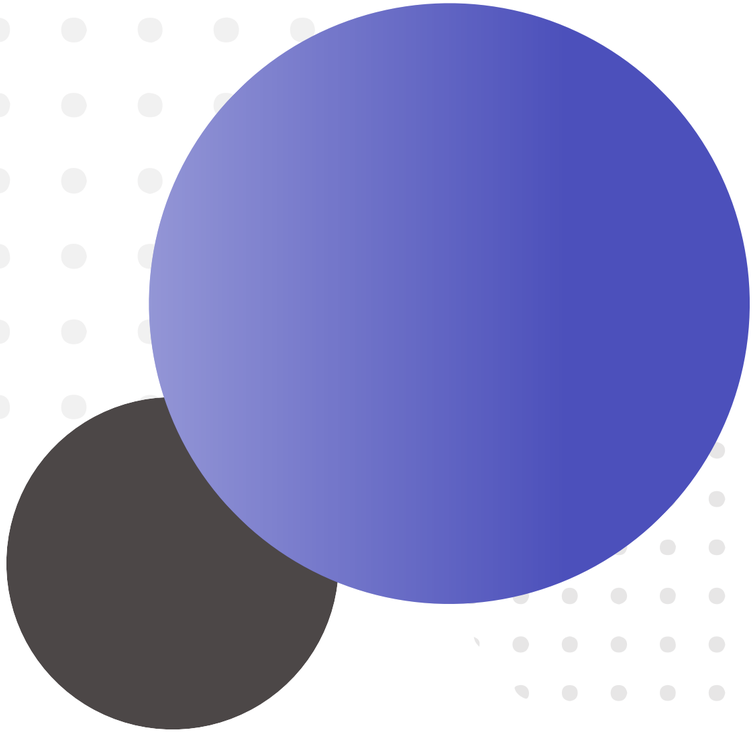
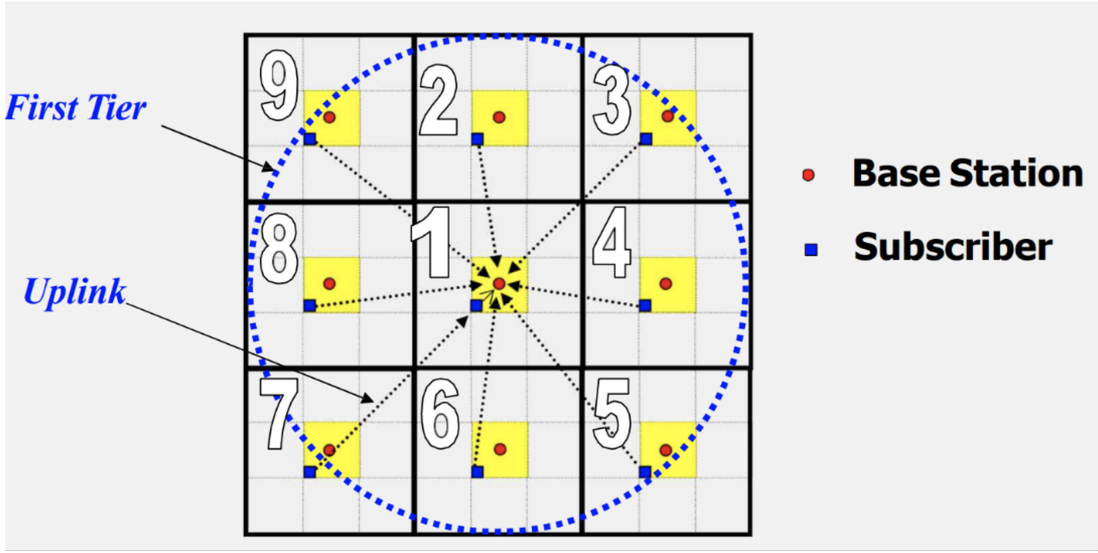


1-Tier TDD Network 분석

소프트웨어전공 20215209 윤미나

1-Tier TDD Network

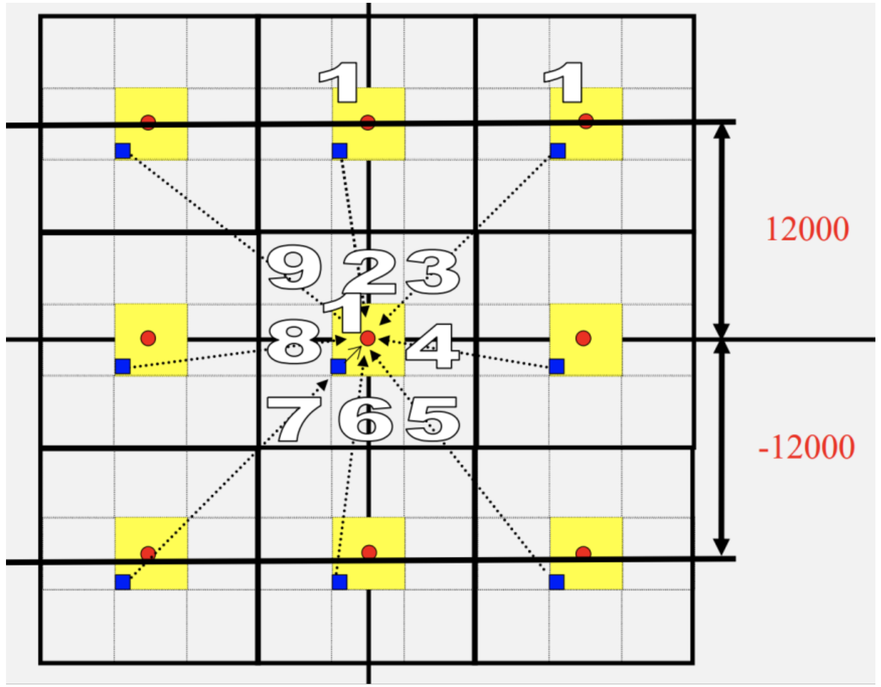


Frequency Reuse를 통해 여러 개의 base station과 user가 동일한 주파수 대역을 공유하면서도
서로 간섭을 최소화하고 효율적인 통신을 가능하게 하는 방식이다.

python 구현

1. 변수 설정

```
In [4]: import numpy as np
   ...: import matplotlib.pyplot as plt
   ...:
In [5]: FreqReUse=9
   ...: NoUpLink=12
   ...: NtwSizeA=-2000
   ...: NtwSizeB=2000
   ...: PlusShift=12000
   ...: MinusShift=-12000
   ...: No_Iterations=1000
   ...:
In [6]: SIR=np.zeros((1,NoUpLink))
```



python 구현

2. User Generation

```
In [6]: SubX = np.random.uniform(NtwSizeA, NtwSizeB, size=[FreqReUse, NoUpLink])
   ...: SubY = np.random.uniform(NtwSizeA, NtwSizeB, size=[FreqReUse, NoUpLink])
   ...:
   ...: Cell_x0 = SubX[0, :]
   ...: Cell_y0 = SubY[0, :]
   ...: Cell_x1 = SubX[1, :]
   ...: Cell_y1 = SubY[1, :] + PlusShift
   ...: Cell_x2 = SubX[2, :] + PlusShift
   ...: Cell_y2 = SubY[2, :] + PlusShift
   ...: Cell_x3 = SubX[3, :] + PlusShift
   ...: Cell_y3 = SubY[3, :]
   ...: Cell_x4 = SubX[4, :] + PlusShift
   ...: Cell_y4 = SubY[4, :] + MinusShift
   ...: Cell_x5 = SubX[5, :]
   ...: Cell_y5 = SubY[5, :] + MinusShift
   ...: Cell_x6 = SubX[6, :] + MinusShift
   ...: Cell_y6 = SubY[6, :] + MinusShift
   ...: Cell_x7 = SubX[7, :] + MinusShift
   ...: Cell_y7 = SubY[7, :]
   ...: Cell_x8 = SubX[8, :] + MinusShift
   ...: Cell_y8 = SubY[8, :] + PlusShift
   ...:
```

```
In [9]: ShiftX = np.array([Cell_x0, Cell_x1, Cell_x2, Cell_x3, Cell_x4, Cell_x5,
   ...:                    Cell_x6, Cell_x7, Cell_x8])
   ...: ShiftY = np.array([Cell_y0, Cell_y1, Cell_y2, Cell_y3, Cell_y4, Cell_y5,
   ...:                    Cell_y6, Cell_y7, Cell_y8])
```

```
In [12]: Dist = np.sqrt(ShiftX**2+ ShiftY**2)
In [13]: Dist
Out[13]:
array([[ 1168.23468481,  1063.86087185,  1155.85264041,  1375.00437455,
         1492.55721178,  2089.41699025,  1184.26887155,   568.80903784,
          755.31938739,  1902.68437076,  1564.36401654,  1867.59761053],
       [13757.00645788, 11895.19226351, 11692.06619456, 13846.81385168,
        12320.56630691, 14003.47299813, 12346.49563364, 11117.88249589,
```

python 구현

3. Power Generation

```
In [14]: NormalDistribution = np.random.randn(FreqReUse, NoUpLink)
    ...: mu = 0
    ...: SD = 8
    ...: LogNormal=mu+SD*NormalDistribution
    ...:
In [15]: 10**(LogNormal/10)
Out[15]:
array([[8.53385692e-01, 1.90419106e+01, 1.89775919e+00, 2.21342710e-02,
        1.94652497e+00, 5.30297402e-01, 2.87176993e-01, 6.96836367e-02,
        7.23459961e+00, 8.15145371e-01, 1.75967471e-01, 2.14588957e-01],
       [6.62584363e-01, 3.20820821e+00, 4.46909181e+00, 1.92940733e+00,
        1.92113199e-01, 1.27900558e+01, 1.72655417e-01, 8.54825161e-01,
        1.82603194e-02, 7.81856115e+00, 7.91250018e+00, 2.10908323e-02],
       [2.94256938e+00, 1.59189691e-01, 1.73537835e-01, 2.50361928e+00,
        1.87875632e+01, 1.19383057e+01, 2.65941914e-01, 4.52576947e-01,
```

```
In [16]: LogNormalP=10**(LogNormal/10)/(Dist**4)
In [17]: LogNormalP
Out[17]:
array([[4.58168231e-13, 1.48652144e-11, 1.06323980e-12, 6.19225601e-15,
        3.92225685e-13, 2.78239646e-14, 1.45998498e-13, 6.65678997e-13,
        2.22275692e-11, 6.21968032e-14, 2.93819842e-14, 1.76390069e-14],
       [1.84988793e-17, 1.60242120e-16, 2.39141096e-16, 5.24837078e-17,
        8.33747458e-18, 3.32605484e-16, 7.43028342e-18, 5.59485683e-17,
        1.24355898e-18, 5.60174731e-16, 2.17690775e-16, 1.44456932e-18],
       [3.73536692e-17, 1.39103102e-18, 1.74122237e-18, 3.71635234e-17,
        1.52155336e-16, 1.39897898e-16, 3.97367674e-18, 6.89200542e-18,
        5.47295041e-17, 6.89924700e-16, 1.38210017e-17, 3.12953397e-17],
```

if) SD = 8, -8 ~ 8사이에 68%가 분포되있다.

python 구현

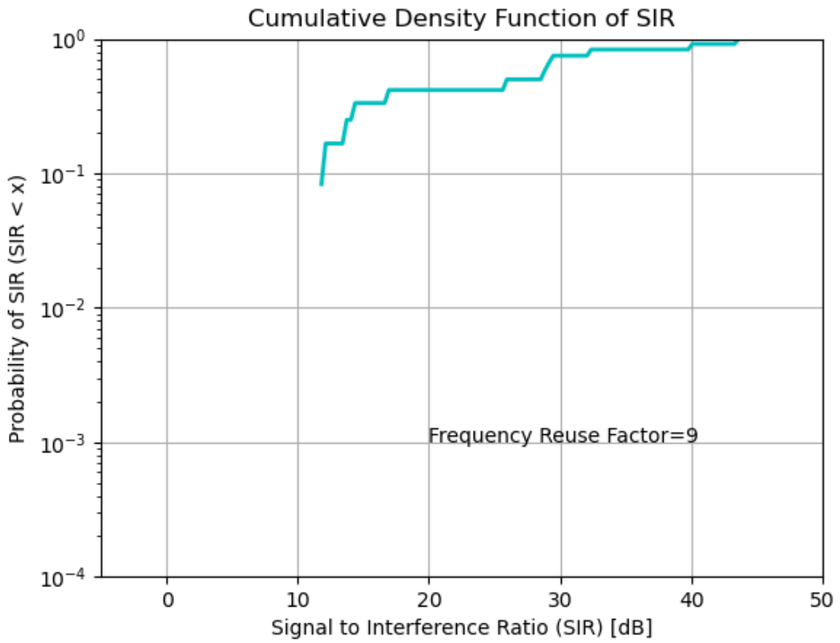
4. SIR 계산

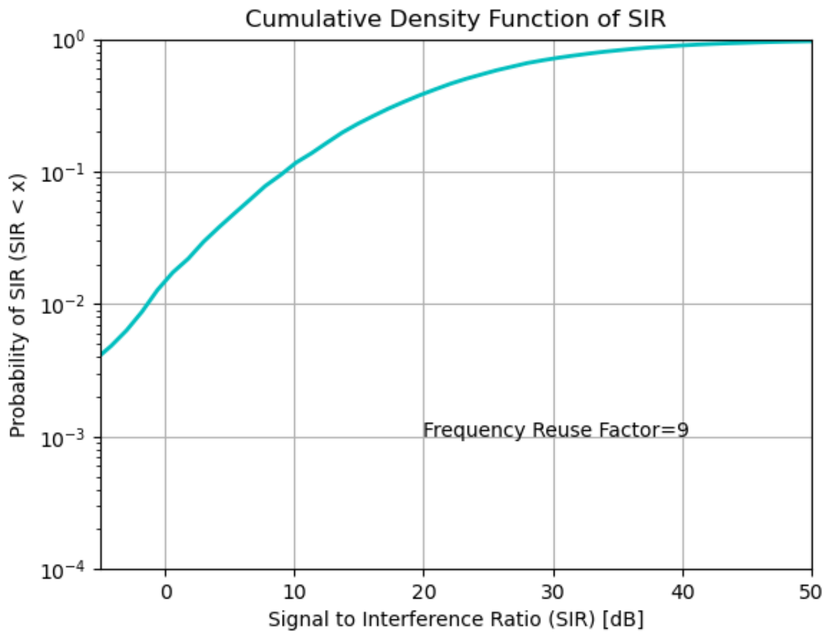
```
In [18]: PS = LogNormalP[0, :]
    ...: PI1 = LogNormalP[1, :]
    ...: PI2 = LogNormalP[2, :]
    ...: PI3 = LogNormalP[3, :]
    ...: PI4 = LogNormalP[4, :]
    ...: PI5 = LogNormalP[5, :]
    ...: PI6 = LogNormalP[6, :]
    ...: PI7 = LogNormalP[7, :]
    ...: PI8 = LogNormalP[8, :]
    ...:
    ...: PI = PI1+PI2+PI3+PI4+PI5+PI6+PI7+PI8
    ...: SIRn = PS/PI
    ...: SIRdB = 10*np.log10(SIRn)
    ...:
In [19]: SIRdB
Out[19]:
array([29.42181998, 43.93558817, 32.59436682, 13.98403261, 29.65806418,
       17.01068283, 26.00666349, 28.91423216, 40.09260953, 14.5683309 ,
       11.81218982, 12.40859799])

In [22]: SIR = np.vstack((SIR, SIRdB))
    ...:
    ...: SIR=np.delete(SIR, 0,0)
    ...: SIR=SIR.flatten()
```

SIR = Wanted Signal / Unwanted Signal
= Signal / Interfernce
= S / I

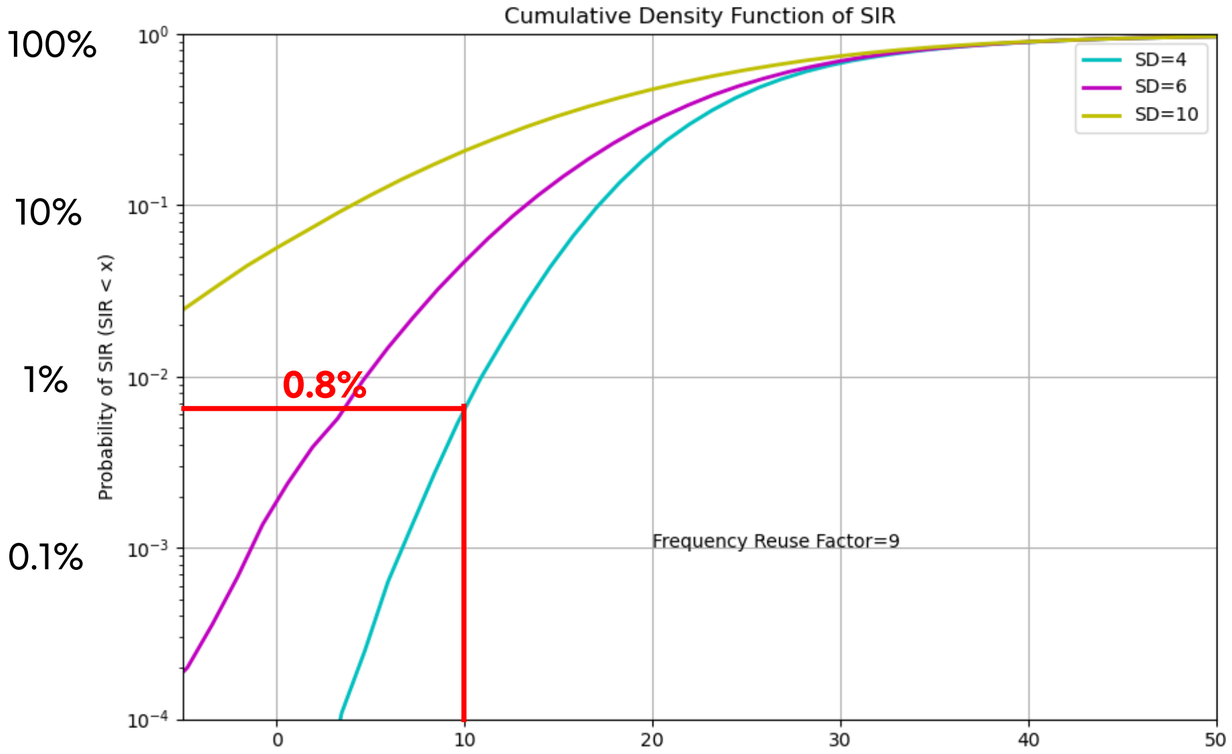
주위 간섭(I)이 적을수록 성능이 좋다.
-> SIR의 값이 클수록 성능이 좋다.

SIR 분석 (Monte Carlo simulation)





Power SD = 4



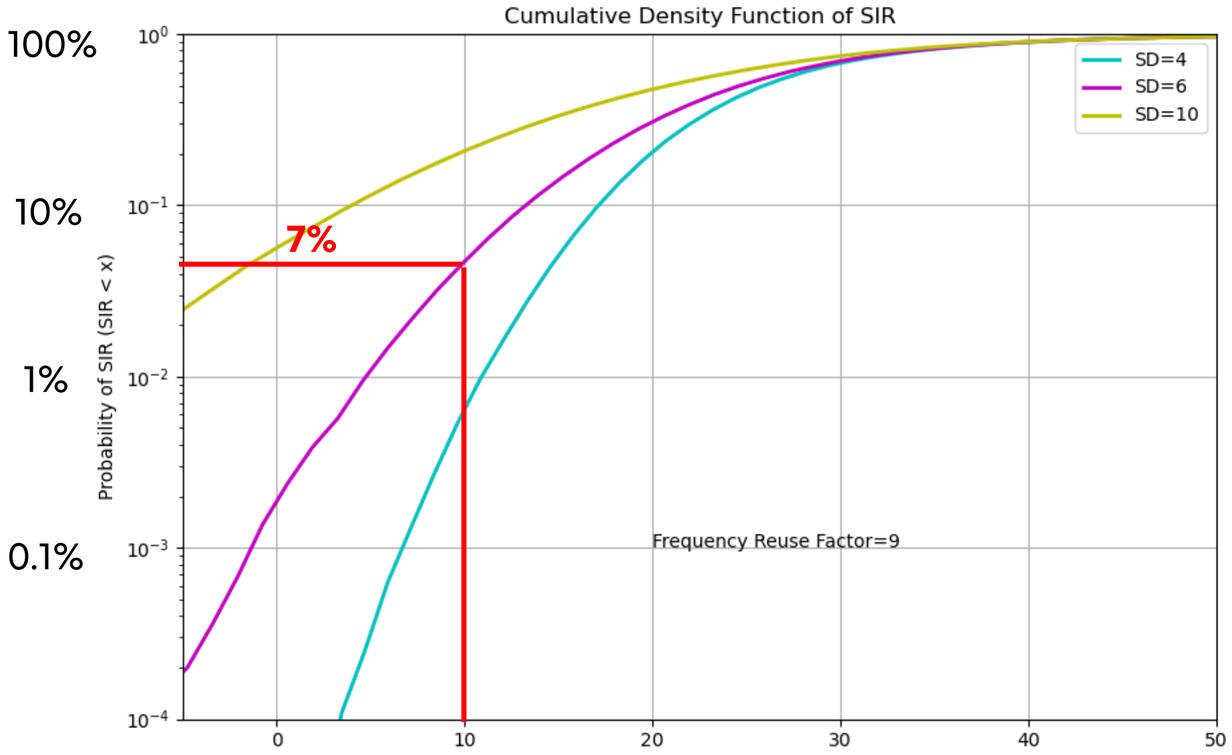
1. SD = 4 예시 상황

농촌 지역 또는 방안과 같은 상황
(건물이나 다른 장애물이 적은 곳)
-> 간섭이 적은 환경

2. 일정 SIR를 기준으로 성능을 측정해보면,

SIR = 10 일 확률 -> 0.8%
SIR > 10 일 확률 -> 99.2%

Power SD = 6



1. SD = 6 예시 상황

건물, 차량 등이 있는 일반적인 도시 환경

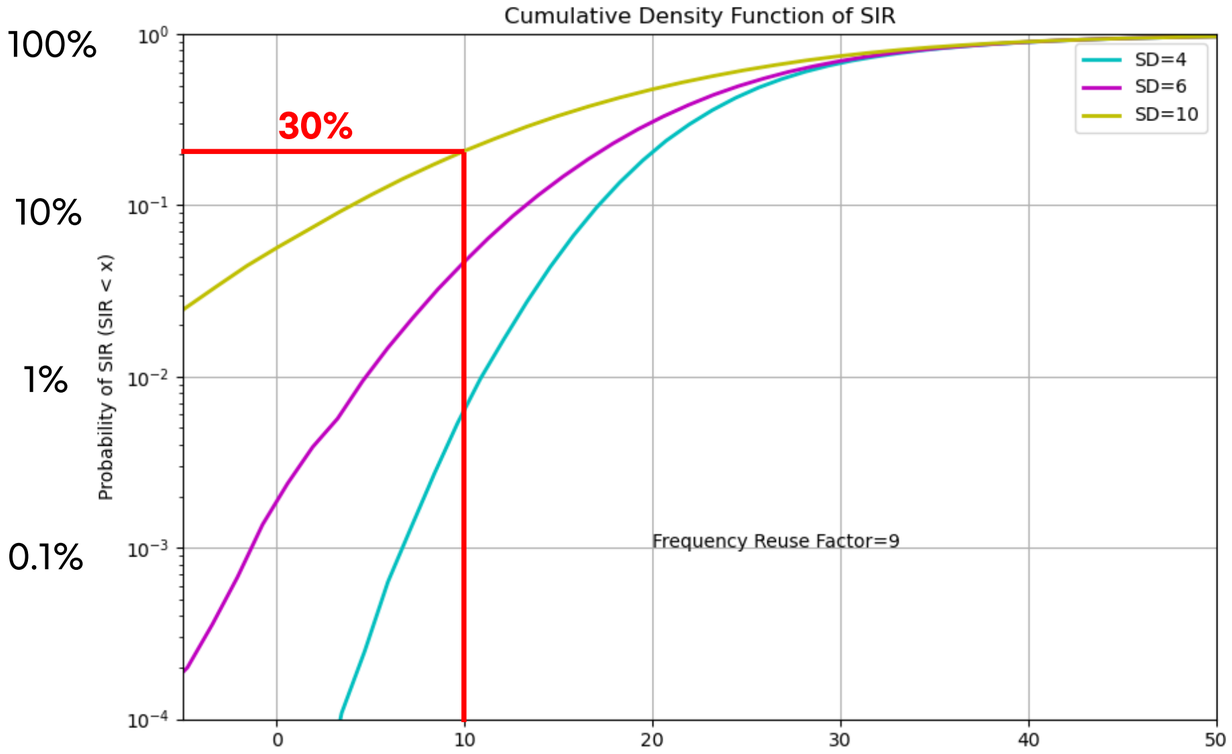
-> 간섭이 보통인 환경

2. 일정 SIR를 기준으로 성능을 측정해보면,

SIR = 10 일 확률 -> 7%

SIR > 10 일 확률 -> 93%

Power SD = 10



1. SD = 10 예시 상황

고층 건물이 밀집된 도심지역이나 복잡한 지형의 환경

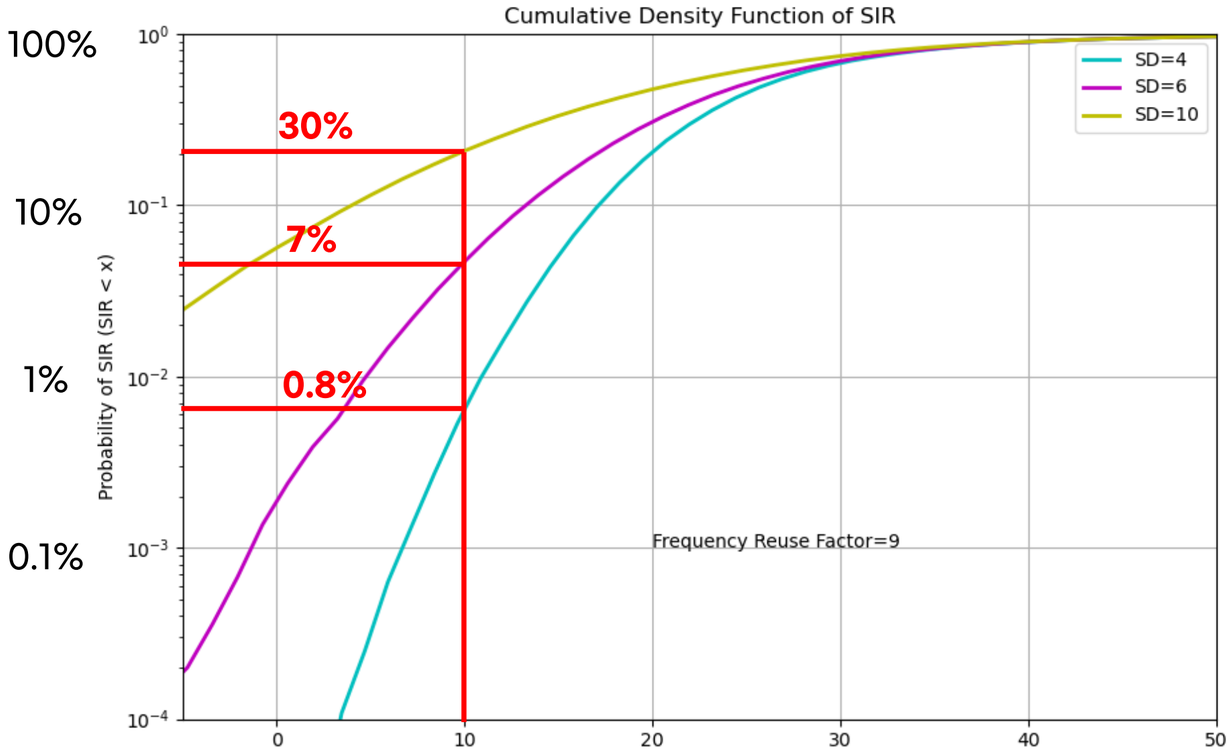
-> 간섭이 많은 환경

2. 일정 SIR를 기준으로 성능을 측정해보면,

SIR = 10 일 확률 -> 30%

SIR > 10 일 확률 -> 70%

SIR 성능 비교



SIR = 10

1. SD=4 -> 대략 0.8%
2. SD=6 -> 대략 7%
3. SD=10 -> 대략 30%

SIR > 10

1. SD=4 -> 대략 **99.2%**
2. SD=6 -> 대략 **93%**
3. SD=10 -> 대략 **70%**

SD = 4 > SD = 6 > SD = 10

SD의 값이 작을수록,
SIR 성능이 좋아진다.

THANK YOU