University of Glasgow | James Watt School of Engineering

1 Digital Communications 4: Carrier Recovery using Costas Loop

1.1 Introduction

This coding project will introduce you to Carrier Recovery. We will be coding in the python programming language in order to make use of a bespoke library for later projects. If you are using your own computer, make sure the Python libraries `scipy`, `numpy`, and `matplotlib` are installed. It is recommended that you use a suitable IDE for your project, such as Spyder, PyCharm or Visual Studio. Tip: if you are using Spyder, you can display, manipulate and save graphics in separate windows by setting

```
Tools>Preferences>IPython console>Graphics>Backend
```

from `Inline` to `Automatic`.

Each project will be scheduled over a two week period, within which there will be 2 scheduled online consultation sessions where you will be able to ask teaching staff for guidance. The project should be written up as a short report describing what you've done and the results you have taken along with any conclusions that you draw. Include your python code(s) in the appendices. Make sure your name and student ID number is on the report. The report should be uploaded to the Moodle assignment by the stated deadline, either using Moodle's inbuilt html editor, or as a single PDF file.

1.2 VCO Cordic Digital Clock

Code an algorithm for a voltage controlled digital clock. The current state of the clock is signified by 2 floating point numbers representing the in-phase and quadrature components sampled values. Then each successive state is obtained by the Cordic transformation using $\cos f_0$ and $\sin f_0$, where $f_0 = f_i + \alpha v$ is the scaled voltage dependent frequency, including a linear applied voltage dependence. Take your previous value for the carrier frequency as the initial value f_i. Check that your clock code is functional by inspecting a plot of the sampled in-phase and quadrature components with sample number.

1.3 Demodulation with Carrier Recovery

It is recommended that you take as your starting point for remainder of this project a copy of your first laboratory BPSK Modulation/Demodulation code. Your input data will remain as the 24 bit representation of your student ID number. Perform modulation in your nested loops as before. The Costas loop involves mixing the received modulated waveform with the VCO outputs, and then low-pass filtering the results. This does require a bit more thought than in the simple demodulation examples, as the VCO is being dynamically driven whilst the digital filtering is taking place, so the `lfilter` function cannot be simply deployed as in the previous

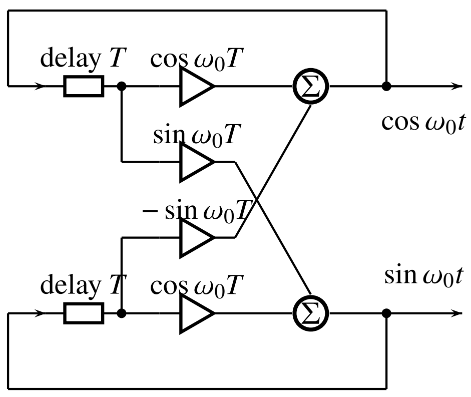

Figure 1: Digital Clock by Cordic Algorithm

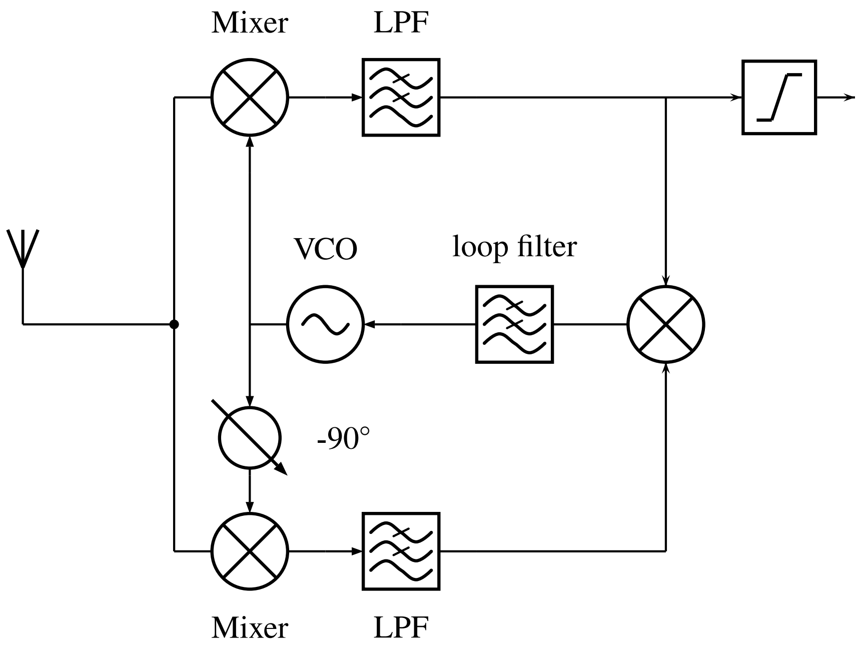

Figure 2: Block diagram of a Classical Costas Loop

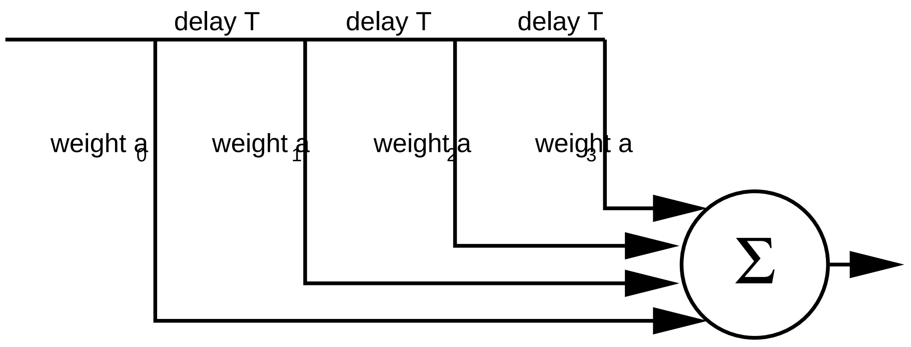

Figure 3: FIR Digital Filter

exercise. Therefore we should consider the discrete convolution that underlies digital filtering using an FIR.

$$[f \otimes a]\,(n) = \sum_{i=0}^{N-1} f(n-i)a(i)$$

So to get the nth sampled filter result we need the $f(n)$ to $f(n+1-N)$ filter inputs and the $a(0)$ to $a(N-1)$ tap weights, with N being the number of filter taps. Therefore we need an array (for each branch) which stores the last N LPF input samples, and this array should be initialised before the demodulation loops.

```
c_mixed = np.zeros(numtaps)
for ...
    for ...
        c_mixed[:] = np.append(c_mixed[1:], c_lpf_input)
        c_lpf_output = np.sum(b1*c_mixed)
```

We will use low-pass FIR filters in the Costas loop, so obtain the tap weights in the initialisation as previously, but with a flip in the ordering to account that it is the last term in `c_mixed` defined above which corresponds to zero delay, e.g.

```
# low-pass filter
numtaps = 64
b1 = np.flip(signal.firwin(numtaps, 0.1))
```

The Costas Loop block diagram in figure **??** shows that the two LPF outputs are multiplied at a mixer,

```
        volt = c_lpf_output*s_lpf_output
```

and then passed through a loop filter. As we have already employed low pass filters with a high rolloff-rate, it is normally not necessary to include a further frequency selective element, but we we should ensure that we set the gain/attenuation appropriately through the VCO α parameter, so you may need to experiment till you find a suitable value.

The final step in the demodulation of digital data is to select an appropriate sample point in `c_lpf_output`, and then use a thresholding function to convert floating point into boolean. We shall again use the most obvious sample point by taking the midpoint of the bit, i.e. `i*bit_len+bit_len//2`. Therefore construct a for loop over `Nbits` and applying a threshold to the bit midpoint. In this case an appropriate threshold function is the heaviside (step) function

```
rx_bin[i] = np.heaviside(c_lpf_output[i], 0)
```

Finally, compare your output binary data with your input data. Do you notice any issues?

You should observe a delay in the output data when compared to the input data. This is because the digital filter comprises delays within each tap, and the filter designs resulting from `signal.firwin` normally have an overall delay of `numtaps//2`. Account for this delay in your code; you will need to add at least `numtaps//2` dummy samples to the end of your mixed signal data prior to filtering.

You will also find that it takes a number of symbols for the Costas loop to lock on the input. Therefore you will also need to preface your digital data with a number of dummy symbols to cover this locking delay. You may want to include a unique pattern so that your data can be identified, say a number of zero bits followed by a single one bit.

1.4 Random phase and frequency

Once you have obtained satisfactory locking, then try locking to a carrier with random phase and frequency offset. Ensure the `numpy random` library is loaded.

```
from numpy import random
```

Then modify the carrier phase and frequency (up to 1% deviation) as follows

```
f_c = f_i*(1.+0.02*(random.rand()-0.5))
ph_c = 2*np.pi*random.rand()
```

Run this code a few times and adjust parameters to obtain locking of the costas loop. Examine the demodulated binary data and compare with the original data. You should find that in some instances your data is recovered, and sometimes the inverse.

1.5 Differential Coding

Finally, we shall implement differential coding to deal with the ambiguity in the recovered data.

```
tx_diff = np.zeros(1, dtype='bool')
for i in range(Nbits):
    tx_diff = np.append(tx_diff, tx_diff[i]^tx_bin[i])
Nbits = Nbits+1

... rest of code ...

rx_bin = np.empty(0, dtype='bool')
Nbits = Nbits-1
for i in range(Nbits):
    rx_bin = np.append(rx_bin, rx_diff[i]^rx_diff[i+1]).astype(bool)
```

Confirm that this change permits error-free transmission of the original digital data with clock recovery.

Your report should demonstrate, ideally in figures appropriately labelled and captioned:

- data input and output, both for without and with differential encoding
- clock output with reference carrier
- voltage driving the vco
- output data before thresholding

1.6 Documentation

python 3 `https://docs.python.org/3/`

numpy and scipy `https://docs.scipy.org/doc/`

matplotlib `https://matplotlib.org/contents.html`

spyder `https://docs.spyder-ide.org/`

Getting the python libraries

If you are using your own computer, make sure the Python libraries `scipy`, `numpy` and `matplotlib` are installed. These libraries are installed by default with the Anaconda python distribution. It is recommended that you use a suitable IDE for your project, such as Spyder.