Tutorial for p-adics in SAGE

David Roe

March 9, 2007

1 Introduction

p-adics in SAGE are currently undergoing a transformation. Previously, SAGE has included a single class representing $\mathbb{Q}_p$, and a single class representing elements of $\mathbb{Q}_p$. Our goal is to create a rich structure of different options that will reflect the mathematical structures of the p-adics. This is very much a work in progress: some of the classes that we eventually intend to include have not yet been written, and some of the functionality for classes in existence has not yet been implemented. In addition, while we strive for perfect code, bugs (both subtle and not-so-subtle) continue to evade our clutches. As a user, you serve an important role. By writing non-trivial code that uses the p-adics, you both give us insight into what features are actually used and also expose problems in the code for us to fix.

Our design philosophy has been to get a robust, usable interface working first, with simpleminded implementations underneath. We want this interface to stabilize rapidly, so that users' code does not have to change. Once we get the framework in place, we can go back and work on the algorithms and implementations underneath. All of the current p-adic code is currently written in pure Python, which means that it does not have the speed advantage of compiled code. Thus our p-adics can be painfully slow at times when you're doing real computations. However, finding and fixing bugs in Python code is *far* easier than finding and fixing errors in the compiled alternative within SAGE (SageX), and Python code is also faster and easier to write. We thus have significantly more functionality implemented and working than we would have if we had chosen to focus initially on speed. And at some point in the future, we will go back and improve the speed. Any code you have written on top of our p-adics will then get an immediate performance enhancement.

If you do find bugs, have feature requests or general comments, please let me know at roed@math.harvard.edu.

This tutorial attempts to outline what you need to know in order to use the p-adics effectively. OUTLINE SECTIONS.

2 Terminology and types of p-adics

To write down a p-adic element completely would require an infinite amount of data. Since computers do not have infinite storage space, we must instead store finite approximations to elements. Thus, just as in the case of floating point numbers for representing reals, we have to store an element to a finite precision level. The different ways of doing this account for the different types of p-adics.

We can think of p-adics in two ways. First, as a projective limit of finite groups:

$$\mathbb{Z}_p = \lim_{\leftarrow n} \mathbb{Z}/p^n\mathbb{Z}.$$

Secondly, as Cauchy sequences of rationals (or integers, in the case of $\mathbb{Z}_p$, under the p-adic metric. Since we only need to consider these sequences up to equivalence, this second way of thinking of the p-adics is the same as considering power series in p with integral coefficients in the range 0 to $p-1$. If we only allow nonnegative powers of p then these power series converge to elements of $\mathbb{Z}_p$, and if we allow bounded negative powers of p then we get $\mathbb{Q}_p$.

Both of these representations give a natural way of thinking about finite approximations to a p-adic element. In the first representation, we can just stop at some point in the projective limit, giving an element of $\mathbb{Z}/p^n\mathbb{Z}$. As $\mathbb{Z}_p/p^n\mathbb{Z}_p \cong \mathbb{Z}/p^n\mathbb{Z}$, this is is equivalent to specifying our element modulo $p^n\mathbb{Z}_p$.

Definition 2.1 *The* absolute precision *of a finite approximation $\bar{x} \in \mathbb{Z}/p^n\mathbb{Z}$ to $x \in \mathbb{Z}_p$ is the non-negative integer n.*

In the second representation, we can achieve the same thing by truncating a series

$$a_0 + a_1 p + a_2 p^2 + \cdots$$

at p^n, yielding

$$a_0 + a_1 p + \cdots + a_{n-1}p^{n-1} + O(p^n).$$

As above, we call this n the absolute precision of our element.

Given any $x \in \mathbb{Q}_p$ with $x \neq 0$, we can write $x = p^v u$ where $v \in \mathbb{Z}$ and $u \in Zpx$. We could thus also store an element of $\mathbb{Q}_p$ (or $\mathbb{Z}_p$) by storing v and a finite approximation of u. This motivates the following definition:

Definition 2.2 *The* relative precision *of an approximation to x is defined as the absolute precision of the approximation minus the valuation of x.*

For example, if $x = a_k p^k + a_{k+1}p^{k+1} + \cdots + a_{n-1}p^{n-1} + O(p^n)$ then the absolute precision of x is n, the valuation of x is k and the relative precision of x is $n-k$.

There are four different representations of $\mathbb{Z}_p$ in Sage and two representations of $\mathbb{Q}_p$: the fixed modulus ring, the capped absolute precision ring, the capped relative precision ring, the capped relative precision field, the lazy ring and the lazy field.

2.1 Fixed Modulus Ring

The first, and simplest, type of $\mathbb{Z}_p$ is basically a wrapper around $\mathbb{Z}/p^n\mathbb{Z}$, providing a unified interface with the rest of the p-adics. You specify a precision, and all elements are stored to that absolute precision. If you perform an operation that would normally lose precision, the element does not track that it no longer has full precision.

The fixed modulus ring provide the lowest level of convenience, but it is also the one that has the lowest computational overhead. Once we have ironed out some bugs, the fixed modulus elements will be those most optimized for speed.

As with all of the implementations of $\mathbb{Z}_p$, one creates a new ring using the constructor `Zp`, and passing in `'fixed-mod'` for the `type` parameter. For example,

```
sage: R = Zp(5, prec = 10, type = 'fixed-mod', print_mode = 'series')
sage: R
5-adic Ring of fixed modulus 5^10
```

One can create elements as follows:

```
sage: a = R(375)
sage: a
3*5^3 + O(5^10)
sage: b = R(105)
sage: b
5 + 4*5^2 + O(5^10)
```

Now that we have some elements, we can do arithmetic in the ring.

```
sage: a + b
5 + 4*5^2 + 3*5^3 + O(5^10)
sage: a * b
3*5^4 + 2*5^5 + 2*5^6 + O(5^10)
sage: a // 5
3*5^2 + O(5^10)
```

Since elements don't actually store their actual precision, one can only divide by units:

```
sage: a / 2
4*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 2*5^8 + 2*5^9 + O(5^10)
sage: a / b
...
<type 'exceptions.ValueError'>: cannot invert non-unit
```

If you want to divide by a non-unit, do it using the `//` operator:

```
sage: a // b
3*5^2 + 3*5^3 + 2*5^5 + 5^6 + 4*5^7 + 2*5^8 + O(5^10)
```

2.2 Capped Absolute Ring

The second type of implementation of $\mathbb{Z}_p$ is similar to the fixed modulus implementation, except that individual elements track their known precision. The absolute precision of each element is limited to be less than the precision cap of the ring, even if mathematically the precision of the element would be known to greater precision (see Appendix A for the reasons for the existence of a precision cap).

Once again, use `Zp` to create a capped absolute p-adic ring.

```
sage: R = Zp(5, prec = 10, type = 'capped-abs', print_mode = 'series')
sage: R
5-adic Ring with capped absolute precision 10
```

We can do similar things as in the fixed modulus case:

```
sage: a = R(375)
sage: a
3*5^3 + O(5^10)
sage: b = R(105)
sage: b
5 + 4*5^2 + O(5^10)
sage: a + b
5 + 4*5^2 + 3*5^3 + O(5^10)
sage: a * b
3*5^4 + 2*5^5 + 2*5^6 + O(5^10)
sage: c = a // 5
sage: c
3*5^2 + O(5^9)
```

Note that when we divided by 5, the precision of `c` dropped. This lower precision is now reflected in arithmetic.

```
sage: c + b
5 + 2*5^2 + 5^3 + O(5^9)
```

Division is allowed: the element that results is a capped relative field element, which is discussed in the next section:

```
sage: 1 / (c + b)
5^-1 + 3 + 2*5 + 5^2 + 4*5^3 + 4*5^4 + 3*5^6 + O(5^7)
```