Project 2

Problem 1

As demonstrated in the lecture, approximate $f_{star}(x) = sin(x)$ on the interval $[0, \pi/2]$ via polynomials using a least squares approach. Use $1,000$ evaluation points. Create a repository on Github holding your code. This repository needs to have a README explaining (in brief) the algorithm, as well as instructions for using your code to reproduce the results. These instructions should also include figures since a picture is worth a thousand words. Your code should be able to show results for all tasks in this project, and your code should be well-written and understandable. Use comments.

Solution:

In [1]:
```
using WGLMakie
```

In [2]:
```
# expansion coefficents
pmax = 4
#100 points are taken for evaluation
npoints = 100
x = LinRange(0, π/2, npoints)
# target function
f(x) = sin(x)
# matrix (description of matrix given in the next block)
A = [x[i]^p for i in 1:npoints, p in 0:pmax]
fstart(x) = sin(x)
y = fstart.(x);
```

Least Square Method

Goal: Given a target function $f_*(x)$ we want to make a approximate p-degree polynomial $f(x) = \sum_{i=0}^{p} b_i x^i$. Next we evaluate $(x_{(1)}, x_{(2)}, \cdots, x_{(n)})$, $n \geq p$ points are beinge evaluated for both $f(x)$ and $f_*(x)$, so that the RMS error got minimized

$$\hat{e} = \sqrt{\sum_{i=0}^{n} \frac{\left(f(x_{(i)}) - f_*(x_{(i)})\right)^2}{n}}$$

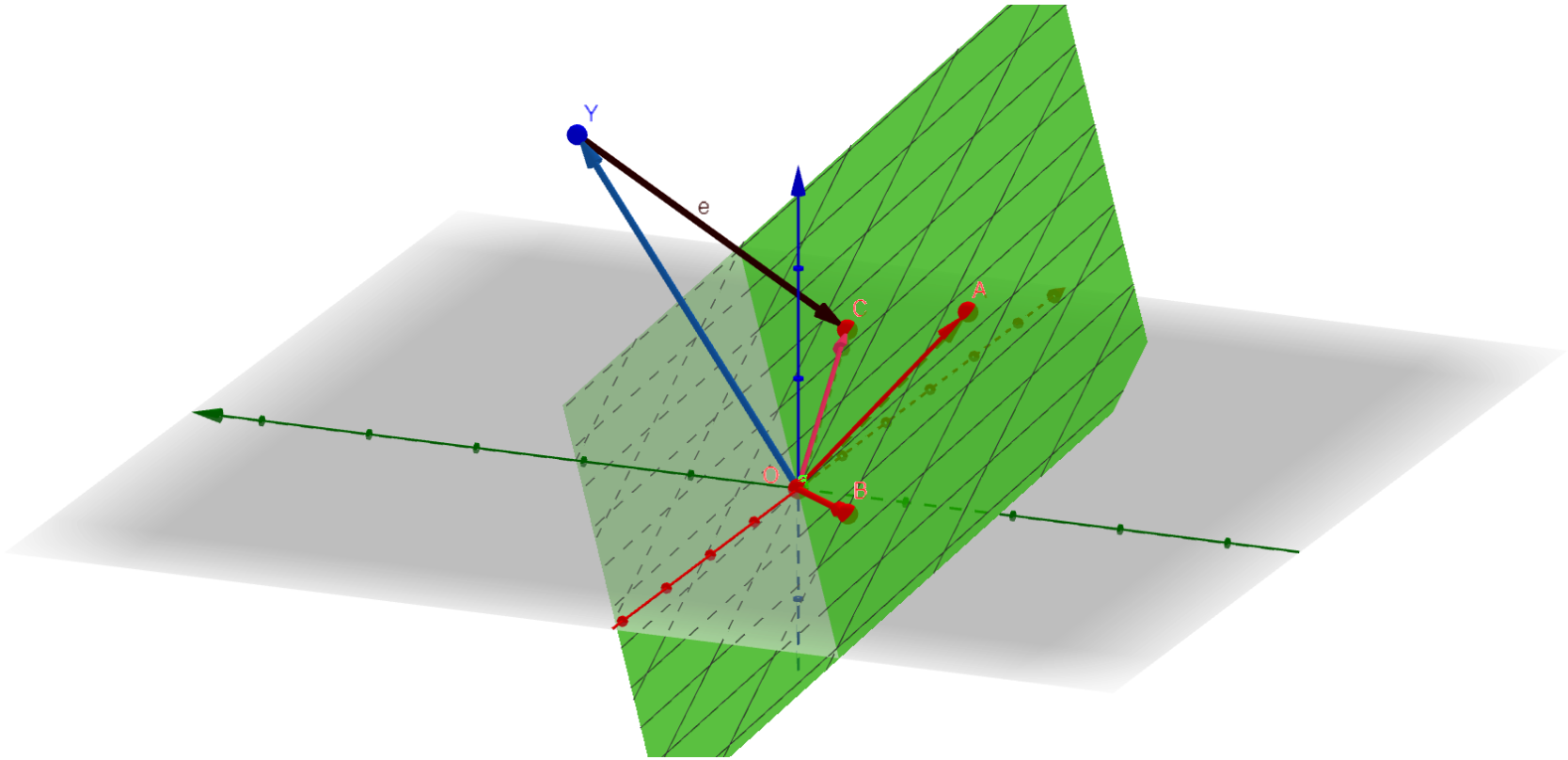
Lets define $\bar{b} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_p \end{pmatrix}$, $\bar{y} = \begin{pmatrix} f_*(x_{(0)}) \\ f_*(x_{(1)}) \\ \vdots \\ f_*(x_{(n)}) \end{pmatrix}$ and $\bar{A} = \begin{pmatrix} 1 & x_{(1)} & \cdots & x_{(1)}^p \\ 1 & x_{(2)} & \cdots & x_{(2)}^p \\ & & & \\ \vdots & \ddots & \ddots & \vdots \\ 1 & x_{(n)} & \cdots & x_{(n)}^p \end{pmatrix}$.

We can rephrase the problem in maxtrix format like,

$$\bar{A}\bar{b} = \bar{y}$$

where $\bar{b}$ contains the coefficient of fitting polynomial $f(x)$.

This can be represented into geometric language. We can say the best fitting can be achieved, if we can find the best approximation of $\bar{y}$ into the column space of $\bar{A}$ matrix. Pictorially, we represent the column space of $\bar{A}$ spanned by $\vec{OA}$ and $\vec{OB}$ vectors in the following picture and $\bar{y}$ vector by $\vec{OY}$. The best approximation we can find for $\vec{OY}$ with in column space of $\bar{A}$ by projecting $\vec{OY}$ over the hyperspace spanned by $\vec{OA}$ and $\vec{OB}$. Because the "error" vector $\vec{OY} - \vec{OC} = \vec{YC} = e$ would have the minimum length (L^2 or Euclidean norm) if we project $\vec{OY}$ over the hyperspace. We identify that the previously mentioned RMS error $\hat{e} = \frac{|e|}{\sqrt{n}}$.



If we want to project some n dimensional vector $\vec{u}$ over the hyperspace if we act $\bar{A}^T$ (transpose matrix of $\bar{A}$) on the $\vec{u}$. Because we know that $\bar{A}$ can be represented as ,

$$\bar{A} = \sum_{i=1}^{n} \sum_{j=0}^{p} |e_i\rangle\langle f_j|$$

where $\{|f_i\rangle\}$ and $\{|e_i\rangle\}$ span the $p+1$ and n-dimesional vector space. Therefore, $\bar{A}^T \bar{u}$ would live in the $p+1$ dimensional vector subspace. Hence, we can find the projected equation in $p+1$-dimensional vector space would be,

$$\bar{A}^T \bar{A} \bar{b} = \bar{A}^T \bar{y}$$

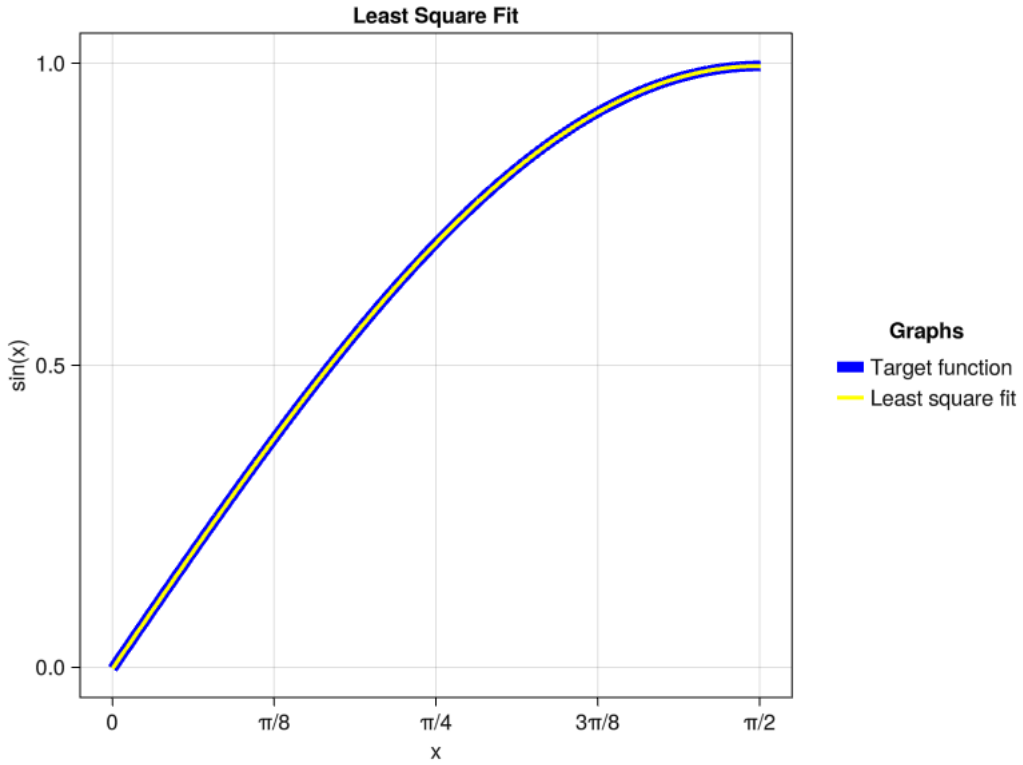
By solving this system of equation we can find the projected vector $\vec{OC}$ in the $p+1$ dimensional vector space with basis vectors $|f_i\rangle, i = 0, \cdots, p$. We compute the solution numerically here.

In [3]:
```
b = (A'*A) \ (A' * y)
f(b, x) = sum(b[p+1]* x^p for p in 0:pmax);
```

We can compare the performance of least square fit by plotting $f_*(x)$ and $f(x)$ in the same plot.

In [4]:
```
fig = Figure()
ax=Axis(fig[1,1], xlabel="x", ylabel="sin(x)", title = "Least Square Fit",
        xticks = (0:π/8:π/2, ["0", "π/8", "π/4", "3π/8", "π/2"]))
xx = LinRange(0, π/2, 100)
yy = [f(b,x) for x in xx]
lines!(xx,fstart.(xx),color=:blue,linewidth=8,label="Target function")
lines!(xx, yy,color=:yellow,linewidth=3,label="Least square fit")
fig[1, 2] = Legend(fig, ax, "Graphs", framevisible = false)
fig
```

Out[4]:



Problem 2

Define the approximation error E as the L^2 norm of the difference between the approximation function $f(x)$ and $f_{star}(x)$. This is also called RMS or "root mean square" of the difference: The square root of the average of $(f(x) - f_{star}(x))^2$ at all evaluation points. What polynomial order ($pmax$) is required to satisfy $E < 1.0e-10$? Plot the relation between $pmax$ and E for $pmax$ from 0 to at least 20. Discuss the figure. Can you achieve $E < 1.0e-20$? Say why.

Solution:

At first we define a function to compute the RMS error.

In [5]:
```
function RMS_error_calculator(target_function, args, pmax)
    """
    This is a simple implemetation of *RMS error* calcuation process for least square curve fitting.

    Inputs:

    * target_function: a function ƒstar(x) that is need to be approximated
    * args           : the required arguments to evalue the target_function
    * pmax           : the order of the approximated polynomial f(x)

    Returns RMS error = √(Σ(f(x)- ƒstar(x))^2/n)

    Usuage:

    ```
    # target function is sin(x)
    xx = LinRange(0, π/2, 100)
    error = RMS_error_calculator(sin, xx, 10)

    ```

    """
    # number of evaluation points
    n = length(args)
    # matrix A
    A = [args[i]^p for i in 1:n, p in 0:pmax]
    y = target_function.(args)
    # solution of the linear equation projected in hyperspace
```

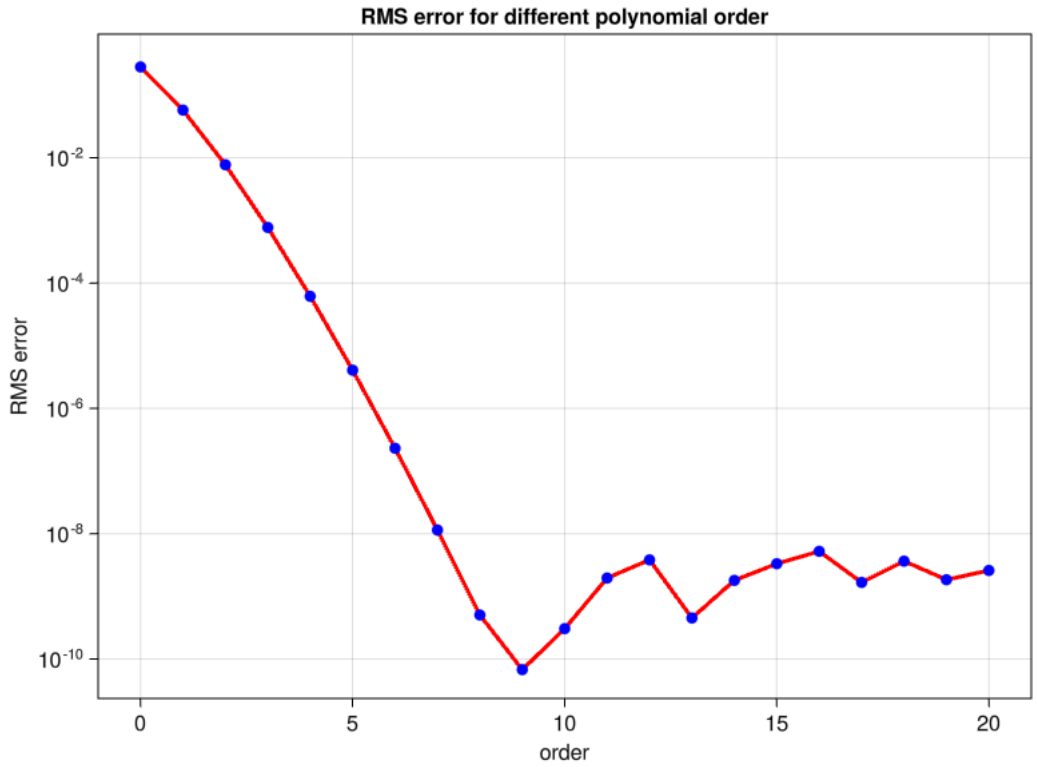
```
    b = (A'*A) \ (A' * y)
    # approximated function defintion
    approx(b, x) = sum(b[p+1]* x^p for p in 0:pmax)
    # estimated value from the approximated function calculation
    yvalue_approx = [approx(b,x) for x in args]
    # now we return the RMS error
    return sqrt(sum((yvalue_approx .- y).^2)/n)
end;
```

Next we evaluate the error for `pmax` from 0 to 20 and plot `E` vs `pmax`.

In [6]:
```
# initialize RMS error collection array for different order of polynomial
xx = LinRange(0, π/2, 1000)
orders = 0:20
E = [RMS_error_calculator(sin, xx, pmax) for pmax in orders];
```

In [7]:
```
fig = Figure()
ax=Axis(fig[1,1], xlabel="order", ylabel="RMS error", yscale = log10,
    title = "RMS error for different polynomial order")
lines!(orders,E,color=:red,linewidth=3)
scatter!(orders,E,color=:blue,label="Target function")
fig
```

Out[7]:



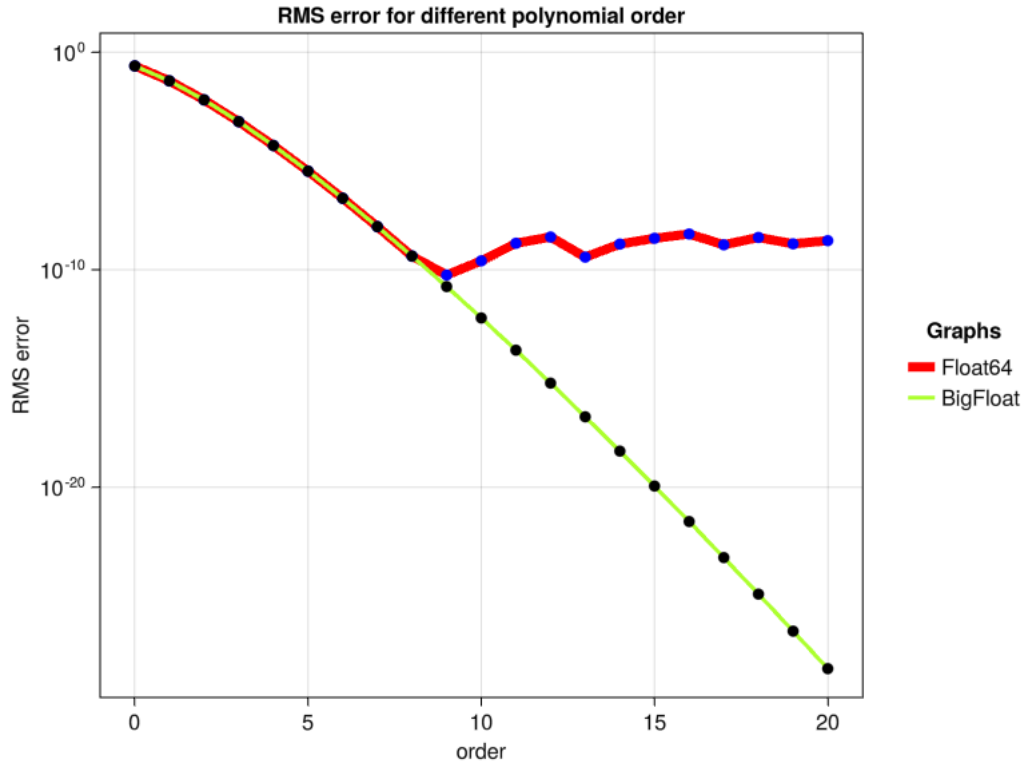
From the plot it is clear that at first RMS error decreases as `pmax` increased, but after `pmax` ≥ 10 it becomes bumpy. This happens because of the limited storage capacity of `Float64` data type. For higher order terms it truncate the real numbers in particular fashion that it becomes difficult to reach error value less than 10^{-10} using this. In Julia we can somewhat eliminate that using `BigFloat` type numbers,even we can achieve error less than 10^{-20}.

In [8]:
```
E_Big = [RMS_error_calculator(sin, BigFloat.(xx), pmax) for pmax in orders];
```

In [9]:
```
fig = Figure()
ax=Axis(fig[1,1], xlabel="order", ylabel="RMS error", yscale = log10,
    title = "RMS error for different polynomial order")
lines!(orders,E,color=:red,linewidth=7,label="Float64")
scatter!(orders,E,color=:blue)
lines!(orders,E_Big,color=:greenyellow,linewidth=3,label="BigFloat")
scatter!(orders,E_Big,color=:black)
```

```
fig[1, 2] = Legend(fig, ax, "Graphs", framevisible = false)
fig
```

Out[9]:



Although using `BigFloat` data type for computation is costly, it can be used to achieve higher accuracy.

Problem 3

The function $sin(x)$ is antisymmetric, i.e. $sin(x) = -sin(-x)$. Modify the method to use only antisymmetric polynomials in your approximation. How does this affect the error? Compare results for the same computational cost, i.e. for the same number of polynomials used (not for the same pmax).

Solution:

To calculate the $sin(x)$ using antisymmetric polynomial $f_a(x) = \sum_{i=0}^{p} b_i x^{2i+1}$ where `pmax` = $2p+1$ in this case.

In [10]:
```
function RMS_error_calculator_anti(target_function, args, pm)
    """
    This is a simple implemetation of *RMS error* calcuation process for least square curve fitting.

    Inputs:

    * target_function: a function ƒstar(x) that is need to be approximated
    * args           : the required arguments to evalue the target_function
    * pm             : the order of the approximated polynomial f(x)

    Returns RMS error = √(Σ(f(x)- ƒstar(x))^2/n)

    Usuage:

    ```
    # target antisymmetric function is sin(x)
    xx = LinRange(0, π/2, 100)
    error = RMS_error_calculator_anti(sin, xx, 10)

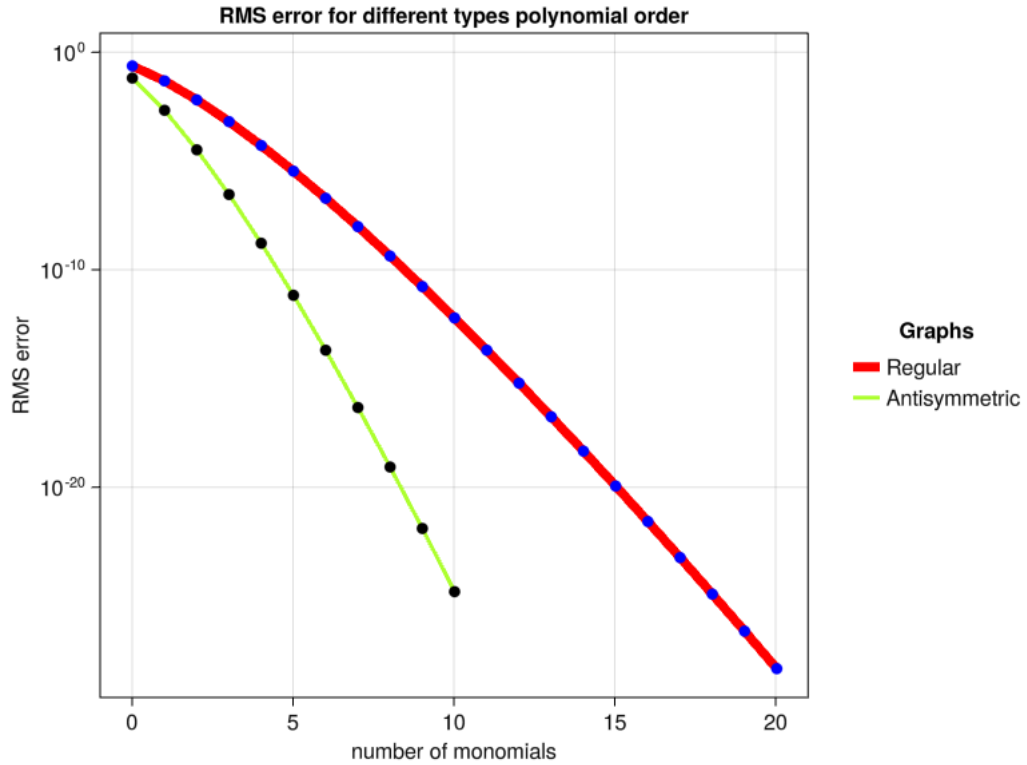
    ```

    """
    # number of evaluation points
    n = length(args)
    # matrix A using antisymmetric
    A = [args[i]^(2*p+1) for i in 1:n, p in 0:pm]
    y = target_function.(args)
    # solution of the linear equation projected in hyperspace
    b = (A'*A) \ (A' * y)
    # approximated function defintion antisymmetric
    approx(b, x) = sum(b[p+1]* x^(2*p+1) for p in 0:pm)
    # estimated value from the approximated function calculation
    yvalue_approx = [approx(b,x) for x in args]
    # now we return the RMS error
    return sqrt(sum((yvalue_approx .- y).^2)/n)
end;
```

In [11]:
```
# initialize RMS error collection array for different order of polynomial
xx = LinRange(0, π/2, 1000)
```

```
norders = 0:10
Eanti = [RMS_error_calculator_anti(sin, BigFloat.(xx), pp) for pp in norders];
```

In [12]:
```
fig = Figure()
ax=Axis(fig[1,1], xlabel="number of monomials", ylabel="RMS error", yscale = log10,
    title = "RMS error for different types polynomial order")
lines!(orders,E_Big,color=:red,linewidth=7,label="Regular")
scatter!(orders,E_Big,color=:blue)
lines!( norders,Eanti,color=:greenyellow,linewidth=3,label="Antisymmetric")
scatter!(norders,Eanti,color=:black)
fig[1, 2] = Legend(fig, ax, "Graphs", framevisible = false)
fig
```

Out[12]:



It is clear that error gets much much smaller for using the antisymmetric polynomial provided the same number of monomials are being used. Therefore, the antisymmetric polynomial approximation for odd functions reduces the computational cost for same amount of error produced in approximation.

Problem 4

- Calculate (manually, not numerically) the derivative of the approximating function $f(c,x) = \sum_p c_p x^p$. Define a new function $g(d,x) = \sum_p d_p x^p$ which depends on a different set of coefficients $|d\rangle$. Set $g(d,x) = f'(c,x)$, and solve for the coefficients $|d\rangle$ as a function of the coefficients $|c\rangle$. Since the derivative is a linear operation, the relation between $|d\rangle$ and $|c\rangle$ can be expressed via a linear operator, the derivative matrix D: $|d\rangle = D|c\rangle$. Calculate D.

Solution:

Let,

$$f(c,x) = \sum_{i=0}^{p} c_i x^i$$

$$g(d,x) = \sum_{j=0}^{p-1} d_j x^j$$

Now we also define

$$g(d,x) = f'(c,x)$$

Explictly the equation becomes:

$$\sum_{i=1}^{p} i\, x^{i-1}\, c_i = \sum_{j=0}^{p-1} x^j d_j \tag{1}$$

$$\implies \sum_{i=0}^{p-1} (i+1)\, x^i\, c_{i+1} = \sum_{i=0}^{p-1} x^i d_i \tag{2}$$

From the previous equation it is clear that $d_i = (i+1)c_{i+1}$ from $i = 0, \cdots p-1$. In matrix format we can write the derivation function in the following fashion.

$$|d\rangle = D|c\rangle \tag{3}$$

$$\begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{p-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & p \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_p \end{pmatrix} \tag{4}$$

Hence,

$$D = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & p \end{pmatrix}$$

- Calculate D numerically, and test your code by comparing (1) approximating $sin(x)$ and calculating the derivative via D, and (2) approximating $sin'(x) = cos(x)$. (Note that $cos(x)$ cannot be approximated well by antisymmetric polynomials, i.e. don't use your code from task 3 above.)

Solution:

Using `LinearAlgebra` package we can easily create D matrix numerically using the inline function `DMatrix` in the following fashion for any order p.

In [13]:
```
using LinearAlgebra
```

In [14]:
```
DMatrix(p) = hcat(zeros(p), diagm(1:p))
```

Out[14]:
```
DMatrix (generic function with 1 method)
```

In [15]:
```
# simple Demo for DMatrix function for order p = 10
D10 = DMatrix(10)
```

Out[15]:
```
10×11 Matrix{Float64}:
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0   0.0
 0.0  0.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0   0.0
 0.0  0.0  0.0  3.0  0.0  0.0  0.0  0.0  0.0  0.0   0.0
 0.0  0.0  0.0  0.0  4.0  0.0  0.0  0.0  0.0  0.0   0.0
 0.0  0.0  0.0  0.0  0.0  5.0  0.0  0.0  0.0  0.0   0.0
 0.0  0.0  0.0  0.0  0.0  0.0  6.0  0.0  0.0  0.0   0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  7.0  0.0  0.0   0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  8.0  0.0   0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  9.0   0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  10.0
```

- **(1)** First we are going to find approximate $sin(x)$ using high order value, then use the D matrix to calculate the $cos(x)$ function. In problem 1 we have already computed the approximated polynomial for target function $sin(x)$.

In [16]:
```
# we have calculated the coefficients for sin(x) in b vector
# therefore we need to multiply the b vector with DMatrix with appropriate length
d = DMatrix(length(b)-1)*b
# f'(d,x) will compute the derivative of sin(x) approximate function
f'(d,x) = sum(d[p+1]* x^p for p in 0:(pmax-1));
```

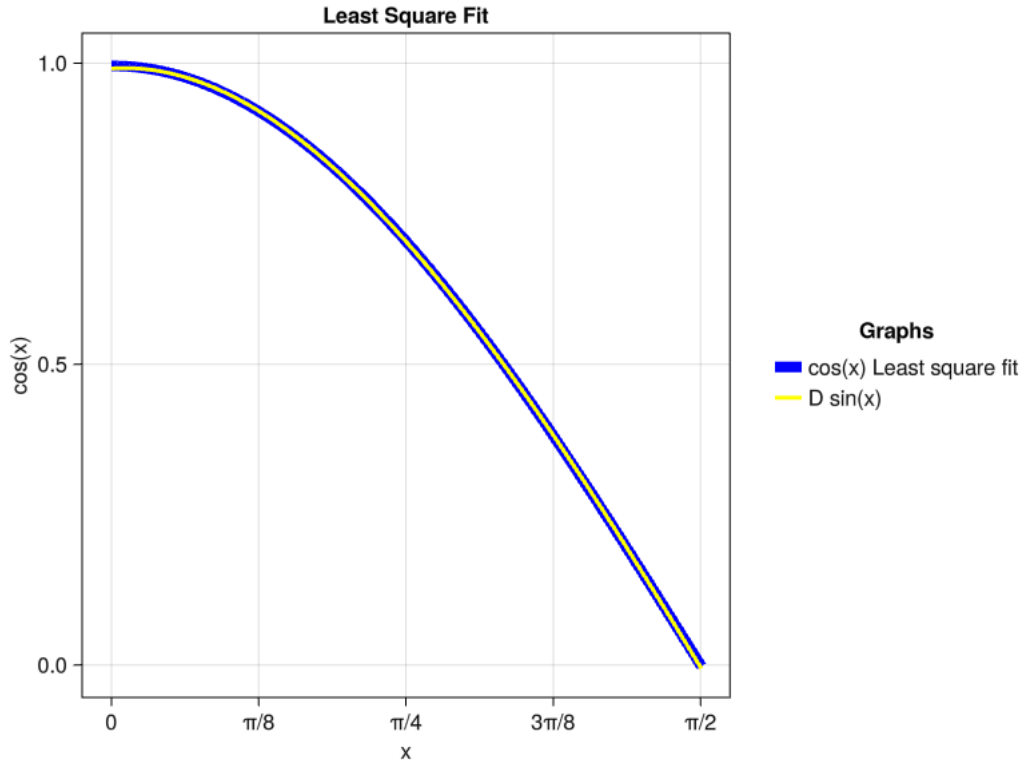
- **(2)** Next we are going to find approximate function of $cos(x)$

In [17]:
```
gstart(x) = cos(x)
yc = gstart.(x);
bc = (A'*A) \ (A' * yc)
g(bc, x) = sum(bc[p+1]* x^p for p in 0:pmax);
```

In [18]:
```
fig = Figure()
ax=Axis(fig[1,1], xlabel="x", ylabel="cos(x)", title = "Least Square Fit",
        xticks = (0:π/8:π/2, ["0", "π/8", "π/4", "3π/8", "π/2"]))
xx = LinRange(0, π/2, 100)

# calculation of the values of cos(x) using D*sin(x)
yy = [f'(d,x) for x in xx]
# calculation fo the approximate function by using linear square fit
yy2 = [g(bc, x) for x in xx]

lines!(xx,yy2,color=:blue,linewidth=8,label="cos(x) Least square fit")
lines!(xx, yy,color=:yellow,linewidth=3,label="D sin(x)")
```

```
fig[1, 2] = Legend(fig, ax, "Graphs", framevisible = false)
fig
```

Out[18]:



Acknowledgements

I didn't collaborate with anyone in this project.