```
from scipy.sparse.linalg import LinearOperator, gmres, cg
import numpy as np


class CSRMatrix(LinearOperator):
        def __init__(self, coo_matrix):
            self.shape = coo_matrix.shape
            self.dtype = coo_matrix.dtype
            self.data = coo_matrix.data
            self.indices = coo_matrix.col
            rows, cols = self.shape
            self_row = np.zeros(rows + 1, dtype=int)
            for i in coo_matrix.row:
                self_row[i+1] += 1
            for j in range(rows):
                self_row[j+1] = self_row[j] + self_row[j+1]
            self.indptr = self_row

        def __add__(self, other):
            rows, cols = self.shape
            answer = np.zeros(shape=(1,2))
            ans_indptr = np.zeros(rows+1)
            for i in range(rows):
                col_1 = self.indices[self.indptr[i]: self.indptr[i+1]]
                col_2 = other.indices[other.indptr[i]: other.indptr[i+1]]
                col1 = np.matrix(col_1)
                col2 = np.matrix(col_2)
                data1 = np.matrix(self.data[self.indptr[i]: self.indptr[i+1]])
                data2 = np.matrix(other.data[other.indptr[i]: other.indptr[i+1]])
                matrix1, matrix2 = np.r_[col1,data1].T, np.r_[col2,data2].T
                counter = 0
                for j in col_1:
                    index = np.where(col_2==j)[0]
                    if len(index) == 1:
                        matrix2[index[0],1] += matrix1[counter,1]
                    else:
                        matrix2 = np.r_[matrix2, matrix1[counter]]
                        ans_indptr[i+1:] += 1
                    counter += 1
                matrix2 = np.array(list(sorted(dict(matrix2.tolist()).items())))
                answer = np.r_[answer, matrix2]
            answer.astype(np.int32)
            other.data = np.array(answer[1:,1]).T
            other.indices = np.array(answer[1:,0]).T.astype(int)
            other.indptr = (ans_indptr + other.indptr).astype(int)
            return other
```

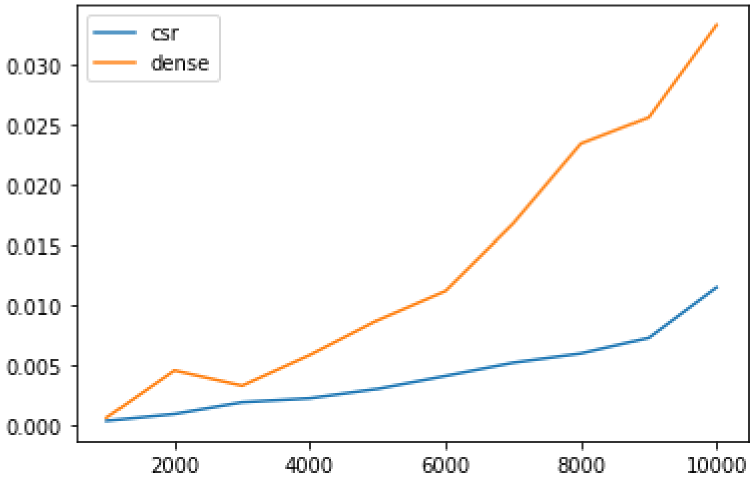
```
    def _matvec(self, vector):
        rows, cols = self.shape
        data = self.data
        indptr = self.indptr
        indices = self.indices
        answer = np.zeros(rows, dtype=np.float64)
        for i in range(rows):
            index_start = indptr[i]
            index_end = indptr[i+1]
            for j in range(index_start, index_end):
                answer[i] += data[j] * vector[indices[j]]
        return answer
```

```
from scipy.sparse import coo_matrix, rand, csr_matrix

matrix_test = np.random.rand(100,100)
matrix_other = np.random.rand(100,100)
vector_test = np.random.rand(100).T

mat_T = CSRMatrix(coo_matrix(matrix_test))
mat_O = CSRMatrix(coo_matrix(matrix_other))
actual_add = csr_matrix(matrix_test + matrix_other)
answer_add = mat_T.__add__(mat_O)
actual_vec = matrix_test@vector_test
answer_vec = mat_T._matvec(vector_test)
assert (actual_add.indptr == answer_add.indptr).all()
assert np.allclose(actual_add.data, answer_add.data)
assert (actual_add.indptr == answer_add.indptr).all()
assert np.allclose(actual_vec, answer_vec)
```

```
import time
from matplotlib import pyplot as plt
N = np.linspace(1000,10000,10,dtype=int)
T_csr, T_dense = np.zeros(10), np.zeros(10)
for i in range(10):
    n = N[i]
    mat_t = rand(n,n,0.0001)
    vector = np.random.rand(n)
    mat_den = mat_t.todense()
    mat_coo = coo_matrix(mat_den)
    mat_csr = CSRMatrix(mat_coo)
    start = time.time()
    answer_vec = mat_csr._matvec(vector)
    end = time.time()
    T_csr[i] = end - start
    start = time.time()
    answer_vec1 = mat_den@vector
    end = time.time()
```

```
    T_dense[i] = end - start

plt.plot(N, T_csr, label='csr')
plt.plot(N, T_dense, label='dense')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f5dfcddde90>
```



Answer: When matrix size increases and huge enough, matvec costs much less time than numpy. And the time numpy costs grows exponentially, but matvec does not.

```
dia = np.random.rand(100)
vec = np.random.rand(100).T
mat_c = coo_matrix(np.diag(dia))
mat1 = CSRMatrix(mat_c)
x_gm, exitCode_gm = gmres(mat1, vec)
x_cg, exitCode_cg = cg(mat1, vec)
print('When trol=1.e-3.', np.allclose(x_gm, x_cg, rtol=1.e-3))
print('When trol=1.e-5.', np.allclose(x_gm, x_cg, rtol=1.e-5))
```

```
When trol=1.e-3. True
When trol=1.e-5. False
```

Answer: The solutions are nearly but not exactly the same. The reason is that both methods converge but with different speed. And GMRES minimizes residual (b - Ax_k), where CG minimizes error (x - x_k), which leads to different levels of accuracy.

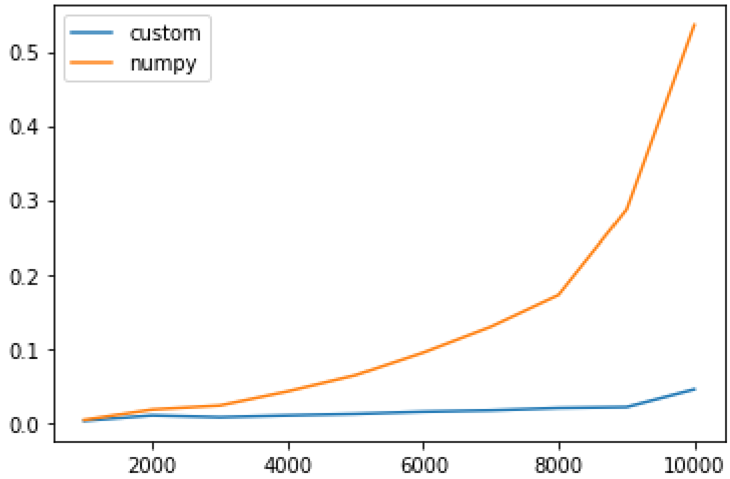
```
from scipy.sparse.linalg import LinearOperator, gmres, cg
import numpy as np


class CUSMatrix(LinearOperator):
```

```
    def __init__(self, diag, T, W):
        n = len(diag)
        self.shape = 2*n, 2*n
        self.diag = diag
        self.t = T
        self.w = W

    def _matvec(self, vector):
        diag = self.diag
        T = self.t
        W = self.w
        n = len(diag)
        ans = np.zeros(2*n)
        mat_Wv = np.zeros(2)
        for i in range(n):
            ans[i] = diag[i]*vector[i]
            mat_Wv[0] += W[0,i]*vector[n+i]
            mat_Wv[1] += W[1,i]*vector[n+i]
        for j in range(n):
            ans[n+j] += T[j,0]*mat_Wv[0] + T[j,1]*mat_Wv[1]
        return ans
```

```
import time
from matplotlib import pyplot as plt



N = np.linspace(1000,10000,10,dtype=int)
T_cust, T_np = np.zeros(10), np.zeros(10)
for i in range(10):
    n = N[i]
    mat_zero = np.zeros(shape=(n,n))
    diag = np.random.rand(n)
    T = np.random.rand(n,2)
    W = np.random.rand(2,n)
    Vec = np.random.rand(2*n)
    mat_tl = np.diag(diag)
    mat_br = T@W
    mat_t = np.append(mat_tl,mat_zero,axis=1)
    mat_b = np.append(mat_zero,mat_br,axis=1)
    A = np.append(mat_t,mat_b,axis=0)
    A_cust = CUSMatrix(diag, T, W)
    start = time.time()
    answer_cust = A_cust._matvec(Vec)
    end = time.time()
    T_cust[i] = end - start
    start = time.time()
    answer_np = A@Vec
    end = time.time()
    T_np[i] = end - start
```

```
plt.plot(N, T_cust, label='custom')
plt.plot(N, T_np, label='numpy')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fd395b14a10>
```



Answer: When matrix size increases and huge enough, custom matvec costs much less time than numpy, and the time numpy costs grows exponentially, where the time for custom matvec grows quite slow. The reason is that the time complexity of custom matvec is only O(n), whereas the time complexity of numpy @ is O(n^2), which means custom matvec will be faster than numpy.