General signal generator

Signals and Systems: Mini project I
Spring 2024

Edited By

Yehia Said Gewily 21010705

Yahya Emad El-Deen El-Kony 21011557

Ahmed Saeed Muhammed 21010091

Muhammed Ibrahim Risk 21011047

University of Alexandria

EEC 271 Signals and Systems

Apr 2024

LaTeX

Objective

The primary objective of this project is to develop a versatile and user-interactive MATLAB application capable of generating and manipulating time-domain signals based on specified parameters. This application aims to provide a practical understanding of various signal properties and transformations, which are foundational concepts in signal processing and related disciplines.

Scope

The application covers a broad spectrum of signal types, including Direct Current (DC), Ramp, Polynomial, Exponential, Sinusoidal, Sinc, and Triangle signals. Users can define these signals over specified intervals, segmented by user-defined breakpoints, thereby allowing complex and piecewise signal constructions. Each segment can be uniquely characterized by parameters appropriate to the signal type, such as amplitude, frequency, phase, or polynomial order, which the user inputs dynamically.

Furthermore, the program includes functionalities for signal operations such as amplitude scaling, time reversal, time shift, signal expansion, compression, clipping, and differentiation. These operations facilitate an in-depth exploration of signal behaviour under various mathematical transformations.

The application visually presents the generated signals and their post-operation states, enabling immediate graphical feedback on the effects of the manipulations performed. This feature is crucial for educational and experimental purposes, allowing users to visually correlate the mathematical definitions and operations with their graphical outcomes.

Program Overview

General Description

The General Signal Generator is a MATLAB-based application designed to facilitate the generation, manipulation, and visualization of various signal types across user-defined time intervals. The program is structured to allow for interactive user inputs, which dictate the characteristics and transformations applied to the signals. It provides a dynamic platform for users to understand the impact of different parameters and operations on the resultant signals through immediate visual feedback.

Program Structure

The program consists of a main function and several helper functions, each responsible for specific tasks such as gathering inputs, generating the initial signal, applying transformations, and plotting the results. Here is a brief overview of each component:

1- “main” Function:

- **Purpose:** *Serves as the central control unit for the application, coordinating between user inputs, signal generation, operations application, and visualization.*
- **Process:**
  - *Collect user-defined parameters for signal generation.*
  - *Generate the signal based on the parameters and breakpoints.*
  - *Allow the user to iteratively apply transformations to the signal.*
  - *Plot both the original and modified signals for comparison.*

2- “getUserInputs” Function:

- **Purpose:** *Captures all necessary parameters from the user, including sampling frequency, time scale, and breakpoints.*
- **Process:**
  - *Prompt the user for the sampling frequency, start and end times of the signal, and the number of breakpoints.*
  - *Collect positions for the breakpoints to define segments for different signal behaviours.*

3- “generateSignal” Function:

- **Purpose:** *Constructs the signal by segmenting the time scale based on the breakpoints and applying user-specified signal characteristics for each segment.*
- **Process:**
  - *Divide the complete time interval into segments as defined by the breakpoints.*
  - *For each segment, generate the appropriate type of signal using specified parameters.*

4- “applyOperations” Function:

- **Purpose:** *Modifies the signal according to user-selected operations such as scaling, reversing, and shifting.*
- **Process:**
  - *Implement the operation selected by the user on the current signal.*
  - *Update the signal display after each operation to show the new state of the signal.*

Function Descriptions

Each function within the General Signal Generator program serves a specific role in the process of generating and manipulating signals. Here is a detailed explanation of the key functions:

1. main Function

```
function main
    % Main function to orchestrate the signal generation and
operations
    [Fs, t, nBreaks, breakPoints] = getUserInputs();
    originalSignal = generateSignal(Fs, t, nBreaks, breakPoints);

    % Plot the original signal immediately after creation
    figure;
    plot(t, originalSignal);
    title('Initial Signal - Before Any Operations');
    xlabel('Time (s)');
    ylabel('Amplitude');

    % Initialize the modified signal
    modifiedSignal = originalSignal;

    % Define operation choices
    operations = {'Scale', 'Reverse', 'Shift', 'Expand', 'Compress',
'Clip', 'Derivative', 'None', 'Stop'};
    operationCodes = {'s', 'rev', 'sh', 'ex', 'co', 'cl', 'd', 'n',
'stop'};

    % Loop to apply operations until the user enters 'Stop'
    while true
        % Display available operations
        disp('Available operations:');
        for i = 1:length(operations)
            disp([num2str(i) '. ' operations{i} ' ('
operationCodes{i} ')']);
        end

        % Get user selection
        operationSel = input('Select an operation number or code to
perform on the signal or enter Stop to finish: ', 's');

        % Check if user input is a number or operation code and get
the corresponding operation
        operationIdx = find(strcmpi(operationSel, operationCodes) |
strcmpi(operationSel, operations));
        if isempty(operationIdx)
            disp('Invalid operation selection. Please try again.');
            continue;
        elseif strcmpi(operationSel, 'stop') ||
strcmpi(operationSel, '9')
            % When 'Stop' is entered, plot the original and the last
modified signals
            figure;
            subplot(2, 1, 1);
            plot(t, originalSignal);
            title('Original Signal');
            xlabel('Time (s)');
            ylabel('Amplitude');

            subplot(2, 1, 2);
            plot(t, modifiedSignal);
            title('Final Modified Signal');
            xlabel('Time (s)');
            ylabel('Amplitude');
            break;
        end

        % Get the actual operation name
        operation = operations{operationIdx};

        % Apply the selected operation
        modifiedSignal = applyOperations(modifiedSignal, t, Fs,
operation);

        % Plot the modified signal after each operation
        figure;
        plot(t, modifiedSignal);
        title(['Modified Signal - ', operation]);
        xlabel('Time (s)');
        ylabel('Amplitude');
    end
end
```

- **Purpose:** *To orchestrate the overall execution of the program, integrating input collection, signal generation, signal manipulation, and visualization.*
- **Process:**
    - **Initialization***: Calls getUserInputs to collect all necessary parameters from the user.*
    - **Signal Generation:** *Utilizes generateSignal to create the initial signal based on the collected parameters.*
    - **Operation Loop:** *Enters a loop that repeatedly prompts the user for operations to apply on the signal. The loop continues until the user decides to stop.*
    - **Visualization:** *Plots the original and the final modified signals for comparison.*

- **Key Components:**
    - *A while loop to handle continuous user interaction until the "Stop" command is entered.*
    - *Calls to other functions (applyOperations, generateSignal, getUserInputs) to manage specific tasks.*

2. getUserInputs Function:

```
function [Fs, t, nBreaks, breakPoints] = getUserInputs()
    % Get general user inputs for signal generation
    Fs = input('Enter the sampling frequency of the signal: ');
    tStart = input('Enter the start of time scale: ');
    tEnd = input('Enter the end of time scale: ');
    nBreaks = input('Enter the number of break points: ');
    breakPoints = zeros(1, nBreaks);
    for i = 1:nBreaks
        breakPoints(i) = input(['Enter the position of break point '
num2str(i) ': ']);
    end
    t = linspace(tStart, tEnd, (tEnd - tStart) * Fs);
end
```

- **Purpose:** *To gather all the initial setup parameters required for signal generation from the user.*
- **Inputs:**
    - *Sampling frequency (Fs).*
    - *Start and end of the time scale (tStart, tEnd).*
    - *Number of breakpoints (nBreaks) and their positions (breakPoints).*
- **Outputs:**
    - *A vector of time points (t).*
    - *Sampling frequency and breakpoints as inputs for signal generation.*
- **Process:**
    - *Prompt user for each input using input() function.*
    - *Validate inputs to ensure they meet expected formats and ranges.*

3. “generateSignal” Function:

```
function signal = generateSignal(Fs, t, nBreaks, breakPoints)
    % Generate the complete signal by combining portions
    regions = [min(t), breakPoints, max(t)];
    signal = zeros(size(t));
    for i = 1:length(regions) - 1
        [signalType, params] = getSignalSpecs();
        signal = signal + generatePortion(signalType, t, regions(i),
regions(i+1), params);
```

```
    end
end
```

- **Purpose:** *To construct the complete signal from various segments, each defined by the user's specifications.*
- **Inputs:** *Sampling frequency, time vector, number of breakpoints, and their positions.*
- **Output:** *A single time-domain signal that combines all specified segments.*
- **Process:**
    - *Segment the time scale into intervals based on breakpoints.*
    - *For each interval, ask the user for the signal type and parameters through getSignalSpecs.*
    - *Generate and sum the signal portions for each interval using generatePortion.*

4. "getSignalSpecs" Function:

```
function [type, params] = getSignalSpecs()
    % Define valid signal types
    validTypes = {'dc', 'ramp', 'polynomial', 'exponential', 
'sinusoidal', 'sinc', 'triangle'};
    type = lower(input('Enter signal type (DC, Ramp, Polynomial, 
Exponential, Sinusoidal, Sinc, Triangle): ', 's'));

    if ~ismember(type, validTypes)
        error('Unsupported signal type entered. Valid types are: 
%s', strjoin(validTypes, ', '));
    end

    params = struct();
    switch type
        case 'dc'
            params.amplitude = input('Enter amplitude: ');
        case 'ramp'
            params.slope = input('Enter slope: ');
            params.intercept = input('Enter intercept: ');
        case 'polynomial'
            params.amplitude = input('Enter amplitude: ');
            params.power = input('Enter power: ');
            params.intercept = input('Enter intercept: ');
        case 'sinusoidal'
            params.amplitude = input('Enter amplitude: ');
            params.frequency = input('Enter frequency: ');
            params.phase = input('Enter phase: ');
        case 'sinc'
            params.amplitude = input('Enter amplitude: ');
            params.centerShift = input('Enter center shift: ');
        case 'triangle'
            params.amplitude = input('Enter amplitude: ');
            params.centerShift = input('Enter center shift: ');
            params.width = input('Enter width: ');
        case 'exponential'
            params.amplitude = input('Enter amplitude: ');
            params.exponent = input('Enter exponent: ');
    end
end
```

- **Purpose:** *To capture the specifications for a signal segment from the user.*
- **Output:** *Signal type and associated parameters as a structure.*
- **Process:**
  - *Prompt the user for the type of signal (e.g., DC, Ramp, etc.) and relevant parameters (e.g., amplitude, frequency).*
  - *Validate the user's input against a list of supported signal types.*

5. “generatePortion” Function:

```
function portion = generatePortion(type, t, start, endd, params)
    % Generate signal portion based on the type and parameters
    idx = t >= start & t <= endd;
    portion = zeros(size(t));
    switch type
        case 'dc'
            portion(idx) = params.amplitude;
        case 'ramp'
            portion(idx) = params.slope * (t(idx) - start) + 
params.intercept;
        case 'polynomial'
            portion(idx) = params.amplitude * (t(idx) - 
start).^params.power + params.intercept;
        case 'sinusoidal'
            portion(idx) = params.amplitude * sin(2 * pi * 
params.frequency * (t(idx) - start) + params.phase);
        case 'sinc'
            shifted_t = t(idx) - (start + params.centerShift);
            portion(idx) = params.amplitude * sinc(shifted_t);
        case 'triangle'
            shifted_t = t(idx) - (start + params.centerShift);
            portion(idx) = params.amplitude * max(1 - 
abs(shifted_t)/params.width, 0);
        case 'exponential'
            portion(idx) = params.amplitude * exp(params.exponent * 
(t(idx) - start));
    end
end
```

- **Purpose:** *To generate a signal portion based on type and parameters over a specified interval.*
- **Inputs:** *Signal type, time vector, interval start and end, signal parameters.*
- **Output:** *A vector representing the signal portion.*

- **Process:**
    - *Identify the subset of the time vector that falls within the given interval.*
    - *Calculate the signal values for this subset based on the specified signal type and parameters.*

6. “applyOperations” Function:

```
function modifiedSignal = applyOperations(signal, t, Fs, op)
    % Modify the signal based on user input operation
    switch lower(op)
        case 'scale'
            factor = input('Enter scale factor: ');
            modifiedSignal = signal * factor;
        case 'reverse'
            modifiedSignal = fliplr(signal);
        case 'shift'
            shift_val = input('Enter shift value (positive for right
shift, negative for left shift): ');
            modifiedSignal = zeros(size(signal));
            if shift_val > 0
                modifiedSignal(shift_val+1:end) = signal(1:end-
shift_val);
            else
                modifiedSignal(1:end+shift_val) = signal(-
shift_val+1:end);
            end
        case 'expand'
            expand_val = input('Enter expansion factor: ');
            modifiedSignal = interp1(t, signal, linspace(min(t),
max(t), numel(t)/expand_val), 'linear', 'extrap');
        case 'compress'
            compress_val = input('Enter compression factor: ');
            modifiedSignal = interp1(t, signal, linspace(min(t),
max(t), numel(t)*compress_val), 'linear', 'extrap');
        case 'clip'
            upper_clip = input('Enter upper clipping value: ');
            lower_clip = input('Enter lower clipping value: ');
            modifiedSignal = min(max(signal, lower_clip),
upper_clip);
        case 'derivative'
            modifiedSignal = computeDerivative(signal, t);  %
Improved derivative function
        case 'none'
            modifiedSignal = signal;
        otherwise
            disp('No valid operation selected. Returning original
signal.');
            modifiedSignal = signal;
    end
end
```

- **Purpose:** *To modify the signal according to the selected operation.*
- **Inputs:** *Current signal, time vector, sampling frequency, and chosen operation.*
- **Output:** *Modified signal.*
- **Process:**
    - *Interpret the user's operation choice.*
    - *Apply the corresponding mathematical transformation to the signal.*
    - *Return the modified signal for further operations or visualization.*

7. "computeDerivative" Function:

```
function modifiedSignal = computeDerivative(signal, t)
    % Compute the derivative of a signal using a more accurate
central difference method
    dt = diff(t);
    modifiedSignal = zeros(size(signal));
    modifiedSignal(1) = (signal(2) - signal(1)) / dt(1);  % forward
difference
    for i = 2:length(signal)-1
        modifiedSignal(i) = (signal(i+1) - signal(i-1)) / (dt(i-1) +
dt(i));  % central difference
    end
    modifiedSignal(end) = (signal(end) - signal(end-1)) / dt(end);
% backward difference
end
```

- **Purpose:** *To compute the derivative of the signal using a numerical approximation method.*
- **Inputs:** *Signal vector and time vector.*
- **Output:** *Derivative of the signal.*
- **Process:**
    - *Use central difference method for internal points and forward/backward difference for the endpoints.*
    - *This function provides a practical approach to numerical differentiation, which is crucial for analyzing signal dynamics.*

Building Numerical Differentiation function

The computeDerivative function was chosen over MATLAB's built-in diff function for computing the derivative of a signal for several key reasons:

1- *Accuracy: The computeDerivative uses the central difference method, which provides second-order accuracy (errors proportional to the square of the step size), compared to the first-order accuracy of the diff function, which uses a forward difference method.*

2- *Noise Reduction: The central difference method averages the slopes on either side of each point, smoothing out noise and providing a more stable derivative estimate, which is particularly beneficial for noisy signal data.*

3- *Boundary Handling: computeDerivative carefully handles the endpoints using forward and backward differences, ensuring accurate derivative calculation across the entire signal, including the boundaries where central difference cannot be applied.*

Overall, the use of computeDerivative offers a more precise and robust approach for numerical differentiation in signal processing applications.

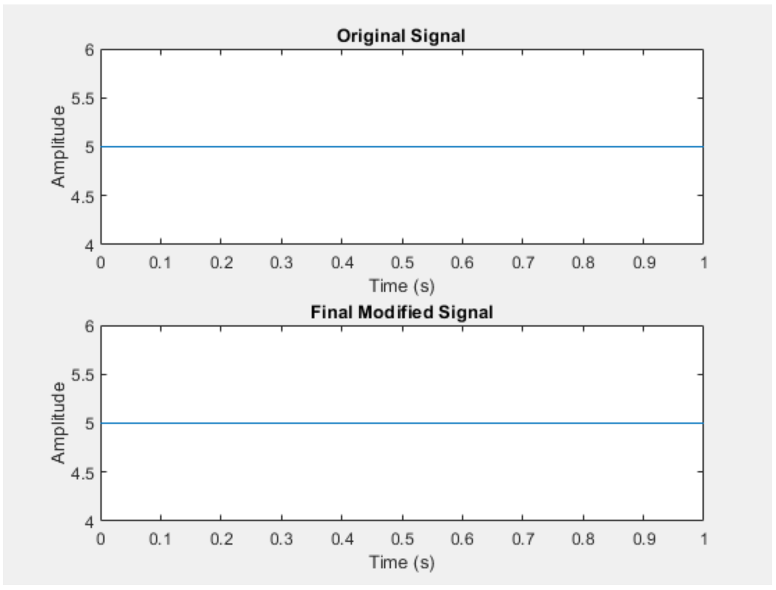
Input and Output Samples

Test Case 1: Basic DC Signal Generation

Inputs:

```
>> main

Enter the sampling frequency of the signal: 100
Enter the start of time scale: 0
Enter the end of time scale: 1
Enter the number of break points: 0
Enter signal type (DC, Ramp, Polynomial, Exponential, Sinusoidal,
Sinc, Triangle): DC
Enter amplitude: 5
Available operations:
1. Scale (s)
2. Reverse (rev)
3. Shift (sh)
4. Expand (ex)
5. Compress (co)
6. Clip (cl)
7. Derivative (d)
8. None (n)
9. Stop (stop)
Select an operation number or code to perform on the signal or enter
Stop to finish: stop
```

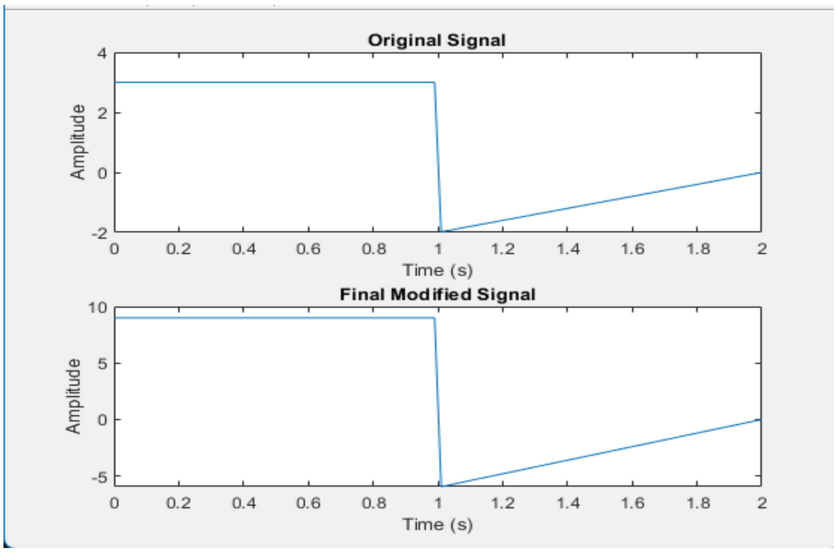


Test Case 2: Signal with Multiple Regions (DC and Ramp)

Inputs:

```
>> main
Enter the sampling frequency of the signal: 50
Enter the start of time scale: 0
Enter the end of time scale: 2
Enter the number of break points: 1
Enter the position of break point 1: 1
Enter signal type (DC, Ramp, Polynomial, Exponential, Sinusoidal, Sinc, Triangle):
DC
Enter amplitude: 3
Enter signal type (DC, Ramp, Polynomial, Exponential, Sinusoidal, Sinc, Triangle):
Ramp
Enter slope: 2
Enter intercept: -2
Available operations:
1. Scale (s)
2. Reverse (rev)
3. Shift (sh)
4. Expand (ex)
5. Compress (co)
6. Clip (cl)
7. Derivative (d)
8. None (n)
9. Stop (stop)
Select an operation number or code to perform on the signal or enter Stop to
finish: s
Enter scale factor: 3
Available operations:
1. Scale (s)
2. Reverse (rev)
3. Shift (sh)
4. Expand (ex)
5. Compress (co)
6. Clip (cl)
7. Derivative (d)
8. None (n)
9. Stop (stop)
Select an operation number or code to perform on the signal or enter Stop to
```

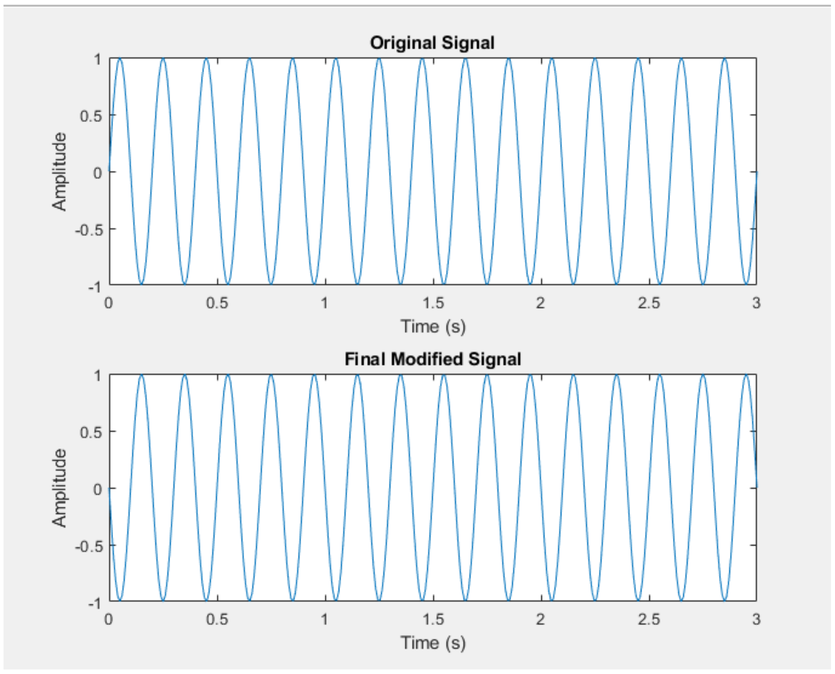
```
finish: stop
>>
```



Test Case 3: Reversal Operation on a Sinusoidal Signal

Inputs:

```
>> main
Enter the sampling frequency of the signal: 100
Enter the start of time scale: 0
Enter the end of time scale: 3
Enter the number of break points: 0
Enter signal type (DC, Ramp, Polynomial, Exponential, Sinusoidal, Sinc, Triangle):
Sinusoidal
Enter amplitude: 1
Enter frequency: 5
Enter phase: 0
Available operations:
1. Scale (s)
2. Reverse (rev)
3. Shift (sh)
4. Expand (ex)
5. Compress (co)
6. Clip (cl)
7. Derivative (d)
8. None (n)
9. Stop (stop)
Select an operation number or code to perform on the signal or enter Stop to
finish: rev
Available operations:
1. Scale (s)
2. Reverse (rev)
3. Shift (sh)
4. Expand (ex)
5. Compress (co)
6. Clip (cl)
7. Derivative (d)
8. None (n)
9. Stop (stop)
Select an operation number or code to perform on the signal or enter Stop to
finish: stop
>>
```

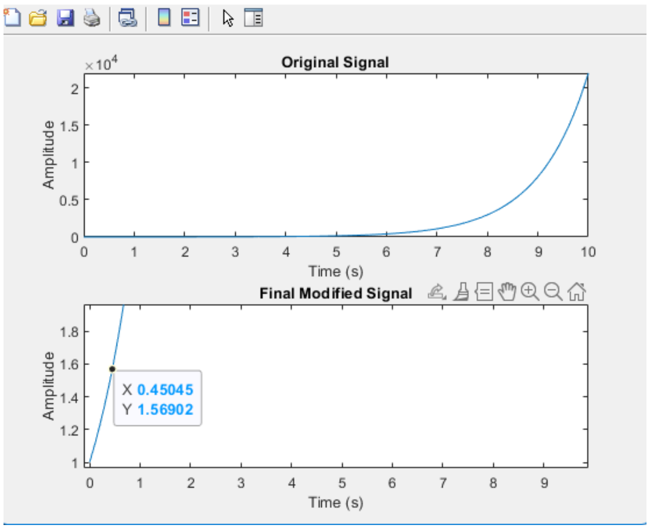


Test Case 4: Clipping on an Exponential Signal

Inputs:

```
>> main
Enter the sampling frequency of the signal: 100
Enter the start of time scale: 0
Enter the end of time scale: 10
Enter the number of break points: 0
Enter signal type (DC, Ramp, Polynomial, Exponential, Sinusoidal, Sinc, Triangle):
Exponential
Enter amplitude: 1
Enter exponent: 1
Available operations:
1. Scale (s)
2. Reverse (rev)
3. Shift (sh)
4. Expand (ex)
5. Compress (co)
6. Clip (cl)
7. Derivative (d)
8. None (n)
9. Stop (stop)
Select an operation number or code to perform on the signal or enter Stop to
finish: cl
Enter upper clipping value: 2
Enter lower clipping value: 1
Available operations:
1. Scale (s)
2. Reverse (rev)
3. Shift (sh)
4. Expand (ex)
5. Compress (co)
6. Clip (cl)
7. Derivative (d)
8. None (n)
9. Stop (stop)
Select an operation number or code to perform on the signal or enter Stop to
```

```
finish: stop
>>
```



Conclusion

The development of the General Signal Generator in MATLAB has successfully met the outlined objectives of this project, providing a robust tool for generating and manipulating a variety of signal types. Through the implementation of the application, we have demonstrated comprehensive capabilities in both the theoretical understanding of signal properties and the practical application of signal processing techniques.

Achievements

- **Versatile Signal Generation**: *The program can generate multiple types of signals including DC, Ramp, Polynomial, Exponential, Sinusoidal, Sinc, and Triangle, each over user-defined time intervals segmented by breakpoints. This flexibility allows for the exploration of complex signal structures in a controlled environment.*
- **Dynamic Signal Manipulation:** *Users can apply various transformations such as scaling, reversing, shifting, expanding, compressing, clipping, and differentiating. These operations are critical for studying signal behavior and for applications in real-world signal processing tasks.*
- **Interactive User Interface**: *The command-line interface provides clear and concise prompts, ensuring that users can easily understand and interact with the program to specify parameters and choose operations.*
- **Visual Feedback**: *Immediate plotting of the original and modified signals offers users visual confirmation of the effects of their manipulations, enhancing the learning and experimental process.*

Challenges and Solutions

- Error Handling: *Initially, operations that adjusted the signal length (such as expansion and compression) caused mismatches in vector lengths, leading to errors during plotting.*

This was resolved by implementing a two-step interpolation process that ensures the modified signal vector always matches the time vector in length.

- User Input Validation: *Ensuring robust input validation was critical to prevent runtime errors and logical faults. Rigorous checks and structured input prompts were added to guide the user effectively through the input process.*

In conclusion, this project has effectively reinforced theoretical concepts of signals and systems and provided a practical toolkit for signal analysis. The General Signal Generator is an invaluable educational tool, enhancing understanding of signal operations and fostering MATLAB proficiency. With potential future enhancements, it promises even greater applicability in academic and professional environments, solidifying its role as a fundamental resource for both students and practitioners.