Georg-August-Universität Göttingen
Institut für Theoretische Physik
Prof. Dr. F. Heidrich-Meisner
Dr. E. Bothmann
S. L. Villani
SoSe 2021

Übungen zur Vorlesung
Computergestütztes Wissenschaftliches Rechnen
Blatt 8

Lernziele dieses Übungsblattes

- Implementing the Gauss elimination algorithm.
- Use the Gauss algorithm to find the determinant and the inverse of a matrix.
- Build a routine to study and find the eigenvalues of a tridiagonal matrix, in this case using specific strategies for this particular class of matrices.

Aufgabenmodus

Die Aufgaben dieser Woche verfügen über keine gesonderten Tutorials. Sie sollten alle Aufgaben mit Ihren bisherigen Kenntnissen bearbeiten können. Dafür fallen die Aufgabenstellungen stellenweise etwas feinschrittiger aus.

Diese Woche sind die Aufgaben noch einmal auf Englisch.

Prüfungsvorleistung

Die Bearbeitung von Aufgabe 21 ist die dritte von vier unbenoteten Prüfungsvorleistungen. Stellen Sie Ihre Bearbeitung bis zum 13. Juni, 12:00 Uhr über Gitlab zur Verfügung.

Ihr/e Tutor/in wird sich dann mit Ihnen in Kontakt setzen und ggf. notwendige Anpassungen mit Ihnen besprechen.

Übungsaufgaben

Aufgabe 20 *Gauss elimination method*

Linear algebra is a fundamental tool for a physicist more often that not problems cannot just be solved by hand and a numerical strategy is needed. In this exercise you are going to implement your own Gauss elimination algorithm and use it to find the determinant and the inverse of a general matrix. The algorithm consist naively in visualizing the matrix as a system of linear equations with which we can perform three operations, i.e. swapping rows, multiplying a row by a non-zero number, and adding (subtracting) a multiple of a row to (from) another one. Using these operations, it is possible to transform a matrix into a triangular matrix, either upper or lower. Once in this form, one can directly get

the determinant of the original matrix by multiplying all the diagonal elements of this triangular matrix (in this case one must be careful, because each time a row is swapped the determinant gets a minus sign), solve the system of linear equations and calculate the inverse of the matrix.

Let's take **A** to be a $n \times m$ matrix, then the algorithm in pseudo code is:

```
for(i = 0; i < n-1; i++)
  for(k = i+1; k < n; k++)
    c = A[k*m+i] / A[i*m+i]
    for (j = 0; j < m; j++)
      A[k*m+j] = A[k*m+j] - c*A[i*m+j]
```

In the code the matrix **A** has been mapped to a vector of dimension $n \cdot m$ using $(i, j) \rightarrow (i \cdot m + j)$, as discussed in the lecture.

You can see that in this procedure we have divided by the diagonal elements. This may cause problems if a diagonal element is zero or even just very small. To fix this we can reorganize the rows of the matrix in order to have the biggest possible values on the diagonal. This procedure is called partial pivoting:

```
for(k = i+1; k < n; k++)
  if(fabs(A[i*m+i]) < fabs(A[k*m+i]))
    for(j = 0; j < m; j++)
      temp = A[i*m+j]
      A[i*m+j] = A[k*m+j]
      A[k*m+j] = temp
```

As has been mentioned, this algorithm can be used to calculate the inverse of a matrix. To do so it's enough to solve N systems of linear equations of the kind $\mathbf{Ax}_j = \mathbf{b}_j$, where $x_{ij} = A^{-1}_{ij}$ and $b_{ij} = \delta_{ij}$ that is the identity matrix. Let's start to implement all of this.

1. *Gauss algorithm*: Add the Gauss elimination algorithm to your code library with and without partial pivoting, saving the information of how many times rows are swapped in order to apply the right sign to the denominator. Implement it as follows: `int gauss(int n, int m,double A[n*m], int pivoting)`. The type of the function is `int` such that you can take into account the number of times you swapped lines. This is needed only for the partial pivoting case because it needs to reorganize the lines according to the pivot found. To do so initialize a variable to 1, e.g. `int sign = 1` before applying the partial pivoting procedure. Then just multiply by -1 this variable each time the above `if` statement is satisfied. Return then this variable.

2. *Determinant*: Now add the function `double det(int n, double A[n*n], int pivoting)`. Copy the matrix into a new one to not destroy it when calling the Gauss algorithm. Apply the Gauss method to this new matrix saving its returned value, i.e. the denominator sign, into a local variable. Return the product of the new matrix diagonal elements by the sign taken from the Gauss routine.

3. *LGS*: To calculate the inverse you need to solve N systems of linear equations so we are now going to add this routine to our library as well. Add the function `void lgs_solve(int n, double A[n*n],double b[n], double x[n], int pivoting)`, where you have to create a new matrix `C[n*(n+1)]`, whose

last column is the vector `b[]`, so `C[i*(n+1)+n]=b[i]`, and the rest is `A[]`. Triangularize this new matrix using Gauss (in this case we don't need the return value, so you can just call the function without storing its return value) and use it to calculate the components of `x[]` with the following backward substitutions

$$x_i = \frac{1}{C_{ii}} \left(C_{iN+1} - \sum_{j=i+1}^{N} C_{ij} x_j \right), \tag{1}$$

where $i = n-1, n-2, \ldots, 1$ starting with $x_N = C_{N,N+1}/C_{NN}$. Remember that mathematical indices go from 1 to N while the indices of an array in `C` go from 0 to $N-1$.

4. *Inverse*: you can now use the function from the previous point to calculate the inverse matrix. Write the function

```
void inverse(int n, double A[n*n],
             double A_inverse[n*n])
{
   ...
}
```

where `A[]` is the matrix to be inverted and `A_inverse[]` will be used to store the inverse. In the scope of the function check first if the determinant is zero, in that case print an error message and return from the function. If it's not, initialize two vectors `b[n]` and `x[n]`, which you need to create and solve the related LGS. The array `b[]` is like described previously, i.e. $b_{ij} = \delta_{ij}$, and `x[]` is the LGS solution and will be one of the inverse matrix vector column. Use the solutions `x[]` at each iteration to populate the columns of `B[]`.

5. Now it is time to test all these algorithms. Use the 10×10 matrix given in the file `matrix.c` by including it as a header file, i.e. `#include "20_matrix.h"`. You can download the file from Stud.IP.

   Calculate the determinant and the inverse of **A** using both approaches. Compare the result with other software like `numpy` or `Mathematica`. For the inverse you may even just write a function to multiply it with **A** and check whether the result is the identity matrix or not.

Aufgabe 21 *Eigenvalue problem of a tridiagonal matrix* *(Prüfungsvorleistung)*

Hinweis: Beachten Sie bitte die Hinweise zur Abgabe auf der ersten Seite.

Although the tools made in the previous exercise are useful in general, many problems in physics can be formulated in a tridiagonal form, that means that one has to deal with those matrices whose non zero elements are only $A_{i,i-1}, A_{ii}, A_{i,i+1}$. Luckily this class of matrices has a dedicated set of algorithms that are generally faster than those implemented in the previous exercise. For example inverting a matrix using Gauss needs $\mathcal{O}(n^3)$ operations while a tridiagonal matrix can be inverted with only $\mathcal{O}(n)$ operations. In this exercise you are going to find the eigenvalues of a tight-binding matrix, i.e. $A_{i,i-1} = A_{ii,+1} = -t$, and $A_{ii} = w$. This particular matrix is both very common in physics and is a tridiagonal

Toeplitz matrix for which eigenvalues are known analytically, so you can compare your solution to the analytical one. The eigenvalues for a tridiagonal Toeplitz matrix are:

$$\lambda_\alpha = w + 2t\cos\left(\frac{\alpha\pi}{N+1}\right). \tag{2}$$

Numerically the eigenvalue problem is just a zero finding problem for the characteristic polynomial $p_N(\lambda) = \det(\mathbf{A} - \mathrm{I}\lambda)$, which for a tridiagonal matrix can be simply gotten from the recursive relation:

$$p_i(\lambda) = (w - \lambda)p_{i-1}(\lambda) - t^2 p_{i-2}(\lambda), \tag{3}$$

using $p_0(\lambda) = 1$ and $p_1(\lambda) = w - \lambda$.

Let's now implement all this:

1. As described during the lecture, for this kind of matrices is enough to initialize two vectors, one for the diagonal and the other for the off-diagonal terms. Let's name them **a** and **b**, where **a** contains the diagonal terms and then it has dimension n, and **b** contains the off-diagonal terms and has dimension $n - 1$. Create also another array, `lambda[N]` to store the results.

2. Write a function

   ```
   double p(int n, double a[n], double b[n],
            double lambda, int grade)
   {
       ...
   }
   ```

   to evaluate the characteristic polynomial using the recursive relation given previously. `grade` is the grade at which we want our polynomial. This is not needed at this point since it will just be equal `n` but is introduced for generality. See the optional part.

3. Evaluate the characteristic polynomial $p_n(\lambda)$ for the tight-binding matrix for $n = 10$ with $t = 1/2$ and $w = 1$, and write the results to a file. Plot it and identify graphically the eigenvalues and what their geometric multiplicity is.

4. Use the function `find_root`, that you should already have in your code library, to find all the eigenvalues numerically. With that function, it is possible to find one zero at a time so use it to find each eigenvalue starting from a point close to it such that the algorithm will converge to the desired eigenvalue.

5. Compare the numerical eigenvalues with the analytical ones.

6. *(Optional)* A fast and efficient way to find these eigenvalues without graphical hints is based on the following two properties

   i) All the roots of $p_N(\lambda)$ lie in the interval $[-||\mathbf{A}||, ||\mathbf{A}||]$, where:

   $$||\mathbf{A}|| = \max_i \left\{ \sum_{j=1}^{N} |A_{ij}| \right\}. \tag{4}$$

ii) The number of roots above a certain value λ_0, is given by the number of agreements of the signs of $p_j(\lambda_0)$ and $p_{j-1}(\lambda_0)$ for $j = 1, 2, \ldots, n$. When one polynomial p_j is zero, the sign of the previous one p_{j-1} is used instead.

Now one can use the bisection approach to search for eigenvalues.

Implement a function which counts how many eigenvalues are above a certain λ_0:

```
int count_eigenvalues(int n, double a[], double b[],
                      double lambda_0)
{
    ...
}
```

In this function you'll need to loop over the different grades j of $p_j(\lambda)$. That's why we introduced the argument '`int grade`' above.

You can now search for eigenvalues using the bisection method in a slightly different way than it's usually defined for root finding (e.g. in Wikipedia), i.e. instead of looking at the sign of a function you'll look at the number of eigenvalues in an interval. This can be done in several ways, so feel free to find your own. In the following, we discuss one possibility. Start by looking for the largest eigenvalue first. To do so you have to count the eigenvalues above the middle point of the interval you are in (the starting interval is $[-||\mathbf{A}||, ||\mathbf{A}||]$). If the counting gives 0 or the number of eigenvalues already found, that means that the eigenvalue is in the lower part of the interval, otherwise it is in the upper part. Shrink the boundary of the interval in order to exclude the part where the eigenvalue cannot be and repeat the procedure until the remaining interval size drops below the desired precision. At this point save the found value λ_i in `lambda[]` and start over again, setting the interval to $[-||\mathbf{A}||, \lambda_i]$, until all the eigenvalues have been found with the desired numerical uncertainty.

Selbsttest

- Was kann beim (naiven) Gaußschen Eliminationsverfahren schief gehen? Wie lässt sich dieses Problem beheben?
- Wie lässt sich die Determinante einer Dreiecksmatrix berechnen? Worauf muss man dabei achten, wenn die Dreiecksmatrix mit dem Gaußschen Eliminationsverfahren erstellt wurde, bei dem partielles Pivotieren zum Einsatz kam?
- Beschreiben Sie in Ihren eigenen Worten ein Verfahren zur Nullstellensuche im chrakteristischen Polynom einer tridiagonalen Matrix, Gl. (3).