

DlangScience

Design Document pre-alpha

John Loughran Colvin

October 13, 2015

DlangScience is an attempt to make a productive, widely applicable scientific programming framework for the D programming language.

General overview

There are 3 main parts to the project:

1. Bindings and wrappers for and ports of 3rd party libraries from different languages, making life easy for users familiar with other software and providing a quick path to lots of features.
2. SciD, the core package/namespace of DlangScience, containing the building blocks of mathematical calculation on which most scientific programming relies.
3. Special purpose packages and modules. These would be libraries for e.g. plotting, statistics, signals analysis etc.

Project layout

Packages:

scid

The scid package/namespace will contain the core elements of the DlangScience project, as can be seen in Fig. 1 it has a similar scope to numpy

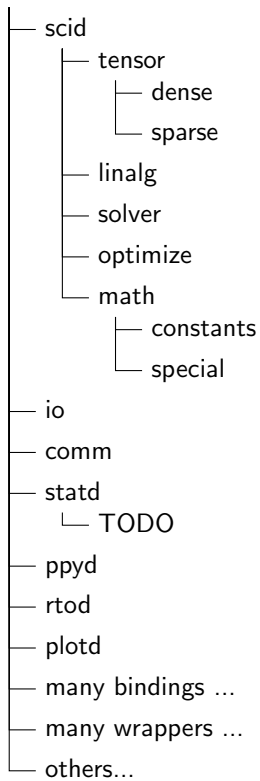


Figure 1: package and module layout

+ some of the less specialist parts of scipy.

io

A generic, unified interface for scientific file-formats. HDF-5 is the important starting point.

comm

A pleasant abstraction over various message-passing interfaces. There might be a certain amount of API overlap between `comm` and `io`, seeing as both deal with serialisation of some sort..

statd

Based on `dstats`, with components from elsewhere (e.g. `atmosphere`), this package will provide a variety of statistical tests, distributions, sampling algorithms etc.

ppyd

A pretty wrapper around `pyd`. This is a work in progress that can currently be found in my Github account

rtod

Lance Bachmeier’s library for $R \leftrightarrow D$ interoperability.

plotd

A hypothetical plotting solution. I don’t think there are any D-based tools out there that are ready, at this point. Interfacing to Lua (via `LuaD`, <https://github.com/JakobOvrum/LuaD>), R, Python or some external tool can provide us with “good enough for now” tools.¹

¹as a matter of fact, ease of use of these plotting libraries is a good test of any of our language interfaces.

Bindings

There is so much fantastic scientific code out there in other languages. While some things are best done natively in D, we should take full advantage of existing work.

Wrappers

C and Fortran APIs are - by and large - pretty awful. D can really help the good underlying work shine with a nice API without sacrificing performance. Additionally, wrappers should enable “clean” use with the various data types and APIs in both DlangScience and phobos. Wrappers should be dependant on and separate from bindings.

Ports

Ports cant take quite a lot of work (and testing!), but in order to help minimise dependencies and take advantage of D features, they can definitely be worth it. `scid` already has some in `scid.ports`.

Others

A catch-all for everything else. Domain specific libraries, interfaces to other languages...

1 Building and packaging

`dub` is the standard. It will soon be packaged with `dmd` releases, so almost everyone will have it. I think it is acceptable to have as the standard for building DlangScience libraries and their dependencies but it shouldn't be a requirement for users to build *their* projects. We should offer dependencies-included standalone source downloads that don't need an internet connection to build, using `dub add-path` to make sure `dub` doesn't have to fetch anything from the internet. Also, we should have some examples that use Makefiles in order to keep us from inadvertently overcomplicating builds for people who don't use `dub`.

1.1 System installations

This is a difficult topic. There are two important use-cases:

a) people who don't want to use `dub` for development

b) end users of software that internally uses DlangScience

a) can be dealt with by setting some import paths and library paths, so it's just a minor inconvenience for people if they don't have system installation. It is b) that is likely to be most important, as some/most linux distros insist on shared libraries and insist that those shared libraries are actually shared system-wide.

1.2 git modules, dub packages, submodules and sub-packages

- One git repository for each of the top-level packages listed in Fig 1, with the exception of the catch-alls at the end, where each individual instance should have its own repo.
- If a module is separately useful, it deserves its own dub sub-package. However, modules that mutually depend on each other (directly or indirectly), no longer fit the "separately useful" criteria.
- A reasonably fine-grained hierarchy of packages and sub-packages will help minimise interdependencies and therefore keep individual parts of DlangScience usable without involving everything else.
- The whole of DlangScience (minus any very young packages in alpha/beta) should be released together and be internally consistent on release. Master branches should be internally consistent and pass tests at all times.

Splitting everything into separate dub packages (or sub-packages) is really important. It allows people to select finer-grained dependencies, but also grab the whole lot at once for quick work (e.g. have a meta-repo/package that includes many other packages). Bindings, wrappers and ports should be separate dub packages in separate repos². Non-dub dependencies should be easy to work-around, i.e. large tracts of non-dependant code shouldn't be "polluted" by being packaged with small amounts of dependant code. Separate packages is better than configuration options, but worst case we can use dub configurations to exclude dependant code.

²at least if they are reasonably complete. No need to create a separate package for one little convenience function

In extremis we could even have dub packages that are just wrappers for makefiles, even including auto-downloads if necessary. This should only ever be for obscure dependencies that are unlikely to be in people's package managers.

1.3 Why/when to port?

Dependencies are a pain, especially ones in different languages. Language barriers make API design harder and prevent optimisations. So, should we always port to D? Where it's sensible. Libraries that are highly optimised and under constant improvement and tuning are a continuous resource that it's easy to become cut off from if you are maintaining a port. Sometimes a binding and a nice wrapper could provide a very similar user experience, while getting modern improvements for free. That said, there's a *lot* of scientific code out there that hasn't seen any changes made in decades, but is still very useful. A port to D could breathe new life in to these old masterpieces (or crusty old wrecks, depending on what you're looking at). In particular, once something is in D it can begin internal changes that allow better APIs that would not be possible with a wrapper or binding., e.g. adding support for arbitrary ranges instead of just arrays. As a rule of thumb, bind first, then consider whether a port is necessary.

2 Extended type functionality

We want to be able to perform operations on arbitrary types, but sometimes it isn't as easy as just overloading some operators and then templating things. E.g. `sin` and other transcendental/special functions require specialised implementations per type. The question is how a generic function finds out when there is a specialised implementation available for use, bearing in mind D's restrictive anti-hijacking rules and not wanting to reimplement all of D's overloading rules. There are a few different options I can think of:

1. Custom types have the relevant functions defined in their own modules/parent, such that any generic function elsewhere can simply import the parent module of a given module and preferentially use those. This has a weakness: for nested types, where is the function defined? It pollutes the namespace of the parent.

2. Custom types have UDAs that are aliases to the relevant functions, so a generic function selects the right function by checking the UDAs. A possible downside is that it requires a little boilerplate: you have to write out the UDA.
3. functions are defined as static methods in custom types, such that they can always be accessed directly from the type. This doesn't work with UFCS.

Here's an example of 2 in action. See the accompanying `extops.d` for the implementation of `ExtOp` and `Overloads`.

```
//customtype.d
import extops;
@(ExtOp!sin, ExtOp!cos) struct MyCustomType
{
    //data members, operator overloads etc...
}
MyCustomType sin(MyCustomType m) { /* ... */ }
MyCustomType cos(MyCustomType m) { /* ... */ }

//complexops.d
import extops;
import std.math;
import scid.math : complex;
    //std.complex is float/double/real only

auto expi(T)(T x)
{
    mixin Overloads!T;
    return complex(cos(x), sin(x));
}
```

`scid.math` should include templated versions of common mathematics operators like so:

```
// scid/math/package.d
import extops;
import std.math;

auto sin(T)(T x)
```

```
{
    mixin Overloads!T;
    return sin(x);
}
```

That is then the module for people to import instead of `std.math`, making the default case very simple:

```
//complexops_simple.d
import scid.math;

auto expi(T)(T x)
{
    return complex(cos(x), sin(x));
}

unittest
{
    static import customtype;
    static import std.math;
    customtype.MyCustomType x = // initialise
    assert(expi(x) ==
        complex(customtype.cos(x), customtype.sin(x)));
    assert(expi(3.4f) ==
        complex(std.math.cos(3.4f), std.math.sin(3.4f)));
}
```

3 General philosophy

A list of points in no particular order:

1. Abstractions are great, but they should be as transparent, assumption-free and state-free as possible. For example, the user of `scid.io` should be able to freely call the HDF-5 C API and modify some part of a file without it invalidating the nice D wrappers they're using elsewhere.
2. D's type system is the key to getting good APIs in D. You have to be able to statically reflect. The first goal of any wrapper project

should be to provide type-safety and introduce more explicit typing. The next layer of a wrapper then has *so* much more to work with.

3. Design-by-introspection and functions are preferable to endless named concrete types with functions, which in turn is better than big objects. We're trying to provide a set of tools for *other* people to model reality with, not a bunch of pre-set models. Ranges are the inspiration here, not `InputRangeObject`.
4. `myData.assumeSorted.quantiles` is so much better than adding a flag to `quantiles`.
5. Lazy is composable and can often greatly alleviate copying and allocations. As much as possible, provide a lazy option. If an eager version is significantly faster (e.g. calling some specialised outside routine), provide that too.
6. Compatibility with phobos: as much as possible, Users shouldn't feel like they are working in an island framework, interoperability should be trivial. In some cases this might require/lead to changes to phobos, e.g. Ilya's pull request for `ndSlice`.

4 scid

4.1 tensor

This is one of the most important parts of the whole project. Everything here should be a wrap/extension of the interface of Ilya Yaroshenko's `ndSlice`.

4.2 linalg

Generic linear algebra functions, with specialisations that call best-of-breed C/Fortran/C++ libraries

4.3 non-linear

A selection of general-purpose non-linear solvers and optimisers

4.4 math

The home of the basic import for mathematical functions with specialised overloads

4.4.1 special

Implementations of the common special functions, as generic as possible. A certain amount of this is already done in `scid.functions` and `libcerf`. For others it might be worth linking to outside code here, at least temporarily (unless anyone really feels like implementing all those Hankel functions, Airy functions, generalised Fs etc...).

5 statd

Distributions, sampling etc.