All-At-Once Fitting Functions

The scheme of calculating one Y value at a time doesn't work well for some fitting functions. This is true of functions that involve a convolution such as might arise if you are trying to fit a theoretical signal convolved with an instrument response. Fitting to a solution to a differential equation might be another example.

For this case, you can create an "all at once" fit function. Such a function provides you with an X and Y wave. The X wave is input to the function; it contains all the X values for which your function must calculate Y values. The Y wave is for output — you put all your Y values into the Y wave.

Because an all-at-once function is called only once for a given set of fit coefficients, it will be called many fewer times than a basic fit function. Because of the saving in function-call overhead, all-at-once functions can be faster even for problems that don't require an all-at-once function.

Here is the format of an all-at-once fit function:

```
Function myFitFunc(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   yw = <expression involving pw and xw>
End
```

Note that there is no return statement because the function result is put into the wave yw. Even if you include a return statement the return value is ignored during a curve fit.

The X wave contains all the X values for your fit, whether you provided an X wave to the curve fit or not. If you did not provide an X wave, xw simply contains equally-spaced values derived from your Y wave's X scaling.

There are some restrictions on all-at-once fitting functions:

1) You can't create or edit an all-at-once function using the Curve Fitting dialog. You must create it by editing in the Procedure window. All-at-once functions are, however, listed in the Function menu in the Curve Fitting dialog.
2) There is no such thing as an "old-style" all-at-once function. It must have the FitFunc keyword.
3) You don't get mnemonic coefficient names.

Here is an example that fits an exponential decay using an all-at-once function. This example is silly — there is no reason to make this an all-at-once function. It is simply an example showing how a real function works without the computational complexities. Here it is, as an all-at-once function:

```
Function allatonce(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   // a wave assignment does the work
   yw = pw[0] + pw[1]*exp(-xw/pw[2])
End
```

This is the same function written as a standard user fitting function:

```
Function notallatonce(pw, x) : FitFunc
   WAVE pw
   Variable x

   return pw[0] + pw[1]*exp(-x/pw[2])
End
```

In the all-at-once function, the argument of exp() includes xw, a wave. The basic format uses the input parameter x, which is a variable containing a single value. The use of xw in the all-at-once version of the function uses the implied point number feature of wave assignments (see **Waveform Arithmetic and Assignments** on page II-74).

There are some things to watch out for when creating an all-at-once fitting function:

1. You must not change the yw wave except for assigning values to it. You must not resize it. This will get you into trouble:

```
Function allatonce(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   Redimension/N=2000 yw                  // BAD!
   yw = pw[0] + pw[1]*exp(-xw/pw[2])
End
```

2. You may not get the same number of points in yw and xw as you have in the waves you provide as the input data. If you fit to a restricted range, if there are NaNs in the input data, or if you use a mask wave to select a subset of the input data, you will get a wave with a reduced number of points. Your fitting function must be written to handle that situation or you must not use those features.
3. It is tricky, but not impossible, to write an all-at-once fitting function that works correctly with the auto-destination feature (that is, _auto_ in the Destination menu, or /D by itself in a FuncFit command).
4. The xw and yw waves are *not* your data waves. They are created by Igor during the execution of CurveFit, FuncFit or FuncFitMD and filled with data from your input waves. They will be destroyed when fitting is finished. For debugging, you can use Duplicate to save a copy of the waves. Altering the contents of the xw wave will have no effect.

The example above uses xw as an argument to the exp function, and it uses the special features of a wave assignment statement to satisfy point 2.

The next example fits a convolution of a Gaussian peak with an exponential decay. It fits a baseline offset, amplitude and width of the Gaussian peak and the decay constant for the exponential. This might model an instrument with an exponentially decaying impulse response to recover the width of an impulsive signal.

```
Function convfunc(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   // pw[0] = gaussian baseline offset
   // pw[1] = gaussian amplitude
   // pw[2] = gaussian position
   // pw[3] = gaussian width
   // pw[4] = exponential decay constant of instrument response

   // Make a wave to contain an exponential with decay
   // constant pw[4]. The wave needs enough points to allow
   // any reasonable decay constant to get really close to zero.
   // The scaling is made symmetric about zero to avoid an X
   // offset from Convolve/A
   Variable dT = deltax(yw)
   Make/D/O/N=201 expwave           // Long enough to allow decay to zero
   SetScale/P x -dT*100,dT,expwave

   // Fill expwave with exponential decay
   expwave = (x>=0)*pw[4]*dT*exp(-pw[4]*x)

   // Normalize exponential so that convolution doesn't change
   // the amplitude of the result
   Variable sumexp
   sumexp = sum(expwave)
   expwave /= sumexp

   // Put a Gaussian peak into the output wave
   yw = pw[1]*exp(-((x-pw[2])/pw[3])^2)

   // Convolve with the exponential. NOTE /A.
   Convolve/A expwave, yw

   // Add the vertical offset AFTER the convolution to avoid end effects
```

```
   yw += pw[0]
End
```

Some things to be aware of with regard to this function:

1. A wave is created inside the function to store the exponential decay. Making a wave can be a time-consuming operation; it is less convenient for the user of the function, but can save computation time if you make a suitable wave ahead of time and then simply reference it inside the function.
2. The wave containing the exponential decay is passed to the convolve operation. The use of the /A flag prevents the convolve operation from changing the length of the output wave yw. You may wish to read the section on **Convolution** on page III-284.
3. The output wave yw is used as a parameter to the convolve operation. Because the convolve operation assumes that the data are evenly-space, this use of yw means that the function *does not satisfy* points 2) or 3) above. If you use input data to the fit that has missing points or unevenly-spaced X values, this function *will fail*.

You may also find the section **Waveform Arithmetic and Assignments** on page II-74 helpful.

Here is a much more complicated version of the fitting function that solves these problems, and also is more robust in terms of accuracy. The comments in the code explain how the function works.

Note that this function makes at least two different waves using the Make operation, and that we have used Make/D to make the waves double-precision. This can be crucial. To improve performance and reduce clutter in your Igor experiment file, we create the intermediate waves as free waves

```
Function convfunc(pw, yw, xw) : FitFunc
   WAVE pw, yw, xw

   // pw[0] = gaussian baseline offset
   // pw[1] = gaussian amplitude
   // pw[2] = gaussian position
   // pw[3] = gaussian width
   // pw[4] = exponential decay constant of instrument response

   // Make a wave to contain an exponential with decay
   // constant pw[4]. The wave needs enough points to allow
   // any reasonable decay constant to get really close to zero.
   // The scaling is made symmetric about zero to avoid an X
   // offset from Convolve/A.

   // resolutionFactor sets the degree to which the exponential
   // will be over-sampled with regard to the problem's parameters.
   // Increasing this number increases the number of time
   // constants included in the calculation. It also decreases
   // the point spacing relative to the problem's time constants.
   // Increasing will also increase the time required to compute

   Variable resolutionFactor = 10
   // dt contains information on important time constants.
   // We wish to set the point spacing for model calculations
   // much smaller than the exponential time constant or gaussian width.
   // Use absolute values to prevent failures if an iteration strays
   // into unmeaningful but mathematically allowed negative territory.

   Variable absDecayConstant = abs(pw[4])
   Variable absGaussWidth = abs(pw[3])
   Variable dT = min(absDecayConstant/resolutionFactor, absGaussWidth/resolutionFactor)

   // Calculate suitable number points for the exponential. Length
   // of exponential wave is 10 time constants; doubled so
   // exponential can start in the middle; +1 to make it odd so
   // exponential starts at t=0, and t=0 is exactly the middle
```

```
// point. That is better for the convolution.

Variable nExpWavePnts = round((10*absDecayConstant)/dT)*2 + 1

// Important: Make double-precision waves.
// We make free waves to improve performance and so that
// the intermediate waves clean themselves up at the end of
// function execution.

Make/D/FREE/O/N=(nExpWavePnts) expwave// Double-precision free wave

// In this version of the function, we make a y output wave
// ourselves, so that we can control the resolution and
// accuracy of the calculation. It also will allow us to use
// a wave assignment later to solve the problem of variable
// X spacing or missing points.

Variable nYPnts = max(resolutionFactor*numpnts(yw), nExpWavePnts)
Make/D/FREE/O/N=(nYPnts) yWave         // Double-precision free wave

// This wave scaling is set such that the exponential will
// start at the middle of the wave
SetScale/P x -dT*(nExpWavePnts/2),dT,expwave

// Set the wave scaling of the intermediate output wave to have
// the resolution calculated above, and to start at the first
// X value.
SetScale/P x xw[0],dT, yWave

// Fill expwave with exponential decay, starting with X=0
expwave = (x>=0)*dT/pw[4]*exp(-x/pw[4])

// Normalize exponential so that convolution doesn't change
// the amplitude of the result
Variable sumexp
sumexp = sum(expwave)
expwave /= sumexp

// Put a Gaussian peak into the intermediate output wave. We use
// our own wave (yWave) because the convolution requires a wave
// with even spacing in X, whereas we may get X values input that
// are not evenly spaced.
// Also, we do not add the vertical offset here - it will cause problems
// with the convolution. Before the convolution Igor zero-pads the
// wave, so the baseline here must be zero. Otherwise there is
// a step function at the start of the convolution.
yWave = pw[1]*exp(-((x-pw[2])/pw[3])^2)

// Now convolve with the exponential. NOTE /A.
Convolve/A expwave, yWave

// Move appropriate values corresponding to input X data into
// the output Y wave. We use a wave assignment involving the
// input X wave. This will extract the appropriate values from
// the intermediate wave, interpolating values where the
// intermediate wave doesn't have a value precisely at an X
// value that is required by the input X wave. This wave
// assignment also solves the problem with auto-destination:
// The function can be called with an X wave that sets any X
// spacing, so it doesn't matter what X values are required.
// This is the appropriate place to add the vertical offset.
```