Exercise 1. Use MATLAB to evaluate the second order accurate approximation

$$u''(x) \approx \frac{u(x+h) + u(x-h) - 2u(x)}{h^2}$$

for $u(x) = \sin x$ and $x = \pi/6$. Try $h = 10^{-1}, 10^{-2}, \ldots, 10^{-16}$, and make a table of values of h, the computed finite difference quotient, and the error. Explain your results.

Solution 1. The table of values can be found in the second part of the document with my code, but the major realization is that the error appears to decrease up until $h = 10^{-4}$ after which the error increases. For values of h smaller than $h = 10^{-7}$ the approximation essentially breaks and is no where close to the true value. This is likely due to rounding error which depend on both h and our choice of $x = \pi/6$ and how close they are to machine ϵ. Regardless, we see that our error in minimized somewhere near 10^{-4}.

Exercise 2. Use the formula in the previous exercise with $h = 0.2$, $h = 0.1$, and $h = 0.05$ to approximate $u''(x)$, where $u(x) = \sin x$ and $x = \pi/6$. Use one step of Richardson extrapolation, combining the results from $h = 0.2$ and $h = 0.1$, to obtain a higher order accurate approximation. Do the same with the results from $h = 0.1$ and $h = 0.05$. Finally do a second step of Richardson extrapolation, combining the two previously extrapolated values, to obtain a still higher order accurate approximation. Make a table of the computed results and their errors. What do you think is the order of accuracy after one step of Richardson extrapolation? How about after two?

Solution 2. Both tables are also in the code section of the document. My naive guess is that the order after one step of Richardson extrapolation is that the order is around $O(h^3)$ since the original centered difference approximation we've used is of order $O(h^2)$. I'd also guess naively that after two steps of Richardson extrapolation we have order $O(h^4)$.

Exercise 3. Using Taylor series, derive the error term for the approximation

$$u'(x) \approx \frac{1}{2h}[-3u(x) + 4u(x+h) - u(x+2h)].$$

Solution 3. We begin by writing out the Talor approximations for $u(x+h)$ and $u(x+2h)$

$$\begin{aligned} u(x+h) &= u(x) + u'(x)h + u''(x)\frac{h^2}{2!} + u'''(x)\frac{h^3}{3!} + u''''(x)\frac{h^4}{4!} + O(h^5) \\ u(x+2h) &= u(x) + 2u'(x)h + 4u''(x)\frac{h^2}{2!} + 8u'''(x)\frac{h^3}{3!} + 16u''''(x)\frac{h^4}{4!} + O(h^5) \end{aligned}$$

From here, it follows that

$$-3u(x) + 4u(x+h) - u(x+2h) = 2u'(x)h - 4u'''(x)\frac{h^3}{3!} + O(h^4),$$

so that

$$\begin{aligned} \frac{-3u(x) + 4u(x+h) - u(x+2h)}{2h} &= u'(x) + 2u'''(x)\frac{h^2}{3!} + O(h^3) \\ &= u'(x) + O(h^2). \end{aligned}$$

Exercise 4. Consider a forward difference approximation for the second derivative of the form

$$u''(x) \approx Au(x) + Bu(x+h) + Cu(x+2h).$$

Use Taylor's theorem to determine the coefficients A, B, and C that give the maximal order of accuracy and determine what this order is.

Solution 4. Using the Taylor approximations of $u(x+h)$ and $u(x+2h)$ from Exercise 2 as a template, we see that

$$\begin{aligned}Au(x) + Bu(x+h) + Cu(x+2h) = {}&(A+B+C)u(x) + (B+2C)u'(x)h \\ &+ (B/2 + 4C/2)u''(x)h^2 \\ &+ (B/6 + 8C/6)u'''(x)h^3 + O(h^4)\end{aligned}$$

This reduces to a system of equations which ideally satisfy

$$\begin{aligned}A + B + C &= 0 \\ B + 2C &= 0 \\ B/2 + 2C &= 1\end{aligned}$$

and additionally if possible $B/6 + 4C/3 = 0$. Starting with the second equation, we observe that $B = -2C$. Plugging this into the third equation, we have that $-C + 2C = 1$, so that $C = 1$ and $B = -2$. By the first equation, it follows $A = -B - C$, so $A = 1$. This leaves us with solution

$$A = 1, B = -2, C = 1.$$

We can additionally test the equation for the coefficients of the $O(h^3)$ term and see

$$B/6 + 4C/3 = -1/3 + 4/3 = 1 \neq 0.$$

Therefore, we have that

$$\frac{u(x) - 2u(x+h) + u(x+2h)}{h^2} = u''(x) + O(h).$$

Exercise 5. Consider the two-point boundary value problem

$$u'' + 2xu' - x^2 u = x^2, \quad u(0) = 1, \quad u(1) = 0.$$

Let $h = 1/4$ and explicitly write out the difference equations, using centered differences for all derivatives.

Solution 5. We'll begin by using the centered differences for both derivatives, so that

$$\begin{aligned} u'(x) &\approx \frac{u(x+h) - u(x-h)}{2h} \\ u''(x) &\approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}, \end{aligned}$$

This allows us to write that

$$\frac{u(x+h) - 2u(x) + u(x-h)}{h^2} + x\left(\frac{u(x+h) - u(x-h)}{h}\right) - x^2 u(x) = x^2$$

Re-arranging this equation, we see that

$$\left(\frac{1}{h^2} - \frac{x}{h}\right) u(x-h) - \left(x^2 + \frac{2}{h^2}\right) u(x) + \left(\frac{1}{h^2} + \frac{x}{h}\right) u(x+h) = x^2.$$

We can now explicitly discretize this problem using step size $h = 1/4$ with grid points $x_0 = 0$, $x_1 = 1/4$, $x_2 = 1/2$, $x_3 = 3/4$, $x_4 = 1$. Starting with x_1, this reduces to

$$\begin{aligned} (16 - 4x_1)u(x_0) - (x_1^2 + 32)u(x_1) + (16 + 4x_1)u(x_2) &= x_1^2 \\ &\Longrightarrow \\ -(x_1^2 + 32)u(x_1) + (16 + 4x_1)u(x_2) &= x_1^2 + (16 - 4x_1), \end{aligned}$$

where we've used that $u(x_0) = u(0) = 1$. The equations for $x_2 = 1/2$ are given by

$$(16 - 4x_2)u(x_1) - (x_2^2 + 32)u(x_2) + (16 + 4x_2)u(x_3) = x_2^2$$

Now moving onto the $x_3 = 1$, we have that

$$\begin{aligned} (16 - 4x_3)u(x_2) - (x_3^2 + 32)u(x_3) + (16 + 4x_3)u(x_4) &= x_3^2 \\ &\Longrightarrow \\ (16 - 4x_3)u(x_2) - (x_3^2 + 32)u(x_3) &= x_3^2 \end{aligned}$$

where we've used that $u(x_4) = u(1) = 0$. This leaves us a system of three equations with three unknowns $u(x_1)$ $u(x_2)$, and $u(x_3)$. I've neglected to plug in the values of the x_i for simplicity.

Exercise 6. A rod of length 1 meter has a heat source applied to it and it eventually reaches a steady-state where the temperature is not changing. The conductivity of the rod is a function of position x and is given by $c(x) = 1 + x^2$. The left end of the rod is held at a constant temperature of 1 degree. The right end of the rod is insulated so that no heat flows in or out from that end of the rod. This problem is described by the boundary value problem:

$$\frac{d}{dx}\left((1+x^2)\frac{du}{dx}\right) = f(x), \quad 0 \le x \le 1,$$
$$u(0) = 1, \quad u'(1) = 0.$$

(a) Write down a set of difference equations for this problem. Be sure to show how you do the differencing at the endpoints. [Note: It is better **not** to rewrite $\frac{d}{dx}((1+x^2)\frac{du}{dx})$ as $(1+x^2)u''(x) + 2xu'(x)$; leave the equation in the form above.]

(b) Write a MATLAB code to solve the difference equations. You can test your code on a problem where you know the solution by choosing a function $u(x)$ that satisfies the boundary conditions and determining what $f(x)$ must be in order for $u(x)$ to solve the problem. Try $u(x) = (1-x)^2$. Then $f(x) = 2(3x^2 - 2x + 1)$.

(c) Try several different values for the mesh size h. Based on your results, what would you say is the order of accuracy of your method?

Solution 6. (a) In what follows, we'll assume that we take a mesh of the interval $[0, 1]$ of size h with grid points $x_j = jh$ with $x_M = 1$. We also set $1 + x^2 = p(x)$. We begin by approximating the outside derivative at a grid point x_j as

$$\frac{d}{dx}\left(p(x)\frac{du}{dx}\right) \approx \frac{p(x_{j+\frac{1}{2}})\frac{du}{dx}(x_{j+\frac{1}{2}}) - p(x_{j-\frac{1}{2}})\frac{du}{dx}(x_{j-\frac{1}{2}})}{h}.$$

We can then estimate $\frac{du}{dx}$ using a centered difference approximation, so that

$$\frac{du}{dx}(x_{j-\frac{1}{2}}) \approx \frac{u(x_j) - u(x_{j-1})}{h}$$
$$\frac{du}{dx}(x_{j+\frac{1}{2}}) \approx \frac{u(x_{j+1}) - u(x_j)}{h}.$$

Combining these expressions, we get the approximation

$$\frac{p(x_{j+\frac{1}{2}})[u(x_{j+1}) - u(x_j)] - p(x_{j-\frac{1}{2}})[u(x_j) - u(x_{j-1})]}{h^2} = f(x_j).$$

In order to accommodate for our boundary conditions, we'll make some adjustments to equations for the first and last end points. Accounting for the left boundary condition i.e. that $u(0) = 1$, we change the first equation i.e. when $j = 1$ so that

$$\frac{p(x_{1+\frac{1}{2}})[u(x_2) - u(x_1)] - p(x_{\frac{1}{2}})[u(x_1) - u(0)]}{h^2} = f(x_1).$$

This shows that our first equation is modified so that

$$\frac{p(x_{1+\frac{1}{2}})[u(x_2)-u(x_1)] - p(x_{\frac{1}{2}})u(x_1)}{h^2} = f(x_1) - \frac{p(x_{\frac{1}{2}})}{h^2}$$

since $u(0)=1$. In order to accommodate the Neumann boundary condition at the right end point, we will add a second order backward difference approximation of $u'(1)=0$ to our current equations

$$\frac{3u(1) - 4u(1-h) + u(1-2h))}{2h} = 0.$$

Writing this out as a system of equations where $\mathbf{U} = [u(x_1), u(x_2), ..., u(x_M) = u(1)]^T$ and $\mathbf{F} = [f(x_1) - p(x_{\frac{1}{2}})/h^2, f(x_2), \dots, f(x_{M-1}), 0]$ with matrix

$$\mathbf{A} = \frac{1}{h^2}\begin{pmatrix} a_1 & b_1 & 0 & 0 & \cdots & \\ b_1 & a_2 & b_2 & 0 & \cdots & \\ 0 & b_2 & a_3 & b_3 & & \\ \vdots & & \ddots & \ddots & \ddots & \\ & & & b_{M-2} & a_{M-1} & b_{M-1} \\ & & & h/2 & -2h & 3h/2 \end{pmatrix},$$

where the coefficients a_j and b_j are given by $a_j = -p(x_{j+\frac{1}{2}}) - p(x_{j-\frac{1}{2}})$ and $b_j = p(x_{j+\frac{1}{2}})$. Finding our solution now reduces to finding the solution of the system of equations

$$\mathbf{AU} = \mathbf{F}.$$

(b) All code implementing this is found at the end of the document.
(c) Halving the input h appears to lead to a 4-fold drop in the error, so I would estimate that this method has error on the order of $O(h^2)$ (according to the infinity norm) which is consistent with the choice to use second order approximations in the derivation of the equations above.

HW-1-Code-Figgins

January 15, 2021

[1]:
```
using Plots, DataFrames, LinearAlgebra, LaTeXStrings, Latexify
```

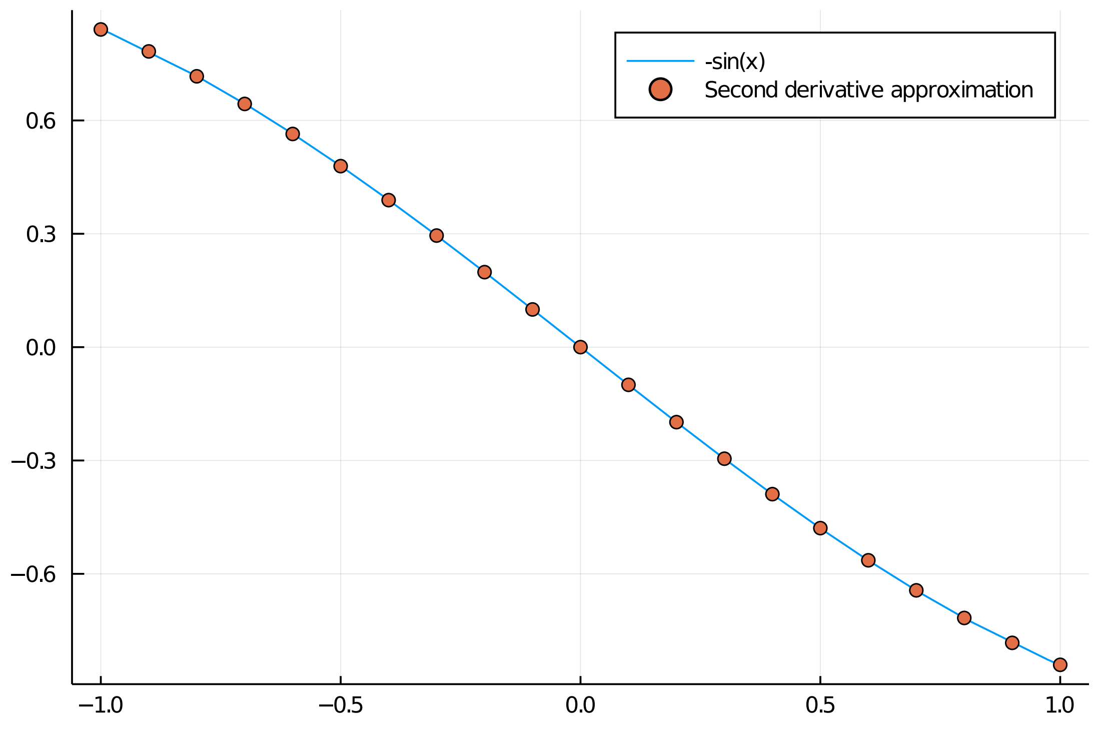
0.1 Exercise 1

[2]:
```
function second_deriv(u::Function, x, h)
     (u(x + h) + u(x - h) - 2*u(x)) / h^2
end
```

[2]:
```
second_deriv (generic function with 1 method)
```

[3]:
```
xs = -1:0.1:1
h = 0.1

# We can test our approximation using
# that second derivative of sin is - sin
sin_xx = [ second_deriv(sin, x, h) for x in xs]

plot(x -> -1*sin(x), -1, 1, label = "-sin(x)")
scatter!(xs, sin_xx, label = "Second derivative approximation")
```

[3]:



```
[4]: hs = [10.0^(-k) for k in 1:16]
     x =  /6

     FDQ = [second_deriv(sin, x, h) for h in hs]
     error = abs.(FDQ .+ sin(x)) # True value is -sin(x)
```

```
[4]: 16-element Array{Float64,1}:
        0.00041652780258211175
        4.166652633530443e-6
        4.167449668690537e-8
        3.0387354299499236e-9
        1.1515932100691906e-6
        6.657201129195434e-5
        0.0003996389186794458
        0.6102230246251563
      111.52230246251564
        0.49999999999999994
        0.49999999999999994
        0.49999999999999994
        1.1102230246751564e10
        1.1102230246246565e12
        0.49999999999999994
        1.1102230246251566e16
```

[5]:

```
df1 = DataFrame(h = hs, FDQ = FDQ, Error = error)
latexify(df1)
```

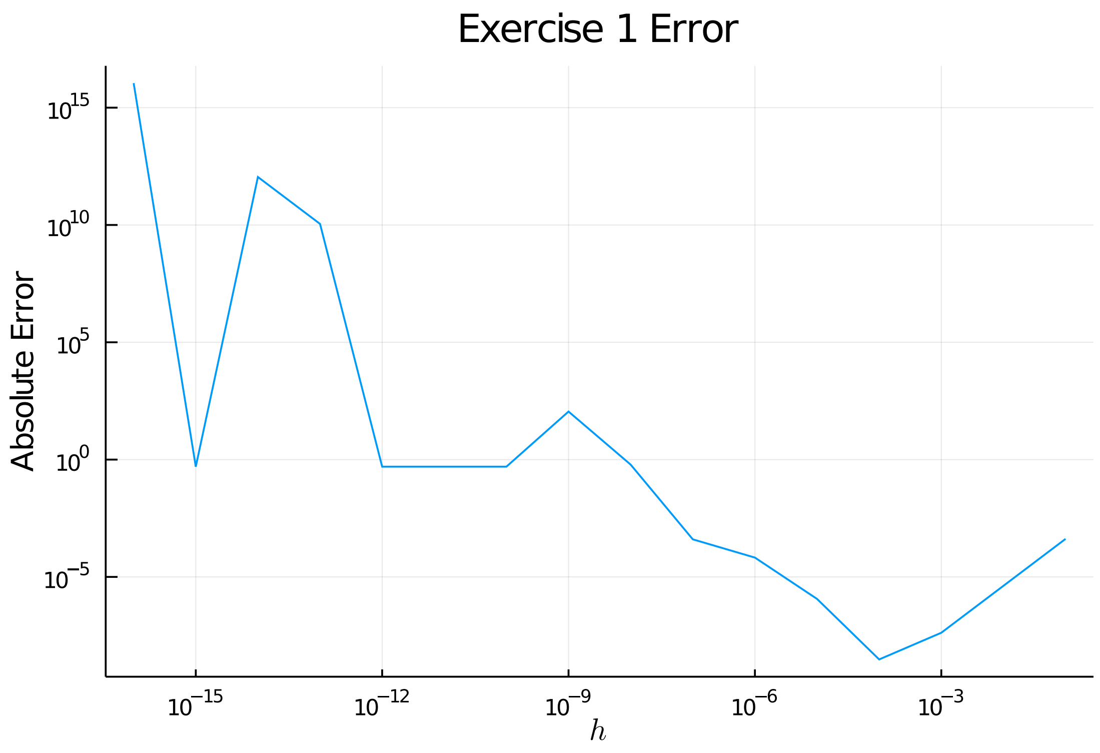
[5]:

h	FDQ	Error
0.1	-0.49958347219741783	0.00041652780258211175
0.01	-0.4999958333473664	$4.166652633530443e-6$
0.001	-0.49999995832550326	$4.167449668690537e-8$
0.0001	-0.4999999969612645	$3.0387354299499236e-9$
$1.0e-5$	-0.50000115159321	$1.1515932100691906e-6$
$1.0e-6$	-0.499933427988708	$6.657201129195434e-5$
$1.0e-7$	-0.4996003610813205	0.0003996389186794458
$1.0e-8$	-1.1102230246251563	0.6102230246251563
$1.0e-9$	111.02230246251564	111.52230246251564
$1.0e-10$	0.0	0.49999999999999994
$1.0e-11$	0.0	0.49999999999999994
$1.0e-12$	0.0	0.49999999999999994
$1.0e-13$	$1.1102230246251564e10$	$1.1102230246751564e10$
$1.0e-14$	$-1.1102230246251565e12$	$1.1102230246246565e12$
$1.0e-15$	0.0	0.49999999999999994
$1.0e-16$	$-1.1102230246251566e16$	$1.1102230246251566e16$

[6]:

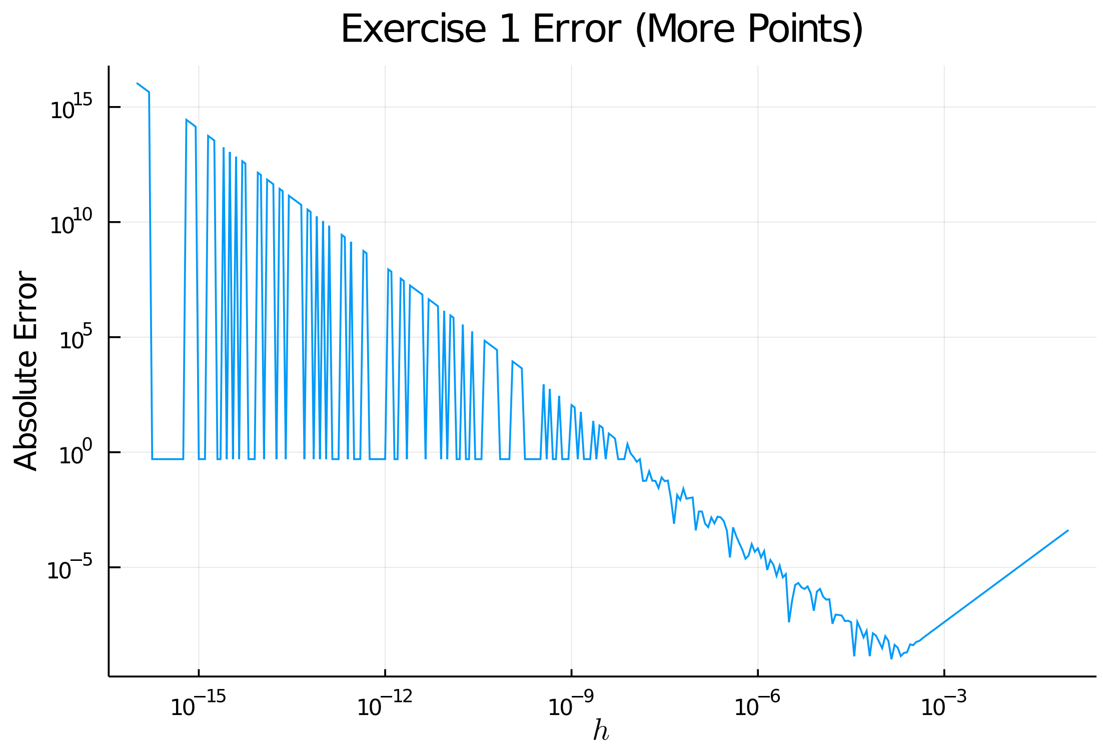
```
plot(hs, error, scale = :log10,
    label = false,
    xlabel = L"h",
    ylabel = "Absolute Error",
    title = "Exercise 1 Error")
```

[6]:



```
[7]: hs = [10.0^(-k) for k in 1:0.05:16]
     x =  /6

     FDQ = [second_deriv(sin, x, h) for h in hs]
     error = abs.(FDQ .+ sin(x)) # True value is -sin(x)

     plot(hs, error, scale = :log10,
         label = false,
         xlabel = L"h",
         ylabel = "Absolute Error",
         title = "Exercise 1 Error (More Points)")
```

[7]:



There are significant issues once we go smaller than $h \approx 10e - 7$

0.2 Exercise 2

```
[8]: function RE(u, x, h, approx, order = 2, t = 2)
         ( t^order * approx(u, x, h / t) - approx(u, x, h) ) / (t^order - 1)
     end
```

```
[8]: RE (generic function with 3 methods)
```

[9]:
```
hs = [0.05, 0.1,0.2]
x =  /6

# No Richardson extrapolation
raw_FDQ = [second_deriv(sin, x, h) for h in hs]
raw_error = raw_FDQ .+ sin(x) # True value is -sin(x)
```

[9]:
```
3-element Array{Float64,1}:
 0.00010415798651314256
 0.00041652780258211175
 0.0016644460310434872
```

[10]:
```
# One step of Richardson extrapolation
RE_1_FDQ = [RE(sin, x, h, second_deriv) for h in hs]
RE_1_error = RE_1_FDQ .+ sin(x) # True value is -sin(x)
```

[10]:
```
3-element Array{Float64,1}:
 2.170097102016655e-9
 3.471449017133921e-8
 5.550597616532649e-7
```

[11]:
```
## RE on the RE
RE_1(u, x, h) = RE(u, x, h, second_deriv)

# Two steps of Richardson extrapolation
RE_2_FDQ = [ RE(sin, x, h, RE_1, 4,2) for h in hs]
RE_2_error = RE_2_FDQ .+ sin(x) # True value is -sin(x)
```

[11]:
```
3-element Array{Float64,1}:
 -5.636047184509607e-13
  4.709010958947601e-13
  2.4805379972292485e-11
```

[12]:
```
df2 = DataFrame(h = hs,
                NoRE = raw_FDQ,
                Richardson1 = RE_1_FDQ,
                Richardson2 = RE_2_FDQ)
df2[1,[3,4]] .= NaN
latexify(df2)
```

[12]:

h	NoRE	Richardson1	Richardson2
0.05	−0.4998958420134868	*NaN*	*NaN*
0.1	−0.49958347219741783	−0.4999999652855098	−0.49999999999952904
0.2	−0.49833555396895646	−0.4999994449402383	−0.49999999997519456

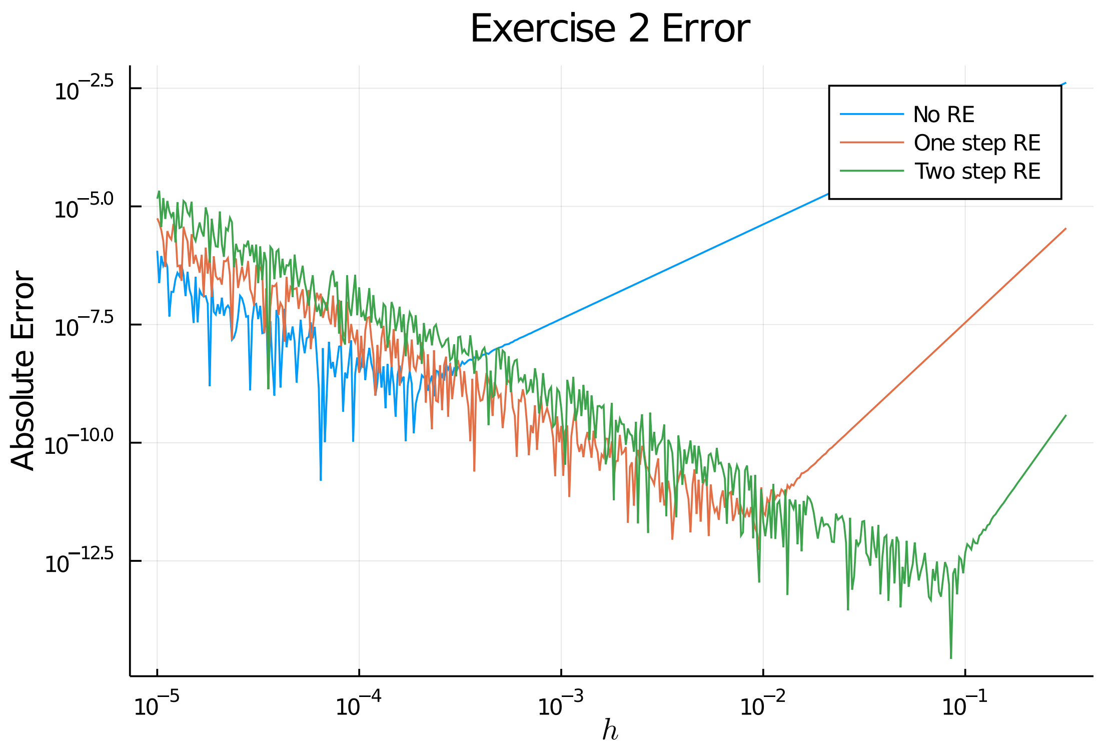
[13]:
```
df2Errors = DataFrame(h = hs,
                NoRE_Error = raw_error,
                Richardson1_Error = RE_1_error,
                Richardson2_Error = RE_2_error)
df2Errors[1,[3,4]] .= NaN
latexify(df2Errors)
```

[13]:

h	NoRE_Error	Richardson1_Error	Richardson2_Error
0.05	0.00010415798651314256	NaN	NaN
0.1	0.00041652780258211175	$3.471449017133921e-8$	$4.709010958947601e-13$
0.2	0.0016644460310434872	$5.550597616532649e-7$	$2.4805379972292485e-11$

[14]:
```
hs = [10.0^(-k) for k in 0.5:0.01:5]

# No Richardson extrapolation
raw_FDQ = [second_deriv(sin, x, h) for h in hs]
raw_error = abs.(raw_FDQ .+ sin(x)) # True value is -sin(x)

# One step of Richardson extrapolation
RE_1_FDQ = [RE(sin, x, h, second_deriv) for h in hs]
RE_1_error = abs.(RE_1_FDQ .+ sin(x)) # True value is -sin(x)

# Two steps of Richardson extrapolation
RE_2_FDQ = [ RE(sin, x, h, RE_1, 4,2) for h in hs]
RE_2_error = abs.(RE_2_FDQ .+ sin(x)) # True value is -sin(x)

plot(hs, raw_error, scale = :log10,
    label = "No RE",
    xlabel = L"h",
    ylabel = "Absolute Error",
    title = "Exercise 2 Error")

plot!(hs, RE_1_error, scale = :log10,
    label = "One step RE")


plot!(hs, RE_2_error, scale = :log10,
    label = "Two step RE")
```

[14]:



0.3 Exercise 6

[15]:
```
M = 25 # Number of grid points
h  = 1/M # Step size
```

[15]:
```
0.04
```

[16]:
```
## Guess $f$ given a $u$

function make_matrix(M)
   # Step size
   h = 1/M

    # Xjs of interest
    xjs = [j*h for j in 1:M]

    # Defining diagonal
    diagA = [ -(1 + (xj + h/2)^2) - (1 + (xj - h/2 )^2)  for xj in xjs]
    diagB = [  (1 + (xj + h/2)^2)  for xj in xjs[1:M-1]]

    ##Building Matrix
    A = Array(Tridiagonal(diagB, diagA, diagB))
    A[end, [M, M-1, M-2]] .= 3*h/2, -2*h, h/2
    A = A ./ h^2

    return A
end

# Generate MxM approximation
A = make_matrix(M)
```

[16]:
```
25×25 Array{Float64,2}:
 -1252.5     627.25      0.0       0.0   …     0.0       0.0       0.0
   627.25  -1258.5     631.25      0.0          0.0       0.0       0.0
     0.0     631.25  -1268.5     637.25         0.0       0.0       0.0
     0.0       0.0     637.25  -1282.5          0.0       0.0       0.0
     0.0       0.0       0.0     645.25         0.0       0.0       0.0
     0.0       0.0       0.0       0.0   …     0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0   …     0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0   …     0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0          0.0       0.0       0.0
     0.0       0.0       0.0       0.0   …     0.0       0.0       0.0
     0.0       0.0       0.0       0.0       1131.25      0.0       0.0
     0.0       0.0       0.0       0.0      -2308.5    1177.25      0.0
     0.0       0.0       0.0       0.0       1177.25  -2402.5    1225.25
     0.0       0.0       0.0       0.0         12.5     -50.0      37.5
```

[17]:
```
# Define test function
u(x) = (1 - x)^2
```

[17]:
```
u (generic function with 1 method)
```

[18]:
```
# Define true solution for test function
true_f(x) = 2*(3*x^2 - 2*x + 1)
```

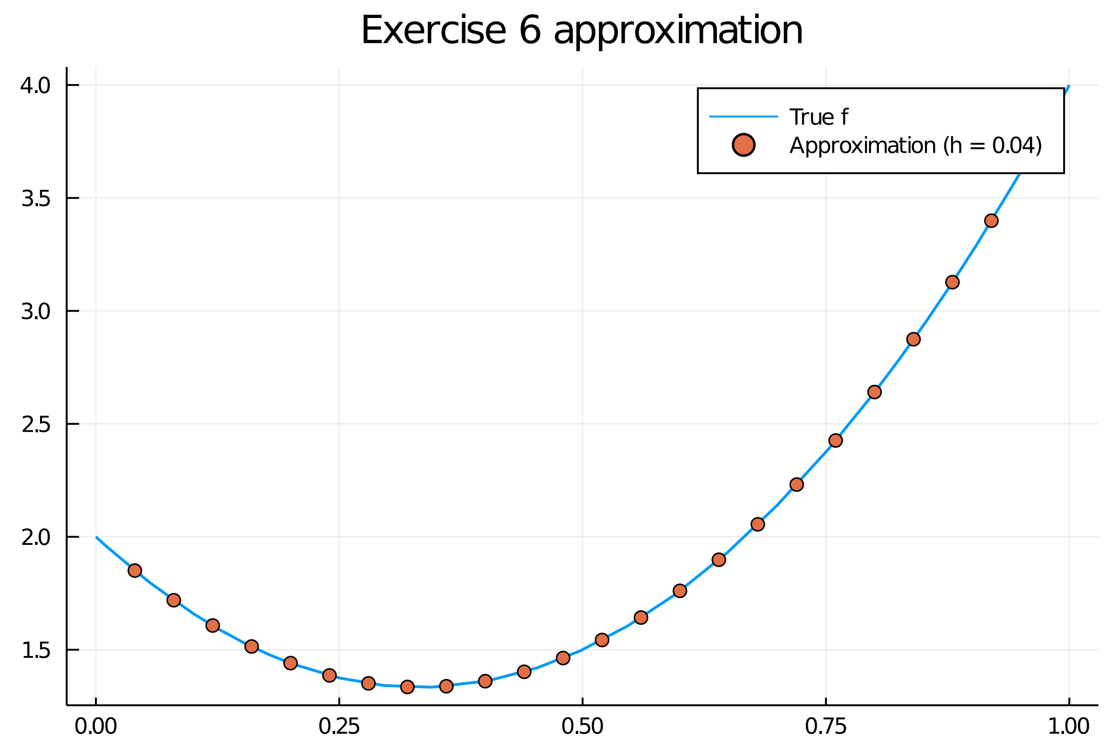
[18]:
```
true_f (generic function with 1 method)
```

[19]:
```
function guess_f(u, M)
    h = 1 / M
    # Discretize u
    xjs = [j*h for j in 1:M]
    U = u.(xjs)

    # Approximate the F vector
    F_approx = make_matrix(M)*U

    # End points will be funky due to boundary conditions

    ## First end point has extra term
    F_approx[1] += (1+ (h/2)^2)/h^2

    ## Last is a derivative and isn't needed
    return F_approx[1:(M-1)]
end

f_approx = guess_f(u, M)
```

[19]:
```
24-element Array{Float64,1}:
 1.850400000000036
 1.719199999999546
 1.607199999999807
 1.5144000000001938
 1.44079999999974
 1.3864000000000942
 1.3511999999999489
 1.335200000000043
 1.3383999999999787
 1.3608000000001539
 1.4024
 1.463200000000029
 1.5432000000000414
 1.642400000000066
 1.7607999999999038
 1.8984000000000094
 2.0552000000000845
 2.2311999999999443
 2.426399999999987
 2.640800000000059
 2.8743999999999517
 3.1271999999999984
 3.39920000000002
 3.690399999999985
```

[20]:
```
plot(true_f, 0, 1,
    label = "True f",
    linewidth = 1.5)

scatter!([j*h for j in 1:(M-1)],
    f_approx,
    label = "Approximation (h = $h)",
    title = "Exercise 6 approximation")
```

[20]:



[21]:
```
function get_infinity_error(M)
    # Discretize True Solution
    xjs = [j/M for j in 1:(M-1)]
    true_f_val = true_f.(xjs)
    f_approx = guess_f(u, M)

    return maximum( abs.(true_f_val .- f_approx) )
end
```

[21]:
```
get_infinity_error (generic function with 1 method)
```

[22]:
```
get_infinity_error(1000)
```

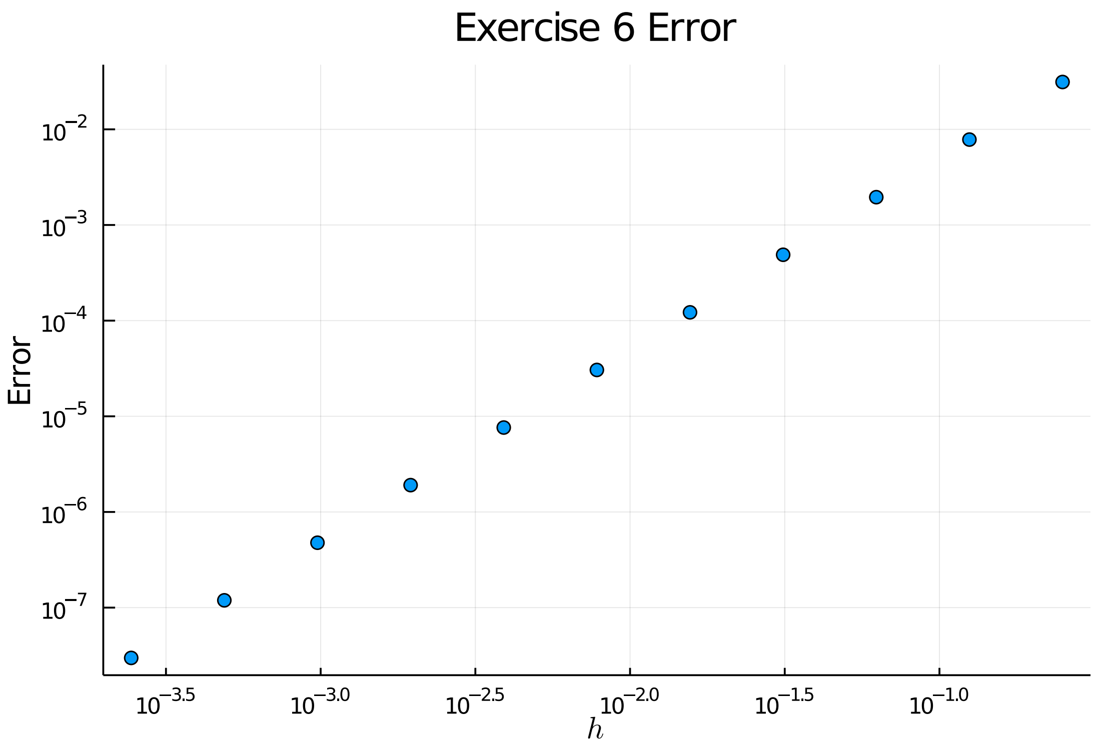
[22]:
```
5.007842034387977e-7
```

[23]:
```
M_vec = [2^k for k in 2:12]

error_vec = []
for M in M_vec
    push!(error_vec, get_infinity_error(M))
end
```

Here's a simple attempt to plot the infinity norm error on our approximations. This is difficult to do for very small values of h since the size of the linear system grows as h increases.

[24]:
```
scatter(1 ./ M_vec, error_vec, scale = :log10,
    xlabel = L"h",
    ylabel = "Error",
    title = "Exercise 6 Error",
    label = false)
```

[24]:



[25]:
```
error_vec
```

[25]:
```
11-element Array{Any,1}:
 0.03125
 0.0078125
 0.001953125
 0.00048828125
 0.0001220703125
 3.0517578125e-5
 7.62939453125e-6
 1.9073486328125e-6
 4.76837158203125e-7
 1.1920928955078125e-7
 2.9802322387695312e-8
```

[26]:
```
./ M_vec
```

[26]:
```
11-element Array{Float64,1}:
 0.25
 0.125
 0.0625
 0.03125
 0.015625
 0.0078125
 0.00390625
 0.001953125
 0.0009765625
 0.00048828125
 0.000244140625
```