Tricks of the Trade

This is a column of programming tricks and techniques, most of which, we hope, will be contributed by our readers. You are encouraged to submit ideas to this column.

Edited by Paul Abbott

Proofs Without Words

In the book *Proofs Without Words: Exercises in Visual Thinking*, by Roger B. Nelson (Mathematical Association of America, 1993) there are a number of nice examples of *visual proofs*. Because *Mathematica* contains numeric, symbolic, and graphic operations, it is an excellent environment for demonstrating such proofs. Steven Skiena (skiena@cs.sunysb.edu) includes a number of examples of visualizing combinatorical objects in his book *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica* (Addison-Wesley, 1990). The code for his book accompanies *Mathematica* as the standard package `DiscreteMath`Combinatorica``. In a similar vein, Ted Courant showed "Pictures of Geometric Series" in the last issue of the *Journal*.

Here are two examples of visual proofs. We first set the `Graphics` option `AspectRatio` to `Automatic` to obtain graphics with equal x and y scales:

```
In[1]:= SetOptions[Graphics, AspectRatio -> Automatic];
```

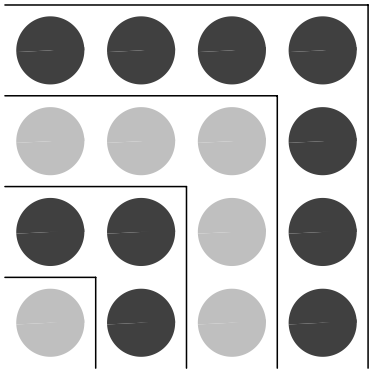
Sums of Odd Integers

The following graphic proof (done here using `Disk` and `Line`) of the formula for the sums of odd integers

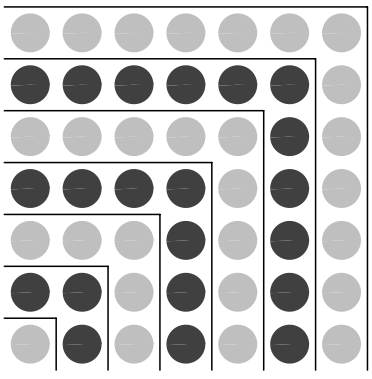
$$1 + 3 + 5 + \ldots + (2n - 1) = n^2$$

is attributed to Nicomachus of Gerasa:

```
In[2]:= lines[n_] := Table[
          {Line[{{0, i}, {i, i}}], Line[{{i, 0}, {i, i}}]},
            {i, n} ]

In[3]:= dots[n_] := Table[{GrayLevel[1/2 - (-1)^Max[i,j]/4],
          Disk[{i - 1/2, j - 1/2}, 3/8]}, {i, n}, {j, n}]

In[4]:= proof[n_] := Show[Graphics[{lines[n], dots[n]}]]

In[5]:= proof[4]
```



```
In[6]:= proof[7]
```



Sums of Squares of Fibonacci Numbers

The Fibonacci numbers:

$$F_1 = 1, F_2 = 1, F_n = F_{n-1} + F_{n-2},$$

can be implemented directly:

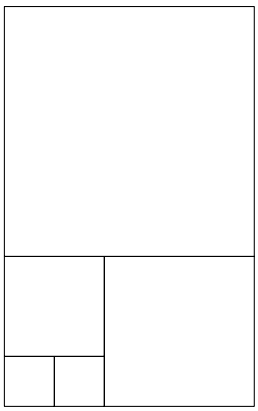
```
In[1]:= fib[1] = 1;
In[2]:= fib[2] = 1;
In[3]:= fib[n_] := fib[n] = fib[n-1] + fib[n-2]
```

Here are the first 15 Fibonacci numbers:

```
In[4]:= Array[fib, 15]
Out[4]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610}
```

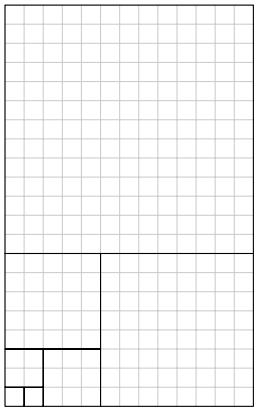
A visual proof, due to Alfred Brousseau, involves constructing a graphic that draws and places boxes with sizes equal to the squares of successive Fibonacci numbers

```
In[5]:= rect[{a_, b_}, {x_, y_}] :=
          {{x, y}, {a+x, y}, {a+x, b+y}, {x, b+y}, {x, y}}

In[6]:= boxes[n_] := Table[Line[rect[{fib[i], fib[i]},
          If[EvenQ[i], {fib[i-1], 0}, {0, fib[i-1]}]]],
            {i, n}]

In[7]:= Show[Graphics[boxes[5]]]
```



It may be helpful to superimpose grid lines onto this picture:

```
In[8]:= grid[n_] :=
          Module[{k, l},
            If[OddQ[n], (k=0; l=1), (k=1; l=0)];
            {Table[Line[{{0, j}, {fib[n+k], j}}],
                {j, 0, fib[n+l]}],
             Table[Line[{{i, 0}, {i, fib[n+l]}}],
                {i, 0, fib[n+k]}]}]

In[9]:= Show[Graphics[{{GrayLevel[0.8], grid[7]}, boxes[7]}]]
```



It is apparent from these graphics that

$$F_1^2 + F_2^2 + \text{L} + F_n^2 = F_n F_{n\ 1}.$$

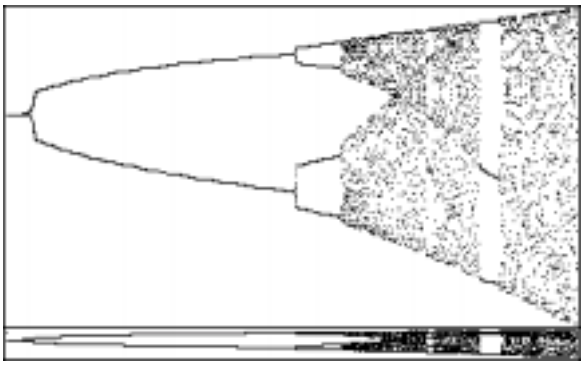
Sound "Visualization"

Every user should be aware that *Mathematica* has excellent graphics capabilities which greatly assist visualization. However, although *Mathematica* includes sound capabilities, perhaps not too many users have taken advantage of them for visualization. Here are a few examples that illustrate the use of `ListPlay`, and show how to use `Compile` to speed up operations that involve lists.

Logistic Map

The logistic map is the mapping x a $\mu x(1-x)$ where $0 \le \mu \le 4$ and $0 \le x \le 1$ (see, for example, Roman Maeder's article "Function Iteration and Chaos" in the *Journal* 5(2): 28–40). An efficient iterative implementation to help visualize the behavior as $\mu = 3 + t/n$ varies from 3 to 4 is

```
In[1]:= logistic[n_Integer, start_:2/3] :=
          Module[ {f, t, x},
            f = Compile[{x, t},
              (3 + t/n) x (1 - x)];
            FoldList[f, N[start], Range[0, n]] ]

In[2]:= ListPlay[logistic[8000], SampleRate -> 2000]
```



Playing this sound one hears the *period-doubling route to chaos*.

It may not be immediately obvious that `f` in this `Module` will be fully compiled. One way to check is to use `With` to provide a suitable (integer) value for the parameter `n`, and then project out the heart of the `InputForm` of the compiled function (which is part `[[1,3]]`):

```
In[3]:= With[{n = 10},
          Module[ {f, t, x},
            f = Compile[{x, t},
              (3 + t/n) x (1 - x)];
            InputForm[f][[1, 3]] ] ]
Out[3]= {{1, 17}, {4, 1, 0}, {4, 2, 1}, {12, 3, 0}, {12, 10, 1},
          {20, 1, 2}, {41, 2, 3}, {36, 1, 3, 4}, {20, 0, 5},
          {33, 5, 4, 6}, {12, 1, 2}, {39, 0, 7}, {20, 2, 8},
          {33, 8, 7, 9}, {36, 6, 0, 9, 10}, {8, 10}}
```

Since this output only contains "op-codes," the function has been fully compiled.

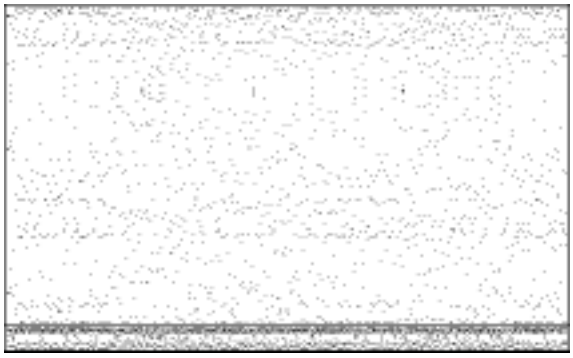
Stabilization of Chaos

In the proceedings of the recent First International *Mathematica* Symposium, A. Iglesias, M.A. Matías, J. Güémez, and J.M. Gutiérrez present a paper entitled "Controlling Chaos with *Mathematica*." Here is a simple implementation of the stabilization of the logistic map using feedback of level λ (that is, x a $x(1+\lambda)$) applied every Δn timesteps:

```
In[4]:= chaos[n_Integer, la_, dn_, mu_, start_:0.3] :=
          Module[ {f},
            f = Compile[{x, {i, _Integer}},
                Module[{y = mu x (1 - x)},
                    If[Mod[i, dn] == 0, y (1 + la), y] ] ];
            FoldList[f[#1, #2]&, start, Range[n]] ]
```

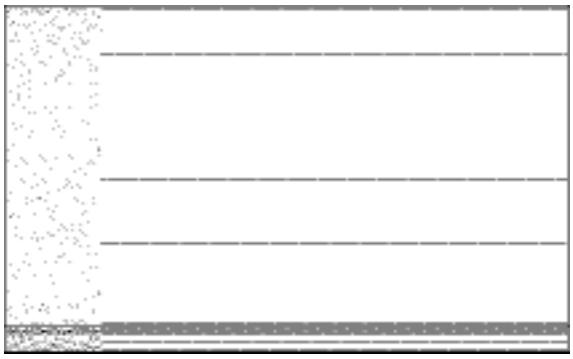
With no feedback ($\lambda = 0.0$):

```
In[5]:= ListPlay[chaos[2000, 0.0, 5, 3.9]]
```



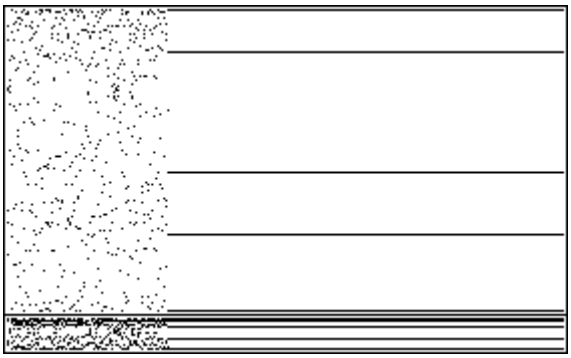
Applying feedback of level $\lambda = -0.05$ every five time steps starting from the default value $x = 0.3$, we see (and hear) that the chaos is quickly stabilized:

```
In[6]:= ListPlay[chaos[2000, -0.05, 5, 3.9]]
```



With a different starting parameter ($x = 0.9$), stabilization takes longer:

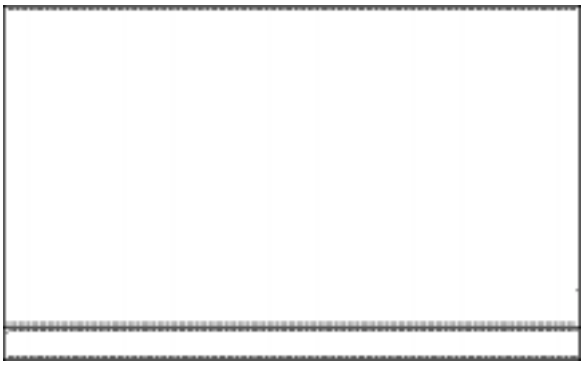
```
In[7]:= ListPlay[chaos[2000, -0.05, 5, 3.9, 0.9]]
```



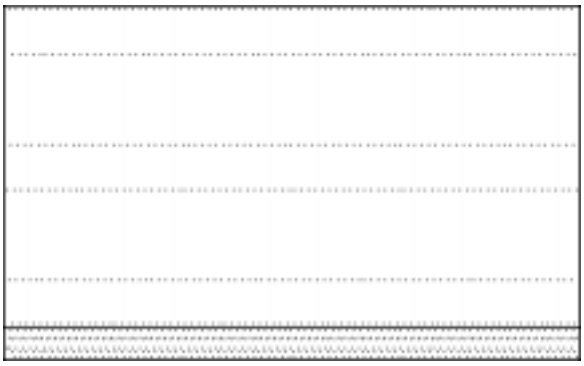
Recurring Decimals

Here are the sounds of a few recurring decimal expansions:

```
In[8]:= ListPlay[First[RealDigits[N[1/11, 500]]]]
```



```
In[9]:= ListPlay[First[RealDigits[N[1/7, 500]]]]
```



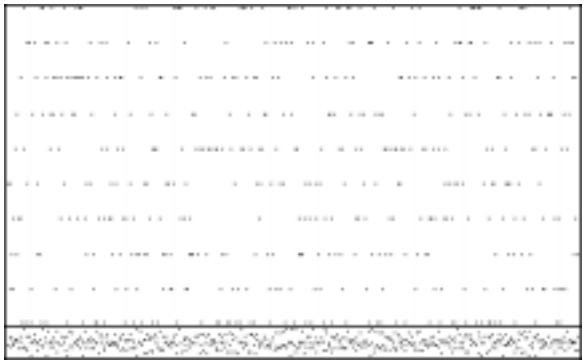
A look at the decimal expansion

```
In[10]:= N[1/7, 18]
Out[10]= 0.142857142857142857
```

shows that the digit sequence repeats every six digits.

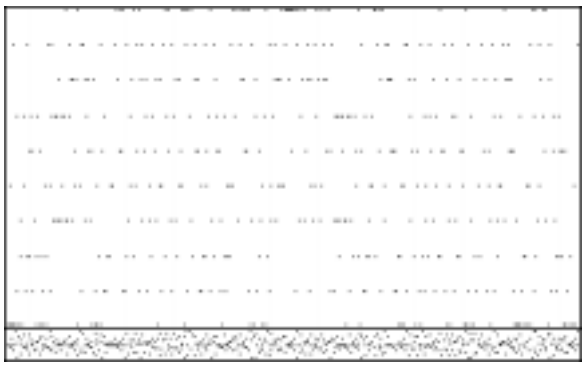
Non-recurrent decimals will just sound like noise,

```
In[11]:= ListPlay[First[RealDigits[N[Sqrt[2], 500]]]]
```



as do rational numbers with digit sequences longer than the sample range:

```
In[12]:= Short[N[1/499, 505], 2]

Out[12]//Short=

0.00200400801603206412825651302605210420841683366733466 9\
   <<423>>53907815631262525050100200400 8

In[13]:= ListPlay[First[RealDigits[%]]]
```



Binary Searching

Geza Makay
makayg@math.u-szeged.hu

The binary search is the fastest way to find an element in an ordered list. The standard package `DiscreteMath`Combinatorica`` contains an implementation,

```
In[1]:= Needs["DiscreteMath`Combinatorica`"]

In[2]:= ?BinarySearch

BinarySearch[l,k,f] searches sorted list l for key k and
   returns the position of l containing k, with f a
   function which extracts the key from an element of l.
```

but it turns out to be too slow for long lists. Looking at the source code of this routine reveals a misprint:

```
BinarySearch[l_List, k_Integer,
       low_Integer, high_Integer, f_] :=
  Module[{mid = Floor[ (low + high)/2 ]},
    If [low > high, Return[low - 1/2]];
    If [f[ l[[mid]] ] == k, Return[mid]];
    If [f[ l[[mid]] ] > k,
      BinarySearch[l, k, 1, mid-1, f],
      BinarySearch[l, k, mid+1, high, f] ] ]
```

The first recursive call to `BinarySearch` should read:

```
BinarySearch[l, k, low, mid-1, f]
```

This error makes the routine run much slower and causes it to use a great deal more memory but, as the routine works, no one would notice much difference when searching short lists. Here are some test results of the corrected routine run on an IBM PC 486/66 with 8 MB of RAM:

No. of list elements	Best times (seconds)
10000	1.373
20000	2.691
50000	9.173
100000	63.933
200000	out of memory
400000	out of memory

Even correcting the misprint does not make the routine fast enough for our purposes. In the following routine, instead of using recursion, which can be quite expensive in execution time, we use a `While` loop.

```
BinarySearch[l_List, k_Integer,
       low_Integer, high_Integer, f_]:=
  Module[{lo, mid, hi, el},
    lo = low;
    hi = high;
    While[lo <= hi,
      If[(el = f[l[[mid = Floor[(lo + hi)/2]]]]) == k,
          Return[mid]];
      If[el > k, hi = mid-1, lo = mid+1] ];
    Return[lo - 1/2] ]
```

Since computing `f` (the function that extracts the key from an element of the list) twice may take a long time, we calculate it once and save the result for later use. The improvement in speed is quite impressive:

No. of list elements	Best times (seconds)
10000	0.124
20000	0.206
50000	0.417
100000	0.928
200000	107.5
400000	out of memory

But it is still not fast enough. Notice that the time to complete a search depends (more or less) linearly on the length of the list although, in theory, it should be logarithmic. The sudden jump of the execution time at 200,000 elements is due to insufficient RAM and the use of virtual memory certainly makes the computation much slower. It is not hard to find out what makes the modified routine slow: When calling the routine, the first argument (a list) is evaluated and copied for the use of the routine. Since we do not want to modify the list in the routine, it is sufficient to pass the list as an unevaluated symbol (using the **HoldFirst** attribute). This gives us the following routine:

```
In[3]:= Unprotect[BinarySearch];

In[4]:= SetAttributes[BinarySearch, HoldFirst]

In[5]:= BinarySearch[l_, k_Integer,
          low_Integer, high_Integer, f_] :=
        Module[{lo = low, mid, hi = high, el, ls = l},
          If[Length[ls] > 0,
            While[lo <= hi,
              If[(el =
                  f[ls[[mid = Floor[(lo + hi)/2]]]]) == k,
                Return[mid] ];
              If[el > k, hi = mid-1, lo = mid+1] ],
            Return[-1] ];
          Return[lo - 1/2] ]
```

This routine is much faster than the previous ones:

No. of list elements	Best times (seconds)
10000	0.065
20000	0.07
50000	0.077
100000	0.081
200000	0.115
400000	0.23

Note that this routine gives run-times depending logarithmically on the length of the list only if the list is passed as a variable in the first argument. Otherwise, the assignment of the local variable **ls=l** evaluates the expression **l**, which gives run-times linear in the length of the list.

To test these routines, we used a randomly generated sorted list of integers,

```
In[6]:= n = 50000;

In[7]:= alist = NestList[# + Random[Integer, 100]&, 0, n];
```

and a random number to search for:

```
In[8]:= rand := Random[Integer, Last[alist]]

In[9]:= BinarySearch[alist, rand, 1, n, Identity] // Timing

Out[9]= {0.25 Second, 15/2}
```

For the final versions of **BinarySearch**, we used a **While** loop 100 times to give more precise timing values, which will (of course) make the times higher than they actually are:

```
In[10]:= i = 0;
         While[i < 100,
           BinarySearch[alist, rand, 1, n, Identity];
           i++] // Timing

Out[10]= {1.76667 Second, Null}
```

In summary: To speed up routines, do not use recursion and keep long lists (such as interpolating functions or arrays of numbers) in unevaluated form.

Steve Skiena, author of the **Combinatorica** *package, responds*: I am certainly chagrined to see the bug/typo on my part which turned **BinarySearch** into a linear-time algorithm (independent of the evaluation issue). The reason it went over five years undetected within **Combinatorica** is because it is not used for any purpose other than exposition, so there is no other function which depends upon it which would reveal its poor run-time behavior.

A more interesting excuse is that **BinarySearch** is a notoriously difficult 'simple' algorithm to implement correctly. Knuth states that "many good programmers have done it wrong" and that that although binary search was first published in 1946, the first completely correct implementation was apparently not published until 1962!

The speedups associated with keeping the list in unevaluated form certainly speak for themselves.

Split

The following utility routine takes a list and a test and splits the list at points failing the test by making successive pairwise comparisons. It makes use of pattern matching and the recursive possibilities provided by rule-based programming. The implementation starts by separating the list into two parts:

```
In[1]:= Split[{a_, b___}, test_:SameQ] :=  Split[{{a}, {b}}, test]
```

The matching condition applies the test to the last element of the first list and the first element of the last list and proceeds recursively:

```
In[2]:= Split[{{a___, b_}, {c_, d___}}, t_] /; t[b, c] :=
          Split[{{a, b, c}, {d}}, t]

In[3]:= Split[{{a___, b_}, {c_, d___}}, t_] :=
          {{a, b}} ~Join~ Split[{c, d}, t]
```

Note that conditional tests can appear on the left-hand side of an expression.

The recursion terminates when the last element is an empty list:

```
In[4]:= Split[{a_, {}}, t_] := {a}
```

Here are two examples. The first uses the default (**SameQ**) test:

```
In[5]:= Split[{a, a, a, a, b, b, c, c, c, c, d}]
Out[5]= {{a, a, a, a}, {b, b}, {c, c, c, c}, {d}}
```

A second application partitions a list of random numbers,

```
In[6]:= Table[Random[], {10}]
Out[6]= {0.515833, 0.107546, 0.114147, 0.682227, 0.304036,
          0.905033, 0.380635, 0.044102, 0.360654, 0.845311}
```

by breaking the list when successive elements differ by more than 0.4:

```
In[7]:= Split[%, Abs[#1 - #2] <= 0.4 &]
Out[7]= {{0.515833}, {0.107546, 0.114147}, {0.682227, 0.304036},
          {0.905033}, {0.380635, 0.044102, 0.360654}, {0.845311}}
```

Catch and Throw

Instead of using **Do**, **For**, or **While** for repetitive computation involving testing, one can use **Nest**, **Fold**, or **FixedPoint** along with **Catch** and **Throw**. Here are four examples.

The first power of 2 that is greater than 10^9:

```
In[1]:= Catch[
            FixedPoint[If[# > 10^9, Throw[#], 2 #]&, 2]]
Out[1]= 1073741824
```

As a check:

```
In[2]:= %/2 < 10^9 < %
Out[2]= True
```

The smallest positive integer n such that $n!$ is greater than 10^8:

```
In[3]:= Catch[Fold[
            If[#1 > 10^8, Throw[#2 - 1], Times[##]]&,
                1, Range[20]]]
Out[3]= 12
```

As a check:

```
In[4]:= 11! < 10^8 < 12!
Out[4]= True
```

The first arithmetic sum $(1 + 2 + 3 + \ldots + n)$ that exceeds 10000:

```
In[5]:= Catch[Fold[
            If[#1 > 10000, Throw[#1], Plus[##]]&,
                0, Range[200]]]
Out[5]= 10011
```

The number of terms n in the harmonic sum $(1 + 1/2 + 1/3 + \ldots + 1/n)$ required to exceed 6.0:

```
In[6]:= Catch[Fold[
            If[#1 > 6.0, Throw[#2 - 1], #1 + 1/#2 ]&,
                0.0, Range[300]]]
Out[6]= 227
```

Alternatively, the sum can be computed using

```
In[7]:= NSum[1/i, {i, ∞}]
Out[7]= 6.00437
```

and the inequality tested using

```
In[8]:= % - 1/%% < 6.0 < %
Out[8]= True
```

Note that when using **Nest** or **Fold** one needs to choose sufficiently large iteration parameters to ensure that a solution is encountered before the iteration terminates.

Vector and Matrix Norms

Mathematica does not include any of the standard vector or matrix norms in the kernel or the standard packages. Perhaps one reason for this is that any specific norm is very easy to construct using built-in list or matrix operations – in fact all the definitions below are "one-liners." Section 16.8.1 of the *Handbook of Applied Mathematics*, edited by Carl E. Pearson (Van Nostrand Reinhold, 1990), lists a number of vector and matrix norms.

The following vector and matrix will be used in the numerical examples:

```
In[1]:= vec = Table[Random[], {5}]
Out[1]= {0.650118, 0.653373, 0.522767, 0.464846, 0.484166}

In[2]:= (mat = Table[Random[], {2}, {2}]) // MatrixForm
Out[2]//MatrixForm=
        0.832695     0.0522864
        0.206315     0.644941
```

Vector norms

The Hölder norm is defined by

$$\left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}, \quad p \geq 1$$

and can be implemented directly as

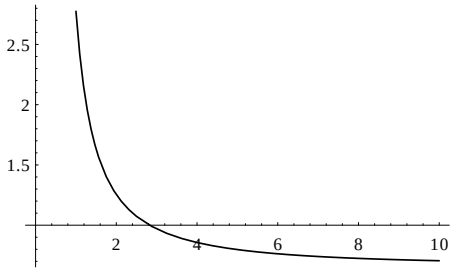
```
In[3]:= HolderNorm[x_?VectorQ, p_ /; p >= 1] :=
            (Plus @@ Flatten[Abs[x]^p])^(1/p)
```

For example,

```
In[4]:= HolderNorm[vec, 3]
Out[4]= 0.969231
```

One can visualize the Hölder norm as a function of p:

```
In[5]:= Plot[HolderNorm[vec, p], {p, 1, 10}]
```



The e-norm is simply the maximum absolute value of the elements of the vector x:

```
In[6]:= eNorm[x_?VectorQ] := Max[Abs[x]]

In[7]:= eNorm[vec]
Out[7]= 0.653373
```

and may be interpreted as a special case of the Hölder norm for $p \to \infty$:

```
In[8]:= HolderNorm[vec, 1000]
Out[8]= 0.65648
```

The e′-norm is defined by

$$\sum_{i=1}^{n} |x_i|$$

and implemented via

```
In[9]:= ePrimeNorm[x_?VectorQ] := Plus @@ Flatten[Abs[x]]

In[10]:= ePrimeNorm[vec]
Out[10]= 2.77527
```

The e′-norm is a special case of the Hölder norm with $p = 1$:

```
In[11]:= HolderNorm[vec, 1]
Out[11]= 2.77527
```

The Euclidean norm is a special case of the Hölder norm with $p = 2$:

$$\sqrt{\sum_{i=1}^{n} |x_i|^2}$$

```
In[12]:= EuclideanNorm[x_?VectorQ] := HolderNorm[x, 2]

In[13]:= EuclideanNorm[vec]
Out[13]= 1.25433
```

Matrix norms

The following definitions apply to square matrices. It is easy to test if a matrix is square using

```
In[14]:= SquareQ[a_?MatrixQ] := Equal @@ Dimensions[a]

In[15]:= SquareQ[mat]
Out[15]= True
```

The maximum norm of an $n \times n$ matrix a is simply the maximum absolute value occuring in the matrix multiplied by n:

```
In[16]:= MaximumNorm[a_?MatrixQ] :=
           Length[a] Max[Abs[a]] /; SquareQ[a]

In[17]:= MaximumNorm[mat]
Out[17]= 1.66539
```

The definition of the Hölder matrix norm is identical to the vector case except for the extra restriction on p:

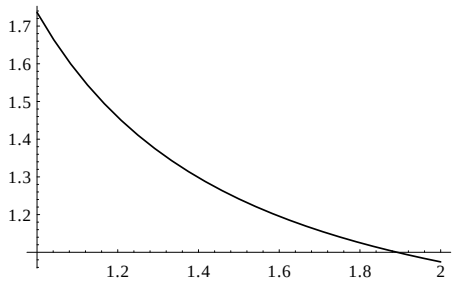
$$\left(\sum_{i,j} |a_{ij}|^p \right)^{1/p}, \quad 1 \le p \le 2$$

The implementation is direct:

```
In[18]:= HolderNorm[a_?MatrixQ, p_ /; 1 <= p <= 2] :=
           (Plus @@ Flatten[Abs[a]^p])^(1/p) /;
             SquareQ[a]

In[19]:= HolderNorm[mat, 1.5]
Out[19]= 1.24159
```

Let's again visualize the Hölder norm as a function of p:

```
In[20]:= Plot[HolderNorm[mat, p], {p, 1, 2}];
```



The Euclidean norm is a special case of the Hölder norm with $p = 2$,

$$\sqrt{\sum_{i,j} |a_{ij}|^2}$$

and we take advantage of this fact in the implementation:

```
In[21]:= EuclideanNorm[a_?MatrixQ] := HolderNorm[a, 2]

In[22]:= EuclideanNorm[mat]
Out[22]= 1.07454
```

There is an alternative equivalent definition of the Euclidean norm:

$$\sqrt{tr(A^{\dagger}A)}$$

where † indicates the conjugate transpose and the trace of a matrix, **tr**, is the sum of its diagonal elements:

```
In[23]:= tr[a_?MatrixQ] := Plus @@ Transpose[a, {1, 1}]

In[24]:= (tr[Conjugate[Transpose[mat]] . mat])^(1/2)
Out[24]= 1.07454
```

The spectrum norm is the largest singular value of the matrix. The singular values of a matrix are defined by

```
In[25]:= Sqrt[Eigenvalues[Conjugate[Transpose[mat]] . mat]]
Out[25]= {0.902607, 0.583035}
```

Alternatively, using the built-in function

```
In[26]:= ?SingularValues

     SingularValues[m] gives the singular value decomposition
       for a numerical matrix m. The result is a list
       {u, w, v}, where w is the list of nonzero singular
       values, and m can be written as
       Conjugate[Transpose[u]].DiagonalMatrix[w].v
```

one can write

```
In[27]:= SpectrumNorm[a_?MatrixQ] :=
           Max[SingularValues[a][[2]]] /; SquareQ[a]

In[28]:= SpectrumNorm[mat]
Out[28]= 0.902607
```

Condition numbers

For a square matrix **A**, the *condition number* $C_{\phi}(\mathbf{A})$ is defined by $\phi(\mathbf{A})\phi(\mathbf{A}^{-1})$, where ϕ is a specified norm. This number gives a bound on the sensitivity of changes in the solution **x** of the equation $\mathbf{Ax} = \mathbf{b}$, with respect to changes in **b**.

From the definition, here is an implementation that uses the Euclidean norm by default:

```
In[29]:= ConditionNumber[a_?MatrixQ, norm_:EuclideanNorm] :=
           norm[a] norm[Inverse[a]] /; SquareQ[a]

In[30]:= ConditionNumber[mat]
Out[30]= 2.19406
```

We can easily work with other norms:

```
In[31]:= ConditionNumber[mat, SpectrumNorm]
Out[31]= 1.54812

In[32]:= ConditionNumber[mat, MaximumNorm]
Out[32]= 5.27034

In[33]:= ConditionNumber[mat, HolderNorm[#, 1]&]
Out[33]= 5.72829
```

The Hilbert matrix is a good example of an ill-conditioned matrix (having a large condition number):

```
In[34]:= Hilbert[n_] := Table[1/(i+j-1), {i, n}, {j, n}]

In[35]:= (mat = Hilbert[3]) // MatrixForm
Out[35]//MatrixForm=
```

$$\begin{matrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{matrix}$$

```
In[36]:= ConditionNumber[N[mat]]
Out[36]= 526.159
```