Numerical methods Project on the approximation of functions

Preben Hast Sørli

April 6, 2020

1 Intro

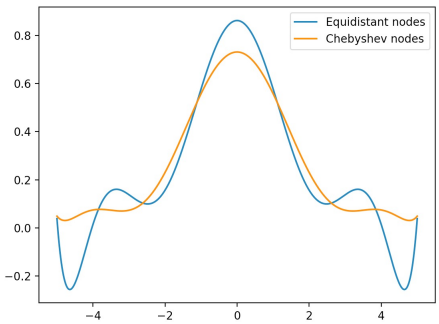
The gradient descent algorithms implemented in problem d) and e) and a lot of the plotting required for the earlier problems as well, required a large amount of computation time and my computer barely tolerated running gradient descent once at a time. I therefore logged onto markov, NTNU's server for calculation, giving me the ability to open multiple jupyter notebooks at once, without stressing my own computer. It did not seem like this gave me any substantial improvements in the running time of my algorithms, but the practicality and reliability of markov made it easy to test different parameters for the time consuming gradient descent algorithms.

plotte cost mot epsilon?

2 Problem a)

The first problem creates a foundation for the entire project in that it demands the implementation of the Lagrange interpolation polynomial. There's multiple ways to do this, each with their own advantages. In the pursuit of a computationally effective and versatile implementation, I decided to try and calculate the Lagrange polynomial in a way that evaluation in x boiled down to simply, well evaluating a polynomial in x. By implementing the interpolation this way, I only needed to calculate the interpolated polynomial again every time I change my nodes, not evey time I want to evaluate in some value x. To accomplish this, I found the numpy function "poly1d", taking an array of coefficients as input and outputting the relevant polynomial P an object we could evaluate in x. This way the interpolation only happens once, and evaluating the interpolated polynomial in an arbitrary value x by writing $P(x)$.
This implementation, LaGrange() in the code, turned out to have some disadvantages, namely an apparent lack of precision when using more than 20 nodes and bad synergy with the autograd-package. As a result I decided to write a new implementation, LaGrange2() in the code, redoing the calculation of the interpolated polynomial for every value of x we want to evaluate our interpolated polynomial in. Of course the interpolated polynomial only changes when changing the nodes, but now we never actually calculates the polynomial coefficients by itself, but its value in a concrete value for x over and over again for different x.

Figure 1: Plot of the Runge function



The problem description asks for a Python function taking interpolating nodes $\{(x_i)_{i\in\mathbb{N}}\}$ and the corresponding values $\{f(x_i)_{i\in\mathbb{N}}\}$ as inputs along with the value of x where we want to evaluate the interpolated polynomial. My implementation takes the function, f, which we want to approximate as input, technically assuming it's known and thereby in conflict with the problem description, but our Python function only ever evaluate f in the nodes x_i, which in practice should make my implementation equivalent to only knowing the function values in the nodes as there would be little to no computational difference between my implementation and simply evaluating the function in the nodes beforehand.
To demonstrate my implementation, Figure 12 shows a plot of the Runge function

$$f(x) = \frac{1}{x^2+1} \quad , \quad x \in [-5,5] \quad , \quad n = 10$$

with both equidistant nodes and Chebyshev nodes. This plot is a great display of the Runge phenomenom6, and with an increase in the degree of our interpolation polynomial we would see an exponential increase in the maximum error $max_{x\in[-5,5]}|f(x) - p_n(x)|$.

3 Problem b)

For this problem, we are asked to study to different functions, I will name them f and g:

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

To judge the quality of our Lagrange polynomials, we need to do some calculations of the approximated error in different norms. Specifically we look at the following approximations for $||f - p_n||$ in the max-norm and 2-norm, respectively:

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

where $N = 100n$ and denotes the number of known points, η_i, on the curve, with $x_0 = \eta_0 < \eta_1 < \ldots \eta_N = x_n$.

From theorem 6.2 in the book, we also get a theoretical error bound for the interpolation error:

$$|f(x) - p_n(x)| \leq \frac{M_{n+1}}{(n+1)!} |\pi_{n+1}(x)|,$$

where $\pi_{n+1}(x) = (x - x_0) \dots (x - x_n)$ and $M_{n+1} = \max_{\zeta \in [a,b]} \left| f^{(n+1)}(\zeta) \right|$. For $f(x) = \cos(2\pi x)$, observe that

$$\frac{\partial^n f(x)}{\partial x^n} = (2\pi)^n \cos\left(\frac{1}{2}\pi(n + 4x)\right) \Rightarrow \frac{\partial^{n+1} f(x)}{\partial x^{n+1}} = (2\pi)^{n+1} \cos\left(\frac{1}{2}\pi(n + 4x + 1)\right), n \in \mathbb{Z}, n \geq 0.$$

Clearly $\max_{\zeta \in [a,b]} \left| f^{(n+1)}(\zeta) \right| = (2\pi)^{n+1}$. Since we are approximating $f(x)$ on $[0, 1]$, every term in $\pi_{n+1}(x)$ will be between -1 and 1, making $|\pi_{n+1}(x)| < 1$. Putting everything together we get

$$|f(x) - p_n(x)| \leq \frac{(2\pi)^{n+1}}{(n+1)!},$$

which we will use as our theoretical error bound.

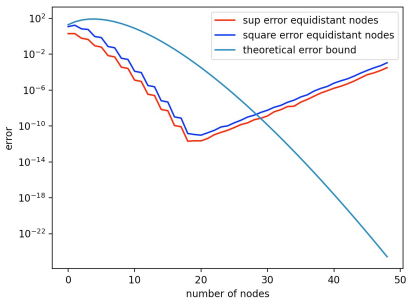


Figure 2: $f(x)$ with equidistant nodes

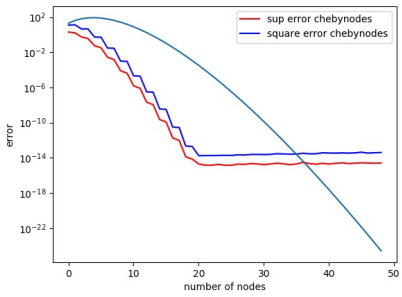


Figure 3: $f(x)$ with Chebyshev nodes

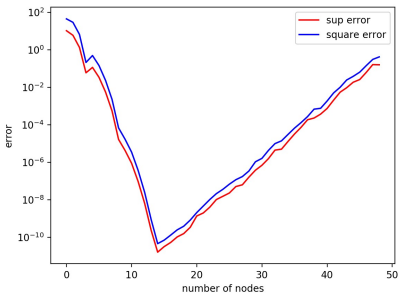


Figure 4: $g(x)$ with equidistant nodes

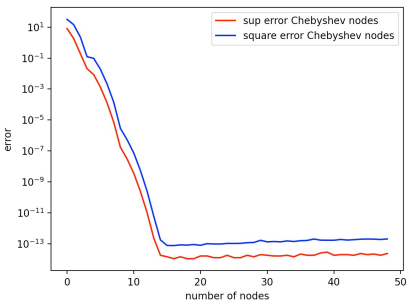


Figure 5: $g(x)$ with Chebyshev nodes

As we see from the figures, the difference between equidistant and Chebyshev nodes are not so apparent for these two functions as for the Runge function,

but something interesting happens when $n > 20$. The approximated errors start eclipsing the theoretical error bound when using equidistant nodes for both functions. To be honest I am not sure about what causes this behavior, but I suspect it's some form of propagation of error related to floating point precision in numpy.
For Chebyshev nodes the approximated error flats out at 10^{-14} presumably because of machine precision, which is around $2 * 10^{-16}$ for numpy floating numbers.

4 Problem c)

In this problem, we are subdividing the interval we are interpolating on into K subintervals, and then doing Lagrange interpolation separately on each subinterval.
I have written the Python function, piecewiseLaGrange(), which simply subdivides the interval $[a, b]$ into K smaller intervals, and then performs LaGrange interpolation on those subintervals by calling the function implemented in Problem a) on each subinterval.

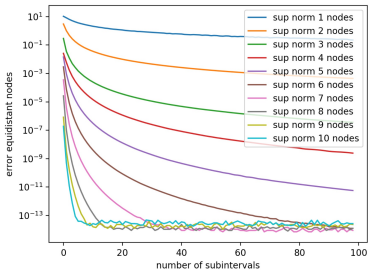


Figure 6: $g(x)$ with equidistant nodes

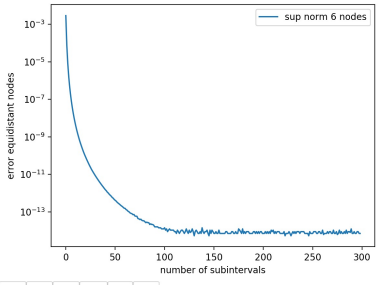


Figure 7: $g(x)$ with equidistant nodes

5 Problem d)

We will in this problem use a gradient descent algorithm to try and optimise the nodes we use for interpolation. I have implemented gradient descent pretty much exactly like it's presented in the appendix of the project description, and only use gradient descent with backtracking throughout all my experiments.
Gradient descent is an algorithm used to minimize some function by iteratively moving in the direction of the steepest descent. The function we want to minimize is the cost function:

$$\mathcal{C}(X) = \frac{b-a}{N}\sum_{k=0}^{N}\left(f(\xi_k) - p_n(\xi_k)\right)^2, \quad p_n(\xi) = \sum_{i=0}^{n}\ell_i(\xi)f\left(x_i\right), \quad \ell_i(\xi) = \prod_{j=0, j\neq i}^{n}\frac{\xi - x_j}{x_i - x_j}$$

p_n is the Lagrange interpolation polynomial as introduced earlier. We will use the Python package autograd for automatic differentiation of our cost function, and therefore it is important that we implement the cost function as a function of only one variable, namely X, our interpolation nodes. Our Python function

implementing $\mathcal{C}(X)$ still needs access to $f(\eta_i)$ for all $\eta_i \notin X$, and to accomplish this we once again assume f is known like in problem a). In the code you will find a function p(), which stores and gives $\mathcal{C}(X)$ access to all the necessary variables.
When running the gradient descent algorithm, there are three different parameters affecting its convergence. There's the step size L and two hyperparameters I've simply called hyp1 and hyp2. Through numerous experimenting with different parameters, I found $L = 100, \quad hyp1 = 2, \quad hyp2 = 0.9$ to work fine. I always initialize with equidistant nodes. To display the effect of gradient descent I plot $\mathcal{C}(X)$ versus the number of iterations in the "outer loop" of the algorithm.

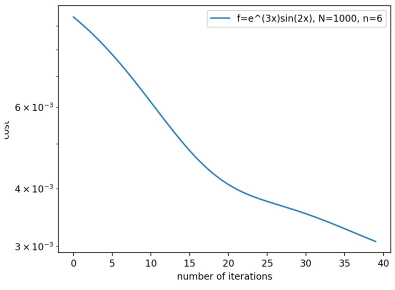


Figure 8: $g(x)$

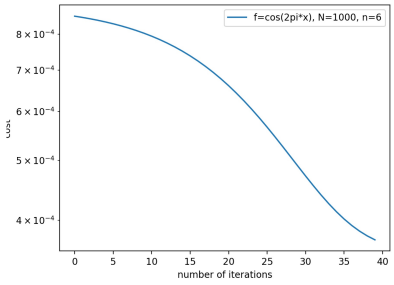
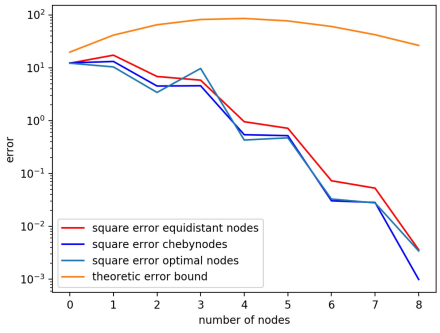


Figure 9: $f(x)$

Clearly the cost decreases as the number of iterations increases, just as desired. The inherent relationship between the cost function and the approximated square error (they are just a square apart) would also imply that the square error would decrease with the increase of iterations. Reusing the setup used in figure 2 and 3, I have plotted the square error for $f(x)$ with $n \in [1, 10]$, and equidistant and Chebyshev nodes alongside the nodes aqquired from 100 iterations of gradient descent:

Figure 10: Square error vs number of nodes



From the graph it is apparent that Chebyshev nodes already fits the function $f(x) = \cos(2\pi x)$ quite nicely as the nodes from gradient descent only seems marginally better for some n and surprisingly worse for other n. In theory, the gradient descent algorithm should be able to optimize the nodes well enough

to beat Chebyshev nodes, and some more experimenting with different parameters and a higher number of iterations would probably accomplish this. In this respect it would probably pay off to invest more time and energy into writing better and more effective code – I am certainly no expert – or even consider calculating the gradient by hand or through other methods than autograd, sacrificing convenience for speed.

6 Problem e)

Up until now we have used Lagrange interpolation for every problem. This time we are going to swap out the interpolation polynomial for a so called radial basis function which we will use to approximate f. We will use the following approximation:

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

with

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

with values $\mathbf{w} = [w_0, \ldots, w_n]^T$ the solution of the linear system $M\mathbf{w} = \mathbf{f}$, where $\mathbf{f} := \left[f\left(x_0\right), \ldots, f\left(x_n\right)\right]^T$, and M is the $(n+1 \times (n+1))$ matrix given by

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

making the weights $\mathbf{w}$ functions of X.
The cost function stays pretty much the same as before, but with the radial basis function as the approximation instead of the interpolated polynomial.

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

To make sure the gradient descent algorithm optimises over both the nodes X and the shape parameter ε, I make the cost function take as input the array $[x_0, \ldots, x_n, \varepsilon]$. We introduce one more function for approximation:

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

To demonstrate that gradient descent does what it should, I once again plot cost against number of iterations.

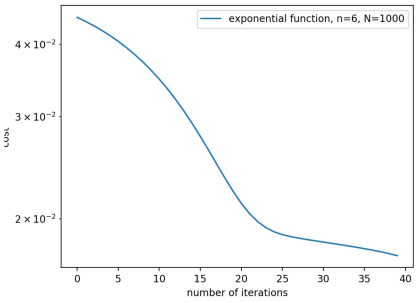


Figure 11: $h(x)$ on $[-1, 1]$

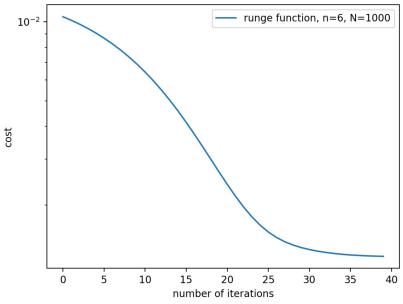


Figure 12: The Runge function on $[-1, 1]$

Comparing this plot to the one done on f and g in the preceding section, we see that the cost is roughly two orders of magnitude bigger for the same number of iterations this time, implying, as the project description describes, that this is a more difficult optimisation problem.
On the other hand, plotting the square error against the number of nodes for equidistant, Chebyshev and gradient descent optimised node reveals that the optimised nodes does much better this time around, perhaps hinting that equidistant and Chebyshev nodes are bad intial guesses for these functions. I will remark that the theory for Chebyshev nodes probably don't make the same sense for radial basis functions as it does for polynomial interpolation, but I decided to include it as a curiosity.

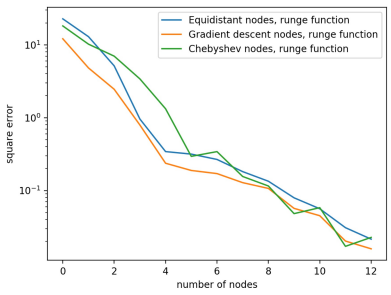


Figure 13: Runge

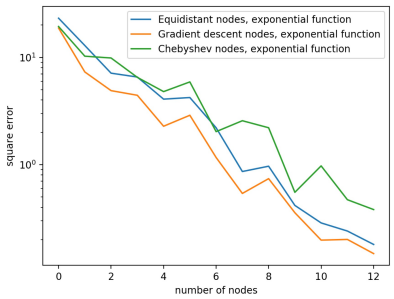


Figure 14: $h(x)$

Here I have used only 40 iterations in the outer loop and 20 in the inner loop, furthering the tendency that equidistant nodes are bad initial guesses for RBF on these two functions. The plots of cost versus iterations heavily suggests that the result would have been even better with a higher number of iterations. For the parameters I decided to stick with the ones found to work well in problem d). As for the shape parameter ε, I decided to use 3.0 after being recommended to try it by fellow students and then experiencing it to perform much better than the other values I tried. Some values even resulted in creating a singular matrix, interrupting the experiment, but this could be due to some other error in the code or perhaps a poor implementation.

nummetboka, page 186,