DIGITAL SIGNAL PROCESSING TECHNIQUES
FOR
LASER-DOPPLER ANEMOMETRY

by

PATRICK P. ERK

B.S., Technische Universität Berlin, West Germany
(1987)

Submitted to the Department of
Aeronautics and Astronautics
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science
in Aeronautics and Astronautics

at the

Massachusetts Institute of Technology
September 1990

Signature of Author____________________________
Department of Aeronautics and Astronautics
September 28, 1990

Certified by____________________________
C. Forbes Dewey, Jr.
Professor of Mechanical Engineering
Thesis Supervisor

Accepted by____________________________
Professor Harold Y Wachman
Chairman, Departmental Graduate Committee

DIGITAL SIGNAL PROCESSING ALGORITHMS FOR LASER-DOPPLER ANEMOMETRY

by

PATRICK P. ERK

Submitted to the Department of Aeronautics and Astronautics
on September 28, 1990 in Partial Fulfillment
of the Requirements for the Degree of
Master of Science in Aeronautics and Astronautics

Abstract

Digital signal processing algorithms in laser-Doppler anemometry still lag behind the standards used in Doppler-radar or sonar technology. The two main problems in laser-Doppler signal processing are signal detection and frequency estimation. In this Thesis, a software system for use in flow experiments with laser-Doppler anemometry has been developed. It features programs for digital prefiltering, for FIR filter design, for burst detection, and for frequency estimation. Spectral estimation is done with an algorithm based on the discrete Fourier transform or with an auto-regressive moving-average algorithm based on a Pade approximation to the signal spectrum. The Modified Covariance Algorithm and the Iterative Filtering Algorithm have also been tested on synthetically generated Doppler signals. Flow experiments with a cone-and-plate flow show the workability of the software system. These results also confirm the validity of the assumptions made in the numerical simulations.

Thesis Supervisor: C. Forbes Dewey, Jr., PhD
Title: Professor of Mechanical Engineering

Acknowledgements

I would like to thank ...

... Prof. Dewey for this great learning opportunity and his support

... Donna for teaching me what the US is all about and who is in the end responsible for this mess

... Natacha for Tacos, housing opportunity in NYC, and a gorgeous flow apparatus

... Bob and Andrew for many, many walks to the local coffee supply, among other things

... Jason for doing most of the work with the laser, asking millions of questions, and a ride on his roller blades

... Hannes for storing my furniture in Berlin for two years and not selling it to my now free but still deprived eastern homeboys

... My family for their suggestions and comments in all questions of life

... Isabella for giving me such a hard time (sigh)

... And my friends for giving me such a good time

PPE

CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. Introduction

Laser-Doppler anemometry (LDA) has proven to be an extremely useful experimental tool for measuring fluid flow velocities. Its advantages over the complementary technique in fluid mechanics, hot-wire anemometry, are the non-intrusive nature of LDA measurements, the large dynamic range from zero to supersonic speeds, and a versatility and extendibility to special experimental conditions such as 3-dimensional velocity measurements and measurement of the different phase velocities in multi-phase flows.

The fundamental principle behind laser-Doppler anemometry is the detection of the Doppler frequency of light scattered from particles traversing the point of intersection of two laser beams. This Doppler frequency is directly proportional to the velocity of the particle. Due to the finite size of the measurement volume, inaccuracies occur in the presence of velocity gradients or turbulence. Proper averaging techniques (cf. Section VI) may reduce the systematic error in the measurements for these cases.

Noise in laser-Doppler anemometry is mainly non-Gaussian [9] and comes from the photodetector or from scattered light from surfaces or distant particles. The signal-to-noise ratio can be significantly improved if fluorescent rather than scattering particles are used [20][34].

Recent publications concentrate on the use of Fourier transform of either the data or the autocorrelation function of the data [2][9][11][17][18][22][26][31]. Both methods represent the classic way of doing spectral analysis. New methods of spectral estimation have been developed over the last ten years [14][19][25]. They are based on fitting the coefficients of a linear constant coefficient equation (cf. Section V.1) to the data using some error minimizing criterion (e.g. least squares). These newer algorithms are now routinely used for detection and analysis of sonar and radar signals [13][14][29]. Their use in laser-Doppler signal processing is only limited [33]. This Thesis applies some of these more "advanced" algorithms to laser-Doppler anemometry.

Numerical simulations were carried out in order to assess the performance of the different algorithms for LDA signals . The "traditional" spectral analysis procedure in LDA is compared to three other methods: an auto-regressive estimator, the Modified Covariance Method (cf. Section V.3.4), an enhancement of this algorithm, the Iterative Filtering Algorithm (cf. Section V.4), and an auto-regressive moving-average estimator based on the Pade approximation (cf. Section V.5). Then, the most promising method, the Pade estimator, is tested in a real flow experiment (cone-and-plate flow, cf. Section IX.1). For these flow experiments, it was necessary to implement a system of programs for processing LDA signals (i.e. sampling, filtering and digital filter design, burst validation, spectral estimation, averaging, plot of spectrum/velocity).

The present paper is divided into three major parts: the first part explains the underlying principles of LDA and of the digital signal processing techniques. The second part describes the numerical simulations, their results, and the software system for the flow experiments. The third section introduces the cone-and-plate flow and the experimental set-up and discusses the results obtained by the software system. A final discussion and evaluation is then presented.

II. The Laser-Doppler Anemometer

The laser-Doppler anemometer (LDA) is an optical experimental method which allows the instantaneous, non-intrusive measurement of the velocities within a flow field. It is based on the interference of two Gaussian laser beams, and on the light scattering properties of particles within the flow. Generally, the analytical treatment can be kept quite simple, because all mathematical approximations are of higher accuracy than the usually noisy signal. Recent improvements in the signal quality involve the use of fluorescent rather than scattering particles [20][34].

A typical set-up of a one-component, fringe-mode LDA is shown in Fig. 1.

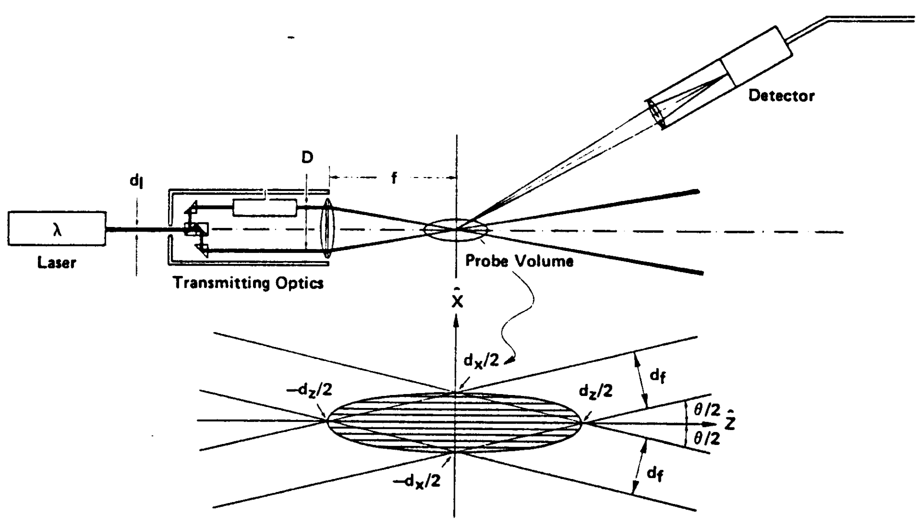


Figure 1. Single-component, fringe-mode laser-Doppler anemometer [from 7]

A Gaussian laser beam is split and the resulting two parallel branches are focused at a point of interest within the flow field. At the point of intersection, a fringe pattern will be formed. Particles crossing that fringe pattern will scatter the light with a frequency which is directly proportional to the particle velocity (Doppler frequency). This signal is received by a photodetector and then further processed.

II.1 The Probe Volume

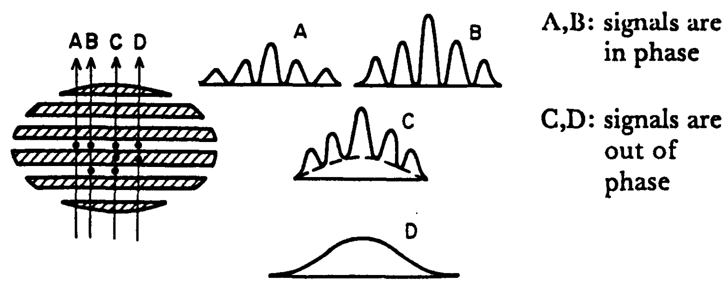
The volume contained in the $\frac{1}{e^2}$-intensity contour of the interference field created by the intersecting laser beams is called the probe volume. It is of ellipsoidal shape and filled with an equidistant and parallel fringe system.

In order for the system to be applicable, the probe volume must meet the following two requirements:

- The probe volume has to be as small as possible, as the velocities at a point in the flow field have to be measured; a large probe volume would only give the spatial average of the velocities around the point of intersection. In addition, a larger probe volume would increase the probability that more than one particle crosses the fringe pattern at the same time. This would result in a random superposition of Doppler frequencies, reducing the signal quality in the case of destructive interference of the two signals (cf. Fig. 2). In practice, the signal processing procedures and optics impose a lower limit on the number of fringes and on the size of the probe volume.

- The fringe pattern has to be symmetric, so that particles traversing the probe volume at different locations cross fringes with the same distance. If this is not the case, the measured velocity will depend on the crossing path of the particles.

The system is maximized with respect to both items if the waists of the *focused* laser beams coincide with the point of intersection.



If there is more than 1 particle present in the measuring control volume, constructive or destructive superpositions of signals can occur.

Figure 2. Effect of several particles crossing the probe volume at the same time [from 8]

Misalignment of the laser beams may result in an elongated and diverging fringe pattern (cf. Fig. 3) and the probe volume will not be the smallest possible one. If the waist of the focused beam, w_f, lies within the probe volume the formulas of Table 1 describing the geometry of the probe volume apply. For computing w_f and the angle of intersection, ϕ, see Section II.4 below.

Usually, these effects can be neglected except for large distances between lens and probe volume. The geometry of the probe volume depends on the angle of intersection, ϕ: If ϕ increases, the length of the ellipsoid, d_z, its height, d_y, the fringe spacing, Δx, and the number of fringes, N_f will decrease.

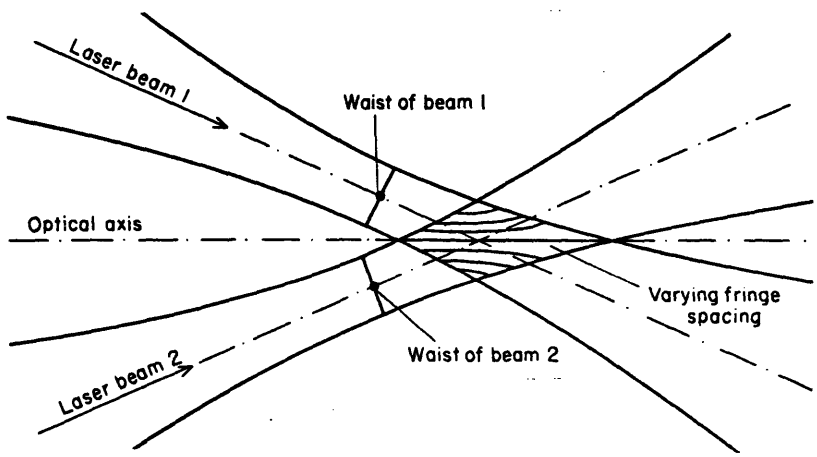


Figure 3. Misalignment of the beams creates an asymmetric fringe pattern [from 8]

The system will detect only the velocity component perpendicular to the fringes. Oblique passing with the same absolute velocity results in a corresponding decrease of the Doppler frequency.

The analytical description of the time behavior of the Doppler signal is:

$$s(t) = s_0 \left\{ 1 + \frac{\Delta x}{2\pi r_p} \sin\left[\frac{2\pi r_p}{\Delta x}\right] \cos\left[\frac{2\pi}{\Delta x}\,(U_o\, t + r_{c,0})\right] \right\} \qquad \text{(II.1)}$$

r_p: radius of the particle
Δx: fringe spacing
$r_{c,0}$: location of center of particle at time t=0
U_o: velocity component perpendicular to fringes
s_0: scaling factor for intensity

The Doppler frequency is immediately recognized in the argument of the sine as $\frac{U_0}{\Delta x}$.

Noise in the Doppler signal may come from the following sources:

Waist diameter of focused beam	d_f
wavelength of laser	λ
focal length of front lens	f
angle of intersection	ϕ
half-angle of intersection	$\theta = \frac{\phi}{2}$
axes of measurement volume	$d_x = d_f$ $d_y = \frac{d_f}{\cos\theta}$ $d_z = \frac{d_f}{\sin\theta}$
fringe spacing	$\Delta x = \frac{\lambda}{2\sin\theta}$
number of fringes	$N_f = \frac{d_x}{\Delta x}$

TABLE 1. Geometry of the probe volume. See Fig. 1 for definition of the variables

- Light from particles not crossing the fringe pattern may hit the photodetector. This effect is taken care of by pinholes in front of the photomultipliers which limit the depth of the optical field.

- Noise generated in the photoelectric cells stems either from the random emission of photons (shot noise, Poisson character), from the random movements of electrons (Johnson noise), and from thermal excitation of electrons. All three types of noise are broad band and signal-independent. The shot noise depends on the mean photocurrent: The higher the mean photocurrent the higher the random fluctuations the higher the shot noise.

II.2 The Scattering Process

The expressions up to now relate only the Doppler frequency to the particle velocity. Statements about the field distribution of the scattered light are necessary to optimize signal detection. Analytical results exist only for the case of spherical and ellipsoidal particles (Mie's Scattering).

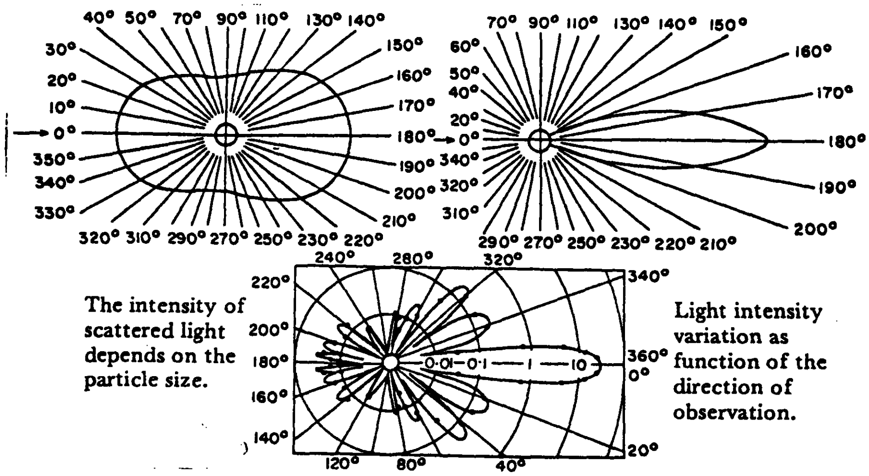


Figure 4. Intensity distribution due to scattering [from 8] (direction of the incident light indicated by arrow, particle size decreasing from from (a) to (b))

The most important result is the non-uniformity of the intensity distribution of the scattered light (cf. Fig. 4). The intensity of the scattered light will be highest in the direction away from the focusing optics. This is exploited in the forward-scatter LDA (cf. Fig. 1). The backscatter mode (cf. Fig. 20), the type used in this project, results in a more compact design but suffers from less intense scattering. Thus, in the backscatter mode, a smaller signal-to-noise ratio can be expected.

The size of the particles has to be matched to the geometry of the probe volume. Particles which are too large will cross more than one fringe at a time, but will scatter more light; particles which are too small produce very weak signals (Fig. 5). It is recommended that the particle diameter is a fourth of the fringe distance [9].

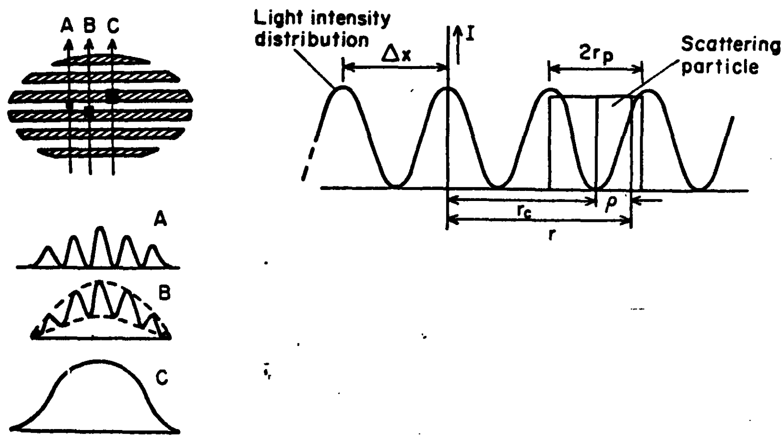


Figure 5. Signal quality and signal strength depend on the particle size (from [9])

II.3 Fluorescent Particles

The signal-to-noise ratio in the backscatter mode can be greatly improved if fluorescent particles are used. Fluorescent particles absorb the incident light and emit radiation at a longer wavelength. In contrast to the scattering process fluorescent radiation is emitted in all directions.

The different wavelength of the fluorescent light allows the use of optical filters to block all other wavelengths. These filters will reduce the total amount of light at the photomultiplier and therefore decrease the amount of shot noise in the signal as the mean photocurrent is reduced.

II.4 Computing the Probe Volume Size

The most efficient way to compute the propagation of laser beams is done with *ray transfer matrices* (cf. [16]). Each optical element has a particular 2×2 transfer matrix. The transfer matrix of a system of optical components is simply the product of the transfer matrices of the single components. Although developed originally for the study of the propagation of paraxial rays, it can also describe the propagation of laser beams without changing the matrices for the different optical building blocks.

A paraxial ray is characterized by its distance x from the optical axis (the z-axis) and by the angle θ with respect to this axis. For paraxial rays, θ will be small enough to allow the approximations $\sin\theta \approx \theta$ and $\cos\theta \approx 1$.

If we define a *ray vector*, **r**, by $\mathbf{r} \equiv \begin{bmatrix} x \\ \theta \end{bmatrix}$, the ray vector behind an optical arrangement, $\mathbf{r}_n$ is obtained by multiplying $\mathbf{r}$ from the right to the system ray transfer matrix, $\mathbf{X}$: $\mathbf{r}_n = \mathbf{X}\,\mathbf{r}$.

The optical system used in this project is depicted in Fig. 6. The paraxial laser beams are focused by a front lens, propagate through a glass plate, and intersect in a medium of different refractive index. The laser beams are not complanar with the optical axis. The ray transfer matrices for the different stages are:

For the focusing lens (focal length f_1):

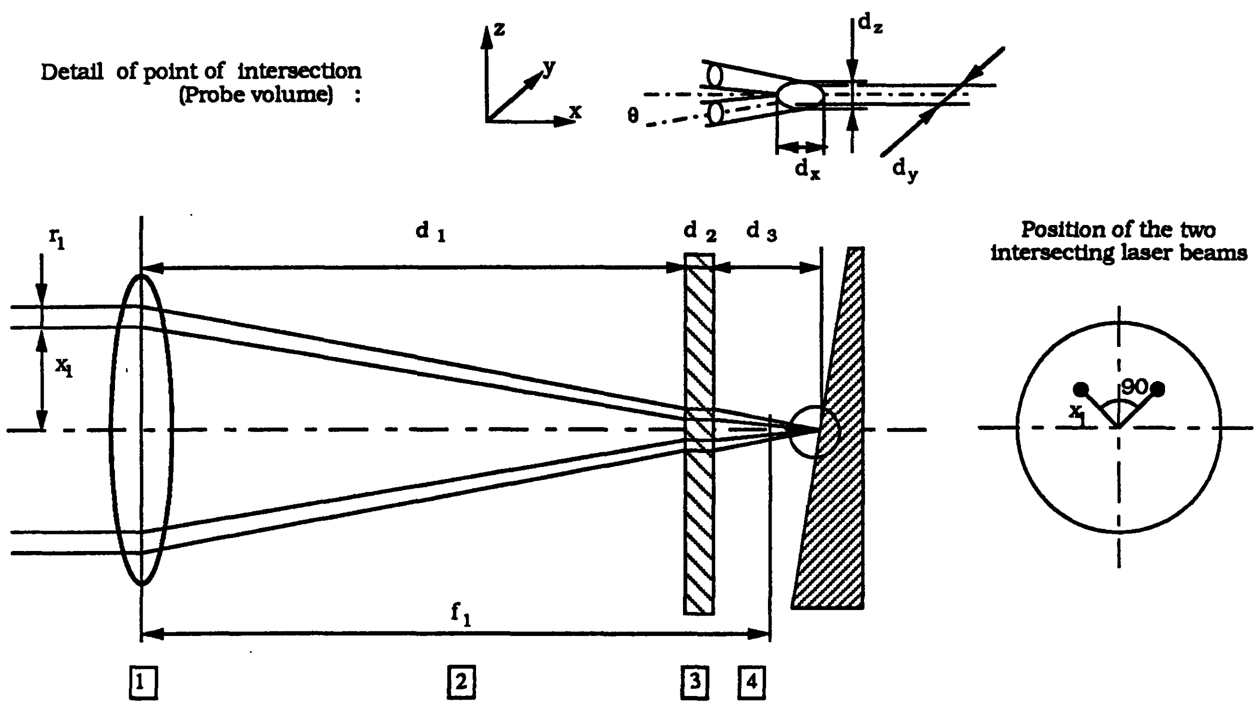


Figure 6. Optical configuration for LDA in this project

$$\mathbf{X}_1 = \begin{bmatrix} 1 & 0 \\ -\dfrac{1}{f_1} & 1 \end{bmatrix} \tag{II.2}$$

For the translation between lens and first surface of the glass plate (index of refraction ≈1, distance d_1):

$$X_2 = \begin{bmatrix} 1 & d_1 \\ 0 & 1 \end{bmatrix} \tag{II.3}$$

For the refractions at the air-glass and the glass-fluid interfaces, the ray transfer matrices will be unity (Snell's law). Thus, they do not contribute to the overall system matrix and can be omitted.

For the propagation within the glass (thickness d_2, index of refraction n_1):

$$X_3 = \begin{bmatrix} 1 & \frac{d_2}{n_1} \\ 0 & 1 \end{bmatrix} \tag{II.4}$$

For the propagation of the laser beams through the fluid (index of refraction n_2) to the point of intersection (distance d_3 from the wall):

$$X_4 = \begin{bmatrix} 1 & \frac{d_3}{n_2} \\ 0 & 1 \end{bmatrix} \tag{II.5}$$

The resulting system matrix is:

$$X = X_4\, X_3\, X_2\, X_1 = \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} 1 - \dfrac{d_1 + \frac{d_2}{n_1} + \frac{d_3}{n_2}}{f_1} & d_1 + \dfrac{d_2}{n_1} + \dfrac{d_3}{n_2} \\ -\dfrac{1}{f_1} & 1 \end{bmatrix} \tag{II.6}$$

The ray vectors at position 1, $r_1 = \begin{bmatrix} x_1 \\ 0 \end{bmatrix}$, and position 4, $r_4 = \begin{bmatrix} 0 \\ \theta' \end{bmatrix}$ are related via:

$r_4 = X\, r_1$. This yields two equations for the two unknowns d_1 and θ':

$$d_1 = f_1 - \frac{d_2}{n_1} - \frac{d_3}{n_2} \tag{II.7a}$$

$$\theta' = -\frac{x_1}{f_1} \tag{II.7b}$$

The half angle of intersection is found from geometry:

$$\sin\theta = \frac{\sin\theta'}{\sqrt{2}} \tag{II.8}$$

A laser beam of wavelength λ and waist diameter w_0 propagating along the x-axis has a $\frac{1}{e}$-intensity envelope given by:

$$w(z) = w_0 \, [\, 1 + \left[\frac{\lambda z}{\pi w_0^2}\right]^2]^{1/2} \tag{II.9}$$

and a curvature of the wave front

$$R(z) = z \, [\, 1 + \left[\frac{\pi w_0^2}{\lambda z}\right]^2]^{1/2} \tag{II.10}$$

Defining a complex propagation parameter at a location z_1, $\frac{1}{q(z_1)} = R(z_1) - j\frac{\lambda}{\pi w(z_1)^2}$ or, equivalently, $q(z_1) = z_1 + j\frac{\pi w_0^2}{\lambda}$, the beam diameter and the field curvature at a position z_2 behind an optical system can be obtained with the complex propagation parameter at this location and the system's transfer matrix via:

$$q(z_2) = \frac{A\ q(z_1) + B}{C\ q(z_1) + D} \tag{II.11}$$

This approach will be used in Section IX.2.1, where we calculate the approximate geometry of the probe volume.

III. Digital Spectral Estimation: Survey of Methods

In the previous chapter we defined the basic problem of measuring the flow velocity as a problem of detecting the burst and estimating the Doppler frequency of a sinusoidal signal in non-Gaussian noise. Estimation of the frequency content (i.e. spectral analysis) from sampled data is a problem occurring in many fields of research. Therefore, a variety of algorithms suitable for implementation on a general purpose computer has been developed. Fig. 7 presents an overview which is by no means exhaustive.

We can roughly divide the available spectral analysis techniques into three categories: classical methods, methods based on parameter estimation, and other methods.

Classical methods are primarily characterized by their robustness at low SNRs and their computational speed. They comprise the periodograms, where the data record is segmented, each segment is then multiplied with a time-window function. Then the power spectral density (PSD) is estimated by averaging the Fourier-transformed data segments. The unmatched speed of the classical spectral estimators results from the use of the fast Fourier transform algorithm (FFT) for evaluating the discrete-time Fourier transform.

Spectral estimators based on parameter models assume that the - unknown - spectrum of the observed data stems from filtering a driving white noise process with a linear time-invariant system (filter) (cf. Fig. 8). Their basic feature is a very high spectral resolution which enables for example the detection of closely spaced sinusoids. If the data contain additive white noise (so-called observation noise), the whole process is

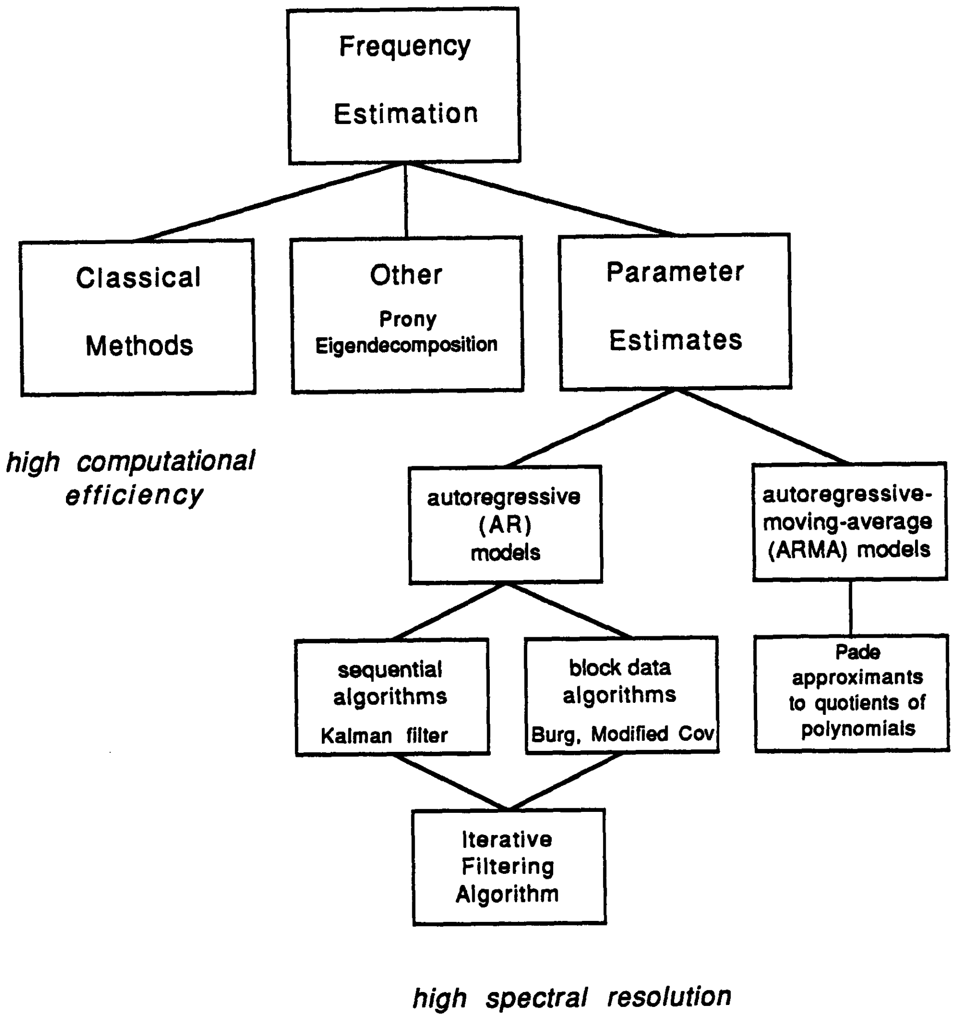


Figure 7. Survey of Digital Spectral Estimation Techniques as applicable to LDA

approximately modeled by a white-noise source whose output is added to the system output.

The underlying assumption of a linear-time invariant filter yields linear constant-coefficient difference equations for the unknown filter coefficients. These may be solved by minimizing for example the mean square error between the actual data and the data

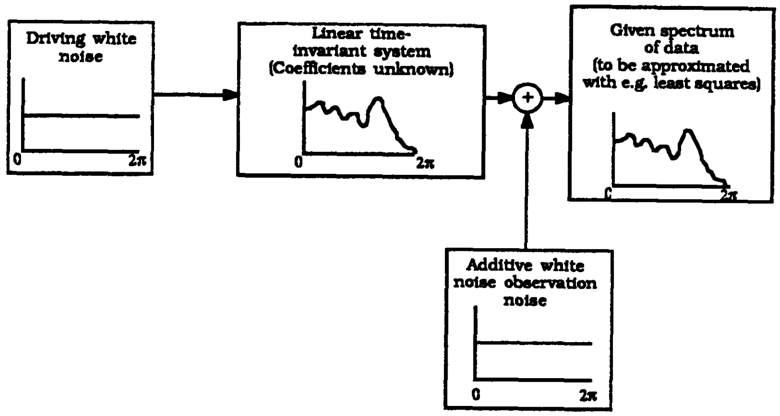


Figure 8. Spectral Estimation by Parameter Models

one would obtain by the filtering process.

Although most real processes do not follow these linear difference equations, i.e. are non-linear, the methods are widely used with good results in many fields [14][16].

As this approach generally requires the data to have zero mean, high-pass filtering the data before their evaluation is indispensable. This, however, is of no major concern in LDA, as this kind of filtering is advisable for removing the signal pedestal and the dc component.

According to the filter type adapted to the data we distinguish between autoregressive (AR) models where the frequency response of the filter driven by white noise has only poles, and autoregressive moving-average (ARMA), where the filter possesses both zeros and poles.

The AR estimation techniques now can be subdivided into block data algorithms and sequential data algorithms [14][19]. Sequential data algorithms update the filter coefficients each time a new datum comes in. As each update requires a certain number of operations, these methods cannot be applied in steady-state sense with the data acquisition. The time required for the update exceeds - at least at the sampling frequencies typical for LDA (500 kHz to a few MHz - by far the time between incoming data. Block data algorithms, on the other hand, take one batch of data at a time and process it. Their performance is slightly superior to that of sequential data algorithms [14][19]. Also, many data acquisition systems, including the one of the computer system used in this project, operate with buffer queues: they fill one buffer in the memory with the sampled data; when the buffer is full it is released to the operating system; then the next buffer waiting in the queue is fetched and so on. A processed buffer is put back in the queue. Thus, block data algorithm seem to be a more natural way to handle spectral estimation.

A widely-used sequential data algorithm is the Kalman filter. Two examples for ARMA methods are the Modified Covariance Method - explained in some detail further below - and the Burg method. Both are very similar, but the former method has - at the same order of computational complexity - more favorable features [14].

ARMA methods are mathematically somewhat more complicated. The resulting least-squares equations are non-linear and cannot be solved efficiently. Certain assumptions, however, lead to linearized forms. ARMA models have the advantage of being able to model noisy processes which are characterized by zeroes in the spectrum.

The method we tested is based the approximation of a high-order polynomial by a quotient of two (finite) polynomials, termed Pade approximation. At the core of this algorithm lies the Euclidean algorithm.

The other methods, the best known is probably Prony's method - do not really produce results of higher quality than the preceding two classes. Also, as their computational complexity is not superior to the parameter estimation techniques, they are not considered in the remainder of this paper. The Pisharenko Harmonic Decomposition for instance is based on the eigenvalues and eigenvectors of the autocorrelation matrix. Further information about parameter models in spectral estimation may be found in [14][19].

The Iterative Filtering Algorithm finally is a technique which enhances the results of the AR spectral estimators, which usually perform poorly in the presence of noise [12][13][14].

IV. Classical Methods of Spectral Estimation

All Classical Methods are based on the computation of the discrete Fourier transform (DFT) either directly (DFT of the data) or indirectly (DFT of the autocorrelation function). In both cases the properties of the DFT are of importance, the are outlined in the first section of this chapter.

For the two situations in LDA, scarcely seeded flow and discontinuous signals *(burst-type LDA),* and densely seeded flows and continuous signal, we have to use different methods for correctly computing the mean Doppler frequency.

The proper averaging strategy for the burst-type LDA, a method termed residence-time weighting will be introduced in Section VI.3.

In the case of densely seeded flows, we may apply a simple arithmetic averaging strategy. Therefore the correlogram and periodogram become valid signal processing algorithms. The computationally most efficient periodogram method is termed the Nuttall-Cramer method and is explained below.

If we are more interested in the velocity fluctuations than in the mean we can present the data in the form of a spectrogram, the way all results of the numerical simulations in Section VII are depicted. In a spectrogram the variation of the spectrum with respect to time is shown.

IV.1 Properties of the Discrete Fourier Transform

The resolution of all classical spectral estimators using the DFT depends on the length of the data set $x[n]$: For a data set of length N, sampled with an effective[1] sampling frequency of f_s, frequencies spaced less than $\frac{f_s}{N}$ cannot be resolved (cf. Fig. 9). Also, zero-padding of the data - i.e. lengthening the data record by adding zeros - does not increase the resolution. The only way to accomplish higher resolution is to include more data points.

The power spectral density (PSD) of a data set $x[n]$ which tells us how the energy is distributed over the frequency bands is the modulus squared of the DFT:

$$P_{xx}(f) = \frac{1}{N} \left| \sum_{n=0}^{N-1} x_w[n]\, e^{j\frac{-2\pi kn}{N}} \right|^2 \qquad \text{(IV.1)}$$

$f = \frac{k}{N} f_s,\ k = 0, 1, \cdots, N-1$
f_s: Sampling frequency
$x_w[n] = x[n]\, w[n]$: windowed time series
$w[n]$: time-window function (e.g. rectangular window: $w[n] = 1, 0 \le n \le N-1,\ w[n] = 0$)

The DFT in Eq. (IV.1) can be evaluated most efficiently by an FFT algorithm which gives rise to the unmatched speed of these methods.

1. the reason I mention an *effective* sampling frequency is because we can change the sampling rate with a discrete-time process: Downsampling (and preceding lowpass filtering) reduces the sampling frequency, upsampling (with successive highpass filtering) increases the sampling frequency.

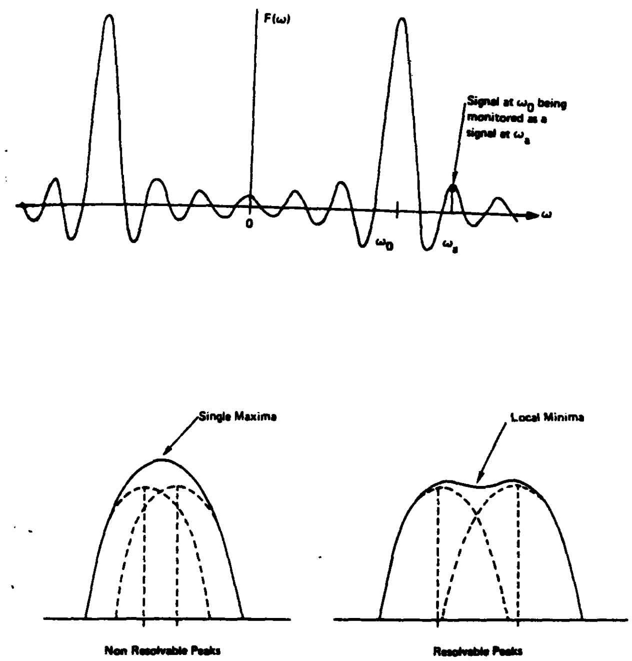


Figure 9. Low resolution resulting from finite-length data set [from 10]

The following argument illustrates the limited resolution of the DFT: If we window an infinite-length data set $x_\infty[n]$ with a time-window function $w[n]$ which is zero for all $n < 0$ and $n \geq N$:

$$\sum_{n=-\infty}^{+\infty} x_\infty[n]\, w[n] e^{-j\frac{2\pi kn}{N}} \rightarrow \sum_{n=0}^{N-1} x_\infty[n]\, w[n] e^{-j\frac{2\pi kn}{N}}, \qquad \text{(IV.2)}$$

then this multiplication in the time domain corresponds to a convolution in the frequency domain. We can rewrite the product in the previous expression:

$$x[n]\, w[n] \rightarrow X[k] * W[k]$$

If the length of the time window increases, i.e. $N \rightarrow \infty$, it follows that $W[k] \rightarrow \delta[k]$, the Fourier transform of the window approaches a delta function and we are left with the original spectrum of our data, as $X[k] * \delta[k] \equiv X[k]$.

For a finite window length however, $W[k]$ will be some function depending on the detailed form of $w[n]$, and will smear the spectrum in the convolution process shown in Fig. 9. As we decrease the distance between two frequency peaks we eventually reach a point where the two peaks are smeared into one single peak and we have come to the limit of our resolution. This whole phenomenon is called spectral leakage.

Window	Highest Side-lobe Level (dB)	Main Lobe Bandwidth (Bins)	3-dB Bandwidth (Bins)	6-dB Bandwidth (Bins)
Rectangular	-13	1.00	0.89	1.21
Hamming	-43	1.36	1.30	1.81
Dolph-Chebyshev ($\alpha = 2.5$)	-50	1.39	1.33	1.85

Equation of Window for $0 \le n \le N-1$		
Rectangular	Hamming	Dolph-Chebyshev
$w[n] = 1$	$w[n] = 0.54 - 0.46 \cos \frac{2\pi n}{N}$	$W[k] = (-1)^k \frac{\cos\left[N \cos^{-1}\left[\beta \cos \frac{\pi k}{N}\right]\right]}{\cosh\left[N \cosh^{-1}\beta\right]}$ $\beta = \cosh\left[\frac{1}{N} \cosh^{-1} 10^{\alpha}\right]$

TABLE 2. Properties of Rectangular, Hamming, and Dolph-Chebyshev window[2] [10]

The choice of the window can be important, as the width of the main lobe influences the resolution and the height of the side lobes control partly the variance in the estimate. Three windows - Rectangular, Hamming, and Dolph-Chebyshev - are presented in Figs. 10, 11, 12, their properties are listed in Table 2. The rectangular window possesses the narrowest main lobe and the highest side lobes of all windows. The

Hamming and the Dolph-Chebyshev window have somewhat broader main lobes and considerably smaller side lobes. They are mostly recommended because they optimize both quantities.

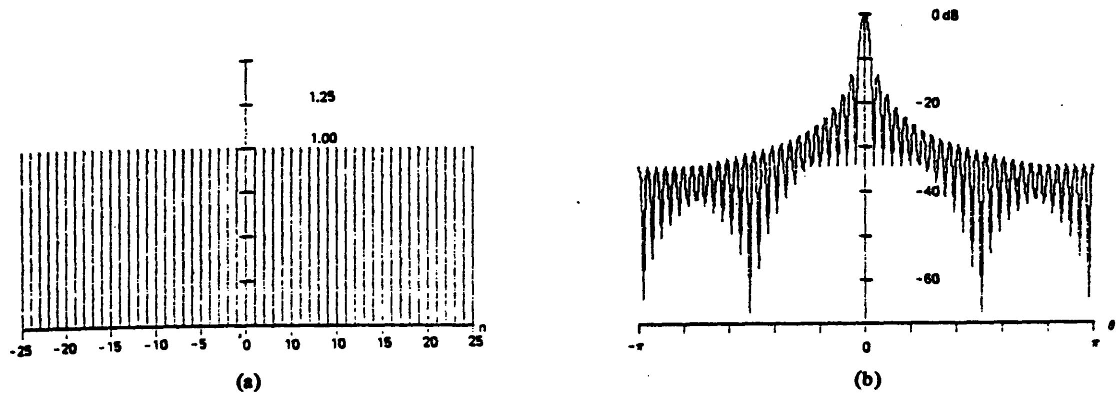


Figure 10. Rectangular window: (a) time series, (b) log-magnitude of DFT [from 10]

IV.2 Periodogram: The Nuttall-Cramer Method

The Nuttall-Cramer Method computes the mean spectrum of a long data record by taking the average of the spectra of segments of this record. This approach yields a low variance in the estimate. The procedure is summarized as follows:

2. The Dolph-Chebyshev window is given by its Fourier Transform. Here, k, the discrete frequency index, has the same range as n.

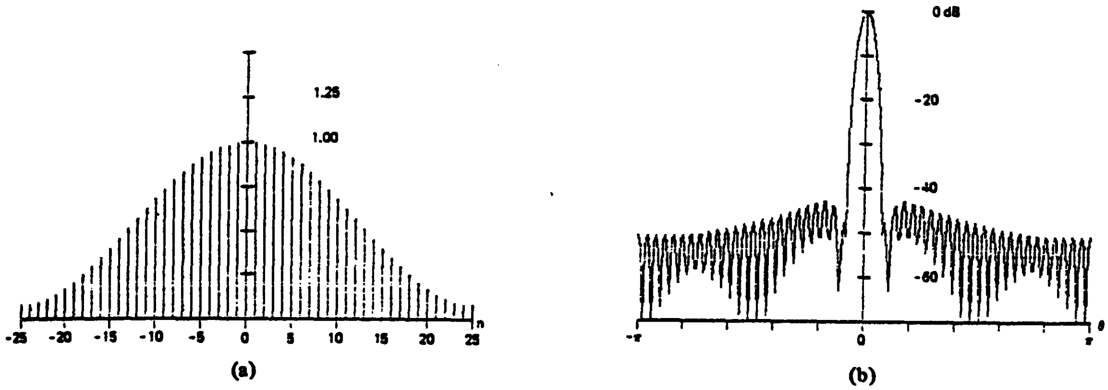


Figure 11. Hamming window: (a) time series, (b) log-magnitude of DFT [from 10]

We divide the data record $x[n]$ of length N into S non-overlapping segments of length M. Then the i^{th} segment is represented by:

$$x_i[m] = x[iM + m] \ ; \quad 0 \le i \le S-1 \ ; \quad 0 \le m \le M-1$$

For each of the segments we compute the PSD via:

$$P_i[k] = \left| \sum_{m=0}^{M-1} x_i[m] \, e^{-j\frac{2\pi km}{M}} \right|^2 \tag{IV.3}$$

Then the arithmetic mean is taken:

$$\bar{P}_{xx}[k] = \sum_{i=0}^{S-1} P_i[k]$$

The inverse DFT of the averaged spectrum is an estimate for the autocorrelation function:

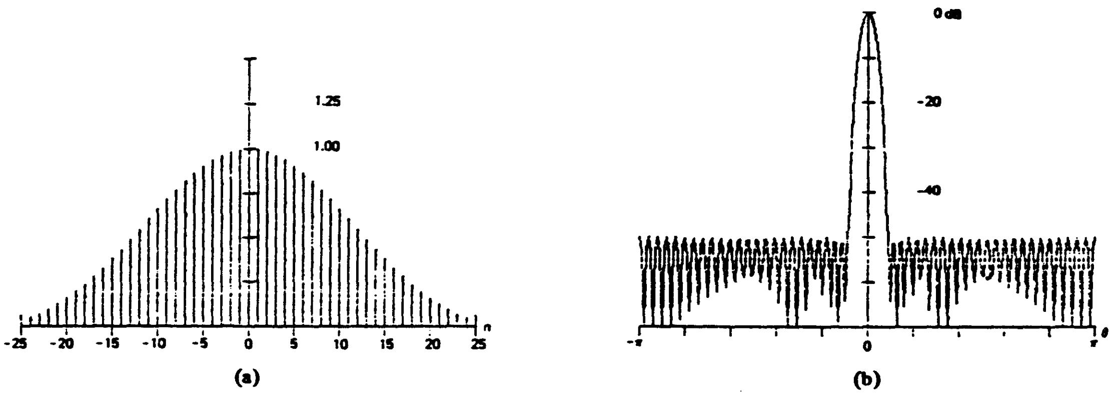


Figure 12. Dolph-Chebyshev window $\alpha = 2.5$: (a) time series, (b) log-magnitude of DFT [from 10]

$$r_{xx}[n] \equiv \bar{p}[n] = \frac{1}{M} \sum_{k=0}^{M-1} \bar{P}[k]\, e^{j\frac{2\pi kn}{M}} \tag{IV.4}$$

We have to consider the conjugate-complex symmetry of the autocorrelation function: $\bar{p}^*[-n] = \bar{p}[n]$.

The final estimate of the mean PSD is obtained by windowing the estimate of the autocorrelation function:

$$P_{xx[k]} = \sum_{n=-M+1}^{M-1} w_a[n]\, r_{xx}[n]\, e^{-j\frac{2\pi kn}{2D-1}} \tag{IV.5}$$

where

$$w_a[n] = \frac{w_i[n]}{\dfrac{r_{rect}[n]}{r[0]}} \tag{IV.6}$$

where
$w_i[n]$: lag window function, ($w_i[n] = 0$ for $|n| \geq L$ where $L \geq M$)

$r_{rect}[n]$: autocorrelation function of the window used for the DFT of the data segments (here: autocorrelation function of the rectangular window)

IV.3 Spectrograms

If we are more interested in the time fluctuations of the velocity we can analyze the bursts with a time-dependent Fourier transform [25]:

$$X[n,k] = \sum_{m=0}^{M-1} x[m+n]\, w[m]\, e^{-j\frac{2\pi km}{M}} \tag{IV.6}$$

$w[m] = 0$ for $m < 0, m \geq M$
n: discrete time
k denotes the discrete frequency.

The process can be visualized as the data sliding behind a window of length M. A time-varying power spectrum results if the PSD of the windowed data is computed.

The use of a spectrogram implies that the signal is continuous, i.e. in terms of LDA, the particle arrival rate must be high. From the spectrogram, the frequencies of velocity fluctuations may be obtained by taking the Fourier transform of the time-varying velocity.

V. Adaptive Spectral Estimation

A great part of the spectral estimation algorithm used in this project relies on so-called adaptive methods. As they are less common than the classical methods, an introduction to the underlying principles is given in this chapter. Given these principles the performance of the algorithms in the numerical simulations and in the real experiment is easier understood.

In the first section the general terms in adaptive spectral estimation are presented. The auto-regressive and the auto-regressive moving-average models are described.

Then, auto-regressive spectral estimation and its relationship with linear prediction is discussed with some detail. The next section is concerned with the implementation of auto-regressive models on computers and the influence of noise on the spectral estimation. The Modified Covariance Algorithm, the auto-regressive method used in this project is also presented.

The poor performance of auto-regressive estimators in the presence of strong noise lead to the development of enhancement algorithms. In this project the Iterative Filtering Algorithm was tested.

In the last section, the theoretical foundations of a particular method for auto-regressive moving-average spectral estimation are presented. This method is based on Pade approximation of a high-order polynomial by the quotient of two lower order polynomials.

V.1 General Introduction

A stochastic process producing the sampled data $x[n]$ may be approximated by the output of linear time-invariant system driven by white noise. The linear time-invariant system is represented in the time domain by the linear constant-coefficient difference equation:

$$x[n] = -\sum_{k=1}^{p} a[k]\, x[n-k] + \sum_{k=0}^{q} b[k]\, u[n-k] \tag{V.1}$$

and in the frequency domain by the frequency response, or transfer function, $H(z)$:

$$H(z) = \frac{B(z)}{A(z)} = \frac{\sum_{k=0}^{q} b[k]\, z^{-k}}{1+\sum_{k=1}^{p} a[k]\, z^{-k}} \tag{V.2}$$

This formulation implies that $a[0]=1$.

An autoregressive moving-average model of order (p,q) for the discrete time series $x[n]$ is the minimum-phase system of Eq. (V.2) whose coefficients $a[k]$ and $b[k]$ have been determined such that the Eq. (V.1) is satisfied in a mean square sense for all data points $x[n], 0 \le n \le N-1$.

The first sum of Eq. (V.1), $\sum_{k=1}^{p} a[k]\, x[n-k]$, forms the autoregressive (AR) branch. Its z-transform, $A(z)$, is responsible for the poles in the system (zeroes of the denominator polynomial in z). The second sum, $\sum_{k=0}^{q} b[k]\, x[n-k]$, is termed the moving-average (MA) branch of the (p,q) ARMA model. Its z-transform, $B(z)$, is responsible for the zeroes in the system (zeroes of the numerator polynomial of z).

V.2 Autoregressive Spectral Estimation

We obtain a pure AR model of order p if we set $b[k]=0$ for all $k \neq 0$, i.e.

$$x[n] = -\sum_{k=1}^{p} a[k]\, x[n-k] + u[n] \tag{V.3}$$

Taking the modulus squared of the Fourier transform of Eq. (V.3) and noting that $u[n]$ is a white noise sequence with mean power σ_u^2 we get an expression for the power spectral density (PSD) of the AR model, P_{AR}:

$$P_{AR}(f) = \frac{T\sigma_u^2}{|A(f)|^2} \tag{V.4}$$

where T is the sampling interval.

Eq. (V.4) shows that an AR model can only represent peaks (poles) in the frequency response, i.e. $|A(f)|^2 = 0$.

The autocorrelation function can be determined from the linear constant-coefficient difference equation for the AR model

$$r_{xx}[n] = E\{x[n]\,x^*[n-m]\} = \begin{cases} -\sum_{k=1}^{p} a[k]\,r_{xx}[m-k] & \text{for } m>0 \\ -\sum_{k=1}^{p} a[k]\,r_{xx}[m-k] + \sigma_u^2 & \text{for } m=0 \\ r_{xx}^*[-m] & \text{for } m<0 \end{cases} \quad \text{(V.5)}$$

$E\{\}$ denotes the expected value of the quantity in brackets.

The autocorrelation function $r_{xx}[k]$ is related to the PSD for an AR process, P_{AR}, via:

$$P_{AR}(f) = \frac{T\,\sigma_u^2}{|A(f)|^2} = T \sum_{k=-\infty}^{+\infty} r_{xx}[k]\, e^{-j2\pi fkT} \quad \text{(V.6)}$$

Note that the autocorrelation $r_{xx}[k]$ for $0 \le k \le p$ alone describes the PSD. The implicit recursive extension of the autocorrelation sequence for $|k| \ge p$, $r_{xx}[m] = -\sum_{k=1}^{p} a[k]\, r_{xx}[m-k]$ is responsible for the superior resolution of the AR spectral estimators. The classical methods all assume the autocorrelation function to be zero outside the interval $[-p,p]$. Their data set for the Fourier transform is shorter, therefore their spectral resolution is lower.

An estimate for the AR parameters and thus an estimate of the PSD can be obtained with the following approach:

Consider the forward linear prediction equation:

$$\hat{x}_f[n] = -\sum_{k=1}^{p} a_f[k]\, x[n-k] \tag{V.7a}$$

where the sample at lag n is to be estimated based on knowledge of the p previous samples, $x[n-k]$, $1 \le k \le p$.

Similarly, the backward linear prediction equation:

$$\hat{x}_b[n] = -\sum_{k=1}^{p} a_f^*[k]\, x[n-k] \tag{V.7b}$$

which is anticausal: the sample at lag n is to be estimated based on knowledge of the p future samples.

Define also the respective errors in the prediction, the forward prediction error power:

$$e_f[n] = E\{|x[n]-\hat{x}_f[n]|^2\} \tag{V.8a}$$

And the backward prediction error power:

$$e_b[n] = E\{|x[n]-\hat{x}_b[n]|^2\} \tag{V.8b}$$

The linear prediction coefficients are chosen in such a way that they minimize the corresponding error powers. As we assume a stationary random process the linear prediction coefficients will be in addition time-invariant.

V.3 Algorithms for AR Spectral Estimation

During the course of the project two algorithms for estimating the PSD with AR models have been tested: the Modified Covariance Algorithm and the Iterative Filtering Algorithm. Before these two algorithms are described, some general remarks on AR spectral estimators are made.

V.3.1 Block Data or Sequential Data Algorithms

There are two different approaches to AR parameter estimation, block data algorithms and sequential data algorithms. Block data algorithms process an entire block of data at a time. Sequential data algorithms update the estimate as soon as a new sample becomes available. An example of a sequential data algorithm would be the fast Kalman filter.

Sequential algorithms seem to be less suited for frequency estimation of laser-Doppler signals than block data algorithms: Sampling rates of the order of 1 to 10 MHz do not allow continuous updating of the AR estimates, which requires approximately N computations per incoming sample (N being the number of previous samples on which the prediction is based). Also, the design of the *MASSCOMP* data acquisition system (one filled buffer from a buffer queue is released at a time) is best used by employing block data algorithms.

Among the block data algorithms, the Burg method and the covariance method are the most popular ones. Both rely on a minimization of the arithmetic mean of the forward *and* the backward prediction error power (Eq. V.10). As, according to the literature, the statistic properties of the modified covariance method (bias and variance in the estimated PSD) are more advantageous than the ones of the Burg algorithm, it was decided to run the tests exclusively with the former method. However, the modified covariance method does not necessarily generate a minimum-phase system, but fortunately in normal circumstances does so. All numerical simulations during this project, for instance, resulted in minimum phase systems.

V.3.2 Influence of Noise on AR parameter estimation

A common property of autoregressive spectral estimators is their susceptibility to additional observation noise.

Assume this additional observation noise to have zero mean and variance σ_o^2. Then from Eq. (V.5) we get (including the constant factor T - the sampling interval - in σ_u^2:

$$P_{AR+noise} = \frac{\sigma_u^2}{|A(z)|^2} + \sigma_o^2 = \frac{\sigma_u^2 + \sigma_o^2\,|A(z)|^2}{|A(z)|^2} \qquad \text{(V.9)}$$

Thus, even if the system has AR character, additional observation noise will add zeros to the frequency spectrum of the process which the all pole model cannot represent. The appropriate model for this case would be an ARMA model, for which only sub-optimal algorithms exist due to the non-linear least-squares equations.

The effect of observation noise on the AR PSD is a flattened (i.e. the variance is significantly decreased) and biased estimate for low signal-to-noise ratios (SNR). This disadvantage can be overcome to a certain degree by the iterative filtering algorithm for sinusoids in white noise.

V.3.3 Order Selection and Sinusoidal Parameter Estimation

For parameter spectral estimation the order of the chosen model is crucial for obtaining valid estimates. In general there is a resolution-variance trade-off for the spectral estimators when applied to noisy data. If the model order is chosen too low the resulting spectrum will not resolve all spectral peaks. On the other hand, too high a model order will cause spurious peaks to appear in the spectrum: The variance of the estimate increases. These spurious peaks appear because the AR estimator tries to model the noise zeros instead of the signal zeros. Here, the Iterative Filtering Algorithm reduces the variance for a given order and SNR. These properties will be illustrated in Section V.4.

AR estimators can be used for modeling the PSD of sinusoidal processes, if the available data $x[n]$ are stationary and the phase of the sinusoid is a random variable of uniform distribution. Then the PSD of such a process can be viewed as the limit of a band limited random process.

An appropriate model order for sinusoidal processes is $2M$ where M is the number of real sinusoids. However, the model order in actual simulations tends to be chosen somewhat larger.

V.3.4 The Modified Covariance Method

As mentioned above, the modified covariance method tries to minimize the arithmetic mean of the forward and the backward prediction error power (ρ_f and ρ_b):

$$\hat{\rho} = \frac{1}{2}(\rho_f + \rho_b) \rightarrow \min! \qquad \text{(V.10a)}$$

where

$$\rho_f = \frac{1}{N-p} \sum_{n=p}^{N-1} |e_f[n]|^2 \qquad \text{(V.10b)}$$

$$\rho_b = \frac{1}{N-p} \sum_{n=p}^{N-1} |e_b[n]|^2 \qquad \text{(V.10c)}$$

($e_f[n], e_b[n]$ were defined in eqns. (V.9a) and (V.9b))

Complex differentiation of Eq. (V.11a) with respect to $a[k]$ and setting the result to zero yields the following matrix equation for the parameters $a[k]$:

$$\mathbf{C_{xx}}\,\mathbf{a} = -\mathbf{c_{xx}} \qquad \text{(V.11a)}$$

where

$$\mathbf{C_{xx}}(j,k) = \frac{1}{2(N-p)} \left[\sum_{n=p}^{N-1} x^*[n-j]\,x[n-k] + \sum_{n=p}^{N-1} x[n+j]\,x^*[n+k] \right] \qquad \text{(V.11b)}$$

for $j,k = 1,\ldots,p$

and

$$\mathbf{a} = \begin{bmatrix} a[1] \\ a[2] \\ \cdots \\ a[p] \end{bmatrix}, \qquad \mathbf{c_{xx}} = \begin{bmatrix} c_{xx}[1,0] \\ c_{xx}[2,0] \\ \cdots \\ c_{xx}[p,0] \end{bmatrix} \qquad \text{(V.11c)}$$

The behavior of the modified covariance as described in the literature shows some very desirable properties especially for frequency estimation of sinusoids in white noise.

[19] provided a fast algorithm which takes $Np + 6p^2$ operations for an AR model of order p and a data segment length of N points. It relies on a special partition of the matrix $\mathbf{C}_{xx}$.

V.4 The Iterative Filtering Algorithm

This algorithm enhances the performance of AR estimation algorithms for low signal-to-noise ratios.

The PSD of an AR process plus observation noise as given by Eq. (V.10) is:

$$P_{AR+noise} = \frac{\sigma_u^2 + \sigma_o^2 \, |A(z)|^2}{|A(z)|^2} \tag{V.12}$$

At low SNR's $\sigma_u^2 \gg \sigma_o^2 \, |A(z)|^2$. Hence Eq. (V.13) can be approximated with:

$$P_{AR,SNRlow} \approx \frac{\sigma_u^2}{|A(z)|^2} = P_{AR} \tag{V.13}$$

The roots of the numerator polynomial $|B(z)|^2 \equiv \sigma_u^2 + \sigma_o^2 \, |A(z)|^2$ are of small magnitude (i.e. negligible) and located near the origin ($|B(z)|^2$ is nearly constant).

For high SNR's $\sigma_o^2 \gg \sigma_u^2$ Eq. (V.13) becomes:

$$P_{AR,SNRhigh} \approx \sigma_o^2 \tag{V.14}$$

Eq. (V.15) results from Eq. (V.13) if the roots of $|B(z)|^2=0$ are close to the roots of $|A(z)|^2=0$, i.e. if pole-zero cancellation takes place.

Furthermore, eqns. (V.14) and (V.15) demonstrate that with decreasing SNR the zeros in the noisy spectrum, $P_{AR+noise}$, approach the poles of the AR estimator. The spectrum flattens more and more.

If the data are prefiltered with $\frac{1}{|A(z)|^2}$, pole-zero cancellation can be avoided, as double-poles are created.

Because $|A(z)|^2$ itself is an unknown, we have to use an approximation of $|A(z)|^2$ to filter the data as follows

- Filter the data segment with the estimate $|\hat{A}(z)|_k^2$. Start the algorithm with $|A(z)|_k^2=1$.
- Determine the new estimate $|A(z)|_{k+1}^2=1$ based on the previously filtered data.
- Continue with the first step with $|A(z)|_k^2=\rightarrow|A(z)|_{k+1}^2$.

These steps define the Iterative Filtering Algorithm.

For the estimation of $|A(z)|^2$ any AR algorithm can be used. We decided to take the Modified Covariance Method because of its properties. The combination "Iterative Filtering Algorithm plus Modified Covariance Method" is not documented in the literature, as the latter produces not necessarily a stable filter. During all simulations, however,

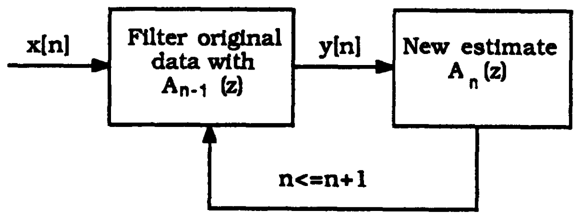


Figure 13. Iterative Filtering Algorithm

the poles fell into the unit circle.

The computational complexity of the IFA is relatively high: At each iterative step, the Modified Covariance Method is applied to the data $(Np + p^2)$ and the data segment is filtered with the current filter estimate $(N \log_2 N)$. Thus, assuming i iterations, the number of operations for each data segment of length N is approximately $i\ (Np+p^{2)(N}\ \log_2 N)$ which is for small p (note that p is around twice the number of real sinusoids in the signal) i^p^N sup 2 ~ log sub 2 N, the computational complexity of the Iterative Filtering Algorithm.

V.5 Autoregressive-Moving Average Spectral Estimation

ARMA spectral estimation results in nonlinear least-squares equations which are in practice not solved efficiently. Therefore, research has been directed towards so-called suboptimal algorithms resulting from linearization of the least-squares equations. In the following section, one of these algorithms, based on a Pade approximation, is described.

At the core of the method is the Euclidean algorithm for the division of two polynomials. The Euclidean is described in the first part of this chapter.

The second part defines the Pade approximant to a given polynomial as the quotient of two polynomials of lower degree: the Pade approximant and the original polynomial have the same remainder if divided by another polynomial. For Pade theory to hold, the degrees of all these polynomials must satisfy certain relationships.

It is then demonstrated that the Euclidean algorithm yields Pade approximants if the two initial polynomials are defined properly.

In the last section it is shown that Pade approximation is applicable to ARMA spectral estimation: we approximate the spectrum of our data (which can be represented with a polynomial using the z-transform) with the quotient of the AR and the MA branch (cf. Section V.1). This is exactly the problem treated in Pade theory. The final Pade estimator with the Euclidean algorithm is then presented.

V.5.1 Greatest Common Divisor of Polynomials: The Euclidean Algorithm

The Euclidean algorithm computes the polynomial of highest degree dividing two given polynomials. Thus, it yields the greatest common divisor (GCD) of two polynomials, which is determined up to a scalar factor. However, it is convenient to use monic polynomials (monic polynomials are normalized with their leading coefficient)[1][3][15].

The common form of the Euclidean algorithm for two input polynomials $A(z)$ and $B(z)$ with $deg\ [B(z)] > deg\ [A(z)]$ is given by the following series of recursive equations:

$$r_{-1}(z) = B(z) \tag{V.15a}$$

$$r_0(z) = A(z) \tag{V.15b}$$

$$r_{i-2}(z) = q_i(z) r_{i-1}(z) + r_i(z). \tag{V.15c}$$

Where $q_i(z)$ is the quotient polynomial of $\frac{r_{i-2}(z)}{r_{i-1}(z)}$ and $r_i(z)$ the remainder polynomial of this division. It is common to write the remainder of such a division as:

$$r_i(z) = r_{i-2}(z) \; mod \; (r_{i-1}(z)). \tag{V.16}$$

The Euclidean algorithm terminates if for a particular $i = i_0$: $r_{i_0}(z) = 0$, if the remainder becomes the null polynomial. The greatest common divisor is then equal to $r_{n-1}(z)$. A basic property of the Euclidean algorithm is that the remainder sequence $r_i(z)$ is of descending order:

$$deg \; [r_i(z)] < deg \; [r_{i-1}(z)].$$

Although the Euclidean algorithm is structurally very simple, its computational complexity of n^3 floating-point operations renders it quite inefficient for finding the GCD.

An efficient recursive doubling (divide-and-conquer) strategy has been developed, depending on the observation that not all coefficients of the two input polynomials contribute to the quotient polynomials [5][32]. The computational complexity of this fast version is $M(n) \log_2^2 n$ ($M(n)$ some linear function of n). This fast version of the Euclidean algorithm was also implemented and tested (cf. Section VI.2). It is, however, only of limited use in LDA.

V.5.2 A Glance at Pade Approximants

Pade theory deals with the approximation of an infinite-order polynomial by the ratio of two finite polynomials. For a general polynomial $G(z) = g_0 + g_1 z + g_2 z^2 + \cdots$, the (μ,ν) Pade approximant to $G(z)$ is defined as the quotient $\frac{B(z)}{A(z)}$ [21], where $B(z)$ and $A(z)$ are two polynomials of lowest degree satisfying the following equations:

$$deg\,[B(z)] \leq \mu \tag{V.17a}$$

$$deg\,[A(z)] \leq \nu \tag{V.17b}$$

$$\mu + \nu = N \tag{V.17c}$$

$$\frac{B(z)}{A(z)}\,(mod\,(x^{N+1})) \equiv G(z) \equiv G_N(z). \tag{V.17d}$$

$G_N(z) = g_0 + g_1 z + \cdots + g_N z^N$, $G_N(z)$ is the N^{th} truncation of $G(z)$.

Here " $\equiv$ " denotes congruence: The left-hand and the right-hand side in Eq. (V.17d) have the same remainder if divided by x^{N+1}.

V.5.3 Pade Approximation and the Euclidean Algorithm

The Euclidean Algorithm algorithm described in Section V.5.1 can be used to generate the Pade approximants to a polynomial if the initial polynomials are initialized in the proper way [21][27]. To apply the Euclidean algorithm to Pade theory we have to define a second sequence of polynomials, the so-called co-multiplier polynomial.

The co-multiplier sequence, $t_i(z)$, for the remainder sequence $r_i(z)$ and the quotient sequence $q_i(z)$ is defined as:

$$t_{-1} = 0; \quad t_0 = 1 \tag{V.18a}$$

$$t_i(z) = t_{i-2}(z) - q_i(z)\, t_{i-1}(z) \tag{V.18b}$$

then the (μ,ν) Pade approximant to $G_N(z)$ is $\frac{r_{i_0}(z)}{t_{i_0}(z)}$ for the iteration index i_0 which is uniquely defined by:

$$deg\ [r_{i_0-1}(z)] \geq \mu + 1 \tag{V.19a}$$

$$deg\ [r_{i_0}(z)] \leq \mu. \tag{V.19b}$$

r_i denotes the remainder polynomial of the Euclidean Algorithm as given by Eq. (V.15c).

The Euclidean Algorithm has to be initialized with $r_0(z) = G_N(z)$ and $r_{-1}(z) = x^{N+1}$.

V.5.4 Pade approximation and ARMA spectral estimation

The z-transform, or equivalently, the discrete Fourier transform (DFT), of our actual data sequence $x[n]$ of length N is defined by the following complex polynomial:

$$X(z) = \sum_{n=0}^{N-1} x[n]\, z^{-n} \tag{V.20}$$

(we obtain the DFT from (V.20) if we set $z \equiv e^{j\frac{2\pi kn}{N}}$, i.e. if we evaluate (V.###) on the unit circle).

The frequency response of the ARMA (p, q) filter was given by $\frac{B(z)}{A(z)}$. This frequency response - a quotient of two finite polynomials of orders p and q, respectively - should be the best approximation in the mean square sense to the actual spectrum of our data, $X(z)$. The problem becomes identical to the problem in the theory of Pade approximation.

Using the co-multiplier polynomial $t_i(z)$, as defined by Eqs. (V.18), we can apply the Euclidean algorithm to ARMA spectral estimation [21][27]. We set

$$r_{-1}(z) = z^{N+1} \tag{V.21a}$$

$$r_0(z) = x[n]\, z^{-n} \tag{V.21b}$$

$x[n]$ are the sampled data.

Then the (μ,ν) Pade approximant to $X(z)$ is the quotient $\frac{r_{i_0}(z)}{t_{i_0}(z)}$. The recursion index i_0 is hereby uniquely determined by:

$$deg\ [r_{i_0-1}(z)] \geq \mu + 1; \qquad deg\ [t_{i_0-1}(z)] < \nu;$$

$$deg\ [r_{i_0}(z)] < \mu; \qquad deg\ [t_{i_0}(z)] \geq \nu + 1.$$

If we employ the Euclidean algorithm in our spectral estimation we will obtain an ARMA (μ,ν) estimation for $X(z)$. The remainder polynomial, $r_i(z)$, (cf. Eq. (V.15)), is of decreasing order and the co-multiplier polynomial $t_i(z)$ is of increasing order, with $deg\ [t_i(z)] = 0$ for $i=0$. Therefore, we start the algorithm with a pure MA model, i.e. $X(z) \approx B(z) = r_1(z)$, and terminate with a pure AR model, $X(z) \approx \frac{1}{A(z)} = \frac{1}{t_n(z)}$. Thus, we can control the character of our spectral estimate, if we exit the Euclidean Algorithm at a given iteration index.

A discussion of the performance of the Pade spectral estimator for laser-Doppler signals is found in Section VII.2.3.

VI. Preliminary Processing of LDA Signals

The Doppler signal has to be preconditioned before spectral estimation with any of the methods presented in the previous chapter can be done. Prior to the A/D conversion high frequency components of more than half the sampling frequency have to be removed with an analog filter to avoid aliasing. Discussion of this standard procedure in digital signal processing is found throughout the literature.

Furthermore, it is advisable to remove the Gaussian pedestal and dc components with a (digital) lowpass filter. This gets rid of high energy contents in the low frequency region which carries no velocity information. An efficient way to implement this filtering is the overlap-add method, described below. This topic and the following glance at digital filter design are kept very short, as they again can be found in a variety of books on digital signal processing (eg. [24][25])

Two modes of operation of the laser-Doppler anemometer are: the *continuous* mode, where at each instant one or more particle are traversing the probe volume, resulting in a continuous signal at the photomultiplier; and the *burst* mode, where information about the flow velocity is only available at the random times at which a particle crosses the fringe pattern. Different sampling strategies arise for the two different types if the signal is instationary. Instationary Doppler signals result from eg. velocity gradients or turbulent flows.

VI.1 Digital Filtering with the Overlap-Add Method

Given a filter impulse response $h[n]$ with $h[n]$=0 for $n > (P-1)$ and $n < 0$ (the filter is said to be of order P), the filtering of an input $x[n]$ can be done in the time domain via a convolution:

$$y[n] = \sum_{k=0}^{P-1} h[k]\, x[n-k] \tag{VI.1}$$

It is straightforward to show that if the input $x[n]$ has length L the resulting filtered signal will have length $P+L-1$. Use of the convolution is only efficient for very short impulse responses or very short input data. The computational complexity of this approach goes like $P\,(P+L-1)$.

A very long signal can be divided into segments $x_i[n]$ of length L (very much like a periodogram) which are convolved separately (cf. Fig. 14). As the filtered segments will be of length $(P+L-1)$, $(P-1)$ points leak over into the next segment and have there to be added to the first $(P-1)$ points of the filtered data. This method is called the overlap-add method (see for example [25]).

The computational effort can be reduced if filtering is done in the Fourier domain. The convolution in the above equation is replaced by the product of the discrete Fourier transform of the filter impulse response (i.e. the filter frequency response), $H[k]$, and the Fourier transform of the data segment, $X[k]$.

$$Y[k] = H[k]\, X[k]$$

The computational complexity in this case is of the order $S \log_2 S$ for an FFT of length S.

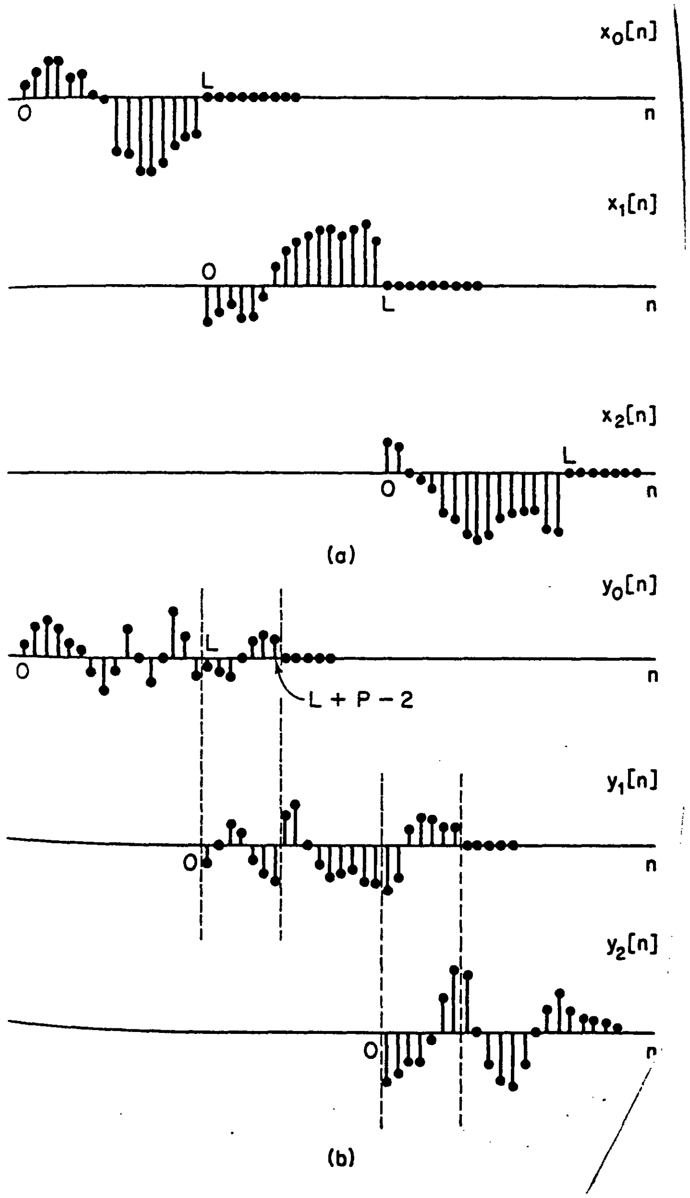


Figure 14. The overlap-add method (from [25]). (a) segmentation of the input data, (b) each segment is convolved with the filter impulse response and overlapped with the following segment

As the discrete Fourier transform implements a circular convolution (see [24][25]), the FFT has to be at least of length $(L + P - 1)$. One way to ensure proper filtering is to adjust the length of the data segments according to a desired length of the FFT, S, and the filter order $P: L = S + 1 - P$.

VI.2 Design of Finite-Impulse Response (FIR) Filters

An ideal low-pass filter with a cut-off frequency ω_c and a frequency response of $H(\omega) = 1$ for $-\omega_c \le \omega \le \omega_c$ and zero otherwise, has an infinitely long impulse response and is therefore in practice not available. The infinitely long impulse responses of these ideal filters are therefore approximated by finite impulse responses.

In the course of this project only two of the many ways to design an FIR filter were considered. One way, called the Kaiser window method, multiplies the impulse response of an ideal filter with a particular window function to approximate the desired filter. The resulting FIR filters are not the shortest ones possible for the given specifications, but the design procedure is fast and simple.

The most common design procedure, the Parks-McClellan algorithm (also termed optimum filter design), ensures the shortest possible impulse response for the given filter specifications. This routine is available as a FORTRAN program from IEEE [8].

The design of FIR filters is a very active field of research and detailed description of the underlying principles would go beyond the scope of this paper. For further reference I may recommend [24][25].

VI.3 Corrections for Velocity Gradients and Velocity Fluctuations

Most experiments in LDA are carried out in the presence of either velocity gradients or velocity fluctuations. Their effect on the signal properties is basically the same: the Doppler frequency will change with time and thus an nonstationarity is induced in the record. For the single burst, however, we can still assume wide-sense stationarity[3] provided that the Doppler frequency stays constant during one burst. Wide-sense stationarity is thus satisfied if the microscale of the flow is larger than the probe volume.

VI.3.1 Sampling Strategies

A continuous signal (there is always a particle present in the probe volume), may be analyzed with periodograms or correlograms [25]. The record is divided into segments of equal length, the final spectral estimate results from averaging the PSDs of the segments. Computationally most efficient in this class of spectral estimators is the Nuttall-Cramer method which is explained in Section IV.2.

A different situation arises for a discontinuous signal where velocity information exists only at the times where a particle traverses the probe volume: One batch of data consisting of (in time) randomly distributed bursts must be much longer than the integral scale of the flow. The bursts can then be thought of as random samples of the flow. The

3. Wide-sense stationarity requires the autocorrelation function to be time-independent, it must only depend on the time-lag.

periodograms are not applicable to this case as they yield a particle-averaged and not a time-averaged velocity. Therefore residence-time weighting has to be employed.

VI.3.2 Burst-type LDA: Residence-Time Weighting for Averages

A velocity gradient over the probe volume (or a time-varying velocity profile) will lead to a biased estimate in the burst-type LDA (towards higher velocities) if simple arithmetic averaging is done, i.e. if we compute the mean velocity via:

$$\bar{u} = \frac{\sum^{N} u_i}{N}$$

where N is the number of all bursts and u_i is the estimated velocity from the i^{th} burst. This bias is due to the fact that (assuming uniform particle distribution) more particles with higher velocity will cross the probe volume than particles with lower velocity. In this case the particle-average as expressed in the preceding equation will not equal the time-averaged velocity, as the arrival rate and the velocity are correlated. This situation is depicted in Fig. 15. In the continuous case, the higher arrival rate of faster particles is balanced by the longer signal duration of bursts from slower particles through the fixed-length processing.

An unbiased averaging process - the so-called residence-time averaging - is derived as follows [6][9]:

The mean velocity of the measurement is a result of both spatial and temporal averaging, i.e.

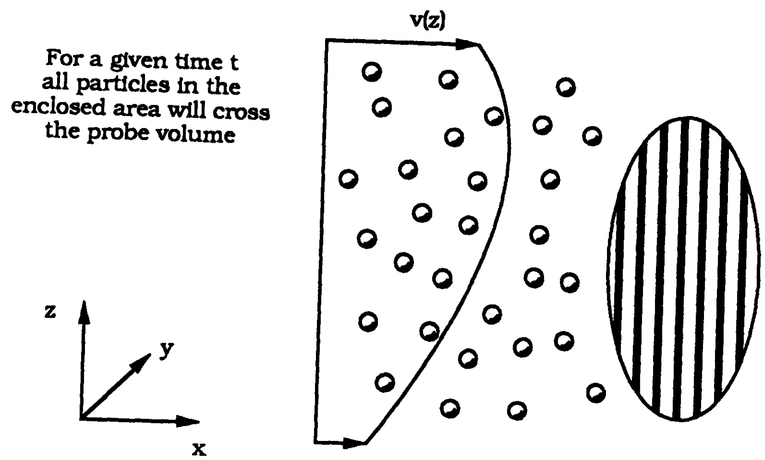


Figure 15. More particles with higher velocity will cross the probe volume than particles with lower velocity

$$\bar{u} = \frac{1}{VT} \int_0^T \int_V u(\mathbf{x}, t)\, dV\, dt \tag{VI.3}$$

V: size of measuring volume
T: duration of sampling

In order to assign the mean velocity to one point within the probe volume, we separate the velocity into the mean at a fixed point $\mathbf{x}_0$ within the measuring volume, $\bar{u}(\mathbf{x}_0)$, the difference of the velocity at a point $\mathbf{x}$ to that mean velocity, $\Delta\bar{u}(\mathbf{x})$, and a fluctuating part, $u'(\mathbf{x}, t)$:

$$u(\mathbf{x}, t) = \bar{u}(\mathbf{x}_0) + \Delta\bar{u}(\mathbf{x}) + u'(\mathbf{x}, t) \tag{VI.4}$$

Inserting this expression into Eq. (VI.1) yields:

$$\bar{u} = \frac{1}{VT} \int_0^T \int_V [\, \bar{u}(\mathbf{x}_0) + \Delta\bar{u}(\mathbf{x})\, d\mathbf{x} + u'(\mathbf{x},t)\,]\, dV\, dt$$

If we assume that the duration of one measurement is long enough for the fluctuations to cancel[4] then, after interchanging the order of integration, $\int u'\, dt$ will vanish and we are left with:

$$\bar{u} = \bar{u}(\mathbf{x}_0) + \frac{1}{V} \int_V \Delta\bar{u}(\mathbf{x})\, dV \qquad \text{(VI.5)}$$

For a certain choice of $\mathbf{x}_0$ the integral in Eq. (VI.5) will be zero. Clearly, this point can only be exactly computed if we have information about the mean velocity gradient.

The form of the recorded signal, however, gives no information about the the location of the transition path of the particle. Therefore we can only employ temporal averaging to reconstruct the theoretical mean velocity:

$$\bar{u}_{LDA} = \frac{1}{T} \int_0^T u(t)\, dt.$$

which becomes the *residence-time weighted sum*

4. This requires also that the presence of a particle within the probe volume is not correlated in any way with the velocity fluctuations.

$$\bar{u}_{LDA} = \frac{\sum^{N} u_i \, \Delta t_i}{\sum^{N} \Delta t_i} \tag{VI.6}$$

With the following substitutions:
$u(t) \to u_i$: velocity estimate of the i^{th} burst
$T \to \sum \Delta t_i$: duration of the measurements
Δt_i: duration of the i^{th} measurement (burst)
N: number of measurements (bursts).

Eq. (VI.6) (and therefore residence-time weighting) makes sense only in sparsely seeded flows, where a Δt_i is well defined. In practice, exact determination of the residence time is hard to accomplish. It is recommended that Δt_i is taken as twice the time from the signal maximum to half this value [9].

In order for $\bar{u}(\mathrm{x}_0) = \bar{u}_{LDA}$ to hold exactly, $\int \Delta\bar{u} \, dV \equiv 0$ must be satisfied. Obviously, this is the case for a uniform mean velocity or a antisymmetric velocity difference $\Delta u(\mathrm{x})$ in the mean around the point x_0. In the case of linear mean velocity gradient x_0 is the location of the center of the probe volume.

In general, however, we have to keep the integral in Eq. (VI.5) as correction term:

$$\bar{u}(\mathrm{x}_0) = \bar{u}_{LDA} - \frac{1}{V} \int_V \Delta\bar{u}(\mathrm{x}) \, dV \, . \tag{VI.7}$$

where $\bar{u}_{LDA}$ is obtained by residence-time weighting the data record.

VII. Numerical Simulations with the MATLAB Software Package

Using the MATLAB™ (The MathWorks) software package, power spectrum estimates based on the direct Fourier transform of windowed data segments, the Modified Covariance Algorithm, the iterative filtering algorithm, and the Pade spectral estimator were tested on a typical laser-Doppler signal with additive white noise. Computer simulations on the MATLAB package allow a fast adaptation of different algorithms due to the object-oriented programming language and the large library of mathematical and signal-processing routines. In addition, as the algorithms can be written in vector notation, transfer of the programs to the *MASSCOMP* array processor is greatly facilitated.

The numerical simulations can be divided into two separate tasks: The generation of signals and the application of the spectral estimators to these signals.

The purpose of these preliminary tests was twofold: The performance of the different methods (and their robustness from the programming point of view) in the numerical simulations allowed to decide whether their implementation on an array processor will be reasonable. Secondly, comparing their performance with artificial signals and signals arising in a real experiment should yield some information about the predictive value of the models for the artificial signals.

VII.1 Signal Generation

The signal generator, a MATLAB script-file, (`mksig.m`) models as close as possible the signal encountered in a "real" laser-Doppler experiment to predict more easily the performance of the frequency estimators. Therefore, the produced signal can represent different experimental set-ups (different seeding densities, velocities, velocity gradients, SNRs etc.).

The data for the optical set-up are based on the DISA 55X Modular LDA Optics with a X51 160 *mm* front lens and a 124B laser type [40].

In the literature, computer generated laser-Doppler signals usually consist of one single burst of well defined signal-to-noise ratio. In my point of view this may not be an appropriate testing procedure. First of all, the local SNR of a laser-Doppler signal varies as different particles cross the probe volume at different paths. This results in signals with different signal intensities in front of the constant background noise.

The following assumptions were made for the signal:

- The photocurrent and the intensity of the scattered light are linearly related: An ideal photodetector is modeled. The three-dimensional intensity distribution within the probe volume is known.

- The particles are assumed to be point-sized. The Doppler-signal of a real particle will be more smoothed out.

- The velocity has just one component perpendicular to the fringe system. There are no directional fluctuations of the velocity.

- The velocity of the particles stays constant within the probe volume. I.e. in unsteady flows the Kolmogoroff microscale must be larger than the probe volume.

The MATLAB routines for the signal generation listed in the Appendix, work as follows:

After initialization of all parameters (lines `10-70`), the time of occurrence (line `76`) y-z-coordinates of the transition path of a particle are randomly chosen by the program (function `transit`). The mean pause between the particle crossings can be adjusted to simulate different seeding densities of the flow (line `21`, variable `meapaus`). Thus the created signal may either simulate a burst-type or a continuous LDA signal.

The signal generator stops if a predefined number of bursts has been created (loop over lines `73-91`). Next, white noise is added to the signal (lines `97-115`). In order to achieve a specified signal-to-noise ratio, the variance of the noise is adjusted in the following manner (lines `105, 107`).

From the definition of the signal-to-noise ratio:

$$SNR = 20 \log_{10} \frac{var\ (signal)}{var\ (noise)} \qquad \text{(VII.1)}$$

(var ( ) stands for the variance of the quantity in parentheses). We can adjust the amplitude of the noise by

$$amp = \left[\frac{var\ (signal)\ 10^{\frac{-SNR}{10}}}{var\,(noise)} \right]^{\frac{1}{2}} \qquad \text{(VII.2)}$$

where $var\ (amp\ noise) = amp^2\ var\ (noise)$ has been used to obtain the desired SNR.

In the last step, the signal is quantized to 12 bits simulating an optimally adjusted A/D converter.

As mentioned above, the signal generation facility exceeds in its complexity the ones described in the literature, where usually *one single* high-pass filtered burst with additive white noise is used as a test signal. For the evaluation especially of autoregressive frequency estimators, a "real" signal can yield totally different results:

Different envelopes and pedestals in the signal which are not completely removed by high-pass filtering may influence the performance of the algorithms. Employing a sliding time window will also result in different local SNRs as the amplitude of the sinusoid varies in front of noisy background. In addition, several particles crossing the probe volume with different velocities will cause beating effects in their Doppler frequencies. Therefore, the behavior of for example an AR estimator at a given order cannot be predicted offhand.

The presented signal generator is easily extended to include simulation of a velocity gradients of arbitrary shape: After the location of the transition of a particle is determined another function may be called where the velocity u is changed accordingly.

VII.2 Testing of the Algorithms Using the MATLAB software

The algorithms are tested on signals with four different SNR's: 2000 dB, 10 dB, 0 dB, and -10 dB. The 2000 dB signal was used to determine the order and the window length of the estimator which best represents the known Doppler frequency. All DFTs used in the simulations were 512 points long. Before computing the PSD the actual data segment is high-pass filtered using an IIR Butterworth filter with the following specifications:

stopband edge frequency	20000 Hz
deviation from unity in stopband	0.001
passband edge frequency	60000 Hz
deviation from zero in stopband	0.001

TABLE 3. Filter specifications for Butterworth filter

Filtering is done in the frequency domain: The discrete Fourier transform (DFT) of the current data segment is multiplied with the frequency response of the filter and then transformed to the time domain. Filtering is also a prerequisite of the AR estimators as they require data with zero mean i.e. with the dc-component removed. The filtering step has not been carefully implemented, as the DFT circularly convolves the signal with the infinitely long filter impulse response. However, the results seem to indicate that this error had no effect on the spectral estimation process.

In most applications of laser-Doppler anemometry the frequency of the signal will change with time due to the flow properties (turbulence, oscillatory flows, velocity gradients over the probe volume). The most appropriate way to show the time-dependence of the spectra of non-stationary signals is to plot the PSD over both time and frequency in a spectrogram. The results are 3-dimensional graphs, where the "height" is the PSD estimate at that particular time-frequency point.

VII.2.1 Description of the Generated Signal

Table 4 lists the data set used for the simulations. Examples of actual signals produced by the MATLAB-routines are shown in the Appendix.

number of bursts	20
sampling rate f_s	$1\,MHz$
velocity (x-component only), $\mathbf{u} = u_x$	$1\,\frac{m}{s}$
mean pause between burst	$0.0001\,s$
focal length of front lens, f	$300\,mm$
wavelength of laser, λ	$633\,nm$
angle of intersection, θ	$7.44°$
waist diameter of unfocused beam, d_w	$1.1\,mm$
Doppler frequency, f_d	$204{,}99\,kHz$
half axis of ellipsoid, d_x, d_y, d_z	$0.1101\,mm$, $0.1099\,mm$, $1.6939\,mm$

TABLE 4. Data set for the numerical simulations

The value of the parameter determining the seeding, `meapaus` was set to 0.0001. This resulted in an almost continuous signal. Fig. 26 in the Appendix shows 20 bursts with a practically infinite signal-to-noise ratio. This signal was used as test input in all subsequent tests. Only the amplitude of the additive white noise was changed to obtain the desired SNR. Fig. 27 shows the same signal with a SNR of 0 dB, Fig. 28 shows the local variation of the SNR of the signal in Fig. 27. The SNR varies depending on the strength of the bursts in front of the uniform noise.

The Doppler frequency of 205 kHz corresponds to bin 105 in the DFT.

VII.2.2 Results of the DFT-based Spectral Estimator with a Hamming Window

The behavior of the classical method is as expected: The peak in the spectrum corresponding to the Doppler frequency is clearly visible. Fig. 29 shows that the method is sensitive to the local SNR in the signal: at the location of lowest SNR in the signal, the method failed (Segment numbers 1,3,5,19,28). Also the Doppler frequency is not resolved over parts of the spectrogram.

VII.2.3 Results of the Modified Covariance Algorithm (AR Method)

First, the appropriate order of the AR model for an LDA signal was chosen at a practically noiseless (2000 dB SNR) signal (cf. Appendix). Best results with the least computational effort were obtained with model orders 3 and 4. Both correspond to a model order used for AR representation of Doppler-radar signals [29]. The results for the third order Modified Covariance Algorithm are shown in Figs. 31 to 34.

The behavior of the Modified Covariance Algorithm applied to laser-Doppler signals corresponds to the general behavior of AR estimators. The resolution-variance trade-off is obvious if Figs. 31 and 35 are compared: At order 3 the spectrum is flat and has no spurious peaks but the spectral resolution is lower because of the low model order; at order 20 the variance in the spectrum is increased by the presence of spurious peaks. The modified covariance shows some bias for lower order models at SNR's of 10 dB and 0 dB respectively.

A look at the local SNR's leads to the conclusion that the modified covariance method seems to work only for SNR's of more than 0 dB to 10 dB.

VII.2.4 Results of the Iterative Filtering Algorithm

The fluctuations in the main spectral line were much smaller with Iterative Filtering Algorithm throughout the tested SNR's. In fact, the IFA turned out to have the lowest variance in the PSD of all tested methods (Fig. 36). However this technique fails also at very low SNRs (Fig. 37).

The simulations were carried out at 10 dB and 0 dB SNR. The results indicate that the IFA indeed represents an improvement of the Modified Covariance Method at low SNRs, however with considerably higher computing effort.

VII.2.5 Results of the Pade Spectral Estimator

The preliminary MATLAB tests were first carried out with the fast recursive Euclidean algorithm, along the line of [27]. There exist different versions of this algorithm in the literature, [1][4][27][32], however the formulations in [4] and [32] did not work. Therefore, the algorithms as shown in [1] was used.

In general, the adoption of this recursive process is quite cumbersome for the relatively high-order polynomials occurring in the present case (64-point or 128-point data segments lead to 64th or 128th order polynomials). The leading coefficients of the polynomials have to be constantly checked whether they in fact represent floating-point zeros Also, the stack operations for the recursive calls of the routine become significant. The last version computed the test data spectrum provided in [14]. Some data segments in an arbitrary Doppler signal, however, let the program terminate with an error.

The recursive Euclidean algorithm will return an AR branch whose degree is approximately half the degree of the input denominator polynomial. I.e. if our input polynomials were $\frac{S[z]}{T[z]}$ with $deg\ T[z] = N$, for N input data points, then after one call to the recursive routine the ARMA estimator $\frac{B[z]}{A[z]}$ will have an AR branch of $deg\,[A(z)] = \frac{N}{2}$.

From the discussion of the AR estimators we learned that too high an AR order of the Pade spectral estimator will result in spurious peaks. For our purpose we may therefore conclude that the Pade spectral estimator using the fast recursive Euclidean algorithm is not appropriate for LDA. Also the highly recursive structure is not easily implemented on an array processor with limited memory capacities. Therefore, the idea of using the fast version of the Pade estimator was abandoned.

We obtain good results, however, if we employ the common Euclidean Algorithm (eqns. (IV.16)) and exit if $deg\ [t_i(z)]$, the order of the AR branch, exceeds a preset order. This order will eventually be low, approximately twice the number of real sinusoids in the signal. As the n^3 Euclidean Algorithm is executed only for the first few i it loses much of its computational complexity.

The results of the Pade estimator employing the Common Euclidean algorithm appear to be comparable with the IFA if not superior (Figs. 38 to 41). The variance in the spectral estimator may be higher but as seen in Figs. 39 and 41 the peak in the spectrum really corresponds to the Doppler frequency. It also yields the correct results at the locations of low local SNR where the DFT estimator failed. Considering also its computational complexity this method seems to be highly recommendable.

VIII. A Software System for Processing LDA Signals

Based on the results of the MATLAB simulations, it was decided to compare the performances of the Pade ARMA spectral estimator to the performance of the classical approach in a real flow experiment (cone-and-plate flow). For this, it was necessary to implement a system of programs capable of analyzing LDA signals.

The first section of this chapter describes the available computer resources with which the LDA signals are to be processed. As most programs make extensive use of the high computational power of an off-board pipelined array processor, some of its properties are mentioned. The programs, listed in the Appendix, are well commented and go along the lines of the underlying theories introduced in the foregoing chapters. Therefore, more emphasis is put on how the programs interact than on a minute description of their operations.

VIII.1 Available Computational Resources

The programs run on two different UNIX machines, an older *MASSCOMP* 550 machine and a new *CONCURRENT* 6400. The *MASSCOMP* is used mainly for sampling the data as it is equipped with a 12-bit 1 MHz A/D converter. All computations are done on the faster *CONCURRENT* which features also an off-board vector accelerator rated at 14.25 MFlops [38].

The vector accelerator operates independently from its host CPU. It consists of two units, a DMA unit responsible for the data transfer from host memory to the vector memory (32768 locations of 32 bit floats) and back, and a math unit doing all the number crunching (in single floating point accuracy) on the vector memory. The action of the two independent units has to be synchronized to prevent transfer of data by the DMA while the math unit is still working on them.

Unfortunately, the vector memory is not accessible from a debugger, which makes programming somewhat cumbersome. Two C language macros, `DUMP` and `MAGC`, have proven to be very useful for program development. They synchronize math and DMA unit, transfer a specified vector to the host, print the contents (`DUMP`) or the complex magnitude squared (`MAGC`), and exit.

VIII.2 Descriptions of the Programs

The system of programs for LDA signal processing consists of eight independent C routines (`SampleData`, `FilterData`, `Variance`, `GetBursts`, `MeanSpec`, `DoPlot`, `CreateFIR`, and `MakeWindow`) communicating via data files. A Bourne shell script (`MasterPlan`) allows the setting of the most relevant parameters and executes the programs in the proper order from the *MASSCOMP* over the Ethernet. The structure of the system is depicted in Fig. 16.

VIII.2.1 `SampleData` - Program for Data Acquisition

The data are sampled by `SampleData` at the *MASSCOMP* and then copied over the Ethernet to the *CONCURRENT*. The user can - as with the rest of the programs - specify all filenames, as well as the sampling frequency and the duration of the sampling

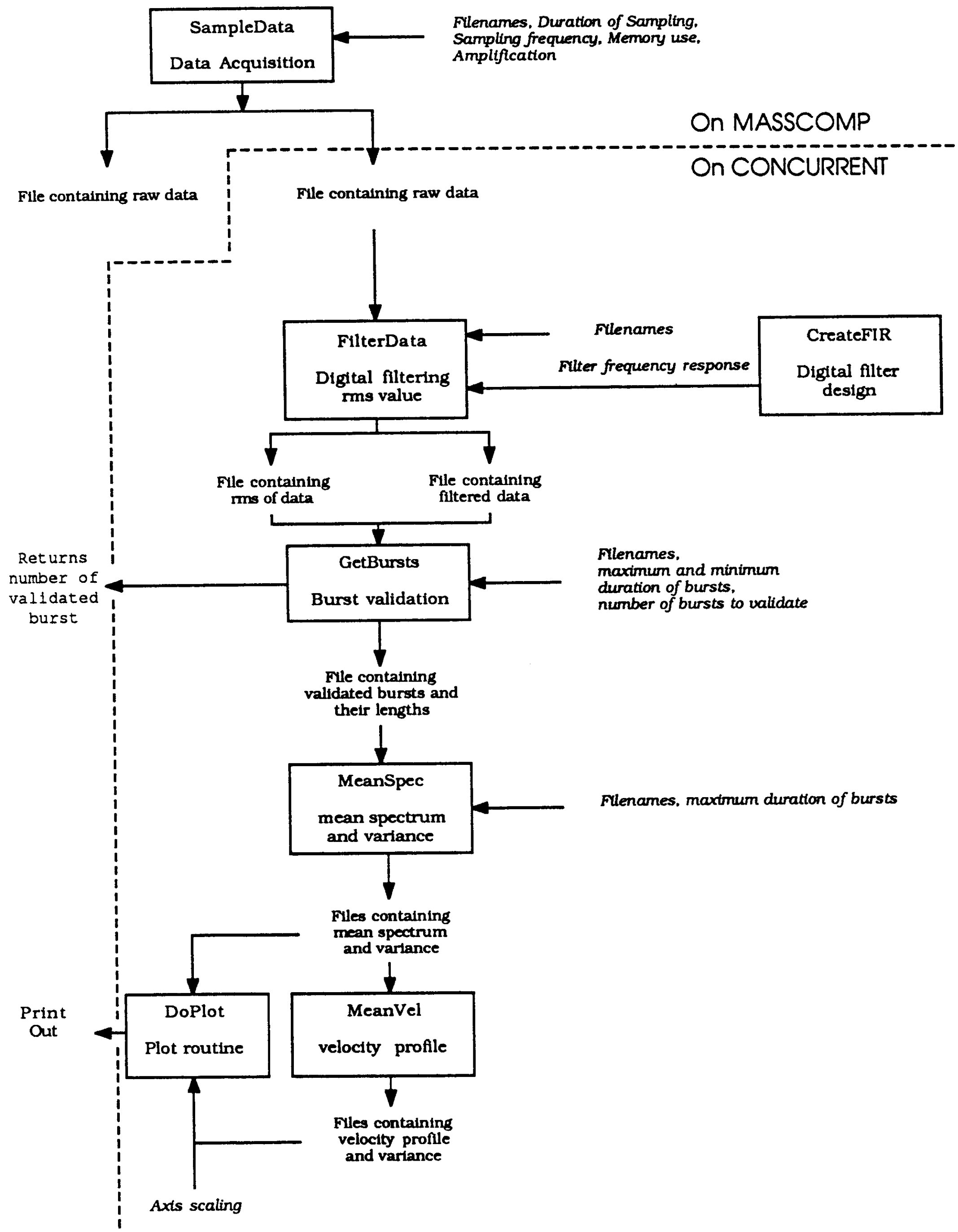


Figure 16. System structure

process. Presently, only 3 MB of memory are available at the *MASSCOMP*, so that longer sampling durations had to be implemented by repeatedly setting up the A/D board, sampling, and writing to disk (Loop over lines `293-341` and `346-375`)[36].

The data record is then transferred to the *CONCURRENT* host, with a system call to `rcp` (lines `382-388`).

VIII.2.2 `FilterData` - Program for Filtering Input Data

`FilterData` filters the raw data (still in integer format) with a digital FIR filter (eg. designed by the routine `CreateFIR`) using the overlap-add method (cf. Section VI.1).

VIII.2.3 `CreateFIR` - Digital Filter Design Routine

`CreateFIR` consists of three routines (`CreateFIR`, `KaiserFIR`, and `OptFIR`). The function `KaiserFIR` designs a FIR filter with the Kaiser window method. `OptFIR` is the IEEE routine for the optimum filter method slightly modified to serve as a subroutine. `CreateFIR` writes the frequency response of the filter together with the filter order and the length of the DFT on file.

VIII.2.4 `Variance` - Program Computing First Order Statistics of the Signal

`Variance` computes the mean, the rms value, and the standard deviation of the data. The `-S` option must be used if the first order statistics of the unfiltered data are computed. The data are then first converted to float format which is usually first done by

`FilterData`. If `GetBursts` (see below) is run with the `-M` option, `Variance` has to be used for `int` to `float` conversion (`-CS` options).

VIII.2.5 **`GetBursts`** - Burst Validation Algorithm

The filtered data are now screened for particle bursts by the routine `GetBursts`. This program basically implements a DISA burst validation circuit [41]. Fig. 17 shows the flowchart of the routine.

The first part of the program performs all initialization tasks: Command line parsing (lines `188-252`), memory allocation, opening of all files (lines `282-338`), and setting the real-time priority of the process (lines `274-280`). The flags, which correspond to the output of the Schmitt-Triggers in the DISA circuit, are initialized (lines `132,133`), the number of bursts processed up to now is read from file, if present, or set to zero (lines `311-338`), the number of samples in the source file is read as the first item in the source file (line `347`).

The trigger levels can be either set directly with the `-M` option or as multiples of a threshold level. The reference trigger level in the file given under the `-t` option, may for example come from the routine `Variance` which computes the first order statistics.

The burst detection algorithm itself is embedded in a loop which exits if all items in the source file are read (lines `410-587`) or if the number of bursts specified under command line option `-b` is reached (lines `486-494`).

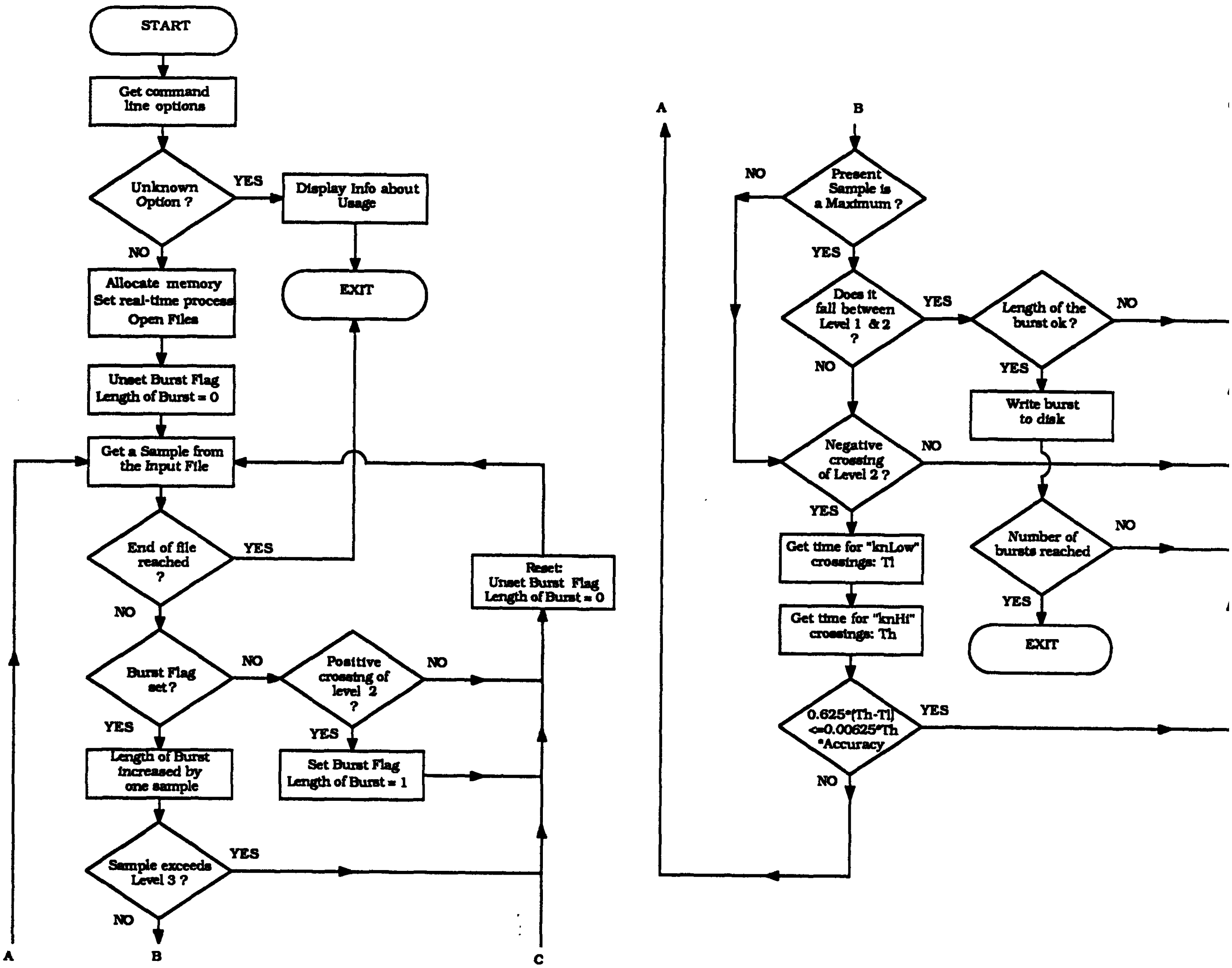


Figure 17. Flowchart of the burst validation algorithm

A crossing of trigger level 2 causes the burst flag to be set and the position of the triggering sample in the file to be saved (lines `572-580`). If a crossing of trigger 2 occurred, i.e. the burst flag has been set, the burst is terminated if a maximum falls between trigger levels 1 and 2 (lines `443-519`).

Rejection of a burst occurs because the value of the sample was too large (this usually is the case if the scattering particle was too large, lines `424-430`), the isolated burst was either too long or too short (line `453`), or if a criterion involving the ratio of the times needed for `knLow` and `knHi` (defaults are `5` and `8`, respectively) crossings of trigger level 2 (line `553`) is violated.

Once a burst is validated, it is written to file, the first item written to file being its length.

The program returns to the calling environment the number of burst it has collected since the last call with the `-N` option which deletes the file containing the current burst count.

In the present form it seems to be straightforward to implement a counter by using either the crossing rate at one of the trigger levels (for example computing `nTimeHi/nTimeLo`) or by using the zero crossing rate. Unfortunately, time did not permit any further experiments in this direction. For the relationship between the zero crossing rate and the frequency see for example [23].

VIII.2.6 **MeanSpec** - Program for Computing the Mean Spectrum of the Bursts

Once the bursts in a record are validated `MeanSpec` obtains the mean spectrum by using either the classical method or the Pade approximation (option `-m`). Averaging is done with residence-time weighting (cf. Sec. VI.3)(lines `579-580`). The `-DDEBUG` compiler option computes the mean spectrum and the variance without the use of the vector accelerator.

The flowchart of the Pade spectral estimator based on the Euclidean algorithm is shown in Fig. 18. The algorithm consists of six routines. `PadeApprox` initializes the polynomials according to Eq. (V.21). A vectorized version of the Euclidean algorithm is implemented by `EucAlgVA`. It calls routines for multiplication of polynomials, `ConvolveVA`, for polynomial division, `PolyDivVA`, and for removal of zero leading coefficients, `CheckOrderVA`. The estimate of the power spectrum, the quotient of remainder and co-multiplier polynomial (cf. Section V.5.4), is finally computed by `ArmaPsd`.

`CheckOrderVA` is important to adjusts the length of the polynomial if leading coefficients represent floating point zeroes. The polynomial is normalized with the coefficient of the highest absolute value. All leading coefficients smaller than a certain parameter `FLT_EPSILON` (ε in the flow chart) are removed from the polynomial. The function `PolyDivVA` is a vectorized version of the program for polynomial division in [28].

As discussed in Section V.5., the spectral estimate of the Pade technique is the quotient of remainder and co-multiplier polynomial if the desired AR branch order (i.e.

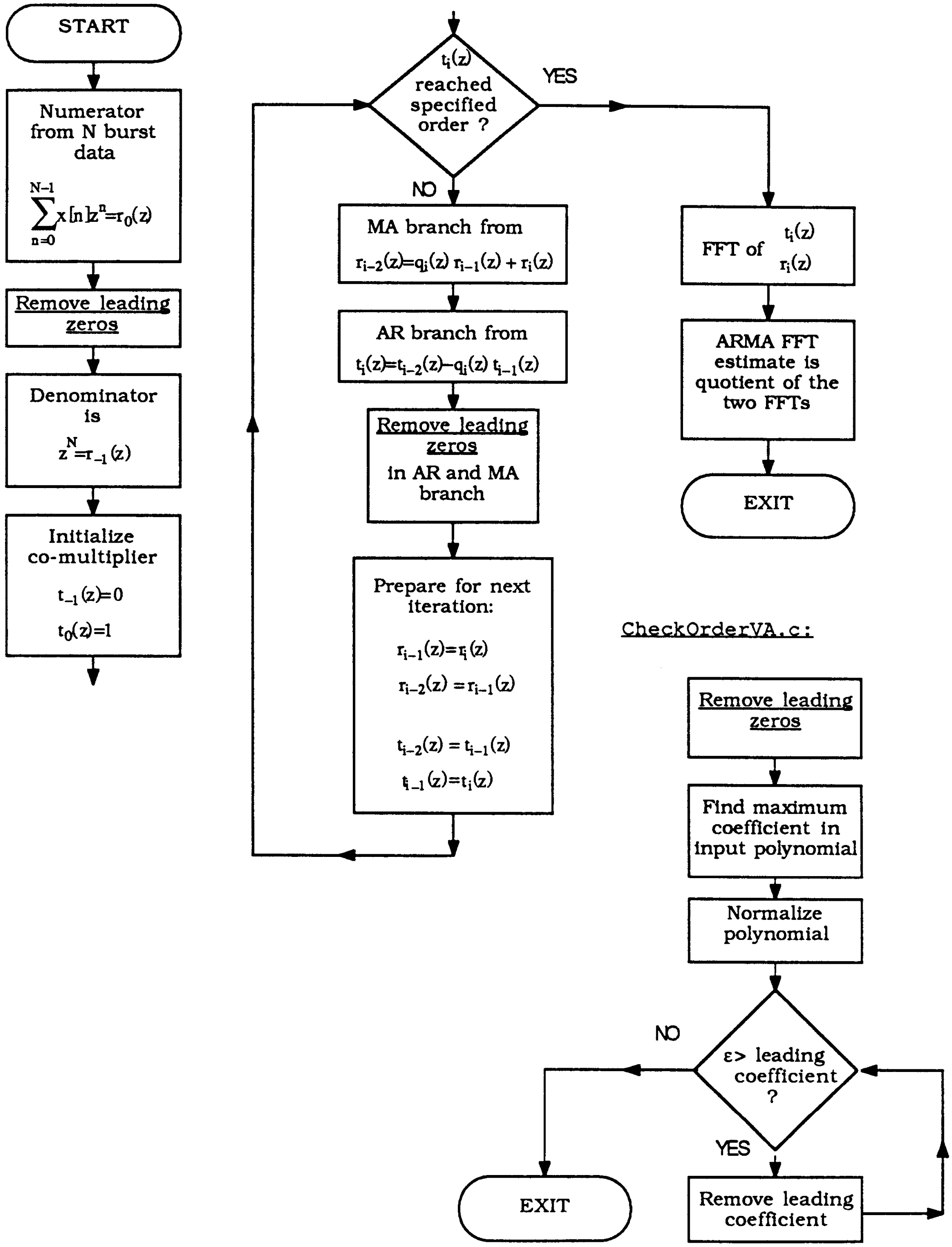


Figure 18. Flowchart of the Pade spectral estimator

the order of the co-multiplier) has been reached.

VIII.2.7 `MeanVel` - Program for Compiling the Velocity Profile

The incentive for `MeanVel` is to automatically obtain a velocity profile by computing the Doppler frequency from the mean spectrum as obtained by `MeanSpec`: First, the bin of the discrete spectrum is found where the mean over a region of the spectrum of length `nWindowLen` reaches a maximum (lines `328-339`). Then, the first and second moment with respect to the dc value (bin 0) of this region is computed (lines `352-357`). The first moment is the estimate of the Doppler frequency, the second moment is an estimate of the variance in this estimate. The `-C` option allows a calibration factor to be specified for the conversion from $[Hz]$ to $[\frac{m}{s}]$. The *-B* option may be used for passing the Bragg cell frequency shift to the routine, so that even in the case of opto-electronic frequency shifting a correctly scaled velocity profile may be obtained.

VIII.2.8 `DoPlot` - Plot Routine

`DoPlot`, the plot routine is capable of displaying both the output from `MeanSpec` and `MeanVel`. Axis scaling and title of the plot can be specified via a command line option. It uses *MASSCOMP/CONCURRENT* graphics system calls [37].

VIII.2.9 `MasterPlan` - Shell Script

`MasterPlan`, a Bourne shell script, is the actual file to be called when running the programs described above: For a given number of measurement positions (specified under the `-n` command line option) the routines are executed as shown in Fig. 16 until the required number of bursts (`-b` option) has been collected at each particular measurement position (line `182-235`. The script then stops and prompts the user for further input (lines `244-277`). At this point, the user may change some parameters (lines `389-454`), discard the present spectrum and repeat the measurement (line `302-313`), go to the next measurement position, (lines `279-299`), or plot the data obtained so far (lines `317-379`). The graphs can all be saved in PostScript.

To keep it more transparent, `MasterPlan` uses only a small selection of all possible options of the single routines. Instead, it makes use of the default values which are consistent throughout the programs.

VIII.3 Caveats of the Programs and Some Hardware Recommendations

- `CreateFIR` does not allow for setting the filter specifications at the command line. This is due to different formats for parameter passing in the functions `KaiserFIR` and `OptFIR`. I leave it to my successor, should there be any, to think of a uniform format. I'd say it's worth the effort, both design procedures work nicely.

- `FilterData` works best if strong low frequency components whose period is much larger than the filtering segment length, `nSegLen`, are removed. Also, results were improved by manually setting the dc coefficient in the filter frequency response to zero. Experiments with a gated linear sweep signal (the frequency of the signal increases linearly with time) showed that without these precautions the portions of the signal of low frequency and the discontinuities were distorted.

- Comparison of the Pade routines at the vector accelerator and MATLAB for some test polynomials shows that after the fourth iteration in the Euclidean algorithm accumulation of round-off errors prevent the routine `CheckOrderVA` from eliminating leading coefficients which should be "zero". For the routine to work properly, the value `FLT_EPSILON` would need to be something like 50, i.e. a leading coefficient of the polynomial is considered as a zero if it is only a $\frac{1}{50}$ of the maximum coefficient. This seems terribly inaccurate to me. I kept the value of `FLT_EPSILON` as it is, as everything works fine for an AR order of two, or three.

- In order to set the trigger in the routine `GetBursts` correctly, some fine tuning is still necessary. The trigger levels were determined after one test run: After the first-order statistics of the signal were determined, the data record (or parts of it) was displayed on the computer and the approximate trigger levels were determined by looking at a typical burst in the record. If the trigger levels are entered as absolute values (`-M` option) their values can be estimated from an oscilloscope.

- If the number of bursts validated by `GetBursts` is 255, it will be mixed up with exit status -1 and the shell function `ErrorCheck` will signal an error and exit. The exit status of a program is a byte integer (unsigned) thus `GetBurst` cannot return a value higher than 254 to the calling environment. One way to avoid this is to make `GetBurst` return the number of bursts it just validated, but then this number may in turn not exceed 254.

- `GetBursts` locks the whole data record into physical memory. This ensures maximum speed but limits the length of the data records.

- At some occasions the vector accelerator `MeanSpec` signaled some error if only one burst was validated by `GetBursts`. In the given time this bug could not be fixed.

- All routines using the vector accelerator (`CreateFIR`, `FilterData`, `Variance`, and `MeanSpec`) need an error trapping routine which preserves all the data up to the error and exit with grace. At the moment any error in the vector accelerator simply causes the routines to continue with the false data.

- The CPU would have been spared of much computational burden if only the A/D converter would allow more programming. Some time of the project was spent going through the microcode of the data acquisition processor. the ultimate goal was to place part of the routine `GetBursts` right there: Sampling should occur only if the signal exceeded a specified threshold and stop if it falls between two other thresholds. However, the instruction cycles of the processor would permit addition of commands only with lower sampling frequencies. Hopefully, at some point, an equally fast (or faster) A/D converter would allow some tapering in his microcode for conditional sampling.

IX. Application of Software System for LDA to Cone-and-Plate Flow

Experiments with a cone-and-plate apparatus were performed to demonstrate the viability of the software system. The properties of the cone-and-plate flow are presented in the first section. The second section describes the experimental set-up, i.e. the optical configuration of the LDA system, the flow apparatus, and the parameters of the flow.

IX.1 The Cone-and-Plate Flow

Similarity analysis of the Navier-Stokes Equation in cylindrical coordinates, assuming radial symmetry and a very small cone angle α, yields the local parameter:

$$\tilde{R} = \frac{r_r^2 \omega \alpha^2}{12\nu} \qquad \text{(IX.1)}$$

r_r: radial position of fluid element from apex
ω: angular velocity of cone
ν: kinematic viscosity of fluid
α: cone angle in *rad*

$\tilde{R}$ may be interpreted as the ratio of centrifugal to viscous forces acting on a fluid element. For $\tilde{R} \ll 1$ centrifugal forces are negligible, the velocity profile in azimuthal direction is essentially that of a plane Couette flow.

Secondary flow will not be present for $\tilde{R} < 0.0625$. Secondary flow becomes significant for $\tilde{R} \approx 1$ and is directed radially outwards at the upper half of the gap and inwards at the bottom half. Transition to turbulence occurs for $\tilde{R} \approx 4$.

In primary flow the velocity gradient is independent of the radial position and is equal to [30]:

$$\frac{\partial v}{\partial z}\Big|_{z=0} = \frac{\partial v}{\partial z}\Big|_{z} = \frac{\omega}{\alpha} = \frac{\tau_W}{\mu} \qquad \text{(IX.2)}$$

z: direction of cone axis
$v(z)$: azimuthal velocity of fluid
ω: angular velocity of cone
α: cone angle in *rad*
τ_W: wall shear stress
μ: dynamic viscosity of fluid

The boundary conditions at cone and plate surface are: $v(z=0) = 0$ at the plate and $v(z_0(r_r)) = \omega r_r$,(r_r: radial position, cf. Fig. 19) at the cone.

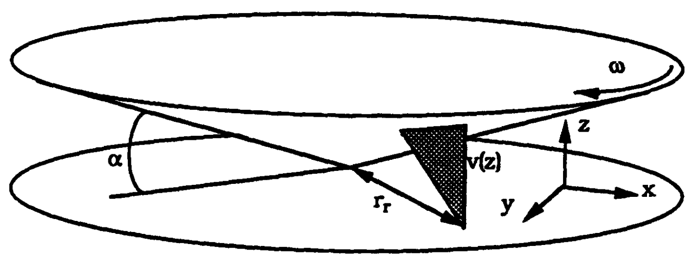


Figure 19. Geometry of the cone-and-plate flow

For the tests of the software system, $\tilde{R}$ was kept small so that no secondary flow occurred.

More detailed descriptions of the cone-and-plate flow may be found in [30].

IX.2 Experimental Design

IX.2.1 The Optics

Experimental equipment included a Lexel 95-3 argon ion laser emitting both green (514.5 nm) and blue (488 nm) light, 2 mirrors on kinematic optical mounts redirecting the beam into the 55X DISA LDA modular optics, a 160 mm anti-reflection coated front lens on a microtranslation stage and a 45° mirror (cf. Fig. 20), reflecting the focused through the bottom glass plate into the flow.

The DISA modular optics consisted of (in order of assembly starting from the laser side):

- 55X20/21 cover and retarder
- 55X22 beam waist adjuster
- 55X25 beam splitter
- 55X29 Bragg cell
- 55X28 beam splitter
- 55X23 support
- 55X30 backscatter section
- 55X31 pinhole section
- 55X32 beam translator
- Another 55X23 support
- 55X33 lens mounting ring
- Two 55X12 beam expanders

The photomultiplier section, mounted on the backscatter section, has a 55X39 polarization separator, two 55X08 PM sections (with two 55L97 power supplies), a 55X36 (488 nm) and a 55X37 interference filter (514.5 nm).

The front lens, 55X56, is an achromatic lens of focal length 160 mm. It can be moved in the x- and y-direction on microstages. The 45° mirror, the microtranslation stages with the front lens, the LDA Optics, and one of the redirecting mirrors were mounted on an optical bench. The optical bench, the laser, and the second redirecting mirror were mounted on a shock absorbing granite table (cf. Fig. 21).

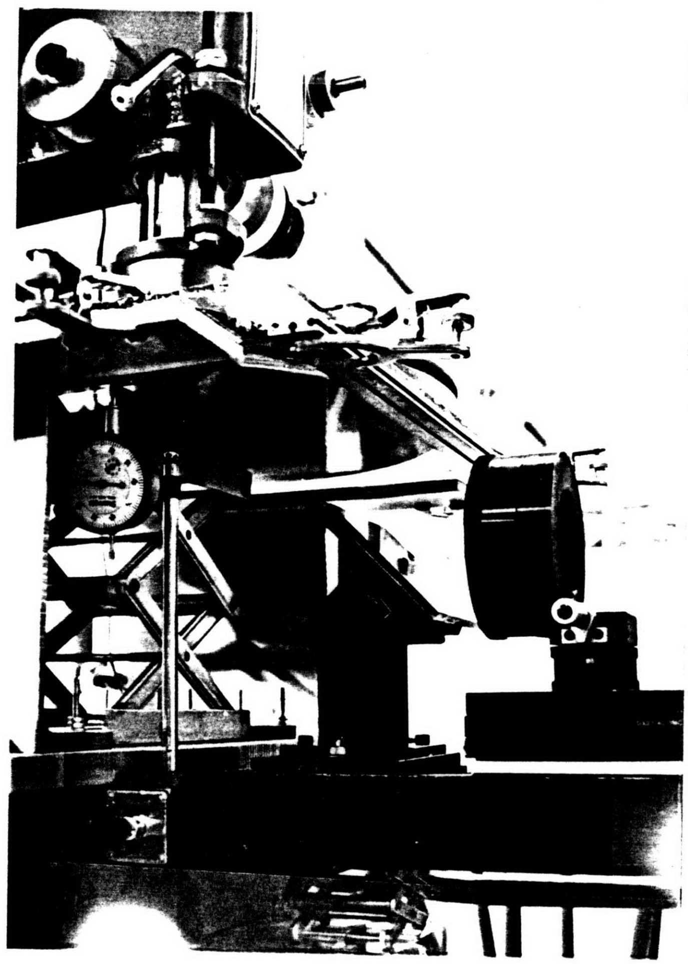


Figure 20. Photo 1 of the experimental set-up: cone-and-plate apparatus with LDA front lens and mirror. The dial indicator is used to determine when the apex of the cone hits the glass plate. The support in the middle of the glass plate prevents bending.

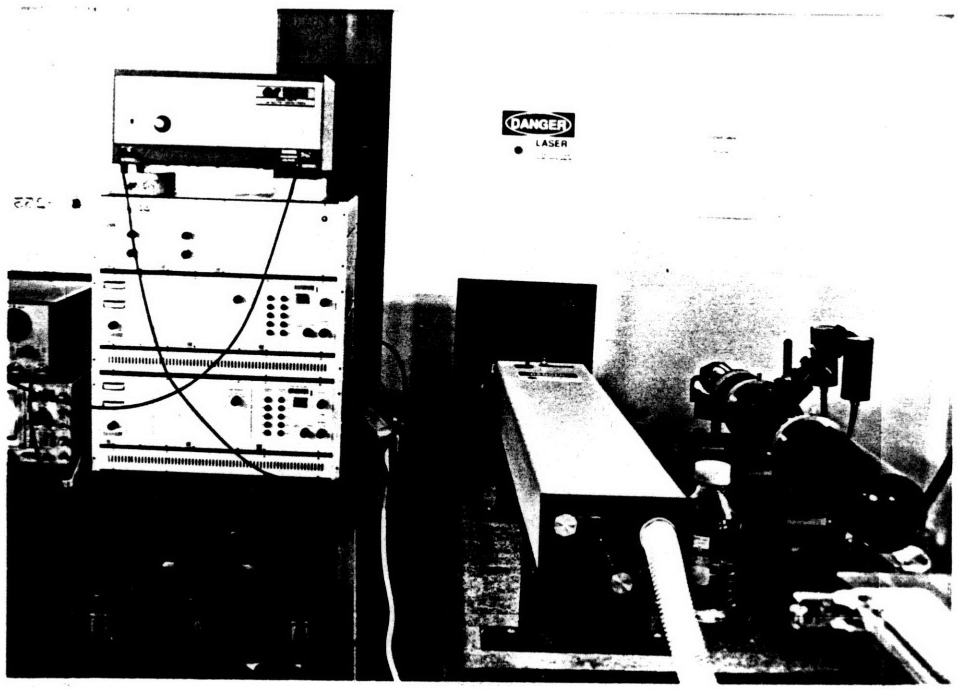


Figure 21. Photo 2 of the experimental set-up. From left to right, bottom to top: oscilloscope, filter; DISA Photomultiplier power supply and counter for blue light, dito for green light, DISA frequency mixer, laser, LDA optics.

Table 5 presents the optical parameters of the experimental set-up for the two visible wavelengths of an argon ion laser.

---

6. From: International Critical Tables

Beam waist before focusing optics	1.3 mm	
distance of beams from optical axis before beam expanders	$x_1' = 6.5\ mm$	
distance of beams from optical axis after beam expanders	$x_1 = 17.3\ mm$	expansion ratio $e_x = 3.7540$
refractive index of glass	$n_1 = 1.52$	
refractive index 100% Glycerine 25° C	$n_2 = 1.4730$[6]	
thickness of glass barrier	$d_2 = 4.7\ mm$	
width of gap at $r_r = 160\ mm$ from apex	$d_3 = 2.79\ mm$	
focal length of front lens	$f_1 = 160\ mm$	
half angle of intersection	$\theta = 0.076456$	Eqs.(II.7b) and (II.8)
fringe spacing	$\Delta x \approx 2.28\ \mu m\,(514.5\ nm)$ $\approx 2.17\ \mu m\ \ (488\ nm)$	Tbl.(1), $\sin\theta \approx \theta$
beam diameter at point of intersection	$d_f = \frac{4}{\pi}\frac{f\lambda}{e_x\, w_0\, n_2} = 14.6\ \mu m\ \ (514.5\ nm)$ $= 13.8\ \mu m\ \ (488\ nm)$	
length of probe volume	$d_z \approx 191.0\ \mu m\ (514.5\ nm), \approx 180.5\ \mu m\ \ (488\ nm)$	Tbl. (1)
number of fringes	$N_f \approx 6$	

TABLE 5. Optical parameters for $\lambda = 514.5nm, 488nm$

IX.2.2 The Flow Apparatus

The flow apparatus (cf. Fig. 20), was originally designed for another project (see Acknowledgements). The body is an Enco Milling and Drilling Machine Model 105-1100. The motor is a Bodine Gearmotor, type 4205BEPM-B2, with a torque of 68 lb. in., which can be adjusted between 0 and 200 rpm with a potentiometer. The rotational speed of the cone is monitored over a tachometer

The Ø 400 mm, 1° cone made of transparent Lucite is mounted on the shaft of the drill press. In order to limit reflections from the cone surface it has been spray-painted with ultra-flat black color. The bottom plate was made of a 4.7 mm thick glass plate.

The angular velocity determined the necessary sampling frequency which may not exceed 500 kHz. Requiring the Doppler frequency to be about a third of the sampling frequency ensures that all aliasing frequencies will be removed with the given filter roll-off. The settings in Tbl. 6 were used.

maximum Doppler frequency maximum sampling frequency	$f_D \approx 350\ kHz$ $f_s = 1\ MHz$
maximum azimuthal velocity	$U_0 = f_D\ \Delta x = 0.798\ \frac{m}{s}$
rotational speed for $r_r = 160mm$	$\omega = \frac{U_0}{r_r} = 299.25\ \frac{rad}{\text{min}}$ $f = 47.6\ rpm$
density of fluid viscosity of fluid (100% Glycerine, 20° C)	$\rho = 1.2609\ \frac{g}{ml}$ $\nu = 0.001120\ \frac{m^2}{s}$
cone angle	$\alpha = 1° = 0.0174\ rad$
	$\tilde{R} < 0.0015$

TABLE 6. Flow parameters

IX.2.3 The Signal Path

The signals from the photomultiplier tubes were fed into two DISA 55L96 Counter Processors. The Counter Processors permits attenuation of the signal in 1 dB steps up to -31 dB and lowpass (edge frequencies 256 kHz, 4, 16, 100 MHz, roll-off 60 dB/decade) and highpass filtering (edge frequencies 1, 4, 16, 64, 256 kHz, 2, 4, 16 MHz, roll-off 40 dB/decade). The signal was then amplified with an Amplifier Research 50A15, anti-alias filtered with a Krohn-Hite 3202 filter, and then digitized by the *MASSCOMP* A/D converter. The signal strength may not exceed ± 5 V in bipolar mode or 10 V in unipolar mode, otherwise clipping in the A/D converter occurs.

IX.3 Experimental Results

The software system was used for the cone-and-plate flow with the parameters of Table 6. Originally, it was decided to use fluorescent Fluoresbrite™ particles of 0.77 μm diameter (corresponding to roughly a $\frac{1}{4}$ of the fringe spacing) together with a Hoya Y-52 optical filter to block out all wavelengths below 520 nm (the maximum emission line of Fluoresbrite is at 540 nm). The specific gravity of 1.05 of these particles was matched by a 20:80 glycerine:water solution. The use of these particles, however, resulted in seeding problems.

A particle concentration of $1 \frac{particle}{probe\ volume} \approx 10^7 \frac{particles}{ml}$ was already so high that most of the laser light was absorbed before it hit the cone surface. Lower particle concentrations resulted in only sporadic bursts, which make the software system very inefficient to use: in order to find a burst, batches of data have to be repeatedly sampled, transferred over the net to the second machine, and analyzed. The transfer over the network accounts for most of the processing time. Sparsely seeded flows, on the other hand, require a large data throughput for validation of a sufficient number of bursts. Limitations on the disk space also did not permit acquisition of the data for the whole experiment followed by automated processing.

In a second attempt, particles covered with silver oxide and a diameter of $2\,\mu m$ were used. Unfortunately, no other information about these particles was available. Preliminary tests showed that 100% glycerine has to be used in the flow to avoid immediate sedimentation. Approximately $4\,mm^3$ of theses particles were put in $1\,l$ glycerine. This resulted in almost continuous transitions of particles through the probe volume. Due to the high viscosity of glycerine, special care must be given to keep the flow free of bubbles from the very beginning.

The system was first calibrated by measuring the Doppler frequency at the cone surface, the gap filled with 100% glycerine. The angular velocity of the cone was increased in steps of 2 rpm from 15 rpm to 41 rpm. A first order polynomial was then fitted to the data. Its two coefficients can then be entered on the command line of the shell script `MasterPlan`. Only one such calibration experiment was conducted, as the goal of the experiments was only to demonstrate the workability of the software system. The result of this calibration are shown in Fig. 22.

The measurements started at the cone surface, where proper focusing was easily monitored: the fringe pattern was propagating in one direction as one of the laser beams was shifted by 40 MHz. This resulted in a sinusoidal signal in the photomultipliers which attained its maximum if the center of the probe volume hit the cone surface. Starting from this position the lens was moved in steps of $\Delta d_1 = 0.127\,mm$ corresponding to a translation of the probe volume by $\Delta d_3 = \Delta d_1\, n_2 = 0.187\,mm$ (cf. Eq. II.7a) which corresponds

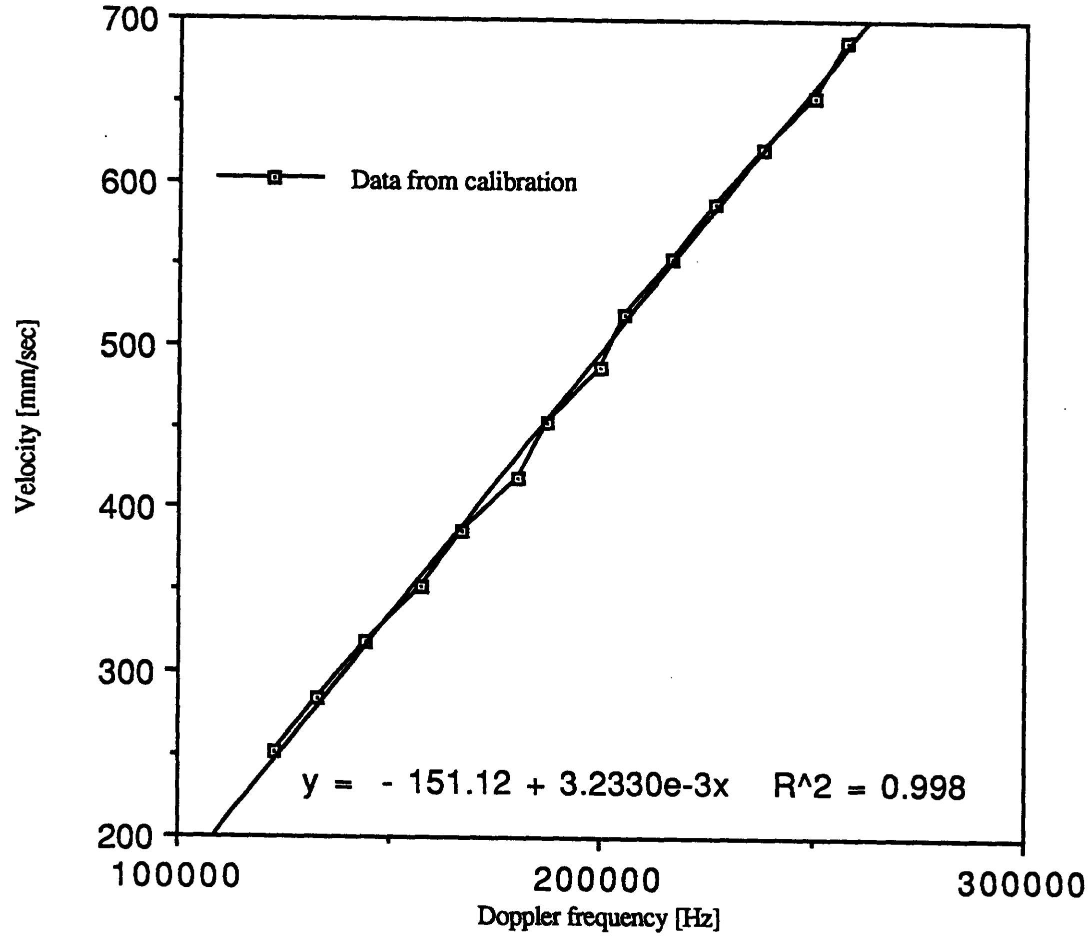


Figure 22. Results from calibration experiments, the gap was filled with 100% glycerine, the angular velocity of the cone was increased from 15 rmp to 41 rpm in 2 rpm steps.

approximately to the length of the probe volume for $\lambda = 488\ nm$. Thus, at $r_r = 160\ mm$, 15 measurements across the gap could be taken. The motion of the probe volume will be parallel with the axis of the cone.

Fig. 23 shows one measurement of the velocity profile in the z-direction. The velocity gradient as obtained by a first order polynomial fit, $169 \frac{mm/sec}{mm}$ is 13% over the theoretical velocity gradient of $150.1 \frac{mm/sec}{mm}$. Higher accuracy can be expected with a

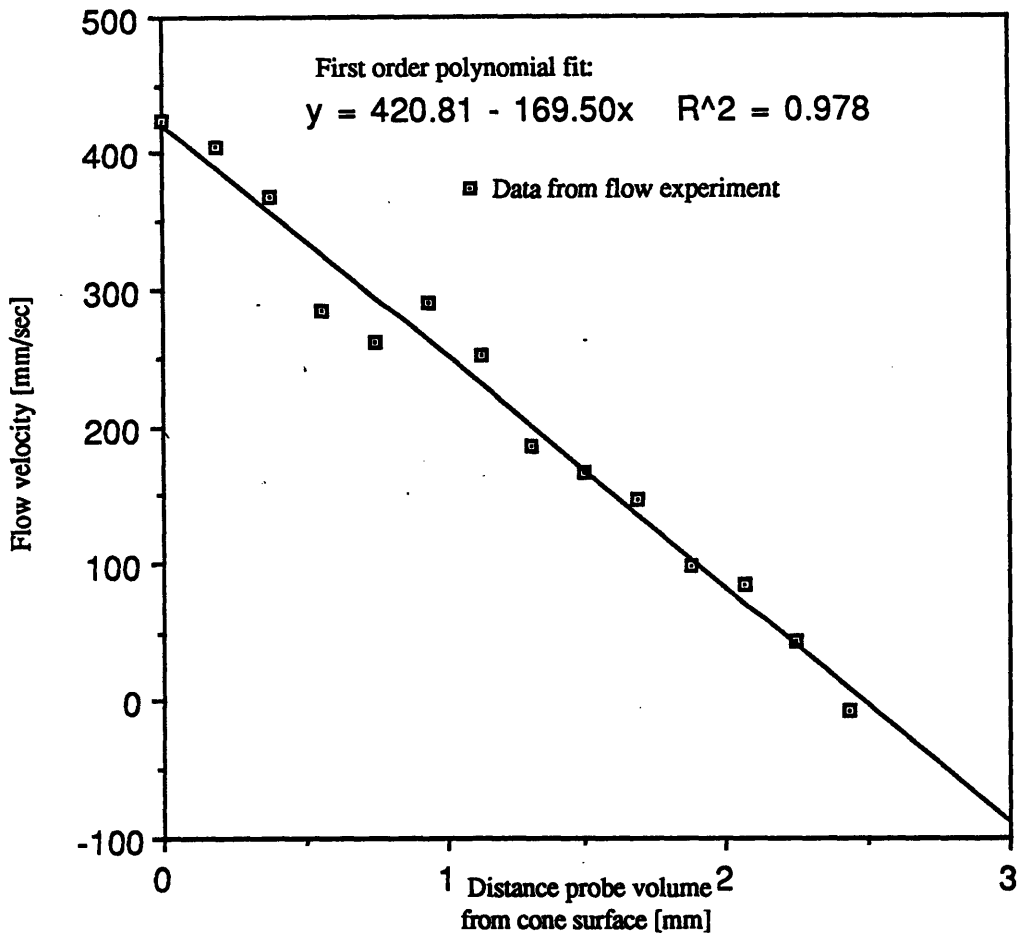


Figure 23. Results from investigation of cone-and-plate flow. At each each position of the probe volume within the flow, 40 bursts of minimum length 15 samples were collected. The signal was bandpass-filtered 20 kHz to 250 kHz, and sampled with $f_s = 500\,kHz$.

higher

number of bursts and careful adjustment of the cone and plate surfaces. The velocity profile in Fig. 23 reaches zero velocity already after 14 positions, 1 less than theoretically necessary. This discrepancy may have resulted from bending of the glass plate if the apex touched the plate and from a tilted cone. Subsequent measurements of the cone showed indeed deviations of up to 0.127 mm. These deformations were most likely due to frequent disassembly for cleaning. A more sturdy apparatus should use an aluminum cone and plate with glass inlets for the laser beams as used in [20].

Of major interest was also whether the general behavior of the algorithms under real flow conditions matches the behavior in the numerical simulations. Fig. 24 shows a representative mean spectrum of two bursts with the DFT-based method. In Fig. 25, the mean spectrum of the Pade estimator, the spectral peak in the Pade estimation is much more visible, in agreement with the numerical simulations. This leads to the conclusion that the numerical simulations with additive white noise indeed predict reliably the behavior of the spectral estimators under real conditions.

Finally the viability of the digital prefiltering was tested. In the resulting sample spectrum, Fig. 42, the filter roll-off is visible in the region around 50 kHz.

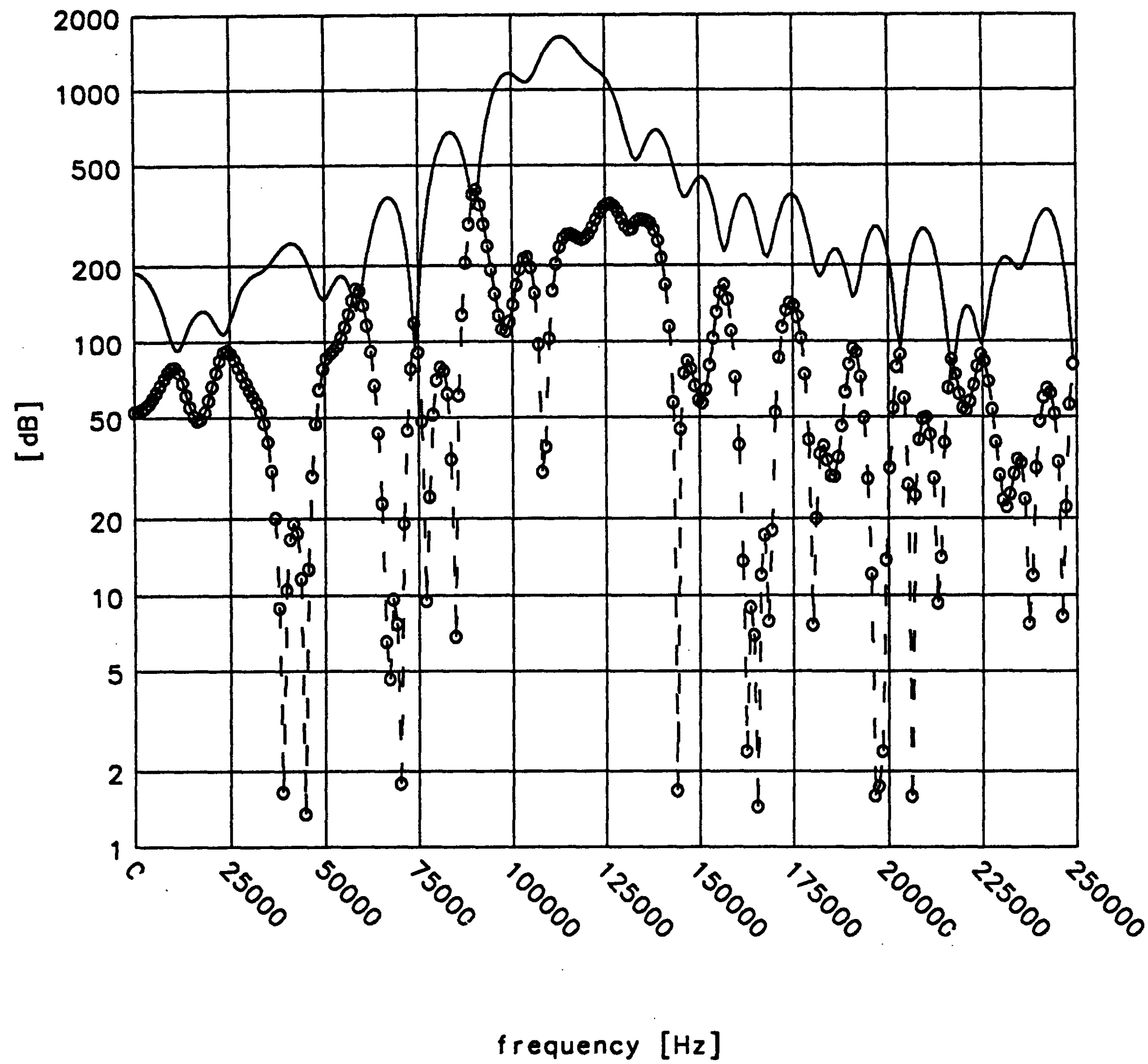


Figure 24. Mean spectrum of two bursts with the DFT method. The signal was bandpass filtered with cut-offs 20 kHz and 200 kHz. The dotted line is the variance at a frequency as computed by `MeanSpec`

X. Conclusions and Direction of Future Work

The applicability of adaptive spectral estimation methods to laser-Doppler anemometry has been demonstrated in numerical simulations using synthetically

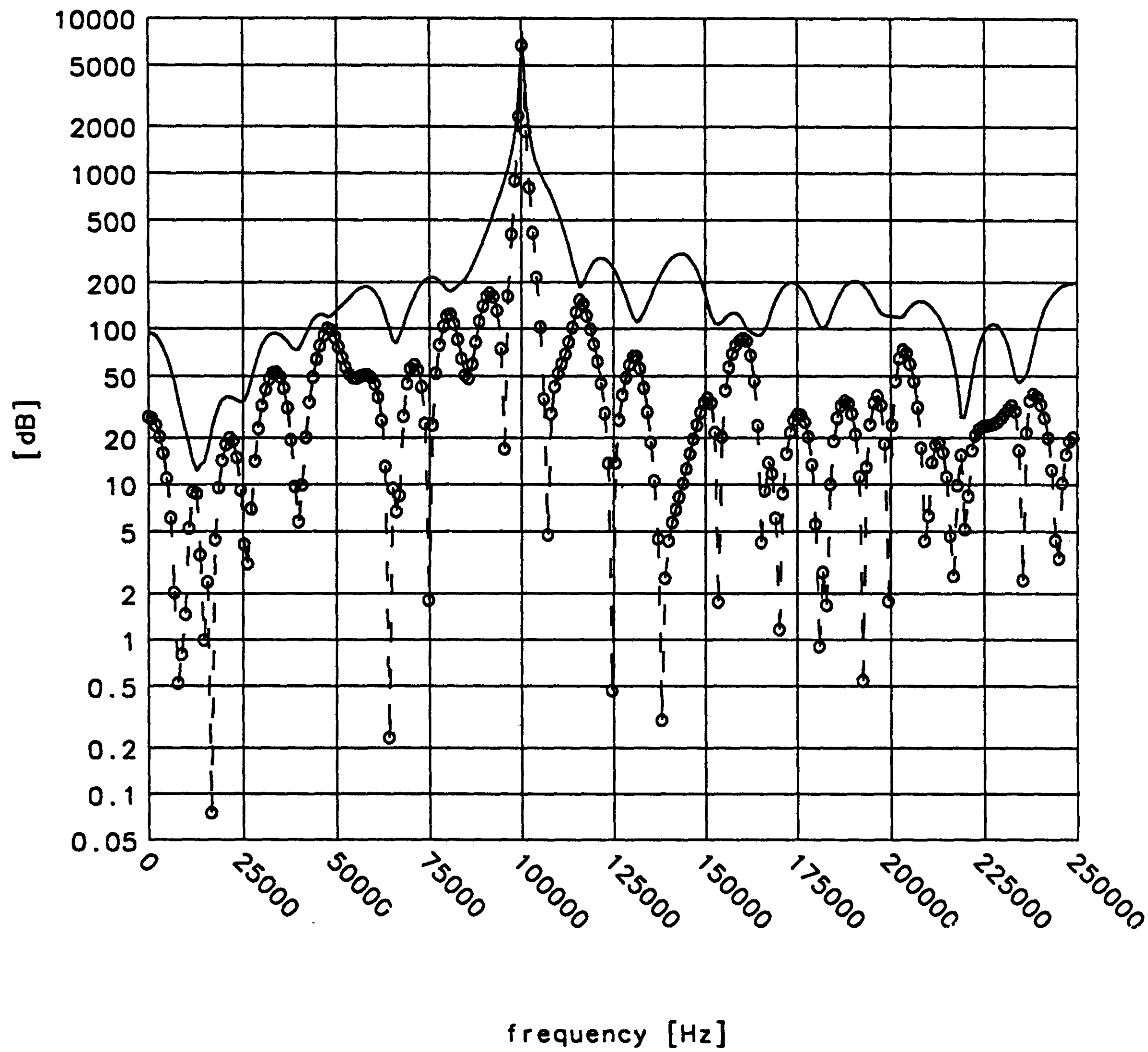


Figure 25. Mean spectrum of two bursts with the Pade estimator.

generated Doppler signals. Adaptive methods for finding the Doppler frequency are not widespread in LDA, although it is shown that for low signal-to-noise ratios their performance is superior to the traditional methods, the direct computation of the DFT of the data.

Three different adaptive algorithms have been compared with the DFT: The Modified Covariance Algorithm, an auto-regressive method, performs poorly for noisy signals. The Iterative Filtering Algorithm, a procedure which enhances the results of auto-regressive estimators, for signals with low signal-to-noise ratio, leads indeed to greater insensitivity against noise. Its computational complexity however makes it not very attractive for high speed data analysis. The last algorithm is an auto-regressive moving-average estimator based on a Pade approximation to the spectrum of the data record. It performed very well even for noisy signals. Its simple algorithmic structure permits easy implementation.

In a second step, a software system for processing LDA signals has been developed. It comprises programs for digital finite impulse response filters and for digital filtering. Burst validation is done in the time domain and is based on the envelope of the signal, triggering can be done either using first order statistics for the trigger levels, or using absolute trigger values. The velocity at a point of a flow field is determined after a specified number of particle transitions has been processed. First, the mean spectrum is computed with the residence-time weighting method. The Doppler frequency is computed as the first moment of the local region in the spectrum having the highest mean. Programs for obtaining the velocity profile and for plotting were also designed.

The viability of this software system was verified in an experimental investigation of a cone-and-plate flow.

The behavior of the auto-regressive moving-average estimator and the DFT corresponds closely to the behavior observed in the numerical simulations. This indicates that for laser-Doppler anemometry the testing of spectral estimation algorithms on signals with additive white noise predicts the behavior in real experiments.

Future work should be directed towards the development of more reliable burst detection algorithm. The procedure used in this project still requires fine tuning from the user to produce an efficient data rate.

Literature

[1] **Aho, A V, Hopcroft, J E, Ullman, J D**, 1974: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

[2] **Bates, C J, Haddad, C**, 1985: Software Concepts for a Transient Recorder and a 6502-based Microprocessor Combination for LDA Signal Processing, *Int. Conf. on Laser Anemometry-Advances and Application,* 16th-18th Dec 1985.

[3] **Bendat, J S, Piersol** , 1986: *Random Data.*Wiley Interscience, New York.

[4] **Blahut, R E**, 1985: *Fast Algorithms for Digital Signal Processing,* Addison-Wesley, Reading, MA.

[5] **Brent, R P, Gustavson, F G, Yun, D Y Y**, 1980: Fast Solution of Toeplitz Systems of Equations and Computation of Pade approximants, *Journal of Algorithms,* **1**, pp 259-295, 1980.

[6] **Buchhave, P**, 1979: The Measurement of Turbulence with the Burst-type Laser-Doppler Anemometer - Errors and Correction Methods, *PhD Thesis,* State University of New York, Buffalo.

[7] **Buchhave, P**, 1984: Three Component LDA Measurements, *DISA Information*, No 29, January 1984.

[8] **Digital Signal Processing Committee [Ed.]**, 1979: *Programs for Digital Signal Processing*, IEEE Acoustics, Speech, and Signal Processing Society, IEEE Press, New York, NY.

[9] **Durst, F, Melling, A, Whitelaw, J H**, 1981: *Principles and Practice of Laser-Doppler Anemometry*. Academic Press New York, New York.

[10] **Harris, F J**, 1978: On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform. *Proc. IEEE, vol. 66,* pp. 51-83, Jan. 1978.

[11] **Jenson, L M, Menon, R K, Miller, J D, Fingerson, L M**, 1988: An Automatic Signal Processor for LDV Systems, *TSI Flow Lines* , 1988.

[12] **Kay, S M**, 1978: Improvement of Autoregressive Spectral Estimates in the Presence of Noise. *IEEE Proc. 1978 IEEE ASSP.*Tula, OK, pp 357-360.

[13] **Kay, S M,** 1984: Accurate Frequency Estimation at Low Signal-To-Noise Ratio*IEEE Transactions on Acoustics, Speech, and Signal Processing,* Vol. ASSP 32, No. 3, June 1984.

[14] **Kay, S M,** 1988: *Modern Spectral Estimation, Theory and Application.* Prentice Hall, Englewood Cliffs, NJ.

[15] **Knuth, D E,** 1981: *The Art of Computer Programming, Volume 2/ Seminumerical Algorithms,* Addison-Wesley, Reading, MA.

[16] **Kogelnik, H,** 1979: Propagation of Laser Beams, in: *Applied Optics and Optical Engineering*, Shannon, R R, Wyant, J C [eds.], **VII**, Academic Press, New York, NY.

[17] **Lading, L,** 1987: Spectrum Analysis of LDA Signals. *Dantec Information 05,* pp. 2-8, Sept. 87.

[18] **Layne, T C, Bomar, B W,** 1987: Discrete Fourier Transform Velocimeter Signal Processor. *IEEE ICIASF '87 Record,* pp 121-124.

[19] **Marple, S L, jr.,** 1987: *Digital Spectral Analysis with Applications.* Prentice Hall, Englewood Cliffs, New Jersey.

[20] **Marquez, R H G,** 1984: *Velocity Measurements in a Shear Flow Using Fluorescent Particles and Laser-Doppler Anemometry,* SM Thesis, MIT, Mech E Department.

[21] **McEliece, R J, Shearer, J B,** 1978: Property of Euclid's Algorithm and an Application to Pade Approximation, *SIAM J. Appl. Math.,* Vol.34, No.4, June, 1978.

[22] **Meyers, J F, Clemmons, J I, jr.,** 1987: Frequency Domain Laser Velocimeter Signal Processor. *NASA TP 2735,* 1987.

[23] **Newland, D E,** 1984: *An Introduction to Random Vibrations and Spectral Analysis*, Longman Scientific and Technical, Essex, UK.

[24] **Oppenheim, A V, Willsky, A S, Young, I T,** 1983: *Signals and Systems.* Prentice Hall, Englewood Cliffs, NJ.

[25] **Oppenheim, A V, Schafer R W,** 1989: *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, NJ.

[26] **Pallek, D,** 1985: Fast Digital Data Acquisition and Analysis of LDA Signals by Means of a Transient Recorder and an Array Processor. *IEEE ICIASF 1985 Record*, pp. 309-312.

[27] **Palmer, R D, Cruz, J R,** 1989: An ARMA Spectral Analysis Technique Based on a Fast Euclidean Algorithm, *IEEE Trans. on ASSP,* Vol. ASSP *37,* pp, 1532-1537, October 1989.

[28] **Press, W H, Flannery, B P, Teukolsky, S A, Vetterling, W T,** 1988: *Numerical Recipes in C - The Art of Scientific Computing*, Cambridge University Press, Cambridge, UK.

[29] **Schaefer, R T, Schafer, R W, Mersereau, R M,** 1987: Digital Signal Processing for Doppler Radar Signals. *IEEE Proc. 1979 ICIASSP*. pp. 170-173.

[30] **Sdougous, H P, Bussolari, S R, Dewey, C F,** 1984, *Secondary Flow and Turbulence in a Cone-and-Plate device*, J. Fluid Mech. (1984), **138,** pp. 379-404.

[31] **Staas, F A,** 1985: On-Line Data Reduction of the Doppler-Difference Autocorrelation Function. *Int. Conf. on Laser Anemometry - Advances and Application,* 16th - 18th Dec 1985, Manchester, UK.

[32] **Sugiyama, Y,** 1986: An Algorithm for Solving Discrete-Time Wiener-Hopf Equations Based upon Euclid's Algorithm, *IEEE Trans. Inform. Theory,* vol. IT-32, pp. 394-409, 1986.

[33] **Swart, P L, Venter, C-V,van der Merwe, D F,** 1985: Parametric Spectral Estimation Applied to Laser Anemometry. *Int. Conf. on Laser Anemometry - Advances and Application,* 16th - 18th Dec 1985, Manchester, UK.

[34] **Tyau, L J,** 1982: *Use of Fluorescent Tracer Particles and Laser-Doppler Anemometry*, SM Thesis, Mech E Department.

[35] **Van Atta, C W, Chen, W Y,** 1969: Correlation Measurements in Turbulence using Digital Fourier Analysis. *The Physics of Fluids Supplement II,* 1969, *High Speed Computing in Fluid Dynamics*.

Manuals

[36] **Concurrent Computer Corporation**: Data Acquisition and Programming Manual

[37] **Concurrent Computer Corporation**: Data Presentation Application Programming Manual

[38] **Concurrent Computer Corporation**: VA Programmer's Manual

[39] **Concurrent Computer Corporation**: DA/CP Manuals

[40] **DISA** 55X Modular LDA Optics - Instruction Manual, DISA Information Department.

[41] **DISA** 55L90a Counter Processor - Instruction Manual, DISA Information Department.

List of Symbols

The following symbols are used throughout this paper:

$A(z)$: denominator polynomial, AR branch of $H(z)$
$a[n]$: coefficients of $A(z)$
a: AR coefficient vector
A, B, C, D: elements of the system ray transfer matrix
$B(z)$: numerator polynomial, MA branch of $H(z)$
$b[n]$: coefficients of $B(z)$
$\mathbf{C}_{xx}$: covariance matrix
$\mathbf{c}_{xx}$: covariance vector
d_1: distance from front lens to glass plate
d_2: thickness of glass plate
d_3: width of gap between cone and glass plate
d_x: length of probe volume in x-direction
d_y: length of probe volume in y-direction
d_z: length of probe volume in z-direction
$deg\ [*]$: degree of polynomial *
$E\ [*]$: expected value, mean of *
$e_f[n]$: forward prediction error
$e_b[n]$: backward prediction error
f, f_1: focal length of front lens
f_s: sampling frequency
f_D: Doppler frequency
$floor\ [*]$: floor operation
$G(z)$: general arbitrary polynomial
$G_N(z)$: N^{th} truncation of $G(z)$
g_i: coefficients of $G(z)$
$H(z)$: system transfer function, frequency response
i: iteration index
i_0: arbitrary but fixed iteration index
k: discrete frequency index
L: length of discrete-time series
l: discrete time index
m: discrete time index
(1)(mod [(2)]): remainder of polynomial (1) divided by polynomial (2)
M: length of a discrete-time series
n: discrete time index
N: length of a discrete-time series
N_f: number of fringes in $\frac{1}{e^2}$-contour
n_1: refractive index of glass plate
n_2: refractive index of fluid
p: order of AR branch
P: length of discrete-time series
P_{AR}: PSD for AR model
P_{xx}: PSD (auto-spectral density) of discrete-time series $x[n]$
P_i: PSD of i^{th} data segment
$\bar{P}_{xx}$: mean PSD for data record $x[n]$
q: order of MA branch
$q_i(z)$: quotient polynomial of i^{th} iteration (Euclidean Algorithm)

$q(z)$: complex propagation parameter for Gaussian laser beams
$\tilde{R}$: dimensionless similarity parameter for cone-and-plate flow
$R(z)$: curvature of Gaussian laser beam along optical axis
$r_i(z)$ remainder polynomial after i^{th} iteration (Euclidean Algorithm)
$\mathbf{r}_i$: paraxial ray vector at position i
$r_{c,0}$: location of center of particle at time $t = 0$
r_p: particle radius
r_{xx}: autocorrelation function of discrete-time series $x[n]$
r_{rect}: autocorrelation function of the rectangular time window
S: length of discrete-time series
$s(t)$: Doppler signal in continuous-time
s_0: scaling factor for Doppler signal
T: sampling period
t: continuous time
$t_i(z)$: co-multiplier polynomial after i^{th} iteration (Euclidean Algorithm)
U_0: velocity component perpendicular to fringe system
$u(\mathbf{x}, t)$: velocity field
$\bar{u}(\mathbf{x}_0)$: mean velocity at point $\mathbf{x}_0$ in flow field
$u'(\mathbf{x}, t)$: velocity fluctuations
$\bar{u}_{LDA}$: measured mean velocity
$u[n]$: white-noise input to a system
V: size of probe volume
$v(z)$: azimuthal velocity in cone-and-plate flow
$w[n]$: discrete-time window function
$w_\alpha[n]$: normalized lag-window function
$W[k]$: DFT of $w[n]$
$w(z)$: $\frac{1}{e}$-radius of Gaussian laser beam along optical axis
w_0: beam waist diameter before focusing optics (after beam expansion)
w_f: waist diameter of focused laser beam
x: optics: distance of beam to optical axis
x: cone-and-plate flow: radial direction
$x[n]$: discrete-time series (usually data or input)
$X(z)$: z-transform of $x[n]$
$x_w[n]$: windowed discrete-time series
$x_\infty[n]$: infinite-length discrete-time series
$x_i[n]$: i^{th} segment of a discrete-time series
$X[n,k]$: time-varying spectrum of instationary $x[n]$
$\hat{x}_f[n]$: forward-predicted sample
$\hat{x}_b[n]$: backward-predicted sample
Δx: fringe spacing
$\mathbf{X}_i$: ray transfer matrix of i^{th} optical component
$\mathbf{X}$: system ray transfer matrix
y: cone-and-plate flow: azimuthal direction
z: complex variable
z: optics: direction along optical axis
z: cone-and-plate flow: direction of axis of cone

α: cone angle
$\delta[n]$: unit impulse function
θ: half-angle of intersection
λ: wavelength of laser
(μ, ν): order of Pade estimator

ν: kinematic viscosity of fluid
ρ: density of fluid
ρ_f : forward prediction error power
ρ_b : backward prediction error power
σ_o^2: variance of observation noise (white)
σ_u^2: variance of input noise (white)
τ_W : wall shear stress
ϕ: angle of intersection at probe volume
ω: angular velocity of cone

Appendix 1: MATLAB Routine `mksig.m` for Generating LDA Signals

```
clear;
% create a Doppler signal with random time intervals between the bursts
%

%---------------------------------------------------------------------
%
%                            DEFINITIONS
%
%---------------------------------------------------------------------
nburst=20;                % we take nburst bursts:

fs = 1000000;             % fix the sampling rate [Hz]:

SNR=-10;                  % define the signal-to-noise ratio:

% flow parameters

u=1000.000;               % set the velocity [mm/sec]

meapaus=0.0001;           % define the mean pause between the bursts [s]:
meapaus=meapaus*fs;       % convert the mean pause in number of samples:

aa=0.5;                   % velocity fluctuations in percent of u

ff=300;                  % frequency for oscillatory flow

%------------------------------------------------------------------------

% set the parameters of the optical set-up:
% we simulate a DISA 55X Modular Optics LDA
% with Laser Type 124B and Front Lens X51


f=300;                    % focal length of the front lens [mm]:

lambda=633/1000000;       % wavelength of the laser [mm]:

th=7.44;                  % angle of intersection [grad]:

dw=1.1;                   % waist diameter of the unfocused beam [mm]:


%-----------------------------------------------------------------------

th=(th*pi)/180;           % convert angle of intersection into rad:

co=cos (th/2);            % abbreviations
si=sin (th/2);

%------------------------------------------------------------------------

delt=1/fs;                % get the time step:

df=(4*lambda*f)/(pi*dw);  % 1/e^2-diameter of the focused laser beam:

var = (df/4)^2;           % get the variance of the Gaussian laser beam:

fd=2*u*si/lambda;         % the Doppler frequency of the burst:


a=df/(2*co);              % a,b,c: half axes of the ellipsoid:
b=df/2;
c=df/(2*si);

scale=[b c];

save;             % ... the parameters on disk

tint=0;
```

```
   sig=[];
n=0;

while (n < nburst)

        % get the 'time' (sample number) of the burst:
        tint=tint + round(rand * 2 * meapaus);

        % get the burst:
        temp=transit(fs,length(sig),scale,delt,var,co,si,u,fd,a,b,c,aa,ff);

        temlen=length(temp);
        sigend=tint+temlen-1;            % new length of vector 'sig'

        % fill sig with zeros to allow addition of new burst
        sig=[sig zeros(1,sigend-length(sig))];

        % now add the new burst allowing for superposition of bursts:
        sig(tint:sigend)=sig(tint:sigend)+temp;

        n=n+1
end

save sig sig;
% now add noise to the "pure" signal such that given signal-to-noise
%     ratio is maintained:

temp=rand(1,length(sig));

% the variances of the vectors containing burst and noise are:
sigvar=cov(sig);
noivar=cov(temp);

% compute the factor we have to multiply the noise with in order to
%    obtain desired snr:
fac=sqrt(sigvar*(10^(-SNR/10))/noivar);

temp = fac .* temp;
save temp temp;
noivar=cov(temp);

if noivar ~= 0
        snrist=10*log10(cov(sig)/cov(temp));
end;

sig = sig + temp;

% quantize the signal with 12bit resolution:
%  (assume no clipping takes place)
sig=quant(sig,12,max(sig));

save dcsig sig;
```

```
function [i]=transit(fs,siglen,scale,delt,var,co,si,u,fd,a,b,c,aa,ff);

% simulate the transition of a particle through the probe volume
%  of a laser-Doppler anemometer and return the resulting signal
%
% the location of the crossing in the z-y-plane is arbitrary

% pick random y-z-coordinates for the transition path of the particle

xo=rand(2,1);
while ((1-(xo(1))^2-(xo(2))^2) < 0)
        xo=rand(2,1);
end

xo=xo .* scale';            % scale the xo-vector accordingly:

% the x-coordinate of the point of intersection with the probe volume:
xstart=-a*sqrt(1-(xo(1)/b)^2-(xo(2)/c)^2);
xend=-xstart;
```

```
    delx=u*delt;              % path in one time step

ut=u;
n=1;


x=[xstart:delx:xend+delx];

%  CHANGE FOR FLOW SIMULATION:

% oscillatory flow
% ut=u + aa*sin( ( (siglen)/fs)*ff*2*pi);

% for white turbulent fluctuations uncomment the next line;
%        ut = u + aa*rand;

i=intens(x,xo(1),xo(2),var,co,si,ut,fd);

```

.sp

```
function [i] = intens(x,y,z,var,co,si,u,fd)

% computes the intensity at the point (x,y,z) in the probe volume

        i1 = 1./(4.*pi*var)*exp(-.5*( ((x*co)-(z*si)).^2 +y^2)/var);
        i2 = 1./(4.*pi*var)*exp(-.5*( ((x*co)+(z*si)).^2 +y^2)/var);
        i = i1+i2+2* sqrt(i1.*i2) .* cos(2*pi*fd*x/u);

```

Appendix 2: Examples of Signals Created by `mksig.m`

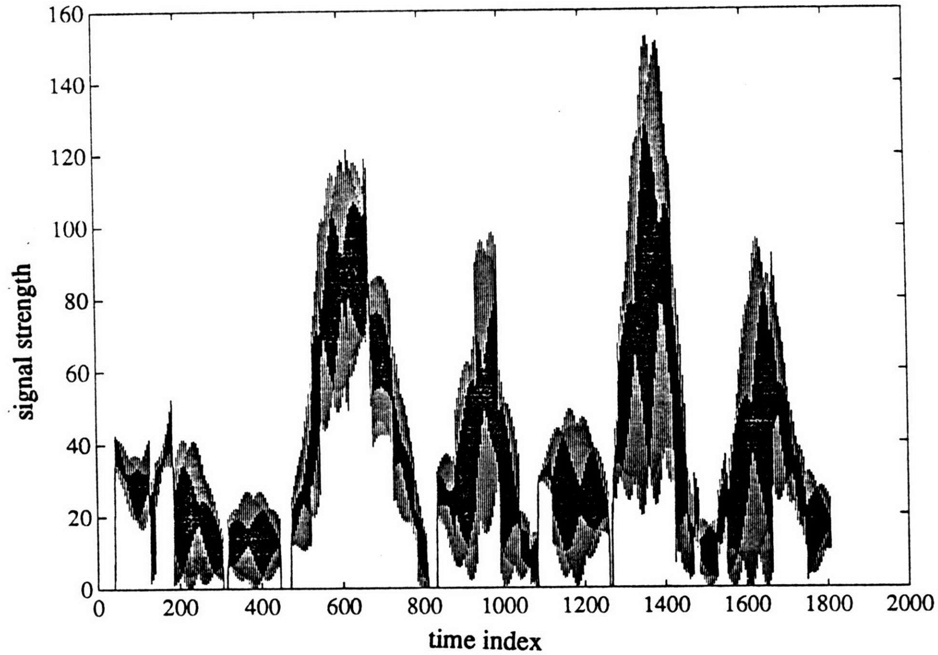


Figure 26. Simulated LDA signal, 20 bursts, SNR = 2000 dB, seeding `meapaus=` 0.0001. The high seeding leads to overlapping bursts. This signal is the basis for all subsequent numerical simulations where only white noise is added to this signal.

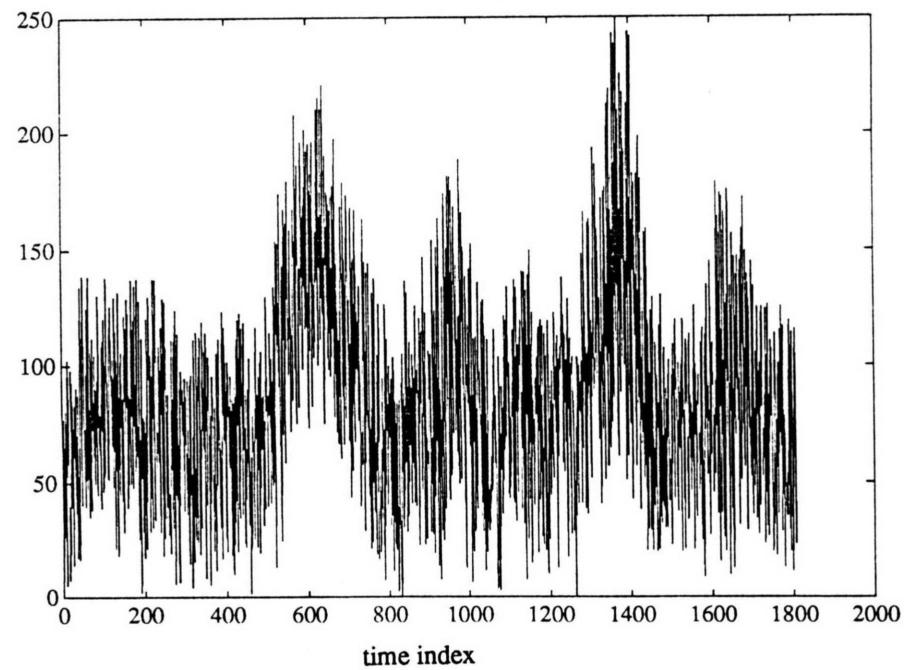


Figure 27. White noise has been added to the signal of the previous Fig. SNR = 0 dB. Burst detectors based on some envelope criteria as the one used in this project will validate only the strongest bursts.

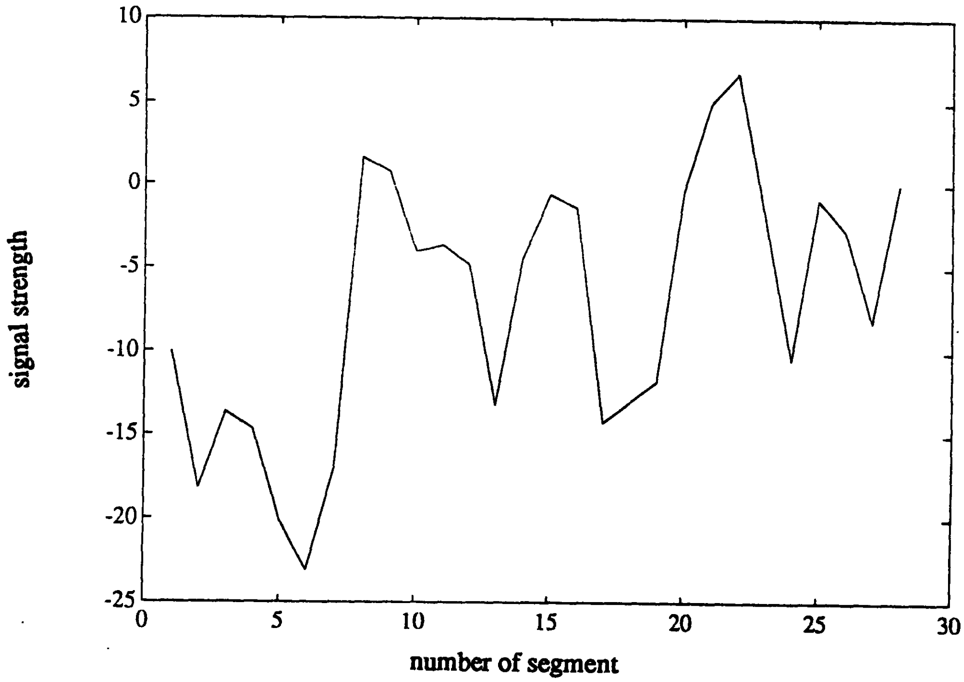


Figure 28. Changes in the local SNR ratio over 128 point long segments overlapping by 50 %. For a constant background noise, the SNR will be lowest for particles crossing the center of the probe volume (strongest signal)

Appendix 3: Spectrograms of Spectral Estimators with Simulated LDA Signals

Appendix 3.1: Results of Classical DFT-based Method

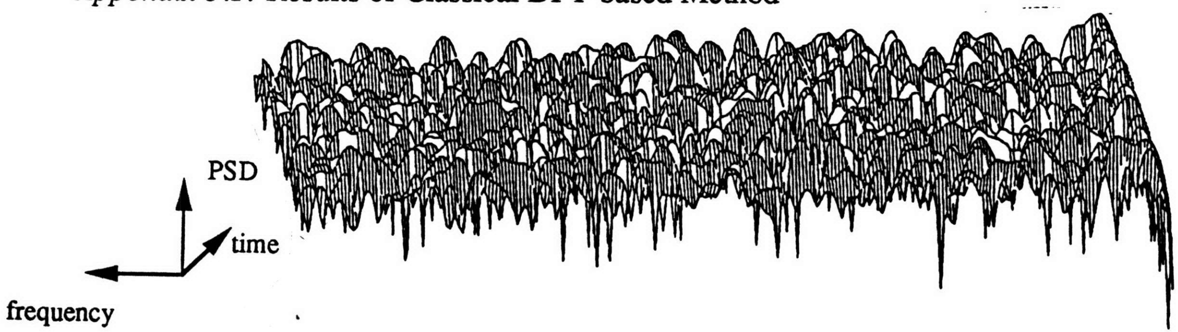


Figure 29. Result of classical spectral estimator: DFT of Hamming windowed (128 points) data. The segments were overlapped 50 %. SNR = 10 dB same signal as in Fig. ####. The variance in the spectral estimate is high.

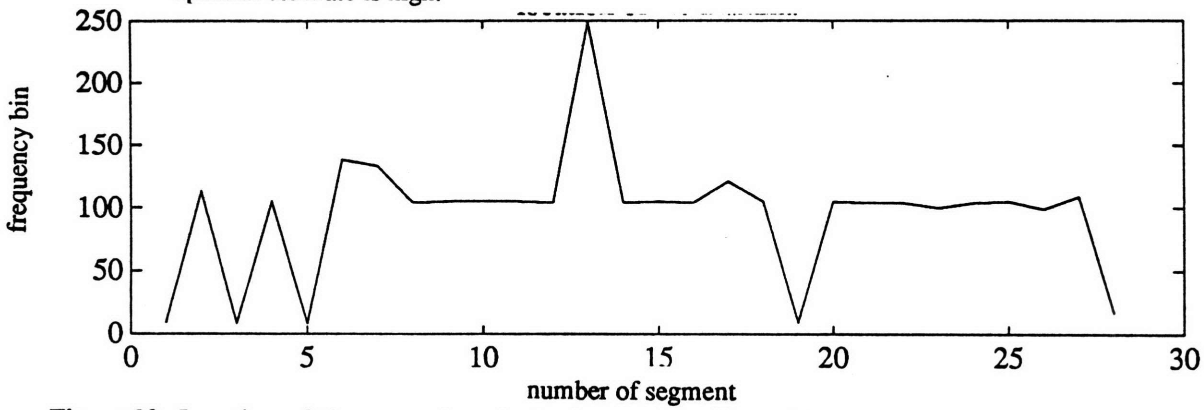


Figure 30. Location of the spectral peaks in the previous Fig. Although the spectral peaks are hardly recognizable in the spectrogram, they correspond well to the Doppler frequency of the signal.

Appendix 3.2: Results of the Modified Covariance Algorithm

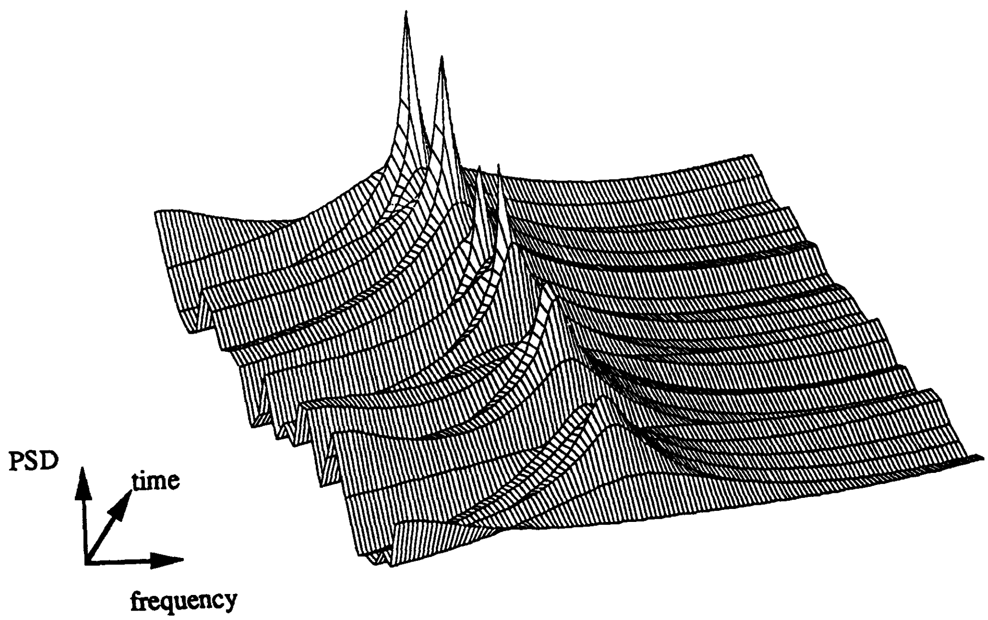


Figure 31. Third order AR model with the 10 dB signal. Note the very low variance in the spectral estimate. The spectra are of 128 point long segments overlapped by 50 %.

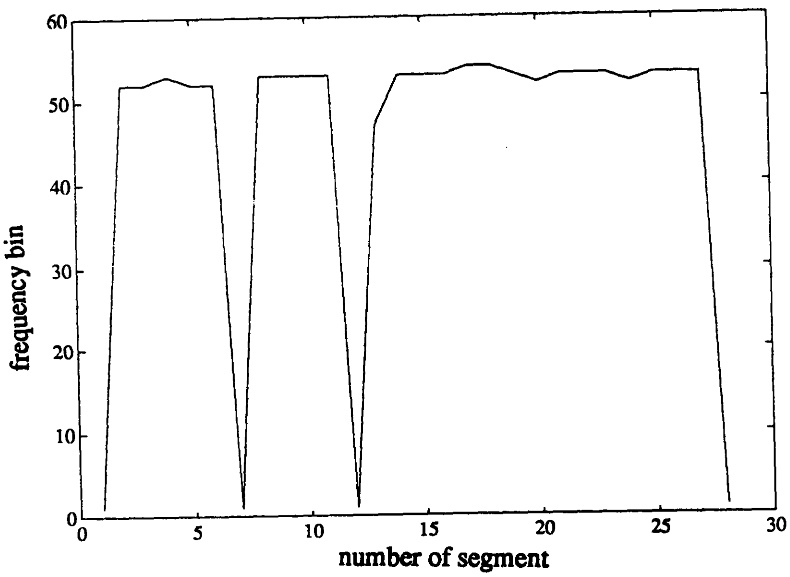


Figure 32. Location of spectral peaks in the previous Fig. The Doppler frequency is well resolved. The drop-outs correspond to the locations of lowest local SNR

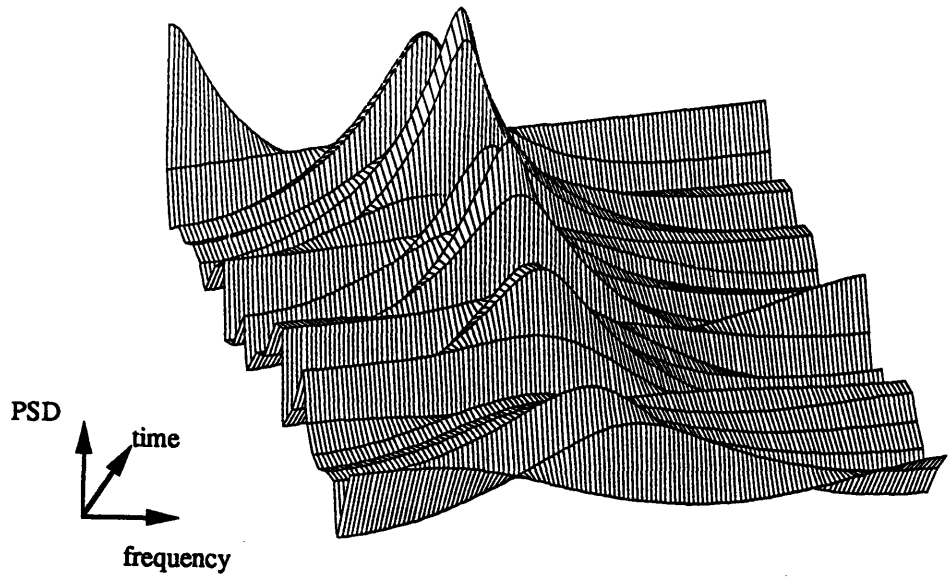


Figure 33. The estimator from the previous two Fig. applied to a 0 dB. The low SNR results in a decrease of the performance.

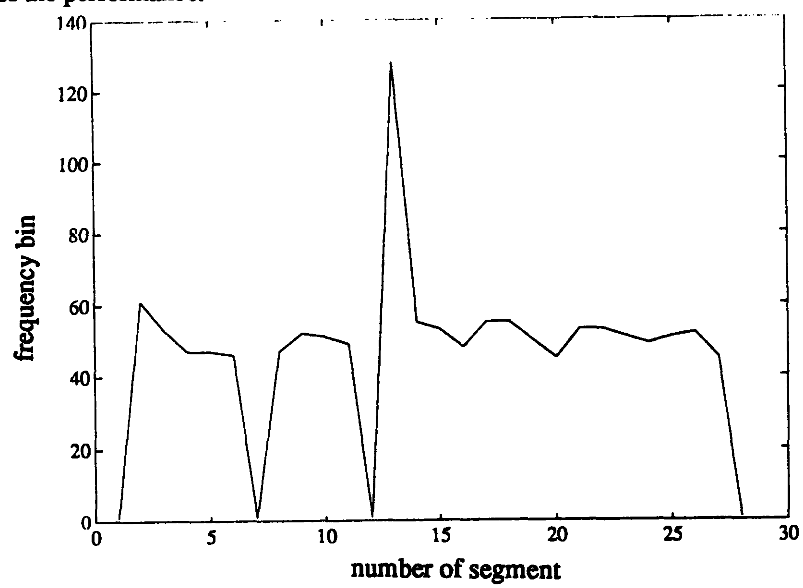


Figure 34. The location of the spectral peaks for the Modified Covariance Algorithm and a 0 dB LDA signal (previous Fig.) shows that the Doppler frequency is not resolved very well.

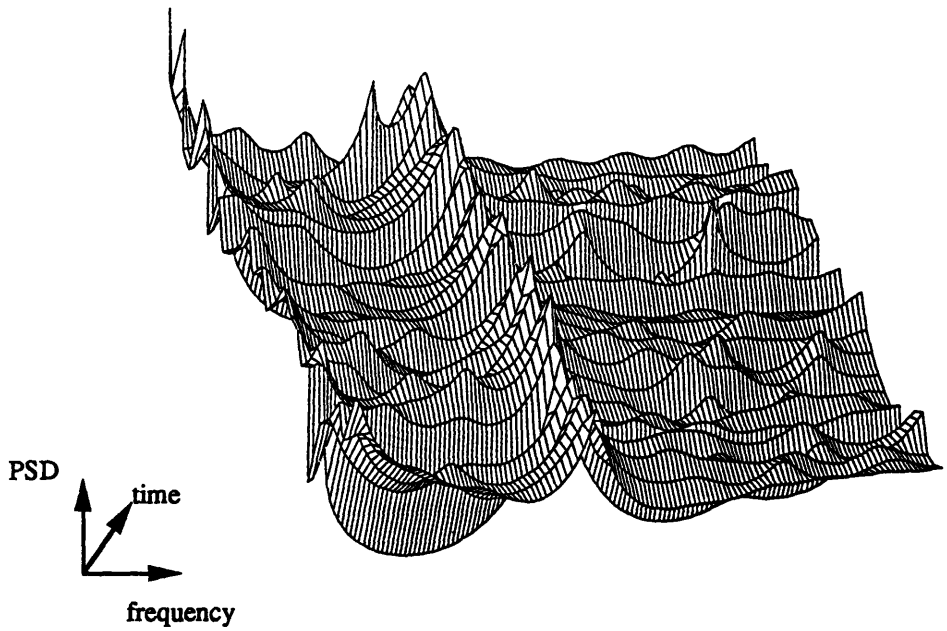


Figure 35. A 20th order AR estimate for a 2000 dB signal again with 128-point segments demonstrates that too high a model order results in spurious peaks in the spectrum. The variance in the estimate increases.

Appendix 3.3: Results of the Iterative Filtering Algorithm

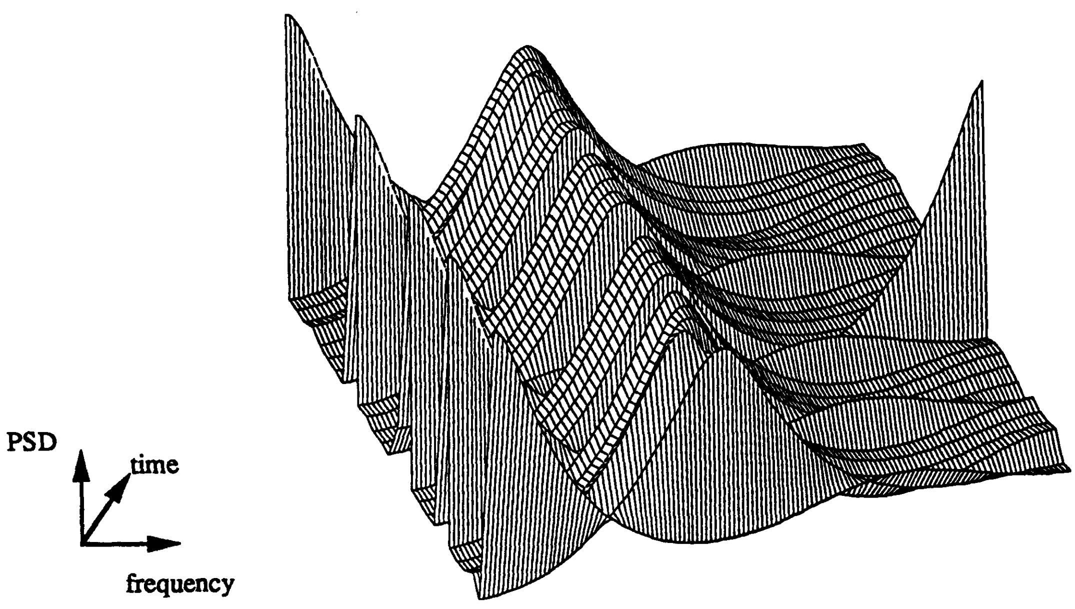


Figure 36. IFA with a third order AR Modified Covariance Agorithm after 8 iterations. The SNR of the signal is 0 dB. The spectrogram is very smooth and of very low variance.

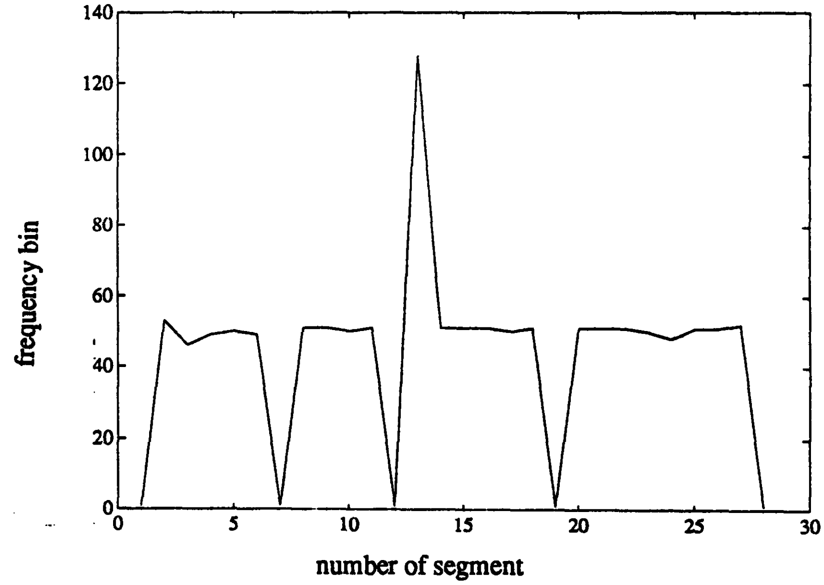


Figure 37. The spectral peaks in the previous Fig. correspond to the Doppler frequqncies. The IFA failed at very low local SNR.

Appendix 3.4: Results of the Pade Estimator Using the Common Euclidean Algorithm

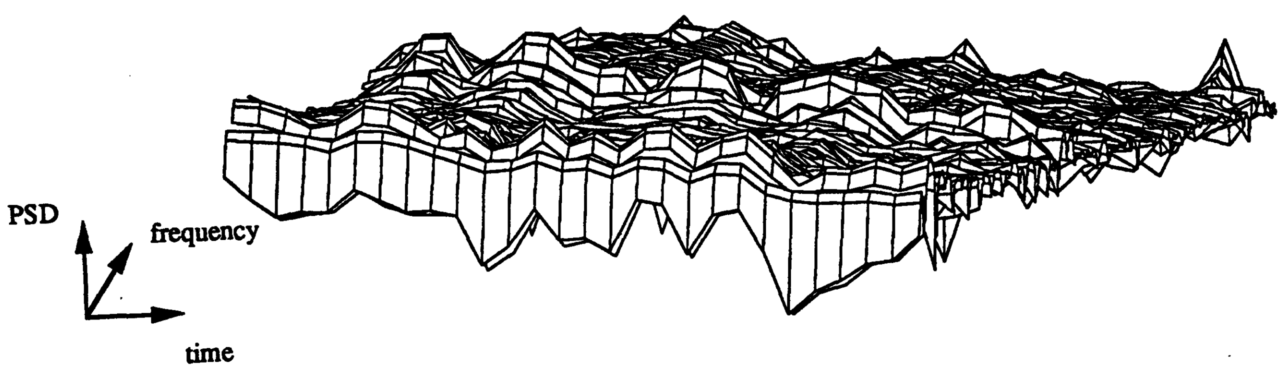


Figure 38. Pade estimator applied to a 10 dB SNR signal. The order of the AR branch is 4. Again, 128 point segements overlapping by 64 points were taken. The variance in the spectra is higher than for purely auto-regressive algorithms but still smaller than for the classical methods.

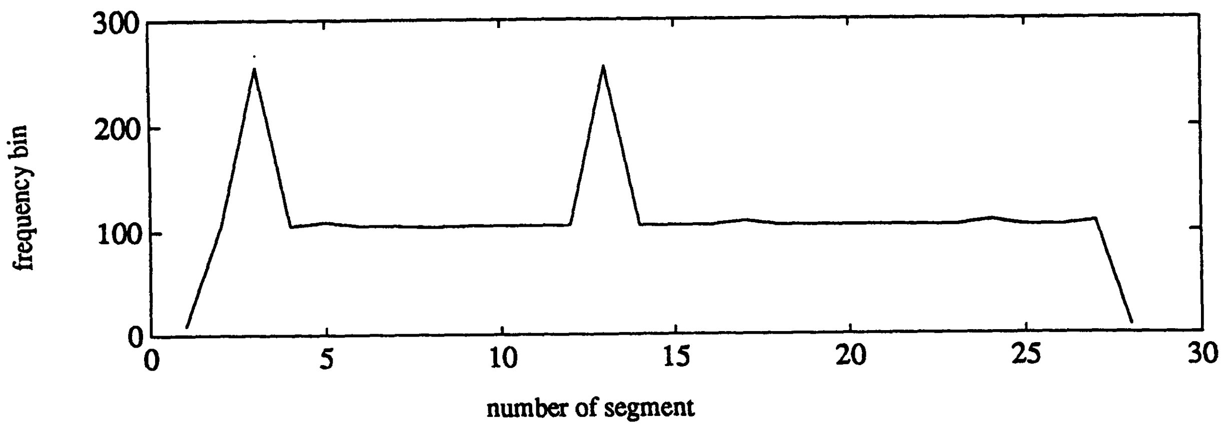


Figure 39. The location of the spectral peaks of the previous Fig. corresponds closely to the Doppler frequency. The estimator has fewer drop-outs at the very low local SNRs indicating a smaller noise sensitivity.

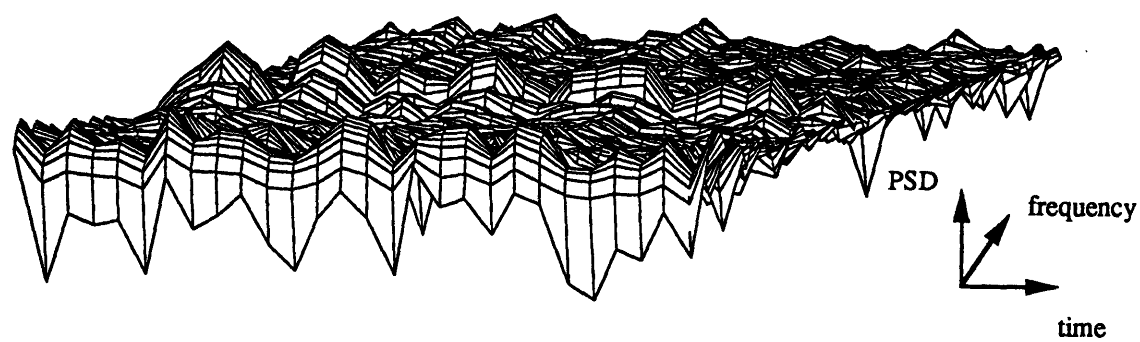


Figure 40. Generally, if shorter data segments were taken, the performance of the auto-regressive models decreased. The Pade estimator seemed to be less sensitive in this case. This Figure shows the performance with 64-point segments overlapped by 50 %. SNR is 10 dB, order of the AR branch is 4.

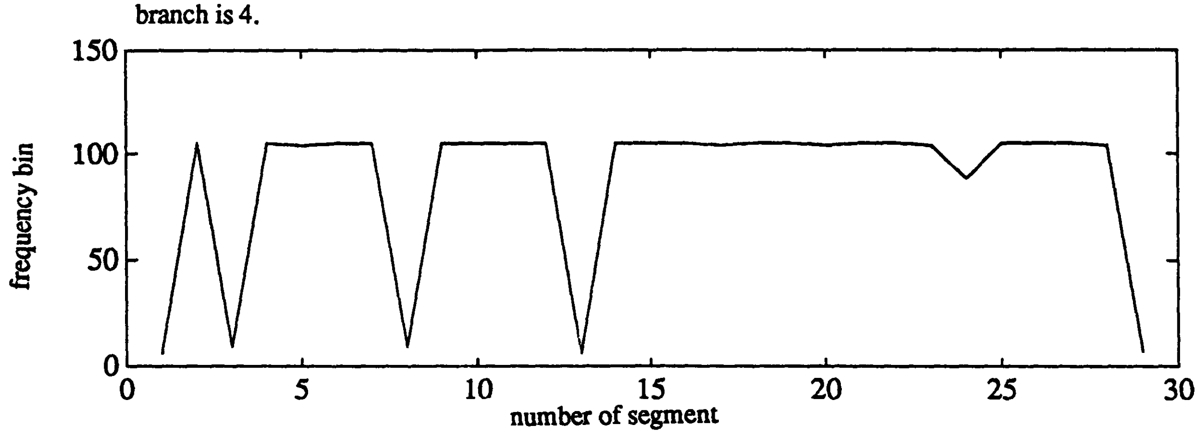


Figure 41. In the case of the shorter data segments of the previous Figure the Doppler frequency is still resolved very well. However, more drop-outs occur at low local SNR.

Appendix 4: Experimental Results

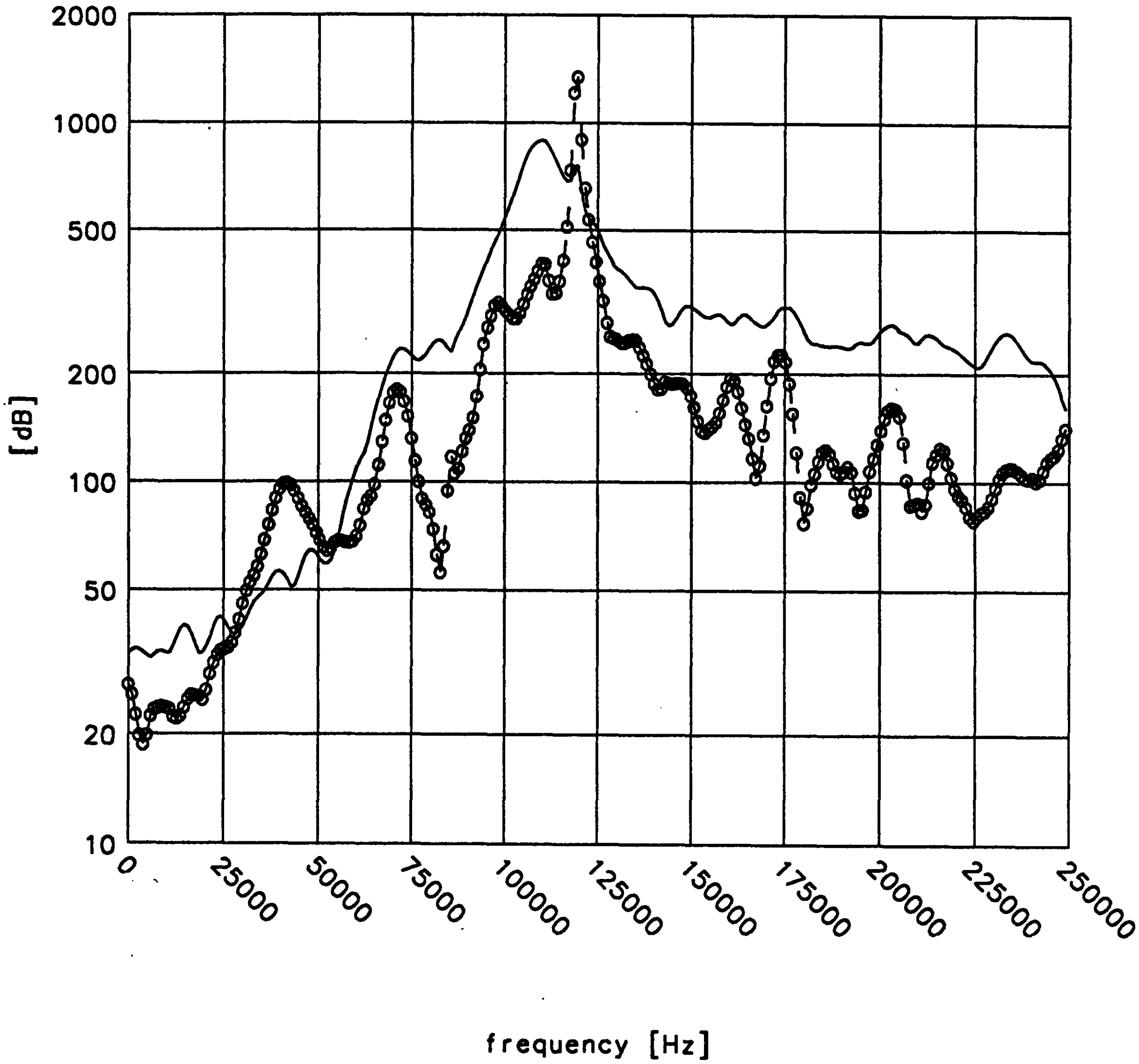


Figure 42. Spectrum of Pade estimator after digital lowpass filtering with `FilterData`. Comparison with Figs. 24 and 25 shows that the low frequncies are indeed attenuated.

Appendix 5: Programs for Processing LDA Signals

Appendix 5.1: `#include` file `FileOp.h` for File Operations

```
#ifndef FILEOP

#define FILEOP
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

/*
 *      Open a file for write access
 */
#define FpOpenW(FileName, FilePointer)                                        \
                if ( (FilePointer = fopen(FileName,"w")) == NULL )       \
                {                                                             \
                        perror();                                             \
                        fprintf(stderr, "%%%%ERROR: Could not open %s\n", FileName);    \
                        exit(-1);                                             \
                }
/*
 *      open a file for read access
 */
#define FpOpenR(FileName, FilePointer)                                        \
                if ( (FilePointer = fopen(FileName,"r")) == NULL )       \
                {                                                             \
                        perror();                                             \
                        fprintf(stderr, "%%%%ERROR: Could not open %s\n", FileName);    \
                        exit(-1);                                             \
                }


/*
 *      open a file for read and write access
 */
#define FpOpenRW(FileName, FilePointer)                                       \
                if ( (FilePointer = fopen(FileName,"rw")) == NULL )           \
                {                                                             \
                        perror();                                             \
                        fprintf(stderr, "%%%%ERROR: Could not open %s\n", FileName);    \
                        exit(-1);                                             \
                }


/*
 *      Open a file for write and read access, update
 */
#define FpOpenRWU(FileName, FilePointer)                                 \
                if ( (FilePointer = fopen(FileName,"r+")) == NULL )           \
                {                                                             \
                        perror();                                             \
                        fprintf(stderr, "%%%%ERROR: Could not open %s\n", FileName);    \
                        exit(-1);                                             \
                }
/*
 *      Open a file with system calls for read access
 */
#define FdOpenR(FileName, FileDescriptor)                                \
                if ( (FileDescriptor = open(FileName,O_RDONLY)) == -1 )       \
                {                                                             \
                        perror();                                             \
                        fprintf(stderr, "%%%%ERROR: Could not open %s, errno: %d\n", FileName, errno);
                        exit(-1);                                             \
                }


/*
```

```
 *      Open a file with system calls for read and write access
 */
#define FdOpenRW(FileName, FileDescriptor)                                      \
                if ( (FileDescriptor = open(FileName,O_RDWR) ) == -1)                   \
                {                                                                       \
                        perror();                                                       \
                        fprintf(stderr, "%%%%ERROR: Could not open %s, errno: %d\n", FileName, errno);
                        exit(-1);                                                       \
                }

/*
 *      Open a contiguous file for write access
 */
#define FdOpenCW(FileName, FileDescriptor, FileSize)                                    \
                if ( (FileDescriptor = open(FileName,O_WRONLY | O_CREAT | O_CTG, 0600, FileSize) ) == -1)
                {                                                                       \
                        perror();                                                       \
                        fprintf(stderr, "%%%%ERROR: Could not open %s, errno: %d\n", FileName, errno);
                        exit(-1);                                                       \
                }

/*
 *      Open a file for write access using system calls
 */
#define FdOpenW(FileName, FileDescriptor)                                       \
                if ( (FileDescriptor = creat(FileName, 0666) ) == -1)                   \
                {                                                                       \
                        perror();                                                       \
                        fprintf(stderr, "%%%%ERROR: Could not open %s, errno: %d\n", FileName, errno);
                        exit(-1);                                                       \
                }


/*
 *      Write to a file (buffered)
 */
#define FpWrite(FilePtr, ArrayPtr, NItems)                                      \
                if (fwrite( (char *)ArrayPtr, sizeof(*ArrayPtr), NItems, FilePtr ) != NItems || ferror(Fi
                {                                                                       \
                        perror();                                                       \
                        fprintf(stderr, "%%%%ERROR: Write error, errno: %d\n", errno);  \
                        exit(-1);                                                       \
                }


/*
 *      Read from a file (buffered)
 */
#define FpRead(FilePtr, ArrayPtr, NItems)                                       \
                if (fread( (char *)ArrayPtr, sizeof(*ArrayPtr), NItems, FilePtr ) != NItems || ferror(Fil
                {                                                                       \
                        perror();                                                       \
                        fprintf(stderr, "%%%%ERROR: Read error, errno: %d\n", errno);   \
                        exit(-1);                                                       \
                }


/*
 *      Read to a file (unbuffered)
 */
#define FdRead(FileDescriptor, ArrayPtr, BytesToRead)                                   \
                if (BytesToRead != read(FileDescriptor,(char *)ArrayPtr, BytesToRead))
                {                                                                       \
                        perror();                                                       \
                        fprintf(stderr, "%%%%ERROR: Read error, errno: %d\n", errno);   \
                        exit(-1);                                                       \
                }


/*
 *      Write to a file (unbuffered)
 */
#define FdWrite(FileDescriptor, ArrayPtr, BytesToWrite)                                 \
                if (BytesToWrite != write (FileDescriptor, (char *)ArrayPtr, BytesToWrite)) \
```

```
                {                                                                       \
                        perror();                                                       \
                        fprintf(stderr, "%%%%ERROR: Write error, errno: %d\n", errno);  \
                        exit(-1);                                                       \
                }



/*
 *      Get current position in file
 */
#define FpGetPos(FilePointer, Position)                                                 \
                Position = ftell(FilePointer);                                          \
                if (errno)                                                              \
                {                                                                       \
                        fprintf(stderr,"Cannot get position, errno: %d\n", errno);  \
                        perror();                                                       \
                        exit(-1);                                                       \
                }                                                                       \



/*
 *      Set current position in file
 */
#define FpSetPos(FilePointer, Position, StartingFrom)                                   \
                fseek(FilePointer, Position, StartingFrom);                             \
                if (errno)                                                              \
                {                                                                       \
                        fprintf(stderr,"Cannot reposition, errno: %d\n", errno);    \
                        perror();                                                       \
                        exit(-1);                                                       \
                }                                                                       \



#define FdSetPos(FileDescriptor, Position, StartingFrom)                        \
        if (lseek(FileDescriptor, (long)Position, StartingFrom) == (-1))        \
        {                                                                               \
                fprintf(stderr,"Cannot move file pointer, errno: %d\n",errno);         \
                perror();                                                               \
                exit(-1);                                                               \
        }

#endif
```

Appendix 5.2: `SampleData` - Program for Data Acquisition

```
/************************************************************************
 *
 *                      SampleData.c
 *
 ************************************************************************
 *
 *
 *  DESCRIPTION
 *    This program samples data from the AD12F and writes them to a file
 *    Due to restricted contiguous disk space and the poor overall
 *    performance of the MASSCOMP at the highest sampling frequencies
 *    we have to take a suboptimal approach, which does not permit
 *    continuous sampling over long periods:
 *    We fill one buffer at a time (of the order of a MB, that's all the
 *    system permits) write it to disk while the sampling is suspended
 *    and then continue until an amount of data has been accumulated
 *    corresponding to the total desired sampling duration.
 *    Let's hope that there is not too much information in the breaks
 *    between the sampling...
 *
 *  USAGE
```

```
 *    SampleData -o <Output File>
 *               -r <Remote File>
 *               -t <Sampling Duration>
 *               -f <Sampling frequency>
 *               -B <Buffer size in items (short)>
 *               -G <A/D converter gain>
 *               -h  Print information about usage
 *
 *  DEFAULTS
 *   Output File:       /usr/data/erk/Direct.dat
 *   Remote File: /usr/erk/DSP/DAT/RawData.dat
 *   Sampling Duration:  410                 [msec]
 *   Sampling frequency: 1                   [MHz]
 *   Buffer size: 100 * 4096       [items short]
 *   A/D converter gain:  0                  [1x]
 *
 *  Default sampling duration and buffer size were chosen such that
 *  one buffer is filled during the whole sampling duration
 */

#include <stdio.h>
#include <aplib.h>
#include <fcntl.h>
#include <errno.h>
#include <math.h>
#include <mrerrs.h>
#include "/usr/erk/DSP/FileOp.h"

#define reg1 register
#define reg2 register
#define reg3 register
#define reg4 register

typedef int bool;

void exit();
void is_an_error();
void perror();
double atof();
char *strcat();
char *malloc();

/*
 *       real-time priority for process
 */
#define knRealTime  -20

/*
 *       Parameters for setting up the A/D clock
 */
#define SQUARE           4
#define LOW              0
#define NEAREST          0

/*
 *       Input channel numbers
 */
#define CHANNEL_0    0
#define CHANNEL_1    1
#define CHANNEL_2    2
#define CHANEL_5

/*
 *       Actions taken by data acquisition upon error
 */
#define IMMEDIATELY 1
#define FOR_REUSE   0

/*
------------------------------------------------------------------------
 */

void Usage()
{
```

```
        fprintf(stderr,"\n");
        fprintf(stderr,"USAGE:\n\n");
        fprintf(stderr,"-o <output file>\n");
        fprintf(stderr,"\t[DEFAULT: /usr/data/erk/Direct.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-f <sampling frequency>\n");
        fprintf(stderr,"\t[DEFAULT: 1.0e6 (Hz)]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-r <remote file name>\n");
        fprintf(stderr,"\t[DEFAULT: /usr/erk/DSP/DAT/RawData.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-t <sampling time>\n");
        fprintf(stderr,"\t[DEFAULT: 410 (msec)]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-B <Buffer size in items short>\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-G <AD12F amplifier gain [-1:3]>\n");
        fprintf(stderr,"\t[DEFAULT: 0]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-h print this message\n");
        fprintf(stderr,"\n");

        exit(-1);
}

/*
------------------------------------------------------------------------
 */


main(argc, argv)
int argc;
char **argv;
{
        static short *pasData, *pasDataSAV, *pasReturnedBuf;

        float   fSampDur = 410.00/1000.00;

        double  fSampFreq = 1.0e6;
        double  fRFreq, fRWidth;

        int     i,j;

        int     nAdPathNo = -1;
        int     nClkPathNo = -1;

        int     nGain = 0;

        int     chOption;

        int     nSamples;
        int     nFileSize;
        int     nItemsWritten = 0;
        int     nItemsLeft;
        int     nLoops;

        int     nItemsInBuf = 100 * 4096;
        int     nBytesInBuf;

        static char     achCommand[100] = "rcp ";
        static char     *pachOutputFile = "/usr/data/erk/Direct.dat";
        static char     *pachCommand;
        static char     *pachMorgana = " morgana:";
        static char     *pachMorganaFile = "/usr/erk/DSP/DAT/RawData.dat";

        extern char *optarg;
        extern int optind;

        int     fdOutput;
        int     *pnDummy;

        pachCommand = &achCommand[0];

/*---------------------------------------------------------------------*/
```

```
	if (14 < argc)
	{
		fprintf(stderr,"Too many options\n\n");
		Usage();
	}

	while ((chOption = getopt(argc, argv, "o:f:r:t:B:G:h")) != EOF)
	{
		switch (chOption)
		{
			case 'o':
				pachOutputFile = optarg;
				break;

			case 'r':
				pachMorganaFile  = optarg;
				break;
			case 'f':
				fSampFreq        = atof(optarg);
				break;

			case 't':
				fSampDur = atof(optarg) / 1000.00;
				break;

			case 'B':
				nItemsInBuf      = atoi(optarg);
				break;

			case 'G':
				nGain            = atoi(optarg);
				break;

			case 'h':
				Usage();
				break;

			case '?':
				Usage();
				break;
		}

	}

/*------------------------------------------------------------------*/
	/*
	 *	Get real-time priority
	 */

	if ( (int) (nice(knRealTime)) != knRealTime )
	{
		fprintf(stderr, "\nGot different priority than requested, errno: %d\n", errno);
		perror();
		exit(-1);
	}

	if ( (nGain<-1) || (3<nGain) )
	{
		fprintf(stderr,"\nWrong gain specified\n");
		Usage();
	}
/*------------------------------------------------------------------*/

	nBytesInBuf = (nItemsInBuf<<1);

	/*
	 *	Allocate the A/D buffer
	 */
	pasDataSAV = (short*)malloc(nBytesInBuf+2);

	/*
	 *	alignment on long-word boundary
	 */
```

```
        pasData = (short *)((int)(pasDataSAV+2) & (~0x3));

/*---------------------------------------------------------------------*/
        /*
         *      Lock the buffer into memory
         */
        if (plockin(pasData, nBytesInBuf) == -1)
        {
                fprintf(stderr,"\nCannot lock buffers, errno: %d\n",errno);
                perror();
                exit(-1);
        }

        fprintf(stderr,"\n\nBuffer size: %d [bytes]\n",nBytesInBuf);
        fprintf(stderr,"\n%d bytes locked into memory\n", nBytesInBuf);

/*---------------------------------------------------------------------*/
        /*
         *      Open the Output file for unbuffered write access
         */
        FdOpenW(pachOutputFile, fdOutput)

        /*
         *      the number of samples to fetch, should fit into the
         *      buffer
         */

        nSamples = (int)(fSampDur * fSampFreq );
        nFileSize = (nSamples<<1);

        fprintf(stderr,"\n\nTotal sampling time:     %f [sec]\n", fSampDur);
        fprintf(stderr,"File size:               %d [bytes]\n",nFileSize);
        fprintf(stderr,"Total number of samples: %d \n",nSamples);

        /*
         *      The first number in the file is the number of
         *      samples contained therein
         */
        pnDummy = &nSamples;
        FdWrite(fdOutput,pnDummy,sizeof(int))


/*---------------------------------------------------------------------*/


        nLoops = (int) ( (float)nSamples / (float)nItemsInBuf);

        fprintf(stderr, "\n\n%d Loops necessary\n", nLoops);

        for (i=1; i <= nLoops; i++)
        {

                /*
                 *      open the A/D board
                 */
                mropen(&nAdPathNo, "/dev/dacp0/adf0",1);

                /*
                 *      we use clock 5 on the CK10 board
                 */
                mropen(&nClkPathNo,"/dev/dacp0/clk5",0);

                /*
                 *      set up clock speed
                 */
                mrclk1(nClkPathNo,NEAREST,fSampFreq, &fRFreq,SQUARE,0.0,&fRWidth,LOW);


                mrclktrig(nAdPathNo,1,nClkPathNo);

                /*
                 *      define the input channel
                 */
                mradinc(nAdPathNo, CHANNEL_0, 1, 0, nGain);
```

```

                /*
                 *      allocation of the buffers
                 */
                mrbufall(nAdPathNo, pasData, 1, nBytesInBuf);

                 mrxinq(nAdPathNo,nItemsInBuf,nItemsInBuf,0);

                mrbufwt(nAdPathNo,0);
                mrbufget(nAdPathNo,0,&pasReturnedBuf);

/***             for (j=0; j <= nItemsInBuf-1; j++)
 ***                     printf("%d\n",*(pasReturnedBuf+j));
 ***/

                 FdWrite(fdOutput,pasReturnedBuf,sizeof(short)*nItemsInBuf)

                 nItemsWritten += nItemsInBuf;

                 fprintf(stderr,"\n%d: %d Samples written to disk\n",i,nItemsWritten);

                mrclosall();

        }


        nItemsLeft = nSamples - nLoops * nItemsInBuf;

        if (nItemsLeft)
        {

                mropen(&nAdPathNo, "/dev/dacp0/adf0",1);

                mropen(&nClkPathNo,"/dev/dacp0/clk5",0);

                mrclk1(nClkPathNo,NEAREST,fSampFreq, &fRFreq,SQUARE,0.0,&fRWidth,LOW);

                mrclktrig(nAdPathNo,1,nClkPathNo);

                mradinc(nAdPathNo, CHANNEL_0, 1, 0, nGain);

                mrbufall(nAdPathNo, pasData, 1, nBytesInBuf);


                mrxinq(nAdPathNo,nItemsInBuf,nItemsLeft,0);

                mrbufwt(nAdPathNo,0);

                mrbufget(nAdPathNo,0,&pasReturnedBuf);

/***            for (j=0; j <= nItemsInBuf-1; j++)
 ***                    printf("%d\n",*(pasReturnedBuf+j));
 ***/
                FdWrite(fdOutput,pasReturnedBuf,sizeof(short)*nItemsLeft);

                nItemsWritten += nItemsLeft;

        }

        fprintf(stderr,"\nTotal: %d items written to disk\n\n", nItemsWritten);

/***    fclose(fpOutput);
 ***/

        pachCommand = strcat(pachCommand, pachOutputFile);
        pachCommand = strcat(pachCommand, pachMorgana);
        pachCommand = strcat(pachCommand, pachMorganaFile);

        fprintf(stderr, "\n>>>>Transfer to morgana<<<<\n");

        system(pachCommand);

        exit(0);
}
```

Appendix 5.3: `FilterData` - Program for Filtering Input Data with FIR Filter

```
/*************************************************************************
*
*                           FILTERDATA.C
*
**************************************************************************
*
* DESCRIPTION
*   This routine filters a string of data with an FIR filter.
*   The filtering is done in the frequency domain using the overlap-save
*   method. This avoids unnecessary calculations usually done if the
*   filter order is much smaller than the length of the data record.
*
*   For creating the FIR Filter see the routine Create_FIR.c.
*   The input data are assumed to be short integers, i.e. to come
*   unmodified from the A/D board.
*
*   This routine becomes more efficient the higher the order of the
*   FIR filter. For low-order FIR filter use the routine AP_conv.c
*   which performs a linear convolution in the time domain. I would say
*   the limit is about FIR_order = 20.
*
* USAGE
*   FilterData -o <Output File Name>
*              -i <Input File Name>
*              -v <rms File Name>
*              -H <Filter File Name>
*              -h
*
* OPTIONS
*   Output File Name: Name of the file which will contain the filtered data
*   Input File Name:  Name of the file containing the raw data from the A/D
*   Filter File Name: Name of the file containing the frequency response
*                        of an FIR filter (cf. CreateFIR.c for the format)
*   rms File Name:    Name of the file containing the root-mean-square of
*                        of trhe filtered data
*
* DEFAULTS
*   Output File Name: /usr/erk/DSP/DAT/Filtered.dat
*   Input  File Name: /usr/erk/DSP/DAT/RawData.c
*   Filter File Name: /usr/erk/DSP/DAT/FIR.dat
*   rms    File Name: /usr/erk/DSP/DAT/Threshold.dat
*
* COMPILING OPTIONS
*
*  -DVARIANCE
*    The mean square of the filtered data is returned to the environment.
*    If filtered with a low pass filter (zero mean) this is equal to the
*    variance of the filtered data
*
*/

#include <stdio.h>
#include <math.h>
#include <aplib.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include "/usr/erk/DSP/FileOp.h"

typedef int vector;
typedef int bool;

```

```
#define reg1 register
#define reg2 register
#define reg3 register
#define reg4 register
#define reg5 register
#define reg6 register

void perror();
void exit();
char *malloc();
long lseek();

#ifdef DEBUG
#define DUMP(Y,length)    mapsyncdma(-1,VA0);                                \
                          mapstrfv(Y,1,pafFilterData,4,length);      \
                          mapbwaitdma(VA0);                                  \
                          for (i=0; i<= length-1; i++)                       \
                                  printf("%f \n", pafFilterData[i]);         \
                          exit(0);


#define MAGC(Y,y_len)     mapsyncmath(-1,VA0);                       \
                          mapnrmsqcfv(Y,2,Y,1,y_len);                \
                          DUMP(Y,y_len)                              \
                          exit(0);
#endif

#define knMaxFFTLen       1024
#define knMaxLogLen       10

#define knPageLen         4096
#define knNumPages        20
#define knArraySize       knNumPages * knPageLen

#define knfUpperBound     4100.0

#define knRealTime        -20
/*
-------------------------------------------------------------------------------
*/

void Usage()
{
        fprintf(stderr,"\n");
        fprintf(stderr,"This program filters data of format short with a FIR filter \n");
        fprintf(stderr,"using the overlap-add method\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"USAGE:\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-i\tInput file containing the data as short (unformatted)\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/RawData.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-o\tOutput file containing the filtered data as float (unformatted)\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Filtered.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-H\tFile containig the FIR filter frequency response\n");
        fprintf(stderr,"\t\t see CreateFIR.c for format of this file\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/FIR.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-h\tPrint this message\n");
        fprintf(stderr,"\n");

        exit(-1);
}

/*
===============================================================================
*/

main(argc,argv)
int argc;
char **argv;
{
        vector   vTemp1, vTemp2;
```

```
	vector	vnScal1, vnScal2, vnScal3;
	vector	vCoeff, vH;
	vector	vData;
	vector	vAdd;
	vector	vFFTRes, vIFFTRes, vIFFTAux;

	int	chOption;

	int	*pnDummy;

/***	int	nRetriedAlready = 0;
 ***	int	nBytesToRead;
 ***	float	nfMaxValue;
 ***	float	nfUpperLimit;
 ***	reg int nBytesRead = 0;
 ***/


	int	nItemsRead = 0;
	int	nOutputFileSize;
	int	nItemsInFile;

	int	nOrderFIR;
	int	nLenFFT;
	int	nLogLen;
	int	nHalfLenFFT;

	int	nLoops;
	int	j;

	float	fScale;

	int	fdInput;
	int	fdOutput;

	FILE	*fpFilter;

	reg1 int nItemsWritten = 0;
	reg2 int nSegLen;
	reg3 int nRemLen;
	reg4 int i;

	static char *pachOutputFile	= "/usr/erk/DSP/DAT/Filtered.dat";
	static char *pachInputFile	= "/usr/erk/DSP/DAT/RawData.dat";
	static char *pachFilterFile	= "/usr/erk/DSP/DAT/FIR.dat";

	static short *pasRawData;
	static short *pasRawDataDummy;
	static float *pafFilterData;
	static float *pafFreqResp;
	static float nfVar;

	extern char	*optarg;
	extern int	optind;

	if (5 < argc)
	{
		fprintf(stderr," Too many options\n\n");
		Usage();
	}

	while ((chOption = getopt(argc, argv, "hi:o:H:n:")) != EOF)
	{
		switch (chOption)
		{
			case 'h':
				Usage();
				break;

			case 'i':
				pachInputFile = optarg;
				break;

			case 'o':
```

```
					pachOutputFile = optarg;
					break;

				case 'H':
					pachFilterFile = optarg;
					break;


				case '?':
					Usage();
					break;
			}
		}
/*-----------------------------------------------------------------*/
		/*
		 *	Get Real-time priority
		 */
		if ( (int) (nice(knRealTime)) != knRealTime )
		{
			fprintf(stderr, "Got different priority than requested, errno: %d", errno);
			perror();
			exit(-1);
		}

/*-----------------------------------------------------------------*/
		/*
		 *	Open the input and the output data files
		 */

/***
 ***	FpOpenR(pachInputFile, fpInput)
 ***/
		FdOpenR(pachInputFile, fdInput)

/***	FpOpenW(pachOutputFile, fpOutput)
 ***/

		FdOpenW(pachOutputFile, fdOutput)

		/*
		 *	the first entry in the input file is the number of samples
		 *	contained therein
		 */
/***
 ***	fscanf(fpInput,"%d:\n", &nItemsInFile);
 ***/
		pnDummy = &nItemsInFile;

		FdRead(fdInput,pnDummy,sizeof(int))

		fprintf(stderr, "\n Input file %s contains %d samples\n",pachInputFile, nItemsInFile);

/*-----------------------------------------------------------------*/
		/*
		 *	Open the file containing the frequency response
		 */

		FpOpenR(pachFilterFile, fpFilter)

		fscanf(fpFilter,"%d:%d\n", &nOrderFIR, &nLenFFT);

		nLogLen	= mapilog2(nLenFFT);
		nHalfLenFFT	= (nLenFFT>>1);
		nSegLen		= nLenFFT - (nOrderFIR - 1);
		fScale		= 1.0 / (float)(nLenFFT << 1);

		/*
		 *	the first item in the output file is the number of items in there
		 */
		nOutputFileSize = nItemsInFile + nOrderFIR - 1;

/***	fprintf(fpOutput,"%d:\n", nOutputFileSize);
 ***/

```

```
        pnDummy = &nOutputFileSize;
        FdWrite(fdOutput,pnDummy,sizeof(int))

        fprintf(stderr, "\n Filter file %s contains an FIR filter of order %d\n",pachFilterFile, nOrderFIR)
        fprintf(stderr, "\n Length of the FFTs will be %d,\tlength of a segment %d\n",nLenFFT, nSegLen);
/*--------------------------------------------------------------------*/
        /*
         *      Allocate memory for:
         *              - frequency response of FIR filter
         *              - array containing the filtered data
         *              - array containing the raw data
         */

        pafFreqResp      = (float *)malloc( sizeof(float) * (nLenFFT+2) );

        pafFilterData    = (float *)malloc( sizeof(float) * nSegLen );

        pasRawDataDummy  = (short *)malloc( sizeof(short) * nSegLen + 2);
        pasRawData       = (short *)((int)(pasRawDataDummy+2) & (~0x3));

/*--------------------------------------------------------------------*/
        /*
         *      Initializing the VA
         */

        mapinitva(1,1,0);

/***    if (plockin(&nfTempVar,4)==-1)
 ***    {
 ***            perror("FilterData:");
 ***            fprintf(stderr, "Cannot lock nfTempVar. errno: ", errno);
 ***    }
 ***/

        vData    = 0;
        vTemp1   = vData  + (nLenFFT<<1);
        vTemp2   = vTemp1 + (nLenFFT<<1);
        vCoeff   = vTemp2 + (nLenFFT<<1);
        vH       = vCoeff + nLenFFT + 2;
        vnScal1  = vH + nLenFFT + 2;
        vnScal2  = vnScal1+ 1;
        vnScal3  = vnScal2+ 1;
        vAdd     = vnScal3+ 1;     /* contains (nOrderFIR-1) elements */

/*-----------------------------------------------------------------------*/
        /*
         *      Read in the frequency response of the filter
         */

        for (i=0; i<= (nLenFFT + 1); i++)
                fscanf(fpFilter, "%f\n", pafFreqResp+i );

        maplodfv(pafFreqResp,4,vH,1,nLenFFT+2);

        maprffttab(vCoeff, nLogLen);

        maplodfs(&fScale,vnScal1);

/*-----------------------------------------------------------------------*/
        /*
         *      Determine the location of the FFT and IFFT results
         */

        if (nLogLen&1)
        {
                vFFTRes   = vTemp1;
                vIFFTAux = vData;
                vIFFTRes = vFFTRes;
        } else {

                vFFTRes   = vData;
                vIFFTAux = vTemp1;
                vIFFTRes = vIFFTAux;
        }
```

```
/*----------------------------------------------------------------------*/

        nLoops = (int) ( (float)nItemsInFile/(float)nSegLen );

        fprintf(stderr, "\n%d loops necessary\n", nLoops);


        mapclrfv(vAdd,1, nOrderFIR-1);
        mapclrfv(vTemp2,1,nSegLen);

        for (j=1; j <= nLoops; j++)
        {
/***            for (i=0; i<=nSegLen-1; i++)
 ***                    fscanf(fpInput, "%f\n", (pasRawData+i));
 ***/
                FdRead(fdInput,pasRawData,nSegLen*sizeof(short));

                nItemsRead += nSegLen;

                /*
                 *      Shows whether data are read correctly
                 *      comment out if necessary
                 *
                 ***     for (i=0; i<nSegLen; i++)
                 ***    printf("%d \n", *(pasRawData+i) );
                 ***    exit(0);
                 ***/



                /*
                 *  Clear the vector (performs zero padding)
                 *      Get the data, convert them to float and filter
                 */

                mapclrfv(vData, 1, nLenFFT);

                mapsyncdma(-1,VA0);

                maplodwv(pasRawData, 2, vData, 1, nSegLen);

                mapsyncmath(-1,VA0);
                mapcvtifv(vData,1,vData,1,nSegLen);
                maprfftnc(vData,1,vCoeff,2,vTemp1,1,nLenFFT);

                /*
                 *      Result of the Fourier transform in vFFTRes
                 */

                mapmulcfvv(vFFTRes,2,vH,2,vFFTRes,2,nHalfLenFFT + 1);

                mapirfftnc(vFFTRes,2,vCoeff,2,vIFFTAux,2,nLenFFT);

                /*
                 *      Result of the inverse Fourier transform in vIFFTRes
                 */

                mapaddfvv(vIFFTRes,1,vAdd,1,vIFFTRes,1,nOrderFIR-1);

                mapmulfsv(vnScall, vIFFTRes, 1, vIFFTRes, 1, nSegLen);

                mapcopfv(vIFFTRes+nSegLen,1,vAdd,1,nOrderFIR-1);

                mapsyncdma(-1,VA0);

                mapstrfv(vIFFTRes,1, pafFilterData, 4, nSegLen);



                mapbwaitdma(VA0);

/***            for (i=0; i<=nSegLen-1; i++)
 ***                    fprintf(fpOutput,"%f\n",*(pafFilterData+i));
```

```
***/
                FdWrite(fdOutput,pafFilterData,sizeof(float)*nSegLen)

                nItemsWritten += nSegLen;


        }

        nRemLen = nItemsInFile - nItemsRead;

        if (nRemLen)
        {

/***            for(i=0; i<= nRemLen-1; i++)
 ***                    fscanf(fpInput,"%f\n",(pasRawData+i));
 ***/
                FdRead(fdInput,pasRawData,nRemLen*sizeof(short));

                nItemsRead += nRemLen;

                mapclrfv(vData, 1, nLenFFT);

                mapsyncdma(-1,VA0);

                maplodwv( pasRawData, 2, vData, 1, nRemLen);

                mapsyncmath(-1,VA0);
                mapcvtifv(vData,1,vData,1,nRemLen);

                maprfftnc(vData,1,vCoeff,2,vTempl,1,nLenFFT);

                mapmulcfvv(vFFTRes,2,vH,2,vFFTRes,2,nHalfLenFFT + 1);

                mapirfftnc(vFFTRes,2,vCoeff,2,vIFFTAux,2,nLenFFT);

                mapaddfvv(vIFFTRes,1,vAdd,1,vIFFTRes,1,nOrderFIR-1);

                mapmulfsv(vnScall, vIFFTRes,1,vIFFTRes,1,nRemLen+nOrderFIR-1);

                mapsyncdma(-1,VA0);

                mapstrfv(vIFFTRes,1,pafFilterData,4,nRemLen+nOrderFIR-1);


                mapbwaitdma(VA0);

/***            for (i=0; i<=nRemLen+nOrderFIR-2; i++)
 ***                    fprintf(fpOutput,"%f\n",*(pafFilterData+i));
 ***/
                FdWrite(fdOutput,pafFilterData,sizeof(float)*(nRemLen+nOrderFIR-1))


                nItemsWritten += nRemLen + nOrderFIR - 1;

        } else {


                mapmulfsv(vnScall, vAdd,1,vAdd,1,nOrderFIR-1);

                mapsyncdma(-1,VA0);

                mapstrfv(vAdd,1,pafFilterData, 4, nOrderFIR - 1);


                mapbwaitdma(VA0);

/***            for (i=0; i<=nOrderFIR-2; i++)
 ***                    fprintf(fpOutput,"%f\n",*(pafFilterData+i));
 ***/
                FdWrite(fdOutput,pafFilterData, sizeof(float)*(nOrderFIR-1));


                nItemsWritten += nOrderFIR - 1;
        }
```

```
        fprintf(stderr, "\n%d items read from file %s\n", nItemsRead, pachInputFile);
        fprintf(stderr, "%d items written to file %s\n", nItemsWritten, pachOutputFile);

/***
 ***   fclose(fpInput);
 ***   fclose(fpOutput);
 ***/

        close(fdInput);
        close(fdOutput);

        fclose(fpFilter);

        exit(0);
}

```

Appendix 5.4: `CreateFIR` - Program for FIR Filter Design

```
/*********************************************************************
 *                                                                   *
 *                          Create_FIR.c                             *
 *                                                                   *
 *********************************************************************
 *
 *  DESCRIPTION
 *    This program creates a finite impulse response (FIR) filter and
 *    write its frequency response into a file.
 *    At the moment the Kaiser window method (program KaiserFIR.c)
 *    and the Optimum FIR filter method (program OptFIR.f) are
 *    supported.
 *    Both programs require different specifications.
 *
 *    Kaiser window method:
 *       Specify edge frequencies (either 2 or 4) and tolerance.
 *       The tolerance must be uniform. The program returns the
 *       filter length which - with a certain range - satisfies
 *       the specifications. The Kaiser window design supports
 *       only low-, high-, and bandpass filter (edge is an array
 *       of four elements)
 *
 *     Optimum filter design:
 *       Input is an array of edge frequencies and a weighting
 *       function allowing for different tolerances in the filter
 *       bands. The filter length has to be specified, the tolerance
 *       has to be checked and the filter length eventually increased.
 *       The program takes its time compared with the Kaiser method,
 *       however, the shorter filter length is worth the effort.
 *       The program is the IEEE routine using the Parks-McClellan
 *       approach with the Remez exchamge algorithm. This is a FORTRAN
 *       program.
 *
 *  USAGE
 *    MakeAFilter  -t <Sampling Period>
 *                 -m <Design Method>
 *                    1 --> Kaiser
 *                    2 --> Optimum Filter Design
 *                 -l <length of impulse response [optimum filter design only]>
 *                 -o <Output File>
 *
 *  DEFAULT VALUES
 *    Sampling Period: 1.0e-6 [sec]
 *    Design Method:   Kaiser Window Method
 *    Output File:     /usr/erk/DSP/DAT/FIR.dat
 *    Filter length:   70
 *
 *  COMMENTS
```

```
 *    - I deliberately did not use option switches for the cut-off
 *      frequencies. The reason being that the filter design routines
 *      allow bandpass and bandstop (Kaiser) or multiple bandstop/pass
 *      (optimum filter design).
 *
 */

typedef int vector;
typedef int bool;

#include <aplib.h>
#include <errno.h>
#include <values.h>
#include <stdio.h>
#include "FilterSpecs.h"

void exit();
void is_an_error();
void opt_fir_();
double atof();

#define knMaxLenFFT     2048

#ifdef DEBUG

/*
 *      watch out for proper synchronisation of math and dma processor
 *      if using these routines
 */

#define DUMP(Y,y_len)     mapstrfv(Y,1,afImpulseResponse,4,y_len);\
                          mapbwaitdma();                            \
                          for (i=0; i <= y_len - 1; i++)     \
                                  printf("%f\n",afImpulseResponse[i]);



#define MAGC(Y,y_len)     mapnrmsqcfv(Y,2,Y,2,y_len>>1);            \
                          DUMP(Y, y_len);
#endif


#define FpOpenW(FilePointer, FileName)   if ( (FilePointer = fopen(FileName,"w")) == NULL )  \
                                          {                                                 \
                                          perror(errno);                                   \
                                          fprintf(stderr, "%%%%ERROR: Could not open %s\n", FileName); \
                                               exit(-1);     \
                                         }

/*
-----------------------------------------------------------------------
 */

void Usage()
{
        fprintf(stderr,"\n");
        fprintf(stderr,"This program creates the frequency response of a FIR Filter\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"USAGE:\n");
        fprintf(stderr,"-o file in which frequency response is to be stored\n");
        fprintf(stderr,"[Default: /usr/erk/DSP/DAT/FIR.dat]\n");
        fprintf(stderr,"\t Storage format: \n");
        fprintf(stderr,"\tlength of impulse response : ");
        fprintf(stderr,"length of FFT used\n");
        fprintf(stderr,"\tFrequency Response (one float per line)\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-m Method of filter design:\n");
        fprintf(stderr,"\t 1 uses Kaiser windows [Default]\n");
        fprintf(stderr,"\t 2 uses Optimum Filter Design\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-l Length of the Filter Impulse Response");
        fprintf(stderr,"\t only with optimum filter design");
        fprintf(stderr,"\t [Default:70]");
        fprintf(stderr,"\n");
```

```
        fprintf(stderr,"-t Sampling period [sec], [Default: 1.0e-6]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-F Length of the Fourier transform \n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-h Print this message\n");
        fprintf(stderr,"\n");

        exit(0);
}
/*
=====================================================================
 */


main(argc, argv)
int argc;
char **argv;
{
        int     nLenFFT = 1;
        int     nLogLen = 0;
        int     nMethod = 1;
        int     jtype=1, nbands, lgrid=0, neg=0;
        int     nOrderFIR = 71;
        int     nHalfOrderOfFIR;
        int     nTest;
        int     chOption;
        int     i;

        static char     *achOutputFileName = "/usr/erk/DSP/DAT/FIR.dat";

        vector  vH, vCoeff, vTemp1, vTemp2;

        static float    afImpulseResponse[knMaxFilterLen];
        static float    afFreqResponse[knMaxLenFFT+2];

        float   edge[20], fx[10], wtx[10], deviat[10];
        float   nfSampTime = 1.0e-6;

        double  nfDel;

        FILE *fpOutputFile;

        extern char     *optarg;
        extern int      optind;

        if ( 7 < argc)
                Usage();

        FpOpenW(fpOutputFile,achOutputFileName)

/*--------------------------------------------------------------------
   Organizing the vector memory
--------------------------------------------------------------------*/
        vTemp1  =  0;
        vTemp2  =  vTemp1 + (knMaxLenFFT<<1);
        vCoeff  =  vTemp2 + (knMaxLenFFT<<1);


        while ((chOption = getopt(argc, argv, "ho:m:t:l:F:")) != EOF)
        {
                switch (chOption)
                {
                        case 'h':
                                Usage();
                                break;

                        case 'l':
                                nOrderFIR = atoi(optarg);
                                break;

                        case 'o':
                                achOutputFileName = optarg;
                                break;

```

```
                        case 'm':
                                nMethod = atoi(optarg);
                                if ( (nMethod<1) || (nMethod>2) )
                                {
                                        fprintf(stderr, "Specify either (1) Kaiser Window or (2) Optimum
                                        exit(1);
                                }
                                break;

                        case 'F':
                                nLenFFT = atoi(optarg);
                                break;

                        case 't':
                                nfSampTime = atof(optarg);
                                break;

                        case '?':
                                Usage();
                                break;
                }
        }


        mapinitva(1,1,0);

        switch (nMethod)
        {
                /* Kaiser window nMethod */
        case 1:

                fprintf(stderr, "Creating Kaiser FIR filter ...");

                edge[0] = kfStopBandEdge;
                edge[1] = kfPassBandEdge;
                edge[2] = 0;
                edge[3] = 0;

                nfDel = kfPassBandDeviation;

                /* filter design routine */
                Kaiser_FIR(nfDel, edge, nfSampTime, afImpulseResponse, &nOrderFIR);

                break;

                /* Optimum filter design */
        case 2:

                fprintf(stderr, "\n Creating optimum filter \n");

                nbands = 2;

                /* the frequency bands */
                edge[0] = 0.0;
                edge[1] = 0.1;
                edge[2] = 0.15;
                edge[3] = 0.5;

                /* the weighting function */
                wtx[0]  = 10.0;
                wtx[1]  = 1.0;
/*              wtx[2]  = 20.0;*/

                /* desired filter frequency response */
                fx[0]   = 0.0;
                fx[1]   = 1.0;
/*              fx[2]   = 0.0;*/

                f_init();

                /* filter design routine */
                opt_fir_
                    (&nOrderFIR,&jtype,&nbands,&lgrid,edge,fx,wtx,&neg,afImpulseResponse,deviat);

```

```
        f_exit();

        if ((nOrderFIR) & 1)
            nHalfOrderOfFIR = ((nOrderFIR + 1) >> 1);
        else
            nHalfOrderOfFIR = (nOrderFIR >> 1);

        /*
         *      I had bad experiences with the reverse copy of
         *      vectors in VA/AP memory
         */


        if (neg)
        {
            for (i=0; i<=nHalfOrderOfFIR-1; i++)
            {
                afImpulseResponse[nOrderFIR-1-i] = (-1.0) * afImpulseResponse[i];
            }
        }
        else
        {
            for (i=0; i<=nHalfOrderOfFIR-1; i++)
            {
                afImpulseResponse[nOrderFIR-1-i] = afImpulseResponse[i];
            }
        }


        break;
    }

    fprintf(stderr, "\n\n Length of FIR Filter: %d \n", nOrderFIR);

    /*
     *      Determine the length of the Fourier transform for the frequency
     *      response of the FIR filter
     */

    if (nLenFFT < nOrderFIR)
    {
        nTest = nOrderFIR;

        while (nTest)
        {
            nTest >>= 1;
            nLogLen ++;
        }

        nLenFFT = (1<<nLogLen);

    }
    else
    {

        nTest = nLenFFT - 1;

        while (nTest)
        {
            nTest >>= 1;
            nLogLen ++;
        }

    }


    fprintf(stderr,"\nLength of FFTs will be: %d = 2**%d\n",nLenFFT,nLogLen);

    if (knMaxLenFFT < nLenFFT)
    {
        is_an_error("Create_FIR: filter too long for FFT\n\t--> change specs\n" , (-2));
        exit(-2);
    }
```

```
        mapclrfv(vTemp2,1,nLenFFT);
        mapsyncdma(-1,VA0);

        maplodfv(afImpulseResponse,4,vTemp2,1, nOrderFIR);


/*---------------------------------------------------------------------
   Create coefficient table for all subsequent FFTs
---------------------------------------------------------------------*/

        maprfftab(vCoeff, nLogLen);

/*---------------------------------------------------------------------
   Compute frequency response of the filter
---------------------------------------------------------------------*/
        maprfftnc(vTemp2,1,vCoeff,2,vTemp1,1,nLenFFT);


        if (nLogLen&1)
        {
                vH     = vTemp1;
                vTemp1 = vTemp2;
        } else {

                vH = vTemp2;
        }

        mapmulfsv(AP_OneHalf, vH,1,vH,1,nLenFFT+2);

        mapsyncdma(-1,VA0);

        mapstrfv(vH,1,afFreqResponse,4,nLenFFT+2);

/*---------------------------------------------------------------------
   filter frequency response now in vH
---------------------------------------------------------------------*/

        mapbwaitdma();

        /*
         * write the frequency response to the file
         */

        fprintf(fpOutputFile,"%d:%d\n",nOrderFIR, nLenFFT);

        for (i=0; i<= (nLenFFT+1); i++)
                fprintf (fpOutputFile,"%f\n",*(afFreqResponse+i));

#ifdef DEBUG
        for (i=0; i<= (nLenFFT+1); i += 2)
        {
                *(afFreqResponse+i) =
                *(afFreqResponse+i) * *(afFreqResponse+i) +
                *(afFreqResponse+i+1) * *(afFreqResponse+i+1);

                fprintf (stderr, "%f\n", *(afFreqResponse+i) );
        }
#endif

        fclose(fpOutputFile);

        mapfree(VA0);

        fprintf (stderr, "Filter frequency response in file %s\n",achOutputFileName);

        return (nOrderFIR);
}
```

Appendix 5.5: `KaiserFIR` - Routine for Kaiser Window Design

```
/****************************************************************
 *                                                              *
 *                         Kaiser_FIR.c                         *
 *                                                              *
 ****************************************************************
 *
 * DESCRIPTION
 *  Creates the frequency response of an FIR filter using the Kaiser
 *  window method.
 *  Either lowpass, hipass, or bandpass filters are possible, the
 *  filter type is selected according to the stop- and passband edge
 *  specifications.
 *
 *  The programm does not support at the moment multiple bandpass or
 *  bandstop filters.
 *
 *  An accurate description of the alorithm can be found in
 *  " Discrete-time Signal Processing" by Alan V. Oppenheim and
 *  Roland W. Schafer, Prentice-Hall Signal Processing Series
 *
 * SYNOPSIS
 *  int Kaiser_FIR(del, edges, T, h, order)
 *  double del, edges[], T;
 *  float h[];
 *  int *order;
 *
 *  PARAMETERS
 *     del  ... tolerance (uniform over all bands)
 *  edges[] ... array of edge frequencies (in [Hz])
 *      T   ... sampling rate (in [sec])
 *   *order ... resulting length of the filter
 *      h   ... pointer to impulse responseof filter
 *
 *  RETURN VALUES
 *      0   ... in ANY EVENT
 *
 */

void is_an_error();

#include <math.h>
#include <values.h>
#include <stdio.h>

#define MAX_LEN       256
#define DBL_EPSILON   1.0e-9

#define swap(a,b)    (temp) = (a); (a)=(b); (b)=(temp);
#define min(a,b)     ((a) < (b)) ? (a) : (b)

/*------------------------------------------------------------------
         modified Bessel function of the zeroth order
------------------------------------------------------------------*/

double i0(x)
double x;
{
        double s  = 0.0;
        double ds = 1.0;
        double f  = 0.0;
        double e;

        e = 0.25 * x *x;

        do
        {
                s  += ds;
                f  += 1.0;
                ds *= e / (f*f);
```

```
	}   while ( DBL_EPSILON <= (ds/s) );

	return s;
}

/*
-----------------------------------------------------------------
*/

int Kaiser_FIR(del, edge, T, h, order)
double del, T;
float h[], edge[];
int *order;
{
	double del_om, del_om_2, om_c, om_c_2, scale;
	double alpha, beta;
	double Nyq = 1.0 / (2.0 * T);
	double A;
	double i_beta;
	double temp;

	int M;
	int i;

	char LoPass = 0;
	char HiPass = 0;
	char BandPass = 0;

	if (!( (edge[2]) && (edge[3]) ))
	{
		if (edge[1] < edge[0])
		{
			LoPass = 1;
			fprintf(stderr, "\n Creating lowpass filter ... \n");

		} else {
			HiPass = 1;
			fprintf(stderr, "\n Creating hipass filter ... \n");
		}

	} else {
		if ( (edge[2] < edge[3]) || (edge[2] < edge[1]) )
		{
			is_an_error(" Kaiser_FIR: Conflict in bandpass edges: ", -2);
			return(-2);

		} else {
			if ( (edge[2] < edge[3]) && (edge[0] < edge[1]) )
			{
				BandPass = 1;
				fprintf(stderr, "\n Creating bandpass filter... \n");

			} else {
				is_an_error("Kaiser_FIR: Bandstop not supported: ", -3);
				return(-3);
			}
		}
	}
	if ( ((HiPass) && ( Nyq < edge[1])) || ((LoPass) && (Nyq < edge[0])) || ((BandPass) && (Nyq < edge[3])
	{
		is_an_error("Kaiser_FIR: Nyquist frequency in specs exceeded: ", -4);
		return(-4);
	}

	/* scalin for discrete-time sampling frequency */
	scale = M_PI / Nyq;

	if ( (HiPass) || (BandPass) )
	{
		swap(edge[1], edge[0])
	}

	/* center frequency of transition band */
	om_c = (( edge[1] + edge[0]) / 2.0) * scale;
```

```

	/* width of the transition band */
	del_om = (edge[0] - edge[1]) * scale;

	if (BandPass)
	{
		om_c_2   = ( (edge[2] + edge[3]) / 2.0 ) * scale;
		del_om_2 = (edge[3] - edge[2]) * scale;
		del_om   = min(del_om, del_om_2);
		del      /= 2.0;
	}

	A = -20.0 * log10(del);

	if (50.0 < A)
		beta = 0.1102 * (A - 8.7);

	else if (A < 21.0)
		beta = 0.0;

	else {
		beta = A - 21.0;
		beta = 0.5842 * pow(beta,0.4) + 0.07886 * beta;
	}

	i_beta = i0(beta);

	/* the prediction of the filter length to keep the specs is
      of accuracy +-2. Make sure that the filter specifications are
	 satisfied by adding 2 */

	*order = M = ceil((A-8.0) / (2.285 * del_om)) + 2;

	/* hipass filters have to be FIR type I (M must be even) */
	if ( (HiPass) && (M&1) )
	{
		M++;
		(*order)++;
	}

	alpha = 0.5 * (double)(M);

	for (i=0; i <= M; i++)
	{
		if ( (i<<1) !=M )
		{
			temp = (double)(i) - alpha;
			h[i] = (sin (om_c * temp) / (M_PI * temp));

			if (HiPass)
				h[i] = (sin (M_PI * temp) / (M_PI*temp)) - h[i];

			if (BandPass)
				h[i] = (sin (om_c_2 * temp) / (M_PI*temp)) - h[i];

			temp /= alpha;
			temp *= temp;
			temp  = beta * sqrt(1.0 - temp);

			h[i] * = i0(temp) / i_beta;

		} else {

			h[i] = om_c / M_PI;

			if (HiPass)
				h[i] = 1.0 - h[i];

			if (BandPass)
				h[i] = om_c_2 / M_PI - h[i];

		}

	}
```

```
        return(0);
}

```

Appendix 5.6: `Variance` - Program Computing First Order Statistics of Signal

```
/*****************************************************************************
*
*                       Variance.c
*
******************************************************************************
*
*
*   DESCRIPTION
*
*       Computes the Mean, the Rms, and the Variance of data with the
*       Vector Accelerator
*
*/

#include <stdio.h>
#include <math.h>
#include <aplib.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include "/usr/erk/DSP/FileOp.h"

typedef int vector;
typedef int bool;

#define reg1 register
#define reg2 register
#define reg3 register
#define reg4 register
#define reg5 register
#define reg6 register

void perror();
void exit();
char *malloc();
long lseek();

#ifdef DEBUG
#define DUMP(Y,length)    mapsyncdma(-1,VA0);                            \
                          mapstrfv(Y,1,pafFilterData,4,length);      \
                          mapbwaitdma(VA0);                              \
                          for (i=0; i<= length-1; i++)                   \
                                  printf("%f \n", pafFilterData[i]);     \
                          exit(0);


#define MAGC(Y,y_len)     mapsyncmath(-1,VA0);                       \
                          mapnrmsqcfv(Y,2,Y,1,y_len);                \
                          DUMP(Y,y_len)                              \
                          exit(0);
#endif

#define knRealTime        -20

/*
-----------------------------------------------------------------------------
*/

void Usage()
{
```

```
	fprintf(stderr,"\n");
	fprintf(stderr,"This program computes the mean, the rms, and the variance of a given data file \r
	fprintf(stderr,"\n");
	fprintf(stderr,"USAGE:\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-C\tDo only short to float conversion\n");
	fprintf(stderr,"\t[with -S option only]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-i\tInput file containing the data \n");
	fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Filtered.dat]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-v\tFile containing the standard deviation of the filtered data\n");
	fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Threshold.dat]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-S\tInput data are short\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-o\tOutput file for data converted to float\n");
	fprintf(stderr,"\t [only with the -S option]\n");
	fprintf(stderr,"\t [DEFAULT: /usr/erk/DSP/DAT/Filtered.dat]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-h\tPrint this message\n");
	fprintf(stderr,"\n");

	exit(0);
}

/*
----------------------------------------------------------------------------
*/

main(argc,argv)
int argc;
char **argv;
{
	vector	vMean;
	vector	vMeanSq;
	vector	vData;

	bool	tFloat;

	int	chOption;

	int	nItemsRead = 0;
	int	nItemsInFile;
	int	*pnDummy;

	int	nChunkSize = 4096;

	int	fdInput;
	int	fdOutput;

	FILE	*fpVariance;

	bool	tInputIsShort = 0;
	bool	tComputeStatistics = 1;

	reg1 int nLoops;
	reg2 int nRemLen;
	reg3 int i;

	static char *pachOutputFile	= "/usr/erk/DSP/DAT/Filtered.dat";
	static char *pachInputFile	= "/usr/erk/DSP/DAT/Filtered.dat";
	static char *pachVarFile = "/usr/erk/DSP/DAT/Threshold.dat";

	static float *pafData;
	static short *pasData;

	static float fMean;
	static float fMeanSq;
	static float fStdDev;

	extern char	*optarg;
	extern int	optind;

```

```
	while ((chOption = getopt(argc, argv, "hi:o:v:SC")) != EOF)
	{
		switch (chOption)
		{
			case 'h':
				Usage();
				break;

			case 'i':
				pachInputFile = optarg;
				break;

			case 'o':
				pachOutputFile = optarg;
				break;

			case 'v':
				pachVarFile = optarg;
				break;

			case 'S':
				tInputIsShort = 1;
				break;

			case 'C':
				tComputeStatistics = 0;
				nChunkSize = 8192;
				break;

			case '?':
				Usage();
				break;
		}
	}

/*----------------------------------------------------------------------*/

	/*
	 *	Get Real-time priority
	 */

	if ( (int) (nice(knRealTime)) != knRealTime )
	{
		fprintf(stderr, "Got different priority than requested, errno: %d", errno);
		perror();
		exit(-1);
	}

/*----------------------------------------------------------------------*/

	/*
	 *	Open the input and the output data files
	 */

	FdOpenR(pachInputFile, fdInput)

	/*
	 *	If the input data are short they are converted to float
	 *	and written to the file in pachOutputFile
	 */
	if (tInputIsShort)
	{
		FdOpenW(pachOutputFile, fdOutput)
	}


	FpOpenW(pachVarFile, fpVariance)

	/*
	 *	the first entry in the input file is the number of samples
	 *	contained therein
	 */

/***	 fscanf(fpInput,"%d:\n", &nItemsInFile);
```

```
 ***/

        pnDummy = &nItemsInFile;

        FdRead(fdInput,pnDummy,sizeof(int))

        fprintf(stderr, "\n Input file %s contains %d samples\n",pachInputFile, nItemsInFile);

        if (tInputIsShort)
        {
                FdWrite(fdOutput,pnDummy,sizeof(int));
                fprintf(stderr, "\n Output file %s contains %d samples\n",pachOutputFile, nItemsInFile);
         }

/*-------------------------------------------------------------------*/
        /*
         *      Allocate memory for the data
         */

        pafData = (float *)malloc( sizeof(float) * nChunkSize );

        if (tInputIsShort)
                pasData = (short *)malloc( sizeof(float) * nChunkSize );

/*-------------------------------------------------------------------*/
        /*
         *      Initializing the VA
         */

        mapinitva(1,1,0);

        vData    = 0;
        vMean    = vData  + nChunkSize;
        vMeanSq  = vMean + nChunkSize;

/*-----------------------------------------------------------------------*/

        nLoops = (int) ( (float)nItemsInFile/(float)nChunkSize );

        fprintf(stderr, "\n%d loops necessary\n", nLoops);

        mapclrfv(vMean,1,nChunkSize);
        mapclrfv(vMeanSq,1,nChunkSize);

        while (nLoops--)
        {

/***            for (i=0; i<=nChunkSize-1; i++)
 ***            fscanf(fpInput, "%f\n", (pafData+i));
 ***/
                if (tInputIsShort)
                {
                        FdRead(fdInput,pasData, nChunkSize * sizeof(short))

/***                    for (i=nChunkSize; 0<i; i--)
 ***                            printf("%d\n",*(pasData+i));
 ***/
                        nItemsRead += nChunkSize;

                        mapsyncdma(-1,VA0);

                        maplodwv(pasData,2,vData,1,nChunkSize);

                        mapsyncmath(-1,VA0);

                        mapcvtifv(vData,1,vData,1,nChunkSize);

                        mapsyncdma(-1,VA0);

                        mapstrfv(vData,1,pafData,4,nChunkSize);

                        mapbwaitdma(-1,VA0);

                        FdWrite(fdOutput, pafData, nChunkSize *sizeof(float))
```

```
                }
                else
                {
                        FdRead(fdInput, pafData, nChunkSize * sizeof(float))
                        nItemsRead += nChunkSize;
                        /*
                         *      Shows whether data are read correctly
                         *      comment out if necessary
                         *
                         ***     for (i=0; i<=(nChunkSize-1); i++)
                         ***    printf("%d \n", *(pafData+i) );
                         ***    exit(0);
                         ***/
                        mapsyncdma(-1,VA0);
                        maplodfv(pafData, 4, vData, 1, nChunkSize);
                        mapsyncmath(-1,VA0);
                }
                if (tComputeStatistics)
                {
                        /*
                         *      Add new data to mean
                         */
                        mapaddfvv(vData,1,vMean,1,vMean,1,nChunkSize);
                        /*
                         *      Add square of new data to mean square
                         */
                        mapmafvvv(vData,1,vData,1,vMeanSq,1,vMeanSq,1, nChunkSize);
                }
        }
        /*
         *      Take care of eventually remaining data
         */
        nRemLen = nItemsInFile - nItemsRead;
        if (nRemLen)
        {
/***            for(i=0; i<= nRemLen-1; i++)
 ***                    fscanf(fpInput,"%f\n",(pafData+i));
 ***/
                if (tInputIsShort)
                {
                        FdRead(fdInput,pasData, (nRemLen * sizeof(short)) )
                        nItemsRead += nRemLen;
                        mapsyncdma(-1,VA0);
                        maplodwv(pasData,2,vData,1,nRemLen);
                        mapsyncmath(-1,VA0);
                        mapcvtifv(vData,1,vData,1,nRemLen);
                        mapsyncdma(-1,VA0);
                        mapstrfv(vData,1,pafData,4,nRemLen);
                        mapbwaitdma(-1,VA0);
                        FdWrite(fdOutput, pafData, nRemLen *sizeof(float))
                }
```

```
			else
			{

				FdRead(fdInput, pafData, (nRemLen * sizeof(float)) )


				nItemsRead += nRemLen;

				/*
				 *	Shows whether data are read correctly
				 *	comment out if necessary
				 *
				 ***	 for (i=0; i<=(nRemLen-1); i++)
				 ***	printf("%d \n", *(pafData+i) );
				 ***	exit(0);
				 ***/

				mapsyncdma(-1,VA0);

				maplodfv(pafData, 4, vData, 1, nRemLen);

				mapsyncmath(-1,VA0);
			}
			if (tComputeStatistics)
			{
			mapaddfvv(vData,1,vMean,1,vMean,1,nRemLen);

			mapmafvvv(vData,1,vData,1,vMeanSq,1,vMeanSq,1, nRemLen);
			}
		}

		fprintf(stderr,"\n %d items read\n",nItemsRead);

		if (tComputeStatistics)
		{
		mapsumfv(vMean, 1, vMean, nChunkSize);

		mapsumfv(vMeanSq,1,vMeanSq, nChunkSize);

		mapsyncdma(-1,VA0);

		mapstrfv(vMean,1,&fMean, 4, 1);

		mapstrfv(vMeanSq,1,&fMeanSq,4,1);

		mapbwaitdma(VA0);

		fMean   /= (float)nItemsRead;
		fMeanSq /= (float)nItemsRead;

		fprintf(stderr,"\nMean of input file:\t%f\n", fMean);

		fprintf(stderr,"\nRms of input file:\t%f\n",sqrt(fMeanSq));

		fStdDev = fMeanSq - fMean *fMean;

		if (fStdDev < 0.0 )
			fStdDev = (-1.0) * fStdDev;

		fStdDev = sqrt(fStdDev);

		fprintf(stderr,"\nStandard Dev of input file:\t%f\n",fStdDev);

		fprintf(fpVariance,"%f:\n", fStdDev);


		close (fdInput);

		if (tInputIsShort)
			close(fdOutput);

/***		fclose(fpInput); ***/
		fclose(fpVariance);
	}
```

```
        exit(0);
}

```

Appendix 5.7: `GetBursts` - Program for Burst Validation

```
/******************************************************************
 *
 *                          GetBursts.c
 *
 ******************************************************************
 *
 * DESCRIPTION
 *   Isolates the bursts from the data. As they data are expected to be
 *   prefiltered in the discrete-time domain, the input array is supposed
 *   to be float.
 *   The bursts are written to file in the following way:
 *       First comes the length of the validated burst in samples
 *        then the burst follows as an array of float.
 *
 * USAGE
 *   Lots of options check function Usage() below for details
 *   or run the program with the -h option
 *
 * COMMENTS
 *   out of some reasons the implemented system acts close to the DISA
 *   LDA counter processor, blame it on insufficient creativity.
 *   it seems only straightforward to me to include an option which
 *   performs zero crossing counting on the bursts, presumably this
 *   would be pretty fast, too.
 *   The way the DISA validation scheme works it looks like there is
 *   a weak spot for a rapidly decreasing signal: If one maximum at the
 *   end of a burst is well over Trigger._2 but he next maximum is below
 *   Trigger_1 then the burst is not considered as terminated.
 *
 *   The trigger levels can either be specified as a multiple of a threshold or
 *   - together with the -D option - as direct values
 */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include "/usr/erk/DSP/FileOp.h"

#define reg1 register
#define reg2 register
#define reg3 register
#define reg4 register
#define reg5 register
#define reg6 register

void exit();
void perror();
double atof();
char *malloc();
long ftell();

typedef int bool;

#define knLow           5
#define knHigh          8

#define knRealTime      -20

#define READ            04
#define WRITE           02
#define EXISTS          00
```

```
#define Reset    nThresholdCrossings        = 0;     \
                 tWithinBurst               = 0;     \
                 tFirstCrossing             = 1;     \
                 nDuration                  = 0;


/*
-----------------------------------------------------------------------------
*/

void Usage()
{
        fprintf(stderr,"This program implements the DISA burst validation\n");
        fprintf(stderr,"scheme for laser Doppler anemometry \n");
        fprintf(stderr,"\n");
        fprintf(stderr,"\tUSAGE:\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-i <input file name>\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Filtered.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-o <output file name>\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Bursts.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-t <file containing the threshold>\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Variance.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-f <file containing the number of bursts\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/NOfBursts.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-D <maximum duration>\n");
        fprintf(stderr,"\t This option MUST be specified\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-b <Number of Bursts>\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-d <minimum duration>\n");
        fprintf(stderr,"\t This option MUST be specified\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-A Factor for obtaining Trigger_1 from threshold");
        fprintf(stderr,"\t [Default: 0.5]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-B Factor for obtaining Trigger_2 from threshold");
        fprintf(stderr,"\t [Default: 1.5]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-C Factor for obtaining Trigger_3 from Threshold");
        fprintf(stderr,"\t [Default: 3.0]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-M The values specified under -A, -B, and -C are not factors");
        fprintf(stderr,"\n");
        fprintf(stderr,"-N Start with a new burst count \n");
        fprintf(stderr,"\t (Delete file specified under the -f option\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-Q Comparator Accuracy (see DISA counter manual)\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-h Print this message\n");
        fprintf(stderr,"\n");

        exit(-1);
}

/*
=============================================================================
*/

main(argc,argv)
int argc;
char **argv;
{

        bool    tWithinBurst = 0;
        bool    tFirstCrossing = 1;

        float   fMax=0.0;

```

```
	float	fTrigger_1;
	float	fTrigger_2 = -1;
	float	fTrigger_3 = 1800.0;

	float	fFactorForTrig1 = 0.5;
	float	fFactorForTrig2 = 1.0;
	float	fFactorForTrig3 = 3.0;

	float	fCompAcc;

	bool	tDelFile = 0;
	bool	tFactor = 1;

	int	nMinDur = 5;
	int	nMaxDur = 512;

	int	nBurstsSpecified;

	int	nTimeLo, nTimeHi;
	int	nThresholdCrossings = 0;
	int	nItemsInFile;
	int	i;
	int	chOption;

	int	*pnDummy;

	reg1	float *pfPrevSample;
	reg2	float *pfSample;
	reg3	int nDuration = 0;

	static int nBursts = 0;

	static char *pachBurstFile	= "/usr/erk/DSP/DAT/NOfBursts.dat";
	static char *pachOutputFile	= "/usr/erk/DSP/DAT/Bursts.dat";
	static char *pachInputFile	= "/usr/erk/DSP/DAT/Filtered.dat";
	static char *pachThreshFile	= "/usr/erk/DSP/DAT/Threshold.dat";

	int	fdInput;

	FILE	*fpOutput;
	FILE	*fpThresh;
	FILE	*fpBurst;

	extern char	*optarg;
	extern int	optind;

/*-------------------------------------------------------------------*/
	/*
	 *	Parse the command line
	 */

	while ((chOption = getopt(argc, argv, "hb:d:f:i:o:t:D:MNA:B:C:Q:")) != EOF)
	{
		switch (chOption)
		{
			case 'h':
				Usage();
				break;

			case 'i':
				pachInputFile = optarg;
				break;

			case 'o':
				pachOutputFile = optarg;
				break;

			case 'f':
				pachBurstFile = optarg;
				break;

			case 'b':
				nBurstsSpecified = atoi(optarg);
				break;
```

```

				case 't':
					pachThreshFile = optarg;
					break;

				case 'D':
					nMaxDur = atoi(optarg);
					break;

				case 'd':
					nMinDur = atoi(optarg);
					break;

				case 'A':
					fFactorForTrig1 = (float)atof(optarg);
					break;

				case 'B':
					fFactorForTrig2 = (float)atof(optarg);
					break;

				case 'C':
					fFactorForTrig3 = (float)atof(optarg);
					break;

				case 'M':
					tFactor = 0;
					break;

				case 'N':
					tDelFile = 1;
					break;

				case 'Q':
					fCompAcc = (float)atof(optarg);
					break;

				case '?':
					Usage();
					break;
			}
		}

/***	if (nMaxDur < 0)
 ***	{
 ***		fprintf(stderr,"No maximum duration specified, but required\n");
 ***		Usage();
 ***		exit(-1);
 ***	}
 ***
 ***	if (nMinDur < 0)
 ***	{
 ***		fprintf(stderr,"No minimum duration specified, but required\n");
 ***		Usage();
 ***		exit(-1);
 ***	}
 ***/

/*-------------------------------------------------------------------*/

	/*
	 *	Get Real-time priority
	 */
	if ( (int) (nice(knRealTime)) != knRealTime )
	{
		fprintf(stderr, "Got different priority than requested, errno: %d", errno);
		perror();
		exit(-1);
	}

/*-------------------------------------------------------------------*/
	FdOpenR(pachInputFile,fdInput)

	FpOpenW(pachOutputFile,fpOutput)
```

```
        FpOpenR(pachThreshFile, fpThresh)

/*-----------------------------------------------------------------*/

        if(tDelFile)
        {
                if (unlink(pachBurstFile) == -1)
                {
                        if (errno != ENOENT)
                        {
                                fprintf(stderr,"\nCannot unlink/delete %s, errno: %d\n",pachBurstFile,errr
                                perror(pachBurstFile);
                                exit(-1);
                        }
                        else
                        {
                                errno = 0;
                        }
                }
        }

/*-----------------------------------------------------------------*/
        /*
         *      Check whether we processed already a data record
         */
        if (access(pachBurstFile, READ | WRITE | EXISTS) < 0)
        {
                /*
                 *      Input file does not exist yet, create it
                 */

                FpOpenW(pachBurstFile, fpBurst)

                nBursts = 0;

                /*
                 *      Clear errno, which is 2 (no such file) if we're here
                 */
                errno=0;
        }
        else
        {

                /*
                 *      File exists, open for update
                 */

                FpOpenRWU(pachBurstFile, fpBurst)

                fscanf(fpBurst, "%d:\n", &nBursts);
                rewind(fpBurst);

        }

/*-----------------------------------------------------------------*/

        /*
         *      The first number in the input file
         *      gives the amount of data contained
         */
        pnDummy = &nItemsInFile;
        FdRead(fdInput,pnDummy,sizeof(int))

        fprintf(stderr, "\nInput file %s contains %d items\n", pachInputFile, nItemsInFile);

        if ( (pfSample = (float *) malloc (nItemsInFile*sizeof(float))) == NULL )
        {
                fprintf(stderr,"Cannot allocate memory, errno: %d\n",errno);
                perror();
                exit(-1);
        }

        if (plockin(pfSample,nItemsInFile*sizeof(float)) < 0)
```

```
	{
		fprintf(stderr,"Cannot lock memory, errno: %d\n",errno);
		perror();
		exit(-1);
	}

	fprintf(stderr,"\n %d bytes locked into memory\n",sizeof(float)*nItemsInFile);

	FdRead(fdInput,pfSample,(sizeof(float)*nItemsInFile))

	fprintf(stderr,"\n %d bytes read from %s\n",sizeof(float)*nItemsInFile, pachInputFile);

	nItemsInFile--;

/*-------------------------------------------------------------------*/

	/*
	 *	Read the trigger value from file
	 *	The trigger value may come from the routine Variance
	 *	which computes the rms value
	 *	The rms value is agood approximation for the standard
	 *	deviation if the data are low pass filtered
	 */

	if (tFactor)
	{
		fscanf(fpThresh,"%f:\n", &fTrigger_2);
		fTrigger_1  = fFactorForTrig1 * fTrigger_2;
		fTrigger_3  = fFactorForTrig3 * fTrigger_2;
		fTrigger_2  = fFactorForTrig2 * fTrigger_2;
	}
	else
	{
		fTrigger_1  = fFactorForTrig1;
		fTrigger_3  = fFactorForTrig3;
		fTrigger_2  = fFactorForTrig2;
	}

/*-------------------------------------------------------------------*/

_Next_Sample_:

	while(nItemsInFile--)
	{


		pfPrevSample = pfSample++;

		/*
		 *	Check whether we have already found a burst
		 *	if yes then look for reset conditions or its end
		 */

		if (tWithinBurst)
		{

			/*
			 *	Duration of the burst
			 */
			nDuration++;

			/*
			 *	Overshoot resets
			 */

			if (fTrigger_3 < *pfSample)
			{
				Reset

				goto _Next_Sample_;
			}

			/*
			 *	Look out for a maximum
```

```
                 *      If a maximum falls between fTrigger_1 and fTrigger_2
                 *      then our burst is terminated
                 */

                if ( *pfPrevSample < *pfSample)
                {
                        fMax = *pfSample;
                }
                else
                {
                        if ((fMax < fTrigger_2) && (fTrigger_1 < fMax))
                        {

                                /*
                                 *      Get the duration of the burst
                                 *      If it has the right length, read it
                                 *      from the input file and copy it to
                                 *      the output file
                                 */

                                if ( (nMinDur < nDuration) && (nDuration < nMaxDur) )
                                {


                                        /*
                                         */

                                        fprintf(fpOutput,"%d:\n",nDuration);
                                        for (i = --nDuration; i>=0; i--)
                                        {
/***
 ***                                            printf("%f\n",*(pfSample-i));
 ***/
                                                fprintf(fpOutput,"%f\n",*(pfSample+i));
                                        }

                                        /*
                                         * Update the count of validated bursts
                                         */

                                        nBursts++;

                                        /*
                                         *      Print a period on standard error for each burst :
                                         */

                                        fprintf(stderr,".");

                                        /*
                                         *      The specified number of bursts occured,
                                         *      exit
                                         */

                                        if (nBursts == nBurstsSpecified)
                                        {
                                                if (unlink(pachBurstFile) == -1)
                                                {
                                                        fprintf(stderr,"\nCannot unlink %s at ex
                                                        exit(-1);
                                                }
                                                goto _Exit_;
                                        }

                                        Reset

                                        goto _Next_Sample_;
                                }

                                else
                                {
                                        /*
                                         *      too short
                                         */

```

```
                Reset

                goto _Next_Sample_;

            }

        }

        /*
         *      We are going downhill again, reset the maximum
         */
        fMax = 0.0;
    }

    if ( (*pfPrevSample < fTrigger_2) && (fTrigger_2 < *pfSample ) )
    {

        nThresholdCrossings ++;

        /*
         *      a positive crossing of Trigger_2 occured
         *      get time between crossings
         *      obtain times for a number of knLow and knHigh threshold crossin
         */


        /*
         *      low count of threshold crossings, get event time
         */

        if (nThresholdCrossings == knLow)
            nTimeLo = nDuration;

        /*
         *      high count of threshold crossings, get event time
         */

        if (nThresholdCrossings == knHigh)
        {
            nTimeHi  = nDuration;

            /*
             *      that's the way DISA does it, aha-aha, I like it
             *      see manual for handwaving argument why this is good an
             */

            if ( (0.625 * (float) (nTimeHi) - (float)(nTimeLo)) > (0.625 *
            {
                Reset

                goto _Next_Sample_;
            }

        }

    }

}
else
{
    /*
     *      A positive crossing of fTrigger_2 sets
     *      tWithinBurst
     */

    if ( (*pfPrevSample < fTrigger_2) && ( fTrigger_2 < *pfSample ) )
    {
        tWithinBurst ++;

        nDuration = 1;

        nThresholdCrossings = 1;

    }
```

```
			}

		/*
		 *	Nothing happened: take next sample
		 */
		}

		/*
		 *	We reached the end of the file containing the sampled data
		 *	without getting the required number of bursts:
		 *	exit with the number of bursts processed so far
		 */

		fprintf(fpBurst,"%d:\n",nBursts);



		/*
		 *	The label _Exit_ is branched to if the number of
		 *	specified bursts has been reached
		 */
_Exit_:
		fclose(fpThresh);
		fclose(fpOutput);
		fclose(fpBurst);

		close(fdInput);

		fprintf(stderr,"\n");

		fprintf(stderr," Total of %d bursts\n",nBursts);

		if(!(nBursts))
		{
			if (unlink(pachOutputFile) == -1)
			{
				fprintf(stderr,"\nCannot unlink/delete %s, errno: %d\n",pachOutputFile,errno);
				perror(pachOutputFile);
				exit(-1);
			}
			fprintf(stderr,"\nDeleted burst file\n");
		}


		exit(nBursts);

}

```

Appendix 5.8: MeanSpec - Program Computing the Mean Spectrum

```
/******************************************************************************
 *
 *				MeanSpec.c
 *
 ******************************************************************************
 *
 *  DESCRIPTION
 *    This routine comutes the mean spectrum and the standard deviation
 *    from the mean on the vector accelerator.
 *    It allows for different methods for the spectral estimation. Presently,
 *    classical direct DFT (via FFT) computation and an ARMA (auto-regressive
 *    moving average) estimator based on Pade approximation to quotient
 *    of polynomials (see thesis) are implemented.
 *    The men is obtained by residence-time weighting: The length of the data
 *    record contributing to the mean is taken into account.
 *
```

```
 *  USAGE
 *     -h  print information about usage
 *     -i  input file containing the data records. See GetBursts.c for format
 *     -o  output file containing the mean and the mean square spectrum
 *     -r  file containing the result, i.e. the latest mean spectrum
 *     -v  file containing the latest standard deviation
 *     -D  expected maximum duration of the bursts, see GetBursts.c
 *     -F  length of the FFT
 *     -N  remove output file before computing starts
 *     -m  method to use for the spectral estimation
 *         1 ... direct computation via FFT
 *         2 ... ARMA Pade estimator
 *
 *  DEFAULTS
 *     Again all default values are organized to provide smooth operation
 *     with the rest off the files used in LDA signal processing. In this
 *     case, the input files stem from the routine GetBursts.c
 *
 *     Input file:         /usr/erk/DSP/DAT/Bursts.dat
 *     Output file:        /usr/erk/DSP/DAT/Working.dat
 *     Result in:          /usr/erk/DSP/DAT/Result.dat
 *     Standard dev in:    /usr/erk/DSP/DAT/SpecVar.dat
 *     Sampling frequency: 1.0e6
 *     Length of FFT:              512
 *     Method:                     1
 *
 *  (note that the file Variance.dat is used by FilterData.c)
 */
#include <math.h>
#include <stdio.h>
#include <signal.h>
#include <aplib.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/file.h>
#include <unistd.h>
#include "/usr/erk/DSP/FileOp.h"

void exit();
void perror();
double atof();
char *malloc();

#ifdef DEBUG

#define DUMP(Y,y_len)   mapsyncdma(-1,VA0);                    \
                         mapstrfv(Y,1,r,4,y_len);  \
                         mapbwaitdma();                        \
                         for(j=0; j<=y_len-1; j++) \
                                 printf("%e\n",r[j]);          \


#define MAGC(Y,y_len)    mapsyncmath(-1,VA0);                  \
                         mapnrmsqcfv(Y,2,Y,1,y_len);           \
                         DUMP(Y, y_len)

#endif

#define READ             04
#define WRITE            02
#define EXISTS           00

#define knRealTime       -20

#define MAX_FFT_LEN   1024

#define vR        vTemp1
#define vT        vTemp2

#define FOREVER   for(;;)

typedef int vector;
typedef int bool;

```

```
/*--------------------------------------------------------------*/

void Usage()
{
	fprintf(stderr,"\n");
	fprintf(stderr,"This program computes the mean spectrum from \n");
	fprintf(stderr,"the output as obtained by the routine GetBursts\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"USAGE:\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-i\tInput file containing the isolated burst\n");
	fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Bursts.dat]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-o\tOutput file containing the mean and mean square spectrum\n");
	fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Working.dat]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-r\tFile containing the latest mean spectrum\n");
	fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Result.dat\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-v\tFile containing the latest standard deviation\n");
	fprintf(stderr,"\t\t of the spectrum\n");
	fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/SpecVar.dat]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-F\tLength of the Fourier transform to use\n");
	fprintf(stderr,"\t [Default: 512]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-D\tMaximum duration of bursts\n");
	fprintf(stderr,"\tMUST BE SPECIFIED\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-T\tSampling frequency\n");
	fprintf(stderr,"\t [Default: 1.0e6]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-N\tRemove file specified under -o first\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-m\t1 ... spectral estimation using FFT\n");
	fprintf(stderr,"\t2 ... spectral estimation using ARMA Pade estimator\n");
	fprintf(stderr,"\t [Default: 1]\n");
	fprintf(stderr,"\n");
	fprintf(stderr,"-h\tPrint this message\n");
	fprintf(stderr,"\n");

	exit(-1);
}

/*
==========================================================================
*/

main(argc, argv)
int	argc;
char	**argv;
{
#ifdef DEBUG
	static float r[MAX_FFT_LEN];
	float m[MAX_FFT_LEN];
	float v[MAX_FFT_LEN];

	int j;
#endif
	vector	vTemp1, vTemp2, vTemp3, vTemp4, vTemp5;
	vector	vResult;
	vector	vVar;
	vector	vMeanSq;
	vector	vMean;
	vector	vCoeff;
	vector	vfScal1, vfScal2;
	vector	vResFFT;
	vector	vBurst;
	vector	vEndOfMem;

	bool	tFileExists = 0;
	bool	tDelFile = 0;

	int	nMethod = 1;
```

```
	int	nDegR;
	int	nDegT;

	int	nLenFFT = 512;
	int	nLogLen;
	int	nHalfLenFFT;
	int	nLenFFTFromFile;

	int	nSamplesInBurst;
	int	nMaxDur = -1;

	int	nProcessed = 0;
	int	i;

	int	chOption;

	static float nfDurationOfBurst;
	static float nfTotalTime = 0.0;

	static float *pafBurst;
	static float *pafMean;
	static float *pafMeanSq;
	static float *pafResult;
	static float *pafVar;


	static char *pachOutputFile	= "/usr/erk/DSP/DAT/Working.dat";
	static char *pachInputFile	= "/usr/erk/DSP/DAT/Bursts.dat";
	static char *pachVarFile = "/usr/erk/DSP/DAT/SpecVar.dat";
	static char *pachResFile = "/usr/erk/DSP/DAT/Result.dat";

	FILE	*fpInput;
	FILE	*fpOutput;
	FILE	*fpVariance;
	FILE	*fpResult;

	extern char	*optarg;
	extern int	optind;

/*--------------------------------------------------------------------------*/
	/*
	 *	Get real-time priority
	 */

	if ( (int)nice(knRealTime) != knRealTime )
	{
		fprintf(stderr,"\nNice: Got different priority than requested, errno: %d\n", errno);
		perror();
		exit(-1);
	}
/*--------------------------------------------------------------------------*/

	while ((chOption = getopt(argc, argv, "hi:o:D:F:Nr:v:m:")) != EOF)
	{
		switch (chOption)
		{
			case 'h':
				Usage();
				break;

			case 'i':
				pachInputFile = optarg;
				break;

			case 'D':
				nMaxDur = atoi(optarg);
				break;

			case 'v':
				pachVarFile = optarg;
				break;

			case 'o':
```

```
					pachOutputFile = optarg;
					break;

				case 'r':
					pachResFile = optarg;
					break;

				case 'F':
					nLenFFT = atoi(optarg);
					break;

				case 'm':
					nMethod = atoi(optarg);
					break;

				case 'N':
					tDelFile = 1;
					break;

				case '?':
					Usage();
					break;
			}
		}

/*----------------------------------------------------------------------------*/

		if(tDelFile)
		{
			if(unlink(pachOutputFile) == -1)
			{
				fprintf(stderr,"\nCannot unlink/delete %s, errno: %d\n",pachOutputFile,errno);
				perror(pachOutputFile);
				exit(-1);
			}
		}

		if ( (nMethod < 1) || (nMethod > 2) )
		{
			fprintf(stderr,"\nWrong argument for method to be used\n");
			Usage();
		}

		if (nMaxDur < 0)
		{
			fprintf(stderr,"\n No or negative maximum duration specified\n\n");
			Usage();
		}

/*-------------------------------------------------------------------------*/

		/*
		 *	If the data file is missing we gracefully exit here
		 */

		if (access(pachInputFile, EXISTS) < 0)
		{
			fprintf(stderr,"\nMeanSpec: Did not find an input file, presumably,");
			fprintf(stderr,"\n\tbecause GetBursts didn't find anything");
			exit(0);
		 }

		/*
		 *	If an output file already exists, read it
		 *	otherwise open it for write
		 */

		if (access(pachOutputFile, READ | WRITE | EXISTS) < 0)
		{
			/*
			 *	Input file does not exist yet, create it
			 */

			FpOpenW(pachOutputFile, fpOutput)
```

```


        nLogLen = mapilog2(nLenFFT);
        nHalfLenFFT     = (nLenFFT>>1);

        pafMean         = (float *)malloc((nHalfLenFFT+1)<<2);
        pafMeanSq       = (float *)malloc((nHalfLenFFT+1)<<2);
        pafBurst = (float *)malloc(nMaxDur<<2);
        pafVar          = (float *)malloc((nHalfLenFFT+1)<<2);
        pafResult       = (float *)malloc((nHalfLenFFT+1)<<2);



    }
    else
    {
        /*
         *      File exists, open for update
         */
        tFileExists = 1;

        FpOpenRWU(pachOutputFile, fpOutput)

        /*
         *      First entry in input file must be length of FFT
         *      if length is not the same as in command line
         *      option: override
         */

        fscanf(fpOutput, "%d:\n", &nLenFFTFromFile);

        if (nLenFFTFromFile != nLenFFT)
        {
            fprintf(stderr, "\nPrevious spectrum has different length\n");
            fprintf(stderr, "...overriding command line option\n");

            fprintf(stderr,"\nNew length of FFTs: %d\n",nLenFFTFromFile);

            nLenFFT = nLenFFTFromFile;
        }

        nLogLen = mapilog2(nLenFFT);
        nHalfLenFFT     = (nLenFFT>>1);

        /*
         *      Now that we know with which FFT length we
         *      have to deal we allocate the memory
         */

        pafMean         = (float *)malloc((nHalfLenFFT+1)<<2);
        pafMeanSq       = (float *)malloc((nHalfLenFFT+1)<<2);
        pafBurst = (float *)malloc(nMaxDur<<2);
        pafVar          = (float *)malloc((nHalfLenFFT+1)<<2);
        pafResult       = (float *)malloc((nHalfLenFFT+1)<<2);

        /*
         *      Read the old mean and mean square spectrum
         *      they are of length (nLenFFT + 2) each
         */

        for (i = 0; i <= nHalfLenFFT; i ++)
        {
            fscanf(fpOutput, "%f:%f\n", pafMean+i, pafMeanSq+i);

#ifdef DEBUG
                m[i] = *(pafMean+i);
                v[i] = *(pafMeanSq+i);
#endif
        }

        /*
         *      Last entry is number of averaged spectra
         *      and total observation time
         */

```

```
			fscanf(fpOutput,"%d:%f",&nProcessed,&nfTotalTime);

			/*
			 *	Rewind for subsequent write
			 */

			rewind (fpOutput);

		}

/*-----------------------------------------------------------------*/
		mapinitva(1,1,0);
/*-----------------------------------------------------------------*/

		/*
		 *	Open the file containing the data to analyze,
		 *	the file which will contain the standard deviation
		 *	and the file which will contain the result
		 */

		FpOpenR(pachInputFile, fpInput)

		FpOpenW(pachVarFile, fpVariance)

		FpOpenW(pachResFile, fpResult)

/*-----------------------------------------------------------------*/
		/*
		 *	Organization of the vector memory
		 */

		vBurst   = 0;
		vTemp1   = vBurst  + (nLenFFT<<1);
		vTemp2   = vTemp1  + (nLenFFT<<1);
		vMean    = vTemp2  + (nLenFFT<<1);
		vMeanSq  = vMean   + (nHalfLenFFT + 1);
		vVar     = vMeanSq + (nHalfLenFFT + 1);
		vResult  = vVar    + (nHalfLenFFT + 1);
		vfScal1  = vResult + (nHalfLenFFT + 1);
		vfScal2  = vfScal1 + 1;
		vCoeff   = vfScal2 + 1;
		vTemp3   = vCoeff  + (nLenFFT + 2);
		vTemp4   = vTemp3  + (nHalfLenFFT + 1);
		vTemp5   = vTemp4  + (nHalfLenFFT + 1);

		vEndOfMem	 = vTemp5  + (nHalfLenFFT + 1);

/*-----------------------------------------------------------------*/

		maprffttab(vCoeff, nLogLen);

		if (tFileExists)
		{
			mapsyncdma(-1,VA0);
			maplodfv(pafMean,4,vMean,1,nHalfLenFFT+1);
			maplodfv(pafMeanSq,4,vMeanSq,1,nHalfLenFFT+1);
		}
		else
		{
			mapclrfv(vMean,1,nHalfLenFFT+1);
			mapclrfv(vMeanSq,1,nHalfLenFFT+1);
		}


/*-----------------------------------------------------------------*/

		vResFFT = (nLogLen&1) ? vTemp1 : vBurst;

		FOREVER
		{
			/*
			 *	Clear VA memory for the new burst
			 */
			mapclrfv(vBurst, 1, nLenFFT+2);
```

```
		mapsyncdma(-1, VA0);

		/*
		 *	get the length of the burst
		 */
		if (fscanf(fpInput, "%d:\n", &nSamplesInBurst) == EOF)
			break;

#ifdef CONTROL
		fprintf(stderr,"%d\t%d\n",nProcessed,nSamplesInBurst);
#endif

		/*
		 *	Compute duration of the burst and
		 *	load it into array processor memory
		 */
		nfDurationOfBurst = (float)nSamplesInBurst;

		maplodfs(&nfDurationOfBurst,vfScall);

		/*
		 *	Load burst from file to VA memory
		 */
		for (i = 0; i <= nSamplesInBurst - 1; i++)
			fscanf(fpInput, "%f\n", pafBurst+i);

		maplodfv(pafBurst, 4, vBurst, 1, nSamplesInBurst);
		mapsyncmath(-1,VA0);

		if (nMethod == 1)
		{

			/*
			 *	Do FFT
			 */
			maprfftnc(vBurst,1,vCoeff,2,vTemp1,1,nLenFFT);

		}
		else
		{
			/*
			 *	Do Arma Pade estimation
			 *	(note that vR and vT are #define'd
			 */

			PadeApprox
			(vBurst,nSamplesInBurst,vTemp3,vR,&nDegR,vT,&nDegT,vEndOfMem);

			/*
			 *	Remainder polynomial now in vR
			 *	Comultiplier polynomial now in vT
			 *	Clean them before FFT
			 */

			mapclrfv(vR+nDegR+1,1,nLenFFT+2-nDegR);
			mapclrfv(vT+nDegT+1,1,nLenFFT+2-nDegT);

			/*
			 *	vBurst is used as a temporary vector here
			 *	In ArmaPsd vR, vT, and vBurst MUST be
			 *	aligned on nLenFFT boundaries
			 */
			vResFFT = ArmaPSD(vR,vT,vCoeff,vBurst,nLenFFT,nLogLen);

		}


#ifdef DEBUG

			/*
			 *	As a check, we perform all the computations in
			 *	parallel the usual way
			 */

```

```
                    mapsyncdma(-1,VA0);
                    mapstrfv(vResFFT,1,r,4,nLenFFT+2);
                    mapbwaitdma(VA0);

                    for (j=0; j<= nHalfLenFFT; j++)
                    {
                            /*
                             *      After a 30 min. discussion we came to the
                             *      conclusion that this is the fastest way to implement i=2j+1 !!!!
                             */

                            i = (j<<1) | 0x1;

                            r[j] = 0.5 * sqrt(r[i] * r[i] + r[--i] * r[i]);

                            m[j] += nfDurationOfBurst * r[j];

                            v[j] += nfDurationOfBurst * r[j] * r[j];
                    }

#endif



            /*
             *      scale FFT, get PSD squared, square root
             */

            mapnrmsqcfv(vResFFT,2,vResFFT,1,nHalfLenFFT+1);
            mapsqrtfv(vResFFT,1,vTemp3,1,vTemp4,1,vTemp5,1,nHalfLenFFT+1);
            mapmulfsv(AP_OneHalf,vTemp5,1,vTemp5,1,nHalfLenFFT+1);

            /*
             *      Mean spectrum
             *
             *      M = M + del_t * PSD
             */
            mapmafsvv(vfScal1,vTemp5,1,vMean,1,vMean,1,nHalfLenFFT+1);

            /*
             *      Mean Square Spectrum (still in vResFFT)
             *
             *      MSQ <= MSQ + (del_t * PSD^2)
             */
            mapmulfvv(vTemp5,1,vTemp5,1,vResFFT,1,nHalfLenFFT+1);
            mapmafsvv(vfScal1,vResFFT,1,vMeanSq,1,vMeanSq,1,nHalfLenFFT+1);

            nfTotalTime += nfDurationOfBurst;
            nProcessed ++;

    }

    /*
     *      Load inverse of total observation time for normalizing
     */
    nfTotalTime = 1.0 / nfTotalTime;

    maplodfs(&nfTotalTime, vfScal2);

    mapsyncdma(-1, VA0);

    mapstrfv(vMean,1, pafMean,4,nHalfLenFFT+1);
    mapstrfv(vMeanSq,1,pafMeanSq,4,nHalfLenFFT+1);

    mapsyncmath(-1,VA0);
    /*
     *      Normalize mean and mean square with total observation time
     *
     *      Save present mean as result
     */
    mapmulfsv(vfScal2,vMean,1,vMean,1,nHalfLenFFT+1);
    mapmulfsv(vfScal2,vMeanSq,1,vMeanSq,1,nHalfLenFFT+1);

    mapsyncdma(-1,VA0);
```

```
        mapstrfv(vMean,1,pafResult,4,nHalfLenFFT+1);
        mapsyncmath(-1,VA0);

        /*
         *      Square of the present mean spectrum
         */
        mapmulfvv(vMean,1,vMean,1,vMean,1,nHalfLenFFT+1);

        /*
         *      Var[x] = E[x^2] - E[x]^2
         */
        mapsubfvv(vMeanSq,1,vMean,1,vVar,1,nHalfLenFFT+1);

        /*
         *      Square root of variance yields standard deviation
         */
        mapsqrtfv(vVar,1,vTemp1,1,vTemp2,1,vTemp3,1,nHalfLenFFT+1);
        mapsyncdma(-1,VA0);
        mapstrfv(vTemp3,1,pafVar,4,nHalfLenFFT+1);


        mapbwaitdma(VA0);

#ifdef DEBUG

                for ( i=0; i<=nHalfLenFFT; i++ )
                {
                        m[i] *= nfTotalTime;

                        v[i] *= nfTotalTime;

                        v[i] -= m[i] * m[i];

                        v[i] = sqrt( v[i] );
                }
#endif

        /*
         *      Write everything in the corresponding files
         *
         *      First the headers then the data
         */

        fprintf(fpOutput, "%d:\n",nLenFFT);
        fprintf(fpVariance,"%d:\n",(nHalfLenFFT+1) );
        fprintf(fpResult,"%d:\n",(nHalfLenFFT+1) );

        for (i=0; i<=nHalfLenFFT; i++)
        {
                fprintf(fpOutput,"%f:%f\n",*(pafMean+i), *(pafMeanSq+i));
                fprintf(fpVariance, "%f\n", *(pafVar+i));
                fprintf(fpResult, "%f\n", *(pafResult+i));

#ifdef WARNING
                if ((m[i]-v[i])<0.0)
                {
                        fprintf(stderr, "Pos: %d\tFPP-Min: %f\t",i,m[i]-v[i]);
                        v[i] -=  *(pafVar+i);
                        m[i] -=  *(pafResult+i);

                        fprintf(stderr,"(FPP-VA) Mean: %f\tStdDev: %f\n",m[i],v[i]);
                }
#endif
        }

        fprintf(fpOutput,"%d:%f\n", nProcessed, (1.0 / nfTotalTime) );

        fclose(fpInput);
        fclose(fpOutput);
        fclose(fpVariance);
        fclose(fpResult);

        return(0);
}
```

Appendix 5.8.1: `PadeApprox` - Initializes polynomials for Euclidean Algorithm

```
/*****************************************************************
*
*                          PadeApprox.c
*
******************************************************************
*
*  DESCRIPTION
*    Prepares an incoming data recording of length nDegA1 in vA1
*    for auto-regressive moving-average (ARMA) spectral estimation
*    using Pade approximation.
*
*  SYNOPSIS
*        int PadeApprox
*        (vA1, nDegA1, vA0, vR, pnDegR, vT, pnDegT, vEndOfMemInMain)
*        int nDegA1;
*        int *pnDegR, *pnDegT;
*        vector vA0, vA1, vR, vT, vEndOfMemInMain;
*
*  PARAMETERS
*        vA1              offset to numerator polynomial (loaded with data)
*        nDegA0           length of the filtered input data
*        vA0              offset to denominator polynomial
*        vR               offset to returned remainder polynomial
*        *pnDegT          returned degree of remainder polynomial
*        vT               offset to returned co-multiplier polynomial
*        *pnDegT          returned degree of co-multiplier polynomial
*        vEndOfMem        end of VA memory occupied by calling routine
*
*/

#include <stdio.h>
#include <aplib.h>

typedef int vector;

/* order of the co-multiplier polynomial */
#define AR_Order    2

#define swap(a,b)     (temp)=(b); (b)=(a); (a)=(temp);


#ifdef DEBUG

static float r[400];
int i;

#define DUMP(vY, nIncY, nLenY)           mapsyncdma(-1,VA0);           \
                                         mapstrfv(vY,nIncY,r,4,nLenY); \
                                         mapbwaitdma();                \
                                         for (i=0; i<=nLenY-1;i++) \
                                                 printf("%f\n",r[i]);  \
                                         exit(0);


#endif

int PadeApprox
(vA1, nDegA1, vA0, vR, pnDegR, vT, pnDegT, vEndOfMemInMain)
int nDegA1;
int *pnDegR, *pnDegT;
vector vA0, vA1, vR, vT, vEndOfMemInMain;
{
        int     nDegA0;

        vector  vTemp1   = vEndOfMemInMain;
        vector  vTemp2   = vTemp1 + nDegA1;
        vector  vfScal1  = vTemp2 + nDegA1;
        vector  vfScal2  = vfScal1+1;
        vector  vEndOfMem        = vfScal2+1;

        nDegA1--;     /* from length to degree of polynomial */

```

```
/*-------------------------------------------------------------------*/

#ifdef DEBUG
        fprintf(stderr,"\nEntering CheckOrderVA from PadeApprox...\n");
#endif

        CheckOrderVA(vA1, &nDegA1, vTempl, vfScal1, vfScal2);

        /*
         *      Degree of numerator polynomial
         */
        nDegA0 = nDegA1 + 1;

        mapclrfv(vA0, 1, nDegA0);

        /*
         *      Initialize numerator polynomial
         */
        mapcopfs(AP_1,vA0+nDegA0,1,1);

        mapbwaitmath();

/*---------------------------------------------------------------------
   start of the Euclidean Algorithm
---------------------------------------------------------------------*/
#ifdef DEBUG
        fprintf(stderr,"\n Entering EucAlgVA...\n");
#endif

        EucAlgVA
            (vA0, &nDegA0, vA1, &nDegA1, vR, pnDegR, vT, pnDegT, AR_Order, vEndOfMem);

#ifdef DEBUG
        fprintf(stderr,"\nIn PadeApprox: Degree of remainder polynomial:\t%d\n", *pnDegR);
        fprintf(stderr,"\nIn PadeApprox: Degree of comultiplier polynomial:\t%d\n", *pnDegT);
#endif

        return(0);
}
```

Appendix 5.8.2: `EucAlgVA` - Vectorized Euclidean Algorithm

```
/*********************************************************************
 *
 *                        EucAlgVA.c
 *
 *********************************************************************
 *
 *
 * DESCRIPTION
 *      Extended Euclidean algorithm using the array processor
 *      Terminates if the co-multiplier polynomial reaches the
 *      order prescribed in nOrderAR
 *
 * SYNOPSIS
 *     int EucAlg
 *     (vR2,pnDegR2,vR1,pnDegR1,vR,pnDegR,vT,pnDegT,nOrderAR,vEndOfMem)
 *     vector vR2, vR1, vR, vT;
 *     int *pnDegR2, *pnDegR1;
 *     int *pnDegR, *pnDegT;
 *     int nOrderAR;
 *     vector vEndOfMem;
 *
 * PARAMETERS
 *        vR2              AP offset of numerator polynomial
 *        pnDegR2          degree of numerator polynomial
 *        vR1              AP offset of denominator polynomial
 *        pnDegR1          degree of denominator polynomial
 *        vR               AP offset of returned remainder polynomial
 *        *pnDegR          pointer to degree of remainder polynomial
 *        vT               AP offset of co-multiplier polynomial
```

```
*       *pnDegT         pointer to degree of co-multiplier polynomial
*       AR_order approximate prescribed order of co-multiplier polynomial
*       vEndOfMem       End of vector memory used so far
*
*/

#include <aplib.h>
#include <stdio.h>


typedef int vector;

#define MAX_LEN         512

#define FOREVER   for (;;)


#ifdef DEBUG
static float r[1000];
#define DUMP(Y,y_length)  mapsyncdma(-1,VA0);                          \
                                mapstrfv(Y,1,r,4,y_length);             \
                                mapbwaitdma();                          \
                                for (i=0; i<=y_length-1; i++)           \
                                        printf( "%f\n",r[i]);           \
                                exit(1);
#endif

int ConvolveVA();
int PolyDivVA();
int CheckOrderVA ();
void is_an_error ();
void exit();

int EucAlgVA
(vR2, pnDegR2, vR1, pnDegR1, vR, pnDegR, vT, pnDegT, nOrderAR, vEndOfMem)
vector  vR2, vR1, vR, vT;
int     *pnDegR2, *pnDegR1;
int     *pnDegR, *pnDegT;
int     nOrderAR;
vector  vEndOfMem;
{

        vector  vQ       = vEndOfMem;
        vector  vT2      = vQ    + (*pnDegR2 + 1);
        vector  vT1      = vT2   + (*pnDegR2 + 1);
        vector  vTemp    = vT1   + (*pnDegR2 + 1);
        vector  vfScal1  = vTemp + (*pnDegR2 + 1);
        vector  vfScal2  = vfScal1 + 1;

        int     nDegT1 = 0;
        int     nDegQ;
        int     i;

        if (*pnDegR2 < *pnDegR1)
        {
                is_an_error
                ("\nEucAlgVA: Numerator polynomial smaller than denominator polynomial\n");
                exit(-1);
        }

        if (MAX_LEN < *pnDegR2)
        {
                is_an_error
                ("\nEucAlgVA: Numerator polynomial too large\n");
                exit(-1);
        }

        if (nOrderAR <= 0)
        {
                is_an_error
                ("\nEucAlgVA: Missing or negative order of AR branch\n");
                exit(-1);
        }

```

```
        /*
         *      Clear the whole vector memory needed by EucAlgVA
         *      and all its functions
         */
        mapclrfv(vQ,1,vfScal2-vQ+1);

        /*
         *      Co-multiplier polynomial: highest coefficient set to 1
         */

        mapcopfs(AP_1,vT1,1,1);

        FOREVER
        {
                /*
                 *      Divide the polynomials
                 */

#ifdef DEBUG
                fprintf(stderr,"\nEntering PolyDivVA...\n");
#endif

                PolyDivVA
                (vR2,*pnDegR2,vR1,*pnDegR1,vQ,&nDegQ,vR,pnDegR,vfScal1,vfScal2,vTemp);

#ifdef DEBUG
                fprintf(stderr,"\nEucAlg: Degree remainder polynomial:\t%d\n",*pnDegR);
#endif

                *pnDegT = nDegQ + nDegT1;

                /*
                 *      Compute the co-multiplier polynomial
                 */

#ifdef DEBUG
                fprintf(stderr,"\nEntering ConvlolveVA\n");
#endif
                ConvolveVA(vQ, nDegQ+1, vT1, nDegT1 + 1, vT, *pnDegT+1);


                mapsubfvv(vT2, 1, vT, 1, vT, 1, *pnDegT+1);


                /*
                 *      Eliminate eventual leading zeros in the coeffs
                 */

#ifdef DEBUG
                fprintf(stderr,"\nEntering CheckOrderVA from EucAlgVA...\n");
#endif

                CheckOrderVA(vT,pnDegT, vTemp, vfScal1, vfScal2);

#ifdef DEBUG
                fprintf(stderr,"\nEucAlg: Degree comultiplier polynomial:\t%d\n",*pnDegR);
#endif

                /*
                 *      The co-multiplier polynomial (responsible for the
                 *      AR branch) reached the specified order, back to
                 *      calling routine
                 */

                if (*pnDegT >= nOrderAR)
                {
                        mapbwaitrbe();
                        return(0);
                }

                /*
                 *      Update the polynomials for the next recursion:
                 *
                 *              R1 --> R2
```

```
                        *               R  --> R1
                        *               T1 --> T2
                        *               T  --> T1
                        */
                       mapcopfv(vR1,1,vR2,1,*pnDegR1+1);
                       *pnDegR2 = *pnDegR1;

                       mapcopfv(vR,1,vR1,1,*pnDegR+1);
                       *pnDegR1 = *pnDegR;

                       mapcopfv(vT1,1,vT2,1,nDegT1+1);

                       mapcopfv(vT,1,vT1,1,*pnDegT+1);
                       nDegT1 = *pnDegT;
               }
}
```

Appendix 5.8.3: `PolyDivVA` - Polynomial Division on the Vector Accelerator

```
/**********************************************************************
 *
 *                          PolyDivVA.c
 *
 **********************************************************************
 *
 * Divides two polynomials f(x) and g(x) with deg(f) > deg(g) and returns
 * the quotient and remainder polynomial using the array processor or the
 * vector accelerator
 *
 * SYNOPSIS
 *       int PolyDivVA
 *       (vA0,nDegA0,vA1,nDegA1,vQ,pnDegQ,vR,pnDegR,vfScal1,vfScal2,vTemp)
 *       vector  vA0, vA1, vQ, vR;
 *       vector  vfScal1, vfScal2, vTemp;
 *       int     nDegA0, nDegA1;
 *       int     *pnDegQ, *pnDegR;
 *
 *
 * INPUT
 *       vA0               offset for denominator polynomial
 *       nDegA0            degree of denominator polynomial
 *       vA1               offset for numerator polynomial
 *       nDegA1            degree of numerator polynomial
 *       vQ                offset of quotient polynomial
 *       *pnDegQ  degree of quotient polynomial
 *       vR                offset of remainder polynomial
 *       *pnDegR           degree of remainder polynomial
 *       vfScal1 etc       auxialiary vectors
 *
 * RETURN VALUES
 *       0    ... normal execution
 *      -1    ... deg_f < deg_g
 *      -2    ... deg_f > MAX_DEG
 *
 */

#include <aplib.h>

#define MAX_LEN  512

#ifdef DEBUG
#define DUMP(Y,y_length)  mapsyncdma(-1,VA0);                   \
                                  mapstrfv(Y,1,r,4,y_length);           \
                                  for (i=0; i<=y_length-1; i++)         \
                                          printf( "[%d] = %f \n",i,r[i]); \
                                  printf( "+++++++++ \n");
#endif

typedef int vector;

int CheckOrderVA();
```

```
void exit();
void is_an_error();

int PolyDivVA
(vA0,nDegA0,vA1,nDegA1,vQ,pnDegQ,vR,pnDegR,vfScal1,vfScal2,vTemp)
vector  vA0, vA1, vQ, vR;
vector  vfScal1, vfScal2, vTemp;
int     nDegA0, nDegA1;
int     *pnDegQ, *pnDegR;
{

#ifdef DEBUG
        static float r[1000];
#endif

        int i;

/*-----------------------------------------------------------------------*/


        if (nDegA0 < nDegA1)
        {
                is_an_error
                ("PolyDivVA: Numerator polynomial larger than denominator\n");
        }

        if ( (MAX_LEN-1) < nDegA0)
        {
                is_an_error
                ("PolyDivVA: Denomiator polynomial too long\n");
        }

        *pnDegQ = nDegA0-nDegA1;
        *pnDegR = nDegA1-1;

        mapcopfv(vA0,1,vR,1,nDegA0+1);

        mapclrfv(vQ,1,*pnDegQ+1);

        maprcpfv(vA1+nDegA1,1,vTemp,1,vfScal1,1,1);

        for (i=nDegA0-nDegA1; i >= 0; i--)
        {
                mapmulfsv(vfScal1, vR+nDegA1+i,1,vQ+i,1,1);

                mapmsfsvv(vQ+i, vA1, 1, vR+i, 1, vR+i, 1, nDegA1);
                mapnegfv(vR+i, 1, vR+i, 1, nDegA1);
        }

        CheckOrderVA(vR,pnDegR,vTemp,vfScal1,vfScal2);

        return(0);
}
```

Appendix 5.8.4: ConvolveVAfR - Program for Polynomial Mutliplication

```
/*************************************************************************
 *
 *                          ConvolveVA.c
 *
 *************************************************************************
 *
 * DESCRIPTION
 *    this routine performs a linear convolution of two vectors
 *    already present in vector memory (if the vectors correspond
 *    to the coefficients of a polynomial, the convolution is
 *    equivalent to polynomial multiplication).
 *    Convolution is done in the time domain in the form that
 *    shifted and scaled replica of vector vY are added.
 *
 * SYNOPSIS
```

```
*     int ConvolveVA(vX, nLenX, vY, nLenY, vZ, nLenZ)
*     vector vX, vY, vZ;
*     int nLenX, nLenY, nLenZ;
*
* PARAMETERS
*         vX    ...   AP offset for source vector 1
*    nLenX ...   its length
*         vY    ...   AP_Offset for source vector 2
*    nLenY ...   its length
*         vZ    ...   AP memory offset for the resulting vector
*    nLenZ ...   its length
*
* RETURN VALUES
*         0  ... in any event
*
*/

#include <aplib.h>
#include <stdio.h>

typedef int vector;

#ifdef DEBUG
#define DUMP(vY,nLenY)    mapsyncdma(-1,VA0);                      \
                          mapstrfv(vY,1,r,4,nLenY);         \
                          for (i=0; i<=nLenY-1; i++)        \
                                  printf( "[%d] = %f \n",i,r[i]);
#endif

int ConvolveVA(vX, nLenX, vY, nLenY, vZ, nLenZ)
vector vX, vY, vZ;
int nLenX, nLenY, nLenZ;
{
#ifdef DEBUG
        static float r[1000];
#endif
        int i;

#ifdef DEBUG
        if (nLenY < nLenX)
                fprintf(stderr,"\n Swap vX and vY input to increase performance \n");
#endif

        mapclrfv(vZ,1,nLenZ);

        for (i=0; i<= (nLenY-1); i++)
                mapmafsvv(vY+i, vX, 1, vZ+i, 1, vZ+i, 1, nLenX);

        return(0);
}
```

Appendix 5.8.5: CheckOrderVA - Program Removing Leading Zero Coefficients

```
/*******************************************************************
*
*                          CheckOrderVA.c
*
********************************************************************
*
* Returns the adjusted order of the polynomial A in AP memory.
* Leading coefficients which representF floating point zeroes
* have been removed.
*
* NOTE
*    TEMP needs space for (deg_a+1) elements
*    ADDR is one element long
*    MvAX is one element long
*
* SYNOPSIS
*    int CheckOrderVA( vA,deg_a, vTemp, vfScal1, vfScal2)
*    vector vTemp, vfScal1, vfScal2;
```

```
 *    int *pnDegA
 *
 * PARAMETERS
 *       vA  ... vector offset of source vector
 *   *pnDegA  .  degree of polynomial represented by vA
 *     vTemp ... temporary vector
 *    vfScal1 .  temporary scalar for maprcpfv
 *    vfScal2 .  temporary scalar for the maximum coeficient
 *
 * RETURN VALUES
 *      0   ... in any case
 *
 */

#include <aplib.h>

#define FLT_EPSILON 1.0e-05
#define MAX_LEN 512

typedef int vector;

int CheckOrderVA(vA,pnDegA, vTemp, vfScal1, vfScal2)
vector vA, vTemp, vfScal1, vfScal2;
int *pnDegA;
{
        static float a[MAX_LEN];

        int nLenA = *pnDegA + 1;

        /*
         *      Absolute value of the polynomial coefficients
         */
        mapabsfv(vA,1,vTemp,1,nLenA);

        /*
         *      Find the maximum coefficient and normalize
         *      the polynomial with this coefficient
         */
        mapmaxfv(vTemp,1,vfScal1,vfScal2,nLenA);
        maprcpfv(vfScal1,1,vfScal2,1,vfScal1,1,1);

        mapmulfsv(vfScal1,vTemp,1,vTemp,1,nLenA);
        mapsyncdma(-1,VA0);

        mapstrfv(vTemp,1,a,4,nLenA);
        mapbwaitdma();

        /*
         *      If any leading (and previously normalized) coefficient
         *      is small discard it
         */

        while (a[*pnDegA] <  FLT_EPSILON)
                *pnDegA -= 1;

        return (0);
}
```

Appendix 5.8.6: `ArmaPsd` - Program Computing the ARMA Spectrum

```
/*****************************************************************
 *
 *                       ArmaPSD.c
 *
 *****************************************************************
 *
 *  DESCRIPTION
 *    Calculates the PSD estimate from the remainder polynomial
 *    and the co-multiplier polynomial as obtained by the Pade
 *    Arma estimation
 *
```

```
 *  SYNOPSIS
 *     int ArmaPSD(vR, vT, vCoeff, vTempl, nLenFFT,nLogLen)
 *     vector vR, vT, vCoeff, vTempl;
 *     int nLenFFT, nLogLen;
 *
 *
 *  PARAMETERS
 *     vR remainder polynomial (aligned on nLenFFT-boundary)
 *     vT co-multiplier polynomial (aligned on nLenFFT-boundary)
 *     vCoeff     offset for FFT coefficient table
 *     vTempl     auxiliary vector (aligned on nLenFFT-boundary)
 *     nLenFFT    length of the FFT
 *     nLogLen    log2(nLenFFT)
 *
 *  RETURN VALUE
 *     Offset of vector containing the FFT estimate for
 *     the current data set
 */

#include <stdio.h>
#include <aplib.h>

typedef int vector;


static int temp;
#define swap(a,b)      (temp)=(b); (b)=(a); (a)=(temp);

/*
 *       Array processor macro for division of two complex vectors
 *       Implements:
 *               (a+jb)/(c+jd) = (a+jb)(c-jd)/(c^2+d^2)
 *
 *       The coriginal content of the two input vectors is lost !!! (sorry)
 *       Also, the vA, vB, and vC have two be distinct !!! (alas)
 */

#define mapdivcfvv(vA,nIncA,vB,nIncB,vC,nIncC,nItems);                  \
                mapmulcgcfvv(vA,nIncA,vB,nIncB,vC,nIncC,nItems);    \
                mapnrmsqcfv(vB,nIncB,vB,nIncB,nItems);                  \
                maprcpfv(vB,nIncB,vA,nIncA,vB,nIncB,nItems);            \
                mapmulfvv(vC,nIncC,vB,nIncB,vC,nIncC,nItems);           \
                mapmulfvv(vC+1,nIncC,vB,nIncB,vC+1,nIncC,nItems);

#ifdef DEBUG
static float r[1000];
int i;

#define DUMP(vY,nIncY,nLenY)       mapsyncdma(-1,VA0);              \
                                   mapstrfv(vY,nIncY,r,4,nLenY);    \
                                   mapbwaitdma();                   \
                                   for (i=0; i <= nLenY-1; i++)     \
                                           printf("%f\n",r[i]);     \
                                   exit(0);

#endif


int ArmaPSD(vR, vT, vCoeff, vTempl, nLenFFT, nLogLen)
vector vR, vT, vCoeff, vTempl;
int nLenFFT,nLogLen;
{
        int nHalfLenFFT = (nLenFFT>>1);

        /*
         *       Fourier transform of remainder polynomial vR
         */
        maprfftnc(vR,1,vCoeff,2,vTempl,1,nLenFFT);

        if (nLogLen&1)
                swap(vR,vTempl);

        /*
         *       Fourier transform of co-multiplier polynomial vT
```

```
         */
        maprfftnc(vT,1,vCoeff,2,vTempl,1,nLenFFT);

        if (nLogLen&1)
                swap(vT,vTempl);

        /*
         *      The estimate of the Fourier transform for the current data
         *      set is the quotient of remainder polynomial (MA branch) and
         *      co-multiplier polynomial (AR branch)
         */
        mapdivcfvv(vR,2,vT,2,vTempl,2,nHalfLenFFT+1);

        return(vTempl);

}
```

Appendix 5.9: `MeanVel` - Program Computing the Mean Velocity Profile

```
/*****************************************************************************
 *
 *                              MeanVel.c
 *
 *****************************************************************************
 *
 *
 *  DESCRIPTION
 *       This program finds the maximum value in the spectrum of the data
 *       and then gets the variance at this point
 *       These values, taken as the velocity estimates, are appended on the
 *       file containing the previous maxima.
 *       The idea behind this all is to create a velocity profile from the
 *       spectra at the different loactions of the probe volume.
 *       For the conversion frequency --> velocity a conversion factor
 *       is needed
 *
 *  USAGE
 *       -i  Input file containing the input data\n");
 *       [Default: /usr/erk/DSP/DAT/Result.dat]\n");
 *
 *       -o  Output file containing the mean\n");
 *       [Default: /usr/erk/DSP/DAT/MeanVel.dat]\n");
 *
 *       -B  Frequency shift at the mixer
 *
 *       -v  File containing the variance\n");
 *       [Default: /usr/erk/DSP/DAT/VarVel.dat]\n");
 *
 *       -f  Calibration factor for conversion frequency to velocity\n");
 *
 *       -N  Remove file specified under -o first\n");
 *
 *       -h  Print this message\n");
 *
 *
 *
 *
 */

#include <math.h>
#include <sys/file.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include "/usr/erk/DSP/FileOp.h"

double atof();
char *malloc();
```

```
void perror();
void exit();

typedef int bool;

#define READ            04
#define WRITE           02
#define EXISTS          00

#define knRealTime      -20


/*-------------------------------------------------------------------------*/

void Usage()
{
        fprintf(stderr,"\n");
        fprintf(stderr,"This program computes the first and second moment\n");
        fprintf(stderr,"of an input array of data\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"USAGE:\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-f\tCalibration factor for conversion frequency to velocity\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-B\tFrequency shift at DiSA mixer\n");
        fprintf(stderr,"\t [Default: 0 Hz]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-i\tInput file containing the input data\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Result.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-o\tOutput file containing the mean\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/MeanVel.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-v\tFile containing the variance\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/VarVel.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-N\tRemove file specified under -o first\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-T\tSampling frequency [Hz]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-W\tLength of window for determining Doppler frequency\n");
        fprintf(stderr,"\t [Default 11 samples]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-h\tPrint this message\n");
        fprintf(stderr,"\n");

        exit(-1);
}

/*
---------------------------------------------------------------------------
*/


main (argc,argv)
int argc;
char **argv;
{
        float   *pafMeanSpec;

        float   *pafMeanVel;
        float   *pafVarVel;

        float   fFrequencyShift  = 0.0;
        float   fCalibration     = 1.0;
        float   fSampFreq        = 1000000.0;

        float   fSum;

        double  dMax;
        double  dMean;
        double  dMeanSq;

        bool    tDelFile         = 0;
```

```
	bool	tFileExists	= 1;

	int	i=0;
	int	j;
	int	i0=0;

	int	nMeans;
	int	nMeans2;
	int	nSpecLen;

	int	nWindowLen = 11;

	int	chOption;

	FILE	*fpMeanSpec;

	FILE	*fpMeanVel;
	FILE	*fpVarVel;

	static char *pachMeanSpecFile	= "/usr/erk/DSP/DAT/Result.dat";
	static char *pachMeanVelFile	= "/usr/erk/DSP/DAT/MeanVel.dat";
	static char *pachVarVelFile	= "/usr/erk/DSP/DAT/VarVel.dat";

	extern char	*optarg;
	extern int	optind;

/*------------------------------------------------------------------------*/
	/*
	 *	Get real-time priority
	 */

	if ( (int)nice(knRealTime) != knRealTime )
	{
		fprintf(stderr,"\nNice: Got different priority than requested, errno: %d\n", errno);
		perror();
		exit(-1);
	}
/*------------------------------------------------------------------------*/

	while ((chOption = getopt(argc, argv, "hi:o:Nv:f:s:T:B:W:")) != EOF)
	{
		switch (chOption)
		{
			case 'h':
				Usage();
				break;

			case 'B':
				fFrequencyShift = (float)atof(optarg);
				break;

			case 'i':
				pachMeanSpecFile = optarg;
				break;

			case 'v':
				pachVarVelFile = optarg;
				break;

			case 'o':
				pachMeanVelFile = optarg;
				break;

			case 'N':
				tDelFile = 1;
				break;

			case 'f':
				fCalibration = (float)atof(optarg);
				break;

			case 'T':
				fSampFreq = (float)atof(optarg);
				break;
```

```
				case 'W':

					nWindowLen = atoi(optarg);
					break;

				case '?':
					Usage();
					break;
			}
		}

/*-------------------------------------------------------------------------*/

		if (tDelFile)
		{
			if(unlink(pachMeanVelFile)==-1)
			{
				fprintf(stderr,"\nCannot unlink/delete %s, errno: %d\n",pachMeanVelFile, errno);
				perror(pachMeanVelFile);
				exit(-1);
			}

			if(unlink(pachVarVelFile)==-1)
			{
				fprintf(stderr,"\nCannot unlink/delete %s, errno: %d\n",pachVarVelFile, errno);
				perror(pachMeanVelFile);
				exit(-1);
			}
		}

/*-------------------------------------------------------------------------*/


		/*
		 *	If an output file already exists, read it
		 *	otherwise open it for write
		 */

		if ( (access(pachMeanVelFile, READ | WRITE | EXISTS) < 0) || (access(pachVarVelFile, READ | WRITE | EXI
		{
			/*
			 *	Input files do not exist yet, create them
			 */

			FpOpenW(pachMeanVelFile, fpMeanVel)
			FpOpenW(pachVarVelFile, fpVarVel)

			tFileExists	= 0;
			nMeans		= 0;

			pafMeanVel	= (float *)malloc((nMeans+1)<<2);
			pafVarVel	= (float *)malloc((nMeans+1)<<2);

		}
		else
		{

			/*
			 *	Input files exist, open them for read/write access
			 *	Read number of items contained therein
			 */

			FpOpenRWU(pachMeanVelFile,fpMeanVel)
			FpOpenRWU(pachVarVelFile,fpVarVel)

			fscanf(fpMeanVel,"%d:\n",&nMeans);
			fscanf(fpVarVel,"%d:\n",&nMeans2);

			if (nMeans != nMeans2)
			{
				fprintf(stderr,"Files for mean and standard deviation have different length\n");
				exit(-1);
			}
```

```
			pafMeanVel	= (float *)malloc((nMeans+1)<<2);
			pafVarVel	= (float *)malloc((nMeans+1)<<2);

			/*
			 *	Read the velocity profile so far, so that the new data can
			 *	be appended by a write of the whole array back on to the file
			 */
			for(i=0; i<nMeans; i++)
			{
				fscanf(fpMeanVel,"%f\n",(pafMeanVel+i));
				fscanf(fpVarVel,"%f\n",(pafVarVel+i));
			}

			rewind(fpMeanVel);
			rewind(fpVarVel);
		}

		/*
		 *	Write updated number of items to file
		 */
		fprintf(fpMeanVel,"%d:\n",nMeans+1);
		fprintf(fpVarVel,"%d:\n",nMeans+1);

		if (tFileExists)
		{
			/*
			 *	Write back to the file the velocity profile so far
			 */

			for(i=0; i<nMeans;i++)
			{
				fprintf(fpMeanVel, "%f\n", *(pafMeanVel+i));
				fprintf(fpVarVel, "%f\n", *(pafVarVel+i));
			}
		}

		/*
		 *	Open the input file
		 */
		FpOpenR(pachMeanSpecFile,fpMeanSpec)

		fscanf(fpMeanSpec,"%d:\n",&nSpecLen);

		pafMeanSpec	= (float *)malloc(nSpecLen<<2);

		for (i=0; i < nSpecLen; i++)
			fscanf(fpMeanSpec,"%f\n",(pafMeanSpec+i));


		/*
		 *	Find the location where a windowed mean is maximum
		 */

		dMax = 0.0;

		for (i=0; i < (nSpecLen-nWindowLen); i++)
		{
			fSum = 0.0;

			for ( j=0; j < nWindowLen; j++)
				fSum += *(pafMeanSpec + i + j);

			if (dMax < (double)fSum)
			{
				i0 = i;
				dMax = (double)fSum;
			}

		}

		/*
		 *	Compute the first moment of this region
		 *	this will be our Doppler frequency
```

```
	 *	Also compute the variance
	 */

	dMean   = 0.0;
	dMeanSq = 0.0;

	for (j = i0; j < (i0 + nWindowLen); j++)
	{
		dMean   += (double)j * (double)(*(pafMeanSpec + j));
		dMeanSq += (double)j * (double)(j) * (double)(*(pafMeanSpec+j));
	}

	dMean   /= dMax;
	dMeanSq /= dMax;

	dMeanSq -= dMean * dMean;

	/*
	 *	Convert the frequency to velocity
	 */

	dMean    = dMean * (double)fSampFreq / (2.0 * (double)(--nSpecLen)) - (double)fFrequencyShift;
	dMean   *= fCalibration;

	if (0.0 < dMeanSq)
	{
		/*
		 *	Compute the standard deviation in the Doppler frequency estimate
		 */

		dMeanSq  = sqrt (dMeanSq);

		dMeanSq  = dMeanSq * (double)fSampFreq / (2.0 * (double)nSpecLen);
		dMeanSq *= fCalibration;
	}
	else
	{	/*
		 *	If negative variance due to float round-off the nothing at all
		 */

		dMeanSq = 0.0;
	}

	/*
	 *	Append new velocity points of velocity
	 *	profile to already present ones
	 */
	fprintf(fpMeanVel,"%f\n",dMean);
	fprintf(fpVarVel,"%f\n",dMeanSq);

	fclose(fpMeanVel);
	fclose(fpVarVel);
	fclose(fpMeanSpec);

	exit(0);
}
```

Appendix 5.10: `DoPlot` - Plot Program

```
/***************************************************************************
*                                                                         *
*                              DOPLOT.C                                   *
*                                                                         *
```

```
************************************************************************
*
*  DESCRIPTION
*    Plots the data contained in two input files
*    This routine also features an arbitrary linear scaling of the x-axis
*
*  USAGE
*
*  COMPILER OPTIONS
*    -DMARK marks the points in the graph with circles
*
*/

#include <math.h>
#include <libmp.h>
#include <stdio.h>
#include <stdio.h>
#include <errno.h>
#include "/usr/erk/DSP/FileOp.h"

#define knRealTime       -20
#define knMaxNumLabels   100
#define NUM_DIGITS_IN_FLOAT  15 /* Each label has 15 digits         */

#define FOREVER   for(;;)

typedef int bool;

void perror();
void exit();
double atof();
char *malloc();

/*-------------------------------------------------------------------*/

void Usage()
{
        fprintf(stderr,"\n");
        fprintf(stderr,"This program plots the data in one data file and\n");
        fprintf(stderr,"subtracts and adds the data (same length) of another file\n");
        fprintf(stderr,"Idea: plot (mean+-standard deviation)\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"USAGE:\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-i\tInput file containing the data\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Result.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-n\tNo logarithmic scale on y-axis\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-v\tFile containing the second set of data\n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Variance.dat]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-o\tOutput graphics file \n");
        fprintf(stderr,"\t [Default: /usr/erk/DSP/DAT/Plot.graph]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-l\tPut labels on x-axis as specified under -L and -H\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-L\tLabel of first data point on x-axis\n");
        fprintf(stderr,"\tREQUIRED\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-H\tLabel of last data point on x-axis\n");
        fprintf(stderr,"\tREQUIRED\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-N\tNumber of labels on x-axis\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-T\tText to be displayed on graph\n");
        fprintf(stderr,"\t [Default: DoPlot Output]\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-X\tTitle for x-Axis\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-Y\tTitle for y-Axis\n");
        fprintf(stderr,"\n");
        fprintf(stderr,"-h\t Print this message\n");
        fprintf(stderr,"\n");
```

```
        exit(-1);
}

/*
==========================================================================
*/

main(argc, argv)
int argc;
char **argv;
{
#ifdef MARK
        int     anChars[3], anBundle[3];
#endif

        float   *pfData1;
        float   *pfData2;

        float   fLow    = -1.0;
        float   fHi     = -1.0;
        float   fTicInt;
        float   fDelX;

        int     i;
        int     nItems;
        int     nItems2;
        int     nLabels = 11;

        bool    tLogScale = 1;
        bool    tLabels = 0;

        FILE    *fpInput;
        FILE    *fpVariance;

        char    ach2dAxisX[knMaxNumLabels][NUM_DIGITS_IN_FLOAT];
        char    *pachStr[knMaxNumLabels];

        int     chOption;

        static char *pachInputFile      = "/usr/erk/DSP/DAT/Result.dat";
        static char *pachVarFile = "/usr/erk/DSP/DAT/SpecVar.dat";
        static char *pachGraphFile      = "/usr/erk/DSP/DAT/Plot.graph";
        static char *pachRemark         = "Output DoPlot";
        static char *pachAxisXTitle     = "";
        static char *pachAxisYTitle     = "";

        long int alGca[SIZEOFGCA];

        extern char *optarg;
        extern int optind;

/*--------------------------------------------------------------------------*/
        /*
         *      Get real-time priority
         */

        if ( (int)nice(knRealTime) != knRealTime )
        {
                fprintf(stderr,"\nNice: Got different priority than requested, errno: %d\n", errno);
                perror();
                exit(-1);
        }

/*--------------------------------------------------------------------------*/
        while ((chOption = getopt(argc, argv, "i:nv:o:lL:H:T:X:Y:N:h")) != EOF)
        {
                switch (chOption)
                {
                        case 'h':
                                Usage();
                                break;

                        case 'i':
```

```
                                pachInputFile = optarg;
                                break;

                        case 'n':
                                tLogScale = 0;
                                break;

                        case 'v':
                                pachVarFile = optarg;
                                break;

                        case 'o':
                                pachGraphFile = optarg;
                                break;

                        case 'l':
                                tLabels=1;
                                break;

                        case 'L':
                                fLow = (float)atof(optarg);
                                break;

                        case 'N':
                                nLabels = atoi(optarg);
                                break;

                        case 'H':
                                fHi = (float)atof(optarg);
                                break;

                        case 'T':
                                pachRemark = optarg;
                                break;

                        case 'X':
                                pachAxisXTitle = optarg;
                                break;

                        case 'Y':
                                pachAxisYTitle = optarg;
                                break;

                        case '?':
                                Usage();
                                break;
                }
        }

        if (knMaxNumLabels < nLabels)
        {
                fprintf(stderr,"Number of labels too large\n");
                exit(-1);
        }

        if (tLabels)
        {
                if (fLow < 0.0)
                {
                        fprintf(stderr,"\nNo or negative first label defined\n");
                        Usage();
                }

                if (fHi < 0.0)
                {
                        fprintf(stderr,"\nNo or negative last label defined\n");
                        Usage();
                }
        }

        FpOpenR(pachInputFile,fpInput)

        FpOpenR(pachVarFile,fpVariance)

```

```
        /*
         *      The first entries in the input file are the number of data
         *      points in the file
         */
        fscanf(fpInput,"%d:\n", &nItems);
        fscanf(fpVariance,"%d:\n", &nItems2);

        if (nItems != nItems2)
                fprintf(stderr,"DoPlot: The two data files have not same length\n");

/*------------------------------------------------------------------*/

        pfData1 = (float *)malloc(nItems<<2);
        pfData2 = (float *)malloc(nItems<<2);

/*------------------------------------------------------------------*/

        for (i=0; i <= nItems-1; i++)
        {
                fscanf(fpInput,"%f\n", (pfData1+i));
                fscanf(fpVariance,"%f\n",(pfData2+i));

                if (tLogScale)
                {
                        if ( (*(pfData2+i) < 0) || (*(pfData1+i) < 0) )
                        {
                                fprintf(stderr,"\nNegative value, system does not permit log-scale\n");
                                tLogScale = 0;
                        }
                }
        }

/*------------------------------------------------------------------*/

        fTicInt = (float)(nItems) / (float)(nLabels-1);

        /*
         *      nLabels marks on the x axis
         */
        fDelX=(fHi-fLow)/(float)(nLabels-1);

        /*
         *      the nLabels labels for the x-Axis
         */
        for (i=0; i<=nLabels-1; i++)
        {
                sprintf(ach2dAxisX[i], "%f", (fLow + i*fDelX));
                pachStr[i] = ach2dAxisX[i];
        }


        mpinit(alGca);

        /*
         *      y-axis with logarithmic scale
         */
        if (tLogScale)
                mplogax(alGca,2,3);

        mplotsrcy(alGca,1,nItems,0,pfData1,"F",1,1,NULL,NULL);
        mplotsrcy(alGca,2,nItems,0,pfData2,"F",1,1,NULL,NULL);

#ifdef MARK
        anBundle[0]=20;
        anBundle[1]=22;

        anChars[0]=0;
        anChars[1]=1;

        mplines(alGca,2,anBundle,anChars);

        /*
         *      mark each data point with "o"
         */
```

```
        mplotchrs(alGca,"o",1,NULL,NULL);
#endif

        /*
         *      send plot to mc graphics screen
         */
        mpdevice(alGca,"xmcd",2,0);

        if (tLabels)
        {
                mpaxvals(alGca,1,UNDEF,UNDEF,fTicInt,(nLabels-1));
                /*
                 *      labels
                 */
                mplabel(alGca, 1, nLabels, 1, -45.0, -1, -1, pachStr);
        }


        /*
         *      axis titles
         */
        mptitle(alGca,1,-1,-1,pachAxisXTitle);
        mptitle(alGca,2,-1,-1,pachAxisYTitle);
        mptitle(alGca,4,-1,-1,"     ");
        mptitle(alGca,4,-1,-1,pachRemark);

        /*
         *      save graph on file
         */
        mpfile(alGca,pachGraphFile,1,2);

        mplot(alGca,0,0,0);
        mpend(alGca);

        return(0);

}

```

Appendix 5.11: `MasterPlan` - Shell Script / User Interface

```

# This is the stomach (or the arm [see Agrippina: De ventro et membris])
# of all the routines for LDA signal processing:
# it takes a reasonable amount of options (for the sake of clarity far
# less than all the programs would permit) and cares for the
# correct order of processing.
#
# Note that it uses some default values of the programs, for example
# the names internally used by the programs for the input and output
# files.
# If somebody is bothered by that she/he can control the names of these
# files with command line switches.

# Some reasonable definitions for a start

# Sampling frequency
SAMPFREQ=1000000

# Length of one batch as sampled by the routine SampleData [msec]
SAMPDUR=409

# Number of bursts required at each point of measurement
NBURSTS=0

# Required minimum duration of bursts for the routine GetBursts [Samples]
MINDUR=10

```

```
# Required maximum duration of bursts for the routine GetBursts [Samples]
MAXDUR=512

# Flag for the removal of files from previous runs of this script
DELFILE=1

# Number of measuring points for the velocity profile
NPOS=2

# Spectral estimator to use, default is Pade estimator
METHOD=2

# Flag dis/enabling digital prefiltering of the data
FILTERON=0

#Gain in the A/D converter (preamplification of the signal by 2**${GAIN}
GAIN=0

#Frequency Shift at the DISA Mixer
FREQSHIFT=40000

#Calibration factor for conversion m/s to Hz (velocity to Doppler frequency)
CALFAC=1.0

#-----------------------------------------------------------------------

# The usual on-line documentation

Usage () { \
        echo
        echo "USAGE:";\
        echo ""
        echo "-B Frequency shift at the DISA frequency mixer unit [40000 Hz]"
        echo "-b Number of bursts to collect [0]";\
        echo "-C Calibration factor [1.0 (m/s)/Hz]";\
        echo "-d Expected minimum duration of bursts [10 Samples]";\
        echo "-F Enable digital prefiltering [Off=0]";\
        echo "-f Sampling frequency [1000000 Hz]";\
        echo "-G Gain in A/D converter [0]";\
        echo "-n Number of measuring points [2]";\
        echo "-s Method for spectral estimation [Pade=2]";\
        echo "-t Sampling Duration [0 msec]";\
        echo "-u Use results from a previous run of this  script [Off=0]";\
        echo "";\
        echo ">>>>The -b option must be present<<<<";\
        echo "";\
        exit 2;\
        }

#-----------------------------------------------------------------------

# All programs are designed to exit with (-1) upon an error
# If (-1=255) was the last exit status then skip the whole business
# and return to the shell

ErrorCheck () {\
        if [ $? = 255 ]
        then
                echo "Last Program exited with error, back to shell"
                exit 1
        fi;\
        }

#-----------------------------------------------------------------------

# Parse the argument line

set -- `getopt B:b:C:d:f:FG:hn:s:t:u $*`

if [ $? != 0 ]
then
        Usage
fi
```

```
for i in $*
do
	case $i in
		-B)	FREQSHIFT=$2; shift 2;;
		-b)	NBURSTS=$2; shift 2;;
		-C)	CALFAC=$2; shift 2;;
		-d)	MINDUR=$2;shift 2;;
		-f)	SAMPFREQ=$2;shift 2;;
		-F)	FILTERON=1;shift;;
		-G)	GAIN=$2;shift 2;;
		-n)	NPOS=$2;shift 2;;
		-s)	METHOD=$2;shift 2;;
		-t)	SAMPDUR=$2;shift 2;;
		-u)	DELFILE=0;shift;;
		-h)	Usage;shift;;
	esac
done

# the Nyquist frequency
NYQUIST=`expr $SAMPFREQ / 2`
VELPOS=0
CURPOS=0

if [ $NBURSTS = 0 ]
then
	echo
	echo "The -b option has to be specified"
	Usage
fi

/usr/bin/clear

if [ $DELFILE = 1 ]
then
	# These are default output files of the routines MeanSpec and
	# MeanVel.
	# They can be changed via command line switches

	echo "Deleting intermediate files at morgana before starting"

	rm /usr/erk/DSP/DAT/Working.dat
	rm /usr/erk/DSP/DAT/MeanVel.dat
	rm /usr/erk/DSP/DAT/VarVel.dat
	rm /usr/erk/DSP/DAT/Bursts.dat
	rm /usr/erk/DSP/DAT/NOfBursts.dat
	rm /usr/erk/DSP/DAT/Threshold.dat
	rm /usr/erk/DSP/DAT/SpecVar.dat
	rm /usr/erk/DSP/DAT/Filtered.dat
else
	echo "Using old results"
	echo "Enter number of measurement positions already done:"

	CURPOS=""
	until [ $CURPOS ]
	do
		read CURPOS
	done

	VELPOS=$CURPOS

	echo "Saving old mean velocity profile in /usr/erk//DSP/DAT/MeanVel.SAV"
	mv /usr/erk/DSP/DAT/MeanVel.dat /usr/erk/DSP/DAT/MeanVel.SAV

	echo "Saving old mean velocity profile in /usr/erk//DSP/DAT/VarVel.SAV"
	mv /usr/erk/DSP/DAT/VarVel.dat /usr/erk/DSP/DAT/VarVel.SAV

fi

# The following series of programs is executed as long as there
# are not enough bursts found

# The program GetBursts exit value equals the number of burst it has
# processed so far

```

```
OLDNYQUIST=$NYQUIST
OLDSAMPFREQ=$SAMPFREQ
OLDFREQSHIFT=$FREQSHIFT

PROCESSED=0;

until [ $CURPOS = $NPOS ]
do

        CURPOS=`expr $CURPOS + 1`

        until [ $PROCESSED = $NBURSTS ]
        do

                # The program which samples the data and transfers them to morgana
                /usr/bin/clear
                echo ">>>>>>>>>>SampleData@merlin<<<<<<<<<<<<"
                rsh merlin /usr/erk/DSP/SAMPLEDATA/SampleData -t $SAMPDUR -f $SAMPFREQ -G $GAIN

                # No error check as rsh does not pass exit status

                /usr/bin/clear

                # Program on morgana which filters the data and gets the rms value
                if [ $FILTERON = 1 ]
                then
                        echo ">>>>>>>>>>FilterData@morgana<<<<<<<<<<<<"
                         /usr/erk/DSP/FILTERDATA/FilterData
                        ErrorCheck
                        echo
                        echo ">>>>>>>>>>Variance@morgana<<<<<<<<<<<<"
                         /usr/erk/DSP/VARIANCE/Variance
                        ErrorCheck
                else
                        echo ">>>>>>>>>>Variance@morgana<<<<<<<<<<<<"
                         /usr/erk/DSP/VARIANCE/Variance -S -C -i /usr/erk/DSP/DAT/RawData.dat
                        ErrorCheck
                fi

                # Program on morgana for isolating and validating LDA bursts
                echo
                echo ">>>>>>>>>>GetBursts@morgana<<<<<<<<<<<<"
                echo
                echo "Minimum length of burst is $MINDUR "
                echo "Found already $PROCESSED bursts of $NBURSTS"
                echo
                /usr/erk/DSP/GETBURSTS/GetBursts -Q 0.1 -b $NBURSTS -C 2000 -B 300 -A 100 -M -d $MINDUR -D

                PROCESSED=$?
                echo "$PROCESSED bursts found"
                echo "$NBURSTS needed"

                # Program on morgana for obtaining the mean spectrum and its variance
                echo
                echo ">>>>>>>>>>MeanSpec@morgana<<<<<<<<<<<<"
                 /usr/erk/DSP/SPECANALYSIS/MeanSpec -D $MAXDUR -m $METHOD

                ErrorCheck

        done

        PROCESSED=0;

        ANSWER=9

        until [ $ANSWER = 0 ]
        do

                /usr/bin/clear

                echo ""
                echo "                    <<<<<<requires ATTENTION:>>>>>>"
                echo
                echo "    $CURPOS measurement positions out of $NPOS "
```

```
echo "    Calibration factor: $CALFAC "
echo ""
echo "Type:"
echo " 1 ... good measurement & ready for next position"
echo " 2 ... not a good measurement, repeat "
echo " 3 ... plot the spectrum at current position "
echo " 4 ... plot the velocity profile after $VELPOS positions"
echo " 5 ... plot final velocity profile & exit"
echo " 6 ... Change the values listed below"
echo
echo " 9 ... Exit "
echo
echo " Current sampling frequency:              $SAMPFREQ [Hz] "
echo " Number of bursts per position:           $NBURSTS "
echo " Minimum number of samples in burst:      $MINDUR "
echo " Duration of sampling:                       $SAMPDUR [msec]"
echo " Opto-electronic frequency shift:            $FREQSHIFT [Hz]"
echo
echo " Sampling frequency at previous position: $OLDSAMPFREQ"
echo " Opto-electronic frequency shift at previous position: $OLDFREQSHIFT"

ANSWER=""
until [ $ANSWER ]
do
        read ANSWER
done

#-----------------------------------------------------

if [ $ANSWER = 1 ]
then
        echo ">>>>>>>>>>MeanVel@morgana<<<<<<<<<<<"

        # For Doppler frequency at previous position we
        # need the old sampling frequency and the old
        # frequency shift
        /usr/erk/DSP/MEANVEL/MeanVel -T $OLDSAMPFREQ -B $OLDFREQSHIFT -f $CALFAC
        ErrorCheck

        rm /usr/erk/DSP/DAT/Working.dat
        rm /usr/DSP/DAT/NOfBursts.dat

        OLDNYQUIST=$NYQUIST
        OLDSAMPFREQ=$SAMPFREQ
        OLDFREQSHIFT=$FREQSHIFT

        ANSWER=0
        VELPOS=`expr $VELPOS + 1`
fi

#-----------------------------------------------------

if [ $ANSWER = 2 ]
then

        echo "Repeat measurement at same position"

        echo "Deleting intermediate files at morgana"
        rm /usr/erk/DSP/DAT/Working.dat

        CURPOS=`expr $CURPOS - 1`

        ANSWER=0
fi

#-----------------------------------------------------

if [ $ANSWER = 3 ]
then
        echo  ">>>>>>>>>>DoPlot@morgana<<<<<<<<<<<"
        /usr/erk/DSP/PLOT/DoPlot -L 0.0 -H $OLDNYQUIST -T "Spectrum at ${CURPOS}th pos

        ErrorCheck
        echo
```

```
		echo "Save current spectrum in PostScript format ??? [y/*]"
		echo "Filename /usr/erk/DSP/DAT/Pos${CURPOS}.Spec"

		ANSWER=""
		until [ $ANSWER ]
		do
			read ANSWER
		done

		if [ $ANSWER = y ]
		then
			echo
			echo "Saving current spectrum"
			echo
			psgps /usr/erk/DSP/DAT/Plot.graph > /usr/erk/DSP/DAT/Pos${CURPOS}.PS &
			echo
			echo "Done ..."
		fi
	fi

	#-----------------------------------------------------

	if [ $ANSWER = 4 ]
	then
		# the files specified under -i and -v are the
		# default output files of the routine MeanVel
		# see also the call to DoPlot below

		echo  ">>>>>>>>>>DoPlot@morgana<<<<<<<<<<<<"
		/usr/erk/DSP/PLOT/DoPlot -T "Velocity profile after ${VELPOS} positions" -i /usr/e


		ErrorCheck

		echo
		echo "Save current velocity profile in PostScript format ??? [y/*]"
		echo "Filename:  /usr/erk/DSP/DAT/Prof${CURPOS}.PS"

		ANSWER=""
		until [ $ANSWER ]
		do
			read ANSWER
		done

		if [ $ANSWER = y ]
		then
			echo
			echo "Saving current velocity profile"
			echo
			psgps /usr/erk/DSP/DAT/Plot.graph > /usr/erk/DSP/DAT/Prof${CURPOS}.PS &
			echo
			echo "Done ..."
		fi
	fi

	#-----------------------------------------------------
	if [ $ANSWER = 5 ]
	then
		echo "Experiment finished"
		echo "Plotting final velocity profile"
		break 2;
	fi

	#-----------------------------------------------------

	if [ $ANSWER = 6 ]
	then
		/usr/bin/clear

		echo "Current sampling frequency is: ${SAMPFREQ} [Hz]"
		echo "Enter new one"

		SAMPFREQ=""
		until [ $SAMPFREQ ]
```

```
			do
				read SAMPFREQ
			done

			NYQUIST=`expr $SAMPFREQ / 2`

			echo "Current necessary number of bursts is: ${NBURSTS} "
			echo "Enter new one"


			NBURSTS=""
			until [ $NBURSTS ]
			do
				read NBURSTS
			done

			echo "Current mimimum number of samples per burst is ${MINDUR} "
			echo "Enter new one"


			MINDUR=""
			until [ $MINDUR ]
			do
				read MINDUR
			done

			echo "Current number of measurement positions is ${NPOS}"
			echo "Enter new one"

			NPOS=""
			until [ $NPOS ]
			do
				read NPOS
			done


			echo "Duration of sampling is ${SAMPDUR} [msec]"
			echo "Enter new one"

			SAMPDUR=""
			until [ $SAMPDUR ]
			do
				read SAMPDUR
			done

			echo "Opto-electronic frequency shift is ${FREQSHIFT} [Hz]"
			echo "Enter new one"

			FREQSHIFT=""
			until [ $FREQSHIFT ]
			do
				read FREQSHIFT
			done



		fi

		#----------------------------------------------------

		if [ $ANSWER =  9 ]
		then
			echo
			echo "Exiting..."
			echo
			exit
		fi

		#----------------------------------------------------

	done

done
```

```
echo
echo ">>>>>>>>>>MeanVel@morgana<<<<<<<<<<<<"
/usr/erk/DSP/MEANVEL/MeanVel -T $SAMPFREQ -B $FREQSHIFT -f $CALFAC
ErrorCheck

# At this point, the measurement at one point is considered to be
# finished

# Program on morgana to plot the data
echo  ">>>>>>>>>>DoPlot@morgana<<<<<<<<<<<<"
/usr/erk/DSP/PLOT/DoPlot -T "Velocity profile after ${CURPOS} positions" -i /usr/erk/DSP/DAT/MeanVel.dat -v

ErrorCheck

echo
echo "Plot the profile before exiting ??? [y/*]"
ANSWER=""
until [ $ANSWER ]
do
        read ANSWER
done

if [ $ANSWER = y ]
then
        rcp /usr/erk/DSP/DAT/Plot.graph merlin:/usr/data/erk/VelProf.g
        rsh merlin psgps /usr/data/erk/VelProf.g "|" lp
        echo
        echo "Done ..."
fi

/usr/bin/clear
echo
echo
echo "                              Pheeewww!"
echo
echo

exit 0
```