April 9 2019

Initial setup

performed according to the description from the GNSS-SDR site.

- Installed dependencies
- cloned gnss-sdr GitHub repository
- installed using cmake and make

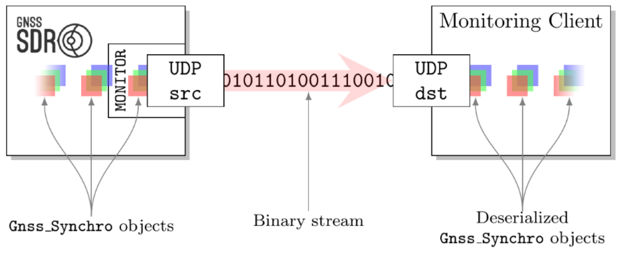
First test

- downloaded a test data set from https://sourceforge.net/projects/gnss-sdr/files/data/
- successfully executed

```
gnss-sdr --config_file=./test01.conf
```

Tutorial 1: Monitoring the internal status of the software receiver

Reproduced the instructions from the gnss-sdr tutorial
The tutorial consists in building a monitoring client.



The monitor block allows inspection of Gnss_Synchro objects holding various parameters before a signal reaches a PVT block by creating an UDP object stream. The stream can then be deserialized by a monitoring client.

The program described in this tutorial monitors the following parameters:

- PRN: Satellite ID processed in each channel. (uint32_t)
- CN0_dB_hz: Carrier-to-Noise density ratio, in [dB-Hz], (double)
- Carrier_Doppler_hz: Doppler frequency shift estimation in each channel, in [Hz]. (double)

In order to activate the monitor block and configure it to stream, the following snippet needs to be added to the config file:

```
;######### MONITOR CONFIG ############
Monitor.enable_monitor=true
Monitor.output_rate_ms=1000
Monitor.client_addresses=127.0.0.1
Monitor.udp_port=1234
```

When building other clients `CMakeCache.txt` and `CMakeLists.txt` need to be updated.

April 10 2019

Tutorial 2: The testing framework

This tutorial shows how from the install directory the program `run_tests` can be executed with various flags in order to perform different kinds of tests:

- list the names of all available tests: `./run_tests --gtest_list_tests`
- run all tests (takes a while since they are a lot): `./run_tests`
- run a subset of the tests: use regex syntax e.g. `./run_tests --gtest_filter=GpsL1CaPcpsAcquisitionTest.*`
- repeat tests: e.g. `./run_tests --gtest_filter=GpsL1CaPcpsAcquisitionTest.* --gtest_repeat=10`
- generate xml test report: e.g. `./run_tests --gtest_filter=CpuMulticorrelatorTest.* --gtest_output=xml`

Creating custom tests for TDD

New tests can be written using the TEST() macro:

```
TEST(test_case_name, test_name)
{
    ... test body ...
}
```

example:

```
#include <gtest/gtest.h> // Include Google Test headers
#include "rtcm.h"        // Include header under test

TEST(RtcmTest, HexToInt) // RtcmTest is the name of the Test Case
                         // HexToInt is the name of this test
{
    auto rtcm = std::make_shared<Rtcm>();
    std::string test1 = "2A";
    long int test1_int = rtcm->hex_to_int(test1);
    long int expected1 = 42;
    EXPECT_EQ(expected1, test1_int);
}
```

Unit tests must be included in `sdr/src/tests/test_main.cc`
System tests must be included in `sdr/src/tests/CMakeLists.txt`.
Once this is done run `make` int the `build` directory. Now the custom test is included in the testing framework.
Additional information can be found on the googletest GitHub page.

April 12 2019

Monitor all exposed parameters

Built a C++ class that prints out the values of all parameters exposed by the monitor block. The sources are in `/myTests/monitorAll/`.

- compile: from `myTests/monitorAll/build` run `cmake ..`, `make`
- run: from `myTests/monitorAll/build` run `./monitorAll`
  from `myTests/monitorAll/work` run `gnss-sdr --config_file=monitorAll.conf`.

April 13 2019

Extract features needed for the classifier using the monitor block

By the 2017 GAD report by the ALaRI institute i noticed at page 7 table 1 a list of features that I assume being the input for the classifier. Some of them are found directly among the parameters expose by the monitor block, others can be computed from these parameters, yet others may need to be extracted in other ways. Following is an attempt to retrieve as many of the needed features as possible. The features marked with TODO probably require an other approach. The code of my attempt to build an extractor based on the data from the monitor block is in `/myTests/featureExtractor/`.

1. Number of valid satellites identified by the receiver

   ASSUMPTION: the number of valid satellites corresponds to the number of distinct PRN which should be equal to the number of channels.

2. Number of valid satellites changed (loss of lock/gain of lock)

   TODO

3. Signal over noise (average among all valid satellites)

   ASSUMPTION: this feature refers to the `CN0_dB_hz` parameter.

4. Pseudorange (pseudo-distance between satellite and receiver)

   ASSUMPTION: this feature refers to the `Pseudorange_m` parameter.

5. Carrier Phase (measure of the range between satellite and receiver expressed in units of cycles of the carrier frequency)

   ASSUMPTION: this feature refers to the `Carrier_phase_rads` parameter.

6. Doppler measured (average among all valid satellites)

   ASSUMPTION: this feature refers to the `Carrier_Doppler_hz` parameter.

7. Easting from the real position

   TODO

8. Northing from the real position

   TODO

9. Height from the real position

   TODO

10. Gap in meters from last known position

    TODO

11. Velocity

    TODO

12. Acceleration

    TODO

13. Amplitude of the raw signal

    TODO

14. Time elapsed with no lock on any satellite signal

    TODO

April 14 2019

Work with the PVT block

After reading the documentation about the PVT block I concluded that the best way to obtain the complete set of features a combination of the outputs from the monitor block (default port 1234 UDP) and the PVT block (default port 1234 UDP) can be a solution. Since both use the same default port the latter must be changed by setting the `PVT.monitor_udp_port` parameter in the config file.
In order to enable the UDP streaming from the PVT block `PVT.enable_monitor=true` must be set in the config file.
Of course it must somehow be checked that the values from the two blocks refer to the same signal.

The following table describes which feature may be retrieved from which source:

feature	Monitor	PVT
Number of valid satellites identified by the receiver	distinct `PRN`	`valid_sats`
Number of valid satellites changed (loss of lock/gain of lock)	x	x
Signal over noise (average among all valid satellites)	avg `CN0_dB_hz`	x
Pseudorange (pseudo-distance between satellite and receiver)	`Pseudorange_m`	x
Carrier Phase	`Carrier_phase_rads`	x
Doppler measured (average among all valid satellites)	`Carrier_Doppler_hz`	x
Easting from the real position	x	x
Northing from the real position	x	x
Height from the real position	x	x
Gap in meters from last known position	x	from `pos_x` ...
Velocity	x	`vel_x` ...
Acceleration	x	from `vel_x` ... over time
Amplitude of the raw signal	x	x
Time elapsed with no lock on any satellite signal	x	x

There are still some features missing. On Easting, Northing and Height from real position I don't know how to define the real position and I did not find a parameter for the raw signal amplitude. The Time elapsed with no lock on any satellite signal, may be measured in some way, but for now I concentrate on the available parameters. The implementation is in `/myTests/combinedExtractor/`.

Problems related to this approach

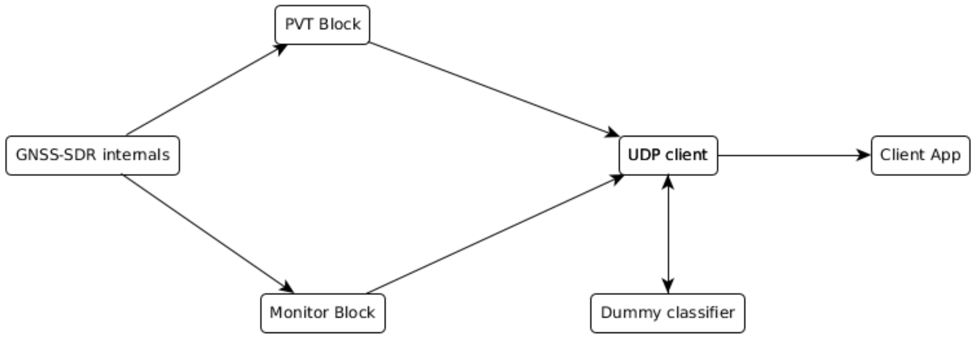
- There are still some features missing.
- In my implementation velocity and acceleration are computed as scalars using vector norm, maybe they should be analyzed component-wise.
- I did not find a way to ensure that the data from the Monitor block and those from the PVT block refer to the same measurement.
- Maybe a more elegant approach would be to integrate the classifier into GNSS-SDR in stead of building client programs.

In spite of these issues, as a next step I will try to build a client connected to a dummy classifier; Probably we will end up doing something completely different, but I think it is useful as an exercise.

April 15-16 2019

UDP client with dummy classifier

Using the parameters exposed by the PVT And Monitor blocks it is possible to build a client using the following structure:



The inputs of the dummy-classifier are combined from the exposed parameters. Its output is an `enum` whose fields are either `OK` (no spoofing), or one of the attack scenarios described in the GAD report at page 5.
The UDP client updates an object of a data class which can be read concurrently; This object contains the standard outputs such as latitude, longitude and height, as well as information about whether spoofing is detected, and if yes what kind of attack it is.
The implementation is in `/myTests/client/`.