```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Initialise the structures we need
// First initialise the fields rational and mod3 then see
"field" equal to their union

typedef struct rational_{
  long num;
  long den;
}rational;

typedef struct mod3_{
  long m;
}mod3;

union field{
  rational F_rat;
  long F_mod3;
}field;

// Set up the matrix structure and its intelligence

typedef struct _intelligence {
  int dim;
  int *m;
} intelligence;

// To remind ourselves that we pass around a pointer to
intelligence we rename
// the intelligence pointer "intelligenceP"

typedef intelligence * intelligenceP;

typedef struct _matrix {
  int dim;
  int row;
  int col;
  intelligenceP status;
  union field det;
  union field *m;
} matrix;

// To remind ourselves that we pass around a pointer to a
matrix we rename
// the matrix pointer "matrixP"

typedef matrix * matrixP;

#define RATIONAL 0
#define MOD3 1
```



```
// global variable fieldType represents the field we are
currently working over
// global variable z is used to break up the interface
structure

int fieldType, z;

/*******************************mod***********************
****************/
  long mod(long a){

    // this function returns the magnitude of an integer

    if (a<0){
      return -a;
    }
    else {
      return a;
    }
  }

/*******************************fieldMod******************
****************/
  union field fieldMod(union field a){

    // this function returns the magnitude of a field element

    a.F_rat.num = mod(a.F_rat.num);
    return a;
  }

/*******************************compareInt****************
****************/
  int compareInt(long a, long b){ // the order of a and b
matters!

    // this function returns: 1  if a>b
    //                        0  if a=b
    //                       -1  if a<b
    // where a and b are integers

    if (a < b){
      return -1;
    }
    else if (a > b){
      return 1;
    }
    else{
      return 0;
    }
  }

/*******************************maxInt********************
****************/
```



```
  int maxInt(int a, int b){

    // this function returns the larger of the two integers

    if (a > b){
      return a;
    }
    else{
      return b;
    }
  }

/*******************************longToMod3****************
****************/
  long longToMod3(long a){

    // this function converts a long integer into modulo 3
and returns the mod 3 form

    if (a >= 0){
      return (a %= 3);
    }
    else{

      // a mathematical trick to convert negative numbers

      return longToMod3(3 + (a %= 3));
    }
  }

/*******************************modgcd********************
****************/
  long modgcd(long a, long b){
    long dummy;

    // this function returns the magnitude of the greatest
common divisor of two long integers

    while (b != 0){
      dummy = b;
      b = (a %= b);
      a = dummy;
    }
    return mod(a);
  }

/*******************************rationalise***************
****************/
  rational rationalise(rational a){
    long ratDen;

    // this function converts a pair of integers into
CANONICAL rational form
```



```
    // if denominator is zero return an error message

    if (a.den == 0){
      a.num=0;
      printf("*******System Error: can not divide by
zero!********\n");
      return a;
    }

  // make sure denominator is positive

    if (a.den < 0){
      a.den = (-1) * a.den;
      a.num = (-1) * a.num;
    }

  // cancel all gcds

    ratDen = modgcd(a.num, a.den);
    a.num = a.num / ratDen;
    a.den = a.den / ratDen;

    return a;
  }

/*******************************stringToRational**********
****************/
  rational stringToRational(char* stringEntered){
    rational temp;
    int buffer, bufferStart;
    char temp2[500];

    // this function reads a string of the form "a/b" picks
out the long integers
    // a and b then returns the CANONICAL form of the
corresponding rational

    buffer = 0;
    bufferStart = 0;

    // search through string untill we find "/"

    while(stringEntered[buffer+bufferStart] != '/'){
      buffer++;
    }

    // convert all the string before "/" to long integer and
store as rational numerator

    temp.num = atol(strncpy(temp2,
&stringEntered[bufferStart], buffer));
    bufferStart = buffer + 1;
```

```
    // convert all the string after "/" to long integer and
store as rational denominator

    buffer = strlen(stringEntered) - 1;

    temp.den = atol(strncpy(temp2,
&stringEntered[bufferStart], buffer));

    return rationalise(temp);
  }

/*******************************displayMod3****************
***************/
  void displayMod3(long a){

    // this function displays a mod 3 field element

    printf("%li", a);
  }

/*******************************displayRational************
***************/
  void displayRational(rational a){

    // this function displays a rational field element

    printf("%li/%li", a.num, a.den);
  }

/*******************************displayField***************
***************/
  void displayField(union field a){
    rational dummyRat;
    long dummyMod3;

    // this function finds the field we are working over and
displays the
    // appropriate field element

    if(fieldType == RATIONAL){
      dummyRat.num = a.F_rat.num;
      dummyRat.den = a.F_rat.den;
      displayRational(dummyRat);
    }
    else{
      dummyMod3 = a.F_mod3;
      displayMod3(dummyMod3);
    }
  }

/*******************************compareMod3****************
***************/
  int compareMod3(long a, long b){ // the order of a and b
matters!
```



```
    // this function returns: 1  if a>b
    //                        0  if a=b
    //                       -1  if a<b
    // where a and b are mod 3 field elements

    return compareInt(longToMod3(a), longToMod3(b));
  }

/*******************************compareRational************
***************/
  int compareRational(rational a, rational b){ // the order
of a and b matters!
    a = rationalise(a);
    b = rationalise(b);

    // this function returns: 1  if a>b
    //                        0  if a=b
    //                       -1  if a<b
    // where a and b are rational field elements

    if (a.den == b.den){
      return compareInt(a.num, b.num);
    }
    else{
      return compareInt(a.num * b.den, b.num * a.den);
    }
  }

/*******************************fieldCompare***************
***************/
  int fieldCompare(union field a, union field b){
    rational dummyRatA, dummyRatB;
    long dummyMod3A, dummyMod3B;

    // this function finds the field we are working over and
    // then returns: 1  if a>b
    //               0  if a=b
    //              -1  if a<b
    // where a and b are arbitrary field elements

    if(fieldType == RATIONAL){
      dummyRatA.num = a.F_rat.num;
      dummyRatA.den = a.F_rat.den;
      dummyRatB.num = b.F_rat.num;
      dummyRatB.den = b.F_rat.den;
      return compareRational(dummyRatA, dummyRatB);
    }
    else{
      dummyMod3A = a.F_mod3;
      dummyMod3B = b.F_mod3;
      return compareMod3(dummyMod3A, dummyMod3B);
    }
  }
```



```
/*******************************mod3Input******************
***************/
  long mod3Input(void){
    long a;

    // this function reads a user input as a long integer,
converts it to mod 3
    // and returns the mod 3 form

    scanf("%li", &a);
    return longToMod3(a);
  }

/*******************************rationalInput**************
***************/
  rational rationalInput(void){
    char word[500];

    // this function reads a user input as a string, converts
it to a rational
    // and returns the CANONICAL rational form

    scanf("%s", word);
    return stringToRational(word);
  }

/*******************************fieldInput*****************
***************/
  union field fieldInput(void){
    union field a;

    // this function finds the field we are working over,
reads the appropriate
    // user input, convert the input into the arbitrary field
element and returns
    // that arbitrary field element

    if(fieldType == RATIONAL){
      a.F_rat = rationalInput();
      return a;
    }
    else{
      a.F_mod3 = mod3Input();
      return a;
    }
  }

/*******************************mod3Add********************
***************/
  long mod3Add(long a, long b){

    // this function returns the sum of two mod 3 elements
```



```
    return longToMod3(a + b);
  }

/*******************************rationalAdd****************
***************/
  rational rationalAdd(rational a, rational b){
    rational x;
    long ratDen, num1, num2;

    // this function returns the CANONICAL form of the sum of
two rational numbers

    // if we can spot a common denominator we reduce the
complexity of addition and
    // multiplication over the integers at the exspense of
one integer comparison

    if (a.den == b.den){
      x.num = a.num + b.num;
      x.den = a.den;
    }
    else{

      // to formulate the sum, we perform the same
calculations several times
      // so we calculate them at the beginning to reduce
complexity

      num1 = a.num * b.den + b.num * a.den;
      num2 = a.den * b.den;
      ratDen = modgcd(mod(num1), num2);

      x.num = (num1 / ratDen);
      x.den = (num2 / ratDen);
    }
    return rationalise(x);
  }

/*******************************fieldAdd*******************
***************/
  union field fieldAdd(union field a, union field b){
    rational dummyRatA, dummyRatB;
    long dummyMod3A, dummyMod3B;

    // this function finds the field we are working over and
returns the sum of the
    // two arbitrary field elements

    if(fieldType == RATIONAL){
      dummyRatA.num = a.F_rat.num;
      dummyRatA.den = a.F_rat.den;
      dummyRatB.num = b.F_rat.num;
      dummyRatB.den = b.F_rat.den;
      a.F_rat = rationalAdd(dummyRatA, dummyRatB);
```

```
      return a;
    }
    else{
      dummyMod3A = a.F_mod3;
      dummyMod3B = b.F_mod3;
      a.F_mod3 = mod3Add(dummyMod3A, dummyMod3B);
      return a;
    }
  }

/*******************************mod3Sub*********************
****************/
  long mod3Sub(long a, long b){ // the order of a and b
matters

    // this function returns mod 3 element a minus mod 3
element b

    return longToMod3(a - b);
  }

/*******************************rationalSub*****************
****************/
  rational rationalSub(rational a, rational b){ // the order
of a and b matters

    // this function returns the CANONICAL form of rational
number a minus
    // rational number b

    b.num = (-1) * b.num;
    return rationalAdd(a, b);
  }

/*******************************fieldSub********************
****************/
  union field fieldSub(union field a, union field b){ // the
order of a and b matters
    rational dummyRatA, dummyRatB;
    long dummyMod3A, dummyMod3B;

    // this function finds the field we are working over and
returns the field
    // element a minus field element b

    if(fieldType == RATIONAL){
      dummyRatA.num = a.F_rat.num;
      dummyRatA.den = a.F_rat.den;
      dummyRatB.num = b.F_rat.num;
      dummyRatB.den = b.F_rat.den;
      a.F_rat = rationalSub(dummyRatA, dummyRatB);
      return a;
    }
    else{
```



```
      dummyMod3A = a.F_mod3;
      dummyMod3B = b.F_mod3;
      a.F_mod3 = mod3Sub(dummyMod3A, dummyMod3B);
      return a;
    }
  }

/*******************************mod3Mult********************
****************/
  long mod3Mult(long a, long b){

    // this function returns the product of the two mod 3
elements

    return longToMod3(a * b);
  }

/*******************************rationalMult****************
****************/
  rational rationalMult(rational a, rational b){
    rational x;
    long ratDen, num1, num2;

    // this function returns the CANONICAL form of the
product of the two
    // rational numbers

    // to formulate the product, we perform the same
calculations several times
    // so we calculate them at the beginning to reduce
complexity

    num1 = a.num * b.num; // this line is the only difference
to addition
    num2 = a.den * b.den;
    ratDen = modgcd(mod(num1), num2);

    x.num = (num1 / ratDen);
    x.den = (num2 / ratDen);

    return rationalise(x);
  }

/*******************************fieldMult*******************
****************/
  union field fieldMult(union field a, union field b){
    rational dummyRatA, dummyRatB;
    long dummyMod3A, dummyMod3B;

    // this function finds the field we are working over and
returns the product
    // of the two arbitrary field elements

    if(fieldType == RATIONAL){
```



```
      dummyRatA.num = a.F_rat.num;
      dummyRatA.den = a.F_rat.den;
      dummyRatB.num = b.F_rat.num;
      dummyRatB.den = b.F_rat.den;
      a.F_rat = rationalMult(dummyRatA, dummyRatB);
    }
    else{
      dummyMod3A = a.F_mod3;
      dummyMod3B = b.F_mod3;
      a.F_mod3 = mod3Mult(dummyMod3A, dummyMod3B);
    }
    return a;
  }

/*******************************mod3Div*********************
****************/
  long mod3Div(long a, long b){

    // this function returns mod 3 element a over mod 3
element b

    a = longToMod3(a);
    b = longToMod3(b);

    if (b == 0){
      return 0;
    }
    else if (b == 1){
      return a;
    }
    else{
      if (a == 1){
     return 2;
      }
      else if (a == 2){
     return 1;
      }
      else{
     return 0;
      }
    }
  }

/*******************************rationalDiv*****************
****************/
  rational rationalDiv(rational a, rational b){
    rational x;

    // this function returns the CANONICAL form of rational
number a over
    // rational number b

    // one very complexity friendly property of rational
numbers is that
```



```
    // division is multiplication of switched components

    x.num = b.den;
    x.den = b.num;

    return rationalMult(a, x);
  }

/*******************************fieldDiv********************
****************/
  union field fieldDiv(union field a, union field b){
    rational dummyRatA, dummyRatB;
    long dummyMod3A, dummyMod3B;

    // this function finds the field we are working over and
returns the field
    // element a over field element b

    if(fieldType == RATIONAL){
      dummyRatA.num = a.F_rat.num;
      dummyRatA.den = a.F_rat.den;
      dummyRatB.num = b.F_rat.num;
      dummyRatB.den = b.F_rat.den;
      a.F_rat = rationalDiv(dummyRatA, dummyRatB);
      return a;
    }
    else{
      dummyMod3A = a.F_mod3;
      dummyMod3B = b.F_mod3;
      a.F_mod3 = mod3Div(dummyMod3A, dummyMod3B);
      return a;
    }
  }

/*******************************zeroSet*********************
****************/
  union field zeroSet(void){
    union field a;

    // this function sets all elements of the field to zero

    a.F_rat.num = 0;
    a.F_rat.den = 1;
    rationalise(a.F_rat);
    a.F_mod3 = 0;
    return a;
  }

/*******************************zeroCheck*******************
****************/
  int zeroCheck(union field a){

    // this function returns 1 if a field element is zero
else 0
```

```
    if (fieldType == RATIONAL){
      if(a.F_rat.num == 0){
     return 1;
      }
    }
    if (fieldType == MOD3){
      if(longToMod3(a.F_mod3) == 0){
     return 1;
      }
    }
    return 0;
  }

/*********************************complexity*****************
**********************/
  int complexity(void){
    int c;

    // this function asks the user if they would like to see
a complexity review
    // and returns 1 if they do else 0

    c = 2;
    printf("\nDo you want to see the complexity review?:\n");
    printf("                              0 - no, just show me
the answer\n");
    printf("                              1 - yes, show me the
complexity review\n");
    while(c != 0 && c != 1){
      printf("Choice: ");
      scanf("%i",&c);
      if (c != 0 && c != 1){
     printf("\nInvalid choice: please choose 0 or 1\n\n");
      }
    }
    return c;
  }

// Now for intelligence

/*
** Intelligence is an array of integers corresponding to a
matrix. Matrices are three
** dimensional in this system, the kth element of the
intelligence array corresponds
** to the kth dimension of a matrix. Each of these integers
is initially 0, when a
** dimension is analysed the integer becomes positive and
corresponds to a particular
** matrix status (i.e. diagonal matrix). This status is then
used in algorithms to make
** decisions as to how to continue. The point of intelligence
is to reduce complexity.
```



```
** The intelligence array is stored as "status" in the struct
"matrix"
*/

/*********************************setIntelligence************
**********************/
  void setIntelligence(intelligenceP I, int dim, int status){

    // this function sets the element of the intelligence
array corresponding to
    // dimension "dim" to the integer "status".

    I->m[dim] = status;
  }

/*********************************getIntelligence************
**********************/
  int getIntelligence(matrixP M, int dim){

    // this function returns the element of the intelligence
array corresponding to
    // dimension "dim".

    return M->status->m[dim];
  }

/*********************************newIntelligence************
**********************/
  intelligenceP newIntelligence(matrixP M){
    int d;
    intelligenceP I;
    int* tmp;

    // this function allocates memory for the pointer to an
intelligence array, namely
    // "intelligenceP" and then for the integer elements of
intelligence. It initialises
    // an intelligence array

    I = (intelligenceP )calloc(1, sizeof(intelligence));

    // the most dimensions that will be needed for a matrix
is the maximum of its number
    // of rows and columns, therefore we allocate this many
spaces for integers in the
    // intelligence array

    I->dim = maxInt(M->row, M->col);
    I->m = (int *)calloc(I->dim, sizeof(int));

    // initially the status of each dimension in the
intelligence array is set to zero

    for(d = 0; d < I->dim; d++){
```



```
      setIntelligence(I, d, 0);
    }
    return I;
  }

/*********************************freeIntelligence************
**********************/
  void freeIntelligence(intelligenceP I){

    // this function frees the memory allocated for the
intelligence array (and the
    // pointer to it)

    free(I->m);
    free(I); // in this order to avoid memory leak
  }

//Now for matrices

/*
** Matrices are stored in three dimensions in this system.
The zero dimension coresponds
** to the initial matrix entered by the user. The other
dimensions can be used for anything.
** The main reason for the multi dimensions is for
calculating different stages of matrix
** functions and been able to apply our intelligence to those
stages independent of the other
** stages. The intelligence array is stored as part of the
struct of a matrix namely,
** "status". The determinant (a field element) is also stored
as part of the struct namely,
** "det". Initially all field elements of the matrix
(including the determinant) are set to an
** UNCANONICAL form of their field. This is the only time
there is an UNCANONICAL representation
** stored in this system. Now we can check if elements have
yet to be set. It is not pratical
** (or possible) to pass arrays representing our matrices
around instead we pass around a pointer
** to the matrices namely, "matrixP".
*/

/*********************************setElement*****************
****************/
  void setElement(matrixP M, int row, int col, int dim, union
field value){

    // this function sets the ("row","col") element of matrix
"M" in dimension "dim"
    // equal to field element "value"

    M->m[(dim * M->row * M->col) + (row * M->col) + col] =
value;
```



```
  }

/*********************************getElement*****************
****************/
  union field getElement(matrixP M, int row, int col, int
dim){

    // this function returns the ("row","col") element of
matrix "M" in dimension "dim"

    return M->m[(dim * M->row * M->col) + (row * M->col) +
col];
  }

/*********************************matrixSet******************
*********/
  matrixP matrixSet(matrixP M, matrixP N, int dim){
    int i, j;

    // this function sets matrix M equal to matrix N

    if(M->row != N->row || M->col != N->col){
      printf("\nincompatible dimensions for setting a new
matrix\n");
      return M;
    }

    for(i = 0; i < M->row; i++){
      for(j = 0; j < M->col; j++){
     setElement(M, i, j, dim, getElement(N, i, j, dim));
      }
    }

    M->det = N->det;
    setIntelligence(M->status, dim, getIntelligence(N, dim));
  }

/*********************************newMatrix******************
*****************/
  matrixP newMatrix(int row, int col){
    int i, j, k;
    union field dummy;

    // this function allocates memory for the pointer to a
matrix array, namely
    // "matrixP" and then for the field elements of matrix.
It allocates memory
    // for the intelligence array "status" via the function
"newIntelligence".
    // It initialises an matrix array

    matrixP M = (matrixP )calloc(1, sizeof(matrix));
```

```
    // "dummy" represents the UNCANONICAL form of field
elements

    if (fieldType == RATIONAL){
      dummy.F_rat.num = 0;
      dummy.F_rat.den = -1;
    }
    else{
      dummy.F_mod3 = 3;
    }

    M->row = row;
    M->col = col;
    M->dim = 0;
    M->det = dummy;
    M->status = newIntelligence(M);

    M->m = (union field *)calloc(row * col * maxInt(row,
col), sizeof(union field));

    // initially set each matrix element equal to "dummy"

    for(i = 0; i < row; i++){
      for(j = 0; j < col; j++){
     for (k = 0; k < maxInt(row, col); k++){
       setElement(M, i, j, k, dummy);
     }
      }
    }
    return M;
  }

/*******************************freeMatrix*****************
****************/
  void freeMatrix(matrixP M){

    // this function frees the memory allocated for the
matrix elements, the
    // pointer to them and the intelligence array

    freeIntelligence(M->status);
    free(M->m);
    free(M); // in this order to avoid memory leak
  }

/*******************************monoMatrix*****************
****************/
  matrixP monoMatrix(matrixP M, int dim, union field value){
    int i, j;

    // this function returns a matrixP of parameter
dimensions with all entries
    // equal to "value"
```



```
    for(i = 0; i < M->row; i++){
      for(j = 0; j < M->col; j++){
        setElement(M, i, j, dim, value);
      }
    }
    return M;
  }

/*******************************inputMatrix****************
****************/
  matrixP inputMatrix(matrixP M){
    int i, j;

    // this function reads user inputs, stores them in the
matrix and returns the matrixP

    for(i = 0; i < M->row; i++){
      printf("\nrow %i: \n\n", i + 1); // i+1 as entries are
stored from 0
       for(j = 0; j < M->col; j++){
     printf("entry(%i,%i): ", i + 1, j + 1);
     setElement(M, i, j, 0, fieldInput());
       }
    }
    printf("\n");
    return M;
  }

/*******************************inputDiagMatrix************
********************/
  matrixP inputDiagMatrix(matrixP M){
    int i;

    // this function does as input matrix but only asks for
diagonal enteries

    M = monoMatrix(M, 0, zeroSet());
    for(i = 0; i < M->row; i++){
      printf("\nrow %i: \n\n", i + 1); // i+1 as entries are
stored from 0
       printf("entry(%i,%i): ", i + 1, i + 1);
       setElement(M, i, i, 0, fieldInput());
    }
    printf("\n");
    return M;
  }

/*******************************inputLowerTriMatrix*********
************************/
  matrixP inputLowerTriMatrix(matrixP M){
    int i, j;

    // this function does as input matrix but only asks for
lower triangular enteries
```



```
    M = monoMatrix(M, 0, zeroSet());

    for (i = 0; i < M->row; i++) {
      printf("\nrow %i: \n\n", i + 1); // i+1 as entries are
stored from 0
       for (j = 0; j < i + 1; j++) {
     printf("entry(%i,%i): ", i + 1, j + 1);
     setElement(M, i, j, 0, fieldInput());
       }
    }

    printf("\n");
    return M;
  }

/*******************************inputUpperTriMatrix*********
************************/
  matrixP inputUpperTriMatrix(matrixP M){
    int i, j;

    // this function does as input matrix but only asks for
upper triangular enteries

    M = monoMatrix(M, 0, zeroSet());

    for (i = 0; i < M->col; i++) {
      printf("\nrow %i: \n\n", i + 1); // i+1 as entries are
stored from 0
       for (j = i; j < M->row; j++) {
     printf("entry(%i,%i): ", i + 1, j + 1);
     setElement(M, i, j, 0, fieldInput());
       }
    }

    printf("\n");
    return M;
  }

/*******************************displayElement*************
****************/
  void displayElement(matrixP M, int row, int col, int dim){

    // this function displays field entry ("row","col") of
matrix "M" in dimension "dim"

    displayField(getElement(M, row, col, dim));
  }

/*******************************displayMatrix**************
****************/
  void displayMatrix(matrixP M, int dim){
    int i, j;
```



```
    // this function displays matrix "M" in dimension "dim"

    for(i = 0; i < M->row; i++){
      printf("(");
      for(j = 0; j < (M->col) - 1; j++){
     displayElement(M, i, j, dim);
     printf(", ");
       }
       displayElement(M, i,(M->col) - 1, dim);
       printf(")\n");
    }
  }

/*******************************createMatrix***************
****************/
  matrixP createMatrix(int s, int row, int col){
    matrixP M;
    int i, j;

    // this function creates a ("row"*"col") matrix of type
"s", then returns the matrixP

    // this function is told the prefered type of matrix
wanted "s"
    // this function is told if their are any necessary pre
defined dimensions ("row" and "col")
    // it sends the matrixP to the appropriate input matrix
function with the correct dimensions

    i = 0;
    j = 0;

    if(s == 5){
      if(row == 0){
     while(i <= 0){
       printf("\nnumber of row(s): ");
       scanf("%i", &i);
     }
      }
      else{
     i = row;
     printf("\nnumber of row(s): %i", i);
      }
      if(col == 0){
     while(j <= 0){
       printf("\nnumber of column(s): ");
       scanf("%i", &j);
     }
      }
      else{
     j = col;
     printf("\nnumber of columns(s): %i\n", j);
      }
    }
```

```
    if(s != 5){
      row = maxInt(row, col);
      col = row;
      if(row == 0){
     while(i <= 0){
       printf("\nnumber of row(s): ");
       scanf("%i", &i);
     }
     j = i;
     printf("\nnumber of column(s): %i\n", j);
      }
      else{
     i = row;
     j = col;
     printf("\nnumber of rows(s): %i", i);
     printf("\nnumber of columns(s): %i\n", j);
      }
    }

    if(s == 1 || s == 5){
      M = inputMatrix(newMatrix(i, j));
    }
    if(s == 2){
      M = inputDiagMatrix(newMatrix(i, j));
    }
    if(s == 3){
      M = inputLowerTriMatrix(newMatrix(i, j));
    }
    if(s == 4){
      M = inputUpperTriMatrix(newMatrix(i, j));
    }

    displayMatrix(M, 0);
    printf("\n");
    return M;
  }

/*******************************matrixElementSwap***********
****************/
  matrixP matrixElementSwap(matrixP M, int i, int j, int y,
int z, int dim){
    union field dummy;

    // this function swaps two elements of a matrix in the
same dimension, then
    // returns the matrixP

    dummy = getElement(M, i, j, dim);
    setElement(M, i, j, dim, getElement(M, y, z, dim));
    setElement(M, y, z, dim, dummy);

    return M;
  }
```



```
/*******************************matrixRowMult**************
****************/
  matrixP matrixRowMult(matrixP M, int row, int dim, union
field value){
    int i;

    // this function multiplies row "row" by a field element
"value" in dimension
    // "dim" and returns the matrixP

    for(i = 0; i < M->col; i++){
      setElement(M, row, i, dim, fieldMult(value,
getElement(M, row, i, dim)));
    }
    return M;
  }

/*******************************matrixColMult**************
****************/
  matrixP matrixColMult(matrixP M, int col, int dim, union
field value){
    int i;

    // this function multiplies column "col" by a field
element "value" in dimension
    // "dim" and returns the matrixP

    for(i = 0; i < M->row; i++){
      setElement(M, i, col, dim, fieldMult(value,
getElement(M, i, col, dim)));
    }
    return M;
  }


/*******************************matrixRowAdd***************
****************/
  matrixP matrixRowAdd(matrixP M, int row1, int row2, int
dim, union field value){
    int i;

    // this function sets row "row1" equal to the sum of
"row1" and "row2" in
    // dimension "dim" and returns the matrixP

    for(i = 0; i < M->col; i++){
      setElement(M, row1, i, dim, fieldAdd(getElement(M,
row1, i, dim), fieldMult(value, getElement(M, row2, i,
dim))));
    }
    return M;
  }
```



```
/*******************************matrixColAdd***************
****************/
  matrixP matrixColAdd(matrixP M, int col1, int col2, int
dim, union field value){
    int i;

    // this function sets column "col1" equal to the sum of
"col1" and "col2" in
    // dimension "dim" and returns the matrixP

    for(i = 0; i < M->row; i++){
      setElement(M, i, col1, dim, fieldAdd(getElement(M, i,
col1, dim), fieldMult(value, getElement(M, i, col2, dim))));
    }
    return M;
  }

/*******************************matrixRowSwap**************
****************/
  matrixP matrixRowSwap(matrixP M, int a, int b, int dim){
    int i;

    // this function swaps rows "a" and "b" then returns the
matrixP

    for(i = 0; i < M->col; i++){
      M = matrixElementSwap(M, a, i, b, i, dim);
    }
    return M;
  }

/*******************************matrixColSwap**************
****************/
  matrixP matrixColSwap(matrixP M, int a, int b, int dim){
    int i;

    // this function swaps columns "a" and "b" then returns
the matrixP

    for(i = 0; i < M->row; i++){
      M = matrixElementSwap(M, i, a, i, b, dim);
    }

    return M;
  }

/*******************************findFirstNonZeroInRow******
**********************/
  int findFirstNonZeroInRow(matrixP M, int row, int dim){
    int j;

    // this function returns the column number of the first
zero entry in row "row"
    // then returns the matrixP
```



```
    for(j = 0; j < M->col; j++){
      if(!zeroCheck(getElement(M, row, j, dim))){
     return j;
      }
    }
    return M->col;
  }

// properties checks

/*
** The following functions check for particular properties of
matrices. They all
** return 1 if the property is true, else 0. They will be
used to determine the
** status of dimensions in intelligence.
*/

/*******************************matrixZeroRowCheck*********
****************/
 int matrixZeroRowCheck(matrixP M, int row, int dim){
   int i;

   // this function checks if row "row" is a row of zeros

   for(i = 0 + dim; i < M->col; i++){
     if(zeroCheck(getElement(M, row, i, dim)) == 0){
       return 0;
     }
   }
   return 1;
 }

/*******************************matrixZeroColCheck*********
****************/
 int matrixZeroColCheck(matrixP M, int col, int dim){
   int i;

   // this function checks if column "col" is a column of
zeros

   for(i = 0 + dim; i < M->row; i++){
     if(zeroCheck(getElement(M, i, col, dim)) == 0){
       return 0;
     }
   }
   return 1;
 }

/*******************************matrixIdenticalRowCheck****
****************/
  int matrixIdenticalRowCheck(matrixP M, int row, int dim){
    int i, j;
```

```
    // this function checks if there are any rows identical
to row "row"
    // (not including itself!)

    for(i = 0; i < M->row; i++){
      for(j = 0; j < M->col; j++){
     if(row != i){
       if(fieldCompare(getElement(M, row, j, dim),
getElement(M, i, j, dim)) != 0){
         return 0;
       }
     }
      }
    }
    return 1;
  }

/*******************************matrixIdenticalColCheck*****
****************/
  int matrixIdenticalColCheck(matrixP M, int col, int dim){
    int i, j;

    // this function checks if there are any columns
identical to column "col"
    // (not including itself!)

    for(j = 0; j < M->col; j++){
      for(i = 0; i < M->row; i++){
     if(col != j){
       if(fieldCompare(getElement(M, i, col,
dim),getElement(M, i, j, dim)) !=0){
         return 0;
       }
     }
      }
    }
    return 1;
  }

/*******************************matrixZeroDiagCheck*********
****************/
  int matrixZeroDiagCheck(matrixP M, int dim){
    int i;

    // this function checks if there is a zero on the
diagonal

    for(i = 0 + dim; i < M->row; i++){
      if(zeroCheck(getElement(M, i, i, dim)) == 1) {
     return 1;
      }
    }
    return 0;
```



```
  }

/*******************************matrixLowerTriCheck*********
****************/
  int matrixLowerTriCheck(matrixP M, int dim){
    int i,j;

    // this function checks if matrix "M" is lower triangular

    for(i = dim; i < M->row - 1; i++){
      for(j = i + 1; j < M->row; j++){
     if (zeroCheck(getElement(M, i, j, dim)) == 0) {
       return 0;
     }
      }
    }
    return 1;
  }

/*******************************matrixUpperTriCheck*********
****************/
  int matrixUpperTriCheck(matrixP M, int dim){
    int i,j;

    // this function checks if matrix "M" is upper triangular

    for(i = 1 + dim; i < (M->row); i++){
      for(j = dim; j < i ; j++){
     if (zeroCheck(getElement(M, i, j, dim)) == 0) {
       return 0;
     }
      }
    }
    return 1;
  }

/*******************************statusSet******************
**************/
  void statusSet(matrixP M, int dim){
    int i, j, status, check1, check2;

    // this function asigns the appropriate value to the
intelligence array
    // corresponding to dimension "dim" of matrix "M". It
uses the fact that every
    // natural number is either prime or a unique product of
primes. Every time a
    // particular property is noticed the status is
multiplied by a particular
    // prime number. See function "statusAnalyse" to see the
unique properties.

    status = 1;
    check1 = 0;
```



```
    check2 = 1;

    // for our purposes it is easier to not know anything
more about rectangular matrices:

    if(M->row != M->col){
      setIntelligence(M->status, dim, 2);
      return;
    }

    // Test for triangular properties

    if(status%2 != 0){
      for(i = 0; i < 1; i++){
        if(matrixLowerTriCheck(M, dim) == 1){
       status *= 3;
       check1 = 1;
        }
     if(matrixUpperTriCheck(M, dim) == 1){
       status *= 5;
       check1 = 1;
        }
     break;
      }
    }

    // Test to check if we can spot the determinant is zero

    for(j = 0; j < 1; j++){
      for(i = 0 + dim; i < M->row; i++){
        if (matrixZeroRowCheck(M, i, dim) == 1) {
          status *= 7;
       check2 = 0;
          break;
        }
      }
      for(i = 0 + dim; i < M->col; i++){
        if (matrixZeroColCheck(M, i, dim) == 1) {
          status *= 11;
       check2 = 0;
          break;
        }
      }
      for(i = 0 + dim; i < M->row; i++){
        if (matrixIdenticalRowCheck(M, i, dim) == 1) {
          status *= 13;
       check2 = 0;
          break;
        }
      }
      for(i = 0 + dim; i < M->col; i++){
        if (matrixIdenticalColCheck(M, i, dim) == 1) {
          status *= 17;
       check2 = 0;
```



```
          break;
        }
      }
    }

    // If their is a row or column of zeros there is
inevitably a zero on the diagonal.
    // We only want to check if there is a zero on the
diagonal when there is not a row
    // or column of zeros.
    // Also we are only interested in the diagonal entries of
triangular matrices. If
    // a matrix is triangular and has 2 identical rows or
columns then there is inevitably
    // a zero on the diagonal. In this case we do not want to
check for this property again

    if(check1 && check2){
      if(matrixZeroDiagCheck(M, dim) == 1){
        status *= 19;
      }
    }

    setIntelligence(M->status, dim, status);
  }

/*****************************************statusAnalyse*****
****************/
  void statusAnalyse(matrixP M, int dim){

    // this function outputs a description of the matrix type
of "M" in dimension "dim"

    if(getIntelligence(M, dim) == 0){
      statusSet(M, dim);
    }

    // no special properties

    if(getIntelligence(M, dim) == 1){
      printf("\nMatrix is square, no detected special
properties\n");
    }
    if(getIntelligence(M, dim) == 2){
      printf("\nMatrix is rectangular, no detected special
properties\n");
    }

    // triangular properties

    if(getIntelligence(M, dim) == 3){
      printf("\nMatrix is square and lower triangular\n");
    }
    if(getIntelligence(M, dim) == 5){
```

```
      printf("\nMatrix is square and upper triangular\n");
    }
    if(getIntelligence(M, dim) == 15){
      printf("\nMatrix is square and diagonal\n");
    }

    // square and zero det properties

    if(getIntelligence(M, dim) == 7){
      printf("\nMatrix is square and has a row of zeros\n");
    }
    if(getIntelligence(M, dim) == 11){
      printf("\nMatrix is square and has a column of
zeros\n");
    }
    if(getIntelligence(M, dim) == 13){
      printf("\nMatrix is square and has at least 2 identical
rows\n");
    }
    if(getIntelligence(M, dim) == 17){
      printf("\nMatrix is square and has at least 2 identical
columns\n");
    }
    if(getIntelligence(M, dim) == 19){
      printf("\nMatrix is square and has a zero on its
diagonal\n");
    }

    // triangular and zero det properties
    // all the following circumstances imply there is a zero
diagonal entry

    if(getIntelligence(M, dim) == 21){
      printf("\nMatrix is square, lower triangular and has a
row of zeros\n");
    }
    if(getIntelligence(M, dim) == 35){
      printf("\nMatrix is square, upper triangular and has a
row of zeros\n");
    }
    if(getIntelligence(M, dim) == 105){
      printf("\nMatrix is square, diagonal and has a row of
zeros\n");
    }
    if(getIntelligence(M, dim) == 33){
      printf("\nMatrix is square, lower triangular and has a
column of zeros\n");
    }
    if(getIntelligence(M, dim) == 55){
      printf("\nMatrix is square, upper triangular and has a
column of zeros\n");
    }
    if(getIntelligence(M, dim) == 165){
```



```
      printf("\nMatrix is square, diagonal and has a column
of zeros\n");
    }
    if(getIntelligence(M, dim) == 39){
      printf("\nMatrix is square, lower triangular and has at
least 2 identical rows\n");
    }
    if(getIntelligence(M, dim) == 65){
      printf("\nMatrix is square, upper triangular and has at
least 2 identical rows\n");
    }
    if(getIntelligence(M, dim) == 195){
      printf("\nMatrix is square, diagonal and at least 2
identical rows\n");
    }
    if(getIntelligence(M, dim) == 51){
      printf("\nMatrix is square, lower triangular and has at
least 2 identical columns\n");
    }
    if(getIntelligence(M, dim) == 85){
      printf("\nMatrix is square, upper triangular and has at
least 2 identical columns\n");
    }
    if(getIntelligence(M, dim) == 255){
      printf("\nMatrix is square, diagonal and has at least 2
identical columns\n");
    }
    if(getIntelligence(M, dim) == 57){
      printf("\nMatrix is square, lower triangular and has a
zero on the diagonal\n");
    }
    if(getIntelligence(M, dim) == 95){
      printf("\nMatrix is square, upper triangular and has a
zero on the diagonal\n");
    }
    if(getIntelligence(M, dim) == 285){
      printf("\nMatrix is square, diagonal and has a zero on
the diagonal\n");
    }
    return;
  }

/*********************************matrixConstMult*************
****************/
  matrixP matrixConstMult(matrixP M, int dimM, union field
value){
    int i, j;

    // this function multiplies matrix "M" by field element
"c", then returns the matrixP

    for(i = 0; i < M->row; i++){
      for(j = 0; j < M->col; j++){
```



```
     setElement(M, i, j, dimM, fieldMult(getElement(M, i, j,
dimM), value));
      }
    }
    return M;
  }

/*********************************matrixElementWiseMult******
********************/
matrixP matrixElementWiseMult(matrixP M, matrixP N, int dim){
    int i, j;
    matrixP X;

    // this function calculates the element wise product of
matrices "M" and "N" in dimension
    // "dim", then returns the matrixP

    if(M->row != N->row || M->col != N->col){
      printf("\nincompatible dimensions for element wise
multiplication\n");
      return M;
    }

    X = newMatrix(M->row, M->col);
    for(i = 0; i < M->row; i++){
      for(j = 0; j < M->col; j++){
     setElement(X, i, j, dim, fieldMult(getElement(M, i, j,
dim), getElement(N, i, j, dim)));
      }
    }
    return X;
}

/*********************************matrixDirectSum************
**************/
matrixP matrixDirectSum(matrixP M, matrixP N, int dim){//the
order of M and N matters
    int i, j;
    matrixP X;

    // this function calculates the direct sum of matrices
"M" and "N" in dimension
    // "dim", then returns the matrixP

    X = newMatrix(M->row + N->row, M->col + N->col);
    monoMatrix(X, dim, zeroSet());
    for(i = 0; i < M->row; i++){
      for(j = 0; j < M->col; j++){
     setElement(X, i, j, dim, getElement(M, i, j, dim));
      }
    }
    for(i = M->row; i < X->row; i++){
      for(j = M->col; j < X->col; j++){
```



```
     setElement(X, i, j, dim, getElement(N, i - M->row, j -
M->col, dim));
      }
    }
    return X;
}

// The following diagonal matrix arithmetic functions only
work with the diagonal entries
// of the matrices. This significantly reduces complexity.
These algorithms can be thought
// of as sparse matrix algorithms

/*********************************diagAdd*********************
**************/
  matrixP diagAdd(matrixP M, matrixP N, int dim){
    int i;
    matrixP X;

    // this function calculates the sum of diagonal matrices
"M" and "N" in dimension
    // "dim", then returns the matrixP

    X = monoMatrix(newMatrix(M->row, M->col), dim,
zeroSet());
    for(i = 0; i < M->row; i++){
      setElement(X, i, i, dim, fieldAdd(getElement(M, i, i,
dim),getElement(N, i, i, dim)));
    }
    return X;
  }

/*********************************diagSub*********************
***************/
  matrixP diagSub(matrixP M, matrixP N, int dim){
    union field c;

    // this function calculates diagonal matrix "M" minus
diagonal matrix "N" in
    // dimension "dim", then returns the matrixP

    if(fieldType == RATIONAL){
      c.F_rat.num = -1;
      c.F_rat.den = 1;
    }
    else{
      c.F_mod3 = 2; // equivalent to longToMod3(-1);
    }
    return diagAdd(M, matrixConstMult(N, dim, c), dim);
  }

/*********************************diagMult********************
***************/
  matrixP diagMult(matrixP M, matrixP N, int dim){
```

```
    int i;
    matrixP X;

    // this function calculates the product of diagonal
matrices "M" and "N" in dimension
    // "dim", then returns the matrixP

    X = monoMatrix(newMatrix(M->row, M->col), dim,
zeroSet());
    for(i = 0; i < M->row; i++){
      setElement(X, i, i, dim, fieldMult(getElement(M, i, i,
dim),getElement(N, i, i, dim)));
    }
    return X;
  }

// The following triangular matrix arithmetic functions only
work with the non zero entries
// of the matrices. This reduces complexity. These algorithms
can be thought of as sparse
//  matrix algorithms

/*******************************lowerTriAdd*****************
***************/
  matrixP lowerTriAdd(matrixP M, matrixP N, int dim){
    int i, j;
    matrixP X;

    // this function calculates the sum of lower triangular
matrices "M" and "N" in dimension
    // "dim", then returns the matrixP

    X = monoMatrix(newMatrix(M->row, M->col), dim,
zeroSet());
    for (i = 0; i < M->row; i++) {
      for (j = 0; j < i + 1; j++) {
        setElement(X, i, j, 0, fieldAdd(getElement(M, i, j,
dim), getElement(N, i, j, dim)));
      }
    }
    return X;
  }

/*******************************upperTriAdd*****************
**************/
  matrixP upperTriAdd(matrixP M, matrixP N, int dim){
    int i, j, k;
    matrixP X;

    // this function calculates the sum of upper triangular
matrices "M" and "N" in dimension
    // "dim", then returns the matrixP
```



```
    X = monoMatrix(newMatrix(M->row, M->col), dim,
zeroSet());
    for (j = 0; j < M->row; j++) {
      for (i = 0; i < j + 1; i++) {
        setElement(X, i, j, dim, fieldAdd(getElement(M, i, j,
dim), getElement(N, i, j, dim)));
      }
    }
    return X;
  }

/*******************************lowerTriSub*****************
***************/
  matrixP lowerTriSub(matrixP M, matrixP N, int dim){
    int i, j;
    union field c;

    // this function calculates lower triangular matrix "M"
minus lower trianuglar matrix "N"
    // in dimension "dim", then returns the matrixP

    if(fieldType == RATIONAL){
      c.F_rat.num = -1;
      c.F_rat.den = 1;
    }
    else{
      c.F_mod3 = 2; // equivalent to longToMod3(-1);
    }
    return lowerTriAdd(M, matrixConstMult(N, dim, c), dim);
  }

/*******************************upperTriSub*****************
***************/
  matrixP upperTriSub(matrixP M, matrixP N, int dim){
    int i, j;
    union field c;

    // this function calculates upper triangular matrix "M"
minus lower trianuglar matrix "N"
    // in dimension "dim", then returns the matrixP

    if(fieldType == RATIONAL){
      c.F_rat.num = -1;
      c.F_rat.den = 1;
    }
    else{
      c.F_mod3 = 2; // equivalent to longToMod3(-1);
    }
    return upperTriAdd(M,  matrixConstMult(N, dim, c), dim);
  }

/*******************************lowerTriMult****************
***************/
  matrixP lowerTriMult(matrixP M, matrixP N, int dim){
```



```
    int i, j, k;
    matrixP X;
    union field temp, accumulate;

    // this function calculates the product of lower
triangular matrices "M" and
    //"N" in dimension "dim", then returns the matrixP

    X = monoMatrix(newMatrix(M->row, M->col), dim,
zeroSet());
    for (i = 0; i < M->row; i++) {
      for (j = 0; j < i + 1; j++) {
        for (k = 0; k < mod(i - j) + 1; k++) {
          temp = fieldMult(getElement(M, i, i - k, dim),
getElement(N, i - k, j, dim));
          accumulate = fieldAdd(temp, accumulate);
        }
        setElement(X, i, j, dim, accumulate);
        accumulate = zeroSet();
      }
    }
    return X;
  }

/*******************************upperTriMult****************
***************/
  matrixP upperTriMult(matrixP M, matrixP N, int dim){
    int i, j, k;
    matrixP X;
    union field temp, accumulate;

    // this function calculates the product of upper
triangular matrices "M" and
    //"N" in dimension "dim", then returns the matrixP

    X = monoMatrix(newMatrix(M->row, M->col), dim,
zeroSet());
    for (j = 0; j < M->row; j++) {
      for (i = 0; i < j + 1; i++) {
     for (k = 0; k < mod(i - j) + 1; k++) {
       temp = fieldMult(getElement(M, i, j - k, dim),
getElement(N, j - k, j, dim));
       accumulate = fieldAdd(temp, accumulate);
     }
     setElement(X, i, j, dim, accumulate);
     accumulate = zeroSet();
      }
    }
    return X;
  }

// The following matrix arithmetic functions are for
arbitrary matrices, including
```



```
// rectanular matrices. The previous matrix arithmetic
functions have obviously all
// been for specific square matrices

/*******************************matrixAdd****************
****************/
  matrixP matrixAdd(matrixP M, matrixP N, int dim){
    matrixP X;
    int i, j;

    // this function calculates the sum of matrices "M" and
"N" in dimension "dim",
    // then returns the matrixP

    if ((M->row != N->row) || (M->col != N->col)){
      printf("incompatible dimensions for addition\n");
      return X = newMatrix(M->row, M->col);
    }

    X = newMatrix(M->row, M->col);
    for(i = 0; i < M->row; i++){
      for(j = 0; j < M->col; j++){
     setElement(X, i, j, dim, fieldAdd(getElement(M, i, j,
dim), getElement(N, i, j, dim)));
      }
    }
    return X;
  }

/*******************************matrixSub****************
****************/
  matrixP matrixSub(matrixP M, matrixP N, int dim){
    union field c;

    // this function calculates matrix "M" minus matrix "N"
in dimension "dim",
    // then returns the matrixP

    if(fieldType == RATIONAL){
      c.F_rat.num = -1;
      c.F_rat.den = 1;
    }
    else{
      c.F_mod3 = 2; // equivalent to longToMod3(-1);
    }
    return matrixAdd(M, matrixConstMult(N, dim, c), dim);
  }

/*******************************matrixMult****************
****************/
  matrixP matrixMult(matrixP M, matrixP N, int dim){
    matrixP X;
    int i, j, k;
    union field accumulate, temp;
```

```
    // this function calculates the product of matrices "M"
and "N" in dimension "dim",
    // then returns the matrixP

    if(M->col != N->row){
      printf("incompatible dimensions for multiplication\n");
      return X = newMatrix(M->row, M->col);
    }

    X = newMatrix(M->row, N->col);
    accumulate = zeroSet();

    for (i = 0; i < M->row; i++) {
      for (j = 0; j < N->col; j++) {
     for (k = 0; k < N->row; k++) {
       temp = fieldMult(getElement(M, i, k, dim),
getElement(N, k, j, dim));
       accumulate = fieldAdd(temp, accumulate);
     }
     setElement(X, i, j, dim, accumulate);
     accumulate = zeroSet();
      }
    }
    return X;
  }

// The following functions transform square matrices into
triangular forms using
// standard row operations. However they will transform a
"fat" matrix such that
// the square sub matrix which is the largest possible square
matrix on the left
// hand side is made triangular and the remaining sub matrix
have been transformed
// by the same standard row operations. This is necessary for
solving systems of
// linear equations, calculatin inverse matrices etc.

/*******************************lowerTriForm***************
****************/
  matrixP lowerTriForm(matrixP M, int dim){
    int i, j, k, skipCol, highest, dummy;
    union field ratio, minus;

    // this function transforms matrix "M" into lower
triangular form in dimension
    // "dim", then returns the matrixP

    if(fieldType == RATIONAL){
      minus.F_rat.num = -1;
      minus.F_rat.den = 1;
    }
    else{
```



```
      minus.F_mod3 = 2;
    }

    for(j = M->row - 1; j > 0; j--){ // j iterates through
the cols despite reference to rows
      skipCol = 0;
      for(i = j - 1; i >= 0; i--){
     if(zeroCheck(getElement(M, j, j, dim)) == 0){
       ratio = fieldDiv(fieldMult(minus, getElement(M, i, j,
dim)), getElement(M, j, j, dim));
       if(zeroCheck(getElement(M, i, j, dim)) == 1){
         skipCol == 1;
       }
     }
     else{

       highest = j;
       for(k = i; k >= 0; k--){
         if(fieldCompare(fieldMod(getElement(M, k, j,
dim)),fieldMod(getElement(M, highest, j, dim))) == 1){
           highest = k;
         }
         if(highest == j){
           skipCol = 1;
         }
         else{
           matrixRowSwap(M, j, highest, dim);
           if(zeroCheck(getElement(M, i, j, dim)) == 0){
          ratio = fieldDiv(fieldMult(minus, getElement(M, i,
j, dim)), getElement(M, j, j, dim));
           }
         }
       }
     }
     if(skipCol == 0){
       matrixRowAdd(M, i, j, dim, ratio);
     }
      }
    }
    return M;
  }

/*******************************upperTriForm***************
****************/
  matrixP upperTriForm(matrixP M, int dim){
    int i, j, k, skipCol, highest, dummy;
    union field ratio, minus;

    // this function transforms matrix "M" into upper
triangular form in dimension
    // "dim", then returns the matrixP

    if(fieldType == RATIONAL){
      minus.F_rat.num = -1;
```



```
      minus.F_rat.den = 1;
    }
    else{
      minus.F_mod3 = 2;
    }

    for(j = 0; j < M->col - 1; j++){
      skipCol = 0;
      for(i = j + 1; i < M->row; i++){
     if(zeroCheck(getElement(M, j, j, dim)) == 0){
       ratio = fieldDiv(fieldMult(minus, getElement(M, i, j,
dim)), getElement(M, j, j, dim));
       if(zeroCheck(getElement(M, i, j, dim)) == 1){
         skipCol == 1;
       }
     }
     else{
       highest = j;
       for(k = i; k < M->row; k++){
         if(fieldCompare(fieldMod(getElement(M, k, j,
dim)),fieldMod(getElement(M, highest, j, dim))) == 1){
           highest = k;
         }
         if(highest == j){
           skipCol = 1;
         }
         else{
           matrixRowSwap(M, j, highest, dim);
           if(zeroCheck(getElement(M, i, j, dim)) == 0){
          ratio = fieldDiv(fieldMult(minus, getElement(M, i,
j, dim)), getElement(M, j, j, dim));
           }
           else{
          skipCol == 1;
           }
         }
       }
     }
     if(skipCol == 0){
       matrixRowAdd(M, i, j, dim, ratio);
     }
      }
    }
    return M;
  }

/*******************************diagonalise****************
****************/
  matrixP diagonalise(matrixP M, int dim){

    // this function transforms matrix "M" into diagonal form
in dimension
    // "dim", then returns the matrixP
```



```
    return lowerTriForm(upperTriForm(M, dim), dim);
  }

// The so called "brain" functions in general make use of the
intelligence array and trys to
// save as much complexity as possible. They are the source
for the menu functions.

/*******************************brainCreateMatrix**********
*******************/
  matrixP brainCreateMatrix(int shape, int row, int col){
    int s;

    // this function creates a matrix with "row" rows and
"col" columns and returns the matrixP.
    // It will create the appropriate sparse matrix if asked
to. If specified (when s = 1) it
    // will only allow the user to create square matrices.

    s = 0;
    if(row != 0 && row == col){
      shape = 1;
    }

    if(row != 0 && col != 0 && row != col){
    return createMatrix(5, row, col);
    }

    printf("\nPlease choose which type of Matrix you wish to
create\n\n");
    printf("The available choices are:\n");
    printf("                              1 - arbitrary square
matrix\n");
    printf("                              2 - diagonal
matrix\n");
    printf("                              3 - lower triangular
matrix\n");
    printf("                              4 - upper triangular
matrix\n");
    if(shape == 0){
      printf("                                5 - rectangular
matrix\n\n");
       while(s != 1 && s != 2 && s != 3 && s != 4 && s != 5){
     printf("Choice: ");
     scanf("%i",&s);
     if (s != 1 && s != 2 && s != 3 && s != 4 && s != 5){
       printf("\nInvalid choice: please choose between 1 and
5\n\n");
     }
      }
    }
    else{
      while(s != 1 && s != 2 && s != 3 && s != 4){
     printf("Choice: ");
```

```
     scanf("%i",&s);
     if (s != 1 && s != 2 && s != 3 && s != 4){
       printf("\nInvalid choice: please choose between 1 and
4\n\n");
     }
      }
    }
    return createMatrix(s, row, col);
  }

/*******************************brainMatrixConstMult*******
**************/
  matrixP brainMatrixConstMult(matrixP M, union field c, int
view){
    matrixP X, DUMMY;

    // this function multpilies matrix "M" by a constant "c"
and returns the matrixP.
    // It uses intelligence to decide the optimum method for
the task

    // view != 0 implies the user wants to know how det was
calculated, else just calculate det

    DUMMY = newMatrix(M->row, M->col);
    matrixSet(DUMMY, M, 0);
    X = matrixConstMult(M, 0, c);
    if(view){
      printf("\nANSWER:\n\n");
      displayField(c);
      printf("\n\n*\n\n");
      displayMatrix(DUMMY, 0);
      printf("\n=\n\n");
      displayMatrix(X, 0);
    }
    freeMatrix(M);

    return X;
  }

/*******************************brainMatrixDirectSum*******
**************/
  matrixP brainMatrixDirectSum(matrixP M, matrixP N, int
view){
    matrixP X;

    // this function calculates the direct sum of matrices
"M" and "N" and returns the matrixP

    // view != 0 implies the user wants to know how det was
calculated, else just calculate det

    X = matrixDirectSum(M, N, 0);
```



```
    if(view){
      printf("\nANSWER:\n\n");
      displayMatrix(M, 0);
      printf("\n(+)\n\n");
      displayMatrix(N, 0);
      printf("\n=\n\n");
      displayMatrix(X, 0);
    }
    freeMatrix(M);
    freeMatrix(N);

    return X;
  }

/*******************************brainMatrixElementWiseMult*
********************/
  matrixP brainMatrixElementWiseMult(matrixP M, matrixP N,
int view){
    matrixP X;

    // this function calculates the element wise product of
matrices "M" and "N"

    // view != 0 implies the user wants to know how det was
calculated, else just calculate det

    X = matrixElementWiseMult(M, N, 0);

    if(view){
      printf("\nANSWER:\n\n");
      displayMatrix(M, 0);
      printf("\nx\n\n");
      displayMatrix(N, 0);
      printf("\n=\n\n");
      displayMatrix(X, 0);
    }
    freeMatrix(M);
    freeMatrix(N);

    return X;
  }

/*****************************************brainMatrixAdd****
**************************/
  matrixP brainMatrixAdd(matrixP M, matrixP N, int view){
    int ms, ns, complex;
    matrixP X;

    // this function calculates the sum of matrices "M" and
"N" and returns the matrixP. It uses
    // intelligence to decide the optimum method for the task

    // view != 0 implies the user wants to know how det was
calculated, else just calculate det
```



```
    X = newMatrix(M->row, M->col);

    if(view){
    complex = complexity();
    }
    else{
      complex = 0;
    }

    statusSet(M, 0);
    statusSet(N, 0);
    ms = getIntelligence(M, 0);
    ns = getIntelligence(N, 0);

    if((ms%15 == 0) && (ns%15 == 0)){
      if(complex){
     printf("\nOverall Complexity: (0, n, 2*n*(n - 1))\n");
      }
      X = diagAdd(M, N, 0);

      // since diagonal matrices are lower triangular, upper
triangular and the
      // prefered matrix to work over, we return the funcion
here if we can.

      if(view){
     printf("\nANSWER:\n\n");
     displayMatrix(M, 0);
     printf("\n+\n\n");
     displayMatrix(N, 0);
     printf("\n=\n\n");
     displayMatrix(X, 0);
      }
      freeMatrix(M);
      freeMatrix(N);

      return X;
    }
    else if((ms%3 == 0) && (ns%3 == 0)){
      if(complex){
     printf("\nOverall Complexity: (0, (½)*n^2 + (½)*n,
2*n*(n - 1))\n");
      }
      X = lowerTriAdd(M, N, 0);
    }
    else if((ms%5 == 0) && (ns%5 == 0)){
      if(complex){
     printf("\nOverall Complexity: (0, (½)*n^2 + (½)*n,
2*n*(n - 1))\n");
      }
      X = upperTriAdd(M, N, 0);
    }
    else{
```



```
      if(complex){
     printf("\nOverall Complexity: (0, n^2, 2*n*(n - 1))\n");
      }
      X = matrixAdd(M, N, 0);
    }
    if(view){
      printf("\nANSWER:\n\n");
      displayMatrix(M, 0);
      printf("\n+\n\n");
      displayMatrix(N, 0);
      printf("\n=\n\n");
      displayMatrix(X, 0);
    }
    freeMatrix(M);
    freeMatrix(N);

    return X;
  }

/*****************************************brainMatrixSub****
**************************/
  matrixP brainMatrixSub(matrixP M, matrixP N, int view){
    int ms, ns, complex;
    matrixP X;
    union field minus;

    // this function calculates matrix "M" minus matrix "N"
and returns the matrixP. It uses
    // intelligence to decide the optimum method for the task

    // view != 0 implies the user wants to know how det was
calculated, else just calculate det

     if(view){
    complex = complexity();
    }
    else{
      complex = 0;
    }

    if(fieldType == RATIONAL){
      minus.F_rat.num = -1;
      minus.F_rat.den = 1;
    }
    else{
      minus.F_mod3 = 2;
    }

    X = newMatrix(M->row, M->col);

    statusSet(M, 0);
    statusSet(N, 0);
    ms = getIntelligence(M, 0);
    ns = getIntelligence(N, 0);
```

```
    if((ms%15 == 0) && (ns%15 == 0)){
      if(complex == 1){
     printf("\nOverall Complexity: (0, n, 2*n*(n - 1))\n");
      }
      X = diagSub(M, N, 0);

      // since diagonal matrices are lower triangular, upper
triangular and the
      // prefered matrix to work over, we return the funcion
here if we can.

      if(view){
     printf("\nANSWER:\n\n");
     displayMatrix(M, 0);
     printf("\n-\n\n");
     displayMatrix(matrixConstMult(N, 0, minus), 0);
     printf("\n=\n\n");
     displayMatrix(X, 0);
      }
      freeMatrix(M);
      freeMatrix(N);

      return X;
    }
    else if((ms%3 == 0) && (ns%3 == 0)){
      if(complex == 1){
     printf("\nOverall Complexity: (0, (⅓)*n^2 + (⅓)*n,
2*n*(n - 1))\n");
      }
      X = lowerTriSub(M, N, 0);
    }
    else if((ms%5 == 0) && (ns%5 == 0)){
      if(complex == 1){
     printf("\nOverall Complexity: (0, (⅓)*n^2 + (⅓)*n,
2*n*(n - 1))\n");
      }
      X = upperTriSub(M, N, 0);
    }

    else{
      if(complex == 1){
     printf("\nOverall Complexity: (0, n^2, 2*n*(n - 1))\n");
      }
      X = matrixSub(M, N, 0);
    }
    if(view){
      printf("\nANSWER:\n\n");
      displayMatrix(M, 0);
      printf("\n-\n\n");
      displayMatrix(matrixConstMult(N, 0, minus), 0);
      printf("\n=\n\n");
      displayMatrix(X, 0);
    }
```



```
    freeMatrix(M);
    freeMatrix(N);

    return X;
  }

/*******************************brainMatrixMult************
*******************/
  matrixP brainMatrixMult(matrixP M, matrixP N, int view){
    int ms, ns, complex;
    matrixP X;

    // this function calculates the product of matrices "M"
and "N" and returns the matrixP.
    // It uses intelligence to decide the optimum method for
the task

    // view != 0 implies the user wants to know how det was
calculated, else just calculate det

    if(view){
    complex = complexity();
    }
    else{
      complex = 0;
    }
    X = newMatrix(M->row, N->col);

    statusSet(M, 0);
    statusSet(N, 0);
    ms = getIntelligence(M, 0);
    ns = getIntelligence(N, 0);


    if((ms%15 == 0) && (ns%15 == 0)){
      if(complex){
     printf("\nOverall Complexity: (, , 2*n*(n - 1))\n");
      }
      X = diagMult(M, N, 0);

      // since diagonal matrices are lower triangular, upper
triangular and the
      // prefered matrix to work over, we return the funcion
here if we can.

      if(complex){
     printf("\nANSWER:\n");
     printf("\n");
     displayMatrix(M, 0);
     printf("\n*\n\n");
     displayMatrix(N, 0);
     printf("\n=\n\n");
     displayMatrix(X, 0);
      }
```



```
      freeMatrix(M);
      freeMatrix(N);

      return X;
    }
    else if((ms%3 == 0) && (ns%3 == 0)){
      if(complex){
     printf("\nOverall Complexity: (, , 2*n*(n - 1))\n");
      }
      X = lowerTriMult(M, N, 0);
    }
    else if((ms%5 == 0) && (ns%5 == 0)){
      if(complex){
     printf("\nOverall Complexity: (, , 2*n*(n - 1))\n");
      }
      X = upperTriMult(M, N, 0);
    }
    else{
      if(complex){
     printf("\nOverall Complexity: (, , 2*n*(n - 1))\n");
      }
      X = matrixMult(M, N, 0);
    }
    if(view){
      printf("\nANSWER:\n\n");
      displayMatrix(M, 0);
      printf("\n*\n\n");
      displayMatrix(N, 0);
      printf("\n=\n\n");
      displayMatrix(X, 0);
    }
    freeMatrix(M);
    freeMatrix(N);

    return X;
  }

/*******************************brainRowEchelonForm*********
*******************/
  matrixP brainRowEchelonForm(matrixP M, int view){
    int i, first;
    union field temp, one;

    // this function calculates the row echelon form of
matrix "M" and returns the matrixP.
    // It uses intelligence to decide the optimum method for
the task

    // view != 0 implies the user wants to know how det was
calculated, else just calculate det

    if(fieldType == RATIONAL){
      one.F_rat.num = 1;
      one.F_rat.den = 1;
```



```
    }
    else{
      one.F_mod3 = 1;
    }

    statusSet(M, 0);

    if(getIntelligence(M, 0)%5 != 0){
    M = upperTriForm(M, 0);
    }

    for(i = 0; i < M->row; i++){
      first = findFirstNonZeroInRow(M, i, 0);
      if(first < M->col){
     temp = fieldDiv(one, getElement(M, i, first, 0));
     matrixRowMult(M, i, 0, temp);
      }
      else{
     matrixRowSwap(M, i, M->row - 1, 0);
     i--;
      }
    }

    if(view){
      printf("\nANSWER:\n");
      printf("Row Echelon Form of A:\n\n");
      displayMatrix(M, 0);
    }
    return M;
  }

/*******************************brainDet********************
*********************/
  union field brainDet(matrixP M, int view){
    int i, j, k, complex, highest, status, cm, ca, cc, sc,
finished;
    union field accumulate, sign, minus, one;

    // this function calculates the determinant of matrix "M"
and returns the field value.
    // It uses intelligence to decide the optimum method for
the task

    // cm counts the multiplication complexity
    // ca counts the addition complexity
    // cc counts the comparison complexity
    // sc counts the comparison complexity of one statusSet
    // view != 0 implies the user wants to know how det was
calculated, else just calculate det

    if(view){
      complex = complexity();
    }
    else{
```

```
    complex = 0;
  }

  if(fieldType == RATIONAL){
    one.F_rat.num = 1;
    one.F_rat.den = 1;
  }
  else{
    one.F_mod3 = 1;
  }

  if(fieldType == RATIONAL){
    minus.F_rat.num = -1;
    minus.F_rat.den = 1;
  }
  else{
    minus.F_mod3 = 2;
  }

  // initialisation steps

  cm = 0;
  ca = 0;
  cc = 0;
  sc = (M->row * M->row * M->row / 2) + (M->row * M->row) +
M->row;
  accumulate = one;
  sign = one;
  finished = 0;

  for(k = 1; k < M->row; k++){
    M = monoMatrix(M, k, zeroSet());
  }

  // if the user wants to see the stages of the algorithm,
first display dimension 0 of the matrix:

  if(complex){
    printf("\n");
    displayMatrix(M, 0);
  }

  // start the multi dimensional loop:

  for(k = 2; k < M->row + 1; k++){

    // call the function status set to find out what we can
about the kth dimension

    statusSet(M, k - 2);
    cc += sc;
    status = getIntelligence(M, k - 2);
```



```
    // if the matrix is triangular (or diagonal) at some
dimension then we can easily calculate the
    // det of that dimension. Consequently there is no need
to calculate the next dimension
    // status = 3, 5 or 15 means the dimension is
triangular. In this case we calculate the det of
    // that dimension as the product of diagonal enteries.
Then we multiply that by the previous
    // top left entries of each dimension (if there are
any) and then by sign. Return this product as
    // our det:

    if(status%3 == 0 || status%5 == 0){
   if(complex){
     printf("\n\nThe system has noticed that:");
     statusAnalyse(M, k - 2);
     printf("\nThe determinant of this dimension matrix is
now simply the");
     printf("\nproduct of its diagonal enteries.");
     printf("\nThis allows us to stop calculating any
further dimensions\n");
     if(status%19 == 0){
       if(complex){
         printf("Obviously since there is a zero on the
diagonal we know the");
         printf("\ndeterminant will be zero\n\n");
       }
       M->det = zeroSet();
       finished = 1;
       break;
     }
   }
   for(i = k - 2; i < M->row; i++){
     accumulate = fieldMult(accumulate, getElement(M, i, i,
k - 2));
     cm += 1;
   }
   for(i = 0; i < k - 2; i++){
     accumulate = fieldMult(accumulate, getElement(M, i, i,
i));
     cm += 1;
   }
   cm += 1;

   M->det = fieldMult(accumulate, sign);
   finished = 1;
   break;
    }

    // if we can spot immediately that the det of any
dimension is zero we know the overall det
    // is zero. In this case we exit the function returning
the zalue zero.
```



```
    // apart from status = 1, 3, 5 or 15, all possible
status for square matrices imply a zero det.
    // after the last check we know status is not 3, 5 or
15, so we only nedd check status is not 1:

    if(status != 1){
   M->det = zeroSet();
   finished = 1;
   break;
    }

    // if the status of our kth dimension = 1, then we have
not detected any special properties about
    // our matrix. In this case we check if the diagonal
entry we are suppose to divide by is zero:

    cc += 1;
    if(zeroCheck(getElement(M, k - 2, k - 2, k - 2)) == 1){

   // if this diagonal entry is zero for dimension k, then
we swap row k with some zero row x > k
   // which has a non zero entry in the kth column. We
choose row x such that we maximise entry(x, k)
   // (before we swap the rows):

   highest = k - 2;
   for(i = k - 1; i < M->row; i++){
     if(fieldCompare(fieldMod(getElement(M, i, k - 2, k -
2)), fieldMod(getElement(M, highest, k - 2, k - 2))) == 1){
       cc += 1;
       highest = i;
     }
   }

   // it will never happen that all the enteries checked
were zero because we would have already
   // known it from our status checks

   // now swap row k with row "highest":

   matrixRowSwap(M, k - 2, highest, k - 2);
   if(view){
     printf("\nNeed to swap rows %i and %i\n\n", 1, highest
+ 3 - k);
     displayMatrix(M, k - 2);
   }

   // swapping a row (or column) of a matrix changes the
sign of the determinant
   // we accumulate the overall sign change by the integer
"sign":

   sign = fieldMult(sign, minus);
   cm += 1;
```



```
   // however their are now 2 possibilities:
   // status can detect modified dimension has det = zero
   // status can detect triangular properties
   // but we are guaranted we will not need to swap any
more rows therefore
   // so rechecking this dimension is worthwhile and will
not lead to any further loops:
   // call the function status set to find out what we can
about the kth dimension

   statusSet(M, k - 2);
   cc += sc;
   status = getIntelligence(M, k - 2);

   // if the matrix is triangular (or diagonal) at some
dimension then we can easily calculate the
   // det of that dimension. Consequently there is no need
to calculate the next dimension
   // status = 3, 5 or 15 means the dimension is
triangular. In this case we calculate the det of
   // that dimension as the product of diagonal enteries.
Then we multiply that by the previous
   // top left entries of each dimension (if there are any)
and then by sign. Return this product as
   // our det:

   if(status%3 == 0 || status%5 == 0){
     if(complex){
       printf("\n\nThe system has noticed that:");
       statusAnalyse(M, k - 2);
       printf("\nThe determinant of this dimension matrix
is now simply the");
       printf("\nproduct of its diagonal enteries.");
       printf("\nThis allows us to stop calculating any
further dimensions\n");
       if(status%19 == 0){
         if(complex){
        printf("Obviously since there is a zero on the
diagonal we know the");
         printf("\ndeterminant will be zero\n");
         }
         M->det = zeroSet();
         finished = 1;
         break;
       }
     }
     for(i = k - 2; i < M->row; i++){
       accumulate = fieldMult(accumulate, getElement(M, i,
i, k - 2));
       cm += 1;
     }
     for(i = 0; i < k - 2; i++){
```

```
        accumulate = fieldMult(accumulate, getElement(M, i,
i, i));
        cm += 1;
       }
       cm += 1;

       M->det = fieldMult(accumulate, sign);
       finished = 1;
       break;
     }

     // if we can spot immediately that the det of any
dimension is zero we know the overall det
     // is zero. In this case we exit the function returning
the zalue zero.
     // apart from status = 1, 3, 5 or 15, all possible
status for square matrices imply a zero det.
     // after the last check we know status is not 3, 5 or
15, so we only nedd check status is not 1:

     if(status != 1){
       M->det = zeroSet();
       finished = 1;
       break;
     }
      }

      // my variation on Bariess' algorithm (this is the main
use of the three dimensional matrices)
      // see chapter 5.2 in the dissertation for a full
explinantion

      for(i = k - 1; i < M->row; i++){
     for(j = k - 1; j < M->row; j++){
       setElement(M, i, j, k - 1, fieldSub(getElement(M, i,
j, k - 2),fieldDiv(fieldMult(getElement(M, i, k - 2, k - 2),
getElement(M, k - 2, j, k - 2)), getElement(M, k - 2, k - 2,
k - 2))));
       cm += 2;
       ca += 1;
     }
      }

      // if the user wants to see the stages of the
algorithm, display each dimension of the matrix:

      if(complex){
     printf("\nthis is next dimension:\n\n");
     displayMatrix(M, k - 1);
     printf("\npress any key to continue: ");
     scanf("%i",&z);
      }
    }
```



```
    // in order to now find the determinant we take the
product of particular elements from the
    // 3 dimensional structure we have built and multiply by
the accumulated sign:

    if(!finished){
      for(i = 0; i < M->row; i++){
     accumulate = fieldMult(accumulate, getElement(M, i, i,
i));
      cm += 1;
       }
      cm += 1;

      // set the det in our matrix struct:

      M->det = fieldMult(sign, accumulate);
    }

    if(view){
      printf("\n\nANSWER:\n");
      printf("\nOverall complexity: (%i,%i,%i)\n", cm, ca,
cc);
      printf("\nThe determinant of the matrix:\n\n");
      displayMatrix(M, 0);
      printf("\n\nis equal to: \n");
      displayField(M->det);
      printf("\n");
    }
    return M->det;
  }

/*****************************brainInverse***************
*********************/
  matrixP brainInverse(matrixP M, int view){
    matrixP X;
    union field one;
    int i, j;

    // this function calculates the inverse of matrix "M" and
returns the matrixP

    // view != 0 implies the user wants to know how inverse
was calculated, else just calculate inverse

    brainDet(M, 0);
    if(zeroCheck(M->det)){
      printf("\nANSWER:\n");
      printf("\nSince the determinant of this matrix is zero
there does not exist an inverse\n");
      return M;
    }

    if(fieldType == RATIONAL){
      one.F_rat.num = 1;
```



```
      one.F_rat.den = 1;
    }
    else{
      one.F_mod3 = 1;
    }

    X = newMatrix(M->row, 2 * M->col);
    for(i = 0; i < M->row; i++){
      for(j = 0; j < M->col; j++){
     setElement(X, i, j, 0, getElement(M, i, j, 0));
      }
    }

    for(i = 0; i < M->row; i++){
      for(j = M->col; j < 2 * M->col; j++){
     setElement(X, i, j, 0, zeroSet());
      }
    }

    for(i = M->col; i < 2 * M->col; i++){
      setElement(X, i - M->col, i, 0, one);
    }

    X = diagonalise(X, 0);

    for(i = 0; i < X->row; i++){
      matrixRowMult(X, i, 0, fieldDiv(one, getElement(X, i,
i, 0)));
    }

    for(i = 0; i < M->row; i++){
      for(j = 0; j < M->col; j++){
     setElement(M, i, j, 0, getElement(X, i, j + M->col, 0));
      }
    }
    if(view){
      printf("\nANSWER:\n");
      printf("\nThe inverse matrix: \n\n");
      displayMatrix(M, 0);
    }
    return M;
  }

/*****************************brainSystemSolve************
******************/
  matrixP brainSystemSolve(matrixP A, matrixP B){
    matrixP X, calcMatrix;
    int i, j;
    union field one;

    // this function solves the system AX = B as described in
the dissertation, then returns
    // the matrixP
```



```
    if(zeroCheck(brainDet(A, 0))){
      printf("\n\n");
      return monoMatrix(B, 0, zeroSet());
    }

    if(fieldType == RATIONAL){
      one.F_rat.num = 1;
      one.F_rat.den = 1;
    }
    else{
      one.F_mod3 = 1;
    }

    X = newMatrix(B->row, B->col);

    calcMatrix = newMatrix(A->row, (A->col + B->col));

    for(i = 0; i < A->row; i++){
      for(j = 0; j < A->col; j++){
     setElement(calcMatrix, i, j, 0, getElement(A, i, j, 0));
      }
    }

    for(i = 0; i < A->row; i++){
      for(j = A->col; j < calcMatrix->col; j++){
     setElement(calcMatrix, i, j, 0, getElement(B, i, j - A-
>col, 0));
      }
    }

    calcMatrix = diagonalise(calcMatrix, 0);

    for(i = 0; i < X->row; i++){
      matrixRowMult(calcMatrix, i, 0, fieldDiv(one,
getElement(calcMatrix, i, i, 0)));
    }

    for(i = 0; i < X->row; i++){
      for(j = 0; j < X->col; j++){
     setElement(X, i, j, 0, getElement(calcMatrix, i, j + A-
>col, 0));
      }
    }

    printf("\nANSWER:\n");
    printf("\nmatrix X =\n\n");
    displayMatrix(X, 0);
    printf("\nSo the solved system says:\n\n");
    displayMatrix(A, 0);
    printf("\n*\n\n");
    displayMatrix(X, 0);
    printf("\n=\n\n");
    displayMatrix(B, 0);
```

```
    freeMatrix(A);
    freeMatrix(B);

    return X;
}

/*******************************brainControllability********
******************/
  union field brainControllability(matrixP A, matrixP B){
    matrixP reachability, Apowers, ApowersB;
    int i, j;

    // this function calculates the determinant of the
reachability matrix for the system
    // described in the dissertation, then returns the field
value. If this value is zero
    // the system is not controllable.

    reachability = newMatrix(A->row, A->col);
    Apowers = newMatrix(A->row, A->col);
    ApowersB = newMatrix(A->row, 1);

    // set the first column of the reachability matrix equal
to column vector B

    for(i = 0; i < reachability->row; i++){
      setElement(reachability, i, 0, 0, getElement(B, i, 0,
0));
    }

    // set the remaining columns of the reachability matrix
equal to column vector A(^j)*B

    matrixSet(Apowers, A, 0);

    for(j = 1; j < reachability->col; j++){
      matrixSet(ApowersB, brainMatrixMult(Apowers, B, 0), 0);
      for(i = 0; i < reachability->row; i++){
     setElement(reachability, i, j, 0, getElement(ApowersB,
i, 0, 0));
      }
      if(j + 1 != reachability->col){
     matrixSet(Apowers, brainMatrixMult(A, ApowersB, 0), 0);
//failed line
      }
    }

    brainDet(reachability, 0);

    printf("\nANSWER:\n");
    printf("\nReachability matrix (A,B):\n\n");
    displayMatrix(reachability, 0);
    printf("\nReachability matrix determinant: \n\n");
    displayField(reachability->det);
```



```
    if(!zeroCheck(reachability->det)){
      printf("\n\nThis implies: YES, the system is
controllable\n");
    }
    else{
      printf("\n\nThis implies: NO, the system is not
controllable\n");
    }

    freeMatrix(Apowers);
    freeMatrix(ApowersB);
    freeMatrix(reachability);

    return(reachability->det);
  }

// the following "menu" functions interact between the system
and the user. In general
// they set up valid user defined matrices for the tasks at
hand and passes them to
// the necessary brain functions.

/*******************************menuFieldChoice************
******************/
  void menuFieldChoice(void){
    int F;

    // this function sets the global variable "fieldType"
equal to the users choice of
    // rational or modulo 3

    F = 2;
    printf("\nFIELD CHOICE\n\n");
    printf("Please choose which field you wish to work
over\n\n");
    printf("The available choices are:\n");
    printf("                          0 - rational\n\n");
    printf("                          1 - modulo 3\n\n");
    while (F != 0 && F != 1){
      printf("Choice: ");
      scanf("%i",&F);
      if (F != 0 && F != 1){
     printf("\nInvalid choice: please choose 0 or 1\n\n");
      }
    }
    fieldType = F;
  }

/*********************************menuMatrixConstMult********
*************/
  int menuMatrixConstMult(void){
    matrixP M, X;
    union field c;
```



```
    // this function lets the user create an arbitrary two
dimensional matrix
    // and define a constant
    // then outputs the result of multiplying the matrix by
that constant
    // (their is no complexity review for this simple
function)

    printf("\nMATRIX MULTIPLICATION BY A CONSTANT\n\n");
    printf("This function asks you to input a matrix and a
constant. It will");
     printf("\nreturn the multiplication of this matrix by
this constant:\n");

    M = brainCreateMatrix(0, 0, 0);
    printf("multiplication constant: ");
    c = fieldInput();

    X = brainMatrixConstMult(M, c, 1);

    printf("\npress any key to continue: ");
    scanf("%i",&z);

    return 1;
  }

/******************menuMatrixElementWiseMult***************
***************/
  int menuMatrixElementWiseMult(void){
    matrixP M, N, X;

    // this function lets the user create an arbitrary two
dimensional matrix
    // and another matrix with the same dimensions
    // then outputs the result of element wise producting the
two matrices

    printf("\nMATRIX ELEMENT WISE MULTIPLICATION\n\n");
    printf("This function asks you to input two matrices, it
will then return the");
    printf("\nelement  wise product of these two matrices.
The dimensions you choose");
    printf("\nfor the first matrix will be the dimensions of
the second matrix also:\n");

    printf("\nANSWER:\n");
    printf("\nMatrix 1:\n");
    M = brainCreateMatrix(0, 0, 0);
    printf("\nMatrix 2:\n\n");
    N = brainCreateMatrix(0, M->row, M->col);

    X = brainMatrixElementWiseMult(M, N, 1);

    printf("\npress any key to continue: ");
```



```
    scanf("%i",&z);

    return 1;
  }

/*****************************menuMatrixDirectSum**********
*********************/
  int menuMatrixDirectSum(void){
    matrixP M, N, X;

    // this function lets the user create two arbitrary two
dimensional matrices
    // then outputs the result of the direct sum of the two
matrices

    printf("\nMATRIX DIRECT SUM\n\n");
    printf("This function asks you to input two matrices, it
will then return the");
    printf("\ndirect sum of these two matrices.\n");

    printf("\nMatrix 1:\n");
    M = brainCreateMatrix(0, 0, 0);
    printf("\nMatrix 2:\n\n");
    N = brainCreateMatrix(0, 0, 0);

    X = brainMatrixDirectSum(M, N, 1);

    printf("\npress any key to continue: ");
    scanf("%i",&z);

    return 1;
  }

/*********************************menuMatrixAdd**********
*******************/
  int menuMatrixAdd(void){
    matrixP M, N, X;

    // this function lets the user create an arbitrary two
dimensional matrix
    // and another matrix with the same dimensions
    // then outputs the result of adding the two matrices
    // (there is the option of a complexity review)

    printf("\nMATRIX ADDITION\n\n");
    printf("This function asks you to input two matrices, it
will then return the");
    printf("\nsum of these two matrices. The dimensions you
choose for the first");
    printf("\nmatrix will be the dimensions of the second
matrix also:\n");

    printf("\nMatrix 1:\n\n");
    M = brainCreateMatrix(0, 0, 0);
```

```
    printf("\nMatrix 2:\n\n");
    N = brainCreateMatrix(0, M->row, M->col);

    X = brainMatrixAdd(M, N, 1);

    printf("\npress any key to continue: ");
    scanf("%i",&z);

    return 1;
  }
**********************************menuMatrixSub************
*******************/
  int menuMatrixSub(void){
    matrixP M, N, X;

    // this function lets the user create an arbitrary two
dimensional matrix
    // and another matrix with the same dimensions
    // then outputs the result of subtracting the second
matrix from the first
    // (there is the option of a complexity review)


    printf("\nMATRIX SUBTRACTION\n\n");
    printf("This function asks you to input two matrices, it
will then return");
    printf("\nthe first matrix minus the second. The
dimensions you choose for the");
    printf("\nfirst matrix will be the dimensions of the
second matrix also:\n");

    printf("\nMatrix 1:\n\n");
    M = brainCreateMatrix(0, 0, 0);
    printf("\nMatrix 2:\n\n");
    N = brainCreateMatrix(0, M->row, M->col);

    X = brainMatrixSub(M, N, 1);

    freeMatrix(X);

    printf("\npress any key to continue: ");
    scanf("%i",&z);

    return 1;
  }

/*********************************menuMatrixMult************
*******************/
  int menuMatrixMult(void){
    matrixP M, N, X;

    // this function lets the user create an arbitrary two
dimensional matrix
```



```
    // and another matrix with compatible post multiplication
dimensions
    // then outputs the result of post multiplying the first
matrix by the second
    // (there is the option of a complexity review)

    printf("\nMATRIX MULTIPLICATION\n\n");
    printf("This function asks you to input two matrices, it
will then return");
    printf("\nthe first matrix post multiplied by the second.
The dimensions");
    printf("\nyou choose for the first matrix will dictate
the dimensions of");
    printf("the second matrix also:\n");

    printf("\nMatrix 1:\n\n");
    M = brainCreateMatrix(0, 0, 0);
    printf("\nMatrix 2:\n\n");
    N = brainCreateMatrix(0, M->col, 0);
    X = brainMatrixMult(M, N, 1);

    freeMatrix(X);

    printf("\npress any key to continue: ");
    scanf("%i",&z);

    return 1;
  }

/*********************************menuRowEchelonForm*********
******************/
  int menuRowEchelonForm(void){
    matrixP M;

    // this function lets the user create an arbitrary two
dimensional square matrix
    // A then outputs a Row Echelon Form of A

    printf("\nRow Echelon Form\n\n");
    printf("This function asks you to input a square matrix
A, it will then return");
    printf("\na Row Echelon Form of A.\n");

    printf("\nmatrix A:\n");
    M = brainCreateMatrix(1, 0, 0);
    brainRowEchelonForm(M, 1);

    freeMatrix(M);

    printf("\npress any key to continue: ");
    scanf("%i",&z);

    return 1;
  }
```



```
/*********************************menuDet*******************
******************/
  int menuDet(void){
    int complex;
    matrixP M;
    union field d;

    // this function lets the user create an arbitrary two
dimensional matrix
    // then outputs the determinant of that matrix
    // (there is the option of a complexity review)

    printf("\nCALCULATE DETERMINANT\n\n");
    printf("The algorithm programmed to calculate the
determinant is an original");
    printf("\nvariation on Bariess' algorithm which is based
upon Gaussian elimination:");
    printf("\n\nFor the inputted square matrix of degree n  a
further n-1 square matrices");
    printf("\nwill be calculated. Each new matrix calculated
has degree 1 smaller than its");
    printf("\npredecessor, which gave rise to the new matrix.
The matrices were");
    printf("\ncalculated in such a way the determinant of
some matrix i, multiplied by");
    printf("\nthe entry (1,1) of the matrix i+1 which gave
rise to matrix i, gives the");
    printf("\ndeterminant of matrix i+1. Consequently the
product of entries (1,1) for");
    printf("\nthe n matrices calculated (including the
inputted matrix) gives the");
    printf("\ndeterminant of the inputted matrix up to its
sign. Multiplying this product");
    printf("\nby the continuously adjusted sign equals the
exact determinant.\n\n");
    printf("Here are the matrices calculated for this
algorithm:\n");
    printf("\n\nBrute force compexity: (%i,%i,0)");

    printf("\nmatrix A:\n");
    M = brainCreateMatrix(1, 0, 0);
    brainDet(M, 1);

    freeMatrix(M);

    printf("\npress any key to continue: ");
    scanf("%i",&z);

    return 1;
  }

/*********************************menuInverse****************
**********************/
```



```
  int menuInverse(void){
    matrixP M;

    // this function lets the user create an arbitrary two
dimensional
    //square matrix, A. Then returns its inverse.

    printf("\nINVERSE\n");
    printf("\nThis function takes input:\n");
    printf("\nA, a square matrix (n*n)");
    printf("\nThe function returns A inverse:\n");

    printf("\nmatrix A:\n");
    M = brainCreateMatrix(1, 0, 0);
    brainInverse(M, 1);

    freeMatrix(M);

    printf("\npress any key to continue: ");
    scanf("%i",&z);

    return 1;
  }

/*********************************menuSystemSolve************
**********************/
  int menuSystemSolve(void){
    matrixP X, A, B;

    // this function lets the user create an arbitrary two
dimensional matrix, A
    // and a rectangular matrix with the same number of rows,
B
    // then outputs the soloution matrix X of the system AX =
B

    printf("\nSOLVE AX=B\n");
    printf("\nThis function takes input:\n");
    printf("\nA, a square matrix (n*n)");
    printf("\nB, a rectangular matrix (n*m)\n");
    printf("\nThis function gives output:\n");
    printf("\nX, a rectangular matrix (n*m) such that
AX=B\n");

    printf("\nmatrix A:\n");
    A = brainCreateMatrix(1, 0, 0);
    printf("\nmatrix B:\n");
    B = brainCreateMatrix(0, A->row, 0);

    X = brainSystemSolve(A, B);

    freeMatrix(X);

    printf("\npress any key to continue: ");
```

```
    scanf("%i",&z);

    return 1;
  }

/*****************************menuControllability*********
*************************/
  int menuControllability(void){
    matrixP A, B;
    int i, j;
    union field one;

    // this function lets the user create an arbitrary two
dimensional matrix, A
    // and a column vector with the same number of rows, B.
    // If the system d/dx = Ax +Bu is controlable the
function outputs yes otherwise no.
    // If yes it means there exist a u such that any target
state.
    // is reachable.
    // (there is no option of a complexity review)

    printf("\nCONTROLLABILITY\n");
    printf("\nA common problem in engineering is the question
of controllability.\n");
    printf("\nThis function considers the system: d/dx = Ax
+Bu (*)");
    printf("\nfor A, a square matrix (n*n)");
    printf("\n    B and x, column vectors (n*1)");
    printf("\n    u, a field element (1*1)\n");
    printf("\nThis function takes input:");
    printf("\nA, a square matrix (n*n)");
    printf("\nB, a column vector (n*1)\n");
    printf("\nThis function gives output:\n");
    printf("\nYes, if (*) is controlable\n");
    printf("\nNo, otherwise\n");

    printf("\nmatrix A:\n");
    A = brainCreateMatrix(1, 0, 0);
    printf("\nmatrix B:\n");
    B = brainCreateMatrix(0, A->row, 1);

    brainControllability(A, B);

    freeMatrix(A);
    freeMatrix(B);

    printf("\npress any key to continue: ");
    scanf("%i",&z);

    return 1;
  }
```



```
/*****************************************menu************
***********************/
  int menu(void){
    int c;

    c = -1;
    printf("\nMENU\n\n");
    printf("Please choose an option:\n\n");
    printf("                          1  - Choose a field to
work over\n\n");
    printf("                          2  - Matrix
Addition\n\n");
    printf("                          3  - Matrix
Subtraction\n\n");
    printf("                          4  - Matrix
Multiplication\n\n");
    printf("                          5  - Matrix
Multiplication by a constant\n\n");
    printf("                          6  - Matrix Element Wise
Multiplication\n\n");
    printf("                          7  - Matrix Direct
Sum\n\n");
    printf("                          8  - Row Echelon
Form\n\n");
    printf("                          9  - Calculate Matrix
Determinant\n\n");
    printf("                          10 - Calculate Matrix
Inverse\n\n");
    printf("                          11 - Solve a System of
Linear Equations\n\n");
    printf("                          12 - Controllability of a
System\n\n");
    printf("                          0  - Exit\n\n");
    while(c < 0 || c > 12){
      printf("Choice: ");
      scanf("%i",&c);
    }
    return c;
  }

/*****************************************MAIN************
***************/
/*****************************************MAIN************
***************/
/*****************************************MAIN************
***************/
/*****************************************MAIN************
***************/

int main(){

long a, b;
rational c, d;
char word[500];
```



```
char F;
union field e, f;
int rows, cols, choice, i;
matrixP M, N;
union field m1, m2, m3, m4, m5, m6, m7, m8, m9;
union field n1, n2, n3, n4, n5, n6, n7, n8, n9;

// the default field the user works over is the rational
numbers

fieldType = RATIONAL;

// introduce the system

printf("\nA SIMPLIFIED COMPUTER LINEAR ALGEBRA SYSTEM FOR
MATRICES\n");
printf("\nThe system is capable of solving simple matrix
arithmetic and some more complicated");
printf("\nmatrix functions. It has been given a human like
intelligence and as the user you");
printf("\nwill be able to see the steps the system takes to
solve some of the Linear Algebra");
printf("\nprobelms it can solve, if you want to. The system
is capable of working over the");
printf("\nrational numbers and modulo 3, the default field is
the rational numbers.\n");


printf("\npress any key to continue: ");
scanf("%i",&z);

// create an infinite loop and keep iterating between the
menu and the system tasks
// untill user wishes to exit.

for(i = 1; i > 0; i++){
  choice = menu();
  if(choice == 0){
    printf("\nEXIT SUCCESSFUL\n\n");
    return 1;
  }
  if(choice == 1){
    menuFieldChoice();
  }
  if(choice == 2){
    menuMatrixAdd();
  }
  if(choice == 3){
    menuMatrixSub();
  }
  if(choice == 4){
    menuMatrixMult();
  }
  if(choice == 5){
```



```
    menuMatrixConstMult();
  }
  if(choice == 6){
    menuMatrixElementWiseMult();
  }
  if(choice == 7){
    menuMatrixDirectSum();
  }
  if(choice == 8){
    menuRowEchelonForm();
  }
  if(choice == 9){
    menuDet();
  }
  if(choice == 10){
    menuInverse();
  }
  if(choice == 11){
    menuSystemSolve();
  }
  if(choice == 12){
    menuControllability();
  }
}
```