Assignment1

May 10, 2021

1 Assignment 1 – 1st-order modulator

By Arthur Admiraal, with feedback provided by Yu Chieh

This isn't stricly a `.m` file, but I believe using Python is in the spirit of the exercise. It allowed me to use a Jupyter notebook here, which I believe is much more readable. Most of the python here is similar to Matlab, but I added ample comments in case you're not familiar. You can also view an interactive version of this notebook here.

1.1 General stuff

This is the siulation framework I'm using for all the sub-exercises. The simulation is executed and stored in the `ModulatorSimulation` object. When it is instantiated, the simulation is run on a provided input signal. The output signals can then be retrieved at a later point. The decimation filtering is performed in short code snippets in the sub-exercises themselves, but I created two convenience functions to get sinc1 and sinc2 impulse response functions. I also defined functions to plot the dc transfer and INL, which I'll use throughout the sub-exercises.

[7]:
```
import numpy as np              # numpy is a Python library for deadling with␣
↪arrays of numbers
import matplotlib.pyplot as plt # matpltotlib will enable plotting of said arrays
```

[8]:
```
class ModulatorSimulation:
    def __init__(self, input_signal): # initialises the data
        # sanity checks of input
        assert np.ndim(input_signal) == 1, "Input signal is not 1D!"

        # store input signal in object
        self.input_signal       = input_signal

        # find length of signals, for allocation
        length                  = np.size(input_signal)

        # initialise output signals
        self.accumulator_signal = np.zeros(length)
        self.output_signal      = np.zeros(length)
```

```
        # define convenience 'time' signal
        self.sample_numbers     = np.arange(length)

    def plot(self): # plot all signals of the modulator
        # sets the size of the figure to the full width of a cell
        plt.figure(figsize=(14, 5), dpi=80)

        # plot the input, output, and internal signals
        plt.plot(self.sample_numbers, self.input_signal, 'k--')         # plot␣
↪input as dashed black line
        plt.plot(self.sample_numbers, self.accumulator_signal)          # plot␣
↪accumulator as connected blue line
        plt.step(self.sample_numbers, self.output_signal, where='post') # plot␣
↪output signal as stepped orange line

        # add labels and legend for clarity
        plt.xlabel("Sample number")
        plt.ylabel("Signal amplitude")
        plt.legend(["Input", "Accumulator", "Output"])

        # push the plot to the screen
        plt.show()

    def get_filtered_output(self): # convenience function to return sinc1␣
↪filtered output
        return np.mean(self.output_signal[1:]) # returns mean of output signal,␣
↪skipping initial condition

    def get_raw_output(self):
        return self.output_signal[1:] # returns output signal, skipping initial␣
↪condition

    @classmethod
    def from_signal(cls, input_signal, ic_accumulator=0, ic_output=1, p=1): #␣
↪define
        # arguments:
        #  input_signal: a 1D array containing the input signal to the modulator
        #  ic_accumulator: initial condition of accumulator signal. Must be␣
↪within [1, -1].
        #  ic_output: initial condition of output signal. Must be either 1 or -1.
        #  p: leakiness of accumulator. 1 for a perfect accumulator, leaks for␣
↪values in [0, 1).

        # instantiate this class to allocate space for signals, and store input␣
↪signal in the class
        sim = cls(input_signal)
```

```
        # set initial conditions
        sim.accumulator_signal[0] = ic_accumulator
        sim.output_signal[0]      = ic_output

        # run through sample times
        for n in sim.sample_numbers[1:]: # skip the initial condition
            sim.accumulator_signal[n] = p*sim.accumulator_signal[n-1] + sim.
↪input_signal[n-1] - sim.output_signal[n-1]
            sim.output_signal[n]      = 2*(sim.accumulator_signal[n] >= 0) - 1

        return sim

    @classmethod
    def from_dc(cls, dc, length=100, ic_accumulator=0, ic_output=1, p=1):
        # convenience function to create modulator simulation for dc input
        # add 1 to length for initial condition
        return cls.from_signal(dc * np.ones(length+1), ic_accumulator=0,␣
↪ic_output=1, p=p)
```

[9]:
```
def get_sinc1(N=8):
    # returns sinc1 impulse response: pulse of length N and unit area
    return np.ones(N)/N

def get_sinc2(N=8):
    # returns sinc2 impulse response: convolution of two sinc1 pulses
    # In order to get odd lengths of impulse responses, the sinc1 pulses have to␣
↪be different sizes.
    # These are calcualted in the form of N1 and N2. Note that 1 is added to the␣
↪length of N2, since
    # convolution product is 1 shorter than the sum of the lengths of both␣
↪impulse responses.
    N1 = int(np.floor(N/2))
    N2 = int(N+1-N1)

    # get the sinc1 impulse responses
    sinc1_small = get_sinc1(N1)
    sinc1_large = get_sinc1(N2)

    # convolve to yield the desired sinc2 impulse response
    return np.convolve(sinc1_small, sinc1_large, mode='full')
```

[10]:
```
# plot an overview of the DC transfer of a modulator
def _plot_dc_transfer_overview(input_list, output_list):
    plt.plot(input_list, output_list)
    plt.xlabel("Input signal")
    plt.ylabel("Output signal")
```

```
    plt.title("Full transfer")
    plt.xlim([-1,1])
    plt.ylim([-1,1])

# plot an zoomed-in view DC transfer of a modulator
def _plot_dc_transfer_zoom_view(input_list, output_list, LSB=2/255, xzoom=0.01,␣
 ↪plot_func=plt.scatter):
    # cast the lists of data points to numpy arrays in order to perform
    # arithmetic on them later, which python lists don't allow
    input_list  = np.array(input_list)
    output_list = np.array(output_list)

    # calculate the indices of the data points within the zoom range
    zoom_indices = (input_list > -xzoom) & (input_list < xzoom)

    # plot the data points within the zoom range in terms of LSBs
    plot_func(input_list[zoom_indices] / LSB, output_list[zoom_indices] / LSB)

    # add labels and title
    plt.xlabel("Input signal (LSB)")
    plt.ylabel("Output signal (LSB)")
    plt.title(r"Zoomed in to $x\in \{{{}; {}\}}$".format(-xzoom, xzoom))

# plot the dc transfer of a modulator:
#  - plot an overview of the entire transfer function over [-1, 1]
#  - plot a zoom view between [-xzoom] and [xzoom]
def plot_dc_transfer(input_list, output_list, LSB=2/255, xzoom=0.01):
    # set the size of the figure to the full width of a cell
    plt.figure(figsize=(14, 5), dpi=80)

    # add overview as left subplot
    plt.subplot(1,2,1)
    _plot_dc_transfer_overview(input_list, output_list)

    # add zoom view as right subplot
    plt.subplot(1,2,2)
    _plot_dc_transfer_zoom_view(input_list, output_list, LSB, xzoom)

    # push the plot to the screen
    plt.show()

# plot the dc transfer of a modulator:
#  - plot an overview of the entire transfer function over [-1, 1]
#  - plot a zoom view between [-xzoom] and [xzoom] from a seperate, more␣
 ↪detailed dataset
def plot_dc_transfer_with_zoom_samples(input_list, output_list,␣
 ↪zoom_input_list, zoom_output_list, LSB=2/255, xzoom=0.01):
```

```
    # set the size of the figure to the full width of a cell
    plt.figure(figsize=(14, 5), dpi=80)

    # add overview as left subplot
    plt.subplot(1,2,1)
    _plot_dc_transfer_overview(input_list, output_list)

    # add zoom view as right subplot
    plt.subplot(1,2,2)
    _plot_dc_transfer_zoom_view(zoom_input_list, zoom_output_list, LSB, xzoom,␣
 ↪plot_func=plt.plot)

    # push the plot to the screen
    plt.show()
```

[11]:
```
# get the INL of a modulator
def get_INL(input_list, output_list, LSB=2/255):
    # cast the lists of data points to numpy arrays in order to perform
    # arithmetic on them later, which python lists don't allow
    input_list  = np.array(input_list)
    output_list = np.array(output_list)

    return (output_list-input_list) / LSB

# plot the INL of a modulator
def plot_INL(input_list, output_list, LSB=2/255):
    # perform the plotting
    plt.figure(figsize=(14, 5), dpi=80)
    plt.plot(output_list, get_INL(input_list, output_list, LSB) )
    plt.xlabel("Output code")
    plt.ylabel("INL")
    plt.show()
```

[12]:
```
def print_quantisation_steps(output_list):
    print("There are {} unique quantisation steps in the output data.".format(np.
 ↪size(np.unique(output_list))))
def print_INL(input_list, output_list, LSB=2/255):
    INL = get_INL(input_list, output_list, LSB)
    min_INL = np.min(INL)
    max_INL = np.max(INL)
    INL_range  = max_INL - min_INL
    INL_offset = (min_INL + max_INL)/2
    print("INL within [{:.2f}, {:.2f}] LSB for an effective resolution of {:.2f}␣
 ↪LSB with an offset of {:.2f} LSB.".format(min_INL, max_INL, INL_range,␣
 ↪INL_offset))
```

1.2 A

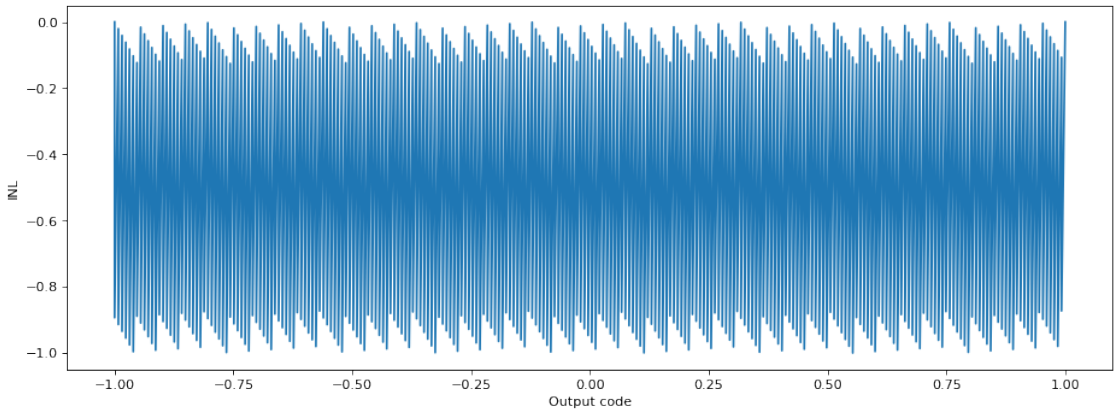
Design a 1st-order modulator with a sinc1 decimation filter. What filter length is necessary to ensure 8-bit resolution over the modulator's entire input range?

For a DC input, the sinc1-filtered modulator essentially acts as a (scrambled) thermometer coder: increases of the input signal cause the amount of outputted ones to increase monotonically. A resolution of 8 bits has 256 quantisation steps. Since it includes 0 as a step, 255 step thermometer encoding is needed to reach an equal amount of quantisation steps. Hence, the length of the sinc1 filter must be 255. This reasoning is validated in the following simulation.

[ ]:
```
input_list  = np.linspace(-1, 1, 2000) # also test at annoying points between␣
↪quantisation steps
output_list = np.zeros(np.size(input_list))

for idx,dc in enumerate(input_list):
    simulation = ModulatorSimulation.from_dc(dc, length=255)
    output_list[idx] = np.mean(simulation.get_raw_output())
```

[ ]:
```
plot_dc_transfer(input_list, output_list)
```

[15]:
```
plot_INL(input_list, output_list)
```



[45]:
```
print_quantisation_steps(output_list)
```

```
There are 256 unique quantisation steps in the output data.
```

Note that the above analysis assumed that the readout of the delta sigma modulation is synchronised with the start of the signal. In a practical system, this would mean that the modulator would have to run for a set amount of cycles on a sampled-and-held signal. While synchronisation is certainly possible, it may not be available. In that case, the added uncertainty means a bigger filter is needed to get the same amount of resolution.

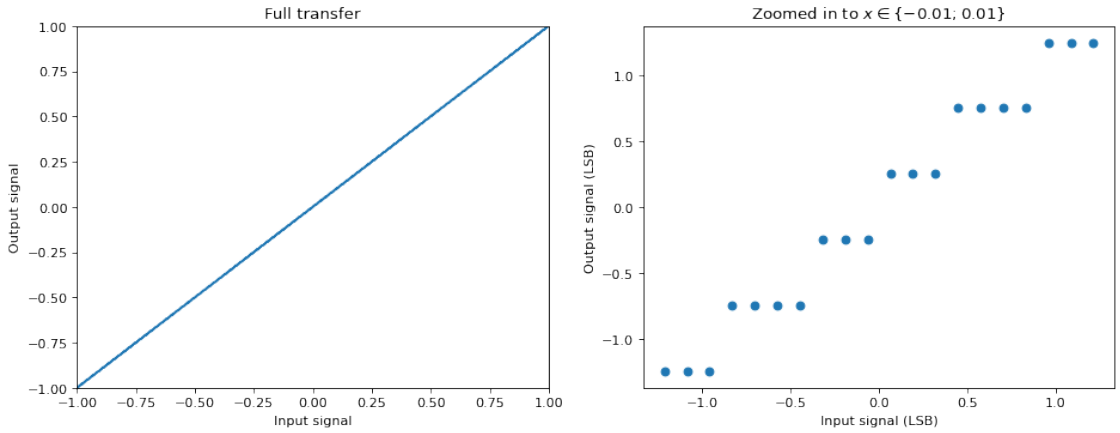
In absence of synchronisation, the amount of ones cannot only change due to the limit cycles getting

a higher average value, but also due to shorther limit cycles shifting the phase of the phase of the limit cycles in the filter window. As a result, the amount of ones does not necessarily increase monotonically, and the INL is approximately doubled. It is difficult to say exactly by how much the INL is increased, but a bound of a factor of 2 can be found. Because of that, I used twice the filter length. This reasoning is validated in the upcoming simulation.

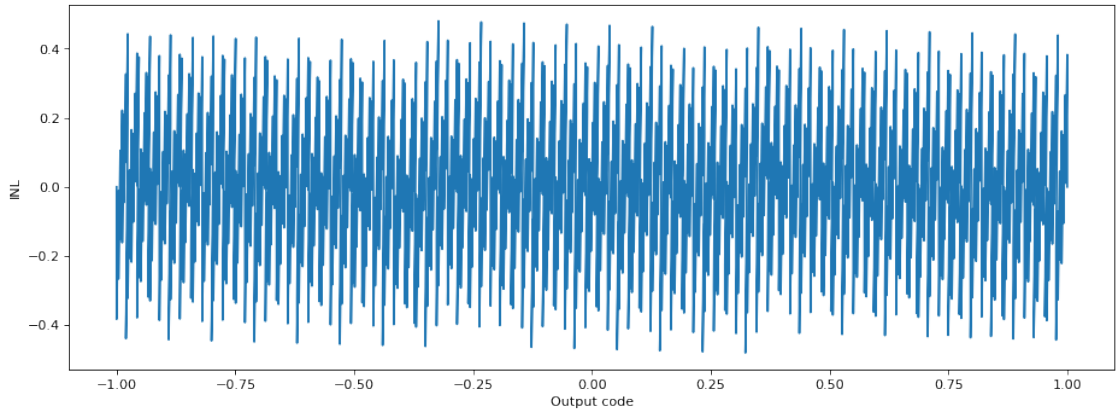
In this case and the coming cases, it no longer makes sense to measure resolution by the amount of unique quantisation steps. Rather, the effective resolution should be determined from the INL.

[161]:
```
input_list  = np.linspace(-1, 1, 2000) # also test at annoying points between␣
↪quantisation steps
output_list = np.zeros(np.size(input_list))

for idx,dc in enumerate(input_list):
    # take longer simulation length than filter length, so that the filter is␣
↪not synchronised to the beginning of the simulation
    simulation = ModulatorSimulation.from_dc(dc, length=600)
    output_list[idx] = np.mean(simulation.get_raw_output()[-511:])
```

[162]:
```
plot_dc_transfer(input_list, output_list, xzoom=0.01)
```



[163]:
```
plot_INL(input_list, output_list)
```



[164]:
```
print_quantisation_steps(output_list)
print_INL(input_list, output_list)
```

```
There are 512 unique quantisation steps in the output data.
INL within [-0.48, 0.48] LSB for an effective resolution of 0.96 LSB with an
offset of 0.00 LSB.
```

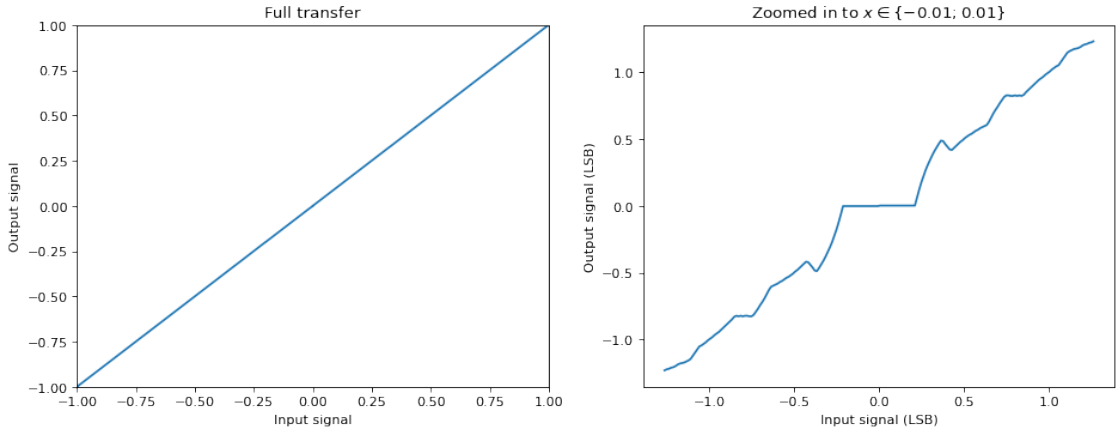
1.3 B

Repeat exercise A for a sinc2 FIR filter. With a sinc2 filter, not all bits ones are counted equally. Due to this weighting, an additional one could also knock other ones to lower weights. Like the unsynchronised sinc1 filtered modulator, this gives non-monotonic behaviour, increasing the spread by a factor of at most 2. Because of that, twice the filter length is required to guarantee the 8 bit effective resolution.

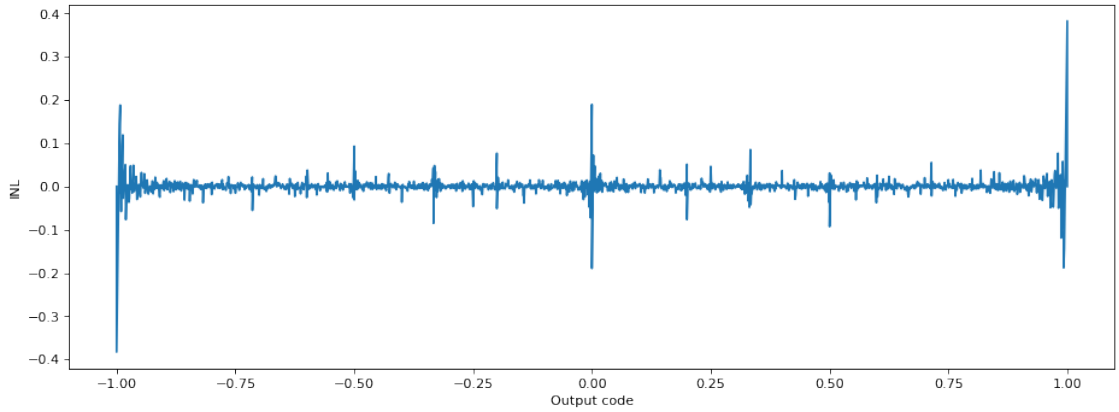
[165]:
```
input_list  = np.linspace(-1, 1, 2000) # also test at annoying points between␣
 ↪quantisation steps
output_list = np.zeros(np.size(input_list))

sinc2 = get_sinc2(511)

for idx,dc in enumerate(input_list):
    simulation       = ModulatorSimulation.from_dc(dc, length=600)
    output           = simulation.get_raw_output()
    output_list[idx] = np.convolve(sinc2, output, mode='valid')[-1]

# do separate simulation for more detailed zoom view
zoom_input_list  = np.linspace(-0.01, 0.01, 200) # also test at annoying points␣
 ↪between quantisation steps
zoom_output_list = np.zeros(np.size(zoom_input_list))

for idx,dc in enumerate(zoom_input_list):
```

```
    simulation              = ModulatorSimulation.from_dc(dc, length=600)
    output                  = simulation.get_raw_output()
    zoom_output_list[idx] = np.convolve(sinc2, output, mode='valid')[-1]
```

[166]:
```
plot_dc_transfer_with_zoom_samples(input_list, output_list, zoom_input_list,␣
↪zoom_output_list)
```



[167]:
```
plot_INL(input_list, output_list)
```



[168]:
```
print_quantisation_steps(output_list)
print_INL(input_list, output_list)
```

```
There are 1990 unique quantisation steps in the output data.
INL within [-0.38, 0.38] LSB for an effective resolution of 0.77 LSB with an
offset of 0.00 LSB.
```

The simulation results show that the choice of twice the synchronised sinc1 filter length seems to leave performance on the table. This can be explained by the fact that this filter length guarantees

bounded INL, but does not guarantee an optimum.

Two other interesting effects can be observed. First of, a dead zone can clearly be seen in the zoom view of the DC transfer. This can be explained by the fact that for those small deviations, the output sequence of the modulator within the filter window won't change, since the accumulated error is too small. A similar effect can be observed in the INL plot towards the extremities of the output range. For small deviations, the output sequence hardly changes, and if it does it does so only towards the end ofo the filter window. In that case, the filter doesn't smooth out as effectively, causing large INL swings. Closer to the middle of the output range, the limit cycles have a higher frequency, so that the ones are better spread out over the filter window, giving better smoothing and smaller INL swings.

1.4 C

Use now a 1024-tap sinc1 decimation filter. What is the minimum DC gain of the accumulator in order to achieve 9-bit resolution?

Finding the gain This is a very complex system to model, so I brute force it by running simulations to find the effective resolution for a large amount of gains, and then interpolate between those results to find which gain is required for the requested resolution.

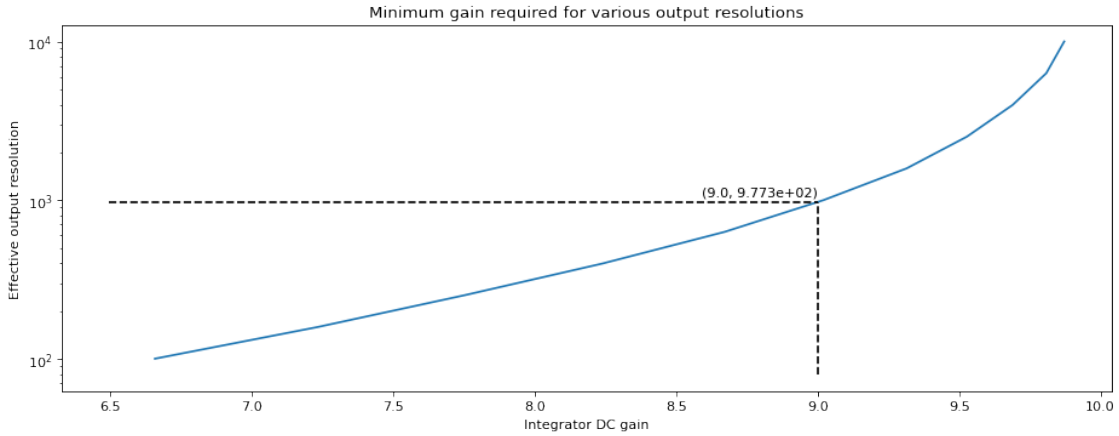
[91]:
```
input_list  = np.linspace(-1, 1, 2000) # also test at annoying points between␣
↪quantisation steps
output_list = np.zeros(np.size(input_list))

gain_list       = np.logspace(2, 4, 11) # test 11 gains between 10^2 and 10^4
resolution_list = np.zeros(np.size(gain_list))

# iterate through 20 gain settings and find resolution
for idx2,gain in enumerate(gain_list):
    print("Iteration {}:\tSimulating gain of {:.2e}...".format(idx2, gain),␣
↪end="")
    p = gain/(gain+1)
    for idx,dc in enumerate(input_list):
        simulation = ModulatorSimulation.from_dc(dc, length=1024, p=p)
        output_list[idx] = np.mean(simulation.get_raw_output())

    INL                 = get_INL(input_list, output_list, LSB=2) # LSB=2 gives␣
↪decimal fractions of range instead of in LSB
    effective_resolution = -np.log2(np.max(INL) - np.min(INL))

    resolution_list[idx2] = effective_resolution

    print("complete")
```

```
Iteration 0:    Trying gain of 1.00e+02…complete
Iteration 1:    Trying gain of 1.58e+02…complete
```

```
Iteration 2:    Trying gain of 2.51e+02…complete
Iteration 3:    Trying gain of 3.98e+02…complete
Iteration 4:    Trying gain of 6.31e+02…complete
Iteration 5:    Trying gain of 1.00e+03…complete
Iteration 6:    Trying gain of 1.58e+03…complete
Iteration 7:    Trying gain of 2.51e+03…complete
Iteration 8:    Trying gain of 3.98e+03…complete
Iteration 9:    Trying gain of 6.31e+03…complete
Iteration 10:   Trying gain of 1.00e+04…complete
```

[ ]:
```
# interpolate to find the required gain
desired_resolution = 9
required_gain      = np.interp(desired_resolution, resolution_list, gain_list)
```

[129]:
```
# plot the resulting resolution, gain diagram
plt.figure(figsize=(14, 5), dpi=80)
plt.semilogy(resolution_list, gain_list)
plt.plot(desired_resolution * np.ones(2), [plt.gca().get_ylim()[0],␣
 ↪required_gain], 'k--')
plt.plot([plt.gca().get_xlim()[0], desired_resolution], required_gain * np.
 ↪ones(2), 'k--')
plt.text(desired_resolution, required_gain*1.1, '({:.1f}, {:.3e})'.
 ↪format(desired_resolution, required_gain), ha='right')
plt.xlabel("Integrator DC gain")
plt.ylabel("Effective output resolution")
plt.title("Minimum gain required for various output resolutions")
plt.show()
```

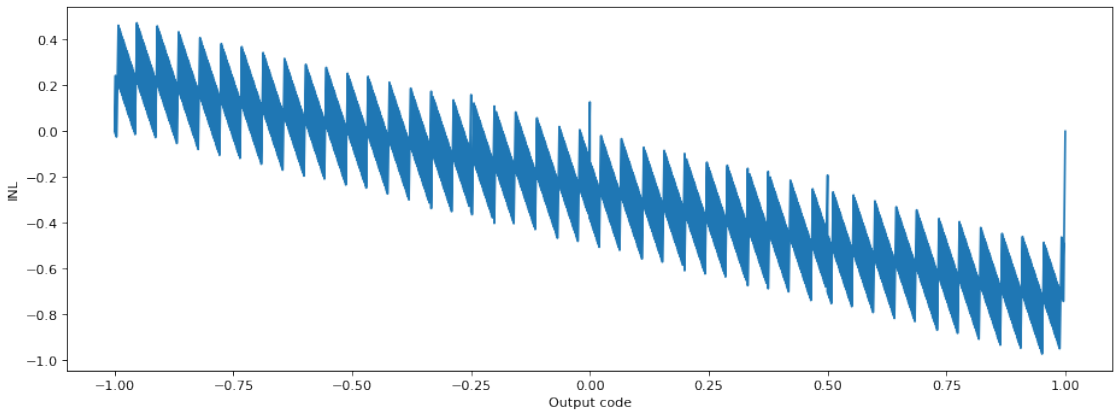


Verification I now run a simulation to validate that the gain found through the brute force simulations does indeed give the required resolution.

[173]:
```
input_list  = np.linspace(-1, 1, 2000) # also test at annoying points between␣
↪quantisation steps
output_list = np.zeros(np.size(input_list))

p = required_gain/(required_gain+1)
for idx,dc in enumerate(input_list):
    simulation = ModulatorSimulation.from_dc(dc, length=1024, p=p)
    output_list[idx] = np.mean(simulation.get_raw_output())
```

plot_dc_transfer(input_list, output_list, LSB=2/511)

[174]:
```
plot_INL(input_list, output_list, LSB=2/511)
```



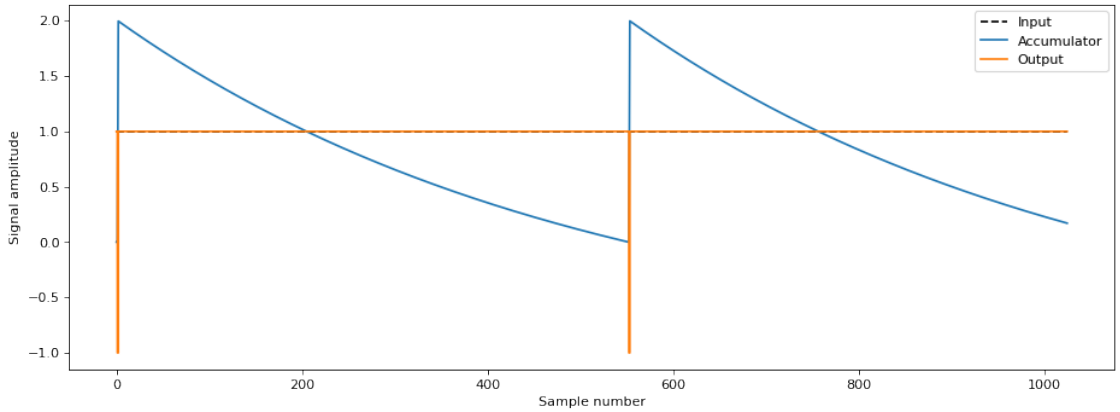
[175]:
```
print_quantisation_steps(output_list)
print_INL(input_list, output_list)
```

```
There are 1024 unique quantisation steps in the output data.
INL within [-0.49, 0.24] LSB for an effective resolution of 0.72 LSB with an
offset of -0.12 LSB.
```

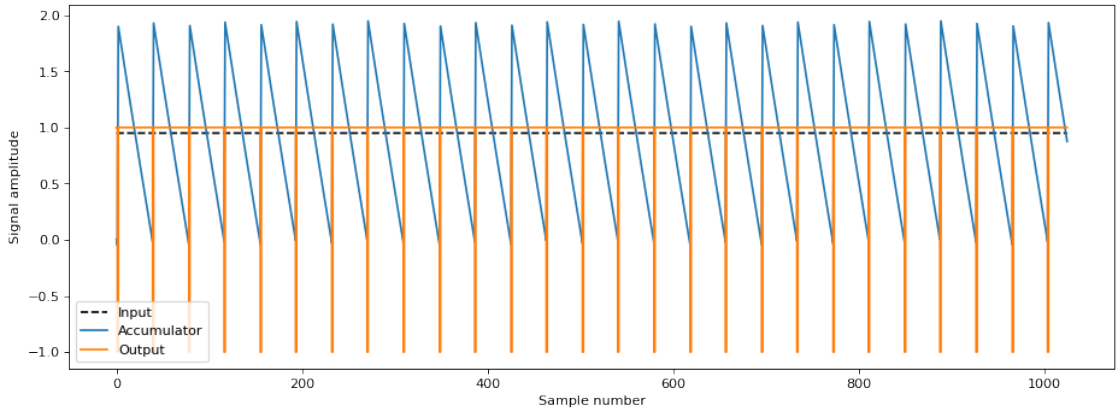
Interestingly, like the sinc2 filtered modulator, the INL graph of the leaky modulator exhibits large swings in INL and a dead zone in the DC transfer zoom view. This can be explained by the fact that small differences slowly leak away, shifting certain transitions between codes. This is especially prevalent towards the extremities and in the middle of the range.

Another odd effect is the slow ramp from high INL to low INL. This can be explained by the fact that for values closer to the extremities of the range, the frequency of the limit cycles increases, increasing the influence of the leakiness. This can be seen from simulations close to and far away from the extremities:

[179]:
```
simulation = ModulatorSimulation.from_dc(0.998, length=1024, p=p)
simulation.plot()
```



[180]:
```
simulation = ModulatorSimulation.from_dc(0.95, length=1024, p=p)
simulation.plot()
```



It can clearly be seen that the slopes of the accumulator in the second simulation, farther away from the extremities, have a more linear behaviour. This gives less deviation from the ideal behaviour, and thus lower INL. This change in deviation from ideal behaviour with distance to the extremities gives the linear slope in INL

1.5 D

Repeat exercise C for a sinc2 FIR filter.

Finding the gain This is another systems that would be very complex system to model, so I use the brute force method again.

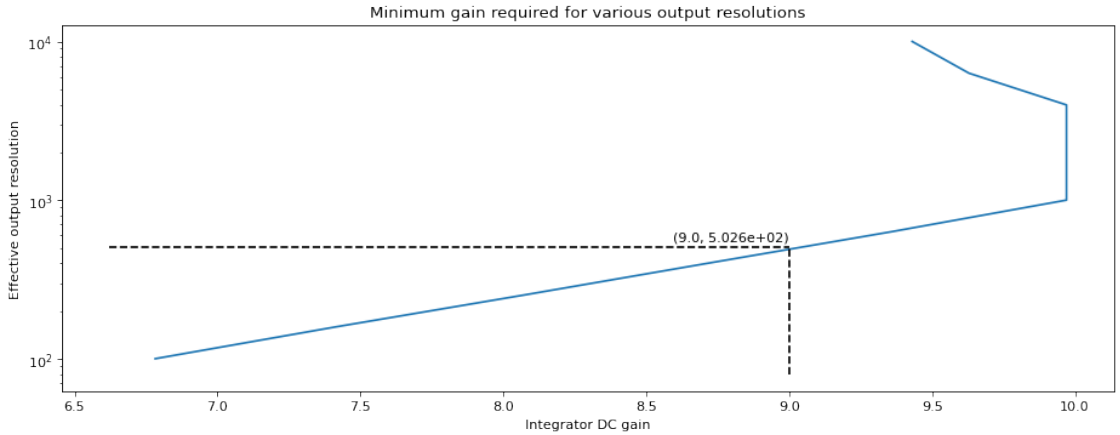
[140]:
```
input_list  = np.linspace(-1, 1, 2000) # also test at annoying points between␣
↪quantisation steps
output_list = np.zeros(np.size(input_list))

gain_list       = np.logspace(2, 4, 11) # test 11 gains between 10^2 and 10^4
resolution_list = np.zeros(np.size(gain_list))

sinc2 = get_sinc2(1024)

# iterate through 20 gain settings and find resolution
for idx2,gain in enumerate(gain_list):
    print("Iteration {}:\tSimulating gain of {:.2e}...".format(idx2, gain),␣
↪end="")
    p = gain/(gain+1)
    for idx,dc in enumerate(input_list):
        simulation       = ModulatorSimulation.from_dc(dc, length=1024, p=p)
        output           = simulation.get_raw_output()
        output_list[idx] = np.convolve(sinc2, output, mode='valid')[-1]

    INL                 = get_INL(input_list, output_list, LSB=2) # LSB=2 gives␣
↪decimal fractions of range instead of in LSB
    effective_resolution = -np.log2(np.max(INL) - np.min(INL))

    resolution_list[idx2] = effective_resolution

    print("complete")
```

```
Iteration 0:    Simulating gain of 1.00e+02...complete
Iteration 1:    Simulating gain of 1.58e+02...complete
Iteration 2:    Simulating gain of 2.51e+02...complete
Iteration 3:    Simulating gain of 3.98e+02...complete
Iteration 4:    Simulating gain of 6.31e+02...complete
Iteration 5:    Simulating gain of 1.00e+03...complete
Iteration 6:    Simulating gain of 1.58e+03...complete
Iteration 7:    Simulating gain of 2.51e+03...complete
Iteration 8:    Simulating gain of 3.98e+03...complete
Iteration 9:    Simulating gain of 6.31e+03...complete
Iteration 10:   Simulating gain of 1.00e+04...complete
```

[141]:
```
# interpolate to find the required gain
desired_resolution = 9
required_gain      = np.interp(desired_resolution, resolution_list, gain_list)
```

[142]:
```
# plot the resulting resolution, gain diagram
plt.figure(figsize=(14, 5), dpi=80)
plt.semilogy(resolution_list, gain_list)
```

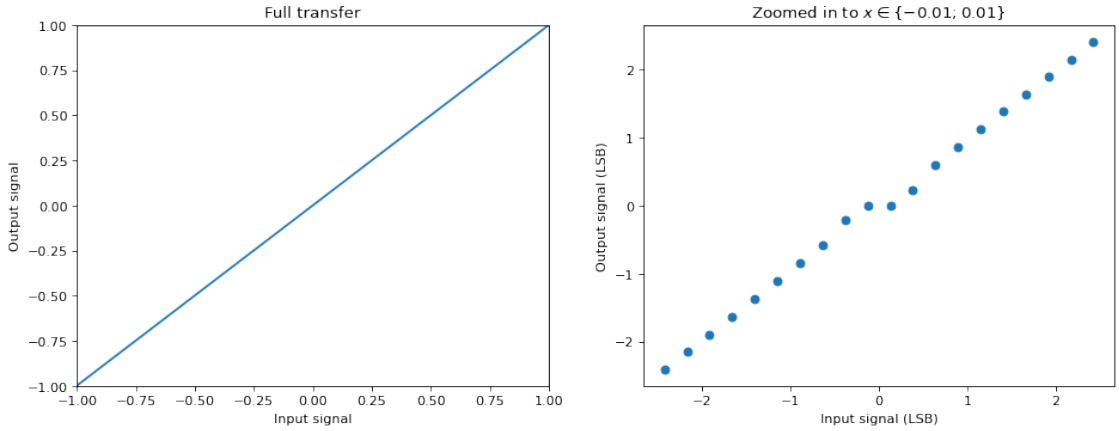
```
plt.plot(desired_resolution * np.ones(2), [plt.gca().get_ylim()[0],␣
 ↪required_gain], 'k--')
plt.plot([plt.gca().get_xlim()[0], desired_resolution], required_gain * np.
 ↪ones(2), 'k--')
plt.text(desired_resolution, required_gain*1.1, '({:.1f}, {:.3e})'.
 ↪format(desired_resolution, required_gain), ha='right')
plt.xlabel("Integrator DC gain")
plt.ylabel("Effective output resolution")
plt.title("Minimum gain required for various output resolutions")
plt.show()
```



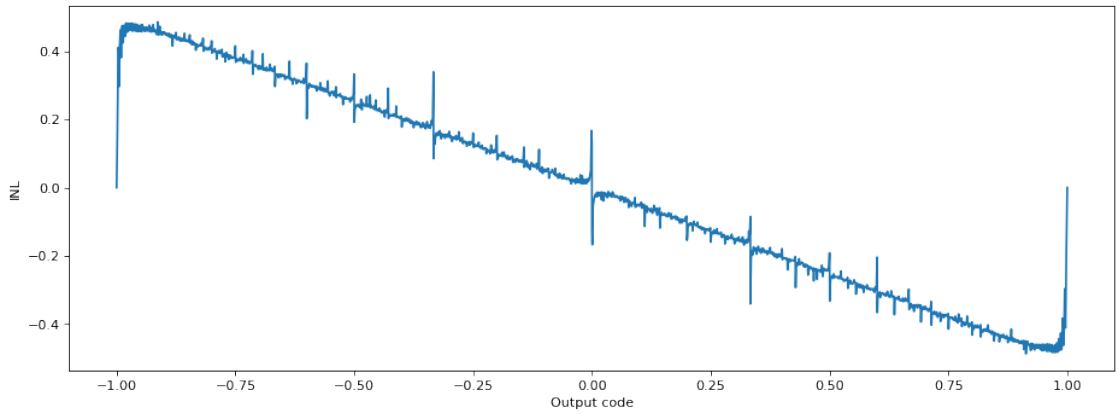
Verification Again, I run a simulation to validate that the gain found through the brute force simulations does indeed give the required resolution.

[146]:
```
input_list  = np.linspace(-1, 1, 2000) # also test at annoying points between␣
 ↪quantisation steps
output_list = np.zeros(np.size(input_list))

sinc2 = get_sinc2(1024)

p = required_gain/(required_gain+1)
for idx,dc in enumerate(input_list):
    simulation       = ModulatorSimulation.from_dc(dc, length=1024, p=p)
    output           = simulation.get_raw_output()
    output_list[idx] = np.convolve(sinc2, output, mode='valid')[-1]
```

[147]:
```
plot_dc_transfer(input_list, output_list, LSB=2/511)
```



[148]:
```
plot_INL(input_list, output_list, LSB=2/511)
```



[149]:
```
print_quantisation_steps(output_list)
print_INL(input_list, output_list)
```

```
There are 1998 unique quantisation steps in the output data.
INL within [-0.49, 0.49] LSB for an effective resolution of 0.97 LSB with an
offset of -0.00 LSB.
```

Like the sinc2 filtered modulator, and the sinc1 filtered modulator, the modulator exhibits a dead zone, spikes in the INL, and a slope of the INL. This can be explained by a combination of the effects discussed before.