12 Fast Fourier Transform

Functions:

four1: fourier transform (FFT) in one dimension
twofft: fourier transform of two real functions
realft: fourier transform of a real functions
sinft: sine transform using the FFT
cosft: cosine transform using the FFT
convlv: convolution or deconvolution of data using the FFT
correl: correlation or autocorrelation of data using the FFT
spctrm: power spectrum estimation using the FFT
memcof: power spectrum estimation, evaluate maximum entropy coefs
evlmem: power spectrum estimation using the maximum entropy coefs
fixrts: roots of a polynomial, reflects inside the unit circle
predic: linear prediction using the MEM coefficients
fourn: multidimensional FFT

---

four1

Description:

This function computes a Fast Fourier Transform in one dimension.

$$\mathbf{four1(data)} \longrightarrow \mathbf{data}$$

Where:

data on input is a real list with length a power of 2 and on output the discrete Fourier transform or inverse discrete Fourier tranform, depending on the value of $four1\_isign$.

Pre-variables:

four1_isign [1] an integer such that $four1$ computes the Fourier transform if the value is 1 and the inverse transform if the value is -1.

Example:

This example tests properties of the FFT.

```
-> four1: fourier transform (FFT) in one dimension #
-> Make function to splice two lists into one, where the first #
-> represents the real parts and the second the imaginary parts #
-> make_complex(x,y) := sprog(z : [],
                       for i to length(x) do sprog(
                         z : insert_after(z,proj(x,i)),
                         z : insert_after(z,proj(y,i))),
                       z);
-> Set up the domain array: #
-> input : 2.0*listn(0,15) + const(1,16)?
[1.00, 3.00, 5.00, 7.00, 9.00, 11.00, 13.00, 15.00, 17.00, 19.00, 21.00, 23.00,
25.00, 27.00, 29.00, 31.00]
-> four1_isign : 1;
-> f(t) = real-valued even-function and so should get #
-> F(n) = F(N-n) and is real #
-> f(x) := sqrt(x*x+1.0);
-> real_part : map(f,input);
-> imag_part : const(0.0,length(real_part));
-> data : make_complex(real_part,imag_part)?
[1.41, 0.00, 3.16, 0.00, 5.10, 0.00, 7.07, 0.00, 9.06, 0.00, 11.05, 0.00,
13.04, 0.00, 15.03, 0.00, 17.03, 0.00, 19.03, 0.00, 21.02, 0.00, 23.02, 0.00,
25.02, 0.00, 27.02, 0.00, 29.02, 0.00, 31.02, 0.00]
-> res : four1(data)?
[257.09, 0.00, -15.46, -80.20, -15.57, -38.44, -15.63, -23.80, -15.66, -15.89,
-15.68, -10.61, -15.69, -6.58, -15.70, -3.16, -15.70, 0.00, -15.70, 3.16,
-15.69, 6.58, -15.68, 10.61, -15.66, 15.89, -15.63, 23.80, -15.57, 38.44,
-15.46, 80.20]
-> Take Inverse-transform (should get data)#
-> four1_isign : -1;
-> 1/length(input)*four1(res)?
[1.41, -0.00, 3.16, -0.00, 5.10, -0.00, 7.07, 0.00, 9.06, 0.00, 11.05, 0.00,
13.04, 0.00, 15.03, -0.00, 17.03, 0.00, 19.03, -0.00, 21.02, -0.00, 23.02,
0.00, 25.02, -0.00, 27.02, -0.00, 29.02, -0.00, 31.02, 0.00]
```

twofft

Description:

This function computes the Fourier Transform of two data sets.

twofft(data1, data2) $\longrightarrow$ **[fft1, fft2]**

Where:

data1 is a real list with length a power of 2,
data2 is a real list with length the same as that of $data1$,
fft1 is a list of two floating point lists being the complex Fourier transform of $data1$,
fft2 is a list of two floating point lists being the complex Fourier transform of $data2$.

Example:

This example takes the two real, alternating-sign, data values given and returns their (complex-valued) FFT.

```
-> data1 : [1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0];
-> data2 : [-1.0, 1.0, -1.0, 1.0, -1.0, 1.0, -1.0, 1.0];
-> res : twofft(data1,data2)?
[[[0.0000000, 0.0000000, 0.0000000, 0.0000000, 8.0000000, 0.0000000, 0.0000000,
0.0000000], [0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000]], [[0.0000000, 0.0000000, 0.0000000, 0.0000000,
-8.0000000, 0.0000000, 0.0000000, 0.0000000], [0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000]]]
-> four1_isign : -1;
-> Test Inverse (should get data1 & data2): #
-> 1/length(data1)*twofft(first(first(res)),first(last(res)))?
[[[1.0000000, -1.0000000, 1.0000000, -1.0000000, 1.0000000, -1.0000000,
1.0000000, -1.0000000], [0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000]], [[-1.0000000, 1.0000000, -1.0000000,
1.0000000, -1.0000000, 1.0000000, -1.0000000, 1.0000000], [0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000]]]
```

realft

Description:

This function computes the real Fourier Transform of a data set.

realft(data) ⟶ data

Where:

data on input is a real list with length a power of two, and on output is the positive frequency half of its complex Fourier transform, with $data_1$ and $data_2$ containing the first and last components respectively of the transform.

Pre-variables:

realft_isign [1] an integer such that $realft$ computes the Fourier transform if the value is 1 and the inverse transform if the value is -1.

Example:

The simple 4 element data set is transformed and then inverted to give the original data once more.

```
-> data : [1.0 ,2.0 ,3.0 ,4.0];
-> res : realft(data)?
[10.0000000, -2.0000000, -2.0000000, -2.0000000]
-> Test Inverse (should get data): #
-> realft_isign : -1;
-> 2/length(data)*realft(res)?
[1.0000000, 2.0000000, 3.0000000, 4.0000000]
```

sinft

Description:

This function computes the Fourier Sine Transform using the FFT.

$$\textbf{sinft(y)} \longrightarrow \textbf{y}$$

Where:

y on input is a real list with length a power of 2 and on output the Fourier Sine Transform.

Example:

Notice that the sine FT is self inverting, needing no equivalent of the *four1_isign* of *four1*.

```
-> y : [0.0, 1.0, 2.0, 3.0];
-> res : sinft(y)?
[0.00, 4.83, -2.00, 0.83]
-> 2/length(y)*sinft(res)?
[0.00, 1.00, 2.00, 3.00]
```

---

cosft

Description:

This function computes the Fourier Cosine Transform using the FFT

$$\textbf{cosft(y)} \longrightarrow \textbf{y}$$

Where:

y on input is a real list with length a power of 2 and on output the Fourier Cosine Transform.

Pre-variables:

cosft_isign [1] an integer such that *cosft* computes the Fourier transform if the value is 1 and the inverse transform if the value is -1.

Example:

The cosine FT, unlike the sine FT, is not self-inverting.

```
-> y : [1.0, 2.0, 3.0, 4.0];
-> res : cosft(y)?
[10.0000000, -0.4142136, -2.0000000, 2.4142136]
```

---

convlv

Description:

This function computes the convolution or deconvolution of data using the FFT.

convlv(data, respns) ⟶ ans

Where:

data is a real list with length a power of 2,
respns is a real list of odd length less than that of *data*,
ans is a floating point list representing the convolution or deconvolution depending on the value of *convlv_isign*.

Pre-variables:

convlv_isign [1] an integer such that *convlv* computes the convolution if the value is 1 and the deconvolution if the value is -1.

Example:

In this example the convolution of a data set and a response are computed using `convlv` and the result is then obtained by direct calculation.

```
-> convlv: convolution or deconvolution of data using the FFT #
-> data : [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0,
        0.0, 0.0, 0.0];
-> n : length(data)?
16
-> respns : [0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0];
-> convlv(data,respns)?
[0.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, -0.0000, 1.0000, 2.0000,
3.0000, 3.0000, 3.0000, 2.0000, 1.0000, 0.0000, 0.0000]
-> Compare with : #
-> cmp : const(0.0,n);
-> for i to n do
  for j to 4 do sprog(
    cmp : put(cmp,i,
      proj(cmp,i)+proj(data,mod(i-j-1+16,16)+1)*proj(respns,j+1)),
    cmp : put(cmp,i,
      proj(cmp,i)+proj(data,mod(i+j-1,16)+1)*proj(respns,9-j+1)));
-> cmp?
[0.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 0.0000, 1.0000, 2.0000,
3.0000, 3.0000, 3.0000, 2.0000, 1.0000, 0.0000, 0.0000]
```

correl

Description:

This function computes the correlation or autocorrelation of data using the FFT.

correl(data1, data2) ⟶ ans

Where:

data1 is a real list of length a power of 2,
data2 is a real list having the same length as *data*1,
ans is a floating point list representing the correlation of *data*1 and *data*2.

Example:

In this example the autocorrelation of a data set with itself is computed using `correl` and then a direct calculation performed.

```
-> correl: correlation or autocorrelation of data using the FFT #
-> data1 : [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,
         1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
         1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
         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];
-> data2 : data1;
-> correl(data1,data2)?
[16.0000, 15.0000, 14.0000, 13.0000, 12.0000, 11.0000, 10.0000, 9.0000, 8.0000,
7.0000, 6.0000, 5.0000, 4.0000, 3.0000, 2.0000, 1.0000, 0.0000, 0.0000,
-0.0000, 0.0000, -0.0000, 0.0000, -0.0000, 0.0000, -0.0000, 0.0000, -0.0000,
0.0000, -0.0000, 0.0000, -0.0000, 0.0000, 0.0000, 0.0000, -0.0000, 0.0000,
-0.0000, 0.0000, -0.0000, 0.0000, -0.0000, 0.0000, -0.0000, -0.0000, -0.0000,
-0.0000, -0.0000, -0.0000, -0.0000, 1.0000, 2.0000, 3.0000, 4.0000, 5.0000,
6.0000, 7.0000, 8.0000, 9.0000, 10.0000, 11.0000, 12.0000, 13.0000, 14.0000,
15.0000]
-> mod(x,y) := x-trunc(float(x/y))*y;
-> cmp : const(0.0,17);
-> for i from 0 by 1 to 16 do
  for j to 64 do
    cmp : put(cmp,i+1,proj(cmp,i+1)+proj(data1,mod(i+j-1,64)+1)*proj(data2,j));
-> cmp?
[16.0000, 15.0000, 14.0000, 13.0000, 12.0000, 11.0000, 10.0000, 9.0000, 8.0000,
7.0000, 6.0000, 5.0000, 4.0000, 3.0000, 2.0000, 1.0000, 0.0000]
```

spctrm

Description:

This function estimates the power spectrum of a data set using the FFT.

spctrm(data, m, k) $\longrightarrow$ p

Where:

data is a string representing a file or path name where the corresponding file contains at least $(2k+1)m$ (if `spctrm_ovrlap` is `true`) or $4km$ real data points separated by spaces or new-lines,
m is an integer,
k is an integer,
p is a floating point list representing the power spectrum.

Pre-variables:

window [`"parzen"`] a string being one of `"parzen"`, `"hanning"` or `"welch"`,
ovrlap [true] a symbol the other permissible value being false.

Example:

This example reads a file called *spctrl.dat* which reads in 1200 numbers formatted as specified above.

```
-> With overlap: #
-> spctrm("spctrl.dat",16,8)?
[0.0019927, 0.0129293, 0.0721972, 0.0205566, 0.0731488, 0.0149080, 0.0081313,
0.0035726, 0.0000494, 0.0076034, 0.0048877, 0.0057931, 0.0243256, 0.0032049,
0.0025303, 0.0019718]
-> Without overlap: #
-> spctrm_ovrlap : false;
-> spctrm("spctrl.dat",16,16)?
[0.0020707, 0.0186441, 0.0715228, 0.0178975, 0.0735840, 0.0215917, 0.0123840,
0.0034132, 0.0000577, 0.0077383, 0.0060252, 0.0053334, 0.0243538, 0.0032364,
0.0042381, 0.0024021]
```

memcof

Description:

This function evaluates the maximum entropy coefficients of the power spectrum.

memcof(data, m) ⟶ [cof, pm]

Where:

data is a real list,
m is a positive integer,
cof is a floating point list representing the coefficients for Maximum Entropy Method spectral estimation a_j,
pm is a floating point number representing the coefficient for Maximum Entropy Method spectral estimation a_0.

Example:

The following example assigns the first 1000 numbers from the file *spctrl.dat* to a list called *data*. The list assignment is carried out in the file *spctrl.sen*. The results contained in the comments below are as per *Numerical Recipes*.

```
-> m : 10;
-> silent_bat("spctrl");
-> memcof(data,m)?
[[1.2615389, -0.0076949, -0.6467774, -0.2806016, 0.1636910, 0.3476751,
0.1112469, -0.3371408, -0.3580436, 0.3787748], 0.0035114]
```

evlmem

Description:

This function estimates the power spectrum using maximum entropy coefficients.

evlmem(fdt, pm, cof) ⟶ value

Where:

fdt is a real number,
pm is a real number as returned by *memcof*,
cof is a real list as returned by *memcof*,
value is a floating point number representing the power spectrum estimate $P(f)$ at fdt.

Example:

The results of *memcof* above are used here.

```
-> cof :  [1.261539, -0.007695, -0.646778, -0.280603, 0.163693,
           0.347674, 0.111247, -0.337141, -0.358043, 0.378774];
-> pm : 0.003511;
-> for i from 0 by 1 to 16 do
     print([i/32.0, evlmem(i/32.0, pm, cof)]);
[0.0000000, 0.0260202]
[0.0312500, 0.0292624]
[0.0625000, 0.1930626]
[0.0937500, 0.1392248]
[0.1250000, 29.9161991]
[0.1562500, 0.0038779]
[0.1875000, 0.0006329]
[0.2187500, 0.0003335]
[0.2500000, 0.0004373]
[0.2812500, 0.0013308]
[0.3125000, 0.0007796]
[0.3437500, 0.0004512]
[0.3750000, 0.0007842]
[0.4062500, 0.0013810]
[0.4375000, 0.0006494]
[0.4687500, 0.0007748]
[0.5000000, 0.0017154]
```

fixrts

Description:

This function computes the roots of the linear prediction polynomial, reflects any inside the unit circle in the complex plane if the modulus is greater than one, and returns a set of modifed coefficients.

fixrts(d) ⟶ d

Where:

d on input a real list representing the coefficients of the characteristic polynomial $z^N - \sum_{j=1}^{N} d_j z^{N-j} = 0$. On output a floating point list representing the coefficients of a modified polynomial all of whose roots are inside the unit circle.

Pre-variables:

npmax [100] an integer representing the largest expected number of roots.

Example:

Take, for example, the polynomial $(z-1)^6 - 1 = z^6 - (6z^5 - 15z^4 + 20z^3 - 15z^2 + 6z)$. The roots of this polynomial are the elements of $\{e^{i\pi n/3} + 1 : n = 0, 1, ..., 5\}$.

```
-> d : [6.0, -15.0, 20.0, -15.0, 6.0, 0.0];
-> new_d : fixrts(d)?
[2.4999999, -3.3333332, 2.4999999, -1.0000000, 0.1666667, 0.0000000]
```

predic

Description:

This function makes linear predictions based on the maximum entropy method spectral estimation method coefficients.

predic(data, d, nfut) ⟶ future

Where:

data is a real list representing data,
d is a real list representing the data's LP coefficients,
nfut is an integer being the number of points to predict,
future is a floating point list representing the predicted values.

Pre-variables:

npmax [100] an integer representing the largest expected number of roots.

Example:

Take the sample function $f(n) = e^{-n/500} sin(2\pi n/50) + e^{-2n/500} sin(2.2\pi n/50)$.

```
-> pi : float(%pi);
-> f(n) := exp(-n/500.0)*sin(2*pi*n/50) + exp(-2*n/500.0)*sin(2.2*pi*n/50);
-> data : map(f,1.0*listn(1,500));
-> d : first(memcof(data,10))?
[3.6667757, -3.8323970, -0.3018626, 1.4169940, 1.0427295, -0.1253339,
-1.1263073, -0.9166208, 1.7689100, -0.5929279]
-> new_d : fixrts(d)?
[3.6295784, -3.7328895, -0.3450613, 1.3617951, 1.0403466, -0.0882578,
-1.0935015, -0.9259536, 1.7249898, -0.5710855]
-> predic(data,new_d,20)?
[0.0645891, 0.1277688, 0.1884934, 0.2457639, 0.2986450, 0.3462805, 0.3879071,
0.4228668, 0.4506175, 0.4707411, 0.4829502, 0.4870918, 0.4831492, 0.4712413,
0.4516198, 0.4246640, 0.3908738, 0.3508598, 0.3053332, 0.2550920]
-> Compare with function values at these points: #
-> map(f,1.0*listn(501,520))?
[0.0645890, 0.1277686, 0.1884925, 0.2457618, 0.2986405, 0.3462716, 0.3878909,
0.4228393, 0.4505729, 0.4706718, 0.4828462, 0.4869406, 0.4829355, 0.4709466,
0.4512225, 0.4241391, 0.3901929, 0.3499918, 0.3042439, 0.2537452]
```

fourn

Description:

This function computes the Fourier Transform (FFT) in many dimensions.

fourn(data, nn) ⟶ data

Where:

data on input is a real list of length twice the product of the elements in *nn* and on output the n-dimensional discrete Fourier transform,
nn is an integer list each element being a power of 2.

Pre-variables:

fourn_isign [1] an integer such that *fourn* computes the Fourier transform if the value is 1 and the inverse transform if the value is -1.

Example:

```
-> nn : [2,2,2];
-> f(i,j,k) := 2*(k+4*(j-1)+32*(i-1)) - 1;
-> data : [];
-> for i to 4 do
     for j to 2 do
       for k to 2 do data : insert_after(data,f(i,j,k));
-> data?
[1, 3, 9, 11, 65, 67, 73, 75, 129, 131, 137, 139, 193, 195, 201, 203]
-> fourn(data,nn)?
[808.0000000, 824.0000000, -32.0000000, -32.0000000, -256.0000000,
-256.0000000, 0.0000000, 0.0000000, -512.0000000, -512.0000000, 0.0000000,
0.0000000, 0.0000000, 0.0000000, 0.0000000, 0.0000000]
```