Demonstration of C++ Implementation of GNSS-RO Forward Operators

Introduction

As explained in the previous issue of this newsletter series, Radio Occultation measurement based on Global Navigation System Satellites (GNSS-RO), such as GPS, GLONASS, and Galileo, is a remote sensing technique for atmospheric limb sounding (Fjeldbo 1971, Melbourne 1994, Kursinski 1997 and Gleisner 2013) and retrieval of atmospheric properties, such as temperature, pressure, and humidity profiles. The mathematical theory behind the evaluation of radio occultation signals is quite elegant (Fjeldbo 1971), and has been applied in many operational GNSS-RO forward operators, as well as the Joint Effort for Data Assimilation Integration (JEDI) system currently being developed by the JCSDA (Shao et al., 2019).

The purpose of this short communication is to provide an overview on how to implement this theory in the C++ programming language (Stroustrup 1997). While research implementations of these methods already exist in Fortran, the motivation for advancing to C++ is clear. The consequent implementation of the Object-Oriented Programming (OOP) paradigm and C++'s template system drastically simplifies cooperation on and maintenance and extension of larger code bases compared to Fortran without sacrificing runtime speed, which is important in the context of the JCSDA. Moreover, connecting GNSS-RO algorithms as Unified Forward Operators (UFOs) to the Joint Effort on Data Assimilation Integration (JEDI) framework (Trémolet 2019) is also simplified, and potential for error is reduced since the architecture and class structure of JEDI are implemented in C++ as well.

The implementation of a selection of GNSS-RO operators will be discussed. Section 2 describes the Abel transform approach, while the more general ray tracing approach is considered in Section 3. All the C++ implementations discussed consist of only minimal proof-of-concept codes and are not included in actual data assimilation frameworks.

The Abel Transform

The Abel transform is an integral equation formulated by Norwegian mathematician Niels Henrik Abel (Abel 1826) to describe the falling motion of a point mass. For the simplified case of a cylindrically symmetric atmosphere, the Abel transform can be used to invert the refractive index profile from the GNSS signal bending angle (Fjeldbo 1971):

(1)
$$\ln\big(n(a)\big) = \frac{1}{\pi} \cdot \int_a^\infty \frac{\alpha(x)}{\sqrt{x^2 - a^2}}\, dx$$

where n is the refractive index profile of the atmosphere, a the impact parameter of the light ray, and α the bending angle of the GNSS signal. In order to evaluate the integral in ***Equation 1*** for an arbitrary α , a quadrature scheme needs to be chosen. In the current naïve implementation, the exp-sinh quadrature is chosen, which is a double-exponential quadrature scheme (Takahasi 1974) suitable for positive half-infinite intervals.

An important aspect of efficient C++ programming is using suitable and well-tested libraries to avoid reinventing the wheel in a figurative sense. A widely used collection of algorithms and data structures in C++ is the so-called boost library. Features first implemented in boost often find their way in the C++ standard library. As boost already contains an implementation of the exp-sinh quadrature, it is not necessary to repeat the coding of this quadrature scheme here. In order to use the quadrature scheme, it is only necessary to include the corresponding boost header file, as shown in ***Listing 1***.

For the sake of simplicity, the Abel transform is implemented as a function here. C++ functions largely work the same way as their familiar Fortran versions. In order to have a more flexible implementation however, the Abel transform will accept and return a template argument; that is, the return type of the function and its input arguments is determined by the compiler. ***(Listing 2)*** In this way, the return type of the function can be changed without having to write a separate implementation of the function for each type.

The output type of the function Abel can now be any arbitrary class that implements the necessary algebra, including float, double, vectors, matrices, quaternions, and so on. The integrand of ***Equation 1*** can be passed to quadrature method as a lambda function ***(Listing 3)***, which is a (usually short) anonymous function that can be defined locally. This is an advanced feature introduced in the C++11 standard.

Listing 1. *Including a static library header file in C++.*

```
#include <iostream>
#include <boost/math/quadrature/exp_sinh.hpp>
```

Listing 2. *The return type of the Abel function and the type of its input argument is the template parameter T.*

```
template<typename T>
T Abel(T a)
{ ...
}
```

Listing 3. *Defining the integrand as a lambda function.*

```
auto integrand = [alpha,a](T x)
{
  return alpha/sqrt(x*x+std::numeric_limits<double>::epsilon() - a*a);
};
```

Finally, in order to compute the integral, the quadrature method is instantiated as an object of class exp_sinh ***(Listing 4)***, and the integrand is passed to the integrate(-) method of the object *integrator*.

This completes a very basic implementation of the Abel transform method in C++.

Hamiltonian Optics

A more general but also more expensive approach to computing the path of the GNSS-RO signal is ray tracing (Fjeldbo 1971). The assumption of a circularly symmetric refractive index profile is not needed anymore, and the method is indeed very general. This is possible because the wavelength of the GNSS signal is very small in relation to all other length scales of the problem. Under these circumstances, the GNSS-RO problem can be described as a Hamiltonian system (i.e., the ray trajectory x and the wave vector *k* of the ray are functions of a single scalar function only, namely the Hamiltonian *H*, which can be identified as the total energy of the system) (Ott 2002 and Stegmann 2016).

(2)
$$\frac{\mathrm{dx}}{\mathrm{dt}} = \frac{\partial H}{\partial k}$$
$$\frac{dk}{dt} = -\frac{\partial H}{\partial x}$$

The system itself has the property that it is symplectic (i.e., the volume of a phase space element $\omega = dx \wedge dk$ is conserved by the system's mapping and the light rays are the path that minimizes the optical length between sender and receiver, which is a function of the refractive index profile). In order to solve ***Equation 2*** numerically, an ODE solver algorithm is required. In particular, it is highly desirable to use an integrator that retains the symplectic property (Ruth 1983) of the ***Equation 2*** instead of e.g., conventional Runge-Kutta methods.

For the practical implementation, we again rely on the boost library, specifically its odeint component. This is a library that contains a broad range of ODE solvers for many kinds of problems.

Looking at ***Equation 2***, the split nature of the ODE in terms of the ray trajectory and wave vector is obvious. Consequently, the RHS functions of Equation 2 are implemented as two different functions in the code . ***(Listings 5 and 6)***

As a matter of fact, the RHS of ***Equation 2*** is now not implemented as a function anymore, but as a functor or function object. As the name implies, a functor is a class object

Listing 4. *Instantiation of an object of class exp_sinh.*

```
boost::math::quadrature::exp_sinh<T> integrator;
double termination = sqrt(std::numeric_limits<double>::epsilon());
double error;
double L1;
return exp(integrator.integrate(integrand,
                                a,
                                boost::math::tools::max_value<double>(),
                                termination,
                                &error,
                                &L1 )/M_PI);
```

Listing 5. *Ray trajectory functor implementation.*

```
//[ coordinate_function

struct ray_coor
{

    ray_coor(){ }

    void operator()( const container_type &p , container_type &dqdt ) const
    {
        for( size_t i=0 ; i<n ; ++i )
            dqdt[i] = p[i];
    }
};
//]
```

(Barton 1994) with an overloaded input(-) operator and is a common programming construct in C++. As a specific simple test case, the Luneburg lens refractive index profile has been chosen here:

Listing 6. *Wave vector functor implementation.*

```
//[ momentum_function
struct ray_momentum
{

    ray_momentum(){ }

    void operator()( const container_type &q , container_type &dpdt ) const
    {
        const size_t n = q.size();
        for( size_t i=0 ; i<n ; ++i )
        {
            dpdt[i].m_val[0] = -1.0*q[i].m_val[0];
            dpdt[i].m_val[1] = -1.0*q[i].m_val[1];
        }
    }
```

(3)

$$n(r) = \sqrt{2 - r^2}$$

After implementing the RHS of ***Equation 2***, an ODE solver type needs to be selected. This is done in ***Listing 7***. A symplectic solver is selected as the stepper_type ***(Listing 7)*** and passed to the integrate_const function provided by boost odeint.

Sample output of the ODE integration is presented in ***Figure 1***. The image shows a single ray trajectory curving back in on itself and moves around the center of the coordinate system on a stable orbit. As the light ray cannot leave the Luneburg lens, it effectively acts as a black hole.

Summary and Future Plans

As discussed in the previous newsletter (Shao et al., 2019), the JEDI system being developed at the JCSDA adopts the operators from the existing operating systems, written in Fortran. However, the goal of the GNSS-RO work at the JCSDA is to develop a GNSS-RO operator with both

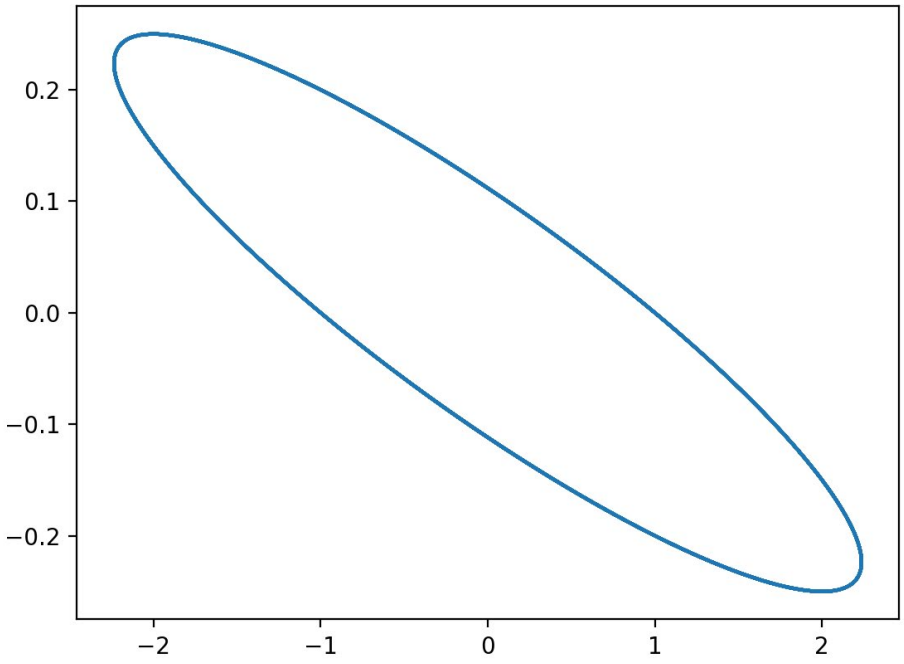
Listing 7. *Integration of the ODE with constant step width.*

```
//[ integration_GPS_signal
typedef symplectic_rkn_sb3a_mclachlan< container_type > stepper_type;
const double dt = 0.1;

integrate_const(
        stepper_type() ,
        make_pair( ray_coor( ) , ray_momentum( ) ) ,
        make_pair( boost::ref( q ) , boost::ref( p ) ) ,
        0.0 , 50.0 , dt , streaming_observer( cout ) );
//]
```

scientific and computational advancements. This article demonstrates an alternative to current implementations of GNSS-RO operators using modern high-level programming languages (e.g., C++ in this article), as well as carefully selected modern mathematical programming libraries. This is the first step to explore the possibility to advance the GNSS-RO assimilation through alternative computational implementation. Using C++, the essential parts of two types of bending angle forward operators were successfully implemented, either in a form of an Abel inversion or through solving the ray trajectory equation in an idealized refractivity environment. While the real atmospheric properties will complicate such an implementation, the results certainly are promising. The JCSDA will further investigate feasibility and sufficiency of such an implementation for GNSS-RO with a goal of improving the use of observations and eventually improving numerical weather forecasts.

Figure 1. *Ray trajectory for the Luneburg lens profile Equation 3.*



Authors
Patrick G. Stegmann (JCSDA), Benjamin T. Johnson (JCSDA) and Hui Shao (JCSDA) stegmann@ucar.edu

Acknowledgements
Funding for this project was provided by the JCSDA partners, including NOAA, NASA, and the Department of Defense, and is gratefully acknowledged. Calculations were carried out at the S4 Supercomputing Facilities of the University of Wisconsin Madison, and the authors gratefully acknowledge the facility staff for their help and assistance, particularly creating a boost module from scratch.

References

Fjeldbo, G., A. J. Kliore, and R. Eshleman (1971): The Neutral Atmosphere of Venus as Studied with the Mariner V Radio Occultation Experiments. Astr. J. 76(2), 123-140.

Melbourne, W. G., E. S. Davis, G. A. Hajj, K. R. Hardy et al. (1994) : Application of Spaceborne GPS to Atmospheric Limb Sounding and Global Change Monitoring. JPL Publication 94-18.

Kursinski, E. R., G. A. Hajj, J. T. Schofield, R. P. Linfield, and K. R. Hardy (1997): Observing Earth's atmosphere with radio occultation measurements using the Global Positioning System. J. Geophys. Res. 102(D19), 23429-23456.

Gleisner, H. and S. B. Healy (2013): A simplified approach for generating GNSS radio occultation refractivity climatologies. Atmos. Meas. Tech. 6, 121-129.

Shao, H., F. Vandenberghe, H. Zhang, B. Ruston, S. Healy, and L. Cucurull (2019): Development of GNSS-RO Operators for JEDI/UFO. JCSDA Quarterly Newsletter 62, DOI:10.25923/w2dh-ep66.

Stroustrup, B. (1997): The C++ Programming Language (Third Ed.). ISBN 0-201-88954-4.

Trémolet, Y. (2019): The Joint Effort for Data Assimilation Integration (JEDI). Seventh AMS Symposium on the Joint Center for Satellite Data Assimilation.

Abel, N. H. (1826). Journal für die reine und angewandte Mathematik 1, 153-57.

Takahasi, H., and M. Mori (1974): Double Exponential Formulas for Numerical Integration. Publ. Res. Inst. Math. Sci. 9(3), 721-741.

https://www.boost.org (Retrieved 2019-04-14).

Ott, E. (2002): Chaos in Dynamical Systems (Second Ed.). Cambridge University Press.

Stegmann, P. G. (2016): Light Scattering by Non-Spherical Particles. Dissertation. URN urn:nbn:de:tuda-tuprints-52570.

Ruth, R. D. (1983): A canonical integration technique. IEEE Trans. Nucl. Sci. NS-30(4), 2669-2671.

Barton, J. J., and L. R. Nackman (1994): Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples. Addison Wesley Longman Inc., Boston.