Determination of poles and zeroes for the Mermaid response

Guust Nolet*
Géoazur, Université Côte d'Azur and Dept. of Geosciences, Princeton University

Olivier Gerbaud, Fréderic Rocca
OSEAN SARL
Le Pradet, France

June 28, 2021

Contents

*nolet@princeton.edu

Summary

We present the instrument response of the latest generation of Mermaid floats (Hello and Nolet, 2020) in terms of poles and zeroes, as determined experimentally, and describe the software developed for the analysis of the instrument response. The response representative for the current fleet of Mermaids such as currently operational in the South Pacific is (in SAC format):

```
* INPUT UNIT : Pa
* OUTPUT UNIT : COUNTS
POLES     7
   0.50151E-01   0.50405E-01
   0.50151E-01  -0.50405E-01
   0.49249E-01   0.59334E-03
   0.49249E-01  -0.59334E-03
  -0.72882        0.0000
  -0.58397E-01   0.85986E-04
  -0.58397E-01  -0.85986E-04
ZEROS     7
   0.49813E-01   0.48929E-01
   0.49813E-01  -0.48929E-01
   0.55271E-01   0.45316E-01
   0.55271E-01  -0.45316E-01
  -0.23688E-01   0.38878E-01
  -0.23688E-01  -0.38878E-01
   0.   0.
CONSTANT  -0.14940E+06
```

This response does not include the anti-alias filtering, which depends on the sampling frequency of transmitted data (normally 20 sps for body waves).

1 Poles and zeroes

The impulse response at circle frequency ω is given by $R(\omega)$

The recorded output $F_{out}(\omega)$ in Counts is related to the input $F_{in}(\omega)$ in Pa through:

$$F_{out}(\omega) = R(\omega)F_{in}(\omega)$$

We use the Fourier sign convention (conform the SEED format for seismograms):

$$F(\omega) = \int_0^\infty f(t) \exp(-i\omega t)dt\,,$$

The response R is defined by n_z zeroes, n_p poles and an amplification A_0:

$$R(\omega) = A_0 \frac{(i\omega - z_1)(i\omega - z_2)...(i\omega - z_{n_z})}{(i\omega - p_1)(i\omega - p_2)...(i\omega - p_{n_p})}\,.$$

We test the Mermaid response using (approximate) step functions $h(t)$ that are ± 1 for $t > 0$ and 0 when t is negative. In practice, our input signal $f_{in}(t)$ consists of a series of pairs of step functions of opposite sign and an amplitude that is large but avoids saturation.

There is a limit to the frequency we can investigate this way. However, we know that the response is flat at high frequency, where $R(\omega) \rightarrow \omega^{n_z}/\omega^{n_p}$. This constraint imposes that $R(\omega)$ has an equal number of poles and zeroes: $n_z = n_p$. Note that we did not impose this condition in an analysis of the first generation of Mermaids by Joubert et al. (2015), when no information on the high frequency response was available. The approach adopted here is obviously a better one, even though the differences may not be noticeable when one does not go much beyond 1 Hz.

We do not model the effects of the anti-alias filtering and the reduction of sampling rate to limit transmission volumes, which can be variable. We assume the data will only be used at frequencies where that filter has no significant effect.

The Fourier transform of a step function is

$$H(\omega) = (i\omega)^{-1}\,.$$

In our experience, the response of the Mermaid to a step function resembles in the time domain a damped responses of the form:

$$f_n(t) \approx t^n \exp(-\alpha t) \qquad (t,\, \alpha > 0) \quad \text{and 0 for } t \leq 0.$$

The Fourier transform of $f_0(t)$ is:

$$F_0(\omega) = \int_0^\infty e^{-(\alpha + i\omega)t}\, dt = \frac{1}{\alpha + i\omega}$$

and that for $f_1(t)$ is found via partial integration:

$$F_1(\omega) = \int_0^\infty t\ e^{-(\alpha+i\omega)t}\, dt = |_0^\infty \frac{te^{-(\alpha+i\omega)t}}{-(\alpha+i\omega)} - \int_0^\infty \frac{e^{-(\alpha+i\omega)t}dt}{-(\alpha+i\omega)} = \frac{1}{(\alpha+i\omega)^2}\,.$$

Repeating this for higher values of n, one easily finds the recursion

$$F_n(\omega) = \frac{nF_{n-1}(\omega)}{(\alpha+i\omega)}$$

From which:

$$F_n(\omega) = \frac{n!}{(\alpha+i\omega)^{n+1}}$$

This is the response to a step function. The equivalent response to a delta-function is found by dividing out the Fourier transform of the step function input $H(\omega)$, which leads to an observed impulse response:

$$R_n(\omega) = \frac{n!(i\omega)}{(\alpha+i\omega)^{n+1}}$$

It is clear that only $n = 0$ satisfies the condition that $n_p = n_z$, even if the observed response seems closer to an $f_n(t)$ with $n > 0$. But all we are looking for is a reasonable start for the nonlinear optimization. We therefore begin the optimization with $z_1 = 0$ and $p_1 = -\alpha$. To assure zero response at DC we keep the first zero fixed. We add two pairs of conjugates: $z_{2,3} = \gamma \pm i\gamma$ and $p_{2,3} = \gamma \pm i\gamma$ to our starting set, using $\gamma = 0.05$. Note that these are the same in numerator and denominator and initially do not affect the fit obtained with R_0. Using Powell's algorithm in the version from Press et al. (1992), we adjust poles and zeroes (keeping them pairwise conjugate) until the fit is optimal. We can iterate by adding two more identical pairs until the fit does not show significant improvements.

An initial guess for α is obtained by timing the decrease to half of the maximum response. If this happens after t_h seconds, α is found from $e^{-\alpha t_h} = 0.5$ or

$$\alpha \approx \frac{0.69}{t_h}$$

2 Experimental set-up

We briefly describe the experimental set-up. For a more complete description see Gerbaud (2021). In order to generate a useful input signal, the hydrophone is lowered by a distance of up to 10 cm in the water basin at OSEAN. Figure 1 gives an idea of the experimental set-up. The change in depth d of the hydrophone is related to the change in pressure p by $p = \rho g d$, where ρ is the density of water at room temperature (997.77 kg/m^3) and g the local acceleration of gravity (9.82 m/s^2 at the latitude of southern France). The depth of the hydrophone is monitored using a potentiometer,

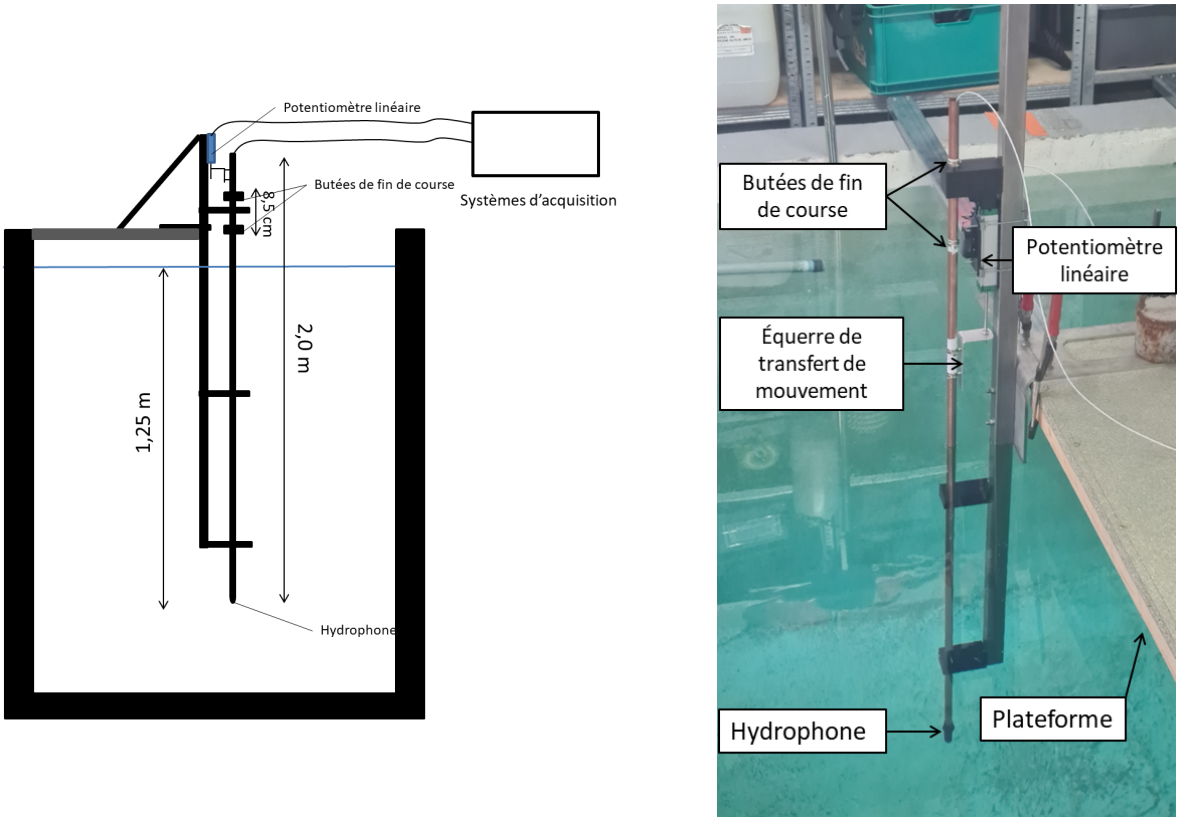

Figure 1: The basic principles of the experimental setup in schematic form (left) and as configured in the test basin at OSEAN (right). From Gerbaud (2021).

by relating the maximum excursion of the hydrophone to the maximum voltage variation in the potentiometer (a camera was used to verify that depth could accurately be deduced this way). The hydrophone was moved by hand in block-like fashion, with pauses of different length ('boxcar' signals). The idea is to average the inevitable noise over a few dozen vertical movements of the hydrophone. We make sure to record the signal until the response last movement is fully relaxed.

3 Noise editing

We did 10 tests, each with several boxcar inputs of different length such as to cover a range of frequencies. A first visual check on the data was done on the Excel files (using plot script `gmtcsv`). This showed some variability in the maximum amplitudes of the responses, probably mostly due to variations in the rise times, which is understandable since the movements were done by hand. For each test we used the 8.5 cm distance between stoppers to calibrate the potentiometer readings and translate them into Pascal. We removed the noise in the potentiometer reading (± 0.04V) by averaging, except during movements.

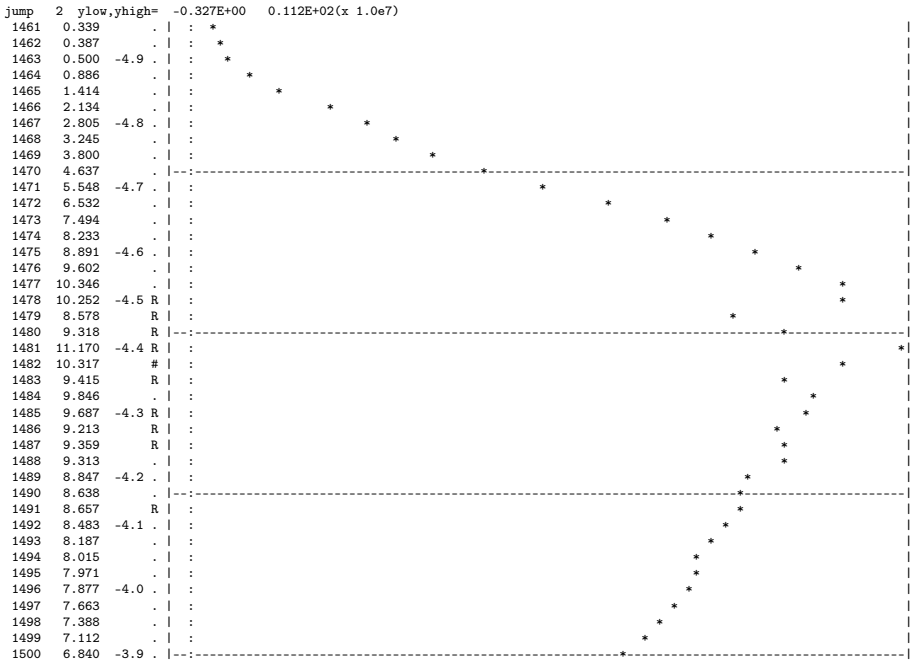

Figure 2: Fragment of the file `out.editcsv` that was used to edit the data. Some points are marked with # in the fourth column, signalling a deviation from a smooth curve, in particular near the peaks. Samples identified as noisy are marked by replacing the # (or dot) with 'R' in the fourth column. The leftmost column lists the sample index, followed by the value of the sample scaled by 10^{-7}, and the time.

The recordings may have some missing data points, show unwanted oscillations near the peak of the response and has occasional noise peaks in the potentiometer channel. We developed a Fortran program `editcsv.f90` that allows for some guided editing as well as automatic filling in of missing data, if any, by linear interpolation. It also attempts to remove spikes from the potentiometer channel. To this end, we convert the binary Excel files to CSV files in ascii format. These CSV files have four columns, with time (s), Potentiometer voltage, hydrophone output (V), and the output of the A/D conversion in Counts (CAN):

```
Temps (s),Potentiomètre (V),Hydrophone (V),CAN,
-41.8692,-0.4,0.96,-3541376,
-41.8442,-0.4,1.08,-3572256,
-41.8193,-0.4,1.04,-3648784,
-41.7942,-0.4,0.96,-3648736,
-41.7693,-0.36,0.88,-3565440,
etcetera
```

The program identifies the sudden movements of the hydrophone and prints the observed data near

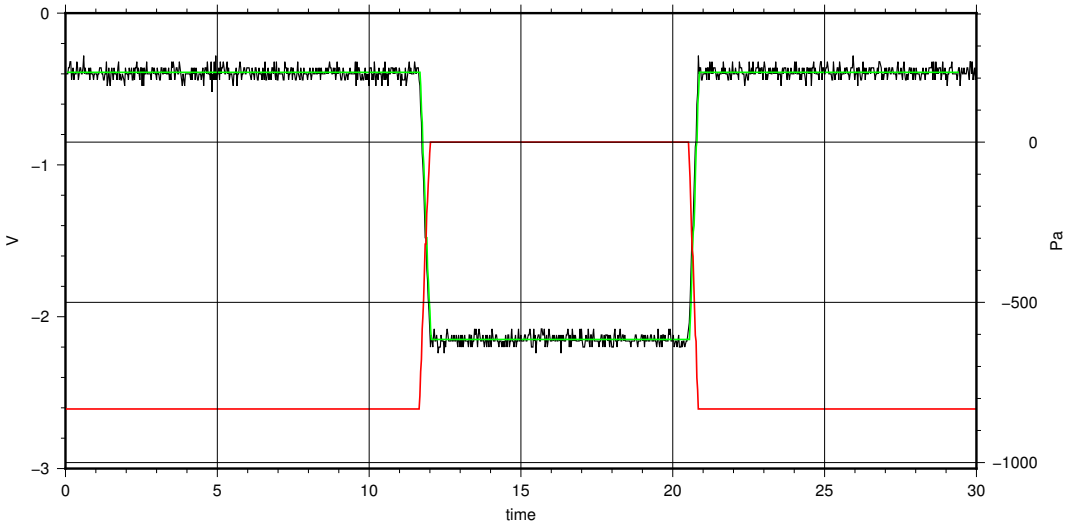

Figure 3: The input signal as recorded by the potentiometer voltage (wiggly black line), and its replacement at the high- and low voltage levels by the average (green line). The red curve and the right vertical axis give the equivalent pressure change in Pascal. Note that the sign of the pressure variation is the reverse of the sign of the voltage, which we attribute to a wrong polarity of the wiring of the card in this test.

the peak to a file `out.editcsv`. (Figure 2).

The user is then directed to an editor (`vi` in the current version, but this can easily be replaced by any other editor), where the user flags bad data with 'R' to indicate data that need to be rejected and subsequently replaced. After rejecting bad data, the missing data are found by interpolation using a harmonic representation of the remaining data over short segments (typically the response to one move) up to a low-pass frequency $\omega_K = 2\pi f_{max}$:

$$y_{int}(t) = \sum_{k=0}^{K} a_k \cos(\omega_k t) + \sum_{k=1}^{K} b_k \sin(\omega_k t)$$

where $\omega_k = 2\pi k/T$ for a segment of T seconds. The coefficients a_k and b_k are found from the 'correct' data using least squares.

Missing data were fixed by linear interpolation using the neighbouring samples (if two or more consecutive samples are missing the program will stop with an error message, but this has not happened).

One can also flag data with 'S' to identify segments that are too noisy and should be split at the time of the 'S' flag – they can then be removed altogether (which needs to be done by hand afterwards).

As an example, Figure 4 shows how this worked for the response shown in Figure 2. In this example 9 data points were rejected and replaced by their interpolated values. It is clear that this leads to a much better quality result. We found an effective low-pass frequency f_{max} of 2.0 Hz by repeating the fitting for a range of frequencies, starting at 0.5 Hz.

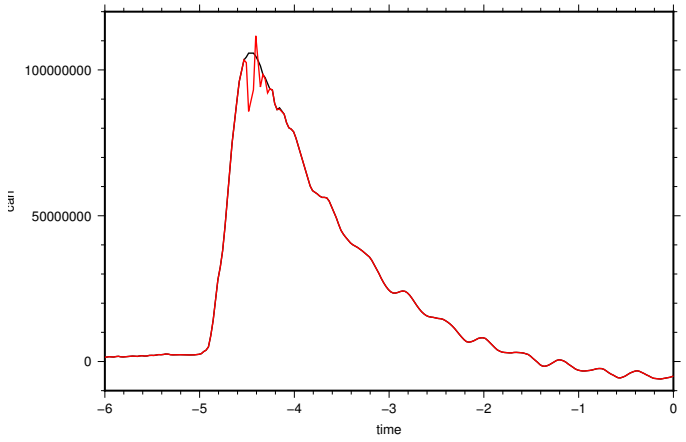

Figure 4: The raw output (red) and the interpolation after rejecting the noisy samples near the maximum response (black line).

Since we assume that the input is in the form of 'boxcar' signals with constant pressure, `editcsv` identifies the plateaus of constant potentiometer voltage and replaces the noisy signal there with a constant value, at both the lower and the higher plateau. We use the difference in depth to then translate this into pressure in Pascal (Figure 3).

Note that the hydrophone has a *positive* response in Counts when the pressure goes *down* (in other words, when the hydrophone goes up in the water). We therefore expect a negative value for the amplification A_0.

Recommendation: Since there should never be any ambiguity about the polarity of Mermaid recordings, I recommend that every Mermaid is tested in the basin after it has been fully assembled. Lowering the float a few cm and bringing it up should enable one to check the wiring of the hydrophone: the response in Counts should go **up** *when the Mermaid is* **raised** *to a shallower depth.*

This editing needs to be done for all tests before submitting the data to analysis for poles and zeroes in program `getpz2`.

The potentiometer and the CAN (Mermaid) have different A/D converters. From a brief inspection of the data sent back from the SPPIM experiment it appears that the sampling interval (as determined by GPS) has minor fluctuations (a random sample showed average $\Delta t = 0.0499253\text{s} \pm 0.2 \times 10^{-6}\text{s}$ in data transmitted at 20 Hz). Although such a small deviation from the theoretical value of 0.025 (at 40 Hz) has no important influence on the response itself, there is a danger that the two will be out of sync near the end of the signal. A separate program `getdeltat.f90` is able to determine the correct sampling interval if a resampling is needed to bring the two in sync. In the present data set, this turned out not to be necessary.

4 Estimating poles and zeroes

The MERMAID response is supposed to be flat above a certain frequency. As noticed in section 1, a constant response at high frequency imposes $n_z = n_p$, an equal number of poles and zeroes.

```
* INPUT UNIT : Pa
* OUTPUT UNIT : COUNTS
POLES     7
   0.50151E-01   0.50405E-01
   0.50151E-01  -0.50405E-01
   0.49249E-01   0.59334E-03
   0.49249E-01  -0.59334E-03
  -0.72882           0.0000
  -0.58397E-01   0.85986E-04
  -0.58397E-01  -0.85986E-04
ZEROS     7
   0.49813E-01   0.48929E-01
   0.49813E-01  -0.48929E-01
   0.55271E-01   0.45316E-01
   0.55271E-01  -0.45316E-01
  -0.23688E-01   0.38878E-01
  -0.23688E-01  -0.38878E-01
   0.   0.
CONSTANT  -0.14940E+06
```

Figure 5: The 'sacpz' file that allows one to convert the transmitted Mermaid signal from Counts to Pascal.

For this to occur at a frequency well below the Nyquist, the absolute values of the poles and zeroes must be much smaller than the Nyquist frequency, which we promote by adding a damping term to the penalty function in the nonlinear optimization. We also impose a zero response at $\omega = 0$. Therefore at least one zero is equal to 0. We also impose that the poles and zeroes are either real, or they occur in complex conjugate pairs.

Program `getpz2.f90` satisfies all these constraints. It uses Powell's algorithm to search for the best set of poles and zeroes, starting from an initial guess. For the initial guess we use $R_0(\omega)$, i.e. one real pole and one real zero:

$$R_0(\omega) = \frac{i\omega}{\alpha + i\omega}$$

where α was estimated from $t_h = 1.2$s (see Section 1).

Program `getpz2` will generally start from a such an initial set, though the user may provide a different and more complete initial set. If so, one should provide at least one zero equal to (0,0) and make sure $n_z = n_p$, even though the program will accept more arbitrary sets. Note that the amplification factor A_0 is not defined as a variable parameter in Powell's algorithm. Rather, for every set of poles and zeroes, we compute A_0 such as to equalize the total energy of in- and output signals before computing the misfit penalty.

We did ten different experiments, each with several boxcar-type input signals for a total of 31 boxcars. An example is shown in figure 6. All ten responses were assembled in a single data volume for optimization with poles and zeroes. We added poles/zeros until the fit did not improve significantly. We iterated with $n_p = n_z = 5, 7, 9$, but the last iteration showed convergence with a response practically identical to that with 7 poles. The optimal fit, obtained with 7 poles and 7

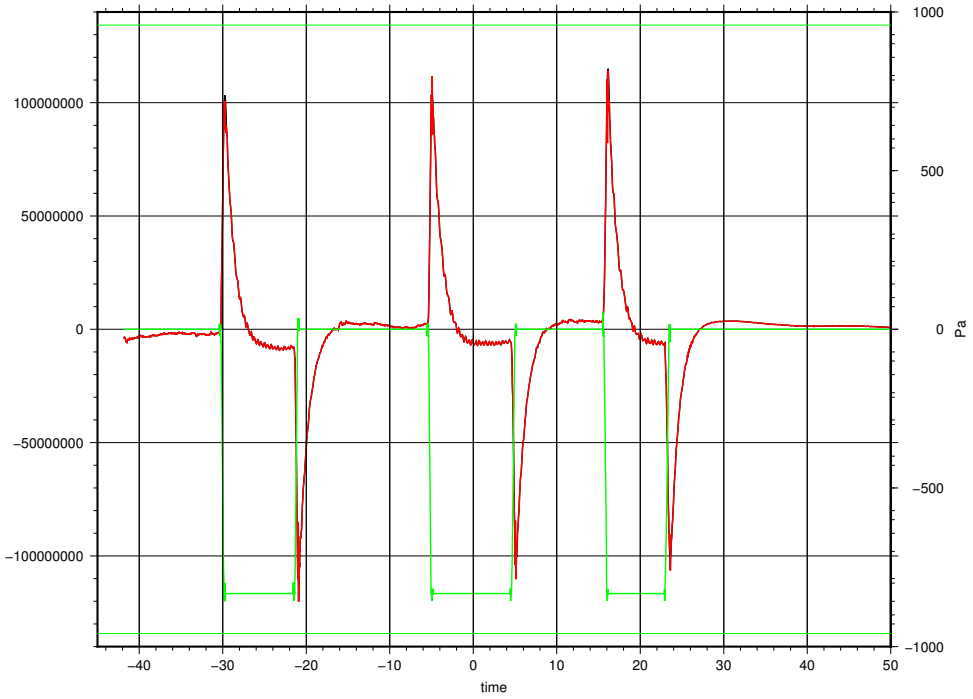

Figure 6: The input signal in Pa (green) as derived from the potentiometer signal, the response after correcting peak oscillations (red) and – almost invisible – the response before corrections (black), for the first test in Counts. The threshold above which the signal saturates is indicated by green horizontal lines near the top and the bottom.

zeroes, and an amplification (negative, to correct the polarity of the hydrophone output to that of the input pressure) that translates 'Counts' as transmitted by the Mermaid into Pascal, is shown in Figure 9. To use this in SAC, one issues a command such as:

```
trans from polezero S sacpz to none freqlimits 0.4 0.2 4 8
```

(SAC will signal an error because it has not yet been adapted to accept Pascal as physical unit, but this error can be ignored).

This response does not take into account the possible effect of the anti-alias filtering and low-pass data compression by the wavelet transform, which depends on the sampling rate of the data that are actually transmitted (e.g. 20 Hz instead of the native 40 Hz). This is probably not important for body-wave signals and the 20 Hz data stored at the IRIS data center, but for possible surface wave data transmission by request at rates below 4 Hz one may need to add a filter stage adapted for that transmission.

Figures 7 and 8 give an idea of the fit of the response as predicted with this pole/zero set and the amplification.

The amplitude response, which should be flat at high frequency, is shown in Figure 9.

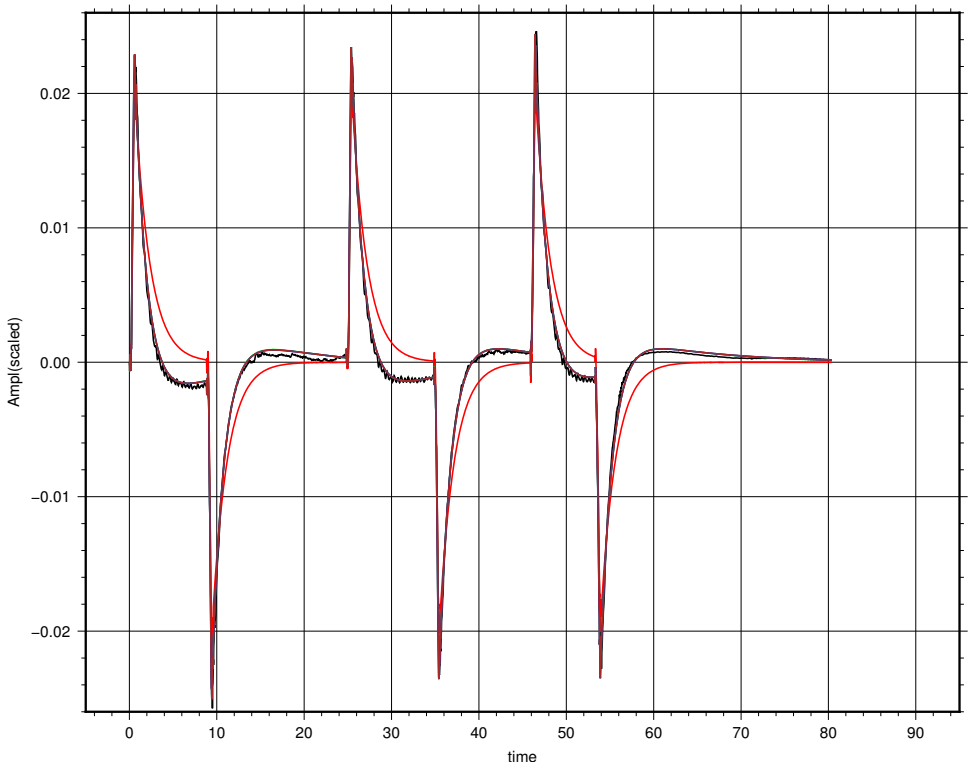

Figure 7: The observed response (black), the fit of the response using $R_0(\omega)$ before optimization (red), and the fit obtained with 9 poles/zeroes (brown). Two earlier iterations are also plotted (blue and green) but disappear behind the brown curve, as the are within the line thickness.

To get an idea of the precision of the amplification factor, we inspected the variability of the difference between the maximum and minimum response after each boxcar input (using the CAN files, i.e. after editing with `editcsv`). The average difference was $0.20 \pm 0.02 \times 10^9$ Counts, where the uncertainty represents one standard variation. This variability of 10% is partly due to slight variations in the duration of the rise, and not due to perturbations, so it represents a conservative estimate. The error in the average amplification over the 31 inputs is thus likely of the order of about 2%. To this must be added the uncertainty in the length of the displacement which we judge to be below 2 mm, or also about 2%. We therefore consider the amplification to have an uncertainty, at the level of one standard deviation, of $\sqrt{2^2 + 2^2} \approx 3\%$.

5 Example: P wave from Honshu earthquake of May 1, 2021.

To test the resulting pole/zero file on an actual seismogram, we use a recent recording of a P wave from an earthquake of magnitude 6.8 near the island of Honshu, Japan. This event occurred on May 1, 2021 at 01:27:27.6 UTC and was located at 38.23°N, 141.66°E at a depth of 47 km. The

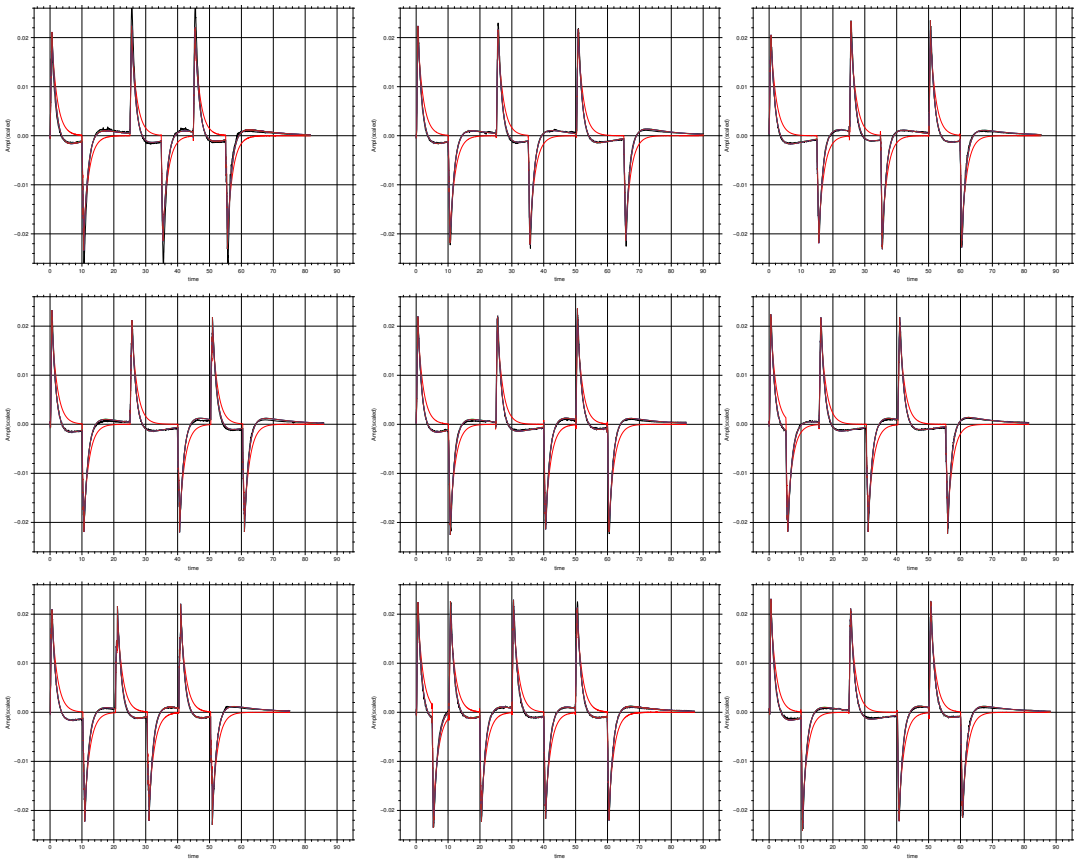

Figure 8: As Figure 7, showing the fit for the other nine tests.

Mermaid was a float of the new 'Lander' type, which is a sort of hybrid between the freely drifting Mermaids and an OBS: rather than drifting, it is programmed to set foot on the seabottom in an area of interest, and come up either at regularly spaced intervals or whenever a strong event triggers it, for transmission of data. Thus, contrary to an OBS, it provides at least some of its data in quasi-real time. The Mermaid of interest to this report was operating off the coast of Nice in the Mediterranean at 43.487°N 7.238°E, depth 1043m. Its distance to the Honshu quake is 88.8°.

Figure 10 shows the P wave signal observed by the Lander in Counts and this signal after conversion to Pa, the correct physical units. On the right of this we show the same P wave as recorded in station MON of the RESIF network, located on land at a distance of 110 km from the Lander. For the conversion we used a trapezoidal bandpass between 0.04 and 4 Hz, which was tapered on either side to become 0 below 0.02 Hz and beyond 8 Hz. The pressure signal (bottom left) starts with a small upswing with an amplitude of about 3 Pa before showing a major downswing to −10 Pa. Theoretically, pressure signal P and vertical particle velocity v_z relate as:

$$P = \sqrt{\kappa\rho}\; v_z$$

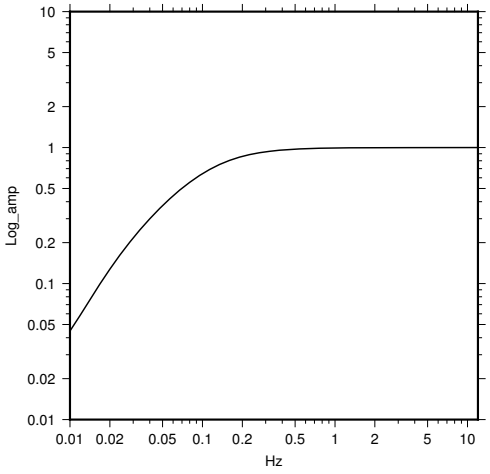

Figure 9: The amplitude response is flat at high frequency.

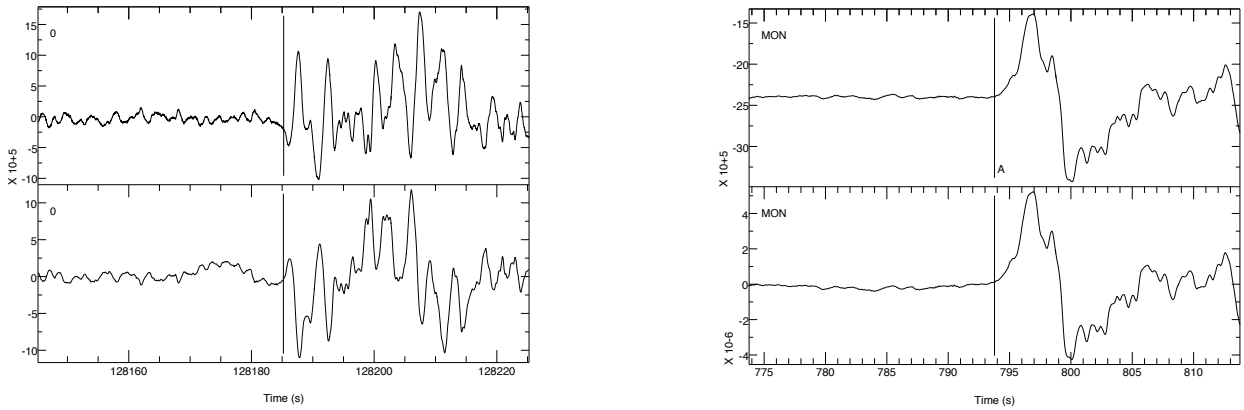

Figure 10: The P wave signal recorded by the Lander for the Honshu earthquake is shown on the left, before conversion in Counts (top) and after conversion to Pa (bottom). On the right we show the raw signal as recorded by RESIF broadband station MON before (top) and after conversion to velocity (m/s, bottom). The time origin for the two insyruments is different and arbitrary. The pick labeled A indicates the onset of the P wave.

where κ is the bulk modulus and ρ the density. Using a typical value of 2×10^9 Pa for κ, we find that $P \approx 1.4 \times 10^6 \; v_z$. The amplitudes in the bottom plots agree with this (rough) order of magnitude computation. Note that the waveforms of the two signals must be different because the pressure signal in the Lander contains the 'ghost' signal, the reflection off the water surface that arrives after about 1.3 s and that may be followed by further reverberations with reflections off surface, sediment and bedrock.

Because there is a (practically) zero pressure boundary condition at the sea surface, reflections have the opposite sign of the incoming wave. Combining this with a delay Δt, the ghosts thus act as if they differentiate the signal. Low frequencies, for which the delay is not significant, will be damped because the reflected wave cancels the incoming wave. But for high frequencies they act as

a differentiator. This explains why the pressure signal on the left is high-passed with respect to the land seismogram on the right.

Note that the polarity of the Lander pressure signal agrees with the onset of the signal in MON and the unconverted signal is indeed reversed. This polarity reversal was long suspected when we compared data from the previous Mermaid generation with Apex floats. Nolet et al. (2019) actually applied a polarity correction when comparing waveforms from Mermaids with those from island stations to help recognize the onset of P-wave in noisy data. But is was never detected by Joubert et al. (2015), who in the laboratory experiments inadvertently wired the hydrophone such as to have pressure and hydrophone output with the same polarity when determining the poles and zeros of that version, unaware that this differed from the actual wiring in the Mermaid.

References

[1] O. Gerbaud. Fonction de transfert de la chaine d'acquisition Mermaid. Internal Report, OSEAN SAS (Le Pradet, France), 2021.

[2] Yann Hello and Guust Nolet. Floating Seismographs (MERMAIDS). In Harsh K. Gupta, editor, *Encyclopedia of Solid Earth Geophysics*, pages 1–6. Springer International Publishing, 2020.

[3] C. Joubert, G. Nolet, A. Sukhovich, A. Ogé, J.F. Argentino, and Y. Hello. Hydrophone calibration at low frequencies. *Bull. Seismol. Soc. Am.*, 105:1797–1802, 2015.

[4] G. Nolet, Y. Hello, S. van der Lee, S. Bonnieux, M. C. Ruiz, N. A. Pazmino, A. Deschamps, M. M. Regnier, Y. Font, Y. J. Chen, and F. J. Simons. Seismic tomography with floating seismometers: a first application to Galàpagos. *Sci. Reports*, 9:1326, 2019.

[5] W.H. Press, S.A. Teukolsky, W.T. Vettering, and B.P. Flannery. *Numerical Recipes*. Cambridge University Press, Cambridge, UK, second edition, 1992.

A Appendices: Fortran codes used

The programs `editcsv` to edit the CSV files and `getpz2` to analyse the response in terms of poles and zeroes are written in Fortran90 (with some older subroutines in Fortran77). Public domain compiler `gfortran` was used to compile them on a Mac PowerBook, but this compiler is widely available for other systems.

For convenience (e.g. for checking results with the `transfer` command) some of the output of `getpz2` is in the form of SAC files (Seismic Analysis Code), available from `https://ds.iris.edu/ds/nodes/dmc/software/downloads/sac/`. If SAC is not available one can simply comment the calls to SAC routines (newhdr, setfhv, setkhv and wsac0) and the program will still work. It also writes plottable files in simple 'x-y' format that can be plotted with some of the GMT scripts provided with this package (see `https://www.generic-mapping-tools.org/download/`).

A.1 Editing the CSV file: a user guide

To compile the fortran code:

```
gfortran -o editcsv editcsv.f90
```

Since the hydrophone has to move fast, the sudden stops at both ends tend to introduce a few reverberations in the response that need to be removed. In addition, there may be data missing in the Excel file, and the potentiometer recording can be noisy. One should therefore subject each recorded output from the experiment to program `editcsv`. Normally, one gets the data in form of an Excel spreadsheet that needs to be exported to CSV format with four columns (see section 3 for an example. A first check of the data can be done with GMT script `gmtcsv`, e.g. to plot file `test1.csv` one calls:

```
gmtcvs test1
```

If everything is fine, one then calls `editcsv` with the file name and the distance moved by the hydrophone in the watercolumn (in cm), e.g.:

```
editcsv test1.csv 8.5
```

The user is then brought into the `vi` text editor, and presented with an editable printplot of the data (see Figure 2 for an example). Every data point that is marked with capital 'R' in the fourth column will be replaced by a value interpolated from the accepted data points. The program may suggest suspect data points by labeling them with a '#' but this is not perfect and the user should still confirm this with an 'R'.

The output of this program is in the form of a `*.can` file (e.g. `test1.can`) that serves as input for `getpz2.f90`. One can check on the quality of the edits by running GMT script `gmtcan` without any arguments, or plot a zoom between times t_1 and t_2 by calling `gmtcan t1 t2`. This script reads

a file `can.xy` that will be overwritten by the next run of `editcsv` – if you wish to preserve them you should rename them or – simpler – use a different directory for each CSV file to be edited. Files `diagnostics.editcsv` and `out.editcsv` can be consulted for quality checks and may provide information if the editing produces unwanted results.

A.2 Analysing the CAN file: a user guide

To compile the Fortran code:

```
gfortran -o getpz2 getpz2.f90 getpzsubs.f sacio.a
```

This is the program that does the real work: find a set of poles and zeros that represents the instrument response (i.e. the response to a delta-function – even though we are using boxcar functions to derive the poles and zeros).

Before running `getpz2` you must create an input file that lists the CAN files to be used, e.g.:

```
'test1/test1.can'
'test2/test2.can'
...
'test8/test8.can'
```

(if the can files are all in the current directory, there is no need to put them in quotes).

The program itself is interactive and asks for input from the screen. The session that gave the result given in this report went as follows:

```
> getpz2
(...)
 There are three options for the starting set of poles and
 zeroes: when nothing else is known, give the time (in s) at which
 the response has relaxed to half its maximum (measured from t=0)
 If this is a Mermaid from a known series, you can answer
 <default> (without the <>) and it will start from the standard
 response. If you have a result from an earlier iteration saved
 in a file, give that file name. Getpz2 saves the latest result
 in file <inpz>.
 Give time to half relaxation, or file name, or <default>:
1.2
 Give low pass frequency, or 0 for no filter (e.g. 4 Hz):
0
Initial misfit:   0.116
Relative misfit:  1.000
 Now optimizing...have patience
```

```
 Calling powell with ndim=           5
 Powel returns fret=   1.48246419E-02
Final misfit: 0.148E-01
Relative misfit:  0.128
 Add more poles/zeroes (1) or stop (0)?
1
 Now optimizing...have patience
 Calling powell with ndim=           9
 Powel returns fret=   1.41166048E-02
Final misfit: 0.141E-01
Relative misfit:  0.122
 Add more poles/zeroes (1) or stop (0)?
1
 Now optimizing...have patience
 Calling powell with ndim=          13
 Powel returns fret=   1.40200416E-02
Final misfit: 0.140E-01
Relative misfit:  0.121
 Add more poles/zeroes (1) or stop (0)?
1
 Now optimizing...have patience
 Calling powell with ndim=          17
 Powel returns fret=   1.39726335E-02
Final misfit: 0.140E-01
Relative misfit:  0.121
 Add more poles/zeroes (1) or stop (0)?
0
```

As one can see, there are three options for choosing the necessary starting values of poles and zeros. If no prior information is available one should measure the time in seconds between the start of the boxcar response and the time when the response is back to half its maximum first deflection. As explained in section 1, the program then uses this to start with three poles and two zeros (plus one zero to be kept equal to 0).

If the hydrophone is of the same type as the one analysed in this report, one can specify `default` and the program will start from the set of 7 poles/zeros reported here.

Finally, if you have your own file with a set of starting pole/zeroes you can give that file name (in SAC format without the two lines at the start with INPUT and OUTPUT).

The low-pass filtering is optional. We have not found it necessary.

As long as the total number of poles and zeroes (ndim) does not exceed 40, the program will ask the user whether to continue with adding an additional pair of complex conjugate poels and zeroes or to stop. If the misfit does not improve in the last iteration, the set obtained in the one before

last iteration is just as good. The results of all iterations are combined in file `inpz`, from which a good SAC-useable file can be extracted by hand.

The quality of the fit can be judged by plotting results with script `gmtpz2`. Further information can be found in files `out.getpz2` and `diagnostics.getpz2`.

Note: we assume that the two columns that we read from the CSV files are perfectly synchrone even though derived from different clocks (in the oscilloscope for the potentiometer, in the electronic card for the CAN).

However, if doubts exist, run `getdeltat.f90` on one of the files to find the sampling interval of the CAN (dtcan) and uncomment these lines so that you can tell `getpz2` to synchronize the CAN with the potentiometer:

```
print *,'We assume you have found the correct sampling interval for'
print *,'the Mermaid card (CAN) using program getdeltat. If you'
print *,'answer 0, it will assume CAN and Tektronics have the same dt.'
print *,'Give the true sampling interval for the CAN (in s):'
read(5,*) dtcan
```

A.3 Codes

The programs included in this package are free software: you can redistribute them and/or modify them under the terms of the GNU General Public License as published by the Free Software Foundation, version 3.

The programs are distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details (`https://www.gnu.org/licenses`).

A.3.1 editcsv.f90

```
program editcsv

! Compile: gfortran -o editcsv editcsv.f90

! This program is used to edit the data obtained with instrument repsonse
! tests in the basin of OSEAN - in which the hydrophone is lowered by
! a few cm (variable: depth)
! For input, transform Excel files into CSV format with four columns:
! time (s), Potentiometer (V), hydrophone (V) and CAN (counts)
! Its skips the first (header) linea in the csv file.
```

```
! It writes input for program getpz2.

! Usage: editcsv filename depth

! Note: the CSV files received from OSEAN have the fourth column (CAN)
! ahead by one sample with respect to the second column (potentiometer).
! This is corrected by this program, immediately after reading the CSV file.

implicit none

integer, parameter :: ND=8192          ! max signal length 2^13 samples

real*4 :: average
real*4, allocatable, dimension(:,:) :: a        ! matrix for LSQR
real*4 :: can(ND)                      ! CAN reading in counts
real*4 :: can0(ND)                     ! CAN reading before editing
real*4 :: density=997.77               ! water density at room temperature
real*4 :: depth                        ! depth range of hydrophone moves
real*4 :: dt,dt1,dt2,halfdt            ! sampling interval, dt/2
real*4 :: dum
real*4 :: dw                           ! frequency step
real*4 :: dy(ND),dy2(ND)               ! 1st and 2nd derivates of can
real*4 :: fmax=2.0                     ! Hz, max frequency used in fitting
real*4 :: g=9.82                       ! local acceleration of gravity
real*4 :: hyd(ND)                      ! hydrophone voltage
integer :: itmax=100                   ! max nr of iterations of LSQR
integer :: histogram(200)              ! potentiometer voltage histogram
integer :: jmp                         ! numer of sudden signal changes
integer :: kbreak(10)                  ! points where to split the signal
integer :: kjmp(200),kjmp1(200)        ! jump indices
integer :: nf                          ! nr of frequencies used in fitting
integer :: nsmp                        ! length of total signal
integer :: m                           ! number of coefficients
real*4 :: pot(ND)                      ! potentiometer V, converted to Pa
real*4 :: pDC                          ! assumed rest pressure
real*4 :: pmid,pmin,pmax               ! mid,min,max pressure or Voltage
real*4 :: PperV                        ! Pa per Volt conversion factor
real*4 :: r                            ! residual of fit by LSQR
integer :: relax=160                   ! nr of samples for relaxation
real*4 :: som,som1,som2                ! used for summing/averaging
real*4 :: time(ND)                     ! time axis for can,pot,hyd
real*4 :: tjmp(200)                    ! time of jumps (start of moves)
real*4 :: t                            ! time variable
real*4 :: tlen                         ! length of segment to edit in sec
real*4 :: twopi=6.283185
real*4 :: ty(ND)                       ! time axis for y()
real*4 :: w                            ! circle frequency
real*4 :: x(257)                       ! fitting coefficients
```

```
real*4 :: xlim                          ! noise level estimate
real*4 :: ylow,yhigh,diff               ! limits of can in segments
real*4 :: y(ND)                         ! edited signal segments
character*1 :: plot(100)                ! for editable print plot
character*1 :: flag,q                   ! for editing
character*5 :: ctime
character*80 :: fname
character*124 :: line
integer :: i,i1,i2,ib,ios,iout,j,k,k0,n,nmin,nmax,ny,npow

open(4,file='out.editcsv',action='write')
open(10,file='diagnostics.editcsv')
n=command_argument_count()
if(n<2) then
  print *, "Usage: editcsv file.csv depth (in cm)"
  stop
endif
call get_command_argument(1,fname)
open(1,file=fname,action='read')
read(1,*)                       ! skip header
write(4,'(2a)') 'CSV file: ',trim(fname)
call get_command_argument(2,line)
read(line,*) depth
write(4,'(a,f8.2,a)') 'Depth range: ',depth,' cm'
depth=depth*0.01                ! convert to m

j=0
k=0
pmax=-1e20
pmin=1e20
histogram=0
i=0
do
  i=i+1
  read(1,*,iostat=ios) time(i),pot(i),hyd(i),can(i)
  if(is_iostat_end(ios)) exit
  if(i.eq.2) dt1=time(2)-time(1)
  if(i.eq.3) dt2=time(3)-time(2)
  if(i.eq.3) write(10,'(a,2f10.4)') 'dt1, dt2=',dt1,dt2
  if(i.eq.4) then               ! get first estimate of dt
    if(abs(dt1-dt2)>0.002) stop 'Check on dt in first 3 samples'
    dt=0.5*(dt1+dt2)
    dt1=0.
  endif
  ! repair missing data
  if(i>4) dt1=time(i)-time(i-1)-dt
  if (i>3.and.dt1>0.002) then
    j=nint(time(i)-time(i-1))/dt
```

```
    if(j>1) stop 'Too many consecutive samples are missing'
    time(i+1)=time(i)
    pot(i+1)=pot(i)
    hyd(i+1)=hyd(i)
    can(i+1)=can(i)
    time(i)=0.5*(time(i-1)+time(i+1))
    pot(i)=0.5*(pot(i-1)+pot(i+1))
    hyd(i)=0.5*(hyd(i-1)+hyd(i+1))
    can(i)=0.5*(can(i-1)+can(i+1))
    k=k+1
    i=i+1
    dt1=0.
    if(k.eq.1) write(10,'(a,/,a)') 'Interpolated missing data',  &
      '    i         t        can'
    write(10,'(i5,f10.3,i11)') i,time(i),nint(can(i))
  endif
  pmax=max(pot(i),pmax)
  pmin=min(pot(i),pmin)
  j=nint((pot(i)+4.0)*25.0)
  histogram(j)=histogram(j)+1
enddo
close(1)
nsmp=i-1

! the CAN runs one sample ahead of the pot, we throw out that first sample
do i=1,nsmp-1
  can(i)=can(i+1)
enddo
nsmp=nsmp-1

can0=can                ! save unedited version for plots

! find upper- and lower constant levels for potentiometer
pmid=0.5*(pmax+pmin)
write(10,*) 'min, max pot=',pmin,pmax
nmin=1
nmax=1
do i=1,nsmp
  j=nint((pot(i)+4.0)*25.0)
  if(pot(i)<pmid) then
    if(histogram(j)>histogram(nmin)) nmin=j
  else
    if(histogram(j)>histogram(nmax)) nmax=j
  endif
enddo
pmax=nmax/25.-4.0
pmin=nmin/25.-4.0
write(10,'(a,2f8.2)') 'Histogram extrema: ',pmin,pmax
```

```
! find average on both upper and lower plateau
pmid=0.
nmax=0
nmin=0
som1=0.
som2=0.
do i=1,nsmp
  if(abs(pot(i)-pmax)<0.1) then
    nmax=nmax+1
    som2=som2+pot(i)
  else if(abs(pot(i)-pmin)<0.1) then
    nmin=nmin+1
    som1=som1+pot(i)
  endif
enddo
pmax=som2/nmax          ! average high constant level
pmin=som1/nmin          ! average low constant level
write(10,'(a,2f8.2)') 'Average constant voltage at ',pmin,pmax

! remove noise
! we try to detect spikes of amplitude up to 0.5V
do i=2,nsmp-1
  i1=max(1,i-20)
  i2=min(nsmp,i+20)
  ! do nothing if we are near jump
  if(abs(pot(i1)-pot(i2))>1.5) cycle
  if(abs(pot(i-1)-pmax)<0.5 .and. abs(pot(i)-pmax)<0.5 .and.  &
     abs(pot(i+1)-pmax)<0.5) pot(i)=pmax
  if(abs(pot(i-1)-pmin)<0.5 .and. abs(pot(i)-pmin)<0.5 .and.  &
     abs(pot(i+1)-pmin)<0.5) pot(i)=pmin
enddo
pDC=pmin
if(abs(pot(1)-pmax)<abs(pot(1)-pmin)) pDC=pmax
pot(1)=pDC
pot(nsmp)=pDC

PperV=depth*g*density/(pmax-pmin)
write(10,'(a,f12.2)') 'Pascal per Volt =',PperV
do i=1,nsmp
  dum=pot(i)
  pot(i)=(pot(i)-pDC)*PperV            ! Convert V to Pa
enddo
dt=(time(nsmp)-time(1))/(nsmp-1.0)
halfdt=0.5*dt

write(4,'(a,i7,a)') 'The CVS file has ',nsmp,' samples'
write(4,'(a,f8.4,a)') 'Sampling interval:',dt,' sec'
```

```
write(4,'(a,f8.1,a,f8.1)') 'Times range from ',time(1),' to ',time(nsmp)
if(k>0) write(4,'(a,i5,a)') 'WARNING: the file missed ',k,' samples'
if(k>0) write(6,'(a,i5,a,/,a)') 'WARNING: the file missed ',k,  &
  ' samples','Missing data have been linearly interpolated, see diagnostics'

! get first and second derivatives of can
dy(1)=can(2)-can(i)
do i=2,nsmp-1
  dy(i)=can(i+1)-can(i-1)
enddo
dy(nsmp)=can(nsmp)-can(nsmp-1)

dy2(1)=dy(2)-dy(1)
do i=2,nsmp-1
  dy2(i)=dy(i+1)-dy(i-1)
enddo
dy2(nsmp)=dy(nsmp)-dy(nsmp-1)

! write gmt files
open(2,file='dy.xy',action='write')
open(3,file='dy2.xy',action='write')
open(7,file='can0.xy',action='write')
open(8,file='Pa.xy',action='write')
open(9,file='hyd.xy',action='write')
do i=1,nsmp
  write(2,'(f10.3,e12.3)') time(i),dy(i)
  write(3,'(f10.3,e12.3)') time(i),dy2(i)
  write(7,'(f10.3,i11)') time(i),nint(can0(i))
  write(8,'(f10.3,f10.1)') time(i),pot(i)
  write(9,'(f10.3,f10.3)') time(i),hyd(i)
enddo
close(2)
close(3)
close(7)
close(8)
close(9)

! find jumps, assume approximate relaxation in relax(=160) samples to
! define the minimum length of segment to edit
! jumps may represent hydrophone up/down moves, but also noise bursts
jmp=0
i=2
xlim=0.                 ! noise level estimate from first few samples
write(10,'(a,/,a)') 'Jumps','jmp  kjmp      tjmp        can        dy2'
do while(i<nsmp-relax)
  i=i+1
  if(jmp.eq.0) xlim=xlim+abs(can(i))
  if(abs(can(i))>1.0e7) then            ! if amplitude indicated a jump
```

```
    jmp=jmp+1
    if(jmp>200) stop 'jmp>200:increase max nr of jumps'
    j=max(1,i-4)
    kjmp(jmp)=j
    if(jmp.eq.1) xlim=xlim/(i-2)
    tjmp(jmp)=time(j)
    write(10,'(i3,i6,f10.3,2i11)') jmp,i,time(i),nint(can(i)),nint(dy2(i))
    i=min(nsmp,i+relax)
    do while (abs(can(i))>xlim.or.abs(dy2(i))>1.0e7)
      i=i+1
      if(i.ge.nsmp) exit
    enddo
    kjmp1(jmp)=i
  endif
enddo

write(4,'(/,i3,a,e12.3)') jmp,' jumps exceeding ',xlim
write(4,'(a)') ' i       t    k1    k2'
do i=1,jmp
  write(4,'(i2,f8.3,2i6)') i,tjmp(i),kjmp(i),kjmp1(i)
enddo

! Now write the segments that may need editing to file out.editcsv
write(4,'(/,a)') 'Edit segments below where needed. # identifies outliers'
write(4,'(a)') 'or clips. A dot signals a probably OK value (scaled by 1e7)'
write(4,'(a)') 'Replace # or . by R to require rejection & interpolation'
write(4,'(a)') 'Replace # or . by S to split the data at this point so that'
write(4,'(a,/)') 'a segment can be rejected. Otherwise leave # or . intact'

do j=1,jmp
  ylow=1.0e20
  yhigh=-1.0e20
  do i=kjmp(j),kjmp1(j)
    ylow=min(ylow,can(i))
    yhigh=max(yhigh,can(i))
  enddo
  write(4,'(a,i3,a,2e12.3,a)') 'jump ',j,'  ylow,yhigh=',ylow*1.0e-7,  &
    yhigh*1.0e-7,'(x 1.0e7)'
  diff=99./(yhigh-ylow)
  k0=(0-ylow)*diff+1
  do i=kjmp(j),kjmp1(j)
    plot=' '
    if(mod(i,10).eq.0) plot='-'
    k=(can(i)-ylow)*diff+1
    plot(k)='*'
    if(k0.ne.k.and.k0>1.and.k0<100) plot(k0)=':'
    flag='.'
    if(abs(dy2(i))>1.0e7) flag='#'
```

```
    if(abs(can(i))>1.34e8) flag='#'     ! clipping
    q='|'
    if(mod(i,5).eq.0) q='+'
    ctime=''
    if(abs(time(i)-nint(10*time(i))*0.1)<halfdt) write(ctime,'(f5.1)') &
      time(i)
    write(4,'(i5,f8.3,1x,a5,2x,a1,a2,101a1)') i,can(i)*1.0e-7,ctime, &
      flag,q,(plot(k),k=1,100),q
  enddo
enddo

! close out.editcsv and call vi to edit it
close(4)

call system('vi out.editcsv')

! reopen the edited file and read it
open(4,file='out.editcsv',action='read')

! skip the headers
line=''
do while(line(1:4).ne.'jump')
  read(4,'(a124)',iostat=ios) line
enddo

! now loop over segments
ib=1
kbreak(1)=1
do
  if(is_iostat_end(ios).or.len_trim(line).eq.0) exit
  if(line(1:5).ne.'break') read(line,*,iostat=ios) ctime,jmp
  if(ios.ne.0) then
    write(6,*) 'ios error  in line',ios
    write(6,*) trim(line)
    stop
  endif

  j=0           ! counts CAN readings in this segment
  do
    read(4,'(a124)',iostat=ios) line
    if(line(1:4).eq.'jump' .or. is_iostat_end(ios)) exit
    if(len_trim(line).eq.0) exit
    read(line,'(i5,16x,a1)',iostat=ios) i,flag
    if(i.le.0.or.ios.ne.0) then
      write(6,*) 'i or ios error ',i,ios
      write(6,*) trim(line)
      stop
    endif
```

```
  if(j.eq.0) i1=i             ! start of segment to interpolate
  if(flag.eq.'.' .or. flag.eq.'#') then       ! accept
    j=j+1
    y(j)=can(i)
    ty(j)=time(i)
  else if(flag.eq.'R') then   ! mark datum for editing
    can(i)=1.1e30
  else if(flag.eq.'S') then
    line(1:5)='break'
    exit
  endif
enddo
if(line(1:5).eq.'break') then
  ib=ib+1
  if(ib>9) stop 'Too many splits. Increase dimension of kbreak'
  kbreak(ib)=i
  write(10,'(a,i7)') 'Break at sample: ',i
endif
n=j           ! nr of samples of segment
i2=i          ! end of segment to interpolate

! interpolate y by fitting harmonics < fmax Hz
tlen=(i2-i1)*dt
if(tlen.le.0.) then
  print *,'Failure in harmonic interpolation between i1,i2=',i1,i2
  print *,'j,y(j),ty(j)=',j,y(j),ty(j)
  print *,'Last line read: ',trim(line)
endif
dw=twopi/tlen
nf=fmax*tlen+1
m=2*nf+1              ! number of coefficients
write(10,'(a,f8.0,2i8)') 'tlen, nf, m=',tlen,nf,m
if(m>257) stop 'Segment too long. Split or increase dimension of array x'

! construct matrix
allocate(a(n,m))
do i=1,n
  t=ty(i)
  w=0.
  j=1
  a(i,1)=1.0          ! zero frequency
  do j=2,m-1,2
    w=w+dw
    a(i,j)=cos(w*t)
    a(i,j+1)=sin(w*t)
  enddo
enddo
! solve A*x=y for coefficients x
```

```
  ! if no pick-up, this is an orthogonal systemand and converges in 1
  itmax=100
  iout=10
  write(10,*) 'LSQR:'
  call flsqr(a,n,n,m,x,y,itmax,iout,r)
  deallocate(a)

  ! identify outliers that need editing
  write(10,'(a,i5,a,i5)') 'Editing from',i1,' to ',i2
  write(10,'(a)') '     i         t         can0          can'
  do i=i1,i2
    if(can(i)<1.0e30) cycle
    som=x(1)
    j=1
    w=0.
    t=time(i)
    do k=2,m,2
      w=w+dw
      som=som+x(k)*cos(w*t)+x(k+1)*sin(w*t)
    enddo
    can(i)=som          ! replace can(i) by sum of cos,sin
    write(10,'(i6,f10.3,2f13.0)') i,t,can0(i),can(i)
  enddo
enddo

! write edited file to can.xy
open(7,file='can.xy',action='write')
do i=1,nsmp
  write(7,'(f10.3,i11)') time(i),nint(can(i))
enddo
close(7)

! Now write the (edited) can(s) in format suitable for getpz2
ib=ib+1
kbreak(ib)=nsmp
k=index(fname,'.csv')
if(k>0) then
  fname(k:k+3)='.can'
endif
k=len_trim(fname)
if(ib.eq.2) then
  write(10,*) 'No breaks detected'
else
  write(10,'(a,/,a)') 'Split signal','ib   j-1     j        n'
endif

! Write input for getpz2. Split if there are breaks (ib>2).
do j=2,ib
```

```
  n=kbreak(j)-kbreak(j-1)+1
  if(ib.eq.2) then
    open(2,file=trim(fname),action='write')
    write(2,'(2a)') 'Edited hydrophone data from ',trim(fname)
  else
    write(fname(k+1:k+1),'(i1)') j-1
    open(2,file=trim(fname),action='write')
    write(2,'(3a,i2)') 'Edited hydrophone data from ',trim(fname),  &
      ', split',j-1
    write(10,'(i2,3i6)') j-1,kbreak(j-1),kbreak(j),n
  endif
  write(2,*) n
  write(2,*) dt
  do i=kbreak(j-1),kbreak(j)
    write(2,'(f10.1,i11)') pot(i),nint(can(i))
  enddo
  close(2)
enddo

write(6,'(a)') 'Succesful end of editcsv.'
write(6,'(a)') 'Plot interpolations with gmtcan, check results'
write(6,'(a)') '  in file diagnostics.editcsh.'
if(ib.eq.2) then
  write(6,'(2a)') 'input getpz2: ',trim(fname)
else
  write(6,'(3a)') 'input getpz2: ',trim(fname),' etc.'
endif


end

! SUBROUTINES

subroutine flsqr(a,nm,n,m,x,rhs,itmax,iout,r)

!   subroutine to solve the linear tomographic problem Ax=u using the
!   lsqr algorithm. As lsqr but for full matrix A, and rhs is preserved.

!   reference: C.C.Paige and M.A.Saunders, ACM Trans.Math.Softw. 8,
!       43-71, 1982
!   and ACM Trans.Math.Softw. 8, 195-209, 1982.
!   See also A.v.d.Sluis and H.v.d.Vorst in: G. Nolet (ed.), Seismic
!       Tomography, Reidel, 1987.

!   Input:
!   a(nm,m) is the matrix
!   n is the number of data (rows),
!   m the number of unknowns (colunms),
```

```
!   rhs contains the data,
!   itmax is the number of iterations allowed
!   iout is the output channel (set iout=6 for screen output, 0 for none)

!   Output: x is the solution

!   If you set itmax very large, the program will quit as soon as the
!   misfit improves less than 1.0e-8 (relative) between two iterations.

!   Scratch (allocatable arrays): arrays v(m) and w(m)

!   Subroutines used (and included with the source):
!   avpu computes u=u+A*v for given input u,v (overwrites u)
!   atupv computes v=v+A(transpose)*u for given u,v )(overwrites v)
!   normlz normalizes the vector u or v

implicit none

integer, intent(in) :: nm,n,m          ! matrix size, rows, columns
integer, intent(in) :: iout            ! output unit (or 0 for none)
integer, intent(in) :: itmax           ! max nr of iterations
real*4, intent(in) :: a(nm,m)          ! matrix
real*4, intent(in) :: rhs(n)           ! right hand side (data)
real*4, intent(out) :: x(m)            ! solution
real*4, intent(out) :: r               ! relative misfit

real*4, allocatable, dimension(:) :: u,v,w
real*4 :: alfa,beta,b1,c,rho,rhobar,phi,phibar,rold,s,t1,t2,teta
integer :: iter

allocate(u(n))
allocate(v(m))
allocate(w(m))
x=0.
v=0.
u=rhs

! initialize
call normlz(n,u,beta)
b1=beta
call atupv(a,nm,n,m,u,v)
call normlz(m,v,alfa)
rhobar=alfa
phibar=beta
w=v
if(iout>0) write(iout,fmt='(a,/,i5,2g12.4,f10.6)')   &
  ' iter        x(1)        beta         r',0,x(1),beta,1.0
rold=1.0e30
```

```
do iter =1,itmax
! repeat for fixed nr of iterations
  u=-alfa*u
! bidiagonalization
  call avpu(a,nm,n,m,u,v)
  call normlz(n,u,beta)
  v=-beta*v
  call atupv(a,nm,n,m,u,v)
  call normlz(m,v,alfa)
  rho=sqrt(rhobar*rhobar+beta*beta) ! modified QR factorization
  c=rhobar/rho
  s=beta/rho
  teta=s*alfa
  rhobar=-c*alfa
  phi=c*phibar
  phibar=s*phibar
  t1=phi/rho
  t2=-teta/rho
  x=t1*w+x
  w=t2*w+v
  r=phibar/b1
  if(iout>0) write(iout,fmt='(i5,2g12.4,f10.6)') iter,x(1),phibar,r
  if(abs(r-rold)<1.0e-8) exit
  rold=r
end do
deallocate(u)
deallocate(v)
deallocate(w)
return
end

subroutine normlz(n,x,s)
dimension x(n)
s=0.
do i=1,n
  s=s+x(i)**2
end do
s=sqrt(s)
ss=1./s
do i=1,n
  x(i)=x(i)*ss
end do
return
end


subroutine atupv(a,mn,m,n,u,v)
```

```
real*4 :: a(mn,n),u(m),v(n)

do j=1,n
  do i=1,m
    v(j)=v(j)+a(i,j)*u(i)
  enddo
enddo
return
end

subroutine avpu(a,mn,m,n,u,v)

real*4 :: a(mn,n),u(m),v(n)

do j=1,n
  do i=1,m
    u(i)=u(i)+a(i,j)*v(j)
  enddo
enddo

return
end
```

A.3.2 getpz2.f90

```
program getpz2

! Compile
! gfortran -o getpz2 getpz2.f90 getpzsubs.f sacio.a

! Given one of more input signals fin (in Pascal) and observed responses
! fobs (in Counts), find the Mermaid response in terms of poles and zeroes

! Use editcsv.f90 to create de-noised input file(s) from the Excel files
! provided by OSEAN

! Input:
! file in.getpz2 with input file names like test1.can
! from screen: low pass frequency and commands to continue or stop

! format of input files is as written by programs editcsh and readcsh
! array fin has the input (in Pascal), fobs the output (in counts)
! of the tests.

! The program can also be run without using the sacio.a library
! if you comment all calls to routine wsac. You can plot the
! *.xy files with GMT or any other plotting program.
```

```
! From a digitized input signal fin(t), and a digitized output
! signal fobs(t), find optimal poles, zeroes and the amplification
! of an instrument. The only other input needed is an estimate of
! the high-pass frequency, which is used to start the iterations,
! a specification of the low-pass that defines the frequency band
! over which we fit in- and output, and a maximum allowable misfit
! (in percent) to stop iterations.

! For a block-type signal, you can create a proper input file
! using makeio.f90

! fin and fobs must start at the same time and have equal sampling
! interval dt. To alleviate edge effects it is good to start and
! end fin and fobs with zero segments (no input, response quieted down).

! We assume that the response has at least one zero equal to 0 (i.e.
! there is zero amplification of a DC signal).

! The structure of the data files is:
! line0: comment header
! line1: n (number of data input)
! line2: dt (in seconds)
! lines3: n data (free format, units Pascal)

! Output:
! out.getpz2 with the result
! sacpz with a file to be used in sac (trans from polezero s sacpz to ...)
! inpz intermediate pz results for SAC
! diagnostics.getpz allows you to follow the optimization
! inputnn.sac SAC file of input nn (Pa, tapered but not scaled)
! outptnn.sac SAC file of the observed response to nn (energy scaled to 1)
! fout0.sac response predicted with the starting set of poles and zeroes
! foutk_nn.sac response to nn predicted after (outer) iteration rkn

implicit none

! we allow for up to NPAR (complex) poles and zeroes
integer, parameter :: ND=32768, M2=15, NPAR=40, NREC=100         ! ND=2**M2
integer, parameter :: NY=2*NPAR

character*80 :: fname(NREC),fnout(NREC),pzfile,header,line80,sacfile
character*1 :: h
character*8 :: pz,kstnm
character :: date*8, time*10
complex(kind=4) :: fcmplx(ND,NREC) ! FT of the input in Pa, with energy 1
complex(kind=4) :: ft(ND)          ! temporary fourier trf
complex(kind=4) :: pdflt(5)        ! default set of poles for startup
```

```
complex(kind=4) :: pole(NPAR)      ! response poles
complex(kind=4) :: r               ! response
complex(kind=4) :: zdflt(4)        ! default set of zeroes for startup‘
complex(kind=4) :: zero(NPAR)      ! response zeroes
integer :: i,ios,i1,i2,j,k,m,n,nerr
integer :: ifl1,ifl2            ! indices of flat spectrum limits
integer :: iter                 ! Number of iterations needed by Powel
integer :: itry                 ! counts trials with increasing nr of poles
integer :: jbug                 ! flags if bug output is wanted
integer :: jday                 ! Julian day (for sac files)
integer :: kontinue             ! 1 if continuing with more poles/zeroes
integer :: kplot                ! flags if plot files are made
integer :: kpow                 ! power of 2 for all FT's
integer :: n2                   ! length of fft (n2=2**kpow)
integer :: ndim                 ! nr of parameters druing trial itry
integer :: nfile                ! number of files with response data
integer :: nmax                 ! length of largest fobs()
integer :: np,nq,nz             ! length of pole(),zero(),q()
integer :: npts(NREC)           ! length of each fin,fobs
integer :: nz0                  ! no of zero() being kept 0.
integer :: t0(6)                ! origin time (for SAC files)
real*4 :: alfa                  ! inverse of relaxation time (exp -alfa*t)
real*4 :: ampDC                 ! small amplitude considered close to DC
real*4 :: ampmax                ! maximum fobs()
real*4 :: ampmin                ! minimum fobs()
real*4 :: ampobs                ! scaling for fobs()
real*4 :: ampred                ! scaling for fout()
real*4 :: df                    ! FFT bin in Herz
real*4 :: dt                    ! sampling interval in seconds (Tektronics)
real*4 :: dtcan                 ! actual interval for the CAN (see below)
real*4 :: dum,x,yy,z
real*4 :: dw                    ! step in circle frequency w
real*4 :: f2                    ! Low pass frequency during optimization
real*4 :: fin(ND,NREC)          ! input in Pa, after scaling/tapering
real*4 :: finit                 ! initial misfit (value of func)
real*4 :: fl1,fl2               ! limits of the ideally flat response
real*4 :: fnyquist              ! Nyquist frequency in Hertz
real*4 :: fobs(ND,NREC)         ! output (scaled, tapered
real*4 :: fret                  ! value of func returned by Powel
real*4 :: ftol                  ! tolerance for convergence of Powel
real*4 :: func                  ! function func computes the misfit
real*4 :: pi=3.14159265, twopi=6.2831853
real*4 :: q(NY)                 ! real mapping of complex poles, zero
real*4 :: t                     ! time
real*4 :: thalf                 ! time between 0 and relaxation to max/2
real*4 :: tshift                ! time until first jump
real*4 :: w                     ! circle frequency
real*4 :: xi(NY,NY)             ! search direction in Powel's algorithm
```

```
real*4 :: Xre,Xim               ! real, imaginary component of pole, zero
real*4 :: y(NY)                 ! mapping of poles & zeroes to parameters y

data pdflt/(0.49343E-01,-0.53623E-02), (0.49343E-01, 0.53623E-02),  &
  (-0.72407, 0.0000), (-0.58644E-01,-0.15366E-03),  &
  (-0.58644E-01, 0.15366E-03)/
data zdflt/(0.55110E-01, 0.45550E-01), (0.55110E-01,-0.45550E-01), &
  (-0.23670E-01, 0.41947E-01), (-0.23670E-01,-0.41947E-01)/

common ampDC,ampred,dt,dw,fcmplx,fl1,fl2,fin,fobs,i2,ifl1,ifl2,jbug,&
  kplot,kpow,n2,ndim,nfile,np,npts,nq,nz,nz0,pole,q,zero,dtcan

! initialize
fl1=4.0                         ! flat spectrum start
fl2=8.0                         ! flat spectrum test limit
ftol=0.0001                     ! tolerance for convergence
open(3,file='out.getpz2',action='write')
open(4,file='diagnostic.getpz2',action='write')

! We assume the two columns in the CSV files are perfectly synchronic.
! However, if doubts exist, run getdeltat.f90 on one of the files to
! find the sampling interval of the CAN (dtcan) & uncomment these lines:
!print *,'We assume you have found the correct sampling interval for'
!print *,'the Mermaid card (CAN) using program getdeltat. If you'
!print *,'answer 0, it will assume CAN and Tektronics have the same dt.'
!print *,'Give the true sampling interval for the CAN (in s):'
!read(5,*) dtcan

! Read input signals
open(15,file='in.getpz2',action='read')
nfile=0
nmax=0
ampobs=0.
ampmax=-1.0e30
ampmin=+1.0e30
fobs=0.                 ! ensures zero padding whatever true length of fobs
write(3,'(a)') ' n    obs      dt tshift     ampmax     ampmin     ampobs'
do
  nfile=nfile+1
  if(nfile>NREC) stop 'Too many files, increase array sizes'
  read(15,*,iostat=ios) fname(nfile)
  if(is_iostat_end(ios) .or. fname(nfile)(1:4).eq.'stop') then
    nfile=nfile-1
    exit
  endif
  print *,'Opening ',trim(fname(nfile))

  open(1,file=fname(nfile),iostat=ios,action='read')
```

```
  if(ios.ne.0) stop 'Cannot open input file'
  read(1,'(a)') header
  read(1,*) npts(nfile)
  if(2*n>ND) stop 'Signal too long'
  read(1,*) x
  if(nfile.eq.1) then
    dt=x
  else
    if(abs(dt-x)>0.0001) stop 'Sampling intervals not equal'
  endif
  if(abs(dtcan).le.0.) dtcan=dt
  do i=1,npts(nfile)
    read(1,*,iostat=ios) fin(i,nfile),fobs(i,nfile)
    if(ios.ne.0) stop 'Error reading input file'
  enddo
  close(1)

  ! If necessary, we resample fobs do match the dt of the Tektronics
  if(abs(dt-dtcan)>0.) then
    kpow=M2
    do while(2**kpow .ge. npts(nfile))
      kpow=kpow-1
    enddo
    kpow=kpow+1
    n2=2**kpow
    call resample(fobs,n2,kpow,dtcan,dt)
  endif

  ! remove start of signal until first pulse
  m=npts(nfile)
  call shift0(fin(1,nfile),fobs(1,nfile),npts(nfile))
  tshift=(npts(nfile)-m)*dt
  m=npts(nfile)
  do i=1,m
    ampobs=ampobs+fobs(i,nfile)**2
  enddo
  write(3,'(i3,i6,2f8.3,2i11,e11.2,1x,a)') nfile, &
    m,dt,tshift,nint(maxval(fobs(1:m,nfile))), &
    nint(minval(fobs(1:m,nfile))),sqrt(ampobs),trim(fname(nfile))
  if(npts(nfile)<400) then
    print *,'Warning: file ',trim(fname(nfile)),' is too short and ignored'
    nfile=nfile-1
    cycle
  endif
  nmax=max(nmax,npts(nfile))          ! find largest file size
enddo
close(15)
```

```
! scale total fobs energy to 1. This scaling is needed to have some
! reference for mthe misfit level
ampobs=sqrt(ampobs)
write(3,'(a,e12.3)') 'Observed response fobs is scaled by ampobs= ',ampobs
fobs=fobs/ampobs
ampmax=maxval(fobs)
ampmin=minval(fobs)
ampDC=0.1*max(abs(ampmin),ampmax)
write(3,'(a,2g13.5)') 'After scaling ampmin,max are:',ampmin,ampmax

! find nearest power of 2
kpow=M2
do while(2**kpow .ge. nmax)
  kpow=kpow-1
enddo
kpow=kpow+1
n2=2**kpow
df=1.0/(n2*dt)            ! frequency interval in Hz
dw=twopi*df

fnyquist=0.5/dt           ! Nyquist frequency
write(3,'(a,i10,a)') 'All signals are zero-padded to ',n2,' samples'
write(3,'(a,f10.6,a)') 'FFT frequency step is ',df,' Hz'
write(3,'(a,f10.3)') 'Nyquist frequency (Hz): ',fnyquist
flush(3)

! write SAC files of unfiltered signals (fin scaled to physical units)
call date_and_time(date,time)
read(date,'(i4,3i2)') (t0(i),i=1,3)
read(time,'(3i2)') (t0(i),i=4,6)
call newhdr
call setnhv('npts',n2,nerr)
call setfhv('b',0.,nerr)
call setfhv('e',(n2-1)*dt,nerr)
call setfhv('delta',dt,nerr)
call setnhv('nzyear',t0(1),nerr)
call julian(t0(1),t0(2),t0(3),jday)
call setnhv('nzjday',jday,nerr)
call setnhv('nzhour',t0(4),nerr)
call setnhv('nzmin',t0(5),nerr)
call setnhv('nzsec',t0(6),nerr)
call setnhv('nzmsec',0,nerr)
call setfhv('o',0.,nerr)
call setihv('iztype','IB',nerr)
! Plot files for fin and fobs (input and response after scaling)
! also output of input and observed signal to GMT plot format
do n=1,nfile
  write(kstnm,'(a5,i2.2)') 'test_',n
```

```
  call setkhv('kstnm',kstnm,nerr)
  kstnm(1:5)='input'
  call setkhv('kstnm',kstnm,nerr)
  call wsac0(trim(kstnm)//'.sac',dum,fin(1,n),nerr)
  open(2,file=trim(kstnm)//'.xy')
  do i=1,npts(n)
    write(2,*) (i-1)*dt,fin(i,n)
  enddo
  close(2)
  kstnm(1:5)='outpt'
  call setkhv('kstnm',kstnm,nerr)
  call wsac0(trim(kstnm)//'.sac',dum,fobs(1,n),nerr)
  open(2,file=trim(kstnm)//'.xy')
  do i=1,npts(n)
    write(2,*) (i-1)*dt,fobs(i,n)
  enddo
  close(2)
enddo

! we always assume the spectrum is flat at high frequency (but see the
! commented section below this segment)
! set limits of deviation from flat spectrum penalty
fl1=4.0                         ! lowest frequency of flat part
fl2=8.0                         ! and highest frequency
! Uncomment this if you want more flexibility on fl1,fl2:
! print *,'You may impose flat amplitude between fl1 and fl2 Hz'
! print *,'Give fl1 and fl2 (or 0,0):'
! read(5,*) fl1, fl2
if(fl2>0.) then
  ifl1=fl1/df+1
  ifl2=fl2/df+1
else
  ifl1=n2
  ifl2=0
endif

write(3,'(a,2f8.1)') 'We want a flat amplitude between (Hz):',fl1,fl2

fl1=twopi*fl1
fl2=twopi*fl2

print *,'There are three options for the starting set of poles and'
print *,'zeroes: when nothing else is known, give the time (in s) at which'
print *,'the response has relaxed to half its maximum (measured from t=0)'
print *,'If this is a Mermaid from a known series, you can answer'
print *,'<default> (without the <>) and it will start from the standard'
print *,'response. If you have a result from an earlier iteration saved'
print *,'in a file, give that file name. Getpz2 saves the latest result'
```

```
print *,'in file <inpz>.'

print *,'Give time to half relaxation, or file name, or <default>:'
read(5,'(a)') pzfile
read(pzfile,*,iostat=ios) thalf

if(ios.eq.0) then
  alfa=0.69/thalf
  nz0=1
  nz=2
  np=3
  ! start with one set of complex conjugates.
  pole(1)=cmplx(-alfa,0.)
  pole(2)=cmplx(0.05,0.05)
  pole(3)=cmplx(0.05,-0.05)
  zero(1)=cmplx(0.05,0.05)
  zero(2)=cmplx(0.05,-0.05)
  write(3,'(a,f8.2)') 'Initial set derived from thalf=',thalf
  write(4,'(a,f10.4)') 'alfa=',alfa
else if(pzfile.eq.'default') then
  nz0=1
  nz=4
  np=5
  pole(1:5)=pdflt
  zero(1:4)=zdflt
  write(3,'(a)') 'Initial set is the default set.'
else
  write(3,*) 'Reading starting values from pz file: ',trim(pzfile)
  open(2,file=pzfile,action='read')
  ! format should as in SAC: first POLES, then ZEROS, then CONSTANT
  ! set nz equal to *all* zeros and include the ones that are fixed (0,0).
  ! (all zeroes that have starting value (0,0) will be kept fixed).
  ! avoid any comments in the pzfile
  read(2,*,iostat=ios) pz,np
  if(ios.ne.0) stop 'Error in first line of pole-zero input'
  do i=1,np
    read(2,*,iostat=ios) Xre,Xim
    pole(i)=cmplx(Xre,Xim)
    if(ios.ne.0) stop 'Error in reading poles'
  enddo
  read(2,*,iostat=ios) pz,nz
  if(ios.ne.0) stop 'Error in reading ZEROS line'
  do i=1,nz
    read(2,*,iostat=ios) Xre,Xim
    zero(i)=cmplx(Xre,Xim)
    if(ios.ne.0) stop 'Error in reading zeroes'
  enddo
  ! remove any zeroes equal to (0,0) from the optimizable set
```

```
  k=nz
  nz0=0
  do i=1,k
    if(abs(zero(i)).le.0.) then         ! remove and move up the rest
      do j=i+1,nz
        zero(j-1)=zero(j)
      enddo
      nz=nz-1
      nz0=nz0+1
    endif
  enddo

  if(nz.ne.k) print *,nz0,' starting zeros are (0,0) & will be kept (0,0)'
  if(nz0.eq.0) print *,'WARNING! DC response is not guaranteed zero'
  close(2)              ! we ignore the CONSTANT
endif

write(3,'(/,a,i2,a)') 'Starting with ',np,' poles:'
write(3,'(i3,2f10.4)') (i,pole(i),i=1,np)
write(3,'(a,i2,a)') 'and ',nz,' zeros:'
write(3,'(i3,2f10.4)') (i,zero(i),i=1,nz)
write(3,'(i3,a)') nz0,' are being kept fixed to (0,0)'
if(np.ne.nz+nz0) then
  print *,'WARNING! np is not equal to nz+nz0, the response at high'
  print *,'frequency is therefore not flat'
  write(3,'(a)') 'WARNING! np is not equal to nz+nz0, the response at high'
  write(3,'(a)') 'frequency is therefore not flat'
endif
flush(3)

! Optional low pass filtering to exclude spurious noise at high frequency
print *,'Give low pass frequency, or 0 for no filter (e.g. 4 Hz):'
read(5,*) f2            ! f2=lowpass
i1=1
i2=n2
if(f2>0.) then
  if(f2>fnyquist) stop 'lowpass frequency exceeds Nyquist frequency'
  write(3,'(a,2f10.3)') 'Low pass frequency: ',f2
  ! taper 10% to f2
  i1=0.9*nint(f2/df)-1
  if(i1<1) stop 'lowpass frequency too low'
  i2=nint(f2/df)
  yy=i2-i1+1
else
  write(3,*) 'No low pass applied'
endif
flush(3)
```

```
! lowpass observed and input signals
if(f2>0.) then
  do n=1,nfile
    ! lowpass fobs, using taper, use ft for fobs spectrum
    ft(1:n2)=fobs(1:n2,n)
    call clogp(kpow,ft,-1.0,dt)      ! compute FT
    do i=i1,i2
      x=float(i-i1)/yy
      z=0.5*(1.0+cos(pi*x))
      ft(i)=z*ft(i)
    enddo
    ft(i2:n2)=0.            ! negative spectrum is added in ftinv
    call ftinv(ft,kpow,dt,fobs(1,n))      ! and back to time domain
    ! DEBUG
    write(sacfile,'(a,i1,a)') 'debugfobs',n,'.sac'
    call wsac0(sacfile,dum,fobs(1,n),nerr)

    ! do the same with fin, and from now on preserve fcmplx for each fin
    fcmplx(1:n2,n)=fin(1:n2,n)
    call clogp(kpow,fcmplx(1,n),-1.0,dt)      ! compute FT
    do i=i1,i2
      x=float(i-i1)/yy
      z=0.5*(1.0+cos(pi*x))
      fcmplx(i,n)=z*fcmplx(i,n)
    enddo
    fcmplx(i2:n2,n)=0.            ! negative spectrum is added in ftinv
  enddo

else            ! if no filtering get FT of fin
  do n=1,nfile
    fcmplx(1:n2,n)=fin(1:n2,n)
    call clogp(kpow,fcmplx(1,n),-1.0,dt)      ! compute FT of fin
  enddo
endif

! from here on fin and fobs contain the scaled (filtered) in- and output
! and fcmplx contains the FFT of fin

! Now start nonlinear search for poles and zeroes, starting with
! np poles and nz+nz0 zeroes. Note that the spectrum can only be flat
! at high frequency if np=nz+nz0
itry=0
kplot=0                 ! causes func to write fout0.sac for initial fit
kstnm(1:5)='iter0'
call setkhv('kstnm',kstnm,nerr)
call pz2y(pole,np,zero,nz,y,ndim,q,nq)
finit=func(y)
kstnm(1:5)='init_'
```

```
call setkhv('kstnm',kstnm,nerr)
kplot=-1                ! suppress plots while optimizing
write(6,'(a,g12.3)') 'Initial misfit:',finit
write(6,'(a,f6.3)') 'Relative misfit: ',1.0

open(7,file='inpz')     ! open file for resulting poles/zeroes

do

  itry=itry+1

  ! set up starting parameter vector (note: real input stays real)
  call pz2y(pole,np,zero,nz,y,ndim,q,nq)

  ! at this point y contains poles, zeroes (except zero(0)=0)

  ! set up the directions for routine powell which will search
  ! iteratively in these directions for a better y vector
  ! (starting with latest additions as first directions)
  xi=0.
  do i=1,ndim
    xi(i,i)=0.02
  enddo

  ! optimize poles and zeroes  (in y) using powell
  print *,'Now optimizing...have patience'
  print *,'Calling powell with ndim=',ndim
  write(4,'(6x,2a)') 'pflat       ysom       asom      worst  n        ', &
    'func  y'
  kplot=-1
  call powell(y,xi,ndim,NY,ftol,iter,fret)
  flush(4)

  kplot=min(9,itry)   ! SAC file for this iteration
  t=func(y)           ! compute penalty and plot results
  write(6,'(a,2g10.3)') 'Final misfit:',fret
  write(6,'(a,f6.3)') 'Relative misfit: ',fret/finit
  write(3,'(a,i4,e11.3,f9.3,e12.3)') 'Results from iter: ',itry,fret,  &
    fret/finit,ampred

  ! map back from y to poles/zeroes
  call y2pz(pole,np,zero,nz,y,ndim,q,nq)

  ! write inpz file with the new solution
  write(7,'(/,a,i3)') '------------- iteration: ',itry
  write(7,'(a,i5)') 'POLES ',np
  do i=1,np
    write(7,'(2g14.5)') pole(i)
```

```
  enddo
  write(7,'(a,i5)') 'ZEROS ',nz+nz0
  do i=1,nz
    write(7,'(2g14.5)') zero(i)
  enddo
  do i=1,nz0
    write(7,'(a)') '   0.   0.'
  enddo

  ! The unscaled fobs is fobs(scaled)*ampobs, the unscaled fout is
  ! fout(scaled)/ampred. The amplificationis thus fobs/fout=ampobs*ampred
  write(7,'(a,g14.5)') 'CONSTANT',ampobs*ampred

  ! tell me what you have
  write(3,'(/,a,i3,a)') 'Results of iteration ',itry,':'
  write(3,'(a,e11.3,a,f9.3)') 'Absolute misfit:',fret,', Relative:',  &
    fret/finit
  write(3,'(a,g14.5,a,e11.3)') 'A0=',ampobs*ampred,', scaling for fout:', &
    ampred
  write(3,*) 'Poles:'
  do i=1,np
    write(3,'(i3,2g14.5)') i,real(pole(i)),aimag(pole(i))
  enddo
  write(3,*) 'Zeroes:'
  if(nz>0) then
    do i=1,nz
      write(3,'(i3,2g14.5)') i,zero(i)
    enddo
  endif
  do i=1,nz0
    write(3,'(i3,2g14.5)') i+nz,0.,0.
  enddo

  if(np+nz<NPAR-3) then
    print *,'Add more poles/zeroes (1) or stop (0)?'
    read(5,*)  kontinue
    if(kontinue.le.0) exit
  else
    exit
  endif

!  np=np+1
!  nz=nz+1
!  pole(np)=cmplx(0.05,0.05)
!  zero(np)=cmplx(0.05,0.05)
!  np=np+1
!  nz=nz+1
!  pole(np)=cmplx(0.05,-0.05)
```

```
!  zero(nz)=cmplx(0.05,-0.05)

  ! add 2 poles and 2 zeros at the start of the parameter list
  do i=np,1,-1
    pole(i+2)=pole(i)
  enddo
  do i=nz,1,-1
    zero(i+2)=zero(i)
  enddo
  pole(1)=cmplx(0.05,0.05)
  zero(1)=cmplx(0.05,0.05)
  pole(2)=cmplx(0.05,-0.05)
  zero(2)=cmplx(0.05,-0.05)
  np=np+2
  nz=nz+2

enddo

end

subroutine resp(omega,ip,p,iz,z,nz0,r)

! gives response at complex frequency s=i*omega=i*2*pi*f
!       but ignoring amplification factor
! ip poles stored in p, iz zeroes in z
! output r is response
!       for complex conjugate pairs, only one is given in p,z
!       nz0 extra zeroes z=0 are added

implicit none
integer, parameter :: NPAR=40
complex(kind=4), intent(in) :: p(NPAR),z(NPAR)
real*4, intent(in) :: omega
integer, intent(in) :: ip,iz,nz0
complex(kind=4), intent(out) :: r
real*4 :: eps=1.0e-20
complex(kind=4) :: s
integer :: j

s=cmplx(0.,omega)               ! in SEED, s = +i*omega

r = cmplx(1.,0.)                ! startup of response r

! multiply the factors in the nominator (zeros)
do j = 1,iz
  r = r*(s-z(j))
enddo
```

```
! multiply by fixed 0's
do j=1,nz0
  r=r*s
enddo

! do the denominator (poles)
do j = 1,ip
  r = r/(s-p(j))
enddo

return
end subroutine resp

! Subroutines pz2y and y2pz are needed because complex poles and
! zeroes exist in conjugate pairs, and only one of each pair needs
! to be optimized, while for real values only the real part of
! the (complex) variable needs to be optimized. The optimization
! therefore deals not with pole() and zero(), but with y().

subroutine pz2y(p,np,z,nz,y,ndim,q,nq)

! maps magnification poles and zeroes into y via intermediate
! q (q remembers 0 imaginary parts of real poles of zeroes)
! q needs to be saved for the inverse mapping with y2pz

implicit none
integer, parameter :: NPAR=40
complex(kind=4), intent(in) :: p(NPAR), z(NPAR)
integer, intent(in) :: np,nz
real*4, intent(out) :: y(2*NPAR),q(2*NPAR)
integer, intent(out) :: ndim,nq
integer :: i,j


! first map p and z into q, ignoring conjugates
j=0
i=0
do while(i<np)       ! loop over poles
  i=i+1
  j=j+2
  q(j-1)=real(p(i))
  q(j)=aimag(p(i))
  if(abs(q(j))>0.) i=i+1        ! conjugate not needed in q
enddo

i=0
do while(i<nz)       ! loop over zeroes
  i=i+1
```

```
  j=j+2
  q(j-1)=real(z(i))
  q(j)=aimag(z(i))
  if(abs(q(j))>0.) i=i+1        ! conjugate not needed in q
enddo
nq=j
if(2*(nq/2).ne.nq) stop 'nq is odd'     ! DEBUG

! now map q into y, ignoring zero imaginary parts
j=0
do i=1,nq
  if(abs(q(i))>0.) then
    j=j+1
    y(j)=q(i)
  endif
enddo
ndim=j

return
end subroutine pz2y


subroutine y2pz(p,np,z,nz,y,ndim,q,nq)

! maps y back into q, poles and zeroes
! existing real/imaginary nature of pole/zero is used to determine whether
! they have a complex conjugate, so arrays pole/zero are both in- and output
! q is needed on input and may be changed on output.

implicit none
integer, parameter :: NPAR=40
real*4, intent(inout) :: q(2*NPAR)
complex(kind=4), intent(out) :: p(NPAR), z(NPAR)
integer, intent(in) :: np,nz
real*4, intent(in) :: y(2*NPAR)
integer, intent(in) :: ndim
integer, intent(out) :: nq
integer :: i,j,k

! map back from y to poles/zeroes
j=0
do i=1,nq,2
  j=j+1
  q(i)=y(j)
  if(abs(q(i+1))>0.) then       ! change nonzero q as well (=imaginary)
    j=j+1
    q(i+1)=y(j)
  endif
```

```
enddo

j=0
k=0
do i=1,nq,2
  if(j<np) then
    j=j+1
    p(j)=cmplx(q(i),q(i+1))
    if(abs(q(i+1))>0.) then
      j=j+1
      p(j)=cmplx(q(i),-q(i+1))
    endif
  else
    k=k+1
    z(k)=cmplx(q(i),q(i+1))
    if(abs(q(i+1))>0.) then
      k=k+1
      z(k)=cmplx(q(i),-q(i+1))
    endif
  endif
enddo

return
end subroutine y2pz

real function func(y)

! given parameter vector y with np poles and nz zeroes, compute the
! instrument response and set func equal to quadratic misfit with data d()

! data etc are passed on via common

! a full history of iterations is written to powell.xy

implicit none

integer, parameter :: ND=32768, M2=15, NPAR=40, NREC=100   ! ND=2**M2
real*4, intent(in) :: y(2*NPAR)
real*4 :: dt,dtcan,dw,twopi=6.283185,w,ampred,t,q(2*NPAR)
real*4 :: som,ysom,asom,aver
real*8 :: somx,somxy
complex(kind=4) :: pole(NPAR),zero(NPAR),fcmplx(ND,NREC),s(ND)
complex(kind=4) :: cc,r
real*4 :: dum,eps,fobs(ND,NREC),fin(ND,NREC),fout(ND,NREC),fl1,fl2,pflat
integer :: n,np,nz,nz0,i,i1,i2,j,kpow,n2,iter,itry,ndim,nq,nerr,kplot
integer ::  jbug,ifl1,ifl2,npts(NREC),nfile,nworst
real*4 :: ampDC,worst
```

```
common ampDC,ampred,dt,dw,fcmplx,fl1,fl2,fin,fobs,i2,ifl1,ifl2,jbug,&
  kplot,kpow,n2,ndim,nfile,np,npts,nq,nz,nz0,pole,q,zero,dtcan

eps=0.005              ! damping term for poles and zeroes (in y)
cc=cmplx(0.,1.)

! map back from y to poles/zeros since resp works with them
call y2pz(pole,np,zero,nz,y,ndim,q,nq)

! compute the penalty due to rms difference from flat response
pflat=0.
do i=ifl1,ifl2
  w=(i-1)*dw
  call resp(w,np,pole,nz,zero,nz0,r)
  pflat=pflat+(abs(r)-1.0)**2
enddo
pflat=0.1*sqrt(pflat/(ifl2-ifl1+1))

! create spectrum of the response to each fin, store in s() and compute the
! penalty to deviations from each fobs
! note that fcmplx has the spectrum of fin
somx=0.d0
somxy=0.d0
do n=1,nfile
  s(1:n2)=fcmplx(1:n2,n)
  w=0.
  do i=1,n2/2-1
    call resp(w,np,pole,nz,zero,nz0,r)
    s(i)=r*s(i)
    w=w+dw
  enddo
  s(n2/2)=0.
  call ftinv(s,kpow,dt,fout(1,n))

  ! find optimal amplification fobs/fout (y/x) with least squares:
  do i=1,npts(n)
    if(abs(fobs(i,n))<ampDC) cycle      ! do not include DC offsets
    somx=somx+fout(i,n)**2
    somxy=somxy+fobs(i,n)*fout(i,n)
  enddo
enddo

! we minimize |fobs - ampred*fout|^2 before computing waveform misfit
ampred=somxy/somx
fout=fout*ampred       ! scales energy of fout to 1
if(kplot.eq.0) write(3,'(2a,e12.3,a)') 'Initial response fout is', &
  ' multiplied by ',ampred, ' to minimize misfit'
```

```
! compute misfit between observed and predicted signals
som=0.                  ! misfit for individual signals
asom=0.                 ! total for al n signals
worst=0.
do n=1,nfile
  if(kplot.ge.0) call plts(n,fout(1,n),kplot,npts(n),dt)
  som=0.
  do i=1,npts(n)
    som=som+(fobs(i,n)-fout(i,n))**2
  enddo
  if(som>worst) then
    worst=som
    nworst=n
  endif
  asom=asom+som
enddo

! damping penalty
ysom=0.
do i=1,ndim
  ysom=ysom+y(i)**2
enddo
ysom=eps*ysom/ndim

func=pflat+asom+ysom          ! add all penalty terms
if(kplot>0) write(4,'(a,i2)') 'func called only for plot=',kplot
write(4,'(4e11.3,i3,e12.4,40f7.3)') pflat,ysom,asom,worst,nworst,  &
  func,(y(i),i=1,ndim)

if(isnan(func)) stop 'ERROR: func returns NaN for misfit'

return
end function func

subroutine plts(n,fout,kplot,npts,dt)

! writes pole-zero response (fout) in both SAC and GMT formats

implicit none
integer, intent(in) :: n,kplot,npts
real*4, intent(in) :: fout(npts),dt
real*4 :: dum
integer :: i,nerr
character*40 :: fname
character*8 :: kstnm

write(fname,'(a4,i1,a1,i2.2,a4)') 'fout',kplot,'_',n,'.sac'
write(kstnm,'(a4,i1,a1,i2.2)') 'iter',kplot,'_',n
```

```
call setnhv('npts',npts,nerr)
call setfhv('b',0.,nerr)
call setfhv('e',(npts-1)*dt,nerr)
call setfhv('delta',dt,nerr)
call setfhv('o',0.,nerr)
call setihv('iztype','IB',nerr)
call setkhv('kstnm',kstnm,nerr)
call wsac0(fname,dum,fout,nerr)

! also output of predicted signal to GMT
fname=fname(1:9)//'xy'
open(2,file=fname)
do i=1,npts
  write(2,*) (i-1)*dt,fout(i)
enddo
close(2)

return
end

subroutine julian(year,month,day,jday)

! returns Julian day if called with year, month, day

implicit none
integer :: year,month,day,jday

integer :: mday(12),kday(12),i,k,m,n,leap
data mday/31,28,31,30,31,30,31,31,30,31,30,31/

leap=0
mday(2)=28
if(mod(year,4).eq.0.and.(mod(year,100).ne.0.or.mod(year,400).eq.0)) leap=1
if(leap.eq.1) mday(2)=29
kday(1)=0
do i=2,12
  kday(i)=kday(i-1)+mday(i-1)
enddo
jday=kday(month)+day

end

subroutine shift0(fin,fobs,npts)

! shift zero time to onset of first pulse (first fin not 0)
integer, parameter :: ND=32768
real*4 :: fin(ND),fobs(ND)
integer :: i,j,npts,n
```

```
n=1
do while (abs(fin(n)).le.0.)
  n=n+1
  if(n>npts) stop 'Error in shift0'
enddo
if(n.le.1) return
fin(1)=fin(n)
fobs(1)=0.
n=n-1
do i=2,npts-n
  fin(i)=fin(i+n)
  fobs(i)=fobs(i+n)
enddo
npts=npts-n
return
end

subroutine resample(y,n,npow,dtold,dtnew)

real*4 :: y(n)
complex(kind=4) :: ft(n)

ft=y
call clogp(npow,ft,-1.0,dtold)
call ftinv(ft,npow,dtnew,y)

return
end

\subsubsection{Subroutines for getpz2.f90}

The routines for nonlinear optimization are from Numerical Recipies
(Press et al., 1992)\nocite{press92}.

\begin{verbatim}
c     FFT routines for real-valued time signals. Not the most efficient,
c     but these have the correct scaling to mimic the Fourier *integral*
c     using the following conventions:

c     H(f) = \int h(t) exp(-2*pi*i*f*t) dt
c     h(t) = \int H(f) exp(+2*pi*i*f*t) df

c     You still need a complex time signal (x) and call clogp with
c     zzsign=-1 to go from time to frequency
c     But if you have a positive spectrum in the first half of (c0mplex)
c     array s, you can use ftinv to go back to a real-valued signal (r)
c     in the time domain.
```

```
      subroutine ftinv(s,npow,dt,r)

      parameter (ND=32768)
      dimension r(ND)
      complex s(ND)

c     Combines construction of negative spectrum, inverse fft, and
c     storage in a real array. It is possible to equivalence s and
c     r in the main program.
c     s(1)...s(nhalf) contain positive complex spectrum of real signal
c     nsmp=2*nhalf=2**npow, dt sampling interval, r destination array
c     WARNING: uses the sign convention of clogp not clogc

      nsmp=2**npow
      nhalf=nsmp/2
      call rspec(s,nhalf)
      call clogp(npow,s,+1.0,dt)
      do 10 i=1,nsmp
10    r(i)=real(s(i))
      return
      end

      subroutine rspec(s,np2)

c     Constructs negative spectrum
c     input: positive spectrum in s(1)...s(np2).
c     output: complete spectrum in s(1)...s(2*np2)

      parameter (ND=32768)
      complex s(ND)
      n=2*np2
      n1=np2+1
c     s(n1)=0.                  ! comment to keep Nyquist frequency
c     s(1)=0.                   ! comment to keep DC offset
      do 20 i=1,np2
20    s(np2+i)=conjg(s(np2+2-i))
      return
      end

      subroutine clogp(n,x,zzign,dt)
c--- performs fft on signals with length 2**n and sampling interval
c--- of dt seconds (if in the time domain; notice that dt*df=1/2**n).
c--- the signal is stored in x. it may be complex.
c--- the spectrum is returned in x. it is almost always complex.
c--- a time-to-frequency transform is done with zign=-1. (conform
c--- the convention adopted in SEED - the alternative
c--- convention may be obtained by taking complex conjugates after
```

```
c--- the call to clogp).
c--- the normalization factor 1./twopi occurs in the frequency-to
c--- time transform.
c--- normalization is such that physical dimensions are respected.
c--- thus, if the time signal is dimensioned in meters, the
c--- resulting spectral density in x is in meters/hz. for example,
c--- if the time signal is the unit sinc function of width dt, centered
c--- at t=0, the spectral density is dt for all values of the frequency.
c
c--- array locations: if x contains the spectrum, it has the spectrum
c--- for positive frequencies in the first 2**n/2+1 elements, such that
c--- x(1) is at 0 hz, x(2) at df hertz, and x(2**n/2+1) at the nyquist,
c--- where df=1./(2**n*dt) and the nyquist is 1./(2*dt) hz.
c--- the second half of x contains the spectrum for negative frequencies
c--- such that x(2**n) is at -df, x(2**n-1) at -2*df hz etcetera.
c--- if x contains the time signal, x(1) is at time 0, x(2)
c--- at time dt etc.
c
      parameter (ND=32768)
      dimension x(ND),m(21)
      complex x,wk,hold,q
      zign=zzign
      if(zign.ge.0.) then
        zign=1.
      else
        zign=-1.
      endif
      lx=2**n
      do 1 i=1,n
    1 m(i)=2**(n-i)
      do 4 l=1,n
      nblock=2**(l-1)
      lblock=lx/nblock
      lbhalf=lblock/2
      k=0
      do 4 iblock=1,nblock
      fk=k
      flx=lx
      v=zign*6.283185308*fk/flx
      wk=cmplx(cos(v),sin(v))
      istart=lblock*(iblock-1)
      do 2 i=1,lbhalf
      j=istart+i
      jh=j+lbhalf
      q=x(jh)*wk
      x(jh)=x(j)-q
      x(j)=x(j)+q
    2 continue
```

```
      do 3 i=2,n
      ii=i
      if(k.lt.m(i)) go to 4
    3 k=k-m(i)
    4 k=k+m(ii)
      k=0
      do 7 j=1,lx
      if(k.lt.j) go to 5
      hold=x(j)
      x(j)=x(k+1)
      x(k+1)=hold
    5 do 6 i=1,n
      ii=i
      if(k.lt.m(i)) go to 7
    6 k=k-m(i)
    7 k=k+m(ii)
      if(zign.lt.0.) go to 9
      flx=flx*dt
      do 8 i=1,lx
    8 x(i)=x(i)/flx
      return
    9 do 10 i=1,lx
   10 x(i)=x(i)*dt
      return
      end


c  optimization routines from Numerical Recipes (Press at el. 1992), adjusted where needed

      FUNCTION BRENT(AX,BX,CX,F,TOL,XMIN)
      PARAMETER (ITMAX=200,CGOLD=.3819660,ZEPS=1.0E-10)
      A=MIN(AX,CX)
      B=MAX(AX,CX)
      V=BX
      W=V
      X=V
      E=0.
      FX=F(X)
      FV=FX
      FW=FX
      DO 11 ITER=1,ITMAX
        XM=0.5*(A+B)
        TOL1=TOL*ABS(X)+ZEPS
        TOL2=2.*TOL1
        IF(ABS(X-XM).LE.(TOL2-.5*(B-A))) GOTO 3
        IF(ABS(E).GT.TOL1) THEN
          R=(X-W)*(FX-FV)
          Q=(X-V)*(FX-FW)
```

```
          P=(X-V)*Q-(X-W)*R
          Q=2.*(Q-R)
          IF(Q.GT.0.) P=-P
          Q=ABS(Q)
          ETEMP=E
          E=D
          IF(ABS(P).GE.ABS(.5*Q*ETEMP).OR.P.LE.Q*(A-X).OR.
     *        P.GE.Q*(B-X)) GOTO 1
          D=P/Q
          U=X+D
          IF(U-A.LT.TOL2 .OR. B-U.LT.TOL2) D=SIGN(TOL1,XM-X)
          GOTO 2
        ENDIF
1       IF(X.GE.XM) THEN
          E=A-X
        ELSE
          E=B-X
        ENDIF
        D=CGOLD*E
2       IF(ABS(D).GE.TOL1) THEN
          U=X+D
        ELSE
          U=X+SIGN(TOL1,D)
        ENDIF
        FU=F(U)
        IF(FU.LE.FX) THEN
          IF(U.GE.X) THEN
            A=X
          ELSE
            B=X
          ENDIF
          V=W
          FV=FW
          W=X
          FW=FX
          X=U
          FX=FU
        ELSE
          IF(U.LT.X) THEN
            A=U
          ELSE
            B=U
          ENDIF
          IF(FU.LE.FW .OR. abs(W-X)<zeps) THEN
            V=W
            FV=FW
            W=U
            FW=FU
```

```
          ELSE IF(FU.LE.FV .OR. abs(V-X)<zeps .OR. abs(V-W)<zeps) THEN
            V=U
            FV=FU
          ENDIF
        ENDIF
11    CONTINUE
      print *, 'Brent exceeds maximum iterations.'
      stop
3     XMIN=X
      BRENT=FX
      RETURN
      END
      FUNCTION F1DIM(X)
      PARAMETER (NMAX=50)
      COMMON /F1COM/ NCOM,PCOM(NMAX),XICOM(NMAX)
      DIMENSION XT(NMAX)
      DO 11 J=1,NCOM
        XT(J)=PCOM(J)+X*XICOM(J)
11    CONTINUE
      F1DIM=FUNC(XT)
      RETURN
      END
      SUBROUTINE LINMIN(P,XI,N,FRET)
      PARAMETER (NMAX=50,TOL=1.E-4)
      EXTERNAL F1DIM
      DIMENSION P(N),XI(N)
      COMMON /F1COM/ NCOM,PCOM(NMAX),XICOM(NMAX)
      !write(13,*) 'calling linmin, n,p=',n,(p(i),i=1,N)
      !write(13,*) 'xi=',(xi(i),i=1,n)
      NCOM=N
      DO 11 J=1,N
        PCOM(J)=P(J)
        XICOM(J)=XI(J)
11    CONTINUE
      AX=0.
      XX=1.
      BX=2.
      CALL MNBRAK(AX,XX,BX,FA,FX,FB,F1DIM)
      FRET=BRENT(AX,XX,BX,F1DIM,TOL,XMIN)
      !write(13,*) 'brent returns fret,xmin=',fret,xmin
      !flush(13)
      DO 12 J=1,N
        XI(J)=XMIN*XI(J)
        P(J)=P(J)+XI(J)
12    CONTINUE
      RETURN
      END
      SUBROUTINE MNBRAK(AX,BX,CX,FA,FB,FC,FUNC)
```

```
      PARAMETER (GOLD=1.618034, GLIMIT=100., TINY=1.E-20)
      FA=FUNC(AX)
      FB=FUNC(BX)
      IF(FB.GT.FA)THEN
        DUM=AX
        AX=BX
        BX=DUM
        DUM=FB
        FB=FA
        FA=DUM
      ENDIF
      CX=BX+GOLD*(BX-AX)
      FC=FUNC(CX)
1     IF(FB.GE.FC)THEN
        R=(BX-AX)*(FB-FC)
        Q=(BX-CX)*(FB-FA)
        U=BX-((BX-CX)*Q-(BX-AX)*R)/(2.*SIGN(MAX(ABS(Q-R),TINY),Q-R))
        ULIM=BX+GLIMIT*(CX-BX)
        IF((BX-U)*(U-CX).GT.0.)THEN
          FU=FUNC(U)
          IF(FU.LT.FC)THEN
            AX=BX
            FA=FB
            BX=U
            FB=FU
            GO TO 1
          ELSE IF(FU.GT.FB)THEN
            CX=U
            FC=FU
            GO TO 1
          ENDIF
          U=CX+GOLD*(CX-BX)
          FU=FUNC(U)
        ELSE IF((CX-U)*(U-ULIM).GT.0.)THEN
          FU=FUNC(U)
          IF(FU.LT.FC)THEN
            BX=CX
            CX=U
            U=CX+GOLD*(CX-BX)
            FB=FC
            FC=FU
            FU=FUNC(U)
          ENDIF
        ELSE IF((U-ULIM)*(ULIM-CX).GE.0.)THEN
          U=ULIM
          FU=FUNC(U)
        ELSE
          U=CX+GOLD*(CX-BX)
```

```
        FU=FUNC(U)
      ENDIF
      AX=BX
      BX=CX
      CX=U
      FA=FB
      FB=FC
      FC=FU
      GO TO 1
    ENDIF
    RETURN
    END
    SUBROUTINE POWELL(P,XI,N,NP,FTOL,ITER,FRET)
    PARAMETER (NMAX=40,ITMAX=1500)
    DIMENSION P(NP),XI(NP,NP),PT(NMAX),PTT(NMAX),XIT(NMAX)
    FRET=FUNC(P)
    DO 11 J=1,N
      PT(J)=P(J)
11  CONTINUE
    ITER=0
1   ITER=ITER+1
    FP=FRET
    IBIG=0
    DEL=0.
    DO 13 I=1,N
      DO 12 J=1,N
        XIT(J)=XI(J,I)
12    CONTINUE
      CALL LINMIN(P,XIT,N,FRET)
      IF(ABS(FP-FRET).GT.DEL)THEN
        DEL=ABS(FP-FRET)
        IBIG=I
      ENDIF
13  CONTINUE
    IF(2.*ABS(FP-FRET).LE.FTOL*(ABS(FP)+ABS(FRET))) then
      RETURN
    endif
    IF(ITER.EQ.ITMAX) print *, 'Powell exceeding maximum iterations.'
    IF(ITER.EQ.ITMAX) stop
    DO 14 J=1,N
      PTT(J)=2.*P(J)-PT(J)
      XIT(J)=P(J)-PT(J)
      PT(J)=P(J)
14  CONTINUE
    FPTT=FUNC(PTT)
    IF(FPTT.GE.FP)GO TO 1
    T=2.*(FP-2.*FRET+FPTT)*(FP-FRET-DEL)**2-DEL*(FP-FPTT)**2
    IF(T.GE.0.)GO TO 1
```

```
      CALL LINMIN(P,XIT,N,FRET)
      DO 15 J=1,N
        XI(J,IBIG)=XIT(J)
15    CONTINUE
      GO TO 1
      END
```