SP2 Documentation

Nile Jocson <novoseiversia@gmail.com>

December 12, 2024

Contents

1 Git

The Git repository for this project can be found at `https://github.com/coe-journal/eee-111/tree/main/sp2`.

2 Documentation

2.1 Usage

2.1.1 Milestone 1.1

- `<freq:float> <dur:float> <pts:int>`
  — Generates the time-voltage format of a sine wave.
- `<freq:float> <dur:float> <pts:int> plot`
  — Same as the first, but shows a plot of the wave.
- `<freq:float> <dur:float> <pts:int> out <filename:str>`
  — Same as the first, but saves the values into a file.

2.1.2 Milestone 1.2

- `<freq0:float> <freq1:float> <dur:float> <pts:int>`
  `<bits:str>`
  — Generates the time-voltage format of a sine wave.
- `<freq0:float> <freq1:float> <dur:float> <pts:int> plot`
  `<bits:str>`
  — Same as the first, but shows a plot of the wave.
- `<freq0:float> <freq1:float> <dur:float> <pts:int> out <filename:str>`
  `<bits:str>`
  — Same as the first, but saves the values into a file.

2.1.3 Milestone 1.3

- `<pts:int>`
  `<t:float> <v:float>`
  `...`
  — Outputs the number of zero-crossings in the given signal.

2.1.4 Milestone 1.4

- `<pts:int> <bits:int>`
  `<t:float> <v:float>`
  `...`
  — Demodulates the given FSK signal.

2.1.5 Common Commands

- `help`
  — Prints the help text.
- `exit`
  — Exits the program.

2.2 Parsing

Parsing is handled by the `parse_rules` utilities that were written for my SP1 submission.

2.3 Time-Value

```
def time_value(
 f: Callable[[float], float],
 start: float,
 end: float,
 points: int
) -> list[tuple[float, float]]:
```

2.3.1 Behavior

The output of a function `f` in time-value format can be generated using the `time_value()` function. This returns a list of time-value pairs where `t` are `points` amount of numbers between and including `start` and `end`, and `v` is equal to `f(t)` for every `t`.

This function can also be used for functions with multiple parameters using `itertools.partial()`. For example, with a sine function:

```
from itertools import partial

def sine(
 frequency: float,
 phase_shift: float,
 t: float
) -> float:
 return math.sin(2 * math.pi * frequency * (t + phase_shift))

time_value(partial(sine, frequency, phase_shift), ...)
```

However, do note that `t` must be the last parameter of the function in order to be compatible with `time_value()`.

2.3.2 Mechanics

`time_value()` simply initializes an empty list of tuples of two floats, iterates through `t in arange(start, end, points)`, appends `(t, f(t))` into the list, and returns it.

2.4 FSK Modulation

```
def fsk(
 frequency0: float,
 frequency1: float,
 duration: float,
 points: int,
 bits: str
) -> list[tuple[float, float]]:
```

2.4.1 Behavior

The time-value format of an FSK signal representing a specified bit sequence can be generated using `fsk()`. This returns a list of time-value pairs with `points` amount of time points between and including `t = 0` and `t = duration`. Each bit in the sequence is represented by a sine wave with a frequency of `frequency0` or `frequency1` if the bit is `0` or `1`, respectively. Note that the sine wave has a phase-shift such that they start at `v = 0` for each bit, and that the last value will also be `v = 0`.

2.4.2 Mechanics

`fsk()` initizliaes an empty list of tuples of two floats, and calculates the endpoints of `t` for each bit using `arange(0, duration, n + 1)` where `n = len(bits)`. It then iterates through `i in range(n)`, initializing the following variables:

- `start = endpoints[i]`.
- `end = endpoints[i + 1]`.
- `frequency = frequency0` or `frequency1` depending on the bit as described before.

The list is then extended using `time_value()` from `start` to `end` with `points \\ n + 1` points (an additional point is added because of a limitation of `arange()`, which will be discussed later), using a sine function with a frequency of `frequency` and a phase-shift of `(start - end) * i`, which will make the wave start at `v = 0`.

After each iteration, the extra point in the list is popped. After the whole iteration, the list is appended with `(duration, 0)`. The list is then returned.

2.5 Zero Crossings

```
def count_crossings(
 tv: list[tuple[float, float]]
) -> int:
```

2.5.1 Behavior

The `count_crossings()` function simply returns the number of zero-crossings in the given time-value input. Touching zero does not count.

2.5.2 Mechanics

`count_crossings()` first removes all time-value pairs in the input where `v = 0`. Then, for each consecutive pair of time-value pairs, it checks if zero has been crossed in the two values by multiplying them and checking their sign. If the sign is negative, then zero has been crossed, and a counter is incremented.

Note that the definition of a zero-crossing is that for two variables a and b, $a > 0, b < 0$ or $a < 0, b > 0$. Since the signs of the two variables always differ, the sign of their product is always negative, i.e. $ab < 0$.

The counter is then returned after the iteration.

2.6 FSK Demodulation

```
def demodulate(
 bits: int,
 tv: list[tuple[float, float]]
) -> list[bool]:
```

2.6.1 Behavior

The `demodulate()` function demodulates a the time-value representation of an FSK-modulated signal by counting the number of zero-crossings and estimating the frequency of each division of the signal based on the given number of bits. The mean of all frequencies is then calculated, and this becomes the threshold frequency. If the frequency of a given division is higher than the threshold frequency, the bit that is represented by that division is a `1`. Otherwise, the bit is `0`. The demodulated bit sequence is then returned.

2.6.2 Mechanics

`demodulate()` initializes an empty list of booleans, which will be the demodulated bit sequence. It then removes all time-value pairs in the input where `v = 0`, and then initializes the following variables:

- `duration` is the duration of the FSK signal.
- `bit_duration` is the duration of each bit in the signal.
- `crossings` is an empty list which will contain the number of crossings for each division of the signal.
- `bit = 0`.

It then iterates with each pair of consecutive time-value pairs, counting each zero-crossing in the current division of the signal. If the `t` of the first pair is above the end-time of the current division, `bit` is incremented, moving on to the next division of the signal. This is calculated using `bit_duration * (bit + 1)`. However, `bit` is not incremented if it is already the last division of the signal (this is to support the final extra point in the time-value representation).

The frequencies are then calculated by `(c / 2) / bit_duration` for the number of crossings of each division `c`. The threshold frequency is calculated by getting the mean of the list of frequencies. It then iterates through each frequency; if the frequency is above the threshold, `True` is appended to the bit sequence, otherwise, `False` is appended. The demodulated bit sequence is then returned.

2.7 Other Utilities

2.7.1 Sine

The `sine()` function returns the value of `sin` given a frequency in Hz, a phase-shift in seconds, and a time in seconds. This is calculated using $\sin(2\pi f(t-c))$, where f is the frequency, c is the phase-shift, and t is the time.

2.7.2 Print Functions

The `print_x()` functions simply prints the output of its corresponding milestone function, i.e. `time_value()`, `fsk()`, `count_crossings()`, and `demodulate()`. It also handles multiline input for all milestones except milestone 1.1.

2.7.3 Plot Functions

For milestones 1.1 and 1.2, the `plot_x()` functions simply plots the output of its corresponding milestone function using `matplotlib`.

2.7.4 Out Functions

For milestones 1.1 and 1.2, the `out_x()` functions simply writes the output of its corresponding milestone function into a given file.

2.7.5 Help and Exit

In each program, inputting `help` will print out the help text, and inputting `exit` will exit the program.

2.8 General Questions

2.8.1 What was your general approach for the software project?

I simply made a dedicated function and file for each milestone. Since each milestone is a simple input-output problem, no state is required, so I didn't use any classes (except for the structs in the `parse_rules` library, which don't contain state and are simply there for cleanliness). I copy-pasted the `parse_rules` library from my last software project, since I think that it is a good, clean and concise way to define the inputs and how they are parsed for each line. Other than than, not really anything out-of-the-ordinary in how I coded this project.

2.8.2 How was your experience in programming?

To be completely honest, I was absolutely burnt out making this. Cramming the whole thing in under one day since I couldn't get the motivation after back-to-back projects and exams. While I love programming, sometimes it just isn't it. Otherwise, I thought the course was good. The discussion could be more hands-on and not simply a lecture on the topic for the week though.