

1 Introduction

This is a brief manual for the GilLib library. The library's goal is to make it easy for people who are used to high level languages like MATLABTM and GAUSSTM to write C programs. I started writing this library while I was learning C. At that time, I still had fresh in my memory all the hurdles that made the learning curve for C so steep. The conception philosophy for GilLib is to make it as easy as possible for a scientist to migrate to C by evacuating from it the concepts that a scientist doesn't need or care to learn. Such concepts as memory allocation and management, pointers and object programming will not have to be understood in order to use the library. GilLib is pronounced *Zhil-lib*, which is slang pronunciation for *Gilles Libre*; this translates to *Free Gilles*.

So far, there is only one object in used, the `matrix`. To allocate memory, just one simple set of commands, `*Mnew`. Furthermore, when you use pointers, you don't have to know that this is what you're using. Contrarily to other libraries, simplicity of understanding (least amount of concept to learn) and speed are always a priority over elegance. Your code will not look good, but it will be fast (fifteen to twenty times faster than high level languages) and you won't have to worry about the side effects of using objects (memory leaks, for example).

1.1 Installation

Once you've downloaded file `gillib-0.9.tgz`, expand it and go in to the directory with commands:

```
tar xzvf gillib-0.9.tgz
cd gillib
```

You can edit file `OPTIONS`, adding what is appropriate to `_MACH = .`. Also, you will get more efficient results if you *tune* it to your particular machine. For example, for GCC version 3, you write `-march=pentium3` on a Pentium III or `-march=athlon-xp` on an Athlon XP. See your compiler DOCS for what to write. Then you copy it to the `work` directory, so both `OPTIONS` are for the same machine. Once this is done, you type `makeall`

Then (as root) move the created libraries in `/usr/local/lib` directory and copy the `*.h` files in `/usr/local/include`.

```
mv *.a /usr/local/lib/
cp *.h /usr/local/include/
```

And your all done! To use this library, you can go to the `work` directory and put your C files there or create your own working directory by copying the files `OPTIONS` and `Makefile` that are in the `work` directory. You will now create programs by typing `make foo` (if your C file is named `foo.c`). You can try by typing `make Msee` which creates a utility for viewing `mat`-files.

2 Basics

Unlike in high level languages, you have to declare variables in C. In GilLib, you have three numeric types: `matrix`, `scalar` and `int`. `int` is an integer, mostly used in loops for counting and for accessing individual elements of a matrix. `scalar` is a real floating point number, its precision varies at compile time. For people familiar with C, `scalar` is an alias to `float`, `double` or `long double` depending on whether the `PRECISION` variable in the file `OPTIONS` is set to `SINGLE`, `DOUBLE` or `TRIPLE`. Always use `scalar` instead of `float`, `double` or `long double`. It's very practical.

The `matrix` variable type is a little more complicated. It is the only object in the library. The indexing either starts at `[0][0]` or at `[1][1]` depending on whether the `INDICES` variable in the file `OPTIONS` is set `MATLAB` or not. The i, j element of matrix A is `A.re[i][j]` if it's a real matrix, `A.re[i][j]` for the real part and `A.im[i][j]` for the imaginary part if it's a complex matrix or `A.it[i][j]` if it's a integer matrix. The number of rows and columns are `A.rows` and `A.cols`; don't change those, they are set by the `*Mnew` and `*Mload` functions. The type of the matrix is `A.tag%10` (`A.tag` modulus ten). It is equal to 0 for real matrices, 1 for complex matrices and 2 for integer matrices. The name of the matrix is in character string `A.name`. In the A case, it will be automatically be set to "A" at memory allocation time. The `A.name` string is mostly used for internal purposes. To change this variable, use the `Mname` function:

```
Mname(A, "B");
```

this change the internal name of A (`A.name`) to "B".

3 Memory allocation and IO functions

Besides declaring variables, you also have to allocate memory before using matrices. The basic functions for memory allocation are the `*Mnew` ones. You use `Mnew` to allocate memory to a real matrix, `cMnew` to a complex matrix and `iMnew` for a matrix of integer. For example, to allocate memory to real matrix A of size m by n , you write:

```
Mnew(A, m, n);
```

Be sure to always allocate memory before using a matrix. The exception here are the `*Mload` functions (`Mload`, `cMload` and `iMload`) when you want to load a matrix from a file. In that case, the memory allocation is done automatically. Usage is:

```
Mload(X, "file.mat");
```

In this example, it load matrix X from file `file.mat`. If the file extension is `mat`, it assumes the file to be a MATLAB 4 mat-file, otherwise, it assumes an ASCII (text) file. In the case of a mat-file, it will load the matrix named "X" in the file. To load a different matrix and put the data in matrix "X", use the `Mloadname` (or `cMloadname`, or `iMloadname`) function:

```
Mloadname(X, "file.mat", "othername");
```

where "othername" is the name of the matrix inside the mat-file.

If you want to re-use a matrix, but with different size, you have to free the memory. Use the `Mfree` (or `cmfree`, or `imfree`) function. Usage is simple:

```
Mfree(X);
```

To save a matrix to a file, use the `Msave` (or `cmsave`, or `imsave`) function. So far, you can only save one matrix by ASCII file, but you can save many in a mat-file. Usage is similar to `Mload`:

```
Msave(X,"file.mat");
```

This will save the matrix X under the name `X.name` in the file `file.mat`. If you want it to be saved under a different name, you can use:

```
Msave(X,"file.mat","othername");
```

If the file extension is `.mat`, it saves in a MATLAB Mat file. Otherwise, if it's `.m`, in an executable MATLAB/Octave file; if it's `.ma`, in a MATLAB ASCII file; if it's `.oa`, in a Octave ASCII file; if it's `.ya`, in a Yorick ASCII file; if it's `.c`, in a C ASCII file. With any other extension, it saves in a generic ASCII file or in the format specified by function `Msavepref`. `Msavepref` formats how `Msave` write ASCII files. 0 is for plain ASCII, 1 for MATLAB ASCII, 2 for Octave ASCII, 3 for Yorick ASCII, 4 for MATLAB `.m` file, 5 for ordinary C file, 6 for GilLib C file and `80+i` for ASCII file with i precision and no E notation. You write:

```
Msavepref(number);
```

Here are other practical functions:

```
Mshow(A);
```

Prints matrix A on standard output (the screen).

```
Mtshow(A);
```

Prints the transpose of matrix A on standard output.

```
Mheader("file.mat", rows, cols, "X");
```

Writes header to file. It must be used before `Mcat` or `Mtcat`. `rows` and `cols` are the final or total number of rows and columns the matrix is going to have. `"X"` is the name the matrix is going to have in file `"file.mat"`.

```
Mtailer("file.mat");
```

Writes end to file, used before `Mcat` or `Mtcat`. It actually does something only in MATLAB m-files, Yorick and C files, not in other types of ASCII files nor in mat-files.

```
Mcat(A,"file.mat");
```

Concatenates matrix A at the end of a file `"file.mat"`, practical when you get result row by row in an iterative way.

```
Mtcat(A,"file.mat");
```

Concatenates the transpose of matrix A at the end of a file "file.mat".

4 Matrix operations

The `Mapp*` functions and the `Mfor` macro can be used for point to point operations. So far, the `Mapp*` functions can only be used with real matrices. For complex or integer matrices, use `Mfor`. For example,

```
MappM(A,exp,B); and
```

```
Mfor(A,i,j) A.re[i][j] = exp(B.re[i][j]);
```

are equivalent. Every element of A will be the exponential of the same element in B . The first argument of a `Mapp*` function is the output matrix, the second is the function being *mapped*. The others are the inputs. The letters at the end of a `Mapp*` function name tells the computer what kind of input the *mapped* function takes. `S` stand for a single scalar, `I` for a single integer, `M` for a real matrix and `Mt` for the transpose of the real matrix. For example, to put the transpose of matrix A in B ($B = A'$), you can write:

```
MappMt(A,equal,B);
```

where `equal` is just a dumb function that returns its input. With `Mfor`, the same would be:

```
Mfor(B,i,j) B.re[i][j] = A.re[j][i];
```

Note how the indices were inverted. `equal` can also be used to fill a matrix with one value:

```
MappS(A,equal,0);
```

Fills matrix A with zeros, same as

```
Mfor(A,i,j) A.re[i][j] = 0;
```

Here is a list of the `Mapp*` functions in the library:

<code>MappI</code>	<code>MappII</code>	<code>MappIM</code>	<code>MappIMt</code>	<code>MappIS</code>
<code>MappM</code>	<code>MappMI</code>	<code>MappMM</code>	<code>MappMMt</code>	<code>MappMS</code>
<code>MappMt</code>	<code>MappMtI</code>	<code>MappMtM</code>	<code>MappMtMt</code>	<code>MappMtS</code>
<code>MappS</code>	<code>MappSI</code>	<code>MappSM</code>	<code>MappSMt</code>	<code>MappSS</code>

Plus `Mapp` which doesn't take any input. For example, `Mapp(A,rndU01);` will fill matrix A with uniform deviates between zero and one; function `rndU01` doesn't take any input. To do the same with `Mfor`, you write:

```
Mfor(A,i,j) A.re[i][j] = rndU01();
```

For matrix multiplication, you use the `Mmul*` where the meaning of the letters representing the inputs are the same as the `Mapp*` functions. For example, to multiply the transpose of X with itself and put the result in Q ($Q = X'X$), you write:

```
MmulMtM(Q,X,X);
```

For multiplication of three matrices, the algorithm will find the fastest way. The output

matrix must be different than the input. The `Mmul*` functions are the only ones in the library that can't have the same input and output. Here is a list of the `Mmul*` functions in the library:

<code>MmulMM</code>	<code>MmulMMt</code>	<code>MmulMtM</code>	<code>MmulMtMt</code>
<code>MmulMMM</code>	<code>MmulMMMt</code>	<code>MmulMMtM</code>	<code>MmulMMtMt</code>
<code>MmulMtMM</code>	<code>MmulMtMMt</code>	<code>MmulMtMtM</code>	<code>MmulMtMtMt</code>
<code>MmulSMM</code>	<code>MmulSMMt</code>	<code>MmulSMtM</code>	<code>MmulSMtMt</code>
<code>MmulSMMM</code>	<code>MmulSMMMt</code>	<code>MmulSMMtM</code>	<code>MmulSMMtMt</code>
<code>MmulSMtMM</code>	<code>MmulSMtMMt</code>	<code>MmulSMtMtM</code>	<code>MmulSMtMtMt</code>

Plus `SmulVV` which multiplies two vectors (row or column vector, the function checks) and returns a scalar. Contrarily to functions where the output is a matrix, when a function returns a scalar, it works like traditional function: `a = Smul(A,B)`;

There are other matrix functions, here are brief descriptions of them. Unless otherwise mentioned, we call by convention the output matrix `Out`, the output scalar `out`, the input matrices `A` or `B` and the input scalars `a`, or `b`.

`Mfill(Out, a)`;

Fills matrix `Out` with scalar `a`. It's equivalent to: `MappS(Out, equal, a)`;

`Meye(Out)`;

Creates an identity matrix with matrix `Out`.

`Mseqa(Out)`;

Puts `Out.re[i][j] = i` for all `j`.

`Mtseqa(Out)`;

Puts `Out.re[i][j] = j` for all `i`.

`Mtrans(Out, A)`;

`Out` is the matrix transpose of `A`. It's equivalent to: `MappMt(Out, equal, A)`;

`Mkron(Out, A, B)`;

`Out` is the Kronecker product of matrices `A` and `B`.

`Minv(Out, A)`;

`MinvC(Out, A)`;

`Out` is the inverse of matrix `A`. If `A` is symmetric, `MinvC` is faster, otherwise use `Minv`.

`Msolve(x, A, b)`;

`MsolveC(x, A, b)`;

Solve the equation system $Ax = b$ where `A` is a square matrix, `b` is a column vector and `x`

is an unknown column vector. If A is symmetric, `MsolveC` is faster, otherwise use `Msolve`.

`Mchol(Out, A);`

Out is the Cholesky decomposition of matrix A .

`Mlu(L, U, A);`

Both outputs L and U are the LU decomposition of matrix A . L is lower triangular and U is upper triangular.

`Mdiag(Out, A);`

If *Out* is a vector (row or column) and A is a matrix, *Out* is the diagonal of matrix A . If *Out* and A are matrices, *Out* is a diagonal matrix equal to the diagonal of A . Otherwise if *Out* is a matrix and A is a vector (row or column), *Out* is a diagonal matrix with the element of A in the diagonal.

`out = Sdet(A);`

`out = SdetC(A);`

out is the determinant of matrix A . If A is symmetric, `SdetC` is faster, otherwise use `Sdet`.

`out = Strace(A);`

out is the trace of matrix A .

`Msum(Out, A);`

Out is a vector, if its size is `A.rows` and different than `A.cols`, *Out* is the sum of all elements in each row of A . If its size is `A.cols` and different than `A.rows`, *Out* is the sum of all elements in each column of A . If A is a square matrix, then *Out* is the sum of all elements in each row of A when *Out* is a column vector and of all elements in each column when *Out* is a row vector. Lastly, if *Out* is a one by one matrix, it is the sum of all elements of A .

`Mmean(Out, A);`

It works like `Msum`, but *Out* is a mean instead of a sum.

`Mvar(Out, A);`

It works like `Msum`, returning variances instead of sums if *Out* is a vector. Otherwise *Out* will be a variance-covariance matrix, it has to be a square matrix. In that latter case, if `Out.rows` (equal to `Out.cols`) is equal to `A.rows`, *Out* gives the variance-covariance matrix of the columns of A . If its equal to `A.cols` and different than `A.rows`, its the variance-covariance matrix of the rows of A .

`Mcov(Out, A);`

It works like `Mvar`, but only produces a variance-covariance matrix. *Out* has to be square

with A having the same number of rows or columns as Out .

```
out = Ssum(A);  
out = Ssum(A, n);
```

out is the sum of all elements of A when there is only one input. With two inputs, out is the sum of all elements of A when $n = 0$, the sum of all elements of row n if $n > 0$ and the sum of all elements of column $-n$ when $n < 0$. For this case, we consider the first element to be one, second to be two, etc.. The value of the variable `INDICES` in the file `OPTIONS` does not influence that.

```
out = Smean(A);  
out = Smean(A, n);
```

It works like `Msum`, but out is a mean instead of a sum.

```
out = Svar(A);  
out = Svar(A, n);
```

It works like `Msum`, but out is a variance instead of a sum.

5 Scalar functions and utilities

The scalar functions are here to complement those in the math library and be used with the `Mapp*` functions. As of version 0.9 of the library, parts of the Cephes library from Stephen Moshier is included. That library is a replacement for the math library with more functions for scientist. You'll find documentation of those functions in the file `cephes.doc` supplied with `GilLib`. All the obvious functions are there (`exp`, `log10`, `sin`, `tanh`, `acos`, ...) and quite a few more.

```
out = max(a,b);
```

out is the maximum of a or b .

```
out = min(a,b);
```

out is the minimum of a or b .

```
out = add(a,b);
```

out is a plus b .

```
out = sub(a,b);
```

out is a minus b .

```
out = mul(a,b);
```

out is a times (multiplied by) b .

```
out = divd(a,b);
```

out is a divided by b (the `div` name was already taken by the math library).

```
out = sgn(a);
```

out is -1 if a is negative, otherwise it's 1 .

```
out = sqr(a);
```

out is a times a , the square of a .

```
out = neg(a);
```

out is the negative of a ($-a$).

```
out = equal(a);
```

out is equal to a .

```
out = d_equal(a);
```

out is equal to 1 .

```
out = neg_for_odd(i);
```

out is 1 if i is even and -1 if i is odd. i is an `int`

```
out = factorial(i);
```

out is the factorial ($i!$) of i , i is an `int`.

```
out = bincoeff(n, i);
```

out is the binomial coefficient ($n!/((n-i)!i!)$). n and i are `int`.

```
out = sigmoid(a);
```

Sigmoid function, *out* is $1/(1 + \exp(a))$.

```
out = d_sigmoid(a);
```

Derivative of the sigmoid, *out* is $-1/\text{sqr}(1 + \exp(a))$.

```
out = invsigmoid(a);
```

Inverse of the sigmoid, *out* is $\log(-1 + 1/a)$.

```
out = Id0(a);
```

out is a if a is positive, otherwise *out* is 0 .

```
out = d_Id0(a);
```

out is 1 if a is positive, otherwise *out* is 0 .

```
out = d_tanh(a);
```

Derivative of the hyperbolic tangent, *out* is $0.5/(1 + \cosh(a))$.

The next few functions are utilities that you could find practical.

```
timerinit();
timer(it, totalit);
```

The timer functions are used when doing long programs with many iterations. Run `timerinit` before using `timer`. `timer` prints a message with how much time is left before the end of the loop. `it` is at what iterations the program is at now, `totalit` is the total number of iterations that has to be done.

The `error` function is used to put error messages in programs. It works like the `printf` function in the standard C library, see your docs on `printf`. `error` will exit the program (`exit(1);`) after printing the message:

```
error("Error message");
```

```
errorM("function", A);
errorMM("function", A, B);
errorMMM("function", A, B, C);
errorMMMM("function", A, B, C, D);
```

These error functions are mostly for internal use. They print a *not conformable* error message than `exit`. Add the name of the function that failed and the matrices used in that function.

```
disp("message", ...);
```

This function works just like `printf` from the standard C library. It's there only to avoid loading `stdio.h`.

6 Statistical functions

These are functions used for statistics. Some are not finished.

```
seed(a);
```

Before using a random number generator, first use the `seed` function. It seeds the pseudo-random number generator. If $a = 0$, the seed will use `time(0)` from the standard C library so the generated sequence will change from one execution of the program to the other. If a is any other positive integer, the sequence will always be the same.

The `cdf*` functions return the cumulative density functions value below x . The `inv*` functions return the inverse of the cdf at probability x . The `pdf*` functions return the probability density functions at value x . The `rnd*` functions return a random deviate. For example, you write:

```
out = cdfX2(x, df)
out = pdfU(x, a, b);
```

```
out = invN(x, mu, sigma);
out = rndT(df);
```

Distribution	Parameters	Deviate	PDF	CDF	Inverse
Beta	SS	rndB	pdfB	cdfB	invB
Binomial	IS	rndBin	pdfBin	cdfBin	invBin
Cauchy	SS	rndCau	pdfCau	cdfCau	invCau
Chi square	I	rndX2	pdfX2	cdfX2	invX2
Exponential	S	rndExp	pdfExp	cdfExp	invExp
F	II	rndF	pdfF	cdfF	invF
Gamma	SS	rndGam	pdfGam	cdfGam	invGam
Geometric	S	rndGeo	pdfGeo	cdfGeo	invGeo
Gumbel type 1	SS	rndGum1	pdfGum1	cdfGum1	invGum1
Gumbel type 2	SS	rndGum2	pdfGum2	cdfGum2	invGum2
Hypergeometric	III	rndHGeo	pdfHGeo	cdfHGeo	invHGeo
Kolmogorov-Smirnov	\emptyset			cdfKS	
Laplace	\emptyset	rndLap	pdfLap	cdfLap	invLap
Logistic	SS	rndLog	pdfLog	cdfLog	invLog
Log-Normal	SS	rndLnN	pdfLnN	cdfLnN	invLnN
Normal	SS	rndN	pdfN	cdfN	invN
Multivariate Normal	VM	MrndN			
Pascal	IS	rndPas	pdfPas	cdfPas	invPas
Poisson	S	rndPois	pdfPois	cdfPois	invPois
Standard Normal	\emptyset	rndN01	pdfN01	cdfN01	invN01
Standard Uniform	\emptyset	rndU01	pdfU01	cdfU01	invU01
Student-t	I	rndT	pdfT	cdfT	invT
Uniform	SS	rndU	pdfU	cdfU	invU
Weibull	SS	rndWeib	pdfWeib	cdfWeib	invWeib
Wiener	I	rndWien			
Wishart	MI	MrndWish			

6.1 Model specific implementations

OLS_(Theta, y, X);

OLS is not finished yet, underscore is temporary. OLS does ols on the model: $y = X\beta + u$.

The dimensions of **Theta** defines the output:

$k \times 1$: returns β ;

$k \times 2$: returns β and standard deviations of β ;

$(k + 1) \times 1$: returns β and s^2 ;

$(k + 1) \times 2$: returns β , s^2 and standard deviations of β , s^2 ;

$(k + 2) \times 1$: returns β , s^2 and R^2 ;

$(k + 2) \times 2$: returns β , s^2 , standard deviations of β , s^2 , R^2 and adjusted R^2 ;

$(k + 3) \times 1$: returns β , s^2 , R^2 and adjusted R^2 ;

```
reindex(iM);
```

Reshuffle the column vector of binary numbers in order so as to eventually go through every possible combinations. iM is a vector matrix of `int` all equal to 0 and 1. For example, from 0000111 to 1110000 or from 000011 to 110000.

```
Wsave(A, "filename");
```

Saves data from `in` to a Wav sound file. Matrix A has two columns.

```
WsaveH(length, "filename");
```

Creates the header of a Wav sound file. $length$ is an `int`.

```
Wload(A, secs, "filename");
```

Loads one second after $secs$ seconds in Wave file. Puts the result in matrix A .

```
out = WloadS("filename");
```

Loads the length of the file. out is an `int`.

Those are functions used for multilayered perceptrons in neural networks. `MPL1` has one layer. `Params` is of size $(NbOfOutput + NbOfInput + 1) \times (NbOfNodes + 1)$. The network is of the form (a, c are scalar, B, D are vectors):

$$Output_i = Ofunc \left(c_i + \sum_j [D_{i,j} Hfunc(a_j + B'_j * Input)] \right)$$

For example, if $NbOfOutput = 2$, $NbOfInput = 3$ and $NbOfNodes = 4$, $Params$ will be

$$\begin{array}{cccccc} \emptyset & a_1 & a_2 & a_3 & a_4 & \\ \emptyset & b_{11} & b_{12} & b_{13} & b_{14} & \\ \emptyset & b_{21} & b_{22} & b_{23} & b_{24} & \\ \emptyset & b_{31} & b_{32} & b_{33} & b_{34} & \\ c_1 & d_{11} & d_{12} & d_{13} & d_{14} & \\ c_2 & d_{21} & d_{22} & d_{23} & d_{24} & \end{array}$$

```
MPL1init(Params);
```

Randomly initiate the parameters.

```
MPL1(OutPut, InPut, Params, Hfunc, Ofunc);
```

```
MPL1ni(OutPut, InPut, Params, Hfunc, Ofunc);
```

Fills $OutPut$ given matrices $InPut$ and $Params$ and functions $Hfunc$ (the hidden layer) and $Ofunc$ (the output layer). `MPL1ni` is the same neural net, but with no intercept.

```
MPL1update(dOutPut, InPut, Params, alpha, Hfunc, dHfunc, dOfunc);
```

```
MPL1updateni(dOutPut, InPut, Params, alpha, Hfunc, dHfunc, dOfunc);
```

Used to update *Params*. *dOutput* will be the derivative of the net multiplied by scalar *alpha*. The functions *dHfunc* and *dOfunc* are the derivative of *Hfunc* and *Ofunc*.

```
dMLP1_dI(dNN, InPut, Params, Hfunc, dHfunc, dOfunc);
```

Gives the derivative of the network in respect to *InPut* for each *OutPut*. *dNN* is of size $(NbOfOutPut \times NbOfInPut)$.

```
dMLP1_dP(dOutput, InPut, Params, dParams, alpha, Hfunc, dHfunc, dOfunc);
```

Gives the derivative of the network in respect to *Params*. *dParams* is of same size as *Params*

7 Example

The following is an example of OLS estimation. There is a OLS function in the library. As one can see the writing is a little tedious and ugly. The advantage here is speed and if you understand this example, you can use the library easily.

```
#include "gilllib.h"

int main()
{
    scalar s2;                /* Declare the scalar */
    matrix X, y, Q, u;        /* Declare the matrices */

    Mload(X,"file.mat");      /* Loads 'X' from file 'file.mat' */
    Mload(y,"file.mat");      /* Loads 'y' from file 'file.mat' */
    Mnew(B, X.cols, 1);        /* Allocate memory to 'B' */
    Mnew(Q, X.cols, X.cols);  /* Allocate memory to 'Q' */
    Mnew(u, y.rows, 1);        /* Allocate memory to 'u' */

    MmulMtM(Q, X, X);         /* Q = X'*X */
    MinvC(Q, Q);              /* Q = inv(Q) */
    MmulMMtM(B, Q, X, y);     /* B = Q*X'*y */
    MmulMM(u, X, B);          /* u = X*B */
    MappMM(u, sub, y, u);     /* u = y - u */
    s2 = SmulVV(u, u)
        / (scalar)(X.rows-X.cols); /* s2 = u'*u / (T-K) */

    Mshow(B);                 /* Prints 'B' */
    disp("s2 = %g\n", s2);    /* Prints 's2' */

    return 0;
}
```