Chapter 14
Statistical Functions

This chapter lists and describes many of Mathcad's built-in statistical functions. These functions perform a wide variety of computational tasks, including statistical analysis, interpolation, regression, and smoothing.

The following sections make up this chapter:

Population and sample statistics

Functions for computing the mean, variance, standard deviation, and correlation of data.

Probability distributions

Functions for evaluating probability densities, cumulative probability distributions and their inverses for over a dozen common distribution functions.

Histogram function

How to count the number of data values falling into specified intervals.

Random numbers

Generating random numbers having various distributions.

Interpolation and prediction functions

Linear and cubic spline interpolation. Functions for multivariate interpolation.

Regression functions

Functions for linear regression, polynomial regression, and regression using combinations of arbitrary functions.

Smoothing functions

Functions for smoothing time series with either a running median, a Gaussian kernel, or an adaptive linear least-squares method.

Population and sample statistics

Mathcad includes eight functions for population and sample statistics. In the following descriptions, m and n represent the number of rows and columns in the specified arrays. In the formulas below, the built-in variable ORIGIN is set to its default value of zero.

---

mean(**A**) Returns the mean of the elements of an $m \times n$ array **A** using the formula:

$$\text{mean}(\mathbf{A}) = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} A_{i,j}$$

median(**A**) Returns the median of the elements of an $m \times n$ array **A**. This is the value above and below which there are an equal number of values. If **A** has an even number of elements, this is the arithmetic mean of the two central values.

var(**A**) Returns the population variance of the elements of an $m \times n$ array **A** using the formula:

$$\text{var}(\mathbf{A}) = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left|A_{i,j} - \text{mean}(\mathbf{A})\right|^2$$

Var(**A**) Returns the sample variance of the elements of an $m \times n$ array **A** using the formula:

$$\text{var}(\mathbf{A}) = \frac{1}{mn-1}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left|A_{i,j} - \text{mean}(\mathbf{A})\right|^2$$

cvar(**A**, **B**) Returns the covariance of the elements in the $m \times n$ arrays **A** and **B** using the formula:

$$\text{cvar}(\mathbf{A}, \mathbf{B}) = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[A_{i,j} - \text{mean}(\mathbf{A})]\overline{[B_{i,j} - \text{mean}(\mathbf{B})]}$$

where the bar indicates complex conjugation.

stdev(**A**) Returns the population standard deviation (square root of the variance) of the elements of the $m \times n$ array **A**:

$$\text{stdev}(\mathbf{A}) = \sqrt{\text{var}(\mathbf{A})}$$

Stdev(**A**) Returns the sample standard deviation (square root of the sample variance) of the elements of the $m \times n$ array **A**:

$$\text{Stdev}(\mathbf{A}) = \sqrt{\text{Var}(\mathbf{A})}$$

corr(**A**, **B**) Returns a scalar: the correlation coefficient (Pearson's *r*) for the two $m \times n$ arrays **A** and **B**.

Probability distributions

Mathcad includes several functions for working with several common probability densities. These functions fall into three classes:

- **Probability densities:** These give the likelihood that a random variable will take on a particular value.
- **Cumulative probability distributions:** These give the probability that a random variable will take on a value *less than or equal to* a specified value. These are obtained by simply integrating (or summing when appropriate) the corresponding probability density over an appropriate range.
- **Inverse cumulative probability distributions:** These functions take a probability as an argument and return a value such that the probability that a random variable will be *less than or equal to* that value is whatever probability you supplied as an argument.

Probability densities

These functions return the likelihood that a random variable will take on a particular value. The probability density functions are the derivatives of the corresponding cumulative distribution functions discussed in the next section.

---

dbeta(x, s_1, s_2) Returns the probability density for the beta distribution:

$$\frac{\Gamma(s_1 + s_2)}{\Gamma(s_1) \cdot \Gamma(s_2)} \cdot x^{s_1 - 1} \cdot (1 - x)^{s_2 - 1}$$

in which $(s_1, s_2 > 0)$ are the shape parameters. $(0 < x < 1)$.

dbinom(k, n, p) Returns $P(X = k)$ when the random variable X has the binomial

distribution:

$$\frac{n!}{k!(n-k)!}p^k(1-p)^{n-k}$$

in which n and k are integers satisfying $0 \le k \le n$. p satisfies $0 \le p \le 1$.

dcauchy(x, l, s) Returns the probability density for the Cauchy distribution:

$$(\pi s(1 + ((x-l)/s)^2))^{-1}$$

in which l is a location parameter and $s > 0$ is a scale parameter.

dchisq(x, d) Returns the probability density for the chi-squared distribution:

$$\frac{e^{-x/2}}{2\Gamma(d/2)}\left(\frac{x}{2}\right)^{(d/2-1)}$$

in which $d > 0$ is the degrees of freedom and $x > 0$.

dexp(x, r) Returns the probability density for the exponential distribution:

$$re^{-rx}$$

in which $r > 0$ is the rate and $x > 0$.

dF(x, d_1, d_2) Returns the probability density for the F distribution:

$$\frac{d_1^{d_1/2}d_2^{d_2/2}\Gamma((d_1+d_2)/2)}{\Gamma(d_1/2)\Gamma(d_2/2)} \cdot \frac{x^{(d_1-2)/2}}{(d_2+d_1x)^{(d_1+d_2)/2}}$$

in which $(d_1, d_2 > 0)$ are the degrees of freedom and $x > 0$.

dgamma(x, s) Returns the probability density for the gamma distribution:

$$\frac{x^{s-1}e^{-x}}{\Gamma(s)}$$

in which $s > 0$ is the shape parameter and $x \ge 0$.

dgeom(k, p) Returns $P(X = k)$ when the random variable X has the geometric distribution:

$$p(1-p)^k$$

in which $0 < p \le 1$ is the probability of success and k is a nonnegative

integer.

dlnorm(x, μ, σ) Returns the probability density for the lognormal distribution:

$$\frac{1}{\sqrt{2\pi}\sigma x}\exp\left(-\frac{1}{2\sigma^2}(\ln(x)-\mu)^2\right)$$

in which μ is the logmean and σ is the logdeviation. $x > 0$.

dlogis(x, l, s) Returns the probability density for the logistic distribution:

$$\frac{\exp(-(x-l)/s)}{s(1+\exp(-(x-l)/s))^2}$$

in which l is the location parameter and $s > 0$ is the scale parameter.

dnbinom(k, n, p) Returns $P(X = k)$ when the random variable X has the negative binomial distribution:

$$\binom{n+k-1}{k}p^n(1-p)^k$$

in which $0 < p \le 1$ and n and k are integers, $n > 0$ and $k \ge 0$

dnorm(x, μ, σ) Returns the probability density for the normal distribution:

$$\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)$$

in which μ and σ are the mean and standard deviation. $\sigma > 0$.

dpois(k, λ) Returns $P(X = k)$ when the random variable X has the Poisson distribution:

$$\frac{\lambda^k}{k!}e^{-\lambda}$$

in which $\lambda > 0$ and k is a nonnegative integer.

dt(x, d) Returns the probability density for Student's t distribution:

$$\frac{\Gamma((d+1)/2)}{\Gamma(d/2)\sqrt{\pi d}}\left(1+\frac{x^2}{d}\right)^{-(d+1)/2}$$

in which d is the degrees of freedom, $d > 0$ and x is real.

dunif(x, a, b) Returns the probability density for the uniform distribution:

$$\frac{1}{b-a}$$

in which b and a are the endpoints of the interval with $a < b$ and $a \le x \le b$.

dweibull(x, s) Returns the probability density for the Weibull distribution:

$$sx^{s-1}\exp(-x^s)$$

in which $s > 0$ is the shape parameter and $x > 0$.

Cumulative probability distributions

These functions return the probability that a random variable is less than or equal to a specified value. The cumulative probability distribution is simply the probability density function integrated from $-\infty$ to the specified value. For integer random variables, the integral is replaced by a summation over the appropriate range.

The probability density functions corresponding to each of the following cumulative distributions are given in the section "Probability distributions" on page 287.

Figure 14-1 at the end of this section illustrates the relationship between these three functions.

cnorm(x) Returns the cumulative standard normal distribution function. Equivalent to pnorm(x, 0, 1).

pbeta(x, s_1, s_2) Returns the cumulative beta distribution with shape parameters s_1 and s_2. (s_1, $s_2 > 0$).

pbinom(k, n, p) Returns the cumulative binomial distribution for k successes in n trials. n is a positive integer. p is the probability of success, $0 \le p \le 1$.

pcauchy(x, l, s) Returns the cumulative Cauchy distribution with scale parameter s and location parameter l. $s > 0$.

pchisq(x, d) Returns the cumulative chi-squared distribution in which $d > 0$ is the degrees of freedom.

pexp(x, r) Returns the cumulative exponential distribution in which $r > 0$ is the rate.

pF(x, d_1, d_2) Returns the cumulative F distribution in which (d_1, $d_2 > 0$) are the degrees of freedom.

pgamma(x, s) Returns the cumulative gamma distribution in which $s > 0$ is the shape parameter.

pgeom(k, p) Returns the cumulative geometric distribution. p is the probability of success. $0 < p \leq 1$.

plnorm(x, μ, σ) Returns the cumulative lognormal distribution having logmean μ and logdeviation $\sigma > 0$.

plogis(x, l, s) Returns the cumulative logistic distribution. l is the location parameter. $s > 0$ is the scale parameter.

pnbinom(k, n, p) Returns the cumulative negative binomial distribution in which $0 \leq p < 1$. n must be a positive integer.

pnorm(x, μ, σ) Returns the cumulative normal distribution with mean μ and standard deviation σ. $\sigma > 0$.

ppois(k, λ) Returns the cumulative Poisson distribution. $\lambda > 0$.

pt(x, d) Returns the cumulative Student's t distribution. d is the degrees of freedom. $d > 0$.

punif(x, a, b) Returns the cumulative uniform distribution. b and a are the endpoints of the interval. $a < b$.

pweibull(x, s) Returns the cumulative Weibull distribution. $s > 0$.

Inverse cumulative probability distributions

These functions take a probability p as an argument and return the value of x such that $P(X \leq x) = p$.

The probability density functions corresponding to each of the following inverse cumulative distributions are given in the section “Probability distributions” on page 287.

qbeta(p, s_1, s_2) Returns the inverse beta distribution with shape parameters s_1 and s_2. $0 \leq p \leq 1$ and $s_1, s_2 > 0$.

qbinom(p, n, r) Returns the number of successes in n trials of the Bernoulli process such that the probability of at most that number of successes is p. r is the probability of success on a single trial. $0 \leq r \leq 1$ and $0 \leq p \leq 1$. n must be an integer greater than zero.

qcauchy(p, l, s) Returns the inverse Cauchy distribution with scale parameter s and

	location parameter *l*. $s > 0$. $0 < p < 1$.
qchisq(*p*, *d*)	Returns the inverse chi-squared distribution in which $d > 0$ is the degrees of freedom. $0 \le p < 1$.
qexp(*p*, *r*)	Returns the inverse exponential distribution in which $r > 0$ is the rate. $0 \le p < 1$.
qF(*p*, d_1, d_2)	Returns the inverse *F* distribution in which $(d_1, d_2 > 0)$ are the degrees of freedom. $0 \le p < 1$.
qgamma(*p*, *s*)	Returns the inverse gamma distribution in which $s > 0$ is the shape parameter. $0 \le p < 1$.
qgeom(*p*, *r*)	Returns the inverse geometric distribution. *r* is the probability of success on a single trial. $0 < p < 1$ and $0 < r < 1$.
qlnorm(*p*, μ, σ)	Returns the inverse lognormal distribution having logmean μ and logdeviation $\sigma > 0$. $0 \le p < 1$.
qlogis(*p*, *l*, *s*)	Returns the inverse logistic distribution. *l* is the location parameter. $s > 0$ is the scale parameter. $0 < p < 1$.
qnbinom(*p*, *n*, *r*)	Returns the inverse negative binomial distribution with size *n* and probability of success *r*. $0 < r \le 1$ and $0 \le p \le 1$.
qnorm(*p*, μ, σ)	Returns the inverse normal distribution with mean μ and standard deviation σ. $\sigma > 0$ and $0 < p < 1$.
qpois(*p*, λ)	Returns the inverse Poisson distribution. $\lambda > 0$ and $0 \le p \le 1$.
qt(*p*, *d*)	Returns the inverse Student's *t* distribution. *d* is the degrees of freedom. $d > 0$ and $0 < p < 1$.
qunif(*p*, *a*, *b*)	Returns the inverse uniform distribution. *b* and *a* are the endpoints of the interval. $a < b$ and $0 \le p \le 1$.
qweibull(*p*, *s*)	Returns the inverse Weibull distribution. $s > 0$ and $0 < p < 1$.

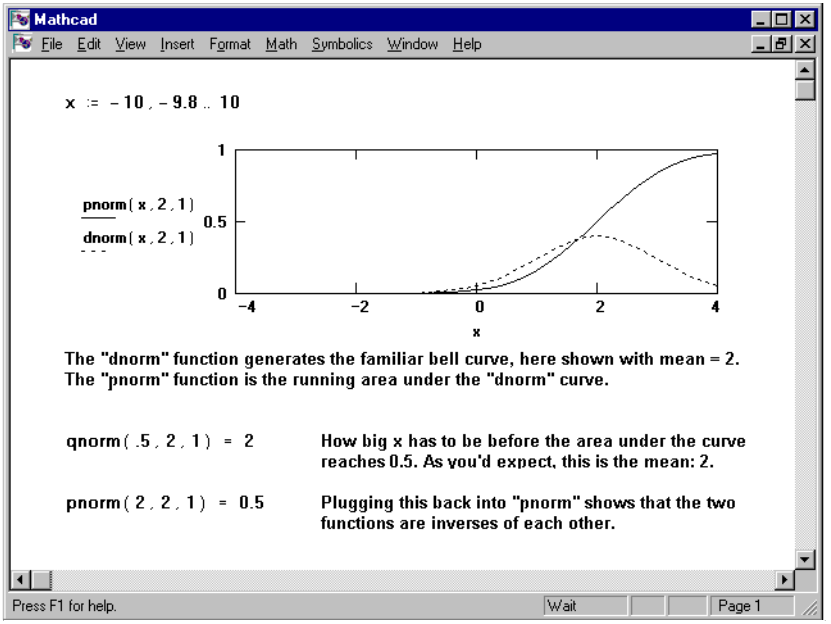

Figure 14-1: Relationship between probability densities, cumulative distributions, and their inverses.

Histogram function

Mathcad includes a function, *hist*, for computing frequency distributions for histograms:

hist(**int**, **A**)	Returns a vector representing the frequencies with which values in **A** fall in the intervals represented by the **int** vector. The elements in both **int** and **A** must be real. In addition, the elements of **int** must be in ascending order. The resulting vector is one element shorter than **int**.

Mathcad interprets **int** as a set of points defining a sequence of intervals in a histogram. The values in **int** must should be in ascending order. The result of this function is a vector **f**, in which f_i is the number of values in **A** satisfying the condition:

$$int_i \le value \le int_{i+1}$$

Mathcad ignores data points less than the first value in **int** or greater than the last value in **int**. Figure 14-2 shows how to use histograms in Mathcad.

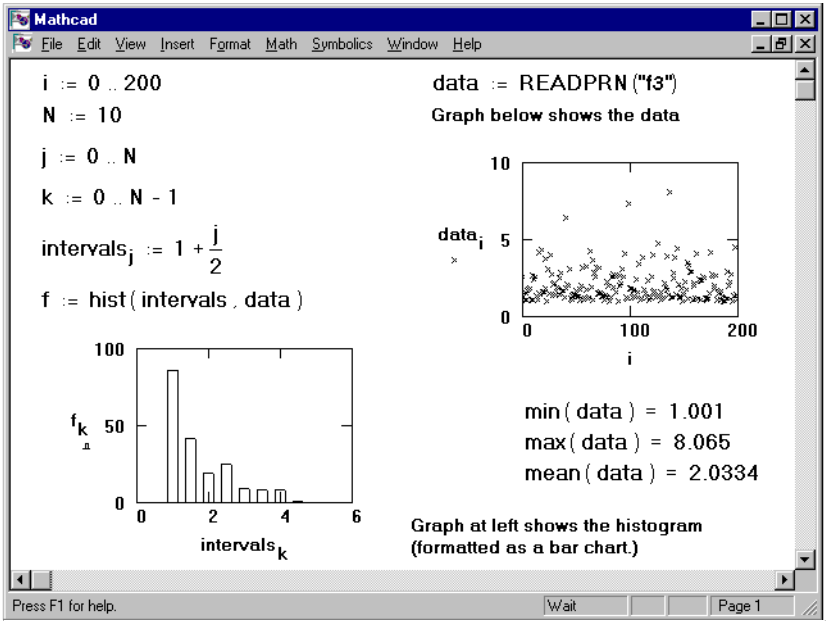

Figure 14-2: A histogram.

Random numbers

Mathcad comes with a number of functions for generating random numbers having a variety of probability distributions. The functional forms of the distributions associated with the following functions are given in the section "Probability distributions" on page 287.

rbeta(m, s_1, s_2)	Returns a vector of m random numbers having the beta distribution. $s_1, s_2 > 0$ are the shape parameters.
rbinom(m, n, p)	Returns a vector of m random numbers having the binomial distribution. $0 \le p \le 1$. n is an integer satisfying $n > 0$.
rcauchy(m, l, s)	Returns a vector of m random numbers having the Cauchy distribution. $s > 0$ is the scale parameter. l is the location parameter.
rchisq(m, d)	Returns a vector of m random numbers having the chi-squared distribution. $d > 0$ is the degrees of freedom.

rexp(m, r)	Returns a vector of m random numbers having the exponential distribution. $r > 0$ is the rate.
rF(m, d_1, d_2)	Returns a vector of m random numbers having the F distribution. d_1, $d_2 > 0$ are the degrees of freedom.
rgamma(m, s)	Returns a vector of m random numbers having the gamma distribution. $s > 0$ is the shape parameter.
rgeom(m, p)	Returns a vector of m random numbers having the geometric distribution. $0 < p \le 1$.
rlnorm(m, μ, σ)	Returns a vector of m random numbers having the lognormal distribution in which μ is the logmean and $\sigma > 0$ is the logdeviation.
rlogis(m, l, s)	Returns a vector of m random numbers having the logistic distribution in which l is the location parameter and $s > 0$ is the scale parameter.
rnbinom(m, n, p)	Returns a vector of m random numbers having the negative binomial distribution. $0 < p \le 1$. n is an integer satisfying $n > 0$.
rnorm(m, μ, σ)	Returns a vector of m random numbers having the normal distribution with mean μ and standard deviation $\sigma > 0$.
rpois(m, λ)	Returns a vector of m random numbers having the Poisson distribution. $\lambda > 0$.
rt(m, d)	Returns a vector of m random numbers having Student's t distribution. $d > 0$.
runif(m, a, b)	Returns a vector of m random numbers having the uniform distribution in which b and a are the endpoints of the interval and $a < b$.
rnd(x)	Returns a uniformly distributed random number between 0 and x. Equivalent to $\mathrm{runif}(1, 0, x)$.
rweibull(m, s)	Returns a vector of m random numbers having the Weibull distribution in which $s > 0$ is the shape parameter.

Each time you recalculate an equation containing one of these functions, Mathcad generates new random numbers. To force Mathcad to generate new random numbers, click on the equation containing the function and choose **Calculate** from the **Math** menu. Figure 14-3 shows an example of how to use Mathcad's random number generator. Figure 14-4 shows how to generate a large vector of random numbers having a specified distribution.

These functions have a “seed value” associated with them. Each time you reset the seed, Mathcad generates new random numbers based on that seed. A given seed value will always generate the same sequence of random numbers. Choosing **Calculate** from the **Math** menu advances Mathcad along this random number sequence. Changing the seed value, however, advances Mathcad along an altogether different random number sequence.

To change the seed value, choose **Options** from the **Math** menu and change the value of “seed” on the Built-In Variables tab. Be sure to supply an integer.

To reset Mathcad's random number generator without changing the seed value, choose **Options** from the **Math** menu, click on the Built-In Variables tab, and click “OK” to accept the current seed. Then click on the equation containing the random number generating function and choose **Calculate** from the **Math** menu. Since the randomizer has been reset, Mathcad generates the same random numbers it would generate if you restarted Mathcad.

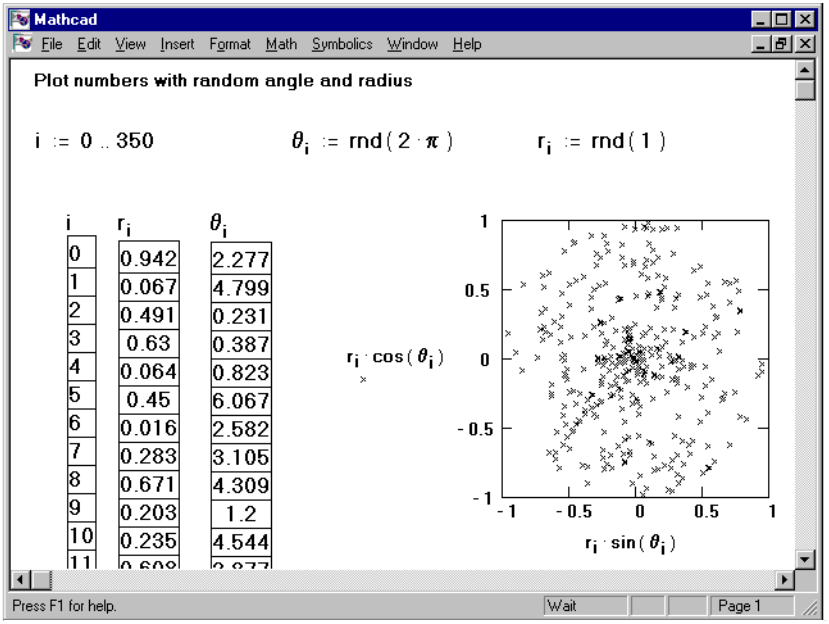

Figure 14-3: Uniformly distributed random numbers. Since the random number generator generates different numbers every time, it's unlikely that you'll be able to reproduce this example exactly as you see it here.

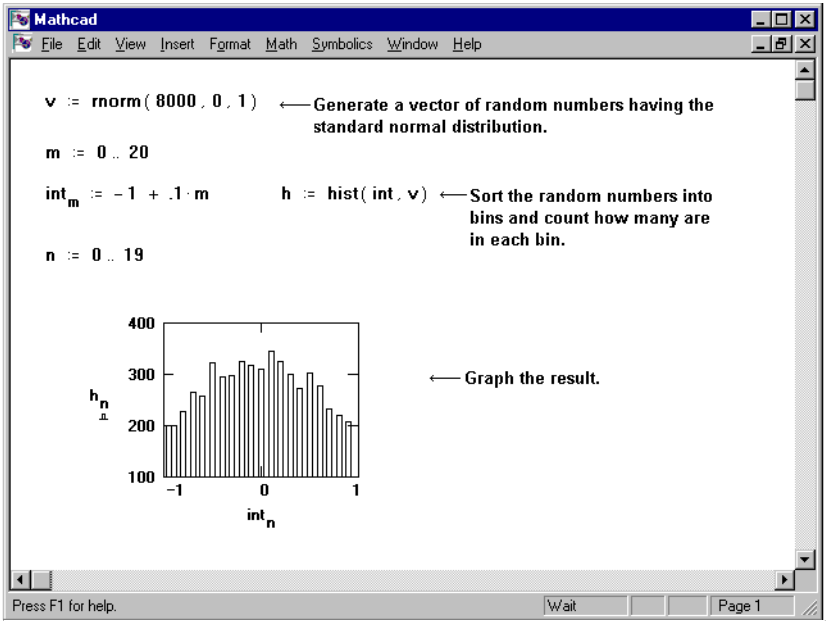

Figure 14-4: A vector of normally distributed random numbers. Since the random numbers are different every time, it's unlikely that you'll be able to reproduce this example exactly as you see it here.

If you want to check a test case several times with the same random numbers, reset the random number generator between calculations as described above.

To see a new set of random numbers, change the seed value as described above. This causes Mathcad to generate a different set of random numbers from what you see when you start Mathcad. Each time you want to reset Mathcad to regenerate these random numbers, reset the seed as described above. To see a different set of random numbers, change the seed value.

Interpolation and prediction functions

Interpolation involves using existing data points to predict values between these data points. Mathcad allows you to either connect the data points with straight lines (linear interpolation) or to connect them with sections of a cubic polynomial (cubic spline interpolation).

Unlike the regression functions discussed in the next section, these interpolation functions return a curve which must pass through the points you specify. Because of this, the resulting function is very sensitive to spurious data points. If your data is noisy, you should consider using the regression functions instead.

Linear prediction involves using existing data values to predict values beyond the existing ones. Mathcad provides a function which allows you to predict future data points based on past data points.

Whenever you use arrays in any of the functions described in this section, be sure that every element in the array contains a data value. Since every element in a array must have a value, Mathcad assigns 0 to any elements you have not explicitly assigned.

Linear interpolation

In linear interpolation, Mathcad connects the existing data points with straight lines. This is accomplished by the *linterp* function described below.

---

linterp(**vx**, **vy**, *x*) Uses the data vectors **vx** and **vy** to return a linearly interpolated *y* value corresponding to the third argument *x*. The arguments **vx** and **vy** must be vectors of the same length. The vector **vx** must contain real values in ascending order.

To find the interpolated value for a particular *x*, Mathcad finds the two points between which the value falls and returns the corresponding *y* value on the straight line between the two points.

For *x* values before the first point in **vx**, Mathcad extrapolates the straight line between the first two data points. For *x* values beyond the last point in **vx**, Mathcad extrapolates the straight line between the last two data points.

For best results, the value of *x* should be between the largest and smallest values in the vector **vx**. The *linterp* function is intended for interpolation, not extrapolation. Consequently, computed values for *x* outside this range are unlikely to be useful. Figure 14-5 shows some examples of linear interpolation.

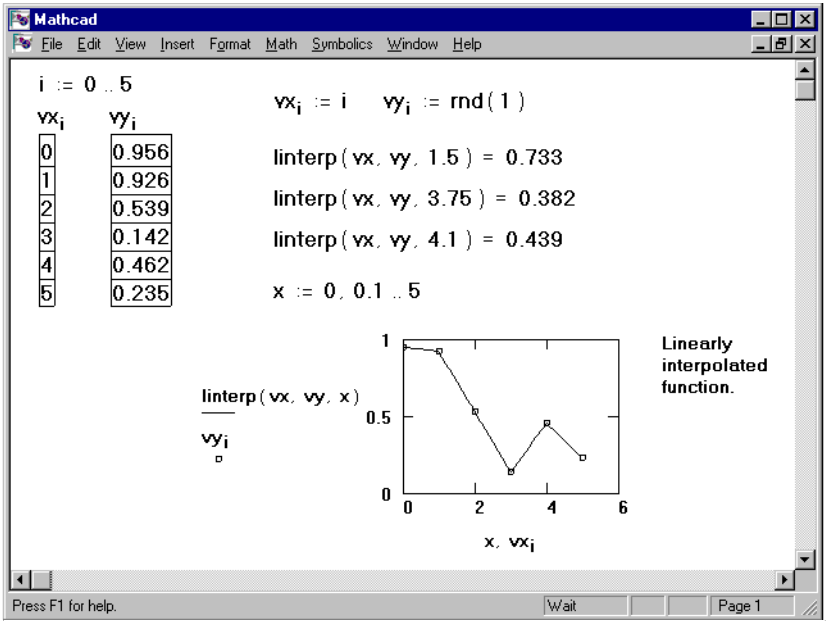

Figure 14-5: Examples of linear interpolation. Since the random number generator gives different numbers every time, you may not be able to recreate this example exactly as you see it.

Cubic spline interpolation

Cubic spline interpolation lets you pass a curve through a set of points in such a way that the first and second derivatives of the curve are continuous across each point. This curve is assembled by taking three adjacent points and constructing a cubic polynomial passing through those points. These cubic polynomials are then strung together to form the completed curve.

To fit a cubic spline curve through a set of points:

- Create the vectors **vx** and **vy** containing the *x* and *y* coordinates through which you want the cubic spline to pass. The elements of **vx** should be in ascending order. (Although we use the names **vx**, **vy** and **vs**, there is nothing special about these variable names; you can use whatever names you prefer in your own work.)
- Generate the vector **vs** := cspline(**vx**, **vy**). The vector **vs** is a vector of intermediate results designed to be used with *interp*.
- To evaluate the cubic spline at an arbitrary point, say *x0*, evaluate (interp(**vs**, **vx**, **vy**, *x*0) where **vs**, **vx** and **vy** are the vectors described earlier.

Note that you could have accomplished the same task by evaluating:

interp(cspline(**vx**, **vy**), **vx**, **vy**, *x*0)

As a practical matter, though, you'll probably be evaluating *interp* for many different points. Since the call to *cspline* can be time-consuming, and since the result won't change from one point to the next, it makes sense to do it once and just reuse the result as described above.

Figure 14-6 shows how to compute the spline curve for the example in Figure 14-5.

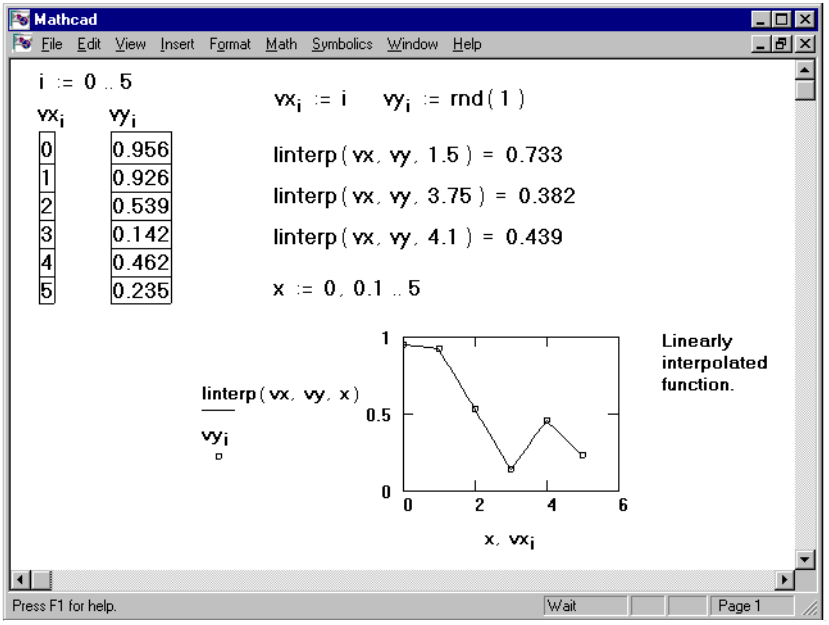

Figure 14-6: Spline curve for the points stored in x and y. Since the random number generator gives different numbers every time, you may not be able to recreate this example exactly as you see it.

Here is a description of the steps involved in the example in Figure 14-6:

- The equation with the *cspline* function computes the array **vs** containing, among other things, the second derivatives for the spline curve used to fit the points in **vx** and **vy**.
- Once the **vs** array is computed, the *interp* function computes the interpolated values.

Note that the **vs** array needs to be computed only once, even for multiple interpolations. Since the spline calculations that lead to **vs** are time-consuming, it is more efficient to store these intermediate results as a vector than it is to recalculate them as needed.

In addition to *cspline*, Mathcad comes with two other cubic spline functions, as described below:

cspline(**vx**, **vy**) pspline(**vx**, **vy**) lspline(**vx**, **vy**)	These all return a vector of intermediate results, **vs**, which is used in the *interp* function described below. Arguments **vx** and **vy** must be real vectors of the same length. The values in **vx** must be real and in ascending order.

These three functions differ only in the boundary conditions:

- The *lspline* function generates a spline curve that approaches a straight line at the endpoints.
- The *pspline* function generates a spline curve that approaches a parabola at the endpoints.
- The *cspline* function generates a spline curve that can be fully cubic at the endpoints.

interp(**vs**, **vx**, **vy**, *x*)	Returns the interpolated *y* value corresponding to the argument *x*. The vector **vs** is a vector of intermediate results obtained by evaluating *lspline*, *pspline*, or *cspline* using the data vectors **vx** and **vy**.

To find the interpolated value for a particular *x*, Mathcad finds the two points between which it falls. It then returns the *y* value on the cubic section enclosed by these two points. For *x* values before the first point in **vx**, Mathcad extrapolates the cubic section connecting the first two points of **vx**. Similarly, for *x* values beyond the last point in **vx**, Mathcad extrapolates the cubic section connecting the last two points of **vx**.

For best results, do not use the *interp* function on values of *x* far from the fitted points. Splines are intended for interpolation, not extrapolation. Consequently, computed values for such *x* values are unlikely to be useful.

Interpolating a vector of points

You can use the vectorize operator to return a whole vector of interpolated values corresponding to a vector of data points. This works with both *interp* and *linterp*.

Figure 14-7 shows how to perform this operation. To apply the vectorize operator to the function, click on the function name and press [**Space**] until the function is between the two editing lines. Then press [**Ctrl**]**-** (hold down the [**Ctrl**] key and press the minus sign).

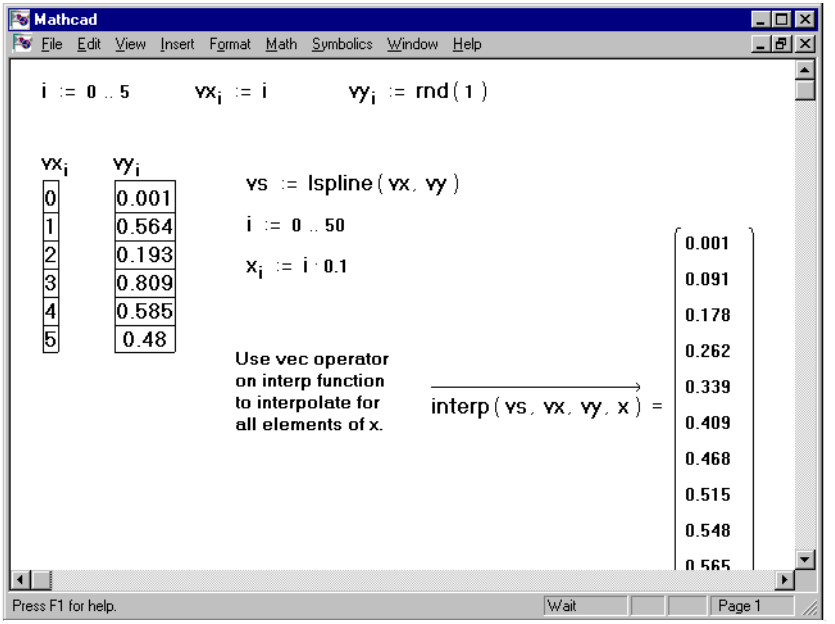

Figure 14-7: Interpolating a vector of points. Note that since these are random numbers, it's unlikely you'll be able to reproduce this example exactly as you see here.

Multivariate cubic spline interpolation

Mathcad handles two dimensional cubic spline interpolation in much the same way as the one-dimensional case discussed earlier. Instead of passing a curve through a set of points in such a way that the first and second derivatives of the curve are continuous across each point, Mathcad passes a surface through a grid of points. This surface corresponds to a cubic polynomial in *x* and *y* in which the first and second partial derivatives are continuous in the corresponding direction across each grid point.

The first step in two-dimensional spline interpolation is exactly the same as that in the one-dimensional case: specify the points through which the surface is to pass. The procedure, however, is more complicated because you now have to specify a grid of points:

- Create the $n \times 2$ matrix **Mxy** whose elements, $Mxy_{i,0}$ and $Mxy_{i,1}$, specify the *x* and *y* coordinates along the *diagonal* of a rectangular grid. This matrix plays the exactly the same role as **vx** in the one-dimensional case described earlier. Since these points describe a diagonal, the elements in each column of **Mxy** be in ascending order ($Mxy_{i,k} < Mxy_{j,k}$ whenever $i < j$).
- Create the $n \times n$ matrix **Mz** whose *ij*th element is the *z* coordinate corresponding to the point $x = Mxy_{i,0}$ and $y = Mxy_{j,1}$. This plays exactly the same role as **vy** in the one-dimensional case described earlier.

- Generate the vector **vs** := cspline(**Mxy, Mz**) . The vector **vs** is a vector of intermediate results designed to be used with *interp*.
- To evaluate the cubic spline at an arbitrary point, say $(x0, y0)$, evaluate

$$\text{interp}\left(\mathbf{vs}, \mathbf{Mxy}, \mathbf{Mz}, \begin{bmatrix} x0 \\ y0 \end{bmatrix}\right)$$

  where **vs**, **Mxy** and **Mz** are the arrays described earlier. The result is the value of the interpolating surface corresponding to the arbitrary point $(x0, y0)$.

Note that you could have accomplished exactly the same task by evaluating:

$$\text{interp}\left(\text{cspline}(\mathbf{Mxy}, \mathbf{Mz}), \mathbf{Mxy}, \mathbf{Mz}, \begin{bmatrix} x0 \\ y0 \end{bmatrix}\right)$$

As a practical matter though, you'll probably be evaluating *interp* for many different points. Since the call to *cspline* can be time-consuming, and since the result won't change from one point to the next, it makes sense to call it once and just keep re-using the result as described above.

In addition to *cspline*, Mathcad comes with two other cubic spline functions. The three spline functions are:

cspline(**Mxy**, **Mz**) pspline(**Mxy**, **Mz**) lspline(**Mxy**, **Mz**)	These all return a vector of intermediate results which we'll call **vs**. This vector, **vs**, is used in the *interp* function described below. **Mxy** is an $n \times 2$ matrix whose elements $Mxy_{i,0}$ and $Mxy_{i,1}$ specify points on the diagonal of an $n \times n$ grid. The *ij*th element of the $n \times n$ matrix **Mz** specifies the value of the interpolating surface at $(Mxy_{i,0}, Mxy_{j,1})$.

These three functions differ only in the boundary conditions:

- The *lspline* function generates a spline curve that approaches a plane along the edges.
- The *pspline* function generates a spline curve that approaches a second degree polynomial in *x* and *y* along the edges.
- The *cspline* function generates a spline curve that approaches a third degree polynomial in *x* and *y* along the edges.

interp(**vs**, **Mxy**, **Mz**, **v**)	Returns the interpolated *z* value corresponding to the point $x = v_0$ and $y = v_1$. The vector **vs** comes from evaluating *lspline*, *pspline*, or *cspline* using the data matrices **Mxy** and **Mz**.

For best results, do not use the *interp* function on values of *x* and *y* far from the grid points. Splines are intended for interpolation, not extrapolation. Consequently, computed values for such *x* and *y* values are unlikely to be useful.

Linear prediction

The functions described so far in this section allow you to find data points lying between existing data points. However, you may need to find data points that lie beyond your existing ones. Mathcad provides the function *predict* which uses some of your existing data to predict data points lying beyond the existing ones. This function uses a linear prediction algorithm which is useful when your data is smooth and oscillatory, though not necessarily periodic. Linear prediction can be seen as a kind of extrapolation method but should not be confused with linear or polynomial extrapolation.

predict(**v**, *m*, *n*)	Returns *n* predicted values based on *m* consecutive values from the data vector **v**. Elements in **v** should represent samples taken at equal intervals.

The *predict* function uses the last *m* of the original data values to compute prediction coefficients. Once it has these coefficients, it uses the last *m* points to predict the coordinates of the $(m + 1)$st point, in effect creating a moving window *m* points wide.

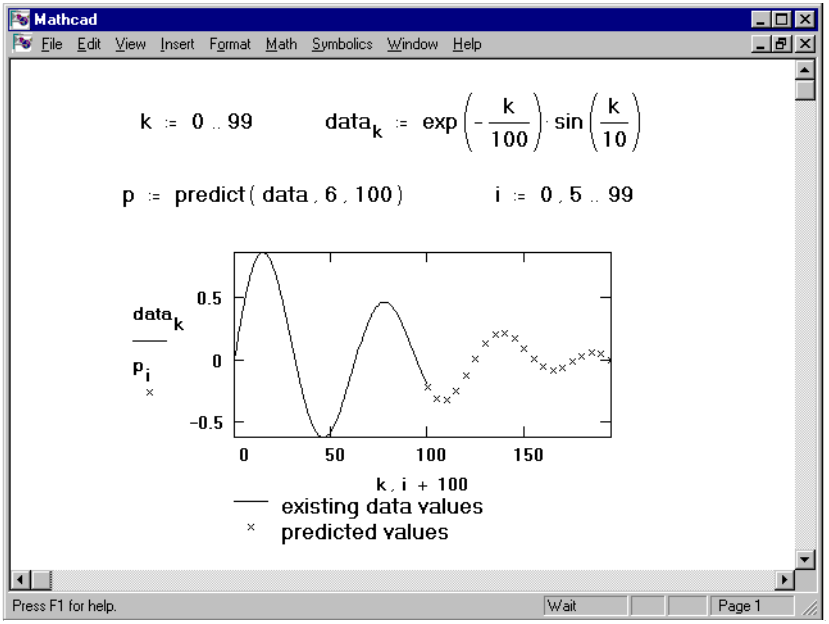

Figure 14-8: Using the predict function to find future data values.

Regression functions

Mathcad includes a number of functions for performing regression. Typically, these functions generate a curve or surface of a specified type which in some sense minimizes the error between itself and the data you supply. The functions differ primarily in the type of curve or surface they use to fit the data.

Unlike the interpolation functions discussed in the previous section, these functions do not require that the fitted curve or surface pass through the data points you supply. The regression functions in this section are therefore far less sensitive to spurious data than the interpolation functions.

Unlike the smoothing functions in the next section, the end result of a regression is an actual function, one that can be evaluated at points in between the points you supply.

Whenever you use arrays in any of the functions described in this section, be sure that every element in the array contains a data value. Since every element in a array must have a value, Mathcad assigns 0 to any elements you have not explicitly assigned.

Linear regression

These functions return the slope and intercept of the line that best fits your data in a least-squares sense. If you place your *x* values in the vector **vx** and your sampled *y* values in **vy**, that line is given by:

$$y = \text{slope}(\mathbf{vx}, \mathbf{vy}) \cdot x + \text{intercept}(\mathbf{vx}, \mathbf{vy})$$

Figure 14-9 shows how you can use these functions to fit a line through a set of data points.

slope(**vx**, **vy**)	Returns a scalar: the slope of the least-squares regression line for the data points in **vx** and **vy**.
intercept(**vx**, **vy**)	Returns a scalar: the *y*-intercept of the least-squares regression line for the data points in **vx** and **vy**.

These functions are useful not only when your data is inherently linear but when it is exponential as well. More specifically, if your *x* and *y* are related by:

$$y = Ae^{kx}$$

You can apply these functions to the log of your data values and make use of the fact that:

$$\log(y) = \log(A) + kx$$

In which case:

$$A = \exp(\text{intercept}(\mathbf{vx}, \mathbf{vy})) \text{ and } k = \text{slope}(\mathbf{vx}, \mathbf{vy})$$

The resulting fit weighs the errors differently from a least-squares exponential fit but is usually a good approximation.

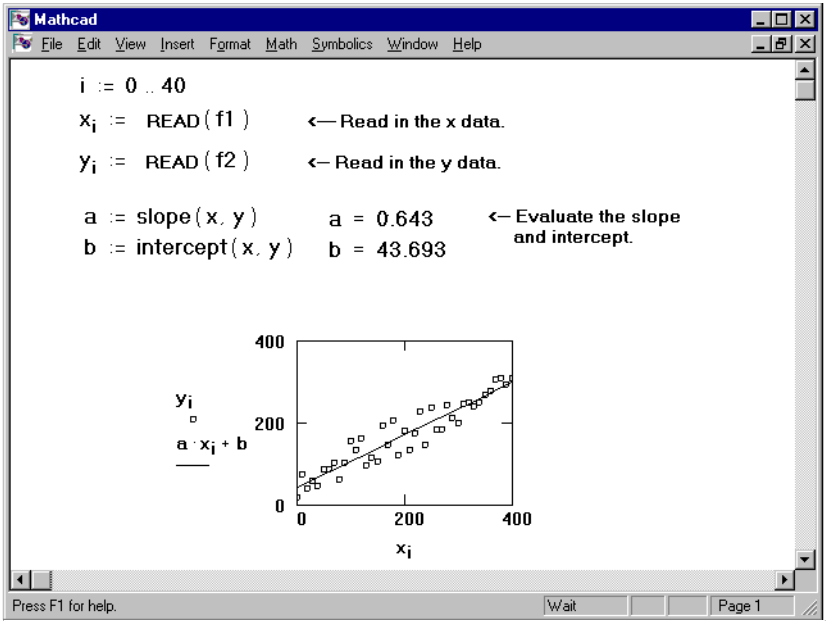

Figure 14-9: Using slope and intercept for linear regression.

Polynomial regression

These functions are useful when you have set of measured *y* values corresponding to *x* values and you want to fit a polynomial through those *y* values.

Use *regress* when you want to use a single polynomial to fit all your data values. The *regress* function lets you fit a polynomial of any order. However as a practical matter, you would rarely need to go beyond $n = 4$.

Since *regress* tries to accommodate all your data points using a single polynomial, it will not work well when your data does not behave like a single polynomial. For example, suppose you expect your y_i to be linear from x_1 to x_{10} and to behave like a cubic equation from x_{11} to x_{20}. If you use *regress* with $n = 3$ (a cubic), you may get a good fit for the second half but a terrible fit for the first half.

The *loess* function, available in Mathcad Professional, alleviates these kinds of problems by performing a more localized regression. Instead of generating a single polynomial the way *regress* does, *loess* generates a different second order polynomial depending on where you are on the curve. It does this by examining the data in a small neighborhood of the point you're interested in. The argument *span* controls the size of this neighborhood. As *span* gets larger, *loess* becomes equivalent to *regress* with $n = 2$. A good default value is $span = 0.75$.

Figure 14-10 shows how *span* affects the fit generated by the *loess* function. Note how a smaller value of *span* makes the fitted curve track fluctuations in data more effectively. A larger value of *span* tends to smear out fluctuations in data and therefore generates a smoother fit.

	regress(**vx**, **vy**, *n*)	A vector required by the *interp* function to find the *n*th order polynomial that best fits data vectors **vx** and **vy**. **vx** is an *m* element vector containing *x* coordinates. **vy** is an *m* element vector containing the *y* coordinates corresponding to the *m* points specified in **vx**.
Pro	loess(**vx**, **vy**, *span*)	A vector required by the *interp* function to find the set of second order polynomials that best fit particular neighborhoods of data points specified in vectors **vx** and **vy**. **vx** is an *m* element vector containing *x* coordinates. **vy** is an *m* element vector containing the *y* coordinates corresponding to the *m* points specified in **vx**. The argument *span*, $span > 0$, specifies how large a neighborhood *loess* will consider in performing this local regression.
	interp(**vs**, **vx**, **vy**, *x*)	Returns the interpolated *y* value corresponding to the *x*. The vector **vs** comes from evaluating *loess* or *regress* using the data matrices **vx** and **vy**.

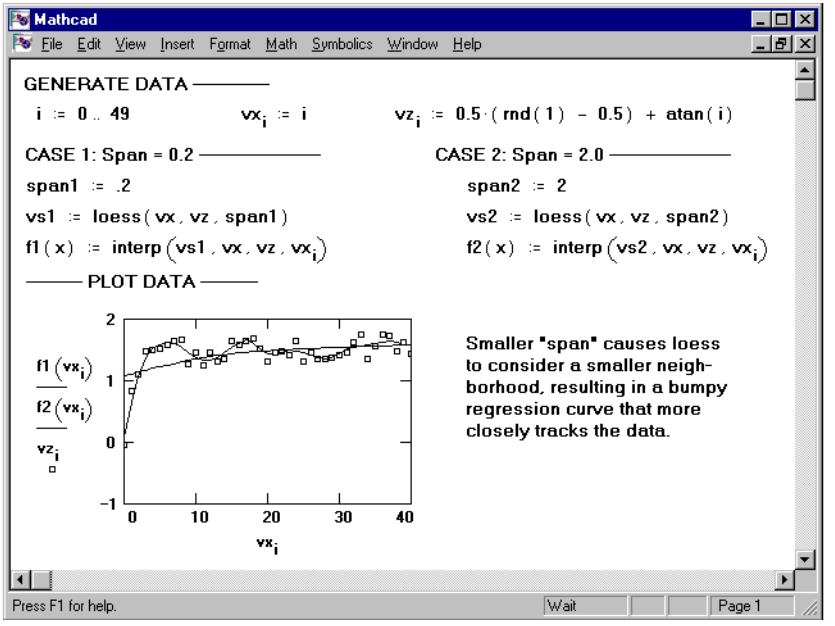

Figure 14-10: Effect of different spans on the loess function. Note that since these are random numbers, it's unlikely you'll be able to reproduce this example exactly as you see here.

Multivariate polynomial regression

The *loess* and *regress* functions discussed in the previous section are also useful when you have set of measured *z* values corresponding to *x* and *y* values and you want to fit a polynomial surface through those *z* values.

The properties of these functions are described in the previous section. When using these functions to fit *z* values corresponding to two independent variables *x* and *y*, the meanings of the arguments must be generalized. Specifically:

- The argument **vx** which was an *m*-element vector of *x* values becomes an *m*-row and 2 column array, **Mxy**. Each row of **Mxy** contains an *x* in the first column and a corresponding *y* value in the second column.
- The argument *x* for the *interp* function becomes a 2-element vector **v** whose elements are the *x* and *y* values at which you want to evaluate the polynomial surface representing the best fit to the data points in **Mxy** and **vz**.

	regress(**Mxy**, **vz**, *k*)	A vector required by the *interp* function to find the *n*th order polynomial that best fits data arrays **Mxy** and **vz**. **Mxy** is an $m \times 2$ matrix containing *x*-*y* coordinates. **vz** is an *m* element vector containing the *z* coordinates corresponding to the *m* points specified in **Mxy**.
Pro	loess(**Mxy, vz**, *span*)	A vector required by the *interp* function to find the set of second order polynomials that best fit particular neighborhoods of data points specified in arrays **Mxy** and **vz**. **Mxy** is an $m \times 2$ matrix containing *x*-*y* coordinates. **vz** is an *m* element vector containing the *z* coordinates corresponding to the *m* points specified in **Mxy**. The argument *span*, $span > 0$, specifies how large a neighborhood *loess* will consider in performing this local regression.)
	interp(**vs**, **Mxy**, **vz**, **v**)	Returns the interpolated *z* value corresponding to the point $x = v_0$ and $y = v_1$. The vector **vs** comes from evaluating *loess* or *regress* using the data matrices **Mxy** and **vz**.

You can add independent variables by simply adding columns to the **Mxy** array. You would then add a corresponding number of rows to the vector **v** that you pass to the *interp* function. The *regress* function can have as many independent variables as you want. However, *regress* will calculate more slowly and require more memory when the number of independent variables and the degree are greater than four. The *loess* function is restricted to at most four independent variables.

Keep in mind that for *regress*, the number of data values, *m* must satisfy

$$m > \binom{n+k-1}{k} \cdot \frac{n+k}{n}$$

where *n* is the number of independent variables (hence the number of columns in **Mxy**), *k* is the degree of the desired polynomial, and *m* is the number of data values (hence the number of rows in **vz**). For example, if you have five explanatory variables and a fourth degree polynomial, you will need more than 126 observations.

Generalized regression

Unfortunately, not all data sets can be modeled by lines or polynomials. There are times when you need to model your data with a linear combination of arbitrary functions, none of which represent terms of a polynomial. For example, in a Fourier series you try to approximate data using a linear combination of complex exponentials. Or you may believe your data can be modeled by a weighted combination of Legendre polynomials, but you just don't know what weights to assign.

The *linfit* function is designed to solve these kinds of problems. If you believe your data could be modeled by a linear combination of arbitrary functions:

$$y = a_0 \cdot f_0(x) + a_1 \cdot f_1(x) + \ldots + a_n \cdot f_n(x)$$

you should use *linfit* to evaluate the a_i. Figure 14-11 shows an example in which a linear combination of three functions: x, x^2, and $(x+1)^{-1}$ is used to model some data.

There are times however when the flexibility of *linfit* is still not enough. Your data may have to be modeled not by a linear combination of data but by some function whose parameters must be chosen. For example, if your data can be modeled by the sum:

$$f(x) = a_1 \cdot \sin(2x) + a_2 \cdot \tanh(3x)$$

and all you need to do is solve for the unknown weights a_1 and a_2, then you have a *linfit* type of problem.

By contrast, if instead your data is to be modeled by the sum:

$$f(x) = 2 \cdot \sin(a_1 x) + 3 \cdot \tanh(a_2 x)$$

and you now have to solve for the unknown parameters a_1 and a_2, you would have a *genfit* problem.

Anything you can do with *linfit* you can also do, albeit less conveniently, with *genfit*. The difference between these two functions is the difference between solving a system of linear equations and solving a system of nonlinear equations. The former is easily done using the methods of linear algebra. The latter is far more difficult and generally must be solved by iteration. This explains why *genfit* needs a vector of guess values as an argument and *linfit* does not.

Figure 14-12 shows an example in which *genfit* is used to find the exponent that best fits a set of data.

linfit(**vx**, **vy**, **F**)	Returns a vector containing the coefficients used to create a linear combination of the functions in **F** which best approximates the data in vectors **vx** and **vy**. **F** is a function which returns a vector consisting of the functions to be linearly combined.
genfit(**vx**, **vy**, **vg**, **F**)	A vector containing the parameters that make a function *f* of *x* and *n* parameters $u_0, u_1, \ldots, u_n$ best approximate the data in **vx** and **vy**. **F** is a function that returns an $n + 1$ element vector containing *f* and its partial derivatives with respect to its *n* parameters. **vg** is an *n*-element vector of guess values for the *n* parameters.

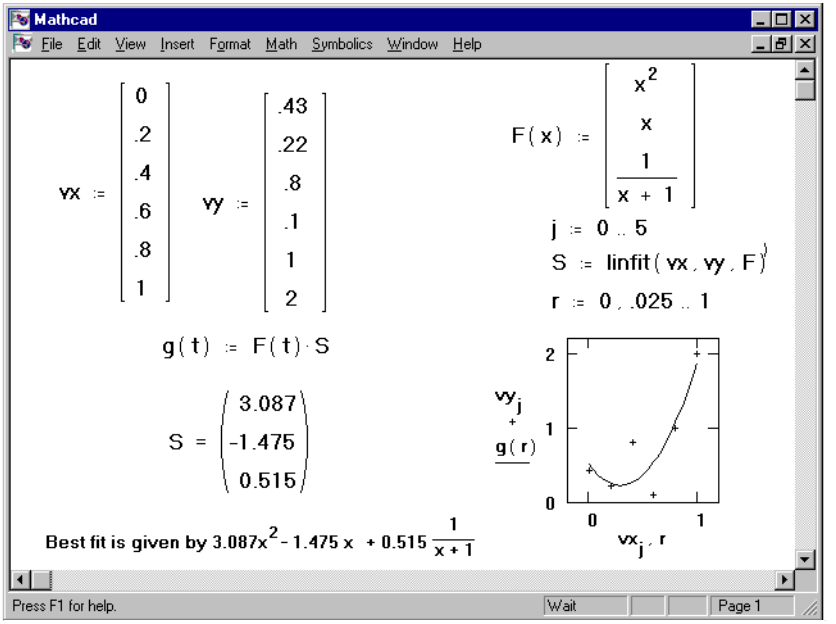

Figure 14-11: Using linfit to find coefficients for a linear combination of functions that best fits the data.

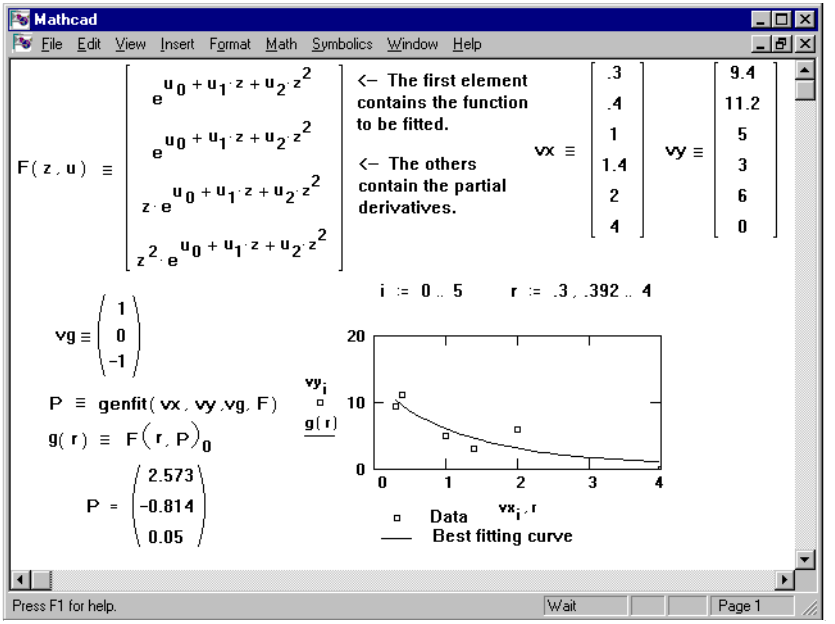

Figure 14-12: Using genfit for finding the parameters of a function so that it best fits the data.

Smoothing functions

Smoothing involves taking a set of *y* (and possibly *x*) values and returning a new set of *y* values that is smoother than the original set. Unlike the regression and interpolation functions discussed earlier, smoothing results in a new set of *y* values, not a function that can be evaluated between the data points you specify. Thus, if you are interested in *y* values *between* the *y* values you specify, you should use a regression or interpolation function.

Whenever you use vectors in any of the functions described in this section, be sure that every element in the vector contains a data value. Since every element in a vector must have a value, Mathcad assigns 0 to any elements you have not explicitly assigned.

The *medsmooth* function is the most robust of the three since it is least likely to be affected by spurious data points. This function uses a running median smoother, computes the residuals, smooths the residuals the same way, and adds these two smoothed vectors together. The details are as follows:

- Evaluation of medsmooth(**vy**, *n*) begins with the running median of the input vector **vy**. We'll call this **vy′** . The *i*th element is given by:

$$vy'_i = \text{median}(vy_{i-(n-1/2)}, \ldots, vy_i, \ldots, vy_{i+(n-1/2)})$$

- It then evaluates the residuals: $\mathbf{vr} = \mathbf{vy} - \mathbf{vy'}$.
- The residual vector, **vr**, is smoothed using the same procedure described in step 1. This creates a smoothed residual vector, $\mathbf{vr'}$.
- The *medsmooth* function returns the sum of these two smoothed vectors: $\text{medsmooth}(\mathbf{vy}, n) = \mathbf{vy'} + \mathbf{vr'}$.

Note that *medsmooth* will leave the first and last $(n-1)/2$ points unchanged. In practice, the length of the smoothing window, *n*, should be small compared to the length of the data set.

The *ksmooth* function in Mathcad Professional uses a Gaussian kernel to compute local weighted averages of the input vector **vy**. This smoother is most useful when your data lies along a band of relatively constant width. If your data lies scattered along a band whose width fluctuates considerably, you should use an adaptive smoother like *supsmooth*, also available in Mathcad Professional.

For each vy_i in the *n*-element vector **vy**, the *ksmooth* function returns a new vy'_i given by:

$$vy'_i = \frac{\sum_{j=1}^{n} K\left(\frac{vx_i - vx_j}{b}\right) vy_j}{\sum_{j=1}^{n} K\left(\frac{vx_i - vx_j}{b}\right)}$$

where:

$$K(t) = \frac{1}{\sqrt{2\pi} \cdot (0.37)} \cdot \exp\left(-\frac{t^2}{2 \cdot (0.37)^2}\right)$$

and *b* is a bandwidth which you supply to the *ksmooth* function. The bandwidth is usually set to a few times the spacing between data points on the *x* axis depending on how big a window you want to use when smoothing.

The *supsmooth* function uses a symmetric *k* nearest neighbor linear least-squares fitting procedure to make a series of line segments through your data. Unlike *ksmooth* which uses a fixed bandwidth for all your data, *supsmooth* will adaptively choose different bandwidths for different portions of your data.

	medsmooth(**vy**, *n*)	Returns an *m*-element vector created by smoothing **vy** with running medians. **vy** is an *m*-element vector of real numbers. *n* is the width of the window over which smoothing occurs. *n* must be an odd number less than the number of elements in **vy**.
Pro	ksmooth(**vx**,**vy**, *b*)	Returns an *n*-element vector created by using a Gaussian kernel to return weighted averages of **vy**. **vy** and **vx** are *n*-element vectors of real numbers. The bandwidth *b* controls the smoothing window and should be set to a few times the spacing between your *x* data points.
Pro	supsmooth(**vx**,**vy**)	Returns an *n*-element vector created by the piecewise use of a symmetric *k*-nearest neighbor linear least-squares fitting procedure in which *k* is adaptively chosen. **vy** and **vx** are *n*-element vectors of real numbers. The elements of **vx** must be in increasing order.

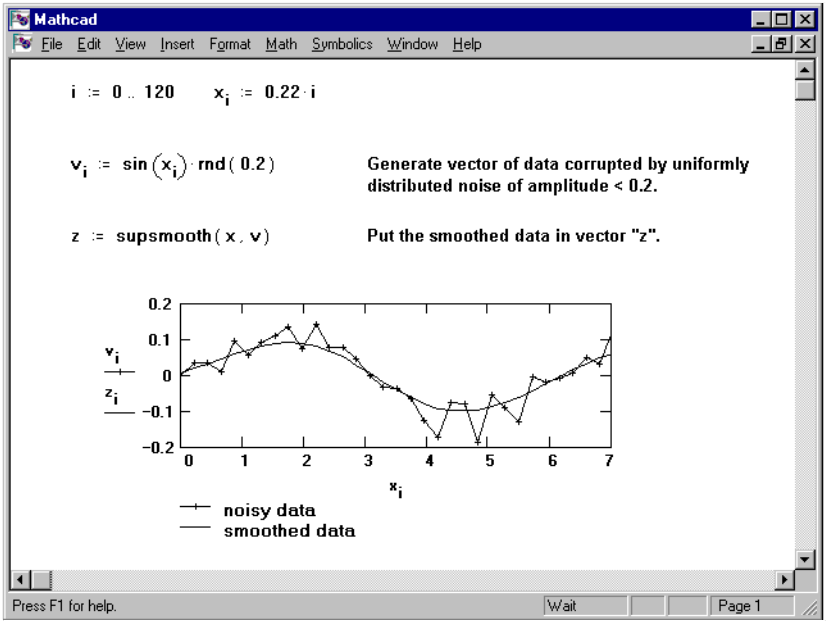

Figure 14-13: Smoothing noisy data with supsmooth.