Polynomial Interpolation in thousands of points

Michal Porvaznik
SN: 110030149

April 3, 2013

Abstract

This project is part of coursework for the module MATH7601 at UCL.

1 Introduction

Lagrange Interpolation is a straightforward interpolation method that was first presented by Waring in 1779 [1] and in its most fundamental form was stated to be of little practical use by many authors [3]. Nevertheless, there appeared several publications showing how can the *Lagrange formula* be modified to attain properties that make it as practical, if not more, as other methods of polynomial interpolation. One of such publications is a paper by Berrut and Trefethen [2] on which, for the most part, this account relies on to further examine properties of *Barycentric Lagrange interpolation* in thousands of points.

We start by introducing the aforementioned modifications to *Lagrange formula* in section 2, after which the optimal choice of interpolation points and the *Runge phenomenon* are discussed in section 3 together with a numerical example demonstrating the theoretical results. In section 4, the convergence rates of interpolants in *Chebyshev points* for functions with different smoothness are examined using variety of numerical experiments. A summary of results on numerical stability of the *Barycentric Lagrange interpolation* will be provided in section 5, followed by a conclusion in section 6.

2 Modifying Lagrange Interpolation

2.1 Lagrange formula

Suppose that $n \geq 0$ and let x_i, for $i = 0, ..., n$, be distinct real numbers and assume that y_i, for $i = 0, ..., n$, are real numbers. If we denote by $\mathcal{P}_n$ the vector space of all polynomials with degree less than or equal to n, then the problem of polynomial interpolation is to find a unique polynomial $p_n \in \mathcal{P}_n$ such that

$$p_n(x_i) = y_i, \qquad i = 0, ..., n \tag{1}$$

Proof of existence and uniqueness of such polynomial is not difficult and can be found in [1]. Moreover, p_n can be expressed as

$$p_n(x) = \sum_{k=0}^{n} L_k(x) y_k \tag{2}$$

where $L_k(x)$, $k = 0, ...n$ is the *Lagrange interpolation polynomial* of degree n for the set of points $\{(x_i, y_i) : i = 0, ..., n\}$ [1] and can be written as

$$L_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i}$$

Upon noting that

$$L_k(x_i) = \begin{cases} 1, & \text{if } i = k \\ 0, & \text{otherwise} \end{cases} \qquad i, k = 0, ..., n$$

it is easy to see that p_n defined by (2) satisfies (1) and thus solves our interpolation problem.

However, as Berrut and Trefethen point out in [2], the Lagrange's formula (2) has several drawbacks. Firstly, evaluation of $p_n(x)$ requires $\mathcal{O}(n^2)$ operations for each x. Secondly, in case we wish to add new data (x_{n+1}, y_{n+1}) we have to repeat the calculation all over. Finally, and perhaps most importantly, the formula is numerically unstable.

Nevertheless, it does not mean that *Lagrange Interpolation* is useless in practical applications and in the two following subsections new, modified versions of (2) will be introduced that were proved to be stable by N.J. Higham [3].

2.2 Modified Lagrange formula

In this subsection we will examine modifications that were presented in [2] to address the aforementioned issues with (2). Let

$$L(x) = \prod_{i=0}^{n} (x - x_i)$$

and note that when divided by $(x - x_k)$, $L(x)$ is equivalent to numerator of $L_k(x)$. Now define

$$w_k = \frac{1}{\prod\limits_{\substack{i=0 \\ i \neq k}}^{n} (x_k - x_i)} \qquad k = 0, ..., n \tag{3}$$

and we can rewrite (2) in the following way

$$p_n(x) = L(x) \sum_{i=0}^{n} \frac{w_i}{x - x_i} y_i \tag{4}$$

We call the w_i's the *Barycentric weights* and since each w_i requires $2n+1$ (n subtractions, n multiplications and one division) floating point operations we need $\mathcal{O}(n^2)$ operations to obtain the set of constants $\{w_k\}_{k=0}^{n}$ for any given set of data after which we only need $\mathcal{O}(n)$ to calculate (4) for any given x, thus eliminating the first problem of (2) as mentioned in [4, p.33]. Furthermore, if we add another data point (x_{n+1}, y_{n+1}), we only need to divide each w_i by $x_i - x_{n+1}$ and compute w_{n+1} resulting only in $\mathcal{O}(n)$ operations to obtain updated weights addressing the second issue of (2). Finally, (4) is numerically stable, which will be discussed in more detail in section 5.

Formula (4) has been called the "first form of the barycentric interpolation formula" by Rautishauser and was first derived by Jacobi in his PhD thesis in 1825 [4, p.33].

2.3 Barycentric formula

We are now just a step away from obtaining the main formula of this paper. Suppose we want to interpolate the constant function $f(x) = 1$. Then $y_i = 1$ *for all* $i = 0, ..., n$ and since the interpolant of f is itself, (2) and (4) give

$$1 = \sum_{k=0}^{n} L_k(x) = L(x) \sum_{k=0}^{n} \frac{w_k}{x - x_k} \tag{5}$$

and by dividing (4) by (5) we arrive at

$$p_n(x) = \frac{\sum_{k=0}^{n} \frac{w_k}{x-x_k} y_k}{\sum_{k=0}^{n} \frac{w_k}{x-x_k}} \tag{6}$$

what Rautishauser called the "second (true) form of the barycentric interpolation formula" [2]. Although, (6) satisfies weaker form of numerical stability than (4) [3], which will be explained in section 5, it features the *Barycentric weights* in both numerator and denominator which means that if in some special case we can figure out the w_k's analytically, then any k-independent factors of w_k can be discarded, thus reducing the possibility of overflow or underflow to which (4) is susceptible for large n [4].

As for the computation of w_k's, the following is a python implementation of the pseudocode provided in [2]. The function *weights(x)* takes a vector x of interpolation points and returns corresponding vector w of *Barycentric weights*. Notice that this algorithm minimises number of divisions, which is done because division is a more complicated operation for a computer to execute, although this is not reflected in the usual algorithm cost analysis for simplicity reasons.

```
def weights(x):
    n = x.shape[0]
    w = np.ones(n)
    for j in range(1,n):
```

```
        for k in range(j):
            w[k] *= (x[k] - x[j])
            w[j] *= (x[j] - x[k])
    for j in range(n):
        w[j] = 1/w[j]
    return w
```

3 Interpolation points distributions

There turns out to be strong correlation between choice of interpolation points and the quality of interpolation as measured by how closely the interpolant approximates the function at points other than x_k, $k = 0, ..., n$. In particular, we will consider equispaced points and Chebyshev points, which will be defined in following subsections. However, first we shall discuss the connection between this topic and potential theory outlined by Trefethen in [5, Chap. 5], which gives some insight to why should one choose interpolation points carefully.

Let p be a monic polynomial of degree n. Then

$$p(z) = \prod_{k=0}^{n} (z - z_k)$$

where $\{z_k\}$ are roots of p that may or may not be complex. We have

$$log|p(z)| = \sum_{k=0}^{n} log|z - z_k|$$

If we now define function ϕ: $\mathbb{C} \longrightarrow \mathbb{C}$ as

$$\phi_n(z) = n^{-1} \sum_{k=0}^{n} log|z - z_k|$$

which can be interpreted as potential at z due to charges $\{z_k\}$ with corresponding potential $n^{-1}log|z - z_k|$, then

$$|p(z)| = e^{n\phi_n}$$

From here, trough some analysis that is not of great importance to this discussion, Trefethen goes on to conclude that if $\{z_k\}$ are evenly distributed on [-1,1] and we take limit as $n \to \infty$ [1] then

$$|p(x)| \simeq e^{n\phi_n} = \begin{cases} (\frac{2}{e})^n, & \text{near } |x| = 1 \\ (\frac{1}{e})^n, & \text{near } x = 0 \end{cases}$$

On the other hand, when $\{z_k\}$ are distributed as Chebyshev points then in the limit we get

$$|p(z)| \simeq e^{n\phi_n} = 2^{-n}, \qquad x \in [-1, 1]$$

[1] Taking a limit is a natural thing to do here as we are interested in behaviour of the interpolant when n gets large, e.g. thousands of points

At this point we can observe that with Chebyshev points, the interpolant will vary only slightly along [-1,1], whereas if we interpolate at equidistant points, the interpolant will be approximately 2^n times larger near the endpoints than around centre of interval.

3.1 Equidistant Points and the Runge phenomenon

The above discussion suggests that when interpolating function at equidistant points one should not expect good approximation even when increasing the number of interpolation points.

Indeed, this phenomenon was revealed in 1901 by Runge [6] and thus became known as Runge phenomenon. He showed that even for a well behaved holomorphic function such as

$$f(x) = \frac{1}{1+x^2} \tag{7}$$

the interpolant diverges exponentially. We will now demonstrate this on a numerical example.

The following python code computes the *Barycentric interpolant* of function f defined in (7) and plots it together with f on the interval [-5,5].

```
def interp(x, f, points=1000, cheb=False):
    n = x.shape[0]
    (a,b) = (min(x[j] for j in range(n)), max(x[j] for j in range(n)))
    x1 = np.linspace(a,b,points)
    numer = np.zeros(points)
    denom = np.zeros(points)
    p = np.zeros(points)
    if cheb:
        w = np.ones(n)
        for i in range(1, n-1):
            w[i] *= (-1)**i
        w[0] = 0.5
        w[n-1] = 0.5
    else:
        w = weights(x)
    for i in range(n):
        xdiff = x1 - x[i]
        if 0 in xdiff:
            for j in range(points):
                if xdiff[j] == 0:
                    xdiff[j] = 1
                    break
        div = w[i] / xdiff
        numer += div*f[i]
        denom += div
    p = numer /  denom
```

```
    for i in range(n):
        for j in range(points):
            if x[i] == x1[j]:
                p[j] = f[i]
    return (x1, p)

def runge_1(n):
    x = np.linspace(-5, 5, n)
    f = 1 / (1 + x**2)
    (x1, p) = interp(x, f, points=max(2*n, 1000))
    f1 = 1/ (1 + x1**2)
    plt.plot(x1, f1, x1, p, '--')
```

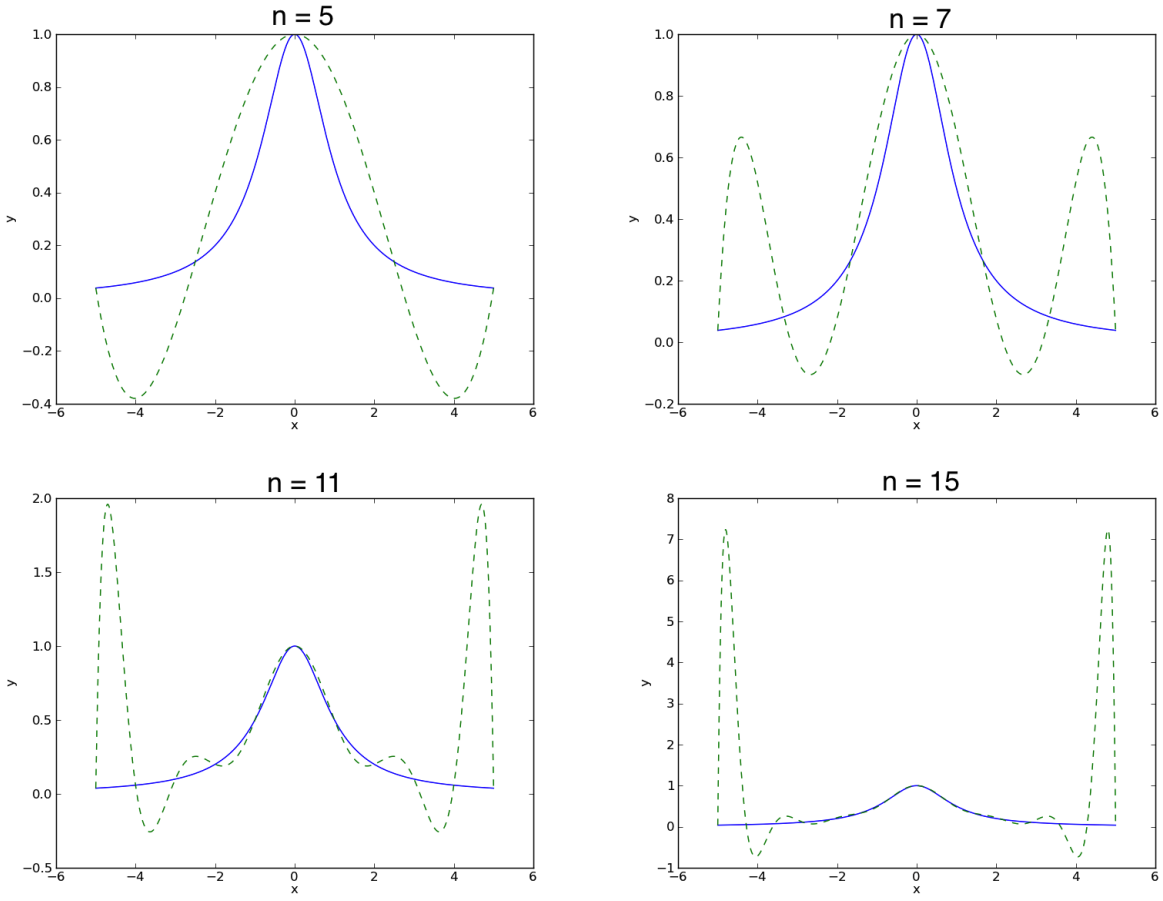

Figure 1: Plots of function f defined by (7) together with its interpolants of degree n. The solid line represents f and the dashed line is the interpolant. We can observe stronger oscillations near the endpoints. These plots were generated by runge 1(n)

Although this phenomenon does not occur with every well-behaved function interpolated at equdistant points, there is a more stable and effcient choice of points, which we will investigate now.

3.2 Chebyshev points

Chebyshev points are perhaps the most common interpolation points clustered near endpoints of the interval and can be obtained by projecting equally spaced points on the unit circle to the interval [-1,1] [2]. There are four standard types of Chebyshev points, however we shall concentrate on the *Chebychev points of the second kind* here as in this case we observe very practical form of the *Barycentric weights*. *Chebychev points of the second kind* are defined by [2]

$$x_j = cos\,\frac{j\pi}{n} \qquad j = 0, ..., n$$

and we obtain [7]

$$w_j = (-1)^j \delta, \qquad \delta_j = \begin{cases} \frac{1}{2}, & \text{if } j = 0 \text{ or } j = n \\ 1, & \text{otherwise} \end{cases}$$

One can easily see that this has several practical advantages. Firstly, no computations are needed to calculate the *Barycentric weights*, which has crucial impact on the computational run-time. Additionally, the variations between *weights* are negligibly small, thus reducing oscillations in the interpolant significantly. We will perform the same numerical example on the function defined in (7) using *Chebyshev nodes* this time to validate the theoretical superiority of this approach.

The python code below uses the same procedure *interp()* as the previous example, except for the calculation of *weights*.

```
def cheb(n):
    x = np.zeros(n)
    for i in range(n):
        x[i]= 5 * np.cos((np.pi * i) / (n-1))
    f = 1 / (1 + x**2)
    (x1,p) = interp(x, f, points=max(2*n, 1000),cheb=True)
    f1 = 1/ (1 + x1**2)
    plt.plot(x1, f1, x1, p, '--')
```

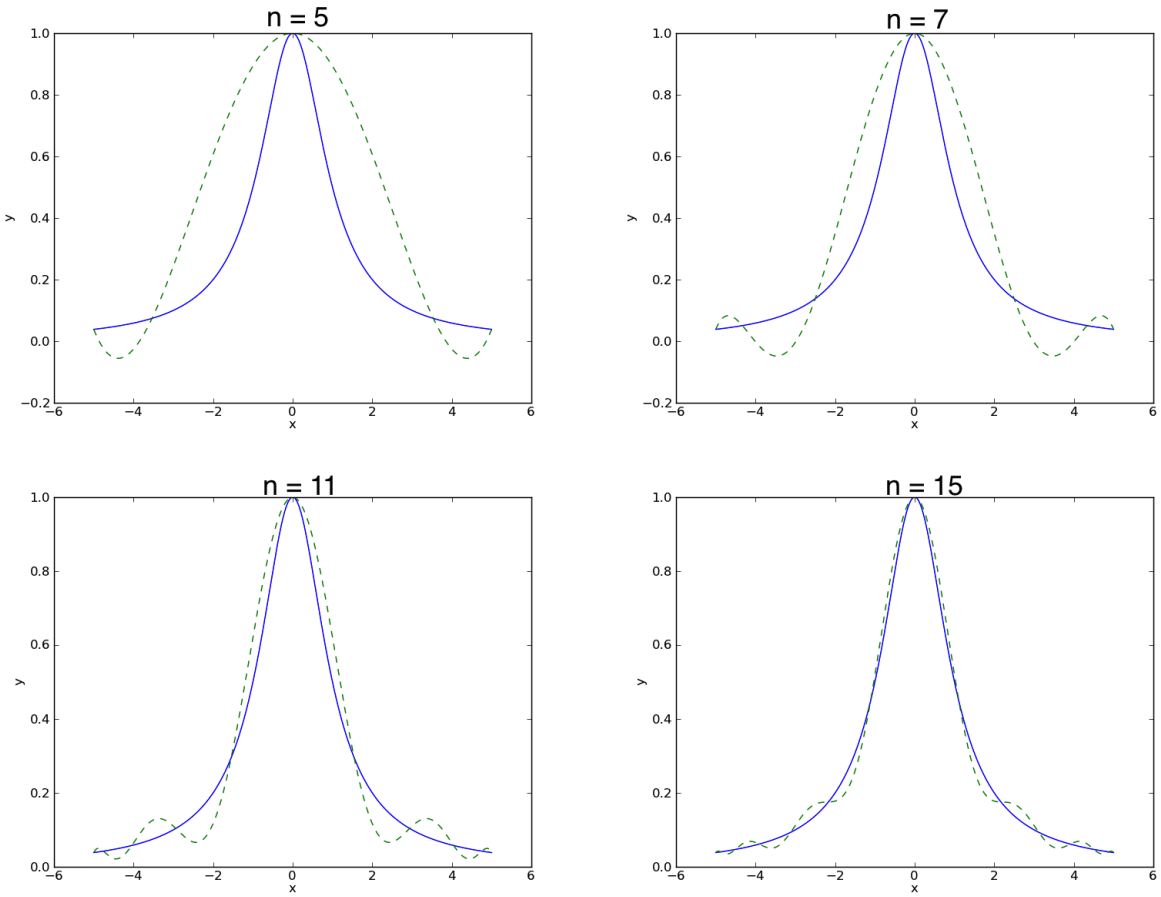

Figure 2: Plots of function f defined by (7) together with its interpolants of degree n. The solid line represents f and the dashed line is the interpolant. We can see that the interpolant approximates f increasingly well as n increases. Plots were generated by cheb(n). We could plot the same graphs for n = 10000, however the interpolant becomes indistinguishable from f soon after n = 15

Figures 1 and 2 were produced using the same algorithm for computing the interpolant as well as same numbers of interpolation points and the difference clearly confirms the theoretical predictions about quality of interpolation in terms of ability of the interpolant to approximate f as the point distributions vary. There are other point sets that converge to

$$\frac{1}{\sqrt{1-x^2}} \quad as \quad n \to \infty$$

such as the *Legendre points*, nevertheless, explicit expressions for their *Barycentric weights* are not know, thus making them less favourable for efficient applications [2].

4 Convergence

Whilst increasing number of interpolation points did not yield better approximation for equidistant points, *Figure 2* suggests that for Chebyshev points the interpolant approximates f increasingly well as n increases. We would now like to quantify these observations and examine how do the convergence rates depend on smoothness of the interpolated function. This section consists predominantly of numerical experiments.

4.1 Smooth functions

We start with functions with which we expect to observe steep convergence. Let f be a function defined on complex plain and let p be its interpolant of degree n and assume f is *holomorphic* in an ellipse with foci ± 1 and axis lengths $2a$ and $2b$. Then, as shown in [5, Chap. 5]

$$\max_{x\in[-1,1]} |f(x) - p(x)| \leq CK^{-n}$$

where $K = a + b$ and $C > 0$. This result suggests that we should observe geometric rate of convergence up to the order of rounding error for f satisfying the above assumptions.

We will now perform a numerical experiment to see if this result holds using the following *python* code.

```
def conv1(n, m, k=1):
       for i in range(n, m, k):
        x = np.zeros(i)
        for j in range(i):
            x[j]= np.cos((np.pi * j) / (i-1))
        f = 1 / (1 + 16*x**2)
        (x1, p) = interp(x, f, cheb=True)
        f1 = 1/ (1 + 16*x1**2)
        mdiff = max(abs(f1[j] - p[j]) for j in range(1000))
        plt.plot(i, mdiff, 'ko')
```

For the function f plotted in *Figure 3* we changed *conv1()* by putting

```
f = np.cos(x)
f1 = np.cos(x1)
```

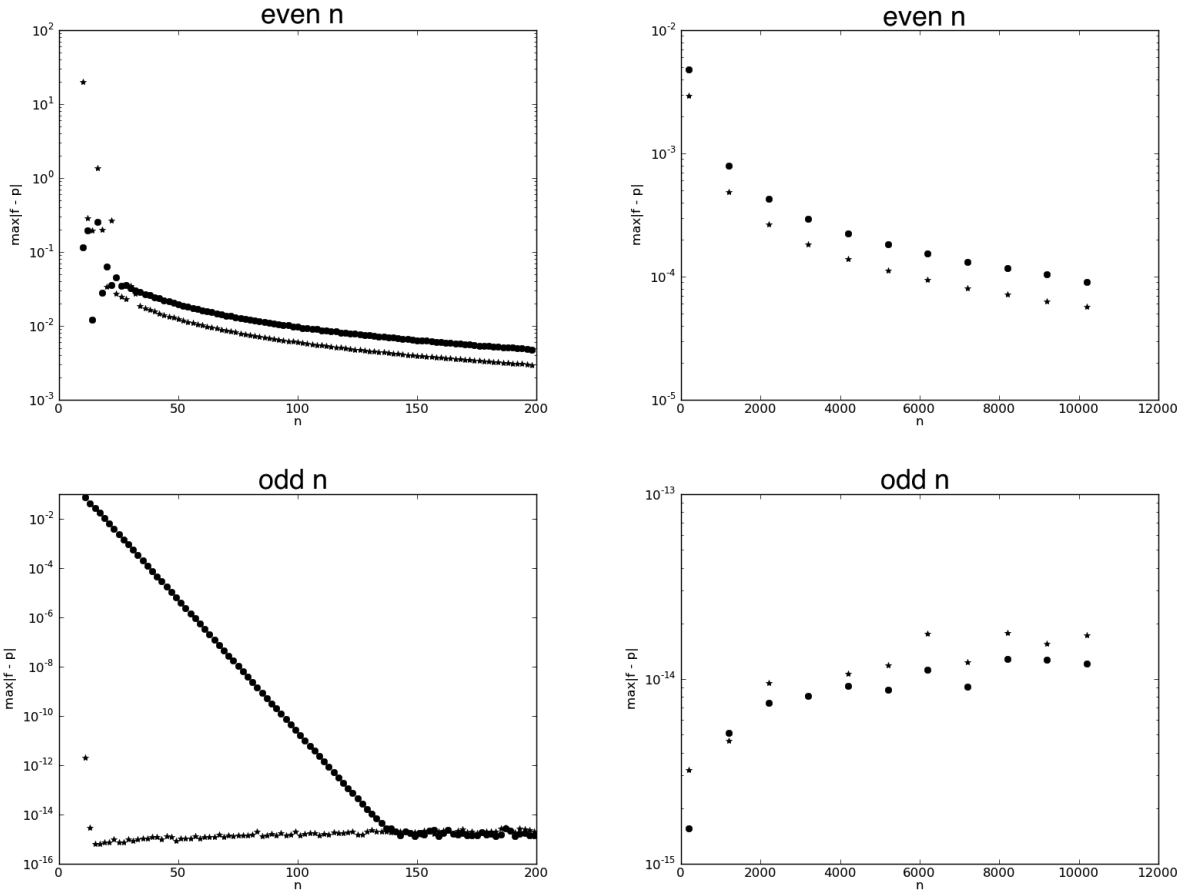

Figure 3: Plots of $\max_{x\in[-1,1]} |f(x) - p(x)|$ against number of interpolation points. The $\star$ markers correspond to the function $f(x) = cos(x)$ and the $\bullet$ markers correspond to the function $g(x) = \frac{1}{1+x^2}$. The plots are divided for readability by even and odd n because we observe significantly different convergence rates in these cases.

For odd n the experiment produced convergence rates that confirm the results above. The function f is *entire* and we see much faster convergence rate then that for the function g, which is singular at $\pm i$. For f and g the error reaches level of rounding error for $n \approx 20$ and $n \approx 140$ respectively. As this project concentrates on *interpolation in thousands of points*, plots include n as large as 10000. Although the error is reasonably small for large n, we see a slightly increasing tendency in error, suggesting we would be better off interpolating in couple hundred points. This is likely to be caused by larger errors in computation as n increases.

For even n however, the results are not coherent with the theoretical predictions. We observe very slow convergence rates, not substantially different for the two functions, achieving error as large as 10^{-4} even for $n = 10000$. Here we see strictly decreasing error with increasing n so it makes sense to take more points if we wish to compute a better approximation, nevertheless the results are far from satisfactory.

Why is there such a big difference between convergence rates for even and odd n? Does there exist some fundamental difference between interpolating by even and

odd polynomials? Neither the theory presented at the beginning of this subsection, nor the results of a very similar experiment presented by Berrut and Trefethen in [2] confirm our findings. The most likely explanation seems to be some small mistake in *python* implementation of the *Barycentric formula*[2] that I was not able to recognise and fix even after many hours of debugging.

4.2 Non-differentiable and discontinuous functions

We saw that the convergence rate for a smooth function can be surprisingly fast. Now we shall perform further experiments to examine the convergence rates for functions that are either non-differentiable or discontinuous at some point in the interval [-1,1].

Let f and g be defined as

$$f(x) = \sqrt{|x|}, \qquad g(x) = sign(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases}$$

f is non-differentiable at $x = 0$ and g is discontinuous at $x = 0$. Interpolating these functions at *Chebyshev nodes* on the interval [-1,1] we can plot the results using the *python* code bellow

```
def f(n):
    x = np.zeros(n)
    for i in range(n):
        x[i]= np.cos((np.pi * i) / (n-1))
    f = np.sqrt(abs(x))
    (x1, p) = interp(x, f, points=max(2 * n + 1, 1001), cheb=True)
    plt.plot(x1, p, '-')

def sign(x):
    n = x.shape[0]
    w = np.ones(n)
    for i in range(n):
        if x[i] < 0:
            w[i] = -1
    return w

def g(n):
    x = np.zeros(n)
    for i in range(n):
        x[i]= np.cos((np.pi * i) / (n-1))
    f = sign(x)
    (x1, p) = interp(x, f, points=max(2*n, 1000), cheb=True)
    plt.plot(x1, p, '-')
```

[2] i.e. in the procedure *interp()* outlined in subsection 3.1

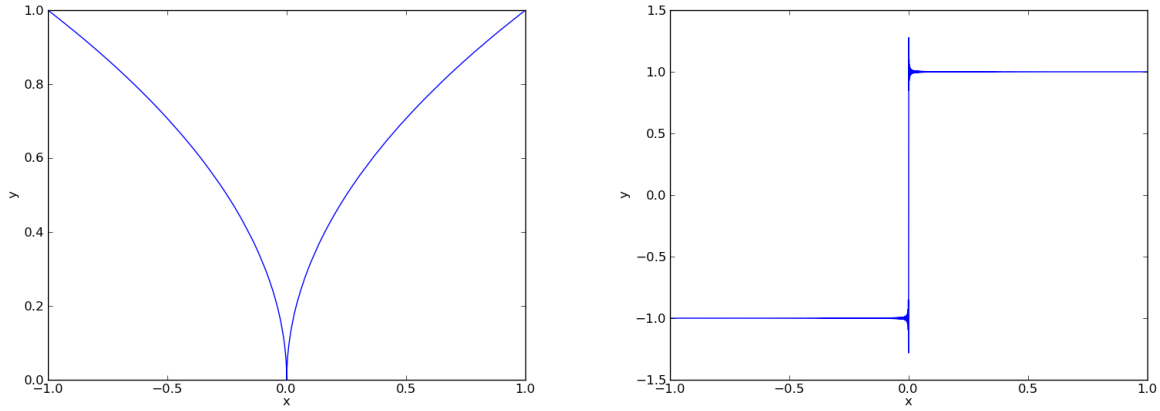

Figure 4: Plot of interpolant of f and g on the left and right respectively using $n = 5001$ in both cases.

Whilst the interpolant of f as plotted in *Figure 4* is indistinguishable from plot of f, this is not the case with function g. This could be expected, of course, since a continuous function can never approximate a discontinuous function perfectly. We will now examine the convergence further by plotting the error of interpolation against number of interpolation points for these functions. Obviously, using the measure of error $\max_{x\in[-1,1]} |g(x) - p(x)|$ will not give much insight to convergence of interpolant of g since there will always be significant difference between g and its interpolant near zero. Instead, we shall use

$$error = \int_{-1}^{1} |p(x) - g(x)|dx \tag{8}$$

which will be approximated using the trapezium rule. Following plots were produced using the function *conv1()* with the amendments

```
f = np.sqrt(abs(x))
f1 = np.sqrt(abs(x1))
```

for function f and

```
f = sign(x)
f1 = sign(x1)
mdiff = trapz(abs(f1 - p), x1)
```

for function g.

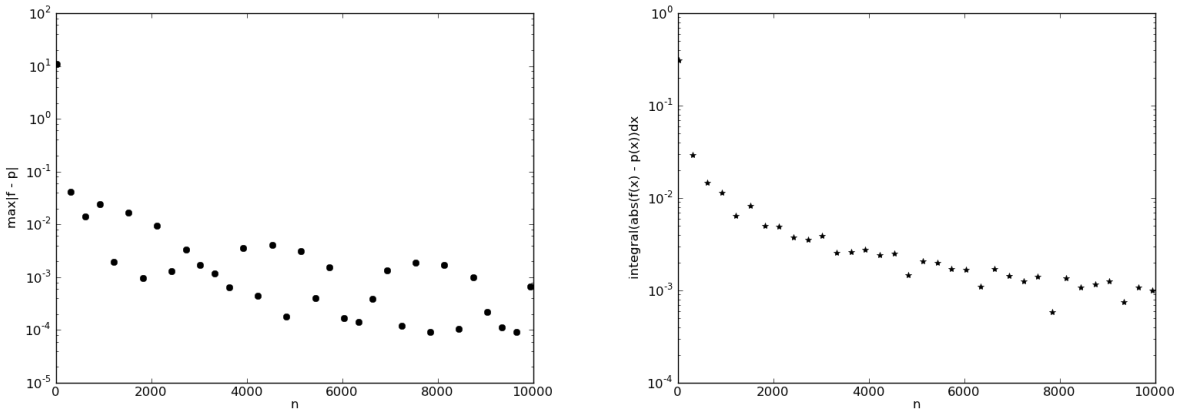

Figure 5: Plots of interpolation of error against number of interpolation points. The plot on the left corresponds to function f and the right to function g. In both cases we use *conv1(10,10000,301)* with the aforementioned amendments to produce these plots.

In *Figure 5* we observe that for function f the convergence is slow and oscillatory in comparison to smoother functions, however when interpolating in more than $n = 6000$ points we can obtain error as small as 10^{-4}.

For the function g we observe even slower convergence rate with error as defined by (8) never dropping under 10^{-3}. Nevertheless, one needs to keep in mind that the measure of error for the two functions are different. Additionally, the plot of convergence for g does not fluctuate nearly as much as for f which is most likely caused by the smoothing effect of integration.

These experiments demonstrate that polynomial interpolation, granted we interpolate in well chosen points, can be used even for less smooth functions if these levels of error are satisfactory for a given application.

5 Numerical stability

When solving a mathematical problem numerically, stability is always an issue that needs to be addressed.

The first problem, that was briefly outlined in section 3, is that of the choice of points. We mentioned that when points are not chosen with the asymptotic density $(1-x^2)^{-1}$ then the value of interpolant can vary exponentially with n along the interval, meaning that small change in interpolation points can result in large difference in the interpolant. Thus this problem is ill-conditioned by definition [2]. Directly related to choice of points is the computation of *Barycentric weights*. If we choose point set where the the *weights* are not known and have to be computed using (3) then this may result in over or underflow [2]. Choosing *Chebyshev nodes* addresses both of these issues, nonetheless even in this case there are further problems that we need to investigate.

Suppose we want to evaluate the interpolant defined by either (4) or (6) at

some point x near x_k for some $k = 0, ..., n$. Instinctively, we would expect to see some instability as we are dividing by very small number, however N.J. Higham proved in [3] that this is in fact not the case. In his paper on this matter Higham proved that (4) is *backward stable* with respect to the values at interpolation points $\{y_i : i = 0, ..., n\}$. That is, if we have two sets of data $D_1 = \{(x_i, y_i) : i = 0, ..., n)\}$ and $D_2 = \{(x_i, f_i) : i = 0, ..., n)\}$ where $f_i \approx y_i$ for all i and if we compute the value of interpolant $p_n(x)$ at some point x using (4) with respect to sets D_1 and D_2 we obtain the same value. Additionally, (6) was shown to be *forward stable* with respect to $\{y_i : i = 0, ..., n\}$ when interpolating in nodes which are clustered near endpoints, meaning that perturbing the data set $\{y_i : i = 0, ..., n\}$ will only result in slight changes in the value $p_n(x)$.

Naturally, we can not compute p_n at x_i for $i = 0, ..., n$ using (4) or (6) directly as we would have zero in the divisor, which can be easily solved by an additional *if* statement[3] followed by setting the values $p_n(x_i) = y_i$ for all i.

6 Conclusion

The theoretical results and numerical experiments in the account above show that after some minor algebraic manipulations, the *Lagrange formula* becomes a powerful technique of polynomial interpolation, especially when interpolating in *Chebyshev nodes* or other point sets clustered near endpoints. Although we did not confirm all the theoretical results on convergence of the interpolant, we saw that the *error* can be made as small as 10^{-15} for smooth functions and 10^{-3} to 10^{-4} for less smooth functions. Additionally, as shown in section 4 and discussed in section 5, the *Barycentric formula* is stable even for as many as 10000 interpolation points. Overall, it is an efficient and fast method of interpolating and even approximating functions on an interval.

References

[1] E. Sli & D. Mayers, *An Introduction to Numerical Analysis*, Cambridge University Press, Cambridge, UK, 2003

[2] J.-P. Berrut & L. N. Trefethen, *Barycentric Lagrange Interpolation*, SIAM Rev., 2004, Vol. 43, No. 3, pp. 501-517

[3] N. J. Higham, *The numerical stability of barycentric Lagrange interpolation*, IMA J. Numer. Anal., 2004, Vol. 24, pp. 547-556

[4] L. N. Trefethen, *Approximation Theory and Approximation Practice*, SIAM, 2012

[5] L. N. Trefethen, *Spectral Methods in Matlab*, SIAM, Philadelphia, 2000 5

[3] second line of the first for-loop in the *interp()* implementation

[6] L. N. Trefethen, *Six Myths of Polynomial Interpolation and Quadrature*, Mathematics Today, July 12, 2011

[7] H. E. Salzer, *Lagrangian interpolation at the Chebyshev points* $x_{n,\nu,} = cos(\nu\pi/n), \nu = 0(1)n$*; some unnoted advantages*, Comput. J., 15 (1972), pp. 156-159.

[8] A. Dutt, M. Gu, & V. Rokhlin, *Fast algorithms for polynomial interpolation, integration, and differentiation*, SIAM J. Numer. Anal., 33 (1996), pp. 1689-1711