Some Revision Exercises

June 6, 2023

1. Let n and p be two positive integers. The multiplicity of p as a factor of n is the maximum integer exponent m such that

   factor of n   m = max.

   $$n = p^m \times q$$ → factor of n

   for some integer q. Particularly, q is a factor of n that can't be divided evenly by p. If p is not a factor of n, then the multiplicity of p as a factor of n is defined to be 0. As examples, shown in Table 1 are the multiplicities of some numbers as factors of 137200. Write a Python function with the `def` statement

m=0

p1, p2, p3, p4, p5, ...

For in p.

137200

2 · 2 · 2 · 2

Factor	Multiplicity	Factorisation of 137200 n
2	4	$2^4 \times 8575$
3	0	$3^0 \times 137200$
5	2	$5^2 \times 5488$
7	3	$7^3 \times 400$
70	2	$70^2 \times 28$
2744	1	$2744^1 \times 50$

q is a factor of n that can be divided evenly by p^m.

while n % p1 == 0

n = n//2

m+=1

multiplicity = [ ] .append.

Table 1: Some factors of 137200 and their multiplicities as factors of the 137200.

```
def count_multiplicity(n, *p)
```

that returns the multiplicity of each element of `p` as a factor of the positive integer `n` in a dictionary whose keys are the elements of `p` with associated values being the respective multiplicities. For instance, the following sample call generates the results as shown in Table 1:

```
>>> count_multiplicity(137200, 2, 3, 5, 7, 70, 2744)
{2: 4, 3: 0, 5: 2, 7: 3, 70: 2, 2744: 1}
```

137200 [2, 3, 5, 7, 70, 2744]

mult → {2:0, 3:0, 5:0, 7:0, 70:0, 2744:0}.

The prime numbers from 1 to 100 are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97. Why is 1 not a prime number? 1 is not a prime number because it has only one factor, namely 1.

1 2 3 4 5 6 7 8 9 10 11
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 . . . .

[2,

pt=1

2. (Adapted from Exam of Semester 1, Session 2022/2023.) Write a Python function with the `def` statement

```
def prime_list(indices)
```

that returns a list of primes indexed by the parameter `indices` that is a Python `list` of positive integers. The `prime_list()` function returns a list whose element indexed by i is the `indices[i]`-th prime. A sample call of `prime_list()` is as follows.

```
>>> prime_list([2, 4, 5, 3, 3])
[3, 7, 11, 5, 5]
```

Note that the 2nd, 4th, 5th, and 3rd primes are 3, 7, 11, and 5, respectively.

3. A list of tuples is said to increase from an integer a to an integer b, where $a \leq b$, provided it has the form

```
[(a,), (a, a+1), (a, a+1, a+2), ..., (a, a+1, a+2, ...
    ..., b)]
```

For instance, the following is a list of tuples that increase from 2 to 6:

```
[(2,), (2, 3), (2, 3, 4), (2, 3, 4, 5), (2, 3, 4, 5, 6)]
```

Likewise, a list of tuples is said to decrease from an integer a to an integer b, where $a > b$, provided it has the form

```
[(a, a-1, ..., b+1, b), (a, a-1, a-2, ..., b+2, b+1), ...
    ..., (b,)]
```

For instance, the following is a list of tuples that decrease from 6 to 2:

```
[(6, 5, 4, 3, 2), (5, 4, 3, 2), (4, 3, 2), (3, 2), (2,)]
```

Write a Python function with the `def` statement

```
def monotonic_tuples(a, b)
```

that returns the list of tuples that increases from `a` to `b` if `a≤b`, otherwise the list of tuples that decreases from `a` to `b`. Some sample calls are as follows.

i = 6 5 4 3 2

i = 3 4 5 6 7

(2,) (2,3) (2,3,4) (2,3,4,5) (2,3,4,5,6)

2nd

```
>>> monotonic_tuples(2, 2)
[(2,)]
>>> monotonic_tuples(2, 6)
[(2,), (2, 3), (2, 3, 4), (2, 3, 4, 5), (2, 3, 4, 5, 6)]
>>> monotonic_tuples(6, 2)
[(6, 5, 4, 3, 2), (5, 4, 3, 2), (4, 3, 2), (3, 2), (2,)]
>>> monotonic_tuples(-3, 3)
[(-3,), (-3, -2), (-3, -2, -1), (-3, -2, -1, 0), (-3, ...
    -2, -1, 0, 1), (-3, -2, -1, 0, 1, 2), (-3, -2, -1, ...
    0, 1, 2, 3)]
>>> monotonic_tuples(3, -3)
[(3, 2, 1, 0, -1, -2, -3), (2, 1, 0, -1, -2, -3), (1, ...
    0, -1, -2, -3), (0, -1, -2, -3), (-1, -2, -3), (-2, ...
    -3), (-3,)]
```

4. Consider the following list of lists:

```
[[9, -1, 3, 9, -1], [-5, 5, -5, 4, 8], [0, 3, 4, -7, ...
    6], [0, 9, -6, -3, 9], [-4, -8, 5, -7, 9], [0, 5, ...
    -7, 6, -1]]
```

We say that 8 precedes 6 in this case because 8 first appears in the second list, while 6 first appears in the third list. Likewise, 9 precedes 0 because 9 first appears in the first list while 0 first appears in the third list. If two numbers, such as 9 and −1, first appear in the same list, then we say that they precede each other.

Write a Python function with the `def` statement

```
def precede(lst, a, b)
```

where `lst` is a list of lists and `a` and `b` are integers, that

- returns `None` if either `a` or `b` is not contained in any list in `lst`,
- returns `True` if `a` and `b` are contained in some lists of `lst` and `a` precedes `b`, and
- returns `False` if `a` and `b` are contained in some lists of `lst` but `a` does not precede `b`.

Some sample calls are as follows:

```
>>> lst = [[9, -1, 3, 9, -1], [-5, 5, -5, 4, 8], [0, ...
    3, 4, -7, 6], [0, 9, -6, -3, 9], [-4, -8, 5, -7, 9], ...
    [0, 5, -7, 6, -1]]
>>> precede(lst, 9, -1)
True
>>> precede(lst, 5, -1)
False
>>> precede(lst, 4, 6)
True
>>> precede(lst, 4, 2)
>>> print(precede(lst, 4, 2))
None
```

5. Any function f that maps a range of consecutive integers $\{m, m+1, \ldots, n\}$ into itself for some integers m and n, where $m \leq n$, can be encoded as a list. For instance, the function f such that

-1 → 0
0 → 3
1 → 4
2 → 3
3 → 1
4 → -1

$$f(-1) = 0,\ f(0) = 3,\ f(1) = 4,\ f(2) = 3,\ f(3) = 1,\ \text{and } f(4) = -1$$

can be encoded as the list

f(-1) f(0) f(1) f(2) f(3) f(4)

```
f = [0, 3, 4, 3, 1, -1]
```

with the understanding that `f[0]` takes the value of $f(-1)$, `f[1]` takes the value of $f(0)$, and so on. Write a Python function with the `def` statement

```
def composite(f, g, m, n)
```

that returns the list encoding the composite $g \circ f$ of two functions f and g, represented by `f` and `g` as lists, sharing the domain from the integer `m` to the integer `n`, where `m`≤`n`, such that $(g \circ f)(x) = g(f(x))$ for all x in the common domain. For instance,

g[f(x)]

-1 0 1 2 3 4

```
>>> f = [0, 3, 4, 3, 1, -1]
>>> g = [-1, 2, 4, 1, 3, 0]
>>> composite(f, g, -1, 4)
[2, 3, 0, 3, 4, -1]
```

f -1:0 0:3 1:4 2:3 3:1 4:-1

g -1:-1 0:2 1:4 2:1 3:3 4:0

g[f(x)]

Here f and g are functions such that

$$f(-1) = 0,\ f(0) = 3,\ f(1) = 4,\ f(2) = 3,\ f(3) = 1,\ \text{and } f(4) = -1$$

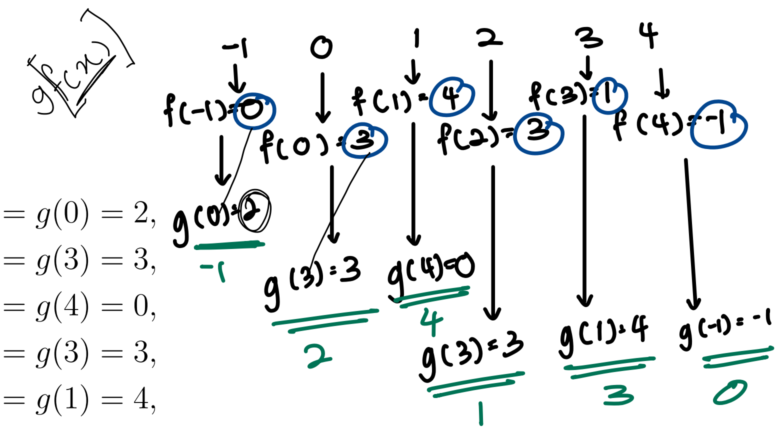
and

$$g(-1) = -1,\ g(0) = 2,\ g(1) = 4,\ g(2) = 1,\ g(3) = 3,\ \text{and } g(4) = 0.$$

-1 → f[0] = 0
0 → f[1] = 3
1 → f[2] = 4
2 → f[3] = 3
3 → f[4] = 1
4 → f[5] = -1

0, 3, 4, 3, 1, -1

-1, 2, 4, 1, 3, 0



Hence

$$
\begin{aligned}
g(f(-1)) &= g(0) = 2,\\
g(f(0)) &= g(3) = 3,\\
g(f(1)) &= g(4) = 0,\\
g(f(2)) &= g(3) = 3,\\
g(f(3)) &= g(1) = 4,\\
g(f(4)) &= g(-1) = -1.
\end{aligned}
$$

FED.

numpy

6. Recall that a numerical series $\sum_{k=1}^{\infty}(-1)^k a_k$ is said to be alternating provided $a_k \geq 0$ for all $k \in \mathbb{N}$ or $a_k \leq 0$ for all $k \in \mathbb{N}$. If

   (i) $|a_k| \geq |a_{k+1}|$ for all $k \in \mathbb{N}$ and

   (ii) $\lim_{k\to\infty} a_k = 0$,

   then the alternating series test guarantees that $\sum_{k=1}^{\infty}(-1)^k a_k$ converges to a finite limit L. Furthermore,

$$< \left|\sum_{k=1}^{n}(-1)^k a_k - L\right| \leq |a_{n+1}|$$

   for any $n \in \mathbb{N}$.

   Write a Python function with the def statement

```
def alternating_harmonic_series(bound=0.1)
```

   that returns an approximate value $\widetilde{L}$ to the limit of the alternating harmonic series

1/bound.

$\overset{0.1}{a} + \overset{0.1}{b} + \overset{0.1}{c} + \overset{0.1}{d} + \overset{0.1}{e} + \overset{0.1}{f} + g < 0.1$

$$\sum_{k=1}^{\infty}\frac{(-1)^{k+1}}{k} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots$$

   such that the $\left|\sum_{k=1}^{\infty}\frac{(-1)^{k+1}}{k} - \widetilde{L}\right|$ is strictly bounded by the parameter bound. Some sample runs are as follows.

```
>>> alternating_harmonic_series()
0.7456349206349207
>>> alternating_harmonic_series(0.001)
0.6936474305598223
>>> alternating_harmonic_series(0.00001)
0.6931521805849815
```

Do it using different approaches.

(a) Using a `while` loop.

(b) Using a `for` loop. You may need to import either **math** or **numpy**.

(c) Without using any loop. You may need to import either **math** or **numpy**.

matplotlib & numpy

7. (Adapted from Exam of Semester 1, Session 2022/2023.) For a sufficiently smooth function f on a bounded interval $[0, b]$, the value of the integral

$$I_b[f] = \int_0^b f(x)\,dx$$

24 23 10 8 2

can be evaluated numerically using the composite trapezoidal rule as follows. Let

$$[x_0, x_1],\ [x_1, x_2], \ldots, [x_{n-1}, x_n]$$

be a partition of the interval $[0, b]$ into subintervals of equal length $h_n = b/n$ such that

$h_3 = 10/3$

$$x_k = kh_n$$

for each $k \in \{0, 1, \ldots, n\}$; particularly, $x_0 = 0$ and $x_n = b$. Let

Hasil Trapezoidal Rule.

$$T_b[f; n] = h_n \left[\frac{1}{2} f(x_0) + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2} f(x_n) \right].$$

Then

$$|I_b[f] - T_b(f; n)| \leq \frac{b^3}{12n^2} \max_{x \in [0,b]} |f''(x)| \leq \text{bound} \qquad (*)$$

where the term on the right-hand side is an error bound.

$\frac{\text{limit}^3}{12n^2} \leq \text{bound}$.

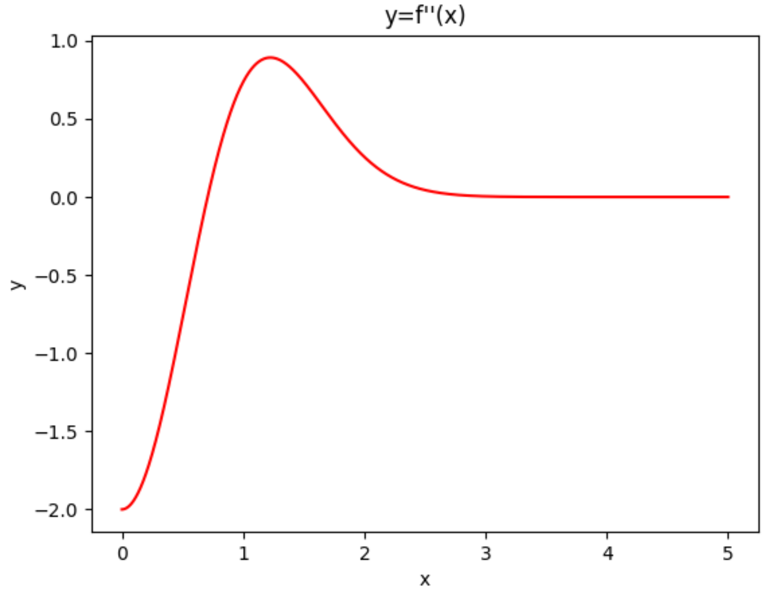
$\frac{\text{limit}^3}{\text{bound}} \leq 12n^2$.

$\sqrt{\frac{\text{limit}^3}{12\,\text{bound}}} \leq n$.

In what follows, let

$$f(x) = e^{-x^2}.$$

(a) With the aid of **matplotlib** and **numpy** modules, write a Python script to plot the graph of $f''(x)$ in red on $[0, 5]$ with 5000 evenly spaced points. The plot should look as follows.



Note that $\max_{[0,b]} |f''(x)|$ is attained at $x = 0$. You may use your knowledge of calculus to verify this.

(b) With the aid of only the **numpy** module, write a Python function with the `def` statement

```
def T(limit, bound)
```

that returns the value of $T_b[f; n]$, where `limit` denotes the value of the limit of integration b, while the error bound stipulated in (*) should be no more than the parameter `bound`.

Let us look at some sample function calls.

```
>>> T(1, 0.1)
0.731370251828563
>>> T(1, 0.01)
0.744368339763667
>>> T(1, 0.001)
0.7464612610366896
>>> T(1, 0.0001)
0.7467876578237479
```

The exact value of $I_1[f]$ is approximately 0.7468241328124279.

Likewise,

```
>>> T(2, 0.1)
0.8806186341245393
>>> T(2, 0.01)
0.8819125868282965
>>> T(2, 0.001)
0.882063560989435
>>> T(2, 0.0001)
0.8820795759852266
```

The exact value of $I_2[f]$ is approximately 0.882081390762423.

As additional requirements, do not use

- any loop and
- the `trapz()` function in **numpy**

in the definition of `T()`.

8. Recall that the probability density function (PDF) of a random variable X with normal distribution $N(\mu, \sigma^2)$ of mean μ and variance σ^2 is

$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2},$$

while the cumulative distribution function (CDF) is

$$\begin{aligned} F_X(x) &= \int_{-\infty}^{x} f_X(t)\,dt \\ &= \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{x} e^{-\frac{1}{2}\left(\frac{t-\mu}{\sigma}\right)^2} dt. \end{aligned}$$

(a) With the aid of **numpy** and the `quad()` function from **scipy.integrate**, write a Python function with the `def` statement

```
def F(x, mean=0, var=1)
```

that returns the values of the CDF F_X evaluated at the **numpy** array `x` elementwise. The parameter `mean` is the mean μ and the parameter `var` is the variance σ^2 of the distribution. Also, you may assume that the PDF f_X has already been defined as follows.

```
# The PDF of a normal distribution with mean 'mean' ...
   and variance 'var'
```

```
def f(x, mean=0, var=1):
    return 1/np.sqrt(2*np.pi*var) * ...
      np.exp(-0.5*(np.array(x)-mean)**2/var)
```

Some sample calls of `F()` is given in the following.

```
>>> F(-1, 0, 1)
array(0.15865525)
>>> F(0, 0, 1)
array(0.5)
>>> F(2, 0, 1)
array(0.97724987)
>>> x = np.arange(-3, 3, 0.5).reshape(3, 4)
>>> x
array([[-3. , -2.5, -2. , -1.5],
       [-1. , -0.5,  0. ,  0.5],
       [ 1. ,  1.5,  2. ,  2.5]])
>>> F(x, 0, 1)
array([[0.0013499 , 0.00620967, 0.02275013, ...
   0.0668072 ],
       [0.15865525, 0.30853754, 0.5       , ...
          0.69146246],
       [0.84134475, 0.9331928 , 0.97724987, ...
          0.99379033]])
>>> F(x, 2, 3)
array([[0.00194621, 0.00468738, 0.01046067, ...
   0.02165407],
       [0.04163226, 0.07445734, 0.12410654, ...
          0.19323812],
       [0.28185143, 0.386415  , 0.5       , ...
          0.613585  ]])
```

(b) Note that for every probability $p \in [0, 1]$, there is a unique x-value in $(-\infty, \infty)$ such that

$F_X(x) - p = 0$

$$F_X(x) = p$$

$F(x) - p$ $(x) = 0$

for any given mean μ and variance σ^2. So it makes sense to talk about the inverse of F given by

$$F_X^{-1}(p) = x$$

$F(x) - 0.5 = 0$ $F(x) = 0.5$ $\downarrow$ p

for all $p \in [0, 1]$. With the aid of **numpy** and the `bisect()` function from **scipy.optimize**, write a Python function with the `def` statement

```
def inv_F(prob, mean=0, var=1)
```

that returns the values of the inverse of F_X evaluated at the **numpy** array `prob` elementwise. As before, the mean μ and the variance σ^2 of the distribution are given by the parameters `mean` and `var`, respectively. Also, if an element of `prop` is out of range, i.e. not within the interval $[0, 1]$, then the corresponding value of the array returned should be `numpy.nan` that denotes that the quantity "**Not a Number**".

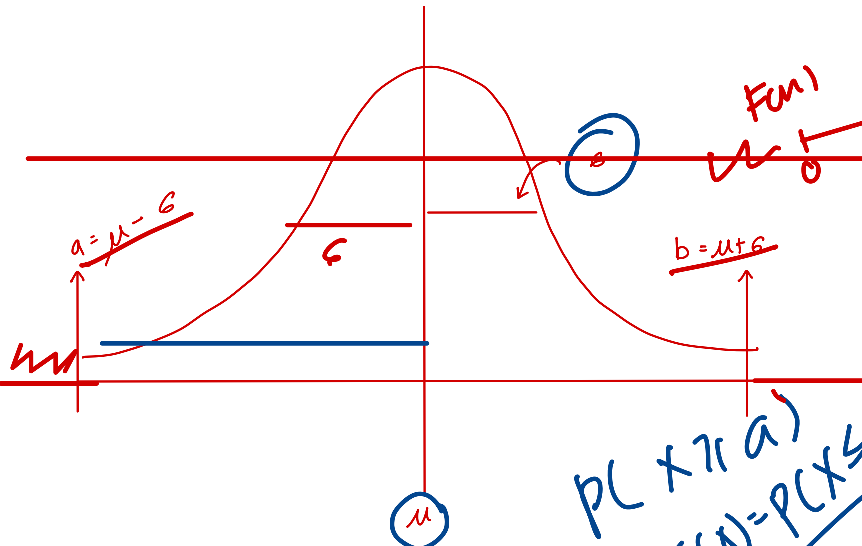
Some sample calls of `inv_F()` are given in the following.

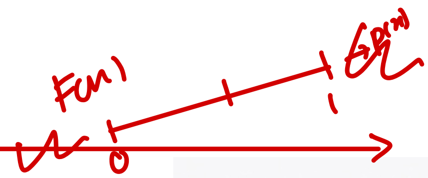
$F(x) =$

$x = F^{-1}$

```
>>> inv_F(0.35)
array(-0.38532047)
>>> F(inv_F(0.35))
array(0.35)
>>> inv_F(0.35, 2, 1)
array(1.61467953)
>>> F(inv_F(0.35, 2, 1), 2, 1)
array(0.35)
>>> prob = np.arange(0, 1, 0.1).reshape(2,5)
>>> prob
array([[0. , 0.1, 0.2, 0.3, 0.4],
       [0.5, 0.6, 0.7, 0.8, 0.9]])
>>> inv_F(prob)
array([[-3.90000000e+01, -1.28155157e+00, ...
   -8.41621205e-01,
        -5.24400513e-01, -2.53347103e-01],
       [ 1.81898940e-12,  2.53347103e-01,  ...
          5.24400513e-01,
         8.41621234e-01,  1.28155157e+00]])
>>> F(inv_F(prob))
array([[0. , 0.1, 0.2, 0.3, 0.4],
       [0.5, 0.6, 0.7, 0.8, 0.9]])
>>> inv_F(prob, 2, 1)
array([[-37.        ,  0.71844843,  1.15837879,  ...
   1.47559949,
          1.7466529 ],
       [  2.        ,  2.2533471 ,  2.52440051,  ...
         2.84162123,
          3.28155157]])
>>> F(inv_F(prob, 2, 1), 2, 1)
array([[0. , 0.1, 0.2, 0.3, 0.4],
       [0.5, 0.6, 0.7, 0.8, 0.9]])
```

```
>>> prob = np.array([0.4, -0.5, 0.7, 1.4])
>>> prob
array([ 0.4, -0.5,  0.7,  1.4])
>>> inv_F(prob)
array([-0.2533471 ,         nan,  0.52440051, ...
        nan])
```



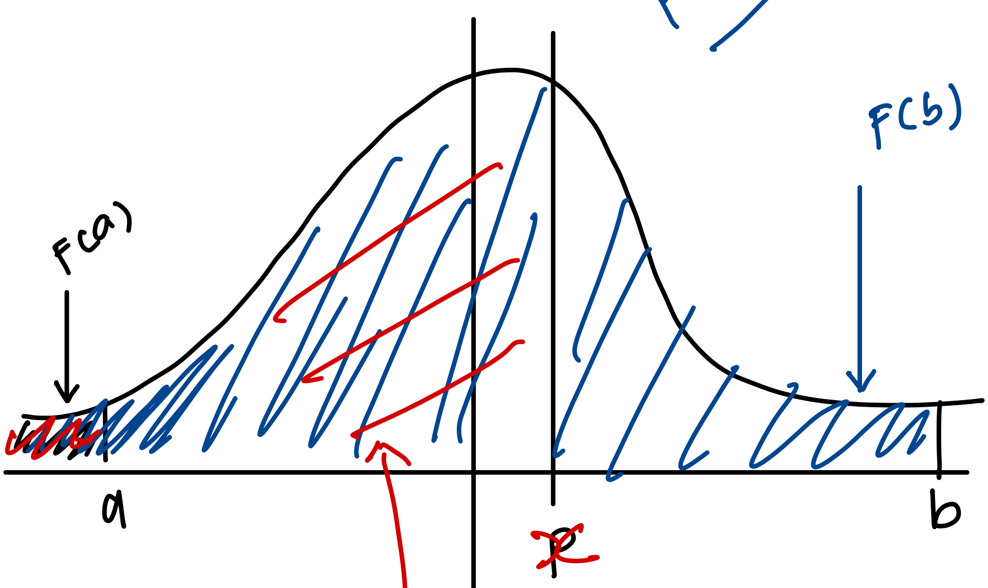


By setting the equation `F(x, mean, var) - p = 0`, we are essentially looking for the value of `x` that corresponds to a specific probability `p` within the normal distribution. This is useful for determining percentiles or quantiles in the distribution. For example, finding `x` such that `F(x, mean, var) - 0.5 = 0` would give you the median of the normal distribution.

The bisection method, as mentioned earlier, can be used to numerically solve this equation and find the corresponding `x` value that satisfies the condition `F(x, mean, var) - p = 0`.

$P(X \geq a)$

$F(X) = P(X \leq a)$



lambda x: 4-x

$F(x) = p$

$g(x) = \underbrace{F(x) - p} = 0$

bisect(g, a, b)

$g(x) = 0$

```
>>> prob = np.array([0.4, -0.5, 0.7, 1.4])
>>> prob
array([ 0.4, -0.5,  0.7,  1.4])
>>> inv_F(prob)
array([-0.2533471 ,         nan,  0.52440051, ...
           nan])
```

9. Consider the function

$$\text{pulse}(x; a, b, f) = \begin{cases} \cos(x)\sin(f \cdot x) & \text{if } a < x < b; \\ \cos(x) & \text{otherwise} \end{cases}$$

where a, b, and f are parameters with $a < b$ and $f > 0$.

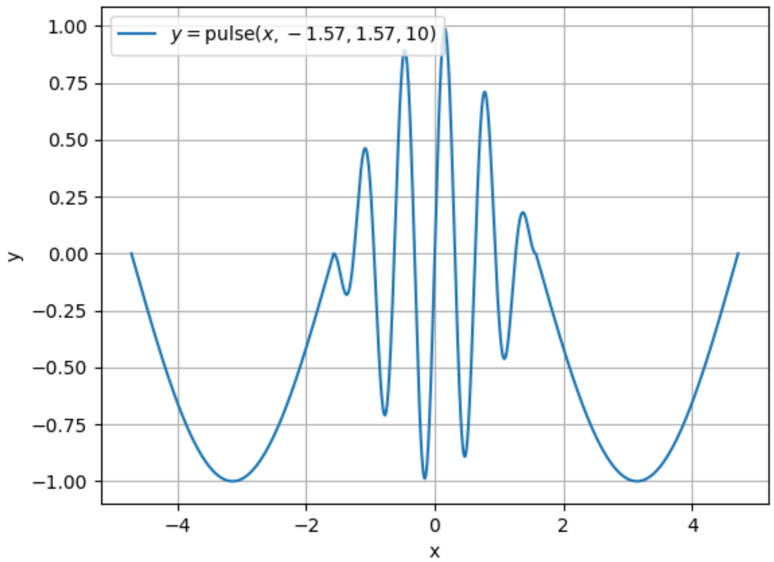
(a) Using the aid of the **NumPu** module, write a Python function with the `def` statement

```
def pulse(x, a, b, f)
```

that returns the value of the function pulse at each element of the **NumPy** array given the parameters `a`, `b`, and `f` that represents the values of a, b, and f, respectively. A sample call of the function is given in the following.

```
>>> x = np.linspace(-np.pi, np.pi, 10)
>>> x.reshape(2, 5)
array([[-3.14159265, -2.44346095, -1.74532925, ...
   -1.04719755, -0.34906585],
[ 0.34906585,  1.04719755,  1.74532925,  ...
   2.44346095,  3.14159265]])
>>> pulse(x, -np.pi/2, np.pi/2, 5)
array([-1.        , -0.76604444, -0.17364818,  ...
   0.4330127 , -0.92541658,
0.92541658, -0.4330127 , -0.17364818, -0.76604444, ...
   -1.        ])
```

(b) With the aid of the **Matplotlib** module and the `pulse()` function you defined in (a), write a Python script to plot the graph of $\text{pulse}(x; -\pi/2, \pi/2, 10)$ as x varies in $[-3\pi/2, 3\pi, 2]$ using 1000 evenly spaced points. The graph should look as follows.



(c) * With the aid of the **Matplotlib** module and the `pulse()` function you defined in (a), write a Python script that shows an animation of the graph of $\mathrm{pulse}(x; -\pi/2, \pi/2, f)$ for $x \in [-\pi, \pi]$ using 1000 evenly space points as f varies from $f = 1$ to $f = 100$ and then back to $f = 1$ with a stepsize of 1. The animation should look like the one shown at `https://youtu.be/eoL49F_GcFA`.

*This is not for revision purpose.