Lesson 09: Fixed-Point Arithmetic

One day, you may have the great fortune of implementing a digital filter on an embedded device where you only have a few microseconds to filter before the next sample. In such a scenario, performing floating-point operations is impractical due to how much slower they are to ***Multiply–Accumulate (MAC)*** operations. To benefit from MAC operations, we turn to fixed-point arithmetic, but in the process, we open Pandora's box and must account for the substantial quantization erorr we've introduced. Luckily for us, MATLAB offers us a Fixed-Point Designer[1] for optimizing and implementing fixed-point algorithms.

Constructing Fixed-Point Types

```
+- sign bit
|
v
1 0 . 0 0 1 1 1 0
^     ^         ^
|     |         |
|     +----+----+
|          |    |
|          +- fraction
|               |
+-------+-------+
        |
        +- word length
```

Figure 1: Fixed-Point Value

First, we must tell MATLAB the type of fixed-point value we wish to construct:

```
T = numerictype(true, 32, 30)
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

[1] https://www.mathworks.com/help/fixedpoint/index.html

In our case, we've constructed a signed 32-bit fixed-point value with a fraction length of 30 bits, giving us a range from -2 to $2 - 2^{-30}$ in increments of 2^{-30}.

Now, to convert a `double` to a fixed-point value, we call `fi()`:

```
mu_0 = fi(4 * pi * 1e-7, T)
```

```
mu_0 =

   1.2564e-06

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Set Fixed-Point Math Settings

At times, we may wish to change MATLAB's default behavior when performing fixed-point math. Say we wanted to have two's complement overflow as opposed to saturation:

```
saturate = 1 / mu_0
```

```
saturate =

    2.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```



```
mu_0.OverflowMode = 'Wrap';
wrap = 1 / mu_0
```

```
wrap =

    1.9096

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

We can also apply these properties to new fixed-point types with `fimath()`:

```
F = fimath('OverflowMode', 'Wrap');
mu_r = fi(0.34, T, F)
```

```
mu_r =

    0.3400

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Fixed-Point Filter Design

For this part of the lesson, we'll be working with the brake pressure sensor data from Assignment 07:[2]

[2] `https://raw.githubusercontent.com/jacobkoziej/jk-ece210/master/src/assignments/07-under-pressure.d/40p_1000ms.csv`

```
RESOLUTION = 2^12;

fs = 80e3;

M = readmatrix("40p_1000ms.csv");

t = M(:, 1) ./ fs;
s = M(:, 2) ./ (RESOLUTION - 1);
```

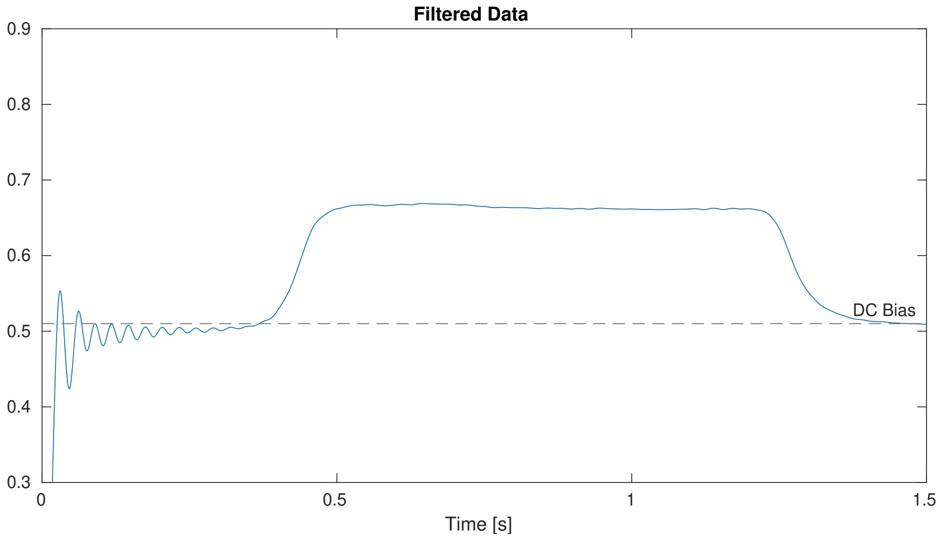
Since fixed-point values are objects, we can't use our standard filtering functions on them, so we turn to filter objects:

Aside— We use the Direct Form I structure over Direct Form II to gain more numerical stability at the cost of using double the amount of state variables.

```
Fp = 35;
Fs = 45;
Rp = 1;
Rs = 40;

D = fdesign.lowpass(Fp, Fs, Rp, Rs, fs);
LPF = design(D, 'ellip', 'SystemObject', true);

LPF.Structure = 'Direct form I';
LPF.DenominatorAccumulatorDataType = T;
LPF.OutputDataType = T;
```

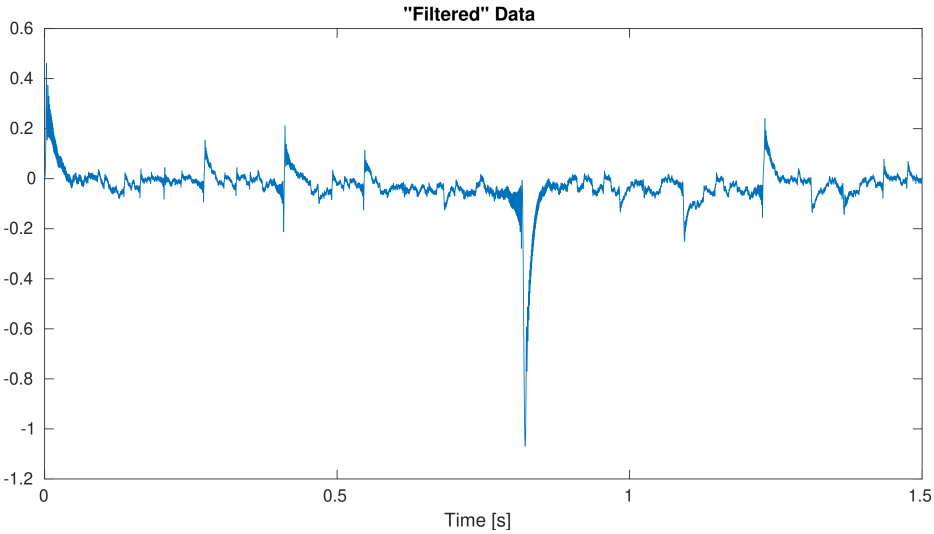
We can then apply the filter to our quantized data like so:

```
s_filtered = LPF(fi(s, T));

figure;
plot(t, s_filtered);
title('"Filtered" Data');
xlabel('Time [s]');
```

Let's half our bit allocation to see how our filter performs:

```
T = numerictype(true, 16, 14)

D = fdesign.lowpass(Fp, Fs, Rp, Rs, fs);
LPF = design(D, 'ellip', 'SystemObject', true);

LPF.Structure = 'Direct form I';
LPF.DenominatorAccumulatorDataType = T;
LPF.OutputDataType = T;

s_filtered = LPF(fi(s, T));

f = figure;
plot(t, s_filtered);
title('"Filtered" Data');
xlabel('Time [s]');
```



Ouch, it looks like the quantization error was too much to handle! To combat this, we can decrease our filter's requirements to avoid higher orders and hence numerical instability in the form of overflows, but keep in mind this isn't a full-proof solution.

