Nummerical Programming I

Personal summary by Gilberto Lem

Winter 2017-2018

Contents

1 Basics

1.1 Floating Point Arithmetics

As a computer has finite memory, it cannot store numbers with infinite number of digits. Instead, the computer has to choose a **precision**, i.e. a fixed number of digits to represent the number.

To increase the range of numbers it can store, the computer defines a "floating" decimal point. This is achieved dedicating some of the digits to represent where the point is, i.e. an exponent.

By convention, the floating point numbers always get normalized: the decimal point is set such that the first digit is different than zero.

A number greater than the biggest possible floating point is an overflow, and a smaller than the smallest possible is an underflow.

1.1.1 Machine Epsilon

Denoted by ϵ, it is a reference to the accuracy of a computational system. A smaller number than this is considered zero. Mathematically, it is the smallest number in the computer such that $1 + \epsilon \neq 1$, i.e. the difference in the computer between 1 and the next possible number.

Note that the difference between consecutive numbers is not going to be always ϵ; it will increase as the number increases. Consider for example a binary number with precission of 4 digits:

$$1.001 \times 10^3 - 1.000 \times 10^3 = 1.000 \times 10^0,$$

however

$$1.001 \times 10^{-3} - 1.000 \times 10^{-3} = 1.000 \times 10^{-6}$$

1.1.2 Rounding

Every real number is rounded to a floating point number in the computer. That floating point number $\hat{a} = \text{round}(a)$ satisfies:

$$\frac{\|a - \hat{a}\|}{a} \leq \frac{\epsilon}{2}$$

1.2 The Condition Number

Denoted by κ, it relates the error due to rounding of an input to an upper limit of error due to rounding of an output. Mathematically:

$$\|f(x) - \text{round}(f(x))\| \leq \kappa_f \|x - \text{round}(x)\|$$

As an example, here is the condition number of the simplest arithmetic operations:

- **Sum:**
  - For the same sign: $\kappa_+ = 1$
  - For different signs: If the numbers are close in absolute value, κ_- can get be very large. This is called **cancellation**. Take for example again $1.001 \times 10^3 - 1.000 \times 10^3 = 1.000 \times 10^0$. Here the last three digits of the result are just speculation, as we did not have that level of accuracy on the input. 3 digits were cancelled.
- **Multiplication:** $\kappa_* = 2$
- **Division:** $\kappa_/ = 2$

In general for functions, if the problem $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable, then:

$$\kappa_f(x) = \frac{\sum_i |\partial_i f(x)||x_i|}{|f(x)|}$$

1.3 Stability of algorithms

When you apply more than one operation, the error starts amplifying. As a general rule of thumb:

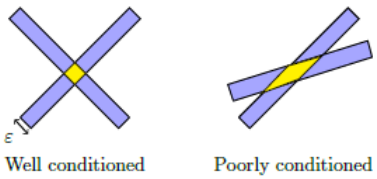
- **Stable algorithm:** $\|f(x) - \text{round}(f(x))\| \approx k_f(x)\epsilon$
- **Unstable algorithm:** $\|f(x) - \text{round}(f(x))\| \gg k_f(x)\epsilon$

2 Linear Systems

We want to solve $Ax = b$, for x. This means b can be expressed as a linear combination of the columns of A. A naive approach to this would be to use the inverse matrix of A. However, getting the inverse is too computationally expensive and numerically unstable.

2.1 Conditioning of Linear Systems

For linear systems, the numeric error of the input can vary considerably the error in the output. See for example the next figure. It represents a 2d system. Due to numeric error, instead of finding the intersection of *lines*, we have to find intersecion of *lines with ranges of errors*. Having similar directions in our line-ranges would have an important effect on the solution-ranges:



The condition number, which is $\kappa(A) = \|A^{-1}\|\|A\|$ can be used to give an idea of how problematic can be. Numerically we asume that our matrix is singular (and thus not solvable) if $1/k(A) < \epsilon$.

We can estimate the quality of our solution x found using a backward error, which is basically a normalized residual $\omega = \frac{\|Ax-b\|}{\|A\|\|b\|}$.

2.2 Gaussian Elimination

Triangular systems are easy to solve because they allow forward or backward substitution. They require $\mathcal{O}(n^2)$ flops. Gaussian elimination allows us to get our matrix to this triangular form without affecting the solution. The idea of Gaussian elimination is used in methods as LU decomposition and QR decomposition to transform our matrix A to triangular forms.

2.2.1 LU decomposition

Consists of decomposing $A = LU$. L is lower triangular and U is upper triangular. Then you can say:

1. $LUx = b$
2. Solve $Lz = b$ for z
3. Solve $Ux = z$ for x

The decomposition requires $\mathcal{O}(n^3)$ flops.

2.3 QR decomposition

This method decomposes $A = QR$. Q is orthogonal (and thus squared) and R is upper triangular. One can achieve this decomposition via Householder reflections, or rotations. As Q is orthogonal $Q = Q^T$, and thus:

1. $QRx = b$
2. Solve $Rx = Q^T b$ for x

One of the advantages of QR decomposition is that it can be done for rectangular matrices, and thus it is useful for least square problems. It is also more stable than LU, because orthogonal matrices have a condition number of 1. However, it is slower than LU decomposition. For a $m \times n$ matrix it requires $2mn^2 - 2n^3/3$ flops.

2.4 Least Square Problems

Suppose we want to adjust m measurements as a function. The errors would be the difference between our model and the measurements. We get the best possible adjustment when we minimize the squares of those errors.

2.4.1 Normal Equations

Analytically, as a solution for this we get the Normal Equation. The solution is unique.

$$A^T Ax = A^T b$$

However, solving it is a very costly process, and its condition number can be very large. To overcome this we can make a QR-decomposition $A = QR$. It can be shown that the solution to $Rx = Q^T b$ minimizes the error.

3 Eigenvalue Problems

The straightforward approach to finding eigenvalues of a matrix A is to find roots its characteristic polynomial. However, this problem is very badly conditioned. We therefore use other methods to find eigenvalues to matrices. In practice actually we use these alternative methods to solve eigenvalue problems and find roots to polynomials.

3.1 Power iteration

If A is full rank, any random vector can be expressed as a linear combination of the matrix eigenvectors. If we multiply that random vector by our matrix repeatedly we are going to converge to the eigenvector with the largest-module eigenvalue. After that, the eigenvalue can be easily computed with a Rayleigh quotient. This of course would need our random vector to not be linearly independent of our largest-module-eigenvalue eigenvector.

This implies iterations, and with that, we have to define a stop-criterion. For that we use again the backward error:

$$\omega = \frac{\|Ax - b\|}{\|A\| \, \|b\|}$$

We cannot expect to go to zero due to numerical error. Instead, we stop iterating when $\omega \lesssim n\epsilon$, where $n \times n$ is the size of the matrix. Note that ω needs the computing of the norm of the matrices. As this error is just a rough indicator of our performance, we can use the Frobenius norm (sum of the squares of all elements in the matrix), which is cheap.

3.1.1 Inverse Power Iteration

We can generalize the power method, using a shift to find the eigenvector with the eigenvalue closest to that shift.

3.2 QR Iteration

We can use QR-decomposition to find eigenvalues too. When we use the orthogonal matrix Q to repeatedly get similar matrices to A: $A_{k+1} = Q_k^T A_k Q_k$, A_{k+1} starts converging to an upper triangular matrix, with the eigenvalues of the original A in the diagonal.

Note that we are getting a QR-decomposition with each iteration, and each one costs $\mathcal{O}(n^3)$. To make the convergence faster, we can preprocess A:

- Transforming the matrix A to Hessenberg form: This costs $\mathcal{O}(n^3)$, which is the same as one QR decompostion, but reduces considerably the number of iterations required.
- Using Rayleigh or Wilkinson shifts. The Wilkinson shifts are the ones used in the matlab implementation. The `eig(A)` method costs $\mathcal{O}(m^3)$.

3.3 Singular Value Decomposition

If a matrix A is symmetric, it can be orthogonally diagonalized, i.e. it can be decomposed in PDP^{-1}, where P is an orthogonal matric with the eigenvectors of A in columns, and D is a diagonal matrix with eigenvalues of A.

However, if A is not squared, it is not symmetric. SVD uses an orthogonal diagonalization of A^TA, which ensures the symmetry even for non-squared matrices. It decomposes the squared matrix and results in a decomposition of $A = USV^T$, where U and V are unitary (normalized complex version of orthogonal matrices: $UU^\dagger = I$) and S has the singular values (square roots of eigenvalues) of A^TA in the diagonal.

The following are some applications of SVD:

- **Computation of the rank:** The rank of A would be simply the number of singular values that are greater than zero (or numeric zero).
- **Compute norms:** The norm of the matrix A is just the greatest singular value, and the Frobenius norm is just the euclidian norm of the diagonal of S.
- **Solve Least Squared Problems:** As S, U and V are easily invertible you can use SVD to compute: $x = V\hat{S}^{-1}\hat{U}^Tb$, where the hat means reduced form.
- **Low rank approximations:** You can decompose A as sums of rank 1 matrices, each one corresponding to one eigenvector, and then consider only the most important ones.

4 Interpolation

Suppose we want to find a function that satisfies $n+1$ interpolation conditions (from 0 to n). This means having $n+1$ pairs of $(x_k, f(x_k))$.

4.1 Polynomial Interpolation

A polynomial of degree n has $n+1$ coefficients, i.e. $n+1$ degrees of freedom. Therefore, there is a **unique** polynomial of degree n that satisfies the $n+1$ conditions.

4.1.1 Linear System

The natural thing is to find the coefficients is to make a linear system. This system is full rank, and thus it is solvable. However it has the following problems:

- The matrix is poorly conditioned which means it has large error
- Experimentally we note that the solutions oscillate extremely in the boundaries
- It is expensive, it requires $\mathcal{O}(n^3)$ flops.

4.1.2 Lagrange Polynomial

Another option is to define $n+1$ polynomials, each one with the value 1 for one of the $n+1$ points and 0 for the rest. After that you can multiply each one of them by the function value in its nonzero point, and sum all of them to get a general polynomial. These method of obtaining the polynomial gets the same analytical polynomial as in the linear system, however is numerically stable.

Furthermore, one can use the **First** and **Second Barocentric Formulas,** as a way to reduce computational effort in this method.

The algorithm for implementing this is stable, however, it is still prone to rounding errors, because the coefficients get very high when n increases.

4.1.3 Condition number and Chebysheff nodes.

If the nodes are equidistant, the condition number increases strongly with n. However one could instead choose nodes given by equidistant points in a circle of diameter of the interval. These are called Chebysheff nodes and result in a very small condition number. For example, for $n = 20$, we have a condition number of $\sim 10^4$ for equidistant nodes and ~ 3 for Chebysheff nodes.

If we introduce the election of nodes to the problem there are many different polynomials that can approach a single function. However, it can be proofed that using Chebysheff nodes is at worst only ~4 times worse than the best possible polynomial.

4.2 Trigonometric Interpolation

Polynomials do not work for periodic functions. Sines and cosines, or complex exponentials do.

4.2.1 Fourier series

In general we can use Fourier series to interpolate a periodic function. This is a sum of sines and cosines with different frequencies, and different weights (coefficients).

4.2.2 Discrete Fourier Transform (DFT)

To get each coefficient of a Fourier series, we need to do an integral. To achieve this computationally we have to discretize, and thus the coefficients can be get by a matrix-vector multiplication with complex numbers, which needs $\mathcal{O}(n^2)$ flops.

4.2.3 Fast Fourier Transform (FFT)

This method plays with indexes to get the DFT result in only $\mathcal{O}(n \log_2 n)$ flops.

4.3 Spline Interpolation

Polynomial interpolation finds one polynomial that fits all condition, however, it is possible to use instead piecewise polynomials to interpolate. Usually this is done for equidistant nodes.

4.3.1 Linear Spline

In this kind of interpolation you make a different line for each set of consecutive nodes. This problem is well conditioned, however as we are just drawing straight lines, the derivative of the global function is discontinuous.

4.3.2 Cubic Spline

Here we use a 3rd degree polynomial for each set of consecutive nodes, with the following conditions: in a given node, both of the polynomials that it *touches* have to have the same value, the same derivative and the same second derivative, to make the global function smooth. Introducing those three additional restrictions in each node, we now have to fulfill $(n+1) + 3(n-1) = 4n - 2$ conditons. However, we have n polynomials, and thus $4n$ coefficients, which give us liberty to choose 2 extra conditions, which can be:

- **Natural spline:** Second derivatives at the boundaries set to 0
- **Complete spline:** Derivatives at the boundaries set to constants.

- **Not a node:** Consistent third derivatives (and thus identical polynomials) in the next-to-boundary nodes.
- **Periodic splines:** First and second derivatives of the boundaries are equal.

To compute this spline we can solve a tridiagonal linear system, which needs only $\mathcal{O}(n)$ operations.

4.4 Interpolation Accuracy

In general, the error between our interpolating polynomial and the real function is $\mathcal{O}(h^{n+1})$, where h is the distance between nodes.

5 Quadrature

In Numerics, "Quadrature" usually refers to computing the value of a definite integral, while "Integration" refers to solving an ODE.

5.1 General Quadrature Formula

Suppose we want to solve the following definite integral:

$$I(f) = \int_a^b f(x)\mathrm{d}x$$

To solve it, we can use the fundamental theorem of calculus, however, this comes with the following disadvantages:

- As it is a difference, it can lead to cancellation
- We need the analytic antiderivative, which can be complicated or impossible to get
- We may not even know our target function

To ensure the integrability, we approximate the function with an interpolation, and then define the general quadrature formula:

$$Q(f) \coloneqq \sum_{j=0}^{n} w_j f(x_j)$$

So the quadrature is just a sum of the values of the function in some nodes, each one multiplied by an associated weight.

5.1.1 Stability of the General Quadrature Formula

The condition number of the analytical integration $I(f)$ is just the length of the interval on which we are integrating

$$\kappa_{_I} = (b - a)$$

On the other side, the condition number of our quadrature formula $Q(f)$ is

$$\kappa_{_Q} = \sum_{j=0}^{n} |w_j|$$

For stability, the condition numbers have to be similar, and thus if the sum of the absolute values of the weights is around the size of the interval, we have a stable formula. If it is much larger we have an unstable formula.

To ensure consistency we have to define the quadrature to be exact for some function. We choose $f(x) = 1$. Then our quadrature formula becomes

$Q(f) = \sum w_k \overset{!}{=} I(f) = \int_a^b \mathrm{d}x = b - a$. A consistent cuadrature formula, if all the weights are positive, would mean $\kappa_I = \kappa_Q$, and thus a consistent quadrature formula with positive weights is stable.

5.1.2 Accuracy of the General Quadrature Formula

In general, using an interpolating polynomial of degree n on equidistant nodes for the quadrature, we obtain an error $\mathcal{O}(h^{n+2})$.

However, if you have a polynomial of order even, interpolating a function, and one of the nodes is the midpoint of the interval, you get the accuracy of the following odd degree. This happens because the area under the polynomial of the following odd degree will be the same as the even degree, due to symmetry around the midpoint. This phenomenon is called **Superconvergence**.

For example, if you have a polynomial of order 0 and your node is the midpoint, you will get an error of $\mathcal{O}(h^3)$, instead of the expected $\mathcal{O}(h^2)$.

5.1.3 Composite Rules

As we have seen in the interpolation chapter, polynomial interpolation for a high number of nodes is unstable. A composite rule consists in partitioning the interval, and in each one using several piecewise polynomials of low degree to integrate.

5.2 Gaussian Quadrature

Recall our general quadrature formula:

$$Q(f) \coloneqq \sum_{j=0}^{n} w_j f(x_j)$$

Until now we have been considering fixed equidistant nodes. This leads to having $n+1$ degrees of freedom (as weights) in our quadrature formula, which we could use to integrate exactly polynomials of at most $n+1$ coefficients, i.e. n-th degree polynomials.

However, if we do not considered fixed the nodes, we now have $2(n+1)$ degrees of freedom, and we could integrate exactly polynomials of degree $2n+1$.

We could get the values of the nodes as the eigenvalues of the following linear system:

$$xp(x) = Tp(x),$$

where the matrix T is a tridiagonal matrix given by:

$$\begin{bmatrix} \alpha_0 & \beta_0 & & \\ \beta_0 & \alpha_1 & \beta_1 & \\ & \ddots & \ddots & \ddots \\ & & \beta_{n-1} & \alpha_n \end{bmatrix}$$

where

$$\beta_{k-1} =< xp_k, p_{k-1} >, \;\; \alpha_k =< xp_k, p_k >,$$

and we obtain p_{k+1} with a simplified Gram-Schmidt orthogonalization process called Lanczos algorithm:

$$\hat{p}_{k+1} = xp_k - \beta_{k-1}p_{k-1} - \alpha_k p_k$$
$$p_{k+1} = \frac{\hat{p}_{k+1}}{\|\hat{p}_{k+1}\|}$$

On the other side, the weights are given by the integral of the Lagrange polynomials. We could get advantage of the orthogonality of this set to get the following expression:

$$w = (b-a)\text{diag}(Q(1,:))Q(1,:)^T$$

where Q is a matrix formed with the normalized eigenvectors of T in columns, and then:

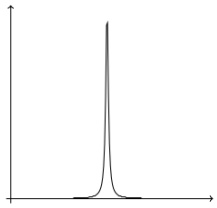
5.2.1 Gaussian Quadrature Algorithm

1. Use Lanczos iteration to get T
2. $[\mathtt{Q}, \mathtt{X}] = \mathtt{eig}(\mathtt{T})$; $\mathtt{x} = \mathtt{diag}(\mathtt{X})$;
3. $\mathtt{w} = (\mathtt{b} - \mathtt{a}) * \mathtt{Q}(1,:).\hat{}\,2$;

If our polynomial is k+1 times differentiable, we will have an error of $\mathcal{O}(n^{-(k+1)})$, and if it is infinitely differentiable, we will have an error of $\mathcal{O}(\rho^{-n})$,with $\rho < 1$.

5.3 Adaptive Quadrature

Some functions as the following may have irregularities:



For integrating them with the previous methods we would need to define a large number of nodes. Instead, we could use a **subdivision algorithm** to have a small error. It consists in starting with a set of nodes, and if the error is high in a certain interval between two nodes, compute the quadrature of that interval with more nodes, until we get to a desired tolerance.

Here we are assuming that we can compute the error, but for that we would need to have the exact value of the integral, which would make the process of finding the quadrature pointless. However, we could use an estimator of the integral with a quadrature formula of higher order (note thas this would be equivalent to comparing our quadrature value with a formula of lower order, considering our high order quadrature as the most exact one). For example, if we are using in our algorithm a Simpson's rule ($\mathcal{O}(h^5)$), we could compare our quadrature value with a trapezoidal rule ($\mathcal{O}(h^3)$) and see if we achieve the desired tolerance. This error estimator is **asymptotically exact**.

5.4 Monte-Carlo Quadrature

Any of the previous methods would need $\mathcal{O}(n^d)$ quadrature nodes to get an error of $\mathcal{O}(n^{-p})$, where d is the dimension of our space and p is the order of our quadrature. For big dimensions this becomes too expensive. To overcome this we could use the definition of expected value:

$$I(f) = \int f(x)\mathrm{d}x = \mathbb{E}(f(x)) = \frac{1}{N}\sum_{k=1}^{N} f(X_k),$$

where X is a random variable with uniform distribution in a d-dimensional space.

Thus we could just compute the mean of random evaluations of the function to get an approximate value of the integral.

The error for this method is $\mathcal{O}(N^{-1/2})$, where N is the number of points considered in the domain. Note that this do not include the number of dimensions at all, so it is useful for large d, however, $N^{-1/2}$ represents a slow convergence (it's not even linear).

6 Optimization

Abstractly, optimization consists on finding the right variables x^* in an admissable domain X, which minimize an objective function f.

A local minimum would be defined in a neighborhood U :

$$f(x^*) \leq f(x), \quad \text{for all } x \in U$$

A global minimum:

$$f(x^*) \leq f(x), \quad \text{for all } x \in X$$

Both would be **strict** if the previos symbols were $<$ instead of $\leq$.

6.1 Optimality conditions

Consider an open domain $X \subset \mathbb{R}^d$, and a twice continuously differentiable $f : X \to \mathbb{R}$.

Here every point x^* with $\nabla f(x^*) = 0$ represents a **stationary point**. However, this condition is not enough to find a minimum. These points could be minima, maxima or saddle points.

For it to be a **local minimum**, in an open domain , the two following conditions would have to hold:

- $\nabla f(x^*) = 0$
- $H_f\,(x^*)$ is positive definite (i.e. $v^T H_f(x^*)v > 0$ for all $v \in \mathbb{R}^d \backslash \left\{\vec{0}\right\}$)

where H_f is the Hessian matrix

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}(x)$$

6.2 Gradient descent

Gradient descent is an optimization method, whose main idea is to start in some point x_0 in the domain and compute another point x_1 such that $f(x_1) < f(x_0)$, and then repeat. To develop it, we use a Taylor expansion:

$$f(x_0 + \vec{v}) = f(x_0) + \nabla f(x_0) \cdot \vec{v} + \mathcal{O}\left(\|v\|^2\right),$$

where $\vec{v}$ is the vector in whose direction we move. This leads to the following algorithm for **gradient descent**:

While $|\nabla f(x_k)| > TOL$

1. Compute direction: v_k s.t. $\nabla f(x_k)v_k < 0$
2. Compute stepsize: h_k s.t. $f(x_k + h_k v_k) < f(x_k)$

For the first stage of the algorithm, we could increase the efficiency of the method by defining v_k always as the direction in which the function decreases most rapidly in x_k, i.e. $v_k = -\frac{\nabla f(x_k)}{\|\nabla f(x_k)\|}$. This is called the **steepest descent** method.

For the second stage of the algorithm, computing the step size, one strategy is to use the **Armijo method**, which works with trial and error. It first defines a $\gamma \in (0,1)$, and then finds the greater number $h_k \in \left\{1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, ...\right\}$ such that $f(x_k + h_k v_k) \leq f(x_k) + \gamma h_k \nabla f(x_k) v_k$.

The steepest descent algorithm with Armijo step size strategy eigher converges to a stationary point, or generates a sequence of stationary points.

6.3 Abstract fixed point iteration

The problem of solving $\nabla f(x) = 0$ (finding fixed points) would be equivalent to solve

$$g(x^*) = 0$$

for some function $g = \nabla f : \mathbb{R}^d \to \mathbb{R}^d$.

This can be analytically impossible even for simple functions as polynomials. Instead we can design an iterative algorithm to compute a sequence of points that converges to a solution:

$$x_{k+1} = \phi(x_k),$$

where ϕ is the iteration function.

How do we know if an iteration sceheme works? For a continuous ϕ :

- If ϕ is **locally convergent** (i.e. it goes to $\bar{x}$ for every initial condition in a neighborhood U), then $\bar{x}$ is a fixed point. This is the case when:
  - ϕ is continuously differentiable, and $\|D\phi(\bar{x})\| < 1$, or
  - $\rho(D\phi(\bar{x})) < 1$, where ρ is the spectral radius (eigenvalue with greatest absolute value).
- If additionally, ϕ is a **strict contraction**, ($\|\phi(x) - \phi(y)\| \leq \Theta \|x - y\|, \quad \Theta \in (0,1)$), then it will converge to a **unique** fixed point. This condition is sufficient, but not necessary.

With the two previous conditions fulfilled, the error is bounded by:

$$\|\bar{x} - x_k\| \leq \frac{\Theta}{1-\Theta}\|x_k - x_{k-1}\| \leq \frac{\Theta^k}{1-\Theta}\|x_1 - x_0\|$$

When do we stop? We stablish a tolerance:

$$\frac{\Theta}{1-\Theta}\|x_k - x_{k-1}\| < TOL,$$

and we approximate Θ as

$$\Theta \approx \tilde{\Theta}_k = \frac{\|\phi(x_k) - \phi(x_{k-1})\|}{\|x_k - x_{k-1}\|}.$$

It does not converge if $\tilde{\Theta}_k \gg 1$.

6.3.1 Speed of Convergence

- **Linear:** $\|x_{k+1} - x_k\| \leq \theta\|x_k - x_{k-1}\|$, with $\theta < 1$. Each step provides $-\log_{10}\theta$ additional correct decimal places.
- **Order** p : $\|x_{k+1} - x_k\| \leq C\|x_k - x_{k-1}\|^p$, with $C > 0$. Each step multiplies by p the number of correct decimal places.

As floating point numbers are limited to 16 digit accuracy, the advantage of a cubic convergence vs. a quadratic convergence is not much. Starting with a 1-digit accuracy, if we want maximal accuracy, we would need 4 steps with a quadratic scheme and 3 steps with a cubic scheme.

6.4 Newton's Method

Newton's method aims to solve the equation $g(x) = 0$. As in the abstract fixed point iteration, this is equivalent to solve $\nabla f(x) = 0$ which makes $g : \mathbb{R}^d \to \mathbb{R}^d$. The idea behind Newton's method is to use a Taylor expansion to approximate a function as a linear function and project it to get an approximation of its intersection with zero. Mathematically:

$$0 = g(x_k) + g'(x_k)\Delta x_k$$

As an algorithm:

1. solve $Dg(x_k)\Delta x_k = -g(x_k)$ for Δx_k
2. set $x_{k+1} = x_k + \Delta x_k$

It converges **locally** and in a **quadratic** fashion.

6.4.1 Damped Newton's Method

The fact that Newton's method has no step-size strategy can lead to a lack of convergence. Damped Newton's Method adds a strategy to overcome this. As an algorithm:

1. solve $Dg(x_k)\Delta x_k = -g(x_k)$ for Δx_k
2. choose a step-size h_k such that $f(x_k + h_k\Delta x_k) < f(x_k)$
3. set $x_{k+1} = x_k + h_k\Delta x_k$

6.4.2 Quasi-Newton Methods

The first step of the two previous algorithms implies the computation of the matrix $Dg(x_k) \in \mathbb{R}^{d\times d}$, and solving a $d \times d$ linear system. This is expensive ($\mathcal{O}(d^3)$). To overcome this, we could approximate $Dg(x_k)$ with another matrix that has an explicit inverse.

The **Quasi-Newton Method** defines an approximation matrix H_k, and updates it with a rank-1 matrix. The algorithm is:

1. compute $\Delta x_k = -H_k^{-1} g(x_k)$
2. choose a step-size h_k s.t. $f(x_k + h_k \Delta x_k) < f(x_k)$, set $s_k = h_k \Delta x_k$
3. set $x_{k+1} = x_k + s_k$, and $y_k = g(x_{k+1}) - g(x_k)$
4. set $H_{k+1}^{-1} = H_k^{-1} - \frac{H_k^{-1} u v^T H_k^{-1}}{1+v^T H_k^{-1} u}$, where

$$v = y_k - H_k s_k,$$

and

$$u = \frac{v}{\langle v | s_k \rangle}$$

The **BFGS Method** defines the inverse of the approximation matrix to be $B_k = H_k^{-1}$, and updates it with a rank-2 matrix. It converges superlinearly (faster than linear but slower than quadratic). The algorithm is:

1. compute $\Delta x_k = -B_k g(x_k)$
2. choose a step-size h_k and set $s_k = h_k \Delta x_k$
3. set $x_{k+1} = x_k + s_k$, and $y_k = g(x_{k+1}) - g(x_k)$
4. compute $B_{k+1} = B_k + \frac{s_k^T y_k + y_k^T B_k y_k}{(s_k^T y_k)^2}(s_k s_k^T) - \frac{B_k y_k s_k^T + s_k y_k^T B_k}{s_k^T y_k}$

6.5 Constrained Optimization

The optimization problem

$$\min_{x\in X} f(x)$$

can be expressed as

$$\min_{x\in\mathbb{R}^d} f(x), \quad h(x) = 0, \quad g(x) \geq 0$$

where $h(x)$ and $g(x)$ are equality and inequality constraints respectively.

A constraint is said to be **active** if it is satisfied with equality, i.e. $h_i(x) = 0$ or $g_j(x) = 0$.

6.5.1 Optimality Conditions:

To assess if x^* is a local minimum we have to satisfy the following two conditions:

1. $\nabla f(x^*)(x - x^*) \geq 0 \quad \forall x \in X$ (because it means that $f(x) < f(x^*)$ in the Taylor expansion $f(x) = f(x^*) + \nabla f(x^*)(x - x^*) + \mathcal{O}(x - x^*)^2)$
2. x^* is a KKT-point, which means that the gradients of the active constraints are linearly independent.

6.5.2 Penalty method

One idea to minimize the function f while validating the restrictions is to include the constraints in the minimized function as a penalty. The minimizing function would be:

$$P_\alpha(x) \coloneqq f(x) + \frac{\alpha}{2}\left(\|h(x)\|^2 + \|\max\{0, -g(x)\}\|^2\right)$$

For large values of α the constrains would become more problematic. Then:

$$\lim_{\alpha \to \infty} x_\alpha^* = x^*$$

As an algorithm:
while $\|\nabla P_\alpha(x_k)\|$ <TOL

1. Compute x_{k+1} such that $P_\alpha(x_{k+1})$ is minimal
2. set $\alpha = 10\alpha$, $k = k + 1$

6.6 Derivatives

All methods of finding minimums rely on knowing the derivative of the given function. This derivative could be complicated to compute or not known. One way to avoid computing the exact expression for $f'(x)$ is to use **forward finite differences:**

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

The error of this is $\mathcal{O}(h)$. For **backward** finite differences the error is linear too. However, for **central** finite differences the error is $\mathcal{O}(h^2)$.

It is possible to derive finite difference formulas of **arbitrary order** via interpolating polynomials.

6.6.1 Automatic Differentiation

Derivatives can be very expensive to compute. The aim of automatic differentiation is to split a given function f into more manageable functions, thus f and f' can be evaluated simultaneously.

This idea can be implemented by **differential numbers**. They are pairs of real or complex numbers, $dx = (x, x')$, with special arithmetic derived from the standard rules of differentiation. In the computer, this is implemented by overloading the standard number arithmetic.