Customizing the X Front End, Handling Large Lists, and Generating Surface Plots

Members of the Wolfram Research technical-support staff present articles on problems and questions we often encounter. In this column, we describe how to customize the appearance and behavior of the X front end, how to construct and manipulate large expressions efficiently, and how to produce surface plots from various types of data sets.

Edited by Todd Gayley

Customizing the X Front End

John Fultz
jfultz@wri.com

One of the great advantages of the X Window System is the ability to customize applications. By modifying a resource file (a file that contains settings pertinent to an application or utility), you can change hundreds of details about the way any X program runs, from color to fonts to button sizes to default directories. Unfortunately, this feature can also be one of the most confusing. X has thousands of modifications that can be made to its resources, and each application program can add hundreds or thousands more. This column will take a brief look at some of the modifications that can be made to the X front end for *Mathematica*. I assume in this discussion that you have at least a basic knowledge of UNIX and that you have used the X front end.

The resource settings (or just resources for short) for the X front end are specified in a resource file called `XMathematica`. This resource file can be found in the `FrontEnd` subdirectory of your *Mathematica* directory. You should copy this file into your home directory and make all changes to the copy. The `XMathematica` file in your home directory will automatically supersede the one in the `FrontEnd` subdirectory. This way, the changes you make to the file will affect only you and not everyone on your system.

Constructing Resources

The complete form of a resource setting looks like the following:

object.subobject[.*subobject*...].*attribute: value*

The object refers to the application (the X front end, in this case). The subobjects refer to a hierarchy of objects that have been programmed into the X front end. For example, the first subobject might refer to all dialog boxes, the second might refer to a specific dialog box, the third might refer to all buttons of that specific dialog box, and the fourth might refer to a specific button of that dialog box. Finally, the attribute might be a color of that button, or the font within the button.

Resource settings are rarely referred to in such an exact manner for two reasons. First, the user usually wants to be able to change many resources at once, such as the font in all dialog boxes. Second, many resource names are quite long and unintelligible. Instead of specifying a resource completely, you can use wildcards. A wildcard, represented by an asterisk (*), simply tells X to set the attribute for all subobjects of an appropriate class. The asterisk is used in place of the period in the resource setting. For example, inserting the line:

```
*Menu*background: gray
```

into the `XMathematica` file will set the background color of all Menu objects (the menu bar and any pull-down menus) to gray. A subobject does not need to be specified. The background attribute of all dialog boxes, menu bars, screen borders, and so on, could be set to gray with the line:

```
*background: gray
```

Colors

A good way to start experimenting with resources is by modifying colors, since they are among the most visible and easiest attributes to alter. First, you need to know which colors are available on your system. The file `/usr/lib/X11/rgb.txt` contains a list of color names. You must use a color name in this list. Many of the usual color names (such as brown, black, gray, orange, and red) should already be there, so you may not need to search this list unless you are looking for an unusual color.

In addition to choosing a color, you must choose a color attribute to modify. There are several such attributes: **background** and **foreground** allow you to change the background

and foreground colors of any specified subobject (as in the example above). Usually, only text is displayed in the foreground color. **selectColor** is the color of the inside of a toggle button when it is pressed in. **topShadowColor** and **bottomShadowColor** are the decorative colors used for shadowing along the top and right sides of a box, and the bottom and left sides of a box, respectively. **highlightColor** is the color of the highlight box that indicates the active region of a dialog box. The following example would change the color of the highlight box:

```
*highlightColor:  red
```

Fonts

It is easy to change a font in a notebook using the Style menu. However, many people would also like to change the fonts in the menu bar or dialog boxes, usually because these fonts show up very small on a high-resolution screen. You can change these fonts by setting a resource and specifying the name of a font file.

It can be difficult to choose the right filename, since there are usually a great many files and their names are long and complex. (Many systems require a separate file for each size and face of the font. A typical filename is `-*-helvetica-medium-r-*-*-*-12-*-*-*-*-*-*-*`.) The best way to choose a font is with the utility *xfontsel*, which comes with the standard X Window System distribution (not the X front end). If you do not have this utility, ask your system administrator to install it for you.

To use *xfontsel*, you simply point to the option you want to change, click, and hold. A menu will come up of all the possible options. Any options you've not yet picked have a wildcard (*) in their place. It's not necessary to pick values for all options; the wildcards can be left where they are when you've found a font you like. In fact, you would usually only need to change four of the options: *fmly* is the font family (such as Courier or Helvetica); *wght* is the font's weight (bold, light, normal, and so on); *slant* usually has the options *r* for roman (no slant), *o* for oblique, and *i* for italic; and *pxlsz* gives the font's size in points.

Once you've chosen a font, simply copy the filename (the line which has a bunch of -'s and *'s in it) from the *xfontsel* window into the appropriate setting in the `XMathematica` file. The resource setting one would normally use to change fonts is **fontList**. The following example would change the fonts of all dialog boxes and menu selections to 12 point Helvetica Roman.

```
*fontList: -*-helvetica-medium-r-*-*-*-12-*-*-*-*-*-*
```

Translations and Actions

In X, every event, such as a keystroke or a mouse button press, is given a standard name, which is called its translation. For example, the translation for pressing button 1 of the mouse is known to X as <Btn1Down>, and the translation for the *g* key is <Key>g (all translations for keypresses have <Key> in them).

Any action that can be taken also has a name. Appendix E of the *User's Guide for the X Front End* contains a list of the actions, and they include most of the menu selections. It is possible to bind a translation to an action so that whenever you press a button or hit a key, a certain action will result. A list of default bindings is shown in Appendix E as well. For example, if the X front end detects the translation Shift<Key>Return, which in English is Shift-Return, it will execute the action **evaluate-selection()**, which is the Evaluate Selection command.

It is possible to add new bindings so that any key that has not already been defined, such as a function key, can be used. (The one exception is the F1 key, which is defined as **On Context help**.) First, you must consider which action you wish to bind to which key. For example, try binding the F2 function key to the Evaluate Next Input action. According to Appendix E, the action for Evaluate Next Input is **evaluate-next-input()**. Now, you must find the name of the keypad Enter key. To do this, bring up the X Environment Info dialog box from the Help menu of the X front end. In this dialog box, click the button Find Keys and Modifiers. Any key you type into the resulting dialog box will cause that dialog box to display its name. In my case, the F2 function key is called F2.

Now that you have the information you need, you simply add the appropriate information to the `XMathematica` file. First, find the section in your `XMathematica` file that reads as follows:

```
*Notebook.translations: #override \n\
<Btn1Up>: selection-put() \n\
<Btn2Down>: selection-get()
```

Even though it spans multiple lines, this entry is a single resource. Each new line adds a new translation, and the block of lines is read as a single resource because of the `\n\` characters which "join" the lines together. There are two translations already defined, for the <Btn1Up> and <Btn2Down> actions. To add your new translation, you first need to add the `\n\` to the end of the last line, then add a new line immediately afterward:

```
*Notebook.translations: #override \n\
<Btn1Up>: selection-put() \n\
<Btn2Down>: selection-get() \n\
<Key>F2: evaluate-next-input()
```

Note the <Key> before the key name. This simply means that the type of event that is expected is a key, specifically F2. If you wish to refer to the combination of Shift-F2, Control-F2, or Mod1-F2, you would use Shift<Key>F2, Ctrl<Key>F2, and Alt<Key>F2, respectively.

Some Practical Applications

While it can be fun just to play with `XMathematica` resources and have the X front end come up in purple and orange, some very productive changes can also be made to the resource settings.

One of the first things you should do after installing the X front end is to set up the `Print` command correctly. When printing to a printer, the X front end actually prints through

a pipe to a UNIX shell command. This command is given by a resource called ***printCommand** and it is already set in the `XMathematica` file to `lpr`. (See Appendix E of the *User's Guide for the X Front End*.) You can change the command, for example, to print to a particular printer called **laser1** by changing the ***printCommand** resource to:

```
*printCommand: lpr -Plaser1
```

On some high-resolution screens, it can be hard to see whether or not toggle buttons in the menus and dialog boxes are pressed. To fix this problem, you can add the following line to your `XMathematica` file:

```
*selectColor: black
```

You can highlight a menu to attract attention to it. For example, if you are a system administrator and you want to encourage users to look at the Help menu before they come running to you, you can change the background color with the line:

```
*Menu*Help*background: red
```

You can also highlight a particular option of a menu. This might be handy if you are teaching students how to use the X front end. For example, to highlight just the Save option of the File menu, you could use:

```
*Menu*File*Save*foreground: red
```

One of the most useful things you can do is change the text for the keyboard shortcuts that are listed beside menu options. The text refers to the Mod1 and Mod2 keys, which are unfamiliar to many users. If the computers at your site use the Alt key for Mod1, for example, you can replace references to Mod1 with Alt. To change the keyboard reference for New, which has a keyboard shortcut of Mod1+N, use the following entry in the `XMathematica` file:

```
*Menu*File*New*acceleratorText: Alt+N
```

There are many creative ways to modify resources – far more than I could list here. After experimenting with the examples here, you should be able to come up with custom revisions to suit yourself or your lab.

Handling Large Lists Efficiently

David Withoff
Research and Development
withoff@wri.com

This note describes the efficiency of various strategies for building and manipulating large expressions. One simple way of building a large expression is by repeatedly appending elements to a smaller expression. Here is a function that constructs a list of the first n even integers by appending the selected integers to a list.

```
In[1]:= f1[n_] :=
            Module[{result},
                result = {};
                Do[If[EvenQ[k], AppendTo[result, k]], {k, n}];
                result ]

In[2]:= f1[10]
Out[2]= {2, 4, 6, 8, 10}
```

An optimal program for building this list would require time proportional to the number of elements in the list, just as counting from 1 to n requires time proportional to n. As can be seen from the following experiment, however, the time required by `f1` is instead proportional to the square of the number of elements in the list. (The timings are from *Mathematica* Version 2.2 on a 33MHz NeXTStation Turbo.)

```
In[3]:= Table[{n, Timing[f1[n]][[1]]},
            {n, 1000, 4000, 1000}] // TableForm
Out[3]//TableForm=
        1000    1.36667 Second
        2000    4.51667 Second
        3000    9.38333 Second
        4000    15.8667 Second
```

This behavior is a consequence of the fact that *Mathematica* expressions are stored as arrays. The use of arrays affords a number of efficiency advantages, such as constant access time for arbitrary elements in an expression. However, when an element is added to an array, it is necessary to create a new array with space for one additional element, and to fill in the remaining elements by copying them from the original array. Creating and copying an array requires time proportional to the number of elements in the array.

In the function `f1`, each evaluation of `AppendTo[result, k]` makes a copy of the list and replaces the old list with the new one. Each copy requires time proportional to the number of elements in the list. The total time for building the final list is proportional to the sum $1 + 2 + \ldots + n$, which is in turn proportional to n^2.

A well-established solution to this problem is to build up the intermediate results using linked lists rather than arrays. A linked list is a common data structure in which each link consists of an element and a pointer to the rest of the list. In *Mathematica*, expressions are actually stored as arrays of pointers, with each pointer pointing to the corresponding element of the expression. As a result, a nested list in *Mathematica* is a linked list, both internally and operationally. The final nested list can be flattened to obtain a result in the form of an array.

```
In[4]:= f2[n_] :=
            Module[{result},
                result = {};
                Do[If[EvenQ[k], result = {result, k}], {k, n}];
                Flatten[result] ]

In[5]:= f2[10]
Out[5]= {2, 4, 6, 8, 10}
```

```
In[6]:= Table[{n, Timing[f2[n]][[1]]},
          {n, 1000, 4000, 1000}] // TableForm
Out[6]//TableForm=
        1000    0.416667 Second
        2000    0.8 Second
        3000    1.23333 Second
        4000    1.58333 Second
```

Unlike `f1`, the function `f2` requires time proportional to the number of terms in the final result, rather than proportional to the square of the number of terms in the final result. The intermediate results are nested lists. As with any linked list, each level in the intermediate result is a pair consisting of an element and an expression representing the rest of the list. After the tenth iteration, for example, the intermediate result is `{{{{{{}, 2}, 4}, 6}, 8}, 10}`, which can be flattened to obtain `{2, 4, 6, 8, 10}`.

If the elements in the final result are themselves lists, you will probably want to use expressions other than lists for accumulating intermediate results. Here is a function that constructs a list of pairs using expressions with a head of `h` to accumulate intermediate results. The optional third element in `Flatten` is used to specify that only expressions with ahead of `h` should be flattened.

```
In[7]:= g[n_] :=
          Module[{result, h},
              result = h[];
              Do[result = h[result, {k-1, k}], {k, n}];
              List @@ Flatten[result, Infinity, h] ]

In[8]:= g[5]
Out[8]= {{0, 1}, {1, 2}, {2, 3}, {3, 4}, {4, 5}}
```

Another approach to the problem of building a list of even integers is to set up an initial list and fill in the result using part assignments.

```
In[9]:= f3[n_] :=
          Module[{result},
              result = Table[Null, {Floor[n/2]}];
              Do[If[EvenQ[k], result[[k/2]] = k], {k, n}];
              result ]
```

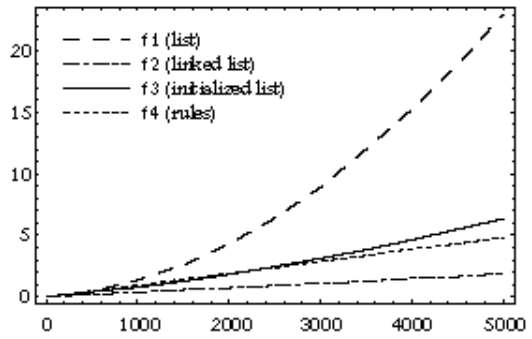
The approach represented by `f3` allows you to avoid the deep recursion (and potential stack overflow) associated with long linked lists. By avoiding the extra structure required for a linked list, `f3` can operate in less than half the memory of `f2`. This approach does, however, require that you know the size of the final result at the outset. Furthermore, although the list is not evaluated, it is sometimes necessary to perform internal operations that require time proportional to the number of elements in the list. For example, if there are other references to elements in the list, the list may be copied to avoid conflict with those references. In the worst-case situation, the time can be proportional to the square of the number of elements in the list, but it is usually much better than that, and it is invariably faster than the method represented by `f1`.

A fourth approach to building a large expression is to use rules attached to a symbol to accumulate intermediate results. Here is a function that constructs a list by attaching rules to asymbol `fi`.

```
In[10]:= f4[n_] :=
           (  Do[If[EvenQ[k], fi[k/2] = k], {k, n}];
              Table[fi[k], {k, Floor[n/2]}]  )

In[11]:= f4[10]
Out[11]= {2, 4, 6, 8, 10}

In[12]:= ?fi

      Global`fi
      fi[1] = 2
      fi[2] = 4
      fi[3] = 6
      fi[4] = 8
      fi[5] = 10
```

This approach is motivated by the fact that *Mathematica* uses hashing to store rules that do not have pattern expressions on their left-hand sides. Hashing is considerably faster than directly modifying the entire list. The greatest disadvantage of this approach is memory. The overhead of adding a rule is typically about 70 bytes, as compared with 4 bytes for adding an element to an expression.

The relative speed of `f1`, `f2`, `f3`, and `f4` can be seen in the following graph.



Once a large expression has been constructed, the speed for selecting, deleting, inserting, and swapping elements in that expression can be analyzed by applying general principles of computer programming, along with a few straightforward bits of information about *Mathematica* internals. Two of the more important bits of information are that the amount of time required to copy or evaluate an expression is asymptotically proportional to the number of elements in the expression, and that automatic hashing of rules attached to symbols can be exploited to speed up certain operations, such as swapping. The relative merits of different data structures depend on the operations you expect to use the most often.

As an example, consider the speed for inserting elements into an array, a linked list, and a set of rules attached to a symbol. Since inserting an element into an expression (array)

requires the creation of a new expression, the time required for the insertion is proportional to the number of elements in the expression. The time is independent of the position of the insertion.

```
In[13]:= data = Range[5000];

In[14]:= Do[data = Insert[data, new, 1], {100}] // Timing
Out[14]= {3.91667 Second, Null}

In[15]:= data = Range[10000];

In[16]:= Do[data = Insert[data, new, 1], {100}] // Timing
Out[16]= {7.61667 Second, Null}
```

This performance can be improved by enclosing the expression in **Hold**. Since evaluation looks only for rules associated with the head of the expression (downvalues) and elements of the expression (upvalues), you can avoid nontrivial evaluations by ensuring that the elements are at least two levels deep in the expression.

```
In[17]:= data = Hold[Evaluate[Range[10]]
Out[17]= Hold[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}]

In[18]:= Insert[data, new, {1, 5}]
Out[18]= Hold[{1, 2, 3, 4, new, 5, 6, 7, 8, 9, 10}]

In[19]:= data = Hold[Evaluate[Range[10000]]];

In[20]:= Do[data = Insert[data, new, {1, 1}], {100}] // Timing
Out[20]= {2.21667 Second, Null}
```

As can be seen by comparing **In[20]** with **In[16]**, the use of **Hold** results in a speed improvement of about a factor of three.

The time for insertion of an element into a linked list depends on the position of the insertion relative to the top of the list, but is otherwise independent of the number of elements in the list. All of the links have exactly two elements, so the time needed for updating long expressions is not an issue.

```
In[21]:= SetAttributes[LLInsert, HoldFirst]

In[22]:= LLInsert[ll_, new_, n_] :=
         ((Part[ll, ##] = {new, Part[ll, ##]})&) @@ Table[2, {n-1}]

In[23]:= ll = {}; Do[ll = {k, ll}, {k, 10, 1, -1}]; ll
Out[23]= {1, {2, {3, {4, {5, {6, {7, {8, {9, {10, {}}}}}}}}}}}

In[24]:= LLInsert[ll, new, 5]; ll
Out[24]= {1, {2, {3, {4, {new, {5, {6, {7, {8, {9, {10,{}}}}}}}}}}}}

In[25]:= Flatten[ll]
Out[25]= {1, 2, 3, 4, new, 5, 6, 7, 8, 9, 10}
```

Here are some timings for repeated insertion into large linked lists.

```
In[26]:= ll = {}; Do[ll = {k, ll}, {k, 1000, 1, -1}]

In[27]:= Do[LLInsert[ll, new, 5], {100}] // Timing
Out[27]= {0.283333 Second, Null}

In[28]:= Do[LLInsert[ll, new, 5], {200}] // Timing
Out[28]= {0.566667 Second, Null}

In[29]:= ll = {}; Do[ll = {k, ll}, {k, 5000, 1, -1}]

In[30]:= Do[LLInsert[ll, new, 5], {200}] // Timing
Out[30]= {0.55 Second, Null}
```

As expected, the time for insertion is proportional to the position of the insertion relative to the top of the linked list, and is independent of the number of elements.

The speed for insertion into a set of rules attached to a symbol depends on the number of rules that must be moved to accommodate the new rule. As mentioned earlier, searching a list of rules with left-hand sides that do not contain pattern expressions is speeded up by automatic hashing. The total time for insertion is therefore proportional to the number of rules that must be moved, and to the time required to insert a rule into the internal hash table.

As a final example, consider swapping elements in the data. The approach based on a list enclosed in **Hold**, and the approach based on rules attached to a symbol, both allow elements to be swapped at a rate that is roughly independent of the number of elements.

```
In[31]:= data = Hold[Evaluate[Range[10000]]];

In[32]:= Do[k = Random[Integer, {1, 9999}];
           temp = data[[1, k]];
           data[[1, k]] = data[[1, k+1]];
           data[[1, k+1]] = temp, {1000}] // Timing
Out[32]= {1.91667 Second, Null}

In[33]:= Do[fi[k] = k, {k, 10000}]

In[34]:= Do[k = Random[Integer, {1, 9999}];
           temp = fi[k];
           fi[k] = fi[k+1];
           fi[k+1] = temp, {1000}] // Timing
Out[34]= {2.93333 Second, Null}
```

In contrast, swapping elements in a list that is not enclosed in **Hold**, or swapping elements in a linked list, will require time that is proportional to the total number of elements.

There is no single data structure that is optimal for all purposes. In most cases, data structures that are optimal for a particular task in other programming languages will be optimal in *Mathematica* as well.

Generating Surface Plots from Data Sets

Robby Villegas
villegas@wri.com

Suppose you have some data representing a set of points in space and you want to plot a surface through the points. The data might be a matrix of z values for a function z = $f(x, y)$ or a list of sample points (x, y, z) that could be neatly

distributed or randomly scattered. There are several functions you can use to generate a surface, depending on the form of the data set. In this article, we will look at all the situations that arise commonly.

One simple type of data set is a matrix of numbers giving the z values of some function on a uniformly spaced grid of (x, y) points. As a *Mathematica* array, the matrix would look like:

```
{ {z11, z12, ..., z1n},
  {z21, z22, ..., z2n},
         ...
         ...
  {zm1, zm2, ..., zmn} }
```

Most likely, the data is stored in a file. In this case, we probably have m lines in the file, each having n real numbers separated by spaces or tabs. Retrieve the data using **ReadList**:

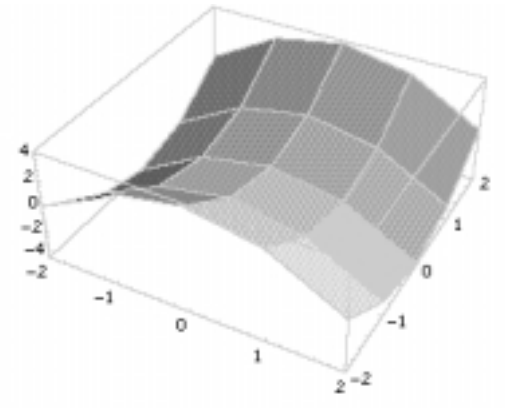
```
In[1]:= zArray = ReadList["datafile1", Number, RecordLists->True];

In[2]:= MatrixForm[zArray]
```

Out[2]//MatrixForm=

−5	0	3	4	3	0	−5
−8	−3	0	1	0	−3	−8
−9	−4	−1	0	−1	−4	−9
−8	−3	0	1	0	−3	−8
−5	0	3	4	3	0	−5

When we assume that the x and y coordinates of the grid are uniformly spaced, all we need to know is the rectangle $[a, b] \times [c, d]$ we are plotting over, and the dimensions of the array, and we can figure out where the grid points lie. **ListPlot3D** will do this for us, if we give it the array and the domain using the **MeshRange** option. For our **zArray**, we get:

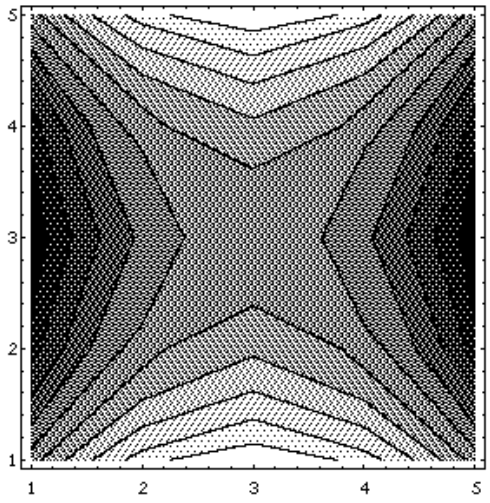
```
In[3]:= ListPlot3D[zArray, MeshRange -> {{-2, 2}, {-3, 3}}]
```



If we don't specify a **MeshRange**, the default for an **m** by **n** matrix will be **{{1,n}, {1,m}}**, so our tick labels will start with 1 on each axis.

The matrix of z values can also be represented by contour or density plots, which are generated in the same way as a surface plot. Use the functions **ListContourPlot** or **ListDensityPlot** in place of **ListPlot3D**. These three are just the **List** forms of their function-plotting brethren: **Plot3D**, **ContourPlot**, and **DensityPlot**.

```
In[4]:= ListContourPlot[zArray]
```

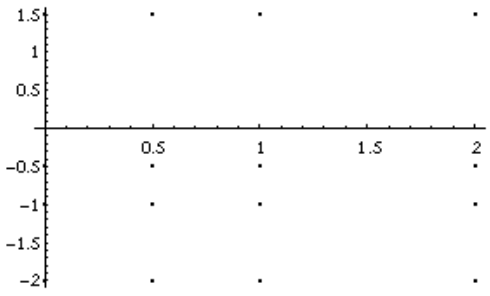


Now, suppose our data set is a list of (x, y, z) triples where the (x, y) points form a grid, but the x and y values aren't uniformly spaced. There are at least two approaches to obtaining a plot from such a data set. One method is to create an interpolating function for the data, which can then be passed to any of the main three function-plotters we mentioned above. The built-in command for converting a data set to an interpolating function is **Interpolation**. It takes a flat list (not an array) of sample points on the graph of the function. The coordinates of the independent variables (the (x, y) values, in this case) must form a full grid. For example,

```
In[5]:= triples = ReadList["datafile2", {Number, Number, Number}];
Out[5]= {{0, -2, -4}, {0, -1, -1}, {0, -0.5, -0.25},
   {0, 1.5, -2.25}, {0.5, -2, -3.75}, {0.5, -1, -0.75},
   {0.5, -0.5, 0.}, {0.5, 1.5, -2.}, {1, -2, -3},
   {1, -1, 0}, {1, -0.5, 0.75}, {1, 1.5, -1.25},
   {2, -2, 0}, {2, -1, 3}, {2, -0.5, 3.75},
   {2, 1.5, 1.75}}
```

We can see the grid-like, but non-uniformly spaced, arrangement of the (x, y) points:

```
In[6]:= ListPlot[ Drop[#, -1]& /@ triples ]
```



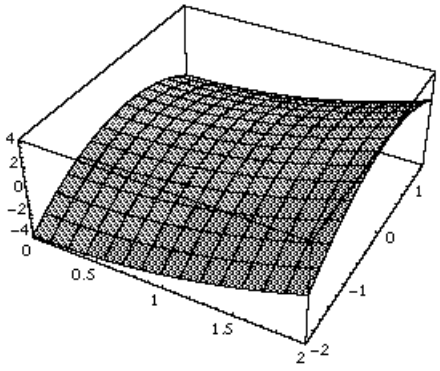
Here is how to create the interpolating function:

```
In[7]:= f[x_, y_] = Interpolation[triples][x, y]
Out[7]= InterpolatingFunction[{{0, 2}, {-2, 1.5}}, <>][x, y]
```

Notice that the first element of the resulting **InterpolatingFunction** object is the domain of the function, given in the same format that **PlotRange** uses for graphics functions.

Now, use a function-plotter, such as **DensityPlot** or **ListPlot**:

```
In[8]:= Plot3D[f[x, y], {x, 0, 2}, {y, -2, 1.5}]
```



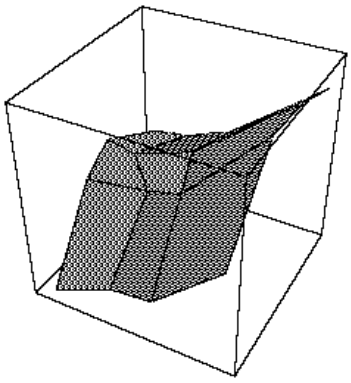
A second approach is to use **ListSurfacePlot3D** from the standard package **Graphics`Graphics`**. **ListSurfacePlot3D** is more general than the **Interpolation** approach in that it doesn't require a data set that corresponds to a grid in the xy plane. This function takes a *matrix* of triples and creates a surface mesh by connecting adjacent points (that is, adjacent in the matrix) to form quadrilateral tiles. In other words, it uses the matrix structure imposed on the points to determine which points to connect, rather than requiring a geometric relation, such as a grid structure, among the (x, y) points. The surface need not even be the graph of a function $z = f(x, y)$; you could, for example, plot a sphere with **ListSurfacePlot3D**).

As an example, let's say that we have a data set similar to **triples**, except the x and y coordinates don't fall on a grid. We can create such a list from **triples** as follows:

```
In[9]:= newtriples =
          Apply[{#1 + Random[Real, {-.2, .2}],
                #2 + Random[Real, {-.2, .2}], #3}&, triples, {1}];
```

We now need to impose a matrix structure on **newtriples** to reflect the rows and columns of the distorted xy "grid". We use **Partition** to group the 16 points into a 4×4 matrix:

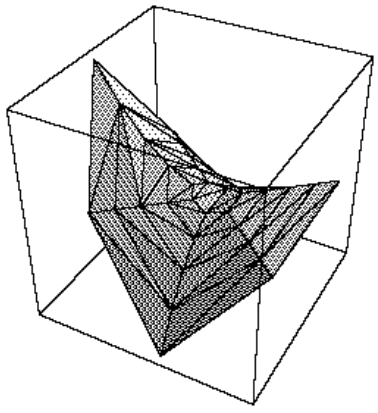
```
In[10]:= triplesArray = Partition[newtriples, 4]
Out[10]= {{{0.00987987, -2.12021, -4}, {-0.103468, -0.963969, -1},
           {-0.141676, -0.313285, -0.25},
           {-0.0740009, 1.49922, -2.25}},
          {{0.550965, -1.81946, -3.75},
           {0.630228, -0.898042, -0.75},
           {0.322184, -0.553424, 0.}, {0.343327, 1.55515, -2.}},
          {{1.13635, -2.00058, -3}, {1.09807, -1.07852, 0},
           {1.00018, -0.502834, 0.75}, {0.85717, 1.57161, -1.25}},
          {{1.98385, -2.1641, 0}, {1.95363, -0.861681, 3},
           {2.09601, -0.41561, 3.75}, {1.98389, 1.62394, 1.75}}}

In[10]:= Needs["Graphics`Graphics3D`"]

In[10]:= ListSurfacePlot3D[triplesArray, BoxRatios -> {1, 1, 1}]
```



If the ranges on the x, y, and z axes for the data set have very different lengths (say, by an order of magnitude or more), the default rendering of the surface will be either very flat or very narrow. You can force the dimensions of the plot to be more reasonable using the **BoxRatios** option. Start with **BoxRatios -> {1, 1, 1}** and then try finer adjustments if necessary.

The functions **ListPlot3D**, **ListContourPlot**, **ListDensityPlot**, and **ListSurfacePlot3D** all require data in matrix form, whereas the **Interpolation** method requires a list of points (x, y, z) such that the points (x, y) form a full grid. It is usually worthwhile to obtain data in one of these forms, if possible. What if our data consists of a scattering of points with no such structure? It is easy to plot the points themselves using **ScatterPlot3D**, from the standard package **Graphics`Graphics3D`**, but tiling out a surface in this situation takes much more computation. The standard package **DiscreteMath`ComputationalGeometry`** contains the function **TriangularSurfacePlot** for rendering a scatter set into a surface. **TriangularSurfacePlot** takes a flat list of triples:

```
In[11]:= Needs["DiscreteMath`ComputationalGeometry`"]

In[11]:= TriangularSurfacePlot[triples, BoxRatios -> {1, 1, 1},
           PlotRange -> All]
```



Note that the **BoxRatios** option is used to scale the axes. Like **ListPlot3D**, **TriangularSurfacePlot** assumes that the surface looks like the graph of a function $z = f(x, y)$, but, unlike **ListPlot3D**, it doesn't require a grid structure on the (x, y) points. It performs a triangulation of the (x, y) points in the plane and then joins the corresponding (x, y, z) points in

space to create the triangular tiles that approximate the surface. This is a computationally intensive process, so **TriangularSurfacePlot** is very slow for large data sets.

The **List** functions we have discussed all create a graphic from a data list (or matrix). With the **Interpolation** method, the data are converted to an approximating function as a pre-processing step, allowing us to use function-plotters such as **Plot3D**. Another common approximating technique is to pick a specific functional form, for example

$$f(x, y) = x^3/(x + a) - y^3/(y + b),$$

and then find values for the parameters (a, b, and c) so that the function of that form fits the data in the sense of least squares. The functional form can be either linear and non-linear.

In the linear variety, we pick a set of functions and ask for the linear combination of them that produces the closest fit to the data. This approximation is the province of **Fit**. **Fit** takes a simple list of data points, a list of basis functions in the independent variables, and the list of independent variables. It produces a linear combination of the basis functions that most closely matches the data.

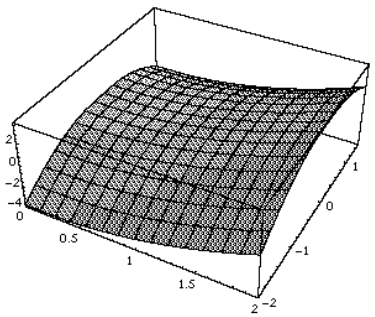
```
In[12]:= Fit[newtriples, {1, x, y, x^2, y^2}, {x, y}]

Out[12]= -0.158349 - 0.18583 x + 1.11653 x^2 +

       0.111044 y - 0.873742 y^2
```

For non-linear functional forms, you can use the **NonlinearFit** function.

```
In[13]:= Needs["Statistics`NonlinearFit`"]

In[14]:= paramvalues =
          NonlinearFit[newtriples, Sqrt[x^4 + a^2] - Sqrt[y^4 + b^2],
            {x, y}, {a, b}]
Out[14]= {a -> 0.176215, b -> 0.357848}

In[15]:= f[x_, y_] = Sqrt[x^4 + a^2] - Sqrt[y^4 + b^2] /.
                       paramvalues

                                    4          4
Out[15]= Sqrt[0.0310518 + x ] - Sqrt[0.128055 + y ]
```

In this case, the chosen model is probably poorly suited to the data, so the graph of the function won't look much like the data plot. In principle, however, this is a fit of a model to the data.

```
In[16]:= Plot3D[f[x, y],  {x, 0, 2}, {y, -2, 1.5}]
```



The advantage of these least-squares function approximation methods is that they require no particular distribution for the points. The disadvantage is that they assume the data represent a function, and they require that you supply a functional form that will reasonably fit the data.

To summarize, the method you choose depends on the form of your data. If you have an array of z values, you can use **ListPlot3D**. It assumes a uniform grid of (x, y) coordinates, whose range you can specify with the **MeshRange** option. If you have a list of (x, y, z) triples, and the (x, y) points form a full (but perhaps not uniformly spaced) grid, you can use the **Interpolation** method. If you have a matrix of triples representing points on a surface, you can use **ListSurfacePlot3D**. It joins adjacent points in the matrix to form the tiles of the surface. If you have just a list of triples representing points on the graph of a function $z = f(x, y)$, and you cannot group them into a sensible matrix structure, you must use **TriangularSurfacePlot**. Finally, if you are satisfied with a best-fit functional characterization of the surface and you can find an acceptable functional form, then you can use the fitting method.