interpolation

November 3, 2025

0.1 Notebook 2: Interpolation

[ ]:
```
using LaTeXStrings
using LinearAlgebra
using Plots
using Polynomials
```

0.1.1 [Exercise 1] Warm-up exercise

Find the polynomial $p(x) = ax^2 + bx + c$ (a parabola) that goes through the points $(0, 1)$, $(1, 3)$ and $(2, 7)$. Plot on the same graph the data points and the interpolating polynomial.

[ ]:
```
### BEGIN SOLUTION
x = [0, 1, 2]
y = [1, 3, 7]
A = [1 x[1] x[1]^2
     1 x[2] x[2]^2
     1 x[3] x[3]^2]
  = A\y

# Interpolating polynomial
p_interp(x) =  [1] +  [2] * x +  [3] * x^2

# Plots
plot(p_interp)
scatter!(x, y)
### END SOLUTION
```

0.1.2 [Exercise 2] Lagrange interpolation

Write from scratch, in particular without using third party software libraries, a function to obtain the polynomial interpolating the data points

$$(x_0, y_0), \ldots, (x_n, y_n).$$

These data points are passed in arguments `x` and `y`. Your function should return the values taken by the interpolating polynomial p when evaluated at the points $X_0, \ldots, X_m$ contained in argument

`X`. To construct the interpolating polynomial, use the Lagrange form of the interpolant:

$$p(x) = \sum_{i=0}^{n} y_i L_i(x), \qquad L_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

You may use code provided below to test your function

[ ]:
```
function my_interp(X, x, y)
    Y = zero(X)
    ### BEGIN SOLUTION
    for i in 1:length(x)
        L(z) = prod(z .- x[1:end .!== i])
        Y += y[i] * L.(X) / L(x[i])
    end
    ### END SOLUTION
    return Y
end

# Test code
n, m = 10, 100
f(t) = cos(2 * t)
x = LinRange(0, 1, n)
X = LinRange(0, 1, m)
y = f.(x)
Y = my_interp(X, x, y)
plot(X, f.(X), label="Original function")
plot!(X, Y, label="Interpolation")
scatter!(x, y, label="Data")
```

0.1.3 [Exercise 3] Chebyshev interpolation and Runge phenomenon

The objective of this exercise is to illustrate the influence of interpolation nodes on the interpolation error.

1. The function `fit` from the `Polynomials.jl` package can be employed as follows to calculate, given arrays `x` and `y` of the same size, the associated interpolating polynomial:

   ```
   p = fit(x, y)
   ```

   Using this function, write a function

   ```
   get_interpolations(f, d)
   ```

   that interpolates the function `f` using a polynomial of degree `d`. The function should return a tuple of `Polynomial` structures, corresponding to equidistant (with endpoints included) and Chebyshev nodes over the interval $[-1, 1]$.

   Hint (click to display)

   Pour calculer rapidement les noeuds de Tchebychev, on peut exploiter la macro `@.` (comme toujours, il est conseillé de se référer à la documentation d'une commande en tapant `?` puis

la commande dans la console). Cette commande évite d'utiliser des `.` après chaque fonction ou avant chaque opérateur.

```
x = @. -cos( *((0:n-1)+1/2)/n)
```

[ ]:
```
function get_interpolations(f, d)
    ### BEGIN SOLUTION
    n = d + 1
    x_equi = LinRange(-1, 1, n)
    x_cheb = @. -cos( *((0:n-1)+1/2)/n)
    p_equi = Polynomials.fit(x_equi, f.(x_equi))
    p_cheb = Polynomials.fit(x_cheb, f.(x_cheb))
    ### END SOLUTION
    return p_equi, p_cheb
end
```

[ ]:
```
p_test = Polynomial([1., 2., 3.])
@assert get_interpolations(cos, 5) |> length == 2
@assert get_interpolations(sin exp, 5)[1].coeffs |> length == 6
@assert get_interpolations(sin exp, 5)[2].coeffs |> length == 6
@assert get_interpolations(p_test, 2)[1]   p_test
@assert get_interpolations(p_test, 2)[2]   p_test
@assert get_interpolations(cos, 4)[1](0)   1
@assert get_interpolations(cos, 4)[2](0)   1
```

2. Let us fix $d = 20$ and take f to be the following function

$$f(x) = \tanh\left(\frac{x+1/2}{\varepsilon}\right) + \tanh\left(\frac{x}{\varepsilon}\right) + \tanh\left(\frac{x-1/2}{\varepsilon}\right), \qquad \varepsilon = .02.$$

Using your `get_interpolations` function, calculate the interpolating polynomials in this case, and print the L^∞ error corresponding to equidistant and Chebyshev polynomials.

Hint (click to display)

- To limit roundoff errors, it is preferable that the function returns a BigFloat type, in other words

```
f(x) = BigFloat(tanh((x+1/2)/ ) + tanh(x/ ) + tanh((x-1/2)/ ))
```

- To calculate the infinity norm of a function in order to evaluate the precision of the interpolation, you can use the `norm(..., Inf)` function from the `LinearAlgebra` library with a sufficiently fine sampling of the function values, or you can use the `maximum` function:

```
maximum(abs, [1, -3, 2])  # = 3
```

Note that converting a number `y` of type `BigFloat` to `Float64` can be done with `convert(Float64, y)` or, more simply in this case, `Float64(y)`.

[ ]:
```
d,   = 20, .02
f(x) = BigFloat(tanh((x+1/2)/ ) + tanh(x/ ) + tanh((x-1/2)/ ))
```

```
# Calculate L^∞ errors below
### BEGIN SOLUTION
X = LinRange(-1, 1, 500)
p_equi, p_cheb = get_interpolations(f, d)
round_error(x) = Float64(round(x, sigdigits=3))
error_inf_equi = maximum(round_error ∘ abs, f.(X) - p_equi.(X))
error_inf_cheb = maximum(round_error ∘ abs, f.(X) - p_cheb.(X))
### END SOLUTION

println("L^∞ error with equidistant nodes: ", error_inf_equi)
println("L^∞ error with Chebyshev nodes: ", error_inf_cheb)
```

[ ]:
```
@assert round(error_inf_equi, sigdigits=1) == 200
@assert round(error_inf_cheb, sigdigits=1) == 0.7
```

3. Plot the interpolating polynomials on top of the function `f`.

   Hint (click to display)

   It can be useful, when comparing the two interpolations, to limit the minimum and maximum values on the `y` axis using the option `ylims = (ymin, ymax)` in a `plot` function, or its equivalent ending with `!`. It's worth noting that, by convention in `Julia` (though not mandatory), a function whose name ends with `!` modifies its arguments. In the case of a graph, the first command initiating the graph should not include `!` (`plot`), while subsequent commands that increment the same graph end with `!` (`plot!`, `scatter!`, etc.). Any omission of the `!` is considered a *reset* of the plot.

[ ]:
```
X = LinRange(-1, 1, 500)
plot(X, f.(X), linewidth=4, label="f")

### BEGIN SOLUTION
plot!(X, p_equi.(X), linewidth=3, color=:green, label="Equidistant␣
 ↪interpolation")
plot!(X, p_cheb.(X), linewidth=3, color=:red, label="Chebyshev interpolation")
plot!(xlims = (-1, 1), ylims = (-3.5, 3.5))
### END SOLUTION
```

0.1.4 [Exercise 4] Solving the Euler-Bernoulli beam equation by interpolation

The aim of this exercise to explore a practical application of polynomial interpolation. More precisely, we will implement a numerical method to approximately solve the Euler-Bernoulli beam equation with homogeneous Dirichlet boundary conditions:

$$u \in C^4([0,1]), \quad \begin{cases} u''''(x) = \varphi(x) & \forall\, x \in (0,1), \\ u(0) = u'(0) = u'(1) = u(1) = 0, \end{cases}$$

where $\varphi(x) = (2\pi)^4 \cos(2\pi x)$ is a given transverse load applied to the beam. In order to solve the equation numericaly, we approximate the right-hand side φ by its interpolating polynomial $\widehat{\varphi}$, and then we solve the equation exactly with the right-hand side $\widehat{\varphi}$ instead of φ.

1. Let us first write a function `fit_values_and_slopes(u , up , u , up )` which returns the unique polynomial p of degree 3 such that

$$p(0) = u_0, \qquad p'(0) = up_0, \qquad p(1) = u_1, \qquad p'(1) = up_1.$$

[ ]:
```
function fit_values_and_slopes(u , up , u , up )
    # We look for polynomials p(x) = a  + a  x + a  x² + a  x³
    A = [1 0 0 0; 0 1 0 0; 1 1 1 1; 0 1 2 3]
      = A\[u ; up ; u ; up ]
    return Polynomial( )
end

# Test our code
p = fit_values_and_slopes(-1, -1, 1, 1)
plot(p, xlims=(0, 1))
```

2. Write a function `approx(n)` implementing the approach described above for solving the PDE. The function should return a polynomial approximation of the solution based on an interpolation of **degree** n of the right-hand side at equidistant points between 0 and 1, inclusive.

   Hint (click to display)

   - You can use the function `Polynomials.fit` library to obtain the interpolating polynomial:

     ```
     p = fit(x, y)
     ```

     where `x` are the interpolation nodes, and `y` are the values of the function to interpolate.
   - To calculate the exact solution with a polynomial right-hand side, notice that all solutions are polynomials, and without boundary conditions, the solution is unique modulo a cubic polynomial.
   - You can use the `integrate` function from the `Polynomials.jl` library, which calculates an antiderivative of a polynomial:

     ```
     P = integrate(p)
     ```
   - Use the `BigFloat` format to limit rounding errors. `julia X = LinRange{BigFloat}(0, 1, n + 1)`

[ ]:
```
# Right-hand side
 (x) = (2 )^4 * cospi(2*x)

# Exact solution (for comparison purposes)
u(x) = cospi(2*x) - 1

function approx(n)
    X = LinRange{BigFloat}(0, 1, n + 1)
    ### BEGIN SOLUTION
    Y =  .(X)
    p = fit(X, Y)
    uh = integrate(integrate(integrate(integrate(p))))
```

```
    uh = derivative(uh)
    uh -= fit_values_and_slopes(uh(0), uh(0), uh(1), uh(1))
    return uh
    ### END SOLUTION
end

plot(approx(3), xlims=(0, 1), label="Exact solution")
plot!(u, xlims=(0, 1), label="Approximate solution")
```

3. Write a function `estimate_error(n)` that approximates the error, in L^∞ norm, between the exact and approximate solutions. Note that the exact solution is given by

$$\varphi(x) = \cos(2\pi x) - 1.$$

[ ]:
```
function estimate_error(n)
    ### BEGIN SOLUTION
    un = approx(n)
    x_fine = LinRange(0, 1, 1000)
    un_fine, u_fine = un.(x_fine), u.(x_fine)
    return maximum(abs.(u_fine - un_fine))
    ### END SOLUTION
end
```

4. Plot the error for n in the range $\{5, \dots, 50\}$. Use a logarithmic scale for the y axis.

[ ]:
```
# ### BEGIN SOLUTION
ns = 5:50
errors = estimate_error.(ns)
plot(ns, errors, marker = :circle, label=L"$L^{\infty}$ Error")
plot!(yaxis=:log, lw=2)
# ### END SOLUTION
```