Architecture Review Goals, Principles and Examples

Gianluca Petrillo

University of Rochester/Fermilab

LArSoft Architecture and Testing Workshop, June 3rd , 2015

Outline

Strategy for architecture review

interoperability	thou shalt work with any detector
maintainability	thou shalt not drive mad people working with your code
factorization	thou shalt not depend on execution environment
common interfaces	thine algorithms shall share generic interfaces
architecture	thou shalt be careful of code design and structure

These strategic areas do overlap; each of them has as a side effect the creation of better code.

Today's goal

- suggest patterns of good practices in authoring compliant code
- appreciate some forms in which problems can manifest
- learn techniques to solve specific architecture issues
- improve a piece of existing LArSoft code

Cum granu salis

- the guidelines I will describe *most often* give better results than if ignored, and they do likely work for your case too;
- yet they are not absolute, and if after some necessary thinking they still don't fit well, they *might* need some adjustment.

You are encouraged to show your case to the LArSoft team!
We'll figure out together.

Buzz word: interoperability

Interoperability

Algorithms are generic enough that they can operate on any LAr TPC detector and readout configuration.

This rule is typically violated by the presence of assumptions, e.g. on

- structure of the detector
- geometry of the detector
- readout parameters

In the few cases where complete interoperability is not possible, the assumptions should be clearly stated in the documentation of the algorithm and possibly also in the lines of code where they are used.

Interoperability: detector assumptions

Algorithms sometimes assume the presence of a single TPC (failing for DUNE), or of exactly three planes (failing for LArIAT).

Guidelines to detection:

- lack of a code loop over TPCs
- queries to Geometry without cryostat and TPC numbers
- data structures that forget the location of their elements
- data structures that mix elements from different TPCs

Guidelines to solution:

- wrap the single TPC code into a TPC loop (e.g., using Geometry iterators)
- use data structures (maps) indexed by TPC or plane ID
- make the code generic and geometry-independent

Detector assumptions example

`Track3DKalmanHit` module collects *all* hits in a single collection:

```
art::Handle< std::vector<recob::Hit> > hith;
evt.getByLabel(fHitModuleLabel, hith);
if(hith.isValid()) {
  int nhits = hith->size();
  for(int i = 0; i < nhits; ++i)
    hits.push_back(art::Ptr<recob::Hit>(hith, i));
}
```

Excepts from `Track3DKalmanHit`: collection of hits

Therefore, algorithms *should have been* designed to sort hits out.

`FuzzyCluster` also collects *all* hits in a single collection:

```
// make a map of the geo::PlaneID to vectors of art::Ptr<recob::Hit>
std::map<geo::PlaneID, std::vector<art::Ptr<recob::Hit>>> planeIDToHits;
for(size_t i = 0; i < hitcol->size(); ++i)
  planeIDToHits[hitcol->at(i).WireID().planeID()]
    .push_back(art::Ptr<recob::Hit>(hitcol, i));
for(auto & itr : planeIDToHits) /* ... */
```

Excepts from `FuzzyCluster`: collection of hits

but they are grouped by plane, and one of them is processed at a time.

Interoperability: geometry assumptions

Code internals may take assumptions on:

- coordinate system (e.g. $y = 0$ is at the middle of TPC)
- relative position of TPCs (e.g. no other TPC on top and bottom)

Detection of these ones typically takes thorough reading and deep understanding of the source code. Still, hints can come from:

- hard-coded numbers
- geometrical calculation “from first principles” with no specific input from `Geometry` service

Guidelines to solution:

- retrieve the information from Geometry service provider
- rework the geometric formula to be more generic

Geometry assumptions example

This code compares the geometric limits of the *TPC* (that's what `DetHalfHeight()` is about) with the intersection between two wires, to find if that intersection is within:

```
// Y,Z limits of the detector
double YHi = geom->DetHalfHeight(tpc, cstat);
double YLo = -YHi;
double ZLo = 0.;
double ZHi = geom->DetLength(tpc, cstat);

// ...
geom->IntersectionPoint(iWire, jWire, ipl, jpl, cstat, tpc, y, z);
if(y < YLo || y > YHi || z < ZLo || z > ZHi) continue;
```

Excerpt from `ClusterCrawlerAlg::VtxMatch()` in LArSoft 4.3.3

The point returned by `IntersectionPoint()` is in global coordinates. It is compared to the TPC boundaries above, that represent the TPC surface only in a specific local coordinate frame.

Interoperability: readout/DAQ assumptions

Assumptions can be taken also on:

- trigger time (t_0) being at the beginning of the readout window
- readout window duration and structure
- topology of TPC by channel numbers

Detection hints are similar to the ones for geometry assumptions:

- again, hard-coded numbers
- no query to `DetectorProperties`, `LArProperties` services
- no query to `TimeService` and no use of the `T0` data product
- use of channel queries in post-readout reconstruction code
- sorting by channel number in post-readout reconstruction code

Once the issues are identified, the resolution is typically easy, as long as the service needed can be communicated to the algorithms.

Readout assumptions example

Trigger time assumed to be 3200 TDC ticks within readout window:

```
// Kludge to remove out of time hits
if (hit->StartTick() > 6400 || hit->EndTick() < 3200) continue;
```

Excerpts from `larana/CosmicRemoval/CRHitRemovalByPCA_module.cc`

A solution: ask `util::TimeService` to convert hit times in microseconds, and compare with the trigger time.

Here hits are Channels assumed to be on contiguous wires:

```
std::vector<int> const& ChannelHits = OrgHits[ThisView][ThisChannel];
for(size_t iOrg = 0; iOrg!= ChannelHits.size(); ++iOrg) {
  if(fabs(ThisTime - HitsFlat.at(ChannelHits[iOrg])->PeakTime()) < eta)
    SpacePointsPerHit.at(ChannelHits[iOrg]).push_back(iSP);
}
```

Excerpts from `larreco/RecoAlg/SeedFinderAlgorithm.cxx`

`OrgHits` organizes hits by view and channel. In fact, it really meant plane and wire. A solution: map by wire ID instead.

Interoperability: multi-experiment development

The problem of conflicting names can arise when developing with multiple experiment repositories, as with:

- libraries and modules with the same name (e.g., `AnaTree`)
- FHiCL files with the same name (e.g., `analysistreemodule.fcl`)

We advocate the following guidelines:

- create libraries with unique names:
  options are given in the LArSoft wiki
  (LArSoftWiki|Developing With LArSoft|The rules and guidelines)
- plug your experiment in the FHiCL file name:
  e.g., `analysistreemodule_uboone.fcl`

⇒ LArSoft itself is not immune from the problem:
conflicts within LArSoft are immediately detected and removed

Buzz word: factorization

Factorization

The code decouples the core algorithms from the interface to specific execution environments, libraries and frameworks.

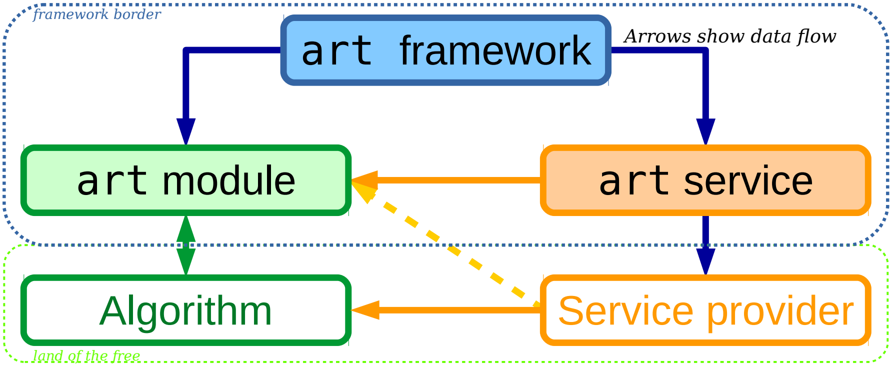
This makes the algorithms easier to develop, test and be exchanged between different frameworks.

Signs that this paradigm might be broken include:

- framework modules that also implement the algorithms
- algorithm classes using framework-specific constructs

Factorization: algorithms, modules and services

We think to factorization model as:



- the algorithm job is contained in the bottom, portable part
- the service/module layer interfaces the workers to the framework

The top part can be replaced, for example, by just as little facility as needed for the task, as in the unit test context.

Factorization: algorithm model

One algorithm should be implemented in one class.

It should follow this life style:

1. constructed by the module with some configuration
2. on each event:
    1. event-specific configuration and setup, if any
    2. receive the service providers from the module
    3. receive the input data from the module
    4. perform the task
    5. make output available as "standard" data structures

```
/// Extensive documentation goes here!
class MyAlgorithmClass {
    public:
  /// (1) Construct and configure
  MyAlgorithmClass
    (fhicl::ParameterSet const& pset);

  /// (2.1, 2.2) Initialize for a new event
  void Init(geo::GeometryCore const* geom);

  /// (2.3) Receive data
  void SetInput
    (std::vector<raw::RawDigit> const& digits);

  /// (2.4) Perform the task
  void Run();

  /// (2.5) Return the results
  std::unique_ptr<std::vector<recob::Oscillation>>
    GetResult();

  /// Destroy temporary data
  void Clear();

}; // class MyAlgorithmClass
```

Example of algorithm class structure

It *may* employ sub-algorithms, implemented as separate classes.

Algorithm model example

This is most of the public interface of `CCHitFinder`:

```
  class CCHitFinderAlg {
  public:
    // ...things that I omit because they should not have been public...

    CCHitFinderAlg(fhicl::ParameterSet const& pset);
    virtual ~CCHitFinderAlg() = default;

    virtual void reconfigure(fhicl::ParameterSet const& pset);

    void RunCCHitFinder(std::vector<recob::Wire> const& Wires);

    /// Returns (and loses) the collection of reconstructed hits
    std::vector<recob::Hit>&& YieldHits() { return std::move(allhits); }

    /// Print the fit statistics
    template <typename Stream> void PrintStats(Stream& out) const;

  private:
    // your signature hell here
};
```

`CCHitFinder` algorithm class (`larreco/RecoAlg/CCHitFinder.h`)

Algorithm model example: comments

This is most of the public interface of `CCHitFinder`: a good start.

- I omitted some public things that should not be public
- input phase is merged with run phase — nothing wrong with that
- no setup: no way to tell the algorithm which geometry to use; and it needs it: currently it's asking it to `art`

Factorization: module model

The module is the algorithm's personal assistant:

1. constructs and owns the algorithm
2. on each event:
    1. updates algorithm configuration
    2. obtains and delivers service providers to the algorithm
    3. transfers input data from the event to the algorithm
    4. ask the algorithm to do the job
    5. transfers results from the algorithm to the event

```
void MyModule::produce(art::Event& evt) {
  // (1) Construct and configure
  MyAlgorithmClass algo(pset);

  // (2.1, 2.2). Initialize for a new event
  algo.Init(&art::ServiceHandle<geo::Geometry>());

  // (2.3) Translate input data from framework to algorithm
  art::ValidHandle<std::vector<raw::RawDigit>> hRD
    = evt.getValidHandle<std::vector<raw::RawDigit>>
      (fRawDigits);
  algo.SetInput(*hRD);

  // (2.4) Perform the task
  algo.Run();

  // (2.5) Fetch results and donate them to the framework
  std::unique_ptr<std::vector<recob::Oscillation>> osc
    (new std::vector<recob::Oscillation>);
  *osc = algo.GetResult();
  evt.put(std::move(osc));

}; // MyModule::produce()
```

Example of factorized `art` module structure

Module model example

`LineCluster` module owns a pointer to `ClusterCrawlerAlg`:

```
fCCAlg.reset(new ClusterCrawlerAlg
  (pset.get< fhicl::ParameterSet >("ClusterCrawlerAlg")));
```

```
art::ValidHandle< std::vector<recob::Hit>> hitVecHandle
  = evt.getValidHandle<std::vector<recob::Hit>>(fHitFinderLabel);

fCCAlg->RunCrawler(*hitVecHandle); // look for clusters in all planes

// access to the algorithm results
ClusterCrawlerAlg::HitInCluster_t const& HitInCluster
  = fCCAlg->GetHitInCluster();
std::unique_ptr<std::vector<recob::Hit>> FinalHits
  (new std::vector<recob::Hit>(std::move(fCCAlg->YieldHits())));
std::vector<ClusterCrawlerAlg::ClusterStore> const& Clusters
  = fCCAlg->GetClusters();

// convert cluster and vertices into recob objects, create associations
// ... and put everything in the event
std::unique_ptr<std::vector<recob::Cluster> > ccol
  (new std::vector<recob::Cluster>(std::move(sccol)));
evt.put(std::move(ccol)); // ... et cetera...
```

Excerpts from `larreco/ClusterFinder/LineCluster_module.cc`

Factorization: service model

Services are also split between:

⇒ framework-independent *provider*

- answers service requests
- assumes it is up-to-date and synchronized with the world

⇒ *framework interface*

- creates, configures and owns the service provider instance
- updates the provider when needed *(e.g., on new event)*
- forwards requests to the provider

Algorithms should be given a pointer to the provider (“`get()`”).

```
class OurServiceProvider {
    public:
  /// Configure, register with the framework
  OurServiceProvider
    (fhicl::ParameterSet const& pset);

  /// Update the information
  void Update(TimeStamp_t event_time);

  /// ... and the actual service
  It_t GetIt() const;
}; // class OurServiceProvider
```

Example of service provider

```
class OurService {
  std::unique_ptr<OurServiceProvider> provider;
    public:
  /// Configure, register with the framework
  OurService(
    fhicl::ParameterSet const& pset,
    art::ActivityRegistry& reg
    );

  /// Return a pointer to the provider
  OurServiceProvider const* get() const;

  /// To be executed by art on each new event;
  /// may call OurServiceProvider::Update()
  void preProcessEvent(const art::Event& evt);
}; // class OurService
```

Example of factorized `art` service

Service model example: the service provider

```
class SimpleTimeService {
    public:

  /// TPC readout start time offset from trigger
  virtual double TriggerOffsetTPC() const { return fTriggerOffsetTPC; }

  /// Trigger electronics clock time in [us]
  double TriggerTime() const { return fTriggerTime; }

  /// Beam gate electronics clock time in [us]
  double BeamGateTime() const { return fBeamGateTime; }

  // ... and more service calls

    protected:
  // configuration is not exposed to the public; e.g.:
  virtual void SetTriggerTime(double trig_time, double beam_time);
};
```

Exerpts from `lardata/Utilities/SimpleTimeService.h`

- *this implementation* requires the framework to derive a class from the provider
- it is in general better practice for the service to contain the provider
- inheritance is only necessary when factorizing an existing service not to break existing code

Service model example: interface to `art`

```
class TimeService : public SimpleTimeService {
    public:
  TimeService(fhicl::ParameterSet const& pset, art::ActivityRegistry& reg);

  /// Override of base class function ... implement DB status check
  virtual double TriggerOffsetTPC() const override;

  //*** All following functions are not for users to execute ***//

  /// Re-configure the service module
  void reconfigure(fhicl::ParameterSet const& pset);

  /// Function to be executed @ run boundary
  void preBeginRun(art::Run const& run);

  /// Function to be executed @ event boundary
  void preProcessEvent(const art::Event& evt);

  /// Function to be executed @ file open
  void postOpenFile(const std::string& filename);

  // ... and the private stuff
}; // TimeService
```

Exerpts from `lardata/Utilities/TimeService.h`

The additional functions are connected to the framework. For example, `TimeService::preProcessEvent()` reads trigger information from the event and updates the provider by calling `SimpleTimeService::SetTriggerTime()`.

Factorization: `art` and library dependences

- algorithms should depend only on *widely used libraries*
- ... but they should not implement common things from scratch

We tend to recommending the following compromise:

+ `LArSoft` data structures: `recob`, `anab`, ...
+ nutools data structures: `simb`
+ ROOT, CLHEP, Boost (if you really have to...)
+ FNAL's message facility, FHiCL

± FNAL CET libraries (`cet::exception` is "unavoidable")

± `art::Assns`, `art::Ptr`

− `art::Event`, `art::FindOneP()` ...

− `art::Handle`, `art::ServiceHandle`... `art::*`

The ongoing work by the LArSoft/LArLite interoperability task force will define some of the orange areas of what should or should not be used.

Factorization: unit testing

- tests should be self-standing...
- ...but they need to fill in for a framework

The service provider developer can provide helpers for easy setup of:

- message facility service
- parsing of a FHiCL file
- set up of user services

Boost unit testing

It is possible, although a bit more complicate, to write “Boost fixtures” providing a service-aware environment for Boost unit tests.

```
class MyServiceTestEnvironment {
    public:
  /// Sets everything up at once (ideally)
  MyServiceTestEnvironment();

  /// Access to FHiCL configuration
  fhicl::ParameterSet const& Parameters() const;

  /// Access to service providers
  MyServiceProvider const* MyService() const
    { return myService.get(); }

    private:
  std::unique_ptr<MyServiceProvider> myService;
}; // MyServiceTestEnvironment
```

Example of test environment class

```
// create test environment (may need arguments)
MyServiceTestEnvironment TestEnv;

// create a test with FHiCL configuration
MyAlgoTestClass TestAlgo(TestEnv.Parameters());

// set up the test
TestAlgo.Setup(TestEnv.MyService());

// run the test
TestAlgo.Run();
```

Example of test in the environment

Buzz word: generic interfaces

Generic interfaces

Algorithms that produce the same data structures (in different ways) should share the interface.

General guidelines:

- think abstract
- use Occam's razor to prune redundant inputs and services
- use "standard" classes for input and output
- if your algorithm needs a larger interface because it does something more than the others in its category... maybe you are effectively dealing with two algorithms there
- don't overdo it: metaprogramming is both cool and hard to read

Generic interfaces: example

This is a mock-up of a hit finder interface:

```
class HitFinderBaseAlg {
    public:
  /// Virtual destructor: we need one
  virtual ~HitFinderBaseAlg() = default;

  /// Reads configuration from a parameter set
  virtual void Configure(fhicl::ParameterSet const&) {}

  /// Acquires pointers to non-owned resources
  virtual void Setup
    (geo::GeometryCore const*, util::DetPropertiesCore const*, util::LArPropertiesCore const*);

  /// Acquires the input; wires must exist until results are claimed
  virtual void SetWires(std::vector<recob::Wire> const& wires);

  /// Performs the actual hit finding
  virtual void Run() = 0;

  /// Returns results; can be called only once
  virtual std::vector<recob::Hit> YieldHits() = 0;

  /// Frees the owned resources
  virtual Clear() {}
}; // HitFinderBaseAlg
```

Example of abstract interface for hit finder algorithms

Some of these functions might have a standard no-op implementation.

Buzz word: maintainability

Maintainability

The code should help people who try to use, fix, or extend it.

General guidelines:

- good design: think ahead (e.g., write documentation first)
- modularity:
  - avoid functions that perform more than one task
  - sublet sub-tasks to other functions
  - don't worry about function call overhead until your profiler says to
- documentation: there is always somebody missing it
- tests are critical when extending the algorithms

Work session organization

1. pick your team (I recommend to work in pairs or singly)
2. pick your module: something you are interested in
3. create a working area and a feature branch, for example:
   `git flow feature start yournametag_ReviewModuleName`
4. evaluate it
   - glance at the code, get its structure

     take a pastry
   - dig into it, apply patterns, assess the problems

     (maybe another pastry)
5. design solutions
   - write it down!
   - if time allows, implement or start it

Work material

Now we want you to pick some code and try to improve it!
Here are some recommendations for modules of interest:

1. `RawHitFinder` *(hit finder from `RawDigit`)*
2. `fuzzyCluster` *(clustering based on Hough transform)*
3. `Track3DKalmanHit` *(tracking based on 3D Kalman filter)*
4. `ShowerReco3D`
5. `Calorimetry`
6. `AnalysisTree` *(reconstructed information into a ROOT tree)*

Pick the one you like, among them or others.
This is not just an exercise:

- you can do real work to improve them
- I don't necessarily know the "solution" (not even the problem!) for each of them

Backup

Work material: want more?

Here is some wider recommendation for modules of interest:

1. `GausHitFinder`
2. `CosmicPFParticleTagger`
3. `RawHitFinder` *(hit finder from `RawDigit`)*
4. `DisambigCheater`
5. `fuzzyCluster` *(clustering based on Hough transform)*
6. `CosmicTracker`
7. `Track3DKalmanHit` *(tracking based on 3D Kalman filter)*
8. `TrackStitcher`
9. `ShowerReco3D`
10. `Calorimetry`
11. `AnalysisTree` *(reconstructed information into a ROOT tree)*

Non-TPC-aware code

Assumes all TPCs are born equal:

```
// Calculate wire coordinate systems
for(size_t n=0; n!=3; ++n)
  {
    geom->WireEndPoints(0,0,n,0,xyzStart1,xyzEnd1);
    geom->WireEndPoints(0,0,n,1,xyzStart2,xyzEnd2);
    fWireDir[n] = TVector3(xyzEnd1[0] - xyzStart1[0],
      xyzEnd1[1] - xyzStart1[1],
      xyzEnd1[2] - xyzStart1[2]).Unit();
    fPitchDir[n] = fWireDir[n].Cross(fXDir).Unit();
    if(fPitchDir[n].Dot(TVector3(xyzEnd2[0] - xyzEnd1[0],
      xyzEnd2[1] - xyzEnd1[1],
      xyzEnd2[2] - xyzEnd1[2]))<0) fPitchDir[n] = -fPitchDir[n];
    fWireZeroOffset[n] =
      xyzEnd1[0]*fPitchDir[n][0] +
      xyzEnd1[1]*fPitchDir[n][1] +
      xyzEnd1[2]*fPitchDir[n][2];
  }
```

SeedFinderAlgorithm::CalculateGeometricalElements()

Simple Geometry-aware unit test (I)

This is a simple test using Boost unit test library.

```
cet_test(geometry_thirdplaneslope_test
  SOURCES geometry_thirdplaneslope_test.cxx
  LIBRARIES Geometry
            ${MF_MESSAGELOGGER}
            ${MF_UTILITIES}
            ${FHICLCPP}
            ${CETLIB}
  USE_BOOST_UNIT
  DATAFILES test_geometry.fcl
  TEST_ARGS test_geometry.fcl
)
```

Linking: `CMakeLists.txt`

```
#define BOOST_TEST_MODULE GeometryThirdPlaneSlopeTest
#include <boost/test/included/unit_test.hpp>

// LArSoft libraries
#include "test/Geometry/geometry_boost_unit_test_base.h"
#include "SimpleTypesAndConstants/PhysicalConstants.h" // util::pi()
#include "Geometry/GeometryCore.h"
#include "Geometry/ChannelMapStandardAlg.h"

// utility libraries
#include "messagefacility/MessageLogger/MessageLogger.h"

// C/C++ standard libraries
#include <string>
```

Part I: `geometry_thirdplaneslope_test.cxx`*: include needed headers*

Simple Geometry-aware unit test (II)

```
using StandardGeometryConfiguration:
  = testing::BoostCommandLineConfiguration<
    testing::BasicGeometryEnvironmentConfiguration<geo::ChannelMapStandardAlg>
    >;
using SimpleGeometryTestFixture
  = testing::GeometryTesterEnvironment<StandardGeometryConfiguration>;

BOOST_FIXTURE_TEST_SUITE(GeometryIterators, SimpleGeometryTestFixture)
```

Part II: environment setup

```
struct StandardGeometryConfiguration:
  public testing::BoostCommandLineConfiguration<
    testing::BasicGeometryEnvironmentConfiguration<geo::ChannelMapStandardAlg>
    >
{
  StandardGeometryConfiguration()
    { SetApplicationName("GeometryThirdPlaneSlopeTest"); }
}; // class StandardGeometryConfiguration

using SimpleGeometryTestFixture
  = testing::GeometryTesterEnvironment<StandardGeometryConfiguration>;

BOOST_FIXTURE_TEST_SUITE(GeometryIterators, SimpleGeometryTestFixture)
```

Part II, with customized test name

Simple Geometry-aware unit test (III)

```
BOOST_AUTO_TEST_CASE( AllTests )
{
  geo::GeometryCore const& geom = *Geometry();

  const double angle_u = 1. / 3. * util::pi<double>();
  const double angle_v = 2. / 3. * util::pi<double>();
  const double angle_w = 1. / 2. * util::pi<double>();

  BOOST_MESSAGE(
    "Wire angles: u=" << angle_u << " v=" << angle_v << " => w=" << angle_w
    );

  const double slope_u = 1. / std::sqrt(3);
  const double slope_v = 1. / std::sqrt(3);

  const double expected_slope_w = 0.5;

  double slope_w = geom.ComputeThirdPlaneSlope
    (angle_u, slope_u, angle_v, slope_v, angle_w);

  BOOST_MESSAGE(
    "Slopes: s(u)=" << slope_u << " s(v)=" << slope_v << " => s(w)=" << slope_w
    );

  BOOST_CHECK_CLOSE(slope_w, expected_slope_w, 0.01); // tolerance: 0.01%

} // BOOST_AUTO_TEST_CASE( AllTests )

BOOST_AUTO_TEST_SUITE_END()
```

Part III: test code