Primer on the OMS code

Robert Harron

Last updated: January 23, 2013

Contents

1 Introduction

The goal of the OMS code is to implement the overconvergent modular symbols of Pollack–Stevens. So, here's a (very) brief description of what that involves. An overconvergent modular symbol is just a certain type of modular symbol[1] and thus is just a certain type of map from $\mathrm{Div}^0 \mathbf{P}^1(\mathbf{Q})$ to some Hecke module. The type of Hecke module in question is what makes the modular symbols of interest "overconvergent". Specifically, these are modules of p-adic distributions. This already gives us an idea of what needs to be implemented:

- A class for modular symbols which carries the data of the map. In view of the Sage coercion model, we'll also want a Sage parent[2] for spaces of modular symbols. A modular symbol is determined by its value on finitely many elements of $\mathrm{Div}^0 \mathbf{P}^1(\mathbf{Q})$ and thus are stored as lists of p-adic distributions.

[1] Unfortunately, "modular symbol" is an overloaded term. Here, as we'll see, we mean the type of thing returned by EllipticCurve.modular_symbol(), not what ModularSymbol creates.

[2] I'll use the term "Sage parent" to refer to the notion of parent in the sense of Sage's coercion model. This is *not* the notion of parent in the sense of class inheritance; rather it is generally some class representing a space of elements.

- A class for p-adic distributions and, as above, a Sage parent for spaces of p-adic distributions. The distributions are approximated by their moments, i.e. their values on the monomials x^n for n running from 0 to some fixed bound. As such, they are represented by lists.

The following sections go more into detail (though not much more!) on the current structure of the code.

Note that the overconvergent modular symbols can be used to compute p-adic L-functions of modular forms and that that code is part of the OMS code, but won't be discussed in this primer (at least not yet).

There's also more recent work of Rob Pollack, Evan Dummit, Marci Hablicsek, Lalit Jain, Daniel Ross, and myself (Rob Harron) on families of overconvergent modular symbols. These can be thought of as modular symbols whose target Hecke module is a space of families of p-adic distributions and so the current OMS code should be able to accommodate these with minor modifications. Some changes are suggested below.

2 Current code structure

2.1 Overconvergent modular symbols

This section describes the implementation of the overconvergent modular symbols themselves. In the file modsym.py, there is an abstract class **PSModularSymbolElement** (which inherits from ModuleElement) that serves as the abstract implementation of overconvergent modular symbols (the PS in front stands for Pollack–Stevens). The main data stored is the attribute ._map, which is a **ManinMap** object. The class ManinMap is implemented in the file manin_map.py and it is important to note that most of what happens with the modular symbols is going on in the ManinMap class. Since PSModularSymbolElement functions as an abstract class, one must create derived classes and the idea is to create one for each type of Hecke module one wants to use. Thus, the file modsym.py contains two derived classes **PSModularSymbolElement_symk** and **PSModularSymbolElement_dist**. The former is for the Hecke module of homogeneous polynomials of degree k (i.e. this gives a new implementation of "classical" modular symbols) while the latter is for overconvergent modular symbols.

2.1.1 The ManinMap class and fund_domain.py

The ManinMap has three main data: ._codomain, which is a space of distributions (really a **Distributions_abstract** object, see a subsequent section for a discussion of this class), ._dict, which contains the (finitely many) values that determine the map, ._manin, which is a **ManinRelations** object. And this is a great segue to talk about the ManinRelations class and what's going on in fund_domain.py. Basically, Pollack and Stevens figured out a way to find a minimal number of elements of $\mathrm{Div}^0\,\mathbf{P}^1(\mathbf{Q})$ on which one must know the value of a modular symbol in order to know all other values. This involves computing explicit fundamental domains for $\Gamma_0(N)$ and related stuff. In the end, all this data is stored in a ManinRelations object. In particular, this object contains the list of elements of $\mathrm{Div}^0\,\mathbf{P}^1(\mathbf{Q})$ on which one needs to know the value of the modular symbol, as well as the data involved in determining all other (infinitely many) values from this finite set of values. So, the ._manin attribute allows the ManinMap object to have meaning.

2.1.2 Suggestions

To implement families, one should create a third derived class **PSModularSymbolElement_fam**, say. Also, the current way to distinguish between the two types of modular symbols is the implementation of a function .is_symk() in each derived class. This needs to be changed to allow for more than two derived classes.

2.2 Spaces of overconvergent modular symbols

This section describes the implementation of spaces of overconvergent modular symbols. The class **PSModularSymbolSpace** in the file space.py implements the Sage parent of PSModularSymbolElement. There's a bit of a complication that those not so familiar with how Sage implements such things might find confusing (not much more confusing than this sentence). Here's the idea: if I create a space of modular symbols of some weight, level, and Hecke module and then you come along and create a space of the same weight, level, and Hecke module, then Sage should know not to actually create a new space, but rather it should simply return the space I already created. This is implemented through the Sage class UniqueFactory. What one does here is to create a class called **PSModularSymbolSpace_constructor** (also in space.py), which inherits from UniqueFactory, and manages a cache of already created modular symbol spaces. Then, one instantiates a PSModularSymbolSpace_constructor object called PSModularSymbols. So, if you want create a space of modular symbols, you don't call the class PSModularSymbolSpace, nor even PSModularSymbolSpace_constructor, rather you would call PSModularSymbols(whatever parameters). Note that PSModularSymbols is not a class, it is a specific instance of the class PSModularSymbolSpace_constructor.

2.3 Distributions

This section describes the implementation of p-adic distributions. Distributions are currently implemented in the cython file dist.pyx. There is an abstract class called **Dist**, which inherits from ModuleElement, as well as two derived classes **Dist_vector** and **Dist_long**. The latter is a C implementation that can be used in certain cases to speed up computations. The global function get_dist_classes in dist.pyx determines if Dist_long can/should be used.

2.3.1 WeightKAction

The Sage coercion model might again create some confusion here with the way the action of GL(2) is implemented. In the file dist.pyx, you will also find an abstract class **WeightKAction**, which inherits from the Sage class Action, as well as two derived classes **WeightKAction_vector** and **WeightKAction_long**. These classes help the Sage coercion model figure out what multiplying a distribution by something else might mean. The actual place where these classes are registered in the coercion mechanism is in the init function of Sage parent of Dist.

2.3.2 Suggestions

In order to implement families of p-adic distributions, one would need to create two new derived classes: **Dist_fam**, inheriting from Dist, and **WeightKAction_fam**, inheriting from WeightKAc-

tion.

2.4 Spaces of distributions

This section describes the implementation of spaces of p-adic distributions. In the file distributions.py, you'll find an abstract class **Distributions_abstract**, which inherits from module, as well as two derived classes **Symk_class** and **Distributions_class**. Each of the latter two has an associated constructor class inheriting from UniqueFactory: **Symk_factory** and **Distributions_factory**, respectively. The associated instances of these constructor classes are called Symk and Distributions, respectively.