UNIVERSITÄT BIELEFELD
Faculty of Business Administration and Economics

DECISION ANALYTICS

Combining OR and Data Science

Summer Term 2022

2. Introductory Case Study, Representing and Computing with Uncertain Quantities

J-Prof. Dr. Michael Römer, Till Porrmann
Decision Analytics Group | Bielefeld University

Overview

- Introducing a case study
- Representing uncertainty
- Computing with uncertain quantities
- Decision-making under uncertainty

A First (and Recurring) Case Study: Capacity Planning

We will consider the following case study:

> *A marketing manager is asked by her boss to forecast demand for a new-generation microchip. The manager builds a Data Science model: She forecasts that demand will lie between 50,000 and 150,000 units. However, the boss insists: "Give me a **number**! My people need to build a production line with a certain capacity!" The marketing manager provides him with her best guess - the average, that is, 100,000 units*

Case Study: The Boss' Spreadsheet

The boss plugs that number into his calculation spreadsheet and is happy:

Fixed Capacity Cost per Unit	$30
Contribution Margin per Unit	$40
Actual Demand (Uncertain)	100.000
Decision: Production Capacity	100.000
Sales	100.000
Total Contribution Margin	$4.000.000
Fixed Cost of Total Capacity	$3.000.000
Profit	$1.000.000

Question: How can we compute the total profit for a given installed capacity and demand?

Case Study: A Simple Model

Calculating total profit

Total Profit = - Capacity Installation Cost + Margin from Sales

A simple model

Total Profit $g = f(x, d) = -30x + 40min(x, d)$

where

x: installed capacity in units (decision)

d: demand in units

In Python:

In [18]:

```
def total_profit(capacity, demand):
    return -30*capacity + 40* min(capacity, demand)
```

Case Study: The Boss' Deterministic Decision

- The boss uses a given estimated demand $d = 100$
  (to enhance readability, we calculate with units of 1000 items),
- Using $x := d$ results in:

Total Profit $g = f(100, 100) = 1000$

In [19]:
```
demand = 100
capacity = demand
total_profit(capacity, demand)
```

Out[19]:

```
1000
```

The determinstic case is easy - but what to do in case of **uncertain demand**?

Overview: Part I of the course

Combining OR and Data Science for Planning under Uncertainty
We learn how to solve the capacity planning problem (and more):

- Representing uncertainty: Probability distributions and how to obtain them
- Calculating under uncertainty: Computing functions of uncertain variables
- Deciding under uncertainty: Taking "simple" decisions under uncertainty
- Optimizing under uncertainty: Chance-constrained and Stochastic Programming
- Learning and optimizing under uncertainty: Combining Machine Learning and Optimization under Uncertainty

Decision Making under Uncertainty

Planning and decision making affects the future

- e.g., we decide upon installing capacity affecting future production volume In general, the future is affected by uncertainty
- e.g. customer demand is not known in advance

→ Planning and decision making under uncertainty is the rule, not an exception or some special case

Dealing with Uncertainties

Facing uncertainty in important parameters such as demand or resource availability we may consider them in our decision-making process as:

- point estimates such as averages
- scenarios (best case, average case, worst case) without probabilities
- (approximations of) probability distributions

Representing Uncertain Numbers with Point Estimates

Use a single value for each uncertain parameter

- estimate the expected value
- plug it into a deterministic decision-making model
- often called **predict-then-optimize**

Disadvantages:

- ignores different outcomes and their probabilities
- in general, leads to the flaw of averages - does not yield the expected outcome of a decision

... not the best approach, but widely used in practice

A Limited Number of Scenarios

Create a limited number of scenarios

- e.g.: worst case, average, best case
- in general, scenarios are not assigned probabilities
- mainly used for what-if-analysis

Disadvantages:

- for each scenario, a different decision may be optimal
- little guidance on how to take a final decision

Using Probability Distributions

Our approach will be to

- represent uncertain quantities by **probability distributions** (distributional instead of point forecasts)
- compute with probability distributions using **Monte Carlo / sample approximations**
- determine decisions using **enumeration**, **unconstrained optimization** and **stochastic linear programming**

Back to the Case Study: Dealing with Uncertain Demands

- from the case study description, we know that demand is not a deterministic value d, but affected by uncertainty
- in that case, what total profit can the boss expect for his capacity decision?
- in case of uncertainty, is $ 100 really the best decision?

To answer these questions, we will need to

- model the uncertain demand
- be able to compute a function with an argument affected by uncertainty
- find a way to determine the capacity decision with the best expected profit

Representing Uncertainty using Distributions

In this course, we will represent uncertainty using **Probability Distributions**
Given a **random variable**

- with a discrete (finite) or continuous **sample space (state space)** of possible outcomes,
- a **probability distribution** gives the **probabilities** of these outcomes such that
- the total probability of all outcomes is 1

Probability Distributions in Python

- in Python, there are various libraries implementing distributions
- we will use *Scipy Stats* here

Discrete Distributions

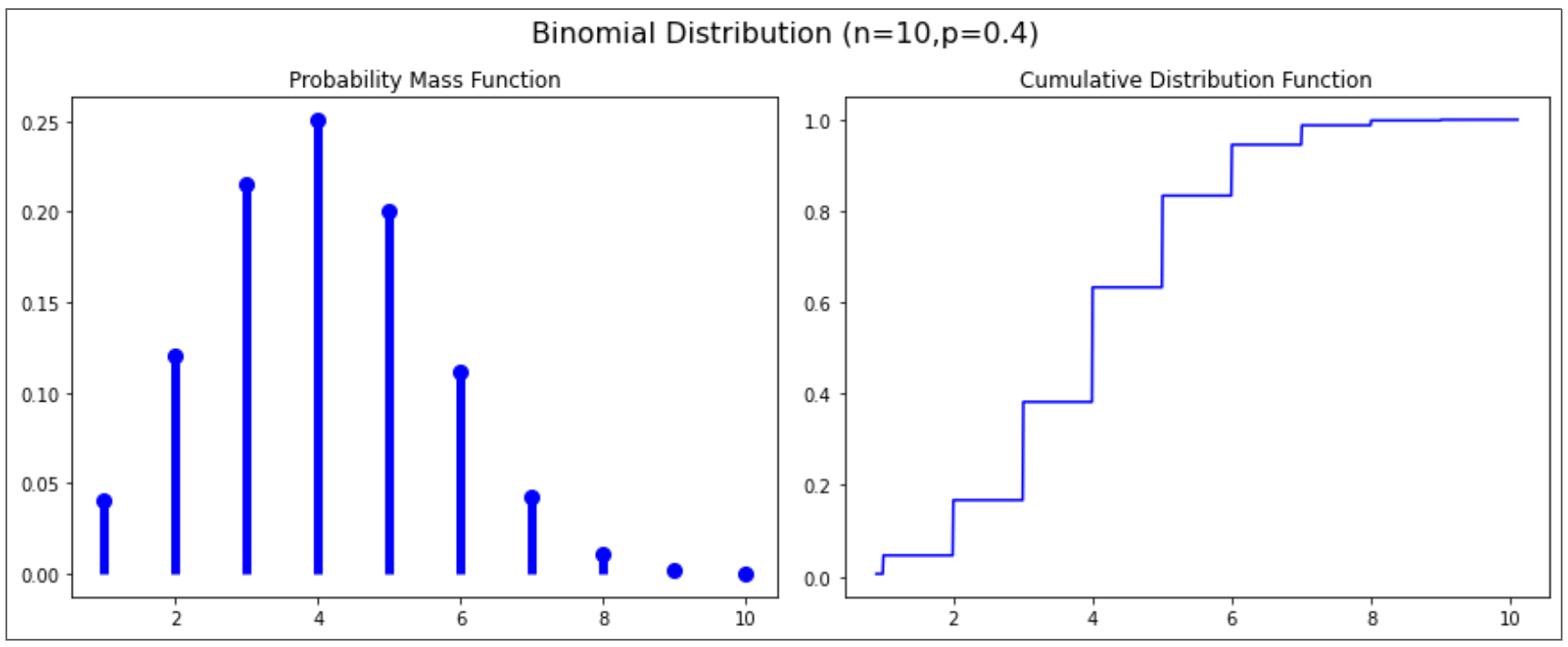
For a **discrete** random variable X with state space $\mathcal{X}$

- the *probability mass function* $p(x)$ gives the probability of obtaining value $x \in \mathcal{X}$
- the *cumulative distribution function* $F(x)$ gives the probability of obtaining a value smaller than x, or, more formally: $F(x) = \sum_{a \in \mathcal{X}: a \leq x} p(x)$

Example: Binomial Distribution

In [21]:

```
from scipy.stats import binom
#distribution parameters

n, p = 10, 0.4

fig, (ax1,ax2) = plt.subplots(1, 2, figsize=(12, 4.8), constrained_layout=True)
fig.suptitle(f'Binomial Distribution (n={n},p={p})',fontsize=16)


xx = np.arange(binom.ppf(0.01, n, p), binom.ppf(1, n, p)+1) # the range of x-values to display, (ppf is the inverse cumulative distribution function.)


ax1.plot(xx, binom.pmf(xx, n, p), 'bo', ms=8)
ax1.vlines(xx, 0, binom.pmf(xx, n, p), color='b', lw=5) # pmf is the probability mass function
ax1.set_title('Probability Mass Function')

xx = np.linspace(binom.ppf(0.01, n, p)-0.1, binom.ppf(1, n, p)+0.1,1000)
ax2.plot(xx, binom.cdf(xx, n, p), 'b')
ax2.set_title('Cumulative Distribution Function')

plt.show()
```



Continous Distributions

For a **continuous** random variable X with state space $\mathcal{X}$

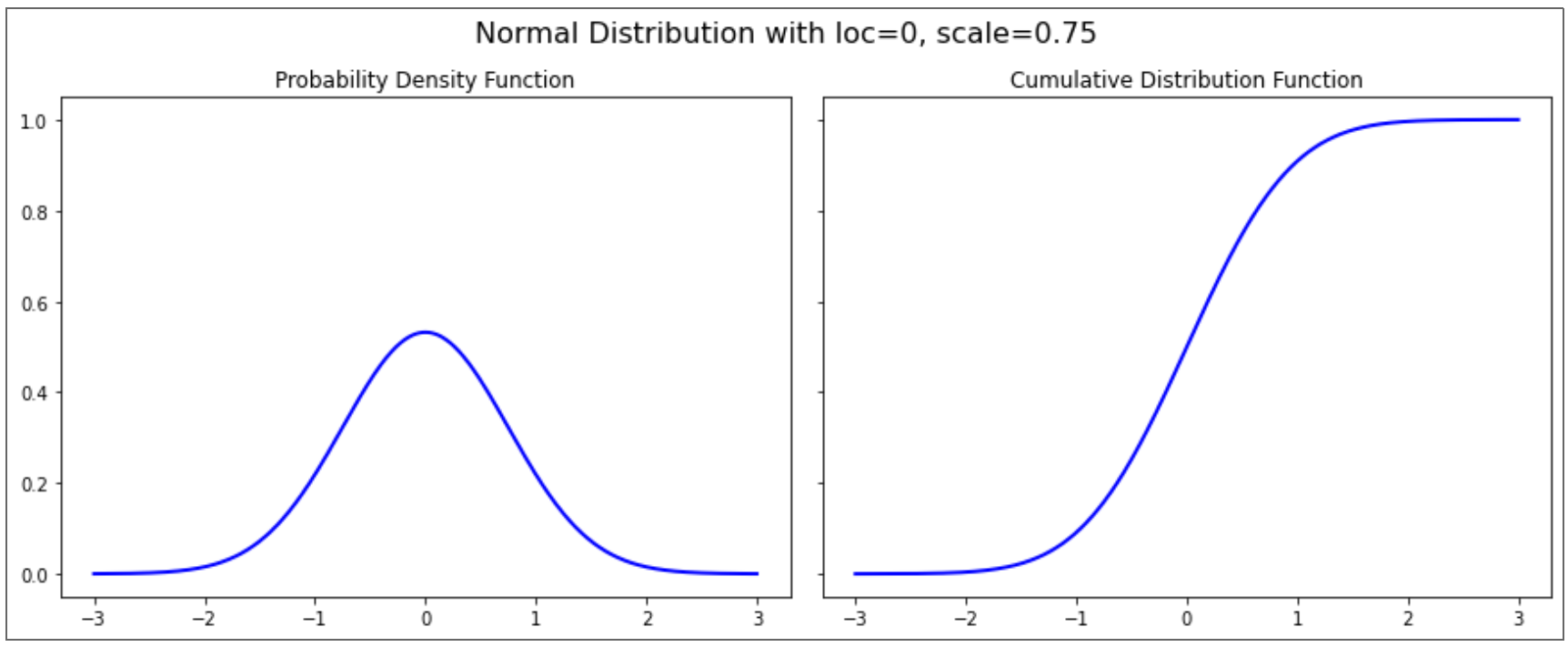
- the **cumulative distribution function** $F(x)$ gives the probability of a value smaller than x: $F(x) = p(X \leq x)$
- the **probability density function** $f(x)$ is the derivative of $F(x)$ and can be used to calculate the probability of obtaining a value x within an interval $[a, b]$, that is, $p(a \leq x \leq b)$:

$$\int_a^b f(x)dx = F(b) - F(a)$$

Example: Normal Distribution

In [29]:

```
from scipy.stats import norm

#parameters
loc = 0 # mean
scale = 0.75 # standard deviation

normal_dist = norm(loc,scale) ## here, we "freeze" the distribution

fig, (ax1,ax2) = plt.subplots(1, 2, figsize=(12, 4.8), constrained_layout=True, sharey=True)
fig.suptitle(f'Normal Distribution with loc={loc}, scale={scale}',fontsize=16)

xx = np.linspace(-3, 3, 1000)

ax1.plot(xx, normal_dist.pdf(xx), linewidth=2, color='b')
ax1.set_title('Probability Density Function')

ax2.plot(xx, normal_dist.cdf(xx), linewidth=2, color='b')
ax2.set_title('Cumulative Distribution Function')


plt.show()
```



Distribution Statistics

SOME STATISTICS AND HOW TO COMPUTE THEM FOR DISCRETE DISTRIBUTIONS:*

- **Expected Value / Mean**: $\mu = E(X) = \sum_{x \in \mathcal{X}} p(x)x$
- **Variance**: $\sigma^2 = E((x-\mu)^2) = \sum_{x \in \mathcal{X}} p(x)(x-\mu)^2$
- **Standard deviation**: σ
- **Mode:** $\arg\max_{x \in \mathcal{X}} p(x)$
- α**-quantile**: $x_\alpha : P(X \leq x_\alpha) = F^{-1}(\alpha)$
- **Median:** 0.5-quantile

In **scipy.stats**, we can obtain these statistics as follows:

In [20]:
```
print(f'Expected value: { normal_dist.mean() }, variance: { normal_dist.var() }, standard deviation: { normal_dist.std() }')
print (f'Quantiles: 0.05: { normal_dist.ppf(0.05) }, 0.5(median): { normal_dist.median() }, 0.05: { normal_dist.ppf(0.95) }')
```

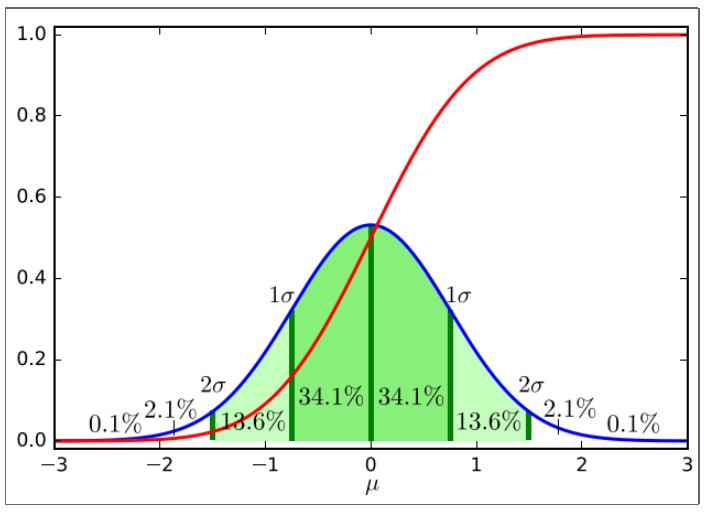
```
Expected value: 0.0, variance: 0.5625, standard deviation: 0.75
Quantiles: 0.05: -1.2336402202136045, 0.5(median): 0.0, 0.05: 1.23
3640220213604
```

Parametric Distributions

- the distributions we saw so far form so-called **parametric distributions**
- they all have different assumptions and "stories" - it is good to know at least some basic properties about distributions
- a great web resource for parametric distributions and their stories is the **Distribution Explorer**
- if we have data, we can select a distribution and fit it to the data (estimate the best parameters), e.g. using the `fit` function from scipy.stats
- if we do not find a parametric distribution that fits to the data, we may as well resort to **non-parametric** distributions

Caution: Fitting a single distribution to a given data set assumes that the data is **i.i.d.** (independent and identically distributed)

An Example for Useful Knowledge About Distributions



- **68-95-99.7 Rule:** Shorthand rule to remember the percentage of probability mass that lies around the mean.

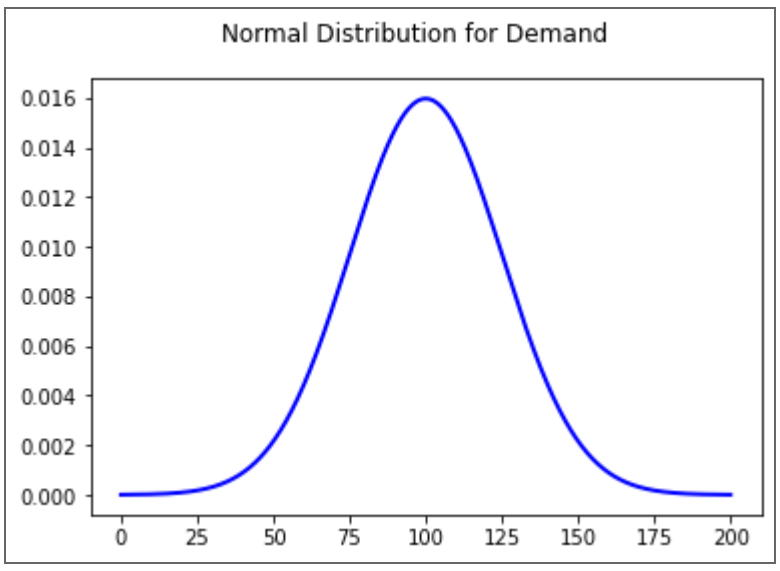
Back to the Case Study

Let us assume that in the case study, the data scientist considers modeling demand as a **Normal Distribution**:

- the data scientist does not think that each demand is equally likely
- she thinks that the mean is 100 000
- she thinks that the distribution is symmetric and about 2/3 of the probability mass lies within 100 000 $\pm$ 25 000 $\rightarrow \sigma$ = 25 000

In [23]:
```
expected_demand = 100
standard_devation_demand = 25
demand_dist = stats.norm(expected_demand, standard_devation_demand)
```

In [24]:
```
xx = np.linspace(0, 200, 1000)
plt.plot(xx, demand_dist.pdf(xx), linewidth=2, color='b', label='pdf')
plt.suptitle('Normal Distribution for Demand')

plt.show()
```



Computing with Uncertain Quantities

Computing Functions of Random Variables

THE PROBLEM WE ADDRESS IS:

Given an arbitrary function $f(d)$ with a parameter d (or multiple parameters) and a random variable D (or multiple random variables), what is the distribution of $Y = f(D)$?

ANALYTICALLY?

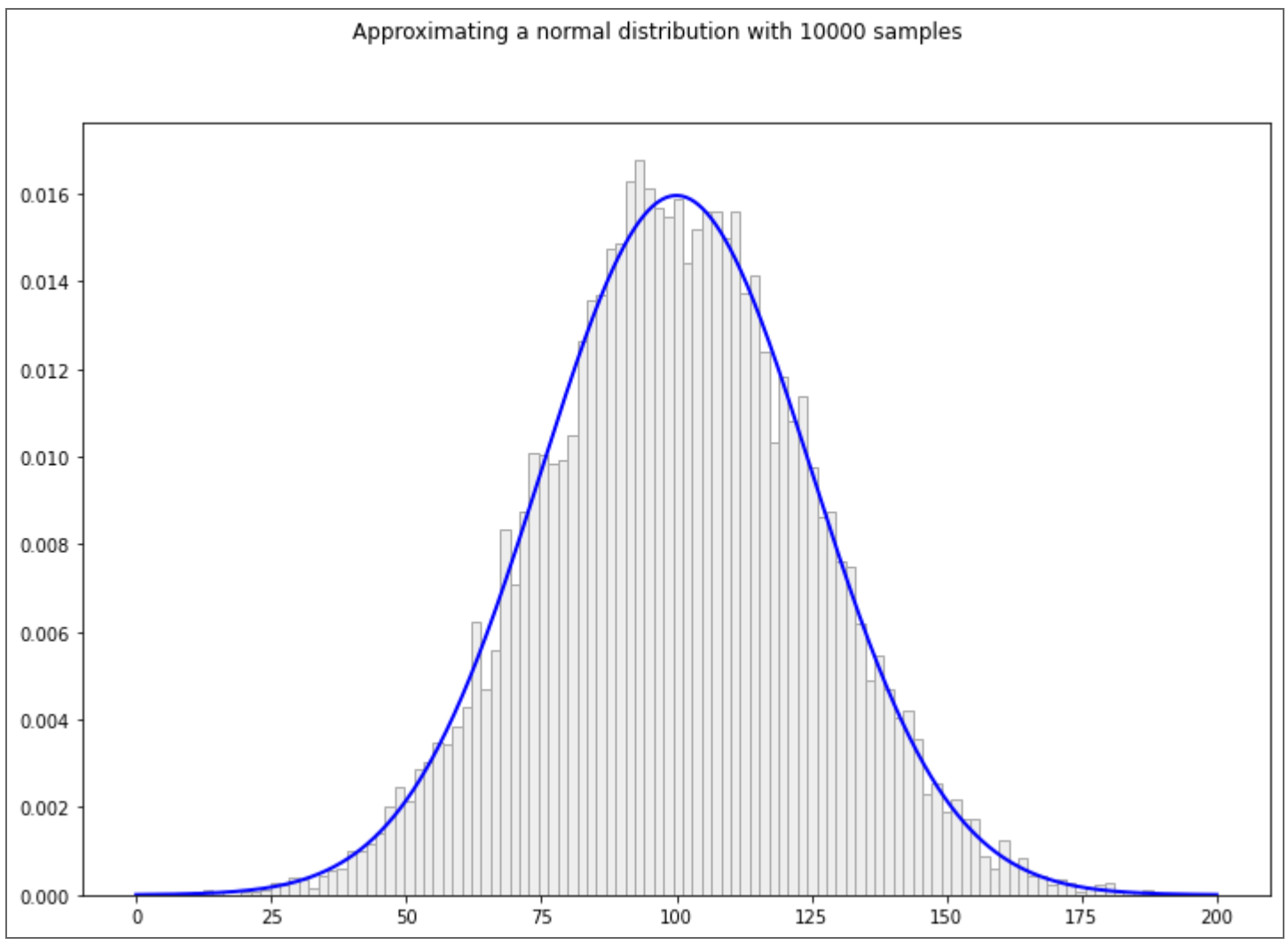
- There are analytical techniques like the so-called *change of variables* method,
- but they quickly become difficult in case of involved multi-variate problems.

Sampling From Distributions

- many techniques in computational statistics are based on *sampling*, that is, on drawing samples from a distribution
- these samples are used for transforming distributions and performing probabilistic inference

Example: Sampling from a Normal Distribution:

In [73]:
```
number_of_samples= 10000
demand_samples = demand_dist.rvs(number_of_samples)
xx = np.linspace(0, 200, 1000)

plt.figure(figsize=(12,8))
plt.plot(xx, demand_dist.pdf(xx), linewidth=2, color='b', label='pdf')
plt.hist(demand_samples, bins=100, density=True, color='#EEEEEE', edgecolor="#AAAAAA")
plt.suptitle('Approximating a normal distribution with 10000 samples')
plt.show()
```



Task: Play with the sample size to see the effect on approximation quality!

Monte-Carlo-Simulation / -Approximation for Computing with Random Variables

The key idea

- use sampling to approximate the **result distribution** of a function f
- by simply computing the f for a set of samples from the dributions of the uncertain (input) parameters

In our capacity planning example

- let d_s be a sample from the demand distribution
- then $g(x)_s = f(x, d_s)$ is a sample from the distribution of the total profit for a given capacity decision x
- we can thus approximate the expected profit $E(G(x))$ of installing a capacity x as:

$$E(G(x)) \approx \frac{1}{|S|} \sum_{s \in S} g(x)_s = \frac{1}{|S|} \sum_{s \in S} f(x, d_s)$$

Monte-Carlo Simulation for our Case Study: A First Implementation

Here, we implement a first loop-based implementation of a Monte-Carlo Simulation for our example case study:

- we draw a new random number in each iteration of a for loop
- and store the profit sample in an array

In [42]:
```
demand_dist = stats.norm(100,25) # demand distribution
capacity = 100 # given capacity decisions
n_samples = 10000

profit_samples = np.zeros(n_samples) # initialize the result array

# the simulation loop
for s in range(n_samples):
    profit_samples[s] = total_profit(capacity, demand_dist.rvs())

expected_profit = np.mean(profit_samples)
print(f'The estimated expected profit is: {expected_profit:.2f}')
```

```
The estimated expected profit is: 595.76
```

The Flaw of Averages

THE BOSS WAS WRONG:

While the boss' calculation returned a profit of around $ 1 m, our simulation reveals that the expected profit is only around \$ 600 tsd
$\rightarrow$ The boss is a victim of the **Flaw of Averages**:

> **The Flaw of Averages:** The results obtained when replacing uncertain quantities with averages are wrong on average.

> *In general, if D is an random variable, and f is a nonlinear function, then*
>
> $$f(E(D)) \neq E(f(D))$$
>
> *Plugging an average/expected value of an random variable into a function does **not** yield the xpected value of that function!*

Do we always overestimate?

Your company is considers purchasing a natural gas reservoir containing a million units of gas. At the moment, the gas price is $10, and it costs you \$9.50 to pump a unit of gas to the market.
The gas price is uncertain and the reservoir can only start operating in a month. At that time, the price may be multiple dollars higher or lower than $10.
Your boss wants you to estimate the gas price in one month in order to valuate the reservoir. In particular, he wants a point estimate: The expected gas price.

Excercise:

- Assume that next month's gas price follows a normal distribution with $\mu = 10$ and $\sigma = 3$
- Flaw of averages analysis: Compare the value of the reservoir obtained by calcuating with the expected price to its (Monte-Carlo-approximated) true value
- Consider that in case of low prices, you have the option *not* to pump gas
- Plot the reservoir value as a function of the gas price

Vector-Oriented Monte Carlo Approximation

- our first implementation was based on drawing random demands **within** a for-loop
- an equivalent implementation would be to draw an array (a vector) of random demands **beforehand** and use them in the implementation
- this is useful for
    - reproducibility
    - variance reduction (common random numbers) when using a Monte-Carlo approximation for decision-making / optimization
    - efficient vectorized implmementations of Monte-Carlo approximations

Some Notation:

Given a random variable, e.g. the random demand D

- d_s is the sample wiht index s
- $[d_s]_{s \in S}$, shortly written as $[d_s]$, is the sample vector $\begin{bmatrix} d_1 \\ \vdots \\ d_{|s|} \end{bmatrix}$
- if the context is clear, we may also use $\mathbf{d} := [d_s]$

where

- $|S|$ is the number of samples / scenarios
- S is the (ordered) index set of samples, $S = \{1 \dots |S|\}$

Vector-Oriented Monte Carlo

Given

- a deterministic function $f(d)$ and
- a random variable (a probability distribution) D we wish to compute the distribution $G = f(D)$

Let $\mathbf{d} = [d_s]$ be a vector of samples from D.
We obtain a Monte Carlo approximation $\mathbf{g}$ of G by applying f **element-wise:**

$$\mathbf{g} = [g_s] = \begin{bmatrix} g_1 \\ \vdots \\ g_{|s|} \end{bmatrix} = [f(d_s)] = \begin{bmatrix} f(d_1) \\ \vdots \\ f(d_{|S|}) \end{bmatrix}$$

We will write $\mathbf{f}$ to indicate element-wise application of f to a vector:
$\mathbf{g} = \mathbf{f}(\mathbf{d}) = [f(d_s)]$
And we will call $\mathbf{f}$ the **vectorized** version of f.

Illustration: Numerical Example

The deterministic function for computing the total profit given capacity decision x and deterministic demand d:

$g = f(x, d) = -30x + 40\min(x.\, d)$

Let us fix x to 100 am the use the sample vector $\mathbf{d} = \begin{bmatrix} 85 \\ 100 \\ 115 \end{bmatrix}$

Then, we can compute $\mathbf{g} = \mathbf{f(100, d)} = \begin{bmatrix} -3000 + 40\min(100, 85) \\ -3000 + 40\min(100, 100) \\ -3000 + 40\min(100, 115) \end{bmatrix} = \begin{bmatrix} 400 \\ 1000 \\ 1000 \end{bmatrix}$

..and we obtain an estimated expected total profit

$$\bar{\mathbf{g}} = \frac{1}{|S|}\sum_{s \in S} g_s = 800$$

Vectorizing Your Python Functions: Numba `@vectorize`

Numba

- is a just-in-time compiler that turns Python functions to C code
- specializing in Numpy-based code
- very simple to use since it providess so-called **decorators** you can add to your existing code

Vectorization

- numba provides the decorator `@vectorize` that
    - turns a function taking scalars in to a vectorized function that can support broadcasting (like numpy's ufuncs)
    - is very fast since it just-in-time compiles the code into C code

\

Vectorizing a Function in our Case Study

Take our function $f(x, d) = -30x + 40\min(x, d)$

In [43]:
```
def total_profit(capacity, demand):
    return -30*capacity + 40*min(capacity, demand)
```

Turn it into its vectorized counterpart $\mathbf{f}(x, d)$ by:

In [13]:
```
from numba import vectorize
@vectorize
def total_profit_vectorized(capacity, demand):
    return -30*capacity + 40*min(capacity, demand)
```

The function will now work with every combination of:

- x scalar, vector (array) or matrix (multi-dimensional array)
- d scalar, vector (array) or matrix (multi-dimensional array)

In [49]:
```
demand_sample = demand_dist.rvs(n_samples)
profit_sample = total_profit_vectorized(capacity, demand_sample)
np.mean(profit_sample)
```

Out[49]:

```
599.8986848901994
```

Comparing Speeds

Let us compare computation speed using timeit

In [76]:

```
            def monte_carlo_python_loop():
    profit_samples = np.zeros(n_samples)
    for s in range(n_samples):
        profit_samples[s] = total_profit(capacity, demand_dist.rvs())
    expected_profit = np.mean(profit_samples)
    return expected_profit

timeit_result_for_loop = %timeit -o monte_carlo_python_loop()
time_in_sec_for_loop = np.mean(timeit_result_for_loop.all_runs) / timeit_result_for_loop.loops

def monte_carlo_vectorized():
    demand_sample = demand_dist.rvs(n_samples)
    profit = total_profit_vectorized(capacity,demand_sample)
    expected_profit = np.mean(profit)
    return expected_profit


timeit_result_vectorized = %timeit -o monte_carlo_vectorized()
time_in_sec_vectorized = np.mean(timeit_result_vectorized.all_runs) / timeit_result_vectorized.loops

relative_time_for_loop =time_in_sec_for_loop / time_in_sec_vectorized
print(f'The for loop-based approach took {relative_time_for_loop:.02f} as much time as the vectorized approach')
```

```
627 ms ± 31.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop eac
h)
533 µs ± 63.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
The for loop-based approach took 1176.96 as much time as the vecto
rized approach
```

Finally: What is the Best Decision?

Decision-Making under Uncertainty: Formalization

We are looking for

- the decision (or decision vector, or more general, solution) x
- from the set of possible decisions (solutions) X
- yielding the *best* expected outcome $E(f(x, D))$
- given the uncertain/random variable(s) D

We can write this as an **optimization** problem under uncertainty:

$$\max_{x \in X} E(f(x, D))$$

Using Monte Carlo, we approximate $E(f(x, D))$ by the mean of the output sample vector $\mathbf{f}(x, \mathbf{d})$, that is by $\frac{1}{|S|} \sum_{s \in S} f(x, d_s)$ \

This results in the following optimization problem:

$$\max_{x \in X} \frac{1}{|S|} \sum_{s \in S} f(x, d_s)$$

Solving by Enumeration

IF

- the set X is finite and not too big (there are just few number of decisions / plans to choose from)
- and $\frac{1}{|S|}\sum_{s\in S} f(x, d_s)$ can be computed efficiently

... WE CAN SIMPLY

- **enumerate** all solutions / decisions $x \in X$
- and *select* one maximizing $\frac{1}{|S|}\sum_{s\in S} f(x, d_s)$

..in our example, we may simply enumerate all (meaningful) capacity decisions x, e.g. from 0 to 200.

Enumeration: Implementation

Task: Implement a for loop that chooses the best capacity decision and print the best decision and the best expected total profit (you may uses the function expected_profit from below)

In [51]:

```
capacities = np.arange(0,200) # the array / vector with the possible values
def expected_profit(capacity):
    return np.mean(total_profit_vectorized(capacity,demand_sample))
```

The Flaw Of Averages and Decision Making

When it comes to *decision making*, we encounter the

> **Strong Form of the Flaw of Averages:** Decisions and plans based on averages are wrong on average

More formally: In general, if x is a (vector of) decision variable(s), D is an random variable, and f is a nonlinear function, then

$$\underset{x}{\operatorname{argmax}}\, f(x, E(D)) \neq \underset{x}{\operatorname{argmax}}\, E(f(x, D))$$

That is, in general, the best average-based decision is usually different and worse than the decision with the best expected performance.

Conclusions

Today

- we had a short tour-de-force in decision-making under uncertainty with implementatios by Python
- using a simple case study

Next week

- we will have a closer look at how to **obtain probability distributions** for our approach to decision-making under uncertainty