Calculation of the Secrecy Capacity

Release 0.1.0

Karl-Ludwig Besser

Nov 05, 2021

CONTENTS:

1 Installation **1**

2 Examples **3**
- 2.1 Loyka Algorithm . . . 3
- 2.2 Low Complexity Algorithm . . . 4
- 2.3 References . . . 4

3 secrecy_capacity Package **5**
- 3.1 Functions . . . 5

4 Indices and tables **9**

Python Module Index **11**

Index **13**

CHAPTER

ONE

INSTALLATION

You can install the package via pip

```
pip install secrecy-capacity
```

If you want to install the latest (unstable) version, you can install the package from source

```
git clone https://github.com/klb2/secrecy-capacity-calculation.git
cd secrecy-capacity-calculation
git checkout dev  # only if you want to install the unstable version
pip install .
```

You can test, if the installation was successful, by importing the package

```
import secrecy_capacity
print(secrecy_capacity.__version__)
```

CHAPTER

TWO

EXAMPLES

In the following, some simple usage examples are given.

You can also find example scripts in the `examples/` directory in the repository.

2.1 Loyka Algorithm

The first algorithm that you can use to calculate the optimal transmit covariance matrix to maximize the secrecy rate is the one from reference [*1*].

A very simple example is provided below. You only need to create the channel matrices to Bob and Eve and define your power constraint.

```
import numpy as np
from secrecy_capacity import cov_secrecy_capacity_loyka, secrecy_rate

# Random generation of 2x2 channels
channel_bob = np.random.randn(2, 2) + 1j*np.random.randn(2, 2)
channel_eve = np.random.randn(2, 2) + 1j*np.random.randn(2, 2)

power = 10  # power constraint (linear)

# Calculate the optimal transmit covariance matrix.
# This will take a while for the Loyka algorithm
opt_cov = cov_secrecy_capacity_loyka(channel_bob, channel_eve, power=power)

# If you want to calculate the secrecy capacity for the found covariance
# matrix, you can use the secrecy_rate function
sec_cap = secrecy_rate(channel_bob, channel_eve, cov=opt_cov)

print("Optimal covariance matrix:")
print(opt_cov)
print("Secrecy capacity: {:.3f}".format(sec_cap))
```

2.2 Low Complexity Algorithm

The second algorithm that you can use to calculate the optimal transmit covariance matrix to maximize the secrecy rate is the low-complexity one from reference [2].

The usage is very similar to the Loyka algorithm.

```
import numpy as np
from secrecy_capacity import cov_secrecy_capacity_low_complexity, secrecy_rate

# Random generation of 2x2 channels
channel_bob = np.random.randn(2, 2) + 1j*np.random.randn(2, 2)
channel_eve = np.random.randn(2, 2) + 1j*np.random.randn(2, 2)

power = 10  # power constraint (linear)

# Calculate the optimal transmit covariance matrix.
# This will be way faster than Loyka's algorithm
opt_cov = cov_secrecy_capacity_low_complexity(channel_bob, channel_eve, power=power)

# If you want to calculate the secrecy capacity for the found covariance
# matrix, you can use the secrecy_rate function
sec_cap = secrecy_rate(channel_bob, channel_eve, cov=opt_cov)

print("Optimal covariance matrix:")
print(opt_cov)
print("Secrecy capacity: {:.3f}".format(sec_cap))
```

2.3 References

CHAPTER

THREE

SECRECY_CAPACITY PACKAGE

3.1 Functions

duplication_matrix_fast(n)	Duplication matrix.
cov_secrecy_capacity_low_complexity(mat_bob, ...)	Optimal covariance matrix (low complexity implementation)
cov_secrecy_capacity_loyka(mat_bob, mat_eve)	Optimal covariance matrix (Loyka's algorithm)
secrecy_rate(mat_bob, mat_eve[, cov])	Calculation of the secrecy rate for given realizations of AWGN channels.

3.1.1 duplication_matrix_fast

secrecy_capacity.**duplication_matrix_fast**(*n*)

Duplication matrix. Implementation from: https://www.mathworks.com/matlabcentral/answers/473737-efficient-algorithm-for-a-duplication-matrix

3.1.2 cov_secrecy_capacity_low_complexity

secrecy_capacity.**cov_secrecy_capacity_low_complexity**(*mat_bob*, *mat_eve*, *power: float = 1*, *tol_eps: float = 1e-12*)

Optimal covariance matrix (low complexity implementation)

Calculate the optimal covariance matrix for a fading wiretap channel using the low complexity algorithm from [*1*].

Parameters

- **mat_bob** (*numpy.array*) – Matrix with the channel realizations of Bob's channels.
- **mat_eve** (*numpy.array*) – Matrix with the channel realizations of Eve's channels.
- **power** (*float*) – Power contraint at the transmitter.
- **tol_eps** (*float*) – Tolerance level for the inner algorithm, cf. *Algorithm 1* in [*1*].

Returns **cov** – Optimal covariance matrix which maximizes the secrecy rate.

Return type numpy.array

References

3.1.3 cov_secrecy_capacity_loyka

`secrecy_capacity.cov_secrecy_capacity_loyka`(*mat_bob*, *mat_eve*, *power: float = 10*, *t: float = 1000.0*, *alpha: float = 0.3*, *beta: float = 0.5*, *mu: float = 2*, *eps: float = 1e-10*, *dirname: Optional[str] = None*, *return_interm_results: bool = False*)

Optimal covariance matrix (Loyka's algorithm)

Calculate the optimal covariance matrix for a fading wiretap channel using the algorithm from [*1*].

Parameters

- **mat_bob** (*numpy.array*) – Matrix with the channel realizations of Bob's channels.
- **mat_eve** (*numpy.array*) – Matrix with the channel realizations of Eve's channels.
- **power** (*float*) – Power contraint at the transmitter.
- **alpha** (*float*) – Parameter α with $0 < \alpha < 0.5$ is a percent of the linear decrease in the residual one is prepared to accept at each step, cf. *Algorithm 1* in [*1*].
- **beta** (*float*) – Parameter β with $0 < \beta < 1$ is a parameter controlling the reduction in step size at each iteration of the algorithm, cf. *Algorithm 1* in [*1*].
- **mu** (*float*) – Parameter μ with $\mu > 1$ defines the multiplicator in the barrier method, cf. *Algorithm 3* in [*1*].
- **eps** (*float*) – Tolerance level for the outer barrier algorithm, cf. *Algorithm 3* in [*1*].
- **dirname** (*str*) – Path of the directory in which checkpoints and log should be saved. If `None`, no intermediate results will be saved.
- **return_interm_results** (*bool*) – If `True`, the history/intermediate results of the algorithm will be return together with the optimal covariance matrix. **This changes the return structure of the function!**

Returns

- **cov** (*numpy.array*) – Optimal covariance matrix which maximizes the secrecy rate.
- **(interm_res_norm, interm_sec_rate)** (*tuple of list of float*) – **Only returned, when** `return_interm_results == True`! Tuple that represents the history of the algorithm. The norm of the residual is stored in `interm_res_norm`, while the intermediate secrecy rates are in `interm_sec_rate`.

References

3.1.4 secrecy_rate

`secrecy_capacity.secrecy_rate`(*mat_bob*, *mat_eve*, *cov=None*) → float

Calculation of the secrecy rate for given realizations of AWGN channels.

Assuming AWGN fading channels to both Bob and Eve, this function calculates the secrecy rate for a given fading realization and covariance matrix at the transmitter.

Let H, G be the channel matrices to Bob and Eve, respectively, and Q the covariance matrix at Alice. The secrecy rate R_S is then given as

$$R_S = \max\left\{\log_2\left(I + HQH^H\right) - \log_2\left(I + GQG^H\right), 0\right\}$$

Parameters

- **mat_bob** (*float, complex or numpy.array*) – Matrix or scalar of Bob's channel realizations.
- **mat_eve** (*float, complex or numpy.array*) – Matrix or scalar of Eve's channel realizations.
- **cov** (*float or np.array*) – Covariance matrix (or transmit power) at Alice. If `None`, the identity matrix will be used.

Returns **sec_rate** – Secrecy rate R_S as described above.

Return type float

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

secrecy_capacity, 5

INDEX

C

`cov_secrecy_capacity_low_complexity()` (*in module secrecy_capacity*), 5
`cov_secrecy_capacity_loyka()` (*in module secrecy_capacity*), 6

D

`duplication_matrix_fast()` (*in module secrecy_capacity*), 5

M

`module`
    `secrecy_capacity`, 5

S

`secrecy_capacity`
    `module`, 5
`secrecy_rate()` (*in module secrecy_capacity*), 6