

APL/UW Doppler Toolkit

Rex K. Andrew

April 25, 2007

1 Introduction

The Doppler toolkit is a collection of C++ classes that provide functions and features necessary for removing the timebase contraction in multi-channel time series data. The principle feature of the toolkit is a multi-channel systolic digital resampling filter; the primary numerics in the toolkit provide the mapping from transmitter time to receiver time, and vice versa. The toolkit includes interfaces for some common (albeit custom) acoustic data file formats, and interfaces to several formats of tabulated platform 3-D position.

This toolkit was developed under the ONR Long Range Deep-Water Propagation effort, primarily for the 2004 Long Range Ocean Acoustic Propagation Experiment (LOAPEX). This experiment involved a ship-suspended transmitter deployed to depths from 350 m to 800 m in the North Pacific. Signals were sent from the transmitter to a variety of receivers, including bottom-moored vertical line arrays (VLAs) and bottom-moored horizontal line arrays. A prominent feature of this experiment is that the transmitter (the VLAs) moved around the design position on time scales of minutes (hours), and on spatial scales of 10 m (hundreds of meters). This experiment therefore involved a moving transmitter platform, and possibly a moving receiver platform. The toolkit evolved from the need to understand the effects on the received signal of the interplatform motion, and the possible need to remove those effects.

Although built for analysis and processing of LOAPEX 2004 data, the toolkit was designed to be easily extended to handle scenarios for other moving platform experiments. The only restriction are that the 3-D positions of the transmitter and receivers with respect to time are known and have been provided in data files, and the platform velocities are much less than

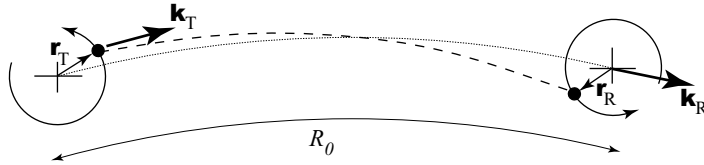


Figure 1: Geometric definitions for the moving transmitter/ moving receiver problem. The structure of the model assumes that the transmitter and receiver coordinate systems are separated by a distance much greater than the spatial displacements of either the transmitter or the receiver in their own coordinate system. Also, the model involves no dependence in the vertical. The problem context involves considerable distances through the earth’s oceans, and therefore the Fermat path of signal propagation between the transmitter site and the receiver site, which is the “line of sight” path, is represented as a dotted curve.

the local Mach number, which is the case in almost all underwater acoustic scenarios.

This documentation includes below a discussion of the mathematical fundamentals and numerical implementations, class interfaces, the suite of class test code, and some examples of using the toolkit in complete applications.

The source was developed under GCC 3.3.5 on various Linux machines and uses the STL. The code can be obtained by email from Rex Andrew, randrew@apl.washington.edu.

2 Principles of Operation

A schematic diagram of the model geometry is shown in Fig. 1. The transmitter is located at $\mathbf{r}_T(t)$ in its own coordinate system, and the receiver at $\mathbf{r}_R(t)$ in its own coordinate system. The vector \mathbf{R}_0 (not shown) locates the receiver frame-of-reference relative to the transmitter. The y -axes of these coordinate systems are aligned along local north-south meridians, anticipating the context of operations on the earth’s surface. The “line of sight” line from transmitter origin to receiver origin is effectively a straight line for short distances, but for long range propagation problems, this line may be a curved geodesic track. In either case, this line has arc length R_0 . The vector at the transmitter along the line of sight is $\hat{\mathbf{k}}_T$ and at the receiver is $\hat{\mathbf{k}}_R$: neither are functions of time.

The toolkit assumes a single speed of signal propagation c_0 . This has

been adequate for the LOAPEX experiment, in which errors (estimated to be one or two seconds) due to inaccurate modeling of the signal propagation speed are much less than the shortest time scale of platform motion, which was tens of seconds.

Let t be the time on the transmitter platform. Following Gondeck[1], the corresponding time on the receiver is

$$\mathcal{T}(t) = t' \quad (1)$$

$$= t + |\mathbf{R}_0 + \mathbf{r}_R(t') - \mathbf{r}_T(t)|/c_0 \quad (2)$$

That is, if a pulse is emitted from the transmitter at time t , when the transmitter is at position $\mathbf{r}_T(t)$, it arrives at the receiver at time t' , when the receiver is at position $\mathbf{r}_R(t')$. If the transmitted signal is $x(t)$, the received signal is $y(t') = y(\mathcal{T}(t)) = x(t)$.

Therefore, to first order in $|\mathbf{r}_T|/|\mathbf{R}_0|$ and $|\mathbf{r}_R|/|\mathbf{R}_0|$ (which here are approximately the same), this becomes

$$\mathcal{T}(t) \approx t + (R_0 + \mathbf{r}_R(t') \cdot \hat{\mathbf{k}} - \mathbf{r}_T(t) \cdot \hat{\mathbf{k}})/c_0 \quad (3)$$

where $\hat{\mathbf{k}} = \mathbf{R}_0/|\mathbf{R}_0|$. However, this ‘‘line of sight’’ vector varies in direction along the propagation path, as shown in Fig. 1, so the local line of sight vector must be substituted at both the transmitter and receiver locations:

$$\mathcal{T}(t) \approx t + (R_0 + \mathbf{r}_R(t') \cdot \hat{\mathbf{k}}_R - \mathbf{r}_T(t) \cdot \hat{\mathbf{k}}_T)/c_0 \quad (4)$$

The mapping $\mathcal{T}(t)$ is one-to-one and monotonic, and therefore admits an inverse mapping

$$t = \mathcal{T}^{-1}(t'). \quad (5)$$

The principle effort, therefore, is to provide the forward and reverse maps $\mathcal{T}(t)$ and $\mathcal{T}^{-1}(t')$.

The following discussion assumes that 3-D positions $\mathbf{r}_T(t)$ and $\mathbf{r}_R(t)$ are continuous smooth functions of time. In reality, track solutions of actual platforms are provided at discrete times t_1, t_2 , etc. The toolkit contains a class that converts tabulated (time,position) pairs into a smooth, continuous representation (see section 3.5), and therefore it will be assumed that such functions exist.

2.1 The Forward Map

We move all the terms on the righthand side of Eq. 4 to the lefthand side to obtain

$$f(t'; t) = t' - t - (R_0 + \mathbf{r}_R(t') \cdot \hat{\mathbf{k}}_R - \mathbf{r}_T(t) \cdot \hat{\mathbf{k}}_T)/c_0 = 0. \quad (6)$$

This is a straightforward 1-D rooting problem in t' , which is solved numerically for tabulated position data using the van Wijngaarden-Dekker-Brent algorithm[2].

During object initialization, the forward problem is solved for \mathcal{T} on a grid of t , and the pairs $(t, \mathcal{T}(t))$ retained as a tabulated function. (This happens in `LOAPEX::CTimeBaseMapEngine::build_map`, which calls the `compute_map()` method on the engine solver. This is primarily a `CBrentMapSolver`, which employs the van Wijngaarden-Dekker-Brent rooter. Alternatively, one can install the secondary `CNewtonMapSolver`, which which incorporates a Newton-Raphson algorithm[2]. This is provided to validate the accuracy of the primary algorithm. This secondary solver, however, requires analytic derivatives of transmitter and receiver position versus time, and is set up only for the test case, where these derivatives are available.

Subsequent calls to evaluate the forward map interpolate on the tabulated function pairs $(t, \mathcal{T}(t))$ to return the interpolant t' for a given t .

I guess I thought this would be faster in the long run.

2.2 The Inverse Map

We move all the terms on the righthand side of Eq. 4 to the lefthand side to obtain

$$g(t; t') = t' - t - (R_0 + \mathbf{r}_R(t') \cdot \hat{\mathbf{k}}_R - \mathbf{r}_T(t) \cdot \hat{\mathbf{k}}_T)/c_0 = 0. \quad (7)$$

This is a straightforward 1-D rooting problem in t , which is solved numerically for tabulated position data again using the van Wijngaarden-Dekker-Brent algorithm[2]. (This code is in `CTimeBaseMapEngine::zbrent` and is different code than that used in the forward case, because f is a function of t' , whereas g is a function of t .)

2.3 Dedopplerization Filter

Dedopplerization is defined here as the operation that converts a received signal $y(t')$ back to an estimate $\hat{x}(\mathcal{T}^{-1}(t'))$ of the transmitted signal $x(t)$. In the absence of any *a priori* information regarding the interplatform motion, the map $\mathcal{T}(t)$ is estimated from the received data itself. This is typically the case with a non-cooperative target, or when the tracking solutions are poor and highly inaccurate, (i.e., a non-cooperative experiment!) In the present scenario, however, we assume a thorough and comprehensive experiment has been conducted, and that we have highly accurate transmitter and receiver

track solutions. Problems of this kind have arisen, for example, in the measurement of radiated noise from vehicles moving past fixed receivers at test ranges[3]. For a generalized problem involving arbitrary interplatform motion, we adopt the approach of Glegg[4]. Formally, the sample dedopplerized process is

$$\begin{aligned}\hat{x}_j &= \hat{x}(t_0 + j\Delta) = y(\mathcal{T}(t_0 + j\Delta)) \\ &= \sum_{k=-\infty}^{\infty} y_k \operatorname{sinc} \sigma[\mathcal{T}(t_0 + j\Delta) - \mathcal{T}(t_0) - k\Delta]\end{aligned}\quad (8)$$

$$\approx \sum_{k=-K}^K y_k \operatorname{sinc} \sigma[\mathcal{T}(t_0 + j\Delta) - \mathcal{T}(t_0) - k\Delta].\quad (9)$$

where σ is the bandwidth of the signal and we assume that the mapping is synchronized at some “epoch” t_0 in the transmitter frame of reference. The summation must be truncated in implementation, hence we use Eq. 9.

Eq. 9 is implemented as an ordinary FIR filter of order $2K + 1$, with weight updates for every new sample j output. The toolkit provides a separate FIR filter for each channel in multi-channel data, see section 3.3.

3 Class Structure

This section highlights the main classes and subclasses of the toolbox. For a complete description of the interfaces, see the Doxygen reference manual.

3.1 Map Engine

The class **CTimeBaseMapEngine** provides the methods that perform the forward and inverse mapping functions, Eqs. 2 and 5. The public interface is:

- **CTimeBaseMapEngine**(const time_t start, const time_t stop): Constructor: takes a start and stop time, in Unix seconds. These are times in the transmitter frame of reference.
- **CTimeBaseMapEngine**(void): This is the destructor.
- void **load_parameters**(const LOAPEX::CGeoParameters& P): Copies the values from the reference into internal private storage. This is also where the “wavenumber unit vectors” $\hat{\mathbf{k}}_T$ and $\hat{\mathbf{k}}_R$ are computed and saved.

- void **load_TX**(const char* name, const int channel): Connects to a netCDF file containing 3-D tracking data. (See **CNetCDFNavigationData** class.) The transmitter is assumed to have only one channel, so the channel option is ignored.
- void **load_TX**(LOAPEX::CBase3DNavigationData *): Overloaded method that takes the pointer to a navigation object. The user is responsible for deallocaating the nav object.
- void **load_RX**(const char* name, const int channel): Connects to a netCDF file containing 3-D tracking data. (See **CNetCDFNavigationData** class.)
- void **load_RX**(LOAPEX::CBase3DNavigationData *): Overloaded method that takes the pointer to a navigation object. The user is responsible for deallocaating the nav object.
- void **arm**(void): Causes the engine to run through some consistency checks, build all necessary internal tables and interpolated objects, and generally get ready to do actual calculations. Must be called before any call to the forward or reverse map methods.

One of the primary consistency checks here is to insure that the start and stop times supplied to the constructor are monotonic. Failure results in this error message:

"Map start time must be earlier than map stop time"

Another consistency checks is to determine if the start and stop times supplied to the constructor fall *within* the start and stop times of the 3-D tracking files. Failure to meet this condition on the start or stop times results respectively in these messages:

"Map start time must be later than initial navigation solutions"

"Map stop time must be earlier than final navigation solutions"

Arming aborts in an invalid state if any of these tests fail.

- bool **is_valid**(void): Returns true if the object has been armed and is ready to do calculations. Returns false if there has been a problem arming, or if **arm**() has not been called.

- void **get_first_time**(time_t *t) :
- void **get_last_time**(time_t *t) : Places the value of the private start or stop time loaded at construction) into t. I put this in so I could conveniently get these times later in code, when I forgotten which times had been supplied to the constructor.
- void **get_forward_time**(const& Tin, *Tout) : Do the forward problem. The arguments are CCompoundTime objects.
- void **get_inverse_time**(const& Tin, *Tout) : Do the inverse problem. The arguments are CCompoundTime objects.

In both of these cases, if the supplied time is outside the span of valid times, the routine emits the error message

"User request outside analysis domain."

For the forward map, the input time t is either before the constructor start time, or after the constructor stop time. For the inverse map, the input time is either prior to $\mathcal{T}(t)$ with t the constructor start time, or after $\mathcal{T}(t)$ with t the constructor stop time. (This is trickier to diagnose, because users typically do not know $\mathcal{T}()$ a priori.)

3.2 Navigation Data Interfaces

These classes encapsulate time-tagged 2-D location and velocity data. They rely internally on a 3-D vector

```
typedef struct
{
    double x, y, z;
} dEuclideanVector_t ;
```

There is a pure virtual base class, **CBase3DNavigationData** which is used primarily to define the interface, but also as a generic navigation object at compile time:

- virtual void **get_file_first_time**(time_t * pt) const: This method returns the time of the first 3-D position.
- virtual void **get_file_final_time**(time_t * pt) const: This method returns the time of the last 3-D position.

- virtual void **get_position**(const LOAPEX::CCompoundTime t, dEuclideanVector_t *X): This method returns $\mathbf{r}(t)$.
- virtual void **get_velocity**(const LOAPEX::CCompoundTime t, dEuclideanVector_t *V): This method returns $\mathbf{v}(t)$.

There are several subclasses provided.

CNetCDFNavigationData : Connects to a netCDF file (i.e., the files I made from Mike Zarnetske’s Matlab files). The position and velocity information is read in and converted into a CInterpolatedTableFunction object.

CFakeRXNavigationData : Class that implements a 3 channel vertical line array tilted with respect to vertical and revolving around the vertical axis. For testing. Geometric parameters hardwired in the file `navigation.hpp`.

CFakeTXNavigationData : Class that implements a single channel transmitter revolving in a horizontal circle. For testing. Geometric parameters hardwired in the file `navigation.hpp`.

CFixedNavigationData : Class that implements an N -channel device whose sensors are all stationary with time. For testing. Geometric parameters hardwired in the file `navigation.hpp`.

3.3 Filter

An object diagram of the filter classes is shown in Fig. 2 and a data flow diagram in Fig. 3. The main class is **CMultiChannelFilter**. There is typically only one instantiation of this class in an application. It contains pointers to multiple single channel filter objects, which implement the “filter bank”. There is one single channel filter object per channel in the input file.

Each single channel filter object is an instantiation of the class **CSingleChannelFilter**. It contains a reference to the **CDemultiplexor** and the **CMultiplexor** object. There is only one **CDemultiplexor** and **CMultiplexor** object per application. The **CDemultiplexor** object provides the interface to the input file. At each clock step, it reads in a scan of (multi-channel) data into a private buffer, and then delivers a sample to each single channel filter in turn as they in turn are clocked. (See the code fragment below for “timing” details.)

Likewise, the **CMultiplexor** object contains a private buffer that receives the output sample from each single channel filter as each is clocked. When the object itself is clocked, it writes the buffer to the output file. (Again, see the code fragment below.)

These classes are described in the following sections.

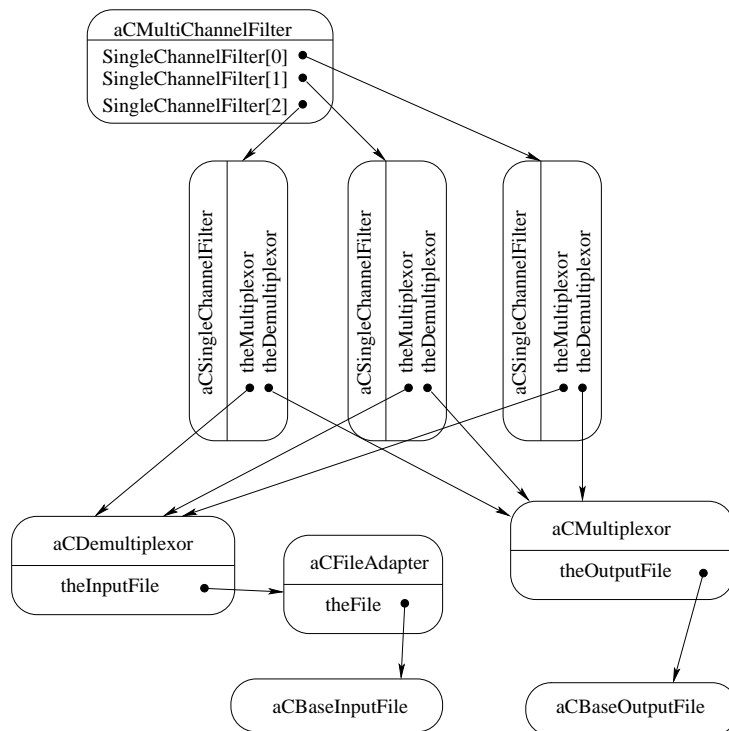


Figure 2: Object diagram for the dedopplerization filter.

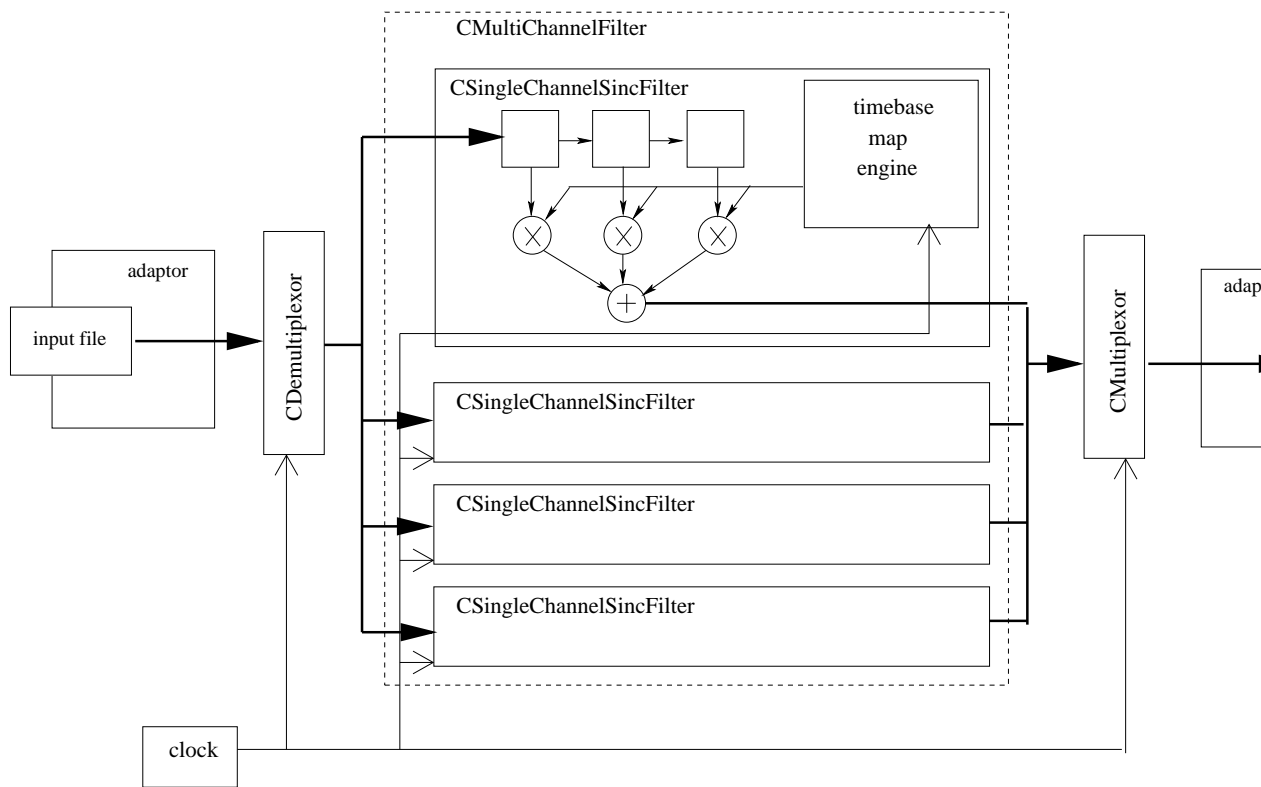


Figure 3: Data flow diagram for the dedopplerization filter.

The following code fragment shows how the multichannel filter works in an application.

```
//instantiate the filter bank
LOAPEX::CMultichannelFilter *MCFB = new LOAPEX::CMultichannelFilter( total_channels );

// instantiate data source and sink
// Here, FA is an input file adaptor, and pMout is a pointer to an output object
LOAPEX::CDemultiplexor * theDemux = new LOAPEX::CDemultiplexor ( FA );
LOAPEX::CMultiplexor * theMux = new LOAPEX::CMultiplexor ( pMout );

// assume we have constructed some LOAPEX::CGeoParameters someParameters;

// now build an individual filter for each channel:
for (int ii=0 ; ii<total_channels; ii++)
{
    // allocate the filter:
    LOAPEX::CSingleChannelSincFilter *p;
    p = new LOAPEX::CSingleChannelSincFilter(ii,ORDER,sample_period);
    // give it the address of the demultiplexor and the multiplexor
    p->set_demultiplexor ( theDemux );
    p->set_multiplexor ( theMux );

    // instantiate a map engine
    LOAPEX::CTimeBaseMapEngine* pMapEngine;
    pMapEngine = new LOAPEX::CTimeBaseMapEngine(1099180800, 1099439999);

    // allocate handles for the transmitter and receiver navigation interfaces:
    LOAPEX::CBase3DNavigationData * pTX;
    LOAPEX::CBase3DNavigationData * pRX;

    // allocate simulated TX and RX objects
    LOAPEX::CFakeTXNavigationData * ptx = new LOAPEX::CFakeTXNavigationData;
    LOAPEX::CFakeRXNavigationData * prx = new LOAPEX::CFakeRXNavigationData(0);
    pTX = dynamic_cast<LOAPEX::CBase3DNavigationData*>(ptx);
    pRX = dynamic_cast<LOAPEX::CBase3DNavigationData*>(prx);

    // store the addresses of these interfaces into the mapper
    pMapEngine->load_TX( pTX );
    pMapEngine->load_RX( pRX );

    // feed the mapper the geometric parameters:
    pMapEngine->load_parameters ( someParameters );

    // arm the engine
    pMapEngine->arm();

    // was arming successful?
    if (!pMapEngine->is_valid())
    {
        throw APL::FatalError("could not construct a valid map engine");
    }

    // finally, pass the handle to the map engine to the filter
}
```

```

    p->set_mapper ( pMapEngine );
    // now the filter is complete

    // put the filter into the multichannel filter bank:
    MCFB->add_filter(p);
}

// multichannel filter ready to go.

```

3.3.1 CBaseFilter class

This is a simple pure virtual base class from which concrete filter classes are derived. The public interface is:

- **CBaseFilter**(const int channel_id): This is the constructor and it takes a channel identifier, in the range $(0, N - 1)$ for an N channel data file.
- **CBaseFilter**(void): This is the destructor.
- virtual void **clock**(void): This is pure virtual, so subclasses must implement this. This is the method that causes the filter to pull a sample from the demultiplexor, rotate the FIR filter, compute the output, and put the output sample to the multiplexor.

3.3.2 CSingleChannelAllPassFilter class

Outputs the input sample at each clock call. For diagnostic use.

3.3.3 CSingleChannelAllStopFilter class

Pulls a sample from the input file but outputs zero at each clock call. For diagnostic use.

3.3.4 CSingleChannelSincFilter class

This is the one you want. This class implements Eq. 9.

3.4 File Interfaces

3.5 Table functions

This is a utility class that interfaces spline code to a tabulated data, with the intent of providing a *function* $f : R^1 \rightarrow R^1$ out of the inherently discrete data. Uses the spline routine from Numerical Recipes in C, 2nd ed.

- **CInterpolatedTableFunction** (const std::vector<double>& x, const std::vector<double>& y) : Constructor, copies x and $y(x)$ into private storage, generates and stores the spline coefficients. I think it may delete the copies of x and $y(x)$ before finishing. (Who needs them at that point?)
- void **evaluate**(const double x, double *y) const: Give it x , get $y(x)$.

3.6 Auxiliary Classes

3.6.1 Compound Time

It seemed necessary to insure that time was uniquely defined at second granularity, and this suggest using integers for seconds, much like Unix time. However, there was also the need to be able to address subsecond instants, but with less accuracy. This argued for a floating point representation. But a floating point representation (such as Matlab's serial date number) may not uniquely identify seconds, due to round-off error. Therefore, I devised a class **CCompoundTime** that has two fields, an integer for Unix-like time and a floating point field for the subsecond "fraction". There are some class methods to normalize the object, i.e., reduce a time with a negative fraction or a fraction > 1.0 to an object with a fraction in $[0, 1.0)$.

3.7 MUX Files

I needed a very simple class and file format for data organized as an integer number of scans, both for reading and (primarily) writing. This is the purpose of the **CMUXFile** class. Such files have the very simple format

```

muxfile
{
    header
    binary data
}

```

This class interfaces either to an existing file (using special mode flag `LOAPEX:CMUXFile::ReadOnly`) or open a new file for writing (using mode flag `LOAPEX:CMUXFile::Write`). Methods are provided to read the basic parameters from the header, or to write the basic parameters into the header. These basic parameters are:

- sample rate (integer)

- start-second (long)
- total scans (long)
- total channels (integer)
- an arbitrary amount of line-oriented description material.

There are also the methods `get_scan()` for reading one scan at a time, or `put_scan()` for writing one scan at a time. The binary data is treated like a stream: no seeking is implemented.

3.8 ADF Files

I wanted in interface to Scripps's `.adf` file format, so I made the class **CADFFile**. This provides simple methods for accessing file collection parameters in the header, and also scans from the binary data section too.

3.9 File Adapters

In order to write somewhat flexible code, I opted to connect the data source file to the demultiplexer via a file adapter. This is just one possible design pattern for doing this, there are others. In this pattern, class **CBaseFileAdapter** is a pure virtual class that defines the interface from the adapter to the demux. The demux only knows about those methods defined in this class.

Two concrete adapters are provided: **CADFFileAdapter** for `.adf` files from Scripps, and the **CMUXFileAdapter** class for my own `/mux` files.

Users wanting to use additional types or formats of input files should consult the code in `adfadapter.cpp` and `muxadapter.cpp` to see how to subclass new adapters.

4 Test Suite

There is a suite of test functions supplied which exercise individual classes or sometimes (for the more complicated tests) a select number of classes working together. There is no universal test command that runs each test and reports success or failure — sorry.

- Test 1: This is a test of the `CADFFile` class. A sample `.adf` file was pulled from random from the SIO website, and its header altered so as to pretend only 5 samples were taken on each channel. The `.adf`

file was truncated to a manageable size. The program `echoADF` reads in this file and exercises several rudimentary class methods. The gold-standard output is in the file `echoADF.output`, verified by hand.

- Test 2: This is a test of the adapter for the `CADFFFile` class. As for the test for the `CADFFFile` class itself, a sample `.adf` file was pulled from random from the SIO website, and its header altered so as to pretend only 5 samples were taken on each channel. The `.adf` file was truncated to a manageable size. The program `testADFadapter` instantiates the `.adf` adapter and registers a `CADFFFile` object in it, and exercises several rudimentary class methods. The gold-standard output is in the file `testADFadapter.output`, verified by hand.
- Test 3: This directory contains a little program that exercises the `CMUXFile` class in write mode. A small output file, named `muxtest.mux`, is defined and created. If this file is then processed through the unix command

```
od -d muxtest.mux > foo
```

then `foo` should be identical to the provided file `muxtest.out.verified`.

- Test 4: This directory contains a test that the `CMUXFile` class operates successfully in file read mode. The wrapper program `testmuxread` read the file `testmuxread.in.mux`, and echos the contents to the screen. The screen dump starts with the class `print()` function (which dumps the object internals) followed by output derived from calling individual methods. These should match. Lastly, the binary data is dumped using the `get_scan()` routine.
The file `testmuxread.output.verified` presents the output when the class is working OK.

- Test 5: This directory contains the test for the program that merges multiple `.adf` files. The test is performed by running `do_test.bsh`. (You may have to make various executables first.)

The test uses `mkfakeADFfile` to make 4 files, naming them `gappy0x.adf`, `x = 1, 2, 3, 4`. then `mergeADFfile` is executed, using the file `gap-spec01.txt` as the input specification. This creates the output "merged" file `test.mux`. The contents of this file are then dumped with the `testmuxread` utility, and piped into a text file `gappytest.trial`. This is compared bytes for byte against the file `gappytest.output.verified`, which

I checked by hand. The two output files will differ slightly in the first 1500 bytes because they contain different creation dates. Otherwise, they should be identical.

The simulation covers 4 files, each 2.5 seconds long, over a total of 12 total seconds. The sample rate is 1200, thus each file contributes 3000 samples, and they are "glued" together with 0.5 second chunks of 600 samples of zero data.

- Test 6: This is a test of the adapter for the CMUXFile class. As for the test for the CMUXFile class itself, a sample .mux file was pulled from test4. The program "testMUXadaper" instantiates the .mux adapter and registers a CMUXFile object in it, and exercises several rudimentary class methods. The gold-standard output is in the file testMUXadapter.output, verified by hand.
- Test 7: This is a test that the demultiplexer and the multiplexer work OK. The filter used is the all pass filter. The data is read from the file testmuxread.in.mux and written out to file testmuxread.out.mux. File testmuxread.out.mux.OK represents the correct output (this is just the same data content as the input file.)
- Test 8: This is a test of the Tx and RX navigation classes that do not open auxiliary files but simply return values from an analytical scenario. In this scenario, the source is moving around a horizontal circle, and there are 3 receivers revolving around consecutively larger circles at consecutively greater depths (i.e, on a cone). Outputs are positions and velocities for the first few minutes of the simulation.
- Test 9: This is a test of the TX and RX navigation classes that do not open auxiliary files but simply return values from an analytical scenario. In this scenario, the source is moving around a horizontal circle, and there are 3 receivers revolving around consecutively larger circles at consecutively greater depths (i.e, on a cone). 50 seconds of motion are simulated, once per second: at each second, the time of arrival is computed at the moving receiver via two methods: (1) an "exact" numerical method, and (2) and "inexact" numerical method. The exact method uses a Newton-Raphson root finder that uses precise derivative information, and the inexact method uses bisection. The tolerance on the solutions is about 1 second. (That is hardcoded into the class.)

The output is the index (0 - 49), the time (at the transmitter), the two solutions from the two root finders, and the difference of those two. The file solvers.output has the results when all is copasetic.

- Test 10: This test exercises the netCDF navigation interface on a file with multiple channels, i.e., a DVLA file.
- Test 11: This test exercises the map engine to insure that it can arm.
- Test 12: This folder contains a test of the dedopplerizing filter.

The data input to the filter is generated via the program “dopplerize”. This data simulator was run as:

```
dopplerize dopplerize.inp
```

In this mode, it instantiates fake navigation transmitter and receiver objects, and uses inverse interpolation to compute a “dopplerized” sinusoid. The parameters of the sinusoid are supplied via the file “dopplerize.inp”: for the results presented here, the sinusoid had a carrier of 75 Hz. The dopplerized signal is “sampled” at 75 Hz (also in the input file) and written to the output file “dopplerized.mux”. (Output file name hardwired.) The file `dopplerized.mux.verified` is a version of this file when I thought things were working OK.

The `dopplerize.inp` contains some additional geometric parameters of the simulation, specifying a range of 50km.

The program `dedopplerize` is hardwired to read in the file `dopplerized.mux` and output a file `dedopplerized.mux`. For this simulation, the filter order was 21 (also hardwired.) This program also reads in `dedopplerize.inp` which is an exact copy of `dopplerize.inp`. The file `dedopplerized.mux.verified` is a copy of this output file when I thought everything was working OK.

The simulation covers 50 seconds of data, enough for exactly one revolution of the transmitter.

Contents of both .mux files were then dumped to auxiliary files via `testmuxread` (this is a utility in `test4`) and loaded into Matlab for frequency analysis. Here, I used the routine `pwelch`, called as follows on data `x`:

```
[Pxx,F] = pwelch(x,ones(3000,1),0,3000,300)
```

which estimates the PSD of x . In the ideal world, 3000 samples will contain exactly 10 periods of the sinusoid, so the spectrum of an undopplerized signal will be a single component at 75 Hz. Note that to produce this numerically, one needs a uniform window: tapered windows will introduce spectral leakage that will misrepresent the signal content.

The file `spectra-results.ps` contains plots of the PSDs (so calculated) for the data in `dopplerized.mux` and `dedopplerized.mux`. The data in the former shows broad skirts due to doppler spreading, and in the latter, the 75 Hz component is well recovered with something like 35 dB over the nearest sidelobe energy.

5 Examples

Two simple examples are provided.

5.1 Dopplerizing a Signal

Here is a code fragment that generates a dopplerized signal.

```

///// // make a MUX file for the output data:
LOAPEX::CMUXFile M(outputfilename, LOAPEX::CMUXFile::Write );
// input various sampling parameters:
M.put_sample_rate ( sample_rate );
M.put_total_scans ( total_scans );
M.put_total_channels ( total_channels );
M.put_start_second ( second_start );
std::string tmp("sinusoid dopplerized with fake transmitter and receiver motions");
M.put_description ( tmp );

// write all this stuff to the file:
M.write_header();

LOAPEX::CTimeBaseMapEngine* pMapEngine;
pMapEngine = new LOAPEX::CTimeBaseMapEngine( TXTIME_START, TXTIME_STOP );

// now we need the navigation interfaces:
LOAPEX::CBase3DNavigationData * pTX;
LOAPEX::CBase3DNavigationData * pRX;
LOAPEX::CNetCDFNavigationData *ptx, *prx;

// these are specific subclasses:
ptx = new LOAPEX::CNetCDFNavigationData(tx_motion_file_name,0);
prx = new LOAPEX::CNetCDFNavigationData(rx_motion_file_name,3);
// cast them back to their base class:
pTX = dynamic_cast<LOAPEX::CBase3DNavigationData*>(ptx);

```

```

pRX = dynamic_cast<LOAPEX::CBase3DNavigationData*>(prx);
// load them:
pMapEngine->load_TX( pTX );
pMapEngine->load_RX( pRX );

pMapEngine->load_parameters ( someParameters ); // we assume these have been assigned
pMapEngine->arm();

// In this application, we know the receiver turn on time because
// we are building a receiver file. Therefore we have to get the
// associated transmitter time of the receiver turn on time:
pMapEngine->get_inverse_time( rx_start_time, &tx_start_time );

std::vector<short> V(total_channels);
for (int ii = 0; ii< total_scans; ii++ )
{
    tprime.seconds_ = rx_start_time.seconds_ ;
    tprime.fraction_ = rx_start_time.fraction_ + ((double)ii)*delta;
    tprime.normalize();
    pMapEngine->get_inverse_time( tprime, &t );

    // compute s(t): End product of fragment is the quantity "word"
    oldtt = tt;
    tt = ((double)(t.seconds_-tx_start_time.seconds_))
        + (t.fraction_-tx_start_time.fraction_);
    double arg = 2.0* M_PI * carrier*tt;          /* omega t */
    // make a sinusoidal signal:
    short word = (short)(amplitude*sin( arg ));

    // store the data word in the output array:
    V[0] = word; // only one output channel, but put_scan still requires a vector

    //write the scan to disk file:
    M.put_scan ( V );
}

M.close();

```

5.2 Dedopplerizing Filter

See the code in Test 12 for an example of this application.

6 References

References

- [1] A. R. Gondeck, "Doppler time mapping", *J. Acoust. Soc. Am.*, **73**(5), pp 1863-1864, 1983.
- [2] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, *Numerical Recipes in C*, 2nd ed., Cambridge University Press, 1992.

- [3] J. Hay, "Improvement of the imaging of moving acoustic sources by the knowledge of their motion", *Proc IEEE Conf. Acoust. Sig. Proc.*, vol. 3, pp 1247-1252, 1981.
- [4] S. A. L. Glegg, "The de-dopplerization of acoustic signals using digital filters", *J. Sound Vib.* **116**(2), pp 384-387, 1987.