Math 124 - Programming for Mathematical Applications ¶

UC Berkeley, Spring 2023

Homework 2

Due Wednesday, February 1

Problem 1

Consider the 4-term recurrence relation

$$a y_{n+1} = b y_n + c y_{n-1} + d y_{n-2}$$

with $a = 1$, $b = 2$, $c = -5/4$, and $d = 1/4$.

Problem 1 (a)

Write a function `four_term(y0, y1, y2, n)` to return y_n in this recurrence, taking in the initial values, y_0 , y_1 , and y_2 . You can assume that $n \geq 2$.

In [7]:
```
function four_term(y0, y1, y2, n)
    yn = y2
    for i = 3:n
        yn = 2*y2 - 5*y1/4 + y0/4
        y0 = y1
        y1 = y2
        y2 = yn
    end
    y2
end
```

Out[7]:
```
four_term (generic function with 1 method)
```

Problem 1 (b)

Print out the results from evaluating the function with $y_0 = 1$, $y_1 = 5$, $y_2 = -2$ and $n = 5, 10, 100, 500, 10000$.

In [8]:
```
println(four_term(1, 5, -2, 5))
println(four_term(1, 5, -2, 10))
println(four_term(1, 5, -2, 100))
println(four_term(1, 5, -2, 500))
println(four_term(1, 5, -2, 10000))
```

```
-20.5
-26.62109375
-26.999999999999982
-26.999999999999957
-26.999999999999982
```

Problem 1(c)

Print out the results from evaluating the function with $y_0 = -2$, $y_1 = 1$, $y_2 = 5$, and $n = 5, 10, 100, 500, 10000$.

In [9]:
```
println(four_term(-2, 1, 5, 5))
println(four_term(-2, 1, 5, 10))
println(four_term(-2, 1, 5, 100))
println(four_term(-2, 1, 5, 500))
println(four_term(-2, 1, 5, 10000))
```

```
11.9375
13.88671875
13.999999999999954
13.999999999999954
13.999999999999954
```

Problem 2

(Adapted from **Think**, P7-4)

The following sequence converges to π:

$$a_n = \sum_{k=0}^{n} \left(\frac{6}{\sqrt{3}}\right) \frac{(-1)^k}{3^k(2k+1)}$$

Moreover, the mathematician Srinivasa Ramanujan found an infinite series that can be used to generate a numerical approximation of $\frac{1}{\pi}$:

$$\frac{1}{\pi} = \sum_{k=0}^{\infty} \left(\frac{2\sqrt{2}}{9801}\right) \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Problem 2 (a)

Write a function `pi_a()` that uses this first formula to compute and return an estimate of π in a way that avoids overflow issues. It should use a while loop to compute terms of the summation until the **difference in two consecutive terms is smaller than** 10^{-15}.

In [10]:
```
function pi_a()
    k = 0
    term = 6/sqrt(3)
    an = 6/sqrt(3)
    while abs(-1/3 * (2k + 1)/(2k + 3) * term - term) > 10^(-15)
        term *= -1/3 * (2k + 1)/(2k + 3)
        k += 1
        an += term
    end
    an
end
```

Out[10]:
```
pi_a (generic function with 1 method)
```

In [11]:
```
pi_a()
```

Out[11]:
```
3.141592653589794
```

Problem 2 (b)

Write a function `pi_ramanu()` that uses Ramanujan's formula to compute and return an estimate of π in a way that avoids overflow issues. It should use a while loop to compute terms of the summation until **the last term is smaller than** 10^{-15}.

In [12]:
```
function pi_ramanu()
    term = 2sqrt(2) * 1103 / 9801
    k = 0
    an = 0
        while term ≥ 10^(-15)
            an += term
            term *= 4 * (4k + 3)/(k + 1)^3 * (4k + 2)/390^4 * (4k + 1)
            k += 1
        end
    1/an
end
```

Out[12]:
```
pi_ramanu (generic function with 1 method)
```

In [13]:
```
pi_ramanu()
```

Out[13]:
```
3.1415926487771606
```

Problem 3

(Adapted from Project Euler, Problem 3)

We define the prime factors of a number as those prime numbers that exactly divide the original number. For instance, the prime factors of 13195 are 5, 7, 13, and 29.

What is the largest prime factor of the number 600855143?

Hint: One way to answer this question would be to create a function `is_prime(test_num)` that determines if a given number is prime, and then use this function to test candidate prime factors of our large number. Over what range of numbers should we loop in order to solve this problem, while testing only a small amount of candidates?

In [14]:
```
function is_prime(test_num)
    i = 2
    while i ≤ sqrt(test_num)
        if test_num % i == 0
            break
        else
            i += 1
        end
    end
    if i == test_num || i > sqrt(test_num)
        x = true
    else
        x = false
    end
    x
end
```

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

In [2]:
```
is_prime(600855143)
```

Out[2]: false

In [3]:
```
function largest_prime_factor(of_num)
    i = 2
    k = 1
    while i ≤ of_num/k
        if is_prime(i) == true
            if of_num % i == 0
                k = i
                i += 1
            else
                i += 1
            end
        else
            i += 1
        end
    end
    if k == 1
        println(of_num, " is the largest prime factor")
    else
        println(k, " is the largest prime factor")
    end
end
```

Out[3]: largest_prime_factor (generic function with 1 method)

In [9]:
```
largest_prime_factor(600855143)
```

```
85836449 is the largest prime factor
```

Problem 4

We wish to solve the equation $x = \cos x$ for $x \in \mathbb{R}$. The fixed point iteration for this equation is defined to be:

$$x_{n+1} = \cos(x_n)$$

where x_n are successively better approximations to the true solution x_*. We start this iteration with the initial guess $x_0 = 1$.

Problem 4 (a)

Write a function `fixed_point(tol)` that computes an approximate solution using this fixed point iteration such that the error $|x_n - x_{n-1}|$ is within a specified tolerance. Test it with `tol=1e-3, 1e-6, 1e-12`. How many iterations does it take each test to converge?

In [24]:
```
function fixed_point(tol)
    x_0 = 1
    iter = 0
    while true
        iter += 1
        x_1 = cos(x_0)
        if abs(x_1 - x_0) ≤ tol
            println(iter, " iterations: ", x_1)
            break
        else
        x_0 = x_1
        end
    end
end
```

Out[24]:
```
fixed_point (generic function with 1 method)
```

In [26]:
```
fixed_point(1e-3)
fixed_point(1e-6)
fixed_point(1e-12)
```

```
17 iterations: 0.7387603198742114
34 iterations: 0.7390855263619245
69 iterations: 0.7390851332147725
```

Problem 4 (b)

Derive Newton's method for solving the equation $x = \cos x$.

The goal is approximating a root of an equation, so the equation should be written as

$$0 = \cos x - x = f(x)$$

Starting with an approximation x_0, we start at the y-value of the equation and using the tangent line through the point, find where the line intersects the x-axis, lets call x_1. So $x_0 + a = x_1$. Therefore,

$$\begin{aligned} f(x_0) + a \cdot f'(x_0) &= 0 \\ a \cdot f'(x_0) &= -f(x_0) \\ a &= -\frac{f(x_0)}{f'(x_0)} \end{aligned}$$

Therefore, our closer approximation is

$$\begin{aligned} x_0 + a &= x_1 \\ x_0 - \frac{f(x_0)}{f'(x_0)} &= x_1 \end{aligned}$$

Or more generally,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Problem 4 (c)

Write a function `newton_method(tol)` to compute an approximate solution using Newton's method, such that the error is within a specified tolerance. Again use the initial guess $x_0 = 1$. Test it with `tol=1e-3, 1e-6, 1e-12`. How many iterations does it take each test to converge?

In [34]:
```
function newton_method(tol)
    x_0 = 1
    x = 1
    y = 1
    iter = 0
    while abs(cos(x) - x) > tol
        iter += 1
        y = x
        x = x_0 - (cos(x_0) - x_0)/(-sin(x_0) - 1)
        x_0 = y
    end
    println(iter, " iterations: ", x)
end
```

Out[34]:
```
newton_method (generic function with 1 method)
```

In [35]:
```
newton_method(1e-3)
newton_method(1e-6)
newton_method(1e-12)
```

```
3 iterations: 0.7391128909113617
5 iterations: 0.739085133385284
7 iterations: 0.7390851332151607
```