Jeffery Hein
MATH 126 - Dr. Barnett
Homework 2
due January 23, 2011

Problem 1. The values we are adding and subtracting are on the same order of magnitude as our machine precision. In this way, we don't expect to yield an answer with many significant digits when we perform operations on them. The order of operation then plays an important role in preserving as much accuracy as possible.

In part (a), method 1 starts by adding 1 and 3.4×10^{-16}. Due to the magnitude of difference between these values, there is significant loss of accuracy due to machine error. When we then subtract 1.1×10^{-16}, a similar problem occurs, yielding an inaccurate answer ($\approx 4.4 \times 10^{-16}$). However, the second method seeks to minimize this inaccuracy by operating on the values near machine precision first, and then adding 1. While the answer is still incorrect ($\approx 2.2 \times 10^{-16}$), it is still relatively accurate compared to the first method. Of course, the true mathematical answer is precisely 2.3×10^{-16}.

For part (b), following the same line of reasoning, we expect the second method to yield a more accurate solution (which it does). This follows by the fact that we begin by adding numbers with small order of magnitude, while introducing larger and larger values to the sum as we iterate the loop. In this way, the errors that occur by adding numbers of small magnitude become increasingly less significant as we progress in the sum. After running the code, we find that the difference between the calculated answer and the true answer is $\approx 3.06 \times 10^{-13}$ for the first method, and $\approx 8.88 \times 10^{-16}$ for the second method. This means that the error was three orders of magnitude more accurate when we added smaller values together first, than when we added larger values first.

In conclusion, this exercise illustrates the importance of performing floating-point operations on smaller order numbers first.

Problem 2. For part (a), we find that $f(x) = 2x$ via $x \oplus x$ is *backwards stable*, and therefore *stable* as well. To see this, we will show that

$$\tilde{f}(x) = f(\tilde{x}) \quad \Rightarrow \quad \Big(x(1+\epsilon_1) + x(1+\epsilon_2)\Big)(1+\epsilon_3) = 2x(1+\epsilon_4)$$

has a solution for ϵ_4 which is $\mathcal{O}(\epsilon_{\text{mach}})$. After removing the terms which are $\mathcal{O}(\epsilon^2)$ and performing some basic algebra, it is easily seen that $\epsilon_4 = 2\epsilon_3 + \epsilon_1 + \epsilon_2$ and so $|\epsilon_4| \leq 4\epsilon_{\text{mach}}$. This means that $\epsilon_4 = \mathcal{O}(\epsilon_{\text{mach}})$, and so we can find some $\tilde{x}$ near x which solves the problem exactly.

For part (b), we will show that $f(x) = 1 + x$ via $1 \oplus x$ *is stable*, yet *not backwards stable*. To see that f is not backwards stable, we look at

$$\tilde{f}(x) = f(\tilde{x}) \quad \Rightarrow \quad \Big(1 + x(1+\epsilon_1)\Big)(1+\epsilon_2) = 1 + x(1+\epsilon_3)$$

and find a solution for ϵ_3. Expanding terms and removing those which are on the order of ϵ^2, we find that

$$\epsilon_2 + x\epsilon_2 + x\epsilon_1 = x\epsilon_3.$$

This says that $\epsilon_3 = \epsilon_1 + \epsilon_2 + \frac{\epsilon_2}{x}$, and so $\epsilon_3 \neq \mathcal{O}(\epsilon_{\text{mach}})$ when we take x to be very small.

On the other hand, we claim that this is stable. To see this, we compute

$$\frac{\tilde{f}(x) - f(\tilde{x})}{f(\tilde{x})} = \frac{\big(1 + x(1+\epsilon_1)\big)(1+\epsilon_2) - \big(1 + x(1+\epsilon_3)\big)}{1 + x(1+\epsilon_3)} \tag{1.1}$$

and take its norm. Sparing the gory algebra details, we find that this is equal to

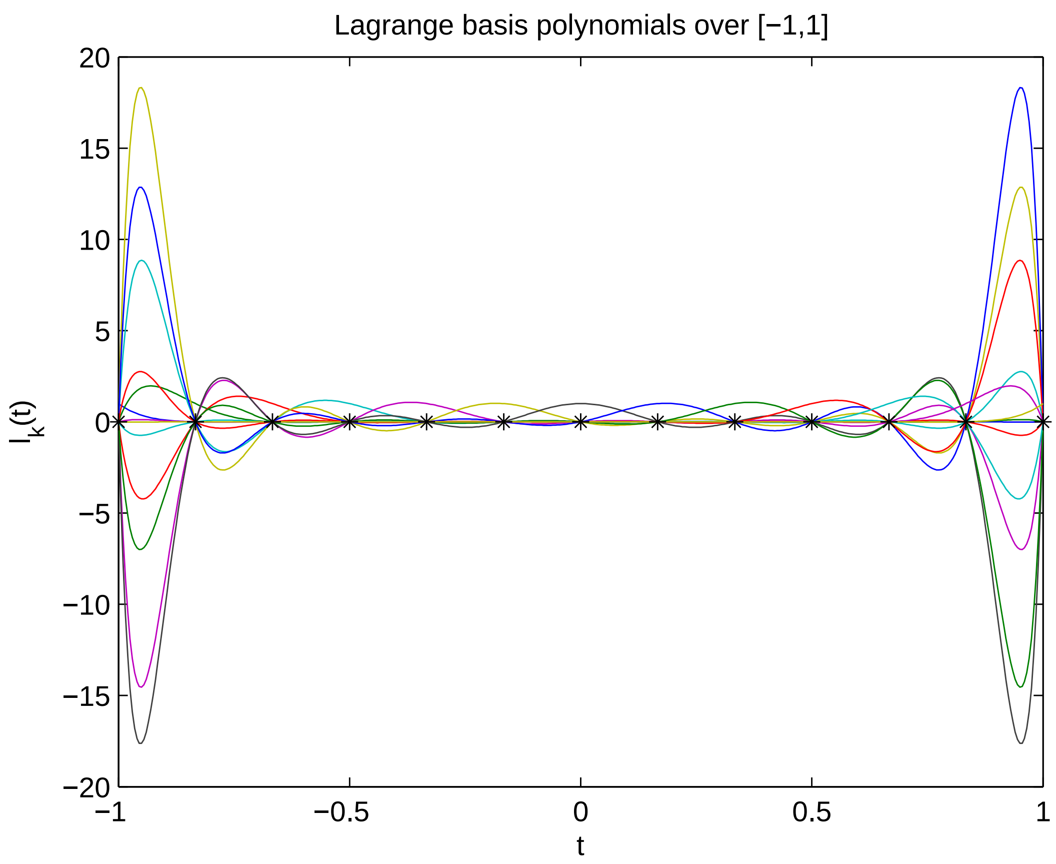
$$\left|\left|\frac{\epsilon_2 + x(\epsilon_2 + \epsilon_1 - \epsilon_3)}{1 + x(1+\epsilon_3)}\right|\right|.$$

Since $|\epsilon_i| \leq \epsilon_{\text{mach}}$, we find that as $\epsilon_{\text{mach}} \to 0$, the norm likewise goes to zero for all x. As a result, the algorithm is stable since the norm of (1.1) is $\mathcal{O}(\epsilon_{\text{mach}})$.

Problem 3. The characteristic polynomial for the identity matrix is $x^2 - 2x + 1$. When we perturb the constant coefficient in this polynomial by ϵ, we have a new polynomial given by $x^2 - 2x + (1 - \epsilon)$. Solving for the roots of this polynomial, we find them to be $1 \pm \sqrt{\epsilon}$. Since we are taking ϵ to be very small, the effect of taking a square root causes the relative difference to be amplified. In effect, perturbing the input by ϵ has caused the output to be perturbed by $\sqrt{\epsilon}$. If we were to say that the algorithm were stable, we would be saying that $\sqrt{\epsilon} = \mathcal{O}(\epsilon_{\text{mach}})$, hence there is a k such that $\sqrt{\epsilon} \leq k\epsilon_{\text{mach}}$ as $\epsilon_{\text{mach}} \to 0$. However, this is not possible, since the slope of $\sqrt{\epsilon}$ gets arbitrarily large as $\epsilon \to 0$.

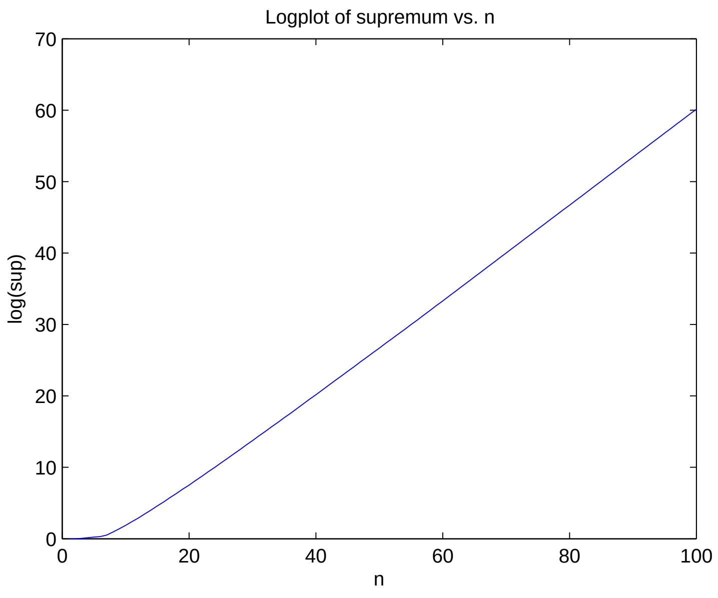
We just demonstrated that the roots of the characteristic polynomial of the identity matrix are sensitive to perturbations of its coefficients. Since this is the simplest case imaginable, it is not unreasonable to assume that this effect will only worsen as the degree of the characteristic polynomial grows. Any algorithm which attempts to find eigenvalues by looking at a characteristic polynomial will inevitably encounter $\mathcal{O}(\epsilon)$ perturbations and will therefore incur relatively large error. This error, as demonstrated above, will cause the algorithm to be unstable.

Problem 4. See "lagrange.m" and "hw2_4.m" for the MATLAB code.

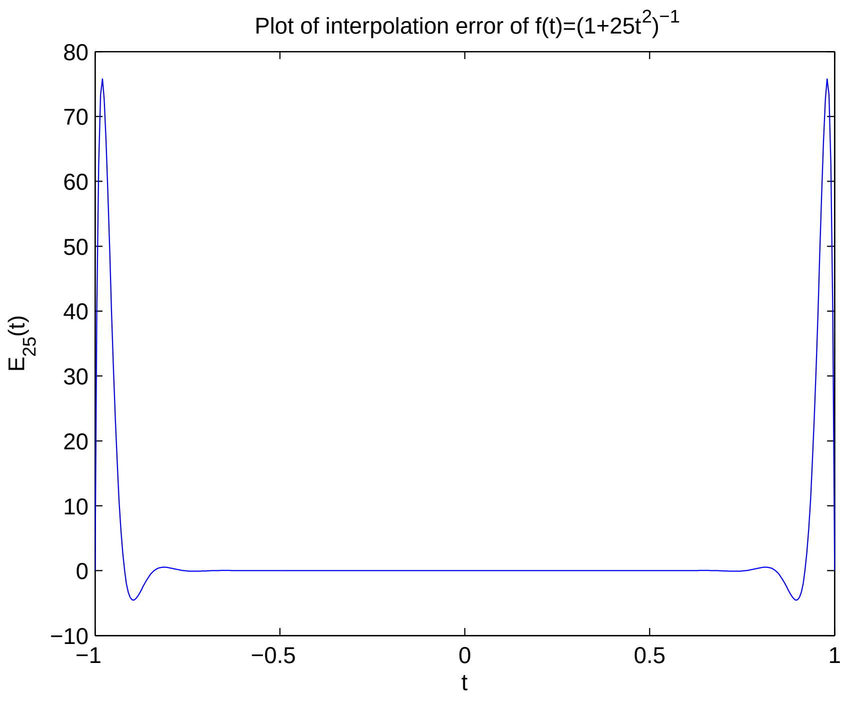


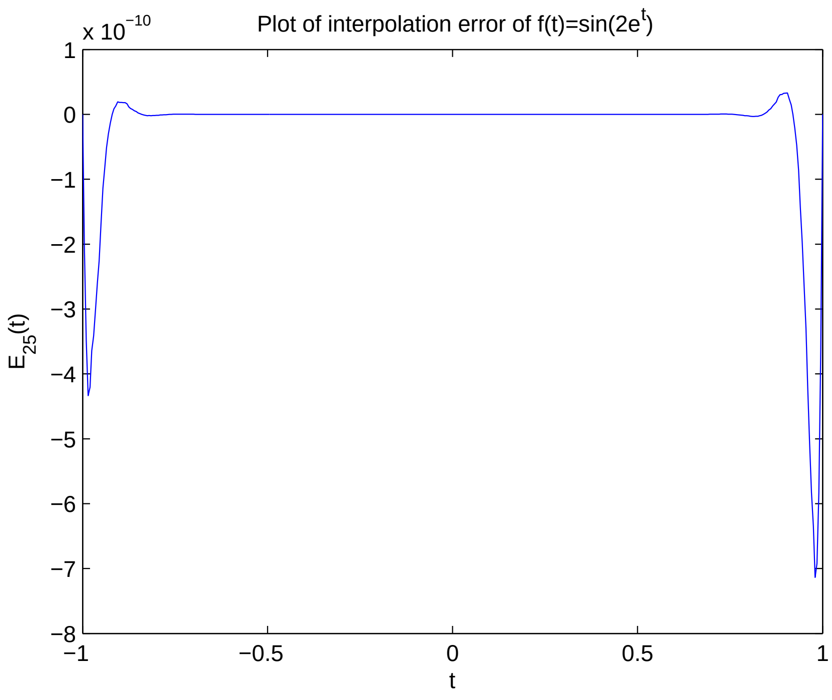
The figure above illustrates the 12 Lagrange polynomials given 12 equally-spaced nodes over the interval $[-1, 1]$.

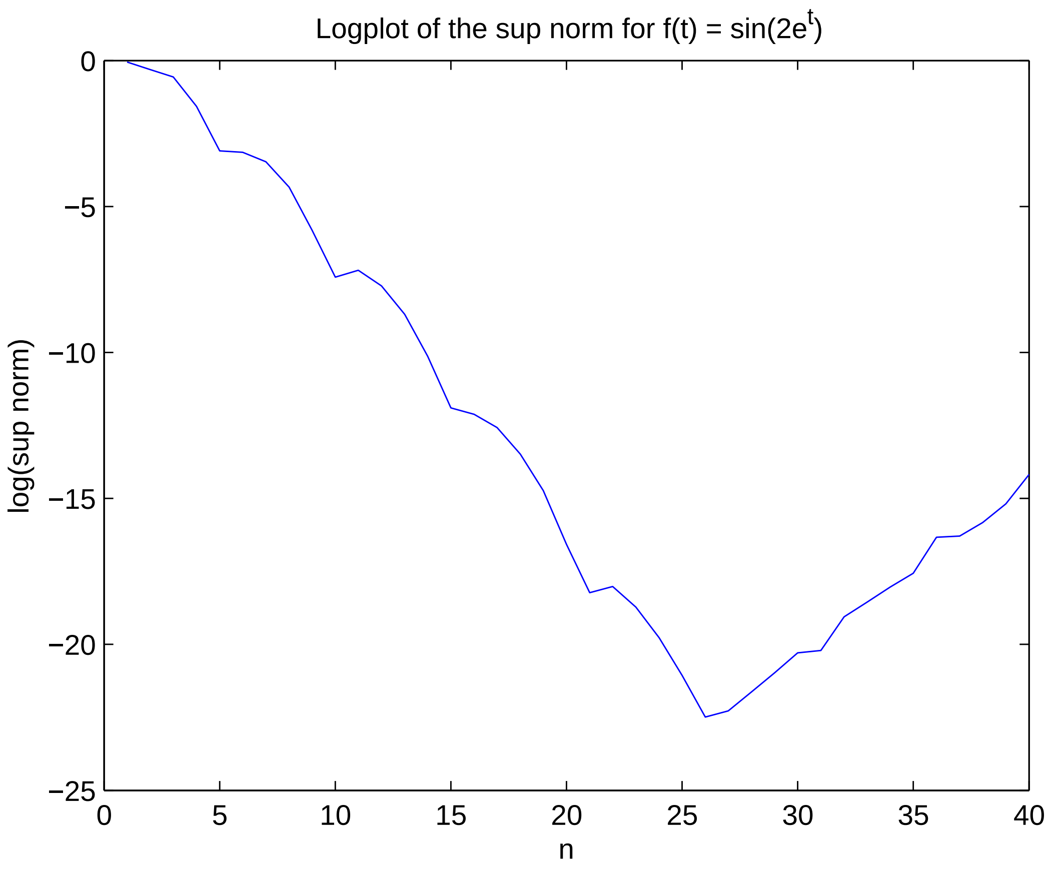
The figure to the right is the logplot of the supremum norm of all the Lagrange polynomials over the interval $[-1, 1]$. Evaluating this logplot at $n = 100$ and $n = 30$ suggests a slope of ≈ 0.6629 and so the desired supremum is growing at a rate of approximately $\mathcal{O}(e^{0.6629n})$.



Problem 5. See "hw2_5.m" for the code pertaining to this problem. The interpolation errors are plotted below.

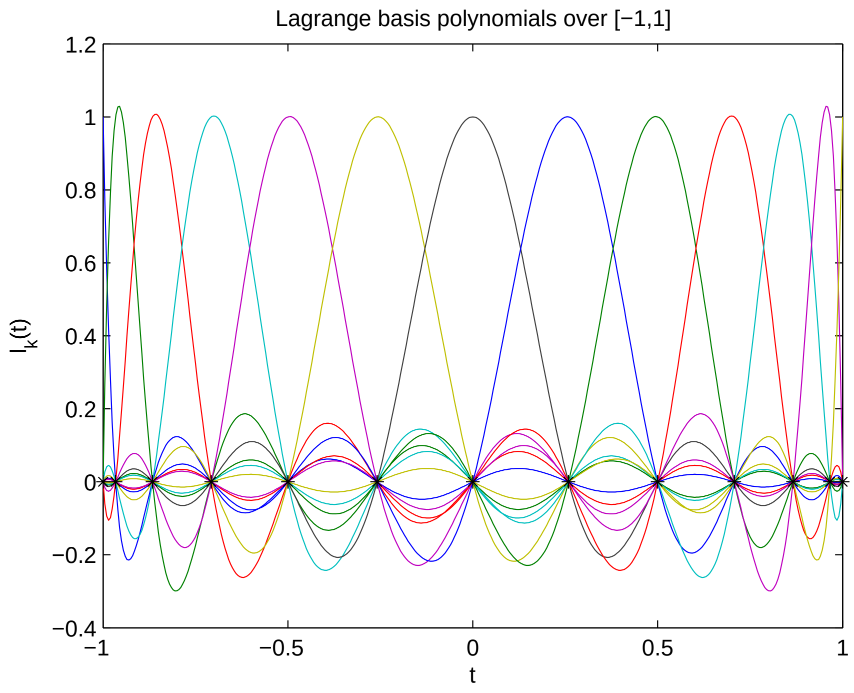




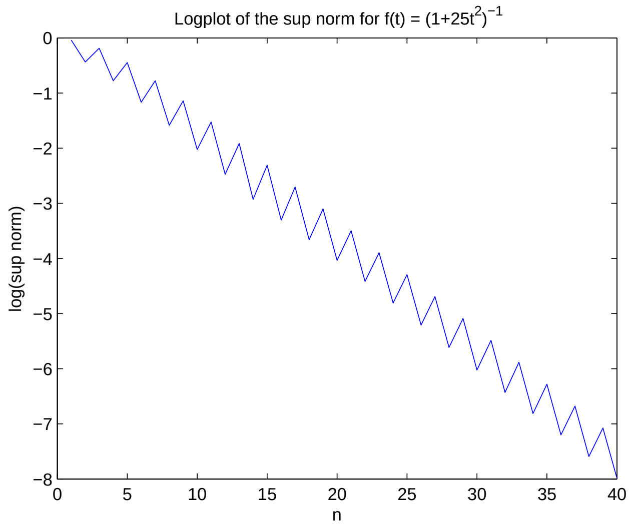


The figure above is the logplot of the supremum norm of the interpolation errors incurred by n equally-spaced nodes over $[-1, 1]$. It is evident by this plot that the supremum norm exponentially decays between 1 and 25; however, for $n > 25$, the supremum norm begins to reverse its trend and begins to grow exponentially! At the point when $n > 25$, we have effectly reached the limits of the machine precision, and so the exponential growth of the Lagrange polynomials are allowed to dominate the interpolation error supremum norm. Looking at the slope of the exponential growth for $n > 25$, we find it to be approximately between $e^{0.5}$ and $e^{0.6}$, which is suspiciously close to the growth rate we found for the Lagrange polynomials in the previous problem.

Problem 6. See "hw2_6a.m" and "hw2_6b.m". When implementing the Chebyshev nodes, we obtained the following Lagrange polynomials.

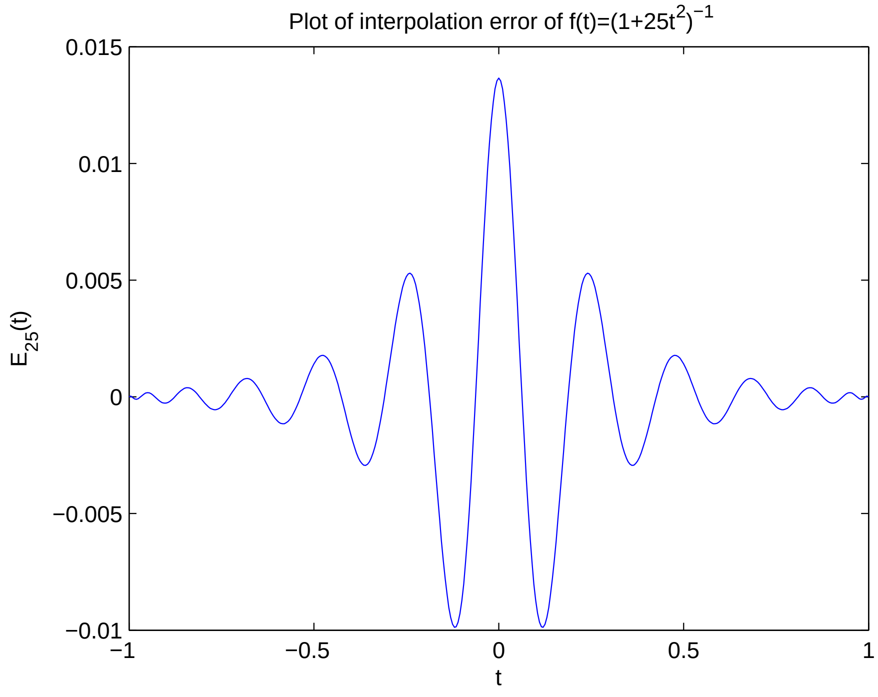


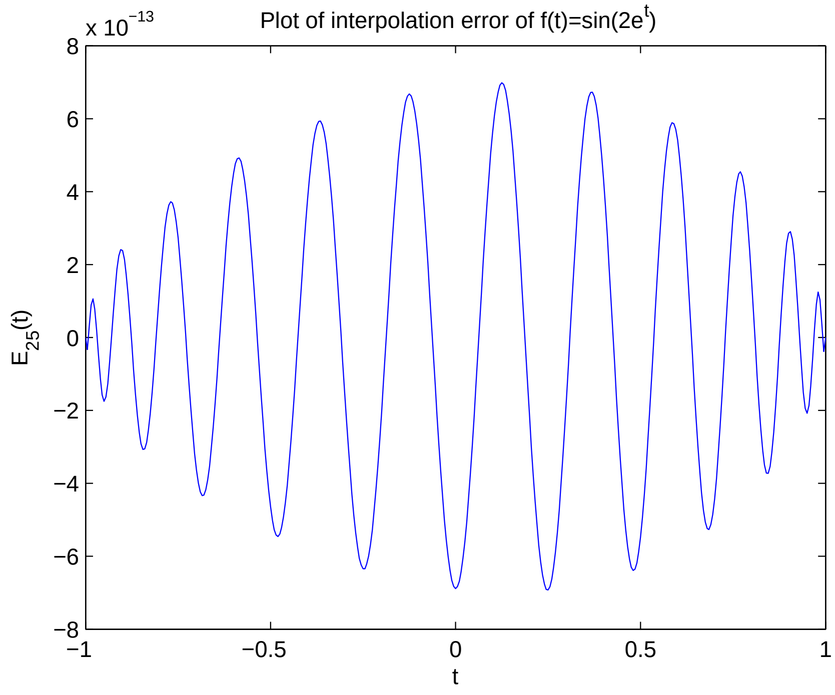
Using Chebyshev nodes to produce the Lagrange polynomials, the supremum norm is now approximately 1 for all n, where as they grew exponentially with equally-spaced nodes.



Furthermore, using these nodes, we no longer observe the Runge phenomenon with $f(t) = (1 + 25t^2)^{-1}$ as illustrated in the figure to the left; the supremum norm now decays exponentially with n.

The following page shows the interpolation error via Chebyshev nodes with $n = 25$, which have been dramatically improved in contrast to the interpolation error we plotted in Problem 4.


Plot of interpolation error of f(t)=(1+25t^2)$^{-1}$
$E_{25}(t)$
t
0.015
0.01
0.005
0
−0.005
−0.01
−1
−0.5
0
0.5
1


Plot of interpolation error of f(t)=sin(2e^t)
x 10^{-13}
$E_{25}(t)$
t
8
6
4
2
0
−2
−4
−6
−8
−1
−0.5
0
0.5
1