Project Report:

CALL C function From MATLAB

Group: Ling Guo, Kaixian Yu,Yilin Dai

Project Purpose:

MATLAB is a numerical computing environment and fourth generation programming language. Developed by The MathWorks, MATLAB allows matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages.

C is one of the most popular programming languages. It is widely used on many different software platforms, and there are few computer architectures for which a C compiler does not exist. C has greatly influenced many other popular programming languages, most notably C++, which originally began as an extension to C.

In this course "intro - Scientific Programming, we are going to explore the interface of Matlab with other programs. Since C is the most popular programming languages in the every platform.Thus, we decide to choose the topic "CALL C function From MATLAB".

In this report, we will mention How to achieve Calling C function From MATLAB and then compare the efficiency and necessity about doing this job. briefly speaking, the reasons includes using pre-existing functions or libraries and increase the speed when deal with huge data and programs with large loops.

Project Work:

Teammate A: Robin

Research the detail of the interface of MATLAB to C and get similar with data type and functions defined in "mex.h" and teach other two teammates. Make some example code about How to deal with the interface to C.

Teammate B: Kai Xian

Select a topic to compare the efficiency using program written in matlab function with program written in c function. Both programs need follow the same algorithm and include the many loops. Result need be made in graph and comparison need be made.

Teammate C: Ling

Select a topic to prove the necessity of calling C from Matlab. C program need be written using gsl library to solve some special scientific problem which matlab build-in function is not so good at.

Each teammate will do 5-min presentation for one's responsible part and finish the one's part of the report.

Part I:

Learn How to Call C functions from MATLAB

The MATLAB is very good for putting together functions or scripts that run many of MATLAB's fast Built-In functions. One nice thing about these files is that they are never compiled and will run on any system that is already running MATLAB. MATLAB achieves this by interpreting each line of the M-File every time it is run.(kind likes java.) This method of running the code can make processing time very slow for large and complicated functions, especially those with many loops because every line within the loop will be interpreted as a new line, each time through the loop.

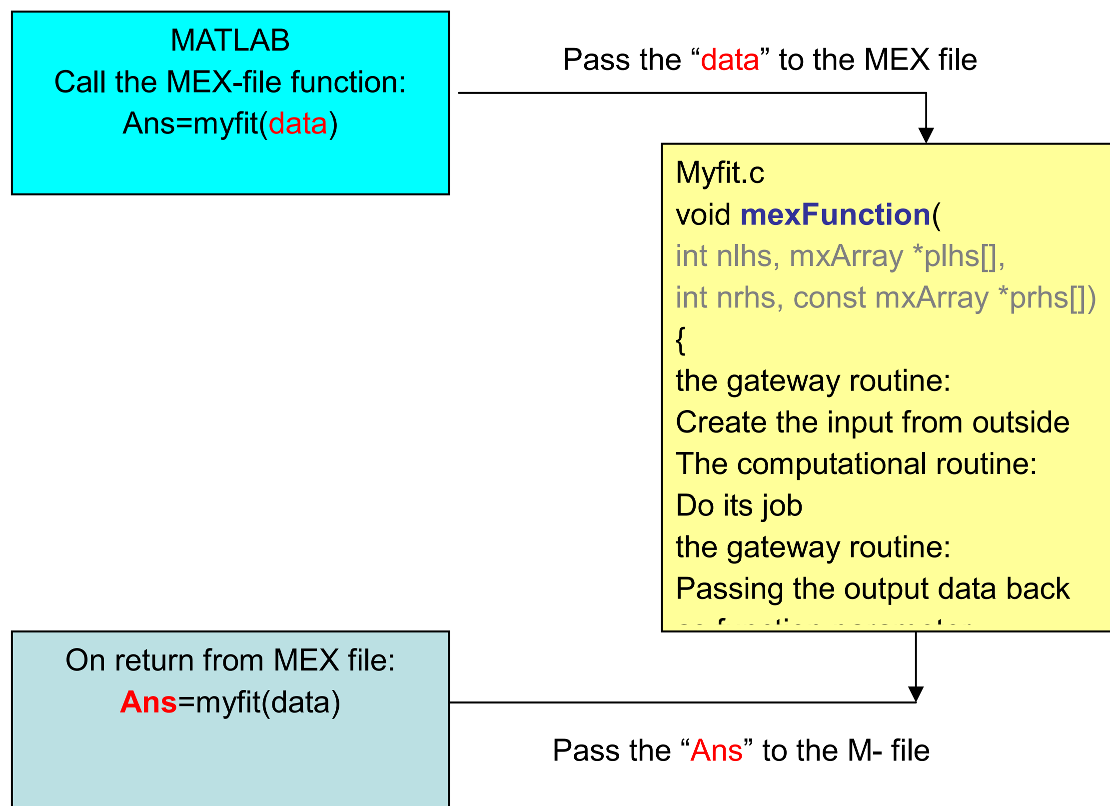
However, for the scientific programs, especially for statisticians, we usually deal with huge dataset and have to involve large loops in the codes. Then only MATLAB is not enough…..
MATLAB has the capability of running functions written in C. The files which hold the source for these functions are called MEX-Files.MEX stands for Matlab EXectuable. The mexFunctions are not intended to be a substitue for MATLAB's Built-In operations however if you need to code many loops and other things that MATLAB is not very good at, this is a good option.

Components of MEX Files

A MEX-file in Matlab consists of two distinct parts:

1. A computational routine: code that does what function is supposed to do.
2. A gateway routine: code that interfaces the computational routine with MATLAB.
   In this file, The main() function is replaced with mexFunction.

Usually the program will have the following structure:



In the MEX-file, the first thing is to tell Matlab this file is the MEX-file. To achieve this, we need include “mex.h” which has been defined by Matlab. We can easily find this file in every computer which has installed the Matlab

There is one example for the readers (Matlab is installed by default option)

Windows:C:\MATLAB6p5\extern\include

Unix:User\local\MATLAB\extern\include

The “mex.h” library contains all of the APIs that MATLAB provides. There are four input parameters to the mexFunction which correspond to the way a function is called in MATLAB

- nlhs (Type = int): This paramter represents the number of "left hand side" arguments.
- plhs (Type = array of pointers to mxArrays): This parameter is the actual output arguments. As we will see later, an mxArray is MATLAB's structure for holding data and each element in plhs holds an mxArray of data.
- nrhs (Type = int): Similar to nlhs, this parameter holds the number of "right hand side" arguments.
- prhs (Type = const array of pointers to mxArrays): This array hold all of the pointers to the mxArrays of input data.

As mentioned above, an mxArray is MATLAB's structure for holding data and each element in plhs holds an mxArray of data. We can find the definition of this instructure in the file “matrix.h” which is in the same folder to the “mex.h”. all of the APIs to deal with mxArray have been claimed here. NOTE: The elements in this structure should not be accessed directly. Inlined MEX-files are NOT guaranteed to be portable from one release of MATLAB to another.

To access the mcArray, we have to has buide-in APIs.we demo some example here:

Get the Data from MATLAB:

```
#include <mex.h>
#include<stdio.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
        mxArray *a;
        mxArray *b;
        double tmp1,tmp2;
////////////////////////////////////////////////////////
////////in
        a=prhs[0];
        b=prhs[1];
        tmp1=(double)(mxGetScalar(a));
        tmp2=(double)(mxGetScalar(b));
        printf("a+b=%lf\n",tmp1+tmp2);
        return;
}
```

In this example, we create two mxArray pointers a and b.and set the values prhs[0] and prhs[1] to them. The access the value of mcArray, we use buide-in API mxGetScalar to get the input from MATLAB.and set them to tmp1 and tmp2.
Then, we output the sum of these two input parameters.
The example is saved as RobinAdd.c.
In Matlab, we need compile this file first and use RobinAdd function as using M-Functions in MATLAB.

Returning Data to MATLAB

Assigning return values and data to the left hand side parameters is very similar to getting the data from the last section. The difference here is that memory must be allocated for the data strucure being used on the output. Here is an example which is the updated version of the first one

```
#include <mex.h>
#include<stdio.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
	double *ans;
	mxArray *a;
	mxArray *b;
	double tmp1,tmp2;
////////////////////////////////////////////////////////
////////in
	a=prhs[0];
	b=prhs[1];
	tmp1=(double)(mxGetScalar(a));
	tmp2=(double)(mxGetScalar(b));
	printf("a+b=%lf\n",tmp1+tmp2);
////////////////////////////////////////////////////////
////////in
	plhs[0]=mxCreateDoubleScalar(0);
	ans=mxGetPr(plhs[0]);
	*ans=tmp1+tmp2;
	return;
}
```

We create an mxArray using mxCreateDoubleScalar function and let it be the output value plhs[0].then we set the value which this pointer points to the sum of input parameters.

Calling Built-In M Functions from a MEX-File

there are so many useful and fast pre-written functions in MATLAB that it would be a crime if we could not use them. Luckily, The Mathworks (creators of MATLAB) has provided this capability. Built-In functions have a parameter list similar to the mexFunction itself.

```
#include <mex.h>
#include<stdio.h>

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
	double *ans;
	mxArray *a;
	mxArray *b;
	double tmp1,tmp2;
	mxArray *output_ptr[1], *input_ptr[1];
//////////////////////////////////////////////////////
////////in
	a=prhs[0];
	b=prhs[1];
	tmp1=(double)(mxGetScalar(a));
	tmp2=(double)(mxGetScalar(b));
	printf("a+b=%lf\n",tmp1+tmp2);

	input_ptr[0]    = mxCreateDoubleScalar(tmp1);
	output_ptr[0] = mxCreateDoubleScalar(0);
	mexCallMATLAB(1, output_ptr, 1, input_ptr, "sin");

	tmp1=mxGetScalar(output_ptr[0]);
	input_ptr[0]    = mxCreateDoubleScalar(tmp2);
	mexCallMATLAB(1, output_ptr, 1, input_ptr, "sin");

	tmp2=mxGetScalar(output_ptr[0]);
//////////////////////////////////////////////////////
////////out
	plhs[0]=mxCreateDoubleScalar(0);
	ans=mxGetPr(plhs[0]);
	*ans=tmp1+tmp2;
	return;
}
```

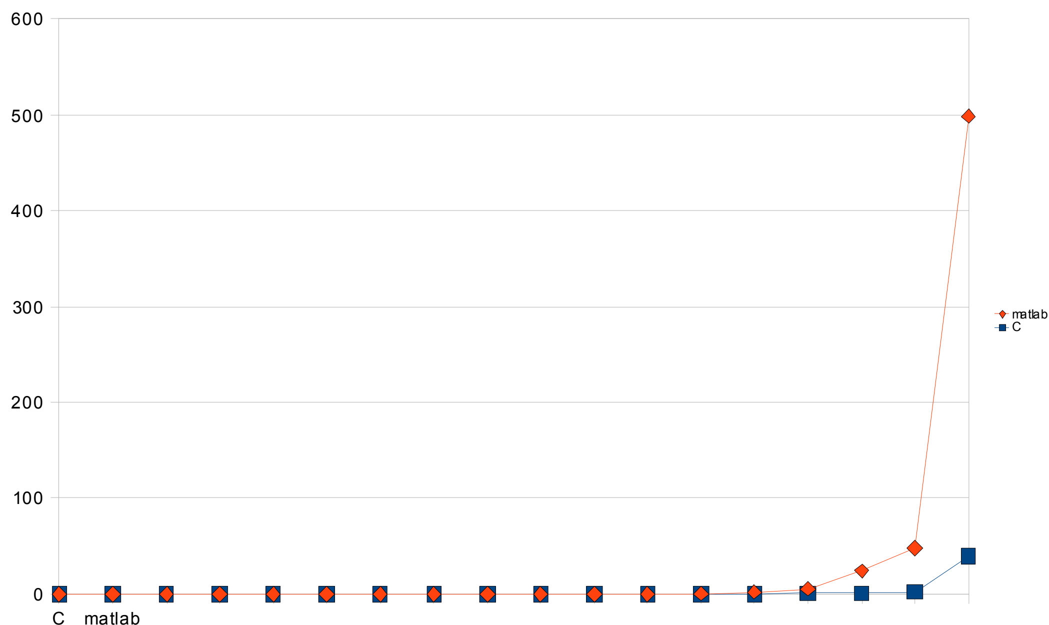
In this example, we Call the build-in function“SIN” by this command:
mexCallMATLAB(1, output_ptr, 1, input_ptr, "sin");
and you can extract the data as you please!

Part II:

The work I did is to write a C code and a matlab code of find prime numbers which are less than some number to compare the efficiency using program written in matlab function with program written in c function. Both the C and matlab code contain 2 loops, the outside one is used to count numbers, which should be less than the number given, the inside one is to check if the number from outside is prime.(code attached at the end).

The result shown in the table and graph:

	<10	<20	<30	<50	<70	<100	<200	<300	<500
C	0	0	0	0	0	0	0	0	0
matlab	0	0	0	0	0	0	0.01	0.01	0.02
	<800	<1,000	<5,000	<10,000	<50,000	<100,000	<500,000	<1,000,000	<10,000,000
C	0	0	0	0	0	1	1	2	40
matlab	0.04	0.05	0.23	0.48	2.34	4.62	23	45.65	458.61



From the table we can see that the time used to find prime of C is almost the same as the one of matlab when the number is small(the small should mean less than 50,000), but as the number increasing the time matlab used is increasing much faster than the one of C.

Improvement:

First, the time of matlab begins to increase too fast, so if take a log of it that will be much better;

Second the time counted in program is not exactly the time cost in running, it contains print time, change a little bit code could make it just counts the time actually used for calculating.

Codes:

C code:

Header.h:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

int checkp(int k);
```

prime.c: // used to check prime

```
#include "header.h"

int checkp(int k)
{
int i;int m=(int)floor(sqrt(k));
for(i=2;i<=m;i++)
{
if((k%i)==0) return 0;
}
return 1;
}
```

kaixian.c:

```
#include "header.h"
int main(void)
{int i,n;
time_t start,end;
double dif;

printf("Please enter the number you wish to compute");
scanf("%d",&n);
time(&start);
for(i=2;i<=n;i++)
{if (checkp(i))
{printf("%d",i);}}
time(&end);
dif = difftime (end,start);
printf("total time is %.8f",dif);
return 0;
}
```

makefile:

```
kaixian: kaixian.c header.h
gcc -lm -Os -o kaixian kaixian.c prime.c

clean:
rm -f kaixian
```

matlab code:

kaixian.m:

```
clear all
tic
count=0;
for i=2:500000// this 500000 is what we can change for different ones to count time.
j=2;
k=sqrt (j);
while j
if rem (i,j)==0
break
else
j=j+1;
end
if j>k
fprintf ('%5d’,i)
count=count+1;
if rem (count,13)==0
fprintf ('\n')
end
break
end
end
end
toc
```

Part III:

Introduction:

This project is to use C gsl library functions to solve a multi-parameter fitting problem.

The program use gsl blas function deal with matrix and vector and use least-squares fitting functions to build and use the model. The purpose of this program is to evaluate the goodness of model prediction.

The method we choose to evaluate the goodness of model prediction is called “leave-one-out cross validation”. It takes one observation out one at a time and use the remaining part to build a model. Based on the model, we can get the predicted value of the “leave-out” observation and the prediction error. If there are total n observations, then by “leave-one-out”, there will be total n models and total n predicted values. Usually the measurement to evaluate the goodness of prediction is the average of those n prediction errors.

This method is used when we want to evaluate the goodness of model prediction but there is no more data in hand except the data used to build the model.

Programming:

The data read in has a form with first row the total number of observations and the number of parameters needed to be estimated. From the second row to the end, the data has the form with the first column the response variable, and the second until the last column the predictor variables. This data form is the common form of data in statistic.

The main function is as follows:

```
#include "header.h"
int main(void)
{
     FILE *fin;
     gsl_matrix *A;
    gsl_vector *c,*ypred,*y1,*z1;
     double
    chisq=0.0,sum=0.0,temp1=0.0,ytrue=0.0,k1=0.0,k2=0.0,sumsse=0.0;
    double static sumerror=0.0;  //use to store the total prediction
errors of n “leave-one-out” process
    double err,est,error;  // error is used to get the prediction error
```

```
for every "leave-one-out" process
    int Matrixn; // matrix has n columns
    int Matrixm; // matrix has m rows
    int i,n,j;
//read in
    scanf("%d",&Matrixm);
    scanf("%d",&Matrixn);

//acllocate matrix and vector
    y1=gsl_vector_alloc(Matrixm);
    z1=gsl_vector_alloc(Matrixm);
    gsl_vector_set_all(z1,1.0);   // z1 is the vector with all 1's
    A=gsl_matrix_alloc(Matrixm,Matrixn);

//read in Matrix
    gsl_matrix_fscanf(stdin,A);
    gsl_matrix_get_col(y1,A,0);// get first column of the matrix into y1,
this is the response variable.
    gsl_matrix_set_col(A,0,z1);  // set the first column of A to 1, this
is the matrix of predictor

// use function myfit to evaluate the prediction error of model
    for(i=0;i<Matrixm;i++)
    {
        error=myfit(A,y1,i,Matrixm,Matrixn);
        sumerror+=error;

    }
    printf("the total of prediction error is %f\n",sumerror);
    printf("the times of prediction is %d\n\n",Matrixm);
    printf("the average of prediction error is %f\n",sumerror/Matrixm);

    return 0;
}
```

In the main function we first deal with the data read in. The prediction matrix X in multi-linear model $y = \beta_0 + \beta_1 X_1 + .. + \beta_{p-1} X_{p-1}$ has a form with first column 1's and remaining part the value of predictor variables for different observations. The y1 vector in the program is the value of response variable.

After the data has a standard from. Then we call the myfit function to realize the multi-linear fit and predict process. Since we use "leave-one-out" method,

then there will be a loop of total n times (n is the total number of observations).

The following is myfit function:

```
#include "header.h"

// this function is to build a least-square model based on leave one out
observations from original data, and then use the model for prediction.

double myfit(const gsl_matrix *A,const gsl_vector *y,int i,int
Matrixm,int Matrixn)
{
    gsl_multifit_linear_workspace *work;  //set up the work space for
gsl_multifit_linear
    gsl_matrix *SubA,*cov;  // SubA is used to store the leave one out X
matrix, cov is var-cov matrix of coefficients of predictor
    gsl_matrix *TmpA;
    gsl_vector *Suby,*c,*ypred,*Tmpy; // Suby is used to store the leave
one out y vecor, ypred is the obsevation leaved out.
    double ytrue;  // the true value of y
    double k1,k2;
    double chisq=0,est,err,error; // est is the estimate value of y based
on the leave one out model. error=yture-yest
    double static sumeer=0;
    SubA=gsl_matrix_alloc(Matrixm-1,Matrixn);
    Suby=gsl_vector_alloc(Matrixm-1);
    TmpA=gsl_matrix_alloc(Matrixm,Matrixn);
    Tmpy=gsl_vector_alloc(Matrixm);
    ypred=gsl_vector_alloc(Matrixn);
    c=gsl_vector_alloc(Matrixn);         //coefficient
    cov=gsl_matrix_alloc(Matrixn,Matrixn); // cov is the
variance-covariance matrix of the model
    work=gsl_multifit_linear_alloc(Matrixm-1,Matrixn);  //set up the
work space for gsl_multifit_linear
    gsl_matrix_memcpy(TmpA,A);  // copy matrix A to TmpA
    gsl_vector_memcpy(Tmpy,y); // copy y to Tmpy

//deal with SubA Matrix

    gsl_matrix_swap_rows(TmpA,i,Matrixm-1);  // change the ith row to the
last row
    gsl_matrix_view
B=gsl_matrix_submatrix(TmpA,k1,k2,Matrixm-1,Matrixn); // get the first
until the Matrixm-1 obaservations to the submatrix B.
```

```
    gsl_matrix_memcpy(SubA,&B.matrix);

// set up the vector ypred for  prediction and get ytrue

    gsl_matrix_get_row(ypred,TmpA,Matrixm-1);  //get the last row as the
observation to predict
     ytrue=gsl_vector_get(Tmpy,i);  // get yture

//deal with Suby Vector/

    gsl_vector_swap_elements(Tmpy,i,Matrixm-1); // change the ith
element and the last element of Tmpy
    gsl_vector_view yy=gsl_vector_subvector(Tmpy,0,Matrixm-1);  // get
the first until the last y value to yy
    gsl_vector_memcpy(Suby,&yy.vector);

//fit the model
     gsl_multifit_linear(SubA,Suby,c,cov,&chisq,work);

// use model for prediction
    gsl_multifit_linear_est(ypred,c,cov,&est,&err);
    printf("the value of estimate y is%lf\n",est);
    printf("the true value of y is%lf\n",ytrue);
    error=abs(ytrue-est);  // get the prediction error
    printf("the value of prediction error is %lf\n\n",error);
    return error;
}
```

In myfit function, we realize leave-one-out by changing the ith row and the last row of matrix A, and get the last row out. Then the data used to build the model is always the first n-1 observations.

We use `gsl_vector_view` and `gsl_matrix_view` to get the subvector and submatrix. A matrix view is a temporary object, stored on the stack, which can be used to operate on a subset of matrix elements. The elements of the view can by accessed using the matrix component of the view object. A pointer `gsl_matrix *` can be obtained by taking the address of the matrix component with the & operator.

The fitting function in C is **gsl_multifit_linear** (*const gsl_matrix * X, const gsl_vector * y, gsl_vector * c, gsl_matrix * cov, double * chisq, gsl_multifit_linear_workspace * work*).
These functions compute the best-fit parameters *c* of the model y = X c for the observations *y* and the matrix of predictor variables X. cov is the var-cov matrix of estimated parameters, chisq is the sum of square of residuals, The best-fit is

found by singular value decomposition of the matrix *X* using the preallocated workspace provided in*work*.

The function use in prediction is **gsl_multifit_linear_est** (*const gsl_vector * x, const gsl_vector * c, const gsl_matrix * cov, double * y, double * y_err*). It uses the best-fit multilinear regression coefficients *c* and their covariance matrix *cov* to compute the fitted function value *y* and its standard deviation *y_err* for the model y = x.c at the point *x*.

The output of this program has a form with its predicted value from the model build by the remaining observations, the true value of each observation, the prediction error. The last three rows show the total prediction error, how many cross validation we have used, and the average prediction error. If the main purpose of our model is used to predict further observation, then we would like this value as small as possible.

The following is a example of output with 5 observations and 3 parameters.

```
the value of estimate y is22.215385
the true value of y is30.000000
the value of prediction error is 7.000000

the value of estimate y is-90.520984
the true value of y is13.000000
the value of prediction error is 103.000000

the value of estimate y is21.931305
the true value of y is45.000000
the value of prediction error is 23.000000

the value of estimate y is39.484967
the true value of y is14.000000
the value of prediction error is 25.000000

the value of estimate y is48.077599
the true value of y is17.000000
the value of prediction error is 31.000000

the total of prediction error is 189.000000
the times of prediction is 5

the average of prediction error is 37.800000
```

In order to use the blas function and the regression function, we need to

include the `gsl/gsl_multifit.h, gsl/gsl_blas.h` in the header file. And also link them in make file use `-L/usr/local/lib -lgsl -lgslcblas`. The following is the header file.

```
#include <stdio.h>
#include <gsl/gsl_multifit.h>
#include<gsl/gsl_blas.h>
#include<math.h>
double myfit(const gsl_matrix *A,const gsl_vector *y,int i,int
Matrixm,int Matrixn);  //prototype of function myfit
```

Reference

- Writing C functions in Matlab(MEX-Files) Jason Laska, http://cnx.org/content/m12348/latest/
- Calling C from Matlab:introduction Andreas Uhl , http://www.cosy.sbg.ac.at/~uhl/C-matlab.pdf
- http://www.ccr.jussieu.fr/ccr/Documentation/Calcul/matlab5v11/docs/00009/009a0.htm