FFTPAK 87/32 Lite

Shareware Version 1.0

Precision Plus Software
38 Longview CRT
London, ON N6K 4J1
Canada

Tel: (519) 657-0633
Fax: (519) 657-0283
E-Mail: 70401.2606@compuserve.com

June 28, 1995

Abstract

This package contains *FFTPAK 87/32 Lite*, Shareware Version 1.0 — a 32-bit assembler coded Fast Fourier Transform (FFT) Library. All software and documentation in this package is © Copyright 1995, Precision Plus Software. All rights reserved.

This software is being distributed as Shareware. If you use this software for 30 days or more, you are expected to register it. Please read the chapter "Shareware" in the documentation for further details.

DISCLAIMER OF WARRANTY

THIS SOFTWARE AND MANUAL ARE PROVIDED "AS IS" AND WITHOUT WARRANTIES AS TO PERFORMANCE OR MERCHANTABILITY.

THIS SOFTWARE IS PROVIDED WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES WHATSOEVER. BECAUSE OF THE DIVERSITY OF CONDITIONS AND HARDWARE UNDER WHICH THIS SOFTWARE MAY BE USED, NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE IS OFFERED. THE USER IS ADVISED TO TEST THIS SOFTWARE THOROUGHLY BEFORE RELYING ON IT. THE USER MUST ASSUME THE ENTIRE RISK OF USING THIS SOFTWARE. HAVING SAID ALL THIS, PRECISION PLUS SOFTWARE NEVERTHELESS HAS AN INTEREST IN MAINTAINING THE RELIABILITY AND EFFICIENCY OF THE SOFTWARE AND WILL MAKE CORRECTIONS AND IMPROVEMENTS WHEREVER PRACTICAL. (USERS ARE NOT LIKELY TO USE AND REGISTER A SHAREWARE PRODUCT THAT DOESN'T PERFORM VERY WELL).

ALL THE INFORMATION IN THIS DOCUMENT IS BELIEVED TO BE CORRECT AT THE TIME OF PUBLICATION. PRECISION PLUS SOFTWARE DOES, HOWEVER, RESERVE THE RIGHT TO MAKE ANY CHANGES IN PRODUCT SPECIFICATIONS AND/OR AVAILABILITY WITHOUT NOTICE.

DOCUMENTATION

Full documentation for this product is provided in postscript format in the file `MANUAL.PS` and in HP LaserJet Plus compatible format in the file `MANUAL.HP`. This manual was printed by printing one of these files.

Contents

Chapter 1

Introduction

1.1 General

Thank you for your interest in the *FFTPAK 87/32 Lite* Fast Fourier Transform (FFT) library. This product is one of a family of high performance scientific and engineering software packages by Precision Plus Software. By distributing this software package as Shareware, we invite you to evaluate it for up to 30 days. If you use this software for more than 30 days, you are expected to register it. Please read the chapter "Shareware" in the documentation for details. We would also like to acquaint you with other scientific and engineering software packages published by Precision Plus Software.

FFTPAK 87/32 Lite is a set of 32-bit assembler coded FFT functions, as well as related scaling and unscrambling functions. The FFT functions in this library are faster than those of any other available FFT library. Unlike earlier 16-bit functions, they are not limited to small FFT's (64 KB) but can handle FFT's of practically unlimited size.

FFTPAK 87/32 Lite shareware version includes a set of double precision complex FFT routines. If this package meets your needs, you can become a registered and licensed user for the modest fee of $89 (US). When you register the software, you will be sent the latest registered version, which also includes a complete set of even faster single precision functions and a printed manual.

For users with a broader set of needs, we also sell the professional version *FFTPAK 87/32*, which includes all the functions in *FFTPAK 87/32 Lite*, as well as special 32-bit assembler coded functions for real valued data, two-dimensional FFTs, fast cosine transforms, fast sine transforms, real and complex correlation, real and complex autocorrelation, real and complex convolution, and various one and two dimensional window functions. The cost of *FFTPAK 87/32* is \$149 (US). We believe that this is the best and fasted FFT library for the PC available at any price.

A special 32-bit integer FFT transform package *FFTPAK INT/32* is also available for processing integer data, such as normally acquired by a real-time data acquisition system. This package includes 32-bit integer FFTs for real and complex data. The integer functions do not require a math coprocessor, and have accuracy comparable to single precision floating point FFTs. However, they run faster than floating point FFTs, particularly on the Pentium processor which has two parallel integer execution units. The cost of *FFTPAK INT/32* is \$199 (US).

For 16-bit DOS and Windows programming, a separate product *FFTPAK 87/16* is available. This package is functionally identical to *FFTPAK 87/32*, but is designed to work with 80286/80287 and earlier processors. The FFT functions in *FFTPAK 87/16* are limited to 64 KB data size. (For a single precision real valued FFT, the largest possible transform size is 16384.) The FFT functions in this package are almost as fast as those in *FFTPAK 87/32*. The cost of *FFTPAK 87/16* is \$149 (US).

The complete source code for *FFTPAK 87/32* is available for licensing. If you require the source code, please inquire for current pricing and license terms. However, unless you are interested in looking at thousands of lines of assembler code, you will probably find that the binary versions of the library are all that you need.

The technology used to produce the *FFTPAK 87/32*, *FFTPAK 87/16* and *FFTPAK INT/32* libraries is not limited to Intel 80x86 processors, but is retargetable to other processors, including DSP and RISC processors. If you have such a special application we may be able to help.

For general mathematical analysis, including solution of linear equation, solution of nonlinear equations, optimization, vector and matrix manipulation, etc., 32-bit *MATHPAK 87/32* and 16-bit *MATHPAK 87* mathematical libraries are available. Almost every major compiler under DOS, Windows and OS/2 is supported. The numerically intensive portions of these libraries are coded in assembler for speed. *MATHPAK 87* has over 10,000 users in universities and industry. The cost of *MATHPAK 87* is \$129 (US) (object code only) or \$198 (US) with complete source code.

We hope that the *FFTPAK 87/32 Lite* library will prove itself worthy of your support.

1.2 Hardware Requirements

FFTPAK 87/32 Lite will run on a 80386/80387 or later processors. It was developed under OS/2 Warp using Borland C++, taking advantage of Borland's inline assembler. Under OS/2, a math coprocessor is not required (although highly recommended), since math coprocessor emulation is provided by the operating system if the math coprocessor is not present. The DOS and Windows versions of the *FFTPAK 87/32* libraries will run under emulation if an emulator library is linked to your program.

1.3 Software Requirements

FFTPAK 87/32 Lite requires OS/2 2.0 (or later), Windows 95, Win32, or a 32-bit DOS extender to run. (As mentioned above, a separate product *FFTPAK 87/16* is available to support 16-bit DOS and Windows users.)

1.4 Software Installation

FFTPAK 87/32 Lite is not a stand-alone program, but a library of functions that you link to your program. For OS/2, Windows 95, and Win32 users, *FFTPAK 87/32 Lite* provides a standard LIB file that contains the objects that you link to when building your program. A 32-bit DLL (dynamic link library) that can be linked to at run-time can also be provided to registered users upon special request.

For successful linking, you need to place the *FFTPAK 87/32* LIB (or DLL) files where they can be found at link time. Please follow the guidelines that come with your compiler about linking to external routines.

For C and C++ programmers, the header file `FFTPAK87.H` is provided with prototypes for all the functions. A sample C program `fftdemo.c` is provided to illustrate how to compile and link a typical program.

Please be sure to read the `READ.ME` since this file may contain hints and last minute changes that you should know about.

Chapter 2

FFT Basics

2.1 Background

There are a number of textbooks that give a good account of the theory and application of the FFT. A few good books are:

1. E.O. Brigham, *The Fast Fourier Transform*, Prentice Hall, Englewood Cliffs, N.J., 1974.

2. L.R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice Hall, Englewood Cliffs, N.J., 1975.

3. H.J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, N.Y., 1982.

4. D.F Elliot and K.R. Rao, *Fast Transforms: Algorithms, Analyses, Applications*, Academic Press, N.Y., 1982.

5. C.S. Burrus and T.W. Parks, *DFT/FFT and Convolution Algorithms*, John Wiley & Sons, 1985.

6. W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, *Numerical Recipes in C*, Second Edition, Cambridge University Press, 1992.

For a detailed understanding of the FFT, you should consult one or more of the these references. For the purposes of this manual, discussion of theory is brief. This manual is not intended to be a textbook on the FFT.

The importance of the FFT since its first successful implementation by Cooley and Tukey[1] has been nothing short of staggering. Applications of the FFT have been found in diverse fields ranging from digital signal processing to solution of nonlinear integral equations in molecular physics. Cooley and Tukey were not the first to discover the "tricks" that are at the core of the FFT, but the publication of their algorithm quickly lead to its widespread use.

The FFT is in essence a clever way of calculating a discrete Fourier transform (DFT) while drastically reduces computation time from $O(N^2)$ to $O(N \log_2 N)$. For large data sets, the decrease in computation time may range from a factor of a thousand to a million or more.

In spite of the efficiency of the FFT algorithm, every additional improvement in efficiency, even by a factor of two or three is welcome. This is because in many applications FFT computation is still a major computational bottleneck, and it is likely to remain so for the foreseeable future. For example, in image processing, resolution is often limited by the rate at which gathered data can be processed. If we can process the data faster, the image becomes sharper. A faster FFT algorithm, therefore directly contributes to better image quality. Similarly, in the solution of convolution type integral equations, a faster FFT algorithm allows larger problems to be solved economically.

2.2 Theory

A Fourier transform $X(f)$ in frequency domain f of a signal $x(t)$ in the time domain is defined as

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-2\pi i f t} dt. \tag{2.1}$$

The inverse Fourier transform is

$$x(t) = \int_{-\infty}^{\infty} X(f) e^{2\pi i f t} df. \tag{2.2}$$

where $i = \sqrt{-1}$. In these equations, if t is measured in seconds, then the frequency f is in cycles per second (Hz)[2]. The Fourier transform may therefore be viewed as an expansion of a signal as a superposition of exponentials,

[1] J.W. Cooley and J.W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, **Math. Comput. 19**, (April 1965), 297–301.

[2] In some texts, the sign in the exponential terms in the forward transform and inverse transforms is reversed. This is just a matter of convention.

in which the frequencies take on all real values. The Fourier transform is a "continuous-frequency" signal representation with no requirement that the signal be periodic.

A discrete-time signal is usually obtained from a continuous-time signal by sampling at equally spaced intervals in time. If the discrete-time signal is denoted as x_n then the sampling of $x(t)$ every T seconds gives:

$$x_n = x(nT), \quad n = \ldots, -2, -1, 0, 1, 2, \ldots \tag{2.3}$$

In the case of a bandlimited signal, with highest frequency B Hz, it can be uniquely recovered from its samples as long as the sampling rate is higher than $2B$ samples per second. Lower sampling rates may result in a phonomenon known as aliasing, where the original signal cannot be unambiguously reconstructed. The frequency $2B$ is known as the Nyquist critical frequency.

By bandlimited, it is meant that the truncated (bandlimited) inverse Fourier transform written as

$$x'(t) = \int_{-B}^{B} X(f) e^{2\pi i f t} df \tag{2.4}$$

is the best approximation to $x(t)$ using frequencies limited to the band $(-B, B)$ in the sense that the integrated squared error, defined as

$$E = \int_{-\infty}^{\infty} \left[x(t) - x'(t)\right]^2 dt \tag{2.5}$$

is minimized.

Suppose that we have N consecutive sampled values x_n at times $t_n = nT$ where $n = 0, 1, \ldots, N-1$. Suppose also that N is even (for convenience). Then if we estimate the Fourier transform $X(f)$ corresponding to the lower and upper limits of the Nyquist critical frequency range:

$$f_k = \frac{k}{NT}, \quad k = -\frac{N}{2}, \ldots, \frac{N}{2}, \tag{2.6}$$

by approximating the integral in the continuous-time transform by a discrete sum, it follows that

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n / N} \tag{2.7}$$

and

$$X(f_k) \approx T X_k. \tag{2.8}$$

The discrete *inverse* Fourier transform that recovers the set of x_n's exactly from the X_k's is

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{2\pi i k n / N}. \tag{2.9}$$

The only difference from the forward transform is the factor $1/N$ and the sign in the exponential term. It should be noted that to obtain reciprocity between the forward and inverse transforms, the $1/N$ factor could have been applied to the forward transform instead of the inverse transform. This is largely a matter of convenience.

For two dimensional DFTs, given a two dimensional array x, the two dimensional forward X is defined as:

$$X_{k,l} = \sum_{n=0}^{N_1-1} \sum_{m=0}^{N_2-1} x_{n,m} \exp[-i2\pi(nk/N_1 + ml/N_2)]. \tag{2.10}$$

The inverse transform is defined as:

$$x_{n,m} = \frac{1}{N_1 N_2} \sum_{k=0}^{N_1-1} \sum_{l=0}^{N_2-1} X_{k,l} \exp[+i2\pi(nk/N_1 + ml/N_2)]. \tag{2.11}$$

From these definitions it follows that a forward transform followed by an inverse transforms will recover the original vector.

Until the mid 1960's, the usual method for calculating a DFT was to directly apply the above equations for the forward and inverse transforms. The number of complex multiplications was therefore N^2 for a 1 dimensional transform.

A particularly lucid derivation of the FFT algorithm is given by Danielson and Lanczos[3] and Singleton[4].

Danielson and Lancsoz showed that a discrete Fourier transform of length N can be rewritten as the sum of two discrete Fourier transforms of length $N/2$, with one formed from the even-numbered points, and the other from the odd-numbered points of the original N. The proof is as follows:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i n k / N}$$

[3]G.C. Danielson and C. Lanczos,*Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids*, **J. Franklin Inst. 233** 365–380; 435–452 (1942).

[4]R.C. Singleton, *On Computing the Fast Fourier Transform*, **Communications of the ACM 10**(10), 647–654 (1967).

$$
\begin{aligned}
&= \sum_{n=0}^{N/2-1} x_{2n} e^{-2\pi i k(2n)/N} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-2\pi i k(2n+1)/N} \\
&= \sum_{n=0}^{N/2-1} x_{2n} e^{-2\pi i k n/(N/2)} + e^{-2\pi i k/N} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-2\pi i k n/(N/2)} \\
&= X_k^e + e^{-2\pi i k/N} X_k^o \qquad (2.12)
\end{aligned}
$$

where X_k^e is the k-th component of the Fourier transform of length $N/2$ formed from the even components, and X_k^o is the corresponding transform of length $N/2$ formed from the odd components.

The significant thing about this derivation is that recursive computation of the FFT requires only $N log_2 N$ complex multiplications.

Perhaps the shortest way to code the FFT algorithm is to have a recursively called FFT function following the above FFT derivation. However, an equivalent non-recursive algorithm is usually preferred for reasons of efficiency.

Some of the complex multiplications in the FFT can be simplified or eliminated. For example, if $n = 0$ the exponential term becomes unity. An efficient algorithm takes advantage of these types of simplifications wherever possible. Also, the sine and cosine values that arise from the exponential term can be precomputed so as to eliminate the need for their time consuming calculation. *FFTPAK 87/32 Lite* uses these and many other techniques for making the FFT calculation as efficient as possible.

Chapter 3

Using *FFTPAK 87/32 Lite*

3.1 A Programming Example

Suppose we have a complex vector of length $N = 16$, how do we calculate the FFT? The following simple program shows how to do this:

```
/* example01.c */
#include <stdio.h>
#include "fftpak.h"

static struct LONGCOMPLEX x[16];
static int i, m, N;

void main(){
  m = 4;                     /* FFT length = 2^m = 16 */
  N = 1 << m;                /* N = 1 << m = 16 */
  /* define a data vector */
  for (i=0; i<N; i++) {
    x[i].r = 1.0/(1.0 + i);  /* define real component */
    x[i].i = 0.0;            /* define imaginary component */
    };
  fft16( x );                /* compute the FFT */
  fftsort( x, m );           /* sort results into natural order */
  /* print the results */
  printf("16-point FFT Results\n");
  for (i=0; i<N; i++)
    printf("x[%2d] = (%e,%e)\n",i,x[i].r,x[i].i);
  /* recover the original function */
  fftinv16( x );
  scalecv( x, m, n );
```

```
    fftsort( x, m );
    printf("Recovered 16-point vector\n");
    for (i=0; i<N; i++)
      printf("x[%2d] = (%e,%e)\n",i,x[i].r,x[i].i);
    };
```

The output from this example program is given below:

```
16-point FFT Results
x[ 0] = (3.380729e+00,0.000000e+00)
x[ 1] = (1.345578e+00,-7.673311e-01)
x[ 2] = (9.869736e-01,-5.737437e-01)
x[ 3] = (8.345081e-01,-4.292854e-01)
x[ 4] = (7.542680e-01,-3.172619e-01)
x[ 5] = (7.081796e-01,-2.248500e-01)
x[ 6] = (6.814025e-01,-1.443064e-01)
x[ 7] = (6.672900e-01,-7.058799e-02)
x[ 8] = (6.628719e-01,0.000000e+00)
x[ 9] = (6.672900e-01,7.058799e-02)
x[10] = (6.814025e-01,1.443064e-01)
x[11] = (7.081796e-01,2.248500e-01)
x[12] = (7.542680e-01,3.172619e-01)
x[13] = (8.345081e-01,4.292854e-01)
x[14] = (9.869736e-01,5.737437e-01)
x[15] = (1.345578e+00,7.673311e-01)
Recovered 16-point vector
x[ 0] = (1.000000e+00,1.387779e-17)
x[ 1] = (5.000000e-01,-1.734723e-17)
x[ 2] = (3.333333e-01,1.040834e-17)
x[ 3] = (2.500000e-01,-1.040834e-17)
x[ 4] = (2.000000e-01,0.000000e+00)
x[ 5] = (1.666667e-01,1.387779e-17)
x[ 6] = (1.428571e-01,-1.387779e-17)
x[ 7] = (1.250000e-01,1.387779e-17)
x[ 8] = (1.111111e-01,-1.387779e-17)
x[ 9] = (1.000000e-01,1.734723e-17)
x[10] = (9.090909e-02,-1.040834e-17)
x[11] = (8.333333e-02,1.040834e-17)
x[12] = (7.692308e-02,0.000000e+00)
x[13] = (7.142857e-02,-1.387779e-17)
x[14] = (6.666667e-02,1.387779e-17)
x[15] = (6.250000e-02,-1.387779e-17)
```

The `LONGCOMPLEX` structure referred to in this example is defined in the header file `fftpak.h` as

```
struct LONGCOMPLEX {
  double r, i;
  };
```

In this example, the FFT is computed using the function `fft16` and then sorted using `fftsort`. The *FFTPAK 87/32 Lite* library contains specially coded functions for FFTs of length 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024. These functions are named `fft2`, `fft4`, ..., `fft1024`. For FFTs of any length there is also a general purpose function `fft` that could have been used. In this case, we would replace the function call `fft16(x)` with `fft(x,m)`. The function `fft` calls `fft16` in any case, so by calling `fft16` directly, a small amount of overhead is avoided. In general, *FFTPAK 87/32* gives you to maximum control at every step of FFT calculation. That is why `fftsort` must be explicitly called. When the FFT calculation is finished, the results are in bit reversed order. In most cases you will want to sort then into natural (ascending) order. However, if a circular convolution is being performed, it is possible to eliminate a sorting step.

In the last part of the example, the original vector is recovered by scaling the transformed complex vector by the factor $2^4 = 1/16$ using the `scalecv` function, applying the `fftinv16` FFT inverse function, and once again sorting the results into natural order using the `fftsort` function. Note that the recovered vector has non-zero imaginary components due to roundoff error in the calculation. These roundoff errors are negligible for all practical purposes, but they can be avoided altogether by using the real valued FFT functions in the full version of *FFTPAK 87/32*. For large FFTs ($N > 512$), the real valued FFT functions also have the advantage of executing in approximately half the time of the corresponding complex FFT functions.

The `scalecv` function that is used to scale the complex vector by the factor 1/16 is very fast and involves no roundoff error, since it only adjusts the binary exponent of the floating point numbers. It doesn't even use the math coprocessor, since only a portion of each floating point number must be adjusted.

A detailed description of every *FFTPAK 87/32 Lite* function is given in Chapter 4.

You can compile this program just as you would any other C program. The easiest way to link to the *FFTPAK 87/32 Lite* functions to add the `FFTPAK87.LIB` file to the list of libraries to which you are linking.

3.2 *FFTPAK 87/32* with Borland C++ for OS/2

Borland C++ for OS/2 provides an easy to use integrated development environment (IDE). The simplest way to use *FFTPAK 87/32* is to create a project that includes your source files, as well as the file `FFTPAK87.LIB`. Also make sure that the header file `fftpak.h` is in your `#include` path. Borland C++ will compile and link to the `FFTPAK87.LIB` file just as for any other library. Also, be sure that the line

```
#define pascal _Pascal
```

at the top of the `fftpak.h` header file is commented out.

3.3 *FFTPAK 87/32* with IBM CSet++

If you are using the IBM CSet++ compiler, you need to uncomment out the statement

```
#define pascal _Pascal
```

at the top of the `fftpak.h` header file. The IBM's CSet++ compiler can then be used to compile and link the program `exampl01.c` using the command:

```
icc exampl01.c FFTPAK87.LIB
```

3.4 Overview of Functions in Shareware Version

The following double precision functions are provided in the shareware version of *FFTPAK 87/32 Lite*:

`fft2` — complex FFT of length 2.

`fft4` — complex FFT of length 4.

`fft8` — complex FFT of length 8.

`fft16` — complex FFT of length 16.

`fft32` — complex FFT of length 32.

`fft64` — complex FFT of length 64.

`fft128` — complex FFT of length 128.

`fft256` — complex FFT of length 256.

`fft512` — complex FFT of length 512.

`fft1024` — complex FFT of length 1024.

`fft` — complex FFT of length 2^m.

`fftinv2` — complex inverse FFT of length 2.

`fftinv4` — complex inverse FFT of length 4.

`fftinv8` — complex inverse FFT of length 8.

`fftinv16` — complex inverse FFT of length 16.

`fftinv32` — complex inverse FFT of length 32.

`fftinv64` — complex inverse FFT of length 64.

`fftinv128` — complex inverse FFT of length 128.

`fftinv256` — complex inverse FFT of length 256.

`fftinv512` — complex inverse FFT of length 512.

`fftinv1024` — complex inverse FFT of length 1024.

`fftinv` — complex FFT of length 2^m.

`fftsort` — bit reverse sort for complex FFT of length 2^m.

`scalecv` — scale complex vector of length N by factor 2^{-m}.

`scalev` — scale real vector of length N by factor 2^{-m}.

`conjcv` — complex conjugate of complex vector of length N.

3.5 Additional Functions in Registered Version

The registered version of *FFTPAK 87/32 Lite* contains the following additional functions:

`fft2_s` — single precision complex FFT of length 2.

`fft4_s` — single precision complex FFT of length 4.

`fft8_s` — single precision complex FFT of length 8.

`fft16_s` — single precision complex FFT of length 16.

`fft32_s` — single precision complex FFT of length 32.

`fft64_s` — single precision complex FFT of length 64.

`fft128_s` — single precision complex FFT of length 128.

`fft256_s` — single precision complex FFT of length 256.

`fft512_s` — single precision complex FFT of length 512.

`fft1024_s` — single precision complex FFT of length 1024.

`fft_s` — single precision complex FFT of length 2^{m}.

`fftinv2_s` — single precision complex inverse FFT of length 2.

`fftinv4_s` — single precision complex inverse FFT of length 4.

`fftinv8_s` — single precision complex inverse FFT of length 8.

`fftinv16_s` — single precision complex inverse FFT of length 16.

`fftinv32_s` — single precision complex inverse FFT of length 32.

`fftinv64_s` — single precision complex inverse FFT of length 64.

`fftinv128_s` — single precision complex inverse FFT of length 128.

`fftinv256_s` — single precision complex inverse FFT of length 256.

`fftinv512_s` — single precision complex inverse FFT of length 512.

`fftinv1024_s` — single precision complex inverse FFT of length 1024.

`fftinv_s` — single precision complex FFT of length 2^{m}.

`fftsort_s` — single precision bit reverse sort for FFT of length 2^{m}.

`scalecv_s` — scale single precision complex vector of length N by factor $2^{-\mathrm{m}}$.

`scalev_s` — scale single precision real vector of length N by factor $2^{-\mathrm{m}}$.

`conjcv_s` — complex conjugate of single precidion complex vector of length N.

Chapter 4

Function Descriptions

4.1 Definitions and Function Prototypes

The FFTPAK 87/32 header file `FFTPAK.h` contains definitions and prototypes for all *FFTPAK 87/32 Lite* functions. This file is listed below for reference:

```
/*********************************************/
/* fftpak.h                                  */
/* Copyright (C) 1995 Precision Plus Software */
/* All rights reserved.                      */
/*********************************************/

#define twopi 6.283185307179586

/* uncomment the line below for IBM CSet++
#define pascal _Pascal
*/

struct LONGCOMPLEX {
  double r;
  double i;
  };

struct COMPLEX {
  float r;
  float i;
  };

void pascal fft2( struct LONGCOMPLEX *x );
```

```
void pascal fft4( struct LONGCOMPLEX *x );
void pascal fft8( struct LONGCOMPLEX *x );
void pascal fft16( struct LONGCOMPLEX *x );
void pascal fft32( struct LONGCOMPLEX *x );
void pascal fft64( struct LONGCOMPLEX *x );
void pascal fft128( struct LONGCOMPLEX *x );
void pascal fft256( struct LONGCOMPLEX *x );
void pascal fft512( struct LONGCOMPLEX *x );
void pascal fft1024( struct LONGCOMPLEX *x );
void pascal fft( struct LONGCOMPLEX *x, int m );

void pascal fftinv2( struct LONGCOMPLEX *x );
void pascal fftinv4( struct LONGCOMPLEX *x );
void pascal fftinv8( struct LONGCOMPLEX *x );
void pascal fftinv16( struct LONGCOMPLEX *x );
void pascal fftinv32( struct LONGCOMPLEX *x );
void pascal fftinv64( struct LONGCOMPLEX *x );
void pascal fftinv128( struct LONGCOMPLEX *x );
void pascal fftinv256( struct LONGCOMPLEX *x );
void pascal fftinv512( struct LONGCOMPLEX *x );
void pascal fftinv1024( struct LONGCOMPLEX *x );
void pascal fftinv( struct LONGCOMPLEX *x, int m );

void pascal fftsort( struct LONGCOMPLEX *x, int m );
void pascal scalecv( struct LONGCOMPLEX *x, int m, int N );
void pascal scalev( double *x, int m, int N );
void pascal conjcv( struct LONGCOMPLEX *x, int N );

void pascal fft2_s( struct COMPLEX *x );
void pascal fft4_s( struct COMPLEX *x );
void pascal fft8_s( struct COMPLEX *x );
void pascal fft16_s( struct COMPLEX *x );
void pascal fft32_s( struct COMPLEX *x );
void pascal fft64_s( struct COMPLEX *x );
void pascal fft128_s( struct COMPLEX *x );
void pascal fft256_s( struct COMPLEX *x );
void pascal fft512_s( struct COMPLEX *x );
void pascal fft1024_s( struct COMPLEX *x );
void pascal fft_s( struct COMPLEX *x, int m );

void pascal fftinv2_s( struct COMPLEX *x );
void pascal fftinv4_s( struct COMPLEX *x );
void pascal fftinv8_s( struct COMPLEX *x );
void pascal fftinv16_s( struct COMPLEX *x );
```

```
void pascal fftinv32_s( struct COMPLEX *x );
void pascal fftinv64_s( struct COMPLEX *x );
void pascal fftinv128_s( struct COMPLEX *x );
void pascal fftinv256_s( struct COMPLEX *x );
void pascal fftinv512_s( struct COMPLEX *x );
void pascal fftinv1024_s( struct COMPLEX *x );
void pascal fftinv_s( struct COMPLEX *x, int m );

void pascal fftsort_s( struct COMPLEX *x, int m );
void pascal scalecv_s( struct COMPLEX *x, int m, int N );
void pascal scalev_s( float *x, int m, int N );
void pascal conjcv_s( struct COMPLEX *x, int N );
```

4.2 Fixed Length Complex Forward Transforms

This section describes a special set of forward complex FFTs for lengths 2 to 1024. These very fast individually coded routines use precomputed sine and cosine values.

4.2.1 `fft2` — Length 2 Forward Transform

The function prototypes for the double precision function `fft2` and the corresponding single precision function `fft2_s` are:

```
void pascal fft2( struct LONGCOMPLEX *x );
void pascal fft2_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 2 in place.

4.2.2 `fft4` — Length 4 Forward Transform

The function prototypes for the double precision function `fft4` and the corresponding single precision function `fft4_s` are:

```
void pascal fft4( struct LONGCOMPLEX *x );
void pascal fft4_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 4 in place. The results are in bit-reversed order.

4.2.3 `fft8` — Length 8 Forward Transform

The function prototypes for the double precision function `fft8` and the corresponding single precision function `fft8_s` are:

```
void pascal fft8( struct LONGCOMPLEX *x );
void pascal fft8_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 8 in place. The results are in bit-reversed order.

4.2.4 `fft16` — Length 16 Forward Transform

The function prototypes for the double precision function `fft16` and the corresponding single precision function `fft16_s` are:

```
void pascal fft16( struct LONGCOMPLEX *x );
void pascal fft16_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 16 in place. The results are in bit-reversed order.

4.2.5 `fft32` — Length 32 Forward Transform

The function prototypes for the double precision function `fft32` and the corresponding single precision function `fft32_s` are:

```
void pascal fft32( struct LONGCOMPLEX *x );
void pascal fft32_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 32 in place. The results are in bit-reversed order.

4.2.6 `fft64` — Length 64 Forward Transform

The function prototypes for the double precision function `fft64` and the corresponding single precision function `fft64_s` are:

```
void pascal fft64( struct LONGCOMPLEX *x );
void pascal fft64_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 64 in place. The results are in bit-reversed order.

4.2.7 `fft128` — Length 128 Forward Transform

The function prototypes for the double precision function `fft128` and the corresponding single precision function `fft128_s` are:

```
void pascal fft128( struct LONGCOMPLEX *x );
void pascal fft128_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 128 in place. The results are in bit-reversed order.

4.2.8 `fft256` — Length 256 Forward Transform

The function prototypes for the double precision function `fft256` and the corresponding single precision function `fft256_s` are:

```
void pascal fft256( struct LONGCOMPLEX *x );
void pascal fft256_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 256 in place. The results are in bit-reversed order.

4.2.9 `fft512` — Length 512 Forward Transform

The function prototypes for the double precision function `fft512` and the corresponding single precision function `fft512_s` are:

```
void pascal fft512( struct LONGCOMPLEX *x );
void pascal fft512_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 512 in place. The results are in bit-reversed order.

4.2.10 `fft1024` — Length 1024 Forward Transform

The function prototypes for the double precision function `fft1024` and the corresponding single precision function `fft1024_s` are:

```
void pascal fft1024( struct LONGCOMPLEX *x );
void pascal fft1024_s( struct COMPLEX *x );
```

This function calculates the FFT of the complex vector `x` of length 1024 in place. The results are in bit-reversed order.

4.3 Fixed Length Complex Inverse Transforms

This section describes a special set of inverse complex FFTs for lengths 2 to 1024. These very fast individually coded routines use precomputed sine and cosine values.

4.3.1 `fftinv2` — Length 2 Inverse Transform

The function prototypes for the double precision function `fftinv2` and the corresponding single precision function `fftinv2_s` are:

```
void pascal fftinv2( struct LONGCOMPLEX *x );
void pascal fftinv2_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 2 in place. The results are in bit-reversed order.

4.3.2 `fftinv4` — Length 4 Inverse Transform

The function prototypes for the double precision function `fftinv4` and the corresponding single precision function `fftinv4_s` are:

```
void pascal fftinv4( struct LONGCOMPLEX *x );
void pascal fftinv4_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 4 in place. The results are in bit-reversed order.

4.3.3 `fftinv8` — Length 8 Inverse Transform

The function prototypes for the double precision function `fftinv8` and the corresponding single precision function `fftinv8_s` are:

```
void pascal fftinv8( struct LONGCOMPLEX *x );
void pascal fftinv8_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 8 in place. The results are in bit-reversed order.

4.3.4 `fftinv16` — Length 16 Inverse Transform

The function prototypes for the double precision function `fftinv16` and the corresponding single precision function `fftinv16_s` are:

```
void pascal fftinv16( struct LONGCOMPLEX *x );
void pascal fftinv16_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 16 in place. The results are in bit-reversed order.

4.3.5 `fftinv32` — Length 32 Inverse Transform

The function prototypes for the double precision function `fftinv32` and the corresponding single precision function `fftinv32_s` are:

```
void pascal fftinv32( struct LONGCOMPLEX *x );
void pascal fftinv32_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 32 in place. The results are in bit-reversed order.

4.3.6 `fftinv64` — Length 64 Inverse Transform

The function prototypes for the double precision function `fftinv64` and the corresponding single precision function `fftinv64_s` are:

```
void pascal fftinv64( struct LONGCOMPLEX *x );
void pascal fftinv64_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 64 in place. The results are in bit-reversed order.

4.3.7 `fftinv128` — Length 128 Inverse Transform

The function prototypes for the double precision function `fftinv128` and the corresponding single precision function `fftinv128_s` are:

```
void pascal fftinv128( struct LONGCOMPLEX *x );
void pascal fftinv128_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 128 in place. The results are in bit-reversed order.

4.3.8 `fftinv256` — Length 256 Inverse Transform

The function prototypes for the double precision function `fftinv256` and the corresponding single precision function `fftinv256_s` are:

```
void pascal fftinv256( struct LONGCOMPLEX *x );
void pascal fftinv256_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 256 in place. The results are in bit-reversed order.

4.3.9 `fftinv512` — Length 512 Inverse Transform

The function prototypes for the double precision function `fftinv512` and the corresponding single precision function `fftinv512_s` are:

```
void pascal fftinv512( struct LONGCOMPLEX *x );
void pascal fftinv512_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 512 in place. The results are in bit-reversed order.

4.3.10 `fftinv1024` — Length 1024 Inverse Transform

The function prototypes for the double precision function `fftinv1024` and the corresponding single precision function `fftinv1024_s` are:

```
void pascal fftinv1024( struct LONGCOMPLEX *x );
void pascal fftinv1024_s( struct COMPLEX *x );
```

This function calculates the inverse FFT of the complex vector `x` of length 1024 in place. The results are in bit-reversed order.

4.4 Variable Length Complex Transforms

The variable length FFTs described in this section call very fast fixed length FFTs described in previous sections for lengths 2 to 1024. For larger transforms, computation of additional sine and cosine values is required.

4.4.1 `fft` Variable Length Forward Transform

The function prototypes for the double precision function `fft` and the corresponding single precision function `fft_s` are:

```
void pascal fft( struct LONGCOMPLEX *x, int m );
void pascal fft_s( struct COMPLEX *x, int m );
```

This function calculates the FFT of the complex vector `x` of length $2^{\mathtt{m}}$ in place. The results are in bit-reversed order.

4.4.2 `fftinv` — Variable Length Inverse Transform

The function prototypes for the double precision function `fftinv` and the corresponding single precision function `fftinv_s` are:

```
void pascal fftinv( struct LONGCOMPLEX *x, int m );
void pascal fftinv_s( struct COMPLEX *x, int m );
```

This function calculates the inverse FFT of the complex vector `x` of length $2^{\mathtt{m}}$ in place. The results are in bit-reversed order.

4.5 Utility Functions

4.5.1 `fftsort` — Complex FFT Bit-Reversed Sort

The function prototypes for the double precision function `fftsort` and the corresponding single precision function `fftsort_s` are:

```
void pascal fftsort( struct LONGCOMPLEX *x, int m );
void pascal fftsort_s( struct COMPLEX *x, int m );
```

This function sorts the complex vector `x` of length $2^{\mathtt{m}}$ in place. If the input vector is in bit-reversed order, then on output the results are in natural order.

4.5.2 `scalecv` — Scale Complex Vector

The function prototypes for the double precision function `scalecv` and the corresponding single precision function `scalecv_s` are:

```
void pascal scalecv( struct LONGCOMPLEX *x, int m, int N );
void pascal scalecv_s( struct COMPLEX *x, int m, int N );
```

This function scales complex vector x of length N by the factor 2^{-m}. This procedure is very fast since it works by adjusting the binary exponent of the floating point numbers, and involves no multiplies or divides.

4.5.3 scalev — Scale Real Vector

The function prototypes for the double precision function scalev and the corresponding single precision function scalev_s are:

```
void pascal scalev( double *x, int m, int N );
void pascal scalev_s( float *x, int m, int N );
```

This function scales real vector x of length N by the factor 2^{-m}. This procedure is very fast since it works by adjusting the binary exponent of the floating point numbers, and involves no multiplies or divides.

4.5.4 conjcv — Conjugate Complex Vector

The function prototypes for the double precision function conjcv and the corresponding single precision function conjcv_s are:

```
void pascal conjcv( struct LONGCOMPLEX *x, int N );
void pascal conjcv_s( struct COMPLEX *x, int N );
```

This function conjugates complex vector x of length N. This procedure is very fast since it works by flipping the sign bit of the imaginary component numbers.

Chapter 5

Shareware

5.1 Shareware Concept and Registration

The *FFTPAK 87/32 Lite* software package is distributed as Shareware.

If you try this software and continue to use it, you are expected to register it with Precision Plus Software. You are granted a 30 day evaluation period before you are required to either register the product or discontinue using it.

This software can be freely distributed, as long as no money is charged for it, all the files are included, unmodified, and with their modification dates preserved.

The library files that contain the *FFTPAK 87/32 Lite* object code cannot be distributed as a part of another product. Registered users who have paid the registration fee for this software package are granted a license to compile and link to *FFTPAK 87/32 Lite* functions and distribute the resulting executable file(s) without further license fees. Upon registration, you will receive the latest version of the product, a complete set of even faster single precision FFT functions, and a printed manual.

This software cannot be used in a commercial environment without the payment of the proper license fee.

Our success will depend not only on the quality of this software but on the willingness of every individual user to "support" its developers. If you use this product, please send in the registration form in the back of this document, along with your registration fee. You may also register this product through CompuServe and other online services. Please refer to the `READ.ME` file for details. Place orders for related software products described in Appendix A directly with Precision Plus Software.

Whether or not you use this product, please give complete copies of this software to others.

5.2 License Terms

Before you register this product and become a licensed user, you are granted a limited 30 day license to evaluate the product to determine whether or not it will fit your needs. Use of this software for any other purpose, before registration and without our written consent, is expressly forbidden.

Registered users are given a non-exclusive license to use this software on any machine that they have access to, but not on more than one at a time (the "treat this software like a book" idea).

Registered users may include portions of the library and runtime code in the programs developed by them, and use or distribute these programs without payment of any additional license fees to Precision Plus Software as long as the *FFTPAK 87/32 Lite* code is embedded in the users executable program. The *FFTPAK 87/32 Lite* library files may not be distributed with user programs. (*FFTPAK 87/32 Lite* dynamic link libraries may be licensed for distribution with user program under a separate license from Precision Plus Software.)

5.3 Support

Support for this product is available to registered users by telephone (519) 657-0633, fax (519) 657-0283, E-Mail 70401.2606@compuserve.com, or mail: Precision Plus Software, 38 Longview CRT, London, ON, Canada N6K 4J1.

User comments are always welcome. Please feel free to send us your suggestions by fax, E-Mail or mail. We will try to accomodate user requests for new features in future releases of this software.

If you are reporting a bug, please specify:

- version of *FFTPAK 87/32 Lite* software in use;
- operating system and compiler in use;
- problem description, with a small sample program to illustrate the problem.

Appendix A

Related Software Products

Precision Plus Software publishes a range of mathematical software libraries for scientists and engineers. Precision Plus Software's MATHPAK 87 product has been available since 1985 and is used by more than 10,000 scientists and engineers around the world. Current uses for the software include radar signal processing, speech recognition, image analysis, linear and nonlinear systems analysis, and even econometric models. The software is also used in a variety of embedded systems applications.

One thing that all these products have in common is ease of use, exceptional performance, and modest price. The philosophy used to write the software has been to choose the best available algorithms, implement time-critical portions in assembler, and provide simple interfaces to support all major compilers in the DOS, Windows and OS/2 environments.

The sections below describe the range of software available today. For ordering information, please refer to the order form in the back of this manual.

A.1 *FFTPAK 87/32*

FFTPAK 87/32 is a set of 32-bit assembler coded FFT functions, as well as related scaling and unscrambling functions. The FFT functions in this library are faster than those of any other available FFT library. Unlike earlier 16-bit functions, they are not limited to small FFT's (64 KB) but can handle FFT's of practically unlimited size.

FFTPAK 87/32 contains single and double precision functions for:

- complex FFTs;

- real valued data FFTs;
- real valued data and complex two-dimensional FFTs;
- fast cosine transforms;
- fast sine transforms;
- real and complex correlation;
- real and complex autocorrelation;
- real and complex circular convolution;
- various one and two dimensional window functions.

A.2 *FFTPAK INT/32*

FFTPAK INT/32 is a special 32-bit integer FFT transform package for processing integer data, such as normally acquired by a real-time data acquisition system. This package includes 32-bit integer FFTs for real and complex data. The integer functions do not require a math coprocessor, and have accuracy comparable to single precision floating point FFTs. However, they run faster than floating point FFTs, particularly on the Pentium processor which has two parallel integer execution units. *FFTPAK INT/32* contains 32-bit integer functions for:

- complex FFTs;
- real valued data FFTs;
- real valued data and complex two-dimensional FFTs;
- fast cosine transforms;
- fast sine transforms;
- real and complex correlation;
- real and complex autocorrelation;
- real and complex circular convolution;
- various one and two dimensional window functions.

A.3 *FFTPAK 87/16*

FFTPAK 87/16 is a set of 16-bit assembler coded FFT functions, as well as related scaling and unscrambling functions. The FFT functions in this library are faster than those of any other available 16-bit FFT library. The 16-bit functions are limited to small FFT's (64 KB).

FFTPAK 87/16 contains single and double precision functions for:

- complex FFTs;
- real valued data FFTs;
- real valued data and complex two-dimensional FFTs;
- fast cosine transforms;
- fast sine transforms;
- real and complex correlation;
- real and complex autocorrelation;
- real and complex circular convolution;
- various one and two dimensional window functions.

A.4 *MATHPAK 87/32*

MATHPAK 87/32 is a set of highly optimized 32-bit assembler coded routines to perform some of the most common mathematical operations used in numerically intensive programs. This is a new product which is upwardly compatible with the 16-bit *MATHPAK 87* product described in the next section.

With *MATHPAK 87/32* you can solve a system of linear equations, invert a matrix, or calculate a fast Fourier transform (FFT) all in one line of your program by simply calling the appropriate *MATHPAK 87/32* routine. You can also perform many simple but time consuming operations that are commonly found in program, for example: summing an array of numbers, calculating mean and standard deviation of a set of data, linear regression, vector and matrix operations and much more — all at the limit of your computer's speed. Using *MATHPAK 87/32*, your programs become *shorter*, are *finished sooner* and run *faster*.

MATHPAK 87/32 routines are written specifically to fully utilize the Intel 80387 and later floating point coprocessors. MATHPAK 87/32's speed makes practical many programs that were previously beyond the reach of your computer and allows you to streamline many of your existing programs to make them run faster and with improved accuracy. Another benefit of using *MATHPAK 87/32* is that it increases your productivity by freeing you to think about how to solve your problem, rather than how to code complex but standard routines for mathematical operations such as matrix inversion, matrix eigenvalue and eigenvector computation, least squares data fitting, et cetera. MATHPAK 87/32 routines are powerful but easy to use. A short tutorial will get you started in minutes.

MATHPAK 87/32 routines typically run from 3 to 10 times faster than equivalent routines written in a high-level language such as C or Pascal. Because the numerically intensive parts are written in assembler, floating point calculations proceed as fast as your hardware allows.

A.4.1 Supported Compilers

MATHPAK 87/32 supports most common 32-bit OS/2, DOS and Windows compilers C/C++. A Pascal version for Speed Pascal for OS/2 is also being developed. Currently supported compilers include Borland C++, Microsoft Visual C++, IBM C-Set++, and Watcom C++.

A.4.2 *MATHPAK 87/32* Library Contents

The *MATHPAK 87/32* library contains the following functions:

- 5 80x87 math coprocessor control functions to determine coprocessor type, save and restore 80x87 contents, store and load 80x87 control word;
- 6 missing functions are provided for Pascal users: `log10`, `sinh`, `cosh`, `tanh`, `XtoY`, `tan`.
- 44 vector and vector-scalar routines. These include routines to fill a vector with a scalar, copy vectors, swap vectors, sum vector, sum square of vector elements, dot product, cross product, find vector minimum or maximum, convert integer vector to floating point vector, etc.
- 28 vector and vector-scalar "skip" routines. These routines allow operation on non-consecutive elements. For example, these routines allow

you to sum every third element of a vector, etc.

- 24 complex number routines to efficiently manipulate complex numbers. These include functions to multiply complex numbers and calculate the square root of a complex number.

- 27 complex vector and vector-scalar routines to manipulate arrays of complex numbers;

- 7 polynomial manipulation routines. These routines include polynomial and polynomial derivative calculation, polynomial multiplication and division, and polynomial root finding.

- 11 simple matrix and vector-matrix routines. These include routines to fill a matrix, copy a matrix, multiply a matrix by a scalar, multiply a matrix times a vector, multiply a matrix times a matrix, add and subtract matrices, calculate a matrix transpose, calculate a matrix plus scalar times matrix, etc.

- 10 routines for solving systems of linear equations. These include routines for LU decomposition and backsolving of real and complex matrices, Gauss-Jordan matrix inversion, Gauss-Seidel solution of linear equations, LU decomposition and backsolving of tridiagonal systems, etc.

- 3 routines for solving systems of nonlinear equations. These include Newton-Raphson and Broyden nonlinear equation solvers, and a routine to calculate a Jacobian matrix by finite differences.

- 2 routines for multidimensional nonlinear minimization. These include a conjugate gradient method and a BFGS method.

- A Levenburg-Marquardt nonlinear parameter fitting routine.

- 8 eigenvalue and eigenvector routines. These include routines for calculating the eigenvalues and eigenvectors of a general real matrix, and specialized routines for symmetric matrices.

- A singular value decomposition (SVD) routine;

- A minimal polynomial extrapolation (MPE) routine for extrapolation of vector sequences.

- Mean, standard deviation, and linear regression routines.
- 8 FFT routines for complex vectors, real vectors, convolution, and two-dimensional FFT.
- 6 spectral analysis routines including Hamming Window, Parzen window, and Cosine window.
- Romberg adaptive numerical integration routine.
- Hamming predictor-corrector and fourth-order Runge-Kutta numerical integration routines.

MATHPAK 87/32 also contains example programs for stress analysis, network analysis, polynomial least squares curve fitting, and multicomponent distillation.

A.5 *MATHPAK 87*

MATHPAK 87 is a set of highly optimized 16-bit assembler coded routines to perform some of the most common mathematical operations used in numerically intensive programs. With *MATHPAK 87* you can solve a system of linear equations, invert a matrix, or calculate a fast Fourier transform (FFT) all in one line of your program by simply calling the appropriate *MATHPAK 87* routine. You can also perform many simple but time consuming operations that are commonly found in program, for example: summing an array of numbers, calculating mean and standard deviation of a set of data, linear regression, vector and matrix operations and much more — all at the limit of your computer's speed. Using *MATHPAK 87*, your programs become *shorter*, are *finished sooner* and run *faster*.

MATHPAK 87 routines are written specifically to fully utilize the Intel 80x87 floating point coprocessors and to allow your microcomputer to perform floating point calculations at speeds rivalling those of much bigger machines. MATHPAK 87's speed makes practical many programs that were previously beyond the reach of your computer and allows you to streamline many of your existing programs to make them run faster and with improved accuracy. Another benefit of using *MATHPAK 87* is that it increases your productivity by freeing you to think about how to solve your problem, rather than how to code complex but standard routines for mathematical operations such as matrix inversion, matrix eigenvalue and eigenvector computation, least squares data fitting, et cetera. MATHPAK 87 routines are powerful but easy to use. A short tutorial will get you started in minutes.

MATHPAK 87 routines typically run from 3 to 10 times faster than equivalent routines written in a high-level language such as C, Pascal, Basic or Fortran. Because the numerically intensive parts are written in assembler, floating point calculations proceed as fast as your hardware allows.

A.5.1 Supported Compilers

MATHPAK 87 versions are available for most DOS and Windows compilers. Each version comes with a detailed and easy to read manual (approx. 180 pages) with lots of example programs. C programmers get a manual with C example programs and documentation, Pascal programmers get a Pascal manual, etc. Specific versions have been developed for each compiler. The following versions are presently supported:

- *MATHPAK 87* for Borland C++ and Turbo C++;
- *MATHPAK 87* for Microsoft C, Quick C, and Visual C++;
- *MATHPAK 87* for Microsoft Quickbasic;
- *MATHPAK 87* for Microsoft Fortran;
- *MATHPAK 87* for Turbo Pascal, Borland Pascal, and Delphi.

A.5.2 *MATHPAK 87* Library Contents

The *MATHPAK 87* Version 3.0 and later library contains the following functions:

- 5 80x87 math coprocessor control functions to determine coprocessor type, save and restore 80x87 contents, store and load 80x87 control word;
- 6 missing functions are provided for Pascal users: `log10`, `sinh`, `cosh`, `tanh`, `XtoY`, `tan`.
- 44 vector and vector-scalar routines. These include routines to fill a vector with a scalar, copy vectors, swap vectors, sum vector, sum square of vector elements, dot product, cross product, find vector minimum or maximum, convert integer vector to floating point vector, etc.

- 28 vector and vector-scalar "skip" routines. These routines allow operation on non-consecutive elements. For example, these routines allow you to sum every third element of a vector, etc.

- 24 complex number routines to efficiently manipulate complex numbers. These include functions to multiply complex numbers and calculate the square root of a complex number.

- 27 complex vector and vector-scalar routines to manipulate arrays of complex numbers;

- 7 polynomial manipulation routines. These routines include polynomial and polynomial derivative calculation, polynomial multiplication and division, and polynomial root finding.

- 11 simple matrix and vector-matrix routines. These include routines to fill a matrix, copy a matrix, multiply a matrix by a scalar, multiply a matrix times a vector, multiply a matrix times a matrix, add and subtract matrices, calculate a matrix transpose, calculate a matrix plus scalar times matrix, etc.

- 10 routines for solving systems of linear equations. These include routines for LU decomposition and backsolving of real and complex matrices, Gauss-Jordan matrix inversion, Gauss-Seidel solution of linear equations, LU decomposition and backsolving of tridiagonal systems, etc.

- 3 routines for solving systems of nonlinear equations. These include Newton-Raphson and Broyden nonlinear equation solvers, and a routine to calculate a Jacobian matrix by finite differences.

- 2 routines for multidimensional nonlinear minimization. These include a conjugate gradient method and a BFGS method.

- A Levenburg-Marquardt nonlinear parameter fitting routine.

- 8 eigenvalue and eigenvector routines. These include routines for calculating the eigenvalues and eigenvectors of a general real matrix, and specialized routines for symmetric matrices.

- A singular value decomposition (SVD) routine;

- A minimal polynomial extrapolation (MPE) routine for extrapolation of vector sequences.
- Mean, standard deviation, and linear regression routines.
- 8 FFT routines for complex vectors, real vectors, convolution, and two-dimensional FFT.
- 6 spectral analysis routines including Hamming Window, Parzen window, and Cosine window.
- Romberg adaptive numerical integration routine.
- Hamming predictor-corrector and fourth-order Runge-Kutta numerical integration routines.

MATHPAK 87 also contains example programs for stress analysis, network analysis, polynomial least squares curve fitting, and multicomponent distillation.

The MS QuickBasic version of *MATHPAK 87* does not contain functions that require a pointer to a function to be passed to another function (i.e., as used in optimization functions), since this feature is missing from the QuickBasic language.

Appendix B

Registration

Please complete and send in the registration form on the following page. You may also register this shareware product online using CompuServe and other online service providers. Please see the `READ.ME` file for details. Orders for one or more of the related software products described in Appendix A must be placed directly with Precision Plus Software.

Upon registration of this product, you will receive a complete set of even faster single precision FFT functions and a printed manual.

Registration and Product Order Form

Please mail this form to

Precision Plus Software
38 Longview CRT
London, ON
CANADA N6K 4J1

If paying by credit card, you may fax this form to (519) 657-0283.

Name:____

Company:_______________________________________

Address:_______________________________________

City, State, Zip:______________________________

Country:_______________________________________

Telephone:___________, Fax:___________, E-Mail:_____________

QTY			
___	x	FFTPAK 87 Lite registration @ $89.00	_________
___	x	FFTPAK 87/32 professional version @ $149.00	_________
___	x	FFTPAK INT/32 professional version @ $199.00	_________
___	x	MATHPAK 87/32 @ $129.00	_________
		Specify compiler:_____________________	
___	x	MATHPAK 87/32 with source code @ $198.00	_________
		Specify compiler:_____________________	
___	x	MATHPAK 87 @ $129.00	_________
		Specify compiler:_____________________	
___	x	MATHPAK 87 with source code @ $198.00	_________
		Specify compiler:_____________________	
___	X	Air mail shipping outside US and Canada (*)	_________

Total enclosed: _________

Note: MATHPAK 87/32 works with MS Visual C++, Borland C++ (DOS/Windows, OS/2), IBM C-Set++, Watcom C++, Speed Pascal for OS/2, etc. MATHPAK 87 (16-bit) is available for Borland C++, MS Visual ++, Borland/Turbo Pascal/Delphi, MS QuickBasic, and MS Fortran.

CREDIT CARD ORDER INFORMATION

[ ] MasterCard   [ ] American Express

Card Number: ______________________________, Expiry Date: _____

Signature:_____________________

(*) We pay air mail shipping in USA and Canada. Add $10 (US) for international air mail. Cheque or money order payable in US dollars please!