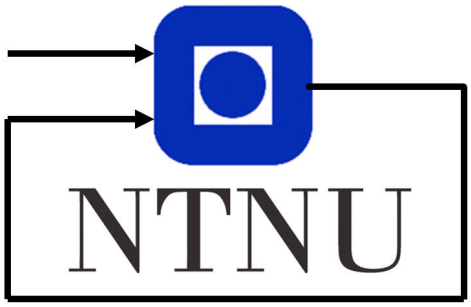
TTT4275

Estimation (1) project report

Solbø, Øystein

Strøm, Christopher

Spring, 2021



Department of Engineering Cybernetics

1 Summary

In many communication processes, a need to estimate certain parameters in various signals arises. One example of this is to estimate the phase and frequency of a complex exponential, embedded in noise. The signal-to-noise ratio [SNR] for the signal may impact the approach to estimation.

One way to perform this estimation is through a fast fourier transform [FFT] based maximum-likelihood estimator [MLE], although as shown in this report, this may become computationally infeasible for large SNRs.

To counteract computational cost, a hybrid between estimation and optimization can be deployed. This method consists of taking a computationally light FFT, then using an optimization technique to fine-tune the estimates provided by the FFT-based MLE. The resulting estimates in this case are shown to perform better than the pure FFT estimator while using less computational power. However, it may be unsuited for low SNR applications compared to a pure FFT estimator with similar computational cost, due to bias in the estimator.

Contents

1 Summary **1**

2 Theory **1**

2.1 General MLE and CRLB 1

2.2 Estimating phase and frequency 1

2.3 Optimization 3

3 Task **5**

4 Implementation and results **6**

4.1 Estimating using MLE 6

4.2 Estimating with optimization 11

5 Conclusion **16**

References **17**

Appendices

A Python implementation

A.1 Main . . . . .

A.2 Signal generation . . . . .

A.3 CRLB . . . . .

A.4 FFT Estimator . . . . .

A.5 Optimization . . . . .

A.6 Configuration parameters . . . . .

A.7 Plot generation . . . . .

A.8 Utility . . . . .

2 Theory

2.1 General MLE and CRLB

A general complex-exponential signal with gaussian noise, can be expressed as

$$x(t) = Ae^{j(2\pi ft+\phi)} + w(t), \tag{1}$$

where A is the magnitude, f is the frequency, ϕ is the phase and $w(t)$ is gaussian noise. All of these parameters are generally unknown, and must be estimated. There are several different techniques to estimate the parameters, and each technique would have different precision and variance.

One popular estimator is the maximum-likelihood estimator [MLE]. The MLE generates an estimate for a general parameter θ from a data vector x, by finding a $\hat{\theta}$ which maximises the likelihood to achieve said data. More mathematically, MLE uses the likelihood-function

$$L(\theta \,|\, x) = p(x;\theta), \tag{2}$$

where $p(x;\theta)$ is the pdf of x with parameter θ, to maximize the probability to get the measured data x. Thus, the MLE can be expressed as

$$\theta_{MLE} \;=\; \arg\max_{\theta} L(\theta \,|\, x), \tag{3}$$

which is a consistent and asymptotically efficient estimator.[1, pp. 25-28]

Ideally, an estimator will produce an estimate that is exactly the correct value every time. However, it can be shown that the estimated variance is lower-bounded by the Cramer-Rao Lower Bound [CRLB]. [1, pp. 6-13] The CRLB is the absolute minimum variance any consistent estimator could possible achieve. If the variance of an estimate is lower than the CRLB, it cannot be unbiased.

2.2 Estimating phase and frequency

Estimating the angular frequency and phase in equation 1, can be performed using MLEs. The MLE for the angular frequency is obtained by analysing the frequency response of the generated signal. For a general M-point FFT with sampling-time T, the sampled signal is distributed to M buckets. Each

bucket has a resolution of $\frac{1}{MT}$ Hz, and collects all frequencies that occur within a range specified by the resolution. The bucket that contains the largest magnitudal value of the FFT can then be expressed as

$$\hat{m} = \arg\max\{|\mathrm{FFT}_M(\boldsymbol{x})|\}, \tag{4}$$

which can be used to find the MLE for the angular frequency,

$$\hat{\omega} = \frac{2\pi\hat{m}}{MT}. \tag{5}$$

The CRLB for the angular frequency estimator is given as

$$var(\hat{\omega}) \geq \frac{12\sigma^2}{A^2T^2N(N^2-1)}, \tag{6}$$

where A is the signal's magnitude and N is the number of samples.

The MLE for the phase is optained from the estimated angular frequency $\hat{\omega}$. The MLE works by calculating the complex distance between an "ideal" signal with known angular frequency and the measured signal. Since the $\hat{\omega}$ in equation 5 is an MLE estimate, this frequency is used in the ideal signal. By using the N-periodic fourier series

$$F(\omega) = \sum_{n=0}^{N-1} x[n]e^{-j\omega nT}, \tag{7}$$

the measured signal is transformed into the frequency domain. The complex distance between signals can therefore be found by dividing this sum by the ideal signal. When disregarding noise, the only difference between the signals is the constant phase, which gives the phase-MLE as

$$\hat{\phi} = \angle\{e^{-j\hat{\omega}n_0T}F(\hat{\omega})\}, \tag{8}$$

where $n_0 = \frac{1-N}{2}$. The CRLB for the phase estimate is given as

$$var(\hat{\phi}) \geq \frac{12\sigma^2(n_0^2N + 2n_0P + Q)}{A^2N^2(N^2-1)}, \tag{9}$$

where $P = \frac{N(N-1)}{2}$, $Q = \frac{N(N-1)(2N-1)}{6}$ and $n_0 = \frac{-P}{N}$.

2.3 Optimization

When using M-point FFT, the resolution of each bucket is given as $\frac{1}{MT}$. When estimating frequency, especially with many frequencies in a close neighborhood, a large FFT-size may be required to get the desired resolution to differentiate the frequencies. This is computationally heavy, and it could be more efficient to utilize an optimization algorithm to estimate both frequency and phase.

An unconstrained optimization problem is defined as the minimization problem

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T G\boldsymbol{x} + c^T\boldsymbol{x} \\ \text{subject to} \quad & \boldsymbol{x}_0 \text{ given} \end{aligned} \tag{10}$$

where $f(\boldsymbol{x})$ is the objective-function. The problem consist of identifying the global solution $\boldsymbol{x}^*$, which gives

$$f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}),\ \forall\, \boldsymbol{x}. \tag{11}$$

Assuming the objective-function in equation 10 is convex, the algorithm will always find a solution $\boldsymbol{x}^*$ such that equation 11 is satisfied. If the problem is nonconvex however, the problem is in general NP-hard. Optimization-algorithms will therefore only try to find a local solution.[2, pp. 12-16]

Optimizing could be used to estimating the angular frequency or phase. A theoretical signal with the given iteration-characteristics, i.e. a noisefree signal with desired frequency or phase, could be compared to the measured signal. The mean-square-error [MSE], given as

$$f(\boldsymbol{x}, \hat{\boldsymbol{x}}) = \frac{1}{N}\sum_{k=0}^{N-1}(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k)^2, \tag{12}$$

could be used as an objective function. Assuming the measured signal having zero noise, the optimization problem is convex. This allows the algorithm to efficiently find the parameter which minimizes the distance between the theoretical and the measured signal. Thus achieving an optimized estimate for either $\hat{\omega}$ or $\hat{\phi}$.

In reality, the noise embedded in the measured signal makes the objective function non-convex. Non-convexity means that the algorithm may

converge at a local minimum which is not the global minimum. Since the objective-function is not continuous, neither the gradient nor Hessian is guaranteed to exist. An approximation like BFGS could work, but derivative-free-optimization algorithms like Nelder-Mead [NM] are to be preferred in this report. NM creates a simplex with $N + 1$ vertices in R^N, and continuously changing the simplex by moving the worst vertex/vertices towards the better ones. The algorithm does this until a termination criteria is fulfilled. [2, pp. 238-239]

3 Task

The project task consists of analysing a signal as described in equation 1. The magnitude and variance of the noise is considered to be known, respectively as

$$A = 1 \tag{13a}$$

$$\sigma^2 = \frac{A^2}{2\,\mathrm{SNR}}. \tag{13b}$$

This leaves the frequency f and the phase ϕ to be estimated. The estimates are found using two different methods, both using different levels for SNR.

The first method uses the classical approach for estimating a parameter, by implementing MLEs as described in section 2.2. The performance of the estimators with respect to varying SNRs and FFT lengths by simulating a sample space of N = 1000 for each combinations of the values in tables 1 and 2.

Table 1: SNR-values used to generate the embedded signal-noise. These values are all given in dB.

-10	0	10	20	30	40	50	60

Table 2: Length of the FFT used to analyze the signal. These values correspond to the parameter M used in equations 4 and 5. All of the lengths are in base 2 for efficiency.

2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}

The second method uses the Nelder-Mead algorithm to fine-tune the estimates. This is to avoid the computational cost associated with large FFT lengths, to be able to use a shorter FFT while achieving good estimator performance. The estimates found with MLE using FFT length of M = 2^{10}, is used as the initial iteration value by the optimization-algorithm. The objective-function is the MSE as described in section 2.3. No criteria is fixed for the optimization algorithm, thus leaving it to terminate once a minimum - within computer resolution - is found.

4 Implementation and results

4.1 Estimating using MLE

The implemented code is shown in appendix A, with flow-diagrams shown in figure 1 and 2. To analyze the performance of the MLEs, the average mean and variance is calculated for different FFT-lenghts and SNRs.

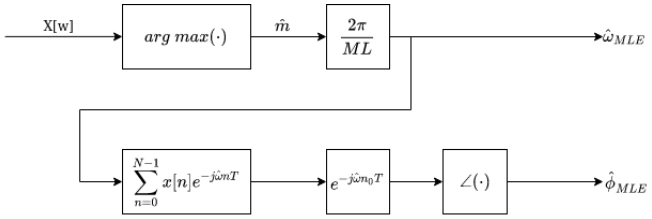


Figure 1: The FFT estimator for ω and ϕ. Note that the estimate $\hat{\phi}$ depends on the estimate $\hat{\omega}$

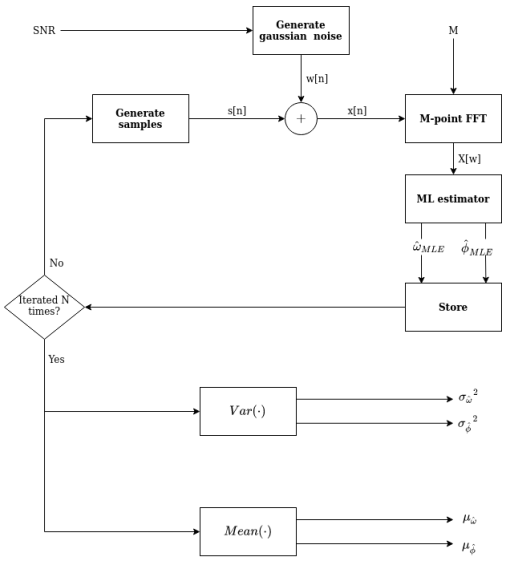


Figure 2: The structure of the system used to calculate the variance of the estimators, for varying FFT size and SNR.

Figures 5 and 6 show how the variance for each estimator depends on SNR and FFT size, and compares this to the CRLB. As can be expected from equation 6 and 9, the CRLB decreases for increasing SNR. The variance

of the estimators follow this trend. However, some estimates fall well below the CRLB.

Using shorter FFT lengths and higher SNRs seem to make the frequency estimates fall below the CRLB. The cause of this comes from bucket size and where each individual estimate lands: With a small FFT size, the frequency resolution is very coarse, meaning that small variations around a central value will not be picked up since they all fall in the same bucket. If every estimate falls in the same bucket, the following estimator will have zero variance and a bias towards the frequency corresponding to this bucket. Similarly, a high SNR means that the estimates will have less spread and outliers, also leading to the estimates landing in the same bucket. This effect can be clearly seen in figure 5, where the variance for the estimator drops to zero at some SNR for each choice of FFT length. Before dropping to zero, the variance increases for a few values of SNR. No apparent explanation for this was found, but it does highlight that the estimator will perform worse under these circumstances.

The effect of the bias in the estimator can also be seen in the mean value for the frequency in figure 3, where the mean value for each length of FFT reaches a constant offset from the true value. This occurs at the same SNRs as where the corresponding variance falls to zero. The SNR at which the estimator collapses increases and the mean frequency moves closer to the true frequency for each increase in FFT length, suggesting that the frequency estimator is asymptotically an MVU estimator for $M \to \infty$.

The variance of the phase estimate can be seen in figure 4 and comparatively it performs much than the frequency estimator since it never drops off below the CRLB. However, since the phase estimate depends on the frequency estimate, any bias in the frequency estimator will carry over to the phase estimator. This can explain the offsets in the mean of the phase estimate for low SNRs, in figure 4. These results suggest that the phase estimator is asymptotically an MVU estimator for $M \to \infty$ and SNR greater than some threshold.

As described in section 2, no unbiased estimator can have a variance lower than the CRLB. The FFT estimator presented here does not break with this theory, despite producing some estimates with variance far lower than the CRLB. This is because the bucket-effect that causes zero variance

only occurs as an artifact of using a finite-precision FFT. That is to say, if the estimates were computed using an FFT with infinite precision, the estimators would be valid for all SNRs.

Whenever an estimate has lower variance than the CRLB, the estimate will be heavily biased towards one of the FFT buckets. Because of this bias, the estimator will be labelled as unsuited for this particular combination of SNR and FFT length. A pattern that emerges in figure 5 is that to get a valid estimator for an increase of 10dB in SNR, the FFT length has to be increased by a factor 10^2. Signals with larger SNR thus require the use of a very large FFT length, which can quickly become computationally infeasible: Since the FFT algorithm is $O(N \cdot logN)$, an increase in length from 2^{10} to 2^{20} requires around 100 times the computation time.

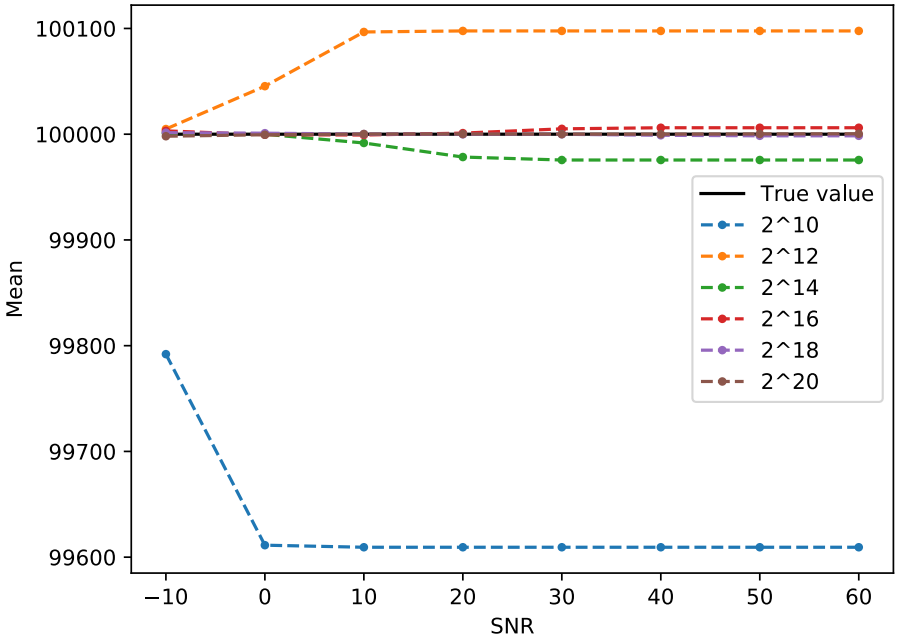


Figure 3: The mean value for the frequency estimate for a sample size of $N = 1000$. The points at which the mean value reaches a steady-state is the point where the estimator falls below the CR bound and becomes invalid.

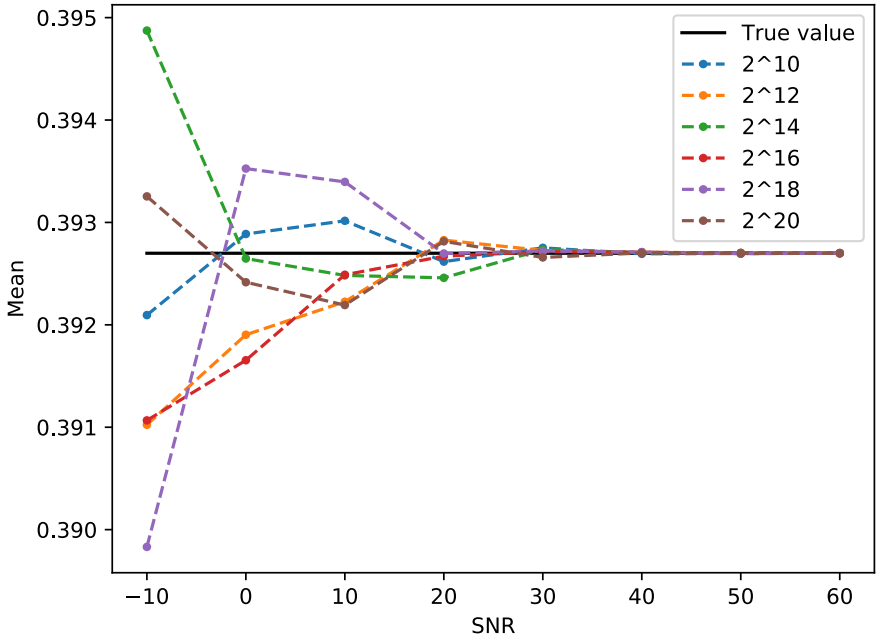


Figure 4: The mean value for the phase estimate for a sample size of $N = 1000$ for different FFT lengths and SNRs. For lower SNRs, there is a clear spread around the true mean, regardless of FFT length. Each choice of FFT length leads to a mean that converges to the true mean at around 30 dB SNR.

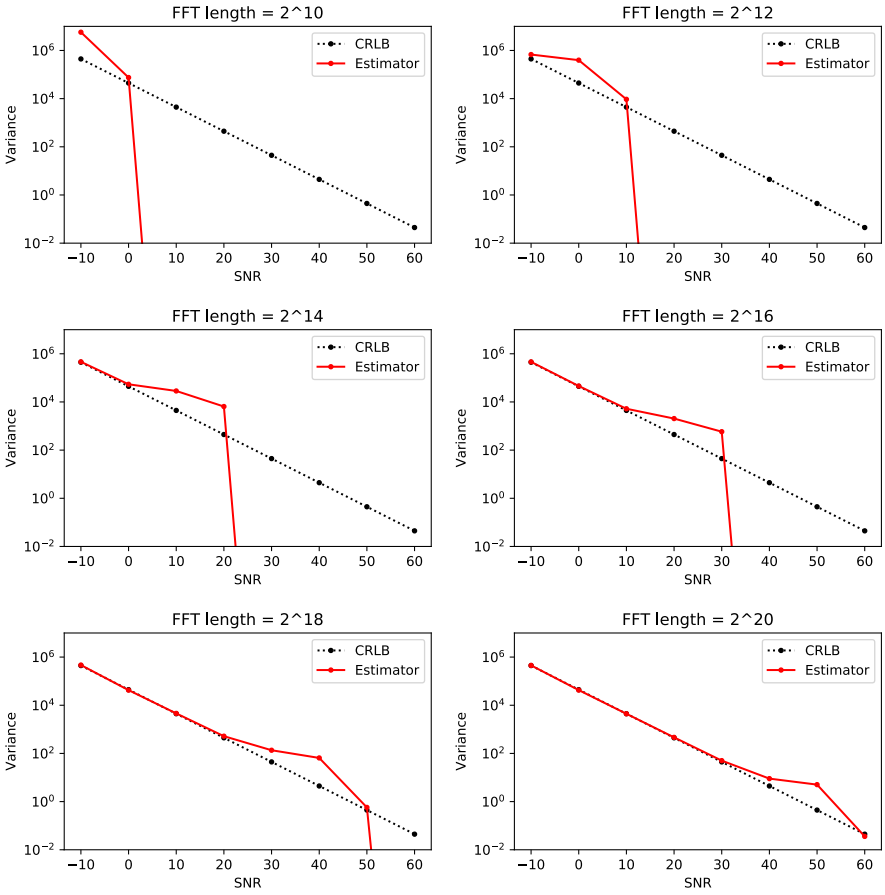


Figure 5: Variance for the frequency estimates for varying FFT lengths and SNRs, using a sample size of $N = 1000$. Note that when the estimator variance falls below the CR bound all the estimates have fallen in the same frequency bucket, and leads to a bias in the estimate.

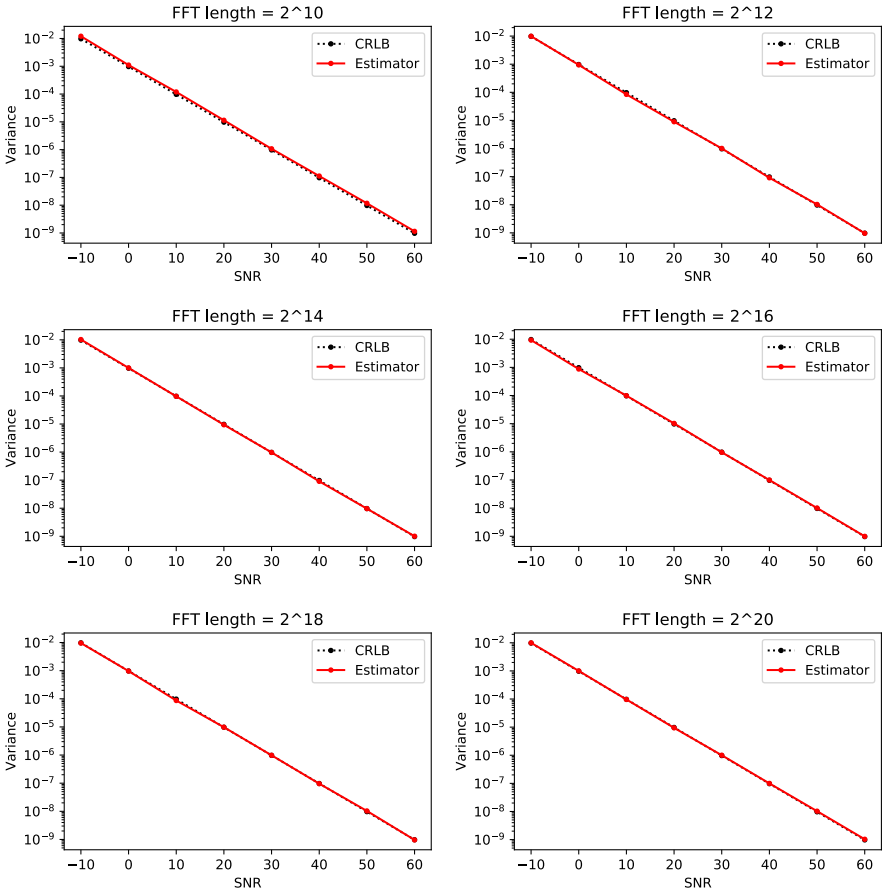


Figure 6: The variance for the phase estimates for varying FFT lengths and SNRs, which follows the CRLB very tightly. These estimates are dependent on the estimates for the frequency presented in figure 5.

4.2 Estimating with optimization

To counteract the computational infeasibility of large FFT lengths, the NM based fine-tuning is introduced to the FFT estimator with a length of 2^{10}. The code is shown in appendix A, and is implemented as described in section 2.3.

Figure 7 shows the variance when optimizing the frequency. Estimates obtained for low SNRs have variances below the CRLB. This is caused by the convexity properties of the objective function used in the optimization.

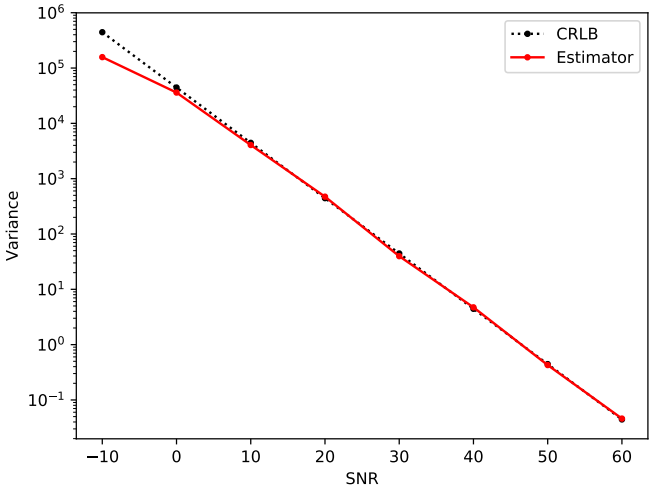


Figure 7: The variance of frequency estimate from the FFT estimator using a length of 2^{10}, fine-tuned by the Nelder-Mead simplex method, using a sample size of $N = 1000$.

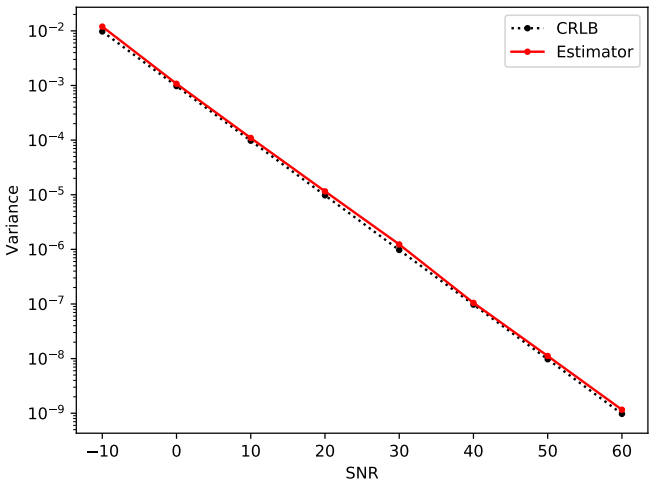


Figure 8: The variance of phase estimate from the FFT estimator using a length of 2^{10}, fine-tuned by the Nelder-Mead simplex method, using a sample size of $N = 1000$.

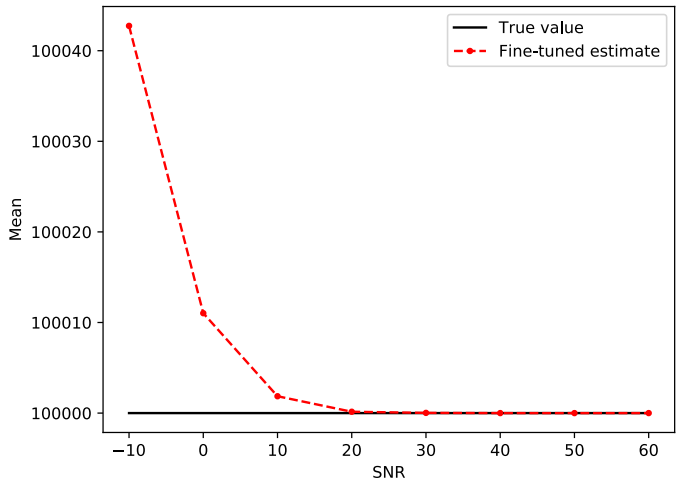


Figure 9: The mean value for the fine-tuned frequency estimate as a function of SNR, using a sample size of $N = 1000$.

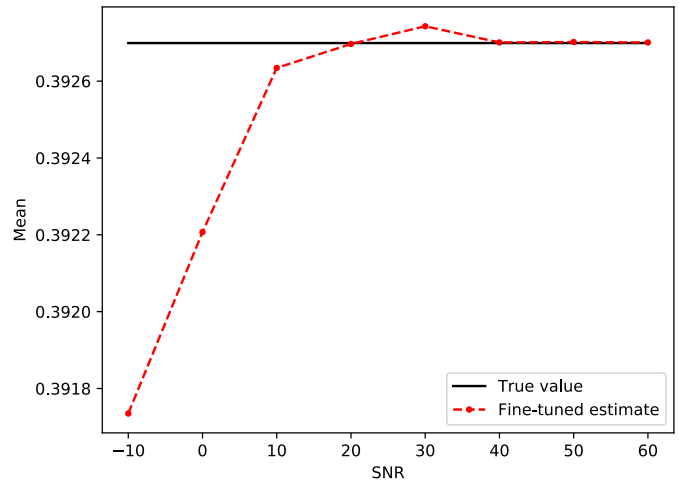


Figure 10: The mean value for the fine-tuned phase estimate as a function of SNR, using a sample size of $N = 1000$.

For low SNRs, the noise creates a non-convex problem with multiple local solutions. Multiple optimizations therefore converges to local minima close to the initial value. This leads to the optimization layer producing estimates with lower variance than CRLB, and the results are therefore biased until the variance surpasses the CRLB at SNR around 20 dB. The effect of this can also be seen in the mean frequency in figure which is biased for SNRs $<$ 20. As the SNR increases, the variance of the frequency estimate surpasses the CRLB, and the mean value follows the true value. For higher SNRs, optimization is therefore preferable, as it gives unbiased estimates with fewer computations.

The variance of the optimized phase estimate is shown in figure 8, and has a similar trajectory to the estimated phase variance in figure 6, where it follows the CRLB very tightly. As the objective function uses the optimized frequency estimate, the phase estimate will be slightly biased for low SNRs. The optimized phase is therefore only unbiased once the SNR surpasses 20 dB. If one allow a slight error from bias and noise, an optimization-algorithm for fine-tuning estimates is usable for even lower SNRs.

The mean of the optimized phase estimate is shown in figure 10. The optimized phase converges to the true phase once the noise gets low enough. The reasoning is similar as for the optimized frequency, where lower noise creates a more convex optimization problem. It is therefore easier for the NM-algorithm to differentiate the global solution from the local solutions, thus achieving a better result. Comparing it to the estimated phase mean for FFT-length of 2^{10} in figure 4, the fine-tuned phase does not directly yield an improved estimate. However, the optimized phase mean is calculated from a less biased frequency estimate, thus being unbiased for a larger range of SNRs.

The optimization algorithm has been implemented without any requirements on convergence. The algorithm therefore uses standard requirements for both tolerance and allowed iterations when deciding to cancel the iterations. To improve the computational efficiency, the tolerance and maximum number of iterations could be increased or decreased, respectively. The drawback is that this reduces the accuracy of the tuned variables, meaning that such errors must be expected. Alternatively, the optimization algorithm

could be limited to less noisy systems.

Using an optimization algorithm to fine-tune the estimates in frequency and phase, appears to be an efficient method to achieve reliable estimates with relatively low computational cost. For low SNRs, the optimized results are biased, such that only using MLE will likely be more efficient. Once the SNR gets high enough, the computational cost of using a pure MLE increases. Using an optimization algorithm to fine tune the MLE estimate is therefore far more efficient and accurate. Thus, the estimation method for estimating phase and frequency relies on SNR. Choosing the correct method could have large impact on the performance.

5 Conclusion

The performance of the FFT estimator was found to be dependent on SNR and FFT length. To increase the valid SNR range of the frequency estimator by 10 dB, the FFT length would have to be increased by a factor of 4. The estimated frequency mean was found to reach a steady state offset depending on the validity range of the frequency estimate. The phase estimator performed better, following the CRLB for every SNR. The estimated phase mean closed in on the true phase for increasing SNR, although it suffered from scatter around the true phase for lower SNRs.

Adding an optimization layer for fine tuning the FFT estimates of length 2^{10}, which led to the validity range of the frequency estimator to span the entire range of SNRs used, while keeping the good performance of the phase estimate. Both the means of the estimated frequency and phase converge to their respective true values, although the same bias as with the FFT estimator alone was observed. Adding the optimization thus gave satisfactory results compared to the pure FFT estimator, since it provided better performance using less computational power. However, for low SNRs, the bias in the estimated means may perform better using longer FFT lengths and no optimization. The threshold for this is not set in this report, although it could be of interest in a real-world implementation of this estimator.

References

[1] T. A. Myrvoll. Estimation Theory, 2018.

[2] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, second edition, 2006.

Appendices

A Python implementation

Below is the entire impmentation of the code used to generate signals and for the estimators. Refer to `https://github.com/chrstrom/monkey_estimation` if you wish to download and run the project.

A.1 Main

```
#!/usr/bin/env python

import csv
import sys
import os, os.path
import numpy as np

from datetime import datetime as dt
from scipy import optimize

from scripts import signals as sig
from scripts import fft_estimator
from scripts import optimizing
from scripts import utility
from scripts import crlb
from scripts import cfg


try:
    task = sys.argv[1]
except Exception as e:
    print("No input task provided, exiting. \n Usage: python main.↘
        py <task>")
    exit(1)


SNR_dBs =[-10, 0, 10, 20, 30, 40, 50, 60]
FFT_Ks = [10, 12, 14, 16, 18, 20]

n = len(SNR_dBs)
m = len(FFT_Ks)
N = 100  # Amount of samples to generate when estimating variance

# Generate unique filename for data file output
```

```
34 run_number = len([name for name in os.listdir('./data') if os.path↘
       .isfile('./data/' + name)])
35
36 if task == 'a':
37     filename = 'data/part_a_run_' + str(run_number) + '_N_' + str(↘
           N) + '.csv'
38     with open(filename, 'ab') as file:
39         writer = csv.writer(file, delimiter=' ')
40
41         total_time_begin = dt.now()
42         for i in range(m):
43             K = FFT_Ks[i]
44             M = 2**K
45
46             for j in range(n):
47                 SNR_dB = SNR_dBs[j]
48
49                 w_estimates = np.zeros(N)
50                 phi_estimates = np.zeros(N)
51
52                 status_bar_progress = 0
53                 run_time_begin = dt.now()
54                 for k in range(N):
55                     x_d = sig.x_discrete(SNR_dB)
56
57                     omega_hat, phi_hat, _, _ = fft_estimator.↘
                           estimator(x_d, M)
58
59                     w_estimates[k] = omega_hat
60                     phi_estimates[k] = phi_hat
61
62                     status_bar_progress = utility.print_status_bar↘
                           (k, status_bar_progress, N)
63
64                 mean_f = np.mean(w_estimates) / (2*np.pi)
65                 mean_phi = np.mean(phi_estimates)
66
67                 var_f = np.var(w_estimates)
68                 var_phi = np.var(phi_estimates)
69
70                 crlb_f = crlb.omega(SNR_dB)
71                 crlb_phi = crlb.phi(SNR_dB)
72
73                 run_time_end = dt.now()
74                 print("")
75                 utility.print_execution_time(run_time_begin, ↘
                       run_time_end)
76
```

```
77                  f_estimate_valid = True
78                  phi_estimate_valid = True
79                  if var_f < crlb_f:
80                      f_estimate_valid = False
81                      print("Variance for frequency lower than CRLB!↘
                            ")
82
83                  if var_phi < crlb_phi:
84                      phi_estimate_valid = False
85                      print("Variance for phi lower than CRLB!")
86
87
88                  writer.writerow([SNR_dB, K, crlb_f, var_f, ↘
                        f_estimate_valid, crlb_phi, var_phi, ↘
                        phi_estimate_valid, mean_f, mean_phi])
89
90                  print("CONFIG | SNR [dB]: {}, M: 2^{}, true ↘
                        frequency: {}, true phase: {}".format(SNR_dB, ↘
                        K, cfg.f0, cfg.phi))
91                  print("FREQUENCY  | estimated mean: {}, estimated ↘
                        variance: {}, crlb: {}".format(mean_f, var_f, ↘
                        crlb_f))
92                  print("PHASE  | estimated mean: {}, estimated ↘
                        variance: {}, crlb: {}".format(mean_phi, ↘
                        var_phi, crlb_phi))
93                  print("")
94
95          total_time_end = dt.now()
96          utility.print_execution_time(total_time_begin, ↘
                total_time_end)
97
98  if task == 'b':
99      filename = 'data/part_b_run_' + str(run_number) + '_N_' + str(↘
            N) + '.csv'
100     with open(filename, 'ab') as file:
101         writer = csv.writer(file, delimiter=' ')
102         M = 2**10
103
104         total_time_begin = dt.now()
105         for SNR_dB in SNR_dBs:
106
107             w_estimates = np.zeros(N)
108             phi_estimates = np.zeros(N)
109
110             status_bar_progress = 0
111             run_time_begin = dt.now()
112
113             for i in range(N):
```

```
114
115                 x_d = sig.x_discrete(SNR_dB)
116
117                 omega_hat, phi_hat, _, _ = fft_estimator.estimator↘
                        (x_d, M)
118
119                 omega_opt = optimize.minimize(optimizing.↘
                        frequency_objective_function, omega_hat, ↘
                        method="Nelder-Mead", args=(M, x_d, phi_hat))
120                 phase_opt = optimize.minimize(optimizing.↘
                        phase_objective_function, phi_hat, method="↘
                        Nelder-Mead", args=(x_d, omega_hat))
121
122                 w_estimates[i] = omega_opt.x[0]
123                 phi_estimates[i] = phase_opt.x[0]
124
125                 status_bar_progress = utility.print_status_bar(i, ↘
                        status_bar_progress, N)
126
127             run_time_end = dt.now()
128             print("")
129             utility.print_execution_time(run_time_begin, ↘
                    run_time_end)
130
131             mean_f = np.mean(w_estimates) / (2*np.pi)
132             mean_phi = np.mean(phi_estimates)
133
134             var_f = np.var(w_estimates)
135             var_phi = np.var(phi_estimates)
136
137             crlb_f = crlb.omega(SNR_dB)
138             crlb_phi = crlb.phi(SNR_dB)
139
140             f_estimate_valid = True
141             phi_estimate_valid = True
142
143             if var_f < crlb_f:
144                 f_estimate_valid = False
145                 print("Variance for f lower than CRLB!")
146
147             if var_phi < crlb_phi:
148                 phi_estimate_valid = False
149                 print("Variance for phi lower than CRLB!")
150
151
152             writer.writerow([SNR_dB, 10, crlb_f, var_f, ↘
                    f_estimate_valid, crlb_phi, var_phi, ↘
                    phi_estimate_valid, mean_f, mean_phi])
```

```
153             print("CONFIG | SNR [dB]: {}, M: 2^{}, true f: {}, ↘
                    true phase: {}".format(SNR_dB, 10, cfg.f0, cfg.phi↘
                    ))
154             print("FREQUENCY  | estimated mean: {}, estimated ↘
                    variance: {}, crlb: {}".format(mean_f, var_f, ↘
                    crlb_f))
155             print("PHASE  | estimated mean: {}, estimated variance↘
                    : {}, crlb: {}".format(mean_phi, var_phi, crlb_phi↘
                    ))
156             print("")
157
158         total_time_end = dt.now()
159         utility.print_execution_time(total_time_begin, ↘
                total_time_end)
```

A.2 Signal generation

```
1  #!/usr/bin/env python
2
3  import sys
4  import os
5
6  if os.name == 'nt':
7      sys.path[0]=os.path.dirname(os.path.realpath(__file__))
8
9  import cfg
10 import numpy as np
11
12 def sigma_squared_from_SNR_dB(SNR_dB):
13     """
14     Calculate the value for sigma^2 according to the
15     definition.
16     """
17     SNR = 10**(SNR_dB/10.0)
18     return cfg.A**2 / (2*float(SNR)) # Float casting prevents ↘
           floor division
19
20 def F(x_d, w):
21     """
22     Calculate F(w) according to Eq. (6) in the project
23     specifications.
24     """
25     sum = 0
26     for n in range(0, cfg.N):
27         sum += x_d[n]*np.exp(-1j*w*n*cfg.Ts)
28
29     return sum / cfg.N
30
```

```
31  def x_discrete(SNR_dB):
32      """
33      Generate a signal according to the problem spec.
34      which consists of a complex exponential with
35      added noise. Noise-to-signal ratio defined by SNR
36      """
37      sigma = np.sqrt(sigma_squared_from_SNR_dB(SNR_dB))
38
39      wr = np.random.normal(0, sigma, cfg.N)
40      wi = np.random.normal(0, sigma, cfg.N)
41      w = wr + 1j*wi
42
43      x = np.empty(cfg.N, dtype=np.complex_)
44
45      for n in range(cfg.N):
46          z = 1j*(cfg.w0 * (n+cfg.n0) * cfg.Ts + cfg.phi)
47          x[n] = cfg.A * np.exp(z)
48
49      return x + w
50
51
52  def x_ideal(omega, phase):
53      """
54      Generates a complex-exponential signal with given frequency
55      and phase. Does not contain noise
56      """
57      x = np.empty(cfg.N, dtype=np.complex_)
58
59      for n in range(cfg.N):
60          z = 1j*(omega * (cfg.n0+n) * cfg.Ts + phase)
61          x[n] = cfg.A * np.exp(z)
62
63      return x
```

A.3 CRLB

```
1  #!/usr/bin/env python
2
3  import cfg
4  import signals as sig
5
6
7  def omega(SNR_dB):
8      sigma_squared = sig.sigma_squared_from_SNR_dB(SNR_dB)
9
10     numerator = 12*sigma_squared
11     denominator = cfg.A**2 * cfg.Ts**2 * cfg.N*(cfg.N**2 - 1)
12
```

```
13      return numerator / denominator
14
15  def phi(SNR_dB):
16      sigma_squared = sig.sigma_squared_from_SNR_dB(SNR_dB)
17
18      numerator = 12*sigma_squared*(cfg.n0**2 * cfg.N + 2*cfg.n0*cfg↘
            .P + cfg.Q)
19      denominator = cfg.A**2 * cfg.N**2 * (cfg.N**2 - 1)
20
21      return numerator / denominator
```

A.4 FFT Estimator

```
1   #!/usr/bin/env python
2
3   import signals
4   import cfg
5
6   import numpy as np
7
8   def calculate_m_star(Fw):
9       """
10      Calculate the m_star as described by
11      Eq. (5) and Eq. (9) in the project spec.
12      """
13      return np.argmax(np.absolute(Fw))
14
15  def calculate_w_hat(m_star, M):
16      """
17      Calculate the angular frequency estimate
18      w_FFT_hat as described by Eq. (8) in the
19      project spec
20      """
21      return 2*np.pi*m_star / (M*cfg.Ts)
22
23  def calculate_phi_hat(x_d, w_hat):
24      """
25      Calculate the phase estimate phi_hat as
26      described by Eq. (7) in the project spec
27      """
28      F_w_hat = signals.F(x_d, w_hat)
29      phi_arg = np.exp(-1j*w_hat*cfg.n0*cfg.Ts)*F_w_hat
30      return np.angle(phi_arg)
31
32  def estimator(x_d, M):
33      """
34      Use the M-point FFT estimator described in the
35      problem spec. to estimate the angular frequency
```

```
36      "w" and phase "phi" of the input signal x_d.
37
38      Returns the estimates w_hat and phi_hat, as well
39      as the FFT of the input signal, for data analysis
40      purposes.
41      """
42      Fw, Ff = M_point_fft(x_d, M)
43
44      m_star = calculate_m_star(Fw)
45
46      w_hat = calculate_w_hat(m_star, M)
47      phi_hat = calculate_phi_hat(x_d, w_hat)
48
49
50      return w_hat, phi_hat, Fw, Ff
51
52  def M_point_fft(x_d, M):
53      Fw = np.fft.fft(x_d, M)
54      Ff = np.fft.fftfreq(M, 1 / cfg.Fs)
55
56      return Fw, Ff
```

A.5 Optimization

```
1  #!/usr/bin/env python
2
3  import csv
4  import matplotlib.pyplot as plt
5  from collections import Counter
6
7  from scipy import optimize
8  from math import pi, floor
9  import numpy as np
10
11 import fft_estimator
12 import signals
13 import cfg
14
15 def mse(list_lhs, list_rhs):
16   """
17   Calculates the MSE between two lists. Throws an error if the ↘
         lists don't
18   have the same lenght
19   """
20   assert(len(list_lhs) == len(list_rhs))
21   return np.square(np.absolute(list_lhs - list_rhs)).mean()
22
23
```

```
24  def frequency_objective_function(x, M, x_d, phi_hat):
25    """
26    Creates the objective-function for optimizing the frequency. The
27    function assumes the input to be a ndarray, with the first value
28    being the next frequency/iteration to minimize for. The ↘
          algorithm
29    uses this frequency to create a theoretical signal, and returns ↘
          the
30    MSE wrt to the measured signal
31    """
32
33    omega_k = x[0]
34    x_f = signals.x_ideal(omega_k, phi_hat) # Phase has no effect as↘
           it removed through FFT
35
36    Fx_d, _ = fft_estimator.M_point_fft(x_d, M)
37    Fx_f, _ = fft_estimator.M_point_fft(x_f, M)
38
39    return mse(np.absolute(Fx_d), np.absolute(Fx_f))
40
41
42  def phase_objective_function(x, x_d, omega_hat):
43    """
44    Creates the objective-function for optimizing the phase. The
45    function assumes the input to be a ndarray, with the first value
46    being the next phase/iteration to minimize for. The algorithm
47    uses this phase to create a theoretical signal, and returns the
48    MSE wrt to the measured signal
49    """
50    phi_k = x[0]
51
52    x_p = signals.x_ideal(omega_hat, phi_k)
53
54    return mse(x_d, x_p)
```

A.6 Configuration parameters

```
1  #!/usr/bin/env python
2
3  from cmath import pi
4
5  N = 513
6  M = 2**10
7
8  Fs = 1e6
9  Ts = 1.0 / Fs
10
11 A = 1
```

```
phi = pi / 8.0

f0 = 1e5
w0 = 2 * pi * f0

P = N * (N - 1) / 2.0
Q = N * (N - 1) * (2 * N - 1) / 6.0

n0 = int(-P / N)

num_opt = 100
```

A.7 Plot generation

```
#!/usr/bin/env python

import csv
import sys
import numpy as np
import matplotlib.pyplot as plt
import cfg


def plot_var_task_a(ax, crlb, estimator_variance, ylim=None):
    plt.tight_layout()
    for i in range(N_ROWS):
        for j in range(N_COLS):
            n = range(N*(2*i+j), N*(2*i+j+1))
            axis = ax[i][j]
            axis.semilogy(SNRs, crlb[n], 'k.:')
            axis.semilogy(SNRs, estimator_variance[n], 'r.-')
            axis.set_title("FFT length = 2^" + str(4*i + 2*j + 10)
                )
            axis.set_xlabel("SNR")
            axis.set_ylabel("Variance")
            axis.legend(['CRLB', 'Estimator'])

            if ylim is not None:
                axis.set_ylim(ylim)

def plot_mean_task_a(true_mean, estimated_mean):
    plt.tight_layout()
    plt.plot([-10, 60], [true_mean, true_mean], 'k')
    for i in range(M):
        n = range(N*i, N*(i+1))
        plt.plot(SNRs, estimated_mean[n], '.--')

    plt.legend(["True value", "2^10", "2^12", "2^14", "2^16", "
```

```
        2^18", "2^20"])
34      plt.xlabel("SNR")
35      plt.ylabel("Mean")
36
37  def plot_var_task_b(crlb, estimator_variance):
38      plt.tight_layout()
39      plt.semilogy(SNRs, crlb[0:N], 'k.:')
40      plt.semilogy(SNRs, estimator_variance[0:N], 'r.-')
41      plt.legend(['CRLB', 'Estimator'])
42      plt.xlabel("SNR")
43      plt.ylabel("Variance")
44
45  def plot_mean_task_b(true_mean, estimated_mean):
46      plt.rc('axes.formatter', useoffset=False)
47      plt.plot([-10, 60], [true_mean, true_mean], 'k')
48      plt.plot(SNRs, estimated_mean, 'r.--')
49      plt.legend(["True value", "Fine-tuned estimate"])
50      plt.xlabel("SNR")
51      plt.ylabel("Mean")
52
53
54  if __name__ == '__main__':
55
56      try:
57          filename = sys.argv[1]
58      except Exception as e:
59          print("No input file provided, exiting. \n Usage: python ↘
                plot.py 'filename.csv'")
60          exit(1)
61
62      if "_a_" in filename:
63          task = 'a'
64      elif "_b_" in filename:
65          task = 'b'
66      else:
67          print("No matching task file provided, exiting...")
68          exit(1)
69
70      SNRs = [-10, 0, 10, 20, 30, 40, 50, 60]
71      Ks = [10, 12, 14, 16, 18, 20]
72
73      N = len(SNRs)
74      M = len(Ks)
75
76      crlb_w = np.empty(N*M)
77      crlb_phi = np.empty(N*M)
78
79      var_w = np.empty(N*M)
```

```
80      var_phi = np.empty(N*M)
81
82      mean_f = np.empty(N*M)
83      mean_phi = np.empty(N*M)
84
85      w_estimate_valid = np.empty(N*M)
86      phi_estimate_valid = np.empty(N*M)
87
88
89      with open(filename) as csvfile:
90
91          reader = csv.reader(csvfile, delimiter=' ')
92
93          i = 0
94          for row in reader:
95              K           = row[1]
96
97              crlb_w[i]   = row[2]
98              var_w[i]    = row[3]
99
100             crlb_phi[i] = row[5]
101             var_phi[i]  = row[6]
102
103             mean_f[i]   = row[8]
104             mean_phi[i] = row[9]
105
106             w_estimate_valid[i]   = bool(row[4])
107             phi_estimate_valid[i] = bool(row[7])
108
109             i += 1
110
111     if task == 'a':
112         N_ROWS = M/2
113         N_COLS = 2
114
115         _, ax = plt.subplots(N_ROWS, N_COLS)
116         plt.figure(1)
117         plot_var_task_a(ax, crlb_w, var_w, [0.01, 1e7])
118
119         _, ax = plt.subplots(N_ROWS, N_COLS)
120         plt.figure(2)
121         plot_var_task_a(ax, crlb_phi, var_phi)
122
123
124         plt.figure(3)
125         plot_mean_task_a(cfg.f0, mean_f)
126
127         plt.figure(4)
```

```
128             plot_mean_task_a(cfg.phi, mean_phi)
129
130         if task == 'b':
131
132             plt.figure(1)
133             plot_var_task_b(crlb_w, var_w)
134
135             plt.figure(2)
136             plot_var_task_b(crlb_phi, var_phi)
137
138             plt.figure(3)
139             plot_mean_task_b(cfg.f0, mean_f[0:N])
140
141             plt.figure(4)
142             plot_mean_task_b(cfg.phi, mean_phi[0:N])
143
144         plt.show()
```

A.8 Utility

```
1  #!/usr/bin/env python
2
3  import sys
4
5  def print_status_bar(i, progress, N, size=50):
6      if i % (N/size) == 0:
7          progress += 1
8
9      sys.stdout.write('\r')
10     sys.stdout.write("Status: [" + "="*progress + " "*(size-↘
           progress) +  "]")
11     sys.stdout.flush()
12
13     return progress
14
15 def print_execution_time(begin, end):
16     total_calculation_time = float((end - begin).total_seconds())
17     print("Calculation time: %f seconds" % total_calculation_time)
```