Nonlinear State Estimation for Carrier Synchronization in GNSS-SDR

A project summary for Google Summer of Code 2019

Gerald Mycko LaMountain

Under the mentorship of Jordi Vilà-Valls, ISAE-SUPAERO
Google Summer of Code 2019
GNSS-SDR. CTTC
August 26, 2019

Contents

BIOGRAPHIES

Gerald LaMountain is a PhD candidate and Dean's Fellow in the Department of Electrical and Computer Engineering at Northeastern University, Boston, MA. His current research involves building upon traditional probabilistic techniques for dynamic state estimation and applying those techniques to problems with applications in positioning and localization. In the past, he has contributed to research efforts in the field of system design for optimal intent classification in the context of human-computer interaction.

Jordi Vilà-Valls is an Associate Professor at the Institut Supérieur de l'Aéronautique et de l'Espace (ISAE-SUPAERO), and member of the Signal, Communications, Antennas, Navigation (SCAN) Research Group, Toulouse, France. He received the PhD degree in Electrical Engineering from Grenoble INP (INPG), France, in 2010. His primary areas of interest include statistical signal processing, estimation and detection theory, robust and computational statistics; with applications to positioning systems and wireless communications.

Abstract

Carrier synchronization in standard, mass-market GNSS receivers typically utilizes well understood locked-loop architectures. The performance obtained with such architectures is sufficient for benign propagation scenarios, but typically deliver poor performance under harsh propagation conditions. Code and carrier tracking, as well as joint code/carrier synchronization, can be formulated as an estimation problem which can be solved using Bayesian filtering methods. It has been shown in the GNSS literature that KF-based synchronization solutions can be used to overcome the performance limitations of standard approaches, offering implicitly adaptive filter bandwidth, and opening up the possibility of using nonlinear models to avoid certain limitations associated with the use of code, phase or frequency discriminators. In this contribution, we will leverage powerful nonlinear tracking algorithms, including cubature, unscented, and sigma point Kalman filters in order to produce realizations of carrier and joint code-carrier tracking blocks which promise to be more effective and adaptable in challenging GNSS environments when compared to traditional architectures.

1. INTRODUCTION

Carrier synchronization in modern mass-market GNSS receivers relies on traditional methods for estimating and tracking the synchronization parameters which are based on delay-locked loop (DLL) and phase-locked loop (PLL) architectures. This standard methodology also forms the basis of the tracking stage of GNSS-SDR. These locked loop architectures derived from analogue signal processing techniques developed in the early 20th century, but have continued to be used in the digital domain due to their relative simplicity and their effectiveness in benign propagation conditions. In some implementations, the PLL used in GNSS and other applications may be coupled to a frequency-locked loop (FLL) in order to provide more reliable estimates of the carrier phase signal parameter offsets, mainly in high-dynamics scenarios. More recently, however, it has been shown in the GNSS literature that that system architectures which are instead based on standard Kalman Filter (KF) methods are not only also applicable to the code-delay and carrier-phase tracking problem, but also that they may offer increased performance relative to traditional locked loop architectures during harsh propagation conditions. Details regarding those and other advantages can be found in the literature [4], however for the purposes of this report it will suffice to say that there is both practical and academic interest in performing GNSS carrier synchronization using Kalman architectures.

One of the practical challenges involved in the use of a Kalman filtering based approach is the assumption of a well defined model characterizing the behavior of both the observed and inferred system parameters. In particular, information about the process and measurement noise covariances is needed for optimal KF performance (i.e., optimal Kalman gain update), but this information is not typically available a priori and in fact may change as channel conditions change during system operation. One way of addressing this challenge is by numerically estimating the system parameters. This may either be done a priori using an "offline" procedure (e.g. using a previously acquired C/N_0 estimation to update the filter parameters), or it may be done in real time using an "online" procedure (e.g. by a modification to the filtering procedure). One of the key goals of our 2018 Google Summer of Code contribution to the GNSS-SDR project was to develop a Bayesian estimation architecture which could be used perform online estimation of the statistical characteristics of the measurement noise during filter operation. In this way, it would be possible to have a GNSS carrier synchronization architecture in which the Kalman filter could be used for long-term carrier phase estimation. Although we developed and tested this methodology in MATLAB using synthetic data [3], we encountered a number of problems with the implementation in the GNSS-SDR data path using real data. In the report for that project, we hypothesized that these issues were likely to have resulted from the use of the suboptimal extended Kalman filter (EKF) algorithm which operates by linearizing the measurement model. This process potentially colors the filter's innovations sequence and can result in a degradation in the performance of the Bayesian covariance estimation (BCE) filter. In order to overcome this limitation we proposed utilizing nonlinear tracking algorithms which make fewer approximations in performing estimation: namely, the unscented Kalman filter and cubature Kalman filter.

In this document we describe the contributions which we have made to the GNSS-SDR project over the course of Google Summer of Code 2019. The primary software contributions are summarized in Sections 2 through 4. In Section 2 we describe the development of several libraries which implement the unscented and cubature Kalman filter algorithms and provide an interface by which they may be used in the tracking block of GNSS-SDR. The subsequent Sections 3 and 4 describe tracking architectures based on these libraries. Section 3 describes a mixed implementation where were Kalman filtering is combined with existing DLL+PLL VEML (Very Early Minus Late) architecture to produce a generic mixed tracking block which tracks carrier phase using the prompt correlator output, and Section 4 describes a joint code-phase tracking architecture based on the early, prompt and late correlator outputs. Finally, in Section 5 we will summarize which of the goals we were able to accomplish during this year's Google Summer of Code, and describe the ways in which this work can be improved and built upon in future contributions.

2. NONLINEAR TRACKING ARCHITECTURE

One of the key goals of our 2018 Google Summer of Code contribution to the GNSS-SDR project was to develop a Bayesian estimation architecture for estimating the covariance of the measurement noise during filter operation. The formulation for that embedded Bayesian filter is described in the literature [3], and relies on the innovations process $\hat{\mathbf{z}}_k = \mathbf{y}_k - \hat{\mathbf{y}}_k$ being a zero-mean white noise sequence. Conveniently, this sequence has been shown to be white in the Kalman filter literature, but only under conditions of *optimal* filter performance (i.e. with precise predictions based on an accurate model). Our initial 2018 effort to incorporate this methodology into GNSS-SDR involved the use of extended Kalman filter (EKF) architectures for performing nonlinear state estimation and tracking. Ultimately, the performance of the Bayesian covariance estimation (BCE) filter under this approach did not meet the standard required for maintaining carrier

synchronization over more than a few seconds. In our report for last year's Google Summer of Code we hypothesized that this issue might be mitigated by the use of a more precise nonlinear tracking procedure. To this end, in our 2019 Google Summer of Code contribution, we developed generalized libraries for modular incorporation of nonlinear, Bayesian tracking algorithms into the GNSS-SDR tracking framework. These libraries can be found in the GNSS-SDR source directory as

`/algorithms/tracking/libs/tracking_Gaussian_filter.h`
`/algorithms/tracking/libs/tracking_models.h`
`/algorithms/tracking/libs/nonlinear_tracking.h`

In the following sections we will describe how these libraries are used, and how they may easily be expanded upon to include new Gaussian filtering methodlogies.

2.1. Gaussian Filtering

The `tracking_Gaussian_filter` library implements the `TrackingGaussianFilter` and `TrackingNonlinearFilter` classes. These classes are based on those defined in the `Tracking_Loop_Filter` library used in DLL+PLL tracking, and are defined as

```
class TrackingGaussianFilter
{
public:
    void set_ncov_process(arma::mat ncov);
    void set_ncov_measurement(arma::mat ncov);
    void set_state(arma::vec state);
    void set_state_cov(arma::mat state_cov);
    void set_params(arma::vec state, arma::mat state_cov, arma::mat p_ncov
    , arma::mat m_ncov);

protected:
    arma::vec d_state;            /* state vector */
    arma::mat d_state_cov;        /* state error covariance matrix */
    arma::mat d_ncov_process;     /* model error covariance matrix */
    arma::mat d_ncov_measurement; /* measurement error covariance matrix
    */
};

template <class NonlinearFilter, class OutputType1, class OutputType2>
class TrackingNonlinearFilter : public TrackingGaussianFilter
{
public:
    arma::vec get_carrier_nco(const OutputType2 meas_in);

    void set_transition_model(ModelFunction<OutputType1>* ft) {
    func_transition = ft; };
    void set_measurement_model(ModelFunction<OutputType2>* fm) {
    func_measurement = fm; };
    void set_model(ModelFunction<OutputType1>* ft, ModelFunction<
    OutputType2>* fm) {
        set_transition_model(ft);
        set_measurement_model(fm);
    };

private:
    ModelFunction<OutputType1>* func_transition;
    ModelFunction<OutputType2>* func_measurement;

    NonlinearFilter GaussFilt;

};
```

Listing 1: tracking_Gaussian_filter library class definitions

The design philosophy of these classes is fairly straightforward: the `TrackingGaussianFilter` class provides a generic interface for storing the basic parameters of a second order state estimation filter: the state, state covariance, and additive noise covariances. The `TrackingNonlinearFilter` class inherits these parameters from `TrackingGaussianFilter` and adds generic functional templates representing the transition and measurement equations. These functionals are intended to be classes which inherit from the `ModelFunction` class laid out in the `tracking_models.h` library, and the functions that they specify are used to compute the model predicted state and mesurements within the Gaussian filter. The way in which these functionals should be constructed and used will be detailed in 2.2.

Use of the `TrackingNonlinearFilter` class is also straightforward, but makes use of templates for modularity. Declaration of an instance of `TrackingNonlinearFilter` requires specifying three parameters: the class specified by `NonlinearFilter` (to be discussed in 2.3) indicates the type of Gaussian filter to use (e.g. UKF, CKF), while `OutputType1` and `OutputType2` specify the output type of the state and measurement equations. These types are expected to be one of the data types from the Armadillo C++ library for linear algebra & scientific computing: in general either `arma::colvec` or `arma::cx_vec`. Once these parameters are specified, the object is instantiated and the filter may be initialized using `set_params`, however it is necessary to attach the model functions before the filter may be invoked with `get_carrier_nco`.

2.2. Model Functions

The `tracking_models` library provides a template from which model functionals may be created. The functionals may be generically defined adjacent to where they are to be used (e.g. in a particular GNU Radio block), but they must inherit from the `ModelFunction` class specified in `tracking_models.h` and must accept a single `arma::colvec` as an input. The output type is specified by the template class parameter `OutputType` and *must* agree with the output type specified by the `TrackingNonlinearFilter` instance in which the model function is to be used. Once defined, an instance of the functional must be created and attached to an instance of `TrackingNonlinearFilter` using `TrackingNonlinearFilter::set_model`. A simple example of such a functional follows:

```
class MixedCarrierTransitionModel : public ModelFunction<arma::vec>
{
public:
    arma::vec operator()(const arma::vec& input) override {
        /*
         * output(0) - Carrier Phase
         * output(1) - Carrier Doppler
         * output(2) - Carrier Doppler Rate
         * output(3) - Correlator Output Amplitude
         */
        arma::vec output = arma::zeros(4,1);
        output(0, 0) = input(0) + PI_2*pdi*input(1) + 0.5*PI_2*std::pow(
    pdi, 2)*input(2);
        output(1, 0) = input(1) + pdi*input(2);
        output(2, 0) = input(2);
        output(3, 0) = input(3);
        return output;
    };
    void set_code_period(const float carrier_pdi) { pdi = carrier_pdi; };
private:
    float pdi;
};
```

Listing 2: Simple Model Functional Example from `mixed_veml_tracking.h`

2.3. Nonlinear Filters

The class specifier `NonlinearFilter` in the template class `TrackingNonlinearFilter` specifies the type of nonlinear filter that should be used. In general, this should refer to a class which inherits from the `GaussianFilter` class specified in the `nonlinear_tracking` library. The `GaussianFilter` class represents the basic implementation of a second order Gaussian filter, and includes setters and getters for the mean and covariance of the state prediction and updated state estimation. The classes which inherit from it must specify the prediction and update steps of the Gaussian filter procedure as `predict_sequential` and `update_sequential` respectively.

In our 2019 Google Summer of Code contribution we have developed and implemented two nonlinear filters: the unscented Kalman filter and the cubature Kalman filter. For details on each of these algorithms, we direct the reader to the literature [1, 2] and focus for the time being on the implementation within GNSS-SDR. The declaration for the unscented and cubature Kalman filter classes follow, and the reader is directed to note that these each provide the same interface to the calling function:

```
class CubatureFilter : public GaussianFilter
{
public:
    // Prediction and estimation
    template <class OutputType>
    void predict_sequential(const arma::vec& x_post, const arma::mat&
    P_x_post, ModelFunction<OutputType>* transition_fcn, const arma::mat&
    noise_covariance);

    template <class OutputType>
    void update_sequential(const OutputType& z_upd, const arma::vec&
    x_pred, const arma::mat& P_x_pred, ModelFunction<OutputType>*
    measurement_fcn, const arma::mat& noise_covariance);
};

class UnscentedFilter : public GaussianFilter
{
public:
    // Prediction and estimation
    template <class OutputType>
    void predict_sequential(const arma::vec& x_post, const arma::mat&
    P_x_post, ModelFunction<OutputType>* transition_fcn, const arma::mat&
    noise_covariance);

    template <class OutputType>
    void update_sequential(const arma::vec& z_upd, const arma::vec& x_pred
    , const arma::mat& P_x_pred, ModelFunction<OutputType>*
    measurement_fcn, const arma::mat& noise_covariance);
};
```

Listing 3: Prediciton and Update function declarations from `nonlinear_tracking.h`

Unit tests for each of these implementaitons can be found in

`/tests/unit-tests/signal-processing-blocks/tracking/`

as `unscented_filter_test.cc` and `cubature_filter_test.cc`. These blocks test the reliability and accuracy of the tracking algorithms by implementing a linear system with random parameters and comparing the output of the nonlinear filters to that of a standard linear Kalman filtering procedure. This test is repeated many times with random initialization and the test is declared as passed if and only the results of the nonlinear filter are within a certain acceptable range of the linear result for each one.

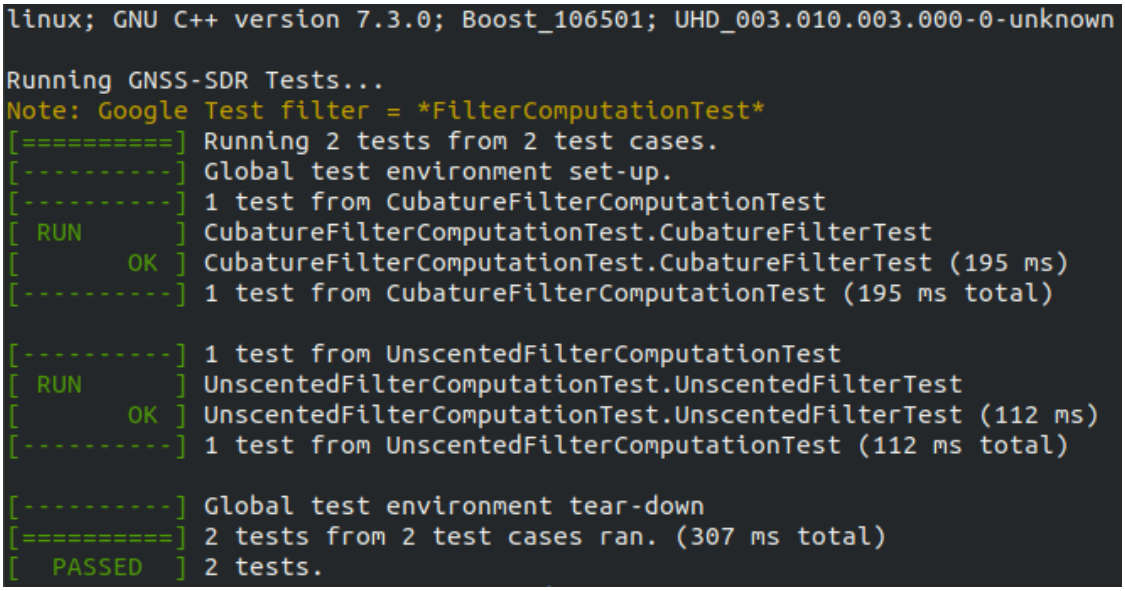

Figure 1: Nonlinear filter unit test outputs for Cubature and Unscented Kalman filter implementations.

3. MIXED DLL CODE AND KALMAN CARRIER TRACKING

Another goal of out 2019 Google Summer of Code contribution to the GNSS-SDR project was to develop a Kalman filter based, discriminator free carrier phase tracking implementation. Such an implementation uses a nonlinear Kalman filter to replace the FLL+PLL portion of the traditional locked loop architecture with a nonlinear Kalman filter for estimating the carrier phase, Doppler and Doppler rate parameters, while leaving estimation of the code error to the standard DLL. In order to be avoid the use of discriminator functions, the filter needs to operate based on a model which directly relates the carrier phase to the output of the correlation step. We attempted to create such an implementation in our 2018 Google Summer of Code contribution but encountered issues getting the method to work with signals which rely on tracking a pilot such as, for example, the those on the Galileo E1 band. To overcome this limitation, we designed a mixed tracking implementation based on the `dll_pll_veml_tracking` GNU radio tracking block which modifies the standard tracking model to be invariant to the bit transitions introduced by the signal being encoded onto a pilot signal. This section provides the details of how this implementation functions, as well as the details of the model under which the Gaussian filter operates.

3.1. Correlator output Carrier Phase Model

Before getting into the details of the GNU radio tracking block implementation, it's useful to understand the underlying model for discriminator free carrier phase estimation. As described in the literature [4], the prompt output of the correlation step of the tracking block is related the carrier phase by way of the following complex equation:

$$\mathbf{y}_k = A_k d_k e^{i\boldsymbol{\Delta\phi_k}} \tag{1}$$

where A_k is the signal amplitude, $d_k \in \{+1, -1\}$ is the bit sign associated with of the pilot signal, and $\Delta\phi_k$ is the carrier phase error, each at sample k. Since the Kalman filter is not well suited to handling rapid changes, such as those associated with bit flips, it is necessary to remove the effect of the d_k bit. This is done by taking the measurement to be square of the correlator outputs, $\tilde{\mathbf{y}} = \mathbf{y}_k^2$. This yields a new complex measurement

model

$$\tilde{\mathbf{y}}_k = \tilde{A}_k e^{2i\boldsymbol{\Delta\phi_k}} \tag{2}$$

where $\tilde{A}_k = A_k^2$. Separating the real and imaginary components of this and accounting for additive noise gives the final real valued measurement model to be used in the filter

$$\Re\{\tilde{\mathbf{y}}_k\} = \tilde{A}_k \cos\left(2\boldsymbol{\Delta\phi_k}\right) + \boldsymbol{\eta_{k,\Re}} \tag{3}$$

$$\Im\{\tilde{\mathbf{y}}_k\} = \tilde{A}_k \sin\left(2\boldsymbol{\Delta\phi_k}\right) + \boldsymbol{\eta_{k,\Im}} \tag{4}$$

The state, $\mathbf{x}_k = [\Delta\phi_k, f_{d,k}, \dot{f}_{d,k}, \tilde{A}_k]$ is designed to incorporate the carrier parameters, namely carrier phase error, Doppler and Doppler rate, as well as the signal amplitude. The carrier parameters are modelled as evolving according to a quadratic transition model, also described in [4], while the correlator output amplitude A_k is assumed as being constant over time. The transition and measurement functions which comprise this model are implemented in `mixed_veml_tracking.h` as

```
class MixedCarrierTransitionModel : public ModelFunction<arma::vec>
{
public:
    arma::vec operator()(const arma::vec& input) override {
        /*
         * input/output(0) - Carrier Phase
         * input/output(1) - Carrier Doppler
         * input/output(2) - Carrier Doppler Rate
         * input/output(3) - Squared Correlator Output Amplitude
         */
        arma::vec output = arma::zeros(4,1);
        output(0, 0) = input(0) + PI_2*pdi*input(1) + 0.5*PI_2*std::pow(
    pdi, 2)*input(2);
        output(1, 0) = input(1) + pdi*input(2);
        output(2, 0) = input(2);
        output(3, 0) = input(3);
        return output;
    };
    void set_code_period(const float carrier_pdi) { pdi = carrier_pdi; };
private:
    float pdi;
};
class MixedCarrierMeasurementModel : public ModelFunction<arma::vec>
{
public:
    arma::vec operator()(const arma::vec& input) override {
        /*
         * input(0) - Carrier Phase
         * input(1) - Carrier Doppler
         * input(2) - Carrier Doppler Rate
         * input(3) - Squared Correlator Output Amplitude
         * output(0) - Real component of squared Prompt
         * output(1) - Imag component of squared Prompt
         */
        using namespace std::complex_literals;
        arma::vec output = arma::zeros<arma::vec>(2,1);
        output(0) = static_cast<double>(input(3)) * std::cos( 2.0 *
    static_cast<double>(input(0)) );
        output(1) = static_cast<double>(input(3)) * (-1) * std::sin( 2.0 *
     static_cast<double>(input(0)) );
        return output;
    };
private:
};
```

Listing 4: Prompt Carrier Phase model as impelemented in `mixed_veml_tracking.h`

3.2. Notes on Code Implementation

The mixed tracking block was implemented into a GNU radio tracking block which may be found in the GNSS-SDR source directory as

`/src/algorithms/tracking/gnuradio_blocks/mixed_veml_tracking*`

This code is adapted from `dll_pll_veml_tracking`. Here, the functionality of the original PLL+FLL for performing carrier tracking is replaced with an instance of the Gaussian filter implementation specified by `TrackingNonlinearFilter`. The parameters of this filter are initialized with the tracking block along with those of the DLL responsible for performing code estimation. Additionally, the parameters of the functionals representing the model are updated throughout this tracking block as needed. The most important differences between the mixed tracking block and the original locked loop tracking block are found in `mixed_veml_tracking::run_dll_pll`. This function is inappropriately named as the function no longer makes use PLL structures, but the original naming convention was left for compatibility with the rest of the GNSS-SDR code. In this function, both the pilot signal and carrier synchronization is performed by invoking the Gaussian filter with `get_carrier_nco`. The filter outputs are then used to compute the parameters that need to be passed to the NCO just as they would be with the standard locked loop architecture.

3.3. Testing and Results

Testing of the `mixed_veml_tracking` GNU radio block was performed by implementing an adapter block and unit test for running this tracking block with Galileo E1 signals. The tracking block and unit test can be found in the GNSS-SDR source repository as

`/src/algorithms/tracking/adapters/galileo_e1_mixed_veml_tracking*`
`/src/tests/.../tracking/galileo_e1_mixed_veml_tracking_test.cc`

The adapter and unit test are based on those implemented for Galileo E1 DLL+PLL tracking and the unit performs a large number of tracking iterations using a simulated Galileo E1B PRN E11 signal.

```
linux; GNU C++ version 7.3.0; Boost_106501; UHD_003.010.003.000-0-unknown

Running GNSS-SDR Tests...
Note: Google Test filter = GalileoE1MixedVemlTracking*
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from GalileoE1MixedVemlTrackingInternalTest
[ RUN      ] GalileoE1MixedVemlTrackingInternalTest.Instantiate
[       OK ] GalileoE1MixedVemlTrackingInternalTest.Instantiate (1 ms)
[ RUN      ] GalileoE1MixedVemlTrackingInternalTest.ConnectAndRun
Tracking of Galileo E1B signal started on channel 0 for satellite Galileo PRN E11 (Block IOV-PFM)
Processed 40000000 samples in 1.23922e+06 microseconds
[       OK ] GalileoE1MixedVemlTrackingInternalTest.ConnectAndRun (1239 ms)
[ RUN      ] GalileoE1MixedVemlTrackingInternalTest.ValidationOfResults
Tracking of Galileo E1B signal started on channel 0 for satellite Galileo PRN E11 (Block IOV-PFM)
Tracked 80000000 samples in 487.427 microseconds
[       OK ] GalileoE1MixedVemlTrackingInternalTest.ValidationOfResults (1 ms)
[----------] 3 tests from GalileoE1MixedVemlTrackingInternalTest (1241 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (1241 ms total)
[  PASSED  ] 3 tests.
```

Figure 2: Galileo E1 unit test output when using `mixed_veml_tracking`.

The mixed tracking implementation passed the unit test, and performed the same number of iterations as the DLL+PLL implementation in a slightly shorter amount of time.

Finally, the performance of the mixed tracking implementation was evaluated with real signals collected using a hardware receiver at CTTC in 2013. The recorded data

```
linux; GNU C++ version 7.3.0; Boost_106501; UHD_003.010.003.000-0-unknown

Running GNSS-SDR Tests...
Note: Google Test filter = GalileoE1DllPllVemlTracking*
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from GalileoE1DllPllVemlTrackingInternalTest
[ RUN      ] GalileoE1DllPllVemlTrackingInternalTest.Instantiate
[       OK ] GalileoE1DllPllVemlTrackingInternalTest.Instantiate (0 ms)
[ RUN      ] GalileoE1DllPllVemlTrackingInternalTest.ConnectAndRun
Tracking of Galileo E1B signal started on channel 0 for satellite Galileo PRN E11 (Block IOV-PFM)
Processed 40000000 samples in 1.28492e+06 microseconds
[       OK ] GalileoE1DllPllVemlTrackingInternalTest.ConnectAndRun (1286 ms)
[ RUN      ] GalileoE1DllPllVemlTrackingInternalTest.ValidationOfResults
Tracking of Galileo E1B signal started on channel 0 for satellite Galileo PRN E11 (Block IOV-PFM)
Tracked 80000000 samples in 499.975 microseconds
[       OK ] GalileoE1DllPllVemlTrackingInternalTest.ValidationOfResults (1 ms)
[----------] 3 tests from GalileoE1DllPllVemlTrackingInternalTest (1287 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran. (1287 ms total)
[  PASSED  ] 3 tests.
```

Figure 3: Galileo E1 unit test output when using `dll_pll_veml_tracking`.

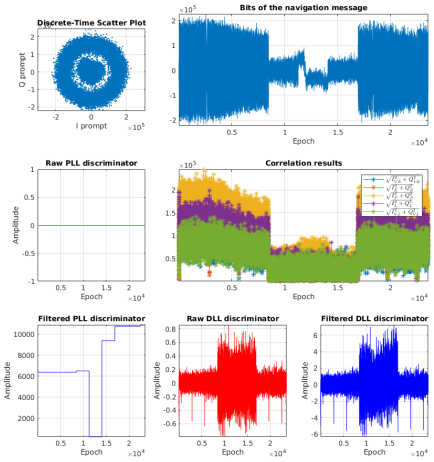

Figure 4: Mixed Tracking

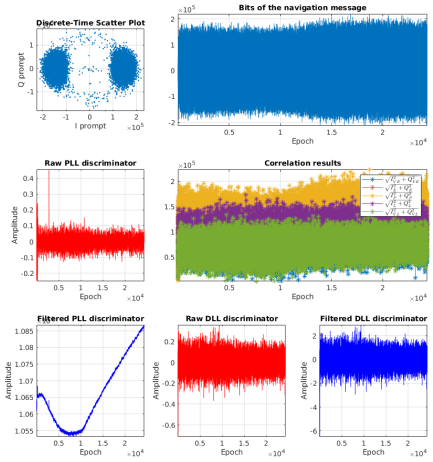

Figure 5: DLL+PLL

was played back through the GNSS-SDR datapath using the Galileo E1 adapter for the `mixed_veml_tracking` GNU radio block and the output of the tracking dataloggers stored and plotted using MATLAB for visual analysis. Figures 4 and 5 show the results of the tracking for Galileo E1 PRN 11. It is evident from these results that the mixed tracking block is not able to track the carrier synchronization parameters with sufficient accuracy to acquire a position fix. Although this is not the result we were hoping for, the filter code was able to perform its task without error. With more time spent tuning the Kalman filter and model parameters, we're confident that the results of our mixed tracking block should match those of its DLL+PLL based counterpart.

4. KALMAN FILTER BASED JOINT CODE-CARRIER TRACKING

The final goal of out 2019 Google Summer of Code contribution to the GNSS-SDR project was to develop a Kalman filter based, discriminator free *joint code-carrier* tracking implementation. Such an implementation uses a nonlinear Kalman filter to replace the FLL+PLL portion of the traditional locked loop for estimating the carrier phase, Doppler and Doppler rate parameters, as well as the DLL portion of that architecture for estimating the code error. As with the discriminator free carrier tracking implementation,

this filter would also have to operate based on a model which directly relates the carrier phase to the output of the correlation step. Unlike the carrier tracking implementation however, estimation of the code error needs to be done using multiple correlator outputs. The number of correlator outputs that may be used depends largely on the computational load, which in turn depends on the tracking algorithm being used. In the case of a sampling algorithm such as the cubature Kalman filter, the prediction step of the filter requires that the measurement equation be evaluated $2N$ times where N is the number of states. Since the number of states that need to be estimated is around $N = 5$, the number of correlations to be computed is $10M$ where M is the number of correlators. In this section we will provide a model for joint tracking based on Early, Prompt, Late (EPL) with $M = 3$, and explain how this model was implemented in GNSS-SDR. Finally, we will explain some of the issues which still need to be resolved before this implementation can be considered functional.

4.1. Correlator output Joint Code-Carrier Model

The joint measurement model extends the model for the output of the correlation step as described in (1) to include the early and late outputs, as well as the relationship between the complex correlator outputs and the code error:

$$\begin{bmatrix} \mathbf{y}_{e,k} \\ \mathbf{y}_{p,k} \\ \mathbf{y}_{l,k} \end{bmatrix} = A_k d_k \begin{bmatrix} R_c(\Delta z_k - \delta z) \\ R_c(\Delta z_k) \\ R_c(\Delta z_k + \delta z) \end{bmatrix} e^{i\boldsymbol{\Delta\phi_k}} \tag{5}$$

As before, A_k is the signal amplitude and $\Delta\phi_k$ is the carrier phase error at sample k. Additionally, $R_c(\cdot) \in \mathbb{R}$ is the autocorrelation of the local code with code error Δz_k, and δz is the spacing between the Early and Prompt or Prompt and Late correlations. A similar trick is used to get rid of the bit transitions as in (2), giving the updated complex measurement model

$$\begin{bmatrix} \tilde{\mathbf{y}}_{e,k} \\ \tilde{\mathbf{y}}_{p,k} \\ \tilde{\mathbf{y}}_{l,k} \end{bmatrix} = \tilde{A}_k \begin{bmatrix} R_c(\Delta z_k - \delta z)^2 \\ R_c(\Delta z_k)^2 \\ R_c(\Delta z_k + \delta z)^2 \end{bmatrix} e^{2i\boldsymbol{\Delta\phi_k}} \tag{6}$$

Separating the real and imaginary components of this and accounting for additive noise gives the final real valued measurement model to be used in the filter

$$\Re\{\tilde{\mathbf{y}}_{e,k}\} = \tilde{A}_k R_c(\Delta z_k - \delta z)^2 \cos(2\boldsymbol{\Delta\phi_k}) + \boldsymbol{\eta_{e,k,\Re}} \tag{7}$$

$$\Im\{\tilde{\mathbf{y}}_{e,k}\} = \tilde{A}_k R_c(\Delta z_k - \delta z)^2 \sin(2\boldsymbol{\Delta\phi_k}) + \boldsymbol{\eta_{e,k,\Im}} \tag{8}$$

$$\Re\{\tilde{\mathbf{y}}_{p,k}\} = \tilde{A}_k R_c(\Delta z_k)^2 \cos(2\boldsymbol{\Delta\phi_k}) + \boldsymbol{\eta_{p,k,\Re}} \tag{9}$$

$$\Im\{\tilde{\mathbf{y}}_{p,k}\} = \tilde{A}_k R_c(\Delta z_k)^2 \sin(2\boldsymbol{\Delta\phi_k}) + \boldsymbol{\eta_{p,k,\Im}} \tag{10}$$

$$\Re\{\tilde{\mathbf{y}}_{l,k}\} = \tilde{A}_k R_c(\Delta z_k + \delta z)^2 \cos(2\boldsymbol{\Delta\phi_k}) + \boldsymbol{\eta_{l,k,\Re}} \tag{11}$$

$$\Im\{\tilde{\mathbf{y}}_{l,k}\} = \tilde{A}_k R_c(\Delta z_k + \delta z)^2 \sin(2\boldsymbol{\Delta\phi_k}) + \boldsymbol{\eta_{l,k,\Im}} \tag{12}$$

The state, $\mathbf{x}_k = [A_k, \Delta z_k, \Delta\phi_k, f_{d,k}, \dot{f}_{d,k}]$ is designed to incorporate the carrier parameters, code parameters, and the signal amplitude. The carrier parameters and signal amplitude are modelled as in 3.1, while the code error is modelled as

$$\Delta z_{k+1} = \Delta z_{k+1} + \beta T_s f_{d,k} + \frac{\beta}{2} T_s^2 \dot{f}_{d,k} \tag{13}$$

Here T_s is the sampling period and $\beta = f_c T_c$ is the number of code chips per radian, computed by multiplying the carrier frequency f_c with the code period T_c in chips. The functionals which implement the transition and measurement functions which comprise this model are located in `joint_veml_tracking.h`, although for brevity they will not be included in full in this report.

4.2. Notes on Code Implementation

An important feature of this measurement model implementation is that, as previously described, the measurement model needs to be able to perform arbitrary evaluations of the autocorrelation function of the local code. To this end, the functional which implements this needs to be extended to include an instance of `Cpu_Autocorrelator_Real_Codes`, which may be found in the `cpu_autocorrelator_real_codes` tracking library. This object adapts the `Cpu_Multicorrelator_Real_Codes` object: instead of accepting a real code and a complex input sequence, it instead accepts only the real code. It then makes a copy of the code as `gr_complex` in order to perform correlation between the real and complex copies. This inefficient solution is a workaround for the fact that the current iteration of the `volk-gnsssdr` library contains functions for correlating complex valued inputs with real valued codes, but does not include the operators necessary to perform standard real-valued autocorrelations. This may be rectified in a future update.

The joint carrier-code tracking block was implemented into a GNU radio tracking block which may be found in the GNSS-SDR source directory as

```
/src/algorithms/tracking/gnuradio_blocks/joint_veml_tracking*
```

Similarly to the mixed carrier tracking implementation described in 3.2, this code is adapted from `dll_pll_veml_tracking`. Here, the functionality of the original PLL+FLL for performing carrier tracking *and* DLL for performing code tracking are replaced with an instance of `TrackingNonlinearFilter`. The most important differences between this joint code-carrier Gaussian tracking block and the original locked loop tracking block are found in `joint_veml_tracking::run_dll_pll`. This function is inappropriately named as the function no longer makes use of either DLL or PLL structures, but the original naming convention was left for compatibility with the rest of the GNSS-SDR code. In this function, the parameters of the filter are initialized with the tracking block, and the parameters of the functionals representing the model are updated as well before performing tracking with `get_carrier_nco`. The filter outputs are then used to compute the parameters that need to be passed to the NCO just as they would be with the standard locked loop architecture.

4.3. Testing and Results

Testing of the `joint_veml_tracking` GNU radio block was performed by implementing an adapter block and unit test for running this tracking block with Galileo E1 signals. The tracking block and unit test can be found in the GNSS-SDR source repository as

```
/src/algorithms/tracking/adapters/galileo_e1_joint_veml_tracking*
/src/tests/.../tracking/galileo_e1_joint_veml_tracking_test.cc
```

As before, the adapter and unit test are based on those implemented for Galileo E1 DLL+PLL tracking, however the joint code-carrier tracking implementation *does not* pass this test as running the tracking block results in a buffer error from the GNU radio kernel. As of the writing of this report, this issue remains unresolved. It is hypothesized that this issue arises from an issue within the GNU radio architecture resulting from having too many instances of `cpu_autocorrelator_real_codes` being polled at a given time. Although it was not possible to investigate this further within the timeframe provided for Google Summer of code 2019, the proposed solution is to develop a more efficient way of evaluating the local code autocorrelation function $R_{cc}(\cdot)$ as described in 4.1 and this should be the subject of future work.

```
linux; GNU C++ version 7.3.0; Boost_106501; UHD_003.010.003.000-0-unknown

Running GNSS-SDR Tests...
Note: Google Test filter = GalileoE1JointVemlTracking*
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from GalileoE1JointVemlTrackingInternalTest
[ RUN      ] GalileoE1JointVemlTrackingInternalTest.Instantiate
[       OK ] GalileoE1JointVemlTrackingInternalTest.Instantiate (0 ms)
[ RUN      ] GalileoE1JointVemlTrackingInternalTest.ConnectAndRun
Tracking of Galileo E1B signal started on channel 0 for satellite Galileo PRN E11 (Block IOV-PFM)
run_tests: /build/gnuradio-BBYmSv/gnuradio-3.7.11/gnuradio-runtime/include/gnuradio/buffer.h:179: uns
igned int gr::buffer::index_add(unsigned int, unsigned int): Assertion `s < d_bufsize' failed.
./run_galileo_dll_pll_veml_test.sh: line 3: 19918 Aborted                 (core dumped) ./run_tests -
-gtest_filter=GalileoE1JointVemlTracking* --gtest_output=xml:test_results.xml
```

Figure 6: Galileo E1 unit test output when using `joint_veml_tracking`.

5. CONCLUSIONS AND FUTURE WORK

Over the course of Google Summer of Code 2019 we have made several contributions to GNSS-SDR open-source GNSS receiver project. As the report shows, although each of the goals was met in part there is still significant work left to be done before these Gaussian filter based implementations are able to perform to the level of the standard locked loop implementations. Nevertheless, significant progress has been made with regard to building the software foundations for each of these these novel tracking architectures, and it is our firm belief that with a bit more work in tuning and bugfixing, these implementations may soon be made operational.

In summary, the contributions we have made this summer began with the development of a set of modular support libraries for building nonlinear models and applying them to the task of tracking with various Gaussian tracking implementations. Following this, we incorporated those nonlinear modelling and tracking libraries into the development of discriminator free, mixed tracking architecture based on cubature Kalman filter and DLL based carrier and code tracking respectiveily. Finally, we expanded the discriminator free mixed implementation to include a nonlinear code error estimation model, including the inclusion of code for performing autocorrelations with the local code.

The immediate future work is self evident: issues remain in getting the mixed tracking architecture to perform to the standard of the traditional DLL+PLL architecture, and in getting the joint tracking architecture to operate correctly within the GNU radio framework. Following this, there are many possible directions where this research can be taken but one in particular stands out as being of immediate interest. Although not included in this year's Google Summer of Code goals, one of the motivations for this year's contribution was to facilitate the use of the Bayesian Covariance estimation method that we worked on building in our 2018 Google Summer of Code contribution to the GNSS-SDR project. Finally, additional nonlinear tracking methodologies, including those based on Monte Carlo methods rather than Kalman architectures, could be built and tested based on the framework that we constructed in this contribution.

6. ACKNOWLEDGMENT

This work has been in part supported by the Spanish Ministry of Economy and Competitiveness through project TEC2015-69868-C2-2-R (ADVENTURE) and the Government of Catalonia under Grant 2017–SGR–1479.

REFERENCES

[1] I. Arasaratnam and S. Haykin. Cubature Kalman filters. *IEEE Trans. Automatic Control*, 54(6):1254–1269, June 2009.

[2] S. J. Julier and J. K. Ulhmann. Unscented filtering and nonlinear estimation. *Proc. of the IEEE*, 92(3):401–422, March 2004.

[3] Gerald LaMountain, Jordi Vila-Valls, and Pau Closas. Bayesian covariance estimation for kalman filter based digital carrier synchronization. In *Proceedings of the 31st International Technical Meeting of The Satellite Division of the Institute of Navigation (ION GNSS+ 2018)*. Institute of Navigation, October 2018.

[4] J. Vilà-Valls, P. Closas, M. Navarro, and C. Fernández-Prades. Are PLLs dead? A tutorial on Kalman filter-based techniques for digital carrier synchronization. *IEEE Aerospace & Electronic Systems Magazine*, July 2017.