Model

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["font.family"] = "IBM Plex Serif"
```

Using the functions from Waks et al_2002_Security aspects of quantum key distribution with sub-Poisson light. The paper investigates quantum key distribution with sub-Poisson light sources. For the model constructed in this notebook equations 5-9.

The disturbance measure inroduces p_{err} and p_d which are the probabilities that the pulse causes an error and the probability of deceting a multiphoton event respectively. The probability that a pulse results in a click at the detector is given by p_{click}. The disturbance error reads as

$$\epsilon = \frac{p_{err} + p_d \, / \, 2}{p_{click}}.$$

In [ ]:
```
def error_rate(pol_err, multi_err, prob_click):
    """
    error rate
    """
    return (pol_err + (multi_err/2))/prob_click
```

In the limit where there are negligable errors from multiphoton events, the disturbance measure can be assumed to just be the bit error rate of transmission. Now $\epsilon = e$. An additional parameter β defines the fraction of events that originated from a the tranmission of a single photon. If p_m is the probability that the photon source emitted more than one photon the parameter is given by

$$\beta = \frac{p_{click} - p_m}{p_{click}}.$$

To account for Eve's attacks a compression function is defined which is a function of the β parameter and the error rate

$$\tau(e) = -\log_2[\frac{1}{2} + 2\frac{e}{\beta} - 2(\frac{e}{\beta})^2]$$

In [ ]:
```
def BETA(prob_click, multi_err):
    """
    """
    return (prob_click - multi_err)/prob_click

def compression_function(pol_err, multi_err, prob_click):
    """
    compression funtion defined above also containing the function Beta
    """

    _error_rate = error_rate(pol_err, multi_err, prob_click)
    _beta = BETA(prob_click, multi_err)

    return -np.log(0.5 + 2*_error_rate/_beta - 2*(_error_rate/_beta)**2)
```

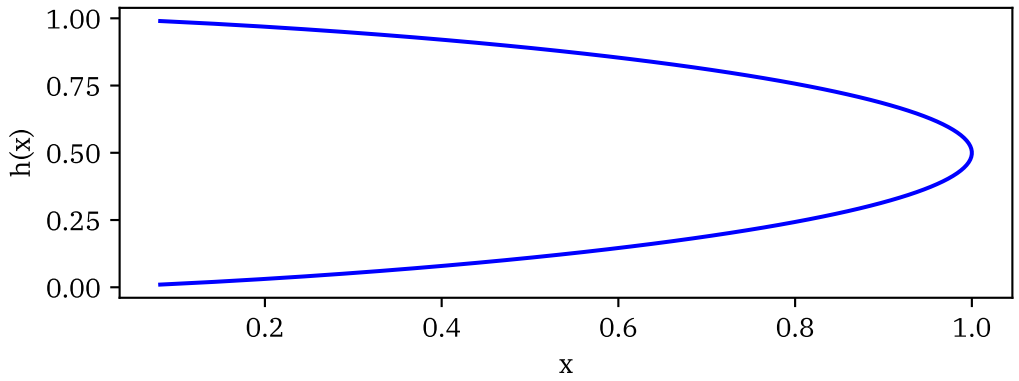
Finally the Shannon entropy is defined as

$$h(e) = -e\log_2 e - (1-e)\log_2(1-e)$$

In [ ]:
```
def binary_entropy(x):
    """
    calculate the binary entropy of x
    x: np.array containing values between 0 and 1
    """
    return -x*np.log2(x)-((1-x)*np.log2(1-x))


# plotting the binary entropy function
iter = np.linspace(0,1,100)
vals = [binary_entropy(i) for i in iter]
plt.figure(figsize=(6,2))
plt.plot(vals, iter, 'b')
plt.xlabel('x')
plt.ylabel('h(x)')
```

```
/var/folders/69/6w_l7snx2337833ct5yy0bn00000gn/T/ipykernel_63501/760511115.
py:6: RuntimeWarning: divide by zero encountered in log2
  return -x*np.log2(x)-((1-x)*np.log2(1-x))
/var/folders/69/6w_l7snx2337833ct5yy0bn00000gn/T/ipykernel_63501/760511115.
py:6: RuntimeWarning: invalid value encountered in double_scalars
  return -x*np.log2(x)-((1-x)*np.log2(1-x))
```

Out[ ]:
```
Text(0, 0.5, 'h(x)')
```



Evaluation of the above function with zero error rate results in a non-physical result due to the asymptotic and non-continuous nature of the function, see the figure above. This means that if the error is 0 or above 0.11, appropriate bounds should be made in the code. Finally, the key rate is given by the expression

$$R = \frac{p_{click}}{2}(\beta\tau(e) - f(e)h(e))\,.$$

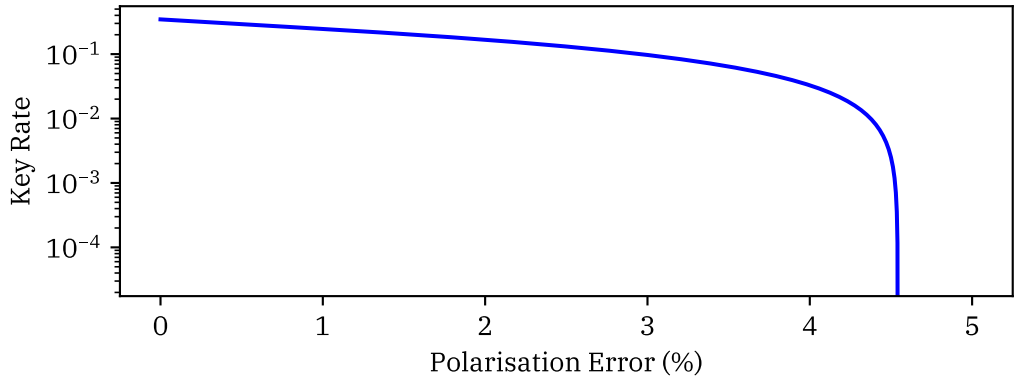
In [ ]:
```
def BB84rate(pol_err, multi_err, prob_click):
    """
    rate, neglecting photon statistics and assuming error correction at the
    """

    if pol_err == 0:
        pol_err = 0.0000000000000001

    _comp_func = compression_function(pol_err, multi_err, prob_click)
    _beta = BETA(prob_click, multi_err)

    return (prob_click/2) * _beta * _comp_func - binary_entropy(error_rate(p
```

Simulation

Taking the polarisation error rate to be anywhere between ~0% and 5%, assuming perfect single photons arriving at the detectors and further assuming perfect detection. The first figure is the key rate purely as a function of polarisation error. The second figure is the fraction of acheivable key rate when polarisation errors are present. So the result shows `key rate from errors / key rate without errors` .

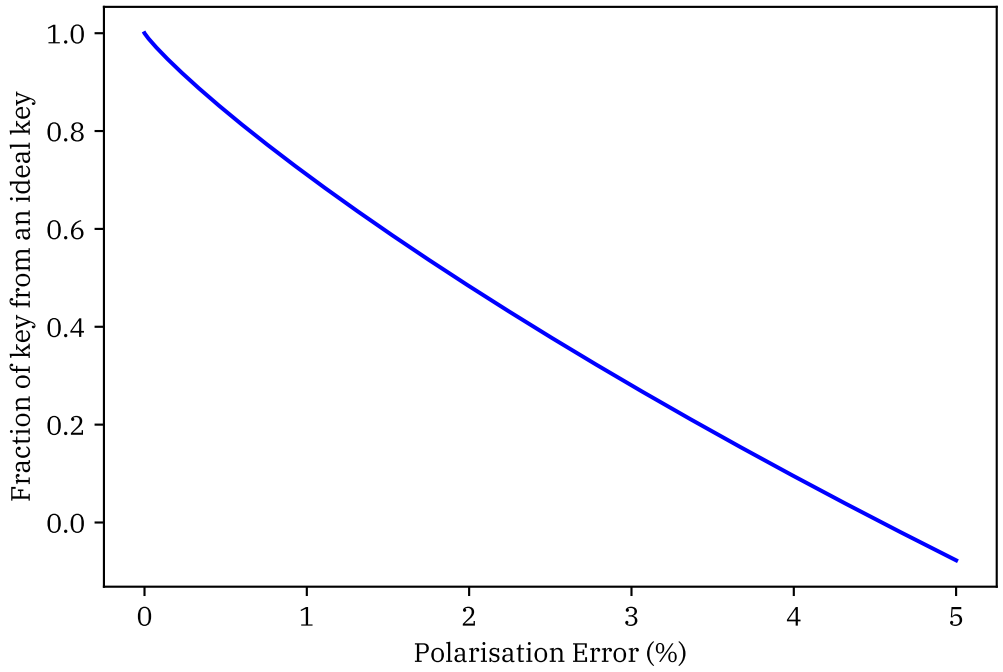
In [ ]:
```
multi_emission_error = 0
prob_click = 1

pol_error = np.linspace(5, 0, 10000)
rate = [BB84rate(it/100, multi_emission_error, prob_click) for it in pol_err

plt.figure(figsize=(6,2))
fig = plt.plot(pol_error, rate, 'b')

plt.yscale('log')
plt.xlabel('Polarisation Error (%)')
plt.ylabel('Key Rate')

# plt.savefig('boo.png', dpi=500, bbox_inches='tight')
```

Out[ ]:
```
Text(0, 0.5, 'Key Rate')
```



In [ ]:
```
ratio = [BB84rate(it/100, multi_emission_error, prob_click)/BB84rate(0, mult

plt.figure(figsize=(6,4))
fig = plt.plot(pol_error, ratio, 'b')

plt.xlabel('Polarisation Error (%)')
plt.ylabel('Fraction of key from an ideal key')

# plt.savefig('boo boo.png', dpi=500, bbox_inches='tight')
```

Out[ ]:
```
Text(0, 0.5, 'Fraction of key from an ideal key')
```



What does this mean wrt polarisation difference between s and p

Need to add a function that will convert `pol_error` into an actual value for the difference between s and p polarisation.

In [ ]:
```
# error is between 0 and 5%
pol_error = np.linspace(5, 0, 10000)
```

What does this mean for the states being prepared. Well, the `pol_error` is effectively the average error across each of the four basis states you perform protocol with. So assuming you use the Z and X bases, then what does the average error mean for the individual states being prepared. Well let's assume an error of 2% between a and p for an example. The probability of measuring in the four different bases are:

$$\mathrm{p}(\backslash\mathrm{braH}) = 0\mathrm{p}(\backslash\mathrm{braV}) = 1 - 0.98\mathrm{p}(\backslash\mathrm{braD}) = \mathrm{p}(\backslash\mathrm{ketA}) = 1 - 0.99,$$

and the average error across the four bases is therefore

$$\mathrm{p}(\backslash\mathrm{braH}) + \mathrm{p}(\backslash\mathrm{braV}) + \mathrm{p}(\backslash\mathrm{braD}) + \mathrm{p}(\backslash\mathrm{braA})) \,/\, 4$$

which for 2% error between s and p is 1% error. So constructing a function that will take an error in polarisation between s and p and give you the average error between basis states.

In [ ]:
```
def measured_error_to_pol_error(_error):

    error_in_X = 0.5+0.5*(1 - _error)
    error_in_Z = 1 - _error

    return (1+(error_in_Z)+(2*(error_in_X)))/4
```

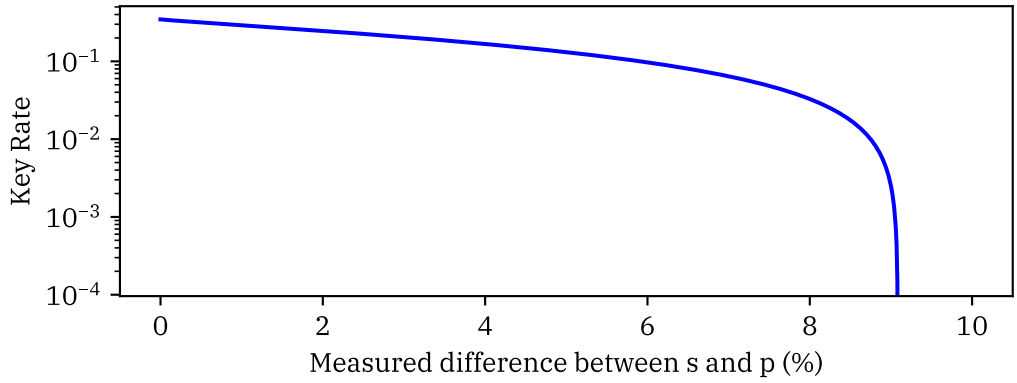
In [ ]:
```
measured_error_to_pol_error(0.05)
```

Out[ ]:
```
0.975
```

Now using the above function with the model results from earlier. Let's assume that the measured error between s and p polarisations is somewhere between 0 and 10%.

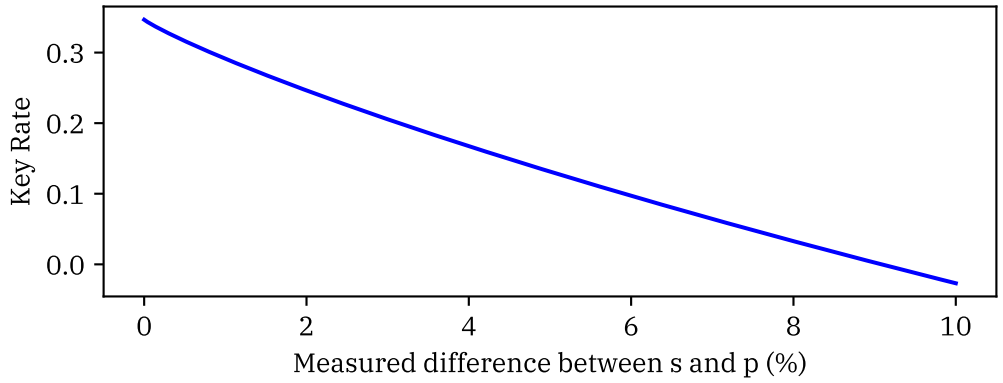
In [ ]:
```
measured_error = np.linspace(0, 10, 1000)

# PARAMS FOR MODEL
pol_error = 1 - measured_error_to_pol_error(measured_error/100)
multi_emission_error = 0
prob_click = 1

rate = [BB84rate(it, multi_emission_error, prob_click) for it in pol_error]

# plot

plt.figure(figsize=(6,2))
fig = plt.plot(measured_error, rate, 'b')

plt.yscale('log')
plt.xlabel('Measured difference between s and p (%)')
plt.ylabel('Key Rate')
```

Out[ ]:
```
Text(0, 0.5, 'Key Rate')
```



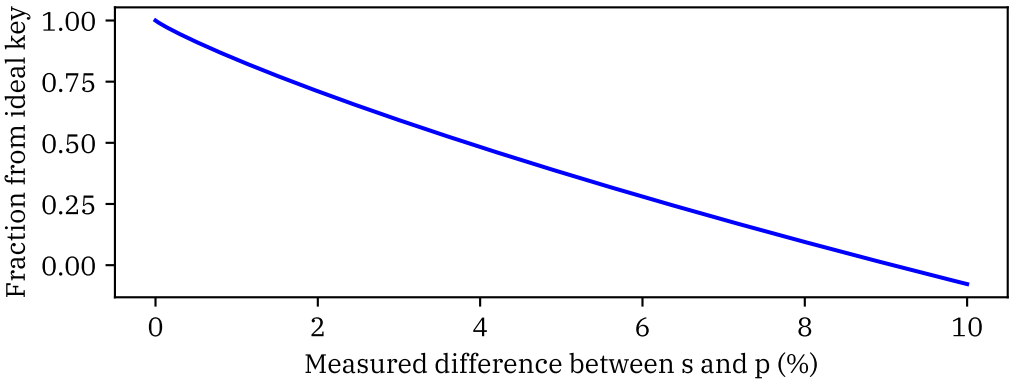
In [ ]:
```
plt.figure(figsize=(6,2))
fig = plt.plot(measured_error, rate, 'b')

plt.xlabel('Measured difference between s and p (%)')
plt.ylabel('Key Rate')
```

Out[ ]:
```
Text(0, 0.5, 'Key Rate')
```



In [ ]:
```
ratio = [BB84rate(it, multi_emission_error, prob_click)/BB84rate(0, multi_em

plt.figure(figsize=(6,2))
fig = plt.plot(measured_error, ratio, 'b')

plt.xlabel('Measured difference between s and p (%)')
plt.ylabel('Fraction from ideal key')

# plt.savefig('boo boo.png', dpi=500, bbox_inches='tight')
```

Out[ ]:
```
Text(0, 0.5, 'Fraction from ideal key')
```


Fraction from ideal key
1.00
0.75
0.50
0.25
0.00
0
2
4
6
8
10
Measured difference between s and p (%)