Appendix C

Computer Programmes

This appendix contains listings of some of the computer programmes developed during the course of research. Code is presented for programmes which were central to the tasks of calibration and measurement of lift forces. The operation of the programmes and the numerical methods used were discussed in chapter 4.

The first section of source code is for the programme *dyncal*, which was used to perform dynamic calibrations of the strain bridges contained in each cylinder transducer segment.

The next section contains the modules which together make up the source code for *adcal*, used for processing of data collected from the fixed and oscillating cylinder.

The final section contains source for the subroutines *included* in the source for *dyncal* and *adcal*.

The variant of the Pascal language used was Berkeley Pascal, which supports a number of non-standard features, such as ability to deal with command-line arguments, an include facility, separate compilation of programme modules, and extensions to file-handling procedures. The Berkeley Pascal User's Manual, by W.N. Joy, S.L. Graham and C.B. Haley provides a complete description of these features.
C.1 Dynca

program dynca(input, output, UFile);
{
    -- Revision 0         0 January 1990
    Author               Hugh M. Blackburn

    -- DYNCA: perform dynamic calibration of force transducer strain bridge using timeseries collected from cylinder model oscillating over calibrated load beam.

    -- Usage: dynca -f 'calfile' < stdin

    -- Summary: DYNCA:--
    * removes the inertial component of force transducer timeseries using recursively computed digital filter coefficients,
    * corrects input data for time skew and anti-aliasing filter transfer functions,
    * then least-squares fits a calibration constant between force transducer output and load beam timeseries.

    -- Input to DYNCA comes from standard input and a number of other files which are named inside a file whose name in turn is given as a command-line argument.

    The standard input is assumed to consist of three channels of text fixed-point data obtained by analogue-to-digital conversion of accelerometer & strain bridge amplifier outputs. The format of this input is columnar, with the columns containing in order the outputs of the accelerometer and strain bridge contained in the transducer segment, and the output of the load-beam strain bridge.

    The file named on the command line (-f 'calfile') contains the name of the data file to be used for adaptation of the inertial cancellation filter the names of the two anti-aliasing filters used in data correction, and calibration of the load beam [N/V]. The sampling rate of analogue-to-digital conversion is also given.

Example CFile
--------------------------------------------------
The first three lines are a header, and are stripped by DYNCA without examination.

891121.1 ( Inertial cancellation adaption )
32 ( Cancellation IRF length )
/usr/hmb/cal/Filters/816.1 ( Strain Bridge filter TFEST file )
/usr/hmb/cal/Filters/816.0 ( Load Beam filter TFEST file )
200.0 ( Hz sampling rate )
0.75 ( Low-pass Filter roll-point, )
 as proportion of Nyquist. )
0.4173 ( Load beam calibration [N/V] )
--------------------------------------------------

-- Output consists of the calibration constant computed by the programme, (dimensions: Newtons / Volt), written to a file called dynca.out and the timeseries of calibrated force transducer output, load beam output & error written on standard output.

N.B. The calibration constant produced as output should be changed in sign if the load beam sign convention is that upwards force
produces positive voltage change (by application of Newton's Third Law).

}  

const  
MAXSTR = 100;
MAXPWT = 4096;
MAXIRF = 64;
MAXORD = 10;


type  
string = packed array [i..MAXSTR] of char;
complex = 
record  
  Re, Im : real
end;
cvector = array [i..MAXPWT] of complex;
tempstore = file of complex;

polname = (num, denom);
filtercoeff = array [polname] of array [0..MAXORD] of real;

filtvect = array [i..MAXIRF] of real;

var  
CalFileName,  
AdaptFileName,  
Filter2Name,  
Filter3Name : string;
UFile : text;
TempFile : tempstore;
LoadBeamCal : real;
IRF : filtvect;
IRFLen : integer;
SampleRate,  
LowPassProprn : real;
Npts : integer;
AccMean,  
ForMean,  
LodMean : real;
AccBuf,  
ForBuf,  
LodBuf : cvector;
AntiAlias : filtercoeff;
Order : integer;
Offset, Slope : real;

#include '/usr/hmb/incl/FFT.i'
#include '/usr/hmb/incl/realFFT.i'
procedure getargs( {return} var fname : string);
{
    Return the name of calibration data file from command-line.
}

var
    paramerror : boolean;
    word      : string;

begin
    paramerror := true;

    if (argc = 3) then begin
        argv(1, word);
        if ((word[1] = '-') and (word[2] = 'f')) then begin
            paramerror := false;
            argv(2, fname)
        end
        else
            message('dyncal: bad flag: ', word:2)
        end
    else
        message('dyncal: arg count');

    if paramerror then begin
        message('Usage: dyncal -f calfile < stdin');
        halt
    end
end;

#include '/usr/hmb/incl/issolid.i'
#include '/usr/hmb/incl/strngread.i'
#include '/usr/hmb/incl/blankstrip.i'

procedure interogate( {from} var infile      : text;
                      {get} var AdaptiFileName,
                           Filter2Name,
                           Filter3Name  : string;
                      var IRFlen         : integer;
                      var SampleRate,
                      LowPassPropn,
                      LoadBeamCal      : real);
{
    Return information from calibration file named on the command-line.
}

begin
    readln(infile);
    readln(infile);
    readln(infile);

    stringread(infile, AdaptiFileName);
    blankstrip(AdaptiFileName);
    readln(infile, IRFlen);
    if (IRFlen > MAXIRF) then begin
        message('dyncal: IRFlen requested is out of range (', MAXIRF:1, ')');
        halt
    end
end;
readln(infile);
stringread(infile, Filter2Name);
blankstrip(Filter2Name);
stringread(infile, Filter3Name);
blankstrip(Filter3Name);
readln(infile);
readln(infile, SampleRate);
readln(infile, LowPassPropn);
readln(infile);
readln(infile, LoadBeamCal)
end;

procedure refile( {from} var infile : text;
{to} var tempfile : tempstore;
{return} var AccMean,
          ForMean : real);
{
   The mean values of the first two channels of the cancellation
   adaption file need to be removed before the cancellation filter
   can be constructed. Therefore all the values in infile must be
   read in before the filter can be constructed. To save having to
   convert all the values to real twice, they are saved in a
   temporary file.
}
var
   N : integer;
   Z : complex;
begin
   N := 0;
   AccMean := 0.0;
   ForMean := 0.0;
   
   while (not eof(infile)) do begin
      with Z do begin
         readln(infile, Re, Im);
         AccMean := AccMean + Re;
         ForMean := ForMean + Im
      end;
      tempfile := Z;
      put(tempfile);
      N := N + 1
   end;

   AccMean := AccMean / N;
   ForMean := ForMean / N
end;

#include './usr/hmb/incl/poweri.i'
#include './usr/hmb/incl/adaptgain.i'
#include './usr/hmb/incl/adaptoffts.i'
procedure identify{ {from} var tempfile : tempstore;
{make} var IRF : filtvec;
{using} IRFLen : integer;
   AccMean,
   ForMean : real;

{ Recursively construct inertial force cancellation digital filter
  using timeseries recorded without loadbeam.
}

var time, i : integer;
Weight : real;
AdGain,
Fcoeff, 
Bcoeff : filtvec;
ForPredEn,
BakPredEn,
PredErrRat : real;
X, Y, Error : real;
Xvect : filtvec;
Z : complex;

begin
  Weight := (IRFLen + sqrt(32*IRFLen) - 1) /
            (IRFLen + sqrt(32*IRFLen));

  time := 0;
  for i := 1 to IRFLen do IRF[i] := 0.0;
  while (not eof(tempfile)) do begin 
    Z := tempfile$;
    get(tempfile);
    X := Z.Re - AccMean;
    Y := Z.Im - ForMean;
    adaptgain(IRFLen, time, X, Weight, Xvect,
              Fcoeff, Bcoeff, AdGain,
              ForPredEn, BakPredEn, PredErrRat);
    adaptcoeffs(IRF, IRFLen, Y,
                PredErrRat, AdGain, Xvect, Error);
    time := time + 1
  end;

procedure readdata{ {from} var infile : text;
{read} var AccBuf,
ForBuf,
LodBuf : cvector;
{return} var AccMean,
ForMean,
LodMean : real;
var Npts : integer;

{ Read in the data from which the force transducer calibration will
  be computed. The data are read in from a text file and are packed
  into complex buffers so that odd & even indexed real points are
  packed together. }
packed into real & imaginary parts of the complex buffer respectively. See procedure realFFT for more detail.

Mean values and number of points read in are returned.
}

var
A, F, L : real;

begin
Npts := 0;
AccMean := 0.0;
ForMean := 0.0;
LodMean := 0.0;

while ((not eof(infile)) and (Npts < 2*MAXPNT)) do begin
  readln(infile, A, F, L);
  Npts := Npts + 1;
  AccMean := AccMean + A;
  ForMean := ForMean + F;
  LodMean := LodMean + L;

  if odd(Npts) then begin
    AccBuf[(Npts + 1) div 2].Re := A;
    ForBuf[(Npts + 1) div 2].Re := F;
    LodBuf[(Npts + 1) div 2].Re := L
  end
  else begin
    AccBuf[Npts div 2].Im := A;
    ForBuf[Npts div 2].Im := F;
    LodBuf[Npts div 2].Im := L
  end
end;

if (not eof(infile)) then begin
  message('dyncal: storage buffers filled before end of input (',
    Npts:i, ',')
  );
  halt;
end;

AccMean := AccMean / Npts;
ForMean := ForMean / Npts;
LodMean := LodMean / Npts
end;

procedure demean( {using}
  {process} var Buf : cvector);
{ Given a mean value for time series, subtract it from all the terms.
}

var
i : integer;

begin
for i := 1 to Npts div 2 do
  with Buf[i] do begin
    Re := Re - Mean;
    Im := Im - Mean
  end
end;
end;

procedure cancelforce( {using} Npts : integer;
var IRF  : filtvect;
IRFLen : integer;
{update} var AccBuf,
ForBuf : cvector);
{
  Conolve accelerometer data in AccBuf with IRF to produce an
  estimate of inertial component of force transducer output.
  Subtract the estimate from force transducer output, contained in
  ForBuf.
}
var
  i, j    : integer;
Avect   : filtvect;
Ppredict : real;
begin
  for i := 1 to IRFLen do Avect[i] := 0.0;

  for j := 1 to Npts do begin
    Ppredict := 0.0;
    for i := IRFLen downto 2 do
      Avect[i] := Avect[i - 1];
    if odd(j) then
    else
      Avect[1] := AccBuf[j div 2].Im;
    for i := 1 to IRFLen do
      Ppredict := Ppredict + IRF[i]*Avect[i];
    if odd(j) then
      with ForBuf[(j + 1) div 2] do
        Re := Re - Ppredict
    else
      with ForBuf[j div 2] do
        Im := Im - Ppredict
  end;

procedure getfilter( {from} var infile : text;
{return} var AntiAlias : filtercoeff;
         var Order : integer);
{
  Return rational polynomial coefficients of antialiasing filter
  transfer function from file named in the calibration file.

  The coefficients of the transfer function are stored line-by-line as
  numerator and denominator coefficients, starting from zeroth order.
  Example (6-th order estimate of low-pass filter):

  NUMERATOR    DENOMINATOR
  9.99984693269768e-01  1.000000000000000e+00
-2.52966838344665e-02  3.14990933798538e-02
  3.93222787101300e-04  5.70367960590046e-04
-4.28797744489220e-06  6.82564980137870e-06

  To get these coefficients, use:

  for i := 1 to Order do begin
    num := getfilter(infile);
    print(num);
  end;
}


begin
readln(infile);
Order := -1;
while ((not eof(infile)) and (Order < MAXORD)) do begin
  Order := Order + 1;
  readln(infile, AntiAlias[num][Order], AntiAlias[denom][Order]);
end;

if (not eof(infile)) then begin
  message('dyndcal: ran out of storage for antialiasing coeffs');
  halt;
end;

procedure response( {using} frequency : real;
  var transf : filtercoeff;
  ordr : integer;
{return} var answer : complex);
{
  Compute the numerator and denominator of the filter
  frequency response using Hoerner scheme, then divide
  to get a complex magnitude.
}

var
  N, D   : complex;
  expnt  : integer;
  RePart, Den : real;

begin
  N.Re := transf[num][ordr];
  N.Im := 0.0;
  D.Re := transf[denom][ordr];
  D.Im := 0.0;

  for expnt := (ordr - 1) downto 0 do begin
    { Nested multiplication of polynomial coefficients with j*freq. }
    RePart := N.Re;
    N.Re := -N.Im * frequency;
    N.Im := RePart * frequency;

    RePart := D.Re;
    D.Re := -D.Im * frequency;
    D.Im := RePart * frequency;

    N.Re := N.Re + transf[num][expnt];
    D.Re := D.Re + transf[denom][expnt]
  end;
{ Complex division of numerator by denominator. }
  Den := sqr(D.Re) + sqr(D.Im);
  answer.Re := (N.Re*D.Re + N.Im*D.Im) / Den;
  answer.Im := (N.Im*D.Re - D.Im*N.Re) / Den;
end;

procedure unfilter( {update} var Buf : cvector;
{using} npts : integer;
var Filter : filtercoeff;
order : integer;
samprate : real;

{ The filter coefficients contain estimates of the poles & zeros of the anti-aliasing filters. Multiply data DFT by the inverse of the transfer functions so generated, to estimate the data in its pre-filtered state. }

var i : integer;
freq, RePart, Den : real;
H : complex;

begin
  with Buff[i] do begin
    Re := Re * Filter[denom][0] / Filter[num][0];
    Im := 0.0 { can't cope with Nyquist frequency value }
  end;

  for i := 2 to (npts div 2) do begin
    freq := (i - 1) * samprate / npts;
    response(freq, Filter, order, H);

    { Complex division of Buff[i] by H :- }
    Den := sqr(H.Re) + sqr(H.Im);
    with Buff[i] do begin
      RePart := Re;
      Re := (Im*H.Im + RePart*H.Re) / Den;
      Im := (Im*H.Re - RePart*H.Im) / Den
    end
  end;

procedure bandpass( {filter} var Zbuf : cvector;
{using} fcut : real;
npts : integer);

{ Carry out a fairly sharp filter of data in Zbuf. Lowpass end of filter rolls at fcutf*dataNyquistfrequency. Use a linear rolloff, nominally 0.05*dataNyquist long. }

var rollstart, rollwidth : integer;
i, halflength : integer;
filtermag : real;

begin
  halflength := npts div 2;
  rollstart := round(fcut*halflength);
  rollwidth := round(0.05*halflength);

  for i := rollstart to (rollstart + rollwidth) do begin
    filtermag := 1.0 - (i - rollstart) / rollwidth;
    with Zbuf[i] do begin
      Re := filtermag * Re;
      Im := filtermag * Im
    end
  end;
end
end;

for i := (rollstart + rollwidth + 1) to halflength do
    with Zbuf[i] do begin
        Re := 0.0;
        Im := 0.0
    end;

    Zbuf[i].Re := 0.0;
    Zbuf[i].Im := 0.0
end;

procedure delay( {update} var Zbuf : cvector;
    {using} npts : integer); {Correct the data for time delay in analogue-to-digital conversion caused by multiplexing of A2D converter between channels.}

Computation is carried out in the frequency domain where a time delay corresponds to a linear phase error.

Load beam data are brought back to zero delay relative to strain bridge data.
}

const
    PI = 3.14159265359;
var
    i : integer;
    Slope,
    alpha, cosA, sinA,
    temp : real;
begin
    Slope := PI / (2 * npts); { PI/4 at Nyquist <= 3 chans }
    for i := 2 to (npts div 2) do begin
        alpha := (i - 1) * Slope;
        cosA := cos(alpha);
        sinA := sin(alpha);

        { multiply DFT by exp(-j*alpha) <=> time domain delay }
        with Zbuf[i] do begin
            temp := Re;
            Re := cosA*Re + sinA*Im;
            Im := cosA*Im - sinA*temp
        end;
    end;
end;

procedure fit( {update} var ForBuf : cvector;
    {using} var Calibration : real;
    var LodBuf : cvector;
    Npts : integer;
    {return} var Offset, Slope : real); {Compute a constant which when multiplied by force transducer timeseries, reproduces the loadbeam timeseries with least squared error. The fit is carried out over the most central 80% of the}
timeseries, to allow for end effects. The fitted relationship is
linear between the applied and residual force, done on a least-squares
basis.

Modify load beam calibration to reflect the fitted coefficient.
(i.e. the calibration constant of the load beam becomes the calibration
constant of the force transducer.)

```
}
var
  i, NFit : integer;
  X, Y,
  SumX, SumX2,
  SumY, SumXY : real;
begin
  SumX := 0.0;  SumX2 := 0.0;
  SumY := 0.0;  SumXY := 0.0;

  NFit := Npts - 2*(Npts div 10);
  for i := (Npts div 10) to (Npts - (Npts div 10)) do begin
    if odd(i) then begin
      X := ForBuf[(i + 1) div 2].Re;
      Y := LodBuf[(i + 1) div 2].Re
    end
    else begin
      X := ForBuf[i div 2].Im;
      Y := LodBuf[i div 2].Im
    end;
    SumX := SumX + X;
    SumX2 := SumX2 + sqr(X);
    SumY := SumY + Y;
    SumXY := SumXY + X*Y
  end;

  Slope := (SumXY - SumX*SumY/NFit) / (SumX2 - sqr(SumX)/NFit);
  Offset := (SumY - Slope*SumX) / NFit;

  Calibration := Slope * Calibration
end;
```

```
procedure printup(  {on}   var outfile : text;
    {print} var ForBuf, LodBuf : cvector;
    {using} Wpts : integer;
            Offset, Slope : real);
{
  Print on standard output the corrected transducer force timeseries,
  the loadbeam timeseries, and the difference between the two (error).
}
var
  i : integer;
  X, Y, Error : real;
begin
  for i := 1 to Wpts do begin
    if odd(i) then begin
      X := ForBuf[(i + 1) div 2].Re;
      Y := LodBuf[(i + 1) div 2].Re
    end
```
else begin
  X := ForBuf[i div 2].Im;
  Y := LodBuf[i div 2].Im
end;

X := Offset + Slope*X;
Error := Y - X;
writeln(outfile, X:14, Y:14, Error:14)
end;

begin
  getargs(CalFileName);

  reset(UFile, CalFileName);
  interrogate(UFile, AdaptFileName, Filter2Name, Filter3Name,
               IRFLen, SampleRate, LowPassPropn, LoadBeamCal);

  reset(UFile, AdaptFileName);
  rewrite(TempFile);
  refile(UFile, TempFile, AccMean, ForMean);
  reset(TempFile);
  identify(TempFile, IRF, IRFLen, AccMean, ForMean);

  readdata(input, AccBuf, ForBuf, LodBuf,
            AccMean, ForMean, LodMean, Npts);

  demean(Npts, AccMean, AccBuf);
  demean(Npts, ForMean, ForBuf);
  demean(Npts, LodMean, LodBuf);

  cancelforce(Npts, IRF, IRFLen, AccBuf, ForBuf);
  realFFT(ForBuf, Npts div 2, true);
  reset(UFile, Filter2Name);
  getfilter(UFile, AntiAlias, Order);
  unfilter(ForBuf, Npts, AntiAlias, Order, SampleRate);
  bandpass(ForBuf, LowPassPropn, Npts);
  realFFT(ForBuf, Npts div 2, false);

  realFFT(LodBuf, Npts div 2, true);
  reset(UFile, Filter3Name);
  getfilter(UFile, AntiAlias, Order);
  unfilter(LodBuf, Npts, AntiAlias, Order, SampleRate);
  delay(LodBuf, Npts);
  bandpass(LodBuf, LowPassPropn, Npts);
  realFFT(LodBuf, Npts div 2, false);

  fit(ForBuf, LoadBeamCal, LodBuf, Npts, Offset, Slope);
  rewrite(UFile, 'dynca1.out');
  writeln(UFile, LoadBeamCal:15, ' N/V');

  printup(output, ForBuf, LodBuf, Npts, Offset, Slope)
end.
C.2 Adcal

C.2.1 Makefile

```
SOURCES = getargs.p io.p fourier.p timedom.p \
         deconv.p integ.p adapt.p main.p
OBJECTS = getargs.o io.o fourier.o timedom.o \
         deconv.o integ.o adapt.o main.o
HEADERS = globals.h
adcal:  $(OBJECTS)
    pc $(OBJECTS) -o $@

$(OBJECTS): $(HEADERS)
getargs.o:  getargs.p
    pc -c -m getargs.p
io.o:  io.p
    pc -c -m io.p
fourier.o:  fourier.p
    pc -c -m fourier.p
timedom.o:  timedom.p
    pc -c -m timedom.p
deconv.o:  deconv.p
    pc -c -m deconv.p
integ.o:  integ.p
    pc -c -m integ.p
adapt.o:  adapt.p
    pc -c -m adapt.p
main.o:  main.p
    pc -c -m main.p
```

C.2.2 main.p

```
program adcal(input, output, AdFile, CFile, RlFile);
{
    -- Revision 3  19 April 1990  -- Real output file option
    Author        Hugh M. Blackburn

    -- ADCAL: process acceleration & force timeseries collected from
    -- oscillating cylinder model in wind tunnel.

    -- Usage: adcal -a 'adfile' -f 'calfile' [-o 'output'] [-u] < stdin

    -- Summary: ADCAL :-
    * removes the inertial component of force transducer timeseries,
    * corrects the remaining signals for different anti-aliasing filter
      transfer functions,
    * corrects accelerometer signals for their transfer functions
      (referenced to a Sundstrand accelerometer),
    * integrates accelerometer timeseries to produce timeseries of
      cylinder crossflow velocities.

    -- Input to ADCAL comes from a number of files and the command line.

    The standard input is assumed to consist of 12 channels of ASCII
```
fixed-point data obtained by analogue-to-digital conversion of the output from accelerometer and strain bridge amplifiers. The format of this input is columnar, with the 12 columns consisting of 6 sets of 2 columns, each set consisting of accelerometer and strain bridge output respectively. (The cylinder has 6 measurement stations, each with a force transducer and accelerometer.) Number of rows of input must be a power of 2, for FFT, and length is checked to ensure that it equals the standard number \( (2 \times \text{MAXPT}) \), i.e. 8192.

A file named on the command line (AdFile in programme header) contains 12 channels of data in the same format, which is assumed to have been obtained from the cylinder in the absence of flow. These data are used to form an estimate of the force transducer Impulse Response Function (IRF) to acceleration for each transducer. The method of Fast Least Squares is utilised to carry out this estimation adaptively [1].

Another file named on the command line (CFile in programme header) contains standard information used by the programme: analogue-to-digital conversion frequency, the lowpass cutoff frequency to be used and the length of impulse response to be used for force cancellation, unit conversion factors, and names of filter & accelerometer calibration files which ADCAL reads during execution.

Example CFile:  

```
Processing of turbulent flow data from November 1989.
Low Reynolds number.

200.0 (Sampling frequency [Hz])
0.825 (Lopass cutoff (as a proportion of Nyquist))
32 (Impulse response length)

9.7447 -0.4347 (accelerometer re SStrand, force cal)
-9.7447 -0.4903 (multiply by these factors to get: )
-9.7447  0.4872 ( acceleration [m/s^-2], force [N] )
  9.7447  0.4707
-9.7447  0.4335
-9.7447  0.5001

/usr/hmb/cal/Accel/ssvib.1x (accelerometer transfer )
/usr/hmb/cal/Accel/ssvib.3x (functions: referenced )
/usr/hmb/cal/Accel/ssvib.5x (to Sundstrand #1005 )
/usr/hmb/cal/Accel/ssvib.7x
/usr/hmb/cal/Accel/ssvib.9x
/usr/hmb/cal/Accel/ssvib.11x
/usr/hmb/cal/Filters/hydroa.1 (filter transfer functions)
/usr/hmb/cal/Filters/hydroa.2
/usr/hmb/cal/Filters/hydroa.3
/usr/hmb/cal/Filters/hydroa.4
/usr/hmb/cal/Filters/616.0
/usr/hmb/cal/Filters/616.1
/usr/hmb/cal/Filters/616.2
/usr/hmb/cal/Filters/616.3
/usr/hmb/cal/Filters/hydrob.1
/usr/hmb/cal/Filters/hydrob.2
/usr/hmb/cal/Filters/hydrob.3
/usr/hmb/cal/Filters/hydrob.4
```

Conversion of voltages to units of velocity, acceleration & force can be denied by using the command line argument `-u`.
Output from ADCAL consists of timeseries of acceleration, velocity, and force. These may be written as ASCII data on standard output, or optionally written as a file of real data, named on the command line (-o 'file' option).

Standard output from ADCAL is written as floating-point ASCII data, 18 columns wide. There are 3 columns for each transducer station, which contain: acceleration timeseries [m/sec^2], cylinder crossflow velocity timeseries [m/sec], crossflow aerodynamic force timeseries [N] respectively. If unit conversion is denied in command line, output is in Volts.

If the output is to be written as a named file of real data, the created file has a slightly different format. The first two entries in the file specify the number of data in each timeseries and the number of timeseries (18 here) respectively. The next fourteen entries are zero (this area is intended for use as a header for standard double precision and floating point files). The rest of the entries in the file are the timeseries, as corrected by ADCAL, written in order of acceleration, velocity and force for one transducer after another.

The operations carried out by ADCAL are as follows:

Read command line for file names and possible denial of unit conversion.

Open CFile and read contents. Extract fitted transfer function estimates for antialiasing filters and accelerometer calibration.

Open AdFile and use it to construct the IRF for each ring. Task is one of system identification. Note that the force transducers are not shielded from the fluid during the recording of this data, so that an added mass equal to the mass of fluid displaced by the transducer volume is coupled to the transducer.

Read all data on standard input and write real data to temporary files. One of the most time-consuming tasks is conversion of fixed point ASCII data to real. Doing it once only for each datum is quickest. Method forced by input structure. For each ring the data stored in the file is alternately accelerometer & strain bridge output.

Loop over transducers and perform the following:

Retrieve real data from temporary storage into complex buffers (one for accelerometer and the other for strain bridge output) for further processing. What were originally the odd- & even-indexed real data in each column of standard input are stored as successive real & imaginary parts of the complex buffers. This storage format is used so that a real-data-only FFT can be performed [3]. Record mean values of timeseries for later removal.

Convolve the estimated IRF through the accelerometer timeseries to produce estimate of inertial component of force transducer output. Subtract from force transducer timeseries. Remove mean values.

Perform FFT on complex buffers. Since that DFT of real
data is conjugate symmetric, only the positive frequency values need to be carried [2], [3].

Carry out delay correction of data. The analogue-to-digital conversion used has only one converter, which is multiplexed to the successive data channels for operation. This has the effect of making each channel appear ADVANCED in time from the previous channel. The effect can be corrected using the time delay -- phase shifting duality of the Fourier Transform [2], by multiplying in the frequency domain by a complex sinusoid. This is equivalent to convolving with an impulse, suitably delayed, in the time domain (convolution with sinc function for bandlimited sampled data).

Correct data for antialiasing filter transfer functions. The 12 data channels use lowpass filters with differing characteristics (couldn't afford identical sets). This has the effect of introducing magnitude, and more importantly, phase distortions (different time delays) between the channels. Using fitted rational polynomial approximations to the filter transfer functions [4], divide the data DFTs by the filter transfer functions at each frequency point. This has the effect of deconvolving the timeseries from the filter impulse responses, thereby approximating the data in its pre-filtered state [2].

Using experimentally derived transfer function estimates, again in the form of fitted rational polynomial approximations, perform inverse filtering on accelerometer signal DFT. This operation tries to account for the fact that the accelerometer transfer functions are not perfectly flat over the frequency range of interest (A Sundstrand accelerometer was used as reference).

Perform bandpass operation on acceleration & force data buffers, according to cutoff frequency specified in CFile.

Since the acceleration DFT will be manipulated to produce velocity, copy the acceleration DFT to the velocity DFT.

Perform frequency-domain version of integration of accelerometer DFT (multiply by 1/j*Omega). This operation uses the fact that the DFT is the Fourier Series of the periodic extension of the input data. The Fourier Series can be integrated term-by-term [5], however zero-frequency component cannot be handled in the frequency domain.

Convert complex buffers back to the time domain using IFFT.

Carry out last step in integration of accelerometer data.

Remove any mean value in velocity timeseries.

If unit conversion flag was left set, use conversion factors to convert voltage signals to acceleration, velocity & force units.

Write processed timeseries data back into real temporary storage.

Print out 18 channels of timeseries data on standard output,
or write it as 64-bit BCD data in file named on command line.

-- References:
Analysis. Marcel Dekker Inc.

Applications. Prentice-Hall.


}
#include 'globals.h'

var
RingNum : integer;
Fcut : real;
Slope : real;
Npts : integer;
SampleRate : real;
AccBuf,
VelBuf,
ForBuf : cvector;
Ring : ringinfo;
AdFile : text;
AdFileName : string;
IRFLen : integer;
CFile : text;
CFileName : string;
UnitConversion : boolean;
FloatOut : boolean;
OFilename : string;
RName : tempfile;

procedure getargs(var adaptfilename,
calfilename : string;
var floatout : boolean;
var outfilename : string;
var convert : boolean); external;

procedure getcalinfo(var df : text;
var ring : ringinfo;
var smprat, fcut : real;
var implslen : integer); external;

procedure identify(var Ring : ringinfo;
IRFLen : integer); external;
procedure refile( var infile : text;  
   var ring : ringinfo);   external;

procedure readdata( var storehandle : tempfile;  
   var Abuf, Fbuf : cvector;  
   var npts : integer);   external;

procedure cancelforce( var ForBuf, AccBuf : cvector;  
   npts, filtlen : integer;  
   var IRF, AcMean, FoMean : real);   external;

procedure realFFT( var Zbuf : cvector;  
   N : integer;  
   forwards : boolean);   external;

procedure delay( var Zbuf : cvector;  
   npts, chan : integer);   external;

procedure deconvolve( var Zbuf : cvector;  
   npts : integer;  
   samprate : real;  
   var filter : filtercoeff;  
   forw : integer);   external;

procedure bandpass( var Zbuf : cvector;  
   fcut : real;  
   npts : integer);   external;

procedure freqinteg( var Zbuf : cvector;  
   npts : integer;  
   samprate : real;  
   var slope : real);   external;

procedure finishinteg( var Zbuf : cvector;  
   npts : integer;  
   slope : real);   external;

procedure removemean( var Zbuf : cvector;  
   npts : integer);   external;

procedure calibrate( var Zbuf : cvector;  
   npts : integer;  
   calfact : real);   external;

procedure writetemp( var storehandle : tempfile;  
   var Abuf, Vbuf, Fbuf : cvector;  
   npts : integer);   external;

procedure writeval( var floating : tempfile;  
   var ring : ringinfo;  
   npts : integer);   external;

procedure printup( var outfile : text;  
   var ring : ringinfo;  
   npts : integer);   external;

begin  
   UnitConversion := true;
FloatOut := false;
getargs(AdFileName, CFileName, FloatOut, CFileName, UnitConversion);

reset(CFile, CFileName);
getcalinfo(CFile, Ring, SampleRate, Fcut, IRFLen);

reset(AdFile, AdFileName);
refile(AdFile, Ring);
identify(Ring, IRFLen);

refile(input, Ring);

for RingNum := 1 to NRINGS do begin
  readdata(Ring[RingNum].Store, AccBuf, ForBuf, Npts);
  cancelforce(ForBuf, AccBuf, Npts,
              IRFLen, Ring[RingNum].IRF,
              Ring[RingNum].AccMean, Ring[RingNum].ForMean);
  realFFT(AccBuf, Npts div 2, true);    // --> frequency domain
  realFFT(ForBuf, Npts div 2, true);
  delay(AccBuf, Npts, 2 * RingNum - 1);
  delay(ForBuf, Npts, 2 * RingNum);
  deconvolve(AccBuf, Npts, SampleRate,
              Ring[RingNum].AccelFiltFit, Ring[RingNum].AFiltOrder);
  deconvolve(ForBuf, Npts, SampleRate,
              Ring[RingNum].StrainFiltFit, Ring[RingNum].SFiltOrder);
  deconvolve(AccBuf, Npts, SampleRate,
              Ring[RingNum].AccelCalFit, Ring[RingNum].ACalOrder);
  bandpass(AccBuf, Fcut, Npts);
  bandpass(ForBuf, Fcut, Npts);
  VelBuf := AccBuf;
  freqinteg(VelBuf, Npts, SampleRate, Slope);
  realFFT(AccBuf, Npts div 2, false);
  realFFT(VelBuf, Npts div 2, false);
  realFFT(ForBuf, Npts div 2, false);    // --> time domain
  finishinteg(VelBuf, Npts, Slope);
  removemean(VelBuf, Npts);
  if UnitConversion then begin
    calibrate(AccBuf, Npts, Ring[RingNum].AccelUnits);
    calibrate(VelBuf, Npts, Ring[RingNum].AccelUnits);
    calibrate(ForBuf, Npts, Ring[RingNum].ForceUnits)
  end;
  writetemp(Ring[RingNum].Store, AccBuf, VelBuf, ForBuf, Npts)
end;

if FloatOut then begin
  rewrite(RLFile, CFileName);
  writereal(RLFile, Ring, Npts)
end
else
    printup(output, Ring, Npts)
end.

C.2.3 globals.h

const
    PI           = 3.14159265358979323844;
    MAXSTR       = 100;
    MAXPNT       = 4096;
    MAXIRF       = 64;
    MAXORD       = 10;
    WRINGS       = 6;

type
    complex =
        record
            Re, Im : real
        end end;

cvector        = array [1..MAXPNT] of complex;
tempfile       = file of real;
string         = packed array [1..MAXSTR] of char;

polname        = (num, denom);
filtercoeff    = array [polname] of array [0..MAXORD] of real;

filtvect       = array [1..MAXIRF] of real;

ringrec        =
    record
        AccelUnits,  
        ForceUnits   : real;
        AccelFiltFit : filtercoeff;
        AFilterOrder : integer;
        StrainFiltFit: filtercoeff;
        SFilterOrder : integer;
        AccelCalFit  : filtercoeff;
        ACalOrder    : integer;
        AccMean,     
        ForMean      : real;
        IRF          : filtvect;
        Store        : tempfile
    end
end;

ringinfo       = array [1..WRINGS] of ringrec;

C.2.4 getargs.p

#include 'globals.h'

#include '/usr/hmb/incl/issolid.i'
#include '/usr/hmb/incl/stringend.i'
#include '/usr/hmb/incl/striglen.i'
#include '/usr/hmb/incl/blankstrip.i'

procedure getargs ( {return} var adaptionfilename, 
calfilename : string; 
var floatingout : boolean; 
var outfilename : string; 
var convert : boolean);
{
    Process command line for file names and unit conversion flag.
}

var
    wellformed,
    paramerror : boolean;
    s : string;
    argnum : integer;

begin
    wellformed := false;

    if (argc >= 5) then begin
        argnum := 1;
        repeat
            argv(argnum, s);
            wellformed := (s[1] = '-');
            if wellformed then begin
                if (s[2] = 'a') then begin
                    argv(argnum, s);
                    argnum := argnum + 1;
                    argv(argnum, adaptionfilename)
                end
                else if (s[2] = 'f') then begin
                    argv(argnum, s);
                    argnum := argnum + 1;
                    argv(argnum, calfilename)
                end
                else if (s[2] = 'o') then begin
                    argv(argnum, s);
                    argnum := argnum + 1;
                    argv(argnum, outfilename)
                end
                else if (s[2] = 'u') then
                    convert := false
                else begin
                    paramerror := true;
                    message('adcal: bad flag ', s[2])
                end
            end
            else begin
                paramerror := true;
                blankstrip(s);
                message('adcal: garbled arg ', s:strlen(s))
            end;
            argnum := argnum + 1
        until (argnum = argc) or paramerror
    else begin
        message('adcal: arg count');
        paramerror := true
    end;
}
if paramerror then begin
    message('Usage: adcal -a adfilename -f calfilename ',
            '[-o outfile] [-u] stdin');
    halt
end;

C.2.5 io.p

#include 'globals.h'

function issolid( c : char ) : boolean; external;
procedure blankstrip( var s : string); external;

#include '/usr/hmb/incl/stringread.i'
#include '/usr/hmb/incl/ispow2.i'
#include '/usr/hmb/incl/stringcomp.i'

procedure getfit( {from} var infile : text;
{return} var TtestFit : filtercoeff;
            var Order : integer);
{
    Return rational polynomial coefficients of transfer function
    fitted by TTEST.
    
    The coefficients of the transfer function are stored line-by-line as
    numerator and denominator coefficients, starting from zeroth order.
    Example (6-th order estimate of low-pass filter):
    
    | NUMERATOR     | DENOMINATOR     |
    |---------------|-----------------|
    | 9.99964693269768e-01 | 1.00000000000000e+00 |
    | -2.52966838344665e-02 | 3.14990933798538e-02 |
    | -3.99222871810398e-04 | 5.708579860590048e-04 |
    | -4.29797744469220e-06 | 6.82564980137870e-06 |
    | 3.49666524616678e-08  | 5.84686702534340e-08  |
    | -1.900875185533426e-10 | 2.87891508750770e-10  |
    | 8.125111468965844e-13  | 1.15487652004231e-12  |
}

begin
    readln(infile);
    Order := -1;
    while (not eof(infile)) and (Order < MAXORD) do begin
        Order := Order + 1;
        readln(infile, TtestFit[num][Order], TtestFit[denom][Order])
    end;

    if (not eof(infile)) then begin
        message('adcal: ran out of storage for ttest-fitted coeffs');
        halt
    end;
end;

procedure getcalinfo( {using} var df : text;
{return} var ring : ringinfo;
            var samprate, fcut : real;
            var INFlen : integer);
{
Take directives & filenames from a file named by getargs.

Die if errors are detected. These take the form of sizes of
variables that exceed prescribed values.
}

var
 i : integer;
current : text;
unixname : string;

begin
readln(df);  \{strip header\}
readln(df);
readln(df);

readln(df, samprate);  \{frequency at which data were sampled [Hz]\}
readln(df, fcut);  \{lowpass cutoff as proportion of Nyquist\}
readln(df, IRFLen);  \{length of impulse response to be fitted\}
readln(df);

if ((fcut > 1.0) or (fcut < 0.0)) then begin
message('adcal: lowpass cutoff out of range (0-1)');
halt
end
else if (IRFLen > MAXIRF) then begin
message('adcal: impulse response too long (', IRFLen:1,',')');
halt
end;

for i := 1 to NRINGS do
with ring[i] do
readln(df, AccelUnits, ForceUnits);

for i := 1 to NRINGS do begin
stringread(df, unixname);
blankstrip(unixname);
reset(current, unixname);
with ring[i] do
getfit(current, AccelCalFit, ACalOrder)
end;
readln(df);

for i := 1 to NRINGS do begin
stringread(df, unixname);
blankstrip(unixname);
reset(current, unixname);
with ring[i] do
getfit(current, AccelFiltFit, AFiltOrder);

stringread(df, unixname);
blankstrip(unixname);
reset(current, unixname);
with ring[i] do
getfit(current, StrainFiltFit, SFiltOrder);
end;

procedure refile(  \{from\}  var infile : text;
{update} var ring : ringinfo);

{To save time later in converting test floating point data, write a
formatted store of all the data available on infile. The columns of
infile are assumed to contain accelerometer & strain bridge outputs
alternately. These are stored as real data, alternating
accelerometer & strain bridge outputs on the temporary file for
each ring.

If the first line of input is not 12 numbers wide, die.

Mean values of input are computed for each channel.
}

} type
row = array [1..NRINGS] of real;
var
Avect, Fvect   : row;
J, N           : integer;
Accel, Force   : real;
begin
{check width of first row}
N := 0;
while ((not eoln(infile)) and (N < 2 * NRINGS)) do begin
N := N + 1;
read(infile, Avect[(N + 1) div 2]);
if (not eoln(infile)) then begin
N := N + 1;
read(infile, Fvect[N div 2])
end
end;
if (N <> 2 * NRINGS) then begin
message('adcal: badly formatted input ('', N:1, ' columns')');
halt
end;

{ store first row }
readln(infile);
N := 1;
for J := 1 to NRINGS do
  with ring[J] do begin
    AccMean := Avect[J];
    ForMean := Fvect[J];
    Accel := Avect[J];
    Force := Fvect[J];
    rewrite(Store);
    Store~ := Accel;
    put(Store);
    Store~ := Force;
    put(Store);
  end;

{process the rest of infile}
while (not eof(infile)) do begin
  for J := 1 to NRINGS do begin
read(infile, Accel, Force);
with ring[J] do begin
  Store := Accel;
  put(Store);
  Store := Force;
  put(Store);
  AccMean := AccMean + Accel;
  ForMean := ForMean + Force;
end;
readln(infile);
N := N + 1
end;
for J := 1 to NRINGS do
with ring[J] do begin
  AccMean := AccMean / N;
  ForMean := ForMean / N
end;

procedure readdata( {from} var Store : tempfile;
{read} var AccBuf, ForBuf : cvector;
{return} var npts : integer);
{
  Retrieve the appropriate lot of data from temporary storage.
  Odd-indexed timeseries data are stored in real parts of Zbuf,
  even-indexed in imaginary parts. This process is complicated
  slightly by the fact that the temporary file contains both
  accelerometer and strain bridge data, alternating.

  Check number of data for FFT (power of 2) and for storage size.

  No check is carried out to ensure that npts is the same for each
  set of data read in.
}
var
datum : real;
i : integer;
begin
reset(Store);

npts := 0;
i := 1;
while (!not eof(Store)) and (i <= MAXPNT) do begin
  datum := Store;
  get(Store);
  npts := npts + 1;
  if odd(npts) then
    if ((npts + i) mod 4 = 0) then
      AccBuf[i].Im := datum
    else
      AccBuf[i].Re := datum
  else

if (npts mod 4 = 0) then begin
    ForBuf[i].Im := datum;
    i := i + 1
end
else
    ForBuf[i].Re := datum
end;

if (not eof(Store)) then begin
    message('adcals: filled complex buffer before end of data (',
        MAXPNT:i, ', ');
    halt
end;

npts := npts div 2;  \{ retrieved two lots of data \}
if (not ispow2(npts)) then begin
    message('adcals: require number of data to be a power of 2');
    halt
end
end;

procedure writetemp(\{on\} var Store : tempfile;
    \{put\} var AccBuf,
        VelBuf,
        ForBuf : cvector;
    \{using\} npts : integer);
{
    After processing of data from one transducer, store information in
    temporary file before using processing buffers for next transducer's
    processing.
}

var
    i : integer;
begin
    rewrite(Store);
    for i := 1 to npts do begin
        if odd(i) then begin
            Store^ := AccBuf[(i + 1) div 2].Re;
            put(Store);
            Store^ := VelBuf[(i + 1) div 2].Re;
            put(Store);
            Store^ := ForBuf[(i + 1) div 2].Re;
            put(Store)
        end
        else begin
            Store^ := AccBuf[i div 2].Im;
            put(Store);
            Store^ := VelBuf[i div 2].Im;
            put(Store);
            Store^ := ForBuf[i div 2].Im;
            put(Store)
        end
    end
end
end;

procedure printup( {on} var outfile : text;
{using} var ring : ringinfo;
   npts : integer);
{
Retrieve processed timeseries for each transducer and write on standard
output. Each line of output contains acceleration, velocity, and force
data for each transducer at each timestep (18 columns wide).
}

var
   i, col : integer;
   Accel,
   Velcy,
   Force : real;

begin
   for i := 1 to NRINGS do
      reset(ring[i].Store);

   for i := 1 to npts do begin
      for col := 1 to NRINGS do
         with ring[col] do begin
            Accel := Store^;
            get(Store);

            Velcy := Store^;
            get(Store);

            Force := Store^;
            get(Store);

            write(outfile, Accel:11, Velcy:11, Force:11)
         end;
      writeln(outfile)
   end
end;

procedure writereal( var floating : tempfile;
var ring : ringinfo;
npts : integer);
{
If real output was selected as an option, write the processed timeseries
to a named file of real data. File contains header data (npts, NCOLS),
followed by acceleration, velocity, and force timeseries (in that order),
for each transducer.
}

const
   NCOLS = 18;
   NVARS = 3;
   NHEAD = 18;

var
   i, j, k : integer;

begin
   { Create header }
   floating^ := 1.0 * npts;
   put(floating);
floating" := 1.0 * MGSLS;
put(floating);
for i := 1 to (NHEAD - 2) do begin
  floating" := 0.0;
  put(floating)
end;

for i := 1 to NRINGS do
  for j := 1 to NVARS do begin
    reset(ring[i].Store);
    with ring[i] do
      for k := 1 to npts do
        if (j = 1) then begin { acceleration }
          floating" := Store";
          get(Store);
          put(floating);
          get(Store);
          get(Store)
        end
        else if (j = 2) then begin { velocity }
          get(Store);
          floating" := Store";
          get(Store);
          put(floating);
          get(Store)
        end
        else if (j = 3) then begin { force }
          get(Store);
          get(Store);
          floating" := Store";
          get(Store);
          put(floating)
        end
        else begin { NEVER HAPPEN }
          message('adcal: illegal number in writereal');
          halt
        end
      end
    end
  end
end;

C.2.6 fourier.p

#include 'globals.h'

#include '/usr/hmb/incl/FFT.i'
#include '/usr/hmb/incl/realFFT.i'

C.2.7 timedom.p

#include 'globals.h'

procedure cancelforce( {update} var ForBuf, {using} var ForBuf,
  AccBuf : cvector;
  npts,
  IRFLen : integer;
  var IRF : filtvec;
  AccMean, ForMean : real);
Convolve accelerometer data in Real part of Zbuf with IRF to produce an estimate of inertial component of force transducer output. Subtract the estimate from force transducer output, contained in Imaginary part of Zbuf. Remove mean values before doing computation for each point.


c
var
i, j : integer;
Avect : filtvect;
Fpredict : real;

begin
  for i := 1 to IRFLen do Avect[i] := 0.0;
  for j := 1 to npts do begin
    for i := IRFLen downto 2 do
      Avect[i] := Avect[i - 1];
    if odd(j) then
      with AccBuf[(j + 1) div 2] do begin
        Re := Re - AccMean;
        Avect[i] := Re
      end
    else
      with AccBuf[j div 2] do begin
        Im := Im - AccMean;
        Avect[i] := Im
      end;
    Fpredict := 0.0;
    for i := 1 to IRFLen do
      Fpredict := Fpredict + IRF[i]*Avect[i];
    if odd(j) then
      with ForBuf[(j + 1) div 2] do
        Re := Re - Fpredict - ForMean
    else
      with ForBuf[j div 2] do
        Im := Im - Fpredict - ForMean
  end;

procedure removemean(  {update} var Zbuf : cvector;
  {using}     npts : integer);
{
  Compute the mean of Zbuf, then subtract it from all the data in it.
}

var
i : integer;
Avg : real;

begin
  Avg := 0.0;
  for i := 1 to npts div 2 do
    with Zbuf[i] do
      Avg := Avg + Re + Im;
Avg := Avg / npts;

for i := 1 to npts div 2 do
  with Zbuf[i] do begin
    Re := Re - Avg;
    Im := Im - Avg
  end;

procedure calibrate( {update} var Zbuf : cvector;
{using} npts : integer;
            ConvFact : real);
{
  Convert voltage units to physical units.
}

var
  i : integer;

begin
  for i := 1 to npts div 2 do
    with Zbuf[i] do begin
      Re := Re * ConvFact;
      Im := Im * ConvFact
    end;

C.2.8 deconv.p

#include 'globals.h'

function ispow2(filech : integer):boolean; external;

#include '/usr/hmb/incl/roundpow2.i'

procedure response( {using} freq : real;
                      var transf : filtercoeff;
                      ordr : integer;
                      {return} var answer : complex);
{
  Compute the numerator and denominator of the filter
  frequency response using Hoerner scheme, then divide
  to get a complex magnitude.
}

var
  N, D : complex;
  expnt : integer;
  RePart, Den : real;

begin
  N.Re := transf[num][ordr];
  N.Im := 0.0;
  D.Re := transf[denom][ordr];
  D.Im := 0.0;
for expnt := (ordr - 1) downto 0 do begin
{ Nested multiplication of polynomial coefficients with j*freq. }
RePart := N.Re;
N.Re := - N.Im  * frequency;
N.Im :=  RePart  * frequency;
RePart := D.Re;
D.Re := - D.Im  * frequency;
D.Im :=  RePart  * frequency;
N.Re := N.Re + transf[num][expnt];
D.Re := D.Re + transf[denom][expnt];
end;
{ Complex division of numerator by denominator. }
Den := sqrt(D.Re) + sqrt(D.Im);
answer.Re := (N.Re*D.Re + N.Im*D.Im) / Den;
answer.Im := (N.Im*D.Re - D.Im*N.Re) / Den;
end;

procedure delay( {update} var Zbuf : cvector;
{using}    npts, ChanNum : integer);
{
Correct the data for time delay in analogue-to-digital conversion
caused by multiplexing of A2D converter between channels.

Computation is carried out in the frequency domain where a time
delay corresponds to a phase change increasing linearly with
frequency.

All data are brought back to zero delay relative to first data
collection channel.
}
var
    i : integer;
    Slope,
    Alpha, CosAlpha, SinAlpha,
    temp : real;
begin
    Slope := (ChanNum - 1)  * PI / (npts  * roundpow2(WRINGS));

    for i := 2 to (npts div 2) do begin
        Alpha := (i - 1)  * Slope;
        CosAlpha := cos(Alpha);
        SinAlpha := sin(Alpha);

        {multiply positive frequency part of DFT by exp(-j*alpha)}
        with Zbuf[i] do begin
            temp := Re;
            Re := CosAlpha*Re + SinAlpha*Im;
            Im := CosAlpha*Im - SinAlpha*temp
        end;
    end;
end;

procedure deconvolve( {update} var Zbuf : cvector;
{using}    npts : integer;
           samprate : real;
           var Filter : filtercoeff;
The filter coefficients contain estimates of the poles & zeros of the anti-aliasing filters or accelerometer transfer functions. Multiply data DFT by the inverse of the transfer functions so generated, to estimate the data in its pre-convolved state.

```
var
  i            : integer;
  freq, RePart, Den  : real;
  H            : complex;

begin
  with Zbuf[1] do begin
    Re := Re * Filter[denom][0] / Filter[num][0];
    Im := 0.0  { can't cope with Nyquist frequency }
  end;

  for i := 2 to (npts div 2) do begin
    freq := (i - 1) * samprate / npts;
    response(freq, Filter, Order, H);
    { complex division of Zbuf[i] by H }
    Den := sqr(H.Re) + sqr(H.Im);
    with Zbuf[i] do begin
      RePart := Re;
      Re := (Im*H.Im + RePart*H.Re) / Den;
      Im := (Im*H.Re - RePart*H.Im) / Den
    end
  end;

procedure bandpass( {filter} var Zbuf : cvector;
{using}
  fcut : real;
  npts : integer);

{ Carry out a fairly sharp filter of data in Zbuf.}
{ Lowpass end of filter rolls at fcultimes dataNyquistfrequency. }
{ Use a linear rolloff, nominally 0.05times dataNyquist long. }
{ Highpass by removing the first 1% of the spectral lines. }

const
  HIPASS = 100;

var
  rollstart, rollwidth : integer;
  i, halflength    : integer;
  filtermag : real;

begin
  halflength := npts div 2;
  rollstart := round(fcut*halflength);
  rollwidth := round(0.05*halflength);

  for i := rollstart to (rollstart + rollwidth) do begin
    filtermag := 1.0 - (i - rollstart) / rollwidth;
    with Zbuf[i] do begin
      Re := filtermag * Re;
      Im := filtermag * Im
    end
```
end;

for i := (rollstart + rollwidth + 1) to halflength do
  with Zbuf[i] do begin
    Re := 0.0;
    Im := 0.0
  end;

for i := 1 to (npts div HIPASS) do
  with Zbuf[i] do begin
    Re := 0.0;
    Im := 0.0
  end

C.2.9 integ.p

#include 'globals.h'

procedure freqinteg(  {update} var Zbuf : cvector;
  {using} npts : integer;
  samprate : real;
  {return} var slope : real);
{
  Term-by-term integration of the Fourier Series reconstruction
  of the sampled time function is carried out.
  This is done by dividing each of the DFT coefficients by j*2PIf.
  The mean value cannot be dealt with in the frequency domain---but it can be
  passed back to the time domain & processed there.
  The Nyquist frequency value cannot be dealt with either,
  since the real value of the DFT at the Nyquist frequency will
  become imaginary after division by j*2PIf, and the data packing
  scheme used for all the FTT routines can't cope with that.
  Since anything at the Nyquist frequency will be filtered later in the
  processing in any case, the extra effort doesn't seem worthwhile.
}

var
  factor, temp, twoPIonN : real;
  i : integer;

begin
  twoPIonN := 2.0 * PI * samprate / npts;
  slope := Zbuf[1].Re / npts;

  Zbuf[1].Re := 0.0;
  Zbuf[1].Im := 0.0; { set dc, nyquist values to zero }

for i := 2 to (npts div 2) do begin
  factor := twoPIonN * (i - 1);
  with Zbuf[i] do begin
    temp := Re;
    Re := Im/factor;
    Im := -temp/factor
  end
end
procedure finishinteg( {update} var Zbuf : cvector; 
{using} npts : integer; 
slope : real); 
{
   In the time domain, add on mean value slope correction 
to complete integration of acceleration values.
}
var
i : integer;
begin
for i := 1 to npts do
   if odd(i) then
      with Zbuf[(i + 1) div 2] do
         Re := Re + slope*(i - 1)
   else
      with Zbuf[i div 2] do
         Im := Im + slope*(i - 1)
end;

C.2.10 adapt.p

#include 'globals.h'
#include '/usr/hmb/incl/poweri.i'
#include '/usr/hmb/incl/adaptgain.i'
#include '/usr/hmb/incl/adaptcoeffs.i'

procedure identify( {update} var Ring : ringinfo; 
{using} IRFLen : integer); 
{
   Use method of Fast Recursive Least Squares to construct Finite Impulse 
   Response digital filters, which relate the strain and acceleration 
   responses in the absence of flow, for each transducer.
}
var
time, i, RingNum : integer;
Weight : real;
AdGain, Fcoeff, Bcoeff : filtvect;
ForPredEn, BakPredEn, PredErrRat : real;
X, Y, Error : real;
Xvect : filtvect;

begin
   Weight := (IRFLen + sqrt(32*IRFLen) - 1)/(IRFLen + sqrt(32*IRFLen));
   for RingNum := 1 to NRINGS do
      with Ring[RingNum] do begin
         time := 0;
         for i := 1 to IRFLen do IRF[i] := 0.0;
reset(Store);

while (not eof(Store)) do begin
  { data will be available in pairs }
  X := Store^;
  get(Store);

  Y := Store^;
  get(Store);

  X := X - AccMean;
  Y := Y - ForMean;

  adaptgain(IRFLen, time, X, Weight, Xvect,
            Pcoeff, Bcoeff, AdGain,
            ForPredEn, BakPredEn, PredErrRat);
  adaptcoeffs(IRF, IRFLen, Y,
              PredErrRat, AdGain, Xvect, Error);
  time := time + 1
end
end
C.3 ‘Include’ Files

C.3.1 adaptgain.i

procedure adaptgain( {using} length, 
time : integer;
xnew, 
WEIGHT : real;
{update} var Xold, 
Acoeff, 
Bcoeff, 
Adgain : filtvec;
var ForwPrednEngy, 
BackPrednEngy, 
PrednErrRatio : real);

{ Compute adaption gain for Fast-Least-Squares FIR adaptive filter. }

This procedure is a translation of the FORTRAN routine FLS2 provided by M. Bellanger in "Adaptive Digital Filters & Signal Analysis", Marcel Dekker Inc 1987. ISBN 0-8247-7784-0

Type filtvec = array [1..MAXORD] of real; MAXORD >= length;

} var
G1 : filtvec;
i : integer;
EAV, ALF!,
EPSA,
EPSB,
EAB, EAB!,
G1head : real;

begin
if (time = 0) then begin
for i := 1 to length do begin
Acoeff[i] := 0.0;
Bcoeff[i] := 0.0;
Adgain[i] := 0.0;
Xold[i] := 0.0;
end;
ForwPrednEngy := 1.0;
BackPrednEngy := poweri(WEIGHT, -length);
PrednErrRatio := WEIGHT
end;

EAV := xnew;
for i := 1 to length do
EAV := EAV - Acoeff[i]*Xold[i];

EPSA := EAV / PrednErrRatio;
G1head := EAV / ForwPrednEngy;
ForwPrednEngy := (ForwPrednEngy + EAV*EPSA) * WEIGHT;

for i := 1 to length do
G1[i] := Adgain[i] - Acoeff[i]*G1head;

for i := 1 to length do
Acoeff[i] := Acoeff[i] + Adgain[i]*EPSA;
EAB1 := G1[length] * BackPrednEngy;
EAB := Xold[length] - Bcoeff[i]*xnew;
for i := 2 to length do
   EAB := EAB - Bcoeff[i]*Xold[i - 1];
Adgain[i] := G1head + Bcoeff[i]*G1[length];
for i := 2 to length do
   Adgain[i] := G1[i - 1] + Bcoeff[i]*G1[length];

ALF1 := PrednErrRatio + G1head*EAV;
PrednErrRatio := ALF1 - G1[length]*EAB;
EPSB := (EAB + EAB - EAB1) / PrednErrRatio;
BackPrednEngy := (BackPrednEngy + EAB*EPSB) * WEIGHT;

for i := 1 to length do
   Bcoeff[i] := Bcoeff[i] + Adgain[i]*EPSB;
for i := length downto 2 do
   Xold[i] := Xold[i - 1];
   Xold[1] := xnew
end;

C.3.2 adaptcoeffs.i

procedure adaptcoeffs( {update} var Filter : filtvec;
{using} length : integer;
Ynew,
   PrednErrRatio : real;
var Adgain,
   Xvect : filtvec;
{return} var ErrPosteriori : real);

{ Update FIR filter coefficients using an adaption gain vector.
   Bellanger, Algorithm F.L.S. 2. }

var
  i : integer;

begin
  ErrPosteriori := Ynew;
  for i := 1 to length do
     ErrPosteriori := ErrPosteriori - Filter[i] * Xvect[i];

  for i := 1 to length do
     Filter[i] :=
       Filter[i] + Adgain[i] * ErrPosteriori / PrednErrRatio
end;

C.3.3 FFT.i

procedure FFT (var a : cvector;
   N : integer;
   forwards : boolean);
{
This procedure performs a Fast Fourier Transform on input. In general, the coding follows the description of the algorithm given by Newland in 'An Introduction to Random Vibrations and Spectral Analysis'.

It differs from his description in that division of the coefficients by \( N \) takes place in IDFT, rather than DFT; this corresponds to the definition given by Bendat & Piersol (1971).

Parameter forwards (true, false) specifies DFT, IDFT respectively.

The type cvector is an one-dimensional array of complex elements.
Type complex is defined as;
complex = record Re, Im : real end;
cvector (can be longer than \( N \)) as; cvector = array [1 .. N] of complex;

```
const
PI = 3.14159265358979323844;

var
j, k, l, m : integer;
T, U, W : complex;

procedure bitrev;
begin
j := 1;
l := 1;
repeat
  if (l < j) then begin
    T := a[j];
    a[j] := a[l];
    a[l] := T
  end;
  k := N div 2;
  while (k < j) do begin
    j := j - k;
    k := k div 2
  end;
  j := j + k;
l := l + 1
  until (l = N - 1)
end; {bitrev}

procedure butterfly;
begin
k := 1;
m := 1;
repeat
  U.Re := 1.0; U.Im := 0.0;
  with W do begin
    Re := cos(PI / k);
    Im := -sin(PI / k);
    if not forwards then Im := -Im
  end;
  j := 1;
  repeat
    l := j;
    repeat
      with a[j+k] do begin
        T.Re := Re*U.Re - Im*U.Im;
        T.Im := Im*U.Re + Re*U.Im;
        Re := a[l].Re - T.Re;
      end
    until (l = N - 1)
  until (k = N - 1)
end;
```
\[ Im := a[1].Im - T.Im \]
end;

with \(a[1]\) do begin
\[ Re := Re + T.Re; \]
\[ Im := Im + T.Im \]
end;
\[ l := l + k*2 \]
until \((l > N)\);
with \(T\) do begin \{trig recursion\}
\[ Re := U.Re*W.Re - U.Im*W.Im; \]
\[ Im := U.Re*W.Im + U.Im*W.Re \]
end;
\[ U := T; \]
\[ j := j + 1 \]
until \((j > k)\);
\[ m := m + 1; \]
\[ k := k * 2 \]
until \((k = N)\)
end; \{butterfly\}

begin
if not forwards then
for \(j := 1\) to \(N\) do
with \(a[j]\) do begin
\[ Re := Re / N; \]
\[ Im := Im / N \]
end;
bitrev;
butterfly
end; \{FFT\}

\section*{C.3.4 realFFT.i}

\textbf{procedure realFFT} (var \(DATA\) : cvector;
\(N\) : integer;
forwards : boolean);

\{ -- Compute the \(N+1\) point complex FFT of \(2N\) real data points.
N.B. \(N\) is the length of the packed DFT buffer, and so is \textsc{half} the
number of real data packed in it.

-- To enable the same routine to be used for forward and reverse
transforms, the real data must previously have been packed into
\(DATA\) such that the odd and even indexed points of the
real data are packed into the real and imaginary parts
of \(DATA\) (assuming arrays start at \(1\)),
e.g. \(DATA[1].Re := \text{realdata}[1], \)
\(DATA[1].Im := \text{realdata}[2]\) etc.

-- If the transformation direction is forwards then the packing
must be done prior to calling realFFT, if the transformation
is backwards (IDFT), then the unpacking is performed after
calling realFFT.

-- The DFTs of real data are conjugate symmetric about zero
and the Nyquist frequencies; the routine takes advantage by
packing the real part of the Nyquist frequency DFT point
(imaginary part is zero) into the imaginary part of the
zero frequency DFT point (which is also zero). In this way
\(N+1\) complex frequency points can be produced from \(2N\) real data.
The inverse transform assumes that the DFT data are packed
in this manner.


}

const
PI = 3.14159265358979323844;
c1 = 0.5;

var
i, backi : integer;
wr, wi, wpr, wpi, wtemp, c2, theta, h1r, h1i, h2r, h2i, s1 : real;
s1 : complex;

begin
theta := PI / N;
if forwards then begin
c2 := -c1;
theta := -theta;
FFT(DATA, N, forwards)
end
else
c2 := c1;

wpr := cos(theta);
wpi := sin(theta);
wr := wpr;
wi := wpi;

for i := 2 to (N div 2 + 1) do begin
backi := N - i + 2;
s1 := DATA[backi];
with DATA[i] do begin
h1r := c1 * (Re + s1.Re);
h1i := c1 * (Im - s1.Im);
h2r := -c2 * (Im + s1.Im);
h2i := c2 * (Re - s1.Re);
Re := h1r + wr*h2r - wi*h2i;
Im := h1i + wr*h2i + wi*h2r
end;
with DATA[backi] do begin
Re := h1r - wr*h2r + wi*h2i;
Im := -h1i + wr*h2i + wi*h2r
end;

wtemp := wr; \{trigonometric recursion\}
wr := wr*wpr - wi*wpi;
wi := wi*wpr + wtemp*wpi
end;

if forwards then \{wrap around to fit\}
with DATA[1] do begin
h1r := Re;
Re := h1r + Im;
Im := h1r - Im
end
else begin \{IDFT\}
with DATA[1] do begin
h1r := Re;
Re := c1 * (h1r + Im);
Im := c1 * (h1r - Im)
end;
C.3.5 poweri.i

function mult (x, y: real): real;
const
  SMALLREAL = 9.99e-50;
begin
  if ((x = 0.0) or (y = 0.0)) then
    mult := 0.0
  else if (abs(x) > 1.0) then
    mult := x * y
  else if (abs(y) > SMALLREAL/abs(x)) then
    mult := x * y
  else
    mult := 0.0 { x * y may underflow }
end;

function poweri (x: real; i: integer): real;
{
  Computes x to the power i, where i is an integer.
  Returns 0.0 in cases where x^i would underflow.
  Writes an error message and causes runtime failure
  if x^i is undefined. Avoids underflow failure by
  using the function mult.

}

var
  power : real;
  k    : integer;
  op   : array [1..128] of integer;

begin
  if (x <> 0.0) then begin
    if (i = 0) then
      poweri := 1.0
    else begin
      if (i < 0) then begin
        x := 1.0/x;
        i := abs(i)
      end;
      {record in op the required sequence of operations}
      k := 0;
      while (i > 1) do begin
        k := k + 1;
        if odd(i) then op[k] := 1 {squaring and multiplication}
        else op[k] := 0; {squaring only}
        i := i div 2
      end;
      power := x;
    end;
    while (k > 0) do begin
      power := mult(power, power);
      if (op[k] = 1) then power := mult(power, x);
      if (power = 0.0) then k := 0
      else k := k - 1
    end;
  end;
  power := 1.0;
poweri := power
end
else if (i > 0) then
  poweri := 0.0
else begin
  message ('poweri: x = 0, i <= 0; ',
    'zero argument with nonpositive exponent');
  halt
end
end;

C.3.6 ispow2.i

function ispow2(num : integer) : boolean;
{
  Is num a strictly positive integer power of two?
}
begin
  while ((not odd(num)) and (num>2)) do
  begin
    num := num div 2;
    if (num <> 2) then
      ispow2 := false
    else
      ispow2 := true
  end;
end;

C.3.7 roundpow2.i

function roundpow2(filech : integer) : integer;
{
  Return next integer power of two greater than or equal to filech.
}
begin
  if (filech > 1) then
    if (not ispow2(filech)) do
      filech := filech + 1;
    roundpow2 := filech
end;

C.3.8 issolid.i

function issolid(c : char) : boolean;
{
  Is c a character that prints ink?
}
begin
  if c in [''..''] then issolid := true
  else
    issolid := false
end;

C.3.9 blankstrip.i

procedure blankstrip(var line : string);
{
    Strips prefix whitespace from line, returns a null terminated variable
    of type string. Trailing whitespace (and subsequent characters) are
    ignored.
}

var
    index, start : integer;

begin
    start := 1;
    while not
        (issolid(line[start]) or (start=MAXSTR) or (line[start] = chr(0))) do
        start := start + 1;
    if (start = MAXSTR) or (line[start] = chr(0)) then
        line[1] := chr(0)  {was all whitespace}
    else begin
        start := start - 1;
        index := 1;
        while (issolid(line[start+index]) and ((start+index)<MAXSTR)) do begin
            line[index] := line[start + index];
            index := index + 1
        end;
        line[index] := chr(0)
    end
end;

C.3.10 stringcomp.i

function stringcomp(str1, str2 : string):boolean;
{
    Tests equality of str1 & str2 up to first null (terminator)
    encountered in either string.
}

var
    ENDSTR : char;
    i : integer;

begin
    ENDSTR := chr(0);
    i := 1;
    while ( str1[i] = str2[i] )
    and ( i < MAXSTR ) do
        i := i + 1;
    if ( str1[i] = ENDSTR )
    or ( str2[i] = ENDSTR )
    or ( i = MAXSTR ) then
        stringcomp := true
    else
        stringcomp := false
end;

C.3.11 stringlen.i

function stringlen(var word : string) : integer;
{

How many characters in word?
}

var
count : integer;
begin
  count := 1;
  while not stringend(word[count]) do count := count + 1;
  strlen := count - 1
end;

C.3.12 stringend.i

function stringend(c : char) : boolean;
{
  By convention, chr(0) delimits a string.
}
begin
  if (c = chr(0)) then
    stringend := true
  else
    stringend := false
end;

C.3.13 stringread.i

procedure stringread(var f:text; var line:string);
{
  Reads a line of input from named file. Returns a null terminated
  variable of type string. Preceding spaces are stripped.
  File pointer is left at start of next line.
}
var
  i : integer;
inchar : char;
begin
  i := 1;
  if not eoln(f) then {strip spaces, get first character}
    repeat
      read(f, inchar);
      if issolid(inchar) then begin
        line[i] := inchar;
        i := 2
      end;
    until (eoln(f) or issolid(inchar));
  while not (eoln(f) or (i > MAXSTR)) do begin
    read(f, line[i]);
    i := i + 1
  end;
inchar := chr(0);
readln(f)
end;