```
\ COPYRIGHT 1991 JULIAN V. NOBLE
TASK INTEGRAL
FIND CP@L 0= ?( FLOAD COMPLEX )
\ define data-type tokens if not already
FIND REAL*4 0= ?(((
        0 CONSTANT REAL*4
        1 CONSTANT REAL*8
        2 CONSTANT COMPLEX
        3 CONSTANT DCOMPLEX )))

FIND 1ARRAY 0= ?( FLOAD MATRIX.HSF )
\ function usage
: USE( [COMPILE] ' CFA ; IMMEDIATE

\ BEHEADing starts here
0 VAR N

: inc.N  N 1+  IS N ;
: dec.N  N 2-  IS N ;

0 VAR type

\ define "stack"
        20 LONG REAL*8 1ARRAY X{
        20 LONG REAL*4 1ARRAY E{
        20 LONG DCOMPLEX 1ARRAY F{
        20 LONG DCOMPLEX 1ARRAY I{

2 DCOMPLEX SCALARS old.I final.I
: )integral ( n - - )\ trapezoidal rule
        X{ OVER }      G@L
        X{ OVER 1- }   G@L
        F- F2/
        F{ OVER }      G@L
        F{ OVER 1- }   G@L
        type 2 AND
        IF      X+ FROT X*F
        ELSE    F+ F*    THEN
        I{ SWAP 1- } G!L ;
0 VAR f.name
: f(x) f.name EXECUTE ;

: INITIALIZE
        IS type \ store type
        type F{ !
        type I{ !\ set types for function
        type ' old.I !
        type ' final.I !     \ and integral(s)
        type 1 AND X{ !
                \ set type for indep. var.
        E{ 0 } G!L      \ store error
        X{ 1 } G!L      \ store B
        X{ 0 } G!L      \ store A
        IS f.name       \ ! cfa of f(x)
        X{ 0 } G@L f(x) F{ 0 } G!L
        X{ 1 } G@L f(x) F{ 1 } G!L
        1 IS N
        N )integral
        type 2 AND   IF F=0 THEN
        F=0 final.I G!L
        FINIT ;

: E/2 E{ N 1- } G@L F2/ E{ N 1- } G!L ;

: }move.down  ( adr n - -)
        } #BYTES >R          ( - - seg off)
        DDUP R@ +
                ( - - s.seg s.off d.seg d.off )
        R> CMOVEL ;

: MOVE.DOWN
        E{ N 1- }move.down
        X{ N    }move.down
        F{ N    }move.down ;

: new.X          ( 87: - - x' )
        X{ N } G@L X{ N 1- } G@L
        F+ F2/     FDUP X{ N } G!L ;

\ cont'd. ...
```

```
\ INTEGRAL cont'd
\ debugging code
: GF.   1 >   IF  FSWAP  E.  THEN E.  ;
: F@.   DUP>R   G@L   R>   GF.  ;
: .STACKS   CR  ." N"
        8 CTAB  ." X"
        19 CTAB  ." Re[F(X)]"
        31 CTAB  ." Im[F(X)]"
        45 CTAB  ." Re[I]"
        57 CTAB  ." Im[I]"
        71 CTAB  ." E"
        N 2+ 0 DO  CR  I .
                  3 CTAB  X{ I } F@.
                  16 CTAB  F{ I }  F@.
                  42 CTAB  I{ I }  F@.
                  65 CTAB  E{ I }  F@.
        LOOP
        CR  5 SPACES ." old.I =" old.I  F@.
        5 SPACES  ." final.I =" final.I  F@.  CR  ;

CASE: <DEBUG>   NEXT .STACKS  ;CASE
0 VAR (DEBUG)
: DEBUG-ON    1 IS (DEBUG)  5 #PLACES !  ;
: DEBUG-OFF   0 IS (DEBUG)  7 #PLACES !  ;
: DEBUG  (DEBUG)   <DEBUG> ;

: SUBDIVIDE
   N 19 >  ABORT" Too many subdivisions!"
   E/2   MOVE.DOWN
   I{ N 1- }  DROP  old.I  #BYTES  CMOVEL
     new.X   f(x)    F{ N } G!L
   N )integral   N 1 + )integral  ;

: CONVERGED?  ( 87: - - I[N] + I'[N-1]-I[N-1] : - - f)
        I{ N } G@L  I{ N 1- } G@L   old.I G@L
        type 2  AND
        IF       CP-  CP +   CPDUP  CPABS
        ELSE     F-    F +   FDUP    FABS
        THEN
        E{ N 1- } G@L   F2*   F<  ;

CASE:  g*f    CP*F   F*   ;CASE
4 S->F  3 S->F  F/    FCONSTANT  F=4/3

: INTERPOLATE   ( 87: I[N] + I'[N-1]-I[N-1] - - )
        F=4/3   type  2/  g*f
        old.I  G@L   final.I G@L
        type 2 AND
        IF       CP +   CP +
        ELSE     F +    F +     THEN
        final.I G!L  ;
\ BEHEADing ends here

: )INTEGRAL    ( 87: A B ERR - - I[A,B] )
        INITIALIZE
        BEGIN   N 0 >
        WHILE     SUBDIVIDE   DEBUG
            CONVERGED?   inc.N
            IF    INTERPOLATE    dec.N
            ELSE    type 2 AND IF   FDROP
            THEN          FDROP
            THEN
        REPEAT   final.I    G@L   ;
BEHEAD"  N  INTERPOLATE      \ optional
\ USE( F.name % A % B % E  type )INTEGRAL
```

The nonrecursive program obviously requires *much* more code than the recursive version. This is the chief disadvantage of a nonrecursive method[16].

16. The memory usage is about the same: the recursive method pushes limits, *etc*. onto the fstack.

§§5–4 Example of)INTEGRAL IN USE

The debugging code ("**DEBUG-ON**") lets us track the execution of the program by exhibiting the simulated stacks. Here is an example, $\int_1^2 dx\,\sqrt{x}$:

```
USE( FSQRT % 1. % 2. % 1.E-3 REAL*4 )INTEGRAL E
```

```
N X          F          I          E
0 1.0000E+00 1.0000E+00 5.5618E-01 5.0000E-04
1 1.5000E+00 1.2247E+00 6.5973E-01 5.0000E-04
2 2.0000E+00 1.4142E+00 1.4983E-01 1.2500E-04
old.I = 1.2071E+00   final.I = 0.0000E+00

0 1.0000E+00 1.0000E+00 5.5618E-01 5.0000E-04
1 1.5000E+00 1.2247E+00 3.1845E-01 2.5000E-04
2 1.7500E+00 1.3228E+00 3.4213E-01 2.5000E-04
3 2.0000E+00 1.4142E+00 1.7396E-01 1.2500E-04
old.I = 6.5973E-01   final.I = 0.0000E+00

0 1.0000E+00 1.0000E+00 5.5618E-01 5.0000E-04
1 1.5000E+00 1.2247E+00 3.1845E-01 2.5000E-04
2 1.7500E+00 1.3228E+00 1.6826E-01 1.2500E-04
3 1.8750E+00 1.3693E+00 1.7396E-01 1.2500E-04
4 2.0000E+00 1.4142E+00 0.0000E+00 0.0000E+00
old.I = 3.4213E-01   final.I = 0.0000E+00

0 1.0000E+00 1.0000E+00 5.5618E-01 5.0000E-04
1 1.5000E+00 1.2247E+00 1.5621E-01 1.2500E-04
2 1.6250E+00 1.2747E+00 1.6235E-01 1.2500E-04
3 1.7500E+00 1.3228E+00 1.7396E-01 1.2500E-04
old.I = 3.1845E-01   final.I = 3.4226E-01
```

```
N X          F          I          E
0 1.0000E+00 1.0000E+00 2.6475E-01 2.5000E-04
1 1.2500E+00 1.1180E+00 2.9284E-01 2.5000E-04
2 1.5000E+00 1.2247E+00 1.6235E-01 1.2500E-04
old.I = 5.5618E-01   final.I = 6.6087E-01

0 1.0000E+00 1.0000E+00 2.6475E-01 2.5000E-04
1 1.2500E+00 1.1180E+00 1.4316E-01 1.2500E-04
2 1.3750E+00 1.1726E+00 1.4983E-01 1.2500E-04
3 1.5000E+00 1.2247E+00 1.7396E-01 1.2500E-04
old.I = 2.9284E-01   final.I = 6.6087E-01

0 1.0000E+00 1.0000E+00 1.2879E-01 1.2500E-04
1 1.1250E+00 1.0606E+00 1.3616E-01 1.2500E-04
2 1.2500E+00 1.1180E+00 1.4983E-01 1.2500E-04
old.I = 2.6475E-01   final.I = 9.5392E-01
1.2189E+00 ok
```

$$\boxed{\int_1^2 dx\,\sqrt{x} = \frac{2}{3}\left(2^{3/2} - 1\right)}$$

Notice that, although $\sqrt{x}$ is perfectly finite at $x = 0$, its first derivative is not. This is not a problem in the above case, because the lower limit is 1.0.

It is an instructive exercise to run the above example with the limits (0.0, 1.0). The adaptive routine spends many iterations approaching $x = 0$ (25 in the range [0., 0.0625] *vs.* 25 in the range [0.0625, 1.0]). This is a concrete example of how an adaptive routine will unerringly locate the (integrable) singularities of a function by spending lots of time near them. The best answer to this problem is to separate out the bad parts of a function by hand, if possible, and integrate them by some other algorithm that takes the singularities into account. By the same token, one should always integrate *up to*, but not *through*, a discontinuity in $f(x)$.

§§6 Adaptive integration in the Argand plane

We often want to evaluate the complex integral

$$I = \oint_{\Gamma} f(z)\, dz \qquad (16)$$

where Γ is a **contour** (simple, closed, piecewise-continuous curve) in the complex z-plane, and $f(z)$ is an **analytic**[17] function of z.

The easiest way to evaluate 16 is to parameterize z as a function of a real variable t ; as t runs from A to B, $z(t)$ traces out the contour. For example, the parameterization

$$z(t) = z_0 + R\cos(t) + iR\sin(t)\,, \quad 0 \le t \le 2\pi \qquad (17)$$

traces out a (closed) circle of radius R centered at $z = z_0$.

We assume that the derivative $\dot{z}(t) \equiv \frac{dz}{dt}$ can be defined; then the integral 16 can be re-written as one over a real interval, with a complex integrand:

$$I = \int_A^B \dot{z}(t) f\big(z(t)\big)\, dt \qquad (18)$$

Now our previously defined adaptive function **)INTEGRAL** can be applied directly, with **F.name** the name of a *complex* function

$$g(t) = \dot{z}(t) f\big(z(t)\big), \qquad (19)$$

of the *real* variable t.

Here is an example of complex integration: we integrate the function $f(z) = e^{1/z}$ around the unit circle in the counter-clockwise (positive) direction.

---

17. "Analytic" means the ordinary derivative df(z)/dz exists. Consult any good text on the theory of functions of a complex variable.

The calculus of residues (Cauchy's theorem) gives

$$\oint_{|z|=1} dz\, e^{1/z} = 2\pi i \tag{20}$$

We parameterize the unit circle as $z(t) = \cos(2\pi t) + i \sin(2\pi t$ hence $\dot{z}(t) = 2\pi i z(t)$, and we might as well evaluate

$$\int_0^1 dt\, z(t)\, e^{1/z(t)} \equiv 1. \tag{21}$$

For reasons of space, we exhibit only the first and last few iteratio

```
FIND FSINCOS  0= ?( FLOAD TRIG )
: Z(T)   F=PI  F*   F2*   FSINCOS ;          ( 87: t - -
: XEXP  FSINCOS  FROT  FEXP  X*F ;
   ( 87: x y - - e^x cos[y]  e^x sin[y] )
: G(T)    Z(T)  XDUP  1/X  XEXP   X* ;
DEBUG-ON
USE( G(T) % 0 % 1 % 1.E-2 COMPLEX )INTEGRAL   X.
```

```
N   X        F                 I               E
0 0.0000  2.7182    0.0000    .58760    0.0000 .0049999
1 .50000  -.36787   -0.0000   .58760    0.0000 .0049999
2 1.0000  2.7182    0.0000    .00000    .00000 .0000000
old.I = 2.7182 0.0000    final.I = 0.0000 0.0000

N   X        F            I          E
0 0.0000  2.7182    0.0000    .58760    0.0000    .0049999
1 .50000  -.36787   -0.0000   .059198   -.067537 .0024999
2 .75000  .84147    -.54030   .44496    -.067537  .0024999
3 1.0000  2.7182    0.0000    .00000    .00000    .0000000
old.I = .58760 0.0000    final.I = 0.0000 0.0000

N   X        F            I          E
0 0.0000 2.7182 0.0000 .58760 0.0000  .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537 .0024999
2 .75000 .84147 -.54030 .17896 -.043682 .0012499
3 .87500 2.0219 -.15862 .29626 -.009914 .0012499
4 1.0000 2.7182 0.0000 .00000 .000000 .0000000
old.I = .44496 -.067537    final.I = 0.0000 0.0000

N   X        F            I          E
0 0.0000 2.7182 0.0000 .58760 0.0000  .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537 .0024999
2 .75000 .84147 -.54030 .17896 -.043682 .0012499
3 .87500 2.0219 -.15862 .14190 -.005745 .00062499
4 .93750 2.5189 -.025229 .16366 -.000788 .00062499
5 1.0000 2.7182 0.0000 .000000 .000000 .00000000
old.I = .29626 -.0099138    final.I = 0.0000 0.0000

N   X        F            I                    E
0 0.0000 2.7182 0.0000  .58760 0.0000           .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537       .0024999
2 .75000 .84147 -.54030 .17896 -.043682         .0012499
3 .87500 2.0219 -.15862 .14190 -.005745         .00062499
4 .93750 2.5189 -.025229 .081022 -.0004467      .00031249
5 .96875 2.6665 -.0033577 .08414 -.0000525      .00031249
6 1.0000 2.7182 0.0000  .000000 .0000000        .000000000
old.I = .16366 -.00078842    final.I = 0.0000 0.0000
N   X        F            I          E
0 0.0000 2.7182 0.0000 .58760 0.0000  .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537 .0024999
2 .75000 .84147 -.54030 .17896 -.043682 .0012499
3 .87500 2.0219 -.15862 .14190 -.0057453 .00062499
4 .93750 2.5189 -.025229 .08102 -.0004467 .00031249
5 .96875 2.6665 -.0033577 .04197 -.0000296 .00015624
6 .98437 2.7052 -.000426 .042371 -.0000033 .00015624
7 1.0000 2.7182 0.0000 0.0000 0.0000  0.0000
old.I = .084137 -.000052465    final.I = 0.0000 0.0000

N   X        F            I          E
0 0.0000 2.7182 0.0000 .58760 0.0000  .0049999
1 .50000 -.36787 -0.0000 .059198 -.067537 .0024999
2 .75000 .84147 -.54030 .17896 -.043682 .0012499
3 .87500 2.0219 -.15862 .14190 -.005745 .00062499
4 .93750 2.5189 -.025229 .040020 -.0002833 .00015624
5 .95312 2.6038 -.011038 .041173 -.0001125 .00015624
6 .96875 2.6665 -.003358 .042371 -.0000033 .00015624
old.I = .081022 -.00044667    final.I = .084404 -.0000263
```

```
N X       F       I       E
0 0.0000 2.7182 0.0000 .29626 .0099138 .0012499
1 .12500 2.0219 .15862 .10753 .016479 .00062499
2 .18750 1.4190 .36873 .039711 .013046 .00031249
3 .21875 1.1224 .46620 .030686 .015726 .00031249
4 .25000 .84147 .54030 .000053670 .0079676 .00015624
old.I = .070842 .028407   final.I = .51343 -.050786N

N X       F       I       E
0 0.0000 2.7182 0.0000 .29626 .0099138 .0012499
1 .12500 2.0219 .15862 .058518 .0065556 .00031249
2 .15625 1.7232 .26094 .049099 .0098386 .00031249
3 .18750 1.4190 .36873 .030686 .015726 .00031249
old.I = .10753 .016479   final.I = .58374 -.021892

N X       F       I       E
0 0.0000 2.7182 0.0000 .16366 .00078842 .00062499
1 .062500 2.5189 .025229 .14190 .0057453 .00062499
2 .12500 2.0219 .15862 .049099 .0098386 .00031249
old.I = .29626 .0099138   final.I = .69139 -.0055268

N X       F       I       E
0 0.0000 2.7182 0.0000 .16366 .00078842 .00062499
1 .062500 2.5189 .025229 .075223 .0015953 .00031249
2 .093749 2.2954 .076875 .067457 .0036796 .00031249
3 .12500 2.0219 .15862 .030686 .015726 .00031249
old.I = .14190 .0057453   final.I = .69139 -.0055268

N X       F       I       E
0 0.0000 2.7182 0.0000 .16366 .00078842 .00062499
1 .062500 2.5189 .025229 .075223 .0015953 .00031249
2 .093749 2.2954 .076875 .034833 .0014942 .00015624
3 .10937 2.1632 .11439 .032896 .0021329 .00015624
4 .12500 2.0219 .15862 .0000537 .0079676 .00015624
old.I = .067457 .0036796   final.I = .69139 -.0055268

N X       F       I       E
0 0.0000 2.7182 0.0000 .16366 .00078842 .00062499
1 .062500 2.5189 .025229 .038546 .00056464 .00015624
2 .078125 2.4150 .047044 .036800 .00096812 .00015624
3 .093749 2.2954 .076875 .032896 .0021329 .00015624
old.I = .075223 .0015953   final.I = .75894 -.0019171

N X       F       I       E
0 0.0000 2.7182 0.0000 .084137 .000052465 .00031249
1 .031250 2.6665 .0033577 .081022 .00044667 .00031249
2 .062500 2.5189 .025229 .036800 .00096812 .00015624
old.I = .16366 .00078842   final.I = .83433 -.00040523

N X       F       I       E
0 0.0000 2.7182 0.0000 .084137 .000052465 .00031249
1 .031250 2.6665 .0033577 .041173 .00011247 .00015624
2 .046875 2.6036 .011036 .040020 .00028334 .00015624
3 .062500 2.5189 .025229 .032896 .0021329 .00015624
old.I = .081022 .00044667   final.I = .83433 -.00040523

N X       F       I       E
0 0.0000 2.7182 0.0000 .042371 .000003331 .00015624
1 .015625 2.7052 .00042642 .041966 .0000296 .000 5624
2 .031250 2.6665 .0033577 .040020 .00028334 .00015624
old.I = .084137 .000052465   final.I = .91558 -.000026373
.99999 .00000001 ok
```

Note:
answer = 1

§2 Fitting functions to data

One of the most important applications of numerical analysis is the representation of numerical data in functional form. This includes fitting, smoothing, filtering, interpolating, *etc*.

A typical example is the problem of table lookup: a program requires values of some mathematical function $-\sin(x)$, say $-$ for arbitrary values of x. The function is moderately or extremely time-consuming to compute directly. According to the Intel timings for the 80x87 chip, this operation should take about 8 tim longer than a floating point multiply. In some real-time applica tions this may be too slow.

There are several ways to speed up the computation of a functio They are all based on compact representations of the functio —either in tabular form or as coefficients of functions that a faster to evaluate. For example, we might represent sin(x) by simple polynomial[18]

$$\sin(x) \approx x\left(0.994108 - 0.147202x\right), \qquad (22)$$

accurate to better than 1% over the range $-\frac{\pi}{2} \le x \le \frac{\pi}{2}$, th requires but 3 multiplications and an addition to evaluate. Th would be twice as fast as calculating sin(x) on the 80x87 chip[19].

To achieve substantially greater speed requires table look-up To locate data in an ordered table, we might employ binar search: that is, look at the x-value halfway down the table and se if the desired value is greater or less than that. On the averag $\log_2(N)$ comparisons are required, where N is the length of th table. For a table with 1% precision, we might need 128 entrie *i.e.* seven comparisons.

Binary search is unacceptably slow — is there a faster method In fact, assuming an ordered table of equally-spaced abscissae th fastest way to locate the desired x-value is **hashing**, a method fo *computing* the address rather than finding it using comparison Suppose, as before, we need 1% accuracy, *i.e.* a 128-point tabl with x in the range $[0,\pi/2]$. To look up a value, we multiply x b $256/\pi \cong 81.5$, truncate to an integer and quadruple it to get (4-byte) floating point address. These operations —includin fetch to the 87stack— take about 1.5-2 fp multiply times, henc the speedup is 4-fold.

The speedup factor does not seem like much, especially for a function such as sin(x) that is built into the fast co-processor. However, if we were speaking of a function that is considerably slower to evaluate (for example one requiring evaluation of an integral or solution of a differential equation) hashed table lookup with interpolation can be several orders of magnitude faster than direct evaluation.

We now consider how to represent data by mathematical functions. This can be useful in several contexts:

- The theoretical form of the function, but with unknown parameters, may be known. One might like to **determine** the parameters from the data. For example, one might have a lot of data on pendulums: their periods, masses, dimensions, *etc*. The period of a pendulum is given, theoretically, by

$$\tau = \left(\frac{2\pi L}{g}\right)^{1/2} f\left(\frac{L}{r}, \frac{m_{bob}}{m_{string}}, \ldots\right) \tag{23}$$

  where L is the length of the string, g the acceleration of gravity, and f is some function of ratios of typical lengths, masses and other factors in the problem. In order to determine g accurately, one generally fits a function of all the measured factors, and tries to minimize its deviation from the measured periods. That is, one might try

$$\tau_n = \left(\frac{2\pi L_n}{g}\right)^{1/2} \left[1 + \alpha\frac{r_n}{L_n} + \beta\left(\frac{m_{bob}}{m_{string}}\right)_n + \ldots\right] \tag{24}$$

  for the n'th set of observations, with g, α, β, ... the unknown parameters to be determined.

- Sometimes one knows that a phenomenon is basically smoothly varying; so that the wiggles and deviations in observations are noise or otherwise uninteresting. How can we filter out the noise without losing the significant part of the data? Several methods have been developed for this purpose, based on the same principle: the data are represented as a sum of functions from a **complete** set of functions, with unknown coefficients. That is, if $\varphi_m(x)$ are the functions, we say (y_n are the data)

$$y_n = \sum_{m=0}^{\infty} c_m \, \varphi_m(x_n) \tag{25}$$

Such representations are theoretically possible under general conditions. Then to filter we keep only a finite sum, retaining the first N (usually simplest and smoothest) functions from the set. An example of a complete set is **monomials,** $\varphi_m(x) = x^m$. Another is **sinusoidal (trigonometric) functions,**

$$\sin(2\pi mx), \ \cos(2\pi mx), \ 0 \le x \le 1,$$

used in Fourier-series representation. **Gram polynomials,** discussed below, comprise a third useful complete set.

The representation in Eq. 25 is called **linear** because the unknown coefficients c_m appear to their first power. Thus, if all the data were to double, we see immediately that the c_m's would have to be multiplied by the same factor, 2. Sometimes, as in the example of the measurement of g above, the unknown parameters appear in more complicated fashion. The problem of fitting with these more general functional forms is called **nonlinear** for obvious reasons. The **simplex algorithm** of Ch. 8 §2.3 below is an example of a nonlinear fitting procedure.

We are now going to write programs to fit both linear and nonlinear functions to data. The first and conceptually simplest of these is the **Fourier transform**, namely representing a function as a sum of sines and cosines.

§§1 Fast Fourier transform

What is a Fourier transform? Suppose we have a function that is **periodic** on the interval $0 \le x \le 2\pi$:

$$f(x + 2\pi) = f(x) \, ;$$

Then under fairly general conditions the function can be expressed in the form

$$f(x) = a_0 + \sum_{n=1}^{\infty} \left(a_n \cos(nx) + b_n \sin(nx)\right) \tag{26}$$

Another way to write Eq. 26 is

$$\mathrm{f}(x) = \sum_{-\infty}^{+\infty} c_n e^{inx}. \tag{27}$$

In either way of writing, the c_n are called **Fourier coefficients** of the function $\mathrm{f}(x)$. Looking, *e.g.*, at Eq. 27, we see that the **orthogonality** of the sinusoidal functions leads to the expression

$$c_n = \frac{1}{2\pi} \int_0^{2\pi} \mathrm{f}(x)\, e^{-inx}\, dx. \tag{28}$$

Evaluating Eq. 28 numerically requires — for given n — at least 2n points[20]. Naively, for each n = 0 to N-1 we have to do a sum

$$c_n \approx \sum_{k=1}^{2N} \mathrm{f}_k e^{-2\pi i n k/N}$$

which means carrying out $2N^2$ complex multiplications.

The **fast Fourier transform** (FFT) was discovered by Runge and König, rediscovered by Danielson and Lanczos and re-rediscovered by Cooley and Tukey[21]. The FFT algorithm can be expressed as three steps:

- Discretize the interval, *i.e.* evaluate $\mathrm{f}(x)$ only for

  $$x_k = 2\pi \frac{k}{N}\,,\quad 0 \le x \le \mathrm{N} - 1.$$

  Call $\mathrm{f}(x_k) \equiv \mathrm{f}_k$.
- Express the Fourier coefficients as

  $$c_n = \sum_{k=0}^{N-1} f_k e^{-2\pi i n k/N}. \tag{29}$$
- With $w_n = e^{-2\pi i n/N}$, Eq. 29 is an N-1'st degree polynomial in w_n. We evaluate the polynomial using a fast algorithm.

---

20. to prevent aliasing.
21. See, *e.g.*, D.E. Knuth, *The Art of Computer Programming*, v. 2 (Addison-Wesley Publishing Co., Reading, MA, 1981) p. 642.

To evaluate rapidly the polynomial

$$c_n = P_N(w_n) \equiv \sum_{k=0}^{N-1} f_k (w_n)^k$$

we divide it into two polynomials of order N/2, dividing each of those in two, *etc*. This procedure is efficient only for $N = 2^\nu$, with ν an integer, so this is the case we attack.

How does dividing a polynomial in two help us? If we segregate the odd from the even powers, we have, symbolically,

$$P_N(w) = E_{N/2}(w^2) + w\, O_{N/2}(w^2)\,. \qquad (30)$$

Suppose the time to evaluate $P_N(w)$ is T_N. Then, clearly,

$$T_N = \lambda + 2T_{N/2} \qquad (31)$$

where λ is the time to segregate the coefficients into odd and even, plus the time for 2 multiplications and a division. The solution of Eq. 31is λ(N-1). That is, it takes O(N) time to evaluate a polynomial.

However, the discreteness of the Fourier transform helps us here. The reason is this: to evaluate the transform, we have to evaluate $P_N(w_n)$ for N values of w_n. But w_n^2 takes on only N/2 values as n takes on N values. Thus to evaluate the Fourier transform for all N values of n, we can evaluate the two polynomials of order N/2 for half as many points.

Suppose we evaluated the polynomials the old-fashioned way: it would take 2(N/2)≡N multiplications to do both, but we need do this only N/2 times, and N more (to combine them) so we have $N^2/2 + N$ rather than N^2. We have gained a factor 2. Obviously it pays to repeat the procedure, dividing each of the sub-polynomials in two again, until only monomials are left.

Symbolically, the number of multiplications needed to evaluate a polynomial for N (discrete) values of w is

$$\tau_N = N\lambda + 2\,\tau_{N/2} \qquad (32)$$

whose solution is

$$\tau_N = \lambda N \log_2(N) . \quad (33)$$

Although the FFT algorithm can be programmed recursively, it almost never is. To see why, imagine how the coefficients would be re-shuffled by Eq. 30: we work out the case for 16 coefficients, exhibiting them in Table 8-1 below, writing only the indices:

Table 8-1 Bit-reversal for re-ordering discrete data

Start	Step 1	Step 2	Step 3	Bin_0	Bin_3
0	0	0	0	0000	0000
1	2	4	8	0001	1000
2	4	8	4	0010	0100
3	6	12	12	0011	1100
4	8	2	2	0100	0010
5	10	6	10	0101	1010
6	12	10	6	0110	0110
7	14	14	14	0111	1110
8	1	1	1	1000	0001
9	3	5	9	1001	1001
10	5	9	5	1010	0101
11	7	13	13	1011	1101
12	9	3	3	1100	0011
13	11	7	11	1101	1011
14	13	11	7	1110	0111
15	15	15	15	1111	1111

The crucial columns are "Start" and "Step 3". Unfortunately, they are written in decimal notation, which conceals a fact that becomes glaringly obvious in binary notation. So we re-write them in binary in the columns Bin_0 and Bin_3 —and see that the final order can be obtained from the initial order simply by reversing the order of the bits, from left to right!

A standard FORTRAN program for complex FFT is shown below. We shall simply translate the FORTRAN into FORTH as expeditiously as possible, using some of FORTH's simplifications.

One such improvement is a word to reverse the bits in a given integer. Note how clumsily this was done in the FORTRAN

```
      SUBROUTINE FOUR1 (DATA, NN, ISIGN)
C
C      from Press, et al., Numerical Recipes, ibid., p. 394.
C
C     ISIGN DETERMINES WHETHER THE FFT
C     IS FORWARD OR BACKWARD
C
C     DATA IS THE (COMPLEX) ARRAY OF DISCRETE INPUT
      COMPLEX  W, WP, TEMP, DATA(N)
      REAL*8 THETA
      J=0
      DO 11 I=0,N-1                \ begin bit.reversal
       IF (J.GT.I) THEN
         TEMP=DATA(J)
         DATA(J) = DATA(I)
         DATA(I) = TEMP
       ENDIF
       M=N/2
1        IF ((M.GE.1).AND.(J.GT.M)) THEN
         J=J-M
         M=M/2
         GO TO 1
       ENDIF
       J=J+M
11     CONTINUE            \ end bit.reversal

          MMAX=1          \ begin Danielson-Lanczos section
2         IF (N.GT.MMAX) THEN
             ISTEP=2*MMAX   \ executed lg(N) times
                            \ init trig recurrence

        THETA=3.14159265358979D0/(ISIGN*MMAX)
        WP=CEXP(THETA)
        W=DCMPLX(1.D0,0.D0)
        DO 13 M=1,MMAX,2         \ outer loop
          DO 12 I=M,N,ISTEP      \ inner loop
            J=I+MMAX             \ total=N times
            TEMP=DATA(J)*W
            DATA(J) = DATA(I)-TEMP
            DATA(I) = DATA(I) + TEMP
12          CONTINUE             \ end inner loop
C
          W=W*WP          \ trig recurrence
C
13       CONTINUE         \ end outer loop
        MMAX=ISTEP
        GO TO 2
       ENDIF           \ end Danielson-Lanczos section
       RETURN
```

program. Since practically every microprocessor permits right-shifting a register one bit at a time and feeding the overflow into another register from the right, **B.R** can be programmed easily in machine code for speed. Our fast bit-reversal procedure **B.R** may be represented pictorially as in Fig. 8-5 below.

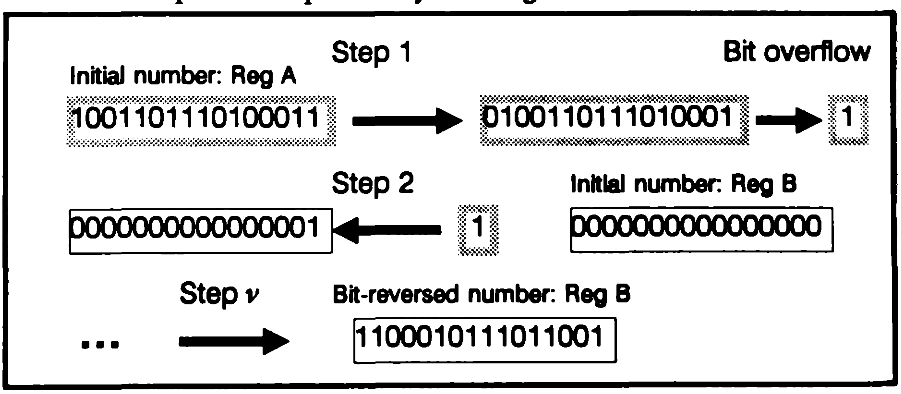

Fig. 8-5 *Pictorial representation of bit-reversal*

Bit-reversal can be accomplished in high-level FORTH *via*

```
: B.R          ( n - - n')      \ reverse order of bits
  0 SWAP       ( - - 0 n )      \ set up stack
  N.BITS 0 DO
       DUP 1 AND                \ pick out 1's bit
       ROT 2* +                 \ left shift 1, add 1's bit
       SWAP 2/                  \ right-shift n
  LOOP DROP ;
```

Note: **N.BITS** is a **VAR**, previously set to $\nu = \log_2(N)$

We will use **B.R** to re-order the actual data array (even though this is slightly more time-consuming than setting up a list of scrambled pointers, leaving the data alone). We forego indirection for two reasons: first, we have to divide by N (N steps) when inverse-transforming, so we might as well combine this with bit-reversal; second, there are N steps in rearranging and dividing by N the input vector, whereas the FFT itself takes $N\log_2(N)$ steps, *i.e.* the execution time for the preliminary N steps is unimportant.

Now, how do we go about evaluating the sub-polynomials to get the answer? First, let us write the polynomials (for our case N = 16) corresponding to taking the (bit-reversed) addresses off the stack in succession, as in Fig. 8-6 below.

$$\left.\begin{array}{l}
\left.\begin{array}{l}
\left.\begin{array}{l} w^7(f_7 + w^8 f_{15}) \\ w^3(f_3 + w^8 f_{11}) \end{array}\right\} w^3(a_3 + w^4 a_7) \\
\left.\begin{array}{l} w^5(f_5 + w^8 f_{13}) \\ w^1(f_1 + w^8 f_9) \end{array}\right\} w^1(a_1 + w^4 a_5)
\end{array}\right\} w^1(b_1 + w^2 b_3) \\
\left.\begin{array}{l}
\left.\begin{array}{l} w^6(f_6 + w^8 f_{14}) \\ w^2(f_2 + w^8 f_{10}) \end{array}\right\} w^2(a_2 + w^4 a_6) \\
\left.\begin{array}{l} w^4(f_4 + w^8 f_{12}) \\ w^0(f_0 + w^8 f_8) \end{array}\right\} w^0(a_0 + w^4 a_4)
\end{array}\right\} w^0(b_0 + w^2 b_2)
\end{array}\right\} c_0 + wc_1$$

Fig. 8-6 *The order of evaluating a 16 pt. FFT*

We see that w_n^8 (for N = 16) has only two possible values, ±1. Thus we must evaluate not 16×8 terms like $f_i + w^8 f_{i+8}$, but only 2×8. Similarly, we do not need to evaluate 16×4 terms of form $f_i + w^4 f_{i+4}$, but only 4×4, since there are only 4 possible values of w_n^4. Thus the total number of multiplications is

$$2\times8 + 4\times4 + 8\times2 + 16\times1 = 64 \equiv 16 \log_2 16,$$

as advertised. This is far fewer than 16×16 = 256, and the ratio improves with N – for example a 1024 point FFT is 100 times faster than a slow FT.

We list the FFT program on page 191 below. Since **}FFT** transforms a one-dimensional array we retain the curly braces notation introduced in Ch. 5. We want to say something like

```
V{ n.pts FORWARD }FFT
```

where **V{** is the name of the (complex) array to be transformed, **n.pts** (a power of 2) is the size of the array, and the flag-setting words **FORWARD** or **INVERSE** determine whether we are taking a FFT or inverting one.

Now we test the program. Table 8-2 on page 192 contains the weekly stock prices of IBM stock, for the year 1983 (the 52 values have been made complex numbers by adding 0i, and the table padded out to 64 entries (the nearest power of 2) with complex zeros)[22]. The first two entries (2, 64) are the type and length of the file. (The file reads from left to right.)

We FFT Table 8-2 using the phrase **IBM{ 64 DIRECT }FFT**. The power spectrum of the resulting FFT (Table 8-3) is shown in Fig. 8-7 on page 192 below.

22. This example is taken from the article "FORTH and the Fast Fourier Transform" by Joe Barnhart, *Dr. Dobb's Journal*, September 1984, p. 34.

```
\ Complex Fast Fourier Transform
\ Usage: Vector.name{ N FORWARD ( INVERSE ) }FFT
TASK FFT
\ check for presence of these extensions and load
FIND C+          0= ?( FLOAD COMPLEX )
FIND 1ARRAY      0= ?( FLOAD MATRIX.HSF )
FIND FILL        0= ?( FLOAD FILEIO.FTH )
FIND TRIG        0= ?( FLOAD TRIG )
\ if not there
DECIMAL
\= = = = = = = = = = = = = = = = = = = = = = = = =
\ auxiliary words
 CODE SHR BX 1 SHR. END-CODE
 : LG2 (n -- lg2[n]) 0 SWAP   (-- 0 n) SHR
    BEGIN ?DUP 0>
    WHILE   SHR SWAP 1+ SWAP  REPEAT ;

0 VAR DIRECTION?
: FORWARD    0 IS DIRECTION? ;
: INVERSE   -1 IS DIRECTION? ;

    0 VAR N.BITS                  \ some VARs
    0 VAR N
    0 VAR MIMAX
    0 VAR f{

: C/N   N S->F C/F ;
: NORMALIZE   DIRECTION?
      IF C/N CPSWAP C/N CPSWAP THEN ;
\ end auxiliary words
\= = = = = = = = = = = = = = = = = = = = = = = = =
\ key bit-reveral routines!
0 VAR I.R
: B.R (n -- n')                 \ reverses order of bits
  0 SWAP (-- 0 n)               \ set up stack
  N.BITS 0 DO DUP 1 AND         \ pick out 1's bit
      ROT 2* +           \ double sum and add 1's bit
      SWAP 2/            \ n - n/2
  LOOP DROP ;

: BIT.REVERSE 0 IS I.R
    N 0 DO  I B.R  IS I.R
        I.R I < NOT        ( I.R >= I ?)
        IF f{ I.R } G@L f{ I } G@L  NORMALIZE
            f{ I.R } G!L f{ I } G!L THEN
    LOOP ;
\ end bit-reversal (N times)
```

```
\= = = = = = = = = = = = = = = = = = = = = = = = =
                                   \ main algorithm
CODE C-+  2 FLD.  1 FXCH.  3 FSUBRP.
    1 FADDP.  1 FXCH.  3 FLD.
    1 FXCH.  4 FSUBRP.  1 FADDP.  1 FXCH.  END-CODE
  ( 87: w z -- w-z w+z)

: THETA  F=PI MIMAX S->F F/  DIRECTION?  FSIGN ;

CREATE WP 16 ALLOT OKLW
: INIT.TRIG  FINIT THETA EXP(I*PHI) WP DCP! C=1 ;

: NEW.W  (87: w -- w') WP  DCP@  C* ;

0 VAR ISTEP

: DO.INNER.LOOP
    DO  MIMAX I + IS I.R
        CPDUP  f{ I.R } G@L  C*  f{ I } G@L
        CPSWAP C-+  f{ I } G!L  f{ I.R }  G!L
    ISTEP +LOOP ;

: }FFT  (adr n --)  IS N  IS f{
    1 IS MIMAX
    N LG2 IS N.BITS
    FINIT
    BIT.REVERSE
    BEGIN
      N MIMAX >
    WHILE
      INIT.TRIG  MIMAX 2* IS ISTEP
      MIMAX 0 DO
        N I DO.INNER.LOOP  NEW.W
      LOOP
      ISTEP IS MIMAX
    REPEAT CPDROP ;

: POWER 0 DO f{ I } G@L CABS CR I . F. LOOP ;
\ power spectrum of FFT                \ end of fft code
\= = = = = = = = = = = = = = = = = = = = = = = = =
                                        \ an example
64 LONG COMPLEX 1ARRAY A{
: INIT.A A{ $" IBM.EX" OPEN-INPUT FILL CLOSE-INPUT ;
INIT.A
A{ 64 DIRECT }FFT
                                     \ end of example
\= = = = = = = = = = = = = = = = = = = = = = = = =
```

How do we know the FFT program actually worked? The simplest method is to inverse-transform the transform, and compare with the input file. The FFT and inverse FFT are given, respectively, in Tables 8-3 and 8-4 on page 193 below. Within roundoff error, Table 8-4 agrees with Table 8-2 on page 192.

Table 8-2 *Weekly IBM common stock prices, 1983*

2 64							
96.63	0.0	99.13	0.0	94.63	0.0	97.38	0.0
97.38	0.0	96.38	0.0	98.63	0.0	100.38	0.0
102.25	0.0	100.75	0.0	99.88	0.0	102.13	0.0
101.63	0.0	103.88	0.0	110.13	0.0	117.25	0.0
117.00	0.0	117.63	0.0	116.50	0.0	110.63	0.0
113.00	0.0	114.00	0.0	114.25	0.0	121.13	0.0
123.00	0.0	121.00	0.0	121.50	0.0	120.13	0.0
124.38	0.0	120.38	0.0	119.75	0.0	118.50	0.0
122.50	0.0	117.83	0.0	119.75	0.0	122.25	0.0
123.13	0.0	126.63	0.0	126.88	0.0	132.25	0.0
131.75	0.0	127.00	0.0	128.00	0.0	122.25	0.0
126.88	0.0	123.50	0.0	121.00	0.0	117.88	0.0
122.25	0.0	120.88	0.0	123.63	0.0	122.00	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

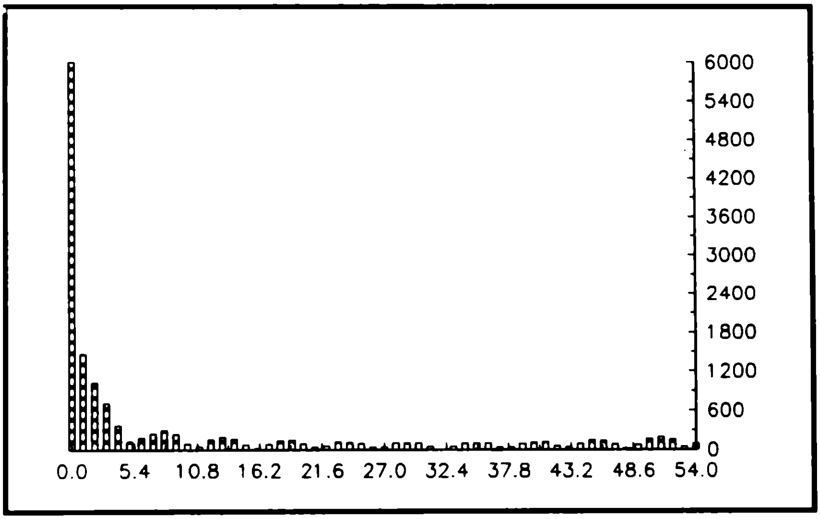

Fig. 8-7 *Power spectrum of FFT of 1983 IBM prices (from Table 8-3)*

Table 8-3 *FFT of IBM weekly stock prices, 1983*

0	5989.4599	0.0000000	32	3.1601582	0.0000000	16	7.2500000	-15.170043	48	7.2500000	15.170043
1	-1356.9239	562.94860	33	13.450067	44.886729	17	-27.548482	81.061904	49	69.775169	29.838697
2	-374.28417	956.64282	34	89.175967	32.346036	18	68.084251	117.80709	50	146.10401	-64.187637
3	305.66314	634.45819	35	99.756584	-25.042837	19	119.02930	71.422492	51	84.502433	-165.21282
4	327.40642	177.60118	36	39.756886	-83.410888	20	87.868364	-24.258874	52	-49.510669	-147.46129
5	125.21042	-8.0063304	37	-30.826173	-14.841442	21	1.8497861	-37.864341	53	-49.475231	5.4212093
6	-178.28504	39.630813	38	19.293210	29.322504	22	-11.327768	52.313049	54	80.279641	33.341926
7	-90.525482	226.27159	39	86.785362	22.908525	23	44.172229	110.77391	55	191.18481	-128.15400
8	150.24284	280.91506	40	107.71731	-34.415077	24	107.71731	34.415077	56	150.24284	-280.91506
9	191.18481	128.15400	41	44.172229	-110.77391	25	86.785362	-22.908525	57	-90.525482	-226.27159
10	80.279641	-33.341926	42	-11.327768	-52.313049	26	19.293210	-29.322504	58	-178.28504	-39.630813
11	-49.475231	-5.4212093	43	1.8497861	37.864341	27	-30.826173	14.841442	59	125.21042	8.0063304
12	-49.510669	147.46129	44	87.868354	24.258874	28	39.756886	83.410888	60	327.40642	-177.60118
13	84.502433	165.21282	45	119.02930	-71.422492	29	99.756584	25.042837	61	305.66314	-634.45819
14	146.10401	64.187637	46	68.084251	-117.80709	30	89.175967	-32.346036	62	-374.28417	-956.64282
15	69.775169	-29.838697	47	-27.548482	-81.061904	31	13.450067	-44.886729	63	-1356.9239	-562.94860

Table 8-4 *Reconstructed IBM prices (inverse FFT)*

0	96.630	0.0000	32	122.50	0.0000	16	117.00	-.000000000	48	122.25	.000000000
1	99.129	.000000705	33	117.82	-.000000245	17	117.62	-.000000518	49	120.87	.000000132
2	94.629	.000000023	34	119.75	.000000148	18	116.50	-.000000119	50	123.63	-.000000052
3	97.379	.000000227	35	122.24	-.000000379	19	110.62	.000001375	51	121.99	-.000000944
4	97.380	.000000385	36	123.13	-.000000385	20	113.00	.000001076	52	.000012664	-.000001076
5	96.379	-.000000362	37	126.62	.000000796	21	114.00	.000000975	53	-.000001277	-.000001434
6	98.630	.000001697	38	126.88	-.000000851	22	114.25	.000000378	54	-.000002880	-.000001224
7	100.38	.000002158	39	132.25	-.000001661	23	121.12	.000000317	55	-.000005683	-.000000315
8	102.25	-.000000000	40	131.75	-.000000000	24	123.00	.000000000	56	-.000001602	-.000000000
9	100.75	.000000799	41	127.00	-.000000373	25	121.00	.000001130	57	-.000002995	-.000001630
10	99.880	-.000000271	42	128.00	.000000401	26	121.50	-.000001469	58	-.000010348	.000001339
11	102.12	-.000000078	43	122.24	-.000001011	27	120.12	.000002522	59	-.000005928	-.000001711
12	101.62	.000000316	44	126.87	-.000000316	28	124.37	.000000027	60	-.000003016	-.000000027
13	103.88	-.000000043	45	123.50	.000000390	29	120.38	.000001261	61	-.000002253	-.000001583
14	110.13	-.000001615	46	121.00	.000001937	30	119.75	-.000000618	62	-.000003121	.000000295
15	117.25	-.000002237	47	117.87	.000001164	31	118.49	.000000573	63	-.000003713	.000000500

§§2 Gram polynomials

Gram polynomials are useful in fitting data by the linear least-squares method. The usual method is based on the following question: What is the "best" polynomial,

$$P_N(x) = \sum_{n=0}^{N} \gamma_n x^n, \tag{34}$$

(of order N) that I can use to fit some set of M pairs of data points,

$$\begin{Bmatrix} x_k \\ f_k \end{Bmatrix}, \; k=0, 1, \ldots, M-1$$

(with M > N) where f(x) is measured at M distinct values of the independent variable x ?

The usual answer, found by Gauss, is to minimize the **squares** of the **deviations** (at the points x_k) of the fitting function $P_N(x)$ from the data — possibly weighted by the uncertainties of the data. That is, we want to minimize the **statistic**

$$\chi^2 = \sum_{k=0}^{M-1} \left(f_k - \sum_{n=0}^{N} \gamma_n x_k^n \right)^2 \frac{1}{\sigma_k^2} \qquad (35)$$

with repect to the N + 1 parameters γ_n .

From the differential calculus we know that a function's first derivative vanishes at a minimum, hence we differentiate χ^2 with respect to each γ_n independently, and set the results equal to zero. This yields N + 1 linear equations in N + 1 unknowns:

$$\sum_m A_{nm} \gamma_m = \beta_n , \quad n=0, 1, \ldots, N \qquad (36)$$

where (the symbol $\hat{=}$ means "is defined by")

$$A_{nm} \hat{=} \sum_{k=0}^{M-1} (x_k)^{n+m} \frac{1}{\sigma_k^2} \qquad (37a)$$

and

$$\beta_n \hat{=} \sum_{k=0}^{M-1} x_k^n f_k \frac{1}{\sigma_k^2} \qquad (37b)$$

In Chapter 9 we develop methods for solving linear equations. Unfortunately, they cannot be applied to Eq. 36 for $N \gtrsim 9$ because the matrix A_{nm} approximates a Hilbert matrix,

$$H_{nm} = \frac{const.}{n+m+1},$$

a particularly virulent example of an exponentially **ill-conditioned matrix**. That is, the roundoff error in solving 36 grows exponentially with N, and is generally unacceptable. We can avoid roundoff problems by expanding in polynomials rather than monomials:

$$\chi^2 = \sum_{k=0}^{M-1} \left(f_k - \sum_{n=0}^{N} \gamma_n p_n(x_k) \right)^2 \frac{1}{\sigma_k^2} . \tag{38}$$

The matrix then becomes

$$A_{nm} = \sum_{k=0}^{M-1} p_n(x_k) p_m(x_k) \frac{1}{\sigma_k^2} \tag{39a}$$

and the inhomogeneous term is now

$$\beta_n = \sum_{k=0}^{M-1} p_n(x_k) f_k \frac{1}{\sigma_k^2} \tag{39b}$$

Is there any choice of the polynomials $p_n(x)$ that will eliminate roundoff? The best kinds of linear equations are those with nearly diagonal matrices. We note the sum in Eq. 39a is nearly an integral, if M is large. If we choose the polynomials so they are **orthogonal** with respect to the weight function

$$w(x) = \frac{1}{\sigma_k^2} \theta(x_k - x)\, \theta(x - x_{k-1}) ,$$

where

$$\theta(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

then A_{nm} will be nearly diagonal, and well-conditioned.