CS395T: Introduction to Scientific and Technical Computing

Scientific Libraries
BLAS, LAPACK, and FFTW

Instructors

Dr. Victor Eijkhout, Research Scientist, TACC
Dr. Karl W. Schulz, Research Associate, TACC

Scientific Libraries

- BLAS (http://www.netlib.org/blas/)
  - Basic Linear Algebra Subprograms
  - basic operations on vectors and matrices
- LAPACK (http://www.netlib.org/lapack/)
  - Linear Algebra PACKage
  - linear systems solvers (regular and least-squares)
  - eigensolvers
  - sigular value decomposition
- FFTW (http://www.fftw.org/)
  - Fastest Fourier Transform in the West
  - DFT (complex or real to complex)
  - DCT and DST (real to real)

BLAS

- Three levels
  - 1 – Scalar, Vector, and Vector-Vector operations
  - 2 – Matrix-Vector operations
  - 3 – Matrix-Matrix operations
- Originally written in FORTRAN, posted to Netlib
  - uses FORTRAN array storage
- Optimized implementations now common
  - Vendors
    - Intel: MKL (http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm)
    - AMD: ACML (http://developer.amd.com/acml.jsp)
    - IBM: ESSL (http://www-03.ibm.com/systems/p/software/essl.html)
  - Third-Party
    - ATLAS (http://math-atlas.sourceforge.net/) automatic search
    - GotoBLAS (http://www.tacc.utexas.edu/resources/software/#blas) superman programmer approach

LAPACK

- FORTRAN routines for linear algebra
  - uses FORTRAN array storage
- Optimized versions available
  - vendors (same as the BLAS)
    - may not implement every function
    - usually pretty complete
  - third-party
    - less coverage
    - ATLAS implements only certain routines
- Replaces
  - LINPACK and EISPACK
  - uses block operations
  - calls the BLAS where possible

FFTW

- ANSI C code callable from C/C++ and FORTRAN
- Written by a code generator written in OCaml
- Used in
  - signal/image processing
  - spectral methods for PDEs

Matrix Data Layout—C vs. FORTRAN

- FORTRAN arrays
  - indices start at `1`
  - 2-dimensional arrays
    - implemented as contiguous storage
    - stored down the columns
      - perhaps, counter-intuitive
      - stride-1 access goes down a column (i.e. increments *i* first)
  - multi-dimensional arrays
    - contiguous
    - left-most argument is stride-1
- Arrays in C
  - indices start at `0`
  - pointers and array indices are equivalent (usually)
  - i.e., `a[5]` *means* `*(a+5)`

Matrix Data Layout—C vs. FORTRAN

- Multi-dimensional arrays in C/C++ are tricky
- Look like pointers of pointers
  - i.e. `a[3][4]` *can mean* `*(*(a+3)+4)`
  - Actual implementation is as contiguous data
- Can't pass a multi-dimensional array to a subroutine (unlike Fortran, C99)
- Easier to emulate:
  - Instead of `a[3][4]` write `a[3*n+4]`

Matrix Data Layout—C vs. FORTRAN

- To have C and FORTRAN interoperate
  - must use the FORTRAN ordering
    - C is flexible on ordering (pointers to pointers)
    - FORTRAN isn't
    - so the FORTRAN ordering "wins"
  - must use contiguous storage
  - use 1-d "polynomial" indexing in C

Matrix Data Layout

foo.f:

```
program foo
       real*8 a(2,2)
       a(1,1)=1
       a(1,2)=2
       a(2,1)=3
       a(2,2)=4
       call prnt(a,2)
       call addrpoly(a,2)
       call arrnot(a,2)
       stop
       end
```

bar.c:

```
#include <stdio.h>
void prnt_(double *a, int *n)
{
  int i=0;
  for(i=0;i<(*n)*(*n);++i)
    {
      printf("%g\n",a[i]);
    }
}
...
```

```
localhost$ ./foo
1
3
2
4
...
```

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Matrix Data Layout

- Two options in C
  - use an addressing polynomial
    - use the CPP to help ()
    - compiler may not optimize the loops very well
      - may have to do some hand-tuning
  - use multi-dimensional array notation
    - Needs pointers
    - have to transpose the arguments
    - compiler may or may not understand what you're doing
    - No Blas3 available
- Real solution: try to use library for low level primitives
- In C++, check out Boost (http://www.boost.org/) for some options
  - extensions of the STL to numerical work
  - usually quite optimizable by the compiler
  - won't beat the vendor or 3rd-party BLAS in most cases

Matrix Data Layout

bar.c:

```
...
#define A(i,j) a[(i)+(j)*(*n)]
void addrpoly_(double*a,int *n)
{
  int i, j;
  for (i=0; i<(*n); ++i)
    {
      for (j=0; j<(*n); ++j)
        printf("%g ",A(i,j));
      printf("\n");
    }
}
...
```

```
localhost$ ./foo
...
1 2
3 4
...
localhost$
```

Matrix Data Layout

bar.c:

```
...
void arrnot_(double *a, int *n)
{
  double **b;
  int i, j;
  for (i=0;i<(*n);++i)
    b[i]=(a+(i*(*n)));
  for (i=0;i<(*n);++i)
    {
      for (j=0;j<(*n);++j)
        printf("%g ",b[j][i]);
      printf("\n");
    }
}
```

```
localhost$ ./foo
...
1 2
3 4
localhost$
```

BLAS

- Dense matrix operations
  - also includes symmetric (complex Hermitian) , triangular, and banded, but the storage is weird
- Level 1—vector-vector
  - copy
  - dot product
  - scale and shift
  - rotations (plane, Givens, etc.)
- Level 2—matrix-vector
  - matrix-vector product (plus a scale and shift)
  - rank-1 updates
- Level 3—matrix-matrix
  - matrix-matrix product (plus a scale and shift)

BLAS Nomenclature

- Function names are prefixed with a letter corresponding to the precision and type of the input:

  `xDOT` `xAXPY` `xGEMV` `xGEMM`

- Prefixes
  - S, single-precision real
    - `real*4` and `float`
    - ...on most architectures
  - D, double-precision real (`real*8` and `double`)
  - C, single-precision complex (`complex*8` and ...)
  - Z, double-precision complex (`complex*16` and ...)

Complex Data in C/C++

- C99 has a `complex` type qualifier

```
#include <complex.h>
...
double complex a[100];
```

  - defines `I` to represent $(-1.0)^{1/2}$

    ```
    double complex b = 1.0 + 3.0*I;
    ```

  - `complex.h` defines
    - `creal(), cimag(), carg(), cabs(),`
    - `csin(), ccos(), ctan(), cexp(), cpow(),`
    - `...`
- C++ has `std::complex<T>`
- Both are bitwise-equivalent to contiguous storage, i.e.

```
typedef double my_complex[2];
```

Dot Product

$$\rho = \mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y}$$

`res = ddot( n, x, incx, y, incy )`

- `x, y, real*8` arrays length `n (integer*8)`
- `incx, incy` increment for `x, y`
- `res, real*8` return value
- `ddot, real*8` function (don’t forget to declare this!)

Scale a Vector

$$\mathbf{y} = a\mathbf{x} + \mathbf{y}$$

```
call daxpy( n, a, x, incx, y, incy )
```

- `n, x, y, incx, incy` same as for the dot product
- `a, real*8` the scaling factor

Matrix-Vector Product

$$\mathbf{y} = \alpha \mathbf{A} \mathbf{x} + \beta \mathbf{y}$$

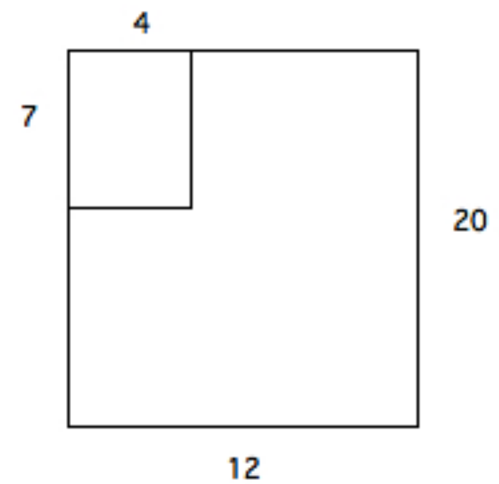
```
call dgemv( trans, m, n, alpha, a, lda, x, incx,
  beta, y, incy )
```

- `m,n, integer*8` matrix dimensions
- `a, real*8 m-by-n` 2-d array
- `x, real*8` vector length `m`
- `y, real*8` vector length `n`
- `incx, incy` as before
- `alpha, beta, real*8` scale factors
- `lda, integer*8` the leading dimension of `a,` usually `m`
- `trans, character*1` whether to do normal (‘N’), conjugate transposed (‘C’), or transposed (‘T’) mode

LDA

Matrix of mxn: element (i,j) is at address (j-1)*m+i-1

lda parameter necessary if stored size different from logical size




```
real A(20,12)

call dgemv(… A, 7,20,4..)

subroutine dgemv(…A,m,lda,n…)
real A(lda,n)
integer row,col

do row=1,n
  do col=1,m
```

Matrix-Matrix Product

$$\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$$

```
call dgemm(transa, transb, m, n, k, alpha, a, lda,
  b, ldb, beta, c, ldc)
```

- `m, n, a, alpha, beta, lda` as before
- `k, integer*8` outermost matrix dimension
- `b, real*8 n-by-k` 2-d array
- `c, real*8 m-by-k` 2-d array
- `ldb, integer*8` leading dimension of `b` (usually `n`)
- `ldc, integer*8` leading dimension of `c` (usually `m`)
- `transa, transb` transpose mode for `a` and `b`

LAPACK

- Can solve linear systems
  - LU factorization
  - full, symmetric, or banded storage
- Also solves least-squares and eigenproblems
  - QR factorization
  - LQ factorization
  - SVD
  - symmetric/non-symmetric eigenvalue problems
- Naming convention identical to the BLAS
- Calls the BLAS where possible

Factor a Linear System

A = PLU

`call dgetrf(m, n, a, lda, ipiv, info)`

- Factorizes *A,* in-place, into *L and U* (doesn't store the unit diagonal of *L*)
- `m,n, integer*8` matrix dimensions
- `a, real*8 m-by-n 2-d array`
- `lda, integer*8` leading dimension of `a` (usually `m`)
- `ipiv, integer*8` vector, length `n`, stores pivoting information
- `info, integer*8`
  - `0` for succes
  - `-i` for error in the `i`th parameter
  - `i for` $U_{ii}=0$ (application will cause division by zero)

Solve a Factored System

$$\mathbf{Ax = PLUx = b}$$

```
call dgetrs(trans, n, nrhs, a, lda, ipiv, b, ldb,
   info)
```

- Solves a pre-factored (as from dgetrf), `n-by-n` linear system and overwrites `b` with the result
- `n,a,lda,ipiv,trans` as before
- `b, real*8 n-by-nrhs` 2-d array
- `nrhs, integer*8` number of right hand sides to solve (usually 1)
- `ldb, integer*8` leading dimension of `b` (usually `n`)
- `info, integer*8`
  - `0` for succes
  - `-i` for error in the i^{th} parameter

Factor and Solve a System

$\mathbf{Ax = b}$

`call dgesv(n, nrhs, a, lda, ipiv, b, ldb, info)`

- Solves a `n-by-n` linear system and overwrites `a` and `b` with the factorization and the result
- All parameters as for `dgetrs`, except
- `info, integer*8`
  - `0` for succes
  - `-i` for error in the `i`th parameter
  - `i for` $U_{ii}=0$ (application will cause division by zero)

A C Example

```c
#include <stdlib.h>
#include <stdio.h>
#define A(i,j) a[(i) + (j)*n]
void dgetrf_();void dgetrs_();
void dgesv_();
int main()
{
  int i,j,n=3,one=1,info;
  double *a=malloc(n*n*sizeof(double));
  double *b=malloc(n*sizeof(double));
  int *ipiv=malloc(n*sizeof(int));
  char trans[1]={'N'};
  A(0,0)=6.;  A(0,1)=-2.;  A(0,2)=2.;
  A(1,0)=12.; A(1,1)=-8.;  A(1,2)=6.;
  A(2,0)=3.;  A(2,1)=-13.; A(2,2)=3.;
  b[0]=16.;   b[1]=26.;    b[2]=-19.;
  printf("\n\n");
  /* dgetrf_(&n,&n,a,&n,ipiv,&info);
     dgetrs_(trans,&n,&one,a,
             &n,ipiv,b,&n,&info); */
  dgesv_(&n,&one,a,&n,ipiv,b,&n,&info);
  for(i=0;i<n;++i)
    {
     printf("| ");
     for(j=0;j<n;++j)
       {
         printf("%8.5g ",A(i,j));
       }
     printf("|      | %8.5g
   |\n",b[i]);
    }
  return(0);
}
```

In FORTRAN

```
program bar
      integer*8 i,j,n,one
      parameter (n=3)
      parameter (one=1)
      integer*8 ipiv(n),info
      real*8 a(n,n)
      real*8 b(n)
      a(1,1)=6.
      a(1,2)=-2.
      a(1,3)=2.
      a(2,1)=12.
      a(2,2)=-8.
      a(2,3)=6.
      a(3,1)=3.
      a(3,2)=-13.
      a(3,3)=3.
      b(1)=16.
      b(2)=26.
      b(3)=-19.
      call dgesv(n,one,a,n,
                 ipiv,b,n,info)
      do i=1,n
      do j=1,n
      write(*,'(F10.5$)') a(i,j)
      enddo
      write(*,'(10X,F8.5)') b(i)
      enddo
      stop
      end
```

Compiling and Linking

- On Lonestar
- Use the MKL
- Set up the environment

```
lslogin1$ module load mkl
```

- Compile and link
  - C

```
icc -o foo foo.c -L$TACC_MKL_LIB -lmkl -
  lmkl_lapack -lpthread -Wl,-rpath,
  $TACC_MKL_LIB
```

  - FORTRAN

```
ifort -o bar bar.f -I$TACC_MKL_INC -
  L$TACC_MKL_LIB -lmkl -lmkl_lapack -
  lpthread -Wl,-rpath,$TACC_MKL_LIB
```

Fourier Transform

- Recall

$$F(u(x)) = \hat{u}(\xi) = \int_{-\infty}^{\infty} u(x) e^{-2\pi i x \xi} dx \quad \text{(forward)}$$

$$F^{-1}(\hat{u}(\xi)) = u(x) = \int_{-\infty}^{\infty} \hat{u}(\xi) e^{2\pi i x \xi} d\xi \quad \text{(backward or inverse)}$$

- Useful properties
  - linear, *F(af+bg)= aF(f) + bF(g)*
  - converts differentiation, order *n,* to multiplication by $(2\pi i\xi)^n$
  - and integration into division by the same factor
  - turns convolution, *f*g,* into multiplication, *F(f)F(g)*

FFTW

- Computes Discrete Fourier Transforms
  - forward

$$Y_k = \sum_{j=0}^{n-1} X_j e^{-2\pi jk\sqrt{-1}/n}$$

  - inverse

$$Z_k = \sum_{j=0}^{n-1} Y_j e^{2\pi jk\sqrt{-1}/n}$$

  - Forward followed by inverse leaves $Z=n*X$
  - Positive frequencies stored in the first half of Z and negative in the second half
- Does 2- and 3-D DFTs, too
- Can do them in parallel (in an older version)

Interface

- FFTW3
  - FFTW2 still around (and available on Lonestar)
  - 2 → 3 changed the API
- FORTRAN
  - uses the `complex*16` intrinsic datatype
- C/C++
  - defines `fftw_complex`

    ```
    typedef double fftw_complex[2];
    ```
  - can use C99's `complex`

    ```
    double complex *x;
    ```
  - can probably use C99's `complex` in C++ codes for most compilers or `std::complex` with some care

Interface

- Define your data
- Make a *plan*
- Execute the plan
  - perhaps several times with new data
- Destroy the plan
- Clean up your data

Data

```
double complex *in, *out;
fftw_plan p;
in = (double complex*)
  fftw_malloc(sizeof(double complex)
  * n);
out = (double complex*)
  fftw_malloc(sizeof(double complex)
  * n);
```

- `fftw_malloc()` guarantees proper data alignment for the fastest code

Plans

```
p = fftw_plan_dft_1d(n, in, out,
 FFTW_FORWARD, FFTW_ESTIMATE);
fftw_execute(p);
fftw_destroy_plan(p);
```

- Sets up a 1-D Forward DFT
- Estimates the fastest method (`FFTW_MEASURE` for best method)
- Executes the plan
- Can modify `in` and re-execute without destroying as long as `n` doesn't change

Data Cleanup

```
fftw_free(in);
fftw_free(out);
```

- Analogous to malloc()/free()
- Should always call when you’re done with the data

1-D Example

- Random time series in [0,1]
- Transform
- Remove DC-offset
- Inverse Transform
- Low-pass filter
  - keep the first 20 modes
  - reuse the inverse transform plan

C Code

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include "fftw3.h"
int main(int argc, char* argv[])
{
  int n=atoi(argv[1]);
  int i, cutoff = atoi(argv[2]);
  double *x=malloc(n*sizeof(double));
  for(i=0; i<n; ++i) x[i]=((double) rand())/((double)RAND_MAX+1.0);
  printf("x=[");
  for(i=0; i<n; ++i) printf("%15.7g ",x[i]);
  printf("];\n\n");
  double complex *in, *out;
  fftw_plan p;
  in  = (double complex*) fftw_malloc(sizeof(double complex) * n);
  out = (double complex*) fftw_malloc(sizeof(double complex) * n);
  for(i=0; i<n; ++i) in[i]=(x[i]+0.*I);
  p = fftw_plan_dft_1d(n, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute(p);
  fftw_destroy_plan(p);
```

C Code

```
  printf("y=["); for(i=0; i<n; ++i)
    printf("%15.7g + %15.7gi ",
           1./((double) n)*creal(out[i]), 1./((double) n)*cimag(out[i]));
   printf("];\n\n");
  out[0]=0.+0.*I;
  p = fftw_plan_dft_1d(n, out, in, FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_execute(p);
  printf("z=["); for(i=0; i<n; ++i)
    printf("%15.7g + %15.7gi ",
           1./((double) n)*creal(in[i]), 1./((double) n)*cimag(in[i]));
   printf("];\n\n");
  for(i=1; i<n/2+1; ++i)
    if (i > cutoff)
      out[i]=out[n-i]=0.+0.*I;
  printf("r=["); for(i=0; i<n; ++i)
    printf("%15.7g + %15.7gi ",
           1./((double) n)*creal(out[i]), 1./((double) n)*cimag(out[i]));
   printf("];\n\n");
  fftw_execute(p);
  printf("q=["); for(i=0; i<n; ++i)
    printf("%15.7g + %15.7gi ",
      1./((double) n)*creal(in[i]), 1./((double) n)*cimag(in[i]));
   printf("];\n\n");
  fftw_destroy_plan(p);  fftw_free(in); fftw_free(out);
}
```

Compiling and Linking

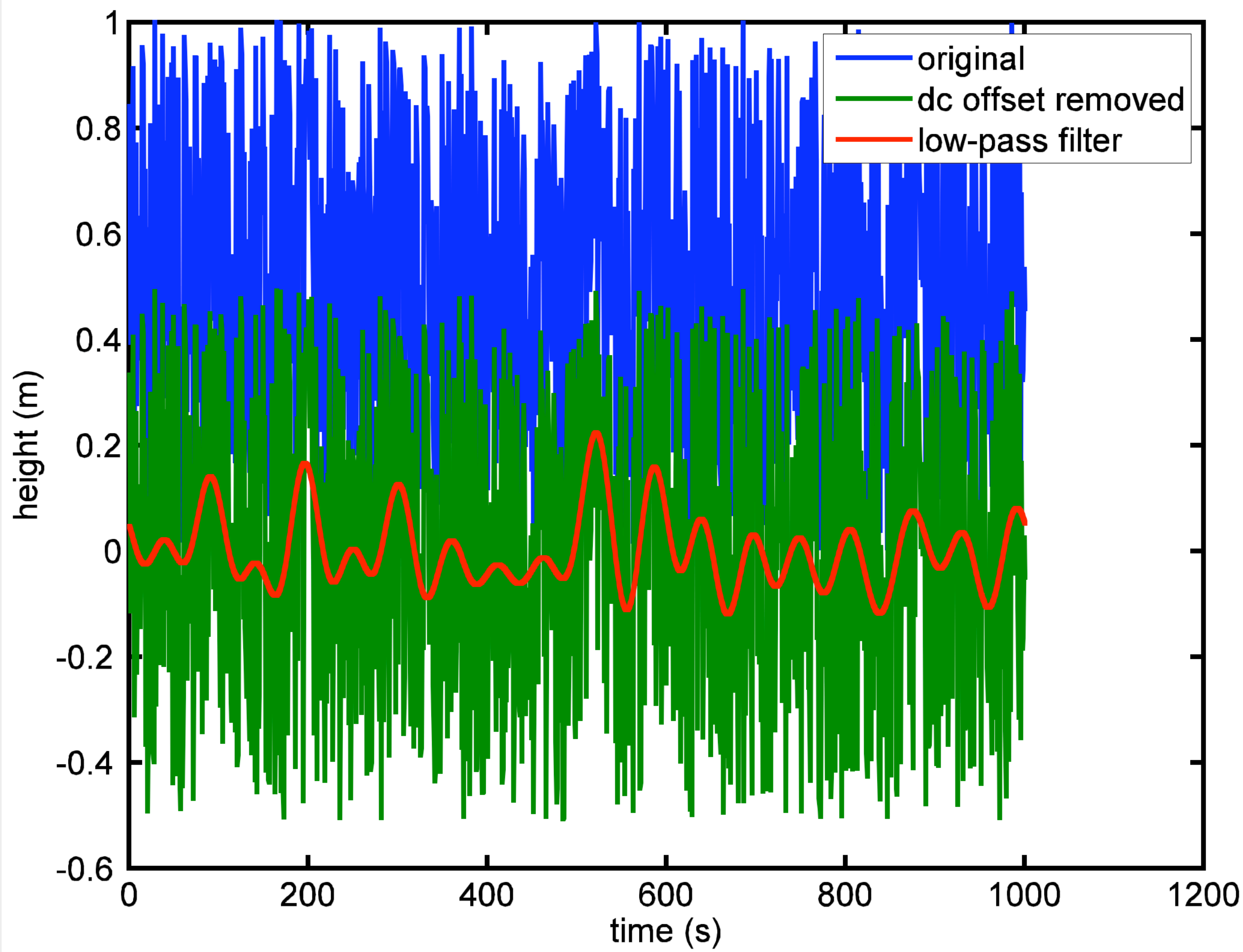
- On Lonestar
- Set up the environment

```
lslogin1$ module load fftw/3.1.1
```

- Compile and link

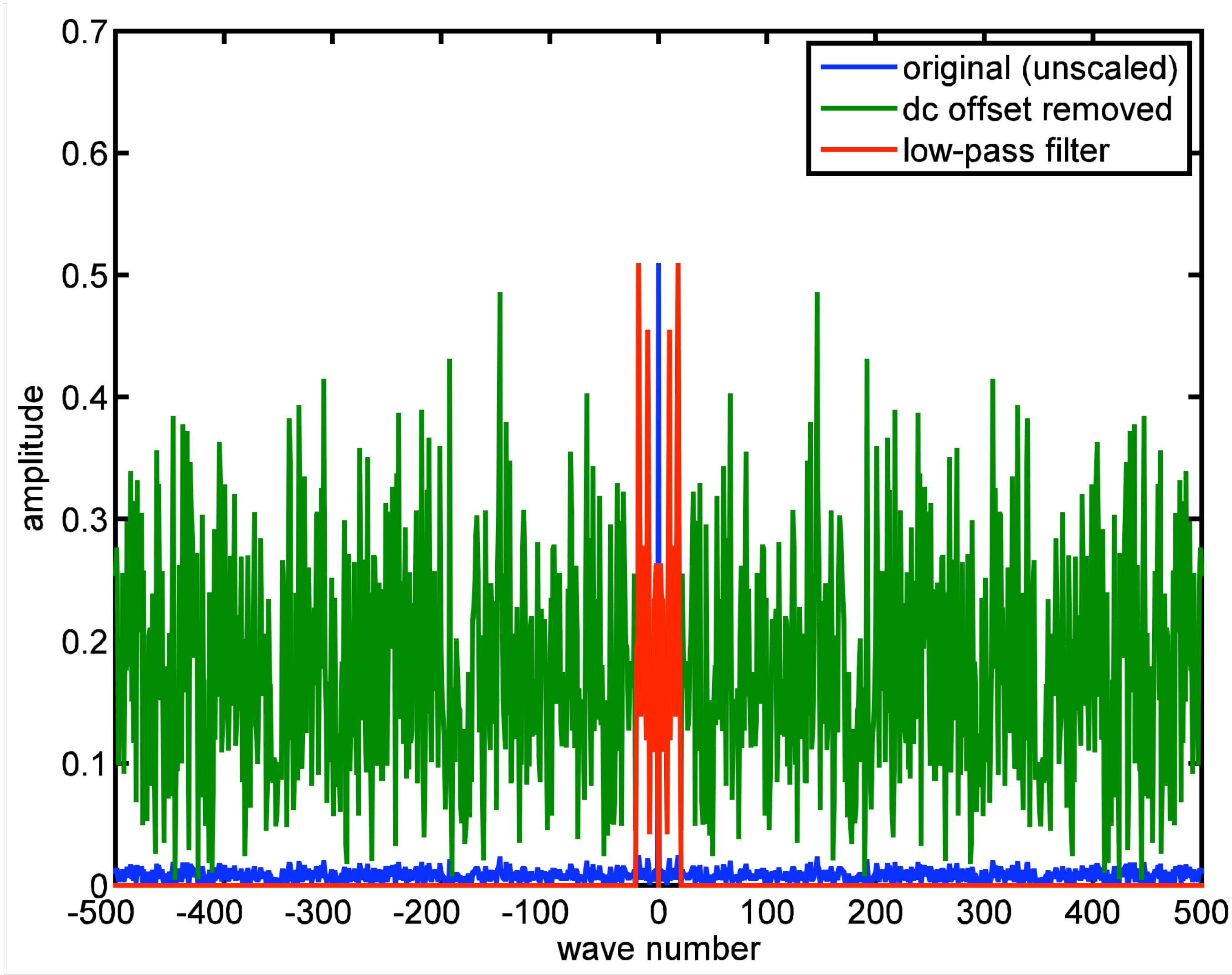
```
icc -o rand_data rand_data.c -
  L$TACC_FFTW3_LIB -lfftw3 -lm -
  I$TACC_FFTW3_INC
```

Results



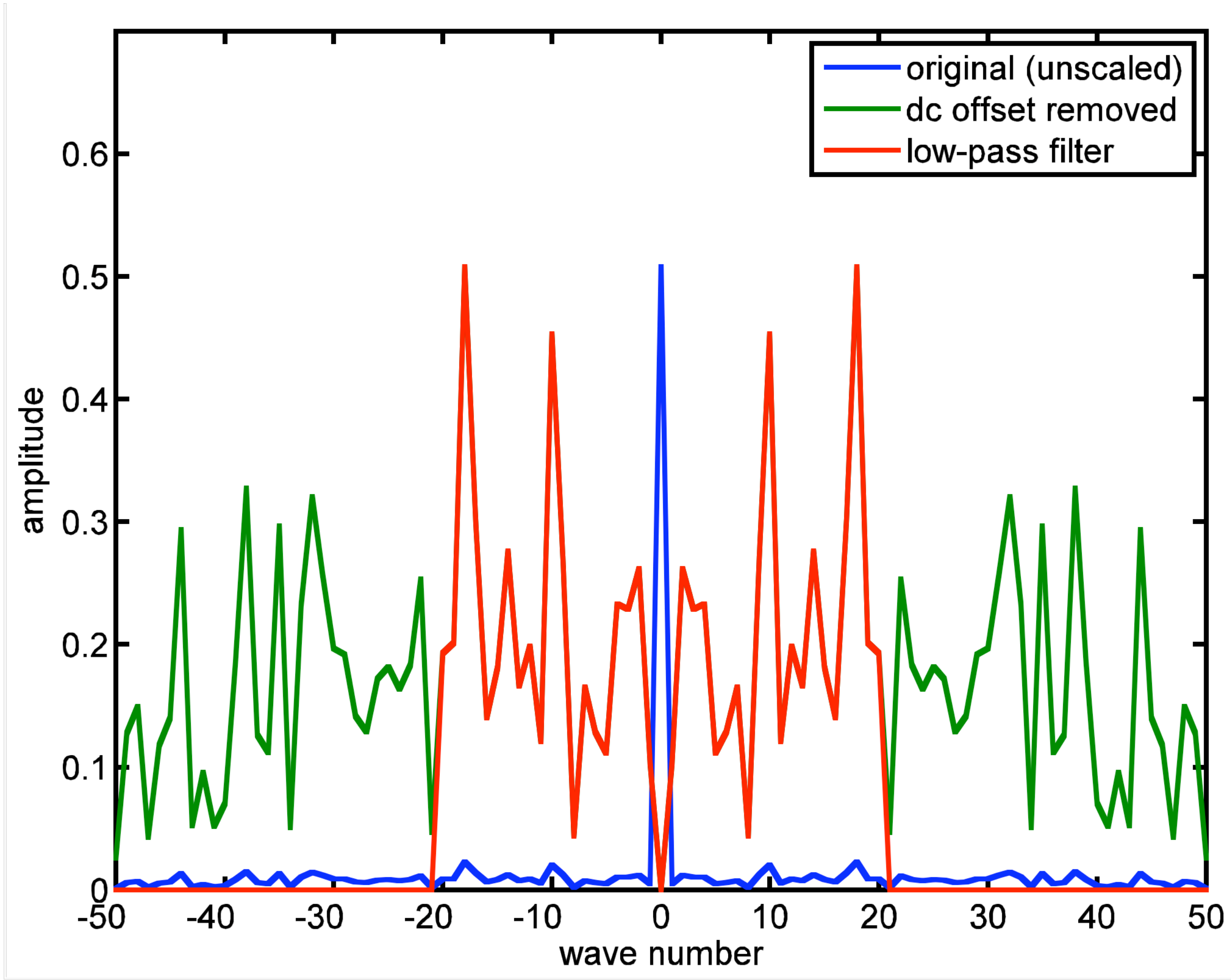
original
dc offset removed
low-pass filter
height (m)
time (s)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0
200
400
600
800
1000
1200

Results



original (unscaled)
dc offset removed
low-pass filter
amplitude
wave number
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-500
-400
-300
-200
-100
0
100
200
300
400
500

Results


original (unscaled)
dc offset removed
low-pass filter
0.6
0.5
0.4
0.3
0.2
0.1
0
amplitude
-50
-40
-30
-20
-10
0
10
20
30
40
50
wave number