After a heated debate with a fellow student about whether Python has actually failed as a general language and is only sold to scientists through all the libraries and whether we should actually be learning a mathematical language, I decided to give old lady c++ a try .

CG and steepest Descent in C++

```
VectorXd steepestDescent(MatrixXd A, VectorXd b, VectorXd x, int maxTolerance)
{
    double tolerance = 1e-6;
    for(int i=0; i<maxTolerance; i++)
    {
        VectorXd r = b - A * x;
        double r_norm = r.norm();

        if (r_norm < tolerance)
        {

            break;
        }
        VectorXd d = r;
        double alpha = r.dot(r) / (d.transpose() * (A * d));
        x = x + alpha * d;
    }
    return x;
}
```

```cpp
void conjugateGradient(MatrixXd A, VectorXd b, VectorXd &x, int maxIterations){
    double tolerance = 1e-6;
    // r0 = b-A*x0
    VectorXd r0 = b - A * x;
    //if r0 is sufficiently small, then return x0 as the result···
    VectorXd rk = r0;
    //p0 = r0
    VectorXd p0 = r0;
    //pk = p0 in the first iteration
    VectorXd pk = p0;
    //Not 100% sure why, but I cant do 1/ (pk * A * pk), so I will use this variable Ap to work around
    VectorXd Ap(b.size());
    //So they dont get initializes every iteration
    double alphak, betak, scalar;

    // repeat for k=0,1,2,....
    for (int i = 0; i < maxIterations; i++) {
        //a_k = r_k^T * r_k / ((p_k)^T * A * p_k)
        scalar = rk.dot(rk); // with scalar = r_k^T * r_k
        Ap = A*pk;
        alphak = scalar / pk.dot(Ap);
        //x_k+1 := x_k + a_k * p_k
        x = x + alphak * pk;
        //r_k+1 = r_k - a_k * A*p
        rk = rk - alphak * Ap;
        //if r_k^T * r_k is too small, stop
        if (sqrt(scalar) < tolerance) {
            break;}
        //beta_k = rk+1^T * rk+1 / (rk^T*rk)
        betak = rk.dot(rk) / scalar;
        //p_k+1 = r_k+1  + beta_k * p_k
        pk = rk + betak*pk;
    }
}
```

Applying the method to the test system

```
// Apply your method to the following test system
MatrixXd A(4, 4);
A << 7.0, 3.0, -1, 2,
    3,8,1,-4
    ,-1,1,4,-1
    ,2,-4,-1,6;
VectorXd b(4);
b << 1, 2,3,4;
VectorXd x(4);
x << 0,0,0,0;
conjugateGradient(A, b, x, 1000);
// Print the result
cout << "Solution vector for conjugate Gradient: \n" << x << "\n";
//Proof
cout << "A*solution should be 1,2,3,4: \n" << A*x << "\n";
A << 7.0, 3.0, -1, 2,
    3,8,1,-4
    ,-1,1,4,-1
    ,2,-4,-1,6;

b << 1, 2,3,4;
x << 0,0,0,0;
x = steepestDescent(A,b,x, 1000);
cout << "Solution vector for steepest Descent:  \n " << x << "\n";
cout << "A*solution should be 1,2,3,4: \n" << A*x << "\n";
```

Output:

```
Solution vector for conjugate Gradient:
-1.27619
 1.87619
0.571429
  2.4381
A*solution should be 1,2,3,4:
1
2
3
4
Solution vector for steepest Descent:
 -1.27619
 1.87619
0.571429
 2.43809
A*solution should be 1,2,3,4:
1
2
3
4
```

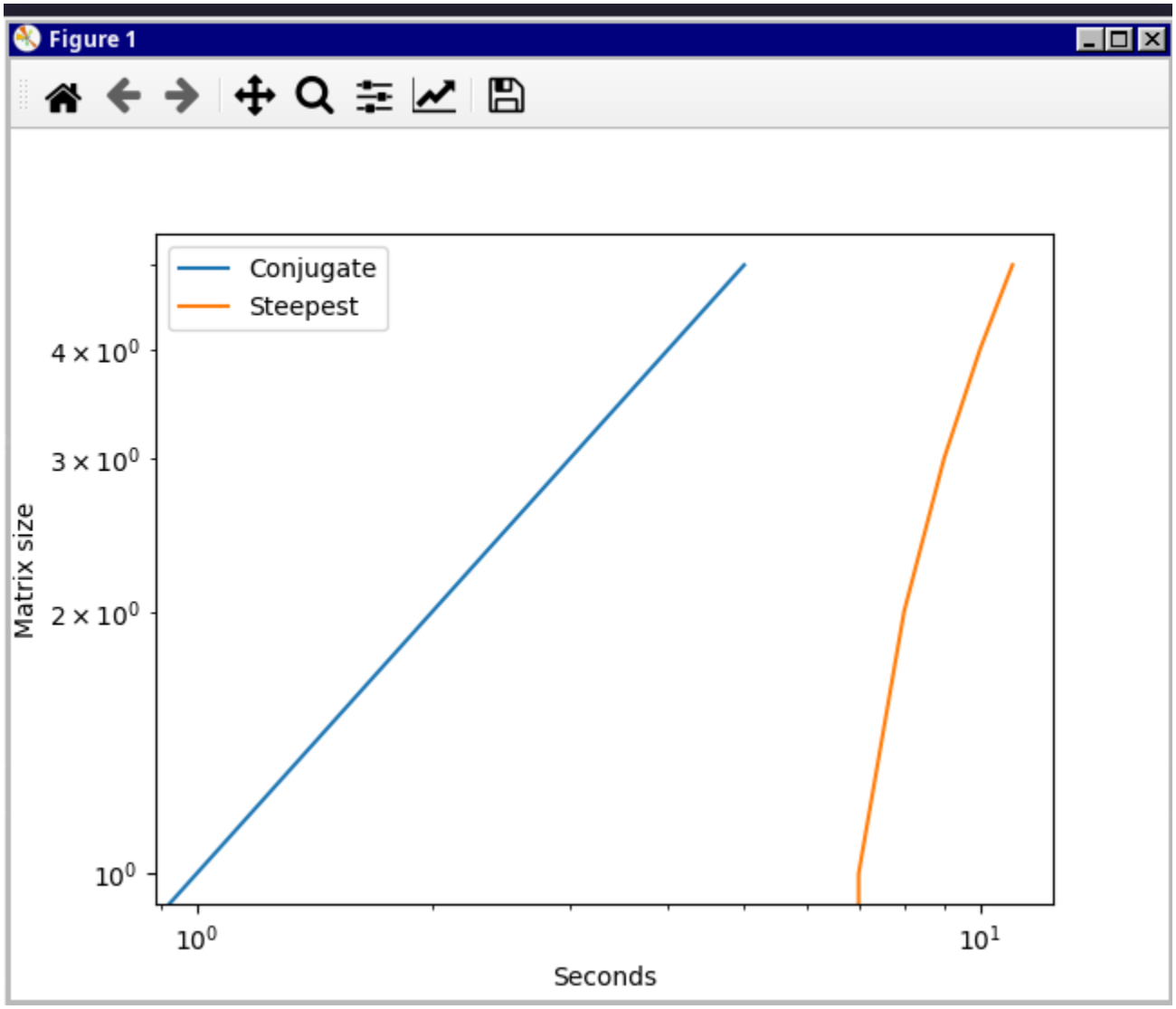
Perform benchmarks for matrices of different sizes

```cpp
int arr[6] = {4,10,20,30, 50, 70};
double averageTimeNeededConjugate[6];
double averageTimeNeededSteepest[6];
//For each of the sizes, solve 10 random systems with both
//Then calculate the average
for (int i = 0; i < 6; i++)
{
    double LocalTimeNeededConjugate[10];
    double LocalTimeNeededSteepest[10];
    for (int j = 0; j < 10; j++)
    {
        //Random A, random B
        MatrixXd randMatrix = generateRandomMatrixForDimension(arr[i]);
        VectorXd randVector = generateRandomVectorForDimension(arr[i]);
        //creates vector 0 for n dimension
        VectorXd v = VectorXd::Zero(arr[i]);
        auto start = chrono::high_resolution_clock::now();

        conjugateGradient(randMatrix, randVector, v, 1000);
        auto stop = chrono::high_resolution_clock::now();

        auto difference = stop - start;
        LocalTimeNeededConjugate[j] = std::chrono::duration<double>(difference).count();
        start = chrono::high_resolution_clock::now();

        VectorXd solution = steepestDescent(randMatrix, randVector, v, 1000);
        stop = chrono::high_resolution_clock::now();
        difference = stop - start;
        LocalTimeNeededSteepest[j] = std::chrono::duration<double>(difference).count();
```

With helper functions to generate random matrices and vectors.

Visualize the benchmark

Since the last time I updated a bunch of linux stuff my Gnu plot is not working anymore, so I just wrote the average time for sizes into a csv file and plotted it in python:



Comparisation with LU solvers

It's hard to compare since I coded LU in python and this exercise in C++, but I think I can say with some information from the lectures that LU might be better for small matrices. The conjugate gradient method does not converge as quick as LU for poorly conditioned matrices.

Same goes for the steepest decent iteration, just that it is not as performant as the conjugate gradient method.

Are there any linear systems which cannot be solved using the algorithm?

The methods are not made for matrices that are positive-definite.