In[] and Out[]

In[] and Out[] offers readers an opportunity to ask questions of the experts. The *Journal* encourages readers to submit problems in care of the editor.

Edited by Paul Abbott

Indexing Output Files

***Q:** I have a list of numbers:*

```
In[1]:= ran = Table[Random[], {5}]
Out[1]= {0.887179, 0.320433, 0.785525, 0.3723, 0.673808}
```

How can I save each number separately to an indexed output file?

Both `Get` and `Put` take the file name as a string:

```
In[2]:= ?Put
        expr >> filename writes expr to a file. Put[expr1, expr2,
           ..., "filename"] writes a sequence of expressions expri
           to a file.
```

It is easy to form a set of indexed file-names using string manipulation:

```
In[3]:= (names = Table[StringJoin["ran.", ToString[i]],
             {i, Length[ran]}]) // InputForm
Out[3]//InputForm= {"ran.1", "ran.2", "ran.3", "ran.4", "ran.5"}
```

The numbers and file names can be paired using `Transpose`:

```
In[4]:= Transpose[{ran, names}]
Out[4]= {{0.887179, ran.1}, {0.320433, ran.2}, {0.785525, ran.3},
           {0.3723, ran.4}, {0.673808, ran.5}}
```

Then `Apply` can be used to `Put` each value into a separate indexed file:

```
In[5]:= Apply[Put, %, 1];
```

As a check, we read in the first value:

```
In[6]:= ReadList["ran.1"] == {ran[[1]]}
Out[6]= True
```

Conditional Limits

***Q:** I have a function of two variables:*

```
In[1]:= f[x_, y_] = Sin[x-y]/(x-y);
```

Evaluating `f[x, x]` *leads to an indeterminate result because of the* `(x-y)` *term in the denominator:*

```
In[2]:= f[x, x]

        Power::infy: Infinite expression 1/0 encountered.

        Infinity::indet:
           Indeterminate expression 0 ComplexInfinity
             encountered.
Out[2]= Indeterminate
```

However, the limit as `x -> y` *clearly exists:*

```
In[3]:= Limit[f[x, y], x -> y]

Out[3]= 1
```

How can I construct a rule that conditionally evaluates the limit of `f[x, y]` *whenever* `x == y`*?*

You simply need to add two additional rules:

```
In[4]:= f[x_, x_] = Limit[f[x, y], y -> x]
Out[4]= 1

In[5]:= f[x_, y_] := f[x, x] /; x == y
```

The first rule computes the general value in the limit and the second rule identifies the case `x == y` using a condition (`/;`). The order in which these rules are entered is important. For symbolic and integer arguments, we get the desired behavior:

```
In[6]:= f[y, y]
Out[6]= 1

In[7]:= f[1, 1]
Out[7]= 1
```

Note that only limited simplification is done in testing the equality condition in the second rule:

 © 1994 Miller Freeman Publications

```
In[8]:= f[a(x-y), -a(y-x)]

         Sin[a (x - y) + a (-x + y)]
Out[8]= ---------------------------
          a (x - y) + a (-x + y)
```

Modifying the equality test can help in a number of cases:

```
In[9]:= f[x_, y_] := f[x, x] /; Simplify[x - y] == 0

In[10]:= f[Sin[2 x], 2 Sin[x] Cos[x]]

Out[10]= 1
```

A case like

```
In[11]:= f[1.,1]

Out[11]= 1
```

works because

```
In[12]:= 1. == 1

Out[12]= True
```

Numerical equality testing in *Mathematica* relies on `$MachineEpsilon`:

```
In[13]:= ?$MachineEpsilon

$MachineEpsilon gives the smallest machine-precision
   number which can be added to 1.0 to give a result not
   equal to 1.0.
```

This number is machine dependent (as its name implies):

```
In[14]:= $MachineEpsilon

                 -16
Out[14]= 2.22045 10
```

However, equality testing is not quite consistent with the definition of `$MachineEpsilon`:

```
In[15]:= 1.0 + 100 $MachineEpsilon == 1.0

Out[15]= True

In[16]:= 1.0 + 150 $MachineEpsilon == 1.0

Out[16]= False
```

Nevertheless, you can use `$MachineEpsilon` to control whether two numbers are to be considered "equal":

```
In[17]:= equal[x_, y_] = "Not Close Enough";

In[18]:= equal[x_, y_] := "Close Enough" /;
            Abs[x - y] < 2 $MachineEpsilon;
```

A test (using the *infix* operator form ~) shows the behavior in each case:

```
In[19]:= (1.0 + $MachineEpsilon) ~equal~ 1.0

Out[19]= Close Enough

In[20]:= (1.0 + 2 $MachineEpsilon) ~equal~ 1.0

Out[20]= Not Close Enough
```

Gradient in *n* Dimensions

***Q:** I am taking the gradient of expressions involving vector quantities:*

```
In[1]:= << Calculus`VectorAnalysis`

In[2]:= k = {kx, ky, kz}; r = {x, y, z};

In[3]:= Grad[Exp[I k.r]]

             I (kx x + ky y + kz z)
Out[3]= {I E                       kx,

            I (kx x + ky y + kz z)
         I E                       ky,

            I (kx x + ky y + kz z)
         I E                       kz}
```

I would like the output to be written as `I k Exp[I k.r]`. *Is there a way to avoid expanding variables to their full form?*

A first improvement is to use replacement rules instead of assignments:

```
In[4]:= Clear[k, r]

In[5]:= kr = {k -> {kx, ky, kz}, r -> {x, y, z}}

Out[5]= {k -> {kx, ky, kz}, r -> {x, y, z}}
```

Now we can compute the required gradient:

```
In[6]:= Grad[Exp[I k.r] /. kr]

             I (kx x + ky y + kz z)
Out[6]= {I E                       kx,

            I (kx x + ky y + kz z)
         I E                       ky,

            I (kx x + ky y + kz z)
         I E                       kz}
```

and then simplify this expression by recognizing that `k.r` appears in the output:

```
In[7]:= % /. (k.r /. kr) -> k.r

             I k . r         I k . r         I k . r
Out[7]= {I E        kx, I E        ky, I E        kz}
```

The reduction of more complicated expressions is, in general, not trivial.

The operator `Grad` in the package `Calculus`VectorAnalysis`` is designed for explicit computations in particular coordinate systems. In Cartesian coordinates, the derivative operator `D` almost does what we want:

```
In[8]:= D[Exp[I k.r], r]

            I k . r
Out[8]= I E        (0 . r + k . 1)
```

If the dot products in this expression were evaluated (`0.r -> 0` and `k.1 -> k`), we would have the desired result. Note that *Mathematica* does not assume that dot products are commutative (`Orderless`):

```
In[9]:= r . s - s . r

Out[9]= r . s - s . r
```

because Dot has been designed to work with matrix and tensor products:

```
In[10]:= ?Dot
         a.b.c or Dot[a, b, c] gives products of vectors, matrices
           and tensors.
```

Hence, two modifications of Dot are required:

```
In[11]:= Unprotect[Dot];
In[12]:= Dot[n_?NumberQ, r_] := n r
In[13]:= Dot[r_, n_?NumberQ] := n r
In[14]:= Protect[Dot];
```

We now get the desired result:

```
In[15]:= D[Exp[I k.r], r]
            I k . r
Out[15]= I E        k
```

The addition of these extra rules for Dot is quite general:

```
In[16]:= D[Exp[a I k.r]/Sqrt[r.r], r]
           I a k . r          I a k . r
          E          r   I a E          k
Out[16]= -(------------) + ----------------
                 3/2         Sqrt[r . r]
            r . r
```

It works because arguments of Dot are assumed to be vectors and all other variables and functions are treated as scalars.

Constants

***Q:** How can I define a test that checks whether an expression is constant?*

Symbols such as E and Pi have the attribute Constant:

```
In[1]:=  Attributes /@ {E, Pi}
Out[1]=  {{Constant, Protected}, {Constant, Protected}}
```

However, numbers and the imaginary unit I do not have this attribute:

```
In[2]:=  Attributes[I]
Out[2]=  {Locked, Protected}

In[3]:=  Attributes[3]
         Attributes::ssle:
           Symbol, string, or Literal[symbol] expected at position
             1 in Attributes[3].
Out[3]=  Attributes[3]
```

What is required is a test (or predicate operation) that checks whether a Symbol has the Constant attribute:

```
In[4]:=  ConstantQ[x_Symbol] := MemberQ[Attributes[x], Constant]
```

and also whether the expression is a number:

```
In[5]:=  ConstantQ[x_?NumberQ] := True
```

These tests alone are not sufficient:

```
In[6]:=  ConstantQ[Sin[3]]
Out[6]=  ConstantQ[Sin[3]]
```

The following rule says that a function applied to constant variables gives a constant:

```
In[7]:=  ConstantQ[f_[x__]] := And @@ (ConstantQ /@ {x})
```

After declaring that a and b are constants:

```
In[8]:=  SetAttributes[a, Constant]; SetAttributes[b, Constant]
```

we see that a function applied to constants is recognized as constant:

```
In[9]:=  ConstantQ[Sin[a + I b]]
Out[9]=  True

In[10]:= ConstantQ[E^Sin[Sin[3 a + Pi b]]]
Out[10]= True
```

Number of Assignments

***Q:** An assignment of the form*

```
v[x1_, x2_, ...] := v[x1, x2, ...] = ...
```

saves the computed values and associates them with v. *The saved values can be inspected using* ?v *or* Information[v]. *How can I find out how many values of* v *have been saved?*

Consider the following example function:

```
In[1]:=  v[x__] := v[x] = Sin[Plus @@ {x}]
```

After entering two inputs:

```
In[2]:=  v[a]
Out[2]=  Sin[a]

In[3]:=  v[a, b]
Out[3]=  Sin[a + b]
```

we see that two values associated with v have been saved along with the initial rule:

```
In[4]:=  ?v
         Global`v
         v[a] = Sin[a]
         v[a, b] = Sin[a + b]
         v[x__] := v[x] = Sin[Apply[Plus, {x}]]
```

However, since the output of ? and `Information` is `Null`, we cannot pass this information to another function to determine how many values have stored.

The rules associated with a symbol can be obtained using `DownValues`:

```
In[5]:= DownValues[v]
Out[5]= {Literal[v[a]] :> Sin[a], Literal[v[a, b]] :> Sin[a + b],
           Literal[v[x__]] :> (v[x] = Sin[Apply[Plus, {x}]])}
```

The number of such rules can be determined using

```
In[6]:= Length[DownValues[v]]
Out[6]= 3
```

Note that this number includes both the general defining rules and the specific cases that have been entered.

Linear Interpolation

Efficiency

***Q:** What is the most efficient way to do linear interpolation in Mathematica?*

The built-in `Interpolation` function:

```
In[1]:= ?Interpolation
        Interpolation[data] constructs an InterpolatingFunction
           object which represents an approximate function that
           interpolates the data. The data can have the forms
           {{x1, f1}, {x2, f2}, ...} or {f1, f2, ...}, where in
           the second case, the xi are taken to have values 1, 2,
           ....
```

does cubic interpolation by default:

```
In[2]:= Options[Interpolation]
Out[2]= {InterpolationOrder -> 3}
```

However, one can do linear interpolation by changing the `InterpolationOrder`:

```
In[3]:= ?InterpolationOrder
        InterpolationOrder is an option to Interpolation.
           InterpolationOrder -> n specifies interpolating
           polynomials of order n.
```

Consider the data set:

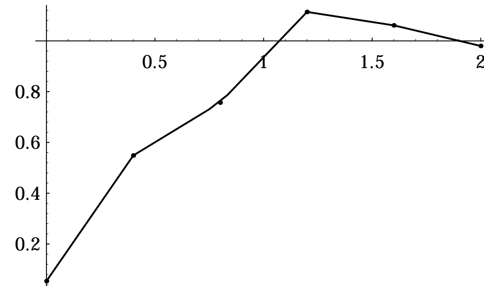
```
In[4]:= data = Table[{x, Sin[x] + Random[]/5}, {x, 0, 2, 0.4}]
Out[4]= {{0, 0.0537956}, {0.4, 0.548621}, {0.8, 0.756845},
           {1.2, 1.11405}, {1.6, 1.06116}, {2., 0.980065}}
```

One can perform linear interpolation using:

```
In[5]:= dataint = Interpolation[data, InterpolationOrder -> 1];
```

Here is a plot of the linearly interpolated data along with the data itself:

```
In[6]:= Plot[dataint[x], {x, 0, 2}, PlotRange -> All,
           Epilog ->
              ListPlot[data, PlotStyle -> PointSize[0.01],
              DisplayFunction -> Identity ][[1]] ];
```



Note the use of the `DisplayFunction` option to suppress the output of `ListPlot`, and `Epilog` to combine the `ListPlot` with the `Plot` of the interpolation function.

Numeric Differentiation

***Q:** I have a list of points in the plane:*

```
In[1]:= list = {{1, 4}, {2, 3}, {3, -4}, {4, 5}};
```

How can I compute the slope between each successive pair of points?

One method is to define a function that computes the slope between a pair of points:

```
In[2]:= slope[{a_, b_}] :=
           (Last[b] - Last[a])/(First[b] - First[a])
```

Construct a list of the successive pairs of points:

```
In[3]:= pairs = Partition[list, 2, 1];
```

Then `Map` the `slope` function over this list:

```
In[4]:= Map[slope, pairs]
Out[4]= {-1, -7, 9}
```

An alternative approach is to use a linear interpolation function:

```
In[5]:= g = Interpolation[list, InterpolationOrder -> 1];
```

and then compute the slopes at the midpoints of each interval by differentiation:

```
In[6]:= Table[g'[x], {x, 1.5, 3.5}]
Out[6]= {-1, -7, 9}
```

Breaking Lines

Q: *How can I control the width of output?*

Computing `Pi` to 100 digits shows the default width:

```
In[1]:= pi = N[Pi, 100]
Out[1]= 3.14159265358979323846264338327950288419716939937510582097\
    4944592307816406286208998628034825342117068
```

The option `PageWidth` specifies the width of each line in an output stream. The output streams are given by the variable `$Output`, so you can use `SetOptions` on this variable:

```
In[2]:= SetOptions[$Output, PageWidth -> 30]
Out[2]= {{FormatType -> OutputForm,
      PageWidth -> 30,
      PageHeight -> 22,
      TotalWidth -> Infinity,
      TotalHeight -> Infinity,
      StringConversion :>
        $StringConversion}}
In[3]:= pi
Out[3]= 3.14159265358979323846264338 3\
    27950288419716939937510582 0\
    97494459230781640628620899 8\
    628034825342117068
```

Setting `PageWidth` to `Infinity` simply means that lines won't be broken:

```
In[4]:= SetOptions[$Output, PageWidth -> Infinity];
In[5]:= pi
Out[5]= 3.14159265358979323846264338327950288419716939937510582097
```

Note that, for front ends supporting notebooks, line breaking and formatting can also be controlled by items under the **Style** menu such as **Cell Style**, **Attributes**, **Alignment**, **Formatter**, and **Ruler**.

MapThread

Q: *How can I compare corresponding elements of two matrices and record the maximum value of each pair?*

Consider the matrices:

```
In[1]:= (m = {{1, 5}, {2, 4}}) // MatrixForm
Out[1]//MatrixForm=   1   5
                      2   4
In[2]:= (n = {{3, 4}, {5, 2}}) // MatrixForm
Out[2]//MatrixForm=   3   4
                      5   2
```

The `MapThread` operator:

```
In[3]:= ?MapThread
MapThread[f, {{a1, a2, ...}, {b1, b2, ...}, ...}] gives
   {f[a1, b1, ...], f[a2, b2, ...], ...}. MapThread[f,
   {xa, xb, ...}, n] maps f over the nth level of the
   n-dimensional tensors xa, xb, ... .
```

applies a general operator at a specified level to matrices (and tensors):

```
In[4]:= MapThread[f, {m, n}, 2] // MatrixForm
Out[4]//MatrixForm=   f[1, 3]   f[5, 4]
                      f[2, 5]   f[4, 2]
```

We can use `MapThread` to apply `Max` to two matrices element-by-element:

```
In[5]:= MapThread[Max, {m, n}, 2] // MatrixForm
Out[5]//MatrixForm=   3   5
                      5   4
```

NumberForm

Q: *How can I save only five decimal digits of a result of a calculation to a file? The command:*

```
In[1]:= Write["file", N[Sqrt[2], 5]]
```

saves the result as:

```
In[2]:= ReadList["file"]
Out[2]= {1.414213562373095049}
```

You need to change the format of your data using `NumberForm`:

```
In[3]:= ?NumberForm
NumberForm[expr, n] prints with approximate real numbers
   in expr given to n-digit precision.
In[4]:= NumberForm[N[Sqrt[2]], 5]
Out[4]//NumberForm= 1.4142
```

Using `Write` again would simply *append* this value to `file`. To re-use `file` we first need to close it:

```
In[5]:= Close["file"];
```

Now we can write out the data to `file` in the appropriate format:

```
In[6]:= Write["file", NumberForm[N[Sqrt[2]], 5]]
```

As a check, we confirm the contents of `file`:

```
In[7]:= ReadList["file"]
Out[7]= {1.4142}
```

Matrix Operations

Replacing Blocks

Q: *I have two matrices:*

```
In[1]:= (i3 = IdentityMatrix[3]) // MatrixForm

Out[1]//MatrixForm=
        1   0   0
        0   1   0
        0   0   1

In[2]:= (m = {{a, b}, {c, d}}) // MatrixForm

Out[2]//MatrixForm=
        a   c
        b   d
```

How can I replace the lower right 2×2 block of `i3` *by* `m`*?*

A number of matrix manipulation tools are defined in the standard package `LinearAlgebra`MatrixManipulation``, but there is no function especially for replacing one part of a matrix by another.

Here is a procedure that returns a matrix in which the block of a matrix `mat` starting at position `{x, y}` has been replaced with the matrix `m`:

```
In[3]:= ReplaceBlock[mat_, m_, {x_, y_}]:=
          Module[{nx, ny, newmat = mat},
            {nx, ny} = Dimensions[m];
            Do[newmat[[i+x-1, j+y-1]] = m[[i, j]],
                {i, nx}, {j, ny}];
            newmat ]
```

For example:

```
In[4]:= ReplaceBlock[i3, m, {2, 2}] // MatrixForm

Out[4]//MatrixForm=
        1   0   0
        0   a   b
        0   c   d
```

Vectorizing Matrices

Q: *How can I stack the rows of a matrix into a vector; stack the upper triangular part of a symmetric matrix into a vector; and convert a vector into a symmetric matrix?*

You can use `Join` to stack the rows of a matrix into a vector:

```
In[1]:= (sym = {{a, b, c}, {b, d, e}, {c, e, f}}) // MatrixForm

Out[1]//MatrixForm=
        a   b   c
        b   d   e
        c   e   f

In[2]:= Join @@ sym

Out[2]= {a, b, c, b, d, e, c, e, f}
```

To stack the upper or lower triangular part of a symmetric matrix into a vector, it is a good idea to construct a test for symmetric matrices first:

```
In[3]:= SymmetricQ[m_?MatrixQ]:= m === Transpose[m]

In[4]:= SymmetricQ[sym]

Out[4]= True
```

Using `MapIndexed`:

```
In[5]:= ?MapIndexed

        MapIndexed[f, expr] applies f to the elements of expr,
           giving the part specification of each element as a
           second argument to f. MapIndexed[f, expr, levspec]
           applies f to all parts of expr on levels specified by
           levspec.
```

it is straightforward to `Take` the lower-triangular part of a matrix:

```
In[6]:= MapIndexed[Take[#1, First[#2]]&, sym] // TableForm

Out[6]//TableForm=
        a
        b   d
        c   e   f
```

Similarly, using `Drop` and `Flatten` we can extract the upper-triangular part:

```
In[7]:= MatrixToVector[mat_?SymmetricQ] :=
          Flatten[MapIndexed[Drop[#1, First[#2 - 1]]&, mat], 1]

In[8]:= vec = MatrixToVector[sym]

Out[8]= {a, b, c, d, e, f}
```

To convert a vector into a symmetric matrix, we note that an $m \times m$ symmetric matrix corresponds to a vector of length $m(m+1)/2$ (a so-called *triangular* number). Inverting this relation:

```
In[9]:= dim[n_] = (m /. Solve[m(m+1)/2 == n, m] // Last)

        -1 + Sqrt[1 + 8 n]
Out[2]= ------------------
                2
```

permits the construction of a procedure for converting a vector into a symmetric matrix:

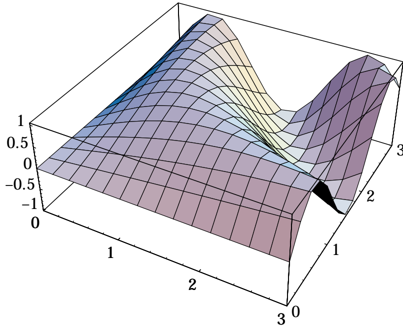
```
In[10]:= VectorToMatrix[vec_ /; IntegerQ[dim[Length[vec]]]] :=
         Module[{m = dim[Length[vec]]},
           Table[vec[[Max[i, j] + (Min[i, j] - 1)(m - Min[i, j]/2)]],
             {i, m}, {j, m} ]]

In[11]:= VectorToMatrix[vec] // MatrixForm
         a   b   c
         b   d   e
         c   e   f
```

Text in 3D Graphics

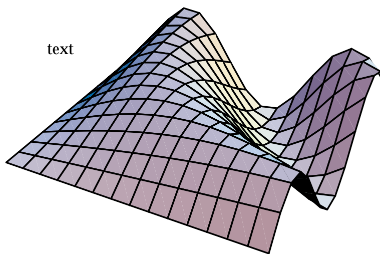
Q: *How can I avoid losing the axes and tick-marks when combining a surface plot with a* `Text` *object?*

Tom Wickham-Jones (`twj@wri.com`) replies: When `Show` receives a number of graphics objects as input, it coerces them to the same type. For example:

```
In[1]:= plot = Plot3D[Sin[x y], {x, 0, 3}, {y, 0, 3}]
```



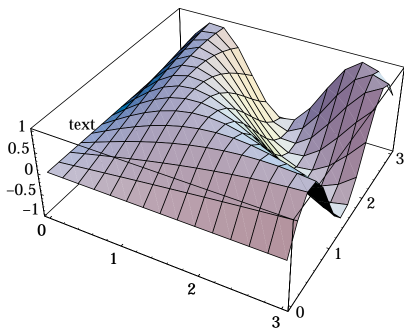

```
In[2]:= Show[{plot, Graphics[Text["text", {0.2, 0.7}, {1, 0}]]}]
```



In this example, the `SurfaceGraphics` object is coerced to a `Graphics` object. When that happens, only the `SurfaceGraphics` primitives are converted, not the options. Consequently, the axes and tick-marks are lost.

The solution is to keep the graphics as `Graphics3D` objects:

```
In[3]:= Show[{plot, Graphics3D[
          Text["text", {0.2, 0.7, 0.5}, {1, 0}]]}]
```



List Operations

Q: *How can I construct a simple function that takes an argument of the form* `{x, {a, b, ...}}`*, where* `x`*,* `a`*,* `b`*, ..., are all one-dimensional lists, and returns* `{Intersection[x, a], Intersection[x, b], ...}`?

Consider an example:

```
In[1]:= data = {{1, 3}, {{1, 3, 5}, {2, 4, 3}}};
```

Using `Map` (`/@`), it is possible to write a one-line expression for the required operation:

```
In[2]:= IntMap[exp_] :=
          Intersection[First[exp], #]& /@ Last[exp]

In[3]:= IntMap[data]

Out[3]= {{1, 3}, {3}}
```

An alternative way of defining `IntMap` expresses the data structure more clearly:

```
In[4]:= Remove[IntMap]

In[5]:= IntMap[{x_, y_List}] := Intersection[x, #]& /@ y

In[6]:= IntMap[data]

Out[6]= {{1, 3}, {3}}
```

This definition is slightly more efficient since it avoids repeated evaluation of `First[exp]`.

Series arithmetic

Q: *How can I generate the first few coefficients of the inverse square of a general series?*

Start with the first few terms of a general series:

```
In[1]:= series = Sum[a[i] x^i, {i, 0, 4}]

                           2        3        4
Out[1]= a[0] + x a[1] + x  a[2] + x  a[3] + x  a[4]
```

and expand the inverse square into a Taylor series:

```
In[2]:= 1/series^2 + O[x]^3

                                      2
                              6 a[1]    4 a[2]   2
                             (------- - ------) x
                                   2
              -2   2 a[1] x     a[0]     a[0]                 3
Out[2]= a[0]   - -------- + ---------------------- + O[x]
                        3                2
                    a[0]           2 a[0]
```

The required coefficients are easily extracted and simplified:

```
In[3]:= CoefficientList[%, x] // Simplify

                                   2
              -2  -2 a[1]   3 a[1]  - 2 a[0] a[2]
Out[3]= {a[0]  , -------, ---------------------}
                       3                4
                   a[0]             a[0]
```

Tone Series

Q: *I have some data from a set of measurements equally spaced over time. Is there a way to play the time series as a tone series so that each note has the same duration and the pitch varies according to the data?*

Consider the following data:

```
In[1]:=  data = Table[Random[Real, {2, 5}], {10}]
Out[1]=  {4.87375, 4.32473, 3.1849, 2.02131, 4.17003, 4.06697,
           2.19075, 2.80902, 2.72934, 4.09121}
```

A function for playing a note of a given frequency is easy to write:

```
In[2]:=  Note[f_, dur_:0.25] := Play[Sin[2 Pi f t], {t, 0, dur}]
```

Each note will have a default duration of 0.25 seconds. Since we have a list of data values, it is a good idea to make `Note` be `Listable`:

```
In[3]:=  SetAttributes[Note, Listable]
```

The range of frequencies in our data is inaudible, so we need to re-scale the data into another frequency range. We can re-scale any list of numbers into the range [0, 1] with this pair of transformations:

```
In[4]:=  data - Min[data]
Out[4]=  {2.85245, 2.30343, 1.1636, 0., 2.14873, 2.04567,
           0.169447, 0.787712, 0.708035, 2.0699}
```

```
In[5]:=  %/Max[%]
Out[5]=  {1., 0.807527, 0.40793, 0., 0.753293, 0.717163,
           0.0594039, 0.276153, 0.24822, 0.725659}
```

Scaling the data linearly into a frequency range from `fmin` to `fmax` is then straightforward:

```
In[6]:=  scale[data_, fmin_:10000, fmax_:20000] :=
           Module[{m = Min[data], M = Max[data], a, b},
             a = (fmax - fmin)/(M - m);
             b = (M fmin - m fmax)/(M - m);
             (a # + b)& /@ data ]
```

I have chosen the default range of frequencies to go from 10 kHz to 20 kHz.

```
In[7]:=  scale[data]
Out[7]=  {20000., 18075.3, 14079.3, 10000., 17532.9, 17171.6,
           10594., 12761.5, 12482.2, 17256.6}
```

Finally, here is the sequence of tones corresponding to the data:

```
In[8]:=  Note[%];
```

You can play the time sequence using **Animate Selected Graphics** and control the speed (the time interval between tones) using the animation controls.

Note also that for large data sets, `ListPlay` provides a better method for listening to the data:

```
In[9]:=  ?ListPlay
         ListPlay[{a1, a2, ...}] plays a sound whose amplitude is
            given by the sequence of levels ai.
```

For example:

```
In[10]:= data = Table[Exp[-t/2000] (Sin[t] + Random[]/10) // N,
                 {t, 5000}];
In[11]:= ListPlay[data];
```

Integer Arguments to Fourier

Q: *Why doesn't Mathematica evaluate* `Fourier` *when it is applied to a list of integers?*

According to Todd Gayley (tgayley@wri.com), the functions `Fourier` and `InverseFourier` were modified in Version 2.2 to be consistent with other functions that don't evaluate numerically when given exact input:

```
In[1]:=  Sin[1]
Out[1]=  Sin[1]
In[2]:=  Fourier[{0, 1, 2}]
Out[2]=  Fourier[{0, 1, 2}]
```

Applying the function `N` to the list transforms the integers into reals, causing `Fourier` to evaluate:

```
In[3]:=  Fourier[N[{0, 1, 2}]]
Out[3]=  {1.73205 + 0. I, -0.866025 - 0.5 I, -0.866025 + 0.5 I}
```