The pAdic library

Abstract. The pAdic library is a standalone C++ 11 library for basic calculations with padic numbers.

We suppose that the reader is familiar with C++11[1] and also with the basics of the theory of padic numbers. The padic numbers are represented by the `pAdic` class in our library. The basic features can be used by including the pAdic library:

```
    #include “pAdic.h”
```

Some special functions are declared in the `pAdic_functions.h` header file.
Everything in the library is put into the `pAdic_lib` namespace, see below. In the examples we do not write out the namespace scope resolution, it is understood to be there.
Before going into details and see examples, we need to talk about precision, since the construction of the padic numbers already relies on this feature.

Some words on the precision

In our library every padic number is defined up to a given precision. This precision can be set and get globally via static functions of the `pAdic` class:

```
    class pAdic {
    public:
        static void set_global_precision(l_int newprec) { global_prec =
        newprec; }
        static l_int get_global_precision() { return global_prec; }
    ...}
```

`l_int` is defined as

```
    using l_int = long long int;
```

The default value of `global_prec` is 10:

```
    l_int pAdic::global_prec{ 10 };
```

This means that by default the padic numbers are stored up to O(p^9) precision. (This is the big “O” notation.) Yes, it is strange that 10 means O(p^9) accuracy. The digits are actually stored up to p^10, but at this power the precision in the last digit might be lost during calculations, so it is safe to print only the guaranteed precise digits.

Constructors

The padic numbers are represented by the `pAdic` class. There are a number of ways for constructing a padic number:

1. Initialization to zero, the only parameter is the prime:

```
    pAdic (l_int);
```

Example:

[1] Some part of the library depend on the new features of C++11, like the vector initialization (http://en.cppreference.com/w/cpp/language/list_initialization) and the syntax of initialization by braces (http://en.cppreference.com/w/cpp/language/value_initialization)

```
        pAdic a(7);
```

In this case the precision of a is set to be the global precision, and the value of a is simply 0. Note that the constructor *does not test* whether p is a prime or not. This task is delegated to the user.

2. Initialization by an integer:

```
        pAdic (l_int, l_int, l_int = global_prec);
```

Example:

```
        pAdic a(318,7);
```

a will represent the number 3+3*7+6*7^2 (=318).

3. Initialization by copy. The value of the number is copied from another, previously constructed padic number, together with the prime and precision:

```
        pAdic (const pAdic n&) = default;
```

Example:

```
        pAdic a(318,7);
        pAdic b(a);
```

4. Initialization by `{coefficient, prime power}` vector pairs:

```
        pAdic(const Vatom&, l_int, l_int = global_prec);
```

Here `Vatom` is defined by

```
        using Vatom = std::vector<p_atom>;
```

and `p_atom` is a `struct` as given below:

```
        struct p_atom {   l_int coeff{}, ppower{}; };
```

Example:

```
        pAdic a({ { 4,-7 },{6,-6},{ 5,-1 },{ 3,4 } }, 13);
```

The variable a then represents the p-adic integer 4*13^-7+6*13^-6+5*13^-1+3*13^4. Note that the `p_atom` elements in the list need not be ordered according to the prime powers, the definition

```
        pAdic a({{6,-6}, { 4,-7 }, { 3,4 }, { 5,-1 } }, 13);
```

is equally okay.

5. Initialization by string literal:

```
        pAdic(const std::string&, l_int, l_int = global_prec);
```

Example:

```
        pAdic a("1+2p+3p^2", 11);
```

a is the padic number 1+2*11+3*11^2. Note that the value of the prime cannot be used instead of the symbol p. Thus "1+2*11+3*11^2" is not a valid string to initialize with.

6. Initialization with a fraction (via the same constructor as in the previous point)

pAdic(const std::string&, l_int, l_int = global_prec);

Example:

```
        pAdic a("164523/59464", 17);
```

Operations

All the standard operations can be used as usual. For example, the code

```
        l_int p{ 17 };
        pAdic a(3520, p);
        pAdic b(1437, p);
        cout << "a= " << a << '\n';
        cout << "b= " << b << '\n';
        cout << "a+b= " << a+b << '\n';
        cout << "a-b= " << a - b << '\n';
        cout << "a*b= " << a * b << '\n';
        cout << "a/b= " << a / b << '\n';
```
results in the following output:
```
        a= 1+3*p+12*p^2
        b= 9+16*p+4*p^2
        a+b= 10+2*p+1*p^3
        a-b= 9+3*p+7*p^2+O(p^9)
        a*b= 9+9*p+9*p^2+9*p^3+9*p^4+3*p^5
        a/b= 2+8*p+12*p^2+12*p^4+4*p^5+3*p^6+7*p^7+8*p^8+O(p^9)
```

Not only two padic numbers can be used for the overloaded `+,-,*,/` operators, but also padic numbers with integers:
```
        l_int p = 7;
        pAdic a(178, p);
        cout << "a= " << a << '\n';
        cout << "a-3= " << a - 3 << '\n';
        cout << "a+3= " << a + 3 << '\n';
        cout << "2a= " << 2 * a << '\n';
        cout << "(-3)*a= " << -3 * a << '\n';
```
will print the following output:
```
        a= 3+4*p+3*p^2
        a-3= 4*p+3*p^2+O(p^9)
        a+3= 6+4*p+3*p^2
        2a= 6+1*p+1*p^3
        (-3)*a= 5+3*p^2+5*p^3+6*p^4+6*p^5+6*p^6+6*p^7+6*p^8+O(p^9)
```

The `+=, -=, *=, /=` operators work as usual. There are no `++, --` operators defined.

The `[i]` operator gives back the coefficient of p^i:
```
        pAdic a("164523/17185096", p);
        cout << "a= " << a << '\n';
        cout << "a[-1] =" << a[-1] << '\n';
```
prints
```
        a= 10*p^-2+11*p^-1+16+12*p+16*p^2+2*p^3+12*p^4+2*p^5+16*p^6
        +5*p^8+O(p^9)
        a[-1] =11
```

The operators == and `!`= work as we expect.

Print styles

A padic number can be printed out with the `std::cout` stream's `<<` operator. Examples were shown above. We can decide whether we print the symbol 'p' or we print the concrete value of p. The
```
        static print_style how_to_print;
```
variable of the pAdic class can be set to two values, taken from the below enum class:

```
	enum class print_style { print_prime, print_p };
```
Thus, we can write
```
	pAdic::how_to_print = pAdic_lib::print_style::print_prime;
```
or
```
	pAdic::how_to_print = pAdic_lib::print_style::print_p;
```
In the first case the value of the prime is printed, in the second case only the symbol 'p'. The following program
```
	l_int p = 7;
	pAdic a(178, p);
	pAdic::how_to_print = pAdic_lib::print_style::print_prime;
	cout << "a= " << a << '\n';
	pAdic::how_to_print = pAdic_lib::print_style::print_p;
	cout << "a= " << a << '\n';
```
has the below output:
```
	a= 3+4*7+3*7^2
	a= 3+4*p+3*p^2
```

Additional member functions in the pAdic class

Apart of the constructors and operators and print style settings we have covered so far, there are some functions in the `pAdic` class that help to use the padic numbers or give some information about them.
```
	l_int valuation() const;
```
This function gives the valuation of the actual pAdic number. Usage:
```
	pAdic a(5*7*7+3*7*7*7,7);
	a.valuation();//this will return 2.
```

```
	l_double norm() const;
```
`norm()` gives the norm of the variable, as a simple floating point number. This can be useful for simple comparisons. Otherwise the exact `valuation()` function should be used.

```
	l_int get_prec() const;
```
This gives the precision of the given variable.

```
	void set_prec(l_int = global_prec);
```
The precision can be set by the `set_prec()` function. The default is the global value `global_prec`. See the beginning of this documentation.

```
	l_int get_p() const;
```
`get_p()` gives back the prime attached to the actual padic number.

```
	Vatom vectorize() const;
```
The `vectorize()` function gives back a Vatom struct. See above what a `Vatom` is.

```
	bool is_zero() const;
	bool is_one() const;
	bool is_unit() const;
	bool is_1unit() const;
```
The above four functions return `true` if the actual padic function is 0, 1, unit (i.e., its valuation is zero), and 1-unit (unit and the coefficient of p^0 is 1).

Exceptions

The different functions during calculation may throw exceptions. These are defined via the following class:

```
class pAdicError {
public:
      enum Error_Types {
            Syntax_Error, NonMatching_primes, IterationError,
            Division_by_Zero, Exp_OutofDomain, Log_OutofDomain,
            Gamma_OutofDomain, Exp_AH_OutofDomain, Pow_OutofDomain
      };
      pAdicError(Error_Types i) : ErrorType{ i } {}
      Error_Types get_error() { return ErrorType; }
private:
      Error_Types ErrorType;
};
```

The `Syntax_Error` exception is thrown when the initializer string is invalid.
The `NonMatching_primes` exception is thrown if we try to perform binary operations on two variables with non-equal primes.
`IterationError` exception is thrown when the division operator reaches too much (20) iterations to find the reciprocal of a number.
`Division_by_Zero` is thrown by the / operator if we try to divide a padic zero.
The rest of the exceptions are thrown when the corresponding function receives un-acceptable argument. (For example, the p-adic exponential function is defined only when the valuation of a number is >0.)

The padic_functions library

There are some functions I wrote, like `exp()`, `log()`. But here the problem arises that without an arbitrary precision package the limitations are strong. The exponential function needs factorials, and here we rapidly get overflow with built in arithmetic. See the final remarks for more details.
All of the functions in the `pAdic_functions` library are provided without any guarantee! Any collaboration is welcome to make this part of the package better! Please browse the `pAdic functions.h` and `pAdic functions.cpp` files to get more information, and again, see the final remarks for more details.

To show how the functions in this library can be used, we provide some codes. For example, we can calculate powers of padic numbers:

```
l_int p{ 7 };
pAdic c("-3/5", p);//the 7-adic number -3/5
cout << "c=   " << c << '\n';
cout << "c^3= " << pAdic_lib::pow(c, 3) << '\n';
cout << "c^-3=" << pAdic_lib::pow(c, -3) << '\n';
```

This will have the following result (when the default precision, 10 is set):

```
c=   5+2*p+1*p^2+4*p^3+5*p^4+2*p^5+1*p^6+4*p^7+5*p^8+O(p^9)
c^3= 6+6*p+4*p^2+5*p^3+4*p^4+2*p^5+6*p^6+2*p^7+3*p^8+O(p^9)
c^-3=6+6*p+1*p^2+1*p^3+5*p^4+6*p^5+5*p^6+2*p^7+4*p^8+O(p^9)
```

The logarithm of a padic number can be determined with a code similar to this:

```
        l_int p{ 5 };
        pAdic b(1 + 2 * p, p);
        pAdic l = log(b);
        cout << "b=" << b << '\n';
        cout << "log(b)=" << l << '\n';
```

Here we defined b via the integer `1+2*p` (= 11). Then we defined a new padic variable `l`, initialized to be the padic number `log(b)`, then we print out b and its logarithm `l`. The result is

```
        b=1+2*p
        log(b)=2*p+3*p^2+4*p^5+3*p^6+3*p^7+1*p^8+O(p^9)
```

Another example shows how to raise a padic number to a padic power (a^b is defined only when a is a 1-unit, i.e., it is a unit and the coefficient of p^0 is 1):

```
        l_int p{ 5 };
        pAdic a("1+p+p^2", p);
        pAdic b("p+p^2", p);
        cout << "pow(" << a << ", " << b << ")=\n" << pow(a,b) << '\n';
```

This prints

```
        pow(1+1*p+1*p^2, 1*p+1*p^2)=
        1+1*p^2+4*p^3+4*p^4+2*p^6+4*p^7+2*p^8+O(p^9)
```

The `exp()`, `gamma()` (latter implements the Morita gamma function), `Volkenborn()` and the `contfract_Schneider()` (which calculates the Schneider continued fraction) are of very limited use, because of the possible overflow during calculations with built in precision arithmetic.
We provide a concrete example which shows how to use the `contfract_Schneider()` function and its companion functions to print the result and get back the represented number from a continued fraction. The below code gives correct result:

```
        pAdic a("1+3p+p^2", p);
        SCF_pairs v;
        v = contfract_Schneider(a);
        cout << "The Schneider continued fraction of " << a << " is\n";
        print_SCF_pairs(v);
        cout << "The number restored from the continued fraction is " <<
        SCF_to_pAdic(v, p) << '\n';
```

It prints

```
        The Schneider continued fraction of 1+3*p+1*p^2 is
        [1,6,2,5,15,16,16,16,16,16,16]
        [17,17,17,17,17,17,17,17,17,17,17]
        The number restored from the continued fraction is 1+3*p+1*p^2+O(p^9)
```

While the similar code with pAdic a("1+3p+p^2+p^5", p); prints, incorrectly

```
        The number restored from the continued fraction is
        1+3*p+1*p^2+1*p^5+9*p^6+10*p^7+12*p^8+O(p^9)
```

The `Volkenborn()` function is intended to calculate the Volkenborn integral of appropriate functions, but it is of so limited use with the standard precision arithmetic that it is almost useless. See the final remarks for more details.

The pAdic_lib namespace

All the code is put in the `pAdic_lib` namespace. For more on namespaces, see the source http://en.cppreference.com/w/cpp/language/namespace
If one, before start using the pAdic library's features, adds the line

```
using namespace pAdic_lib;
```

to the code, then does not need to care about the namespace, everything can be used without particular attention. Otherwise one must specify in any occasions to the compiler where to find the requested declaration. For example, without the `using namespace pAdic_lib;` line one cannot write

```
pAdic a(10,7);
```

Instead, one should specify that the pAdic class is in the `pAdic_lib` library, by using the scope resolution operator `::`. Something like this will do the job:

```
pAdic_lib::pAdic a(10,7);
```

The functions declared in the `pAdic_fuctions.h` header are put in the `pAdic_lib` namespace, too.

Files (.h and .cpp):

pAdic: the definition and declaration of the class pAdic.
pAdic_parser: routines for string to padic number conversion (see the construction of a padic number)
pAdic_utilities: routines for internal computations
pAdic_functions: the library for power, exponential, Artin-Hasse exponential, logarithm, Morita gamma, Schneider continued fraction and Volkenborn integrals

Final remarks

There are some ways how one could (help to) further develop this library:

1. Add more features.
The basic features of this library enable the user to do basic arithmetic on the padic numbers. Still, it would be good to add plus features, like other special functions (padic zeta, and related functions). Many of these are feasible if arbitrary precision arithmetic is present.

2. Put the library on arbitrary precision arithmetic base.
There is a good and modern arbitrary precision package developed for C++ (using the new features of C++11, too). It is the NTL (Number Theory Library), developed and maintained by Victor Shoup:
http://shoup.net/ntl/
If one would really like usable Morita gamma function, or padic Zeta, this development would be necessary.

3. Collaboration. I count with the help of likeminded people out there on the internet to help in the improvement of the library. Therefore all the code is freely available on GitHub.

Enjoy the code, as much as I enjoyed writing it (during the hasty days when moving from one apartment to another, and sending my daughter to school first time in her life ☺)!

István Mező