Norwegian University of Science and Technology
Department of Mathematical Sciences

MA2501

Deadline March 31, 2020

You can work in groups of maximum 3 persons.

Practical information

- The project counts for 15% of the final mark. If you don't hand it in, you may still take the exam, but you cannot achieve more than 85% in the course.
- You may work on the project either alone or in groups of maximum three.
- You should produce a short report with your solutions in a pdf-file, preferably using LATEX. The report should preferably have no more than 6 pages. Students working alone, should not write more than six pages. Students working in groups can write one or tow pages more.
- A major part of the project will be the implementation (and testing) of codes in Python. A Jupiter notebook file with the code will also have to be handed in. Make sure that your code contains a meaningful documentation, in particular a reasonable number of comments in the code. Also test the programs before submission and comment on all strange behaviour of the programs in your report.
- When you present numerical results, these should be reproducible. This means that you will have to provide all relevant parameters.
- The report and the code should be submitted in Blackboard. The deadline is Tuesday, March 31, 2020.
- There will be three exercise sessions devoted to the supervision of the project and in addition a full week of work with no lectures dedicated to this project (March 23 to 27). You are welcome to ask questions about the project, please send an e-mail to agree on a time if you want to be sure that we are available.

Approximation of functions

Approximation theory is at the core of applied mathematics. Many tasks in scientific computing and data analysis are hinging on the use of methods for the approximation of functions in one or in several variables. For example, numerical methods for quadrature, for the numerical solution of ordinary and partial differential equations are founded on approximation methods. In the recent years machine learning algorithms have entered massively the scene of scientific computing. Machine learning provides a framework for solving scientific problems through optimisation methods when large amounts of data are available. As such machine learning is a framework for approximation of functions.

In this project the goal is to gain experience with different simple classical techniques for the approximation of functions, and with some other techniques for approximation based on optimisation which are to some extent related to machine learning.

You will have to implement different numerical methods for the interpolation of real valued functions defined on an interval $[a, b]$ of the real line. Your findings should be described in a report. Your codes should be well documented and submitted as a Jupiter note-book file for evaluation.

Univariate interpolation

Problem 1

We consider a continuous function $f : [a, b] \to \mathbb{R}$ which we want to approximate with different methods.

a) Implement the Lagrange interpolation polynomial for an arbitrary set of distinct nodes $x_0, \ldots, x_n$ and for values $y_0, \ldots, y_n$. Test you code on equidistant nodes and on Chebishev nodes considering first a function $f \in C^\infty$ defined on the interval $[-1, 1]$. Then generalise the method based on Chebishev nodes transforming the nodes to an arbitrary interval $[a, b]$ by applying the one-to-one transformation $\Psi : [-1, 1] \to [a, b]$, $\Psi(x) = \frac{b-a}{2}x + \frac{b+a}{2}$.

Write your code as a Python function.

Input:

- The data that we want to interpolate: x_i and y_i, $i = 0, \ldots, n$.
- The value of x where we want to evaluate the interpolation polynomial, x can be an array.

Output: the value of the interpolation polynomial in x.

Your answer to this point should include a plot of the interpolation of the Runge function $f(x) = 1/(x^2 + 1)$ with equidistant nodes and Chebishev nodes on the interval $[-5, 5]$ with $n = 10$.

b) Interpolation on both equidistant nodes and Chebishev nodes should converge in both the max-norm and 2-norm for the following functions:

- $f(x) = \cos(2\pi x)$, $x \in [0, 1]$;
- $f(x) = e^{3x} \sin(2x)$, $x \in [0, \pi/4]$.

Of course on the interpolation nodes the error is zero, however on other points of the interval $[a, b]$ the error might be small but is not zero. In order to check convergence with a numerical experiment, we want to estimate $\|f - p_n\|$ in the max-norm and in the 2-norm for different values of n. Consider a grid which has 100 times as many points as the interpolation polynomial: $x_0 = \eta_0 < \eta_1 < \cdots < \eta_N = x_n$ with N "large", e.g. $N = 100 n_{\max}$, where $n_{\max}$ is the largest degree of the interpolation polynomial that you consider in your numerical experiments. Compute the following approximations of the max-norm and of the 2-norm respectively

$$\|f - p_n\|_\infty \approx \max_{\eta_0,\ldots,\eta_N} |f(\eta_i) - p_n(\eta_i)|, \qquad \|f - p_n\|_2 \approx \frac{\sqrt{b-a}}{\sqrt{N}} \left(\sum_{i=0}^{N} (f(\eta_i) - p_n(\eta_i))^2 \right)^{\frac{1}{2}}.$$

Make a plot of the estimated error $\|p_n - f\|$ as a function of n to check convergence. Use the approximations of $\|\cdot\|_\infty$ and $\|\cdot\|_2$ given above.

Consider $f(x) = \cos(2\pi x)$, $x \in [0, 1]$. Can you use the well known error bound for the interpolation polynomial, derive a bound for the max-norm of each of the derivatives of f on $[0, 1]$, and derive an error bound for the interpolation error valid for all n for this particular function? Plot the error bound as a function of n and compare with the numerical estimates of the norm of the error that you have computed earlier.

Your answer should include two plots one for each of the two functions with the estimated error in the max and 2-norm, one of the plots should include also the the derived error bound as a function of n.

c) Piecewise-polynomial approximation

Subdivide the interval $[a, b]$ in the disjoint union of K subintervals $a = v_0 < v_1 < \cdots < v_K = b$. Implement now a method that performs Lagrangian interpolation on $n + 1$ nodes on each subinterval $[v_i, v_{i+1}]$. Use equidistant nodes on the subintervals. Fix $n = 1, 2, \ldots, 10$. Make a plot of the interpolation error in the max-norm as a function of K, give numerical evidence that the method converges as $K \to \infty$. Produce a plot of the max-norm of the error as a function of K.

If you have implemented the interpolation polynomial in part **a)** and **b)** in a wise way, you should be able to reuse that function here for the interpolation on each subinterval.

Discuss comparison of the results with the polynomial interpolation of the previous question. Use the same test function for comparison. Produce a plot of the error in max-norm as a function of the number of discretization points. What are the advantages and the disadvantages of this and of the previous methods? Which one would you recommend to use in which situation? Discuss computational cost for the various methods.

d) Optimal distribution of the nodes. Assume now that the function f is known on $N + 1$ points, $\xi_0, \ldots, \xi_N$ in the interval $[a, b]$ including the endpoints, e.g. equidistant points and with a large value of N, say $N = 1000$.

Fixed the degree of the interpolation polynomial to be equal to n, and fixed a function f that we want to interpolate, we seek for an optimal distribution of the $n+1$ interpolation

nodes $x_0, \ldots, x_n$ on $[a, b]$ minimising the error of the interpolation polynomial in the 2-norm. To this purpose we solve the following optimisation problem

$$\min_{x_0,\ldots,x_n \in [a,b]} \frac{b-a}{N} \sum_{k=0}^{N} (f(\xi_k) - p_n(\xi_k))^2 \quad \text{subject to} \quad p_n(x_j) = f(x_j),\ j = 0, \ldots, n.$$

Problems like this are *least squares* problems. You should solve the problem with an optimisation algorithm based on gradient descent, see the appendix for a quick review of gradient descent. This is an approach often used in machine learning.

Let $\mathbf{x} = [x_0, \ldots, x_n]$ be a vector containing the nodes of the interpolation. Here the cost function that we want to mimimise is $\mathcal{C} : \mathbb{R}^{n+1} \to \mathbb{R}$,

$$\mathcal{C}(\mathbf{x}) := \frac{b-a}{N} \sum_{k=0}^{N} (f(\xi_k) - p_n(\xi_k))^2, \qquad p_n(\xi) = \sum_{i=0}^{n} \ell_i(\xi) f(x_i), \quad \ell_i(\xi) = \prod_{j=0, j\neq i}^{n} \frac{\xi - x_j}{x_i - x_j}, \tag{1}$$

where $c_k := f(\xi_k)$ are known values, $p_n(\xi)$ is the interpolation polynomial in Lagrange form and $\ell_i(\xi)$ are the Lagrange basis functions. So the cost function depends on the interpolation nodes $\mathbf{x}$ through p_n. Initialize the gradient descent iteration with equidistant nodes: $x_j = a + j(b-a)/n$, $j = 0, \ldots, n$.

How to proceed:

- You need to write a function in Python implementing the cost function (1) taking a set of nodes $\mathbf{x}$ in input and returning the value $\mathcal{C}(\mathbf{x})$.
- It is convenient that this function takes in input only $\mathbf{x}$, because then the automatic differentiation routine will compute the gradient with respect to $\mathbf{x}$ and nothing else. However, $\mathcal{C}$ will need to have access to the values $c_k = f(\xi_k)$, $k = 0, \ldots, N$, the data that you generate at the start of your experiments. You will have to make sure that the function you write in Python to implement $\mathcal{C}(\mathbf{x})$ has access to the correct data and to the necessary information.
- You can then use `autograd` in Python to compute the gradient $\nabla\mathcal{C}$.
- Finally you will implement the gradient descent iteration with a stopping criterion using the computed gradient. Use also a maximum number of iterations to prevent your code to run for a very long time.

Plot the convergence history of the gradient descent iteration to see that indeed the cost function decreases as the number of iterations increases.

Compare the results you obtain against the exact value of the function, and against the interpolation methods of the previous point: compute the optimal nodes for different, increasing values of n, then compare the 2-norm of the error with the error of the interpolation on equidistant nodes and on Chebishev nodes as functions of n. You can make a table with the results. Comment the results you obtain. How is the size of N affecting your results?

e) We will now use the same method as above but replace the interpolation polynomial with a completely different method for approximating f:

$$f(x) \approx \tilde{f}(x) = \sum_{i=0}^{n} w_i \phi(|x - x_i|), \quad \phi(r) = \exp(-(\varepsilon r)^2),$$

this is an example of a radial basis function (RBF), and ε is the shape parameter. Imposing interpolation conditions

$$\tilde{f}(x_i) = f(x_i), \quad i = 0, \ldots, n,$$

one finds that the values $\mathbf{w} = [w_0, \ldots, w_n]^T$ are obtained solving the linear system

$$M\mathbf{w} = \mathbf{f},$$

where $\mathbf{f} := [f(x_0), \ldots, f(x_n)]^T$, and M is an $(n+1) \times (n+1)$ matrix with entries

$$M_{i,j} := \phi(|x_i - x_j|),$$

so the weights $\mathbf{w}$ are functions of the nodes $\mathbf{x}$.

In this task we consider the cost function

$$\mathcal{C}([\mathbf{x}, \varepsilon]) := \frac{b-a}{N} \sum_{k=0}^{N} (f(\xi_k) - \tilde{f}(\xi_k))^2, \qquad \tilde{f}(\xi) = \sum_{i=0}^{n} w_i \phi(|\xi - x_i|), \quad \mathbf{w} = M^{-1}\mathbf{f}, \quad (2)$$

and use gradient descent to optimise over the location of the nodes and the shape parameter ε. You should expect to get an optimisation problem more difficult to solve compared to the one of the previous task (in practice this means that the gradient descent will require a higher number of iterations to produce a reasonable approximation to an optimum). Also be aware that gradient descent produces local optima.

For your experiments use the Runge function on $[-1, 1]$ and the function

$$f(x) = \frac{3}{4}\left(e^{-(9x-2)^2/4} + e^{-(9x+1)^2/49}\right) + \frac{1}{2}e^{-(9x-7)^2/4} - \frac{1}{10}e^{-(9x-4)^2} \quad (3)$$

on $[-1, 1]$.

Implement this method following similar principles as for task **d)**. Compute the optimal nodes for different, increasing values of n, then compare the 2-norm of the error with the error of the RBF interpolation on equidistant nodes (and ε not optimised) as functions of n. As starting values for the nodes you can use equidistant nodes.

APPENDIX

Gradient Descent

Assume we want to minimise the function $\mathcal{C} : \mathbb{R}^m \to \mathbb{R}$ which we assume to be continuously differentiable. A necessary condition for $\mathbf{x}^* \in \mathbb{R}^m$ to be a minimiser of the function is that

$$\nabla\mathcal{C}(\mathbf{x}^*) = 0,$$

where $\nabla\mathcal{C}$ is the gradient of $\mathcal{C}$. An algorithm for finding mimima of $\mathcal{C}$ can be therefore obtained simply searching for zeros of the function $\nabla\mathcal{C}$.

The gradient descent algorithm is a fixed point itearation to solve $\nabla\mathcal{C} = 0$ of the following form

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \tau\, \nabla\mathcal{C}(\mathbf{x}^k), \quad k = 0, 1, 2, \ldots$$

Algorithm 1 Gradient descent with backtracking.

Input: initial guess for nodes of interpolation $\mathbf{x}$ and parameter L,
hyperparameters $\overline{\rho} > 1$ and $\underline{\rho} < 1$.

compute gradient $\nabla\mathcal{C}$ of the cost function $\mathcal{C}$ using `autograd`
compute $\phi = \mathcal{C}(\mathbf{x})$
for $k = 1, \ldots$ **do**
    evaluate the gradient in $\mathbf{x}$: $\mathbf{g} = \nabla\mathcal{C}(\mathbf{x})$
    for $t = 1, \ldots$ **do**
        update: $\tilde{\mathbf{x}} = \mathbf{x} - \frac{1}{L}\mathbf{g}$
        compute $\tilde{\phi} = \mathcal{C}(\tilde{\mathbf{x}})$
        if $\tilde{\phi} \leq \phi + \langle \mathbf{g}, \tilde{\mathbf{x}} - \mathbf{x} \rangle + \frac{L}{2}\|\tilde{\mathbf{x}} - \mathbf{x}\|^2$ **then**
            accept: $\mathbf{x} = \tilde{\mathbf{x}}$, $\phi = \tilde{\phi}$, $L = \underline{\rho}L$
            break inner loop
        else reject: $L = \overline{\rho}L$

where $\tau > 0$ is a step-size parameter which can be chosen adaptively (i.e. changed at every step). The iteration is stopped when a stopping criterion is met, e.g. when $\|\mathbf{x}^k - \mathbf{x}^{k-1}\| \leq TOL$ where $TOL > 0$ is a specified tolerance, an alternative stopping criterion is $\|\nabla\mathcal{C}(\mathbf{x}^k)\| \leq TOL$. Use small values of τ, e.g. $\tau = 0.1$ or $\tau = 0.01$. To improve the performance of the gradient descent you might consider using a *backtracking* strategy, i.e. a simple strategy to select τ at each step that guarantees that the cost function is reduced at every iteration.

Backtraking for gradient descent

This is a strategy that allows you to chose the step-size of the gradient descent adaptively at each iteration and in such a way that the cost function is guaranteed to decrease of a certain amount at every step.

Use equidistant nodes as initial $\mathbf{x}$, use $L = 1, 10, 100$ as initial L.