EE312 Term Project Report
Spring 2025

14/05/2025

Ahmet Emre Eser

31273

Introduction

The purpose of this report is to outline the details and results of an implementation of “Adaptive Savitzky-Golay Filtering in Non-Gaussian Noise” by John et. al (2021). The implementation was done in the Matlab scientific computing environment.

Savitzky-Golay Filter Background

The Savitzky-Golay (SG) filter works by isolating a window of sample points of a predetermined length and applying polynomial regression of a predetermined order over the isolated window. Afterwards, the datapoint at the center of the window will be replaced by the value of the polynomial and the window will slide 1 unit to the future and begin the next iteration.

One of the most critical tasks in using the SG filter is determining the filter length and order of the polynomial fits. The authors of the chosen article have proposed several algorithms where they select SG parameters adaptively. The results were reported in the form of output signal-to-noise ratio (SNR) - input SNR plots, where the denoising performance of the adaptive filter could be visualized and analyzed.

Implementation

Risk Functions

Calculating the error between the filtered signal and the original (noiseless) signal requires knowing the original signal, which is often impossible in real life applications. The authors instead proposed a prediction framework for the MSE of the reconstructed signal termed “Generalized Unbiased Estimate of MSE” (GUE-MSE). GUE-MSE is a modified version of Stein’s unbiased risk estimate (John, 2021, p. 5022). This report and the corresponding Matlab implementation refer to GUE-MSE by ‘risk function’.

The authors present two versions of the risk function, base and regularized. Regularized risk function contains an additional term designed to penalize sudden transitions (p. 5026). Definitions and Matlab functions for the risk functions could be found below:

Base Risk Function

The base GUE-MSE/risk function is defined as follows (John, 2021, p. 5025):

$$\widehat{R} = \frac{1}{2M+1}\left(||AHx||^2 - 2x^T H^T A^T x + 2\sigma^2 \sum_{l=-M}^{M} (H^T A^T)_{l,l} + ||x||^2 \right) - \sigma^2$$

where M refers to the window size, $(...)_{n,n}$ refers to the element at the nth row and nth column of the matrix enclosed in parentheses, H and A refer to the ‘hat’ and ‘design’ matrices used in

polynomial regression, ||...|| refers to the vector normalization operation (||x|| = $\sqrt{x_1^2 + \ldots + x_n^2}$ for length(x) = n) and x refers to the noisy signal vector.

The noise in the signal is assumed to have a standard deviation of σ^2 (p. 5027). The authors used a table or prediction values for the noise types examined (gaussian, uniform, laplacian) (p. 5028). This project's Matlab implementation generates noise from given standard deviation values, hence prediction based values were not necessary for σ^2.

Matlab implementation of the base risk function is provided below:

```
function R = UNregularized_risk_estimate(M, n0, A, H, signal, sigma)
   sigma_sq = sigma * sigma;
   wind_size = 2*M + 1;
   x = signal(n0-M:n0+M)';
   term1 = norm(A * H * x)^2;
   term2 = -2 * x' * H' * A' * x;
   term3 = 2 * sigma_sq * sum( diag(H' * A') );
   term4 = norm(x)^2;
   R = (1/wind_size) * (term1 + term2 + term3 + term4) - sigma_sq;
end
```

The parameter n0 refers to the instance of the noisy signal vector for which we're calculating the risk. The function will calculate the risk for the window of length M with point n0 at its center.

Regularized Risk

The regularized risk function introduces an extra additive term to the base risk function in order to penalize spurious transitions (John, 2021, p. 5025). The regularized risk function is provided below:

$$R_\lambda = \widehat{R} + \frac{\lambda}{2M+1}\left(\sum_{m=-M}^{M} (AH)_{m,m}^2 \right)$$, where $\lambda = 12\sigma^2$

The second term calculates the sensitivity of the denoised estimate of the original signal to changes in the noisy signal. Hence risk value increases for more spontaneously transitioning estimates. Please refer to the article for the derivation of the regularizer term (p. 5026). The implementation of the regularized risk function is provided below:

```
function R = regularized_risk_estimate(M, n0, A, H, signal, sigma)
   R_base = UNregularizedrisk_estimate(M, n0, A, H, signal, sigma);
   R = R_base + risk_regularizer(M, A, H, sigma);
end

function L = risk_regularizer(M, A, H, sigma)
   lambda = 12 * sigma.^2;
   L = (lambda / (2 * M + 1) ) * sum( diag( (A * H).^2 ) );
end
```

where `UNregularizedrisk_estimate` refers to the implementation of the base risk function outlined in the previous section.

Risk calculations have been unified under a single wrapper matlab function for ease of use:

```
function R = estimate_risk(type, M, n0, A, H, signal, sigma)
   switch type
       case {'REGULAR','regularized', 'regular'}
           R = risk_regularizer(M, A, H, sigma) +
UNregularized_risk_estimate(M, n0, A, H, signal, sigma);
       otherwise
           R = UNregularized_risk_estimate(M, n0, A, H, signal, sigma);
   end
end
```

Optimal Order Selection

As discussed in the previous sections, optimization of the savitzky golay filter boils down to two essential decisions: (1) selection of the order of the polynomial, (2) selection of the filter length. The baseline/un-extended savitzky-golay filter requires the specification of a single set of parameters for the entire noisy signal. The paper introduces the algorithm presented below to select the optimal order at a point on the noisy signal for a given filter length (John, 2021, p. 5025):

Algorithm 1: Order Selection Algorithm: Calculate Optimum Order p_{opt} at an Instant n_0.

Require: $2M > p_{\max}$
$p \leftarrow p_{\min}$
while $p \leq p_{\max}$ **do**
  Employ least-squares fit over $[(n_0 - M), (n_0 + M)]$
  Evaluate $\hat{\mathcal{R}}$ using (15)
  $p \leftarrow p + 1$
end while
$p_{\text{opt}} = \arg\min_{p} \hat{\mathcal{R}}(p)$

The algorithm iterates over all data instances (indexed by n0) and produces a fit of order p over the window [(n0-M), (n0+M] and evaluates the associated risk. After iterating through the order interval [p_min, p_max], it selects the order with the least risk value as the denoised estimate at index n0. The matlab implementation is provided below:

```
function [R_min, p_opt] = select_opt_order(M, n0, signal, sigma, p_min, p_max,
regularized)

   assert(2*M > p_max); % require
   assert(p_max > p_min);
   Rs = zeros(1, p_max - p_min + 1);
   for p = p_min : p_max
       [A, H] = construct_least_sq_matrices(M, p);
       switch regularized
           case {'regularized', 'REGULAR', 'regular'}
               Rs(p - p_min + 1) = estimate_risk('regular', M, n0+M, A, H,
[zeros(1,M), signal, zeros(1,M)], sigma);
           otherwise
               Rs(p - p_min + 1) = estimate_risk('NONE', M, n0+M, A, H,
[zeros(1,M), signal, zeros(1,M)], sigma);
       end % switch
   end
   [R_min, p_opt] = min(Rs); % p_opt is assigned the index of the min. R val's
order, it is not necessarily the min order
   p_opt = p_opt + p_min - 1; % must add p_min to get the correct order, -1
since arrays start at index 1 in matlab
end
```

Optimal Filter Length Selection

The optimal filter length selection algorithm works in a similar way to the optimal order selection algorithm. For a given order, the algorithm computes the risk associated with every window length value within the specified interval and chooses the one with the lowest risk value. You may find the algorithm from the original article below (John, 2021, 5028):

Algorithm 2: Optimum Filter Length Selection: Calculates the Optimum M_{opt} at an Instant n_0

Require: $2M_{\min} > p$
$M \leftarrow M_{\min}$
while $M \leq M_{\max}$ **do**
  Employ least-squares fit over $[(n_0 - M), (n_0 + M)]$
  Evaluate $\hat{\mathcal{R}}$ using (15)
  $M \leftarrow M + 1$
end while
$M_{\text{opt}} = \arg\min_{M} \hat{\mathcal{R}}(M)$

The corresponding matlab implementation is presented below:

```
% we have to pad the signal within this function since the window length
% changes with every iteration
function [R_min, M_opt] = select_opt_filt_len(p, n0, signal, sigma, M_min,
M_max, regularized)
   assert(2*M_min > p);
   assert(M_max > M_min);

   Rs = zeros(1, M_max - M_min + 1);
   for m = M_min : M_max
       [A, H] = construct_least_sq_matrices(m, p);
       % we pad the signal so that we can calculate the filter lengths at
       % the beginning and the end
       % we need to pass n0+m since we padded the signal
       switch regularized
           case {'regular', 'REGULAR', 'regularized'}
               Rs(m - M_min + 1) = estimate_risk('regular', m, n0+m, A, H,
[zeros(1, m), signal, zeros(1, m)], sigma);
           otherwise
               Rs(m - M_min + 1) = estimate_risk('NONE', m, n0+m, A, H,
[zeros(1, m), signal, zeros(1, m)], sigma);
       end % switch
   end
   [R_min, M_opt] = min(Rs); % M_opt is assigned the index where we ge tthe min
R val. We need to add M_min to get the real
   M_opt = M_opt + M_min - 1;
end
```

Unlike the optimize order function, the noisy signal passed to this function needs to be unpadded. Any necessary padding is done within this function since the window length, which determines the amount of padding necessary, changes at every iteration.

Simultaneous Optimization

Simultaneous optimization of the polynomial order and the filter length could be seen as an extension to the optimization algorithms presented above. Instead of calculating the risk value associated with a single parameter (order or filter length), we calculate the risk value for each unique (order, filter length) pair. The algorithm presented in the paper (p. 5030) could be found below:

Algorithm 3: Order and window-length selection: Determine optimum order p_{opt} and window length M_{opt} at instant n_0.

Require: $2M_{\text{min}} > p_{\text{max}}$
$p \leftarrow p_{\text{min}}$
while $p \leq p_{\text{max}}$ **do**
  $M \leftarrow M_{\text{min}}$
  while $M \leq M_{\text{max}}$ **do**
    Least-squares polynomial fit over $[n_0 - M, n_0 + M]$
    Evaluate $\hat{\mathcal{R}}$ using (15)
    $M \leftarrow M + 1$
  end while
  $p \leftarrow p + 1$
end while
$p_{\text{opt}}, M_{\text{opt}} = \arg\min_{p,M} \hat{\mathcal{R}}(p, M)$

The matlab implementation is as follows:

```
function [R_min, p_opt, M_opt] = simult_optim(n0, signal, sigma, p_min, p_max,
M_min, M_max, regularized)
   assert(2*M_min > p_max);
   assert(M_max > M_min);
   assert(p_max > p_min);

   Rs = zeros(p_max - p_min + 1, M_max - M_min +1);
   for p = p_min : p_max
       for m = M_min : M_max
           [A, H] = construct_least_sq_matrices(m, p);
           switch regularized
               case {'regularized', 'REGULAR', 'regular'}
                   Rs(p - p_min + 1, m - M_min + 1) = estimate_risk('regular',
m, n0+m, A, H, [zeros(1,m), signal, zeros(1,m)], sigma);
               otherwise
                   Rs(p - p_min + 1, m - M_min + 1) = estimate_risk('NONE', m,
n0+m, A, H, [zeros(1,m), signal, zeros(1,m)], sigma);
           end
       end % for: M
   end % for: p
   [R_min, min_flat_idx] = min( Rs(:) ); % this get shte flattened vector's
index
   [p_opt, M_opt] = ind2sub(size(Rs), min_flat_idx); % we convert it to matrix
index
   p_opt = p_opt + p_min -1;
   M_opt = M_opt + M_min -1;
end
```

Configurations Used in the Article

The article presents the following optimization strategies/configurations (John, 2021, p. 5028)

1. G-O: optimize the order of the fit polynomial using the base risk variant
2. G-O-R: optimize the order of the fit polynomial using the regularized risk variant
3. G-FL: optimize the filter length using the base risk variant
4. G-FL-R: optimize the filter length using the regularized risk variant
5. G-FL-O: simultaneously optimize the filter length and the order of the fit polynomial only using the base risk variant
6. G-FL-O-R: simultaneously optimize the filter length and the order of the fit polynomial only using the regularized risk variant

Matlab functions corresponding to the configurations above are presented below:

```
function [G_O_filtered_signal, G_O_orders] = GO(m, noisy_signal, p_min, p_max,
sigma)
   G_O_filtered_signal = zeros(1, length(noisy_signal));
   G_O_orders = zeros(1, length(noisy_signal));
   for n0 =  [1:length(noisy_signal)] % [m+1:length(noisy_signal)-m]
       [~, p] = select_opt_order(m, n0, noisy_signal, sigma, p_min, p_max,
'NONE');
       G_O_orders(n0) = p;
       [~, H] = construct_least_sq_matrices(m, p);
       G_O_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
   end
end

function [G_O_R_filtered_signal, G_O_R_orders] = GOR(m, noisy_signal, p_min,
p_max, sigma)
   G_O_R_filtered_signal = zeros(1, length(noisy_signal));
   G_O_R_orders = zeros(1, length(noisy_signal));

   for n0 = [1:length(noisy_signal)] % [m+1:length(noisy_signal)-m]
       [~, p] = select_opt_order(m, n0, noisy_signal, sigma, p_min, p_max,
'regular');
       G_O_R_orders(n0) = p;
       [~, H] = construct_least_sq_matrices(m, p);
```

```
        G_O_R_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
    end
end % function

function [G_FL_filtered_signal, G_FL_filter_lens] = GFL(p, noisy_signal, M_min,
M_max, sigma)
    G_FL_filtered_signal = zeros(1, length(noisy_signal));
    G_FL_filter_lens = zeros(1, length(noisy_signal));
    for n0 = [1:length(noisy_signal)]
        [~, m] = select_opt_filt_len(p, n0, noisy_signal, sigma, M_min, M_max,
'NONE');
        G_FL_filter_lens(n0) = m;
        [~, H] = construct_least_sq_matrices(m, p);
        G_FL_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
    end
end % function

function [G_FL_R_filtered_signal, G_FL_R_filter_lens] = GFLR(p, noisy_signal,
M_min, M_max, sigma)
    G_FL_R_filtered_signal = zeros(1, length(noisy_signal));
    G_FL_R_filter_lens = zeros(1, length(noisy_signal));
    for n0 = [1:length(noisy_signal)]
        [~, m] = select_opt_filt_len(p, n0, noisy_signal, sigma, M_min, M_max,
'regular');
        G_FL_R_filter_lens(n0) = m;
        [~, H] = construct_least_sq_matrices(m, p);
        G_FL_R_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
    end
end % function

function [G_FL_O_filtered_signal, G_FL_O_orders, G_FL_O_lengths] =
GFLO(noisy_signal, p_min, p_max, M_min, M_max, sigma)
    G_FL_O_filtered_signal = zeros(1, length(noisy_signal));
    G_FL_O_orders  = zeros(1, length(noisy_signal));
    G_FL_O_lengths = zeros(1, length(noisy_signal));
```

```
   for n0 = [1:length(noisy_signal)]
       [~, p, m] = simult_optim(n0, noisy_signal, sigma, p_min, p_max, M_min,
M_max, 'NONE');
       [~, H] = construct_least_sq_matrices(m, p);
       G_FL_O_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
       G_FL_O_lengths(n0) = m;
       G_FL_O_orders(n0) = p;
   end
end % function

function [G_FL_O_R_filtered_signal, G_FL_O_R_orders, G_FL_O_R_lengths] =
GFLOR(noisy_signal, p_min, p_max, M_min, M_max, sigma)
   G_FL_O_R_filtered_signal = zeros(1, length(noisy_signal));
   G_FL_O_R_orders  = zeros(1, length(noisy_signal));
   G_FL_O_R_lengths = zeros(1, length(noisy_signal));

   for n0 = [1:length(noisy_signal)]
       [~, p, m] = simult_optim(n0, noisy_signal, sigma, p_min, p_max, M_min,
M_max, 'regular');
       [~, H] = construct_least_sq_matrices(m, p);
       G_FL_O_R_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
       G_FL_O_R_lengths(n0) = m;
       G_FL_O_R_orders(n0) = p;
   end
end % function
```

Omitted Parts of the Article

The paper introduces two optimizations for the order and filter length selection algorithms: (1) combining neighbourhood estimates (John, 2021, p. 5030) and (2) SG filtering for correlated noise (p. 5031). (2) involves an optimization to the regularizer and was found to be beneficial “to an appreciable extent” (p. 5031) (the article is vague in terms of under which conditions the correlation regularizer performs well) for a set of signals with certain properties such as correlated noise, and rely heavily on the algorithms presented above (p. 5031). (1) utilizes a weighted combination of the past M and future M sample points in an attempt to improve smoothing (p. 5030). (1) was found to be beneficial in the low SNR (high noise power per signal

power) range but performed subpar in the high SNR range (p. 5030). Hence these sections were skipped in the implementation.

Results and Evaluation

The authors tested their algorithms on signals featuring consecutive portions of polynomials of different orders (termed ‘piecewise regular’) in addition to several ECG signals taken from a database. I decided to test mainly on piecewise regular signals since I had more control over their properties which in turn enabled me to test a variety of test cases.

Test Case #1

The first test case contains polynomial segments of orders [0,7,1,6,2,5,3,4]. The noise type used is gaussian with a signal-to-noise ratio of 12.

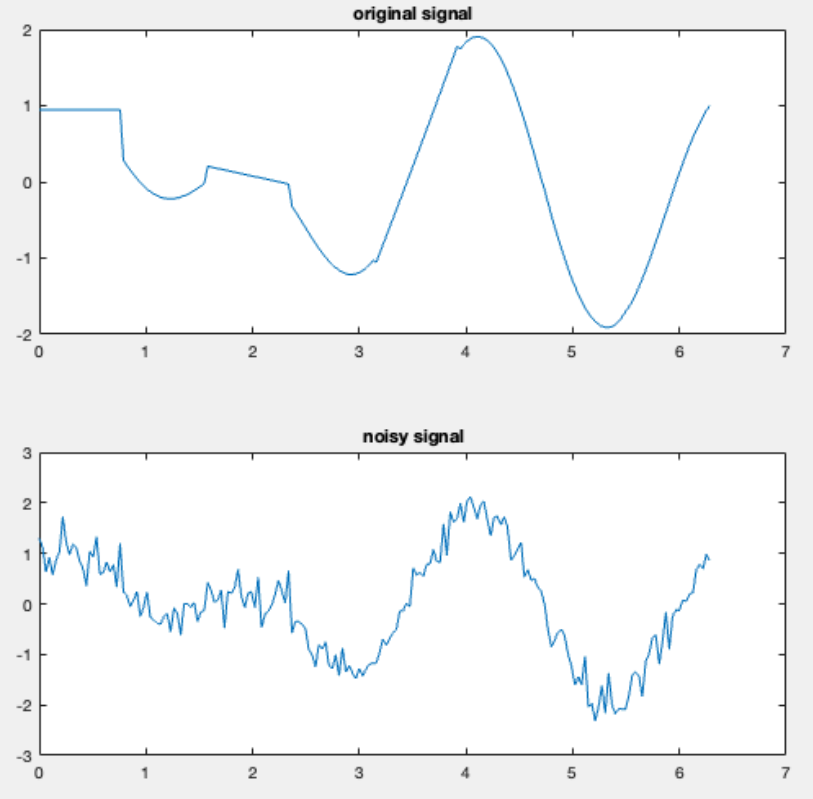

Fig. 1: Original and noisy signals. Gaussian noise was used.

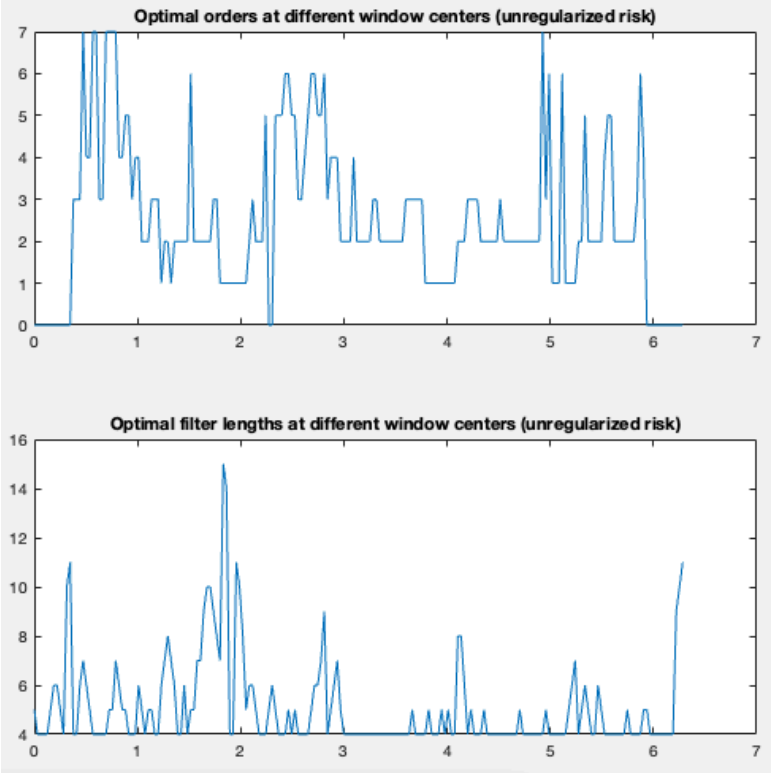

Fig. 2: Optimal orders and filter lengths at each datapoint of the signal for M = 12 and p = 3.

Comparing figures 2 and 1, we see that sudden transitions cause surges in both optimal order and filter lengths. This is natural since higher order polynomials are more able to adapt to sudden changes.

Provided below are different configurations' (G-O(-R), G-FL(-R), G-FL-O(-R)) fits for the test case alongside calculated values of order and/or filter length.

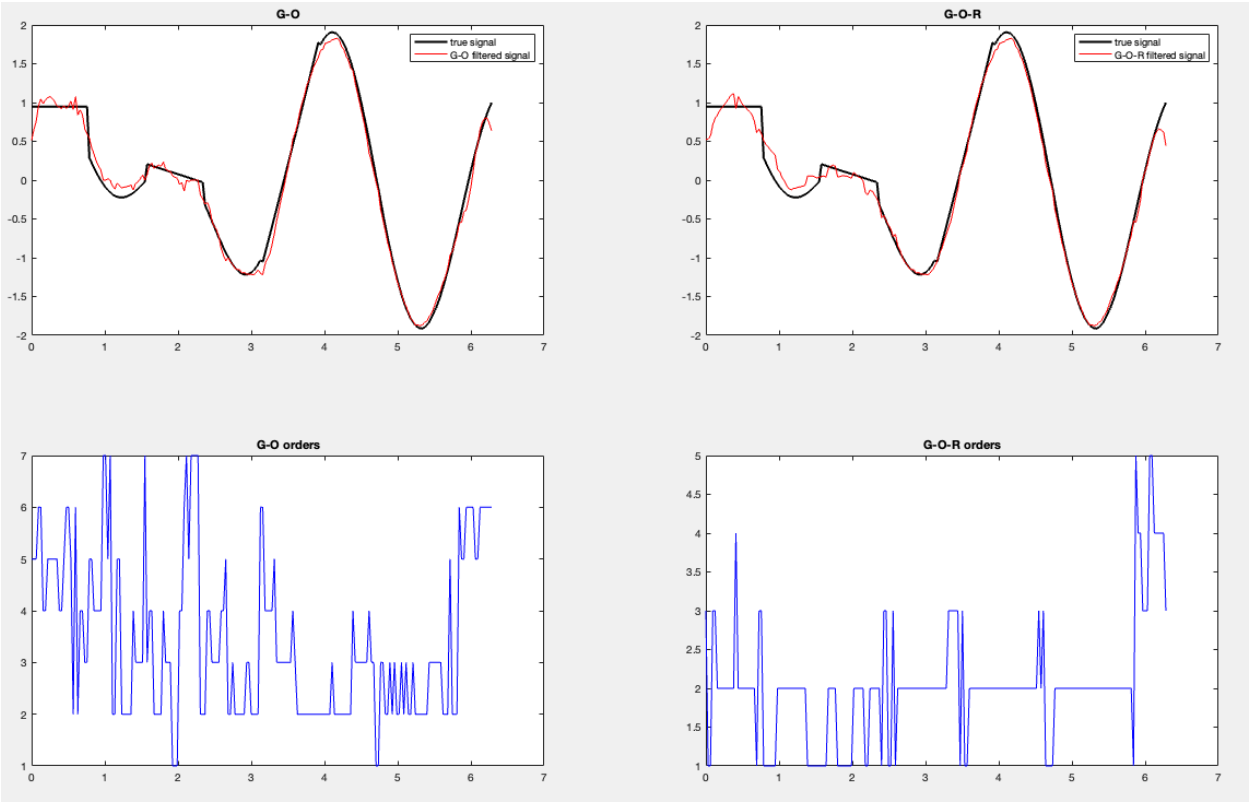

Fig. 3: G-O and G-O-R's fit with order plots at each datapoint. Window length has been fixed at 15 and (p_min, p_max) = (1, 7). X axis denotes the data point index/time.

Fig. 3 reveals that the regularization function succeeds in preventing sudden changes in the fitted signal. Comparing G-O and G-O-R's fits between 0 and 1, we see that G-O-R converged to the y value of 1 more slowly compared to G-O. Similarly the orders in G-O-R tend to change by smaller increments compared to the highly volatile G-O. As discussed in the previous sections the regularizer penalizes sudden changes and spreads them across multiple time instances. Hence values closer to the previous order value are more likely to result in the lowest risk function value among all possible order values.

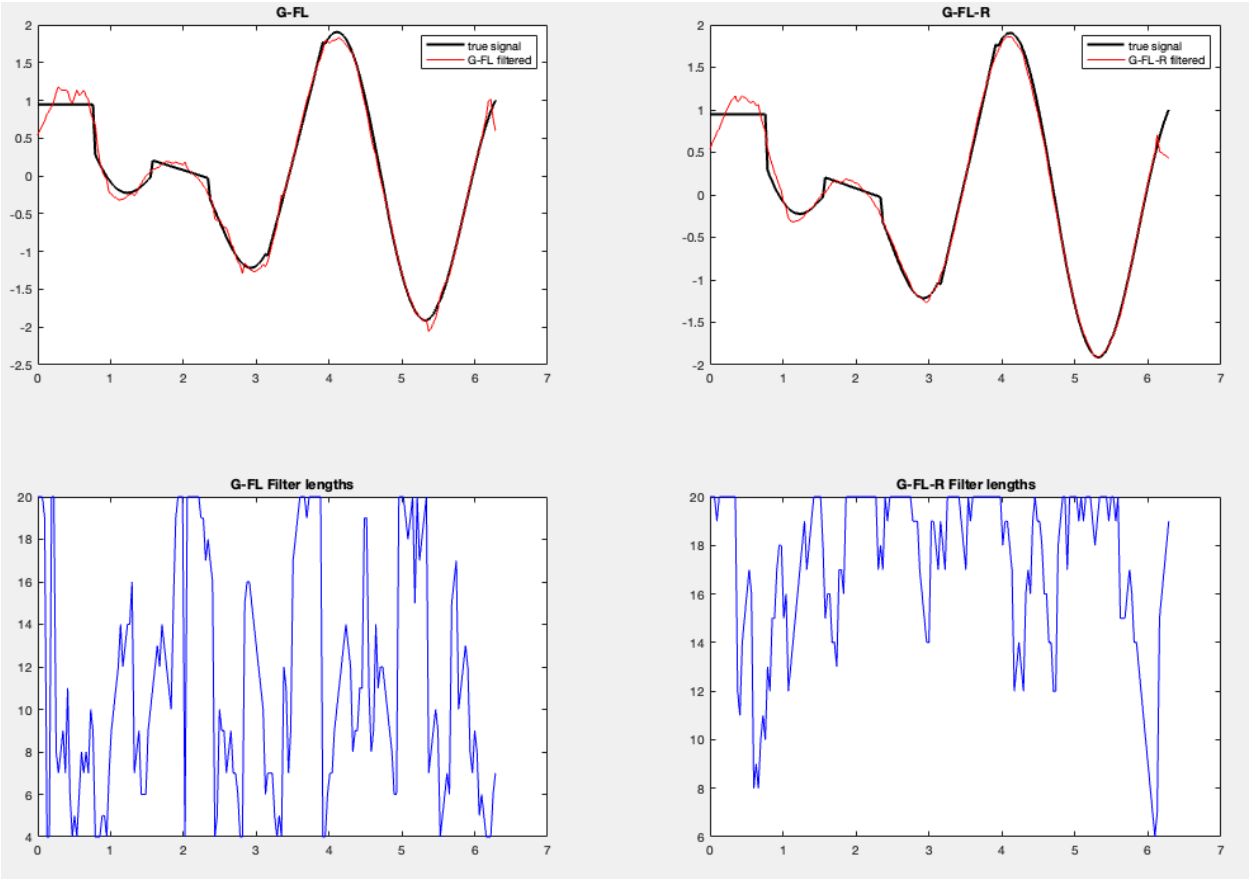

Fig. 4: G-FLand G-FL-R's fit for test case 1. X axis denotes the indexes of data values/time. The order of the polynomial has been fixed at 3 while (M_min, M_max) was specified as (4, 20).

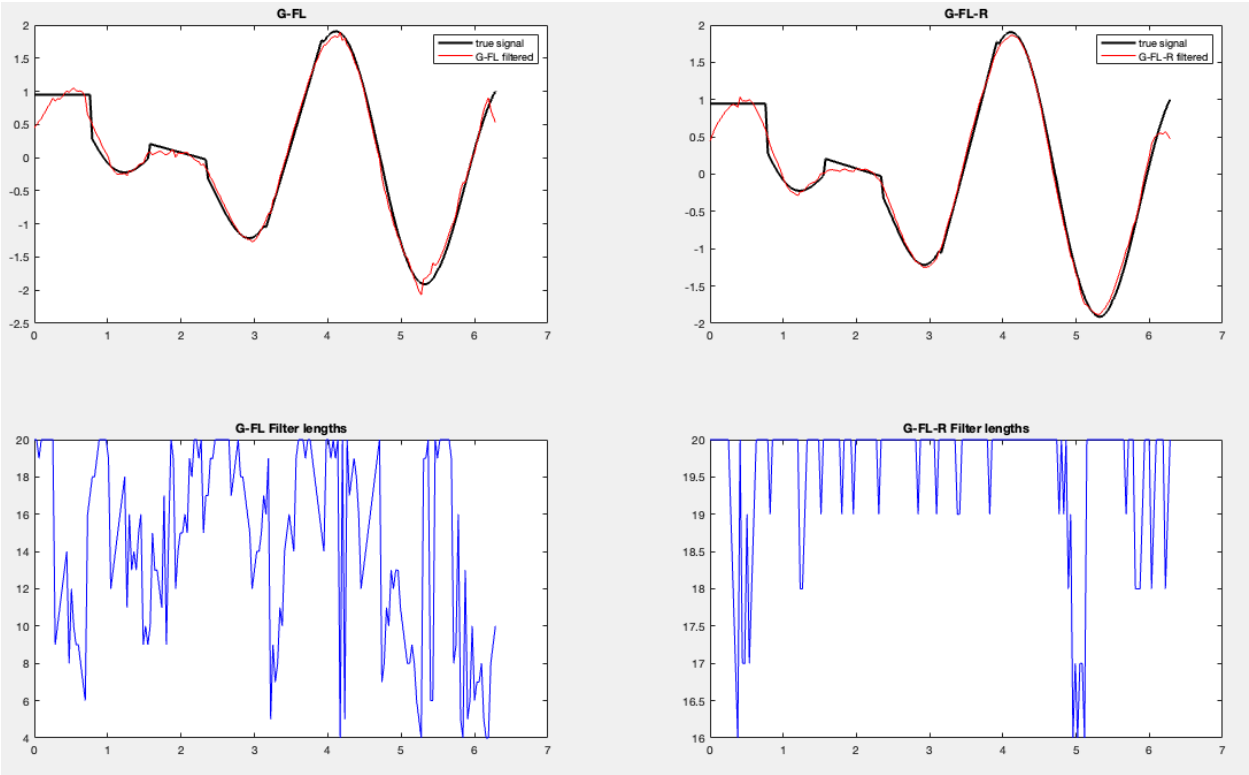

Fig. 5: G-FLand G-FL-R's fit for test case 1 with order fixed at p=5. The rest of the parameters have not been changed.

Similar to Fig. 3, we see that the regularized function is less prone to large filter length jumps. Fig. 5's order demonstrates a better fit compared to Fig. 4's. Another observation is that a more suitable order selection resulted in more stability in the filter lengths selected.

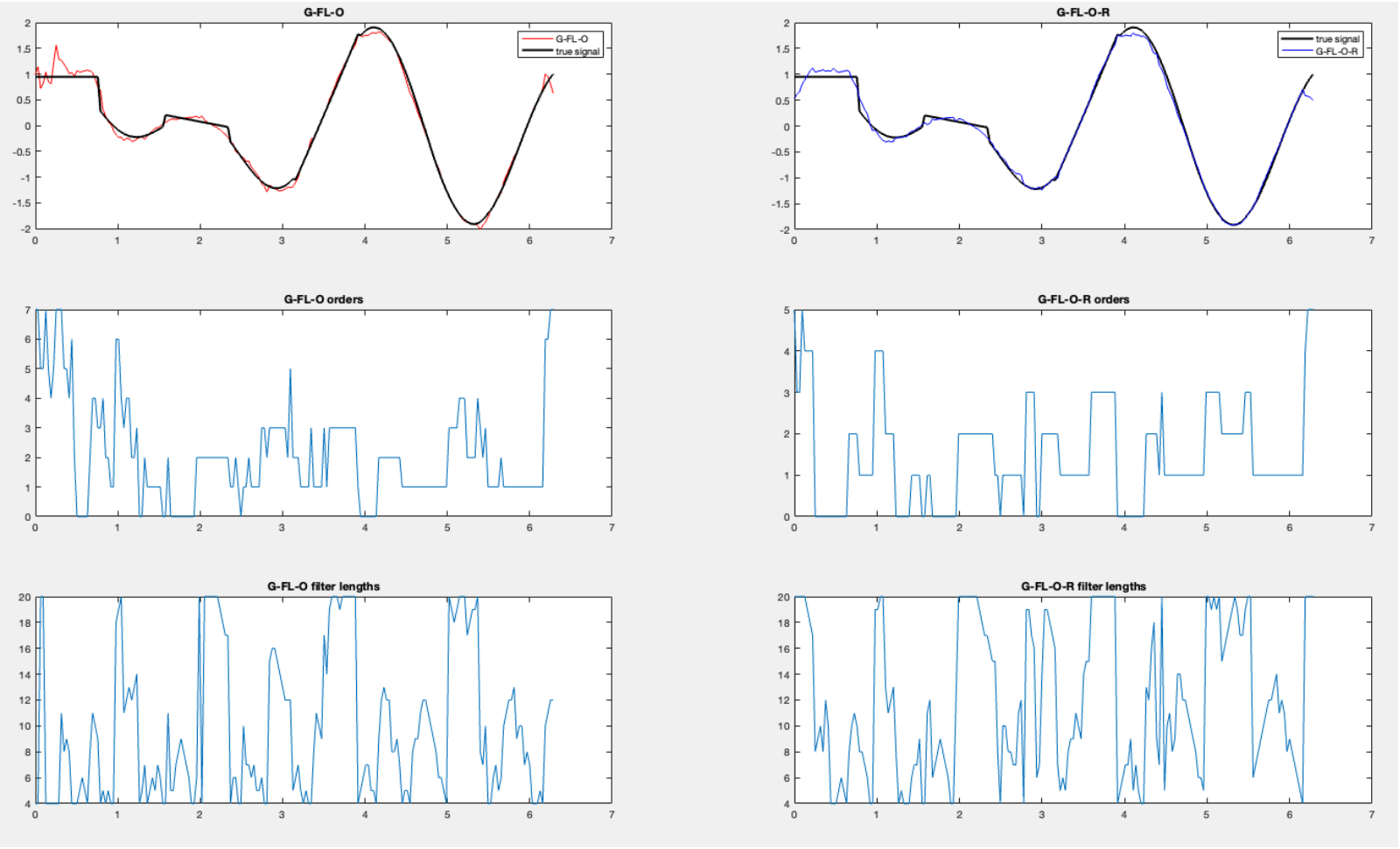

Fig. 6: G-FL-O and G-FL-O-R's fits for test case 1.

Visually inspecting the plots, the most accurate results have been those of G-FL-O-R's. This is not surprising since the algorithm optimizes both the window length and the polynomial order at the same time which allows for more flexibility.

Provided below are MSE calculations between different configurations' fits and the original signal. The outputs have been obtained from the project matlab script and will be printed to the command window:

- MSE of G-FL-O: 0.012634
- MSE of G-FL-O-R: 0.015934 (p = 5)
- MSE of G-FL: 0.014426
- MSE of G-FL-R: 0.013119
- MSE of G-O: 0.013154
- MSE of G-O-R: 0.016791

Overall, the lowest MSE has been produced by the G-FL-O algorithm. One possible explanation to why its regularized variant was less accurate is that the signal contains sudden changes (around marks t=1, 1.5, 2.5) to which the regularized version cannot respond quickly to.

As a baseline I've selected the regular savitzky-golay filter. The matlab implementation of the base savitzky-golay filter has been taken from Mathworks forums (Mathworks, 2018) and modified. The fit and MSE produced by base SG are provided below:

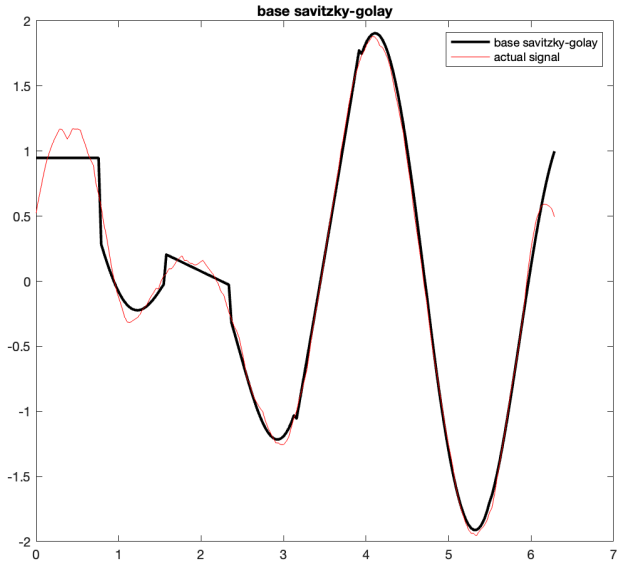

Fig. 7: Base SG fit for the noisy signal.

The MSE of the base SG filtered signal was measured as 0.013420. Among all configurations, G-FL-O-R, G-O-R, G-FL have failed to outperform baseline SG.

Test Case #2

Test case 2 features a piecewise regular signal with orders [0,1,2,3,4,5,6,7]. The noise type is gaussian with an snr value of 12. I decided to discuss issues related to noise type in the sections to follow and keep the noise type constant to see what effect the sequencing of polynomial orders would have on the filtering results. Most of the observations made for test case #1 also apply to test case #2, and have been omitted.

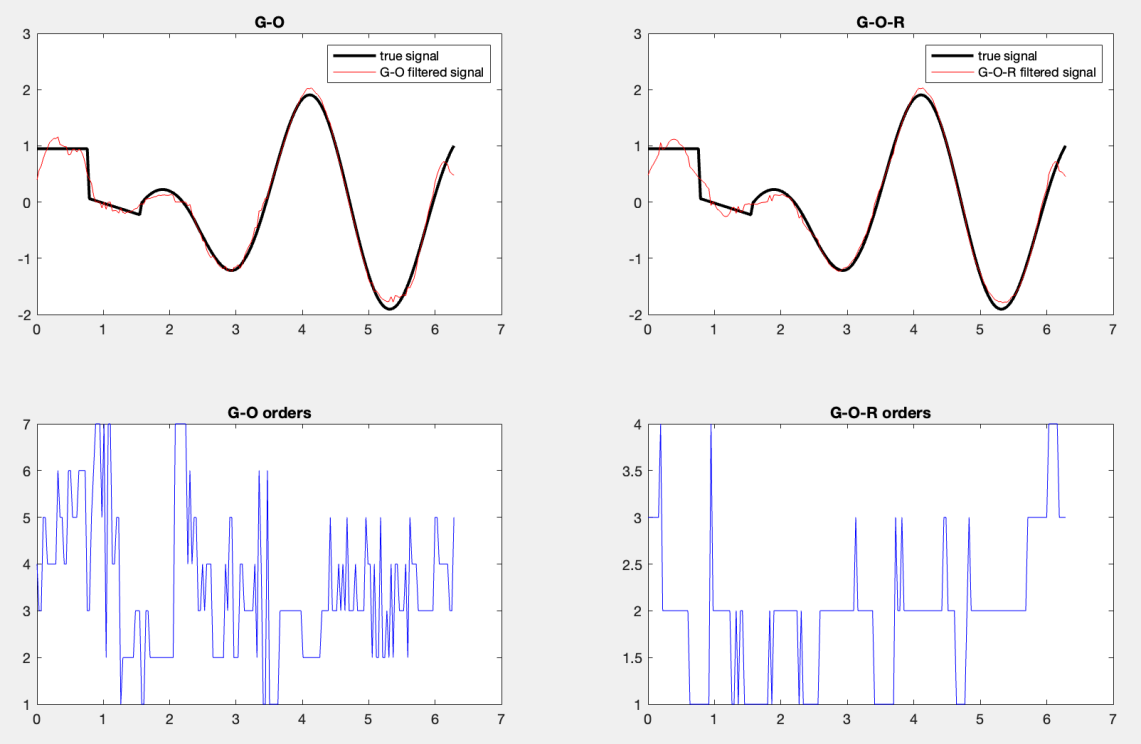

Fig. 8: G-O fits for test case 2. M=15 and (p_min, p_max) = (1, 7)

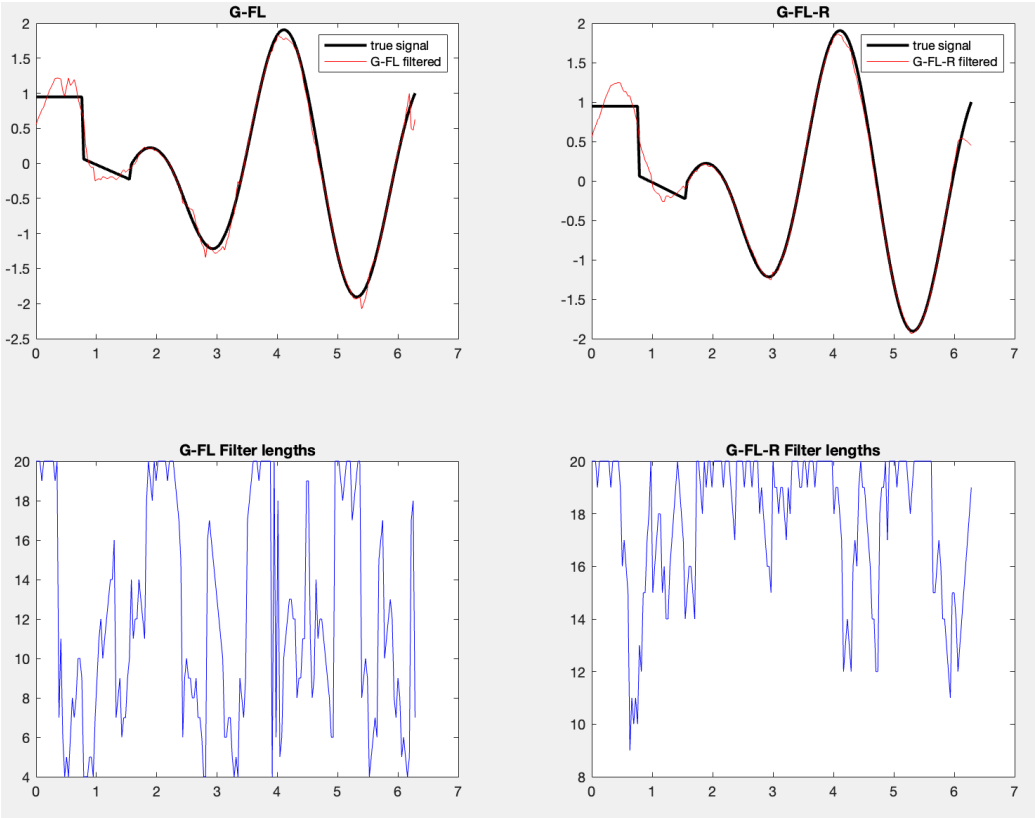

Fig. 9: G-FL fits for test case 2. p=3 and (M_min, M_max) = (4, 20)

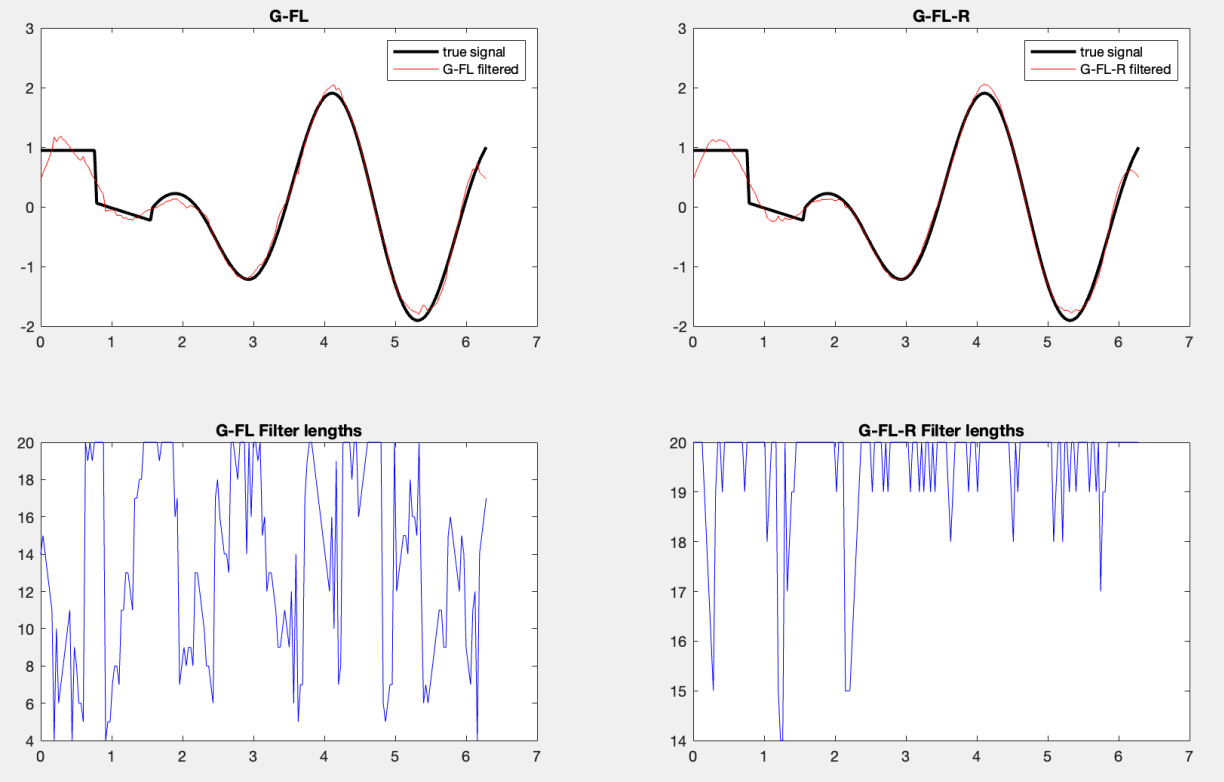

Fig. 10: G-FL fits for test case 2. p=5 and (M_min, M_max) = (4, 20)

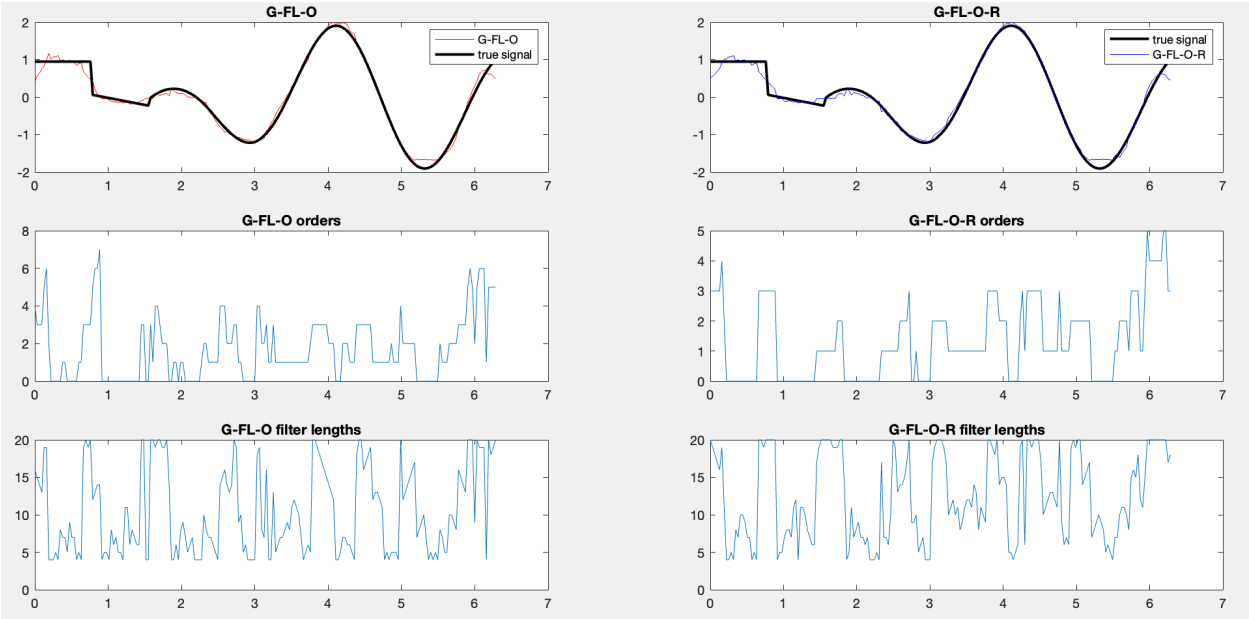

Fig. 11: G-FL-O fits for test case 2

MSE Calculations for test case #2:

- MSE of G-FL-O: 0.016740
- MSE of G-FL-O-R: 0.013357
- MSE of G-FL: 0.015525 (for p = 5)
- MSE of G-FL-R: 0.016040
- MSE of G-O: 0.013599
- MSE of G-O-R: 0.016531

Base SG filter MSE: 0.015435

The best configuration among all was G-FL-O-R with an MSE of 0.013357. The base SG fit is provided below:

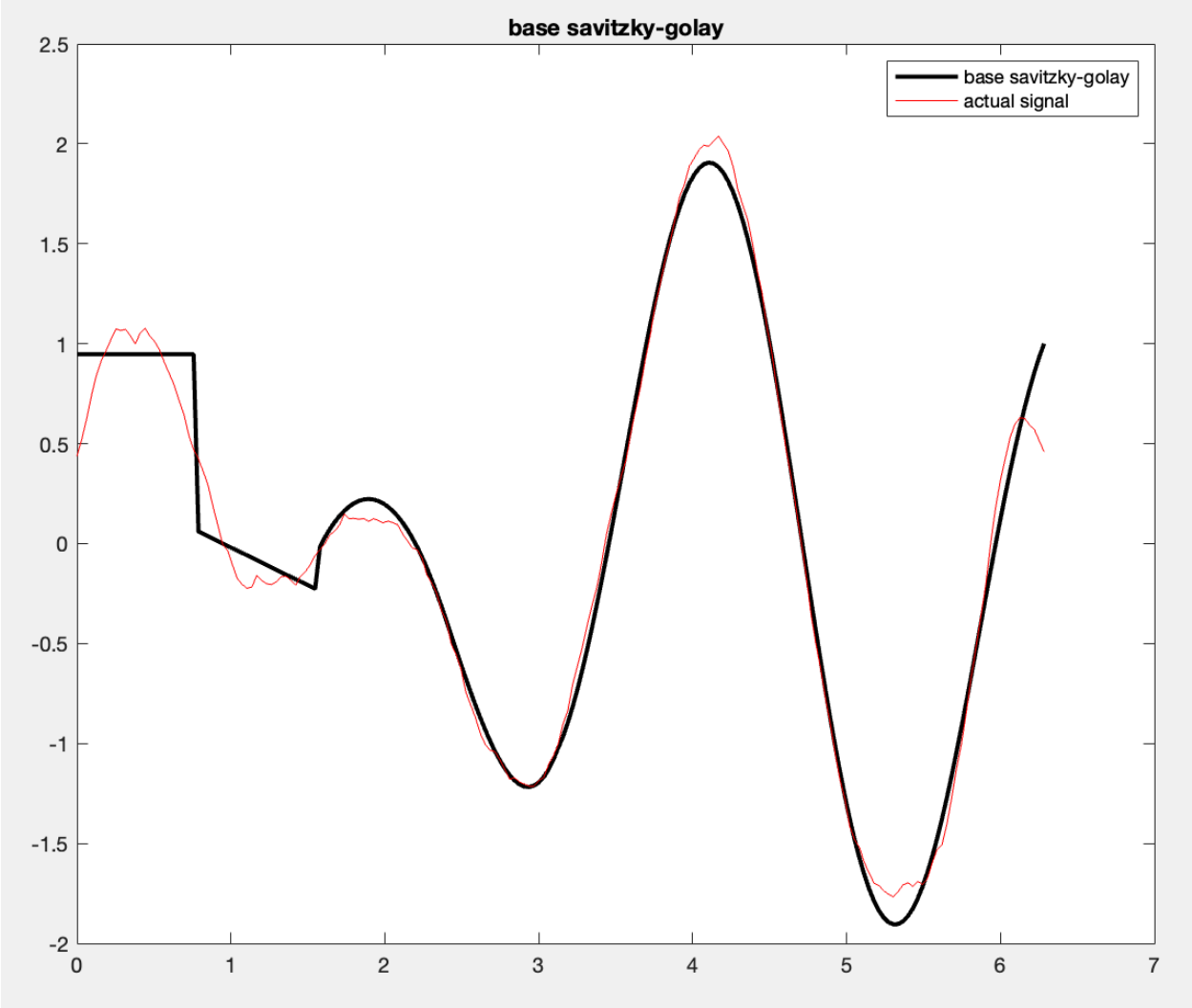

Fig. 12: Base Savitzky-Golay fit to the 2nd test case

Output SNR vs Input SNR - Denoising Performance

The main performance measure for the project was selected as the SNR of the filtered signal vs the SNR of the noisy input signal.

The final part of the project produces noise of all types for snr values of -5,0,5,..,25 and measures the snr values of the output filter for all configurations. Provided below is the SNR testing and plotting loop:

```
noise_types = ["un", "gs", "lp"];
noise_type_full_name = ["uniform", "gaussian", "laplace"];
for noise_type = noise_types
   fprintf("\nDiffernt SNRs testing - noise type: %s\n", noise_type);

   input_snr_space = [-5:5:25];
   input_noise_space = zeros(length(input_snr_space), length(true_signal));
   noisy_signal_space = zeros(length(input_snr_space), length(true_signal));

   GO_output_snr_space = zeros(1, length(input_snr_space));
   GOR_output_snr_space = zeros(1, length(input_snr_space));

   GFL_output_snr_space = zeros(1, length(input_snr_space));
   GFLR_output_snr_space = zeros(1, length(input_snr_space));

   GFLO_output_snr_space = zeros(1, length(input_snr_space));
   GFLOR_output_snr_space = zeros(1, length(input_snr_space));

   for snr_i = 1:length(input_snr_space)
       [noise, sigma] = gen_noise(noise_type, true_signal,
input_snr_space(snr_i));
       input_noise_space(snr_i, :) = noise;
       noisy_signal_space(snr_i, :) = input_noise_space(snr_i, :) +
true_signal;

   end

   for snr_i = 1:length(input_snr_space)
       noisy_signal = noisy_signal_space(snr_i, :);

       [GO_out, ~] = GO(m, noisy_signal, p_min, p_max, sigma);
       [GOR_out, ~] = GOR(m, noisy_signal, p_min, p_max, sigma);
       GO_output_snr_space(snr_i) = get_snr(true_signal, GO_out);
       GOR_output_snr_space(snr_i) = get_snr(true_signal, GOR_out);

       [GFL_out, ~] = GFL(p, noisy_signal, M_min, M_max, sigma);
       [GFLR_out, ~] = GFLR(p, noisy_signal, M_min, M_max, sigma);
       GFL_output_snr_space(snr_i) = get_snr(true_signal, GFL_out);
       GFLR_output_snr_space(snr_i) = get_snr(true_signal, GFLR_out);
```

```
        [GFLO_out, ~, ~] = GFLO(noisy_signal, p_min, p_max, M_min, M_max,
sigma);
        [GFLOR_out, ~, ~] = GFLOR(noisy_signal, p_min, p_max, M_min, M_max,
sigma);
        GFLO_output_snr_space(snr_i) = get_snr(true_signal, GFLO_out);
        GFLOR_output_snr_space(snr_i) = get_snr(true_signal, GFLOR_out);
    end

    figure;
    plot(input_snr_space, GO_output_snr_space, "r-o"); hold on;
    plot(input_snr_space, GOR_output_snr_space, "g-s"); hold on;
    plot(input_snr_space, GFL_output_snr_space, "b-x"); hold on;
    plot(input_snr_space, GFLR_output_snr_space, "k-*"); hold on;
    plot(input_snr_space, GFLO_output_snr_space, "m-^"); hold on;
    plot(input_snr_space, GFLOR_output_snr_space, "c-d");

    legend("GO", "GOR", "GFL", "GFLR", "GFLO", "GFLOR");
    title(sprintf("output snr vs input snr for %s noise", noise_type_full_name(
find(noise_types == noise_type)) ));
    xlabel("input snr (db)");
    ylabel("output snr (db)");
end % for noise type
```

Test Case #1

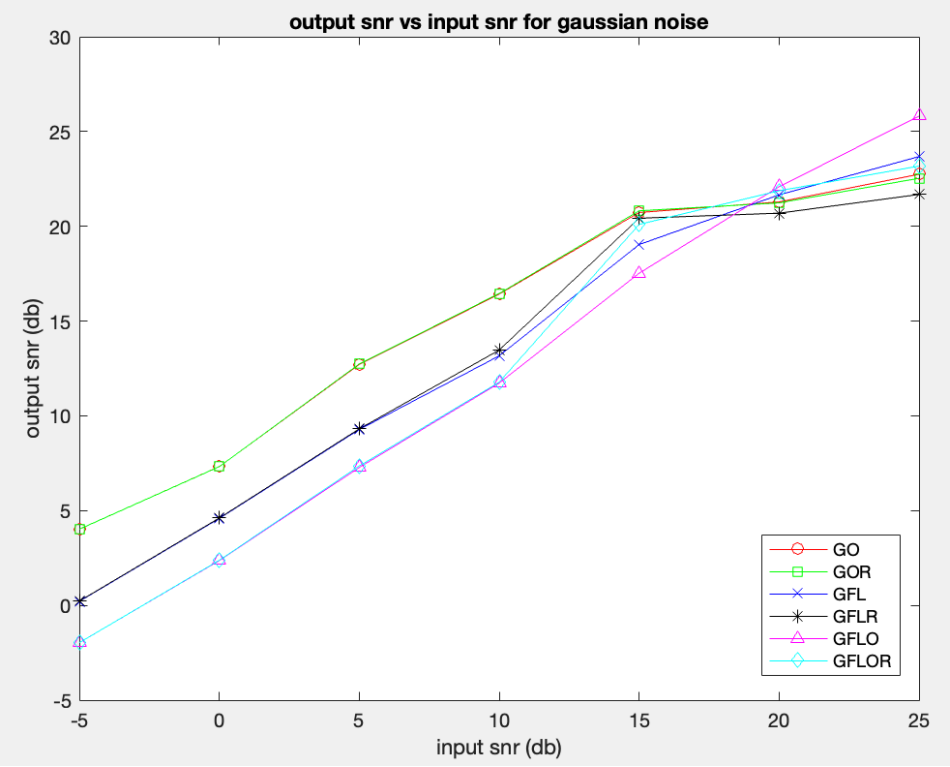

Fig. 13: Output SNR of the filtered signal per SNR of the input signal for gaussian noise

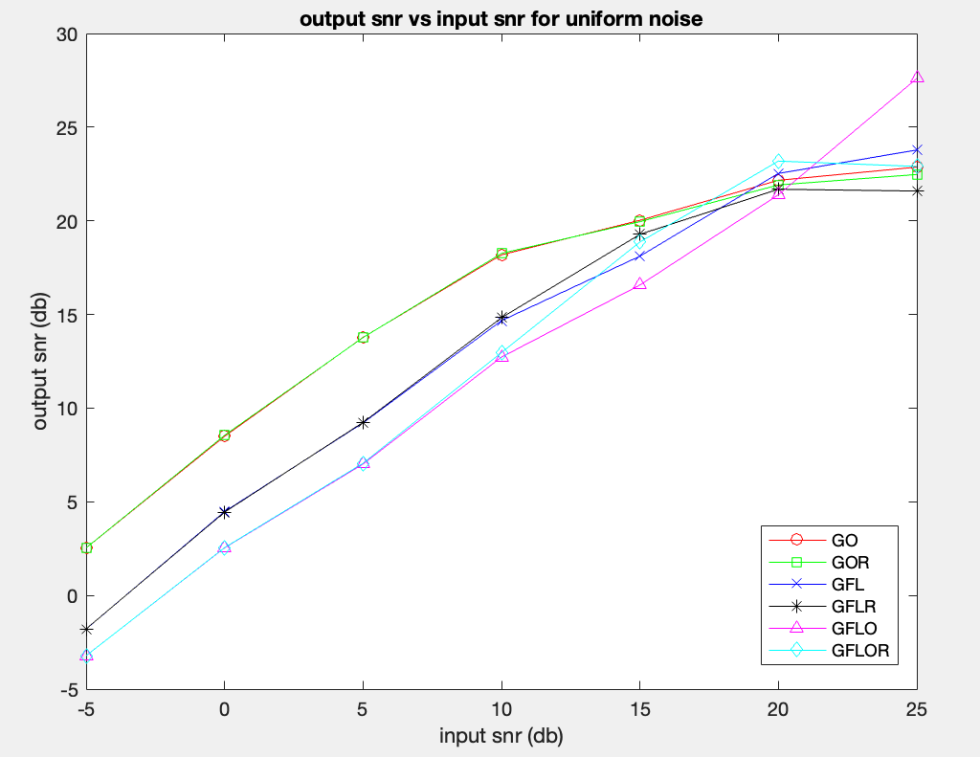

Fig. 14: Output SNR of the filtered signal per SNR of the input signal for uniform noise

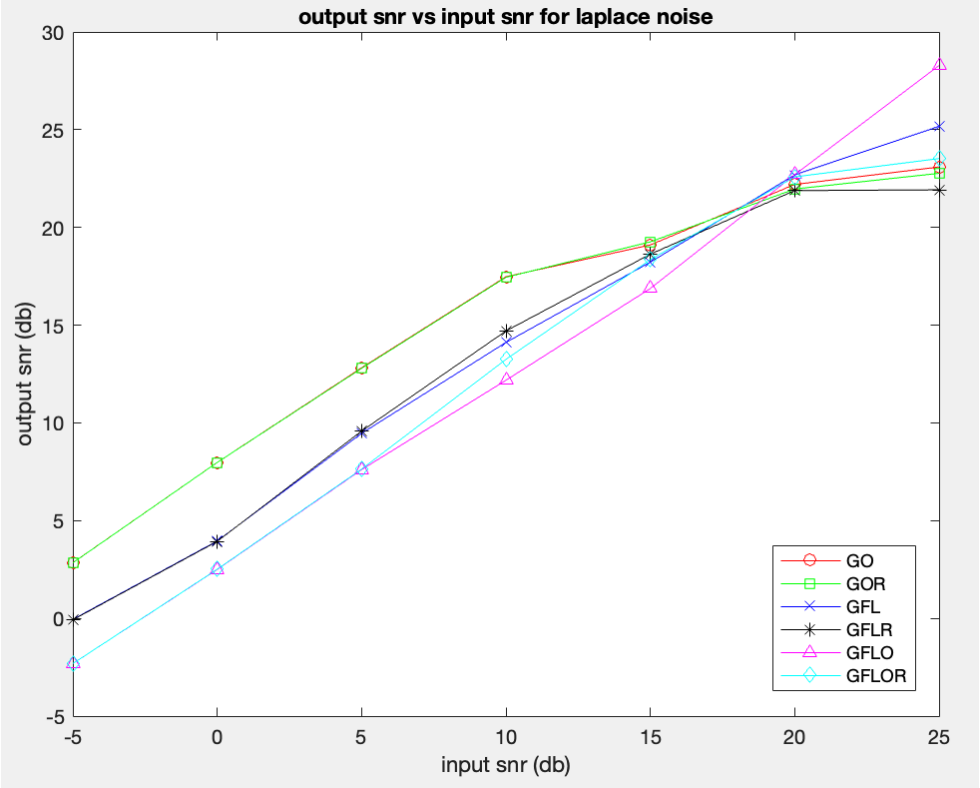

Fig. 15: Output SNR of the filtered signal per SNR of the input signal for laplace noise

The outputs above were generated for the base signal in test case #1. Observing the data we see that GOR and GO (green and red in the plots) perform best for lower SNR values (more noise power per signal power) while GFLO performs best for cleaner signals. These results support those obtained in the article for piecewise regular signals (John, 2021, p. 5028).

It is interesting to note that all regularized configurations' performance was worse compared to their base variants for higher SNR values. Regularization was a measure against sudden changes in the signal's values. For high SNR values (less noise power per signal power) sudden changes start representing the underlying signal more than they represent noise. Hence penalizing sudden changes hampers the accuracy of the produced fit.

Test Case #2

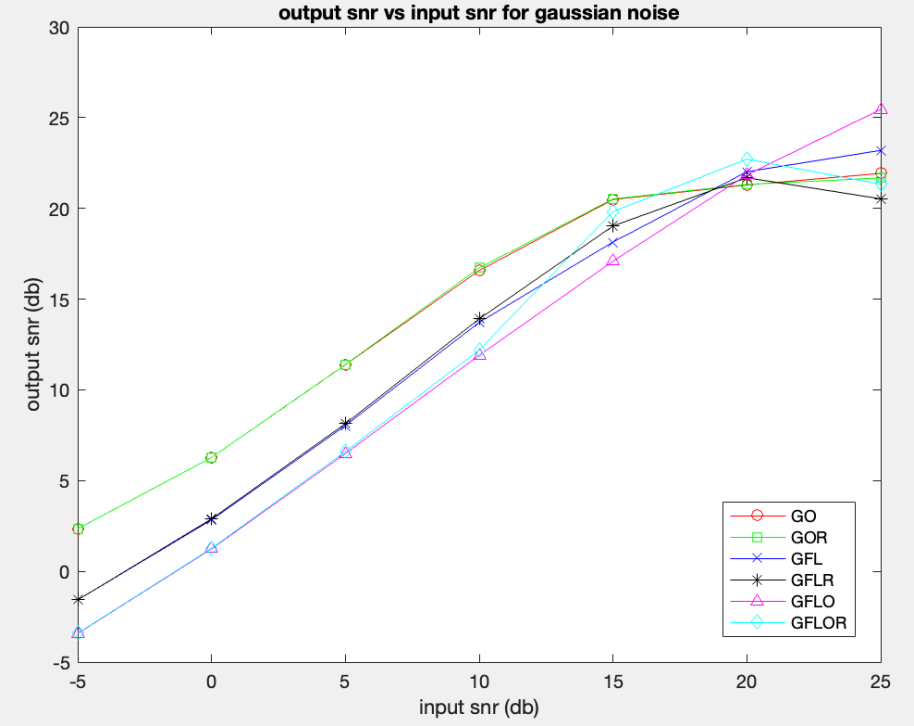

Fig. 16: Output SNR of the filtered signal per SNR of the input signal for gaussian noise

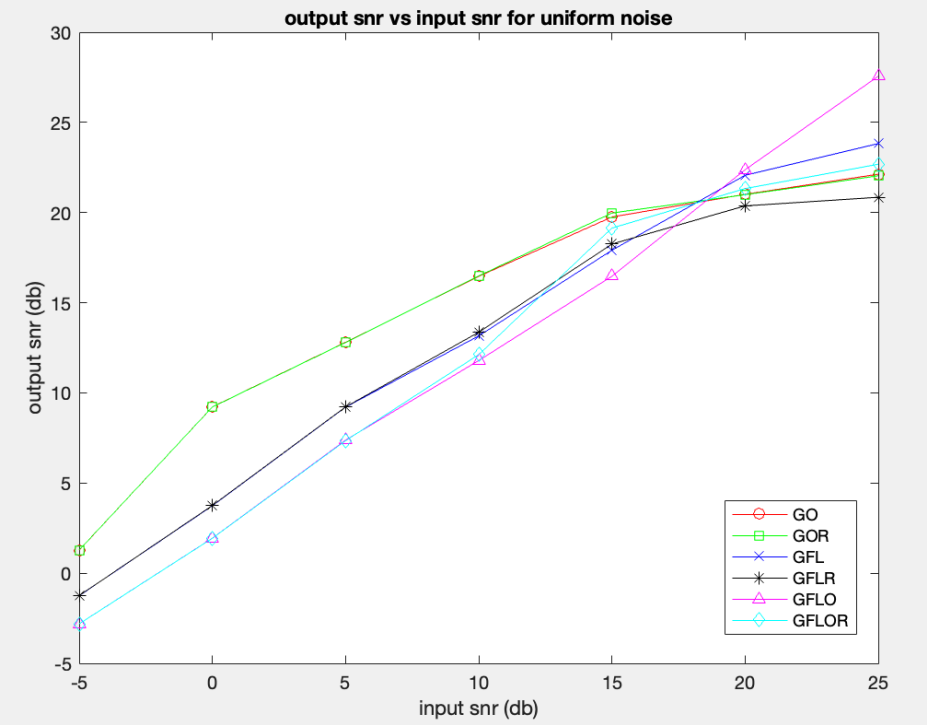

Fig. 17: Output SNR of the filtered signal per SNR of the input signal for uniform noise

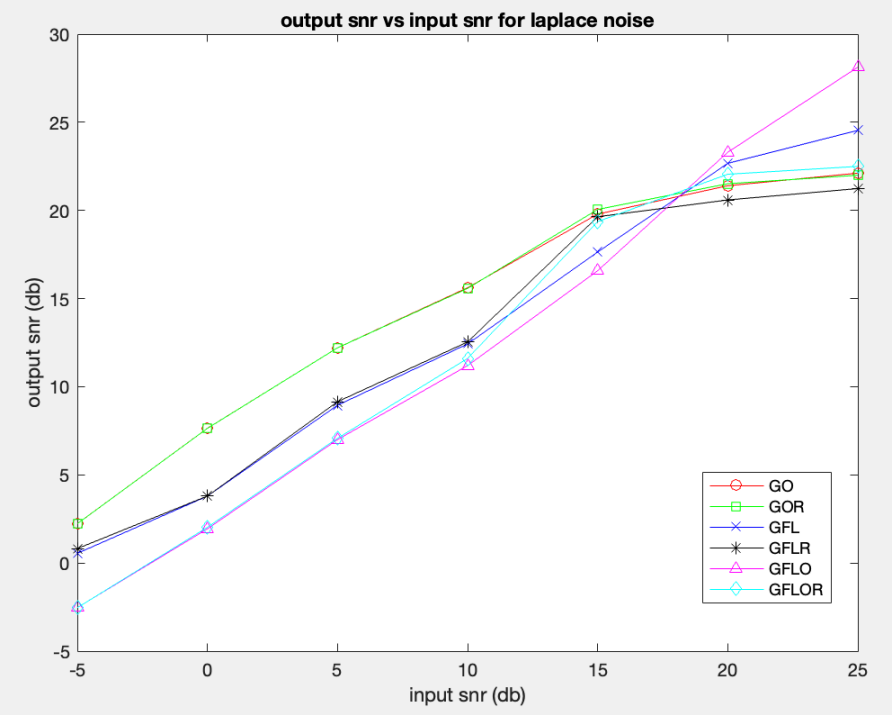

Fig. 18: Output SNR of the filtered signal per SNR of the input signal for laplace noise

The observations made for test case #1 also apply to test case #2's results. An additional observation could be made about the differences in regularized and base risk function variants

of the configurations starting to arise between snr values of 10 and 15. While not highlighting this explicitly, the article defined 12.5 db as “medium input SNR” (John, 2021, p. 5034) which supports this observation.

References

How to Implement the Savitzky-Golay Filter Without Using Built-in Functions. (2018, May 28). Mathworks.com. https://www.mathworks.com/matlabcentral/answers/335433-how-to-implement-savitzky-golay-filter-without-using-inbuilt-functions

John, A., Sadasivan, J., & Seelamantula, C. S. (2021). Adaptive Savitzky-Golay Filtering in Non-Gaussian Noise. IEEE Transactions on Signal Processing, 69, 5021–5036. doi:10.1109/TSP.2021.3106450

Appendix A. Project Matlab File Contents

```
%% clear
clear all;
%% paper's algo's implementation
% A and H come from least squares regression
% this is formula 15 from the paper
% GUE-MSE (general unbiased estimate of mean squared error)
% !!!! n0 has to be the center of the window
function R = UNregularized_risk_estimate(M, n0, A, H, signal, sigma)
   % fprintf("dims A: %dx%d ", size(A,1), size(A,2));
   % fprintf("dims H: %dx%d ", size(H,1), size(H,2));
   % fprintf("dims signal: %dx%d\n", size(signal,1), size(signal,2));
   sigma_sq = sigma * sigma;
   wind_size = 2*M + 1;
   % fprintf("n0: %d, M: %d\n", n0, M);
   x = signal(n0-M:n0+M)'; % signal window transposed
   term1 = norm(A * H * x)^2;
   term2 = -2 * x' * H' * A' * x;
   term3 = 2 * sigma_sq * sum( diag(H' * A') );
   term4 = norm(x)^2;
   R = (1/wind_size) * (term1 + term2 + term3 + term4) - sigma_sq;
end
function R = regularized_risk_estimate(M, n0, A, H, signal, sigma)
   R_base = UNregularizedrisk_estimate(M, n0, A, H, signal, sigma);
   R = R_base + risk_regularizer(M, A, H, sigma);
end
function L = risk_regularizer(M, A, H, sigma)
   lambda = 12 * sigma.^2;
   L = (lambda / (2 * M + 1) ) * sum( diag( (A * H).^2 ) );
end
function R = estimate_risk(type, M, n0, A, H, signal, sigma)
   switch type
       case {'REGULAR','regularized', 'regular'}
           R = risk_regularizer(M, A, H, sigma) +
UNregularized_risk_estimate(M, n0, A, H, signal, sigma);
       otherwise
           R = UNregularized_risk_estimate(M, n0, A, H, signal, sigma);
   end
end
function [R_min, p_opt] = select_opt_order(M, n0, signal, sigma, p_min, p_max,
regularized)
   % todo: use the arguments or whatever thing matlab has for param
   % checking
   assert(2*M > p_max); % require
   assert(p_max > p_min);
   Rs = zeros(1, p_max - p_min + 1);
   % disp(Rs)
   % p = p_min;
```

```
   for p = p_min : p_max
       % disp(p);
       [A, H] = construct_least_sq_matrices(M, p);
       switch regularized
           case {'regularized', 'REGULAR', 'regular'}
               Rs(p - p_min + 1) = estimate_risk('regular', M, n0+M, A, H,
[zeros(1,M), signal, zeros(1,M)], sigma);
           otherwise
               Rs(p - p_min + 1) = estimate_risk('NONE', M, n0+M, A, H,
[zeros(1,M), signal, zeros(1,M)], sigma);
       end % switch
       % disp(Rs);
   end
   [R_min, p_opt] = min(Rs); % p_opt is assigned the index of the min. R val's
order, it is not necessarily the min order
   p_opt = p_opt + p_min - 1; % must add p_min to get the correct order, -1
since arrays start at index 1 in matlab
end
% we have to pad the signal within this function since the window length
% changes with every iteration
function [R_min, M_opt] = select_opt_filt_len(p, n0, signal, sigma, M_min,
M_max, regularized)
   assert(2*M_min > p);
   assert(M_max > M_min);

   Rs = zeros(1, M_max - M_min + 1);

   for m = M_min : M_max
       % fprintf("inside opt filt, m: %d\n", m);
       [A, H] = construct_least_sq_matrices(m, p);
       % we pad the signal so that we can calculate the filter lengths at
       % the beginning and the end
       % we need to pass n0+m since we padded the signal
       switch regularized
           case {'regular', 'REGULAR', 'regularized'}
               Rs(m - M_min + 1) = estimate_risk('regular', m, n0+m, A, H,
[zeros(1, m), signal, zeros(1, m)], sigma);
               % fprintf("Risk: %f\n", Rs(m - M_min + 1));
           otherwise
               Rs(m - M_min + 1) = estimate_risk('NONE', m, n0+m, A, H,
[zeros(1, m), signal, zeros(1, m)], sigma);
       end % switch
   end
   [R_min, M_opt] = min(Rs); % M_opt is assigned the index where we ge tthe min
R val. We need to add M_min to get the real
   % disp(Rs);
   % fprintf("M_opt: %d\n", M_opt);
   M_opt = M_opt + M_min - 1;
end
```

```
% this is the G-FL-O algo
function [R_min, p_opt, M_opt] = simult_optim(n0, signal, sigma, p_min, p_max,
M_min, M_max, regularized)
   assert(2*M_min > p_max);
   assert(M_max > M_min);
   assert(p_max > p_min);

   Rs = zeros(p_max - p_min + 1, M_max - M_min +1);
   for p = p_min : p_max
       for m = M_min : M_max
           [A, H] = construct_least_sq_matrices(m, p);
           switch regularized
               case {'regularized', 'REGULAR', 'regular'}
                   Rs(p - p_min + 1, m - M_min + 1) = estimate_risk('regular',
m, n0+m, A, H, [zeros(1,m), signal, zeros(1,m)], sigma);
               otherwise
                   Rs(p - p_min + 1, m - M_min + 1) = estimate_risk('NONE', m,
n0+m, A, H, [zeros(1,m), signal, zeros(1,m)], sigma);
           end
           % fprintf("p: %d, m: %d, R: %d\n", p, m, Rs(p - p_min + 1, m - M_min
+ 1));
       end % for: M
   end % for: p
   [R_min, min_flat_idx] = min( Rs(:) ); % this get shte flattened vector's
index
   [p_opt, M_opt] = ind2sub(size(Rs), min_flat_idx); % we convert it to matrix
index
   p_opt = p_opt + p_min -1;
   M_opt = M_opt + M_min -1;
end
%% defs
% base savitzky golay implementation
% from
https://www.mathworks.com/matlabcentral/answers/335433-how-to-implement-savitzk
y-golay-filter-without-using-inbuilt-functions
function [A, H, y] = base_sg(signal, M, p)
% refer IEEE paper of Robert Schafer 'What is Savitzky Golay Filter?' for
better understanding.
   len = length(signal);
   xn = [zeros(1, M), signal, zeros(1, M)];
   y = zeros(1, len);

   [A, H] = construct_least_sq_matrices(M, p);

   for i = 1:len
       in = xn(1, i : M+M+i);
       in = in(:);
       y(1,i) = H(1,:) * in; % convolution of the sgfilter's impulse response
with the signal values in each window
```

```
        % we only need the first row of H -> check page 4 on the article
        % (right column)
        % H(1,:) is the flipped/time-reversed impulse response of the SG filter
    end
end
% applies at a single point only
% must pass the padded signal
function y = apply_sg(n0, extended_signal, H, M)
    window = extended_signal(n0-M : n0+M);
    window = window(:);
    y = H(1,:) * window;
end
% A and H calculation part of
% from
https://www.mathworks.com/matlabcentral/answers/335433-how-to-implement-savitzk
y-golay-filter-without-using-inbuilt-functions
function [A, H] = construct_least_sq_matrices(M, p)
    d = [-M : M]';
    l = length(d);
    A = zeros(l,p+1); % 2M + 1 many rows, p+1 many cols
    A(:,1) = 1;
    for i = 1:p
        A(:,i+1) = d(:,1).^i;
    end
    % disp(A)
    % disp(size(A))

    H = pinv(A' * A) * A';
    % disp(size(H))
end
% noise is quoted in terms of both snr and sigma (std dev) in the article
function sigma = sigma_from_snr(signal, snr_db)
    signal_power = mean(signal(:).^2);
    snr_linear = 10^(snr_db / 10);
    noise_power = signal_power / snr_linear;
    sigma = sqrt(noise_power);
end
function snr = get_snr(clean_signal, noisy_signal)
    if length(clean_signal) ~= length(noisy_signal)
        error('signals must be the same length');
    end
    noise = noisy_signal - clean_signal;
    signal_power = mean(clean_signal.^2);
    noise_power  = mean(noise.^2);
    snr = 10 * log10(signal_power / noise_power);
end
%% test generation
N = 200;
t = linspace(0, 2*pi, N);
```

```
true_signal = zeros(1,N);
fprintf("Poly orders:\n");
poly_orders = [0,7,1,6,2,5,3,4] % abrupt changes case
% poly_orders = [0,1,2,3,4,5,6,7] % gradual changes case
% poly_orders = [3]
segment_length = N / length(poly_orders);
for i = 1:length(poly_orders)
   start_idx = round((i-1) * segment_length) + 1;
   end_idx = round(i * segment_length);
   t_segment = t(start_idx:end_idx);
   true_signal(start_idx:end_idx) = polyval(polyfit(t_segment, sin(2*t_segment)
+ cos(3*t_segment), poly_orders(i)), t_segment);
end
% THESE FUNCTIONS DO NOT ADD THE GENERATED NOISE TO THE TRUE SIGNAL!!!
function [noise, sigma] = gen_gauss_noise(true_signal, snr)
   signal_power = mean(true_signal.^2);
   noise_power = signal_power / (10^(snr/10));
   noise = sqrt(noise_power) * randn(size(true_signal));
   % sigma = std(noise);
   sigma = sqrt(noise_power); % more optimal
end
function [noise, sigma] = gen_lapl_noise(true_signal, snr)
   signal_power = mean(true_signal.^2);

   noise_power = signal_power / (10^(snr/10));

   sigma = sqrt(noise_power);

   % For Laplacian distribution, std = b*sqrt(2)
   % where b is the scale parameter
   laplacian_scale = sigma / sqrt(2);

   u1 = rand(size(true_signal));
   u2 = rand(size(true_signal));
   noise = laplacian_scale * (log(u1) - log(u2));
end
function [noise, sigma] = gen_unif_noise(true_signal, snr)
   signal_power = mean(true_signal.^2);

   noise_power = signal_power / (10^(snr/10));

   sigma = sqrt(noise_power);

   % For uniform distribution in range [-a,a], std = a/sqrt(3)
   % So a = std*sqrt(3)
   uniform_range = sigma * sqrt(3);

   noise = (2*rand(size(true_signal)) - 1) * uniform_range;
end
```

```
function [noise, sigma] = gen_noise(type, true_signal, snr)
   switch type
       case 'gs'
           [noise, sigma] = gen_gauss_noise(true_signal, snr);
       case 'lp'
           [noise, sigma] = gen_lapl_noise(true_signal, snr);
       case 'un'
           [noise, sigma] = gen_unif_noise(true_signal, snr);
       otherwise
           assert(0); % fail
   end % switch
end
snr = 12; % db
noise_type = 'gs';
% noise = sigma * randn(size(true_signal));
% [gauss_noise, sigma] = gen_gauss_noise(true_signal, snr);
[noise, sigma] = gen_noise(noise_type, true_signal, snr);
noisy_signal = true_signal + noise;
% OBSERVATIONS:
% regularized outperforms all in this case with uniform noise:
% [0,7,1,6,2,5,3,4] (other than that it almost always is behind
% unregularized.
% unregularized performs best in laplacian noise'
% unregularized always is better than base sg
% under very specific conditions regularized might perform the worst.
% regularized is less sensitive to decreases in the snr (more noise per
% signal)
%% main
fprintf("Results for snr: %d, noise type: %s, num different orders in signal:
%d\n", snr, noise_type, length(poly_orders));
%
% PARAMETERS:
% base sg params:
M = 12; p = 3;
% optim params
p_min = 0; p_max = 7;
M_min = 4; M_max = 20;
%
% PLOTTING PARAMS
grid_r = 2; grid_c = 2;
figure;
% orig signal
subplot(grid_r, grid_c, 1);
plot(t, true_signal);
title("original signal");
% nosiy signal
subplot(grid_r, grid_c, 3);
plot(t, noisy_signal);
title("noisy signal");
```

```
% sg filtered signal
% subplot(grid_r, grid_c, 5);
% [A, H, base_sg_res] = base_sg(noisy_signal, M, p);
% plot(t, base_sg_res, 'r-');
% hold on;
% plot(t, true_signal, 'b-');
% title(sprintf("base sg filtered signal (M: %d, p: %d)", M, p));
% legend('sg fit', 'original signal');
% RISK ESTIMATES PLOT
Rs = zeros(1, length(noisy_signal));
% evaluate R at all centers of windows of size 2*M+1
[A, H, base_sg_res] = base_sg(noisy_signal, M, p);
for n0_i = [M+1:length(noisy_signal)-M]
   R = estimate_risk('NONE', M, n0_i, A, H, [zeros(1,M), noisy_signal, zeros(1,
M)], sigma);
   Rs(n0_i) = R;
   % fprintf("Risk estimate for %d to %d: %d\n",n0-M, n0+M, R);
end
% subplot(grid_r, grid_c, 2);
% plot(t, Rs);
% title(sprintf("UNregularized Risk estimates at different window centers (M:
%d, p: %d)", M, p));
% OPTIMAL ORDER PLOT
opt_orders = zeros(1, length(noisy_signal));
for n0_i = [M+1:length(noisy_signal)-M]
   [~, order] = select_opt_order(M, n0_i, [zeros(1,M), noisy_signal, zeros(1,
M)], sigma, p_min, p_max, 'NONE');
   opt_orders(n0_i) = order;
end
subplot(grid_r, grid_c, 2);
plot(t, opt_orders);
title("Optimal orders at different window centers (unregularized risk)");
% OPTIMAL LENGTHS PLOT
opt_lens = zeros(1, length(noisy_signal));
% we pass the actual order of the signal based on where n0 is
% this is an ideal assumption to see how the filter length selection
% performs
segment_progress = 1;
segment = 1;
% fprintf("length section\n");
for n0_i = [1:length(noisy_signal)]
   p = poly_orders(segment); % this part ensures that we fit with the correct p
   if segment_progress > segment_length
       segment = segment + 1;
       segment_progress = 1;
   end
   [~, len] = select_opt_filt_len(p, n0_i, noisy_signal, sigma, M_min, M_max,
'NONE');
   opt_lens(n0_i) = len;
```

```
end
subplot(grid_r, grid_c, 4);
plot(t, opt_lens);
title("Optimal filter lengths at different window centers (unregularized
risk)");
%%
% % simultaneous optim
% sim_opt_lens = zeros(1, length(noisy_signal));
% sim_opt_ords = zeros(1, length(noisy_signal));
%
% for n0_i = [1:length(noisy_signal)]
%     [~, p, m] = simult_optim(n0_i, noisy_signal, sigma, p_min, p_max, M_min,
M_max, 'NONE');
%     sim_opt_ords(n0_i) = p;
%     sim_opt_lens(n0_i) = m;
%     % fprintf("p: %d, m: %d\n", p, m);
% end % for: n0
%
% subplot(grid_r, grid_c, 7);
% plot(t, sim_opt_lens, "g");
% title("Simulatenously optimized filter lengths (unregularized risk)");
%
% subplot(grid_r, grid_c, 8);
% plot(t, sim_opt_ords, "g");
% title("Simulatenously optimized orders (unregularized risk)");
% Base savitzky golay
figure
grid_r = 2; grid_c = 1;
plot(t, true_signal, "black", "LineWidth", 2);
hold on;
plot(t, base_sg_res, "red");
title("base savitzky-golay")
legend('base savitzky-golay', 'actual signal');
fprintf("MSE between base savitzky-golay and the TRUE signal:\n");
mse_base_sg = mse(true_signal, base_sg_res);
fprintf("\tMSE of base sg filtered signal: %f\n", mse_base_sg);
% The article has its own lingo for different configurations of the
% adaptive filter:
% G-FL: fixed order, no regularization
%   M is elt [2, 20], p = 3
% G-FL-R: fixed order, with regularization
%   same as G-FL, lambda = 12 * sigma^2
% G-O: fixed length, no regularization
%   M = 15, p is elt [1, 5]
% G-O-R: fixed length, with regularization
%   same as G-O, lambda = 12 * sigma^2
% G-FL-O: simult. optim, no regularization
%   M is elt [10, 20], p is elt [1, 7]
% G-FL-O-R: simult. optim, with regularization
```

```
%   M is elt [10, 20], p is elt [1, 7], lambda = 12 * sigma^2
% G-FL-O(-R) - simultaneous optimization with unregularized risk
function [G_FL_O_filtered_signal, G_FL_O_orders, G_FL_O_lengths] =
GFLO(noisy_signal, p_min, p_max, M_min, M_max, sigma)
   G_FL_O_filtered_signal = zeros(1, length(noisy_signal));
   G_FL_O_orders  = zeros(1, length(noisy_signal));
   G_FL_O_lengths = zeros(1, length(noisy_signal));

   for n0 = [1:length(noisy_signal)]
       [~, p, m] = simult_optim(n0, noisy_signal, sigma, p_min, p_max, M_min,
M_max, 'NONE');
       [~, H] = construct_least_sq_matrices(m, p);
       G_FL_O_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
       G_FL_O_lengths(n0) = m;
       G_FL_O_orders(n0) = p;
   end
end % function
[G_FL_O_filtered_signal, G_FL_O_orders, G_FL_O_lengths] = GFLO(noisy_signal,
p_min, p_max, M_min, M_max, sigma);
function err = mse(true_signal, signal)
   err = sum((true_signal - signal).^2) / (length(signal));
end
grid_r = 3; grid_c = 2;
figure;
subplot(grid_r, grid_c, 1);
plot(t, G_FL_O_filtered_signal, "r-");
hold on;
title("G-FL-O");
plot(t, true_signal, "black", "LineWidth", 2);
legend('G-FL-O', 'true signal');
subplot(grid_r, grid_c, 3);
plot(t, G_FL_O_orders);
title("G-FL-O orders");
subplot(grid_r, grid_c, 5);
plot(t, G_FL_O_lengths);
title("G-FL-O filter lengths");
fprintf("\nMSE between G-FL-O(-R) and the TRUE signal:\n");
mse_filtered = mse(true_signal, G_FL_O_filtered_signal);
fprintf("\tMSE of G-FL-O: %f\n", mse_filtered);
% G-FL-O-R - simultaneous optim with regularized risk
function [G_FL_O_R_filtered_signal, G_FL_O_R_orders, G_FL_O_R_lengths] =
GFLOR(noisy_signal, p_min, p_max, M_min, M_max, sigma)
   G_FL_O_R_filtered_signal = zeros(1, length(noisy_signal));
   G_FL_O_R_orders  = zeros(1, length(noisy_signal));
   G_FL_O_R_lengths = zeros(1, length(noisy_signal));

   for n0 = [1:length(noisy_signal)]
```

```
        [~, p, m] = simult_optim(n0, noisy_signal, sigma, p_min, p_max, M_min,
M_max, 'regular');
        [~, H] = construct_least_sq_matrices(m, p);
        G_FL_O_R_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
        G_FL_O_R_lengths(n0) = m;
        G_FL_O_R_orders(n0) = p;
    end
end % function
[G_FL_O_R_filtered_signal, G_FL_O_R_orders, G_FL_O_R_lengths] =
GFLOR(noisy_signal, p_min, p_max, M_min, M_max, sigma);
subplot(grid_r, grid_c, 2);
plot(t, true_signal, "black", "LineWidth", 2);
hold on;
plot(t, G_FL_O_R_filtered_signal, "blue");
legend("true signal", "G-FL-O-R");
title("G-FL-O-R");
subplot(grid_r, grid_c, 4);
plot(t, G_FL_O_R_orders);
title("G-FL-O-R orders");
subplot(grid_r, grid_c, 6);
plot(t, G_FL_O_R_lengths);
title("G-FL-O-R filter lengths");
fprintf("\tMSE of G-FL-O-R: %f\n", mse(true_signal, G_FL_O_R_filtered_signal));
% TODO:
% visualize the boundaries between different orders of polynomials on the
% plots
% output snr vs input snr graphs for all algos
% report the results
% G-FL(-R) - Only optimize filter length
p = 3;
function [G_FL_filtered_signal, G_FL_filter_lens] = GFL(p, noisy_signal, M_min,
M_max, sigma)
    G_FL_filtered_signal = zeros(1, length(noisy_signal));
    G_FL_filter_lens = zeros(1, length(noisy_signal));
    for n0 = [1:length(noisy_signal)]
        [~, m] = select_opt_filt_len(p, n0, noisy_signal, sigma, M_min, M_max,
'NONE');
        G_FL_filter_lens(n0) = m;
        [~, H] = construct_least_sq_matrices(m, p);
        G_FL_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
    end
end % function
[G_FL_filtered_signal, G_FL_filter_lens] = GFL(p, noisy_signal, M_min, M_max,
sigma);
figure;
grid_r = 2;
grid_c = 2;
```

```
subplot(grid_r, grid_c, 1);
plot(t, true_signal, "black", "LineWidth", 2);
hold on;
plot(t, G_FL_filtered_signal, "red");
title("G-FL");
legend("true signal", "G-FL filtered");
subplot(grid_r, grid_c, 3);
plot(t, G_FL_filter_lens, "blue");
title("G-FL Filter lengths");
p = 5;
p_min = 0; p_max = 7;
M_min = 4; M_max = 20;
function [G_FL_R_filtered_signal, G_FL_R_filter_lens] = GFLR(p, noisy_signal,
M_min, M_max, sigma)
   G_FL_R_filtered_signal = zeros(1, length(noisy_signal));
   G_FL_R_filter_lens = zeros(1, length(noisy_signal));
   for n0 = [1:length(noisy_signal)]
       [~, m] = select_opt_filt_len(p, n0, noisy_signal, sigma, M_min, M_max,
'regular');
       G_FL_R_filter_lens(n0) = m;
       [~, H] = construct_least_sq_matrices(m, p);
       G_FL_R_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
   end
end % function
[G_FL_R_filtered_signal, G_FL_R_filter_lens] = GFLR(p, noisy_signal, M_min,
M_max, sigma);
subplot(grid_r, grid_c, 2);
plot(t, true_signal, "black", "LineWidth", 2);
hold on;
plot(t, G_FL_R_filtered_signal, "red");
title("G-FL-R");
legend("true signal", "G-FL-R filtered");
subplot(grid_r, grid_c, 4);
plot(t, G_FL_R_filter_lens, "blue");
title("G-FL-R Filter lengths");
fprintf("\nMSE between G-FL(-R) and the TRUE signal:\n");
fprintf("\tMSE of G-FL: %f\n", mse(true_signal, G_FL_filtered_signal));
fprintf("\tMSE of G-FL-R: %f\n", mse(true_signal, G_FL_R_filtered_signal));
% G-O(-R) - optimize polynomial order only
m = 15;
% p_max = 5; p_min = 1;
function [G_O_filtered_signal, G_O_orders] = GO(m, noisy_signal, p_min, p_max,
sigma)
   G_O_filtered_signal = zeros(1, length(noisy_signal));
   G_O_orders = zeros(1, length(noisy_signal));
   for n0 =  [1:length(noisy_signal)] % [m+1:length(noisy_signal)-m]
       [~, p] = select_opt_order(m, n0, noisy_signal, sigma, p_min, p_max,
'NONE');
```

```
        G_O_orders(n0) = p;
        [~, H] = construct_least_sq_matrices(m, p);
        G_O_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
    end
end
[G_O_filtered_signal, G_O_orders] = GO(m, noisy_signal, p_min, p_max, sigma);
figure;
subplot(grid_r, grid_c, 1);
plot(t, true_signal, "black", "LineWidth", 2);
hold on;
plot(t, G_O_filtered_signal, "red");
title("G-O");
legend("true signal", "G-O filtered signal");
subplot(grid_r, grid_c, 3);
plot(t, G_O_orders, "blue");
title("G-O orders");
function [G_O_R_filtered_signal, G_O_R_orders] = GOR(m, noisy_signal, p_min,
p_max, sigma)
    G_O_R_filtered_signal = zeros(1, length(noisy_signal));
    G_O_R_orders = zeros(1, length(noisy_signal));

    for n0 = [1:length(noisy_signal)] % [m+1:length(noisy_signal)-m]
        [~, p] = select_opt_order(m, n0, noisy_signal, sigma, p_min, p_max,
'regular');
        G_O_R_orders(n0) = p;
        [~, H] = construct_least_sq_matrices(m, p);
        G_O_R_filtered_signal(n0) = apply_sg(n0+m, [zeros(1,m), noisy_signal,
zeros(1,m)], H, m);
    end
end % function
[G_O_R_filtered_signal, G_O_R_orders] = GOR(m, noisy_signal, p_min, p_max,
sigma);
subplot(grid_r, grid_c, 2);
plot(t, true_signal, "black", "LineWidth", 2);
hold on;
plot(t, G_O_R_filtered_signal, "red");
title("G-O-R");
legend("true signal", "G-O-R filtered signal");
subplot(grid_r, grid_c, 4);
plot(t, G_O_R_orders, "blue");
title("G-O-R orders");
fprintf("\nMSE between G-O(-R) and the TRUE signal:\n");
fprintf("\tMSE of G-O: %f\n", mse(true_signal, G_O_filtered_signal));
fprintf("\tMSE of G-O-R: %f\n", mse(true_signal, G_O_R_filtered_signal));
%%
% TESTING ON PIECEWISE SIGNALS WITH DIFFERENT SNRs & noise types
% SNR SWEEP PART
% this part will take some time
```

```
% noise_type = 'lp';
noise_types = ["un", "gs", "lp"];
noise_type_full_name = ["uniform", "gaussian", "laplace"];
for noise_type = noise_types
   fprintf("\nDiffernt SNRs testing - noise type: %s\n", noise_type);

   input_snr_space = [-5:5:25];
   input_noise_space = zeros(length(input_snr_space), length(true_signal));
   noisy_signal_space = zeros(length(input_snr_space), length(true_signal));

   GO_output_snr_space = zeros(1, length(input_snr_space));
   GOR_output_snr_space = zeros(1, length(input_snr_space));

   GFL_output_snr_space = zeros(1, length(input_snr_space));
   GFLR_output_snr_space = zeros(1, length(input_snr_space));

   GFLO_output_snr_space = zeros(1, length(input_snr_space));
   GFLOR_output_snr_space = zeros(1, length(input_snr_space));

   for snr_i = 1:length(input_snr_space)
       [noise, sigma] = gen_noise(noise_type, true_signal,
input_snr_space(snr_i));
       input_noise_space(snr_i, :) = noise;
       noisy_signal_space(snr_i, :) = input_noise_space(snr_i, :) +
true_signal;

   end

   % disp(input_snr_space);
   % disp(noisy_signal_space);

   for snr_i = 1:length(input_snr_space)
       noisy_signal = noisy_signal_space(snr_i, :);

       [GO_out, ~] = GO(m, noisy_signal, p_min, p_max, sigma);
       [GOR_out, ~] = GOR(m, noisy_signal, p_min, p_max, sigma);
       GO_output_snr_space(snr_i) = get_snr(true_signal, GO_out);
       GOR_output_snr_space(snr_i) = get_snr(true_signal, GOR_out);

       [GFL_out, ~] = GFL(p, noisy_signal, M_min, M_max, sigma);
       [GFLR_out, ~] = GFLR(p, noisy_signal, M_min, M_max, sigma);
       GFL_output_snr_space(snr_i) = get_snr(true_signal, GFL_out);
       GFLR_output_snr_space(snr_i) = get_snr(true_signal, GFLR_out);

       [GFLO_out, ~, ~] = GFLO(noisy_signal, p_min, p_max, M_min, M_max,
sigma);
       [GFLOR_out, ~, ~] = GFLOR(noisy_signal, p_min, p_max, M_min, M_max,
sigma);
       GFLO_output_snr_space(snr_i) = get_snr(true_signal, GFLO_out);
```

```
        GFLOR_output_snr_space(snr_i) = get_snr(true_signal, GFLOR_out);
    end

    figure;
    plot(input_snr_space, GO_output_snr_space, "r-o"); hold on;
    plot(input_snr_space, GOR_output_snr_space, "g-s"); hold on;
    plot(input_snr_space, GFL_output_snr_space, "b-x"); hold on;
    plot(input_snr_space, GFLR_output_snr_space, "k-*"); hold on;
    plot(input_snr_space, GFLO_output_snr_space, "m-^"); hold on;
    plot(input_snr_space, GFLOR_output_snr_space, "c-d");

    legend("GO", "GOR", "GFL", "GFLR", "GFLO", "GFLOR");
    title(sprintf("output snr vs input snr for %s noise", noise_type_full_name(
find(noise_types == noise_type)) ));
    xlabel("input snr (db)");
    ylabel("output snr (db)");
end % for noise type
```