Project 1

Zack Owens

6/9/2023

1.1

In [4]:
```
eps32 = eps(0f0)
```

```
1.0f-45
```

In [5]:
```
eps64 = eps(0.0)
```

```
5.0e-324
```

Meanings of these values: The method esp() according to the julia documentation is the distance between 1.0 and the next largest floating-point value of Float64. [Source (https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/)](https://docs.julialang.org/en/v1/manual/integers-and-floating-point-numbers/)

Chat GPT answer:

In Julia, eps(x) returns the difference between x and the next representable value in the floating-point number system. The behavior you described, where eps(0f0) and eps(0) yield different values, can be explained by the fact that 0f0 and 0 are represented as different types of floating-point numbers.

In Julia, the f0 suffix is used to denote single-precision floating-point numbers, also known as Float32, while the default floating-point type is double-precision, known as Float64.

The reason eps(0f0) and eps(0) yield different results is because they are based on the precision of their respective floating-point types. The Float32 type has lower precision compared to Float64, meaning it can represent smaller differences between values. Consequently, the value returned by eps(0f0) will be smaller than the value returned by eps(0).

In [11]:
```
typeof(0f0)
```

```
(Float32, Inf32)
```

In [7]:
```
typeof(0.0)
```

```
Float64
```

The reason that 0f0 has a larger value is because the values is of type float32. This esp value the smaller the value the more precise value that can be stored in the type. The reason that 0.0 is a smaller values is due to its type being a float64 which has more bytes to store the float values. This gives teh float64 more percision than a float32 allowing smaller values to be stored in smaller increments.

1.2

In [16]:
```
v = fill(eps32,2_000_000_000)
```

```
2000000000-element Vector{Float32}:
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 ⋮
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
 1.0f-45
```

In [ ]:
```
function sum_values_does_not_work(v)
    sum = 0.0
    for i in v
        sum += i
    end
    return sum
end
```

In [22]:
```
function sum_values(v)
    sum = 0.0
    for i in v
        sum += i
    end
    return sum
end
```

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

In [23]:
```
@time sum_values(v)
```

```
  2.419582 seconds (7.53 k allocations: 510.795 KiB, 0.65% compilation time)
```

```
2.802596928649634e-36
```

In [28]:
```
@time foldr(+,v)
```

```
  2.612031 seconds (4 allocations: 64 bytes)
```

```
2.3509887f-38
```

In [27]:
```
@time foldr(+,v,init=0.0)
```

```
  2.552371 seconds (33.84 k allocations: 2.340 MiB, 1.89% compilation time)
```

```
2.802596928649634e-36
```

In [20]:
```
@time reduce(+,v)
```

```
  0.633802 seconds (108.92 k allocations: 7.286 MiB, 9.50% compilation time)
```

```
2.802597f-36
```

In [21]:
```
@time sum(v)
```

```
  0.584358 seconds (33.30 k allocations: 2.195 MiB, 5.93% compilation time)
```

```
2.802597f-36
```

The values above can be summarized by looking at the speeds and number of allocations for each function. The sum function performed the best with a correct solution. This function only took 0.584358 seconds with only 2.195 Megabytes of memory allocation. The next fastest function was the reduce function which took 0.633802 seconds with only 7.286 Megabytes of allocation. This method also created the correct answer. The foldr method was more than 4 times slower than the reduce and sum functions and performed the worst. This method took 2.552371 with 2.340 Megabytes of allocations. This method also did not create the correct answer unless the init paramters was specified. The for loop function performed the second worst out of all the fuctions and only returned the correct answer when sum was intitialized at value 0.0. This function completed in 2.419582 seconds with less allocations than reduce and sum.

The for loop and foldr did not create the correct answers unless the inital values specifed was a float32 instead julia assuming the answer is a float16 (a 0 or 0f0).

2.1

In [25]:
```
fib_1(n) = first([1 1;1 0]^(n-1)*[1,0])
fib_2(n) = last( [1 1;1 0]^(n-1)*[1,1])
```

```
fib_2 (generic function with 1 method)
```

In [34]:
```
[1 1; 1 0]^(3)
```

```
2×2 Matrix{Int64}:
 3  2
 2  1
```

The array holds the state for the fibonanaci sequence. By exponentiating the array you are creating the sequence with the previous values populating different elements in the array. The final multiplication will yeild the final number in the first elemend of the matrix or the last depending of the final multipling array.

Chat GPT:

(n-1) subtracts 1 from the input number n. This is because the matrix representation assumes a 0-based index for Fibonacci numbers (F(0) = 0, F(1) = 1, F(2) = 1, etc.), while n may be given as a 1-based index.

^ is the exponentiation operator in Julia. It raises the matrix [1 1; 1 0] to the power of (n-1). This operation involves matrix exponentiation, as explained earlier.

- is the matrix multiplication operator in Julia. It multiplies the result of the exponentiation by the column vector [1, 0].

Finally, first() returns the first element of the resulting vector, which corresponds to the Fibonacci number F(n).

2.2

In [26]:
```
function fib_rec(n)
    if(n==0 || n==1)
        return 1
    else
        return fib_rec(n-1) + fib_rec(n-2)
    end
end
```

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

2.3

In [32]:
```
@time fib_1(44)
```

```
  0.000009 seconds (11 allocations: 1024 bytes)
```

```
701408733
```

In [31]:
```
@time fib_2(44)
```

```
  0.000005 seconds (11 allocations: 1024 bytes)
```

```
701408733
```

In [30]:
```
@time fib_rec(44)
```

```
  4.172246 seconds
```

```
1134903170
```

Overall the recursive function is the worst perfomring as it requires multiples calculations of the same function call as there is no dynamic programming simplification. While the fib_1 and fib_2 work in linear time as the variables are store in matrix. This causes there to not be recursive calls to the same function. Instead this function store the last values in the matrix. The fib_2 is the fastest taking 0.000005 seconds to run. fib_1 has a similar runtime as fib_2 but slightly slower at 0.000009. The fib_rec function is the slowest function by far as it takes 4.172246 seconds this is very slow compared to the other fib functions.

3.1

In [5]:
```
function sinx(x, n = 5; degree = true)
    if degree
        xRad = deg2rad(x % 360)
    else
        xRad = x
    end
    value = xRad
    numer = xRad^3.0
    denom = 6.0
    sign = -1.0
    for i in 1:n
        value += sign*numer/denom
        numer *= xRad^2.0
        denom *= (2.0 * i + 2.0) * (2.0 * i +3.0)
        sign *= -1.0
    end
    return value
end
```

```
sinx (generic function with 2 methods)
```

In [153]:
```
sum = 0.0
for i in 0:1:90
    sum += abs(sinx(i,6) - sind(i))
end
valueToBeat =  0.1e-7
if sum < valueToBeat
    print("N is within exceptable range")
end
```

```
N is within exceptable range
```

The sinx function will convert x to radians if the degrees paramter is true. The value n represents the number of terms that is used in the taylor seires to generate the output value. The larger the n value the smaller the difference between actual sin value and the calculated value.

The second code block sums the absolute differences between the sinx function and the sind function each of these differences are then added to the sum. If the sum is less than the value to beat which was specfifically 0.1e7 than the program block will print out "N is within exceptable range". The smallest accepted value is n=6.

3.2

In [162]:
```
values = [sinx(x,6) for x in 0:10:90]
pdegrees = rpad("degrees",12)
pvalue = rpad("value",12)
printstyled(pdegrees," ", pvalue, " ", color=:yellow)
println()
j = 1
for i in 0:10:90
    pdegree = rpad(i,12)
    pvalue = rpad(values[j],12)
    println(pdegree,pvalue)
    j +=1
end
```

```
degrees      value
0           0.0
10          0.17364817766693033
20          0.34202014332566877
30          0.5
40          0.6427876096865427
50          0.7660444431190767
60          0.8660254037859597
70          0.9396926208012458
80          0.984807753125684
90          1.0000000006627803
```

In [17]:
```
function runSinx(inc)
    for k in inc
        sinx(k,10)
    end
end

function runSind(inc)
    for k in inc
        sind(k)
    end
end

function timeSins(inc)
    pinc = rpad("increment",12)
    psinx = rpad("sinx time",12)
    psind = rpad("sind time",12)
    printstyled(pinc," ", psinx, " ", psind, "\n",color=:yellow)

    for i in inc
        sinxTime = @elapsed runSinx(1.0:i:90.0)
        sindTime = @elapsed runSind(1.0:i:90.0)
        println(rpad("0:$i:90",12),rpad(sinxTime,12),rpad(sindTime,13))
    end
end
```

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

In [18]:
```
inc = [0.1, 0.01, 0.001, 0.000_1, 0.000_01, 0.000_001]
timeSins(inc)
```

```
increment    sinx time    sind time
0:0.1:90    6.35e-5     1.79e-5
0:0.01:90   0.0005987   0.0001792
0:0.001:90  0.0066951   0.0015582
0:0.0001:90 0.0593895   0.0164536
0:1.0e-5:90 0.5756526   0.169668
0:1.0e-6:90 5.7955757   1.8052115
```

Overall sind fucntion is always faster than the sinx function. Usually the sinx function takes about 3.5 time longer to calculate the increment than the sind function. For increment 0.1 the sind function was faster by a factor of 3.5474860335195526. For increment 0.01 sind was faster by a factor of 3.3409598214285716. For 0.0001 the sind function was faster by a factor of 3.6095140273253272. For increment 1.0e-5 the sind function was faster by a factor of 3.392817738171016. For increment 1.0e-6 the sind function was faster by a factor of 3.2104690780000014. By looking at the trend of the elapsed times we can see that on average the function sind is much faster and as the increment get larger the gap between the two functions shrinks by a very small margin the smaller the increments become.