Appendix A

Numerical methods

A.1 Numerical root solvers

The purpose of a root solver is to find a solution (a root) to the equation

$$f(x) = 0, \tag{A.1}$$

or in general to a multi-dimensional equation

$$\vec{f}(\vec{x}) = 0. \tag{A.2}$$

Numerical root solvers should be well known from the numerics courses and we will just review three simple root solvers here. Keep in mind that in any serious calculation it is usually best to use a well optimized and tested library function over a hand-coded root solver.

A.1.1 The Newton and secant methods

The Newton method is one of best known root solvers, however it is not guaranteed to converge. The key idea is to start from a guess x_0, linearize the equation around that guess

$$f(x_0) + (x - x_0)f'(x_0) = 0 \tag{A.3}$$

and solve this linearized equation to obtain a better estimate x_1. Iterating this procedure we obtain the **Newton method**:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{A.4}$$

If the derivative f' is not known analytically, as is the case in our shooting problems, we can estimate it from the difference of the last two points:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \tag{A.5}$$

Substituting this into the Newton method (A.4) we obtain the **secant method**:

$$x_{n+1} = x_n - (x_n - x_{n-1})\frac{f(x_n)}{f(x_n) - f(x_{n-1})}. \tag{A.6}$$

The Newton method can easily be generalized to higher dimensional equations, by defining the matrix of derivatives

$$A_{ij}(\vec{x}) = \frac{\partial f_i(\vec{x})}{\partial x_j} \tag{A.7}$$

to obtain the **higher dimensional Newton method**

$$\vec{x}_{n+1} = \vec{x}_n - A^{-1}\vec{f}(\vec{x}) \tag{A.8}$$

If the derivatives $A_{ij}(\vec{x})$ are not known analytically they can be estimated through finite differences:

$$A_{ij}(\vec{x}) = \frac{f_i(\vec{x} + h_j\vec{e}_j) - f_i(\vec{x})}{h_j} \qquad \text{with} \qquad h_j \approx x_j\sqrt{\varepsilon} \tag{A.9}$$

where ε is the machine precision (about 10^{-16} for double precision floating point numbers on most machines).

A.1.2 The bisection method and regula falsi

Both the bisection method and the regula falsi require two starting values x_0 and x_1 surrounding the root, with $f(x_0) < 0$ and $f(x_1) > 0$ so that under the assumption of a continuous function f there exists at least one root between x_0 and x_1.

The **bisection method** performs the following iteration

1. define a mid-point $x_m = (x_0 + x_1)/2$.
2. if $\mathrm{sign}f(x_m) = \mathrm{sign}f(x_0)$ replace $x_0 \leftarrow x_m$ otherwise replace $x_1 \leftarrow x_m$

until a root is found.

The **regula falsi** works in a similar fashion:

1. estimate the function f by a straight line from x_0 to x_1 and calculate the root of this linearized function: $x_2 = (f(x_0)x_1 - f(x_1)x_0)/(f(x_1) - f(x_0)$
2. if $\mathrm{sign}f(x_2) = \mathrm{sign}f(x_0)$ replace $x_0 \leftarrow x_2$ otherwise replace $x_1 \leftarrow x_2$

In contrast to the Newton method, both of these two methods will always find a root.

A.1.3 Optimizing a function

These root solvers can also be used for finding an extremum (minimum or maximum) of a function $f(\vec{x})$, by looking a root of

$$\nabla f(\vec{x}) = 0. \tag{A.10}$$

While this is efficient for one-dimensional problems, but better algorithms exist.

In the following discussion we assume, without loss of generality, that we want to minimize a function. The simplest algorithm for a multi-dimensional optimization is

steepest descent, which always looks for a minimum along the direction of steepest gradient: starting from an initial guess $\vec{x}_n$ a one-dimensional minimization is applied to determine the value of λ which minimizes

$$f(\vec{x}_n + \lambda \nabla f(\vec{x}_n)) \tag{A.11}$$

and then the next guess $\vec{x}_{n+1}$ is determined as

$$\vec{x}_{n+1} = \vec{x}_n + \lambda \nabla f(\vec{x}_n) \tag{A.12}$$

While this method is simple it can be very inefficient if the "landscape" of the function f resembles a long and narrow valley: the one-dimensional minimization will mainly improve the estimate transverse to the valley but takes a long time to traverse down the valley to the minimum. A better method is the **conjugate gradient** algorithm which approximates the function locally by a paraboloid and uses the minimum of this paraboloid as the next guess. This algorithm can find the minimuim of a long and narrow parabolic valley in one iteration! For this and other, even better, algorithms we recommend the use of **library functions**.

One final word of warning is that all of these minimizers will only find a **local minimum**. Whether this local minimum is also the global minimum can never be decided by purely numerically. A necessary but never sufficient check is thus to start the minimization not only from one initial guess but to try many initial points and check for consistency in the minimum found.

A.2 The Lanczos algorithm

Sparse matrices with only $O(N)$ non-zero elements are very common in scientific simulations. We have already encountered them in the winter semester when we discretized partial differential equations. Now we have reduced the transfer matrix of the Ising model to a sparse matrix product. We will later see that also the quantum mechanical Hamilton operators in lattice models are sparse.

The importance of sparsity becomes obvious when considering the cost of matrix operations as listed in table A.1. For large N the sparsity leads to memory and time savings of several orders of magnitude.

Here we will discuss the iterative calculation of a few of the extreme eigenvalues of a matrix by the Lanczos algorithm. Similar methods can be used to solve sparse linear systems of equations.

To motivate the Lanczos algorithms we will first take a look at the power method for a matrix A. Starting from a random initial vector u_1 we calculate the sequence

$$u_{n+1} = \frac{Au_n}{||Au_n||}, \tag{A.13}$$

which converges to the eigenvector of the largest eigenvalue of the matrix A. The Lanczos algorithm optimizes this crude power method.

Table A.1: Time and memory complexity for operations on sparse and dense $N \times N$ matrices

operation	time	memory
storage		
dense matrix	—	N^2
sparse matrix	—	$\mathrm{O}(N)$
matrix-vector multiplication		
dense matrix	$\mathrm{O}(N^2)$	$\mathrm{O}(N^2)$
sparse matrix	$\mathrm{O}(N)$	$\mathrm{O}(N)$
matrix-matrix multiplication		
dense matrix	$\mathrm{O}(N^{\ln 7/\ln 2})$	$\mathrm{O}(N^2)$
sparse matrix	$\mathrm{O}(N) \dots \mathrm{O}(N^2)$	$\mathrm{O}(N) \dots \mathrm{O}(N^2)$
all eigen values and vectors		
dense matrix	$\mathrm{O}(N^3)$	$\mathrm{O}(N^2)$
sparse matrix (iterative)	$\mathrm{O}(N^2)$	$\mathrm{O}(N^2)$
some eigen values and vectors		
dense matrix (iterative)	$\mathrm{O}(N^2)$	$\mathrm{O}(N^2)$
sparse matrix (iterative)	$\mathrm{O}(N)$	$\mathrm{O}(N)$

Lanczos iterations

The Lanczos algorithm builds a basis $\{v_1, v_2, \dots, v_M\}$ for the Krylov-subspace $K_M = span\{u_1, u_2, \dots, u_M\}$, which is constructed by M iterations of equation (A.13). This is done by the following iterations:

$$\beta_{n+1} v_{n+1} = A v_n - \alpha_n v_n - \beta_n v_{n-1}, \tag{A.14}$$

where

$$\alpha_n = v_n^\dagger A v_n, \qquad \beta_n = |v_n^\dagger A v_{n-1}|. \tag{A.15}$$

As the orthogonality condition

$$v_i^\dagger v_j = \delta_{ij} \tag{A.16}$$

does not determine the phases of the basis vectors, the β_i can be chosen to be real and positive. As can be seen, we only need to keep three vectors of size N in memory, which makes the Lanczos algorithm very efficient, when compared to dense matrix eigensolvers which require storage of order N^2.

In the Krylov basis the matrix A is tridiagonal

$$T^{(n)} := \begin{bmatrix} \alpha_1 & \beta_2 & 0 & \cdots & 0 \\ \beta_2 & \alpha_2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \beta_n \\ 0 & \cdots & 0 & \beta_n & \alpha_n \end{bmatrix}. \tag{A.17}$$

The eigenvalues $\{\tau_1, \dots, \tau_M\}$ of T are good approximations of the eigenvalues of A. The extreme eigenvalues converge very fast. Thus $M \ll N$ iterations are sufficient to obtain the extreme eigenvalues.

Eigenvectors

It is no problem to compute the eigenvectors of T. They are however given in the Krylov basis $\{v_1, v_2, \ldots, v_M\}$. To obtain the eigenvectors in the original basis we need to perform a basis transformation.

Due to memory constraints we usually do not store all the v_i, but only the last three vectors. To transform the eigenvector to the original basis we have to do the Lanczos iterations a second time. Starting from the same initial vector v_1 we construct the vectors v_i iteratively and perform the basis transformation as we go along.

Roundoff errors and ghosts

In exact arithmetic the vectors $\{v_i\}$ are orthogonal and the Lanczos iterations stop after at most $N-1$ steps. The eigenvalues of T are then the exact eigenvalues of A.

Roundoff errors in finite precision cause a loss of orthogonality. There are two ways to deal with that:

- Reorthogonalization of the vectors after every step. This requires storing all of the vectors $\{v_i\}$ and is memory intensive.
- Control of the effects of roundoff.

We will discuss the second solution as it is faster and needs less memory. The main effect of roundoff errors is that the matrix T contains extra spurious eigenvalues, called "ghosts". These ghosts are not real eigenvalues of A. However they converge towards real eigenvalues of A over time and increase their multiplicities.

A simple criterion distinguishes ghosts from real eigenvalues. Ghosts are caused by roundoff errors. Thus they do not depend on on the starting vector v_1. As a consequence these ghosts are also eigenvalues of the matrix $\tilde{T}$, which can be obtained from T by deleting the first row and column:

$$\tilde{T}^{(n)} := \begin{bmatrix} \alpha_2 & \beta_3 & 0 & \cdots & 0 \\ \beta_3 & \alpha_3 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \beta_n \\ 0 & \cdots & 0 & \beta_n & \alpha_n \end{bmatrix}. \tag{A.18}$$

From these arguments we derive the following heuristic criterion to distinguish ghosts from real eigenvalues:

- All multiple eigenvalues are real, but their multiplicities might be too large.
- All single eigenvalues of T which are *not* eigenvalues of $\tilde{T}$ are also real.

Numerically stable and efficient implementations of the Lanczos algorithm can be obtained from netlib or fromhttp://www.comp-phys.org/software/ietl/ .