Scientific Programming in C
XII. GNU Scientific Library

Susi Lehtola

4 December 2012
Introduction

The GNU Scientific Library is an extensive collection of mathematical routines – over 1000 functions.

- Complex Numbers
- Roots of Polynomials
- Special Functions
- Vectors and Matrices
- Permutations
- Sorting
- BLAS Support
- Linear Algebra
- Eigensystems
- Fast Fourier Transforms
- Quadrature
- Random Numbers
- Quasi-Random Sequences
- Random Distributions
- Statistics
- Histograms
- N-Tuples
- Monte Carlo Integration
- Simulated Annealing
- Differential Equations
- Interpolation
- Numerical Differentiation
- Chebyshev Approximation
- Series Acceleration
- Discrete Hankel Transforms
- Root-Finding
- Minimization
- Least-Squares Fitting
- Physical Constants
- IEEE Floating-Point
- Discrete Wavelet Transforms
- Basis splines
GSL has a very good manual that you can read, e.g., online http://www.gnu.org/software/gsl/manual/html_node/
License

GSL is licenced under the GNU General Public License (GPL).

Q: If I write an application which uses GSL, am I forced to distribute that application?
A: No. The license gives you the option to distribute your application if you want to. You do not have to exercise this option in the license.

Q: If I wanted to distribute an application which uses GSL, what license would I need to use?
A: The GNU General Public License (GPL).

http://www.gnu.org/software/gsl/
Installation

GSL is available on practically all *nix distributions.

- Red Hat / Fedora distributions: install the gsl-devel package
- in Debian / Ubuntu: install libgsl0-dev

GSL is already installed on, e.g., punk and mutteri.
Linking to GSL is done with `-lgsl`.

- GSL also contains linear algebra routines
- May need to add a BLAS library with a CBLAS interface to the link command, e.g.
  - `-lgsl -L/usr/lib64/atlas -lcblas -latlas` to use the ATLAS library
  - `-lgsl -lgslcblas` to use GSL’s CBLAS library (slower)

If you use linear algebra routines, use an optimized BLAS library such as ATLAS or OpenBLAS (recommended).
Random number generation

Quite often one needs random numbers in computational science

- initial speeds for molecular dynamics
  - sampled from Maxwell-Boltzmann distribution
- optimization and parameter fitting
- Monte Carlo integration
  - more efficient than grid-based methods in $\mathbb{R}^n$ when $n > 4$
- Monte Carlo simulations
  - drift of pollution in the atmosphere
  - forest fires
  - Quantum Monte Carlo
Random number generation, cont’d

Although the C standard library contains a function `rand()` that generates random numbers, it cannot be (always) trusted.

- repeat interval?
- maximum value $N$ generated?
  - $N^{-1}$ gives the density of points on the interval $[0, 1)$
- correlation of sequential numbers?
  - when plotted in $n$ dimensions can form planes
  - or even point clouds (really bad generator)

Monte Carlo simulations can easily go through billions of random numbers. Some implementations of `rand()` are extremely deficient – don’t use `rand()` except for trivial purposes.
Random number generation with GSL

Although some random number generators (RNG) are easy to implement in your own code, others are very complicated.
Random number generation with GSL

Although some random number generators (RNG) are easy to implement in your own code, others are very complicated.

Don’t reinvent the wheel - use a library.
Random number generation with GSL

Although some random number generators (RNG) are easy to implement in your own code, others are very complicated.

Don’t reinvent the wheel - use a library.

GSL provides a uniform interface for lots of RNGs, e.g.

- Mersenne twister
- **RANLUX, RANLXS**
- **RANMAR**
- Park-Miller
- Knuth’s algorithms

Changing the RNG can be done with a single line of code!
Random number generation with GSL, example

```c
#include <stdio.h>
#include <gsl/gsl_rng.h>

int main (void) {
    const int n = 5;
    gsl_rng * r;
    int i;
    /* Use Park–Miller generator */
    r = gsl_rng_alloc (gsl_rng_minstd);
    for (i=0;i<n;i++) {
        double u = gsl_rng_uniform (r);
        printf ("%.5f\n", u);
    }

    gsl_rng_free (r);
    return 0;
}
```
Random number generation with GSL, example cont’d

Running the program gives

```
$ ./a.out
0.00001
0.13154
0.75561
0.45865
0.53277
```
Random number generation with GSL, example cont’d

Running the program gives

$ ./a.out
0.00001
0.13154
0.75561
0.45865
0.53277

A re-run gives

$ ./a.out
0.00001
0.13154
0.75561
0.45865
0.53277

i.e., the identical result.
Pseudorandom numbers

The numbers are pseudorandom – although they satisfy a lot of properties required for random numbers, by observing the sequence long enough it is possible to calculate the next number beforehand.
Pseudorandom numbers

The numbers are pseudorandom – although they satisfy a lot of properties required for random numbers, by observing the sequence long enough it is possible to calculate the next number beforehand.

Random numbers are used also in cryptography. Here the quality of the random numbers is of utmost importance – a small amount of numbers is required, but they have to be very hard to crack.
In computational science lots of random numbers are needed, so the speed of obtaining the numbers is more important than a slightly better "randomness".

- A RNG that works for one application might not work as well for another
- Need to run the simulations with many RNGs and check if the results differ
Changing the RNG can also be done without any changes in the code by using environment variables.

```c
#include <stdio.h>
#include <gsl/gsl_rng.h>

int main(void) {
    gsl_rng * r;

    /* Initialize environment */
    gsl_rng_env_setup();
    /* Initialize generator */
    r = gsl_rng_alloc (gsl_rng_default);

    printf ("generator type: %s\n", gsl_rng_name (r));
    printf ("seed = %lu\n", gsl_rng_default_seed);
    printf ("first value = %lu\n", gsl_rng_get (r));

    gsl_rng_free (r);
    return 0;
}
```
Random number generation with GSL, cont’d

The default is the Mersenne Twister algorithm, which has a period of $2^{19937} - 1 \approx 4.31 \cdot 10^{6001}$

```bash
$ ./a.out
generator type: mt19937
seed = 0
first value = 4293858116
```
Random number generation with GSL, cont’d

The default is the Mersenne Twister algorithm, which has a period of $2^{19937} - 1 \approx 4.31 \cdot 10^{6001}$

$ ./a.out$

generator type: mt19937
seed = 0
first value = 4293858116

.. but it can be changed using environment variables:

$ GSL_RNG_TYPE="minstd" ./a.out
GSL_RNG_TYPE=minstd

generator type: minstd
seed = 0
first value = 16807

which would use the Park-Miller minimal standard generator, with a period of $2^{31} - 1 \approx 2.15 \cdot 10^9$
Matrices, vectors & linear algebra

GSL has functionality for normal, dense matrices.

- not as full as full BLAS & LAPACK
Matrices, vectors & linear algebra

GSL has functionality for normal, dense matrices.
  ▶ not as full as full BLAS & LAPACK

...but it is native C code!

Can also interface with (vendor) optimized CBLAS library.
Matrices, vectors & linear algebra cont’d

Matrices and vectors are defined as structs in GSL.

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

typedef struct {
    size_t size;
    size_t stride;
    double * data;
    gsl_block * block;
    int owner;
} gsl_vector;

typedef struct {
    size_t size1;
    size_t size2;
    size_t tda;
    double * data;
    gsl_block * block;
    int owner;
} gsl_matrix;
Matrix-vector example

```c
#include <stdio.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>

int main(void) {
    const int N=100;
    int i, j;

    gsl_matrix *A;
    gsl_vector *x, *y;

    /* Initialize A_{ij} = 1/(i+j) */
    A=gsl_matrix_alloc(N,N);
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            gsl_matrix_set(A, i, j, 1.0/(i+j+2.0));

    /* Initialize x_i = i */
    x=gsl_vector_alloc(N);
    for (i=0; i<N; i++)
        gsl_vector_set(x, i, i+1.0);
```
Matrix-vector example, cont’d

`/* Allocate y. */`
y = gsl_vector_alloc(N);

`/* Compute y = Ax */`
y = 1.0 A x + 0.0 y ;
gsl_blas_dgemv(CblasNoTrans, 1.0, A, x, 0.0, y);

`/* Print y */`
for (i = 0; i < N; i++)
    printf("%.8f
", gsl_vector_get(y, i));

gsl_vector_free(x);
gsl_vector_free(y);
gsl_matrix_free(A);
return 0;
};
Matrix-vector example, cont’d

$ ./a.out
95.803
92.586
89.850
87.428
85.237

(clip)

31.939
31.733
31.529
31.329
31.130
30.935
Eigenvectors and eigenvalues

Let's find the eigenvectors and eigenvalues of $A_{ij} = 1/(i + j)$.

```c
#include <stdio.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_eigen.h>

int main(int argc, char **argv) {
    /* Size and indices */
    int N, i, j;
    /* Matrix and its eigendecomposition */
    gsl_matrix *A;
    gsl_matrix *Avec;
    gsl_vector *Aval;
    /* Workspace */
    gsl_eigen_symmv_workspace *wrk;

    if (argc!=2) {
        printf("Usage: %s N\n", argv[0]);
        return 1;
    }

    /* Read size of problem */
    N = atoi(argv[1]);
```
Eigenvectors and eigenvalues, cont’d

/* Initialize A_{ij} = 1/(i+j) */
A = gsl_matrix_alloc(N, N);
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        gsl_matrix_set(A, i, j, 1.0/(i+j+2.0));

/* Compute eigendecomposition */
Aval = gsl_vector_alloc(N);
Avec = gsl_matrix_alloc(N, N);
wrk = gsl_eigen_symmv_alloc(N);
gsl_eigen_symmv(A, Aval, Avec, wrk);

/* Print eigenvectors and eigenvalues */
for (i = 0; i < N; i++) {
    printf("%3i: th eigenvector, eigenvalue = ", i+1,
            gsl_vector_get(Aval, i));
    for (j = 0; j < N; j++)
        printf("\n%e", gsl_matrix_get(Avec, j, i));
    printf("\n");
}
Eigenvectors and eigenvalues, cont’d

/* Clean up */
gsl_eigen_symmv_free(wrk);
gsl_vector_free(Aval);
gsl_matrix_free(Avec);
gsl_matrix_free(A);

return 0;
};
Running the code gives

<table>
<thead>
<tr>
<th>Command</th>
<th>1:th eigenvector, eigenvalue</th>
<th>2:th eigenvector, eigenvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>./a.out 1</td>
<td>5.000000e−01: 1.000000e+00</td>
<td></td>
</tr>
<tr>
<td>./a.out 2</td>
<td>7.310002e−01: 8.219256e−01</td>
<td>5.695948e−01</td>
</tr>
<tr>
<td>./a.out 3</td>
<td>8.751151e−01: 7.431359e−01</td>
<td>5.278434e−01</td>
</tr>
<tr>
<td></td>
<td>4.090495e−02: 6.387867e−01</td>
<td>−3.766118e−01</td>
</tr>
<tr>
<td></td>
<td>6.466589e−04: 1.992501e−01</td>
<td>−7.612784e−01</td>
</tr>
</tbody>
</table>
GSL contains routines for calculation of definite integrals of the form

\[ I = \int_a^b f(x)w(x) \, dx \]

where \( w(x) \) is a weight function (for general integrals \( w(x) = 1 \)).

The routines (try to) calculate the wanted integral within the given absolute and relative precisions \( \epsilon_a \) and \( \epsilon_r \):

\[ \left| \tilde{I} - I \right| \leq \max\{\epsilon_a, \epsilon_r |I|\} \]

where \( \tilde{I} \) is the estimated value of the integral.
For general functions $w(x) = 1$ the integration is done using Gauss-Kronrod rules:

$$
I = \int_a^b f(x) \, dx \\
\approx \sum_{i=1}^n w_i f(x_i)
$$

where the weights $w_i$ and the points $x_i$ are obtained by demanding that the rule is exact for polynomials of the highest possible order ($n$-point Gaussian rule exact for polynomials up to $2n - 1$:th degree).
Integration, general

For general functions \( w(x) = 1 \) the integration is done using Gauss-Kronrod rules:

\[
I = \int_{a}^{b} f(x) \, dx \\
\approx \sum_{i=1}^{n} w_i f(x_i)
\]

where the weights \( w_i \) and the points \( x_i \) are obtained by demanding that the rule is exact for polynomials of the highest possible order (\( n \)-point Gaussian rule exact for polynomials up to \( 2n - 1 \):th degree).

The Kronrod extension adds \( n + 1 \) points to the \( n \) point rule, so that the resulting rule is of order \( 3n + 1 \). The difference between the \( n + 1 \) and \( 2n + 1 \) point rules gives an error estimate for the integral.
Integration over weight functions

Integration over weight functions applies the same principle: the integrand is expanded as a polynomial that can be integrated exactly.

This is done using the Clenshaw-Curtis quadrature rules.
Integration examples

See documentation and examples at
Multidimensional minimization

Another very useful feature of GSL are the multidimensional minimizers.

For functions with (analytical) derivatives:
- Steepest descent
- Fletcher-Reeves conjugate gradient
- Polak-Ribiere conjugate gradient
- Broyden-Fletcher-Goldfarb-Shanno (BFGS)

For functions without derivatives:
- Simplex algorithm of Nelder and Mead

Once again general routines that use `void *`
Multidimensional minimization, cont’d

See overview and examples at