PHY1610: Fitting and Fourier Transforms

Ramses van Zon, Marcelo Ponce

March 4, 2021

SciNet

Today's class

Today we will discuss:

- Fitting
- Fourier transforms

1

Fitting

Fitting data

- Common task in science is to fit data to a theoretical function.
- Much of machine learning is a form of fitting.
- Even for a simple case, how are we going about this numerically?

First let's generate some data

```
///@file lmdata.h
#ifndef LMDATAH
#define LMDATAH
#include <rarray>
#include <utility>
std::pair<rvector<double>,rvector<double>>
 lmdata(int n, double xmin, double xmax, double a,
        double b, double sigma, unsigned long s);
#endif
```

```
///@file lmdata.cpp
#include "lmdata.h"
#include <random>
std::pair<rvector<double>,rvector<double>>
 lmdata(int n, double xmin, double xmax, double a,
        double b, double sigma, unsigned long s)
{
    std::mt19937 rng(s);
    std::normal_distribution<> gaussian(0, sigma);
    rvector<double> x = linspace(xmin, xmax, n);
    rvector<double> y(n);
    for (int i = 0; i < n; i++) {
        y[i] = a*x[i] + b + gaussian(rng);
    }
    return {x,y};
}
```

```
///@file fitwgsl.cpp
#include "lmdata.h"
#include <fstream>
#include <gsl/gsl_fit.h>

int main() {
    int n = 10;
    double xmin=0.0, xmax=2.0, a=2.5, b=1.0, sigma=0.3;
    unsigned long int seed=13;
    std::pair<rvector<double>,rvector<double>> xypair;
    xypair = lmdata(n,  xmin, xmax, a, b, sigma, seed);
    rvector<double> x=xypair.first;
    rvector<double> y=xypair.second;
    // fit (x,y) to a linear model, write result to file
}
```

```
$ g++ -c -g -O2 -std=c++14 lmdata.cpp -o lmdata.o
$ g++ -c -g -O2 -std=c++14 fitwgsl.cpp -o fitwgsl.o
$ g++ -g lmdata.o fitwgsl.o -o fitwgsl -lgsl -lgslcblas
$ ./fitwgsl
```

GSL fit details

From the GSL docs:

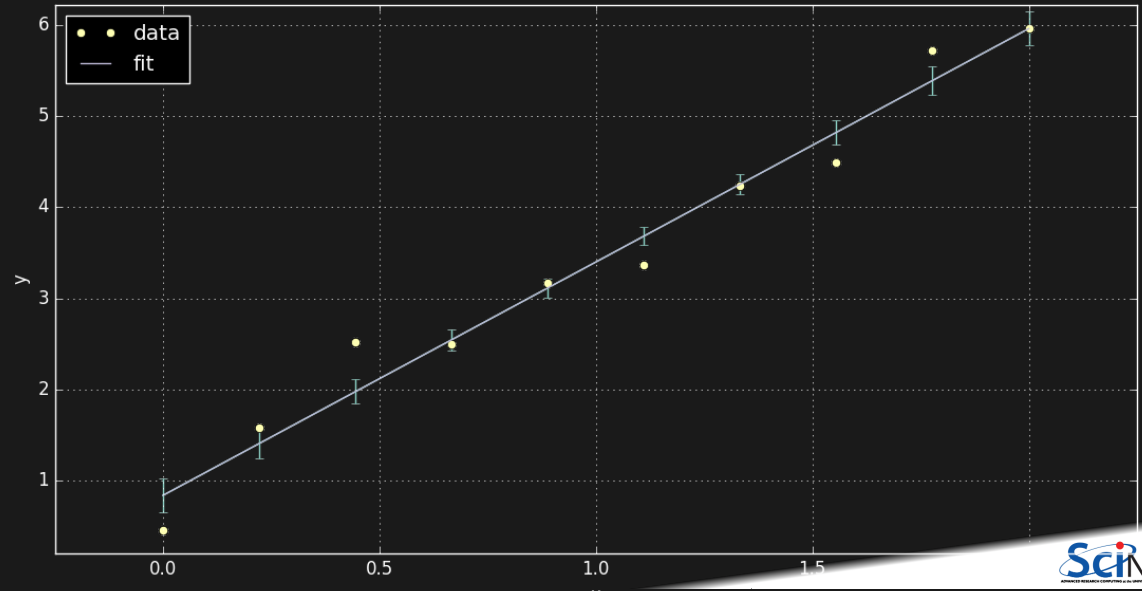
```
40.2.1 Linear regression with a constant term

The functions described in this section can be used to perform
least-squares fits to a straight line model, Y(c,x) = c_0 + c_1 x.

-- Function: int gsl_fit_linear (const double * x, const
       size_t xstride, const double * y, const size_t ystride,
       size_t n, double * c0, double * c1, double * cov00, double
       * cov01, double * cov11, double * sumsq)

This function computes the best-fit linear regression coefficients
('c0', 'c1') of the model Y = c_0 + c_1 X for the dataset ('x',
'y'), two vectors of length 'n' with strides 'xstride' and
'ystride'.  The errors on 'y' are assumed unknown so the
variance-covariance matrix for the parameters ('c0', 'c1') is
estimated from the scatter of the points around the best-fit line
and returned via the parameters ('cov00', 'cov01', 'cov11').  The
sum of squares of the residuals from the best-fit line is returned
in 'sumsq'.  Note: the correlation coefficient of the data can be
computed using gsl_stats_correlation(), it does not
depend on the fit.
```

(https://www.gnu.org/software/gsl/doc/html)

```
$ g++ -c -g -O2 -std=c++14 lmdata.cpp -o lmdata.o
$ g++ -c -g -O2 -std=c++14 fitwgsl.cpp -o fitwgsl.o
$ g++ -g lmdata.o fitwgsl.o -o fitwgsl -lgsl -lgslcblas
$ ./fitwgsl
a=2.32208
b=1.26666
```

```cpp
// fitwgsl.cpp
#include "lmdata.h"
#include <fstream>
#include <iostream>
#include <gsl/gsl_fit.h>
int main() {
    int n = 10;
    double xmin=0.0, xmax=2.0, a=2.5, b=1.0, sigma=0.3;
    unsigned long int seed=13;
    std::pair<rvector<double>,rvector<double>> xypair;
    xypair = lmdata(n,  xmin, xmax, a, b, sigma, seed);
    rvector<double> x=xypair.first, y=xypair.second;
    // fit (x,y) to a linear model
    double c0, c1, cov00, cov01, cov11, rss;
    gsl_fit_linear(x.data(), 1, y.data(), 1,
                   x.size(), &c0, &c1,
                   &cov00, &cov01, &cov11, &rss);
    std::cout << "a=" << c1 << std::endl;
    std::cout << "b=" << c0 << std::endl;
    // estimate some points
    std::ofstream out("fit.dat");
    out << "# x y yfit yerr" << std::endl;
    for (int i = 0; i < x.size(); i++) {
        double y, yerr;
        gsl_fit_linear_est(x[i], c0, c1, cov00,
                           cov01, cov11, &y, &yerr);
        out << x[i] << " " << y[i] << " "
            << y << " " << yerr << std::endl;
    }
}
```

Result



What did the GSL actually do?

- We've got data (x and y pairs)
- It's assumed that there is an underlying linear relation between x and y, with parameters a and b, plus noise.

$$y = f(x; a, b) + \varepsilon = ax + b + \varepsilon$$

- We need estimates for a and b.
- With these estimates, one could predict y values given any other x values ("machine learning").
- If noise is normally distributed with the same variance independent of x, one can use the residuals, i.e. the difference between the observed and the predicted values.

$$\mathrm{RSS}(a, b) = [y_i - f(x_i; a, b)]^2$$

- GSL maximizes the likelihood of the data by minimizing the sum of the square of residuals:

$$a^*, b^* = \mathrm{argmin}_{a,b}\, \mathrm{RSS}(a, b)$$

- Model is linear in parameters = Ordinary Least Squares

More components, non-linear fitting, ...

- Can generalize to incorporate error estimates in y.
- Can generalize to incorporate error estimates in x.
- Generalizes to forms are non-linear in x, but still linear in parameters.
- Be careful using polynomial fits: easy to overfit.
- For non-linear parameter forms, can still try to minimize least squares: need solving.
- Can use different "cost functions" (RRS is sensitive to outliers).

Least-squares and linear algebra

- If our model f is linear in B parameters β_j, i.e.:

$$f(x) = \sum_{j=1}^{B} \beta_j f_j(x) \qquad \text{(e.g. } B = 2 \; f_1 = 1,\; f_2 = x,\; \beta_1 = b,\; \beta_2 = a\text{)}$$

- Given N data points (x_i, y_i), we need to minimize:

$$\sum_{i=1}^{N} \| y_i - \sum_{j=1}^{B} f_j(x_i)\beta_j \|^2$$

- View $f_j(x_i)$ as a matrix with components F_{ji}.

  F has B rows and N columns.

- Taking the derivative w.r.t β_j gives:

$$FF^T\beta = Fy$$

- Solving for β is linear algebra! We saw this already: (C)BLAS and LAPACK(E)!

Least-squares and linear algebra

```cpp
///@file fitlapack.cpp
#include <iostream>
#include <cblas.h>
#include <lapacke.h>
#include "lmdata.h"
int main()
{
    int n = 10;
    double xmin=0.0, xmax=2.0, a=2.5, b=1.0, sigma=0.3;
    unsigned long int seed=13;
    std::pair<rvector<double>,rvector<double>> xypair;
    xypair = lmdata(n,  xmin, xmax, a, b, sigma, seed);
    rvector<double> x=xypair.first, y=xypair.second;
    // fit (x,y) to a linear model
    int nterms = 2;
    rmatrix<double> F(nterms,n);
    rmatrix<double> FFt(nterms,nterms);
    rvector<double> Fy(nterms);
    int ipiv[nterms];
    for (int i = 0; i < n; i++) {
        F[0][i] = 1.0;
        F[1][i] = x[i];
    }
    cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasTrans,
                nterms, nterms, n,
                1.0, F.data(), n, F.data(), n,
                0.0, FFt.data(), nterms);
    cblas_dgemv(CblasRowMajor, CblasNoTrans,
                nterms, n,
                1.0, F.data(), n,
                y.data(), 1,
                0.0, Fy.data(), 1);
    rvector<double> resultcoef = Fy.copy();
    LAPACKE_dgesv(LAPACK_ROW_MAJOR, nterms, 1,
                  FFt.data(), nterms,
                  ipiv, resultcoef.data(), 1);
    std::cout << "a=" << resultcoef[1] << std::endl;
    std::cout << "b=" << resultcoef[0] << std::endl;
}
```

```
$ g++ -c -g -O2 -std=c++14 fitlapack.cpp -o fitlapack.o
$ g++ -g lmdata.o fitlapack.o -o fitlapack -lopenblas
$ ./fitlapack
a=2.32208
b=1.26666
```

Least-squares for frequency analysis

- Your data may be a signal of which you only want to get rid of higher frequencies (noise).
- Fitting to periodic functions may come to mind.
- E.g.

$$y = \beta_1 \sin(\omega_1 x) + \beta_2 \sin(\omega_2 x) + \ldots$$

- Linear in β_i, so we could do least squares.
- However, this can lead to very oscillitory behavior to fit the data.
- There's a better way to do this.

2

(Discrete) Fourier Transform

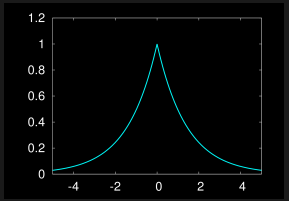
Fourier Transform

In this part of the lecture, we will discuss:

- The Fourier transform,
- The discrete Fourier transform
- The fast Fourier transform
- Examples using the FFTW library

Fourier Transform recap

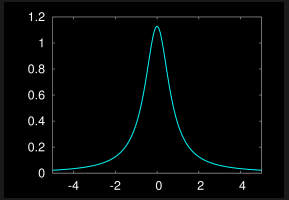
- Let f be a function of some variable x.



$f(x) = e^{-|x|}$

- Transform to a function $\hat{f}$ of k:

$$\hat{f}(k) \propto \int f(x)\, e^{\pm i\, k\cdot x}\, dx$$



$f(x) = (1 + k^2)^{-1}$

- Inverse transformation:

$$f(x) \propto \int \hat{f}(k)\, e^{\mp i\, k\cdot x}\, dk$$

Fourier Transform

- Fourier made the claim that any function can be expressed as a harmonic series.
- The FT is a mathematical expression of that.
- Constitutes a linear (basis) transformation in function space.
- Transforms from spatial to wavenumber, or time to frequency, etc.
- Constants and signs are just convention.*

* some restritions apply.

Discrete Fourier Transform

C. F. Gauss

- Given a set of n function values on a regular grid:

$$f_j = f(j\Delta x)$$

- Transform to n other values

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j \, e^{\pm 2\pi i \, j \, k/n}$$

- Easily back-transformed:

$$f_j = \frac{1}{n} \sum_{j=0}^{n-1} \hat{f}_k \, e^{\mp 2\pi i \, j \, k/n}$$

- Solution is periodic: $f_{-k} = f_{n-k}$. You run the risk of aliasing, as k is equivalent to $k + \ell n$. Cannot resolve frequencies higher than $k = n/2$ (Nyquist).

Slow Fourier Transform

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j \, e^{\pm 2\pi i \, j \, k/n}$$

- Discrete fourier transform is a linear transformation.
- In particular, it's a matrix-vector multiplication.
- Naively, costs $\mathcal{O}(n^2)$. Slow!

Slow DFT

```
#include <complex>
#include <rarray>
#include <cmath>

typedef std::complex<double> complex;

void fft_slow(const rvector<complex>& f, rvector<complex>& fhat, bool inverse)
{
  int n = fhat.extent(0);
  double v = (inverse?-1:1)*2*M_PI/n;
  for (int k=0; k<n; k++)
  {
    fhat[k] = 0.0;
    for (int m=0; m<n; m++) {
      fhat[k] += complex(cos(v*k*m), sin(v*k*m)) * f[m];
    }
  }
}
```

Even Gauss realized $\mathcal{O}(n^2)$ was too slow and came up with ...

Fast Fourier Transform

- Derived in partial form several times before and even after Gauss, because he'd just written it in his diary in 1805 (published later).
- Rediscovered (in general form) by Cooley and Tukey in 1965.

Basic idea

- Write each n-point FT as a sum of two $\frac{n}{2}$ point FTs.
- Do this recursively $^2\log n$ times.
- Each level requires $\sim n$ computations: $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$.
- Could as easily divide into 3, 5, 7, ... parts.

Fast Fourier Transform: How is it done?

- Define $\omega_n = e^{2\pi i/n}$.
- Note that $\omega_n^2 = \omega_{n/2}$.
- DFT takes form of matrix-vector multiplication:

$$\hat{f}_k = \sum_{j=0}^{n-1} \omega_n^{kj} f_j$$

- With a bit of rewriting (assuming n is even):

$$\hat{f}_k = \underbrace{\sum_{j=0}^{n/2-1} \omega_{n/2}^{kj} f_{2j}}_{\text{FT of even samples}} + \omega_n^k \underbrace{\sum_{j=0}^{n/2-1} \omega_{n/2}^{kj} f_{2j+1}}_{\text{FT of odd samples}}$$

- Repeat, until the lowest level (for $n = 1$, $\hat{f} = f$).
- Note that a fair amount of shuffling is involved.

Fast Fourier Transform: Already done!

We've said it before and we'll say it again: Do not write your own: use existing libraries!

Why?

- Because getting all the pieces right is tricky;
- Getting it to compute fast requires intimate knowledge of how processors work and access memory;
- Because there are libraries available.

  Examples:

  - FFTW3 (Faster Fourier Transform in the West, version 3)
  - Intel MKL
  - IBM ESSL
- Because you have better things to do.

Example of using a library: FFTW

Rewrite of previous (slow) ft to a fast one using fftw

```
#include <complex>
#include <rarray>
#include <fftw3.h>

typedef std::complex<double> complex;

void fft_fast(const rvector<complex>& f, rvector<complex>& fhat, bool inverse)
{
  int n = f.size();
  fftw_plan p = fftw_plan_dft_1d(n,
                  (fftw_complex*)f.data(), (fftw_complex*)fhat.data(),
                  inverse?FFTW_BACKWARD:FFTW_FORWARD,
                  FFTW_ESTIMATE);
  fftw_execute(p);
  fftw_destroy_plan(p);
}
```

Inverse DFT

- Inverse DFT is similar to forward DFT, up to a normalization: almost just as fast.

$$f_j = \frac{1}{n} \sum_{k=0}^{n-1} \hat{f}_k \, e^{\mp 2\pi i \, j \, k/n}$$

  Many implementations (almost all in fact) leave out the 1/n normalization.

- FFT allows quick back-and-forth between x and k domain (or e.g. time and frequency domain).
- Allows parts of the computation and/or analysis to be done in the most convenient or efficient domain.

Working example

- Create a 1d input signal: a discretized $sinc(x) = \sin(x)/x$ with 16384 points on the interval [-30:30].
- Perform forward transform
- Write to standard out
- Compile, and linking to fftw3 library.
- Continous FT of sinc(x) is the rectangle function:

$$\text{rect}(f) = \begin{cases} 0.5 & \text{if } \|k\| \leq 1 \\ 0 & \text{if } \|k\| > 1 \end{cases}$$

  up to a normalization.
- Does it match?

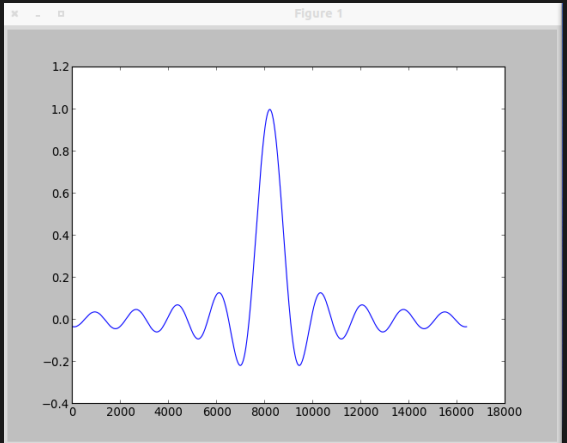
Code for the working example

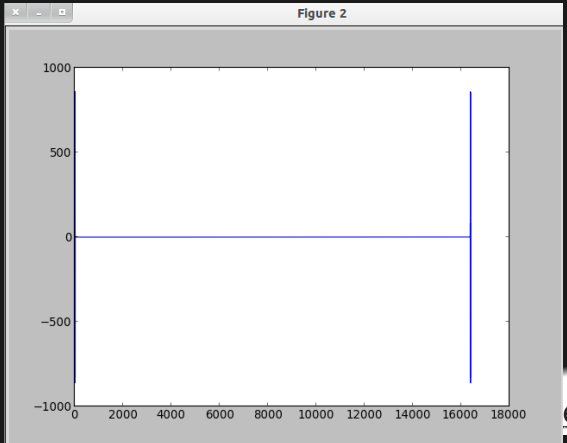
```
//sincfftw.cpp
#include <iostream>
#include <complex>
#include <rarray>
#include <fftw3.h>
typedef std::complex<double> complex;
int main() {
  const int n = 16384;
  rvector<complex> f(n), fhat(n);
  for (int i=0; i<n; i++) {
    double x = 60*(i/double(n)-0.5); // x-range from -30 to 30
    if (x!=0.0) f[i] = sin(x)/x; else f[i] = 1.0;
  }
  fftw_plan p = fftw_plan_dft_1d(n,
                    (fftw_complex*)f.data(), (fftw_complex*)fhat.data(),
                    FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute(p);
  fftw_destroy_plan(p);
  for (int i=0; i<n; i++)
    std::cout << f[i] << "," << fhat[i] << std::endl;
  return 0;
}
```

Compile, link, run, plot

```
$ module load gcc fftw anaconda3
$ g++ -std=c++14 -c -O2 sincfftw.cpp -o sincfftw.o
$ g++ sincfftw.o -o sincfftw -lfftw3
$ ./sincfftw > output.dat
$ ipython --pylab
```

```
>>> data = genfromtxt('output.dat')
>>> plot(data[:,0])
>>> figure()
>>> plot(data[:,2])
```

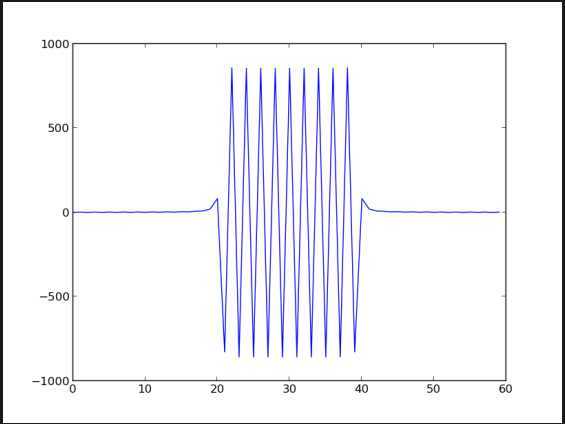




Plots of the output, rewrapped

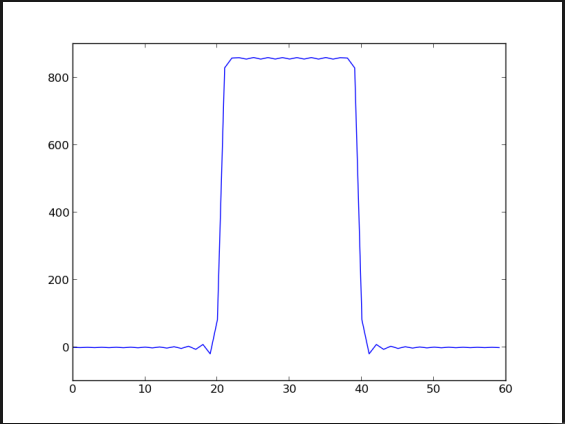
Pick the first and the last 30 points.

```
>>> x1=range(30)
>>> x2=range(len(data)-30,len(data))
>>> y1=data[x1,2]
>>> y2=data[x2,2]
>>> figure()
>>> plot(hstack((y2,y1)))
```



Undo phase factor due to shifting

```
>>> plot(hstack((y2,y1))*array([1,-1]*30)
```



We retrieved our rectangle function!

Notes

- Always create a plan first.
- An fftw_plan contains all information necessary to compute the transform, including the pointers to the input and output arrays.
- Plans can be reused in the program, and even saved on disk!
- When creating a plan, you can have FFTW measure the fastest way of computing dft's of that size (FFTW_MEASURE), instead of guessing (FFTW_ESTIMATE).
- FFTW works with doubles by default, but you can install single precision too.

Multidimensional transforms

In principle a straighforward generalization:

- Given a set of $n \times m$ function values on a regular grid:

$$f_{ab} = f(a\Delta x, b\Delta y)$$

- Transform these to n other values $\hat{f}_{kl}$

$$\hat{f}_{kl} = \sum_{a=0}^{n-1} \sum_{b=0}^{m-1} f_{ab}\, e^{\pm 2\pi i\,(a\,k + b\;l)/n}$$

- Easily back-transformed:

$$f_{ab} = \frac{1}{nm} \sum_{k=0}^{n-1} \sum_{l=0}^{m-1} \hat{f}_{kl}\, e^{\mp 2\pi i\,(a\,k + b\;l)/n}$$

- Negative frequencies: $f_{-k,-l} = f_{n-k,m-l}$.

Multidimensional FFT

- We could successive apply the FFT to each dimension
- This may require transposes, can be expensive.
- Alternatively, could apply FFT on rectangular patches.
- Mostly should let the libraries deal with this.
- FFT scaling still $n \log n$.

Symmetries for real data

- All arrays were complex so far.
- If input f is real, this can be exploited.

$$f_j^* = f_j \leftrightarrow \hat{f}_k = \hat{f}_{n-k}^*$$

- Each complex number holds two real numbers, but for the input f we only need n real numbers.
- If n is even, the transform $\hat{f}$ has real $\hat{f}_0$ and $\hat{f}_{n/2}$, and the values of $\hat{f}_k > n/2$ can be derived from the complex valued $\hat{f}_{0<k<n/2}$: again n real numbers need to be stored.

Symmetries for real data

- A different way of storing the result is in "half-complex storage' '. First, the $n/2$ real parts of $\hat{f}_{0<k<n/2}$ are stored, then their imaginary parts in reversed order.
- Seems odd, but means that the magnitude of the wave-numbers is like that for a complex-to-complex transform.
- These kind of implementation dependent storage patterns can be tricky, especially in higher dimensions.