Go Finite Field – A Linear Finite-Field Solver

Yu-Ting Liu

SLAC National Accelerator Laboratory, Stanford University, Stanford, CA 94309, USA

(Dated: February 4, 2023)

In these notes I document Go Finite Field, a linear solver over finite fields specifically optimized for solving sparse linear systems. The program implements forward Gaussian elimination with back-substitution. All entries in the linear equations, as well as all intermediate steps, are stored sparsely in hash tables. The solver solves equations in batches, and leverages the easy-to-manage `goroutine`s provided by the `Go` programming language to implement partial parallelization in both forward elimination and back-substitution.

CONTENTS

I. Introduction 2

II. Linear Systems 2

A. Gaussian elimination 3

B. Parallelization 4

C. Modular arithmetics 4

III. How to Use the Program 4

A. Starting up the program 4

B. Workflow 5

IV. Input/Output 7

A. Equations 7

B. Solutions 8

Acknowledgments 8

References 8

I. INTRODUCTION

Linear systems abound in computations of theoretical high energy physics as well as many other scientific disciplines. And oftentimes, analytic calculations require solving linear equations over rational numbers. The simplest algorithm for doing this is Gaussian elimination, or row reduction. However, even in cases where the final solution is expected to be simple, the complexity of rational numbers in intermediate steps can grow beyond control. A way to circumvent this problem is to solve the linear system over finite fields, that is, solving the linear equations modulo a prime number p. Now the size of each entry, even in intermediate steps, is bounded by the size of p. Furthermore, basic arithmetics such as taking the product of two numbers is much faster if the numbers are integers of finite size.

In these notes I present a linear solver, Go Finite Field, that does just that; it solves linear equations modulo prime numbers. Moreover, it is optimized for sparse linear systems, which one often encounters in practice. The equations of the linear system, as well as all intermediate steps, are stored in hash tables to avoid unnecessary memory consumption. For the actual equation solving, the classic algorithm of Gaussian elimination proves to be robust: it does the job without introducing unnecessary complexities. Gaussian elimination consists of two major steps. A first step of forward elimination is performed to render the linear equations into row-echelon form. Then a second step of back-substitution generates the solution. Go Finite Field implements a variation of this which will be discuss in more detail below. To further improve performance, both these steps are parallelized to take advantage of modern computers with multi-core processors.

The rest of the notes are organized as follows,

II. LINEAR SYSTEMS

A linear system (over the rational numbers) is a set of unknown variables $x_1, x_2, \ldots, x_N$ related to each other through a set of M linear equations,

$$\begin{cases} c_{1,0} + c_{1,1}\, x_1 + \cdots + c_{1,N}\, x_N &= 0 \\ c_{2,0} + c_{2,1}\, x_1 + \cdots + c_{2,N}\, x_N &= 0 \\ & \vdots \\ c_{M,0} + c_{M,1}\, x_1 + \cdots + c_{M,N}\, x_N &= 0 \end{cases} \tag{1}$$

where $c_{i,j}$ are known rational coefficients which might be zero. M might be greater than, equal to, or smaller than N. If the number of non-zero $c_{i,j}$ is small compared to $N \times M$, we say that the system is *sparse*. Also, the set of equations may or may not admit a solution for $x_1, x_2, \ldots, x_N$; and when a solution exists, it may or may not be unique. We say the system has a *partial solution* if the solution is not unique.

A. Gaussian elimination

A classic algorithm for solving linear equations is Gaussian elimination. First, let us rewrite Eq. (1) as follows,

$$\begin{cases} c_{1,1}\,x_1 + c_{1,2}\,x_2 + \cdots + c_{1,N}\,x_N &= -c_{1,0} \\ c_{2,1}\,x_1 + c_{2,2}\,x_2 + \cdots + c_{2,N}\,x_N &= -c_{2,0} \\ & \vdots \\ c_{M,1}\,x_1 + c_{M,2}\,x_2 + \cdots + c_{M,N}\,x_N &= -c_{M,0}\,. \end{cases} \tag{2}$$

Without loss of generality, we assume $c_{1,1} \neq 0$. Then by subtracting suitable multiples of the first row, we can get rid of x_1 from all the other rows,

$$\begin{cases} c_{1,1}\,x_1 + c_{1,2}\,x_2 + \cdots + c_{1,N}\,x_N &= -c_{1,0} \\ c'_{2,2}\,x_2 + \cdots + c'_{2,N}\,x_N &= -c'_{2,0} \\ & \vdots \\ c'_{M,2}\,x_2 + \cdots + c'_{M,N}\,x_N &= -c'_{M,0}\,. \end{cases} \tag{3}$$

We call this step *forward elimination* (we eliminated x_1 from all the other equations,) and now we have a smaller linear system with only variables $x_2, \ldots, x_N$ and some new coefficients $c'_{i,j}$. Recursing on the size of the system, we can obtain a (partial) solution for $x_2, \ldots, x_N$ using the second through M-th equations. Plugging the solution for $x_2, \ldots, x_N$ back into the first equation, we obtain the solution for x_1 as well; this is called *back-substitution*.

B. Parallelization

C. Modular arithmetics

When solving linear equations over rational numbers, the complexity of coefficients $c'_{i,j}$ in Gaussian elimination (3) can become very large after many steps. One way to control the complexity is to solve the equations modulo a prime number p. This way, any coefficient is only allowed to take value in $\{0, 1, \ldots, p-1\}$, and the memory it takes to store such numbers is therefore bounded by p. However, p cannot be chosen to be too large either; the allowed size of p depends on the internal data type used for integer arithmetics. For example, Go Finite Field uses the built-in 64-bit integer `int64` in `Go`, which holds integers in the range -9223372036854775808 to 9223372036854775807 [1]. In order to do modular arithmetics, we need to be able to multiply two integers modulo p, without getting out of range of `int64`. We therefore require that $p^2 < 9223372036854775807$.

III. HOW TO USE THE PROGRAM

Go Finite Field is built using the programming language `Go` [1], which first needs to be installed on the user's system; otherwise the program has no external dependencies. Then the source code needs to be cloned into the local file system,

```
> git clone https://github.com/andyytliu/go-finite-field.git
> cd go-finite-field
```

A. Starting up the program

Now, the program can be started by the command,

```
> go run main.go
```

Alternatively, a binary file can be built first and then be called,

```
> go build main.go
> ./main
```

The program should look like the following once started,

```
*********************************************
****
****    Go Finite Field v0.7
****        A Linear Finite-Field Solver
****
****      Author:  Andy Yu-Ting Liu
****
*********************************************
********
********   Log file:  logs.txt
********   NumCPU/GOMAXPROCS: 16/16
********   Prime:  46337
********   Block size:  200
********
*********************************************
<Please enter command>
```

From the start-up screen above we also learn the following: 1) the log file `logs.txt` will be used to save information of intermediate steps when solving equations, including some error messages, 2) there are 16 processors on this computer to be used for parallelization, 3) the prime number 46,337 is used for modular arithmetics, and 4) a block size of 200 will be used when solving equations in parallel. The log file, the prime, and block size can all be set using flags when starting the program. The flags are as follows,

```
-block:  Block size of equations to solve in parallel (default 200)
-log:  Name for the log file (default "logs.txt")
-p:  Prime number to use in modular calculation (default 46337)
```

For example,

```
> go run main.go -p=2038074743 -block=1000 -log=MyLogs.txt
```

will use the prime 2038074743 and block size 1,000 when solving equations, and save all log messages to the file `MyLogs.txt`.

B. Workflow

A typical workflow for using the program consists of the following steps. First, any existing partial solutions can be read in at the very beginning by typing in

```
<Please enter command>  read_sol   some_solutions_file.txt
```

This step is optional, but `read_sol` should be called at most once and only at the very beginning of the program. Then one can read in any equations to be solved,

```
<Please enter command>  read_eq   some_equations_file.txt
```

The format for solutions and equations are described in Sec. IV. Unlike `read_sol`, `read_eq` can be called multiple times to read in multiple equation files, one at a time, and can be called at any time, even after some solving has been done; equations from the new file will simply be added to the list of unsolved equations.

Now one can start solving equations by calling `solve` followed by the number of equations to be solved. For example,

```
<Please enter command>  solve  40000
```

would solve the first 40,000 equations among the list of unsolved equations. The equations are solved in batches of size `Block size`, which can be changed by the command `block`; for example,

```
<Please enter command>  block  500
```

changes the `Block size` to 500. At any moment, the progress of solving equations can be checked with the command `status`. For example, here is what it might return,

```
<Please enter command>  status
*******************************************
Prime:  46337
Block size:  200
# of equations:  58420
# of solutions:  1987
*******************************************
```

showing that there are still 58,420 unsolved equations while there are already 1,987 (partial) solutions.

At the end, (partial) solutions can be saved to file,

```
<Please enter command>  write_sol   solutions_file.txt
```

Also, a help message containing all available commands can be obtained by `help`,

```
<Please enter command>  help
*********************************************
Available commands:
help:        Print this help message
status:      Print status of the solver
including numbers of equations
left and current solutions
block:       Change the block size
read_eq:     Read equations from file
read_sol:    Read solutions from file
write_sol:   Write solutions to file
solve:       Solve given number of equations
*********************************************
```

IV. INPUT/OUTPUT

The input/output equations and solutions are stored in plain text in a sparse format, so the disk space they take up is proportional to the number of non-zero entries.

A. Equations

For a linear system with N unknown variables $x_1, x_2, \ldots, x_N$, each variable x_i is represented by its index i; by definition, $x_0 \equiv 1$ and the index 0 represents the constant term. An equation with k variables (including the constant term) is represented by a line inside the input equation file with $2k$ numbers separated by space; every two numbers represent an index of a variable and the coefficient the variable is multiplied with. That is, an equation

$$c_0 + c_{i_1} x_{i_1} + c_{i_2} x_{i_2} + \cdots + c_{i_k} x_{i_k} = 0$$

is represented as a line

$$\mathtt{0} \quad \mathtt{c_0} \quad \mathtt{i_1} \quad \mathtt{c_{i_1}} \quad \mathtt{i_2} \quad \mathtt{c_{i_2}} \quad \cdots \quad \mathtt{i_k} \quad \mathtt{c_{i_k}}$$

in the file; the order of all the pairs $\mathtt{i_j}$ $\mathtt{c_{i_j}}$ with respect to each other does not matter. The indices $\{\mathtt{i_j}\}$ are non-negative integers and should not contain duplicates. The coefficients $\{\mathtt{c_{i_j}}\}$ should be rational numbers. As a concrete example, the following two equations

$$2\,x_1 - 7\,x_2 + x_3 + 17 = 0$$

$$-13\,x_1 + 81\,x_3 = 0$$

would be written as the following two lines in the file,

```
1 2 2 -7 3 1 0 17
1 -13 3 81
```

B. Solutions

(Partial) solutions of a linear system take the following form,

$$\begin{aligned} &\vdots \\ x_{j-1} &= \cdots \\ x_j &= c_{j,0} + c_{j,i_1} x_{i_1} + \cdots + c_{j,i_{k_j}} x_{i_{k_j}} \\ x_{j+1} &= \cdots \\ &\vdots \end{aligned}$$

Each equality above is reprented as a line in the input/output solution file. For example, the equation $x_j = \cdots$ above is represented as a line

```
j  0  c_{j,0}  i_1  c_{j,i_1}  i_2  c_{j,i_2}  ···  i_{k_j}  c_{j,i_{k_j}}
```

in the input/output file. For example, an equation

$$x_5 = -489 + 17\,x_6 - 32\,x_7$$

would be represented as

```
5 0 -489 6 17 7 -32
```

ACKNOWLEDGMENTS

This work was supported by the US Department of Energy under contract DE–AC02–76SF00515.

[1] The Go Programming Language, `https://go.dev`.