In science and engineering an important goal is to become a skilled practitioner by doing it yourself. The computer labs provide computational experience related to the analytic theory presented in the lectures.

Lagrange Polynomial

A striking property of polynomials evident when studying calculus is that the integrals and derivatives of polynomials are again polynomials. This along with the fact that any continuous function can be approximated arbitrarily well by a suitable polynomial suggests polynomials as fundamental building blocks for numerical calculations. In practice, therefore, we need efficient ways to represent and evaluate polynomials on the computer.

In this lab you will find the interpolating polynomial p of degree 4 passing through the points

$$(x_i, \log x_i) \quad \text{where} \quad x_i = 1 + 0.5i \quad \text{and} \quad i = 1, \ldots, 5, \tag{8.1}$$

evaluate $p(2.25)$ and then compute the error $E = |p(2.25) - \log 2.25|$. Note in this example that $\log x$ represents the natural logarithm.

There is frequently a trade off between complexity, efficiency and correctness when writing computer programs:

- The simpler an algorithm is the less likely it is to be the most efficient.
- The more complex an algorithm is the more likely the code has a blunder which produces incorrect answers.

Producing incorrect answers no matter how efficiently can lead to trouble, because in the real world you would not be programming a computer in the first place if the answer were already known. For this reason, anything that helps ensure correctness is a priority.

Development methodologies such as Agile, DevOps, RAD, Scrum, Prototyping and Waterfall represent attempts to produce software at lowest cost with the best quality in the shortest time. Ideally what constitutes a deliverable product guides the assessment of risk in the development process and informs what methodology to use. For example, the risk related to bugs in a video game or website may be less than a banking system or industrial control program.

In the present context program bugs affect only a course grade and even then not so much. However, depending on the domain of application the consequence of getting an incorrect result may be greater as well as more difficult to detect. In many fields of human endeavor “a wrong answer is worse than no answer.” When having the right answer makes a difference “getting the wrong answer and running with it causes even more problems.”

I found the above quotes in a blog by Laurie Barth about the causes and dangers of wrong answers at

```
https://tenmilesquare.com/resources/technology-leadership/a-wrong-
answer-is-worse-than-no-answer/
```

Please do not spend too much time, if any, reading this link during class.

In this lab the program we shall write uses the simplest technique available to help ensure getting a correct answer. In particular, the Lagrange polynomial basis—though not the most efficient—leads to a diagonal Vandermonde matrix that makes solving for the coefficients trivial. As a result, there is little chance to make a blunder.

Once a working but slow algorithm is produced, a more efficient method can be developed keeping the original program as a reference to ensure the answers are consistent and correct. For example, techniques based on Newton's divided difference formula may be more efficient in cases where polynomials of varying degrees are needed to satisfy some error criterion. While doing things in the simplest way first and later refining the program is similar to prototyping, this technique is so common in scientific computing that such terminology is seldom mentioned.

Consider the unique polynomial of degree $n-1$ passing through n points given by

$$(x_i, y_i) \qquad \text{where} \qquad i = 1, \ldots, n.$$

Here the values of x_i are assumed distinct; otherwise, no function would pass through the points at all. The corresponding Lagrange basis functions $\ell_k(t)$ are given by

$$\ell_k(t) = \prod_{i \neq k} \frac{x_i - t}{x_i - x_k} \qquad \text{for} \qquad k = 1, \ldots, n.$$

We begin by writing code in Julia to compute these functions.

Some people experienced in with programming may already be thinking how to express $\ell_k(t)$ in their favorite language. Depending on language and personal preference, that code might be procedural and involve a `for` loop or a functional technique with vector operators. Since Julia is just-in-time compiled with built-in vector operations it can handle either paradigm well.

Assuming the values of x_i are stored in the array `x[i]`, code for the functional approach is illustrated by `ellf(k,t)` and the procedural approach by `ellp(k,t)` as follows

```julia
ellf(k,t)=prod([x[1:k-1];x[k+1:length(x)]].-t)/
          prod([x[1:k-1];x[k+1:length(x)]].-x[k])

function ellp(k,t)
    p=1
    q=1
    for i=1:length(x)
        if i!=k
            p*=x[i]-t
            q*=x[i]-x[k]
        end
    end
    return p/q
end
```

Although `ellp(k,t)` consists of more lines of code and introduces the extra variables `p` and `q`, which version is easier to understand is related more to familiarity with the paradigm rather than intrinsic differences in complexity. While we are already sunk from a performance point of view by using the Lagrange basis in the first place, it is still interesting to know how the computational efficiency of the functional version compares with the procedural implementation. Note also since both routines express the exact same calculation, they should return the same results. Computing the same thing two different ways—which we now do—helps detect bugs.

For your convenience this PDF document contains an attachment called `lagrange.jl` with machine-readable copies of `ellf(k,t)` and `ellp(k,t)`. To save this file to your local computer

- Open the side panel of your PDF viewer.
- Click on the paperclip icon.
- Right click on `lagrange.jl` and select save.

If you are following the naming convention suggested in the first lab, save the file to the directory `lab03`.

At this point open a terminal window, change to the directory where you saved the file and start Julia. Then type `include("lagrange.jl")` to load the code into the REPL. Your screen should look like

```
$ cd lab03
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.6 (2022-03-28)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> include("lagrange.jl")
pp (generic function with 1 method)

julia>
```

If instead you are treated to the lengthy error message

```
julia> include("lagrange.jl")
ERROR: SystemError:
    opening file "/x/libb/ejolson/lab03/lagrange.jl":
    No such file or directory
Stacktrace:
  [1] systemerror(p::String, errno::Int32; extrainfo::Nothing)
    @ Base ./error.jl:168
  [2] #systemerror#62
    @ ./error.jl:167 [inlined]  ...the rest omitted...
```

do not panic. This is simply Julia's way of explaining you saved the attachment in the wrong place. Make sure both Julia and `lagrange.jl` are in the `lab03` directory and try again. The commands `pwd()` and `readdir()` can be entered from the Julia REPL to figure out what is going on.

After including `lagrange.jl` test `ellf(k,t)` and `ellp(k,t)` to see if they both return the same values. Create some random data by typing `x=rand(5)` and then enter `ellf(2,0.5)` followed by `ellp(2,0.5)`. It is a good sign if the return values are the same. Please check some additional points on your own. Note that `k` must be an integer between 1 and 5 while `t` can be any valid floating-point number.

Since the random vector `x` will be different for each person the output will vary accordingly. Here is a representative example

```
julia> x=rand(5)
5-element Vector{Float64}:
 0.949333046824119
 0.5352022327558354
 0.9268708406141919
 0.29664466861532257
 0.6995775657397101

julia> ellf(2,0.5)
1.2239001449856663

julia> ellp(2,0.5)
1.2239001449856663
```

Note that both versions of $\ell_k(t)$ produce the same value.

By design the functions $\ell_k(t)$ are each polynomials of degree $n-1$ which satisfy the orthogonality property

$$\ell_k(x_j) = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k. \end{cases}$$

Thus, the Vandermonde matrix V where $V_{j,k} = \ell_k(x_j)$ is diagonal. This can be verified numerically with the commands

```
Vf=[ellf(k,x[j]) for j=1:5, k=1:5]
Vp=[ellp(k,x[j]) for j=1:5, k=1:5]
```

The output should look like

```
julia> Vf=[ellf(k,x[j]) for j=1:5, k=1:5]
5×5 Matrix{Float64}:
  1.0   0.0   0.0  -0.0  -0.0
 -0.0   1.0   0.0   0.0  -0.0
 -0.0  -0.0   1.0   0.0   0.0
  0.0  -0.0  -0.0   1.0   0.0
  0.0   0.0  -0.0  -0.0   1.0

julia> Vp=[ellp(k,x[j]) for j=1:5, k=1:5]
5×5 Matrix{Float64}:
  1.0   0.0   0.0  -0.0  -0.0
 -0.0   1.0   0.0   0.0  -0.0
 -0.0  -0.0   1.0   0.0   0.0
  0.0  -0.0  -0.0   1.0   0.0
  0.0   0.0  -0.0  -0.0   1.0
```

Although it is satisfying both `Vf` and `Vp` appear exactly the same, the fact that zero sometimes prints as `-0.0` seems weird. Note also that

```
julia> Vf[1,2]
0.0

julia> Vf[2,1]
-0.0

julia> Vf[1,2]==Vf[2,1]
true
```

indicates the zero which prints with a negative sign is equal to the zero that prints with a positive sign. Explaining what is going on is an opportunity for extra credit. If interested, please do not spend time right now, but study negative zeros at home and turn your extra-credit in next week.

Back on topic, since V is diagonal, the polynomial $p(t)$ of degree $n-1$ passing through the points (x_i, y_i) may be expressed as

$$p(t) = \sum_{k=1}^{n} y_k \ell_k(t).$$

The above sum may be coded using either a functional or procedural approach in Julia. Two possible implementations of $p(t)$ are

```
16 pf(t)=sum((k->y[k]*ellf(k,t)).(1:length(x)))
17
18 function pp(t)
19     s=0
20     for k=1:length(x)
21         s+=y[k]*ellp(k,t)
22     end
23     return s
24 end
```

Here `pf(t)` is the functional version and `pp(t)` is written in a procedural style. Note how the broadcast `.` in the functional version applies the function $k \to y_k \ell_k(t)$ to an iterator that represents the indices for k.

Having grown up with Basic, C, Fortran and Pascal, the kind of functional programming in line 16 often seems unnaturally clever to me. Fortunately, due to the level of optimization provided by the just-in-time compiler, the `for` loops in Julia typically run just as fast as vectorized code. Thus, one can switch back and forth between whatever programming style seems easiest without worrying too much about performance.

The computation requested in (8.1) can now be performed by specifying the vectors `x` and `y` as

`x=1.5:0.5:3.5` and `y=log.(x)`

and typing `pp(2.25)` and `abs(pp(2.25)-log(2.25))` to evaluate the interpolating polynomial and compute the requested error.

Submitting Your Work

Two things should be uploaded for grading:

- A program that evaluates and computes the error in the polynomial.
- The output from running that program.

To help with the items above, here is a program that does just that:

```
# soln03.jl -- evaluate and compute the error in p(2.25)

include("lagrange.jl")
x=1.5:0.5:3.5
y=log.(x)
println("p(2.25)=",pp(2.25))
println("E=",abs(pp(2.25)-log(2.25)))
```

After running the above code with

```
julia soln03.jl >soln03.out
```

convert `soln03.jl` and `soln03.out` to PDF or Postscript format and upload the converted versions of these files for grading to the course management system. Please reboot into Microsoft Windows before leaving.