[0]:pair[1]]):
                b[i] += a_ij * x[j]
    else:
        # make sure A and x have the correct format for the @ operator
        A = scs.csr_matrix(A)
        x = np.array(x)
        # compute matrix-vector product
        b=A.dot(x)
    return np.array(b)


print("\nIn order to get the docstring of our function we can type  \
      \n\n     help(functionName)\n\nFor example: ")
print(help(matvec_dense))


if __name__ == "__main__":
    # Note: the following part is only executed if the current script is
    #       run directly, but not if it is imported into another script

    #-----------------------------------------------------------------#
    #    EXPERIMENT
    #-----------------------------------------------------------------#
    # the experiment
    n = int(1e4)
    m = n
    runs = 1
    x = np.random.rand(n)

    # test arrays for which we know the result
    xtest = np.ones(n)
    btest = np.zeros(m)
    btest[[0,-1]] = 1

    # just some strings for printing commands
    expstr  = ["Time dot:     ",
               "Time hand:    "]
    teststr = ["Test dot:     ",
               "Test by hand: "]

    #-----------------------------------------------------------------#
    #    NUMPY DENSE
    #-----------------------------------------------------------------#
    print("\n---- Numpy Dense ----")
    A = 2 * np.eye(n) - np.eye(n , k=1) - np.eye(n , k=-1)
    print("Memory:", np.round(A.nbytes * 10**-9, decimals=4), "Gbytes\n")
    for byhand in [0,1]:
        print(teststr[byhand],np.allclose(btest,matvec_dense(A, xtest, byhand=byhand)))
        def dense():
            return matvec_dense(A, x, byhand=byhand)
        print(expstr[byhand], timeit.timeit("dense()", setup="from __main__ import dense",
    number=runs), "\n")

    #-----------------------------------------------------------------#
    #   SCIPY SPARSE
    #-----------------------------------------------------------------#
    print("\n---- Scipy Sparse ----")
    A = 2*scs.eye(n,k=0) - scs.eye(n , k=1) - scs.eye(n , k=-1)
    print("Memory:", np.round((A.data.nbytes + A.indptr.nbytes + A.indices.nbytes)* 10**-9,
    decimals=4), "Gbytes\n")
    for byhand in [0,1]:
```

```
        print(teststr[byhand],np.allclose(btest,matvec_sparse(A, xtest, byhand=byhand)))
        def sparse():
            return matvec_sparse(A, x, byhand=byhand)
        print(expstr[byhand], timeit.timeit("sparse()", setup="from __main__ import sparse",
    number=runs), "\n")
```