# 4. Boosting the efficiency of inverse transform

## Overview

In this lecture, we discuss how to boost the efficiency of the inverse search algorithm

Learning Objective:

- Able to improve the efficiency based on ordered PMF
- Understand how to implement derive conversion for a known CDF.

## [1] Finding a better search sequence

Let's evaluate the efficiency of the previous algorithm we implemented. Here, the efficiency is evaluated based on the number of times a $u$ needs to be compared to different levels of $CDF$ in order to get converted to a sample value. As a $u$ is not fixed, we compute the expected value of the number of comparisons (searches).

It is easy to see that the number of searches needed follows a discrete distribution. The following table lists the number of searches needed and the probability of a certain number of searches happening. Based on this table, we can compute the expected number of searches needed for the algorithm.

**Strategy 1: Searching from the smallest variable outcome to the largest variable outcome**

| Outcome | Probability | CDF | Searches |
|---|---|---|---|
| 1 | 0.05 | 0.05 | 1 |
| 2 | 0.1 | 0.15 | 2 |
| 4 | 0.05 | 0.2 | 3 |
| 5 | 0.1 | 0.3 | 4 |
| 7 | 0.3 | 0.6 | 5 |
| 8 | 0.4 | 1 | 6 |

For example, if $u = 0.388$, we will first compare with 0.05, and then with 0.15, and then with 0.2, and then with 0.3, and finally with 0.4. The number of searches we did is 5 in order to determine that the sample value is 7.
The expected value of number of searches for a random $u$ is
$E(s) = \sum s_i p_i = 0.05 \times 1 + 0.1 \times 2 + 0.05 \times 3 + 0.1 \times 4 + 0.3 \times 5 + 0.4 \times 6 = 5.5$

**Strategy 2: Searching from the largest probability to the smallest probability**

Now, let's re-order the outcomes based on the probabilities from largest to the smallest.

| Outcome | Probability | CDF | Searches |
|---|---|---|---|
| 8 | 0.4 | 0.4 | 1 |
| 7 | 0.3 | 0.7 | 2 |
| 2 | 0.1 | 0.8 | 3 |
| 5 | 0.1 | 0.9 | 4 |
| 1 | 0.05 | 0.95 | 5 |
| 4 | 0.05 | 1 | 6 |

In this case, if we have $u = 0.38$, we will convert it to 8 with 1 search. If we have $u = 0.58$, we will convert it to 7 with 2 searches.

The expected value of the number of searches for a random $u$ is

$$E(s) = 1 \times 0.4 + 2 \times 0.3 + 3 \times 0.1 + 4 \times 0.1 + 5 \times 0.05 + 6 \times 0.05 = 2.25$$

Thus, searching the outcomes based on probability from largest to smallest helps us improve the efficiency.

## [1] Python Implementation

| Outcome | Probability | CDF | Searches |
|---|---|---|---|
| 1 | 0.05 | 0.05 | 1 |
| 2 | 0.1 | 0.15 | 2 |
| 4 | 0.05 | 0.2 | 3 |
| 5 | 0.1 | 0.3 | 4 |
| 7 | 0.3 | 0.6 | 5 |
| 8 | 0.4 | 1 | 6 |

- Search following the table sequence
- Resort the outcomes and the probabilities. Search based on the new sequence

To sort the outcomes based on the probabilities following a descending order, we can use `np.sort()` statement and `np.argsort()` statement.

| Function | Description |
|---|---|
| np.sort(A) | sort the elements in A from smallest to the largest |
| np.sort(A)[::-1] | sort the elements in A from largest to smallest |
| np.argsort(A) | sort the elements in A from smallest to the largest but record their indices instead |
| np.argsort(A)[::-1] | sort the elements in A from the largest to the smallest but record their indice instead |

Run PYTHON

```python
import numpy as np

Array1=np.array([5,4,3,2,9])
#sort Array1 from the smallest to the largest
print(np.sort(Array1))

#sort Array 1 from the smallest to the largest and record their indices
#3 in the array means element the largest value in Array1
print(np.argsort(Array1))

#Using these indices, we can recover their values
indices=np.argsort(Array1)
#the indices, we will put the 4th element first, 3rd element second, ...
#Thus, reording the sequence
```

## [2] Sampling from a Poisson distribution

The PMF of a Poisson distribution is as follows:

$P(x) = \frac{exp(-\lambda)\lambda^x}{x!}$

Again, our target is to find the CDF corresponding to each $x$. Then the conversion will be

- if $u < CDF(x = 0)$, then $x = 0$
- else if $u < CDF(x = 1)$, then $x = 1$
- else if $u < CDF(x = 2)$, then $x = 2$
- ...

Unfortunately, there is no closed-form solution for the CDF of a Poisson distribution. Thus, we will use PMF to construct the CDF.

- $CDF(x = 0) = PMF(x = 0)$
- $CDF(x = 1) = CDF(x = 0) + PMF(x = 1)$
- $CDF(x = 2) = CDF(x = 1) + PMF(x = 2)$
- ...

Regarding the Python programming part, we do not want to write conditional statements or for loop, because there will be infinite number comparison statements ($x$ could take 0 to positive infinitely).

Thus, we would like to think if we can convert it to a while loop.

- Compute $CDF(x = 0) = PMF(x = 0)$
- While $u$ is larger than or equal to $CDF(x)$
  - update $x$ to $x + 1$
  - Compute $CDF(x + 1)$ using $CDF(x) + PMF(x + 1)$

In addition, since the possible conversion goes high as $\infty$. We need to use a `while` loop.

### `while` loop

Besides `for` loop, another way to write a loop is to use `while` statement. The Syntex of a while loop is as follows:

```
while (condition):
   operation
```

Every single time, we check whether the condition is satisfied. If the condition is not satisfied anymore, we will stop the loop. In this case, instead of explicitly defining the number of loops, we will stop the loop when the condition is not satisfied anymore.

`while` loop is especially useful when we only know the stopping condition but do not know how many loops are needed.
In the following code, we calculate how many times we need to divide $b = 1$ by 1.01 until $b < 0.07$.

Run PYTHON

```python
b=1
i=0
while (b>=0.07):
  b=b/1.01
  i=i+1
print(i)
```

Group discussion: here

## [2] Improve the efficiency of Poisson sampling

First, let's observe the distribution of a Poisson distribution.

From observing the distribution of this distribution, we can see that a more efficient way is to start from somewhere around $\lambda$. After comparing $u$ with the $CDF$ at the new starting point, we can then decide to search upward or downward based on whether $u \geq CDF$ or not.

Indeed, a good option is to start with $CDF(x = \lfloor\lambda\rfloor)$.

This will give a higher efficiency due to two reasons:

(1) Reason 1
If $u \geq CDF(x = \lfloor\lambda\rfloor)$, we can directly eliminate $x \leq \lfloor\lambda\rfloor$
if $u < CDF(x = \lfloor\lambda\rfloor)$, we can directly eliminate $x > \lfloor\lambda\rfloor$
However, if we search from x=0, we can only eliminate the $x$ values one by one.

(2) Reason 2
The reason is similar to the reason in the previous case. When we search downwards, we search the $x$ values based on their probabilities from the largest to the smallest. However, if we use a pure upward search algorithm, those cases will be searches based on a sequence from the smallest

probability to the largest probability.

**General idea of this algorithm:** Conditional on that, we might decide to either search upward or downward conditional on the relationship between $U$ and the $CDF(x = \lfloor\lambda\rfloor)$. If $U \geq CDF(\lfloor\lambda\rfloor)$, we will search upward, until U<CDF. However, if $U < CDF(\lfloor\lambda\rfloor)$, we will search download until $U \geq CDF$.

**The detailed algorithm is given as follows:**

- Step 1: Compute the starting point of the search: $start = \lfloor\lambda\rfloor$.
- Step 2: Computing the $CDF(x = start)$ using the iterative method.
- Step 3: Generate $u$.
- Step 4: Compare $u$ with $CDF(x = start)$
    - If $u \geq CDF(x = start)$
        - Compare with $u$ with $CDF(x = start + 1)$, $CDF(x = start + 2)$, .... until $u < CDF(x_k)$. $x_k$ will be the sample we draw from the Poisson distribution.
    - else if $u < CDF(x = start)$
        - Compare with $u$ with $CDF(x = start - 1)$, $CDF(x = start - 2)$, .... until $u \geq CDF(x_k)$. $x_k + 1$ will be the sample we draw from the Poisson distribution.

## [2] Python Implementation of Case 2

*This code slide does not have a description*

## [2] Measuring the Efficiency of Refined Poisson

## Upward Search

For upward search, the number of searches we need is $1 + E(x) = 1 + \lambda$

To see why this is the case, let's list the outcomes, their corresponding probabilities, and the number of searches.

| **Outcomes** | **Number of Searches** | **Probabilities** |
|---|---|---|
| $0$ | $1$ | $p(0)$ |
| $1$ | $2$ | $p(1)$ |
| $2$ | $3$ | $p(2)$ |
| $\dots$ | $\dots$ | $\dots$ |
| $n$ | $n+1$ | $p(n)$ |
| $\dots$ | $\dots$ | $\dots$ |

As a result, $E(searches) = \sum_{n=0}^{\infty}(n+1)P(n) = \sum_{n=0}^{\infty} nP(n) + \sum_{n=0}^{\infty} P(n) = \lambda + 1$

In the last step, notice that $\sum_{n=0}^{\infty} nP(n)$ is computing the expected value of a Poisson distribution, while $\sum_{n=0}^{\infty} P(n)$ is computing the $CDF(x = \infty)$ of the Poisson distribution.

## Upward/Downward Search

| **Outcomes** | **Number of Searches** | **Probabilities** |
|---|---|---|
| $\dots$ | $\dots$ | $\dots$ |
| $\lfloor\lambda\rfloor - 2$ | $4$ | $P(x = \lfloor\lambda\rfloor - 2)$ |
| $\lfloor\lambda\rfloor - 1$ | $3$ | $P(x = \lfloor\lambda\rfloor - 1)$ |
| $\lfloor\lambda\rfloor$ | $2$ | $P(x = \lfloor\lambda\rfloor)$ |
| $\lfloor\lambda\rfloor + 1$ | $2$ | $P(x = \lfloor\lambda\rfloor + 1)$ |
| $\lfloor\lambda\rfloor + 2$ | $3$ | $P(x = \lfloor\lambda\rfloor + 2)$ |
| $\lfloor\lambda\rfloor + 3$ | $4$ | $P(x = \lfloor\lambda\rfloor + 3)$ |
| $\dots$ | $\dots$ | $\dots$ |

When summing up the number of searches times the probability, we can get the expected number of searches.

To derive the closed-form solution, notice that

- when x= 0,1,2, 3,..., $\lfloor\lambda\rfloor$
  - number of searches $s = \lfloor\lambda\rfloor + 2 - x$

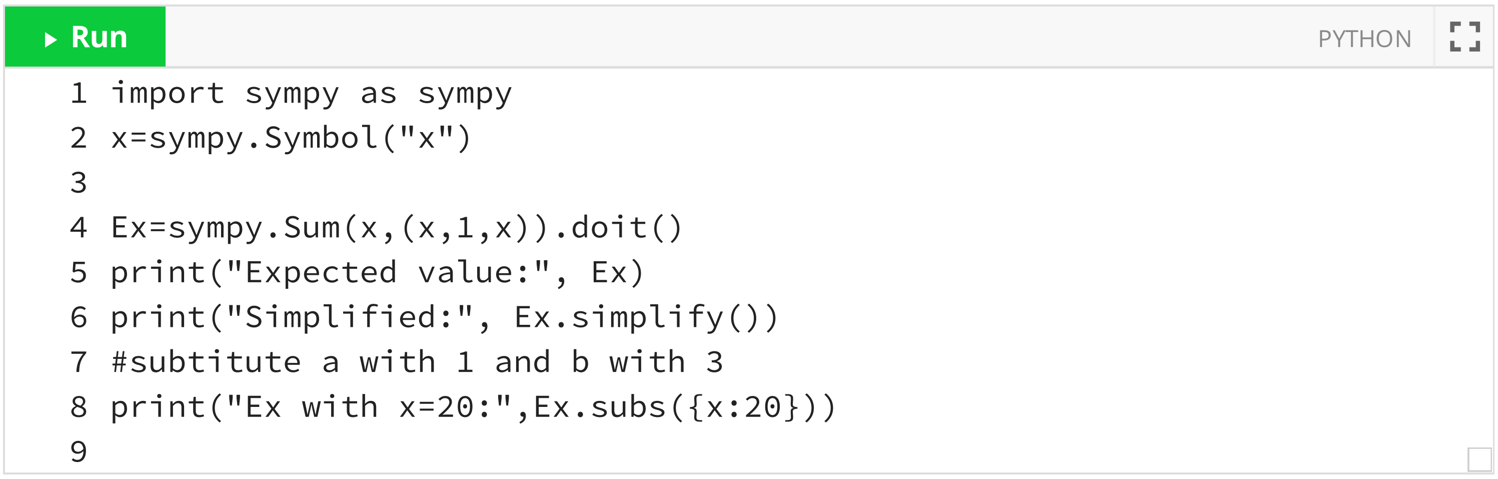
- when x=$\lfloor\lambda\rfloor+1$,$\lfloor\lambda\rfloor+2$,...
  - number of searches $s=x-\lfloor\lambda\rfloor+1$

Thus, the expected number of searches is equal to

$\sum_{x=0}^{\lfloor\lambda\rfloor} P(x)(\lfloor\lambda\rfloor+2-x)+\sum_{x=\lfloor\lambda\rfloor+1}^{\infty} P(x)(x-\lfloor\lambda\rfloor+1)$

We can also use `sympy` package (link) to help us compute integrals this expected value. `sympy` package has many functions that allow us to perform symbolic operations.

- Declare `sympy` symbols. For a uniform distribution, we have $x$ as our variable.
  - `x=sympy.Symbol("x",assumptions)`
  - We will later use x as a `Symbol` object. For the `sympy` operation output, x will be displayed as x.
  - For a uniform distribution, we can define $a$, $b$, $x$.
  - a list of assumptions can be found here
    - The assumptions are important variables that are constrained. For example, for a normal distribution, the standard deviation can only be positive.
- Construct the function that needs to be integrated.
  - `sympy.exp(x)`
  - `sympy.log(x)`
  - `sympy.abs(x)`
  - `sympy.pi`
  - `sympy.factorial(x)`
  - ...
- use `sympy.Sum(f,(x,lowerbound,upperbound)).doit()` to integrate.
  - If the lower bound is $-\infty$, please use `-sympy.oo`
  - If the upper bound is $\infty$, please use `sympy.oo`
- `simplify()` can be used to simplify an expression
  - simplify an expression. It tries all the major simplification operations in `SymPy`, and uses heuristics to determine the simplest result.
- After this, we can use `f.subs({x:val1;y:val2})` to replace the `sympy` `Symbols` with actual values to evaluate the results.
- In the end, we can use `sympy.N()` to evaluate the numerical results.



```python
import sympy as sympy
x=sympy.Symbol("x")

Ex=sympy.Sum(x,(x,1,x)).doit()
print("Expected value:", Ex)
print("Simplified:", Ex.simplify())
#subtitute a with 1 and b with 3
print("Ex with x=20:",Ex.subs({x:20}))

```

## [3] Direct conversion (Geometric)

Assume that $x$ is our variable. The outcomes of $x$ are all integers. We might be able to use the following theorem to directly convert $u$ to $x$.

**Theorem: Assume that m is a real number and x is an integer, then $\lfloor m \rfloor = x$, if and only if $m - 1 < x \leq m$.**

For example, assume that 4.9<x<=5.9, we know that x has to be 5, which is $\lfloor 5.9 \rfloor = 5$.

Thus, our goal is to bound $x$ between $m-1$ and $m$. Let's use the following example to see how this can be done.

**Example 1: Geometric distribution**

One definition of geometric distribution measures the number of indepedent trials needed **before** getting the first success. The number of trials ($x$) needed has the following PMF

$PMF(x) = (1-p)^x p$, $x = 0, 1, 2, \ldots \infty$

where $p$ is the probability of success.

It can be derived that $CDF(x) = 1 - (1-p)^{(x+1)}$

For a geometric distribution, we know that the following rule is true: if $x$ corresponds to $u$, then $CDF(x-1) \leq u < CDF(x)$. From this relationship, let's try to derive $m - 1 < x \leq m$.

Let's first solve $1 - (1-p)^x \leq u$

$$\begin{aligned}
&1 - (1-p)^x \leq u \\
&\Rightarrow (1-p)^x \geq 1 - u \\
&\Rightarrow x ln(1-p) \geq ln(1-u) \\
&\Rightarrow x \leq \frac{ln(1-u)}{ln(1-p)}
\end{aligned}$$

Let's also solve $u < 1 - (1-p)^{(x+1)}$

$$\begin{aligned}
&u < 1 - (1-p)^{(x+1)} \\
&\Rightarrow (1-p)^{(x+1)} < 1 - u \\
&\Rightarrow (x+1) ln(1-p) < ln(1-u) \\
&\Rightarrow x > \frac{ln(1-u)}{ln(1-p)} - 1
\end{aligned}$$

Thus, we get $\frac{ln(1-u)}{ln(1-p)} - 1 < x \leq \frac{ln(1-u)}{ln(1-p)}$.

Applying the theorem we presented at the beginning, we know that

$x = \lfloor \frac{ln(1-u)}{ln(1-p)} \rfloor$

In addition, since both $1 - u$ and $u$ follow unif(0,1), we can further simplify the formula to $x = \lfloor \frac{ln(u)}{ln(1-p)} \rfloor$.

## [4] Direct Conversion (Uniform Discrete)

If $x$ follows a uniform discrete distribution between $a$ and $b$, derive the formula to help us directly convert $u$ to a sample of $x$.

## [4] Solution for Case 4

(1) Solve CDF

```python
import sympy
a=sympy.Symbol("a")
b=sympy.Symbol("b")
x=sympy.Symbol("x")
PMF=1/(b-a+1)
CDF=sympy.Sum(PMF,(x,a,x)).doit().simplify()
print(CDF)
```

We can get that the CDF is equal to

$CDF(x) = \frac{x+1-a}{b+1-a}$

(2) Solve $x$

we can get that $x$ will be gotten from $CDF(x-1) \leq u < CDF(x)$

$\frac{x-a}{b+1-a} \leq u < \frac{x+1-a}{b+1-a}$

From $\frac{x-a}{b+1-a} \leq u$, we get $x \leq a + (b+1-a)u$

From $u < \frac{x+1-a}{b+1-a}$, we get $x > a + (b+1-a)u - 1$

Thus, we can get $a + (b+1-a)u - 1 < x \leq a + (b+1-a)u$

Thus, $x = \lfloor a + (b+1-a)u \rfloor$

For the last step, we can also use use `sympy.solve(function, symbol...)` function to solve $x$. Here, the function is F(x)-u, while the symbol will be x. It will help us solve $x$ as a function of $u$

Run PYTHON

```python
import sympy
a=sympy.Symbol("a")
b=sympy.Symbol("b")
x=sympy.Symbol("x")
u=sympy.Symbol("u")
PMF=1/(b-a+1)
CDF=sympy.Sum(PMF,(x,a,x)).doit().simplify()
#solve u<CDF(x)
S1=sympy.solve(CDF-u,x)

print(S1)
S1=S1[0].collect(u)

#solve CDF(x)<=u
```

We know from the last step that $x$ must follow

$$a + u * (-a + b + 1) - 1 < x \leq a + u * (-a + b + 1)$$

Thus, $x = \lfloor a + (b + 1 - a)u \rfloor$