

# Polynomial models of time series over $(\mathbb{Z}/p)^n$

Micah J. Leamer  
Advisor Edward Green

May 13, 2003

Mathematical modeling has become an integral part of understanding how biological systems work. Some biological systems involve large numbers of components but it is difficult to observe the state of each of these components. In particular, gene regulatory networks i.e. systems of DNA strands and proteins, often involve a large number of components but the number of observations that may be taken are limited. Biologists are currently collecting data by acquiring different quantities of DNA and proteins then mixing them together and measuring how the quantities change over a series of time steps. Often thousands of quantities may be relevant to one another. Unfortunately in these experiments, biologists are only able to take measurements of the proteins and DNA at a small number of time steps, relative to the number of quantities. Given these constraints on the amount of data acquired, the problem of determining a system of functions that describes how the components interact becomes a purely mathematical one. Ideally, we would like to devise a computer program that allows biologists to enter their data along with some constraints on what is known about the way the data interacts. Then the program should return a network of functions that best fits the data and the constraints. The functions should map each observed time step to the next time step. Additionally the biologist should be able to choose what indeterminates each function is dependent upon, and be as simple as possible. The biologist could then change the constraints based on interpreting the network of functions and repeat the process until satisfied with the network.

A experimentalist observes the values of agents  $S_1, S_2, \dots, S_n$  in a gene regulatory network at times  $1, 2, \dots, t$

	$A_1$	$A_2$	$\dots$	$A_n$
time 1	$\alpha_{1,1}$	$\alpha_{1,2}$	$\dots$	$\alpha_{1,n}$
time 2	$\alpha_{2,1}$	$\alpha_{2,2}$	$\dots$	$\alpha_{2,n}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
time t	$\alpha_{t,1}$	$\alpha_{t,2}$	$\dots$	$\alpha_{t,n}$

such that  $\alpha_{i,j} \in \mathbb{R}^+$

We represent the quantities at each time step as points  $P_1, P_2, \dots, P_t \in \mathbb{R}^n$ , where  $t$  is the number of time steps and  $n$  the number of quantities of proteins or DNA. We assume that  $t \ll n$ .

$$\begin{aligned} P_1 &= (\alpha_{1,1}, \alpha_{1,2}, \dots, \alpha_{1,n}) \\ P_2 &= (\alpha_{2,1}, \alpha_{2,2}, \dots, \alpha_{2,n}) \\ &\vdots \\ P_t &= (\alpha_{t,1}, \alpha_{t,2}, \dots, \alpha_{t,n}) \end{aligned}$$

We then discretize the data by rounding it. Then we map the discretized points into  $(\mathbb{Z}/q)^n$  for some appropriately large prime  $q$ . The mapping of discretized points that one uses  $\pi : \mathbb{Z}^n \rightarrow (\mathbb{Z}/q)^n$  should be one that is reversible for all discretized points in the range of experimental data. For instance  $\pi$  could be the natural projection. So discretising gives us points  $P'_1, P'_2, \dots, P'_t \in \mathbb{Z}^n$ . Then we map  $\pi(P'_i) = p_i$  to obtain  $p_1, p_2, \dots, p_t \in (\mathbb{Z}/q)^n$

$$\begin{aligned} p_1 &= (a_{1,1}, a_{1,2}, \dots, a_{1,n}) \\ p_2 &= (a_{2,1}, a_{2,2}, \dots, a_{2,n}) \\ &\vdots \\ p_t &= (a_{t,1}, a_{t,2}, \dots, a_{t,n}) \end{aligned}$$

So now the problem is to find functions  $F : (\mathbb{Z}/q)^n \rightarrow (\mathbb{Z}/q)^n$  that best fits what we know about the biological situation. More specifically we must choose  $F$  such that  $F(p_i) = p_{i+1}$  for all  $i \in \{1, \dots, t-1\}$ . We may write  $F : (\mathbb{Z}/q)^n \rightarrow (\mathbb{Z}/q)^n$  componentwise as  $F = (f_1, f_2, \dots, f_n)$ , where  $f_i = f_i(x_1, x_2, \dots, x_n) : (\mathbb{Z}/q)^n \rightarrow \mathbb{Z}/q$ . It is known that all functions  $f : (\mathbb{Z}/q)^n \rightarrow \mathbb{Z}/q$  are polynomials. Since  $x_i^q = x_i$  in  $(\mathbb{Z}/q)^n$  we may limit the choices for  $f_i$  to those polynomials where the degree of each variable is bounded by  $n-1$ . Consequently there are only a finite number of choices for  $F$ . We show that when  $p_1, \dots, p_t$  are distinct there exists  $F = (f_1, f_2, \dots, f_n)$  such that  $F(p_i) = p_{i+1}$  with the total degree of  $f_i$  less than or equal to  $t-2$ . We then go on to explain a method for obtaining an  $f_i$  of lowest degree and only dependent upon a chosen set of variables.

Given a set  $S = \{p_1, \dots, p_t\}$  of  $t$  distinct data points in  $(\mathbb{Z}/q)^n$ , such that  $p_i = (a_{i,1}, a_{i,2}, \dots, a_{i,n})$ ,  $\forall i \in \{1, \dots, t\}$ . Then there exists a function  $F : (\mathbb{Z}/q)^n \rightarrow (\mathbb{Z}/q)^n$ .  $F = (f_1, f_2, \dots, f_n)$  such that  $F(p_j) = p_{j+1}$  for  $j \in \{1, \dots, t-1\}$ . Where each  $f_i = f_i(x_1, x_2, \dots, x_n)$  is a polynomial in  $n$  variables and the total degree of  $f_i \leq t-2$ . The proof of this is an extension of Lagrange interpolation to the multivariate case, which will work over any field not just  $\mathbb{Z}/q$

Proof:

$\forall p_i, p_j \in S$ , with  $i \neq j$ .  
 Choose  $k \in \{1, \dots, n\}$  such that  $a_{i,k} \neq a_{j,k}$   
 Such a  $k$  exists since  $p_i \neq p_j$  when  $i \neq j$ .

$$\forall i \text{ and } \forall j \neq i \text{ let } g_{i,j} = \frac{x_k - a_{j,k}}{a_{i,k} - a_{j,k}} \text{ such that } a_{i,k} \neq a_{j,k}$$

$$\text{Then } g_{i,j}(p_i) = 1 \quad \text{and} \quad g_{i,j}(p_j) = 0$$

$$\text{Let } h_i = \prod_{\substack{j \neq i \\ j \leq t-1}} g_{i,j}$$

Then  $h_i(p_i) = 1$  and  $h_i(p_j) = 0$ ,  $\forall j \neq i, j \in \{1, \dots, t-1\}$ . The degree of  $h_i$  is  $t-2$ .

$$\text{Let } f_j = \sum_{i=1}^{t-1} a_{i+1,j} h_i$$

Then  $f_j(p_i) = a_{i+1,j}$ ,  $\forall j \in \{1, \dots, t-1\}$ .  
 Therefore  $F(p_i) = (f_1(p_i), \dots, f_n(p_i)) = (a_{i+1,1}, \dots, a_{i+1,n}) = p_{i+1}$ , for  $i \in \{1, \dots, t-1\}$

For each  $i$  the degree of  $f_i$  is  $t-2$ .

Now that we have established an upper bound for the degree of existing  $f_i$ 's, lets consider another approach for finding simpler  $f_i$ 's of lower degree.

$$\text{Let } f \in K[x_1, x_2, \dots, x_n]$$

$$\text{Let } f = \sum_{i=1}^r c_i x^{\gamma_i} \text{ where } \gamma_i = (\gamma_{i,1}, \dots, \gamma_{i,n}) \text{ is an exponent in } n \text{ variables.}$$

$$\text{Such that } x^{\gamma_i} = \prod_{j=1}^n x_i^{\gamma_{i,j}}$$

$$\text{Let } \vec{x} = (x^{\gamma_1}, x^{\gamma_2}, \dots, x^{\gamma_r}) \text{ and } \vec{c} = (c_1, c_2, \dots, c_r) \text{ then } f = \vec{x} \cdot \vec{c}$$

$f$  is a solution for the  $k^{th}$  component of  $F$  iff

$$\begin{array}{rclcl} f_k(p_1) & = & \vec{x}(p_1) \cdot \vec{c} & = & a_{2,k} \\ f_k(p_2) & = & \vec{x}(p_2) \cdot \vec{c} & = & a_{3,k} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ f_k(p_{t-1}) & = & \vec{x}(p_{t-1}) \cdot \vec{c} & = & a_{t,k} \end{array}$$

this is equivalent to.

$$\begin{bmatrix} x^{\gamma_1}(p_1) & \dots & x^{\gamma_r}(p_1) \\ \vdots & \ddots & \vdots \\ x^{\gamma_1}(p_{t-1}) & \dots & x^{\gamma_r}(p_{t-1}) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_r \end{bmatrix} = \begin{bmatrix} a_{2,k} \\ a_{3,k} \\ \vdots \\ a_{t,k} \end{bmatrix}$$

This is written as  $[x^{\gamma_j}(p_i)] \cdot \vec{c} = \vec{a}_k$

$[x^{\gamma_j}(p_i)] \cdot \vec{c} = \vec{a}_k$  is a  $t - 1$  by  $r$  matrix whose  $i, j$  entry is  $x^{\gamma_j}[p_i]$  and  $\vec{c}$  is the column vector associated with  $f$ . The rows of  $[x^{\gamma_j}(p_i)]$  correspond to the points and each column corresponds to a monomial. So it follows that if we would like to find an  $f_k$  restricted to a certain set of monomials  $S = \{x^{\gamma_1}, x^{\gamma_2}, \dots, x^{\gamma_r}\}$ . Then all we must do is solve for  $\vec{c}$  in the matrix equation  $[x^{\gamma_j}[p_i]] \cdot \vec{c} = a_k$ . If the equation yields no solutions then there are no solutions for  $f_k$  in those monomials.

One obvious biological constraint on the function  $F$  that we end up with is that when  $p \approx p'$  then  $F(p) \approx F(p')$ . So small perturbations in the data should not dramatically change the out come over one time step. A simple way of accommodating this constraint is to limit our choice of  $f_i$  to those of low degree. So one may decide that any  $f_i$  over a certain degree  $d$  is not biologically relevant.

So we can find a solution for each  $f_i$  of lowest degree restricted to set of variables. The program I wrote utilizes standard linear algebra techniques to find such a solution. I will outline the method that my program uses to quickly obtain solutions. Let  $S = \{x^{\gamma_1}, x^{\gamma_2}, \dots, x^{\gamma_r}\}$  be the set of all monomials of degree less than or equal to  $d$  and only in the variables that prior biological understanding allows. Also order the monomials from lowest degree to highest degree. Let  $M$  be the  $t - 1$  by  $t - 1$  matrix made up of the first  $t - 1$  columns of  $[x^{\gamma_j}(p_i)]$ . Compute the reduce row echelon form of  $M$  adjoined with  $\vec{a}_i = (a_{2,i}, a_{3,i}, \dots, a_{t,i})^T$ . If the columns of  $M$  are linearly independent there will always be a solution. So if there is no solution we may throw out one of the columns of  $M$  and replace it with the next unused column in  $[x^{\gamma_j}(p_i)]$ . A linearly dependent column is one where there is no leading one in the reduced row echelon form. The program remembers the row operations that were already used and performs them on the next column before substituting it into  $M$  for one of the linearly dependent columns. Then continue performing row operations and replacing columns until either a solution exists or there are no unused columns from  $[x^{\gamma_j}(p_i)]$ . The first solution that exists will be of lowest degree since we ordered the monomials from lowest degree to highest degree. If we run out of columns without producing a solution then no solution exists given the constraints.

Proof: Let  $v_1, v_2, \dots, v_r$  be the columns of  $[x^{\gamma_j}(p_i)]$ . Let  $v_{l_1}, v_{l_2}, \dots, v_{l_{t-1}}$  be the columns used in  $M$  at the point that the last column substituted was  $v_k$  for some  $k$ , such that  $t \leq k \leq r$ . Only linearly dependent columns have been substituted for in  $M$ , which implies

$\text{span}(\{v_{l_1}, v_{l_2}, \dots, v_{l_{t-1}}\}) = \text{span}(\{v_1, v_2, \dots, v_k\})$  a solution exist for  $f_j$  in the monomials  $x^{\gamma_1}, x^{\gamma_2}, \dots, x^{\gamma_k}$  iff  $\vec{a}_j \in \text{span}(\{v_1, v_2, \dots, v_k\})$

Also a solution exists to the equation  $M\vec{c} = \vec{a}_j$  at this iteration iff  $\vec{a}_j \in \text{span}(\{v_{l_1}, v_{l_2}, \dots, v_{l_{t-1}}\}) = \text{span}(\{v_1, v_2, \dots, v_k\})$ .

So a solution exists in the monomials  $x^{\gamma_{i_1}}, x^{\gamma_{i_2}}, \dots, x^{\gamma_{i_{t-1}}}$  iff a solution exists in the monomials  $x^{\gamma_1}, x^{\gamma_2}, \dots, x^{\gamma_k}$

It follows that since no solution existed at the previous iteration then a solution does not exist in the monomials  $x^{\gamma_1}, x^{\gamma_2}, \dots, x^{\gamma_{k-1}}$ .

The monomials are ordered from lowest to highest degree so if a solution exists after the current substitution then it is a solution of lowest degree.

If the columns span  $(\mathbb{Z}/q)^{t-1}$  then that is a sufficient condition for a solution to exist since they span the whole space. The probability that  $m$  random columns of size  $t - 1$  will span  $(\mathbb{Z}/q)^{t-1}$  is

$$\prod_{i=0}^{t-2} \left(1 - \frac{1}{q^{m-i}}\right)$$

when  $m > t$  this probability is approximately 1.

Proof: This is the same probability that  $t - 1$  random row vectors of size  $m$  will have rank  $t - 1$  let  $v_1, \dots, v_{i-1}$  be linearly independent vectors in  $(\mathbb{Z}/q)^m$ . the probability that a random vector  $v_i$  will be linearly independent is.

$$\begin{aligned} & \frac{\# \text{ of choices for } v_i \text{ that are linearly independent}}{\# \text{ of possible vectors for } v_i} \\ &= \frac{\# \text{ of possible vectors for } v_i - \# \text{ of vectors in } \text{span}(v_1, \dots, v_{i-1})}{\# \text{ of possible vectors for } v_i} \\ &= \frac{q^m - q^{i-1}}{q^m} = 1 - \frac{1}{q^{m-(i-1)}} \end{aligned}$$

So the total probability is the product from  $i$  equals 1 to  $t - 1$

$$\prod_{i=1}^{t-1} \left(1 - \frac{1}{q^{m-(i-1)}}\right) = \prod_{i=0}^{t-2} \left(1 - \frac{1}{q^{m-i}}\right)$$

Thus probabilistically we need to only use on the order of  $t$  columns to find a solution for each  $f_i$ . Although it is simple to create large sets of non-spanning columns it is not likely to occur in experimentation, which retains some elements of chaotic behavior.

As a side note. Given points  $\{p_1, \dots, p_m\}$  in  $(\mathbb{Z}/q)^n$ ,  $q$  a prime, if we are concerned with finding all Functions  $F : (\mathbb{Z}/q)^n \rightarrow (\mathbb{Z}/q)^n$  such that  $F(p_i) = F(p_{i+1})$ . All functions  $F : (\mathbb{Z}/q)^n \rightarrow (\mathbb{Z}/q)^n$  may be written componentwise as  $F = (f_1, \dots, f_n)$  where each  $f_i = f_i(x_1, \dots, x_n)$  is a polynomial in  $n$  variables over  $\mathbb{Z}/q$ . Let  $p_i = (\alpha_{i,1}, \dots, \alpha_{i,n})$ . Given any two such functions  $F = (f_1, \dots, f_n)$  and  $F' = (f'_1, \dots, f'_n)$ ,  $f_i(p_j) = f'_i(p_j) = \alpha_{j+1,i}$ , implies

$(f_i - f'_i)(p_j) = 0 \quad (\forall j \in \{1, \dots, m-1\})$ . Let  $\mathcal{I}(V)$  be the ideal of a variety  $V$ . For all  $f$  in  $\mathcal{I}(p_1, \dots, p_{m-1})$ ,  $(f_i + f)(p_j) = f_i(p_j) + f(p_j) = \alpha_{j+1_i} + 0 = \alpha_{j+1_i}$ . There for, given a particular solution  $F = (f_1, \dots, f_n)$  all solutions are of the form  $F = (f_1 + g_1, \dots, f_n + g_n)$  where  $g_1, \dots, g_n \in \mathcal{I}(p_1, \dots, p_{m-1})$ .

Thus given a particular solution the problem of finding all solutions then becomes one of representing the ideal  $\mathcal{I}(p_1, \dots, p_{m-1})$  in a nice way, which traditionally means finding a gröbner basis.

Here is are sample input and output files for the program.

file input.txt

```
0 1 3 2;
2 0 3 1;
3 4 0 2;
0 2 3 2;
2 0 3 2;
4 0 0 2;
```

The rows correspond to points.

file inputs.txt

```
1 2 4 1 4
```

The first number is indicates the program is looking for  $f_1$ .

The second number indicates the solution will be based on two variables.

The third number is the maximum degree to check for solutions.

The last two numbers correspond to the numbers or variables to be used.

file output.txt

$$f_1 = -5x_1^1 + -6x_4^1 + -4x_1^2$$

$$0 \quad -5 \quad -4 \quad 0 \quad -5 \quad -3$$

This solution was computed over  $\mathbb{Z}/7$ . One can check that the row of numbers after the function , which is the output for the function when one enters the points is the same as the first column of the data over  $\mathbb{Z}/7$ .

The code in C++ follows.

```

#include <iostream.h>
#include <fstream.h>
#include <math.h>

typedef int Bool;

const int prime=7;
const int Var=60;
const int Points=27;
const int pl=20;

int numVar( ifstream& );

int numPoints( ifstream& , int&);

int readData( ifstream& , ifstream& , int myData[Points][Var] , int& , int& ,
bool BtwoPrime[pl],
                int theData[Points][Var] , int solVar , int
                numRelVar , int Power ,
                int InTermsof[Var] , int AllExp[Var][Points-1]);

void initRREF(int myData[Points][Var], int Matrix[Points - 1][Points - 1], int
swaps[Points - 1],
                int curSol[Points - 1], int& V , int& P , int
                funcVars[Points - 1] , int& curVar);

void linIndy(int myData[Points][Var] , int Matrix[Points - 1][Points - 1],
int curRow[2], int swaps[Points - 1] , int
mults[Points - 1],
int adds[Points - 1][Points - 1] , bool BtwoPrime[pl]
,
int funcVars[Points - 1] , int& curVar , int& V ,
int& P , int solRed[Points -1] ,
int& solVar , int theData[Points - 1][Var] ,
ofstream&);

int Swap(int myData[Points][Var] ,int Matrix[Points - 1][Points - 1], int&
Clm,
int swaps[Points - 1] , int mults[Points - 1],int
adds[Points - 1][Points - 1] ,
int funcVars[Points - 1] , int curRow[2] , int& P , int& V ,
int solRed[Points -1] ,
int& solVar , int theData[Points - 1][Var] , ofstream&);

int Reduce(int Matrix[Points - 1][Points - 1], int& Clm,
int swaps[Points - 1] , int mults[Points - 1],int
adds[Points - 1][Points - 1] ,
bool BtwoPrime[pl] , int& P , int& V , ofstream&);

void makeFunc(int swaps[Points - 1] , int mults[Points - 1],int adds[Points -
1][Points - 1] ,
int& P , int& V , int funcVars[Points - 1] , int
funcCoef[Points - 1][Var] ,
int myData[Points][Var] , int theData[Points][Var]
, ofstream ,
int AllExp[Var][Points-1] , int& solVar , int
inTermsOf[Var] , int& numRelVar);

void rowReduce(int swaps[Points - 1] , int mults[Points - 1],int adds[Points -
1][Points - 1] ,
int& P , int& V , int newClm[Points - 1] , int&
upto);

int main()
{

```

```

int V;
int P;
int solVar = 0;
int Power = 0;
int numRelVar = 0;
int InTermsOf[Var];
int AllExp[Var][Points - 1];
int curVar=1;
bool BtwoPrime[pl];
int funcVars[Points - 1];
int funcCoef[Points - 1][Var];
int curSol[Points - 1];
int myData[Points][Var];
int theData[Points][Var];
int swaps[Points - 1];
int mults[Points - 1];
int adds[Points - 1][Points - 1];
int Matrix[Points - 1][Points - 1];
int newData[Points][Var];
int solRed[Points - 1];
int curRow[2];

ifstream Data;
ifstream Parameters;
ofstream Result;

Data.open("input.txt");

Parameters.open("inputs.txt");

Result.open("output.txt");

V=numVar(Data);

P=numPoints(Data , V);

Data.close();
Data.open("input.txt");

Parameters >> solVar >> numRelVar >> Power;
curRow[1]=P;

V=readData(Data , Parameters , myData , V , P , BtwoPrime , theData ,
solVar , numRelVar ,
Power , InTermsOf , AllExp);

initRREF(myData, Matrix , swaps , curSol , V, P, funcVars , curVar);

linIndy(myData , Matrix , curRow , swaps , mults , adds , BtwoPrime ,
funcVars , curVar , V , P , solRed , solVar , theData ,
Result);

makeFunc(swaps , mults , adds , P , V , funcVars , funcCoef , myData ,
theData , Result ,
AllExp , solVar , InTermsOf , numRelVar);

return 0;
}

int numVar( ifstream& Data)
{
int V;
char inChar;
int x;
V=1;
Data >> x;
Data.get(inChar);

```

```

        while (inChar != ';')
        {
            V++;
            Data >> x;
            Data.get(inChar);
        }
        return V;
    }

int numPoints( ifstream& Data ,int& V)
{
    int P;
    Data.ignore(5*V, ';');
    P=1;
    while (Data)
    {
        Data.ignore(5*V, ';');
        P++;
    }
    return P;
}

int readData( ifstream& Data , ifstream& Parameters , int myData[Points][Var]
, int& V , int& P ,
                bool BtwoPrime[p1] , int theData[Points][Var] ,
                int solVar , int numRelVar , int Power , int
                InTermsOf[Var] ,
                int AllExp[Var][Points-1])
{
    int x;
    int N;
    int Sorter[Points];
    int Exponent[Points];
    int i;
    int j;
    char newChar;
    int curPow=1;
    for (i=1; i <= P; i++)
    {
        for (j=1; j<= V; j++)
        {
            Data >> x;
            theData[i][j]=(x % prime);
        }
        Data.get(newChar);
    }
    for (i=1; i<= p1; i++)
    {
        if (((prime - 2) % int(pow(2, i))) >= pow(2, i - 1))
        {
            BtwoPrime[i]=1;
        }
        else
        {
            BtwoPrime[i]=0;
        }
    }

    cout << solVar << " " << numRelVar << " " << Power ;
    for (i=1; i<= numRelVar; i++)
    {
        Parameters >> InTermsOf[i];
    }
    for (i=1; i<= P; i++)
    {
        myData[i][1]=1;
    }
}

```

```

N=2;
Sorter[1]=1;
Exponent[1]=1;
for (i=2 ; i<= Power+1 ; i++)
{
    Sorter[i]=0;
}
for (i=2 ; i<= numRelVar ; i++)
{
    Exponent[i]=0;
}

for (i=1; i<=numRelVar; i++)
{
    AllExp[1][i]=0;
}
while ((N<=(Var - 1)) && (curPow<=Power))
{
    for (i=1; i<=numRelVar; i++)
    {
        AllExp[N][i]=Exponent[i];
    }
    for (i=1; i<=P; i++)
    {
        myData[i][N]=1;
        for (j=1; j<=numRelVar; j++)
        {
            myData[i][N]=(myData[i][N]*(int(pow(theData[i]
[InTermsOf[j]],Exponent[j])) % prime)) %
prime;
        }
    }
    if (Sorter[1]==numRelVar)
    {
        i=1;
        while (Sorter[i]==numRelVar)
        {
            i++;
        }
        i=i-1;
        if (i==curPow)
        {
            curPow++;
            for (j=1; j<=curPow; j++)
            {
                Sorter[j]=1;
            }
            Exponent[1]=curPow;
            Exponent[numRelVar]=0;
        }
        else
        {
            Sorter[i+1]++;
            for (j=1; j<=i; j++)
            {
                Sorter[j]=Sorter[i+1];
            }
            Exponent[numRelVar]=0;
            Exponent[Sorter[i+1]]=i+1;
        }
    }
    else
    {
        Exponent[Sorter[1]]=Exponent[Sorter[1]]-1;
        Sorter[1]++;
        Exponent[Sorter[1]]=Exponent[Sorter[1]]+1;
    }
    N++;
}

```

```

return N-1;

}

void initRREF(int myData[Points][Var], int Matrix[Points - 1][Points - 1], int
swaps[Points - 1],
                int curSol[Points - 1], int& V, int& P, int
                funcVars[Points - 1] , int& curVar)
{
    for (int i=1; i <= P-1; i++)
    {
        curSol[i]=myData[i][curVar];
        for (int j=1; j<= P-1; j++)
        {
            Matrix[i][j]=myData[i][j];
        }
    }
    for (i=1; i <= P-1; i++)
    {
        funcVars[i]=i;
        swaps[i]=i;
    }
}

void linIndy(int myData[Points][Var] , int Matrix[Points - 1][Points - 1],
            int curRow[2] , int swaps[Points - 1] , int
            mults[Points - 1],
            int adds[Points - 1][Points - 1] , bool BtwoPrime[pl]
            ,
            int funcVars[Points - 1] , int& curVar , int& V ,
            int& P , int solRed[Points -1] ,
            int& solVar , int theData[Points - 1][Var], ofstream&
            Result)
{
    int Clm=1;
    int i;
    int j;
    while ( Clm <= (P - 1))
    {
        if (Matrix[Clm][Clm]==0)
        {
            cout << " swaps " << Clm << " " ;
            Clm=Swap( myData , Matrix , Clm , swaps , mults , adds
                , funcVars , curRow , P , V ,
                solRed , solVar , theData , Result);
            cout << "row " << curRow[1] << " " ;
        }
        else
        {
            cout << " reduce " << Clm << " ";
            Clm = Reduce(Matrix, Clm, swaps , mults , adds ,
                BtwoPrime , P , V , Result);
        }
    }
    /*for (i=1;i<=P-1;i++)
    {
        for (j=1; j<=P-1; j++)
        {
            Result << Matrix[i][j] << " ";
        }
        Result << endl;
    }
    Result << endl;*/
}

```

```

    }
}

int Swap(int myData[Points][Var] ,int Matrix[Points - 1][Points - 1], int&
Clm,
        int swaps[Points - 1] , int mults[Points - 1],int
        adds[Points - 1][Points - 1] ,
        int funcVars[Points - 1] , int curRow[2] , int& P , int& V ,
        int solRed[Points -1] ,
        int& solVar , int theData[Points - 1][Var] , ofstream&
        Result)
{
    int entry=0;
    int row=Clm-1;
    int newClm[Points - 1];
    int column=Clm-1;
    int i;
    int j;
    int cur;
    while ((entry == 0) && (column < P-1))
    {
        column++;
        row=Clm-1;
        while ((entry == 0) && (row < P-1))
        {
            row++;
            entry=Matrix[row][column];
        }
    }
    /*Result << "swaps " << entry << endl;*/

    if ((entry==0) && (curRow[1] <= V))
    {
        for ( i=1; i<=P-1; i++)
        {
            solRed[i]=theData[i+1][solVar];
        }
        int upto=Clm-1;
        rowReduce(swaps , mults , adds , P , V , solRed , upto);
        cout << " solution " << solRed[Clm] << " ";
        if (solRed[Clm]==0)
        {
            funcVars[Clm]=0;
            Clm++;
            /*Result << "swaps 1.1 Clm " << Clm << " Linearly
            dependent solution " << endl << endl;*/
        }
        else
        {
            funcVars[Clm]=curRow[1];
            for (i=1; i <= (P-1); i++)
            {
                newClm[i]=myData[i][curRow[1]];
            }
            int upto = Clm - 1;
            rowReduce(swaps , mults , adds , P , V , newClm ,
            upto);
            for ( i=1; i <= (P-1); i++)
            {
                Matrix[i][Clm]=newClm[i];
            }
            /*Result << "swaps 1.2" << " substituted myData clm "
            << curRow[1] << " for clm " << Clm << endl << endl;*/
            curRow[1]++;
        }
    }

}

else if ((curRow[1] > V) && (entry == 0))

```

```

{
    for (i=1; i<=P-1; i++)
    {
        solRed[i]=theData[i+1][solVar];
    }
    int upto=Clm-1;
    rowReduce(swaps , mults , adds , P , V , solRed , upto);
    if (solRed[Clm]==0)
    {
        funcVars[Clm]=0;
        /*Result << "swaps 2.1" << endl << endl;*/
        Clm++;
    }
    else
    {
        for ( i=1 ; i<=P-1; i++)
        {
            for ( j=1 ; j<=P-1; j++)
            {
                Matrix[i][j]=0;
            }
            Matrix[i][i]=1;
        }
        /*Result << "swaps 2.2 no solution" << endl <<
endl;*/
        Result << "no solution" << endl;
    }
}
else
{
    swaps[Clm]=row;
    for ( cur=1; cur <= P - 1; cur++)
    {
        entry=Matrix[Clm][cur];
        Matrix[Clm][cur]=Matrix[row][cur];
        Matrix[row][cur]=entry;
    }
    entry=funcVars[Clm];
    funcVars[Clm]=funcVars[column];
    funcVars[column]=entry;
    for ( cur=1; cur <= P - 1; cur++)
    {
        entry=Matrix[cur][Clm];
        Matrix[cur][Clm]=Matrix[cur][column];
        Matrix[cur][column]=entry;
    }
    /*Result << "swaps 3" << endl << endl;*/
}
return Clm;
}

int Reduce(int Matrix[Points - 1][Points - 1], int& Clm,
           int swaps[Points - 1] , int mults[Points - 1],int
           adds[Points - 1][Points - 1] ,
           bool BtwoPrime[pl] , int& P , int& V , ofstream& Result)
{
    int inv = 1;
    int invList[pl];
    invList[1]=Matrix[Clm][Clm];
    if (BtwoPrime[1]==1)
    {
        inv = invList[1];
    }
    for (int i=2; i<= pl - 1; i++)
    {
        invList[i]=(invList[i-1]*invList[i-1]) % prime;
    }
}

```

```

        if (BtwoPrime[i]==1)
        {
            inv = (inv*invList[i]) % prime;
        }
    }
    mults[Clm]=inv;
    for (i=Clm; i<= P-1; i++)
    {
        Matrix[Clm][i] = (Matrix[Clm][i]*inv) % prime;
    }
    for (i=1; i<=Clm - 1; i++)
    {
        adds[Clm][i] = Matrix[i][Clm];
        for (int j=Clm; j<=P-1; j++)
        {
            Matrix[i][j] = (Matrix[i][j] - adds[Clm][i]
                *Matrix[Clm][j]) % prime;
        }
    }
    adds[Clm][i] = Matrix[i][Clm];

    adds[Clm][Clm]=0;
    for (i= Clm + 1; i<=P-1; i++)
    {
        adds[Clm][i]= Matrix[i][Clm];
        for (int j=Clm; j<=P-1; j++)
        {
            Matrix[i][j] = (Matrix[i][j] - adds[Clm][i]
                *Matrix[Clm][j]) % prime;
        }
    }

    return Clm+1;
}

void makeFunc(int swaps[Points - 1] , int mults[Points - 1],int adds[Points -
1][Points - 1] ,
                int& P , int& V , int funcVars[Points - 1] , int
                funcCoef[Points - 1][Var] ,
                int myData[Points][Var] , int theData[Points][Var]
                , ofstream Result ,
                int AllExp[Var][Points-1] , int& solVar , int
                inTermsOf[Var] , int& numRelVar)
{
    int upto = P-1;
    int v;
    int p;
    int x;
    int i;
    int j;
    int c;
    int k;
    int newClm[Points - 1];
    for (p=1; p<= P - 1; p++)
    {
        newClm[p]=theData[p + 1][solVar];
    }

    rowReduce(swaps , mults , adds , P , V , newClm , upto);
    Result << "f_" << solVar << " = ";

    i=1;
    while ((newClm[i] == 0) && (i<=P-1))
    {
        i++;
    }
}

```

```

if (i<=P-1)
{
    Result << newClm[i];
    for (int j=1; j<=numRelVar; j++)
    {
        if (AllExp[funcVars[i]][j] != 0)
        {
            Result << "x_" << inTermsOf[j] << "^" <<
            AllExp[funcVars[i]][j];
        }
    }
}
for (j=i+1; j<=P-1; j++)
{
    if (newClm[j] != 0)
    {
        Result << " + " ;
        Result << newClm[j];
        for (i=1; i<=numRelVar; i++)
        {
            if (AllExp[funcVars[j]][i] !=0)
            {
                Result << "x_" << inTermsOf[i] << "^"
                << AllExp[funcVars[j]][i];
            }
        }
    }
}
Result << endl << endl;
Result << theData[1][solVar];
for (i=1; i<=P-1; i++)
{
    x=0;
    for (j=1; j<=P-1; j++)
    {
        c=newClm[j];
        for (k=1; k<=numRelVar; k++)
        {
            c = (c*(int(pow(theData[i]
[inTermsOf[k]],AllExp[funcVars[j]]
[k])) % prime))%prime;
        }
        x = (x+c) % prime ;
    }
    Result << " " << x;
}
Result << endl << endl;

/*for (i=1; i<=P; i++)
{
    for (j=1; j<=V; j++)
    {
        Result << myData[i][j] << " ";
    }
    Result << endl;
}*/
}

void rowReduce(int swaps[Points - 1] , int mults[Points - 1],int adds[Points -
1][Points - 1] ,
                int& P , int& V , int newClm[Points - 1] , int&
upto)
{
    int entry;
    int j;
    int k;
    for (j=1; j<= upto; j++)
    {
        entry=newClm[swaps[j]];

```

```
newClm[swaps[j]]=newClm[j];
newClm[j]=entry;

newClm[j]=(newClm[j]*mults[j]) % prime;
for (k=1; k<=P-1; k++)
{
    newClm[k] = (newClm[k] - adds[j][k]*newClm[j]) %
prime;
}
}
```