GNU SL 库帮助文档

马宇恒
匡亚明学院 *171240510*

目录

GNU SL 是一个免费的 C 语言数值计算例程集合。数值计算课程中曾经需要直接调用一部分功能，包含且不限于

- 常用数学函数与常数
- 复数
- 多项式
- 特殊函数
- 向量、矩阵和线性代数
- …

此外，一些功能是课程本身的实现目标，可以使用 GSL 库进行验证性工作，包含且不限于

- 数值积分
- 差值
- 一般方程数值解
- 线性方程组
- …

本文将会介绍这些可能用到的功能。虽然这些功能可以使用 Matlab 等高级语言轻松实现，但作为 C/C++ 的一个功能强大的数学库，学习 GSL 仍然具有一定意义。在某些领域，C/C++ 具有的速度和安全性优势使得 GSL 仍然有着相当多的使用。我第一次接触 GSL 也是因为处理的问题使用 Matlab 运算速度低的可怕。GSL 的相关声明以及反馈途径见官方网站

$$http://www.gnu.org/software/gsl/gsl.html.$$

本文接下来的命令行操作将以 $ 作为标志。

1 下载与安装

1.1 下载

在 http://mirrors.ibiblio.org/gnu/ftp/gnu/gsl/或者其他镜像下载最新的 GSL 包及其认证文件。下载完成后跟随 install.txt 里的指示完成安装，步骤如下：

- 打开命令行。（接下来的每步操作之后需要等待运行完成）
- 输入 $ *cd location*。location 为源代码的文件夹路径（例：/Users/mayuheng/Desktop/gsl）
- 输入 $ *./configure*
- 输入 $ *make*
- 输入 $ *make check*。若测试结果均为 pass 且命令行正常运行，可以继续。（耗时可能较长，一般可以跳过）
- 输入 $ *make install*
- 安装完成

不过该安装过程默认将库安装在/usr/local/bin 目录下面，如果要成功调用还需要进行操作。本文以 MACOS10.14，XCODE9.3 为例说明，其他平台和编译器请自行查询对应操作。

1.2 调用头文件

最便捷的操作方式是如图 1 添加 include 读取路径。

1. 选择你的项目
2. 找到 Build Setting.
3. 找到 Search Paths.
4. 找到 Header Search Paths.

5. 单机右侧输入你所安装 GSL 的路径，一般默认是/usr/local/include，可以在 README 中查看。

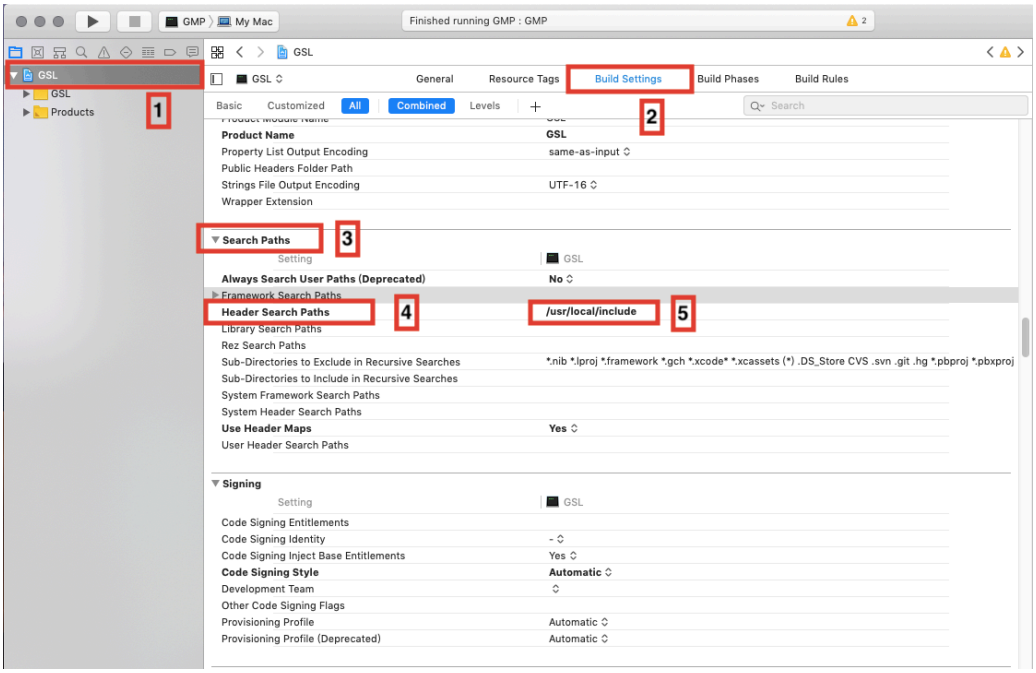

图 1: 调用头文件示意

1.3 连接库

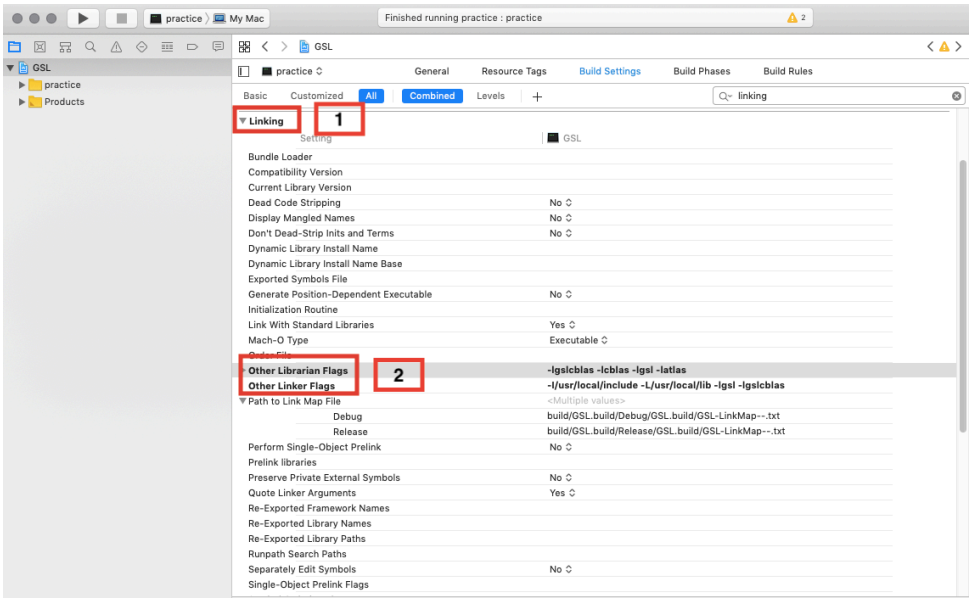

图 2: 连接库示意

1. 在同一菜单下找到 Linking
2. 找到 Other Librarian Flags 和 Other Linker Flags
3. 单击右侧，在 Other Librarian Flags 中添加-lgslcblas、-lcblas、-lgsl、-latlas
4. 单击右侧，在 Other Linker Flags 中添加-I/usr/local/include、-L/usr/local/lib、-lgsl、-lgslcblas（如果安装路径不是/usr/local，则变为相应目录下的-I···/include 和-L···/lib）

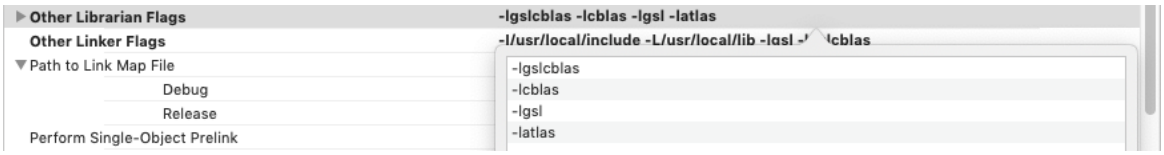

图 3: Other Librarian Flags

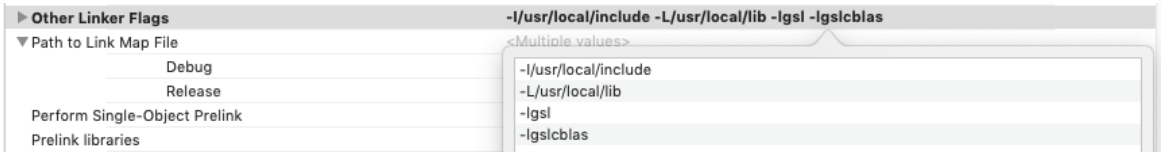

图 4: Other Linker Flags

到此，配置任务基本完成，可以使用以下样例程序测试输出 Bessel 函数在 x=5 的值。

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
int
main (void) {
double x = 5.0;
double y = gsl_sf_bessel_J0 (x);
printf ("J0(\%g) = \%.18e\n", x, y);
return 0;
}
```

Out : $J0(5) = -1.775967713143382642e-01$

注意到我们的 gsl 安装之后是以一个文件夹的形式与其他头文件并列，所以调用时要在前面加上路径 “gsl/”。

2 错误调试

GSL 库函数在出现不同错误时会返回相应的指针，将其输出可以查看错误类型。需要调用的头文件是 *gsl_errno.h*, 程序样例如下

```
int status = gsl_function(...)
if (status){
// 如果发生了错误则运行这段程序
printf("error:%s\n",gsl_strerror(status));
}
```

举例

```
int status=gsl_sf_bessel_J0(0);
printf("error:%s\n",gsl_strerror(status));

Out: error:input domain error
```

在 Debug 不出的绝望时刻可以考虑使用。

表 1: 常量表

M_E	e	M_2_SQRTPI	$\frac{2}{\sqrt{\pi}}$
M_LOG2E	$log_2 e$	M_1_PI	$\frac{1}{\pi}$
M_LOG10E	$log_{10} e$	M_2_PI	$\frac{2}{\pi}$
M_SQRT2	$\sqrt{2}$	M_LN10	$ln10$
M_SQRT1_2	$\sqrt{\frac{1}{2}}$	M_LN2	$ln2$
M_SQRT3	$\sqrt{3}$	M_LNPI	lne
M_PI	π	M_EULER	$Euler\,Constant\ \gamma$
M_PI_2	$\frac{\pi}{2}$	GSL_POSINF	$+\frac{1.0}{0.0}$
M_PI_4	$\frac{\pi}{4}$	GSL_NEGINF	$-\frac{1.0}{0.0}$
M_SQRTPI	$\sqrt{\pi}$	GSL_NAN	$\frac{0.0}{0.0}$

3 函数与常数

本部分一般需要调用 *gsl_math.h*。

3.1 常数与类型

GSL 中使用宏储存了很多常数的精确值，我们可以直接使用。举例

```
1  printf("%f",1/M_E);
2
3  Out: 0.367879
```

在检查程序的时候也常常用到下面三个函数

$$int\ gsl\_isnan(const\ double\ x)$$

$$int\ gsl\_isinf(const\ double\ x)$$

$$int\ gsl\_finite(const\ double\ x)$$

显然，他们用来检查异常值的类型，如果 x 是所属的类型则返回 1，否则返回 0。

3.2 函数

下面给出一些 GSL 常用的或者相对于一般 C 语言库函数特有的函数。GSL 还包含了一些特殊函数，课程作业有可能用到，如参数为 n 的 Bessel 函数

$$gsl\_sf\_bessel\_Jm(int\ n,\ double\ x)$$

表 2: 常用函数表

double gsl_log1p(const double x)	返回 ln(1+x) 的值，对很小的 x 精确
double gsl_expm1(const double x)	返回 $e^x - 1$ 的值，对很小的 x 精确
double gsl_hypot(const double x, const double y)	返回 $\sqrt{x^2+y^2}$ 的值，且尽量避免了超限
double gsl_hypot3(double x, double y, const double z)	返回 $\sqrt{x^2+y^2+z^2}$ 的值，且尽量避免了超限
double gsl_pow_int(double x, int n)	计算 x^n 的值，且采用快速幂，效率很高
double gsl_pow_uint(double x, unsigned int n)	同上，输入类型有别
GSL_IS_ODD(n)	判断 n 是否为奇数，是返回 1，否则为 0
GSL_IS_EVEN(n)	判断 n 是否为偶数，是返回 1，否则为 0
GSL_MAX(a,b)	返回两者的较大值
int gsl_fcmp(double x, double y, double epsilon)	判断 x 与 y 在 ε 水平上是否相等，返回 -1（x>y）, 0（x=y）, 1(x<y)

这些特殊函数的名称一般为 *gsl_sf_function*，需要调用 *gsl_sf_function.h*。

4 变量类型的扩展

4.1 复数

这部分内容一般需要调用 *gsl_complex.h* 和 *gsl_complex_math.h*。
GSL 中的复数类型为 *gsl_complex*，可以使用如下示例的两个函数定义一个复数。

```
gsl_complex z1=gsl_complex_rect(1, 1);// 直角坐标系下的横纵坐标
gsl_complex z2=gsl_complex_polar(1, M_PI/2);// 极坐标下的模长和幅角

printf("%f+%f i\n",z1.dat[0],z1.dat[1]);// 输出复数,
                                        // dat 是复数结构中的一个元素
printf("%f+%f i\n",z2.dat[0],z2.dat[1]);

Out: 1.000000+1.000000 i
     0.000000+1.000000 i
```

而获得一个复数的性质我们有下列函数:
改变一个复数，我们可以使用函数

void GSL_SET_COMPLEX(zp,x,y)

表 3: 复数返回函数表

$GSL\_REAL(z)$	返回一个复数的实部
$GSL\_IMAG(z)$	返回一个复数的虚部
$gsl\_complex\_arg(z)$	返回一个复数的幅角
$gsl\_complex\_abs(z)$	返回一个复数的模长
$gsl\_complex\_abs2(z)$	返回一个复数模长的平方
$gsl\_complex\_logabs(z)$	返回 $ln\|z\|$，在 $\|z\| \to 1$ 的时候精确

来为 z 赋值，也可以使用指针单独改变实部或者虚部的值，使用

$$GSL\_SET\_REAL(zp,x)$$

$$GSL\_SET\_IMAG(zp,y)$$

也可以直接改变 z 的 struct 中二元数组 dat 的值。示例如下

```
1       gsl_complex z=gsl_complex_rect(1,1);
2       GSL_SET_COMPLEX(&z, 2, 2);
3       printf("%f+%f i\n",z.dat[0],z.dat[1]);
4       z.dat[1]=0;
5       printf("%f+%f i\n",z.dat[0],z.dat[1]);
6       GSL_SET_IMAG(&z, -2);
7       printf("%f+%f i\n",z.dat[0],z.dat[1]);
8       GSL_SET_REAL(&z, -4);
9       printf("%f+%f i\n",z.dat[0],z.dat[1]);
10
11  Out: 2.000000+2.000000 i
12       2.000000+0.000000 i
13       2.000000+-2.000000 i
14       -4.000000+-2.000000 i
```

而复数的运算通过函数来完成（相对于直接编写复数类运算的表达上有些麻烦，但是图个方便吗这不是）。

表 4: 复数运算函数表

函数	返回值
$gsl\_complex\_add(gsl\_complex\, a,\ gsl\_complex\, b)$	$a+b$
$gsl\_complex\_sub(gsl\_complex\, a,\ gsl\_complex\, b)$	$a-b$
$gsl\_complex\_mul(gsl\_complex\, a,\ gsl\_complex\, b)$	ab
$gsl\_complex\_div(gsl\_complex\, a,\ gsl\_complex\, b)$	$\frac{a}{b}$
$gsl\_complex\_conjugate(gsl\_complex\, z)$	$\overline{z}$
$gsl\_complex\_inverse(gsl\_complex\, z)$	$\frac{1}{z}$
$gsl\_complex\_negative(gsl\_complex\, z)$	$-z$

表 5: 复变量函数表

函数	返回值
$gsl\_complex\_sqrt(gsl\_complex\, z)$	$\sqrt{z}$
$gsl\_complex\_pow\_real(gsl\_complex\, z,\ double\ x)$	z^x
$gsl\_complex\_exp(gsl\_complex\, z)$	e^z
$gsl\_complex\_log\_b(gsl\_complex\, z,\ gsl\_complex\, b)$	$log_b z$

常用的复变量函数列表如表 5。

对于复变函数中经常使用的三角函数、反三角函数、双曲函数、反双曲函数，GSL 中都有收录，调用形式诸如

$$gsl\_complex\_cos(z)$$

$$gsl\_complex\_sinh(z)$$

其中有一些对输入或输出的值有要求，比如要求为实数，跟随编译器的指引选择相应的函数名即可。

4.2 多项式

本部分一般需要调用 *gsl_poly.h*。

多项式的储存形式是数组。一个多项式和其储存在计算机中的形式是

$$a_0 + a_1 x + \cdots a_n x^n$$

$$[a_0, a_1, a_2, \cdots, a_n]$$

然而，对多项式本身的操作，例如求两个多项式的积，并没有在 GSL 中给出。本文给出可以参考的实现方式如表 6。具体代码放在附录中。

表 6: 多项式操作

函数	返回值
double plus(double p1[], int n1, double p2[], int n2)	求 n1-1 次的多项式系数组 p1 和 n2-1 次的多项式系数组 p2 的和
double minus(double p1[], int n1, double p2[], int n2)	求差
double multiple(double p1[], int n1, double p2[], int n2)	求积
double generate(double root[])	求以 root 中的数为根的首一多项式

计算一个多项式在 x 点处的各种值有如下方法。

表 7: 多项式操作

函数	返回值
gsl_poly_eval(*double c*[], *int len*, *double x*)	计算实值多项式
gsl_poly_complex_eval(*double c*[], *int len*, *gsl_complex z*)	计算实值复变多项式
gsl_complex_poly_complex_eval (*gsl_complex c*[], *int len*, *gsl_complex z*)	复变复值多项式
gsl_poly_eval_derivs(*double c*[], *size_t lenc*, *double x*, *double res*[], *size_t lenres*)	在 x 处的 k 阶导，返回值为一个数组
gsl_poly_dd_init(*double dd*[], *double xa*[], *double ya*[], *size_t size*)	通过一个点列（横纵坐标储存在 xa 和 ya 中）和均差（dd 中）返回一个多项式

有关多项式方程的函数。

表 8: 多项式方程操作

int gsl_poly_solve_quadratic(*double a*, *double b*, *double c*, *double* ∗ *x*0, *double* ∗ *x*1)
解一元二次方程
int gsl_poly_solve_cubic(*double a*, *double b*, *double c*, *double* ∗ *x*0, *double* ∗ *x*1, *double* ∗ *x*2)
解一元三次方程
int gsl_poly_complex_solve(*double* ∗*a*, *size_t n*, *gsl_poly_complex_workspace* ∗*w*, *gsl_complex_packed_ptr z*)
解出多项式的所有根，返回值为一个长度为 2（n-1）的数组，相邻两位依次为实部和虚部

4.3 块

这部分内容一般需要使用 *gsl_block.h*。
GSL 中的向量和矩阵都是使用数据类型块（block）来表示的，其定义如下

```
typedef struct
{
size_t size;
  double * data;
} gsl_block;
```

一个块有长度和数据，在使用时需要声明，示例如下

```
gsl_block * b = gsl_block_alloc (100); //分配空间
printf ("length of block = %zu\n", b->size);//输出长度
printf ("block data address = %p\n", b->data);//输出数组指针
gsl_block_free (b);//释放空间

 Out:
 length of block = 100
 block data address = 0x804b0d8
```

4.4 向量

这部分内容需要使用 gsl_vector.h。
下面我们使用一个例程来说明向量使用的方法。

```
int i;
    gsl_vector * v = gsl_vector_alloc (3);
    for (i = 0; i < 3; i++) {
        gsl_vector_set (v, i, 1.23 + i);//分配储存空间
    }
    for (i = 0; i < 3; i++) {
        printf ("v_%d = %g\n", i, gsl_vector_get (v, i)); }//输出
    gsl_vector * u = gsl_vector_alloc (3);//再声明一个向量
    //下面很多操作要求两向量长度相同，请自行体会
    //并且，许多操作是对复数可行的，可以自行探索
    gsl_vector_set_all(u,1);//将所有位置设为1
    gsl_vector_swap(u,v);
    for (i = 0; i < 100; i++)  {//下标超限报错
        printf ("v_%d = %g\n", i, gsl_vector_get (v, i)); }//输出
    gsl_vector_swap_elements(u,1,2);//交换向量元素
    gsl_vector_reverse(u);//倒置向量
    gsl_vector_add(u,v);//相加，减乘除只需把add换成sub、mul、div
    gsl_vector_scale(u,1);//数乘
    gsl_vector_add_constant(u,1);//加常数
    gsl_vector_max(u);//找出最大值，最小值同理
    gsl_vector_min_index(u);//找出最小值的下标，最大值同理
    gsl_vector_isnull(u);//判断是否是零向量
    gsl_vector_ispos(u);//是正向量
    gsl_vector_isneg(u);//是负向量
    gsl_vector_equal(u,v);//两向量是否相等
    gsl_vector_free (v);//释放储存空间
    gsl_vector_free(u);
```

4.5 矩阵

这部分内容需要使用 *gsl_matrix.h*。
我们仍然使用一个例程来说明。

```
int i,j;
    gsl_matrix * m = gsl_matrix_alloc (10, 3);//分配空间
    for (i = 0; i < 10; i++) for (j = 0; j < 3; j++)
        gsl_matrix_set (m, i, j, 0.23 + 100*i + j);//赋值
    for(i=0;i<10;i++){//输出
        printf("\n");
        for (j = 0; j < 3; j++)
        printf ("%g\t", i, j, gsl_matrix_get (m, i, j));}
    gsl_matrix_set_identity(m);//令m为单位对角
    gsl_matrix * ma = gsl_matrix_alloc (10, 3);//再声明一个矩阵
    gsl_matrix_set_all(ma,0);//把ma所有元素设为0
    gsl_matrix_swap(m,ma);//交换两者的值
    gsl_matrix_memcpy(m,ma);//把ma的值赋给m
```

```
14      gsl_vector *v1=gsl_vector_alloc(10);
15      gsl_vector *v2=gsl_vector_alloc(3);
16      gsl_matrix_get_row(v2,m,2);// 读取第二行的值到v2中，要求长度相同
17      gsl_matrix_get_col(v1,m,2);// 读取第二列
18      gsl_matrix_swap_rows(m,1,2);// 交换第一二行，交换列同理
19      gsl_matrix_free(m);// 释放空间
20      gsl_matrix_add(m,ma);// 对应元素相加
21      gsl_matrix_mul_elements(m,ma);// 对应元素相乘(不是矩阵乘法)
22      gsl_matrix_max(m);// 矩阵元素最大值
23      int imax,jmax;
24      gsl_matrix_min_index(m,&imax,&jmax);// 矩阵元素最大值下标
25      gsl_matrix_equal(m,ma);// 判断是否相等
```

5 线性代数

5.1 矩阵运算

GSL 没有定义简洁的矩阵乘法，略显美中不足，本文给出可以参考的实现[1]，具体代码放在附录中。

5.2 矩阵分解

这部分内容一般需要使用 *gsl_linalg.h*。

GSL 提供了很多矩阵分解的方式，常用的如 LU 分解、Cholesky 分解和完全正交分解。此处以 LU 分解为例。

$$int\ gsl\_linalg\_LU\_decomp(gsl\_matrix * A,\ gsl\_permutation * p,\ int * signum)$$

返回时 A 的上三角区域（包括对角线）储存，下三角区域储存 L，由于 L 为单位下三角，对角线可以不储存。的有时 LU 分解写做 PA=LU，P 是一个用来行置换的初等矩阵，即这里的 P。signum 是逆序数，可以理解为 det P。如果矩阵为复数域上的，那么函数名为 *gsl_linalg_complex_LU_decomp*，下面的函数同理。

LU 分解还可以实现很多其他的功能，例如

$$int\ gsl\_linalg\_LU\_solve(gsl\_matrix * LU,\ gsl\_permutation * p,\ gsl\_vector * b, gsl\_vector * x)$$

可以使用 LU 分解来解方程组 Ax=b（LUx=b）。常用的功能还有求逆矩阵、求行列式、求行列式符号（均为使用 LU 分解）。

$$int\ gsl\_linalg\_LU\_invert(const\ gsl\_matrix * LU,\ const\ gsl\_permutation * p,\ gsl\_matrix * inverse)$$

$$double\ gsl\_linalg\_LU\_det(gsl\_matrix * LU,\ int\ signum)$$

$$int\ gsl\_linalg\_LU\_sgndet(gsl\_matrix * LU,\ int\ signum)$$

[1] Personcom，https://blog.csdn.net/ouc_09dx/article/details/19158435

5.3 特征值

这部分内容需要用到 *gsl_eigen.h*。
求不同类型的矩阵特征值只需要调用相应函数，比如对于对称矩阵有

$$int\ gsl\_eigen\_symmv(gsl\_matrix * A, gsl\_vector * eval, gsl\_matrix * evec, gsl\_eigen\_symmv\_workspace * w)$$

函数名只需要根据对应的矩阵类型，在编译器的引导下使用。

6 数值积分

这部分内容一般需要使用 *gsl_integration.h*。
GSL 提供了一系列数值积分的函数，因为名字太长所以一般用缩写表示，缩写字母意义如下。

- Q - quadrature routine
- N - non-adaptive integrator
- A - adaptive integrator
- G - general integrand (user-defined)
- W - weight function with integrand
- S - singularities can be more readily integrated
- P - points of special difficulty can be supplied
- I - infinite range of integration
- O - oscillatory weight function, cos or sin
- F - Fourier integral
- C - Cauchy principal value

以一般性的自适应积分（QAG）为例

$$int\ gsl\_integration\_qag(gsl\_function * f, double\ a, double\ b, double\ epsabs, double\ epsrel,$$

$$size\_t\ limit, int\ key, gsl\_integration\_workspace * workspace, double * result, double * abserr)$$

参数意义分别是：函数，积分上下限，绝对和相对误差，积分区间个数上限，积分方式选择（使用哪一种高斯积分，key 的值可以选择整数 1-6，在数值计算课程应用范围内区别不大），积分储存空间声明，积分结果，积分结果绝对误差。给出例程如下。

```
1  gsl_integration_workspace * w//声明空间
2  = gsl_integration_workspace_alloc (1000);
3    double result, error;
4    double expected = -4.0;
5    double alpha = 1.0;
6    gsl_function F;
7    F.function = &f;
8    F.params = &alpha;
9    gsl_integration_qags
10   (&F, 0, 1, 0, 1e-7, 1000/*声明了1000，所以最多设为1000*/,
11    w, &result, &error);
12 printf ("result = %.18f\n", result);
13 printf ("exact result = %.18f\n", expected);
```

使用 GSL 提供的数值积分，既可以对自己的程序起到验证性作用，又可以通过比较不同类型积分得到的绝对误差来比对不同方法的特性，不失为一种好的拓展学习的方式。

7 插值

这部分内容一般需要使用 *gsl_interp.h* 和 *gsl_spline.h*。

对于一位差值，GSL 提供很多函数如线性、样条和牛顿，牛顿我们在之前的多项式部分已经提到。我们仍旧用一个例程来说明。

```
int i;
double xi, yi, x[10], y[10];//给出差值点列
for (i = 0; i < 10; i++) {
x[i] = i + 0.5 * sin (i);
y[i] = i + cos (i * i);
}
  {
    gsl_interp_accel *acc
      = gsl_interp_accel_alloc ();//声明差值和加速所需要的储存空间
    gsl_spline *spline
      = gsl_spline_alloc (gsl_interp_cspline, 10);
    gsl_spline_init (spline, x, y, 10);//初始化
for (xi = x[0]; xi < x[9]; xi += 0.01) {
        yi = gsl_spline_eval (spline, xi, acc);//计算差值
printf ("%g %g\n", xi, yi); }
    gsl_spline_free (spline);//释放空间
    gsl_interp_accel_free (acc);
  }
```

不过事实上差值的部分使用 C 的意义并不大，因为函数只会返回差值点处的值。而且如果想要做出差值图像，需要找到 a.out，又需要学习更多知识，所以此处略微提及。

A 多项式操作

```
double *plus(double p1[],int l1,double p2[],int l2){//加函数
    int l=gsl_max(l1, l2);
    if(l>l1){
        double *temp;
        int tem;
        temp=p2;
        p2=p1;
        p1=temp;
        tem=l2;
        l2=l1;
        l1=tem;
    }
    double result[l];
```

```
    for(int i=0;i<l;i++){result[i]=p1[i];}
    for(int i=0;i<l2;i++){result[i]+=p2[i];}
    while(result[l-1]==0&&l>0){l=l-1;}
    double *p;
    p=(double*)malloc(sizeof(double)*l);
    return p;
}
```

```
double *minus(double p1[],int l1,double p2[],int l2){//减函数
    int l=gsl_max(l1, l2);
    int sig=1;
    if(l>l1){
        double *temp;
        int tem;
        temp=p2;
        p2=p1;
        p1=temp;
        tem=l2;
        l2=l1;
        l1=tem;
        sig=-1;
    }
    double result[l];
    for(int i=0;i<l;i++){result[i]=p1[i];}
    for(int i=0;i<l2;i++){result[i]+=p2[i];
    result[i]=result[i]*sig;}
    while(result[l-1]==0&&l>0){l=l-1;}
    double *p;
    p=(double*)malloc(sizeof(double)*l);
    return p;
}
```

```
double *multiple(double p1[],int l1,double p2[],int l2){//乘函数
    int l=l2+l2-1;
    double *p;
    p=(double*)malloc(sizeof(double)*l);
    for(int i=0;i<l;i++){p[i]=0;}
    for(int i=0;i<l;i++){
        for(int j=0;j<l1;j++){
            for(int k=0;k<l2;k++){
                if(j+k==i){
                    p[i]+=p1[j]*p2[k];
                }
            }
        }
    }
```

```
    return p;
}
```

```
double *generate(double root[],double n){//通过根生成多项式
    double *c;
    c=(double*)malloc(sizeof(double)*(n+1));
        c[0]=-root[0];c[1]=1;
    for(int i=2;i<n+1;i++){
        double temp[2];
        temp[1]=1;temp[0]=-root[i-1];
        c=multiple(c,i,temp,2);
    }
    return c;
}
```

为了防止丢失空间，以上四个函数务必在使用之后将使用 malloc 的空间释放，示例如下。

```
double *c;
    c=generate(a,2);
    //检查malloc是否成功，可以不写
    if(c==NULL){
        perror("error");
        exit(1);
    }
    释放空间
    free(c);
```

B 矩阵乘法

```
gsl_matrix *gsl_matrix_mul(gsl_matrix *a,gsl_matrix *b){//矩阵乘法
        gsl_matrix *c=gsl_matrix_alloc(a->size1,b->size2);
        for (size_t i=0;i<a->size1;i++){
        for (size_t j=0;j<b->size2;j++){
                double sum=0.0;
                for (size_t k=0;k<b->size1;k++){
                sum+=gsl_matrix_get(a,i,k)*gsl_matrix_get(b,k,j);}
                gsl_matrix_set(c,i,j,sum);
                }
                return c;
        }
```