Problem 9: Maximum likelihood and floating-point

This problem concerns floating-point arithmetic, motivated by the statistical concept of maximum likelihood estimation. It has four exercises, numbered 0-3, and is worth a total of ten (10) points.

Setup. This problem involves a number of functions from the Python standard library. Here are some of them; run the code cell below to make them available for use.

In [ ]:
```
# The test cells need these:
from random import choice, randint, uniform, shuffle
from math import isclose

# You'll need these in Exercises 1 & 3:
from math import exp, sqrt, pi, log
```

Products

Suppose you are given a collection of n data values, named x_0, x_1, ..., x_{n-1}. Mathematically, we denote their sum as

$$x_0 + x_1 + \cdots + x_{n-1} \equiv \sum_{k=0}^{n-1} x_i.$$

In Python, it's easy to implement this formula using the `sum()` function, which can sum the elements of any iterable collection, like a list:

In [ ]:
```
x = [1, 2, 3, 4, 5]
print("sum({}) == {}".format(x, sum(x)))

sum(x)
```

```
sum([1, 2, 3, 4, 5]) == 15
```

Out[ ]: 15

Suppose instead that we wish to compute the *product* of these values:

$$x_0 \cdot x_1 \cdot \cdots \cdot x_{n-1} \equiv \prod_{k=0}^{n-1} x_i.$$

Exercise 0 (3 points). Write a function, `product(x)`, that returns the product of a collection of numbers `x`.

In [ ]:
```
def product(x):
    p=1
    for e in x:
        p = e*p
    return p

# Demo:
print("product({}) == {}?".format(x, product(x))) # Should be 120
```

```
product([1, 2, 3, 4, 5]) == 120?
```

In [ ]:
```
# Test cell: `product_test0` (1 point)

def check_product(x_or_n):
    import numpy as np
    eps = np.finfo(float).eps
    def delim_vals(x, s=', ', fmt=str):
        return s.join([fmt(xi) for xi in x])
    def gen_val(do_int):
        if do_int:
            v = randint(-100, 100)
            while v == 0:
                v = randint(-100, 100)
            assert v != 0
        else:
            v = uniform(-10, 10)
        return v

    if type(x_or_n) is int:
        n = x_or_n
        do_int = choice([False, True])
        x = [gen_val(do_int) for _ in range(n)]
    else:
        x = x_or_n
        n = len(x)

    if n > 10:
        msg_values = "{}, ..., {}".format(n, delim_vals(x[:5]), delim_vals(x[-5:]))
    else:
        msg_values = delim_vals(x)
    msg = "{} values: [{}]".format(n, msg_values)
    print(msg)
    p = product(x)
    print("  => Your result: {}".format(p))

    # Check
    for xi in x:
        p /= xi
    abs_err = p - 1.0
    print("  => After dividing by input values: {}".format(p))
    assert abs(p-1.0) <= (20.0 / n) * eps, \
           "Dividing your result by the individual values is {}, which is a bit too

check_product([1, 2, 3, 4, 5]) == 120
print("\n(Passed first test!)")
```

```
5 values: [1, 2, 3, 4, 5]
  => Your result: 120
  => After dividing by input values: 1.0

(Passed first test!)
```

In [ ]:
```python
# Test cell: `product_test1` (2 points)
for k in range(5):
    print("=== Test {} ===".format(k))
    check_product(10)
    print()
print("(Passed second battery of tests!)")
```

```
=== Test 0 ===
10 values: [1.0442416771892304, 4.839451708401663, -8.487521513363369, 9.49573975285
1698, -0.7560372986797521, 8.526111154548069, -3.7573897340219586, 6.35583339800697
9, -7.460760700520204, -5.176830503096317]
  => Your result: -2421625.0033283895
  => After dividing by input values: 0.9999999999999998

=== Test 1 ===
10 values: [5.203155961718464, 1.8707327433597438, -4.794145789572353, -5.8924316422
82839, -1.9460999329644135, 3.05699948480685, 4.969980364936237, -8.659541244501243,
-6.640015096714659, -0.08124736741673644]
  => Your result: 37981.56077295323
  => After dividing by input values: 0.9999999999999998

=== Test 2 ===
10 values: [5.471970149858265, -3.834404223018777, 5.625153169969707, -4.46972741489
12225, 2.6805762364724472, -0.4595007926993606, -2.078807137798135, -3.6231628533219
844, 8.44576859209753, 6.6650632044922276]
  => Your result: -275497.04040165443
  => After dividing by input values: 1.0

=== Test 3 ===
10 values: [99, 9, 86, 29, 35, -35, 94, -68, -90, -44]
  => Your result: 68903644593168000
  => After dividing by input values: 1.0

=== Test 4 ===
10 values: [8.189197650865868, -6.660946114017998, 8.660522397279124, -7.48053315603
3893, 0.6648794777752052, -3.217486491873734, 8.24893671182815, 9.867908079727968, -
7.774648861714564, 4.481241899445056]
  => Your result: 21439557.308486696
  => After dividing by input values: 1.0

(Passed second battery of tests!)
```

Gaussian distributions

Recall that the probability density of a *normal* or *Gaussian* distribution with mean μ and variance σ^2 is,

$$g(x) \equiv \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right].$$

While σ^2 denotes the variance, the *standard deviation* is σ. You may assume $\sigma > 0$.

Exercise 1 (1 point). Write a function `gaussian0(x, mu, sigma)` that returns $g(x)$ given one floating-point value `x`, a mean value `mu`, and standard deviation `sigma`.

For example,

`gaussian0(1.0, 0.0, 1.0)`
should return the value $\frac{1}{\sqrt{2\pi}}\exp(-0.5) \approx 0.2419707\ldots$.

> In the signature below, `mu` and `sigma` are set to accept default values of 0.0 and 1.0, respectively. But your function should work for any value of `mu` and any `sigma > 0`.

In [ ]:
```python
import math
def gaussian0(x, mu=0.0, sigma=1.0):
    denom = sigma*((2*pi)**(1/2))
    ex = math.e
    g = -(1/2)*((x-mu)/sigma)**2
    return (1/denom)*(ex**g)

print(gaussian0(1.0)) # Should get 0.24197072451914...
```

```
0.24197072451914337
```

In [ ]:
```python
x=2.33760432604155
mu=4.527248601712158
sigma=7.172075044316745
gaussian0(x, mu, sigma)
```

Out[ ]:
```
0.05309152260691058
```

In [ ]:
```python
# Test cell: `gaussian0_test` (1 point)

def check_gaussian0(x=None, mu=None, sigma=None, k=None):
    if x is None:
        x = uniform(-10, 10)
    if mu is None:
        mu = uniform(-10, 10)
    if sigma is None:
        sigma = uniform(1e-15, 10)
    if k is None:
        k_str = ""
    else:
        k_str = " #{}".format(k)
    assert type(x) is float and type(mu) is float and type(sigma) is float
    print("Test case{}: x={}, mu={}, sigma={}".format(k_str, x, mu, sigma))
    your_result = gaussian0(x, mu, sigma)
    log_your_result = log(your_result)
```

```
    log_true_result = -0.5*((x - mu)/sigma)**2 - log(sigma*sqrt(2*pi))
    # Use an f-string to include variable values in the assertion message
    assert isclose(log_your_result, log_true_result, rel_tol=1e-9), f"Test case{k_s
    print("==> Passed.")

check_gaussian0(x=1.0, mu=0.0, sigma=1.0, k=0)

for k in range(1, 6):
    check_gaussian0(k=k)

print("\n(Passed!)")
```

```
Test case #0: x=1.0, mu=0.0, sigma=1.0
==> Passed.
Test case #1: x=-8.467156385293675, mu=-1.1770277997990526, sigma=5.973960156931857
==> Passed.
Test case #2: x=-4.241101477002955, mu=0.034662516465370885, sigma=6.3679436133394
==> Passed.
Test case #3: x=-8.601852714587842, mu=2.645402325185362, sigma=7.737157075593329
==> Passed.
Test case #4: x=8.853998110737056, mu=4.797121292755303, sigma=1.9607421224863693
==> Passed.
Test case #5: x=-4.874793399482826, mu=-3.3662170837924865, sigma=5.809597512390849
==> Passed.

(Passed!)
```

Exercise 2 (1 point). Suppose you are now given a *list* of values, x_0, x_1, $\ldots$, x_{n-1}. Write a function, `gaussians()`, that returns the collection of $g(x_i)$ values, also as a list, given specific values of μ and σ.

For example:

```
gaussian0(-2, 7.0, 1.23) == 7.674273364934753e-13
gaussian0(1, 7.0, 1.23) == 2.2075380785334786e-06
gaussian0(3.5, 7.0, 1.23) == 0.0056592223086500545
```

Therefore,

```
gaussians([-2, 1, 3.5], 7.0, 1.23) == [7.674273364934753e-13,
2.2075380785334786e-06, 0.0056592223086500545]
```

In [ ]:

```
def gaussian0(x, mu=0.0, sigma=1.0):
    denom = sigma*((2*pi)**(1/2))
    ex = math.e
    g = -(1/2)*((x-mu)/sigma)**2
    return (1/denom)*(ex**g)

def gaussians(X, mu=0.0, sigma=1.0):
    assert type(X) is list
    g_list = [gaussian0(x,mu,sigma) for x in X]
    return g_list



print(gaussians([-2, 1, 3.5], 7.0, 1.23))
```

```
[7.674273364934764e-13, 2.2075380785334803e-06, 0.005659222308650056]
```

In [ ]:
```python
# Test cell: `gaussians_test` (1 point)

mu = uniform(-10, 10)
sigma = uniform(1e-15, 10)
X = [uniform(-10, 10) for _ in range(10)]
g_X = gaussians(X, mu, sigma)
for xi, gi in zip(X, g_X):
    assert isclose(gi, gaussian0(xi, mu, sigma))

print("\n(Passed!)")
```

```
(Passed!)
```

Likelihoods and log-likelihoods

In statistics, one technique to fit a function to data is a procedure known as *maximum likelihood estimation (MLE)*. At the heart of this method, one needs to calculate a special function known as the *likelihood function*, or just the *likelihood*. Here is how it is defined.

Let x_0, x_1, $\ldots$, x_{n-1} denote a set of n input data points. The likelihood of these data, $L(x_0, \ldots, x_{n-1})$, is defined to be

$$L(x_0, \ldots, x_{n-1}) \equiv \prod_{k=0}^{n-1} p(x_i),$$

where $p(x_i)$ is some probability density function that you believe is a good model of the data. The MLE procedure tries to choose model parameters that maximize $L(\ldots)$.

In this problem, let's suppose for simplicity that $p(x)$ is a normal or Gaussian distribution with mean μ and variance σ^2, meaning that $p(x_i) = g(x_i)$. Here is a straightforward way to implement $L(\ldots)$ in Python.

In [ ]:
```python
def likelihood_gaussian(x, mu=0.0, sigma=1.0):
    assert type(x) is list

    g_all = gaussians(x, mu, sigma)
    L = product(g_all)
    return L

print(likelihood_gaussian(X))
```

```
5.121487128441623e-65
```

The problem is that you might need to multiply many small values. Then, due to the limits of finite-precision arithmetic, the likelihood can quickly go to zero, becoming meaningless, even for a small number of data points.

In [ ]:
```python
# Generate many random values
N = [int(2**k) for k in range(8)]
```

```
X = [uniform(-10, 10) for _ in range(max(N))]

# Evaluate the likelihood for different numbers of these values:
for n in N:
    print("n={}: likelihood ~= {}.".format(n, likelihood_gaussian(X[:n])))
```

```
n=1: likelihood ~= 0.3167448662637015.
n=2: likelihood ~= 0.00013837849557031605.
n=4: likelihood ~= 3.9831573042102027e-17.
n=8: likelihood ~= 5.294839241922683e-21.
n=16: likelihood ~= 1.7405059349750127e-99.
n=32: likelihood ~= 4.759969036820765e-233.
n=64: likelihood ~= 0.0.
n=128: likelihood ~= 0.0.
```

Recall that the smallest representable value in double-precision floating-point is $\approx 10^{-308}$. Therefore, if the true exponent falls below that value, we cannot store it. You should see this behavior above.

One alternative is to compute the *log-likelihood*, which is defined simply as the (natural) logarithm of the likelihood:

$$\mathcal{L}(x_0, \ldots, x_{n-1}) \equiv \log L(x_0, \ldots, x_{n-1}).$$

Log-transforming the likelihood has a nice feature: the location of the maximum value will not change. Therefore, maximizing the log-likelihood is equivalent to maximizing the (plain) likelihood.

Let's repeat the experiment above but also print the log-likelihood along with the likelihood:

In [ ]:
```
for n in N:
    L_n = likelihood_gaussian(X[:n])
    try:
        log_L_n = log(L_n)
    except ValueError:
        from math import inf
        log_L_n = -inf
    print("n={}: likelihood ~= {} and log-likelihood ~= {}.".format(n, L_n, log_L_n
```

```
n=1: likelihood ~= 0.3167448662637015 and log-likelihood ~= -1.149658667445929.
n=2: likelihood ~= 0.00013837849557031605 and log-likelihood ~= -8.885517905680441.
n=4: likelihood ~= 3.9831573042102027e-17 and log-likelihood ~= -37.7618717835775.
n=8: likelihood ~= 5.294839241922683e-21 and log-likelihood ~= -46.68755433463702.
n=16: likelihood ~= 1.7405059349750127e-99 and log-likelihood ~= -227.4017483682216
6.
n=32: likelihood ~= 4.759969036820765e-233 and log-likelihood ~= -534.94208550426.
n=64: likelihood ~= 0.0 and log-likelihood ~= -inf.
n=128: likelihood ~= 0.0 and log-likelihood ~= -inf.
```

At first, it looks good: the log-likelihood is much smaller than the likelihood. Therefore, you can calculate it for a much larger number of data points.

But the problem persists: just taking $\log L(\ldots)$ doesn't help. When $L(\ldots)$ rounds to zero, taking the $\log$ produces minus infinity. For this last exercise, you need to fix this problem.

Exercise 3 (5 points). Using the fact that $\log$ and $\exp$ are inverses of one another, i.e., $\log(\exp x) = x$, come up with a way to compute the log-likelihood that can handle larger values of `n`.

For example, in the case of `n=128`, your function should produce a finite value rather than $-\infty$.

> *Hint.* In addition to the inverse relationship between $\log$ and $\exp$, use the algebraic fact that $\log(a \cdot b) = \log a + \log b$ to derive a different way to comptue log-likelihood.

In [ ]:
```
def log_likelihood_gaussian(X, mu=0.0, sigma=1.0):
    def log_gaussian0(x):
        return -0.5*((x-mu)/sigma)**2 - log(sigma*sqrt(2*pi))
    log_gaussians = [log_gaussian0(xi) for xi in X]
    return sum(log_gaussians)
```

In [ ]:
```
# Test cell: `log_likelihood_gaussian_test0` (2 points)

# Check that the experiment runs to completion (no exceptions)
for n in N:
    log_L_n = log_likelihood_gaussian(X[:n])
    print("n={}: log-likelihood ~= {}.".format(n, log_L_n))

print("\n(Passed!)")
```

```
n=1: log-likelihood ~= -1.149658667445929.
n=2: log-likelihood ~= -8.885517905680441.
n=4: log-likelihood ~= -37.761871783577504.
n=8: log-likelihood ~= -46.68755433463702.
n=16: log-likelihood ~= -227.4017483682217.
n=32: log-likelihood ~= -534.94208550426.
n=64: log-likelihood ~= -1082.2042340181447.
n=128: log-likelihood ~= -2302.6803534746878.

(Passed!)
```

In [ ]:
```
# Test cell: `log_likelihood_gaussian_test1` (3 points)

for k in range(100):
    mu = uniform(-10, 10)
    sigma = uniform(1e-15, 10)
    x0 = uniform(-10, 10)
    nc = randint(1, 5)
    n0s = [randint(1, 16384) for _ in range(nc)]
    x0s = [uniform(-10, 10) for _ in range(nc)]
    log_L_true = 0.0
    X = []
    for c, x0, n0 in zip(range(nc), x0s, n0s):
        X += [x0] * n0
        log_L_true += n0 * (-0.5*((x0 - mu)/sigma)**2 - log(sigma*sqrt(2*pi)))
    shuffle(X)
```

```
    log_L_you = log_likelihood_gaussian(X, mu, sigma)
    msg = "Test case {} failed: mu={}, sigma={}, nc={}, x0s={}, n0s={}, N={}, true=
    assert isclose(log_L_you, log_L_true, rel_tol=len(X)*1e-10), msg

print("\n(Passed!)")
```

```
(Passed!)
```

Fin! This cell marks the end of this problem. If everything works, congratulations! If you haven't done so already, be sure to submit it to get the credit you deserve.