# Recitation 10

Recitation Instructor: Shivam Verma
Email: shivamverma@nyu.edu
Ph: 718-362-7836
Office hours: WWH 605 (2.50 - 4.50 pm, Tuesdays)

## Brief Overview

- Hermite interpolation
- Spline interpolation
- Sample problems on polynomial, spline interp.
- Floating point system
- HW5 tips

## Verifying your Integration solutions using MATLAB

Once you've written the Adaptive Simpson algorithm, it would be useful to verify that you get the right solution by comparing it with MATLAB's solution to the integration.

Note: Remember to use **".^"** instead of **"^"** for element-wise exponentiation. Otherwise, you'll probably get an error, as MATLAB expects the function to be able to take a vector.

```
%Use: q = integral(fun,xmin,xmax)

%Eg. 1

func = @(x) sqrt(x)
integral(func,0,1)

%Eg. 2

func = @(x) @(x)(1- x.^2).^(3/2)
integral(func,0,1)
```

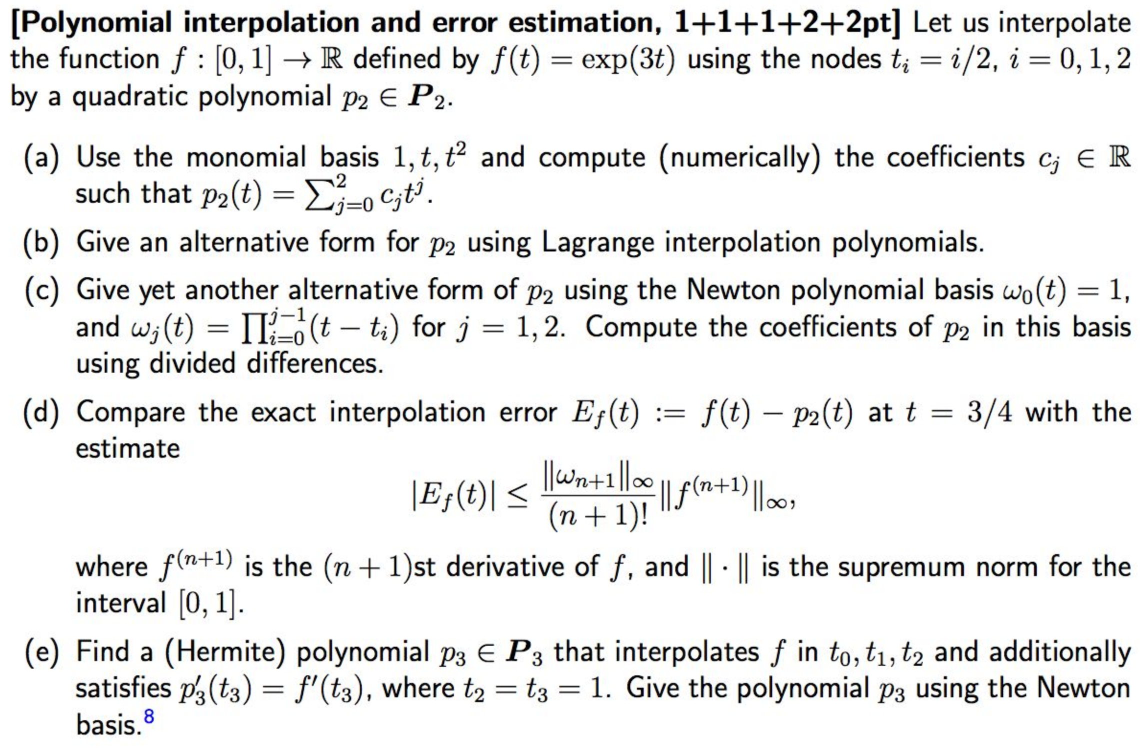
## Cubic splines

I will cover the following topics, using both my own notes and provided references.

- Recap of Runge's phenomenon from previous recitation.
- Overview of Hermite interpolation, Newton's basis and divided-differences from [8, 9]. Also see [10].
- Overview of sec. 1.7 in [10] for spline interpolation. Read [11] for a barebones introduction to spline, and [14] for extensive discussion.
- See [12, 13, 15] for periodic cubic splines (for homework).

## Problem 1

We will solve all the parts in the following problem.



**[Polynomial interpolation and error estimation, 1+1+1+2+2pt]** Let us interpolate the function $f : [0,1] \to \mathbb{R}$ defined by $f(t) = \exp(3t)$ using the nodes $t_i = i/2$, $i = 0, 1, 2$ by a quadratic polynomial $p_2 \in \boldsymbol{P}_2$.

(a) Use the monomial basis $1, t, t^2$ and compute (numerically) the coefficients $c_j \in \mathbb{R}$ such that $p_2(t) = \sum_{j=0}^{2} c_j t^j$.

(b) Give an alternative form for $p_2$ using Lagrange interpolation polynomials.

(c) Give yet another alternative form of $p_2$ using the Newton polynomial basis $\omega_0(t) = 1$, and $\omega_j(t) = \prod_{i=0}^{j-1}(t - t_i)$ for $j = 1, 2$. Compute the coefficients of $p_2$ in this basis using divided differences.

(d) Compare the exact interpolation error $E_f(t) := f(t) - p_2(t)$ at $t = 3/4$ with the estimate

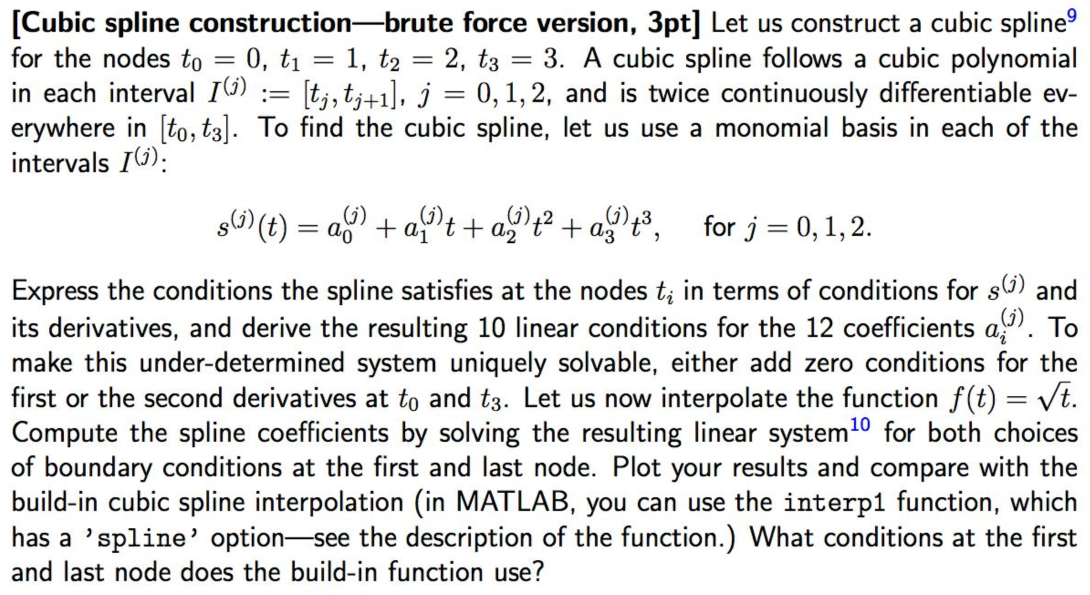
$$|E_f(t)| \leq \frac{\|\omega_{n+1}\|_\infty}{(n+1)!} \|f^{(n+1)}\|_\infty,$$

where $f^{(n+1)}$ is the $(n+1)$st derivative of $f$, and $\|\cdot\|$ is the supremum norm for the interval $[0, 1]$.

(e) Find a (Hermite) polynomial $p_3 \in \boldsymbol{P}_3$ that interpolates $f$ in $t_0, t_1, t_2$ and additionally satisfies $p_3'(t_3) = f'(t_3)$, where $t_2 = t_3 = 1$. Give the polynomial $p_3$ using the Newton basis.[8]

## Problem 2

We will solve the first part, and make a MATLAB script to solve the second part.



**[Cubic spline construction—brute force version, 3pt]** Let us construct a cubic spline[9] for the nodes $t_0 = 0$, $t_1 = 1$, $t_2 = 2$, $t_3 = 3$. A cubic spline follows a cubic polynomial in each interval $I^{(j)} := [t_j, t_{j+1}]$, $j = 0, 1, 2$, and is twice continuously differentiable everywhere in $[t_0, t_3]$. To find the cubic spline, let us use a monomial basis in each of the intervals $I^{(j)}$:

$$s^{(j)}(t) = a_0^{(j)} + a_1^{(j)}t + a_2^{(j)}t^2 + a_3^{(j)}t^3, \quad \text{for } j = 0, 1, 2.$$

Express the conditions the spline satisfies at the nodes $t_i$ in terms of conditions for $s^{(j)}$ and its derivatives, and derive the resulting 10 linear conditions for the 12 coefficients $a_i^{(j)}$. To make this under-determined system uniquely solvable, either add zero conditions for the first or the second derivatives at $t_0$ and $t_3$. Let us now interpolate the function $f(t) = \sqrt{t}$. Compute the spline coefficients by solving the resulting linear system[10] for both choices of boundary conditions at the first and last node. Plot your results and compare with the build-in cubic spline interpolation (in MATLAB, you can use the `interp1` function, which has a `'spline'` option—see the description of the function.) What conditions at the first and last node does the build-in function use?

Problems are from [16]. Sample solution can be found at [17] (problems 5 & 6).

## Floating Point System

- Due to finite precision in floating point number representation, there are gaps between consecutive numbers.
- Size of these gaps depends on the size of the number and on the precision (e.g., double or single precision).
- MATLAB has the function eps(), which returns, for a number, the distance to the next floating point number in the same precision.
- A double precision (64-bit) float is represented by 64 bits, with 52 bits in significand, 11 bits in exponent and 1 bit for sign, and implicit integer bit of value 1. 53 bits in significand is equivalent to roughly 16 decimal digits, since $log_{10}(2^{53}) \approx 15.955.$
- A single precision (32-bit) float is represented by 32 bits, with 23 bits in significand, 8 bits in exponent and 1 bit for sign, and implicit integer bit of value 1.

- 64 bit: Emin = -1022, Emax = 1023, 32 bit: Emin = -126, Emax = 127. Comes from two-complement (see 6,7), $-2^{N-1} to\ (2^{N-1}-1)$.

Examples:

- eps(1)
    - $eps(1) = 2.2204e-16$
    - 1 is represented as $1.00..000 \times 2^0$ where there are 52 zeros in significand. The next largest number is thus $1+2^{-52}$, with $2^{-52} \approx 2.2204e-16$.
- eps(single(1))
    - $eps(single(1)) = 1.1921e-07$
    - For single precision format, there are only 23 bits in the significand (after the implicit 1). This means the next largest representable number is $1 + 2^{-23}$ with $2^{-23} \approx 1.1921e-07$.
- eps(2^(40))
    - $eps(2^{40}) = 2.4414e-04$
    - In the 64-bit representation the number $2^{40}$ is given as $1.00..0 \cdot \times 2^{40}$ so the next number is $2^{40} + 2^{-12}$ with $2^{-12} \approx 2.4414e-04$.
- eps(single(2^(40)))
    - $eps(single(2^{40})) = 131072$
    - In the 32-bit representation there are only 23 zeros so the next number is $2^{40} + 2^{17}$ with $2^{17} = 131072$ which is what single(eps(2^40)) gives.

Try the following in MATLAB:

```
>> a = 0.8;
>> b = 0.7;
>> a - b == 0.1
ans = 0
>> a - b - 0.1
ans = 8.3267e-17

>> a = 0.8;
>> b = 0.4;
>> a - b == 0.4
ans = 1

>> a - b - 0.4
ans = 0
```

Can you explain this behavior?

We'll use the concept of a **dyadic rational,** which is a number of the form $a/2^b$, i.e. its denominator is a power of 2. Such numbers have a finite binary expansion. See [5].

Since neither 0.8 nor 0.7 is a dyadic rational, they cannot be expressed exactly using the 64-bit floating point format. More precisely, we have

$$0.8 \approx a = \tfrac{3602879701896397}{4503599627370496},\ 0.7 \approx b = \tfrac{3152519739159347}{4503599627370496},$$

where a, b are the 64-bit floating point representations. The result of a − b is the closest 64-bit representation of the difference of these 2 approximating dyadic rationals. The exact difference of these fractions is
$a - b = \tfrac{225179981368525}{2251799813685248}$, which is exactly representable as a 64-bit value.

As a result you get:

```
>> a = 0.8;
>> b = 0.7;
>>  a-b == 225179981368525/2251799813685248
ans = 1
```

The value 0.1 is also not dyadic and is approximated as:

$$0.1 \approx c = \tfrac{3602879701896397}{36028797018963968}.$$

We note that $c + (2^{-51} - 2^{-52} - 2^{-53} - 2^{-55}) = a - b.$

These are the bits of 0.1 that were lost in the subtraction of 0.8 − 0.7 since their difference is nearly 8 times smaller than either of them individually. In other words, the errors in representing 0.8 and 0.7 were magnified by the cancellation. This shows that a − b will not equal c when compared as 64-bit values. Note that

```
>> 2^(-51) - 2^(-52) - 2^(-53) - 2^(-55)
ans = 8.3267e-17
```

which is what you obtain from a − b − 0.1.

For the second part:

The value 0.4 is also not a dyadic rational, so the 64-bit approximation gives

$$0.4 \approx d = \frac{3602879701896397}{9007199254740992},$$

and using exact arithmetic we have

$$0.8 - 0.4 \approx a - d = \frac{3602879701896397}{9007199254740992},$$

which is exactly representable with 64-bits and is equal to d. The reason we don't have the same issue as the previous part is that, ignoring overflow and underflow, x and x/2 have the same significand when approximated in 64-bit arithmetic. The subtraction (typically performed in 80-bit arithmetic) then gives the 64-bit approximation to x/2. It is tempting to think the lack of error here is fully explained by the fact that result of the subtraction is close to the order of the operands, but note that .7-.3==.4 gives an answer of 0 (false) debunking that theory.

## Helpful links

1. https://en.wikipedia.org/wiki/IEEE_floating_point
2. Numerical Computing with IEEE Floating Point Arithmetic, Michael L. Overton (NYU)
3. Sec. 2.5, Numerical Mathematics, Alfio Quarteroni et al.
4. Sec. 2.1, Numerical Analysis in Modern Scientific Computing, Peter Deuflhard and Andreas Hohmann.
5. https://en.wikipedia.org/wiki/Dyadic_rational
6. https://en.wikipedia.org/wiki/Two%27s_complement
7. https://en.wikipedia.org/wiki/Double-precision_floating-point_format
8. https://www3.nd.edu/~coast/jjwteach/www/www/30125/pdfnotes/lecture5_9v14.pdf
9. http://math.nyu.edu/~stadler/num1/material/num1_interpolation
10. http://www.math.vt.edu/people/adjerids/homepage/teaching/S05/Math5466/interpolation.pdf
11. https://www.math.ohiou.edu/courses/math3600/lecture19.pdf
12. http://nikolavitas.blogspot.com/2013/09/cubic-spline-interpolation-periodic.html
13. http://vis.uni-kl.de/~alggeom/pdf/ws1213/alggeom_script_ws12_02.pdf
14. http://www.math.uconn.edu/~leykekhman/courses/MATH3795/Lectures/Lecture_15_poly_interp_splines.pdf
15. http://www.maths.lth.se/na/courses/FMN081/FMN081-06/lecture11.pdf
16. http://math.nyu.edu/~stadler/num1/material/assignment5.pdf
17. https://s3.amazonaws.com/piazza-resources/ie2ucbo1ftp4kb/ihz8liwjpvwpn/AA_hw5sol.pdf?AWSAccessKeyId=AKIAIEDNRLJ4AZKBW6HA&Expires=1460542999&Signature=SULn2odXi81HYtrMtl3SwZO9XRs%3D