Hack-A-Sat 3 Quals write-ups

by Poland Can Into Space

Matters of State (Revenge of Space Math)

In the task we're given orbital elements of some object and are asked to provide position and velocity vectors of this object at a given point in time.
We solved it mostly just by using the poliastro library, with a small twist, where we had to convert mean anomaly and eccentricity (provided in the task) into true anomaly (required by poliastro).
The solution is basically:

1. Parse inputs
2. Compute true anomaly
3. Use Orbit.from_classical() from poliastro
4. Read back position and velocity vectors
5. Repeat for next target

```
import astropy
import datetime
import numpy as np
from astropy.time import Time
from poliastro.core.angles import (
    D_to_nu as D_to_nu_fast,
    E_to_nu as E_to_nu_fast,
    F_to_nu as F_to_nu_fast,
    M_to_D as M_to_D_fast,
    M_to_E as M_to_E_fast,
    M_to_F as M_to_F_fast,
)


def true_anomaly_from_mean_poliastro_fast(ecc, M):
    ecc, M = ecc.value, M.to(u.rad).value
    if ecc < 1:
        M = (M + np.pi) % (2 * np.pi) - np.pi
        return E_to_nu_fast(M_to_E_fast(M, ecc), ecc) * u.rad
    elif ecc == 1:
        return D_to_nu_fast(M_to_D_fast(M)) * u.rad
    else:
        return F_to_nu_fast(M_to_F_fast(M, ecc), ecc) * u.rad


from pwn import *
from re import findall

from astropy import units as u

from poliastro.bodies import Earth
from poliastro.twobody import Orbit

ticket = "<PUT YOUR TEAM TICKET HERE>"

r = remote("matters_of_state.satellitesabove.me", 5300)

r.recvuntil(b"Ticket please:\n")

r.send(ticket.encode() + b"\n")
context.log_level = "DEBUG"

while True:
    data_set = r.recvuntil(b"Position: X,Y,Z").decode()

    pattern = r".*: ([\d.\-T:]+) *(.*)"

    values = findall(pattern, data_set)
    print(values)

    a = float(values[0][0]) * u.km
    ecc = float(values[1][0]) * u.one
    inc = float(values[2][0]) * u.deg
    raan = float(values[3][0]) * u.deg
    nu = float(values[4][0]) * u.deg
    argp = float(values[5][0]) * u.deg
    t = astropy.time.Time(datetime.datetime.fromisoformat(values[6][0]))

    nu = true_anomaly_from_mean_poliastro_fast(ecc, nu)

    target = Orbit.from_classical(attractor=Earth, a=a, ecc=ecc, inc=inc,
raan=raan, argp=argp, nu=nu, epoch=t)
```

```
    r.sendline(",".join(str(x) for x in target.r.to_value(u.km)))
    r.sendline(",".join(str(x) for x in target.v.to_value(u.km / u.s)))
    r.recvline()
    r.recvline()
    r.recvline()
```

Crosslinks (Revenge of Space Math)

In the task we are given a set of TLEs of some satellites and then a list of (name, time, distance, rate) from the point of view of one of the satellites. The goal is to figure out which of the satellites performed those measurements.

We ignored the rate completely and used only distance to find a matching satellite. The idea is quite simple:

1. Parse all the inputs
2. For each of the measurements we have, compute the distance between the measured satellite and every other satellite we have, and find the satellite with the smallest distance difference from the target measurement.
3. Count for which satellite those minimum differences were most common and send it as answer

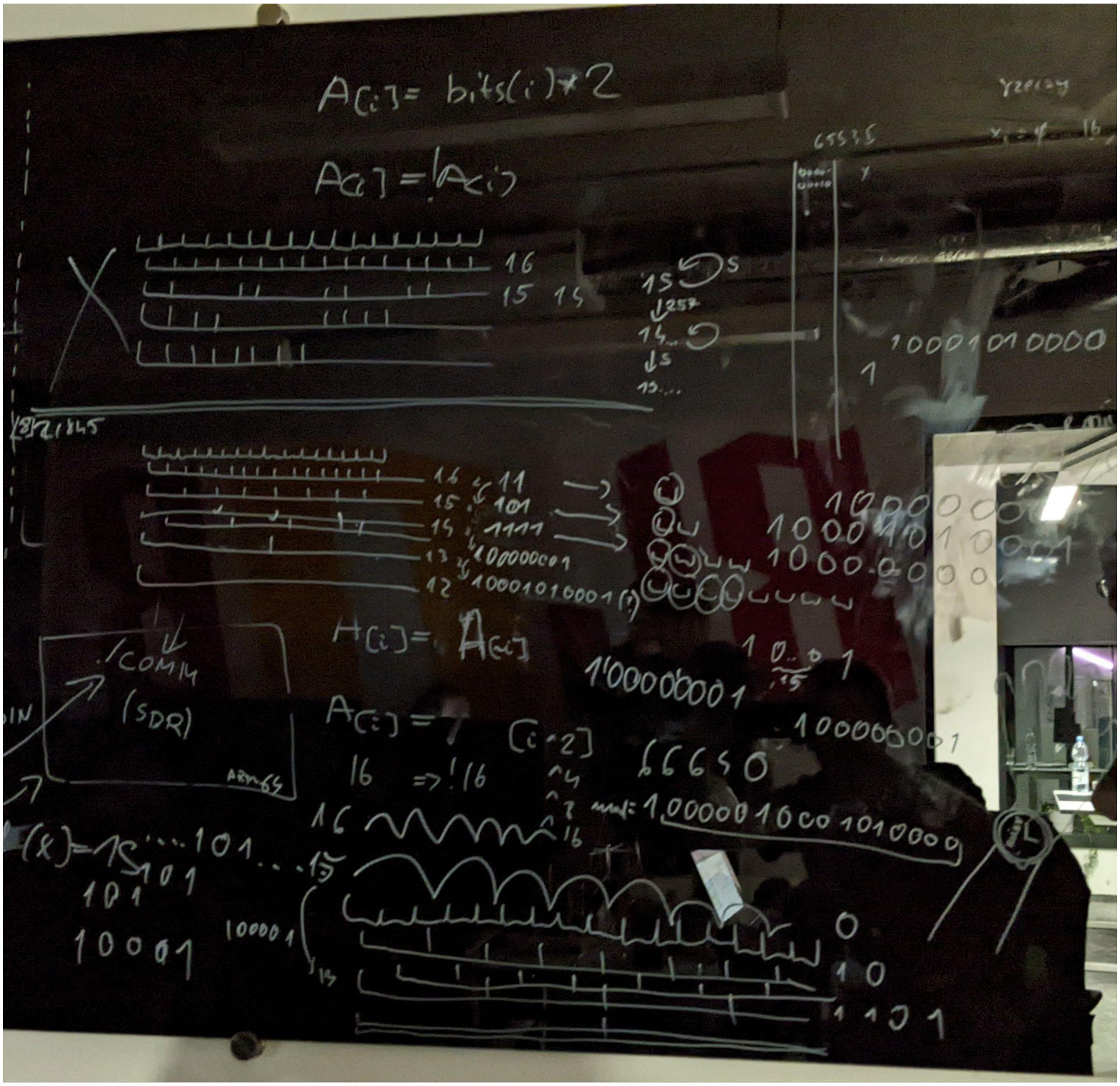
Since poliastro doesn't have loading orbits from TLE out-of-the-box and we were too lazy to implement it, we loaded TLEs using ephem library and wrote position vector and distance calculation by hand.

```
from pwn import *
import ephem
import numpy as np
import math
from collections import defaultdict


def calc_r(sat):
    xs = math.cos(sat.dec) * math.cos(sat.ra)
    ys = math.cos(sat.dec) * math.sin(sat.ra)
    zs = math.sin(sat.dec)
    ds = (sat.elevation + ephem.earth_radius) / 1e6
    s = np.array([xs, ys, zs]) * ds
    return s


def calc_dist(s1, s2, t):
    s1.compute(t)
    s2.compute(t)
    r1 = calc_r(s1)
    r2 = calc_r(s2)
```

```python
    r = r1 - r2
    return (r[0] ** 2 + r[1] ** 2 + r[2] ** 2) ** 0.5 * 1000


rr = remote("crosslinks.satellitesabove.me", 5300)
rr.sendline("<PUT YOUR TEAM TICKET HERE>")

context.log_level = "DEBUG"

while True:
    rr.recvuntil("TLE")
    tles = rr.recvuntil("Measurements").strip().decode()

    meas = rr.recvuntil("What").strip().decode()

    print(tles)
    print(meas)

    tles = tles.splitlines()[:-1]
    tles_parsed = {}
    for i in range(0, len(tles), 3):
        line1, line2, line3 = tles[i:i + 3]
        line1 = line1.strip()
        tle = ephem.readtle(line1, line2, line3)
        tles_parsed[line1] = tle

    meas = meas.split("Observations ")[1:]
    meas_parsed = {}
    for m in meas:
        m = m.splitlines()
        name = m[0].strip()
        meas_parsed[name] = []
        for line in m[2:]:
            if "What" in line: continue
            t, r, v = line.split(", ")
            t = t.replace("T", " ")
            t = t.split(".")[0]
            meas_parsed[name].append((t, float(r), float(v)))

    d = defaultdict(int)
    for name in meas_parsed:
        meas_tle = tles_parsed[name]
        for t, r, v in meas_parsed[name]:
```

```
        mn, mnind = 1e99, -1
        for name2 in tles_parsed:
            if name == name2: continue
            dist = calc_dist(meas_tle, tles_parsed[name2], t)
            delta = abs(dist - r)
            if delta < mn:
                mn, mnind = delta, name2
        print(mn, mnind)
        d[mnind] += 1

mx, mxind = 0, ""
for name in d:
    if d[name] > mx:
        mx, mxind = d[name], name

print("Best", mxind)

rr.sendline(mxind)
rr.recvuntil("Correct")
```

Bit Flipper (The Only Good Bug is a Dead Bug)



In this challenge we were given a binary and a remote server where it was hosted, which consisted of three mini-challenges, each similar to the others, but with raising hardness.

It used an array `h`, which was loaded from a file which wasn't distributed with the challenge (but was available to the binary running remotely). Each level picked a random 16-bit word `acc` (with a constraint that `h[acc] == 16`) and our task was to bring `h[acc]` to zero. The only move we could do was to send a 16-bit word, which was then bit-rotated randomly and then xored to `acc`. After each query we could learn the new value of `h[acc]`. And here's where the levels differed:

lvl1 picked the random rotation before the loop, lvl2 randomized it for each iteration, and not sure what lvl3 exactly did, because our solver for 1 and 2 managed to solve also it :)

The biggest problem in this challenge was that `h` array was completely unknown to us. But we could learn some of its properties, even despite the random input rotation! The first one was the histogram of the values: we could keep sending random 16-bit words and reading the resulting `h[acc]`. Because our input was completely random, the rotation didn't matter. What we got after a bunch of data-collection iterations was: (<value>: <occurrences>)

```
16: 1441,
15: 737,
14: 364,
13: 164,
12: 68,
11: 48,
10: 24,
9: 12,
8: 8,
7: 3,
5: 2
```

So, it seemed that this was some measure of how close we are to the right `acc` value - values were from 16 to 0, we always started at 16 and won if we got to 0, and each value was 2x more "rare" than the previous. This suggested that there's +/- 1 `acc` value which gives `h[acc] == 0`. So, no chance at hitting 0 by luck in reasonable time (actually, it might have been doable in a few hours, assuming we wouldn't get banned for flooding the server 🙃).

Then we started guessing if there's any simple function we know which `h` could be, to match the histogram. Some candidates were e.g. the number of leading ones or zeros, but we managed to contradict all the guesses experimentally.
So, we moved to finding some less trivial properties than just the histogram, at the same time verifying them automatically on the remote server to ensure that they actually hold.
One of the interesting properties (and the first non-trivial one we've found) was that if you were at 16, then sending a number with a single bit set always moved us to a non-16, and from any non-16 number doing the same sent us back to 16.
So, 16 -> 15 transition seemed to be checking whether the bit parity was correct (correct i.e. the same as the winning `acc`).
Another, weirder property we've found was that if we were at 15 and xored with 0b101 (or any of its rotations) then we always stayed at 15.
We started thinking about correction codes (e.g. hamming codes) and xor-arithmetic (that's how I call it, but the formal math name is "arithmetic on polynomials in GF(2)", it's also the one used in CRC). But we couldn't find any model which would stand against the experiments.
Then we thought that maybe these patterns like 0b101 also exist for lower numbers. And we actually managed to find a few of them!

```
16: 11
15: 101
14: 10001
13: 100000001
12: 100000101000001
```

They looked quite regular, but we got stuck at this point for quite some time trying to find a relation between them and extend them to lower numbers (where bruteforce search was taking much more time).

One property they all had (and we tested this experimentally) is that they all worked regardless of rotations.

The realization we needed was that these weren't actually the smallest patterns for each of the numbers! We started searching for minimal patterns and got this:

```
16: 11
15: 101
14: 1111
13: 10001
...
```

So, these were the consecutive powers of 0b11 in xor-arithmetic!

Now we could see that each transition to the lower value in the array was a check whether `acc` modulo the given pattern matches the modulo of the winning number. That explained why xoring by these patterns never changed the number (nor their powers, which were the "non-optimal" entries we've found at first) - they were the same as the modulo on that level, so they couldn't change the outcode of that check, but couldn't also lower our score, because all modulos in the previous checks were divisors of this one.

Knowing this we could easily solve all levels of the challenge by sending the value of `0b11^(16 - i)` on level `i` - notice that this is a modulo from one level above the current one - it cannot lower our score, but can (and will) raise it.

Final solver:

```python
from pwn import *
from random import randint

def int2bits(val):
    if val == 0:
        return []
    bits = f'{val:b}'
    bits = list(map(int, bits))
    return bits

def bits2int(bits):
    if len(bits) == 0:
        return 0
    return int(''.join(map(str, bits)), 2)

def poly_add(a, b):
    if len(a) < len(b):
        a,b = b,a
    a = a[:]
    for i in range(len(b)):
        a[-1-i] ^= b[-1-i]
    while len(a) and a[0] == 0:
        a = a[1:]
    return a

def poly_mul(a, b):
    res = []
    for i in range(len(a)):
        if a[i] == 1:
            res = poly_add(res, b + [0] * (len(a)-i-1))
    while len(res) and res[0] == 0:
        res = res[1:]
    return res

def poly_mod(bits, mod):
    bits = bits[:]
    while len(bits) >= len(mod):
        if bits[0] == 0:
            bits = bits[1:]
        else:
            for i in range(len(mod)):
                bits[i] ^= mod[i]
    while len(bits) and bits[0] == 0:
        bits = bits[1:]
    return bits

pows = []
x = int2bits(0b1)
for i in range(16):
    pows.append(x)
    print(f'{i}: {bits2int(x)} {x}')
    x = poly_mul(x, [1,1])

def guess(x):
    r.recvuntil(b'Guess: ')
    r.sendline(str(x).encode())
    d = r.readline().strip().decode()
    if 'Reset' in d:
        print('XXX', d)
        return 0
    return int(d)

results = []
with remote('bitflipper.satellitesabove.me', 5100) as r:
    r.sendline(b'<PUT YOUR TEAM TICKET HERE>')
    res = guess(1)
    while res > 0:
        inp = poly_mod(pows[16-res] + [0]*randint(0, 15), [1]+[0]*15+[1])
        res = guess(bits2int(inp))
    print('win!')

    res = guess(1)
    while res > 0:
        inp = poly_mod(pows[16-res] + [0]*randint(0, 15), [1]+[0]*15+[1])
        res = guess(bits2int(inp))
    print('win!')

    res = guess(1)
    while res > 0:
        inp = poly_mod(pows[16-res] + [0]*randint(0, 15), [1]+[0]*15+[1])
        res = guess(bits2int(inp))
    print('win!')
    r.interactive()
```

Leggo My Steggo! (Crypto Category Placeholder Name)

In this challenge, we’re provided with the service network location and a zip archive that contains a few satellite image captures.
The service is an interactive command prompt. We figured out that it responds to some commands from the scripts mentioned in the challenge description like “`SHELL_CMD`” or “`SEND_FILE`”. The response is a sha256 hexdigest of one the satellite images.

After many attempts at responding to the digest with a acceptable answer, we’ve noticed that “steghide” is able to extract some text data from the images:

```
┌──(root㉿kali)-[/images]
└─# steghide extract -sf chicago.jpg -p ''
wrote extracted data to "emb.txt".
```

```
┌──(root㉿kali)-[/images]
└─# cat emb.txt
a6ec1dcd107fc4b5373003db4ed901f3
```

The “inverted” value of the outputted md5 hash (“deepdish”) was accepted by the service. All we had to do now was to extract the hidden data from all images and automate the communication.

We’ve noticed that one of the images did not produce a md5 hash but a message: “Use the command mnemonic that outputs moon.png”. By looking up the filename in repository, we now know that we have to use the command “`PLAYBACK_FILE`”.

```python
from pwn import *
from pathlib import Path

HASHES = {
    "d6bc6fbee628c3278ef534fd22700ea4017914c2214aa86447805f858d9b8ad4":"boston.jpg",
    "04ca7d835e92ae1e4b6abc44fa2f78f6490e0058427fcb0580dbdcf7b15bbb55":"chicago.jpg",
    "5d69a46fb16ed453c6a5f0b07612ce9339b6df7b0bdd81ad69a236f3891ff8c3":"hashes.txt",
    "242f693263d0bcc3dd3d710409c22673e5b6a58c1a1d16ed2e278a8d844d7b0b":"la.jpg",
    "2aa0736e657a05244e0f8a1c10c4492dde39907c032dba9f3527b49873f1d534":"miami.jpg",
    "983b1cc802ff33ab1ceae992591f55244538a509cd58f59ceee4f73b6a17b182":"nyc.jpg",
    "7e03349fe5fa4f9e56642e6787a0bfda27fb4f647e3c283faaf7bd91dbfd1d39":"oahu.jpg",

"b4447c4b264b52674e9e1c8113e5f29b5adf3ee4024ccae134c2d12e1b158737":"portland.jpg",
    "f37e36824f6154287818e6fde8a4e3ca56c6fea26133aba28198fe4a5b67e1a1":"sf.jpg",
    "088f26f7c0df055b6d1ce736f6d5ffc98242b752bcc72f98a1a20ef3645d05c1":"slc.jpg",
    "2de4702c81d32b4cb484919d4f7b3d61860880b552f0305bdb12c7fa37108555":"ticket",
    "3b20a3b5b327c524674ca5a8310beb2d9efc5c257e60c4a9b0709d41e63584a3":"vegas.jpg",
}

FILES = {
    "boston.jpg":"beanpot",
    "chicago.jpg":"deepdish",
    "la.jpg":"cityofangels",
    "miami.jpg":"springbreak",
    "nyc.jpg":"fuhgeddaboudit",
    "oahu.jpg":"Use the command mnemonic that outputs moon.png",
    "portland.jpg":"stayweird",
    "sf.jpg":"housingmarket",
    "slc.jpg":"skiparadise",
    "vegas.jpg":"letsgolasvegas",
}


ticket = "<PUT YOUR TEAM TICKET HERE>"
r = remote("leggo.satellitesabove.me", 5300)
```

```
r.recvuntil(b"Ticket please:\n")
r.send(ticket.encode() + b"\n")

r.recvuntil(b"CMD")
r.send(b"PLAYBACK_FILE\n")

recv_hash = r.recvuntil(b"\n").decode().strip()[1:-1]
print(recv_hash)

filename = HASHES[recv_hash]
dupa = FILES[filename]

context.log_level = "debug"

for i in range(10):
    r.send(dupa.encode() + b"\n")
    r.recvline()
    new_hash = r.recvuntil(b"?").decode().split("\n")[-1]
    print(new_hash)

    new_hash = new_hash.strip("?")

    filename = HASHES[new_hash]
    dupa = FILES[filename]
```

The Wrath of Khan (Rocinante Strikes Back)

In this task, we're given a compiled binary that receives data from sensors and uses this information to control the thrusters, heat shields, and parachute deployment.
After reverse-engineering the binary and trying to figure out the problem we decided to reimplement the binary from the ground up to allow us to modify the logic more easily.
The communication protocol was pretty straight-forward - it was using standard TCP sockets and fixed structures.

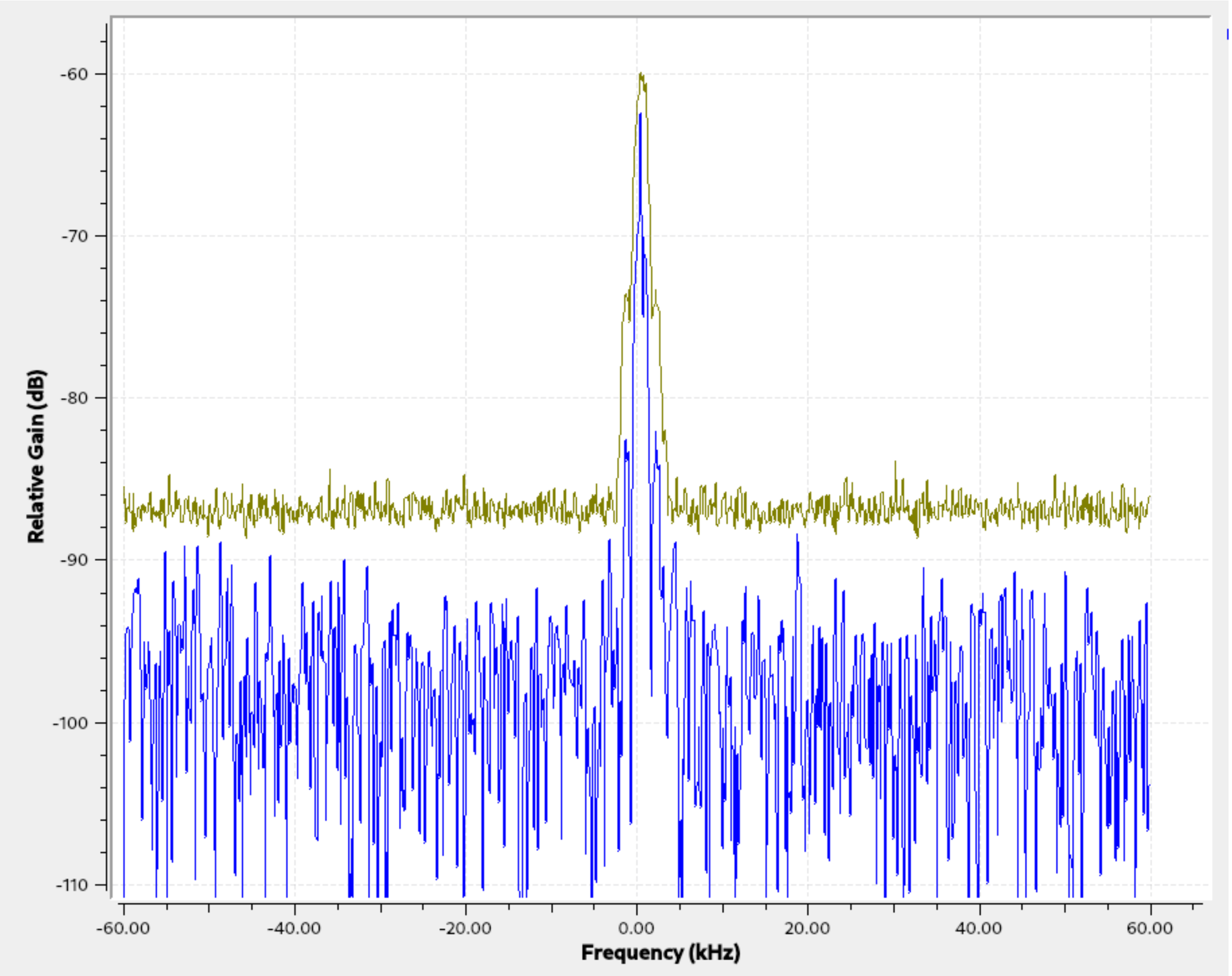
We've managed to partially reimplement the original logic and tune the program to safely land the probe.

```
const std = @import("std");
const c = @cImport({
  @cInclude("stdlib.h");
});

const Sensor = packed struct {
  pos_x: f32,
  pos_y: f32,
  pos_z: f32,
  vel_x: f32,
  vel_y: f32,
  vel_z: f32,
  g_force: f32,
};

const Command = packed struct {
  // blue
  vel_x: f32,
  // orange
  vel_y: f32,
  // green
  vel_z: f32,
  deploy_chute: u32,
  deploy_shield: u32,

  pub fn zero() @This() {
    var cmd = Command {
      .vel_x = 0,
      .vel_y = 0,
      .vel_z = 0,
      .deploy_chute = 0,
      .deploy_shield = 0,
    };
    return cmd;
  }
};

comptime {
  std.debug.assert(@sizeOf(Command) == 0x14);
  std.debug.assert(@sizeOf(Sensor) == 0x1c);
}


const Vec3 = struct {
  x: f32,
  y: f32,
  z: f32,
  pub fn init(x: f32, y: f32, z: f32) @This() {
    return Vec3{.x = x, .y = y, .z = z};
  }

  fn norm(self: @This()) f32 {
    return std.math.sqrt(self.x * self.x + self.y * self.y + self.z *
self.z);
  }
};

fn boost(sensor: *Sensor) Command {
    var pos = Vec3.init(sensor.vel_x, sensor.vel_y, sensor.vel_z);
    const factor: f32 = 3.0;
    var tmp_x = factor * pos.x / pos.norm();
    var tmp_y = factor * pos.y / pos.norm();
    var tmp_z = factor * pos.z / pos.norm();

    var cmd = Command {
      .vel_x = tmp_x,
      .vel_y = tmp_y,
      .vel_z = tmp_z,
      .deploy_chute = 0,
      .deploy_shield = 0,
    };

    return cmd;
}

fn decision(ticks: u32, sensor: *Sensor) Command {
    var pos = Vec3.init(sensor.vel_x, sensor.vel_y, sensor.vel_z);
    const factor: f32 = if (ticks % 50 == 0) 0.025 else 0.0;
    var tmp_x = factor * pos.x / pos.norm();
    var tmp_y = factor * pos.y / pos.norm();
    var tmp_z = factor * pos.z / pos.norm();

    //const k1 = ticks > 10000;
    const k1 = sensor.g_force < -6.0;

    var cmd = Command {
      .vel_x = -tmp_x,
      .vel_y = -tmp_y,
      .vel_z = -tmp_z,
      .deploy_chute = if (k1) 1 else 0,
      .deploy_shield = if (k1) 1 else 0,
    };

    return cmd;
}
```

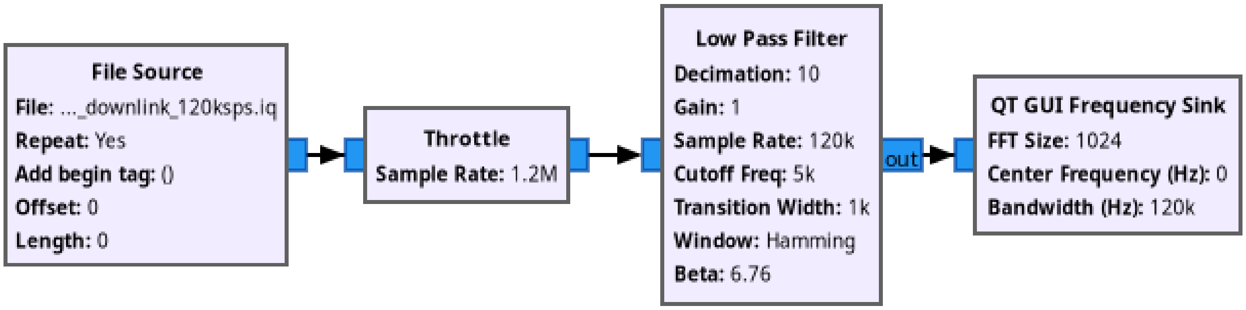
```
pub fn main() !void {
  const addr = try std.net.Address.parseIp("127.0.0.1", 6000);
  var sock = try std.net.tcpConnectToAddress(addr);
  defer sock.close();
  var writer = sock.writer();
  var reader = sock.reader();


  var count: u32 = 0;
  var stdout = std.io.getStdOut().writer();
  while (true) : (count += 1) {
    var sensor: Sensor = undefined;
    _ = try reader.readAll(std.mem.asBytes(&sensor));
    _ = stdout;
    // try stdout.print("x={} y={} z={}\n", .{sensor.pos_x, sensor.pos_y,
sensor.pos_z});
    // try stdout.print("g={}\n", .{sensor.g_force});
    var cmd: Command = undefined;
    if (count < 30) {
      cmd = boost(&sensor);
    } else {
      cmd = decision(count, &sensor);
    }
    try writer.writeAll(std.mem.asBytes(&cmd));
  }
}
```

Doppler

In this challenge, we get a IQ file to decode. The only hint we get is that it contains some doppler shift. During solving, we used GNURadio framework and python to decode data. After inspecting the file with FFT we can see a signal, not centered exactly at f = 0 Hz (some shift to the right), with very slow drift with time:



So, we will start with filtering and decimating the signal to get only the interesting part:

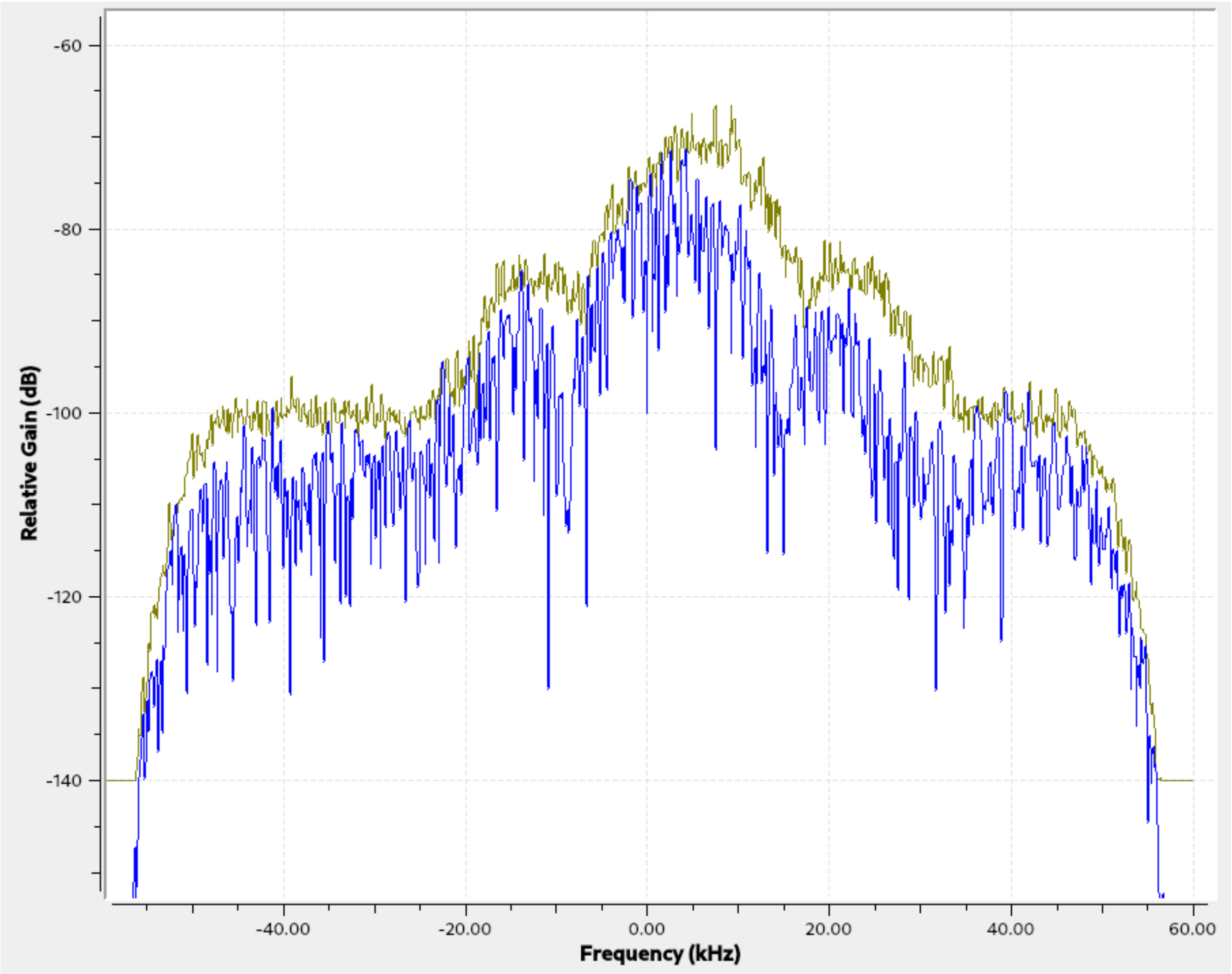


File Source
File: ..._downlink_120ksps.iq
Repeat: Yes
Add begin tag: ()
Offset: 0
Length: 0

Throttle
Sample Rate: 1.2M

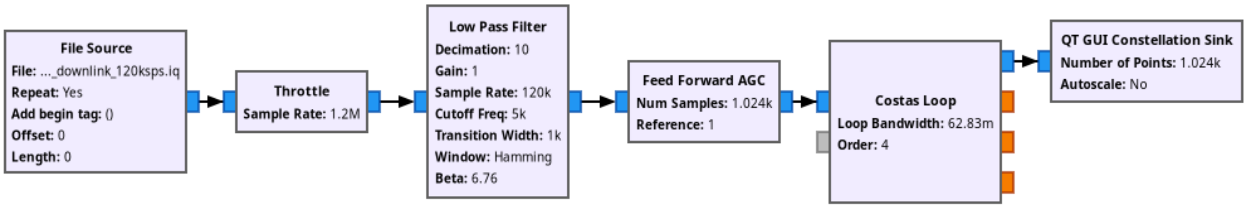
Low Pass Filter
Decimation: 10
Gain: 1
Sample Rate: 120k
Cutoff Freq: 5k
Transition Width: 1k
Window: Hamming
Beta: 6.76

QT GUI Frequency Sink
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 120k



We can see some kind of modulation. Best guess, given previous tasks, was the phase modulation.

Let’s try demodulating PSK with Costas’ loop to eliminate frequency shift:



File Source
File: ..._downlink_120ksps.iq
Repeat: Yes
Add begin tag: ()
Offset: 0
Length: 0

Throttle
Sample Rate: 1.2M

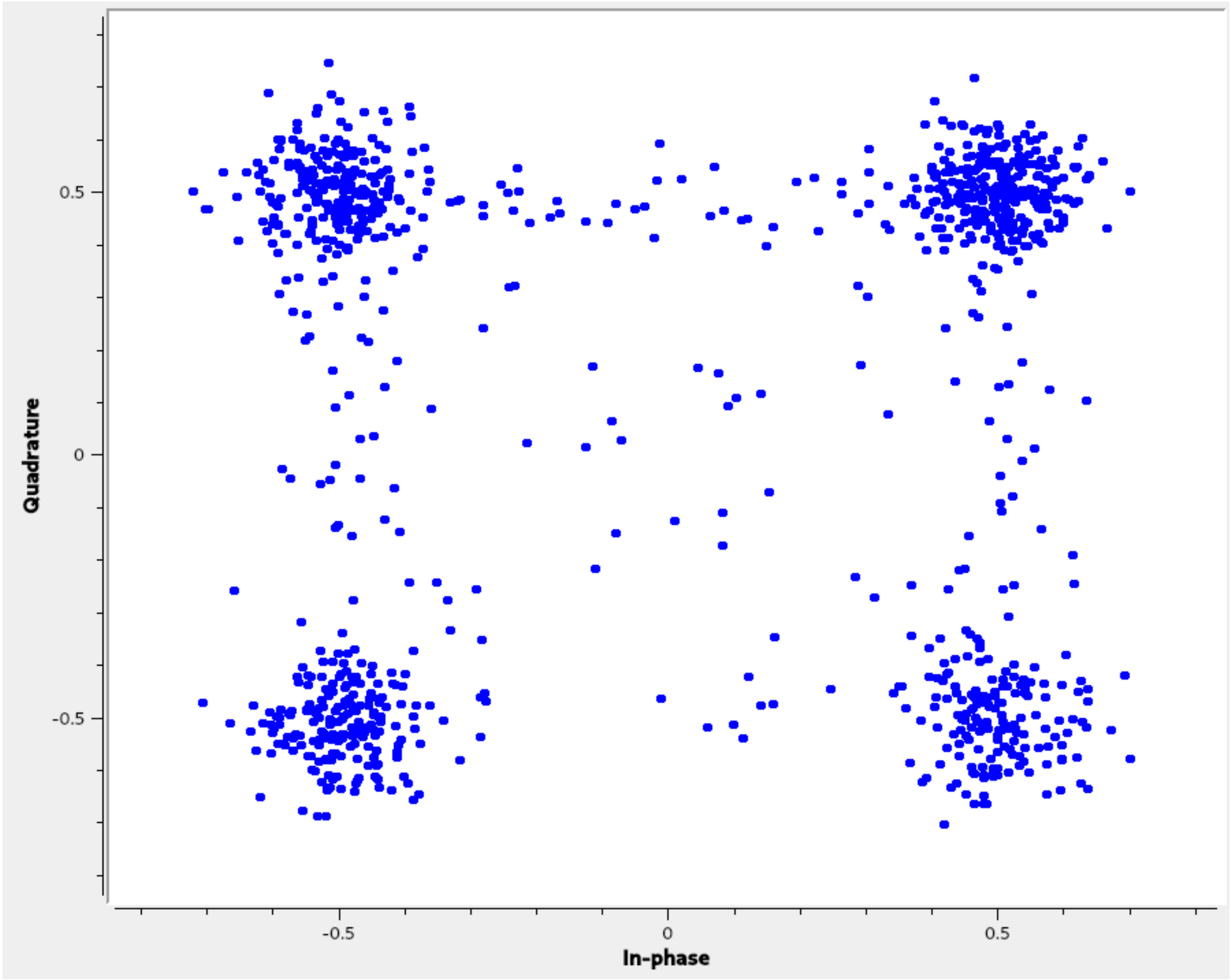
Low Pass Filter
Decimation: 10
Gain: 1
Sample Rate: 120k
Cutoff Freq: 5k
Transition Width: 1k
Window: Hamming
Beta: 6.76

Feed Forward AGC
Num Samples: 1.024k
Reference: 1

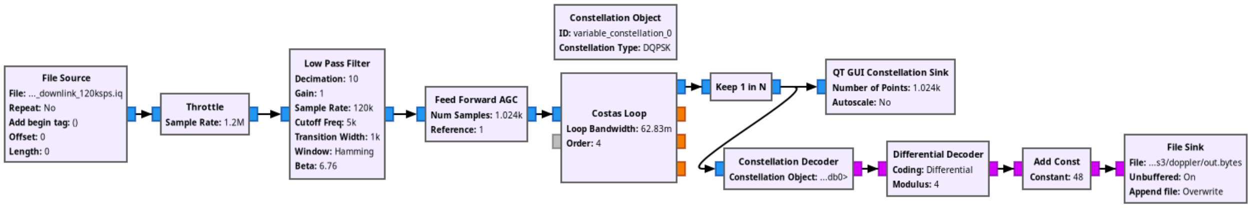
Costas Loop
Loop Bandwidth: 62.83m
Order: 4

QT GUI Constellation Sink
Number of Points: 1.024k
Autoscale: No

We’re getting somewhere:



Last task done in gnuradio was to decimate data to be left only with correct samples. After trial and error we've found that decimation = 10 was the correct one. It turned out that the bit transition alignment was not necessary.
We are exporting samples to file for analysis with different modulations (QPSK, DQPSK) and differential decoder enabled or bypassed to provide all possibilities to try decoding.



File Source
File: ..._downlink_120ksps.iq
Repeat: No
Add begin tag: ()
Offset: 0
Length: 0

Throttle
Sample Rate: 1.2M

Low Pass Filter
Decimation: 10
Gain: 1
Sample Rate: 120k
Cutoff Freq: 5k
Transition Width: 1k
Window: Hamming
Beta: 6.76

Feed Forward AGC
Num Samples: 1.024k
Reference: 1

Constellation Object
ID: variable_constellation_0
Constellation Type: DQPSK

Costas Loop
Loop Bandwidth: 62.83m
Order: 4

Keep 1 in N

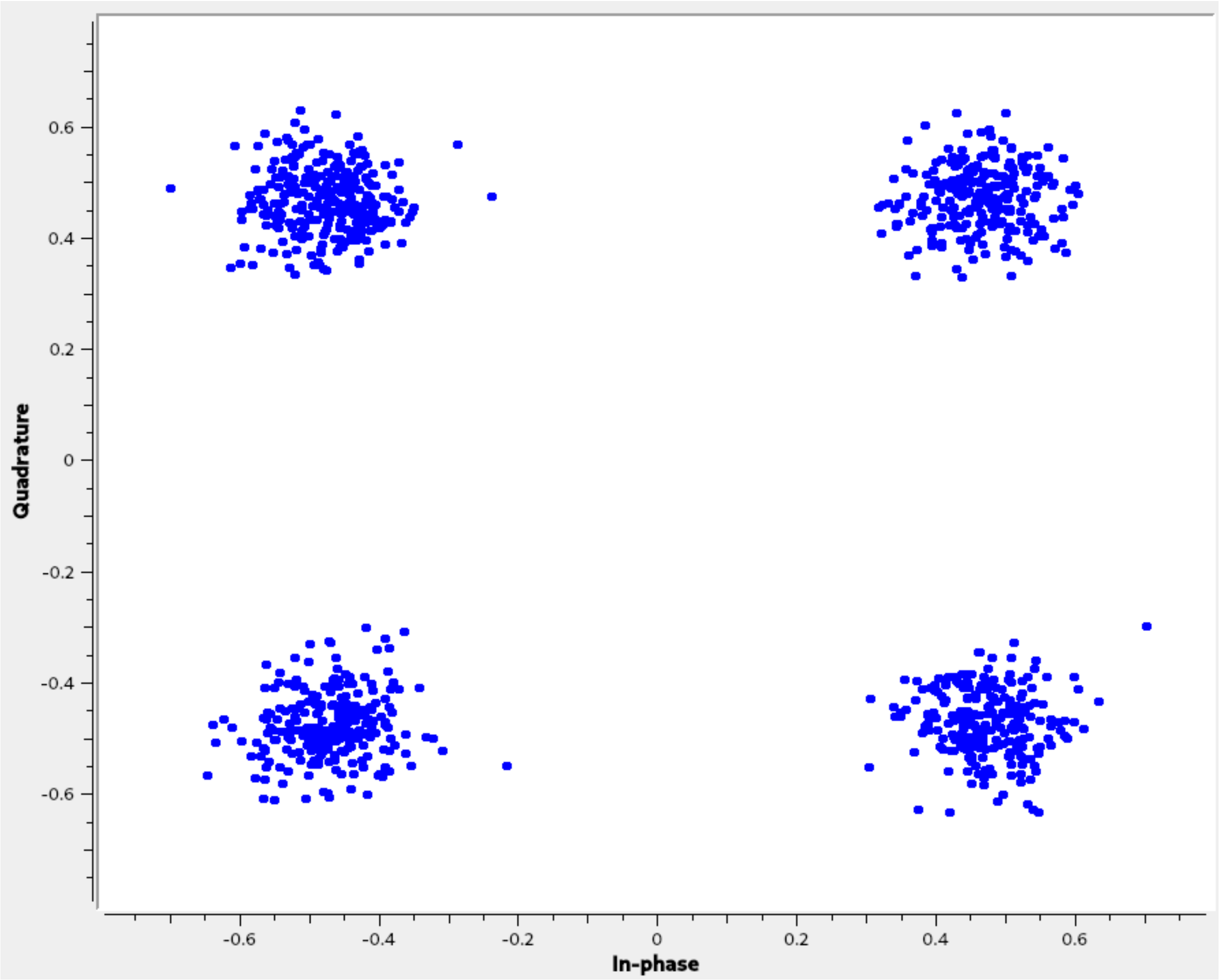
QT GUI Constellation Sink
Number of Points: 1.024k
Autoscale: No

Constellation Decoder
Constellation Object: ...db0>

Differential Decoder
Coding: Differential
Modulus: 4

Add Const
Constant: 48

File Sink
File: ...s3/doppler/out.bytes
Unbuffered: On
Append file: Overwrite



Last, we feed all the exported possibilities to Python, where we test all the different samples -> bits mappings.:

```
#!/usr/bin/env python3
from pwn import *
from random import randint
from collections import Counter
from pprint import pprint
import itertools

symbols = read('./out.symb').decode()

map_list = ['00', '01', '10', '11']

gotit = None
for mapowanie in list(itertools.permutations(map_list)):
    print()
    print(mapowanie)

    # normal encoding
    ss = []
    for symbol in symbols:
        s = int(symbol)
        ss.append(mapowanie[s])
    ss = ''.join(ss)

    for start in list(range(0, 8)):
        x = unbits(ss[start:], 'little')
        if b'flag' in x:
            gotit = x
            break
    if gotit is not None:
        break

if gotit is not None:
    print('got flag!')
    x = gotit.split(b'[THERE_B20BYTES_O_FL4G_IN_THIS_PACKET]')
    for i in x:
        print(i[:20])
```

One of the mappings is the correct one, and we start to see ASCII packets. Some of the packets consist of the text `THERE_B20BYTES_O_FL4G_IN_THIS_PACKET -` we print them, and after manual assembly - flag is ready.