Decoding digital weather satellite images: the LRPT protocol from Meteor-M2

QPSK, Viterbi, Reed Solomon and JPEG: from IQ coefficients to images

J.-M Friedt, February 17, 2019

Analyzing digital satellite communication protocols is an opportunity to explore all the layers of signal routing, from the physical layer with the phase modulated signal we acquire, to the error correcting mechanisms (convolution code to be decoded using the Viterbi algorithm and block code with Reed Solomon) and finally collect data blocks containing the JPEG image thumbnails to be assembled to provide a complete picture. This whole demonstration requires, from a hardware perspective, about ten euros worth of investment for finally having the satisfaction of mastering the whole communication chain as used in space links.

1 Introduction

Shifting from analog to digital communication is an unquestionable trend (analog television to DVB-T, analog commercial broadcast FM to DAB, telephone), and the radiofrequency links are no exception, aimed at optimizing the radiofrequency spectrum usage and the quality of the transmitted signals. While the analog APT (Automatic Picture Transmission) protocol of the the low-Earth polar orbiting NOAA satellites is doomed to disappear with the end of this satellite constellation, the succession seems to be taken care of with a protocol using the same bandwidth but digital, allowing the transmission of pictures with improved resolution: LRPT (Low Rate Picture Transmission). We shall see that this performance gain is achieved at the expense of a significantly increased processing complexity.

Most readers will probably hardly ever care about LRPT, if only because some functional free opensource decoding software is available. Why then spend so much time decoding images transmitted from the Russian Meteor-M2 [1] satellite, only source currently easily accessible (low Earth orbiting satellite) transmitting a LRPT data stream (the LRPT emitter of the European Metop-A is broken, and Metop-C is being commissioned while these lines are written) ? On the one hand, LRPT is only one example of the general class of digital communication protocols currently in use, with increasingly complex modulation schemes and abstraction levels ranging over all the OSI layers. We hence not only have an opportunity to explore these layers and understand practically their meaning, but other protocols close to LRPT are being used for high resolution picture transmissions from low-Earth orbiting satellites (even Terra and Aqua from the MODIS constellation) or geostationary orbit [2]. Although we shall depart at some point from the processing chain described in this last reference, the beginning of the acquisition and signal processing chain will be the same for LRPT decoding. Furthermore, if we are to believe the documents provided by the various space agencies during the last two decades, LRPT must be the future of low-bandwidth space communication, even though Russians are the only ones practically exploiting the protocol in the VHF (Very High Frequency – 30 to 300 MHz, hence including the band around 137 MHz dedicated to satellite communications) band currently. Beyond these applied engineering aspects, the techniques we will enforce are used in many current digital interfaces used daily, from data communication and storage to television: [3] estimated in 2005 that 10^{15} bits/second were being processed using convolution codes for television alone.

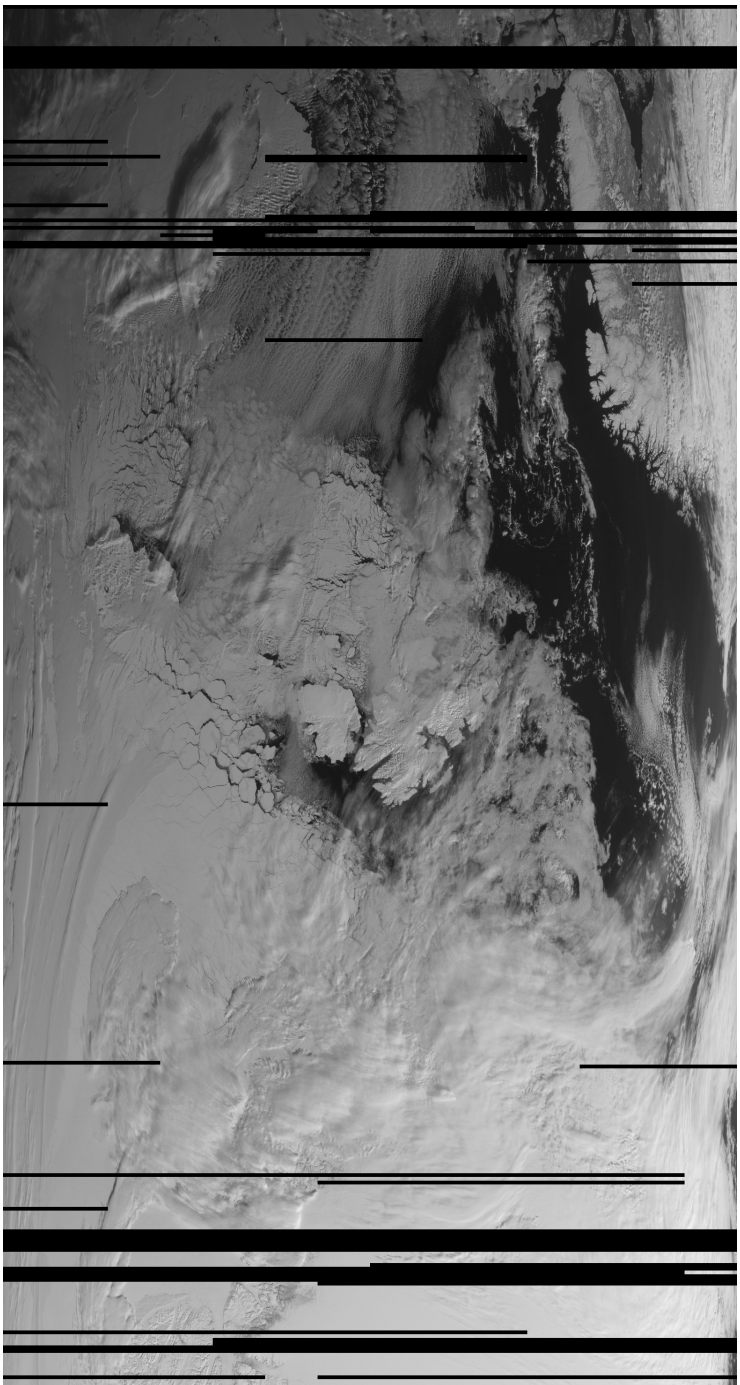


Figure 1: Meteor-M2 image acquired from Spitzbergen, Northern Scandinavia visible on the top right of the image as the North Pole is towards the bottom-left of the image.

Let us first provide some inspiration to the reader by demonstrating the targeted result (Fig. 1). Meteor M2 transmits on a 137,9 MHz carrier, so that current NOAA APT receiver installations [4] are perfectly suited. Analyzing the reception quality for LRPT is a bit more complex than in the case of APT: the digital mode is not suitable for an audio frequency analysis of the link quality as is the case for APT with its sweet melody at 2400 Hz, and only a constellation diagram observation (see

section 5.6) allows for assessing whether the receiver and its antenna are functional. We had to record tens of failed passes before acquiring a usable signal. The challenge with polar-orbiting satellites is that they often fly over the poles, and only once every day over any location at our temperate latitudes. We are lucky enough to collect signals from Spitsbergen, at 79°N, a latitude at which a polar orbiting satellite is seen every 100 minutes. If we do not take care, we might even spend most of our time monitoring radiofrequency signals from satellite rather than getting work done ! An additional bonus is to fetch a direct view of the North Pole, with no geographical interest other than showing that it can be done. The picture shown in Fig. 2 is aimed at demonstrating how simple hardware is used to receive Meteor M2 signals: as for NOAA satellites, two rigid wires and a Digital Video Broadcast-Terrestrial (DVB-T) receiver based on the R820T(2) frontend and RTL2832U analog to digital converter and USB transceiver used as software defined radio are enough. Such hardware, compact and light weight, is easily improvised even in remote areas if not readily available.

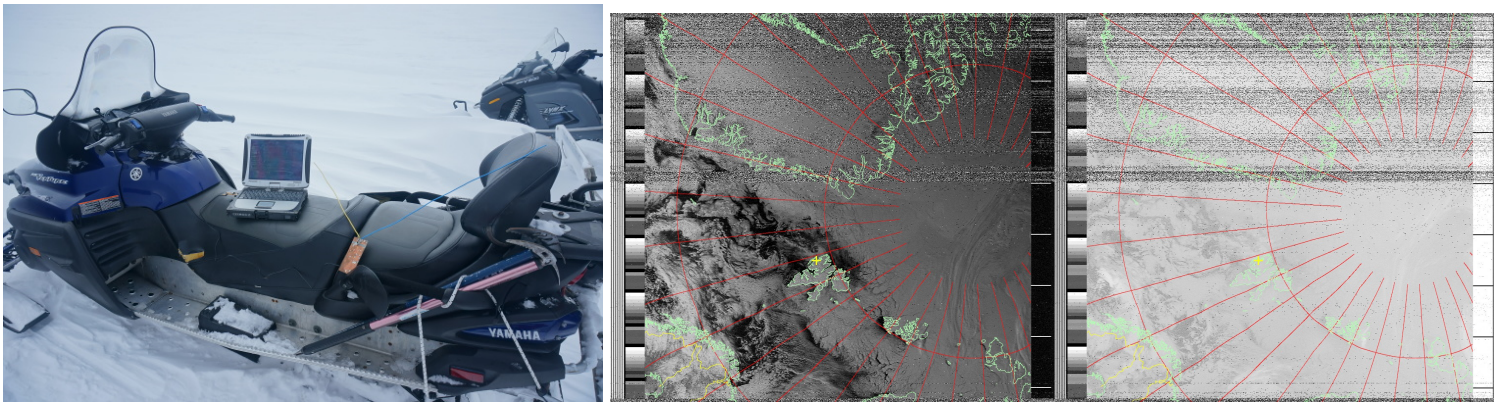
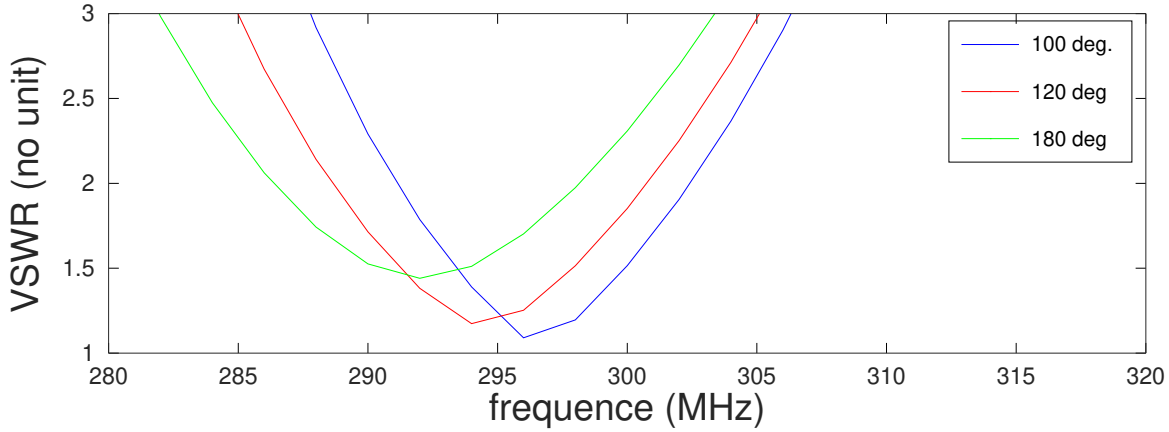


Figure 2: Left: experimental setup. A dipole antenna, a DVB-T receiver and a computer running GNU Radio are enough to collect data from Meteor M2 on a 137.9 MHz carrier. Right: result after processing the signal received from a NOAA satellite (analog communication) over Spitsbergen. From such a site, low-Earth polar orbiting satellites reappear every 100 minutes, for passes lasting about ten minutes.

The careful reader will have noticed (Fig. 2) that we have replaced a basic dipole antenna with a dipole exhibiting an angle of 120° as advised by `lna4all.blogspot.com/2017/02/diy-137-mhz-wx-sat-v-dipole-antenna.html`. Why this strange angle ? for the same reason than the elements of a discone antenna exhibit some angle with respect to the radiating vertical monopole and are not horizontal: in order to match impedance. A quick NEC2 [5] simulation demonstrates how impedance at resonance is dependent on the angle between the wires:



and indeed, an angle of 120° helps us get close to the targeted 50 Ω of the radiofrequency connectors and cables. Practically, considering how close ground and other disturbing metallic elements including the snow mobile are, the capacitive parasitic elements are so significant that the antenna is necessarily mismatched. Meeting theoretical expectations can never hurt nevertheless...

Acquisition and processing are completed as two different steps: GNU Radio is used not only to collect complex, as identity and quadrature (I,Q), samples from the radiofrequency receiver, but also to perform pre-processing steps to compensate for frequency offsets between the moving satellite and the ground-based receiver (section 5) and lock the bit-detection clock on the transitions observed on the digital data stream. Doing so, we reduce the data-rate and hence the size of the file storing the data on the hard disk for further processing. The image (Fig. 1) was decoded using `meteor_decoder` available at `github.com/artlav/meteor_decoder.git`. This software, written in Pascal (`apt-get install fpc`), is trivially compiled with `./build_medet.sh` to generate the program `medet` which is used with `./medet file.s output -s` with `file.s` the file generated from GNU Radio. Doing so,

we have obtained a very nice image, have understood nothing of the processing steps, and remain slaves of an excellent developer who has provided a perfectly functional tool. No interest whatsoever.

Our aim, in this presentation, is to analyze the detailed operations of `medet`, understand its operating principles, and without disregarding the comfort of a functional application, benefit from this opportunity of understanding the LRPT protocol to improve our knowledge on "modern" digital communication techniques.

2 When will the satellite fly overhead ?

The first question to be answered in a project aimed at receiving a low-Earth polar orbiting satellite signal is to know its flight schedule. Indeed, with an orbit at an altitude of about 800 km from the surface of the Earth, the satellite performs one complete orbit every 100 minutes, and is visible from a given location on the surface of the planet for a dozen minutes at most. For historical reasons, our preferred satellite pass prediction tool is `SatTrack` which, despite its Y2K bug (`http://pe1chl.nl.eu.org/SatTrack/`), remains an excellent command line software perfectly functional. We fill its `data/cities.dat` configuration file with a new entry including an identifier, and latitude and longitude (negative towards the east !) coordinates of the site at which the receiver is located, as well as the orbital parameters of the satellite in `tle/tlex.dat`: the file fetched at `www.celestrak.com/NORAD/elements/weather.txt` provides such regularly updated parameters. Identifying the Metero M2 satellite as `METEOR-M 2`, we obtain a list of passes in the following format

```
BTS SatTrack V3.1 Orbit Prediction

Satellite #40069  : METEOR-M 2
Data File         : tlex.dat
Element Set Number: 999 (Orbit 21896)
Element Set Epoch : 27Sep18  21:20:50.463 UTC   (2.3 days ago)
Orbit Geometry    : 816.87 km x 823.72 km at 98.593 deg
Propagation Model : SGP4
Ground Station    : NYA   ---   JQ58WW
Time Zone         : UTC (+0.00 h)

 Date (UTC)          Time (UTC) of        Duration   Azimuth at  Peak  Vis Orbit
                  AOS      MEL      LOS     of Pass  AOS MEL LOS  Elev
Sun  30Sep18  05:28:54 05:35:40 05:42:25  00:13:31  355  56 117  17.5  DDD 21930
              07:10:16 07:17:34 07:25:00  00:14:44   10  81 153  28.4  DDD 21931
              08:51:18 08:58:56 09:06:38  00:15:20   25 107 187  48.1* DDD 21932
              10:31:55 10:39:41 10:47:31  00:15:37   41 131 220  77.6* DDD 21933
              12:12:20 12:20:02 12:27:48  00:15:28   60 334 250  77.5* DDD 21934
              13:52:24 14:00:07 14:07:53  00:15:28   83   1 276  68.9* DDD 21935
              15:32:29 15:40:15 15:48:01  00:15:33  109  20 299  76.7* DDD 21936
              17:12:46 17:20:32 17:28:22  00:15:37  139 226 318  79.1* DDD 21937
              18:53:39 19:01:17 19:08:59  00:15:20  171 253 334  49.3* VVV 21938
              20:35:17 20:42:35 20:50:01  00:14:44  206 277 349  29.2  VVV 21939
              22:17:44 22:24:29 22:31:19  00:13:35  241 303   5  17.9  VVV 21940

Mon  01Oct18  00:00:55 00:07:00 00:13:09  00:12:14  277 330  23  11.9  VVV 21941
              01:44:06 01:49:51 01:55:39  00:11:33  308 357  46   9.8  VVV 21942
              03:26:49 03:32:46 03:38:46  00:11:58  333  24  76  11.1  VVV 21943
              05:08:47 05:15:20 05:21:58  00:13:11  352  51 110  16.0  DDD 21944
              06:50:13 06:57:23 07:04:41  00:14:28    7  76 146  25.7  DDD 21945
              08:31:14 08:38:53 08:46:31  00:15:16   22 102 180  43.3* DDD 21946
              10:12:00 10:19:42 10:27:32  00:15:33   38 124 213  71.4* DDD 21947
              11:52:25 12:00:07 12:07:57  00:15:33   56 334 244  81.3* DDD 21948
              13:32:33 13:40:15 13:48:02  00:15:28   78 355 271  69.3* DDD 21949
              15:12:38 15:20:20 15:28:06  00:15:28  103  20 295  73.9* DDD 21950
              16:52:51 17:00:37 17:08:27  00:15:37  133 228 314  84.9* DDD 21951
              18:33:32 18:41:14 18:49:00  00:15:28  165 247 331  54.7* VVV 21952
              20:15:02 20:22:28 20:29:54  00:14:52  199 273 346  32.3  VVV 21953
```

We only consider passes with a sufficient elevation for the satellite to be clearly visible (typically about 60° as indicated in the `Peak Elev` column) and the time is here given in Universal Time (+1 or +2h with respect to French time).

Many users might prefer a graphical interface to get such results. When an internet connection is available, the simplest solution is probably to fetch the information from the Heavens Above (`www.heavens-above.com`) web site, requesting pass predictions for the satellite identified as NORAD 40069 (Fig. 3): the results are consistent with those of SatTrack, considering that Heavens Above provides results in local time rather than universal time (hence a 2 h difference at the date of October 1st we are concerned with here), that Heavens Above does not consider passes with a maximum elevation below 10°, and that the schedule for the Acquisition Of Signal (ACQ or AOS) and the Loss Of Signal (LOS) is given for an elevation of 10° and not 0°, inducing an offset of about 2 to 5 minutes between the schedules.

When no internet connection is available, the proprietary and now bygone software WXtoImg (`https://wxtoimgrestored.xyz/`) – a long time useful tool for NOAA low-Earth orbiting receiver enthusiasts – can be lured to predict Meteor M2 passes. This exercise is mostly an opportunity to quickly investigate the structure of the Two Line Elements (TLE) orbital parameter descriptions. The information allowing to compute the position of a satellite around the Earth is provided by Celestrak as

METEOR M2 - All Passes

Search period start: 01 October 2018 00:00 < >
Search period end: 11 October 2018 00:00
Orbit: 820 x 826 km, 98.6° (Epoch: 30 September)

Passes to include: ○ visible only ◉ all

Click on the date to see the ground track during the pass.

Date	Brightness	Start			Highest point			End			Pass type
	(mag)	Time	Alt.	Az.	Time	Alt.	Az.	Time	Alt.	Az.	
01 Oct	-	00:20:53	10°	W	00:24:30	18°	WNW	00:28:08	10°	NNW	visible
01 Oct	-	02:04:58	10°	NW	02:06:59	12°	NNW	02:09:00	10°	N	visible
01 Oct	-	05:31:12	10°	N	05:32:44	11°	NNE	05:34:16	10°	NE	visible
01 Oct	-	07:12:02	10°	NNE	07:15:20	16°	NE	07:18:36	10°	E	daylight
01 Oct	-	08:52:54	10°	NNE	08:57:23	26°	ENE	09:01:53	10°	SE	daylight
01 Oct	-	10:33:40	10°	NNE	10:38:50	43°	E	10:44:00	10°	S	daylight
01 Oct	-	12:14:17	10°	NE	12:19:43	71°	SE	12:25:08	10°	SSW	daylight
01 Oct	-	13:54:41	10°	NE	14:00:07	81°	NNW	14:05:33	10°	WSW	daylight
01 Oct	-	15:34:52	10°	ENE	15:40:16	69°	N	15:45:39	10°	W	daylight
01 Oct	-	17:14:56	10°	ESE	17:20:20	74°	NNE	17:25:45	10°	WNW	daylight
01 Oct	-	18:55:10	10°	SE	19:00:36	85°	SW	19:06:03	10°	NW	daylight
01 Oct	-	20:35:56	10°	S	20:41:14	55°	WSW	20:46:35	10°	NW	visible

Figure 3: Pass previsions using the web site Heavens Above.

```
METEOR-M 2
1 40069U 14037A   18272.86150993 -.00000016  00000-0  12287-4 0  9997
2 40069  98.5921 321.1231 0004724 305.5966  54.4754 14.20654510219244
```

The name of the satellite is given as a free text field in the first line, followed by the orbital parameters starting with the line number and the NORAD identifier of the satellite, followed for its first appearance by the classification level of the satellite ("U"nclassified, "C"lassified, "S"ecret). Making WXtoImg believe that we will predict the pass schedule of a satellite it is designed to work with – NOAA APT – we must forge an erroneous TLE with the orbital parameters of Meteor M2 and the identifier of a NOAA satellite. Let us for example consider NOAA 15 which is no longer fully operational

```
NOAA 15
1 25338U 98030A   18283.48448267  .00000100  00000-0  60924-4 0  9992
2 25338  98.7662 299.9712 0009347 251.7061 108.0475 14.25879899 61199
```

Having observed that WxtoImg does not consider the first line, we only modify the identifier in the Meteor M2 sentences by the NORAD identifier of NOAA 15, namely 25338. Such a simple modification however fails: the last number of each line is indeed a checksum computed by summing all the characters of each line modulo 10, considering that the "-" is equal to 1 and other letters and symbols are not considered when computing the sum. Using GNU/Octave, this computation is achieved with

```
a="2 25338  98.7662 299.9712 0009347 251.7061 108.0475 14.25879899 6119"
a=strrep(a,'-','1');mod(sum(a(find((a<='9')&(a>'0')))-48),10)
```

which, in this example, answers 9, which is indeed the last number of the second TLE line of NOAA 15. Having understood these issues, we forge the new sentence

```
METEOR-M 2
1 25338U 14037A   18272.86150993 -.00000016  00000-0  12287-4 0  9999
2 25338  98.5921 321.1231 0004724 305.5966  54.4754 14.20654510219246
```

in which the NORAD Meteor M2 identifier was replaced with the one from NOAA 15, and the new checksums were computed and updated. Under such conditions, WxtoImg performs the computation and displays the result, consistent with Heavens Above and SatTrack. Indeed, the output shown on Fig. 3 compares favorably with the output of WxtoImg shown below

```
Satellite passes for ny-alesund, norway (78o55'N 11o54'E)
while above 0.1 degrees with a maximum elevation (MEL) over 40.0 degrees
from 2018-09-30 21:46:04 CEST (2018-09-30 19:46:04 UTC).

2018-10-01 UTC
  Satellite   Dir  MEL  Long         Local Time    UTC Time Duration      Freq
  NOAA 15       S  43E   40E     10-01 10:31:13    08:31:13    15:11  137.6200
  NOAA 15       S  71E   20E     10-01 12:11:57    10:11:57    15:30  137.6200
  NOAA 15       S  81W    9E     10-01 13:52:22    11:52:22    15:29  137.6200
  NOAA 15       S  69W   10E     10-01 15:32:33    13:32:33    15:25  137.6200
  NOAA 15       N  74E   16E     10-01 17:12:38    15:12:38    15:26  137.6200
  NOAA 15       N  85W   10E     10-01 18:52:51    16:52:51    15:31  137.6200
  NOAA 15       N  55W    7W     10-01 20:33:34    18:33:34    15:24  137.6200
```

We have seen that the 2 minute difference between SatTrack and other prediction software lies in the fact that the former considers that AOS is achieved when the satellite rises over the horizon while the other two consider higher elevations (with a default value of 8°). We notice in all cases the benefits of working at higher latitudes (here 79°N) where passes repeat every 100 minutes, allowing for many more attempts than at the French latitudes.

We could not resist while watching S. Prüfer present *Space Ops 101* at `media.ccc.de/v/35c3-9923-space_ops_101#t=1265` the urge to reproduce the figures exhibited between 16 and 18 minutes along the presentation illustrating why listening to satellite from polar regions is most favorabl. In order to achieve the results depicted below using QGis (here version 3.4):

1. install a tool for predicting satellite positions based on the TLE and providing a Shapefile formatted output: we have used `https://github.com/anoved/Ground-Track-Generator/` which is trivially compiled,

2. generate the ground trace of the satellite we are interested in. By providing the Meteor M2 TLE collected around October 1st 2018 on the Celestrak web site as described in the text, we execute
   `gtg --input meteor.tle --output m2 --start epoch --end epoch+24h --interval 30s`
   to get the `m2.shp` file which includes the position, in spherical coordinates (WGS 72), for the satellite as seen from ground,

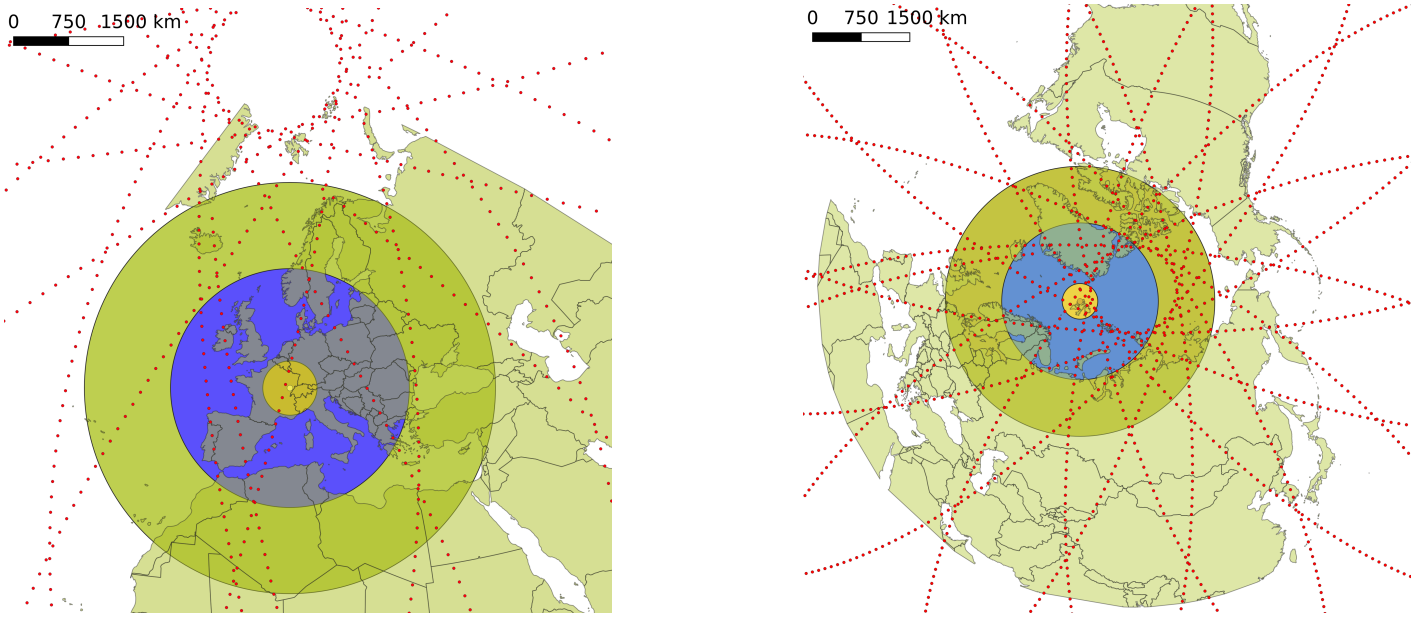
3. download a coast and border database, again in Shapefile format. In our case we used the archive available at `https://www.naturalearthdata.com/downloads/50m-cultural-vectors/50m-admin-0-countries-2/`

4. we must always work in a projected cartesian framework to process geometrical transforms: all spherical coordinates are hence projected to a cartesian framework in the local tangent planes. For France, WGS84/UTM31N yields good results (EPSG:32631) while `https://epsg.io/3576` teaches us that EPSG:3576 will provide an acceptable projection framework around the North Pole

5. having clipped (`Vector` → `Geoprocessing Tools` → `Clip ...`) the border map to the countries around the Arctic regions (down to about fifty degree north: such a result is achieved by creating a polygon using `Layer` → `Create Layer` → `New Shapefile Layer...` → `Geometry Type: Polygon`), place the receiver location site on the map, for example by importing an ASCII file including its longitude and latitude.

6. Trace circles of known circumference on the map. Doing so is achieved by saving the receiver site location coordinates in a cartesian framework (right mouse button on the layer including the symbol of the receiver site, and `Export` → `Save Feature As ...` by selecting a CRS with the appropriate projection) then we will trace an exclusion zone with `Vector` → `Geoprocessing Tool` → `Buffer ....` We are left with identifying the circumference of these circles. The case of the angle with respect to the center of the Earth at which the satellite appears over the horizon is trivial since we have a right triangle between the observer, the center of the Earth at radius R, and the satellite at a distance $R + r$ from the center of the Earth (r the flight altitude of the satallite). Hence, the angle between the observer, the center of the Earth and the satellite is $\vartheta = \arccos(R/(R + r))$ and the length of the arc visible from the satellite is $R\cos\vartheta = R \cdot R/(R + r) = 3050$ km. This yields the radius of the largest circle visible on the figures exhibited below. Practically, we can hardly receive a usable signal from an elevation below $\vartheta' = 15°$. In this case, identifying the radius of the circle defining the visibility zone requires a bit more complex trigonometric relations, by replacing the right angle with an angle equal to $(90 + \vartheta')$, but the solution to the problem remains unique for a known given angle (the satellite angle at AOS), a Earth center-observer distance known (R) and a Earth center-satellite distance known ($R + r$). The solution for $\vartheta' = 15°$ is a radio of the visibility zone of 1760 km. Finally, we shall not bother with facing the Arctic cold if the satellite maximum elevation during a pass remains below $\vartheta' = 60°$. In this case, the radius of the satellite visibility zone for such a minimum elevation is reduced to 400 km around the observer location. These two circles are concentric and centered on the observation site on the figures below.

We deduce, by observing these figures, that a single pass at best will yield usable results every day over France with an elevation of at least 60°, while 9 to 10 passes out of the daily 14 will meet this condition over Spitsbergen.



Left: ground trace (red dots) of Meteor M2 passes as seen from Besançon during 24 h. Right: ground traces of Meteor M2 passes as seen from Ny Ålesund in Spitsbergen. Notice that a single trace lies within the circle defining an elevation of at least 60° during a pass over France, while at least 9 orbits meet this condition over Spitsbergen. The left figure is given in WGS84/UTM31N projection, while the one on the right is WGS84/North Pole LAEA Russia. The largest circle centered on each observer site indicates when a satellite becomes visible over the horizon (3030 km radius), the intermediate circle indicates a visibility at elevations of at least 15° (1760 km radius), and the smallest circle describes locations at which the elevation of the satellite is at least 60° (400 km radius).

3 Why such a complex protocol ?

APT for communication analog signal emitted from NOAA satellites is trivial [6]: a dual amplitude (pixel intensity) and frequency (to get rid of Doppler shift as the satellite travels along its orbit) modulation transmits the information decoded on the ground by successive FM followed by AM demodulators. Why then leave this simple communication protocol at the expense of a digital communication mode embedding packets in multiple protocol layers (Fig. 4) which will keep us busy along the pages that follow ?

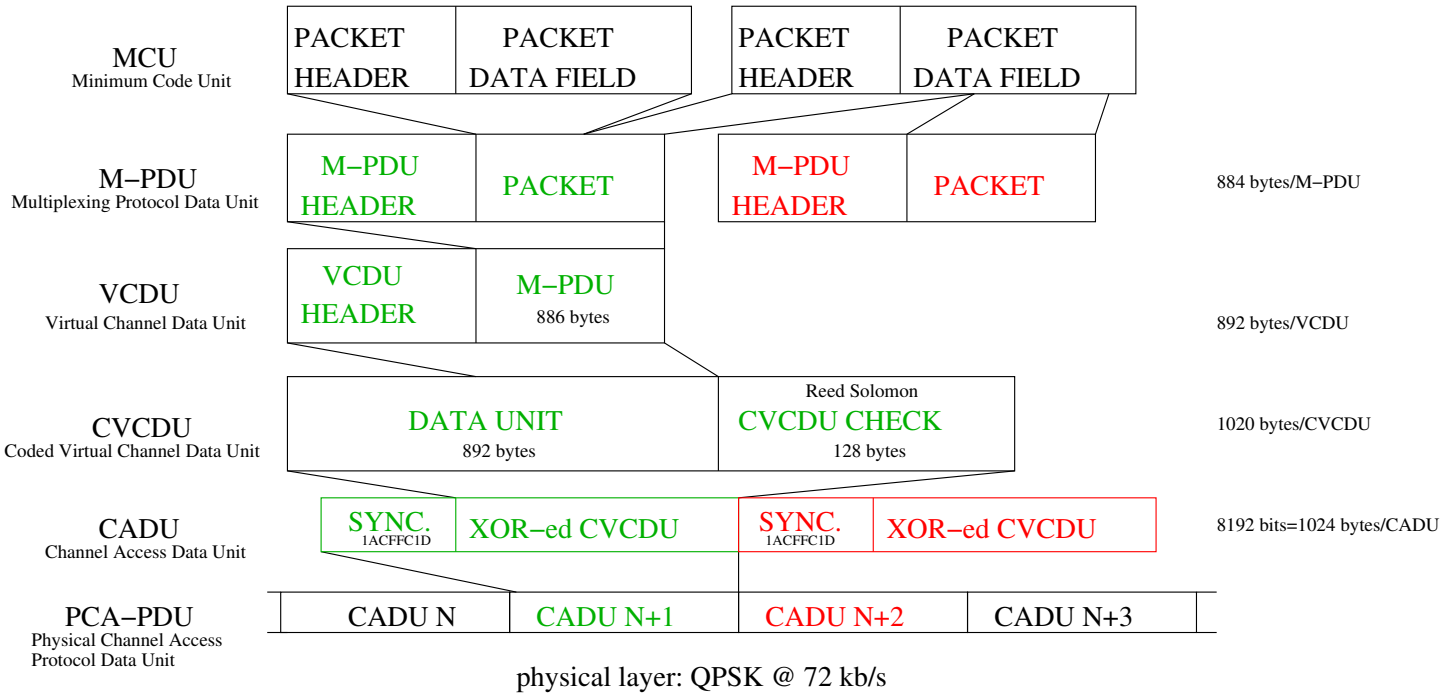


Figure 4: Protocol layers to be addressed to convert the physical signal (bottom) to an image (top): the complexity of the protocol lies in its general purpose and the virtualization of multiple digital channels transmitted by the satellite.

The amount of information transmitted is limited by the allocated bandwidth, but a given bandwidth can be used more or

less efficiently to transmit an image with a better or worse quality. The radiofrequency spectrum is a scarce and busy resource: Meteor-M2 allows for an improved resolution for a given spectral usage thanks to the spectrum optimization provided by the digital modulation. More importantly, this data encapsulation as packets in successive layers – similar to the OSI layers for ground networks – fits in with a logic of sharing resources needed for space communication as illustrated in Fig. 5 (NASA image) following the presentation [7].

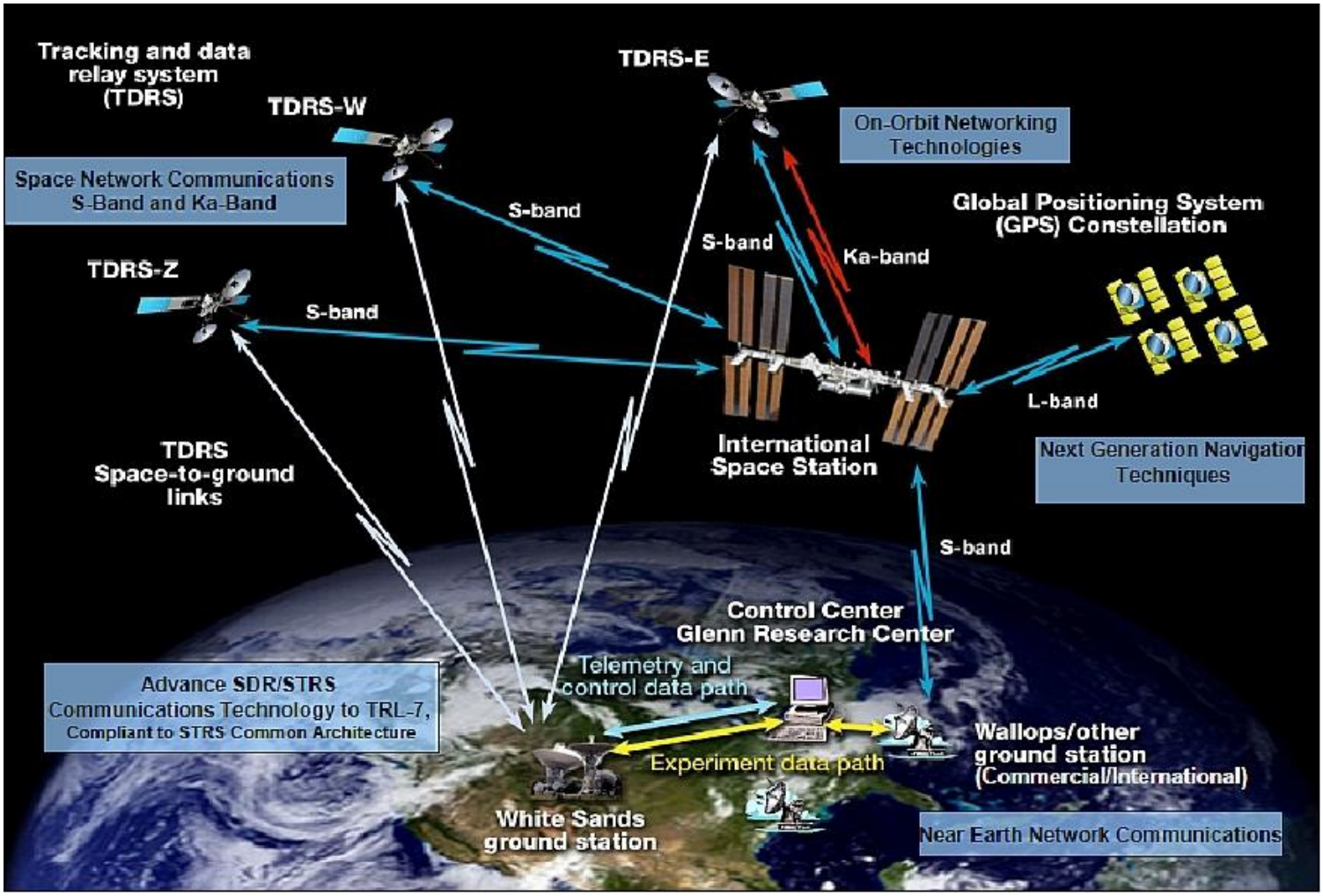


Figure 5: Illustration depicting the complexity of space communications to transmit information acquired around terrestrial orbits, especially from low Earth orbiting satellites (ex-space shuttle and now ISS at 400 km, Hubble at 550 km) which could not be efficiently used and monitored by being visible only a few minutes every day from a ground station. Only by using a network of geostationary communication satellites (TDRS in the United States, future EDRS in Europe for the Sentinel satellites) ensures a nearly permanent link between a space agency and its satellites. Picture taken on the web site `earth.esa.int/web/eoportal/satellite-missions/i/iss-scan`.

As an example, a low Earth orbiting satellite such as the International Space Station (ISS, 400 km altitude) or a weather satellite (800 km altitude) will travel from horizon to horizon in 7 to 11 minutes under best circumstances (maximum elevation at zenith). With a period of 90 minutes, this means that for a continuous link with the ISS, 13 stations would be needed along each orbit, or one station every 3000 km, not very practical to implement despite being used at the early days of the space race [8]. The solution is to communicate through the TDRS (Tracking and Data Relay Satellite), a set of geosynchronous satellites acting as relays between the low and mid-Earth orbiting satellites and ground. This means that not only is each flying platform fitted with a multitude of instruments which must share the available bandwidth and hence use a communication protocol for sharing the communication channel more subtle than simply sequentially communication each instrument result as was seen for NOAA [6], but also that a given satellite might be used to route information from different origins, and later from different orbits (e.g Moon or Mars [9]). Such functionalities are for example mandatory to fully exploit a satellite such as the Hubble space telescope orbiting at about 500 km from the surface of the Earth, which is no more no less than a spying satellite looking in the wrong direction. All these elements hints at the development of a packaging and routing protocol which must be robust to the temporary visibility of the satellites from the ground station, with possibly the ability to hand-over from one station no another due to Earth rotation (see for example the Deep Space Network and its stations distributed on all continents) without the final used being aware of these successive data sources.

Hence, we find again the same problems of IP packet routing followed by encapsulation of data in TCP or UDP packets, but

without the robust and user-friendly libraries provided by the various opensource implementations of the networking layers. We must hence unstack ourselves the layers of the protocol to understand each subtle operating principle. Luckily, all information is available, assuming we know where to look for them [1].

4 How to tackle the challenge ?

Communicating digital data on a radiofrequency communication channel as variable as a space link requires a few data protection strategies to prevent corruption and losses, and even correction capability. These various protocol layer are described in documents published by the CCSDS, the Consultative Committee for Space Data Systems, at `public.ccsds.org` (Fig. 6).

5.2.1 BLOCK DIAGRAM OF THE GROUND SEGMENT

Figure 5-2 shows a functional block diagram of the user ground segment.

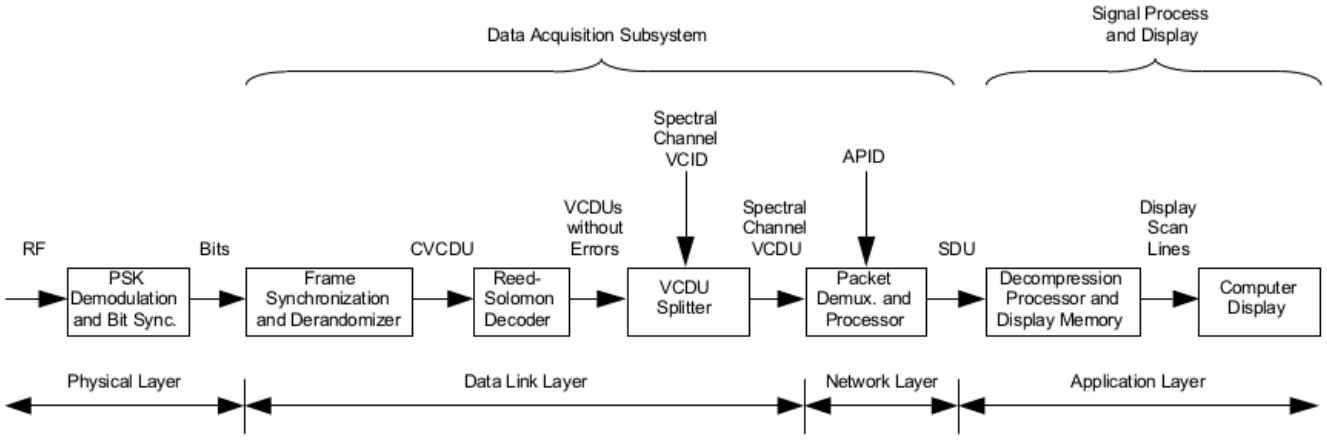


Figure 6: OSI layers and acronyms used in the literature describing the LRPT image transmission. This image is extracted from CCSDS 120.0-G-2 at `public.ccsds.org/Pubs/120x0g2s.pdf`

.

Reading the document is, to say the least ... challenging. Let us attempt to ease the challenge by starting from the end (transmitting an image) and reach the signal we have received (the radiofrequency wave):

1. an image is split to be transmitted by the satellite as a set of 8×8 pixel thumbnails,
2. each one of these thumbnails is compressed (lossy compression) using JPEG: each image thus exhibits a variable size depending on the amount of details being displayed in each thumbnail (few coefficients for a smooth area, many coefficients for areas with many features such as mountains). These steps of image assembly are discussed in section 7,
3. one line of the final image is made of 196 thumbnails, for a total width of 1586 pixels
4. transmitting the images collected a various wavelengths (various instruments) is alternated by sending the set of 196 thumbnails from one wavelength, then the 196 thumbnails from another wavelength. Between image transmission, a telemetry sentence is transmitted (section 5.7)
5. these variable size datasets are collected in fixed-size packets. Each packet holds a payload made of 892 bytes followed by 128 bytes for an optional transmission error correction, to which a 4 byte header for synchronization is added (total: 1024 bytes). This grouping of bytes into sentences is described in section 6,
6. a convolutional code, which will be described in details since it is the core theoretical challenge of the whole work, allows for correcting noise distributed uniformly during the transmission. Each bit is doubled to create sentences which are $2 \times 8 \times 1024 = 16384$ bit long (section 5.3),
7. the hardware layer is defined as a QPSK (4-PSK) transmission in which each bit pair is encoded as one of four possible phase states $\{0, 90, 180, 270\}^\circ$. This transmission runs at a rate of 72 kb/s (section 5).

Having described the outline of the encoding which follows the outline of the OSI layers (Fig. 6), we must unwrap the problem in the opposite direction to go from the radiofrequency signals received by a digital video broadcast-terrestrial receiver used as I/Q coefficient source for software defined radiofrequency signal processing.

[1] notice that the author of `libfec` library which will be used here, Phil Karn KA9Q, is also the author of the TCP/IP stack for MS-DOS that we used during our first steps to to discover internet connectivity at the beginning of Linux in 1994/1995, when MS-DOS was still the most common operating system running on personal computers

5 From the radiofrequency signal to bits

Acquiring digital signals following frequency transposition to get to baseband does not involve any significant challenge: the local oscillator of the DVB-T receiver used as sample source for radiofrequency signal processing is tuned to the center frequency of the emitted signal – 173.9 MHz in the case of Meteor M2 – and the bandwidth adjusted to be wide enough to collect all spectral components of the signal modulating the carrier. He have discussed at length in [10] how phase modulation requires regenerating on the receiver side a local copy of the carrier prior to modulation in order to identify the phase of the signal by mixing and filtering. In the case of binary modulations (BPSK – Binary Phase Shift Keying), we have seen [11, 10] that the un-modulated carrier was recovered by processing the (I, Q) coefficients with an estimator insensitive to π rotations (since the encoding is added to the carrier by rotating the phase to 0 or π), either by using the $\arctan(Q/I)$ function or by squaring the signal in order to double the phase, namely 0 or 2π, and hence a carrier having lost its modulation but at double the frequency offset between the emitted signal and the frequency of the local oscillator on the receiver side.

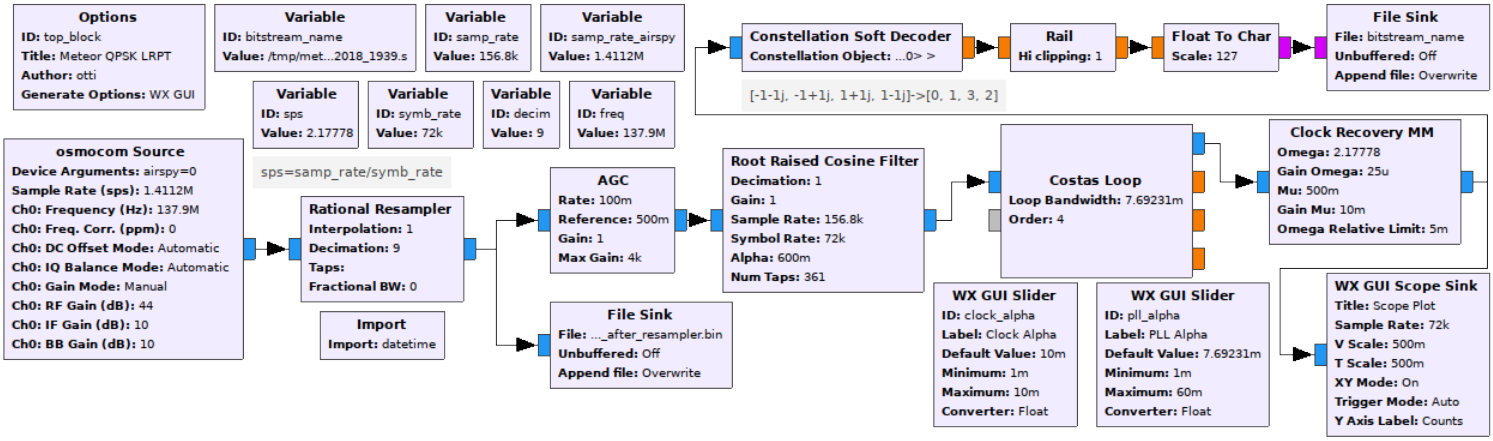


Figure 7: Acquisition sequence aimed at minimizing the size of the file storing data by maximizing the number of processing steps taken care of by GNU Radio, namely creating a copy of the carrier (Costas loop) and the clock synchronizing data sampling, in order to recover the bit sequence and only save one 8-bit sample for each information (bit) transmitted. These samples with be discretized (1 or 0) during post-processing taken care of later.

The same principle is exactly transposed to quadrature phase modulation, in which the information is applied as phase rotations of the carrier with values equal to 0, $\pi/2$, π or $3\pi/2$ (Fig. 7, strongly inspired by `github.com/otti-soft/meteor-m2-lrpt/blob/master/airspy_m2_lrpt_rx.grc`). However, instead of simple squaring, getting rid of the phase now requires computing the fourth power of the signal, yielding a beat signal at four times the offset between the emitted frequency and the receiver local oscillator (Fig. 8) frequency. Hence, for a given acquisition bandwidth, we can only allow for a lower difference between these two frequencies than in the case of BPSK.

We have seen on the other hand [10] that once the carrier has been reproduced (Costas), the question of the bit sampling rate remains since the emitter and receiver clocks are not synchronous, hence the need to detect transitions from one bit value to another to control the clock sampling the phase. This job is taken care of by clock synchronization blocks such as *Clock Recovery Mueller & Müller* or *Polyphase Clock Sync* which aim at only providing a single sample for each symbol after controlling the clock sampling the datastream on its transitions.

These two tasks are taken care of by GNU Radio not only because they are perfectly functional in this environment, but most significantly to reduce the size of the files stored for further processing. The more the datastream is decimated prior to storage, the smaller the file: in our case, we aim at storing the bit stream as the output of the processing chain, or one byte representing each bit since the recovered values have not yet been discretized to 1 or 0 but remain a probability to be maybe 1 or maybe 0. We will see that keeping this uncertainty, considering the clever encoding scheme used during emission, will maximize our chance of recovering the correct value of each bit. This data storage format is named **soft samples**, as opposed to **hard samples** which have already been discretized to attribute a value of 0 or 1 to each bit [12, p.8].

5.1 Data format

The first question we might wish to answer is whether our understanding of the data format is correct and if the file is worth processing. Before having the slightest idea on the encoding format, we might simply wonder whether a pattern is repeated. Indeed, when encapsulating messages as packets, it is quite likely that the size of packets is constant, and that some pattern such as the header will repeat. Hence, the autocorrelation of the signal must exhibit some peaks spaced by the repetition period of the messages (Fig. 9).

We observe correlation peaks every multiple of 16384 samples (bytes). Trespassing on the description that will follow, we will learn that each packet transmitted by Meteor M2 is 1024 byte long or 8192 bits, and that the coding scheme used (convolutional code [13]) doubles the number of bits to 16384, while in the soft bit format we have one byte representing each bit of the transmitted

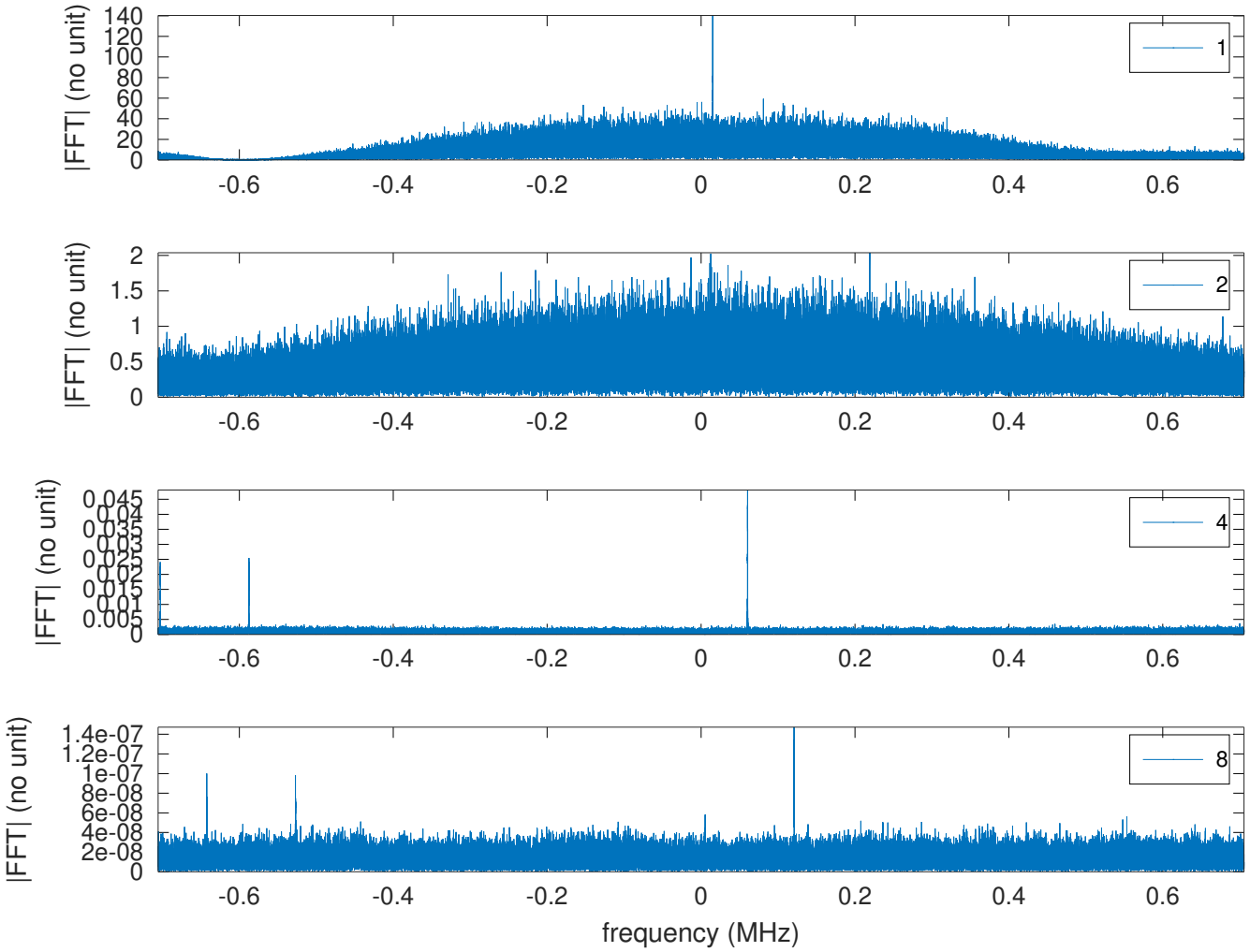


Figure 8: Top to bottom: spectrum of the I+jQ coefficients exhibiting the spectrum spreading ; squaring the signal ; computing the fourth power and the eighth power. Squaring the signal does not allow for spread spectrum compression to recover the carrier: the modulation is not BPSK. The fourth power allows for removing the modulation and only the carrier is left: the modulation is QPSK.

message. The position of the correlation peaks observed in Fig. 9 is hence indeed consistent with the expected shape of the signal: not only have we verified that we understand how to read the file saved by GNU Radio and interpret correctly its content, but we know that the acquired signal contains the information transmitted by the satellite and is worth further analysis.

5.2 Decoding data

We have so far obtained a sequence of bytes whose value will be most probably be equal to 1 or 0. As with all continuous streams of data, we must have a starting point to know when to start processing the bit stream, assemble them into letters (bytes), then words, sentences and paragraphs. The classical approach to identify the beginning of a transmission is to provide a known sequence in the continuous bitstream, and search for the occurrence of this pattern. The estimator of resemblance between the successive phases of the signal and the searched pattern is the cross-correlation. Indeed, the technical documentation of LRPT [14] teaches us that all space transmissions are synchronized on the word `0x1ACFFC1D`. The job sounds easy: by cross-correlating this word, we shall find the synchronization as the maximum of the cross-correlation.

Not so fast. First of all, the bits received from a satellite orbiting at more than 800 km from the surface of the Earth are corrupted by noise. We must thus search for some kind of repeated pattern to maximize the chances of properly detecting the transmitted message. A basic approach would consist in simply repeating the message multiple times, but how to select the good one if two transmissions are not identical ? Better, convolutional encoding uses as input a continuous bit stream, and creates a new (longer) sequence so that each new bit is a combination of the input bits. This combination is designed to maximize our chances of recovering the initial message: this processing step is convolutional encoding. Bits encoded this way no longer exhibit the synchronization word sequence `0x1ACFFC1D` buts its convolutional encoded version, which must be determined and searched

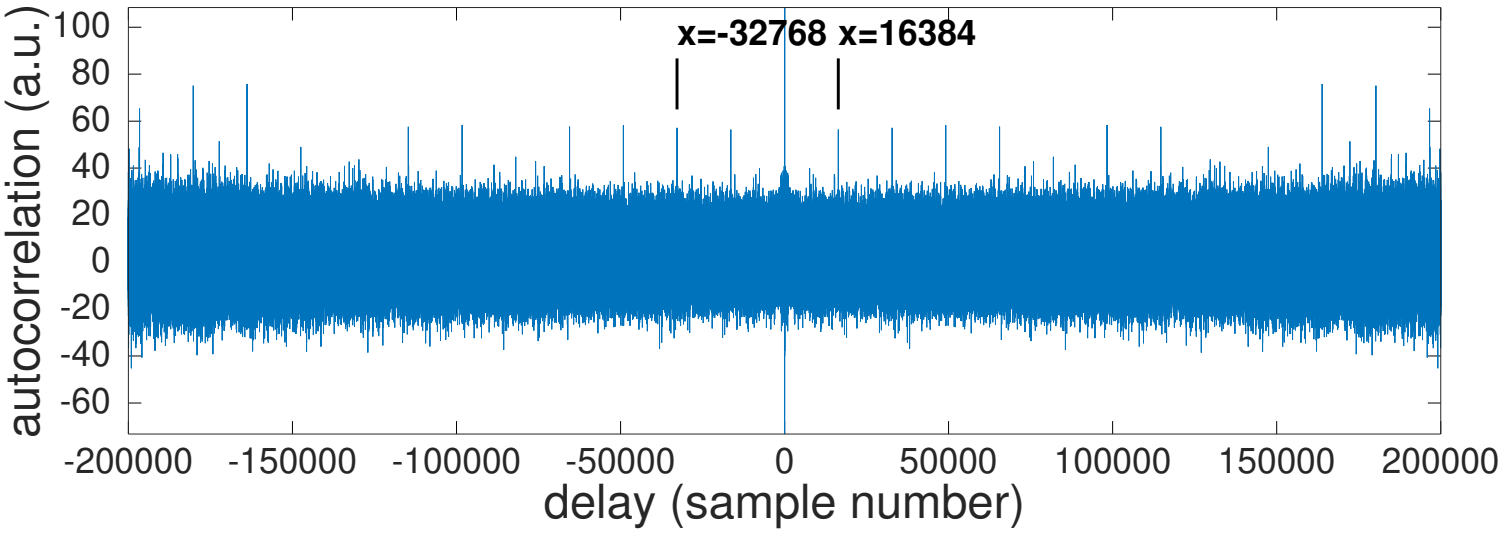


Figure 9: Autocorrelation of 400 ksamples of soft bits stored as the output of the GNU Radio demodulating chain that was in charge of recovering the carrier frequency and the QPSK clocking rate. Correlation peaks are seen every 16384 samples.

for in the received bit stream.

5.3 Convolutional encoding of the synchronization word

In order to maximize our chances of recovering the value of each bit, a convolution code spreading the information over time is used in order to introduce redundancy. While encoding is excessively simple to implement, decoding the most probable sequence following corruption of some of the bits during transmission is possibly very complex. An optimal approach to the dual problem is to implement decoding as the Viterbi algorithm [15] – named after its author, also co-founder of the Qualcomm company – and we must thus master these concepts in order to recover usable bit sequences.

Input data are represented as *soft bits*, or sequences of 8-bit values encoding each one possible bit value. The convolution to generate the emitted bits has used as input each source bit, and created two output bits as combination of a given number of input bits: the convolution algorithm is qualified as $r = 1/2$ since it provides twice as many bits on its output as input bits, and $K = 7$ since the shift register used as memory of the input bit sequence is 7-bit long [14, p.23]. Convolution can be tackled in many ways: one approach, closest to the hardware or programmable logic (FPGA) implementations of the convolutional encoding, consists in a shift register used as a memory, fed by the new bit of the sequence to be encoded, and feeding one or more XOR gates to provide $1/r \geq 1$ output bits for each input bit (Fig. 10). One way of defining which bits of the shift register feed the XOR gate is to provide the polynomial whose non-null powers match the connecting points from the shift register to the XOR gate [16] (Fig. 10, reading from right to left the binary representation of the byte defining each polynomial coefficient). From this polynomial expression of the convolutional code, we can state the sequence of operations as a matrix operation, as described at `http://www.invocom.et.put.poznan.pl/~invocom/C/P1-7/en/P1-7/p1-7_1_6.htm` by considering that the register content shift at each time step is achieved by shifting the polynomial coefficients along the (long) bit sequence to be encoded. This *generator matrix* expression is natural when considering a Matlab implementation since it expresses the convolution as a matrix operation, and is programmed in this simple case in GNU/Octave by:

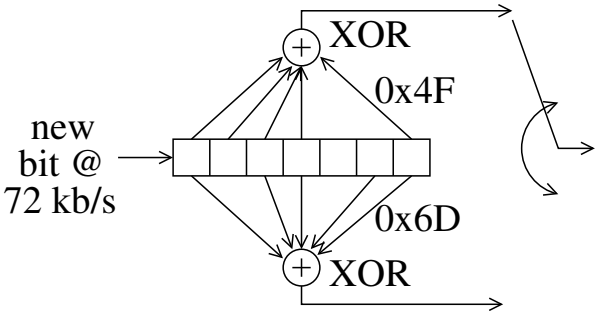


Figure 10: Convolutional coding: bits in the shift register are sampled at the positions whose indices define the polynomials, added as binary values (exclusive OR operation) and the two resulting bits are concatenated as output, inducing an output data rate double of the input data rate.

```
d=[0 1 0 1 1 1]
G1=[1 1 1]  % 0x7  r=1/2, K=3 (3-bit long shift register)
G2=[1 0 1]  % 0x5
G=[G1(1) G2(1) G1(2) G2(2) G1(3) G2(3)   0     0     0     0     0    0;
     0    0    G1(1) G2(1) G1(2) G2(2) G1(3) G2(3)   0     0     0    0;
     0    0       0    0   G1(1) G2(1) G1(2) G2(2) G1(3) G2(3)   0    0;
     0    0       0    0     0     0   G1(1) G2(1) G1(2) G2(2) G1(3) G2(3);
     0    0       0    0     0     0     0     0   G1(1) G2(1) G1(2) G2(2);
     0    0       0    0     0     0     0     0     0     0   G1(1) G2(1);
]
mod(d*G,2)  % matrix product modulo 2 = XOR
```

with the alternating coefficients of the two polynomials named `G1` and `G2`, which indeed provides a result consistent with the one shown at `home.netcom.com/~chip.f/viterbi/algrthms.html`: `0 1 0 1 1 1`→`0 0 1 1 1 0 0 0 0 1 1 0`.

The creation of the convolution matrix, generated here manually, is generalized to the case we are interested in of a code made of two polynomials applied to a 7-bit long shift register, with

```
% bytes to be encoded
d=[0 0 0 1 1 0 1 0  1 1 0 0 1 1 1 1  1 1 1 1 1 1 0 0  0 0 0 1 1 1 0 1 ]; % 1A CF FC 1D
G1=[1 1 1 1 0 0 1 ] % 4F polynomial 1
G2=[1 0 1 1 0 1 1 ] % 6D polynomial 2
Gg=[];
for k=1:length(G1)
  Gg=[Gg G1(k) G2(k)]; % creates the interleaved version of the two generating polynomials
end
G=[Gg zeros(1,2*length(d)-length(Gg))] % first line of the convolution matrix
for k=1:length(d)-length(G1)
  G=[G ; zeros(1,2*k) Gg zeros(1,2*length(d)-length(Gg)-2*k)] ;
end
for k=length(Gg)-2:-2:2
  G=[G ; zeros(1,2*length(d)-(length(Gg(1:k)))) Gg(1:k)];
end     i             % last lines of the convolution matrix
res=1-mod(d*G,2);      % mod(d*G,2)
dec2hex(res(1:4:end)*8+res(2:4:end)*4+res(3:4:end)*2+res(4:4:end))'
printf("\n v.s. 0x035d49c24ff2686b or 0xfca2b63db00d9794\n")
```

We observe by executing these few lines that encoding the synchronization word 1ACFFC1D00 with the two polynomials 4F and 6D defining the taps of the XOR gates to the 7-bit long shift register yields the sequence FCA2B63DB00D9794 which is indeed the one given at [2]. Note that it seems that two standards seem to exist within NASA, in which G1 and G2 seem to be swapped. Hence, some of the encoding and decoding software available on the web stating that they use the right code do not yield the expected result. We now understand how to encode the synchronization word using convolutional code.

5.4 Convolutional code representation as state machines

We shall need later, when describing the Viterbi decoding algorithm, to understand state transitions, i.e. not only understand convolutional codes in terms of matrix operations but also as a finite state machine with a decision to be taken considering the most probable path from one state to another. How to express the problem we have just described as a matrix product as a state machine ?

The convolutional code we are interested in uses a 6-bit long shift register to which another new 7th bit is inserted as new input data. The encoder hence has $2^6 = 64$ possible states. We cannot show them all here, but will only exhibit the first ones needed to encode the first byte of the synchronization word.

Let us start from a state in which all bits in the shift register are equal to 0 (state that we shall call "a"). Two possible outcomes are possible: either the new input value is 0 or 1, so in the previous Octave code we have the two cases `d=[0 0 0 0 0 0 0]` or `d=[1 0 0 0 0 0 0]`. In the former case, the two output bits generated by the convolutional code are `mod(d*G1',2)=0` and `mod(d*G2',2)=0` or 00, and in the latter case `mod(d*G1',2)=1` and `mod(d*G2',2)=1` or 11. In the former case the output state is the same as the input state so that a→a while in the latter case the "1" value was input in the register so we have reached the internal state [1 0 0 0 0 0] which will be called "b". This analysis continues to generate the following table:

Input bit	Input state	Name	Output bits	Output state	Transition
0	000000	a	00	000000	a→a
1	000000	a	11	100000	a→b
0	100000	b	10	010000	b→c
1	100000	b	01	110000	b→d
0	010000	c	11	001000	c→...
1	010000	c	00	101000	c→...
0	110000	d	01	011000	d→e
1	110000	d	10	111000	d→...
0	011000	e	00	001100	e→...
1	011000	e	11	101100	e→f
0	101100	f	01	010110	f→...
1	101100	f	10	110110	f→...
...	...	...	...	...	...
0	100001	u	01	010000	u→c
1	100001	u	10	110000	u→d
...	...	...	...	...	...
0	000001	z	11	000000	z→a
1	000001	z	00	100000	z→b

where the ellipsis (...) in the names of the final state represent cases that are not needed when encoding the first byte of the synchronization code (state "c" will not be needed either, but is included to make the demonstration clearer). States "u" and "z"

are inserted to exhibit loops that will appear when the state machine is followed long enough. We invite the reader to ascertain by himself this result to be convinced of the relevance of the approach.

This state transition table is exploited to illustrate how the 0x1A byte (first byte of the synchronization word) is encoded. The first three bits at 0 of the most significant nibble are encoded as 00 and keep the state machine in state "a", while the last bit at 1 is encoded as 11 and leads to state "b". The most significant bit of the A nibble is 1 and we are in state "b" so that we generate 01 and reach state "d". The next bit is 0 (reminder: 0xA=0b1010) and with the current state at "d" we output 01 to reach "e". State "e" takes as input 1 to generate 11 and reach "f", and finally "f" with an input of 0 generates 01. As a summary, by encoding 0x1A we have produced 0b0000001101011101=0x035D which is indeed the expected value (notice that 0x035D is the opposite of 0xFCA2 that was mentioned earlier as the beginning of the encoding solution – both solutions are the same assuming a 180-degree phase rotation of the bits).

A state machine representation can hence be given as shown in Fig. 11, with on top the successive states from "a" to "f" as a function of the input bit, and on the bottom the output bits for each of these transitions.

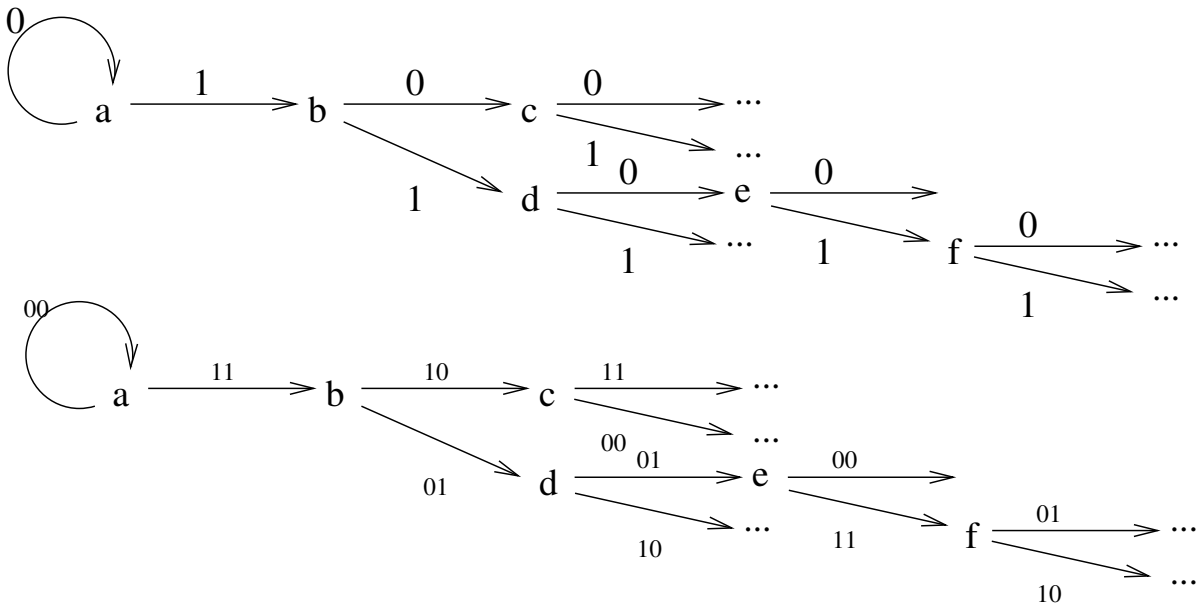


Figure 11: Top: states, named from "a" to "f", and transitions as a function of the input bit value. Bottom: output bit values as a function of the transitions. By following the path described in the text, the input sequence 0x1A=0b00011010 is encoded as 0b0000001101011101=0x035D.

5.5 Decoding a convolutional code: Viterbi algorithm

Having described the state machine used to convert an input word to an output word with double length, we now wish to understand the decision sequence that will maximize the probability of reverting the process, considering that some of the received data might have been corrupted during the transmission. Let us consider the result before addressing the explanation.

We will use later a library efficiently implementing (in C language) convolutional encoding and decoding: `libfec` (`github.com/quiet/libfec`) is described at [2] and its sample code `vtest27.c` is used as starting point to implement the decoder. Alternatively, `www.spiral.net/software/viterbi.html` provides a code generator to decode, using the Viterbi algorithm, meeting the requirements of LRPT. We start assessing `libfec` with the simple case of decoding the sentence encoded as `FC A2 B6 3D B0 0D 97 94` – that we have already seen as resulting from the convolutional code encoding of the synchronization word – to check whether we are indeed able to recover the initial word:

```
1  #include <stdio.h>  // gcc -Wall -o t t.c -I./libfec ./libfec/libfec.a
2  #include <fec.h>
3  #define MAXBYTES (4)  // final message is 4-byte long
4
5  #define VITPOLYA 0x4F // 0d79  : polynomial 1 taps
6  #define VITPOLYB 0x6D // 0d109 : polynomial 2 taps
7
8  int viterbiPolynomial[2] = {VITPOLYA, VITPOLYB};
9  unsigned char symbols[MAXBYTES*8*2]=  // *8 for byte->bit, and *2 Viterbi
10       {1,1,1,1,1,1,0,0, // fc
11        1,0,1,0,0,0,1,0, // a2
12        1,0,1,1,0,1,1,0, // b6
13        0,0,1,1,1,1,0,1, // 3d
```

```
      1,0,1,1,0,0,0,0, // b0
      0,0,0,0,1,1,0,1, // 0d
      1,0,0,1,0,1,1,1, // 97
      1,0,0,1,0,1,0,0};// 94

int main(int argc,char *argv[]){
  int i,framebits;
  unsigned char data[MAXBYTES]; // *8 for bytes->bits & *2 Viterbi
  void *vp;
  framebits = MAXBYTES*8;
  for (i=0;i<framebits*2;i++) symbols[i]=1-symbols[i];   // flip bits
  for (i=0;i<framebits*2;i++) symbols[i]=symbols[i]*255; // bit -> byte /!\
  set_viterbi27_polynomial(viterbiPolynomial);   // definition of taps
  vp=create_viterbi27(framebits);
  init_viterbi27(vp,0);
  update_viterbi27_blk(vp,symbols,framebits+6);
  chainback_viterbi27(vp,data,framebits,0);
  for (i=0;i<MAXBYTES;i++) printf("%02hhX",data[i]);
  printf("\n");
  exit(0);
}
```

with the only **subtlety of the code**, which takes as input the successive bit values (here hard bits since already saturated as 0 or 1) of the word to be decoded, to encode each bit **on the whole range of the a byte (hence 0 to 255 and not 0 or 1)**, and possibly flipping the bits ($0 \leftrightarrow 1$) in order not to recover the complement of the word we are looking for (phase rotation by 180° of the initial phase modulation for example). By executing this code, we indeed recover `1ACFFC1D` which is the word we are looking for, so we understand how to use `libfec`. Let us try now to understand the underlying algorithm.

The convolution code implemented during the encoding step was designed to introduce some memory in the bit sequence in order to make the transmission robust to random errors which might be introduced by a uniformly distributed noise (as opposed to noise bursts that would corrupt a whole sequence of bits). The problem of decoding hence means going through the state machine proposed in Fig. 11 the other way around, aiming to find the most probable path considering the received bit sequence. Let us consider that we have received the sequence 0b0000001111011101: what emitted bit sequence has most probably generated the transmitted code ? The Viterbi decoder is initialized with all bits set to 0, or state "a" if referring to Fig. 11. We receive 00, so we stay in state "a" and know that the transmitted bit was 0. Similarly for the next to 00 sequences that follow, and the pair 11 indicates that we have switched to "b" after a 1 was encoded. When receiving 11 we are in state "b", which is not consistent: "b" can only produce 01 (if 1 had been transmitted) or 10 (if 0 had been transmitted). An erroneous bit has thus been received. At the moment we cannot decide which path to follow, so we continue analyzing the two possible cases, "c" and "d". The next two bits are 01, and here we conclude that "c" is hardly possible since "c" can only produce 00 or 11, while "d" is indeed able to produce the observed 01 (emitted bit equal to 0) to yield to state "e". Thus, the we give up following the path along "c" in order to continue along "d" and then "e". When in state "e" we receive 11 which is a possible bit state (emitted bit 1) to yield to "f" and finally 01 is a possible value of "f" which allows us to conclude that the last transmitted bit was 0. We have this been able to correct one erroneous bit and deduce that the most probable transmitted sequence was 0b00011010=0x1A which should have led to 0b0000001101011101 in which the erroneous bit is highlighted in red (Fig. 12).

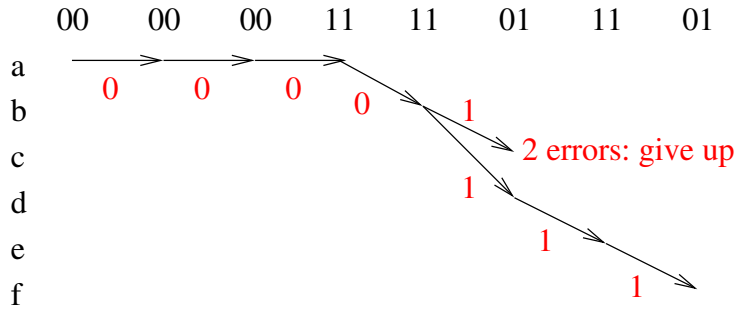


Figure 12: Top, the sequence of received bits, grouped by 2 since the convolution code generates two output bits for each encoded input bit. On the vertical axis, the states of the machine describing the encoder. In red characters, the numbers of accumulated errors along each path of the decoder. Practically, giving up on one branch would be decided when two paths meet the same state: in this case, the sequence accumulating most errors is left at the benefit of the most probable sequence since exhibiting fewer accumulated errors.

Based on this knowledge, we can now search by correlating with the received sequence the synchronization word once encoded by the convolution code, with correlation peaks demonstrating our understanding of the message encoding, and the ability to identify the beginning of each transmitted packet. When correlating the encoded word with the I/Q coefficient dataset and

converted to bits through the symbol to bit mapping shown in Fig. 13 ... the result does not show any correlation peak and is not usable (Fig. 15, top). Something is still missing in our analysis ...

5.6 Constellation rotation

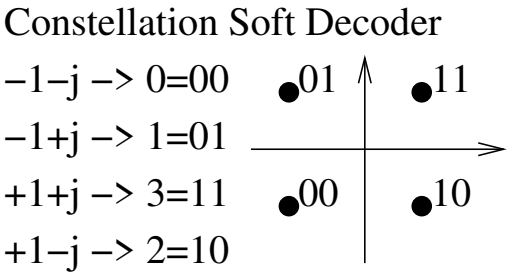


Figure 13: QPSK Constellation QPSK: each possible phase state encodes 2 bits. Assigning each symbol, from 0 to 3, to the appropriate bit pairs will be part of the decoding challenge.

The lack of correlation hints at the lack of understanding as to how bits are encoded in the acquired message, assuming that the convolutional code encoded synchronization word is correct, an assumption we will take as true considering the information provided at [2]. When we investigated binary phase modulation (BPSK – Binary Phase Shift Keying), we have seen that we had to consider two possible cases [11, 10]: either the acquired signal was in phase with the local oscillator, and the bit states obtained by comparing the received signal with the local oscillator carrier copy (Costas loop) were those expected, or the signal was in phase opposition and the bits were flipped with respect to their expected value. In both cases, the correlation with a synchronization word accumulates energy along the bit sequence and yields, by considering the absolute value, to a correlation peak, whether we have the right sequence in the collected sentence or its opposite. This simple case is more complex when considering QPSK (Quadrature Phase Shift Keying) in which the phase takes one of four possible states each encoding a pair of bits. Assigning the phase value to a pair of bits is not obvious, but most significantly any error in the symbol-bit pair mapping yields an erroneous sequence that will not correlate with the word we are looking for. As an example, let us consider the mapping stating that 0° matches the bit pair 10 and 90° to 11 (Gray code in which only a single bit varies between two adjacent phase values), then the sequence 0-90° yields 1011 while the mapping 0° to 11 and 90° to 01 will interpret the same phase sequence to 1101: the two messages generated by the same phase sequence are completely different and have no chance of accumulating energy needed to generate the correlation peak along the comparison with the reference synchronization word. We have not identified any other scheme for identifying the mapping from phase to bit pairs other than brute force by testing all possible bit combinations, as seen when reading the source code of `m.c.patts[][]` from `medet.dpr` in `meteor_decoder` :

```
11111100101000101011011000111101101100000000110110010111100101000
01010110111110111101001110010100110101010010011000001110000010
00000011010111010100100111000010010011111111001001101000011010111
10101001000001000010110001101011001001010101101100111110001111101
11111100010100010111100100111110011100000000111001101011011010000
01010110000010000001110010010111000110101010011100111101001111110
00000011101011101000011011000001100011111111000110010100100101111
10101001111101111110001101101000111001010101100011000010110000001
```

provides all possible bit combinations in the sentence encoded by convolution code, and the correlations of the received message with all these variations of the code are computed.

However, one issue remains: these bit inversions are easily implemented on the binary values of the reference word by swapping 1 and 0, but how can we perform the same operation with *soft bits* in which the phase of the received message is encoded with continuous values quantified on 8-bit values ? Shall we decide from now on which phase value to attribute (*soft* → *hard bits*), or can we keep on handling raw values ? We must interpret bit swapping operations as constellation rotation or symmetry (Fig. 14). We observe that swapping bits is interpreted as operations between the real and imaginary part, either by rotating, or by symmetry along one of the complex axis. Hence, by manipulating the real and imaginary parts of the acquired data, we can achieve the same result as the one obtained by swapping bits, but by keeping the continuous values of *soft bits* and postpone the attribution of each bit (0 or 1) to each phase during the decoding by the Viterbi algorithm.

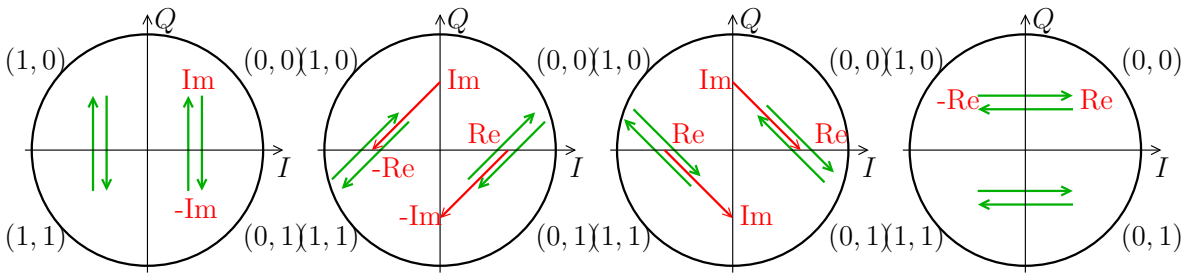


Figure 14: Rotations and symmetry of the constellation, and corresponding result on the real and imaginary axis (I and Q) of the complex plane on which the raw collected data are represented.

Identifying the mapping between the four QPSK phase conditions in the constellation diagram (I on the X axis, Q on the Y axis) and the matching bit pairs hence require a brute for attack, in which all possible combinations are tester. Correlating the phase of the signals with the various combinations of the header word following the convolutional encoder is shown on Fig. 15.

Only one mapping to a given bit pair yields a periodic sequence of correlation peaks (Fig. 15, bottom): this is the right mapping that will be used throughout the following decoding steps.

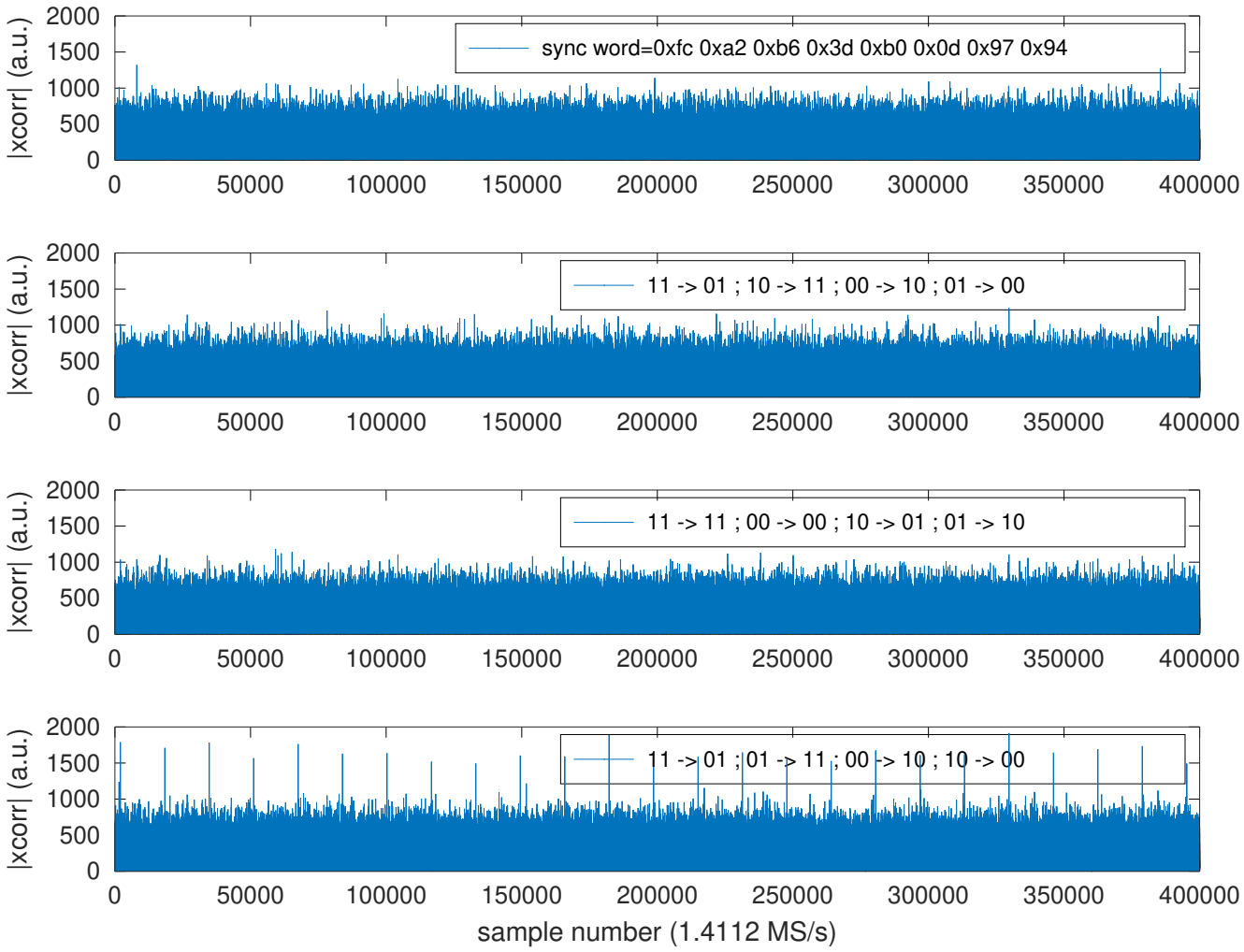


Figure 15: Correlation for the four possible cases of QPSK constellation rotation with the known header word. We observe that only the fourth case – bottom – yields period correlation peaks representative of the beginning of new sentences. This mapping between the four QPSK symbols and bit pairs is the correct one.

This permutation will from now on be applied to all I/Q sets of the acquired message since we know that doing so will yield the original bit sequence sent by the satellite during the Viterbi decoding applied to the resulting soft bits.

5.7 From bits to sentences: applying the Viterbi algorithm decoding

We now have a sequence of phases with values in the set $[0; \pi/2; \pi; 3\pi/2]$ properly organized to become a sequence a bits in which the synchronization word was found, and a unique mapping from the various symbols $\{00; 01; 11; 10\}$ to each phase value provides a solution exhibiting this correlation. We are now left with decoding to remove the convolutional code encoding, and then apply to the resulting bits (which were hence the bits encoded by the satellite prior to the convolutional code) a sequence of XOR (exclusive OR) with a polynomial designed to maximize the randomness of the resulting dataset and hence avoid long repetitions of the same bit state.

We have mentioned the availability of `libfec` efficiently implementing the decoding of convolutional code encoded signal. We extend the previous basic example to the practical case of decoding full sentences.

Our first idea was to feed the library and decode the whole file of the acquired dataset. Doing so, we hide the initialization and terminations issues of the convolutional decoding using the Viterbi algorithm. This works, since we observe after decoding that, every 1024 bytes, we recover the synchronization word 0x1ACFFC1D.

Warning: we have met a segmentation fault error when trying to allocate an array large enough to be filled with the whole dataset. Indeed, the file containing the soft bits as one byte for each sample is 11.17 MB large, so we tried to allocate a static array as would any good embedded systems developer which does not have access to dynamic memory allocation through `malloc` due to its excessive resource requirements. However, doing so attempts to allocate the array on the stack, and the default stack size on GNU/Linux is 8192 kB, as shown by `ulimit -s`: 8192. Rather than increasing the stack size, we have used the operating system's dynamic memory allocation to locate the array on the heap rather than on the stack, hence removing the constraint on the available memory space.

```
#include <stdio.h>  // from libfec/vtest27.c
#include <stdlib.h> // gcc -o demo_libfec demo_libfec.c -I./libfec ./libfec/libfec.a
#include <fcntl.h>
#include <unistd.h> // read
#include <fec.h>
#define MAXBYTES (11170164/16)  // file size /8 (bytes-> bits) /2 (Viterbi)

#define VITPOLYA 0x4F
#define VITPOLYB 0x6D
int viterbiPolynomial[2] = {VITPOLYA, VITPOLYB};

int main(int argc,char *argv[]){
  int i,framebits,fd;
  unsigned char data[MAXBYTES],*symbols;
  void *vp;

  symbols=(unsigned char*)malloc(8*2*(MAXBYTES+6)); // *8 for bytes->bits & *2 Viterbi
// root@rugged:~# ulimit -a
// stack size              (kbytes, -s) 8192
// -> static allocation (stack) of max 8 MB, after requires malloc on the heap
  fd=open("./extrait.s",O_RDONLY); read(fd,symbols,MAXBYTES*16); close(fd);

  for (i=1;i<MAXBYTES*16;i+=2) symbols[i]=-symbols[i]; // I/Q constellation rotation
  framebits = MAXBYTES*8;
  set_viterbi27_polynomial(viterbiPolynomial);
  vp=create_viterbi27(framebits);
  init_viterbi27(vp,0);
  update_viterbi27_blk(vp,&symbols[4756+8],framebits+6);
  chainback_viterbi27(vp,data,framebits,0);
  for (i=0;i<20;i++) printf("%02hhX",data[i]);
  printf("\n");
  fd=open("./sortie.bin",O_WRONLY|O_CREAT,S_IRWXU|S_IRWXG|S_IRWXO);
  write(fd,data,framebits);
  close(fd);
  exit(0);
}
```

L. Teske [2] tells us however that this approach is not optimum, since it requires that the whole file is loaded in memory at once. We know that 1024 byte blocks (2048 bytes after encoding) are individually encoded, and hence rather than decoding the whole file, we might focus on searching for the encoded header synchronization word, and decode the 2048 next bytes starting from this position. For safety and let the decoder initialize, we will make sure we fetch a few samples before and after the block to be decoded. The resulting `main` function looks like

```
fdi=open("./extrait.s",O_RDONLY);
fdo=open("./sortie.bin",O_WRONLY|O_CREAT,S_IRWXU|S_IRWXG|S_IRWXO);
read(fdi,symbols,4756+8);  // offset
framebits = MAXBYTES*8;

do {
  res=read(fdi,symbols,2*framebits+50);  // fetches a bit more data
  lseek(fdi,-50,SEEK_CUR);               // go back
  for (i=1;i<2*framebits;i+=2) symbols[i]=-symbols[i]; // I/Q constellation rotation
  set_viterbi27_polynomial(viterbiPolynomial);
  vp=create_viterbi27(framebits);
  init_viterbi27(vp,0);
  update_viterbi27_blk(vp,symbols,framebits+6);
  chainback_viterbi27(vp,data,framebits,0);
```

```
  write(fdo,data,MAXBYTES);              // decoding result, as long
} while (res==(2*framebits+50));         // ... as more data is available
close(fdi); close(fdo);
```

Additionally, this block-based processing will allow us later to add Reed Solomon block error correction (section A). Using this block error correction is optional: the data blocks obtained after convolutional code decoding using the Viterbi algorithm can be used as is, and in a first step we shall bypass the Reed Solomon block error correction, but only using the first 1024-4-128=892 bytes in each block (after removing the first 4 bytes of the synchronization word at the beginning of the block and the last 128 bytes needed for block error correction). This point will be addressed in the appendix.

Alternatively, the reader who prefers to keep on working with GNU/Octave instead of using the C language can use the program provided at `https://github.com/Filios92/Viterbi-Decoder/blob/master/viterbi.m` which is also perfectly functional with

```
f=fopen("extrait.s");   % soft bits generated from GNURadio
d=fread(f,inf,'int8');  % read file
d(2:2:end)=-d(2:2:end); % constellation rotation
phrase=(d<0)';          % soft -> hard bits
[dv,e]=viterbi([1 1 1 1 0 0 1 ; 1 0 1 1 0 1 1 ],phrase,0);
data=(dv(1:4:end)*8+dv(2:4:end)*4+dv(3:4:end)*2+dv(4:4:end));
```

Finally, the reader willing to explore Meteor M2 decoding using exclusively GNU Radio is not forgotten: the error correcting code libraries and `libfec` are implemented in `gr-satellite` at `github.com/daniestevez/gr-satellites` and described in details in the blog of D. Estévez at `destevez.net`. Thanks to his help following multiple email exchanges, we ended up decoding the synchronization word by using the Viterbi algorithm convolutional code decoding block following the signal processing scheme shown in Fig. 16. Starting with a binary file including the encoded word `fca2b63db00d9794`, generated for example with

```
echo -e -n "\xfc\xa2\xb6\x3d\xb0\x0d\x97\x94" > input.bin
```

we aim at recovering the synchronization word `1acffc1d` which would demonstrate our understanding of the decoding process. **Warning**: the aspect that stopped us was not the decoder configuration, which exactly uses the same definition as `libfec`, but the format of the input data. Indeed, GNU Radio expects, as hidden in one of the documentations of the `fec_decode_ccsds_27_fb` processing block (GNU Radio Companion `Decode CCSDS 27` block) of `gr-fec` block, to be given as input a value between -1 and +1 (and not 0 to +1 as would have been expected for a bitwise representation: we are here using *soft bits* between $\exp(j\pi) = -1$ and $\exp(j0) = +1$ and not *hard bits*). We hence process the bits read from the input file including the encoded word, multiply by 2 (hence a scale factor inverse of 0.5, value to be filled in the `Char to Float` block), subtract 1 and possibly switch bit values (multiplication by -1) to achieve the expected result and not its opposite. We check on the graphical output of Fig. 16 that reading the file yields the proper bit sequence, and by observing the output of the processed bit stream (using `xxd` for example) we obtain

```
$xxd result.bin | cut -d: -f2
 1acf fc1d 1acf fc1d 0334 53c0 1acf fc1d  .........4S.....
```

which is not perfect but close enough to our expectation. We indeed find the sequence `1acffc1d` but mixed with a few erroneous sequences `033453c0` which are included due to the wrong Viterbi decoder initialization state. Indeed, Viterbi assumes that the initialization state is all shift register values equal to 0, which is not necessarily true when the encoded word is repetitively read in a loop. D. Estévez presents at `destevez.net/2017/01/coding-for-hit-satellites-and-other-ccsds-satellites/` the various declinations of the decoder configuration to meet the configuration of the various CCSDS norm declinations.

We claim to understand how to decode the messages sent by the satellite, but are we truly sure of the validity of these bits ? In order to quickly assess a known bit sequence, we follow up on the strategy described in the beginning of this article, namely correlate with a know pattern. We are told that, without knowing anything about the sentence encoding which will be described in the next section, the protocol described at [17] in Appendix A includes a frame description of the telemetry claimed to include the date onboard the satellite: this sentence is identified with the PRS-64 code made of the sequence "2 24 167 163 146 221 254 191". Are we able to find this byte sequence in the sentences we have decoded ? Having obtained bits that we assume to be stored in an array called `dv`, we concatenate using GNU/Octave bits to nibbles and nibbles to bytes (`data` array) and to sentences (`fin` matrix below): array

```
% data is a byte array from Viterbi decoding as discussed previously
for k=1:24                              % analyze first 24 blocks
 d(:,k)=data(1+(k-1)*2048:k*2048);      % 2048 nibble sentences
 dd(:,k)=d(1:2:end,k)*16+d(2:2:end,k); % nibbles -> bytes
 fin(:,k)=dd(5:end,k);                  % remove the synchronization header from each sentence
 fin(:,k)=[bitxor(fin(1:255,k)',pn) bitxor(fin(1+255:255+255,k)',pn) ...
             bitxor(fin(1+255*2:255+255*2,k)',pn) bitxor(fin(1+255*3:255+255*3,k)',pn)];
end
```

We observe that we had to apply the `pn` code which was designed to make the bit sequence as random as possible (and hence spread the information) using the bijective function of the XOR mask. This random structure of the sentences avoids long sequences of the same bit value, which would make clock recovery difficult. The 255-byte long `pn` sequence is given at `https:`

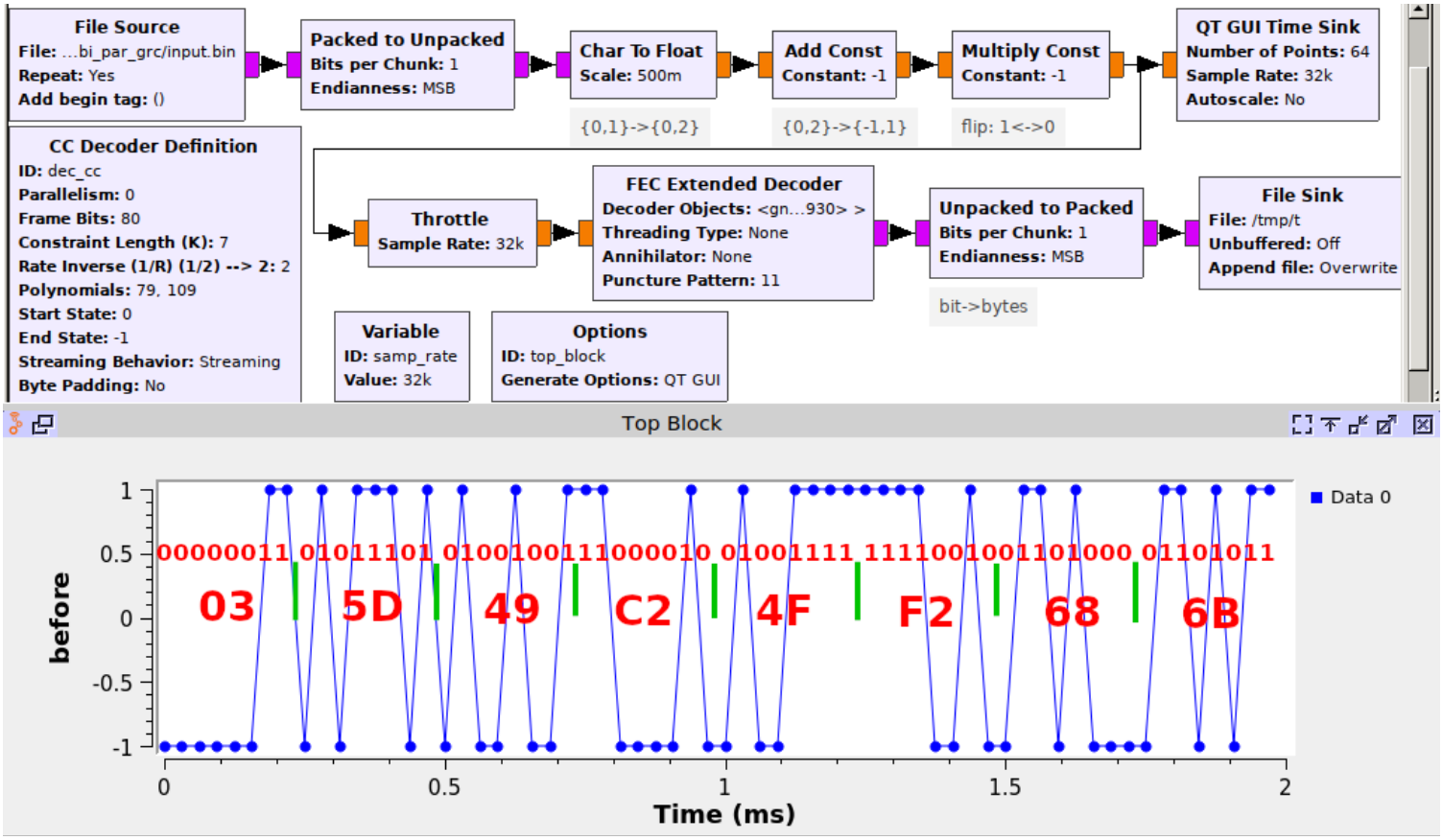


Figure 16: GNU Radio Companion signal processing chain exploiting `gr-satellite` and the generalized Viterbi algorithm convolutional code decoding block `FEC Extended Decoder` configured to meet the CCSDS standard to demonstrate synchronization word decoding. The input can be generalized to the file recorded with the soft bits representing the Meteor-M2 transmission. The input word annotated on the bottom chart is opposite of the word written in the file so that the output meets our expectation: inverting the bits is taken care of by the multiplication by -1 just before the display on the top-right of the processing chain.

`//www.teske.net.br/lucas/2016/11/goes-satellite-hunt-part-4-packet-demuxer/` and we just apply this mask to our bytes grouped by 255-element long packets (the 4-byte header, which are not affected by this transform, have already been removed).

Finally, these sentences are analyzer to search for the magic sequence PRS-64 known to prefix the telemetry sentence. Finding this sequence would demonstrate that our procedure is consistent, since not only we identify the telemetry identification sequence – a set of bytes with little change of random occurrence – but furthermore the analysis of the telemetry values provides consistent results with the acquisition time and the output of `meteor_decoder` (`Onboard time:  11:48:33.788`) as

```
date_header=final(589:589+7,9)' % found in CADU 9 (out of the 24 processed)
date=final(589+8:589+11,9)'
ans =
    11
    48
    33
   197
```

We have found the time it was onboard the satellite when the image was acquired, demonstrating the validity of the byte recovery from the I/Q stream. Now that we are confident the bit sequence is correct, we are left to analyze how the sentences are assembled to decode the image, an activity closer to computer science than signal processing.

6 From sentences to paragraphs

The bit sequences matches the norm described in the technical documentation, so we have finished the decoding investigation. Not completely ... the satellite transmits an image, and we have only recovered a data. Can we go beyond this basic result ?

This is the point where OSI layers appear. An image is a large set of information, too large to be stored in a unique packet transmitted from the satellite. Worst, the satellite sends information acquired at least on three spectral ranges (depending on whether it is on the dayside or nightside these three ranges change ... but what is day and night in Spitsbergen, with its 3 months of day and 3 months of complete darkness ?!) interleaved in various packets. We understand better why the OSI standard separates

the various abstraction layers: an image is one large entity split into color layers which are themselves split into blocks (JPEG encoding) which are themselves split into packets transmitted to the ground with all the error correction code and redundancy to maximize chances that the receiver recovers an error-free datastream. We have indeed stated JPEG compression above: for someone who was trained with all the drawbacks of lossy compression and artifacts induced by the JPEG compression, is it possible that images transmitted from satellites are thus encoded ? The tradeoff lies probably in the available bandwidth around the relatively low frequency carrier with respect to the large data size of the high resolution picture to be transmitted: we shall of course be careful, when processing such images, not to focus on artifacts related to the 8×8 pixel block encoding which will be the topic of the forthcoming discussion.

An aspect that was quite unclear to us in the various documentations lies in the definition of the *Minimum Code Unit* (MCU): we are taught that each MCU include 196 adjacent areas of a picture, made of 14 thumbnails of 8×8 pixels. The part that was not clear to us in the documentation is that *successive MCUs are independent from each other*. Hence, a picture line is made of 14 MCUs, each including 14 thumbnails of 8×8 pixels: 14×14=196 and $14 \times 14 \times 8 = 1568$ pixels is indeed the width of an image transmitted by Meteor M2. Thus, our objective is to decode the 8×8 pixel JPEG thumbnails, concatenate the resulting image, and so on until a full line of the final image is assembled for a given wavelength. The procedure is then repeated for the other two wavelengths, before starting again following a maintenance telemetry sentence (identifier 70 in the APID field, *APplication IDentifier* – the image being defined with APID identifiers in the range from 64 to 69 [17]).

Furthermore, one MCU packet might not completely fit inside the payload of the M-PDU layer protocol: it could be that one M-PDU includes multiple successive MCUs (for example when the JPEG thumbnails are small due to the lack of features in an homogeneous area being imaged) or that one MCU is shared among two M-PDU. Hence, the VCDU packet header includes a pointer to the index of the starting point of the next M-PDU. Before this pointer, we collect the left-over the the previous M-PDU.

We had stored in the `fin` matrix the successive sentences decoded after application of the Viterbi algorithm and derandomization: we display the content of the first lines for this matrix to notice the consistency of several patterns – header of the packet – before reaching the random payload. The documents that appeared clearest to understand how to decode sentences is [18].

```
octave:28> fin(1:16,:) % see 20020081350.pdf NASA p.9 of PDF
    64    64    64    64    64    64    64    64    64    64    64    64    64    64  Version
     5     5     5     5     5     5     5     5     5     5     5     5     5     5  Type
   140   140   140   140   140   140   140   140   140   140   140   140   140   140  \
   163   163   163   163   163   163   163   163   163   163   163   163   163   163  - counter
    43    44    45    46    47    48    49    50    51    52    53    54    55    56  /
     0     0     0     0     0     0     0     0     0     0     0     0     0     0  sig. field
     0     0     0     0     0     0     0     0     0     0     0     0     0     0  VCDU insert
     0     0     0     0     0     0     0     0     0     0     0     0     0     0  ... zone
     0     0     2     1     0     0     0     0     0     0     0     0     2     0  5 bits @ 0
    18   142    28    54    18   130    78   226     0    20    70    28    82    32  M_PDU header

    77   166   239   222    73    82    83   199     8    28   232   247   165   183  M_PDU...
   133   188   229    42    24    23   220    94    68   117    92   151    87   203  ... 882 bytes
    75   177   221   215     0    48    49   128    13   166     8   218   126   212
    42   138   254    87    12    32   249    87    34   172   247   107     9   142
   146   238   236    80   215    96   143   121     0   124    12    89    86   191
   179   227    64   144    89    59   240   105   105   251    46    43     0   199
                                                     ^
```

which is analyzed as follows, from the first to the last line, beginning with the 6-byte long *VCDU Primary Header*:

1. 64 matches the version constant equal to 01 followed by 6 zeros for the most significant bits of the VCDU Id (S/C id). Hence, the first 8 bits are 0100 0000,

2. the identified of the transmitting satellite follows (field Type of the VCDU Id): this field is described at [17] as being equal to 5 if the instrument is present and 63 if the instrument is absent. Here a value of 5 is a positive omen for the next steps of image decoding. Furthermore, [19, p.149] indicates that a VCDU Id of 5 (AVHRR LR) is associated with channels APID 64..69 as will be seen later.

3. the 3-byte long VCDU Counter is incremented at each new packet, as observed on the last byte (140 163 43..56) of the three byte matching the sentence counter,

4. all following fields (*signaling field*) are filled with zeros to indicate real time data transfer, as are the fields VCDU Insert zone and absence of cryptography [19, p.150],

5. finally, the last two bytes of the header provide the pointer indicating the address of the the first packet included in the current sentence. This information is arguably the most important since an M_PDU packet most certainly spans along multiple sentences, and hence knowing where the first M_PDU included in the current packets starts allows for synchronizing the beginning of the decoding process of a new image. The first 5 bits are always equal to 0 [19, p.147] while the 11 last bits provide the address, within the sentence, of the first useful packet. In this processing sequence, the pointer is computed as

   ```
   x=fin(9,:)*256+fin(10,:)+12 = 30 154 552 322 30 142 90 238 12 32 82 40 606 44
   ```

6. the 882 bytes that follow are the M-PDU payload including the *Virtual channel field*. We become convinced that the position of the header computed above is correct by

   ```
   for k=1:length(x);fin(x(k),k),end
   ```

   which returns `64 64 64 64 65 65 65 65 68 64 64 64 64 65` which is the list of the virtual channel identifiers we shall analyze later, i.e. the various wavelengths at which the images are collected (APID in the 64 to 69 range [17])

7. the ninth column is a bit unusual since it includes the first packet of the transmission of the image with APID 68, so exhibits a header offset equal to 0 with respect to the end of the VCDU header, allowing to start tackling the M_PDU payload format without having to search for the beginning address pointer. We shall thus see that 8=0000 1000 is the version (ID=000/Type Indicator=0/Secondary Header 1=present/000 APID), then APID=68 is one of the measurement channels [17] ad finally the length of the packet (in bytes) is provided by {0 105}.

7 So much text ... pictures now

We have identified how to decode the VCDU sentence, so that now we have to analyze the M_PDU payload. Multiple M_PDU can be grouped inside one VCDU sentence (for example when the JPEG thumbnail payload is highly compressed) and one M_PDU can be distributed between two successive VCDU – there is not reason for the M_PDU payload size to be a multiple of the VCDU sentence length.

We have previously identified the pointers, in the VCDU header, towards the beginning of the first M_PDU payload in the VCDU. Displaying the first bytes of each M_PDU, we observe a consistent pattern

8	8	8	8	8	8	8	8	8	8	8
68	68	68	68	68	68	68	70	64	64	64
13	13	13	13	13	13	13	205	77	13	13
34	35	36	37	38	39	40	41	42	43	44
0	0	0	0	0	0	0	0	0	0	0
105	47	49	69	81	107	57	57	97	77	79
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
2	2	2	2	2	2	2	2	2	2	2
136	136	136	136	136	136	136	136	136	136	136
181	181	181	181	181	181	181	181	186	186	186
124	124	124	124	124	124	124	124	76	76	76
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
98	112	126	140	154	168	182	2	0	14	28
0	0	0	0	0	0	0	24	0	0	0
0	0	0	0	0	0	0	167	0	0	0
255	255	255	255	255	255	255	163	255	255	255
240	240	240	240	240	240	240	146	240	240	240
77	77	77	77	77	77	77	221	77	79	81
243	186	210	178	136	175	242	154	173	238	235
197	41	160	177	253	120	216	191	166	148	77
60	194	210	146	236	9	151	11	88	100	166
240	156	80	106	84	81	201	48	42	228	208
105	41	5	65	152	245	135	33	131	208	9
254	104	41	23	193	172	56	197	38	210	28
91	52	20	205	1	233	249	0	62	115	118

which we analyze [17] by following the first column:

1. "68" is the instrument *packet identifier* onboard the satellite transmitting the information, APID. As the image collected by one instrument spans multiple successive packets, we expect to find the same APID along multiple successive columns. One interesting case is APID number 70 in column 8 which indicates a telemetry sentence, which allowed us to identify earlier the time on board the satellite when the image was grabbed.

2. "13 34" is made of the first two bits indicating whether this is the first packet of a sequence (01) or the followup of a transmission (00), followed by the packet number counter encoded on 14 bits, which will be used to check whether we have lost an image thumbnail along a line. We indeed observe that the least significant byte is incremented for each new M_PDU.

3. the next two bytes indicate the length of the M_PDU packet, here 105 bytes,

4. follows the date encoded on 64 bits, namely a date on two bytes defined as "0 0", a number of milliseconds within the day encoded on 32 bits "2 136 181 124" valid for all the packets related to a given image, and finally a date complement in microsecond encoded on 16 bits and fixed, for Meteor M2, to "0 0" [17].

5. The payload description indicates the index of the first MCU (*Minimum Code Unit*), thumbnails whose assembly will create the final image. This MCU index is incremented by 14 between two successive packets, here `98 112 126`, since thumbnails are grouped by 14 to improve compression capability [17]

6. finally, the image header includes 16 bits set to 0 [17] (*Scan Header* followed by the *Segment Header* including an indicator of the presence of the quality factor encoded on 16 bits and set to 0xFF 0xF0 or `255 240` [17] followed by the value of this quality factor which will be used in the quantization levels when decoding the JPEG thumbnail – in our case 77 but can be variable along a line of the final image.

7. The data of the 14 successive MCUs follow, generating 14 thumbnails each 8×8 pixels, or 64 bytes in the naming convention of [17] (in the case of the first sentence, this payload starts with `243 197`).

This somewhat lengthy description is needed to understand properly the link between the VCDUs and the M_PDU which finally represent two different abstraction layers of the datastream at different levels of the OSI framework. Once this difference is understood, assembling JPEG thumbnails to create an image is only a matter of rigorous implementation of the standards. With GNU/Octave, the bytes representing each MCU forming 14 thumbnails are grouped in individual files by the following piece of software

```
for col=1:23                                % column number = VCDU frame number
  first_head=fin(9,col)*256+fin(10,col) % 70 for column 11
  fin([1:first_head+1]+9,col)';       % beginning of line 11: 1st header in 70
  fin([1:22]+first_head+11,col)';     % start of MCU of line 11

  clear l secondary apid m
  l=fin(first_head+16-1,col)*256+fin(first_head+16,col); % vector of packet lengths
  secondary=fin(first_head+16-5,col);     % initializes header list
  apid=fin(first_head+16-4,col);          % initializes APID list
  m=fin([first_head+12:first_head+12+P],col);
  k=1;
  while ((sum(l)+(k)*7+first_head+12+P)<(1020-128))
    m=[m fin([first_head+12:first_head+12+P]+sum(l)+(k)*7,col)];
    secondary(k+1)=fin(first_head+16+sum(l)+(k)*7-5,col);
    apid(k+1)=fin(first_head+16+sum(l)+(k)*7-4,col);
    l(k+1)=fin(first_head+16-1+sum(l)+(k)*7,col)*256+fin(first_head+16+sum(l)+(k)*7,col);
          % 16=offset from VDU beginning
    k=k+1;
  end
  for k=1:length(l)-1  % saves each MCU bytes in a new file
    jpeg=fin([1:l(k)]+first_head+12+19+sum(l(1:k-1))-1+7*(k-1),col);
    f=fopen(['jpeg',num2str(apid(k),'%03d'),'_',num2str(col,'%03d'),'_',num2str(k,'%03d'),'.bin'],'w');
    fwrite(f,jpeg,'uint8');
    fclose(f);
  end

  k=length(l);                                      % last incomplete packet
  jpeg=fin([1+first_head+12+19+sum(l(1:k-1))-1+7*(k-1):end],col);
  first_head=final(9,colonne+1)*256+final(10,col+1); % looks for next VCDU
  jpeg=[jpeg ; final([1:first_head]+10,col+1)];
  % we expected 79 bytes in the last packet: 925-892=33 are missing
  f=fopen(['jpeg',num2str(apid(k),'%03d'),'_',num2str(col,'%03d'),'_',num2str(k,'%03d'),'.bin'],'w');
  fwrite(f,jpeg,'uint8');
  fclose(f);
end
```

Having saved the content of the MCUs in individual files named `jpeg*.bin`, we have given on on the task of re-implementing Huffman decoding, RLE and the discrete cosine transform needed to convert each JPEG compressed thumbnail into a pixel matrix ready to be displayed. Huffman decoding is especially annoying to implement since it handles data as bit packets whose size is not necessarily multiple of 8 but depends on the size of each information in the encoding binary tree. Have have just read and understood the source code of the decoder from `meteor_decoder`, translated to C++ at `github.com/infostellarinc/starcoder/blob/master/gr-starcoder/lib/meteor`, and used the small part of the library to validate the content of the bit packets obtained in the preceding processing steps and that we claim to contain JPEG thumbnails. This time, the encoding scheme (and hence decoding) is very well described in [20]. The fact that `meteor_decoder` considers the MCUs we feed it as valid and that the resulting thumbnails are consistent since they can be assembled correctly as pictures prove that our decoding of VCDUs and then MCUs is correct.

```
#include "meteor_image.h"
using namespace gr::starcoder::meteor;
```

```
3
4  int main(int argc,char **argv)
5  {int fd,len,k,quality=77;    // fixed quality ...
6   unsigned char packet[1100]; // will be provided as argument later
7   imager img=imager();
8   if (argc>1) quality=atoi(argv[1]);
9   fd=open("jpeg.bin",O_RDONLY);
10  len=read(fd, packet, 1100);
11  close(fd);
12  img.dec_mcus(packet, len, 65,0,0,quality);
13 }
```

is linked with `meteor_image.cc` and `meteor_bit_io.cc` taken from the github archive cited above. Exploiting this program by feeding with binary files including the MCU payload, we are provided as output with a 14×64 element matrix which we shall name `imag`, each 64-byte long line being itself a 8×8 pixel thumbnail. Reorganizing in GNU/Octave these 64 elements using

```
m=[];for k=1:size(imag)(1) a=reshape(imag(k,:),8,8); m=[m  a'];end
```

we obtain a 112×8 pixel matrix displayed using `imagesc(m)` in order to visualize the image as shown in Fig. 17 (left). this procedure is repeated for the 14 MCUs each final image line is made of: Fig. 17 illustrates the concatenation of the first set of thumbnails (left) with the second series, demonstrating how continuous the patterns are. These processing steps are repeated for a whole line of the image acquired at one wavelength by a given instrument (and hence a given APID) before a new APID follows and prompts the processing to restart and so on to create in parallel multiple images acquired at various wavelengths. Notice the excellent compression ratio brought by JPEG on these homogeneous and feature-free areas: only 60 bytes are needed to encode these $14 \times 64 = 896$ pixel images. Areas exhibiting more features still require bigger MCUs with a few hundred bytes and up to 700 byte large.

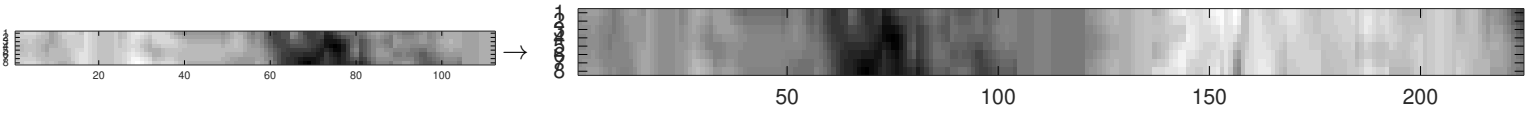


Figure 17: Decoding one MCU (left) made of 14 successive thumbnails each 8×8 pixel large, and concatenation with the next MCU (right) to create a picture $28 \times 8 = 224$ pixel wide and 8 pixel high. The complete final image is thus assembled with small MCU parts. This example is demonstrated on APID 68.

The result of assembling uncompressed JPEG thumbnails to generate 8×8 pixel matrices is shown in Fig. 18 for the instrument with APID 68. We start seeing some consistent feature of an image, but clearly a few thumbnails are missing at the end of each line since some packets were corrupted and could not be decoded (Fig. 18).

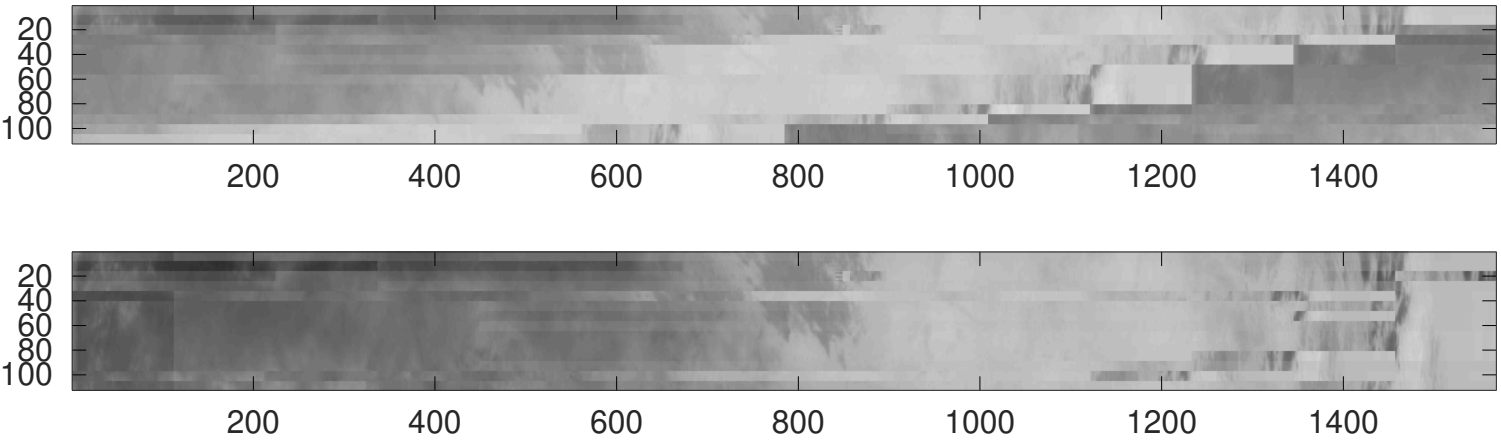


Figure 18: Top: result of decoding JPEG thumbnails and assembling into a complete picture without consider the counter. We clearly observe that the pattern is shifting from one line to another as packets are missing, resulting in a poor image hardly usable. Bottom: if the packet counter does not reach the expected threshold of 14 thumbnails/MCU, then some dummy packets are inserted to compensate for the missing thumbnails: this time the image is properly aligned. Here the APID is 68.

We compensate for the missing thumbnails, at least temporarily, by duplicating each thumbnail for missing information as indicated by the MCU counter: rather than recover the missing information, we can at least align along the vertical axis of the picture the adjacent thumbnails and hence achieve a usable picture (Fig. 19). In this example, we have not used the quality information which modified the quantization coefficients during the JPEG compression depending on the content of the picture, and some discontinuity in the greyscale pattern remain visible.

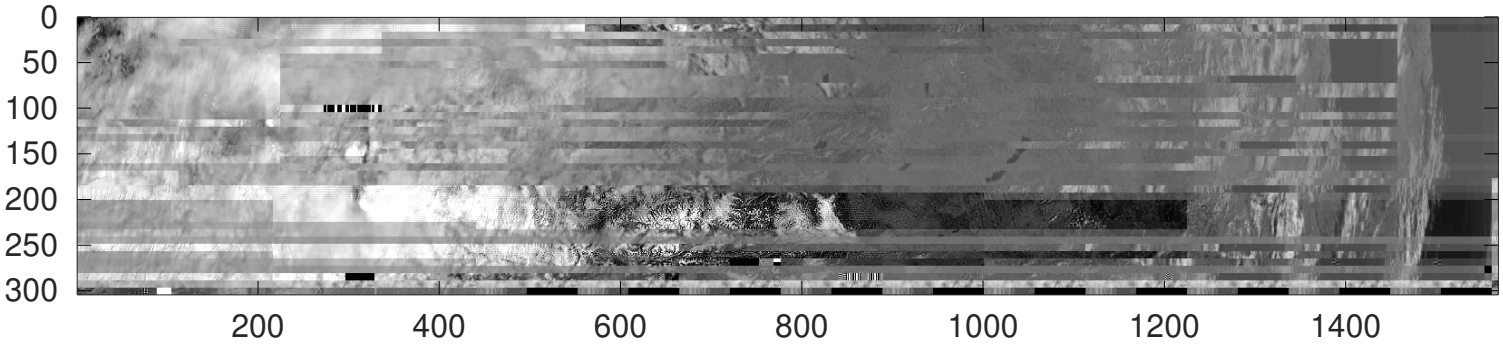


Figure 19: Result of decoding APID 65 while using the counter to identify missing thumbnails, but without exploiting the quality information provided in the JPEG header. The main geographical features are visible, but strong contrast variations exist within the picture, making the result poorly suited.

By integrating the quality factor as an argument provided to the thumbnails decoder of `meteor_decoder` on which our program is linked, the greyscale values become homogeneous to yield a convincing result (Fig. 20) closely resembling the reference picture fully decoded by `meteor_decoder` (Fig 21). Notice that the satellite pass was far from optimal for a listening station located in France since we can clearly see the Istrie region, Austrian Alps as well as the Balaton lake in Hungary (long dark structure around abscissa 1050 in the middle of the picture), hinting at a pass close to the Eastern horizon.

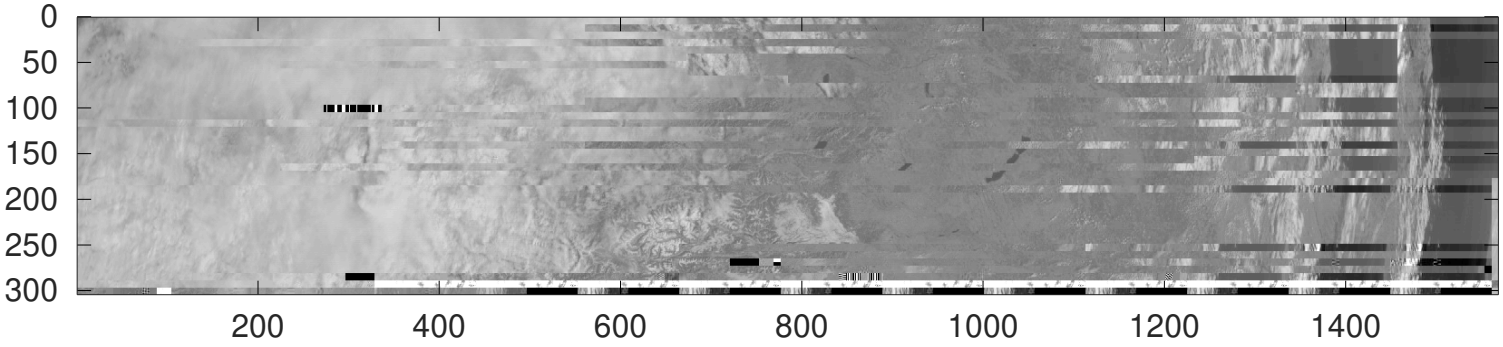


Figure 20: Result of decoding APID 65 while exploiting the counter to identify missing thumbnails, and the JPEG quality information. The individual thumbnails become hardly visible and the greyscale evolve continuously along the picture.

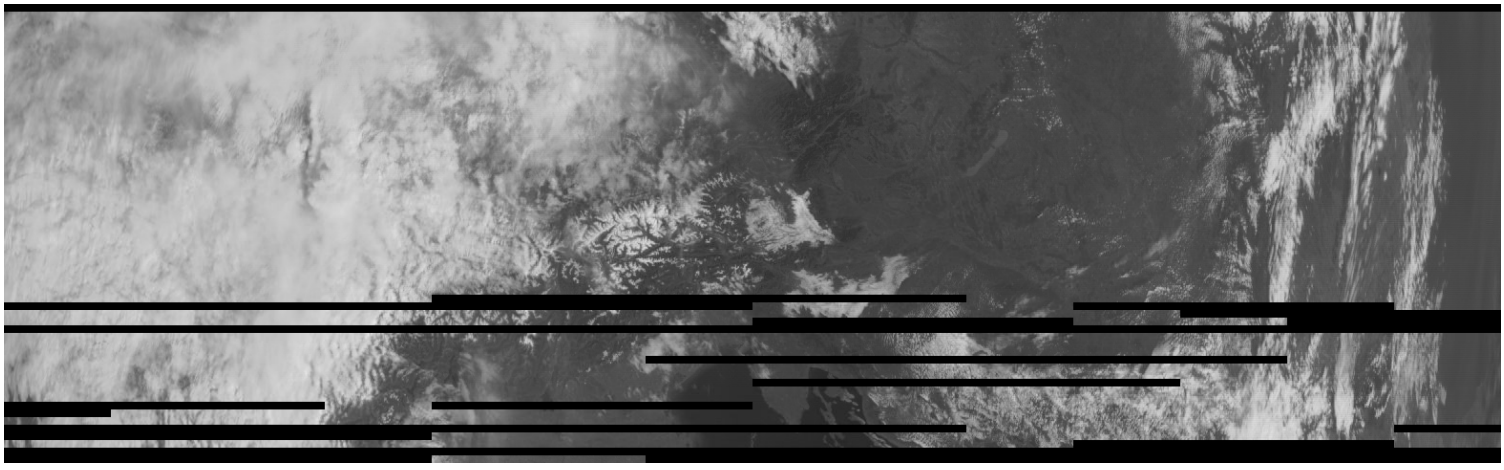


Figure 21: Result of decoding APID 65 with `medet` used as reference picture for comparison with figure 20.

8 Conclusion

This exploration of the LRPT protocol defining digital communications between weather satellites and ground was the opportunity to address all the OSI layers, from the physical layer (transmission frequency) to the coding (QPSK and convolutional encoding) to packets (words) and images (sentences) broadcast in the messages. This presentation was aimed at demonstrating how a useful

tool software defined radio could be for teaching: each processing step involved a new physical or mathematical principle, each one incredibly boring if tackled independently from a purely theoretical perspective, but becoming fascinating in the complex framework of the data transmission concluded with decoding an image.

Hence, we have practically discovered some of the subtleties of QPSK modulation and the various solutions when assigning to each of the 4 possible phase state a bit pair – problem that had escaped us when investigating a BPSK modulated signal – and then implementing convolutional code decoding using the Viterbi algorithm. Having demonstrated the consistency of the bit sequence generated by the convolutional code decoding, we have temporarily skipped the Reed Solomon block code error correction to unstack the various protocol layers encapsulating the thumbnails which the final image is made of. The motivated reader might want to implement manually JPEG decoding which we simply implemented here without reviewing the theoretical background. Finally, the Reed Solomon block error decoding algorithm is implemented and its proper operation validated, despite a rather minor benefit in this particular demonstration.

These fundamental principles have been described to provide the basics to decode many other space-borne remote sensing data streams, as demonstrated by the rich list of applications of the blog written by D. Estévez who applies his expertise on a multitude of amateur and professional satellite links at `destevez.net/`, and most significantly for example `destevez.net/2017/01/ks-1q-decoded/`.

References

[1] *Meteor-M 2 satellite*, at `planet.iitp.ru/english/spacecraft/meteor-m-n2_eng.htm`

[2] L. Teske, *GOES Satellite Hunt* at `www.teske.net.br/lucas/satcom-projects/satellite-projects/`

[3] G.D. Forney Jr, *The Viterbi Algorithm: A Personal History*, at `https://arxiv.org/abs/cs/0504020v2` (2005)

[4] D. Bodor, *Réception de vos premières images satellite*, Hackable **25** (Juillet 2018) [in French]

[5] `www.nec2.org`

[6] J.-M Friedt, *Satellite image eavesdropping: a multidisciplinary science education project*, European J. of Physics, **26** 969–984 (2005)

[7] D. Israel, *A Space Mobile Network*, WiSEE conference (Dec. 2018), or `https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20170009966.pdf`

[8] G. Kranz, *Failure is not an option – Mission Control From Mercury to Apollo 13 and Beyond*, Simon & Schuster (2000)

[9] A. Makovsky, A. Barbieri & R. Tung, *Odyssey Telecommunications*, DESCANSO Design and Performance Summary Series **6** (2002) at `descanso.jpl.nasa.gov/DPSummary/odyssey_telecom.pdf`: p.34 tells us that "The command format currently used for deep-space missions, including Odyssey, is defined in the CCSDS standard CCSDS 201.0-B-1."

[10] J.-M Friedt, *Radio Data System (RDS) – analyse du canal numérique transmis par les stations radio FM commerciales, introduction aux codes correcteurs d'erreur*, GNU/Linux Magazine France **204** (Mai 2017) [in French]

[11] J.-M Friedt, G. Cabodevila, *Exploitation de signaux des satellites GPS reçus par récepteur de télévision numérique terrestre DVB-T*, OpenSilicium **15**, Juillet-Sept. 2015 [in French]

[12] S. Lin & D.J. Costello, *Error Control Coding: Fundamentals and Applications*, Prentice Hall (1983)

[13] A. Viterbi, *Error bounds for convolutional codes and an asymptotically optimum decoding algorithm*, IEEE Trans. on Information Theory **13** (2), pp.260–269 (1967)

[14] E. Neri *& al.*, *Single space segment – HRPT/LRPT Direct broadcast services specification*, doc. MO-DS-ESA-SY-0048 rev. 8, ESA EUMETSAT EPS/METOP (1 Nov. 2000) at `mdkenny.customer.netspace.net.au/METOP_HRPT_LRPT.pdf`

[15] G.D. Forney, *The viterbi algorithm*, Proc. IEEE **61** (3), pp.268–278 (1973)

[16] R. Sniffin, *Telemetry Data Decoding*, Deep Space Network **208**, p.12 (2013) at https://deepspace.jpl.nasa.gov/dsndocs/810-005/208/208B.pdf, or `http://www.ka9q.net/amsat/ao40/2002paper/` illusting the many possible declinations from the same formal definition of the correcting code

[17] *Structure of "Meteor-M 2" satellite data transmitted through VHF-band in direct broadcast mode*, at `planet.iitp.ru/english/spacecraft/meteor_m_n2_structure_2_eng.htm`

[18] W. Fong *& al.*, *Low Resolution Picture Transmission (LRPT) Demonstration System – Phase II Report, Version 1.0* (2002), at `https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20020081350.pdf`

[19] P. Ghivasky *& al.*, *MetOp Space to Ground Interface Specification*, doc. MO-IF-MMT-SY0001 rev. 07C, EADS/ASTRIUM/METOP (29 Mars 2004) at `http://web.archive.org/web/20160616220044/http://www.meteor.robonuka.ru/wp-content/uploads/2014/08/pdf_ten_eps-metop-sp2gr.pdf`. The web site `www.meteor.robonuka.ru` was a source of inspiration throughout this investigation but has unfortunately disappeared: only `archive.org` keeps trace of the documents no longer available elsewhere. A similar information is found at a A. Le Ber, *Metop HRPT/L-RPT User Station Design Specification*, EUMETSAT Polar System Core Ground Segment, document EPS-ASPI-DS-0674 (05/03/03) at `www.eumetsat.int/website/wcm/idc/idcplg?IdcService=GET_FILE&dDocName=PDF_ASPI_0674_EPS_CGS-US-SP&RevisionSelectionMethod=LatestReleased`

[20] *Information technology – Digital compression and coding of continuous-tone still images – requirements and guidelines*, ITU CCITT recommandation T.81 (1993) at `www.w3.org/Graphics/JPEG/itu-t81.pdf`

A Reed Solomon block error correcting code

Convolutional coding is designed to compensate for noise distributed along the bits due to radiofrequency communication noise between the emitted and the receiver, assuming a uniform random noise that might impact each bit independently of its neighbors. Such a coding scheme would be however unable to correct for a corrupted block of data due to a burst of noise: this type of error is taken care of by the block error correcting code, as implemented for example by Reed Solomon. This code is similar to the block correcting strategy we have already discussed when decoding RDS with the BCH encoding [10]. Here, each 255 byte long packet is made of 223 payload bytes and 32 error correcting code bytes, allowing to identify transmission errors et possible correct part of them. This type of error correcting code is thus named RS(255,223) since for 255 transmitted bytes, 223 are data and the last 32 are error correcting code bytes. `libfec` provides the library needed to correct transmission errors using RS(255,223), as described in [2]. Since the aim of block error correcting code is to correct for a set of erroneous bits, it is wise to distribute information along the sentence in order to minimize the impact of an interference affecting multiple adjacent bits. Hence, instead of splitting a 1020 byte sentence in 4 neighboring sentences each 255 byte long, the data structure interleaves 4 sequences of data and their error correcting code as illustrated in Fig. 22, with the error correcting code located at the end of the sentences.

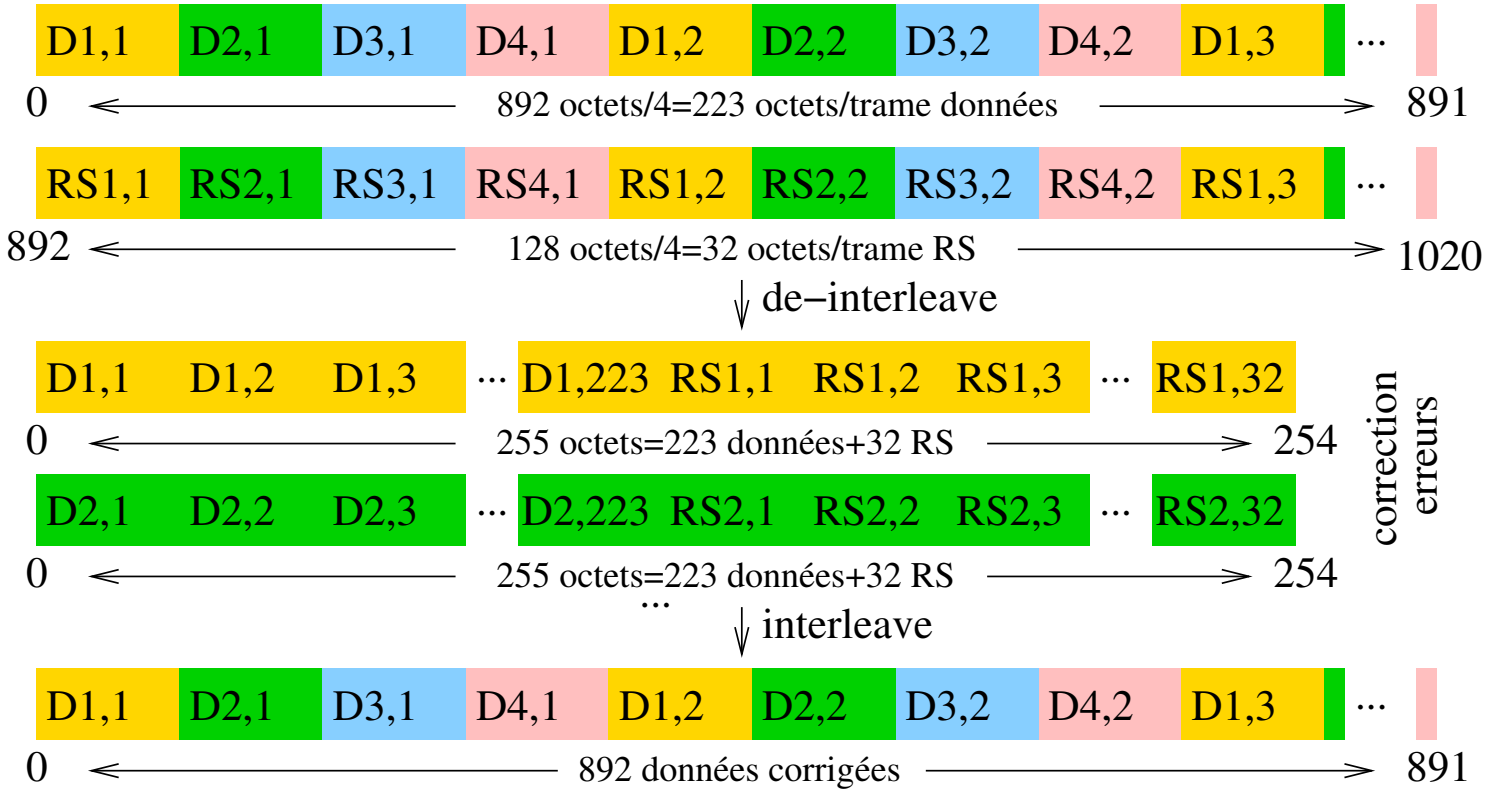


Figure 22: Organization of data along a 1020 byte long CVCDU sentence (we have already removed the 4-byte long synchronization word header). The first 892 bytes include the data payload `D` considered as interleaved for the 4 sets of 223 byte long sequences which can be corrected using Reed Solomon RS(255,232), and the last 128 bytes include, still interleaved, the 4 sequences of 32 byte long error correcting code `RS`. Applying the correction algorithm requires first de-interleaving the data (top → middle), apply Reed Solomon to identify and correct errors (middle), and re-interleave the data (middle→bottom) to put them back in their original position, but after correcting some of the bytes possibly corrupted during the radiofrequency link.

We can train first to understand how Reed Solomon is implemented in `libfec`:

```c
#include <fec.h>    // gcc -o jmf_rs jmf_rs.c -I./libfec ./libfec/libfec.a

int main()
{int j;
 uint8_t rsBuffer[255];

 uint8_t tmppar[32];
 uint8_t tmpdat[223];

 for (j=0;j<255; j++) rsBuffer[j]=(rand()&0xff);  // received data
 for (j=0;j<223;j++)  tmpdat[j]=rsBuffer[j];      // backup data
 encode_rs_ccsds(tmpdat,tmppar,0);                // create RS code
 for (j=223;j<255;j++) rsBuffer[j]=tmppar[j-223]; // append RS after data
 rsBuffer[42]=42; tmpdat[42]=42;                  // introduce errors
 rsBuffer[43]=42; tmpdat[43]=42;                  // ... on purpose
 rsBuffer[44]=42; tmpdat[44]=42;    // ... to check correction capability
 rsBuffer[240]=42;tmppar[240-223]=42;
 printf("RS:%d\n",decode_rs_ccsds(rsBuffer, NULL, 0, 0)); // check that RS can correct
 for (j=0;j<223;j++)
   if (rsBuffer[j]!=tmpdat[j]) {printf("%d:␣%hhd␣->␣%hhd␣;␣",j,tmpdat[j],rsBuffer[j]);}
 for (j=223;j<255;j++)
   if (rsBuffer[j]!=tmppar[j-223]) {printf("%d:␣%hhd␣->␣%hhd␣;␣",j,tmppar[j-223],rsBuffer[j]);}
}
```

in this example we create a (random) dataset which is encoded, then modify 4 bytes of the payload and 1 byte of the error correcting code, and test how the decoder can correct these errors. The result

```
RS:4
42: 42 -> 5 ; 43: 42 -> 23 ; 44: 42 -> 88 ; 240: 42 -> 95
```

is consistent with our expectations: 4 errors were identified and corrected.

Applying this sample program to the 128 bytes at the end of a sentence designed to correct the first 892 bytes at the beginning of the sentence ... does not work at all ! Yet another trick indicated by Lucas Teske that we had not found in the documentation: the implemented algorithm is a *dual basis Reed Solomon* in which the bytes are once again run through a randomization table as described at `github.com/opensatelliteproject/libsathelper/blob/master/src/reedsolomon.cpp`. Once this transform has been applied, the error correcting code operates properly, as demonstrated with the piece of software below which includes the whole decoding sequence, namely Viterbi algorithm deconvolution, application of the polynomial bijective XOR operation to remove the randomization of the data, de-interleaving the data to be grouped with their Reed Solomon error correcting code, applying the transposition polynomial, error correction, removing the transposition polynomial to finally recover the corrected data:

```c
#include <fec.h> // gcc -o demo_rs demo_rs.c -I./libfec ./libfec/libfec.a

// github.com/opensatelliteproject/libsathelper/blob/master/src/reedsolomon.cpp
// dual basis Reed Solomon !
#include "dual_basis.h"

unsigned char pn[255] ={           // randomization polynomial
    0xff, 0x48, 0x0e, 0xc0, 0x9a, 0x0d, 0x70, 0xbc, \
    0x8e, 0x2c, 0x93, 0xad, 0xa7, 0xb7, 0x46, 0xce, \
[...]
    0x08, 0x78, 0xc4, 0x4a, 0x66, 0xf5, 0x58  };

#define MAXBYTES (1024)

#define VITPOLYA 0x4F
#define VITPOLYB 0x6D

#define RSBLOCKS 4

#define PARITY_OFFSET 892

void interleaveRS(uint8_t *idata, uint8_t *outbuff, uint8_t pos, uint8_t I) {
  for (int i=0; i<223; i++) outbuff[i*I+pos]=idata[i];
}

int viterbiPolynomial[2] = {VITPOLYA, VITPOLYB};

```

```c
int main(int argc,char *argv[]){
 int res,i,j,framebits,fdi,fdo;
 unsigned char data[MAXBYTES],symbols[8*2*(MAXBYTES+6)]; // *8 for bytes->bits & *2 Viterbi
 void *vp;
 int derrors[4] = { 0, 0, 0, 0 };
 uint8_t rsBuffer[255],*tmp;
 uint8_t rsCorData[1020];

 fdi=open("./extrait.s",O_RDONLY);
 fdo=open("./sortie.bin",O_WRONLY|O_CREAT,S_IRWXU|S_IRWXG|S_IRWXO);
 read(fdi,symbols,4756+8);  // offset
 framebits = MAXBYTES*8;

 do {
  res=read(fdi,symbols,framebits*2+50);  // 50 additional bytes to finish viterbi decoding
  lseek(fdi,-50,SEEK_CUR);               // go back 50 bytes
  for (i=1;i<2*framebits;i+=2) symbols[i]=-symbols[i]; // I/Q constellation rotation
  set_viterbi27_polynomial(viterbiPolynomial);
  vp=create_viterbi27(framebits);        // convolution -> Viterbi
  init_viterbi27(vp,0);
  update_viterbi27_blk(vp,symbols,framebits+6);
  chainback_viterbi27(vp,data,framebits,0);
  tmp=&data[4];                          // rm synchronization header
  for (i=0;i<1020; i++) tmp[i]^=pn[i%255];  // XOR decode (dual basis)

  for (i=0; i<RSBLOCKS; i++)
      { for (j=0;j<255; j++) rsBuffer[j]=tmp[j*4+i];        // deinterleave
        for (j=0;j<255; j++) rsBuffer[j]=ToDualBasis[rsBuffer[j]];
        derrors[i] = decode_rs_ccsds(rsBuffer, NULL, 0, 0); // decode RS
        for (j=0;j<255; j++) rsBuffer[j]=FromDualBasis[rsBuffer[j]];
        interleaveRS(rsBuffer, rsCorData, i, RSBLOCKS); // interleave
        printf(":%d",derrors[i]);
      }
  write(fdo,data,4);                      // header
  write(fdo,rsCorData,MAXBYTES-4);  // corrected frame
 } while (res==(2*framebits+50));
 close(fdi);
 close(fdo);
 exit(0);
}
```

This code is the culmination of the whole phase conversion process from the I/Q coefficients to bits ready to be assembled to recover sentences and the JPEG images, including the two convolutional and block error correcting codes. The result of this additional correction is illustrated in Fig. 23 and demonstrates how adding the block error correcting code allows for extending the range of the JPEG image analysis as the satellite reaches close to the horizon.

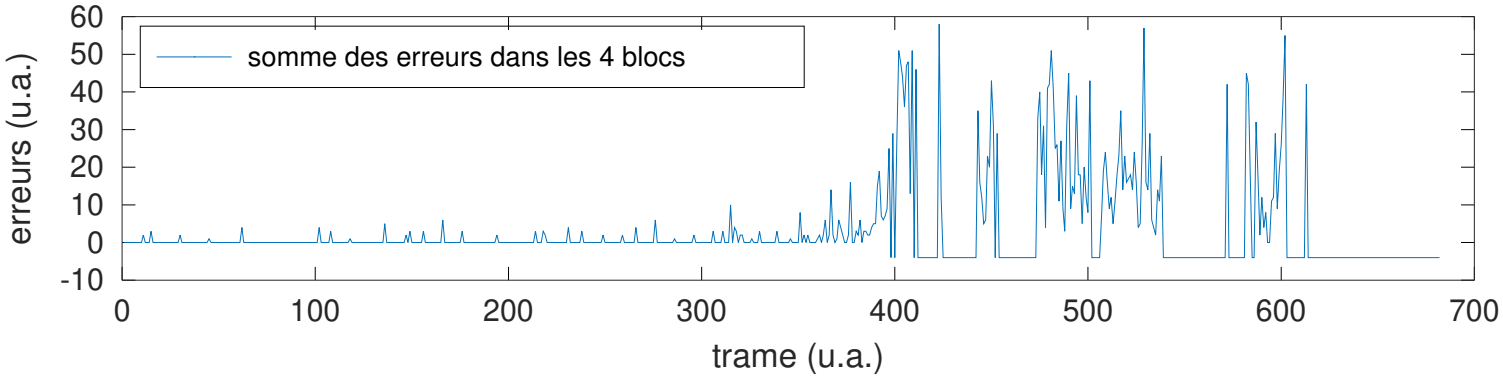


Figure 23: Result of the error correction along the satellite pass. We clearly observe that block correction becomes more efficient as the satellite gets lower on the horizon, degrading the link budget. -1 indicates that too many errors corrupted the payload during the reception to allow for correcting the erroneous bits.

Notice that some of the codes have been truncated of the standard input/output library header files to make the sample listings more compact: the reader will not be challenged in recovering the missing header files (`stdio.h`, `stdlib.h`, `unistd.h` ...) allowing to open, read, write and close files as well as communicate with the console.