# Workshop 9

Owen Miller, Perry Deng, Clarrisa Xue

```
@enum Methods begin
    natural
    adjusted
    clamped
    paraterm
    notaknot
end
```

# 1. Code: Splines

cubic_splines (generic function with 2 methods)

```
function cubic_splines(inter_points, method, slopes=nothing)
    n, _ = size(inter_points)

    xs = inter_points[:, 1]
    ys = inter_points[:, 2]


    matrix = zeros(n, n)
    RHS = zeros(1, n)'


    for i in 2:(n-1)
        RHS[i] = 6.0 * (((ys[i+1] - ys[i]) / (xs[i+1] - xs[i])) - ((ys[i] - ys[i-1]) / (xs[i] - xs[i-1])))

        for j in 1:n
            if j == i-1
                matrix[i, j] += (xs[i] - xs[i-1])
            elseif j == i
                matrix[i, j] += 2.0 * (xs[i+1] - xs[i-1])
            elseif j == i+1
                matrix[i, j] += (xs[i+1] - xs[i])
            end
        end
    end

    # Natrual Spline Boundary Conditions
    if method == natural
        matrix[1, 1] = 1.0
        RHS[1] = 0.0
        matrix[n, n] = 1.0
        RHS[n] = 0.0
    elseif method == adjusted
        matrix[1, 1] = 1.0
        RHS[1] = slopes[1]
        matrix[n, n] = 1.0
        RHS[n] = slopes[2]
    elseif method == clamped
```

```
        matrix[1, 1] = 2.0
        matrix[1, 2] = 1.0
        RHS[1] = 6.0 * ((ys[2] - ys[1]) / (xs[2] - xs[1]) - slopes[1])
        matrix[n, n-1] = 2.0
        matrix[n, n] = 1.0
        RHS[n] = 6.0 * (slopes[2] - (ys[n] - ys[n-1]) / (xs[n] - xs[n-1]))
    elseif method == paraterm
        matrix[1, 1] = 1.0
        matrix[1, 2] = -1.0
        RHS[1] = 0.0
        matrix[n, n-1] = -1.0
        matrix[n, n] = 1.0
        RHS[n] = 0.0
    elseif method == notaknot
        matrix[1, 1] = xs[3] - xs[2]
        matrix[1, 2] = xs[1] - xs[3]
        matrix[1, 3] = xs[2] - xs[1]
        RHS[1] = 0.0
        matrix[n, n-2] = xs[n] - xs[n-1]
        matrix[n, n-1] = xs[n-2] - xs[n]
        matrix[n, n]   = xs[n-1] - xs[n-2]
        RHS[n] = 0.0
    else
        println("ERROR: Unknown Boundry Condition Method")
        return nothing
    end



    solution = matrix \ RHS

    outputs = []
    points = 20
    for i in 1:(n-1)
        dx = xs[i+1] - xs[i]

        for j in 2:points
            b = float(j) / float(points)
            a = 1-b
            x = xs[i] + b*dx

            c = ((a^3 - a) * dx^2) / 6.0
            d = ((b^3 - b) * dx^2) / 6.0

            y = a*ys[i] + b*ys[i+1] + c*solution[i] + d*solution[i+1]

            push!(outputs, (x,y))
        end
    end
    return outputs
end
```

# 2. Algebra

$$\int_{x_j}^{x_{j+1}} y_{spline}(x)dx = \int_{x_j}^{x_{j+1}} Ay_j dx + \int_{x_j}^{x_{j+1}} By_{j+1} dx + \int_{x_j}^{x_{j+1}} Cy_j'' dx + \int_{x_j}^{x_{j+1}} Dy_{j+1}'' dx$$

After splitting the integral apart we can solve each piece:

$$\int_{x_j}^{x_{j+1}} Ay_j dx = \int_{x_j}^{x_{j+1}} \left(\frac{x_{j+1} - x}{x_{j+1} - x_j}\right) y_j dx = y_j \left(\frac{x_{j+1} - x_j}{2}\right)$$

$$\int_{x_j}^{x_{j+1}} By_{j+1} dx = \int_{x_j}^{x_{j+1}} (1 - A) y_{j+1} dx = y_{j+1} \left(\frac{x_{j+1} - x_j}{2}\right)$$

$$\int_{x_j}^{x_{j+1}} Cy_j'' dx = \frac{y_j''(x_{j+1} - x_j)^2}{6} \int_{x_j}^{x_{j+1}} (A^3 - A) dx = y_j'' \left(\frac{(x_{j+1} - x_j)^3}{24}\right)$$

$$\int_{x_j}^{x_{j+1}} Dy_{j+1}'' dx = \frac{y_{j+1}''(x_{j+1} - x_j)^2}{6} \int_{x_j}^{x_{j+1}} (B^3 - B) dx = y_{j+1} \left(\frac{-(x_{j+1} - x_j)^3}{8}\right)$$

$$\int_{x_j}^{x_{j+1}} y_{spline}(x)dx = y_j \left(\frac{x_{j+1} - x_j}{2}\right) + y_{j+1} \left(\frac{x_{j+1} - x_j}{2}\right) + y_j'' \left(\frac{(x_{j+1} - x_j)^3}{24}\right) + y_{j+1} \left(\frac{-(x_{j+1} - x_j)^3}{8}\right)$$

---

# Function $f(x)$ and It's Derivatives

$$f(x) = x^3 - 5x^2 + 6x - 1$$

$$f'(x) = 3x^2 - 10x + 6$$

$$f''(x) = 6x - 10$$

```
f (generic function with 1 method)
```

```
function f(x)
    (x.^3) .- (5 .*x.^2) .+ (6 .*x) .- 1
end
```

```
f_prime (generic function with 1 method)
```

```
function f_prime(x)
    (3 .* x.^2) .- (10 .* x) .+ 6
end
```

```
f_double_prime (generic function with 1 method)
```

```
function f_double_prime(x)
    (6 .*x) .- 10
end
```

fx_values_1ton (generic function with 1 method)

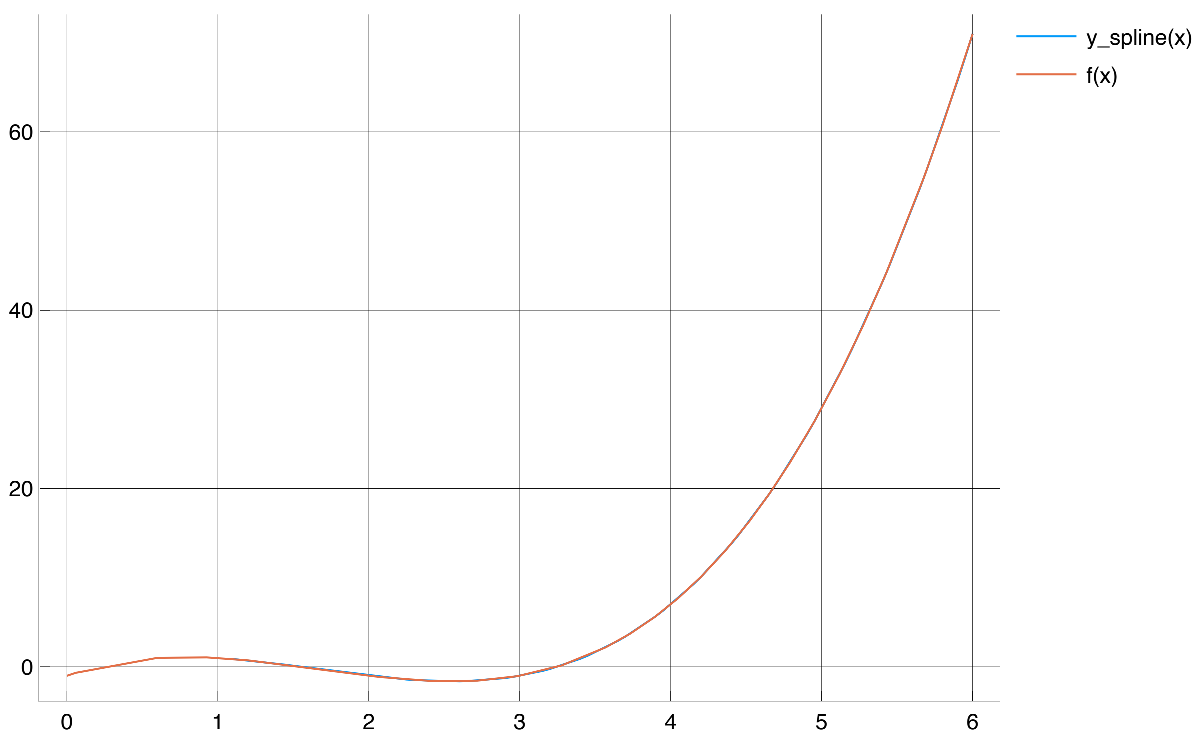
```
function fx_values_1ton(n::Int, func)
    # assumes func takes in a scalar and returns a scalar
    # returns a matrix of dimension [n, 2]
    x_vals = Vector(1:n)
    y_vals = func.(x_vals)
    return [x_vals y_vals]
end
```

# 3. Validation

Here we can see that our cubic spline, $y_{spline}(x)$, completely overlaps our oringial function $f(x)$ when using the curvature adjusted boundary conditions.

**spline_adjusted** = Any[(1.1, 0.881), (1.15, 0.808375), (1.2, 0.728), (1.25, 0.640625

```
spline_adjusted = cubic_splines(fx_values_1ton(6, f), adjusted, (f_double_prime(1), f_double_prime(6)))
```




```
begin
    if length(size(spline_adjusted)) == 1
        using Plots
        plotly()
        plot([p[1] for p in spline_adjusted], [p[2] for p in spline_adjusted], label="y_spline(x)")
        plot!(f, 0, 6, label="f(x)")
    end
```

```
end
```

# 4. Invalidation?

From the following two plots, $y_{spline}(x)$ vs. $f(x)$ and $y_{spline}(x) - f(x)$ vs. $x$, we can see that all methods are relatively close to the original function. However, from the error plot it is clear that the natural, clamped, and parabolically terminated boundry conditions do not perform as well as the curvature-adjusted and not-a-knot boundary conditions.

```
spline_natural =  Any[(1.1, 0.786737), (1.15, 0.680356), (1.2, 0.574278), (1.25, 0.4
```

```
spline_natural = cubic_splines(fx_values_1ton(6, f), natural)
```

```
spline_clamped =  Any[(1.1, 0.880839), (1.15, 0.808033), (1.2, 0.727429), (1.25, 0.6
```

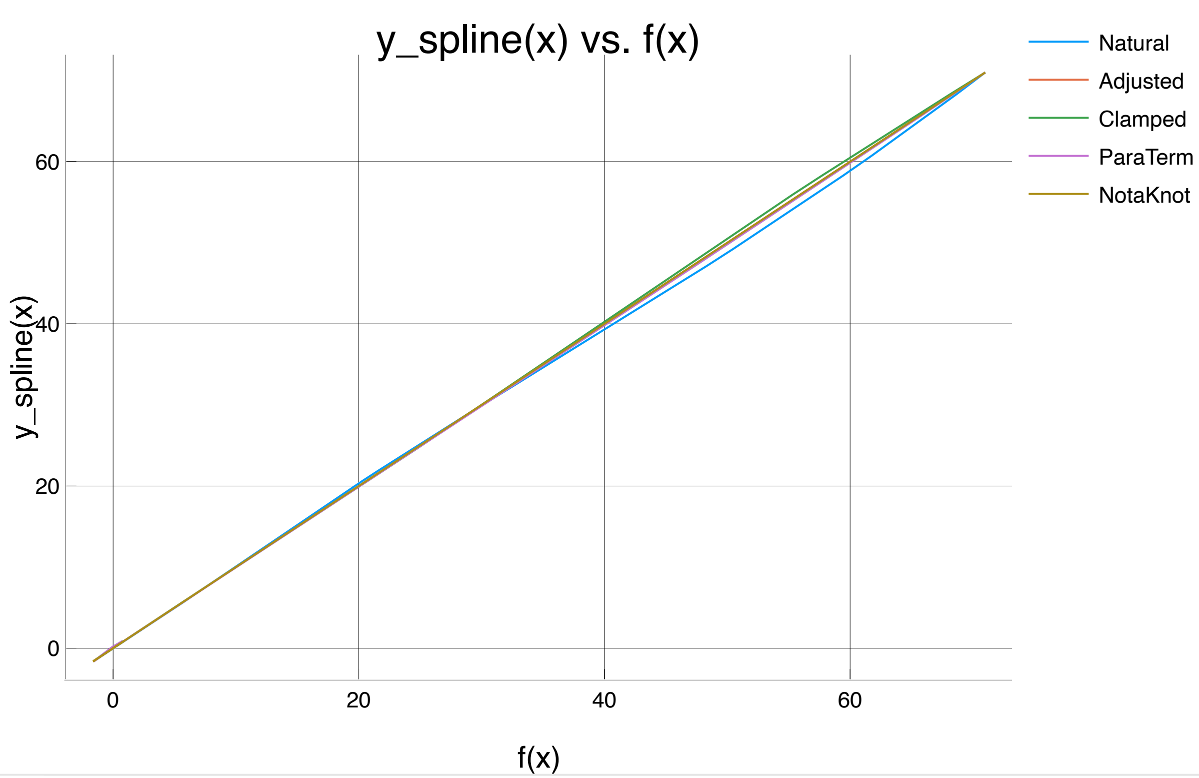
```
spline_clamped = cubic_splines(fx_values_1ton(6, f), clamped, (f_prime(1), f_prime(6)))
```

```
spline_paraterm =  Any[(1.1, 0.767857), (1.15, 0.654464), (1.2, 0.542857), (1.25, 0.
```

```
spline_paraterm = cubic_splines(fx_values_1ton(6, f), paraterm)
```

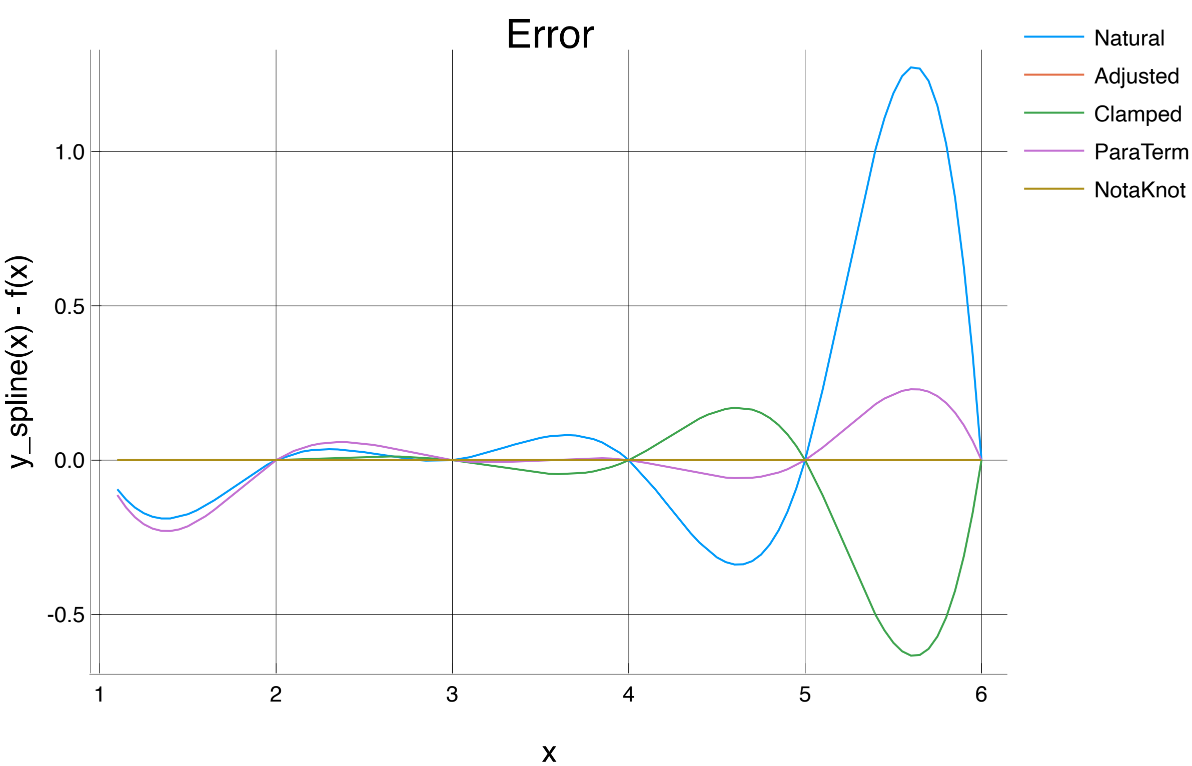
```
spline_notaknot =  Any[(1.1, 0.881), (1.15, 0.808375), (1.2, 0.728), (1.25, 0.640625
```

```
spline_notaknot = cubic_splines(fx_values_1ton(6, f), notaknot)
```




```
begin
    if length(size(spline_adjusted)) == 1
        plotly()
```

```
        plot( [p[2] for p in spline_natural], f([p[1] for p in spline_natural]), label="Natural")
        plot!([p[2] for p in spline_adjusted], f([p[1] for p in spline_adjusted]), label="Adjusted")
        plot!([p[2] for p in spline_clamped], f([p[1] for p in spline_clamped]), label="Clamped")
        plot!([p[2] for p in spline_paraterm], f([p[1] for p in spline_paraterm]), label="ParaTerm")
        plot!([p[2] for p in spline_notaknot], f([p[1] for p in spline_notaknot]), label="NotaKnot")
        title!("y_spline(x) vs. f(x)")
        xaxis!("f(x)")
        yaxis!("y_spline(x)")
    end
end
```




```
begin
    if length(size(spline_adjusted)) == 1
        plotly()
        plot( [p[1] for p in spline_natural], [p[2] for p in spline_natural] .- f([p[1] for p in spline_natural]), label="Natural")
        plot!([p[1] for p in spline_adjusted], [p[2] for p in spline_adjusted] .- f([p[1] for p in spline_adjusted]), label="Adjusted")
        plot!([p[1] for p in spline_clamped], [p[2] for p in spline_clamped] .- f([p[1] for p in spline_clamped]), label="Clamped")
        plot!([p[1] for p in spline_paraterm], [p[2] for p in spline_paraterm] .- f([p[1] for p in spline_paraterm]), label="ParaTerm")
        plot!([p[1] for p in spline_notaknot], [p[2] for p in spline_notaknot] .- f([p[1] for p in spline_notaknot]), label="NotaKnot")
        title!("Error")
        xaxis!("x")
        yaxis!("y_spline(x) - f(x)")
    end
end
```