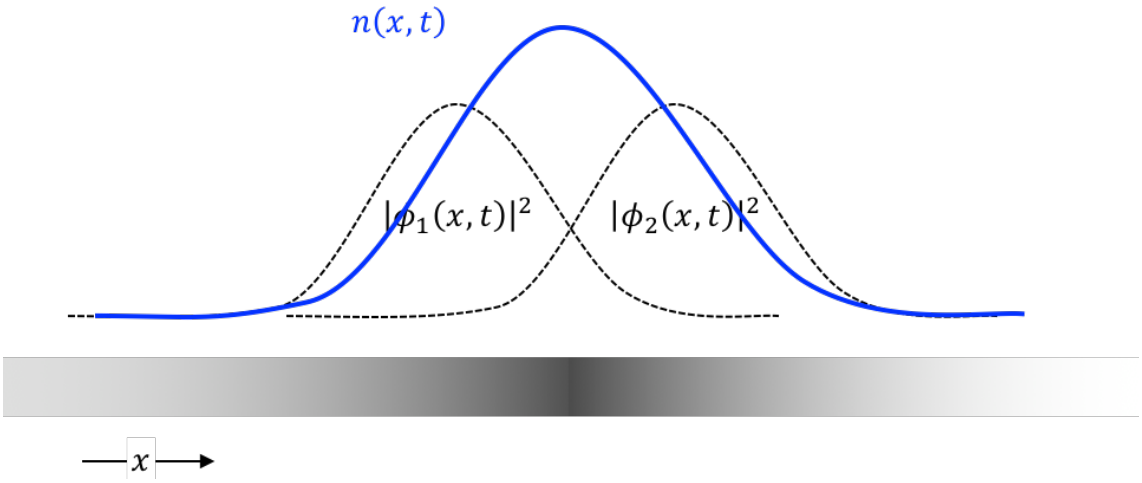
NUMERICAL TECHNIQUES FOR PHYSICISTS

Geert Brocks

Applied Physics, University of Twente

1st edition: November 2019



Contents

Part I

NUMERICAL BACKGROUND

Chapter 1

LINEAR AND NON-LINEAR PROBLEMS

Prologue. Numerical linear algebra is one of the more well-developed fields of numerical mathematics, powered by its countless uses in science and engineering. Its applications range from very applied civil engineering to very theoretical quantum field theory. Generations of scientists have worked on optimizing linear algebra algorithms, which has resulted in numerical libraries with efficient, robust, and fast numerical routines for solving linear problems.[1]

You will probably not write something yourself that can compete with such routines in speed or robustness. So if you encounter a physics problem that needs linear algebra, you are advised to look first for an appropriate routine in a standard library. Nevertheless, it is not very scientific to use such routines as black boxes only, without having a clue about how they actually work. Think inside of the box!! [Presents are inside boxes, outside is just hot air.] I will do that from a physicist's perspective, i.e., try and understand something of the underlying mathematics without going through all the ϵ's and δ's.[2]

Moreover, whereas a numerical mathematician's goal usually is to come up with a method that is applicable to as broad a class of problems as possible, a physicist does not care too much about broadness. In most cases, he/she wants to solve one particular problem. Knowing the insides of numerical methods then allows you to customize the right one for your particular application, and sometimes produce amazing speed-ups.

Linear problems. This chapter describes the simplest class of linear algebra problems, sets of linear equations, called the *linear problem* in the following. There are basically three approaches to solve linear problems $\mathbf{Ax} = \mathbf{b}$, with $\mathbf{A}$ an $N \times N$ matrix, $\mathbf{b}$ a known $N \times 1$ vector, and $\mathbf{x}$ an unknown $N \times 1$ vector. The *direct methods* solve the linear problem within a predetermined number of steps. That number increases with increasing N, and is usually proportional to N^3. For large N, the direct methods then become computationally too expensive, and one turns to *iterative methods.* These start with an approximate solution

[1] For instance, LAPACK, short for "linear algebra package", is the name of a popular library [1]. LAPACK routines are embedded in many commercial and open-source programs and packages. They have become the standard for solving linear algebra problems involving non-sparse matrices. For handling sparse matrices, i.e., matrices where most of the elements are zero, the variety in software packages is larger. Huge sparse matrix problems involve special numerical techniques, which are, for instance, implemented in the ARPACK library [2].

[2] If you want those, have a look at Golub & van Loan [3]. Very thorough, but rather tedious for a physicist.

$\mathbf{x}^{(0)}$, and iteratively improve the solution $\mathbf{x}^{(0)} \to \mathbf{x}^{(1)} \to ... \to \mathbf{x}^{(n)}$, in the hope that one converges to the solution $\mathbf{x}$ within a tolerably small number of iterations n.[3] That is far from guaranteed, which presents the main computational problem of these techniques. The methods have a long history, some of them even predating the invention of computers, and can be found in any textbook on numerical analysis. Some of the more accessible sources for physicists are Refs. [4, 5, 6, 7].[4]

The third way to solve a linear problem is by *subspace methods*. One starts again from an approximate solution $\mathbf{x}^{(0)}$, which one tries to improve iteratively, but now one keeps the information gathered during the iterations to generate subspaces of increasing dimension, $\mathbf{x}^{(0)} \to c_0^{(1)}\mathbf{x}^{(0)} + c_1^{(1)}\mathbf{x}^{(1)} \to ... \to c_0^{(n)}\mathbf{x}^{(0)} + c_1^{(n)}\mathbf{x}^{(1)} + ... + c_n^{(n)}\mathbf{x}^{(n)}$. In each cycle j, not only a new vector $\mathbf{x}^{(j)}$ is generated, but also the coefficients $c_i^{(j)}$; $i = 0, .., j$, are optimized to find the best possible solution within the $(j+1)$ dimensional subspace. Subspace methods are more complicated than either direct or simple iterative methods. For many large linear problems (large N), they represent the state of the art.[5]

Non-linear problems. The situation is more diffuse for non-linear problems, as there are many ways in which a problem can be non-linear.[6] The current methods for solving non-linear problems are intrinsically iterative. The usual strategy for solving a non-linear problem is to approximate it by a linear problem, solve that by one of the techniques discussed above, use the solution to find an improved linear approximation, solve that again, and repeat until further improvements are no longer useful.

I will discuss a number of standard and not-so-standard numerical algorithms to solve non-linear equations. Because of the vastness of the area I will be able to cover only a small subset of techniques. Chances are that, if you have a non-linear problem to solve, you have to do some shopping and testing before you will find the appropriate approach for your particular problem.

Contents of this chapter. I start by introducing the standard textbook direct method for solving linear problems: *LU decomposition*, also called *LU factorization*, in Sec. 1.1. I will not spend too many words on it, as the dos and don'ts of this method are described elaborately in many numerics textbooks [3]-[7]. *Cholesky decomposition*, discussed in Sec. 1.1.5, is a numerically very stable flavor of this type of algorithm, but suitable only for a special type of linear problem. A numerically more robust (but also computationally slightly more expensive) method suitable for all linear problems, *QR factorization*, also called *QR*

[3] IT people and computer programmers will call all methods I will describe, iterative. In numerical linear algebra, the word "iterative" has a more specific meaning. Applying an *iterative technique* is open-ended, in the sense that you cannot tell on forehand when (or sometimes even whether) the algorithm will decide to finish. In contrast, *direct techniques* will give their result after a predetermined number of computational steps. For safety, even for iterative techniques you put of course a maximum on the number of cycles you allow it to take. But you might not get convergence within this maximum number.

[4] Regarding Ref. [7], this book has its fans and foes. The foes criticize its lack of mathematical rigor, or its old-fashioned style of programming. The fans (I am one) welcome it as a book beyond kindergarten level accessible to non-specialists (read non-mathematicians), covering most of the topics in applied numerics in a pragmatic way. The "stylish" people are right; the programming style is largely obsolete.

[5] Many of them still have to find their way into textbooks suitable for physicists. A mathematical discussion can be found in Ref. [3].

[6] Calling a problem "non-linear" does not give away much information. It's like, upon asked "give me a number", you give the answer "it is not 42".

decomposition, is reviewed in Sec. 1.2. In its heart, QR factorization is a technique for constructing a set of orthonormal vectors (orthogonal and normalized), which also has applications outside solving linear problems. One such application, the construction of orthogonal functions, is discussed in Sec. 1.2.2.

Some of the standard iterative algorithms for solving linear and non-linear problems are discussed briefly in Sec. 1.3. Although generally applicable in principle, these algorithms often do not converge sufficiently fast to be practical for large-scale problems, or worse, they do not converge at all. In that case one tries to achieve convergence using subspace techniques. There are many of those around. In Sec. 1.4 I discuss one that is popular among physicists and chemists, Pulay iteration or DIIS.

The "starred" sections (*) can be omitted upon first reading. The pieces of text between horizontal lines are supplementary notes, containing mathematical details or proofs.[7]

Details: in paragraphs like these I give some mathematical background information, or a proof. They are useful (I think), but not vital. $\mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix}$ is a $N \times 1$ *column vector*, and $\mathbf{A} = \begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix}$ is a $N \times N$ *matrix*. $\mathbf{A}^\dagger$ is its *Hermitian conjugate*, with matrix elements $a^\dagger_{ij} = a^*_{ji}$. If $a^*_{ji} = a_{ij}$, the matrix $\mathbf{A}$ is called *Hermitian*. Hermitian matrices occur often in physics problems, so they have a special place. A Hermitian matrix has to be square, but the Hermitian conjugate is also defined for non-square matrices, including vectors.

So, $\mathbf{a}^\dagger = \begin{pmatrix} a^*_1 & \cdots & a^*_N \end{pmatrix}$ is a *row vector*, and the matrix product $\mathbf{a}^\dagger\mathbf{b} = a^*_1 b_1 + \cdots + a^*_N b_N$ is a *scalar*. We can use this to define the Hermitian *inner product* between two vectors $\mathbf{a}$ and $\mathbf{b}$ as $\mathbf{a}\cdot\mathbf{b} \equiv \mathbf{a}^\dagger\mathbf{b} = \sum_{i=1}^N a^*_i b_i$. Setting $\mathbf{b} = \mathbf{a}$ defines the *norm* of a vector $\|\mathbf{a}\| = \sqrt{\mathbf{a}^\dagger\mathbf{a}} = \sqrt{\sum_{i=1}^N |a_i|^2}$. This is called the "2-norm" or "euclidean norm" by mathematicians. Mathematicians employ a more general definition of "norm". Physicists mostly use the 2-norm, so if no confusion can arise, by "norm" I mean this one.

The *matrix norm* (2-norm) is defined as $\|\mathbf{A}\| = \max_{\mathbf{x}} [\|\mathbf{A}\mathbf{x}\| \,/\, \|\mathbf{x}\|]$. From this definition one can easily prove properties such as $\|\mathbf{A}\|\,\|\mathbf{B}\| \geq \|\mathbf{A}\mathbf{B}\|$.

In many cases it is useful to consider complex vectors and matrices (in quantum mechanics, or signal transformation, for instance). In other cases it is sufficient to stick to real vectors and matrices. The Hermitian conjugate of a real matrix $a^\dagger_{ij} = a^*_{ji} = a_{ji}$ is called its *transpose*, indicated by $\mathbf{A}^T$. So for real matrices $\mathbf{A}^\dagger = \mathbf{A}^T$. A real Hermitian matrix has $a_{ji} = a_{ij}$, and is called *symmetric*, $\mathbf{A}^T = \mathbf{A}$. The *inner product* between two real vectors is defined as as $\mathbf{a}\cdot\mathbf{b} \equiv \mathbf{a}^T\mathbf{b} = \sum_{i=1}^N a_i b_i$. Note that for real vectors $\mathbf{a}^T\mathbf{b} = \mathbf{b}^T\mathbf{a}$. Setting $\mathbf{b} = \mathbf{a}$ defines the *norm* of a real vector $\|\mathbf{a}\| = \sqrt{\mathbf{a}^T\mathbf{a}} = \sqrt{\sum_{i=1}^N a_i^2}$.

1.1 LU decomposition

Consider a set of linear equations represented by

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1.1}$$

[7] That is, my sorry versions. Proper versions can be found in the mathematical literature, such as Ref. [3]. I also skipped many proofs. As an old professor of mine used to say: "physics is a bit of mathematics and a bit of plumbing". In the end, the plumbing is most practical.

where $\mathbf{A}$ is a square $N \times N$ matrix, and $\mathbf{x}$ and $\mathbf{b}$ are $N \times 1$ column vectors. We assume that $\mathbf{b}$ is known, and $\mathbf{x}$ has to be calculated. If $\mathbf{A} = \mathbf{U}$ is an upper triangular matrix, $u_{ij} = 0;\ j < i$, then this set of equations is easily solved. Write out the matrix-vector product

$$\begin{aligned} u_{1,1}x_1 + u_{1,2}x_2 + ... + u_{1,N-1}x_{N-1} + u_{1,N}x_N &= b_1 \\ u_{2,2}x_2 + ... + u_{2,N-1}x_{N-1} + u_{2,N}x_N &= b_2 \\ \vdots &= \vdots \\ u_{N-1,N-1}x_{N-1} + u_{N-1,N}x_N &= b_{N-1} \\ u_{N,N}x_N &= b_N. \end{aligned} \tag{1.2}$$

From the N'th equation one obtains $x_N = b_N/u_{N,N}$. Substituting this in the $(N-1)$'th equation gives $x_{N-1} = \left(b_{N-1} - u_{N-1,N}x_N\right)/u_{N-1,N-1}$, etcetera. The algorithm is called *back substitution.*

Details: Note that none of the diagonal elements, $u_{i,i};\ i = 1, ..., N$ can be zero, otherwise the matrix $\mathbf{U}$ would be singular, and the linear problem has no solution in general. So it is safe to divide by a diagonal element. The easiest way to see this, is by $\det \mathbf{U} = \prod_{i=1}^{N} u_{ii}$; so if one or more of the diagonal elements are zero, then $\det \mathbf{U} = 0$, signaling a singular matrix.

If the matrix $\mathbf{A} = \mathbf{L}$ is lower triangular, $l_{ij} = 0;\ j > i$, then the matrix-vector product is

$$\begin{aligned} l_{1,1}x_1 &= b_1 \\ l_{2,1}x_1 + l_{2,2}x_2 &= b_2 \\ \vdots &= \vdots \\ l_{N-1,1}x_1 + l_{N-1,2}x_2 ... + l_{N-1,N-1}x_{N-1} &= b_{N-1} \\ l_{N,1}x_1 + l_{N,2}x_2 + ... + l_{N,N-1}x_{N-1} + l_{N,N}x_N &= b_N, \end{aligned} \tag{1.3}$$

and we use *forward substitution* to solve this problem. From the 1st equation one obtains $x_1 = b_1/l_{1,1}$. Substituting this in the 2nd equation gives $x_2 = \left(b_2 - l_{2,1}x_1\right)/l_{2,2}$, etcetera. For the same reason as discussed above, none of the diagonal elements $l_{i,i};\ i = 1, ..., N$ can be zero, so it is safe to divide by them.

If $\mathbf{A}$ is not triangular, then a general strategy to solve Eq. 1.1 is to first convert $\mathbf{A}$ into a triangular matrix, and then solve Eqs. 1.2 or 1.3. The conversion can be done by Gaussian elimination.[8] An inspired variation on this theme is called *LU decomposition* or *LU factorization* (or *Doolittle or Crout decomposition/factorization* after its inventors [4]).[9] It is based upon the following theorem.

Theorem: a square, non-singular, matrix $\mathbf{A}$ can be written as a product of a permutation matrix $\mathbf{P}$, a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$

$$\mathbf{A} = \mathbf{PLU}; \quad \text{with} \quad l_{ij} = 0,\ j > i;\ u_{ij} = 0,\ j < i, \quad \text{and} \quad l_{ii} = 1. \tag{1.4}$$

A permutation matrix $\mathbf{P}$ can be derived from the identity matrix $\mathbf{I}$ by permuting its rows (or columns). Operating on a vector $\mathbf{Px}$, it permutes the elements of $\mathbf{x}$. It is not difficult to

[8] Called "matrix vegen" in Dutch linear algebra classes.

[9] Mathematicians often give their algorithms short two-letter names like LU or QR. This can't be OK.

prove that $\mathbf{P}$ is an orthogonal matrix, so $\mathbf{P}^{-1} = \mathbf{P}^T$, and $\mathbf{P}\tilde{\mathbf{b}} = \mathbf{b}$ means $\tilde{\mathbf{b}} = \mathbf{P}^{-1}\mathbf{b} = \mathbf{P}^T\mathbf{b}$, see Sec. 2.4.1. Solving the linear problem $\mathbf{Ax} = \mathbf{PLUx} = \mathbf{b}$, Eqs. 1.1 and 2.38, is then done in thee steps. First write the problem as $\mathbf{LUx} = \tilde{\mathbf{b}}$. Defining $\mathbf{y} = \mathbf{Ux}$, the equation $\mathbf{Ly} = \tilde{\mathbf{b}}$ can be solved for $\mathbf{y}$ by forward substitution, Eq. 1.3. Subsequently, the equation $\mathbf{Ux} = \mathbf{y}$ can be solved by back substitution, Eq. 1.2.

The proof of Eq. 1.4 is based upon explicit construction of the matrices $\mathbf{P}$, $\mathbf{L}$ and $\mathbf{U}$, which is a bit tedious, however. In the next section I will demonstrate it for special matrices $\mathbf{A}$. The matrices we will encounter in the physical problems of Part II are of this special type.

1.1.1 LU decomposition of tridiagonal matrices

First of all, assume that $\mathbf{P} = \mathbf{I}$, which means that we don't have to permute anything, and $\tilde{\mathbf{b}} = \mathbf{b}$. This will work in many cases (although not in all). Furthermore, suppose that $\mathbf{A} = \mathbf{T}$ is a *tridiagonal matrix*, $t_{ij} = 0$, if $|i - j| > 1$. Tridiagonal matrices quite often occur naturally in many physical applications; they allow for efficient calculations (see also the lecture notes on eigenvalue problems)

$$\mathbf{T} = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & 0 \\ 0 & t_{32} & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \ddots & t_{N-1,N} \\ 0 & 0 & \cdots & t_{N,N-1} & t_{NN} \end{pmatrix} = \mathbf{LU}. \tag{1.5}$$

If a LU factorization of this matrix exists, then the lower and upper triangular matrices look like

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 & 0 \\ 0 & l_{32} & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \ddots & 0 \\ 0 & 0 & \cdots & l_{N,N-1} & 1 \end{pmatrix}; \quad \mathbf{U} = \begin{pmatrix} u_{11} & u_{12} & 0 & 0 & 0 \\ 0 & u_{22} & u_{23} & 0 & 0 \\ 0 & 0 & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \ddots & u_{N-1,N} \\ 0 & 0 & \cdots & 0 & u_{NN} \end{pmatrix}. \tag{1.6}$$

One can figure out the matrix elements l_{ij}, u_{ij} by writing out the matrix product $\mathbf{LU} = \mathbf{T}$. This gives $t_{i,i-1} = l_{i,i-1}u_{i-1,i-1}$, $t_{i,i} = l_{i,i-1}u_{i-1,i} + u_{i,i}$, $t_{i-1,i} = u_{i-1,i}$; $i = 2, ..., N$, and $t_{1,1} = u_{1,1}$. Inverting these relations gives

$$u_{11} = t_{11};\ \text{do } i = 2, ..., N\ \ l_{i,i-1} = \frac{t_{i,i-1}}{u_{i-1,i-1}};\ u_{i-1,i} = t_{i-1,i};\ u_{i,i} = t_{i,i} - l_{i,i-1}u_{i-1,i}. \tag{1.7}$$

In the tridiagonal case, the forward substitution $\mathbf{Ly} = \mathbf{b}$ is quite simple

$$y_1 = b_1;\ \text{do } i = 2, ..., N\ \ y_i = b_i - l_{i,i-1}y_{i-1}, \tag{1.8}$$

as is the back substitution $\mathbf{Ux} = \mathbf{y}$

$$x_N = \frac{y_N}{u_{NN}};\ \text{do } i = N, ..., 2\ \ x_{i-1} = \frac{y_{i-1} - u_{i-1,i}x_i}{u_{i-1,i-1}}. \tag{1.9}$$

Details: for the die-hards, who wish to see LU decomposition for a full $N \times N$ matrix $\mathbf{A}$, I can add some details. The matrix multiplication $\mathbf{LU} = \mathbf{A}$, with $\mathbf{L}$ a lower triangular matrix, $l_{ik} = 0;\ k > i,\ \ l_{ii} = 1$, and $\mathbf{U}$ an upper triangular matrix, $u_{kj} = 0;\ k > j$, gives $\sum_{k=1}^{N} l_{ik} u_{kj} = a_{ij}$, with constraints $k \leq i$ and $k \leq j$. This gives

$$\text{(a)} \sum_{k=1}^{j} l_{ik} u_{kj} = a_{ij}, \quad j < i, \quad \text{and} \quad \text{(b)} \sum_{k=1}^{i-1} l_{ik} u_{kj} + u_{ij} = a_{ij}, \quad j \geq i, \tag{1.10}$$

making use of $l_{ii} = 1$. Eqs. 1.10(a,b) for the matrix elements l_{ik} and u_{kj} can be solved in a specific order. (i) Start with $i = 1$. It follows from (b) that $u_{1j} = a_{1j},\ \ j = 1, ..., N$. (ii) From $j = 1$ in (a) it then follows $l_{i1} u_{11} = a_{i1}$, or $l_{i1} = a_{i1}/u_{11},\ \ i = 2, ..., N$. (iii) Again from (b) we get for $i = 2$, $l_{21} u_{1j} + u_{2j} = a_{2j}$, or $u_{2j} = a_{2j} - l_{21} u_{1j},\ j = 2, ..., N$. (iv) Then $j = 2$ in (a) gives $l_{i1} u_{12} + l_{i2} u_{22} = a_{i2}$, or $l_{i2} = (a_{i2} - l_{i1} u_{12}) / u_{22},\ i = 3, ..., N$. The rest should be clear to you now. Proceed with $i = 3$ in (a), then $j = 3$ in (b), $i = 4$ in (a), $j = 4$ in (b), etcetera.

The tricky steps from a numerical perspective are those where you have to divide by a matrix element u_{ii}. If the latter is zero, then the algorithm breaks down, as dividing by zero cannot be done, of course. Even is u_{ii} is nonzero but small, dividing by a small number can spell numerical trouble. More of this in the next section. The problem is caused by the fact that the algorithm goes through the rows and columns of $\mathbf{A}$ in a fixed order, and that particular order might be unlucky.

One can try to reorder the row/columns of $\mathbf{A}$ in such a way, that one always divides by the largest possible u_{ii}. This technique is called *pivoting*. Formally, such a reordering permutes the variables $x_i,\ \ i = 1, ..., N$, so the solution to the linear problem should remain the same (apart from this permutation). The effect of this permutation is caught in the matrix $\mathbf{P}$ in Eq. 2.38. Numerically, pivoting is the best possible strategy to avoid numerical problems in $\mathbf{LU}$ decomposition, so it is incorporated in most LU routines you find in numerical algebra libraries.

By the way, as a matter of principle, using such permutations it should be possible to always construct a matrix $\mathbf{U}$ where all diagonal matrix elements u_{ii} are nonzero. If one or more of these elements are zero, then the matrix $\mathbf{U}$ would be singular, as $\det(\mathbf{U}) = \prod_{i=1}^{N} u_{ii}$ for an upper triangular matrix, see the details in the previous section. By construction the matrix $\mathbf{A}$ would also be singular, as $\det(\mathbf{A}) = \det(\mathbf{LU}) = \det(\mathbf{L}) \det(\mathbf{U})$, and the linear problem $\mathbf{Ax} = \mathbf{b}$ cannot be solved.

1.1.2 Cost and stability

In assessing the quality of algorithms, two questions play a central role

(1) How expensive is the calculation?

(2) How stable is the algorithm?

The computational cost (1) of a calculation is determined by two factors,

(1a) How many numbers have to be kept in the (main) memory of the computer?

(1b) How many operations have to be performed on these numbers?

In physics and engineering we need calculations on real or complex numbers, which are represented by *floating point numbers* on the computer. So to answer these questions, this is the only data type that matters. In answering question (1), we are not much interested

in the nitty-gritty details, but foremost in how the costs scale as a function of N, where N is the dimension of the matrices/vectors defining the problem.

Starting with question (1a), a vector occupies N units of memory, and a matrix requires N^2 units. On many computers the unit of storing a floating point number is eight bytes for a real number, and sixteen bytes for a complex number. We don't care too much about such details, as long as there is some well-defined fixed unit. Storing a matrix can become a problem if N becomes large. The boundary between "small" and "large" is a function of time and computer technology. In the old days $N = 10^3$ was considered to be an extremely large number. Nowadays, something like $N = 10^5$ may be largish.

Irrespective of this boundary changing over time, important is that there is a boundary, and most likely we want to cross it in many physics and engineering applications. For small problems, we can afford to keep everything in memory, whereas for large problems, we need to be more inventive. Luckily, many of the matrices we encounter in large physics problems are *sparse*, meaning that most of their matrix elements are zero.[10] An example of a sparse matrix is the tridiagonal matrix encountered in the previous section. The memory requirement of a sparse matrix tends to be *linear* in N,[11] so nature has solved that problem for us.

Answering question (1b), in counting the number of operations on floating point numbers (called *flops*, floating point operations), we count the number of summations/subtractions, multiplications, and divisions. These operations are actually not equally expensive. Divisions are most expensive, and should be avoided if they are not indispensable. Multiplications are more expensive than summations/subtractions. If we are not interested in all details, we don't count the number of flops exactly, but only look at how that count scales as a function of N. For a fast assessment of the scaling, it is then often sufficient to calculate the number of multiplications.

An inner product between two vectors of dimension N, $\mathbf{a}^\dagger\mathbf{b} = a_1^* b_1 + \cdots + a_N^* b_N$, takes N multiplications. We say that this calculation scales as $O(N)$ (pronounced "order enn"), which expresses that the cost is proportional to N. Likewise, a matrix-vector multiplication $\mathbf{Ax} = \mathbf{b}$, $\sum_{j=1}^{N} a_{ij}x_j = b_j;\ j = 1, ..., N$, involves N^2 multiplications to calculate all elements of the vector $\mathbf{b}$. So, it is a $O(N^2)$ (order enn square) calculation. This implies that, if the dimension N is doubled, the computing time goes up roughly by a factor of $2^2 = 4$. A matrix-matrix multiplication $\mathbf{AB} = \mathbf{C}$, $\sum_{j=1}^{N} a_{ij}b_{jk} = c_{ik};\ i = 1, ..., N;\ k = 1, ..., N$, needs N^3 multiplications to calculate all elements of the matrix $\mathbf{C}$, and is a $O(N^3)$ (order enn cube) calculation. If the dimension N is doubled, the computing time goes up roughly by a factor $2^3 = 8$. It will be clear that this can become expensive fast.

If you don't take care, some algorithms (in particular, those involving non-linear problems) scale worse than that, i.e., $O(N^p);\ p > 3$, but $p = 3$ can already be sufficiently bad. If N becomes large, then even $p = 2$ can become too expensive. The holy grail for tackling large-scale problems is to find algorithms that scale as $O(N)$.[12]

[10] Finite difference, finite element, and Fourier transform techniques applied to physics problems mostly give sparse matrices.

[11] A tridiagonal matrix of dimension N has $3N - 2$ non-zero elements, so its memory requirement is certainly linear in N.

[12] Things can be worse. Some problems scale exponentially, such as $O(2^N)$, or even worse. This is the common computational bottleneck in problems derived from many-particle quantum mechanics, for instance. Such problems, involving "correlated" particles, remain extremely tough to solve even when using savage approximations, until we have a quantum computer (if ever). Truly hopeless are some combinatorial problems where the scaling is $O(N!)$. These are for hopeless people (mathematicians). At the other end of the scale,

The answer to question (1) gives you an indication of how long it takes to find a solution to the problem. The answer to question (2) should tell you how reliable that solution is. The computer uses a *fixed number of digits* P to represent a real or complex number (usually $P \approx 16$ for a real number, or for the real and imaginary parts of a complex number). This representation inevitably leads to *round-off* errors. Whereas P is larger than the number of significant digits you typically require for the solution to your physics problem, this does not automatically mean that you are safe. Round-off errors can accumulate in the multiple steps of your algorithm, resulting in the ultimate loss of significant digits, see Sec. 1.1.3. You should therefore choose your algorithm with care. At the least, you should be aware of possible problems with accuracy.

LU factorization. Let us analyze the LU decomposition algorithm of of Eqs. 1.7-1.9 along the lines discussed above. A full matrix $\mathbf{A}$ requires N^2 of memory; this is called a *dense matrix*. The special matrices $\mathbf{T}$, $\mathbf{L}$ and $\mathbf{U}$ we have used in Sec. 1.1.1, are *sparse*. In practical implementations, one will *never store* the sparse matrices $\mathbf{T}$, $\mathbf{L}$ and $\mathbf{U}$ *in dense matrix form*. As they mainly consist of zeros, not only would that be a waste of storage, but the architecture of most CPUs is also such, that operations on numbers stored far apart in memory are less efficient than operations on numbers that are adjacent in memory.

The tridiagonal matrix $\mathbf{T}$ can be represented by one vector of length N (the diagonal), plus two vectors of length $N-1$ (the subdiagonal, and the superdiagonal). The matrix $\mathbf{U}$ can be represented by one vector of length N (the diagonal), plus one vector of length $N-1$ (the superdiagonal), and the matrix $\mathbf{L}$ by one vector of length $N-1$ (the subdiagonal; no need to store the 1's on the diagonal; we know their values on forehand). This means that the algorithm needs $pN-q$ units of memory, where p, q are fixed, small integers.

Counting the number of operations in the algorithm of Eqs. 1.7-1.9 reveals that it scales as $O(N)$. That is pretty good, considering that LU decomposition of a dense matrix $\mathbf{A}$ scales as $O(N^3)$. So the important message is that, *whenever your matrices have a special structure, make use of that structure.*

That answers question (1) above. To answer question (2), note that the algorithm of Eq. 1.7 breaks down if $t_{11} = 0$. Dividing by zero is not possible of course; the computer will give "Inf" as result, which stands for ∞. Whereas the case $t_{11} = 0$ is perhaps not so common in physics-derived problems, $|t_{11}| \approx 0$ also spells trouble. Dividing by a very small number, you run the risk of an "overflow", i.e., a number that is too large for the computer to represent. The computer can only represent real numbers (or real and imaginary parts of complex numbers) r with $c < |r| < C$, where c, C are some fixed small and large numbers, determined by the computer architecture. If an operation would give you a number with $|r| > C$, this is called an *overflow*, and the computer returns a "Inf" result, with or without a warning or error message, depending on the programming language and settings you use. Similarly, if an operation would give you a number with $|r| < c$, this is called an *underflow*, and the computer returns "Zero", again with or without a warning or error message.

In other words, you get into trouble with the algorithm of Eq. 1.7 if $|t_{11}|$ is small, and $|t_{11}| = 0$ spells disaster. The same problem reoccurs if one of the $|u_{i-1,i-1}|$ of Eq. 1.7 is small or even zero. To try and avoid this problem, one can reorder the rows of the matrices involved, such that matrix elements with largest possible values $|u_{i-1,i-1}|$ are used in the division operations. This is called (partial) *pivoting*. Pivoting markedly increases the

I am not aware of a practical physics problem that scales better than $O(N)$, e.g., as $O(N^\alpha)$; $\alpha < 1$, or $O(\ln N)$.

stability of the algorithm. It is a fairly straightforward, but somewhat tedious, bookkeeping operation. The effect of it can be captured in the permutation matrix $\mathbf{P}$ in Eq. 2.38. For tridiagonal matrices one typically dislikes pivoting, as reordering the rows would spoil the tridiagonal character of the matrix. Fortunately, for the matrices we will encounter in the physics problems I have picked out, we get away without pivoting.

If even pivoting cannot get you out of trouble, your problem is ill-defined, i.e., singular or nearly singular. More about distinguishing well-defined from ill-defined problems in the next section. Meanwhile, if numerical problems occur, the approach of a practical physicist is to shop around for alternative algorithms. LU decomposition is the standard algorithm for solving linear problems. Although satisfactory in many cases, it is not the most robust algorithm around. Another option is QR factorization, see Sec. 1.2. It is more stable, but it is also computationally somewhat more expensive [3, 7]. For special matrices (Hermitian and positive definite), you can use a special form of LU decomposition, called Cholesky decomposition, see Sec. 1.1.5, which is also more stable. The worst problems can be attacked with the SVD algorithm (which stands for *singular value decomposition*). You can find descriptions in Refs. [3, 7]. It is very stable, but also more expensive.[13]

1.1.3 Error analysis

A matrix $\mathbf{A}$ that is nearly singular, is bad news. Write the solution to Eq. 1.1 as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. If $\mathbf{A}$ is singular, then $\mathbf{A}^{-1}$ does not exist, which means that there is no solution. If $\mathbf{A}$ is nearly singular, it is extremely tricky to find a solution. The mathematicians call that an *ill-conditioned* problem. To obtain a more quantitative feeling of what that means, one can analyze the error in the solution to the linear problem.

An error analysis of the linear problem $\mathbf{Ax} = \mathbf{b}$ starts as follows. Suppose the source vector $\mathbf{b}$ is modified to $\mathbf{b} + \boldsymbol{\delta}$, where $\boldsymbol{\delta}$ represent an uncertainty in a measurement, or, as is typical for numerical calculations, a round-off error caused by the computer representing each number by a fixed number of bits. The solution vector $\mathbf{x}$ is then modified to $\mathbf{x} + \mathbf{e}$, where $\mathbf{e}$ is called the *error vector*. The typical question then is: what is the relative error $\|\mathbf{e}\| \,/\, \|\mathbf{x}\|$, given the relative uncertainty $\|\boldsymbol{\delta}\| \,/\, \|\mathbf{b}\|$?

1.1.3.1 Condition number

We have

$$\frac{\|\mathbf{e}\| \,/\, \|\mathbf{x}\|}{\|\boldsymbol{\delta}\| \,/\, \|\mathbf{b}\|} = \frac{\|\mathbf{A}^{-1}\boldsymbol{\delta}\| \,/\, \|\mathbf{x}\|}{\|\boldsymbol{\delta}\| \,/\, \|\mathbf{Ax}\|} = \frac{\|\mathbf{A}^{-1}\boldsymbol{\delta}\|}{\|\boldsymbol{\delta}\|} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|}. \tag{1.11}$$

Using the definition of the norm of a matrix

$$\|\mathbf{A}\| = \max_{\mathbf{x}} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|}, \tag{1.12}$$

then the maximum relative error is

$$\max \frac{\|\mathbf{e}\| \,/\, \|\mathbf{x}\|}{\|\boldsymbol{\delta}\| \,/\, \|\mathbf{b}\|} = \left\|\mathbf{A}^{-1}\right\| \|\mathbf{A}\| \equiv \kappa(\mathbf{A}) \geq 1. \tag{1.13}$$

[13] If a problem becomes too bad, ask a mathematician. That's what they are there for. You run the risk however that he/she will spend his/her time proving that a solution exists, and still leave you the task to find it.

$\kappa(\mathbf{A})$ is called the *condition number* of the matrix. As $\|\mathbf{A}\|\,\|\mathbf{B}\| \geq \|\mathbf{AB}\|$ (easy to prove from the definition of matrix norm), we have $\|\mathbf{A}^{-1}\|\,\|\mathbf{A}\| \geq \|\mathbf{A}^{-1}\mathbf{A}\| = \|\mathbf{I}\| = 1$, so the condition number ≥ 1.

If the condition number is large, we have a problem, as the relative error in the solution $\|\mathbf{e}\| \,/\, \|\mathbf{x}\|$ can be as large as $\kappa(\mathbf{A})$ times the relative uncertainty in the data $\|\boldsymbol{\delta}\| \,/\, \|\mathbf{b}\|$. If P is the number of digits used to represent a number on the computer, then round-off of numbers gives $\|\boldsymbol{\delta}\| \,/\, \|\mathbf{b}\| = 10^{-P}$. This means that the number of significant digits S in the solution $\|\mathbf{x}\|$ is

$$0 \leq S \leq P - \log_{10} \kappa(\mathbf{A}). \tag{1.14}$$

The rightmost "$\leq$" becomes an "$=$" if during the steps of your algorithm you don't not loose additional digits. Needless to say that, if the number of significant digits S becomes small, i.e., if the condition number $\kappa(\mathbf{A})$ is large, then the solution becomes relatively useless, which is the hallmark of an ill-conditioned problem. A matrix with a large condition number is called an *ill-conditioned matrix*. If $\log_{10} \kappa(\mathbf{A}) \geq P$, we have no significant digits left. This is worse than an ill-conditioned problem, it is a hopeless problem.

To obtain a feeling for the condition number, we can have a look at Hermitian matrices. If $\mathbf{A}$ is a Hermitian matrix, then the condition number, Eq. 1.13, is given by

$$\kappa(\mathbf{A}) = \left| \frac{\lambda_{\max}}{\lambda_{\min}} \right|. \tag{1.15}$$

Here $\lambda_{\max}$ is the eigenvalue of $\mathbf{A}$ with the largest absolute value. So, if $\lambda_i;\ i = 1, ..., N$ are the eigenvalues of $\mathbf{A}$, then $|\lambda_i| \leq |\lambda_{\max}|$. Similarly, $\lambda_{\min}$ is the eigenvalue of $\mathbf{A}$ with the smallest absolute value, i.e., $|\lambda_i| \geq |\lambda_{\min}|$. To have a well-conditioned linear problem, the eigenvalues of $\mathbf{A}$ should therefore not differ too much in absolute value.

Proof: the eigenvectors $\mathbf{x}_i;\ i = 1, ..., N$ of a $N \times N$ Hermitian matrix form an orthonormal basis, i.e. $\mathbf{x}_i^\dagger \mathbf{x}_j = \delta_{ij}$, see Sec. 2.2.1. Any vector can be expanded as $\mathbf{x} = \sum_{i=1}^N c_i \mathbf{x}_i$. We then have

$$\frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} = \frac{\left\|\sum_{i=1}^N c_i \lambda_i \mathbf{x}_i\right\|}{\left\|\sum_{i=1}^N c_i \mathbf{x}_i\right\|} = \frac{\sqrt{\sum_{i=1}^N \sum_{j=1}^N c_i^* c_j \lambda_i \lambda_j \mathbf{x}_i^\dagger \mathbf{x}_j}}{\sqrt{\sum_{i=1}^N \sum_{j=1}^N c_i^* c_j \mathbf{x}_i^\dagger \mathbf{x}_j}} = \frac{\sqrt{\sum_{i=1}^N |c|_i^2 \, \lambda_i^2}}{\sqrt{\sum_{i=1}^N |c|_i^2}}. \tag{1.16}$$

This expression is maximal for $c_i = 1$, i labeling $\lambda_{\max}$, and $c_i = 0$ for all other i, which means

$$\|\mathbf{A}\| = \max_{\mathbf{x}} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} = |\lambda_{\max}| \equiv \rho(\mathbf{A}) \tag{1.17}$$

The quantity $\rho(\mathbf{A})$ is called the *spectral radius* of the matrix $\mathbf{A}$. It is always equal to $|\lambda_{\max}|$, and for Hermitian matrices it is also equal to $\|\mathbf{A}\|$.

One can also show that

$$\left\|\mathbf{A}^{-1}\right\| = \frac{1}{|\lambda_{\min}|} \equiv \rho\left(\mathbf{A}^{-1}\right), \tag{1.18}$$

where $\lambda_{\min}$ is the eigenvalue of $\mathbf{A}$ with the smallest absolute value. The proof of this starts from the observation that, if λ_i is an eigenvalue of $\mathbf{A}$, then $1/\lambda_i$ is an eigenvalue of $\mathbf{A}^{-1}$ and vice versa. Start from $\mathbf{Ax}_i = \lambda_i \mathbf{x}_i$, multiplying both sides from the left with $\mathbf{A}^{-1}$ gives $\mathbf{x}_i = \lambda_i \mathbf{A}^{-1} \mathbf{x}_i$, which means $\lambda_i^{-1} \mathbf{x}_i = \mathbf{A}^{-1} \mathbf{x}_i$. The largest eigenvalue (in the absolute sense) of $\mathbf{A}^{-1}$ then corresponds to 1 / smallest eigenvalue of $\mathbf{A}$, so $\left\|\mathbf{A}^{-1}\right\| = 1/\left|\lambda_{\min}\right|$. Using these results, Eq. 1.15 then follows from Eq. 1.13.

For a truly singular matrix $\lambda_{\min} = 0$, and the condition number is ∞. For a near-singular matrix $\lambda_{\min}$ is very small, and the condition number becomes very large. As discussed in the previous section, the best option you have to attack an ill-conditioned problem is the SVD algorithm.

1.1.3.2 Preconditioning*

A standard trick to improve the conditioning of your problem, is to change the linear problem $\mathbf{Ax} = \mathbf{b}$ into

$$\mathbf{A}'\mathbf{x} = \mathbf{b}' \text{ with } \mathbf{A}' = \mathbf{BA} \text{ and } \mathbf{b}' = \mathbf{Bb}. \tag{1.19}$$

If the matrix $\mathbf{B}$ is chosen such that $\kappa(\mathbf{A}') < \kappa(\mathbf{A})$, then the error in the solution is smaller than that of the original problem, according to Eq. 1.13. In numerical folklore this modification is called *preconditioning* and the matrix $\mathbf{B}$ is called the *preconditioner*. A good preconditioner can make a problem solvable that is very difficult to solve otherwise. In case one uses iterative techniques to solve a problem, see Secs. 1.3 and 1.4, a good preconditioner can dramatically reduce the number of iterations required to find an acceptable solution.

Coming up with a good preconditioner is easier said then done. The best preconditioner is $\mathbf{B} = \mathbf{A}^{-1}$, as then $\kappa(\mathbf{A}') = \kappa(\mathbf{I}) = 1$, which is the minimal condition number, see Eq. 1.13. This statement is fairly useless however, because, if we had the matrix $\mathbf{A}^{-1}$, we would not need to solve the linear problem in the first place, as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ would give the solution straight away. However, an easily constructed approximation to $\mathbf{A}^{-1}$ can be very helpful as a preconditioner.

For instance, the diagonal matrix with elements

$$\left(\mathbf{D}^{-1}\right)_{ij} = \delta_{ij}\frac{1}{a_{ii}}, \tag{1.20}$$

is an inverse of the diagonal matrix $\mathbf{D}_{ij} = \delta_{ij}a_{ii}$, which is the matrix obtained from $\mathbf{A}$ by neglecting all its off-diagonal elements. If the full matrix $\mathbf{A}$ is *diagonally dominant,* i.e., if the off-diagonal elements of $\mathbf{A}$ are small, one may expect that Eq. 1.20 is still a decent approximation to its inverse, and is suitable as a preconditioner. In all honesty, even if $\mathbf{A}$ is not diagonally dominant, Eq. 1.20 is often tried as a preconditioner, for lack of a better idea. On a more positive note, inside knowledge of your physical problem (for instance, the ability to solve it analytically in certain limits) can enable you to construct a good preconditioner.

1.1.4 Iterative improvement*

As explained above, you loose at least $\log_{10}\kappa(\mathbf{A})$ significant digits in solving the linear problem $\mathbf{Ax} = \mathbf{b}$, see Eq. 1.14. If the solution is not sufficiently accurate for your application, you can use a technique called *iterative improvement*. After solving the problem you may not have the exact solution $\mathbf{x}$, but you do have an approximate solution $\mathbf{x}' = \mathbf{x} + \mathbf{e}$. As $\mathbf{Ax}' = \mathbf{Ax} + \mathbf{Ae}$, it means that

$$\mathbf{Ae} = \mathbf{r}, \quad \text{with } \mathbf{r} = \mathbf{Ax}' - \mathbf{b}, \tag{1.21}$$

called the *residual vector*. We can solve Eq. 1.21 quite easily, as we already have the LU decomposition of the matrix $\mathbf{A}$. Solving the problem then consists of forward substitution, followed by backward substitution.

Of course you also loose $\log_{10}\kappa(\mathbf{A})$ significant digits in the solution $\mathbf{e}$. However, one may expect that $\|\mathbf{e}\| \,/\, \|\mathbf{x}'\| \propto 10^{-S}$, Eq. 1.14, so $\|\mathbf{x}' - \mathbf{e}\|$ should have $2S$ significant digits instead of the original S found in $\|\mathbf{x}'\|$. One can repeat this procedure a couple of times in order to increase the accuracy,

$$\begin{aligned} &\mathbf{r}_0 = \mathbf{b};\ \text{solve}\ \mathbf{A}\mathbf{e}_1 = \mathbf{r}_0;\ \mathbf{x}_1 = \mathbf{e}_1 \\ &\text{do}\ n = 1, \ldots\ \ \mathbf{r}_n = \mathbf{r}_{n-1} + \mathbf{A}\mathbf{e}_n;\ \text{solve}\ \mathbf{A}\mathbf{e}_{n+1} = \mathbf{r}_n;\ \mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{e}_{n+1};\ \text{until converged;} \end{aligned} \tag{1.22}$$

The phrase "until converged" marks a criterion of the type $\|\mathbf{e}_{n+1}\| \,/\, \|\mathbf{x}_1\| < \epsilon$ with ϵ a tolerance set by the user.

Obviously, it makes no sense to set the tolerance sharper than machine precision, so $\epsilon > 10^{-P}$, or $|\log_{10}\epsilon| < P$. At each iteration step one may expect the accuracy of the solution to increase by S digits. The number of iterations $n_{\max}$ required to obtain an accuracy ϵ is then estimated as

$$n_{\max} \approx \frac{|\log_{10}\epsilon|}{S} \approx \frac{|\log_{10}\epsilon|}{P - \log_{10}\kappa(\mathbf{A})}, \tag{1.23}$$

see Eq. 1.14. If $\kappa(\mathbf{A}) \approx 1$, one does not need to iterate, and the result $\mathbf{x}_1$ is already good.

Theoretically, as long as there is any significant digit in the solution, $S > 0$, you should be able to iterate to a solution. However, I would not try to live on the edge. Avoid small S, i.e., large $\kappa(\mathbf{A})$. At least, give it a serious try to get the condition number down by preconditioning.

1.1.5 Cholesky decomposition (CD)*

The matrices we encounter in physics often have special properties. Suppose, for instance, we have a basis of non-orthogonal, linearly independent axes, $\overrightarrow{a}_i;\ i = 1, ..., N;\ \overrightarrow{a}_i \cdot \overrightarrow{a}_j \neq 0$. We can form the matrix $\mathbf{A}$, with matrix elements $A_{ij} = \overrightarrow{a}_i \cdot \overrightarrow{a}_j;\ i = 1, ..., N;\ j = 1, ..., N$. This $N \times N$ matrix has the properties

$$\begin{aligned} \mathbf{A}^\dagger &= \mathbf{A}\ \ \text{(Hermitian)}; \\ \mathbf{x}^\dagger\mathbf{A}\mathbf{x} &> 0,\ \text{for any}\ \mathbf{x} \neq \mathbf{0}\ \ \text{(positive definite)}. \end{aligned} \tag{1.24}$$

The last property follows from $\overrightarrow{x} = \sum_{i=1}^N x_i \overrightarrow{a_i}$, and $\|\overrightarrow{x}\|^2 = \sum_{i=1}^N \sum_{j=1}^N x_i^* \overrightarrow{a}_i \cdot \overrightarrow{a}_j x_j = \mathbf{x}^\dagger\mathbf{A}\mathbf{x}$, which is > 0, as the norm of a (non-zero) vector is always a positive number. In physics and chemistry, such a matrix $\mathbf{A}$ is often called the *overlap matrix*; $A_{ij} = \overrightarrow{a}_i \cdot \overrightarrow{a}_j \neq 0$ indicates that the axes $\overrightarrow{a}_i$ and $\overrightarrow{a}_j$ "overlap"; they point partially in the same direction. A Hermitian, positive definite matrix like this is also often used to define the *metric* of a space, where it is then called the *metric tensor* or simply the *metric*.

Linear problems $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a Hermitian, positive definite $N \times N$ matrix, can be solved by a special technique, called *Cholesky decomposition* (CD). It is a form of LU decomposition that is adapted to this special kind of matrix. A Hermitian positive definite matrix $\mathbf{A}$ can be decomposed as

$$\mathbf{A} = \mathbf{L}\mathbf{L}^\dagger;\ \ \text{with}\ l_{ij} = 0,\ j > i, \tag{1.25}$$

a lower triangular matrix. I leave the general proof of this statement to the mathematicians.

For tridiagonal Hermitian positive definite matrices $\mathbf{T}$, the proof is easy. One just has to follow the reasoning of Sec. 1.1.1. Write out the matrix product $\mathbf{L}\mathbf{L}^\dagger = \mathbf{T}$. This gives $t_{i,i-1} = l_{i,i-1}l^*_{i-1,i-1}$, $t_{i,i} = l_{i,i-1}l^*_{i,,i-1} + l_{i,i}l^*_{i,i},$; $i = 2, ..., N$, and $t_{1,1} = |l_{1,1}|^2$. Inverting these relations, then gives the *Cholesky decomposition* (CD) algorithm

$$l_{11} = \sqrt{t_{11}};\ \text{do } i = 2, ..., N \quad l_{i,i-1} = \frac{t_{i,i-1}}{l_{i-1,,i-1}};\ l_{i,i} = \sqrt{t_{i,i} - |l_{i,i-1}|^2}. \tag{1.26}$$

The diagonal matrix elements of $\mathbf{T}$ are real and positive, $t_{i,i} > 0$. This follows directly from Eq. 1.24. Using $\mathbf{x} = \mathbf{e}_i$, a unit vector with elements $(\mathbf{e}_i)_j = \delta_{ij}$, one has $\mathbf{e}_i^\dagger \mathbf{A}\mathbf{e}_i = A_{ii} > 0$. This means in Eq. 1.26 that l_{11} is a real positive number. One can show that the $l_{i,i};\ i > 1$ in Eq. 1.26 are also all real positive numbers.

Solving the linear problem is then done by forward substitution, followed by back substitution, as when using LU factorization, Eqs. 1.8 and 1.9, but using $\mathbf{L}^\dagger$ instead of $\mathbf{U}$ in the latter.

CD is a stable algorithm, but it requires a matrix $\mathbf{A}$ with special properties, Eq. 1.24. It is often not so simple to figure out a priori whether a (Hermitian) matrix $\mathbf{A}$ is positive definite. CD can be used to test whether this is the case. If $\mathbf{A}$ is *not* positive definite, then the argument of the square root in Eq. 1.26 becomes zero or negative for at least one of the $i = 1, ..., N$. If this is the case, then $\mathbf{A}$ is *not* positive definite. If the algorithm proceeds to the end without zero or negative arguments, then $\mathbf{A}$ is positive definite. It is simple to build in a test on the argument of the square root into Eq. 1.26.

Referring back to our original example, define a set of vectors

$$\overrightarrow{b}_i = \sum_{k=1}^{N} \left(L^{-1}_{ik}\right)^* \overrightarrow{a}_k;\ i = 1, ..., N, \quad \text{then} \tag{1.27}$$

$$\begin{aligned}
\overrightarrow{b}_i \cdot \overrightarrow{b}_j &= \sum_{k,\ell=1}^{N} L^{-1}_{ik} \overrightarrow{a}_k \cdot \overrightarrow{a}_\ell \left(L^{-1}_{j\ell}\right)^* = \sum_{k,\ell=1}^{N} L^{-1}_{ik} A_{k\ell} L^{-\dagger}_{\ell j} \\
&= \sum_{k,\ell,m=1}^{N} L^{-1}_{ik} L_{km} L^\dagger_{m\ell} L^{-\dagger}_{\ell j} = \sum_{m=1}^{N} \delta_{im}\delta_{mj} = \delta_{ij},
\end{aligned}$$

using Eq. 1.25, and the fact that $\mathbf{B}^{-1}\mathbf{B} = \mathbf{I}$, for any matrix $\mathbf{B}$. In other words, Cholesky decomposition can be also be looked upon as a technique for constructing an orthogonal set of vectors. It is fairly robust, but more expensive than the standard orthogonalization technique discussed in the next section.

1.2 QR factorization

QR factorization is the standard technique for orthogonalizing a set of vectors. In many applications in physics one prefers to use orthogonal basis sets, as non-orthogonal ones are difficult to work with, and can cause serious distress in vital body parts (such as chronic headaches). QR factorization supplies an orthogonal basis set. QR factorization is also a vital ingredient of other algorithms, see, e.g., Sec. 1.4.3, or the lecture notes on eigenvalue problems. Last, but not least, QR factorization supplies a technique for solving a linear problem, $\mathbf{A}\mathbf{x} = \mathbf{b}$, as a (more stable, but somewhat more expensive) alternative to LU decomposition. QR factorization is based upon the following.

***Theorem*:** a square matrix $\mathbf{A}$ can be written as

$$\mathbf{A} = \mathbf{QR} \text{ with } \mathbf{Q} \text{ a unitary matrix, and } R_{ij} = 0,\ i > j, \tag{1.28}$$

an upper-triangular matrix.[14] A unitary matrix of dimension N is defined by the property

$$\mathbf{Q} = (\mathbf{q}_1\ \mathbf{q}_2\ ...\ \mathbf{q}_N);\quad \mathbf{q}_j^\dagger \mathbf{q}_k = \delta_{jk};\ j = 1, ..., N;\ k = 1, ..., N. \tag{1.29}$$

Here $\mathbf{q}_j;\ j = 1, ..., N$ are the $N \times 1$ vectors that make up the columns of $\mathbf{Q}$. The columns of a unitary matrix are thus orthonormal (orthogonal and normalized). From this property it is then trivial to prove that $\mathbf{Q}^\dagger\mathbf{Q} = \mathbf{I}$, the $N \times N$ identity matrix, as well as $\mathbf{QQ}^\dagger = \mathbf{I}$. An alternative definition of a unitary matrix $\mathbf{Q}$ is a matrix with the property

$$\mathbf{Q}^\dagger = \mathbf{Q}^{-1}. \tag{1.30}$$

One can prove Eq. 1.28 by explicit construction of $\mathbf{Q}$ and $\mathbf{R}$, which we will do in the next section.

As the columns of $\mathbf{Q}$ are orthogonal, and the columns of $\mathbf{A}$ are not, QR factorization can be viewed as a transformation between an orthogonal basis set and a non-orthogonal one. In other words, QR factorization is a way of orthogonalizing a basis set. An application of that is shown in Sec. 1.2.2.

QR factorization can also be used to solve a linear problem. Write $\mathbf{Ax} = \mathbf{QRx} = \mathbf{b}$, then we can multiply left and right by $\mathbf{Q}^\dagger$ to get

$$\mathbf{Ax} = \mathbf{b} \Leftrightarrow \mathbf{Rx} = \mathbf{Q}^\dagger\mathbf{b}. \tag{1.31}$$

Because $\mathbf{R}$ is an upper-triangular matrix, this linear problem is solved easily by back substitution. The computational scaling of QR factorization and LU decomposition is the same. For instance, for a tridiagonal matrix $\mathbf{A}$ the number of operations required scales as $pN - q$, with p, q numbers independent of N. One has $p_{\mathrm{QR}} > p_{\mathrm{LU}}$. The main advantage of QR is that it is more stable than LU decomposition.

1.2.1 QR factorization by Gram-Schmidt orthogonalization

The textbook way of orthogonalizing a basis set is by *Gram-Schmidt orthogonalization*. Calling $\mathbf{a}_i;\ i = 1, ..., N$ the columns of $\mathbf{A}$, and $\mathbf{q}_i;\ i = 1, ..., N$ the columns of $\mathbf{Q}$, we can write

$$\begin{aligned}
&\text{do } j = 1, ..., N \quad \mathbf{q}_j = \mathbf{a}_j; \\
&\text{do } j = 2, ..., N \quad \text{do } k = 1, ..., j-1 \quad \mathbf{q}_j = \mathbf{q}_j - \mathbf{q}_k \left(\mathbf{q}_k^\dagger \mathbf{a}_j\right) / \left(\mathbf{q}_k^\dagger \mathbf{q}_k\right); \\
&\text{do } j = 1, ..., N \quad \mathbf{q}_j = \mathbf{q}_j / \left\|\mathbf{q}_j\right\|;
\end{aligned} \tag{1.32}$$

The second line is the orthogonalization step, and the third line is the normalization step. The unitary matrix $\mathbf{Q}$ is then simply formed by the columns $\mathbf{Q} = (\ \mathbf{q}_1\ \ \mathbf{q}_2\ \ \cdots\ \ \mathbf{q}_{N-1}\ \ \mathbf{q}_N\ )$. A two-dimensional graphical example of Gram-Schmidt orthogonalization is given in figure 1.1.

[14] I stick to the common nomenclature here. Sometimes the literature uses $\mathbf{U}$ to denote an upper-triangular matrix, and sometimes it uses $\mathbf{R}$ for the same thing. As far as I know, the algorithms are called QR and LU; I have never heard of QU or LR within the linear problems context.

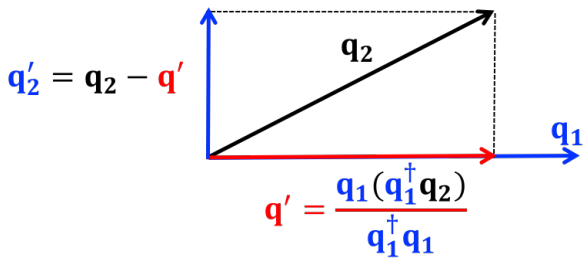


Figure 1.1: Gram-Schmidt orthogonalization in 2D. Define a unit vector in the direction of $\mathbf{q}_1$ as $\mathbf{u} = \mathbf{q}_1 / \|\mathbf{q}_1\|$, with $\|\mathbf{q}_1\| = \sqrt{\mathbf{q}_1^\dagger \mathbf{q}_1}$. Then $\mathbf{q}' = \mathbf{u}\left(\mathbf{u}^\dagger \mathbf{q}_2\right) = \mathbf{q}_1 \left(\mathbf{q}_1^\dagger \mathbf{q}_2\right) / \mathbf{q}_1^\dagger \mathbf{q}_1$ represents the projection of $\mathbf{q}_2$ onto $\mathbf{u}$. It follows that $\mathbf{q}_2' = \mathbf{q}_2 - \mathbf{q}'$ is orthogonal to $\mathbf{q}_1$, i.e., $\mathbf{q}_1^\dagger \mathbf{q}_2' = 0$.

Linear algebra textbooks give Gram-Schmidt as the standard algorithm for orthogonalization. Numerical mathematics textbooks will tell you that the algorithm of Eq. 1.32 is very bad, because it is numerically unstable. Luckily, those textbooks also tell you how to fix the algorithm, such that it becomes more stable

$$\begin{aligned} &\text{do } j = 1, ..., N \quad \mathbf{q}_j = \mathbf{a}_j; \\ &\text{do } j = 2, ..., N \quad \text{do } k = 1, ..., j-1 \quad \mathbf{q}_j = \mathbf{q}_j - \mathbf{q}_k \left(\mathbf{q}_k^\dagger \mathbf{q}_j\right) / \left(\mathbf{q}_k^\dagger \mathbf{q}_k\right); \\ &\text{do } j = 1, ..., N \quad \mathbf{q}_j = \mathbf{q}_j / \|\mathbf{q}_j\|; \end{aligned} \tag{1.33}$$

This algorithm is called *modified Gram-Schmidt orthogonalization* (MGS). When I will talk of Gram-Schmidt in future, I will mean this modified algorithm.

Note that the modification from Eq. 1.32 to Eq. 1.33 is actually quite subtle; the $\mathbf{a}_j$ in the second line is replaced by $\mathbf{q}_j$. If one had as many digits as one likes to represent numbers, this modification would not have any effect, and both versions of Gram-Schmidt produce exactly the same set of orthonormal vectors $\mathbf{q}_j$; $j = 1, ..., N$. However, on a computer numbers are represented by a fixed number of digits, which produces errors due to the round-off of numbers. In the Gram-Schmidt algorithm of Eq. 1.32 such round-off errors accumulate in the final vectors $\mathbf{q}_j$, and vectors become increasingly inaccurate for increasing j. In the Gram-Schmidt algorithm of Eq. 1.33 one uses in the inner loop iteration k, the $\mathbf{q}_j$ obtained in the previous iteration $k-1$, including its round-off error, and orthogonalizes this to $\mathbf{q}_k$. One tries to compensate for the round-off error of the previous cycle in the next cycle. Hence, modified Gram-Schmidt is a better algorithm than the original Gram-Schmidt.[15]

In practice, one always uses MGS if Gram-Schmidt is required.[16] Nonetheless, it can still break down, in particular if the original vectors $\mathbf{a}_j$; $j = 1, ..., N$ are close to being linearly dependent. We will assume that MGS is adequate for our needs. Besides its simplicity, one of the advantages of the algorithm is its versatility. For instance, the algorithm also works if the length of the vectors $\mathbf{a}_j$ is $M > N$.

Extracting the matrix elements of the upper-triangular matrix $\mathbf{R}$ involves a simple modification of MGS

[15] If you are not satisfied with this explanation, you have my sympathy. For a proper analysis of the errors you should consult the mathematics literature.

[16] Gram-Schmidt is not the only way to produce an orthogonal set, or perform QR factorization. An alternative is, for instance, using Householder transformations, see Sec. 1.4.4. Householder or Givens transformations are actually more preferred methods in the numerical community, as they are more robust, see Ref. [3]. They are also more complicated, however.

$$\begin{array}{l} \text{do } j = 1, ..., N \quad \mathbf{q}_j = \mathbf{a}_j; \\ \text{do } j = 1, ..., N \quad \{R_{jj} = \|\mathbf{q}_j\|\,;\ \mathbf{q}_j = \mathbf{q}_j / R_{jj}; \\ \qquad\qquad\qquad \text{do } k = 1, ..., j-1 \quad R_{kj} = \mathbf{q}_k^\dagger \mathbf{q}_j;\ \mathbf{q}_j = \mathbf{q}_j - \mathbf{q}_k R_{kj}\}; \end{array} \tag{1.34}$$

Details: We still have to prove that this algorithm gives the QR factorization stated in Eq. 1.28. As the vectors $\mathbf{q}_j;\ j = 1, ..., N$ form an orthonormal basis, we can write

$$\mathbf{a}_k = \sum_{j=1}^{N} \mathbf{q}_j \left(\mathbf{q}_j^\dagger \mathbf{a}_k\right) = \sum_{j=1}^{k-1} \mathbf{q}_j R_{jk} + \mathbf{q}_k \left(\mathbf{q}_k^\dagger \mathbf{a}_k\right) + \sum_{j=k+1}^{N} \mathbf{q}_j \left(\mathbf{q}_j^\dagger \mathbf{a}_k\right). \tag{1.35}$$

The final term on the right-hand side is zero, as the algorithm of Eq. 1.34 ensures that $\mathbf{q}_j^\dagger \mathbf{a}_k = 0$, for $j > k$. Focussing on the second term on the right-hand side of Eq. 1.35, we have

$$\mathbf{q}_k^\dagger \mathbf{a}_k = \frac{1}{R_{kk}} \left(\mathbf{a}_k - \sum_{j=1}^{k-1} \mathbf{q}_j R_{jk}\right)^\dagger \mathbf{a}_k = \frac{1}{R_{kk}} \left(\|\mathbf{a}_k\|^2 - \sum_{j=1}^{k-1} |R_{jk}|^2\right) = R_{kk}. \tag{1.36}$$

The last equality is derived from

$$R_{kk}^2 = \left(\mathbf{a}_k - \sum_{j=1}^{k-1} \mathbf{q}_j R_{jk}\right)^\dagger \left(\mathbf{a}_k - \sum_{j=1}^{k-1} \mathbf{q}_j R_{jk}\right) = \|\mathbf{a}_k\|^2 - 2\sum_{j=1}^{k-1} |R_{jk}|^2 + \sum_{j=1}^{k-1} |R_{jk}|^2. \tag{1.37}$$

Using Eq. 1.36 in Eq. 1.35 then gives

$$\mathbf{a}_k = \sum_{j=1}^{k} \mathbf{q}_j R_{jk}, \tag{1.38}$$

or in matrix form $\mathbf{A} = \mathbf{QR}$, which is Eq. 1.28.

The QR factorization algorithm of Eq. 1.34 works for a $M \times N$ matrix $\mathbf{A}$, with $M \geq N$. The matrix $\mathbf{Q}$ is then also $M \times N$, with columns $\mathbf{q}_j^\dagger \mathbf{q}_k = \delta_{jk};\ j = 1, ..., N;\ k = 1, ..., N$, and the the matrix $\mathbf{R}$ is $N \times N$ upper triangular, $R_{ij} = 0,\ i > j$.

Should $\mathbf{A} = \mathbf{T}$ be a tridiagonal matrix, then the algorithm simplifies. The columns of a tridiagonal matrix obey $\mathbf{t}_k^\dagger \mathbf{t}_j = 0$ for $k < j - 2$ or $k > j + 2$, meaning that they are already orthogonal for $|k - j| > 2$. Eq. 1.34 is then modified to

$$\begin{array}{l} \text{do } j = 1, ..., N \quad \mathbf{q}_j = \mathbf{t}_j; \\ \text{do } j = 1, ..., N \quad \{R_{jj} = \|\mathbf{q}_j\|\,;\ \mathbf{q}_j = \mathbf{q}_j / R_{jj}; \\ \qquad\qquad\qquad \text{do } k = j-2, j-1 \quad R_{kj} = \mathbf{q}_k^\dagger \mathbf{q}_j;\ \mathbf{q}_j = \mathbf{q}_j - \mathbf{q}_k R_{kj}\}; \end{array} \tag{1.39}$$

The upper triangular matrix $\mathbf{R}$ has also become simpler. It has a bandwidth of 2, i.e., $R_{kj} = 0;\ k > j$ or $k < j - 2$.

1.2.2 Orthogonal basis functions

In many applications in physics and engineering we need to approximate a function $g(x)$ in an interval $[a, b]$ by a linear combination of known elementary functions, called basis functions. Think, for instance, of signal analysis, or of quantum mechanics.

$$g(x) \approx \sum_{k=1}^{N} c_k f_k(x) \equiv \tilde{g}(x). \tag{1.40}$$

We would like to choose the coefficients c_k such, that $\tilde{g}(x)$ gives the best possible approximation of $g(x)$. Here "best possible" is commonly defined by a least squares fit, i.e.,

$$\min_{c_1,...,c_N} F(c_1, ..., c_N) \quad \text{with } F(c_1, ..., c_N) = \int_a^b [g(x) - \tilde{g}(x)]^2 \, dx, \tag{1.41}$$

gives the "best possible" coefficients $c_1, ..., c_N$.

Finding a minimum of a function can be done by setting the first derivatives to zero, $\partial F/\partial c_k = 0;\ k = 1, ..., N$. This gives the following set of N linear equations[17]

$$\mathbf{Ac} = \mathbf{b} \text{ with components } \ A_{kl} = \int_a^b f_k(x) f_l(x) dx \text{ and } b_k = \int_a^b f_k(x) g(x) dx. \tag{1.42}$$

One could use one of the standard numerical techniques to solve this linear problem. The matrix $\mathbf{A}$ is in fact of the type discussed in Sec. 1.1.5, i.e., Hermitian and positive definite, so we can apply Cholesky decomposition. We have seen, however, that the results can be inaccurate if the matrix $\mathbf{A}$ is ill-conditioned. Things would be a lot better if the functions $f_k(x);\ k = 1, ..., N$ are orthonormal, as then $A_{kl} = \int_a^b f_k(x) f_l(x) dx = \delta_{kl}$, or, in other words $\mathbf{A} = \mathbf{I}$. Not only is the condition number then minimal, $\kappa(\mathbf{I}) = 1$, but Eq. 1.42 is then trivially solved by $c_k = b_k = \int_a^b f_k(x) g(x) dx$.

In practice, the functions $f_k(x);\ k = 1, ..., N$ we have to deal with, are often not orthonormal. In that case we can use QR factorization to construct orthonormal functions. Define

$$f_k(x) = \sum_{j=1}^{k} q_j(x) R_{jk} \text{ with } \int_a^b q_k(x) q_l(x) dx = \delta_{kl}; \quad 1 \le k, l \le N, \tag{1.43}$$

and $R_{jk} = 0,\ j > k$, an upper-triangular matrix. Using this in Eq. 1.40 gives

$$\tilde{g}(x) = \sum_{j=1}^{N} d_j q_j(x) \text{ with } d_j = \sum_{k=j}^{N} R_{jk} c_k. \tag{1.44}$$

Using the orthonormal functions $q_j(x)$ instead of $f_k(x)$, the linear problem is trivially solved by

$$d_j = \int_a^b q_j(x) g(x) dx. \tag{1.45}$$

[17] Mathematicians can prove that F has a minimum, that $\partial F/\partial c_k = 0$ has a solution, and that this solution gives the minimum. Most likely, this is true only if one uses "reasonable" functions, but it makes little sense to use unreasonable ones, and physicists are sensible people.

Should one want the coefficients c_k of the original functions in Eq. 1.40, one can always obtain them from Eq. 1.44. As R_{jk} is an upper triangular matrix, the linear problem for c_k can be solved by back substitution.

Can we construct a set of functions $q_j(x)$, and an upper triangular matrix R_{jk}, such that Eq. 1.43 holds? Yes we can. A numerical procedure goes as follows. Define vectors $\mathbf{f}_k$ with elements $(f_k)_i = f_k(x_i);\ k = 1, ..., N;\ i = 1, ..., M \geq N$, then the QR factorization algorithm of Eq. 1.39 yields the vectors $\mathbf{q}_j$ with elements $(q_j)_i = q(x_i);\ j = 1, .., N;\ i = 1, ..., M$ and the matrix $\mathbf{R}$ with elements R_{jk}. The inner products used in that algorithm are of type

$$\mathbf{a}^\dagger \mathbf{b} = \sum_{i=1}^{M} a(x_i) b(x_i) \approx \int_a^b a(x) b(x) dx, \tag{1.46}$$

Obviously, the larger M is, the better the approximation.

1.3 Iterative methods for (non-)linear problems

The direct methods discussed in Secs. 1.1 and 1.2 solve the linear problem $\mathbf{Ax} = \mathbf{b}$ within a predetermined number of steps. For dense matrices, where most of the matrix elements are non-zero, that number is proportional to N^3, where N is the dimension of the matrix. It means that these methods become computationally too expensive for large N.

Alternatives for solving the linear problem $\mathbf{Ax} = \mathbf{b}$ are the iterative methods. They start with an approximate solution $\mathbf{x}^{(0)}$, and cycle to improve the solution $\mathbf{x}^{(0)} \to \mathbf{x}^{(1)} \to ... \to \mathbf{x}^{(n)}$, in the hope that one converges to the solution $\mathbf{x}$ within a tolerably small number of iterations n. This is an old problem, and many of the classical algorithms bear the names of famous mathematicians/physicists of bygone eras, see Sec. 1.3.2. Once you have decided to iterate, there is actually not such a great distinction anymore between solving a linear problem and solving a non-linear one, $\mathbf{Ax} = \mathbf{b}(\mathbf{x})$. So in the following I will start the story from the general perspective of the non-linear problem, see Sec. 1.3.1, and treat the linear problem as a special case.

One main problem of iterative methods is that they are not guaranteed to converge to a solution. Mathematicians have established conditions for the matrix $\mathbf{A}$ that are sufficient for convergence of specific algorithms (diagonal dominance, for instance), but more often than not, the matrix you are interested in, does not obey these conditions. In many cases, they have also figured out conditions that are necessary for specific algorithms to converge, but frequently you cannot check these conditions on forehand (because this would involve an elaborate calculation on itself). So unfortunately, applying iterative methods in practice can be a matter of trial and error. In addition, many of the classical methods discussed below, can have the drawback that, when they converge, their convergence is too slow to your liking. To alleviate such problems, you can turn to the subspace methods discussed in Sec. 1.4.

1.3.1 Fixed-point iteration

We start from the non-linear problem

$$\mathbf{Ax} = \mathbf{b}(\mathbf{x}), \tag{1.47}$$

where, as before, $\mathbf{A}$ is a square $N \times N$ matrix, $\mathbf{x}$ and $\mathbf{b}$ are column vectors of length N, and $b_i(x_1, x_2, ..., , x_N)$; $i = 1, 2, ..., N$ are known functions. Non-linear problems can be notoriously difficult to solve. Many algorithms start from treating them as linear problems, discovering the error made by that treatment, repeating the same mistake, discovering the error again, and so on. It is unreasonable to assume that such a procedure leads to a result, but surprisingly often it does.[18]

The simplest technique for solving the non-linear problem of Eq. 1.47 is simply to iterate on solving a linear problem. Start with a guess $\mathbf{v}$ of the solution and then

$$\mathbf{x}_0 = \mathbf{v};\ \text{do } n = 1, ...\ \text{ solve } \mathbf{A}\mathbf{x}_n = \mathbf{b}(\mathbf{x}_{n-1});\ \text{ until converged.} \tag{1.48}$$

One hopes that upon iterating, one converges to a point $\mathbf{x}$, which is then a solution to Eq. 1.47. This point $\mathbf{x}$ that solves Eq. 1.47 is called a *fixed point*, and Eq. 1.48 is an example of a *fixed-point iteration* algorithm. The feasibility of the algorithm hangs on how easy it is to solve the linear problem in the loop. If we use LU decomposition, Eqs. 1.4-1.7, the factorization, $\mathbf{A} = \mathbf{LU}$, has to be done only once. We can do it before the loop, and store the factors $\mathbf{L}$ and $\mathbf{U}$. The linear problem inside the loop can then be solved by forward substitution and back substitution, Eqs. 1.8 and 1.9.

Convergence criterion. The phrase "until converged" marks a criterion such as

$$\frac{\|\mathbf{x}_n - \mathbf{x}_{n-1}\|}{\|\mathbf{x}_n\|} < \epsilon, \tag{1.49}$$

with ϵ a tolerance set by the user. Alternatively, one can let the target vector determine the convergence,

$$\frac{\|\mathbf{b}(\mathbf{x}_n) - \mathbf{b}(\mathbf{x}_{n-1})\|}{\|\mathbf{b}(\mathbf{x}_n)\|} < \epsilon. \tag{1.50}$$

Obeying a criterion such as Eq. 1.49, indicates $\mathbf{x}_n \approx \mathbf{x}_{n-1}$, which either means convergence, or non-convergence if the algorithm gets stuck. Therefore, it is always safe to check how well the equation is actually solved by the end result, e.g., check

$$\frac{\|\mathbf{A}\mathbf{x}_n - \mathbf{b}(\mathbf{x}_n)\|}{\|\mathbf{b}(\mathbf{x}_n)\|} < \epsilon. \tag{1.51}$$

Note that for the algorithm of Eq. 1.48 the two criteria Eq. 1.50 and 1.51 are actually identical.

Instead of the global criterion of Eq. 1.49, one can also use a stricter local criterion, such as

$$\max_{i=1,...,N} \left\{ \frac{|x_{i,n} - x_{i,n-1}|}{|x_{i,n}|} \right\} < \epsilon, \tag{1.52}$$

or a similarly modified Eq. 1.50.

Details: this criterion comes from $\max_{i=1,...,N} |a_i| \equiv \|\mathbf{a}\|_\infty$, which is called the ∞-norm by mathematicians. Mathematicians define a p-norm by

$$\|\mathbf{a}\|_p = \sqrt[p]{|a_1|^p + |a_2|^p + ... + |a_N|^p}. \tag{1.53}$$

[18] "The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore, all progress depends on the unreasonable man." George Bernard Shaw, *Man and Superman*, (1903)

The p-norm obeys the mathematical criteria for a norm for any real number $p \geq 1$. The euclidean norm is given by $p = 2$, which is normally the one we use in physics. I usually write $\|\mathbf{a}\|_2 \equiv \|\mathbf{a}\|$, if no confusion can arise. The ∞-norm is defined by

$$\|\mathbf{a}\|_\infty = \lim_{p\to\infty} \|\mathbf{a}\|_p = \max_{i=1,\ldots,N} |a_i| \,. \tag{1.54}$$

Sometimes a physical property of interest, such as the energy $E(\mathbf{x}_n)$, dictates what the convergence criterion should be, provided it can be calculated easily. This makes good physical sense; if one is interested in the energy, then one should make sure that that is the property that should be converged.

1.3.1.1 Convergence

The algorithm of Eq. 1.48 is not guaranteed to converge to a solution of the problem, Eq. 1.47. A sufficient condition is

$$\|\mathbf{b}(\mathbf{y}) - \mathbf{b}(\mathbf{z})\| \leq K \|\mathbf{A}\mathbf{y} - \mathbf{A}\mathbf{z}\| \quad \text{for all } \mathbf{y}, \mathbf{z}, \text{ with } K < 1 \text{ constant.} \tag{1.55}$$

Details: This is called the *Banach fixed-point theorem* in mathematics. I skip its formal proof, but give you a flavor of what it does. Eq. 1.55 ensures that $\|\mathbf{A}\mathbf{x}_{n+1} - \mathbf{A}\mathbf{x}_n\| = \|\mathbf{b}(\mathbf{x}_n) - \mathbf{b}(\mathbf{x}_{n-1})\| \leq K \|\mathbf{A}\mathbf{x}_n - \mathbf{A}\mathbf{x}_{n-1}\|$, according to Eq. 1.48, so $\|\mathbf{A}\mathbf{x}_{n+1} - \mathbf{A}\mathbf{x}_n\| \leq K^n \|\mathbf{A}\mathbf{x}_1 - \mathbf{A}\mathbf{x}_0\|$. As $K < 1$, this means that $\|\mathbf{A}\mathbf{x}_{n+1} - \mathbf{A}\mathbf{x}_n\|$ goes to zero for large n, indicating that a limit point $\mathbf{A}\mathbf{x}$ is reached. The series $\mathbf{A}\mathbf{x}_0, \mathbf{A}\mathbf{x}_1, ...$, is called a *Cauchy sequence*. In Hilbert spaces, which are the spaces physicists typically use, such a sequence has a limit point. This means we have solved Eq. 1.47.

We can find a conservative estimate of how fast an algorithm reaches convergence if Eq. 1.55 holds. Call $e_n = \|\mathbf{x}_n - \mathbf{x}\|$ the "error", indicating how far we are from convergence. Then, multiplying the arguments of Eq. 1.55 left and right with $\mathbf{A}^{-1}$, we get $\|\mathbf{x}_n - \mathbf{x}\| = \|\mathbf{A}^{-1}\mathbf{b}(\mathbf{x}_{n-1}) - \mathbf{A}^{-1}\mathbf{b}(\mathbf{x})\| \leq K \|\mathbf{x}_{n-1} - \mathbf{x}\|$, which means

$$e_n \leq K e_{n-1}, \quad \text{with } K < 1 \text{ constant.} \tag{1.56}$$

This is called *linear convergence* in the literature. The phrase is somewhat misleading, as Eq. 1.56 is an exponential relationship. Iterating one has

$$e_n \leq K^n e_0 \quad \text{or} \quad \ln e_n \leq nK + \ln e_0. \tag{1.57}$$

So, *linear convergence* means that $\ln e_n$ is a linear function of the number of iterations n, and the error e_n itself is an *exponential function* of n. This is not so bad.

However, if you are solving large problems, one may look for algorithms that do better than that. Q*uadratic convergence* is defined by

$$e_n \leq C e_{n-1}^2, \quad \text{with } C < 1 \text{ constant.} \tag{1.58}$$

Iterating one has

$$e_n \leq (C' e_0)^{2^n} \quad \text{or} \quad \ln e_n \leq 2^n \ln C' e_0. \tag{1.59}$$

with $C' = C^2$. So, *quadratic convergence* means that $\ln e_n$ is an exponential function of the number of iterations n, and the error e_n itself is a *double exponential function* of n. This is fantastic, but very difficult to achieve.

Often, one is happy with *superlinear convergence*,

$$e_n \leq C e_{n-1}^{\alpha}, \quad \text{with } C < 1 \text{ constant and } 1 < \alpha < 2. \tag{1.60}$$

Iterating one has

$$e_n \leq (C' e_0)^{\alpha^n} \quad \text{or} \quad \ln e_n \leq \alpha^n \ln C' e_0, \tag{1.61}$$

with $C' = C^{\frac{\alpha}{\alpha-1}}$. Like quadratic convergence, *superlinear convergence* means that $\ln e_n$ is an exponential function of the number of iterations n, and the error e_n itself is a double exponential function of n.

One can monitor convergence by looking at $\|\mathbf{x}_n - \mathbf{x}_{n-1}\|$. For linear convergence, Eq. 1.56, one has

$$\ln \|\mathbf{x}_n - \mathbf{x}_{n-1}\| \leq n \ln K + \ln \left[\left(1 + K^{-1}\right) e_0\right]. \tag{1.62}$$

So plotting $\ln \|\mathbf{x}_n - \mathbf{x}_{n-1}\|$ versus n, one gets a straight line with slope $\ln K < 0$ (and offset $\ln\left[(1 + K^{-1})\, e_0\right]$). Superlinear or quadratic convergence go much faster. For large n one gets

$$\ln \|\mathbf{x}_n - \mathbf{x}_{n-1}\| \approx \alpha^n \ln (C' e_0). \tag{1.63}$$

with $\ln (C' e_0) < 0$.

Details: Eq. 1.61 is derived by writing $\ln e_n \leq \ln C + \alpha \ln e_{n-1} \leq \ln C + \alpha \ln C + \alpha^2 \ln e_{n-2}$, which iterating further gives $\ln e_n \leq \left(1 + \alpha^2 + ... + \alpha^n\right) \ln C + \alpha^n \ln e_0 = \frac{\alpha^{n+1}-1}{\alpha-1} \ln C + \alpha^n \ln e_0 \approx \frac{\alpha}{\alpha-1} \alpha^n \ln C + \alpha^n \ln e_0$ for large n. Using $\alpha = 2$ leads to Eq. 1.59. Eq. 1.62 is derived by writing $\|\mathbf{x}_n - \mathbf{x}_{n-1}\| = \|(\mathbf{x}_n - \mathbf{x}) - (\mathbf{x}_{n-1} - \mathbf{x})\| = \|\mathbf{e}_n - \mathbf{e}_{n-1}\|$, which gives $\|\mathbf{x}_n - \mathbf{x}_{n-1}\| \leq e_n + e_{n-1}$. Linear convergence gives $\ln \|\mathbf{x}_n - \mathbf{x}_{n-1}\| \leq \ln \left[\left(K^n + K^{n-1}\right) e_0\right]$.

1.3.1.2 Damping

If the criterion of Eq. 1.55 is not obeyed, convergence is not guaranteed, but not all is necessarily lost. One option is to try a simple modification of the algorithm of Eq. 1.48

$$\mathbf{x}_0 = \mathbf{v};\ \text{do } n = 1, ...\ \text{ solve } \mathbf{A}\mathbf{x}' = \mathbf{b}(\mathbf{x}_{n-1});\ \mathbf{x}_n = \mathbf{x}_{n-1} + \alpha_n \left(\mathbf{x}' - \mathbf{x}_{n-1}\right) \quad \text{until converged,} \tag{1.64}$$

where α_n is called the *damping parameter*.[19] Note that if $\alpha_n = 1$, we have the algorithm of Eq. 1.48 back, so typically one chooses $0 < \alpha_n < 1$. This means that we are not incorporating the whole correction $\Delta\mathbf{x}' = \mathbf{x}' - \mathbf{x}_{n-1}$ into the new vector $\mathbf{x}_n$, but only a fraction α_n of it. Combining Eqs. 1.55 and 1.64, it follows that this form of damping is going to work if

$$\|\mathbf{A}\mathbf{x}_n - \mathbf{A}\mathbf{x}_{n-1} - [\alpha_n \mathbf{r}(\mathbf{x}_n) - \alpha_{n-1}\mathbf{r}(\mathbf{x}_{n-1})]\| \leq K \|\mathbf{A}\mathbf{x}_n - \mathbf{A}\mathbf{x}_{n-1}\| \ \text{ with } K < 1, \tag{1.65}$$

with $\mathbf{r}(\mathbf{y}) = \mathbf{A}\mathbf{y} - \mathbf{b}(\mathbf{y})$ the *residual vector*. In principle, one can try to find an optimal damping parameter at each step. We will take $\alpha_n = \alpha$ fixed. If that does not work, we leave it at that, and choose a different method.

[19] or *mixing parameter* or *relaxation parameter* in some of the literature.

1.3.2 The classical methods

Textbooks typically begin with a survey of classical iterative algorithms for solving (non)linear problems. I will briefly discuss some of these from the point-of-view of fixed-point iteration. By modern standards, the Jacobi and Gauss-Seidel methods, and their variants, such as SOR, often converge too slowly, if they converge at all. To accelerate convergence, these methods are frequently used in combination with techniques such as subspace iteration, Sec. 1.4.

The Newton-Raphson technique works fantastically if it converges, but obtaining convergence can be tricky. Its major drawback is that one needs to calculate additional information, which may or may not be feasible.

The philosophy. A number of algorithms start from modifying the problem of Eq. 1.47 as

$$(\mathbf{A} - \mathbf{A}')\,\mathbf{x} = \mathbf{b}(\mathbf{x}) - \mathbf{A}'\mathbf{x}, \tag{1.66}$$

with $\mathbf{A}'$ a "suitable" matrix. Fixed-point iteration is then performed as in Eq. 1.48 with a correspondingly modified algorithm, in the hope that the convergence condition

$$\|\mathbf{b}(\mathbf{y}) - \mathbf{b}(\mathbf{z}) - \mathbf{A}'(\mathbf{y}-\mathbf{z})\| \le K\,\|(\mathbf{A}-\mathbf{A}')(\mathbf{y}-\mathbf{z})\| \quad \text{for all } \mathbf{y},\mathbf{z}, \text{ with } K<1, \tag{1.67}$$

is more readily obeyed than Eq. 1.55. A couple of popular algorithms define a suitable matrix $\mathbf{A}'$ by starting from writing the original matrix $\mathbf{A}$ as a sum of its strictly lower triangular part, its diagonal part, and its strictly upper diagonal part

$$\begin{aligned}&\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}; \quad \text{with} \quad L_{ij} = A_{ij},\ j<i;\ U_{ij} = A_{ij},\ j>i, \quad \text{and} \quad D_{ii} = A_{ii}, \\ &\text{and} \quad L_{ij} = 0,\ j \ge i;\ U_{ij} = 0,\ j \le i, \quad \text{and} \quad D_{ij} = 0,\ j \ne i.\end{aligned} \tag{1.68}$$

1.3.2.1 Jacobi iteration

One choice, ascribed to Jacobi, is $\mathbf{A}' = \mathbf{A} - \mathbf{D} = \mathbf{L} + \mathbf{U}$, so $\mathbf{A} - \mathbf{A}' = \mathbf{D}$. The algorithm of Eq. 1.48, modified to solve Eq. 1.66, then becomes

$$\mathbf{x}_0 = \mathbf{v};\ \text{do } n = 1, \ldots\ \mathbf{r}_{n-1} = \mathbf{A}\mathbf{x}_{n-1} - \mathbf{b}(\mathbf{x}_{n-1});\ \mathbf{x}_n = \mathbf{x}_{n-1} - \mathbf{D}^{-1}\mathbf{r}_{n-1};\quad \text{until converged.} \tag{1.69}$$

This is called *Jacobi iteration*. Note the inverse $\mathbf{D}^{-1}$ of a diagonal matrix is easily calculated. If $D_{ij} = \delta_{ij} d_{ii}$, then $D^{-1}_{ij} = \delta_{ij}/d_{ii}$. This means that one avoids having to do a LU factorization of the matrix $\mathbf{A}$, as in the algorithm of Eq. 1.48. Such a factorization is cheap for a tridiagonal matrix, but becomes expensive for a large dense matrix.

In fact, in such a case, Jabobi iteration can also be used to iteratively solve the *linear* problem $\mathbf{A}\mathbf{x} = \mathbf{b}(\mathbf{x}) = \mathbf{b}_0$, with $\mathbf{b}_0$ a fixed vector. For small matrices $\mathbf{A}$ an iterative method can usually not compete in speed and efficiency with the direct methods discussed in Secs. 1.1 and 1.2, but for large matrices it may become the only viable option. For a linear problem, the convergence condition of Eq. 1.67 becomes

$$\|\mathbf{A}'(\mathbf{y}-\mathbf{z})\| \le K\,\|(\mathbf{A}-\mathbf{A}')(\mathbf{y}-\mathbf{z})\| \quad \text{for all } \mathbf{y},\mathbf{z}, \text{ with } K<1, \tag{1.70}$$

which in case of Jacobi iteration can be rewritten as

$$\left\|(\mathbf{I}-\mathbf{D}^{-1}\mathbf{A})(\mathbf{y}-\mathbf{z})\right\| \leq K\left\|\mathbf{y}-\mathbf{z}\right\| \quad \text{for all } \mathbf{y},\mathbf{z}, \text{ with } K<1. \tag{1.71}$$

with $\mathbf{I}$ the identity matrix. Using Eq. 1.12, this then means

$$\left\|\mathbf{I}-\mathbf{D}^{-1}\mathbf{A}\right\| \leq K, \quad \text{with } K<1. \tag{1.72}$$

If this is the case, then $\mathbf{D}^{-1}$ is called an *approximate inverse* of $\mathbf{A}$.[20] Of course, not for all matrices $\mathbf{A}$ will its diagonal part $\mathbf{D}$ yield an approximate inverse. If not, then Jabobi iteration generally does not converge. If it does not converge for a linear problem, then you are asking for trouble if you want to apply it to a related non-linear problem.

Details: One can prove that if the matrix $\mathbf{A}$ is *strictly diagonally dominant*, i.e., if $|A_{ii}| > \sum_{j=1,j\neq i}^{N}|A_{ij}|$, then $\mathbf{D}^{-1}$ is indeed an approximate inverse of $\mathbf{A}$. Unfortunately, most of the matrices encountered in physics are not diagonally dominant. On the bright side, this is a sufficient condition, but not a necessary one. Even if $\mathbf{A}$ is not diagonally dominant, Jacobi iteration may converge. However, one feels intuitively that, if the diagonal does not play a dominant role, then Jacobi is in trouble.

We don't need this right now, but as a reference later on. For Hermitian matrices, one may write Eq. 1.72 also in terms of a spectral radius

$$\left\|\mathbf{I}-\mathbf{D}^{-1}\mathbf{A}\right\| = \left\|\mathbf{D}^{-1}\left[\mathbf{L}+\mathbf{L}^{\dagger}\right]\right\| = \rho\left(\mathbf{D}^{-1}\left[\mathbf{L}+\mathbf{L}^{\dagger}\right]\right) \leq K, \tag{1.73}$$

where we have used Eq. 1.68, the fact that for Hermitian matrices $\mathbf{U}=\mathbf{L}^{\dagger}$, and Eq. 1.17.

1.3.2.2 Gauss-Seidel iteration

Considering Eq. 1.68, an obvious modification of Jacobi is the choice $\mathbf{A}'=\mathbf{A}-\mathbf{D}-\mathbf{U}=\mathbf{L}$, so $\mathbf{A}-\mathbf{A}'=\mathbf{U}+\mathbf{D}$. The algorithm of Eq. 1.48, modified to solve Eq. 1.66, then becomes

$$\mathbf{x}_0=\mathbf{v};\ \text{do } n=1,\ldots \ \text{solve } (\mathbf{U}+\mathbf{D})\mathbf{x}_n=\mathbf{b}(\mathbf{x}_{n-1})-\mathbf{L}\mathbf{x}_{n-1};\ \ \text{until converged.} \tag{1.74}$$

This is called *Gauss-Seidel iteration*. The equation inside the loop, $(\mathbf{U}+\mathbf{D})\mathbf{y}=\mathbf{b}$, can easily be solved by back substitution, Eq. 1.9. It is possible to rewrite the line in the do-loop in the same form as in Eq. 1.69 to highlight the resemblance

$$\mathbf{x}_n=\mathbf{x}_{n-1}-(\mathbf{U}+\mathbf{D})^{-1}\,\mathbf{r}_{n-1}. \tag{1.75}$$

In practice, one of course avoids having to calculate the inverse explicitly, and one uses Eq. 1.74. Again, this means that one avoids a LU factorization of the matrix $\mathbf{A}$, which is beneficial if one deals with large matrices. In such a case, Gauss-Seidel iteration is also used to iteratively solve the *linear* problem $\mathbf{A}\mathbf{x}=\mathbf{b}_0$. The convergence criterion, Eq. 1.72, then becomes

$$\left\|\mathbf{I}-(\mathbf{U}+\mathbf{D})^{-1}\,\mathbf{A}\right\| \leq K, \quad \text{with } K<1, \tag{1.76}$$

where $(\mathbf{U}+\mathbf{D})^{-1}$ is now the approximate inverse of $\mathbf{A}$. Not for all matrices $\mathbf{A}$ will $\mathbf{U}+\mathbf{D}$ yield an approximate inverse, but intuitively one feels that the latter should do better than

[20] For the real inverse $\mathbf{A}^{-1}$ one has of course $\left\|\mathbf{I}-\mathbf{A}^{-1}\mathbf{A}\right\|=0$.

just **D** of the Jacobi case. Nevertheless, if Eq. 1.76 does not hold, then Gauss-Seidel iteration generally does not converge.

Interestingly, if **A** is positive definite, see Sec. 1.1.5, one can prove that Gauss-Seidel converges. Whereas by no means all matrices we encounter in physics are positive definite, there are some interesting problems that have positive definite matrices.

1.3.2.3 Successive over-relaxation (SOR)*

Both Gauss-Seidel iteration, as well as Jacobi iteration, can be combined with damping, Eq. 1.64, should convergence be troublesome. If convergence exists, but is too slow, it can be improved by *overdamping*, i.e., by choosing $\alpha > 1$. In the numerical mathematics folklore, the Gauss-Seidel method with overdamping is called *successive over-relaxation (SOR)*. Typical choices are $1.3 < \alpha < 1.8$. It has been applied with some success.

1.3.2.4 Newton-Raphson*

The last classical algorithm I am going to discuss is based upon the following. If $\mathbf{z}$ is not too far from $\mathbf{y}$, we can make a Taylor expansion

$$\mathbf{b}(\mathbf{z}) \approx \mathbf{b}(\mathbf{y}) + \mathbf{B}(\mathbf{y})\,(\mathbf{z}-\mathbf{y}) \quad \text{with matrix elements} \quad B_{ij}(\mathbf{y}) = \frac{\partial b_i}{\partial y_j}(\mathbf{y}). \tag{1.77}$$

One can use this Taylor expansion to approximate $\mathbf{b}(\mathbf{x}_n) \approx \mathbf{b}(\mathbf{x}_{n-1}) + \mathbf{B}(\mathbf{x}_{n-1})\,(\mathbf{x}_n - \mathbf{x}_{n-1})$, and solve $\mathbf{A}\mathbf{x}_n = \mathbf{b}(\mathbf{x}_n)$ using this approximation. Reordering the terms a little, this then gives the algorithm

$$\mathbf{x}_0 = \mathbf{c};\ \text{do } n = 1, \ldots \ \text{solve}\ \ (\mathbf{A} - \mathbf{B}(\mathbf{x}_{n-1}))\,\mathbf{x}_n = \mathbf{b}(\mathbf{x}_{n-1}) - \mathbf{B}(\mathbf{x}_{n-1})\mathbf{x}_{n-1};\quad \text{until converged.} \tag{1.78}$$

This is called the *Newton-Raphson* algorithm.[21] It is computationally more costly than any of the previous algorithms. At each iteration the matrix $\mathbf{B}(\mathbf{x}_{n-1})$ has to be calculated, and the matrix $(\mathbf{A} - \mathbf{B}(\mathbf{x}_{n-1}))$ has to be **LU** decomposed, Eq. 1.7. The feasibility of the algorithm very much hangs on how costly it is to calculate the partial derivatives, Eq. 1.77, and how costly it is to do the LU decomposition. In the problems we are going to look at, we assume that this overhead is computationally too costly, and we won't use Newton-Raphson. Unfortunately, this is typical for physical problems. However, you cannot brush aside the great man that easily, nor his algorithm. Newton-Raphson has spawned some very clever optimization algorithms, see the lecture notes on optimization.

Even if you can afford the computational costs, Newton-Raphson works only if the Taylor expansion of Eq. 1.77 is sufficiently accurate, or, in other words, if the steps $\|\mathbf{x}_n - \mathbf{x}_{n-1}\|$ are not too large. If that is the case, you stand a chance that Newton-Raphson converges. One can use damping, Eq. 1.64, to ensure that the steps stay sufficiently small. It can be shown that, if you are sufficiently close to the solution, Newton-Raphson converges quadratically, Eq. 1.58, which is of course fantastic. The catch lies in the phrase "sufficiently close". Newton-Raphson is sometimes combined with one of the other algorithms. The latter are

[21] Indeed, the great man himself. Raphson was a contemporary of Newton. Both formulated a similar algorithm, independent of one another.

used first to come "sufficiently close", before switching to Newton-Raphson for the final spurt to the solution.

1.4 Subspace methods for (non-)linear problems

The iterative procedures for solving (non-)linear problems we have looked at in the previous sections, have in common a propagation rule of the type

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{M}^{-1}\mathbf{r}_n, \quad \text{where } \mathbf{r}_n = \mathbf{A}\mathbf{x}_n - \mathbf{b}(\mathbf{x}_n), \tag{1.79}$$

is the *residual vector* (also called the *residual* for short, or sometimes *residue*[22]). The matrix $\mathbf{M}$ depends on the particular method used. In Jacobi iteration $\mathbf{M} = \mathbf{D}$, see Eq. 1.69, and in Gauss-Seidel iteration $\mathbf{M} = \mathbf{U} + \mathbf{D}$, see Eq. 1.75. Simple fixed-point iteration, Eq. 1.48, can be written as Eq. 1.79, with $\mathbf{M} = \mathbf{A}$, whereas Newton-Raphson, Eq. 1.78, can be written in this form with $\mathbf{M} = \mathbf{A} - \mathbf{B}(\mathbf{x}_{n-1})$. In all cases, the vector $\mathbf{M}^{-1}\mathbf{r}_n$ is called the *scaled residual vector.*[23]

Writing all these algorithms in the form of Eq. 1.79 stresses that the correction vector $\Delta\mathbf{x}_n = \mathbf{x}_{n+1} - \mathbf{x}_n$ is a function only of the last point before the step

$$\Delta\mathbf{x}_n = -\mathbf{M}^{-1}\mathbf{r}(\mathbf{x}_n). \tag{1.80}$$

I call such an algorithm a *one-step memory* technique. Convergence is signaled by the step sizes $\|\Delta\mathbf{x}_n\|$ getting smaller and smaller, or, in other words, by the sizes of the scaled residual vector $\|\mathbf{M}^{-1}\mathbf{r}(\mathbf{x}_n)\|$ getting smaller and smaller.

It is an interesting idea to keep a memory of all previous steps, $\Delta\mathbf{x}_m$; $m = n, n-1, n-2, ..., 1$, and use the knowledge gained in all these steps, before deciding on the new step forward. This is the idea behind *subspace techniques.* One uses the vectors $\Delta\mathbf{x}_m$ to build a n-dimensional linear vector space called the subspace. In this subspace one constructs a vector $\boldsymbol{\delta} = \sum_{m=1}^{n} b_m \Delta\mathbf{x}_m$, with coefficients b_m chosen such, that $\boldsymbol{\delta}$ is the best possible correction vector for the next step, so $\mathbf{x}_{n+1} = \mathbf{x}_n + \boldsymbol{\delta}$. Each iteration the dimension of the subspace is increased by one.[24] Hopefully this process converges before the dimension n becomes too large. Convergence is signaled by $\|\Delta\mathbf{x}_n\| = \|\mathbf{x}_{n+1} - \mathbf{x}_n\|$ becoming small.

There is some overhead involved in subspace techniques, in particular in finding the optimal coefficients b_m, but in general they converge much faster than one-step memory techniques. Subspace methods are the state of the art in solving (non-)linear problems of a large dimension, and are still actively developed and improved. There are many algorithms for subspace iteration to solve large-scale (non-)linear problems. One important class of such methods, the so-called quasi-Newton or Broyden methods are inspired by Newton-Raphson, and are discussed in the lecture notes on optimization. Conjugate gradient methods, which are applied to solve a special (but important) kind of (non-)linear problems, are also discussed in those notes.

Here, I pick out an algorithm that is popular among physicists and chemists. It is called Pulay iteration or direct inversion of the iterative subspace (DIIS), see Sec. 1.4.1

[22] Not to be mistaken for the same phrase used in complex function theory. It has nothing to do with that.
[23] Or, preconditioned residual vector, in some of the literature.
[24] This is the simplest way to build a subspace. For refinements, see Sec. 1.4.4

[8, 9, 10]. Among mathematicians this technique is called Anderson acceleration [11, 12].[25] This method turns out to be very versatile (it is also used in solving large-scale eigenvalue problems, for instance), and it is easily implemented. The original Pulay algorithm [8], Sec. 1.4.1, involves some overhead, and sometimes becomes unstable. It is possible to modify it such, that the overhead is less, and its stability increases. I call it orthogonalized Pulay iteration [12], see Sec. 1.4.3. Despite its venerable age, the Pulay/DIIS/Anderson technique has not made a prominent entrance into textbooks yet.

The Pulay algorithm can be combined with various algorithms for solving non-linear problems, Sec. 1.3.2, and is thus frequently called a technique for *convergence acceleration*. This gives it too little credit in my opinion. Yes, Pulay iteration can give you fast, quadratic convergence, but, more importantly, Pulay iteration can give you convergence in case the "classical" techniques diverge. An example of that will be shown in one of the physical examples in Part II. For the diehards, I will also briefly indicate the connection of Pulay iteration to GMRES, see Sec. 1.4.5 [3]. The latter is currently one of the most popular iterative algorithms in numerical libraries for solving linear problems, .

1.4.1 Pulay iteration or DIIS

In a subspace method one does not simply take $\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x}_n$ as the next iterate, Eqs. 1.79 and 1.80, but one keeps a memory of the previous steps. Consider an intermediate vector

$$\tilde{\mathbf{x}}_n = \mathbf{x}_n + \sum_{m=1}^{n-1} a_m \Delta\mathbf{x}_m. \tag{1.81}$$

The optimal coefficients, a_m; $m = 1, ..., n$, can be found by minimizing the size of the scaled residual vector $\|\Delta\tilde{\mathbf{x}}_n\|$

$$\Delta\tilde{\mathbf{x}}_n = -\mathbf{M}^{-1}\mathbf{r}(\tilde{\mathbf{x}}_n). \tag{1.82}$$

As $\mathbf{r}(\mathbf{x})$ is a non-linear function, Eq. 1.79, this minimization job is likely to become too time-consuming to be practical, in particular as one has to do it each iteration.

We will make an approximation. Start by rewriting Eq. 1.81 as

$$\tilde{\mathbf{x}}_n = \sum_{m=1}^{n} c_m \mathbf{x}_m, \tag{1.83}$$

with the constraint

$$\sum_{m=1}^{n} c_m = 1. \tag{1.84}$$

Proof: use $\Delta\mathbf{x}_m = \mathbf{x}_{m+1} - \mathbf{x}_m$ in Eq. 1.81. Then $\tilde{\mathbf{x}}_n = (1 + a_{n-1})\,\mathbf{x}_n + \sum_{m=2}^{n-2}(a_{m-1} - a_m)\,\mathbf{x}_m - a_1\mathbf{x}_1$. This can be rewritten in the form of Eq. 1.83, with $\sum_{m=1}^{n} c_m = 1 + a_{n-1} + \sum_{m=2}^{n-1}(a_{m-1} - a_m) - a_1 = 1$.

---

[25] The mathematicians are probably correct in naming the technique after Anderson, as he seems to have been the first one to formulate it. To keep in line with the physics/chemistry literature, I will stick to calling the algorithm Pulay iteration or DIIS. The mathematicians have their own literature. They make sure it is not very accessible to non-mathematicians, which contributes to their algorithms getting reinvented multiple times.

The approximation we will use is

$$\mathbf{r}(\tilde{\mathbf{x}}_n) \approx \bar{\mathbf{r}}_n = \frac{\sum_{m=1}^{n} c_m \mathbf{r}(\mathbf{x}_m)}{\sum_{m=1}^{n} c_m} = \sum_{m=1}^{n} c_m \mathbf{r}(\mathbf{x}_m). \tag{1.85}$$

In other words, we approximate $\mathbf{r}(\tilde{\mathbf{x}}_n)$ by the average (or the linear interpolation, if you prefer) of all $\mathbf{r}(\mathbf{x}_m)$, $m \leq n$.[26] Using this approximation in Eq. 1.82, instead of minimizing $\|\Delta\tilde{\mathbf{x}}_n\|$, one then minimizes $\|\boldsymbol{\Delta}\|$, where

$$\boldsymbol{\Delta} = -\mathbf{M}^{-1}\bar{\mathbf{r}}_n = \sum_{m=1}^{n} c_m \Delta\mathbf{x}_m, \tag{1.86}$$

and we have used Eqs. 1.85 and 1.80. Note that in optimizing the coefficients c_m; $m = 1, ..., n$, we have to obey the constraint of Eq. 1.84. How this is done, is explained in the next section. The new iterate is then expressed analogous to Eq. 1.79 as

$$\mathbf{x}_{n+1} = \tilde{\mathbf{x}}_n - \mathbf{M}^{-1}\bar{\mathbf{r}}_n = \tilde{\mathbf{x}}_n + \boldsymbol{\Delta} = \sum_{m=1}^{n} c_m \left(\mathbf{x}_m + \Delta\mathbf{x}_m\right) \tag{1.87}$$

Details: An alternative way to see that the constraint of Eq. 1.84 is reasonable, is to write $\mathbf{x}_m + \Delta\mathbf{x}_m = \mathbf{x} + \mathbf{e}_m$, where $\mathbf{x}$ is the exact solution, and $\mathbf{e}_m$ is the exact error vector. We then have

$$\mathbf{x}_{n+1} = \mathbf{x} \sum_{m=1}^{n} c_m + \sum_{m=1}^{n} c_m \mathbf{e}_m. \tag{1.88}$$

Iterating and optimizing, we should try and get the second term on the right-hand side to zero. In order for the first term to approach the exact solution $\mathbf{x}$, Eq. 1.84 must hold.

The *Pulay iteration* algorithm is

$$\mathbf{x}_1 = \mathbf{v};\ \mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - \mathbf{b}(\mathbf{x}_1);\ \mathbf{X}_0 = \text{nil};\ \mathbf{Y}_1 = \mathbf{x}_1; \tag{1.89}$$

$$\text{do } n = 1, \ldots \ ;\ \text{solve } \mathbf{A}\Delta\mathbf{x}_n = -\mathbf{r}_n;\ \mathbf{X}_n = (\mathbf{X}_{n-1}\ \Delta\mathbf{x}_n)\,; \tag{1.90}$$

$$\mathbf{K}_n = \mathbf{X}_n^{\dagger}\mathbf{X}_n;\ \mathbf{c}_n = \text{Pulay minimize } \mathbf{K}_n; \tag{1.91}$$

$$\mathbf{x}_{n+1} = (\mathbf{X}_n + \mathbf{Y}_n)\,\mathbf{c}_n;\ \mathbf{Y}_{n+1} = (\mathbf{Y}_n\ \mathbf{x}_{n+1})\,; \tag{1.92}$$

$$\mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \mathbf{b}(\mathbf{x}_{n+1});\quad \text{until converged.} \tag{1.93}$$

The convergence criterion is something like $\|\mathbf{r}_{n+1}\| < \epsilon$, or $\max_{i=1,\ldots,N} |(r_{n+1})_i| < \epsilon$, with ϵ a tolerance set by the user. Initialization is done in Eq. 1.89, and estimating the next correction vector $\Delta\mathbf{x}_n$ is done in Eq. 1.90. That is mostly done by a step in the form of Eq. 1.79. You can use your own favorite prescription for this step; often the simplest approach is used, such as Jacobi, where $\mathbf{M} = \mathbf{D}$. Next the subspace is extended, constructing the $N \times n$ matrix $\mathbf{X}_n$ by appending the vector $\Delta\mathbf{x}_n$ (an $N \times 1$ matrix) as a column to the $N \times (n-1)$ matrix $\mathbf{X}_{n-1}$.

[26] Of course, if our problem is truly linear, then $\mathbf{r}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$, and the approximation is *exact*, see Eqs. 1.83 and 1.84. In that case Pulay iteration is equivalent to GMRES, see Sec. 1.4.5.

The $n \times n$ Pulay matrix $\mathbf{K}_n$ is constructed in Eq. 1.91. This can be done very efficiently, see the next section. The Pulay matrix is subsequently used in solving the constrained minimization problem to produce the optimal vector $\mathbf{c}_n$, which is a $n \times 1$ column vector, containing the optimal coefficients $c_m;\ m = 1, ..., n$. The vector $\mathbf{x}_{n+1}$ is constructed in Eq. 1.92, following the prescription of Eq. 1.87. Finally, the residual vector $\mathbf{r}_{n+1}$ is calculated, and convergence is checked in Eq. 1.93. We keep both sets of vectors $\mathbf{x}_m, \Delta\mathbf{x}_m;\ m = 1, ..., n$ in memory, collected in the $(N \times n)$ matrices $\mathbf{X}_n$ and $\mathbf{Y}_n$.

Pulay called his algorithm *direct inversion in the iterative subspace*, or DIIS for short, and as such it is also known in the literature.[27] It is versatile, because it can be combined with very different algorithms for solving non-linear problems. Those algorithms each have their own rule for estimating correction vectors $\Delta\mathbf{x}_n$. I have given a couple of examples in Sec. 1.3.2 where Eq. 1.90 is effectively substituted by an expression like Eq. 1.80. The algorithmic world is however much bigger than that. For instance, in self-consistent electronic structure problems the rule for generating correction vectors looks nothing like Eqs. 1.79 or 1.90. It does not matter, however; whatever the rule for generating correction vectors is, as long as it produces a $\Delta\mathbf{x}_n$, the Pulay algorithm can do its work.

Cost and stability If we stick to the linear problem of Eq. 1.90, solving it head on is a $O(N^3)$ calculation, which is very expensive if N is large. It can be avoided by using a simpler rule in the form of Eq. 1.79. Using Jacobi, for instance, makes it a $O(N)$ calculation, whereas Gauss-Seidel gives a $O(N^2)$ calculation. The Pulay minimization step, Eq. 1.91, can be implement such, that it is an $O(nN)$ operation for $N \gg n$, see Sec. 1.4.2. Likewise, the Pulay update step, Eq. 1.92 is $O(nN)$. In a special version of Pulay iteration, called limited memory Pulay, n is made a fixed (typically small, $n \lesssim 10$) number, which makes both of these steps $O(N)$, see Sec. 1.4.4. The most costly step is then Eq. 1.93, which is $O(N^2)$ if $\mathbf{A}$ is a full (dense) matrix. The cost of this step can be decreased to $O(N)$ if $\mathbf{A}$ is a sparse matrix. All these steps have to be done each iteration, which means that Pulay iteration as a whole is a $O(nN^2)$ calculation for a dense matrix $\mathbf{A}$, and a $O(n^2N)$ calculation for a sparse matrix $\mathbf{A}$.

Note that this is *not more expensive* than the one-step memory techniques discussed in Sec. 1.3.2. Some people therefore state that one should always use Pulay iteration or something comparable, and do away with one-step memory techniques all together.

The stability of Pulay iteration hangs on the stability of the Pulay minimization step, Eq. 1.91. In cases where this step fails, (re)building the subspace cleverly often solves the problem, see Sec. 1.4.2. The next section discusses a number of techniques for the Pulay minimization step.

1.4.2 Minimizing the Pulay vector

We still have to formulate how to minimize the size of the Pulay vector $\|\mathbf{\Delta}\|$, Eq. 1.86, under the constraint of Eq. 1.84, to find the optimal coefficients $c_m;\ m = 1, ..., n$. This step is really at the heart of Pulay iteration, and over the years several techniques have been developed of increasing ingenuity and sophistication. They all have in common that, instead of minimizing $\|\mathbf{\Delta}\|$, Eq. 1.86, they minimize $\|\mathbf{\Delta}\|^2$. The latter is easier, as that is a more-or-less standard least-squares problem.

[27] It is called *Anderson acceleration* in the mathematical literature, see the previous footnote.

The following section is too long, for which I apologize. I struggled a bit with the many formulations, and found no time to merge them elegantly into one. **Methods 1 and 2** are the most easily implemented versions within the standard Pulay iteration algorithm, Eqs. 1.84-1.93. **Method 5** can be integrated into a very stable version of Pulay, called orthogonalized Pulay iteration, which I will give in Sec. 1.4.3.

Method 1: constrained minimization. This is how Pulay originally formulated his technique [8]. The problem is that of minimizing the function

$$F(c_1, ..., c_n, \lambda) = \sum_{k=1}^{n} \sum_{m=1}^{n} c_k^* c_m \Delta \mathbf{x}_k^\dagger \Delta \mathbf{x}_m - \lambda \left(\sum_{k=1}^{n} c_k^* - 1 \right) - \lambda \left(\sum_{k=1}^{n} c_k - 1 \right), \tag{1.94}$$

where the double sum comes from $\|\mathbf{\Delta}\|^2$, see Eq. 1.86, and the constraint of Eq. 1.84 is introduced by means of a Lagrange multiplier λ (see the lecture notes on optimization). I have expressed the problem in complex numbers. If you don't like this, you may use real numbers $c_k^* = c_k$; the algorithm stays essentially the same.

Minimizing the function is done by solving the equations

$$\text{(i)} \ \frac{\partial F}{\partial c_k} = 0; \ \text{(ii)} \ \frac{\partial F}{\partial c_k^*} = 0; \ k = 1, ..., n; \ \text{(iii)} \ \frac{\partial F}{\partial \lambda} = 0. \tag{1.95}$$

The equations (i) are omitted in the following, because they give essentially the same thing as equations (ii). Eqs. 1.95 (ii) and (iii) then lead to the following linear problem of dimension $n+1$

$$\begin{pmatrix} K_{11} & K_{12} & \cdots & K_{1n} & -1 \\ K_{21} & K_{22} & \cdots & K_{2n} & -1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ K_{n1} & K_{n2} & \cdots & K_{nn} & -1 \\ -1 & -1 & -1 & -1 & 0 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \\ \lambda \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ -1 \end{pmatrix}, \quad \text{where } K_{km} = \Delta \mathbf{x}_k^\dagger \Delta \mathbf{x}_m; \tag{1.96}$$

$k = 1, ..., n; \ m = 1, ..., n$. This a small linear problem, as the number of iterations n is small, otherwise we are in serious trouble anyway. This linear problem can be comfortably solved using a standard technique, such as LU decomposition, see Sec. 1.1.1.[28] Solving the linear problem gives the optimal coefficients $c_m; \ m = 1, .., n$, which minimize the size squared of the residual vector, $\|\mathbf{\Delta}\|^2$.

Solving the linear problem also gives you the Lagrange multiplier λ, which has some meaning. Writing out the matrix-vector product of Eq. 1.96 gives

$$\text{(a)} \ \sum_{m=1}^{n} K_{km} c_m - \lambda = 0; \ k = 1, ..., n \ \text{and (b)} \ \sum_{m=1}^{n} c_m = 1 \tag{1.97}$$

Multiplying each of the n equations (a) with c_k^* and summing them, gives $\sum_{k=1}^{n} \sum_{m=1}^{n} c_k^* K_{km} c_m - \lambda \sum_{k=1}^{n} c_k^* = 0$. Using (b), this then gives $\sum_{k=1}^{n} \sum_{m=1}^{n} c_k^* K_{km} c_m = \lambda$, or

$$\lambda = \|\mathbf{\Delta}\|^2, \tag{1.98}$$

which means that the calculated λ gives the size squared of the minimal residual vector.

[28] As the matrix $\mathbf{K}$ is symmetric, one can use a slightly more efficient version of LU decomposition, and write $\mathbf{K} = \mathbf{L}\mathbf{D}\mathbf{L}^\dagger$, with $\mathbf{D}$ a diagonal matrix. This is called LDL decomposition. It costs roughly half as much as full LU decomposition, but it has the same stability issues.

Cost and stability. Solving the linear problem of Eq. 1.96 is $O((n+1)^3)$, which is tolerable as n is a small number. Constructing the matrix $\mathbf{K}$ is an $O(n^2N)$ process, as in involves the calculation of n^2 inner products $\Delta\mathbf{x}_k^\dagger\Delta\mathbf{x}_m$. If we do this each Pulay iteration, this would be the most costly step, as $N \gg n$. We can reduce the cost by keeping the $(n-1)\times(n-1)$ matrix of the previous iteration $\mathbf{K}^{(n-1)}$ in store, with matrix elements $K_{km}^{(n-1)} = \Delta\mathbf{x}_k^\dagger\Delta\mathbf{x}_m;\ k = 1, ..., n-1;\ m = 1, ..., n-1$. In constructing the $n\times n$ matrix $\mathbf{K}$, one only has to calculate $K_{nm} = \Delta\mathbf{x}_n^\dagger\Delta\mathbf{x}_m;\ m = 1, ..., n$, and $K_{mn} = K_{nm}^*$, and extend the matrix $\mathbf{K}^{(n-1)}$ by one row and one column. The cost per iteration of this step are then reduced to $O(nN)$.

If the Pulay matrix $\mathbf{K}$ in Eq. 1.96 is ill-conditioned (has a large condition number $\kappa(\mathbf{K})$), the solution to the linear problem becomes inaccurate, see Sec. 1.1.3. The Pulay matrix becomes ill-conditioned if the vectors $\Delta\mathbf{x}_m$ become close to linear dependence. One can try to avoid that problem by clever subspace building, see Sec. 1.4.4. Nonetheless, once convergence sets in, then inevitably the new $\Delta\mathbf{x}_n$ becomes an increasingly small vector, which means that the new row and column added to $\mathbf{K}^{(n-1)}$ consist of increasingly small numbers. This means that, once convergence sets in, $\mathbf{K}$ becomes closer to being singular, and the condition number increases, $\kappa(\mathbf{K}) > \kappa(\mathbf{K}^{(n-1)})$. $\mathbf{K}$ would of course become truly singular, if the row and column added would consist of zeros, but adding a row and column of almost-zero numbers also spells trouble. The closer to convergence we get, the more trouble of this sort we may expect.

In practice, one mostly finds an acceptable solution to the original (non-)linear problem before the linear-dependence problem of the Pulay matrix sets in, and the latter does not present a practical problem. Should the problem occur anyway, the use of orthogonalized Pulay can alleviate it.

Method 2: enforcing the constraint a posteriori. This is how Pulay's original technique is usually implemented. Firstly, ignore the constraint of Eq. 1.84 for the moment, and solve the unconstrained linear problem

$$\begin{pmatrix} K_{11} & K_{12} & \cdots & K_{1n} \\ K_{21} & K_{22} & \cdots & K_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ K_{n1} & K_{n2} & \cdots & K_{nn} \end{pmatrix} \begin{pmatrix} c'_1 \\ c'_2 \\ \vdots \\ c'_n \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}, \quad \text{where } K_{mk} = \Delta\mathbf{x}_m^\dagger\Delta\mathbf{x}_k; \tag{1.99}$$

$m = 1, ..., n;\ k = 1, ..., n$. Again, this linear problem can be solved using a standard technique, such as LU decomposition. In fact, as the matrix $\mathbf{K}$ is symmetric and positive definite, we can use Cholesky decomposition, see Sec. 1.1.5. Secondly, normalizing *a posteriori*

$$\lambda = 1/\sum_{m=1}^{n} c'_m \text{ and } c_m = \lambda c'_m;\ m = 1, ..., n, \tag{1.100}$$

then gives coefficients $c_m;\ m = 1, ..., n$ that obey Eq. 1.97(i) and (ii). In other words, we have solved the original problem, Eq. 1.96.

Cost and stability. The cost is very similar to that of method 1. Solving the linear problem of Eq. 1.99 is $O(n^3)$, and constructing $\mathbf{K}$ is $O(nN)$, when keeping a memory of the previous iteration. The stability hangs on the condition number $\kappa(\mathbf{K})$, so, in principle, it is

the same as of method 1. The main difference is that the linear problem of Eq. 1.99 involves a symmetric positive definite matrix, whereas in Eq. 1.96 this is not necessarily the case. To solve Eq. 1.99 one can use Cholesky decomposition, whereas to solve Eq. 1.96 one has to use the more general form of LU decomposition. Cholesky decomposition has the reputation of being much more stable, so most people prefer it.

Method 3: orthogonalization.* This is how Pulay's technique can be implemented using orthogonalization. The linear problem of Eq. 1.99 is a least-squares problem of the type discussed in Sec. 1.2.2. As explained in that section, one can try and improve the numerical stability by orthogonalization. The matrix $N \times n$ matrix $\mathbf{X}$ has columns $\Delta\mathbf{x}_m$; $m = 1, ...n$, Eq. 1.90. This matrix can be QR factorized, $\mathbf{QR} = \mathbf{X}$, Sec. 1.2, by Gram-Schmidt orthogonalization of the vectors $\Delta\mathbf{x}_m$; $m = 1, ...n$, Eq. 1.34. Here $\mathbf{Q}$ is a $N \times n$ matrix with orthonormal columns, $\mathbf{q}_k^\dagger \mathbf{q}_m = \delta_{km}$; $k = 1, ..., n$; $m = 1, ..., n$, and $\mathbf{R}$ is a $n \times n$ upper triangular matrix. As the $n \times n$ matrix $\mathbf{K} = \mathbf{X}^\dagger\mathbf{X}$, Eq. 1.91, we then have

$$\mathbf{K} = \mathbf{X}^\dagger\mathbf{X} = \mathbf{R}^\dagger\mathbf{R}. \tag{1.101}$$

Proof: $\mathbf{K} = \mathbf{X}^\dagger\mathbf{X}$, Eq. 1.91. Applying QR factorization gives $\mathbf{K} = (\mathbf{QR})^\dagger \mathbf{QR} = \mathbf{R}^\dagger\mathbf{Q}^\dagger\mathbf{QR} = \mathbf{R}^\dagger\mathbf{R}$. The last step follows from $\mathbf{Q}^\dagger\mathbf{Q} = \mathbf{I}_n$, with $\mathbf{I}_n$ the $n \times n$ identity matrix.

Eq. 1.99 can then be written as $\mathbf{Kc}' = \mathbf{R}^\dagger\mathbf{Rc}' = \mathbf{1}$ (where $\mathbf{1}$ is the column vector with n elements 1), which can be solved in two steps

$$(1)\ \mathbf{R}^\dagger\mathbf{d} = \mathbf{1};\quad (2)\ \mathbf{Rc}' = \mathbf{d}. \tag{1.102}$$

The matrix $\mathbf{R}$ is upper triangular, which means that the matrix $\mathbf{R}^\dagger$ is lower triangular. So, Eq. 1.102(1) is easily solved by forward substitution, and (2) is solved by back substitution. The two steps then solve Eq. 1.99. Followed by normalization, Eq. 1.100, we have then solved the original problem, Eq. 1.96.

Cost and stability. Solving the linear problem of Eq. 1.102 is an $O(n^2)$ process, so that is cheap. The QR factorization of the $N \times n$ matrix $\mathbf{X}$ is $O(n^2N)$, so this is the most costly step. We may, however, use a trick similar to that used in method 1 to reduce the costs. Suppose we have a QR factorization of the previous iteration $\mathbf{Q}^{(n-1)}\mathbf{R}^{(n-1)} = \mathbf{X}^{(n-1)}$. $\mathbf{X}$ can be constructed by adding one column to the $N \times (n-1)$ matrix $\mathbf{X}^{(n-1)}$, i.e. $\mathbf{X} = \left(\mathbf{X}^{(n-1)}\ \Delta\mathbf{x}_n\right)$. Similarly, the $N \times n$ matrix $\mathbf{Q}$ can be constructed from the $N \times (n-1)$ matrix $\mathbf{Q}^{(n-1)}$ by adding one column. This column can be obtained by Gram-Schmidt orthogonalizing the new vector $\Delta\mathbf{x}_n$ on the columns $\mathbf{q}_k$; $k = 1, ..., n-1$, of $\mathbf{Q}^{(n-1)}$

$$\begin{aligned} &\mathbf{q}_n = \Delta\mathbf{x}_n; \\ &\text{do } k = 1, ..., n-1 \quad R_{kn} = \mathbf{q}_k^\dagger\mathbf{q}_n;\ \mathbf{q}_n = \mathbf{q}_n - \mathbf{q}_k R_{kn}; \\ &R_{nn} = \|\mathbf{q}_n\|;\ \mathbf{q}_n = \mathbf{q}_n/R_{nn}; \end{aligned} \tag{1.103}$$

The vector $\mathbf{q}_n$ is then appended to $\mathbf{Q}^{(n-1)}$, i.e. $\mathbf{Q} = \left(\mathbf{Q}^{(n-1)}\ \mathbf{q}_n\right)$. At the same time, the $(n-1) \times (n-1)$ matrix $\mathbf{R}^{(n-1)}$ is extended to the $n \times n$ matrix $\mathbf{R}$ by adding one column R_{kn}; $k = 1, ..., n$, calculated in Eq. 1.103, and one row $R_{nm} = \delta_{nm}R_{nn}$; $m = 1, ..., n$. Eq.

1.103 is a $O(nN)$ calculation (essentially the calculation of n inner products $\mathbf{q}_k^\dagger \mathbf{q}_n$), which make the costs similar to that of methods 1 and 2.

The error in solving Eq. 1.102(1) depends on the condition number $\kappa(\mathbf{R}^\dagger) = \kappa(\mathbf{R})$, Sec. 1.1.3. Likewise the error in solving Eq. 1.102(2) depends on $\kappa(\mathbf{R})$. The total error is thus determined by $\kappa(\mathbf{R})^2$. This is comparable to methods 1 and 2, as in this case $\kappa(\mathbf{K}) = \kappa(\mathbf{R}^\dagger\mathbf{R}) = \kappa(\mathbf{R}^\dagger)\kappa(\mathbf{R}) = \kappa(\mathbf{R})^2$, see Eqs. 1.101 and 1.102, so there is no improvement of the condition number.

Method 4: enforcing the constraint a priori.* This is the Anderson formulation of the Pulay technique (more or less) [11, 12]. Rewrite Eq. 1.86 as

$$\boldsymbol{\Delta} = \Delta\mathbf{x}_n + \sum_{m=1}^{n-1} c_m \left(\Delta\mathbf{x}_m - \Delta\mathbf{x}_n\right). \tag{1.104}$$

Proof: $\boldsymbol{\Delta} = \sum_{m=1}^{n} c_m \Delta\mathbf{x}_m = c_n\Delta\mathbf{x}_n + \sum_{m=1}^{n-1} c_m\Delta\mathbf{x}_m = \Delta\mathbf{x}_n - (1-c_n)\Delta\mathbf{x}_n + \sum_{m=1}^{n-1} c_m\Delta\mathbf{x}_m = \Delta\mathbf{x}_n - \sum_{m=1}^{n-1} c_m\Delta\mathbf{x}_n + \sum_{m=1}^{n-1} c_m\Delta\mathbf{x}_m$, using Eq. 1.84.

The coefficients $c_m;\ m = 1, ..., n-1$ in Eq. 1.104 can now be optimized without using any constraint, as per definition $c_n = 1 - \sum_{m=1}^{n-1} c_m$.

The function to minimize is now

$$\|\boldsymbol{\Delta}\|^2 = G(c_1, ..., c_{n-1}) = \sum_{k=1}^{n-1}\sum_{m=1}^{n-1} c_k^* c_m \Delta'\mathbf{x}_k^\dagger \Delta'\mathbf{x}_m - \sum_{k=1}^{n-1}\left(c_k \Delta\mathbf{x}_n^\dagger \Delta'\mathbf{x}_k + c_k^* \Delta'\mathbf{x}_k^\dagger \Delta\mathbf{x}_n\right), \tag{1.105}$$

with

$$\Delta'\mathbf{x}_m = \Delta\mathbf{x}_m - \Delta\mathbf{x}_n. \tag{1.106}$$

Setting $\partial G/\partial c_k^* = 0;\ k = 1, ..., n-1$ gives the equations

$$\begin{pmatrix} V_{11} & V_{12} & \cdots & V_{1,n-1} \\ V_{21} & V_{22} & \cdots & V_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ V_{n-1,1} & V_{n-1,2} & \cdots & V_{n-1,n-1} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \end{pmatrix}, \tag{1.107}$$

where $V_{km} = \Delta'\mathbf{x}_k^\dagger \Delta'\mathbf{x}_m$ and $v_m = -\Delta'\mathbf{x}_m^\dagger \Delta\mathbf{x}_n;\ k = 1, ..., n-1;\ m = 1, ..., n-1$.

It is possible to solve this linear problem in a way that is completely analogous to method 3 above. First form the $N \times (n-1)$ matrix $\mathbf{X}' = (\Delta'\mathbf{x}_1\ \Delta'\mathbf{x}_2 ... \Delta'\mathbf{x}_{n-1})$, which has $\Delta'\mathbf{x}_m;\ m = 1, ..., n-1$ as its columns. Then QR factorize $\mathbf{Q}'\mathbf{R}' = \mathbf{X}'$ with $\mathbf{Q}'$ a $N \times (n-1)$ matrix with orthonormal columns, and $\mathbf{R}'$ a $(n-1) \times (n-1)$ upper-triangular matrix. Then $\mathbf{V} = \mathbf{X}'^\dagger\mathbf{X}' = \mathbf{R}'^\dagger\mathbf{R}'$ and $\mathbf{v} = -\mathbf{X}'^\dagger\Delta\mathbf{x}_n = -\mathbf{R}'^\dagger\mathbf{Q}'^\dagger\Delta\mathbf{x}_n$, so the linear problem of Eq. 1.107 can be written as

$$\mathbf{V}\mathbf{c} = \mathbf{v} \Leftrightarrow \mathbf{R}'^\dagger\mathbf{R}'\mathbf{c} = -\mathbf{R}'^\dagger\mathbf{Q}'^\dagger\Delta\mathbf{x}_n. \tag{1.108}$$

We might as well drop the $\mathbf{R}'^\dagger$ left and right of this equation, and write

$$\mathbf{R}'\mathbf{c} = -\mathbf{Q}'^\dagger\Delta\mathbf{x}_n. \tag{1.109}$$

This can be solved by back transformation, as $\mathbf{R}'$ is an upper triangular matrix.

Starting from Eq. 1.104, and the definitions in the previous paragraph, we can write $\mathbf{\Delta} = \Delta\mathbf{x}_n + \mathbf{X}'\mathbf{c} = \Delta\mathbf{x}_n + \mathbf{Q}'\mathbf{R}'\mathbf{c}$. Using Eq. 1.109 we then get

$$\mathbf{\Delta}_{\min} = \Delta\mathbf{x}_n - \mathbf{Q}'\mathbf{Q}'^{\dagger}\Delta\mathbf{x}_n = \Delta\mathbf{x}_n - \sum_{m=1}^{n-1} \mathbf{q}'_m \left(\mathbf{q}'^{\dagger}_m \Delta\mathbf{x}_n\right), \tag{1.110}$$

where $\mathbf{q}'_m;\ m = 1, ..., n-1$ are the orthonormal columns of $\mathbf{Q}'$. They span the same space as the (generally non-orthogonal) vectors $(\Delta\mathbf{x}_m - \Delta\mathbf{x}_n);\ m = 1, ..., n-1$. The last term on the right-hand side of Eq. 1.110 represents the projection of $\Delta\mathbf{x}_n$ onto that space, and $\mathbf{\Delta}_{\min}$ is the *orthogonal projection* of $\Delta\mathbf{x}_n$ perpendicular to that space. Note that

$$\mathbf{q}'^{\dagger}_k \mathbf{\Delta}_{\min} = 0,\ k = 1, ..., n-1. \tag{1.111}$$

Minimizing $\|\mathbf{\Delta}\|$ can then be given a simple graphical interpretation, as is illustrated by Fig. 1.2. The vector $\mathbf{\Delta}_{\min}$ with minimal $\|\mathbf{\Delta}\|$ is given by the orthogonal projection of $\Delta\mathbf{x}_n$ perpendicular to the hyperplane spanned by the vectors $(\Delta\mathbf{x}_m - \Delta\mathbf{x}_n);\ m = 1, ..., n-1$. The vectors $\mathbf{q}'_k;\ k = 1, ..., n-1$ form an orthonormal set that span this hyperplane.

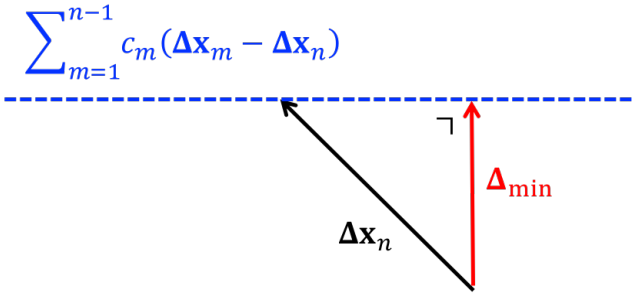


Figure 1.2: A graphical interpretation of Eq. 1.104. The vectors $\sum_{m=1}^{n-1} c_m (\Delta\mathbf{x}_m - \Delta\mathbf{x}_n)$ describe a $n-1$ dimensional hyperplane, and $\mathbf{\Delta} = \Delta\mathbf{x}_n + \sum_{m=1}^{n-1} c_m (\Delta\mathbf{x}_m - \Delta\mathbf{x}_n) = \sum_{m=1}^{n} c_m \Delta\mathbf{x}_m$ describes the hyperplane displaced by the vector $\Delta\mathbf{x}_n$. The vector $\mathbf{\Delta}_{\min}$ that has a minimum norm $\|\mathbf{\Delta}\|$, is given by the orthogonal projection of $\Delta\mathbf{x}_n$ perpendicular to that hyperplane.

Cost and stability.* Solving the linear problem of Eq. 1.109 is an $O(n^2)$ process, so that is negligibly cheap. Calculating the vector $\mathbf{Q}'^{\dagger}\Delta\mathbf{x}_n$ in Eq. 1.109 is $O(nN)$, so that is tolerable too. The QR factorization of the $N \times (n-1)$ matrix $\mathbf{X}'$ is $O(n^2 N)$, so this is the most costly step. In method 3, we made use of the matrices assembled in the previous iterations to bring that cost down to $O(nN)$. In the present method 4, however, we cannot use the same trick. The matrix $\mathbf{X}'$ changes with each new vector $\Delta\mathbf{x}_n$ in a non-trivial way, see Eq. 1.106, as $\mathbf{X}'$ is not just $\mathbf{X}'^{(n-1)}$ with one extra column added. This means we cannot easily make use of the $\mathbf{Q}'^{(n-1)}$ and $\mathbf{R}'^{(n-1)}$ matrices obtained in the previous iteration.

The stability of method 4 is better than that of methods 1-3, though. The error in solving the linear problem of Eq. 1.109 depends on the condition number $\kappa(\mathbf{R}')$. This is much better than for methods 1-3, where the condition numbers of the matrices involved in the corresponding problems are approximately the *square* of the present one, $\kappa(\mathbf{K}) = \kappa(\mathbf{R})^2$, see Eqs. 1.101 and 1.102. So method 4 promises to be more accurate and more stable. To become computationally competitive for very large problems, however, requires getting rid of the unfavorable $O(n^2 N)$ scaling.

Method 5: enforcing the constraint a priori, efficient implementation.* This is how Anderson's formulation (method 4) can also be made efficient. The main trick is to rewrite Eq. 1.104 as

$$\mathbf{\Delta} = \Delta\mathbf{x}_n + \sum_{m=1}^{n-1} a_m \left(\Delta\mathbf{x}_m - \Delta\mathbf{x}_{m+1}\right). \tag{1.112}$$

The coefficients a_m are related to the coefficients c_m of Eqs. 1.86 and 1.104 by

$$c_1 = a_1; \quad c_m = a_m - a_{m-1}; \; m = 2, ..., n-1; \quad c_n = 1 - a_{n-1}. \tag{1.113}$$

The coefficients a_m; $m = 1, ..., n-1$ can be varied independently, while the constraint of Eq. 1.84 is obeyed.

Proof: write in Eq. 1.104 $\Delta\mathbf{x}_m - \Delta\mathbf{x}_n = (\Delta\mathbf{x}_m - \Delta\mathbf{x}_{m+1}) + (\Delta\mathbf{x}_{m+1} - \Delta\mathbf{x}_{m+2}) + ... + (\Delta\mathbf{x}_{n-1} - \Delta\mathbf{x}_n)$. Insert in Eq. 1.112, then $a_m = \sum_{k=m}^{n-1} c_k$. We also need the reverse mapping; rewrite Eq. 1.112 as $\Delta\mathbf{x}_n - \sum_{m=1}^{n-1} a_m \Delta\mathbf{x}_{m+1} + \sum_{m=1}^{n-1} a_m \Delta\mathbf{x}_m = \Delta\mathbf{x}_n - \sum_{m=2}^{n} a_{m-1}\Delta\mathbf{x}_m + \sum_{m=1}^{n-1} a_m \Delta\mathbf{x}_m = a_1\Delta\mathbf{x}_1 + \sum_{m=2}^{n-1} (a_m - a_{m-1})\,\Delta\mathbf{x}_m + (1 - a_{n-1})\,\Delta\mathbf{x}_n$. This is equal to $\sum_{m=1}^{n} c_m \Delta\mathbf{x}_m$, with $c_1 = a_1$, $c_m = a_m - a_{m-1}$; $m = 2, ..., n-1$, $c_n = 1 - a_{n-1}$. Note that we have $\sum_{m=1}^{n} c_m = a_1 + \sum_{m=2}^{n-1} (a_{m-1} - a_m) + 1 - a_{n-1} = 1$. This means that we can choose a_m; $m = 1, ..., n-1$ arbitrarily, and obey the constraint.

The change from Eq. 1.104 to Eq. 1.112 can again be given a simple graphical interpretation, see Fig. 1.3. It is a simple transformation among the vectors describing the hyperplane. The hyperplane itself is not changed, which means that the vector $\mathbf{\Delta}_{\min}$ with minimal $\|\mathbf{\Delta}\|$ is still given by the orthogonal projection of $\Delta\mathbf{x}_n$ perpendicular to the hyperplane.

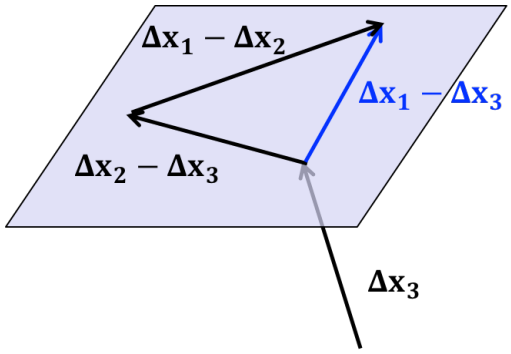


Figure 1.3: A 3D example, $n = 3$; the vectors $\mathbf{\Delta} = \Delta\mathbf{x}_n + \sum_{m=1}^{n-1} c_m \left(\Delta\mathbf{x}_m - \Delta\mathbf{x}_n\right)$ span the hyperplane described in Fig. 1.2. The same hyperplane is spanned by $\mathbf{\Delta} = \Delta\mathbf{x}_n + \sum_{m=1}^{n-1} a_m \left(\Delta\mathbf{x}_m - \Delta\mathbf{x}_{m+1}\right)$. This means that $\mathbf{\Delta}_{\min}$ is still the projection of $\mathbf{\Delta}$ along the normal to the hyperplane.

This means that the technique followed in method 4 can be copied. We only have to redefine the vectors $\Delta''\mathbf{x}_m$; $m = 1, ..., n-1$. Instead of Eq. 1.106, we now have

$$\Delta''\mathbf{x}_m = \Delta\mathbf{x}_m - \Delta\mathbf{x}_{m+1}. \tag{1.114}$$

We QR factorize $\mathbf{Q}''\mathbf{R}'' = \mathbf{X}''$ with $\mathbf{X}'' = (\Delta''\mathbf{x}_1\ \Delta''\mathbf{x}_2 ... \Delta''\mathbf{x}_{n-1})$, which has $\Delta''\mathbf{x}_m$; $m = 1, ..., n-1$ as its columns, $\mathbf{Q}''$ a $N \times (n-1)$ matrix with orthonormal columns, and $\mathbf{R}''$ a

$(n-1)\times(n-1)$ upper triangular matrix. Analogous to Eqs. 1.108 and 1.109, the equation to solve here is

$$\mathbf{R}''\mathbf{a} = -\mathbf{Q}''^{\dagger}\Delta\mathbf{x}_n, \tag{1.115}$$

with $\mathbf{a}$ the vector with coefficients $a_m;\ m = 1, ..., n-1$. The equation can be solved by back transformation, as $\mathbf{R}''$ is an upper triangular matrix. The original coefficients $c_m;\ m = 1, ..., n-1$ are then be obtained by using Eq. 1.113. Analogous to Eq. 1.110, the optimum vector can be expressed as

$$\mathbf{\Delta}_{\min} = \Delta\mathbf{x}_n - \mathbf{Q}''\mathbf{Q}''^{\dagger}\Delta\mathbf{x}_n = \Delta\mathbf{x}_n - \sum_{m=1}^{n-1}\mathbf{q}''_m\left(\mathbf{q}''^{\dagger}_m\Delta\mathbf{x}_n\right). \tag{1.116}$$

Cost and stability.* We come to the main reason why method 5 is computationally advantageous. As before, QR factorization of the $N\times(n-1)$ matrix $\mathbf{X}''$ is $O(n^2N)$, so this is the most costly step. However, unlike method 4, now we can bring that cost down. We apply the same trick as in method 3.

Suppose we have a QR factorization of the previous iteration $\mathbf{Q}''^{(n-1)}\mathbf{R}''^{(n-1)} = \mathbf{X}''^{(n-1)}$. $\mathbf{X}''$ can be constructed by adding one column to the $N\times(n-2)$ matrix $\mathbf{X}''^{(n-1)}$, i.e. $\mathbf{X}'' = \left(\mathbf{X}''^{(n-1)}\ \Delta''\mathbf{x}_n\right)$. Similarly, the $N\times(n-1)$ matrix $\mathbf{Q}''$ can be constructed from the $N\times(n-2)$ matrix $\mathbf{Q}''^{(n-1)}$ by adding one column. This column can be obtained by Gram-Schmidt orthogonalizing the new vector $\Delta''\mathbf{x}_n$ on the columns $\mathbf{q}''_k;\ k = 1, ..., n-1$, of $\mathbf{Q}''^{(n-1)}$

$$\begin{aligned}
&\mathbf{q}''_n = \Delta''\mathbf{x}_n;\\
&\text{do } k = 1, ..., n-2 \quad R''_{kn} = \mathbf{q}''^{\dagger}_k\mathbf{q}''_n;\ \mathbf{q}''_n = \mathbf{q}''_n - \mathbf{q}''_k R''_{kn};\\
&R''_{nn} = \|\mathbf{q}''_n\|;\ \mathbf{q}''_n = \mathbf{q}''_n/R''_{nn};
\end{aligned} \tag{1.117}$$

The vector $\mathbf{q}''_n$ is then appended to $\mathbf{Q}''^{(n-1)}$, i.e. $\mathbf{Q}'' = \left(\mathbf{Q}''^{(n-1)}\ \mathbf{q}''_n\right)$. At the same time, the $(n-2)\times(n-2)$ matrix $\mathbf{R}''^{(n-1)}$ is extended to the $(n-1)\times(n-1)$ matrix $\mathbf{R}''$ by adding one column $R''_{kn};\ k = 1, ..., n-1$, calculated in Eq. 1.117, and one row $R''_{nm} = \delta_{nm}R''_{nn};\ m = 1, ..., n-1$. Eq. 1.117 is a $O(nN)$ calculation (essentially the calculation of n inner products $\mathbf{q}''^{\dagger}_k\mathbf{q}''_n$), which make the costs similar to that of method 3.

Method 5 thus combines the low cost of method 3 (a $O(nN)$ calculation) and the accuracy/stability of method 4 (low condition number $\kappa(\mathbf{R}'')$). Perfect!

1.4.3 Orthogonalized Pulay iteration.*

In this section I will integrate the QR factorization of Eq. 1.117 in the Pulay algorithm of Eqs. 1.89-1.93. The present algorithm is called *orthogonalized Pulay iteration*. This is a state-of-the-art method.

$$\mathbf{x}_0 = \mathbf{v};\ \mathbf{r}_0 = \mathbf{A}\mathbf{x}_0 - \mathbf{b}(\mathbf{x}_0);\ \text{solve } \mathbf{A}\Delta\mathbf{x}_0 = -\mathbf{r}_0;\ \mathbf{x}_1 = \mathbf{x}_0 + \Delta\mathbf{x}_0;\ \mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - \mathbf{b}(\mathbf{x}_1); \tag{1.118}$$

$$\mathbf{Q}_0 = \text{nil};\ \mathbf{Y}_1 = \mathbf{x}_0 - \mathbf{x}_1;$$

$$\text{do } n = 1, \ldots\ ;\ \text{solve } \mathbf{A}\Delta\mathbf{x}_n = -\mathbf{r}_n;\ \mathbf{q}_n = \Delta\mathbf{x}_{n-1} - \Delta\mathbf{x}_n; \tag{1.119}$$

$$\{\text{do } k = 1, \ldots, n-2\ \ R_{kn} = \mathbf{q}_k^\dagger \mathbf{q}_n;\ \mathbf{q}_n = \mathbf{q}_n - \mathbf{q}_k R_{kn}\};\ R_{nn} = \|\mathbf{q}_n\|\ ;\ \mathbf{q}_n = \mathbf{q}_n / R_{nn}; \tag{1.120}$$

$$\mathbf{Q}_n = (\mathbf{Q}_{n-1}\ \mathbf{q}_n)\ ;\ \mathbf{s}_n = -\mathbf{Q}_n^\dagger \Delta\mathbf{x}_n;\ \text{solve } \mathbf{R}\mathbf{a}_n = \mathbf{s}_n; \tag{1.121}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{Y}_n\mathbf{a}_n + \Delta\mathbf{x}_n + \mathbf{s}_n;\ \mathbf{Y}_{n+1} = (\mathbf{Y}_n\ \mathbf{x}_n - \mathbf{x}_{n+1})\ ; \tag{1.122}$$

$$\mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \mathbf{b}(\mathbf{x}_{n+1});\quad \text{until converged.} \tag{1.123}$$

The algorithm looks complicated, but it is quite elegant. After initialization in Eq. 1.118, the next correction vector $\Delta\mathbf{x}_n$ is produced in Eq. 1.119, by a rule of the type of Eq. 1.80, for instance. The new difference vector, Eq. 1.114, is then orthogonalized to all previous vectors in Eq. 1.120, and the matrix $\mathbf{R}$ is extended. The matrix $\mathbf{Q}$ is extended in Eq. 1.121, which completes the QR factorization for this cycle. The correction vector is then projected on the hyperplane, and the coefficients $\mathbf{a}_n$ are found by back substitution. Eq. 1.122 then describes the subsequent Pulay move. The next guess for the solution vector $\mathbf{x}_{n+1}$ is constructed, following the Pulay prescription

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \sum_{m=1}^{n-1} a_m \left(\mathbf{x}_m - \mathbf{x}_{m+1}\right) + \mathbf{\Delta}_{\min}. \tag{1.124}$$

Finally, in Eq. 1.123 the next residual vector $\mathbf{r}_{n+1}$ is calculated, and convergence is tested. As before, the convergence criterion is something like $\|\mathbf{r}_n\| < \epsilon$, or $\max_{i=1,\ldots,N} |(r_n)_i| < \epsilon$, with ϵ a tolerance set by the user.

Cost and stability. The cost of orthogonalized Pulay is similar to that of the original Pulay algorithm, Eqs. 1.89-1.93. i.e., a $O(nN^2)$ calculation for a dense matrix $\mathbf{A}$, and a $O(n^2N)$ calculation for a sparse matrix $\mathbf{A}$. The memory overhead is also similar, i.e., $2nN$, as the matrices $\mathbf{Y}_n$ and $\mathbf{Q}_n$ need to be stored. Usually this is not a problem.

The stability of Pulay iteration is greatly improved by orthogonalization.

1.4.4 The art of subspace building - limited-memory Pulay*

Building subspaces is an art-form in itself, and there are numerous modifications in the literature for doing this in a clever way. For instance, it is typically not a good idea to start subspace iteration right from the start. Unless one has some pre-knowledge, the initial guess $\mathbf{x}_1$ will be a lousy approximation to the solution. The first couple of correction vectors $\Delta\mathbf{x}_n$ will likely not bring you close to the solution either, so it makes little sense to keep a memory of those initial steps. It is a bad memory. One strategy is to start iterating with a one-step memory technique, until you have a reasonable vector $\mathbf{x}_\ell$. Then switch on subspace iteration. If convergence is slow, subspace iteration can be restarted, which means that you throw away a number (or all) of the "oldest" vectors in the space $\mathbf{X}_n$, Eq. 1.90.

Many people swear that you should always do that, and keep the dimension of $\mathbf{X}_n$ fixed at n', after an initial build-up. That means, whenever you add a new vector, you also throw out the oldest one.[29] Alternatively, you can, for instance, throw out the correction vector $\Delta\mathbf{x}_m$ with the largest norm, as that should represent the largest error. Typically the fixed dimension is kept small, $n' \lesssim 10$, or even $n' \lesssim 5$. This is called Pulay iteration with *limited memory,*

The standard Pulay algorithm, Eqs. 1.89-1.93, is trivially adapted to subspace rebuilding, including limited memory. Here is a version

$$\mathbf{x}_1 = \mathbf{v};\ \mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - \mathbf{b}(\mathbf{x}_1);\ \mathbf{X}_0 = \text{nil};\ \mathbf{Y}_1 = \mathbf{x}_1; \tag{1.125}$$
$$\text{do } n = 1, \ldots\ ;\ \text{solve } \mathbf{A}\Delta\mathbf{x}_n = -\mathbf{r}_n;\ \mathbf{X}_n = (\mathbf{X}_{n-1}\ \Delta\mathbf{x}_n)\,; \tag{1.126}$$
$$\text{if } n > n' \text{ then remove 1st column of } \mathbf{X}_n \text{ and of } \mathbf{Y}_n; \tag{1.127}$$
$$\mathbf{K}_n = \mathbf{X}_n^\dagger \mathbf{X}_n;\ \mathbf{c}_n = \text{Pulay minimize } \mathbf{K}_n; \tag{1.128}$$
$$\mathbf{x}_{n+1} = (\mathbf{X}_n + \mathbf{Y}_n)\,\mathbf{c}_n;\ \mathbf{Y}_{n+1} = (\mathbf{Y}_n\ \mathbf{x}_{n+1})\,; \tag{1.129}$$
$$\mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \mathbf{b}(\mathbf{x}_{n+1});\quad \text{until converged.} \tag{1.130}$$

Eq. 1.127 is the only line we have to add in order to keep $\mathbf{X}_n$ and $\mathbf{Y}_n$ $N \times n'$ matrices. Note that in this version we throw away the oldest vector when we add a new one. Of course, one has to do a little bit of administration to keep building up the matrix $\mathbf{K}_n$ efficiently, i.e., throw away the row/column corresponding to the oldest vector, and add the row/column corresponding to the new vector, see method 1 above.

To adapt orthogonalized Pulay, Eqs. 1.118-1.123, is more difficult. In particular, it is somewhat tedious to keep the QR factorization efficient, such that it can use the information gathered in previous iterations. One could argue that there is less need to be efficient, if one keeps n' small, and you only need to orthogonalize n' vectors. Here is a version of orthogonalized Pulay that redoes the QR factorization each iteration

$$\mathbf{x}_0 = \mathbf{v};\ \mathbf{r}_0 = \mathbf{A}\mathbf{x}_0 - \mathbf{b}(\mathbf{x}_0);\ \text{solve } \mathbf{A}\Delta\mathbf{x}_0 = -\mathbf{r}_0;\ \mathbf{x}_1 = \mathbf{x}_0 + \Delta\mathbf{x}_0;\ \mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - \mathbf{b}(\mathbf{x}_1); \tag{1.131}$$
$$\mathbf{X}_0 = \text{nil};\ \mathbf{Y}_1 = \mathbf{x}_0 - \mathbf{x}_1;$$
$$\text{do } n = 1, \ldots\ ;\ \text{solve } \mathbf{A}\Delta\mathbf{x}_n = -\mathbf{r}_n;\ \mathbf{X}_n = (\mathbf{X}_{n-1}\ \ \Delta\mathbf{x}_{n-1} - \Delta\mathbf{x}_n)\,; \tag{1.132}$$
$$\text{if } n > n' \text{ then remove 1st column of } \mathbf{X}_n \text{ and of } \mathbf{Y}_n; \tag{1.133}$$
$$\mathbf{Q}\mathbf{R} = \mathbf{X}_n;\ \mathbf{s}_n = -\mathbf{Q}^\dagger\Delta\mathbf{x}_n;\ \text{solve } \mathbf{R}\mathbf{a}_n = \mathbf{s}_n; \tag{1.134}$$
$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{Y}_n\mathbf{a}_n + \Delta\mathbf{x}_n + \mathbf{s}_n;\ \mathbf{Y}_{n+1} = (\mathbf{Y}_n\ \mathbf{x}_n - \mathbf{x}_{n+1})\,; \tag{1.135}$$
$$\mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \mathbf{b}(\mathbf{x}_{n+1});\quad \text{until converged.} \tag{1.136}$$

As in the standard Pulay algorithm, we keep $\mathbf{X}_n$ and $\mathbf{Y}_n$ $N \times n'$ matrices by throwing away the oldest vector, whenever we add a new one. QR factorization in Eq. 1.134 is a $O(n'^2 N)$ calculation, so this is manageable only if we keep n' small.

Details: I did my homework, in case you are wondering. It is possible (if somewhat tedious) to avoid redoing the QR factorization from scratch, each time a single vector (column of $\mathbf{X}_n$ or $\mathbf{Y}_n$) is

[29] Like in human resources management.

removed. The QR factorization $\mathbf{QR} = \mathbf{X}_n$ reads

$$\begin{pmatrix} q_{11} & q_{12} & q_{13} & \cdots & q_{1n'} \\ q_{21} & q_{22} & q_{23} & \cdots & q_{2n'} \\ q_{31} & q_{32} & q_{33} & \cdots & q_{3n'} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ q_{N1} & q_{N2} & q_{N3} & \cdots & q_{Nn'} \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1n'} \\ 0 & r_{22} & r_{23} & \cdots & r_{2n'} \\ 0 & 0 & r_{33} & \cdots & r_{2n'} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & r_{n'n'} \end{pmatrix} = \begin{pmatrix} X_{11} & X_{12} & X_{13} & \cdots & X_{1n'} \\ X_{21} & X_{22} & X_{23} & \cdots & X_{2n'} \\ X_{31} & X_{32} & X_{33} & \cdots & X_{3n'} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X_{N1} & X_{N2} & X_{N3} & \cdots & X_{Nn'} \end{pmatrix}. \tag{1.137}$$

Removing the first column of $\mathbf{X}_n$, as requested by Eq. 1.133, gives

$$\begin{pmatrix} q_{21} & q_{22} & q_{23} & \cdots & q_{2n'} \\ q_{31} & q_{32} & q_{33} & \cdots & q_{3n'} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ q_{N1} & q_{N2} & q_{N3} & \cdots & q_{Nn'} \end{pmatrix} \begin{pmatrix} r_{12} & r_{13} & \cdots & r_{1n'} \\ r_{22} & r_{23} & \cdots & r_{2n'} \\ 0 & r_{33} & \cdots & r_{2n'} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{n'n'} \end{pmatrix} = \begin{pmatrix} X_{21} & X_{22} & X_{23} & \cdots & X_{2n'} \\ X_{31} & X_{32} & X_{33} & \cdots & X_{3n'} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X_{N1} & X_{N2} & X_{N3} & \cdots & X_{Nn'} \end{pmatrix}. \tag{1.138}$$

The matrix $\mathbf{R}$ is now a $(n'-1)\times(n'-1)$ matrix, instead of the original $n'\times n'$ matrix. However, it is not upper triangular anymore, as its subdiagonal contains the nonzero elements r_{ii}; $i = 2, ..., n'-1$. In mathematics this form is called the *upper Hessenberg* form.

There are several ways to get rid of the subdiagonal elements. One of these is by Householder transformations, see Sec. 1.2.1 in the lecture notes on eigenvalue problems. One can write

$$\mathbf{R} = \mathbf{P}^{(1)}\mathbf{P}^{(2)}...\mathbf{P}^{(n'-2)}\mathbf{R}', \tag{1.139}$$

where $\mathbf{R}$ is the matrix in the middle of Eq. 1.138, and $\mathbf{R}'$ is a $(n'-1)\times(n'-1)$ upper triangular matrix

$$\mathbf{R}' = \begin{pmatrix} r'_{11} & r'_{12} & \cdots & r'_{1n'} \\ 0 & r'_{22} & \cdots & r'_{2n'} \\ 0 & 0 & \cdots & r'_{2n'} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r'_{n'n'} \end{pmatrix}. \tag{1.140}$$

The Householder matrices $\mathbf{P}^{(k)}$ have the form

$$\mathbf{P}^{(k)} = \mathbf{I} - \frac{1}{w^{(k)}}\mathbf{v}^{(k)}\mathbf{v}^{(k)\dagger}, \tag{1.141}$$

where $\mathbf{I}$ is the $(n'-1)\times(n'-1)$ identity matrix, and $\mathbf{v}^{(k)}$ is a $(n'-1)\times 1$ column vector that has two nonzero elements

$$\mathbf{v}^{(k)} = \begin{pmatrix} 0 \\ \vdots \\ v_k^{(k)} \\ v_{k+1}^{(k)} \\ \vdots \\ 0 \end{pmatrix}. \tag{1.142}$$

How to construct these vectors $\mathbf{v}^{(k)}$ and the numbers $w^{(k)}$ can be found in Sec. 1.2.1 of the lecture notes on eigenvalue problems.

The matrix product of Eq. 1.138 can now be written as

$$\mathbf{QR} = \mathbf{QP}^{(1)}\mathbf{P}^{(2)}...\mathbf{P}^{(n'-2)}\mathbf{R}' = \mathbf{Q}'\mathbf{R}' \quad \text{with} \quad \mathbf{Q}' = \mathbf{QP}^{(1)}\mathbf{P}^{(2)}...\mathbf{P}^{(n'-2)}. \tag{1.143}$$

The matrix $\mathbf{Q}'$ is unitary, because $\mathbf{Q}$ is unitary, and the Householder matrices $\mathbf{P}^{(k)}$ are unitary, Eq. 1.141, and the product of unitary matrices is unitary. Moreover, the calculation of the products in Eq. 1.143 is very efficient. Look at

$$\mathbf{QP}^{(1)} = \mathbf{Q} - \frac{1}{w^{(1)}} \left[\mathbf{Qv}^{(1)}\right] \mathbf{v}^{(1)\dagger}. \tag{1.144}$$

As $\mathbf{v}^{(1)}$ has only two nonzero elements ($v_1^{(1)}$ and $v_2^{(1)}$), the matrix-vector product $\mathbf{Qv}^{(1)} = \mathbf{z}$ involves $2N$ multiplications, resulting in the $N \times 1$ column vector $\mathbf{z}$. Calculating the $N \times 2$ matrix $\left[\mathbf{Qv}^{(1)}\right] \mathbf{v}^{(1)\dagger} = \mathbf{zv}^{(1)\dagger} = \mathbf{Z}$ again involves $2N$ multiplications. Finally, constructing $\mathbf{QP}^{(1)} = \mathbf{Q} - \frac{1}{w^{(1)}}\mathbf{Z}$ involves $2N$ multiplications/subtractions.[30] Summarizing, constructing the matrix $\mathbf{QP}^{(1)}$ in Eq. 1.144 is a $O(N)$ calculation. As we have to do this $n' - 2$ times, Eq. 1.143, calculating $\mathbf{Q}'$ is a $O(n'N)$ calculation.

This takes care of the removal of the first column of $\mathbf{X}_n$. One also adds a column, see Eq. 1.132. One can take care of this by another Householder transformation. Alternatively, one can use MGS orthogonalization, Eq. 1.120, which is also a $O(n'N)$ calculation. Note that this has to be done *after* the calculation of Eq. 1.143. Using the steps just outlined makes QR factorization a $O(n'N)$ calculation, instead of the $O(n'^2N)$ calculation if one does it from scratch. If N is very large, and $n' \sim 10$, this can pay off. But remember the basic rule of life,[31] it is somewhat complicated.

1.4.5 Generalized minimized residual algorithm (GMRES)*

Although Pulay designed his iterative method to solve *non-linear* problems, it can, in principle, also be applied to iteratively solve *linear* problems. One replaces $\mathbf{b}(\mathbf{x})$ in Eqs. 1.89 and 1.91, or in Eqs. 1.118 and 1.123 by a constant vector $\mathbf{b}$. Of course, Eqs. 1.90 and 1.119 then only make sense, if a rule of the type of Eq. 1.80 is used. For linear problems, Pulay iteration is logically equivalent to the generalized minimized residual algorithm (GMRES), which is a popular algorithm for solving a linear system by subspace iteration [10, 12]. You can find the standard formulation of GMRES in the literature [3]. Here, I only show you the connection between the two algorithms, Pulay and GMRES.

For linear problems Eq. 1.80 is rewritten as

$$\Delta\mathbf{x}_n = -\mathbf{r}'_n \text{ with } \mathbf{r}'_n = \mathbf{A}'\mathbf{x}_n - \mathbf{b}'; \quad \mathbf{A}' = \mathbf{M}^{-1}\mathbf{A}; \; \mathbf{b}' = \mathbf{M}^{-1}\mathbf{b}. \tag{1.145}$$

As explained in Sec. 1.1.3, the matrix $\mathbf{M}^{-1}$ is called the *preconditioner*. If we choose it wisely, the matrix $\mathbf{A}'$ has a smaller condition number than the matrix $\mathbf{A}$. For iterative algorithms convergence goes faster for matrices with smaller condition numbers.[32] In the following I drop the primes in the notation, but I am assuming that we always use the preconditioned form of Eq. 1.145.

With this in mind, one observes that with each iteration n in Eq. 1.119, the subspace is increased by adding $\mathbf{r}_n$, and the subspace dimension is increased by 1. One can write

$$\mathbf{r}_n = \mathbf{Ax}_n - \mathbf{b} = \mathbf{r}_{n-1} - \mathbf{Ar}_{n-1}. \tag{1.146}$$

[30] For the real diehards, who want to squeeze the last ms out of the calculation, one can avoid repeated divisions by $w^{(1)}$, which are relatively expensive. Instead, one calculates $w' = 1/w^{(1)}$ once, and multiplies with w'.

[31] "Elluk foordeil hep sun nadeil" (J. Cruyff).

[32] Ask a mathematician (or a hairdresser) for a proof of this statement.

Proof: write $\mathbf{r}_{n-1} = \mathbf{A}\mathbf{x}_{n-1} - \mathbf{b}$, subtract this from $\mathbf{r}_n$ to get $\mathbf{r}_n - \mathbf{r}_{n-1} = \mathbf{A}(\mathbf{x}_n - \mathbf{x}_{n-1}) = -\mathbf{A}\mathbf{r}_{n-1}$.

Applying this rule recursively proves that one can write $\mathbf{r}_n$ as a linear combination of the vectors

$$\mathbf{r}_1, \mathbf{A}\mathbf{r}_1, \mathbf{A}^2\mathbf{r}_1, ..., \mathbf{A}^{n-1}\mathbf{r}_1. \tag{1.147}$$

In numerical cultural tradition the space spanned such a sequence of vectors is called a *Krylov subspace*, and the individual vectors are called the *Krylov vectors*. Provided the Krylov vectors are linearly independent, the Krylov subspace has dimension n. Provided the residual vectors $\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_n$ are linearly independent, they span the same space, i.e., the Krylov subspace

$$\text{span}\left\{\mathbf{r}_m : \ m = 1, ..., n\right\} = \text{span}\left\{\mathbf{A}^{m-1}\mathbf{r}_1 : \ m = 1, ..., n\right\} \tag{1.148}$$

The Pulay update of Eq. 1.87 can be written as

$$\mathbf{x}_{n+1} = \sum_{m=1}^{n} c_m (\mathbf{x}_m - \mathbf{r}_m) = \mathbf{x}_1 + \sum_{m=1}^{n-1} d_m \mathbf{A}^{m-1}\mathbf{r}_1. \tag{1.149}$$

Proof: The first expression follows from Eqs. 1.87 and 1.145. Now write $\mathbf{x}_m = \mathbf{x}_1 + (\mathbf{x}_m - \mathbf{x}_1)$ $= \mathbf{x}_1 + (\mathbf{x}_m - \mathbf{x}_{m-1}) + (\mathbf{x}_{m-1} - \mathbf{x}_{m-2}) + ... + (\mathbf{x}_2 - \mathbf{x}_1) = \mathbf{x}_1 + \mathbf{r}_{m-1} + \mathbf{r}_{m-2} + ... + \mathbf{r}_1$, using Eq. 1.145. Using this in Eq. 1.87 gives $\tilde{\mathbf{x}}_{n+1} = \sum_{m=1}^{n} c_m \mathbf{x}_1 + \sum_{m=1}^{n} c'_m \mathbf{r}_m = \mathbf{x}_1 + \sum_{m=1}^{n} c'_m \mathbf{r}_m$, using Eq. 1.84. Then use Eq. 1.148.

The Pulay constrained minimization step, Eqs. 1.87 and 1.84, can be written symbolically as

$$\min_{c_m; \ m=1,...,n} \left\{ \left\| \sum_{m=1}^{n} c_m r_m \right\|^2 \middle| \sum_{m=1}^{n} c_m = 1 \right\}, \tag{1.150}$$

and is equivalent to the unconstrained minimization

$$\min_{c_m; \ m=1,...,n-1} \left\{ \|\mathbf{A}(\mathbf{x}_1 + \mathbf{v}_n) - \mathbf{b}\|^2 \right\}, \ \text{with } \mathbf{v}_n = \sum_{m=1}^{n-1} c_m \mathbf{A}^{m-1}\mathbf{r}_1. \tag{1.151}$$

Proof: start from $\mathbf{A}(\mathbf{x}_1 + \mathbf{v}_n) - \mathbf{b} = \mathbf{A}\mathbf{x}_1 - \mathbf{b} + \sum_{m=1}^{n-1} c_m \mathbf{A}^m \mathbf{r}_1 = \mathbf{r}_1 + \sum_{m=1}^{n-1} c_m (\mathbf{r}_m - \mathbf{r}_{m+1})$, using Eq. 1.146. Writing $\mathbf{r}_m - \mathbf{r}_{m+1} = (\mathbf{r}_m - \mathbf{r}_1) - (\mathbf{r}_{m+1} - \mathbf{r}_1)$, one can convert this expression to the form $\mathbf{r}_1 + \sum_{m=1}^{n-1} c'_m (\mathbf{r}_{m+1} - \mathbf{r}_1)$. This is equivalent to $\sum_{m=1}^{n} c''_m \mathbf{r}_m$ with $\sum_{m=1}^{n} c''_m = 1$, by defining $c''_1 = 1 - \sum_{m=1}^{n-1} c'_m$, and $c''_{m+1} = c'_m$; $m = 1, ..., n-1$.

In other words, the Pulay constrained minimization step of Eq. 1.150 is equivalent to the unconstrained minimization of Eq. 1.151. The Pulay algorithm then logically does the following thing

$$\mathbf{x}_0 = \mathbf{c}; \ \mathbf{K}_0 = \text{nil}; \ \mathbf{a}_0 = \mathbf{x}_0; \tag{1.152}$$

$$\text{do } n = 1, ... \ ; \ \mathbf{a}_n = \mathbf{A}\mathbf{a}_{n-1}; \ \mathbf{K}_n = (\mathbf{K}_{n-1} \ \mathbf{a}_n); \tag{1.153}$$

$$[\mathbf{v}_n, \ r_n] = \min \left\{ \|\mathbf{A}(\mathbf{x}_1 + \mathbf{v}_n) - \mathbf{b}\|^2 \, | \mathbf{v}_n \in \mathbf{K}_n \right\}; \ \text{ until converged.} \tag{1.154}$$

The best current guess of the optimal vector is $\mathbf{x}_1 + \mathbf{v}_n$, with $\mathbf{v}_n$ the vector returned by the minimization, and $r_n = \|\mathbf{A}(\mathbf{x}_1 + \mathbf{v}_n) - \mathbf{b}\|$ the minimal residue found so far. After some initialization in Eq. 1.152, one vector is added to the Krylov sequence, Eq. 1.147, in Eq. 1.153, thereby increasing the dimension of the Krylov subspace, Eq. 1.148, by one. The vector of the type given by Eq. 1.149 with minimum residue is then found in Eq. 1.154.

The algorithm just described is called *generalized minimized residual* algorithm (GMRES). The "minimized residual" part of this should be clear. The algorithm is called "generalized", because it is suitable for any general (non-singular) matrix. It is a popular algorithm for solving a linear system using subspace iteration. Pulay iteration applied to a linear system, either in its original form, Eqs. 1.89-1.91, or in its orthogonalized form, Eqs. 1.118-1.123, are particular realizations of the minimized residual algorithm.

Chapter 2

EIGENVALUE PROBLEMS

Prologue. Matrix diagonalization, finding the eigenvalues and eigenvectors of a matrix, is an important topic for a physicist. Judging from its numerous applications in science and engineering, it is one of the most important topics, and one on which large amounts of computer time are spent. The topic is part of numerical linear algebra, which is one of the most well-developed fields of numerical mathematics. The statements made in the lecture notes on linear and non-linear problems remain valid here.

Generations of scientists have worked on the construction of numerical libraries with efficient, robust, and fast numerical routines for matrix diagonalization.[1] For standard problems it is unlikely that you can come up with something that beats these routines, so the best option usually to look for an appropriate routine in such a library. Nevertheless, using such routines merely as black boxes, is usually not a good idea. Think inside of the box !![2] Moreover, physics problems quite often deal with matrices that have a special structure, dictated by the underlying physics. Very often, it pays off to think about how to use such a special structure in order to limit the computational time spend on diagonalization, but you can only do that if you know how diagonalization works.

In these notes I will explain some of the current techniques used to solve the linear eigenvalue problem $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, where $\mathbf{A}$ is an $N \times N$ matrix, λ is an eigenvalue, and $\mathbf{x}$ is the corresponding $N \times 1$ eigenvector. I focus on Hermitian matrices, $\mathbf{A}^\dagger = \mathbf{A}$, as these are the most important ones for physicists. I will do my best to discuss some of the more important techniques without using too many ϵ's and δ's. Very thorough (and very mathematical) exposés can be found in Refs. [3, 13]. Introductory material can be found in numerics textbooks, e.g., Refs. [4, 5, 6, 7].

One also encounters non-linear problems of the type $\mathbf{A}(\mathbf{x})\mathbf{x} = \lambda\mathbf{x}$, where the matrix elements $A_{ij}(x_1, x_2, ..., x_N)$ are (known) functions of the eigenvector, or even more complicated non-linear problems, where they are functions of more than one eigenvector. The latter occur, for instance, in quantum mechanical electronic structure problems, a number of which are discussed in Chapter 5. Such problems are typically solved using a combination of diagonalization techniques, discussed below, and techniques for solving non-linear problems, discussed in the corresponding lecture notes. We will encounter a couple of such non-linear eigenvalue problems in Part II. Another type of non-linear problem is $\mathbf{A}\mathbf{x} = f(\lambda)\mathbf{x}$, where $f(\lambda)$ is a known function of the eigenvalue. These I will not discuss.

[1] These are incorporated in numerical libraries such as LAPACK and ARPACK, see the lecture notes on (non-)linear problems [1, 2].

[2] The presents are inside the box, outside of the box is just hot air.

2.1 Introduction

There are many routines available that solve eigenvalue problems $\mathbf{A}\mathbf{x}_n = \lambda_n \mathbf{x}_n$, where $\mathbf{A}$ is an $N \times N$ matrix, λ_n is an eigenvalue, and $\mathbf{x}_n$ is the corresponding $N \times 1$ eigenvector. (As a matrix typically has more than one eigenvalue/eigenvector I use a subscript to distinguish between them.) Which routine you use depends upon the properties of your matrix $\mathbf{A}$.

- Is your matrix $\mathbf{A}$ Hermitian, i.e., $\mathbf{A}^\dagger = \mathbf{A}$? Finding eigenvalues λ_n of non-Hermitian matrices typically requires more work. Therefore, in the following I will cover Hermitian matrices only. These are dominant in physical applications anyway. Hermitian matrices always lead to a complete and orthogonal set of eigenvectors, $\mathbf{x}_n;\ n = 1, ..., N$, with $\mathbf{x}_m^\dagger \mathbf{x}_n = 0$, if $m \neq n$.

- How large is your matrix $\mathbf{A}$? For a small matrix you can afford (and you should afford) using an routine that finds all the eigenvalues $\lambda_n;\ n = 1, ..., N$ simultaneously (and the eigenvectors, if you need them). Whether you actually need all the eigenvalues is not terribly important, as the calculation for small matrices is relatively fast. For a large matrix a calculation of all eigenvalues would take too long, and for applications to physical problems you would not use all of them anyway.[3]

- If you don't want all N eigenvalues, which ones do you need? There are routines that find the smallest eigenvalue, or the largest one, or even one in the middle of the spectrum. You might be interested in the lowest one $\lambda_1 < \lambda_n;\ n = 2, ..., N$. For instance, in quantum mechanics the lowest eigenvalue λ_1 and eigenvector $\mathbf{x}_1$ of the Hamiltonian represent the ground state of the system. Quite frequently you might like to have the lowest M eigenvalues, $\lambda_m < \lambda_n;\ m = 1, ..., M;\ n = M+1, ..., N$. One uses these to construct an excitation spectrum, for comparison to experimental spectra, or for use in a statistical physics calculation, for instance. There are special techniques for finding M eigenvalues simultaneously.

- An important question is, does your matrix $\mathbf{A}$ have a special structure? For instance, operations on a tridiagonal matrix are much more efficient than operations on a full, dense matrix. If the latter is larger than very small, but still smaller than very large, it pays off to first transform it into a tridiagonal matrix, before starting the eigenvalue search.[4]

- Large scale linear algebra calculations on modern computers make heavy use of parallelization (multiple processors get involved in a single matrix operation.) Some routines can be parallelized more efficiently than others, so you should keep this criterion in the back of your mind.

All routines for finding eigenvalues are based on open-ended iterative techniques, where by "open-ended" I mean that you cannot tell on forehand how many iterations it will take to produce a meaningfully converged result. As this does not sound too encouraging, one needs to carefully monitor convergence, and interfere if convergence fails, or is too slow. Many library routines have build-in schemes that interfere automatically, but in any case the user is advised to keep paying attention.

[3] The distinction between small and large is a function of time and computer technology. Matrices that were considered large in the old days, became smaller while technology progressed.

[4] Other sparse matrices (matrices where most of the elements are zero, but are not tridiagonal) require other special techniques, but I won't talk about those [2].

Contents of this chapter. After a short recapitulation on Hermitian matrices in Sec. 2.2, I give descriptions of some of the techniques that are used to solve eigenvalue problems of Hermitian matrices. Most techniques, even the more sophisticated ones, are variations on a relatively simple method, called *power iteration*. This method, and its siblings, *inverse power iteration* and *Rayleigh quotient iteration* are explained in Sec. 2.3. Those techniques enable you to find one selected eigenvalue, and the corresponding eigenvector. In principle they are suitable for small and large matrices.

Using such routines, one can subsequently search for other eigenvalues one at a time, but this is usually thought to be too cumbersome. If you wish to have all eigenvalues of a matrix, then *QR iteration* is the standard technique. This is explained in Sec. 2.6. It requires some explanation of the general background, which is done in Sec. 2.4. In principle, QR iteration works for any matrix, but if you want it to be efficient, it is advised to apply it to a *tridiagonal matrix* only. There are general methods to convert any Hermitian matrix into a tridiagonal one. One of the more popular ones is explained in Sec. 2.5.

A typical problem encountered in physics is finding the lowest eigenvalue/eigenvector of a very large matrix. Where (inverse) power iteration or Rayleigh quotient iteration can do that, there are more sophisticated and faster methods available. These methods use *subspace techniques*, similar to the more sophisticated methods for solving (non-)linear problems, see the corresponding lecture notes. I will discuss two of these in Sec. 2.7. *Davidson iteration* uses subspaces to find a single eigenvalue. It is popular among physicists and chemists. *Pulay iteration* (or DIIS), which we have seen before in (non-)linear problems, is also a popular subspace technique for finding single eigenvalues/eigenvectors. I will compare the two.[5]

As discussed above, it would be both impossible and silly to require all eigenvalues of a very large matrix. Subspace techniques are also suited to find a limited number of eigenvalues simultaneously, which is explained in Sec. 2.8. *Subspace iteration* is the most straightforward technique for doing this. Davidson iteration can be extended to search for a number of eigenvalues simultaneously, where it is called *block Davidson*.

2.2 The eigenvalue problem

Consider the eigenvalue problem

$$\mathbf{A}\mathbf{x}_n = \lambda_n \mathbf{x}_n, \tag{2.1}$$

where $\mathbf{A}$ is a square $N \times N$ matrix, $\mathbf{x}_n$ is a column vector of length N, and λ_n is a number. The number λ_n is called an *eigenvalue* and $\mathbf{x}_n$ is the corresponding *eigenvector*. As a matrix has typically more than one of these, a subscript is used to distinguish the different eigenvalues and eigenvectors. By writing Eq. 2.1 as $(\mathbf{A} - \lambda_n \mathbf{I})\,\mathbf{x}_n = 0$, with $\mathbf{I}$ the $N \times N$ identity matrix, one can see that a non-trivial solution, $\mathbf{x}_n \neq \mathbf{0}$, exists only if

$$\det(\mathbf{A} - \lambda_n \mathbf{I}) = 0. \tag{2.2}$$

As $\det \mathbf{B}$ is a sum of products of N matrix elements B_{ij}, Eq. 2.2 is an equation of type $P_N(\lambda) = 0$, where $P_N(\lambda)$ is a polynomial in λ of degree N. The fundamental theorem of

[5] There are many more subspace techniques. Many mathematicians favor a technique called Lanczos diagonalization. Many users in the physics and chemistry communities are lukewarm about Lanczos. Who is right with his/her opinion probably depends on the particular application. Conjugate gradient is a general optimization technique, which can also be applied to solving eigenvalue problems. I will discuss it in the lecture notes on optimization.

algebra states that we can always factorize a polynomial $P_N(\lambda) = a\prod_{n=1}^{N}(\lambda - \lambda_n)$, with λ_n the roots of the polynomial, $P_N(\lambda_n) = 0$, provided we allow the roots to be complex numbers. So Eq. 2.2 always has N solutions λ_n. Of course, they do not have to be all different from one another.

Obtaining the eigenvalues λ_n by finding the roots of the polynomial $P_N(\lambda)$ is a possibility in principle. However, working out a determinant for large N is extremely cumbersome. It is also computationally inefficient, as the determinant is the sum of $N!$ terms, which becomes an impossibly large number for large N.[6] In other words, we need something more clever.

In the following, we restrict ourselves to *Hermitian* matrices $\mathbf{A}^\dagger = \mathbf{A}$. The main reason for this is that Hermitian matrices have special properties. Many of the matrices we encounter in physics are Hermitian.

2.2.1 Hermitian matrices

A complex, square matrix is called *Hermitian* if $\mathbf{A}^\dagger = \mathbf{A}$, or in matrix elements $\left(A^\dagger\right)_{ij} = A^*_{ji} = A_{ij}$. A real Hermitian matrix is called *symmetric,* or in matrix elements $\left(A^\dagger\right)_{ij} = A^*_{ji} = A_{ji} = A_{ij}$. In the following I deal with (complex) Hermitian matrices, but the results are equally valid for (real) symmetric matrices. The properties that are of interest to us are listed below.

(I) The *eigenvalues* of a Hermitian matrix are *real*.

Proof: define the inner product between two vectors $\mathbf{a}$ and $\mathbf{b}$ by $\mathbf{a}^\dagger\mathbf{b} = \sum_{i=1}^{N} a_i^* b_i$.[7] Setting $\mathbf{b} = \mathbf{a}$ defines the norm of a vector $\|\mathbf{a}\| = \sqrt{\mathbf{a}^\dagger\mathbf{a}} = \sqrt{\sum_{i=1}^{N}|a_i|^2}$. We have $\left(\mathbf{x}^\dagger\mathbf{A}\mathbf{x}\right)^\dagger = \mathbf{x}^\dagger\mathbf{A}^\dagger\mathbf{x} = \mathbf{x}^\dagger\mathbf{A}\mathbf{x}$; in other words $\mathbf{x}^\dagger\mathbf{A}\mathbf{x}$ is real. If $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, then $\mathbf{x}^\dagger\mathbf{A}\mathbf{x} = \lambda\mathbf{x}^\dagger\mathbf{x} = \lambda\|\mathbf{x}\|^2$ is real. As $\|\mathbf{x}\|$ is real, so must λ be real.

(II) The *eigenvectors* of a Hermitian matrix are *orthogonal*.

Proof: consider $\mathbf{A}\mathbf{x}_1 = \lambda_1\mathbf{x}_1$ and $\mathbf{A}\mathbf{x}_2 = \lambda_2\mathbf{x}_2$. Then (i) $\mathbf{x}_1^\dagger\mathbf{A}\mathbf{x}_2 = \lambda_2\mathbf{x}_1^\dagger\mathbf{x}_2$. Furthermore (ii) $\mathbf{x}_1^\dagger\mathbf{A}\mathbf{x}_2 = \left(\mathbf{x}_2^\dagger\mathbf{A}^\dagger\mathbf{x}_1\right)^\dagger = \left(\mathbf{x}_2^\dagger\mathbf{A}\mathbf{x}_1\right)^\dagger = \lambda_1^*\left(\mathbf{x}_2^\dagger\mathbf{x}_1\right)^\dagger = \lambda_1\mathbf{x}_1^\dagger\mathbf{x}_2$. Subtracting (i) rom (ii) gives $(\lambda_1 - \lambda_2)\mathbf{x}_1^\dagger\mathbf{x}_2 = 0$. If $\lambda_1 \neq \lambda_2$, then $\mathbf{x}_1^\dagger\mathbf{x}_2 = 0$, meaning that $\mathbf{x}_1$ and $\mathbf{x}_2$ are orthogonal. If $\lambda_1 = \lambda_2$ we can make them orthogonal. Define $\mathbf{x}_2' = \mathbf{x}_2 - \mathbf{x}_1(\mathbf{x}_1^\dagger\mathbf{x}_2)/\|\mathbf{x}_1\|^2$, then $\mathbf{x}_1^\dagger\mathbf{x}_2' = 0$. Furthermore $\mathbf{A}\mathbf{x}_2' = \lambda_2\mathbf{x}_2'$.

(III) The *eigenvectors* of a Hermitian matrix form a *basis set*, i.e., every vector can be written as a linear combination of these eigenvectors.

Proof: a counting proof. As $\det(\mathbf{A} - \lambda\mathbf{I}) = P(\lambda) = 0$ has at least one solution, $\lambda = \lambda_1$ (according to the fundamental theorem of algebra), there is at least one eigenvector $\mathbf{x}_1$. Define the space $V^\perp$

[6] Remember from your statistical physics courses that according to Stirling's formula $N! \propto N^N e^{-N}$ for large N.

[7] $\mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix}$ is a *column vector*, whereas $\mathbf{a}^\dagger = \begin{pmatrix} a_1^* & \cdots & a_N^* \end{pmatrix}$ is a *row vector*, so the matrix product $\mathbf{a}^\dagger\mathbf{b} = a_1^*b_1 + \cdots + a_N^*b_N$ is a *scalar*, which represents the (Hermitian) inner product.

spanned by vectors that are orthogonal to $\mathbf{x}_1$, $V^\perp = \left\{\mathbf{v} \mid \mathbf{x}_1^\dagger \mathbf{v} = 0\right\}$. This space has dimension $N-1$. Moreover $\mathbf{x}_1^\dagger \mathbf{A}\mathbf{v} = \left(\mathbf{v}^\dagger \mathbf{A}^\dagger \mathbf{x}_1\right)^\dagger = \left(\mathbf{v}^\dagger \mathbf{A}\mathbf{x}_1\right)^\dagger = \lambda_1^* \left(\mathbf{v}^\dagger \mathbf{x}_1\right)^\dagger = \lambda_1^* \mathbf{x}_1^\dagger \mathbf{v} = 0$. In other words $\mathbf{A}\mathbf{v} \in V^\perp$, so within this space it should be possible to represent $\mathbf{A}$ by a $(N-1)\times(N-1)$ Hermitian matrix. Then, recycle the arguments. By invoking the fundamental theorem of algebra, this matrix has at least one eigenvalue λ_2, and one eigenvector $\mathbf{x}_2$. Use this to construct a $N-2$ dimensional space spanned by vectors that are orthogonal to both $\mathbf{x}_2$ and $\mathbf{x}_1$. Within this space one can construct a $(N-2)\times(N-2)$ matrix representing $\mathbf{A}$, etcetera. After m cycles the dimension of the matrix is $N-m$, so when $m = N-1$, the argument stops.

2.3 Simple techniques: power iteration

Suppose we are not interested in all the eigenvalues, but just in one. In this section we will look at the basic technique to find one eigenvalue and its corresponding eigenvector. Unfortunately, it may not be the eigenvalue you are actually interested in. In the next section I will discuss how to have more control over obtaining the eigenvalue you want.

According to **(III),** any vector $\mathbf{b}$ can be written as a linear combination of the eigenvectors of $\mathbf{A}$, $\mathbf{b} = \sum_{i=1}^N c_i \mathbf{x}_i$. This means that

$$\mathbf{A}\mathbf{b} = \sum_{i=1}^N c_i \mathbf{A}\mathbf{x}_i = \sum_{i=1}^N c_i \lambda_i \mathbf{x}_i. \tag{2.3}$$

Now operate n times with $\mathbf{A}$, i.e., with $\mathbf{A}\mathbf{A}...\mathbf{A} = \mathbf{A}^n$, then

$$\mathbf{A}^n\mathbf{b} = \sum_{i=1}^N c_i \mathbf{A}^n \mathbf{x}_i = \sum_{i=1}^N c_i \lambda_i^n \mathbf{x}_i = \lambda_1^n \left\{ c_1\mathbf{x}_1 + \sum_{i=2}^N c_i \left(\frac{\lambda_i}{\lambda_1}\right)^n \mathbf{x}_i \right\}. \tag{2.4}$$

Suppose we have ordered the eigenvalues such that

$$|\lambda_1| > |\lambda_2| > ... > |\lambda_N| \tag{2.5}$$

If n becomes large, then the second term on the right-hand side of Eq. 2.4 goes to zero; because $|\lambda_i/\lambda_1| < 1$; $i = 2, ..., N$, one has $\lim_{n\to\infty} (\lambda_i/\lambda_1)^n = 0$. So, the first term on the right-hand side of Eq. 2.4 becomes dominant

$$n \text{ large } \mathbf{A}^n\mathbf{b} \to \lambda_1^n c_1 \mathbf{x}_1. \tag{2.6}$$

In other words, $\mathbf{A}^n\mathbf{b}$ converges to the eigenvector belonging to the eigenvalue with the largest absolute value. Moreover, the *Rayleigh quotient*

$$R_n = \frac{\mathbf{b}_n^\dagger \mathbf{A}\mathbf{b}_n}{\mathbf{b}_n^\dagger \mathbf{b}_n} \quad \text{with} \quad \mathbf{b}_n = \mathbf{A}\mathbf{b}_{n-1} \quad \text{and} \quad \mathbf{b}_0 = \mathbf{b}. \tag{2.7}$$

converges to the eigenvalue, because

$$R_n = \lambda_1 + O\left(\frac{\lambda_2^{2n+1}}{\lambda_1^{2n+1}}\right) \quad \text{so} \quad \lim_{n\to\infty} R_n = \lambda_1. \tag{2.8}$$

[Physicists with knowledge of quantum mechanics will call the Rayleigh quotient *the expectation value.*] We now have a simple algorithm to calculate the eigenvalue with the largest

absolute value: calculate the Rayleigh quotient R_n for $n = 1, 2, ...$ and monitor its convergence towards a fixed value λ_1. This algorithm is called *power iteration*.

As according to Eq. 2.4 the norm $\|\mathbf{A}^n\mathbf{b}\|$ grows in each cycle if $|\lambda_1| > 1$, it cannot hurt to normalize this vector in order to avoid overflow.[8] The *power iteration algorithm* then becomes

$$\begin{aligned} &\mathbf{b}_0 = \frac{\mathbf{b}}{\|\mathbf{b}\|};\ \mathbf{b}_1 = \mathbf{A}\mathbf{b}_0;\ R_0 = \mathbf{b}_0^\dagger \mathbf{b}_1; \\ &\text{do } n = 1, ...\ \ \mathbf{b}_n = \frac{\mathbf{b}_n}{\|\mathbf{b}_n\|};\ \mathbf{b}_{n+1} = \mathbf{A}\mathbf{b}_n;\ R_n = \mathbf{b}_n^\dagger \mathbf{b}_{n+1};\quad \text{until converged.} \end{aligned} \tag{2.9}$$

The criterion "until converged" will be something like $|R_n - R_{n-1}| < \epsilon$, with ϵ a tolerance chosen by the user. The Rayleigh quotient R_n is a measure for the eigenvalue λ_1, according to Eq. 2.8, and the vector $\mathbf{b}_n$ represents the eigenvector $\mathbf{x}_1$, see Eq. 2.6. Note that we have normalized this vector, i.e., $\|\mathbf{b}_n\| = 1$. In principle, the starting vector $\mathbf{b}$ can be anything you like, but of course the more "reasonable" it is, i.e., the closer it is to $\mathbf{x}_1$, the faster the convergence will be. In an implementation, only two vectors, $\mathbf{b}_0$ and $\mathbf{b}_1$, have to be kept in storage; they can be overwritten in the do-loop.

As we know that the second eigenvector, $\mathbf{x}_2$, is perpendicular to the first one, $\mathbf{x}_1$, one can search for the next eigenvalue λ_2 by starting with a vector $\mathbf{b}'$ that is perpendicular to $\mathbf{x}_1$, so $\mathbf{b}' = \sum_{i=2}^{N} c_i \mathbf{x}_i$. Assuming Eq. 2.5, and following the reasoning from Eqs. 2.3-2.9, one expects that power iteration based upon $\mathbf{b}'$ as a starting vector, gives λ_2 and $\mathbf{x}_2$. Somewhat disappointingly, this is not always what happens in an actual calculation. Because $\mathbf{A}\mathbf{b}'$ is calculated with finite precision on a computer, it will not be exactly perpendicular to to $\mathbf{x}_1$. Moreover, we have not obtained the exact $\mathbf{x}_1$, only an approximation $\tilde{\mathbf{x}}_1$ to it, and its quality depends on how strict our "until converged" criterion in Eq. 2.9 is. So in practice, $\mathbf{b}' = a_1\mathbf{x}_1 + \sum_{i=2}^{N} a_i \mathbf{x}_i$, with a_1 a small number. If you iterate according to Eq. 2.7 or 2.9, that number grows in each cycle, as it will be amplified by the factor λ_1^n. The $\mathbf{x}_1$ term can eventually dominate, and power iteration will just give you λ_1 and $\mathbf{x}_1$ again. To avoid this collapse onto $\mathbf{x}_1$, one can enforce orthogonality once and a while. In the following we will do it each cycle.

Power iteration for finding the second eigenvalue λ_2 and eigenvector $\mathbf{x}_2$ then becomes

$$\begin{aligned} &\mathbf{b}' = \mathbf{b}' - \mathbf{x}_1(\mathbf{x}_1^\dagger \mathbf{b}');\ \mathbf{b}_0 = \frac{\mathbf{b}'}{\|\mathbf{b}'\|};\ \mathbf{b}_1 = \mathbf{A}\mathbf{b}_0;\ \mathbf{b}_1 = \mathbf{b}_1 - \mathbf{x}_1(\mathbf{x}_1^\dagger \mathbf{b}_1);\ R_0 = \mathbf{b}_0^\dagger \mathbf{b}_1; \\ &\text{do } n = 1, ...\ \ \mathbf{b}_n = \frac{\mathbf{b}_n}{\|\mathbf{b}_n\|};\ \mathbf{b}_{n+1} = \mathbf{A}\mathbf{b}_n;\ \mathbf{b}_{n+1} = \mathbf{b}_{n+1} - \mathbf{x}_1(\mathbf{x}_1^\dagger \mathbf{b}_{n+1}); \\ &\qquad R_n = \mathbf{b}_n^\dagger \mathbf{b}_{n+1};\quad \text{until converged.} \end{aligned} \tag{2.10}$$

Note that we have assumed that $\|\mathbf{x}_1\| = 1$, which is the case if we have obtained this vector from the algorithm of Eq. 2.9. The Rayleigh quotient R_n is a measure for the eigenvalue λ_2 and the vector $\mathbf{b}_n$ represents the eigenvector $\mathbf{x}_2$, with $\|\mathbf{b}_n\| = 1$.

[8] An overflow occurs when a number becomes so large that the computer cannot represent it. The chance on overflow may be small, but why not avoid it, if the cost is small. If $|\lambda_1| < 1$, we run the risk of underflow, i.e., a number becoming too small for the computer to represent it. Normalization also solves this problem.

In principle, you can find all N eigenvalues and eigenvectors by applying this scheme iteratively. Suppose you have already found $\lambda_i, \mathbf{x}_i;\ i = 1, ..., M$. Then you can find the next pair by working with vectors that are perpendicular to all these $\mathbf{x}_i;\ i = 1, ..., M$. The orthogonalization steps in Eq. 2.10 then have to be replaced by

$$\text{do } i = 1, ..., M \quad \mathbf{b}_{n+1} = \mathbf{b}_{n+1} - \mathbf{x}_i(\mathbf{x}_i^\dagger \mathbf{b}_{n+1}), \tag{2.11}$$

for all n (including the initial step on $\mathbf{b}_1$). In practice, this algorithm becomes cumbersome if you want to calculate a large number of eigenvalues and eigenvectors. Better ways for that task are discussed below.

2.3.1 Inverse power iteration

Suppose one is not interested in the eigenvalue with the largest absolute value, $|\lambda_1|$, but in the eigenvalue with the smallest absolute value, $|\lambda_N|$. Of course, one can calculate all eigenvalues $\lambda_1, \lambda_2, ..., \lambda_N$ one at a time in the way discussed in the previous section, but that is a waste of time if you are only interested in the last one, λ_N. We can make use of the following trick. If λ_i is an eigenvalue of $\mathbf{A}$, then $1/\lambda_i$ is an eigenvalue of $\mathbf{A}^{-1}$ and vice versa.

Proof: start from $\mathbf{A}\mathbf{x}_i = \lambda_i \mathbf{x}_i$; multiplying both sides from the left with $\mathbf{A}^{-1}$ gives $\mathbf{x}_i = \lambda_i \mathbf{A}^{-1}\mathbf{x}_i$, which means $\lambda_i^{-1}\mathbf{x}_i = \mathbf{A}^{-1}\mathbf{x}_i$.

From Eq. 2.5 we have

$$|\lambda_N|^{-1} > |\lambda_{N-1}|^{-1} > ... > |\lambda_1|^{-1}. \tag{2.12}$$

In other words, to have immediate access to λ_N, we can perform power iteration with $\mathbf{A}^{-1}$ instead of $\mathbf{A}$. For obvious reasons this is called *inverse power iteration.*

In many physical problems we are neither terribly interested in the eigenvalue with the largest absolute value, $|\lambda_1|$, nor in the eigenvalue with the smallest absolute value, $|\lambda_N|$. Instead, we may be interested in the lowest eigenvalue, $\lambda_{\min} = \min(\lambda_1, \lambda_2, ..., \lambda_N)$. For instance, in quantum mechanics, the lowest eigenvalue of the Hamiltonian corresponds to the energy of the ground state. To search for that eigenvalue we apply the following trick. If λ_i is an eigenvalue of $\mathbf{A}$, then $\lambda_i - s$ is an eigenvalue of $\mathbf{A} - s\mathbf{I}$, where s is a number, and $\mathbf{I}$ is the $N \times N$ identity matrix.

Proof: is again trivial $(\mathbf{A} - s\mathbf{I})\,\mathbf{x}_i = \lambda_i \mathbf{x}_i - s\mathbf{x}_i = (\lambda_i - s)\,\mathbf{x}_i$.

If we choose $s < \lambda_{\min}$, then we have

$$|\lambda_{\min} - s|^{-1} > |\lambda_j - s|^{-1} \quad j \text{ any other eigenvalue.} \tag{2.13}$$

In other words, we have immediate access to $\lambda_{\min}$ by inverse power iteration with $(\mathbf{A} - s\mathbf{I})^{-1}$. For obvious reasons, the number s is called the *shift*, and the method is called *shifted inverse power iteration.*

Having to construct the matrix inverse $(\mathbf{A} - s\mathbf{I})^{-1}$ is a nuisance, because it is time-consuming. We can avoid it by using the following trick. What we need for shifted inverse power iteration is the operation $\mathbf{b}_n = (\mathbf{A} - s\mathbf{I})^{-1}\,\mathbf{b}_{n-1}$ in Eq. 2.9. Writing this as $(\mathbf{A} - s\mathbf{I})\,\mathbf{b}_n = \mathbf{b}_{n-1}$, it becomes a linear problem, with $\mathbf{b}_{n-1}$ known, and $\mathbf{b}_n$ unknown. We

can use one of the standard algorithms to solve this linear problem. For instance, we can LU decompose $\mathbf{A} - s\mathbf{I} = \mathbf{LU}$, to get $\mathbf{LU}\mathbf{b}_n = \mathbf{b}_{n-1}$, which we can solve by forward substitution, followed by back substitution, as discussed in the lecture notes on linear problems. The substitutions we have to do every cycle, but the LU decomposition has to be done only once, provided we store the matrices $\mathbf{L}$ and $\mathbf{U}$. The algorithm of Eq. 2.9 then becomes

$$\begin{aligned} &\mathbf{LU} = \mathbf{A} - s\mathbf{I};\ \mathbf{b}_0 = \frac{\mathbf{b}}{\|\mathbf{b}\|};\ \mathbf{LU}\mathbf{b}_1 = \mathbf{b}_0;\ S_0 = \mathbf{b}_0^\dagger \mathbf{b}_1; \\ &\text{do } n = 1, \ldots\ \mathbf{b}_n = \frac{\mathbf{b}_n}{\|\mathbf{b}_n\|};\ \mathbf{LU}\mathbf{b}_{n+1} = \mathbf{b}_n;\ S_n = \mathbf{b}_n^\dagger \mathbf{b}_{n+1};\quad \text{until converged.} \end{aligned} \tag{2.14}$$

Note that, instead of Eq. 2.8, we have for large n

$$S_n = \frac{1}{\lambda_{\min} - s} + \min_j O\left(|\lambda_{\min} - s|^{2n+1} / |\lambda_j - s|^{2n+1}\right). \tag{2.15}$$

[You can derive this result by substituting $\mathbf{A}$ by $(\mathbf{A} - s\mathbf{I})^{-1}$ in the analysis that lead to Eq. 2.8]. If we choose s close $\lambda_{\min}$, the second term on the right-hand side of Eq.2.15 becomes small. The best current guess for $\lambda_{\min}$ is then given by the first term, which means

$$\lambda_{\min} \approx \frac{1}{S_n} + s. \tag{2.16}$$

It is therefore wise to modify the "until converged" criterion to

$$\left| \frac{1}{S_n} - \frac{1}{S_{n-1}} \right| < \epsilon, \tag{2.17}$$

with ϵ a tolerance set by the user.

As with power iteration, one can find the second eigenvalue by inverse power iteration by repeated orthogonalization on the first eigenvector. You can modify the algorithm of Eq. 2.10 for that purpose.

2.3.1.1 Cost and efficiency

One can use some simple rules to estimate the cost of (inverse) power iteration. Elementary operations are multiplications or additions of floating point numbers, which are the computer's approximation to real or complex numbers. The computational cost is determined by the total number c of those operations. The cost of solving a matrix-vector problem is then expressed as

$$c = \sum_{r=0}^{p} a_r N^r, \tag{2.18}$$

with N the dimension of the matrix/vector. For sufficiently large N, the highest power N^p is the dominant term in c. If we are not terribly interested in the prefactor a_p, we say that the cost c scales as order N^p, or, alternatively, that the calculation is $O(N^p)$ (pronounced "order enn pee"). In numerical linear algebra calculations. we typically find $p \leq 3$.

For example, calculating an inner product $\mathbf{a}^\dagger\mathbf{b} = \sum_{i=1}^{N} a_i^* b_i$ requires N multiplications and $N-1$ additions. Both scale the same with the dimension N, so calculating an inner product

is $O(N)$. Calculating a matrix-vector product $\mathbf{v}' = \mathbf{A}\mathbf{v}$, or $v'_i = \sum_{j=1}^{N} A_{ij}v_j;\ i = 1, ..., N$, requires N^2 multiplications and $N(N-1)$ additions. Again, both scale the same with N, so matrix-vector multiplication is $O(N^2)$. Finally, a matrix-matrix multiplication $\mathbf{C} = \mathbf{A}\mathbf{B}$, or $C_{ik} = \sum_{j=1}^{N} A_{ij}B_{jk};\ i = 1, ..., N;\ k = 1, ..., N$ is $O(N^3)$.[9] In a compound algorithm, which consists of several of such steps, the highest power of N dominates for N sufficiently large. So, if the algorithm contains a matrix-matrix multiplication somewhere, it is $O(N^3)$.

A single iteration step in the power iteration algorithm contains a matrix-vector product as its most costly operation. It thus scales as $O(N^2)$. The algorithm as a whole scales as $O(nN^2)$, where n is the number of cycles required before convergence. There is no telling on forehand how large n will be, as that depends on how the eigenvalues are spaced.[10] If one wishes to obtain all N eigenvalues by subsequent power iteration, using orthogonalization as discussed in Sec. 2.3, then the scaling becomes $O(nN^3)$. For large N, this is so bad, that ministries of health and welfare around the world are strongly advising against its use.[11]

Things become much better if $\mathbf{A} = \mathbf{T}$ is tridiagonal. The matrix-vector product $\mathbf{v}' = \mathbf{A}\mathbf{v}$, $v'_i = A_{i,i-1}v_i + A_{i,i}v_i + A_{i,i+1}v_{i+1};\ i = 1, ..., N$ requires $3N$ multiplications and is thus $O(N)$. Power iteration on a single eigenvalue/vector is then $O(nN)$, which is not bad at all. Power iteration to obtain all eigenvalues/vectors is $O(nN^2)$. In principle this is not so bad either, but there are better alternatives (QR iteration), which we will discuss below.

The number of iterations is determined by the leading error in Eq. 2.8, which is $O\left(|\lambda_2/\lambda_1|^{2n+1}\right)$. In other words, if $|\lambda_2/\lambda_1| \ll 1$, then power iteration converges fast. However, if $|\lambda_2/\lambda_1| \approx 1$, convergence is slow and we have a problem. For inverse power iteration, the leading error is $\min_j O\left(|\lambda_{\min} - s|^{2n+1} / |\lambda_j - s|^{2n+1}\right)$, so we are OK if $|(\lambda_{\min} - s)/(\lambda_j - s)| \ll 1$ for all j, and we are in trouble is that is not the case. Obviously the choice of shift s plays a decisive role here. There are strategies to find optimal shifts, which are incorporated in the routines found in the the standard numerical linear algebra libraries. Note that this problem reoccurs for each eigenvalue/vector M you need to calculate. In power iteration, the leading error is given by $O\left(|\lambda_{M+1}/\lambda_M|^{2n+1}\right)$, so only if $|\lambda_{M+1}/\lambda_M| \ll 1$, then power iteration converges fast.

2.3.2 Rayleigh quotient iteration

If $\mathbf{b}$ is close to the eigenvector $\mathbf{x}_1$ belonging to $\lambda_{\min}$, and both vectors are normalized, then $R = \mathbf{b}^\dagger\mathbf{A}\mathbf{b} \approx \mathbf{x}_1^\dagger\mathbf{A}\mathbf{x}_1 = \lambda_{\min}$. In other words, one expects the Rayleigh quotient R to be close to the eigenvalue $\lambda_{\min}$. In that case, R is an optimal choice for the shift s in inverse power iteration, in the sense of obeying Eq. 2.13. One has to modify the algorithm of Eq. 2.14 to incorporate the calculation of the Rayleigh quotient

$$\mathbf{b}_0 = \frac{\mathbf{b}}{\|\mathbf{b}\|};\ R_0 = \mathbf{b}_0^\dagger\mathbf{A}\mathbf{b}_0;$$
$$\text{do } n = 1, ...\ \ \mathbf{L}\mathbf{U} = \mathbf{A} - R_{n-1}\mathbf{I};\ \mathbf{L}\mathbf{U}\mathbf{b}_n = \mathbf{b}_{n-1};\ \mathbf{b}_n = \frac{\mathbf{b}_n}{\|\mathbf{b}_n\|};\ R_n = \mathbf{b}_n^\dagger\mathbf{A}\mathbf{b}_n;\quad \text{until converged.} \tag{2.19}$$

[9] Surprisingly, there are algorithms for matrix-matrix multiplication that scale better than that. For instance, the Strassen algorithm scales as $O(N^{\log_2 7})$. As $\log_2 7 \approx 2.81 < 3$, it makes a noticeable difference for N sufficiently large. However, for our estimates we stick to $O(N^3)$.

[10] Unfortunately, in many practical problems it is not so uncommon to find that $n \propto N$.

[11] The US are opting out, as it is not in their national interest. The British are against it, because they are British, and the nos have it.

This variant of shifted inverse power iteration is known as *Rayleigh quotient iteration.*

Obviously, if R is not a good approximation to $\lambda_{\min}$, the algorithm does not work well. In practice, it mostly converges, but it can converge to the wrong eigenvalue $\lambda_i \neq \lambda_{\min}$, depending very much on the choice of the initial vector $\mathbf{b}_0$. So for the algorithm to work we need a vector $\mathbf{b}_0$ that is sufficiently close to $\mathbf{x}_1$. It would pay off, because, if Rayleigh quotient iteration converges, it converges much faster than inverse power iteration with a fixed shift s. A practical strategy then could be to use inverse power iteration with a fixed shift s until the vector $\mathbf{b}_n$ is sufficiently close to $\mathbf{x}_1$. Then switch to Rayleigh quotient iteration for the final end spurt to a converged result.

One can avoid calculating the Rayleigh quotient as in Eq. 2.19, if one wishes to avoid an explicit calculation of $\mathbf{A}\mathbf{b}_n$. As $(\mathbf{A} - R_{n-1}\mathbf{I})\,\mathbf{b}_n = \mathbf{b}_{n-1}$, we have $\mathbf{A}\mathbf{b}_n = \mathbf{b}_{n-1} + R_{n-1}\mathbf{b}_n$. Then

$$R_n = \frac{\mathbf{b}_n^\dagger \mathbf{A}\mathbf{b}_n}{\mathbf{b}_n^\dagger \mathbf{b}_n} = \frac{\mathbf{b}_n^\dagger \mathbf{b}_{n-1}}{\|\mathbf{b}_n\|^2} + R_{n-1}. \tag{2.20}$$

This can be used in Eq. 2.19 before the normalization step. (After calculating the Rayleigh quotient, one can normalize $\mathbf{b}_n$.)

There is a clear drawback to Eq. 2.19. Each cycle requires the solution of a linear problem. In Eq. 2.19 this is done by LU factorization, and in contrast to inverse power iteration with a fixed shift, Eq. 2.14, this has to be done in each cycle. As LU factorization of a full matrix is an $O(N^3)$ process, it is costly, and rapidly becomes too costly for increasing N. If $\mathbf{A}$ is a tridiagonal matrix, then $\mathbf{A} - R_{n-1}\mathbf{I}$ is also a tridiagonal matrix, and LU factorization is only a $O(N)$ process. In that case, Rayleigh quotient iteration becomes more attractive. Alternatively, the linear problem $(\mathbf{A} - R_{n-1}\mathbf{I})\,\mathbf{b}_n = \mathbf{b}_{n-1}$ in Eq. 2.19 can be solved via an iterative method, see the lecture notes on (non-)linear problems. This adds an extra do-loop inside the one of Eq. 2.19.

As with (inverse) power iteration, the Rayleigh quotient iteration can be modified to find the second eigenvalue by repeated orthogonalization on the first eigenvector.

2.4 Diagonalization by unitary transformations

There are situations in which we are interested in obtaining all eigenvalues, $\lambda_1, \lambda_2, ..., \lambda_N$, of a matrix. The collection of all eigenvalues is called the *spectrum.* In many physical problems, the spectrum plays an important role, and even approximate knowledge of it helps. Searching for $\lambda_1, \lambda_2, ..., \lambda_N$, one at a time, as in power iteration, is then not the most efficient approach. We will set up an algorithm with which you can obtain all the eigenvalues simultaneously. I will first explain some of the linear algebra folklore involved, and then explain the general procedure in Sec. 2.4.3. As you will see, this procedure involves two steps, which are explained in Secs. 2.5 and 2.6, respectively.

A well-known theorem of linear algebra states that, if $\mathbf{A}$ is Hermitian, then a specific *unitary* matrix $\mathbf{U}$ exists, such that

$$\mathbf{U}^\dagger \mathbf{A}\mathbf{U} = \mathbf{\Lambda}, \quad \text{with} \quad \mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 & 0 \\ 0 & \lambda_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_{N-1} & 0 \\ 0 & 0 & \cdots & 0 & \lambda_N \end{pmatrix}, \tag{2.21}$$

a diagonal matrix, with the eigenvalues on the diagonal. A unitary matrix has the property $\mathbf{U}^\dagger = \mathbf{U}^{-1}$; some of the properties of unitary matrices are discussed in Sec. 2.4.1. A transformation $\mathbf{B} = \mathbf{S}^{-1}\mathbf{A}\mathbf{S}$ is called a *similarity transformation*, and when $\mathbf{S}$ is unitary, it is called a *unitary transformation*. The unitary transformation of $\mathbf{A}$ to a diagonal matrix $\mathbf{\Lambda}$, Eq. 2.21, is called the *diagonalization* of the matrix $\mathbf{A}$.[12]

Eq. 2.21 is one of the more useful theorems of linear algebra when it comes to applications in physics.

Proof: of Eq. 2.21. We have already shown that the eigenvectors $\mathbf{x}_i$ of a Hermitian matrix form an orthogonal basis set. Define $\mathbf{u}_i = \mathbf{x}_i / \|\mathbf{x}_i\|$; $i = 1, ..., N$ as the basis set of normalized eigenvectors, i.e. $\mathbf{u}_i^\dagger \mathbf{u}_j = \delta_{ij}$. Construct a matrix by writing these vectors as columns, $\mathbf{U} = (\ \mathbf{u}_1\ \ \mathbf{u}_2\ \ \cdots\ \ \mathbf{u}_{N-1}\ \ \mathbf{u}_N\ )$, then $\mathbf{U}\mathbf{U}^\dagger = \mathbf{U}^\dagger\mathbf{U} = \mathbf{I}$. In other words, this matrix is unitary; see the next section. Rewrite the eigenvalue equation, Eq. 2.1, as $\mathbf{A}\mathbf{U} = \mathbf{U}\mathbf{\Lambda}$. Multiply both terms from the left by $\mathbf{U}^\dagger$, and we have $\mathbf{U}^\dagger\mathbf{A}\mathbf{U} = \mathbf{\Lambda}$.

If we had our specific $\mathbf{U}$, our problem would be solved, as Eq. 2.1 can be transformed into

$$\mathbf{U}^\dagger\mathbf{A}\mathbf{x}_n = \mathbf{U}^\dagger\lambda_n\mathbf{x}_n = \lambda_n\mathbf{U}^\dagger\mathbf{x}_n \Longleftrightarrow \mathbf{U}^\dagger\mathbf{A}\mathbf{U}\mathbf{U}^\dagger\mathbf{x}_n = \lambda_n\mathbf{U}^\dagger\mathbf{x}_n \Longleftrightarrow \mathbf{\Lambda}\mathbf{e}_n = \lambda_n\mathbf{e}_n, \tag{2.22}$$

with $\mathbf{e}_n = \mathbf{U}^\dagger\mathbf{x}_n$. The solution is then rather trivial: the eigenvalues are given by the numbers on the diagonal of $\mathbf{\Lambda}$, and the eigenvectors $\mathbf{e}_n$ are simply the unit vectors with elements $(e_n)_i = \delta_{ni}$. This means that the eigenvectors of the original problem $\mathbf{x}_n = \mathbf{U}\mathbf{e}_n$ are given by the columns of $\mathbf{U}$, i.e., $(x_n)_i = U_{ni}$.

Our problem is of course that we do not have $\mathbf{U}$, so we have to come up with an algorithm to construct it. First I will give you some general properties of unitary matrices, and a specific example of a unitary matrix we will use later on.

2.4.1 Unitary matrices

A complex, square matrix is called unitary if (i) $\mathbf{U}^\dagger = \mathbf{U}^{-1}$. It follows that (ii) $\mathbf{U}\mathbf{U}^\dagger = \mathbf{U}^\dagger\mathbf{U} = \mathbf{I}$, with $\mathbf{I}$ the identity matrix. As (ii) also implies (i), either statements (i) or (ii) can be used as the definition of a unitary matrix. Some properties are

I. If $\mathbf{U}$ and $\mathbf{V}$ are both unitary matrices, then the product $\mathbf{U}\mathbf{V}$ is also a unitary matrix: $(\mathbf{U}\mathbf{V})^\dagger\,\mathbf{U}\mathbf{V} = \mathbf{V}^\dagger\mathbf{U}^\dagger\mathbf{U}\mathbf{V} = \mathbf{V}^\dagger\mathbf{V} = \mathbf{I}$. Unitary matrices form a *group* (in the mathematical sense), which is usually indicated by the symbol $U(N)$, where N is the dimension of the matrices.[13] Such groups play an important role in quantum field theory, for instance.[14] Geometrical operations such as rotations or reflections, can be represented by unitary matrices. We will see an example in the next section.[15]

[12] Diagonalization using similarity transformations can be extended to non-Hermitian matrices, but not with unitary transformations. This generalization is somewhat tedious, so I will skip it.

[13] Oddly enough, mathematicians don't form a group in the mathematical sense.

[14] For instance, the Pauli spin matrices σ_x, σ_y, σ_z plus the 2×2 identity matrix, form the group $U(2)$. Bono is not a member.

[15] Hermitian matrices do not form a group. A product of two Hermitian matrices is not Hermitian in general: $(\mathbf{A}\mathbf{B})^\dagger = \mathbf{B}^\dagger\mathbf{A}^\dagger = \mathbf{B}\mathbf{A} \neq \mathbf{A}\mathbf{B}$.

II. If $\mathbf{C} = \mathbf{U}^\dagger\mathbf{B}\mathbf{U}$, with $\mathbf{U}$ a unitary matrix, and $\mathbf{B}$ a Hermitian matrix, then $\mathbf{C}^\dagger = \left(\mathbf{U}^\dagger\mathbf{B}\mathbf{U}\right)^\dagger = \mathbf{U}^\dagger\mathbf{B}\mathbf{U} = \mathbf{C}$, is a Hermitian matrix. In other words, unitary transformations conserve the Hermitian property.

III. Writing $\mathbf{U} = \begin{pmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_{N-1} & \mathbf{u}_N \end{pmatrix}$, where $\mathbf{u}_i$ is the column vector with elements $U_{i1}, U_{i2}, ..., U_{i,N-1}, U_{iN}$, it follows from (ii) that $\mathbf{u}_i^\dagger\mathbf{u}_j = \delta_{ij}$. This means that the columns of a unitary matrix are orthogonal and normalized vectors. The same holds for the rows of a unitary matrix.

IV. As $\det(\mathbf{I}) = 1$, it follows that $\det(\mathbf{U}\mathbf{U}^\dagger) = \det(\mathbf{U})\det(\mathbf{U}^\dagger) = \det(\mathbf{U})\det(\mathbf{U})^* = 1$, or $|\det(\mathbf{U})| = 1$.

V. A unitary transformation conserves the inner product

$$(\mathbf{U}\mathbf{a})^\dagger(\mathbf{U}\mathbf{b}) = \mathbf{a}^\dagger\mathbf{U}^\dagger\mathbf{U}\mathbf{b} = \mathbf{a}^\dagger\mathbf{I}\mathbf{b} = \mathbf{a}^\dagger\mathbf{b}. \tag{2.23}$$

Setting $\mathbf{b} = \mathbf{a}$, it follows that a unitary transformation conserves the norm

$$\|\mathbf{U}\mathbf{a}\| = \|\mathbf{a}\|. \tag{2.24}$$

2.4.2 Householder reflections*

An example of a unitary matrix that we will use in the following, is the reflection

$$\mathbf{P} = \mathbf{I} - 2\mathbf{u}\mathbf{u}^\dagger \quad \text{or in components} \quad P_{ij} = \delta_{ij} - 2u_iu_j^*, \tag{2.25}$$

where $\mathbf{u}$ is a fixed column vector with unit norm, $\|\mathbf{u}\| = 1$. Note that the product of a column vector and a row vector $\mathbf{u}\mathbf{u}^\dagger = \mathbf{Z}$ is *not* a number, but a matrix with elements $Z_{ij} = u_iu_j^*$. As $\mathbf{P}^\dagger = \mathbf{I}^\dagger - \left(2\mathbf{u}\mathbf{u}^\dagger\right)^\dagger = \mathbf{I} - 2\mathbf{u}\mathbf{u}^\dagger = \mathbf{P}$, this matrix is in fact Hermitian. It is also unitary

$$\mathbf{P}^\dagger\mathbf{P} = \mathbf{P}^2 = \mathbf{I} - 4\mathbf{u}\mathbf{u}^\dagger + 4\mathbf{u}\left(\mathbf{u}^\dagger\mathbf{u}\right)\mathbf{u}^\dagger = \mathbf{I} - 4\mathbf{u}\mathbf{u}^\dagger + 4\|\mathbf{u}\|^2\mathbf{u}\mathbf{u}^\dagger = \mathbf{I}.$$

To demonstrate that $\mathbf{P}$ represents a reflection, let it operate on a vector

$$\mathbf{P}\mathbf{a} = \mathbf{a} - 2\mathbf{u}(\mathbf{u}^\dagger\mathbf{a}), \tag{2.26}$$

and consider the geometrical construction shown in Fig. 2.1. $\mathbf{P}$ represents a reflection with respect to a (hyper)plane defined by its normal $\mathbf{u}$. In the context we use it here, it is called a *Householder reflection.*[16]

2.4.3 Diagonalization strategy

Our strategy is to write $\mathbf{U}$ in Eq. 2.21 as a product

$$\mathbf{U} = \mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3...\mathbf{Q}_M, \tag{2.27}$$

where each of the $\mathbf{Q}_i$ is a "simple" unitary matrix. Each unitary transformation $\mathbf{Q}_i^\dagger\mathbf{B}\mathbf{Q}_i = \mathbf{C}$ is supposed to make the matrix $\mathbf{C}$ closer to a diagonal matrix than the matrix $\mathbf{B}$ (in

[16] A reflection is a concept so elementary, that it seems strange to give it a special name. Householder formulated numerical linear algebra algorithms on the basis of these reflections. Hence they were given his name.

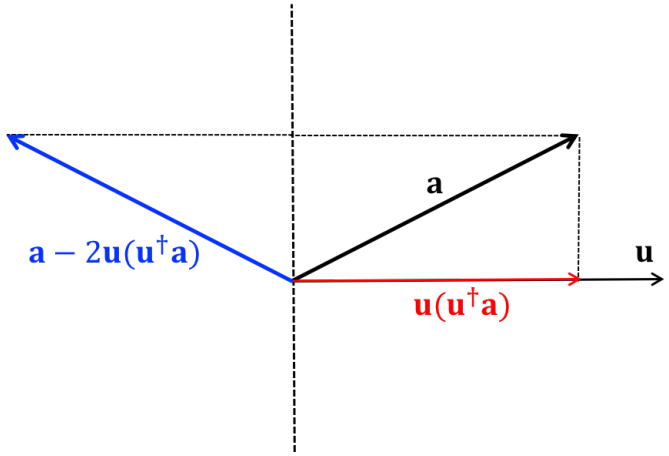


Figure 2.1: $\mathbf{u}\left(\mathbf{u}^\dagger\mathbf{a}\right)$ represents the projection of $\mathbf{a}$ onto $\mathbf{u}$, where $\mathbf{a}$ is any vector, and $\mathbf{u}$ is a unit vector. Then $\mathbf{a}-2\mathbf{u}(\mathbf{u}^\dagger\mathbf{a})$ is the vector obtained by reflecting $\mathbf{a}$ with respect to a plane (dashed line) of which $\mathbf{u}$ is the normal.

what sense is explained below). Despite all all the king's horses and all the king's men, all algorithms used today are still variations on the power iteration theme discussed above. So, similar to Eq. 2.9, all available algorithms are iterative, and one has to keep adding matrices $\mathbf{Q}_i$ "until converged". Whenever that happens depends on the details of the spectrum, in particular how the eigenvalues, $\lambda_1, \lambda_2, ..., \lambda_N$, are spaced. This means that one cannot predict on forehand the number of terms M in Eq. 2.27 that are required to do the job.[17]

This presents a problem. Matrix multiplications are expensive, so one would like to avoid them as much as possible. A full matrix product $\mathbf{C}=\mathbf{AB}$, $C_{ij}=\sum_{k=1}^{N}A_{ik}B_{kj}$; $i=1,...,N$; $j=1,...,N$, requires N^3 multiplications. So it scales as $O(N^3)$, meaning that the full algorithm would scale as $O(MN^3)$, which makes it unhealthily time-consuming. If, however, all the matrices would have a special form, such as tridiagonal, then the matrix product becomes much cheaper. $C_{ij}=\sum_{k=i-1}^{i+1}A_{ik}B_{kj}$; $i=1,...,N$; $j=i-1,i,i+1$ only requires $9N$ multiplications, and thus scales as $O(N)$.

Luckily for us, algorithms exist that can convert any matrix $\mathbf{A}$ into a tridiagonal one $\mathbf{T}$, by a sequence of unitary transformations. As will be shown in Sec. 2.5, we can construct a matrix $\mathbf{V}=\mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3...\mathbf{Q}_{N-2}$ that has the property

$$\mathbf{V}^\dagger\mathbf{A}\mathbf{V}=\mathbf{T},\quad\text{with}\quad\mathbf{T}=\begin{pmatrix}t_{11}&t_{12}&0&0&0\\t_{21}&t_{22}&t_{23}&0&0\\0&t_{32}&\ddots&\vdots&\vdots\\0&0&\cdots&\ddots&t_{N-1,N}\\0&0&\cdots&t_{N,N-1}&t_{NN}\end{pmatrix}, \tag{2.28}$$

a tridiagonal matrix. The strength of this construction is that it is guaranteed to work with maximally $N-2$ terms $\mathbf{Q}_i$. Moreover, calculating the individual products $\mathbf{Q}_i^\dagger\mathbf{B}\mathbf{Q}_i$ is relatively cheap, and the whole transformation of Eq. 2.28 scales as $O(N^3)$. For obvious reasons the transformation is called *tridiagonalization* of the matrix $\mathbf{A}$. Because it involves a predictable number of operations, in practice this step is almost *always* performed on matrices that are *not tridiagonal*, before the actual eigenvalue search is started.

[17] That is, for a general matrix $\mathbf{A}$; for special matrices it may be possible.

In a second step, to be discussed in Sec. 2.6, we construct a matrix $\mathbf{W} = \mathbf{Q}_{N-1}\mathbf{Q}_N\mathbf{Q}_{N+1}...\mathbf{Q}_M$ that has the property

$$\mathbf{W}^\dagger\mathbf{T}\mathbf{W} = \mathbf{\Lambda}, \quad \text{with} \quad \mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 & 0 \\ 0 & \lambda_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_{N-1} & 0 \\ 0 & 0 & \cdots & 0 & \lambda_N \end{pmatrix}. \tag{2.29}$$

This construction is *not* guaranteed to work with a small number of terms M, which make it seem that we have not gained much. However, because these transformations on tridiagonal matrices are relatively cheap, one can afford doing many of them. In fact, I will show that the transformation of Eq. 2.29 is $O(MN)$. This means that the transformation of Eq. 2.28 is in practice the most time-consuming step.

The unitary transformation in Eq. 2.21 can then be written as $\mathbf{U} = \mathbf{V}\mathbf{W}$.

2.5 Tridiagonalization by Householder transformation*

There are zillions of algorithms to tridiagonalize a Hermitian matrix, Eq. 2.28.[18] The current champion (for non-sparse matrices) in terms of speed and robustness is explained below. It uses Householder reflections, and is called the *Householder transformation*. Start with

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1N} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2N} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & \cdots & a_{NN} \end{pmatrix}, \tag{2.30}$$

and transform this into

$$\mathbf{A}' = \mathbf{Q}_1^\dagger\mathbf{A}\mathbf{Q}_1 = \begin{pmatrix} a_{11} & a'_{12} & 0 & \cdots & 0 \\ a'_{21} & a'_{22} & a'_{23} & \cdots & a'_{2N} \\ 0 & a'_{32} & a'_{33} & \cdots & a'_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a'_{N2} & a'_{N3} & \cdots & a'_{NN} \end{pmatrix}, \tag{2.31}$$

i.e., clean up the first row and first column by making $a'_{1j} = a'_{j1} = 0$ for $j > 2$. Note that the corner element a_{11} stays untouched. Then apply a second transformation

$$\mathbf{A}'' = \mathbf{Q}_2^\dagger\mathbf{A}'\mathbf{Q}_2 = \begin{pmatrix} a_{11} & a'_{12} & 0 & \cdots & 0 \\ a'_{21} & a'_{22} & a''_{23} & \cdots & 0 \\ 0 & a''_{32} & a''_{33} & \cdots & a''_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a''_{N3} & \cdots & a''_{NN} \end{pmatrix}, \tag{2.32}$$

i.e., clean up the second row and second column by making $a''_{2j} = a''_{j2} = 0$ for $j > 3$. Note that the first row and first column, as well as the element a'_{22}, stay untouched. The algorithm

[18] Well, at least more than two.

proceeds with a third transformation that cleans up the third row and third column, leaving the first two rows and columns, as well as the element a''_{33} untouched, etcetera. The algorithm ends by a $(N-2)^{\text{th}}$ transformation cleaning up the $(N-2)^{\text{th}}$ row and column, after which the transformed matrix is fully tridiagonal. The full transformation matrix needed in Eq. 2.28 is then given by $\mathbf{V} = \mathbf{Q}_1\mathbf{Q}_2...\mathbf{Q}_{N-2}$.

Let us now see what the individual transformations look like. Choose

$$\mathbf{Q}_1 = \begin{pmatrix} 1 & \mathbf{0}^\dagger \\ \mathbf{0} & \mathbf{P} \end{pmatrix}, \tag{2.33}$$

with $\mathbf{0}$ a column vector of $N-1$ zeros, $\mathbf{0}^\dagger$ a row vector of $N-1$ zeros,[19] and $\mathbf{P}$ a unitary $(N-1)\times(N-1)$ matrix. Using a $\mathbf{Q}_1$ as in Eq. 2.33 ensures that a_{11} is left untouched in the transformation of Eq. 2.31

$$\mathbf{A}' = \mathbf{Q}_1^\dagger \mathbf{A}\mathbf{Q}_1 = \begin{pmatrix} 1 & \mathbf{0}^\dagger \\ \mathbf{0} & \mathbf{P}^\dagger \end{pmatrix} \begin{pmatrix} a_{11} & \mathbf{a}^\dagger \\ \mathbf{a} & \boldsymbol{A} \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0}^\dagger \\ \mathbf{0} & \mathbf{P} \end{pmatrix} = \begin{pmatrix} a_{11} & (\mathbf{P}\mathbf{a})^\dagger \\ \mathbf{P}\mathbf{a} & \mathbf{P}^\dagger \boldsymbol{A} \mathbf{P} \end{pmatrix}, \tag{2.34}$$

where $\mathbf{a}^\dagger = (a_{12}, a_{13}, ..., a_{1N})$ is a row vector of $N-1$ elements consisting of the first row of $\mathbf{A}$, Eq. 2.30, without the corner element a_{11}; $\mathbf{a}$ is the column vector consisting of the first column of $\mathbf{A}$ without the corner element a_{11}; $\boldsymbol{A}$ is the $(N-1)\times(N-1)$ matrix obtained from $\mathbf{A}$ by removing the first row and the first column.

In order get the required form of Eq. 2.31, we must demand

$$\mathbf{P}\mathbf{a} = a'_{12}\mathbf{e}, \tag{2.35}$$

where $\mathbf{e}$ is the unit vector of length $N-1$ with elements $e_i = \delta_{1i}$. Note that $\|\mathbf{P}\mathbf{a}\|^2 = |a'_{12}|^2$. As a unitary transformation conserves the norm of a vector, Eq. 2.24, we also have

$$\|\mathbf{P}\mathbf{a}\|^2 = \|\mathbf{a}\|^2 \Rightarrow |a'_{12}|^2 = \sum_{i=2}^{N} |a_{1i}|^2 \quad \text{or} \quad a'_{12} = e^{i\phi}\|\mathbf{a}\|, \tag{2.36}$$

with ϕ a phase factor we can choose freely.

After we have found a matrix $\mathbf{P}$ that obeys Eq. 2.35, we choose the second transformation as

$$\mathbf{Q}_2 = \begin{pmatrix} 1 & 0 & \mathbf{0}^\dagger \\ 0 & 1 & \mathbf{0}^\dagger \\ \mathbf{0} & \mathbf{0} & \mathbf{P}' \end{pmatrix} \tag{2.37}$$

with $\mathbf{0}^\dagger$ a row vector of $N-2$ zeros, and $\mathbf{P}'$ a unitary $(N-2)\times(N-2)$ matrix. The transformation of Eq. 2.32 then becomes

$$\mathbf{A}'' = \mathbf{Q}_2^\dagger \mathbf{A}'\mathbf{Q}_2 = \begin{pmatrix} a_{11} & a'_{12} & \mathbf{0}^\dagger \\ a'_{21} & a'_{22} & (\mathbf{P}'\mathbf{a}')^\dagger \\ \mathbf{0} & \mathbf{P}'\mathbf{a}' & \mathbf{P}'^\dagger \boldsymbol{A}' \mathbf{P}' \end{pmatrix}, \tag{2.38}$$

where $\mathbf{a}'^\dagger = (a'_{23}, a'_{24}, ..., a'_{1N})$ is a row vector of $N-2$ elements with the corresponding matrix elements of $\mathbf{A}'$, see Eq. 2.31, $\mathbf{a}'$ is the corresponding column vector, and $\boldsymbol{A}'$ is the

[19] I apologize for the notation, but my mathematical friends insist on this. It is good to have a few mathematical friends. Too many, and life becomes unbearable.

$(N-2)\times(N-2)$ matrix obtained from $\mathbf{A}'$ by removing the first two rows and the first two columns. To get the form required by Eq. 2.32, we must demand

$$\mathbf{P}'\mathbf{a}' = a''_{23}\mathbf{e}', \tag{2.39}$$

where $\mathbf{e}'$ is the unit vector of length $N-2$ with elements $e'_i = \delta_{1i}$, where we have

$$\|\mathbf{P}'\mathbf{a}'\|^2 = \|\mathbf{a}'\|^2 \Rightarrow |a''_{23}|^2 = \sum_{i=3}^{N} |a'_{2i}|^2. \tag{2.40}$$

Comparing Eqs. 2.37-2.40 to Eqs. 2.33-2.36 one observes that transformation using $\mathbf{Q}_2$ is completely analogous to that using $\mathbf{Q}_1$, except that the dimension of the non-trivial part of the matrix $\mathbf{P}'$ is $N-2$, instead of $N-1$ as in $\mathbf{P}$. The subsequent transformations $\mathbf{Q}_n$; $n = 3, ..., N-2$ are analogous, involving matrices $\mathbf{P}_n$ of dimension $N-n$.

2.5.1 Apply Householder reflections

All that rests now is to find the $\mathbf{P}_n$ matrices. I will explain the first one $\mathbf{P}_1 \equiv \mathbf{P}$, which has dimension $N-1$. The others can be found in the same way. We will use the Householder reflection, Eq. 2.26 and Fig. 2.1. Fig. 2.2 illustrates what Eqs. 2.35 and 2.36 mean: one has to find a $\mathbf{u}$, such that the reflection of $\mathbf{a}$ is in the direction of $\mathbf{e}$.

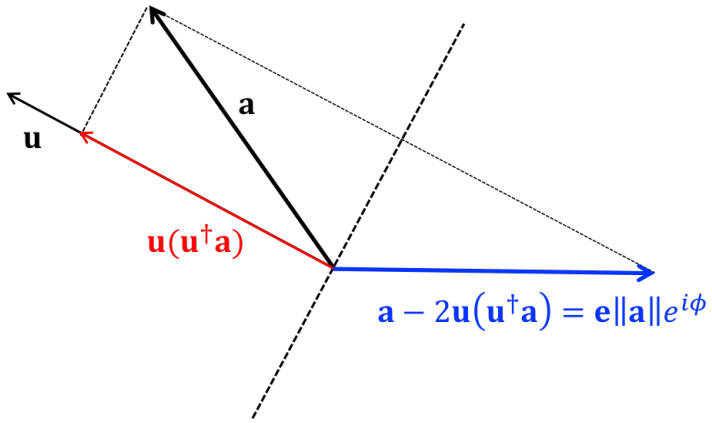


Figure 2.2: Eqs. 2.35 and 2.36 demand that the reflection $\mathbf{a} - 2\mathbf{u}(\mathbf{u}^\dagger\mathbf{a})$ of the vector $\mathbf{a}$ is along the direction $\mathbf{e}$, where $e_i = \delta_{1i}$. One has to find a reflection plane, with a normal $\mathbf{u}$, that does this.

According to Fig. 2.2 this holds for

$$2\mathbf{u}(\mathbf{u}^\dagger\mathbf{a}) = \mathbf{a} - \mathbf{e}\,\|\mathbf{a}\|\,e^{i\phi}. \tag{2.41}$$

Multiplying both sides from the left with $\mathbf{a}^\dagger$ gives

$$2\mathbf{a}^\dagger\mathbf{u}(\mathbf{u}^\dagger\mathbf{a}) = 2\left|\mathbf{u}^\dagger\mathbf{a}\right|^2 = \mathbf{a}^\dagger\mathbf{a} - \mathbf{a}^\dagger\mathbf{e}\,\|\mathbf{a}\|\,e^{i\phi} = \|\mathbf{a}\|^2 - a_{12}\,\|\mathbf{a}\|\,e^{i\phi}. \tag{2.42}$$

Obviously the result has to be a real number, so we have

$$\phi = -\phi_{12} + \pi \text{ or } \phi = -\phi_{12}, \tag{2.43}$$

where $a_{12} = |a_{12}|\, e^{i\phi_{12}}$, which gives

$$\left|\mathbf{u}^\dagger \mathbf{a}\right|^2 = \frac{1}{2}\left\{\|\mathbf{a}\|^2 \pm |a_{12}|\, \|\mathbf{a}\|\right\}. \tag{2.44}$$

The + and − signs correspond to the two choices of phase of Eq. 2.43. Using this result in Eq. 2.41 gives

$$\mathbf{u} = \frac{1}{2\mathbf{u}^\dagger \mathbf{a}}\left(\mathbf{a} - \mathbf{e}\,\|\mathbf{a}\|\, e^{i\phi}\right) = \frac{\left(\mathbf{a} \pm \mathbf{e}\,\|\mathbf{a}\|\, \frac{a_{21}}{|a_{21}|}\right)}{\sqrt{2\left\{\|\mathbf{a}\|^2 \pm |a_{21}|\, \|\mathbf{a}\|\right\}}}, \tag{2.45}$$

where we have used $a_{21} = a_{12}^*$, and $e^{-i\phi_{12}} = a_{12}^*/\,|a_{12}| = a_{21}/\,|a_{21}|$. We can choose either the + or the − signs (consistently in numerator and denominator). For reasons of numerical stability, usually the sign is chosen that gives the largest $\|\mathbf{u}\|$.

The specific Householder reflection, Eq. 2.26, can then be written as

$$\mathbf{P} = \mathbf{I} - \frac{1}{w}\mathbf{v}\mathbf{v}^\dagger \quad \text{with} \quad \mathbf{v} = \mathbf{a} \pm \mathbf{e}\,\|\mathbf{a}\|\, \frac{a_{21}}{|a_{21}|} \quad \text{and} \quad w = \|\mathbf{a}\|\left(\|\mathbf{a}\| \pm |a_{21}|\right), \tag{2.46}$$

The only step left to complete the transformation is to calculate

$$\mathbf{P}^\dagger \boldsymbol{A}\mathbf{P} = \boldsymbol{A} - \frac{1}{w}\mathbf{v}\mathbf{v}^\dagger \boldsymbol{A} - \frac{1}{w}\boldsymbol{A}\mathbf{v}\mathbf{v}^\dagger + \frac{1}{w^2}\mathbf{v}\mathbf{v}^\dagger \boldsymbol{A}\mathbf{v}\mathbf{v}^\dagger = \boldsymbol{A} - \frac{1}{w}\mathbf{v}\mathbf{v}'^\dagger - \frac{1}{w}\mathbf{v}'\mathbf{v}^\dagger + \frac{v''}{w^2}\mathbf{v}\mathbf{v}^\dagger \tag{2.47}$$

with $\mathbf{v}' = \boldsymbol{A}\mathbf{v}$, and $v'' = \mathbf{v}^\dagger \boldsymbol{A}\mathbf{v}$.

2.5.1.1 Cost and stability

The Householder transformation seems to be more stable than other tridiagonalization algorithms, which explains its popularity. Calculating the cost of the Householder transformation can be done using the simple rules introduced before. Calculating an inner product $\mathbf{a}^\dagger\mathbf{b}$ scales as $O(N)$, whereas calculating a matrix-vector product $\mathbf{v}' = \mathbf{A}\mathbf{v}$ scales as $O(N^2)$. Calculating a product $\mathbf{Z} = \mathbf{v}'\mathbf{v}^\dagger$ also scales as $O(N^2)$, as $Z_{ij} = v_i' v_j^*$; $i = 1, ..., N$; $j = 1, ..., N$ requires N^2 multiplications. As before, if N is sufficiently large, one can neglect $O(N)$ terms, if $O(N^2)$ terms are present. The transformation of Eq. 2.47 is therefore $O(N^2)$. As we have to do $N - 2$ subsequent transformations of this sort in the Householder algorithm, the whole Householder transformation is $O(N^3)$. The matrices that do most work in the transformation have smaller and smaller dimension as the transformation progresses, but the leading term in the costs still scales as αN^3, with α some fixed number.

2.6 Diagonalization by QR iteration

After the Householder transformation, we end up with a tridiagonal Hermitian matrix $\mathbf{T}$, which we still have to diagonalize, see Eqs. 2.28 and 2.29. There are zillions of algorithms to do this.[20] I will give you the currently most popular one, which is called *QR iteration*. It is actually a very clever variant of simultaneous power iteration, so I will start with the latter.

[20] Well, more than two at least.

2.6.1 Simultaneous power iteration

Discussing the power iteration method in Sec. 2.3, we argued that two eigenvalues λ_1, λ_2 can be obtained the following way. First find λ_1 by power iterating on a starting vector $\mathbf{b}$ until it converges to the eigenvector $\mathbf{x}_1$, see Eq. 2.9. Then find λ_2 by power iterating on a starting vector $\mathbf{b}'$ under the constraint of remaining orthogonal to $\mathbf{x}_1$, until it converges to the eigenvector $\mathbf{x}_2$, see Eq. 2.10.

One can try to find the two eigenvalues/vectors *simultaneously* is to merge the two algorithms of Eqs. 2.9 and 2.10 into one

$$\mathbf{b}_0 = \frac{\mathbf{b}}{\|\mathbf{b}\|};\quad \mathbf{b}_1 = \mathbf{A}\mathbf{b}_0;\ R_0 = \mathbf{b}_0^\dagger \mathbf{b}_1;\ \mathbf{b}_1 = \frac{\mathbf{b}_1}{\|\mathbf{b}_1\|};$$

$$\mathbf{b}'_0 = \frac{\mathbf{b}'}{\|\mathbf{b}'\|};\ \mathbf{b}'_1 = \mathbf{A}\mathbf{b}'_0;\ \mathbf{b}'_1 = \mathbf{b}'_1 - \mathbf{b}_1(\mathbf{b}_1^\dagger \mathbf{b}'_1);\ R'_0 = \mathbf{b}_0'^\dagger \mathbf{b}'_1;\ \mathbf{b}'_1 = \frac{\mathbf{b}'_1}{\|\mathbf{b}'_1\|} \tag{2.48}$$

$$\text{do } n = 1, \ldots\ \mathbf{b}_{n+1} = \mathbf{A}\mathbf{b}_n;\ R_n = \mathbf{b}_n^\dagger \mathbf{b}_{n+1};\ \mathbf{b}_{n+1} = \frac{\mathbf{b}_{n+1}}{\|\mathbf{b}_{n+1}\|}; \tag{2.49}$$

$$\mathbf{b}'_{n+1} = \mathbf{A}\mathbf{b}'_n;\ \mathbf{b}'_{n+1} = \mathbf{b}'_{n+1} - \mathbf{b}_{n+1}(\mathbf{b}_{n+1}^\dagger \mathbf{b}'_{n+1});\ R'_n = \mathbf{b}_n'^\dagger \mathbf{b}'_{n+1};\ \mathbf{b}'_{n+1} = \frac{\mathbf{b}'_{n+1}}{\|\mathbf{b}'_{n+1}\|};\quad \text{until converged.}$$

Not extremely elegant perhaps, but the general idea should be clear. We start with two vectors $\mathbf{b}$ and $\mathbf{b}'$, orthogonalize them and use power iteration simultaneously on these vectors. To avoid both of them converging to the same eigenvector $\mathbf{x}_1$, we keep on orthogonalizing the second one to the first one in the step $\mathbf{b}'_{n+1} = \mathbf{b}'_{n+1} - \mathbf{b}_{n+1}(\mathbf{b}_{n+1}^\dagger \mathbf{b}'_{n+1})$. After a sufficient number of cycles the vector $\mathbf{b}_n$ converges to the first eigenvector $\mathbf{x}_1$, since for this vector we perform power iteration, just as in Eq. 2.9. Its Rayleigh quotient R_n then converges to the first eigenvalue λ_1. The second vector $\mathbf{b}'_n$ is forced to be orthogonal to $\mathbf{b}_n$ all the time, so in the end it will be orthogonal to $\mathbf{x}_1$. Power iteration then ensures that $\mathbf{b}'_n$ converges to a multiple of the second eigenvector $\mathbf{x}_2$, and its Rayleigh quotient R'_n to the second eigenvalue λ_2.[21] This variant of power iteration is called *simultaneous power iteration.* The "until converged" test should then be a criterion of the type $|R_n - R_{n-1}| < \epsilon$ and $\left|R'_n - R'_{n-1}\right| < \epsilon$, with ϵ a tolerance chosen by the user.

If you want to have two eigenvalues, one can either use simultaneous power iteration via Eq. 2.49, or first finding one via Eq. 2.9, and subsequently the other via Eq. 2.10. It is largely a matter of taste, as the costs are similar in both approaches. But, once you are used to the idea of simultaneous power iteration, why not go all the way? Start with N vectors $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$, orthogonalize them, and cycle on them simultaneously, while orthogonalizing and normalizing the vectors between two cycles. The orthogonalization has to be done in strict order: first $\mathbf{b}_2$ is orthogonalized on $\mathbf{b}_1$, then $\mathbf{b}_3$ is orthogonalized on the previous two, then $\mathbf{b}_4$ on the previous three, etcetera.[22] The Rayleigh quotients of the orthogonalized vectors then converge to $\lambda_1, \lambda_2, \ldots, \lambda_N$, respectively, and the powers to the corresponding eigenvectors. In other words, you have diagonalized the whole matrix in one process.

We can write this algorithm in matrix form. Construct a $N \times N$ matrix where the columns are formed by the $\mathbf{b}_i$ vectors: $\mathbf{B} = (\ \mathbf{b}_1\ \ \mathbf{b}_2\ \ \cdots\ \ \mathbf{b}_{N-1}\ \ \mathbf{b}_N\ )$. Operating with $\mathbf{A}$ on these

[21] In principle I should perhaps give a more formal derivation of this, but guess intuitively you agree with me that it should work this way.

[22] The technique is called *Gram-Schmidt orthogonalization*, see the lecture notes on (non-)linear problems.

vectors then becomes the matrix product, $\mathbf{AB}$. If a priori we have no clue about what a good set of starting vectors $\mathbf{b}_i$ would be (and why should we?), then why not choose the unit vectors $\mathbf{e}_i$, with elements $(e_i)_j = \delta_{ij}$? These vectors are orthogonal and normalized, $\mathbf{e}_i^\dagger \mathbf{e}_j = \delta_{ij}$, which saves us the initial orthogonalization and normalization step. Moreover, as $(\ \mathbf{e}_1\ \ \mathbf{e}_2\ \ \cdots\ \ \mathbf{e}_{N-1}\ \ \mathbf{e}_N\ ) = \mathbf{I}$, the identity matrix, we have $\mathbf{AI} = \mathbf{A}$, which saves us the first power step as well.

The columns of $\mathbf{A}$ are in general not orthogonal, i.e., writing $\mathbf{A} = (\ \mathbf{a}_1\ \ \mathbf{a}_2\ \ \cdots\ \ \mathbf{a}_{N-1}\ \ \mathbf{a}_N\ )$, then $\mathbf{a}_i^\dagger \mathbf{a}_j \neq 0$ for $i \neq j$. We know already how to orthogonalize them. Applying *QR factorization*, $\mathbf{QR} = \mathbf{A}$, gives a unitary matrix $\mathbf{Q} = (\ \mathbf{q}_1\ \ \mathbf{q}_2\ \ \cdots\ \ \mathbf{q}_{N-1}\ \ \mathbf{q}_N\ )$, where the columns are orthonormal, $\mathbf{q}_k^\dagger \mathbf{q}_\ell = \delta_{k\ell}$, and an upper triangular matrix $\mathbf{R}$, which defines the transformation between $\mathbf{a}_i;\ i = 1, ..., N$ and $\mathbf{q}_k;\ k = 1, ..., N$.

In the power iteration cycle we let $\mathbf{A}$ operate on the orthogonal vectors, $\mathbf{Aq}_k;\ k = 1, ..., N$, or in matrix form, we construct the matrix $\mathbf{Z} = \mathbf{AQ}$. Next we construct the matrix $\mathbf{Y} = \mathbf{Q}^\dagger \mathbf{AQ} = \mathbf{Q}^\dagger \mathbf{Z}$, which has matrix elements $Y_{k\ell} = \mathbf{q}_k^\dagger \mathbf{A} \mathbf{q}_\ell$. The diagonal elements Y_{kk} correspond to the Rayleigh quotients.

The full-blown *simultaneous power iteration* algorithm then reads

$$\mathbf{Z}_0 = \mathbf{A};\ \ \text{do } n = 1, ...\ \ \mathbf{Q}_n \mathbf{R}_n = \mathbf{Z}_{n-1};\ \ \mathbf{Z}_n = \mathbf{AQ}_n;\ \ \mathbf{Y}_n = \mathbf{Q}_n^\dagger \mathbf{Z}_n;\ \ \text{until converged.} \tag{2.50}$$

The first step within the do-loop represents the orthogonalization step, the second step is the power operation, and the third step calculates the Rayleigh quotients. The criterion "until converged" can be something like $|(Y_n)_{ii} - (Y_{n-1})_{ii}| < \epsilon$, for $i = 1, ..., N$, with ϵ a tolerance chosen by the user. If converged, the matrix $\mathbf{Y}_n$ will be diagonal, with the eigenvalues on its diagonal. The matrix $\mathbf{Q}_n$ contains the corresponding eigenvectors as columns.

Details: As the Rayleigh matrix becomes more and more diagonal while iterating, a stricter convergence criterion could be to test whether the off-diagonal elements of the Rayleigh matrix are close to zero, $\left|(Y_n)_{ij}\right| < \epsilon$, for $i = 1, ..., N$, $j = 1, ..., N$, $i \neq j$. As there are a lot of off-diagonal matrix elements if N is large, this test is usually thought to be too expensive. If, however, the matrix $\mathbf{A}$ is tridiagonal, then the matrices $\mathbf{Y}_n$ are tridiagonal for all n, see Sec. 1.2. So to test whether they are sufficiently close to being diagonal, it is sufficient to test whether $\left|(Y_n)_{i,i+1}\right| < \epsilon$, for $i = 1, ..., N-1$.

2.6.2 The Francis-Kublanovskaya algorithm: QR iteration

There is one trick that makes simultaneous power iteration super brilliant. Each cycle in Eq. 2.50 consists of one QR factorization, followed by calculating the two matrices $\mathbf{Z}_n$ and $\mathbf{Y}_n$. That is one matrix more than is strictly necessary; the matrix $\mathbf{Y}_n$ actually suffices. To show this write

$$\begin{aligned} &\mathbf{Y}_{n-1} = \mathbf{Q}_{n-1}^\dagger \mathbf{Z}_{n-1} = \mathbf{Q}_{n-1}^\dagger \mathbf{Q}_n \mathbf{R}_n \text{ and} \\ &\mathbf{Y}_n = \mathbf{Q}_n^\dagger \mathbf{Z}_n = \mathbf{Q}_n^\dagger \mathbf{AQ}_n = \mathbf{Q}_n^\dagger \mathbf{AQ}_{n-1} \mathbf{Q}_{n-1}^\dagger \mathbf{Q}_n = \mathbf{Q}_n^\dagger \mathbf{Z}_{n-1} \mathbf{Q}_{n-1}^\dagger \mathbf{Q}_n = \mathbf{R}_n \mathbf{Q}_{n-1}^\dagger \mathbf{Q}_n \end{aligned} \tag{2.51}$$

Define a matrix $\mathbf{Q}'_n = \mathbf{Q}_{n-1}^\dagger \mathbf{Q}_n$, which is unitary, then $\mathbf{Y}_{n-1} = \mathbf{Q}'_n \mathbf{R}_n$ and $\mathbf{Y}_n = \mathbf{R}_n \mathbf{Q}'_n$. The algorithm of Eq. 2.50 can then be simplified to

$$\mathbf{Y}_0 = \mathbf{A};\ \text{do } n = 1, \ldots\ \mathbf{Q}'_n\mathbf{R}_n = \mathbf{Y}_{n-1};\ \mathbf{Y}_n = \mathbf{R}_n\mathbf{Q}'_n;\ \text{until converged.} \tag{2.52}$$

This beautifully elegant algorithm is called *QR iteration* or the *Francis-Kublanovskaya algorithm*, after its inventors.[23] The first step within the do-loop represents the QR factorization, and the second step calculates the Rayleigh quotients. Through iteration you find all eigenvalues simultaneously. They just appear as diagonal elements of the matrix $\mathbf{Y}_n$, and approaching convergence that matrix will become diagonal. As before, the criterion "until converged" is then something like $|(Y_n)_{ii} - (Y_{n-1})_{ii}| < \epsilon$, for $i = 1, ..., N$, with ϵ a tolerance chosen by the user.[24]

Because in Eq. 2.52 we are not keeping track of $\mathbf{Q}_n$, but rather of $\mathbf{Q}'_n = \mathbf{Q}^\dagger_{n-1}\mathbf{Q}_n$, the latter converges to $\mathbf{I}$, the identity matrix, or to a matrix derived from the permutation of the columns of $\mathbf{I}$. To be able to find the eigenvectors, one must keep track of $\mathbf{Q}_n = \mathbf{Q}_{n-1}\mathbf{Q}'_n$. As before, the eigenvectors appear as the columns of $\mathbf{Q}_n$. The routine for finding eigenvalues **and** eigenvectors then becomes

$$\mathbf{Y}_0 = \mathbf{A};\ \mathbf{Q}_0 = \mathbf{I};\ \text{do } n = 1, \ldots\ \mathbf{Q}'_n\mathbf{R}_n = \mathbf{Y}_{n-1};\ \mathbf{Q}_n = \mathbf{Q}_{n-1}\mathbf{Q}'_n;\ \mathbf{Y}_n = \mathbf{R}_n\mathbf{Q}'_n;\ \text{until converged.} \tag{2.53}$$

2.6.2.1 Cost, efficiency and deflation

In principle, QR iteration is suited to find all the eigenvalues of any matrix, but to make it efficient it is advisable to use matrices of a special form only. As matrix multiplications scale as $O(N^3)$, and you need these in every cycle of the QR iteration, the algorithm scales as $O(nN^3)$, with n is the number of cycles required before convergence. This is the unhealthy expensive scaling we have been trying to avoid.

For tridiagonal matrices the story is different. In the QR factorization, we start with $\mathbf{A} = \mathbf{T}$ a tridiagonal matrix, $T_{ij} = 0;\ j < i-1$ or $j > i+1$. This will be, for instance, the matrix resulting from the Householder transformation, Eq. 2.28. We have shown in the lecture notes on linear problems that the matrix $\mathbf{Q}$ obtained by QR factorization is tridiagonal, and the matrix $\mathbf{R}$ is upper triangular with a bandwidth of two. The latter means $R_{ij} = 0;\ j < i$ or $j > i+2$. Then the matrix product $Y_{ij} = \sum_{k=j}^{j+2} R_{ik}Q_{kj};\ i = 1, ..., N;\ j = i-1, i, i+1$, required in Eq. 2.52, gives a tridiagonal matrix. In other words, all the matrices involved in QR iteration are tridiagonal. Calculating the matrix product $\mathbf{Y} = \mathbf{RQ}$ involves $9N$ multiplications, which means we have a $O(N)$ scaling. QR iteration on a tridiagonal matrix then has $O(nN)$ scaling. This is the same favorable scaling we had for obtaining a single eigenvalue using power iteration, but QR iteration finds all the eigenvalues in one go. Elegant!

The convergence of QR iteration, or lack thereof, is subjected to the same conditions as power iteration.[25] One needs a spectrum that obeys Eq. 2.5. If some eigenvalues are too close, i.e., if $|\lambda_i/\lambda_j| \approx 1$ for some i and j, then convergence is slow. Again the standard trick to alleviate this problem is to find an appropriate shift s, such that $|(\lambda_i - s)\,/\,(\lambda_j - s)| \ll 1$.

[23] Should there ever be a prize for elegancy of algorithms, this one is one of my favorites.

[24] Or, alternatively, you can check whether the off-diagonal elements $\left|(Y_n)_{ij}\right| < \epsilon$.

[25] Or, as the philosopher says: "elluk foordeil hep sun nadeil".

Within the QR algorithm there are ways of estimating these shifts automatically (*Wilkinson shifts*). They are incorporated in the routines found in the standard numerical linear algebra libraries.

Details: While iterating, some eigenvalues tend to converge faster than others. It makes little sense to keep iterating eigenvalues that are already converged. Call λ_1 an eigenvalue already converged, with $\mathbf{x}_1$ the corresponding eigenvector. Define the space $V^\perp$ spanned by vectors that are orthogonal to $\mathbf{x}_1$, $V^\perp = \left\{\mathbf{v} \mid \mathbf{x}_1^\dagger \mathbf{v} = 0\right\}$. Then $\mathbf{A}\mathbf{v} \in V^\perp$, see Sec. 2.2.1, so within this space it is possible to represent $\mathbf{A}$ by a $(N-1)\times(N-1)$ matrix. QR iteration can then proceed with this smaller matrix. As more and more eigenvalues converge, the matrix becomes smaller and smaller, which makes calculating the remaining eigenvalues faster. This technique is called *deflation*. The routines found in the standard numerical linear algebra libraries incorporate techniques for deflation.

2.6.3 QR factorization by Householder transformations*

Modified Gram-Schmidt can be used as QR factorization algorithm, as explained in the lecture notes on linear problems. In practice, other techniques are more popular at present. For completeness, I will explain in this section one of these techniques. It can be done for any square matrix, but I will show it for a tridiagonal matrix only

$$\mathbf{T} = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & 0 \\ 0 & 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & t_{NN} \end{pmatrix}. \tag{2.54}$$

Multiply this with a unitary matrix $\mathbf{P}^{(1)}$, such that

$$\mathbf{T}^{(1)} = \mathbf{P}^{(1)}\mathbf{T} = \begin{pmatrix} t'_{11} & t'_{12} & t'_{13} & 0 & 0 \\ 0 & t'_{22} & t'_{23} & 0 & 0 \\ 0 & t_{32} & t_{33} & 0 & 0 \\ 0 & 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & t_{NN} \end{pmatrix}, \tag{2.55}$$

which means we have made $t_{21} = 0$. Note that we have only changed the block t'_{ij}; $i = 1, 2$; $j = 1, 2, 3$, and we have created a nonzero element t'_{13}. Now multiply with a second unitary matrix $\mathbf{P}^{(2)}$, such that

$$\mathbf{T}^{(2)} = \mathbf{P}^{(2)}\mathbf{T}^{(1)} = \begin{pmatrix} t'_{11} & t'_{12} & t'_{13} & 0 & 0 \\ 0 & t''_{22} & t''_{23} & t''_{24} & 0 \\ 0 & 0 & t''_{33} & t''_{34} & 0 \\ 0 & 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & t_{NN} \end{pmatrix}, \tag{2.56}$$

i.e., we have made $t''_{32} = 0$. We have only changed the block $t''_{ij};\ i = 2,3;\ j = 2,3,4$, and created a nonzero element t''_{24}. After $N-1$ of such steps, we have created a matrix

$$\mathbf{T}^{(N-2)} = \mathbf{P}^{(N-2)}\mathbf{P}^{(N-3)}...\mathbf{P}^{(1)}\mathbf{T} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 & 0 \\ 0 & r_{22} & r_{23} & r_{24} & 0 \\ 0 & 0 & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \ddots & r_{N-1,N} \\ 0 & 0 & \cdots & 0 & r_{NN} \end{pmatrix} \equiv \mathbf{R}. \tag{2.57}$$

This is our upper triangular matrix. Note that it has a bandwidth of two, i.e. $r_{ij} = 0;\ j > i+2$, and if $j < i$.

There are several possible choices for the elementary unitary transformations $\mathbf{P}^{(i)}$. I will recycle our *Householder reflections*.[26] Eq. 2.55 requires

$$\mathbf{P}^{(1)}\mathbf{t}_1 = \left(\mathbf{I} - 2\mathbf{u}\mathbf{u}^\dagger\right)\mathbf{t}_1 = t'_{11}\mathbf{e}, \tag{2.58}$$

with $\mathbf{t}_1$ the first column of $\mathbf{T}$ and $\mathbf{e}$ the unit vector of length N with elements $e_i = \delta_{1i}$. This looks very much like the story following Eq. 2.35, so we know the outcome. According to Eq. 2.46, a proper transformation is given by

$$\mathbf{P}^{(1)} = \mathbf{I} - \frac{1}{w}\mathbf{v}\mathbf{v}^\dagger \quad \text{with} \quad \mathbf{v} = \mathbf{t}_1 \pm \mathbf{e}\,\|\mathbf{t}_1\|\,\frac{t_{11}}{|t_{11}|} \quad \text{and} \quad w = \|\mathbf{t}_1\|\left(\|\mathbf{t}_1\| \pm |t_{11}|\right), \tag{2.59}$$

where $\|\mathbf{t}_1\| = \sqrt{|t_{11}|^2 + |t_{21}|^2}$. Note the subtle difference between this expression and that of Eq. 2.46. This time we are starting at the first element of the vector, t_{11}, whereas formerly we were starting at the second element a_{21}. Our matrix $\mathbf{P}^{(1)}$ is thus $N \times N$, instead of $(N-1)\times(N-1)$, as in Eq. 2.46. Also note that $\mathbf{t}_1$ only has two nonzero elements, t_{11} and t_{21}, which means that $\mathbf{v}$ only has two nonzero elements, v_1 and v_2. This means that $\mathbf{P}^{(1)}$ is different from the identity matrix only in the upper-left 2×2 block, $P^{(1)}_{ij} = \delta_{ij} - v_i v^*_j/w;\ i = 1,2;\ j = 1,2$. For all other i,j, $P^{(1)}_{ij} = \delta_{ij}$ is just the identity matrix. The matrix multiplication $\mathbf{P}^{(1)}\mathbf{T}$, Eq. 2.55, then only affects the block $t'_{ij};\ i = 1,2;\ j = 1,2,3$. The matrix elements for all other i,j are untouched by this multiplication, $t'_{ij} = t_{ij}$.

For the next cycle we construct

$$\mathbf{P}^{(2)} = \begin{pmatrix} 1 & \mathbf{0}^\dagger \\ \mathbf{0} & \mathbf{P} \end{pmatrix}, \tag{2.60}$$

as in Eq. 2.33 and proceed to Eq. 2.46, with $\mathbf{P}$ a $(N-1)\times(N-1)$ matrix

$$\mathbf{P} = \mathbf{I} - \frac{1}{w'}\mathbf{v}'\mathbf{v}'^\dagger \quad \text{with} \quad \mathbf{v}' = \mathbf{t}'_2 + \mathbf{e}\,\|\mathbf{t}'_2\|\,\frac{t'_{22}}{|t'_{22}|} \quad \text{and} \quad w' = \|\mathbf{t}'_2\|\left(\|\mathbf{t}'_2\| + |t'_{22}|\right), \tag{2.61}$$

where $\mathbf{t}'_2$ is the first column of the matrix obtained from $\mathbf{T}^{(1)}$ by deleting the first row and the first column. It is a vector of $N-1$ elements, but only the first two elements are nonzero t'_{22} and t_{32}, so $\|\mathbf{t}'_2\| = \sqrt{|t'_{22}|^2 + |t_{32}|^2}$. This means that $\mathbf{P}^{(2)}$ is different from the identity matrix only for the elements $P^{(2)}_{ij};\ i = 2,3;\ j = 2,3$. For all other i,j, $P^{(2)}_{ij} = \delta_{ij}$ is just

[26] Another possibility is to use *Givens rotations* [13, 3].

the identity matrix. The matrix multiplication $\mathbf{P}^{(2)}\mathbf{T}^{(1)}$, Eq. 2.56, then only affects the block t''_{ij}; $i = 2, 3$; $j = 2, 3, 4$. The matrix elements for all other i, j are untouched by this multiplication.

We proceed with steps similar to Eq. 2.57. Remembering that Householder reflections are both unitary and Hermitian, $\left(\mathbf{P}^{(i)}\right)^{-1} = \left(\mathbf{P}^{(i)}\right)^{\dagger} = \mathbf{P}^{(i)}$, we write

$$\mathbf{T} = \mathbf{P}^{(1)}...\mathbf{P}^{(N-3)}\mathbf{P}^{(N-1)}\mathbf{R}, \tag{2.62}$$

where the product $\mathbf{P}^{(1)}...\mathbf{P}^{(N-3)}\mathbf{P}^{(N-2)} = \mathbf{Q}$. We never have to construct the matrix $\mathbf{Q}$ explicitly. In the next step of QR iteration, Eq. 2.52, we calculate the product

$$\mathbf{T}' = \mathbf{RQ} = \mathbf{RP}^{(1)}...\mathbf{P}^{(N-3)}\mathbf{P}^{(N-1)}. \tag{2.63}$$

2.6.3.1 Cost and efficiency*

We have shown above that, if $\mathbf{T}$ is tridiagonal, then $\mathbf{Q}$ is a product of matrices that differ from the identity matrix only in one diagonal 2×2 block, and $\mathbf{R}$ has a bandwidth of two. It is straightforward to show that the product $\mathbf{T}' = \mathbf{RQ}$, which we need for the QR iteration algorithm, is a tridiagonal matrix. According to the discussion in the previous section, the $\mathbf{RQ}$ product then scales as $O(N)$. Our explicit construction of the $\mathbf{QR}$ product (the factorization) is also an $O(N)$ algorithm. That is because each step requires a fixed number of operations, and the total number of steps is $N - 1$.

2.7 Subspace techniques for single eigenvalues*

Let's revisit the problem of finding a single eigenvalue/eigenvector of $\mathbf{A}$. Power iteration, Eq. 2.9, and its siblings inverse power iteration, Eq. 2.14, and Rayleigh quotient iteration, Eq. 2.19, do just this. They turn out not to be the most efficient algorithms in terms of the number of cycles n it takes to achieve convergence. These techniques cycle on a single vector only. They generate a new vector $\mathbf{b}_{n+1}$ from the previous vector $\mathbf{b}_n$ by applying a rule $\mathbf{b}_{n+1} = \mathbf{M}\mathbf{b}_n$, with $\mathbf{M} = \mathbf{A}$ for power iteration, $\mathbf{M} = (\mathbf{A} - s\mathbf{I})^{-1}$ for shifted inverse power iteration, and $\mathbf{M} = (\mathbf{A} - R(\mathbf{b}_n)\mathbf{I})^{-1}$ for Rayleigh quotient iteration. Once the new vector is calculated, the old one can be discarded. In this respect, these algorithms resemble the *one-step memory* iterative techniques we encountered in solving (non-)linear problems.

As for the latter type of problems, one can think of techniques where one keeps a memory of vectors obtained in all previous iterations. These vectors span a linear space, called the *subspace*. In each iteration a new vector is constructed that is linearly independent of all previous vectors. Adding the new vector to the subspace, its dimension is increased by one. Solving the eigenvalue problem within this subspace should give a better approximation to the eigenvalue than just the Rayleigh quotient of a single vector. While iterating, the subspace is continuously expanded, and we should obtain a better and better approximation. This scheme is successful if convergence is reached while the number of iterations n is still much smaller than the dimension N of the matrix.

One critical point is to construct suitable vectors to expand the subspace. In view of the power iteration method, it seems logical to consider spaces spanned by the vectors $\mathbf{v}, \mathbf{A}\mathbf{v}, \mathbf{A}^2\mathbf{v}, ..., \mathbf{A}^{n-1}\mathbf{v}$, with $\mathbf{v}$ some initial vector. In the next iteration the next power $\mathbf{A}^n\mathbf{v}$ is then added. These spaces are called *Krylov spaces*. They are natural subspaces for

eigenvalue searches. Given the Krylov space, one still has to solve the eigenvalue problem in this subspace, of course. The *Lanczos method* is a favorite of the mathematicians. Physicists and chemists are only lukewarm about Lanczos, so I refer you to the mathematical literature [3, 13].

Alternatively, one can build a subspace by focussing on the *residual vector* (also called *residual* or *residue*)

$$\mathbf{r} = [\mathbf{A} - R\mathbf{I}]\,\mathbf{v} \quad \text{with } R = \frac{\mathbf{v}^\dagger \mathbf{A}\mathbf{v}}{\mathbf{v}^\dagger \mathbf{v}} \text{ the Rayleigh quotient,} \tag{2.64}$$

and $\mathbf{v}$ some initial vector. If $\mathbf{r} = \mathbf{0}$, then R is an eigenvalue of $\mathbf{A}$, with $\mathbf{v}$ the corresponding eigenvector.

Proof: If $\mathbf{r} = \mathbf{0}$, then $\mathbf{A}\mathbf{v} = R\mathbf{v}$. The number R is then obviously an eigenvalue of $\mathbf{A}$, and $\mathbf{v}$ is an eigenvector. The reverse is also true, if $\mathbf{x}$ is an eigenvector with eigenvalue λ, then $R(\mathbf{x}) = \mathbf{x}^\dagger\mathbf{A}\mathbf{x}/\mathbf{x}^\dagger\mathbf{x} = \lambda\mathbf{x}^\dagger\mathbf{x}/\mathbf{x}^\dagger\mathbf{x} = \lambda$, and $\mathbf{r} = \mathbf{A}\mathbf{x} - R\mathbf{x} = 0$.

The residual vector gives a measure of how far the vector $\mathbf{v}$ is removed from being an eigenvector of $\mathbf{A}$. In some way the residual vector measures the error in $\mathbf{v}$, and gives the direction in which it can be improved. One idea to use the residual vector for expanding the subspace stems from Davidson [14], although some of the idea seems to go back to Jacobi. The technique has been generalized by Sleijpen and van der Vorst, who called the algorithm *generalized Jacobi-Davidson* [15]. Various flavors of *Davidson iteration* for finding the lowest eigenvalue/eigenvector of a very large matrix, are popular in physics and chemistry communities.

One can take the concept of the residual representing an error vector more literally, and apply *Pulay iteration* or *DIIS*, which we encountered in solving (non-)linear problems, to find an eigenvalue/eigenvector [9, 17]. It uses subspaces that are exclusively build from residual vectors. DIIS is currently also a very popular algorithm.

2.7.0.1 Ritz values and the variational principle

A single vector $\mathbf{b}_1$ spans a subspace of dimension one, so within this space all $\mathbf{x} = a_1\mathbf{b}_1$. Projected onto this subspace, the eigenvalue equation $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ becomes $\mathbf{b}_1^\dagger\mathbf{A}\mathbf{x} = \lambda\mathbf{b}_1^\dagger\mathbf{x}$, which gives $\lambda = \mathbf{b}_1^\dagger\mathbf{A}\mathbf{b}_1/\mathbf{b}_1^\dagger\mathbf{b}_1 = R(\mathbf{b}_1)$. In other words, the Rayleigh quotient is the best approximation to the eigenvalue in a subspace of dimension one.

If we have a subspace spanned by two vectors $\mathbf{b}_1, \mathbf{b}_2$, then with $\mathbf{x} = a_1\mathbf{b}_1 + a_2\mathbf{b}_2$, we cannot do worse than in a one-dimensional subspace. Assuming, without loss of generality, orthonormal vectors, $\mathbf{b}_i \cdot \mathbf{b}_j = \delta_{ij}$, then the eigenvalue equation $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ projected on this two-dimensional subspace becomes

$$\mathbf{b}_i^\dagger\mathbf{A}\mathbf{x} = \lambda\mathbf{b}_i^\dagger\mathbf{x} \Longleftrightarrow \mathbf{b}_i^\dagger\mathbf{A}\sum_{j=1}^{2} a_j\mathbf{b}_j = \lambda\mathbf{b}_i^\dagger\sum_{j=1}^{2} a_j\mathbf{b}_j \Longleftrightarrow \sum_{j=1}^{2}\mathbf{b}_i^\dagger\mathbf{A}\mathbf{b}_j a_j = \lambda\sum_{j=1}^{2}\mathbf{b}_i^\dagger\mathbf{b}_j a_j = \lambda\delta_{ij}a_j,$$

or, in matrix form

$$\begin{pmatrix} \mathbf{b}_1^\dagger\mathbf{A}\mathbf{b}_1 & \mathbf{b}_1^\dagger\mathbf{A}\mathbf{b}_2 \\ \mathbf{b}_2^\dagger\mathbf{A}\mathbf{b}_1 & \mathbf{b}_2^\dagger\mathbf{A}\mathbf{b}_2 \end{pmatrix}\begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \lambda_{1,2}\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, \tag{2.65}$$

where I have used a subscript to distinguish the two eigenvalues.

Another way of approaching the problem is to write the Rayleigh quotient for an arbitrary vector in this subspace

$$R(\mathbf{x}) = R(a_1\mathbf{b}_1 + a_2\mathbf{b}_2) = \frac{a_1^* a_1 \mathbf{b}_1^\dagger \mathbf{A}\mathbf{b}_1 + a_1^* a_2 \mathbf{b}_1^\dagger \mathbf{A}\mathbf{b}_2 + a_2^* a_1 \mathbf{b}_2^\dagger \mathbf{A}\mathbf{b}_1 + a_2^* a_2 \mathbf{b}_2^\dagger \mathbf{A}\mathbf{b}_2}{a_1^* a_1 + a_1^* a_2 + a_2^* a_1 + a_2^* a_2}. \tag{2.66}$$

The stationary points of the Rayleigh quotient as a function of a_1^* and a_2^* can be found by setting $\partial R/\partial a_1^* = 0$ and $\partial R/\partial a_2^* = 0$. This leads to the same equations as Eq. 2.65, with λ_1 and λ_2 the values of R at the stationary points.[27] The easiest way to prove this is to write

$$a_1^* a_1 \mathbf{b}_1^\dagger \mathbf{A}\mathbf{b}_1 + a_1^* a_2 \mathbf{b}_1^\dagger \mathbf{A}\mathbf{b}_2 + a_2^* a_1 \mathbf{b}_2^\dagger \mathbf{A}\mathbf{b}_1 + a_2^* a_2 \mathbf{b}_2^\dagger \mathbf{A}\mathbf{b}_2 = R(\mathbf{x})\left(a_1^* a_1 + a_1^* a_2 + a_2^* a_1 + a_2^* a_2\right),$$

and take $\partial/\partial a_1^*$ and $\partial/\partial a_2^*$ of the left- and right-hand sides. The values of the Rayleigh quotient at its stationary points are called the *Ritz values*. They are thus equal to the eigenvalues $\lambda_{1,2}$ of the problem projected on the subspace, Eq, 2.65. This remains true if we extend the subspace, writing $\mathbf{x} = a_1\mathbf{b}_1 + a_2\mathbf{b}_2 + ... + a_n\mathbf{b}_n$; the Ritz values are equal to the eigenvalues $\lambda_1^{(n)}, \lambda_2^{(n)}, ..., \lambda_n^{(n)}$ of the projected $n \times n$ eigenvalue problem. The proof is a straightforward extension of the two-dimensional case.

From Eqs. 2.65. and 2.66, it is fairly easy to show that the lowest eigenvalue λ_1 is a *minimum* of the Rayleigh quotient (and the highest eigenvalue λ_2 is a *maximum*). This also remains true if we enlarge the subspace. The lowest eigenvalue $\lambda_1^{(n)}$ within the n-dimensional subspace is the *minimum* of the Rayleigh quotient $R(\mathbf{x})$ (and the highest eigenvalue $\lambda_n^{(n)}$ is the *maximum*). Enlarging the subspace by adding an extra (linear independent) vector $\mathbf{b}_{n+1}$, the minimum cannot go up, so

$$\lambda_1^{(N)} \leq \lambda_1^{(n+1)} \leq \lambda_1^{(n)}, \tag{2.67}$$

where $\lambda_1^{(N)}$ is the exact lowest eigenvalue of the matrix $\mathbf{A}$. This is essentially an expression of the *variational principle* as you know it from quantum mechanics. For linear spaces it is called the *Rayleigh-Ritz variational principle*. The trick is to construct a subspace such, that $\lambda_1^{(n)} \approx \lambda_1^{(N)}$ for n as small as possible.

2.7.1 Davidson and GJD

Start from a normalized vector $\mathbf{v}$, and its Rayleigh quotient $R = \mathbf{v}^\dagger \mathbf{A}\mathbf{v}$. Write the exact eigenvector as $\mathbf{x} = \mathbf{v} + \mathbf{\Delta}$, and the exact eigenvalue as $\lambda = R + \alpha$. The eigenvalue equation

$$[\mathbf{A} - \lambda\mathbf{I}]\,\mathbf{x} = 0 \quad \text{becomes} \quad [\mathbf{A} - (R+\alpha)\,\mathbf{I}]\,(\mathbf{v} + \mathbf{\Delta}) = 0. \tag{2.68}$$

This can be reorganized as

$$[\mathbf{A} - (R+\alpha)\,\mathbf{I}]\,\mathbf{\Delta} = -\mathbf{r} + \alpha\mathbf{v}, \quad \text{with } \mathbf{r} = [\mathbf{A} - R\mathbf{I}]\,\mathbf{v}, \tag{2.69}$$

the residual vector. This does not seem to bring us further, as this equation is as difficult to solve as the original eigenvalue equation. However, it inspires us to find and approximate

[27] One can also view the Rayleigh quotient as a function of a_1 and a_2, and find the stationary points by setting $\partial R/\partial a_1 = 0$ and $\partial R/\partial a_2 = 0$. This just leads to the complex conjugate of Eq. 2.65, giving the same stationary points.

$\boldsymbol{\Delta}'$. Assume that $\alpha \approx 0$, i.e., the Rayleigh quotient is already a decent approximation to the eigenvalue. Then define $\boldsymbol{\Delta}'$ by solving

$$[\mathbf{A} - R\mathbf{I}]\,\boldsymbol{\Delta}' = -\mathbf{r}, \tag{2.70}$$

as a linear problem. In this form, this is unfortunately not too helpful either. The solution is $\boldsymbol{\Delta}' = -\mathbf{v}$, which is a pretty useless, as it means $\mathbf{v} + \boldsymbol{\Delta}' = 0$, and we are not closer to the eigenvector $\mathbf{x}$.

As we are aiming at using $\boldsymbol{\Delta}'$ to expand our subspace, it is a good idea to solve Eq. 2.70 under the constraint that the correction vector is orthogonal to $\mathbf{v}$, i.e., use

$$\boldsymbol{\Delta}'' = \boldsymbol{\Delta}' - \mathbf{v}\left(\mathbf{v}^\dagger \boldsymbol{\Delta}'\right) = \left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)\boldsymbol{\Delta}', \tag{2.71}$$

instead of $\boldsymbol{\Delta}'$. Eq. 2.70 then becomes

$$[\mathbf{A} - R\mathbf{I}]\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)\boldsymbol{\Delta}' = -\mathbf{r}. \tag{2.72}$$

We normalize the solution $\boldsymbol{\Delta}'/\left\|\boldsymbol{\Delta}'\right\| = \mathbf{q}$ and form the $N \times 2$ matrix $\mathbf{Q} = (\mathbf{v}\ \mathbf{q})$. Calculating the 2×2 Rayleigh quotient matrix $\mathbf{Y} = \mathbf{Q}^\dagger\mathbf{A}\mathbf{Q}$, one expects the lowest eigenvalue of this matrix ρ_1 (the Ritz value) to be closer to λ than R (see the previous section, Eq. 2.67).

One technical problem is that, whereas $\mathbf{A}$ is a Hermitian matrix, $\mathbf{A}\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)$ is not.

Proof: $\left[\mathbf{A}\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)\right]^\dagger = \left(\mathbf{I} - \left(\mathbf{v}^\dagger\right)^\dagger \mathbf{v}^\dagger\right)\mathbf{A}^\dagger = \left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)\mathbf{A} \neq \mathbf{A}\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)$ for general $\mathbf{A}$ and $\mathbf{v}$.

If possible, we would like to stick to Hermitian matrices. The product $\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)\mathbf{A}\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)$ is a Hermitian matrix. Multiplying Eq. 2.72 from the left with $\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)$ turns it into the Hermitian problem

$$\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)[\mathbf{A} - R\mathbf{I}]\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)\boldsymbol{\Delta}' = -\mathbf{r}. \tag{2.73}$$

Proof: $\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)\mathbf{r} = \mathbf{r} - \mathbf{v}\mathbf{v}^\dagger\mathbf{r} = \mathbf{r}$, because $\mathbf{v}^\dagger\mathbf{r} = \mathbf{v}^\dagger\left[\mathbf{A} - R\mathbf{I}\right]\mathbf{v} = \mathbf{v}^\dagger\mathbf{A}\mathbf{v} - R = 0$.

The matrix $\mathbf{P} = \mathbf{I} - \mathbf{v}\mathbf{v}^\dagger$ is a *projection matrix*; it projects any vector $\mathbf{y}$ on its component orthogonal to $\mathbf{v}$, $\mathbf{P}\mathbf{y} \perp \mathbf{v}$, see Eq. 2.71. The linear problem of Eq. 2.73 is formulated in a space that is entirely orthogonal to $\mathbf{v}$.

It is unlikely that a linear combination of $\mathbf{v}$ and $\mathbf{q}$ will lead you immediately to the exact eigenvalue and eigenvector. Therefore, in the next iteration one constructs a vector $\mathbf{q}_2$ in the space orthogonal to $\mathbf{v}$ *and* $\mathbf{q}$, and so on. This algorithm is guaranteed to converge in N iterations, as N is the dimension of the full space, and the Ritz value ρ_1 in an N-dimensional space is per definition the eigenvalue λ_1. Hopefully, however, it converges in $n \ll N$ iterations.

Matrix multiplications, as in $\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)\mathbf{A}$, should be avoided, because they are $O(N^3)$, and therefore costly. They can be avoided by writing

$$\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)[\mathbf{A} - R\mathbf{I}]\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right) = \mathbf{A} - R\mathbf{I} - (\mathbf{A}\mathbf{v})\,\mathbf{v}^\dagger - \mathbf{v}\,(\mathbf{A}\mathbf{v})^\dagger. \tag{2.74}$$

Proof: $\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)\mathbf{A}\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right) = \mathbf{A} - \mathbf{v}\mathbf{v}^\dagger\mathbf{A} - \mathbf{A}\mathbf{v}\mathbf{v}^\dagger + \mathbf{v}\mathbf{v}^\dagger\mathbf{A}\mathbf{v}\mathbf{v}^\dagger = \mathbf{A} - \mathbf{v}\,(\mathbf{A}\mathbf{v})^\dagger - (\mathbf{A}\mathbf{v})\,\mathbf{v}^\dagger + R\mathbf{v}\mathbf{v}^\dagger$. Furthermore, as $\mathbf{I}\mathbf{B} = \mathbf{B}\mathbf{I}$, i.e., the identity matrix commutes with any matrix $\mathbf{B}$, and $\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)^2 =$

$\mathbf{I} - 2\mathbf{v}\mathbf{v}^\dagger + \mathbf{v}\mathbf{v}^\dagger\mathbf{v}\mathbf{v}^\dagger = \mathbf{I} - \mathbf{v}\mathbf{v}^\dagger$, as $\mathbf{v}^\dagger\mathbf{v} = 1$, we have $\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right) R\mathbf{I}\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right) = R\mathbf{I}\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right)^2 = R\mathbf{I}\left(\mathbf{I} - \mathbf{v}\mathbf{v}^\dagger\right) = R\mathbf{I} - R\mathbf{v}\mathbf{v}^\dagger$.

This means the matrix-vector multiplication $\mathbf{Av}$ suffices to construct the full matrix, which is $O(N^2)$.

2.7.1.1 Generalized Jacobi-Davidson iteration (GJD)

The ideas discussed above lead to the *generalized Jacobi-Davidson iteration* algorithm (GJD) for finding the lowest eigenvalue, as formulated by Sleijpen and van der Vorst [13, 15].

$$\mathbf{q}_0;\ \rho_1 = \mathbf{q}_0^\dagger\mathbf{A}\mathbf{q}_0;\ \mathbf{r}_1 = [\mathbf{A} - \rho_1\mathbf{I}]\,\mathbf{q}_0;\ \mathbf{Q}_0 = \mathbf{q}_0;\ \mathbf{v}_1 = \mathbf{q}_0; \tag{2.75}$$

$$\text{do } n = 1, \ldots \text{ solve } \left(\mathbf{A} - \rho_n\mathbf{I} - (\mathbf{A}\mathbf{v}_n)\,\mathbf{v}_n^\dagger - \mathbf{v}_n\,(\mathbf{A}\mathbf{v}_n)^\dagger\right)\boldsymbol{\Delta}_n = -\mathbf{r}_n; \tag{2.76}$$

$$\text{do } m = 1, \ldots, n-1\ \ \boldsymbol{\Delta}_n = \boldsymbol{\Delta}_n - \mathbf{q}_m\left(\mathbf{q}_m^\dagger\boldsymbol{\Delta}_n\right)\ \text{ end};\ \mathbf{q}_n = \frac{\boldsymbol{\Delta}_n}{\|\boldsymbol{\Delta}_n\|}; \tag{2.77}$$

$$\mathbf{Q}_n = (\mathbf{Q}_{n-1}\ \mathbf{q}_n);\ \mathbf{Y}_n = \mathbf{Q}_n^\dagger\mathbf{A}\mathbf{Q}_n;\ \text{diagonalize } \mathbf{Y}_n = \mathbf{W}^\dagger\boldsymbol{\Lambda}'\mathbf{W}; \tag{2.78}$$

$$\rho_{n+1} = \Lambda_1';\ \mathbf{v}_{n+1} = \mathbf{Q}_n\mathbf{w}_1;\ \mathbf{r}_{n+1} = [\mathbf{A} - \rho_{n+1}\mathbf{I}]\,\mathbf{v}_{n+1}\ \text{ until converged.} \tag{2.79}$$

In Eq. 2.76 the linear problem is solved to find a correction vector $\boldsymbol{\Delta}$, which is perpendicular to $\mathbf{v}$. This vector is then orthogonalized to the subspace spanned by all previously constructed vectors $\mathbf{q}_k;\ k = 1, \ldots, n-1$ in Eq. 2.77 and normalized. The result $\mathbf{q}_n$ is used to expand the subspace in Eq. 2.78. Here $\mathbf{Q}_n = (\mathbf{Q}_{n-1}\ \mathbf{q}_n)$ means appending the $N \times (n-1)$ matrix $\mathbf{Q}_{n-1}$ by the column vector $\mathbf{q}_n$ to make the $N \times n$ matrix $\mathbf{Q}_n$. The Rayleigh matrix $\mathbf{Y}_n$ is then calculated, and subsequently diagonalized. Eq. 2.79 then selects the lowest Ritz value and constructs the corresponding Ritz vector and the residual vector for the next iteration. The "until converged" criterion is something like $|\rho_{n+1} - \rho_n| < \epsilon$, with ϵ a tolerance chosen by the user, where the Ritz value ρ_{n+1} is the current best guess for the eigenvalue, with the Ritz vector $\mathbf{v}_{n+1}$ the corresponding eigenvector.

Some technical comments are in order here.

- In the calculation of the of the Rayleigh matrix $\mathbf{Y}_n$ one can of course make use of the matrix $\mathbf{Y}_{n-1}$. The Hermitian $n \times n$ matrix $\mathbf{Y}_n$ has the form

$$\mathbf{Y}_n = \begin{pmatrix} \mathbf{Y}_{n-1} & \mathbf{Q}_{n-1}^\dagger\mathbf{A}\mathbf{q}_n \\ \left(\mathbf{Q}_{n-1}^\dagger\mathbf{A}\mathbf{q}_n\right)^\dagger & \mathbf{q}_n^\dagger\mathbf{A}\mathbf{q}_n \end{pmatrix}, \tag{2.80}$$

  and the $(n-1)\times(n-1)$ block $\mathbf{Y}_{n-1}$ does not have to be recalculated. The other terms require the calculation of one matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{q}_n$, which is $O(N^2)$, and n inner products $\mathbf{q}_k^\dagger\mathbf{y};\ k = 1, \ldots, N$, each of which are $O(N)$.

- Although we only explicitly need the lowest eigenvalue of $\mathbf{Y}_n$, we may as well completely diagonalize this matrix by a unitary transformation (using the Householder transformation plus QR iteration, for instance). As the dimension n of this matrix is small ($n \ll N$, provided Davidson iteration converges sufficiently fast), this diagonalization is $O(n^3)$, so it hardly adds any overhead.

- The tricky point in the GJD algorithm is solving the linear problem of Eq. 2.76. Note that LU decomposition cannot be done outside the loop, as the matrix of Eq. 2.74 changes in each iteration. $\mathbf{A}$ is a large matrix, otherwise we would not be using GDJ in the first place. As the computational cost of factorization scales as $O(N^3)$, it quickly becomes too expensive for increasing N. One must use an efficient iterative technique for solving the linear problem; GMRES is popular, for instance, see Sec. 1.4.5. That adds an extra do-loop for the linear problem, inside the do-loop over n. The cost of that do-loop scales as $O(pN^2)$, with p the number of GMRES iterations. In practice, one does not need an extremely accurate solution of the linear problem. An approximate solution also works. This means that p can be kept small.
- The computational costs of the GJD algorithm scale as $O(npN^2)$. For a sparse matrix, such as a tridiagonal matrix, all matrix-vector multiplications are $O(N)$, and in that case the GDJ algorithm scales as $O(npN)$.

2.7.1.2 Davidson iteration

Davidson's original solution to the linear problem of Eq. 2.76 is a very drastic one [14]. He stresses that we don't need an extremely accurate solution to the linear problem, as we only search for a suitable vector $\mathbf{\Delta}$ to expand our subspace. So he approximates $\mathbf{A} - (\mathbf{Av})\,\mathbf{v}^\dagger - \mathbf{v}\,(\mathbf{Av})^\dagger$ by the diagonal part $\mathbf{D}$ of $\mathbf{A}$, with matrix elements $D_{ij} = \delta_{ij}A_{ii}$. The solution to the linear problem then is

$$\mathbf{\Delta}_n \approx -\left(\mathbf{D} - \rho_n \mathbf{I}\right)^{-1} \mathbf{r}_n. \tag{2.81}$$

This is easy to calculate, as the inverse of a diagonal matrix is trivial. Using this instead of Eq. 2.76, is the original version of the algorithm sketched above. It is called *Davidson iteration.* If it works, it is a brilliant simplification. As you might expect, it works if the matrix $\mathbf{A}$ is diagonally dominant, but if it is not, then this simplification can fail.

Mathematicians are typically interested in techniques that can solve any eigenvalue problem. Physicists are usually only interested in solving their own particular eigenvalue problem, and care little about other problems. With some inside knowledge of what the matrix $\mathbf{A}$ looks like in the particular case, it may be possible to come up with an approximation $\mathbf{M} \approx \mathbf{A}$, which is easy to invert. That should simplify the solution of the linear problem of Eq. 2.76.

$$\mathbf{\Delta}_n \approx -\left(\mathbf{M} - \rho_n \mathbf{I}\right)^{-1} \mathbf{r}_n. \tag{2.82}$$

This is the way Davidson iteration is used in the popular electronic structure program VASP, for instance [9, 16]. Its success or failure hangs on the ability to come up with a decent matrix $\mathbf{M}$. The technique is called using an *approximate inverse.* Some people also call it *preconditioning,* analogous to the use of such a matrix in solving linear problems iteratively, see the lecture notes on (non-)linear problems. [28]

2.7.2 Pulay iteration or DIIS

2.7.2.1 The eigenvalue problem as a non-linear problem

Methods to solve non-linear problems by fixed-point iteration, as discussed in the lecture notes on (non-)linear problems, can, in principle, also be used to find eigenvalues and eigen-

[28] Using the phrase "preconditioning" in the context of Davidson iteration might be wrong (ask a mathematician), but it is useless to fight common folklore anyway.

vectors. Define the non-linear problem

$$\mathbf{A}\mathbf{x} = R(\mathbf{x})\mathbf{x}, \quad \text{with} \quad R(\mathbf{x}) = \frac{\mathbf{x}^\dagger \mathbf{A}\mathbf{x}}{\mathbf{x}^\dagger \mathbf{x}}, \tag{2.83}$$

the Rayleigh quotient. This is simply a restatement of the eigenvalue problem, as argued below Eq. 2.64.

One can derive from Eq. 2.83 the fixed-point iteration rule

$$\mathbf{A}\mathbf{x}_{n+1} = R(\mathbf{x}_n)\mathbf{x}_n. \tag{2.84}$$

Using LU factorization to solve this linear problem, this then very much looks like inverse power iteration, Eq. 2.14 (with shift $s = 0$, and $\mathbf{x}$ instead of $\mathbf{b}$). The main difference is that the vector $\mathbf{x}_n$ in Eq. 2.84 is not normalized, but multiplied by $R(\mathbf{x}_n)$ instead.

It becomes more interesting if LU factorization is too expensive (as it is for large matrices), and we have to use an iterative method, which is based upon splitting the matrix

$$\mathbf{A} = \mathbf{M} + (\mathbf{A} - \mathbf{M}), \tag{2.85}$$

where $\mathbf{M}$ should be a matrix that is easily invertible. One can rewrite Eq. 2.83 as $\mathbf{M}\mathbf{x} = -(\mathbf{A} - \mathbf{M})\mathbf{x} + R(\mathbf{x})\mathbf{x}$ or $\mathbf{x} = \mathbf{x} - \mathbf{M}^{-1}(\mathbf{A}\mathbf{x} - R(\mathbf{x})\mathbf{x})$, which leads to the fixed-point iteration rule

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{M}^{-1}\mathbf{r}_n, \quad \text{with } \mathbf{r}_n = \mathbf{A}\mathbf{x}_n - R(\mathbf{x}_n)\mathbf{x}_n, \tag{2.86}$$

the residual vector. The matrix $\mathbf{M}^{-1}$ is used to scale the residual vector, and is often called the *preconditioner* in this context. For instance, in the Jacobi method $\mathbf{M} = \mathbf{D}$, the diagonal part of matrix $\mathbf{A}$, whereas in the Gauss-Seidel method $\mathbf{M} = \mathbf{U} + \mathbf{D}$, the upper triangular part plus the diagonal part of matrix $\mathbf{A}$. In principle one can use any matrix $\mathbf{M}$ that accelerates solving the problem. As argued above, inside knowledge on the properties of $\mathbf{A}$, resulting from physical insight in the particular problem at hand, sometimes helps to come up with a good $\mathbf{M}$.

Interestingly, the splitting of Eq. 2.85 also allows for rewriting Eq. 2.83 as $\mathbf{M}\mathbf{x} - R(\mathbf{x})\mathbf{x} = -(\mathbf{A} - \mathbf{M})\mathbf{x}$, or $\mathbf{x} = -(\mathbf{M} - R(\mathbf{x})\mathbf{I})^{-1}(\mathbf{A} - \mathbf{M})\mathbf{x}$, or $\mathbf{x} = \mathbf{x} - (\mathbf{M} - R(\mathbf{x})\mathbf{I})^{-1}(\mathbf{A}\mathbf{x} - R(\mathbf{x})\mathbf{x})$, which gives the fixed-point iteration rule

$$\mathbf{x}_{n+1} = \mathbf{x}_n - (\mathbf{M} - R(\mathbf{x}_n)\mathbf{I})^{-1}\mathbf{r}_n, \quad \text{with } \mathbf{r}_n = \mathbf{A}\mathbf{x}_n - R(\mathbf{x}_n)\mathbf{x}_n. \tag{2.87}$$

Besides Eqs. 2.86 and 2.87, there are probably many more ways to set up an iteration rule from Eqs. 2.83 and 2.85, but these are the ones I know of that are used frequently. Using the iteration rule of Eqs. 2.86 as an example, and written in the notation of Eq. 2.14, we have the algorithm

$$\begin{aligned} &\mathbf{b}_0 = \frac{\mathbf{b}}{\|\mathbf{b}\|};\ R_0 = \mathbf{b}_0^\dagger \mathbf{A}\mathbf{b}_0; \\ &\text{do } n = 1, \ldots \quad \mathbf{r}_{n-1} = \mathbf{A}\mathbf{b}_{n-1} - R_{n-1}\mathbf{b}_{n-1};\ \mathbf{b}_n = \mathbf{b}_{n-1} - \mathbf{M}^{-1}\mathbf{r}_{n-1}; \\ &\qquad\qquad R_n = \mathbf{b}_n^\dagger \mathbf{A}\mathbf{b}_n; \quad \text{until converged.} \end{aligned} \tag{2.88}$$

To apply the rule of Eq. 2.87, one has to replace $\mathbf{M}^{-1}$ in Eq. 2.88 by $(\mathbf{M} - R_{n-1}\mathbf{I})^{-1}$. These algorithms are variations on the power iteration theme. In the language of (non-)linear equations they are *one-step memory* techniques.

It is interesting is to apply the rules of Eqs. 2.86 and 2.87 in the context of subspace iteration. Writing Eq. 2.87 as $\mathbf{x}_{n+1} + \mathbf{\Delta}$, then $\mathbf{\Delta}$ is a correction vector as in Davidson iteration, Eq. 2.82. Instead of throwing these vectors away as soon as they have been used once, as you would do normally in fixed-point iteration, one can keep these vectors to build up a subspace.

2.7.2.2 DIIS

We can keep a memory of the vectors of past iterations, and use Pulay iteration. Rewrite Eq. 2.86 as

$$\Delta\mathbf{x}_n = \mathbf{x}_{n+1} - \mathbf{x}_n = -\mathbf{M}^{-1}\mathbf{r}_n(\mathbf{x}_n), \tag{2.89}$$

build a subspace spanned by $\Delta\mathbf{x}_m;\ m = 1, ..., n$, and

$$\text{minimize} \quad \|\mathbf{\Delta}\|^2, \ \text{where } \mathbf{\Delta} = \sum_{m=1}^{n} c_m \Delta\mathbf{x}_m \tag{2.90}$$

under the constraint

$$\sum_{m=1}^{n} c_m = 1. \tag{2.91}$$

This is *Pulay iteration* or DIIS (*direct inversion in the iterative subspace*) [9, 17]. For a more detailed description of the Pulay algorithm, see the lecture notes on (non-)linear problems.

Here, I just copy the algorithm from the lecture notes on (non)-linear problems.

$$\mathbf{x}_1 = \mathbf{c};\ \mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - R(\mathbf{x}_1)\mathbf{x}_1;\ \mathbf{X}_0 = \text{nil}; \tag{2.92}$$

$$\text{do } n = 1, \ldots\ ;\ \Delta\mathbf{x}_n = -\mathbf{M}^{-1}\mathbf{r}_n(\mathbf{x}_n);\ \mathbf{X}_n = (\mathbf{X}_{n-1}\ \Delta\mathbf{x}_n)\,; \tag{2.93}$$

$$\mathbf{C}_n = \mathbf{X}_n^{\dagger}\mathbf{X}_n;\ \mathbf{x}_{n+1} = \text{Pulay minimize } \mathbf{C}_n;\ \mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - R(\mathbf{x}_{n+1})\mathbf{x}_{n+1}; \tag{2.94}$$

$$\text{until converged.} \tag{2.95}$$

The convergence criterion can be something like $\|\mathbf{r}_{n+1}\| < \epsilon$, or $\max_{i=1,\ldots,N} |(r_{n+1})_i| < \epsilon$, with ϵ a tolerance set by the user. Initialization is done in Eq. 2.92, and guessing the next correction vector $\Delta\mathbf{x}_n$ is done in Eq. 2.93, by a step in the form of Eq. 2.89. You can use your own favorite $\mathbf{M}$ in this step. The simplest approach would be Jacobi, where $\mathbf{M} = \mathbf{D}$. Next the subspace is extended, constructing the $N \times n$ matrix $\mathbf{X}_n$ by appending the vector $\Delta\mathbf{x}_n$ (an $N \times 1$ matrix) as a column to the $N \times (n-1)$ matrix $\mathbf{X}_{n-1}$. The $n \times n$ Pulay matrix $\mathbf{C}_n$ is constructed in Eq. 2.94, which is subsequently used in solving the constrained minimization problem to produce the optimal vector $\mathbf{x}_{n+1}$, see the lecture notes on (non-)linear problems. Finally, the residual vector $\mathbf{r}_{n+1}$ is calculated, and convergence is checked in Eq. 2.95.

Different flavors of Pulay iteration can be used. For instance, instead of Eq. 2.93, one can use

$$\Delta\mathbf{x}_n = -\left[\mathbf{M} - R\left(\mathbf{x}_n\right)\mathbf{I}\right]^{-1}\mathbf{r}_n(\mathbf{x}_n), \tag{2.96}$$

which is based upon the iteration rule of Eq. 2.87. This step then becomes identical to the corresponding step in Davidson iteration, see Eq. 2.82.

Pulay iteration mostly works fine, but one can get into trouble, for instance, if the matrix $\mathbf{C}_n$ becomes ill-conditioned. As discussed in the lecture notes on (non-)linear problems, one can try *orthogonalized Pulay iteration* in that case.

2.7.2.3 C^2-DIIS and RMM-DIIS

Over the years several flavors of Pulay iteration or DIIS have been developed, in order to improve its convergence properties. In this section I mention one that is popular among people that use DIIS in the context of solving eigenvalue problems.

Instead of the constraint typical of Pulay, Eq. 2.91, one can use a different constraint [18].

$$\sum_{m=1}^{n} c_m^2 = 1. \tag{2.97}$$

The problem then becomes that of minimizing Eq. 2.90 with the constraint of 2.97. Mathematically, one minimizes the function

$$F(c_1, ..., c_n, \lambda) = \sum_{m=1}^{n}\sum_{k=1}^{n} c_m c_k \Delta\mathbf{x}_m^\dagger \Delta\mathbf{x}_k - \lambda\left(\sum_{m=1}^{n} c_m^2 - 1\right), \tag{2.98}$$

where the double sum comes from $\|\mathbf{\Delta}\|^2$, Eq. 2.90, and the constraint of Eq. 2.97 is introduced by means of a Lagrange multiplier λ. Minimizing the function is done by solving the equations

$$\frac{\partial F}{\partial c_m} = 0;\ m = 1, ..., n, \tag{2.99}$$

which leads to the eigenvalue problem of dimension n

$$\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nn} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \lambda \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, \quad \text{where } C_{mk} = \Delta\mathbf{x}_m^\dagger \Delta\mathbf{x}_k; \tag{2.100}$$

$m = 1, ..., n;\ k = 1, ..., n.$

This is a small eigenvalue problem $\mathbf{Cc} = \lambda\mathbf{c}$, with $\mathbf{C}$ a $n \times n$ matrix, where the dimension n is the number of iterations (n should be small, otherwise we are in trouble anyway). Note that the matrix $\mathbf{C}$ is Hermitian, so this eigenvalue problem can be comfortably solved using a standard technique, such as Householder tridiagonalization plus QR iteration.

Writing $\mathbf{c}^\dagger\mathbf{Cc} = \lambda\mathbf{c}^\dagger\mathbf{c}$ gives

$$\lambda = \frac{\sum_{m=1}^{n}\sum_{k=1}^{n} c_m c_k \Delta\mathbf{x}_m^\dagger \Delta\mathbf{x}_k}{\sum_{m=1}^{n} c_m^2}. \tag{2.101}$$

Normalizing the vector $\mathbf{c}$, according to Eq. 2.97, then gives

$$\lambda = \|\mathbf{\Delta}\|^2, \tag{2.102}$$

so it will be clear that, in order to find the minimal residue, we are looking for the *lowest eigenvalue* in Eq. 2.100.[29]

[29] Eq. 2.101 also shows that all eigenvalues are positive, i.e., $\mathbf{C}$ is a positive definite matrix. One could try and find only the lowest eigenvalue, by shifted inverse power iteration, or Rayleigh quotient iteration, but as the dimension of the matrix n is small, one might as well find all the eigenvalues by the Householder + QR technique. This should produce hardly any overhead.

The DIIS constraint of Eq. 2.91 has some logic behind it, see the lecture notes on (non-)linear problems, but the constraint of Eq. 2.97 is purely *ad hoc*. The remedy is simple; define

$$c'_m = \frac{c_m}{\sum_{k=1}^{n} c_k}, \quad \text{then} \quad \sum_{m=1}^{n} c'_m = 1, \tag{2.103}$$

obeys the DIIS constraint, Eq. 2.91.

The lowest eigenvalue of 2.100 gives the minimal residue, Eq. 2.102, and the eigenvector $\mathbf{c}'$, normalized according to Eq. 2.103, is the solution to the minimization problem. This technique is called c^2-DIIS, referring to Eq. 2.97, and using the original constraint, 2.91, is then rechristened as c^1-DIIS. The users of c^2-DIIS argue that it is numerically more stable than c^1-DIIS. In the context of finding eigenvalues/eigenvectors of large matrices, Eqs. 2.92-2.95, c^2-DIIS is also called *RMM-DIIS*, where *RMM* stands for *Residual Minimization Method* [16, 17].[30]

Details: For those of you who are worried that the constraints of Eqs. 2.91 (c^1-DIIS) or 2.97 (c^2-DIIS) do not lead to the same minimization vector $\mathbf{c}$: you are right, in general they do not. However, one can show that, if both methods converge, and the residue λ, Eq. 2.102, becomes small, both methods converge to the same minimization vector.

2.7.3 Davidson versus Pulay

Both Davidson and Pulay iteration are popular algorithms, but they are used for slightly different purposes [9]. Davidson iteration typically finds the lowest eigenvalue of a matrix $\mathbf{A}$ (and the corresponding eigenvector), no matter what the starting vector is. That is because in subspace diagonalization, you select the lowest Ritz value, and that value is an approximation of the lowest eigenvalue.[31]

In contrast, Pulay iteration typically finds the eigenvector that is closest to the starting vector $\mathbf{x}_1$ (and the corresponding eigenvalue), Eq. 2.92. That is because you are minimizing the size of a (scaled) residual vector, see Eq. 2.89 and the description below that. That size is minimal for any eigenvector. In fact, it is zero for any exact eigenvector, see the discussion below Eq. 2.64. Upon minimization you typically find the local minimum that is closest to your starting point. So, if one wants to have the lowest eigenvalue using Pulay, one needs a decent guess for the corresponding eigenvector to start with.

This dependence upon the starting vector $\mathbf{x}_1$ is a disadvantage of Pulay. However, by now you know the universal rule: "elluk nadeil hep sun foordeil". Suppose I have a set of starting vectors $\mathbf{v}_k$; $k = 1, ..., M$ that are reasonable guesses for M different eigenvectors. I can then use Pulay iteration on each of these vectors individually, independent of the other vectors, to find much better approximations to the M eigenvectors, and the corresponding eigenvalues. Alternative techniques to find M eigenvectors need orthogonalization steps each cycle, see

[30] Somewhat of a pleonasm if you ask me, as the objective of DIIS already is to minimize the residual. But, again, you cannot argue with nomenclature.

[31] Unless you have the unlikely situation where the entire subspace is orthogonal to the lowest eigenvector, and stays that way during all iterations.

Sec. 2.8.[32] This is to make sure that not every vector collapses to the one belonging to the lowest eigenvalue. Pulay iteration is then simpler and cheaper in computing time. But, as "elluk foordeil hep sun nadeil", having a decent set of starting vectors $\mathbf{v}_k;\ k = 1, ..., M$ is the key factor here.

2.8 Subspace techniques for multiple eigenvalues

There are actually two different categories of algorithms that use subspace techniques to find eigenvalues/eigenvectors. In the previous section we discussed subspace techniques for finding a single eigenvalue/eigenvector. Suppose we are not interested in finding all N eigenvalues of a matrix $\mathbf{A}$, but we still like to have more than just one. This actually is quite a common situation. For instance, finite difference approaches for solving (partial) differential equations lead to matrices whose dimension N is determined by the number of grid points used, which is easily $O(10^3\text{-}10^6)$. To analyze the physics one commonly only needs $O(10^1\text{-}10^2)$ of its lowest eigenvalues.

I will discuss two techniques to find multiple eigenvalues. Defining a subspace of dimension $M < N$, one can apply power iteration in this subspace to find M eigenvalues/eigenvectors simultaneously. A more refined algorithm of this kind is called *Rayleigh-Ritz subspace iteration [13].* There is also a variant of Davidson iteration that allows for finding $M < N$ eigenvalues/eigenvectors simultaneously. It is called *block Davidson [13, 19].*

2.8.1 Rayleigh-Ritz subspace iteration

We are interested in finding M eigenvalues of a matrix $\mathbf{A}$, where $M < N$, the dimension of the matrix. The simultaneous power iteration algorithm of Eq. 2.50 can be adapted to fulfill that need. One starts with a $N \times M$ matrix $\mathbf{Z}_0$, where $M < N$ is the number of eigenvalues/vectors one requires. The columns of that matrix represent our initial guesses for the eigenvectors. QR factorization also works for such non-square matrices, see the lecture notes on (non-)linear problems. The result of such a QR factorization is a $N \times M$ matrix $\mathbf{Q}$, whose M columns are orthonormal, and a $M \times M$ upper triangular matrix $\mathbf{R}$.

2.8.1.1 Simultaneous power iteration

The simultaneous power iteration algorithm of Eq. 2.50 becomes

$$\mathbf{Z}_0;\ \text{ do } n = 1, ...\ \ \mathbf{Q}_n\mathbf{R}_n = \mathbf{Z}_{n-1};\ \mathbf{Z}_n = \mathbf{A}\mathbf{Q}_n;\ \mathbf{Y}_n = \mathbf{Q}_n^\dagger\mathbf{Z}_n;\ \text{ until converged.} \tag{2.104}$$

Note that all $\mathbf{Z}_n$ and $\mathbf{Q}_n$ are $N \times M$ matrices, whereas all $\mathbf{R}_n$ and $\mathbf{Y}_n$ are $M \times M$ matrices, and $\mathbf{A}$ is a $N \times N$ matrix. The criterion "until converged" is something like $|(Y_n)_{ii} - (Y_{n-1})_{ii}| < \epsilon$, for $i = 1, ..., M$, with ϵ a tolerance chosen by the user. Because of the reduced dimension ($M < N$), this is called *subspace iteration.*

This algorithm produces M eigenvalues and eigenvectors

$$\lambda_i \approx (Y_n)_{ii};\ \mathbf{x}_i \approx (\mathbf{q}_i)_n \quad i = 1, ..., M, \tag{2.105}$$

[32] Orthogonalization adds an $O(M^2N)$ calculation to each iteration, so it is not necessarily cheap.

where $(\mathbf{q}_i)_n$ are the columns of $\mathbf{Q}_n$. Of course, as it is, power iteration finds the M eigenvalues with the largest absolute values, which is probably not what one needs in an actual application. But the algorithm is easily converted into a shifted inverted power iteration algorithm to find the lowest M eigenvalues

$$\mathbf{Z}_0;\ \text{do } n = 1, ...\ \mathbf{Q}_n\mathbf{R}_n = \mathbf{Z}_{n-1};\ \text{solve}\ (\mathbf{A} - s\mathbf{I})\,\mathbf{Z}_n = \mathbf{Q}_n;\ \mathbf{S}_n = \mathbf{Q}_n^\dagger\mathbf{Z}_n;\ \text{until converged.} \tag{2.106}$$

Solving the linear problem inside the loop can for instance be done by LU factorization of $\mathbf{A} - s\mathbf{I}$ outside the loop. Inside the loop, the linear problem is then of the type $\mathbf{LU}\mathbf{z}_k = \mathbf{q}_k$, where $k = 1, ..., M$ denote the columns of the respective matrices. Such problems are easily solved using forward substitution, followed by back substitution. The approximations to the eigenvalues are formed by

$$\lambda_i^{(n)} = \frac{1}{(S_n)_{ii}} + s, \tag{2.107}$$

so one should adapt the convergence criterion accordingly.

2.8.1.2 Rayleigh-Ritz

A more direct way to test convergence is to replace the matrix $\mathbf{S}_n$ by the Rayleigh quotient matrix $\mathbf{Y}_n = \mathbf{Q}_n^\dagger\mathbf{A}\mathbf{Q}_n$ in Eq. 2.106, and proceed with the Rayleigh quotients as convergence criterion. This has the added advantage that the Rayleigh quotient matrix can be used to speed up convergence, if the algorithm of Eq. 2.106 converges too slowly. Because $\mathbf{Y}_n$ is a relatively small $M \times M$ matrix, it can be diagonalized relatively cheaply by a unitary transformation, $\mathbf{Y}_n = \mathbf{W}_n^\dagger\mathbf{\Lambda}'_n\mathbf{W}_n$, by using the Householder transformation plus QR iteration, for instance. From the result one can calculate $\mathbf{V}_n = \mathbf{Q}_n\mathbf{W}_n$, and one may expect that

$$\lambda_i \approx (\Lambda'_n)_i\,;\ \mathbf{x}_i \approx (\mathbf{v}_n)_i \quad i = 1, ..., M, \tag{2.108}$$

where $\mathbf{v}_i$ are the columns of $\mathbf{V}$, are better approximations to the true eigenvalues/eigenvectors than Eq. 2.105. In fact, they are the best approximation to the eigenvalues and eigenvectors one can make within the space spanned by the vectors $\mathbf{q}_k$; $k = 1, ..., M$, see Sec. 2.7.0.1. The vectors $\mathbf{v}$; $k = 1, ..., M$ are called the *Ritz vectors*, and Λ'_k; $k = 1, ..., M$ are of course the *Ritz values*.

The *Rayleigh-Ritz subspace iteration* algorithm is then

$$\mathbf{Z}_0;\ \mathbf{V}_0\mathbf{R}_0 = \mathbf{Z}_0;$$
$$\text{do } n = 1, ...\ \text{solve}\ (\mathbf{A} - s\mathbf{I})\,\mathbf{Z}_n = \mathbf{V}_{n-1};\ \mathbf{Q}_n\mathbf{R}_n = \mathbf{Z}_n \tag{2.109}$$
$$\mathbf{Y}_n = \mathbf{Q}_n^\dagger\mathbf{A}\mathbf{Q}_n;\ \text{diagonalize}\ \mathbf{Y}_n = \mathbf{W}_n^\dagger\mathbf{\Lambda}'_n\mathbf{W}_n;\ \mathbf{V}_n = \mathbf{Q}_n\mathbf{W}_n\ \text{until converged,} \tag{2.110}$$

where "until converged" is a criterion of the type $\left|(\Lambda'_n)_i - \left(\Lambda'_{n-1}\right)_i\right| < \epsilon$, for $i = 1, ..., M$, with ϵ a tolerance chosen by the user. The Ritz values give you an approximation to the eigenvalues $(\Lambda'_n)_i \approx \lambda_i$; $i = 1, ..., M$, and the columns of the $N \times M$ matrix $\mathbf{V}$ give approximations to the corresponding eigenvectors.

One of the tricky points in the algorithm is solving the linear problem in Eq. 2.109. Although LU decomposition can be done outside the loop, $\mathbf{A}$ is a large matrix. As the computational cost of LU factorization scales as $O(N^3)$, it may become too expensive for large N. One can be forced to use an iterative technique for solving the linear problem, which then adds an extra do-loop for the linear problem, inside the do-loop over n.

2.8.2 Block Davidson*

Davidson iteration and GJD are fairly complex algorithms, see Secs. 2.7.1.1 and 2.7.1.2, but one of their advantages is that they can be straightforwardly extended to calculate $M < N$ eigenvalues/eigenvectors simultaneously [19]. I won't repeat the algorithm here, but just tell what changes in Eqs. 2.75-2.82, in case you want M eigenvalues simultaneously, where $1 < M < N$.

Instead of starting from a single vector $\mathbf{q}_0$ (a $N \times 1$ matrix) in Eq. 2.75, one can start from a $N \times M$ matrix $\mathbf{Q}_0$, whose columns $(\mathbf{q}_0)_i\,;\ i = 1, ..., M$ are orthonormal, $(\mathbf{q}_0)_i^\dagger (\mathbf{q}_0)_j = \delta_{ij}$. Then $\mathbf{r}_0$ and $\mathbf{v}$ also become $N \times M$ matrices, whereas ρ_0 becomes an $M \times M$ diagonal matrix, with matrix elements $(\rho_0)_{ij} = \delta_{ij} (\mathbf{q}_0)_i^\dagger \mathbf{A} (\mathbf{q}_0)_i$.

The matrix $\mathbf{\Delta}$ in Eq. 2.76 then becomes an $N \times M$ matrix. The linear problem is of the type $\mathbf{A}'\mathbf{\Delta} = \mathbf{B}$, with $\mathbf{B}$ a $N \times M$ matrix. It consists of M linear problems of type $\mathbf{A}' (\mathbf{\Delta})_i = (\mathbf{B})_i\,;\ i = 1, ..., M$, one for each column of $\mathbf{\Delta}$ and $\mathbf{B}$. In practice one often uses an approximate inverse, as in Eq. 2.82.

The next step, Eq. 2.77, consist of orthogonalizing all vectors $(\mathbf{\Delta})_i\,;\ i = 1, ..., M$ to all previously found vectors $\mathbf{q}_k;\ k = 1, ..., M(n-1)$, and orthogonalizing the vectors $(\mathbf{\Delta})_i$ among themselves, to produce the orthonormal set $(\mathbf{q}_n)_i\,;\ i = 1, ..., M$. This can be done by modified Gram-Schmidt, or by another type QR factorization, see Secs. 1.2.1 and 2.6.3.

In Eq. 2.78 the subspace is extended by adding to the $N \times (n-1)M$ matrix $\mathbf{Q}_{n-1}$ the newly found M vectors $(\mathbf{q}_n)_i$ to construct the $N \times nM$ matrix $\mathbf{Q}_n$. Note that after each iteration the dimension of the subspace has grown by M. The Rayleigh matrix $\mathbf{Y}_n$ is then a $nM \times nM$ matrix. We assume that this is still small enough to diagonalize it completely by a unitary transformation, using the Householder transformation plus QR iteration, for instance.

Eq. 2.79 then selects the lowest M Ritz values, constructs the corresponding Ritz vectors, and the residual vectors for the next iteration.

2.9 The generalized eigenvalue problem

Besides the eigenvalue problem of Eq. 2.1, in physics one also regularly encounters an eigenvalue problem of the type

$$\mathbf{A}\mathbf{x}_n = \lambda_n \mathbf{B}\mathbf{x}_n, \tag{2.111}$$

where both $\mathbf{A}$ and $\mathbf{B}$ are Hermitian $N \times N$ matrices, $\mathbf{x}_n$ is a column vector of length N, and λ_n is a number. This is called the *generalized eigenvalue problem*. As before, λ_n is an *eigenvalue* and $\mathbf{x}_n$ is the corresponding *eigenvector*.

Formally, this problem is equivalent to the ordinary eigenvalue problem $\mathbf{C}\mathbf{x}_n = \lambda_n \mathbf{x}_n$, with $\mathbf{C} = \mathbf{B}^{-1}\mathbf{A}$, so one can apply many of the techniques for solving eigenvalue problems introduced in the previous sections. Of course, one should avoid an explicit construction of

the matrix $\mathbf{B}^{-1}$, and calculating the matrix product $\mathbf{B}^{-1}\mathbf{A}$. Both of these calculations are $O(N^3)$ processes and expensive if N is large.

The techniques we discussed in Sec. 2.3 for finding a single eigenvalue/eigenvector can be modified straightforwardly. For instance, power iteration, which is based upon the iteration rule $\mathbf{b}_{n+1} = \mathbf{C}\mathbf{b}_n = \mathbf{B}^{-1}\mathbf{A}\mathbf{b}_n$, Eq. 2.9, can now use the iteration rule

$$\mathbf{B}\mathbf{b}_{n+1} = \mathbf{A}\mathbf{b}_n. \tag{2.112}$$

This can be solved as a linear problem, by LU decomposition, for instance. As the matrix $\mathbf{B}$ is fixed, its LU decomposition can be done outside the power iteration loop. Inside the loop, one does the usual forward substitution, followed by back substitution to solve the linear problem. Note that the Rayleigh quotient as calculated in Eq. 2.9, is still correct, as $R_n = \mathbf{b}_n^\dagger \mathbf{B}^{-1}\mathbf{A}\mathbf{b}_n = \mathbf{b}_n^\dagger \mathbf{b}_{n+1}$, using Eq. 2.112.

Shifted inverse power iteration can be adapted similarly. The iteration rule $(\mathbf{B}^{-1}\mathbf{A} - s\mathbf{I})\,\mathbf{b}_{n+1} = \mathbf{b}_n$, see Eq. 2.14, can be rewritten as

$$(\mathbf{A} - s\mathbf{B})\,\mathbf{b}_{n+1} = \mathbf{B}\mathbf{b}_n, \tag{2.113}$$

which again can be solved as a linear problem: LU decomposition of $\mathbf{A} - s\mathbf{B}$ outside the loop, and forward substitution, followed by back substitution, inside the loop.

Similarly, one can modify Rayleigh quotient iteration, Eq. 2.19, using the iteration rule

$$(\mathbf{A} - R_n\mathbf{B})\,\mathbf{b}_{n+1} = \mathbf{B}\mathbf{b}_n, \tag{2.114}$$

Calculating the Rayleigh quotient $R_n = \mathbf{b}_n^\dagger \mathbf{B}^{-1}\mathbf{A}\mathbf{b}_n$ might seem somewhat of a problem. However, one can go through the moves leading to Eq. 2.20, and show that is still valid.

Calculating multiple eigenvalues/eigenvectors for Eq. 2.111 is less straightforward. The matrix product $\mathbf{C} = \mathbf{B}^{-1}\mathbf{A}$ is *not Hermitian* in general, even if both matrices $\mathbf{A}$ and $\mathbf{B}$ are Hermitian (and the inverse of a Hermitian matrix $\mathbf{B}^{-1}$ is Hermitian). This means we cannot use any of the techniques based upon unitary transformations, introduced in Secs. 2.4-2.6.[33]

We can, however, save the furniture if $\mathbf{A}$ and $\mathbf{B}$ are Hermitian and $\mathbf{B}$ is also *positive definite [7]*. In the lecture notes on (non-)linear problems we have seen that for positive definite Hermitian matrices we can use the Cholesky decomposition to write $\mathbf{B} = \mathbf{L}\mathbf{L}^\dagger$, with $\mathbf{L}$ a lower triangular matrix. Very often in physical problems, the matrix $\mathbf{B}$ in Eq. 2.111 is an overlap matrix or a metric, and is indeed positive definite, see Sec. 1.1.5.

One can rewrite $\mathbf{A}\mathbf{x}_n = \lambda_n \mathbf{B}\mathbf{x}_n = \lambda_n \mathbf{L}\mathbf{L}^\dagger \mathbf{x}_n$ as $\mathbf{L}^{-1}\mathbf{A}\mathbf{x}_n = \lambda_n \mathbf{L}^\dagger \mathbf{x}_n$, and finally as

$$\mathbf{L}^{-1}\mathbf{A}\left(\mathbf{L}^\dagger\right)^{-1}\left(\mathbf{L}^\dagger \mathbf{x}_n\right) = \lambda_n \left(\mathbf{L}^\dagger \mathbf{x}_n\right). \tag{2.115}$$

It is not difficult to show that $\left(\mathbf{L}^\dagger\right)^{-1} = (\mathbf{L}^{-1})^\dagger$, so

$$\mathbf{G} = \mathbf{L}^{-1}\mathbf{A}\left(\mathbf{L}^\dagger\right)^{-1} \quad \text{is Hermitian.} \tag{2.116}$$

Proof: $\left(\mathbf{L}^\dagger\right)^{-1}\mathbf{L}^\dagger = \mathbf{I} = \left(\mathbf{L}\mathbf{L}^{-1}\right)^\dagger = \left(\mathbf{L}^{-1}\right)^\dagger \mathbf{L}^\dagger$, and $\mathbf{L}^\dagger\left(\mathbf{L}^\dagger\right)^{-1} = \mathbf{I} = \left(\mathbf{L}^{-1}\mathbf{L}\right)^\dagger = \mathbf{L}^\dagger\left(\mathbf{L}^{-1}\right)^\dagger$, so $\left(\mathbf{L}^\dagger\right)^{-1} = \left(\mathbf{L}^{-1}\right)^\dagger$. Then $\mathbf{G}^\dagger = \left[\mathbf{L}^{-1}\mathbf{A}\left(\mathbf{L}^\dagger\right)^{-1}\right]^\dagger = \left[\mathbf{L}^{-1}\mathbf{A}\left(\mathbf{L}^{-1}\right)^\dagger\right]^\dagger = \mathbf{L}^{-1}\mathbf{A}^\dagger\left(\mathbf{L}^{-1}\right)^\dagger = \mathbf{L}^{-1}\mathbf{A}\left(\mathbf{L}^\dagger\right)^{-1} = \mathbf{G}$.

[33] One problem, for instance, is that the eigenvectors $\mathbf{x}_n$ are, in general, *not orthogonal*, as there is no a priori reason for the eigenvectors of a non-Hermitian matrix to be orthogonal. This means we cannot use any techniques that enforce the orthogonality of vectors, such as Gram-Schmidt or QR factorization.

Instead of the generalized eigenvalue problem, Eq. 2.111, one then solves the ordinary eigenvalue problem

$$\mathbf{G}\mathbf{y}_n = \lambda_n \mathbf{y}_n, \tag{2.117}$$

which has the same eigenvalues as Eq. 2.111. The eigenvectors of the original problem can be found from

$$\mathbf{L}^\dagger \mathbf{x}_n = \mathbf{y}_n, \tag{2.118}$$

by back substitution.

In order to construct $\mathbf{G}$, Eq. 2.116, one does not have to construct the inverse $\mathbf{L}^{-1}$ explicitly, nor do the matrix products explicitly. Writing Eq. 2.116 as $\mathbf{LG} = \mathbf{A}\left(\mathbf{L}^\dagger\right)^{-1} \equiv \mathbf{V}^\dagger$, one first finds the matrix $\mathbf{V}$ by solving the linear problem $\mathbf{V}^\dagger\mathbf{L}^\dagger = \mathbf{A}$, or its conjugate problem $\left(\mathbf{V}^\dagger\mathbf{L}^\dagger\right)^\dagger = \mathbf{A}^\dagger$, or

$$\mathbf{LV} = \mathbf{A}. \tag{2.119}$$

As a linear problem it is equivalent to N linear vector problems, one for each column of $\mathbf{V}$, which can be solved by forward substitution. In a second step, one finds the matrix $\mathbf{G}$ by solving the linear problem

$$\mathbf{LG} = \mathbf{V}^\dagger. \tag{2.120}$$

Again, this is a linear problem for each of the N columns of $\mathbf{G}$, which can be solved by forward substitution.

Chapter 3

OPTIMIZATION

Optimization is a problem that occurs frequently in physics. The mathematical problem is, given a function $f(x_1, x_2, ..., x_N)$ of the variables $x_i;\ i = 1, ..., N$, find the point $x_i = x_i^{\min};\ i = 1, ..., N$ where the function f is minimal. In these notes I will stick to real functions and real variables.[1]

In physics, functions that have a such a stationary point get the label *variational*, or are said to *obey a variational principle*. Such functions f exist in different fields of physics. In mechanics and in field theory, for instance, one has a function called the *action*, and the corresponding variational principle is called the *principle of least action*.[2] In the variational principle of quantum mechanics, the function f is the total energy E, and the variables x_i describe parameters in the wave function that you can modify to find the stationary points of $E(x_1, x_2, ..., x_N)$. Finite-element methods in engineering are often accompanied by a variational principle (the Galerkin method, for instance).

Defining a function $\tilde{f} = -f$, means that finding the minimum of f is equivalent to finding the maximum of $\tilde{f}$. Finding extrema of functions is commonly dubbed *optimization*.[3] As by changing the sign of the function, a maximum can be turned into a minimum, I will only discuss finding minima in these notes.

I will discuss optimization algorithms that share the same procedure for finding a minimum of a function $f(x_1, x_2, ..., x_N)$

1. Start at a point $x_i;\ i = 1, ..., N$. If you have a guess of the region where the minimum could be, start with a point in that region; if not, just pick a random point. Then iterate on the following steps:

2. Calculate $f(x_1, x_2, ..., x_N)$. Find a direction $p_i;\ i = 1, ..., N$ that, if you move along it, brings you closer to the minimum of f.

3. Determine the step size α that minimizes $f(x_1 + \alpha p_1, x_2 + \alpha p_2, ..., x_N + \alpha p_N)$. Your new guess now is $x_i = x_i + \alpha p_i$. If you are sufficiently close to the minimum, stop. If not, return to 2.

[1] There are problems, where x_i are complex variables, in quantum mechanics, for instance. However, in the practical cases I know of, f is a real number. Such problems can be dealt with by treating $\mathrm{Re}x_i$ and $\mathrm{Im}x_i$ as two separate real variables, for instance.

[2] Calculating the stationary point of the action is a lot of toil and trouble, so to call it "least action" is clever advertising.

[3] Again clever advertising. Minimization sounds too negative, maximization sounds too greedy, and who ever would object to doing things optimal? Economists are in constant need of finding extrema of functions. So they optimize the cost by optimizing the work force in order to optimize the profit.

The different methods vary in the amount of data you are willing to calculate. If you can only calculate function values $f(x_1, x_2, ..., x_N)$, but no additional information, then determining proper directions (step 2.) and step sizes (step 3.) is difficult. There are methods for doing that, but I will skip these, and assume that we can get our hands on more information. If you can calculate derivatives $\partial f(x_1, x_2, ..., x_N)/\partial x_i;\ i = 1, ..., N$, then you can say something about the direction into which to search (step 2.). Defining the N-dimensional gradient by $\nabla = (\partial/\partial x_1, \partial/\partial x_2, ..., \partial/\partial x_N)$, you know that $-\nabla f(x_1, x_2, ..., x_N)$ points in the direction of *maximal decrease* of the function f at the point $(x_1, x_2, ..., x_N)$.

Setting $p_i = -\partial f(x_1, x_2, ..., x_N)/\partial x_i;\ i = 1, ..., N$ defines the method of *steepest descent*. It sounds attractive, but it can take a disappointingly large number of iterations to find the minimum. A modification of this idea uses the gradient to determine the search direction in a more clever way. This leads to the *conjugate gradient* method, which is absolutely brilliant, and has become one of the *standard methods* for optimization.

Having dealt with step 2., one turns to step 3.. This is essentially a one-dimensional optimization problem, i.e., find the minimum of $F(\alpha) \equiv f(x_1 + \alpha p_1, x_2 + \alpha p_2, ..., x_N + \alpha p_N)$. It is called a *line search.* There are many techniques available, whose relative attraction depends on how accurate you want the minimum, and how much computational work you can afford to put into it. I will give a couple of examples.

Step 3. is a lot easier if, besides the gradient, you can also determine second derivatives $\partial^2 f(x_1, x_2, ..., x_N)/\partial x_i \partial x_j;\ i = 1, ..., N;\ j = 1, ..., N$. In that case, we can use the *Newton-Raphson* method, which actually combines steps 2. and 3. into one step. In principle, it is a good method for finding a function minimum. Unfortunately, in most physics problems it is simply computationally too costly to determine the second derivatives, and without them you cannot use the Newton-Raphson method. However, not all is lost, as there are methods around that use gradients to cleverly build approximations to the second derivatives. These methods go under the name *quasi-Newton methods* or *Broyden methods* (the mathematicians also call them *secant methods*). One particular flavor, the BFGS method (also called a *variable metric method*), has become one of the most widely used tools in function optimization. Last, but not least, the *Pulay algorithm,* discussed in the lecture notes on (non-)linear problems, can also be formulated as a Broyden method.

One problem remains. All methods I will discuss, and indeed all other methods I know of, if they work, they find you **a minimum** of $f(x_1, x_2, ..., x_N)$. There is no guarantee that that is **the absolute minimum**, i.e., the minimum that is lower than all other minima. It is usually the absolute minimum you want; it represents the ground state energy, or the minimum free energy, for instance. The algorithms only give you a local minimum, a point where the function value is lower than at points in its surrounding. How do you know that there is not some other minimum that is lower than the one the algorithm finds? **You don't**, unless you have some additional knowledge of the physical problem at hand telling you otherwise (some physical functions have a single, unique minimum, for instance). The "solution" to the problem of finding the absolute minimum is **trial and error**. You can try a number of widely different starting points (step 1.), and see where the algorithm takes you. Alternatively, once you have found a minimum, you can make a step that takes you far away (step 1.), and optimize again. Exploring the function landscape is called an *exhaustive search.*[4]

Optimization is a subject treated, or at least introduced, in most textbooks on numerical

[4]Not exhaustive for the computer, but for the user. It's advertising again. One uses "exhaustive search" to intimidate the reader into thinking that he/she cannot possibly do better.

mathematics, see, e.g., Refs. [4, 5, 6]. There are also excellent and accessible monographs on this subject, see Refs. [20, 21, 22].[5] The popular "Numerical Recipes" book has a very nice chapter on optimization [7]. For the conjugate gradient and steepest descent methods in particular, I can recommend Ref. [23].[6] For those who cannot survive without in-depth mathematics, see Ref. [3].

3.1 The quadratic problem

The mother of all optimization problems, which has spawned most of the optimization algorithms, is the quadratic problem, where

$$f(x_1, x_2, ..., x_N) = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_{ij}x_i x_j - \sum_{i=1}^{N} b_i x_i + c. \tag{3.1}$$

Here $a_{ij};\ i = 1, ..., N;\ j = 1, ..., N$, $b_i;\ i = 1, ..., N$, and c are fixed (positive or negative) numbers. The factor $\frac{1}{2}$ before the first term and the $-$ sign before the second term are just for convenience. It is sensible to define a $N \times 1$ column vector $\mathbf{x}$ with elements $(\mathbf{x})_j = x_j$. From this one can define a $1 \times N$ row vector $\mathbf{x}^T$ with elements $\left(\mathbf{x}^T\right)_i = x_i$, and rewrite Eq. 3.1 in matrix-vector form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b} + c, \tag{3.2}$$

where the $N \times N$ matrix $\mathbf{A}$ has the elements $(\mathbf{A})_{ij} = a_{ij}$. We assume that this matrix is non-singular.

Details: $\mathbf{a}^T\mathbf{b}$ is just another way of writing the inner product $\mathbf{a} \cdot \mathbf{b}$. The first notation has to be treated as matrix multiplication between a $1 \times N$ matrix (a row vector) $\mathbf{a}^T$ and a $N \times 1$ matrix (a column vector) $\mathbf{b}$. The result is $\mathbf{a}^T\mathbf{b} = \sum_{i=1}^{N} a_i b_i$. This notation is more versatile than the "dot" notation, as it treats vectors and matrices on the same footing. Note that $\mathbf{a}^T\mathbf{b} = \mathbf{b}^T\mathbf{a}$.

When you are faced with finding the minimum of a function $f(x_1, x_2, ..., x_N)$, you would start by setting all first derivatives to zero $\partial f(x_1, x_2, ..., x_N)/\partial x_i = 0;\ i = 1, ..., N$. Applying this to Eq. 3.1, and writing the result in matrix-vector form as in Eq. 3.2, you get

$$\frac{1}{2}\left(\mathbf{A} + \mathbf{A}^T\right)\mathbf{x}^{\min} - \mathbf{b} = 0. \tag{3.3}$$

This is a linear problem, i.e., a set of N linear equations. We can simplify it somewhat by assuming that $\mathbf{A}$ is a symmetric matrix, $\mathbf{A}^T = \mathbf{A}$.[7] The matrices used in optimization problems in physics are typically symmetric anyway, so this is not much of a restriction in practice. Eq. 3.3 then becomes

$$\mathbf{A}\mathbf{x}^{\min} = \mathbf{b}, \tag{3.4}$$

which is just a regular linear problem.

[5] Written by mathematicians! They can make things accessible, if they want to.
[6] Written by a mathematician who has pity on us mortals.
[7] Remember, a symmetric matrix is the real version of a Hermitian matrix.

Finding a point where all first derivatives are zero, means you have found a *stationary point* of f.[8] It does not automatically mean that the stationary point is a minimum; it could be a maximum or a saddle point.[9] In one dimension, you would look at the sign of the second derivative of the function at the point where the first derivative is zero. If it is positive, then the point represents a minimum. In N dimensions, the second derivatives of Eq. 3.1 are the coefficients a_{ij}, in other words, the matrix $\mathbf{A}$. The "positive second derivative demand" is then generalized to

$$\mathbf{A} \text{ positive definite, which means } \mathbf{x}^T\mathbf{A}\mathbf{x} > 0 \text{ for all } \mathbf{x}. \tag{3.5}$$

Proof: to study whether the solution to Eq. 3.4 represents a minimum of Eq. 3.2, we write any point $\mathbf{x} = \mathbf{x}^{\min}+\boldsymbol{\Delta}$, where $\mathbf{x}^{\min} = \mathbf{A}^{-1}\mathbf{b}$. After a bit of rewriting one gets $f(\mathbf{x}) = \frac{1}{2}\left(\mathbf{x}^{\min}+\boldsymbol{\Delta}\right)^T\mathbf{A}\left(\mathbf{x}^{\min}+\boldsymbol{\Delta}\right) - \left(\mathbf{x}^{\min}+\boldsymbol{\Delta}\right)^T\mathbf{b} + c = \frac{1}{2}\boldsymbol{\Delta}^T\mathbf{A}\boldsymbol{\Delta} - \frac{1}{2}\mathbf{b}^T\mathbf{A}^{-1}\mathbf{b}$. This has a minimum at $\boldsymbol{\Delta} = \mathbf{0}$, provided $\boldsymbol{\Delta}^T\mathbf{A}\boldsymbol{\Delta} > 0$ for all $\boldsymbol{\Delta}$.

In summary, finding the minimum of the quadratic function of Eq. 3.1 amounts to solving the linear problem of Eq. 3.4. The matrix $\mathbf{A}$ has to be positive definite, Eq. 3.5, otherwise the function does not have a minimum. Eq. 3.4 has one, unique solution, so the quadratic function has one, unique minimum. In principle, you can use any of the techniques we discussed in the lecture notes on (non-)linear problems to solve the linear problem of Eq. 3.4. Using *Cholesky decomposition* would be perfect, as that also allows you to establish at the same time whether the matrix $\mathbf{A}$ is positive definite (in case you wouldn't know on forehand from the physics of your problem). Indeed that is what I suggest as a standard technique for minimizing a quadratic function in N dimensions.

Cost Cholesky decomposition costs $O(N^3)$ operations for a general matrix $\mathbf{A}$, so there is a size N where this scaling catches up with you, and it becomes too expensive. In the lecture notes on (non-)linear problems I have discussed iterative algorithms that solve linear problems in $O(nN^2)$ operations, with n the number of iterations. In the next sections we will have a look at a couple of iterative techniques that are particularly suited for solving a linear problem for the special case where $\mathbf{A}$ is a symmetric, positive definite matrix. In view of the discussion above, these methods can be interpreted as techniques for finding the minimum of a quadratic function. They scale as $O(nN^2)$, and are robust (in contrast to some of the methods discussed in the lecture notes on (non-)linear problems).

[8] The mathematicians also use the phrase *critical point* for this, but I avoid that phrase. "Critical point" has a very specific (and very different) meaning within thermodynamics and statistical physics, and as in these subjects one often has to deal with minimization problems, confusion is bound to arise. "Stationary point" indicates a point where the forces are zero. With f representing a potential, the forces are the first derivatives of the potential. Rule no. 1: in case different phrases in mathematics and physics are used for the same thing, follow the physics. Rule no. 2: in case you are in doubt, follow the physics. Rule no. 3: in case you are sure, follow the physics.

[9] In two dimensions, a *saddle point* is a point where the function is at a minimum with respect to one direction, and at a maximum with respect to another direction (like the highest point of a mountain pass). The center of the top plane of a horse saddle is such a point, hence the name. In N dimensions, any point where the function is at a minimum with respect to $M < N$ linear independent directions is called a saddle point. It figures that it should be at a maximum with respect to the other $N - M$ linear independent directions.

3.2 Steepest descent (SD)

We start by picking some point $\mathbf{x}_0$. Unless we are lucky beyond believe, $f(\mathbf{x}_0)$ is not a minimum. So, where do we go from here? For points $\mathbf{x}$ close to $\mathbf{x}_0$, write $\mathbf{x} = \mathbf{x}_0 + \mathbf{\Delta}$, and make a Taylor expansion around $\mathbf{x}_0$

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \mathbf{\Delta}^T \nabla f(\mathbf{x}_0) \quad \text{with } \nabla = \begin{pmatrix} \partial/\partial x_1 \\ \partial/\partial x_2 \\ \vdots \\ \partial/\partial x_N \end{pmatrix}, \tag{3.6}$$

the N-dimensional gradient vector operator. Write $\mathbf{\Delta}^T \nabla f(\mathbf{x}_0) = \|\mathbf{\Delta}\| \, \|\nabla f(\mathbf{x}_0)\| \cos\phi$, with ϕ the angle between the vectors $\mathbf{\Delta}$ and $\nabla f(\mathbf{x}_0)$. Choose the direction of $\mathbf{\Delta}$ defined by $\phi = \pi$, then $\cos\phi = -1$, and a step in that direction sends you downhill in the steepest possible way. In other words, choose the next point as

$$\mathbf{x}_1 = \mathbf{x}_0 - \alpha \nabla f(\mathbf{x}_0), \tag{3.7}$$

with $\alpha > 0$. The gradient of a quadratic function f, Eq. 3.2, is easily calculated

$$\nabla f(\mathbf{x}_0) = \mathbf{A}\mathbf{x}_0 - \mathbf{b} \equiv \mathbf{r}_0, \tag{3.8}$$

with $\mathbf{r}_0$ the *residual vector* (see the lecture notes on (non-)linear problems).

Once we have set the direction in which to go, the next question is, how far should we go? In other words, what is the best choice for α? We can work out $f(\mathbf{x}_1) = f(\mathbf{x}_0 - \alpha \mathbf{r}_0)$ from Eq. 3.2

$$f(\mathbf{x_0} - \alpha \mathbf{r_0}) = f(\mathbf{x}_0) - \alpha \left[\mathbf{r}_0^T \mathbf{A}\mathbf{x}_0 - \mathbf{r}_0^T \mathbf{b}\right] + \frac{1}{2}\alpha^2 \mathbf{r}_0^T \mathbf{A}\mathbf{r}_0 \equiv F(\alpha), \tag{3.9}$$

where we have used $\mathbf{x}_0^T \mathbf{A}\mathbf{r}_0 = \mathbf{r}_0^T \mathbf{A}\mathbf{x}_0$, as $\mathbf{A} = \mathbf{A}^T$, and $\mathbf{a}^T\mathbf{b} = \mathbf{b}^T\mathbf{a}$. Interpreting this as a one-dimensional quadratic function $F(\alpha)$, the optimum $\alpha^{\min}$ is found from $dF(\alpha^{\min})/d\alpha = 0$, which gives

$$\alpha^{\min} = \frac{\mathbf{r}_0^T \mathbf{r}_0}{\mathbf{r}_0^T \mathbf{A}\mathbf{r}_0}, \tag{3.10}$$

where we have used Eq. 3.8. We are guaranteed that the stationary point found from $dF(\alpha^{\min})/d\alpha = 0$ gives a minimum. A quadratic function $F(\alpha)$ (a parabola) either has a single minimum or a single maximum, and as we started walking downhill from the point $\mathbf{x}_0$, it has to be a minimum.

Going through this procedure repeatedly gives us the method of *steepest descent* for minimizing a quadratic function

$$\mathbf{x}_0 = \mathbf{v}; \quad \mathbf{r}_0 = \mathbf{A}\mathbf{x}_0 - \mathbf{b};$$
$$\text{do} \quad n = 0, \ldots \quad \alpha_n = \frac{\mathbf{r}_n^T \mathbf{r}_n}{\mathbf{r}_n^T \mathbf{A}\mathbf{r}_n}; \; \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \mathbf{r}_n; \; \mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \mathbf{b}; \quad \text{until converged;} \tag{3.11}$$

For the "until converged" criterion we have several options. One can consider $\|\mathbf{x}_{n+1} - \mathbf{x}_n\| < \epsilon$, for instance, with ϵ a tolerance set by the user, or $\|\mathbf{r}_{n+1}\| < \epsilon'$ to make sure that the gradient has become sufficiently small. Whatever you do, it makes sense to test the latter as well.

Cost The computational costs of the algorithm are foremost determined by the cost of matrix-vector multiplications $\mathbf{Av}$, which scale as $O(N^2)$. There are two of these multiplications per cycle in Eq. 3.11. This can be reduced by writing

$$\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha_n \mathbf{A}\mathbf{r}_n. \tag{3.12}$$

Proof: $\mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \mathbf{b} = \mathbf{A}\left(\mathbf{x}_n - \alpha_n \mathbf{r}_n\right) - \mathbf{b} = \mathbf{r}_n - \alpha_n \mathbf{A}\mathbf{r}_n.$

Per iteration in Eq. 3.11 one then needs only one matrix-vector multiplication $\mathbf{A}\mathbf{r}_n$. However, following life's universal rule,[10] using Eq. 3.12 repeatedly, brings the disadvantage of accumulating computer round-off errors. So, every 50 cycles or so, one should replace it by the original Eq. 3.11.

The total algorithm scales as $O(nN^2)$, where n is the number of cycles required to achieve convergence. It is thus very important to keep n as small as possible. To perform better than the direct methods for solving Eq. 3.4 (Cholesky decomposition), we must have $n < N$.

3.2.1 Convergence of steepest descent

The steepest descent algorithm, Eq. 3.11, is simple, but unfortunately not always successful. For all quadratic problems, it does converge to the minimum, so it is robust. For some quadratic problems it converges fast in a few iterations, but for others it converges slow, requiring a large number of iterations. Worse than that, for some quadratic problems, the number of iterations needed to converge, depends super-critically on your starting point $\mathbf{x}_0$. In the analysis of why this happens, usually a figure similar to Fig. 3.1 is shown. It is a contour plot of a two-dimensional quadratic function $f(x_1, x_2)$. The starting point $\mathbf{x}_0$ is the dot in the upper right quadrant. The staircase is the path through $\mathbf{x}$-space, generated by the steepest descent algorithm, with the points $\mathbf{x}_n$; $n = 1, 2, 3, ...$ on the corners of the staircase.

At all these points the path takes a 90° turn. Eventually the path reaches the minimum of the function at $(x_1, x_2) = (0, 0)$ in the center of the graph, but the obvious question is: why does the path meander around so much? The reason is that the 90° turns are forced by the steepest descent algorithm. The graphical explanation goes as follows. At the starting point $\mathbf{x}_0$ we head off in a direction perpendicular to a contour line of $f(x_1, x_2)$. We have to, as contour lines connect points with the same function value f, and the gradient $\mathbf{r}_0$ is perpendicular to such lines.[11] The gradient at the next point $\mathbf{r}_1$ is perpendicular to $\mathbf{r}_0$.

This can be concluded from a *reductio ad absurdam.* The point $\mathbf{x}_1$ is determined by minimizing along the direction of $\mathbf{r}_0$. If the gradient at $\mathbf{x}_1$ had a component along $\mathbf{r}_0$, then walking in the (opposite) direction of this component would bring us to a lower function value. This is impossible, as $\mathbf{x}_1$ represents a minimum, which means that $\mathbf{r}_1$ cannot have a component along $\mathbf{r}_0$. One can use the same reasoning at each point $\mathbf{x}_n$, which means

$$\mathbf{r}_{n+1}^T \mathbf{r}_n = 0 \quad \text{for } n = 0, 1, ... \tag{3.13}$$

Of course, one can also prove this algebraically from Eq. 3.11.

---

[10] "Elluk foordeil hep sun nadeil"

[11] An example from electrostatics. The contour lines are equipotential lines, and the gradient is the electric field.

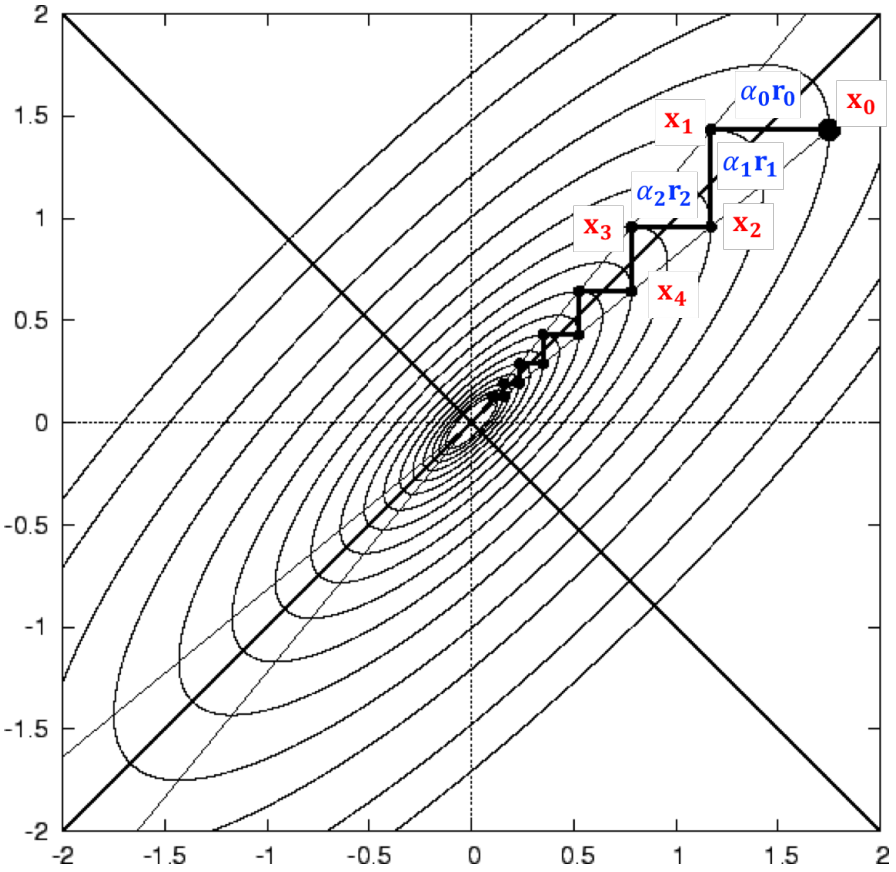


Figure 3.1: The method of steepest descent, Eq. 3.11, presented graphically in two dimensions. The ellipses represent a contour plot of a two-dimensional function $f(\mathbf{x})$. The thick solid line represents the path generated by steepest descent. Each straight line piece begins of at a point $\mathbf{x}_n$, and end at a point $\mathbf{x}_{n+1}$. At its begin, the line piece is orthogonal to a contour, because its direction is proportional to the gradient, $\alpha_n\mathbf{r}_n$. At its end, the line piece is parallel to a contour, because we have minimized the function along the line. That however means that the next line piece should be in an orthogonal direction.

Proof: use Eq. 3.12. It then follows that $\mathbf{r}_{n+1}^T\mathbf{r}_n = \mathbf{r}_n^T\mathbf{r}_n\left(1 - \mathbf{r}_n^T\mathbf{A}\mathbf{r}_n/\mathbf{r}_n^T\mathbf{A}\mathbf{r}_n\right) = 0$.

Because of Eq. 3.13, any path generated by the steepest descent method in two dimensions is a staircase. In $N > 2$ dimensions the path also consists of pieces of straight lines (of length $\alpha_n \|\mathbf{r}_n\|$), connected at 90° angles. In the worst case scenario, you are spiraling in N-dimensional space very slowly toward the minimum.

The severity of this staircase effect very much depends upon you starting point. As you can see in Fig. 3.1, there are starting points that bring you to the minimum in an instant. If you start with a point somewhere on the $x_1 = x_2$ or $x_1 = -x_2$ lines, a single calculation of the gradient puts you on a path that leads you straight to the minimum. Of course, this only holds for this particular example. In other quadratic problems, these perfect lines lie at other positions, and you don't know on forehand where they lie. In practice you will always miss them, and you suffer from the staircase effect.[12]

[12] Murphy's law.

There are two different remedies against it, one simple (and not very effective), and one more complicated (and very effective). In the simple remedy, one alters the step sizes α_n. That gets rid of the 90° angles between the line pieces, as that is the result of Eq. 3.13, which only holds for α_n as used in Eq. 3.11. Choosing any other α gives

$$\mathbf{r}_{n+1}^T \mathbf{r}_n = \mathbf{r}_n^T \mathbf{r}_n - \alpha \mathbf{r}_n^T \mathbf{A} \mathbf{r}_n \neq 0, \tag{3.14}$$

using Eq. 3.12. It is not clear on forehand what the optimal α in each step would be in order to reduce the number of iterations. Some people try their luck with a fixed α, which leads to a very simple algorithm

$$\begin{array}{l} \mathbf{x}_0 = \mathbf{v}; \quad \mathbf{r}_0 = \mathbf{A}\mathbf{x}_0 - \mathbf{b}; \\ \text{do} \quad n = 0, ... \quad \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \mathbf{r}_n; \quad \mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \mathbf{b}; \quad \text{until converged;} \end{array} \tag{3.15}$$

This algorithm is also called *steepest descent* in the literature, although the phrase should be more properly reserved for the algorithm of Eq. 3.11. The parameter α is chosen by trial and error. Choose it too large, and the algorithm might not converge at all, unlike Eq. 3.11, which does always converge for quadratic problems. Choose it too small, and convergence might take forever. More elaborate schemes have been invented, which I skip as there are better options.

A better option to accelerate the algorithm is to focus on the search directions. Looking at Fig. 3.1, the gradient $\mathbf{r}_n$ gives you the direction to go down *steepest* from a point $\mathbf{x}_n$, but it does not give the *best* direction to the minimum. It should be possible to find a better direction.

3.3 Conjugate gradient (CG)

The conjugate gradient method is one of the most popular and widespread optimization techniques [20, 21]. Good introductions to the why and how of this method can be found in Refs. [23] and [7], respectively.

3.3.1 Conjugate basis vectors

Let us go back to the original problem, Eq. 3.2. Define a set of linear independent vectors $\mathbf{p}_i$; $i = 1, ..., N$, and use these as a basis set. Then all vectors $\mathbf{x}$ can be expressed as $\mathbf{x} = \sum_{i=1}^N c_i \mathbf{p}_i$, with c_i; $i = 1, ..., N$ some coefficients. Eq. 3.2 becomes

$$f(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N c_i c_j \mathbf{p}_i^T \mathbf{A} \mathbf{p}_j - \sum_{i=1}^N c_i \mathbf{p}_i^T \mathbf{b} + c \equiv f(c_1, c_2, ..., c_N). \tag{3.16}$$

We minimize f as a function of the coefficients c_i. This would be a lot easier if we choose our basis set such, that

$$\mathbf{p}_i^T \mathbf{A} \mathbf{p}_j = 0 \quad \text{if } i \neq j; \quad i = 1, ...N; \quad j = 1, ..., N. \tag{3.17}$$

Eq. 3.16 then becomes

$$f\left(c_1, c_2, ..., c_N\right) = \sum_{i=1}^{N} \left[\frac{1}{2} c_i^2 \mathbf{p}_i^T \mathbf{A} \mathbf{p}_i - c_i \mathbf{p}_i^T \mathbf{b}\right] + c, \tag{3.18}$$

which is easily minimized; setting $\partial f / \partial c_i = 0;\ i = 1, ..., N$ gives

$$c_i = \frac{\mathbf{p}_i^T \mathbf{b}}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}; \quad i = 1, ..., N. \tag{3.19}$$

Vectors that obey Eq. 3.17 are said to be *conjugate*, and the basis set is called a *conjugate set*.

Details: One way to look at conjugation is as generalization of orthogonalization. Use 3D space as an example. Choose orhonormal unit vectors, $\overrightarrow{e}_i;\ i = 1, 2, 3$, with $\overrightarrow{e}_i \cdot \overrightarrow{e}_j = \delta_{ij}$. Then two vectors $\overrightarrow{x} = \sum_{i=1}^{3} x_i \overrightarrow{e}_i$ and $\overrightarrow{y} = \sum_{j=1}^{3} y_j \overrightarrow{e}_j$ are orthogonal if $\overrightarrow{x} \cdot \overrightarrow{y} = \sum_{i=1}^{3} \sum_{j=1}^{3} x_i y_j \overrightarrow{e}_i \cdot \overrightarrow{e}_j = \sum_{i=1}^{3} x_i y_j = 0$. In row/column vector notation, we write $\mathbf{x}^T \mathbf{y} = 0$. Now pick some arbitrary set of linear independent vectors $\overrightarrow{a}_i;\ i = 1, 2, 3$, not necessarily orthogonal, nor normalized (the vectors describing a unit cell of a triclinic crystal, for instance), and write $\overrightarrow{x} = \sum_{i=1}^{3} x_i' \overrightarrow{a}_i$ and $\overrightarrow{y} = \sum_{j=1}^{3} y_j' \overrightarrow{a}_j$. Then $\overrightarrow{x} \cdot \overrightarrow{y} = 0$ means $\sum_{i=1}^{3} \sum_{j=1}^{3} x_i' y_j' \overrightarrow{a}_i \cdot \overrightarrow{a}_j = 0$. Defining matrix elements $A_{ij} = \overrightarrow{a}_i \cdot \overrightarrow{a}_j;\ i = 1, 2, 3;\ j = 1, 2, 3$, we can form the positive definite symmetric matrix $\mathbf{A}$ (see also the lecture notes on (non-)linear problems; Cholesky decomposition). Then write $\overrightarrow{x} \cdot \overrightarrow{y} = 0$ in row/column vector notation as $\mathbf{x}'^T \mathbf{A} \mathbf{y}' = 0$. In other words, the row/column vectors $\mathbf{x}'$ and $\mathbf{y}'$ are *conjugate*. In physics, such a positive definite matrix $\mathbf{A}$ is often called a *metric*.

3.3.1.1 Conjugate search directions

With a conjugate set, the minimization problem, Eq. 3.1, becomes a sum of N independent minimization problems, Eq. 3.3, one for each coefficient c_i. We have the solution, Eq. 3.4, but it is instructive to present it in the form of a search algorithm similar to Eq. 3.11. Start with a point $\mathbf{x}_1$ and a search direction $\mathbf{p}_1$; then find α_1, such that $f(\mathbf{x}_1 - \alpha_1 \mathbf{p}_1)$ is minimal. Similar to the analysis leading to Eq. 3.10, this gives $\alpha_1 = \mathbf{p}_1^T \mathbf{r}_1 / \mathbf{p}_1^T \mathbf{A} \mathbf{p}_1$, with $\mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - \mathbf{b}$ the residual vector. Note that

$$\mathbf{x}_2 = \mathbf{x}_1 - \alpha_1 \mathbf{p}_1 = \mathbf{x}_1 - \left(\mathbf{p}_1^T \mathbf{A} \mathbf{x}_1 / \mathbf{p}_1^T \mathbf{A} \mathbf{p}_1\right) \mathbf{p}_1 + c_1 \mathbf{p}_1, \tag{3.20}$$

with c_1 given by Eq. 3.19. We proceed from point $\mathbf{x}_2$ and a search direction $\mathbf{p}_2$, and then find α_2, such that $f(\mathbf{x}_2 - \alpha_2 \mathbf{p}_2)$ is minimal

$$\mathbf{x}_3 = \mathbf{x}_2 - \alpha_2 \mathbf{p}_2 = \mathbf{x}_1 - \left(\mathbf{p}_1^T \mathbf{A} \mathbf{x}_1 / \mathbf{p}_1^T \mathbf{A} \mathbf{p}_1\right) \mathbf{p}_1 - \left(\mathbf{p}_2^T \mathbf{A} \mathbf{x}_1 / \mathbf{p}_2^T \mathbf{A} \mathbf{p}_2\right) \mathbf{p}_2 + c_1 \mathbf{p}_1 + c_2 \mathbf{p}_2, \tag{3.21}$$

where I have made use of Eqs. 3.20, 3.19, and $\mathbf{p}_2^T \mathbf{A} \mathbf{p}_1 = 0$. After n steps of this procedure, we get

$$\mathbf{x}_{n+1} = \mathbf{x}_1 - \sum_{i=1}^{n} \left(x_i - c_i\right) \mathbf{p}_i \quad \text{with } x_i = \frac{\mathbf{p}_i^T \mathbf{A} \mathbf{x}_1}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}, \tag{3.22}$$

and c_i given by Eq. 3.19. After N steps we have

$$\mathbf{x}_{N+1} = \mathbf{x}_1 - \sum_{i=1}^{N} \left(x_i - c_i\right) \mathbf{p}_i = \sum_{i=1}^{N} c_i \mathbf{p}_i. \tag{3.23}$$

We have $\mathbf{x}_1 = \sum_{i=1}^{N} x_i \mathbf{p}_i$, because the latter is just the representation of $\mathbf{x}_1$ on the basis $\mathbf{p}_i;\ i = 1, ..., N$.

Proof: write $\mathbf{x} = \sum_{j=1}^{N} x_j \mathbf{p}_j$, which is possible, as $\mathbf{p}_j;\ j = 1, ..., N$ is a basis set. Then $\mathbf{p}_i^T \mathbf{A}\mathbf{x} = \sum_{j=1}^{N} x_j \mathbf{p}_i^T \mathbf{A}\mathbf{p}_j = x_i \mathbf{p}_i^T \mathbf{A}\mathbf{p}_i$, as it's linear algebra, and Eq. 3.17 holds. This gives $x_i = \left(\mathbf{p}_i^T \mathbf{A}\mathbf{x}\right) / \left(\mathbf{p}_i^T \mathbf{A}\mathbf{p}_i\right);\ i = 1, ..., N$. Incidentally, this also proves that $\sum_{i=1}^{n} x_i \mathbf{p}_i$ appearing in Eq. 3.22 is the projection of $\mathbf{x}_1$ on the space spanned by $\mathbf{p}_i;\ i = 1, ..., N$.

From Eq. 3.23 one notices that we find the exact answer in N steps. In other words, using conjugate vectors we have constructed a direct algorithm, where the number of iterations is bounded. A look at Eqs. 3.20 and 3.21 shows you why this is. The first step along the search direction $\mathbf{p}_1$ immediately gives you optimal coefficient c_1 as it is needed in the final solution. The second step along $\mathbf{p}_2$ finds the optimal coefficient c_2, and each subsequent step in the direction $\mathbf{p}_i$ finds the optimal coefficient c_i. One never has to revisit previous search directions $\mathbf{p}_j;\ j = 1, ..., i-1$ ever again. This is in stark contrast to the steepest descent algorithm, as the staircase path keeps alternating between the same search directions, see Fig. 3.1. Using conjugate directions in this 2D example ($N = 2$), we find the minimum in two steps, see Fig. 3.2.

The *conjugate search directions* algorithm is given by

$$\begin{aligned} &\mathbf{x}_1 = \mathbf{v};\quad \mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - \mathbf{b}; \\ &\text{do}\quad n = 1, ..., N\quad \alpha_n = \frac{\mathbf{p}_n^T \mathbf{r}_n}{\mathbf{p}_n^T \mathbf{A}\mathbf{p}_n};\ \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \mathbf{p}_n;\ \mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \mathbf{b}; \end{aligned} \tag{3.24}$$

As in Eqs. 3.11 and 3.12, one can reduce the number of matrix-vector multiplications per iteration in Eq. 3.24 by writing

$$\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha_n \mathbf{A}\mathbf{p}_n. \tag{3.25}$$

This means that per iteration one only requires the matrix-vector product $\mathbf{A}\mathbf{p}_n$.

3.3.1.2 Construct a conjugate basis

Of course, the key component for this algorithm to work, is the conjugate set $\mathbf{p}_i;\ i = 1, ..., N$. In principle, we can construct a conjugate set starting from any, arbitrary, basis set $\mathbf{s}_i;\ i = 1, ..., N$, by applying a modified form of Gram-Schmidt orthogonalization. I copy the algorithm from the lecture notes on (non-)linear problems

$$\begin{aligned} &\text{do } j = 1, ..., N\quad \mathbf{p}_j = \mathbf{s}_j; \\ &\text{do } j = 2, ..., N\quad \text{do } k = 1, ..., j-1\quad \mathbf{p}_j = \mathbf{p}_j - \mathbf{p}_k \left(\mathbf{p}_k^T \mathbf{A}\mathbf{p}_j\right) / \left(\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k\right); \end{aligned} \tag{3.26}$$

It works similar to ordinary Gram-Schmidt. At the end of the procedure, the vectors $\mathbf{p}_i;\ i = 1, ..., N$ form a conjugate set, Eq. 3.17.[13] Let us call this algorithm *Gram-Schmidt conjugation*.

[13] For $\mathbf{A} = \mathbf{I}$ (the identity matrix) we retrieve the ordinary Gram-Schmidt routine. The resulting vectors are then orthogonal.

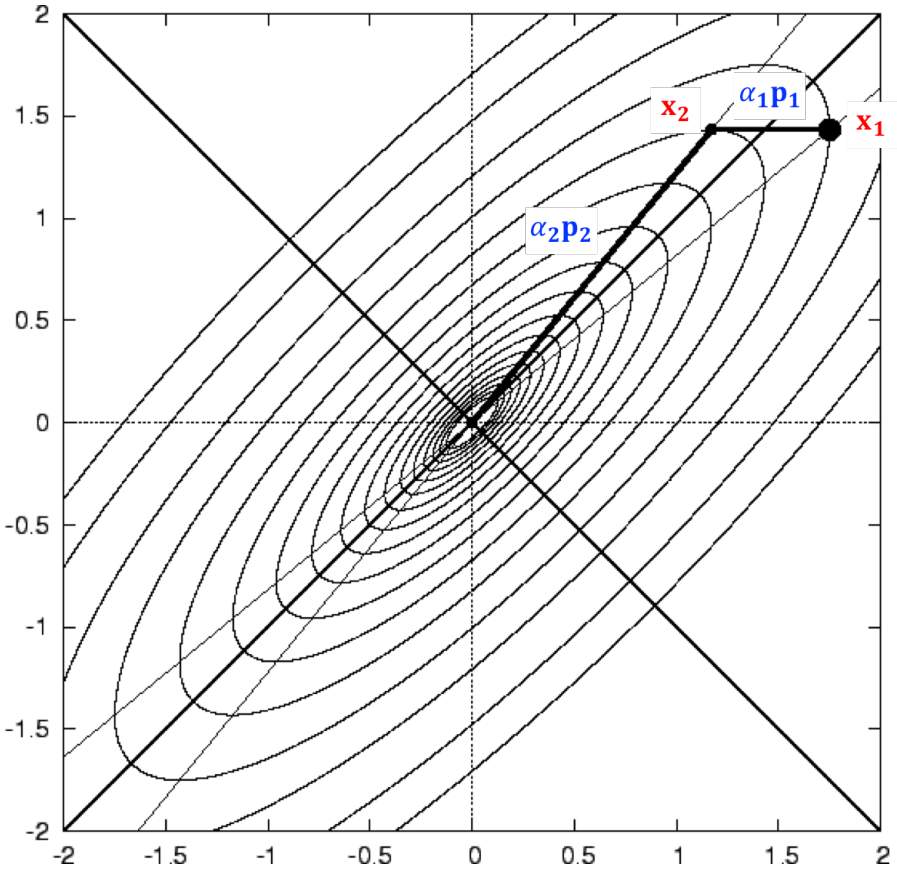


Figure 3.2: The method of conjugate search directions, Eq. 3.24, presented graphically in two dimensions. The ellipses represent a contour plot of a two-dimensional function $f(\mathbf{x})$. The thick solid line represents the path generated by conjugate search directions, Eq. 3.17. Each straight line piece begins of at a point $\mathbf{x}_n$, and ends at a point $\mathbf{x}_{n+1}$, with a direction $\mathbf{p}_n$ and a step parameter α_n. In two steps we reach the minimum.

Proof: start from a vector $\mathbf{p}_1 = \mathbf{q}_1$, and subtract the projection $\mathbf{p}_1 \left(\mathbf{p}_1^T \mathbf{A} \mathbf{q}_j\right) / \left(\mathbf{p}_1^T \mathbf{p}_1\right)$ from all vectors $\mathbf{q}_j$; $j = 2, ..., N$.

$$\mathbf{q}'_j = \mathbf{q}_j - \mathbf{p}_1 \left(\mathbf{p}_1^T \mathbf{A} \mathbf{q}_j\right) / \left(\mathbf{p}_1^T \mathbf{A} \mathbf{p}_1\right) \quad \text{to ensure} \quad \mathbf{p}_1^T \mathbf{A} \mathbf{q}'_j = 0; \; j = 2, ..., N.$$

Then set $\mathbf{p}_2 = \mathbf{q}'_2$, and subtract $\mathbf{p}_2 \left(\mathbf{p}_2^T \mathbf{A} \mathbf{q}'_j\right) / \left(\mathbf{p}_2^T \mathbf{p}_2\right)$ from all remaining vectors $\mathbf{q}'_j$; $j = 3, ..., N$.

$$\mathbf{q}''_j = \mathbf{q}'_j - \mathbf{p}_2 \left(\mathbf{p}_2^T \mathbf{A} \mathbf{q}'_j\right) / \left(\mathbf{p}_2^T \mathbf{A} \mathbf{p}_2\right) \quad \text{to ensure} \quad \mathbf{p}_2^T \mathbf{A} \mathbf{q}''_j = 0; \; j = 3, ..., N.$$

At the same time $\mathbf{p}_1^T \mathbf{A} \mathbf{q}''_j = \mathbf{p}_1^T \mathbf{A} \mathbf{q}'_j - \mathbf{p}_1^T \mathbf{A} \mathbf{p}_2 \left(\mathbf{p}_2^T \mathbf{A} \mathbf{q}'_j\right) / \left(\mathbf{p}_2^T \mathbf{A} \mathbf{p}_2\right) = 0$, as $\mathbf{p}_1^T \mathbf{A} \mathbf{q}'_j = 0$ and $\mathbf{p}_1^T \mathbf{A} \mathbf{p}_2 = 0$. Then set $\mathbf{p}_3 = \mathbf{q}''_3$, and repeat the process until we have defined $\mathbf{p}_N$. Along similar lines one could make a more formal proof on the basis of total induction.

Combining the algorithms of Eqs. 3.26 and 3.24 minimizes our function, Eq. 3.16. A straightforward combination, however, is not such a good idea. The nested do-loops in Eq. 3.26 requires $\frac{1}{2}N(N-1)$ iterations. In each iteration, inner products need to be calculated, which are $O(N)$. This means that the total cost of Eq. 3.26 scales as $O(N^3)$. A direct

method such as Cholesky decomposition also solves our quadratic minimization problem in $O(N^3)$ steps.

3.3.2 Conjugate gradients

In other words, it not wise to start from an arbitrary basis set $\mathbf{s}_i$; $i = 1, ..., N$. The brilliant idea is to use as a basis set the residuals $\mathbf{r}_n$; $n = 1, ..., N$ that are generated during the iterations of Eq. 3.24. These residuals have the property

$$\mathbf{p}_m^T \mathbf{r}_{n+1} = 0; \ m = 1, ..., n. \tag{3.27}$$

One can also prove that the residuals have the property

$$\mathbf{r}_m^T \mathbf{r}_{n+1} = 0; \ m = 1, ..., n. \tag{3.28}$$

Note that the statement of Eq. 3.28 is much stronger than the corresponding statement for the steepest descent algorithm, Eq. 3.13. Using conjugate search directions makes sure that a subsequent residual is orthogonal to *all* previous residuals, and that previous search spaces are not revisited. In steepest descent a subsequent residual is orthogonal *only* to the one residual just before, and it has no longer memory; hence a steepest descent search can meander.

One additional property of the residuals is very useful

$$\mathbf{p}_m^T \mathbf{A} \mathbf{r}_{n+1} = 0; \ m = 1, ..., n-1. \tag{3.29}$$

Proofs: Eq. 3.27, $\mathbf{p}_m^T \mathbf{r}_{n+1} = \mathbf{p}_m^T \mathbf{r}_n - \alpha_n \mathbf{p}_m^T \mathbf{A} \mathbf{p}_n$, using Eq. 3.25. Using Eq. 3.24 $\alpha_n = \left(\mathbf{p}_n^T \mathbf{r}_n\right) / \left(\mathbf{p}_n^T \mathbf{A} \mathbf{p}_n\right)$, then shows that $\mathbf{p}_n^T \mathbf{r}_{n+1} = 0$, which proves the case for $m = n$. For $m < n$, $\mathbf{p}_m^T \mathbf{A} \mathbf{p}_n = 0$, so $\mathbf{p}_m^T \mathbf{r}_{n+1} = \mathbf{p}_m^T \mathbf{r}_n$. Then use $\mathbf{p}_m^T \mathbf{r}_n = \mathbf{p}_m^T \mathbf{r}_{n-1} - \alpha_n \mathbf{p}_m^T \mathbf{A} \mathbf{p}_{n-1}$ to prove the case for $n = m - 1$, etcetera.

Eq. 3.28, Gram-Schmidt conjugation, Eq. 3.26, can be written as $\mathbf{p}_m = \mathbf{r}_m - \sum_{i=1}^{m-1} B_{mi} \mathbf{p}_i$, with $B_{mi} = \mathbf{r}_m^T \mathbf{p}_i$. This means $\mathbf{r}_m = \mathbf{p}_m + \sum_{i=1}^{m-1} B_{mi} \mathbf{p}_i$, and $\mathbf{r}_m^T \mathbf{r}_{n+1} = \mathbf{p}_m^T \mathbf{r}_{n+1} + \sum_{i=1}^{m-1} B_{mi} \mathbf{p}_i^T \mathbf{r}_{n+1} = 0$, because of Eq. 3.27.

Eq. 3.29, from Eq. 3.25 $\alpha_n \mathbf{p}_m^T \mathbf{A} \mathbf{r}_{n+1} = \mathbf{r}_m^T \mathbf{r}_{n+1} - \mathbf{r}_{m+1}^T \mathbf{r}_{n+1}$. Using Eq. 3.28 then proves Eq. 3.29.

The reason why this is useful, is because we can simplify the Gram-Schmidt conjugation of Eq. 3.26

$$\begin{aligned} &\mathbf{p}_1 = \mathbf{r}_1; \\ &\text{do } j = 1, ..., N-1 \quad \mathbf{p}_{j+1} = \mathbf{r}_{j+1} - \mathbf{p}_j \left(\mathbf{p}_j^T \mathbf{A} \mathbf{r}_{j+1}\right) / \left(\mathbf{p}_j^T \mathbf{A} \mathbf{p}_j\right); \end{aligned} \tag{3.30}$$

To construct $\mathbf{p}_{j+1}$ we only have to subtract the projection of $\mathbf{r}_{j+1}$ on $\mathbf{p}_j$, and not the projections of $\mathbf{r}_{j+1}$ on all $\mathbf{p}_m$; $m = 1, ..., j$, as in Eq. 3.26. This is because those projections are zero for $\mathbf{p}_m$; $m = 1, ..., j-1$, Eq. 3.29. This algorithm involves $N - 1$ matrix-vector multiplications, and is thus $O(N^3)$. Although as such this does not sound better than the the $O(N^3)$ of Eq. 3.26, it will be clear that Eq. 3.30 is cheaper.

We don't know the residuals $\mathbf{r}_n$ on forehand, but construct them on the fly, following the algorithm of Eq. 3.24. Therefore, we must also construct the conjugate vectors $\mathbf{p}_n$ on the fly,

and merge the two algorithms of Eqs. 3.24 and 3.30. This gives the world-famous *conjugate gradient* algorithm.

$$\mathbf{x}_1 = \mathbf{v}; \quad \mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - \mathbf{b}; \quad \mathbf{p}_1 = \mathbf{r}_1;$$

$$\text{do} \quad n = 1, ..., N \quad \alpha_n = \frac{\mathbf{p}_n^T \mathbf{r}_n}{\mathbf{p}_n^T \mathbf{A}\mathbf{p}_n}; \; \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \mathbf{p}_n; \; \mathbf{r}_{n+1} = \mathbf{r}_n - \alpha_n \mathbf{A}\mathbf{p}_n; \tag{3.31}$$

$$\beta_n = \frac{\mathbf{r}_{n+1}^T \mathbf{A}\mathbf{p}_n}{\mathbf{p}_n^T \mathbf{A}\mathbf{p}_n}; \; \mathbf{p}_{n+1} = \mathbf{r}_{n+1} - \beta_n \mathbf{p}_n; \tag{3.32}$$

Eq. 3.31 finds the minimum $\mathbf{x}_{n+1}$, starting from $\mathbf{x}_n$ and going along the search direction $\mathbf{p}_n$, according to Eq. 3.24. Then at this minimum the residual $\mathbf{r}_{n+1}$ is constructed according to Eq. 3.25. Eq. 3.32 constructs the new search direction $\mathbf{p}_{n+1}$ from $\mathbf{r}_{n+1}$ by making sure that it is conjugate to all previous search directions, according to Eq. 3.30.

Cost The algorithm needs one matrix-vector multiplication $\mathbf{A}\mathbf{p}_n$ per cycle, so the algorithm is $O(N^3)$. In other words, it solves the quadratic minimization problem, Eq. 3.1, in $O(N^3)$ steps. As the latter is equivalent to Eq. 3.4, conjugate gradient can also be viewed as a technique for solving a linear problem for the special case that the matrix $\mathbf{A}$ is symmetric and positive definite.[14] Like most famous algorithms, conjugate gradient is beautifully elegant.[15]

3.3.3 Convergence of conjugate gradient

If you are a skeptic (as a physicist, you should be of course), you might wonder what the big deal is with conjugate gradient. It is a $O(N^3)$ algorithm for solving the quadratic minimization problem. But, we already had other $O(N^3)$ algorithms perfectly suited for that task, such as Cholesky decomposition. The *main idea* is that conjugate gradient can deliver a *decent answer after* n cycles, where, with a bit of luck, $n \ll N$. If this is the case, we stop after n cycles, which makes conjugate gradient a $O(nN^2)$ algorithm. This is unlike techniques such as Cholesky decomposition, which have to be pursued right until the end ($n = N$) in order to get an answer.[16]

It is worth while to modify the *conjugate gradient algorithm* of Eqs. 3.31 and 3.32 into

[14] As such, it is an alternative to Cholesky decomposition.

[15] Put it on your top-10 list. Unfortunately, it is also a little bit of a misnomer. The gradients are not conjugate, they are orthogonal, see 3.28. The gradients are used to construct conjugate search directions, so the proper phrase would be *conjugated gradients*.

[16] Conjugate gradient has a second advantage. For some physical problems it is possible to calculate the matrix-vector product $\mathbf{A}\mathbf{v}$ much more efficiently than $O(N^2)$. Some quantum mechanical or signal transformation problems rely heavily on fast Fourier transforms, for instance, where such a product can be done much more efficiently, i.e., in $O(N \log N)$ steps. In those cases one neither has to construct the matrix $\mathbf{A}$ explicitly, nor store it, which is advantageous when dealing with large N. The computational and the storage advantages are lost in methods such as Cholesky decomposition.

$$\mathbf{x}_1 = \mathbf{v}; \quad \mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - \mathbf{b}; \quad \mathbf{p}_1 = \mathbf{r}_1;$$

$$\text{do} \quad n = 1, ..., \quad \alpha_n = \frac{\mathbf{p}_n^T \mathbf{r}_n}{\mathbf{p}_n^T \mathbf{A}\mathbf{p}_n}; \; \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \mathbf{p}_n; \; \mathbf{r}_{n+1} = \mathbf{r}_n - \alpha_n \mathbf{A}\mathbf{p}_n; \quad \text{until converged;} \tag{3.33}$$

$$\beta_n = \frac{\mathbf{r}_{n+1}^T \mathbf{A}\mathbf{p}_n}{\mathbf{p}_n^T \mathbf{A}\mathbf{p}_n}; \; \mathbf{p}_{n+1} = \mathbf{r}_{n+1} - \beta_n \mathbf{p}_n; \tag{3.34}$$

The "until converged" criterion can be something like $\|\mathbf{r}_{n+1}\| < \epsilon$, for instance, where ϵ is set by the user.

A full convergence analysis is too much for these notes, but I will give you some of its results [23]. Write $\mathbf{x}_n = \mathbf{x}^{\min} + \mathbf{e}_n$, with $\mathbf{e}_n$ the error vector. For the conjugate gradient algorithm one has

$$\|\mathbf{e}_{n+1}\|_{\mathbf{A}} \leq K \|\mathbf{e}_n\|_{\mathbf{A}} \quad \text{with } K = 2\left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right) \tag{3.35}$$

Here $\|\mathbf{x}\|_{\mathbf{A}} = \sqrt{\mathbf{x}^T \mathbf{A}\mathbf{x}}$ is the so-called $\mathbf{A}$-*norm*, and $\kappa \geq 1$ is the *condition number* of the matrix $\mathbf{A}$, see the lecture notes on (non-)linear problems. There it is also shown that for a positive definite symmetric matrix

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}, \tag{3.36}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the maximum and minimum eigenvalues of $\mathbf{A}$.[17]

If the condition number is large, $\kappa \gg 1$, then $K \approx 1$ in Eq. 3.35, and convergence is slow. However, if $\kappa = 1 + \delta$, with $\delta \ll 1$, then

$$K \approx \frac{\delta}{2} \ll 1, \tag{3.37}$$

and the conjugate gradient algorithm converges very fast. In order to achieve this, one must have

$$\delta = \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\min}} \ll 1. \tag{3.38}$$

This requires a matrix $\mathbf{A}$ with eigenvalues that are very narrowly spaced, i.e., a matrix with a *dense spectrum.*

The identity matrix $\mathbf{I}$ would be perfect. All its eigenvalues $\lambda_i = 1; \; i = 1, ..., N$, which gives $\delta = 0$, and perfect convergence. This is of course an unreasonable example, because we can write down the solution of a problem with $\mathbf{A} = \mathbf{I}$ immediately, without any calculation. However, it indicates that matrices $\mathbf{A}$ that are close to the identity matrix are close to perfect. The technique of preconditioning aims at getting a matrix that is closer to the identity matrix, in order to speed up the convergence of conjugate gradient. Many people in fact suggest never to use conjugate gradient without preconditioning.

3.3.4 Preconditioning

The condition for fast convergence of conjugate gradient, Eq. 3.38, looks like a purely mathematical result, i.e., perfectly true and perfectly useless at the same time. Typically

[17] All eigenvalues $\lambda_i; \; i = 1, ..., N$ of a positive definite symmetric matrix are of course positive.

we don't know the eigenvalues of a matrix $\mathbf{A}$. If we did, we could solve the minimization problem in an instant. Write $\mathbf{x} = \sum_{i=1}^{N} a_i \mathbf{z}_i$, with $\mathbf{A}\mathbf{z}_i = \lambda_i \mathbf{z}_i$; $i = 1, ..., N$ the eigenvectors of $\mathbf{A}$. Then Eq. 3.2 transforms into

$$f(a_1, a_2, ..., a_N) = \sum_{i=1}^{N} \left[\frac{1}{2} \lambda_i a_i^2 - a_i \mathbf{b}^T \mathbf{z}_i \right] + c, \tag{3.39}$$

and the minimization of this is trivial

$$a_i = \frac{\lambda_i}{\mathbf{b}^T \mathbf{z}_i}. \tag{3.40}$$

However, finding all eigenvalues and eigenvectors of a matrix is a bigger problem than minimization, so this route is not of much practical use.

The power of Eq. 3.38 is in its use as a guideline. Suppose we can come up with a matrix $\mathbf{M}$, such that $\kappa(\mathbf{MA}) \ll \kappa(\mathbf{A})$, and, instead of the original problem, Eq. 3.4, we solve the problem

$$\mathbf{MA}\mathbf{x}^{\min} = \mathbf{M}\mathbf{b}. \tag{3.41}$$

Applying the conjugate gradient algorithm to this problem, it should converge faster than applying it to the original problem, in view of the discussion in the previous section. Solving Eq. 3.41 instead of Eq. 3.4 is called *preconditioning*. One multiplies the matrix $\mathbf{A}$ with a matrix $\mathbf{M}$, such that $\mathbf{MA}$ is closer to the identity matrix $\mathbf{I}$ than the original matrix $\mathbf{A}$. To make that happen, the matrix $\mathbf{M}$ should be an *approximate inverse* of $\mathbf{A}$. The matrix $\mathbf{M}$ is called the *preconditioner*. The perfect $\mathbf{M}$ is $\mathbf{A}^{-1}$, as then $\mathbf{MA} = \mathbf{I}$, but $\mathbf{A}^{-1}$ is too difficult to calculate. Like in our discussion of iterative techniques for solving (non-)linear problems, we require an approximate inverse that can be easily calculated.

There is a catch. The matrix $\mathbf{MA}$ is in general not symmetric, nor positive definite, even if the matrices $\mathbf{M}$ and $\mathbf{A}$ are,[18] and we need a positive definite symmetric matrix in order to apply the conjugate gradient algorithm. The solution to this problem can be found in the lecture notes on eigenvalue problems, see the section on generalized eigenvalue problems. Start with $\mathbf{M}$ symmetric and positive definite, and write $\mathbf{M} = \mathbf{L}\mathbf{L}^T$, with $\mathbf{L}$ a lower triangular matrix (Cholesky decomposition). Then rewrite Eq. 3.41 as

$$\mathbf{A}'\mathbf{y}^{\min} = \mathbf{b}' \quad \text{with } \mathbf{A}' = \mathbf{L}^T\mathbf{A}\mathbf{L}, \quad \mathbf{b}' = \mathbf{L}^{-1}\mathbf{L}^T\mathbf{b} \quad \text{and } \mathbf{y}^{\min} = \mathbf{L}^{-1}\mathbf{x}^{\min}. \tag{3.42}$$

The matrix $\mathbf{A}'$ is symmetric and positive definite, so we can proceed with using conjugate gradient for the minimization problem. The conjugate gradient algorithm can actually be rewritten in such a way that we don't need to calculate the factors $\mathbf{L}$, $\mathbf{L}^T$, and $\mathbf{L}^{-1}$ explicitly. The knowledge of the matrix $\mathbf{M}$ is sufficient.

I won't go into the clever, but elaborate, preconditioners people have come up with over the years, but just consider the simplest scheme. Have a look at the $N \times N$ diagonal matrix

$$\mathbf{D} \quad \text{with } D_{ii} = a_{ii} \quad \text{and } D_{ij} = 0 \text{ for } i \neq j. \tag{3.43}$$

Then choose $\mathbf{M} = \mathbf{D}^{-1}$, with matrix elements $M_{ij} = \delta_{ij}/a_{ii}$; $i = 1, ..., N$; $j = 1, ..., N$. With a bit of luck $\kappa(\mathbf{D}^{-1}\mathbf{A}) \ll \kappa(\mathbf{A})$. In this case, the factorization is easy, $\mathbf{M} = \mathbf{L}\mathbf{L}^T$ gives $L_{ij} = \delta_{ij}/\sqrt{a_{ii}}$. The transformed problem of Eq. 3.42 becomes

$$\mathbf{A}'\mathbf{y}^{\min} = \mathbf{b}' \quad \text{with } A'_{ij} = \frac{a_{ij}}{\sqrt{a_{ii}a_{jj}}} \quad \text{and } b'_i = b_i; \; i = 1, ...N; \; j = 1, ...N. \tag{3.44}$$

[18] $(\mathbf{MA})^T = \mathbf{A}^T\mathbf{M}^T = \mathbf{AM} \neq \mathbf{MA}$ in general.

This form of preconditioning is called *Jacobi preconditioning*. Note that it is well-defined, as all diagonal elements $a_{ii} > 0;\ i = 1, ..., N$ for a positive definite matrix $\mathbf{A}$. The transformed matrix has diagonal elements $A'_{ii} = 1;\ i = 1, ..., N$. If $\left|A'_{ij}\right| \ll 1;\ i \neq j$, then $\mathbf{A}' \approx \mathbf{I}$, and as $\kappa(\mathbf{I}) = 1$, we have succeeded in making $\kappa(\mathbf{A}') < \kappa(\mathbf{A})$. The original parameters can be obtained from

$$x_i^{\min} = \frac{y_i^{\min}}{\sqrt{a_{ii}}}. \tag{3.45}$$

The rescaling operation of Eq. 3.44 is simple to apply. I don't know of practical cases where it really hurts, but it is of course far from guaranteed that it helps. If it doesn't, you should dig a bit deeper into the literature. Preconditioning is a bit of a black art to me.

3.4 The general optimization problem

Let us have the bad news first. There is currently *no algorithm* that finds you guaranteed the absolute minimum of any function $f(x_1, x_2, ..., x_N)$. The most popular algorithms start from the following.

Assume that the function is analytical, so all (higher) derivatives exist. Choose some point $x_i^{(0)};\ i = 1, ..., N$ as the origin, write $x_i = x_i^{(0)} + \Delta x_i;\ i = 1, ..., N$, and make a Taylor expansion

$$f(x_1, x_2, ..., x_N) \approx f_0 + \sum_{i=1}^{N} \left.\frac{\partial f}{\partial x_i}\right|_0 \Delta x_i + \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \left.\frac{\partial^2 f}{\partial x_i \partial x_j}\right|_0 \Delta x_i \Delta x_j, \tag{3.46}$$

with $f_0 = f(x_1^{(0)}, x_2^{(0)}, ..., x_N^{(0)})$, and the first and second derivatives also evaluated at the point $(x_1^{(0)}, x_2^{(0)}, ..., x_N^{(0)})$. If $\Delta x_i;\ i = 1, ..., N$ are sufficiently small, then terms in the Taylor expansion beyond the quadratic ones, may be neglected. The truncated expansion then makes the function look like a quadratic function, Eq. 3.1, and we can apply the methods developed in Secs. 3.2 and 3.3 for minimizing a quadratic problem.

How exactly this is implemented, will be discussed below. First some general stuff. As before, we collect the variables into a (column) vector $\mathbf{x}$, with elements $x_i;\ i = 1, ..., N$. The column vector

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_N} \end{pmatrix}, \tag{3.47}$$

is called the *gradient vector,* or simply the *gradient*. The matrix

$$\mathbf{H}(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_N} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_N \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_N \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_N \partial x_N} \end{pmatrix}, \tag{3.48}$$

is called the *Hessian matrix,* or simply the *Hessian*. Obviously, this matrix is symmetric $\mathbf{H}^T(\mathbf{x}) = \mathbf{H}(\mathbf{x})$. If the function of Eq. 3.46 is to have a minimum, then the Hessian $\mathbf{H}(\mathbf{x})$

needs to be positive definite, see the discussion in Sec. 3.1. Eq. 3.46 can be rewritten as

$$f(x_1, x_2, ..., x_N) \approx f_0 + \Delta\mathbf{x}^T \nabla f(\mathbf{x}_0) + \frac{1}{2}\Delta\mathbf{x}^T \mathbf{H}(\mathbf{x}_0)\Delta\mathbf{x}, \tag{3.49}$$

which is a quadratic problem, Eq. 3.2.

If both the gradient, Eq. 3.47, and the Hessian, Eq. 3.48, can be calculated easily, then the classical method for optimization is *Newton-Raphson*. I will discuss that in Sec. 3.4.3. Why not immediately? In most physics problems I have encountered, it is computationally *very expensive* to calculate the Hessian, up to, practically speaking, *impossible*. That rules out Newton-Raphson. In contrast, in many physics problems one can calculate the gradient within a reasonable computational cost. The methods we have developed above, steepest descent, Sec. 3.2, and conjugate gradient, Sec. 3.3, can be modified such, that they only need the gradient, and not the Hessian. The same is true for the quasi-Newton methods, which we will discuss in Sec. 3.5

To be complete, methods do exist for optimizing a function that use neither the gradient nor the Hessian, but only calculations of function values $f(x_1, x_2, ..., x_N)$. I will skip those; in my physics practice I have not encountered a problem where these methods can compete with the methods that use gradients.[19]

3.4.1 Steepest descent (SD)

Starting from Eq. 3.49, we apply the steepest descent algorithm, as developed in Sec. 3.2. As calculating the Hessian is very expensive, we would like to avoid that. The residual vector at the point $\mathbf{x}_{n+1}$, required by Eq. 3.11, can be expressed as the gradient at that point

$$\mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \mathbf{b} = \nabla f(\mathbf{x}_{n+1}). \tag{3.50}$$

This is proven straightforwardly from Eqs. 3.46 and 3.17. So, the residual only requires the gradient, not the Hessian.

The step parameter α_n needed in Eq. 3.11 poses a bigger problem, as it seems to require the calculation of $\mathbf{A}\mathbf{r}_n$. We can avoid this calculation by revisiting the derivation of Eq. 3.11. The step parameter α_n is the result of a one-parameter minimization

$$\alpha_n = \min_{\alpha} f\left(\mathbf{x}_n - \alpha\mathbf{r}_n\right). \tag{3.51}$$

This is called a *line search*. The expression $\mathbf{x}_n - \alpha\mathbf{r}_n$, with $\mathbf{x}_n$ and $\mathbf{r}_n$ fixed, and α variable, defines a line in N-dimensional space. Along this line we search for the minimum of the function f. In Sec. 3.6 I will show you how to do efficient line searches, using evaluations of function values and gradients only, without the need of a Hessian. For the moment we assume that we can do a line search efficiently.

With the modifications suggested by Eqs. 3.50 and 3.51, the *steepest descent* algorithm becomes

$$\begin{array}{l} \mathbf{x}_0 = \mathbf{v}; \quad \mathbf{r}_0 = \nabla f(\mathbf{x}_0); \\ \text{do} \quad n = 0, ... \quad \alpha_n = \min_{\alpha} f\left(\mathbf{x}_n - \alpha\mathbf{r}_n\right); \ \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n\mathbf{r}_n; \ \mathbf{r}_{n+1} = \nabla f(\mathbf{x}_{n+1}); \quad \text{until converged;} \end{array} \tag{3.52}$$

[19] For those who are interested: Ref. [7], Chapter 10.

The "until converged" criterion is again something of the type $\|\mathbf{r}_{n+1}\| < \epsilon$. In this form, the steepest descent algorithm is suited to find the minimum of any function $f(x_1, x_2, ..., x_N)$, not just a quadratic one. As you can see, it is a simple algorithm that uses the gradient to figure out in which direction to step to next, and how far.

Convergence and stability The steepest descent algorithm applied to a general function, Eq. 3.52, inherits all the shortcomings of steepest descent for quadratic functions, including the staircase effect discussed in Sec. 3.2.1. For a general function, there are some additional problems. Steepest descent searches for a point $\mathbf{x}^{\min}$ where the gradient $\nabla f(\mathbf{x}^{\min}) = 0$. This marks a stationary point of the function, which can be a minimum, a maximum, or a saddle point, and we only want the first one. The classical way to distinguish between the different types of stationary points, is to have a look at the Hessian, as a minimum is characterized by a positive definite Hessian. We are not willing to calculate the Hessian, so we do not have this tool.

Mathematicians seem to be worried about this problem, but physicists are more optimistic. Near a maximum or a saddle point, there is always a gradient directed away from the point. In an optimization problem it is typically difficult to reach such the point, as there is always a descent direction leading away from it. Physicists call maxima and saddle points *unstable*, and minima *stable* stationary points. In most cases, because of the numerical noise created by round-offs, for instance, the algorithm either never reaches an unstable point, or if it does, it falls off it again.[20] If you are nervous about this, perturb the point $\mathbf{x}^{\min} + \boldsymbol{\delta}$ in a random way, and optimize again.

As remarked before, the minimum that is found, is the one that is reached in direct descent from the starting point. There is no guarantee that this is the absolute minimum. There is no solution to this problem. The only thing you can do is to perturb the point $\mathbf{x}^{\min} + \boldsymbol{\delta}$, optimize again, and see where it brings you.

3.4.2 Conjugate gradient (CG)

We have presented conjugate gradient as a descent algorithm, where the descent directions are chosen more cleverly than in steepest descent. Once you have searched in one direction, you never have to revisit that direction again, unlike steepest descent, which can keep oscillating between directions, compare Figs. 3.1 and 3.2. Just as steepest descent can be generalized to any function, so should this also be possible for conjugate gradient. As for steepest descent, we want to do this without requiring calculations of the Hessian.

The residual $\mathbf{r}_{n+1}$ in Eq. 3.33 can be found from Eq. 3.50. As for steepest descent, the step parameter α_n can be determined from a line search

$$\alpha_n = \min_{\alpha} f\left(\mathbf{x}_n - \alpha \mathbf{p}_n\right), \tag{3.53}$$

where one can use one of the line search algorithms of Sec. 3.6. The parameter $\beta_n = \left(\mathbf{r}_{n+1}^T \mathbf{A}\mathbf{p}_n\right) / \left(\mathbf{p}_n^T \mathbf{A}\mathbf{p}_n\right)$ in Eq. 3.34, required to determine the new search direction, should also be reformulated. The denominator can be written as

$$\mathbf{p}_n^T \mathbf{A}\mathbf{p}_n = \mathbf{r}_n^T \mathbf{A}\mathbf{p}_n. \tag{3.54}$$

[20] The motto is: it is difficult to walk a tightrope, it is easy to fall to the ground.

Proof: $\mathbf{p}_n = \mathbf{r}_n - \beta_{n-1}\mathbf{p}_{n-1}$, which means $\mathbf{p}_n^T\mathbf{A}\mathbf{p}_n = \left(\mathbf{r}_n^T - \beta_{n-1}\mathbf{p}_{n-1}^T\right)\mathbf{A}\mathbf{p}_n = \mathbf{r}_n^T\mathbf{A}\mathbf{p}_n - \beta_{n-1}\mathbf{p}_{n-1}^T\mathbf{A}\mathbf{p}_n$, and $\mathbf{p}_{n-1}^T\mathbf{A}\mathbf{p}_n = 0$, Eq. 3.17.

Using Eq. 3.25 and 3.28, one then has

$$\beta_n = \frac{\mathbf{r}_{n+1}^T\mathbf{A}\mathbf{p}_n}{\mathbf{r}_n^T\mathbf{A}\mathbf{p}_n} = \frac{\mathbf{r}_{n+1}^T\left(\mathbf{r}_n - \mathbf{r}_{n+1}\right)}{\mathbf{r}_n^T\left(\mathbf{r}_n - \mathbf{r}_{n+1}\right)} = -\frac{\mathbf{r}_{n+1}^T\mathbf{r}_{n+1}}{\mathbf{r}_n^T\mathbf{r}_n}. \tag{3.55}$$

All of this leads to the following *conjugate gradient* algorithm, or *Fletcher-Reeves conjugate gradient*, after its inventors.

$$\mathbf{x}_1 = \mathbf{v};\quad \mathbf{r}_1 = \nabla f(\mathbf{x}_1);\quad \mathbf{p}_1 = \mathbf{r}_1;$$
$$\text{do}\quad n = 1, \ldots\quad \alpha_n = \min_{\alpha} f\left(\mathbf{x}_n - \alpha\mathbf{p}_n\right);\ \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n\mathbf{p}_n;\ \mathbf{r}_{n+1} = \nabla f(\mathbf{x}_n);\quad \text{until converged;} \tag{3.56}$$

$$\beta_n = \frac{\mathbf{r}_{n+1}^T\mathbf{r}_{n+1}}{\mathbf{r}_n^T\mathbf{r}_n};\ \mathbf{p}_{n+1} = \mathbf{r}_{n+1} + \beta_n\mathbf{p}_n; \tag{3.57}$$

The algorithm is suited to find the minimum of any function $f(x_1, x_2, ..., x_N)$, and it uses only function and gradient evaluations to do so. No calculations of the Hessian are required. The algorithm is also called *non-linear conjugate gradient* in the literature.[21] Note that the upper bound N in the do-loop is gone, compare Eqs. 3.56 and 3.33. The algorithm is only guaranteed to converge within N cycles for purely quadratic problems, not for a general function. In practice this is not much of a problem, as we want the number of cycles $n \ll N$ anyway, for the algorithm to be useful.

Conjugate gradient is one of the champion algorithms for minimizing functions, but it is not guaranteed to work for any function that has a minimum. That is because it is derived from a quadratic expansion, Eq. 3.46. This expansion is certainly valid if you are sufficiently close to the minimum, but it might fail if you are far from the minimum. The latter can result in a "strange" search direction $\mathbf{p}_n$, which does not get you closer to the minimum. The step parameter α_n, obtained in the line search, gets then close to zero, which brings the algorithm to a stand-still.

Polak and Ribière suggested a simple modification to mend this problem. Instead of using β_n as in Eq. 3.57, they proposed to use

$$\beta_n = \frac{\mathbf{r}_{n+1}^T\left(\mathbf{r}_{n+1} - \mathbf{r}_n\right)}{\mathbf{r}_n^T\mathbf{r}_n}. \tag{3.58}$$

If conjugate gradient proceeds normally, then $\mathbf{r}_{n+1}^T\mathbf{r}_n = 0$, see Eq. 3.28, so there is no operational difference with Eq. 3.57. However, if the algorithm gets stuck and $\alpha_n \approx 0$, then $\mathbf{x}_{n+1} \approx \mathbf{x}_n$, meaning $\mathbf{r}_{n+1} \approx \mathbf{r}_n$, which results in $\beta_n \approx 0$. In Eq. 3.57 this then means $\mathbf{p}_{n+1} \approx \mathbf{r}_{n+1}$, i.e., the new search will be along the direction of the gradient. This is effectively a steepest descent step, which we know will always bring you closer to the minimum, if not always in the most efficient way. As the first step in conjugate gradient is a

[21] A bit of a misnomer, if you ask me. Non-quadratic conjugate gradient would have been a better term, as it is applied to find the minimum of non-quadratic functions, in contrast to the original conjugate gradient algorithm of Sec. 3.3.2.

steepest-descent step anyway ($\mathbf{p}_1 = \mathbf{r}_1$), one can also say that Eq. 3.58 forces the conjugate gradient algorithm to restart after a standstill. The algorithm that uses Eq. 3.58 is called *Polak-Ribière conjugate gradient*.

By no means all problems are solved this way, and several other modifications of conjugate gradient have been proposed. However, to my knowledge, Polak-Ribière and Fletcher-Reeves are the versions that are used most (in that order).

Convergence and stability Conjugate gradient suffers from some of the problems also encountered in steepest descent. It might reach stationary points that are not minima, and it finds a local minimum reachable by direct descent from the starting point, which is not necessarily the global minimum. The first problem turns out to be a rare event in practical calculations, and is therefore not much of a problem. There is no solution to the second problem, so it is also not much of a problem. The algorithm inherits most of the good properties of conjugate gradient for quadratic problems, making it one of the best we have for minimizing functions.

In fact, conjugate gradient has become so popular that it is also applied to solve other types of problems. Sec. 3.7 discusses the use of conjugate gradient for solving linear and non-linear problems, and in Sec. 3.8 we apply conjugate gradient to solve eigenvalue problems.

3.4.3 Newton-Raphson

The quadratic optimization problem, Eqs. 3.1-3.4, is formally solved in one step by $\mathbf{x}^{\min} = \mathbf{A}^{-1}\mathbf{b}$. Similarly, the general optimization problem in the second order Taylor approximation, Eqs. 3.46 and 3.49, is formally solved in one step by $\Delta\mathbf{x}^{\min} = -\mathbf{H}(\mathbf{x}_0)^{-1}\nabla f(\mathbf{x}_0)$. Writing $\Delta\mathbf{x}^{\min} = \mathbf{x}^{\min} - \mathbf{x}_0$, this gives

$$\mathbf{x}^{\min} = \mathbf{x}_0 - \mathbf{H}(\mathbf{x}_0)^{-1}\nabla f(\mathbf{x}_0). \tag{3.59}$$

For a general function $f(x_1, x_2, ..., x_N)$, the second order Taylor expansion, Eq. 3.46, is of course just an approximation, but we can deduce from Eq. 3.59 a propagation rule, which leads to the algorithm

$$\mathbf{x}_1 = \mathbf{v}; \quad \text{do} \quad n = 1, ... \quad \mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}(\mathbf{x}_n)^{-1}\nabla f(\mathbf{x}_n); \quad \text{until converged;} \tag{3.60}$$

This is the *Newton-Raphson* algorithm. It proceeds by making a second order Taylor expansion of the function $f(\mathbf{x})$ around the point $\mathbf{x}_n$, calculating the new point $\mathbf{x}_{n+1}$ from this expansion, making a new Taylor expansion around $\mathbf{x}_{n+1}$, etcetera, until converged. The "until converged" criterion is then something like $\|\nabla f(\mathbf{x}_n)\| < \epsilon$.

The Newton-Raphson algorithm works fine, as long as the second order Taylor expansion is sufficiently accurate to represent the true function $f(\mathbf{x})$ around a point $\mathbf{x}_n$. We know that this Taylor expansion is sufficiently accurate, as long as we stay sufficiently close to $\mathbf{x}_n$. However, Eq. 3.60 does not really give us control over the step size $\Delta\mathbf{x} = \mathbf{x}_{n+1} - \mathbf{x}_n$. A safer approach is the following

$$\mathbf{x}_1 = \mathbf{v}; \quad \mathbf{g}_1 = \nabla f(\mathbf{x}_1);$$
$$\text{do} \quad n = 1, ... \quad \mathbf{p}_n = \mathbf{H}(\mathbf{x}_n)^{-1}\mathbf{g}_n; \tag{3.61}$$
$$\alpha_n = \min_{\alpha} f\left(\mathbf{x}_n - \alpha\mathbf{p}_n\right); \quad \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n\mathbf{p}_n; \quad ; \mathbf{g}_{n+1} = \nabla f(\mathbf{x}_{n+1}) \quad \text{until converged;} \tag{3.62}$$

This is the *Newton-Raphson* algorithm as it is usually applied. The Taylor expansion is used to determine a search direction $\mathbf{p}_n$. Subsequently the function f is minimized along this direction using one of the line search algorithms of Sec. 3.6.

Convergence Newton-Raphson behaves well in cases you would expect it to behave well: around the point $\mathbf{x}_n$ there should be a sufficiently large region where the second order Taylor expansion is a good approximation to the real function. This is guaranteed if $\mathbf{x}_n$ is sufficiently close to the minimum $\mathbf{x}^{\min}$. However, for $\mathbf{x}_n$ far from $\mathbf{x}^{\min}$, Newton-Raphson may be very inaccurate and might wander off in the wrong direction. Even in cases where one can afford to calculate the Hessian, it is probably better to do a couple of steepest-descent or conjugate-gradient steps, until you arrive at a point $\mathbf{x}_n$ from where it is reasonable to start Newton-Raphson.

Cost Newton-Raphson is a pretty costly algorithm, because it needs the Hessian, Eq. 3.48. For most of the functions $f(x_1, x_2, ..., x_N)$ encountered in physical problems, calculating the Hessian $\mathbf{H}$ is too expensive. On rare occasions, analytical expressions of the second derivatives $\partial^2 f/\partial x_i \partial x_j$ are available, making it possible to calculate the Hessian $\mathbf{H}$ relatively cheap. The next stumbling block is then to calculate the inverse $\mathbf{H}^{-1}$ for Eq. 3.61. Of course, one would never do this explicitly, but rather solve the set of linear equations for $\mathbf{p}_n$

$$\mathbf{H}(\mathbf{x}_n)\mathbf{p}_n = \nabla f(\mathbf{x}_n). \tag{3.63}$$

Using one of the standard algorithms (LU decomposition, for instance), this is still a formidable task, which scales as $O(N^3)$. This makes the overall scaling of Newton-Raphson $O(nN^3)$, with n the number of cycles until convergence. That does not sound too attractive, except for smallish problems.

3.5 Quasi-Newton methods

Whereas Newton-Raphson is seldom employed, because it is too expensive, there are important techniques based upon it that are frequently used. These go under the name *quasi-Newton methods*. Besides conjugate gradient, the quasi-Newton methods are another class of popular and wide-spread optimization techniques [20, 21, 22]. The *basic idea* is not to calculate the Hessian $\mathbf{H}$ explicitly, but to *approximate* it by finite difference expressions.

Suppose one has two points, $\mathbf{x}_n$ and $\mathbf{x}_{n+1}$, with corresponding gradients, $\nabla f(\mathbf{x}_n)$ and $\nabla f(\mathbf{x}_{n+1})$. We can Taylor approximate

$$\nabla f(\mathbf{x}_n) \approx \nabla f(\mathbf{x}_{n+1}) + \mathbf{H}(\mathbf{x}_{n+1})\left(\mathbf{x}_n - \mathbf{x}_{n+1}\right), \tag{3.64}$$

which can be used to define an approximate Hessian by

$$\mathbf{H}_{n+1}\Delta\mathbf{x}_n = \nabla f_n \quad \text{with } \Delta\mathbf{x}_n = \mathbf{x}_{n+1} - \mathbf{x}_n \quad \text{and } \nabla f_n = \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n) \tag{3.65}$$

Mathematicians call this the *secant equation*, and call quasi-Newton methods *secant methods*.[22] The various quasi-Newton methods vary in the way this secant equation is used to find an update for the approximate Hessian.

[22] Some of them seem to hate physicists so much, that they even refuse to refer to the man who invented calculus for them.

Broyden methods are the oldest, Sec. 3.5.1, and today are used mainly to solve non-linear equations, Sec. 3.7.[23] A further development, *Broyden-Fletcher-Goldfarb-Shanno* (BFGS), Sec. 3.5.2, is especially suited to solve optimization problems. Besides conjugate gradient, BFGS is one of the most popular techniques around for this purpose. The original Broyden techniques, including BFGS, keep the approximate Hessian matrix or its inverse in memory, and apply matrix-vector multiplications of the type as in Eq. 3.65. One can avoid these $O(N^2)$ operations, and the N^2 memory requirement, by using *recursive* BFGS, Sec. 3.5.3. This is one of the most efficient optimization algorithms around, in particular the variant called *limited-memory* BFGS, or L-BFGS. Last, but not least, our old friend *Pulay iteration* can be reformulated as a Broyden technique, Sec. 3.5.4.

3.5.1 Broyden methods

In one dimension, the secant equation, Eq. 3.65, leads to the expression

$$H_{n+1} = \frac{\frac{d}{dx}f(x_{n+1}) - \frac{d}{dx}f(x_n)}{x_{n+1} - x_n}, \tag{3.66}$$

which indeed is a finite-difference expression for the second derivative.[24] In $N > 1$ dimensions, the secant equation does not fix all second derivatives. One would need N different points $\mathbf{x}_{n+1}$ to do that. As one would need to do that for each cycle n, this is of course way too much work.

3.5.1.1 The good

Let us see what one can do with just one point $\mathbf{x}_{n+1}$. Assume that we have some approximation of the Hessian $\mathbf{H}_n$ obtained from the previous iterations, and we want to find a better approximation $\mathbf{H}_{n+1}$. Broyden's ansatz is to write it as

$$\mathbf{H}_{n+1} = \mathbf{H}_n + \mathbf{u}\,(\Delta\mathbf{x}_n)^T. \tag{3.67}$$

The added term makes the projection onto $\Delta\mathbf{x}_n$, and multiplies the result with the vector $\mathbf{u}$. We force this expression to obey the secant equation, Eq. 3.65.

$$\mathbf{H}_{n+1}\Delta\mathbf{x}_n = \mathbf{H}_n\Delta\mathbf{x}_n + \mathbf{u}\,\|\Delta\mathbf{x}_n\|^2 = \nabla f_n, \tag{3.68}$$

from which we obtain $\mathbf{u}$ as

$$\mathbf{u} = \frac{\nabla f_n - \mathbf{H}_n\Delta\mathbf{x}_n}{\|\Delta\mathbf{x}_n\|^2}. \tag{3.69}$$

Used in Eq. 3.67, this gives

$$\mathbf{H}_{n+1} = \mathbf{H}_n + \frac{\nabla f_n - \mathbf{H}_n\Delta\mathbf{x}_n}{\|\Delta\mathbf{x}_n\|^2}\,(\Delta\mathbf{x}_n)^T. \tag{3.70}$$

Note that $\mathbf{H}_{n+1}$ only updates $\mathbf{H}_n$ in the direction of $\Delta\mathbf{x}_n$. For any orthogonal direction $\mathbf{y}$, $(\Delta\mathbf{x}_n)^T\mathbf{y} = 0$, one has $\mathbf{H}_{n+1}\mathbf{y} = \mathbf{H}_n\mathbf{y}$, which means the approximate Hessian is *not* updated

[23] I hate to say this, but also Broyden was originally trained as a physicist.

[24] Apparently, a secant (a finite difference) technique was already used by the Babylonians to find a point where a function is zero. This shows you to what length mathematicians go to prove that they were first. Of course, those Babylonians have built bridges and cities, i.e., they were engineers.

in orthogonal directions. The update of Eq. 3.70 thus is a *one-dimensional* update of the Hessian in the direction of $\Delta\mathbf{x}_n$. The mathematicians call this a *rank-one* update. This is *the Broyden method* in its original form.

What we actually need in a Newton-Raphson-like algorithm is not the Hessian itself, but its inverse, see Eq. 3.61. The inverse of a matrix of the type of Eq. 3.67 is easily found, following the so-called *Sherman-Morrison formula*

$$\left(\mathbf{B}+\mathbf{v}\mathbf{w}^T\right)^{-1}=\mathbf{B}^{-1}-\frac{\left(\mathbf{B}^{-1}\mathbf{v}\right)\left(\mathbf{B}^{-1}\mathbf{w}\right)^T}{1+\mathbf{w}^T\mathbf{B}^{-1}\mathbf{v}}. \tag{3.71}$$

Proof: write $\left(\mathbf{B}+\mathbf{v}\mathbf{w}^T\right)^{-1}=\left[\mathbf{B}\left(\mathbf{I}+\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T\right)^{-1}\right]=\left(\mathbf{I}+\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T\right)^{-1}\mathbf{B}^{-1}$. Taylor expand $\left(\mathbf{I}+\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T\right)^{-1}=\mathbf{I}-\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T+\left(\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T\right)^2-\left(\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T\right)^3+...$, which gives $\mathbf{I}-\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T+\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T-\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T\mathbf{B}^{-1}\mathbf{v}\mathbf{w}^T+....$

Write this as $\mathbf{I}-\mathbf{B}^{-1}\mathbf{v}\left(1-\mathbf{w}^T\mathbf{B}^{-1}\mathbf{v}+\left(\mathbf{w}^T\mathbf{B}^{-1}\mathbf{v}\right)^2-...\right)\mathbf{w}^T=\mathbf{I}-\mathbf{B}^{-1}\mathbf{v}\left(1+\mathbf{w}^T\mathbf{B}^{-1}\mathbf{v}\right)^{-1}\mathbf{w}^T$. Multiplying this from the right with $\mathbf{B}^{-1}$ gives Eq. 3.71.

Applying Eq. 3.71 to Eq. 3.70 gives

$$\mathbf{H}_{n+1}^{-1}=\mathbf{H}_n^{-1}+\frac{\left(\Delta\mathbf{x}_n-\mathbf{H}_n^{-1}\nabla f_n\right)}{\Delta\mathbf{x}_n^T\mathbf{H}_n^{-1}\nabla f_n}\left(\mathbf{H}_n^{-1}\Delta\mathbf{x}_n\right)^T. \tag{3.72}$$

Calculation of $\mathbf{H}_{n+1}^{-1}$ is $O\left(N^2\right)$, provided we already have $\mathbf{H}_n^{-1}$, as essentially one only has to calculate the two matrix-vector products $\mathbf{H}_n^{-1}\nabla f_n$ and $\mathbf{H}_n^{-1}\Delta\mathbf{x}_n$. Over time many variations of Broyden's scheme have been formulated, some of which we will see below. To distinguish Eq. 3.72 of from other, similar, expressions, this particular the update of the inverse Hessian is called the *good Broyden method*, or the *first Broyden method*.

Using this approximation in the Newton-Raphson algorithm of Eqs. 3.61 and 3.62, gives a quasi-Newton algorithm called the *first Broyden method*

$$\mathbf{x}_1=\mathbf{v};\quad \mathbf{N}_1=\mathbf{I};\quad \mathbf{g}_1=\nabla f(\mathbf{x}_1);\quad \mathbf{p}_1=\mathbf{g}_1; \tag{3.73}$$

$$\text{do}\quad n=1,...\quad \alpha_n=\min_\alpha f\left(\mathbf{x}_n-\alpha\mathbf{p}_n\right);\ \mathbf{x}_{n+1}=\mathbf{x}_n-\alpha_n\mathbf{p}_n; \tag{3.74}$$

$$\mathbf{g}_{n+1}=\nabla f(\mathbf{x}_{n+1});\quad \text{until converged}; \tag{3.75}$$

$$\mathbf{d}_n=\mathbf{x}_{n+1}-\mathbf{x}_n;\quad \mathbf{h}_n=\mathbf{g}_{n+1}-\mathbf{g}_n;$$

$$\mathbf{N}_{n+1}=\mathbf{N}_n+\left(\mathbf{d}_n^T\mathbf{N}_n\mathbf{h}_n\right)^{-1}\left(\mathbf{d}_n-\mathbf{N}_n\mathbf{h}_n\right)\left(\mathbf{N}_n\mathbf{d}_n\right)^T; \tag{3.76}$$

$$\mathbf{p}_{n+1}=\mathbf{N}_{n+1}\mathbf{g}_{n+1}; \tag{3.77}$$

Eq. 3.73 initializes the inverse Hessian to the identity matrix $\mathbf{I}$, which makes the first step ($n=1$) in Eq. 3.74 a steepest-descent step. The "until converged" criterion in Eq. 3.75 is something like $\|\mathbf{g}_{n+1}\|<\epsilon$. The Broyden update of the inverse Hessian is done in Eq. 3.76, following the update rule of Eq. 3.72. While iterating, one gathers information about more and more points $\mathbf{x}_n$, allowing one to construct a better and better approximation to the inverse Hessian. Finally, in Eq. 3.77 one obtains the new search direction $\mathbf{p}_{n+1}$, according to the Newton-Raphson rule, Eq. 3.61.

3.5.1.2 The bad

The Broyden update, of Eq. 3.70, is by no means the only way to update the Hessian such, that it obeys the secant equation, 3.65. Rewrite the secant equation as

$$\mathbf{H}_{n+1}^{-1}\nabla f_n = \Delta\mathbf{x}_n. \tag{3.78}$$

This looks like Eq. 3.65, but with the roles of $\Delta\mathbf{x}_n$ and ∇f_n interchanged, and $\mathbf{H}_{n+1}$ replaced by $\mathbf{H}_{n+1}^{-1}$. In the spirit of Broyden's ansatz, Eq. 3.67, we write a rank-one update as

$$\mathbf{H}_{n+1}^{-1} = \mathbf{H}_n^{-1} + \mathbf{v}\left(\nabla f_n\right)^T. \tag{3.79}$$

Assuming that we know $\mathbf{H}_n^{-1}$, we find an expression of $\mathbf{v}$, such that Eq. 3.78 is obeyed

$$\mathbf{v} = \frac{\Delta\mathbf{x}_n - \mathbf{H}_n^{-1}\nabla f_n}{\|\nabla f_n\|^2}, \tag{3.80}$$

which gives the update as

$$\mathbf{H}_{n+1}^{-1} = \mathbf{H}_n^{-1} + \frac{\Delta\mathbf{x}_n - \mathbf{H}_n^{-1}\nabla f_n}{\|\nabla f_n\|^2}\left(\nabla f_n\right)^T. \tag{3.81}$$

This update is called the *second Broyden method* or the *bad Broyden method*.

Here is the algorithm

$$\mathbf{x}_1 = \mathbf{v};\quad \mathbf{N}_1 = \mathbf{I};\quad \mathbf{g}_1 = \nabla f(\mathbf{x}_1);\quad \mathbf{p}_1 = \mathbf{g}_1; \tag{3.82}$$

$$\text{do}\quad n = 1, \ldots\quad \alpha_n = \min_\alpha f\left(\mathbf{x}_n - \alpha\mathbf{p}_n\right);\ \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n\mathbf{p}_n; \tag{3.83}$$

$$\mathbf{g}_{n+1} = \nabla f(\mathbf{x}_{n+1});\quad \text{until converged;} \tag{3.84}$$

$$\mathbf{d}_n = \mathbf{x}_{n+1} - \mathbf{x}_n;\quad \mathbf{h}_n = \mathbf{g}_{n+1} - \mathbf{g}_n;$$

$$\mathbf{N}_{n+1} = \mathbf{N}_n + \left(\mathbf{h}_n^T\mathbf{h}_n\right)^{-1}\left(\mathbf{d}_n - \mathbf{N}_n\mathbf{h}_n\right)\left(\mathbf{h}_n\right)^T; \tag{3.85}$$

$$\mathbf{p}_{n+1} = \mathbf{N}_{n+1}\mathbf{g}_{n+1}; \tag{3.86}$$

Like in real life, the distinction between good and bad is not as clearcut as it sounds. Many people state that the good Broyden method works better than the bad Broyden method, but I have also heard the opposite view, so apparently it depends very much on the particular problem at hand. Both methods are applied in practical calculations with success (and also with failures). It is more neutral to call them the first and the second Broyden methods.

3.5.1.3 Good and bad*

It is possible to put the first and the second Broyden methods on the same footing. Both make use of the vector

$$\boldsymbol{\delta}_n = \Delta\mathbf{x}_n - \mathbf{H}_n^{-1}\nabla f_n, \tag{3.87}$$

which measures how well the previous (inverse) Hessian $\mathbf{H}_n$ obeys the secant equation, Eq. 3.78. They then make a rank-one (a one-dimensional) update of the type

$$\mathbf{H}_{n+1}^{-1} = \mathbf{H}_n^{-1} + \frac{\boldsymbol{\delta}_n\boldsymbol{\zeta}_n^T}{\left(\nabla f_n\right)^T\boldsymbol{\zeta}_n}. \tag{3.88}$$

This ensures that the secant equation, Eq. 3.78, is obeyed for any projection vector $\boldsymbol{\zeta}_n$, as

$$\mathbf{H}_{n+1}^{-1}\nabla f_n = \mathbf{H}_n^{-1}\nabla f_n + \boldsymbol{\delta}_n = \Delta\mathbf{x}_n. \tag{3.89}$$

In the good Broyden method one uses the projection vector $\boldsymbol{\zeta}_n = \mathbf{H}_n^{-1}\Delta\mathbf{x}_n$, Eq. 3.72, whereas in the bad Broyden method $\boldsymbol{\zeta}_n = \nabla f_n$ is used, Eq. 3.81. It will be clear that other choices for the projection vector $\boldsymbol{\zeta}_n$ are possible, in principle, which basically lead to a similar scheme. In Sec. 3.5.4 I will show that Pulay iteration is in fact such a scheme.

3.5.2 Broyden-Fletcher-Goldfarb-Shanno (BFGS)

I have actually cheated a little. Newton-Raphson and quasi-Newton methods are primarily techniques for solving non-linear equations, see Sec. 3.7. The equations we solve here, are of the type $\nabla f(\mathbf{x}) = 0$. The solution to these equations correspond to a minimum of $f(\mathbf{x})$, but only if the Hessian $\mathbf{H}(\mathbf{x})$ is positive definite. Moreover, the Hessian is symmetric. In this respect Broyden updates, Eq. 3.88, look a bit weird, as they are not symmetric. This means that the approximate Hessian is *not symmetric*. Moreover, Broyden updates do not guarantee that the approximate Hessian stays positive definite. You could ignore this, as we are not interested in the approximate Hessian as such.

You can also think that it makes sense to keep the approximate Hessian positive definite and symmetric, as that might guide us smoother to a minimum. Several modifications of the Broyden method have been proposed to deal with this. One of the most successful, and most widely used, methods is due to *Broyden, Fletcher, Goldfarb, and Shanno*, and is therefore dubbed BFGS. Instead of a one-dimensional update as in Eq. 3.67, one tries a two-dimensional update (a rank-two update)

$$\mathbf{H}_{n+1} = \mathbf{H}_n + \chi\mathbf{u}\mathbf{u}^T + \eta\mathbf{v}\mathbf{v}^T. \tag{3.90}$$

The matrix $\mathbf{H}_{n+1}$ is obviously symmetric, if the matrix $\mathbf{H}_n$ is. The secant equation, Eq. 3.65, demands

$$\mathbf{H}_{n+1}\Delta\mathbf{x}_n = \mathbf{H}_n\Delta\mathbf{x}_n + \chi\mathbf{u}\mathbf{u}^T\Delta\mathbf{x}_n + \eta\mathbf{v}\mathbf{v}^T\Delta\mathbf{x}_n = \nabla f_n \tag{3.91}$$

A bit of pondering shows that a possibility is

$$\mathbf{u} = \nabla f_n;\ \chi = \frac{1}{(\nabla f_n)^T(\Delta\mathbf{x}_n)};\ \mathbf{v} = \mathbf{H}_n\Delta\mathbf{x}_n;\ \eta = -\frac{1}{(\mathbf{H}_n\Delta\mathbf{x}_n)^T(\Delta\mathbf{x}_n)}. \tag{3.92}$$

The BFGS update then becomes

$$\mathbf{H}_{n+1} = \mathbf{H}_n + \frac{(\nabla f_n)(\nabla f_n)^T}{(\nabla f_n)^T(\Delta\mathbf{x}_n)} - \frac{(\mathbf{H}_n\Delta\mathbf{x}_n)(\mathbf{H}_n\Delta\mathbf{x}_n)^T}{(\mathbf{H}_n\Delta\mathbf{x}_n)^T(\Delta\mathbf{x}_n)}. \tag{3.93}$$

It obeys the secant equation, Eq. 3.65. The matrix $\mathbf{H}_{n+1}$ is positive definite, if the matrix $\mathbf{H}_n$ is, and if

$$(\nabla f_n)^T(\Delta\mathbf{x}_n) > 0. \tag{3.94}$$

Proof: if $\mathbf{e}_i^T\mathbf{B}\mathbf{e}_i > 0$ for $\mathbf{e}_i$; $i = 1, ..., N$, an orthonormal set, then the matrix $\mathbf{B}$ is positive definite. Choose $\mathbf{e}_i$ the eigenstates of $\mathbf{H}_n$, i.e., $\mathbf{H}_n\mathbf{e}_i = \lambda_i\mathbf{e}_i$, where $\lambda_i > 0$; $i = 1, ..., N$, as $\mathbf{H}_n$ is positive definite. Express $\boldsymbol{\Delta}\mathbf{x}_n = \sum_{j=1}^N c_j\mathbf{e}_j$, then $\mathbf{H}_n\Delta\mathbf{x}_n = \sum_{j=1}^N \lambda_j c_j\mathbf{e}_j$, and $\mathbf{e}_i^T\mathbf{H}_n\Delta\mathbf{x}_n = \lambda_i c_i$, and $(\Delta\mathbf{x}_n)^T\mathbf{H}_n\Delta\mathbf{x}_n = \sum_{j=1}^N \lambda_j c_j^2$. Use all of this to calculate the third term on the right-hand side of

Eq. 3.93 between $\mathbf{e}_i^T$ and $\mathbf{e}_i$. It gives $\lambda_i^2 c_i^2 / \sum_{j=1}^N \lambda_j c_j^2$ for the third term. The first term on the right-hand side gives $\mathbf{e}_i^T \mathbf{H}_n \mathbf{e}_i = \lambda_i$. As $1 > \lambda_i c_i^2 / \sum_{j=1}^N \lambda_j c_j^2$, this means that $\lambda_i - \lambda_i^2 c_i^2 / \sum_{j=1}^N \lambda_j c_j^2 > 0$. In other words, the first term minus the third term in Eq. 3.93 gives a positive definite contribution. The second term on the right-hand side gives $\mathbf{e}_i^T \nabla f_n \nabla f_n^T \mathbf{e}_i = \left(\mathbf{e}_i^T \nabla f_n\right)^2 > 0$ in the numerator, so the sign of this term depends on the sign of the denominator $(\nabla f_n)^T (\Delta \mathbf{x}_n)$. If Eq. 3.94 holds, then also the second term gives a positive contribution.

Using $\Delta \mathbf{x}_n = \mathbf{x}_{n+1} - \mathbf{x}_n = -\alpha_n \mathbf{p}_n$, with $\alpha_n > 0$, see Eq. 3.74, the condition of Eq. 3.94 can also be written as

$$\mathbf{p}_n^T \nabla f(\mathbf{x}_{n+1}) < \mathbf{p}_n^T \nabla f(\mathbf{x}_n). \tag{3.95}$$

It states that the gradient ∇f projected onto the search direction $\mathbf{p}_n$, should decrease when stepping from $\mathbf{x}_n$ to $\mathbf{x}_{n+1}$. What one needs to do in an optimization, is find the point where the gradient is zero. So the least one could demand of a search algorithm, is that the gradient gets smaller in a step. This exactly what Eq. 3.95 expresses. It is called the *curvature condition.* It will play a major role in the line-search algorithms discussed in Sec. 3.6.

3.5.2.1 The ugly

As before, it is not the Hessian we want, but its inverse. One can generalize the Sherman-Morrison equation, Eq. 3.71, to the form of Eq. 3.90 and derive

$$\mathbf{H}_{n+1}^{-1} = \mathbf{H}_n^{-1} + \frac{\left(\Delta \mathbf{x}_n^T \nabla f_n + (\nabla f_n)^T \mathbf{H}_n^{-1} \nabla f_n\right) \Delta \mathbf{x}_n \Delta \mathbf{x}_n^T}{\left(\Delta \mathbf{x}_n^T \nabla f_n\right)^2} - \frac{\Delta \mathbf{x}_n \left(\mathbf{H}_n^{-1} \nabla f_n\right)^T}{\Delta \mathbf{x}_n^T \nabla f_n} - \frac{\left(\mathbf{H}_n^{-1} \nabla f_n\right) \Delta \mathbf{x}_n^T}{\Delta \mathbf{x}_n^T \nabla f_n}. \tag{3.96}$$

This is a rather ugly expression. This mainly comes from wanting an explicit expression for the approximate inverse Hessian. The update for the Hessian itself, Eq. 3.93, has a clear philosophy.

The algorithm of *Broyden, Fletcher, Goldfarb, and Shanno* (BFGS) is given by

$$\mathbf{x}_1 = \mathbf{v}; \quad \mathbf{N}_1 = \mathbf{I}; \quad \mathbf{g}_1 = \nabla f(\mathbf{x}_1); \quad \mathbf{p}_1 = \mathbf{g}_1; \tag{3.97}$$

$$\text{do} \quad n = 1, \ldots \quad \alpha_n = \min_{\alpha} f\left(\mathbf{x}_n - \alpha \mathbf{p}_n\right); \quad \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \mathbf{p}_n; \tag{3.98}$$

$$\mathbf{g}_{n+1} = \nabla f(\mathbf{x}_{n+1}); \quad \text{until converged;} \tag{3.99}$$

$$\mathbf{d}_n = \mathbf{x}_{n+1} - \mathbf{x}_n; \quad \mathbf{h}_n = \mathbf{g}_{n+1} - \mathbf{g}_n;$$

$$\mathbf{N}_{n+1} = \mathbf{N}_n + \left(\mathbf{d}_n^T \mathbf{h}_n\right)^{-2} \left(\mathbf{d}_n^T \mathbf{h}_n + \mathbf{h}_n^T \mathbf{N}_n \mathbf{h}_n\right) \mathbf{d}_n \mathbf{d}_n^T - \left(\mathbf{d}_n^T \mathbf{h}_n\right)^{-1} \left(\mathbf{d}_n \left(\mathbf{N}_n \mathbf{h}_n\right)^T + \left(\mathbf{N}_n \mathbf{h}_n\right) \mathbf{d}_n^T\right); \tag{3.100}$$

$$\mathbf{p}_{n+1} = \mathbf{N}_{n+1} \mathbf{g}_{n+1}; \tag{3.101}$$

The update formula, Eq. 3.100 looks far more complicated than Eq. 3.76, and one might wonder whether such a complication is justified. It turns out that it is. As the approximation of the (inverse) Hessian is kept positive definite and symmetric in BFGS, and a positive definite symmetric matrix is also called a metric, the BFGS method is also called a *variable metric method.*

Convergence Broyden methods work well in situations that are similar to where Newton-Raphson works. If a second order Taylor expansion around the minimum is sufficiently accurate in a sufficiently large region, then calculation of a Hessian makes sense. In Newton-Raphson and in Broyden methods, the Hessian determines the search direction in the same way, compare Eqs. 3.61, 3.77, and 3.101. Whereas in Newton-Raphson the true Hessian is used, Broyden methods apply secant (finite difference) approximations to the Hessian.

Somewhat surprisingly, the BFGS algorithm also seems to works well in cases where Newton-Raphson fails. In order to guarantee a downward descent in Eqs. 3.62, 3.74, and 3.98, the (inverse) Hessian needs to be positive definite. Whether this is guaranteed if $\mathbf{x}_n$ is sufficiently close to the minimum $\mathbf{x}^{\min}$, for $\mathbf{x}_n$ far from $\mathbf{x}^{\min}$, the (inverse) Hessian need not be positive definite at all. In that case Newton-Raphson can move us away from the minimum, if we are unlucky. The Broyden update, Eq. 3.76, does not help us in this respect, but the BFGS update, 3.100, keeps the approximate (inverse) Hessian positive definite. In other words, even in case the BFGS Hessian is not very accurate, it does give us a search direction where we move downward and thus closer to the minimum. Brilliant!

Cost Broyden methods do not calculate the Hessian explicitly, and, like steepest descent and conjugate gradient, they only need evaluations of the gradient. In addition, matrix-vector multiplications are required, which scale as $O(N^2)$. Broyden-type updates (including BFGS) need two of these per iteration, $\mathbf{N}_n\mathbf{d}_n$ and $\mathbf{N}_n\mathbf{h}_n$. The total costs of a Broyden method is then $O(nN^2)$, where n is the number of cycles needed for convergence.

Broyden methods need to store the (non-sparse) N-dimensional matrix $\mathbf{N}_n$ (the inverse Hessian). As the matrix is symmetric, we need to store $\frac{1}{2}N(N+1)$ matrix elements. This is unlike the steepest descent or conjugate gradient methods, which need no such matrix. For very large N both this storage, as well as the $O(N^2)$ parts of the calculation become a problem. Things can be improved by formulating the Broyden algorithms in a recursive way.[25] I will show in the next section how to do that for the BFGS algorithm.

3.5.3 Recursive BFGS and limited-memory BFGS*

BFGS is well-liked, and people have worked on extending its range of applications. For large N, we need something that avoids having to store the inverse Hessian matrix, and lowers the computational cost. Start by rewriting Eq. 3.100 as

$$\mathbf{N}_{n+1} = \left[\mathbf{I} - \frac{\mathbf{d}_n\mathbf{h}_n^T}{\mathbf{d}_n^T\mathbf{h}_n}\right]\mathbf{N}_n\left[\mathbf{I} - \frac{\mathbf{h}_n\mathbf{d}_n^T}{\mathbf{d}_n^T\mathbf{h}_n}\right] + \frac{\mathbf{d}_n\mathbf{d}_n^T}{\mathbf{d}_n^T\mathbf{h}_n} \tag{3.102}$$

$$= \mathbf{V}_n^T\mathbf{N}_n\mathbf{V}_n + \rho_n\mathbf{d}_n\mathbf{d}_n^T \quad \text{with } \rho_n = \frac{1}{\mathbf{d}_n^T\mathbf{h}_n} \text{ and } \mathbf{V}_n = \mathbf{I} - \rho_n\mathbf{h}_n\mathbf{d}_n^T. \tag{3.103}$$

Using the prescription of Eq. 3.103 recursively, gives

$$\mathbf{N}_{n+1} = (\mathbf{V}_1\mathbf{V}_2...\mathbf{V}_n)^T\mathbf{N}_1\mathbf{V}_1\mathbf{V}_2...\mathbf{V}_n + \rho_1(\mathbf{V}_2...\mathbf{V}_n)^T\mathbf{d}_1\mathbf{d}_1^T\mathbf{V}_2...\mathbf{V}_n + ...$$
$$... + \rho_{n-1}\mathbf{V}_n^T\mathbf{d}_{n-1}\mathbf{d}_{n-1}^T\mathbf{V}_n + \rho_n\mathbf{d}_n\mathbf{d}_n^T. \tag{3.104}$$

[25] Recursive in the mathematical sense, as in a mathematical recursion relation. It might, but does not necessarily lead to recursive programming. In the examples I will show you, I have no need for recursive programming.

Note that I have chosen $\mathbf{N}_1 = \mathbf{I}$, Eq. 3.97. Eq. 3.104 demonstrates that the inverse Hessian $\mathbf{N}_{n+1}$ can be expressed completely in terms of the vectors $\mathbf{d}_i;\ i = 1, ..., n$ and $\mathbf{h}_i;\ i = 1, ..., n$. Storing only these vectors, the memory requirement is then $2nN$, which is a lot less than storing the full Hessian, if $n \ll N$. We also store the numbers $\rho_i;\ i = 1, ..., n$, Eq. 3.103, as we will need them later on.

We don't actually want to calculate the inverse Hessian explicitly, otherwise we would lose this advantage again. We need to calculate $\mathbf{N}_{n+1}\mathbf{g}_{n+1}$, Eq. 3.101. In view of Eq. 3.104, it makes sense to calculate as intermediates the vectors $\mathbf{q}_n = \mathbf{V}_n\mathbf{g}_{n+1}$, $\mathbf{q}_{n-1} = \mathbf{V}_{n-1}\mathbf{V}_n\mathbf{g}_{n+1}$,..., $\mathbf{q}_1 = \mathbf{V}_1\mathbf{V}_2...\mathbf{V}_n\mathbf{g}_{n+1}$. The following routine does this

$$\mathbf{q}_{n+1} = \mathbf{g}_{n+1};\quad \text{do } i = n, ..., 1\quad \sigma_i = \rho_i \mathbf{d}_i^T \mathbf{q}_{i+1};\quad \mathbf{q}_i = \mathbf{q}_{i+1} - \sigma_i \mathbf{h}_i; \tag{3.105}$$

where we make use of the expressions for $\mathbf{V}_i;\ i = 1, ..., n$, Eq. 3.103. Note that the do-loop runs backwards, the vectors $\mathbf{q}_i;\ i = 1, ..., n+1$ are produced in reverse order. We need to store all numbers $\sigma_i;\ i = 1, ..., N$, but we only require the last vector $\mathbf{q}_1$. This is because

$$\begin{aligned}
\mathbf{N}_{n+1}\mathbf{g}_{n+1} &= \left(\mathbf{V}_1\mathbf{V}_2...\mathbf{V}_n\right)^T \mathbf{N}_1\mathbf{q}_1 + \\
&+ \rho_1 \left(\mathbf{V}_2...\mathbf{V}_n\right)^T \mathbf{d}_1\mathbf{d}_1^T\mathbf{q}_2 + ... + \rho_{n-1}\mathbf{V}_n^T\mathbf{d}_{n-1}\mathbf{d}_{n-1}^T\mathbf{q}_n + \rho_n\mathbf{d}_n\mathbf{d}_n^T\mathbf{q}_{n+1} \\
&= \mathbf{V}_n^T...\mathbf{V}_2^T\mathbf{V}_1^T\mathbf{N}_1\mathbf{q}_1 + \\
&+ \sigma_1\mathbf{V}_n^T...\mathbf{V}_2^T\mathbf{d}_1 + ... + \sigma_{n-1}\mathbf{V}_n^T\mathbf{d}_{n-1} + \sigma_n\mathbf{d}_n,
\end{aligned} \tag{3.106}$$

using Eqs. 3.104 and 3.105. The last two lines of Eq. 3.106 can be written in a do-loop as

$$\mathbf{s}_1 = \mathbf{N}_1\mathbf{q}_1;\quad \text{do } i = 1, ..., n\quad \mathbf{s}_{i+1} = \mathbf{V}_i^T\mathbf{s}_i + \sigma_i\mathbf{d}_i; \tag{3.107}$$

Making use of the expressions for $\mathbf{V}_i;\ i = 1, ..., n$, Eq. 3.103, this becomes

$$\mathbf{s}_1 = \mathbf{N}_1\mathbf{q}_1;\quad \text{do } i = 1, ..., n\quad \tau_i = \rho_i\mathbf{h}_i^T\mathbf{s}_i;\quad \mathbf{s}_{i+1} = \mathbf{s}_i + (\sigma_i - \tau_i)\,\mathbf{d}_i; \tag{3.108}$$

Per definition, we now have $\mathbf{s}_{n+1} = \mathbf{N}_{n+1}\mathbf{g}_{n+1}$.

In conclusion, the two do-loops of Eqs. 3.105 and 3.108 perform the matrix-vector product $\mathbf{N}_{n+1}\mathbf{g}_{n+1}$. Each of the do-loops requires n times an inner product $\mathbf{a}^T\mathbf{b}$, which is a $O(N)$ operation. So, the total cost for this matrix-vector product is $O(nN)$, which is a lot less than the original $O(N^2)$, provided $n \ll N$.

The *recursive BFGS algorithm* is given by [24]

$$\mathbf{x}_1 = \mathbf{v};\quad \mathbf{N}_1 = \mathbf{I};\quad \mathbf{g}_1 = \nabla f(\mathbf{x}_1);\quad \mathbf{p}_1 = \mathbf{g}_1; \tag{3.109}$$

$$\text{do}\quad n = 1, ...\quad \alpha_n = \min_{\alpha} f\left(\mathbf{x}_n - \alpha\mathbf{p}_n\right);\ \mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n\mathbf{p}_n; \tag{3.110}$$

$$\mathbf{g}_{n+1} = \nabla f(\mathbf{x}_{n+1});\quad \text{until converged;} \tag{3.111}$$

$$\mathbf{d}_n = \mathbf{x}_{n+1} - \mathbf{x}_n;\quad \mathbf{h}_n = \mathbf{g}_{n+1} - \mathbf{g}_n; \tag{3.112}$$

$$\mathbf{q}_{n+1} = \mathbf{g}_{n+1};\quad \text{do } i = n, ..., 1\quad \sigma_i = \rho_i\mathbf{d}_i^T\mathbf{q}_{i+1};\quad \mathbf{q}_i = \mathbf{q}_{i+1} - \sigma_i\mathbf{h}_i; \tag{3.113}$$

$$\mathbf{s}_1 = \mathbf{N}_1\mathbf{q}_1;\quad \text{do } i = 1, ..., n\quad \tau_i = \rho_i\mathbf{h}_i^T\mathbf{s}_i;\quad \mathbf{s}_{i+1} = \mathbf{s}_i + (\sigma_i - \tau_i)\,\mathbf{d}_i; \tag{3.114}$$

$$\mathbf{p}_{n+1} = \mathbf{s}_{n+1}; \tag{3.115}$$

The last two lines of the previous, non-recursive, BFGS algorithm, Eqs. 3.100 and 3.101, have been replaced by Eqs. 3.113, 3.114, and 3.115.

Limited-memory BFGS (L-BFGS) The algorithm as formulated above allows for an interesting option. As is stands, the do-loops of Eqs. 3.113 and 3.114 take into account the points $\mathbf{x}_i;\ i = 1, ..., n+1$ and gradients $\mathbf{g}_i;\ i = 1, ..., n+1$ generated in all previous iterations. In other words, the algorithm has a perfect memory. The first few points and gradients generated may not be that good, as typically the first few points are still far from the minimum and the gradients are large. There is no need to keep a memory of bad points.

As discussed in the lecture notes on (non-)linear problems (the art of subspace building) one could make a selection of points to incorporate, for instance by throwing out the points with large gradients. Another strategy is to keep the memory fixed, i.e., only keep a memory of the last m points [24]. So whenever a new point $\mathbf{x}_{n+1}$ is generated, the point $\mathbf{x}_{n+1-m}$ is removed. The do-loops of Eqs. 3.113 and 3.114 then run between $n+1-m$ and n. This version of the algorithm is called *limited-memory* BFGS or L-BFGS for short. Surprisingly small numbers $m \lesssim 10$ often give decent results. L-BFGS is currently very popular, for instance in the machine learning community.

Convergence Recursive BFGS is equivalent to ordinary BFGS, so everything said in the previous section remains valid.

Cost Like the previously discussed methods, recursive BFGS needs evaluations of the gradient. Due to the recursive formulation, matrix-vector multiplications with the (inverse) Hessian only cost $O(nN)$ operations, where we need two of these per iteration. The total costs of recursive BFGS method is then $O(n^2 N)$, with n the number of cycles needed for convergence. The memory requirements are $\sim 2nN$, as discussed above.

In L-BFGS one has a fixed memory of m vectors. leading to a memory requirement of $\sim 2mN$. Matrix-vector multiplications with the Hessian then cost $O(mN)$ operations. With n the number of cycles needed for convergence, the total computational costs of L-BFGS are $O(mnN)$. Typical numbers could be $N = 10^4$, $n = 10^2$, $m = 10^1$.

3.5.4 Pulay iteration*

In the lecture notes on (non-)linear problems I have presented Pulay iteration as a subspace technique for iteratively solving a non-linear problem. We can also apply Pulay iteration for optimizing a function $f(\mathbf{x})$. Here I want to show the connection between Pulay iteration and the Broyden methods [25, 10].

The background idea is that the minimum of $f(\mathbf{x})$ is given by the solution of the equation $\nabla f(\mathbf{x}) = 0$. We write the latter in the style of a fixed-point iteration rule $\mathbf{x} = \mathbf{x} - \nabla f(\mathbf{x})$. The "$-$" is introduced because $-\nabla f(\mathbf{x})$ gives a descent direction, which is always handy in a minimization problem. From the fixed-point rule, one would extract the propagation rule

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \nabla f(\mathbf{x}_n). \tag{3.116}$$

In Pulay iteration, instead of using Eq. 3.116 directly, one keeps a memory of previous iteration steps

$$\mathbf{x}_{n+1} = \tilde{\mathbf{x}}_n - \nabla f(\tilde{\mathbf{x}}_n) \text{ with } \tilde{\mathbf{x}}_n = \mathbf{x}_n + \sum_{m=1}^{n-1} a_m \left(\mathbf{x}_m - \mathbf{x}_{m+1}\right). \tag{3.117}$$

Ideally, one would optimize the coefficients a_m; $m = 1, ..., n-1$ such that $\|\nabla f(\tilde{\mathbf{x}}_n)\|^2$ is minimal. That is a difficult non-linear optimization problem by itself, and certainly not something one wants to do each cycle.

The central ansatz in Pulay iteration is

$$\nabla f(\tilde{\mathbf{x}}_n) = \nabla f\left(\mathbf{x}_n + \sum_{m=1}^{n-1} a_m \left(\mathbf{x}_m - \mathbf{x}_{m+1}\right)\right) \approx \boldsymbol{\Delta} = \nabla f(\mathbf{x}_n) + \sum_{m=1}^{n-1} a_m \left(\nabla f(\mathbf{x}_m) - \nabla f(\mathbf{x}_{m+1})\right). \tag{3.118}$$

The gradient at $\tilde{\mathbf{x}}_n$ is approximated by an interpolation of the gradients at positions $\mathbf{x}_m$; $m = 1, ..., n$. We minimize the interpolated gradient

$$\min_{a_1,...,a_{n-1}} \|\boldsymbol{\Delta}\|^2, \tag{3.119}$$

and the new point becomes

$$\mathbf{x}_{n+1} = \tilde{\mathbf{x}}_n - \boldsymbol{\Delta}_{\min}. \tag{3.120}$$

How to solve the minimization problem, Eq. 3.119, is explained in the lecture notes on (non-)linear problems.

It is shown there that

$$\boldsymbol{\Delta}_{\min} = \nabla f(\mathbf{x}_n)^{\perp} = \nabla f(\mathbf{x}_n) - \nabla f(\mathbf{x}_n)^{\parallel} \equiv \nabla f(\mathbf{x}_n) - \mathbf{P}_n \nabla f(\mathbf{x}_n), \tag{3.121}$$

where $\nabla f(\mathbf{x}_n)^{\perp}$ and $\nabla f(\mathbf{x}_n)^{\parallel}$ are the components of the gradient $\nabla f(\mathbf{x}_n)$ perpendicular, respectively parallel, to the hyperplane spanned by

$$\text{span}\left\{\nabla f_m \mid \nabla f_m = \nabla f(\mathbf{x}_m) - \nabla f(\mathbf{x}_{m+1});\quad m = 1, ..., n-1\right\}. \tag{3.122}$$

$\mathbf{P}_n$ is then projection operator onto that hyperplane.

One can write Eq. 3.120 such, that it resembles a Broyden update formula. Define an "inverse Hessian" by

$$\mathbf{H}_n^{-1} \equiv \mathbf{M}_n \mathbf{P}_n + \mathbf{I} - \mathbf{P}_n, \tag{3.123}$$

using the projection operator of Eq. 3.121. For any vector $\nabla f^{\parallel}$ within the hyperplane defined by Eq. 3.122, we have $\mathbf{P}_n \nabla f^{\parallel} = \nabla f^{\parallel}$, and $\mathbf{H}_n^{-1} \nabla f^{\parallel} = \mathbf{M}_n \nabla f^{\parallel}$. Within the hyperplane, we insist that the matrix $\mathbf{M}_n$ obeys the secant equation, Eq. 3.78

$$\mathbf{M}_n \nabla f_m = \Delta \mathbf{x}_m; \text{ with } \Delta \mathbf{x}_m = \mathbf{x}_{m+1} - \mathbf{x}_m \text{ and } \nabla f_m = \nabla f(\mathbf{x}_{m+1}) - \nabla f(\mathbf{x}_m);\ m = 1, ..., n-1. \tag{3.124}$$

For any vector $\nabla f^{\perp}$ perpendicular to the hyperplane of Eq. 3.122, we have $\mathbf{P}_n \nabla f^{\perp} = 0$, so $\mathbf{H}_n^{-1} \nabla f^{\perp} = \nabla f^{\perp}$ behaves like the identity operator. With these definitions, one can rewrite Eq. 3.120 as

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{H}_n^{-1} \nabla f(\mathbf{x}_n), \tag{3.125}$$

which reads like a Newton-Raphson like update, Eq. 3.60.

We still need to have explicit expressions for the matrices of Eq. 3.123. Similar to the Broyden methods discussed in the previous sections, one can construct a recursive expression

$$\mathbf{H}_{n+1}^{-1} = \mathbf{H}_n^{-1} + \frac{\Delta \mathbf{x}_n - \mathbf{H}_n^{-1} \nabla f_n}{\left(\nabla f_n^{\perp}\right)^T \nabla f_n} \left(\nabla f_n^{\perp}\right)^T \text{ and } \mathbf{H}_0^{-1} = \mathbf{I}. \tag{3.126}$$

This indeed is a Broyden update of the general type discussed in Sec. 3.5.1.3, Eqs. 3.87 and 3.88. It mostly resembles the "bad Broyden" method, Eq. 3.81. The latter uses the projection vector $\boldsymbol{\zeta}_n = \nabla f_n$, whereas in Pulay iteration, Eq. 3.126, one applies $\boldsymbol{\zeta}_n = \nabla f_n^{\perp}$, which is the vector orthogonal to the hyperplane of Eq. 3.122.

Proof: of Eq. 3.125. One uses Eqs. 3.117 and 3.121 to rewrite Eq. 3.120 as

$$\begin{aligned}
\mathbf{x}_{n+1} &= \mathbf{x}_n + \sum_{m=1}^{n-1} a_m \left(\mathbf{x}_m - \mathbf{x}_{m+1}\right) - \nabla f(\mathbf{x}_n)^{\perp} \\
&= \mathbf{x}_n + \sum_{m=1}^{n-1} a_m \mathbf{M}_n \left(\nabla f(\mathbf{x}_m) - \nabla f(\mathbf{x}_{m+1})\right) - \nabla f(\mathbf{x}_n)^{\perp} \\
&= \mathbf{x}_n + \mathbf{M}_n \left[\sum_{m=1}^{n-1} a_m \left(\nabla f(\mathbf{x}_m) - \nabla f(\mathbf{x}_{m+1})\right)\right] - \nabla f(\mathbf{x}_n)^{\perp} \\
&= \mathbf{x}_n - \mathbf{M}_n \nabla f(\mathbf{x}_n)^{\parallel} - \nabla f(\mathbf{x}_n)^{\perp} = \mathbf{x}_n - \mathbf{H}_n^{-1} \nabla f(\mathbf{x}_n).
\end{aligned} \tag{3.127}$$

The second line uses the definition of Eq. 3.124, the third line uses the fact that the matrix $\mathbf{M}_n$ defines a linear operation, and the fourth line uses Eqs. 3.118 and 3.121.

Proof: of Eq. 3.127 by total intimidation. One has to prove that the recursion relation of Eq. 3.126 leads to Eqs. 3.123 and 3.124. Assuming that this is the case for $\mathbf{H}_n^{-1}$, we first prove Eq. 3.124 for $\mathbf{H}_{n+1}^{-1}$

$$\mathbf{H}_{n+1}^{-1} \nabla f_m = \Delta \mathbf{x}_m; \;\; m = 1, ..., n. \tag{3.128}$$

For $m = 1, ..., n-1$, we have $\mathbf{H}_{n+1}^{-1} \nabla f_m = \mathbf{H}_n^{-1} \nabla f_m$, as $\left(\nabla f_n^{\perp}\right)^T \nabla f_m = 0$. We have assumed that $\mathbf{H}_n^{-1}$ obeys Eq. 3.124, so Eq. 3.128 is correct for $m = 1, ..., n-1$. For $m = n$, we have

$$\mathbf{H}_{n+1}^{-1} \nabla f_n = \mathbf{H}_n^{-1} \nabla f_n + \frac{\Delta \mathbf{x}_n - \mathbf{H}_n^{-1} \nabla f_n}{\left(\nabla f_n^{\perp}\right)^T \nabla f_n} \left(\nabla f_n^{\perp}\right)^T \nabla f_n = \Delta \mathbf{x}_n. \tag{3.129}$$

This proves Eq. 3.128 for vectors within the hyperplane define by

$$\text{span}\left\{\nabla f_m \mid \nabla f_m = \nabla f(\mathbf{x}_m) - \nabla f(\mathbf{x}_{m+1}); \;\; m = 1, ..., n\right\}. \tag{3.130}$$

Vectors with a component outside this hyperplane can be decomposed as $\nabla f = \nabla f^{\parallel} + \nabla f^{\perp}$. As we already have given the proof for the component $\nabla f^{\parallel}$ in the hyperplane, we only need to consider the component $\nabla f^{\perp}$ perpendicular to the hyperplane. We have $\mathbf{H}_{n+1}^{-1} \nabla f^{\perp} = \mathbf{H}_n^{-1} \nabla f^{\perp}$, as $\left(\nabla f_n^{\perp}\right)^T \nabla f^{\perp} = 0$. The latter holds because $\nabla f_n^{\perp} \in$ hyperplane of Eq. 3.130. As by assumption $\mathbf{H}_n^{-1} \nabla f^{\perp} = \nabla f^{\perp}$, we have proved Eq. 3.123. Repeating the arguments for $n = 0$ shows that indeed it holds for this case, which completes the proof.

Convergence See the lecture notes on (non-)linear problems. I think Pulay iteration should behave somewhat better than the (good or bad) Broyden methods, but that is speculation from my side.

Cost One could use Eqs. 3.125 and 3.126 as a starting point for an implementation of Pulay iteration. This has the usual drawbacks of a non-recursive Broyden algorithm, see Sec. 3.5.2. Storing the inverse Hessian costs $\sim N^2$ of memory, and the calculation is $O(nN^2)$. Using the techniques explained in the lecture notes on (non-)linear problems make Pulay

iteration cost only $\sim N$ of memory, and gives a $O(n^2N)$ calculation. Maximum efficient use of the information gathered in previous iterations can actually turn Pulay iteration into a $O(nN)$ calculation, which is very efficient.

3.6 Line searches

Line searches are an integral part of optimization algorithms. Such algorithms map out a path in N-dimensional space $\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_n \rightarrow \mathbf{x}^{\min}$, where $\mathbf{x}_0$ is a starting point, and $f(\mathbf{x}^{\min})$ is a (local) minimum of the function. The path is constructed by a rule of type $\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha_n \mathbf{p}_n$, with $\mathbf{p}_n$ a search direction, and α_n a step size. The algorithms discussed above differ in how they determine optimal search directions $\mathbf{p}_n$. Steepest descent, conjugate gradient, Newton-Raphson, and quasi-Newton algorithms all use different strategies and find different $\mathbf{p}_n$.

However, they all need an additional algorithm to find an optimal step size α_n. This is done by a *line search*, which has the following task. With vectors $\mathbf{x}$ and $\mathbf{p}$ fixed, and α variable, find

$$\min_{\alpha} f(\mathbf{x} - \alpha\mathbf{p}) \equiv \min_{\alpha} F(\alpha). \tag{3.131}$$

$\mathbf{x} - \alpha\mathbf{p}$ describes a line in N-dimensional space, so that is where the phrase "line search" comes from. It is an optimization of a function of one variable, $F(\alpha)$.

That sounds simple, and there are loads of algorithms around for that task, but there are a couple of snakes in the grass. Line searches are kind of the Achilles heel of optimization algorithms. Heuristics can play an important role, i.e., inside knowledge of the function you are trying to minimize, giving you guesses for "reasonable" values of α, or "reasonable" intervals. Here, I can only give you general guidelines and strategies, which, unfortunately, makes this section rather long. The algorithms are discussed in Sec. 3.6.2 and 3.6.4, which you might consult right away if you don't want to know about the why, but only about the how.

What is the cost of a line search? Searching for the solution to Eq. 3.131, one needs to be able to evaluate the function $F(\alpha)$ at any point α, and preferably some of its derivatives. Using the chain rule on Eq. 3.131 one gets

$$\frac{dF(\alpha)}{d\alpha} = -\mathbf{p}^T \nabla f(\mathbf{x} - \alpha\mathbf{p}), \tag{3.132}$$

and

$$\frac{d^2F(\alpha)}{d\alpha^2} = \mathbf{p}^T \mathbf{H}(\mathbf{x} - \alpha\mathbf{p})\,\mathbf{p}. \tag{3.133}$$

This means that evaluating derivatives is *not cheap*! To evaluate the first derivative, one has to calculate the gradient ∇f, which is a N-dimensional vector, Eq. 3.47. Evaluating the second derivative involves calculating the Hessian $\mathbf{H}$, which is a $N \times N$ matrix, Eq. 3.48. As discussed in Sec. 3.4.3, calculating the Hessian is impractical to impossible in many physics problems. That rules out many line-search algorithms that use second derivatives, such as Newton-Raphson.

In the following I will suppose that we have access to first derivatives, and that the prize tag on evaluating Eq. 3.132 is comparable to that of evaluating a function value $f(\mathbf{x} - \alpha\mathbf{p})$. There are algorithms that only need access to function values, and not to gradients, such

as the golden-section search. I won't discuss these. The common wisdom seems to be: it is better/faster to use gradients if they are available and not too expensive. Then there are algorithms that dominantly use function values, and a few gradient evaluations. To limit the discussion somewhat, I will assume that we can freely use function values and gradients at about the same cost.

Does a line search make sense? We demand that the search direction $-\mathbf{p}_n$ is a *descent* direction, so

$$\frac{dF(0)}{d\alpha} = -\mathbf{p}^T \nabla f(\mathbf{x}) < 0. \tag{3.134}$$

This is guaranteed in steepest descent, and in BFGS, for instance. It is not guaranteed in conjugate gradient, or in the original Broyden method. If Eq. 3.134 is obeyed, then we look for a minimum in $F(\alpha)$ for $\alpha > 0$. If Eq. 3.134 is not obeyed, then we decide that a line search makes no sense. Let the line search routine signal a failure, and return $\alpha = 0$. The calling optimization routine can then react by restarting the algorithm with a steepest descent step, for instance, which will guarantee Eq. 3.134 gets obeyed next time you try. Note that the Polak-Ribière variant of conjugate gradient, Eq. 3.58, does this automatically.

How accurate does a line search have to be? One classifies line search algorithms according to the accuracy with which they find the minimum of $F(\alpha)$. You can find the minimum very accurately, called an *exact line search*, or not accurate at all, called an *inexact line search*. One does not always choose the most accurate method, because *accuracy comes at a prize*. More accurate methods need more evaluations of gradient values $\nabla f(\mathbf{x} - \alpha\mathbf{p})$ and/or function values $f(\mathbf{x} - \alpha\mathbf{p})$ for different α's.

In the algorithms we have been discussing in the previous sections, steepest descent, conjugate gradient, (quasi-)Newton, evaluations of the gradient are also needed to determine the search directions $\mathbf{p}$. Ultimately, the total number of times the gradient/function needs to be calculated determines the computational cost of the algorithm. It makes sense to divide that number in a balanced way over calculating search directions, and calculating step sizes. Spending all your time on one or the other, does not bring you to the minimum fast.

The common folklore states that conjugate gradient needs fairly exact line searches, whereas (quasi-)Newton methods, or steepest descent, can get away with inexact line searches.

Details: to generate conjugate search directions $\mathbf{p}_n$ for quadratic functions, Eq. 3.30, one needs Eqs. 3.27-3.29 to hold. Those, however, only hold exactly with exact line searches. With inexact line searches the conjugacy deteriorates, the search directions are less optimal, and the algorithm deteriorates. This does not usually happen in a dramatic way, but you should not try and push your luck by choosing line searches that are too inaccurate.

For quasi-Newton methods the generated points $\mathbf{x}_n$ are also used to build up a better approximation to the Hessian. It does not matter too greatly if a point is a bit off, provided it is roughly in the right region. The updated Hessian will steer the next search direction toward the minimum, so quasi-Newton algorithms are sort of self-correcting. Newton-Raphson is self-correcting because one uses the true Hessian. Steepest descent does not bring you very fast to the minimum anyway, but it does not seem get much worse by inexact line searches.

3.6.1 Convergence criteria: the Armijo-Wolfe conditions

If we know that a minimum of $F(\alpha)$ exists, then finding the minimum is equivalent to finding the root

$$\frac{dF(\alpha)}{d\alpha} = -\mathbf{p}^T \nabla f\left(\mathbf{x} - \alpha\mathbf{p}\right) = 0. \tag{3.135}$$

You should worry about accidentally finding a maximum, if you rely on Eq. 3.135 only. To monitor convergence, we therefore need two criteria, one that monitors the size of $|dF(\alpha)/d\alpha|$, and one that monitors $F(\alpha)$ going down. Together these two criteria are called the *Armijo-Wolfe conditions*.

The Wolfe condition A convergence condition on the derivative is

$$\left|\frac{dF(\alpha)}{d\alpha}\right| < c_2 \left|\frac{dF(0)}{d\alpha}\right| \quad \text{or} \quad \left|\mathbf{p}^T \nabla f\left(\mathbf{x} - \alpha\mathbf{p}\right)\right| < c_2 \left|\mathbf{p}^T \nabla f\left(\mathbf{x}\right)\right| \quad \text{with} \quad 0 < c_2 < 1, \tag{3.136}$$

a constant. This is called the *strong Wolfe condition on the curvature*. The smaller c_2, the more accurate Eq. 3.135 is solved, i.e., the exacter the line search. A typical number for conjugate gradient (an exact line search) would be $c_2 = 0.1$, whereas quasi-Newton can get away with values of up to $c_2 = 0.9$ (an inexact line search). Using BFGS and the convergence criterion of Eq. 3.136, ensures that the approximate Hessian stays *positive definite*, which is required for that algorithm.

Proof: Eq. 3.136 very much resembles Eq. 3.95. In fact, if Eq. 3.136 holds, then certainly

$$-\frac{dF(\alpha)}{d\alpha} < -c_2 \frac{dF(0)}{d\alpha} \quad \text{or} \quad \mathbf{p}^T \nabla f(\mathbf{x} - \alpha\mathbf{p}) < c_2 \mathbf{p}^T \nabla f(\mathbf{x}) \quad \text{with} \quad 0 < c_2 < 1, \tag{3.137}$$

holds. As we demanded that $dF(0)/d\alpha < 0$, Eq. 3.134, the right-hand side of the inequalities of Eqs. 3.137 and 3.136 are identical. If, $dF(\alpha)/d\alpha < 0$, the left-hand sides are also identical. If $dF(\alpha)/d\alpha > 0$, then Eq. 3.137 holds anyway (but Eq. 3.136 not necessarily). Eq. 3.137 is called the *weak Wolfe condition on the curvature*. I will use the strong form, Eq. 3.136, throughout.

The Armijo rule Suppose we have found a point $\alpha > 0$ that obeys the the Wolfe condition, Eq. 3.136, meaning that the gradient has decreased sufficiently with respect to the gradient at $\alpha = 0$. *This is not enough!* To establish whether we are on a route toward a minimum, we have to see whether the function value at $\alpha > 0$ has decreased sufficiently with respect to the function value at $\alpha = 0$. Consider the following test

$$F(\alpha) < F(0) + c_1 \alpha F'(0) \quad \text{or} \quad f(\mathbf{x} - \alpha\mathbf{p}) - f(\mathbf{x}) < -c_1 \alpha \mathbf{p}^T \nabla f(\mathbf{x}) \quad \text{with} \quad 0 < c_1 < \frac{1}{2}, \tag{3.138}$$

a constant, and $F'(0) = dF(0)/d\alpha$. This is called the *Armijo rule*.[26] What it does, is shown in Fig. 3.3. The Armijo rule is also called the *condition for sufficient decrease*, i.e., it ensures that the function $F(\alpha)$ has decreased sufficiently, compared to the starting value $F(0)$. In practice, quite small numbers for c_1 are often sufficient, e.g., $c_1 = 10^{-2}$-10^{-4}. The Armijo rule dictates that the function value $F(\alpha)$ should be lower than $F(0)$, but by how much exactly, is apparently not super-critical.

[26] "Armijo" pronounced the Spanish way, i.e., the "j" as a hard Dutch "g".

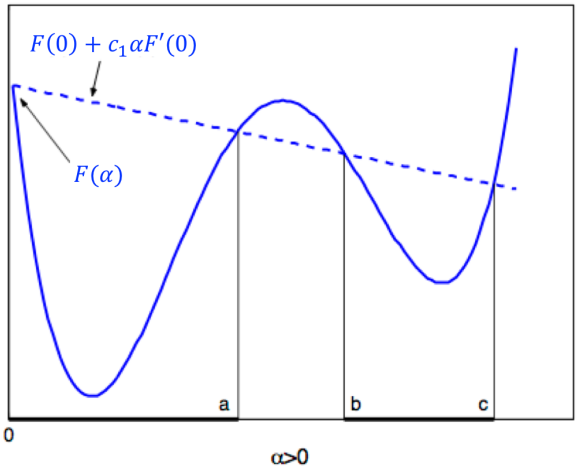


Figure 3.3: The Armijo rule is used to determine whether the function value has decreased sufficiently in a step α. In this case, α's in both the intervals $[0; a]$ and $[b; c]$ are allowed by the rule.

The interval $[0; a]$, where $a > 0$ is the smallest value for which $F(a) = F(0) + c_1 a F'(0)$, see Fig. 3.3, contains a(t least one) point $F(\alpha)$ that obeys the Wolfe condition Eq. 3.136, provided $0 < c_1 < c_2 < 1$. This means that this interval contains a(t least one) minimum, which, in principle, we can find with any accuracy we like.[27]

Proof: take a point μ in the open interval $(0; a)$, with $F(0) > F(\mu) > F(a)$. We know that $F(\mu) < F(0) + c_1 \mu F'(0)$, and $F(a) = F(0) + c_1 a F'(0)$. The mean value theorem says that there exists a point α in the interval $(\mu; a)$, where $F'(\alpha) = (F(a) - F(\mu)) / (a - \mu)$. Using what we know about the two points, it follows $F'(\alpha) > c_1 F'(0) > c_2 F'(0)$, provided $0 < c_1 < c_2$, as we know that $F'(0) < 0$. But as also $F'(\alpha) < 0$, we have $|F'(\alpha)| < c_2 |F'(0)|$, which is the Wolfe condition.

Theorem on convergence of optimization algorithms: One can prove that line search algorithms that obey both *the Armijo rule* (function sufficiently decreased), Eq. 3.138, and the *strong Wolfe condition* (gradient sufficiently decreased), Eq. 3.136, make both the conjugate gradient algorithm and quasi-Newton algorithms such as the BFGS algorithm, *always converge*, provided $0 < c_1 < c_2 < 1$.[28] The two conditions combined, i.e., Eqs. 3.136 and 3.138, are sometimes called the *Wolfe conditions*. I will call them the *Armijo-Wolfe conditions*.

3.6.2 Simple searches: interpolation and extrapolation

Now we know which convergence conditions we need to obey, we can finally start our line search. In our optimization algorithms, the starting point for the a search is known, $\alpha = 0$. First we must have an idea about what a sensible scale η would be for an optimal step in α. For a quadratic problem, Newton-Raphson gives $\alpha = 1$ as the optimal parameter. For

[27] In the case of Fig. 3.3, the same is actually true for the interval $[b; c]$.
[28] The proof of this theorem is discussed, for instance, in Refs. [20, 21].

(quasi-)Newton methods, an sensible scale is then

$$\eta = 1. \tag{3.139}$$

For conjugate gradient, finding a proper scale η is more tricky. That is because the scale of the search vector $\mathbf{p}$ is the scale of the gradient ∇f, which also sets the scale for α.

Details: in one dimension, suppose that

$$f(x) = \frac{1}{2}\eta^{-1}\left(x - x_0\right)^2 + c, \tag{3.140}$$

is a quadratic function. For quasi-Newton methods $\mathbf{p} \approx \mathbf{H}^{-1}\nabla f$, with $\mathbf{H}^{-1}$ the inverse Hessian. In one dimension, $\nabla f = df/dx = \eta^{-1}\left(x - x_0\right)$, and $\mathbf{H} = d^2f/dx^2 = \eta^{-1}$, so $p \approx x - x_0$, which means that $dp/dx = 1$, i.e., p runs on the same scale as x. In this particular case $x - \alpha p$ gives the minimum for $\alpha_{\min} = 1$. For conjugate gradient or steepest descent, $\mathbf{p} \approx \nabla f$, which for our one-dimensional example means $p \approx \eta^{-1}\left(x - x_0\right)$. In that case $x - \alpha p$ gives the minimum for $\alpha_{\min} = \eta$.

There are several strategies for finding an optimal scale. From two points $F(\alpha_0)$ and $F(\alpha_1)$, and the derivative $dF(\alpha_0)/d\alpha$, we can construct a *quadratic function* $F_2(\alpha; \alpha_0, \alpha_1)$ in α (a parabola), see *"Interpolation/extrapolation"* below. The minimum of this function gives an estimate for the optimal parameter η,

$$\eta = \min_\alpha F_2(\alpha; \alpha_0, \alpha_1) = \frac{\left(\alpha_1 - \alpha_0\right)^2 \mathbf{p}^T\nabla f(\mathbf{x} - \alpha_0\mathbf{p})}{2\left[f(\mathbf{x} - \alpha_1\mathbf{p}) - f(\mathbf{x} - \alpha_0\mathbf{p}) + \left(\alpha_1 - \alpha_0\right)\mathbf{p}^T\nabla f(\mathbf{x} - \alpha_0\mathbf{p})\right]}. \tag{3.141}$$

If we use $\alpha_0 = 0$, we already have the function value and gradient at this point. We have to calculate the same information at α_1 (and make sure that we choose some reasonable value for α_1).

Alternatively, from two points $F(\alpha_0)$ and $F(\alpha_1)$, and the two derivatives $dF(\alpha_0)/d\alpha$ and $dF(\alpha_1)/d\alpha$, we can construct a *cubic function* $F_3(\alpha; \alpha_0, \alpha_1)$ in α. The minimum of this function is then given by

$$\begin{aligned}
\eta &= \min_\alpha F_3(\alpha; \alpha_0, \alpha_1) = \left(\alpha_1 - \alpha_0\right)\frac{\mathbf{p}^T\nabla f(\mathbf{x} - \alpha_1\mathbf{p}) - a_1 - a_2}{\mathbf{p}^T\nabla f(\mathbf{x} - \alpha_0\mathbf{p}) - \mathbf{p}^T\nabla f(\mathbf{x} - \alpha_1\mathbf{p}) + 2a_2} \quad \text{with} \\
a_1 &= \mathbf{p}^T\nabla f(\mathbf{x} - \alpha_0\mathbf{p}) + \mathbf{p}^T\nabla f(\mathbf{x} - \alpha_1\mathbf{p}) + 3\left(\alpha_1 - \alpha_0\right)^{-1}\left[f(\mathbf{x} - \alpha_1\mathbf{p}) - f(\mathbf{x} - \alpha_0\mathbf{p})\right] \quad \text{and} \\
a_2 &= \left[a_1^2 - \mathbf{p}^T\nabla f(\mathbf{x} - \alpha_0\mathbf{p})\mathbf{p}^T\nabla f(\mathbf{x} - \alpha_1\mathbf{p})\right]^{\frac{1}{2}}.
\end{aligned} \tag{3.142}$$

The expression looks formidable, but that is mainly due to the technicalities of cubic interpolation, see below. Cubic interpolation/extrapolation is the *standard technique* for guessing a new point in line searches (in this case η). However, it is not guaranteed to work, as in Eq. 3.142 one should have $a_1^2 > \mathbf{p}^T\nabla f(\mathbf{x} - \alpha_0\mathbf{p})\mathbf{p}^T\nabla f(\mathbf{x} - \alpha_1\mathbf{p})$. If cubic interpolation/extrapolation does not work, one can fall back on quadratic interpolation/extrapolation, Eq. 3.141.

A third alternative makes use of information accumulated at a previous point in our conjugate gradient or quasi-Newton search. Suppose $\mathbf{x} = \mathbf{x}_n$; $n > 1$, then we have all information of the previous point $\mathbf{x}_{n-1}$ in our search available. Assuming that the first-order change in the function $f(\mathbf{x} - \alpha\mathbf{p}) - f(\mathbf{x}) \approx -\alpha\mathbf{p}^T\nabla f(\mathbf{x})$ is the same for $\mathbf{x}, \mathbf{p} = \mathbf{x}_n, \mathbf{p}_n$ and for $\mathbf{x}, \mathbf{p} = \mathbf{x}_{n-1}, \mathbf{p}_{n-1}$ (a big if, and not guaranteed at all), then we can guess the scale η_n at the new point from $\eta_n\mathbf{p}_n^T\nabla f(\mathbf{x}_n) = \alpha_{n-1}\mathbf{p}_{n-1}^T\nabla f(\mathbf{x}_{n-1})$, or

$$\eta_n = \alpha_{n-1} \frac{\mathbf{p}_{n-1}^T \nabla f(\mathbf{x}_{n-1})}{\mathbf{p}_n^T \nabla f(\mathbf{x}_n)}. \tag{3.143}$$

Whatever the formula is you are going to use to determine the scale, Eqs. 3.139, 3.141, 3.142, or 3.143, remember that these are guesses (educated guesses, but nevertheless guesses). In particular, if extrapolation is involved, the resulting η might be off scale, and it is safe to set a cap $\eta_{\max}$. It is good to have an idea from your physical problem at hand, whether the step from $\mathbf{x}$ to $\mathbf{x} - \eta\mathbf{p}$ is reasonable.[29]

Interpolation/extrapolation: one way of approximating a function $f(x)$ is by a parabola, $f_2(x; x_1, x_2) = a\,(x - x_1)^2 + b\,(x - x_1) + c$. (Notation: x is the variable, x_1 and x_2 are constants). Two function values $f(x_1)$ and $f(x_2)$, and one derivative $f'(x_1)$ suffice to define the parabola uniquely.

$$a = \frac{f(x_2) - f(x_1) - (x_2 - x_1)\, f'(x_1)}{(x_2 - x_1)^2}; \quad b = f'(x_1); \quad c = f(x_1). \tag{3.144}$$

Note that the parabola is constructed such, that $f_2(x_1; x_1, x_2) = f(x_1)$, $f_2'(x_1; x_1, x_2) = f'(x_1)$, and $f_2(x_2; x_1, x_2) = f(x_2)$. This is called a *quadratic interpolation*. The stationary point of this parabola is given by

$$x_m - x_1 = -\frac{f'(x_1)\,(x_2 - x_1)^2}{2\,[f(x_2) - f(x_1) - (x_2 - x_1)\, f'(x_1)]}. \tag{3.145}$$

In principle, this can be a minimum or a maximum, but if $f'(x_1) < 0$, it is a minimum. Now define a function $F(\alpha) = f(x - \alpha p)$. Suppose we have the information at two points α_0 and α_1, then the minimum of $F_2(\alpha; \alpha_0, \alpha_1)$ is given by

$$\alpha_{\min} = \frac{(\alpha_1 - \alpha_0)^2\, p f'(x - \alpha_0 p)}{2\,[f(x - \alpha_0 p) - f(x - \alpha_0 p) + (\alpha_1 - \alpha_0)\, p f'(x - \alpha_0 p)]}. \tag{3.146}$$

You can also try *cubic interpolation*. Given the four pieces of information $f(x_1)$, $f(x_2)$, $f'(x_1)$ and $f'(x_2)$ one define a cubic function $f_3(x; x_1, x_2) = a\,(x - x_1)^3 + b\,(x - x_1)^2 + c\,(x - x_1) + d$ in a unique way such, that $f_3(x_1; x_1, x_2) = f(x_1)$, $f_3'(x_1; x_1, x_2) = f'(x_1)$, $f_2(x_2; x_1, x_2) = f(x_2)$, $f_3'(x_2; x_1, x_2) = f'(x_2)$.

$$\begin{aligned} a &= \frac{-2\,(f(x_2) - f(x_1)) + (x_2 - x_1)\,(f'(x_1) + f'(x_2))}{(x_2 - x_1)^3}; \\ b &= \frac{3\,(f(x_2) - f(x_1)) - (x_2 - x_1)\,(2f'(x_1) + f'(x_2))}{(x_2 - x_1)^2}; \quad c = f'(x_1); \quad d = f(x_1). \end{aligned} \tag{3.147}$$

'The stationary points of this cubic function are given by the roots x_m of $f_3'(x; x_1, x_2) = 3a\,(x - x_1)^2 + 2b\,(x - x_1) + c$, or

$$x_m - x_1 = \frac{-b \pm \left[b^2 - 3ac\right]^{\frac{1}{2}}}{3a}. \tag{3.148}$$

We need $b^2 > 3ac$, otherwise the cubic interpolation does not work. For a minimum, we want the curvature at x_m to be positive, i.e., $f''(x_m) = 6a\,(x_m - x_1) + 2b > 0$. This holds only for the "+" in Eq. 3.148.

---

[29] In geometry optimizations of molecular structures, for instance, $\mathbf{x}$ comes with a natural scale: Å. The scale of $\eta\mathbf{p}$ should then be fractions of Å.

Again, define a function $F(\alpha) = f(x - \alpha p)$. Suppose we have the information at two points α_0 and α_1, then the minimum of $F_3(\alpha; \alpha_0, \alpha_1)$ is given by

$$\begin{aligned}
\alpha_{\min} &= (\alpha_1 - \alpha_0) \frac{pf'(x - \alpha_1 p) - a_1 - a_2}{pf'(x - \alpha_0 p) - pf'(x - \alpha_1 p) + 2a_2} \quad \text{with} \\
a_1 &= pf'(x - \alpha_0 p) + pf'(x - \alpha_1 p) + 3\,(\alpha_1 - \alpha_0)^{-1}\,[f(x - \alpha_1 p) - f(x - \alpha_0 p)] \quad \text{and} \\
a_2 &= \left[a_1^2 - pf'(x - \alpha_0 p) pf'(x - \alpha_1 p)\right]^{\frac{1}{2}}.
\end{aligned} \tag{3.149}$$

A simple "search" Having obtained a sensible scale η, either via interpolation/extrapolation, Eqs. 3.141 or 3.142, or via the simpler rules, Eqs. 3.139 or 3.143, the simplest line search algorithm is

$$\text{set } \eta; \quad \text{if } F(\eta) < F(0) + c_1 \eta F'(0) \text{ and } |F'(\eta)| < c_2\,|F'(0)|\,; \text{ then success; else failure;} \tag{3.150}$$

Note that this is not much of a search; we just try out the one point η.

Cost Calculating η via Eqs. 3.141 or 3.142 involves one evaluation of the function $F(\alpha_1)$ and its derivative $F'(\alpha_1)$, as the information at $\alpha_0 = 0$ is already available from the previous iteration. Applying Eq. 3.143 one has all information already available from the previous two iterations. Obviously, applying Eq. 3.139, one does not need any additional information. Having set η, applying Eq. 3.150 involves one function evaluation $F(\eta)$ and one derivative evaluation $F'(\eta)$.

Robustness Obviously, the algorithm is anything but robust. Nevertheless, it is worth a try, as it is the cheapest one around. All the algorithms discussed below are more expensive in terms of function and gradient evaluations.

3.6.3 Damage control: bracketing

If our first trial, Eq. 3.150, fails, things become complicated. Common wisdom says that, before starting a more detailed search, one should first establish an interval $[b; c]$ that is guaranteed to contain a minimum of $F(\alpha)$. Determining such an interval is called *bracketing*. The basic idea is that first you focus on obeying the Armijo rule, Eq. 3.138. In a second step, you narrow down the interval trying to obey the Wolfe condition, Eq. 3.136. This is called a *bracketed search*. The second step is discussed in the next section, which also contains the practical algorithms. In the present section I focus on the bracketing stage, and explain step-by-step what it is about.

A simple algorithm sets the initial bracket to $[0; \tau_1\eta]$, and finds a right bracket obeying the the Armijo rule, by a technique called *backtracking*

$$\begin{aligned}
&\tau_1 = 1;\ \tau_2 = 1/2;\ \alpha_0 = 0;\ \alpha_1 = \tau_1\eta;\ \text{if } F(\alpha_1) > F(0) + c_1\alpha_1 F'(0); \\
&\text{then do } n = 2, \ldots\ \alpha_n = \tau_2 \alpha_{n-1};\ \text{until } F(\alpha_n) < F(0) + c_1\alpha_n F'(0);
\end{aligned} \tag{3.151}$$

We *decrease* the value $\alpha_n \leftarrow \alpha_1 \tau_2^{n-1}$ geometrically. At exit, α_n obeys the Armijo rule, and $\alpha_i > \alpha_n;\ i = 1, ..., n-1$ do not. Of course, in a real program you have to put an upper bound $n_{\max}$ to the number of iterations, or set a minimum acceptable value for α, to ensure that the do-loop finishes. I omit that sort of practicalities in the following. There is nothing magical about the values for τ_1 and τ_2. They are *not guaranteed* to work, and if they work, they are not guaranteed to work optimal. If you have reasons to choose different values, please do so. Backtracking is most popular in combination with quasi-Newton algorithms (with $\eta = 1$).

I don't think there is anything magical about them. One can write the same algorithm as *forward tracking*

$$\begin{aligned} &\tau_1 = 1/64;\ \tau_2 = 2;\ \alpha_0 = 0;\ \alpha_1 = \tau_1\eta;\ \text{if } F(\alpha_1) < F(0) + c_1\alpha_1 F'(0); \\ &\text{then do } n = 2, ...\ \ \alpha_n = \tau_2\alpha_{n-1};\ \text{until } F(\alpha_n) > F(0) + c_1\alpha_n F'(0); \end{aligned} \tag{3.152}$$

Now we *increase* the value $\alpha_n \leftarrow \alpha_1 \tau_2^{n-1}$ geometrically. On exit, α_n does not obey the Armijo rule, and $\alpha_i < \alpha_n;\ i = 1, ..., n-1$ do. Again, you are free to chose other values for τ_1 and τ_2, if you feel like it. The choice between backtracking and forward tracking is largely heuristic, depending on which value $\tau_1\eta$ of your initial right bracket you think is most reasonable.

At every trial value α_n you are calculating the function value $F(\alpha_n)$, and probably also the derivative $F'(\alpha_n) = dF(\alpha_n)/d\alpha$, so it feels a bit like a shame not to do anything with the latter information. One can replace the line $\alpha_n = \tau_2\alpha_{n-1}$ in the backtracking algorithm, Eq. 3.151, by the more sophisticated

$$\begin{aligned} &\text{decrease } \alpha_n(\alpha_{n-1}, \alpha_{n-2}): \quad \tau_2 = 1/2; \\ &\qquad \alpha_n = \min_\alpha\ F_3(\alpha; \alpha_{n-1}, \alpha_{n-2});\ \ \text{if } \alpha_n > \alpha_{n-1} \text{ or } \alpha_n < \alpha_0;\ \text{then } \alpha_n = \tau_2\alpha_{n-1}; \end{aligned} \tag{3.153}$$

where we look for the minimum of the cubic interpolation/extrapolation function, Eq. 3.142, based upon the last two calculated points, α_{n-1} and α_{n-2}, or, if that does not work, the quadratic one, Eq. 3.141. However, only in case the new point α_n is smaller than the previous one, α_{n-1}, we accept it; after all we are performing backtracking. Otherwise we use the simple rule. Likewise, the corresponding rule in the forward tracking algorithm, Eq. 3.152, can be replaced with

$$\begin{aligned} &\text{increase } \alpha_n(\alpha_{n-1}, \alpha_{n-2}): \quad \tau_2 = 2; \\ &\qquad \alpha_n = \min_\alpha\ F_3(\alpha; \alpha_{n-1}, \alpha_{n-2});\ \ \text{if } \alpha_n < \alpha_{n-1} \text{ or } \alpha_n < \alpha_0;\ \text{then } \alpha_n = \tau_2\alpha_{n-1}; \end{aligned} \tag{3.154}$$

In the algorithms of Eqs. 3.151 and 3.152 the left bracket $\alpha_0 = 0$ is kept constant, whereas the right bracket α_n is updated. If we also keep track of $F'(\alpha_n) = dF(\alpha_n)/d\alpha$, we can adapt both brackets, and potentially create a smaller interval. Suppose we have an interval $[b; c]$, where $F(b) < F(0) + c_1 b F'(0)$ obeys the Armijo rule, but $F(c) > F(0) + c_1 c F'(0)$ does not, and furthermore $F'(b) < 0$. Then the interval $[b; c]$, contains a(t least one) point $F(\alpha)$ that obeys both the Armijo rule, Eq. 3.138, and the Wolfe condition, Eq. 3.136, provided $0 < c_1 < c_2 < 1$. This means that this interval contains a(t least one) minimum. The same is actually true for an interval $[b; c']$ under the conditions $F(b) < F(0) + c_1 b F'(0)$, $F'(b) < 0$, and $F(c') > F(b)$. For a picture, see Fig. 3.4.

Proof: a geometrical construction. The "Armijo line" is defined by $L_0(\alpha) = F(0) + c_1\alpha F'(0)$. Drawing a line L_1 parallel to L_0 that goes through the point $(b, F(b))$ gives $L_1(\alpha) = F(b) + c_1(\alpha - b)F'(0)$, The line L_1 has to cut the function F in a point $d < c$. Then we are back in the situation of the proof at the end of Sec. 3.6.2, with the interval $(b; d)$ playing the role of $(0; a)$. We can actually apply the same reasoning if $F(b) < F(0) + c_1 bF'(0)$, and $F'(b) < 0$, but $F(c') > F(b)$, see Fig. 3.4.

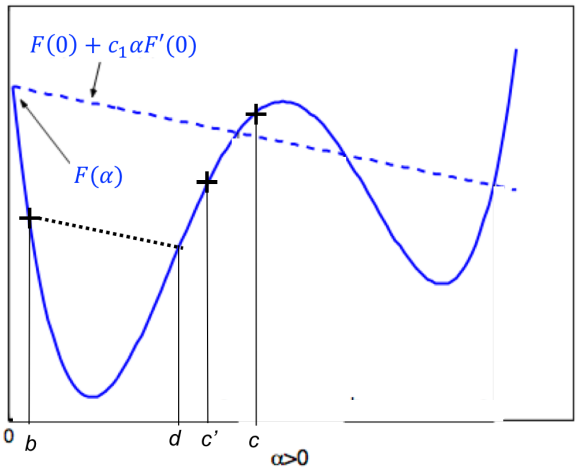


Figure 3.4: The blue dashed line indicates the Armijo rule $L_0(\alpha) = F(0) + c_1\alpha F'(0)$. The black dotted line indicates the line $L_1(\alpha) = F(b) + c_1(\alpha - b)F'(0)$. The minimum is contained in the interval $[b; c]$ and in the interval $[b; c']$. The proof of this follows that in Sec. 3.6.2, using the interval $(b; d)$.

If $F(b) < F(0) + c_1 bF'(0)$, and $F'(b) < 0$, but $F(c'') < F(b)$ and $F(c'') < F(0) + c_1 bF'(0)$, then we have a further possibility to narrow down the interval, provided $F'(c'') > 0$. Also then the interval $[b; c'']$, contains a(t least one) point $F(\alpha)$ that obeys both the Armijo rule, Eq. 3.138, and the Wolfe condition, Eq. 3.136, provided $0 < c_1 < c_2 < 1$, see Fig. 3.5.

Proof: $F'(b) < 0$ and $F'(c'') > 0$ means that $F'(\alpha) = 0$ for a $b < \alpha < c$, and that $F(\alpha)$ is a minimum there. There may be other roots, and some of them may represent a maximum or an inflection point of F, but there is at least one minimum. In this minimum the Wolfe condition is obeyed; furthermore $F(\alpha) < F(b)$, so the Armijo rule is obeyed as well.

Note that, in the cases depicted in Fig. 3.4, $F(b) < F(c)$ and $F(b) < F(c')$, in Fig. 3.5, $F(c'') < F(b)$.

I modify the forward tracking algorithm.

$$\begin{aligned} &\tau_1 = 1/64;\ \alpha_0 = 0;\ \alpha_1 = \tau_1\eta; \\ &\text{then do } n = 1, \ldots \text{ if } F(\alpha_n) > F(0) + c_1\alpha_n F'(0) \text{ or } F(\alpha_n) > F(\alpha_{n-1});\ \text{then stop}; \\ &\qquad \text{if } F'(\alpha_n) > 0;\ \text{then stop};\ \ \text{increase } \alpha_n(\alpha_{n-1}, \alpha_{n-2}); \end{aligned} \tag{3.155}$$

On exit, the interval $[\alpha_{n-1}; \alpha_n]$ contains a(t least one) point that obeys the Armijo-Wolfe conditions. Note that, on exit, either $F(\alpha_{n-1}) < F(\alpha_n)$ and $F'(\alpha_{n-1}) < 0$, Fig. 3.4, or

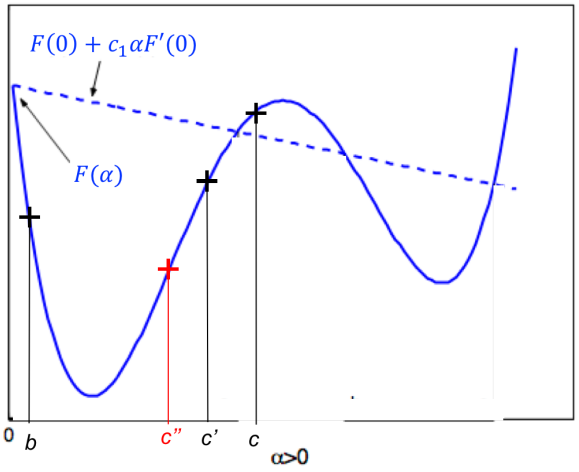


Figure 3.5: The blue dashed line indicates the Armijo rule $L_0(\alpha) = F(0) + c_1\alpha F'(0)$. We have $F(c'') < F(b)$ and $F'(c'') > 0$. The minimum is contained in the interval $[b; c'']$.

$F(\alpha_{n-1}) > F(\alpha_n)$ and $F'(\alpha_n) > 0$ (and still $F'(\alpha_{n-1}) < 0$), Fig. 3.5. If I define my brackets as $b = \alpha_{n-1}$ and $c = \alpha_n$, then I have to keep distinguishing between the situations depicted in Figs. 3.4 and 3.5.

It is wiser to define brackets b and c according to

$$\text{if } F(\alpha_{n-1}) < F(\alpha_n);\ \text{then } b = \alpha_{n-1};\ c = \alpha_n;\ \text{else } b = \alpha_n;\ c = \alpha_{n-1}; \tag{3.156}$$

We loose the strict order, as we can have $b < c$ or $b > c$. The advantage is that now we always have

$$F(b) < F(c) \text{ and } F'(b)\,(c - b) < 0. \tag{3.157}$$

Cost All the algorithms in this section, Eqs. 3.151-3.155, involve one function evaluation $F(\alpha_n)$ and one derivative evaluation $F'(\alpha_n)$ per iteration. Therefore, if the algorithm stops after n iterations, its costs n function and derivative evaluations. It is vital to keep n as small as possible. Monitor the values obtained while running the algorithm, and don't be shy to use heuristic shortcuts particular to your specific problem.

Robustness The algorithms are robust in the sense that the algorithms supply an interval that contains the requested α. That is, provided of course that there is a minimum in $F(\alpha)$. We have made sure of this by checking Eq. 3.134 on forehand, and by our assumption that we are dealing with a well-behaved function (continuous, differentiable, bounded, lacking anything the mathematicians might come up with to spoil the fun).

3.6.4 Bracketed searches

After the bracketing stage we have an interval $[b; c]$ that contains a(t least one) point α, where the function $F(\alpha)$ obeys the Armijo-Wolfe criteria. Because we have a well-behaved function $F(\alpha)$, points that obey these criteria form a continuous interval, or multiple of such intervals. We will try and find such an interval by narrowing down the starting interval $[b; c]$. This is called a *bracketed search.*

Simple backtracking Because even the most efficient of such searches cost additional function and derivative evaluations, it is advised to test whether such searches are really necessary. The algorithms of Eqs. 3.151-3.155 can be modified in a simple way such, that each time a point α_n is found that obeys the Armijo rule, also test whether it obeys the Wolfe condition. If that is the case, we are finished, and we don't need to search any further. Backtracking, Eq. 3.151, can be modified to such a full line search algorithm. It is called a *backtracking line search*

$$\begin{aligned} &\alpha_0 = 0;\ \alpha_1 = \eta;\ \text{if } F(\alpha_1) < F(0) + c_1\alpha_1 F'(0) \text{and } |F'(\alpha_n)| < c_2\,|F'(0)|\,;\ \text{then stop;} \\ &\text{else do } n = 2, \ldots \text{ decrease } \alpha_n(\alpha_{n-1}, \alpha_{n-2})\ ; \\ &\qquad \text{until } F(\alpha_n) < F(0) + c_1\alpha_n F'(0) \text{ and } |F'(\alpha_n)| < c_2\,|F'(0)|\,; \end{aligned} \tag{3.158}$$

The algorithm is far from a guarantee to success. It is very popular in combination with a quasi-Newton method, where a large value for c_2 is allowed, e.g. $c_2 = 0.9$. The chances of success are then quite high.

Cost and robustness The algorithm involves one function evaluation $F(\alpha_n)$ and one derivative evaluation $F'(\alpha_n)$ per iteration. Therefore, if the algorithm stops after n iterations, its costs n function and derivative evaluations. The algorithm is of course far from robust.

Zooming in If c_2 is quite small, e.g., $c_2 = 0.1$, as is typical for line searches in combination with the conjugate gradient method, then the algorithm of Eq. 3.158 often fails, and we have to come up with something more sophisticated. Let us start by modifying the bracketing algorithm of Eq. 3.155 such, that we make use of success, if it is given to us

$$\begin{aligned} &\tau_1 = 1/64;\ \alpha_0 = 0;\ \alpha_1 = \tau_1\eta; \\ &\text{then do } n = 1, \ldots \text{ if } F(\alpha_n) > F(0) + c_1\alpha_n F'(0) \text{ or } F(\alpha_n) > F(\alpha_{n-1});\ \text{then stop;} \\ &\qquad \text{if } |F'(\alpha_n)| < c_2\,|F'(0)|\,;\ \text{then success;} \\ &\qquad \text{if } F'(\alpha_n) > 0;\ \text{then stop;}\ \ \text{increase } \alpha_n(\alpha_{n-1}, \alpha_{n-2}); \end{aligned} \tag{3.159}$$

The bracketing algorithm now either brings success, meaning a point α_n that obeys the Armijo-Wolfe conditions, or it gives an interval $[\alpha_{n-1}, \alpha_n]$ where such a point can be found.

If the latter is the case, we need to narrow down this interval such, that the points within the interval obey the Armijo-Wolfe conditions. This stage is called *zooming in,* for obvious reasons.

$$\text{if } F(\alpha_{n-1}) < F(\alpha_n);\ b_0 = \alpha_{n-1};\ c_0 = \alpha_n;\ \text{else } b_0 = \alpha_n;\ c_0 = \alpha_{n-1}; \tag{3.160}$$

$$\text{do } m = 1, \ldots;\ \ \text{interpolate } \alpha_m(b_{m-1}, c_{m-1}); \tag{3.161}$$

$$\text{if } F(\alpha_m) > F(0) + c_1\alpha_m F'(0) \text{ or } F(\alpha_m) > F(b_{m-1});\ \text{then } b_m = b_{m-1};\ c_m = \alpha_m; \tag{3.162}$$

$$\text{else } b_m = \alpha_m;\ \text{if } F'(\alpha_n)\,(c_{m-1} - b_{m-1}) > 0;\ \text{then } c_m = b_{m-1};\ \text{else } c_m = c_{m-1}; \tag{3.163}$$

$$\text{if } |F'(\alpha_m)| < c_2\,|F'(0)|\,;\ \text{then success;} \tag{3.164}$$

The initial bracket is set in Eq. 3.160, according to Eq. 3.156, such that $F(b_0)$ contains the current lowest function value, Eq. 3.157. A new point α_m is found in Eq. 3.161 by interpolation. If it obeys either of the conditions shown in Fig. 3.4, then in Eq. 3.162 the c-bracket is adapted to the new point, and the b-bracket is untouched. If not, then $F(\alpha_m) < F(0) + c_1\alpha_m F'(0)$ and $F(\alpha_m) < F(b_{m-1})$, as in Fig. 3.5. We have found a new lowest function value, and adapt the b-bracket accordingly in Eq. 3.163. The c-bracket is adapted such, that the minimum stays bracketed, using Eq. 3.157. Finally, success is tested in Eq. 3.164.

The interpolation routine is similar to Eqs. 3.153 and 3.154

$$\begin{aligned} &\text{interpolate } \alpha_n(b,c): \\ &\qquad \alpha_n = \min_{\alpha} \; F_3(\alpha; b, c); \\ &\qquad \text{if } \alpha_n < \min b, c \text{ or } \alpha_n > \max b, c; \text{ then } \alpha_n = (b+c)/2; \end{aligned} \tag{3.165}$$

We use cubic interpolation (or, if that should not work, quadratic interpolation). If that should give a point outside the bracket, then we simply use bisection.

Cost and robustness The bracketing algorithm involves one function evaluation $F(\alpha_n)$ and one derivative evaluation $F'(\alpha_n)$ per iteration. If the algorithm stops after n iterations, its costs n function and derivative evaluations. The subsequent zooming-in algorithm involves one function evaluation $F(\alpha_m)$ and one derivative evaluation $F'(\alpha_m)$ per iteration. If that algorithm stops after m iterations, its costs m function and derivative evaluations. The algorithm is robust in the sense that a point is found that obeys the Armijo-Wolfe conditions. The full algorithm can be time-consuming, as one needs a total number of $n+m$ iterations. That number of iterations should be carefully monitored. If it becomes too large, it is maybe advised to increase the parameter c_2.

3.7 Non-linear problems*

The way all the optimization routines discussed so far (steepest descent, conjugate gradient, (quasi-)Newton) solve the minimization problem, is by searching for the point $\mathbf{x}^{\min}$, where

$$\nabla f(\mathbf{x}^{\min}) = 0. \tag{3.166}$$

This is a set of non-linear equations. For a quadratic function $f(\mathbf{x})$, Eq. 3.2, we get

$$\nabla f(\mathbf{x}^{\min}) = \mathbf{A}\mathbf{x}^{\min} - \mathbf{b} = 0, \tag{3.167}$$

which is a set of linear equations. The subject of optimization is therefore strongly linked to that of solving (non-)linear problems. In the literature, optimizing a quadratic function is often called *linear optimization,* whereas optimizing a general function is often called *non-linear optimization,* referring to Eqs. 3.167 and 3.166, respectively.

The reverse is not necessarily true, i.e., not every (non-)linear problem can be interpreted as an optimization problem. Still, we can learn from the techniques used in solving the latter. Suppose we have a set of (non-)linear equations of the type

$$\mathbf{A}\mathbf{x} = \mathbf{b}(\mathbf{x}), \tag{3.168}$$

see the lecture notes on (non-)linear problems. Define a "gradient" by

$$\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}(\mathbf{x}), \tag{3.169}$$

then solving Eq. 3.166 with an optimization algorithm solves the (non-)linear problem. We don't have the function $f(\mathbf{x})$ to go with the gradient, Eq. 3.169, and we don't actually want it. Here is a nice thing: we we don't need it, provided we pick out the right optimization algorithm, and we use a line search algorithm based upon gradients only, and not on function values. Examples of the latter will be discussed in Secs. 3.7.2 and 3.7.3.

Optimization algorithms have of course their restrictions. The conjugate gradient method, Sec. 3.4.2, and the BFGS method, Sec. 3.5.2, assume that the matrix $\mathbf{A}$ is symmetric and positive definite. Broyden's original quasi-Newton methods, Sec. 3.5.1, do not assume this, and thus can be used with any non-singular matrix $\mathbf{A}$. This is true for both the good Broyden method, Sec. 3.5.1.1, as well as for the bad Broyden method, Sec. 3.5.1.2, and for Pulay iteration, Sec. 3.5.4.

However, the functions $\mathbf{b}(\mathbf{x})$ should not be too crazy. It is quite possible to come up with functions $\mathbf{b}(\mathbf{x})$ where the algorithms fail. What I said in the lecture notes on (non-)linear problems remains true: for truly non-linear equations (not derived from an optimization problem) nothing is guaranteed to work.

3.7.1 Linear problems

Optimization algorithms also work for $\mathbf{b}(\mathbf{x}) = \mathbf{b}$, a constant vector, in Eq. 3.169. So we have some additional techniques for solving the linear problem $\mathbf{A}\mathbf{x} = \mathbf{b}$. The Broyden methods (the good, the bad, and the ugly) do not seem to be used for this purpose. I don't know why, but it probably means that they have no advantages over the more traditional methods, see the lecture notes on (non-)linear problems. As discussed in those notes, Pulay iteration, which according to Sec. 3.5.4 is a Broyden method, is logically equivalent to GMRES, which is one of the most popular algorithms for iteratively solving a linear problem.

The conjugate gradient algorithm, Eqs. 3.33 and 3.34, also has clear advantages for the linear problem. It only needs one matrix-vector multiplication of $O(N^2)$ cost per cycle, and it does not necessarily need to store the $N \times N$ matrix $\mathbf{A}$ explicitly. As such, it is suited to solve large linear problems. In fact, conjugate gradient is one of the most popular algorithms around for this particular task.

Of course, in its original form, the use of conjugate gradient is restricted to symmetric positive definite matrices $\mathbf{A}$. However, given its success, people have looked at ways to generalize conjugate gradient to general matrices. The simplest extension is based upon the matrix $\mathbf{A}^T\mathbf{A}$, which is always symmetric and positive definite for any non-singular matrix $\mathbf{A}$. Then minimization of the residual

$$f(\mathbf{x}) = \frac{1}{2}\left\|\mathbf{A}\mathbf{x} - \mathbf{b}\right\|^2 = \frac{1}{2}\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{A}\mathbf{b} + \mathbf{b}^T\mathbf{b}, \tag{3.170}$$

gives

$$\nabla f(\mathbf{x}^{\min}) = \mathbf{A}^T\mathbf{A}\mathbf{x}^{\min} - \mathbf{A}\mathbf{b} = 0. \tag{3.171}$$

This is of the same form as Eq. 3.167, but with the matrix $\mathbf{A}^T\mathbf{A}$ and the vector $\mathbf{A}\mathbf{b}$, and it can be solved using conjugate gradient. The algorithm can be organized in such a way that you don't have to calculate the matrix product $\mathbf{A}^T\mathbf{A}$ explicitly, but only make use of matrix-vector products $\mathbf{A}\mathbf{x}$. The cost of the algorithm therefore stays the same.

A practical problem is that the condition number $\kappa(\mathbf{A}^T\mathbf{A}) > \kappa(\mathbf{A})$. In fact, in case $\mathbf{A}$ is symmetric, it is easy to prove that

$$\kappa(\mathbf{A}^T\mathbf{A}) = [\kappa(\mathbf{A})]^2 . \tag{3.172}$$

Proof: if $\mathbf{A}$ is symmetric, then $\kappa(\mathbf{A}) = |\lambda_{\max}/\lambda_{\min}|$, where $\lambda_{\max}$ and $\lambda_{\min}$ are the eigenvalues of $\mathbf{A}$ with the maximum absolute value, respectively the minimum absolute value, see the lecture notes on (non-)linear problems. If λ_i is an eigenvalue of $\mathbf{A}$, then λ_i^2 is an eigenvalue of $\mathbf{A}^T\mathbf{A}$, as $\mathbf{A}^T(\mathbf{A}\mathbf{x}_i) = \mathbf{A}^T(\lambda_i\mathbf{x}_i) = \lambda_i(\mathbf{A}^T\mathbf{x}_i) = \lambda_i(\mathbf{A}\mathbf{x}_i) = \lambda_i^2\mathbf{x}_i$. Moreover, we find all N eigenvalues of $\mathbf{A}^T\mathbf{A}$ this way, with $\lambda_{\max}^2$ and $\lambda_{\min}^2$ the maximum and minimum ones, respectively.

Eq. 3.172 is bad news, as the rate of convergence of conjugate gradient depends very much on the condition number, see the discussion around Eq. 3.35. This means that the problem might need serious preconditioning, Sec. 3.3.4, before conjugate gradient becomes sufficiently efficient for solving your general linear problem.

3.7.2 Bisection line search

Applying optimization routines to solve Eq. 3.166, without having access to the function $f(\mathbf{x})$, we need a corresponding line-search algorithm. Such an algorithm has to assume that one doesn't have function values $F(\alpha)$, but that one can calculate derivatives $F'(\alpha)$. Obviously, we will not worry about the Armijo rule then, Eq. 3.138, and our only convergence criterion is the Wolfe condition, Eq. 3.136. Without function values we cannot use Eqs. 3.141 and 3.142 to assess the scale η, but we have to rely on Eqs. 3.139 or 3.143.

A simple forward-tracking routine to set up an initial bracket is

$$\begin{aligned} &\tau_1 = 1/64;\ \tau_2 = 2;\ \alpha_0 = 0;\ \alpha_1 = \tau_1\eta;\ \text{if } F'(\alpha_1)F'(0) > 0; \\ &\text{then do } n = 2, \ldots\ \alpha_n = \tau_2\alpha_{n-1};\ \text{until } F'(\alpha_n)F'(\alpha_{n-1}) < 0; \end{aligned} \tag{3.173}$$

where final interval is $[\alpha_{n-1}; \alpha_n]$. Note that we go on until $F'(\alpha_n)$ and $F'(\alpha_{n-1})$ have a different sign, meaning that the interval contains a root of $F'(\alpha)$.

The classical method to zoom in on the root is the *bisection method*

$$\begin{aligned} &\text{set bracket } [\alpha_1; \beta_1]; \\ &\text{do } n = 1, \ldots\ \alpha = (\alpha_n + \beta_n)/2;\ \text{ until converged;} \\ &\quad \text{if } F'(\alpha_n)F'(\alpha) > 0 \text{ then } \alpha_{n+1} = \alpha;\ \beta_{n+1} = \beta_n;\ \text{else } \alpha_{n+1} = \alpha_n;\ \beta_{n+1} = \alpha; \end{aligned} \tag{3.174}$$

where the "set bracket" uses the final outcome of the bracketing routine, Eq. 3.173, and "until converged" is the strong Wolfe criterion, Eq. 3.136, with α your best guess of the root. Note that at all times $F'(\alpha_n)F'(\beta_n) < 0$, so the root is caught in the interval $[\alpha_n; \beta_n]$.

Robustness The algorithm is generally safe. Bisection has the root trapped between the initial bounds, and it corners it down liked a trapped animal, closing down the space to maneuver by halving it each cycle.[30]

[30] Favored by biologists, before they apply vivisection.

Cost Organized properly, bisection takes one function evaluation per cycle. Although safe, it also typically converges quite slowly. Therefore, it is rarely used in its original form. However, it can be used as a safety back-up in case a more sophisticated method fails occasionally, see, e.g., the next section.

3.7.3 Van Wijngaarden-Dekker-Brent line search

A fast and safe method for searching the root of a function $f(x)$, has been formulated by Van Wijngaarden, Dekker, and Brent.[31] It makes use of quadratic interpolation, i.e., a parabola, and it needs three points to construct the parabola. Moreover, it does not represent $f(x)$ as a quadratic function, but rather its inverse function $x(f)$. Many people appoint the Van Wijngaarden-Dekker-Brent algorithm as the *method of choice* for finding roots of functions.

One assumes that three points are available (x_0, f_0), (x_m, f_m), and (x_1, f_1), where $f_i \equiv f(x_i)$, $x_0 < x_m < x_1$, and $f_0 f_1 \leq 0$. The latter reflects that we have bracketed the root in the interval $[x_0; x_1]$. As a start one can choose x_m as the midpoint $x_m = (x_0 + x_1)/2$. The inverse quadratic function (parabola) that goes through these three points is given by

$$x(f) = \frac{(f-f_1)(f-f_m)}{(f_0-f_1)(f_0-f_m)} x_0 + \frac{(f-f_m)(f-f_0)}{(f_1-f_m)(f_1-f_0)} x_1 + \frac{(f-f_0)(f-f_1)}{(f_m-f_0)(f_m-f_1)} x_m. \tag{3.175}$$

An estimate for the root is then obtained by setting $f = 0$, which leads to

$$x = x(0) = \frac{f_1 f_m}{(f_0-f_1)(f_0-f_m)} x_0 + \frac{f_m f_0}{(f_1-f_m)(f_1-f_0)} x_1 + \frac{f_0 f_1}{(f_m-f_0)(f_m-f_1)} x_m. \tag{3.176}$$

The three points (x_0, f_0), (x_m, f_m), and (x_1, f_1) are used to narrow down the search interval, in the same way as in the bisection method, Eq. 3.174. If the calculated x falls within this narrowed-down interval, then it is taken as the new point x_m. If not, then we just take the midpoint of the narrowed-down interval as the new x_m, as in the bisection method. The basis idea is shown in Fig. 3.6.

The *Van Wijngaarden-Dekker-Brent* algorithm then reads

set bracket $[\alpha_1; \beta_1]$; $\gamma_1 = (\alpha_1 + \beta_1)/2$;
do $n = 1, \ldots$ $\gamma_{n+1} = \text{brent}(\alpha_n, \gamma_n, \beta_n)$; (3.177)
  if $F'(\alpha_n)F'(\gamma_n) > 0$ then $\alpha_{n+1} = \gamma_n$; $\beta_{n+1} = \beta_n$; else $\alpha_{n+1} = \alpha_n$; $\beta_{n+1} = \gamma_n$; (3.178)
  if $\gamma_{n+1} < \alpha_{n+1}$ or $\gamma_{n+1} > \beta_{n+1}$ then $\gamma_{n+1} = (\alpha_{n+1} + \beta_{n+1})/2$; (3.179)
until converged;

Eq. 3.177 calculates the estimate for the root according to the inverse quadratic interpolation of Eq. 3.176 (with $x_0 \leftarrow \alpha_n$, $x_1 \leftarrow \beta_n$, $x_m \leftarrow \gamma_n$, and $f \leftarrow F'$). Note that this calculation involves divisions by factors $f_i - f_j$. If $|f_i - f_j| < \epsilon$, with ϵ a small number, then it is numerically not wise to do this. So it is better to check it on forehand before trying. If it is the case, then let the function "brent" return a number outside the bounds $[\alpha_n; \beta_n]$, which will be rejected by the subsequent algorithm as a reasonable root.

[31] In the literature sometimes referred to as "Brent's method". Apparently, "van Wijngaarden" is too difficult to pronounce for English speakers. But then again, what isn't.

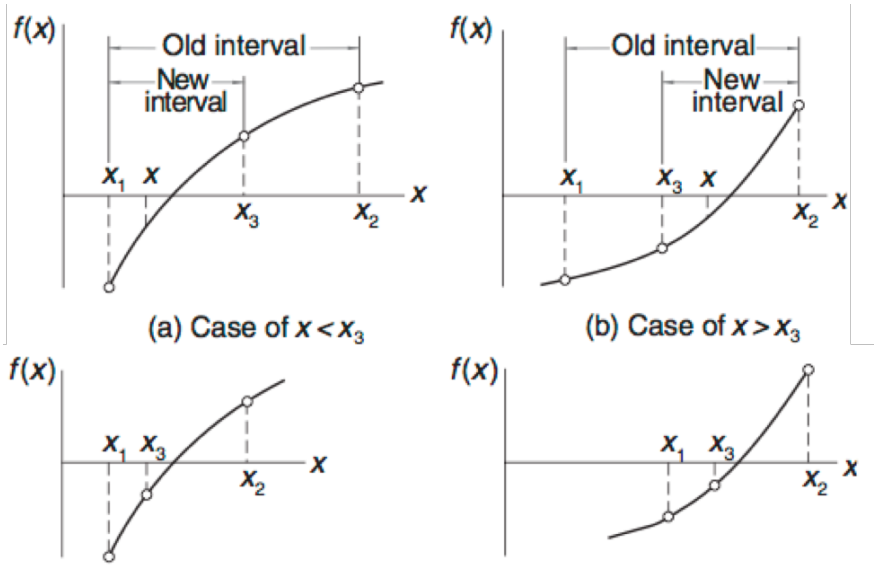


Figure 3.6: The Van Wijngaarden-Dekker-Brent method uses inverse quadratic interpolation, based upon the points $x_1 < x_3 < x_2$ to guess a new value x for the root. If x falls within the new interval, then the points for the next cycle are chosen as in the bottom two figures. If x falls outside the new interval, then x_3 is set to the midpoint $(x_1 + x_2)/2$.

Eq. 3.178 then narrows down the interval in the same way as the bisection method. Eq. 3.179 checks whether the number returned by the call to "brent" falls within the bounds of the narrowed-down interval $[\alpha_{n+1}; \beta_{n+1}]$. If not, then we just use the midpoint of that interval. In any case, γ_{n+1} is the current best guess for the root, and the "until converged criterion" is the Wolfe condition.

Robustness Like all bracketing methods, the Van Wijngaarden-Dekker-Brent method is safe, as it always keeps the root within the bracket. The inverse quadratic interpolation speeds up the convergence significantly for functions that are reasonably well-behaved.

Cost Organized properly, the Van Wijngaarden-Dekker-Brent method only takes one function evaluation per cycle. It needs some bookkeeping of points and function values to make that happen, not shown in Eqs. 3.177-3.179, but I am sure you can figure that out. Once you have the function values, the interpolation, Eq. 3.176, creates virtually no overhead.

3.8 Eigenvalue problems

Some interesting techniques for solving eigenvalue problems have been left out of the corresponding lecture notes, but it is time to discuss them now. They are derived from the fact that an eigenvalue problem can also be interpreted as an optimization problem, so the techniques used for the latter can also be applied to the former. In particular, conjugate gradient has popular in the context of electronic structure calculations [26, 9], and it has also attracted renewed attention from the mathematicians as a technique that can be used to

solve eigenvalue problems [13, 27, 28]. In Sec. 3.8.1 I will discuss how to apply the steepest descent and conjugate gradient algorithms to find the lowest eigenvalue of a matrix. You can see how this can be useful in physics eigenvalue problems, in particular those derived from quantum mechanics. In Sec. 3.8.4 I will show you how to subsequently find higher eigenvalues, whereas Sec. 3.8.5 discusses a technique to find a block of lowest eigenvalues in one go.

3.8.1 Lowest eigenvalue; Rayleigh quotient minimization

Conjugate gradient or steepest descent can be used to solve eigenvalue problems. Recall from the lecture notes on eigenvalue problems that the lowest eigenvalue $\lambda^{(1)}$ of a matrix $\mathbf{A}$ is given by

$$\lambda^{(1)} = \min_{\mathbf{x}} R(\mathbf{x}) \quad \text{with } R(\mathbf{x}) = \frac{\mathbf{x}^T\mathbf{A}\mathbf{x}}{\mathbf{x}^T\mathbf{x}}, \tag{3.180}$$

the Rayleigh quotient, see Sec. 2.7.0.1. In other words, we have a minimization problem. Clearly $R(\mathbf{x})$ is more complicated than a quadratic function, so minimizing $R(\mathbf{x})$ is more complicated than solving a linear problem, Eqs. 3.1-3.4. To run the conjugate gradient or steepest descent algorithms, we need the gradient

$$\nabla R(\mathbf{x}) = 2\frac{\mathbf{A}\mathbf{x} - R(\mathbf{x})\mathbf{x}}{\mathbf{x}^T\mathbf{x}}. \tag{3.181}$$

Proof: write $\mathbf{x}^T\mathbf{A}\mathbf{x} = \mathbf{x}^T\mathbf{x}R(\mathbf{x})$, take the derivatives $\partial/\partial x_i$ of both sides and use the chain rule, to get $2\mathbf{A}\mathbf{x} = 2\mathbf{x}R(\mathbf{x}) + \left(\mathbf{x}^T\mathbf{x}\right)\nabla R(\mathbf{x})$. This then gives Eq. 3.181.

From Eqs. 3.180 and 3.181 it follows that

$$\mathbf{x}^T\nabla R(\mathbf{x}) = 0, \tag{3.182}$$

so the gradient vector is always orthogonal to the current solution vector. We will restrict ourselves to symmetric (but not necessarily positive definite) matrices $\mathbf{A}$.[32]

3.8.2 Steepest descent

In the steepest-descent algorithm, the search direction is opposite to the gradient

$$\mathbf{p} = -\nabla R(\mathbf{x}). \tag{3.183}$$

In a line search one then minimizes the function $F(\alpha) = R\left(\mathbf{x} - \alpha\mathbf{p}\right)$. In other words, one minimizes the Rayleigh quotient R in the two-dimensional space spanned by the vectors $\mathbf{x}$ and $\mathbf{p}$. This is equivalent to solving the eigenvalue problem in this two-dimensional space and obtain the Ritz value in this space, see Sec. 2.7.0.1. As the vector $\mathbf{p} \propto \mathbf{r}$, where

$$\mathbf{r} = \mathbf{A}\mathbf{x} - R(\mathbf{x})\mathbf{x}, \tag{3.184}$$

[32] A complex Hermitian matrix would also be OK; the algorithms can easily be generalized to Hermitian matrices. I have no experience with non-Hermitian (or non-symmetric) matrices, so I don't know what to do in that case.

is the residual vector, we may as well use the vectors $\mathbf{x}$ and $\mathbf{r}$ to define the two-dimensional space in which the Ritz value should be obtained. Note that $\mathbf{x}^T\mathbf{r} = 0$, so these two vectors form an orthogonal basis of this two-dimensional space.

The *steepest descent algorithm* for finding the *lowest eigenvalue* is then formulated as

$$\mathbf{x}_1 = \mathbf{v}/\sqrt{\mathbf{v}^T\mathbf{v}}; \quad \mathbf{r}_1 = \mathbf{A}\mathbf{x}_1 - R(\mathbf{x}_1)\mathbf{x}_1;$$

$$\text{do} \quad n = 1, \ldots \quad \text{solve} \begin{pmatrix} \mathbf{x}_n^T\mathbf{A}\mathbf{x}_n & \mathbf{x}_n^T\mathbf{A}\mathbf{r}_n \\ \mathbf{r}_n^T\mathbf{A}\mathbf{x}_n & \mathbf{r}_n^T\mathbf{A}\mathbf{r}_n \end{pmatrix} \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix} = \tilde{\lambda}_n \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix}; \tag{3.185}$$

$$\mathbf{x}_{n+1} = \alpha_{n,1}\mathbf{x}_n + \alpha_{n,2}\mathbf{r}_n; \quad \mathbf{x}_{n+1} = \mathbf{x}_{n+1}/\sqrt{\mathbf{x}_{n+1}^T\mathbf{x}_{n+1}}; \tag{3.186}$$

$$\mathbf{r}_{n+1} = \mathbf{A}\mathbf{x}_{n+1} - \tilde{\lambda}_n\mathbf{x}_{n+1}; \quad \text{until converged;} \tag{3.187}$$

The 2×2 problem of Eq. 3.185 can be solved analytically, but you can also let the computer solve it, of course. One should take the lowest eigenvalue as $\tilde{\lambda}_n$ and the corresponding eigenvector $(\alpha_{n,1}, \alpha_{n,2})$. The lowest eigenvalue $\tilde{\lambda}_n$ is our current best guess for the true lowest eigenvalue $\lambda^{(1)}$, and $\mathbf{x}_{n+1}$ is our current best guess for the corresponding eigenvector $\mathbf{x}^{(1)}$. During the whole process we keep $\mathbf{x}_{n+1}$ normalized. The "until converged" criterion can be something such as $\left|\tilde{\lambda}_n - \tilde{\lambda}_{n-1}\right| < \epsilon$, with ϵ a user-set tolerance, meaning that we stop after the Ritz value does not change appreciable anymore. Alternatively, one can use something like $\sqrt{\mathbf{r}_{n+1}^T\mathbf{r}_{n+1}} < \epsilon$, , meaning that we stop after the size of the residual vector is below the threshold.

Cost The algorithm as is written in Eqs. 3.185-3.187 can be made a little bit more efficient. It actually needs only one matrix-vector multiplication $\mathbf{A}\mathbf{r}_n$ per cycle, as one can write $\mathbf{A}\mathbf{x}_{n+1} = \alpha_{n,1}\mathbf{A}\mathbf{x}_n + \alpha_{n,2}\mathbf{A}\mathbf{r}_n$, and we keep $\mathbf{A}\mathbf{x}_n$ from the previous cycle. To illustrate this, rewrite the steepest descent algorithm with storage of intermediate vectors, as

$$\mathbf{x}_1 = \mathbf{v}/\sqrt{\mathbf{v}^T\mathbf{v}}; \quad \mathbf{y}_1 = \mathbf{A}\mathbf{x}_1; \quad \mathbf{r}_1 = \mathbf{y}_1 - R(\mathbf{x}_1)\mathbf{x}_1;$$

$$\text{do} \quad n = 1, \ldots \quad \mathbf{z}_n = \mathbf{A}\mathbf{r}_n; \tag{3.188}$$

$$\text{solve} \begin{pmatrix} \mathbf{x}_n^T\mathbf{y}_n & \mathbf{x}_n^T\mathbf{z}_n \\ \mathbf{r}_n^T\mathbf{y}_n & \mathbf{r}_n^T\mathbf{z}_n \end{pmatrix} \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix} = \tilde{\lambda}_n \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix}; \tag{3.189}$$

$$\mathbf{x}_{n+1} = \alpha_{n,1}\mathbf{x}_n + \alpha_{n,2}\mathbf{r}_n; \quad \mathbf{y}_{n+1} = \alpha_{n,1}\mathbf{y}_n + \alpha_{n,2}\mathbf{z}_n; \tag{3.190}$$

$$\gamma_{n+1} = 1/\sqrt{\mathbf{x}_{n+1}^T\mathbf{x}_{n+1}}; \quad \mathbf{x}_{n+1} = \gamma_{n+1}\mathbf{x}_{n+1}; \quad \mathbf{y}_{n+1} = \gamma_{n+1}\mathbf{y}_{n+1}; \tag{3.191}$$

$$\mathbf{r}_{n+1} = \mathbf{y}_{n+1} - \tilde{\lambda}_n\mathbf{x}_{n+1}; \quad \text{until converged;} \tag{3.192}$$

The matrix-vector multiplication of Eq. 3.188, which is $O(N^2)$, is the most costly step. All other operations, such as the inner products for the matrix elements in Eq. 3.189, are $O(N)$. This means the whole algorithm scales as $O(nN^2)$, where n is the number of cycles before convergence. In principle this is not bad.

Convergence Moreover, the algorithm is relatively stable. As $\mathbf{r}_n$ represents a search direction where the Rayleigh quotient should go down, the Ritz value $\tilde{\lambda}_n$ should go down

each cycle and bring us closer to the minimum. However, you can run into the usual problem one has with steepest descent, namely that convergence can be unbearably slow, and it takes a very large number of cycles n before converge.

Let $\tilde{\lambda}_n$ and $\tilde{\lambda}_{n+1}$ be the Ritz values obtained in cycles n, respectively $n+1$. One has $\lambda^{(1)} < \tilde{\lambda}_{n+1} < \tilde{\lambda}_n$, where $\lambda^{(1)}$ is the lowest eigenvalue, which means that the Ritz value goes down in each cycle, and is an upper bound to the lowest eigenvalue. One can prove that asymptotically (for n sufficiently large) the error in the eigenvalue is bounded by [30]

$$\frac{\tilde{\lambda}_{n+1} - \lambda^{(1)}}{\tilde{\lambda}_n - \lambda^{(1)}} < K = \left(\frac{\kappa - 1}{\kappa + 1}\right)^2 \quad \text{with} \quad 1 < \kappa = \frac{\lambda^{(N)} - \lambda^{(1)}}{\lambda^{(2)} - \lambda^{(1)}}. \tag{3.193}$$

If $\kappa \gg 1$, then $K \approx 1$, and convergence tends to be very slow. This happens if the distance between the highest eigenvalue $\lambda^{(N)}$ and the lowest eigenvalue $\lambda^{(1)}$, is much larger than the distance between the second eigenvalue $\lambda^{(2)}$ and $\lambda^{(1)}$. The algorithm converges fastest if $\kappa \approx 1$, meaning $K \ll 1$, or, in other words, if the *lowest eigenvalue* $\lambda^{(1)}$ is *separated from the rest* $\lambda^{(i)}$; $i = 2, ..., N$, but at the same time, the *rest is narrowly spaced*.[33]

Note that this criterium is a lot *stricter* than the criterion we used in power iteration methods, see Sec. 2.3.1.1. There we also needed $\lambda^{(1)}$ to be well-separated from the rest to obtain a good convergence, but we did not need to care about the spacings in the rest of the spectrum. To obtain $K \ll 1$ in Eq. 3.193, one can use preconditioning to modify the spectrum accordingly. Actually, more often than not, *preconditioning is absolutely necessary* to make steepest descent a practical algorithm.

3.8.2.1 Preconditioning

As a first step, instead of minimizing the Rayleigh quotient of Eq. 3.180 directly, we might define a coordinate transformation $\mathbf{a} = \mathbf{E}\mathbf{x}$, with $\mathbf{E}$ some convenient fixed (non-singular) matrix, and minimize

$$\bar{R}(\mathbf{a}) = \frac{\mathbf{a}^T\mathbf{B}\mathbf{a}}{\mathbf{a}^T\mathbf{M}^{-1}\mathbf{a}} \quad \text{with} \quad \mathbf{B} = \mathbf{E}^{-T}\mathbf{A}\mathbf{E}^{-1} \quad \text{and} \quad \mathbf{M} = \mathbf{E}\mathbf{E}^T, \tag{3.194}$$

with respect to $\mathbf{a}$. As $\bar{R}(\mathbf{a}) = R(\mathbf{x})$, Eq. 3.180, both have the same minimum $\lambda^{(1)}$, the lowest eigenvalue of the matrix $\mathbf{A}$.

Proof: $\bar{R}(\mathbf{a}) = \mathbf{a}^T\mathbf{B}\mathbf{a}/\mathbf{a}^T\mathbf{M}^{-1}\mathbf{a} = \mathbf{x}^T\mathbf{E}^T\mathbf{B}\mathbf{E}\mathbf{x}/\mathbf{x}^T\mathbf{E}^T\mathbf{M}^{-1}\mathbf{E}\mathbf{x}$
$= \mathbf{x}^T\mathbf{E}^T\left(\mathbf{E}^{-T}\mathbf{A}\mathbf{E}^{-1}\right)\mathbf{E}\mathbf{x}/\mathbf{x}^T\mathbf{E}^T\left(\mathbf{E}^{-T}\mathbf{E}^{-1}\right)\mathbf{E}\mathbf{x} = \mathbf{x}^T\mathbf{A}\mathbf{x}/\mathbf{x}^T\mathbf{x} = R(\mathbf{x}).$

The advantage of using Eq. 3.194 instead of 3.180 as a starting point is that, with a clever choice of $\mathbf{E}$, one can have $\kappa_\mathbf{B} \ll \kappa_\mathbf{A}$, Eq. 3.193, so that the steepest descent algorithm converges faster. That can be done, but it is a nuisance to construct the matrix $\mathbf{B}$ explicitly, as it involves two matrix multiplications, which can be expensive. To avoid that, a second step is appropriate.

The matrices $\mathbf{B}$ and $\mathbf{G} = \mathbf{C}\mathbf{B}\mathbf{C}^{-1}$, where $\mathbf{C}$ is a non-singular matrix, have the same set of eigenvalues $\bar{\lambda}^{(1)} = \lambda^{(1)}, \bar{\lambda}^{(2)}, ..., \bar{\lambda}^{(N)}$.

---

[33] It can be observed that κ is the condition number of the matrix $\mathbf{A} - \lambda^{(1)}\mathbf{I}$ in the space perpendicular to the eigenvector $\mathbf{x}^{(1)}$.

Proof: If λ is an eigenvalue of $\mathbf{B}$, $\mathbf{Bx} = \lambda\mathbf{x}$, then $\mathbf{CBx} = \lambda\mathbf{Cx}$, which is equivalent to $\mathbf{G}(\mathbf{Cx}) = \lambda(\mathbf{Cx})$ with $\mathbf{G} = \mathbf{CBC}^{-1}$, so λ is an eigenvalue of $\mathbf{G}$. For proving the reverse, write $\mathbf{B} = \mathbf{C}^{-1}\mathbf{GC}$, and use the same argument to show that, if λ is an eigenvalue of $\mathbf{G}$, it is also an eigenvalue of $\mathbf{B}$. In other words $\mathbf{B}$ and $\mathbf{G}$ have the same eigenvalues.

Now use $\mathbf{C} = \mathbf{E}^{-1}$, and consider the Rayleigh quotient

$$\underline{R}(\mathbf{x}) = \frac{\mathbf{x}^T\mathbf{Gx}}{\mathbf{x}^T\mathbf{M}^{-1}\mathbf{x}} = \frac{\mathbf{x}^T\mathbf{M}^{-1}\mathbf{Ax}}{\mathbf{x}^T\mathbf{M}^{-1}\mathbf{x}}. \tag{3.195}$$

For sure $\underline{R}(\mathbf{x}) \neq R(\mathbf{x})$ in general, but both functions have the same minimum $\lambda^{(1)}$ at the same point $\mathbf{x}^{(1)}$.

Proof: the fact that the two functions have the same minimum value $\lambda^{(1)}$ follows from the previous discussion. So one has to prove that the minimum occurs at the same point $\mathbf{x}^{(1)}$. Start from $\min_{\mathbf{x}} R(\mathbf{x}) = \lambda^{(1)}$ at the point $\mathbf{x}^{(1)}$, where we have $\nabla R(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(1)}} = 0$, or $\mathbf{Ax}^{(1)} = \lambda^{(1)}\mathbf{x}^{(1)}$. But then $\mathbf{M}^{-1}\mathbf{Ax}^{(1)} = \lambda^{(1)}\mathbf{M}^{-1}\mathbf{x}^{(1)}$, which means $\lambda^{(1)} = \underline{R}(\mathbf{x}^{(1)})$ is a stationary point of $\underline{R}(\mathbf{x})$, where $\nabla\underline{R}(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^{(1)}} = 0$. But as $\lambda^{(1)}$ is a minimum, this stationary point is a minimum.

Using the function $\underline{R}(\mathbf{x})$ instead of $R(\mathbf{x})$ is called *preconditioning* and the matrix $\mathbf{M}$ is called a *preconditioner*. The idea is to choose $\mathbf{M}$ such, that $\kappa_{\mathbf{M}^{-1}\mathbf{A}} = \kappa_{\mathbf{B}} \ll \kappa_{\mathbf{A}}$, such that convergence of the steepest descent algorithm goes much faster. Obviously, the idea is practical only, if such an $\mathbf{M}$ is easily found, and $\mathbf{M}^{-1}$ is easily calculated. Often, for lack of a better idea, one uses $\mathbf{M} = \mathbf{D}$, where $D_{ij} = \delta_{ij}A_{ii}$ is the diagonal of $\mathbf{A}$. Sometimes the physics of the problem at hand gives you a concrete idea for building a preconditioner. In the projects I will give you an example where the use of preconditioning makes a spectacular difference in the convergence rate.

Implementation of preconditioning The implementation of preconditioning in the steepest descent algorithm is actually really easy. The gradient of $\underline{R}(\mathbf{x})$ is given by

$$\nabla\underline{R}(\mathbf{x}) = 2\frac{\mathbf{M}^{-1}\left[\mathbf{Ax} - \underline{R}(\mathbf{x})\mathbf{x}\right]}{\mathbf{x}^T\mathbf{M}^{-1}\mathbf{x}}, \tag{3.196}$$

the proof of which is similar to that of Eq. 3.181. As before, we focus on the direction of the gradient (to find the minimum along this direction), which is also given by the residual vector

$$\underline{\mathbf{r}} = \mathbf{M}^{-1}\left[\mathbf{Ax} - \underline{R}(\mathbf{x})\mathbf{x}\right], \tag{3.197}$$

compare Eq. 3.184. To implement preconditioning, it seems therefore, that the only thing we have to do, is to multiply $\mathbf{r}_{n+1}$ in Eq. 3.187 by $\mathbf{M}^{-1}$.

There is however one further obstacle. Unlike Eq. 3.182, we have in general

$$\mathbf{x}^T\nabla\underline{R}(\mathbf{x}) \neq 0, \quad \text{which means } \underline{\mathbf{r}}^T\mathbf{x} \neq 0. \tag{3.198}$$

Therefore, if we search for the minimum of the Ritz value in the space spanned by $\mathbf{x}$ and $\underline{\mathbf{r}}$, we have to take care of the fact that these two vectors are not orthogonal. This is easily done by transforming the 2×2 eigenvalue problem of Eq. 3.185 into a generalized 2×2 eigenvalue problem

$$\begin{pmatrix} \mathbf{x}_n^T\mathbf{Ax}_n & \mathbf{x}_n^T\mathbf{A}\underline{\mathbf{r}}_n \\ \underline{\mathbf{r}}_n^T\mathbf{Ax}_n & \underline{\mathbf{r}}_n^T\mathbf{A}\underline{\mathbf{r}}_n \end{pmatrix} \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix} = \tilde{\lambda}_n \begin{pmatrix} \mathbf{x}_n^T\mathbf{x}_n & \mathbf{x}_n^T\underline{\mathbf{r}}_n \\ \underline{\mathbf{r}}_n^T\mathbf{x}_n & \underline{\mathbf{r}}_n^T\underline{\mathbf{r}}_n \end{pmatrix} \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix}. \tag{3.199}$$

Using the techniques of Sec. 2.7.0.1 it is easy to show that $\tilde{\lambda}_n$ is the minimum Ritz value in the space spanned by $\mathbf{x}$ and $\underline{\mathbf{r}}$. So, finally, the steepest descent algorithm incorporating preconditioning, reads

$$\mathbf{x}_1 = \mathbf{v}/\sqrt{\mathbf{v}^T\mathbf{v}}; \quad \mathbf{y}_1 = \mathbf{A}\mathbf{x}_1; \quad \mathbf{r}_1 = \mathbf{M}^{-1}\left[\mathbf{y}_1 - R(\mathbf{x}_1)\mathbf{x}_1\right];$$

$$\text{do} \quad n = 1, \ldots \quad \mathbf{z}_n = \mathbf{A}\mathbf{r}_n; \tag{3.200}$$

$$\text{solve} \begin{pmatrix} \mathbf{x}_n^T\mathbf{y}_n & \mathbf{x}_n^T\mathbf{z}_n \\ \mathbf{r}_n^T\mathbf{y}_n & \mathbf{r}_n^T\mathbf{z}_n \end{pmatrix} \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix} = \tilde{\lambda}_n \begin{pmatrix} \mathbf{x}_n^T\mathbf{x}_n & \mathbf{x}_n^T\mathbf{r}_n \\ \mathbf{r}_n^T\mathbf{x}_n & \mathbf{r}_n^T\mathbf{r}_n \end{pmatrix} \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix}; \tag{3.201}$$

$$\mathbf{x}_{n+1} = \alpha_{n,1}\mathbf{x}_n + \alpha_{n,2}\mathbf{r}_n; \quad \mathbf{y}_{n+1} = \alpha_{n,1}\mathbf{y}_n + \alpha_{n,2}\mathbf{z}_n; \tag{3.202}$$

$$\gamma_{n+1} = 1/\sqrt{\mathbf{x}_{n+1}^T\mathbf{x}_{n+1}}; \quad \mathbf{x}_{n+1} = \gamma_{n+1}\mathbf{x}_{n+1}; \quad \mathbf{y}_{n+1} = \gamma_{n+1}\mathbf{y}_{n+1}; \tag{3.203}$$

$$\mathbf{r}_{n+1} = \mathbf{M}^{-1}\left[\mathbf{y}_{n+1} - \tilde{\lambda}_n\mathbf{x}_{n+1}\right]; \quad \text{until converged;} \tag{3.204}$$

The changes with respect to the steepest descent algorithm without preconditioning, Eqs. 3.188-3.192, are indicated in red.

3.8.3 Conjugate gradient (RQMIN)

If steepest descent does not converge sufficiently fast, even with preconditioning, the remedy is the same as that discussed in Sec. 3.3, i.e., use conjugate search directions instead of gradients. The search directions $\mathbf{p}_i$ introduced in conjugate gradient are constructed by manipulating the gradients, Eq. 3.57. Each added search direction is conjugate to all previous ones, and increases the dimension of the search space by one. After n cycles you effectively solve the eigenvalue problem in the space spanned by $\{\mathbf{x}_1, \mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n\}$. Conjugate gradient therefore is a subspace technique, see the lecture notes on eigenvalue problems. It is a very nice one, because, in contrast to the methods we discussed in those lecture notes, conjugate gradient does not require explicitly solving an eigenvalue problem in the whole subspace in its full dimension.

In addition to the search directions, we also need step size parameters α_n in conjugate gradient, Eq. 3.56. For minimizing the Rayleigh quotient, there is a particular nice way to find those, while avoiding any explicit line searches of the type discussed in Sec. 3.6. Finding the minimum of the Rayleigh quotient $F(\alpha) = R(\mathbf{x} - \alpha\mathbf{p})$ in a line search, amounts to solving the eigenvalue problem in the restricted two-dimensional space spanned by the vectors $\mathbf{x}$ and $\mathbf{p}$, see Sec. 2.7.0.1.

$$\begin{pmatrix} \mathbf{x}^T\mathbf{A}\mathbf{x} & \mathbf{p}^T\mathbf{A}\mathbf{x} \\ \mathbf{p}^T\mathbf{A}\mathbf{x} & \mathbf{p}^T\mathbf{A}\mathbf{p} \end{pmatrix} \begin{pmatrix} 1 \\ -\alpha \end{pmatrix} = \lambda \begin{pmatrix} \mathbf{x}^T\mathbf{x} & \mathbf{p}^T\mathbf{x} \\ \mathbf{p}^T\mathbf{x} & \mathbf{p}^T\mathbf{p} \end{pmatrix} \begin{pmatrix} 1 \\ -\alpha \end{pmatrix}. \tag{3.205}$$

The *lowest eigenvalue* λ is then $\min_\alpha F(\alpha)$. The eigenvector is normalized such, that

$$\begin{pmatrix} 1 \\ -\alpha \end{pmatrix}^T \begin{pmatrix} \mathbf{x} \\ \mathbf{p} \end{pmatrix} = \mathbf{x} - \alpha\mathbf{p}. \tag{3.206}$$

Note that in general the vectors $\mathbf{x}$ and $\mathbf{p}$ are not orthogonal, $\mathbf{x}^T\mathbf{p} \neq 0$. So, we need an overlap matrix on the right-hand side of Eq. 3.205 to define the eigenvalue problem. This

makes it a generalized eigenvalue problem, see Sec. 2.9.[34] A 2×2 problem, generalized or not, can of course be solved analytically to find α. Alternatively, one can solve the 2×2 problem numerically; it should create no overhead.

Details: there is also another way of doing it. A line search finds the solution of $dF(\alpha)/d\alpha = -\mathbf{p}^T \nabla R\,(\mathbf{x} - \alpha\mathbf{p}) = 0$, see Eq. 3.132. Because the gradient is a relatively simple function, Eq. 3.181, this amounts to finding the positive root of the quadratic expression

$$\begin{aligned} G(\alpha) = {} & 3\alpha^2 \left[\left(\mathbf{p}^T\mathbf{A}\mathbf{x}\right)\left(\mathbf{p}^T\mathbf{p}\right) - \left(\mathbf{p}^T\mathbf{A}\mathbf{p}\right)\left(\mathbf{x}^T\mathbf{p}\right) \right] \\ & - \alpha \left[\left(\mathbf{p}^T\mathbf{A}\mathbf{p}\right)\left(\mathbf{x}^T\mathbf{x}\right) - \left(\mathbf{x}^T\mathbf{A}\mathbf{x}\right)\left(\mathbf{p}^T\mathbf{p}\right) \right] + \left(\mathbf{p}^T\mathbf{A}\mathbf{x}\right)\left(\mathbf{x}^T\mathbf{x}\right) - \left(\mathbf{x}^T\mathbf{A}\mathbf{x}\right)\left(\mathbf{p}^T\mathbf{x}\right). \end{aligned} \tag{3.207}$$

This might look menacing, but expressions such as $\mathbf{p}^T\mathbf{x}$ are just inner products. So, to obtain the factors in this expression, it only requires calculation of the matrix-vector products $\mathbf{a} = \mathbf{A}\mathbf{x}$ and $\mathbf{b} = \mathbf{A}\mathbf{p}$, and the evaluation of all inner products between $\mathbf{x}$, $\mathbf{p}$, $\mathbf{a}$, and $\mathbf{b}$. Solving Eq. 3.205 and finding the root of Eq. 3.207 should give the same result for α.

Eq. 3.205 shows what conjugate gradient is doing. With each new search direction $\mathbf{p}_n$, you are solving the eigenvalue problem in the two-dimensional space spanned by $\mathbf{x}_n$ and $\mathbf{p}_n$. The idea is that, as $\mathbf{x}_n$ already carries the information of minimizing the Rayleigh quotient in the n-dimensional space build up in the previous iterations, the (2×2) eigenvalue problem suffices to effectively minimize the Rayleigh quotient in the $(n+1)$-dimensional space.[35] Conjugate gradient finds the lowest eigenvalue of an N-dimensional matrix $\mathbf{A}$ by a sequence of solving (2×2) eigenvalue problems. As it is a subspace technique, it is much more efficient than any of the techniques related to the power method. Regarding speed and efficiency, it is competitive with the subspace methods discussed in the lecture notes on eigenvalue problems.

If $\mathbf{x}$ is an eigenvector, $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, then $a\mathbf{x}$ is also an eigenvector with the same eigenvalue, $\mathbf{A}(a\mathbf{x}) = a\mathbf{A}\mathbf{x} = a\lambda\mathbf{x} = \lambda(a\mathbf{x})$. One can say that the norm of an eigenvector is irrelevant, so we may as well look for a normalized solution vector, $\mathbf{x}^T\mathbf{x} = 1$, in Eq. 3.205. We can use this to simplify things somewhat. The search direction $\mathbf{p}$ is in general not orthogonal to $\mathbf{x}$, $\mathbf{p}^T\mathbf{x} \neq 0$. Decompose $\mathbf{p} = \mathbf{p}_\parallel + \mathbf{p}_\perp$, with $\mathbf{p}_\parallel = \mathbf{x}\left(\mathbf{x}^T\mathbf{p}\right)$ and $\mathbf{p}_\perp = \mathbf{p} - \mathbf{x}\left(\mathbf{x}^T\mathbf{p}\right)$, then $\mathbf{p}_\perp\mathbf{x} = 0$. Searching along $\mathbf{p}_\parallel$ does not bring anything, as $\mathbf{x} - \alpha\mathbf{p}_\parallel = \mathbf{x} - \alpha\mathbf{x}\left(\mathbf{x}^T\mathbf{p}\right) = \left(1 - \alpha\left(\mathbf{x}^T\mathbf{p}\right)\right)\mathbf{x}$ only changes the norm of the solution vector, which, as we said already, is irrelevant. Therefore, one should only search along $\mathbf{p}_\perp$, i.e., $\mathbf{x} - \alpha\mathbf{p}_\perp$. One can also use a normalized $\mathbf{p}'_\perp = \gamma\mathbf{p}_\perp$, with $\gamma = 1/\sqrt{\mathbf{p}_\perp^T\mathbf{p}_\perp}$, so $\mathbf{p}_\perp'^T\mathbf{p}'_\perp = 1$. All of this simplifies Eq. 3.205 to

$$\begin{pmatrix} \mathbf{x}^T\mathbf{A}\mathbf{x} & \gamma\mathbf{p}_\perp^T\mathbf{A}\mathbf{x} \\ \gamma\mathbf{p}_\perp^T\mathbf{A}\mathbf{x} & \gamma^2\mathbf{p}_\perp^T\mathbf{A}\mathbf{p}_\perp \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \lambda \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \quad \text{with } \gamma = \frac{1}{\sqrt{\mathbf{p}_\perp^T\mathbf{p}_\perp}}. \tag{3.208}$$

Note that we have a normal eigenvalue equation now, in contrast to the generalized one of

[34] Note that this is different from the steepest descent algorithm, where the position and search direction vectors are always orthogonal, $\mathbf{x}^T\mathbf{p} = 0$, so the 2×2 eigenvalue problem, Eq. 3.185, becomes a normal eigenvalue problem,

[35] If you are a mathematician, you are probably wondering whether this might guarantee to give you the solution within N cycles, with N the dimension of the problem. The answer is no. There is no eigenvalue solver I am aware of that can guarantee you that. As a physicist you don't care too much about this, as you will apply the method to cases where the dimension of the matrix N is large, and the only limit that is practical for applications, is $n \ll N$, where n is the dimension of the subspace (which corresponds to the number of iterations). If it does not work in this limit, one should try something else anyway.

Eq. 3.205. The *lowest eigenvalue* λ is still $\min_\alpha R(\mathbf{x}-\alpha\mathbf{p})$. The vector

$$\begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}^T \begin{pmatrix} \mathbf{x} \\ \mathbf{p}_\perp \end{pmatrix} = \alpha_1\mathbf{x} + \alpha_2\gamma\mathbf{p}_\perp = \mathbf{x}', \tag{3.209}$$

is the vector whose Rayleigh quotient is minimal in the two-dimensional space spanned by $\mathbf{x}$ and $\mathbf{p}$. If we normalize such, that $\sqrt{\alpha_1^2+\alpha_2^2}=1$, then $\mathbf{x}'^T\mathbf{x}'=1$. This means

$$\min_\alpha R(\mathbf{x}-\alpha\mathbf{p}) = \mathbf{x}'^T\mathbf{A}\mathbf{x}' = \lambda \tag{3.210}$$

Adapted from Eqs. 3.56 and 3.57, we can write down a *conjugate gradient algorithm* for finding the *lowest eigenvalue.* It is also called *Rayleigh quotient minimization* in the literature, or RQMIN.

$$\mathbf{x}_1 = \mathbf{v}/\sqrt{\mathbf{v}^T\mathbf{v}};\quad \mathbf{r}_1 = 2\left[\mathbf{A}\mathbf{x}_1 - R(\mathbf{x}_1)\mathbf{x}_1\right];\quad \mathbf{p}_1=\mathbf{r}_1;\ \gamma_1 = \left(\mathbf{p}_1^T\mathbf{p}_1\right)^{-\frac{1}{2}}$$

$$\text{do}\quad n=1,\ldots\quad \text{solve}\ \begin{pmatrix} \mathbf{x}_n^T\mathbf{A}\mathbf{x}_n & \gamma_n\mathbf{p}_n^T\mathbf{A}\mathbf{x}_n \\ \gamma_n\mathbf{p}_n^T\mathbf{A}\mathbf{x}_n & \gamma_n^2\mathbf{p}_n^T\mathbf{A}\mathbf{p}_n \end{pmatrix}\begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2}\end{pmatrix} = \tilde{\lambda}_n \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2}\end{pmatrix}; \tag{3.211}$$

$$\mathbf{x}_{n+1} = \alpha_{n,1}\mathbf{x}_n + \alpha_{n,2}\gamma_n\mathbf{p}_n; \tag{3.212}$$

$$\mathbf{r}_{n+1} = 2\left[\mathbf{A}\mathbf{x}_{n+1} - \tilde{\lambda}_n\mathbf{x}_{n+1}\right];\quad \text{until converged;} \tag{3.213}$$

$$\beta_n = \frac{\mathbf{r}_{n+1}^T\mathbf{r}_{n+1}}{\mathbf{r}_n^T\mathbf{r}_n};\ \mathbf{p}_{n+1} = \mathbf{r}_{n+1} + \beta_n\mathbf{p}_n; \tag{3.214}$$

$$\mathbf{p}_{n+1} = \mathbf{p}_{n+1} - \mathbf{x}_{n+1}\left(\mathbf{x}_{n+1}^T\mathbf{p}_{n+1}\right);\ \gamma_{n+1} = \left(\mathbf{p}_{n+1}^T\mathbf{p}_{n+1}\right)^{-\frac{1}{2}}; \tag{3.215}$$

The 2×2 problem of Eq. 3.211 can be solved either analytically or numerically. One should take the lowest eigenvalue as $\tilde{\lambda}_n$ and normalize the corresponding eigenvector $\sqrt{\alpha_{n,1}^2+\alpha_{n,2}^2}=1$. The lowest eigenvalue $\tilde{\lambda}_n$ is our current best guess for the true lowest eigenvalue $\lambda^{(1)}$, and $\mathbf{x}_{n+1}$ is our current best guess for the corresponding eigenvector $\mathbf{x}^{(1)}$. During the whole process $\mathbf{x}_{n+1}$ stays normalized. As before, the "until converged" criterion can be a test on the size of the residual vector, i.e., something like $\sqrt{\mathbf{r}_{n+1}^T\mathbf{r}_{n+1}} < \epsilon$.

Cost As with the steepest descent algorithm discussed in the previous section, the algorithm actually only needs one matrix-vector multiplication $\mathbf{A}\mathbf{p}_n$ per cycle, as $\mathbf{A}\mathbf{x}_{n+1} = \alpha_{n,1}\mathbf{A}\mathbf{x}_n + \alpha_{n,2}\gamma_n\mathbf{A}\mathbf{p}_n$, and we keep $\mathbf{A}\mathbf{x}_n$ from the previous cycle. As with steepest descent, you can reformulate the algorithm a bit with storage of intermediate vectors to make it more efficient. The matrix-vector multiplication is the most costly step, and is $O(N^2)$. All other operations, such as inner products, are $O(N)$. This means the whole algorithm scales as $O(nN^2)$, where n is the number of cycles before convergence. This is the same scaling as for the steepest descent algorithm. However, as conjugate gradient uses more clever search directions, it generally needs a much smaller n to achieve convergence than steepest descent.

Convergence One can prove that the error in the eigenvector with the number of iterations within the conjugate gradient algorithm, asymptotically goes as Eq. 3.35, but with κ given by [13]

$$1 < \kappa = \frac{\lambda^{(N)}-\lambda^{(1)}}{\lambda^{(2)}-\lambda^{(1)}}. \tag{3.216}$$

Note that this is the same condition number as in steepest descent, see Eq. 3.193. If this condition number is large, $\kappa \gg 1$, then $K \approx 1$ in Eq. 3.35, and convergence is very slow, whereas if $\kappa \approx 1$, then $K \ll 1$, and convergence is fast. The latter is the case if the *lowest eigenvalue* $\lambda^{(1)}$ is *separated from the rest* $\lambda^{(i)}$; $i = 2, ..., N$, but at the same time, the *rest is narrowly spaced*. Note that this is exactly the same condition as we needed for steepest descent to converge fast, see the discussion below Eq. 3.193. However, conjugate gradient will generally be much faster than steepest descent.

Following the same analysis as below Eq. 3.35, $K \sim \delta \ll 1$, if

$$\delta = \frac{\lambda^{(N)} - \lambda^{(2)}}{\lambda^{(2)} - \lambda^{(1)}} \ll 1. \tag{3.217}$$

Again, this criterium is a lot *stricter* than the criterion we used in power iteration methods, see Sec. 2.3.1.1, where we needed $\lambda^{(1)}$ to be well-separated from the rest, but we did not need to care about the spacings between the rest. We end with the same remark as for the steepest descent method. If your matrix $\mathbf{A}$ does not obey Eq. 3.217, it really pays off to try and design a *preconditioner* to make it happen. It works along the same lines as discussed for steepest descent.

3.8.4 Multiple eigenvalues*

Suppose we have found the first eigenvalue $\lambda^{(1)}$ and corresponding normalized eigenvector $\mathbf{x}^{(1)}$. Subsequently, we can look for the second eigenvalue $\lambda^{(2)}$ and corresponding eigenvector $\mathbf{x}^{(2)}$ by

$$\lambda_2 = \min_{\mathbf{x}} R_\perp(\mathbf{x}) \quad \text{with } R_\perp(\mathbf{x}) = \frac{\mathbf{y}^T\mathbf{A}\mathbf{y}}{\mathbf{y}^T\mathbf{y}} \text{ and } \mathbf{y} = \mathbf{x} - \mathbf{x}^{(1)}\mathbf{x}^{(1)T}\mathbf{x}. \tag{3.218}$$

$R_\perp(\mathbf{x})$ is the Rayleigh quotient calculated on the basis of the vector $\mathbf{y} = \mathbf{x} - \mathbf{x}^{(1)}\mathbf{x}^{(1)T}\mathbf{x}$. As $\mathbf{x}^{(1)T}\mathbf{y} = 0$, provided $\mathbf{x}^{(1)}$ is normalized, $\mathbf{x}^{(1)T}\mathbf{x}^{(1)} = 1$, we are minimizing a Rayleigh quotient of a vector $\mathbf{y}$ that is orthogonal to the first eigenvector $\mathbf{x}^{(1)}$. The minimum of that Rayleigh quotient is the second eigenvalue $\lambda^{(2)}$.

The expressions can be made a little compacter by defining

$$\mathbf{y} = \mathbf{x} - \mathbf{x}^{(1)}\mathbf{x}^{(1)T}\mathbf{x} = \mathbf{P}\mathbf{x} \quad \text{with } \mathbf{P} = \mathbf{I} - \mathbf{x}^{(1)}\mathbf{x}^{(1)T} \text{ a projection operator,} \tag{3.219}$$

which projects on the orthogonal complement of the space spanned by $\mathbf{x}^{(1)}$. We rewrite the Rayleigh quotient of Eq. 3.218 as

$$R_\perp(\mathbf{x}) = \frac{\mathbf{x}^T\mathbf{P}\mathbf{A}\mathbf{P}\mathbf{x}}{\mathbf{x}^T\mathbf{P}\mathbf{x}}, \tag{3.220}$$

as $\mathbf{P}^T = \mathbf{P}$, and $\mathbf{P}^2 = \mathbf{P}$, which is easily proved from Eq. 3.219. The gradient, analogous to 3.181, is

$$\nabla R_\perp(\mathbf{x}) = 2\frac{\mathbf{P}\mathbf{A}\mathbf{P}\mathbf{x} - R_\perp(\mathbf{x})\mathbf{P}\mathbf{x}}{\mathbf{x}^T\mathbf{P}\mathbf{x}} = \frac{2}{\mathbf{x}^T\mathbf{P}\mathbf{x}}\mathbf{P}\left[\mathbf{A}\mathbf{P}\mathbf{x} - R_\perp(\mathbf{x})\mathbf{P}\mathbf{x}\right]. \tag{3.221}$$

Basically, we can follow the algorithm of Eqs. 3.211-3.214, but use $\mathbf{P}\mathbf{x}$ instead of $\mathbf{x}$, and use the orthogonalized gradient of Eq. 3.221.

This procedure can be generalized to subsequent higher eigenvalues. Suppose we have found the eigenvalue $\lambda^{(1)}, \lambda^{(2)}, ..., \lambda^{(k-1)}$ and the corresponding normalized eigenvectors

$\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(k-1)}$, and we wish to find the next eigenvalue $\lambda^{(k)}$ and eigenvector $\mathbf{x}^{(k)}$. Analogous to Eq. 3.219, we define a projection operator

$$\mathbf{P}^{(k)} = \mathbf{I} - \sum_{i=1}^{k-1} \mathbf{x}^{(i)} \mathbf{x}^{(i)T}, \tag{3.222}$$

and use $\mathbf{P}^{(k)}$ instead of $\mathbf{P}$. We can use this operator to set up an algorithm that finds the eigenvalues and corresponding eigenvectors one at a time.

set $\mathbf{v}^{(1)}, \mathbf{v}^{(2)}, ..., \mathbf{v}^{(k)}$; $\mathbf{P}^{(1)} = \mathbf{I}$;

do $m = 1, ..., k$ (3.223)

$$\mathbf{x}_1^{(m)} = \mathbf{P}^{(m)} \mathbf{v}^{(m)} / \sqrt{\mathbf{v}^{(m)T} \mathbf{P}^{(m)} \mathbf{v}^{(m)}}; \quad \mathbf{r}_1 = 2\mathbf{P}^{(m)} \left[\mathbf{A}\mathbf{x}_1 - R(\mathbf{x}_1)\mathbf{x}_1 \right]; \quad \mathbf{p}_1 = \mathbf{r}_1; \quad \gamma_1 = \left(\mathbf{p}_1^T \mathbf{p}_1 \right)^{-\frac{1}{2}} \tag{3.224}$$

do $n = 1, ...$ solve $\begin{pmatrix} \mathbf{x}_n^T \mathbf{A} \mathbf{x}_n & \gamma_n \mathbf{p}_n^T \mathbf{A} \mathbf{x}_n \\ \gamma_n \mathbf{p}_n^T \mathbf{A} \mathbf{x}_n & \gamma_n^2 \mathbf{p}_n^T \mathbf{A} \mathbf{p}_n \end{pmatrix} \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix} = \tilde{\lambda}_n \begin{pmatrix} \alpha_{n,1} \\ \alpha_{n,2} \end{pmatrix}$; (3.225)

$$\mathbf{x}_{n+1} = \alpha_{n,1} \mathbf{x}_n + \alpha_{n,2} \gamma_n \mathbf{p}_n; \tag{3.226}$$

$$\mathbf{r}_{n+1} = 2\mathbf{P}^{(m)} \left[\mathbf{A}\mathbf{x}_{n+1} - \tilde{\lambda}_n \mathbf{x}_{n+1} \right]; \quad \text{until converged;} \tag{3.227}$$

$$\beta_n = \frac{\mathbf{r}_{n+1}^T \mathbf{r}_{n+1}}{\mathbf{r}_n^T \mathbf{r}_n}; \quad \mathbf{p}_{n+1} = \mathbf{r}_{n+1} + \beta_n \mathbf{p}_n; \tag{3.228}$$

$$\mathbf{p}_{n+1} = \mathbf{p}_{n+1} - \mathbf{x}_{n+1} \left(\mathbf{x}_{n+1}^T \mathbf{p}_{n+1} \right); \quad \gamma_{n+1} = \left(\mathbf{p}_{n+1}^T \mathbf{p}_{n+1} \right)^{-\frac{1}{2}}; \quad \text{end do } n; \tag{3.229}$$

$$\lambda^{(m)} = \tilde{\lambda}_n; \quad \mathbf{x}^{(m)} = \mathbf{x}_{n+1}; \quad \mathbf{P}^{(m+1)} = \mathbf{P}^{(m)} - \mathbf{x}^{(m)} \mathbf{x}^{(m)T}; \quad \text{end do } m; \tag{3.230}$$

The outer loop, Eqs. 3.223-3.230 find the eigenvalues $\lambda^{(m)}$; $m = 1, ..., k$ and the corresponding normalized eigenvectors $\mathbf{x}^{(m)}$; $m = 1, ..., k$, one at a time. The inner loop, Eqs. 3.225-3.229, uses the conjugate gradient minimization of the Rayleigh quotient to find a single eigenvalue $\lambda^{(m)}$ and eigenvector $\mathbf{x}^{(m)}$. The inner loop is just a copy of Eqs. 3.211-3.215.

One has to make sure that all $\mathbf{x}_n$, $\mathbf{r}_n$, and $\mathbf{p}_n$; $n = 1, 2, ...$ within the inner loop are orthogonal to the already found eigenvectors $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m-1)}$, by using the projection operator $\mathbf{P}^{(m)}$. We have used $\mathbf{P}^{(m)}$ only once in the loop, in Eq. 3.227, which ensures that $\mathbf{r}_n$ is orthogonal to all eigenvectors already obtained. As that also holds for the starting $\mathbf{x}_1$, $\mathbf{r}_1$, and $\mathbf{p}_1$, it then holds for all subsequent $\mathbf{x}_n$ and $\mathbf{p}_n$. If you fear to loose the orthogonality in these vectors due to numerical noise, one may introduce statements $\mathbf{x}_{n+1} = \mathbf{P}^{(m)} \mathbf{x}_{n+1}$ and $\mathbf{p}_{n+1} = \mathbf{P}^{(m)} \mathbf{p}_{n+1}$ at the end of Eqs. 3.226 and 3.229.

The "until converged" criterion is again a test like $\sqrt{\mathbf{r}_{n+1}^T \mathbf{r}_{n+1}} < \epsilon$. If obeyed, the algorithm jumps to Eq. 3.230 to store the newly found eigenvalue and eigenvector, and proceeds with the next one, Eq. 3.224. This type of *Rayleigh quotient minimization* algorithm is called state-by-state for obvious reasons. In the physics community it is called band-by-band, which, if you know something of solid state physics, means the same thing [9, 26]. Different flavors of this algorithm exist, and such conjugate-gradient-based algorithms have become popular instruments for calculating eigenvalues and eigenvectors of large matrices over the last two decades [13, 27, 28].

Cost and convergence The cost of obtaining the m'th eigenvalue/eigenvector is like that of obtaining the 1st eigenvalue/eigenvector. We need one matrix-vector multiplication $\mathbf{A}\mathbf{p}_n$ per inner loop cycle, which costs $O(N^2)$. Applying the projection $\mathbf{P}^{(m)}$ costs $O(mN)$, which gives little overhead as long as $m \ll N$. If n is a typical number of cycles required for the inner loop to converge, then the overall costs of finding k eigenvalues/eigenvectors is $O(knN^2)$. Note that it makes sense to do it this way only if $kn \ll N$. Typical values could be $N = 10^4$-10^6 and $k = 10^0$-10^1, so with $n = 10^1$-10^2 we would be alright.

Analogous to Eq. 3.216 the error in the eigenvector $\mathbf{x}^{(m)}$ with the number of iterations, should asymptotically go as Eq. 3.35, with κ the condition number given by

$$\kappa = \frac{\lambda^{(N)} - \lambda^{(m)}}{\lambda^{(m+1)} - \lambda^{(m)}}. \tag{3.231}$$

To get a fast convergence, Eq. 3.35, $\kappa \approx 1$ for **all** $m = 1, 2, ..., k$. So one must have a spectrum with appreciable spacing between the lowest k eigenvalues $\lambda^{(m)};\ m = 1, ..., k$, but all other eigenvalues $\lambda^{(i)};\ i = k+1, ..., N$ must be close to $\lambda^{(k)}$. One may require a special preconditioner to make this happen.[36]

3.8.5 Block Rayleigh quotient minimization (BRQMIN)*

Finding eigenvalues/eigenvectors one by one, as in the previous section, can become cumbersome if $\kappa \gg 1$. This means Eq. 3.231 is not obeyed, which typically happens if two or more of the eigenvalues $\lambda^{(m)};\ m = 1, ..., k$ are closely spaced. To avoid this problem, one can modify the algorithm the previous sections to calculate multiple eigenvalues of a symmetric matrix simultaneously. Define the function

$$f(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(k)}) = \sum_{m=1}^{k} R(\mathbf{x}^{(m)}) \quad \text{with constraints } \mathbf{x}^{(j)T}\mathbf{x}^{(m)} = 0;\ j = 1, ..., k;\ m = 1, ..., k. \tag{3.232}$$

Here $\mathbf{x}^{(m)};\ m = 1, ..., k$ mutually orthogonal N-dimensional vectors, and $R(\mathbf{x}^{(m)})$ are the corresponding Rayleigh quotients, Eq. 3.180. One keeps the vectors orthogonal, $\mathbf{x}^{(j)T}\mathbf{x}^{(m)} = 0$, as otherwise they would all collapse in the minimization process to the eigenvector belonging to the lowest eigenvalue. Without the constraints, the minimum of $f(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(k)})$ would simply be $k\lambda^{(1)}$. With the orthogonality constraints $f(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(k)})$ is minimal for

$$\min_{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(k)}} f(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(k)}) = \sum_{m=1}^{k} \lambda^{(m)}, \tag{3.233}$$

with $\lambda^{(m)};\ m = 1, ..., k$ the lowest k eigenvalues. At the minimum, the vectors $\mathbf{x}^{(m)};\ m = 1, ..., k$ represent the corresponding eigenvectors. To identify the individual eigenvalues $\lambda^{(m)}$ from the sum, Eq. 3.233, one simply has to calculate the corresponding Rayleigh quotient $R(\mathbf{x}^{(m)})$.

Once we use constraints anyway, we might as well introduce some extra ones,

$$\mathbf{x}^{(m)T}\mathbf{x}^{(m)} = 1;\ m = 1, ..., k. \tag{3.234}$$

[36] The electronic structure code VASP uses such a special preconditioner. It likely plays a major part in the success such plane wave codes have in the physics community [9, 26].

This keeps the eigenvectors normalized. One then minimizes the function

$$f(\mathbf{x}^{(1)},\mathbf{x}^{(2)},...,\mathbf{x}^{(k)}) = \sum_{m=1}^{k} \mathbf{x}^{(m)T}\mathbf{A}\mathbf{x}^{(m)} \text{ with constraints } \mathbf{x}^{(j)T}\mathbf{x}^{(m)} = \delta_{jm};\ j=1,...,k;\ m=1,...,k. \tag{3.235}$$

The following algorithm finds the k lowest eigenvalues/eigenvectors of the $(N \times N)$ matrix $\mathbf{A}$. The k vectors form the columns of a $(N \times k)$ matrix $\mathbf{X} = \left(\mathbf{x}^{(1)}\ \mathbf{x}^{(2)}\ ...,\ \mathbf{x}^{(k)}\right)$, which is called a *block*. The algorithm is called *block Rayleigh quotient minimization* or BRQMIN.

set unitary $\mathbf{X}_1$; diagonalize $\mathbf{X}_1^T\mathbf{A}\mathbf{X}_1\mathbf{U} = \mathbf{\Lambda}_1\mathbf{U}$; rotate $\mathbf{X}_1 = \mathbf{X}_1\mathbf{U}$; (3.236)

projector $\mathbf{Q}_1 = \mathbf{I} - \mathbf{X}_1\mathbf{X}_1^T$;

gradient $\mathbf{R}_1 = 2\mathbf{Q}_1\left[\mathbf{A}\mathbf{X}_1 - \mathbf{X}_1\mathbf{\Lambda}_1\right]$; $\mathbf{B}_1 = \mathrm{diag}\left(\mathbf{R}_1^T\mathbf{R}_1\right)$; (3.237)

search directions $\mathbf{P}_1 = \mathbf{R}_1$; $\mathbf{\Gamma}_1 = \mathrm{diag}\left(\mathbf{P}_1^T\mathbf{P}_1\right)^{-\frac{1}{2}}$;

do $n = 1, ...$ (3.238)

$$\mathrm{diag}\begin{pmatrix} \mathbf{X}_n^T\mathbf{A}\mathbf{X}_n & \mathbf{\Gamma}_n\mathbf{P}_n^T\mathbf{A}\mathbf{X}_n \\ \mathbf{\Gamma}_n\mathbf{P}_n^T\mathbf{A}\mathbf{X}_n & \mathbf{\Gamma}_n\mathbf{P}_n^T\mathbf{A}\mathbf{P}_n\mathbf{\Gamma}_n \end{pmatrix}\begin{pmatrix} \mathbf{C}_{n,1} \\ \mathbf{C}_{n,2} \end{pmatrix} = \begin{pmatrix} \mathbf{C}_{n,1} \\ \mathbf{C}_{n,2} \end{pmatrix}\begin{pmatrix} \tilde{\mathbf{\Lambda}}_{n,1} & \mathbf{0} \\ \mathbf{0} & \tilde{\mathbf{\Lambda}}_{n,2} \end{pmatrix};$$

$\mathbf{X} = \mathbf{X}_n\mathbf{C}_{n,1} + \mathbf{P}_n\mathbf{\Gamma}_n\mathbf{C}_{n,2}$; Gram-Schmidt $\mathbf{X}_{n+1}$; $\mathbf{Q}_{n+1} = \mathbf{I} - \mathbf{X}_{n+1}\mathbf{X}_{n+1}^T$; (3.239)

$\mathbf{R}_{n+1} = 2\mathbf{Q}_{n+1}\left[\mathbf{A}\mathbf{X}_{n+1} - \mathbf{X}_{n+1}\tilde{\mathbf{\Lambda}}_{n,1}\right]$; (3.240)

$\mathbf{B}_{n+1} = \mathrm{diag}\left(\mathbf{R}_{n+1}^T\mathbf{R}_{n+1}\right)$; until converged;

$\mathbf{P}_{n+1} = \mathbf{R}_{n+1} + \mathbf{P}_n\mathbf{B}_{n+1}\mathbf{B}_n^{-1}$; $\mathbf{P}_{n+1} = \mathbf{Q}_{n+1}\mathbf{P}_{n+1}$; $\mathbf{\Gamma}_{n+1} = \mathrm{diag}\left(\mathbf{P}_{n+1}^T\mathbf{P}_{n+1}\right)^{-\frac{1}{2}}$; (3.241)

This algorithm looks somewhat complicated, because of all the matrices involved. $\mathbf{X}_n = \left(\mathbf{x}_n^{(1)}\ \mathbf{x}_n^{(2)}\ ...,\ \mathbf{x}_n^{(k)}\right)$ are $(N \times k)$ matrices, whose $(N \times 1)$ columns $\mathbf{x}_n^{(m)}$; $m = 1, ..., k$ contain our current best guesses for the eigenvectors. Likewise, $\mathbf{R}_n = \left(\mathbf{r}_n^{(1)}\ \mathbf{r}_n^{(2)}\ ...,\ \mathbf{r}_n^{(k)}\right)$ are $(N \times k)$ matrices, whose $(N \times 1)$ columns $\mathbf{r}_n^{(m)}$; $m = 1, ..., k$ contain the gradients (or residual vectors). Then, $\mathbf{P}_n = \left(\mathbf{p}_n^{(1)}\ \mathbf{p}_n^{(2)}\ ...,\ \mathbf{p}_n^{(k)}\right)$ are $(N \times k)$ matrices, whose $(N \times 1)$ columns $\mathbf{p}^{(m)}$; $m = 1, ..., k$ contain the search directions.

$\tilde{\mathbf{\Lambda}}_{n,1}$ are *diagonal* $(k \times k)$ matrices, containing our current best guesses of the eigenvalues, whereas $\tilde{\mathbf{\Lambda}}_{n,2}$ are diagonal $(k \times k)$ matrices, whose elements we don't use any further. Likewise, $\mathbf{B}_n$ are diagonal $(k \times k)$ matrices containing the products $\mathbf{r}_n^{(m)T}\mathbf{r}_n^{(m)}$; $m = 1, ..., k$ on the diagonal, and $\mathbf{\Gamma}_n$ are diagonal $(k \times k)$ matrices containing $1/\sqrt{\mathbf{p}_n^{(m)T}\mathbf{p}_n^{(m)}}$; $m = 1, ..., k$ on the diagonal. Then, $\mathbf{C}_{n,1}$ and $\mathbf{C}_{n,2}$ are $(k \times k)$ matrices, containing the step sizes. Finally, $\mathbf{A}$ and $\mathbf{Q}_n$ are $N \times N$ matrices.

The problem is initialized in Eq. 3.236. One starts with k orthonormal vectors $\mathbf{x}_1^{(m)}$, which are assembled in the unitary matrix $\mathbf{X}_1$. One could just start from these vectors, and their corresponding Rayleigh quotients $R(\mathbf{x}_1^{(m)})$. However, it is better to start with an ordered series

$$R(\mathbf{x}_1^{(1)}) \leq R(\mathbf{x}_1^{(2)}) \leq ... \leq R(\mathbf{x}_1^{(k)}). \tag{3.242}$$

One can achieve this by diagonalizing the matrix $\mathbf{A}$ in the space spanned by $\mathbf{x}_1^{(m)}$; $i = 1, ..., k$,

i.e., diagonalizing the $(k \times k)$ matrix $\mathbf{X}_1^T \mathbf{A} \mathbf{X}_1$. Then use the eigenvectors of this problem, assembled in the $(k \times k)$ matrix $\mathbf{U}$, to rotate the initial vectors, such that the eigenvalues $\mathbf{\Lambda}_1$ now represent the Rayleigh quotients. The latter is called *subspace rotation.* Diagonalization of a $(k \times k)$ matrix should be cheap, as k is small, so it can be performed by a direct method, such as a Householder transformation + QR iteration.

The gradient is initialized in Eq. 3.237. Note that one must not calculate $\mathbf{Q}_1 \mathbf{A}$ as a product between $(N \times N)$ matrices. That would scale as $O(N^3)$, which is horrible. Because of the special form of $\mathbf{Q}_1$ (it is a projection operator), one can calculate $\mathbf{X}_1^T \mathbf{A}$, which scales as $O(kN^2)$. That is much better, as $k \ll N$. The notation "diag($\mathbf{K}$)" means: select only the diagonal elements of $\mathbf{K}$, and put the off-diagonal elements to zero. Of course, if you only need the diagonal elements, then only calculate the diagonal elements.

The step sizes of conjugate gradient are calculated in Eq. 3.238 by solving the $(2k \times 2k)$ eigenvalue problem in the space spanned by $\mathbf{x}_n^{(m)};\ m = 1, ..., k$, and $\mathbf{p}_n^{(m)};\ m = 1, ..., k$, analogous to Eq. 3.211 in the single vector case. These are then used in Eq. 3.239 to construct the new guesses for the eigenvectors $\mathbf{X}_{n+1}$. These eigenvectors are subsequently orthogonalized and normalized using Gram-Schmidt. Finally, one determines the new gradients and search directions in Eqs. 3.240 and 3.241, respectively. The "until converged" condition checks the diagonal elements

$$\sqrt{(\mathbf{B}_{n+1})_{mm}} < \epsilon;\ m = 1, ..., k. \tag{3.243}$$

The algorithms found in the literature include all sorts of clever modifications. For instance, some eigenvectors may converge much faster than others, meaning that the convergence criterion of Eq. 3.243 holds for some m, but not for others. There is no need to try and converge eigenvectors that are already converged. One can store an already converged vector $\mathbf{x}^{(m)}$, and remove the corresponding column from $\mathbf{X}_n$, making it a $N \times (k-1)$ matrix. Consequently, the matrices $\mathbf{R}_n$ and $\mathbf{P}_n$ also become $N \times (k-1)$. The eigenvalue problem of Eq. 3.238 then becomes $2(k-1) \times 2(k-1)$. Any time an eigenvector is converged, one may remove it from the problem this way, making the problem smaller and smaller. This technique is called *deflation.* It is a common technique found in many eigenvalue solvers.

There is of course a catch. The eigenvectors in the remaining problem must remain orthogonal to the ones already found. So one must define a projection operator

$$\mathbf{Q}' = \mathbf{I} - \sum_{i=1}^{\text{found}} \mathbf{x}^{(i)} \mathbf{x}^{(i)T}, \tag{3.244}$$

and replace $\mathbf{Q}_{n+1}$ in Eqs. 3.239-3.241 by $\mathbf{Q}'\mathbf{Q}_{n+1}$.

One can also combine the present algorithm with that of Sec. 3.8.4, with an inner loop that finds k eigenvalues/eigenvectors, and an outer loop that cycles over subsequent blocks of dimension k.

Cost and convergence The cost of obtaining k eigenvalues/eigenvectors is $O(nkN^2)$, which makes it comparable to the algorithms of Sec. 3.8.4.

The convergence is determined by a similar parameter as in Eq. 3.231.

$$\kappa = \frac{\lambda^{(N)} - \lambda^{(k)}}{\lambda^{(k+1)} - \lambda^{(k)}}. \tag{3.245}$$

This is a potential advantage of the present algorithm over the one in Sec. 3.8.4. To get a fast convergence in the latter case, Eq. 3.231 must be true for **all** $m = 1, 2, ..., k$. In the

present case, it only needs to be true for $m = k$. So, if you have a spectrum where

$$\lambda^{(m+1)} - \lambda^{(m)};\ m = 1, ..., k-1 \text{ is small, but } \lambda^{(k+1)} - \lambda^{(k)} \text{ is large,} \tag{3.246}$$

then the present algorithm would converge fast, whereas the algorithm of Sec. 3.8.4 would converge slow. According to a solid state physicist, Eq. 3.246 represents a spectrum with a band gap, $E_g = \lambda^{(k+1)} - \lambda^{(k)}$, which defines an insulator or a semiconductor.

One can also apply the present algorithm in the case where $q < k$ eigenvalues are required, but where Eq. 3.246 holds. Simply modify the "until converged" criterion to $\sqrt{(\mathbf{B}_{n+1})_{ii}} < \epsilon;\ i = 1, ..., q$. Cycling the extra $k - q$ states along brings some overhead, but if Eq. 3.246 holds, the number of cycles n it takes to convergence can be a lot smaller. In the end, it is the product nk that counts for the costs.

3.9 Constrained optimization: Lagrange multipliers

The method of Lagrange multipliers is part of the standard toolbox of a physicist. It has numerous applications, in classical mechanics, in electronic structure theory, or in statistical physics, for instance. As an elementary mathematical technique, it should be taught in a course on calculus. Unfortunately, it is often skipped in class, although it can be found in many textbooks on calculus [29]. Below you will find a discussion of Lagrange multipliers in the context of optimization problems [20, 21].

Up till now we have assumed that in the search for

$$\min_{x_1, x_2, ..., x_N} f(x_1, x_2, ..., x_N), \tag{3.247}$$

we can allow for any value of $x_i;\ i = 1, ..., N$ for which the function f is defined. In physical problems, quite frequently the variables are subject to constraints. For instance, in classical mechanics, one could study the motion of a system confined to a 2D plane, instead of in 3D space. In our eigenvalue problem of Eq. 3.235, we have build in the constraints by hand, but quite often the constraints are too complicated to do that.

A $(N-1)$-dimensional hyperplane in an N-dimensional space is defined by

$$g(x_1, x_2, ..., x_N) = 0. \tag{3.248}$$

This is a *constraint* that is much more general than that in Eq. 3.235.

If we could rewrite Eq. 3.248 in the form

$$x_N = h(x_1, x_2, ..., x_{N-1}), \tag{3.249}$$

then we could use this to eliminate the variable x_N and minimize the function

$$f'(x_1, x_2, ..., x_{N-1}) \equiv f\left(x_1, x_2, ..., x_{N-1}, h(x_1, x_2, ..., x_{N-1})\right), \tag{3.250}$$

as a function of $x_i;\ i = 1, ..., N-1$. Unfortunately, the step from Eq. 3.248 to Eq. 3.249 is often too difficult, if not unfeasible. In my classical mechanics example, if the 2D plane is simple, such as the surface of a sphere, one can eliminate a coordinate (the radius) by switching to the right coordinate system (spherical coordinates). However, if the 2D plane has a more complicated curvature, then finding the right coordinates (let alone differential operators) in this plane can be difficult, up to unfeasible.

We make use of the following property.

(I) At the constrained minimum $\mathbf{x}^{\min}$, the gradient $\nabla f(\mathbf{x}^{\min})$ is perpendicular to the hyperplane defined by Eq. 3.248.

Proof: consider the alternative. If $\nabla f(\mathbf{x}^{\min})$ is not perpendicular, then it has a component $\nabla f(\mathbf{x}^{\min})_{\|}$, parallel to the hyperplane. But then we could walk along the $-\nabla f(\mathbf{x}^{\min})_{\|}$ direction to decrease f, just as we did in the steepest descent algorithm, while staying on the hyperplane, i.e., while obeying the constraint. But then $\mathbf{x}^{\min}$ would not be the constrained minimum.

As the gradient $\nabla g(\mathbf{x})$ is always perpendicular to any plane defined by $g(\mathbf{x}) = \text{constant}$, it follows from Eq. 3.248 and the property **(I)**

$$\mathbf{x} = \mathbf{x}^{\min} \Rightarrow \nabla f(\mathbf{x}) = \beta \nabla g(\mathbf{x}). \tag{3.251}$$

The reverse is not necessarily true, i.e., not all points $\mathbf{x}$ where the two gradients are proportional, correspond to a minimum. However, such points always correspond to a constrained stationary point, i.e., a maximum, a minimum, or a saddle point. The proof of that is the same as the one above. It means that we can use Eq. 3.251 to look for a stationary point. Whether this point is a minimum, instead of a maximum, or a saddle point, can be tested after we have found it.

Lagrange came up with a very elegant solution for minimizing a function, Eq. 3.247, under a constraint such as Eq. 3.248, making use of Eq. 3.251. Instead of *eliminating* a variable, as in Eq. 3.250, he *adds* one, and defines

$$\tilde{f}(x_1, x_2, ..., x_N, \lambda) \equiv f(x_1, x_2, ..., x_N) + \lambda g(x_1, x_2, ..., x_N). \tag{3.252}$$

The variable added, λ, is called a *Lagrange multiplier*. We now linearize the expression by looking at points $x_i + \Delta x_i;\ i = 1, ..., N$, and making a Taylor expansion

$$\begin{aligned} &\tilde{f}(x_1 + \Delta x_1, x_2 + \Delta x_2, ..., x_N + \Delta x_N, \lambda + \Delta\lambda) \approx \tilde{f}(x_1, x_2, ..., x_N, \lambda), \\ &+ \sum_{i=1}^{N} \frac{\partial \tilde{f}(x_1, x_2, ..., x_N, \lambda)}{\partial x_i} \Delta x_i + \frac{\partial \tilde{f}(x_1, x_2, ..., x_N, \lambda)}{\partial \lambda} \Delta\lambda. \end{aligned} \tag{3.253}$$

Then a stationary point is defined by

$$\frac{\partial \tilde{f}(x_1, x_2, ..., x_N, \lambda)}{\partial x_i} = 0;\ i = 1, ..., N \quad \text{and} \quad \frac{\partial \tilde{f}(x_1, x_2, ..., x_N, \lambda)}{\partial \lambda} = 0. \tag{3.254}$$

Using Eq. 3.252 this translates into

$$\frac{\partial f(x_1, x_2, ..., x_N)}{\partial x_i} = \lambda \frac{\partial g(x_1, x_2, ..., x_N)}{\partial x_i};\ i = 1, ..., N \quad \text{and} \tag{3.255}$$

$$g(x_1, x_2, ..., x_N) = 0. \tag{3.256}$$

But Eq. 3.255 is nothing but Eq. 3.251, and Eq. 3.256 is just the constraint, Eq. 3.248. In other words, by solving the *unconstrained* minimization problem with $\tilde{f}$ in the $N+1$ variables $x_i;\ i = 1, ..., N$ and λ, Eq. 3.254, we find a solution to the *constrained* minimization problem in $N - 1$ variables, Eqs. 3.247 and 3.248. Brilliant!![37]

[37] It is depressing to see how smart these old guys were. Lagrange reformulated the whole of classical mechanics, and he did this bit of mathematics just on the side (or maybe the other way around).

Constrained optimization is easily generalized to multiple constraints of the form

$$g_j(x_1, x_2, ..., x_N) = 0;\ j = 1, ..., M < N. \tag{3.257}$$

Define a function

$$\bar{f}(x_1, x_2, ..., x_N, \lambda_1, \lambda_2, ..., \lambda_M) \equiv f(x_1, x_2, ..., x_N) + \sum_{j=1}^{M} \lambda_j g_j(x_1, x_2, ..., x_N), \tag{3.258}$$

with $\lambda_j;\ j = 1, ..., M$ Lagrange multipliers. Then the *unconstrained* optimization of $\bar{f}$ in the $N + M$ variables $x_i;\ i = 1, ..., N;\ \lambda_j;\ j = 1, ..., M$ gives the solution to the *constrained* optimization problem in $N - M$ variables, Eqs. 3.247 and 3.257.

The method of Lagrange multipliers comes at a prize. The more constraints M you have, and the smaller the dimension $N - M$ of the search space should be, the larger the dimension $N + M$ of the search space actually becomes, if using Lagrange multipliers.[38] For large M one should seriously consider, whether it is possible to eliminate variables, as in Eqs. 3.249 and 3.250. Having said that, Lagrange multipliers remain a brilliant method.

[38] Part of his life Lagrange was a politician (!) Maybe that's where this solution comes from.

Part II

THEORETICAL BACKGROUND

Chapter 4

Electrostatic Screening

In this chapter I consider some physical examples that involve solving (non-)linear equations. Sec. 4.1 discusses the Poisson-Boltzmann equation, and its linearized cousin, the DeBye-Hückel equation, which describe the potential profile inside a liquid that contains mobile charges (ions), if subjected to an electric field.[1] This is of relevance for plasma physics, for electrochemistry, and for biological processes where such liquids play a role. The charge density generated by those ions is treated as a smooth, continuous function $\rho(\mathbf{r})$. On an atomic scale the ions are discrete entities, and the medium surrounding them consists of discrete molecules. As an example of electrostatics on the atomic scale, Sec. 4.2 discusses the polarization of a medium consisting of discrete molecules, by a charge or a multipole embedded in that medium.

4.1 Poisson-Boltzmann equation

Consider a parallel plate capacitor with a homogeneous dielectric between the (infinite) plates, see Fig. 4.1. The dielectric has a (static) permittivity ϵ. If we keep the two metal plates at fixed potentials Φ_1 and Φ_2, respectively, electrostatics tells us that the potential in the dielectric is given by a simple linear function

$$\Phi(x) = \Phi_1 + \frac{x}{d}\left(\Phi_2 - \Phi_1\right); \quad 0 < x < d, \tag{4.1}$$

with d the distance between the plates. The electric field

$$E_x = -\frac{d\Phi}{dx} = \frac{\Phi_1 - \Phi_2}{d}, \tag{4.2}$$

is constant. The two metal plates acquire an interface charge density σ_0, and $-\sigma_0$, respectively, which is situated at the interfaces with the dielectric. From $Q = CV$, with $Q = \sigma_0 A$ the total charge on the plate with area A, and $C = \epsilon A/d$ the capacity, one has

$$\sigma_0 = \left(\Phi_1 - \Phi_2\right)\frac{\epsilon}{d}. \tag{4.3}$$

Now assume the more interesting case where the material between the plates contains mobile charges. One can think of a liquid that contains solvated ions, for instance, which

[1] The corresponding wikipedia page gives a nice introduction.

is called an *electrolyte* [31]. In water this is almost always the case, as many salts naturally dissolve into ions. Earth biology could not proceed without such watery electrolyte solutions. Even if one does one's uttermost best to remove such salts, even then water contains a natural concentration of protons, H^+, and hydroxyl ions, OH^-, which result from the splitting of water molecules $H_2O \rightleftharpoons H^+ + OH^-$. Many applications using renewable energy where electricity plays a role, such as batteries, fuel cells, and electrolysis, would not be possible without electrolytes. Sometimes one has to avoid water, as that can cause unwanted chemical reactions if the potential difference becomes too large. In that case, the electrolyte consists of more complex (mostly organic) molecules. Still, the principle is the same, the electrolyte contains a concentration of mobile charges in the form of solvated ions. In the following we assume that the electrolyte is stable, and no chemical reactions occur.

We assume that the electrolyte as a whole is neutral, meaning that the total charge of the solvated cations (positive) has to be equal to the total charge of the solvated anions (negative). Nevertheless, locally the concentration of the cations can be higher or lower than that of the anions. In fact, putting the liquid between the plates, while keeping the potentials Φ_1 and Φ_2 fixed, changes the local ion concentration. Anions are attracted to the positive plate, and cations are attracted to the negative plate. Separation of positive and negative ions of course changes the potential profile. The potential has to obey the Poisson equation[2]

$$\frac{d^2\Phi}{dx^2}(x) = -\frac{\rho(x)}{\epsilon}, \quad \text{with} \quad \rho(x) = \sum_i^{\text{species}} \rho_i(x) \tag{4.4}$$

where $\rho_i(x)$ is the charge density resulting from ions of type i, and the sum is over all species of ions (positive and negative); ϵ is the permittivity of the "background" liquid, i.e., of the liquid without the mobile ions.

The ions move toward the metal plates until the charges in the plates are fully compensated. This reasoning follows from considering the liquid+ions as a conductor. Charges move towards the surfaces of the liquid, until the electric field inside the liquid is zero. At zero temperature, ions would simply adsorb onto the metal-dielectric interface until the charge on the metal side of the interface is compensated. In case you should worry, there is plenty of room at an interface. It takes relatively few adsorbed ions to change a potential considerably, and make the interface neutral.

At finite temperature, the ions are subjected to thermal motion, and have to obey Boltzmann statistics. In equilibrium, the local concentration of ion species i is given by the Boltzmann expression

$$c_i(x) = c_{0,i} \exp\left[-\frac{V_i(x)}{k_B T}\right], \tag{4.5}$$

where $c_{0,i}$ is the concentration of the ions if the potential difference between the plates is zero, and $V_i(x)$ is the energy of the ions if the potential difference is non-zero. Electrostatics tells us that $V_i(x) = q_i\,(\Phi(x) - \Phi_0)$, where $\Phi_0 = \frac{1}{2}\,(\Phi_2 + \Phi_1)$ is the average potential, and q_i is the charge of ionic species i. The charge density resulting from ionic species i is given by $\rho_i(x) = q_i c_i(x)$. In other words, Eq. 4.4 becomes

$$\frac{d^2\Phi}{dx^2}(x) = -\frac{\rho(x)}{\epsilon}, \quad \text{with} \quad \rho(x) = \sum_i^{\text{species}} q_i c_{0,i} \exp\left[-\frac{q_i\,(\Phi(x) - \Phi_0)}{k_B T}\right]. \tag{4.6}$$

[2]The 1D version of the Poisson equation. We assume that in directions parallel to the plates (y and z), the potential is constant.

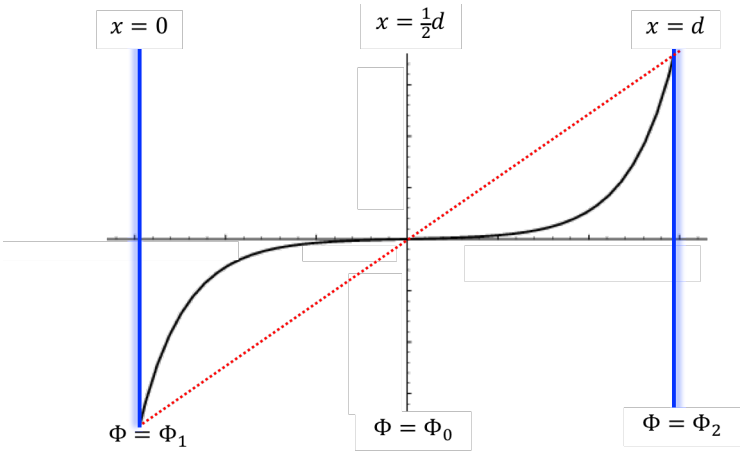


Figure 4.1: Two parallel metal plates at $x = 0$ and $x = d$; the plates are assumed to be infinite in the directions parallel to the plates. The plates are at constant potentials Φ_1 and Φ_2, respectively. The red dotted line represents the potential $\Phi(x)$ in a dielectric between the plates, Eq. 4.1. The solid black line represents the potential in a liquid containing mobile ions (such Na^+ and Cl^- in water), according to the DeBye-Hückel approximation, see Eq. 4.10. Away from the plates the potential decays to its average value as $\propto \exp[-\kappa r]$, defining a charged layer of effective thickness $\lambda_D = \kappa^{-1}$, see Eq. 4.11.

This nonlinear differential equation is called the *Poisson-Boltzmann equation*. Its solution has to obey the boundary conditions

$$\Phi(0) = \Phi_1;\ \Phi(d) = \Phi_2 \quad (\text{and, of course, } \Phi_0 = \frac{1}{2}(\Phi_2 + \Phi_1)) \tag{4.7}$$

In the general case one has to solve the Poisson-Boltzmann equation numerically. Under special circumstances, analytical solutions are possible. Suppose, for instance, that only two species of ions are present, with $q_1 = q$, and $q_2 = -q$. Then $c_{0,1} = c_{0,2} = c_0$, and

$$\rho(x) = -2qc_0 \sinh\left[\frac{q(\Phi(x) - \Phi_0)}{k_B T}\right]. \tag{4.8}$$

For this special $\rho(x)$, Eq. 4.6 has been solved analytically by Gouy and Chapman more than a century ago. Hence this theory of electrolytes is also called *Gouy-Chapman* theory *[31]*. It is possible to solve Eq. 4.6 analytically for some other special charge distributions $\rho(x)$, but the solutions tend to be rather complicated special functions. [3]We will solve this equation numerically.

4.1.1 DeBye-Hückel approximation

Most of the textbook discussions concerning this problem deal with the linear approximation to Eq. 4.6. We restrict ourselves to the case described by Eq. 4.8, and assume that $k_B T \gg q(\Phi(x) - \Phi_0)$, which for obvious reasons is called the high temperature limit.

[3]A differential equation of type $d^2\Phi/dx^2 \propto \sinh\Phi$ looks a bit like the equation of motion of a pendulum with large amplitude, where one has a normal "sin" instead of the hyperbolic "sinh". As $\sinh y = i\sin(iy)$, one can make use of the solutions of the pendulum problem. These solutions are special functions, called *Jacobi elliptical functions*. If you are a mathematician, you should look them up. If you are not (like me), you just use this story to show off.

As $\sinh(y) \approx y$ for small y (our old friend Taylor again), Eq. 4.6 then transforms to

$$\text{if } k_BT \gg q\left(\Phi(x)-\Phi_0\right); \quad \text{then } \frac{d^2\Phi}{dx^2}(x) = \kappa^2\left(\Phi(x)-\Phi_0\right); \quad \text{with } \kappa^2 = \frac{2q^2c_0}{\epsilon k_BT}, \tag{4.9}$$

This is called the *DeBye-Hückel equation* .[4] This linear differential equation is easily solved and made to obey the boundary conditions, Eq. 4.7, to give[5]

$$\Phi(x) = \frac{\Phi_2-\Phi_1}{2}\frac{\sinh\left[\kappa\left(x-\frac{1}{2}d\right)\right]}{\sinh\left(\frac{1}{2}\kappa d\right)} + \frac{\Phi_2+\Phi_1}{2}. \tag{4.10}$$

For a plot, see Fig. 4.1. If $\kappa d \ll 1$, then we can approximate $\sinh y \approx y$, and Eq. 4.10 becomes Eq. 4.1 again. In that case, according to the Poisson equation, $\rho(x) = -\epsilon d^2\Phi/dx^2 = 0$, i.e., the positive and negative ions are distributed evenly over the whole system. The parameter κ is small if the temperature T is very large and/or if the ion concentration c_0 is very small, see Eq. 4.9. The thermal motion of the ions then dominates everything. As you know from statistical physics, that motion wants to distribute particles evenly over all available space, as that gives a maximum entropy.

In contrast, if $\kappa d \gg 1$, then $\Phi(x) - \Phi_0 \propto \exp\left[-\kappa x\right]$ for x near the first plate, i.e., the potential decays exponentially away from that plate. The parameter

$$\lambda_D = \frac{1}{\kappa} = \sqrt{\frac{\epsilon k_BT}{2q^2c_0}}, \tag{4.11}$$

is called the *DeBye screening length*. It typifies the length scale over which the potential varies from Φ_1 on the plate, to Φ_0 of the neutral liquid. As $\rho(x) = -\epsilon d^2\Phi/dx^2 \propto \exp\left[-\kappa x\right]$ in this limit, λ_D is also the length scale over which the charge density drops from a finite value at the plate to zero in the neutral liquid. For a 0.1 mol/L solution of NaCl in water at 25°C, $\lambda_D \approx 1$ nm. Obviously it increases with increasing temperature, and with decreasing salt concentration.

4.1.2 The double layer

One can obtain the charge density from the Poisson equation $\rho(x) = -\epsilon d^2\Phi/dx^2$ and Eq. 4.10

$$\rho(x) = -\frac{\left(\Phi_2-\Phi_1\right)\epsilon\kappa^2}{2}\frac{\sinh\left[\kappa\left(x-\frac{1}{2}d\right)\right]}{\sinh\left(\frac{1}{2}\kappa d\right)}, \tag{4.12}$$

and calculate the total charge per m^2 in the half space

$$\sigma' = \int_0^{d/2}\rho(x)dx = \frac{\left(\Phi_2-\Phi_1\right)\epsilon\kappa}{2}\tanh\left(\frac{1}{4}\kappa d\right). \tag{4.13}$$

As $\lim_{y\to 0}\tanh(y) = 0$, σ' vanishes for $\kappa d \to 0$, as we have already concluded in the previous subsection. In contrast, $\lim_{y\to\infty}\tanh y = 1$, so for $\kappa d \gg 1$ we have

$$\sigma' \approx \frac{\left(\Phi_2-\Phi_1\right)\epsilon\kappa}{2} = \frac{\left(\Phi_2-\Phi_1\right)\epsilon}{2\lambda_D}. \tag{4.14}$$

[4] I am told that the proper pronunciation should be "debéé" (in Dutch "de bie") and not "debáie", as DeBye was a Limburger.

[5] If you don't see this right away, you can always start with the ansatz $\Phi(x)-\Phi_0 = A\exp\left[\kappa x\right]+B\exp\left[-\kappa x\right]$ and find A, B such that the boundary conditions are obeyed.

This charge in the liquid has to be compensated exactly by counter charges σ on the metal side of the metal/liquid interface, as the electric field in the middle of the liquid has to be zero. This follows from the usual conductor argument. Non-compensated charges would lead to an electric field inside the liquid, but this would set the ions in motion, until the electric field is brought to zero.

So $\sigma = -\sigma'$. A comparison to the simple dielectric case, Eq. 4.3, then shows that

$$\frac{\sigma}{\sigma_0} \approx \frac{d}{2\lambda_D} \gg 1, \tag{4.15}$$

as λ_D is of order nm for a typical liquid with ions, see the example above, and d is usually of macroscopic size (at least μm) By using a liquid one can thus tremendously increase the charge density on the metal plate. This becomes especially interesting if we replace the metal by a semiconductor, because we now have a means of attaining a large charge density in the semiconductor at its interface with the liquid. Creating such charge densities by applying an electric field is typically done by the gate electrode in a semiconductor *field-effect transistor (FET)*. Using a *liquid gate* electrode is thus an efficient means of generating a large field effect.

The charged liquid layer close to the interface plus the counter charges on the metal side are called *the double layer*. It plays an important role in describing solid/liquid interfaces in electrochemistry and batteries, for instance. A plasma is basically an gas of ions. As these also obey Boltzmann statistics, it means we can use the formalism introduced above also to describe processes in plasma physics.

At room temperature $k_BT \approx 26$ meV, so the Debye-Hückel equation, Eq. 4.9, is actually valid only for very small voltage differences. If the voltages become significantly larger, one has to use the full nonlinear Poisson-Boltzmann equation, Eq. 4.6. Features such as the screening length and the double layer still have a physical meaning, but their numerical value will be very different from Eqs. 4.11, 4.14 and 4.15. We will solve the Poisson-Boltzmann equation numerically, and have a look at these differences.

4.2 The self-consistent polarization field

A continuum charge distribution as used in the previous section suffices to model electrostatics at a macroscopic level. At a microscopic level, that charge distribution is divided over discrete molecules and ions. Taking into account the microscopic molecular structure of the material defines a so-called *micro-electrostatic model*. In such a model the material is represented by discrete (point) polarizabilities each representing a molecule or molecular fragment. The "applied"' field is the electric field from a microscopic distribution of ions, usually represented by point charges. This field polarizes the molecules. The *polarization energy* is the energy gained by the response of a material to an applied electrostatic field. Here it is described by the electrostatic interaction between induced dipoles and the applied field. This polarization energy plays a key role in describing the solvation of ions in liquids, or, more general, in describing the energies of charged species in molecular or biological matter.

Example: molecular crystals* One nice example where polarization plays an important role can be found in molecular crystals, which are 3D ordered packings of organic molecules.

Typically, such materials are insulators, but for crystals of molecules such as pentacene or C_{60} the band gap is sufficiently small (2-3 eV), such that these materials are called *organic semiconductors*. Transport of charge carriers (electrons and holes) in these materials is quite different from transport in conventional semiconductors such as silicon. Organic semiconductors consist of loosely packed molecules with relatively weak interactions between the molecules, so there is hardly a formation of bands as in conventional metals or semiconductors. A charge carrier is localized on one molecule, and proceeds through the crystal by jumping to one of the neighboring molecules.

If you believe this story, then the properties of an organic semiconductor should be easily deduced from those of the individual molecules. For instance, the semiconductor band gap should correspond to the difference between the ionization potential (IP) and the electron affinity (EA) of a molecule, as these are, per definition, the energies of attaching a hole, respectively an electron, to the molecule. However, if one calculates IP − EA, with IP and EA measured for isolated molecules, the result is different from the measured semiconductor band gap by several eV!

This paradox can be resolved by accounting for the polarization energy. An electron or hole localized on a molecule that is surrounded by other, neutral, molecules, polarizes those surrounding molecules, thereby lowering its energy. Both the ionization potential and the electron affinity are then reduced to IP $- E_p$ and EA $- E_p$, with E_p the polarization energy (see below), which, depending on the material, is easily on the scale of eV. The band gap of the molecular crystal is then predicted to be $E_g = \text{IP} - \text{EA} - 2E_p$, and, lo and behold, this expression gives values quite close to the measured band gaps.

Polarization Let a material consist of N_D molecules, with each molecule i characterized by its position $\mathbf{r}_i$ and its polarizability tensor $\alpha_{i,ab}; a, b = x, y, z$. The induced dipole μ_{ia} on molecule i in the direction a is given by the expression

$$\mu_{ia} = \sum_{b=x,y,z} \alpha_{i,ab} E_{ib}; \quad i = 1, ..., N_D; a = x, y, z, \tag{4.16}$$

where $E_{ib} \equiv E_b(\mathbf{r}_i)$ is the total electric field operating on molecule i in direction $b = x, y, z$. That field can be divided into an field $E_{ib}^{(0)}$, resulting from a fixed microscopic charges, and a field E_{ib}^{pol} resulting from all induced dipoles

$$E_{ib} = E_{ib}^{(0)} + E_{ib}^{\text{pol}}. \tag{4.17}$$

We represent the field $E_{ib}^{(0)}$ as

$$E_{ib}^{(0)} = -\frac{1}{4\pi\epsilon_0} \sum_{k=1}^{N_q} \frac{q_k r_{ik,b}}{r_{ik}^3}, \tag{4.18}$$

in terms of N_q point charges q_k at positions $\mathbf{r}_k$, where $r_{ik} = |\mathbf{r}_k - \mathbf{r}_i|$ and $r_{ik,b}$ is the b component of $\mathbf{r}_k - \mathbf{r}_i$. Keeping in line with the nomenclature in the literature, this field is called the *external field* (although the point charge are usually dispersed within the material). The field resulting from the induced dipoles on all molecules can be written as

$$E_{ib}^{\text{pol}} = \sum_{j=1, j\neq i}^{N_D} \sum_{c=x,y,z} T_{ib,jc} \mu_{jc} \quad \text{with} \quad T_{ib,jc} = \frac{1}{4\pi\epsilon_0} \frac{3 r_{ij,b} r_{ij,c} - \delta_{bc} r_{ij}^2}{r_{ij}^5}, \tag{4.19}$$

the elements familiar from the dipole-dipole interaction. Note that $j \neq i$ in the sum over the molecules, as a molecule does not interact with itself, of course.

Eqs. 4.16-4.19 are often solved using an iterative scheme, which is called a *self-consistent polarization field* (SCPF) calculation. One starts, for instance, by solving Eq. 4.16 using $E_{ib}^{(0)}$ instead of E_{ib}. Then one computes the polarization field from Eq. 4.19, and plugs this back in Eq. 4.16 to find new dipoles, constructs the polarization field again via Eq. 4.19, etcetera. This process is repeated until the dipoles do not change anymore, in other words, when the dipoles and the polarization field are consistent. Although straight-forward, It is not advised to use this particular algorithm for solving Eqs. 4.16-4.19. Better algorithms are discussed in Sec. 4.2.2 [32, 33].

4.2.1 Polarization energy

We define the polarization energy as the energy change while building up the induced dipoles, going from zero field to $\mathbf{E}^{(0)}(\mathbf{r}_i)$

$$\mathcal{E}_P = -\sum_{i=1}^{N_D} \int_0^{\mathbf{E}_i^{(0)}} \boldsymbol{\mu}_i \left(\mathbf{E}'\right) \cdot d\mathbf{E}'. \tag{4.20}$$

We don't have to do the integral explicitly. Once the induced dipoles are computed for a given field $\mathbf{E}^{(0)}(\mathbf{r}_i)$, the polarization energy can also be calculated from the expression

$$\mathcal{E}_P = -\frac{1}{2}\sum_{i=1}^{N_D} \sum_{a=x,y,z} E_{ia}^{(0)} \mu_{ia}. \tag{4.21}$$

Proof: From Eqs. 4.16-4.19 it follows that all induced dipoles are a homogeneous function of the external field, $\boldsymbol{\mu}_i\left(\kappa \mathbf{E}_i^{(0)}\right) = \kappa \boldsymbol{\mu}_i\left(\mathbf{E}_i^{(0)}\right)$, with κ any constant. We can use Euler's theorem for homogeneous functions: $\mu_{ia}\left(E'_x, E'_y, E'_z\right) = \sum_{b=x,y,z} E'_b \partial \mu_{ia} / \partial E'_b$, and use this expression in Eq. 4.20. Partial integration then leads to Eq. 4.21.

I can offer a second justification of Eq. 4.21, but it is slightly more complicated, so you may skip it if you want. Write the polarization energy as

$$\mathcal{E}_P = -\sum_{i=1}^{N_D} \sum_{a=x,y,z} E_{ia}^{(0)} \mu_{ia} + \frac{1}{2}\sum_{i=1}^{N_D} \sum_{b=x,y,z} \sum_{j=1, j\neq i}^{N_D} \sum_{c=x,y,z} \mu_{ib} T_{ib,jc} \mu_{jc} - \frac{1}{2}\sum_{i=1}^{N_D} \sum_{a=x,y,z} \sum_{b=x,y,z} \mu_{ia} \alpha_{i,ab}^{-1} \mu_{ib}, \tag{4.22}$$

The first term represents the interaction of all dipoles with the external field, and the second term represents the interaction between all pairs of dipoles. The third term comes from the energy change by building up a single dipole

$$-\int_0^{\boldsymbol{\mu}_i} \mathbf{E}_i\left(\boldsymbol{\mu}'_i\right) \cdot d\boldsymbol{\mu}'_i = -\frac{1}{2}\sum_{a=x,y,z} \sum_{b=x,y,z} \alpha_{i,ab}^{-1} \mu_{ia} \mu_{ib}, \tag{4.23}$$

where $\boldsymbol{\alpha}_i^{-1}$ is the inverse of the 3×3 matrix $\boldsymbol{\alpha}_i$. I have used Eq. 4.16 in this derivation, written formally as $\mathbf{E}_i = \boldsymbol{\alpha}_i^{-1} \boldsymbol{\mu}_i$. Using the latter expression in Eq. 4.22, and using Eqs. 4.17 and 4.19 then proves the equivalence of Eqs. 4.21 and 4.22. In the following I will use Eq. 4.21 as it is the simpler one.

Convergence* The number of dipoles N_D to include in the calculation of Eqs. 4.19 and 4.21 should be large enough to converge the polarization energy $\mathcal{E}_P$. However, convergence as a function of N_D is typically slow, as can be argued on the basis of simple dimensional arguments. Suppose the field of Eq. 4.18 is that of a single point charge. The field at a distance r from that charge is proportional to $1/r^2$, which, according to Eq. 4.16, means that the induced dipole on a molecule at a distance r has a contribution that is proportional to $1/r^2$. The polarization energy, Eq. 4.21, then involves a sum over terms proportional to $1/r^4$. Suppose we we sum over all dipoles within a sphere of radius $R = N_R d$, with d a convenient scale, such as the nearest neighbor intermolecular distance. Assuming that d is also the minimum distance between the charged ions and the induced dipoles, neglecting all anisotropy, and convert the sum into an integral, one has

$$\mathcal{E}_P \propto -\int_d^R r^{-4} r^2 dr = 1/d - 1/R \tag{4.24}$$

The convergence error is then

$$\varepsilon \equiv \frac{\mathcal{E}_P(R=\infty) - \mathcal{E}_P(R)}{\mathcal{E}_P(R=\infty)} \propto \frac{d}{R} \sim \frac{1}{N_R} \sim \frac{1}{\sqrt[3]{N_D}}, \tag{4.25}$$

where $N_D \propto N_R^3$, is the number of dipoles within the sphere.

In practice, one calculates $\mathcal{E}_P$ by performing a series of calculations for increasing N_R, and extrapolating the series to $N_R \to \infty$, making use of Eq. 4.24. Even so, the maximum N_D required to obtain an extrapolated $\mathcal{E}_p$ that is accurate within 10^{-3}-10^{-2} eV can be substantial. As we will see in the next section, the computing time for solving the SCPF problem scales as N_D^n; $n = 2$ or 3. In view of the slow convergence of the error $\varepsilon(N_D)$, Eq. 4.25, one must therefore choose the algorithm with care.

4.2.2 Algorithms

We can transform the SCPF problem of Eqs. 4.16–4.19 into a standard linear problem. Defining vectors $\mathbf{m}$ with $3N_D$ components μ_{ia}, and $\mathbf{f}$ with components E_{ia}, and defining $3N_D \times 3N_D$ matrices $\mathbf{T}$ and $\mathbf{A}$ with coefficients

$$(\mathbf{T})_{ia,jb} = T_{ia,jb}\,(1 - \delta_{ij}) \quad \text{and} \quad (\mathbf{A})_{ia,jb} = \delta_{ij}\alpha_{i,ab}, \tag{4.26}$$

Eq. 4.16 becomes

$$\mathbf{m} = \mathbf{A}\,(\mathbf{f} + \mathbf{T}\mathbf{m})\,. \tag{4.27}$$

This is equivalent to the set of linear equations

$$\begin{aligned} [\mathbf{I} - \mathbf{A}\mathbf{T}]\,\mathbf{m} &= \mathbf{A}\mathbf{f} \quad \text{or} \\ \left[\mathbf{A}^{-1} - \mathbf{T}\right]\mathbf{m} &= \mathbf{f}, \end{aligned} \tag{4.28}$$

from which the induced dipoles $\mathbf{m}$, Eq. 4.16, can be obtained, given the fields $\mathbf{f}$, Eq. 4.18. The polarization energy of Eq. 4.21 is then given by

$$\mathcal{E}_P = -\frac{1}{2}\mathbf{f}^\dagger\mathbf{m}. \tag{4.29}$$

Solving the linear problem of Eq. 4.28 is the main task. Calculating $\mathbf{A}^{-1}$ should not be a problem. It only involves calculating the inverses of the 3×3 matrices $\boldsymbol{\alpha}_i$; $i = 1, ..., N_D$,

see Eq. 4.26. The matrix $\mathbf{A}^{-1} - \mathbf{T}$ however does not have this simple block-diagonal form. As its dimension $3N_D$ can be very large, we will solve the linear problem using an iterative technique. An iterative scheme involves repeated calculations of the matrix-vector products $\mathbf{Tv}$ and $\mathbf{A}^{-1}\mathbf{v}$. The matrix elements of the $\mathbf{T}$ and $\mathbf{A}^{-1}$ matrices are relatively simple, see Eqs. 4.19 and 4.26, so we recalculate them each time they are needed. This means that no matrices need to be stored, but only vectors. The memory requirements then scale as $O(N_D)$. Computing matrix-vector products is the rate limiting step, which means that the computational costs scale as $O(nN_D^2)$, with n the number of iterations required to converge. A further advantage of using an iterative technique is that calculating matrix-vector products is suitable for parallelization.

Compare this to a direct algorithm, such as LU decomposition, where the memory requirements scale as $O(N_D^2)$ (here, there is no way around storing the matrix), and the computational costs scale as $O(N_D^3)$.

Ad hoc iterative scheme A simple *ad hoc* SCPF iterative scheme found in the literature [32, 33] can be derived from Eq. 4.27 using the fixed-point iteration rule

$$\begin{aligned} \mathbf{m}^{(n)} &= \mathbf{Af} + \mathbf{ATm}^{(n-1)} \quad \text{or, in components} \\ \mu_{ia}^{(n)} &= \sum_{b=x,y,z} \alpha_{i,ab} \left[E_{ib}^{(0)} + \sum_{c=x,y,z} \sum_{j=1, j\neq i}^{N_D} T_{ib,jc} \mu_{jc}^{(n-1)} \right], \end{aligned} \tag{4.30}$$

where one obtains the dipole vector at the (n)'th iteration from that of the $(n-1)$'th iteration, starting with $\mathbf{m}^{(0)} = \mathbf{0}$.[6]

Defining the error vector as the difference between the (n)'th iterate and the exact solution $\mathbf{m}$, i.e. $\mathbf{e}^{(n)} = \mathbf{m} - \mathbf{m}^{(n)}$, one obtains from Eq. 4.30

$$\mathbf{e}^{(n)} = \mathbf{ATe}^{(n-1)} \tag{4.32}$$

The asymptotic error for large n then becomes

$$\left\|\mathbf{e}^{(n)}\right\| \approx \rho(\mathbf{AT}) \left\|\mathbf{e}^{(n-1)}\right\| \tag{4.33}$$

where $\rho(\mathbf{AT}) = |\lambda_{\max}|$ is the spectral range (the eigenvalue of maximum size) of the matrix $\mathbf{AT}$.

If $\rho(\mathbf{AT}) < 1$ then this algorithm converges linearly, see Eq. 1.56. If $\rho(\mathbf{AT}) > 1$, we may be in trouble and not reach convergence at all. It is not that hard to reach $\rho(\mathbf{AT}) > 1$, so it is not that hard to get into trouble.

Details: consider a simple example, where the molecular polarizability is isotropic, $\alpha_{i,ab} = \delta_{ab}\alpha$, and the molecules are placed on a simple cubic lattice with lattice parameter d. We include all molecules in the sum of Eq. 4.19 that lie within a sphere of radius $R = N_R d$. A simple rough

[6] The iterative rule can also be written as

$$\mathbf{m}^{(n)} = \mathbf{m}^{(n-1)} + \mathbf{AT}(\mathbf{m}^{(n-1)} - \mathbf{m}^{(n-2)}) \tag{4.31}$$

as can be verified by direct substitution, starting with $\mathbf{m}^{(0)} = \mathbf{0}; \mathbf{m}^{(1)} = \mathbf{Af}$. Both rules, Eq. 4.30 or 4.31, can be found in the literature.

estimate for the spectral range is then given by

$$\rho(\mathbf{AT}) \leq \|\mathbf{AT}\|_\infty = \alpha \sum_{j=2}^{N_D} |T_{1x,jx}| \approx$$
$$\frac{\alpha}{d^3}\frac{1}{d^3}\int_d^R \int_{-1}^1 \int_0^{2\pi} \frac{|3\cos^2\theta - 1|}{r^3} r^2 dr d(\cos\theta) d\phi = \frac{\alpha}{d^3}\frac{2^6\pi}{3^3}\ln\frac{R}{d} = C\ln\frac{R}{d}, \tag{4.34}$$

where the subscript 1 refers to a position at the center of the sphere and a continuum approximation is used in the second line. As $\ln(R/d)$ increases monotonically with R, only for small spheres we are guaranteed $\rho(\mathbf{AT}) < 1$. Eq. 4.34 is a "worse case" estimate, but convergence problems for the algorithm above a certain size of R should be expected.

Proof: The ∞-norm of any $N \times N$ matrix $\|\mathbf{B}\|_\infty$ is defined as $\max_{\mathbf{x}} \|\mathbf{Bx}\|_\infty / \|\mathbf{x}\|_\infty$, similar to the matrix norm in Eq. 1.12, where the ∞-norm of a vector is defined as in Eq. 1.54. From the proof to Eq. 1.15 we know $\rho(\mathbf{B}) = \|\mathbf{B}\|$ (provided the matrix is Hermitian, which it is). Furthermore $\|\mathbf{B}\| \leq \|\mathbf{B}\|_\infty$ (for a Hermitian matrix) and $\|\mathbf{B}\|_\infty = \max_{1\leq i\leq N} \sum_{j=1}^N |B_{ij}|$ (see, for instance, in Ref. [4], p. 173). For a matrix $\mathbf{T}$ defined by Eqs. 4.19 and 4.26, the max is for $i = 1$. Also the continuum limit in the second line of Eq. 4.34 follows from these equations.

Damping If the algorithm does not converge, or it converges very badly, we may try a simple damping technique in order to improve it. Instead of the iteration rule of Eq. 4.30 we can try

$$\mathbf{m}^{(n)} = (1-\gamma)\,\mathbf{m}^{(n-1)} + \gamma\left(\mathbf{Af} + \mathbf{ATm}^{(n-1)}\right) \quad \text{with} \quad 0 < \gamma < 1. \tag{4.35}$$

Defining the error vector as above, $\mathbf{e}^{(n)} = \mathbf{m} - \mathbf{m}^{(n)}$, one obtains from Eq. 4.35

$$\mathbf{e}^{(n)} = (1-\gamma)\,\mathbf{e}^{(n-1)} + \gamma\mathbf{ATe}^{(n-1)}. \tag{4.36}$$

The asymptotic error for large n then becomes

$$\left\|\mathbf{e}^{(n)}\right\| \approx \rho(\mathbf{B}) \left\|\mathbf{e}^{(n-1)}\right\| \quad \text{with} \quad \mathbf{B} = (1-\gamma)\,\mathbf{I} + \gamma\mathbf{AT}. \tag{4.37}$$

For γ sufficiently small we should get $\rho(\mathbf{B}) \ll \rho(\mathbf{AT})$, and reach the situation where $\rho(\mathbf{B}) < 1$, where the algorithm converges.

Jacobi iteration The SCPF iterative scheme of Eq. 4.30 can be interpreted as Jacobi iteration. Write Eq. 4.28 in the standard form

$$\mathbf{Bm} = \mathbf{f} \quad \text{with} \quad \mathbf{B} = \mathbf{A}^{-1} - \mathbf{T}. \tag{4.38}$$

$\mathbf{B}$ is a symmetric matrix, because $\mathbf{A}$ and $\mathbf{T}$ are symmetric matrices. If one splits $\mathbf{B}$ as

$$\mathbf{B} = \mathbf{L} + \mathbf{D} + \mathbf{L}^\dagger, \tag{4.39}$$

with matrix elements

$$D_{ia,jb} = \delta_{ij} B_{ia,ib}; \quad L_{ia,jb} = B_{ia,jb};\ i > j; \quad L_{ia,jb} = 0;\ i < j,$$

then the Jacobi iteration algorithm on the linear problem of Eq. 4.38, is defined by the propagation rule

$$\mathbf{m}^{(n)} = \mathbf{m}^{(n-1)} - \mathbf{D}^{-1}\left[\mathbf{B}\mathbf{m}^{(n-1)} - \mathbf{f}\right] = \mathbf{D}^{-1}\mathbf{f} - \mathbf{D}^{-1}(\mathbf{L}+\mathbf{L}^{\dagger})\mathbf{m}^{(n-1)} \tag{4.40}$$

The first equality follows from Eq. 1.69 and the second equality from Eq. 4.39 Comparing Eqs. 4.38 and 4.39 we see that $\mathbf{D} = \mathbf{A}^{-1}$ and $\mathbf{T} = -(\mathbf{L}+\mathbf{L}^{\dagger})$. This means that the Jacobi iteration rule of Eq. 4.40 is identical to the SCPF algorithm of Eq. 4.30. No surprise then that the asymptotic error rate, Eq. 4.33, is that of Jacobi iteration, Eq. 1.73.

Gauss-Seidel iteration A simple modification of the SCPF algorithm improves the convergence rate. Based upon the splitting of $\mathbf{B}$ as in Eq. 4.39 a corresponding Gauss-Seidel iteration algorithm, see Eqs. 1.74 and 1.75, is given by

$$(\mathbf{D}+\mathbf{L})\mathbf{m}^{(n)} = \mathbf{f} - \mathbf{L}^{\dagger}\mathbf{m}^{(n-1)}. \tag{4.41}$$

This set of linear equations is solved by forward substitution. Writing this out in terms of matrix elements, using the matrix elements of $\mathbf{A}$ and $\mathbf{T}$, this becomes

$$\mu_{ia}^{(n)} = \sum_{b=x,y,z} \alpha_{i,ab} \times \left[F_{ib} + \sum_{c=x,y,z}\left(\sum_{j=1}^{i-1} T_{ib,jc}\mu_{jc}^{(n)} + \sum_{j=i+1}^{N_D} T_{ib,jc}\mu_{jc}^{(n-1)}\right)\right]. \tag{4.42}$$

As always with forward substitution, the components of the dipole vector $\mathbf{m}$ are not updated simultaneously, but sequentially. Updating the dipole of molecule i, all the updated dipoles of molecules j, with $j < i$, are used.

As discussed in Sec. 1.3.2.2, Gauss-Seidel iteration is guaranteed to converge if $\mathbf{B} = \mathbf{A}^{-1} - \mathbf{T}$ is positive-definite. Whereas the latter is mathematically not true for any $\mathbf{B}$, one can argue that it is true for a $\mathbf{B}$ representing a system that is physically realized. The polarization energy of Eq. 4.29 must be negative for any $\mathbf{f}$ that represents a physically reasonable field, since an external field has to perform work in order to polarize the medium. Using Eqs. 4.28 and 4.38 one can write

$$\mathcal{E}_p = -\frac{1}{2}\mathbf{m}^{\dagger}\mathbf{B}\mathbf{m} < 0, \tag{4.43}$$

implying $\mathbf{m}^{\dagger}\mathbf{B}\mathbf{m} > 0$ for any physically reasonable $\mathbf{m}$. In other words, the matrix $\mathbf{B}$ is positive definite in the space of physically reasonable $\mathbf{m}$. In this space, the Gauss-Seidel algorithm always converges, in contrast to the Jacobi algorithm. A disadvantage of the Gauss-Seidel algorithm is that it is not easy to parallelize it, since the updates in Eq. 4.42 are done sequentially.

Improved iterative scheme Both Jacobi iteration, Eq. 4.30, as well as Gauss-Seidel iteration, Eq. 4.41, give $\mathbf{m}^{(n)}$ solely in terms of $\mathbf{m}^{(n-1)}$, i.e. they are one-step memory techniques. The problem is actually well-suited for applying Pulay iteration, Eqs. 1.89-1.93. One can use the Jacobi rule, Eq. 4.30, to generate the update

$$\Delta\mathbf{m}^{(n)} = \mathbf{m}^{(n)} - \mathbf{m}^{(n-1)} = \mathbf{A}\mathbf{f} + (\mathbf{A}\mathbf{T} - \mathbf{I})\,\mathbf{m}^{(n-1)}, \tag{4.44}$$

and use these updates to construct the Pulay matrix, Eq. 1.90. Actually, Pulay iteration works very well here. It converges in cases where Jacobi iteration does not converge at all [33].

Details: to be complete, there are also other fast and robust techniques to solve the SCPF problem. Conjugate gradient is one of them, BFGS can be another. Both are discussed in the lecture notes on optimization.

Chapter 5

Quantum Mechanics and Electrodynamics

In this chapter I discuss some examples of eigenvalue problems as they occur in physics. Most of these examples are taken from quantum mechanics, as that happens to be my trade.[1] Sec. 5.1 discusses an elementary linear eigenvalue problem you know from elementary quantum mechanics, an electron in a radial potential. Sec. 5.2 discusses a less elementary linear eigenvalue problem, which concerns the rotational states of a molecule.

In Secs. 5.3, 5.4, and 5.6 I discuss a couple of non-linear eigenvalue problems, all of them resulting from the coupling between electrons and phonons in condensed matter. One of these is essentially a one-particle problem, where the electron-phonon coupling leads to a new kind of (quasi-)particle, called the polaron, see Sec. 5.3. As always in quantum mechanics, particles can interact with one another, and can form a compound particle, which in case of two interacting polarons is called a bipolaron, see Sec. 5.4. This also has an interesting mathematical connection to the formation of bonds in molecules, see Sec. 5.4.3. Finally, electron-phonon coupling is responsible for the formation of Cooper pairs, which are at the origin of the phenomenon of superconductivity, see Sec. 5.6.

Last, but not least, Sec. 5.5 is a bit of an outlier, as it discusses a non-linear eigenvalue problem borrowed from optics, the emergence of solitons in optical fibers. Mathematically, solitons in optical fibers are closely related to polarons in condensed matter, as both phenomena are described by the non-linear Schrödinger equation.

5.1 Atoms: the radial Schrödinger equation

Atoms have a complicated electronic structure. In the simplest approximation one assumes that each electron moves independently of all other electrons in an effective potential $V(\mathbf{r})$. We assume that this potential only depends on the relative distance, $r = |\mathbf{r}|$, between the electron and the nucleus, i.e., $V(\mathbf{r}) = V(r)$. The Schrödinger equation for the electron then becomes [34]

$$\left\{-\frac{\hbar^2}{2\mu}\nabla^2 + V(r)\right\}\psi(\mathbf{r}) = E\psi(\mathbf{r}), \tag{5.1}$$

[1] It allows me to type this part of the notes in my backyard from memory, without needing to access the library or the internet.

where μ is the (effective) mass of the electron. We know how to solve this equation. Switch to spherical coordinates

$$\nabla^2 = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2}{\partial\varphi^2}. \tag{5.2}$$

Smart 19th century physicists and mathematicians have solved the following eigenvalue equation

$$\left\{\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right) + \frac{1}{\sin^2\theta}\frac{\partial^2}{\partial\varphi^2}\right\}Y_m^l(\theta,\varphi) = -l(l+1)Y_m^l(\theta,\varphi), \tag{5.3}$$

and they have shown that this equation only has physically well-behaved solutions if $l = 0, 1, 2, ..,$ and $m = -l, -l+1, ..., l$. The solutions $Y_m^l(\theta,\varphi)$ are called *spherical harmonics*. In atomic physics, the functions for $l = 0, 1, 2, 3$ are known as s, p, d, f functions, respectively.

We use this knowledge to try a separation-of-variables solution $\psi(r,\theta,\varphi) = \chi(r)Y_m^l(\theta,\varphi)$ to Eq. 5.1. Using Eqs. 5.2 and 5.3 we then arrive at an equation for $\chi(r)$

$$\left\{-\frac{\hbar^2}{2\mu}\left[\frac{d^2}{dr^2} + \frac{2}{r}\frac{d}{dr} - \frac{l(l+1)}{r^2}\right] + V(r)\right\}\chi(r) = E\chi(r), \tag{5.4}$$

where the partial derivative has become a normal derivative, as there is only one coordinate left. The combination of first- and second derivatives in Eq. 5.4 is a bit of a nuisance. Numerical approaches for solving a differential equation, become cleaner if one has to deal with a second derivative only.

There is a standard trick to get rid of the first derivative. Define a function $\zeta(r) = r^\alpha\chi(r)$. Then

$$\frac{d^2\zeta}{dr^2} = r^\alpha\frac{d^2\chi}{dr^2} + 2\alpha r^{\alpha-1}\frac{d\chi}{dr} + \alpha(\alpha-1)r^{\alpha-2}\chi. \tag{5.5}$$

Choosing $\alpha = 1$, one can rewrite Eq. 5.4 as

$$\left\{-\frac{\hbar^2}{2\mu}\frac{d^2}{dr^2} + \frac{\hbar^2 l(l+1)}{2\mu r^2} + V(r)\right\}\zeta(r) = E\zeta(r), \tag{5.6}$$

which looks like a one-dimensional Schrödinger equation with an effective potential

$$V_{eff}(r) = \frac{\hbar^2 l(l+1)}{2\mu r^2} + V(r). \tag{5.7}$$

The domain is of course restricted to $r \in [0,\infty)$. Note that $\lim_{r\to 0}\zeta(r) = 0$ if $\chi(r)$ is a well-behaved, bounded, function, which it should be, as we would like to to be able to construct well-behaved probability distributions from a wave function.

The simplicity of Eq. 5.6 is deceptive. It cannot be solved analytically for a general potential $V(r)$. Only for special potentials such as $V(r) \propto r^{-1}$ (as in the Hydrogen atom), or $V(r) \propto r^2$ (as in the harmonic oscillator) we can solve the problem using pen and paper only. Before you start moaning about quantum mechanics, the situation in classical mechanics is no different. Also Newton's equations can only be solved for special potentials, often the same ones as in quantum mechanics. For instance, the solution for $V(r) \propto r^{-1}$ gives you Keppler's laws of planetary motion. As Newton has already found that solution, you may wonder how much progress we have actually made in three centuries. Not much, but the computer has been invented. That allows you to solve Eq. 5.6 numerically for any potential $V(r)$.[2]

[2] A mathematician would object, as for sure he/she can come up with a potential for which it is impossible to find a solution. Physicists call such potentials "unreasonable". Reasonable people avoid those.

5.1.1 Numerical approach

Consider the radial Schrödinger equation, Eq. 5.6. We are interested in bound states, which gives us the conditions

$$E < V(\infty); \quad \lim_{r\to\infty} \zeta(r) = 0; \quad \lim_{r\to 0} \zeta(r) = 0. \tag{5.8}$$

We will solve this eigenvalue problem numerically. We know that Eq. 5.6 only has solutions obeying the boundary conditions Eq. 5.8, for certain discrete energies $E = E_n$ that can be labeled by an integer quantum number n.[3] These E_n are then the eigenvalues and the corresponding solutions $\zeta_n(r)$ are the eigenfunctions.

Before one starts, it is useful to convert the equation into one based on dimensionless variables. These should be chosen such, that one does not get into numerical troubles right from the start (e.g., numbers that become too small or too large for the computer to represent). It is useful to introduce a parameter

$$r_0 = \hbar/\sqrt{2\mu V_0}, \tag{5.9}$$

where V_0 is a constant of dimension energy that is a sensible unit for the strength of the potential.[4] The parameter r_0 is then a real positive number of dimension meter, and is a natural unit of length in the problem. If we introduce a dimensionless radius $\rho = r/r_0$, the wave function and the potential can be expressed as $\zeta(\rho)$ and $V(\rho)$.[5] This allows one to rewrite Eq. 5.6 as

$$\frac{d^2\zeta(\rho)}{d\rho^2} = \{W(\rho) - \lambda\}\,\zeta(\rho) \quad \text{with} \quad W(\rho) = \frac{V(\rho)}{V_0} + \frac{l(l+1)}{\rho^2} \quad \text{and} \quad \lambda = \frac{E}{V_0}, \tag{5.10}$$

with the constraints

$$\lambda < V(\infty); \quad \lim_{\rho\to\infty} \zeta(\rho) = 0; \quad \lim_{\rho\to 0} \zeta(\rho) = 0. \tag{5.11}$$

We discretize Eq. 5.10 on an equidistant grid $\rho_j = jh;\ j = 0, 1, .., N$, where h is the step size, and N is such that $\zeta(Nh) \approx 0$. For the second derivative we start from the central difference approximation

$$\frac{d^2\zeta(\rho)}{d\rho^2} \approx \frac{1}{h}\left[\frac{d\zeta(\rho + h/2)}{d\rho} - \frac{d\zeta(\rho - h/2)}{d\rho}\right] = \frac{1}{h}\left[\left(\frac{\zeta(\rho+h) - \zeta(\rho)}{h}\right) - \left(\frac{\zeta(\rho) - \zeta(\rho-h)}{h}\right)\right]. \tag{5.12}$$

This then transforms Eq. 5.10 into

$$\begin{cases} [\zeta(\rho_{j+1}) - 2\zeta(\rho_j) + \zeta(\rho_{j-1})]/h^2 = \{W(\rho_j) - \lambda\}\,\zeta(\rho_j); \quad j = 1, ..., N-1 \\ \zeta(\rho_0) = 0;\ \zeta(\rho_N) = 0. \end{cases}. \tag{5.13}$$

[3] The energies are "quantized"; after all, it is quantum mechanics. The phenomenon is quite general however. If you enclose electromagnetic waves or acoustic waves in a cavity, then standing waves are the only possible sustainable solutions, and these can be found only at certain discrete frequencies.

[4] If the potential has a minimum and $V(r \to \infty) = 0$, $V_0 = V_{\min}$, would a sensible choice. For the Coulomb potential, which does not have a minimum, one can choose the Rydberg as a unit of energy. For the harmonic oscillator, where $V(r \to \infty) \to \infty$, $V_0 = \hbar\omega$ would make sense.

[5] Mathematicians tell me I should have used different symbols, but I have enough of them already. The idea is to express the wave and the potential as a function of a dimensionless variable.

We rewrite this in matrix-vector form

$$\mathbf{Tz} = \lambda \mathbf{z}; \text{ with } \mathbf{T} = \frac{1}{h^2} \begin{pmatrix} 2+h^2W(\rho_1) & -1 & 0 & \cdots & 0 & 0 \\ -1 & 2+h^2W(\rho_2) & -1 & \cdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -1 & 2+h^2W(\rho_{N-1}) \end{pmatrix}, \tag{5.14}$$

a tridiagonal $(N-1) \times (N-1)$ matrix, and

$$\mathbf{z} = \begin{pmatrix} \zeta(\rho_1) \\ \zeta(\rho_2) \\ \vdots \\ \zeta(\rho_{N-1}) \end{pmatrix}; \tag{5.15}$$

a vector of length $N-1$. Note that we don't have to enforce the boundary conditions explicitly, they are included in the matrix eigenvalue problem of Eq. 5.14. We can solve this problem using numerical techniques. A typical number of grid points $N = 10^2$-10^3, and we typically wish to have the $M = 10^0$-10^1 lowest eigenvalues.

5.2 Rotation of molecules

Many properties is physics are described by a symmetric tensor of rank 2

$$\boldsymbol{\alpha} = \begin{pmatrix} \alpha_{xx} & \alpha_{xy} & \alpha_{xz} \\ \alpha_{yx} & \alpha_{yy} & \alpha_{yz} \\ \alpha_{zx} & \alpha_{zy} & \alpha_{zz} \end{pmatrix} \text{ with } \alpha_{pq} = \alpha_{qp};\ p = x,y,z;\ q = x,y,z. \tag{5.16}$$

For instance, the dipole $\mathbf{d}$ of a molecule induced by an external electric field $\mathbf{E}$ is given to a good accuracy by

$$d_p = \sum_{q=x,y,z} \alpha_{pq} E_q;\ p = x,y,z;$$

if the field $\mathbf{E}$ is not extremely large. The tensor $\boldsymbol{\alpha}$ is called the polarizability tensor. Other examples are the dielectric tensor $\boldsymbol{\epsilon}$, which gives the relation between the electric displacement and the electric field $\mathbf{D} = \boldsymbol{\epsilon}\mathbf{E}$ (which should be interpreted as a matrix-vector multiplication, with $\boldsymbol{\epsilon}$ a matrix of the type of Eq. 5.16), or the elasticity tensor $\boldsymbol{\chi}$, which gives the relation between strain and stress if a solid body is deformed by external forces.

We will look at an example inspired by classical mechanics [35]. Suppose we have a body consisting of N point masses $m_n;\ n = 1,...,N$, each at a position $\mathbf{r}_n = (x_n, y_n, z_n);\ n = 1,...,N$. The inertia tensor is defined by the elements

$$I_{pq} = \sum_{n=1}^{N} m_n \left(\delta_{pq} r_n^2 - p_n q_n\right);\ p = x,y,z;\ q = x,y,z;\ r_n = \sqrt{x_n^2 + y_n^2 + z_n^2}. \tag{5.17}$$

Suppose the body rotates around an axis with angular velocity $\boldsymbol{\omega}$, such that the velocity of each point mass is given by $\mathbf{v}_n = \boldsymbol{\omega} \times \mathbf{r}_n$. From the definition of total angular momentum $\mathbf{J} = \sum_{n=1}^{N} m_n (\mathbf{r}_n \times \mathbf{v}_n)$ and some algebra, one can derive

$$\mathbf{J} = \mathbf{I}\boldsymbol{\omega} \text{ or in components } J_p = \sum_{q=x,y,z} I_{pq}\omega_q;\ p = x,y,z, \tag{5.18}$$

where the matrix $\mathbf{I}$ is the inertia tensor with elements I_{pq} given by Eq. 5.17. Likewise, the kinetic energy $E = \frac{1}{2}\sum_{n=1}^{N} m_n |\mathbf{v}_n|^2$ can be rewritten as

$$E = \frac{1}{2}\boldsymbol{\omega}^T \mathbf{I}\boldsymbol{\omega} \quad \text{or} \quad E = \frac{1}{2}\sum_{p=x,y,z}\sum_{q=x,y,z} \omega_p I_{pq} \omega_q. \tag{5.19}$$

Newton's equation of motion for a rotating body becomes $d\mathbf{J}/dt = \mathbf{N}$, where $\mathbf{N} = \sum_{n=1}^{N}(\mathbf{r}_n \times \mathbf{F}(\mathbf{r}_n))$ is called the torque, and $\mathbf{F}$ is an external force.[6] The general rotational motion of a body can be decomposed into (i) a rotation around an axis going through its center of mass, and (ii) a motion of its center of mass, which involves the body's total mass [35].[7] We are not interested in (ii), and will only consider (i). This means that in the following we will take the positions $\mathbf{r}_n$; $n = 1, ..., N$ of the particles are with respect to the center of mass of the whole system. Nevertheless, the motion of rotating bodies can be horribly complicated,[8] but luckily we have a way out.

We are interested in the rotational motion of small bodies, such as molecules [36, 37]. For small objects one has to use quantum mechanics instead of classical mechanics, and, as usual, quantum mechanics is easier than classical mechanics. One has to convert the energy of Eq. 5.19 into a Hamiltonian. However, as you know, velocities do not come into in a Hamiltonian, nor do angular velocities. Momenta do, as do angular momenta. One can invert Eq. 5.18, $\boldsymbol{\omega} = \mathbf{I}^{-1}\mathbf{J}$, and use this in Eq. 5.19 to obtain

$$\hat{\mathbf{H}} = \frac{1}{2}\hat{\mathbf{J}}^T \left(\mathbf{I}^{-1}\right)^T \hat{\mathbf{J}}, \tag{5.20}$$

where, in accordance with quantum mechanics, we have turned the energy into an operator $E \to \hat{\mathbf{H}}$ (the Hamiltonian), and the angular momentum into an operator $\mathbf{J} \to \hat{\mathbf{J}}$ (the angular momentum operator). Written out in components, the Hamiltonian of Eq. 5.20 is

$$\hat{\mathbf{H}} = \frac{1}{2}\sum_{p=x,y,z}\sum_{q=x,y,z} \hat{J}_p \left(\mathbf{I}^{-1}\right)_{qp} \hat{J}_q, \tag{5.21}$$

which does not look very attractive.

Things would be much nicer if the inertia tensor would be diagonal, i.e., $I_{pq} = \delta_{pq} I_p$. The inverse of a diagonal matrix is simple $(\mathbf{I}^{-1})_{pq} = \delta_{pq}/I_p$, and so is $(\mathbf{I}^{-1})^T_{pq} = \delta_{pq}/I_p$. The Hamiltonian of Eq. 5.21 then becomes the relatively simple

$$\hat{\mathbf{H}} = \frac{1}{2}\sum_{p=x,y,z} \frac{\hat{J}_p^2}{I_p}. \tag{5.22}$$

The problem is that for a general distribution of masses in space, the inertia tensor of Eq. 5.17 is not diagonal, which would means that the simplification from Eq. 5.21 to Eq. 5.22 is not possible.

[6] Torque is one of the favorite topics of "Top Gear", where you can see by the look upon the presenters' faces that they haven't got a clue what they are talking about. Having said that, they always look like that.

[7] Think of the motion of the earth, rotating around its axis, and orbiting the sun.

[8] Felix Klein and Arnold Sommerfeld wrote a massive four volume book series on this topic only, called *Über die Theorie des Kreisels*. Less than 10% of the people who cite the books have read any of the volumes (including me).

We have, however, a trick up our sleeve. The inertia tensor is symmetric, i.e., $I_{pq} = I_{qp}$, and a symmetric matrix can always be transformed into a diagonal matrix by a unitary transformation[9]

$$\mathbf{Q}^T\mathbf{I}\mathbf{Q} = \begin{pmatrix} I_1 & 0 & 0 \\ 0 & I_2 & 0 \\ 0 & 0 & I_3 \end{pmatrix}. \tag{5.23}$$

The numbers I_i; $i = 1, 2, 3$ are called the *principal values* of the tensor, in this case the *principal moments of Inertia*. The three columns $\mathbf{q}_j$; $j = 1, 2, 3$ of the unitary transformation $\mathbf{Q}$, with elements $(\mathbf{q}_j)_i = Q_{ij}$, describe a system of axes, which are called the *principal axes*.

It is easy to see that the same transformation also diagonalizes the inverse inertia tensor, i.e., $\left(\mathbf{Q}^T\mathbf{I}^{-1}\mathbf{Q}\right)_{ij} = \delta_{ij}/I_i$. Rest us to use the same transformation on the angular momenta, and define $\hat{\mathbf{J}}' = \mathbf{Q}^T\hat{\mathbf{J}}$. With these transformations, the Hamiltonian of Eq. 5.21 becomes

$$\hat{\mathbf{H}} = \frac{1}{2}\sum_{i=1}^{3}\frac{\hat{J}_i'^2}{I_i}. \tag{5.24}$$

This is as simple as we can make it. The operators $\hat{J}_i' = \mathbf{q}_i^T\hat{\mathbf{J}}$ are the projections of the angular momentum vector on the principal axes, or, in other words, the components of the angular momentum operator with respect to the principal axes.

5.2.1 The symmetric top

To indicate how the eigenvalues of this Hamiltonian can be found, let us consider the special case where $I_1 = I_2 \neq I_3$. In the rotational motion folklore this is called a *symmetric top*.[10] The Hamiltonian then becomes

$$\hat{\mathbf{H}} = \frac{\hat{J}_1'^2 + \hat{J}_2'^2}{2I_1} + \frac{\hat{J}_3'^2}{2I_3} = \frac{\hat{J}'^2}{2I_1} + \hat{J}_3'^2\left(\frac{1}{2I_3} - \frac{1}{2I_1}\right), \tag{5.25}$$

where $\hat{J}'^2 = \hat{J}_1'^2 + \hat{J}_2'^2 + \hat{J}_3'^2$ is the total angular momentum operator. One can treat this as the $\hat{L}^2$ angular momentum operator you know from your introductory quantum mechanics classes [34]. The operator $\hat{J}'^2$ then has eigenvalues $\hbar^2 J(J+1)$, with $J = 0, 1, ...$ an non-negative integer. Likewise, $\hat{J}_3'$ can be treated as the $\hat{L}_z$ angular momentum operator you know [34]. Obviously $\hat{J}'^2$ and $\hat{J}_3'$ are then commuting operators, and their common eigenstates can be labeled by two quantum numbers $|JK\rangle$.[11] The operator $\hat{J}_3'$ has eigenvalues $\hbar K$, with

[9] A symmetric matrix is a real Hermitian matrix, so $\mathbf{A}^\dagger = \mathbf{A}^T = \mathbf{A}$. It can be diagonalized using a real unitary matrix, so $\mathbf{Q}^\dagger = \mathbf{Q}^T = \mathbf{Q}^{-1}$. The proof of this statement is similar to the proof used for complex Hermitian matrices. A real unitary matrix is also called an orthogonal matrix. Its columns $\mathbf{q}_j$; $j = 1, 2, 3$ with elements $(\mathbf{q}_j)_i = Q_{ij}$ form a set of orthogonal and normalized vectors, i.e., $\mathbf{q}_i^T\mathbf{q}_j = \delta_{ij}$.

[10] In this context "top" translates into Dutch as "tol". Most "tollen" that are used as children's toys are symmetric.

[11] For the quantum purists: it takes three angles to define the orientation of a body in space (consider the earth: it takes two angles to define the orientation of its rotation axis in space, and a further angle to define the position of the Greenwich meridian), i.e., one has three degrees of freedom. A quantum state should thus have three quantum numbers $|JKM\rangle$ (instead of just two). A full analysis of the rotational problem gives $M = 0, \pm 1, ..., \pm J$. This third quantum number is a boring one, however, as the energy levels do not depend on it, see Eq. 5.26, so I have omitted it. It does play a role, however, if one needs to count energy levels, as in statistical physics. So each level we calculate here has a $2J + 1$ degeneracy [36, 37].

$K = 0, \pm 1, ..., \pm J$ an integer that takes the $2J+1$ values between $-J$ and J. The eigenvalues of the Hamiltonian of Eq. 5.25 are then given by

$$E_{JK} = \frac{\hbar^2 J(J+1)}{2I_1} + \hbar^2 K^2 \left(\frac{1}{2I_3} - \frac{1}{2I_1} \right) \quad \text{with} \quad J = 0, 1, ...; \; K = 0, \pm 1, ..., \pm J. \tag{5.26}$$

That's it, the whole spectrum. Simple, isn't it?[12] This simplicity hangs on our ability to perform the transformation of Eq. 5.23, and here is one point where numerical linear algebra comes in handy.

It is clear from Eq. 5.26 that the spectrum of a molecule where $I_1 = I_2 = I_3$, is particularly simple.

$$E_J = \frac{\hbar^2 J(J+1)}{2I_1} \quad \text{with} \quad J = 0, 1, ...; \tag{5.27}$$

a non-negative integer. Such a molecule is called a *spherical top*. The name is a little misleading. The principal moments of inertia of a sphere certainly obey $I_1 = I_2 = I_2$. However, there are other objects with less symmetry that also have this property, such as a cube, a tetrahedron, or an octahedron. As far as their quantum mechanical rotational motion is concerned, all these objects behave the same.

5.2.2 The asymmetric top

If $I_1 \neq I_2 \neq I_3$, the molecule is called an *asymmetric top*. The eigenvalues of its Hamiltonian, Eq. 5.24, no longer have a simple analytical expression like Eq. 5.26, but it is no problem to find the eigenvalues numerically. I will show you how. As $\hat{J}'_3$ can be treated as the $\hat{L}_z$ angular momentum operator you know from introductory quantum mechanics, it is logical to assume that $\hat{J}'_1$ and $\hat{J}'_2$ can be treated as $\hat{L}_x$ and $\hat{L}_y$, respectively. This means that we can write them as $\hat{J}'_1 = \frac{1}{2}\left(\hat{J}'_+ + \hat{J}'_-\right)$ and $\hat{J}'_2 = \frac{1}{2i}\left(\hat{J}'_+ - \hat{J}'_-\right)$, where $\hat{J}'_\pm = \hat{J}'_1 \pm i\hat{J}'_2$ are the angular momentum step-up/step-down operators with properties [34]

$$\hat{J}'_\pm |JK\rangle = \hbar\sqrt{J(J+1) - K(K \pm 1)}\,|JK \pm 1\rangle \tag{5.28}$$

After some algebra, the Hamiltonian of Eq. 5.24 can be rewritten as

$$\hat{\mathbf{H}} = \frac{\hat{J}'^2}{4}\left[\frac{1}{I_1} + \frac{1}{I_2}\right] + \frac{\hat{J}_3'^2}{4}\left[\frac{2}{I_3} - \frac{1}{I_1} - \frac{1}{I_2}\right] + \frac{\hat{J}_+'^2 + \hat{J}_-'^2}{8}\left[\frac{1}{I_1} - \frac{1}{I_2}\right]. \tag{5.29}$$

The last term of this expression makes it an asymmetric top. If $I_1 = I_2$, we are back to the Hamiltonian of the symmetric top, Eq. 5.25. From Eq. 5.28 we deduce that

$$\hat{J}'^2_\pm |JK\rangle = \hbar^2 \left[J(J+1) - K(K \pm 1)\right]^{\frac{1}{2}} \left[J(J+1) - (K \pm 1)(K \pm 2)\right]^{\frac{1}{2}} |JK \pm 2\rangle. \tag{5.30}$$

This means that

$$\hat{\mathbf{H}}|JK\rangle = \alpha(J,K)|JK\rangle + \gamma_+(J,K)|JK+2\rangle + \gamma_-(J,K)|JK-2\rangle, \tag{5.31}$$

[12] If you are still not convinced that quantum mechanics is much easier than classical mechanics, look up the classical theory for the motion of a symmetric top (something for Halloween).

with

$$\begin{aligned}\alpha(J,K) &= \frac{\hbar^2 J(J+1)}{4}\left[\frac{1}{I_1}+\frac{1}{I_2}\right]+\frac{\hbar^2 K^2}{4}\left[\frac{2}{I_3}-\frac{1}{I_1}-\frac{1}{I_2}\right] \quad \text{and} \\ \gamma_{\pm}(J,K) &= \frac{\hbar^2}{8}\left[\frac{1}{I_1}-\frac{1}{I_2}\right]\left[J(J+1)-K(K\pm 1)\right]^{\frac{1}{2}}\left[J(J+1)-(K\pm 1)(K\pm 2)\right]^{\frac{1}{2}}.\end{aligned} \tag{5.32}$$

Note from Eq. 5.31 that J is still a good quantum number, but K is not anymore, as $|JK\rangle$ is not an eigenstate of the Hamiltonian. Note furthermore that

$$\gamma_-(J,K+2) = \gamma_+(J,K), \tag{5.33}$$

so we need only calculate explicitly $\gamma_+(J,K)$ for all K.

We can find the eigenstates as follows. Write the eigenstates as $|\Psi_{Jn}\rangle = \sum_{K=-J}^{J} c_{K,n}^{(J)}|JK\rangle$; then the coefficients $c_{K,n}^{(J)}$ can be found from the eigenvalue equation

$$\sum_{K'=-J}^{J} H_{KK'}^{(J)} c_{K',n}^{(J)} = E_n^{(J)} c_{K,n}^{(J)};\ K = -J, -J+1, \ldots J-1, J, \tag{5.34}$$

with $H_{KK'}^{(J)} = \left\langle JK\left|\hat{\mathbf{H}}\right|JK'\right\rangle$ matrix elements of the Hamiltonian, and $E_n^{(J)}$ the eigenvalues. Assembling the $2J+1$ coefficients $c_{K,n}^{(J)}$ in a column vector $\mathbf{c}_n^{(J)}$, Eq. 5.34 can be written as a matrix eigenvalue equation $\mathbf{H}^{(J)}\mathbf{c}_n^{(J)} = E_n^{(J)}\mathbf{c}_n^{(J)}$. The matrix elements $H_{KK'}^{(J)}$ of the $(2J+1)\times(2J+1)$ matrix $\mathbf{H}^{(J)}$ can be deduced from Eqs. 5.31 and 5.32, and we can write Eq. 5.34 as

$$\begin{pmatrix} \alpha(J,-J) & 0 & \gamma_+(J,-J) & 0 & \cdots & 0 \\ 0 & \alpha(J,-J+1) & 0 & \gamma_+(J,-J+1) & \cdots & 0 \\ \gamma_+(J,-J) & 0 & \alpha(J,-J+2) & 0 & \cdots & 0 \\ 0 & \gamma_+(J,-J+1) & 0 & \alpha(J,-J+3) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \alpha J(J,J) \end{pmatrix} \begin{pmatrix} c_{-J,n}^{(J)} \\ c_{-J+1,n}^{(J)} \\ c_{-J+2,n}^{(J)} \\ c_{-J+3,n}^{(J)} \\ \vdots \\ c_{J,n}^{(J)} \end{pmatrix}$$
$$= E_n^{(J)} \begin{pmatrix} c_{-J,n}^{(J)} \\ c_{-J+1,n}^{(J)} \\ c_{-J+2,n}^{(J)} \\ c_{-J+3,n}^{(J)} \\ \vdots \\ c_{J,n}^{(J)} \end{pmatrix}. \tag{5.35}$$

Note that this $(2J+1)\times(2J+1)$ matrix is nicely symmetric, so it can be diagonalized using the methods we have described in Part I. It has $2J+1$ eigenvalues $E_n^{(J)};\ n = 1, 2, \ldots, 2J+1$, which represent the energy levels of the asymmetric top for a particular J. Solving the eigenvalue problem for each $J = 0, 1, \ldots$ then gives all the energy levels of the asymmetric top.

5.2.3 The water molecule

One of the simplest asymmetric tops is the water molecule H_2O. It consists of two hydrogen atoms with mass m_H and one oxygen atom with mass m_O in a bend geometry as in Fig. 5.1.

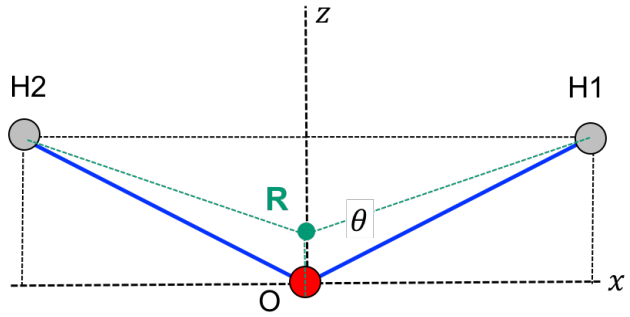


Figure 5.1: The water molecule H_2O consists of two hydrogen atoms H1 and H2 and one oxygen atom O in a bend geometry. We choose a coordinate system where the molecule is in the xz-plane, with θ the angle between the O-H1 bond and the z-axis. The center of mass of the molecule, Eq. 5.37, is indicated by $\mathbf{R}$ (in green).

Choosing the coordinate system as is this figure, the positions of the atoms are

$$\mathbf{r}'_O = (0,0,0); \quad \mathbf{r}'_{H1} = a(\sin\theta, 0, \cos\theta); \quad \mathbf{r}'_{H2} = a(-\sin\theta, 0, \cos\theta), \tag{5.36}$$

with a the O-H bond length. As discussed below Eq. 5.19, we need the the positions of the atoms with respect to the center of mass. The center of mass of the water molecule is given by

$$\mathbf{R} = \frac{m_O\mathbf{r}'_O + m_H\mathbf{r}'_{H1} + m_H\mathbf{r}'_{H2}}{m_O + 2m_H} = \frac{2m_H a}{m_O + 2m_H}(0,0,\cos\theta). \tag{5.37}$$

The positions of the atoms with respect to the center of mass $\mathbf{r} = \mathbf{r}' - \mathbf{R}$ are

$$\mathbf{r}_O = a\,(0,0,-f\cos\theta)\,; \quad \mathbf{r}_{H1} = a\,(\sin\theta, 0, (1-f)\cos\theta)\,; \quad \mathbf{r}_{H2} = a\,(-\sin\theta, 0, (1-f)\cos\theta)\,, \tag{5.38}$$

with $f = 2m_H/(m_O + 2m_H)$.

Using these positions, the inertia tensor, Eq. 5.17, becomes

$$\mathbf{I} = \begin{pmatrix} I_{xx} = a^2\left[m_O f^2 + 2m_H(1-f)^2\right]\cos^2\theta & 0 & 0 \\ 0 & I_{yy} = I_{xx} + I_{zz} & 0 \\ 0 & 0 & I_{zz} = 2m_H a^2 \sin^2\theta \end{pmatrix}. \tag{5.39}$$

On the positive side, this tensor is nicely diagonal, so we can use the "simple" Hamiltonian, Eq. 5.22, straight away, without having to invoke the unitary transformation of Eq. 5.23. On the negative side, $I_{xx} \neq I_{yy} \neq I_{zz}$, so the water molecule is an asymmetric top, and we have to follow the procedure outlined in Sec. 5.2.2 to find its rotational energy levels.

5.3 Polarons

The wave equations you have encountered in you bachelor physics courses are linear (partial) differential equations. This holds for the classical wave equation, describing sound waves in

matter, or electromagnetic waves, as well as for the Schrödinger equation, describing matter waves in quantum mechanics. The basic solutions to these equations are plane waves, and any solution can be constructed as a linear combination of such plane waves, using Fourier transforms.

Non-linear wave equations do exist, however, and describe some interesting physical phenomena. One of these, first described in the mid-19th century, is the emergence of a solitary wave pulse, which moves at a constant velocity, and does not change its shape over time. The latter in particular is remarkable, as the wave pulses constructed from linear waves tend to disperse over time. Apparently, non-linearity is able to overcome dispersion. Such solitary wave pulses are nowadays called *solitons.*

As often in physics, once you have found and understood a phenomenon, you can find them all over the place. Solitons were first discovered on the surface of water flowing down a narrow channel. By now we know many non-linear wave equations that give rise to solitons. For these notes, I have picked out one in particular, the *non-linear Schrödinger equation.* It describes quantum-mechanical solitons in condensed matter, for instance. Actually, many solitons of slightly different type exist in quantum mechanics. I have picked out one in particular, the *polaron*; it is described in Sec. 5.3.1 [38, 39]. I use this example to derive the non-linear Schrödinger equation. You will find this in Sec. 5.3.2, where I will also discuss its most basic solution.

Remaining true to my trade, I describe the polaron in somewhat more detail. In particular, two polarons can interact and form a bound state, called the *bipolaron*, discussed in Sec. 5.4. This discussion has a surprising connection to a physical phenomenon that has nothing to do with solitons or polarons, namely the formation of a chemical bond. Modeling the latter gives rise to non-linear Schrödinger equations that look mathematically similar to the equations describing the bipolaron. This is discussed in Sec. 5.4.3.

Somewhat closer to the original 19th century idea, Sec. 5.5 describes solitons as solitary electromagnetic wave pulses in optical fibers. Somewhat surprisingly, these solitons are also described by the non-linear Schrödinger equation, which for the fans I will rederive from the Maxwell equations.

Non-linear wave equations can be notoriously difficult to solve analytically (well, what would you expect?), and, except for the simplest cases, we will leave that job for the mathematical physicists. Numerical solutions tend to be more manageable. To restrict the computational effort required, I have restricted myself to one-dimensional (1D) examples. Real-life applications often require (at least) three dimensions, but I hope that the 1D world at least gives you an insight in the physical phenomena.

5.3.1 Polarons in condensed matter

In your classes on solid state physics you have studied the Schrödinger equation for an electron in the periodic potential of a crystal, which leads to Bloch waves. You have also studied the Schrödinger equation for vibrations of the crystal lattice, which gives phonons. You might have been left with the impression that the two subjects, electrons and phonons, are unrelated. There are good physical reasons for separating the two subjects. The electrons are light particles with a large kinetic energy (of order 10 eV), whereas phonons involve the motion of relatively heavy particles (the atomic nuclei) with a much smaller kinetic energy (of order 10 meV). In quantum mechanics, as well as in classical mechanics, the motions of particles that proceed at widely different energy scales, can be decoupled from one another

to a good degree of accuracy.[13]

Whereas a full decoupling of electrons and phonons works reasonably well for many systems, it is not appropriate for some systems. For instance, in crystals where the effective mass of the electron is large, the corresponding kinetic energy is relatively small, and of a similar scale as that of the phonons. Decoupling electrons from phonons is then a questionable approach. This is the case in molecular crystals, for instance.[14] Another example is the motion of an electron in an ionic crystal, where the "atoms" are ions with a positive or negative charge, cations and anions. Such ions generate Coulomb potentials, and when they vibrate, the potentials vibrate along. Coulomb potentials are strong, and strong vibrating potentials can strongly affect the motion of an electron. In both cases mentioned we speak of a large *electron-phonon coupling*.[15]

If the interaction between electrons and phonons is sufficiently strong, it leads to novel (quasi)particles called *polarons*. Historically, the phrase comes from ionic crystals, where the electron polarizes (in the electrostatic sense) the crystal environment, and drags this polarization along while moving through the crystal. The combined object, electron+polarization, is called a polaron. Nowadays, the term polaron is used whenever the electron-phonon interaction is strong, whether it is of Coulombic nature or not. Because of the strong electron-phonon interaction, an electron distorts the lattice, and it carries this distortion along while moving through the lattice. The "dressed"charge carrier, electron+lattice distortion, is called a polaron.

If the electron-phonon coupling is very strong, then the electron carries the surrounding distortion of the medium adiabatically (in the mechanics sense) along. In that case one can derive an effective equation of motion for the electron only, which is called the *non-linear Schrödinger equation*. One of the features of this equation is that it gives rise to a bound state (or multiple bound states), even if the electron is initially free. Because of the strong electron-phonon interaction, the electron digs a hole for itself in the elastic medium from which it cannot escape, and it becomes a self-trapped polaron.

Below I will give a full derivation of the non-linear Schrödinger equation. If you are only interested in the end result, you can go to Sec. 5.3.2 right away.

5.3.1.1 One-dimensional continuum model*

The full quantum mechanical description of polarons is horribly complicated. We will not try that here, but instead focus on a limiting case, just to illustrate the concept. If all phonon frequencies are low ($\hbar\omega$ small compared to the other energy scales in the problem), then phonons can be considered as classical degrees of freedom, i.e., as classical vibrations. If the scale of the lattice distortion is much larger than the distance between nearest neighbor atoms, then we may as well approximate the crystal lattice by a continuous elastic medium [35]. That saves us a lot of bookkeeping of atomic positions. In three dimensions one needs some non-trivial vector calculus to model an elastic medium. This can be avoided by

[13] If you have ever studied (classical) vibrations in a mechanical system, you know that slow vibrations do not influence the fast ones much, and vice versa. It is similar in quantum mechanics; the separation of motions at different energy scales can be formalized by a theory called the *Born-Oppenheimer approximation* (Oppenheimer, the atomic bomb guy, the leader of the Manhatten project,).

[14] See the lecture notes on the self-consistent polarization field.

[15] Even if the electron-phonon coupling is small, it can lead to observable effects if the energy scales one considers are small. The phenomenon of superconductivity, for instance, is entirely caused by electron-phonon coupling (at least, for conventional superconducting materials).

restricting ourselves to a one-dimensional (1D) elastic medium, which suffices to explain the essential physics.

Below I will derive the equations of motion of a 1D quantum mechanical electron, embedded in a classical 1D elastic medium, in the presence of an electron-phonon interaction. The result will be two coupled differential equations, one for the electron, and one for the medium. We will not try to solve these in general, but instead focus on one particular case, where the electron-phonon coupling is so strong, that the electron carries the distortion of the medium along adiabatically. In that case the motion of the electron only is described by the non-linear Schrödinger equation.

One-dimensional (1D) elastic medium. We start from a confined medium that stretches from $x = a$ to $x = b > a$, where we can choose the interval $[a; b]$ anyway we like, see Fig. 5.2.[16] A homogeneous elastic medium in rest has a constant mass density μ (dimension kg/m). Discretizing the medium into $N + 1$ blocks of size Δx, each block then has a mass $m = \mu\Delta x$. In rest the blocks are situated at positions $x_n = a + n\Delta x,\ n = 0, ..., N$. We deform the medium by displacing each block to $x_n + u_n,\ n = 0, ..., N$, with u_n describing the displacement with respect to the rest situation. One needs boundary conditions for u_0 and u_N, but we will worry about those later.

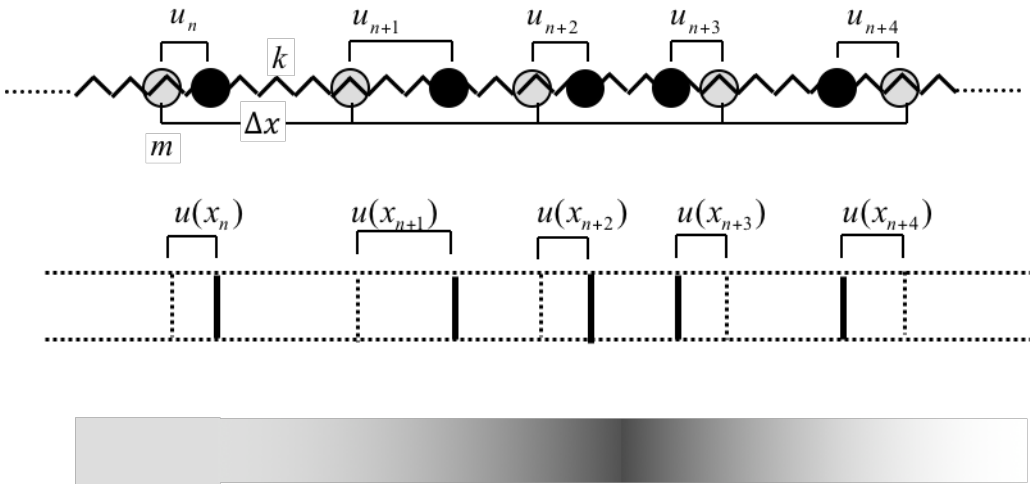


Figure 5.2: Artist (not a very good one) impression of a 1D elastic medium. Top: (gray) discrete masses m, in equilibrium at equal distances Δx, connected by springs with spring constants k; (black) masses displaced by u_n, giving rise to a kinetic energy, Eq. 5.40, and a potential energy, Eq. 5.41. Middle: a continuous elastic medium is obtained by $\lim_{\Delta x\to 0}$, while $\lim_{\Delta x\to 0} m/\Delta x = \mu$, and $\lim_{\Delta x\to 0} k/\Delta x = \kappa$, which are materials' constants, called the *mass density* and the *elasticity*, respectively. This gives rise to the total energy of Eq. 5.47. Bottom: gray scale impression of the displacement field of the elastic medium.

The kinetic energy T associated with such displacements is obtained by summing over the kinetic energies of all blocks

$$T = \frac{1}{2}\sum_{n=0}^{N} m\left(\frac{du_n}{dt}\right)^2 = \frac{1}{2}\sum_{n=0}^{N}\mu\left(\frac{du_n}{dt}\right)^2\Delta x. \tag{5.40}$$

[16] Usually we take the interval as $[-\infty; \infty]$. I always get into trouble with infinities, so I prefer to take a finite interval, and take the limit to infinity after I have all the results I want.

The potential energy U associated with the displacements is constructed by assuming that all blocks are connected to their neighbors by harmonic springs with spring constants k.[17]

$$U = \frac{1}{2}\sum_{n=0}^{N-1} k\left(u_{n+1}-u_n\right)^2 = \frac{1}{2}\sum_{n=0}^{N-1}\frac{\kappa}{\Delta x}\left(u_{n+1}-u_n\right)^2 = \frac{1}{2}\sum_{n=0}^{N-1}\kappa\left(\frac{u_{n+1}-u_n}{\Delta x}\right)^2\Delta x, \quad (5.41)$$

with $\kappa\Delta x = k$, where κ (dimension N/m^2), is a materials' property called the *elasticity*, which for a linear elastic medium is a constant. Each block moves according to Newton's equation of motion

$$m\frac{d^2u_n}{dt^2} = F_n;\ n = 0, ..., N, \quad (5.42)$$

where the force F_n on a single block n is given by the two springs connecting the block to its neighbors

$$F_n = k\left(u_{n+1}-u_n\right) - k\left(u_n - u_{n-1}\right) = \kappa\left[\frac{u_{n+1}+u_{n-1}-2u_n}{\Delta x^2}\right]\Delta x. \quad (5.43)$$

If we take the limit $\Delta x \to 0$, and at the same time $N \to \infty$, then the medium becomes continuous, and the displacement becomes a continuous function $u(x,t)$, which is called a *field* in physics. The equation of motion, Eqs. 5.42 and 5.43, becomes

$$\mu\frac{\partial^2 u}{\partial t^2} = \kappa\frac{\partial^2 u}{\partial x^2}. \quad (5.44)$$

This is a classical wave equation of the type

$$\frac{1}{v^2}\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}, \quad \text{with} \quad v = \sqrt{\frac{\kappa}{\mu}}, \quad (5.45)$$

the speed of the waves. The "phonons" in an elastic medium are *sound waves*. Depending on the boundary conditions at a and b one can have standing waves, or traveling waves. The simplest traveling wave is a harmonic wave $u(x,t) = A\cos\left(kx - \omega t\right)$. Trying this in the wave equation, Eq. 5.45, one finds the *linear dispersion relation*

$$\omega = kv. \quad (5.46)$$

The energy stored in the wave can be obtained by $\lim_{\Delta x\to 0}$ in Eqs. 5.40 and 5.41

$$H = T + U = \int_a^b \left[\frac{1}{2}\mu\left(\frac{\partial u}{\partial t}\right)^2 + \frac{1}{2}\kappa\left(\frac{\partial u}{\partial x}\right)^2\right]dx. \quad (5.47)$$

All of this is classical mechanics up till now. One might transform it into quantum mechanics by defining a proper Hamiltonian operator $\hat{H}$. As $u(x,t)$ is a field, this then becomes an example of a *quantum field theory*. This complication is for other courses;[18] here we assume that we can get away with treating the phonons in the elastic medium classically.

[17] If two neighboring blocks have the same displacement, $u_{n+1} = u_n$, the connecting spring is not compressed nor stretched, and does not contribute to the potential energy. That is why only differences $u_{n+1} - u_n$ appear in the expression.

[18] *Advanced Quantum Mechanics*, for instance.

Lagrangian mechanics. If you are a fan of Lagrangian mechanics, you know that the equations of motion can be derived via the Lagrangian, and a variational principle called *Hamilton's principle*, or, alternatively, the *principle of least action [35, 40]*. The Lagrangian of the elastic medium is defined by

$$L = T - U = \int_a^b \mathcal{L} dx, \quad \text{where} \quad \mathcal{L} = \mu \left(\frac{\partial u}{\partial t}\right)^2 - \kappa \left(\frac{\partial u}{\partial x}\right)^2, \tag{5.48}$$

is called the *Lagrangian density*. Lagrange has shown that the *action integral* (or *action* for short)

$$S = \int_{t_a}^{t_b} L dt = \int_{t_a}^{t_b} \int_a^b \mathcal{L} dx dt, \tag{5.49}$$

is *variational*, i.e., the true physical function $u(x,t)$ corresponds to a stationary point (minimum, maximum, or saddle point) of the action S. The stationary point, and more importantly the function $u(x,t)$, can be found by solving the corresponding *Euler-Lagrange equations*

$$\frac{d}{dt}\left[\frac{\partial \mathcal{L}}{\partial\left(\frac{\partial u}{\partial t}\right)}\right] + \frac{d}{dx}\left[\frac{\partial \mathcal{L}}{\partial\left(\frac{\partial u}{\partial x}\right)}\right] = \frac{\partial \mathcal{L}}{\partial u}. \tag{5.50}$$

In this case $\partial\mathcal{L}/\partial u = 0$, and the derivatives of the left-hand side are easily obtained from Eq. 5.48. It yields

$$\kappa \frac{\partial^2 u}{\partial t^2} - \mu \frac{\partial^2 u}{\partial x^2} = 0, \tag{5.51}$$

which is the (classical) wave equation from Eq. 5.44.

Electron in one dimension. A single electron is described by the Schrödinger equation

$$i\hbar \frac{\partial}{\partial t}\psi = \left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right]\psi. \tag{5.52}$$

We assume that the electron wave $\psi(x,t)$ is defined on the same interval $a \le x \le b$ as the elastic medium, and worry about boundary conditions later on.

Lagrangian formulation. It is possible to obtain the Schrödinger equation as an equation of motion from a Lagrangian [40]. Just pretend that the electron wave $\psi(x,t)$ is a classical field. Actually, it is a bit more complicated than that. In classical physics all fields are real functions,[19] but quantum mechanical waves are intrinsically complex functions. One can write the electron wave as $\psi(x,t) = \psi_R(x,t) + i\psi_I(x,t)$, where ψ_R and ψ_I are real functions, which in the Lagrangian formalism one can pretend to be two independent classical fields. It turns out to be more convenient to use two linear combinations of those fields $\psi = \psi_R + i\psi_I$ and $\psi^* = \psi_R - i\psi_I$, if we just pretend that ψ and ψ^* are independent fields (just like ψ_R and ψ_I are independent fields). In terms of these fields we can write a Lagrangian

$$L = \int_a^b \mathcal{L} dx \quad \text{with} \quad \mathcal{L} = \frac{i\hbar}{2}\psi^*\frac{\partial \psi}{\partial t} - \frac{i\hbar}{2}\frac{\partial \psi^*}{\partial t}\psi - \frac{\hbar^2}{2m}\frac{\partial \psi^*}{\partial x}\frac{\partial \psi}{\partial x} - V\psi^*\psi, \tag{5.53}$$

[19] We often use complex numbers, especially if we are dealing with traveling waves, but that is just for mathematical convenience, to more easily keep track of phases, for instance. The measurable waves are always real.

the corresponding Lagrangian density. The corresponding Euler-Lagrange equations for the two fields are then

$$\frac{d}{dt}\left[\frac{\partial\mathcal{L}}{\partial\left(\frac{\partial\psi^*}{\partial t}\right)}\right]+\frac{d}{dx}\left[\frac{\partial\mathcal{L}}{\partial\left(\frac{\partial\psi^*}{\partial x}\right)}\right]=\frac{\partial\mathcal{L}}{\partial\psi^*}\text{ and }\frac{d}{dt}\left[\frac{\partial\mathcal{L}}{\partial\left(\frac{\partial\psi}{\partial t}\right)}\right]+\frac{d}{dx}\left[\frac{\partial\mathcal{L}}{\partial\left(\frac{\partial\psi}{\partial x}\right)}\right]=\frac{\partial\mathcal{L}}{\partial\psi},\tag{5.54}$$

which give

$$-\frac{i\hbar}{2}\frac{\partial\psi}{\partial t}-\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2}=\frac{i\hbar}{2}\frac{\partial\psi}{\partial t}-V\psi\text{ and }i\hbar\frac{\partial\psi^*}{\partial t}-\frac{\hbar^2}{2m}\frac{\partial^2\psi^*}{\partial x^2}=-\frac{i\hbar}{2}\frac{\partial\psi^*}{\partial t}-V\psi^*.\tag{5.55}$$

These are the Schrödinger equation and its complex conjugate, respectively.[20]

The main reason people like the Lagrangian formalism is because it also works if multiple fields are involved of a different type. For instance, the electron and the elastic medium can be combined to a single Lagrangian density

$$\mathcal{L}=\left[\frac{i\hbar}{2}\psi^*\frac{\partial\psi}{\partial t}-\frac{i\hbar}{2}\frac{\partial\psi^*}{\partial t}\psi-\frac{\hbar^2}{2m}\frac{\partial\psi^*}{\partial x}\frac{\partial\psi}{\partial x}-V\psi^*\psi\right]+\left[\frac{1}{2}\mu\left(\frac{\partial u}{\partial t}\right)^2-\frac{1}{2}\kappa\left(\frac{\partial u}{\partial x}\right)^2\right],\tag{5.56}$$

where the first term between [ ] describes the electron field, and the second term between [ ] describes the elastic medium field. The Schrödinger equation for the electron and the wave equation for the elastic medium can then both be derived from this one Lagrangian through the Euler-Lagrange equations for the corresponding fields, Eq. 5.54 and 5.50. Here, of course, that does not add anything new; it just reproduces the equations of motions we already had, Eqs. 5.54 and 5.51. The reason for this simplicity is that the two fields, electron and elastic medium, are totally independent. The first term of the Lagrangian density in Eq. 5.56 only contains electron variables, and the second term only contains elastic medium variables, and there are no terms in the Lagrangian that couple ψ with u.

5.3.1.2 Electron-phonon coupling*

Such coupling terms are exactly what emerges if one considers electron-phonon coupling. Consider modifying the potential energy of the elastic medium, Eq. 5.41, to

$$U=\frac{1}{2}\sum_{n=0}^{N-1}k\left(u_{n+1}-u_n\right)^2-\sum_{n=0}^{N-1}kb_n(t)\left(u_{n+1}-u_n\right).\tag{5.57}$$

We might rewrite this a little

$$U=\frac{1}{2}\sum_{n=0}^{N-1}k\left(u_{n+1}-u_n-b_n\right)^2-\frac{1}{2}\sum_{n=0}^{N-1}kb_n^2,$$

to stress that the $b_n(t)$ terms add a (possibly time-dependent) bias to the displacements. Taking the continuum limit of Eq. 5.57 as before, gives the expression

$$U=\int_a^b\left[\frac{1}{2}\kappa\left(\frac{\partial u}{\partial x}\right)^2-\kappa\beta\left(x,t\right)\frac{\partial u}{\partial x}\right]dx,\tag{5.58}$$

[20] Incidentally, this can also turned into a quantum field theory, by considering ψ and ψ^* as field operators. This field theory can be used to describe many-electron systems. As the the Schrödinger equation is already quantum, quantum field theory is also called *second quantization*. If you are interested in what this means, do a course on advanced quantum mechanics or quantum field theory.

with $\beta = b/\Delta x$ defining a density, which is called the *deformation potential*. Note that, despite its name, $\beta(x,t)$ is a dimensionless field.

We assume that the deformation potential is caused by the presence of an electron, and that its strength at (x,t) is proportional to the probability of an electron being at (x,t)

$$\beta(x,t) = \gamma |\psi(x,t)|^2, \quad \text{with } \gamma > 0 \text{ constant}, \tag{5.59}$$

see Fig. 5.3. Using the modified potential energy, Eq. 5.58, in the Lagrangian density, Eq. 5.56, the Euler-Lagrange equations, Eqs. 5.50 and 5.54, now lead to the two equations

$$i\hbar \frac{\partial}{\partial t}\psi(x,t) = \left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} - \kappa\gamma \frac{\partial}{\partial x}u(x,t)\right]\psi(x,t), \tag{5.60}$$

$$\mu \frac{\partial^2}{\partial t^2}u(x,t) = \kappa \frac{\partial}{\partial x}\left[\frac{\partial}{\partial x}u(x,t) - \gamma |\psi(x,t)|^2\right]. \tag{5.61}$$

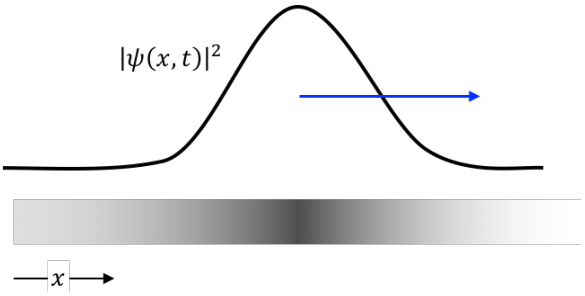


Figure 5.3: Artist impression of an electron interacting with a 1D elastic medium. Top: the probability distribution $|\psi(x,t)|^2$ of an electron causes, bottom: a deformation of the elastic medium. This deformation forms an attractive potential, that can capture the electron. A moving electron drags this deformation along with it.

These coupled (non-linear) partial differential equations are difficult to solve, and we are not going to try it. We assume that the medium is very light compared to the electron, so it can follow the motion of the electron adiabatically.[21] We then "solve" Eq. 5.61 in the limit μ very small, so

$$\mu \frac{\partial^2}{\partial t^2}u(x,t) \approx 0 \Rightarrow \frac{\partial}{\partial x}u(x,t) - \gamma |\psi(x,t)|^2 = const. \tag{5.62}$$

In quantum mechanics one usually assumes that $|\psi(x,t)|^2$ is localized, so $\psi(x \to \pm\infty, t) = 0$. In this particular case it makes sense to use the boundary conditions $\psi(a,t) = 0$ and $\psi(b,t) = 0$. If, in addition, we apply the boundary conditions $\partial u(a,t)/\partial x = 0$ and $\partial u(b,t)/\partial x = 0$, then $const = 0$, and we have from Eq. 5.62

$$\frac{\partial}{\partial x}u(x,t) = \gamma |\psi(x,t)|^2. \tag{5.63}$$

[21] Note that this is quite the opposite of the usual assumption, where the electron is presumed to be very light, and the background medium, the atoms in the crystal, are heavy. A light electron would just fly through fast, without affecting the medium. Only a heavy (and therefore, slow) electron has time to distort the medium. A heavy electron is created by having flat bands, and/or by an extraordinarily strong electron-phonon coupling.

5.3.2 The non-linear Schrödinger equation

Using Eq. 5.63 in Eq. 5.60 leads to

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = \left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} - \alpha\left|\psi(x,t)\right|^2\right]\psi(x,t) \quad \text{with} \quad \alpha = \kappa\gamma^2. \tag{5.64}$$

This equation is called the *non-linear Schrödinger equation.*[22] Note that the dimension of α is Jm (in SI units). As the dimension of $|\psi(x,t)|^2$ is m^{-1}, the dimension of $\alpha\,|\psi(x,t)|^2$ is J. Comparing Eq. 5.64 to the normal Schrödinger equation, Eq. 5.52, one observes that the term $-\alpha\,|\psi|^2$ plays the role of an attractive potential. By deforming the elastic medium, the electron digs a potential hole for itself. If the electron moves, it drags the lattice distortion, and the corresponding potential, along with it, see Fig. 5.3. The moving object, electron + distortion, is called a *polaron.*[23]

5.3.2.1 Self-trapped polarons

First we explore the stationary states of Eq. 5.64, in other words, a stationary polaron. We extract from the time-dependent Schrödinger equation a time-independent one, using the standard approach. Use the separation-of-variables ansatz $\psi(x,t) = \phi(x)f(t)$ in Eq. 5.64,

$$i\hbar\frac{1}{f(t)}\frac{\partial}{\partial t}f(t) = -\frac{\hbar^2}{2m}\frac{1}{\phi(x)}\frac{d^2}{dx^2}\phi(x) - \alpha\left|\phi(x)\right|^2\left|f(t)\right|^2,$$

and make the assumption that $|f(t)|^2 = 1$, then left and right of this expression have to be equal to a constant, which we call E. The solution to the time dependence is then

$$f(t) = \exp\left[-\frac{i}{\hbar}Et\right], \tag{5.65}$$

and the function $\phi(x)$ can be found from

$$\left[\frac{d^2}{dx^2} + \frac{2m\alpha}{\hbar^2}\left|\phi(x)\right|^2 + \frac{2mE}{\hbar^2}\right]\phi(x) = 0, \tag{5.66}$$

which is the *time-independent non-linear Schrödinger equation.*

Surprisingly, it has an analytical solution (well, at least one). Making use of

$$\begin{aligned}\frac{d^2}{dx^2}\left(\frac{1}{\cosh x}\right) = \frac{d}{dx}\left(-\frac{\sinh x}{\cosh^2 x}\right) &= 2\frac{\sinh^2 x}{\cosh^3 x} - \frac{\cosh x}{\cosh^2 x} \\ &= 2\frac{\cosh^2 x - 1}{\cosh^3 x} - \frac{\cosh^2 x}{\cosh^3 x} + = \frac{1}{\cosh x} - \frac{2}{\cosh^3 x},\end{aligned}$$

we have as solution

$$\phi(x) = \sqrt{\frac{a}{2}}\frac{1}{\cosh(ax)}, \quad a = \frac{m\alpha}{2\hbar^2}, \quad E = -\frac{m\alpha^2}{8\hbar^2}, \tag{5.67}$$

[22] There are of course many ways in which a Schrödinger equation can become non-linear, but this particular form has obtained some fame on its own, so in the literature it is frequently referred to as "the" non-linear Schrödinger equation.

[23] There are actually many types of polarons. The present model describes one of them, called a *small polaron.* Size is relative; a small polaron can still be many nearest neighbor atom-atom distances in size.

where the prefactor $\sqrt{a/2}$ ensures that the wave function is normalized, $\int_{-\infty}^{\infty} \phi^2(x)dx = 1$.[24] A free electron has energies $E > 0$, because it only has kinetic energy. The potential the electron creates an for itself by deforming the medium, is attractive, and the electron gets trapped in this potential in a bound state with $E < 0$. This state is called a *self-trapped polaron*. The wave function $\phi(x)$ has a typical pulse-like shape, as shown in Fig. 5.4.

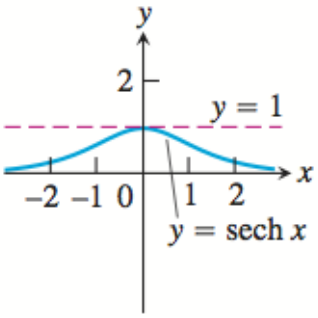


Figure 5.4: The function $y = \text{sech}x = 1/\cosh x$. Stretching and normalizing gives the function $\sqrt{a/2}/\cosh(ax)$, which has a typical width $\sim 2/a$ and a maximum $\sqrt{a/2}$ at $x = 0$.

5.3.2.2 Traveling polarons

The non-linear Schr"odinger equation, Eq. 5.64, not only has a stationary bound state, but it also has *traveling waves* as solutions. Inspired by the stationary polaron solution discussed above, traveling waves can be found by assuming that $\psi(x,t)$ is of the form

$$\psi(x,t) = \exp\left[i\left(kx - \omega t\right)\right]\phi\left(x - vt\right). \tag{5.68}$$

Using this ansatz in Eq. 5.64, we obtain

$$\hbar\omega\phi - i\hbar v\frac{d\phi}{d\eta} = \frac{\hbar^2k^2}{2m}\phi - i\frac{\hbar^2k}{m}\frac{d\phi}{d\eta} - \frac{\hbar^2}{2m}\frac{d^2\phi}{d\eta^2} - \alpha\left|\phi\right|^2\phi,$$

with $\eta = x - vt$. We can simplify this by assuming

$$v = \frac{\hbar k}{m},$$

and write

$$\left[\frac{d^2}{d\eta^2} + \frac{2m\alpha}{\hbar^2}\left|\phi\right|^2 + \frac{2mE'}{\hbar^2}\right]\phi = 0 \quad \text{with} \quad E' = \hbar\omega - \frac{\hbar^2k^2}{2m}.$$

But this is just like Eq. 5.66, so we already know one solution, Eq. 5.67. This means that Eq. 5.68 is a solution, in the form

$$\psi(x,t) = \sqrt{\frac{a}{2}}\frac{\exp\left[i\left(kx - \omega t\right)\right]}{\cosh\left[a\left(x - vt\right)\right]},$$
$$\text{with} \quad a = \frac{m\alpha}{2\hbar^2}, \quad \hbar k = mv, \quad \text{and} \quad \hbar\omega = \frac{\hbar^2k^2}{2m} - \frac{m\alpha^2}{8\hbar^2} \tag{5.69}$$

[24] The wave function defines a probability density $P(x) = \left|\phi(x)\right|^2$, and the polaron should be somewhere, $1 = \int_{-\infty}^{\infty} P(x)dx$.

Comparing this *traveling polaron*, Eq. 5.68, with the stationary one, Eq. 5.67, one can see that is obtained by letting the latter move with a constant group velocity v, and adding the kinetic energy $\hbar^2k^2/2m = \frac{1}{2}mv^2$ of this motion to the energy of the bound state to obtain the total energy $\hbar\omega$, Eq. 5.69.

5.4 Bipolarons and binding

If we bring two polarons in close proximity, they interact. If that interaction is attractive, it can result in a state where the two polarons are bonded to one another. Such a state is called a *bipolaron*. How such an attractive interaction is formed is explained in the following, and how it leads to a bound state is discussed in Sec. 5.4.1. The attractive interaction has to fight against the Coulomb repulsion between polarons, which occurs because polarons, like electrons, are charged particles. Ultimately, it is the balance between attraction and repulsion that determines whether a bound state can be formed, as discussed in Sec. 5.4.2.

Singlet two-particle state: bipolaron. As electrons are fermions, and polarons are electrons "dressed" with a lattice deformation, we can safely assume that polarons are also fermions. Imagine having two polarons at a large distance, such that one is described by a wave function $\phi_1(x,t)$, and the other by a wave function $\phi_2(x,t)$. If the two polarons are far apart and do not interact, the product $|\phi_1(x_1,t)\phi(x_2,t)|^2$ gives the probability density of finding one electron at (x_1,t) and the other at (x_2,t). However, as polarons are indistinguishable fermionic particles, just like electrons, then, according to the *Pauli principle*, their total wave function has to be anti-symmetric

$$\psi(x_1\sigma_1, x_2\sigma_1, t) = -\psi(x_2\sigma_2, x_1\sigma_1, t), \quad \text{where} \quad \sigma_{1,2} = \uparrow, \downarrow \tag{5.70}$$

are the spin-states of polaron 1 and 2. One such anti-symmetric wave function, obeying Eq. 5.70, and based upon the two functions ϕ_1 and ϕ_2, is the *singlet function*

$$\psi_S(x_1,x_2,t) = \frac{1}{\sqrt{2+2\,|S|^2}}\left\{\phi_1(x_1,t)\phi(x_2,t) + \phi_1(x_2,t)\phi_2(x_1,t)\right\}\frac{1}{\sqrt{2}}\left\{\uparrow_1\downarrow_2 - \downarrow_1\uparrow_2\right\}, \tag{5.71}$$

where

$$S = \int \phi_2^*(x,t)\phi_1(x,t)dx, \tag{5.72}$$

is the called the *overlap integral*. The prefactor on the right-hand side of Eq. 5.71 is chosen such that the function is normalized,

$$\int\int |\psi_S(x_1,x_2,t)|^2\, dx_1 dx_2 = 1. \tag{5.73}$$

The number (density) of electrons found at x is given by

$$\begin{aligned} n(x,t) &= \int |\psi_S(x,x_2,t)|^2\, dx_2 + \int |\psi_S(x_1,x,t)|^2\, dx_1 = 2\int |\psi_S(x,x',t)|^2\, dx' \\ &= \frac{1}{1+|S|^2}\left\{|\phi_1(x,t)|^2 + |\phi_2(x,t)|^2 + 2\mathrm{Re}\left[\phi_1^*(x,t)\phi_2(x,t)S\right]\right\}, \end{aligned} \tag{5.74}$$

where I have assumed that the individual wave functions are normalized $\int |\phi_1(x,t)|^2\, dx = \int |\phi_2(x,t)|^2\, dx = 1$.

Separate polarons. If the two polarons are well-separated, as in Fig. 5.5, then their wave functions do not overlap, $S = 0$. The number density is then simply the sum of the probability distributions of the two individual polarons

$$n_0(x,t) = |\phi_1(x,t)|^2 + |\phi_2(x,t)|^2 . \tag{5.75}$$

Moreover, for two non-overlapping wave functions, if x is in the range where $\phi_1(x,t) \neq 0$, then automatically $\phi_2(x,t) = 0$, and $n_0(x,t) = |\phi_1(x,t)|^2$, whereas if x is in the range where $\phi_2(x,t) \neq 0$, then $\phi_1(x,t) = 0$, and $n_0(x,t) = |\phi_2(x,t)|^2$. In other words, the description of the two polarons is not changed by the singlet function, and the wave functions ϕ_1 and ϕ_2 are those discussed In Secs. 5.3.2.1 and 5.3.2.2. In particular, the total energy E_0 is simply the sum of the energies of the two individual polarons. For stationary polarons, Eq. 5.67, this means

$$E_0 = 2 \times \left(-\frac{m\alpha^2}{8\hbar^2}, \right) = -\frac{m\alpha^2}{4\hbar^2}. \tag{5.76}$$

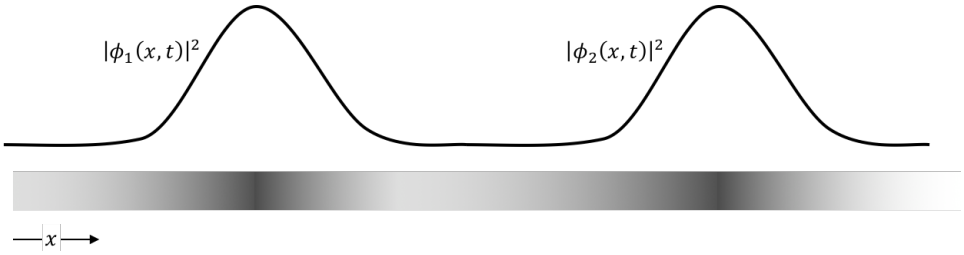


Figure 5.5: Two polarons far apart are described by wave functions ϕ_1 and ϕ_2 that have no overlap, $\int \phi_2^*(x)\phi_1(x)dx = 0$.

5.4.1 Stability of the bipolaron

If, however, the two polaron wave functions overlap, as in Fig. 5.6, then $n(x,t) > n_0(x,t)$ in the region where the functions overlap, compare Eqs. 5.74 and 5.75. The singles state, Eq. 5.71, can then describe a *bipolaron*.

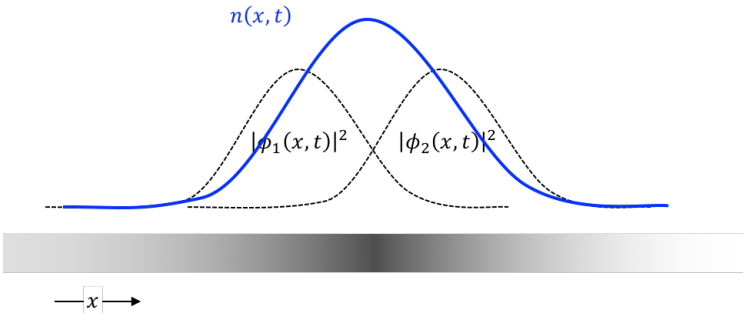


Figure 5.6: A bipolaron state is formed if $\int \phi_2^*(x)\phi_1(x)dx \neq 0$. Then we have $n(x,t) > n_0(x,t)$, see Eqs. 5.74 and 5.75.

We can study whether a bipolaron is stable with respect to separation into two individual polarons. In view of the discussion leading up to Eq. 5.63, we can generalize the latter

equation to

$$\frac{\partial}{\partial x}u(x,t) = \gamma n(x,t), \tag{5.77}$$

i.e. the displacement of the elastic medium at (x,t) is determined by the number density n of electrons present at (x,t). This then gives rise to an attractive potential $-\alpha n(x,t)$ in the Schrödinger equation, see Eq. 5.64. This attractive potential is larger than that of the individual polarons, as $n(x,t) > n_0(x,t)$, so one might expect that the two polarons attract one another, and form a bipolaron state.

Let us do the stationary state as in Sec. 5.3.2.1. The equivalent of Eq. 5.66 is two equations, one for each of the wave functions $\phi_1(x)$ and $\phi_2(x)$ in Eq. 5.71

$$\left[\frac{d^2}{dx^2} + \frac{2m\alpha}{\hbar^2}n(x) + \frac{2mE_1}{\hbar^2}\right]\phi_1(x) = 0; \quad \left[\frac{d^2}{dx^2} + \frac{2m\alpha}{\hbar^2}n(x) + \frac{2mE_2}{\hbar^2}\right]\phi_2(x) = 0. \tag{5.78}$$

The total energy of the singlet state, Eq. 5.71, is then given by

$$E_S = E_1 + E_2. \tag{5.79}$$

The larger the number density $n(x,t)$, the more attractive the potential becomes, and the lower the energy of the singlet state.

The two equations of Eq. 5.78 are difficult to solve for the situation shown in Fig. 5.6, but we can look at the special case $\phi_1(x) = \phi_2(x)$, i.e., when the two polaron wave functions coincide exactly. In that case the overlap $S = 1$, Eq. 5.72, and the number density, Eq. 5.74, becomes

$$n(x) = 2\left|\phi_1(x)\right|^2. \tag{5.80}$$

Eq. 5.78 is now easily solved along the same lines as in Sec. 5.3.2.1

$$\phi_1(x) = \sqrt{\frac{b}{2}}\frac{1}{\cosh(bx)}, \; b = \frac{m\alpha}{\hbar^2}, \; E_1 = -\frac{m\alpha^2}{2\hbar^2}. \tag{5.81}$$

The total energy for this case is given by

$$E_S = 2E_1 = -\frac{m\alpha^2}{\hbar^2}. \tag{5.82}$$

The energy of two separate polarons is given by Eq. 5.76, so the binding energy E_b of the two polarons in the bipolaron is

$$E_b = E_S - E_0 = -\frac{3m\alpha^2}{4\hbar^2}. \tag{5.83}$$

This is clearly negative, so the bipolaron is stable with respect to separation into two polarons.

In the popular literature, this story is often told as follows. Each polaron drags a lattice distortion along. If two polarons meet, then their lattice distortions start to overlap. One polaron is attracted to the lattice distortion of the other, and vice versa, which results in an effective attraction between the polarons, and hence the formation of a bipolaron.

5.4.2 Bipolarons with repulsion

One should argue that the discussion regarding the stability of bipolarons in Sec. 5.4.1 is too naive. We have only looked at the attractive interaction between two polarons caused by the lattice deformation. However, just as the electrons they originate from, polarons are charges particles (with charge $-e$), so they should also experience a Coulomb repulsion. The equations of Eq. 5.78 should be modified to

$$\left\{\frac{d^2}{dx^2}+\frac{2m}{\hbar^2}\left[\alpha n(x)-V_2(x)\right]+\frac{2mE_1}{\hbar^2}\right\}\phi_1(x)=0;$$
$$\left\{\frac{d^2}{dx^2}+\frac{2m}{\hbar^2}\left[\alpha n(x)-V_1(x)\right]+\frac{2mE_2}{\hbar^2}\right\}\phi_2(x)=0. \tag{5.84}$$

Here $V_1(x)=-e\Phi_1(x)$ is the Coulomb potential energy, originating from polaron 1,[25] with

$$\Phi_1(x)=\frac{1}{4\pi\epsilon}\int\frac{\rho_1(x')}{|x-x'|}dx', \quad \text{with} \quad \rho_1(x')=-e\left|\phi_1(x')\right|^2, \tag{5.85}$$

the charge density resulting from polaron 1, and ϵ the permittivity of the medium. Likewise $V_2(x)=-e\Phi_2(x)$ is the Coulomb potential energy originating from polaron 2, with

$$\Phi_2(x)=\frac{1}{4\pi\epsilon}\int\frac{\rho_2(x')}{|x-x'|}dx', \quad \text{with} \quad \rho_2(x')=-e\left|\phi_2(x')\right|^2, \tag{5.86}$$

the charge density resulting from polaron 2.

The energy of the bipolaron is now given by

$$E'_S=E_1+E_2-\int V_2(x)\rho_1(x)dx. \tag{5.87}$$

The last term on the right-hand side of Eq. 5.87 needs to be subtracted, because otherwise we would count the Coulomb interaction double.

Details: The total Coulomb energy between the two polarons is given by

$$E_C=\frac{1}{4\pi\epsilon}\int\int\frac{\rho_1(x)\rho_2(x')}{|x-x'|}dxdx'=\int\Phi_2(x)\rho_1(x)dx=\int\Phi_1(x)\rho_2(x)dx, \tag{5.88}$$

see Eqs. 5.85 and 5.86. But as

$$E_C=\int\Phi_2(x)\rho_1(x)dx=\int V_2(x)\left|\phi_1(x)\right|^2dx=\int\phi_1^*(x)V_2(x)\phi_1(x)dx=\left\langle\phi_1\left|V_2\right|\phi_1\right\rangle, \tag{5.89}$$

this means that

$$E_1=\left\langle\phi_1\left|-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}-\alpha n(x)\right|\phi_1\right\rangle+\left\langle\phi_1\left|V_2\right|\phi_1\right\rangle=\left\langle\phi_1\left|-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}-\alpha n(x)\right|\phi_1\right\rangle+E_C, \tag{5.90}$$

see Eq. 5.84 contains the full Coulomb energy. Likewise,

$$E_2=\left\langle\phi_2\left|-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}-\alpha n(x)\right|\phi_2\right\rangle+\left\langle\phi_2\left|V_1\right|\phi_2\right\rangle=\left\langle\phi_2\left|-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}-\alpha n(x)\right|\phi_2\right\rangle+E_C, \tag{5.91}$$

[25] $\Phi(x)$ is the potential as you know it from electrostatics. The "potential" $V(x)$ that enters the Schrödinger equation, is actually the potential energy $V(x)=q\Phi(x)=-e\Phi(x)$. Most people, including me, are a bit sloppy when using the term "potential".

contains the full Coulomb energy. The sum E_1+E_2 then contains $2E_C$, i.e., twice the total Coulomb energy. One can correct for this by subtracting the total Coulomb energy once,

$$E'_S = E_1 + E_2 - E_C, \tag{5.92}$$

as in Eq. 5.87.

The energy of two separate polarons at a large distance is still E_0, Eq. 5.76, as also the Coulomb repulsion goes to zero at a large distance. Whether the energy of the bipolaron is lower than that of two separate polarons

$$E'_S < E_0, \tag{5.93}$$

now totally depends on the balance between the attractive potential $-\alpha n(x)$, and the repulsive potentials $V_{1,2}(x)$. If the former wins, then the bipolaron is stable; if the latter wins, then the bipolaron splits into two separate polarons.

Short range potentials. In Eqs. 5.85 and 5.86 we have assumed that the Coulomb potentials are screened by the environment through a permittivity ϵ. Depending upon the environment, more drastic forms of screening are also possible. If, besides the two polarons, there are other mobile charges present, these can screen the Coulomb potential, and give potentials of the type

$$\Phi'_1(x) = \frac{1}{4\pi\epsilon}\int \frac{\rho_1(x')}{|x-x'|}\exp\left[-\frac{x'}{\lambda}\right]dx', \quad \text{with} \quad \rho_1(x') = -e\,|\phi_1(x')|^2\,, \tag{5.94}$$

and a similar expression for $\Phi'_2(x)$. Here λ is the screening length, see Sec. 4.1.1, Eq. 4.11. It is not common that the mobile charges are classical particles, as in Sec. 4.1.1, so λ is not likely to be a DeBye-Hückel screening length. However, even if the mobile charges are quantum particles, such as electrons, then still the potential of Eq. 5.94 is a reasonable expression, see Sec. 5.6, and the discussion around Eq. 5.141.

The effect of adding the $\exp[-x'/\lambda]$ term in Eq. 5.94 is to make the Coulomb interaction *short range*, as in the integral only points $-\lambda \lesssim x' \lesssim \lambda$ give an appreciable contribution. In other words, only the charge distribution $\rho_1(x')$ with $-\lambda \lesssim x' \lesssim \lambda$ generates a repulsive potential. Outside this range, the potential is screened. If the screening is very large, then λ is very small.

In calculations one often uses a model potential of the type

$$\Phi'_1(x) = \frac{U}{e^2}\int \rho_1(x')\delta\,(x-x')\,dx' = \frac{U}{e^2}\rho_1(x) = -\frac{U}{e}\,|\phi_1(x)|^2\,, \tag{5.95}$$

and a similar expression for $\Phi'_2(x)$. The idea is that the screening is so large (λ so small), and the screened Coulomb potential is so short range, that it effectively becomes a *contact potential*, signaled by the $\delta(x-x')$ term in Eq. 5.95. The parameter U is then a measure for the strength of the screened Coulomb potential.[26] Solving the bipolaron problem then

[26] In some literature, the parameter U/e^2 is called the Hubbard U, after the person who first introduced this type of interaction potential. My parameter U has the dimension Jm (in SI units). I have chosen it such that it has the same dimension as α in Eq. 5.84.

becomes somewhat simpler, as Eqs. 5.84 become

$$\left\{\frac{d^2}{dx^2}+\frac{2m}{\hbar^2}\left[\alpha n(x)-U\left|\phi_2(x)\right|^2\right]+\frac{2mE_1}{\hbar^2}\right\}\phi_1(x)=0;$$
$$\left\{\frac{d^2}{dx^2}+\frac{2m}{\hbar^2}\left[\alpha n(x)-U\left|\phi_1(x)\right|^2\right]+\frac{2mE_2}{\hbar^2}\right\}\phi_2(x)=0. \tag{5.96}$$

The expression for the total energy, Eq. 5.87, then becomes

$$E'_S=E_1+E_2-U\int\left|\phi_1(x)\right|^2\left|\phi_2(x)\right|^2dx. \tag{5.97}$$

5.4.3 Chemical bonds; density functional theory

The bipolaron discussion above has a surprising connection to something that has noting to do with bipolarons, i.e., the formation of chemical bonds in molecules. The bipolaron is a charged particle, with charge $-2e$. Suppose we change this into a neutral system by adding two charges $+e$ at fixed positions X_1 and X_2. These charges of course repel one another, but they give a Coulomb potential attraction to the two polarons,

$$W(x)=-\frac{e^2}{4\pi\epsilon}\left\{\frac{1}{|X_1-x|}+\frac{1}{|X_2-x|}\right\}. \tag{5.98}$$

The two coupled Schrödinger equations, Eq. 5.84, then become

$$\left\{\frac{d^2}{dx^2}+\frac{2m}{\hbar^2}\left[\alpha n^p(x)-V_2(x)-W(x)\right]+\frac{2mE_1}{\hbar^2}\right\}\phi_1(x)=0;$$
$$\left\{\frac{d^2}{dx^2}+\frac{2m}{\hbar^2}\left[\alpha n^p(x)-V_1(x)-W(x)\right]+\frac{2mE_2}{\hbar^2}\right\}\phi_2(x)=0. \tag{5.99}$$

Here I have generalized the αn term to αn^p, with p some (fixed) power.

These equations have acquired some fame that is completely unrelated to the physics of (bi)polarons. We omit the elastic medium and all the interactions with it, and focus on the electrons only. We describe the interactions between electrons by a theory called *density functional theory*[27] This results in an attractive potential $V_n(x)$ on each electron that depends only on the total electron density n; this potential is of a purely quantum-mechanical nature. The exact form of $V_n(x)$ is unknown, but in simplest approximation, it is given by[28]

$$V_n(x)=-\alpha n^{\frac{1}{3}}(x). \tag{5.100}$$

Meanwhile, the "classical" Coulomb repulsion potentials $V_1(x)$ and $V_2(x)$ are still active, of course. All of this then leads to equations like Eq. 5.100 with $p=\frac{1}{3}$.

The total energy of the system is found by modifying Eq. 5.87

$$E_S=E_1+E_2-\frac{1}{2}\frac{1}{4\pi\epsilon}\int\int\frac{\rho_1(x)\rho_2(x')}{|x-x'|}dxdx'+\frac{1}{2}\frac{1}{4\pi\epsilon}\frac{e^2}{|X_1-X_2|}, \tag{5.101}$$

[27] A theory devised by Kohn, Hohenberg, Sham, and Slater. Walter Kohn got the Nobel prize in chemistry for this in 1998. I will not explain that theory here. You either accept what I am stating here, or look up density functional theory (DFT) yourself [41, 42].

[28] The approximation is derived from the exchange energy in the homogeneous electron gas.

where the last term on the right-hand side describes the Coulomb repulsion between the fixed positive charges. What I have described just now is a simple model for the hydrogen molecule H_2. The hydrogen nuclei (^{1}H = proton) are the positive charges kept at fixed positions, and the singlet state $\psi_S(x_1, x_2, t)$, Eq. 5.71, is an approximation for the ground state wave function of the two electrons.

One can calculate E_S as a function of the distance $\Delta X = |X_1 - X_2|$ between the two hydrogen nuclei. This energy has a well-defined minimum, and the corresponding distance $\Delta X_{\rm eq}$, is called the *(equilibrium) bonding distance*. The potential of Eq. 5.100 is essential to develop this minimum. Therefore, the inevitable conclusion is that chemistry is essentially a quantum mechanical phenomenon.[29]

Of course, the model as presented here is one-dimensional, instead of three-dimensional, as it should be for a realistic hydrogen molecule. On the positive side, 1D calculations are easy, whereas 3D calculations can become time-consuming.

5.5 Solitons in optical fibers

Also in the field of optics several different types of solitons exist, depending on the type of media and the geometry used. I will discuss only one type, temporal (bright) solitons in optical fibers [43, 44]. Mathematically, these solitons have a large resemblance to the solid state polarons discussed above.

A main problem in transporting signals through optical fibers is dispersion, which is a consequence of EM waves with different frequencies moving at a different speeds. If signals are transported as a sequence of pulses, dispersion results in spreading out these pulses with time, as depicted in Fig. 5.7. This limits the pulse density and the practical range over which a signal can be transported, before the information is lost.

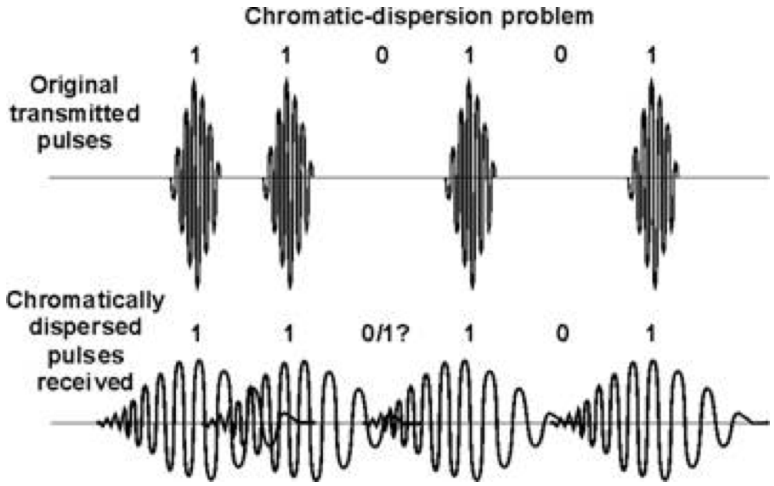


Figure 5.7: Pulses are superpositions of EM waves of different frequency. As the velocity of such waves is frequency-dependent, the pulses disperse over time. In the case shown here, the pulses move from left to right, and the low frequency components move faster than the high frequency ones. This describes the normal case when the refractive index increases with increasing frequency.

[29] The chemists will disagree with this, of course; some people are beyond recall.

Help is underway from non-linear materials. In such materials it is possible to transport pulses in special wave forms called *solitons*, which have the property that they do not spread out with time; they do not disperse, but retain a fixed shape. The refractive index of non-linear materials increases with the intensity of the EM wave. The wave length decreases with increasing refractive index, so this has the effect of contracting an EM pulse. The contraction opposes dispersion, and under the right conditions, prevent it.

5.5.1 From the Maxwell equations to the soliton equation

It is possible to derive a type of wave equation that describes the behavior of solitons in optical fibers, starting from the Maxwell equations. If you are not interested in this background, you can proceed to Sec. 5.5.2.

The Maxwell equations and the wave equation for a non-linear material. The Maxwell equations for EM fields in a medium without free charges and currents, read

$$\begin{aligned} \nabla \cdot \mathbf{D} = 0, \;\; \nabla \times \mathbf{E} = -\frac{\partial}{\partial t}\mathbf{B}, \\ \nabla \cdot \mathbf{B} = 0, \;\; \nabla \times \mathbf{B} = \mu_0 \frac{\partial}{\partial t}\mathbf{D}, \end{aligned} \tag{5.102}$$

where we have assumed that the permeability of the medium is that of vacuum, $\mu = \mu_0$.[30] The displacement field is as usual

$$\mathbf{D} = \epsilon_0 \mathbf{E} + \mathbf{P}, \tag{5.103}$$

with $\mathbf{P}$ the polarization of the medium. We assume that the medium is non-linear, and write

$$\mathbf{P} = \epsilon_0 \chi \mathbf{E} + \epsilon_0 \eta \left|\mathbf{E}\right|^2 \mathbf{E}, \tag{5.104}$$

with χ and η materials' parameters fixed by the medium, where we assume that both parameters are real (we neglect absorption), and for normal materials, both $\chi > 0$ and $\eta > 0$.

Details: Think of Eq. 5.104 as the first terms of a Taylor expansion. We expand $\mathbf{P}(\mathbf{E})$ around $\mathbf{E} = (E_x, E_y, E_z) = (0, 0, 0)$. For isotropic materials, $\mathbf{P}$ is inverted if $\mathbf{E}$ is inverted, i.e., if $\mathbf{E} \to -\mathbf{E}$, then $\mathbf{P} \to -\mathbf{P}$.[31] This means that in a Taylor expansion, only terms can appear that are odd in the components of $\mathbf{E}$. Even terms, such as $E_i E_j$, $i.j = x, y, z$, do not change sign upon inversion: $E_i \to -E_i$ and $E_j \to -E_j$, and are thus excluded. Normally we stop the Taylor expansion after the first non-zero term, which is the linear $\chi\mathbf{E}$ term. Now we admit the next non-zero higher order term in the expansion, which is a cubic term, in order to explore some of the physical implications of non-linearity. If furthermore we demand that the material is isotropic, meaning that $\mathbf{P}$ is in the same direction as $\mathbf{E}$, and its size cannot depend on the direction of $\mathbf{E}$, then the linear and cubic terms must have the form as in Eq. 5.104.

Combining Eqs. 5.103 and 5.104, we write

$$\mathbf{D} = \epsilon_0 \left(\epsilon_r + \eta \left|\mathbf{E}\right|^2\right) \mathbf{E} \;\; \text{with} \;\; \epsilon_r = 1 + \chi. \tag{5.105}$$

[30] An excellent approximation for most (diamagnetic and paramagnetic) media, where μ differs from μ_0 only in the ppm range.

[31] Technically, "isotropic" is too strong a demand. It is sufficient that the material is centrosymmetric. Isotropic materials always are, but many crystals are also centrosymmetric. Glasses and liquids are most often isotropic.

Via the usual trick, $\nabla \times (\nabla \times \mathbf{E})$, we can derive from the Maxwell equations a wave equation

$$\frac{\epsilon_r}{c^2}\frac{\partial^2}{\partial t^2}\mathbf{E} - \nabla\left(\nabla \cdot \mathbf{E}\right) - \nabla^2\mathbf{E} = -\frac{\eta}{c^2}\frac{\partial^2}{\partial t^2}\left(|\mathbf{E}|^2\,\mathbf{E}\right), \tag{5.106}$$

where we have made the usual substitution $\epsilon_0\mu_0 = 1/c^2$. The non-linear term on the right-hand side of Eq. 5.106 makes this a complicated equation that cannot be solved in general (at least, not by me).

The wave equation for an optical fiber. For the special case of EM waves propagating in optical fibers we can make some special assumptions. Arrange the fiber in the x-direction, and assume that the EM wave travels along the fiber. Consider a TE mode, meaning that the electric field is perpendicular to the x-direction, and has the general form[32]

$$\mathbf{E} = \mathbf{E}^{\perp}(y,z)E(x,t) \quad \text{with} \quad \mathbf{E}^{\perp}\cdot\hat{\mathbf{x}} = 0. \tag{5.107}$$

$\mathbf{E}^{\perp}$ describes how the field varies over the cross section of the fiber. Just as for the wave guides you know from your electrodynamics course, it obeys a Helmholtz equation

$$\left(\frac{\partial^2}{\partial y} + \frac{\partial^2}{\partial z^2}\right)\mathbf{E}^{\perp} = -\kappa^2\mathbf{E}^{\perp}, \tag{5.108}$$

and a set of boundary conditions on the wall of the fiber. The parameter $\kappa > 0$ depends on the specific mode, on the cross section geometry of the fiber and on the materials the fiber and the cladding are made of. We do not need to specify $\mathbf{E}^{\perp}$ in detail for the physics that follows. Just assume that we pick out one specific TE mode with a fixed κ.

Using Eqs. 5.107 and 5.108 in 5.106 gives

$$\frac{\epsilon_r}{c^2}\frac{\partial^2}{\partial t^2}E - \frac{\partial^2}{\partial x^2}E + \kappa^2 E = -\frac{\zeta}{c^2}\frac{\partial^2}{\partial t^2}\left(|E|^2\,E\right), \tag{5.109}$$

where

$$\zeta = \frac{\eta}{A_{\text{cs}}}\int_{\text{cs}}\left|\mathbf{E}^{\perp}\right|^2 dydz, \tag{5.110}$$

is a non-linear parameter averaged over the cross-section of the fiber, with A_{cs} the area of the cross section. It depends on the mode we have chosen, of course.

So far we have actually been cheating by treating ϵ_r as a constant, whereas we know that it is frequency-dependent. This can be remedied by Fourier transforming with respect to position and time

$$E(x,t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\tilde{E}(k',\omega')\exp\left[i\left(k'x - \omega' t\right)\right]dk'd\omega'. \tag{5.111}$$

Using this expression for the right-hand side of Eq. 5.109 (we consider the left-hand side later), and pulling the ϵ_r inside the integral one obtains

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\left[-\frac{\epsilon_r(\omega')\omega'^2}{c^2} + k'^2 + \kappa^2\right]\tilde{E}(k',\omega')\exp\left[i\left(k'x - \omega' t\right)\right]dk'd\omega' = -\frac{\zeta}{c^2}\frac{\partial^2}{\partial t^2}\left(|E|^2\,E\right).$$

[32] TM modes can be treated similarly, but starting from a wave equation for the magnetic field.

We assume that $\epsilon_r(\omega)$ is real, i.e., we neglect absorption.[33] The dispersion relation for an EM mode in the fiber is then defined by

$$k(\omega) = \frac{\omega}{c}\sqrt{\epsilon_r(\omega) - \frac{c^2\kappa^2}{\omega^2}}, \tag{5.112}$$

where $k(\omega)$ is a real functions, and the wave equation becomes

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\left[k'^2 - k^2(\omega)\right]\tilde{E}(k',\omega')\exp\left[i\left(k'x-\omega' t\right)\right]dk'd\omega' = -\frac{\zeta}{c^2}\frac{\partial^2}{\partial t^2}\left(|E|^2 E\right). \tag{5.113}$$

The soliton equation for an optical fiber. This formidable looking equation can be solved in certain limits. We try a solution of the form

$$E(x,t) = \exp\left[i\left(k_0 x - \omega_0 t\right)\right]F(x,t) \quad \text{with} \quad k_0 = k(\omega_0), \tag{5.114}$$

and assume that $F(x,t)$ is a function that varies *very slowly* in x and t, as compared to the typical length scale $\lambda_0 = 2\pi/k_0$ and time scale $t_0 = 1/\omega_0$ over which the prefactor varies, see Fig. 5.8. This can be formalized by stating

$$\frac{1}{\omega_0^n}\frac{\partial^n F}{\partial t^n} \ll 1, \quad n = 1,2,..., \quad \text{and} \quad \frac{1}{k_0^n}\frac{\partial^n F}{\partial x^n} \ll 1, \quad n = 1,2,... \tag{5.115}$$

The Fourier transform of Eq. 5.111 can now be written as

$$E(x,t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\tilde{F}(k'-k_0,\omega'-\omega_0)\exp\left[i\left(k'x-\omega' t\right)\right]dk'd\omega',$$

where $\tilde{F}(k'-k_0,\omega-\omega_0)$ is a function that has a sharp peak centered around $\omega' = \omega_0$ and $k' = k_0$, , see Fig. 5.8.[34]

The Fourier integral of Eq. 5.113 becomes

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\left[k'^2 - k^2(\omega')\right]\tilde{F}(k'-k_0,\omega'-\omega_0)\exp\left[i\left(k'x-\omega' t\right)\right]dk'd\omega'. \tag{5.116}$$

As $\tilde{F}$ is a sharply peaked function, non-zero contributions to the Fourier integral only come from small intervals around the peak center, $[k_0 - \Delta k'; k_0 + \Delta k']$ and $[\omega_0 - \Delta\omega'; \omega_0 + \Delta\omega']$. In this interval we can approximate

$$k'^2 - k^2(\omega') = \left[k' + k(\omega')\right]\left[k' - k(\omega')\right] \approx 2k_0\left[k' - k(\omega')\right], \tag{5.117}$$

and we Taylor expand (yes, there it is again!)

$$2k_0\left[k' - k(\omega')\right] \approx 2k_0\left\{k' - \left[k_0 + \beta_1\left(\omega'-\omega_0\right) + \frac{1}{2}\beta_2\left(\omega'-\omega_0\right)^2\right]\right\} \tag{5.118}$$

[33] This is not too bad, provided we have a good quality fiber, and in frequency we stay away from optical resonances of the material the fiber is made of.

[34] Think about what a Fourier transform does. Fourier transforming a function $F(x,t)$ that varies very slowly in x and t, it becomes a sharply peaked function $\tilde{F}(k,\omega)$. The exponential prefactor in Eq. 5.114 shifts this peak to (k_0,ω_0).

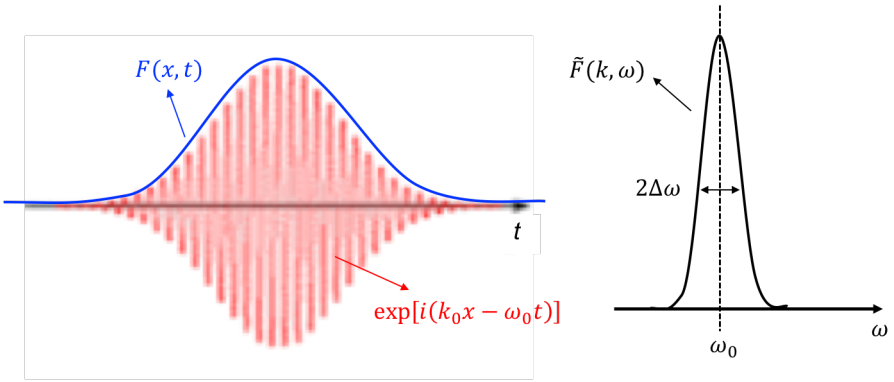


Figure 5.8: Left: the electric field $E(x,t) = \exp\left[i\left(k_0 x - \omega_0 t\right)\right] F(x,t)$ of a pulse with a rapidly varying phase factor $\exp\left[i\left(k_0 x - \omega_0 t\right)\right]$ and a slowly varying envelope function $F(x,t)$. Right: the Fourier transform of the electric field is a function $\tilde{F}(k,\omega)$ that is sharply peaked around $\omega = \omega_0$ (and, not shown, around $k = k_0$).

with

$$\beta_1 = \left.\frac{dk}{d\omega'}\right|_{\omega_0} \quad \text{and} \quad \beta_2 = \left.\frac{d^2k}{d\omega'^2}\right|_{\omega_0}, \tag{5.119}$$

where we only retain up to and including the quadratic factors. We use Eqs. 5.117 and 5.118 in 5.116, and substitute the integral variables by $k'' = k' - k_0$ and $\omega'' = \omega' - \omega_0$ to get for the Fourier integral

$$2k_0 \exp\left[i\left(k_0 x - \omega_0 t\right)\right] \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \left[k'' - \beta_1\omega'' - \frac{1}{2}\beta_2\omega''^2\right] \tilde{F}(k'',\omega'') \exp\left[i\left(k''x - \omega''t\right)\right] dk''d\omega''. \tag{5.120}$$

The integral can be done formally, and we get

$$2k_0 \exp\left[i\left(k_0 x - \omega_0 t\right)\right] \left[-i\frac{\partial}{\partial x} - i\beta_1\frac{\partial}{\partial t} + \frac{1}{2}\beta_2\frac{\partial^2}{\partial t^2}\right] F(x,t). \tag{5.121}$$

Now to the left-hand side of Eq. 5.113

$$\begin{aligned}\frac{\partial^2}{\partial t^2}\left(|E|^2 E\right) &= |F|^2 \frac{\partial^2}{\partial t^2} E, \\ &= |F|^2 \exp\left[i\left(k_0 x - \omega_0 t\right)\right] \left\{-\omega_0^2 F - 2i\omega_0 \frac{\partial F}{\partial t} + \frac{\partial^2 F}{\partial t^2}\right\} \\ &\approx -\omega_0^2 |F|^2 F \exp\left[i\left(k_0 x - \omega_0 t\right)\right], \end{aligned} \tag{5.122}$$

because of Eq. 5.115.

5.5.2 Optical solitons from the non-linear Schrödinger equation

Combining Eqs. 5.121 and 5.122 in 5.113 then gives the approximate wave equation

$$\left[-i\frac{\partial}{\partial x} - i\beta_1\frac{\partial}{\partial t} + \frac{1}{2}\beta_2\frac{\partial^2}{\partial t^2}\right] F = \frac{\zeta\omega_0^2}{2k_0 c^2} |F|^2 F, \tag{5.123}$$

which should be valid for wave functions of the type defined by Eqs. 5.114 and 5.115. It is called the *soliton equation*.

To get rid of the $\partial/\partial t$ derivative, we make a coordinate transformation

$$(x,t) \to (\xi,\tau) = (x, t - \beta_1 x). \tag{5.124}$$

This then leads to a soliton equation of the form

$$i\frac{\partial}{\partial x}F(x,\tau) = \left[\frac{1}{2}\beta_2\frac{\partial^2}{\partial\tau^2} - \alpha\left|F\right|^2\right]F(x,\tau) \quad \text{with} \quad \alpha = \frac{\zeta\omega_0^2}{2k_0c^2}. \tag{5.125}$$

This soliton equation looks remarkably similar to the non-linear Schrödinger equation discussed in Sec. 5.3.2, see Eq. 5.64, but withe the roles of the variables, time t or τ, and position x, interchanged, and with a substitution of constants

$$\begin{aligned} &x \to \tau, \quad ,t \to x; \\ &\hbar \to 1, \quad -\frac{1}{m} \to \beta_2, \quad \alpha \to \frac{\zeta\omega_0^2}{2k_0c^2}. \end{aligned} \tag{5.126}$$

Note that, because $m > 0$, this analogy only works if

$$\beta_2 < 0. \tag{5.127}$$

"Stationary" solutions We can follow the same solution technique for solving Eq. 5.125 as in Sec. 5.3.2. A "stationary" solution can be obtained from the separation-of-variables ansatz $F(x,t) = f(x)g(\tau)$, and $\left|f(x)\right|^2 = 1$, giving

$$f(x) = \exp\left[-iKx\right], \tag{5.128}$$

with K a constant, and $g(\tau)$ following from the equation

$$\left[\frac{\partial^2}{\partial\tau^2} - \frac{2\alpha}{\beta_2}\left|F\right|^2 - \frac{2K}{\beta_2}\right]g(\tau) = 0. \tag{5.129}$$

Comparison to Eqs. 5.66 and 5.67 shows that we can try a solution of type[35]

$$g(\tau) = \frac{b}{\cosh(a\tau)}. \tag{5.130}$$

Note that there is a *difference* with the polaron of Sec. 5.3.2. For an optical soliton there is no need to have a normalized function. It does not represent a probability distribution, so it can have any amplitude b we like, as an electric field can have any amplitude we like. Trying Eq. 5.130 in 5.129 shows that it is a solution, provided

$$a = b\sqrt{-\frac{\alpha}{\beta_2}}; \quad K = -\frac{\alpha}{2}b^2. \tag{5.131}$$

Note that, as $\alpha > 0$, and $\beta_2 < 0$ (Eq. 5.127), a is a real number. The width of the pulse $2/a$ is proportional to the inverse amplitude $1/b$, see Fig. 5.4, so the *stronger* the pulse (larger b), the *narrower* the soliton peak.

[35] In case you are worried about the $-$ signs, keep Eq. 5.127 in mind.

Traveling solutions The "stationary" solution we found in Eq. 5.130 is not stationary in the physical sense, because we have applied the coordinate transformation of Eq. 5.124. Transforming back to the physical coordinates (x,t), and using Eq. 5.114, gives us the electric field

$$E(x,t) = \exp\left[i\left(k_0 x - \omega_0 t\right)\right] f(x) g(\tau) = b \frac{\exp\left\{i\left[\left(k_0 - K\right) x - \omega_0 t\right]\right\}}{\cosh\left[a\left(t - \beta_1 x\right)\right]}, \tag{5.132}$$

with the constants a and K given by Eq. 5.131. We can get rid of these constants by choosing a convenient scaling for the amplitude

$$b = A\omega_0 \sqrt{-\frac{\beta_2}{\alpha}} = A\sqrt{\frac{-2k_0 c^2 \beta_2}{\zeta}}, \tag{5.133}$$

where A now denotes the amplitude; note that A is a dimensionless number. The constants a and K are then given by

$$a = A\omega_0 \quad \text{and} \quad K = \frac{1}{2} A^2 \beta_2 \omega_0^2, \tag{5.134}$$

and using Eqs. 5.133 and 5.134 in 5.132 gives for the electric field

$$E(x,t) = A\sqrt{\frac{-2k_0 c^2 \beta_2}{\zeta}} \frac{\exp\left\{i\left[\left(k_0 - \frac{1}{2}A^2\beta_2\omega_0^2\right) x - \omega_0 t\right]\right\}}{\cosh\left[A\omega_0\left(t - \beta_1 x\right)\right]}. \tag{5.135}$$

Let us try to put a little more physical interpretation into this. From Eq. 5.119 we find that

$$\beta_1 = \frac{1}{v_g} \quad \text{and} \quad \beta_2 = -\frac{1}{v_g^2} \left.\frac{dv_g}{d\omega}\right|_{\omega_0}, \tag{5.136}$$

where

$$v_g = \left.\frac{d\omega}{dk}\right|_{\omega_0} > 0, \tag{5.137}$$

is the *group velocity* of the waves in the fiber. Obviously $v_g > 0$ is the normal situation. The parameter β_2 is called the *group velocity dispersion* or GVD[36] in the literature. It is a key quantity in describing the dispersion of an optical pulse in a medium.

For a normal medium, glass or air, for instance, the group velocity v_g of optical waves *decreases* with *increasing* frequency ω. That is, basically, because the index of refraction increases with increasing frequency (from red to blue light, look at the rainbow), and light speed decreases with increasing refractive index. In that case $dv_g/d\omega < 0$, and thus $\beta_2 > 0$. This is called *normal dispersion.* For solitons of the type discussed to develop, we need $\beta_2 < 0$, see the discussion at the beginning of Sec. 5.5.2. This is called *anomalous dispersion.* In the optical fiber geometry one actually often has the situation that v_g *increases* with *increasing* frequency ω. Like in the wave guides you have had in your electrodynamics course, this behavior of $v_g(\omega)$ is a consequence of the geometry of the wave guide, and the boundary conditions introduced on the walls of the wave guide, rather than of the material the wave guide interior is made of. We then have $dv_g/d\omega > 0$, and thus $\beta_2 < 0$, which is just what we need to develop our solitons.

[36] No offense intended; the English are ignorant about Dutch acronyms.

The electric field of Eq. 5.135 describes a pulse whose shape is determined by the $1/\cosh$ function, see Fig. 5.4. Its motion is described by the argument

$$t - \beta_1 x = -\frac{x - v_g t}{v_g}. \tag{5.138}$$

It shows that the pulse moves with speed v_g.[37] The phase of the wave is described by the exp function. Compared to an ordinary plane wave, it has an extra phase factor depending on the amplitude, and the GVD

$$\exp\left[-\frac{i}{2}A^2\beta_2\omega_0^2 x\right]. \tag{5.139}$$

5.6 Cooper pairs

The electron-electron interaction. In Secs. 5.4 and 5.4.3 we looked at several two-particle problems. In the current section we will have a look at a very famous two-particle problem, which has been at the root of understanding the phenomenon of superconductivity, the *Cooper pair*, named after the person who first discovered its significance. A Cooper pair is a *bound state* formed by two electrons, typically a singlet state, see Eq. 5.71. One needs an *attractive potential* to create a bound state, so first we have to discuss how one can get attraction between electrons. Unlike our discussion in Secs. 5.4 and 5.4.3, we immediately go to three dimensions.[38]

Like in Sec. 5.4 we start with two electrons. One expects that the interaction $V(\mathbf{r})$ between two electrons is the *strongly repulsive* Coulomb potential (energy)

$$V_{\text{Coul}}(r) = \frac{e^2}{4\pi\epsilon r}. \tag{5.140}$$

This is certainly the case for two electrons in free space. A potential that is purely repulsive, does not have bound states,[39] and only results in electrons-electron scattering, i.e., electrons bouncing off one another. However, imagine that the two electrons reside in a medium that contains lots of mobile charges, as in Sec. 4.1. The Coulomb potential is then screened into

$$V_{\text{screen}}(r) = \frac{e^2}{4\pi\epsilon r}\exp\left[-\frac{r}{\lambda}\right], \tag{5.141}$$

which implies that its effective range is limited to λ, and the potential goes to zero fast for $r > \lambda$, see Fig. 4.1. Metals are media that contain plenty of mobile charges. For most ordinary metals the screening length λ is of the order of the distance between nearest neighbor atoms in the lattice. As the potential of Eq. 5.141 goes to zero fast, it means that for distances $r \gg \lambda$, two electrons interact as neutral particles!

A word of caution is in order here; λ is not a Debye screening length as in Sec. 4.1, Eq. 4.11. The mobile charges in a metal are electrons, for which one cannot use Boltzmann statistics. Using the proper Fermi-Dirac statistics for electrons applied to a free electron

[37] Don't worry about the overall $-$ sign of the argument, as $\cosh(-y) = \cosh(y)$. The overall direction of the wave should be fixed by the experimentalist.

[38] A one-dimensional approach would be too crude to give the right physics here.

[39] There is no potential "well" to trap the particles.

gas, Thomas and Fermi obtained an expression like Eq. 5.141, named the *Thomas-Fermi potential*. The Thomas-Fermi screening parameter is given by $\lambda = \sqrt{\pi a^*/(4k_F)}$, with $a^* = \hbar^2/(m^*e^2)$ the effective Bohr radius (m^* is the effective electron mass), and k_F the Fermi wave vector. At ordinary metallic electron densities, $k_F = O(1/a^*)$, so $\lambda = O(a^*)$. If you are really cautious, the Thomas-Fermi model of screening, which leads to Eq. 5.141, is relatively crude. A better description of screening in a metal gives a much more complicated potential, which is frequency-dependent, for instance. For our story here that does not matter. We are not interested in the details of a repulsive potential, as such a potential only gives scattering, not bound states. The only feature of the repulsive potential that is relevant here, is that it has a short range, as a result of screening.

There is another interaction potential for two electrons in a solid medium, which can lead to an attractive interaction. We encountered it in Sec. 5.4, where we discussed bipolarons. Electrons can exchange phonons, and this exchange results in an interaction between the electrons. In the popular press this story is told like my story of bipolarons; an electron at position $\mathbf{r}_1$ pulls the nuclei of the atoms close to it towards $\mathbf{r}_1$. After all, the electron is negatively charged, whereas the nuclei are positively charged. Another electron at position $\mathbf{r}_2$ sees these displaced nuclei and wants to follow them (for the same reason; the electron and the nuclei are oppositely charged). In other words, there is an net attractive interaction $V(\mathbf{r}_1-\mathbf{r}_2)$ between the two electrons. The popular story is OK, as long as you see it as a story, not as a theory. It works for bipolarons, but that is a special case, where the electrons are "heavy" particles and the nuclei are "light" particles, and one may assume that light particles adiabatically follow the heavy ones, like the planets follows the sun in its motion through the galaxy, see the discussion in Sec. 5.3.1. In a normal metal, however, the situation is reversed; the electrons are the light particles, whereas the nuclei are the heavy particles. The polaron theory does not hold in that case.

The proper starting point is to treat the electron-phonon interaction as perturbation, and apply (quantum) perturbation theory to derive the potential $V(\mathbf{r}_1 - \mathbf{r}_2)$ between two electrons, resulting from this interaction. The coupling between an electron and the phonons results in a net interaction with a second electron.[40] The interaction can be attractive, as well as repulsive, depending on the details of this electron-phonon coupling, but we will not try to model it in detail here. We just assume that such a $V(\mathbf{r}_1 - \mathbf{r}_2)$ exists, that it has an attractive part, and that it is relatively long range, i.e., that it extends far beyond the screening length λ. The latter is not such a big assumption. The potential basically results from the elasticity of the lattice, and elastic forces can be long range.

In one dimension, an attractive potential between two electrons always gives a bound state. In three dimensions, it does not have to! In three dimensions, a potential must have a

[40] The story as the high-energy physicists tell it, is closer to reality. Electrons are fermions, and phonons are bosons. The fermions are the primary particles here (carrying mass and charge, for instance), the bosons are temporary particles used to transmit forces. A moving fermion sends out a boson, which is caught by a second moving fermion. This exchange of a boson (with momentum and energy) gives a net interaction between the two fermions. Quantum electrodynamics works this way, where the fermions are charged particles, such as electrons, and the bosons are photons. This also indicates that the interaction can be repulsive, like the electrostatics between particles with the same charge, or attractive, like the magnetostatics with two parallel currents (particles flying in the same general direction), depending on the nature of the boson that is exchanged. The net range of the interaction depends on the mass of the boson that is exchanged. A massive boson leads to a short-range interaction, for instance of the type of Eq. 5.141. High-energy physicists call this the *Yukawa potential*, where $\lambda = \hbar/(Mc)$, with M the mass of the boson. For a massless boson, $M = 0$, such as a photon, we have $\lambda = \infty$, and the Yukawa potential becomes the ordinary long-range Coulomb potential, Eq. 5.140.

minimum strength to create a bound state. If the potential is too weak, it cannot bind the two particles together, even if the potential is attractive. We will check this statement as part of this computational project. Unfortunately, the reasonable potentials one can come up with, resulting from electron-phonon coupling, are too weak to create a bound state. Nevertheless, Cooper pairs exist, so something special must be going on in order to create those bound states. This is another element of this computational project.

Superconductivity. If two electrons are bound in a Cooper pair, they typically form a singlet state, see Eq. 5.71. A singlet has spin $S = 0$, so the Cooper pair is a (compound) bosonic particle. Unlike fermions, (identical) bosons can all be in the same quantum state. In fact, at very low temperature, all (identical) bosons prefer to be in the same quantum state (the energetically lowest one), as the spectacular phenomenon of Bose-Einstein condensation in dilute gases demonstrates. If that quantum state carries a net momentum, it carries a current, and as all bosons are in this state, that current will be sizable. One can destroy that current by breaking up the bosons (the Cooper pairs), which requires putting in the energy to break up the bound state. As long as there is nothing that can deliver that energy, i.e., if we remain at a low temperature and a low enough current, the current goes through unhindered without resistance. This is called a *supercurrent* and the phenomenon is called *superconductivity*.

Bardeen, Cooper, and Schrieffer devised a microscopic theory for superconductivity, named after them the BCS theory [39, 45].[41] Here we are going to look at the first ingredient of that theory, the formation of a Cooper pair.

5.6.1 The two-particle Schrödinger equation

The (time-independent) Schrödinger equation for two particles reads

$$\left\{-\frac{\hbar^2}{2m_1}\nabla_1^2 - \frac{\hbar^2}{2m_2}\nabla_2^2 + V(\mathbf{r}_1,\mathbf{r}_2)\right\}\Psi(\mathbf{r}_1,\mathbf{r}_2) = E\Psi(\mathbf{r}_1,\mathbf{r}_2), \tag{5.142}$$

where $\mathbf{r}_1 = (x_1, y_1, z_1)$, $\hat{\mathbf{p}}_1 = \frac{\hbar}{i}\nabla_1 = \frac{\hbar}{i}\left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial y_1}, \frac{\partial}{\partial z_1}\right)$, m_1 are the position, momentum operator, and mass of the first particle, and $\mathbf{r}_2 = (x_2, y_2, z_2)$, $\hat{\mathbf{p}}_2 = \frac{\hbar}{i}\nabla_2 = \frac{\hbar}{i}\left(\frac{\partial}{\partial x_2}, \frac{\partial}{\partial y_2}, \frac{\partial}{\partial z_2}\right)$, m_2 the position, momentum operator, and mass of the second particle. In case the interaction potential $V(\mathbf{r}_1,\mathbf{r}_2) = V(\mathbf{r}_1 - \mathbf{r}_2)$ only depends on the relative position of the two particles, it pays off to make a coordinate transformation to that relative position and the center of mass position

$$\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2; \quad \mathrm{R} = \frac{m_1\mathbf{r}_1 + m_2\mathbf{r}_2}{m_1 + m_2}, \tag{5.143}$$

to get

$$\left\{-\frac{\hbar^2}{2M}\nabla_R^2 - \frac{\hbar^2}{2\mu}\nabla_r^2 + V(\mathbf{r})\right\}\Psi(\mathbf{R},\mathbf{r}) = E\Psi(\mathbf{R},\mathbf{r}), \tag{5.144}$$

[41] John Bardeen is the only physicist who received the Nobel prize for physics *twice*; the first one, together with Shockley and Brattain, for the invention of the transistor, and the second one, with Cooper and Schrieffer, for the BCS theory of superconductivity. The transistor is a landmark in applied physics, and the biggest step forward in civilization since stone age (wo)man learned how to use fire. BCS theory is a landmark in theoretical physics (with massive implications for experimental physics). Oddly enough, Bardeen does not come to many people's mind when they have to name famous physicists. That may illustrate how much physics has gone the "dark force (mass, energy?)" way.

where $M = m_1 + m_2$ is the *total mass* of the two particles, $\mu = m_1 m_2/(m_1 + m_2)$ is called the *reduced mass*, $\hat{\mathbf{P}} = \frac{\hbar}{i}\nabla_R = \frac{\hbar}{i}\left(\frac{\partial}{\partial R_x}, \frac{\partial}{\partial R_y}, \frac{\partial}{\partial R_z}\right)$ is the total momentum operator of the particle pair, and $\hat{\mathbf{p}} = \frac{\hbar}{i}\nabla_r = \frac{\hbar}{i}\left(\frac{\partial}{\partial r_x}, \frac{\partial}{\partial r_y}, \frac{\partial}{\partial r_z}\right)$ is the relative momentum operator.

The potential $V(\mathbf{r})$ does not depend on the center of mass position $\mathbf{R}$, which means that we can try a separation-of-variables solution to Eq. 5.144. Try a solution $\Psi(\mathbf{R}, \mathbf{r}) = \phi(\mathbf{R})\psi(\mathbf{r})$, then the two factors have to be solutions of

$$-\frac{\hbar^2}{2M}\nabla_R^2\phi(\mathbf{R}) = E_{\text{CM}}\phi(\mathbf{R}), \quad \text{and} \tag{5.145}$$

$$\left\{-\frac{\hbar^2}{2\mu}\nabla_r^2 + V(\mathbf{r})\right\}\psi(\mathbf{r}) = (E - E_{\text{CM}})\,\psi(\mathbf{r}). \tag{5.146}$$

Eq. 5.145 is easily solved, as it is the Schrödinger equation for a free particle with mass M

$$\phi(\mathbf{R}) = \frac{1}{\sqrt{V}}\exp\left[i\mathbf{P}\cdot\mathbf{R}/(\hbar M)\right] \quad \text{and} \quad E_{\text{CM}} = \frac{P^2}{2M}. \tag{5.147}$$

Here $\mathbf{P} = \mathbf{p}_1 + \mathbf{p}_2$ is the total momentum of the particle pair, E_{CM} is the kinetic energy of the center-of-mass motion, and V is some convenient normalization volume.

Eq. 5.146 describes the relative motion of the two particles; it looks like a one-particle Schrödinger equation. Whether is has bound states or not depends on whether the potential $V(\mathbf{r})$ is attractive; it should create a potential well that can trap the particles. Moreover, in three dimensions, the potential should be sufficiently strong for a bound state to develop; in other words, the potential well should be sufficiently large and deep. This is unlike the one-dimensional case, discussed in Sec. 5.4, where one always gets a bound state for an attractive potential.[42] The difference between the 3D and the 1D cases lies in the kinetic energy. In the 3D case

$$E_{\text{kin}} = \left\langle \psi \left| \frac{|\hat{\mathbf{p}}|^2}{2\mu} \right| \psi \right\rangle = \int \psi^*(\mathbf{r})\left[-\frac{\hbar^2}{2\mu}\nabla_r^2\right]\psi(\mathbf{r})d^3r > 0, \tag{5.148}$$

gives a larger positive contribution to the total energy than in the 1D case, and it can counteract the negative potential energy

$$E_{\text{pot}} = \langle\psi\,|V|\,\psi\rangle = \int \psi^*(\mathbf{r})V(\mathbf{r})\psi(\mathbf{r})d^3r, \tag{5.149}$$

completely. A bound state only develops if the total internal energy

$$E_{\text{tot}} = E - E_{\text{CM}} = E_{\text{kin}} + E_{\text{pot}} < 0. \tag{5.150}$$

Unfortunately, for a physically reasonable strength of electron-phonon coupling (and electron reduced mass μ), it turns out that $E_{\text{tot}} > 0$, which prevents the formation of a bound electron pair.

[42] Well, almost always. The statement is true for physically reasonable potentials, but I haven't explored it mathematically. As I said before, physicists stay away from unreasonable potentials.

5.6.2 Cooper's model

Now Cooper appears on the stage. He reasons as follows. It is not sufficient to solve Eq. 5.146 in order to obtain the wave function for two interacting electrons. In a metal there are lots of electrons around. Even if they do not interact much because their Coulomb potentials are screened, they are still fermions and have to obey the *Pauli principle*, meaning that no two of them can occupy the same quantum state. Imagine we have a metal of free electrons that do not interact at all with one another, as in the (in)famous fee-electron model. The wave functions of these free electrons are plane waves $\phi_{\mathbf{k}}(\mathbf{r}) = \exp[\mathbf{k}\cdot\mathbf{r}]/\sqrt{V}$ (with V the volume of the sample). In equilibrium, at low temperature, all states with $|\mathbf{k}| \leq k_F$ (states within the Fermi sphere) are occupied, whereas all states with $|\mathbf{k}| > k_F$ (states outside the Fermi sphere) are empty. In solid state physics folklore, such a system is called a (homogeneous) *electron gas.*

Now add two additional electrons that interact with one another via a weak attractive potential of the sort discussed above.[43] *Here is the key point.* Their wave function and energy can be found by solving Eq. 5.146, but under the *constraint* that they cannot occupy states that are already occupied by the other electrons that are present in the electron gas. Let us expand the wave function of Eq. 5.142 in plane waves

$$\Psi(\mathbf{r}_1,\mathbf{r}_2) = \frac{1}{V}\sum_{\mathbf{k}_1}\sum_{\mathbf{k}_2} a_{\mathbf{k}_1,\mathbf{k}_2}\exp[i\mathbf{k}_1\cdot\mathbf{r}_1]\exp[i\mathbf{k}_2\cdot\mathbf{r}_2]\,;\quad |\mathbf{k}_1|>k_F;\ |\mathbf{k}_2|>k_F. \tag{5.151}$$

This is just a (double) Fourier expansion, so it should always be possible. However, note that we insist that only plane waves with $|\mathbf{k}| > k_F$ participate. This is the result of the Pauli principle. Make the coordinate transformation of Eq. 5.143, and set $m_1 = m_2 = m_e$ (the mass of an electron; the effective mass, if you like) to get

$$\Psi(\mathbf{r}_1,\mathbf{r}_2) = \frac{1}{V}\sum_{\mathbf{k}_1}\sum_{\mathbf{k}_2} a_{\mathbf{k}_1,\mathbf{k}_2}\exp\left[i\frac{1}{2}(\mathbf{k}_1-\mathbf{k}_2)\cdot\mathbf{r}\right]\exp[i(\mathbf{k}_1+\mathbf{k}_2)\cdot\mathbf{R}]\,. \tag{5.152}$$

We note that $\mathbf{P} = \hbar\mathbf{K} = \hbar(\mathbf{k}_1+\mathbf{k}_2)$ is the total momentum of the electron pair. In the following we restrict ourselves to a *stationary electron pair,* i.e., an electron pair with zero total momentum. This state has $\mathbf{P} = 0$, $E_{\mathrm{CM}} = 0$, and $\phi(\mathbf{R}) = 1\sqrt{V}$, see Eqs. 5.145 and 5.147. Note that then

$$\mathbf{P} = 0 \Longleftrightarrow \mathbf{k}_1 = -\mathbf{k}_2 \equiv \mathbf{k}, \tag{5.153}$$

meaning that the momenta of the two electrons are always opposed.[44] Of the double sum in Eq. 5.152 only one sum remains, and we obtain a function that only depends on the relative coordinates $\mathbf{r}$

$$\psi(\mathbf{r}) = \frac{1}{\sqrt{V}}\sum_{\mathbf{k}} c(\mathbf{k})\exp[i\mathbf{k}\cdot\mathbf{r}]\,;\quad |\mathbf{k}|>k_F \ \text{ with } \ c(\mathbf{k}) \equiv a_{\mathbf{k},-\mathbf{k}}. \tag{5.154}$$

A bound state, i.e. a Cooper pair, is formed if the total energy that accompanies this wave function

$$E < 2E_F, \tag{5.155}$$

as the Fermi energy E_F is the energy at which a free electron can be added to the electron gas, while obeying the Pauli principle.

[43] It is a bit artificial to let only these two electrons interact, and leave the others as free electrons. This is how Cooper started. A generalization to all electrons interacting is part of the full BCS theory.

[44] It is not very difficult to extend the formalism to an electron pair where $\mathbf{P} \neq 0$, because the internal motion is decoupled from the center-of-mass motion, see Eqs. 5.145 and 5.146, but I skip that here.

Setting up the Schrödinger equation. Using Cooper's wave function, Eq. 5.154, in the Schrödinger equation, Eq. 5.146, one obtains

$$\frac{1}{\sqrt{V}}\sum_{\mathbf{k}'} c(\mathbf{k}')\left[\frac{\hbar^2 k'^2}{2\mu}+V(\mathbf{r})\right]\exp\left[i\mathbf{k}'\cdot\mathbf{r}\right]=\frac{E}{\sqrt{V}}\sum_{\mathbf{k}'} c(\mathbf{k}')\exp\left[i\mathbf{k}'\cdot\mathbf{r}\right];\quad |\mathbf{k}|>k_F \tag{5.156}$$

Taking the inner products

$$\frac{1}{\sqrt{V}}\int \exp\left[-i\mathbf{k}\cdot\mathbf{r}\right]\{...\}\,d^3r,$$

where $\{...\}$ is the left-hand side or the right-hand side of Eq. 5.156, and using

$$\frac{1}{V}\int \exp\left[i\left(\mathbf{k}-\mathbf{k}'\right)\cdot\mathbf{r}\right]d^3r=\delta_{\mathbf{k},\mathbf{k}'},$$

one arrives at a matrix-vector equation for the coefficients $c(\mathbf{k})$

$$\sum_{\mathbf{k}'}\left\{\frac{\hbar^2k^2}{2\mu}\delta_{\mathbf{k},\mathbf{k}'}+V(\mathbf{k}-\mathbf{k}')\right\}c(\mathbf{k}')=Ec(\mathbf{k});\quad |\mathbf{k}'|>k_F;\ |\mathbf{k}|>k_F. \tag{5.157}$$

Here

$$V(\mathbf{k}-\mathbf{k}')=\frac{1}{V}\int_{-\infty}^{\infty}V(\mathbf{r})\exp\left[i\left(\mathbf{k}'-\mathbf{k}\right)\cdot\mathbf{r}\right]d^3r, \tag{5.158}$$

which is essentially the Fourier transform of the interaction potential.

Spherical symmetry. It is possible to solve Eq. 5.157 numerically, but it is still in three dimensions, an therefore not entirely trivial. We can simplify it by assuming that the interaction potential in Eq. 5.146 is spherically symmetric, i.e., $V(\mathbf{r})=V(r)$ only depends on the distance r between the two particles. This is a reasonable assumption for most metals. The Fourier transform of a spherically symmetric potential, Eq. 5.158, $V(\mathbf{k}-\mathbf{k}')=V(|\mathbf{k}-\mathbf{k}'|)$, is also spherically symmetric.

The lowest energy state of a spherically symmetric potential has angular momentum $l=0$; its wave function is a so-called *s-wave.*[45] A s-wave is spherically symmetric, $\psi_S(\mathbf{r})=\psi(r)$. The Fourier transform of a spherically symmetric function is also spherically symmetric

$$c(\mathbf{k})=\int_{-\infty}^{\infty}\psi(r)\exp\left[i\mathbf{k}\cdot\mathbf{r}\right]d^3r=4\pi\int_0^{\infty}r^2\psi(r)j_0(kr)dr=c(k), \tag{5.159}$$

where $j_0(x)=\sin(x)/x$ is a spherical Bessel function.

Proof: the proof of Eq. 5.159 is straightforward. Write $d^3r=r^2\sin\theta dr d\theta d\phi$ and $\mathbf{k}\cdot\mathbf{r}=kr\cos\theta$, and integrate over the angles θ and ϕ.

The radial integral of Eq. 5.159 is called a *Fourier-Bessel transform.* It is easy to check that the inverse of such a transform is the same[46]

$$\psi(\mathbf{r})=\int_{-\infty}^{\infty}c(\mathbf{k})\exp\left[-i\mathbf{k}\cdot\mathbf{r}\right]d^3k=4\pi\int_0^{\infty}k^2c(k)j_0(kr)dk=\psi(r), \tag{5.160}$$

[45] Although in most, not in all superconductors are the Cooper pairs described by s-waves. If they are not, then the pairing potential $V(\mathbf{r})$ is not spherically symmetric.

[46] Apart from some constant prefactor involving 2 and π, where, unfortunately, different conventions exist for dividing this prefactor over the Fourier transform, and the inverse Fourier transform. Don't worry, the program packages we are going to use, take care of this prefactor.

Summarizing, in order to calculate spherically symmetric wave functions and potentials using Fourier transforms, we can use 1D Fourier-Bessel transforms, instead of 3D Fourier transforms.

Fourier-Bessel transforms (also called Hankel transforms) can be found in some numerical packages, but not in all (not in MATLAB, for instance). Luckily for us, we only need the zeroth order Bessel function j_0, and we can rewrite Eqs. 5.159 and 5.160 as

$$\gamma(k) = \int_0^\infty \zeta(r)\sin(kr)dr, \quad \zeta(r) = \int_0^\infty \gamma(k)\sin(kr)dr,$$
$$\text{with} \quad \zeta(r) = r\psi(r) \quad \text{and} \quad \gamma(k) = kc(k) \tag{5.161}$$

making use of the radial function $\zeta(r) = r\psi(r)$, as in Sec. 5.1, Eq. 5.6.[47] One-dimensional Fourier sine transforms should be part of most numerical packages [7].

5.6.2.1 Solving the Schrödinger equation.

Considering the discussion above, there is then a straight-forward way to solve the Schrödinger equation. As the problem is spherically symmetric, we can use the radial Schrödinger equation of Sec. 5.1, in particular Eq. 5.6 with $l = 0$ (s-wave)

$$\left\{-\frac{\hbar^2}{2\mu}\frac{d^2}{dr^2} + V(r)\right\}\zeta(r) = E\zeta(r), \tag{5.162}$$

where the reduced mass $\mu = \frac{1}{2}m_e$ for two electrons with mass m_e, see Eq. 5.144. We have to build in the constraint imposed by the Pauli principle, see Sec. 5.6.2. The most straightforward way of doing this, is to use Eq. 5.161

$$\gamma(k) = \text{FST}\left[\zeta(r)\right]; \quad \text{and for } k \le k_F \text{ set } \gamma(k) = 0, \tag{5.163}$$

where FST stands for the Fourier Sine Transform of Eq. 5.161.

In numerical calculations one would typically represent the wave function on a grid, such a $\zeta(r_j)$ with $r_j = j\Delta r; \quad j = 0, ..., N$. The radial Schrödinger equation, Eq. 5.162, with boundary conditions $\zeta(0) = \zeta(r_N) = 0$, then becomes a matrix eigenvalue equation of dimension $N - 1$, on the vector

$$\boldsymbol{\zeta} \quad \text{with} \quad [\boldsymbol{\zeta}]_j = \zeta(r_j), \quad j = 1, ..., N-1, \tag{5.164}$$

see Sec. 5.1.1, Eqs. 5.14, 5.15 in particular. The problem with that here, is that the constraint, Eq. 5.163, is not easily incorporated in many algorithms for finding eigenvalues/eigenvectors.

Details (skip this section; it is mainly there to remind myself of things that can go wrong): One might think of using the discretized function $\tilde{\zeta}(r_j) = \text{iFST}\left[\gamma(k_n)\right]$, where γ obeys the constraint, Eq. 5.163, and iFST is the (discrete) inverse Fourier sine transform. Formally one would have $\tilde{\boldsymbol{\zeta}} = \mathbf{P}\boldsymbol{\zeta}$, where $\boldsymbol{\zeta}$ is a general vector, and $\mathbf{P}$ is a matrix projecting on the space of functions that have $k > k_F$. One then has an eigenvalue equation of the type $\mathbf{PTP}\left(\mathbf{P}\boldsymbol{\zeta}\right) = \lambda\mathbf{P}\boldsymbol{\zeta}$, compare Eq. 5.14, where $\mathbf{PTP}$ projects the matrix $\mathbf{T}$ on the same space. As $\mathbf{P}^2 = \mathbf{P}$ (which holds for any projection operator), we have $\mathbf{PTP}\left(\mathbf{P}\boldsymbol{\zeta}\right) = \mathbf{PT}\left(\mathbf{P}\boldsymbol{\zeta}\right)$, and the projection can be carried out by

[47] See previous footnote.

simply projecting the result of $\mathbf{T}(\mathbf{P}\boldsymbol{\zeta})$ on that space. Any algorithm that uses an operation with $\mathbf{T}$ only, can be adapted rather simply to this projection. For instance, in power iteration we have the step $\boldsymbol{\zeta}_{n+1} = \mathbf{T}\boldsymbol{\zeta}_n$, see Eq. 2.9, which can be modified to $\tilde{\boldsymbol{\zeta}}_{n+1} = \mathbf{P}\mathbf{T}\tilde{\boldsymbol{\zeta}}_n$. However, power iteration gives you the largest eigenvalue, whereas we typically want to have the lowest one. One could think of using shifted inverse power iteration with the matrix $(\mathbf{T} - s\mathbf{I})^{-1}$, and s a convenient shift, see Sec. 2.3.1. To apply it in projected form, we need to construct $[\mathbf{P}(\mathbf{T} - s\mathbf{I})\mathbf{P}]^{-1}$ (implicitly, by solving $[\mathbf{P}(\mathbf{T} - s\mathbf{I})\mathbf{P}]\boldsymbol{\zeta}_{n+1} = \tilde{\boldsymbol{\zeta}}_n$, compare Eq. 2.14). Unfortunately, we don't have the matrix $\mathbf{P}(\mathbf{T} - s\mathbf{I})\mathbf{P}$, and we don't want to construct it explicitly (for instance, it will not be a tridiagonal matrix, nor a sparse matrix). Note, for instance, that $[\mathbf{P}(\mathbf{T} - s\mathbf{I})\mathbf{P}]^{-1} \neq \mathbf{P}(\mathbf{T} - s\mathbf{I})^{-1}\mathbf{P}$, so $(\mathbf{T} - \mathbf{s}\mathbf{I})\boldsymbol{\zeta}_{n+1} = \tilde{\boldsymbol{\zeta}}_n$; $\tilde{\boldsymbol{\zeta}}_{n+1} = \mathbf{P}\boldsymbol{\zeta}_{n+1}$, is not the correct projection rule.[48] In summary, the constraint of Eq. 5.163 is incorporated easily, if we use algorithms that only use the operation $\mathbf{T}\boldsymbol{\zeta}_n$. Forget about algorithms that use more complicated operations. At the same time, we still want to find the lowest eigenvalue.

As the constraint is most easily expressed in k-space, it is advantageous to also express the radial Schrödinger equation, Eq. 5.162, in k-space, which we can do by taking the Fourier sine transform of the left- and right-hand sides.

$$\frac{\hbar^2 k^2}{2\mu}\gamma(k) + \mathrm{FST}\left[V(r)\zeta(r)\right](k) = E\gamma(k); \quad k > k_F. \tag{5.165}$$

A discrete Fourier sine transform gives a representation of the function $\gamma(k_n)$ on the points $k_\ell = \ell\Delta k$; $\ell = 1, ..., N-1$, where $\Delta k = \pi/r_N$. The eigenvalue equation, Eq. 5.165, does not seem to be easily attacked by any of the methods we have discussed, which require a matrix-vector form

$$\mathbf{T}\boldsymbol{\gamma} = E\boldsymbol{\gamma} \;\text{ with }\; [\boldsymbol{\gamma}]_\ell = \gamma(k_\ell) \;\text{ and }\; [\mathbf{T}]_{\ell m} = \delta_{\ell m}\frac{\hbar^2 k_\ell^2}{2\mu} + V_{\ell m}, \tag{5.166}$$

for $\ell = 1, ..., N-1$ and $m = 1, ..., N-1$. From Eq. 5.165 we don't have the matrix $V_{\ell m}$. In principle we could construct it,[49] but I don't want to. It is not a sparse matrix (not tridiagonal, for instance), and would be very cumbersome to work with if the dimension N becomes large.

Fortunately, if we restrict ourselves to algorithms that only need the matrix-vector product $\boldsymbol{\theta} = \mathbf{V}\boldsymbol{\gamma}$, then we can use a trick. Suppose we have the vector $\boldsymbol{\gamma}$. Then

$$\boldsymbol{\zeta} = \mathrm{iFST}\left[\boldsymbol{\gamma}\right]; \quad \boldsymbol{\theta} = \mathrm{FST}\left[\mathbf{v}\boldsymbol{\zeta}\right] \;\text{ with }\; [\mathbf{v}]_j = V(r_j), \quad j = 1, ..., N-1, \tag{5.167}$$

where $\mathbf{v}\boldsymbol{\zeta}$ is the element-by-element product of $\mathbf{v}$ and $\boldsymbol{\zeta}$, does what we need, without ever having to construct a matrix explicitly. If we use iFST as the inverse Fourier sine transform, then the prefactors associated with discrete Fourier transforms are taken into account automatically. The constraint of Eq. 5.163 is incorporated easily, by setting

$$[\boldsymbol{\theta}]_\ell = 0 \;\text{ for }\; k_\ell \leq k_F. \tag{5.168}$$

---

[48] *Details of details:* $\mathbf{P}\mathbf{A}^{-1}\mathbf{P} = \left[\mathbf{P}\mathbf{A}\mathbf{P} - \mathbf{P}\mathbf{A}\mathbf{Q}\left(\mathbf{Q}\mathbf{A}\mathbf{Q}\right)^{-1}\mathbf{Q}\mathbf{A}\mathbf{P}\right]^{-1}$, where $\mathbf{Q} = \mathbf{I} - \mathbf{P}$, which is obviously not equal to $[\mathbf{P}\mathbf{A}\mathbf{P}]^{-1}$, unless $\mathbf{Q} = \mathbf{0}$, but then $\mathbf{P} = \mathbf{I}$, which is a trivial case.

[49] Using the *convolution theorem* particularly adapted to (discrete) Fourier sine transforms, for instance.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide – Release 2.0*, SIAM, Philadelphia, PA, 1994. (Software and guide are available from Netlib at URL http://www.netlib.org/lapack/).

[2] R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems by Implicitely Restarted Arnoldi Methods*, SIAM, Philadelphia, PA, 1998. (The software and this manual are available at URL http://www.caam.rice.edu/software/ARPACK/).

[3] G. H. Golub and C. F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 2nd ed., 1989.

[4] S. D Conte and C. de Boor, *Elementary Numerical analysis: An Algorithmic Approach*, 3rd ed., McGraw-Hill Book Company, New York, 1980.

[5] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, 2nd ed., Springer-Verlag, NewYork, 1993.

[6] W. Y. Yang, W. Cao, T.-S. Chung, and J. Morris, *Applied Numerical Methods Using Matlab*™, Wiley-Interscience, Hoboken, NJ, 2005.

[7] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed., Cambridge University Press, Cambridge, 2002.

[8] P. Pulay, *Convergence Acceleration of Iterative Sequences. The Case of SCF Iteration*, Chem Phys. Lett. 73, 393-398 (1980).

[9] G. Kresse and J Furthmüller, *Efficiency of Ab-Initio Total Energy Calculations for Metals and Semiconductors Using a Plane-Wave Basis Set*, Comput. Mater. Science 6, 15-50 (1996).

[10] T. Rohwedder and R. Schneider, *An Analysis for the DIIS Acceleration Method Used in Quantum Chemistry Calculations*, J. Math. Chem. 49, 1889-1914 (2011).

[11] D. G. Anderson, *Iterative Procedures for Nonlinear Integral Equations,* J. Assoc. Comput. Mach. 12, 547-560 (1965).

[12] H. F. Walker and P. Ni, *Anderson Acceleration for Fixed-Point Iterations*, SIAM J. Numer. Anal. 49, 1715-1735 (2011).

[13] P. Arbenz, *Lecture Notes on Solving Large Scale Eigenvalue Problems*, ETH Zürich, 2016.

[14] E. R. Davidson, *The Iterative Calculation of a Few of the Lowest Eigenvalues and Corresponding Eigenvectors of Large Real Symmetric Matrices*, J. Comput. Phys. 17, 87 (1975).

[15] G. L. G. Sleijpen and H. A. van der Vorst, *A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems*, SIAM J. Matrix Anal. Appl. 17, 401 (1996).

[16] G. Kresse and J. Furthmüller, *Efficient Iterative Schemes for Ab-Initio Total-Energy Calculations Using a Plane-Wave Basis Set*, Phys. Rev, B 54, 11169 (1996).

[17] D. M. Wood and A. Zunger, *A New Method for Diagonalising Large Matrices*, J. Phys. A: Math. Gen. 18, 1343 (1985).

[18] H. Sellers, *The C^2-DIIS Convergence Acceleration Algorithm*, Int. J. Quant. Chem. 45, 31 (1993).

[19] B. Liu, *The Simultaneous Expansion Method for the Iterative Solution of Several of the Lowest Eigenvalues and Corresponding Eigenvectors of Large Real-symmetric Matrices*, in Report on the *Workshop Numerical Algorithms in Chemistry: Algebraic Methods*, Technical Report LBL-8158, Lawrence Berkeley Laboratory, University of California, Berkeley (1978).

[20] R. Fletcher, *Practical Methods of Optimization*, 2nd ed., Wiley-Interscience, Chichester, UK, 1987.

[21] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer, New York, 1999.

[22] M. Hintermüller, *Nonlinear Optimization, Part I: Unconstrained and box-constrained problems*, Berlin School of Mathematics – Basic Course, Humboldt-University of Berlin, 2009.

[23] J. R. Shewchuk, *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, ed. $1\frac{1}{4}$, School of Computer Science, Carnegie Mellon University, 1994.

[24] J. Nocedal, *Updating Quasi-Newton Matrices With Limited Storage*, Mathematics of Computation 35, 773-782 (1980).

[25] H.-R. Fang and Y. Saad, *Two Classes of Multisecant Methods for Nonlinear Acceleration*, Numer. Linear Algebra Appl. 16, 197–221 (2009).

[26] M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos, *Iterative Minimization Techniques for Ab Initio Total-Energy Calculations: Molecular Dynamics and Conjugate Gradients*. Rev. Mod. Phys. 64. 1045–1097 (1992).

[27] S. Tomov, J. Langou, A. Canning, L.-W. Wang, and J. Dongarra, *Conjugate-Gradient Eigenvalue Solvers in Computing Electronic Properties of Nanostructure Architectures*, 2005.

[28] P. Arbenz, U. L. Hetmaniuk, R. B. Lehoucq, and R. S. Tuminaro, *A Comparison of Eigensolvers for Large-Scale 3D Modal Analysis Using AMG-Preconditioned Iterative Methods*, Int. J. Numer. Meth. Engng. 64, 204–236 (2005).

[29] R. Wrede and M. R. Spiegel, *Schaum's Outline of Advanced Calculus*, 2nd ed., McGraw-Hill, New York, 2002.

[30] K. Neymeyr, E. Ovtchinnikov, and M. Zhou, *Convergence Analysis of Gradient Iterations for the Symmetric Eigenvalue Problem*, SIAM J. Matrix Anal. Appl. 32, 443-456 (2011).

[31] V. S. Bagotsky, *Fundamentals of Electrochemistry*, 2nd ed., Wiley-Interscience, Hoboken, NJ, 2006.

[32] G. Mazur, *An Improved SCPF Scheme for Polarization Energy Calculations*, J. Comput. Chem. 29, 988–993 (2008).

[33] L. Lagardère, F. Lipparini, É. Polack, B. Stamm, É. Cancès, M. Schnieders, P. Ren, Y. Maday, and J.-P. Piquemal, *Scalable Evaluation of Polarization Energy and Associated Forces in Polarizable Molecular Dynamics: II.Towards Massively Parallel Computations using Smooth Particle Mesh Ewald*, J Chem Theory Comput. 10, 1638–1651 (2014).

[34] D. J. Griffiths, *Introduction to Quantum Mechanics*, Prentice Hall, Upper Saddle River, NJ, 1995.

[35] S. T. Thornton and J. B. Marion, *Classical Dynamics of Particles and Systems*, 5th ed., Thomson/Brooks/Cole, 2005.

[36] P. W. Atkins and J. de Paula, *Physical Chemistry*, 8th ed., Oxford University Press, 2006.

[37] C.H. Townes and A. L. Schawlow, *Microwave Spectroscopy*, Dover, Mineola, NJ, 1975.

[38] J. M. Ziman, *Electrons and Phonons*, Oxford University Press, London, UK, 1960.

[39] G. Grosso and G. Pastori Parravicini, *Solid State Physics*, Academic Press, Amsterdam, 2000.

[40] H. Goldstein, C. Poole, and J. Safko, *Classical Mechanics*, 3rd ed., Addison-Wesley, San Francisco, US, 2000.

[41] E. Kaxiras, *Atomic and Electronic Structure of Solids*, Cambridge University Press, Cambridge, UK, 2003.

[42] R. M. Martin, *Electronic Structure*, Cambridge University Press, Cambridge, UK, 2004.

[43] M. Stone and P. Goldbart, *Mathematics for Physics*, Pimander-Casaubon, Alexandria, 2008.

[44] H. Römer, *Theoretical Optics*, Wiley-VCH, Weinheim, BRD, 2005.

[45] M. Tinkham, *Introduction to Superconductivity*, McGraw-Hill, New York, 1975.