Solving 2D Laplace equation using Jacobi Iteration method

The goal of this assignment was to implement a program for solving the two-dimensional Laplace equation with an iterative Jacobi method. For this, we provide three steps:
A first draft of the program, following the instructions of the problem decription. Next, we changed the code slightly to use dynamic memory allocation for variable problem sizes.
In the last step, we improve the code to achieve optimization, that means a faster execution, while obtaining the same results.

1 First draft

1.1 Header and Definitions

Outside of the main program, the used .h-files are included. In this version of the program, not only the constant variable π, but also the static sizes of the matrix are defined as global variables.
All used parameters are defined and initialized, the matrices are declared as static, i.e., their size is known at compile-time.
Finally, the user is asked to provide the number of iterations of the Jacobi scheme.

```
#include<stdio.h>
#include<math.h>

#define PI 3.14159265358979
#define n 7
#define m 7

int main(int argc, char** argv){

double global_error = 1;
double error = 0;
double diff;
double tol = 0.01;
int iter_max = 0;
int iter = 0;

static float A[n][m];
static float Anew[n][m];


printf("Number of iterations: ");
scanf("%d", &iter_max);
```

1.2 Initialization

Before the first iteration, the values of the matrices are initialised, following specific boundary conditions.

```
for(int j = 0; j < m; j++){
  A[0][j] = 0;
  A[n - 1][j] = 0;
  }
for(int i = 0; i < n; i++){
  A[i][0] = sin(i * PI / (n-1));
  A[i][m - 1] = sin(i * PI / (n-1)) * exp(-PI);
  }
```

1.3 Main loops

As long as the maximal number of iterations is not reached and the error is bigger than a tolerance value, a while loop is performed.
In a double for loop the new values for the matrix Anew are calculated, according to a 5-point-stencil scheme. Another double for loop calculates the error of the recent iteration step.
Finally, in a third double for loop, the old values of A are overwritten by the new values of Anew.

```
while(iter < iter_max && global_error > tol){
  iter++;
  for(int i = 1; i < n - 1; i++){
    for(int j = 1; j < m - 1; j++){
      Anew[i][j] = (A[i-1][j] + A[i+1][j]
        + A[i][j-1] + A[i][j+1]) / 4.;
    }
  }

  //calculate maximum error values
  for(int i = 1; i < n - 1; i++){
    for(int j = 1; j < m - 1; j++){}
      error_h=fmaxf(error_h,sqrtf(fabsf(Anew[i][j]-A[i][j])));
    }
  }
  if(error_h < global_error){
    global_error = error_h;
    error_h = 0;
  }
  if(iter%10 == 0){
    printf("Error after %d iterations:
      %f\n", iter, global_error);
  }


  for(int i = 1; i < n - 1; i++){
    for(int j = 1; j < m - 1; j++){
      A[i][j] = Anew[i][j];
    }
  }
}
```

1.4 Print Matrix

```
//print matrix
for(int i = 0; i < n; i++){
  for(int j = 0; j < m; j++){
    printf("%f ", A[i][j]);
    if(j == m - 1){
      printf("\n");
    }
  }
}
}
```

2 Dynamic allocated memory

In this improved version of the program, the user can provide the size of the matrices. Since this happens at run time, it is not possible anymore to allocate the memory for the matrices already at compile time.
The only changes here concern the Initialization and a necessary freeing of memory at the end of the program. All remaining commands can be used as in the first draft, for example the access of the matrix via A[ ][ ] is the same for static and allocated arrays.

2.1 Initialization

Here, the user is asked for the dimensions of the matrices. For the memory allocation, we use calloc(), which takes the number of arguments and the size in byte of every single element. In contrast to malloc(), calloc() initializes all values to zero. First, we allocate the rows of the matrix, in the next step, we have to go through all rows and also allocate the single elements in every row.

```
...
int main(int argc, char** argv){

...

int m,n;

printf("Please provide the number of rows: ");
scanf("%d",&n);
printf("...and the number of columns: ");
scanf("%d",&m);


float **A = (float**) calloc(n,sizeof(float*));
float **Anew = (float**) calloc(n, sizeof(float*));
//*(A+1) = 2 //A[1] = 2
for(int i = 0; i < n; i++){
  A[i] = calloc(m,sizeof(int));
  Anew[i] = calloc(m,sizeof(int));
}
...
```

2.2 Freeing of memory

Manual allocation of memory always requires the freeing after its use. For the deallocation, we use free and proceed reverse to the allocation.

```
for(int i = 0; i < m; i++){
  free(A[i]);
  free(Anew[i]);
}
free(A);
free(Anew);
```

3 Optimization

In the optimized version, the most changes affect the main loop, since this is the part which needs the vast majority of the execution time. There are 5 main changes that improve the speed of the code:

1. We join the first and the third double loop into one double loop
2. C stores it values in row-major order, so we should follow this scheme in our loops to have a better performance due to reduced jumps in memory access. So, a change of the loop order should improve speed
3. Division is slower than multiplication, so we can change the division by 4, which has to be applied $m \cdot n$ times, by multiplication with 0.25
4. Since the square root is a monotonous function, the max of the root equals the root of the max, which allows us to put the runtime-costly sqrtf( ) function outside the loops
5. Double buffer: instead of always calculating new values for Anew with the recent values from A and then copying the results back to A, we can go back and forth in every step

3.1 Main loops

```
while(iter < iter_max && global_error > tol){
  iter++;
  error = 0;
  for(int j = 1; j < m - 1; j++){
    for(int i = 1; i < n - 1; i++){
      if(iter %2 ==0){
        Anew[i][j] = (A[i-1][j] + A[i+1][j]+ A[i][j-1] + A[i][j+1])
            *0.25;
        error=maxf(error,fabsf(Anew[i][j]-A[i][j]));
      }
      else{
        A[i][j] = (Anew[i-1][j] + Anew[i+1][j]+ Anew[i][j-1] + Anew[i
            ][j+1]) *0.25;
        error=fmaxf(error,fabsf(Anew[i][j]-A[i][j]));
      }
    }
```

```
  }
}
  error=sqrtf(error);

  if(iter%10 == 0){
    printf("Error after %d iterations:
      %f\n", iter, global_error);
  }


}// end main
```

4 Results

During tests or the runtime and under usage of optimization commands of the gcc compiler, we found that the improvements listed in the last section can improve (for big problem dimensions) the speed of the program over a considerable magnitude.