The Multi-Sensor Navigation Analysis Tool (MuSNAT) – Architecture, LiDAR, GPU/CPU GNSS Signal Processing

T. Pany, D. Dötterböck, H. Gomez-Martinez, M. Subhan Hammed, F. Hörkner, T. Kraus, D. Maier, D. Sanchez-Morales, A. Schütz, *Universität der Bundeswehr München, Neubiberg, Germany*
P. Klima, D. Ebert, *Axtesys GmbH, Graz, Austria*

BIOGRAPHIES

Thomas Pany is with the Universität der Bundeswehr München at the faculty of aerospace engineering and leads the navigation group within Institute of Space Technology and Space Applications (ISTA). He is working with GNSS since 1997 and with software radio technology since 2002. He has around 200 publications including one book and five patents.

Dominik Dötterböck received his diploma in Electrical Engineering and Information Technology from the Technical University Munich. Since 2007 he is a senior research associate at the Bundeswehr University Munich at the Institute of Space Technology and Space Applications. His current research interests include signal design, signal-processing algorithms for GNSS receivers and software receivers.

Harvey Gómez-Martínez is a Research Associate at the Universität der Bundeswehr München. His research area is visual relative navigation based on active and passive cameras, and LiDAR, mainly for aerospace applications. He holds a M.Sc. in Aeronautics and Space Technology from Technische Universität München, a M.Sc. in Astronautics and Space Engineering from Cranfield University, and he is PhD in Engineering from the Universität der Bundeswehr München.

Muhammad Subhan Hameed studied Electrical Engineering at National University of Sciences and Technology (NUST) in Pakistan and is currently pursuing a masters degree in space science and technology from Technical University Munich (TUM). His research interests focus on satellite navigation and GNSS receiver technology.

Florian Hörkner is a student of the Universität der Bundeswehr München at the faculty of electronics and technical informatics. He participates in designing and implementing the software quality management and supports the setup of the automatical test environment of the software.

Thomas Kraus is a research associate at the Bundeswehr University Munich. His research focus is on future receiver design offering a superior detection and mitigation capability of RF interferences. He holds a master's degree in Electrical Engineering from the Technical University of Darmstadt, Germany.

Daniel Simon Maier has a professional training as a technical draftsman and received a bachelor in Physics in 2015 and a master in Applied and Engineering Physics in 2017 from the Technical University of Munich (TUM), Germany. Since 2017 he has been a research associate at the Institute of Space Technology and Space Applications of the "Universität der Bundeswehr München." His current research interests include GNSS signal generation, signal authentication, and signal performance analysis.

Daniela Sánchez-Morales studied Telematics Engineering at Instituto Tecnológico Autónomo de México (ITAM) in Mexico City and holds a Master's degree in satellite applications engineering from the Technical University Munich (TUM). Currently her research focuses on LiDAR, relative navigation algorithms particularly for terrestrial applications and sensor fusion.

Andreas Schütz is a Research Associate at the Universität der Bundeswehr München focusing on Sensor Fusion, mainly of GNSS Software Receiver Technology and inertial navigation sensors. He holds a Master's degree in Geodesy and Geoinformation from the Technical University of Munich (TUM).

ABSTRACT

In this paper we summarize enhancements to the Global Navigation Satellite System (GNSS)-software receiver ipexSR developed at the Universität der Bundeswehr München since 2004. The ipexSR was conceived as a multi-GNSS, multi-frequency software receiver and achieved very soon all-in-view real-time tracking capability with a reasonable geodetic quality of the produced raw data. Like other software receiver developments, it has been used in numerous GNSS research projects always allowing to do an in-depth analysis of every GNSS signal processing aspect. However, during the last two years, it became obvious that several architectural changes are necessary to improve the usability and scope of this analysis tool. In current high-end receivers, GNSS signal processing is deeply fused with other sensor processing pipelines. The various processes influence each other, and a timely synchronized data analysis is necessary to gain in-depth insight when developing new algorithms. Therefore, Inertial and Light Detection and Ranging (LiDAR) processing has been included (to some extend also camera) in our software package and it was consequently renamed MuSNAT (MUlti-Sensor Navigation Analysis Tool). GNSS signal tracking has been reworked and verified and supports a fast and a precision mode on the Central Processing Unit (CPU). Signal processing on the Graphics Processing Unit (GPU) has been added and this paper discusses the chosen approach and limitations when porting the precision mode to a GPU. Standard GNSS-signals, AltBOC-signals and via the flexible GPU-language also new GNSS-signal candidates are easily supported. LiDAR processing currently targets relative navigation making use of the point-cloud library PCL. The integration filter merges GNSS, inertial and LiDAR data and is an extension of the Real-Time-Kinematic-Library (RTK-LIB). RTK-LIB is also used for RTK and Precise point Positioning (PPP) processing. The GNSS data flow can be inverted to use the receiver as a GNSS signal generator (transceiver concept). For ease of data analysis, a dedicated data visualization software named MuSNAT-Analyzer is developed which interfaces to the software receiver MuSNAT-Core via a Structured Query Language (SQL)-database. The SQL-database serves also as basis for further MATLAB based analysis. For complex receiver configurations, the demands on the SQL-performance are significant. An automated test framework ensures the stability of the software and is based on the Jenkins-environment.

INTRODUCTION AND OVERVIEW

Current navigation research faces several challenges due to the ever-increasing demands on navigation performance and the availability of new sensor technologies or ranging signals. To illustrate this with a few examples let's start with navigation for autonomous driving. If GNSS is used for this purpose not only a carrier phase-based position method is used but also fusion with inertial data is considered to be essential to derive the required continuity and integrity [1]. LiDAR is also often mention in the context of autonomous cars not only for obstacle detection but also for relative navigation. In such a combined positioning system (GNSS+Inertial Measurement Unit (IMU)+LiDAR) the obtained positioning performance results from stable GNSS signal tracking, correct identification of outliers and carrier phase cycle slips, provision of RTK or PPP assistance data, calibration of inertial errors (e.g. gyro and accelerometer biases) and bridging PPP/RTK outages with inertial navigation or LiDAR odometry (which itself implies clustering and identification of point clouds). The processes are totally linked to each other as for example an undetected cycle-slip can not only cause an GNSS position outlier but can also invalidate inertial bias parameters. Integrity of the position is derived by comparing position solutions from individual sensors. Another relevant application case is the modernization of the GNSSs themselves. This is a continuously running process and currently signals for the 2nd generation of Galileo are defined but also other radio-navigation systems like the use of Low Earth Orbit (LEO) mega-constellations or the fusion of GNSS with 5G signals is discussed. There exists a canonical way to evaluate the performance of a new signal or a new constellation (e.g. ranging errors, Dilution of Precision (DOP)-factor, …) but it is often difficult to understand how those new signals impact an integrated multi-sensor navigation system. Those systems typically show in open-sky conditions already today a remarkable performance and achieve centimeter accuracy in an absolute sense [2]. The will benefit from new signals only in difficult (urban, canopy, indoor, …) conditions where all the different operation modes of an integrated navigation system are relevant.

For those two and many more applications a dedicated analysis tool is needed to analyze and understand the complex positioning process. Such a tool has been designed at the Institute of Space Technology and Space Applications (ISTA) and is being actively developed since 2017. The design drivers for this tool are:

- Realize all functions of an integrated navigation system with real-time capability
- Provide in-depth analysis functions for all sensor streams and processing algorithms
- Use of a conventional operating system for easy post-processing operation and use in teaching
- Support of industry standards for data-input and output
- High processing performance for real-time operation and fast scripted systematic data analysis

Several approaches have been considered to develop such a tool including the use of high-level script languages like MATLAB or python, the use of a middleware like GNU radio, Simulink or the robot operating system (ROS). Finally, it was decided to base the tool on the existing GNSS software receiver ipexSR [3] mostly because this receiver is based on C++ and the direct use of C++ provides on one side enough structures and convenience to develop and maintain a large scale software project and is on the other side still performant and flexible enough to support the high bandwidth data streams that occur during GNSS signal processing. For C++ many software engineering tools are available for efficient development (Visual Studio IDE), debugging (Visual Studio and nVidia nSight), documentation (doxygen) and profiling (Intel Profiler).

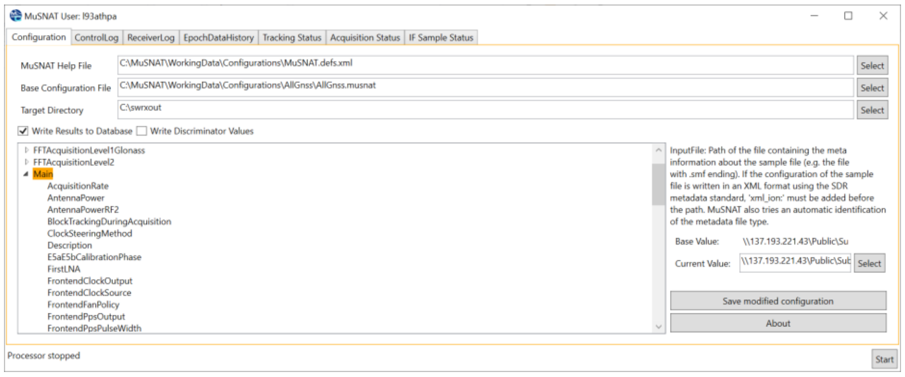
The existing software receiver ipexSR was realized as a Visual Studio 2013 project and used Qt for the graphical user interface. Especially the user interface had some hard constraints regarding flexibility and introduction of new data structures and thus the project was refactored and the user interface was developed from scratch in C# (see sect. “User Interface and SQL data storage”). Furthermore, the following key libraries are used to provide the necessary algorithmic and computing background:

- Intel Performance Primitives for signal processing on the CPU
- nVidia Compute Unified Device Architecture (CUDA) for signal processing on the GPU
- RTK-LIB for RTK/PPP processing (and later for GNSS/IMU/LiDAR-filter)
- Point cloud library (PCL) for LiDAR data management and processing
- Eigen for generic for matrix operations
- SQLite for data archiving

Adding libraries to a larger software project is often a nontrivial task due to mutual conflicts between the libraries that are often hard to understand and to correct. For MuSNAT we made use of the Microsoft vcpkg system that ensures to a large extend the compatibility between various open source and commercial libraries but unfortunately not to 100 % and a few manual adaptations were necessary [4]. As development environment Microsoft Visual Studio 2017 was chosen and as primary target operating system Windows 10. The use of Windows 10 and the use of C# for the user interface implies various further libraries which are partly described in the sect. “User Interface and SQL data storage”.

It was realized that the requirement of real-time operation and extensive post-processing analysis is better fulfilled when separating those features into two different programs: one doing the actual processing and the other one for visualizing and analyzing the logged data. The first one is called MuSNAT-Core and screenshots are shown in Fig. 1 and Fig. 2. The second one, MuSNAT-Analyzer, is shown in Fig. 3. and Fig. 4. There is a unidirectional data flow from the Core to the Analyzer via an SQL database. For each processing run of the Core one individual database file is created which can then be selected by the Analyzer.

MuSNAT-Core is configured by a configuration file which contains all processing parameters and by defining the output directory. The configuration files can be quite large and a help file provides a short explanation for each parameter. After loading the configuration file, the processing can be started. Depending on the input source, it is post-processing or real-time. If used in post-processing the software behaves identical to real-time. During processing the relevant logging information is displayed in various tabs. Currently input data (Intermediate Frequency (IF) samples), acquisition and tracking results (e.g. Fig. 2) are displayed for the current epoch. The current frame of LiDAR data can also be displayed (Fig. 11). Further extensions are planned like e.g. multi-correlator display.



MuSNAT Help File: C:\MuSNAT\WorkingData\Configurations\MuSNAT.defs.xml

Base Configuration File: C:\MuSNAT\WorkingData\Configurations\AllGnss\AllGnss.musnat

Target Directory: C:\swrxout

[✓ Write Results to Database] Write Discriminator Values

- FFTAcquisitionLevel1Glonass
- FFTAcquisitionLevel2
- Main
  - AcquisitionRate
  - AntennaPower
  - AntennaPowerRF2
  - BlockTrackingDuringAcquisition
  - ClockSteeringMethod
  - Description
  - E5aE5bCalibrationPhase
  - FirstLNA
  - FrontendClockOutput
  - FrontendClockSource
  - FrontendFanPolicy
  - FrontendPpsOutput
  - FrontendPpsPulseWidth

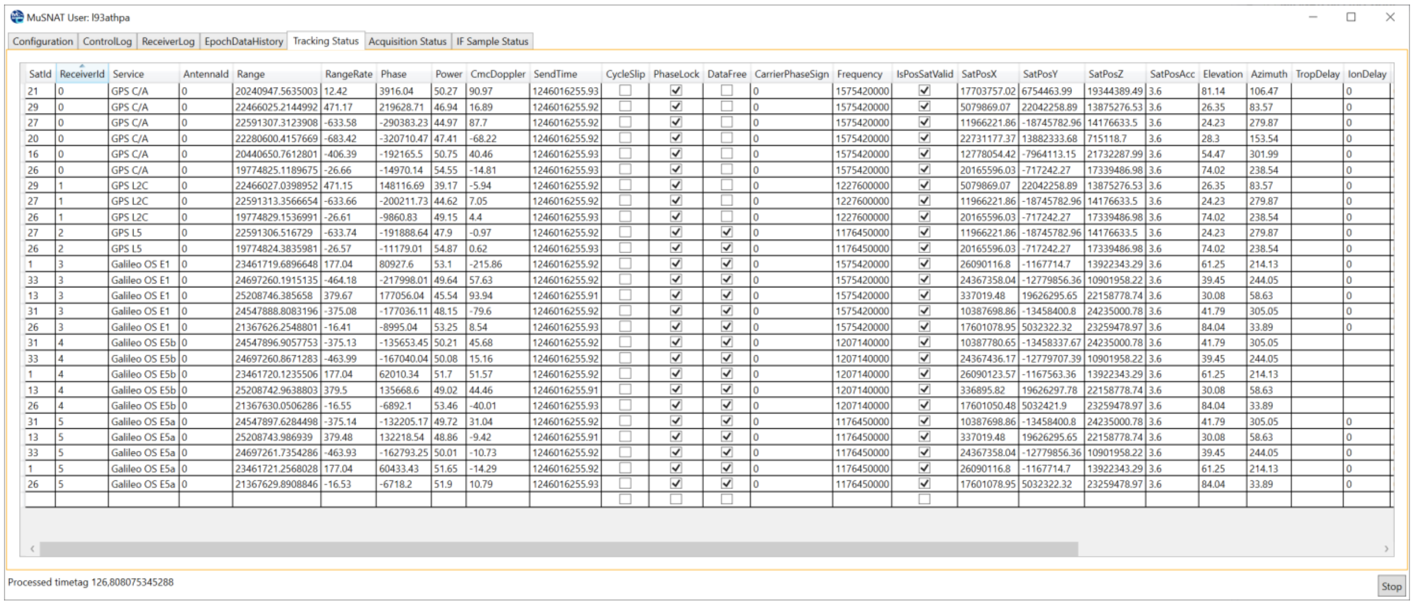
InputFile: Path of the file containing the meta information about the sample file (e.g. the file with .smf ending). If the configuration of the sample file is written in an XML format using the SDR metadata standard, 'xml_ion:' must be added before the path. MuSNAT also tries an automatic identification of the metadata file type.

Base Value: \\137.193.221.43\Public\Su

Current Value: \\137.193.221.43\Public\Sub

Processor stopped

Fig. 1: Screenshot of MuSNAT-Core, start and configuration window



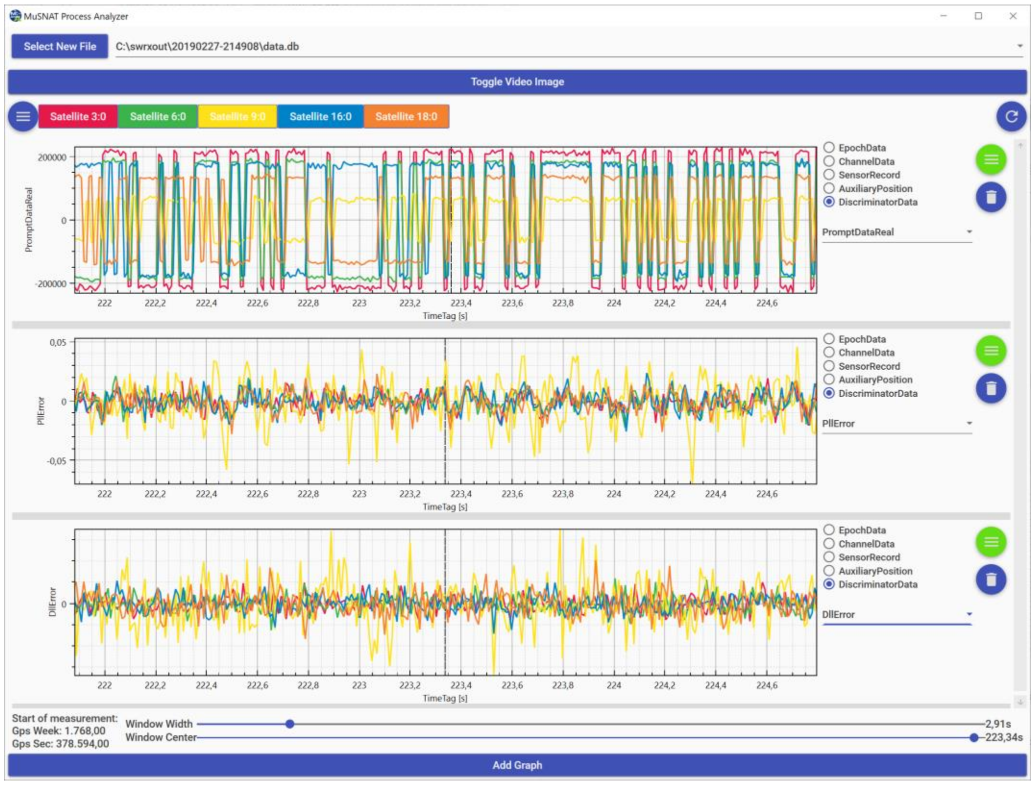
SatId	ReceiverId	Service	AntennaId	Range	RangeRate	Phase	Power	CmcDoppler	SendTime	CycleSlip	PhaseLock	DataFree	CarrierPhaseSign	Frequency	IsPosSatValid	SatPosX	SatPosY	SatPosZ	SatPosAcc	Elevation	Azimuth	TropDelay	IonDelay
21	0	GPS C/A	0	20240947.5635003	12.42	3916.04	50.27	90.97	1246016255.93	☐	✓	☐	0	1575420000	✓	17703757.02	6754463.99	19344389.49	3.6	81.14	106.47		0
29	0	GPS C/A	0	22466025.2144992	471.17	219628.71	46.94	16.89	1246016255.92	☐	✓	☐	0	1575420000	✓	5079869.07	22042258.89	13875276.53	3.6	26.35	83.57		0
27	0	GPS C/A	0	22591307.3123908	-633.58	-290383.23	44.97	87.7	1246016255.92	☐	✓	☐	0	1575420000	✓	11966221.86	-18745782.96	14176633.5	3.6	24.23	279.87		0
20	0	GPS C/A	0	22280600.4157669	-683.42	-320710.47	47.41	-68.22	1246016255.92	☐	✓	☐	0	1575420000	✓	22731177.37	13882333.68	715118.7	3.6	28.3	153.54		0
16	0	GPS C/A	0	20440650.7612801	-406.39	-192165.5	50.75	40.46	1246016255.93	☐	✓	☐	0	1575420000	✓	12778054.42	-7964113.15	21732287.99	3.6	54.47	301.99		0
26	0	GPS C/A	0	19774825.1189675	-26.66	-14970.14	54.55	-14.81	1246016255.93	☐	✓	☐	0	1575420000	✓	20165596.03	-717242.27	17339486.98	3.6	74.02	238.54		0
29	1	GPS L2C	0	22466027.0398952	471.15	148116.69	39.17	-5.94	1246016255.92	☐	✓	☐	0	1227600000	✓	5079869.07	22042258.89	13875276.53	3.6	26.35	83.57		0
27	1	GPS L2C	0	22591313.3566654	-633.66	-200211.73	44.62	7.05	1246016255.92	☐	✓	☐	0	1227600000	✓	11966221.86	-18745782.96	14176633.5	3.6	24.23	279.87		0
26	1	GPS L2C	0	19774829.1536991	-26.61	-9860.83	49.15	4.4	1246016255.93	☐	✓	☐	0	1227600000	✓	20165596.03	-717242.27	17339486.98	3.6	74.02	238.54		0
27	2	GPS L5	0	22591306.516729	-633.74	-191888.64	47.9	-0.97	1246016255.92	☐	✓	✓	0	1176450000	✓	11966221.86	-18745782.96	14176633.5	3.6	24.23	279.87		0
26	2	GPS L5	0	19774824.3835981	-26.57	-11179.01	54.87	0.62	1246016255.93	☐	✓	✓	0	1176450000	✓	20165596.03	-717242.27	17339486.98	3.6	74.02	238.54		0
1	3	Galileo OS E1	0	23461719.6896648	177.04	80927.6	53.1	-215.86	1246016255.92	☐	✓	✓	0	1575420000	✓	26090116.8	-1167714.7	13922343.29	3.6	61.25	214.13		0
33	3	Galileo OS E1	0	24697260.1915135	-464.18	-217998.01	49.64	57.63	1246016255.92	☐	✓	✓	0	1575420000	✓	24367358.04	-12779856.36	10901958.22	3.6	39.45	244.05		0
13	3	Galileo OS E1	0	25208746.385658	379.67	177056.04	45.54	93.94	1246016255.91	☐	✓	✓	0	1575420000	✓	337019.48	19626295.65	22158778.74	3.6	30.08	58.63		0
31	3	Galileo OS E1	0	24547888.8083196	-375.08	-177036.11	48.15	-79.6	1246016255.92	☐	✓	✓	0	1575420000	✓	10387698.86	-13458400.8	24235000.78	3.6	41.79	305.05		0
26	3	Galileo OS E1	0	21367626.2548801	-16.41	-8995.04	53.25	8.54	1246016255.93	☐	✓	✓	0	1575420000	✓	17601078.95	5032322.32	23259478.97	3.6	84.04	33.89		0
31	4	Galileo OS E5b	0	24547896.9057753	-375.13	-135653.45	50.21	45.68	1246016255.92	☐	✓	✓	0	1207140000	✓	10387780.65	-13458337.67	24235000.78	3.6	41.79	305.05		
33	4	Galileo OS E5b	0	24697260.8671283	-463.99	-167040.04	50.08	15.16	1246016255.92	☐	✓	✓	0	1207140000	✓	24367436.17	-12779707.39	10901958.22	3.6	39.45	244.05		
1	4	Galileo OS E5b	0	23461720.1235506	177.04	62010.34	51.7	51.57	1246016255.92	☐	✓	✓	0	1207140000	✓	26090123.57	-1167563.36	13922343.29	3.6	61.25	214.13		
13	4	Galileo OS E5b	0	25208742.9638803	379.5	135668.6	49.02	44.46	1246016255.91	☐	✓	✓	0	1207140000	✓	336895.82	19626297.78	22158778.74	3.6	30.08	58.63		
26	4	Galileo OS E5b	0	21367630.0506286	-16.55	-6892.1	53.46	-40.01	1246016255.93	☐	✓	✓	0	1207140000	✓	17601050.48	5032421.9	23259478.97	3.6	84.04	33.89		
31	5	Galileo OS E5a	0	24547897.6284498	-375.14	-132205.17	49.72	31.04	1246016255.92	☐	✓	✓	0	1176450000	✓	10387698.86	-13458400.8	24235000.78	3.6	41.79	305.05		0
13	5	Galileo OS E5a	0	25208743.986939	379.48	132218.54	48.86	-9.42	1246016255.91	☐	✓	✓	0	1176450000	✓	337019.48	19626295.65	22158778.74	3.6	30.08	58.63		0
33	5	Galileo OS E5a	0	24697261.7354286	-463.93	-162793.25	50.01	-10.73	1246016255.92	☐	✓	✓	0	1176450000	✓	24367358.04	-12779856.36	10901958.22	3.6	39.45	244.05		0
1	5	Galileo OS E5a	0	23461721.2568028	177.04	60433.43	51.65	-14.29	1246016255.92	☐	✓	✓	0	1176450000	✓	26090116.8	-1167714.7	13922343.29	3.6	61.25	214.13		0
26	5	Galileo OS E5a	0	21367629.8908846	-16.53	-6718.2	51.9	10.79	1246016255.93	☐	✓	✓	0	1176450000	✓	17601078.95	5032322.32	23259478.97	3.6	84.04	33.89		0
										☐	☐	☐			☐								

Processed timetag 126,808075345288

Fig. 2: Screenshot of MuSNAT-Core, tracking window

The analyzer reads the logged data from the SQL database and provide several different analysis windows for data at signal level (spectra), Position Velocity and Time (PVT) level, at ranging level, and tracking level and at sensor data level. The considered satellite signals can be selected, and the data is displayed in time-synchronized plots. For example, Fig. 3 shows typical tracking data at channel level. The corresponding position on a map (e.g. Fig. 4), as well as the corresponding video image can also be selected. Further planned extensions are data analysis function for the selected graphs (mean, trend, root mean square (rms), …), multi-correlator plots as well as waterfall plots for the GNSS spectra to identify interference events.

To ensure software stability a build server has been setup which runs the Jenkins system for automated builds and tests after each new commit of source code or configuration parameters. MuSNAT-Core and the MuSNAT-Analyzer are finally produced as installer packages such that they can be used on any Windows 10 system without any prerequisites. They are delivered with the respective reference configuration that are also used for the automated tests.



C:\swrxout\20190227-214908\data.db

EpochData / ChannelData / SensorRecord / AuxiliaryPosition / [✓ DiscriminatorData] — PromptDataReal

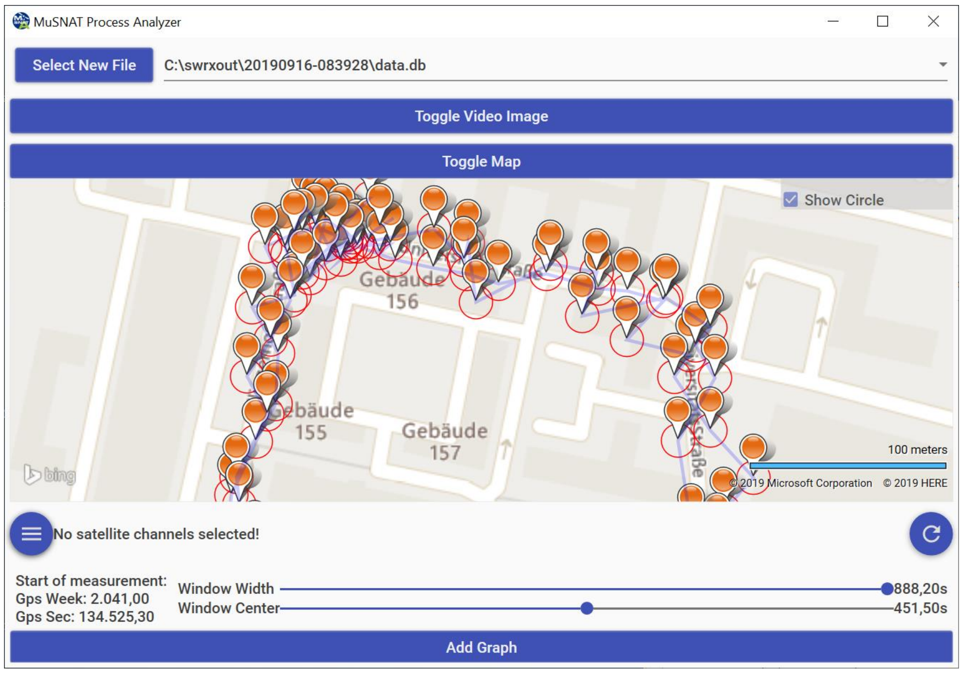
EpochData / ChannelData / SensorRecord / AuxiliaryPosition / [✓ DiscriminatorData] — PllError

EpochData / ChannelData / SensorRecord / AuxiliaryPosition / [✓ DiscriminatorData] — DllError

Start of measurement:
Gps Week: 1.768,00
Gps Sec: 378.594,00

Window Width 2,91s
Window Center 223,34s

Fig. 3: Screenshot of MuSNAT-Analyzer (alpha-version) with correlator, Phase Lock Loop (PLL) discriminator and Delay Lock Loop (DLL) discriminator values for several GPS C/A signals



C:\swrxout\20190916-083928\data.db

[✓ Show Circle]

No satellite channels selected!

Start of measurement:
Gps Week: 2.041,00
Gps Sec: 134.525,30

Window Width 888,20s
Window Center 451,50s

Fig. 4: Screenshot of MuSNAT-Analyzer (alpha-version) with Map-View

INPUT DATA

The legacy software ipexSR used as input data only GNSS IF samples to acquire and track the signals and to compute the PVT solution. The number of input sources for MuSNAT has been extended considerable and not only includes data from other sensors like IMU, LiDAR or camera data but also (for a software receiver) nonstandard input sources like Receiver Independent Exchange Format (RINEX) observation data or trajectories. The input sources are described in the following sub-sections.

Support of ION-Software Defined Radio (SDR) sample standard

The current standard for IF samples [5] is now fully supported in addition to the legacy reading routines from separated IF sample files (one for each frequency band). This enhancement gives us the possibility to process data from virtually any GNSS signal source or from signals of opportunity like Long Term Evolution (LTE) signals. The ion-sdr reader is still slower than our internal reading routine but future work on it will also target the ion-sdr reader performance improvement.

Generic real-time frontend interface via TCP/IP

The legacy software receiver ipexSR provided real-time data access via hard-coded low-level Universal Serial Bus (USB) 2.0 functions to one Radio Frequency (RF) frontend from Fraunhofer IIS and to the NavPort-4 frontend from IFEN. To increase the flexibility of the real-time operation a more generic Transmission Control Protocol/Internet Protocol (TCP/IP) based interface was introduced thus realizing an open real-time interface to virtually any frontend. The interface is implemented as TCP/IP client within MuSNAT-Core that connects to the signal source (i.e. the frontend) which realizes a TCP/IP-server. The server (SDR frontend) and the client (software receiver) can run both on one PC or can be separated on different locations via external TCP/IP connection.

To start the development of the server a first TCP/IP sample server written C++ has been developed which reads its data from a file and this C++ code will serve as template for servers integrating Universal Software Radio Peripheral (USRP) SDRs from National Instruments. Thereby a dual-band 20 MHz/16-bit I/Q sampling rate capability will be established in the first step (Q4/2019). The server for the USRP frontend is programmed with LabVIEW and runs as one process in the background continuously. In later extensions, the USRP frontend will provide four bands with a I/Q sampling rate of up to 80 MHz each with a selectable bit-depth of 2, 4, 8 or 16 bits. Also, other sample servers using low-cost frontends (Hack-RF or rtl-sdr) are planned. As primary RF frontend we focus on the USRP 2955 as it provides four phase-aligned RF-shielded superheterodyne receiver channels up to a frequency of 6 GHz and each with an 80 MHz bandwidth. Furthermore, sensors can be attached to this frontend via a low-level hardware connection thus allowing a tight synchronization to the GNSS signals. To fully exploit all those features, further extensions of the server software are necessary, which is also including onboard Field Programmable Gate Array (FPGA) programming supported by LabVIEW directly to e.g. reduce the bit width for a lower data traffic to the Personal Computer (PC) or adding pre-correlation RF interference mitigation techniques. The USRP 2955 is connected to the processing PC via PCIe card (NI PCIe-8371) over an MXI-Express x4 copper cable. The four channels will be fed by a 4-port splitter (Mini-Circuits® ZN4PD-272-S+) and a bias tee (Mini-Circuits® ZFBT-4R2GW-FT+), which gets the power supply for an active antenna from the GPS disciplined clock of the USPR itself (5V, max. 140 mA). The USRP connected to a PC and a schematic of the concept is given in Fig. 5.

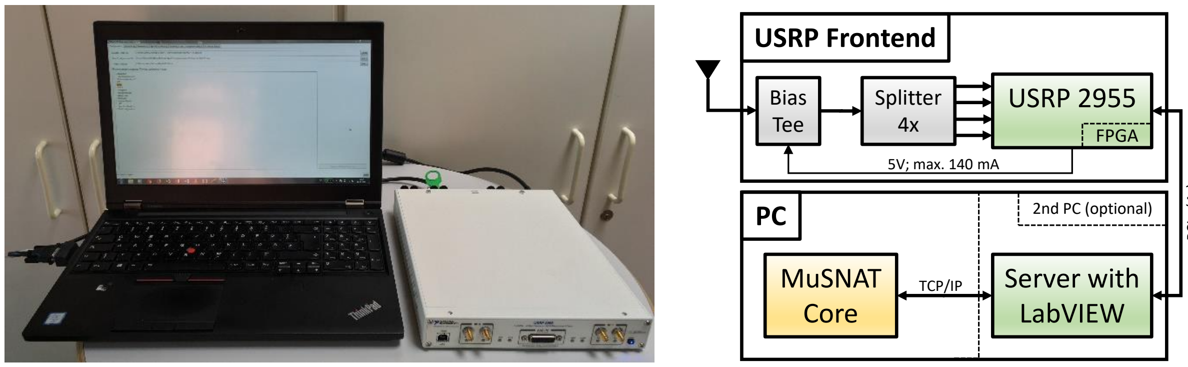


Fig. 5: NI USRP 2955 connected to a processing PC

The sample server uses TCP/IP sockets to transfer the binary data stream of the requested file to the client. The server acts like a file on the file system. It supports the operations “Open File”, “Seek File”, “Read File” and “Disconnect”. The client within MuSNAT is implemented in C++ and uses the TCP socket from boost::asio [6]. The above mentioned prototype server was built in C#, using System.Net.Sockets from the .NET framework.

IF-sample-less operation of the receiver using RINEX-data as input
As far as numerous standard post-processing tasks are concerned, IF samples are not always the type of data available, especially when it comes to data acquired by commercial off-the-shelf (COTS) receivers. To be able to evaluate GNSS raw data as RINEX files from any receiver in a feasible way (e.g. without the overhead in processing time caused by the IF signal processing compartment), IF sample processing is circumvented by simulating a dummy (empty) IF-Stream, signal processing is deactivated, and raw data is fed directly into the positioning modules. This is also the method of choice when using MuSNAT to analyze raw data from Android smartphones [7].

Support of LiDAR data via PCL
The implementation of LiDAR 3D-data will enhance the navigation solution along with additional sources like GNSS/INS/camera. Additionally, it helps to estimate the position in GNSS-denied environments, as well as 3D mapping of the environment. The open source point-cloud-library (PCL) is currently integrated into the MuSNAT [8]. The current version of PCL in the MuSNAT is 1.9.1, which adds more features and stability with newer versions of CUDA, specifically with versions 9.x and 10.0.

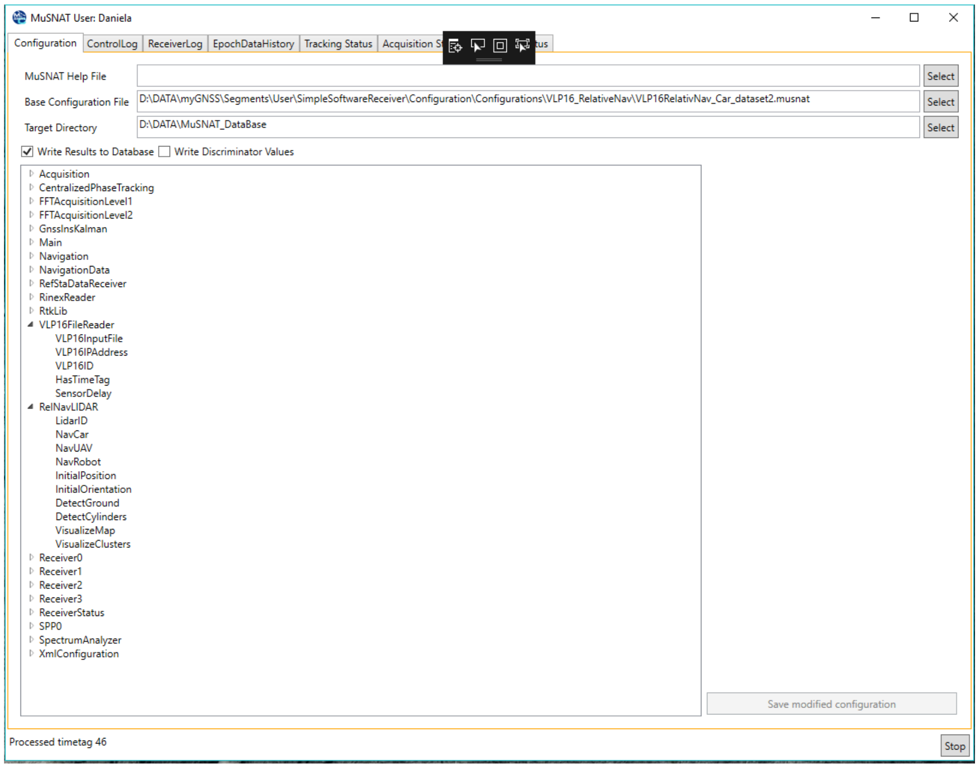
The specific LiDAR sensor supported by MuSNAT is the Velodyne VLP-16 [9]. The VLP-16 has 16 lasers that spin 360 degrees to scan the surrounding environment. It acquires information like range (up to 100 meters), calibrated reflectivities and the exact timing of the firing of each data point, among others. The VLP-16 can operate in different return modes, the dual return mode is not yet supported by MuSNAT. The VLP-16 is connected with ethernet to the PC running MuSNAT and a dedicated GNSS receiver is connected to the sensor allowing the synchronization within the software receiver. Future updates of MuSNAT target to remove this extra GNSS receiver and to synchronize the LiDAR data directly with GNSS samples via either via a hardware connection over the USRP frontend or via a low-level software connection. LiDAR data can also be read from a file having the same format as the real-time data stream. The retrieval of the data can be done off- and on-line, which is particularly useful to test navigation algorithms both in post-processing and real-time. For this purpose, boost::asio and pcap libraries are used in order to read the streamed data from the sensor or from PCAP files.

The data is clustered by frames and stored in point cloud objects for further processing (e.g. to be used in the relative navigation module). The point cloud object is a structure that represents a multidimensional point, traditionally Euclidean x,y,z coordinates, but it can also contain more values e.g. Red Green Blue (RGB), intensity, range, normal coordinates, etc. The software also has the capability of saving the data in PCD (Point Cloud Data) format. For the visualization of the LiDAR data OpenCV cv::viz [10] is used.

In the near future, the capability of building intensity images (2D) as a separate channel for feature extraction is contemplated, as it has been proofed to be a useful alternative to conventional registration methods [11], especially when RGB data is not available. The implementation within the MuSNAT is planned to handle more than one LiDAR device at a time. In Fig. 6, one can see some of the parameters that can be configured for the retrieval of data and its processing.

Support of camera data via OpenCV
Long-term plans include navigation aiding with further visual sensors, e.g. cameras. The basic capability of time-tagging and storing video frames for GNSS-aligned epochs was already added, rendering them for further feature extraction and frame-to-frame relative navigation within the core navigation modules. Currently the data handling and storage routines are using the OpenCV library for efficiency [12]. Respective camera frames can be visualized in the MuSNAT-Analyzer respectively. This is a useful feature for data analysis and interpretation if respective video material (by e.g. a zenith looking fish-eye camera) is captured simultaneously.

Inertial Data
Inertial data may be provided to the MuSNAT to enable the GNSS/INS module(s). Currently a proprietary format for inertial data is required, consisting of the data packet counter, a Universal Time Control (UTC) time stamp, the accelerometer measurements as acceleration in [m/s²], the gyroscope measurements as turn rates in either [rad/s] or [deg/s] and magnetometer measurements, although the latter are not used yet but were included for possible further necessity. Data synchronization with respect to the GNSS measurements is done according to the UTC timestamp of the inertial data. Known hardware delays within the INS regarding the timestamping of the data, e.g. from sensor calibration, may be specified in the MuSNAT configuration as an offset of the inertial data timestamp and are corrected before the actual processing.



MuSNAT User: Daniela

```
MuSNAT Help File
Base Configuration File   D:\DATA\myGNSS\Segments\User\SimpleSoftwareReceiver\Configuration\Configurations\VLP16_RelativeNav\VLP16RelativNav_Car_dataset2.musnat
Target Directory          D:\DATA\MuSNAT_DataBase
[✓] Write Results to Database  [ ] Write Discriminator Values

▷ Acquisition
▷ CentralizedPhaseTracking
▷ FFTAcquisitionLevel1
▷ FFTAcquisitionLevel2
▷ GnssInsKalman
▷ Main
▷ Navigation
▷ NavigationData
▷ RefStaDataReceiver
▷ RinexReader
▷ RtkLib
◢ VLP16FileReader
    VLP16InputFile
    VLP16IPAddress
    VLP16ID
    HasTimeTag
    SensorDelay
◢ RelNavLIDAR
    LidarID
    NavCar
    NavUAV
    NavRobot
    InitialPosition
    InitialOrientation
    DetectGround
    DetectCylinders
    VisualizeMap
    VisualizeClusters
▷ Receiver0
▷ Receiver1
▷ Receiver2
▷ Receiver3
▷ ReceiverStatus
▷ SPP0
▷ SpectrumAnalyzer
▷ XmlConfiguration

Processed timetag 46
```

Fig. 6: Example of the configuration parameters available in the LiDAR related modules.

OUTPUT DATA

MuSNAT-Core provides processing results at four different levels and formats:

- Standard output formats like RINEX observation and navigation files or PVT solution in National Marine Electronics Association (NMEA) or Comma-separated values (CSV) format
- Proprietary American Standard Code for Information Interchange (ASCII) formats for decoded navigation message or RTK, GNSS/IMU, … positioning solution
- Low-level ASCII logging outputs like tracking logs (*.dumplog) or acquisition logs
- Integrated logging of all data above into a SQL database

Each data format has it advantages and provides different analysis options. In terms of GNSS signal processing capability the SQL-database provides the most complete overview and is also easier to handle as only a single file is involved.

GNSS SIGNAL PROCESSING

For an integrated navigation system, a high quality of the GNSS signal processing is essential to obtain optimum results. Small ranging errors and robustness against cycle slips are for example essential for RTK or PPP processing. In the context of sensor fusion, tracking is considered to be more important than acquisition and thus the focus shall be laid on it in this paper. Already within the ipexSR a number of different signal tracking algorithms have been implemented and tested [3]. With various numerical optimizations and approximations, like e.g. use of an unsigned 8-bit representation of IF samples a standard desktop CPU could realize a huge number of tracking channels (>1000) for moderate signal bandwidths (like e.g. 20 MHz). [14] Using 1-bit correlators provides an even larger number of channels or allows use of the software on embedded devices.

However, for MuSNAT we identified three design drivers for signal tracking that are partly incompatible of the above-mentioned approximate approaches:

- Utmost precision
- Textbook like implementation
  - to minimize implementation losses
  - for good traceability of the achieved performance
- Ease of implementation for new GNSS signals

It was therefore decided to focus within MuSNAT on a conventional correlator-based (early, prompt, late, very-early, ,…) tracking scheme and on a 16-bit representation for IF samples and replica signals. Correlator based tracking and its performance is described in [13]. To consolidate this performance a dedicated verification campaign with Galileo E1BC, E5a and E5b as well as GPS L1, L2CM and L5 signal has been conducted and the results are reported here. Furthermore a wide-band AltBOC tracking capability has been added. It also turned out that with this textbook like implementation the number of achievable real-time channels dropped down and thus an alternate processing unit for signal tracking was found in the GPU. The GPU implementation strategy and constraints are reported below. More enhanced signal analysis methods (e.g. chip estimation, ranging authentication) require GNSS signal samples to be logged after carrier wipe-off. This logging function is also described below.

Verification of Tracking Accuracy with Zero-Baseline Tests

The performance analysis of MuSNAT was realized by performing a zero-baseline test and then analyzing the code and phase residuals as functions of changing carrier-to-noise ratio. The experimental setup consisted of an IFEN SX3 RF frontend and a Trimble NetR9 receiver connected to the same receiving antenna located on the roof-top of the observatory building at ISTA. The SX3 frontend recorded IF samples at a sampling rate of 20 MHz for four frequency bands (1.5754 GHz, 1.2276 GHz, 1.17645 GHz and 1.20714 GHz) which allowed signal capture of GPS L1, L2CM and L5 as well as Galileo E1BC, E5a and E5b. The IF samples from SX3 were logged on a desktop PC through USB for later processing with MuSNAT to generate the RINEX observation file. The SX3 frontend and MuSNAT together can thus be considered as one receiver unit of this experimental setup with the other being the Trimble NetR9. Further verification tests are currently being conducted with the NI USRP frontend, and results will be presented on follow-up publications. For given similar sampling parameters we expect the results to be largely independent of the used frontend and they verify mostly the MuSNAT internal tracking algorithms.

A data set of 1 hour was recorded for the experiment. For post-processing, double-difference measurements were computed using RTK-LIB from the RINEX observation files, obtained from both receiver units. The RTK-LIB campaign was configured to be in static mode with the Trimble NetR9 set as the reference receiver and the frequencies were set to L1+L2+L5+E5b to utilize all available signals. Thereby, a position fix was achieved for 99.2 % of the complete duration. The code/phase residuals and carrier-to-noise ratios were exported as text files for each signal and MATLAB was used to obtain the code/phase noise plots (see Fig. 13 - Fig. 16). Each point in the plots of Fig. 13 - Fig. 16 is the standard deviation of the code/phase residual time series against the average C/N0 over an interval of 30 s.

Overall, the results illustrate a good tracking performance with similar tracking results for L1 and E1BC with code noise in the decimeter level for carrier-to-noise ratios greater than 43 dB-Hz. E5a and E5b exhibit even better performance with code noise below 50 cm for carrier-to-noise ratios greater than 46 dB-Hz. Moreover, L5 exhibits the slowest rise in code noise with decreasing C/No that indicate best code noise performance. L2CM exhibits the steepest rise in phase noise with decreasing C/No that indicates worst phase noise performance.

As part of receiver performance verification of MuSNAT a comparison between the measured and analytic code discriminator noise was also performed. MuSNAT provides the option of generating satellite specific data logs of various receiver related parameters including code, phase and frequency discriminator noise during the tracking process (Fig. 17). These can then be plotted for a given satellite and then can be compared against analytical values for the same Signal to Noise Ratio (SNR) level to verify the tracking performance of receiver loops. As part of the campaign, the code discriminator noise value of each signal, for a duration of 500 epochs, was obtained from the log for one satellite. The analytic code discriminator noise value was then computed for each signal using the semi-analytic verification methodology presented in [13]. Fig. 17 shows the loop discriminator noise plots generated for each signal. Table 1 shows the comparison between measured and analytic values.

Table 1: Code noise for the considered verification signals

Signal	Coherent Int Time (ms)	SNR (dB)	Code Disc. Noise Measured (m)	Code Disc. Noise Analytic (m)	Percentage Deviation (%)
L1C/A	5	31.52	1.3066	1.2921	1.12
L2C	20	32.51	1.1852	1.1621	1.99
L5	10	34.6	0.2627	0.2762	4.89
E1 BC	4	29.11	0.9656	0.9743	0.89
E5a	4	27.94	0.7367	0.6027	22.23
E5b	4	29.57	0.5986	0.4959	20.71

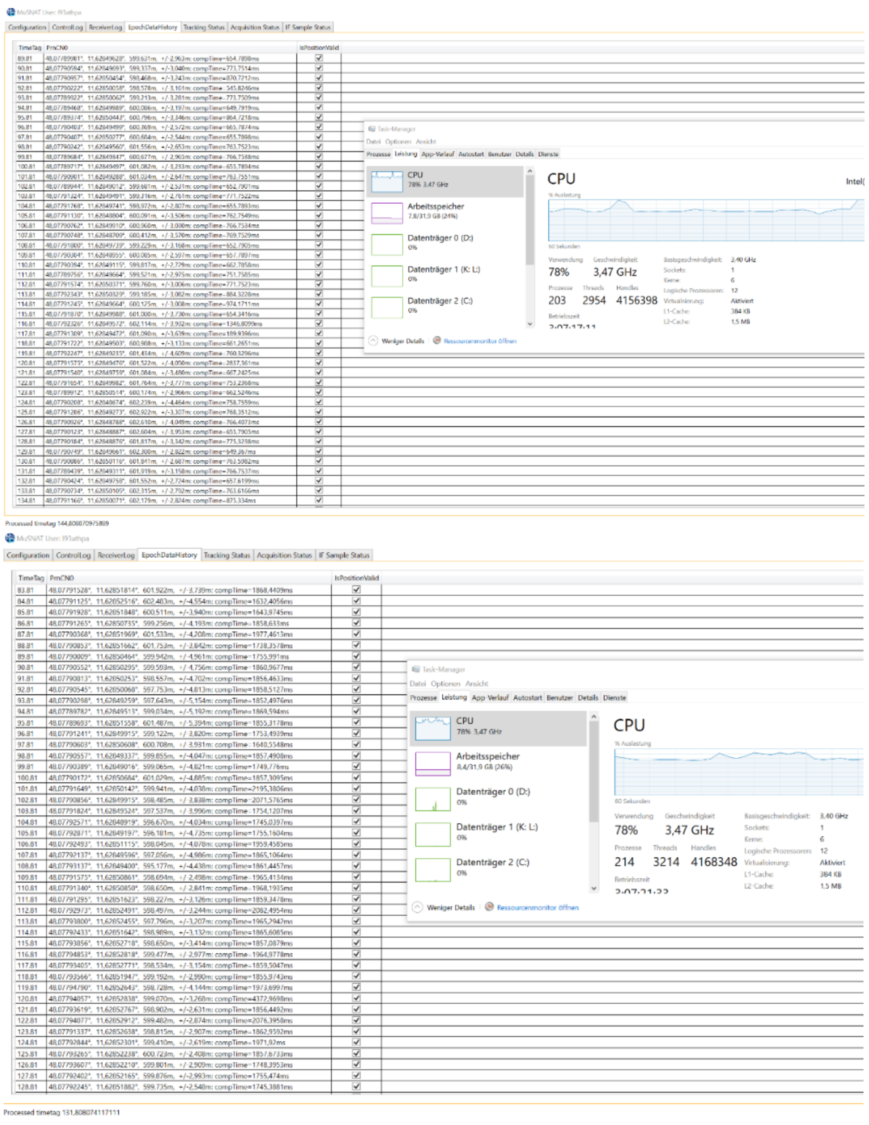
As can be observed, the deviation from the analytic values fall within 5 % for L1 C/A, L2C, L5 and E1BC which indicates effective implementation of correlator-based tracking on MuSNAT. Given the same DLL bandwidth and correlator spacing assigned to the tracking loops of all signals, the larger deviations in E5a and E5b can be attributed to frontend related parameters but further investigation is required to validate this claim.

True-AltBOC tracking capability

In order to have an optimum tracking performance for wideband AltBOC tracking, a precise correlator implementation was implemented for Galileo AltBOC as a new feature in our receiver. This uses both pilot components of the powerful wideband E5 signal. Based on the time synchronization to one of the secondary codes of the Galileo E5aQ or E5bQ components, the true AltBOC tracking can be initialized. Given the two secondary code symbols of E5aQ and E5bQ, the complex composite signal composed out of the E5 pilot components has four possible realizations of a primary period. Based on the timing information of the tracking, the AltBOC correlator can generate the current complex replica. For performance reasons raw replicas for the four possible primary periods are pre-computed including the subcarriers. This largely simplifies the processing and computational load of the AltBOC tracking without scarifying accuracy. In each update step the replicas are resampled based on the current Numerically Controlled Oscillator (NCO) values.

Numerical Tracking Performance on CPU

The correlator-based tracking scheme runs on the CPU of the PC running MuSNAT and utilizes optimized libraries (Intel Integrated Performance Primitives (IPP)) to compute the correlator values. The required dot-product operation is supported on an Intel x86 CPU via SIMD multiply-and-add commands and is a standard function used in many signal processing algorithms. Fig. 7 shows a screen shot of MuSNAT tracking GPS/Galileo L1/L2/L5/E1/E5a/E5b signals on the CPU in fast mode and in precise mode. The basic difference between fast and precise mode is that in precise mode the replica signals are for each integrate-and-dump interval generated newly based on the current NCO values. For the fast mode, replica signals are reused for several integrate-and-dump intervals if the replica signal parameters differ only by a small amount and this amount can be accounted for within the code discriminator value. Fig. 7 shows that the computational effort in precise mode (e.g. 1745 ms to process one 1 second of data) is roughly by a factor of 2.7 larger than in the fast mode (645 ms). This is in accordance with the well-known finding that the required look-up operations on a CPU to generate the signal replica cannot be vectorized [13]. For this test 17 dual-component channels (E1BC, L5, E5a and E5b) plus 9 single-component channels (L1C/A, L2CM) are used each with a sampling rate of 20 MHz and 3-5 correlators. The CPU is an Intel i7-6800K running at 3.4 GHz. It should be noted that for the purposes of this paper no runtime optimization of the configuration has been carried out (compared to e.g. [14]) and the investigation serves merely to outline the relative difference between precise and fast mode.



TimeTag	PrnCN0	IsPositionValid
89.81	48,07789981°, 11,62849628°, 599,631m, +/-2,963m: compTime=654,7898ms	✓
90.81	48,07790594°, 11,62849693°, 599,337m, +/-3,040m: compTime=773,7514ms	✓
91.81	48,07790957°, 11,62850454°, 598,468m, +/-3,243m: compTime=870,7212ms	✓
92.81	48,07790222°, 11,62850058°, 598,578m, +/-3,161m: compTime=545,8246ms	✓
93.81	48,07789922°, 11,62850062°, 599,213m, +/-3,281m: compTime=773,7509ms	✓
94.81	48,07789468°, 11,62849989°, 600,086m, +/-3,197m: compTime=649,7919ms	✓
95.81	48,07789374°, 11,62850443°, 600,796m, +/-3,346m: compTime=864,7218ms	✓
96.81	48,07790403°, 11,62849499°, 600,369m, +/-2,572m: compTime=665,7874ms	✓
97.81	48,07790407°, 11,62850277°, 600,684m, +/-2,544m: compTime=655,7898ms	✓
98.81	48,07790242°, 11,62849560°, 601,556m, +/-2,653m: compTime=763,7523ms	✓
99.81	48,07789684°, 11,62849347°, 600,677m, +/-2,965m: compTime=766,7538ms	✓
100.81	48,07789717°, 11,62849497°, 601,082m, +/-3,233m: compTime=655,7894ms	✓
101.81	48,07790901°, 11,62849288°, 601,034m, +/-2,647m: compTime=763,7551ms	✓
102.81	48,07789944°, 11,62849012°, 599,681m, +/-2,531m: compTime=652,7901ms	✓
103.81	48,07791324°, 11,62849491°, 599,316m, +/-2,761m: compTime=771,7522ms	✓
104.81	48,07791768°, 11,62849741°, 598,972m, +/-2,807m: compTime=655,7893ms	✓
105.81	48,07791130°, 11,62848804°, 600,091m, +/-3,506m: compTime=762,7549ms	✓
106.81	48,07790762°, 11,62849910°, 600,960m, +/-3,030m: compTime=766,7534ms	✓
107.81	48,07790748°, 11,62848709°, 600,412m, +/-3,570m: compTime=769,7529ms	✓
108.81	48,07791800°, 11,62849739°, 599,229m, +/-3,168m: compTime=652,7905ms	✓
109.81	48,07790304°, 11,62848955°, 600,085m, +/-2,597m: compTime=657,7897ms	✓
110.81	48,07790394°, 11,62849115°, 599,817m, +/-2,729m: compTime=662,7858ms	✓
111.81	48,07789756°, 11,62849664°, 599,521m, +/-2,975m: compTime=751,7585ms	✓
112.81	48,07791574°, 11,62850371°, 599,760m, +/-3,006m: compTime=771,7523ms	✓
113.81	48,07792343°, 11,62850329°, 599,185m, +/-3,082m: compTime=884,3228ms	✓
114.81	48,07791245°, 11,62849664°, 600,125m, +/-3,008m: compTime=974,1711ms	✓
115.81	48,07791870°, 11,62849988°, 601,000m, +/-3,730m: compTime=654,3416ms	✓
116.81	48,07792326°, 11,62849572°, 602,114m, +/-3,932m: compTime=1346,8099ms	✓
117.81	48,07791309°, 11,62849472°, 601,090m, +/-3,639m: compTime=189,9396ms	✓
118.81	48,07791722°, 11,62849503°, 600,908m, +/-3,133m: compTime=661,2651ms	✓
119.81	48,07792247°, 11,62849235°, 601,434m, +/-4,609m: compTime=760,3296ms	✓
120.81	48,07791575°, 11,62849476°, 601,522m, +/-4,050m: compTime=2837,361ms	✓
121.81	48,07791540°, 11,62849759°, 601,084m, +/-3,480m: compTime=667,2425ms	✓
122.81	48,07791654°, 11,62849982°, 601,764m, +/-3,777m: compTime=753,2368ms	✓
123.81	48,07789912°, 11,62850514°, 600,174m, +/-2,966m: compTime=662,5246ms	✓
124.81	48,07790208°, 11,62848674°, 602,239m, +/-4,464m: compTime=758,7559ms	✓
125.81	48,07791286°, 11,62849273°, 602,922m, +/-3,307m: compTime=768,3512ms	✓
126.81	48,07790926°, 11,62848788°, 602,610m, +/-4,049m: compTime=766,4073ms	✓
127.81	48,07790123°, 11,62848887°, 602,604m, +/-3,953m: compTime=655,7905ms	✓
128.81	48,07790184°, 11,62848876°, 601,817m, +/-3,342m: compTime=775,3238ms	✓
129.81	48,07790749°, 11,62849661°, 602,300m, +/-2,822m: compTime=649,367ms	✓
130.81	48,07790886°, 11,62850116°, 601,841m, +/-2,687m: compTime=763,5982ms	✓
131.81	48,07789439°, 11,62849311°, 601,919m, +/-3,158m: compTime=766,7537ms	✓
132.81	48,07790424°, 11,62849758°, 601,552m, +/-2,724m: compTime=657,6199ms	✓
133.81	48,07790734°, 11,62850105°, 602,315m, +/-2,792m: compTime=763,6166ms	✓
134.81	48,07791166°, 11,62850071°, 602,179m, +/-2,824m: compTime=875,334ms	✓

CPU
78% 3,47 GHz
Arbeitsspeicher
7,8/31,9 GB (24%)
Datenträger 0 (D:)
0%
Datenträger 1 (K: L:)
0%
Datenträger 2 (C:)
0%

CPU
Verwendung 78%
Geschwindigkeit 3,47 GHz
Prozesse 203
Threads 2954
Handles 4156398
Betriebszeit 2:07:17:11
Basisgeschwindigkeit: 3,40 GHz
Sockets: 1
Kerne: 6
Logische Prozessoren: 12
Virtualisierung: Aktiviert
L1-Cache: 384 KB
L2-Cache: 1,5 MB

Processed timetag 144,808070975889

TimeTag	PrnCN0	IsPositionValid
83.81	48,07791528°, 11,62851814°, 601,922m, +/-3,739m: compTime=1868,4409ms	✓
84.81	48,07791125°, 11,62852516°, 602,483m, +/-4,554m: compTime=1632,4056ms	✓
85.81	48,07791928°, 11,62851848°, 600,511m, +/-3,940m: compTime=1643,9745ms	✓
86.81	48,07791265°, 11,62850735°, 599,256m, +/-4,193m: compTime=1858,633ms	✓
87.81	48,07790368°, 11,62851969°, 601,533m, +/-4,208m: compTime=1977,4613ms	✓
88.81	48,07790853°, 11,62851662°, 601,753m, +/-3,842m: compTime=1738,3578ms	✓
89.81	48,07790009°, 11,62850464°, 599,942m, +/-4,961m: compTime=1755,991ms	✓
90.81	48,07790552°, 11,62850295°, 599,593m, +/-4,756m: compTime=1860,9677ms	✓
91.81	48,07790813°, 11,62850253°, 598,557m, +/-4,702m: compTime=1856,4633ms	✓
92.81	48,07790545°, 11,62850068°, 597,753m, +/-4,813m: compTime=1858,5127ms	✓
93.81	48,07790298°, 11,62849259°, 597,643m, +/-5,154m: compTime=1852,4976ms	✓
94.81	48,07789782°, 11,62849513°, 599,034m, +/-5,192m: compTime=1869,594ms	✓
95.81	48,07789693°, 11,62851558°, 601,487m, +/-5,394m: compTime=1855,3178ms	✓
96.81	48,07791241°, 11,62849915°, 599,122m, +/-3,820m: compTime=1753,4939ms	✓
97.81	48,07790603°, 11,62850608°, 600,708m, +/-3,931m: compTime=1640,5548ms	✓
98.81	48,07790557°, 11,62849337°, 599,855m, +/-4,047m: compTime=1857,4908ms	✓
99.81	48,07790389°, 11,62849016°, 599,065m, +/-4,821m: compTime=1749,776ms	✓
100.81	48,07790172°, 11,62850684°, 601,029m, +/-4,885m: compTime=1857,3095ms	✓
101.81	48,07791649°, 11,62850142°, 599,941m, +/-4,038m: compTime=2195,3806ms	✓
102.81	48,07790856°, 11,62849915°, 598,485m, +/-3,838m: compTime=2071,5765ms	✓
103.81	48,07791824°, 11,62849524°, 597,537m, +/-3,996m: compTime=1754,1207ms	✓
104.81	48,07792571°, 11,62848919°, 596,670m, +/-4,034m: compTime=1745,0397ms	✓
105.81	48,07792871°, 11,62849197°, 596,181m, +/-4,735m: compTime=1755,1604ms	✓
106.81	48,07792493°, 11,62851115°, 598,045m, +/-4,078m: compTime=1959,4585ms	✓
107.81	48,07792137°, 11,62849596°, 597,056m, +/-4,986m: compTime=1865,1064ms	✓
108.81	48,07793137°, 11,62849400°, 595,177m, +/-4,438m: compTime=1861,4457ms	✓
109.81	48,07791575°, 11,62850861°, 598,694m, +/-2,498m: compTime=1965,4134ms	✓
110.81	48,07791340°, 11,62850850°, 598,650m, +/-2,841m: compTime=1968,1935ms	✓
111.81	48,07791295°, 11,62851623°, 598,227m, +/-3,126m: compTime=1859,3478ms	✓
112.81	48,07792973°, 11,62852491°, 598,497m, +/-3,244m: compTime=2082,4954ms	✓
113.81	48,07793800°, 11,62852455°, 597,796m, +/-3,207m: compTime=1965,2942ms	✓
114.81	48,07792433°, 11,62851642°, 598,989m, +/-3,132m: compTime=1865,6085ms	✓
115.81	48,07793856°, 11,62852718°, 598,650m, +/-3,414m: compTime=1857,0879ms	✓
116.81	48,07794853°, 11,62852818°, 599,477m, +/-2,977m: compTime=1964,9778ms	✓
117.81	48,07793405°, 11,62852771°, 598,534m, +/-3,154m: compTime=1859,5047ms	✓
118.81	48,07793566°, 11,62851947°, 599,192m, +/-2,990m: compTime=1855,9743ms	✓
119.81	48,07794790°, 11,62852643°, 598,728m, +/-4,144m: compTime=1973,6997ms	✓
120.81	48,07794057°, 11,62852838°, 599,070m, +/-3,268m: compTime=4372,9698ms	✓
121.81	48,07793619°, 11,62852767°, 598,902m, +/-2,631m: compTime=1856,4492ms	✓
122.81	48,07794077°, 11,62852912°, 599,482m, +/-2,874m: compTime=2076,3958ms	✓
123.81	48,07791337°, 11,62852638°, 598,815m, +/-2,907m: compTime=1862,9592ms	✓
124.81	48,07792844°, 11,62852301°, 599,410m, +/-2,619m: compTime=1971,92ms	✓
125.81	48,07793265°, 11,62852238°, 600,723m, +/-2,408m: compTime=1857,6733ms	✓
126.81	48,07793607°, 11,62852210°, 599,801m, +/-2,909m: compTime=1748,3953ms	✓
127.81	48,07792402°, 11,62852165°, 599,876m, +/-2,993m: compTime=1755,474ms	✓
128.81	48,07792245°, 11,62851882°, 599,735m, +/-2,548m: compTime=1745,3881ms	✓

CPU
78% 3,47 GHz
Arbeitsspeicher
8,4/31,9 GB (26%)
Datenträger 0 (D:)
0%
Datenträger 1 (K: L:)
0%
Datenträger 2 (C:)
0%

CPU
Verwendung 78%
Geschwindigkeit 3,47 GHz
Prozesse 214
Threads 3214
Handles 4168348
Betriebszeit 2:07:21:33
Basisgeschwindigkeit: 3,40 GHz
Sockets: 1
Kerne: 6
Logische Prozessoren: 12
Virtualisierung: Aktiviert
L1-Cache: 384 KB
L2-Cache: 1,5 MB

Processed timetag 131,808074117111

Fig. 7: Screenshot of MuSNAT tracking GNSS signals on the CPU in fast mode (upper) and precise mode (lower), both referring to the configuration shown in Fig. 2

Precise correlator implementation on the GPU

The computational performance of a GPU is typically much higher as for a CPU and thus porting the GNSS signal processing to the GPU can potentially provide a solution to realize an efficient implementation of the precise correlation mode. Using GPUs for GNSS signal processing has already been considered some years ago and several implementations have been reported [15-21]. GPUs have also found use within software GNSS simulators [22-23]. Due to the parallel nature of GPU processing they are particularly attractive for processing data from antenna arrays [23-24]. Other specialized use cases for GPUs are in time transfer [25] or the use of embedded GPUs [26].

Older GPU architectures provided direct hardware support only for 32-bit floating point numbers, which is an inefficient format to represent GNSS signal samples causing a significant memory bandwidth overhead. However, the Turing architecture of nVidia introduced 16-bit computing elements (mostly to support AI-algorithms) and graphics cards like the RTX 2080 Ti provide a computational power of 23.5 TFLOPS (16-bit, half-precision) at a competitive price [27]. The 16-bit float format is not exactly the same as the 16-bit integer format used for CPU signal processing, but due to the high number of bits a loss-less conversion from 16-bit integer to 16-bit float samples can be expected.

The GPU exhibits a highly parallel architecture. For example, on an RTX 2080 Ti a number of 68 so-called symmetric multiprocessors (SM) can be found each having 64 FP32 cores. One FP32 core can execute two 16-bit float multiply-and-add instructions per clock cycle. Assuming a base clock rate of 1350 MHz this yields 1350 MHz x 68 (#SM) x 64 (#FP32) x 2 (two FP16) x 2 (multiply-and-add) = 23.5 TFLOP. Expressed differently, a number of 17408 arithmetic units are running in parallel on that GPU. This high computational power can for example be fully utilized for algorithms which need only a few operands and to not require to access memory (e.g. computation of Mandelbrot sets). To which is extend this computing power can also be exploited for dot-products (e.g. correlation) was assessed with a dedicated benchmark program.

Dot-product computation requires to read the two input-vectors, compute the dot-product and to store the results. For the sake of GNSS signal processing, we have in principle three different locations where the received signal and the replica signal can be stored during the computation of the dot-product (and from where the 17408 compute elements can access the data):

- CPU main memory
- GPU main (global) memory
- GPU (SM-wise) shared memory

The first option would be the most convenient one, as the CPU memory is the standard storage area for all data, but the bandwidth of the GPU-CPU bus (PCIe x 16) is very limited (approx 15.76 GByte/s) and thus clearly not suitable. To assess the performance of the two other options two benchmark programs have been written and the relevant source code is shown in App. B.2 and B.3. The dot-product computation has been executed sufficiently often and has been repeated for different vector lengths (which corresponds for a certain sampling rate to the coherent integration time) and a different number of correlators (each correlator corresponds to two (I/Q) dot-products). The effective computational performance has been measured with timers and is shown in Fig. 8. It is clearly seen, that the performance increases for an increasing number of correlators. The use of shared memory clearly improves the performance, but the code also implies some restrictions as the shared memory architecture of the GPU is limited in size. For example, only a limited number of correlators (though up to 256) are supported with the shared memory code. With (from the GPU-point of view) optimal parameters around 3.5 TFLOP can be achieved but under more realistic conditions 1 TFLOP seems to be a maximum. We state here clearly that further algorithmic research is required to come up with algorithms which better utilize the highly parallel structure of a GPU for dot-product computation. It shall also be mentioned that currently non of the existing libraries (cuBLAS, thrust) provide suitable functions.

After the dot-product performance assessment has been conducted, the code has been integrated into MuSNAT signal tracking. Therefore, a GNSS replica generation engine was developed which takes the current channel NCO parameters and produces a GNSS replica in the GPU global memory. The respective code is shown in App. B.1 and demonstrates one of the advantages of the CUDA based approach. With only a few lines the code and carrier replicas can be generated and multiplied. The code uses no approximations like look-up tables (as it is the case for the CPU implementation) and can be easily linked to the mathematical GNSS signal model. Thus, it is expected that the code is well maintainable and new signal processing methods like meta-signal processing or variable carrier frequency offsets are easily implemented as well as new GNSS/5G signal models.

The resulting GPU-based signal tracking engine within MuSNAT is mathematically equivalent to the CPU based precise mode and produces identical results. The resulting timing including task division on GPU (and CPU) is shown in App. B.4. There a time span of 4 ms (in signal time) is shown while tracking 8 Galileo E1BC signals with 100 MHz sampling rate and 256 correlators per channel. For each signal component a separated GPU stream is created. All signals (IF samples and replicas) reside on the GPU but the tracking itself is controlled by the CPU. It can be seen that the processing performance is around real-time which is suitable for some dedicated specialized (e.g. signal quality monitoring) applications but can be considered as too low for generic multi-frequency multi-GNSS tracking. Thus, we further investigated what the bottle-neck of this implementation could be.

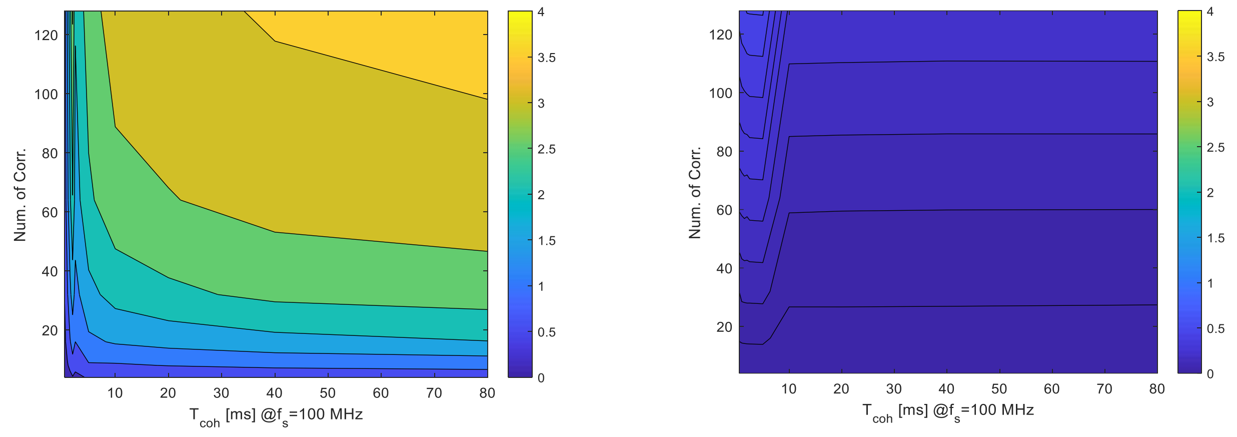


Fig. 8: Achieved TFLOP as a function of coherent integration time T_{coh} and number of correlators for dot-product computation (left: GPU shared memory – App. B.3, right: GPU global memory – App. B.2), RTX 2080 Ti

For GPU signal tracking, three different core tasks/criteria have been identified:

- The CPU has to instruct the GPU to do something (launch-limit)
- GPU to generate replica signals (replica limit)
- GPU to correlate replica with received signal (correlator limit)

For each criteria the number of achievable channels #*N* can be bounded by applying generic constraints. The launch limit says that the number of channels is smaller than the coherent integration time T_{coh} divided by the time T_{ker} to start one task (=kernel) on the GPU multiplied by the number of tasks k required per each coherent integration,

$$\#N < \frac{T_{coh}}{kT_{ker}}. \tag{1}$$

In the presented test case we have a coherent integration time of 4 ms and we have to generate two replica signals (due to the IF sample packet structure), perform two correlations, need to provide for those two replicas the parameters and retrieve the results. These are in total *k*=7 tasks. As a low cost GPU, the RTX2080 Ti supports only the Windows Display Driver Model (WDDM) driver model (and not the TCC model) and thus the typical kernel launch time was estimated to be around 15 µs. With that parameters chosen, the number of channels is limited to be less than 38! The launch limit can be shifted to a higher number of channels if the GPU kernels are started simultaneously for all channels within one launch. This can be done in MuSNAT for all channels belonging to the same signal (e.g. to E1BC). If this is done (e.g. during the next software update in the year 2020) the launch limit increases by the number of signal-components-per-service, e.g. by 24 for 12 E1BC signals, and can then be considered as irrelevant.

Replica generation on the GPU with the code shown in App. B.1 allows to generate replicas with a rate of f_g=25 GSamples/s. This figure has been extracted from App. B.4 showing that a signal time span of 4 ms at 100 MHz sampling rate is generated within approximately 16 µs. The number of channels is then limited by the replica limit given as the ratio of the generation rate divided by the sampling rate f_s,

$$\#N < \frac{f_g}{f_s}. \tag{2}$$

For the used settings, the replica limit is 250 channels. If the replica code is further optimized a higher number of channels deems feasible, but further benchmarking would be required to confirm this conjecture. It could also be that the GPU signal generation is limited by the access to the global GPU memory and in that case the replica limit is independent of the specific code. This would have the advantage that more complex signal models (e.g. requiring more computations at sample level) have the same replica limit.

The correlation limit is the ratio of the computational performance divided by the sample rate, the number of correlators and a factor of 4 accounting for the two multiplications and two additions per sample,

$$\#N < \frac{F}{4 n_c f_s} . \tag{3}$$

If 1 TFLOP is available and assuming 32 correlators and 100 MHz sampling rate results in 78 channels. With the full 23.5 TFLOP available a (hypothetical) number of 1835 channels could be achieved.

To conclude, it can be stated that a first operational GPU tracking engine has been implemented which achieves for a low number of channels real-time performance. It does this virtually independent of the number of correlators (e.g. the processing speed seems to be the same regardless if there are 5 or 200 correlators). This engine may have already now applications for specialized applications like signal-quality-monitoring (monitoring the shape of the correlation function for deformations), anti-spoofing measures via detecting multiple peaks or for Bayesian direct position estimation [28]. If a higher number of channels is sought, multiple GPUs can be installed in a PC, however, it is clearly understood that further code optimization is the way to go. If all the channels within one receiver are launched simultaneously on the GPU a multi-frequency and multi-GNSS configuration should be able to run in real-time on the GPU (and still use many correlators).

FPGA acceleration cards also become more common for use within standard PCs [29]. In principle they will allow a more efficient exploitation of hardware resources as for example for each channel and correlator exactly one multiplication and addition unit (accumulator) can be realized (and you don't need to exploit dozens of them as for the GPU or utilize them in a time-shared manner as on the CPU). However, FPGA cards are more cost intensive, require more dedicated hardware and driver support and are way more complex to program as a GPU (especially data transfer and synchronization). For this reason they are currently not considered to be a cost efficient hardware extension to improve the GNSS signal tracking for MuSNAT.

Extraction of baseband sample snippets

The precise correlator has been modified to log the IF samples in regular intervals to a file after the carrier wipe off has been performed. This is useful for processing of signals where the spreading code is not known. Examples are e.g. GPS Y-code, Galileo Public Related Service (PRS) or signals considered for spreading code authentication [30]. This information can be extracted by using a third-party software, e.g. MATLAB, in order to analyze unknown spreading code sequences. For example, Fig. 9 shows the Galileo E1A signal that has been converted to baseband taking parameters from the respective E1BC channel. As the signal has been captured with a 2.4 m dish antenna, already in the raw logged signal the E1A spreading code sequence is visible. If further filtering is applied (convolution with the chip-form) the E1A chip values can be extracted.

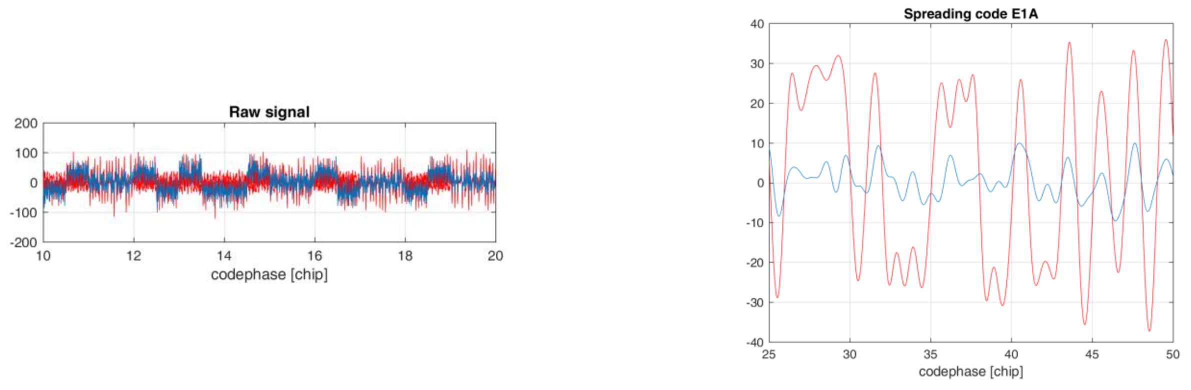


Fig. 9: Baseband signal snippet (left graph) for Galileo PRN 26 on E1 taken by a 2.4 m dish antenna. This data is used to estimate the E1-A chips (right graph).

GNSS SIGNAL GENERATOR (TRANSCEIVER)

Studying and testing new and possible future GNSS signals in combination with sensor fusion capabilities require a signal generator which is flexible and fully modifiable. To overcome the need of implementing a signal generator from scratch, the MuSNAT software receiver was extended to a GNSS-software transceiver to generate and process GNSS signals. The approach to modify the MuSNAT into a software transceiver is based on exploiting the vector tracking feature of the MuSNAT software receiver. The underlying architecture is sketched in Fig. 10. Due to the replacement of the position in the vector tracking loop, it is possible to manipulate the NCO and thereby force the code and carrier generator to generate a signal replica which fits to the induced position. Multiplying the replica with the desired symbol value and the desired amplitude yields an entire line of sight signal. The replica signals of all satellites in tracking match the predefined user trajectory. Saving the added replica signals results in a signal stream at IF-level which can then be converted to an analog RF signal. The MuSNAT-software-transceiver can track, generate, or re-generate tracked signals. A detailed description of the needed modifications of transforming a GNSS-software receiver into a GNSS-software transceiver was presented by Maier et al. [31] with an assessment on the achieved signal quality.

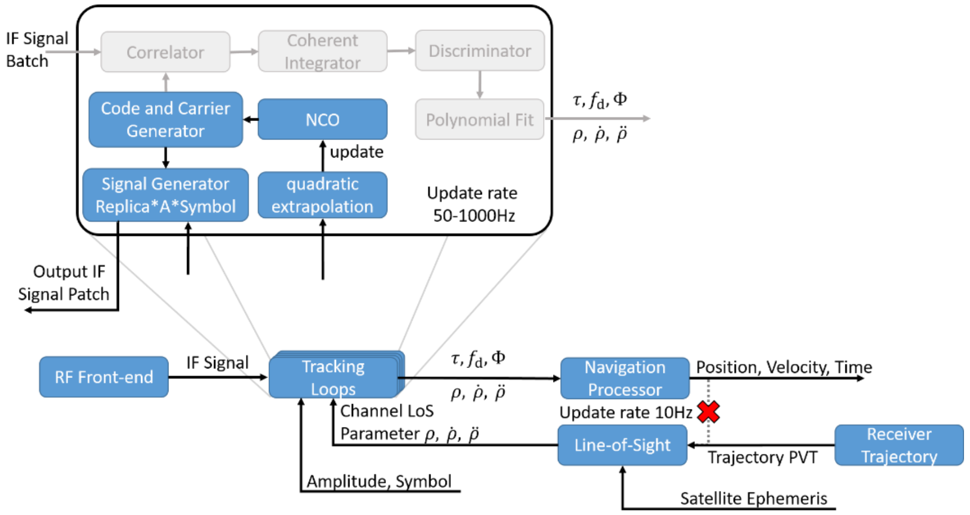


Fig. 10: Sketch of MuSNAT signal generation architecture, using the vector tracking feature [31].

NAVIGATION PROCESSING

MuSNAT provides all the navigation processing options as the ipexSR did and this includes the standard options like single point positioning, GNSS Kalman-filter positioning, Receiver Autonomous Integrity Monitoring (RAIM) (for GPS L1 only). More advanced features are signal quality monitoring (SQM) [32], ionospheric monitoring or Differential Global Navigation Satellite System (DGNSS) positioning. MuSNAT supports a serial navigation processing flow. A list of processing modules is specified which is executed in serial way for each measurement epoch. For example, a typical pipeline may look like: Single Point Positioning (SPP) (for outlier detection), RINEX output, RTK positioning, GNSS/INS filter. In the context of the change from ipexSR to MuSNAT a few advanced modules have been added which are described in the following paragraphs. Due to the modular software structure, extension with further modules is easily possible.

RTK-LIB integration

The MuSNAT Receiver software is now capable of performing RTK positioning. The RTK-LIB integrated in MuSNAT supports GPS/Galileo dual frequency RTK positioning feature [33]. The RTK positioning is supported for IF sample data streams and for direct RINEX 3.0 observation file input. This feature is for example used for the context of Android raw data analysis [7].

Support of inertial and GNSS/INS/LiDAR-integration

With the transition of the ipexSR to the MuSNAT this software is no longer a pure GNSS-software receiver, but rather multi sensor PVT and navigation engine. The basic capability therefore also includes a state of the art Kalman-filter implementation currently consisting of 20 states for attitude, position, velocity, accelerometer- and gyro-biases, clock error, clock error drift, and GPS/Glonass respective GPS/Galileo clock offsets and inter-frequency-biases. Full INS error models can be provided to the filter. Additionally, founding on a cooperation between the Tokyo University of Marine Science (TUMSAT) and our Institute, a GNSS/INS module is being developed for RTK-LIB. It will fully comply with RTK-LIB standards, those being open accessibility, coding style, documentation and scientific basis. The module comprises of a 15 state loosely coupled GNSS/INS filter according to [34] resolved in the local navigation frame and is currently being evaluated. Future work aims at an additional close coupling and real time capability. After completion, the module will be incorporated into RTK-LIB by TUMSAT through a dedicated widget in the Graphical user Interface (GUI) for configuration and evaluation of the results. Furthermore, due to the nature of the RTK-LIB, the module will also be portable as it is part of an already portable software suite. This filter will also yield the basis for a GNSS/INS/LiDAR integration in the future.

Relative Navigation with LiDAR

Due to the implementation of the LiDAR support in the MuSNAT Receiver, now it is possible to process point cloud data, provided by the VLP-16. Relative navigation using LiDAR is based on estimating the ego-position of a moving platform by employing the not-moving objects around the agent [35]. As the latter moves, the position of the objects in the sensor's reference frame changes in the same proportion as the agent's movement. This apparent change in position of the fixed objects provides the real pose variation of the sensor in the environment. In principle, the estimation of the relative navigation of a moving agent (e.g. a car) is based in three main tasks: 1) data preprocessing and object segmentation, 2) object recognition and association, and 3) point clouds registration.

1) As the LiDAR produces raw data, it should be cleaned for a better outcome. Elimination of outliers is required to enhance the integrity of the navigation solution. Some data is purposely discarded for sake of better quality at the results, e.g. the ground point cloud could be eliminated, under the assumption that it is even. A range filter, based on considered distance thresholds in each sensor axis, encloses the most useful points. It is required to subdivide the scene scanned into smaller sets, called point clusters [36]. Each cluster is assumed to be one object in the surroundings. Euclidean segmentation method is employed, where the point cloud is subdivided into smaller sets based on the distance among the points, under a given threshold. A cluster is composed by all the points whose distance to their closest neighbors is lower than the settled limit.

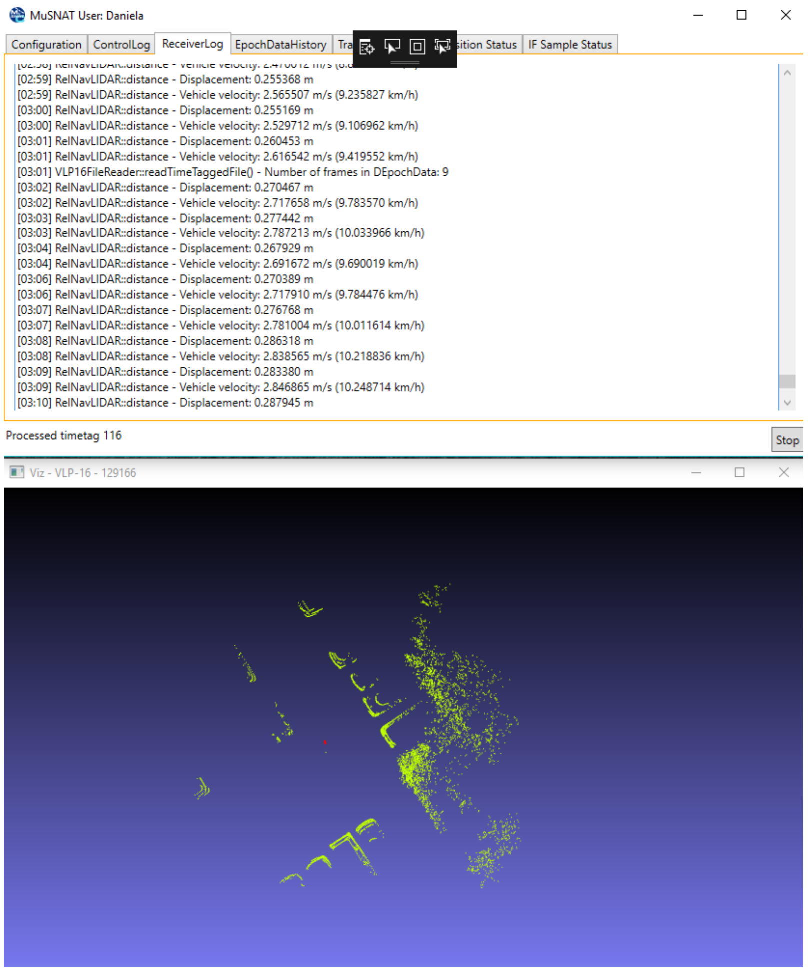
2) Global descriptors play a big role in the object recognition task. After the cluster segmentation at time $t = k$, the Ensemble of Shape Functions (ESF) [37] descriptor is calculated for each of the clusters. The descriptors are stored for further matching against the signatures of the point cloud clusters obtained at time $t = k + 1$. In order to compare the likeliness of two descriptors, the Chi-square distance is calculated among all the stored signatures. There is a match, indicating that both descriptors belong to the same cluster observed at different epochs, when the calculated histogram distance is smaller than a given threshold. Under the certainty of having a cluster identified in two consecutive point data sets, the object association is required in order to link the same cluster in both scans. The latter is obtained by coupling the clusters' centroids of the correspondent matched descriptors.

3) Point clouds registration overlaps the scans in order to perform the most accurate possible alignment between them, finding the correspondent transformation matrix that allows the positioning estimation. The centroids association gives the initial step for the registration procedure based on Singular Value Decomposition (SVD) [38] finding the initial rotation matrix, and then the correspondent translation vector after the rotation transformation. Furthermore, fine alignment aids to reduce this error, making the registration more accurate. For that, the Iterative Closest Point (ICP) [39] algorithm reduces the rotation and translation values in a given number of iterations until the error is an admissible value for the correspondent application. The final estimated agent position in sensor coordinates can be written as

$$x_{k+1} = \begin{bmatrix} R_{\mathrm{ICP}} R_{\mathrm{SVD}} & R_{\mathrm{ICP}} \mathrm{t}_{\mathrm{SVD}} + \mathrm{t}_{\mathrm{ICP}} \\ 0 & 1 \end{bmatrix} x_k, \tag{4}$$

where the final translation vector $t_{f_{3\times1}} = R_{\mathrm{ICP}} \mathrm{t}_{\mathrm{SVD}} + \mathrm{t}_{\mathrm{ICP}}$ indicates the estimated displacement, and the final rotation matrix $R_{f_{3\times3}} = R_{\mathrm{ICP}} R_{\mathrm{SVD}}$ shows the change in orientation. Those two elements are the input for the integration filter. For faster calculations, the rotation matrix R_f is converted into a quaternion q_f [40].

The algorithm has been implemented and is currently under test. In Fig. 11 the debugging output within MuSNAT is displayed for a test drive with the VLP16 mounted on the roof top of our measurement van while driving on a parking lot. Within the context of LiDAR processing it is important to understand which clusters have been identified and matched and thus a graphical debugging output was also added.



```
[02:59] RelNavLIDAR::distance - Displacement: 0.255368 m
[02:59] RelNavLIDAR::distance - Vehicle velocity: 2.565507 m/s (9.235827 km/h)
[03:00] RelNavLIDAR::distance - Displacement: 0.255169 m
[03:00] RelNavLIDAR::distance - Vehicle velocity: 2.529712 m/s (9.106962 km/h)
[03:01] RelNavLIDAR::distance - Displacement: 0.260453 m
[03:01] RelNavLIDAR::distance - Vehicle velocity: 2.616542 m/s (9.419552 km/h)
[03:01] VLP16FileReader::readTimeTaggedFile() - Number of frames in DEpochData: 9
[03:02] RelNavLIDAR::distance - Displacement: 0.270467 m
[03:02] RelNavLIDAR::distance - Vehicle velocity: 2.717658 m/s (9.783570 km/h)
[03:03] RelNavLIDAR::distance - Displacement: 0.277442 m
[03:03] RelNavLIDAR::distance - Vehicle velocity: 2.787213 m/s (10.033966 km/h)
[03:04] RelNavLIDAR::distance - Displacement: 0.267929 m
[03:04] RelNavLIDAR::distance - Vehicle velocity: 2.691672 m/s (9.690019 km/h)
[03:06] RelNavLIDAR::distance - Displacement: 0.270389 m
[03:06] RelNavLIDAR::distance - Vehicle velocity: 2.717910 m/s (9.784476 km/h)
[03:07] RelNavLIDAR::distance - Displacement: 0.276768 m
[03:07] RelNavLIDAR::distance - Vehicle velocity: 2.781004 m/s (10.011614 km/h)
[03:08] RelNavLIDAR::distance - Displacement: 0.286318 m
[03:08] RelNavLIDAR::distance - Vehicle velocity: 2.838565 m/s (10.218836 km/h)
[03:09] RelNavLIDAR::distance - Displacement: 0.283380 m
[03:09] RelNavLIDAR::distance - Vehicle velocity: 2.846865 m/s (10.248714 km/h)
[03:10] RelNavLIDAR::distance - Displacement: 0.287945 m
```

Processed timetag 116

Fig. 11: Receiver log showing relevant information related to the navigation module: number of frames per epoch, displacement and estimated velocity of the vehicle [Top]. Visualization of a point cloud after clustering (green points) at a certain epoch and the displacement vector (red line) computed between the current and previous frame [Bottom].

USER INTERFACE AND SQL DATA STORAGE

The GUI framework was changed from C++ with Qt to C# with Windows Presentation Foundation (WPF). This allows faster development with modern frameworks and programming languages. The advantages of C# include modern language features, a managed address space that supports garbage collection and a larger collection of open-source libraries that can be used. The GUIs have been separated to provide a light weight installation for the analysis (MuSNAT-Analyzer) and a full installation for the processing (MuSNAT-Core). The analysis GUI can be installed on different working stations, without the core libraries, to read previously processed data files (SQLite).

The legacy software stored all processing results in ASCII files, which were then analyzed with MATLAB. This approach was simple to use but had two major disadvantages. Longer processing runs created numerous files which were difficult to handle and establishing a common timeline could only be achieved with significant programming effort. Therefore, it was decided that the MuSNAT-Core stores all processing results (correlator values, discriminator values, sensor data, navigation message bits, GNSS raw data and the various PVT solutions) in a single SQL data base that can easily be accessed by Matlab tools and by the MuSNAT-Analyzer.

The SQLite [41] database provides a simple and fast way to store relational data. The processed data are stored in a single file, and no database server needs to be installed. This allows to store all results in an extra file that can be copied to other computers. For each processing run, a new output folder is created which contains the SQL database, the used MuSNAT configuration file and all legacy ASCII output files.

SQLite is a rational database allowing to establish certain relationships between the entries. The time epoch is the most important relationship but also other exist. For example, a batch of DLL discriminator values is linked to a single code pseudorange and the set of all code pseudoranges is linked to a single position fix. There exist other database structures, like time a series databases, that might be more efficient in this context and this will be further evaluated. SQLite is an embedded database, i.e. running within the same process as MuSNAT and no further installation of a server is required. SQLite provides interfaces to many other programming languages and is widely spread.

The process of extraction of the processing data until writing to the database has some complexity. It is based on an Object-Relational-Mapper (ORM) which is a framework analyzing the relevant data structures within MuSNAT (C# classes based on C++ classes) and then generates automatically the SQL tables and rows. This ensures a homogeneous naming convention within the source code and the SQL database for the many types data elements.

The following process is implemented in MuSNAT:

- Backend of MuSNAT-Core to process data and continuously to produce processing results in the form of C++ objects.
- This data is passed via callback functions synchronously to a consumer thread, that converts the C++ objects to C# (managed) objects and stores them in an array.
- Another tread runs in parallel (at GUI level) and based on a timer it polls the data from the consumer thread.
- The consumer thread is thread-safe and protected via a mutex from simultaneous read or write operations; all operations occurring in the consumer thread are very fast and do not block the navigation processing.
- The C# objects are passed to a further thread that writes them asynchronously to the database applying the ORM
- The write process is monitored and in case the write speed speed is too low, the whole processing needs to be paused in order to not run into a buffer overrun situation. Currently the buffer is set to 10 measurement epochs.

The whole logging process has been implemented including GNSS raw data, tracking logs, acquisition functions, tracking multi-correlator functions, GNSS IF sample spectra, camera images, and inertial sensor data. Currently extensive testing is performed to see if the performance is sufficient. LiDAR raw data as well as the raw GNSS IF samples are not logged into the database.

AUTOMATED BUILD AND TEST ENVIRONMENT

To guarantee the stability of the builds and the program logic, a build server was installed. This build server is based on Jenkins [42]. It uses batch files to build the program on a daily basis or a changed repository and start unit tests. The test results are stored for each build and failed tests can be monitored and analyzed. The current configuration of the Jenkins builds the release and the debug version of the MuSNAT to ensure a fast and total test framework. Until now the Jenkins runs 7 different tests two for the release and five for the debug version. Two routines check the integrated filters and their output. One routine checks the correct starting and stopping process. Another test verifies the sending and receiving process of the generated data. A further test inspects the behavior of the software and points out memory leaks. Additionally, one routine ensures the correct values and accuracy of the calculated data in a long-term test that runs the software for twenty times in a row. Finally, the last test checks the correct generation of the signal generator, this is a two staged test shown Fig. 12. In the first stage the stream file is generated and afterwards in the second stage this generated file is tested on accuracy and signal power. The output of all routines is stored in an Extensible Markup Language (XML) file that contains specific information which tests and substages succeeded or failed. On this basis the Jenkins server decides if the software is stable, unstable or a failure. The statistics and current building information is provided via a web interface of the Jenkins server. It is visualized and sorted to simplify the error detection process.

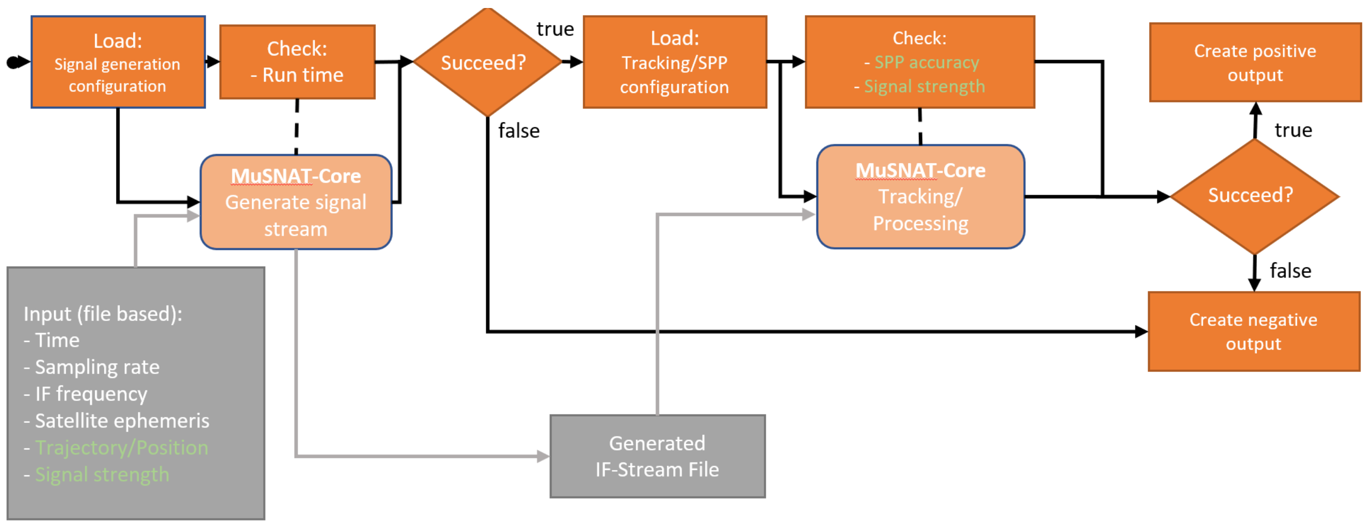


Fig. 12: Test procedure for GNSS signal generator and receiver

CONCLUSIONS AND LESSONS LEARNT

The development of MuSNAT is currently in a late alpha/early beta phase. The software is fully operational but usability and stability still need improvement. Many ipexSR legacy processing modules are of course finished. MuSNAT is understood to be continuously updated as new modules and algorithms will be added over time.

As for every large-scale software project maintaining code stability and development of high-performance modules are the core tasks that need to be fulfilled by the development team. Good communication and documentation form the basis for that. The early use in research projects and for teaching purposes demonstrated the already achieved “added value” to the researcher or to the student.

Generally, it can be said that with CPU based signal processing real-time GNSS signal processing can be achieved for standard multi-GNSS and multi-frequency configurations operating on signal-in-space. The fast correlation mode provides for many use cases sufficient accuracy. The use of a GPU seems promising but a few more steps need to achieved before the GPU will become the workhorse of GNSS signal processing.

Regarding use of sensor data and navigation processing it was noted that only LiDAR processing (in this early version) seems to require a slightly higher computational effort. This has to be monitored and eventually be optimized. The use of the SQL database to store all processing data in a single file was highly appreciated and gives more concise processing results but the SQL writer routines clearly deserves some optimization.

The number of external libraries has now been consolidated and for the foreseeable future no further libraries need to be added. But if this turns out not to be true (e.g. if a Simultaneous Localization and Mapping (SLAM) library shall be added) the use of the vcpkg system provides means for a hopefully easy integration.

During the early development stages, it was also investigated if a platform independent software can be created which runs on PCs (Windows) and on mobile devices (Android). A number of ideas has been pursued starting from the Android support of Visual Studio to finally cmake-based solutions. It was found that Visual Studio 2017 seems not to cover the full range vcpkg libraries for Android, and the variety of dependences (low level Win32 calls, IPP, CUDA, pcl, …) renders creating of a cmake based toolchain a time-consuming software engineering task. Finally, it was decided that for future navigation projects at ISTA related to smartphone positioning, the Android devices serve ‘only’ as data logger and the data is then analyzed via MuSNAT on a PC in post-processing. If key algorithms shall run in real-time on an Android device porting of those algorithms without the MuSNAT framework shall be considered.

ACKNOWLEDGMENTS

This work is funded by the German Federal Ministry for Economic Affairs and Energy on the basis of a decision by the German Bundestag. It is administrated by the German Aerospace Center in Bonn, Germany (FKZ: 50 NA 1703 – SatNavAuth, FKZ: 50 NA 1910 – OPA3L, FKZ: 50 NA 1720 - RAMBO).

REFERENCES

[1] https://www.bosch.co.jp/aee2018/pdf/bosch-automotive-engineering-exposition-2018-automated-cc-vehicle-motion-and-position-sensor-vmps-02.pdf

[2] https://positioningservices.trimble.com/services/rtx/centerpoint-rtx/

[3] Pany, Thomas, Förster, Frank, Eissfeller, Bernd, "Real-Time Processing and Multipath Mitigation of High-Bandwidth L1/L2 GPS Signals With a PC-Based Software Receiver," Proceedings of the 17th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS 2004), Long Beach, CA, September 2004, pp. 971-985.

[4] https://github.com/microsoft/vcpkg

[5] sdr.ion.org

[6] https://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio.html

[7] Sharma, Himanshu, Schütz, Andreas, Pany, Thomas, "Preliminary Analysis of the RTK Positioning using Android GNSS Raw Measurements and Application Feasibility for the Trajectory Mapping using UAV's," Proceedings of the 31st International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2018), Miami, Florida, September 2018, pp. 432-444.

[8] http://www.pointclouds.org/

[9] https://velodynelidar.com/vlp-16.html

[10] https://docs.opencv.org/master/d1/d19/group__viz.html

[11] Scaioni, M., et al. "Methods from information extraction from lidar intensity data and multispectral lidar technology." 2018 ISPRS TC III Mid-Term Symposium on Developments, Technologies and Applications in Remote Sensing. Vol. 42. No. 3. International Society for Photogrammetry and Remote Sensing, 2018.

[12] https://opencv.org/

[13] Pany, T, *Navigation signal processing for GNSS software receivers*. Artech House, ISBN: 9781608070275, 2010.

[14] Pany, T., Dampf, J., Bär, W., Winkel, J., Stöber, C., Fürlinger, K., Closas, P., Garcia-Molina, J.A., "Benchmarking CPUs and GPUs on Embedded Platforms for Software Receiver Usage," Proceedings of the 28th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2015), Tampa, Florida, September 2015, pp. 3188-3197.

[15] HOBIGER, Thomas, et al. A GPU based real-time GPS software receiver. GPS solutions, 2010, 14. Jg., Nr. 2, S. 207-216.

[16] Gao, Yang, Yao, Zheng, Lu, Mingquan, "Design and implementation of a real-time software receiver for BDS-3 signals", NAVIGATION, Journal of The Institute of Navigation, Vol. 66, No. 1, Spring 2019, pp. 83-97.

[17] WU, Jyun-Cheng; CHEN, Lei; CHIUEH, Tzi-Dar. Design of a real-time software-based GPS baseband receiver using GPU acceleration. In: Proceedings of Technical Program of 2012 VLSI Design, Automation and Test. IEEE, 2012. S. 1-4.

[18] Fernández-Prades, Carles, Arribas, Javier, Closas, Pau, "Accelerating GNSS Software Receivers," Proceedings of the 29th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2016), Portland, Oregon, September 2016, pp. 44-61.

[19] Park, Kwi Woo, Jang, Woo Jin, Park, Chansik, Kim, Sunwoo, Lee, Min Jun, "Implementation and Analysis of GNSS Software Receiver on Embedded CPU-GPU Heterogeneous Architecture," Proceedings of the 29th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2016), Portland, Oregon, September 2016, pp. 70-76.

[20] Zhou, Xiaosong, Zhang, Min, Yang, Dongkai, Xue, Qing, "An Algorithm Optimization of GNSS Signal Acquisition and Tracking Based on GPU," Proceedings of the ION 2015 Pacific PNT Meeting, Honolulu, Hawaii, April 2015, pp. 612-617.

[21] HUANG, Bin, et al. STARx—a GPU based multi-system full-band real-time GNSS software receiver. In: ION GNSS. 2013. S. 1549-1559.

[22] Bartunkova, Iva, Eissfeller, Bernd, "Broadband Multi-frequency GNSS Signal Simulation with GPU," Proceedings of IEEE/ION PLANS 2016, Savannah, GA, April 2016, pp. 477-490.

[23] Guifeng, Xu, Xueyong, Cui, Xiaowei, Lu, Mingquan, "A High-Fidelity Wideband Signal Software Simulator for GNSS Antenna Arrays Accelerated by GPU," Proceedings of the 2019 International Technical Meeting of The Institute of Navigation, Reston, Virginia, January 2019, pp. 197-208.

[24] SEO, Jiwon, et al. A real-time capable software-defined receiver using GPU for adaptive anti-jam GPS sensors. Sensors, 2011, 11. Jg., Nr. 9, S. 8966-8991.

[25] Gotoh, Tadahiro, et al. "Development of a GPU-based two-way time transfer modem." IEEE Transactions on Instrumentation and Measurement 60.7 (2010): 2495-2499.

[26] Pany, T., Dampf, J., Bär, W., Winkel, J., Stöber, C., Fürlinger, K., Closas, P., Garcia-Molina, J.A., "Benchmarking CPUs and GPUs on Embedded Platforms for Software Receiver Usage," Proceedings of the 28th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2015), Tampa, Florida, September 2015, pp. 3188-3197.

[27] https://en.wikipedia.org/wiki/GeForce_20_series

[28] Jürgen Dampf, Christian A. Lichtenberger, Thomas Pany: Probability Analysis for Bayesian Direct Position Estimation in a Real-Time GNSS Software Receiver:, Universität der Bundeswehr München, Institute of Space Technology and Space Applications, Germany. Proc. ION-GNSS+ 2019, Miami, 2019.

[29] https://www.intel.com/content/www/us/en/programmable/solutions/acceleration-hub/solutions.html

[30] Gkougkas, Elias, Dötterböck, Dominik, Pany, Thomas, Eissfeller, Bernd, "A Low-power Authentication Signal for Open Service Signals," Proceedings of the 30th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2017), Portland, Oregon, September 2017, pp. 3865-3878.

[31] Maier, Daniel S., Frankl, Kathrin, Pany, Thomas, "The GNSS-Transceiver: Using Vector-tracking Approach to Convert a GNSS Receiver to a Simulator; Implementation and Verification for Signal Authentication," Proceedings of the 31st International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS+ 2018), Miami, Florida, September 2018, pp. 4231-4244.

[32] Stöber, C., Kneissl, F., Eissfeller, Bernd, Pany, T., "Analysis and Verification of Synthetic Multicorrelators," Proceedings of the 24th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS 2011), Portland, OR, September 2011, pp. 2060-2069.

[33] http://www.rtklib.com/

[34] Groves, Paul D: *Principles of GNSS, inertial, and multisensor integrated navigation systems*. Artech house, 2013.

[35] Sánchez, Daniela E., Gómez, Harvey C., Pany, Thomas, "Modelling and Understanding LiDAR Data for Absolute and Relative Positioning," Proceedings of the 31st International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2018), Miami, Florida, September 2018, pp. 3055-3069.

[36] A. Nguyen and B. Le, Eds., 3D point cloud segmentation: A survey, 2013.

[37] W. Wohlkinger and M. Vincze, Eds., Ensemble of shape functions for 3d object classification: IEEE, 2011.

[38] G. H. Golub and C. Reinsch, “Singular value decomposition and least squares solutions,” Numerische mathematik, vol. 14, no. 5, 1970, pp. 403–420.

[39] P. J. Besl and N. D. McKay, Eds., Method for registration of 3-D shapes: International Society for Optics and Photonics, 1992.

[40] I. Y. Bar-Itzhack, "New method for extracting the quaternion from a rotation matrix", AIAA Journal of Guidance, Control and Dynamics, 23 (6), November-December 2000, pp. 1085-1087.

[41] https://www.sqlite.org/index.html

[42] https://jenkins.io/

APPENDIX A – TRACKING PERFORMANCE VERIFICATION

Code/phase noise vs carrier-to-noise ratio

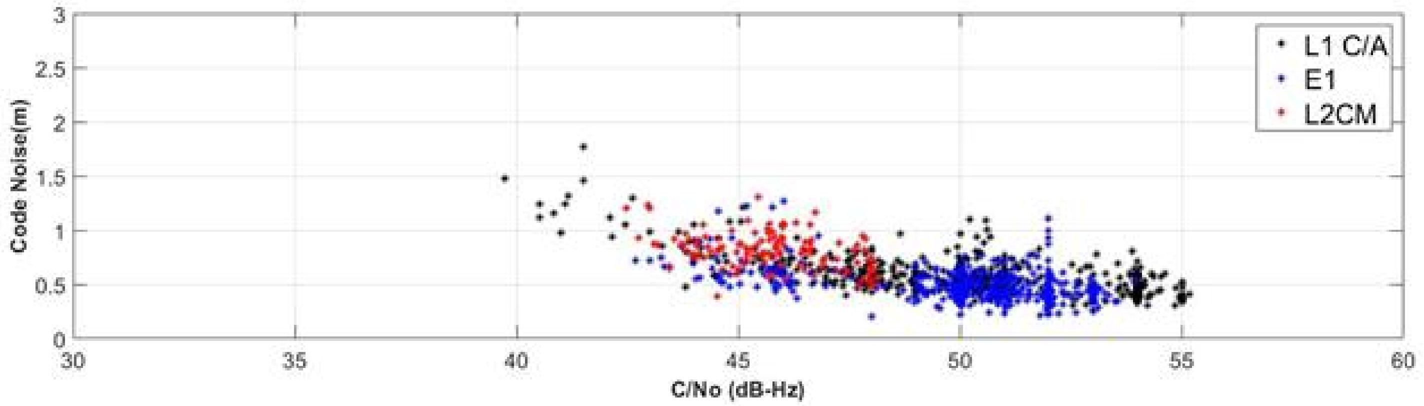


Fig. 13: L1/E1/L2CM Code Noise

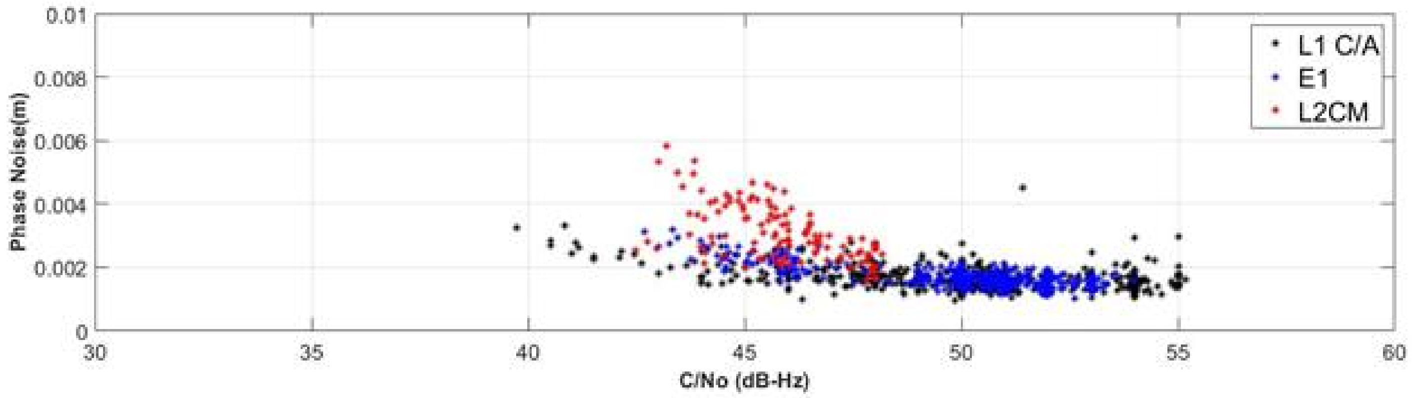


Fig. 14: L1/E1/L2CM Phase Noise

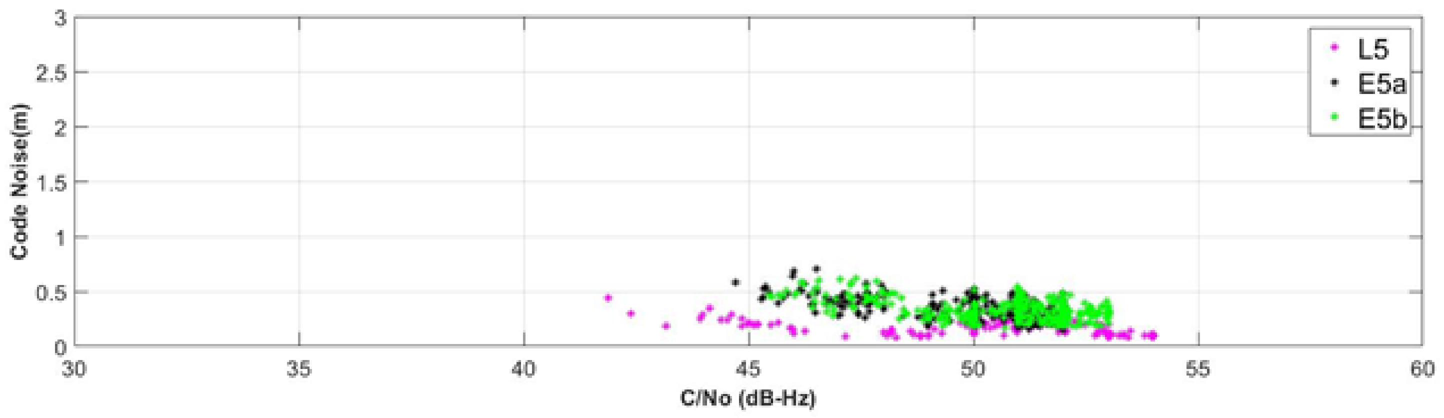


Fig. 15: - L5/E5a/E5b Code Noise

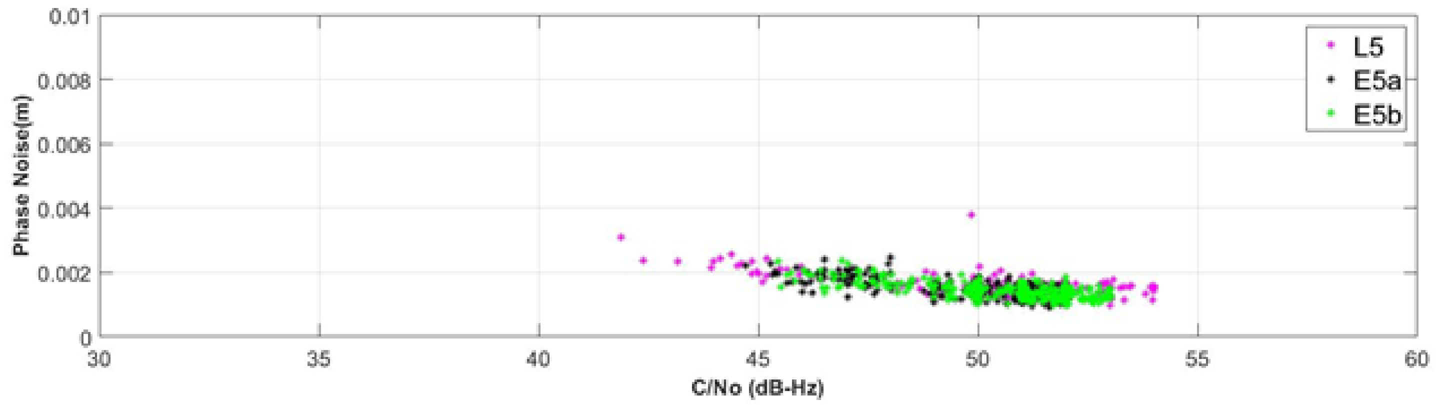


Fig. 16: - L5/E5a/E5b Phase Noise

Measured receiver loop discriminator noise plots

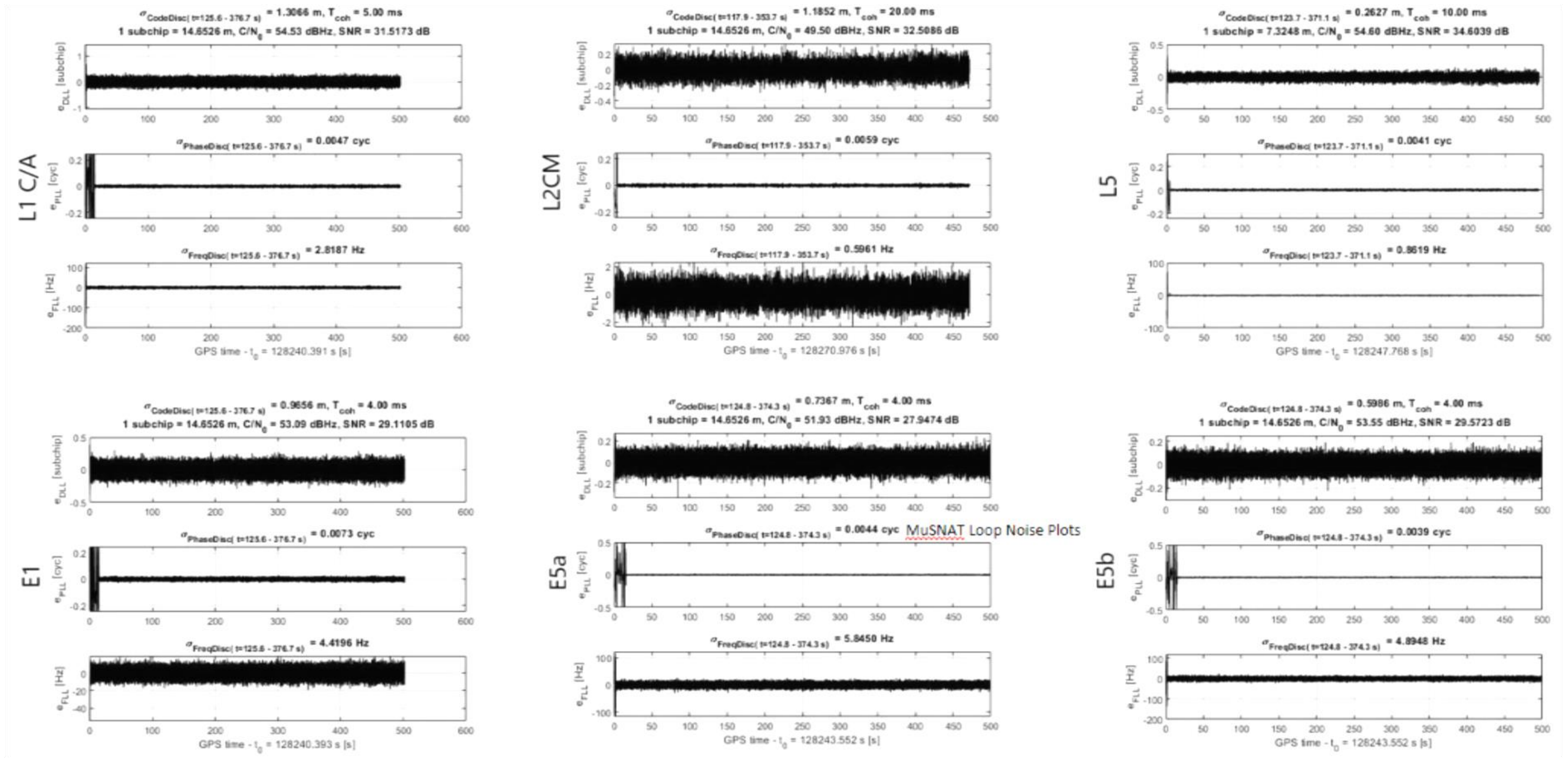


Fig. 17: DLL, PLL and FLL discriminator logs for the considered verification signals

APPENDIX B.1 – GPU CODE SNIPPET FOR REPLICA GENERATION

```
// compute GNSS replica
// replica signal as half2 (I/Q-signal)
// basenband waveform real-values (as half)
// fIniPhase ... initial carrier phase in [rad]
// fDeltaPhase ... delta carrier phase in [rad]
// fIniCodePhase ... intial code phase in [subchip]
// fDeltaCodePhase ... delta code phase in [subchip]
// N ... number of samples to compute
__global__ void genGnssReplicaKernel(CUDA_SAMPLE_TYPE *replica, const half *baseband, const float fIniPhase, const float fDeltaPhase, const
float fIniCodePhase, const float fDeltaCodePhase, const int N)
{
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        int stride = blockDim.x * gridDim.x;

        half2 hCarrier;
        float fCodePhase;
        float fCarrPhase;
        int nSubchipIdx;
        half2 hSubChipValue;
        half2 hCarrOffset;
        float f2Pi=6.28318530717959;

        hCarrOffset.x = 1.5707963267949;
        hCarrOffset.y = 0.0;

        // CUDA loop over all samples
        for (int k = index; k < N; k += stride)
        {
                // code replica
                fCodePhase = fIniCodePhase + k * fDeltaCodePhase;
                nSubchipIdx = floorf(fCodePhase);
                hSubChipValue = __half2half2(baseband[nSubchipIdx]);

                // carrier replica (cos in .x, sin in .y)
                fCarrPhase = fIniPhase + k * fDeltaPhase;
                fCarrPhase = (fCarrPhase - floorf(fCarrPhase))*f2Pi; // phase computation done with 32-bit float
                hCarrier = h2sin(       // carrier NCO
                                        __hadd2(
                                                __half2half2(
                                                        __float2half(fCarrPhase) // convert 32-bit phase to 16-bit
                                                )
                                        , hCarrOffset ) // apply pi/2 offset for cosine
                                );

                // product
                replica[k] = __hmul2(hSubChipValue, hCarrier );
        }
}
```

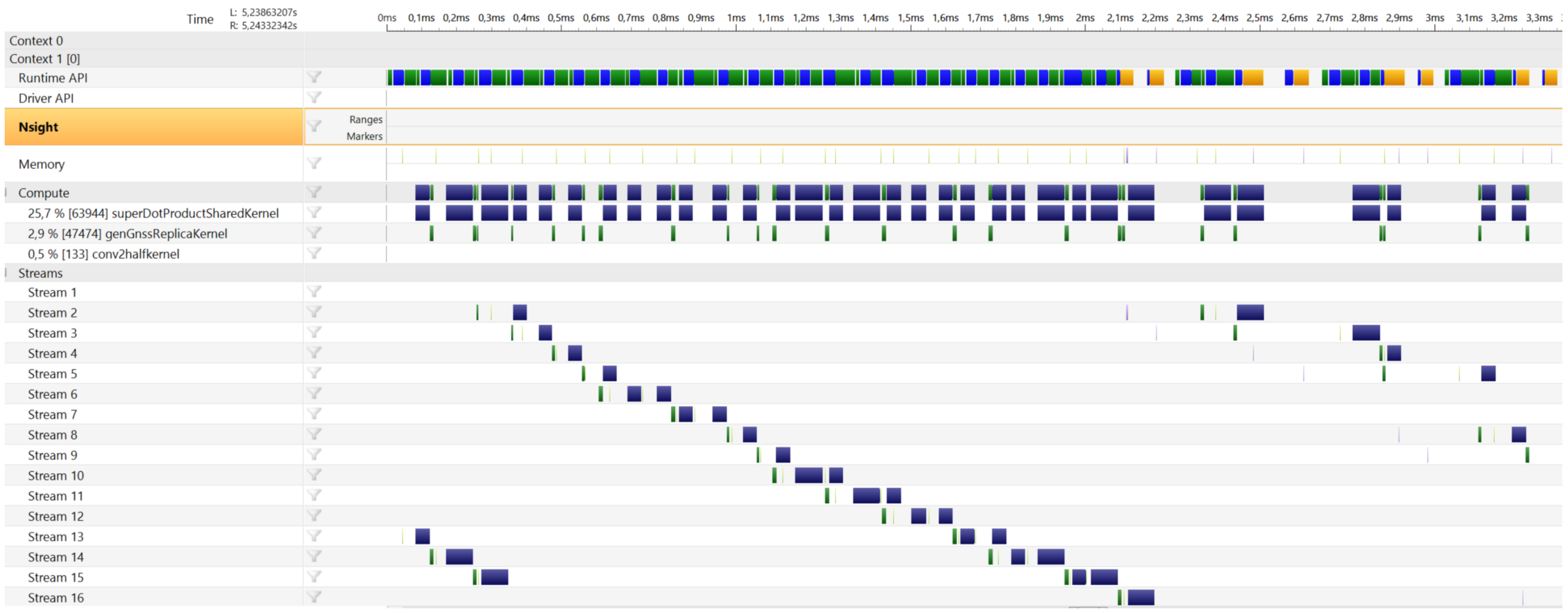
APPENDIX B.2 – GPU CODE SNIPPET CORRELATION (DOT-PRODUCT, DIRECT VERSION WITH GLOBAL MEMORY)

```
// parallel computation of dot-products
// x ... list of first vectors
// y ... list of second vectors
// dot ... list of resulting dot products
// n ... list of vector lengths
// maxn ... maximum length
// N ... number of vectors
__global__ void superDotProductKernel(const CUDA_SAMPLE_TYPE **x, const CUDA_SAMPLE_TYPE **y, float *dot, unsigned int *n, const unsigned int
maxn, const unsigned int N  )
{
        half2 temp = __float2half2_rn(0.f);
        const half2   *myx;
        const half2   *myy;
        unsigned int myn;
        int corridx = blockIdx.x, s2, s = STRIDE;
        __shared__ half2 _dot[STRIDE];

        // init part-sums to 0.0
        _dot[threadIdx.x] = __float2half2_rn(0.f);
        __syncthreads();

        myn = n[corridx]; // length of product
        myx = x[corridx]; // input 1
        myy = y[corridx]; // input 2

        // do part sums (note: there is only one warp per correlator I or Q)
        for (int i = (threadIdx.x); i < myn; i += STRIDE ) temp = __hfma2(myx[i], myy[i], temp);  // temp += myx[i]*myy[i]

        // sum correlator I/Q values in _dot (one warp per value)
        _dot[threadIdx.x] = __hadd2( _dot[threadIdx.x],  temp );
        __syncthreads();

        // horizontal summation over all _dot values within on warp to obtain resp. correlator I/Q value
        while (s > 1)
        {
                s2 = s >> 1;
                if (threadIdx.x < s2)
                        _dot[threadIdx.x] = __hadd2( _dot[threadIdx.x], _dot[threadIdx.x + s2] );
                __syncthreads();
                s = s2;
        }

        // finally convert correlator value from FP16 to FP32
        if (threadIdx.x == 0)
        {
                dot[2 * corridx] = float(_dot[0].x);
                dot[2 * corridx + 1] = float(_dot[0].y);
        }
}
```

APPENDIX B.3 – GPU CODE SNIPPET CORRELATION (DOT-PRODUCT, SHARED MEMORY VERSION)

```
// x ... first signal
// y ... second signal (plus SUPERDOTPRODUCTKERNEL_BARRIER zeros padded at beginning and end)
// dot ... list of resulting dot products
// len ... correlation length
// off ... list of offsets
// N ... number of correlators
// stride ... internal parameter to distribute among kernels
__global__ void superDotProductSharedKernel(const half2  *x, const half2  *y, float *dot, const unsigned int len, const int *off, const
unsigned int N)
{
        __shared__ half2 reg_dot[SUPERDOTPRODUCTKERNEL_MAX_THREAD]; // dot-product values for one region and one correlator values
        __shared__ half2 reg_x[SUPERDOTPRODUCTKERNEL_REGION_SIZE]; // region copy of vector x
        __shared__ half2 reg_y[SUPERDOTPRODUCTKERNEL_REGION_SIZE + 2 * SUPERDOTPRODUCTKERNEL_BARRIER]; // region copy of vector y (plus
barrier)

        // compute required number of regions
        int no_regions = (len + SUPERDOTPRODUCTKERNEL_REGION_SIZE - 1) / SUPERDOTPRODUCTKERNEL_REGION_SIZE;
        int reg_offset;

        // init correlation products to 0.0
        if (blockIdx.x == 0)
        {
                for (int n = threadIdx.x; n < 2 * N; n += blockDim.x)
                {
                        dot[threadIdx.x] = 0.0;
                }
        }
        __syncthreads();

        // loop over all regions
        for (int region_idx = blockIdx.x; region_idx < no_regions; region_idx += gridDim.x)
        {
                // copy samples from device global memory to shared memory
                reg_offset = region_idx * SUPERDOTPRODUCTKERNEL_REGION_SIZE;

                for (int n = threadIdx.x; n < SUPERDOTPRODUCTKERNEL_REGION_SIZE; n += blockDim.x)
                {
                        reg_x[n] = x[n + reg_offset];
                        reg_y[n] = y[n + reg_offset];
                        reg_y[n + SUPERDOTPRODUCTKERNEL_REGION_SIZE] = y[n + reg_offset + SUPERDOTPRODUCTKERNEL_REGION_SIZE];
                }
                __syncthreads();

                // define correlator/thread layout
                int stride = blockDim.x / N;
                int corr_no = threadIdx.x / stride; // correlator number
                int loc_thread_idx = threadIdx.x % stride;

                // compute correlation value "corr_no" in that region
                half2 temp = __float2half2_rn(0.f);
                int myoff;

                if (corr_no < N) myoff = off[corr_no];

                // thread-local loop for one correlation value and one region and one stride value
                int reg_length = min(len - reg_offset, SUPERDOTPRODUCTKERNEL_REGION_SIZE);
                if (corr_no < N)
                {
                        for (int i = loc_thread_idx; i < reg_length; i += stride)
                        {
                                temp = __hfma2(reg_x[i], reg_y[i + myoff + SUPERDOTPRODUCTKERNEL_BARRIER], temp);
                        }
                        reg_dot[loc_thread_idx + stride * corr_no] = temp; // assume atomic add
                }
                // put stride-wise sums into shared memory

                __syncthreads();

                // stride-wise summation
                int s = stride;
                int s2;
                while (s > 1)
                {
                        s2 = s >> 1;
                        if (loc_thread_idx < s2 && corr_no < N)
                                reg_dot[loc_thread_idx + stride * corr_no] = reg_dot[loc_thread_idx + stride * corr_no] +
reg_dot[loc_thread_idx + stride * corr_no + s2];
                        __syncthreads();
                        s = s2;
                }

                // sum-up regions to obtain full correlation values
                if (loc_thread_idx == 0 && corr_no < N)
                {
                        atomicAdd(dot + 2 * corr_no, float(reg_dot[stride * corr_no].x));
                        atomicAdd(dot + 2 * corr_no + 1, float(reg_dot[stride * corr_no].y));
                }
                __syncthreads();
        }
}
```

APPENDIX B.4 – GPU TIMING OF MUSNAT WHILE TRACKING GNSS SIGNALS WITH GPU


Time
L: 5,23863207s
R: 5,24332342s
0ms 0,1ms 0,2ms 0,3ms 0,4ms 0,5ms 0,6ms 0,7ms 0,8ms 0,9ms 1ms 1,1ms 1,2ms 1,3ms 1,4ms 1,5ms 1,6ms 1,7ms 1,8ms 1,9ms 2ms 2,1ms 2,2ms 2,3ms 2,4ms 2,5ms 2,6ms 2,7ms 2,8ms 2,9ms 3ms 3,1ms 3,2ms 3,3ms
Context 0
Context 1 [0]
Runtime API
Driver API
Nsight
Ranges
Markers
Memory
Compute
25,7 % [63944] superDotProductSharedKernel
2,9 % [47474] genGnssReplicaKernel
0,5 % [133] conv2halfkernel
Streams
Stream 1
Stream 2
Stream 3
Stream 4
Stream 5
Stream 6
Stream 7
Stream 8
Stream 9
Stream 10
Stream 11
Stream 12
Stream 13
Stream 14
Stream 15
Stream 16