Elliptic Curve Point Multiplication in Fortran

Rian Neogi

1 Goal and Features

In this writeup, I describe details about my project, which involves writing code that does elliptic curve point multiplication in Fortran. The goal was to write some code that can multiply a point in some elliptic curve by an integer and compute the output. I wrote code that does this for Elliptic Curves over finite fields modulo some prime p (except $p = 2$ and $p = 3$), and also over the field of reals.

Along, the way I also had to write code to compute inverses modulo p, code to generate points of a finite field elliptic curve and code to add two points on an elliptic curve. As a result, I have added these functions as extra features of the program.

2 How to Use

In this section, I describe how to use and interact with the program.

There are two fortran files: *elliptic_finite_field* and *elliptic_reals*. These files contain code that do elliptic curve point multiplication over their respective field (either a finite field modulo p or the field of reals). First, one must compile the code using any preferred fortran compiler. More details about compilation are in Section 4.

Upon running the code, you will first need to enter the prime p (if you are running the finite field version). Note that you cannot enter 2 or 3 as a prime. I would like to remark that the code does not actually check whether the number you entered in prime or not, and the behaviour of the program may be undefined if you do not enter a prime. Next, you will be prompted to enter the coefficients a, b of the elliptic curve $y^2 = x^3 + ax + b$. You must input the coefficients separated by a space between them.

After the program accepts the coefficients, if its the finite field version, it will print out a list of all the points that lie in the elliptic curve. You may use this list as reference when you need to input points to the program. You will then be prompted with a few options:

1. Add two points on the curve
2. Multiply a point on the curve by an integer
3. Compute inverse of an element modulo p (only available for the finite field version)
4. Quit the program

Enter the number corresponding to the option that you wish to execute (that is, enter either 1, 2, 3 or 4).

Adding two points. If you chose option 1, you will be prompted to enter two points on the curve. The points *must* lie on the curve. For the finite field version, you may use the previously printed list to see which

points lie on the curve. You must enter 3 coordinates for each point, each coordinate must be seperated by a space. The first two coordinates denote the x, y-coordinates of that point. The last coordinate denotes whether the point is at infinity or not. We follow the convention from Sage, where $(0, 1, 0)$ denotes the point at infinity, and the third coordinate must be set to 1 if the point is not at infinity. After entering the coordinates for the two points, the program will add them and output their sum. The solution is displayed with upto 2 digits of precision.

Multiplying a point by an integer. If you chose option 2, you will be prompted to enter a point on the curve and an integer n. The point *must* lie on the curve. For the finite field version, you may use the previously printed list to see which points lie on the curve. You must enter 3 coordinates for the point. Each coordinate must be seperated by a space. The first two coordinates denote the x, y-coordinates of that point. The last coordinate denotes whether the point is at infinity or not. We follow the convention from Sage, where $(0, 1, 0)$ denotes the point at infinity, and the third coordinate is 1 if the point is not at infinity. After entering the coordinates for the point and the integer n, the program will multiply them. We use the doubling point multiplication algorithm to do this. More details are in Section 3. The solution is displayed with upto 2 digits of precision.

Computing inverses. This option is only available in the finite field version of the code. You will be prompted to enter an element of the field. The program will then run the Extended GCD algorithm to compute its inverse modulo p.

3 Explaining the Code

In this section, I will briefly explain each part of the code and the underlying algorithms that are used. Both files *elliptic_finite_field* and *elliptic_reals* are structured the same way. The code consists a main code block called *elliptic*, that handles the interface of the program, and various functions and subroutines that implement the various algorithms. The subroutines/functions are *pointMultiplication*, *addPoints*, *inv* and *checkPoint*.

Inverse. The computation of inverses modulo p is done by the function *inv*. This function takes as input an element x and a prime p and returns the inverse of x modulo p. The function does this by running the Extended Euclidean algorithm for computing GCD and the Bezout coefficients. The algorithm then returns the Bezout coefficient corresponding to x as its inverse.

Checking points on the Elliptic curve. The function *checkPoint* returns 1 or 0 depending on whether the input point in on the Elliptic curve or not. It does so by simply computing y^2 and $x^3 + ax + b$ and checking whether they evaluated to the same element in the field.

Adding Points. The addition of points over the elliptic curve is done by the subroutine *addPoints*. We use the method that was described in class, which provides explicit formulas for the x and y coordinate of the sum. First, there is some code that handles the case when one of the points is at infinity. Next, we check if the input points of reflections of each other over the x-axis. In this case, their sum is the point at infinity. After that, we branch into two cases. **Case 1.** The two input points are distinct: In this case, we compute the slope s between these points and explicitly compute the x coordinate of the sum to be $x^* = s^2 - x_1 - x_2$ (where x_1, x_2 are the x-coordinates of the input points), and the y coordinate of the sum to be $y^* = s(x_1 - x^*) - y_1$ (where y_1 is the y-coordinate of the first input point). **Case 2.** The two input points are the same: This case is similar to the previous one, except now s is the slope of the tangent line at the input point. The computation of x^*, y^* is then done similarly as before, but with this different value of s.

In both cases, we must take inverses modulo p when computing the slope if we are in the finite field setting. The code uses the *inv* function to do this.

Point Multiplication. The multiplication of a point on the elliptic curve by an integer is done by the subroutine *pointMultiplication*. The function takes as input the point (x, y) and an integer n.

The point multiplication algorithm that we use runs as follows: The algorithm first generates the binary representation of the integer n. The algorithm maintains a current solution (x^*, y^*) that is initialized to (x, y). The algorithm then traverses the binary representation in reverse and does the following for each bit: If the i-th bit is 0, it will add (x^*, y^*) to itself and set this to be the new current point. Otherwise, the i-th bit is 1, and it will add (x^*, y^*) to itself and then add (x, y) to this point. The resulting point is then set to be the new current point. After traversing all the bits in the binary representation, the algorithm will return the current point as the final solution. The algorithm will also check if n is negative, and if so, the algorithm will flip the y-coordinate before returning the final solution.

Running the algorithm this way ensures that it will terminate in $O(\log n)$ steps, rather than $O(n)$ steps using the naive method. This results in very fast point multiplication. While testing, I was able to multiply points by integers as large as 123456789, and the program would immediately return the solution (however, don't use integers that are too big since fortran has a limit on the size of integers that it can store).

4 Compiling and Running Fortran

In this section, I will describe how to compile Fortran code.

Linux. If you are on Linux, you will first need to install the gfortran compiler on your machine. You can do this by running *sudo apt install gfortran*. After that, run gfortran giving it as input one of the files. That is, either run *gfortran elliptic_finite_field.f90* or *gfortran elliptic_reals.f90*, depending on which field you want to work with. The compiler will then compile then code and produce a file named *a.out*. Then you may type *./a.out* onto the command line to run the program. If you want to change the field, you will need to recompile the code on the other file.

Windows. On Windows, you will need to download MinGW from here or some other site. Once you download the file, run it and you can choose to install gfortran from the options. After that, gfortran.exe will appear on *C:/MinGW/bin*. In order to compile the code, run *gfortran.exe* from command line giving it one of the code files as a parameter. This will generate an executable *a.exe* file. Run this executable to run the program.

For more instructions on how to install and use the compiler, you may refer to this.

5 Correctness

In order to verify that my code is correct, I have cross-checked the results of this program using Sage. That is, I plugged in some random points on some random elliptic curves and checked if multiplying them with some large integers produces the same points in my program as it does in Sage. So far, all the inputs I have tested match with sage in the finite field case. However, sometimes floating point errors in the real field code will compound when multiplying by a large integer. This will cause the output of my program to mismatch with that of sage. But things should be fine when multiplying a point by a small integer.