University of Porto

Master's Thesis

SAR Algorithms for Embedded Platforms

Author:
Gonçalo Santos

Supervisor:
Dr. Sérgio Reis Cunha

Department of Electrical Engineering
Faculty of Engineering of the University of Porto

28th June 2021

Abstract

Due to the recent commoditization of COTS radio components, more specifically, of software-defined radio (SDR) platforms, having optimized implementations of synthetic-aperture radar (SAR) capture and processing software is now more needed than ever.

The SARIA experiment is one such example where having optimized software is critical. With the clock ticking until the end of the mission, losing precious time on slow processing software is undesirable.

In this dissertation, we present an end-to-end SAR capture and processing system that is made with performance in mind. Being designed and implemented from the ground-up, every step - from pulse transmission to image output - was scrutinized to ensure it has the fastest possible performance.

We also present a novel time-domain multilook algorithm that is feasible to use even in embedded systems. Using this algorithm, we are able to perform look-local motion compensation in order to improve the image quality of long SAR sessions.

In general, the results show that this system is able to generate high-quality images, while doing so in a fraction of the time.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Sérgio Cunha, for all his never-ending support, patience and challenging ideas.

I also want to thank my girlfriend, Tânia, and my family in general for always having had my back and making me laugh when I need it the most.

I would then like to thank the SARIA team as, without them, this thesis wouldn't have existed.

Finally, I would like to thank Zaida Silva, Bruno Correira, and Manuel Azevedo for all their help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Radar is an electromagnetic remote sensing system for the detection of reflecting targets [1]. A radar system is composed by a transmitter, a receiver, an antenna (which is usually shared between the transmitter and the receiver), and the processing unit.

Like many other technological advances, it was originally developed for military purposes during World War II, in order to track ships and aircrafts during day or night, through clouds and even rain. The first radar systems were very simple, only detecting how far away targets were, by measuring the time delay between the transmission of an electromagnetic pulse and the reception of its echo, and in which direction they were, by using an antenna with high directivity.

By transmitting pulses of very short duration, high resolution in the range direction (direction where the antenna is pointing) can be easily achieved. However, traditional radar cannot distinguish between two side-by-side targets that are illuminated by the antenna beam at the same time. Furthermore, the radar beam diverges with increasing range, so the resolution in this direction (called azimuth) becomes coarser with range.

In 1951, Carl Wiley discovered that if we move the radar platform linearly while transmitting pulses in an oblique direction, the received echoes will become Doppler shifted. By using this information and processing the echoes coherently, he found that very fine azimuth resolution could be achieved, and that the resolution would be range-independent. With this technique, called synthetic-aperture radar (SAR), high-resolution 2D images of the target can be created [2, 3].

Synthetic-aperture radar data was initially processed using the principles of Fourier optics [4, 5], where lenses provided real-time 2D Fourier transforms, and diffraction gratings were used to focus the data. This method had the advantage of providing real-time SAR processing, but the equipment had to be precisely aligned and calibrated, which made obtaining high-quality images difficult.

The first satellite with a digital SAR processor was NASA's 1978 oceanographic SEASAT

program. Even though the employed digital processing was much slower than the optical counterparts (taking as long as 40 hours to process an image in its on-board computer), the gains in quality were significant. In general, the use of digital signal processing techniques has allowed us to innovate and improve radar systems year after year.

Since the SEASAT mission, synthetic-aperture radar has become a very active field of development and vastly popular among the remote sensing community, with a broad range of applications, both civil and military. Besides the already mentioned oceanographic use, other SAR applications include topography, glaciology, geology and forestry.

Initially, radar systems (and radio systems, in general) had to be built using specific components that were hard and expensive to obtain or design. However, the wireless revolution [6], which started in the 1990s with the advent of digital wireless networks, has filled the market with highly efficient, small, and accessible commercial off-the-shelf (COTS) radio components. Among those are the software-defined radio (SDR) platforms, now widely avaible, which are becoming the state of the art for complex signal transmission and reception. These SDR platforms, combined with other COTS components and with the increase of computing power in general, have enabled the creation of synthetic-aperture radar systems even at the hobbyist level [7].

1.2 Motivation

With the progressive mainstreaming of SDR platforms, the usage of embedded computing devices for performing the required SAR processing to produce imagery from the transmitted signal is sought. However, the complexity of such algorithms makes this a challenging task.

The main motivation of this dissertation is the Synthetic-Aperture Radar using an Inflatable Antenna (SARIA) experiment, which was selected, in 2019, to fly as part of cycle 13 of the Balloon Experiments for University Students (BEXUS) program. This program enables university students to fly scientific and technological payloads on board of a stratospheric balloon, which happens to be a great platform for performing SAR, due to its (relatively) slow velocity and short-term stable trajectory. The primary objective of SARIA is to obtain synthetic-aperture radar images of the ground under the flight path using an inflatable antenna, and using the license-exempt 5.8 GHz Industrial, Scientific, and Medical (ISM) band, within all associated power restricions.

We could choose to only perform data acquisition during the launch campaign, and wait for the recovery of the gondola to then extract the raw data, process it, and produce the final images. However, this would introduce a very significant data loss risk, as the gondola could crash while landing and destroy the storage drives. Also, we can't just transmit the raw SAR data to the ground station, because the unprocessed data is very large and generally incompressible with generic file compressors, and the available telemetry link has very strict bit rate limits.

Therefore, in order to mitigate the risk of complete data loss, we'll perform the processing

of at least some SAR sessions directly on the on-board computer of the experiment and transmit only the final image (which is several orders of magnitude smaller than the raw data) to the ground station. This way, even *if* the source data is lost, we'll at least have some results that may be used to characterize and obtain conclusions about the system quality.

Another advantage of obtaining SAR images in real-time, is that we can inspect the resulting images during the launch campaign and infer if there are any problems with, for example, the inflatable antenna or the radio frequency (RF) front end. Without this feedback loop, these problems could go unnoticed for the entire mission, so it further mitigates the risk of having a failed mission.

The main challenge with this approach is that the on-board computer has externally imposed power consumption limits and relatively low computing-power (considering modern desktop computer standards). This means that the implementation of the processing algorithms will have to be significantly optimized, because we have real-time requirements relative to the processing time:

- First of all, we want to obtain the images in an useful time frame, in order to be able to download them from the low bit rate telemetry link while the launch campaign is still ongoing.
- Due to the high CPU and memory usage of the image processing algorithms, we won't be able to process previous sessions *while* performing the acquisition of the following session. Note that the typical floating time (during which we are able to acquire data) is *only* around 1h30min, so we have a trade-off between the amount of SAR sessions captured and the amount of time spent processing images. The faster we are able to process each single image, the more sessions we are able to acquire, for the same number of processed images.

In general, obtaining highly-optimized SAR processing software is something that is greatly desired by the remote sensing community, because it allows them to reduce processing and power requirements due to faster and more efficient algorithms and better utilization of the available hardware.

Another large challenge of this approach is that we are only able to see the end result of the entire signal processing chain, as opposed to being able to interactively stop the processing in some steps and inspect intermediate results. Therefore, the system must be able to obtain data from several data sources in a synchronized manner (eg. navigation data matched with the received samples) and also apply some automated heuristics in order to minimize time wasted on reprocessing images several times while tuning processing parameters by hand.

1.3 Objectives

Following the motivation described in the previous section, the main objective of this work is to design and implement an end-to-end SAR acquisition and processing system, capable of generating images within the real-time requirements of the SARIA experiment. This includes the:

- Study and definition of suitable radar pulses to be transmitted.
- Implementation of the interface with the SDR, to transmit pulses and receive the respective echoes in full-duplex.
- Integration with a navigation system for motion compensation and geo-tagging.
- Integration with a global time reference for synchronization.
- Study and implementation of suitable SAR algorithms.
- Optimization of the data flow and of the processing software for performance.

The SARIA mission will only have a single launch campaign, so it only has one chance to be successful. Therefore, this system must be extensively tested to ensure it is robust to errors and that it works in a variety of situations.

1.4 Organization

This dissertation has a very straightforward organization:

Chapter 2 covers the synthetic-aperture radar fundamentals on which the rest of the thesis is built. Everything from the problem geometry and pulse compression, to motion compensation and multilook is reviewed, and some problems with the current techniques are brought up. At the end of the chapter, a brief overview of general optimization techniques is presented.

Chapter 3 presents a thorough exposition of SARSYS, the end-to-end SAR capture and processing system designed and implemented from the ground-up to tackle the problem at hand. A more efficient time-domain multilook algorithm is discussed, along with its potential advantages for improving motion compensation of long SAR sessions.

Chapter 4 contains the results of some of the tests made to ensure the correct functionality of SARSYS.

Chapter 5 wraps up the thesis with a summary of the key takeaways.

Chapter 2

Synthetic-aperture radar (SAR)

Synthetic-aperture radar (SAR) is becoming more and more prevalent in the modern world, be it for exploring our own blue Earth or any other planet or celestial body. In this chapter, the fundamentals of SAR are explained to provide a foundation for building upon in the following chapters.

2.1 Radar fundamentals

A radar system is composed by a transmitter, a receiver, an antenna (in many cases switched between the transmitter and the receiver), and a data processor.

As explained in the introduction, the fundamental principle of radar is very simple to understand [1]:

- The radar transmits an electromagnetic pulse to propagate in space, some of which is reflected off of a reflecting object.
- The reflected energy starts, once again, propagating through space (in all directions).
- When a portion of the reflected energy (echo) reaches the receiver, the processor detects the presence of the echo and, using the time difference between transmission of the pulse and reception of the corresponding echo, we may calculate how far that object was.

Side-looking airborne radar (SLAR) is a type of imaging radar in which the radar sensor is mounted onto an airplane or satellite and, while the platform is moving linearly in some direction (azimuth, along-track direction), the radar is pointed in a perpendicular direction while emitting a sequence of pulses with a regular frequency. A diagram representing this type of imaging radar is present in fig. 2.1.

The capability of the radar system being able discern between two targets in range is called *range resolution*, δ_s, and it mostly depends on the characteristics of the signal we are transmitting. Note that, in this dissertion, unless otherwise specified, when speaking about "range" we are referring to *slant* range, see section 2.2. Assuming we send a signal

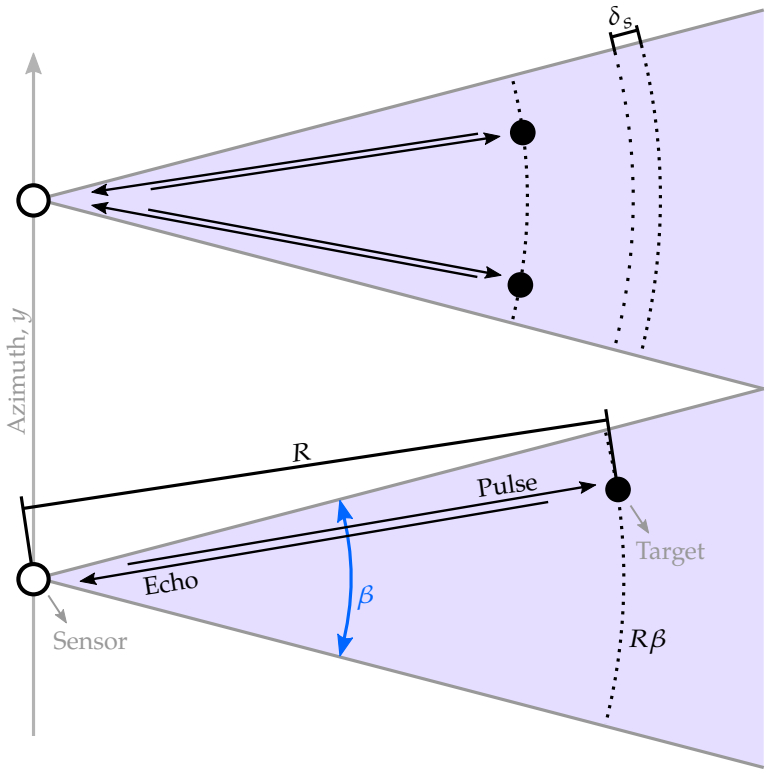

Figure 2.1: Side-looking airborne radar (SLAR) geometry (slant range view)

of duration τ and that we just check for replicas of that signal in the received echoes, we are able to distinguish between two targets separated by Δx if this separation is large enough that the echo reflected by the farthest target doesn't collide with the echo of the closest target. Knowing that the signals travel at the speed of light, c_0, this means that:

$$\frac{2\Delta x}{c_0} \geq \tau$$
$$\Delta x \geq \frac{\tau c_0}{2}$$

The factor of 2 is needed in this derivation due to the fact that the pulse has to travel Δx both in the forward and reverse direction. By definition, a resolution is the *minimum* possible separation with which the system is still able to distinguish two targets, therefore the range resolution, in this case, is given by

$$\delta_s = \frac{\tau c_0}{2} \tag{2.1}$$

In theory, this means that we can achieve arbitrarily high resolutions by just making the transmitted signal be as short as needed. However, in practice, this is infeasible because electronic amplifiers aren't able to send arbitrarily short signals with very high peak power. To solve this problem, pulse compression is employed in almost all radar systems

(see section 2.4), in which a matched receiver is able to transform long, low-power signals into short, high-peak power pulses [3].

While the range resolution analysis is the same for *traditional* SLAR and SAR systems, the main difference between them lies in the *azimuth resolution*, δ_a. In fig. 2.1, β represents the angular half-power beamwidth (HPBW) associated with the main lobe of the power radiation pattern of the antenna [8], in radians. We are considering the use of a directional antenna, with a larger gain in this direction, so most of the power transmitted by this antenna is within this main lobe. As this beamwidth is an angular property, the power diverges with increasing range and, at a range of R meters, the antenna illuminates a length of $R\beta$.

In traditional SLAR, the echoes from two targets at the same range and that are both illuminated by the antenna (ie. are within the $R\beta$ length) will arrive at the sensor at the same time, so it can't discern between them, like in the example at the top of fig. 2.1. Therefore, the azimuth resolution (ie. the minimum distance between two targets for them to be discernible) of traditional SLAR is

$$\delta_a = R\beta$$

As explained before, this resolution is not only very coarse (eg. in comparison with the achievable range resolution), but also it degrades with range! We'll see in section 2.5 that, by coherently processing echoes from pulse transmissions at different azimuth positions, synthetic-aperture radar is able to achieve very fine azimuth resolution.

In the following sections we'll analyze, in depth, how synthetic-aperture radar works.

2.2 Geometry

In fig. 2.2 we have a representation of a SAR sensor and the projection of its beam – the swath – on the ground below. In this dissertation, we'll only deal with with monostatic radar, in which the transmitter and the receiver are collocated [1] (ie. the same antenna is used for transmission and reception).

As before, we'll use β for referring to the angular HPBW of the main lobe of the antenna. In this case, we consider that the antenna is not pointing perpendicularly to the azimuth direction, but has a squint angle γ offset from the normal. The gray area is the region that is being imaged over time.

We refer to the speed of the plaform as v_p, and the speed of the ground swath (or beam speed) as v_g. The point given by the normal projection of the radar platform onto the ground plane is called nadir.

As the platform moves, the radar system *regularly* sends pulses and receives the respective echoes. The pulses are sent out every pulse repetition interval (PRI) seconds, where $\text{PRI} = \frac{1}{\text{PRF}}$ and where PRF is the pulse repetition frequency in Hz.

Note that this representation is a simplified model with some assumptions:

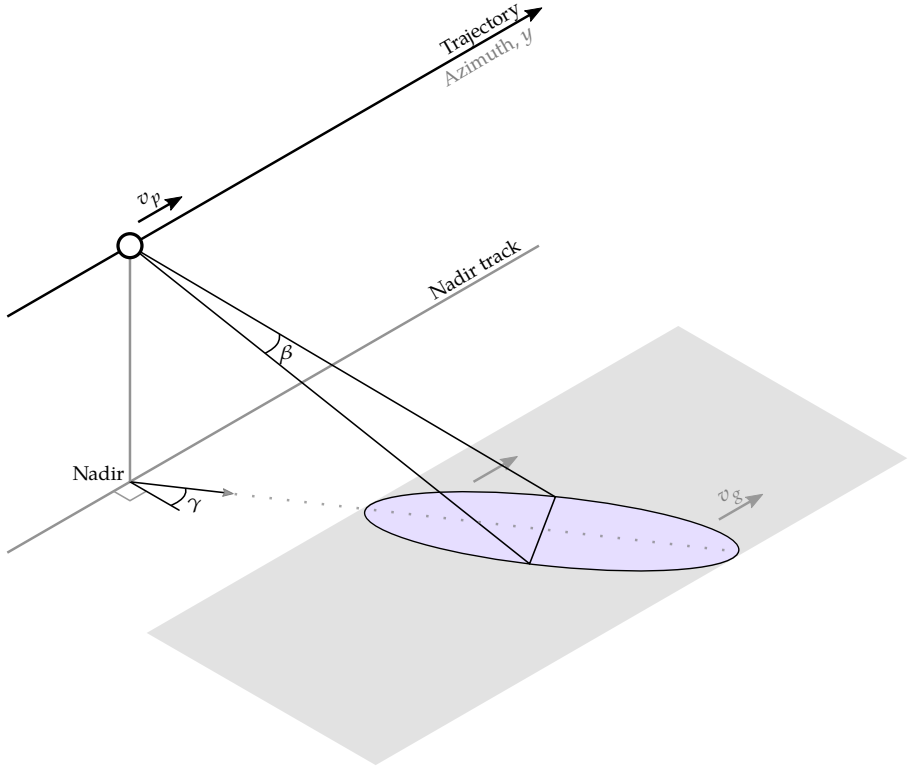

Figure 2.2: Synthetic-aperture radar beam geometry

- It is assumed that the system is using side-looking stripmap SAR, where the antenna direction is fixed relative to the platform, so the swath sweeps along the flight direction linearly. Several SAR modes of operation are described in section 2.3.
- This representation is only valid when the altitude of the platform relative to the ground, h, is *much* smaller than the Earth radius (6371 km), as we are considering both the trajectory to be rectilinear and the ground surface to be flat (implying that the platform speed is the same as the beam speed, $v_p \equiv v_g$). This is valid in the case of a high-altitude balloon (HAB) or airplane, but in the case of a satellite, the platform speed is significantly larger than the beam speed due to the difference between radii, so the analysis requires a more rigorous model which better approximates the problem – a circular model is usually sufficient in those cases.

Nevertheless, this model is very good for gaining intuition and setting a base reference for the SAR nomenclature in this thesis, and is sufficient for processing the SAR data correctly in the SARIA experiment.

The radar's line-of-sight is the defined as the direction where the antenna is pointing, and cross-range is the direction perpendicular to this line-of-sight. When squint is non-null, cross-range is *not* parallel to the azimuth direction. Synthetic-aperture radar (SAR) is actually developed along this cross-range direction but, in stripmap SAR and

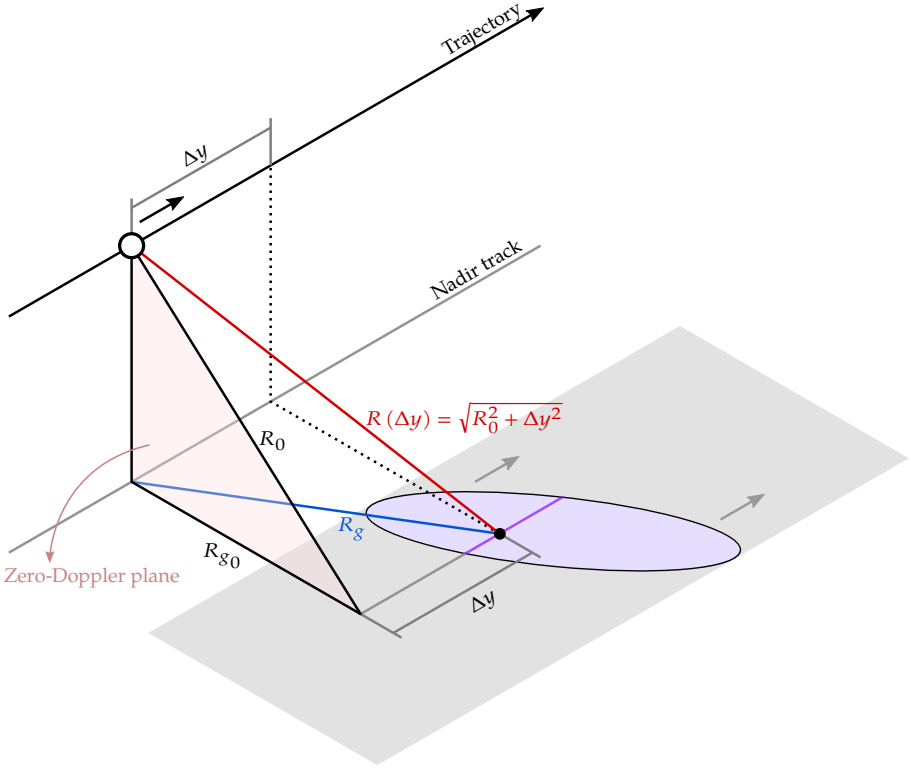

Figure 2.3: Synthetic-aperture radar target geometry

with low squint angles, the difference is insignificant.

As SAR is a linear system, we can fully characterize it through its impulse response [3, section 2.8]. In this case, the impulse response is the output of a single, isolated reflective target on the ground, which is called a *point target*.

In fig. 2.3, we now present the same model but with a point target on the ground. While that point target is inside the swath (ie. over the purple line), its echoes are received by the sensor.

The following terms may now be defined:

Slant range, R Distance between the sensor and a given point target on the ground. In general, the range is not orthogonal to the azimuth direction (as can be seen in the figure). When nothing else is said, "range" means slant range.

Ground range, R_g In this case, the distance is measured on the ground plane, from nadir to the point target.

Range of closest approach Minimum range that is achievable from the flight trajectory. It may be defined as slant range R_0, or ground range R_{g_0}. When the target is at the point of the range of closest approach, the line that connects it to the sensor is perpendicular to the trajectory line.

Zero-Doppler plane When a target is at the range of closest approach, its Doppler shift

is null. In cases such as the one depicted in fig. 2.3, when a significant squint is present, the targets are not illuminated by the beam when they pass through the zero-Doppler plane. This Doppler shift is further explained in section 2.5.

2.3 SAR modes of operation

SAR has several possible modes of operation, with different characteristics and trade-offs.

Examples of those modes are:

Stripmap SAR In this mode, the antenna is fixed to the platform, so the sensor pointing direction (line-of-sight) is always the same (relatively to the platform). The radar swath sweeps the ground at approximately uniform rate, similarly to the movement of the platform, forming a continuous image. This mode of operation will be the main focus of this dissertation, as it will be used in SARIA.

Spotlight SAR SAR coherently processes several echoes from different azimuths to improve the azimuth resolution, relatively to traditional SLAR. Therefore, the more samples we obtain, the finer resolution can be achieved. Instead of creating a continuous image like stripmap SAR, in spotlight SAR the antenna is steered to the area that is being imaged. This results in that area being exposed to the beam for longer (so we obtain a finer resolution), but only doing a spot at a time, instead of obtaining a continuous image. This mode of operation is further described in [9, 10].

ScanSAR This mode is similar to stripmap SAR, but instead of being fixed, the antenna is scanned in range, producing an image of a larger area. While in spolight SAR mode the resolution was improved by having more samples of the same region, ScanSAR does the opposite and reduces the number of looks of the same region (in order to scan a larger area). The finest achievable resolution with this mode is the azimuth resolution of stripmap SAR multiplied by the number of scans (which is worse than stripmap SAR).

Interferometric SAR (InSAR) This mode uses the differential phase from multiple observations of the same target (from either different passes at different times, or displaced observations in a single pass) to extract topographic information by means of an interferogram [11, 12].

Inverse SAR This is a very special mode of SAR where the sensor is stationary and what moves is the target. This may be used to track airplanes or satellites from a fixed location [13, 14].

Staggered SAR As seen in section 2.6, the delay of the arrival of the echoes in wide swaths places an upper bound on PRF, which may reduce image quality. This mode of operation tries to overcome this problem by continuously varying the PRF [15, 16].

2.4 Pulse compression and range resolution

We have already seen that the range resolution of a radar system that only tries to detect the transmitted pulse in the received data is given by the expression in eq. (2.1), where the limiting factor is the signal duration.

Without pulse compression, we have to transmit an arbitrarily short signal (eg. a single sine wave) so that it has the intended resolution, but it also has to have high peak transmission power in order to still be detectable above the noise when received, after performing the roundtrip to the target. However, RF front ends and amplifiers are limited in both how much they can amplify a signal, and in how short that transmitted signal may be. Besides, unbounded signal amplification is unwanted because most frequency bands have limits on how much peak power we may transmit.

The solution to all this is a technique called *pulse compression*, used in virtually all radar systems nowadays, which consists in transmitting longer *modulated* signals (as opposed to single sine waves), and using *matched filtering* in the receiver to compress the large transmitted signals into very short pulses with high peak power, just like some spread spectrum techniques for communication systems. The matched filtering just means performing a cross-correlation of the received signal with the transmitted signal (the signal we want to search for). If there are replicas of the transmitted signal in the received one (even if buried, to a degree, in the noise floor), their samples constructively add up when performing the correlation, and form high-magnitude peaks that become visible above the noise.

As we are processing digitally, and considering that $r[n]$ is the sampled received signal, and $s[n]$ is the sampled reference signal we want to search for, to obtain the pulse compressed signal $s_c[n]$ we have to perform the cross-correlation between them:

$$s_c[n] = r[n] \star s[n] = \sum_{u=-\infty}^{+\infty} r[u] s^*[u-n]$$

In fact, this cross-correlation isn't implemented directly using this expression, due to very large number of required operations, but with the help of the fast Fourier transform (FFT). This step corresponds to filtering the received signal $r[n]$ with a finite impulse response (FIR) filter. As widely known from the digital processing literature [17, 18], filtering with long FIR filters is more efficiently done in the frequency domain than by implementing the naive approach. Element-wise multiplication in the frequency domain corresponds to *circular convolution* in the time domain. Therefore, by applying the appropriate zero-padding to the signals in the time domain (obtaining $r'[n]$ and $s'[n]$), and obtaining the correct FIR filter for convoluting with (eg. by complex conjugating the reference signal in the frequency domain, $[\cdot]^*$), we can just multiply them and perform the IFFT to obtain the linearly correlated result.

$$s_c[n] = r[n] \star s[n] = \text{IFFT}\left(\text{FFT}\left(r'[n]\right) \text{ FFT}\left(s'[n]\right)^*\right)$$

For this pulse compression to work well, we need to carefully choose what signals we transmit. There are many types of signals to choose from, but the *most common* for radar

applications is the linear frequency modulation (FM) pulse, also called linear chirp, as its auto-correlation produces a very clear peak with small side lobes.

A linear chirp has an instantaneous frequency of $f(t) = f_0 + Kt$, where t is the time in seconds in the range from $-T/2$ to $T/2$, K is the chirp sweep rate, and f_0 is the middle frequency in Hz. By definition, the angular frequency of a signal is the time derivative of its phase $\phi(t)$,

$$\begin{aligned}
\omega(t) = 2\pi f(t) &= \frac{\mathrm{d}\phi(t)}{\mathrm{d}t} \\
\mathrm{d}\phi(t) &= 2\pi f(t)\mathrm{d}t \\
\phi(t) &= \int 2\pi f(t)\mathrm{d}t = \phi_0 + 2\pi\left(f_0 t + \frac{Kt^2}{2}\right) \\
\Rightarrow \phi(t) &= 2\pi f_0 t + \pi K t^2
\end{aligned}$$

With this, we obtain following chirp expression

$$s(t) = \exp\left[j\left(2\pi f_0 t + \pi K t^2\right)\right] \tag{2.2}$$

From this point forward, we'll consider, without loss of generality, that the chirp is centered at $f_0 = 0$. Given the instantaneous frequency expression, we also find that the chirp bandwidth is

$$\mathrm{BW}_{\mathrm{chirp}} = |K|\,T \tag{2.3}$$

In fig. 2.4 the spectral composition of a chirp signal is present. It has a bandwidth of 10 MHz and a duration of 70 µs.

All SDRs work with complex sampling (ie. I/Q sampling), so that means that we need a sampling rate of

$$f_s \geq \mathrm{BW}_{\mathrm{chirp}} \tag{2.4}$$

to prevent aliasing [19]. For optimal results, we must have an oversampling ratio $\alpha_{os} = \frac{f_s}{\mathrm{BW}_{\mathrm{chirp}}}$ between 1.1 and 1.4 [3].

In [3], it is shown that the *auto*-correlation of a continuous-time chirp is given by

$$s(t) \star s(t) \approx T\,\mathrm{sinc}\,(KTt)$$

We define the resolution of a signal to be the duration of the interval of time between the -3 dB points. In the case of this sinc function, that duration is given by

$$\tau = \frac{0.886}{|K|\,T} \approx \frac{1}{|K|\,T} = \frac{1}{\mathrm{BW}_{\mathrm{chirp}}} \tag{2.5}$$

The factor in the numerator is usually ignored because, most of the time, there are reasons for the resolution to be larger (eg. when smoothing windows are employed), which is also sometimes considered with a broadening factor ψ. One of the reasons for

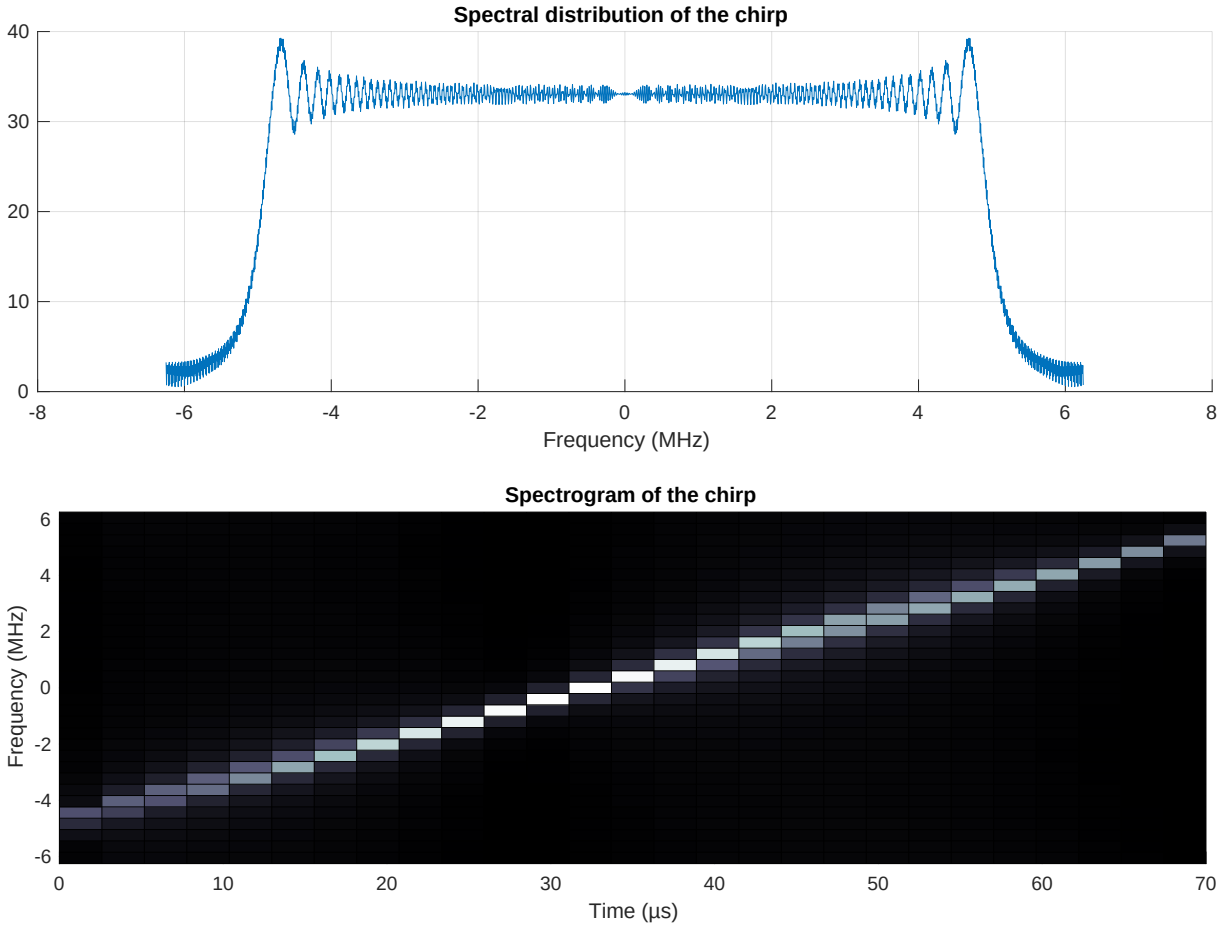

Figure 2.4: Chirp spectrum (top) and time spectrogram (bottom)

applying a smoothing window is to smoothen the large ripple seen in the pure chirp spectral content.

We may now calculate the effective range resolution obtained with this technique, which is

$$\delta_s = \frac{\tau c_0}{2} = \frac{c_0}{2\mathrm{BW}_{\mathrm{chirp}}} \tag{2.6}$$

This expression is now completely independent of the duration before compression, and only depends on the frequency bandwidth of the utilized chirp. Besides compressing the signal in time, this technique also improves the signal-to-noise ratio (SNR), as quantified by the range compression gain in the link budget, section 2.8.

2.5 Synthetic aperture and azimuth resolution

Now, we'll describe how SAR is able to achieve such fine azimuth resolutions in comparison to traditional SLAR systems.

One of the most important parameters in SAR processing is the value of the slant range from the radar to the target [3]. Considering the simplified model from section 2.2, we may calculate the range to be

$$R\left(\Delta y\right) = \sqrt{R_0^2 + \Delta y^2} \tag{2.7}$$

We note, once again, that this model is valid only in the airplane or HAB cases, where the rectilinear approximation doesn't introduce significant errors.

In SAR, the pulses are transmitted every PRI seconds, while the radar platform is moving linearly. If we consider a single point target, the range from the sensor to this point target R varies, due to progressive changes in Δy. The insight that Wiley, the inventor of the synthetic-aperture radar technique [20], had when researching this topic is that the phase of the received echoes relative to the transmitted signal is given by

$$\phi\left(\Delta y\right) = -2\pi\frac{2R\left(\Delta y\right)}{\lambda} \tag{2.8}$$

where $\lambda = \frac{c_0}{f}$ is the wavelength of the transmitted bandpass signal.

When processing SAR data, the complete range expression is used. However, for gaining some insight, we may approximate the range as a parabolic expression

$$R\left(\Delta y\right) \approx R_0 + \frac{\Delta y^2}{2R_0}$$

which results in the following phase expression

$$\phi\left(\Delta y\right) \approx -\frac{4\pi}{\lambda}R_0 - \underbrace{\frac{2\pi}{\lambda}\frac{\Delta y^2}{R_0}}_{\text{chirp}}$$

What this means is that the phase of the echo will have an offset due to the range of closest approach, and that as the platform moves, the echoes will have a chirp-like phase change. This is analogous to the Doppler shift in physics, as when the platform is moving towards the target, we have a higher frequency, and when the platform moves away from the target the frequency is lower. Note, however, that the SAR Doppler shift is a *spatial* one! We may emulate SAR using several static antennas, distributed linearly and, even though nothing is moving relatively to anything else, we would still have this effect! The only requirement is that we have several observations of the target taken from a linear path.

In fact, when we have a *moving* platform, the raw data is actually Doppler shifted in a temporal manner. While this effect is not very noticeable on HAB platforms, it needs to be accounted for in satellite platforms. Also, the temporal Doppler shift affects the range signal, which is the signal taken at the fast time, see section 2.6 for more details.

Now that we know what type of modulation the azimuthal signal has, and it is actually a chirp-like modulation, we can do precisely what we did for compressing the range signals, but now in the azimuth direction (slow time), and with a slightly different signal. Also, this reference signal varies with range, so we should use the correct one for each range gate (see section 2.6).

As this azimuth signal is frequency modulated, we can obtain the instantaneous Doppler frequency by calculating the spatial derivative of the phase:

$$f_{\text{dop}} = \frac{\mathrm{d}}{\mathrm{d}\Delta y}\left[\frac{1}{2\pi}\phi\left(\Delta y\right)\right] = -\frac{2\Delta y}{\lambda R_0} \tag{2.9}$$

The Doppler centroid, $f_{\text{dop, cen}}$, is defined to be the Doppler of the targets that are under the center of the antenna beam at any given time. In this model, this just depends on the squint angle γ.

This is the part where we need to *coherently* process several echoes to obtain a finer resolution. Coherent processing means that the radio platform must carefully control the transmission start time and phase of each pulse, and the phase of the received signal must be coherent with the transmitted pulse. This is especially hard to do when imaging distant targets, due to the large round-trip time; in such cases, *very* precise phase stability of the internal SDR clocks is required!

There are many ways to find what resolution is achievable with this method. In fig. 2.5, we see a target that is at the range of closest approach R_0. Assuming, once again, the rectilinear model, we see that the beam is projected in a length of $L \approx R_0\beta$. While the point target is inside the swath, the radar system is able to send pulses and receive echoes reflected by it.

For the same PRF, if the target was farther away, the system would receive more echoes coming from it. In traditional SLARs (also called real-aperture radar), this was a problem, as it meant that the resolution was getting coarser with range (ie. a single target spans more azimuth cells).

However, with SAR, this is actually a good thing! By coherently processing echoes from several locations of the trajectory, SAR can be conceptually seen as beamforming with a

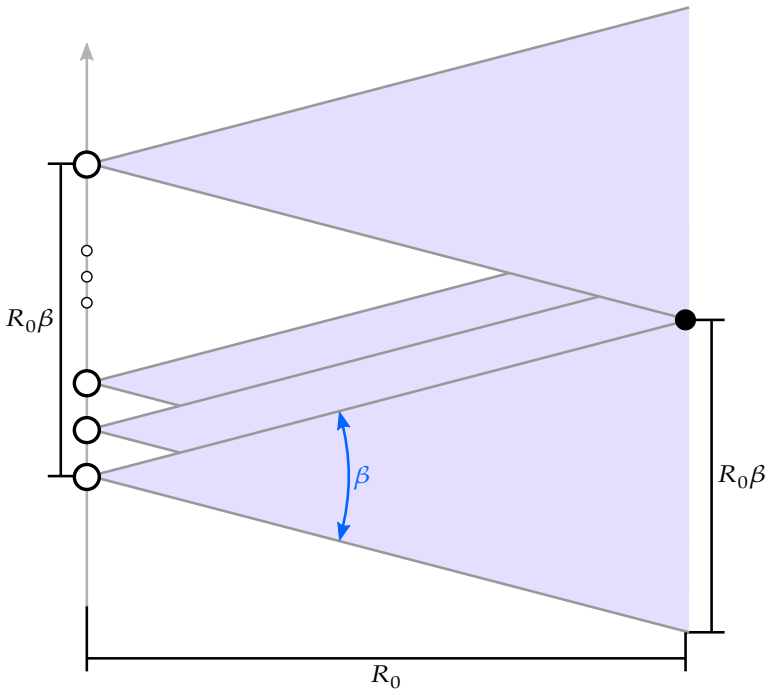

Figure 2.5: Synthetic-aperture radar data acquisition

moving platform. Therefore, by obtaining more echoes from the target, we obtain more data for its phase history, which means that the "beamforming" has a larger (synthetic) aperture due to the increased number of elements and, in turn, a finer beam.

Seen from other perspective, similarly to what happened in the range compression, the azimuth resolution is obtained by the reciprocal of the Doppler bandwidth [3]. The Doppler bandwidth may be calculated from eq. (2.9),

$$\mathrm{BW}_{\mathrm{dop}} = \frac{2L}{\lambda R_0} = \frac{2\beta}{\lambda} \tag{2.10}$$

from which we arrive at an azimuth resolution of

$$\delta_a = \frac{\lambda}{2\beta}$$

The HPBW, β, may be approximated by the geometry of the antenna. In this case, considering the length of the antenna in the azimuth direction to be L_a (in meters), the approximation for the HPBW is that $\beta = \frac{\lambda}{L_a}$. Replacing this in the previous resolution expression, we obtain

$$\delta_a = \frac{L_a}{2} \tag{2.11}$$

This means that SAR has a *range-independent* azimuth resolution that *only* depends on the geometry of the antenna! Even more so, that resolution improves when the antenna is shorter, which is counterintuitive because this widens the HPBW, and in traditional SLAR would degrade the resolution. However, in SAR, the resolution improves because

we obtain more observations of the target with wider displacements, which increases our synthetic aperture.

Some details have been left-out until now:

- Like explained, a shorter antenna has a wider beamwidth (less directivity), and better SAR resolution. However, using a smaller antenna without increasing the transmission power may degrade the link budget (section 2.8) so much that no echoes are detected at all.
- Furthermore, as seen in eq. (2.10), the Doppler bandwidth increases with the beamwidth, which is larger in a shorter antenna. Unfortunately, due to the sampling theorem, we have absolute limits on how large a signal bandwidth may be before it becomes aliased after sampling. Therefore, using a shorter antenna without having a high enough PRF introduces ambiguities in the generated image. The PRF bounds are further analyzed in section 2.6.
- The range equation is also very important for another reason, which is the range cell migration (RCM). This is discussed in section 2.6.

2.6 SAR raw data

SAR raw data is typically represented in a matrix, with the axis being azimuth and *slant* range. In this thesis, for consistency's sake, we'll consider that range varies along the matrix lines, and the azimuth along the columns. In this case, each cell contains a complex sample, each column is called a range gate, and each line is called a range line. This is, of course, just a conceptual model that has no impact on how the data is actually disposed in memory.

When a pulse is transmitted, it will take some amount of time for its echoes arrive. In fig. 2.6, we have a side view of the previously shown geometry. We see that the depression angle is referred to by ζ (in radians), and that the platform is at an altitude (referred to ground level) of h meters. Also, the HPBW for the elevation plane is given by β_{el}.

With this information, it is trivial to calculate the distances a and b and, with that, the useful capture time that contains the echoes,

$$t_{\text{cap,b}} = \frac{2a}{c_0} \leq t \leq \frac{2b}{c_0} + T = t_{\text{cap,e}}$$

Note that the upper limit is extended by the duration of the transmitted pulse, T, because we want to capture the entire echo of the farthest considered target. We'll define the capture window duration to be $\Delta t_{\text{cap}} = t_{\text{cap,e}} - t_{\text{cap,b}}$.

When the echoes are sampled (with a suitable sampling rate, see eq. (2.4)), they are saved to the range line associated with the current azimuth (coherently!). As this pulse and the echoes travel at the speed of light, the time in this axis is also called *fast time*. Due to the much slower speed of the platform (by comparison to the speed of light), the time in the azimuth axis is called *slow time*.

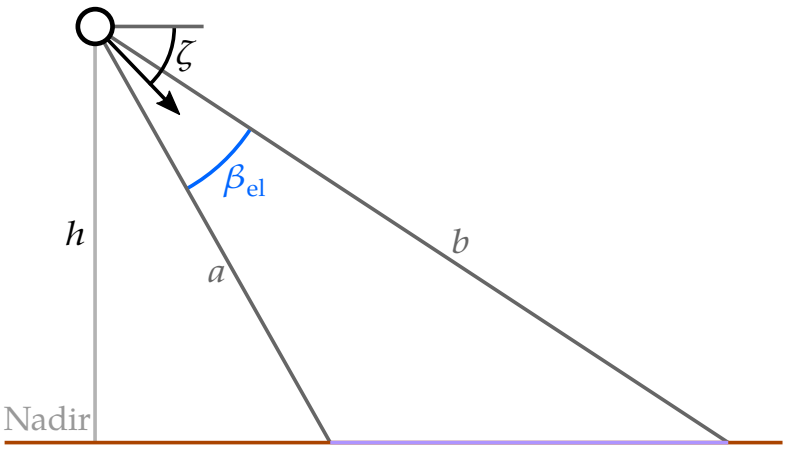

Figure 2.6: Side view of a radar platform

Having already spoken about the range sampling rate, we have yet to set bounds on how fast we must send pulses and capture their echoes; in other words, what limits do we have on the PRF?

- On one hand, we have a minimum bound on the PRF due to the fundamental result of the Nyquist-Shannon sampling theorem [17]. If the system has a sampling rate smaller than this bound, we'll have aliasing in the image. This bound is defined by the azimuth Doppler bandwidth, $\mathrm{BW}_{\mathrm{dop}}$. The easiest way to find what this lower bound is, is that we have to sample in azimuth *at least* enough to have a sample for each azimuth resolution cell, which means that, as the platform is moving, it has to send a pulse at least every δ_a meters [2]. If the platform moves faster, the PRF must increase; if it moves slower, it may decrease.

- On the other hand, we also have a maximum PRF bound due to the swath width. As described just before, we'll have to capture a window of echoes of duration Δt_{cap}. If the PRI is smaller than $T + \Delta t_{\mathrm{cap}}$, then the echoes associated with a transmitted pulse will overlap with the end of the echoes of the previous one. When using conventional stripmap SAR with just a single pulse, this causes range ambiguity and ghosts of the image may appear in the range direction. Therefore, in order to prevent this range ambiguity, we need to have a PRI of *at least* $T + \Delta t_{\mathrm{cap}}$, which means we have a maximum PRF of $\frac{1}{T+\Delta t_{\mathrm{cap}}}$.

- In fact, if we want to generate *continuous* images, there is actually another problem that usually shows up first. As seen, the echoes of each pulse take $t_{\mathrm{cap,e}} = t_{\mathrm{cap,b}} + \Delta t_{\mathrm{cap}}$ to be received. Even if the PRI complies with the previous restriction relative to the range ambiguity ($\mathrm{PRI} > T + \Delta t_{\mathrm{cap}}$), it may still be less than this total time. More precisely, when $t_{\mathrm{cap,b}} < \mathrm{PRI} < t_{\mathrm{cap,e}}$, we'll have what is called *blind ranges*, as there are some ranges whose echoes will arrive when the radio platform is transmitting another pulse, which effectively blocks those echoes from being received.

This is what staggered SAR was designed to solve: by continuously varying the PRF, we can spread these blind ranges over the image, and then the azimuth compression "evens out" and fills the blind ranges with based on the data from other azimuths (and at the cost of some SNR) [15, 16].

Considering the case where we only have a single point target, if we plot the SAR data after having filled all range lines, and performing range pulse compression on each one, we'll see that we now have a linear trail over several range lines, created by the echoes of this target. However, due to an effect called range cell migration (RCM), this trail is not on a single range gate, but passing through several, with a shape similar to a parabola. This happens because the *slant* range (that is represented by the horizontal axis) is minimum when the target is at the point of closest approach, but is larger when not on that point. In other words, precisely what gives us an advantage of processing SAR as opposed to real-aperture radar – the phase variation along azimuth, due to the varying range to the target – is what causes causes this problem to appear, as the range variation is generally large enough that it moves the echoes off of the correct range gate. If RCM correction is not performed, the usable antenna aperture available for processing is reduced, as the azimuth correlation is done along each range gate column.

Finally, we must also refer that having precise navigation data (eg. from Global Positioning System (GPS), ideally coupled with an inertial navigation system (INS)) is crucial for correctly focusing SAR data, as it is used while performing the motion compensation (MoCo) step before coherently processing the azimuth data (section 2.9). Besides, the navigation data also provides the possibility of generating geocoded images.

2.7 The range Doppler algorithm (RDA)

The range Doppler algorithm (RDA) is the most commonly used algorithm for performing synthetic-aperture radar processing. It was designed to achieve block processing efficiency using frequency domain operations in both range and azimuth [3].

It is composed by the following steps:

1. **Range compression** just as described in section 2.4

2. **Azimuth FFT** In this step, with the time domain in both axis, an azimuth FFT converts the matrix to a range-Doppler domain. As before, note that this Doppler is *spatial*, so the units in the vertical axis are cycles per *meter*. The Doppler associated with line m is $f_{\text{dop}} = \frac{m}{M}\frac{1}{\Delta y}$, where M is the number of lines.

3. **RCM correction** The migration of each cell depends on its distance to the zero Doppler plane. So, the migration distance Δx may be written as a function of the

Doppler of each cell [3]:

$$R_0 + \Delta x = \frac{R_0}{\sqrt{1 - \left(\frac{\lambda f_{\text{dop}}}{2}\right)^2}}$$

$$\Rightarrow \Delta x = R_0 \left[\frac{1}{\sqrt{1 - \left(\frac{\lambda f_{\text{dop}}}{2}\right)^2}} - 1 \right] \tag{2.12}$$

In the range-Doppler domain, the problem becomes very simple, as the Doppler is given by the vertical axis. The only thing that is needed is to *reinterpolate* the data such that the mapping $R_0 + \Delta x \rightarrow R_0$ is performed.

4. **Azimuth compression** Now that the RCM has been corrected, azimuth compression is analogous to the range compression.

5. **Azimuth IFFT and final fixes** Finally, the data is converted back into range-azimuth domain. Final fixes may include, for example, the reprojection of the *slant* range data onto *ground* range. This projection is done by reinterpolation with the mapping $R' = \sqrt{R^2 - h^2}$.

While this algorithm is relatively simple to prototype and generate images given a set of pre-processed or simulated SAR data, it is actually very hard to implement and integrate it in an end-to-end system, that has to process real data and be optimized well enough to comply with real-time requirements. The latter is the goal of this thesis, and the design process and implementation details will be extensively described later on.

Besides, note that this algorithm is only part of the entire process, and is actually still missing some steps (eg. motion compensation).

2.8 Link budget

The link budget is one of the first steps in the design of a radar system (or any radio system, really). It is usually presented in table format, and lists all gains and losses of the system (in dB), in order to sum them all at the end and arrive at the conclusion if the system closes the link budget (ie. works) or not.

The following terms are usually considered when dealing with a SAR system:

TX power How much power, in dBm (or other power unit), is being transmitted by the system. This is the average power of each pulse.

TX antenna gain Characteristic of the antenna in use, associated with how directional it is. The free-space losses (FSL) consider an isotropic antenna (unitary gain in all directions). This gain term is what ajusts the link budget to the antenna used by the system [8].

Forward free-space losses (FSL) Characterizes the divergence of power through the beamwidth. Derived from the Friis transmission formula,

$$\mathrm{FSL} = 20\log_{10}\left(\frac{\lambda}{4\pi R}\right)$$

In this case, we consider this term to be negative so that it can be summed with the others when calculating the link budget. Note that the 20 implies that the slant range R has a quadratic contribution to this loss term.

Radar projected cross section This is a characterization of how much a 100 % reflective target of area A would reflect back to the sensor. The formula for calculating this term is

$$\sigma = 10\log_{10}\left(\frac{A}{\frac{\lambda^2}{4\pi}}\right)$$

A is the area that we are interested in (related to the resolution of the image).

Minimum distributed target reflectivity Lowest reflectivity index that is desirable to be detected from the noise level. This term deals with how clear we want an image to be and what we want to distinguish or not. As this deals with perception, there isn't a formula to calculate this value. Together with the radar projected cross section, they form the radar cross section (RCS).

Backward FSL and RX antenna gain Similar to the previous terms, but now in the reverse direction. In monostatic SAR, they have the same values.

System losses This term characterizes the RF RX front end. It is calculated from the noise figures of the amplifiers chain, up until the digital sampling device. For this term, the most important aspect is the noise figure and the gain from the first stage of amplification [21]. This term also includes some margin for dealing with unexpected system losses.

Noise Calculated from the equivalent noise temperature T (in kelvins, K), the system reception bandwidth B and the Boltzmann constant k_B,

$$P_{\mathrm{No}} = -10\log_{10}\left(k_B T B\right)$$

Range compression gain When perfoming pulse compression, the average power of the signal is conserved. So, the power that was once spread over T seconds, becomes now compressed into a short peak of length $\tau' = \frac{1}{\mathrm{BW}_{\mathrm{chirp}}}$, eq. (2.5). Therefore, we have a gain of

$$G_{\mathrm{R}} = 10\log_{10}\left(\frac{T}{\tau'}\right) = 10\log_{10}\left(\mathrm{BW}_{\mathrm{chirp}} T\right)$$

The product $\mathrm{BW}_{\mathrm{chirp}} T$ is actually known as the time-bandwidth product and is an important parameter to have in consideration while designing any radar system.

Azimuth compression gain Likewise, in azimuth, the power that was spread over several samples within aperture $L \approx R_0\beta$ becomes compressed into a single pixel.

Given the PRF and v_p, we know that the distance between consecutive points is $\frac{v_p}{\text{PRF}}$. So, the gain is given by

$$G_{\text{A}} = 10\log_{10}\left(\frac{L}{\frac{v_p}{\text{PRF}}}\right) = 10\log_{10}\left(\frac{R_0\beta\text{PRF}}{v_p}\right)$$

Once again, this can be seen as a TBP, where time is the target exposure time $\frac{R_0\beta}{v_p}$, and the bandwidth is the PRF.

Low-pass filter (LPF) gain and decimation Finally, to improve image quality, a low-pass filter is usually employed. This LPF reduces the resolution but usually even increases clarity and attenuates the effect of problems like the SAR speckles (see section 2.10). The gain associated with this step is simply the conversion of the decimation factor into dB.

The calculations for the link budget are mostly derived from the so called radar equation [22].

The main difference between the link budgets of a real-aperture radar and a SAR system, is the azimuth compression gain factor. Note that, without term, the range would have power of 4 contribution, from the forward and revere FSL. With SAR, we are able to reduce the power of the range to only have a cube contribution which is a very significant improvement.

2.9 Motion compensation (MoCo)

Motion compensation is an absolutely critical step of processing real SAR data. As the radar platform moves, its real path won't be the idealized straight line that is considered in the underlying model of our analysis. These errors, if uncompensated, result in very significant adverse effects on the generated image, such as unfocusing and ghosting.

To give a sense of perspective, at the carrier frequency of the SARIA experiment (5.8 GHz), we have a wavelength of $\lambda \approx 5$ cm, and a difference in slant range of just a quarter of this wavelength, which is 1.25 cm, is enough to make the sample be completely incoherent. Of course if it is just one sample, it won't have much effect, but this gives a sense of how sensitive SAR is to unaccounted random variations in range.

In the literature [23], we find that the motion errors are usually decomposed into 3 different types:

Translational errors occur when the platform is displaced from the idealized straight line that is assumed throughout the model, see fig. 2.7.

This introduces a (possibly *range-variant*!) phase error in the targets phase history and, if significant, if may even shift the targets off of the expected range gate.

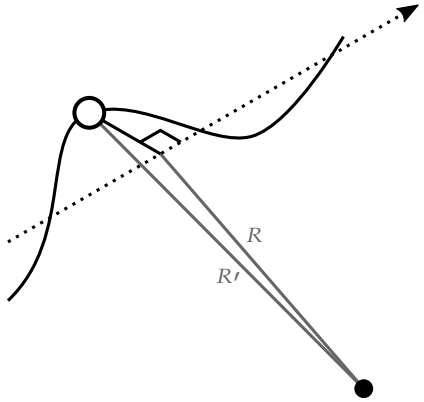

Figure 2.7: Representation of the translational error.

Attitude errors occur when the platform changes yaw, pitch, or roll unexpectedly. When rolling, the projected swath moves in relation to the platform nadir, so it is basically just a gain error. However, errors in pitch and yaw introduce squints that cause a shift of the Doppler centroid, which is very hard to correct when varying temporally. When this Doppler centroid is not accounted for, the azimuth spectrum may become aliased, resulting in an ambiguous image in the azimuth direction.

PRF spacing errors occur when the velocity of the platform is not constant. In the underlying model we have, we assume that the azimuth samples are taken at regularly spaced samples on ideal the straight line. For the azimuthal matched filtering to work correctly, we have to correctly compensate this error in order to obtain a regular spacing between samples.

Note that these errors are measured relative to the phase center of the antenna! For example, a roll measured on the center of an airplane would actually *additionally* cause a significant translational error if the antenna was placed on, eg. a wing. When the offset is not insignificant, in order to properly analyze these errors, appropriate coordinate transformations have to be done.

There are many ways to perform motion compensation, as seen in the literature [23–25]. The way motion compensation is implemented in the system presented in this thesis is both a mixture of several techniques, while also a simplification in some ways. It is described in detail in section 3.5.3.

2.10 Multilook

Radar speckle is the granular noise, sometimes described as a “salt and pepper” texture, that all radar images have in common when not filtered. It results from the constructive or destructive interference of many elementary scatterers within a single cell, and results in the magnitude of each pixel following a Rayleigh distribution [3].

Due to its random nature, the speckle complicates image interpretation. There are two main ways of fighting it: one way is to apply general low-pass filtering techniques after the image is generated, and the other is to use multilook processing.

There are two main ways of generating multilook images:

Multilook in the frequency domain In this method, the RDA is performed as if there was no multilook present, right up until the last step of the azimuth IFFT that converts the data from the range-Doppler domain to the final image.

The signal in azimuth has a close to linear Doppler FM correspondence, with the positive frequencies corresponding to the directions forward from radar platform, the close to null frequencies being the region near the zero Doppler plane, and the negative frequencies being directions backward from the radar platform (assuming a null Doppler centroid, but the analogous applies to a non-null Doppler centroid). So, if we divide the azimuth spectrum in several blocks, we are effectively extracting data that was captured from several sections in the *antenna beamwidth*.

Therefore, *after* applying the azimuth matched filter to the entire spectrum, instead of performing a single IFFT of the entire azimuth data for each range, we actually divide the spectrum in several "looks", with bandpass filters applied symmetrically around the Doppler centroid, perform the IFFT of *each one* of those looks, and finally combine them *non-coherently* just by either summing their magnitude values, or by summing their magnitude squared values and performing a square root of every pixel in the end.

Also note that this method automatically reduces the azimuth resolution of the image by approximately the number of looks taken in the spectrum (as expected, we are only using a fraction of the bandwidth and the width of a pulse is given by the bandwidth of that pulse).

Multilook in the time domain As opposed to the previous method, this one is different right off the bat. After performing the range compression and MoCo, this method segments the azimuth data in several equally sized blocks. Then, it just applies the remaining steps of the RDA, just like it was the original data vector.

As the azimuth impulse response is usually very large (remember that its size grows with $L \approx R_0\beta$), this means that the image generated by each look will probably have a size that is on the same order of magnitude of the image that would result from processing the original data block. However, just like the previous multilook algorithm, it will have reduced resolution! As we only select a subset of the captured data, the azimuth compression will only have a fraction of the data associated with each target. As the frequency is almost linearly modulated in azimuth, each target will only have a fraction of the bandwidth, so the compressed peak of each will widen accordingly.

Therefore, even though this multilook technique generates large images, we may decimate them by the same proportion we have of each target's bandwidth. And, just like the previous method, this one also just sums the single look images together *incoherently* to form the final image.

Both these methods reduce the speckle in the image because the origin of the speckle was the summation of *coherent* samples that had random variation due to the elemental scatterers – these methods join the looks together by summing them *incoherently*, so the

problem is attenuated similarly to how a low-pass filter reduces random noise.

Now, either method has advantages and disadvantages. At first, it may seem that the second method is strictly inferior compared to the first one; and that is true to a degree, both methods do the same thing in the end, and the second one has to do a lot more work to obtain the same result.

However, the second method has a very significant advantage! As this method processes subsets of the slow-time extracted directly from the raw data, it allows the generation of partial images, *as the data is being streamed in* (the first method requires the entire data to be present before starting to process it) and, perhaps more importantly, it also allows the MoCo to be *local* to each single look!

As referred, MoCo is a very difficult process that usually focus data in a place at the cost of defocusing other parts. By doing this MoCo locally, we have more guarantees that the MoCo doesn't need to be so agressive – the platform has less probability of significantly deviating from the ideal path if the time in consideration is shorter – and, when joining the images together, they are already in a lower resolution state, so it is more flexible to errors when stiching the partial images.

Nevertheless, we recognize that the second method is more expensive in comparison to the first one, even more so as we are looking for ideas to optimize the processor in order to have it run within real-time constraints in embedded systems so, while it may be very useful, it is prohibitively expensive for us. That is why, in section 3.5.3, we present a novel multilook method that is as flexible as the time-domain technique, but that is able to cut down the costs in a significant manner.

2.11 Optimization of SAR processing software for performance

Given that the goal of this dissertation is the design and development of a system that is able to generate SAR images within the real-time constraints of SARIA, we must also focus on techniques that are used to improve the runtime performance of software. Some of these techniques are domain independent, and may be applied to all kinds of software, and some techniques require specific domain knowledge to be done, so their usefulness is limited in scope.

In general, we consider there to be two large types of optimization that lead to better performing software:

Algorithm optimization This type of optimization works by changing the algorithm itself, which means that it changes *what* work is done – of course, while still obtaining the same (or good enough) results. This is the type that we consider to be domain-specific, based on insights of how the program being optimized works in a foundational level.

A good, classical example of this type of optimization was the invention of the FFT algorithm. Before the invention of this new algorithm, the calculation of the

discrete Fourier transform (DFT) was a $O(N^2)$ process. However, a change in *what* steps are taken, that still arrived at the same final result – the DFT of the input signal –, allows us to achieve the same transformation in just $O(N \log N)$. In order to invent the general FFT algorithm, deep knowledge of mathematics and signal processing was required.

In our case, this type of optimization must be done in such way that, while the final images may not be a pixel perfect match with the original algorithm, they must still be generated without noticeable adverse effects and distortions, in order to maintain the expected quality. Sometimes, removing operations or replacing them by alternatives may speed up the processing by a large margin while still creating very clear images.

Implementation optimization This type of optimization changes *how* the work is done. Even after having defined the several steps required to perform a given algorithm, the way it is implemented will probably affect its runtime performance *very* significantly. The reason for this is that the underlying computer systems that execute the chosen algorithm are *not* pure Turing machines, they all have their own implementation details and different feature sets. That is why we consider the *techniques* for performing this kind of optimization to be more general and independent from each specific application, as the optimization is more guided towards making it easy for the computer to execute our algorithm, independently of what algorithm it is.

Therefore to do these optimizations we usually have a specific target computer architecture in mind. We first need to know what is the least common denominator of features that we are willing to support, so that we can use them to our advantage.

A trivial example, that illustrates the optimization problem in a simple manner is the optimization of a loop that multiplies two arrays together. How long are the arrays? Do they fit memory or are they being streamed in? Do we support single instruction multiple data (SIMD), so that we may perform this multiplication wide? May the operation be done in-place? If the arrays are extremely long, maybe having several threads to perform the work would provide some speed-up.

Before optimization, we should have a baseline of how long the execution time should be in the end – by analyzing the ideal runtime performance by hand, from first principles, or by measuring other implementations and finding that they could be faster in some way –, so that we know what is feasible or not.

Besides, we *shouldn't* also start optimizing the entire program. Writing code for performance may hinder its readibility and the flexibility of quickly performing changes to that code, and the optimization may be unnecessary anyway! We should always start by profiling the program to find what its slowest parts or the bottlenecks actually are, so that we start optimizing the most relevant parts. We should also profile it afterwards to find if the runtime actually improved or if we should continue exploring alternatives.

Besides profiling, the CPU performance counters (that may be obtained using a utility such as `perf` or Intel's VTune) may be useful to find if there are many cache

misses, mispredicted branches, port usage, etc. Sometimes, just optimizing by improving the cache coherency provides very significant speed ups – compared to the CPU clock frequency, fetching data from random-access memory (RAM) is slow!

Finally, we should also warn that we can't trust the compiler to generate good assembly code on its own. When performing this type of optimization, we usually want the compiler to generate very specific instructions. To achieve this, we may use compiler intrinsics, that usually map more directly to specific instructions, or inline assembly, that provides the greatest degree of freedom but is also much more difficult to do, and we should always check the disassembly to see if the compiler generated what was expected.

Note that even though we separated optimization work into two different types, they are actually not orthogonal! Some algorithms may lend themselves better to a certain computer architecture, and therefore being able to have a more optimized implementation, so we must always take both things into consideration.

An interesting example of software with strict real-time constraints, that really has to be optimized using both types of optimizations are video games, because they usually have very small time spans to work with (16 ms if running at 60 FPS, and even less when targeting virtual reality (VR) headsets), and have to perform gameplay logic, physics simulations, frame rendering, etc. Video games are known for using very ingenious tricks in order to make them feasible to run in real-time ("algorithm" optimization), and are, many times, what push the computer technology forward, by always using bleeding edge features provided by modern hardware and using the resources of the computer efficiently (implementation optimization). That's why the course *Handmade Hero*, where Muratori teaches game development "from scratch" is a very good resource for this work as they have already focused on many implementation optimization topics like cache and CPU utilization, SIMD, multithreading, and profiling [26].

More excellent resources, this time solely focused on performance optimization techniques for the x86 and x86-64 family microprocessors (and focusing on C/C++), are [27–29].

Chapter 3

The SARIA System (SARSYS)

In this chapter, we present the system that was designed and developed as part of this dissertation, which we have named SARIA System (SARSYS).

3.1 Brief experiment description

As explained in the introduction, SARIA was selected, in 2019, to fly as part of cycle 13 of the BEXUS program, which is a cooperation between the German Aerospace Center (DLR) and the Swedish National Space Agency (SNSA) that allows students from higher education institutions to study experiments on board of stratospheric balloons. Through a collaboration with European Space Agency (ESA), the opportunity has been made available for students across all ESA member states, Slovenia and Canada [30].

A HAB is an almost ideal platform for performing SAR experiments, due to its (relatively) slow velocity and short-term stable trajectory. The primary objective of SARIA is to obtain synthetic-aperture radar images of the ground under the flight path using an inflatable antenna, and using the license-exempt 5.8 GHz ISM band, within all associated power restricions.

In a macro scale, the experiment is composed by the inflatable reflector antenna, a static reflector antenna (that will be used as a reference and backup for the inflatable one), and the main box that houses all the electronics. In fig. 3.1, you may see how this is disposed on the BEXUS gondola – the other team's experiments are not shown.

In fig. 3.2, a functional diagram of the experiment is present. In gray are the blocks that control the experiment, such as the on-board computer (OBC) and a microcontroller (Arduino). In orange, we have the blocks associated with navigation data acquisition, and in blue are the RF-related components that will be used to perform the SAR sessions. Finally, the purple blocks are image/video components that are used as control for the experiment, and in green we have the pressure control system for the inflatable antenna.

The system developed as part of this dissertation, SARSYS, will run on the OBC and has to interact with and automatically integrate the SDR and the navigation sensors. The

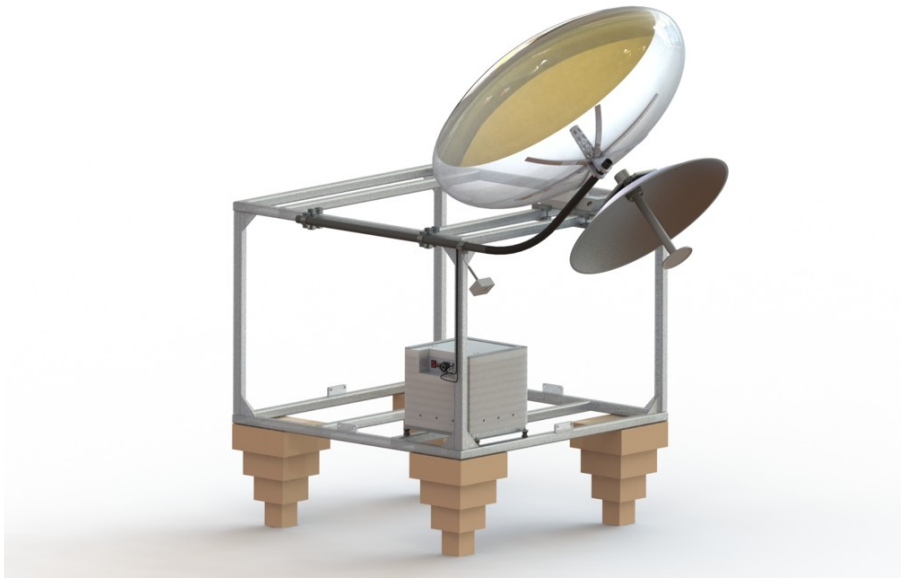

Figure 3.1: BEXUS gondola with the SARIA experiment mounted.

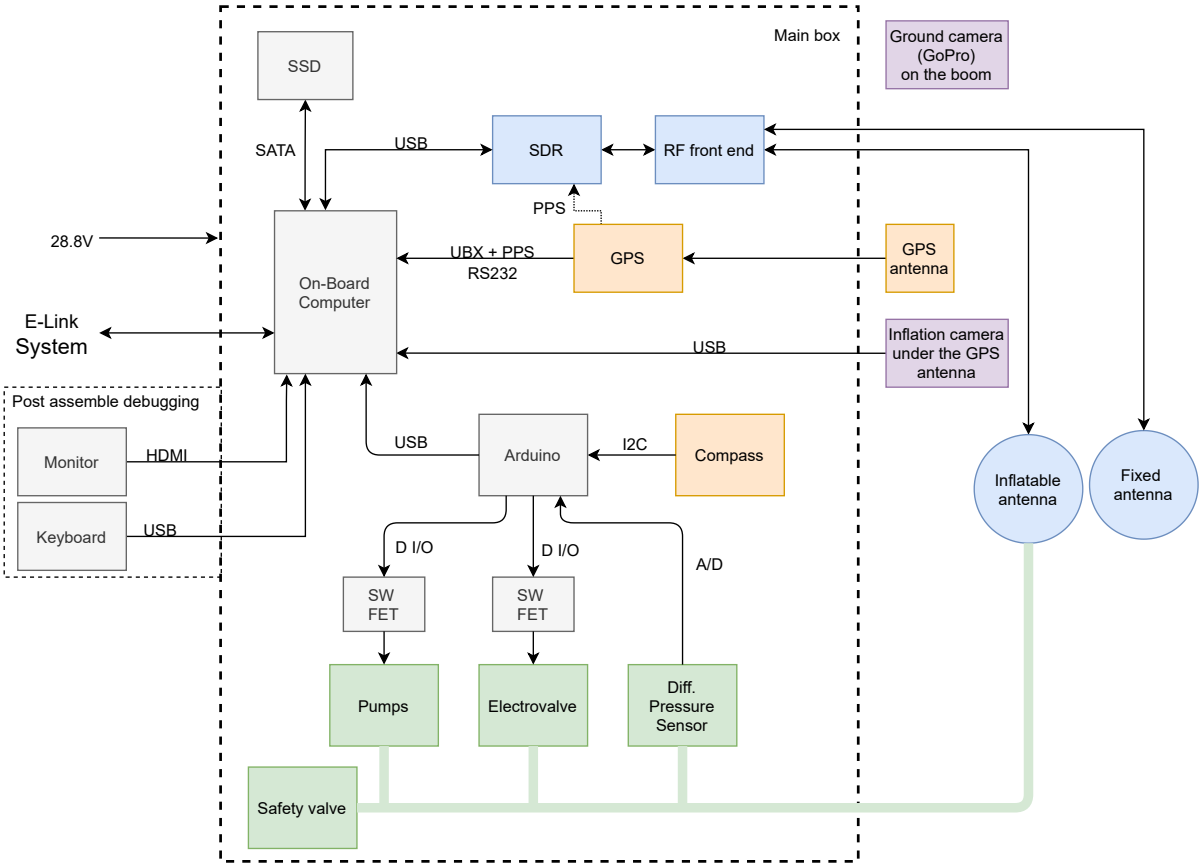

Figure 3.2: Functional diagram of SARIA.

OBC is an ASRock Q1900M, with an Intel Quad-Core 2 GHz processor J1900, and 8 GB of RAM. The target platform is Debian Buster, which comes with Linux 4.19.

The SDR we are using is an Ettus B200mini, and the GPS receiver module is an u-blox TIM-LF.

As explained, in order to prevent complete loss of data if a crash occurs when landing, we decided that the SAR processing of at least some sessions should happen entirely in the OBC, and the resulting images should be downloaded through the rate-limited telemetry link. This also allows us to better diagnose if there is any problem with the experiment during the campaign, mitigating some risks of mission failure.

As seen in the literature review, SAR processing usually assumes either a linear or circular model. In this case, we'll use a linear model as it approximates our problem well. Therefore, the best time to perform SAR sessions is during the gondola's *float phase*, in which the BEXUS ballon has already arrived at its final altitude and is traveling at approximately uniform pace, while slowly rotating. Considering fig. 3.3, we can assume we'll have at least 1h30min of float time. However, we'll have only a fraction of that time to perform our measurements, as the antennas may be pointing in a direction with *too much* squint for them to be usable to generate clear images.

Due to the large CPU and RAM/bus bandwidth usage of both the acquisition and the SAR processing, they can't run in parallel – note that the acquisition program has very strict real-time requirements due to the high sampling frequency of the SDR and, if it misses the deadline, the SDR queues overflow and drop samples, which deteriorates image quality. Therefore both programs must be optimized so that we are able to acquire several sessions and actually process some of them during the launch campaign. These requirements are quantified in section 3.1.2.

For more information on the inflatable antenna characteristics, check Santos [32], and for more information relating to the RF front end, see Martins [33].

3.1.1 SARIA link budget

We are planning each session to be 1 min long, and the antennas to be pointing at a 45° depression angle. The specific characteristics of the transmitted pulse and range compression are further described in section 3.4.1.

The link budget in table 3.1 follows the nomenclature of section 2.8.

The transmitted power (20 dBm) is within the specified limits of the frequency band that will be used, the 5.725 GHz–5.875 GHz ISM band.

The expected antenna gain corresponding to a parabolic dish with a diameter of 1 m for 5.8 GHz and 50 % efficiency is 30 dB.

For a flight altitude of 30 km (fig. 3.3) and performing stripmap SAR with 45° depression angle, the target distance is roughly 45 km, corresponding to −141 dB of free space losses at 5.8 GHz.

A signal bandwidth of 10 MHz (section 3.4.1) gives us a range resolution of 15 m. The

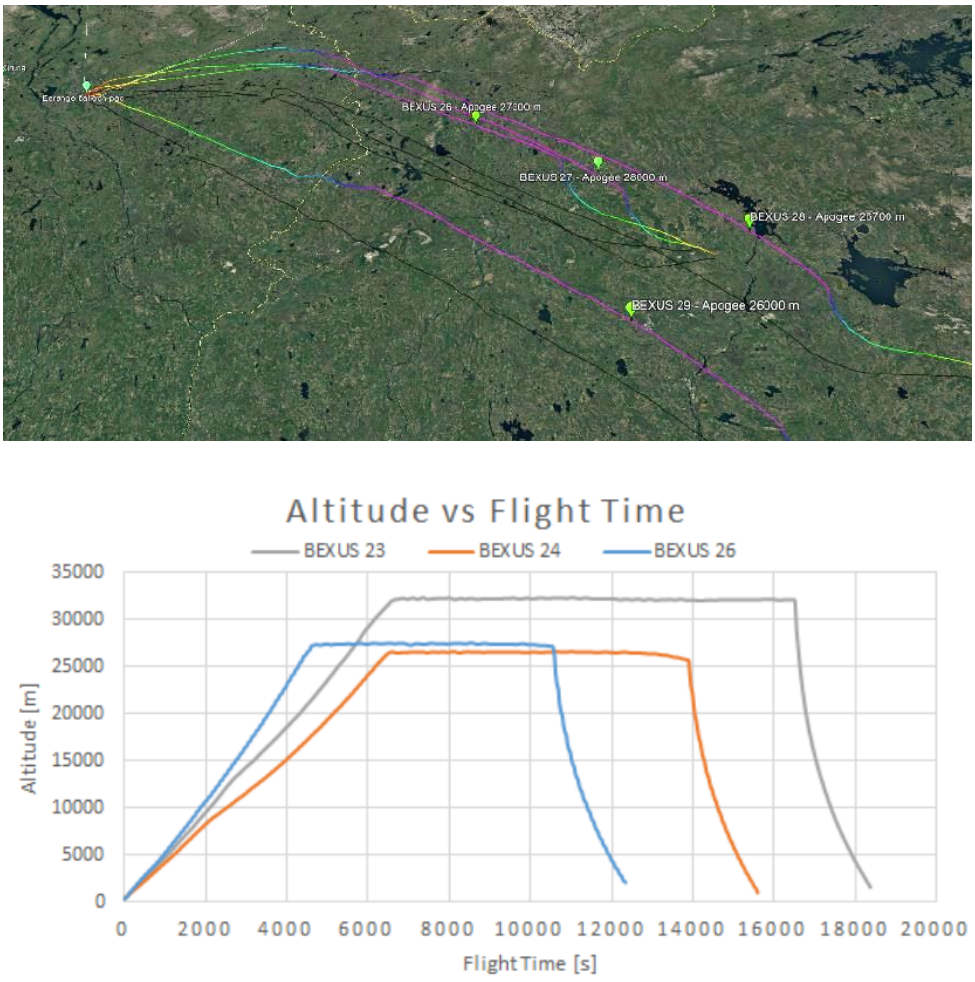

Figure 3.3: BEXUS past flight profiles, taken from *BEXUS User Manual* [31].

raw azimuth resolution is half the antenna length (0.5 m). This leads to a 7.5 m^2 projected cross section area, corresponding to 45 dB radar projected cross section at 5.8 GHz.

The minimum target reflectivity is the lowest reflectivity index that is desirable to be detected from noise level (−25 dB). Together with the projected cross section, a RCS of 20 dB is obtained.

Diverse system losses correspond to receiver noise figure and other losses, which amount up to 5 dB.

As the signal has a duration of 70 µs and a bandwidth of 10 MHz (section 3.4.1), the range compression gain is 28 dB. The azimuth compression gain is given by the number of pulses integrated within the aperture $L = \frac{\lambda}{L_a} R = \frac{0.052}{1} 45\,000 = 2340$ m With a PRF of 1 kHz and knowing the gondola is moving, at most, at 200 km/h (55 m/s) the distance between 2 consecutive points is 0.055 m. This leads to an azimuth compression gain of at least 46 dB. The transmission duty cycle is roughly 7 %, compatible with ISM requirements.

The azimuth image LPF changes the target resolution from 0.5 × 15 m^2 to 15 × 15 m^2, a

Designation	Value	Units
TX Power	20	dBm
TX Antenna Gain	30	dB
Forward Free Space Losses (@ 45 km)	-141	dB
Radar Projected Cross Section	45	dB
Minimum Distributed Target Reflectivity	-25	dB
Backward Free Space Losses	-141	dB
RX Antenna Gain	30	dB
Diverse System Losses	-5	dB
Total	-187	dBm
Noise	-105	dBm
SNR	-82	dB
Range Compression Gain	28	dB
Azimuth Compression Gain	46	dB
Total	-8	dB
Azimuth Image LPF (30×)	15	dB
Link Margin	7	dB

Table 3.1: Link budget for SARIA

30× difference in azimuth, providing an additional SNR gain of 16 dB.

With all this, we find that the link budget is closed with a 7 dB margin.

3.1.2 Requirements for SARSYS

Given the SARIA context, we are now in position to define quantitative requirements for SARSYS. We'll use the word *shall* for hard requirements, and *should* for goals that are a nice-to-have but that are not a must. Also, by "typical system load" we mean the approximate system load that the OBC will have during the launch campaign, while running the background services for telemetry, tracking, and command (TT&C), navigation data handling, etc.

R1 SARSYS shall generate SAR images from raw data within 1min, when running on SARIA's OBC with typical system load.

R2 Under typical system load, SARSYS shall lose no samples when receiving data (ie. no overflows shall occur).

R3 Under typical system load, SARSYS shall be able to transmit samples without underflows.

R4 SARSYS shall be able to transmit and receive samples in parallel (ie. full duplex),

for the entire duration of a SAR session.

R5 SARSYS shall never overwrite or delete previously captured or processed data of any kind.

R6 SARSYS shall continuously capture navigation data.

R7 SARSYS shall be able to extract navigation data for a given time range in the past, without conflicting with the navigation data capture process.

R8 SARSYS shall be able to synchronize the different data sources without human intervention.

R9 SARSYS shall detect transmissions without human intervention, and extract the respective echoes samples coherently.

R10 SARSYS shall generate geocoded images.

G1 SARSYS should be able to heuristically choose the imaging region.

G2 SARSYS should be able to use multilook.

G3 When using multilook, SARSYS should be able to perform MoCo withing each single look, instead of globally.

G4 SARSYS should be able to perform SAR using multiple pulses within a single session.

G5 SARSYS should be able to perform SAR even with some considerable amount of squint.

G6 SARSYS should be able to acquire and process staggered SAR.

To quantify an *upper bound* for the processing time (req. R1), we thought that it being the same as the time required to capture a session would be a good compromise between the amount of images captured and the amount of images processed. We probably won't process all sessions that are captured, so, when well optimized, this should allow for some sessions to be processed without taking up too much useful capture time. Besides, as said before, as the gondola rotates, we'll have some periods of time when capturing data isn't useful due to the amount of squint, so we may dedicate them for processing previously captured sessions.

The reason to want to use multilook on this experiment (goals G2 and G3) is that, while the gondola has a relatively stable velocity and orientation, 1min ends up being a relatively long time for the gondola to deviate from a best fit line, or to start without squint but end up with a moderate amount of squint. So, the idea is to try to perform shorter looks in which each one is motion compensated, processed, and geocoded in isolation, and only after are they merged together (incoherently) into a final image, rotating and scaling the looks according to their coordinates.

For this stitching to work, we must also define a physical region that we want to be imaged in the end. With goal G1, this happens automatically, with just some general input about how big the image should be, that can be the same for every session. Without this automatic heuristic, the user would have to manually define the coordinates of each corner of the final image before processing, which is slow, tiresome, and error-prone, as

it must be done differently for each session, and is everything we don't want to have during a mission where time is of the essence.

Goal G4 is not effectively needed for the SARIA launch campaign, but is a nice-to-have in case we want to have preliminar tests from an airplane, where the range distances are much shorter. This is further explained in section 3.4.1.

By allowing the processing sessions with larger squints, goal G5 extends the variety of images that may be generated live during the launch campaign.

Finally, by supporting staggered SAR out of the box (goal G6), the applicability of SARSYS is extended, so it may be more easily reused in a variety of situations.

3.2 SARSYS architecture overview

Now that we have defined all the requirements associated with the system under study, we must define a suitable architecture that ensures those requirements are met. The proposed system architecture is present in fig. 3.4, which follows the same color coding as the previous functional diagram, and adds 3 new types of blocks: the ones with sidebars represent processes running on the OBC, the diamond blocks are interfaces between the OBC and its peripherals, and the cylindrical one represents a database.

Out of the processes shown in the diagram, the `capture` and `processor` are the only ephemeral ones, as they are ran on-demand when we want to acquire or process a SAR session, respectively. The others should run continuously, in background.

As seen in our motivation and requirements (eg. R5), preventing data loss of any kind is paramount for us. That is why we defined that navigation data coming from either source (GPS or compass) should be, first and foremost, dumped into a new file even before any kind of parsing occurs. In the case there are errors handling the data on launch day (eg. synchronization or parsing errors), this raw unaltered byte stream dump should allow us to process it offline and work around the errors.

After dumping the raw data stream, we are now free to try to parse it and format it into something that is easily usable by the SAR processing software. For this, we need to have some kind of formatted file that allows for concurrent reading and writing (ie. both the GPS and the compass handlers writing incoming data and the processing software fetching it). This may be implemented from scratch using OS synchronization primitives, but we opted for using SQLite, which is a simple to integrate C library that has been battle-tested over time in billions of devices [34] and has a very significant test suite that ensures its robustness. Using Structured Query Language (SQL) commands to access this data is actually very useful as we may now specify time ranges when fetching data from the database in order to obtain just the data that is relevant to each SAR session.

Also, we should note that this isn't like other commercial databases (such as MySQL or PostgreSQL) which require having a daemon running in background, permanently consuming some resources and requiring extra communication between a process and the daemon just to fetch data – SQLite just happens to be a glorified formatted file library,

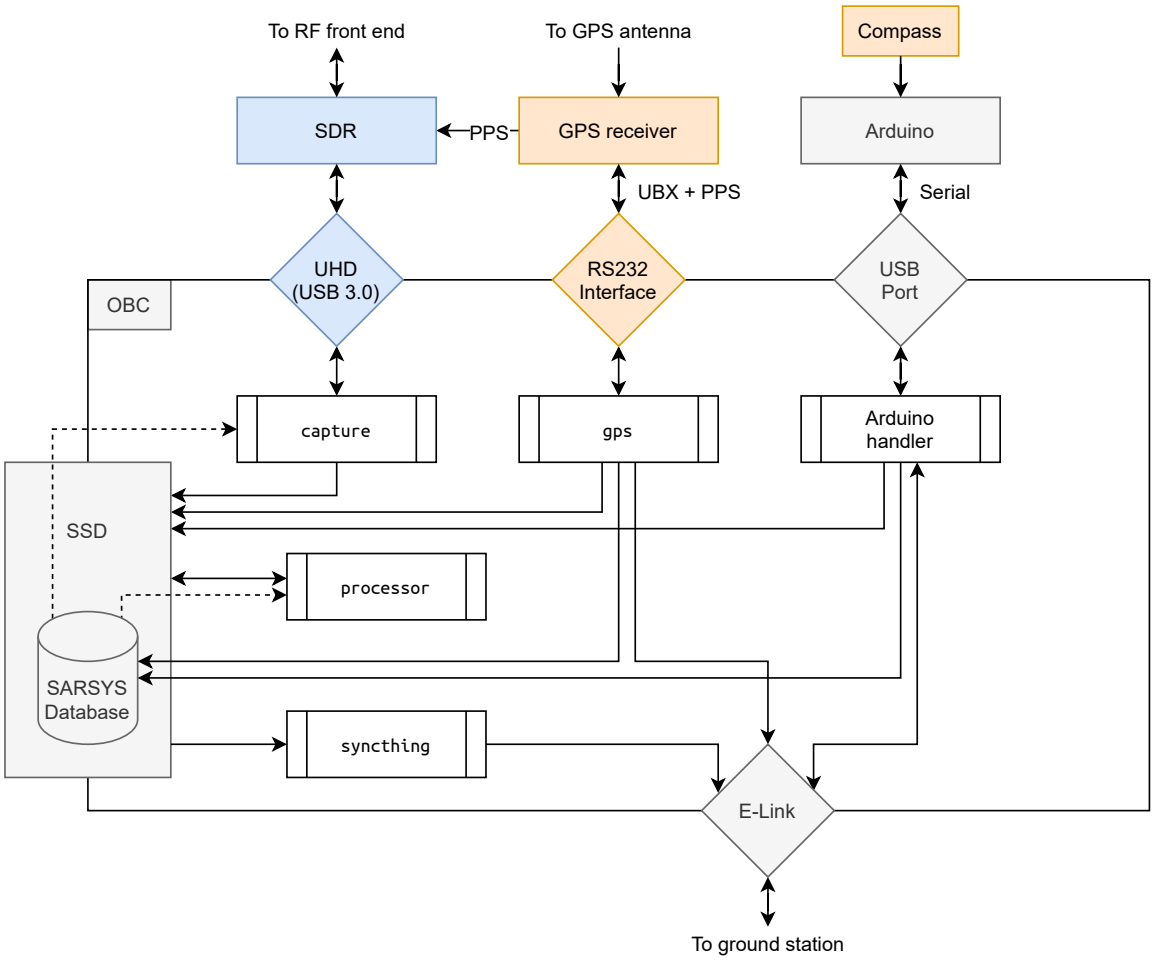

Figure 3.4: Functional representation of the different components of SARSYS.

that supports simultaneous reading and writing, and in which the access happens to be done via SQL commands. Besides, interaction with SQLite is far from being in a performance critical path, so the tiny performance impact of using a SQL engine is outweighed by the advantage of being able to perform queries for specific ranges in time while keeping all writes and reads consistent. It also comes with out of the box support for gracefully handling and being robust to application and OS-level crashes and power cycles during transactions.

More information about the navigation data handlers, the database format, and the interaction with the rest of the SARSYS software is present in section 3.3.

The SAR capture software, having to interact with an Ettus SDR, uses the official application programming interface (API) and driver for the purpose, provided Ettus Research, the USRP hardware driver (UHD). Due to the relatively high sampling rate of 12.5 MHz that we'll use in SARIA (section 3.4.1), the interface must be done over USB 3.0 (as opposed to 2.0, which would be too slow and result in over/underflows). See section 3.4.3 for more information.

The processing software uses the data captured from all sources and outputs a final image which is saved to the solid state drive (SSD). The processing algorithm is described

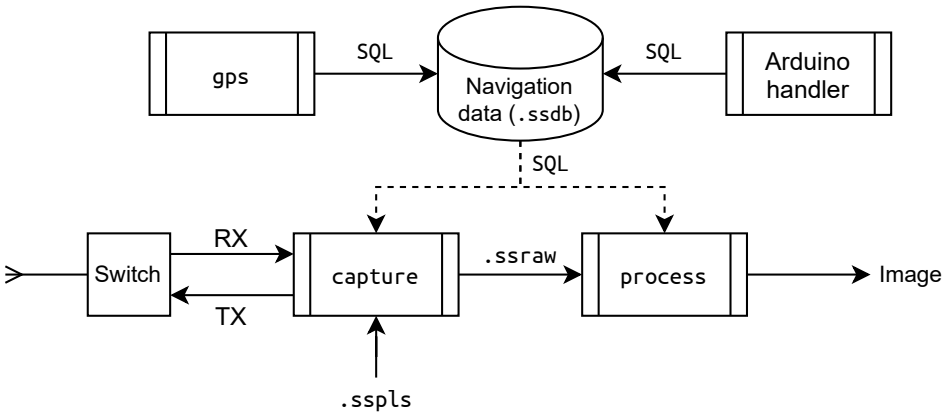

Figure 3.5: Data flow between the different components of SARSYS.

in depth in section 3.5.

This image may be placed in a pre-defined folder which is tracked by the *Syncthing* software [35]. This utility then proceeds to transfer the new images to the ground station via Esrange Airborne Data Link (E-Link), while ensuring the bit rate is within the stipulated limit.

Also note that both navigation data handlers also send data directly to the ground station, as it is needed for TT&C.

3.2.1 Data flow

The following file formats where defined for this system:

SARSYS Pulses (.sspls) This file contains the definition of the pulses that are to be transmitted at the PRF when capturing a SAR session. It may contain just one or multiple pulses, as per goal G4. When it contains multiple pulses, the capture software will, by default, transmit each one in order, and return to the first one when there are no pulses left. The format rationale and specification is present in section 3.4.2.

SARSYS Raw (.ssraw) This file contains the raw data of a single SAR session, along with all the associated metadata. The format rationale and specification is present in section 3.4.4.

SARSYS Config (.sscfg) This file configures the behavior of several steps in the SAR processing pipeline. It follows the simple INI file format [36].

SARSYS Database (.ssdb) This file is the SQLite database that is mostly used to store navigation data. Its tables are described in section 3.3.

In fig. 3.5, we may see how these files and programs interact together to generate an image.

All binary file formats that have been defined by us start with an unique, file type identifier 32-bit magic number (also called a file signature). These magic numbers are

used in the developed software in order to ensure we are loading the correct files at the right moments.

3.3 Navigation data acquisition & handling

In order to integrate the navigation data into our system, we had to develop a handler that interacts with the chosen GPS receiver. Most, if not all, GPS receivers support sending the data in the NMEA format; however, if available, the proprietary format of each manufacturer usually provides more information and is more flexible. In our case, as we are using a u-blox TIM-LF [37], we opted for the proprietary UBX protocol [38]. Besides the referred configuration and information flexibility of using a proprietary protocol, it is also binary, which makes it more efficient to both transmit and easier to parse than NMEA – as opposed to enconding the messages in ASCII, the binary representation allows for a much more compact representation so, for the same information and baud rate, the binary packets are transmitted faster, and may be directly interpreted by the CPU without any conversion at all.

In order to guarantee that req. R5 is met, when the `gps` process starts it *exclusively* creates (using the Linux flag `O_EXCL` to ensure the file didn't exist before) a file with the current date as name – this file serves as the dump for the received byte stream. Note that the OBC is not connected to the internet during the mission, so we also considered the case when the time was reset, during which a file with the "current" date may already exist; when this happens, we'll just retry to create another file but this time with an increasing number appended to the end of the file name.

While we receive and store other types of UBX packets that may be useful when performing a post-mortem or diagnosing a problem, for processing SAR we are mostly interested in 3 specific types: `NAV-POSLLH`, `NAV-VELNED`, and `NAV-TIMEUTC`, which contain, respectively, position, velocity, and time information. When receiving these packets, we accumulate the information in a buffer and, when complete, it is sent to the database.

The SARSYS database has two navigation-related tables, named after both sources of the data, `gps` and `compass`. The two tables have the columns `secs` and `nanos`, containing the sampling time of each row, as their *primary key*.

For each GPS epoch, the `gps` table stores the columns `lat`, `lon`, `alt`, `altmsl`, `vele`, `veln`, and `velu`, corresponding to the position (latitude, longitude and altitude – both ellipsoidal and orthometric) and velocity components (in the right-handed east north up (ENU) coordinate system). All columns are stored as 32-bit signed integers with the appropriate scaling (1×10^7 for latitude/longitude in degrees, 1×10^3 for altitude in meters, and 1×10^2 for velocity in meters per second). This storage format matches the one that is utilized in the UBX protocol, so it ensures data is preserved *without loss of precision*.

We store both altitude types in order to be more flexible configuration-wise. What we are actually interested in is the altitude relative to the ground, and for that we need to use a digital elevation model (DEM) to obtain the altitude of the terrain, which we can then subtract from the platform altitude to obtain the intended value. Having stored both altitude values, we may choose the one that uses the same reference as the DEM,

without needing to convert.

The `compass` table currently only stores the platform heading in degrees, also as a signed integer, scaled by 1×10^5. The implementation of the compass handler was actually the responsibility of another member of the team, so we only had to define the interfaces between their program and SARSYS.

3.3.1 Synchronizing time

Up until now, we have assumed that all components are synchronized between them. However, without a specific effort to do so, it would actually be very difficult to get the samples of the different sources to match up in time, and we probably would have to do it manually for each processed session, which is far from ideal.

However, the use of a GPS receiver in SARSYS makes our life easier. Due to the way GPS works, its receivers are actually a *very* good reference of time. That is why, besides sending the current time in a packet through the serial port, GPS receivers usually have a pulse per second (PPS) signal output. This pulse is, like the name says, sent every time a second transition occurs, but with an astounding precision, in some cases accurate to better than 1 ns!

As we are receiving GPS data through the OBC RS-232 serial port, we may actually use one of the extra pins as a PPS input. In Linux, we may do that by using the PPS API [39, 40]; as soon as the PPS line discipline has been attached to the serial port, a new *char device* named `/dev/pps...` appears, and we may send the reference signal through the data (DCD) pin, which should now be detected by the Linux kernel.

To effectively synchronize the OBC system time with the GPS time, we use the Network Time Protocol (NTP), more specifically, the NTPsec distribution. As we are handling the GPS data in a fully custom way, we also had to implement the NTP interface. To send the current UTC time to the NTP program, we use the shared memory (SHM) reference clock driver [41] – our program uses the `mode = 1` alternative and memory barriers when writing to the SHM structure, to ensure NTP never receives corrupt values. NTP will start by using this information *exclusively* to slowly synchronize the clock.

When it considers the clock is *sufficiently* synchronized, it will start using the PPS clock discipline driver [42] to fine-tune the clock with extraordinary precision. We also had to measure and compensate the delay between the start of the GPS epoch and the moment it is delivered to NTP, otherwise both clock references would be classified as false tickers.

Having dealt with the OBC-GPS time synchronization, we now need to deal with synchronizing the other two sources, the SDR and the compass. For the compass measurements, nothing really needs to be done. As we are expecting the gondola to rotate slowly (up to 45 deg /min), having a sufficiently regular measurement (eg. every second) that is timestamped on arrival to the OBC with a delay of some milliseconds isn't critical.

However, providing a time reference to the SDR is critical and, when done right, even has an extra advantage. In comparison to the compass, where a measurement is unbuffered

and infrequent, the SDR is much more complex and has several layers of buffering: not only does the SDR have its own receive and transmit queues, but the software stack on the OBC also has more buffering, to minimize overhead, such that it transfers the high-sampling rate data in large blocks. With this multi-layer queuing, it becomes very hard to estimate when an event actually occurred. That is why the SDR itself has an internal real-time clock that is used for timestamping data, and whose value is returned as metadata associated with the first sample of each block.

The problem now becomes ensuring that we set this SDR internal real-time clock precisely, as even a simple "set time" message also passes through those OS and SDR buffers. The answer is, once again, to use the precise PPS signal returned by the GPS as a global time reference. In general, in order to provide a PPS signal to Ettus's SDRs, we *also* need to provide a precise (and in-phase) 10 MHz reference that is used to derive their other clocks. However, in the case of a few specific models, such as the one we are using, the Ettus B200mini, we may actually *only* provide *either* the 10 MHz reference *or* the PPS. When we provide an external 10 MHz reference, we are only providing a frequency reference; when providing an external PPS, it is actually used *both* as a time reference, as expected, and also as a frequency reference, because the SDR will derive all their internal clocks using a phase-locked loop (PLL) based on this PPS. This is why we referred that there was actually an extra advantage of providing a stable time reference to the SDR in our case, as the frequency stability of the PLL has been measured to be accurate up to 10 ppb, as long as the PPS is precise and the SDR is in a well controlled environment without too much temperature variation [43].

These are great news for SAR processing, because SAR requires *coherent* data to work, which implies having very good phase stability between transmission and reception.

So, to set the SDR time, we just wait until the computer time is far from second transitions (to ensure that all components have received the latest PPS edge), and then use the function `uhd_usrp_set_time_next_pps(...)` that updates the SDR internal registers and updates the time in the *next* pulse.

3.4 SAR signal generation & acquisition

In this section, we present our SAR acquisition program, called `capture`. This program shall be able to transmit radar pulses while receiving continuously, outputting a file containing the raw data and its associated metadata in the end. Before describing the program itself, we'll actually start by describing the pulse that shall be used in SARIA.

3.4.1 Pulse definition for SARIA

In fig. 3.6, we find a plot of the pulse that is to be used in the SARIA experiment. The signal is decomposed into two parts:

- The first part is a dummy trigger for activating the TX path of the RF front end. As the antenna is shared between TX and RX, we deactivate the path that is not

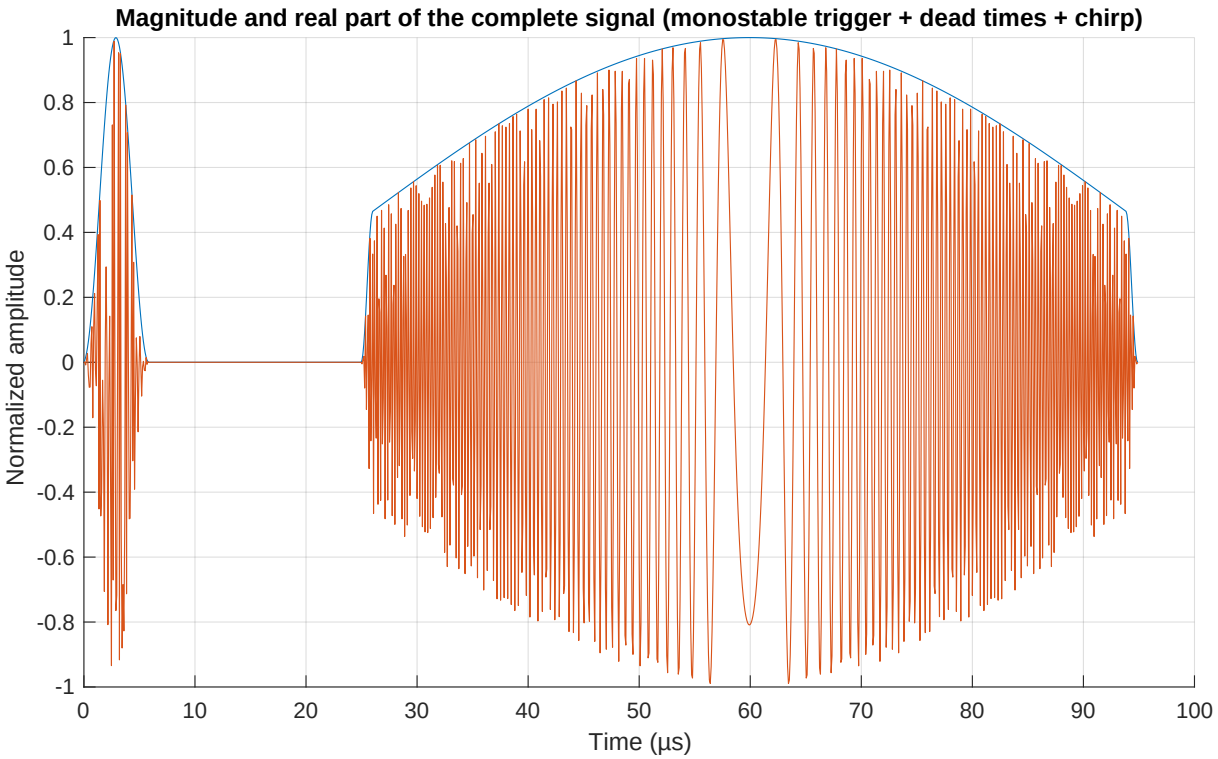

Figure 3.6: Magnitude and real part of the signal for SARIA.
Has a trigger (first part), chirp (second part), and dead times.

being used in order to reduce noise and prevent saturation (or even damage) of the SDR.

By default, the active path is the RX one. When we want to transmit, a very short trigger is sent, which is electronically detected in the front end. After a small delay (to ensure we aren't transmitting the trigger anymore), the active path is switched to TX and, after a pre-defined duration that is just enough for the pulse to be transmitted, it reverts to the RX path.

- The second part is the pulse signal itself, that will be pulse-compressed as part of SAR processing. As seen in section 2.4, this pulse must have very good auto-correlation characteristics, and must also be *as uncorrelated as possible* with the trigger to prevent the appearance of fake echoes. For SARIA, as there is no need to use a variety of different pulses, we opted for a classic chirp signal.

  When performing pulse compression, this second part is what shall be used as a reference signal; the trigger part is just an implementation detail related to the activation of the TX path.

The chirp has a $BW_{chirp} = 10\,\text{MHz}$ and a duration of $70\,\mu\text{s}$. The sampling rate in SARIA will be of 12.5 MHz.

The dead time between the trigger and the chirp ensures that the TX signal path is completely initialized when the chirp starts being sent. In order to smoothen the transition between the dead times and the signals, a Tukey window with a coefficient of 0.03 was

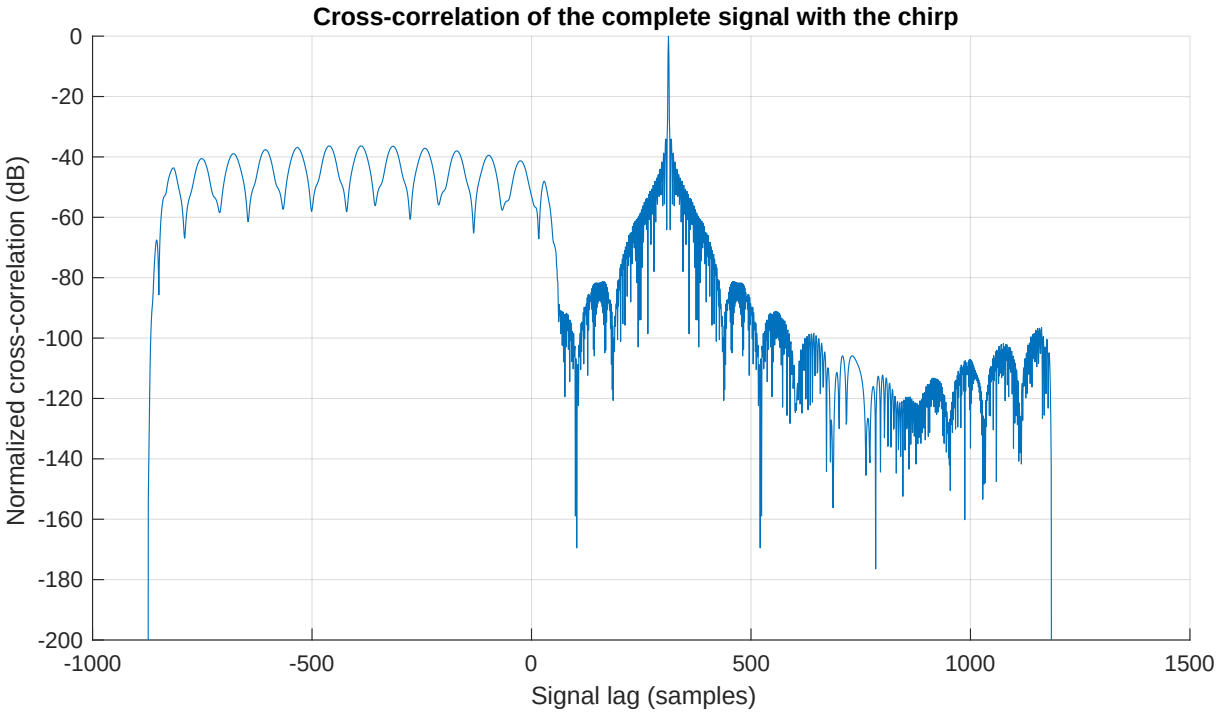

Figure 3.7: Cross correlation between the complete signal and the chirp

applied.

For a more precise correlation peak, a Kaiser window with a coefficient of around 2.0 is applied to the chirp. This helps reduce the side lobes of the correlated signal, making the peaks stand out even more over the interference that unwanted signals may cause. As discussed in section 2.4, the use of windows in general broadens the compressed peak a little bit.

To ensure the chirp is uncorrelated with the trigger signal, we chose the trigger to be the chirp itself, *but* conjugated *and* decimated. An additional Hanning window is applied to the trigger in order to further smoothen it and reduce side lobes.

In fig. 3.7, we can see that the correlation of the complete signal with the chirp is as expected. There is a clear peak indicating where the chirp is, and everything else is at least 38 dB below the peak. The side lobes at the beginning are due to the presence of the trigger. As they are 38 dB below the peak, we may consider the trigger sufficiently uncorrelated with the chirp.

Polyphase pulses

The chirps have one large disadvantage: for the same bandwidth, there are only two chirps, the up-chirp and the down-chirp. In some cases, like in the SARIA experiment, this doesn't matter at all, because the echoes are never overlapping. However, when we enter the domain of wide-swath SAR, the echoes become so large that we have to allow overlapping between them in order to be able to have an high-enough PRF for the captured Doppler bandwidth.

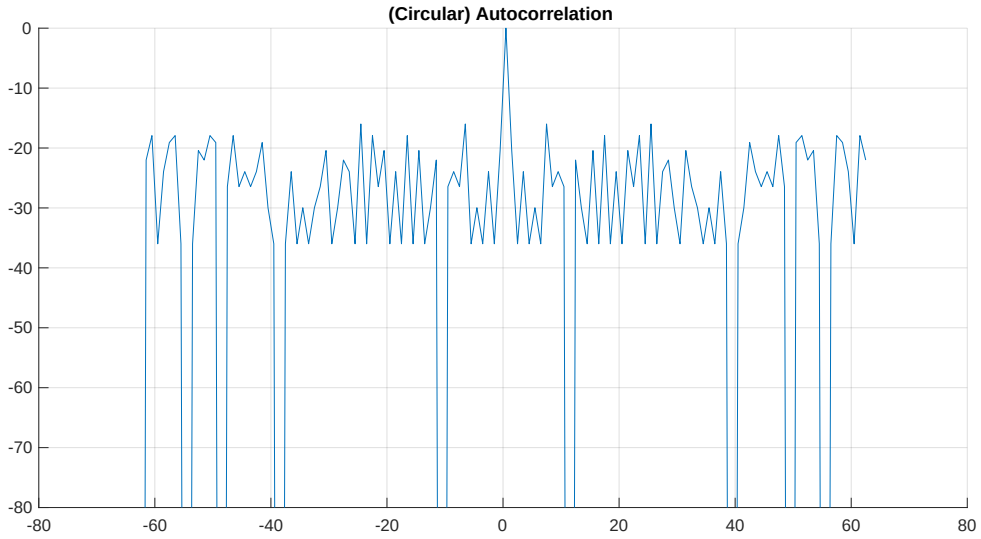

Figure 3.8: Circular autocorrelation of a polyphase pulse.

Staggered SAR solves one part of the problem, which is the fact that the transmissions themselves, when regular, will create blind ranges.

However, the overlapping echoes must be solved another way; that's what polyphase pulses are used for. Polyphase pulses are usually based on pseudo-noise (PN) sequences that have cross-correlation guarantees. That way, if the echoes of pulse i overlap with the echoes of pulse $i+1$, that is not a problem because they are "mostly" uncorrelated.

Besides, they have another feature which is that their range behavior is actually improved when the data is processed in azimuth. For example, in fig. 3.8 shows the autocorrelation of a single polyphase pulse of length 127, based on Gold sequences of order 7. It is not great, as it has relatively low peak isolation.

However, when the azimuth processing occurs, all the different polyphase pulses of a set are *coherently* processed, and that is just like adding them together in phase. The result of calculating the mean of all autocorrelations is seen in fig. 3.9. Notice how the peak, besides being very sharp, is also now completely above the noise floor.

Using these polyphase signals also brings another advantage. Sometimes, when the same signal is sent again and again, the transmitter has some non-linear response to that and causes a DC component that has to be filtered out from the impulse response. When using these signals, as they are all different and "random", no DC component is created, so we may use the entire impulse response – note that the maximum magnitude of the impulse response is precisely DC, so being able to use it may actually improve the image quality significantly.

We have designed a set of 127 of these pulses, each one of length 127 (coming from the Gold sequences of order 7), to use in a future experiment when we test SARSYS from an airplane. As the distances are much shorter, we needed a smaller signal; however, only sending a smaller chirp reduces the SNR a lot. With the polyphase pulses, we gain the SNR back after the coherent azimuth processing.

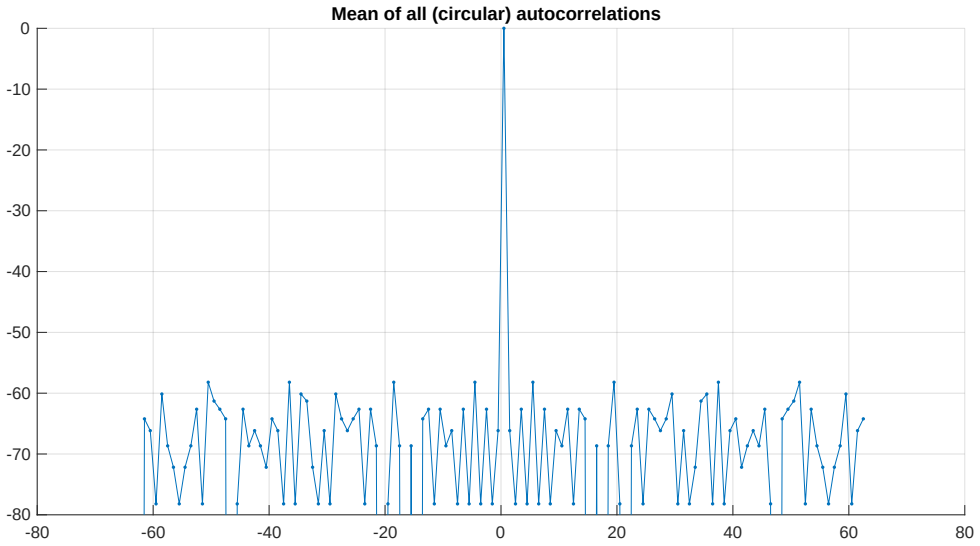

Figure 3.9: Mean of the circular autocorrelations of the set of polyphase pulses.

3.4.2 SARSYS Pulses file format

As seen in the data flow diagram (fig. 3.5), besides the global configuration that is passed to every program, the `capture` program also reads a SARSYS Pulses file (`.sspls`), containing the pulses that shall be transmitted. The format of this file was kept very simple, and is shown in fig. 3.10.

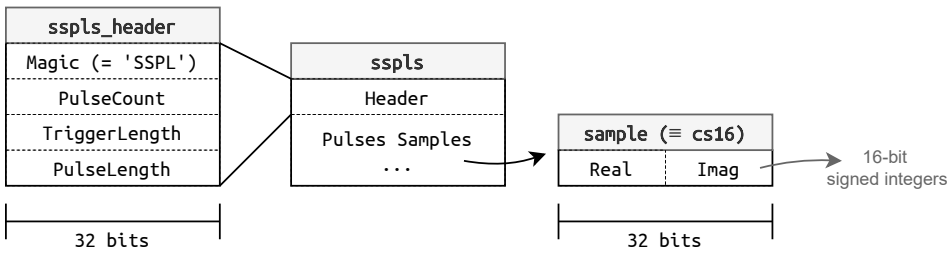

Figure 3.10: SARSYS Pulses (`.sspls`) file format.

In the file, each pulse has a *total* length of $\texttt{TotalLength} = \texttt{TriggerLength} + \texttt{PulseLength}$. This total length is the number of samples shall be transmitted, containing both the trigger and the pulse per se. When performing the auto-correlation, only the last `PulseLength` samples shall be used.

The total number of samples that must be stored in a well-formed SARSYS Pulses file must be $\texttt{PulseCount} \cdot \texttt{TotalLength}$: the first `TotalLength` samples are for the first pulse (ie. index 0), the following `TotalLength` samples are for the second one (ie. index 1), and so on.

The chosen format for storing the samples themselves is as two 16-bit signed integers (also depicted in the diagram), one for in-phase and the other for quadrature [19]. This is precisely the format that is used by the SDR over-the-wire, for both transmission and reception of samples, so no precision is lost, and we save half of the space in comparison

to the alternative of storing them as 2 32-bit floats values. Internally, we call this format `cs16` (for Complex *S*igned *16*-bit).

3.4.3 SARSYS acquisition software

With this, we are now able to describe the SARSYS `capture` software.

When starting up, it loads all pulses from the provided file, and also opens and configures UHD handles for the SDR. When a PPS is available, time synchronization is done by following the method established in section 3.3.1. However, we also handle the case where a PPS may be missing (eg. due to an connection failure) so, as a *fallback*, we let the SDR use the internal frequency reference and only provide the current time in a best-effort way, without doing it in a second transition.

When processing SAR, we need to know very precisely the time and location (in the sample stream) of each pulse transmission, and the transmissions should be as close as possible to an uniform sampling in azimuth (otherwise we lose the linear time-invariant (LTI) property that is assumed in the processing). When the SDR is directly integrated with the system, we may be able to perform this acquisition by receiving samples by default and interrupting it when we want to transmit, returning back to the reception mode when the transmission is done. However, this is infeasible to do when the SDR is a peripheral component: due to the communication and buffering delays, we have no guarantees about when a transmission occurs.

The solution we found to this problem was to perform full-duplex transmission *at all times*, as specified in requirement R4. Even if the RF frontend had complete isolation between TX and RX (which it doesn't), the SDR itself also has some cross-talk between the ports. This means that, when we are receiving continuously, we are able to find, in the RX stream, when the transmissions occur (and they are found in a very precise manner due to the matched filtering). As the RX sample blocks are timestamped, we may extract the time and location of all TXs. This explanation only provides the rationale behind the requirement R4; this TX search process is further explained in section 3.5.1.

In order to transmit and receive in full-duplex, we divided the program into two worker threads, one for each endpoint. The TX worker thread transmits the pulses defined in the SARSYS Pulses file (in sequential order, by default), at the pulse repetition frequency. It may actually randomly shift the location of the transmissions by multiples of the length of each pulse to comply with goal G6 of having staggered SAR support out of the box. This behavior is, of course, configurable. Between pulses, the TX worker sends the appropriate number of null samples to the SDR.

The RX worker thread starts by calculating how many samples it shall receive (taking into account an estimate of the target range to ensure all echoes are captured), and allocates a large enough block of memory capable of containing all of them. This is acceptable in the case of SARIA, as the session duration is, *at most*, less than 1min30s and that would "only" need $\Delta t \cdot f_s \cdot 4\,\mathrm{B/samp} = 90 \cdot 12.5 \times 10^6 \cdot 4\,\mathrm{B} \approx 4.2\,\mathrm{GiB}$, which is within our memory budget. This also has the advantage of reducing disk activity (and thus, less I/O pressure), as we are writing everything to memory up until the end – this improves our chances of not having any overflows or underflows on the SDR (reqs. R2

and R3). For longer duration experiments, where the memory requirements would be prohibitively expensive, we could trivially extend this current system in order to have several receive buffers that would be used sequentially and that would be written to disk by another lower-priority thread in the background, so that they could be reused – however, it should be thoroughly tested to ensure we are able to transmit and receive all samples without overflowing.

The RX worker thread also stores metadata relative to each arriving block of samples, such as the timestamp of its first sample and how many missed samples there were between the previous and this block (inferred by the timestamps).

When both threads have finished their assigned job, the process starts writing the output SARSYS Raw file. Its format is fully specified in section 3.4.4. Besides the expected information about the received samples, the format also supports having navigation data. Therefore, if configured to do so, this `capture` program waits for some extra time and then tries to fetch navigation data associated with the captured session from the SARSYS database. If successful, the generated SARSYS Raw file becomes standalone, so it may be processed in isolation. In that case, we may, for example, transfer just this one `.ssraw` file to another computer, and it shall have everything needed to generate a final image.

The described threads require no synchronization primitives to work, as they have such distinct jobs that there is no contention anywhere. This is ideal when optimizing for performance, as those synchronization primitives are costly. The only type of synchronization present here is to start the transmission and reception of samples at the same time, but that is done at the SDR level, as it already provides scheduling these TX and RX streams for a specific time in the future. By default, we schedule it to 3 s into the future so that everything is guaranteed to be set-up when the deadline occurs, and also to ensure the PLL has had some time to stabilize.

3.4.4 SARSYS Raw file format

In fig. 3.11 we have a diagram representing the general structure of this file.

There were three main points that influenced its design:

- During the SARIA mission, these files will have at least 3 GiB. Therefore we want to minimize both the number of syscalls made to extract data from the file and the number of times we have to re-read the same data multiple times.
- We also have to gracefully support the fact that samples might be missing! This means that in order to obtain sample with cumulative index j given that we know sample i is at file offset o_i, it is not just a question of doing $o_j = o_i + (j - i)$, because the cumulative indices count *all* samples, but the file will only have stored the *received* samples, therefore this would be incorrect if any sample was missing in-between.
- The code for handling these files should be trivial. We want to ensure robustness of the developed software, so the easier it is to understand and implement, the easier it is to test.

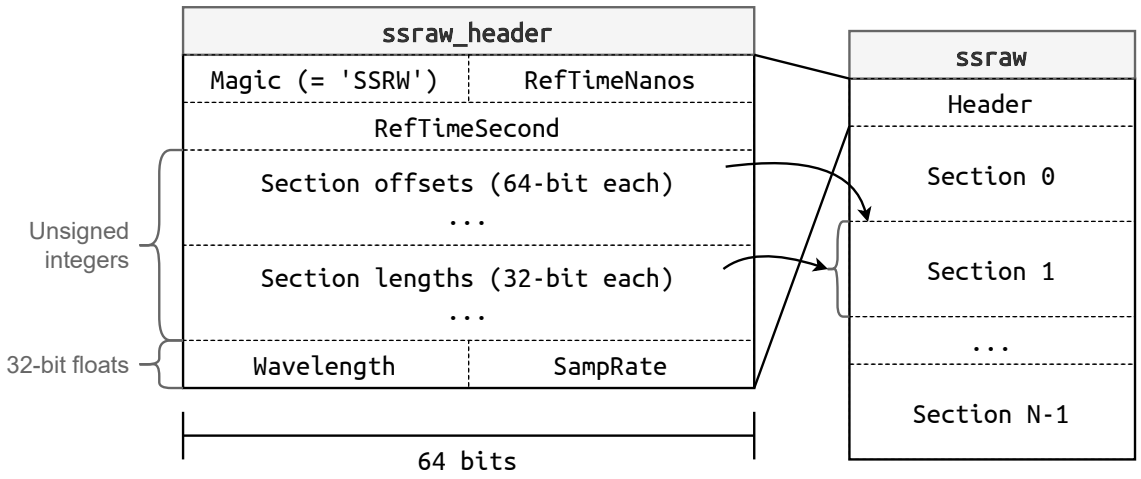

Figure 3.11: SARSYS Raw (`.ssraw`) file format.

After testing different formats, we found the ideal one would be with out-of-band information (ie. with section offsets & lengths in a header, decoupled from the data) – we ended up using this idea for the macro file structure and also for some specific sections. Each section has a base structure that is repeated for the number of times specified in the corresponding length attribute.

Some of the sections need to use timestamping; however, a full Unix timestamp with nanoseconds requires using at least 64 + 32 bits, so storing a full timestamp in every repeated structure would be wasteful. The solution we found was to have a full reference timestamp stored in the file header, and then the subsequent structures that need to refer to a point in time only store a signed 64-bit integer that corresponds to the nanoseconds elapsed since that reference time, this way saving 32 bits for every used timestamp.

We have defined the following sections:

TXs info This section is composed by several `ssraw_tx_infos` (see fig. 3.11), each one containing information about the a pulse transmission; specifically, which pulse was utilized (see goal G4) and how long the delay (in samples) until the next transmission (may vary between transmissions when doing staggered SAR, see goal G6).

Pulses samples This section is just a sequence of `cs16` samples (the native format used by the SDR over-the-wire, see section 3.4.2). All pulses are assumed to have the same length, specified in the header. The number of pulses and how long they are (assumed to be the same for all of them) are specified in the header. These pulses shall only contain the relevant part for performing pulse compression (see section 3.4.1).

RX blocks This section is composed by several `ssraw_rx_blocks` (see fig. 3.11), each one *describing* a block of samples returned by UHD. `FirstSampIdx` corresponds to the *cumulative* index of the first sample in this block, since the reception started – considering *all* samples that should have been received. On the other hand, `FirstSampOffset` corresponds to the sample *offset* in the file. These two values

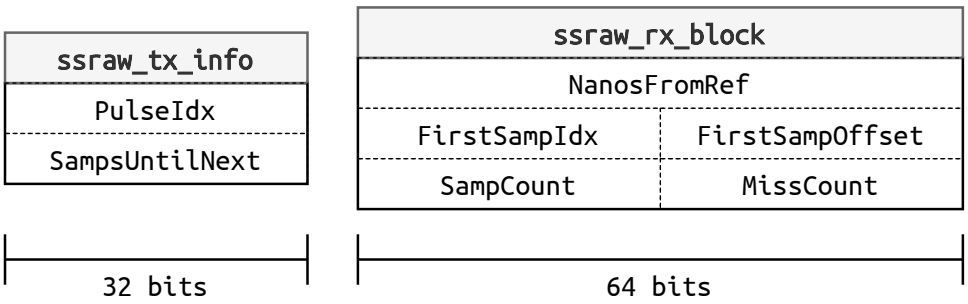

Figure 3.12: Format of the base structures for the TX info and RX block sections, respectively.

are only equal if, up until that sample, there were *no* missing samples. Finally, `SampCount` and `MissCount` refer to how many samples there are in this block and how many are missing between this block and the *next*, respectively.

RX samples This section is also just a sequence of `cs16` samples. The RX blocks section is required to correctly parse and extract data from this section. The sample offset in the RX blocks is an offset relative to the first sample of this section.

GPS and compass data These sections are both very similar to the format specified for the database. The only difference is the timestamp which, in this format, is specified as a nanoseconds delta from the reference point in time. These are the only two optional sections.

While, at first glance, it may seem like this file format is complicated to handle, it is actually very simple! When opening the file, we immediately read the header (and check that it is what we expect via the magic number), and also the entire TX and RX block metadata. With this, it is now trivial to extract some samples of specific blocks or to extract the pulses for pulse compression.

The only "challenge" is extracting large regions of samples due to the possibility of having missing samples between blocks. However, this isn't a problem with the file format itself, but with the underlying data it is providing. With the help of some abstractions, it actually becomes a very easy and efficient file format to use.

3.5 The SARSYS processor

Finally, we have arrived at the heart of SARSYS – the SAR processing software.

As you may see in the processor overview (fig. 3.13), we have opted for a multilook design. This multilook technique uses a novel algorithm which takes ideas from both methods described in section 2.10. The single looks are segmented directly from the azimuth in the time domain. By default, we also perform MoCo locally in each single look instead of globally.

Over the following subsections we'll discuss the ideas and some implementation details behind each one of these steps.

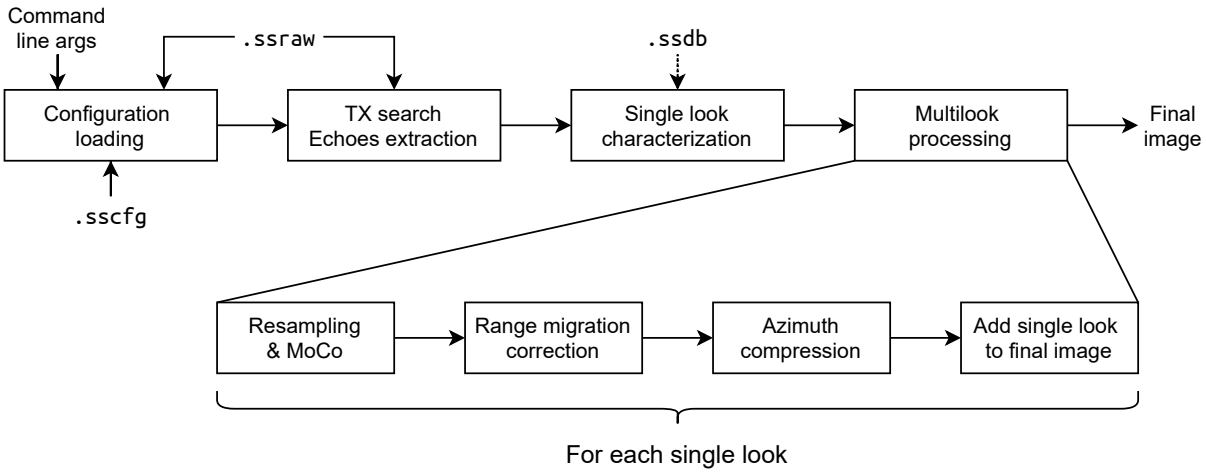

Figure 3.13: Overview of the different steps taken by the SARSYS processor.

3.5.1 Coherent echoes extraction

The first step taken by our processsing software is to find, in the RX sample stream, where the transmissions are, and extract the samples that should contain respective echoes.

As described in section 3.4.3, the internal SDR crosstalk and the finite isolation between TX and RX of the RF front end results in the RX sample stream containing a very strong image of the transmitted pulses. If we find these images in the raw sample stream, we know precisely when the pulses were effectively transmitted by the SDR, which is required to obtain a coherent extraction of the echoes (req. R9).

This step was implemented, in our software, as composed by two sub-steps: the search for transmissions, and the echoes extraction. Both of them have to perform pulse compression on the sample stream; however, while for the echoes extraction we want to perform pulse compression of a specific range of samples, given that we already found their associated transmission, during the TX search we are not sure where the pulses may be, so we may have to wander in order to find the correct location.

It would be impractical and *very* inefficient to perform pulse compression of the *entire* RX sample stream just to find the transmissions and extract the echoes in some select locations. The way we implemented the search was, therefore, by taking advantage of the block convolution algorithms [44]. These algorithms are designed to efficiently perform discrete convolution of a very long (or of unknown length) signal with a FIR filter (which is what matched filtering is, see section 2.4).

While overlap-add is *asymptotically* and result-wise equivalent to overlap-save (also known as overlap-scrap), we opted for the latter because, in practice, it has to perform fewer additions. Let's consider we have a reference pulse of length L and a block length of B, so that $B > L$. If we perform the *circular* correlation, using the DFT, between a block of B RX samples and the reference pulse (zero-padded to have the same length) – similar to what was described in section 2.4 – we'll obtain a result in which the $B - (L - 1)$ first samples are linearly correlated, and the last $L - 1$ samples are the result of circular

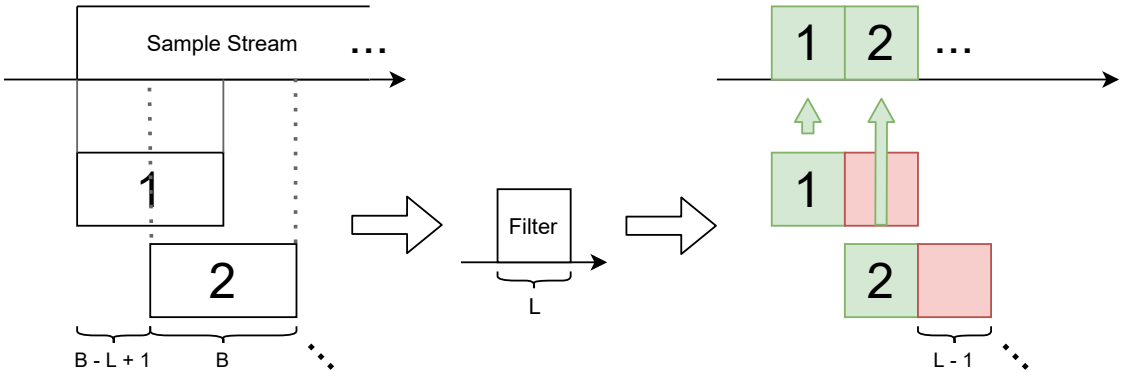

Figure 3.14: Simplified representation of the canonical overlap-save block algorithm.

correlation. In the overlap-save algorithm, *only* the linearly correlated samples are kept. To continue the process, we advance the signal sample stream by $B - (L - 1)$ (*not* B!) and repeat the process, *appending* the new linearly correlated samples to the output buffer. See fig. 3.14 for a diagram representing the described algorithm.

Now, how can we use this for performing the TX search efficiently? Instead of building a correlation output buffer, which we are not interested in, we have integrated the block correlation algorithm directly into the search. For every block of linear correlation we have, we try to find the TX there. If it is not found, we continue to the next block; otherwise, we save the location information in a table, and move on to search for the next.

We actually have a very configurable and efficient correlation peak search implementation. It allows defining a minimum magnitude value the peaks must have and how large the search context is – we only consider it to be a peak if it is *both* greater than the minimum and *the largest* within its own context window (this context window is handled even in-between overlap-save blocks). When a peak is found, it starts searching for the next after an offset, by taking into account the `SampsUntilNext` information from the SARSYS Raw file (section 3.4.4), in order to minimize performing pulse compression of regions where there are no TXs to be found (this automatically handles staggering, goal G6). We also allow the search to "give up" if an expected peak isn't found within a determined number of samples, but there is extra leeway when searching for the first TX, as that is the first synchronization point. Finally, to perform the pulse compression, the search uses the pulse specified by `PulseIdx` in the raw file, so that goal G4 is also achieved.

After building a table with the locations of all TXs that were found, we then extract the echoes coherently. In the configuration, we define the echoes offset, in samples, counting from the TX correlation peak, and also how many samples should be compressed for each one (see section 2.6) – if we want to have, in the end, M compressed samples for each range line, we actually have to extract $M + (L - 1)$ at a time, where L is the filter length, to prevent having a "fade out" transient in the end. It is then trivial to perform the pulse compression and add the result to the range lines table.

In the cases where transmissions occur during the reception of echoes (usually when we

are dealing with wide swaths and staggered SAR), the processor automatically clears out those regions before performing the pulse compression.

In both sub-steps, we are using power-of-2 lengths for the compression, as that ensures maximum one-dimensional FFT performance (as it uses the radix-2 Cooley-Tukey algorithm [45]).

- For the final range compression, we just calculate the next power-of-2 relative to the number of samples that were requested.
- The optimal block length for continuous overlap-save is given by the *first* power-of-2 B that makes the expression $\frac{2B}{2+\log_2 B} \geq L$ valid. However, if the "give-up length" is too small, this block length may be unnecessarily large.

  So, for the TX search algorithm, we use *either* the next power-of-2 relative to the "give-up length", or the optimal length for performing overlap-save for the chosen filter size, whichever is smallest.

When the `processor` initializes, it loads the pulses from the `.ssraw` file and calculates the two required DFTs for each of them (one for the TX search, and the other for the range compression), conjugating them right away, and stores the results in a lookup table that is used by this step, preventing needless calculations.

3.5.2 Single look characterization

From this step on, we have to start looking at the received data in azimuth. More specifically, this step characterizes how many range lines belong to each single look, what is their reference line and centroid, and what are the geographical characteristics of the image that shall be generated by this look. This explanation assumes the default behavior of the processor, which is to perform MoCo locally in each single look.

The first part of the this step is to obtain navigation data for the moment of each transmission. As described in section 3.4.3, the raw navigation data may be obtained either directly by reading the information embedded in the `.ssraw`, if it exists, or it may fetched from the SARSYS database.

Performing distance and vector calculations with latitude/longitude isn't easy. To work around this, we define the position corresponding to the central navigation entry to be the reference position, and perform an equirectangular local projection of all entries. The navigation data is now converted from latitude/longitude/altitude to the coordinates in a local tangent plane (LTP) centered on the reference.

After this, to filter out noise from the position data, we pass it through a low-pass filter. We've had great results by using the elliptic infinite impulse response (IIR) filter design, with very low normalized pass-band edge frequency, such as 0.01 or 0.05. There are two details that need special attention:

- The elliptic filter must have odd order for it to maintain the DC component intact, otherwise everything will be shifted

- The phase delay must be accounted for, to ensure we are getting the correct low-pass data for the queried time. Furthermore, IIR filters don't even have linear phase delay, making it more difficult to compensate than linear-phase FIR filters.

  The solution in our case was to apply forward-backward filtering, which achieves zero-phase delay with any kind of filter by applying it in a non-causal way [17]. Our base filtering function implements the direct-form II transposed filter realization structure. Then, the filter is applied twice in a row, the first time like the usual, and the second time reversed, from the end to the beginning of the signal (also storing the result in the reverse order).

  We also take special care with the start and end transients, by extending the signal with a small reflected portion at both ends and by initializing the state registers according to Likhterov and Kopeika [46].

  This method achieves zero-phase delay, so we may now access the data just like we would without filtering.

Finally, we perform linear interpolation to obtain the navigation data associated with each transmission time.

With this, we may now start characterizing each single look. This will be further explained in section 3.5.3, but the multilook processing operates on a fixed number of range lines per single look. However, before doing that, a resampling occurs as part of the MoCo step, in order to obtain a better uniform sampling approximation in azimuth. This resampling depends on the navigation data, and it may merge the echoes of several transmissions into a single range line.

The problem with this is, if we don't know, a priori, how many *input* (ie. pre-resampling) range lines each single look will operate on, how do we calculate a reference line for MoCo? Put another way, the reference line should be a "best fit" to the data of each single look, but the data that each single look operates on depends on the resampling which, in turn, depends on the reference line, so we have a circular dependency.

Our solution to this is to have the user provide a configuration which tells the processor to calculate a reference line based on the first $\#TX_{\text{guess}}$ transmissions of each block, and then extrapolate linearly for the remaining transmissions that are still considered part of the single look by the resampling operation. By knowing the average speed of the platform and the PRF, and choosing a target azimuth delta, a rough estimate of $\#TX_{\text{guess}}$ is easy to derive: $\#TX_{\text{guess}} = \frac{\Delta y \cdot \text{PRF}}{v_p}$.

This may not work very well in cases were motion deviations only occur at the end of the block. In the cases where those are a cause of problems in the images, this method could trivially be extended to do several iterations of this proposed method, while considering, for the $i+1$ reference line, the number of TXs found for iteration i, until it converges (which should be in only a few iterations). In our tests until now, the simpler version hasn't had any problematic effect.

Having obtained the reference line, we project the TXs positions onto it up until we have filled all range lines. When two different TXs are projected onto the same range line slot,

the final value of that range line is the mean of their MoCo'ed echoes.

Finally, we also calculate the geocoding associated with this single look. First we obtain the coordinate axes relative to each look: one of them is the direction of the reference line, and the other is the direction where the antenna is pointing (so we need to take into account the compass data). Then, by knowing how long the largest azimuth impulse response is (from the antenna projection), we extract the look image azimuth size, and by knowing the echoes delay and duration, converting them to slant ranges, and projecting these ranges onto ground range (see section 2.7), we obtain the look image range offset from the centroid and size, respectively.

3.5.3 Multilook processing

This is the step that effectively performs the SAR-specific processing, and is the one that takes the longest to execute. We have divided this step into the smaller ones seen in the processor overview (fig. 3.13) so that the explanation is more manageable. Each single look block is processed independently until the very last sub-step.

Motion compensation

In SARSYS, we decided to only compensate non-uniform azimuth sampling and translational errors of the antenna. We also make use of the *narrow-beam approximation* [23], so we use consider range axis of the look image as our range direction.

In the general case, and mainly when we are closer to the target, the phase compensation applied in this step is *range-variant*. This means that the phase difference caused by the motion deviation depends on the range cell we are talking about. So, we first start by calculating a reference 3D vector for each range cell, which represents the direction from the radar platform to a target on the ground that is at the corresponding slant range.

Then, for each TX whose echoes are included in this single look:

1. We calculate the perpendicular vector from where the transmission was made to the ideal reference line (obtained during the look characterization, see section 3.5.2)
2. By assuming that we are we are at a large enough distance, we may consider that the echoes come in planar waves from the target. That way, the phase difference can be well-approximated by a linear function of the *projection* of the 3D distance vector calculated in the previous item onto the range cell direction. So, for each range cell, we calculate this projection d_{proj}, and the compensation we have to perform is to multiply the original sample coming from the range compression by $e^{-j\frac{4\pi}{\lambda}d_{\text{proj}}}$.
3. We then sum the compensated sample to the range line cell, also multiplicating by the inverse number of TXs that were merged into this cell (to obtain the mean value).

With this, we have been able to achieve satisfactory results, while using a simplified version of the methods usually described in the literature.

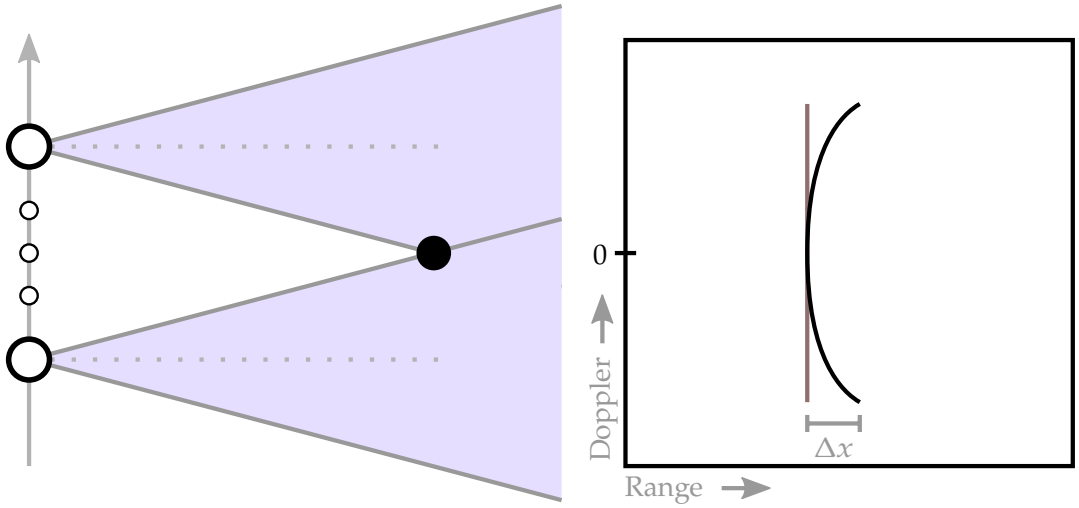

Figure 3.15: Simplified representation of range cell migration, without squint.

Range cell migration correction

When the echoes of a single target are range compressed and we visualize the resulting SAR data matrix, the trail left by the target is *not* a linear path that only varies in azimuth and stays constant in range. The path actually resembles a parabola as it varies in slant range (see section 2.6).

The cause for this is precisely what gives the signal its azimuth modulation. However, the system performs the azimuth compression for each range gate, so this "smile" distortion must be corrected before compression.

In the case of fig. 3.15, we already know how to find Δx (in eq. (2.12)), and that this correction is very easily done when in the range-Doppler domain (see section 2.7). So, the first thing we do in this step is to perform the azimuth FFT of all ranges, to make go into that domain. To correct the range migration itself, we now need to perform the reinterpolation of $R_0 + \Delta x \rightarrow R_0$.

Substituting the Δx in the expression, and replacing R_0 by any range R, we find that the mapping is given by

$$R' = \frac{R}{\sqrt{1 - \left(\frac{\lambda f_{\text{dop}}}{2}\right)^2}} = \frac{R}{\text{MF}} \rightarrow R$$

Given that $\text{MF} \leq 1$, this means that all we actually want to do is to compress the range axis by the migration factor MF. This kind of compression reinterpolation is very easily done with the help of the DFT. First of all, if we start with a signal of length N' and perform this mapping, we should end up with a compressed signal of length $N = \text{MF} \cdot N'$ – this means we have reduced the signal by $N_{\text{removed}} = N' - N = (1 - \text{MF}) \cdot N'$. Therefore, if after calculating the DFT, we remove the N_{removed} highest frequency bins (half negative, half

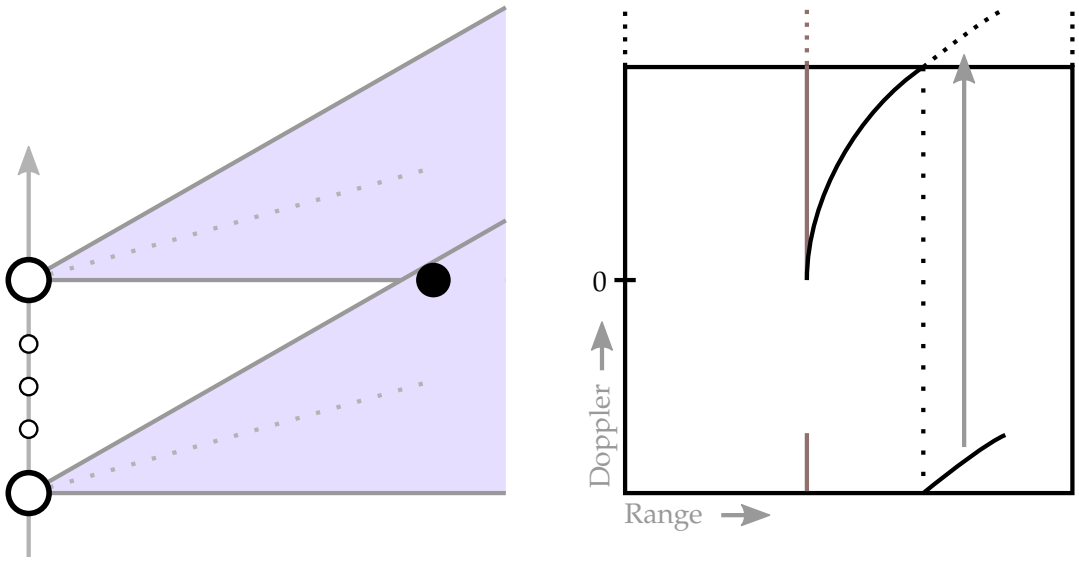

Figure 3.16: Simplified representation of range cell migration, *with* squint.

positive) and calculate a *smaller* IDFT of length N, then we obtain the precise resampling we wanted!

However, this is not yet the full story, as this only works when the transformation is fully linear; as we extract the echoes with a delay n_0, we actually have to perform an *affine* transformation! For that, we need to derive the full expression:

$$\begin{aligned} R &= \delta_s n_0 + \delta_s n = \mathrm{MF}\,(\delta_s n_0 + \delta_s n') \\ \Rightarrow n &= \underbrace{n_0\,(\mathrm{MF}-1)}_{\text{Affine}} + \underbrace{\mathrm{MF}\cdot n'}_{\text{Linear}} \end{aligned}$$

The linear part is precisely the same as what we had before, so we also start by removing the high frequency bins as before. However, we now also need to apply an offset corresponding to the affine component. So, before the IDFT, we apply a phase shift to every sample corresponding to a shift $n_0\,(\mathrm{MF}-1)$ in the time domain. With this, now the RCM correction is done correctly even with the initial extraction delay.

Two important notes:

- This is the first step that *forces* us to use a non-power-of-2 IDFT (due to the arbitrary reduction in range length). As this is done for most Dopplers, having a very optimized FFT/IFFT procedure of even non-power-of-2 is critical.
- When performing the linear compression, if the *new* DFT bin count is *even* we should average the last highest frequency bins (positive and negative) instead of removing either of them.

As we want SARSYS to be able of processing images with a considerable amount of squint (goal G5), the RCM corretion needs to be lightly adjusted. An example of a squinted target trail is present in fig. 3.16.

As you may see, as the radar platform moves upwards, when the target initially enters the antenna beam, it should have been a positive Doppler fingerprint. However, you may see that the Doppler is so large that it actually wrapped around to the negative Doppler frequencies. By itself, this is no problem at all! This occurred because the antenna is now squinted forward, and the targets are never seen with negative Doppler (in this case). So it is just a matter of the Doppler centroid being non-null and the spectrum wrapping around the limits.

However, if we apply the RCM procedure described earlier, *that* is a problem, as the algorithm assumes the symmetry around null Doppler, and shifts the negative frequencies just like the positive ones, when it should shift that bottom part in a completely different way. The solution to this is represented at the top of the diagram; we should make the processing use a virtual offset spectrum that doesn't range from $-f_{\text{dop, max}}$ to $f_{\text{dop, max}}$, but actually *acts* like it goes from $f_{\text{dop, cen}} - f_{\text{dop, max}}$ to $f_{\text{dop, cen}} + f_{\text{dop, max}}$ (where $f_{\text{dop, cen}}$ is the Doppler centroid). With this, the RCM in the image should be corrected and the "half-smile" should be transformed into the non-dashed, wrapped vertical line.

Azimuth compression

Finally, it is time to explain how our multilook algorithm works, and how it differs from the time-domain multilook from the literature.

If we were to follow the literature, we would now calculate or lookup the impulse responses for the different ranges, each one with length L_i, and correlate each range gate (of length M) with them by:

- Zero-padding the impulse responses (reference signals) and the range gates to a length of $L_i + M - 1$ – to allow for the correlation to be linear
- Doing the FFTs
- Multiplicating with the conjugate of the transformed references
- And finally performing the IFFTs.

These steps are depicted in fig. 3.17. The segment in green represents the length of captured azimuth data considered for processing a given single look.

As you can see, the main problem with this approach is that when we perform the segmentation into relatively short single looks of azimuth length M (eg. $= 512$), the impulse response length becomes so large by comparison that this whole process becomes extremely wasteful due to the extraneous zero-padding. Besides, if this zero-padding and correlations were done only once, it wouldn't be a problem. However, with multilook, the idea is that we have *multiple* single looks, so this process must be repeated over and over again.

To solve this problem, the first insight required is referred in the first couple chapters of every digital signal processing (DSP) book: convolution is commutative [17]. In practice, what this means for us, is that given a FIR filter and a signal that is to be filtered (just like our azimuth impulse response and the range gates signal), it doesn't matter which one is considered the "filter" or the "signal". If we use this property, then we

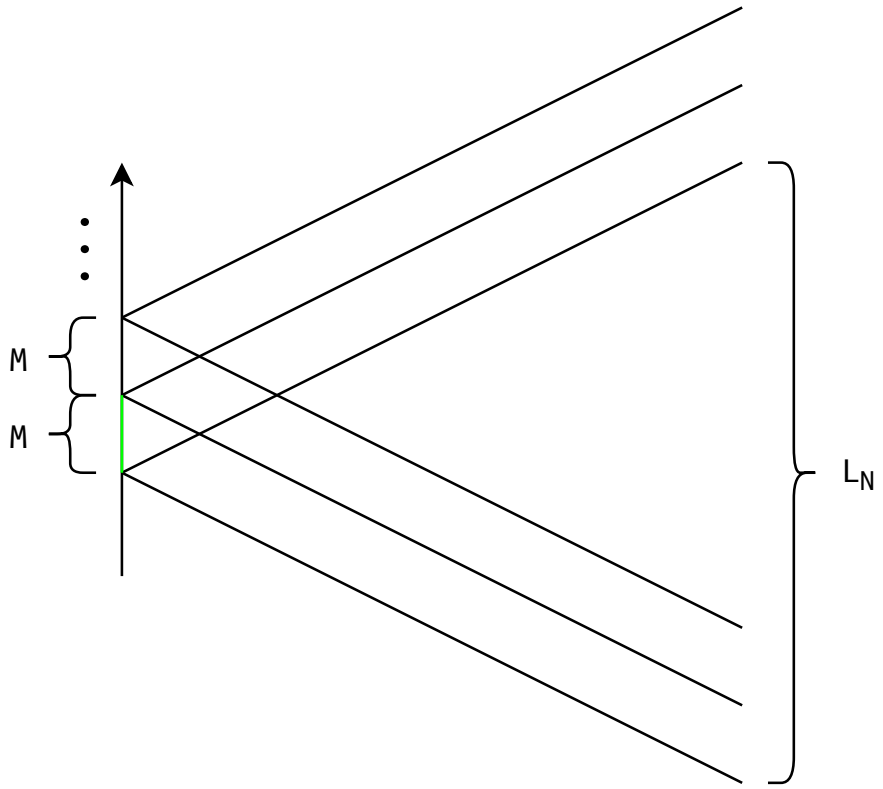

Figure 3.17: Naive time-domain multilook.

may consider the M-length azimuth data our filter, and the L_i-length impulse response our signal. Putting things this way, it starts resembling what we had during TX search in the sample stream (section 3.5.1): to filter a very long signal with a relatively small filter, it is *much* more efficient to do it with block-convolution.

Implementing it this way should result in some notable performance gains; nevertheless, there is still another problem that, if solved, could improve it further. By segmenting and processing the SAR azimuth data in looks, we are automatically reducing the output azimuth resolution, due to the smaller exposure time of the targets (ie. smaller considered Doppler bandwidth, which is what determines the resolution, see sections 2.5 and 2.10). Fortunately, there is another insight that let's us optimize some part of this resolution problem: the azimuth reference signal (ie. the impulse response) is very close to being linearly frequency modulated – this insight is actually used by the frequency-domain multilook method, so this proposed method takes ideas from both! In isolation, it is probably not immediately obvious what this implies. However, we are now performing the azimuth compression using block convolution; more specifically, we are *segmenting* the azimuth *impulse response* as if it was the signal being filtered!

Each one of those blocks, due to the approximately linear frequency modulation, will only contain a *small portion* of the bandwidth with non-negligible magnitude – see fig. 3.18. As we are performing the convolution in the frequency domain, this means that the result after the multiplication between the azimuth signal and the impulse response block DFTs will only contain some samples with significant magnitude. Therefore, if before performing the IDFT of the block, we segment it into D_B sub-blocks and add

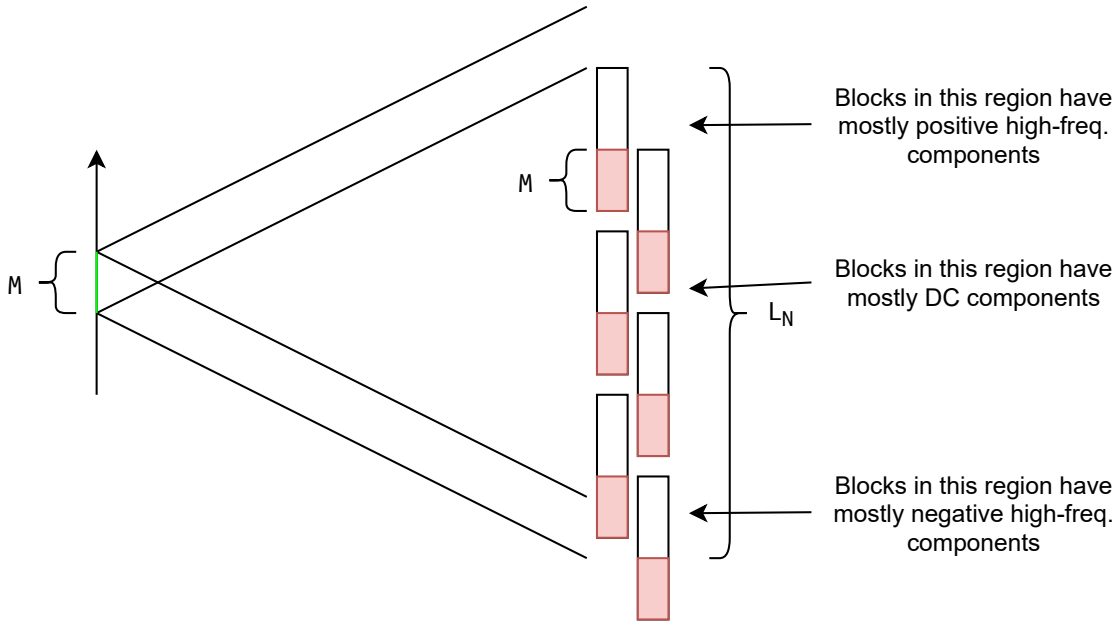

Figure 3.18: Single look azimuth compression using block convolution.

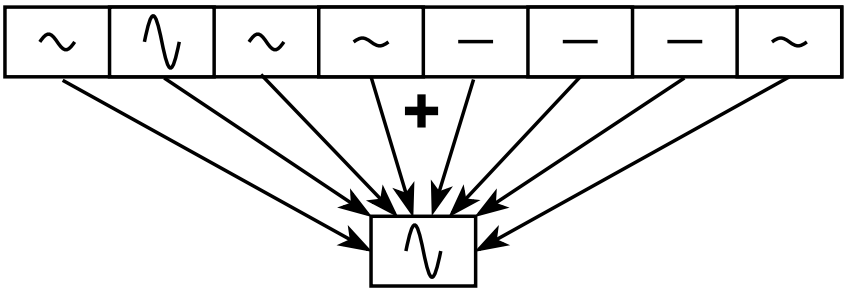

Figure 3.19: Decimation of the spectrum after convolution, with $D_B = 8$ – only one segment of the block has non-negligible magnitude.

them together – as if the spectrum wrapped due to aliasing –, then we only have to perform an IDFT that is $D_B\times$ smaller! Besides, the look image is now pre-decimated by D_B so even the compositing into the final image becomes less hardware intensive, as there are less "pixels" to compose together. In fig. 3.19 you may see a diagram that shows this wrapping decimation in practice.

Yet another advantage of this method is how everything becomes more uniform. The impulse response signals grow in length with range, and so we had, for example, to prepare various optimal lengths of DFTs for each one. With this, everything is measured and processed in blocks! There are more or less blocks, but there is only one length of DFT that is used, and the memory structure also becomes much more regular.

It also allows for the impulse response blocks to be calculated, partitioned and transformed to the frequency domain (and conjugated!) even before running the processor, and the processor may just load a file with all this data when it wants to process sessions with the baked characteristics. This reduces some seconds on the setup time of the program; for now we usually just re-calculate them everytime, while experimenting,

but in SARIA, where the mission characteristics are known ahead of the time and we want to make the processing time as minimum as possible, it will probably load the pre-calculated DFT blocks.

Of course, like always, the devil is in the details:

- Even though convolution in general is commutative, when dealing with block convolution, the blocks need to be processed in the correct order and it may be counterintuitive when implementing.
- When outputting the result of this azimuth compression into an image, consideration has to be put into how the range gates are centered between each other to form a coherent image.
- Choosing appropriate sizes for the blocks and decimation amount may have a large impact on runtime performance and image quality. For example, the total block size should be the optimal size for doing block convolution with a filter of length M, otherwise performance will be deteriorated.
- When processing squinted data (goal G5), the impulse response must also have a frequency offset that depends on the squint magnitude. If blocks within the same session have drastically different amounts of squint, the same set of impulse responses cannot be reutilized for them all. It may be possible, if memory allows, to have several sets of reference blocks with different squints all loaded in RAM and then use whichever fits the bill; if the memory isn't sufficient, it will have to recalculate the reference blocks between pulses or load another pre-calculated from the disk.
- There are many opportunities for doing implementation-level optimizations when doing a system like this from scratch. Some of those optimizations are discussed in section 3.6.

Compositing of single looks in the final image

With the previous step complete, if we now take the magnitude of the complex values in the data matrix, we obtain a single look image in isolation; this will usually cover only a small area and with very rough quality. To obtain a clear image, we have to compose several of these single looks into a final image.

After the characterization of all single looks is done (section 3.5.2), our software actually performs some heuristics on their geographical information to obtain the centroid and axis of a final image, as proposed by goal G1.

This simplifies our problem into a simple software-based compositing, just like a software rasterizer which has to blit rotated, scaled, and possibly sheared images into a backbuffer.

When doing global MoCo, the problem actually becomes even simpler, as the partial images are just added at an azimuth offset from one another. However, as we want to be able of performing look-local MoCo, it is now crucial to use the coordinate basis derived

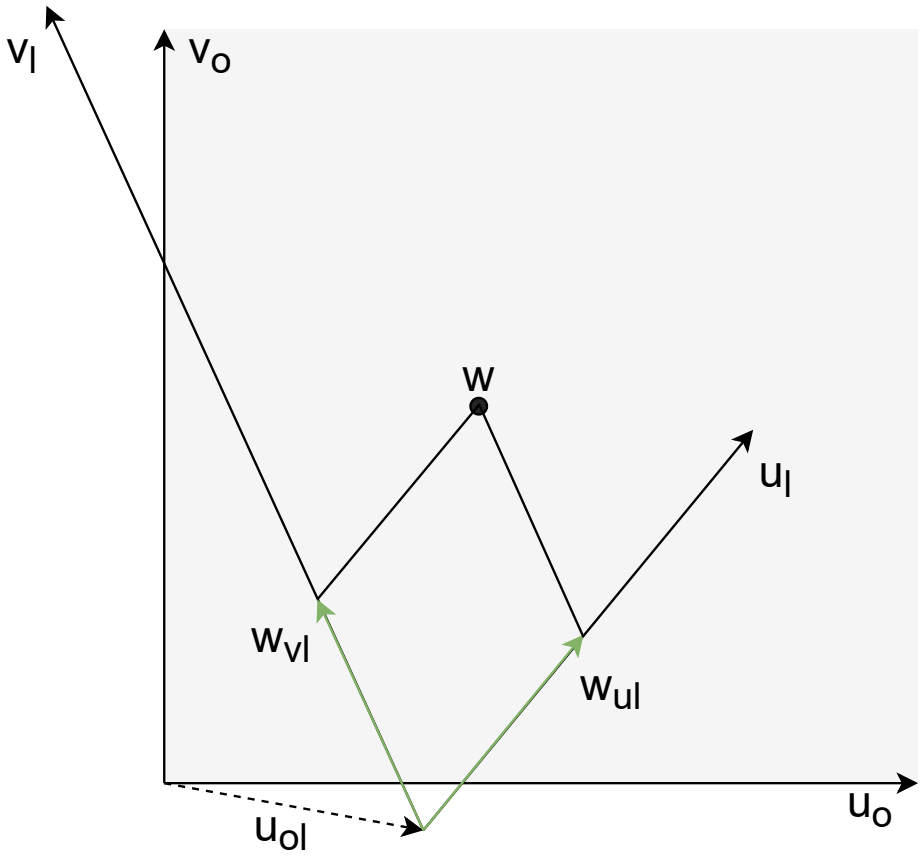

Figure 3.20: Diagram of the general case of compositing a single look image onto the final one (with rotation and shearing).

in the look characterization step, otherwise the incoherent image addition will generate unfocused images.

The way we implemented this was to, for each pixel on the final image that is broadly covered by the single look image, to change the basis of the coordinates of the queried pixel from the global basis into the single look one, and fetch the corresponding magnitude. This magnitude then gets summed to the current pixel and the magnitude for the next one is fetched.

When fetching the magnitude, we also have to consider that the axis, to make geographical sense, are defined in terms of *ground* range. So, after the change of basis, the range must be unprojected from the ground into slant range, and only then can we fetch the magnitude from the corresponding range gate. Also, in the vast majority of the cases, the queried coordinate will be between samples, so some interpolation must be made, like bilinear filtering or 2D spline interpolation. As we want to be as fast as possible in SARIA, and don't mind a minor quality hit, we have actually implemented this "interpolation" as just a nearest-neighbor fetch, and it actually doesn't seem to degrade the quality very much.

The change of basis is slightly more difficult than usual due to the possibility of squint processing, which makes the single look axis not be orthogonal to each other (therefore

requiring a shearing transform). Figure 3.20 shows this general case – and while the figure shows the final image axis aligned with the page, they might also be arbitrarily rotated. So, to handle this general case, the following algorithm was devised:

- We start by finding the origins offset u_{ol}, and the axes u_l and v_l expressed as function of the final image axes, by performing the corresponding dot products with u_o and v_o. The reason for this is that w is also written as function of these final image axes, so if we transform those at the start, we don't have to worry anymore.
- For each w, we then perform the coordinate transform. As the axes are orthogonal, we have to use the general case of

$$w_{ul} = \frac{v_l \times w}{v_l \times u_l} \qquad w_{vl} = \frac{u_l \times w}{u_l \times v_l}$$

  We are abusing the cross-product ($\times$) notation, as all these vectors are 2D only; so, in this case, the cross product represents the norm of the resultant vector, that would be on the axis orthogonal to this 2D plane.

  Note also how the denominators are constant for each single look! We cache these values and do the numerator operations in a slightly different way so that we are able to save partial results in the loops that are then reutilized in several iterations.
- This w_{ul} vector is then unprojected from ground to slant range, and the queried value is returned if it is inside the image, otherwise returns 0.

The SARSYS processor also has the configuration for summing the single looks images in squared magnitude and only doing all the square roots in the end. This prioritizes high-magnitude values (eg. values that come from regions that are more in front of the antenna during that look), and *seems* to generate more uniform images. Also, when doing nearest-neighbor "interpolation", it also reduces the number of square root instructions required, as it is easier to calculate the magnitude squared of a complex number than it is to calculate the magnitude itself.

After all looks are merged into the final image, SAR processing is finalized and the output image may be written to disk. For easy display on a map, the geocoding of each corner of the image should be embedded in the image file itself. For that, we may use a format like GeoTIFF or a custom field in Exif.

3.6 General implementation and optimization details

SARSYS was implemented from scratch in C-like C++, compiled with `clang` 7.0. The only reason for using a C++ compiler was to make use of the operator and function overloading features, which are very useful for improving readibility when programming computation-intensive programs.

The software is fully compiled with internal linkage (ie. single translation unit build) to let the compiler inline and optimize the functions however it sees fit. Without internal linkage, this may now be done with link-time optimization (LTO), but it is a slower and less effective way of doing the same thing.

Although the processing software should be very well optimized in order to meet the real-time deadlines, we prefer having a slower but *robust* program, than to have a fast but incorrect one – remember that SARIA only has a single launch campaign to work. That is why we disabled both strict overflow and strict aliasing, as these two properties are the source of many undefined behavior (UB) bugs due to the compilers optimizers aggressively using them and assuming things that are not always what the developer had in mind. Many high-reliability programs disable these two properties; a very well-known example is the Linux kernel itself. Besides, in the places where it matters, the strict aliasing rule may be reenabled on a per-variable basis by using a `restrict`-equivalent keyword.

As we know precisely what the target platform is (ie. the Intel J1900 CPU), we provide that information to the compiler with `-march=silvermont` and `-msse4.2` – this helps it generate code that is more fine-tuned for the platform. Also, to further help it with automatic vectorization, we used the flags `-fno-trapping-math`, `-fno-math-errno`, and `-fno-signed-zeros`.

The shared code base containing the source code for all developed programs clocks in at around 6500 lines of code (not considering blank or comment lines, nor third-party dependencies). The only third-party libraries we depend on are SQLite for the database, the UHD drivers for interacting with the SDR, and Intel oneAPI Math Kernel Library (MKL) for the high-performance FFT implementation (and the standard library, which is almost impossible to avoid in Linux).

As the usage of the FFT is so prolific in the SARSYS processor, we started by surveying and benchmarking several different FFT algorithms and third-party implementations right from the start [47]. The chosen alternative was to use Intel MKL library. Due to being made by the target CPU vendor, it is very hard to beat in terms of performance. Besides, it is feature-rich, supporting at least radices 2, 3, 4, 5, 7, 8, 11, 13, 16, and has a performant implementation of the Bluestein FFT algorithm (chirp-Z based) for all other lengths. Finally, the license is very permissive (as opposed to the most popular FFT library, the FFTW3, which has a very strict license).

To further improve the robustness and reliability of the SARSYS programs, we completely separated the "program logic" from platform code. The platform code has a simple API designed to do the things required by SARSYS (like creating files exclusively without overwriting, open files and seria ports, accessing the database, etc).

It also introduces platform handles for those resources, and any base function that works with one of these handles doesn't do anything (ie. are no-ops) if they are invalid – this is similar to the concept of monad. We also defined that initializing an handle to 0 is equivalent to an invalid handle so, if we initialize everything to zero by default, operations on handles that aren't further initialized just do nothing – this technique is called zero is initialization (ZII) and was coined by Muratori [26].

This means that code paths may be extremely simplified without compromising robustness (even improving it!). For example, we may perform several operations on a file, like reading and writing, without performing any check at all in between the calls. In the end, if the handle is valid, everything succeeded. If not, one of the operations failed

and logged a warning to `stdout`, and all of the remaining ones were no-ops due to an invalid handle. The program may decide, from that point, what to do: If it was essential data, it might throw a fatal error; otherwise just warn the user about what happened and move on.

Another advantage of having done this separation is that it is much easier if, in the future, we need to change the target platform from Linux to anything else. We just need to implement the platform layer over that new target platform, providing the required APIs, and it should work as expected.

Having already talked, throughout the previous sections, about the algorithmic changes we did to improve the performance of the algorithms, we'll now briefly list some implementation-level optimizations that also helped further improve the performance (in some cases, very significantly) of SARSYS:

- In general, we never think about some "object" or "thing" in isolation. We always consider how those "things" are used in practice and perform block allocations of several "things" instead of allocating each one individually. This is trivial to do and has a much better performance in comparison to regularly calling a general-purpose memory allocator like `malloc` or `new` for allocating many small things (eg. in a loop), which is extremely expensive and creates a lot bookkeeping overhead that it has to manage.

- We actually never call `malloc` either. We implemented our own arena allocator that has the "push" operation to request for a block of memory. However, *even* when pushing individual things, this arena allocator is very fast because it maintains a larger (1 MiB by default) *current* block of memory that has, in the majority of the cases, space for what we want to allocate, so most of the times it will just check that the block has enough space and then increment the "used" count by the allocated size. When the block has no space for the requested byte size, it uses our platform code to allocate another block (this just allocates a block of memory from `mmap`).

  To free memory, these arenas may either be completely cleared, or may use "temporary memory regions", where it just stores what was the state of the memory arena at a determined point, and then, after any amount of allocations, it just resets the arena to the stored state. This mostly eliminates the common "ownership" and memory leaking problems, as they are usually introduced when managing things in isolation.

  Finally, these memory arenas also handle allocating blocks that are aligned to any power of 2 and, given that we are using ZII, it should come with no surprise that it also supports clearing the memory blocks to zero before returning them.

- The type of optimization that made the most noticeable difference was improving cache utilization. When we allocate blocks of contiguous things, operating on them all becomes much less expensive due to the CPU prefetching memory regions around the areas we have written to or read from, so the next item in an array will probably already be in the cache.

  To give a concrete example, in SARSYS, when calculating the impulse responses

DFTs, we first have a pass through each range where we obtain their characteristics (such as length, number of used blocks, etc), and then have a second pass in which we allocate a *single* large buffer capable of containing them all (in the overlap-save block format), and both when filling and using during SAR processing, we are accessing this memory sequentially, so the cache helps us a lot!

Another example is that, at the start, we use range-major memory layout, so that range compression and RCM correction are efficient but, before performing the azimuthal IDFT after the RCM correction, we actually transpose this layout into azimuth-major, as the processing in now azimuth-dominant.

Both of these optimizations helped reduce the execution time very significantly!

- Another important thing is to don't mix working with double-precision and single-precision floats, as the conversion between them is expensive if used in a hot loop. For the most part, we used single-precision floating points, except in very few select places where the extra precision was really beneficial.

Finally, we also tried to perform an algorithm optimization that ended up being abandoned because it wasn't the correct trade-off to make. Usually, at the end of range compression we perform a range IDFT and at the beginning of RCM correction, we perform a range DFT. The idea was to try and discard both operations, keeping the range always in the frequency domain, which is possible due to the linearity of the DFT. However there were two main problems with this:

- First of all, we lose the possibility of performing MoCo with range-variant compensation, as MoCo is done in-between those steps, and to be range-variant, it needs the time-domain version of the range.
- Secondly, it was actually more expensive to do this, because the RCM correction requires performing an azimuth DFT for each range gate. With this, as we didn't perform the range IDFT after compression, the number of azimuth DFTs was larger than otherwise, and would make it globally slower most of the time.

Now, we actually even use non-power-of-2 DFTs in range when doing the RCM correction, as it minimizes the execution time according to our measurements.

Chapter 4

Results

In this chapter, we present the results obtained by the tests that were performed to SARSYS. As there was no chance to perform a test flight up until the delivery of the dissertation, we tested the system in a divide and conquer manner: by testing each component thoroughly, in isolation, and then testing their interfaces.

Note that most tests were *not* performed on the OBC that is to be used on SARIA. The reason is to maintain a constant frame of reference while developing, and then we periodically tested in the OBC. The development environment is a laptop connected to AC power, and which has an Intel Quad-Core i7-6700HQ (8 virtual cores), running at 2.60 GHz, and also has 8 GiB of RAM.

4.1 Capture and GPS handling software

The capture software, running in SARIA OBC with all the background services running and a remote shell, is able to:

- Correctly configure the SDR (using the PPS if available, or only with the internal oscillator as fallback)
- Transmit pulses from the provided set of pulses (just one, or sequentially if multiple), with or without staggering, all the while being able to receive continuously for at least 1min30s, without any relevant overflow or underflow occurring
- Save, to the `.ssraw` file, all the required information for correct processing. This was checked in two ways:
  - Besides the described programs, we have, over the course of this project, developed several utilities that help us check if the data is what we expect, and help visualize when debugging problems. One of those utilities is called `extractor`, and it is able to dump all information from an `.ssraw` file. When asked to print the header information about a captured file, we get this:

    ```
    $ ./extractor 2021-06-17-182618.ssraw i
    INFO
    ```

```
Reference time: 1623954313.558657369
Reference date: 2021-06-17 18:25:13.558657369
    Wavelength: 0.051688
  Carrier freq: 5.800e+09
 Sampling rate: 12500000.000000

OFFSETS
     TX infos: 108
Pulse samples: 480108
    RX blocks: 544624
   RX samples: 544648
          GPS: 3100496216
      Compass: 0

COUNTS
           TX count: 60000
       Pulse  count: 127
       Pulse length: 127
     RX block count: 1
    RX sample count: 774987892
    RX missed count: 0
    RX total  count: 774987892
    GPS entry count: 73
Compass entry count: 0
```

As you can see in the listing, this file has all possible sections except for the compass, which wasn't connected at the time. The stored frequency and sampling rate are the ones the SDR accepted, so it used what we requested. You can see, it transmitted 60000 pulses (in this case it was also staggered, with a *nominal* PRF of 1 kHz). The pulses it transmitted weren't the chirp – in this test, we transmitted a set of 127 polyphase pulses, created from a Gold sequence of order 7. Finally, you may see that there were no missed samples and that the SDR returned all these samples in a single RX block (only 1 `recv` call).

We may also ask the extractor to print the TX information:

```
$ ./extractor 2021-06-17-182618.ssraw t
=====> TX 0
       Pulse index: 0
Samples until next: 12500
 Cumulative offset: 0

=====> TX 1
       Pulse index: 1
Samples until next: 12627
 Cumulative offset: 12500
```

```
=====> TX 2
       Pulse index: 2
Samples until next: 12373
 Cumulative offset: 25127

=====> TX 3
       Pulse index: 3
Samples until next: 12627
 Cumulative offset: 37500

 ... TRUNCATED ...
```

This listing shows us that the pulses were transmitted in their increasing index order, and that it is effectively using staggering (see the `Samples until next` varying around 12500, which is the nominal sample delay).

This extraction utility allows for the extraction of all parts of the file format, but we'll keep it short and move on to the following test which is to see if the file *effectively* has what it announces.

– In this test, we'll try to search for the TXs. To do this, we modified the processor to bail out right after the range compression, because the following steps don't make sense, as the SDR was not even connected to an antenna. This is just to test if both the file was correctly saved, and if the processor is able to find the transmissions that reach RX via cross-talk. Running the modified processor command (on the development laptop), we obtain the following output:

```
$ time ./processor 2021-06-17-182618.ssraw
Loading pulses DFTs...
Searching for TXs...
Found all 60000 transmissions successfully
Performing range compression...
TX SEARCH TEST DONE, BAILING OUT...

real    0m1.544s
user    0m1.125s
sys     0m0.400s
```

This means that:

* The SDR effectively had received all the transmitted pulses via cross-talk (TX and RX ports were terminated with a load).
* The `capture` correctly interfaced with the SDR and stored all RX samples correctly, just like described in the information dump from the `extractor`.
* Both the `capture` and the `processor` are correctly handling staggering and

transmissions with multiple pulses.

Relative to the `gps` program, we can already infer that it is working by seeing that the `extractor` printed that the previous `.ssraw` file already had GPS data. The extraction from the database also seems to be working well, as we see that it extracted around 73 GPS epochs, where each epoch has a duration of a second. This means that it extracted for a little bit more than a minute, which is expected as the TX and RX were working for around a minute, and we configured the `capture` software to extract that duration with margins at the beginning and end (to reduce filtering transients on actual data).

Nevertheless, we checked that the data matched by dumping it with the `extractor` and then comparing it with a custom SQL query for the same period.

The GPS handler has also had a *soak test* of 10 hours, in which it never encountered any error, stored everything in the database and the stream dump correctly, and also updated the OBC time via NTP. We have obtained offsets of the system clock relative to the PPS better than 5 µs, which is very good.

4.2 SDR phase stability

One very important test that must verified is relative to the phase stability of the SDR itself. In the past, we've had problems achieving this stability; the plot of the phase of the cross-talk correlation peaks, over time, would evolve over time. This is terrible for SAR because it requires coherent echoes to perform its processing correctly. If the SDR itself introduces a phase modulation over time, it would cause problems unless compensated.

From the data analyzed in the previous section and another that was made later, we extracted the 60000 correlation peaks from each and plotted their phase in fig. 4.1. It looks like we don't have any problem now! The phase has some periodic noise due the transmission of different pulses, but is otherwise constant. We suspect that having the PPS signal reference has helped stabilize this.

4.3 SAR dataset 1

In this section, we'll process the first dataset that was used as a test for our system. This is data from 2018 taken from an airplane flying at an altitude of around 520 m from the ground, over Vilar de Luz, in Porto. It uses the 2.34 GHz band, with a sampling rate of 60 MHz. With a PRF of around 2 kHz, an azimuth sampling delta of around 2.5 cm is obtained.

While the data was not taken with SARSYS, so it doesn't come in any format the is understood by our programs, we made a converter that extracted the data from the MATLAB MAT-files and created a `.ssraw` file that is now understood by the `processor`.

To process this data, we resample the azimuth data to a target azimuth delta of 5 cm.

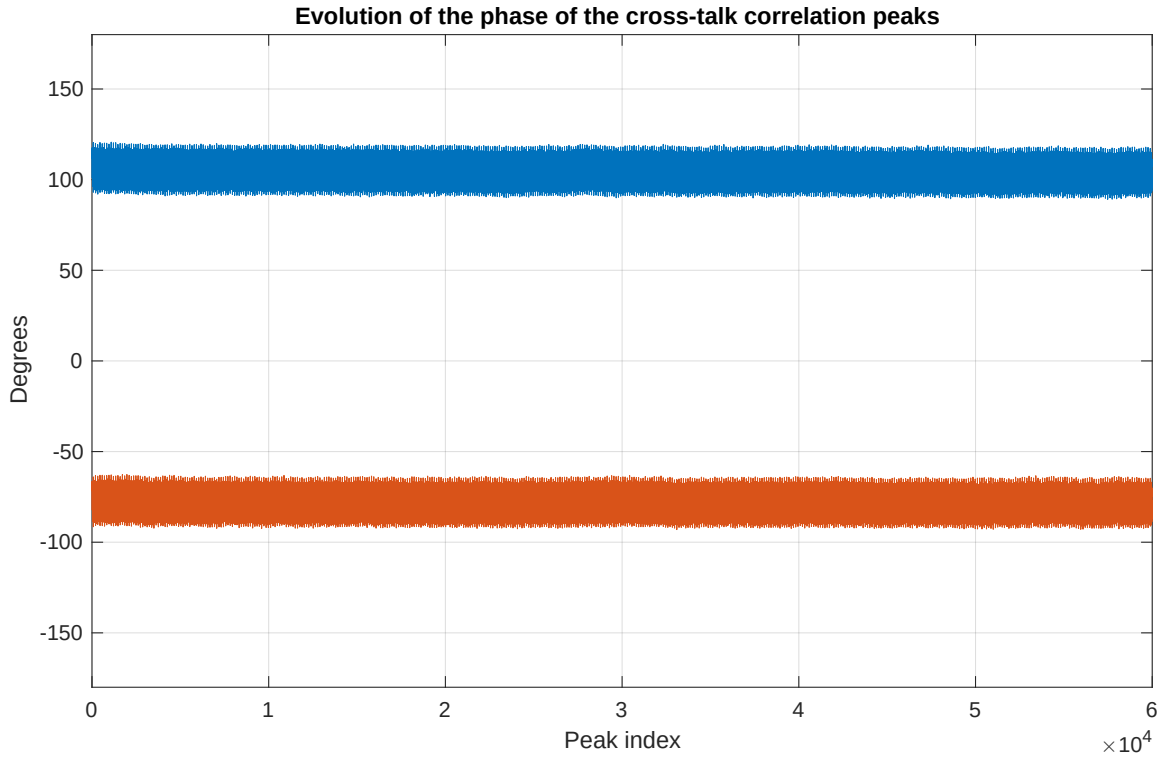

Figure 4.1: Evolution of phase of the correlation peaks, over time

4.3.1 With global MoCo

The initial way we implemented our processing algorithm was with global motion compensation. This way, each single look could more easily be added into the final image, as it would only need a constant offset in azimuth given by the decimated block length times the index of the current image. The axes and more advanced compositing as described in fig. 3.20 are not required for this processing to work.

With this algorithm we generated the image of figure fig. 4.2, which was made from 30 single looks. Generating this image on the development laptop takes around 10 s. Our MATLAB prototype of similar algorithms only processed the image from the part where the data was already range compressed, motion compensated and resampled, and took almost 2.5× the duration.

4.3.2 Local MoCo without inter-look compensation

Then, instead of global MoCo, we started doing look-local MoCo. To understand how bad the the problem of inter-look movement not being compensated, we just generated an image as if this was using global MoCo. This image is present in fig. 4.3.

If you look closely, you'll find that the image is more or less there, but it is much more unfocused than the global MoCo counterpart, mainly at larger ranges (right part of the image). For example, by looking at that diagonal line in the bottom right corner, we see that it has become smudged.

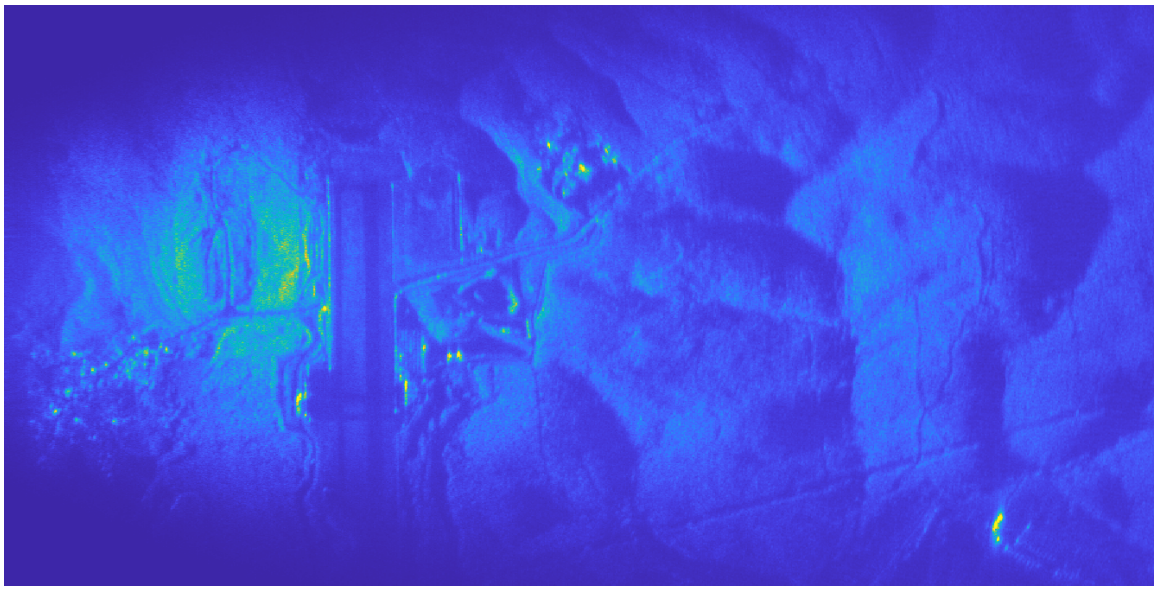

Figure 4.2: Image generated with global motion compensation

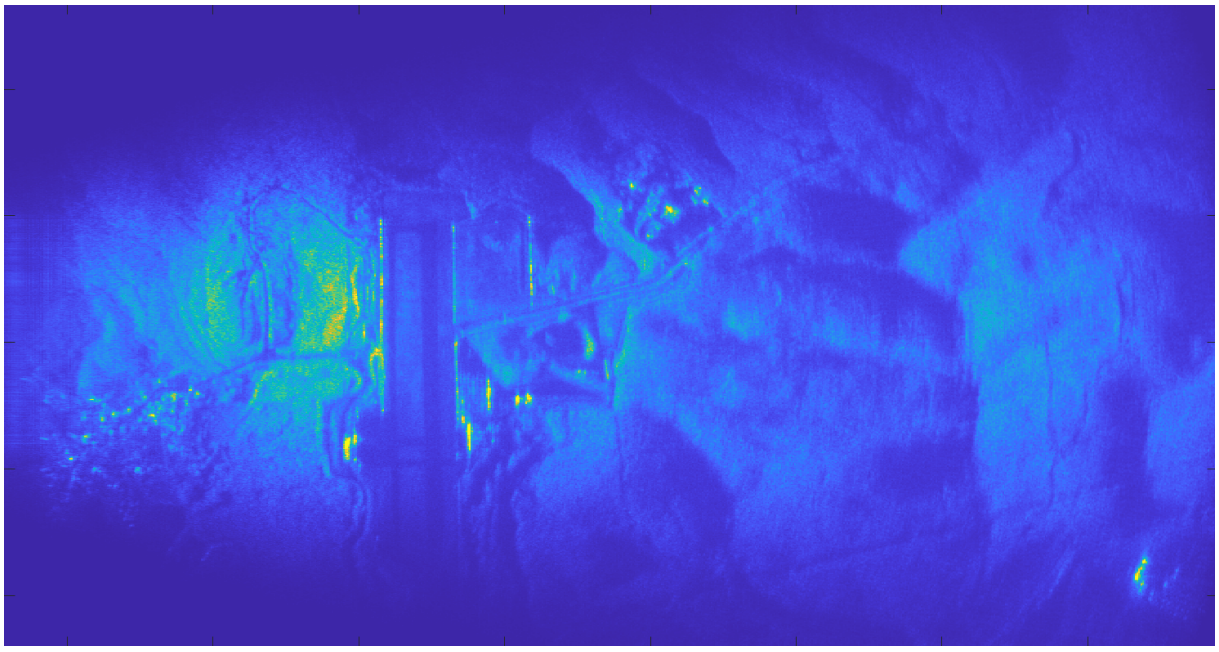

Figure 4.3: Image generated with local motion compensation, without taking into consideration inter-look non-ideal movement.

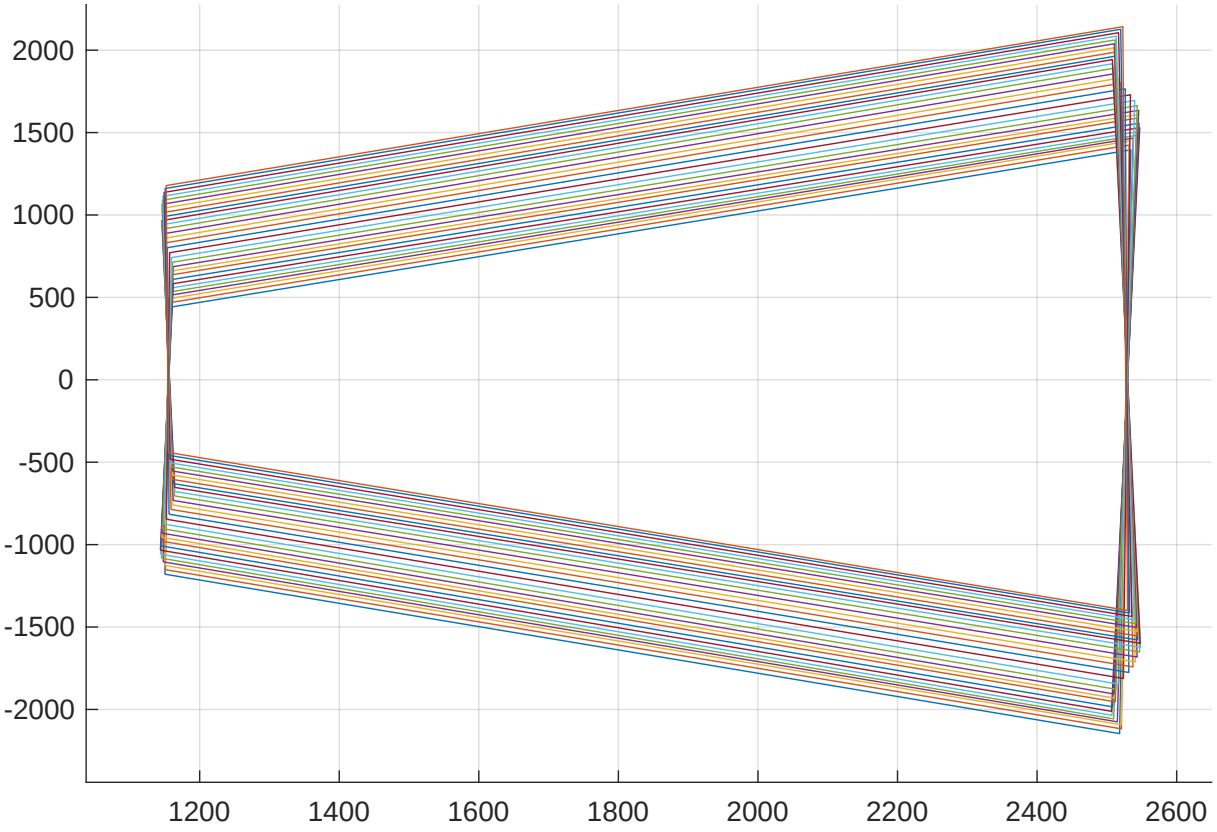

Figure 4.4: Rough estimate of the antenna swaths for each of the 30 single looks.

To visualize the problem even better, we made a diagram with the navigation data, that you may see in fig. 4.4. Now, it is clear that the airplane took very gente curves while capturing this SAR session – by looking at the corners, it is clear that they become unaligned, and therefore, that the naive single look summation to form the final image is *not sufficient*.

4.3.3 Local MoCo with inter-look compensation

Finally, we present the result of the complete algorithm that was described in chapter 3. Even though this is more complex than the global MoCo alternative, this also takes just under 10 s (single threaded) on the development laptop. This is probably because the final image is now defined and only of the region of interest, so we don't lose time performing summation outside boundaries.

The generated image is present on fig. 4.5.

In comparison to the local MoCo without inter-look compensation, this image is now crystal clear!

In comparison to the one with global MoCo, as the airplane was relatively stable most of the flight, the differences are very minor, but we believe that this one is also clearer. For example, in the top central part, those stronger reflecting targets look crispier. Therefore, if even this dataset, where the airplane was very stable, some parts of the image became

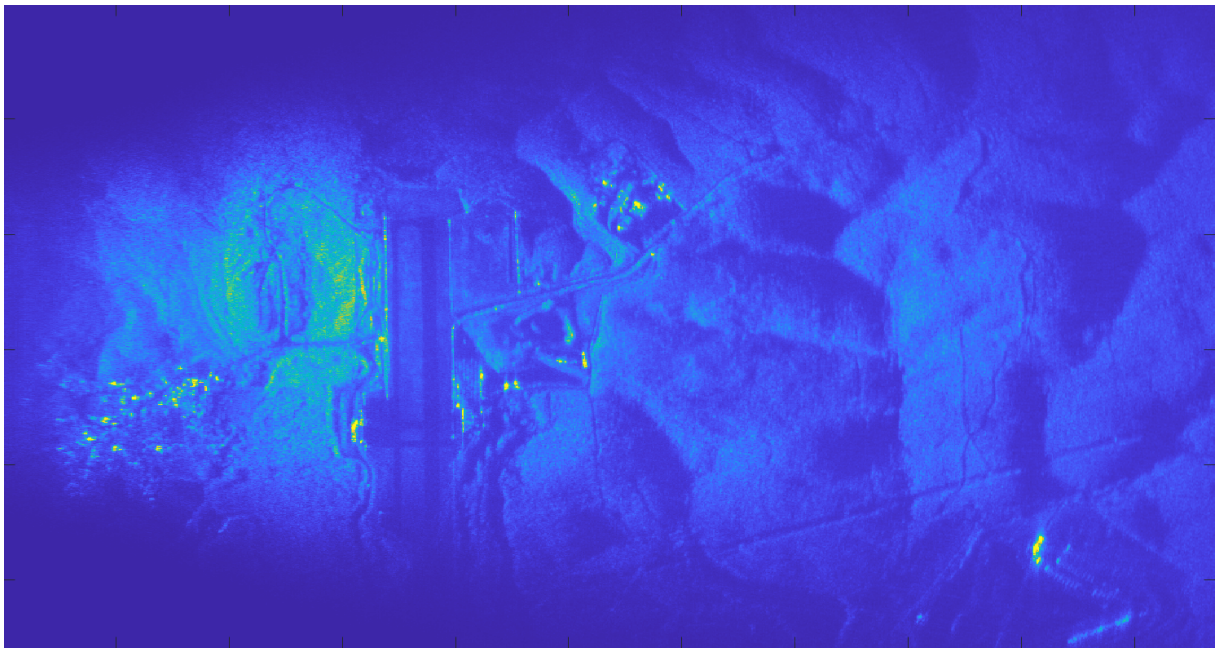

Figure 4.5: Multilook image generated by using the proposed algorithm.

clearer by using this multilook method with look-local motion compensation, we believe, without any doubt, that this method has many advantages for cases where the radar platform is not so stable or slowly rotating over time – SARIA is one example of such platform.

In fig. 4.6, we find that the ground projection closely matches the orthorectified imagery from Google Maps. The differences are mostly from elevation changes in the ground. As we project considering a planar ground, these differences are bound to occur in regions with hills.

Processing this image on SARIA's OBC takes around 50 s using just a single thread, and where most of the time is taken calculating the single looks. This image, while having different processing parameters from SARIA, is actually a good stand-in for the processing complexity that we are going to encounter. In SARIA, even though the platform is at an higher altitude, the impulse response in terms of samples because we are going to use a resampling closer to 25 cm (as opposed to the 5 cm in this case), and also the antenna beamwidth is finer due to larger antenna gain (we are going to use parabolic reflectors with around 30 dBi). So, the projection is both smaller due to the finer beamwidth, and less frequently sampled than in this case.

Besides, In the range direction, we also only sample at 12.5 MHz, which is much smaller than 60 MHz.

So, all things considered, this image is actually representative of the computational complexity expected for SARIA – and if the processing of this image is within requirement R1, then so are SARIA's. If required to be faster, we may also *trivially* put the single looks calculation running in parallel with each other; given the architecture of the program, a change like that is very easy to do.

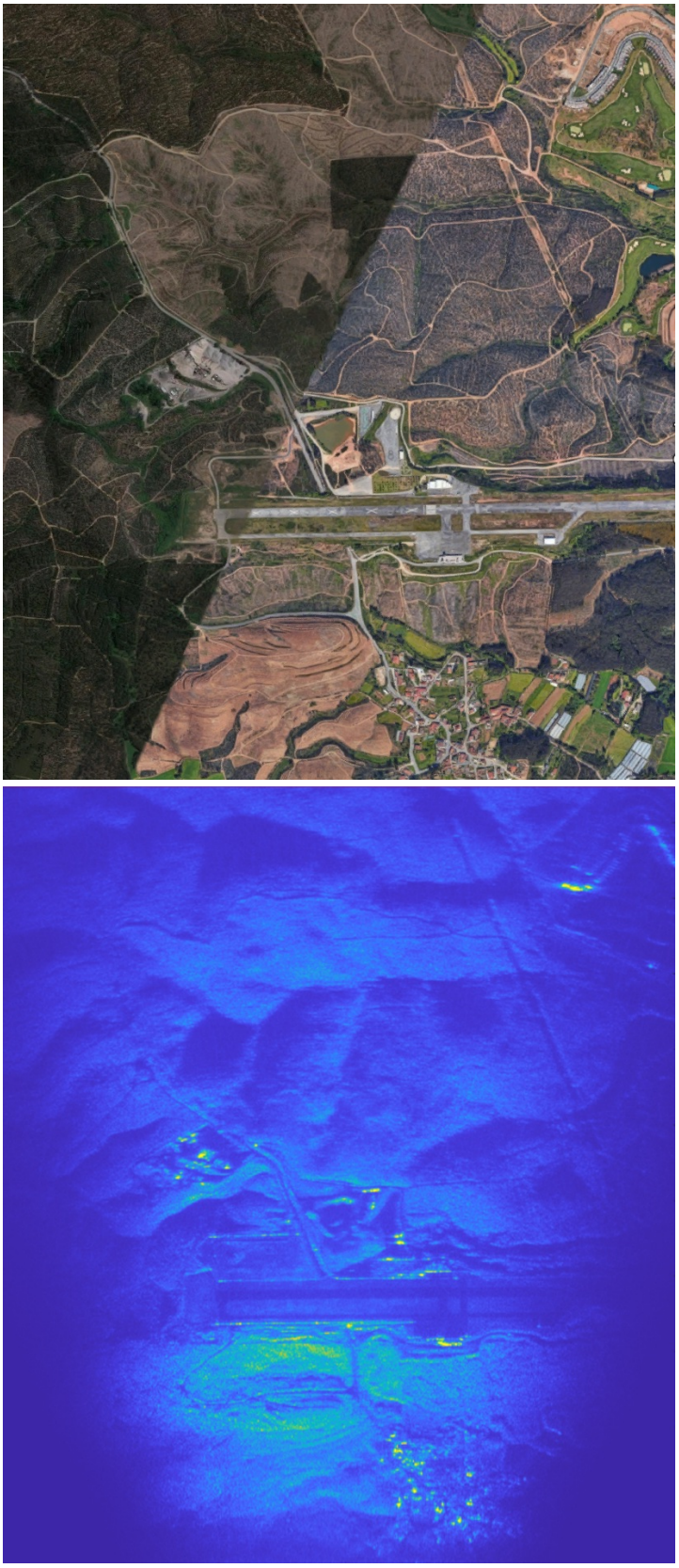

Figure 4.6: Comparison of the ground-projected image to orthorectified satellite imagery (source: Google Maps).

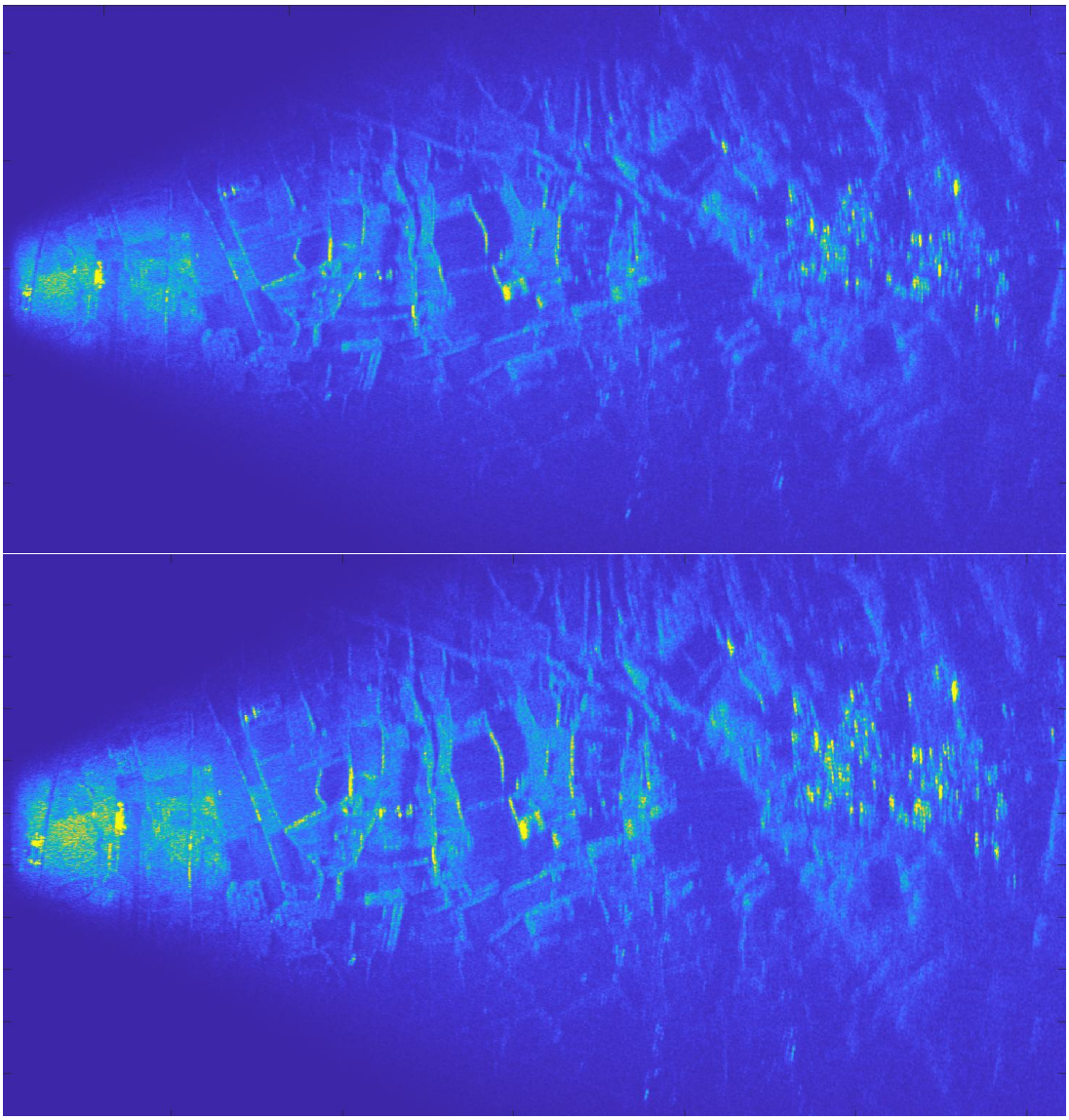

Figure 4.7: Comparison of global (top) vs local (bottom) motion compensation.

4.4 SAR dataset 2

We have also processed another dataset with different characteristics (staggered SAR with polyphase pulses, and a much higher PRF), to further test our system. The images are present in fig. 4.7.

They are pretty clear in the beginning, but end up getting unfocused. As both of them are unfocused in the right part of the image, we think that the navigation data itself may have been too noisy to compensate properly, and could require additional filtering for correct compensation.

Chapter 5

Conclusion

Synthetic-aperture radar is a fascinating technology with many diversified applications. As more and more small, cheap, and accessible COTS radio components fill the market, so grows the need of having highly optimized processing software. In the course of this dissertation we have implemented, from the ground-up, an end-to-end SAR processing system that is able to use available resources very effectively.

One of the problems of long-duration SAR sessions is that, if processed in one go, motion compensation becomes hard to do correctly. Time-domain multilook processing allows for the compensation to become local to each single look, but the classical version of that algorithm consumes too many resources for it to be useful. In the proposed system, we have implemented a novel way of performing the time-domain multilook processing that makes it feasible even for embedded systems to use it as their primary algorithm.

Being an end-to-end system, everything had to be designed and thought of, from the highly technical parts – such as what pulses to transmit and why, and how the algorithms are implemented in an efficient way – to the trivial ones – such as how the data flows from one component to another, and how the user interacts with the system.

While this system was designed specifically for SARIA, it actually ended up being so much more than that. All objectives have been achieved, all imposed requirements have been followed, and all extra goals have been implemented.

Appendix A

Glossary

chirp signal in which the frequency monotonically changes with time. 12, 14, 15, 41, 42

full-duplex simultaneous transmission and reception. 4, 45

multithreading ability of running several threads of execution concurrently. 27

real-time describes computing operations that must guarantee response times within a specified time. 3, 4, 27, 31, 62

SQLite library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. 35–37, 62

staggered SAR mode of operation in which the pulse repetition frequency is continuously varied. 10, 19, 34, 35, 43, 45, 47, 51, 74

stripmap SAR the antenna pointing direction is constant as the platform moves. 8, 10, 18

UBX u-blox proprietary binary protocol to transmit GPS data to a host computer using asynchronous RS232 ports. 38

Unix time system for describing a point in time. 47

Appendix B

Acronyms

AC alternated current. 65

API application programming interface. 36, 39, 62, 63

ASCII American Standard Code for Information Interchange. 38

BEXUS Balloon Experiments for University Students. vii, 2, 29–32

COTS commercial off-the-shelf. i, 2, 75

CPU central processing unit. 3, 26, 27, 31, 38, 62, 63

DC direct current. 43, 51

DCD data. 39

DEM digital elevation model. 38

DFT discrete Fourier transform. 26, 49, 51, 54, 55, 57–59, 64

DLR German Aerospace Center. 29

DSP digital signal processing. 56

E-Link Esrange Airborne Data Link. 37

ENU east north up. 38

ESA European Space Agency. 29

FFT fast Fourier transform. 11, 19, 25, 26, 51, 54–56, 62

FIR finite impulse response. 11, 49, 52, 56

FM frequency modulation. 12, 24

FPS frames per second. 27

FSL free-space losses. 20–22

GPS Global Positioning System. 19, 31, 35, 38–40, 48, 68

HAB high-altitude balloon. 8, 14, 15, 29

HPBW half-power beamwidth. 7, 16, 17

I/O input/output. 45

IDFT inverse discrete Fourier transform. 55, 57, 58, 64

IFFT inverse fast Fourier transform. 11, 20, 24, 55, 56

IIR infinite impulse response. 51, 52

INS inertial navigation system. 19

ISM Industrial, Scientific, and Medical. 2, 29, 31

LPF low-pass filter. 22, 32, 33

LTI linear time-invariant. 45

LTO link-time optimization. 61

LTP local tangent plane. 51

MKL Math Kernel Library. 62

MoCo motion compensation. vii, viii, 19, 20, 24, 25, 34, 48, 51–53, 59, 64, 69–71, 74

NMEA National Marine Electronics Association. 38

NTP Network Time Protocol. 39, 68

OBC on-board computer. 2, 3, 29, 31, 33, 35, 38–40, 65, 68, 72

OS operating system. 35, 36, 40

OTW over-the-wire. 44, 47

PLL phase-locked loop. 40, 46

PN pseudo-noise. 43

PPS pulse per second. 39, 40, 45, 65, 68

PRF pulse repetition frequency. 7, 10, 15, 17–19, 23, 32, 37, 42, 45, 52, 66, 68, 74, 77

PRI pulse repetition interval. 7, 14, 18

RAM random-access memory. 27, 31, 59, 65

RCM range cell migration. vii, 17, 19, 20, 54–56, 64

RCS radar cross section. 21, 32

RDA range Doppler algorithm. 19, 24

RF radio frequency. 3, 11, 21, 29, 31, 40, 45, 49

RX reception. vii, 21, 40, 41, 45–49, 66–68

SAR synthetic-aperture radar. i, vii, 1–5, 7–10, 14–22, 25, 29, 31, 33–38, 40–43, 45, 47, 48, 51, 53, 54, 57, 61, 64, 68, 71, 74, 75, 77

SARIA Synthetic-Aperture Radar using an Inflatable Antenna. i, iii, vii, ix, 2, 4, 8, 10, 22, 25, 29, 30, 33, 35, 36, 40–42, 45, 46, 59, 60, 62, 65, 72, 75

SARSYS SARIA System. vii, 4, 29, 33–39, 43–51, 53, 55, 61–63, 65, 68

SDR software-defined radio. i, 2, 4, 12, 15, 29, 31, 36, 39–41, 44–47, 49, 62, 65–68

SHM shared memory. 39

SIMD single instruction multiple data. 26, 27

SLAR side-looking airborne radar. vii, 5–7, 10, 14–16

SNR signal-to-noise ratio. 14, 19, 33, 43

SNSA Swedish National Space Agency. 29

SQL Structured Query Language. 35, 36, 68

SSD solid state drive. 36

TBP time-bandwidth product. 21, 22

TT&C telemetry, tracking, and command. 33, 37

TX transmission. vii, 20, 40, 41, 45–53, 57, 66–68

UB undefined behavior. 62

UHD USRP hardware driver. 36, 45, 47, 62

USB Universal Serial Bus. 36

UTC Coordinated Universal Time. 39

VR virtual reality. 27

ZII zero is initialization. 62, 63

Appendix C

Bibliography

[1] M. I. Skolnik, Ed., *Radar Handbook*, 3rd ed. McGraw-Hill, 2008, ISBN: 9780071485470.

[2] K. Tomiyasu, "Tutorial review of synthetic-aperture radar (SAR) with applications to imaging of the ocean surface," *Proceedings of the IEEE*, vol. 66, no. 5, pp. 563–583, 1978. DOI: 10.1109/PROC.1978.10961.

[3] I. Cumming and F. Wong, *Digital Processing of Synthetic Aperture Radar Data: Algorithms and Implementation*, ser. Artech House Remote Sensing Library. Artech House, 2005, ISBN: 9781580530583.

[4] J. W. Goodman, *Introduction to Fourier Optics*, 4th ed. W.H. Freeman, 2017, ISBN: 9781319119164.

[5] R. O. Harger, *Synthetic Aperture Radar Systems: Theory and Design*, ser. Electrical science series. Academic Press, 1970, ISBN: 9780123250506.

[6] T. S. Rappaport, "The wireless revolution," *IEEE Communications Magazine*, vol. 29, no. 11, pp. 52–71, Nov. 1991, ISSN: 0163-6804. DOI: 10.1109/35.109666.

[7] M. Scarito, *Build your own synthetic aperture radar*, DEF CON 19, 2011. [Online]. Available: https://www.youtube.com/watch?v=DsNfpSXvQ3M.

[8] C. A. Balanis, *Antenna Theory: Analysis and Design*, 4th ed. Wiley, 2016, ISBN: 9781118642061.

[9] W. G. Carrara, R. S. Goodman, and R. M. Majewski, *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*, ser. The Artech House Remote Sensing Library. Artech House, 1995, ISBN: 9780890067284.

[10] C. V. Jakowatz, D. E. Wahl, P. H. Eichel, D. C. Ghiglia, and P. A. Thompson, *Spotlight-Mode Synthetic Aperture Radar: A Signal Processing Approach*. Springer, 1996, ISBN: 978-1461285731.

[11] J. A. Richards, *Remote Sensing with Imaging Radar*. Springer, 2009, ISBN: 9783642020193.

[12] A. Hein, *Processing of SAR Data: Fundamentals, Signal Processing, Interferometry*. Berlin New York: Springer, 2004, ISBN: 978-3540050438.

[13] D. R. Wehner, *High Resolution Radar*. Artech House, 1987, ISBN: 9780890061947.

[14] C. Özdemir, *Inverse Synthetic Aperture Radar Imaging with MATLAB Algorithms*, ser. Wiley Series in Microwave and Optical Engineering. Wiley, 2012, ISBN: 9780470284841.

[15] M. Villano, G. Krieger, and A. Moreira, "Staggered-SAR for high-resolution wide-swath imaging," in *IET International Conference on Radar Systems (Radar 2012)*, 2012, pp. 1–6. DOI: 10.1049/cp.2012.1600.
[16] ——, "A novel processing strategy for staggered SAR," *IEEE Geoscience and Remote Sensing Letters*, vol. 11, no. 11, pp. 1891–1895, 2014. DOI: 10.1109/LGRS.2014.2313138.
[17] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 3rd ed. Pearson, 2010, ISBN: 9780131988422.
[18] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, 1st ed. California Technical Pub, 1997, ISBN: 9780966017632.
[19] B. E. Dunne, "The what, how and why of complex sampling for SDR transceivers," in *Proceedings of the 2019 ASEE North Central Section Conference*, ASEE North Central Section Conference, 2019.
[20] C. A. Wiley, "Pulsed doppler radar methods and apparatus," U.S. Patent 3196436A, Aug. 13, 1954.
[21] D. M. Pozar, *Microwave Engineering*, 4th ed. Wiley, 2012, ISBN: 9780470631553.
[22] J. C. Curlander and R. N. McDonough, *Synthetic Aperture Radar: Systems and Signal Processing*, ser. Wiley Series in Remote Sensing. Wiley, 1991, ISBN: 9780471857709.
[23] D. P. Duncan, "Motion compensation of interferometric synthetic aperture radar," M.S. thesis, Brigham Young University, Aug. 2004.
[24] D. Kaliyari, A. Shukla, Y. Rao, and H. B. Hablani, "Motion compensation of airborne synthetic aperture radar," *IFAC Proceedings Volumes*, vol. 47, no. 1, pp. 627–634, 2014, ISSN: 14746670. DOI: 10.3182/20140313-3-IN-3024.00065. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1474667016327215.
[25] G. Wang, L. Zhang, J. Li, and Q. Hu, "Range cell migration correction analysis of one-step and two-step motion compensation for millimeter-wave airborne SAR imaging," *EURASIP Journal on Advances in Signal Processing*, 2016. DOI: 10.1186/s13634-016-0416-1. [Online]. Available: https://asp-eurasipjournals.springeropen.com/articles/10.1186/s13634-016-0416-1.
[26] C. Muratori. "Handmade hero." (2000–2099), [Online]. Available: https://handmadehero.org/.
[27] A. Fog, "Optimizing software in c++: An optimization guide for Windows, Linux, and Mac platforms," Jan. 31, 2021. [Online]. Available: https://www.agner.org/optimize/.
[28] ——, "Optimizing subroutines in assembly language: An optimization guide for x86 platforms," Jan. 31, 2021. [Online]. Available: https://www.agner.org/optimize/.
[29] ——, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," Jan. 31, 2021. [Online]. Available: https://www.agner.org/optimize/.
[30] "REXUS/BEXUS." (), [Online]. Available: http://rexusbexus.net.
[31] A. Frenea-Schmidt, Ed., *Bexus user manual*, English, version Version 8.0, REXUS/BEXUS Programme, Jun. 3, 2021, 78 pp.
[32] M. Santos, "Inflatable reflector antenna," In development, M.S. thesis, Faculty of Engineering of the University of Porto, 2020.
[33] T. Martins, "RF front-end for radar," In development, M.S. thesis, Faculty of Engineering of the University of Porto, 2020.

[34] SQLite, Ed. "Most widely deployed and used database engine." (2000–2099), [Online]. Available: https://sqlite.org/mostdeployed.html.
[35] *Syncthing*. [Online]. Available: https://syncthing.net.
[36] Wikipedia contributors. "Ini file — Wikipedia." (Jun. 22, 2021), [Online]. Available: https://en.wikipedia.org/wiki/INI_file.
[37] u-blox, *TIM-LF, GPS receiver module*, GPS.G3-MS3-03017-G, Feb. 6, 2006.
[38] ——, *ANTARIS positioning engine, NMEA and UBX protocol specification*, GPS.G3-X-03002-D, version 5.00, 2003.
[39] J. Mogul, D. Mills, J. Brittenson, J. Stone, and U. Windl, "Pulse-Per-Second API for UNIX-like operating systems," RFC Editor, RFC 2783, Mar. 2000, pp. 1–31. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2783.txt.
[40] R. Giometti. "Pps - pulse per second." (2007), [Online]. Available: https://www.kernel.org/doc/html/latest/driver-api/pps.html.
[41] NTP. "Shared memory driver." (Aug. 8, 2014), [Online]. Available: http://doc.ntp.org/current-stable/drivers/driver28.html.
[42] D. L. Mills. "PPS clock discipline driver." (Mar. 31, 2014), [Online]. Available: http://doc.ntp.org/current-stable/drivers/driver22.html.
[43] M. West. "Re: B200mini: Using gpio pin for pps reference (re-send)." Message to the usrp-users mailing list. (Apr. 21, 2016), [Online]. Available: https://lists.ettus.com/empathy/thread/QKLIYEKOY2VGSHLLSG75HJUZMRUDU37L#SGXBCTD7QVZMABIMPXFELZJXMEQCAWXF.
[44] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing*, 4th ed. Pearson, 2014, New International Edition, ISBN: 9781292025735.
[45] Intel Corporation. "FFT length and layout advisor." (Jul. 12, 2020), [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/fft-length-and-layout-advisor.html.
[46] B. Likhterov and N. S. Kopeika, "Hardware-efficient technique for minimizing startup transients in direct form ii digital filters," *International Journal of Electronics*, vol. 90, no. 7, pp. 471–479, 2003. DOI: 10.1080/00207210310001612482.
[47] GEMMI. "Comparison of fast Fourier transform libraries." (Feb. 22, 2020), [Online]. Available: https://github.com/project-gemmi/benchmarking-fft/.