Alex Beckwith

Math856 - Spring 23

HW2

In [ ]:
```
from functools import partial
from math import e, factorial, inf, sqrt, pi

from matplotlib import pyplot as plt
import numpy as np
from matplotlib.pyplot import subplots, fill_between
from numpy import linspace
from scipy.integrate import quad
from scipy.stats import expon, uniform, norm
```

Problem 1:

Let random variables X1, X2, ..., Xn (iid exp(1)); n = 20

1a

In [ ]:
```
# Since X1 follows an exponential distribution with λ = 1,
# X1 can be approximated using the density function X1 ~ fx(x)

fx1 = lambda x: e ** (-x) if x > 0 else 0
```

$$X_n = \begin{cases} e^{-x}, & x > 0 \\ 0, & x \le 0 \end{cases}$$

$$P(X_1 < 1) = \int_0^1 e^{-x}dx = -e^{-x}\Big|_0^1 = -\frac{1}{e} + 1$$

$$= 0.6321$$

In [ ]:
```
# Using scipy to create a random variable object
# The default parameters for a scipy expon rv are where λ = 1
rv = expon(scale=1)
x1_prob_cdf = rv.cdf(1)
print("1a:")
print(f"P(X1 < 1) computed using rv object cdf: {x1_prob_cdf}")
```

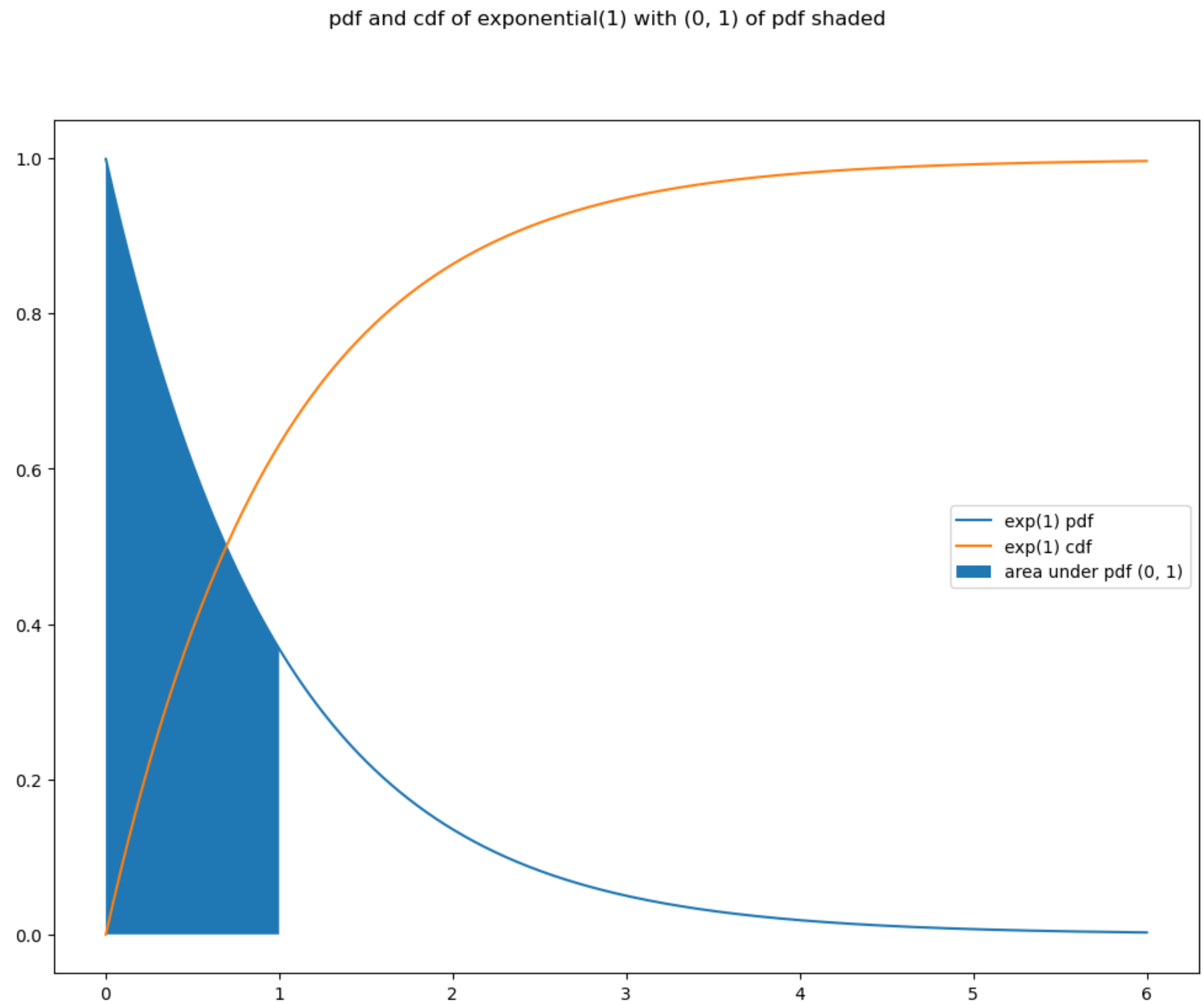
```
1a:
P(X1 < 1) computed using rv object cdf: 0.6321205588285577
```

In [ ]:
```
# Using the density function to integrate over [0, 1]
x1_prob_integration = quad(fx1, 0, 1)[0]
print(f"P(X1 < 1) computed by integrating fx over (0, 1): {x1_prob_integration}")
```

```
P(X1 < 1) computed by integrating fx over (0, 1): 0.6321205588285578
```

In [ ]:
```
# creating density functions for chart
n_points = 1000
xmin = 0 + 1 / n_points
xmax = 6 - 1 / n_points

# x, pdf over x, and cdf over x
x1 = linspace(xmin, xmax, n_points)
y_fx1 = [fx1(z) for z in x1]
y_Fx1 = [quad(fx1, xmin, z)[0] for z in x1]
```

In [ ]:
```
# creating chart object
fig, ax = subplots(figsize=(12, 9))
fig.suptitle("pdf and cdf of exponential(1) with (0, 1) of pdf shaded")

# plotting pdf and cdf
ax.plot(x1, y_fx1, label="exp(1) pdf")
ax.plot(x1, y_Fx1, label="exp(1) cdf")

# creating filled section for area under pdf
section_x = linspace(0.001,0.999,1000)
section_y = [fx1(z) for z in section_x]
fill_between(section_x, section_y, label="area under pdf (0, 1)")

legend = ax.legend()
```



1b: Let X(1), X(2), ..., X(n) be their order statistic. Find P(X(1) < 1)

An order statistic density function can be expressed using the function "order_pdf" below.

functools.partial was used to create a density function for the case when k = 1, n = 20, and dist = exp(1).

In [ ]:

```
def order_pdf(x: float, k: int, n: int, func):
    term1 = factorial(n) / (factorial(k - 1) * factorial(n - k))
    term2 = func(x)
    term3 = quad(func, 0, x)[0] ** (k - 1)
    term4 = (1 - quad(func, 0, x)[0]) ** (n - k)
    return term1 * term2 * term3 * term4

fx_k1 = partial(order_pdf, k = 1, n = 20, func = fx1)
Fx_k1 = quad(fx_k1, 0, 1)[0]

print("1b: the density function ")
print("To find P(X(1) < 1), the density function was integrated over (0, 1)")
print(f"P(X(1) < 1) = {Fx_k1}")
```

$f_{X_{(k)}}(x) = \frac{n!}{(k-1)!(n-k)!} f_X(x) F_X(x)^{k-1} (1-F_X(x))^{n-k}$

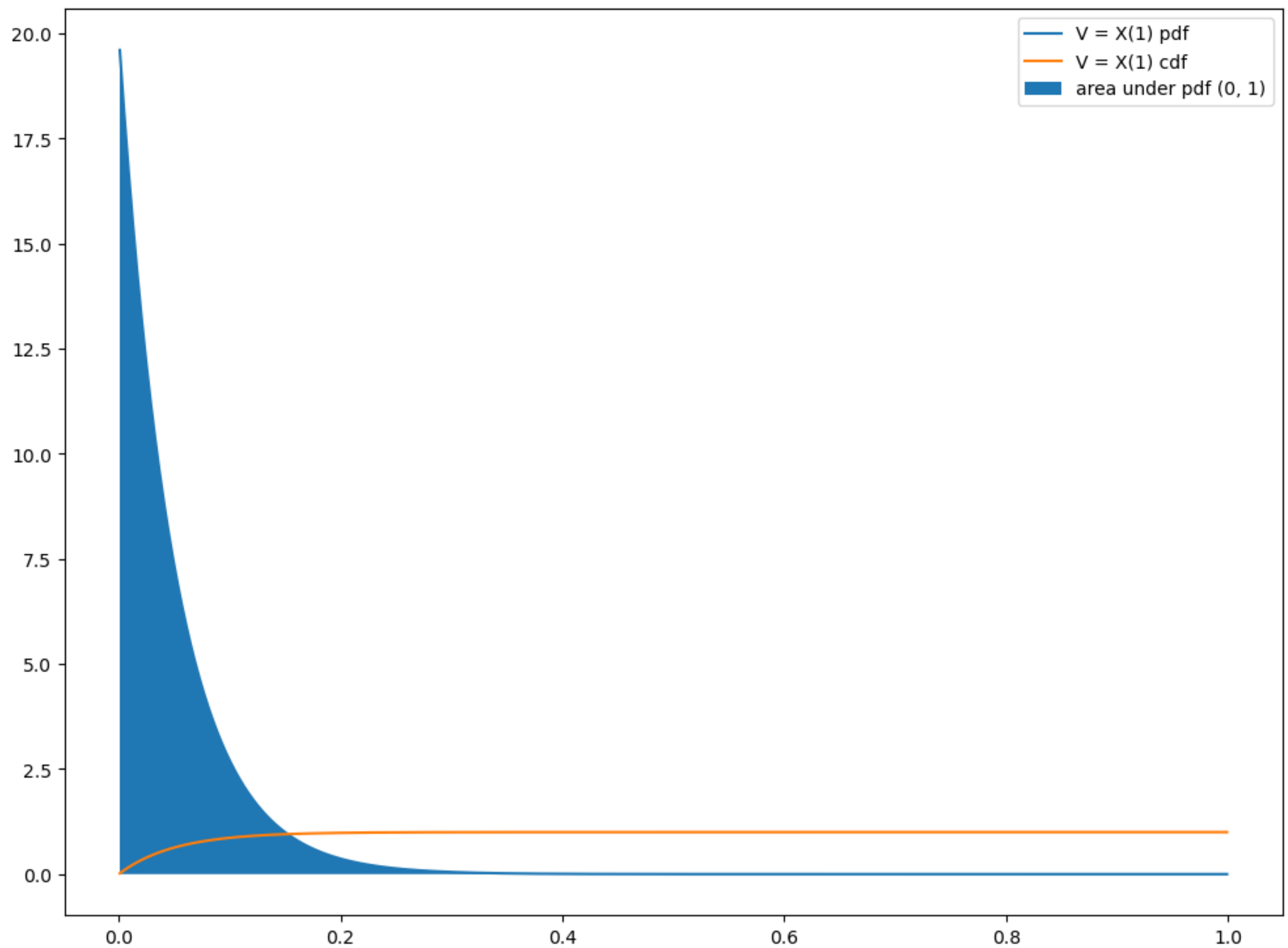
```
1b: the density function
To find P(X(1) < 1), the density function was integrated over (0, 1)
P(X(1) < 1) = 0.9999999979388465
```

In [ ]:

```
x1_k1 = linspace(xmin, 0.999, n_points)
y_fx_k1 = [fx_k1(z) for z in x1_k1]
y_Fx_k1 = [quad(fx_k1, 0, z)[0] for z in x1_k1]
# creating chart object
fig, ax = subplots(figsize=(12, 9))
fig.suptitle("pdf and cdf of X(1) with (0, 1) of pdf shaded")

# plotting pdf and cdf
ax.plot(x1_k1, y_fx_k1, label="V = X(1) pdf")
```

```
ax.plot(x1_k1, y_Fx_k1, label="V = X(1) cdf")

# creating filled section for area under pdf
fill_between(x1_k1, y_fx_k1, label="area under pdf (0, 1)")

legend = ax.legend()
```

pdf and cdf of X(1) with (0, 1) of pdf shaded



In [ ]:

```
fx_k20 = partial(order_pdf, k = 20, n = 20, func = fx1)
x1_k20_integration = quad(fx_k20, 0, 1)[0]

print("1c:")
print("A new density function was created for the case when k = 20")
print("To find P(X(20) < 1), the density function was integrated over (0, 1)")
print(f"P(X(20) < 1) = {x1_k20_integration}")
```

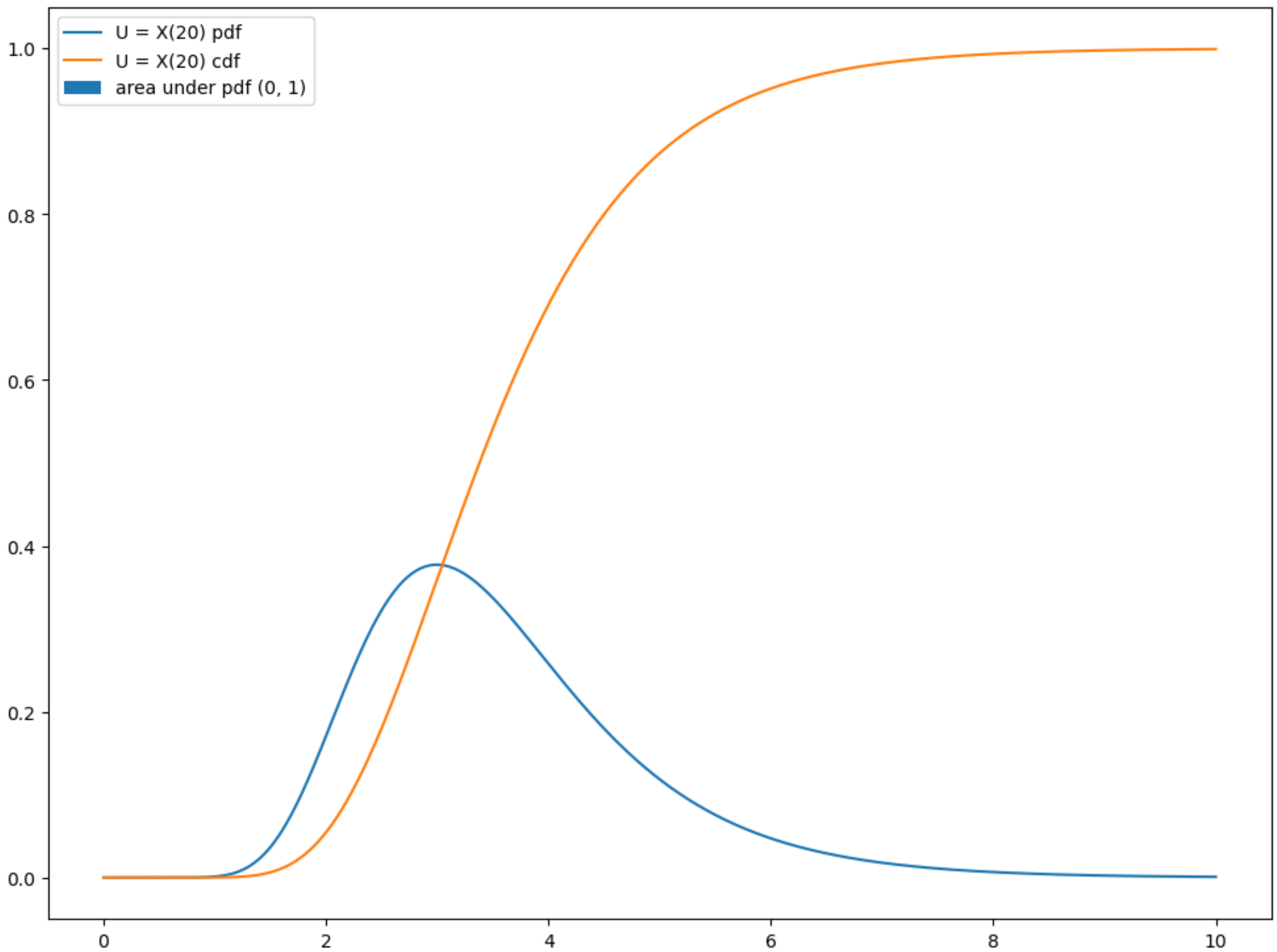
```
1c:
A new density function was created for the case when k = 20
To find P(X(20) < 1), the density function was integrated over (0, 1)
P(X(20) < 1) = 0.00010375243723147873
```

In [ ]:

```
xmin = 0.001
xmax = 10
n_points = 1000

x1_k20 = linspace(xmin, xmax, n_points)
y_fx_k20 = [fx_k20(z) for z in x1_k20]
y_Fx_k20 = [quad(fx_k20, 0, z)[0] for z in x1_k20]
# creating chart object
fig, ax = subplots(figsize=(12, 9))
fig.suptitle("pdf and cdf of X(20) with (0, 1) of pdf shaded")

# plotting pdf and cdf
ax.plot(x1_k20, y_fx_k20, label="U = X(20) pdf")
ax.plot(x1_k20, y_Fx_k20, label="U = X(20) cdf")
```

```
# creating filled section for area under pdf
xfill = linspace(0.001, 0.999, n_points)
yfill = [fx_k20(z) for z in xfill]
fill_between(xfill, yfill, label="area under pdf (0, 1)")

legend = ax.legend()
```

pdf and cdf of X(20) with (0, 1) of pdf shaded



The pdfs of V = X(1) and U = X(20) are shown graphically above.

To calculate expected value, we'll create a function that takes another function as input.

This function will create a new function, y = x * f(x), and integrate over (-inf, inf)

In [ ]:
```
def expect(func):
    exp_func = lambda x: x * func(x)
    return round(quad(exp_func, -inf, inf)[0], 12)

v = fx_k1
u = fx_k20
print("1d: the expected value of v =", expect(v))
print("1e: the expected value of u =", expect(u))
```

$EX = \int_{-\infty}^{\infty} x f(x)\, dx$

```
1d: the expected value of v = 0.05
1e: the expected value of u = 3.597739657144
```

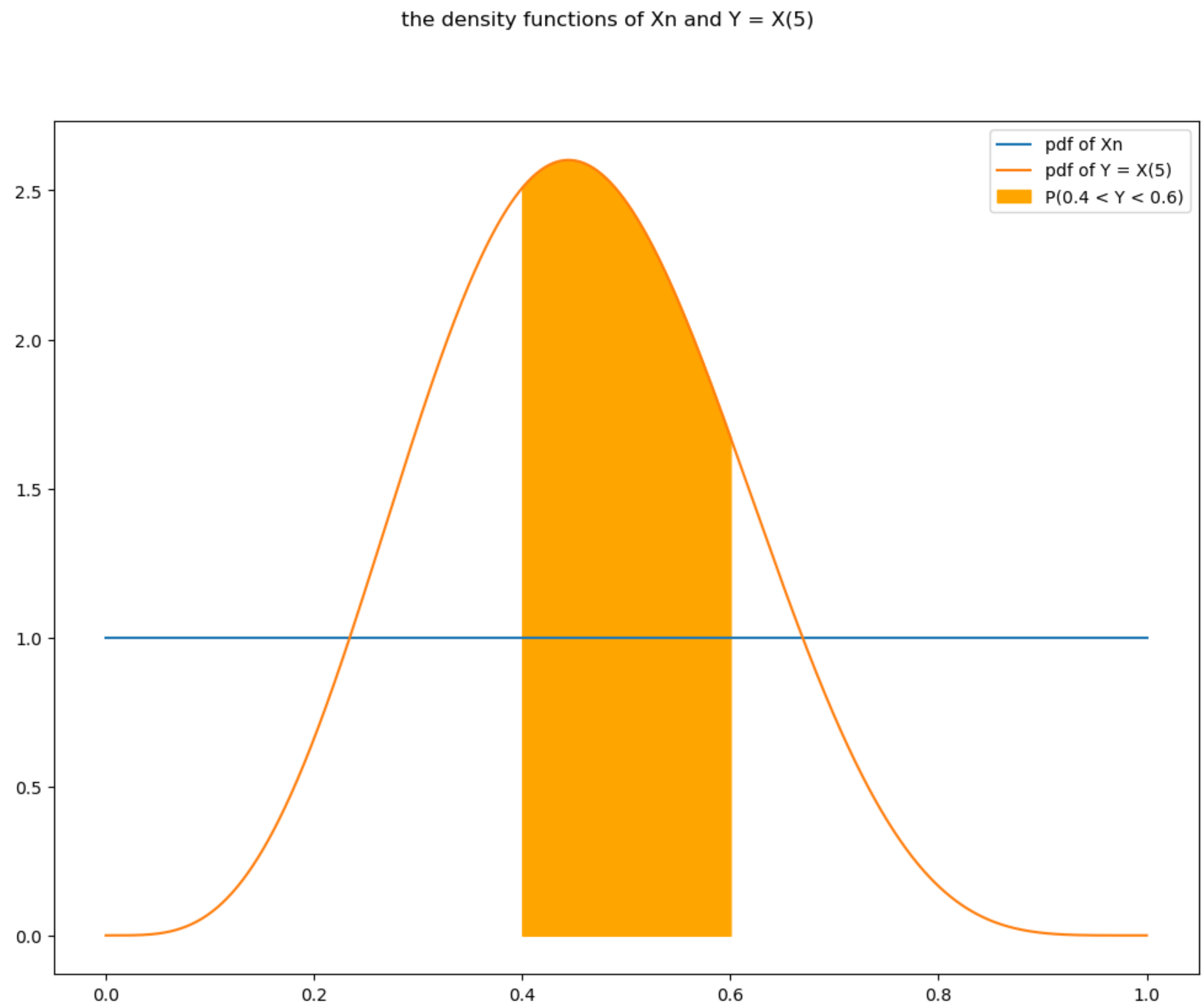
Problem 2:

Let random variables X1, X2, ..., Xn ~ U[0, 1]

In [ ]:
```
# pdf of U[0, 1]
fx2 = lambda x: 1 if (x >= 0 and x <= 1) else 0

# pdf of X(5) when n = 10
```

```
fx2_k5 = partial(order_pdf, k = 5, n = 10, func = fx2)

x = linspace(0,1,1000)
y_fx2 = [fx2(z) for z in x]
y_fx2_k5 = [fx2_k5(z) for z in x]

fig, ax = subplots(figsize=(12, 9))
fig.suptitle("the density functions of Xn and Y = X(5)")
ax.plot(x, y_fx2, label="pdf of Xn")
ax.plot(x, y_fx2_k5, label="pdf of Y = X(5)")

a, b = 0.4, 0.6
xshade = linspace(a, b, 100)
yshade = [fx2_k5(z) for z in xshade]
ax.fill_between(xshade, yshade, color="orange", label = "P(0.4 < Y < 0.6)")
legend = ax.legend()

print("2a: Y appears to follow a normal distribution")
print(f"P({a} < Y < {b}) =", quad(fx2_k5, 0.4, 0.6)[0])
```

```
2a: Y appears to follow a normal distribution
P(0.4 < Y < 0.6) = 0.4668646399999999
```

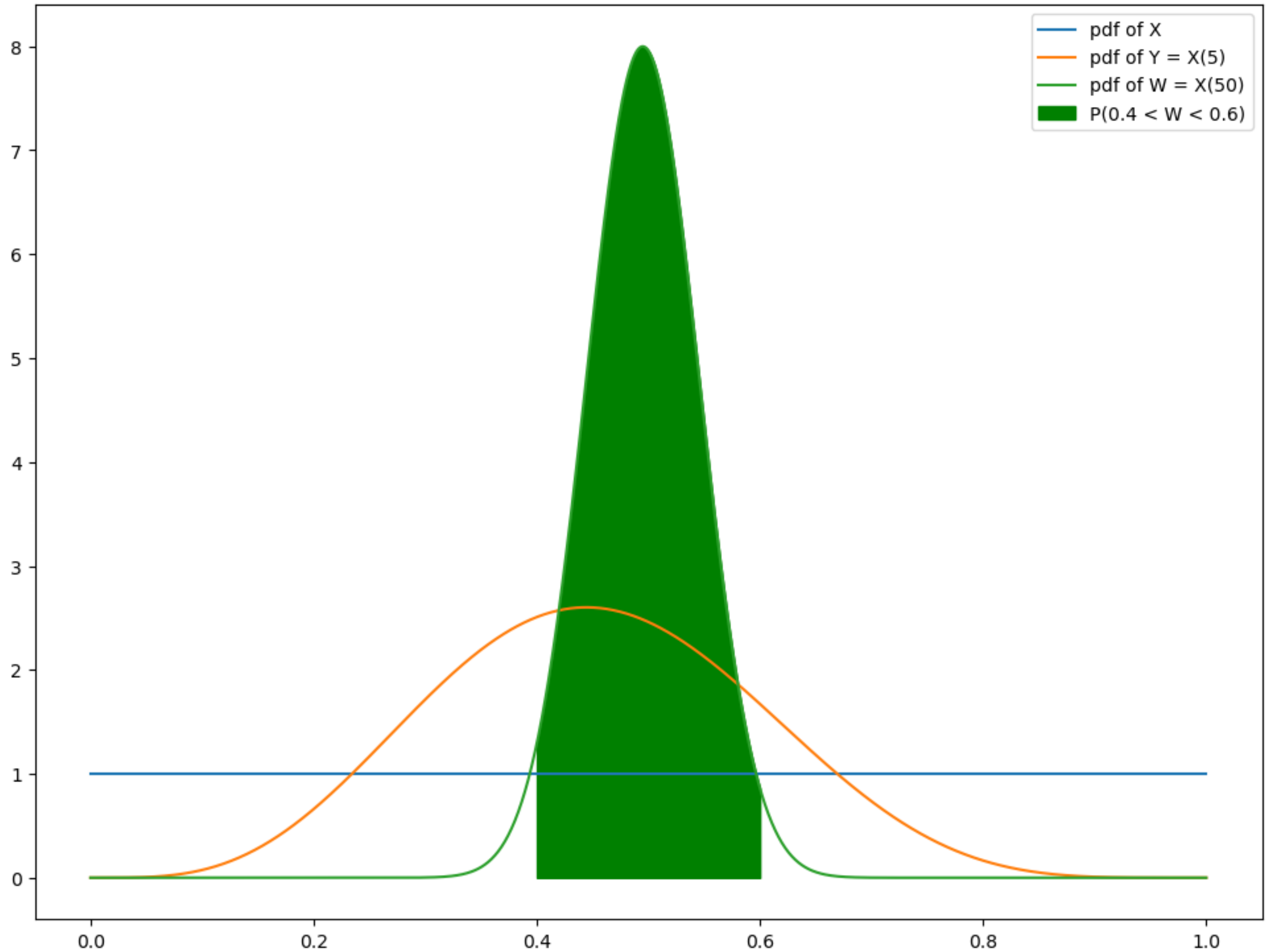


In [ ]:

```
fx2_k50 = partial(order_pdf, k = 50, n = 100, func = fx2)
y_fx2 = [fx2(z) for z in x]
y_fx2_k50 = [fx2_k50(z) for z in x]

fig, ax = subplots(figsize=(12, 9))

ax.plot(x, y_fx2, label="pdf of X")
ax.plot(x, y_fx2_k5, label="pdf of Y = X(5)")
ax.plot(x, y_fx2_k50, label="pdf of W = X(50)")

yshade = [fx2_k50(z) for z in xshade]
ax.fill_between(xshade, yshade, color="green", label = "P(0.4 < W < 0.6)")
```

```
legend = ax.legend()

print("2b: W appears to follow a tighter normal distribution")
theoretical_result = quad(fx2_k50, 0.4, 0.6)[0]
print(f"P({a} < Y < {b}) =", theoretical_result)
```

```
2b: W appears to follow a tighter normal distribution
P(0.4 < W < 0.6) = 0.9561391157398291
```



2c: To simulate P(0.4 < W 0.6), we'll start by creating a uniform random variable using scipy.stats.uniform

In [ ]:
```
uni_rv = uniform()
```

To calculate an order statistic, we'll create function get_order_stat.

This function will work by using the input random variable to generate an array of values. The function will then sort the array (ascending) and return the value at index k - 1 (the "kth" value).

In [ ]:
```
def get_order_stat(k, n, rv):
    rvs = rv.rvs(n)
    rvs.sort()
    return rvs[k-1]
ex = get_order_stat(3, 5, uni_rv)
print(f"""example -> the 3rd smallest value from an array of 5 from a U[0,1] dist is {ex}""")
```

```
example -> the 3rd smallest value from an array of 5 from a U[0,1] dist is 0.9239081367099681
```

To do the simulation, we'll create the function "gen_val_array" that executes a function over a list of input length "size". We'll also create "filter_val_array", which filters the values from the above function within specified bounds. Putting both together, "sim_discrete_prob" will create the array of values, create another list of filtered values, and return the proportion of filtered values to the original array.

In [ ]:
```
gen_val_array = lambda func, size: [func() for i in range(size)]
filter_val_array = lambda a, b, array: [x for x in array if (x > a and x < b)]

def sim_discrete_prob(a, b, func, size):
```

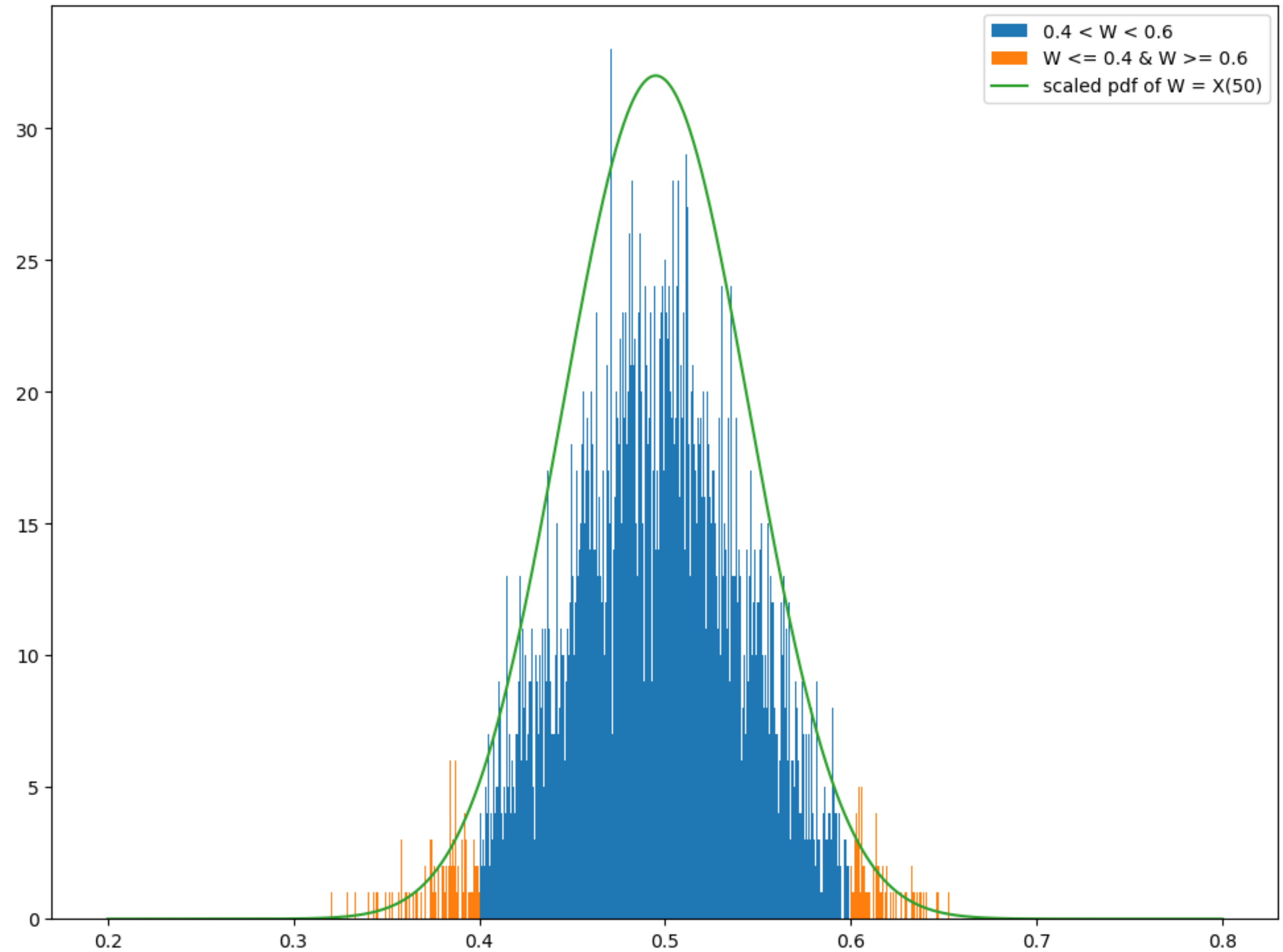
```python
    all_vals = gen_val_array(func, size)
    in_vals = filter_val_array(a, b, all_vals)
    return len(in_vals) / len(all_vals)
```

In [ ]:

```python
k = 50
n = 100
a = 0.4
b = 0.6
func = lambda: get_order_stat(k, n, uni_rv)
size = 25_000
bins = np.arange(0.2, 0.8, 0.0001)

simulated_result = sim_discrete_prob(a, b, func, size)
print("2c: simulated P(0.4 < W < 0.6) =", round(simulated_result, 8))
print("2c: theoretical P(0.4 < W < 0.6) =", round(theoretical_result, 8))

fig, ax = subplots(figsize=(12, 9))
fig.suptitle(f"hist of {size} generated vals from X({k}), n = {n}")

array = gen_val_array(func, size)
in_array = filter_val_array(a, b, array)
out_array = [y for y in array if y not in in_array]

in_hist = ax.hist(in_array, bins=bins, label = f"0.4 < W < 0.6")
out_hist = ax.hist(out_array, bins=bins, label = f"W <= 0.4 & W >= 0.6")

vals_per_bin = int(size/len(bins))
wx = bins
wy = [fx2_k50(z) * vals_per_bin for z in wx]

ax.plot(wx, wy, label="scaled pdf of W = X(50)")

legend = ax.legend()
```

```
2c: simulated P(0.4 < W < 0.6) = 0.95596
2c: theoretical P(0.4 < W < 0.6) = 0.95613912
```

hist of 25000 generated vals from X(50), n = 100



Problem 3:

Let random variables X1, X2, ..., Xn ~ iid N(100, 100)

We'll use scipy.stats.norm to create a random variable object.

In [ ]:

```
mean, var, stddev = 100, 100, 10
norm_rv = norm(loc=mean, scale=stddev)

# confirmed vals using function
def norm_pdf(x, mean=mean, stddev=stddev):
    term1 = 1 / (stddev * sqrt(2 * pi))
    exp = (-0.5) * (((x - mean) / stddev) ** 2)
    return term1 * (e ** exp)
norm_cdf = lambda x: quad(norm_pdf, 0, x)[0]

print("rv", norm_rv.cdf(120))
print("calc", norm_cdf(120))
```

testing scipy.stats object vs my own function

```
rv 0.9772498680518208
calc 0.9772498680518209
```

P(X1 > 120) = 1 - P(X1 < 120)

In [ ]:

```
prob_x_over120 = 1 - norm_cdf(120)
print("3a: P(X > 120) = 1 - P(X < 120) =", prob_x_over120)
```

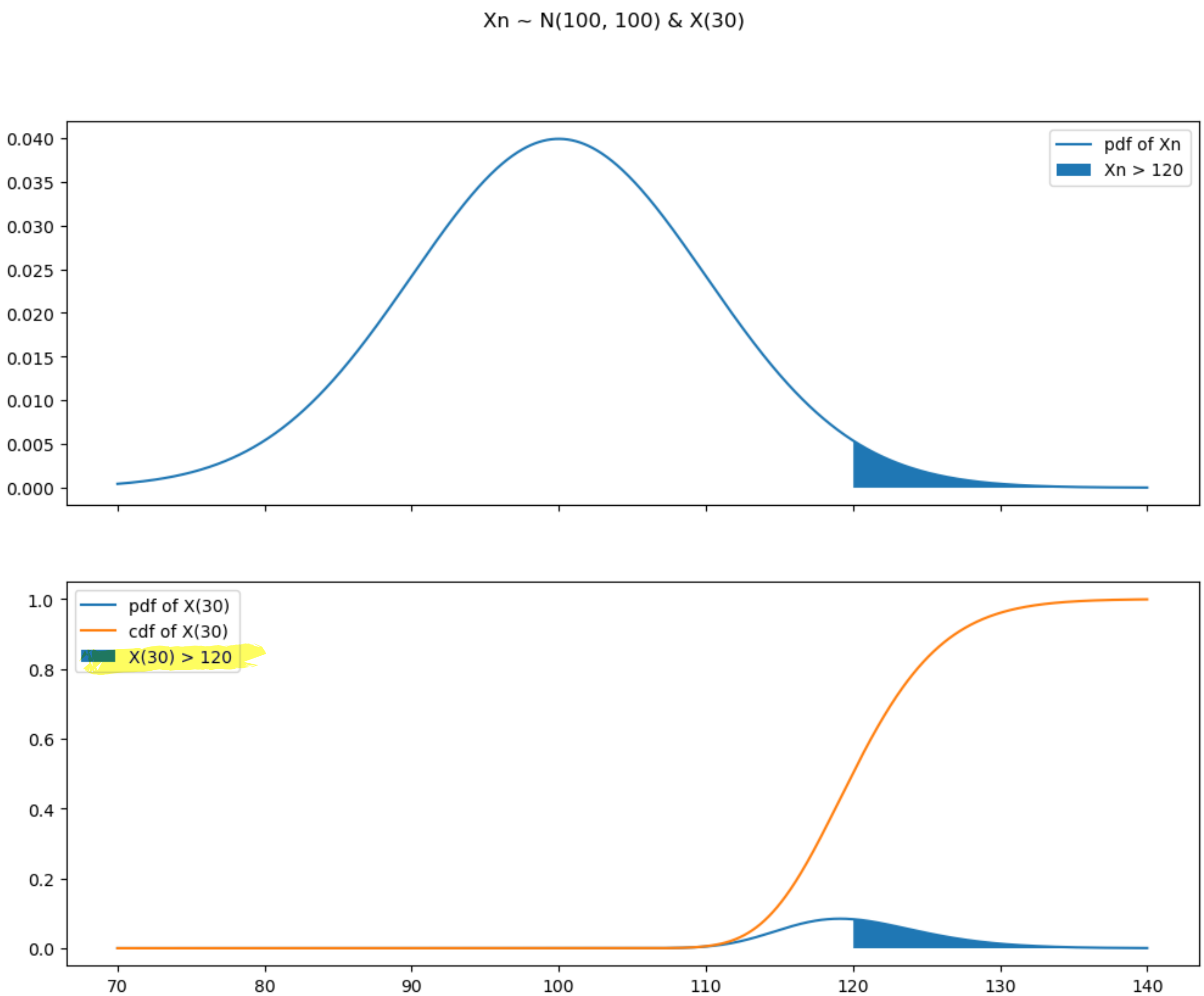
```
3a: P(X > 120) = 1 - P(X < 120) = 0.022750131948179098
```

In [ ]:

```
fx3_k30 = partial(order_pdf, k = 30, n = 30, func = norm_pdf)
Fx3_k30 = lambda x: quad(fx3_k30, 0, x)[0]

x = linspace(mean - 3 * stddev, mean + 4 * stddev, 1_000)
```

```
y = [norm_pdf(z) for z in x]
yk = [fx3_k30(z) for z in x]
Yk = [Fx3_k30(z) for z in x]

shadex = [z for z in x if z > 120]
shadey0 = [norm_pdf(z) for z in shadex]
shadey1 = [fx3_k30(z) for z in shadex]

fig, ax = subplots(nrows=2, sharex=True, figsize=(12, 9))
fig.suptitle("Xn ~ N(100, 100) & X(30)")

ax[0].plot(x, y, label = "pdf of Xn")
ax[1].plot(x, yk, label = "pdf of X(30)")
ax[1].plot(x, Yk, label = "cdf of X(30)")

ax[0].fill_between(shadex, shadey0, label = "Xn > 120")
ax[1].fill_between(shadex, shadey1, label = "X(30) > 120")
legend = ax[0].legend()
legend = ax[1].legend()
prob_fx3_k30_over120 = 1 - Fx3_k30(120)
print("3b: P(X(30) > 120)", prob_fx3_k30_over120)
```

```
3b: P(X(30) > 120) 0.49861814363845225
```

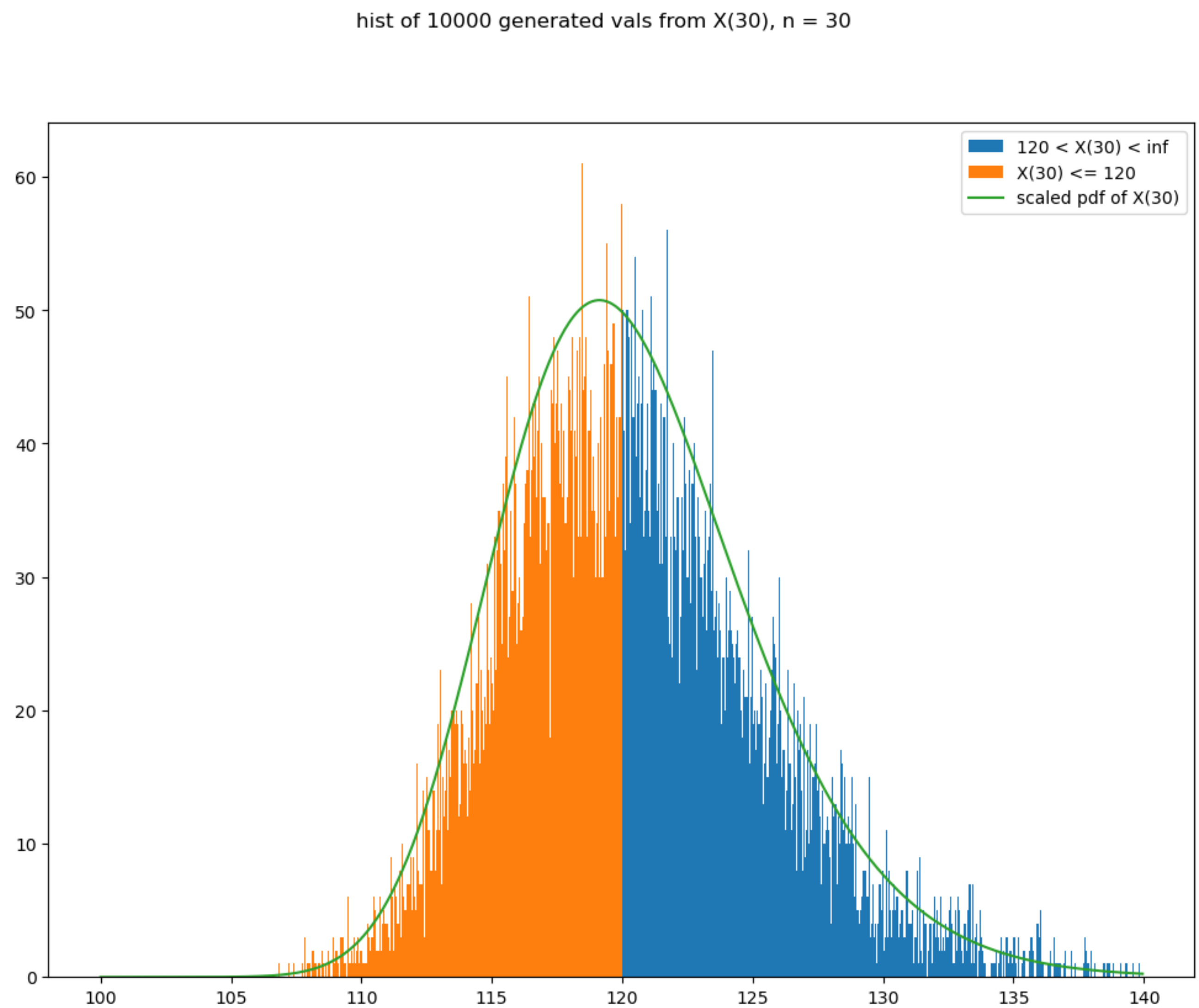


In [ ]:

```
k = 30
n = 30
a = 120
b = inf

func = lambda: get_order_stat(k, n, norm_rv)
size = 10_000

bins = np.arange(100, 140, 0.05)
simulated_result = sim_discrete_prob(a, b, func, size)
```

```
print(f"3c: with n = {n}, simulated P(X({k}) > {a}) =", round(simulated_result, 8))
print(f"3c: with n = {n}, theoretical P(X({k}) > {a}) =", round(1 - Fx3_k30(120), 8))

fig, ax = subplots(figsize=(12, 9))

fig.suptitle(f"hist of {size} generated vals from X({k}), n = {n}")

array = gen_val_array(func, size)
in_array = filter_val_array(a, b, array)
out_array = [y for y in array if y not in in_array]

in_hist = ax.hist(in_array, bins=bins, label = f"{a} < X({k}) < {b}")
out_hist = ax.hist(out_array, bins=bins, label = f"X({k}) <= {a}")

vals_per_bin = int(size/len(bins))
wx = bins
wy = [fx3_k30(z) * 600 for z in wx]

ax.plot(wx, wy, label=f"scaled pdf of X({k})")

legend = ax.legend()
```

```
3c: with n = 30, simulated P(X(30) > 120) = 0.5046
3c: with n = 30, theoretical P(X(30) > 120) = 0.49861814
```



In [ ]:

```
k = 100
n = 100
a = 120
b = inf

fx3_k100 = partial(order_pdf, k=k, n=n, func = norm_pdf)
Fx3_k100 = lambda x: quad(fx3_k100, 0, x)[0]

func = lambda: get_order_stat(k, n, norm_rv)
size = 10_000
```

```python
bins = np.arange(110, 150, 0.1)
simulated_result = sim_discrete_prob(a, b, func, size)

print(f"3d: with n = {n}, simulated P(X({k}) > {a}) =", round(simulated_result, 8))
print(f"3d: with n = {n}, theoretical P(X({k}) > {a}) =", round(1 - Fx3_k100(120), 8))

fig, ax = subplots(figsize=(12, 9))

fig.suptitle(f"hist of {size} generated vals from X({k}), n = {n}")

array = gen_val_array(func, size)
in_array = filter_val_array(a, b, array)
out_array = [y for y in array if y not in in_array]

in_hist = ax.hist(in_array, bins=bins, label = f"{a} < X({k}) < {b}")
out_hist = ax.hist(out_array, bins=bins, label = f"X({k}) <= {a}")

wx = bins
wy = [fx3_k100(z) * len(bins) * 3 for z in wx]

ax.plot(wx, wy, label=f"scaled pdf of X({k})")

legend = ax.legend(loc="upper right")
```

```
3d: with n = 100, simulated P(X(100) > 120) = 0.8938
3d: with n = 100, theoretical P(X(100) > 120) = 0.8998705
```